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

Оглавление

[Введение 3](#_Toc185425783)

[Цель работы 4](#_Toc185425784)

[Задачи работы 5](#_Toc185425785)

[1. Структура проекта 6](#_Toc185425786)

[2. Реализация программы 7](#_Toc185425787)

[3. Проверка с помощью тестов 9](#_Toc185425788)

[4. Проверка утечек памяти с помощью Valgrind 11](#_Toc185425789)

[Заключение 12](#_Toc185425790)

[Библиографический список 13](#_Toc185425791)

[Приложение 1. Основная программа 14](#_Toc185425792)

[Приложение 2. Файл тестов 31](#_Toc185425793)

[Приложение 3. Демонстрационный файл 49](#_Toc185425794)

Введение

Современные информационные технологии предъявляют высокие требования к эффективности хранения и обработки данных. Для решения этой задачи активно используются различные виды деревьев, среди которых особое внимание заслуживает AVL-дерево. Это сбалансированное бинарное дерево, которое обеспечивает высокую скорость операций вставки, удаления и поиска благодаря своей структуре, поддерживающей баланс между высотой поддеревьев.

В рамках исследования будут детально рассмотрены теоретические основы AVL-дерева, включая принципы его построения, операции поддержания баланса и особенности использования. На основе полученных знаний будет создана практическая реализация структуры данных на выбранном языке программирования. Разработка пройдет через серию тестов, направленных на оценку производительности и корректности кода.

Результаты данного проекта найдут свое применение как в углубленном изучении алгоритмов и структур данных, так и в реальных приложениях, требующих оптимизации процессов обработки информации.

Цель работы

Целью данной работы является разработка и реализация программного кода, представляющего собой структуру AVL-дерева для хранения данных, с последующим показом выполнения основных операций, таких как вставка, удаление, поиск и обход узлов дерева. Выбор AVL-дерева обусловлен его высокой эффективностью при доступе и обработке данных, что делает эту структуру востребованной в разнообразных программных задачах, требующих быстрой работы с большими объемами информации.

Задачи работы

1. Разработать структуру данных для AVL дерева на выбранном языке программирования.
2. Реализовать алгоритмы выполнения основных операций с бинарным деревом.
3. Провести тестирование разработанного кода на различных наборах данных для проверки корректности выполнения операций.
4. Структура проекта

Проект будет содержать три директории, которые будут располагаться в основной. Первая директория будет названа Demo и будет хранить файл, необходимый для демонстрационной работы программы. Вторая директория, носящая название Domain, будет хранить заголовочный файл с реализацией кода. Третья директория Test будет содержать Unit-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 бинарное дерево, поэтому среди основных функций – вставка элемента, удаление, корректный поиск элемента. Код реализует структуру узла бинарного дерева, которая содержит ключевые элементы: данные, указатели на левого и правого потомков. Основной класс бинарного дерева включает методы для добавления новых узлов, удаления существующих элементов, поиска данных и их вывода.

Особое внимание уделено обработке крайних случаев, таких как удаление узлов с одним потомком или с двумя, поиск в пустом дереве, а также попытки вставки уже существующих данных. Для каждого метода предусмотрены проверки корректности работы.

Кроме того, в программе предусмотрены функции визуализации дерева в текстовом формате для удобства анализа текущей структуры дерева и отладки.

Для тестирования разработанного кода были подготовлены сценарии, включающие:

* создание дерева из случайного набора данных,
* последовательную вставку и удаление элементов,
* проверку корректности поиска элементов,
* оценку работы алгоритмов на больших объемах данных.

Результаты тестирования показали, что программа успешно справляется с поставленными задачами, обеспечивая корректность выполнения операций. Производительность алгоритмов соответствует стандартным временным характеристикам бинарных деревьев: в случае несбалансированного дерева.

Программа разработана с учетом расширяемости: базовые методы могут быть дополнены функциями для реализации, например, балансировки дерева (алгоритмы AVL или красно-черного дерева) в будущем. Реализация заголовочного файла представлена в Приложении 1, реализация тестов представлена в Приложении 2, реализация демонстрационного файла находится в Приложении 3.

Проект был собран с использованием CMake, одного из самых популярных инструментов для управления сборкой в современных проектах. Выбор CMake обусловлен его гибкостью, кроссплатформенностью и широкой поддержкой современных инструментов разработки. CMake позволяет автоматизировать процесс конфигурации и сборки проекта, обеспечивая поддержку различных компиляторов и операционных систем. Кроме того, он упрощает интеграцию сторонних библиотек, таких как Google Test и Doxygen, что сделало процесс разработки более удобным и структурированным. Благодаря понятному и легко масштабируемому подходу, использование CMake способствует улучшению управляемости проекта, особенно в условиях его дальнейшего роста и расширения.

1. Проверка с помощью тестов

Для проверки корректности работы программы была использована библиотека Google Test, 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:

* Вставка элементов в дерево: проверялась корректность добавления узлов, включая обработку случаев, когда добавляется корневой элемент, а также элементов в левое и правое поддеревья.
* Удаление узлов: проверялось удаление листовых узлов, узлов с одним потомком и узлов с двумя потомками, с проверкой корректной перестройки дерева.
* Поиск элемента: тестировалась корректность нахождения существующих элементов и обработка случаев, когда элемент отсутствует в дереве.

По результатам выполнения все тесты завершаются успешно, ошибок не обнаруживается, что можно видеть на Рисунке 1.

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

1. Проверка утечек памяти с помощью Valgrind

Для проверки корректности управления памятью программа была протестирована с использованием инструмента Valgrind, который позволяет выявить утечки памяти и другие проблемы, связанные с динамическим выделением памяти.

Проверка проводилась на следующих сценариях:

1. Создание дерева, добавление элементов и его последующее удаление.
2. Множественное добавление и удаление элементов в процессе работы программы.
3. Создание дерева большого размера (более 100 000 узлов) и его удаление для оценки корректности освобождения памяти.

Результат одной из проверок можно видеть на Рисунке 2. По результатам всех проверок утечек памяти не обнаружено.

|  |
| --- |
|  |
|  |
| Рисунок 2 – Проверка на наличие утечек памяти(утечек нет).  Для демонстрации работы с инструментом была вызвана утечка (Рисунок 3). |

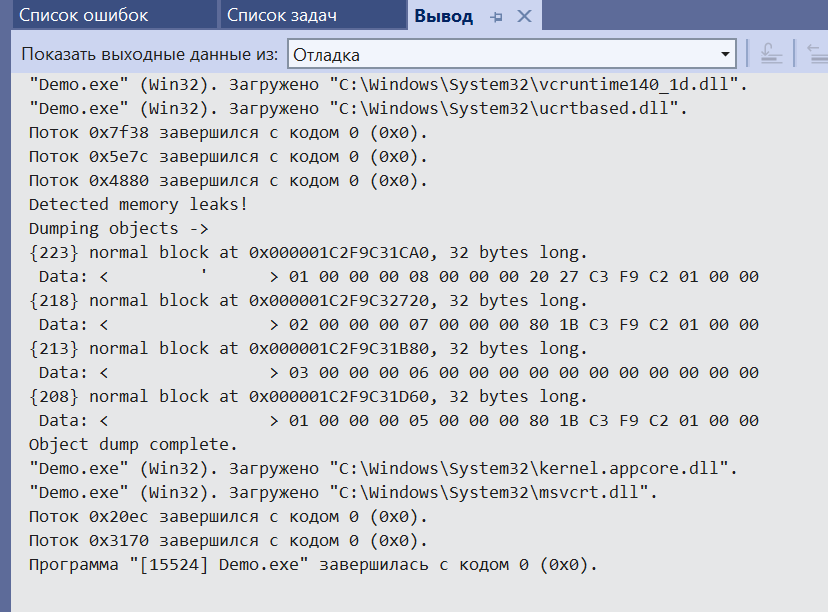


Рисунок 3 – Приложение с утечками

Заключение

В рамках данной работы была разработана программа, демонстрирующая работу с бинарным деревом данных, которая прошла тщательное тестирование и документирование. Реализованы базовые операции над деревом, включающие вставку, удаление, поиск элементов и различные методы обхода узлов. Правильность функционирования программы подтверждена современными инструментами тестирования и анализа кода.

Тестирование с использованием Unit Test показало надежность алгоритмов, корректную обработку граничных ситуаций и полное соответствие результатов ожиданиям. Анализ с применением Valgrind подтвердил отсутствие утечек памяти и правильное управление динамической памятью, что говорит о высоком уровне качества реализации.

Эта работа подчеркнула значимость структур данных, таких как бинарные деревья, для эффективной организации и обработки информации. Результаты исследования могут служить основой для дальнейших улучшений, среди которых автоматическая балансировка дерева (например, внедрение AVL-дерева или красно-чёрного дерева), повышение производительности или адаптация под конкретные задачи.

Библиографический список

1. Кормен, Т. Х., Лейзерсон, Ч. Э., Ривест, Р. Л., Штайн, К. Алгоритмы: построение и анализ. — Пер. с англ. — Москва: Вильямс, 2005. — 1296 с.
2. Рудакова, И. В. Структуры данных в C++. — Москва: ЛитРес, 2021. — 320 с.
3. Ничушкина, Т. Иерархические структуры данных: бинарные деревья. — Москва: ЛитРес, 2023. — 150 с.
4. Страуструп, Б. Язык программирования C++. — Пер. с англ. — Москва: Бином, 2015. — 1376 с.

Приложение 1. Основная программа

#include "Tree.h"

//Конструктор по умолчанию для "Дерева"

template<typename T>

Tree<T>::Tree() : root(nullptr)

{

}

//Конструктор копирования

template<typename T>

Tree<T>::Tree(const Tree& other)

{

copyTree(this->root, nullptr, other.root);

}

//Конструктор перемещения

template<typename T>

Tree<T>::Tree(Tree&& other) noexcept

{

this->root = other.root;

other.root = nullptr;

}

//Деструктор

template<typename T>

Tree<T>::~Tree()

{

destroyTree(root);

root = nullptr;

}

//Оператор присваивания с копированием

template<typename T>

Tree<T>& Tree<T>::operator=(const Tree<T>& other)

{

//проверяем не одинаковые ли адреса у this и other

if (this != &other)

{

destroyTree(this->root);

copyTree(this->root, nullptr, other.root);

}

return \*this;

}

//Оператор присваивания с перемещением

template<typename T>

Tree<T>& Tree<T>::operator=(Tree<T>&& other) noexcept

{

//проверяем не одинаковые ли адреса у this и other

if (this != &other)

{

destroyTree(this->root);

this->root = other.root;

other.root = nullptr;

}

return \*this;

}

//Очистка дерева

template<typename T>

void Tree<T>::clear()

{

destroyTree(this->root);

this->root = nullptr;

}

//Проверка дерева на пустоту

template<typename T>

bool Tree<T>::isEmpty() const

{

return (root == nullptr);

}

//Вставка элемента в дерево

template<typename T>

bool Tree<T>::insert(T value)

{

return insertTo(root, nullptr, value);

}

//Поиск элемента в дереве

template<typename T>

bool Tree<T>::find(T value) const

{

root->value;

return findIn(root, value);

}

//Удаление элемента из дерева

template<typename T>

bool Tree<T>::remove(T value)

{

return deleteIn(root, value);

}

//Получение строкового представления дерева

template<typename T>

std::string Tree<T>::toString() const

{

std::string result;

putToString(root, result);

if (!result.empty())

{

result.pop\_back();

}

return result;

}

//Печать структуры дерева

template<typename T>

void Tree<T>::printAsTree(std::ostream& out) const

{

print(root, out, 0);

}

//Получение итератора на первый элемент

template<typename T>

typename Tree<T>::Iterator Tree<T>::begin() const

{

if (root == nullptr)

{

return Iterator();

}

return Iterator(root->getTheMostLeft());

}

//Получение итератора на последний элемент

template<typename T>

typename Tree<T>::Iterator Tree<T>::last() const

{

if (root == nullptr)

{

return Iterator();

}

return Iterator(root->getTheMostRight());

}

//Получение итератора на элемент, следующий за последним

template<typename T>

typename Tree<T>::Iterator Tree<T>::end() const

{

return Iterator();

}

//Левый поворот

template<typename T>

void Tree<T>::leftRotate(Node\* a)

{

Node\* b = a->right;

//Узел a забирает левого ребёнка у b

a->right = b->left;

if (a->right != nullptr)

{

a->right->parent = a;

}

//Узел b забирает родителя a

b->parent = a->parent;

if (b->parent == nullptr)

{

root = b;

}

else

{

(a->parent->right == a ? b->parent->right : b->parent->left) = b;

}

//Узел a становится левым ребёнком b

b->left = a;

a->parent = b;

//Пересчёт высот для a и b

a->updateHeight();

b->updateHeight();

}

//Правый поворот

template<typename T>

void Tree<T>::rightRotate(Node\* a)

{

Node\* b = a->left;

//Узел а забирает правого ребенка у b

a->left = b->right;

if (a->left != nullptr)

{

a->left->parent = a;

}

//Узел b забирает родителя а

b->parent = a->parent;

if (b->parent == nullptr)

{

root = b;

}

else

{

(a->parent->left == a ? b->parent->left : b->parent->right) = b;

}

//узел а становится правым ребенком

b->right = a;

a->parent = b;

//Пересчет высот

a->updateHeight();

b->updateHeight();

}

//Правый-левый поворот

template<typename T>

void Tree<T>::rightLeftRotate(Node\* a)

{

rightRotate(a->right);

leftRotate(a);

}

//Левый-правый поворот

template<typename T>

void Tree<T>::leftRightRotate(Node\* a)

{

leftRotate(a->left);

rightRotate(a);

}

//Балансировка поддерева

template<typename T>

void Tree<T>::balanceSubtree(Node\* subtreeRoot)

{

int balanceFactor = subtreeRoot->getBalanceFactor();

if (balanceFactor == -2)

{

if (subtreeRoot->right->left == nullptr)

{

leftRotate(subtreeRoot);

}

else

{

rightLeftRotate(subtreeRoot);

}

}

else if (balanceFactor == +2)

{

if (subtreeRoot->left->right == nullptr)

{

rightRotate(subtreeRoot);

}

else

{

leftRightRotate(subtreeRoot);

}

}

}

//Рекурсивная функция вставки в поддерево

//subtreeRoot - корень поддерева, в которое вставляем новый элемент

template<typename T>

bool Tree<T>::insertTo(Node\*& subtreeRoot, Node\* subtreeParent, T newValue)

{

//найдено место для вставки

if (subtreeRoot == nullptr)

{

subtreeRoot = new Node(newValue, subtreeParent);

return true;

}

bool isInserted = false;

if (newValue < subtreeRoot->value)

{

isInserted = insertTo(subtreeRoot->left, subtreeRoot, newValue);

}

else if (newValue > subtreeRoot->value)

{

isInserted = insertTo(subtreeRoot->right, subtreeRoot, newValue);

}

if (isInserted)

{

subtreeRoot->updateHeight();

balanceSubtree(subtreeRoot);

}

//newValue == subtreeRoot->value

return isInserted;

}

//Рекурсивная функция поиска в поддереве

//subtreeRoot - корень поддерева, в котором ищем элемент

template<typename T>

bool Tree<T>::findIn(const Node\* subtreeRoot, T valueToFind)

{

//Если нет узла в поддереве

if (subtreeRoot == nullptr)

{

return false;

}

if (valueToFind < subtreeRoot->value)

{

return findIn(subtreeRoot->left, valueToFind);

}

if (valueToFind > subtreeRoot->value)

{

return findIn(subtreeRoot->right, valueToFind);

}

//valueToFind == subtreeRoot->value

return true;

}

//Рекурсивная функция удаления элемента в поддереве

//subtreeRoot - корень поддерева, в котором ищем элемент

template<typename T>

bool Tree<T>::deleteIn(Node\*& subtreeRoot, T valueToDelete)

{

//Если нет узла в поддереве

if (subtreeRoot == nullptr)

{

return false;

}

if (valueToDelete == subtreeRoot->value)

{

deleteNode(subtreeRoot);

return true;

}

bool isDeleted = false;

if (valueToDelete < subtreeRoot->value)

{

isDeleted = deleteIn(subtreeRoot->left, valueToDelete);

}

else if (valueToDelete > subtreeRoot->value)

{

isDeleted = deleteIn(subtreeRoot->right, valueToDelete);

}

if (isDeleted)

{

subtreeRoot->updateHeight();

balanceSubtree(subtreeRoot);

}

return isDeleted;

}

//Функция удаления узла из дерева

//linkToNode - ссылка(!) на удаляемый узел дерева

template<typename T>

void Tree<T>::deleteNode(Node\*& linkToNode)

{

Node\* parent = linkToNode->parent;

Node\* nodeToDelete = linkToNode;

if (linkToNode->left == nullptr && linkToNode->right == nullptr)

{

//удаление узла без детей

delete nodeToDelete;

linkToNode = nullptr;

return;

}

if (linkToNode->left != nullptr && linkToNode->right != nullptr)

{

//удаление узла с двумя детьми

//ищем наименьший элемент в правом дереве

Node\*& min = getMinIn(linkToNode->right);

std::swap(min->value, linkToNode->value);

deleteNode(min);

return;

}

//удаление узла с одним ребёнком

linkToNode = (linkToNode->left != nullptr) ? linkToNode->left : linkToNode->right;

linkToNode->parent = parent;

delete nodeToDelete;

}

//Рекурсивная функция поиска минимумального (наиболее левого) узла

template<typename T>

typename Tree<T>::Node\*& Tree<T>::getMinIn(Node\*& subtreeRoot)

{

if (subtreeRoot->left == nullptr)

{

return subtreeRoot;

}

return getMinIn(subtreeRoot->left);

}

//Рекурсивная функция печати в поток

template<typename T>

void Tree<T>::print(const Node\* subtreeRoot, std::ostream& out, int lvl)

{

if (subtreeRoot == nullptr)

{

return;

}

print(subtreeRoot->right, out, lvl + 1);

for (int i = 0; i < lvl; ++i)

{

out << " ";

}

out << subtreeRoot->value << std::endl;

print(subtreeRoot->left, out, lvl + 1);

}

//Рекурсивная функция вывода дерева в строку

template<typename T>

void Tree<T>::putToString(const Node\* subtreeRoot, std::string& str)

{

if (subtreeRoot == nullptr)

{

return;

}

putToString(subtreeRoot->left, str);

str += std::to\_string(subtreeRoot->value);

str += " ";

putToString(subtreeRoot->right, str);

}

//Рекурсивная функция копирования дерева

template<typename T>

void Tree<T>::copyTree(Node\*& subtreeRootTo, Node\* subtreeParent, const Node\* subtreeRootFrom)

{

//если нет узла для создания клона

if (subtreeRootFrom == nullptr)

{

subtreeRootTo = nullptr;

return;

}

subtreeRootTo = new Node(subtreeRootFrom->value, subtreeParent);

subtreeRootTo->height = subtreeRootFrom->height;

copyTree(subtreeRootTo->left, subtreeRootTo, subtreeRootFrom->left);

copyTree(subtreeRootTo->right, subtreeRootTo, subtreeRootFrom->right);

}

//Рекурсивная функция удаления дерева из памяти

template<typename T>

void Tree<T>::destroyTree(const Node\* subtreeRoot)

{

if (subtreeRoot == nullptr)

{

return;

}

destroyTree(subtreeRoot->left);

destroyTree(subtreeRoot->right);

delete subtreeRoot;

}

//Конструктор с параметрами для "Узла"

template<typename T>

Tree<T>::Node::Node(T value, Node\* parent)

{

height = 1;

Node::value = value;

Node::parent = parent;

left = nullptr;

right = nullptr;

}

//Пересчитывает высоту на узле

template<typename T>

void Tree<T>::Node::updateHeight()

{

//находимся в узле (this - это ЭТОТ узел):

int leftHeight = (this->left == nullptr) ? 0 : (this->left->height);

int rightHeight = (this->right == nullptr) ? 0 : (this->right->height);

if (leftHeight > rightHeight)

{

this->height = leftHeight + 1;

}

else

{

this->height = rightHeight + 1;

}

}

//Рассчёт фактора балансировки

template<typename T>

int Tree<T>::Node::getBalanceFactor() const

{

int leftHeight = (this->left == nullptr) ? 0 : (this->left->height);

int rightHeight = (this->right == nullptr) ? 0 : (this->right->height);

return (leftHeight - rightHeight);

}

//Получение самого левого потомка

template<typename T>

typename const Tree<T>::Node\* Tree<T>::Node::getTheMostLeft() const

{

const Node\* ptr = this;

while (ptr->left != nullptr)

{

ptr = ptr->left;

}

return ptr;

}

//Получение самого правого потомка

template<typename T>

typename const Tree<T>::Node\* Tree<T>::Node::getTheMostRight() const

{

const Node\* ptr = this;

while (ptr->right != nullptr)

{

ptr = ptr->right;

}

return ptr;

}

//Конструктор по умолчанию для пустого итератора

template<typename T>

Tree<T>::Iterator::Iterator()

{

element = nullptr;

}

//Конструктор с параметрами для создания итератора на данный узел

template<typename T>

Tree<T>::Iterator::Iterator(const Node\* ptr)

{

element = ptr;

}

//Проверка итераторов на равенство

template<typename T>

bool Tree<T>::Iterator::operator==(const Iterator& other) const

{

return this->element == other.element;

}

//Проверка итераторов на неравенство

template<typename T>

bool Tree<T>::Iterator::operator!=(const Iterator& other) const

{

return !(this->operator==(other));

}

//Перегрузка оператора инкремента (префиксного)

template<typename T>

typename Tree<T>::Iterator& Tree<T>::Iterator::operator++()

{

//Если элемент end(), то идти дальше нельзя

if (element == nullptr)

{

throw std::out\_of\_range("Попытка выхода за end().");

}

//Проверяем, есть ли узлы справа

if (element->right != nullptr)

{

//Ищем самый левый справа

element = element->right->getTheMostLeft();

return \*this;

}

//Если некуда подниматься

if (element->parent == nullptr)

{

//Элементы закончились, устанавливаем end()

element = nullptr;

return \*this;

}

//Если узел - левый ребёнок

if (element->parent->left == element)

{

//Идём к родителю

element = element->parent;

return \*this;

}

//Если узел - правый ребёнок

//Идём к прародителю, у которого узел слева

while (element->parent != nullptr && element->parent->right == element)

{

element = element->parent;

}

//Если при подъёме предки закончились, то end()

if (element->parent == nullptr)

{

element = nullptr;

return \*this;

}

//Если нашли родителя, большего, чем узел, то переходим на него

element = element->parent;

return \*this;

}

//Перегрузка оператора инкремента (постфиксного)

template<typename T>

typename Tree<T>::Iterator Tree<T>::Iterator::operator++(int)

{

Iterator copy = \*this;

this->operator++();

return copy;

}

//Перегрузка оператора декремента (префиксного)

template<typename T>

typename Tree<T>::Iterator& Tree<T>::Iterator::operator--()

{

//Если элемент end(), то от него двигаться нельзя

if (element == nullptr)

{

throw std::out\_of\_range("Попытка прохода до end().");

}

//Проверяем, есть ли узлы слева

if (element->left != nullptr)

{

//Ищем самый правый слева

element = element->left->getTheMostRight();

return \*this;

}

//Если некуда подниматься

if (element->parent == nullptr)

{

//Элементы закончились, устанавливаем end()

element = nullptr;

return \*this;

}

//Если узел - правый ребёнок

if (element->parent->right == element)

{

//Идём к родителю

element = element->parent;

return \*this;

}

//Если узел - левый ребёнок

//Идём к прародителю, у которого узел справа

while (element->parent != nullptr && element->parent->left == element)

{

element = element->parent;

}

//Если при подъёме предки закончились, то end()

if (element->parent == nullptr)

{

element = nullptr;

return \*this;

}

//Если нашли родителя, меньшего, чем узел, то переходим на него

element = element->parent;

return \*this;

}

//Перегрузка оператора декремента (постфиксного)

template<typename T>

typename Tree<T>::Iterator Tree<T>::Iterator::operator--(int)

{

Iterator copy = \*this;

this->operator--();

return copy;

}

//Перегрузка оператора разыменования

template<typename T>

const T& Tree<T>::Iterator::operator\*() const

{

//Если элемент end(), то значение получить нельзя

if (element == nullptr)

{

throw std::out\_of\_range("Попытка чтение в end().");

}

return element->value;

}

Приложение 2. Файл тестов

#include "pch.h"

#include "CppUnitTest.h"

#include "../AVL\_tree/Tree.hpp"

#include <utility>

#include <string>

#include <sstream>

#include <stdexcept>

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

// Функция для сравнения двух итераторов методом AreEqual

static std::wstring ToString(const Tree<int>::Iterator& it) {

if (it == Tree<int>::Iterator())

{

return L"END";

}

return std::to\_wstring(\*it);

}

namespace Test

{

TEST\_CLASS(TreeTest)

{

public:

TEST\_METHOD(DefaultConstructor\_Success)

{

// Act

Tree<int> t;

// Assert

Assert::IsTrue(t.isEmpty());

}

TEST\_METHOD(Destructor\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Act

original.~Tree();

// Assert

Assert::IsTrue(original.isEmpty());

}

TEST\_METHOD(CopyConstructor\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Act

Tree<int> clone = original;

// Assert

for (int i = 1; i < 10; ++i)

{

Assert::IsTrue(original.find(i));

}

for (int i = 1; i < 10; ++i)

{

Assert::IsTrue(clone.find(i));

}

}

TEST\_METHOD(MoveConstructor\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Act

Tree<int> clone = std::move(original);

// Assert

Assert::IsTrue(original.isEmpty());

for (int i = 1; i < 10; ++i)

{

Assert::IsTrue(clone.find(i));

}

}

TEST\_METHOD(CopyAssignmentOperator\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

Tree<int> clone;

for (int i = 20; i < 40; ++i)

{

clone.insert(i);

}

// Act

clone = original;

// Assert

for (int i = 1; i < 10; ++i)

{

Assert::IsTrue(original.find(i));

}

for (int i = 1; i < 10; ++i)

{

Assert::IsTrue(clone.find(i));

}

}

TEST\_METHOD(MoveAssignmentOperator\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

Tree<int> clone;

for (int i = 20; i < 40; ++i)

{

clone.insert(i);

}

// Act

clone = std::move(original);

// Assert

Assert::IsTrue(original.isEmpty());

for (int i = 1; i < 10; ++i)

{

Assert::IsTrue(clone.find(i));

}

}

TEST\_METHOD(Clear\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Act

original.clear();

// Assert

Assert::IsTrue(original.isEmpty());

}

TEST\_METHOD(isEmpty\_EmptyTree\_Success)

{

// Arrange

Tree<int> original;

// Assert

Assert::IsTrue(original.isEmpty());

}

TEST\_METHOD(isEmpty\_NotEmptyTree\_Fail)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Assert

Assert::IsFalse(original.isEmpty());

}

TEST\_METHOD(Insert\_UniqueElement\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Act

bool isInserted = original.insert(100);

// Assert

Assert::IsTrue(isInserted);

Assert::IsTrue(original.find(100));

}

TEST\_METHOD(Insert\_RepeatingElement\_Fail)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Act

bool isInserted = original.insert(5);

// Assert

Assert::IsFalse(isInserted);

Assert::IsTrue(original.find(5));

}

TEST\_METHOD(Find\_InsertedElement\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Assert

for (int i = 1; i < 10; ++i)

{

Assert::IsTrue(original.find(i));

}

}

TEST\_METHOD(Find\_NotInsertedElement\_Fail)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Assert

for (int i = 20; i < 40; ++i)

{

Assert::IsFalse(original.find(i));

}

}

TEST\_METHOD(Remove\_InsertedElement\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Act

bool isRemoved = original.remove(5);

// Assert

Assert::IsTrue(isRemoved);

Assert::IsFalse(original.find(5));

}

TEST\_METHOD(Remove\_NotInsertedElement\_Fail)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Act

bool isRemoved = original.remove(100);

// Assert

Assert::IsFalse(isRemoved);

Assert::IsFalse(original.find(100));

}

TEST\_METHOD(toString\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

std::string result = "1 2 3 4 5 6 7 8 9";

// Assert

Assert::AreEqual(original.toString(), result);

}

TEST\_METHOD(printAsTree\_Success)

{

// Arrange

Tree<int> original;

original.insert(5);

original.insert(2);

original.insert(8);

original.insert(1);

original.insert(4);

original.insert(7);

original.insert(9);

std::string a = " 9\n";

std::string b = " 8\n";

std::string c = " 7\n";

std::string d = "5\n";

std::string e = " 4\n";

std::string f = " 2\n";

std::string g = " 1\n";

std::string result = a + b + c + d + e + f + g;

// Act

std::stringstream stream;

original.printAsTree(stream);

// Assert

Assert::AreEqual(stream.str(), result);

}

TEST\_METHOD(Begin\_EmptyTree\_Success)

{

// Arrange

Tree<int> original;

// Assert

Assert::AreEqual(original.begin(), original.end());

}

TEST\_METHOD(Begin\_NotEmptyTree\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Assert

Assert::AreNotEqual(original.begin(), original.end());

}

TEST\_METHOD(Last\_EmptyTree\_Success)

{

// Arrange

Tree<int> original;

// Assert

Assert::AreEqual(original.last(), original.end());

}

TEST\_METHOD(Last\_NotEmptyTree\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Assert

Assert::AreNotEqual(original.last(), original.end());

}

TEST\_METHOD(End\_EmptyTree\_Success)

{

// Arrange

Tree<int> original;

// Assert

Assert::AreEqual(original.end(), original.end());

}

TEST\_METHOD(End\_NotEmptyTree\_Success)

{

// Arrange

Tree<int> original;

for (int i = 1; i < 10; ++i)

{

original.insert(i);

}

// Assert

Assert::AreEqual(original.end(), original.end());

}

};

TEST\_CLASS(IteratorTest)

{

public:

TEST\_METHOD(DefaultConstructor\_Success)

{

// Act

Tree<int>::Iterator a, b;

// Assert

Assert::AreEqual(a, b);

}

TEST\_METHOD(EqualOperator\_EqualEmptyIterators\_Success)

{

// Arrange

Tree<int> original;

// Assert

Assert::IsTrue(original.begin() == original.last());

}

TEST\_METHOD(EqualOperator\_EqualNotEmptyIterators\_Success)

{

// Arrange

Tree<int> original;

original.insert(4);

// Assert

Assert::IsTrue(original.begin() == original.last());

}

TEST\_METHOD(EqualOperator\_NotEqualEmptyIterators\_Fail)

{

// Arrange

Tree<int> original;

original.insert(4);

// Assert

Assert::IsFalse(original.begin() == original.end());

}

TEST\_METHOD(EqualOperator\_NotEqualNotEmptyIterators\_Fail)

{

// Arrange

Tree<int> original;

original.insert(4);

original.insert(5);

original.insert(3);

// Assert

Assert::IsFalse(original.begin() == original.last());

}

TEST\_METHOD(NotEqualOperator\_EqualEmptyIterators\_Fail)

{

// Arrange

Tree<int> original;

// Assert

Assert::IsFalse(original.begin() != original.last());

}

TEST\_METHOD(NotEqualOperator\_EqualNotEmptyIterators\_Fail)

{

// Arrange

Tree<int> original;

original.insert(4);

// Assert

Assert::IsFalse(original.begin() != original.last());

}

TEST\_METHOD(NotEqualOperator\_NotEqualEmptyIterators\_Success)

{

// Arrange

Tree<int> original;

original.insert(4);

// Assert

Assert::IsTrue(original.begin() != original.end());

}

TEST\_METHOD(NotEqualOperator\_NotEqualNotEmptyIterators\_Success)

{

// Arrange

Tree<int> original;

original.insert(4);

original.insert(5);

original.insert(3);

// Assert

Assert::IsTrue(original.begin() != original.last());

}

TEST\_METHOD(PrefixIncrementOperator\_ValidIterator\_Success)

{

// Arrange

Tree<int> original;

for (int i = 10; i >= 5; --i)

{

original.insert(i);

}

for (int i = 0; i < 5; ++i)

{

original.insert(i);

}

// Assert

Tree<int>::Iterator it = original.begin();

for (int i = 1; i <= 10; ++i)

{

Assert::AreEqual(\*(++it), i);

}

}

TEST\_METHOD(PrefixIncrementOperator\_InvalidIterator\_Fail)

{

// Arrange

Tree<int> original;

for (int i = 10; i >= 5; --i)

{

original.insert(i);

}

for (int i = 0; i < 5; ++i)

{

original.insert(i);

}

// Act

Tree<int>::Iterator it = original.begin();

for (int i = 0; i <= 10; ++i)

{

++it;

}

// Создание лямбда-функции для провоцирования исключения

auto callToFail = [&it]() {++it; };

// Assert

Assert::ExpectException<std::out\_of\_range>(callToFail);

}

TEST\_METHOD(PostfixIncrementOperator\_ValidIterator\_Success)

{

// Arrange

Tree<int> original;

for (int i = 10; i >= 5; --i)

{

original.insert(i);

}

for (int i = 0; i < 5; ++i)

{

original.insert(i);

}

// Assert

Tree<int>::Iterator it = original.begin();

for (int i = 0; i <= 10; ++i)

{

Assert::AreEqual(\*(it++), i);

}

}

TEST\_METHOD(PostfixIncrementOperator\_InvalidIterator\_Fail)

{

// Arrange

Tree<int> original;

for (int i = 10; i >= 5; --i)

{

original.insert(i);

}

for (int i = 0; i < 5; ++i)

{

original.insert(i);

}

// Act

Tree<int>::Iterator it = original.begin();

for (int i = 0; i <= 10; ++i)

{

it++;

}

// Создание лямбда-функции для провоцирования исключения

auto callToFail = [&it]() { it++; };

// Assert

Assert::ExpectException<std::out\_of\_range>(callToFail);

}

TEST\_METHOD(PrefixDecrementOperator\_ValidIterator\_Success)

{

// Arrange

Tree<int> original;

for (int i = 10; i >= 5; --i)

{

original.insert(i);

}

for (int i = 0; i < 5; ++i)

{

original.insert(i);

}

// Assert

Tree<int>::Iterator it = original.last();

for (int i = 9; i >= 0; --i)

{

Assert::AreEqual(\*(--it), i);

}

}

TEST\_METHOD(PrefixDecrementOperator\_InvalidIterator\_Fail)

{

// Arrange

Tree<int> original;

for (int i = 10; i >= 5; --i)

{

original.insert(i);

}

for (int i = 0; i < 5; ++i)

{

original.insert(i);

}

// Act

Tree<int>::Iterator it = original.last();

for (int i = 0; i <= 10; ++i)

{

--it;

}

// Создание лямбда-функции для провоцирования исключения

auto callToFail = [&it]() {--it; };

// Assert

Assert::ExpectException<std::out\_of\_range>(callToFail);

}

TEST\_METHOD(PostfixDecrementOperator\_ValidIterator\_Success)

{

// Arrange

Tree<int> original;

for (int i = 10; i >= 5; --i)

{

original.insert(i);

}

for (int i = 0; i < 5; ++i)

{

original.insert(i);

}

// Assert

Tree<int>::Iterator it = original.last();

for (int i = 10; i >= 0; --i)

{

Assert::AreEqual(\*(it--), i);

}

}

TEST\_METHOD(PostfixDecrementOperator\_InvalidIterator\_Fail)

{

// Arrange

Tree<int> original;

for (int i = 10; i >= 5; --i)

{

original.insert(i);

}

for (int i = 0; i < 5; ++i)

{

original.insert(i);

}

// Act

Tree<int>::Iterator it = original.last();

for (int i = 0; i <= 10; ++i)

{

it--;

}

// Создание лямбда-функции для провоцирования исключения

auto callToFail = [&it]() { it--; };

// Assert

Assert::ExpectException<std::out\_of\_range>(callToFail);

}

TEST\_METHOD(DereferenceOperator\_ValidIterator\_Success)

{

// Arrange

Tree<int> original;

for (int i = 10; i >= 5; --i)

{

original.insert(i);

}

for (int i = 0; i < 5; ++i)

{

original.insert(i);

}

// Act

Tree<int>::Iterator it1 = original.begin();

Tree<int>::Iterator it2 = original.last();

while (it1 != it2)

{

++it1;

--it2;

}

// Assert

Assert::AreEqual(\*it1, \*it2);

Assert::AreSame(\*it1, \*it2);

}

TEST\_METHOD(DereferenceOperator\_InvalidIterator\_Fail)

{

// Arrange

Tree<int> original;

// Act

Tree<int>::Iterator it = original.begin();

// Создание лямбда-функции для провоцирования исключения

auto callToFail = [it]() { \*it; };

// Assert

Assert::ExpectException<std::out\_of\_range>(callToFail);

}

};

Приложение 3. Демонстрационный файл

include <iostream>

#include <iomanip>

#include "../AVL\_tree/Tree.hpp"

#define \_CRTDBG\_MAP\_ALLOC

#include <stdlib.h>

#include <crtdbg.h>

//Вывод дерева в поток

template <typename Type>

std::ostream& operator<<(std::ostream& out, const Tree<Type>& tree)

{

out << "TREE = ";

for (Tree<int>::Iterator it = tree.begin(); it != tree.last(); ++it)

{

out << \*it << ", ";

}

if (tree.last() != tree.end())

{

out << \*tree.last() << ". ";

}

return out;

}

//Ввод дерева из потока

template <typename Type>

std::istream& operator>>(std::istream& in, Tree<Type>& tree)

{

tree.clear();

if (&in == &std::cin)

{

std::cout << "Enter how many values do you want to place in the tree?" << std::endl;

}

int count;

in >> count;

if (&in == &std::cin)

{

std::cout << "Enter values:" << std::endl;

}

for (int i = 0; i < count; ++i)

{

Type value;

in >> value;

bool isInserted = tree.insert(value);

if (&in == &std::cin)

{

std::cout << (isInserted ? "Added!" : "Failed!") << std::endl;

}

}

return in;

}

int main()

{

//Блок, в котором "живёт" дерево

{

Tree<int> t;

std::cout << "Options:" << std::endl;

std::cout << "i <number> - insert number to the tree" << std::endl;

std::cout << "f <number> - find number in the tree" << std::endl;

std::cout << "d <number> - delete number from the tree" << std::endl;

std::cout << "r - read" << std::endl;

std::cout << "t - tree-print" << std::endl;

std::cout << "p - print" << std::endl;

std::cout << "e - check iterators" << std::endl;

std::cout << "x - exit" << std::endl;

std::cout << std::endl;

char choice;

std::cout << "Input:" << std::endl;

do

{

std::cin >> choice;

int number;

switch (choice)

{

case 'i':

std::cin >> number;

std::cout << std::setw(6) << std::left << "Insert";

std::cout << std::setw(4) << std::right << number;

std::cout << std::left << ": " << (t.insert(number) ? "SUCCEED" : "FAILED") << std::endl;

break;

case 'f':

std::cin >> number;

std::cout << std::setw(6) << std::left << "Find";

std::cout << std::setw(4) << std::right << number;

std::cout << std::left << ": " << (t.find(number) ? "FOUND" : "NOT FOUND") << std::endl;

break;

case 'd':

std::cin >> number;

std::cout << std::setw(6) << std::left << "Delete";

std::cout << std::setw(4) << std::right << number;

std::cout << std::left << ": " << (t.remove(number) ? "SUCCEED" : "FAILED") << std::endl;

break;

case 'r':

std::cin >> t;

std::cout << "DONE" << std::endl;

break;

case 't':

t.printAsTree(std::cout);

std::cout << "[" << t.toString() << "]" << std::endl;

break;

case 'p':

std::cout << t << std::endl;

break;

case 'e':

for (auto element : t)

{

std::cout << element << " > ";

}

std::cout << "END" << std::endl;

for (Tree<int>::Iterator it = t.last(); it != t.end(); --it)

{

std::cout << \*it << " > ";

}

std::cout << "END" << std::endl;

break;

case 'x':

break;

default:

std::cout << "INVALID INPUT!\a" << std::endl;

}

} while (choice != 'x');

}

//дерево перестало "жить"

//Функция проверки утечки памяти (результат смотреть в MS VS в "Вывод", рядом со "Список ошибок")

\_CrtDumpMemoryLeaks();

return 0;

}