Санкт-Петербугский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: объектно-ориентированное программирование

Тема: АВЛ Дерево

Студенты гр. 3331506 / 20102 Акулов А.А.

Преподаватель Ананьевский М. С.

Санкт-Петербург

2025

Оглавление

Введение

АВЛ-дерево — это сбалансированное бинарное дерево поиска, в котором для каждой вершины разность высот левого и правого поддеревьев не превышает 1. Благодаря этому обеспечивается логарифмическая сложность операций поиска, вставки и удаления.

Этот тип данных был впервые введен в 1962 году Г. М. Адельсоном-Вельским и Е. М. Ландисом [1], первые буквы фамилий которых стали названием их изобретения.

У АВЛ Дерева разница высот правого и левого поддерева любого узла лежит в диапазоне $\{-1, 0, 1\}$. Ввиду этого высоту дерева с n элементами можно представить как:

$$h = O(\log n).$$

Для сохранения сбалансированности дерева после каждой операции добавления или удаления вершины нужно производить балансировку. Есть четыре типа балансировки: малое левое вращение, малое правое вращение, большое левое вращение и большое правое вращение. Подробнее они описаны в основной части работы. Балансировка требует $O(1)$.

Так как в процессе добавления, удаления или поиска вершины мы рассматриваем не более, чем $O(h)$. вершин дерева, и для каждой запускаем балансировку не более одного раза, то суммарное количество операций при включении новой вершины в дерево составляет $O(\log\ n)$ операций. Зависимость времени операций от количества вершин представлена на рисунке 1.

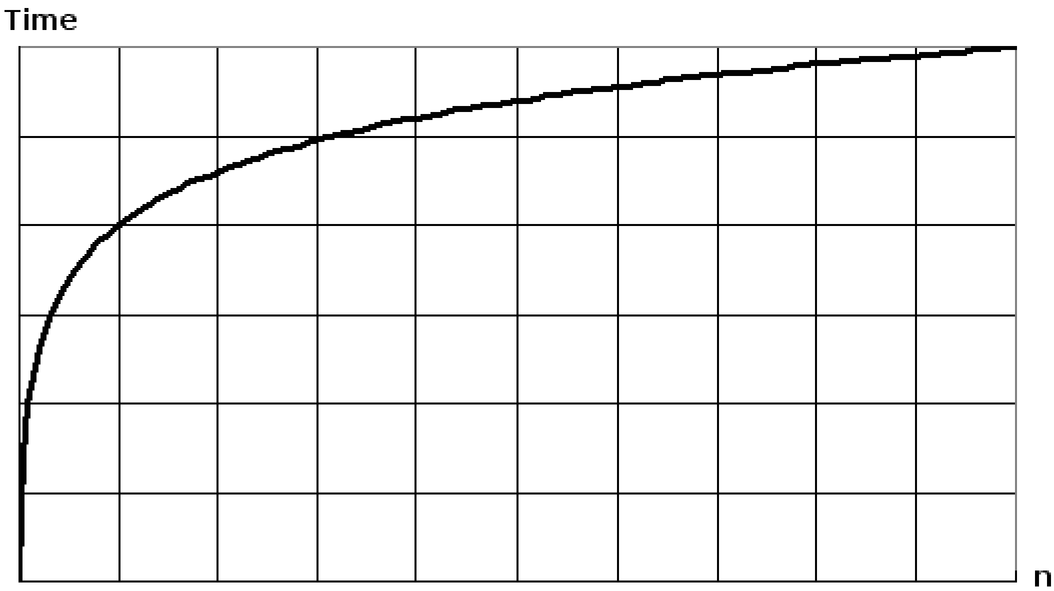

Рисунок 1 – Зависимость времени от числа элементов

АВЛ Деревья широко применяются для хранения, поиска и сортировки данных ввиду эффективности организации.

Основная часть

Организация дерева

Для узлов дерева создадим отдельный класс *NodeTree*. Главные элементы этого класса:

- *data* – данные, хранимые в узле
- *key* – ключ, по которому осуществляется сортировка
- *left_child* – указатель на левого ребенка
- *right_child* – указатель на правого ребенка
- *height* – высота наибольшего из поддеревьев
- *bf* – коэффициент сбалансированности (разность высот левого и правого поддеревьев)

Само АВЛ Дерево реализовано классом *Tree*. Дерево определяется корнем *root* – указателем на узел, являющийся корнем.

Основные операции

1. Малое левое вращение

Этот метод балансировки применяется в случае, если коэффициент сбалансированности узла меньше -1, а коэффициент сбалансированности его правого ребенка неположительный. Схема вращения представлена на рисунке 2.

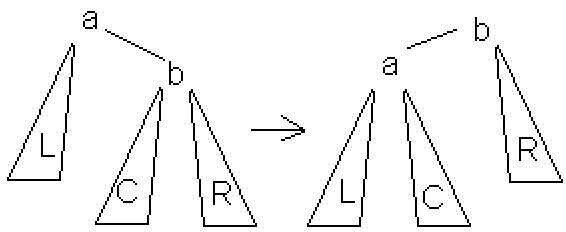

Рисунок 2

Малое левое вращение реализовано в методе класса *Tree l_rotate*.

2. Малое правое вращение

Этот метод балансировки применяется в случае, если коэффициент сбалансированности узла больше 1, а коэффициент сбалансированности его правого ребенка неотрицательный. Схема вращения представлена на рисунке 3.

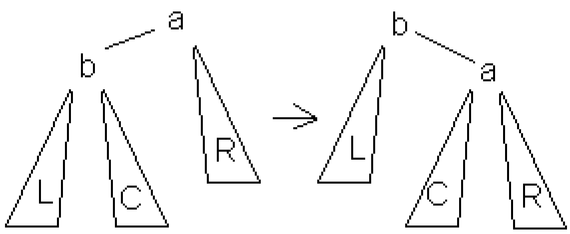

Рисунок 3

Малое правое вращение реализовано в методе класса *Tree r_rotate*.

3. Большое левое вращение

Этот метод балансировки применяется в случае, если коэффициент сбалансированности узла меньше -1, а коэффициент сбалансированности его правого ребенка положительный. Схема вращения представлена на рисунке 4.

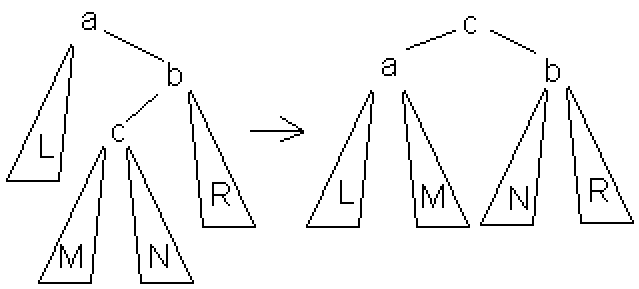

Рисунок 4

Большое левое вращение реализовано в методе класса *Tree rl_rotate*.

4. Большое правое вращение

Этот метод балансировки применяется в случае, если коэффициент сбалансированности узла меньше -1, а коэффициент сбалансированности его правого ребенка неположительный. Схема вращения представлена на рисунке 5.

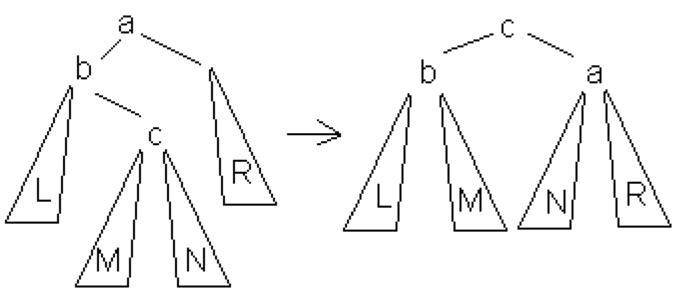

Рисунок 5

Большое правое вращение реализовано в методе класса *Tree lr_rotate*.

5. Балансировка

После каждого добавления или удаления узла стек со всеми узлами, которые находятся выше добавленного или удаленного, передается в функцию, которая обновляет значения *height* и *bf* и определяет, необходима ли балансировка и, если необходима, то какого типа.

Балансировка реализована в методе класса *Tree balance*.

6. Добавление узла

Новые элементы вставляются на место листа (отсутствующего ребенка). Для добавления нового узла мы сравниваем новый ключ с ключом текущего узла (начиная с корня) пока текущий узел не станет *nullptr*: если новый ключ больше текущего, текущим узлом становится правый ребенок, если меньше – левый, если новый ключ совпадает с текущим – выводим ошибку: “ Node with this key already exists”. Каждый текущий узел последовательно вносится в стек, который после добавления узла будет передан для балансировки.

Добавление узла реализовано в методе класса *Tree add*.

7. Удаление узла

Для удаления узла мы сравниваем удаляемый ключ с ключом текущего узла (начиная с корня) пока текущий узел не станет *nullptr*: если новый ключ больше текущего, текущим узлом становится правый ребенок, если меньше – левый, если новый ключ совпадает с текущим – начинаем процесс удаления и прерываем цикл. Если цикл завершился натуральным образом, выводим ошибку: “Node does not exist”. Каждый текущий узел последовательно вносится в стек, который после удаления узла будет передан для балансировки.

Процесс удаления делится на при типа:

- У удаляемого узла нет детей: заменяем удаляемый узел на *nullptr*.
- У удаляемого узла 1 ребенок: заменяем удаляемый узел на ребенка.
- У удаляемого узла 2 ребенка: заменяем удаляемый узел на крайнего правого потомка левого ребенка удаляемого узла.

Добавление узла реализовано в методе класса *Tree del_by_key*.

Также реализован метод *del_by_data*. Он вызываем последовательно функции поиска и удаления по ключу.

8. Поиск

С помощью очереди совершаем обход дерева в ширину, сравнивая искомое значение с текущими значениями узлов. Если находим совпадение, возвращаем ключ узла, если не находим – выводим ошибку: “ Node does not exist”.

Поиск узла реализован в методе класса *Tree get_key*.

Исследование

Протестируем полученное дерево. При помощи программы, представленной ниже, оценим время, затраченное на добавление и удаление узлов при разном количестве узлов. Результаты измерений представлены на рисунках 7 – для добавления новых узлов, 8 – для удаления.

```
int main() {
    clock_t big_start = clock();
    const int iter = 10000;
    Tree A;
    int values[iter];

    for (int i = 0; i < iter; ++i) {
        int rand = std::rand();
        values[i] = rand;

        clock_t start = clock();
        try {
            A.add(rand);
        }
        catch (const Tree_Exception& e) {
            std::cerr << e.what() << std::endl;
        }
        clock_t end = clock();

        double seconds = (double)(end - start);
        std::cout << seconds << std::endl;

    }

    std::cout << "\nDelete:\n\n";

    for (int i = 0; i < iter; ++i) {
        int rand = std::rand() % iter;
```

```
        int value = values[rand];

        clock_t start = clock();
        try {
            A.del_by_key(value);
        }
        catch (const Tree_Exception& e) {
            std::cerr << e.what() << std::endl;
        }
        clock_t end = clock();

        double seconds = (double)(end - start);
        std::cout << seconds << std::endl;

    }
    clock_t big_end = clock();
    double seconds = (double)(big_end - big_start) / CLOCKS_PER_SEC;
    std::cout << seconds << std::endl;

    return 0;
}
```

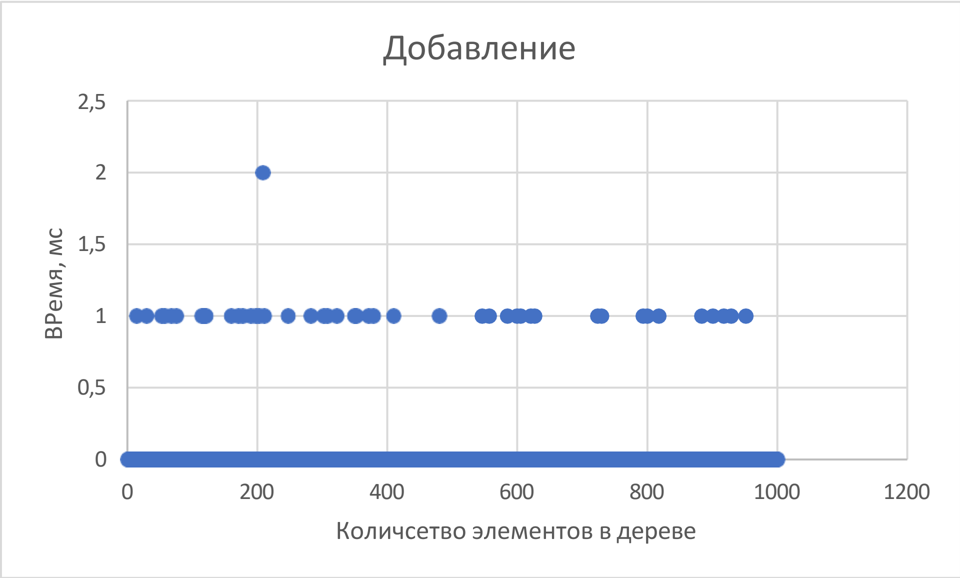

Рисунок 6

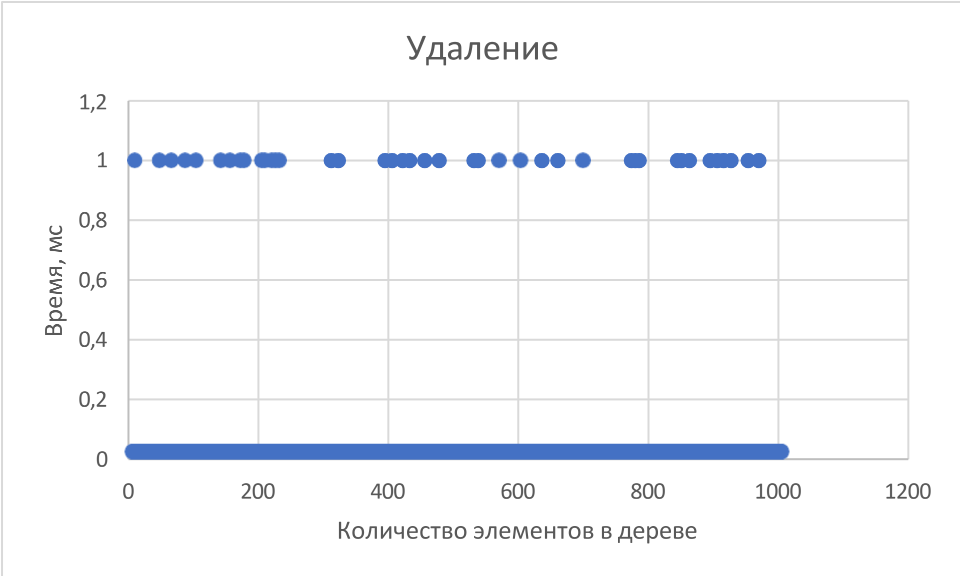

Рисунок 7

Заключение

После реализации алгоритма АВЛ-дерева и анализа его характеристик можно заключить следующее:

Применение АВЛ-деревьев вместо обычных бинарных деревьев поиска ускоряет операции поиска, добавления и удаления элементов при равном их количестве. Однако это усложняет алгоритм работы с деревом, поскольку после каждой вставки или удаления требуется проверять балансировку и при необходимости выполнять ротацию узлов.

АВЛ-деревья целесообразно применять при обработке значительных объемов данных, где важна эффективность операций.

Список литературы

- Вирт Н. Алгоритмы и структуры данных. — М.: Мир, 1989. — С. 272—286.
- Адельсон-Вельский Г. М., Ландис Е. М. Один алгоритм организации информации // Доклады АН СССР. — 1962. — Т. 146, № 2. — С. 263—266.

Приложение

Код программы:

```
#include <iostream>
#include <queue>
#include <stack>
#include <ctime>

class Tree_Exception : public std::exception
{
public:
    Tree_Exception(const char* const& msg) : exception(msg)
    {}
};

Tree_Exception ALREADY_EXISTS("Node with this key already exists");
Tree_Exception DOES_NOT_EXISTS("Node does not exist");

typedef double T;
T GEN = 0;

class NodeTree {
protected:
    T data;
    int key;
    NodeTree* left_child;
    NodeTree* right_child;
    int height;
    int bf; // balance factor = diiference between hights of left and right subtrees
```

```
public:
    NodeTree(int key, NodeTree* left_child = nullptr, NodeTree* right_child =
nullptr, int height = 0, int bf = 0);
    NodeTree(const NodeTree& node);
    ~NodeTree();
    void update(); // updates balance factor and height

    friend class Tree;
};

typedef std::stack<NodeTree**> node_stack;

class Tree {
protected:
    NodeTree* root = nullptr;

public:
    Tree() = default;
    ~Tree();

    void add(int new_key);

    void del_by_data(T del_data);
    void del_by_key(int del_key);

    T get_data(int key);
    int get_key(T data);


    void balance(node_stack stack);
```

```
    void l_rotate(NodeTree** node);
    void r_rotate(NodeTree** node);
    void lr_rotate(NodeTree** node);
    void rl_rotate(NodeTree** node);
};

NodeTree::NodeTree(int key, NodeTree* left_child, NodeTree* right_child, int
height, int bf) {

    this->data = GEN;
    this->key = key;
    this->left_child = left_child;
    this->right_child = right_child;
    this->bf = bf;
    this->height = height;

    ++GEN;
}

NodeTree::NodeTree(const NodeTree& node) {
    data = node.data;
    key = node.key;
    bf = node.bf;
    height = node.height;
    left_child = node.left_child;
    right_child = node.right_child;
}

NodeTree::~NodeTree() {
    data = NULL;
```

```
    left_child = nullptr;
    right_child = nullptr;
}

void NodeTree::update() {
    int lheight = 1 + (left_child == nullptr ? -1 : left_child->height);
    int rheight = 1 + (right_child == nullptr ? -1 : right_child->height);
    bf = lheight - rheight;
    height = lheight > rheight ? lheight : rheight;
}

Tree::~Tree() {
    root = nullptr;
}

void Tree::add(int new_key) {
    NodeTree* new_node = new NodeTree(new_key);
    NodeTree** temp = &root;
    node_stack stack;
    while (*temp != nullptr) {
        if ((*temp)->key == new_key) throw ALREADY_EXISTS;
        stack.push(temp);
        ((*temp)->key > new_key) ? temp = &((*temp)->left_child) : temp =
&((*temp)->right_child);
    }
    *temp = new_node;
    balance(stack);
}

T Tree::get_data(int key) {
```

```
    if (root == nullptr) throw DOES_NOT_EXISTS;
    NodeTree** temp = &root;
    while (temp != nullptr) {
        if ((*temp)->key == key) {
            return (*temp)->data;
        }
        if ((*temp)->key > key) {
            temp = &((*temp)->left_child);
        }
        else {
            temp = &((*temp)->right_child);
        }
    }
    throw DOES_NOT_EXISTS;
}

int Tree::get_key(T data) {
    if (root == nullptr) throw DOES_NOT_EXISTS;
    T result = NULL;
    NodeTree* temp = root;
    std::queue<NodeTree*> queue;
    queue.push(temp);

    while (!queue.empty()) {
        temp = queue.front();
        queue.pop();
        if (temp->data == data) {
            return temp->key;
        }
        if (temp->left_child != nullptr) queue.push(temp->left_child);
```

```
        if (temp->right_child != nullptr) queue.push(temp->right_child);
    }
    throw DOES_NOT_EXISTS;
}


void Tree::del_by_key(int del_key) {
    NodeTree** temp = &root;
    node_stack stack;
    while (*temp != nullptr) {
        if ((*temp)->key == del_key) {

            if ((*temp)->left_child == nullptr && (*temp)->right_child == nullptr) {
                *temp = nullptr;
                balance(stack);
                return;
            }
            // 1 child
            if ((*temp)->left_child != nullptr && (*temp)->right_child == nullptr) {
                *temp = (*temp)->left_child;
                balance(stack);
                return;
            }
            if ((*temp)->right_child != nullptr && (*temp)->left_child == nullptr) {
                *temp = (*temp)->right_child;
                balance(stack);
                return;
            }
            // 2 children
            NodeTree* change = (*temp)->left_child;
```

```
            while (change->right_child != nullptr) {
                change = change->right_child;
            }
            int change_key = change->key;
            T change_data = change->data;
            del_by_key(change_key);
            (*temp)->key = change_key;
            (*temp)->data = change_data;
            return;
        }

        stack.push(temp);

        if ((*temp)->key > del_key) {
            temp = &((*temp)->left_child);
        }
        else {
            temp = &((*temp)->right_child);
        }
    }
    throw DOES_NOT_EXISTS;
}

void Tree::del_by_data(T del_data) {
    del_by_key(get_key(del_data));
}

void Tree::balance(node_stack stack) {
    NodeTree** temp;
    while (!stack.empty()) {
```

```
        temp = stack.top();

        //std::cout << (*temp)->key << " " << (*temp)->height << " " << (*temp)->bf
<< std::endl;

        (*temp)->update();

        //std::cout << (*temp)->key << " " << (*temp)->height << " " << (*temp)->bf
<< std::endl;

        if ((*temp)->bf < -1) {
            (*temp)->right_child->bf <= 0 ? l_rotate(temp) : rl_rotate(temp);
            stack.~stack();
            return;
        }
        else if ((*temp)->bf > 1) {
            (*temp)->left_child->bf >= 0 ? r_rotate(temp) : lr_rotate(temp);
            stack.~stack();
            return;
        }
        stack.pop();
    }
}

void Tree::l_rotate(NodeTree** node) {
    NodeTree* child = (*node)->right_child;

    (*node)->right_child = child->left_child;
    child->left_child = *node;
    *node = child;
```

```
    (*node)->left_child->update();
    (*node)->update();
}

void Tree::r_rotate(NodeTree** node) {
    NodeTree* child = (*node)->left_child;

    (*node)->left_child = child->right_child;
    child->right_child = *node;
    *node = child;
    (*node)->right_child->update();
    (*node)->update();
}

void Tree::lr_rotate(NodeTree** node) {
    l_rotate(&((*node)->left_child));
    r_rotate(node);
}

void Tree::rl_rotate(NodeTree** node){
    r_rotate(&((*node)->right_child));
    l_rotate(node);
}

int main() {
    clock_t big_start = clock();
    const int iter = 10000;
    Tree A;
    int values[iter];
```

```
for (int i = 0; i < iter; ++i) {
    int rand = std::rand();
    values[i] = rand;

    clock_t start = clock();
    try {
        A.add(rand);
    }
    catch (const Tree_Exception& e) {
        std::cerr << e.what() << std::endl;
    }
    clock_t end = clock();

    double seconds = (double)(end - start);
    std::cout << seconds << std::endl;

}

std::cout << "\nDelete:\n\n";

for (int i = 0; i < iter; ++i) {
    int rand = std::rand() % iter;

    int value = values[rand];

    clock_t start = clock();
    try {
        A.del_by_key(value);
    }
    catch (const Tree_Exception& e) {
```

```
        std::cerr << e.what() << std::endl;
      }
      clock_t end = clock();

      double seconds = (double)(end - start);
      std::cout << seconds << std::endl;

    }
    clock_t big_end = clock();
    double seconds = (double)(big_end - big_start) / CLOCKS_PER_SEC;
    std::cout << seconds << std::endl;

    return 0;
}
```