Lab 3

Mục Lục

PHẦN 1: CÂY NHỊ PHÂN TÌM KIẾM BST

Cơ bản

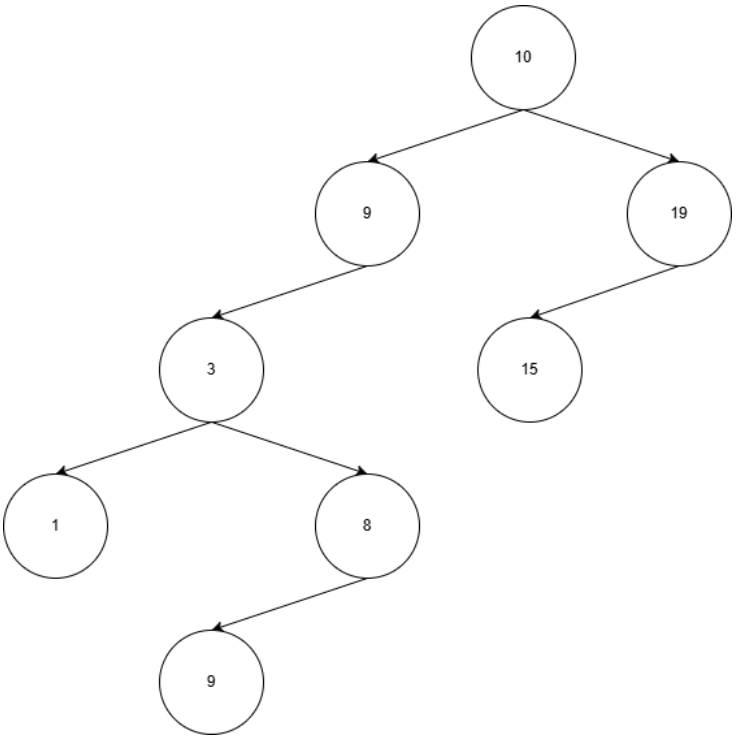
Yêu cầu

1. Biên dịch đoạn chương trình nêu trên.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS

Glasspham on E:/Code/Github/DSA-GTVTHCM
# cd "e:\Code\Github\DSA-GTVTHCM\Lab_3\Mau\" ; if ($?) { g++ *.cpp -o main } ; if ($?) { .\main }
BST.cpp: In member function 'Node* BST::search_x(int)':
BST.cpp:85:1: warning: no return statement in function returning non-void [-Wreturn-type]
   85 | }
      | ^
10
9
3
1
8
4
19
15
Glasspham on E:/Code/Github/DSA-GTVTHCM/Lab_3/Mau
#
```

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



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

```
#include "BST.h"

#include <iostream>
```

```
using namespace std;
BST::BST() { this->root = nullptr; }
BST::~BST() {}
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()) {
            delete n;
            return false;
        }
    }
    if (T->Getkey() > n->Getkey())
        T->Setleft(n);
    else T->Setright(n);
    return true;
}
bool BST::InsertNodeRe(Node *root, Node *p) {
    if (root == nullptr) {
        root = p;
        return true;
```

```
    }
    if (root->Getkey() == p->Getkey()) {
        delete p;
        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() << " ";
        NLR(r->Getleft());
        NLR(r->Getright());
    }
}
void BST::LNR(Node *r) {
    // sinh vien code
    if (r != nullptr) {
        LNR(r->Getleft());
        cout << r->Getkey() << " ";
        LNR(r->Getright());
    }
}
void BST::LRN(Node *r) {
    // sinh vien code
    if (r != nullptr) {
        LRN(r->Getleft());
        LRN(r->Getright());
        cout << r->Getkey() << " ";
```

```
    }
}
void BST::TravelNLR() { NLR(this->root); }
void BST::TravelLNR() {
    // sinh vien code
    LNR(this->root);
}
void BST::TravelLRN() {
    // sinh vien code
    LRN(this->root);
}
Node *BST::search_x(int k) {
    // sinh vien code
    if(!this->root) return nullptr;
    Node* p = this->root;
    while(p != nullptr) {
        if(p->Getkey() == k) return p;
        else if(p->Getkey() > k) p = p->Getleft();
        else p = p->Getright();
    }
    return nullptr;
}
void BST::deleteNode(Node* n) {
    // sinh vien code
    if (!n) return;
    root = deleteNode(root, n->Getkey());
}
Node* BST::deleteNode(Node* root, int x) {
    if(!root) return root;
    if(x < root->Getkey()) root->Setleft(deleteNode(root->Getleft(), x));
```

```
    else if(x > root->Getkey()) root->Setright(deleteNode(root->Getright(), x));
    else {
        if(root->Getleft() == NULL) {
            Node* tmp = root->Getright();
            delete root;
            return tmp;
        } else if(root->Getright() == NULL) {
            Node* tmp = root->Getleft();
            delete root;
            return tmp;
        } else {
            Node* tmp = FindMin(root->Getright());
            root->Setkey(tmp->Getkey());
            root->Setright(deleteNode(root->Getright(), tmp->Getkey()));
        }
    } return root;
}
```

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

```
int BST::SumTree() { return SumTree(this->root); }
int BST::SumTree(Node* root) {
    if(!root) return 0;
    return root->Getkey() + SumTree(root->Getleft()) + SumTree(root->Getright());
}
```

*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.*

```
Node *BST::FindMax() { return FindMax(this->root); }
Node *BST::FindMax(Node* root) {
    if(!root) return nullptr;
    while(root->Getright()) root = root->Getright();
```

```
    return root;
}
Node *BST::FindMin() { return FindMin(this->root); }
Node *BST::FindMin(Node *root) {
    if(!root) return nullptr;
    while(root->Getleft()) root = root->Getleft();
    return root;
}
```

*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**.*

```
int BST::CountNode() { return CountNode(this->root); }
int BST::CountNode(Node *root) {
    if(!root) return 0;
    return 1 + CountNode(root->Getleft()) + CountNode(root->Getright());
}
```

*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**.*

```
int BST::CountLeaf() { return CountLeaf(this->root); }
int BST::CountLeaf(Node *root) {
    if(!root) return 0;
    if(!root->Getleft() && !root->Getright()) return 1;
    return CountLeaf(root->Getleft()) + CountLeaf(root->Getright());
}
```

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

```
int BST::DistinctValues() { return CountNode(); }
```

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

```
// File: 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 Getquantity() { return quantity; }
    void Setquantity(int val) { quantity = val; }
    void IncrementQuantity() { quantity++; }
protected:
private:
    Node *left;
    Node *right;
    Node *parent;
    int key;
    int quantity;
};
#endif // NODE_H
```

```
// File: BST.cpp
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()) {
            p->IncrementQuantity();
            delete n;
            return false;
        }
    }
    if (T->Getkey() > n->Getkey())
        T->Setleft(n);
    else T->Setright(n);
    return true;
}
bool BST::InsertNodeRe(Node *root, Node *p) {
    if (root == nullptr) {
        root = p;
        return true;
    }
    if (root->Getkey() == p->Getkey()) {
        p->IncrementQuantity();
        delete p;
        return false;
    }
```

```
    else if (root->Getkey() > p->Getkey())
        return InsertNodeRe(root->Getleft(), p);
    else return InsertNodeRe(root->Getright(), p);
}
void BST::PrintDistinctValues() { PrintDistinctValues(this->root); }
void BST::PrintDistinctValues(Node *root) {
    if(!root) return;
    PrintDistinctValues(root->Getleft());
    cout << "Value: " << root->Getkey() << ", Frequency: " << root->Getquantity() << endl;
    PrintDistinctValues(root->Getright());
}
```

BÀI TẬP ỨNG DỤNG

Bài 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

```
// File: Node.h
#ifndef NODE_H
#define NODE_H
class Node {
public:
    Node();
    Node(char);
    virtual ~Node();
    Node *Getleft() { return left; }
    void Setleft(Node *val) { left = val; }
    Node *Getright() { return right; }
    void Setright(Node *val) { right = val; }
    char Getkey() { return key; }
    void Setkey(char val) { key = val; }
```

```
    int Getquantity() { return quantity; }
    void Setquantity(int val) { quantity = val; }
    void IncrementQuantity() { quantity++; }
protected:
private:
    Node *left;
    Node *right;
    char key;
    int quantity;
};
#endif // NODE_H
// File: Node.cpp
#include "Node.h"
Node::Node() {
    // ctor
    this->key = this->quantity = 0;
    this->left = nullptr;
    this->right = nullptr;
}
Node::Node(char k) {
    // ctor
    this->key = k;
    this->quantity = 1;
    this->left = nullptr;
    this->right = nullptr;
}
Node::~Node() {
    // dtor
    this->key = this->quantity = 0;
    this->left = nullptr;
```

```
    this->right = nullptr;
}
// File: BST.h
#ifndef BST_H
#define BST_H
#include <string>
#include <iostream>
#include <fstream>
using namespace std;
#include "Node.h"
class BST {
public:
    BST();
    virtual ~BST();
    Node* Getroot() { return root; }
    void Setroot(Node* val) { root = val; }
    void ImportByFile(ifstream&);
    bool InsertNode(Node*);
    bool InsertNodeRe(Node*, Node*);
    void LNR(Node*);
    void TravelLNR();
    Node* search_x(int);
    void PrintDistinctValues();
    void PrintDistinctValues(Node*);
    void Find(char);
protected:
private:
    Node* root;
};
#endif // BST_H
```

```
// File: BST.cpp
#include "BST.h"
BST::BST() { this->root = nullptr; }
BST::~BST() {}
void BST::ImportByFile(ifstream &file) {
    string str;
    while (getline(file, str)) {
        for(char c : str) {
            Node *n = new Node(c);
            InsertNode(n);
        }
    }
    file.close();
}
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()) {
            p->IncrementQuantity();
            delete n;
```

```
            return false;
        }
    }
    if (T->Getkey() > n->Getkey())
        T->Setleft(n);
    else T->Setright(n);
    return true;
}
void BST::LNR(Node *r) {
    if (r != nullptr) {
        LNR(r->Getleft());
        cout << r->Getkey() << " ";
        LNR(r->Getright());
    }
}
void BST::TravelLNR() { LNR(this->root); }
void BST::PrintDistinctValues() { PrintDistinctValues(this->root); }
void BST::PrintDistinctValues(Node *root) {
    if(!root) return;
    PrintDistinctValues(root->Getleft());
    cout << "Value: " << root->Getkey() << ", Frequency: " << root->Getquantity() << endl;
    PrintDistinctValues(root->Getright());
}
void BST::Find(char x) {
    Node *p = this->root;
    while (p != nullptr) {
        if (p->Getkey() == x) {
            cout << "Value: " << p->Getkey() << ", Frequency: " << p->Getquantity() << endl;
            return;
        }
```

```
        else if (p->Getkey() > x)
            p = p->Getleft();
        else p = p->Getright();
    }
    cout << "Value: " << x << ", Frequency: 0" << endl;
}
// File: main.cpp
#include "BST.h"
int main() {
    BST *tree = new BST();
    int select;
    do {
        cout << "1. Import by file " << endl;
        cout << "2. Import by terminal " << endl;
        cout << "Select: ";
        cin >> select;
        if (select != 1 && select != 2)
            cout << "Wrong input" << endl;
    } while(select != 1 && select != 2);
    if (select == 1) {
        string path;
        cout << "Enter path: "; cin >> path;
        cin.ignore();
        ifstream file(path);
        if (!file.is_open()) {
            cout << "Cannot open file: " << endl;
            return 0;
        }
        tree->ImportByFile(file);
    } else {
```

```
        cin.ignore();
        string str;
        cout << "Enter string: " << endl;
        getline(cin, str);
        for(char c : str) {
            Node *n = new Node(c);
            tree->InsertNode(n);
        }
    }
    tree->PrintDistinctValues();
    cout << "Enter character: ";
    char c; cin >> c;
    tree->Find(c);
    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

```
// File: Node.h
#ifndef NODE_H
#define NODE_H
#include <iostream>
#include <string>
using namespace std;
class Node {
public:
    Node();
```

```
    Node(string);
    virtual ~Node();
    Node *Getleft() { return left; }
    void Setleft(Node *val) { left = val; }
    Node *Getright() { return right; }
    void Setright(Node *val) { right = val; }
    string Getkey() { return key; }
    void Setkey(string val) { key = val; }
    int Getquantity() { return quantity; }
    void Setquantity(int val) { quantity = val; }
    void IncrementQuantity() { quantity++; }
protected:
private:
    Node *left;
    Node *right;
    string key;
    int quantity;
};
#endif // NODE_H
// File: Node.cpp
#include "Node.h"
Node::Node() {
    // ctor
    this->key = "";
    this->quantity = 0;
    this->left = nullptr;
    this->right = nullptr;
}
Node::Node(string k) {
    // ctor
```

```
    this->key = k;
    this->quantity = 1;
    this->left = nullptr;
    this->right = nullptr;
}
Node::~Node() {
    // dtor
    this->key = "";
    this->quantity = 0;
    this->left = nullptr;
    this->right = nullptr;
}
// File: BST.h
#ifndef BST_H
#define BST_H
#include <fstream>
#include "Node.h"
class BST {
public:
    BST();
    virtual ~BST();
    Node* Getroot() { return root; }
    void Setroot(Node* val) { root = val; }
    void ImportByFile(ifstream&);
    bool InsertNode(Node*);
    bool InsertNodeRe(Node*, Node*);
    void LNR(Node*);
    void TravelLNR();
    Node* search_x(int);
    void PrintDistinctValues();
```

```
    void PrintDistinctValues(Node*);
    void Find(string);
protected:
private:
    Node* root;
};
#endif // BST_H
// File: BST.cpp
#include "BST.h"
BST::BST() { this->root = nullptr; }
BST::~BST() {}
void BST::ImportByFile(ifstream &file) {
    string str;
    while (file >> str)
        InsertNode(new Node(str));
    file.close();
}
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()) {
            p->IncrementQuantity();
            delete n;
            return false;
        }
    }
    if (T->Getkey() > n->Getkey())
        T->Setleft(n);
    else T->Setright(n);
    return true;
}
void BST::LNR(Node *r) {
    if (r != nullptr) {
        LNR(r->Getleft());
        cout << r->Getkey() << " ";
        LNR(r->Getright());
    }
}
void BST::TravelLNR() { LNR(this->root); }
void BST::PrintDistinctValues() { PrintDistinctValues(this->root); }
void BST::PrintDistinctValues(Node *root) {
    if(!root) return;
    PrintDistinctValues(root->Getleft());
    cout << root->Getkey() << ": " << root->Getquantity() << endl;
    PrintDistinctValues(root->Getright());
}
void BST::Find(string x) {
    Node *p = this->root;
    while (p != nullptr) {
        if (p->Getkey() == x) {
```

```
            cout << p->Getkey() << ": " << p->Getquantity() << endl;
            return;
        }
        else if (p->Getkey() > x)
            p = p->Getleft();
        else p = p->Getright();
    }
    cout << x << ": 0" << endl;
}
// File: main.cpp
#include "BST.h"
int main() {
    BST *tree = new BST();
    int select;
    do {
        cout << "1. Import by file " << endl;
        cout << "2. Import by terminal " << endl;
        cout << "Select: ";
        cin >> select;
        if (select != 1 && select != 2)
            cout << "Wrong input" << endl;
    } while(select != 1 && select != 2);
    if (select == 1) {
        string path;
        cout << "Enter path: "; cin >> path;
        cin.ignore();
        ifstream file(path);
        if (!file.is_open()) {
            cout << "Cannot open file: " << endl;
            return 0;
```

```
        }
        tree->ImportByFile(file);
    } else {
        string str;
        while(cin >> str)
            tree->InsertNode(new Node(str));
    }
    tree->PrintDistinctValues();
    cout << "Enter character: ";
    string c; cin >> c;
    tree->Find(c);
    return 0;
}
```

PHẦN 2: CÂY NHỊ PHÂN CÂN BẰNG AVL

Cơ bản

Yêu cầu

1. Xây dựng cấu trúc cây AVL

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

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

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,.

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

```
// File: Node.h
#pragma once
#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; }
    int Getquantity() { return quantity; }
```

```
    void Setquantity(int val) { quantity = val; }
    void IncrementQuantity() { quantity++; }
protected:
private:
    Node *left;
    Node *right;
    Node *parent;
    // int bal; // -1 0 1
    int key;
    int height;
    int quantity;
};
#endif // NODE_H
// File: Node.cpp
#include "Node.h"
Node::Node() {
    // ctor
    this->key = this->quantity = 0;
    this->left = nullptr;
    this->right = nullptr;
    this->parent = nullptr;
    this->height = 0;
}
Node::Node(int k) {
    // ctor
    this->key = k;
    this->quantity = 1;
    this->left = nullptr;
    this->right = nullptr;
    this->parent = nullptr;
```

```
    this->height = 0;
}
Node::~Node() {
    // dtor
    this->key = this->quantity = 0;
    this->left = nullptr;
    this->right = nullptr;
    this->parent = nullptr;
    this->height = 0;
}
// File 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*);
    Node *deleteNode(Node*, int);
    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();
    int GetHeight(Node*);
    Node *search_x(int);
    int SumTree();
    int SumTree(Node*);
    Node* FindMax();
    Node* FindMax(Node*);
    Node* FindMin();
    Node* FindMin(Node*);
    int CountNode();
    int CountNode(Node*);
    int CountLeaf();
    int CountLeaf(Node*);
    int DistinctValues();
    void PrintDistinctValues();
    void PrintDistinctValues(Node*);
protected:
private:
    Node *root;
    int nNum;   // Node number of tree
    int height; // height of tree
};
#endif // AVL_TREE_H
// File: AVL_Tree.cpp
#include "AVL_tree.h"
```

```
#include <iostream>
#include <algorithm>
#include "Node.h"
using namespace std;
AVL_tree::AVL_tree() {
    // ctor
    this->root = nullptr;
}
AVL_tree::~AVL_tree() {
    // dtor
    this->root = nullptr;
    this->nNum = this->height = 0;
}
bool AVL_tree::InsertNode(Node *n) {
    Node *p = this->root;
    Node *T = nullptr;
    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()) {
            p->IncrementQuantity();
            delete n;
            return false;
```

```
    }
}
if (T->Getkey() > n->Getkey())
    T->Setleft(n);
else T->Setright(n);
n->Setparent(T);
Node *x = n;
Node *parentX = x->Getparent();
while (x != nullptr) {
    int bal = this->CheckBal(x);
    switch (bal) {
    case -1: // Trường hợp mất cân bằng nhẹ bên trái (Left-heavy), không cần xoay
    case 0: // Can bang
    case 1: // Trường hợp mất cân bằng nhẹ bên phải (Right-heavy), không cần xoay
        break;
    case 2:
        if (this->CheckBal(x->Getright()) >= 0) { // LEFT-LEFT
            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
            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; // Thoát vòng lặp sau khi xử lý mất cân bằng LEFT-LEFT hoặc LEFT-RIGHT
case -2:
    if (this->CheckBal(x->Getright()) <= 0) { // RIGHT-RIGHT
        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
        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; // Thoát vòng lặp sau khi xử lý mất cân bằng RIGHT-RIGHT hoặc RIGHT-LEFT
  }
  x = x->Getparent();
 }
 return true;
}
Node *AVL_tree::InsertNode(Node *r, Node *p) {
 if (r == nullptr) return p;
 if (p->Getkey() < r->Getkey())
  r->Setleft(InsertNode(r->Getleft(), p));
 else if (p->Getkey() > r->Getkey())
  r->Setright(InsertNode(r->Getright(), p));
 else {
  r->IncrementQuantity();
  return r;
 }
 r->Setheight(1 + 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;
}
void AVL_tree::InsertNodeRe(Node *p) {
    this->root = InsertNode(this->root, p);
}
Node *AVL_tree::deleteNode(Node *root, int key) {
    if (root == nullptr) return root;
    if (key < root->Getkey())
        root->Setleft(deleteNode(root->Getleft(), key));
    else if (key > root->Getkey())
        root->Setright(deleteNode(root->Getright(), key));
    else {
        if(root->Getleft() == NULL) {
            Node* tmp = root->Getright();
            delete root;
            return tmp;
        } else if(root->Getright() == NULL) {
            Node* tmp = root->Getleft();
            delete root;
            return tmp;
        } else {
            Node* tmp = root->Getright();
            while (tmp->Getleft() != nullptr)
```

```
            tmp = tmp->Getleft();
        root->Setkey(tmp->Getkey());
        root->Setright(deleteNode(root->Getright(), tmp->Getkey()));
      }
   }
   if (root == nullptr) return root;
   root->Setheight(1 + max(GetHeight(root->Getleft()), GetHeight(root->Getright())));
   int balance = CheckBal(root);
   if (balance > 1 && CheckBal(root->Getleft()) >= 0)
      RightRotate(root);
   if (balance > 1 && CheckBal(root->Getleft()) < 0) {
      Node* nLeft = root->Getleft();
      LeftRotate(nLeft);
      root->Setleft(nLeft);
      RightRotate(root);
   }
   if (balance < -1 && CheckBal(root->Getright()) <= 0)
      LeftRotate(root);
   if (balance < -1 && CheckBal(root->Getright()) > 0) {
      Node* nRight = root->Getright();
      RightRotate(nRight);
      root->Setright(nRight);
      LeftRotate(root);
   }
   return root;
}
void AVL_tree::deleteNode(Node *n) {
   this->root = deleteNode(this->root, n->Getkey());
}
void AVL_tree::NLR(Node *r) {
```

```
    if (!r) return;
    cout << r->Getkey() << " ";
    NLR(r->Getleft());
    NLR(r->Getright());
}
void AVL_tree::LNR(Node *r) {
    // sinh vien code
    if (!r) return;
    NLR(r->Getleft());
    cout << r->Getkey() << " ";
    NLR(r->Getright());
}
void AVL_tree::LRN(Node *r) {
    // sinh vien code
    if (!r) return;
    NLR(r->Getleft());
    NLR(r->Getright());
    cout << r->Getkey() << " ";
}
void AVL_tree::TravelNLR() {
    NLR(this->root);
}
void AVL_tree::TravelLNR() {
    // sinh vien code
    LNR(this->root);
}
void AVL_tree::TravelLRN() {
    // sinh vien code
    LRN(this->root);
}
```

```
void AVL_tree::LeftRotate(Node *&P) {
    Node *Q = P->Getright();
    Node *T = Q->Getleft();
    P->Setright(T);
    Q->Setleft(P);
    P->Setparent(Q);
    if (T != nullptr)
        T->Setparent(P);
    P = Q;
}
void AVL_tree::RightRotate(Node *&P) {
    Node *Q = P->Getleft();
    Node *T = Q->Getright();
    P->Setleft(T);
    Q->Setright(P);
    P->Setparent(Q);
    if (T != nullptr)
        T->Setparent(P);
    P = Q;
}
int AVL_tree::CheckBal(Node *p) {
    int bal = this->GetHeight(p->Getleft()) - this->GetHeight(p->Getright());
    return bal;
}
int AVL_tree::GetHeight() { return GetHeight(this->root); }
int AVL_tree::GetHeight(Node *p) {
    if (p == nullptr) return 0;
    else return 1 + max(GetHeight(p->Getleft()), GetHeight(p->Getright()));
}
Node *AVL_tree::search_x(int k) {
```

```
    // sinh vien code
    if (!root) return nullptr;
    Node *p = this->root;
    while (p != nullptr) {
        if (p->Getkey() == k) return p;
        else if (p->Getkey() > k) p = p->Getleft();
        else p = p->Getright();
    }
    return nullptr;
}
int AVL_tree::SumTree() { return SumTree(this->root); }
int AVL_tree::SumTree(Node* root) {
    if(!root) return 0;
    return root->Getkey() + SumTree(root->Getleft()) + SumTree(root->Getright());
}
Node *AVL_tree::FindMax() { return FindMax(this->root); }
Node *AVL_tree::FindMax(Node* root) {
    if(!root) return nullptr;
    while(root->Getright()) root = root->Getright();
    return root;
}
Node *AVL_tree::FindMin() { return FindMin(this->root); }
Node *AVL_tree::FindMin(Node *root) {
    if(!root) return nullptr;
    while(root->Getleft()) root = root->Getleft();
    return root;
}
int AVL_tree::CountNode() { return CountNode(this->root); }
int AVL_tree::CountNode(Node *root) {
    if(!root) return 0;
```

```
    return 1 + CountNode(root->Getleft()) + CountNode(root->Getright());
}
int AVL_tree::CountLeaf() { return CountLeaf(this->root); }
int AVL_tree::CountLeaf(Node *root) {
    if(!root) return 0;
    if(!root->Getleft() && !root->Getright()) return 1;
    return CountLeaf(root->Getleft()) + CountLeaf(root->Getright());
}
int AVL_tree::DistinctValues() { return CountNode(); }
void AVL_tree::PrintDistinctValues() {PrintDistinctValues(this->root); }
void AVL_tree::PrintDistinctValues(Node *root) {
    if(!root) return;
    PrintDistinctValues(root->Getleft());
    cout << "Value: " << root->Getkey() << ", Frequency: " << root->Getquantity() << endl;
    PrintDistinctValues(root->Getright());
}
```

Áp dụng – Nâng cao

Bài 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?

Bài 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

```
// File: Node.h
#pragma once
#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; }
    int Getquantity() { return quantity; }
    void Setquantity(int val) { quantity = val; }
    void IncrementQuantity() { quantity++; }
protected:
private:
    Node *left;
    Node *right;
    Node *parent;
    // int bal; // -1 0 1
    int key;
    int height;
    int quantity;
};
#endif // NODE_H
```

```
// File: Node.cpp
#include "Node.h"
Node::Node() {
    // ctor
    this->key = this->quantity = 0;
    this->left = nullptr;
    this->right = nullptr;
    this->parent = nullptr;
    this->height = 0;
}
Node::Node(int k) {
    // ctor
    this->key = k;
    this->quantity = 1;
    this->left = nullptr;
    this->right = nullptr;
    this->parent = nullptr;
    this->height = 0;
}
Node::~Node() {
    // dtor
    this->key = this->quantity = 0;
    this->left = nullptr;
    this->right = nullptr;
    this->parent = nullptr;
    this->height = 0;
}
// File: 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*);
    Node* deleteNode(Node*, int);
    void deleteNode(Node*);
    void NLR(Node*);
    void TravelNLR();
    Node *search_x(int, int&);
protected:
private:
    Node* root;
};
#endif // BST_H
// File: BST.cpp
#include "BST.h"
#include <iostream>
using namespace std;
BST::BST() { this->root = nullptr; }
BST::~BST() {}
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()) {
            p->IncrementQuantity();
            delete n;
            return false;
        }
    }
    if (T->Getkey() > n->Getkey())
        T->Setleft(n);
    else T->Setright(n);
    return true;
}
void BST::NLR(Node *r) {
    if (r != nullptr) {
        cout << r->Getkey() << " ";
        NLR(r->Getleft());
        NLR(r->Getright());
    }
}
void BST::TravelNLR() { NLR(this->root); }
Node *BST::search_x(int k, int &compareCount) {
    // sinh vien code
    if(!this->root) return nullptr;
    Node* p = this->root;
```

```
    while(p != nullptr) {
        ++compareCount;
        if(p->Getkey() == k) return p;
        else if(p->Getkey() > k) p = p->Getleft();
        else p = p->Getright();
    }
    return nullptr;
}
void BST::deleteNode(Node* x) {
    if (!x) return;
    root = deleteNode(root, x->Getkey());
}
Node* BST::deleteNode(Node* root, int x) {
    if(!root) return root;
    if(x < root->Getkey()) root->Setleft(deleteNode(root->Getleft(), x));
    else if(x > root->Getkey()) root->Setright(deleteNode(root->Getright(), x));
    else {
        if(root->Getleft() == NULL) {
            Node* tmp = root->Getright();
            delete root;
            return tmp;
        } else if(root->Getright() == NULL) {
            Node* tmp = root->Getleft();
            delete root;
            return tmp;
        } else {
            Node* tmp = root->Getright();
            while (tmp->Getleft()) tmp = tmp->Getleft();
            root->Setkey(tmp->Getkey());
            root->Setright(deleteNode(root->Getright(), tmp->Getkey()));
```

```
        }
    } return root;
}
// File: 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; }
    Node *InsertNode(Node*, Node*);
    void InsertNodeRe(Node*);
    Node *deleteNode(Node*, int);
    void deleteNode(Node*);
    void NLR(Node*);
    void TravelNLR();
    void LeftRotate(Node*&);
    void RightRotate(Node*&);
    int CheckBal(Node*);
    int GetHeight(Node*);
    Node *search_x(int, int&);
protected:
private:
    Node *root;
    int nNum;   // Node number of tree
    int height; // height of tree
```

```
};
#endif // AVL_TREE_H
// File: AVL_Tree.cpp
#include "AVL_tree.h"
#include <iostream>
#include <algorithm>
#include "Node.h"
using namespace std;
AVL_tree::AVL_tree() { this->root = nullptr; }
AVL_tree::~AVL_tree() {}
Node *AVL_tree::InsertNode(Node *r, Node *p) {
    if (r == nullptr)
        return p;
    if (p->Getkey() < r->Getkey())
        r->Setleft(InsertNode(r->Getleft(), p));
    else if (p->Getkey() > r->Getkey())
        r->Setright(InsertNode(r->Getright(), p));
    else {
        r->IncrementQuantity();
        return r;
    }
    r->Setheight(1 + 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;
}
void AVL_tree::InsertNodeRe(Node *p) { this->root = InsertNode(this->root, p); }
Node *AVL_tree::deleteNode(Node *root, int key) {
    if (root == nullptr) return root;
    if (key < root->Getkey())
        root->Setleft(deleteNode(root->Getleft(), key));
    else if (key > root->Getkey())
        root->Setright(deleteNode(root->Getright(), key));
    else {
        if(root->Getleft() == NULL) {
            Node* tmp = root->Getright();
            delete root;
            return tmp;
        } else if(root->Getright() == NULL) {
            Node* tmp = root->Getleft();
            delete root;
            return tmp;
        } else {
            Node* tmp = root->Getright();
            while (tmp->Getleft()) tmp = tmp->Getleft();
            root->Setkey(tmp->Getkey());
```

```
            root->Setright(deleteNode(root->Getright(), tmp->Getkey()));
        }
    }
    if (root == nullptr) return root;
    root->Setheight(1 + max(GetHeight(root->Getleft()), GetHeight(root->Getright())));
    int balance = CheckBal(root);
    if (balance > 1 && CheckBal(root->Getleft()) >= 0)
        RightRotate(root);
    if (balance > 1 && CheckBal(root->Getleft()) < 0) {
        Node* nLeft = root->Getleft();
        LeftRotate(nLeft);
        root->Setleft(nLeft);
        RightRotate(root);
    }
    if (balance < -1 && CheckBal(root->Getright()) <= 0)
        LeftRotate(root);
    if (balance < -1 && CheckBal(root->Getright()) > 0) {
        Node* nRight = root->Getright();
        RightRotate(nRight);
        root->Setright(nRight);
        LeftRotate(root);
    }
    return root;
}
void AVL_tree::deleteNode(Node *n) { this->root = deleteNode(this->root, n->Getkey()); }
void AVL_tree::NLR(Node *r) {
    if (!r) return;
    cout << r->Getkey() << " ";
    NLR(r->Getleft());
    NLR(r->Getright());
```

```
}
void AVL_tree::TravelNLR() { NLR(this->root); }
void AVL_tree::LeftRotate(Node *&P) {
    Node *Q = P->Getright();
    Node *T = Q->Getleft();
    P->Setright(T);
    Q->Setleft(P);
    P->Setparent(Q);
    if (T != nullptr)
        T->Setparent(P);
    P = Q;
}
void AVL_tree::RightRotate(Node *&P) {
    Node *Q = P->Getleft();
    Node *T = Q->Getright();
    P->Setleft(T);
    Q->Setright(P);
    P->Setparent(Q);
    if (T != nullptr)
        T->Setparent(P);
    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()));
}
```

```
Node *AVL_tree::search_x(int k, int &compareCount) {
    if(!this->root) return nullptr;
    Node* p = this->root;
    while(p != nullptr) {
        compareCount++;
        if(p->Getkey() == k) return p;
        else if(p->Getkey() > k) p = p->Getleft();
        else p = p->Getright();
    }
    return nullptr;
}
```

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

```
// File: Student.h
#ifndef _STUDENT_H_
#define _STUDENT_H_
#include <string>
using namespace std;
class Student {
private:
    int studentID;
    string name;
    string dateOfBirth;
    float gpa;
public:
    // Constructors
    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) {}

    // Getters and setters
```

```
    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_
// File: Node.h
#ifndef _NODE_H_
#define _NODE_H_
#include "Student.h"
using namespace std;
class Node{
private:
    Student student;
    Node* left;
    Node* right;
    int height;
public:
    // Constructors
    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* n) { left = n; }
    Node* getRight() const { return right; }
    void setRight(Node* n) { right = n; }
    int getHeight() const { return height; }
    void setHeight(int h) { height = h; }
};
#endif // _NODE_H_
// File: AVL_Tree.h
#ifndef _AVL_TREE_H_
#define _AVL_TREE_H_
#include "Node.h"
#include <iostream>
#include <string>
using namespace std;
class AVL_tree {
private:
    Node* root;
    int getHeight(Node*);
    int checkBalance(Node*);
    void rotateLeft(Node*&);
    void rotateRight(Node*&);
    Node* deleteNode(Node*, int);
    Node* insert(Node*, const Student&);
    bool search(Node*, int);
public:
    AVL_tree() : root(nullptr) {}
    void insert(const Student&);
    void deleteNode(int);
    bool search(int);
    void LNR(Node*);
```

```
    void TravelLNR();
    void NLR(Node*);
    void TravelNLR();
};
#endif // _AVL_TREE_H_
// File: AVL_Tree.cpp
#include "AVL_tree.h"
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()) 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);
        root->setLeft(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);
        root->setRight(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);
        node->setLeft(nLeft);
        rotateRight(node);
    }
    if (balance < -1 && student.getStudentID() < node->getRight()->getStudent().getStudentID()) {
        Node* nRight = node->getRight();
        rotateRight(nRight);
        node->setRight(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) return;
    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) return;
    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); }
// File: main.cpp
#include <iostream>
#include "AVL_tree.h"
#include "Student.h"
using namespace std;
int main() {
    AVL_tree tree;
    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();
    cout << "-------------\n";
    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 << "-------------\n";
    int del_id = 103;
    tree.deleteNode(del_id);
    tree.TravelLNR();
    cout << "-------------\n";
    return 0;
}
```

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

Đã cài đặt trong phần **Cơ Bản**, và **Áp dụng – Nâng cao** ở trên.

BÀI TẬP THÊM

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

```
// File: Word.h
#pragma once
#ifndef _WORD_H_
#define _WORD_H_
#include <string>
using namespace std;
class Word {
private:
    string wordEnglish;
    string wordVietnamese;
public:
    Word() : wordEnglish(""), wordVietnamese("") {}
    Word(string e, string v) : wordEnglish(e), wordVietnamese(v) {}
    string getWordEnglish() const { return wordEnglish; }
    string getWordVietnamese() const { return wordVietnamese; }
    void setWordEnglish(const string& e) { wordEnglish = e; }
    void setWordVietnamese(const string& v) { wordVietnamese = v; }
    bool operator==(const Word& other) const { return wordEnglish == other.wordEnglish; }
```

```
    bool operator!=(const Word& other) const { return wordEnglish != other.wordEnglish; }
    bool operator<(const Word& other) const { return wordEnglish < other.wordEnglish; }
    bool operator>(const Word& other) const { return wordEnglish > other.wordEnglish; }
    bool operator<=(const Word& other) const { return wordEnglish <= other.wordEnglish; }
    bool operator>=(const Word& other) const { return wordEnglish >= other.wordEnglish; }
};
#endif // _WORD_H_
// File: Node.h
#pragma once
#ifndef _NODE_H_
#define _NODE_H_
#include "Word.h"
using namespace std;
class Node {
private:
    Word key;
    Node* left;
    Node* right;
    int height;
public:
    Node() : left(nullptr), right(nullptr), height(1) {}
    Node(Word key) : key(key), left(nullptr), right(nullptr), height(1) {}
    ~Node() {};
    Word getKey() { return key; };
    Node* getLeft() { return left; };
    Node* getRight() { return right; };
    int getHeight() { return height; };
    void setKey(Word key) { this->key = key; };
    void setLeft(Node* left) { this->left = left; };
    void setRight(Node* right) { this->right = right; };
```

```
    void setHeight(int height) { this->height = height; };
};
#endif // _NODE_H_
// File: AVL_Tree.h
#pragma once
#ifndef _AVL_TREE_H_
#define _AVL_TREE_H_
#include "Node.h"
#include <iostream>
using namespace std;
class AVL_tree {
private:
    Node* root;
    int getHeight(Node*);
    int checkBalance(Node*);
    void rotateLeft(Node*&);
    void rotateRight(Node*&);
    Node* deleteNode(Node*, string);
    Node* insert(Node*, Word);
    Node* search(Node*, string);
public:
    AVL_tree() : root(nullptr) {}
    void insert(Word);
    void deleteNode(string);
    bool search(string, string&);
    bool modify(string, string);
    void LNR(Node*);
    void TravelLNR();
};
#endif // _AVL_TREE_H_
```

```
// File: AVL_Tree.cpp
#include "AVL_tree.h"
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, string key) {
    if (!root) return root;
```

```
if (key < root->getKey().getWordEnglish())
    root->setLeft(deleteNode(root->getLeft(), key));
else if (key > root->getKey().getWordEnglish())
    root->setRight(deleteNode(root->getRight(), key));
else {
    if (!root->getLeft() || !root->getRight()) {
        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()) temp = temp->getLeft();
        root->setKey(temp->getKey());
        root->setRight(deleteNode(root->getRight(), temp->getKey().getWordEnglish()));
    }
}
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);
    root->setLeft(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);
        root->setRight(nRight);
        rotateLeft(root);
    }
    return root;
}
Node* AVL_tree::insert(Node* node, Word word) {
    if (node == nullptr) return new Node(word);
    if (word < node->getKey()) node->setLeft(insert(node->getLeft(), word));
    else if (word > node->getKey()) node->setRight(insert(node->getRight(), word));
    else return node;
    node->setHeight(1 + max(getHeight(node->getLeft()), getHeight(node->getRight())));
    int balance = checkBalance(node);
    if (balance > 1 && word < node->getLeft()->getKey())
        rotateRight(node);
    if (balance < -1 && word > node->getRight()->getKey())
        rotateLeft(node);
    if (balance > 1 && word > node->getLeft()->getKey()) {
        Node* nLeft = node->getLeft();
        rotateLeft(nLeft);
        node->setLeft(nLeft);
        rotateRight(node);
    }
    if (balance < -1 && word < node->getRight()->getKey()) {
```

```
        Node* nRight = node->getRight();
        rotateRight(nRight);
        node->setRight(nRight);
        rotateLeft(node);
    }
    return node;
}
Node* AVL_tree::search(Node* root, string key) {
    if (!root) return nullptr;
    if (key == root->getKey().getWordEnglish()) return root;
    if (key < root->getKey().getWordEnglish()) return search(root->getLeft(), key);
    return search(root->getRight(), key);
}
void AVL_tree::insert(Word word) { root = insert(root, word); }
void AVL_tree::deleteNode(string key) { root = deleteNode(root, key); }
bool AVL_tree::search(string key, string& res) {
    Node *result = search(root, key);
    if (result) {
        res = result->getKey().getWordVietnamese();
        return true;
    }
    return false;
}
bool AVL_tree::modify(string english, string vietnamese) {
    Node* node = search(root, english);
    if (node != nullptr) {
        Word modifiedWord = node->getKey();
        modifiedWord.setWordVietnamese(vietnamese);
        node->setKey(modifiedWord);
        return true;
```

```
    }
    return false;
}
void AVL_tree::LNR(Node* root) {
    if (!root) return;
    LNR(root->getLeft());
    cout << "English: " << root->getKey().getWordEnglish() << " -> Vietnamese: " << root->getKey().getWordVietnamese() << endl;
    LNR(root->getRight());
}
void AVL_tree::TravelLNR() { LNR(root); }
// File: Dictionary.h
#pragma once
#ifndef DICTIONARY_H
#define DICTIONARY_H
#include "AVL_tree.h"
#include <string>
using namespace std;
class Dictionary {
public:
    Dictionary();
    ~Dictionary();
    void addWord(string, string);
    bool lookupWord(string, string&) ;
    bool modifyWord(string, string);
    void displayAll() ;

private:
    AVL_tree tree;
};
```

```
#endif // DICTIONARY_H
// File: Dictionary.cpp
#include "Dictionary.h"
#include <fstream>
#include <iostream>
Dictionary::Dictionary() {}
Dictionary::~Dictionary() {}
void Dictionary::addWord(string english, string vietnamese) {
    Word word(english, vietnamese);
    tree.insert(word);
}
bool Dictionary::lookupWord(string english, string& vietnamese) {
    return tree.search(english, vietnamese);
}
bool Dictionary::modifyWord(string english, string new_vietnamese) {
    return tree.modify(english, new_vietnamese);
}
void Dictionary::displayAll() { tree.TravelLNR(); }
// File: main.cpp
#include "Dictionary.h"
#include <iostream>
#include <limits>
using namespace std;
void menu() {
    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: ";
}
int main() {
    Dictionary dict;
    int choice;
    string eng, vie;
    while(true) {
        menu();
        cin >> choice;
        if (choice == 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";
        } else if (choice == 2) {
            cout << "Nhap tu Anh can tim: ";
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            getline(cin, eng);
            vie = "";
            dict.lookupWord(eng, vie);
        }
        else if (choice == 3) {
            cout << "Nhap tu Anh can sua: ";
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            getline(cin, eng);
            cout << "Nghia tieng Viet moi: ";
            getline(cin, vie);
```

```
            dict.modifyWord(eng, vie);
            cout << "Sua thanh cong.\n";
        } else if (choice == 4) {
            dict.displayAll();
        } else if (choice == 5) {
            cout << "Goodbye!\n";
            break;
        } else cout << "Lua chon khong hop le. Vui long nhap lai.\n";
    }
    return 0;
}
```