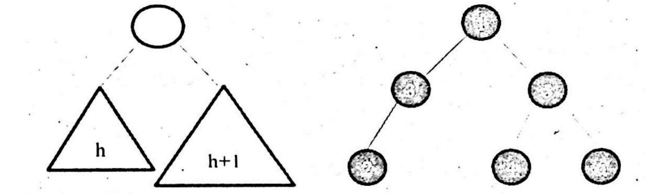
# I. Cây AVL

Với CNPTK khi thêm phần tử có thể sẽ dẫn đến tình trạng cây bị suy biến về danh sách liên kết. Để giải quyết tình trạng này nhiều chuyên gia đã đề xuất nhiều hướng giải quyết khác nhằm làm sao cây cân bằng (chiều cao của cây với N là số phần tử của cây) như: cây 2-3-4, cây đỏ đen, .... Trong số đó cây AVL được Adelson-Velskii và Landis trình bày đầu tiên năm 1962 ở bài báo “An algorithm for the organization of information”. Có hai loại cân bằng:

* **Cân bằng tương đối:** Cây cân bằng tương đối là cây nhị phân mà tại mỗi node trên cây chiều cao cây con trái và chiều cao cây con phải hơn kém nhau không quá một.
* **Cân bằng hoàn toàn:** Cây cân bằng hoàn toàn là một cây nhị phân mà tại mỗi node trên cây tổng số node ở cây con bên trái và tổng số node ở cây con bên phải hơn kém nhau không quá một. Từ định nghĩa ta thấy rằng một cây cân bằng hoàn toàn thì chắc chắn là cây cân bằng tương đối. Một cây nhị phân đầy đủ chắc chắn là một cây cân bằng hoàn toàn.

1. **Định nghĩa**

Cây AVL là cây nhị phân tìm kiếm cân bằng (tương đối) tại mỗi node của cây có độ cao của cây con trái và độ cao của cây con phải chênh lệch không quá một, hình 4.11. Cây AVL có chiều cao là log N (với N là số phần tử của cây).

1. **Cấu trúc Cây AVL**

Cấu trúc của một cây AVL bao gồm các node, mỗi node chứa:

* **key**: Giá trị của node.
* **left** và **right**: Con trỏ (hoặc tham chiếu) đến node con trái và phải.
* **height**: Chiều cao của node. Chiều cao của mỗi node là số lượng cạnh dọc theo đường đi dài nhất từ node đó đến lá cây.

Đặc biệt, chiều cao của cây AVL được duy trì theo công thức:

* **Chiều cao của một node** = 1 + max(Chiều cao cây con trái, Chiều cao cây con phải).
* **Cây AVL** luôn có chiều cao **log N**, với NNN là số phần tử trong cây, do đó đảm bảo hiệu quả cho các thao tác.

1. **Các Thao tác Cơ bản trên Cây AVL**

3.1. **Chèn Phần Tử vào Cây AVL**

Khi chèn một phần tử vào cây AVL, ta thực hiện các bước sau:

1. **Chèn như trong Cây Nhị Phân Tìm Kiếm (BST)**: Thêm phần tử vào cây theo quy tắc của cây BST.
2. **Cập nhật chiều cao của mỗi node**: Sau khi chèn, ta cập nhật lại chiều cao của các node, vì chiều cao của cây có thể thay đổi.
3. **Kiểm tra và tái cân bằng cây**: Kiểm tra chênh lệch chiều cao của cây con trái và cây con phải của mỗi node. Nếu chênh lệch này lớn hơn 1, cây sẽ mất cân bằng và cần phải thực hiện các phép quay để tái cân bằng.

Các phép quay bao gồm:

* **Quay trái (Left Rotation)**: Thực hiện khi cây mất cân bằng tại cây con phải.
* **Quay phải (Right Rotation)**: Thực hiện khi cây mất cân bằng tại cây con trái.
* **Quay trái-phải (Left-Right Rotation)** và **Quay phải-trái (Right-Left Rotation)**: Thực hiện khi cây mất cân bằng ở vị trí phức tạp hơn, yêu cầu kết hợp cả quay trái và phải.

3.2. **Xóa Phần Tử khỏi Cây AVL**

Xóa phần tử khỏi cây AVL được thực hiện theo các bước:

1. **Xóa như trong Cây Nhị Phân Tìm Kiếm**: Tìm và xóa phần tử theo quy tắc của BST.
2. **Cập nhật chiều cao**: Sau khi xóa phần tử, chiều cao của các node cần được cập nhật.
3. **Kiểm tra và tái cân bằng cây**: Sau khi xóa, ta kiểm tra lại chênh lệch chiều cao của cây con trái và phải, nếu cần, thực hiện các phép quay để tái cân bằng.

3.3. **Tìm Kiếm Phần Tử trong Cây AVL**

Tìm kiếm trong cây AVL tương tự như tìm kiếm trong cây BST. Tuy nhiên, do cây AVL luôn được duy trì cân bằng, các thao tác tìm kiếm luôn có độ phức tạp là O(log⁡N)O(\log N)O(logN), giúp tăng hiệu quả tìm kiếm.

1. **Ưu điểm và Nhược điểm của Cây AVL**

**Ưu điểm**:

* **Hiệu suất cao**: Các thao tác tìm kiếm, chèn và xóa có độ phức tạp luôn là O(log⁡N)O(\log N)O(logN), do đó cây AVL rất hiệu quả khi xử lý các bộ dữ liệu lớn.
* **Cân bằng tự động**: Sau mỗi thao tác chèn hoặc xóa, cây AVL tự động duy trì tính cân bằng mà không cần sự can thiệp của người dùng.

**Nhược điểm**:

* **Chi phí tính toán cao hơn cây BST**: Mặc dù cây AVL mang lại hiệu suất tối ưu, nhưng việc duy trì cân bằng sau mỗi thao tác thay đổi cấu trúc cây yêu cầu tính toán thêm, như việc cập nhật chiều cao của node và thực hiện các phép quay, điều này làm tăng độ phức tạp trong các thao tác.
* **Phức tạp hơn trong việc triển khai**: Cây AVL đòi hỏi phải thực hiện nhiều phép quay và kiểm tra cân bằng sau mỗi thao tác, điều này khiến cho việc triển khai trở nên phức tạp hơn so với cây BST thông thường.

1. **Mã nguồn C++ cho Cây AVL**

Dưới đây là mã nguồn C++ để triển khai các thao tác cơ bản trên cây AVL chứa các số nguyên, bao gồm các thao tác chèn, tìm kiếm và xóa.

#include <iostream>

#include <algorithm>

using namespace std;

class Node {

public:

    int key;

    Node\* left;

    Node\* right;

    int height;

    Node(int value) {

        key = value;

        left = right = nullptr;

        height = 1;

    }

};

class AVLTree {

private:

    Node\* root;

    int height(Node\* node) {

        if (node == nullptr) return 0;

        return node->height;

    }

    int getBalance(Node\* node) {

        if (node == nullptr) return 0;

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

    }

    Node\* leftRotate(Node\* x) {

        Node\* y = x->right;

        Node\* T2 = y->left;

        y->left = x;

        x->right = T2;

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

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

        return y;

    }

    Node\* rightRotate(Node\* y) {

        Node\* x = y->left;

        Node\* T2 = x->right;

        x->right = y;

        y->left = T2;

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

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

        return x;

    }

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

        if (node == nullptr)

            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

            return node;

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

        int balance = getBalance(node);

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

            return rightRotate(node);

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

            return leftRotate(node);

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

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

            return rightRotate(node);

        }

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

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

            return leftRotate(node);

        }

        return node;

    }

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

        if (node == nullptr || node->key == key)

            return node;

        if (key < node->key)

            return search(node->left, key);

        return search(node->right, key);

    }

    Node\* minNode(Node\* node) {

        Node\* current = node;

        while (current && current->left != nullptr)

            current = current->left;

        return current;

    }

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

        if (root == nullptr)

            return root;

        if (key < root->key)

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

        else if (key > root->key)

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

        else {

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

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

                if (temp == nullptr) {

                    temp = root;

                    root = nullptr;

                } else {

                    \*root = \*temp;

                }

                delete temp;

            } else {

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

                root->key = temp->key;

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

            }

        }

        if (root == nullptr)

            return root;

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

        int balance = getBalance(root);

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

            return rightRotate(root);

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

            return leftRotate(root);

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

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

            return rightRotate(root);

        }

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

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

            return leftRotate(root);

        }

        return root;

    }

    void inorder(Node\* root) {

        if (root != nullptr) {

            inorder(root->left);

            cout << root->key << " ";

            inorder(root->right);

        }

    }

public:

    AVLTree() {

        root = nullptr;

    }

    void insert(int key) {

        root = insert(root, key);

    }

    bool search(int key) {

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

        return node != nullptr;

    }

    void deleteNode(int key) {

        root = deleteNode(root, key);

    }

    void inorder() {

        inorder(root);

        cout << endl;

    }

};

int main() {

    AVLTree tree;

    tree.insert(10);

    tree.insert(20);

    tree.insert(30);

    tree.insert(25);

    tree.insert(5);

    cout << "In-order traversal of AVL tree: ";

    tree.inorder();

    cout << "Searching for 25: " << (tree.search(25) ? "Found" : "Not Found") << endl;

    tree.deleteNode(20);

    cout << "In-order traversal after deletion: ";

    tree.inorder();

    return 0;

}