Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа
Дисциплина: Программирование на языках высокого уровня
Тема: АВЛ-дерево (удаление узла)

Студент группы 3331506/80401 В.С. Редров
Преподаватель Е.М. Кузнецова
«__» _____ 2021 г.

Санкт-Петербург
2021 г

Оглавление

Введение

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Дерево АВЛ названо в честь двух его советских разработчиков, Георгия Адельсона-Вельского и Евгения Ландиса, которые опубликовали его в своей статье 1962 года «Алгоритм организации информации».

В дереве АВЛ высота двух дочерних поддеревьев любого узла отличается не более чем на единицу; если в любой момент они отличаются более чем на единицу, выполняется перебалансировка для восстановления этого свойства. Поиск, вставка и удаление занимают время O (log n) как в среднем, так и в худшем случаях, где n - количество узлов в дереве до операции. Вставки и удаления могут потребовать перебалансировки дерева путем одного или нескольких вращений дерева.

Деревья АВЛ часто сравнивают с красно-черными деревьями, потому что оба поддерживают один и тот же набор операций и занимают O(log n) в для основных операций. Для приложений с интенсивным поиском деревья АВЛ быстрее, чем красно-черные деревья, потому что они более строго сбалансированы. Подобно красно-черным деревьям, деревья АВЛ сбалансированы по высоте.

В рамках данной курсовой работы будет рассмотрена реализация АВЛ дерева и алгоритм удаления узла на языке программирования *C++* с использованием методов объектно-ориентированного программирования.

Принцип работы

Алгоритм был реализован при помощи языка программирования C++. Узел дерева представлен классом Node, полями которой являются значение ключа в узле, полезные данные узла, высота дерева, указатель на объект Node для левой и правой ветви.

Отсутствие узлов слева или справа будем обнаруживать при помощи нулевого указателя в поле left и right соответственно. Для правильной работы программы необходимо реализовать следующую функцию:

Функция bfactor возвращает разницу между высотой правой и левой ветви. По свойству АВЛ дерева он может принимать значения -1, 0, 1. При добавлении и удалении узлов может возникать ситуация, когда это условие нарушится. Для этого в программе предусмотрена функция балансировки дерева.

В качестве вспомогательных функций также выступают функция height, которая возвращает значение высоты поддерева, и update_height, которая обновляет значение высоты поддерева.

Балансировка узлов может быть осуществлена с помощью двух типов поворота дерева: большого и малого.

Малый левый поворот показан на рисунке 1. Применяется, когда bfactor узла «a» равен 2 и bfactor узла «b» больше либо равен нулю. Ее суть заключается в том, что корневым узлом становится узел «b», его левый потомок, становится правым потомком узла «a», а левым потомком узла «b» становится узел «a».

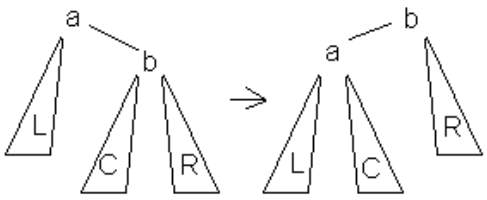

Рисунок 1 – Малый левый поворот

Большой левый поворот показан на рисунке 2. Применяется, когда bfactor узла «a» равен 2 и bfactor узла «b» меньше либо равен 0. Суть этого поворота заключается в том, что корневым узлом становится узел «c», его левый потомок становится правым потомком узла «a», и правый потомок становится левым потомком узла «b». Сами узлы «a» и «b» становятся левым и правым потомками узла «c» соответственно.

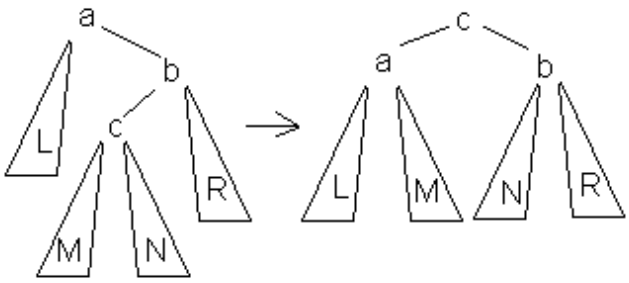

Рисунок 2 – Большой левый поворот

Малый правый и большой правый повороты представлены на рисунках 3 и 4 и являются зеркальным отражением левых поворотов.

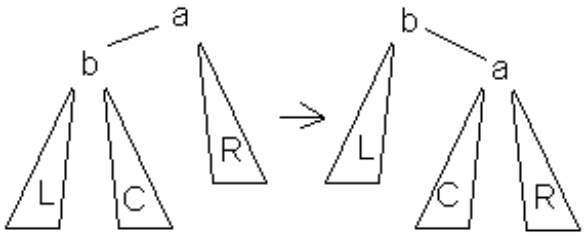

Рисунок 3 – Малый правый поворот

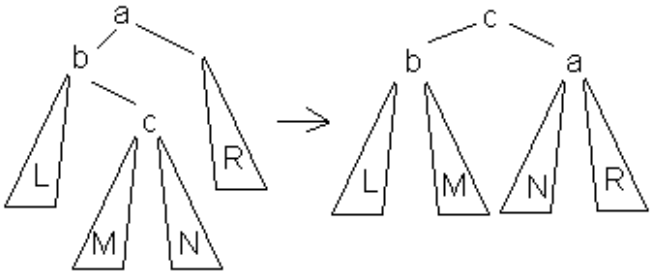

Рисунок 4 – Большой правый поворот

Функция балансировки основана на проверке всех этих условий и возвращает тот же узел, который был подан на вход, но со сбалансированными ветвями.

Функция удаления узла основана на рекурсивном алгоритме, который сначала идет вглубь дерева и ищет необходимый узел, затем, если он лист, то удаляет его и вызывает балансировку для каждого родителя, поднимаясь по рекурсии. Если узел – не лист, то функция находит в поддереве наибольшей длины ближайший по значению ключа элемент и заменяет удаляемый элемент на заменяющий, вызвав функцию удаления для заменяющего элемента. После удаления вызывается функция балансировки для каждого родителя, на каждом этапе возврата из рекурсии. Функция возвращает логический тип данных: false – в случае, если элемент с указанным ключом не найден, true – в случае удачного удаления.

Оценка скорости и памяти

Самая затратная операция в удалении – поиск узлов. В представленном алгоритме в худшем случае осуществляется 3 поиска – удаляемого узла, заменяющего узла и узла-родителя для удаляемого элемента. Эта операция занимает в худшем случае O(log(N)) операций. Также сама функция удаления вызывается максимум два раза (для изначального узла и для заменяющего). Все это дает оценку сложности порядка O(log(N)).

График скорости работы алгоритма уделения узла от количества узлов изображен на рисунке 5

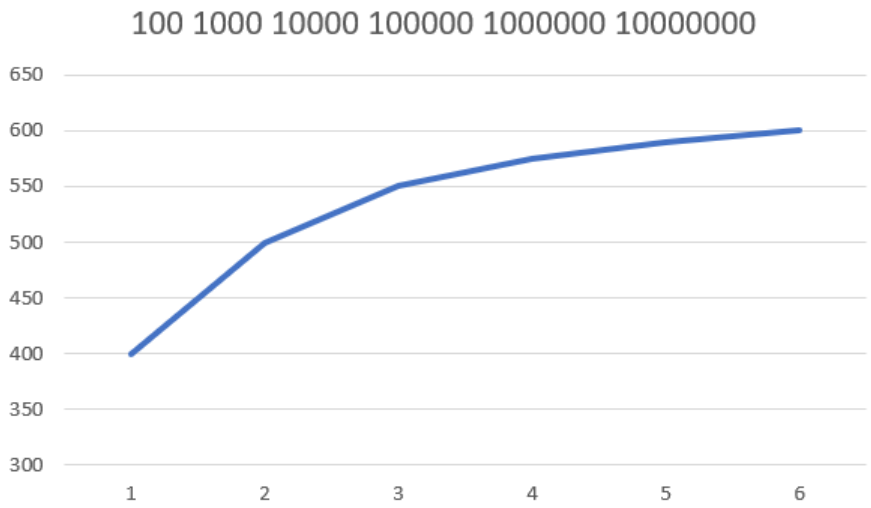

Рисунок 5 – График скорости работы алгоритма от количества узлов

Расход памяти O(N).

Применение алгоритма

АВЛ-деревья могут быть применены для упорядоченного хранения элементов, вставки, поиска и удаления за время порядка O(log(N)), что требуется, например, для баз данных.

Список литературы

1. Адельсон-Вельский Г. М., Ландис Е. М. Один алгоритм организации информации: Доклады АН СССР, 1962, с. 263—266.

2. Thomas H. Cormen Introduction to algorithms: учебное пособие / Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Massachusetts Institute of Technology, 2009, с. 266 – 338.

3. АВЛ – дерево // [Wikipedia]. URL: https://ru.wikipedia.org/wiki/АВЛ-дерево

Приложение – Листинг кода дерева

```
#include <iostream>
#include <string>
#include <stdlib.h>
#include <chrono>

using std::chrono::steady_clock;
using std::chrono::duration_cast;
using std::chrono::microseconds;


std::string NO_KEY = "no_key";

class Node {
public:
    int key;
    unsigned short height;
    std::string data;
public:
    Node *left;
    Node *right;
public:
    Node(const int key, const std::string data, unsigned short height);
};

Node::Node(const int key, const std::string data, unsigned short height){
    this->key = key;
    this->data = data;
    this->height = height;
    this->left = nullptr;
    this->right = nullptr;
}

class Tree { // avl tree
private:
    Node* small_turn_left(Node* cur_root);
    Node* small_turn_right(Node* cur_root);
    Node* big_turn_left(Node* cur_root);
    Node* big_turn_right(Node* cur_root);
    Node* balance(Node* cur_root);
    void update_height(Node* root);     // recalculates height value of the subtr
ee
    unsigned short height(Node * root); // get current height of subtree
    int8_t bfactor(Node * root);        // calculates bfactor of subtree as diffe
rence between left and right subtree heights
                                        // if value is positive, right subtree is
 deeper
    Node* find_node(Node* root, const int key);
```

```
    Node* find_replace_node(Node* root);
    Node* find_parent(Node* root, const int key);
public:
    Node* root;
    bool del(Node* root, const int key);
    Node* add(Node* root, const int key, const std::string data);
public:
    Tree(){ root = nullptr; };
    ~Tree(){ while(root != nullptr){ del(root, root->key); } }
};


int8_t Tree::bfactor(Node * root) {
    int8_t hright = root -> right ? height(root -> right) : 0;
    int8_t hleft = root -> left ? height(root -> left) : 0;
    return hright - hleft;
}

unsigned short Tree::height(Node * root) {
    return root ? root -> height : 0;
}

void Tree::update_height(Node* root){
    unsigned short hleft = height(root -> left);
    unsigned short hright = height(root -> right);
    root -> height = (hleft > hright ? hleft : hright) + 1;
}

Node* Tree::small_turn_left(Node* cur_root){
    Node* new_root = cur_root -> right;
    cur_root -> right = new_root -> left;
    new_root -> left = cur_root;
    update_height(cur_root);
    update_height(new_root);
    return new_root;
}

Node* Tree::small_turn_right(Node* cur_root){
    Node* new_root = cur_root -> left;
    cur_root -> left = new_root -> right;
    new_root -> right = cur_root;
    update_height(cur_root);
    update_height(new_root);
    return new_root;
}

Node* Tree::big_turn_left(Node* cur_root){
    Node* right_subtree = cur_root -> right;
    Node* new_root = right_subtree -> left;
    right_subtree -> left = new_root -> right;
    cur_root -> right = new_root -> left;
```

```
    new_root -> left = cur_root;
    new_root -> right = right_subtree;
    update_height(right_subtree);
    update_height(cur_root);
    update_height(new_root);
    return new_root;
}

Node* Tree::big_turn_right(Node* cur_root){
    Node* left_subtree = cur_root -> left;
    Node* new_root = left_subtree -> right;
    left_subtree -> right = new_root -> left;
    cur_root -> left = new_root -> right;
    new_root -> left = left_subtree;
    new_root -> right = cur_root;
    update_height(left_subtree);
    update_height(cur_root);
    update_height(new_root);
    return new_root;
}

Node* Tree::balance(Node* root){
    update_height(root);
    if (bfactor(root) == 2){
        if (bfactor(root -> right) > 0){
            return small_turn_left(root);
        }
        else{
            return big_turn_left(root);
        }
    }
    if (bfactor(root) == -2){
        if (bfactor(root -> left) < 0){
            return small_turn_right(root);
        }
        else{
            return big_turn_right(root);
        }
    }
    return root;
}

Node* Tree::add(Node* root, const int key, const std::string data){
    if (!root) {
        Node * root = new Node(key, data, 1);
        return root;
    }
    if (key < root -> key)
        root -> left = add(root -> left, key, data);
    else
        root -> right = add(root -> right, key, data);
```

```
    return balance(root);
}

Node* Tree::find_replace_node(Node* root){
    if (bfactor(root) > 0){
        Node* replace_node = root -> right;
        while (replace_node -> left){
            replace_node = replace_node -> left;
        }
        return replace_node;
    }

    Node* replace_node = root -> left;
    while (replace_node -> right){
        replace_node = replace_node -> right;
    }
    return replace_node;


}

Node* Tree::find_parent(Node* root, const int key){
    if (root -> key == key){
        return root;
    }
    if (((root -> left) && (root -> left -> key == key)) || ((root -
> right) && (root -> right -> key == key))){
        return root;
    }
    if (key > root -> key){
        return find_parent(root -> right, key);
    }
    return find_parent(root -> left, key);
}

bool Tree::del(Node* root, const int key){
    bool status = false;
    if (root == nullptr){
        return false;
    }
    if (root -> key == key){
        // if node to delete is found
        Node* parent = find_parent(this->root, key);
        if (!(root -> left) && !(root -> right)){
            // if it has no children (is leaf)
            if (root != this->root){
                // if it is not root of current tree
                if (parent -> left == root)
                    parent -> left = nullptr;
                else
                    parent -> right = nullptr;
```

```
            }
            else
                root = nullptr;
        }
        else{
            // if it has at least one child we sholud find node, which is going t
o replace node to delete
            // then we should delete reaplace node and finale replace node to del
ete with replace node
            Node* replace_node = find_replace_node(root);
            if (!del(root, replace_node -> key)){
                return false;
            }

            replace_node -> left = root -> left;
            replace_node -> right = root -> right;
            if (parent != root){
                if (parent -> left == root)
                    parent -> left = replace_node;
                else
                    parent -> right = replace_node;
            }
            else{
                root = replace_node;
            }
        }
    }

    else if (key > root -> key){
        status = del(root -> right, key);
    }
    else {
        status = del(root -> left, key);
    }
    if (!status){ return false; }
    balance(root);
    return true; // balance tree, while going up the recursion
}


int main()
{
    Tree tree1;

    tree1.root = tree1.add(tree1.root, rand(), "HelloWorld!");
    int key;

    for (size_t i = 0; i < 1000000; i++)
    {
        key = rand();
        tree1.add(tree1.root, key, "str");
```

```
    }


    auto t1 = steady_clock::now();
    std::cout << tree1.del(tree1.root, key);
    auto t2 = steady_clock::now();

    auto delta = duration_cast<microseconds>(t2-t1);

    while (1);

    return 0;
}
```