МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Факультет автоматики и вычислительной техники

Кафедра вычислительной техники



**ОТЧЁТ**

**по лабораторной работе №3**

**«Коллекция данных – сбалансированное дерево поиска»**

по дисциплине: «Алгоритмы и структуры данных»

Вариант – 2

Выполнили:Проверила:

студенты гр. АВТ-119

Перцевой Н. Н. Романенко Т.А.

Подкорытов А. Н. «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г.

«\_\_\_» \_\_\_\_\_\_\_\_\_ 2024 г.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(оценка, подпись)

Новосибирск

2024

1. **Цель работы**

Изучение и исследование методов балансировки двоичных деревьев поиска на примере АТД «Сбалансированное дерево поиска». Освоение методики модификации коллекций с помощью механизма наследования классов.

1. **Общее задание**

1. Спроектировать, реализовать и провести тестовые испытания АТД «Сбалансированное-дерево поиска» для коллекции, содержащей данные произвольного типа. Тип данных задается клиентской программой.

АТД «Сбалансированное-дерево поиска» представляет собой модифицированную версию АТД «**BST**-дерево» с трудоемкостью операций

Интерфейс АТД «Сбалансированное-дерево поиска» включает следующие операции:

* конструктор
* конструктор копирования
* деструктор
* опрос размера дерева
* очистка дерева
* проверка дерева на пустоту
* доступ по чтению/записи к данным по ключу
* включение данных с заданным ключом
* удаление данных с заданным ключом
* запрос прямого итератора, установленного на узел дерева с минимальным ключом *begin*()
* запрос обратного итератора, установленного на узел дерева с максимальным ключом *rbegin*()
* запрос «неустановленного» прямого итератора *end*()
* запрос «неустановленного» обратного итератора *rend*()

Операции прямого и обратного итератора:

* операция доступа по чтению и записи к данным текущего узла \*
* операция перехода к следующему (для обратного – к предыдущему) по ключу узлу в дереве ++
* операция перехода к предыдущему (для обратного – к следующему) по ключу узлу в дереве –
* проверка равенства однотипных итераторов ==
* проверка неравенства однотипных итераторов !=

Для тестирования коллекции интерфейс АТД «Сбалансированное дерево поиска» включает дополнительные операции:

* вывод структуры дерева на экран (отображать ключи и параметры балансировки).
* опрос числа узлов дерева, просмотренных предыдущей операцией.

2. Выполнить отладку и тестирование всех операций АТД «Сбалансированное дерево поиска» с помощью меню операций.

3. Выполнить сравнительное тестирование трудоемкости операций вставки, удаления и поиска для коллекций «**BST**-дерево» и «Сбалансированное дерево поиска» для случайной и вырожденной структуры дерева.

4. Выполнить сравнительны анализ теоретических и экспериментальных показателях трудоемкости операций.

5. Составить отчёт по лабораторной работе. Отчёт должен содержать следующие пункты:

1) Титульный лист

2) Цель лабораторной работы

3) Общее задание и вариант задания

4) Формат АТД «Сбалансированное дерево поиска»

5) Формат АТД «Прямой итератор сбалансированного дерева поиска»

6) Формат АТД «Обратный итератор сбалансированного дерева поиска»

7) Справочное определение класса для коллекции «Сбалансированное дерева поиска», предназначенное для клиентской программы.

8) Описание методики тестирования трудоёмкости операций **BST**-дерева и сбалансированного дерева поиска.

9) Таблицы и графики с полученными оценками средней трудоёмкости операций для худшего и среднего случаев функционирования **BST**-дерева и сбалансированного дерева поиска. Должны быть приведены графики трудоемкости для операций поиска, вставки и удаления (графики обеих коллекций совмещены в одной системе координат)

10) Сравнительный анализ теоретических и экспериментальных оценок трудоёмкости для операций **BST**-дерева и сбалансированного дерева поиска.

11) Выводы

12) Список использованной литературы

13) Приложение с текстами программ:

* текст полного определения и методов класса «Сбалансированное дерево поиска».
* текст программы-меню для тестирования отдельных операций сбалансированного дерева поиска.
* Текст программы сравнительного тестирования трудоемкости операций **BST**-дерева и сбалансированного дерева поиска.

# **Вариант**

АВЛ-дерево как модификация **BST**-дерева. Алгоритмы операций вставки, удаления и поиска реализуются в итеративной форме.

# **Формат АТД «АВЛ-дерево»**

АВЛ-дерево представляет собой сбалансированное по высоте двоичное дерево поиска. Для каждой его вершины высота ее двух поддеревьев различается не более чем на 1. Каждый элемент АВЛ-дерева состоит из данных, критерия сбалансированности и уникального ключевого значения, идентифицирующего данные среди прочих в дереве. Положение элемента в дереве определяется ключевым значением при сопоставлении его с другими ключами, присутствующими в дереве.

**ДАННЫЕ:**

Текущий размер дерева **size –** количество элементов в дереве.

Структура данных: двоичное дерево поиска на основе адресных указателей

**ОПЕРАЦИИ:**

*Конструктор:*

**Вход**: нет

**Начальные значения**: нет

**Процесс:** создание пустого дерева

**Постусловие:** создано пустое дерево, **size** = 0

*Конструктор копирования*

**Вход**: ссылка на другое дерево

**Начальные значения**: нет

**Процесс:** копирование данных из входного дерева в текущее

**Постусловие:** создано дерево с данными входного дерева

*Деструктор*

**Вход**: нет

**Предусловие**: нет

**Процесс:** удаление дерева

**Выход:** нет

**Постусловие:** дерево удалено

*Проверка дерева на пустоту*

**Вход**: нет

**Предусловие**: нет

**Процесс:** проверка дерева на пустоту

**Выход:** 1 – дерево пусто, 0 – дерево не пусто

**Постусловие:** нет

*Включение данных с заданным ключом*

**Вход**: ключ **key** и значение данных **data** элемента для включения

**Предусловие**: в дереве не должно быть элемента с ключом **key**

**Процесс:** добавление нового элемента с ключом **key** и значением **data**

**Выход:** 1 – элемент включен успешно, 0 **–** предусловие не выполнено

**Постусловие:** элемент добавлен в дерево, **size**++

*Удаление данных с заданным ключом*

**Вход**: ключ **key** удаляемого элемента

**Предусловие**: нет

**Процесс:** поиск и удаление элемента с заданным ключом

**Выход:** 1 – элемент удален, 0 **–** элемент с заданным ключом не найден

**Постусловие:** элемент удален из дерева, **size**—

*Очистка дерева*

**Вход**: нет

**Предусловие**: нет

**Процесс:** удаление всех узлов дерева

**Выход:** нет

**Постусловие: size** = 0

*Доступ к данным по ключу*

**Вход**: ключ **key** искомого элемента

**Предусловие**: ключ **key** существует в дереве

**Процесс:** поиск и получение данных по ключу **key**

**Выход:** ссылка на значение данных с ключом **key** или исключение, если не выполнено предусловие

**Постусловие:** нет

*Опрос размера дерева*

**Вход**: нет

**Предусловие**: нет

**Процесс:** опрос размера дерева **size**

**Выход:** **size**

**Постусловие:** нет

*Запрос прямого итератора*

**Вход**: нет

**Предусловие**: нет

**Процесс:** установка прямого итератора на элемент с наименьшим значением **key**

**Выход:** прямой итератор, установленный на первый элемент дерева

**Постусловие:** нет

*Запрос «неустановленного» прямого итератора*

**Вход**: нет

**Предусловие**: нет

**Процесс:** установка «неустановленного» прямого итератора

**Выход:** «неустановленный прямой итератор»

**Постусловие:** нет

*Запрос обратного итератора*

**Вход**: нет

**Предусловие**: нет

**Процесс:** установка обратного итератора на элемент с наибольшим значением **key**

**Выход:** обратный итератор, установленный на последний элемент дерева

**Постусловие:** нет

*Запрос «неустановленного» обратного итератора*

**Вход**: нет

**Предусловие**: нет

**Процесс:** установка «неустановленного» обратного итератора

**Выход:** «неустановленный» обратный итератор

**Постусловие:** нет

# **5. Формат АТД «Прямой итератор АВЛ-дерева»**

Прямой итератор – объект, используемый для последовательного доступа к значениям в узлах дерева и для перемещения по дереву. Текущий элемент дерева – это элемент дерева, на который указывает итератор. Возможно перемещение к следующему или предыдущему элементу дерева. Существует 2 состояния итератора: установлен (в пределах дерева) и не установлен (за пределами дерева). Также, итератор не установлен, если дерево пусто.

**ДАННЫЕ:**

Указатель на дерево **owner**

Указатель на текущий элемент дерева **cur**

**ОПЕРАЦИИ:**

*Доступ по чтению и записи к данным текущего узла*

**Вход**: нет

**Предусловие**: итератор установлен

**Процесс:** формирование ссылки на данные

**Выход:** ссылка на данные, исключение при невыполнении предусловия

**Постусловие:** нет

*Операция инкремента для перехода к следующему элементу в дереве*

**Вход**: нет

**Предусловие**: нет

**Процесс:** переход итератора к следующему по ключу элементу дерева

**Выход:** нет

**Постусловие:** итератор установлен на следующий по ключу элемент дерева или переведен в состояние «не установлен»

*Операция декремента для перехода к предыдущему элементу в дереве*

**Вход**: нет

**Предусловие**: нет

**Процесс:** переход итератора к предыдущему по ключу элементу дерева

**Выход:** нет

**Постусловие:** итератор установлен на предыдущий по ключу элемент дерева или переведен в состояние «не установлен»

*Проверка равенства итераторов*

**Вход**: итератор для сравнения

**Предусловие**: нет

**Процесс:** сравнение итераторов по параметрам

**Выход:** 1 – итераторы равны, 0 – итераторы не равны

**Постусловие:** нет

*Проверка неравенства итераторов*

**Вход**: итератор для сравнения

**Предусловие**: нет

**Процесс:** сравнение итераторов по параметрам

**Выход:** 1 – итераторы не равны, 0 – итераторы равны

**Постусловие:** нет

# **6. Формат АТД «Обратный итератор АВЛ-дерева»**

Обратный итератор – объект, используемый для последовательного доступа к значениям в узлах дерева и для перемещения по дереву. Текущий элемент дерева – это элемент дерева, на который указывает итератор. Возможно перемещение к следующему или предыдущему элементу дерева. Существует 2 состояния итератора: установлен (в пределах дерева) и не установлен (за пределами дерева). Также итератор не установлен, если дерево пусто.

**ДАННЫЕ:**

Указатель на дерево **owner**

Указатель на текущий элемент дерева **cur**

**ОПЕРАЦИИ:**

*Доступ по чтению и записи к данным текущего узла*

**Вход**: нет

**Предусловие**: итератор установлен

**Процесс:** формирование ссылки на данные

**Выход:** ссылка на данные, исключение при невыполнении предусловия

**Постусловие:** нет

*Операция инкремента для перехода к предыдущему элементу в дереве*

**Вход**: нет

**Предусловие**: нет

**Процесс:** переход итератора к предыдущему по ключу элементу дерева

**Выход:** нет

**Постусловие:** итератор установлен на предыдущий по ключу элемент дерева или переведен в состояние «не установлен»

*Операция декремента для перехода к следующему элементу в дереве*

**Вход**: нет

**Предусловие**: нет

**Процесс:** переход итератора к следующему по ключу элементу дерева

**Выход:** нет

**Постусловие:** итератор установлен на следующий по ключу элемент дерева или переведен в состояние «не установлен»

*Проверка равенства итераторов*

**Вход**: итератор для сравнения

**Предусловие**: нет

**Процесс:** сравнение итераторов по параметрам

**Выход:** 1 – итераторы равны, 0 – итераторы не равны

**Постусловие:** нет

*Проверка неравенства итераторов*

**Вход**: итератор для сравнения

**Предусловие**: нет

**Процесс:** сравнение итераторов по параметрам

**Выход:** 1 – итераторы не равны, 0 – итераторы равны

**Постусловие:** нет

# **7. Шаблонный класс для коллекции «Сбалансированное дерево поиска»**

template<class K, class T>

class AvlTree : public Tree<K, T>

{

int size; //Количество элементов в дереве

public:

Tree(); //Конструктор

Tree(Tree<K, T>& t); //Конструктор копирования

~Tree(); //Деструктор

bool IsEmpty(); //Проверка дерева на пустоту

bool Insert(K key, T val, double& IA); //Включение данных с заданным ключом

bool Delete(K key, double& DA); //Удаление данных с заданным ключом

void Clear(); //Очистка дерева

T& GetData(K key, double& SA); //Доступ к данным по ключу

int GetSize(); //Опрос размера дерева

class Iterator

{

Tree<K, T>\* owner; //Указатель на дерево

public:

Node\* cur; //Указатель на текущий элемент дерева

T& operator \*(); //Доступ по чтению и записи к данным текущего узла

void operator ++();//Операция инкремента для перехода к следующему

void operator --();//Операция декремента для перехода к предыдущему

bool operator ==(Iterator& iterator); //Проверка равенства итераторов

bool operator !=(Iterator& iterator); //Проверка неравенства итераторов

};

Iterator beg(); //Запрос прямого итератора

Iterator end(); //Запрос «неустановленного» прямого итератора

class ReverseIterator

{

Tree<K, T>\* owner; //Указатель на дерево

public:

Node\* cur; //Указатель на текущий элемент дерева

T& operator \*(); //Доступ по чтению и записи к данным текущего узла

void operator ++();//Операция инкремента для перехода к предыдущему

void operator --();//Операция декремента для перехода к следующему

bool operator ==(ReverseIterator& reviterator); //Проверка равенства

bool operator !=(ReverseIterator& reviterator); //Проверка неравенства

};

ReverseIterator rbeg(); //Запрос обратного итератора

ReverseIterator rend(); //Запрос «неустановленного» обратного итератора

};

# **8. Описание методики сравнительного тестирования трудоёмкости операций BST-дерева и сбалансированного дерева поиска**

Тестирование трудоёмкости операций **BST**-дерева и сбалансированного дерева поиска проводилось для двух случаев: вырожденных и случайно заполненных деревьев

При вызове тестирующей программы деревья заполняются либо случайно сгенерированными ключами, либо в случае вырожденного дерева ключами с шагом 10000. Тип ключей unsigned long long. Далее происходит поток операций вставки, поиска и удаления по ключу, а также формирование счетчиков суммарных трудоемкостей вставки, удаления и поиска.

По завершению потока значения усредняются. На основе этих данных были построены таблицы и графики зависимостей трудоемкости вставки, поиска и удаления по ключу от количества элементов для случайных и вырожденных деревьев.

# **9. Таблицы и графики с полученными оценками средней трудоемкости**

*Табл. 1. Средние трудоёмкости операций вставки, удаления и поиска случайных BST и АВЛ деревьев.*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **BST-дерево** | | | | | **АВЛ-дерево** | | | |
| **N** | **1,39log2N** | **Вставка** | **Удаление** | **Поиск** | **log2N+0.25** | **Вставка** | **Удаление** | **Поиск** |
| **100** | 9,23496 | 7,52 | 7,38 | 7,1 | 6,89386 | 6,8 | 6,76 | 6,28 |
| **1000** | 13,85244 | 13,284 | 12,904 | 12,422 | 10,2158 | 10,082 | 10,118 | 9,174 |
| **3000** | 16,05554 | 14,7193 | 14,6927 | 13,774 | 11,8007 | 11,7053 | 11,7567 | 10,87 |
| **5000** | 17,07992 | 16,4692 | 16,2928 | 15,488 | 12,5377 | 12,4468 | 12,5036 | 11,6144 |
| **10000** | 18,46992 | 18,1704 | 18,0186 | 17,2012 | 13,5377 | 13,4376 | 13,5154 | 12,4534 |

Рис.1 Сравнение результатов операций случайных BST-дерева и АВЛ-дерева.

*Табл. 2. Средние трудоёмкости операций вставки, удаления и поиска вырожденных BST и АВЛ деревьев.*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **BST-дерево** | | | | | **АВЛ-дерево** | | | |
| **N** | **N/2** | **Вставка** | **Удаление** | **Поиск** | **log2N+0.25** | **Вставка** | **Удаление** | **Поиск** |
| **100** | 50 | 37,82 | 38,36 | 38,96 | 6,89386 | 6,6 | 6,84 | 6 |
| **500** | 250 | 193,184 | 203,536 | 203,008 | 9,21578 | 8,888 | 9,036 | 8,208 |
| **1000** | 500 | 419,684 | 396,734 | 398,178 | 10,2158 | 9,888 | 9,938 | 9,126 |
| **1500** | 750 | 612,241 | 587,724 | 609,641 | 10,8007 | 10,4747 | 10,664 | 9,79467 |
| **2000** | 1000 | 787,914 | 805,704 | 798,255 | 11,2158 | 10,845 | 10,952 | 10,072 |

Рис.2 Сравнение результатов операций вырожденных BST-дерева и АВЛ-дерева.

# **10. Сравнительный анализ теоретических и экспериментальных оценок трудоёмкости для операций BST-дерева и сбалансированного дерева поиска**

1. Как можно заметить из рисунков 1 и 2, полученные результаты соответствуют теоретической зависимости, которая определяется как , где n – размер дерева.

2. В ходе сравнительного анализа экспериментальных оценок трудоемкости для различных операций АВЛ-дерева было установлено, что данные оценки практически полностью совпадают друг с другом.

3. Как видно из представленных графиков, АВЛ-дерево выигрывает у BST-дерева в отношении трудоемкости любой операции, однако для АВЛ-дерева операции вставки и удаления сопровождаются дополнительными временными затратами на повороты ветвей дерева.

# **11. Выводы**

В результате проделанной работы был создан абстрактный тип данных – АВЛ-дерево. Оно представляет собой сбалансированное по высоте двоичное дерево поиска. Для каждой его вершины высота ее двух поддеревьев различается не более чем на 1.

Спроектировали, реализовали и провели тестовые испытания АТД «АВЛ-дерево» для коллекции, содержащей данные произвольного типа. Тип данных задается клиентской программой.

Работоспособность класса была проверена с помощью созданной программы-меню, дающей доступ ко всем методам созданного класса. Созданный класс был протестирован на работоспособность, выдав все необходимое результаты без ошибок.

Тестирование трудоёмкости дерева проводилось для двух случаев: вырожденного и случайно заполненного дерева. Был проведён расчёт трудоёмкости для трёх типовых операций на случайно сгенерированном дереве произвольного размера, а также на вырожденном дереве. Результаты были занесены в таблицу и представлены на графиках. Характер зависимостей совпадает с теоретическими данными. Полученные результаты были сопоставлены с трудоемкостями BST-дерева, в ходе сравнительного анализа было выявлено, что АВЛ-дерево имеет преимущество перед BST-деревом в отношении трудоемкости для любой операции.

# **12. Список использованной литературы**

1. Романенко Т.А. Разработка программных коллекций данных: учебное пособие. – Новосибирск: Издательство НГТУ, 2020

# **13. Приложение**

**Полное определение класса и текстов методов класса:**

//

//BST

#pragma once

#include <iostream>

#include <stack>

using namespace std;

class TreeException

{

string code;

public:

TreeException() { code = "Исключение"; }

string What() { return code; }

};

template<typename K, typename T>

class Tree

{

public:

class Node

{

public:

K key;

T data;

int criteria;

Node\* left, \* right;

Node(K k, T d)

{

key = k;

data = d;

criteria = 0;

left = right = NULL;

}

};

Node\* root;

int size;

int cnt;

public:

Tree(); //Конструктор

Tree(Tree<K, T>& t); //Конструктор копирования

~Tree(); //Деструктор

int GetSize(); //Опрос размера дерева

void Clear(); //Очистка дерева

bool IsEmpty(); //Проверка дерева на пустоту

T& GetData(K key); //Чтение данных по ключу

bool Insert(K key, T val);//Включение данных по ключу

bool Delete(K key); //Удаление данных по ключу

void PostorderTraversal(); //Вывод по LRT

void FindCritStab(); //Определение критерия сбалансированности

void Show(); //Вертикальная печать дерева

int getCountNodesView(); //Число узлов, просмотренных пред. операцией

private:

void CopyTree(Node\* r); //Вспомогательная функция для копирования дерева

void \_Postorder(Node\* r); //Вспомогательная функция для обхода дерева

void \_Show(Node\* r, int level); //Вспомогательная функция для вертикальной печати

int \_Crit(Node\* r); //Вспомогательная функция для определения критерия сбалансированности

Node\* \_GetData(K key, Node\* root); //Вспомогательная функция для чтения данных по ключу

Node\* \_Insert(K key, T val, Node\* root, bool\* inserted); //Вспомогательная функция для записи данных по ключу

Node\* \_Del(Node\* targetReplaceWith, Node\* current); //Вспомогательная функция для удаления данных по ключу

Node\* \_Delete(Node\* current, Node\* targetParent, K key, bool\* deleted); //Вспомогательная функция для удаления данных по ключу

Node\* BSTPredecessor(Node\* x); //Ищем узел с предыдущим по значению ключом

Node\* BSTSuccessor(Node\* x); //Ищем узел со следующим по значению ключом

Node\* Max(Node\* t); //Ищем узел с максимальным ключом

Node\* RParent(Node\* t, Node\* x);

Node\* Min(Node\* t); //Ищем узел с минимальным ключом

Node\* LParent(Node\* t, Node\* x);

public:

class Iterator

{

Tree\* ptr;

Tree<K, T>\* owner;

public:

Node\* cur;

Iterator(Tree<K, T>\* owner);

bool is\_off();

T& operator \*();

void operator ++();

void operator --();

bool operator ==(Iterator& iterator);

bool operator !=(Iterator& iterator);

};

Iterator beg();

Iterator end();

class ReverseIterator

{

Tree\* ptr;

Tree<K, T>\* owner;

public:

Node\* cur;

ReverseIterator(Tree<K, T>\* owner);

bool is\_rev\_off();

T& operator \*();

void operator ++();

void operator --();

bool operator ==(ReverseIterator& reviterator);

bool operator !=(ReverseIterator& reviterator);

};

ReverseIterator rbeg();

ReverseIterator rend();

};

//Конструктор

template <class K, class T>

Tree<K, T>::Tree() {

root = NULL;

size = 0;

cnt = 0;

}

//Конструктор копирования

template <class K, class T>

Tree<K, T>::Tree(Tree<K, T>& t)

{

root = NULL;

size = 0;

CopyTree(t.root);

}

//Вставка

template <class K, class T>

typename Tree<K, T>::Node\* Tree<K, T>::\_Insert(K key, T val, Node\* current, bool\* inserted) {

cnt++;

if (current == NULL) {

\*inserted = true;

root = new Node(key, val);

size++;

return root;

}

if (key == current->key) {

\*inserted = false;

return NULL;

}

if (key < current->key && current->left == NULL) {

\*inserted = true;

current->left = new Node(key, val);

size++;

return current->left;

}

else if (key < current->key && current->left != NULL) return \_Insert(key, val, current->left, inserted);

if (key > current->key && current->right == NULL) {

\*inserted = true;

current->right = new Node(key, val);

size++;

return current->right;

}

else if (key > current->key && current->right != NULL) return \_Insert(key, val, current->right, inserted);

return NULL;

}

template <class K, class T>

bool Tree<K, T>::Insert(K key, T val) {

bool inserted = false;

\_Insert(key, val, root, &inserted);

return inserted;

}

//Получение данных по ключу

template<class K, class T>

typename Tree<K, T>::Node\* Tree<K, T>::\_GetData(K key, Node\* current) {

cnt++;

if (current != NULL) {

if (current->key != key) {

if (key < current->key)

return \_GetData(key, current->left);

else

return \_GetData(key, current->right);

}

}

return current;

}

template<class K, class T>

T& Tree<K, T>::GetData(K key) {

Node\* targetNode = \_GetData(key, root);

if (targetNode == NULL) throw TreeException();

return targetNode->data;

}

//Удаление

template <class K, class T>

typename Tree<K, T>::Node\* Tree<K, T>::\_Delete(Node\* current, Node\* targetParent, K key, bool\* deleted) {

if (current == NULL) {

\*deleted = false;

return NULL;

}

cnt++;

if (key < current->key) {

\_Delete(current->left, current, key, deleted);

return current;

}

if (key > current->key) {

\_Delete(current->right, current, key, deleted);

return current;

}

if (\*deleted == true) return current;

\*deleted = true;

size--;

if (current->left == NULL && current->right == NULL) {

if (targetParent == NULL) root = NULL;

else {

if (targetParent->left == current) targetParent->left = NULL;

if (targetParent->right == current) targetParent->right = NULL;

}

delete current;

return NULL;

}

if (current->left == NULL && current->right != NULL) {

Node\* x = current->right;

if (targetParent == NULL) root = current->right;

else {

if (targetParent->left == current) targetParent->left = current->right;

if (targetParent->right == current) targetParent->right = current->right;

}

return x;

}

if (current->right == NULL && current->left != NULL) {

Node\* x = current->left;

if (targetParent == NULL) root = current->left;

else {

if (targetParent->left == current) targetParent->left = current->left;

if (targetParent->right == current) targetParent->right = current->left;

}

return x;

}

current->right = \_Del(current->right, current);

return current;

}

template <class K, class T>

typename Tree<K, T>::Node\* Tree<K, T>::\_Del(Node\* targetReplaceWith, Node\* current) {

cnt++;

if (targetReplaceWith->left != NULL) {

targetReplaceWith->left = \_Del(targetReplaceWith->left, current);

return targetReplaceWith;

}

current->key = targetReplaceWith->key;

current->data = targetReplaceWith->data;

Node\* x = targetReplaceWith->right;

delete targetReplaceWith;

return x;

}

template <class K, class T>

bool Tree<K, T>::Delete(K key) {

bool deleted = false;

\_Delete(root, NULL, key, &deleted);

return deleted;

}

//Вспомогательная функция для копирования дерева

template <class K, class T>

void Tree<K, T>::CopyTree(typename Tree<K, T>::Node\* r)

{

if (r == NULL)

return;

Insert(r->key, r->data);

CopyTree(r->left);

CopyTree(r->right);

}

//Деструктор

template <class K, class T>

Tree<K, T>::~Tree() {

Clear();

}

//Очистка дерева

template <class K, class T>

void Tree<K, T>::Clear()

{

if (IsEmpty()) return;

stack<Node\*> s1, s2;

Node\* t;

s1.push(root);

while (!s1.empty())

{

t = s1.top();

s1.pop();

s2.push(t);

if (t->left != NULL)

s1.push(t->left);

if (t->right != NULL)

s1.push(t->right);

while (!s2.empty())

{

t = s2.top();

s2.pop();

delete t;

size--;

}

}

root = NULL;

size = 0;

}

//Получение размера

template <typename K, typename T>

int Tree<K, T>::GetSize() {

return this->size;

}

//Проверка на пустоту

template <class K, class T>

bool Tree<K, T>::IsEmpty()

{

return (this->size == 0);

}

//Определение критерия стабильности

template <class K, class T>

void Tree<K, T>::FindCritStab()

{

if (root != NULL) \_Crit(root);

else cout << "Дерево пусто";

cout << endl;

}

template <class K, class T>

int Tree<K, T>::\_Crit(typename Tree<K, T>::Node\* r) {

if (r == NULL) return 0;

int left = \_Crit(r->left);

int right = \_Crit(r->right);

int h = ((left > right) ? left : right) + 1;

//1cout << right - left << " ";

r->criteria = right - left;

return h;

}

//Обратный обход

template <class K, class T>

void Tree<K, T>::PostorderTraversal()

{

if (root != NULL) \_Postorder(root);

else cout << "Дерево пусто";

cout << endl;

}

//Вспомогательная функция для вертикальной печати

template <class K, class T>

void Tree<K, T>::\_Postorder(typename Tree<K, T>::Node\* r)

{

if (r == NULL) return;

\_Postorder(r->left);

\_Postorder(r->right);

cout << r->key << " ";

}

template <class K, class T>

void Tree<K, T>::\_Show(typename Tree<K, T>::Node\* r, int level) {

if (r == NULL) return;

\_Show(r->right, level + 1);

for (int i = 0; i < 3 \* level; i++) cout << " ";

cout << r->key << "(" << r->criteria << ")" << endl;

\_Show(r->left, level + 1);

return;

}

//Вертикальная печать

template <class K, class T>

void Tree<K, T>::Show() {

if (root == NULL) {

cout << "Дерево пусто" << endl;

return;

}

\_Show(root, 0);

}

//Функции для итератора

//Узел с предыдущим по значению ключом

template <class K, class T>

typename Tree<K, T>::Node\* Tree<K, T>::BSTPredecessor(typename Tree<K, T>::Node\* x) {

if (x == NULL) return NULL;

if (x->left != NULL) return Max(x->left);

else return RParent(root, x);

}

//Узел с максимальным ключом

template <class K, class T>

typename Tree<K, T>::Node\* Tree<K, T>::Max(typename Tree<K, T>::Node\* t) {

if (t == NULL) return NULL;

while (t->right != NULL) t = t->right;

return t;

}

template <class K, class T>

typename Tree<K, T>::Node\* Tree<K, T>::RParent(typename Tree<K, T>::Node\* t, typename Tree<K, T>::Node\* x) {

if (t == x) return NULL;

if (x->key > t->key) {

Node\* rp = RParent(t->right, x);

if (rp != NULL) return rp;

else return t;

}

else return RParent(t->left, x);

}

//Узел со следующим по значению ключом

template <class K, class T>

typename Tree<K, T>::Node\* Tree<K, T>::BSTSuccessor(typename Tree<K, T>::Node\* x) {

if (x == NULL) return NULL;

if (x->right != NULL) return Min(x->right);

else return LParent(root, x);

}

//Узел с минимальный ключом

template <class K, class T>

typename Tree<K, T>::Node\* Tree<K, T>::Min(typename Tree<K, T>::Node\* t) {

if (t == NULL) return NULL;

while (t->left != NULL) t = t->left;

return t;

}

template <class K, class T>

typename Tree<K, T>::Node\* Tree<K, T>::LParent(typename Tree<K, T>::Node\* t, typename Tree<K, T>::Node\* x) {

if (t == x) return NULL;

if (x->key < t->key)

{

Node\* lp = LParent(t->left, x);

if (lp != NULL) return lp;

else return t;

}

else return LParent(t->right, x);

}

//Получение счетчика пройденных узлов

template <class K, class T>

int Tree<K, T>::getCountNodesView()

{

int temp = cnt;

cnt = 0;

return temp;

}

//Итератор

//Конструктор итератора

template <class K, class T>

Tree<K, T>::Iterator::Iterator(Tree<K, T>\* owner) {

ptr = owner;

cur = NULL;

}

//В начало

template <class K, class T>

typename Tree<K, T>::Iterator Tree<K, T>::beg()

{

Iterator iter = Iterator(this);

Node\* cur = root;

if (cur != nullptr)

while (cur->left != nullptr)

cur = cur->left;

iter.cur = cur;

return iter;

}

//В конец

template <class K, class T>

typename Tree<K, T>::Iterator Tree<K, T>::end()

{

Iterator iter = Iterator(this);

Node\* cur = root;

if (cur != nullptr)

while (cur->right != nullptr)

cur = cur->right;

iter.cur = cur;

return iter;

}

//К предыдущему

template <class K, class T>

void Tree<K, T>::Iterator::operator--() {

cur = ptr->BSTPredecessor(cur);

}

//К следующему

template <class K, class T>

void Tree<K, T>::Iterator::operator++() {

cur = ptr->BSTSuccessor(cur);

}

//Проверка состояния

template <class K, class T>

bool Tree<K, T>::Iterator::is\_off() {

return (cur == NULL);

}

//Доступ к данным текущего

template <class K, class T>

T& Tree<K, T>::Iterator::operator \*() {

if (is\_off())

throw TreeException();

return cur->data;

}

//Проверка на равенство прямых итераторов

template <class K, class T>

bool Tree<K, T>::Iterator::operator ==(Iterator& iterator) {

return cur == iterator.cur;

}

//Проверка на неравенство прямых итераторов

template <class K, class T>

bool Tree<K, T>::Iterator::operator !=(Iterator& iterator) {

return cur != iterator.cur;

}

//Обратный итератор

template <class K, class T>

Tree<K, T>::ReverseIterator::ReverseIterator(Tree<K, T>\* owner) {

ptr = owner;

cur = NULL;

}

//Запрос обратного итератора

template <class K, class T>

typename Tree<K, T>::ReverseIterator Tree<K, T>::rbeg()

{

ReverseIterator iter = ReverseIterator(this);

Node\* cur = root;

if (cur != nullptr)

while (cur->right != nullptr)

cur = cur->right;

iter.cur = cur;

return iter;

}

//запрос "неустановленного" обратного итератора

template <class K, class T>

typename Tree<K, T>::ReverseIterator Tree<K, T>::rend()

{

ReverseIterator iter = ReverseIterator(this);

Node\* cur = root;

if (cur != nullptr)

while (cur->left != nullptr)

cur = cur->left;

iter.cur = cur;

return iter;

}

//К предыдущему

template <class K, class T>

void Tree<K, T>::ReverseIterator::operator--() {

cur = ptr->BSTSuccessor(cur);

}

//К следующему

template <class K, class T>

void Tree<K, T>::ReverseIterator::operator++() {

cur = ptr->BSTPredecessor(cur);

}

//Проверка состояния

template <class K, class T>

bool Tree<K, T>::ReverseIterator::is\_rev\_off() {

return (cur == NULL);

}

//Доступ к данным текущего

template <class K, class T>

T& Tree<K, T>::ReverseIterator::operator \*() {

if (is\_rev\_off())

throw TreeException();

return cur->data;

}

//Проверка на равенство обратных итераторов

template <class K, class T>

bool Tree<K, T>::ReverseIterator::operator ==(ReverseIterator& reviterator) {

return cur == reviterator.cur;

}

//Проверка на неравенство обратных итераторов

template <class K, class T>

bool Tree<K, T>::ReverseIterator::operator !=(ReverseIterator& reviterator) {

return cur != reviterator.cur;

}

//

//AVL

#pragma once

#include <iostream>

#include <stack>

using namespace std;

class AvlTreeException

{

string code;

public:

AvlTreeException() { code = "Исключение"; }

string What() { return code; }

};

template<class K, class T>

class AvlTree : public Tree<K, T>

{

public:

bool Insert(K key, T val, double& IA);//Включение данных по ключу

bool Delete(K key, double& DA); //Удаление данных по ключу

void Show(); //Вертикальная печать дерева

T& GetData(K key, double& SA); //Чтение данных по ключу

private:

void \_Show(typename Tree<K,T>::Node\* r, int level); //Вспомогательная функция для вертикальной печати

typename Tree<K, T>::Node\* \_GetData(K key, typename Tree<K, T>::Node\* root, double& SA); //Вспомогательная функция для чтения данных по ключу

typename Tree<K, T>::Node\* RDel(typename Tree<K, T>::Node\* t, bool& del); //RDel - однократный правый поворот для операции удаления

typename Tree<K, T>::Node\* LDel(typename Tree<K, T>::Node\* t, bool& del); //LDel - однократный левый поворот для операции удаления

typename Tree<K, T>::Node\* RLDel(typename Tree<K, T>::Node\* t, bool& del); //RLDel - двукратный поворот для операции удаления

typename Tree<K, T>::Node\* LRDel(typename Tree<K, T>::Node\* t, bool& del); //LRDel - двукратный поворот для операции удаления

typename Tree<K, T>::Node\* LR(typename Tree<K, T>::Node\* t); //LR-поворот

typename Tree<K, T>::Node\* RL(typename Tree<K, T>::Node\* t); //RL-поворот

};

//Вставка

template <typename K, typename T>

bool AvlTree<K, T>::Insert(K key, T val, double& IA) {

//Вводим два дополнительных стека

//Стек для хранения проиденных элементов

typename Tree<K, T>::Node\*\* node\_stack = new typename Tree<K, T>::Node \* [this->size + 1];

//Стек для хранения направления движения (влево от вершины или вправо)

int\* dir\_stack = new int[this->size + 1];

//Счётчик, отражающий количество элементов в стеках

int dir = -1;

//Элемент для хранения текущего значения

typename Tree<K, T>::Node\* t = this->root;

//Ищем место вставки

while (t)

{

this->cnt++;

IA++;

//Если ключи равны, то очищаем память и и возвращаем false

if (key == t->key) {

delete[] node\_stack;

delete[] dir\_stack;

return false;

}

//Сравниваем ключи и перемещаемся по дереву

if (key > t->key) {

node\_stack[++dir] = t;

dir\_stack[dir] = -1; //-1 - движение вправо

t = t->right;

}

else

{

node\_stack[++dir] = t;

dir\_stack[dir] = 1; //1 означает движение влево

t = t->left;

}

}

//Создаем новый элемент

t = new typename Tree<K, T>::Node(key, val);

//Вставляем элемент

if (!this->root) this->root = t;

else

{

if (dir\_stack[dir] == -1) node\_stack[dir]->right = t;

else node\_stack[dir]->left = t;

}

bool h = true; //Критерий увеличения высоты

while (dir + 1)

{

if (h)

{

//Корректируем значение коэффицента сбалансированности

node\_stack[dir]->criteria += dir\_stack[dir];

typename Tree<K, T>::Node\* buffer = node\_stack[dir];

switch (node\_stack[dir]->criteria)

{

case 0:

h = 0;

break;

case 1:

h = 1;

break;

case -1:

h = 1;

break;

case 2:

node\_stack[dir]->criteria = 1;

if ((node\_stack[dir]->left->criteria >= 0)) buffer =

RDel(node\_stack[dir], h);

else buffer = LR(node\_stack[dir]);

h = 0;

break;

case -2:

node\_stack[dir]->criteria = -1;

if ((node\_stack[dir]->right->criteria <= 0)) buffer =

LDel(node\_stack[dir], h);

else buffer = RL(node\_stack[dir]);

h = 0;

break;

}

//Корректируем общую структуру дерева

if (!dir) this->root = buffer;

else if (dir\_stack[dir - 1] == -1) node\_stack[dir - 1]->right = buffer;

else node\_stack[dir - 1]->left = buffer;

}

dir--;

}

//Очищаем память

delete[] node\_stack;

delete[] dir\_stack;

this->size++;

return true;

}

//Удаление

template <class K, class T>

bool AvlTree<K, T>::Delete(K key, double& DA) {

//Выделяем память под стеки

typename Tree<K, T>::Node\*\* node\_stack = new typename Tree<K, T>::Node \* [this->size];

int\* dir\_stack = new int[this->size];

int top = -1; //Указатель на вершину стека

if (!this->root) return false;

typename Tree<K, T>::Node\* t = this->root;

//Ищем необходимый ключ

while (t)

{

this->cnt++;

DA++;

if (key == t->key) break;

node\_stack[++top] = t;

if (key < t->key) {

dir\_stack[top] = 1; //1 - пошли налево

t = t->left;

}

else {

dir\_stack[top] = -1; //-1 - пошли направо

t = t->right;

}

}

if (!t) return false;

bool isroot = false;

if (t == this->root) isroot = true;

bool del = true;

//Случай, когда удаляемая вершина терминальная

if ((t->right == NULL) && (t->left == NULL)) {

delete t;

if (isroot) this->root = NULL;

else {

if (dir\_stack[top] == -1) node\_stack[top]->right = NULL;

else node\_stack[top]->left = NULL;

}

}

else

//Случай, когда нет левого сына

if ((t->right != NULL) && (t->left == NULL)) {

if (isroot) this->root = t->right;

else {

if (dir\_stack[top] == -1) node\_stack[top]->right = t->right;

else node\_stack[top]->left = t->right;

}

delete t;

}

else

//Случай, когда нет правого сына

if ((t->right == NULL) && (t->left != NULL)) {

if (isroot) this->root = t->left;

else

{

if (dir\_stack[top] == -1) node\_stack[top]->right = t->left;

else node\_stack[top]->left = t->left;

}

delete t;

}

else

//Случай, когда есть оба сына

if ((t->right != NULL) && (t->left != NULL)) {

//У левой вершины нет правого сына

if (t->left->right == NULL) {

this->cnt++;

DA++;

typename Tree<K, T>::Node\* nd = t->left;

nd->right = t->right;

nd->criteria = t->criteria; //Перерасчёт коэффицентов сбалансированности

node\_stack[++top] = nd;

dir\_stack[top] = 1;

if (!top) this->root = nd;

else {

if (dir\_stack[top - 1] == 1) node\_stack[top - 1]->left = nd;

else node\_stack[top - 1]->right = nd;

}

delete t;

}

else {

node\_stack[++top] = t;

dir\_stack[top] = 1;

this->cnt++;

DA++;

typename Tree<K, T>::Node\* nd = t->left;

node\_stack[++top] = nd;

dir\_stack[top] = -1;

this->cnt++;

DA++;

nd = nd->right;

while (nd->right)

{

this->cnt++;

DA++;

node\_stack[++top] = nd;

dir\_stack[top] = -1;

nd = nd->right;

}

t->data = nd->data;

t->key = nd->key;

if (nd->left == NULL) node\_stack[top]->right = NULL;

else node\_stack[top]->right = nd->left;

delete nd;

}

}

this->size--;

while (top + 1)

{

if (del)

{

//Корректируем критерии сбалансированности

node\_stack[top]->criteria -= dir\_stack[top];

typename Tree<K, T>::Node\* buffer = node\_stack[top];

//Теперь анализируем текущее состояние

switch (node\_stack[top]->criteria)

{

case 0:

del = true;

buffer = node\_stack[top];

break;

case -1:

del = false;

buffer = node\_stack[top];

break;

case 1:

del = false;

buffer = node\_stack[top];

break;

case 2: //Разбалансировка узла

node\_stack[top]->criteria = 1;

if ((node\_stack[top]->left->criteria >= 0)) buffer =

RDel(node\_stack[top], del);

else buffer = LRDel(node\_stack[top], del);

break;

case -2: //Разбалансировка узла

node\_stack[top]->criteria = -1;

if ((node\_stack[top]->right->criteria <= 0)) buffer =

LDel(node\_stack[top], del);

else buffer = RLDel(node\_stack[top], del);

break;

}

//Теперь необходимо поддержать общую структуру дерева в целостности

if (!top) this->root = buffer;

else if (dir\_stack[top - 1] == -1) node\_stack[top - 1]->right =

buffer;

else node\_stack[top - 1]->left = buffer;

}

top--;

}

//Очищаем за собой память

delete[] node\_stack;

delete[] dir\_stack;

return true;

}

//Вертикальная печать

template <class K, class T>

void AvlTree<K, T>::Show() {

if (this->root == NULL) {

cout << "Дерево пусто" << endl;

return;

}

\_Show(this->root, 0);

}

template <class K, class T>

void AvlTree<K, T>::\_Show(typename Tree<K, T>::Node\* r, int level) {

if (r == NULL) return;

\_Show(r->right, level + 1);

for (int i = 0; i < 3 \* level; i++) cout << " ";

cout << r->key << "(" << r->criteria << ")" << endl;

\_Show(r->left, level + 1);

return;

}

//Получение данных по ключу

template<class K, class T>

typename Tree<K,T>::Node\* AvlTree<K, T>::\_GetData(K key, typename Tree<K, T>::Node\* current, double& SA) {

while (current != NULL && key != current->key) {

SA++;

this->cnt++;

if (key < current->key)

current = current->left;

else

current = current->right;

}

return current;

}

template<class K, class T>

T& AvlTree<K, T>::GetData(K key, double& SA) {

SA++;

this->cnt++;

typename Tree<K, T>::Node\* targetNode = \_GetData(key, this->root, SA);

if (targetNode == NULL) throw AvlTreeException();

return targetNode->data;

}

//RDel - однократный правый поворот для операции удаления

template <class K, class T>

typename Tree<K,T>::Node\* AvlTree<K, T>::RDel(typename Tree<K,T>::Node\* t, bool& del) {

typename Tree<K, T>::Node\* x = t->left;

t->left = x->right;

x->right = t;

if (x->criteria == 1) {

x->criteria = t->criteria = 0;

del = true;

}

else if (x->criteria == 0) {

x->criteria = -1;

t->criteria = 1;

del = false;

}

return x;

}

//LDel - однократный левый поворот для операции удаления

template <class K, class T>

typename Tree<K,T>::Node\* AvlTree<K, T>::LDel(typename Tree<K, T>::Node\* t, bool& del) {

typename Tree<K, T>::Node\* x = t->right;

t->right = x->left;

x->left = t;

if (x->criteria == -1) {

x->criteria = t->criteria = 0;

del = true;

}

else if (x->criteria == 0) {

x->criteria = 1;

t->criteria = -1;

del = false;

}

return x;

}

//RL-поворот

template <class K, class T>

typename Tree<K,T>::Node\* AvlTree<K, T>::RL(typename Tree<K, T>::Node\* t) {

typename Tree<K, T>::Node\* x = t->right;

typename Tree<K, T>::Node\* y = x->left;

x->left = y->right;

y->right = x;

t->right = y->left;

y->left = t;

if (y->criteria == 0) {

y->criteria = 0;

t->criteria = 0;

x->criteria = 0;

}

else if (y->criteria == 1) {

y->criteria = 0;

t->criteria = 0;

x->criteria = -1;

}

else if (y->criteria == -1) {

y->criteria = 0;

t->criteria = 1;

x->criteria = 0;

}

else {

cout << "Ошибка RL поворота";

}

return y;

}

//LR-поворот

template <class K, class T>

typename Tree<K,T>::Node\* AvlTree<K, T>::LR(typename Tree<K, T>::Node\* t) {

typename Tree<K, T>::Node\* x = t->left;

if ((!x) || (!x->right)) { cout << "Ошибка LR поворота"; }

typename Tree<K, T>::Node\* y = x->right;

x->right = y->left;

y->left = x;

t->left = y->right;

y->right = t;

if (y->criteria == 0) {

y->criteria = 0;

t->criteria = 0;

x->criteria = 0;

}

else if (y->criteria == 1) {

y->criteria = 0;

t->criteria = -1;

x->criteria = 0;

}

else if (y->criteria == -1) {

y->criteria = 0;

t->criteria = 0;

x->criteria = 1;

}

return y;

}

//Двукратный поворот RLDel

template <class K, class T>

typename Tree<K,T>::Node\* AvlTree<K, T>::RLDel(typename Tree<K, T>::Node\* t, bool& del) {

typename Tree<K, T>::Node\* x = t->right;

typename Tree<K, T>::Node\* y = x->left;

t->right = y->left;

y->left = t;

x->left = y->right;

y->right = x;

if (y->criteria == 1) {

x->criteria = -1;

t->criteria = 0;

}

else if (y->criteria == -1) {

x->criteria = 0;

t->criteria = 1;

}

else x->criteria = t->criteria = 0;

y->criteria = 0;

del = true;

return y;

}

//Двукратный поворот LRDel

template <class K, class T>

typename Tree<K,T>::Node\* AvlTree<K, T>::LRDel(typename Tree<K, T>::Node\* t, bool& del) {

typename Tree<K, T>::Node\* x = t->left;

typename Tree<K, T>::Node\* y = x->right;

t->left = y->right;

y->right = t;

x->right = y->left;

y->left = x;

if (y->criteria == 1) {

x->criteria = 0;

t->criteria = -1;

}

else if (y->criteria == -1) {

x->criteria = 1;

t->criteria = 0;

}

else x->criteria = t->criteria = 0;

y->criteria = 0;

del = true;

return y;

}

**Текст программы-меню тестирования отдельных операций АТД:**

#include "BST.h"

#include "AVL.h"

#include <conio.h>

#include <string>

namespace TestEffect

{

void test\_rand(int n);

void test\_sort(int n);

typedef unsigned long long INT\_64;

INT\_64 lrand() { return rand() << 16 | rand(); }

}

double IA = 0;

double DA = 0;

double SA = 0;

//Меню

void main() {

AvlTree<int, int> Avltree;

//AvlTree<int, int> betatree;

//betatree.AvlInsert(10, 10, IA);

//AvlTree<int, int> Avltree(betatree);

AvlTree<int, int>::Iterator it = AvlTree<int, int>::Iterator(&Avltree);;

AvlTree<int, int>::ReverseIterator revit = AvlTree<int, int>::ReverseIterator(&Avltree);

bool useReverseiter = false;

setlocale(LC\_ALL, "RUS");

Avltree.Insert(10, 10, IA); Avltree.Insert(20, 20, IA); Avltree.Insert(30, 30, IA);

Avltree.Insert(40, 40, IA); Avltree.Insert(50, 50, IA);

Avltree.Insert(60, 60, IA); Avltree.Insert(70, 70, IA);

while (true)

{

cout << "1. Вертикальная печать дерева" << endl;

cout << "2. Очистка дерева" << endl;

cout << "3. Проверка на пустоту" << endl;

cout << "4. Доступ к данным с заданным ключом" << endl;

cout << "5. Включение данных с заданным ключом" << endl;

cout << "6. Удаление данных с заданным ключом" << endl;

cout << "7. Опрос размера дерева" << endl;

cout << "8. Меню итераторов" << endl;

cout << "9. Опрос числа просмотренных предыдущей операцией узлов дерева" << endl;

cout << "10.Тестирование трудоемкости для случайного дерева" << endl;

cout << "11.Тестирование трудоемкости для вырожденного дерева" << endl;

cout << "12.Выход" << endl;

int t;

int value, key;

cin >> t;

switch (t)

{

case 1:

cout << "-------------------------------" << endl;

Avltree.Show();

cout << "-------------------------------" << endl << endl;

break;

case 2:

Avltree.Clear();

it = Avltree.beg();

revit = Avltree.rbeg();

cout << endl;

break;

case 3:

cout << Avltree.IsEmpty() << endl << endl;

break;

case 4:

cin >> key;

try

{

char c;

int& datalink = Avltree.GetData(key, SA);

cout << datalink << endl;

cout << "Изменить значение? (y/n)" << endl;

cin >> c;

if (c == 'y') {

cout << "Введите новое значение" << endl;

cin >> value;

datalink = value;

}

}

catch (AvlTreeException e)

{

cerr << e.What() << endl;

}

cout << endl;

break;

case 5:

cin >> key;

cin >> value;

cout << Avltree.Insert(key, value, IA) << endl << endl;

break;

case 6:

cin >> key;

cout << Avltree.Delete(key, DA) << endl << endl;

if (!useReverseiter && !it.is\_off()) it = Avltree.beg();

if (useReverseiter && !revit.is\_rev\_off()) revit = Avltree.rbeg();

break;

case 7:

cout << Avltree.GetSize() << endl << endl;

break;

case 8: {

system("cls");

bool exit = false;

while (!exit) {

cout << "1. Чтение данных текущего узла" << endl;

cout << "2. Запрос прямого итератора" << endl;

cout << "3. Запрос неустановленного прямого итератора" << endl;

cout << "4. Запрос обратного итератора" << endl;

cout << "5. Запрос неустановленного обратного итератора" << endl;

cout << "6. Операция перехода к следующему(для обратного к предыдущему)" << endl;

cout << "7. Операция перехода к предыдущему(для обратного к следующему)" << endl;

cout << "8. Проверка состояния" << endl;

cout << "9. Запись данных в текущий узел" << endl;

cout << "10.Назад" << endl;

int it\_t;

cin >> it\_t;

switch (it\_t) {

case 1:

try

{

if (!useReverseiter) cout << \*it << endl;

else cout << \*revit << endl;

}

catch (AvlTreeException e)

{

cerr << e.What() << endl;

}

cout << endl;

break;

case 2:

it = Avltree.beg();

useReverseiter = false;

cout << endl;

break;

case 3:

it = Avltree.end();

useReverseiter = false;

cout << endl;

break;

case 4:

revit = Avltree.rbeg();

useReverseiter = true;

cout << endl;

break;

case 5:

revit = Avltree.rend();

useReverseiter = true;

cout << endl;

break;

case 6:

try {

if (!useReverseiter) ++it;

else ++revit;

}

catch (AvlTreeException e)

{

cerr << e.What() << endl;

}

cout << endl;

break;

case 7:

try {

if (!useReverseiter) --it;

else --revit;

}

catch (AvlTreeException e)

{

cerr << e.What() << endl;

}

cout << endl;

break;

case 8:

if (!useReverseiter) cout << it.is\_off() << endl;

else cout << revit.is\_rev\_off() << endl;

cout << endl;

break;

case 9:

cin >> value;

try {

if (!useReverseiter) \*it = value;

else \*revit = value;

}

catch (AvlTreeException e)

{

cerr << e.What() << endl;

}

cout << endl;

break;

case 10:

exit = true;

system("cls");

break;

case 777:

if (useReverseiter) useReverseiter = false;

else useReverseiter = true;

cout << endl;

break;

default:

cout << "Неверный номер команды" << endl << endl;

break;

}

}

}

break;

case 9:

cout << Avltree.getCountNodesView() << endl << endl;

break;

case 10:

int n;

cout << "Введите размер дерева:" << endl;

cin >> n;

TestEffect::test\_rand(n);

cout << endl;

break;

case 11:

int n2;

cout << "Введите размер дерева:" << endl;

cin >> n2;

TestEffect::test\_sort(n2);

cout << endl;

break;

case 12:

exit(0);

break;

default:

cout << "Неверный номер команды" << endl << endl;

}

}

}

//Программа тестирования случайного дерева

void TestEffect::test\_rand(int n)

{

Tree<INT\_64, int> BSTtree;

AvlTree<INT\_64, int> AVLtree;

INT\_64\* m = new INT\_64[n];

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

{

m[i] = lrand();

BSTtree.Insert(m[i], 1);

AVLtree.Insert(m[i], 1, IA);

}

BSTtree.getCountNodesView();

IA = 0;

DA = 0;

SA = 0;

double I = 0;

double D = 0;

double S = 0;

for (int i = 0; i < n / 2; i++)

if (i % 10 == 0)

{

BSTtree.Delete(lrand());

AVLtree.Delete(lrand(), DA);

D += BSTtree.getCountNodesView();

BSTtree.Insert(m[rand() % n], 1);

AVLtree.Insert(m[rand() % n], 1, IA);

I += BSTtree.getCountNodesView();

try

{

BSTtree.GetData(m[rand() % n]);

S += BSTtree.getCountNodesView();

}

catch (TreeException&) { S += BSTtree.getCountNodesView(); }

try

{

AVLtree.GetData(m[rand() % n], SA);

}

catch (AvlTreeException&) { SA += AVLtree.getCountNodesView(); };

}

else

{

int ind = rand() % n;

BSTtree.Delete(m[ind]);

AVLtree.Delete(m[ind], DA);

D += BSTtree.getCountNodesView();

INT\_64 key = lrand();

BSTtree.Insert(key, 1);

AVLtree.Insert(key, 1, IA);

I += BSTtree.getCountNodesView();

m[ind] = key;

try {

BSTtree.GetData(m[rand() % n]);

S += BSTtree.getCountNodesView();

}

catch (TreeException&) { S += BSTtree.getCountNodesView(); }

try

{

AVLtree.GetData(m[rand() % n], SA);

}

catch (AvlTreeException&) { SA += AVLtree.getCountNodesView(); };

}

cout << "AVL-Дерево" << endl;

cout << "log2(n)+0.25=" << (log((double)n) / log(2.0)) + 0.25 << endl;

cout << "Размер дерева после:" << AVLtree.GetSize() << endl;

cout << "Трудоемкость вставки: " << IA / (n / 2) << endl;

cout << "Трудоемкость удаления: " << DA / (n / 2) << endl;

cout << "Трудоемкость поиска: " << SA / (n / 2) << endl;

cout << "BST-дерево" << endl;

cout << "1.39\*log2(n)=" << (log((double)n) / log(2.0)) \* 1.39 << endl;

cout << "Размер дерева после:" << BSTtree.GetSize() << endl;

cout << "Трудоемкость вставки: " << I / (n / 2) << endl;

cout << "Трудоемкость удаления: " << D / (n / 2) << endl;

cout << "Трудоемкость поиска: " << S / (n / 2) << endl;

delete[] m;

}

//Программа тестирования вырожденного дерева

void TestEffect::test\_sort(int n)

{

Tree<INT\_64, int> BSTtree;

AvlTree<INT\_64, int> AVLtree;

INT\_64\* m = new INT\_64[n];

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

{

m[i] = i \* 10000;

if (n <= 2000) BSTtree.Insert(m[i], 1);

AVLtree.Insert(m[i], 1, IA);

}

BSTtree.getCountNodesView();

IA = 0;

DA = 0;

SA = 0;

double I = 0;

double D = 0;

double S = 0;

double SAtemp = 0;

for (int i = 0; i < n / 2; i++)

if (i % 10 == 0)

{

int k = lrand() % (10000 \* n);

k = k + !(k % 2);

if (n <= 2000) BSTtree.Delete(k);

AVLtree.Delete(k, DA);

D += BSTtree.getCountNodesView();

if (n <= 2000) BSTtree.Insert(m[rand() % n], 1);

AVLtree.Insert(m[rand() % n], 1, IA);

I += BSTtree.getCountNodesView();

k = lrand() % (10000 \* n);

k = k + !(k % 2);

try

{

if (n <= 2000) BSTtree.GetData(m[rand() % n]);

S += BSTtree.getCountNodesView();

}

catch (TreeException&) { S += BSTtree.getCountNodesView(); }

try

{

AVLtree.GetData(m[rand() % n], SA);

}

catch (AvlTreeException&) { SA += AVLtree.getCountNodesView(); };

}

else

{

int ind = rand() % n;

if (n <= 2000) BSTtree.Delete(m[ind]);

AVLtree.Delete(m[ind], DA);

D += BSTtree.getCountNodesView();

int k = lrand() % (10000 \* n);

k = k + k % 2;

if (n <= 2000) BSTtree.Insert(k, 1);

AVLtree.Insert(k, 1, IA);

I += BSTtree.getCountNodesView();

m[ind] = k;

try {

if (n <= 2000) BSTtree.GetData(m[rand() % n]);

S += BSTtree.getCountNodesView();

}

catch (TreeException&) { S += BSTtree.getCountNodesView(); }

try

{

AVLtree.GetData(m[rand() % n], SA);

}

catch (AvlTreeException&) { SA += AVLtree.getCountNodesView(); };

}

cout << "AVL-Дерево" << endl;

cout << "log2(n)+0.25=" << (log((double)n) / log(2.0)) + 0.25 << endl;

cout << "Размер дерева после:" << AVLtree.GetSize() << endl;

cout << "Трудоемкость вставки: " << IA / (n / 2) << endl;

cout << "Трудоемкость удаления: " << DA / (n / 2) << endl;

cout << "Трудоемкость поиска: " << SA / (n / 2) << endl;

cout << "BST-дерево" << endl;

cout << "n/2=" << n / 2 << endl;

cout << "Размер дерева после:" << BSTtree.GetSize() << endl;

cout << "Трудоемкость вставки: " << I / (n / 2) << endl;

cout << "Трудоемкость удаления: " << D / (n / 2) << endl;

cout << "Трудоемкость поиска: " << S / (n / 2) << endl;

delete[] m;

}