**BÁO CÁO CÀI ĐẶT CÂY AVL**

**1. GIỚI THIỆU**

Cây AVL là một loại cây tìm kiếm nhị phân tự cân bằng, được đặt theo tên của hai nhà khoa học máy tính Soviet Georgy Adelson-Velsky và Evgenii Landis, những người đã phát minh ra nó vào năm 1962. Điểm đặc biệt của cây AVL là mọi nút trong cây có hệ số cân bằng (chiều cao cây con trái trừ chiều cao cây con phải) không vượt quá 1 về giá trị tuyệt đối.

Báo cáo này trình bày chi tiết về việc cài đặt cây AVL sử dụng ngôn ngữ C++, bao gồm các thao tác cơ bản như thêm, xóa, tìm kiếm và các thao tác duyệt cây.

**2. CẤU TRÚC DỮ LIỆU**

**2.1 Cấu trúc Node**

struct Node {

int key; // Giá trị của nút

Node\* left; // Con trỏ tới nút con bên trái

Node\* right; // Con trỏ tới nút con bên phải

int height; // Chiều cao của nút

// Hàm tạo

Node(int k) {

key = k;

left = right = NULL;

height = 1; // Nút lá có chiều cao là 1

}

};

**2.2 Lớp AVLTree**

class AVLTree {

private:

Node\* root;

// Các hàm hỗ trợ

int height(Node\* N);

int getBalance(Node\* N);

void updateHeight(Node\* N);

Node\* rightRotate(Node\* y);

Node\* leftRotate(Node\* x);

Node\* insert(Node\* node, int key);

Node\* minValueNode(Node\* node);

Node\* deleteNode(Node\* root, int key);

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

void preOrder(Node\* root);

void inOrder(Node\* root);

void postOrder(Node\* root);

void freeTree(Node\* root);

public:

// Các phương thức giao diện công khai

AVLTree();

~AVLTree();

void insert(int key);

void remove(int key);

bool search(int key);

void displayPreOrder();

void displayInOrder();

void displayPostOrder();

};

**3. THUẬT TOÁN CHÍNH**

**3.1 Tính chiều cao và hệ số cân bằng**

// Lấy chiều cao của nút

int height(Node\* N) {

if (N == NULL)

return 0;

return N->height;

}

// Lấy giá trị cân bằng của nút

int getBalance(Node\* N) {

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

// Cập nhật chiều cao của nút

void updateHeight(Node\* N) {

if (N != NULL)

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

}

**3.2 Phép xoay**

**3.2.1 Xoay phải**

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

// Thực hiện xoay

x->right = y;

y->left = T2;

// Cập nhật chiều cao

updateHeight(y);

updateHeight(x);

// Trả về nút gốc mới

return x;

}

**3.2.2 Xoay trái**

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

// Thực hiện xoay

y->left = x;

x->right = T2;

// Cập nhật chiều cao

updateHeight(x);

updateHeight(y);

// Trả về nút gốc mới

return y;

}

**3.3 Thêm (Insert)**

Thao tác thêm nút vào cây AVL gồm hai bước chính:

1. Thêm nút như trong cây BST thông thường
2. Cân bằng lại cây nếu cần thiết

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

// 1. Thực hiện thêm bình thường như BST

if (node == NULL)

return new Node(key);

if (key < node->key)

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

else if (key > node->key)

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

else // Không cho phép giá trị trùng lặp

return node;

// 2. Cập nhật chiều cao của nút hiện tại

updateHeight(node);

// 3. Kiểm tra cân bằng và cân bằng lại cây nếu cần

int balance = getBalance(node);

// Trường hợp Left Left

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

return rightRotate(node);

// Trường hợp Right Right

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

return leftRotate(node);

// Trường hợp Left Right

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

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

return rightRotate(node);

}

// Trường hợp Right Left

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

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

return leftRotate(node);

}

// Trả về nút gốc không thay đổi

return node;

}

**3.4 Xóa (Delete)**

Quá trình xóa nút cũng gồm hai bước chính:

1. Xóa nút như trong cây BST thông thường
2. Cân bằng lại cây nếu cần thiết

Node\* deleteNode(Node\* root, int key) {

// 1. Thực hiện xóa bình thường như BST

if (root == NULL)

return root;

if (key < root->key)

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

else if (key > root->key)

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

else {

// Nút có một hoặc không có con

if((root->left == NULL) || (root->right == NULL)) {

Node\* temp = root->left ? root->left : root->right;

// Không có con

if (temp == NULL) {

temp = root;

root = NULL;

}

// Một con

else {

\*root = \*temp; // Sao chép nội dung của con không NULL

}

delete temp;

}

// Nút có hai con

else {

// Tìm nút kế nhiệm (nhỏ nhất ở cây con phải)

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

// Sao chép dữ liệu của nút kế nhiệm vào nút hiện tại

root->key = temp->key;

// Xóa nút kế nhiệm

root->right = deleteNode(root->right, temp->key);

}

}

// Nếu cây chỉ có một nút thì trả về

if (root == NULL)

return root;

// 2. Cập nhật chiều cao

updateHeight(root);

// 3. Kiểm tra cân bằng và cân bằng lại cây nếu cần

int balance = getBalance(root);

// Các trường hợp cân bằng:

// Trường hợp Left Left

if (balance > 1 && getBalance(root->left) >= 0)

return rightRotate(root);

// Trường hợp Left Right

if (balance > 1 && getBalance(root->left) < 0) {

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

return rightRotate(root);

}

// Trường hợp Right Right

if (balance < -1 && getBalance(root->right) <= 0)

return leftRotate(root);

// Trường hợp Right Left

if (balance < -1 && getBalance(root->right) > 0) {

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

return leftRotate(root);

}

return root;

}

**4. PHÂN TÍCH ĐỘ PHỨC TẠP**

**4.1 Thêm (Insert)**

* Tìm vị trí chèn: O(log n)
* Thực hiện cân bằng: O(1) (số lượng xoay hằng số)
* Cập nhật chiều cao: O(log n)
* Tổng: O(log n)

**4.2 Xóa (Delete)**

* Tìm nút cần xóa: O(log n)
* Xóa nút: O(log n)
* Thực hiện cân bằng: O(1) (số lượng xoay hằng số)
* Cập nhật chiều cao: O(log n)
* Tổng: O(log n)

**4.3 Tìm kiếm (Search)**

* Độ phức tạp: O(log n)

**4.4 Duyệt cây (Traversal)**

* Độ phức tạp: O(n) với n là số nút trong cây

**5. ĐÁNH GIÁ VÀ KẾT LUẬN**

Cây AVL là một cấu trúc dữ liệu hiệu quả cho các thao tác tìm kiếm, chèn và xóa với độ phức tạp O(log n). So với cây BST thông thường, cây AVL đảm bảo cân bằng, do đó tránh được trường hợp suy biến thành danh sách liên kết với độ phức tạp O(n).

Tuy nhiên, chi phí cho việc duy trì cân bằng của cây AVL bao gồm:

* Lưu trữ thêm thông tin chiều cao tại mỗi nút
* Thực hiện các phép xoay trong quá trình chèn và xóa

Trong nhiều ứng dụng thực tế, cây AVL là lựa chọn tốt khi cần một cấu trúc dữ liệu tìm kiếm có hiệu suất ổn định và có thể dự đoán trước.