МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

АНАЛИЗ АВЛ - ДЕРЕВА

ОТЧЁТ

студента 2 курса 251 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Лазаревой Виктории Владимировны

Проверено:

доцент, к. ф.-м. н. _____ М. И. Сафрончик

Саратов 2025

СОДЕРЖАНИЕ

1 Анализ АВЛ - дерева

1.1 Описание структуры и спецификаций методов

```
#pragma once

#include <string>

struct AVLNode {
  int value;
  AVLNode *left;
  AVLNode *right;
  int height;

  explicit AVLNode(int val)
      : value(val), left(nullptr), right(nullptr), height(1) {}
};

struct AVLTree {
  AVLNode *root;

  AVLTree() : root(nullptr) {}

  int calculateHeight(AVLNode *node);
  int getBalanceFactor(AVLNode *node);
  AVLNode *rotateRight(AVLNode *y);
  AVLNode *rotateLeft(AVLNode *x);
  AVLNode *insertNode(AVLNode *node, int value);
  AVLNode *findMinValueNode(AVLNode *node);
  AVLNode *deleteNode(AVLNode *node, int value);
  void printTreeHelper(AVLNode *node, std::string indentation, bool
isLast);
```

```
  void insert(int value);
  void remove(int value);
  void print();
};
```

1.2 Реализация структуры и методов

```
#include "AVL.h"
#include <algorithm>
#include <iostream>

int AVLTree::calculateHeight(AVLNode *node) {
  if (node == nullptr)
    return 0;
  return node->height;
}

int AVLTree::getBalanceFactor(AVLNode *node) {
  if (node == nullptr)
    return 0;
  return calculateHeight(node->left) - calculateHeight(node-
>right);
}

AVLNode *AVLTree::rotateRight(AVLNode *y) {
  AVLNode *x = y->left;
  AVLNode *T2 = x->right;

  x->right = y;
  y->left = T2;

  y->height = std::max(calculateHeight(y->left), calculateHeight(y-
>right)) + 1;
```

```
  x->height = std::max(calculateHeight(x->left), calculateHeight(x->right)) + 1;

  return x;
}

AVLNode *AVLTree::rotateLeft(AVLNode *x) {
  AVLNode *y = x->right;
  AVLNode *T2 = y->left;

  y->left = x;
  x->right = T2;

  x->height = std::max(calculateHeight(x->left), calculateHeight(x->right)) + 1;
  y->height = std::max(calculateHeight(y->left), calculateHeight(y->right)) + 1;

  return y;
}

AVLNode *AVLTree::insertNode(AVLNode *node, int value) {
  if (node == nullptr) {
    return new AVLNode(value);
  }
  if (value < node->value) {
    node->left = insertNode(node->left, value);
  } else if (value > node->value) {
    node->right = insertNode(node->right, value);
  } else {
    return node;
```

```
  }

  node->height =
      1 + std::max(calculateHeight(node->left),
calculateHeight(node->right));
  int balance = getBalanceFactor(node);

  if (balance > 1 && value < node->left->value)
    return rotateRight(node);

  if (balance < -1 && value > node->right->value)
    return rotateLeft(node);

  if (balance > 1 && value > node->left->value) {
    node->left = rotateLeft(node->left);
    return rotateRight(node);
  }

  if (balance < -1 && value < node->right->value) {
    node->right = rotateRight(node->right);
    return rotateLeft(node);
  }

  return node;
}

AVLNode *AVLTree::findMinValueNode(AVLNode *node) {
  AVLNode *current = node;
  while (current->left != nullptr)
    current = current->left;
  return current;
```

```
}

AVLNode *AVLTree::deleteNode(AVLNode *node, int value) {
  if (node == nullptr)
    return node;

  if (value < node->value) {
    node->left = deleteNode(node->left, value);
  } else if (value > node->value) {
    node->right = deleteNode(node->right, value);
  } else {
    if (node->left == nullptr || node->right == nullptr) {
      AVLNode *temp = node->left ? node->left : node->right;

      if (temp == nullptr) {
        temp = node;
        node = nullptr;
      } else {
        *node = *temp;
      }
      delete temp;
    } else {
      AVLNode *temp = findMinValueNode(node->right);
      node->value = temp->value;
      node->right = deleteNode(node->right, temp->value);
    }
  }

  if (node == nullptr)
    return node;
```

```
  node->height =
      1 + std::max(calculateHeight(node->left),
calculateHeight(node->right));
  int balance = getBalanceFactor(node);

  if (balance > 1 && getBalanceFactor(node->left) >= 0)
    return rotateRight(node);

  if (balance > 1 && getBalanceFactor(node->left) < 0) {
    node->left = rotateLeft(node->left);
    return rotateRight(node);
  }

  if (balance < -1 && getBalanceFactor(node->right) <= 0)
    return rotateLeft(node);

  if (balance < -1 && getBalanceFactor(node->right) > 0) {
    node->right = rotateRight(node->right);
    return rotateLeft(node);
  }

  return node;
}

void AVLTree::insert(int value) { root = insertNode(root, value); }

void AVLTree::remove(int value) { root = deleteNode(root, value); }

void AVLTree::printTreeHelper(AVLNode *node, std::string indent,
bool isLast) {
  if (node != nullptr) {
```

```
    std::cout << indent;
    if (isLast) {
      std::cout << "R----";
      indent += "   ";
    } else {
      std::cout << "L----";
      indent += "|  ";
    }
    std::cout << node->value << "(" << node->height << ")" <<
std::endl;
    printTreeHelper(node->left, indent, false);
    printTreeHelper(node->right, indent, true);
  }
}

void AVLTree::print() {
  if (root == nullptr) {
    std::cout << "Дерево пустое." << std::endl;
  } else {
    std::cout << "AVL дерево:" << std::endl;
    printTreeHelper(root, "", true);
  }
}
```

В лучшем случае никакие свойства не нарушены и нам не нужно делать ничего особенного: O(log n)

В худшем требуется не более 2 поворотов для восстановления баланса за константное время. И вновь временная сложность для операций вставки, удаления и поиска будет O(log n).

Таким образом, АВЛ, как и КЧД, решает проблему балансировки обычного бинарного дерева