|  |  |
| --- | --- |
| **Российский университет транспорта (МИИТ)**  **Институт транспортной техники и систем управления**  **Кафедра «Управление и защита информации»** | |
| **Курсовая работа**  **на тему**  **«Создание структуры данных AVL-дерево»**  **по дисциплине**  **«Системы управления базами данных и основы построения защищённых баз данных»** | |
|  | Выполнил:  Студент группы ТКИ-441,  Самохвалова М.Д. |
|  | Проверил:  Доцент кафедры УиЗИ, к.т.н.,  Васильева М. А.  Доцент кафедры УиЗИ, к.т.н.,  Балакина Е. П. |
|  |  |
| Москва – 2024 | |
|  | |

Оглавление

[Введение 3](#_Toc185944913)

[Цель работы 6](#_Toc185944914)

[Задачи работы 7](#_Toc185944915)

[1. Структура проекта 8](#_Toc185944916)

[2. Реализация программы 9](#_Toc185944917)

[3. Проверка с помощью тестов 11](#_Toc185944918)

[4. Проверка утечек памяти с помощью PVS Studio 13](#_Toc185944919)

[5. Схема алгоритма AVL-дерева 14](#_Toc185944920)

[Заключение 15](#_Toc185944921)

[Приложение 1. Реализация методов класса 16](#_Toc185944922)

[Приложение 2. Тесты 25](#_Toc185944923)

Введение

AVL-дерево — это самобалансирующееся двоичное дерево поиска (BST). Это означает, что оно сочетает в себе свойства BST (быстрый поиск, вставка и удаление при правильной организации) с механизмом автоматической балансировки, гарантирующим, что дерево не выродится в список, что могло бы привести к замедлению операций.

**Ключевые особенности AVL-дерева:**

1. **Двоичное дерево поиска (BST):**
   * Для каждого узла все ключи в его левом поддереве меньше, чем ключ самого узла.
   * Все ключи в его правом поддереве больше, чем ключ самого узла.
2. **Самобалансировка:**
   * Главная особенность AVL-дерева. После каждой операции вставки или удаления дерева, проверяется "фактор баланса" каждого узла.
   * **Фактор баланса (balance factor):** Разница между высотой его левого и правого поддерева. В AVL-дереве фактор баланса *каждого узла* всегда должен быть равен -1, 0 или 1.
   * Если фактор баланса выходит за эти пределы (то есть становится -2 или 2), то дерево перебалансируется с помощью специальных операций - **поворотов (rotations)**.
3. **Оптимальная высота:**
   * Благодаря самобалансировке, AVL-дерево имеет логарифмическую высоту относительно количества узлов. Это значит, что операции поиска, вставки и удаления выполняются за время O (log n), где n — количество узлов в дереве.

**Зачем нужно AVL-дерево?**

* **Производительность:** Гарантированная логарифмическая сложность для большинства операций, что делает его эффективным для работы с большими наборами данных.
* **Избежание вырождения:** В отличие от обычных BST, AVL-дерево не вырождается в список, который имеет линейную сложность для поиска.
* **Использование:**
  + Реализация словарей, где требуется быстрый поиск и гарантированное время выполнения операций.
  + Базы данных и индексы для быстрого доступа к данным.
  + Ситуации, где важно иметь гарантированную производительность, а не зависеть от случайного распределения данных.

**Основные операции в AVL-дереве:**

1. **Вставка (Insertion):**
   * Как в обычном BST, но после вставки дерево проверяется на балансировку.
   * Если фактор баланса нарушен, дерево перебалансируется с помощью поворотов.
2. **Удаление (Deletion):**
   * Как в обычном BST, но также после удаления дерево проверяется на балансировку.
   * Если фактор баланса нарушен, выполняется перебалансировка.
3. **Поиск (Search):**
   * Как в обычном BST, очень быстрая операция (O (log n)) благодаря сбалансированной структуре дерева.

**Виды поворотов:**

* **Правый поворот (Right Rotation):** используется, когда левое поддерево стало слишком высоким.
* **Левый поворот (Left Rotation):** используется, когда правое поддерево стало слишком высоким.
* **Комбинации поворотов (Left-Right и Right-Left):** применяются в более сложных ситуациях, когда нарушения баланса происходят не в самом высоком узле, а в его поддереве.

**Преимущества AVL-дерева:**

* **Гарантированная производительность:** O (log n) для основных операций.
* **Простота реализации:** несмотря на балансировку, алгоритмы достаточно понятны.
* **Устойчивость:** нет риска вырождения в список.

**Недостатки AVL-дерева:**

* **Сложность реализации:** Сложность алгоритмов перебалансировки.
* **Нагрузка на балансировку:** Операции вставки и удаления могут быть немного медленнее из-за необходимости перебалансировки дерева.
* **Потребление памяти:** требует дополнительное хранение информации о высоте узла.

**В заключение:**

AVL-дерево – это мощная структура данных, которая обеспечивает эффективные операции поиска, вставки и удаления, избегая при этом проблему вырождения. Оно подходит для задач, где требуется гарантированная производительность и не допускаются замедления из-за несбалансированной структуры.

Цель работы

Цель данной работы заключается в разработке и реализации программного кода, описывающего структуру данных AVL-дерева для хранения последовательностей элементов, с последующей демонстрацией основных операций, таких как добавление, удаление, доступ к элементам и итерация.

Задачи работы

1. Разработать структуру данных для AVL-дерева на выбранном языке программирования.
2. Реализовать алгоритмы выполнения основных операций с AVL-деревом, таких как добавление, удаление, доступ к элементам и итерация.
3. Провести тестирование разработанного кода на различных наборах данных для проверки корректности выполнения операций.
4. Выявить возможные узкие места и оптимизировать работу с памятью.
5. Структура проекта

Проект будет содержать три директории, которые будут располагаться в основной. Первая директория будет названа AVL-Tree и будет хранить файл, необходимый запуска проекта. Вторая будет хранить заголовочный файл и файл реализации. Третья директория Tests будет содержать CppUnit-тесты для программы.

1. Реализация программы

Программа реализована на языке программирования C++. В неё заложен основной функционал, необходимый для хранения данных и операций над ними. Программа реализует структуру данных AVL-дерево, поэтому среди основных функций – добавление элементов в конец, удаление элементов по индексу, доступ к элементам по индексу и итерация. Код реализует основные атрибуты AVL-дерева, включая массив для хранения данных, текущий размер и вместимость. Основной класс предоставляет методы для автоматического увеличения вместимости массива, управление памятью и работы с итераторами для удобного перебора элементов.

Особое внимание уделено обработке крайних случаев, таких как доступ к элементу по некорректному индексу, добавление в пустое AVL-дерево, удаление элементов в массиве минимального размера и управление динамически выделенной памятью. Для каждого метода предусмотрены проверки корректности работы, включая выброс исключений при ошибочных действиях.

Кроме того, в программе предусмотрена функция преобразования AVL-дерева в строку, что упрощает визуализацию текущего состояния структуры данных для анализа и отладки.

Для тестирования разработанного кода были подготовлены сценарии, включающие:

* добавление элементов;
* удаление элементов из различных позиций;
* проверку корректности доступа по индексу;
* оценку работы алгоритмов с большими объемами данных.

Результаты тестирования показали, что программа успешно справляется с поставленными задачами, обеспечивая корректность выполнения операций.

Программа разработана с учетом расширяемости: базовые методы могут быть дополнены функциями для работы с любыми типами данных через шаблоны, что позволит повысить универсальность структуры. Реализация методов класса представлена в Приложении 1, тесты находятся в Приложении 2.

Проект был собран с использованием среды разработки Visual Studio, которая является одним из наиболее популярных инструментов для создания приложений на C++ в операционной системе Windows. Visual Studio предоставляет мощные средства для управления сборкой, включая встроенный отладчик, профилировщик и поддержку модульного тестирования, что делает процесс разработки более удобным и продуктивным.

Использование Visual Studio позволило эффективно интегрировать тестовые модули с использованием CppUnitTest и автоматизировать процесс тестирования. Инструмент также обеспечивает удобный интерфейс для анализа кода, выявления потенциальных ошибок и профилирования производительности.

Благодаря интуитивно понятной настройке проектов, Visual Studio упростила управление зависимостями, а встроенные средства рефакторинга и анализа кода позволили улучшить читаемость и качество разработанного программного обеспечения.

1. Проверка с помощью тестов

Для проверки корректности работы программы была использована библиотека CppUnitTest, которая является одним из наиболее популярных инструментов для модульного тестирования на языке C++. Библиотека позволила автоматизировать процесс тестирования и выявить возможные ошибки на ранних этапах разработки.

Тесты были разработаны для следующих ключевых функций:

* Добавление элементов в вектор: проверялась корректность добавления элементов в конец вектора, включая обработку случаев, когда вместимость массива исчерпана, и требуется увеличение памяти.
* Удаление элементов: проверялось удаление элементов по различным индексам, включая обработку случаев, когда удаляется последний элемент, а также попытки удаления по некорректным индексам.
* Доступ к элементам: тестировалась корректность обращения к элементам по индексу, а также обработка случаев, когда индекс выходит за пределы текущего размера массива.
* Итерация по элементам: проверялась работа встроенного итератора, включая корректное прохождение по всем элементам вектора.

По результатам выполнения все тесты завершаются успешно, ошибок не обнаружено. Это подтверждает корректность работы реализации структуры данных, убедиться в этом возможно обратив внимание на Рисунок 1.

|  |
| --- |
|  |
|  |
| Рисунок 1 – Выполнение тестов для программы. |

1. Проверка утечек памяти с помощью PVS Studio

Для проверки корректности управления памятью программа была протестирована с использованием инструмента PVS Studio. PVS-Studio — это статический анализатор кода, который может использоваться для выявления утечек памяти, ошибок и других потенциальных проблем в C++ коде. Хотя PVS-Studio не предназначен специально для выявления утечек памяти в реальном времени, он помогает обнаружить ошибки в управлении памятью на уровне кода, которые могут привести к утечкам.

Результат проверки решения (solution), с фильтром под .h можно видеть на Рисунке 2. По результатам проверки утечек памяти не обнаружено.

|  |
| --- |
|  |
|  |
| Рисунок 2 – Проверка на наличие утечек памяти. |

1. Схема алгоритма avl-дерева

|  |
| --- |
|  |
|  |
| Рисунок 3 – Схема алгоритма AVL-дерева. |

Заключение

В ходе выполнения работы была реализована программа, демонстрирующая функциональность структуры данных AVL-дерева, а также произведено ее всестороннее тестирование и документирование. Реализация включила базовые операции с AVL-деревом, такие как добавление, удаление, изменение размера, доступ к элементам и итерации. Корректность программы была подтверждена с использованием современных инструментов тестирования и анализа.

Проведенное тестирование с помощью CppUnitTest подтвердило надежность алгоритмов, корректность обработки крайних случаев, таких как изменение размера AVL-дерева и доступ к несуществующим элементам, а также соответствие работы программы ожидаемым результатам. Проверка с использованием PVS-Studio показала отсутствие ошибок и предупреждений, связанных с утечками памяти, что свидетельствует о высоком качестве реализации.

Приложение 1. Реализация методов класса

#pragma once

#ifndef TREE\_H

#define TREE\_H

#include <iostream>

#include <string>

#include <sstream>

#include <queue>

class AVLTree {

private:

struct Node {

int data;

Node\* left;

Node\* right;

int height;

Node(int data);

~Node();

friend std::ostream& operator<<(std::ostream& os, const Node& node)

{

os << node.toString();

return os;

}

std::string toString() const;

};

Node\* root;

int height(Node\* node) const;

int balanceFactor(Node\* node) const;

Node\* rotateRight(Node\* y);

Node\* rotateLeft(Node\* x);

Node\* insertNode(Node\* node, int data);

Node\* findNode(Node\* node, int data) const;

Node\* findMinNode(Node\* node) const;

Node\* deleteNode(Node\* node, int data);

void deleteSubtree(Node\* node);

public:

AVLTree();

~AVLTree();

void insert(int data);

bool find(int data) const;

void remove(int data);

std::string toString() const;

void printTree() const;

friend std::ostream& operator<<(std::ostream& os, const AVLTree& tree);

friend std::ofstream& operator<<(std::ofstream& os, const AVLTree& tree);

friend std::ifstream& operator>>(std::ifstream& is, AVLTree& tree);

friend std::istream& operator>>(std::istream& is, AVLTree& tree);

};

// Реализация методов Node

AVLTree::Node::Node(int data) : data(data), left(nullptr), right(nullptr), height(1) {}

AVLTree::Node::~Node() {

}

std::string AVLTree::Node::toString() const {

std::stringstream ss;

ss << "Node(" << data << ", h=" << height << ")";

return ss.str();

}

// Реализация методов AVLTree

AVLTree::AVLTree() : root(nullptr) {}

AVLTree::~AVLTree() {

deleteSubtree(root);

}

int AVLTree::height(Node\* node) const {

return node ? node->height : 0;

}

int AVLTree::balanceFactor(Node\* node) const {

return node ? height(node->left) - height(node->right) : 0;

}

AVLTree::Node\* AVLTree::rotateRight(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

AVLTree::Node\* AVLTree::rotateLeft(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

AVLTree::Node\* AVLTree::insertNode(Node\* node, int data) {

if (!node) {

return new Node(data);

}

if (data < node->data) {

node->left = insertNode(node->left, data);

}

else if (data > node->data) {

node->right = insertNode(node->right, data);

}

else {

return node; // Дубликаты не добавляем

}

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

int balance = balanceFactor(node);

if (balance > 1 && data < node->left->data) {

return rotateRight(node);

}

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

return rotateLeft(node);

}

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

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

return rotateRight(node);

}

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

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

return rotateLeft(node);

}

return node;

}

void AVLTree::insert(int data) {

root = insertNode(root, data);

}

AVLTree::Node\* AVLTree::findNode(Node\* node, int data) const {

if (!node) {

return nullptr;

}

if (data == node->data) {

return node;

}

else if (data < node->data) {

return findNode(node->left, data);

}

else {

return findNode(node->right, data);

}

}

bool AVLTree::find(int data) const {

return findNode(root, data) != nullptr;

}

AVLTree::Node\* AVLTree::findMinNode(Node\* node) const {

while (node && node->left != nullptr) {

node = node->left;

}

return node;

}

AVLTree::Node\* AVLTree::deleteNode(Node\* node, int data) {

if (!node) {

return node;

}

if (data < node->data) {

node->left = deleteNode(node->left, data);

}

else if (data > node->data) {

node->right = deleteNode(node->right, data);

}

else {

if (!node->left || !node->right) {

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

delete node;

node = temp;

}

else {

Node\* temp = findMinNode(node->right);

node->data = temp->data;

node->right = deleteNode(node->right, temp->data);

}

}

if (!node) {

return node;

}

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

int balance = balanceFactor(node);

if (balance > 1 && balanceFactor(node->left) >= 0) {

return rotateRight(node);

}

if (balance > 1 && balanceFactor(node->left) < 0) {

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

return rotateRight(node);

}

if (balance < -1 && balanceFactor(node->right) <= 0) {

return rotateLeft(node);

}

if (balance < -1 && balanceFactor(node->right) > 0) {

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

return rotateLeft(node);

}

return node;

}

void AVLTree::remove(int data) {

root = deleteNode(root, data);

}

void AVLTree::deleteSubtree(Node\* node) {

if (node) {

deleteSubtree(node->left);

deleteSubtree(node->right);

delete node;

}

}

std::string AVLTree::toString() const {

if (!root) {

return "Tree is empty";

}

std::stringstream ss;

std::queue<Node\*> queue;

queue.push(root);

while (!queue.empty()) {

Node\* node = queue.front();

ss << node->toString() << ", ";

queue.pop();

if (node->left) queue.push(node->left);

if (node->right) queue.push(node->right);

}

std::string result = ss.str();

if (!result.empty()) {

result.pop\_back();

result.pop\_back();

}

return "{" + result + "}";

}

void AVLTree::printTree() const {

std::queue<Node\*> queue;

if (root)

queue.push(root);

while (!queue.empty())

{

Node\* temp = queue.front();

std::cout << temp->data << " ";

queue.pop();

if (temp->left)

queue.push(temp->left);

if (temp->right)

queue.push(temp->right);

}

std::cout << std::endl;

}

// Объявление операторов вне класса

std::ostream& operator<<(std::ostream& os, const AVLTree& tree);

std::ofstream& operator<<(std::ofstream& os, const AVLTree& tree);

std::ifstream& operator>>(std::ifstream& is, AVLTree& tree);

std::istream& operator>>(std::istream& is, AVLTree& tree);

#endif

Приложение 2. Тесты

#include "pch.h"

#include "CppUnitTest.h"

#include "../Tree/Node.h" // Убедись, что путь к tree.h верный

#include "../AVL-Tree/main.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace AVLTreeTests

{

TEST\_CLASS(AVLTreeTest)

{

public:

TEST\_METHOD(ConstructorDestructor)

{

AVLTree\* tree = new AVLTree();

delete tree;

Assert::IsTrue(true);

}

TEST\_METHOD(InsertFind)

{

AVLTree tree;

tree.insert(10);

tree.insert(20);

tree.insert(30);

Assert::IsTrue(tree.find(20));

Assert::IsFalse(tree.find(40));

}

TEST\_METHOD(Remove)

{

AVLTree tree;

tree.insert(10);

tree.insert(20);

tree.insert(30);

tree.remove(20);

Assert::IsFalse(tree.find(20));

Assert::IsTrue(tree.find(10));

Assert::IsTrue(tree.find(30));

}

TEST\_METHOD(ToString)

{

AVLTree tree;

tree.insert(10);

tree.insert(20);

tree.insert(30);

std::string expected = "{Node(20, h=2), Node(10, h=1), Node(30, h=1)}";

Assert::AreEqual(expected, tree.toString());

}

TEST\_METHOD(EmptyTreeToString)

{

AVLTree tree;

std::string expected = "Tree is empty";

Assert::AreEqual(expected, tree.toString());

}

TEST\_METHOD(FileOutputInput)

{

AVLTree tree;

tree.insert(10);

tree.insert(20);

tree.insert(30);

std::ofstream outputFile("output.txt");

outputFile << tree;

outputFile.close();

AVLTree tree2;

tree2.insert(10);

tree2.insert(20);

tree2.insert(30);

std::ifstream inputFile("output.txt");

inputFile >> tree2;

inputFile.close();

Assert::AreEqual(tree.toString(), tree2.toString());

}

TEST\_METHOD(ConsoleInput)

{

AVLTree tree;

std::stringstream ss;

ss << "10 20 30 40 abc";

std::istream is(ss.rdbuf());

is >> tree;

std::string expected = "{Node(20, h=3), Node(10, h=1), Node(30, h=2), Node(40, h=1)}";

Assert::AreEqual(expected, tree.toString());

}

};

// Переопределение операторов сдвига (чтобы работало с тестами)

std::ostream& operator<<(std::ostream& os, const AVLTree& tree) {

os << tree.toString();

return os;

}

std::ofstream& operator<<(std::ofstream& os, const AVLTree& tree)

{

os << tree.toString();

return os;

}

std::ifstream& operator>>(std::ifstream& is, AVLTree& tree) {

int value;

while (is >> value) {

tree.insert(value);

}

return is;

}

std::istream& operator>>(std::istream& is, AVLTree& tree) {

int value;

while (is >> value) {

tree.insert(value);

}

return is;

}

}