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 3](#_bookmark0)

[Основная часть 5](#_bookmark1)

[Организация дерева 5](#_bookmark2)

[Основные операции 5](#_bookmark3)

1. [Малое левое вращение 5](#_bookmark4)
2. [Малое правое вращение 6](#_bookmark5)
3. [Большое левое вращение 7](#_bookmark6)
4. [Большое правое вращение 7](#_bookmark7)
5. [Балансировка 8](#_bookmark8)
6. [Добавление узла 8](#_bookmark9)
7. [Удаление узла 8](#_bookmark10)
8. [Поиск 9](#_bookmark11)

[Заключение 12](#_bookmark12)

[Список литературы 13](#_bookmark13)

[Приложение 14](#_bookmark14)

# 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.

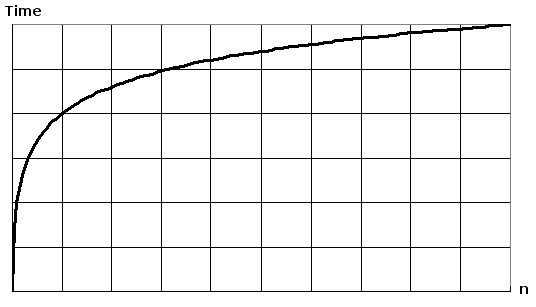


Рисунок 1 – Зависимость времени от числа элементов

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

## 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*.

## 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*.

## 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*.

## 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*.

## 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*.

## 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*.

## 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.

## 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;

}

ВРемя, мс

Рисунок 6



Добавление

2,5

2

1,5

1

0,5

0

0

200

400

600

800

1000

1200

Количсетво элементов в дереве

Время, мс



Количество элементов в дереве

1200

1000

800

600

400

200

0

1,2

1

0,8

0,6

0,4

0,2

0

Удаление

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;

}