МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра вычислительной техники

Лабораторная работа №2

по дисциплине

«Структуры и алгоритмы обработки данных»

**«Коллекция данных – дерево поиска»**

Группа: АВТ-610

Вариант: 7

Студенты: Дунаев Н.Ю., Князев Р.И.

Преподаватель: Романенко Т.А.

Новосибирск

2018

# Цели лабораторной работы:

1. Освоение технологии реализации ассоциативных нелинейных коллекций на примере АТД «Двоичное дерево поиска».
2. Освоение методики программирования рекурсивных и итеративных алгоритмов для структуры данных.

# Задание к лабораторной работе:

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

Двоичное дерево поиска (Binary Search Tree – BST) представляет упорядоченное, иерархическое, ассоциативное множество элементов, между которыми существуют структурные отношения «предки – потомки».

Интерфейс АТД «**BST**-дерево» включает следующие операции:

* опрос размера дерева;
* очистка дерева;
* проверка дерева на пустоту;
* доступ к данным с заданным ключом;
* включение данных с заданным ключом;
* удаление данных с заданным ключом;
* обход узлов дерева по схеме, заданной в варианте задания;
* дополнительная операция, заданная в варианте задания;
* итератор для доступа к данным в дереве с операциями:

– установка на первый узел в дереве с минимальным ключом,

– установка на последний узел в дереве с максимальным ключом,

– проверка состояния итератора,

– доступ по чтению и записи к данным текущего узла в дереве,

– переход к следующему по значению ключа узлу в дереве,

– переход к предыдущему по значению ключа узлу в дереве.

Для тестирования коллекции интерфейс АТД «**BST**-дерево» включает дополнительные операции:

* вывод структуры дерева на экран;
* опрос числа просмотренных операцией узлов дерева.

1. Выполнить отладку и тестирование всех операций АТД «**BST**-дерево» с помощью меню операций.
2. Выполнить тестирование средней трудоемкости операций поиска, вставки и удаления элементов для вырожденного и случайного **BST**-дерева.
3. Провести анализ экспериментальных показателей трудоемкости операций.

# Вариант задания:

* Алгоритмы операций АТД реализуются в рекурсивной форме.
* Схема операции обхода: ***t → Lt → Rt.***
* Дополнительная операция: определение критерия сбалансированности для всех узлов дерева (итеративная форма алгоритма).

# Формат АТД “Двоичное дерево поиска”

Двоичное дерево поиска представляет собой упорядоченное, иерархическое, ассоциативное множество элементов, между которыми существуют структурные отношения «предки-потомки». Каждый элемент ассоциативного множества состоит из данных и уникального ключевого значения, идентифицирующего данные среди прочих в множестве. Положение элемента в дереве определяется ключевым значением при сопоставлении его с другими ключами, присутствующими в дереве.

**Данные:**

*Size* - количество элементов, находящихся на текущий момент в дереве.

**Операции:**

**Конструктор**

*Вход***:** нет

*Начальные значения:* нет

*Процесс***:** создание пустого дерева

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

**Деструктор**

*Вход***:** нет

*Начальные значения:* нет

*Процесс***:** очистка дерево

*Постусловия***:** дерево очищено, *size* = 0

**Конструктор копирования**

*Вход***:** ссылка на копируемое дерево

*Предусловие***:** нет

*Процесс***:** копирование дерева

*Выход***:** нет

*Постусловия***:** создана копия дерева

**Опрос размера дерева**

*Вход***:** нет

*Предусловие***:** нет

*Процесс***:** опрос размера

*Выход***:** *size*

*Постусловия***:** нет

**Очистка дерева**

*Вход***:** нет

*Предусловие***:** нет

*Процесс***:** удаление элементов дерева

*Выход***:** нет

*Постусловия***:** удалены все элементы, *size* = 0

**Проверка дерева на пустоту**

*Вход***:** нет

*Предусловие***:** нет

*Процесс***:** проверка на пустоту

*Выход***:** *true*, если дерево пусто, иначе *false*

*Постусловия***:** нет

**Доступ к данным с заданным ключом**

*Вход***:** заданный ключ *key*

*Предусловие***:** существование узла с заданным ключом *key* в дереве

*Процесс***:** поиск в дереве узла по заданному ключу *key*

*Выход***:** данные узла с заданным ключом *key*; если предусловие не выполнено – генерация исключения

*Постусловия***:** нет

**Включение данных с заданным ключом**

*Вход***:** заданный ключ *key*, значение включаемого объекта *data*

*Предусловие***:** ключ *key* отсутствует в дереве

*Процесс***:** включение значения *data* в дерево по заданному ключу *key*

*Выход***:** true – значение включено в дерево, false – предусловие не выполнено-значение не добавлено

*Постусловия***:** добавлен элемент с ключом *key* и данными *data.* Размер дерева увеличен на 1

**Удаление данных с заданным ключом**

*Вход***:** заданный ключ *key*

*Предусловие***:** наличие узла в дереве с заданным ключом *key*

*Процесс***:** удаление узла из дерева по заданному ключу *key*

*Выход***:** true – значение удалено, false – предусловие не выполнено-значение не удалено

*Постусловия***:** удален элемент с ключом *key*, размер дерева уменьшен на 1

**Обход узлов дерева по схеме**

*Вход***:** нет

*Предусловие***:** нет

*Процесс***:** обход дерева по заданной схеме t → Lt → Rt,печать узлов.

*Выход***:** нет

*Постусловия***:** нет

**Определение критерия сбалансинованности для всех узлов дерева**

*Вход***:** нет

*Предусловие***:** нет

*Процесс***:** определение критерия сбалансированности для всех узлов дерева

*Выход***:** нет

*Постусловия***:** нет

**Конец АТД Tree**

# Формат АТД “Итератор”

Итератором является объект, обеспечивающий последовательный доступ к данным элементов дерева для чтения и записи.

Имеется возможность установки итератора на первый узел дерева, последний узел дерева, перемещение к следующему или предыдущему элементу дерева.

Существует 2 состояния итератора: «установлен» (в пределах дерева) и «не установлен» (за пределами дерева). Итератор принимает состояние «не установлен» если дерево пустое.

**Данные:**

* Указатель на дерево *tree*
* Указатель на узел *cur*

**Операции:**

**Конструктор**

*Вход***:** указатель на объект АТД «Двоичное дерево поиска»

*Начальные значения***:** нет

*Процесс***:** привязка итератора к заданному дереву

*Постусловия***:** создан объект итератора

**Установка итератора на первый узел дерева**

*Вход***:** нет

*Предусловие***:** нет

*Процесс***:** установка итератора на первый узел дерева

*Выход***:** нет

*Постусловия***:** итератор находится в состоянии «установлен», если установлен на первый узел дерева; если дерево пусто, итератор принимает состояние «не установлен»

**Установка итератора на последний узел дерева**

*Вход***:** нет

*Предусловие***:** нет

*Процесс***:** установка итератора на последний узел дерева

*Выход***:** нет

*Постусловия***:** итератор находится в состоянии «установлен», если установлен на последний узел дерева; если дерево пусто, итератор принимает состояние «не установлен»

**Установка на следующий узел**

*Вход***:** нет

*Предусловие***:** итератор находится в состоянии «установлен»

*Процесс***:** перевод итератора на следующий узел дерева

*Выход***:** true – итератор принимает состояние «установлен», false – не установлен

*Постусловия***:** установка итератора на следующий узел дерева

**Установка на предыдущий узел**

*Вход***:** нет

*Предусловие***:** итератор находится в состоянии «установлен»

*Процесс***:** перевод итератора на предыдущий узел дерева

*Выход***:** true – итератор принимает состояние «установлен», false – не установлен

*Постусловия***:** установка итератора на предыдущий узел дерева

**Проверка состояния итератора**

*Вход***:** нет

*Предусловие***:** нет

*Процесс***:** проверка состояния итератора

*Выход***:** true – если итератор находится в состоянии «не установлен», false - « установлен»

*Постусловия***:** нет

**Доступ по чтению и записи к текущему значению**

*Вход***:** нет

*Предусловие***:** итератор находится в состоянии «установлен»

*Процесс***:** получение ссылки на текущий узел дерева

*Выход***:** ссылка на текущий узел дерева; генерация сообщения об ошибке при невыполнении предусловия

*Постусловия***:** нет

**Конец АТД**

**Определение шаблонного класса**

template <class Data, class Key> class Tree

{

class Iterator

{

public:

Node \*cur; // Указатель на текущий узел

Tree<Data, Key> \*tree; // Указатель на дерево

Iterator(Tree<Data, Key> \*tree); // Конструктор

~Iterator(); // Деструктор

bool begin(); // Установка на первый узел

bool end(); // Установка на поледний узел

bool isOff(); // Проверка состояния

bool next(); // Установка на следующий узел

bool prev(); // Установка на предыдущий узел

Data &operator \*(); // Чтение и запись

};

Tree(); // Конструктор

Tree(const Tree<Data, Key>& tr); // Конструктор копирования

~Tree(); // Деструктор

int getSize(); // Получение размера массива

void clear(); // Очистка дерева

bool isEmpty(); // Проверка на пустоту

Data getData(Node \*root, Key key); // Поиск по ключу

bool insert(Key key, Data data, Node \*root); // Вставка

Node\* deleteByKey(Node \*root, Key key); // Удаление по ключу

void preOrderTravers(Node \*root); // прямой обход узлов

void checkBalance(Node \*root); // Проверка критерия сбалансированности

void print(Node \*root, int level); // Печать

int getOperations(); // Опрос числа просмотренных операцией узлов дерева

private:

int size; // Размер дерева

};

**Описание методики тестирования операций дерева**

Тестирование трудоёмкости дерева проводилось для двух случаев: вырожденного и случайно заполненного дерева. Драйвер тестирования задает модель потока операций гипотетической клиентской программы. Поток состоит из чередующихся равновероятных операций поиска, вставки и удаления данных. Общее количество операций пропорционально размеру коллекции. Для каждого вида операций задается вероятность промаха, равная 10%. Для операций поиска или удаления промахом считается поиск или удаление ключа, отсутствующего в коллекции, а для операции вставки – вставка уже существующего ключа.

Полученные значения количества шагов при просмотре дерева были занесены в переменные-счетчики, а затем усреднены. Усредненные значения счетчиков для различных значений размеров списка были занесены в таблицу.

По значениям в таблице построены графики зависимости трудоемкости вставки, удаления и поиска от количества элементов в дереве.

**Тестирование средней трудоемкости операций поиска, вставки и удаления элементов для вырожденного BST-дерева.**

# *Таблица 1. Полученные оценки трудоемкости операций* *для вырожденного BST-дерева*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | Вставка | Удаление | Поиск | Теория |
| 100 | 52 | 57 | 54 | 50 |
| 500 | 217 | 240 | 237 | 250 |
| 1000 | 442 | 484 | 481 | 500 |
| 1500 | 652 | 721 | 718 | 750 |
| 2000 | 830 | 914 | 911 | 1000 |
| 2500 | 1041 | 1143 | 1140 | 1250 |
| 3000 | 1239 | 1367 | 1364 | 1500 |

*Рис. 1*. Сравнение трудоемкости алгоритмов для вырожденного BST-дерева

**Тестирование средней трудоемкости операций поиска, вставки и удаления элементов для случайного BST-дерева.**

# *Таблица 2. Полученные оценки трудоемкости операций* *для случайного BST-дерева*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | Вставка | Удаление | Поиск | Теория |
| 100 | 7,20 | 7,70 | 5,98 | 9,20 |
| 500 | 10,70 | 11,30 | 9,33 | 12,50 |
| 1000 | 12,21 | 12,89 | 10,76 | 13,85 |
| 1500 | 13,19 | 13,92 | 11,79 | 14,66 |
| 2000 | 13,43 | 14,18 | 12,24 | 15,24 |
| 2500 | 13,76 | 14,42 | 12,46 | 15,68 |
| 3000 | 14,26 | 14,89 | 12,95 | 16,06 |

*Рис. 2.* Сравнение трудоемкости алгоритмов для случайного BST-дерева

Экспериментальные показатели трудоемкости операций вставки, удаления и поиска для случайного и вырожденного BST-деревьев совпадают с теоретическими.

Из первого графика видно, что операция поиска обладает наименьшей трудоемкостью. Причина в том, что операция поиска останавливается, после того как узел с заданным ключом найден, в то время как операция удаления после нахождения заданного узла в случае наличия потомков, пойдет дальше вглубь до конца ветви дерева. Так же очевидно, что операция вставки обладает наибольшей трудоемкостью. Это объясняется тем, что при вставке алгоритм всегда доходит до конца ветви.

**Выводы:**

В ходе лабораторной работы была реализована и протестирована программа, демонстрирующая особенности работы с АТД «BST-дерево» для коллекции, содержащей данные произвольного типа.

Тестирование трудоёмкости дерева проводилось для двух случаев: вырожденного и случайно заполненного дерева. Был проведён расчёт трудоёмкости для трёх типовых операций на дереве произвольного размера, составленном из случайных чисел. Результаты были занесены в таблицу и представлены на графиках. Экспериментальные показатели трудоемкости операций вставки, удаления и поиска для случайного и вырожденного BST-деревьев совпадают с теоретическими.

**Реализация методов на языке С++.**

Файл Tree.h

#include "Except.h"

#include <stack>

#include <math.h>

using namespace std;

template <class Data, class Key> class Tree

{

private:

class Node

{

public:

Data data; // Значение

Key key; // Ключ

int balance; // Критерий сбалансированности

Node \*left; // Указатель на левый узел

Node \*right; // Указатель на правый узел

Node(); // Конструктор без параметров

~Node(); // Деструктор

Node(Data dat, Key k); // Конструктор с параметрами

};

Node \*root; // Указатель на корень

int operations; // Количество операций (для тестирования)

int size; // Размер

void privateCheckBalance(Node \*root);

Data privateGetData(Node \*root, Key key);

bool privateInsert(Key key, Data data, Node \*root);

Node \*privateDeleteByKey(Node \*root, Key key, bool \*del);

void privatePreOrderTravers(Node \*root);

void privatePrint(Node \*root, int level);

Node\* copy(Node \*root); // Копирование

Node\* findPrevKey(Node \*localRoot, Node \*node); // Рекурсивный поиск меньшего ключа

Node\* findNextKey(Node \*localRoot, Node \*node); // Рекурсивный поиск большего ключа

public:

class Iterator

{

public:

Node \*cur; // Указатель на текущий узел

Tree<Data, Key> \*tree; // Указатель на дерево

Iterator(Tree<Data, Key> \*tree); // Конструктор

~Iterator(); // Деструктор

bool begin(); // Установка на первый узел

bool end(); // Установка на поледний узел

bool isOff(); // Проверка состояния

bool next(); // Установка на следующий узел

bool prev(); // Установка на предыдущий узел

Data &operator \*(); // Чтение и запись

};

friend class Iterator;

Tree(); // Конструктор

Tree(const Tree<Data, Key>& tr); // Конструктор копирования

~Tree(); // Деструктор

int getSize(); // Получение размера массива

void clear(); // Очистка

bool isEmpty(); // Проверка на пустоту

Data getData(Key key); // Поиск по ключу

bool insert(Key key, Data data); // Вставка

bool deleteByKey(Key key); // Удаление по ключу

void preOrderTravers(); // прямой обход узлов

void checkBalance(); // Проверка критерия сбалансированности

void print(); // Печать

int getOperations(); // Опрос числа просмотренных операцией узлов дерева

};

//---------------------------------------------------------------------------------------------- Методы списка

template<class Data, class Key>

int Tree<Data, Key>::getOperations()

{

return operations;

}

template<class Data, class Key>

void Tree<Data, Key>::clear()

{

if (root == NULL)

return;

else

{

deleteByKey(root->key);

clear();

}

}

template<class Data, class Key>

bool Tree<Data, Key>::isEmpty()

{

if (size == 0)

return true;

else

return false;

}

template<class Data, class Key>

void Tree<Data, Key>::privatePrint(Node \*root, int level)

{

if (root == NULL)

return;

privatePrint(root->right, level + 1);

for (int i = 0; i < 5 \* level; i++)

cout << " ";

cout << root->key << "," << root->data << "(" << root->balance << ")" << endl;

privatePrint(root->left, level + 1);

}

template<class Data, class Key>

void Tree<Data, Key>::print()

{

privatePrint(root, 0);

}

template<class Data, class Key>

bool Tree<Data, Key>::deleteByKey(Key k)

{

bool \*deleted = new bool;

\*deleted = false;

operations = 0;

root = privateDeleteByKey(root, k, deleted);

return \*deleted;

}

template<class Data, class Key>

typename Tree<Data, Key>::Node\* Tree<Data, Key>::privateDeleteByKey(Node \*root, Key k, bool \*deleted)

{

if (root == NULL)

{

\*deleted = false;

return NULL;

}

operations++;

if (root->key > k)

root->left = privateDeleteByKey(root->left, k, deleted);

else if (root->key < k)

root->right = privateDeleteByKey(root->right, k, deleted);

else

{

if (root->left == NULL && root->right == NULL)

{

root = NULL;

size--;

\*deleted = true;

}

else if (root->right == NULL)

{

Node \*tmp = root;

root = root->left;

tmp = NULL;

size--;

\*deleted = true;

}

else if (root->left == NULL)

{

Node \*tmp = root;

root = root->right;

tmp = NULL;

size--;

\*deleted = true;

}

else

{

Node \*tmp = root->left;

while (tmp->right != NULL)

{

tmp = tmp->right;

operations++;

}

root->key = tmp->key;

root->data = tmp->data;

root->left = privateDeleteByKey(root->left, root->key, deleted);

}

}

return root;

}

template<class Data, class Key>

typename Tree<Data, Key>::Node\* Tree<Data, Key>::findNextKey(Node \*localRoot, Node \*node)

{

if (localRoot == node || localRoot == NULL)

return NULL;

if (node->key < localRoot->key)

{

Node \*tmp = findNextKey(localRoot->left, node);

if (tmp != NULL)

return tmp;

else

return localRoot;

}

else

return findNextKey(localRoot->right, node);

}

template<class Data, class Key>

typename Tree<Data, Key>::Node\* Tree<Data, Key>::copy(Node \*root)

{

if (root == NULL)

return NULL;

Node \*newRoot = new Node(root->data, root->key);

newRoot->left = copy(newRoot->left);

newRoot->right = copy(newRoot->right);

return newRoot;

}

template<class Data, class Key>

typename Tree<Data, Key>::Node\* Tree<Data, Key>::findPrevKey(Node \*localRoot, Node \*node)

{

if (localRoot == node || localRoot == NULL)

return NULL;

if (node->key > localRoot->key)

{

Node \*tmp = findPrevKey(localRoot->right, node);

if (tmp != NULL)

return tmp;

else

return localRoot;

}

else

return findPrevKey(localRoot->left, node);

}

template<class Data, class Key>

bool Tree<Data, Key>::privateInsert(Key k, Data dat, Node \*root)

{

if (k < 0)

return false;

operations++;

if (size == 0)

{

this->root = new Node(dat, k);

size++;

return true;

}

else if (root->key == k)

return false;

else if (root->key > k)

{

if (root->left == NULL)

{

root->left = new Node(dat, k);

size++;

return true;

}

else

return privateInsert(k, dat, root->left);

}

else if (root->key < k)

{

if (root->right == NULL)

{

root->right = new Node(dat, k);

size++;

return true;

}

else

return privateInsert(k, dat, root->right);

}

}

template<class Data, class Key>

bool Tree<Data, Key>::insert(Key k, Data dat)

{

operations = 0;

return privateInsert(k, dat, root);

}

template<class Data, class Key>

void Tree<Data, Key>::privateCheckBalance(Node \*root)

{

if (root == NULL)

return;

int counter = 0;

stack <Node\*> s1, s2;

s1.push(root);

while (!s1.empty())

{

root = s1.top();

s1.pop();

s2.push(root);

if (root->left != NULL)

s1.push(root->left);

if (root->right != NULL)

s1.push(root->right);

}

while (!s2.empty())

{

counter = 1;

root = s2.top();

s2.pop();

s1.push(root);

if (root->left == NULL && root->right == NULL)

root->balance = 1;

if (s2.empty())

continue;

while (s2.top()->left == root || s2.top()->right == root)

{

counter++;

root = s2.top();

s1.push(root);

if (root->left == NULL || root->right == NULL)

root->balance = counter;

else if (root->left->balance > root->right->balance)

root->balance = root->left->balance + 1;

else

root->balance = root->right->balance + 1;

s2.pop();

if (s2.empty())

break;

}

}

while (!s1.empty())

{

root = s1.top();

s1.pop();

if (root->left == NULL && root->right == NULL)

root->balance = 0;

else if (root->left == NULL && root->right != NULL)

root->balance = -root->right->balance;

else if (root->left != NULL && root->right == NULL)

root->balance = root->left->balance;

else

root->balance = root->left->balance - root->right->balance;

}

}

template<class Data, class Key>

void Tree<Data, Key>::checkBalance()

{

privateCheckBalance(root);

}

template<class Data, class Key>

void Tree<Data, Key>::privatePreOrderTravers(Node \*root)

{

if (root != NULL)

{

printf("%d ", root->key);

privatePreOrderTravers(root->left);

privatePreOrderTravers(root->right);

}

}

template<class Data, class Key>

void Tree<Data, Key>::preOrderTravers()

{

privatePreOrderTravers(root);

}

template<class Data, class Key>

int Tree<Data, Key>::getSize()

{

return size;

}

template<class Data, class Key>

Data Tree<Data, Key>::privateGetData(Node\* root, Key k)

{

if (root == NULL)

throw new TreeIsEmpty;

if (root->key == k)

return root->data;

operations++;

if (root->key >= k)

return privateGetData(root->left, k);

else

return privateGetData(root->right, k);

}

template<class Data, class Key>

Data Tree<Data, Key>::getData(Key k)

{

operations = 0;

return privateGetData(root, k);

}

//---------------------------------------------------------------------------------------------- Итератор

template<class Data, class Key>

bool Tree<Data, Key>::Iterator::next()

{

if (tree->root == NULL || cur == NULL)

return 0;

if (cur->right == NULL)

cur = tree->findNextKey(tree->root, cur);

else

{

cur = cur->right;

while (cur->left != NULL)

cur = cur->left;

}

return 1;

}

template<class Data, class Key>

bool Tree<Data, Key>::Iterator::prev()

{

if (tree->root == NULL || cur == NULL)

return 0;

if (cur->left == NULL)

cur = tree->findPrevKey(tree->root, cur);

else

{

cur = cur->left;

while (cur->right != NULL)

cur = cur->right;

}

return 1;

}

template<class Data, class Key>

bool Tree<Data, Key>::Iterator::begin()

{

if (tree->root == NULL)

{

cur = NULL;

return false;

}

else

{

cur = tree->root;

while (cur->left != NULL)

cur = cur->left;

return true;

}

}

template<class Data, class Key>

bool Tree<Data, Key>::Iterator::end()

{

if (tree->root == NULL)

{

cur = NULL;

return false;

}

else

{

cur = tree->root;

while (cur->right != NULL)

cur = cur->right;

return true;

}

}

template<class Data, class Key>

bool Tree<Data, Key>::Iterator::isOff()

{

if (cur == NULL)

return true;

else

return false;

}

template<class Data, class Key>

Data& Tree<Data, Key>::Iterator::operator\*()

{

if (cur == NULL)

throw new IterOutsideTheTree;

else

return cur->data;

}

//---------------------------------------------------------------------------------------------- Конструкторы, деструкторы

template<class Data, class Key>

Tree<Data, Key>::Tree()

{

root = NULL;

size = 0;

operations = 0;

}

template<class Data, class Key>

Tree<Data, Key>::Tree(const Tree<Data, Key>& tr)

{

root = NULL;

size = 0;

root = copy(tr->root);

}

template<class Data, class Key>

Tree<Data, Key>::~Tree()

{

clear();

}

template<class Data, class Key>

Tree<Data, Key>::Node::Node()

{

data = NULL;

key = NULL;

left = NULL;

right = NULL;

balance = 0;

}

template<class Data, class Key>

Tree<Data, Key>::Node::Node(Data dat, Key k)

{

data = dat;

key = k;

left = NULL;

right = NULL;

balance = 0;

}

template<class Data, class Key>

Tree<Data, Key>::Node::~Node()

{}

template<class Data, class Key>

Tree<Data, Key>::Iterator::Iterator(Tree<Data, Key> \*tre)

{

tree = tre;

cur = NULL;

}

template<class Data, class Key>

Tree<Data, Key>::Iterator::~Iterator()

{

tree = NULL;

cur = NULL;

}

Файл except.h

#include <conio.h>

#include <cstdlib>

#include <iostream>

using namespace std;

class TreeError

{

public:

virtual void Error() = 0;

};

class TreeIsEmpty : public TreeError

{

void Error()

{

cout << "Exception" << endl;

\_getch();

}

};

class IncorrectPosition : public TreeError

{

void Error()

{

cout << "Exception" << endl;

\_getch();

}

};

class IterOutsideTheTree : public TreeError

{

void Error()

{

cout << "Exception" << endl;

\_getch();

}

};

class KeyNotFound : public TreeError

{

void Error()

{

cout << "Exception" << endl;

\_getch();

}

};

Файл main.cpp

#include "Tree.h"

#include <time.h>

using namespace std;

void printMenu()

{

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \n";

cout << "0. Меню итератора \n";

cout << "1. Опрос размера дерева \n";

cout << "2. Очистка дерева \n";

cout << "3. Проверка дерева на пустоту \n";

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

cout << "5. Включение данных с заданным ключом \n";

cout << "6. Удаление данных с заданным ключом\n";

cout << "7. Обход узлов (t,L,R) \n";

cout << "8. Расчет критерия сбалансированности \n";

cout << "9. Печать \n";

cout << "10. Тестирование \n";

cout << "15. Выход \n";

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \n" << endl;

}

void printIterMenu()

{

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \n";

cout << "1. Итератор на начало списка \n";

cout << "2. Итератор на конец списка \n";

cout << "3. Следующий узел \n";

cout << "4. Предыдущий узел \n";

cout << "5. Чтение \n";

cout << "6. Запись \n";

cout << "7. Проверка состояния \n";

cout << "8. Выход \n";

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \n";

}

long long longRandom()

{

return rand() << 16 | rand();

}

void main()

{

setlocale(LC\_ALL, "Russian");

srand(time(NULL));

Tree<int, int> \*tree = new Tree<int, int>;

Tree<int, int>::Iterator iter(tree);

long long \*buf;

int menu = 0, iterMenu = 0, key = 0, data = 0, size = 0;

double insertCounter = 0.0, deleteCounter = 0.0, searchCounter = 0.0;

bool flagIter = true;

long long r = 0;

while (true)

{

printMenu();

cin >> menu;

switch (menu)

{

case 0:

flagIter = true;

while (flagIter)

{

printIterMenu();

cin >> iterMenu;

switch (iterMenu)

{

case 1:

cout << iter.begin() << endl;

\_getch();

break;

case 2:

cout << iter.end() << endl;

\_getch();

break;

case 3:

cout << iter.next() << endl;

\_getch();

break;

case 4:

cout << iter.prev() << endl;

\_getch();

break;

case 5:

try

{

cout << \*iter << endl;

}

catch (TreeError \*error){ error->Error(); }

\_getch();

break;

case 6:

cout << "Введите значение " << endl;

try

{

cin >> \*iter;

}

catch (TreeError \*error){ error->Error(); }

\_getch();

break;

case 7:

cout << iter.isOff() << endl;

\_getch();

break;

case 8:

flagIter = false;

break;

}

}

break;

case 1:

cout << "Размер: " << tree->getSize() << endl;

\_getch();

break;

case 2:

iter.cur = NULL;

tree->clear();

cout << "Дерево удалено" << endl;

\_getch();

break;

case 3:

cout << tree->isEmpty() << endl;

\_getch();

break;

case 4:

cout << "Введите ключ" << endl;

cin >> key;

try

{

cout << "Данные: " << tree->getData(key) << endl;

}

catch (TreeError \*error){ error->Error(); }

\_getch();

break;

case 5:

cout << "Введите ключ " << endl;

cin >> key;

cout << "Введите данные" << endl;

cin >> data;

cout << tree->insert(key, data) << endl;

\_getch();

break;

case 6:

cout << "Введите ключ " << endl;

cin >> key;

cout << tree->deleteByKey(key) << endl;

break;

case 7:

tree->preOrderTravers();

cout << endl;

\_getch();

break;

case 8:

tree->checkBalance();

break;

case 9:

tree->print();

\_getch();

break;

case 10:

searchCounter = 0;

deleteCounter = 0;

insertCounter = 0;

cout << "Размер дерева: " << endl;

cin >> size;

buf = new long long[size];

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

{

buf[i] = longRandom();

tree->insert(buf[i], buf[i]);

}

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

{

if (i % 10 != 0)

{

r = buf[rand() % size];

tree->operations = 0;

try

{

tree->getData(r);

}

catch (TreeError \*error){}

searchCounter += tree->getOperations();

tree->operations = 0;

tree->deleteByKey(r);

deleteCounter += tree->getOperations();

tree->operations = 0;

tree->insert(r, r);

insertCounter += tree->getOperations();

}

else

{

r = longRandom();

tree->operations = 0;

try

{

tree->getData(r);

}

catch (TreeError \*error){}

searchCounter += tree->getOperations();

tree->operations = 0;

tree->deleteByKey(r);

deleteCounter += tree->getOperations();

tree->operations = 0;

r = buf[rand() % size];

tree->insert(r, r);

insertCounter += tree->getOperations();

r = longRandom();

}

}

cout << "Теоретическая трудоемкость для операций: " << 1.39\*(log((double)size) / log((double)2)) << endl;

cout << "Вставка: " << (insertCounter / (size / 2)) << endl;

cout << "Удаление: " << deleteCounter / (size / 2) << endl;

cout << "Поиск: " << searchCounter / (size / 2) << endl;

cout << "Размер: " << tree->getSize() << endl << endl;

tree->clear();

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

{

r = i \* 10000;

tree->insert(r, r);

buf[i] = r;

}

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

{

if (i % 10 != 0)

{

r = buf[rand() % size];

tree->operations = 0;

try

{

tree->getData(r);

}

catch (TreeError \*error){}

searchCounter += tree->getOperations();

tree->operations = 0;

tree->deleteByKey(r);

deleteCounter += tree->getOperations();

tree->operations = 0;

tree->insert(r, r);

insertCounter += tree->getOperations();

}

else

{

r = longRandom();

tree->operations = 0;

try

{

tree->getData(r);

}

catch (TreeError \*error){}

searchCounter += tree->getOperations();

tree->operations = 0;

tree->deleteByKey(r);

deleteCounter += tree->getOperations();

tree->operations = 0;

r = buf[rand() % size];

tree->insert(r, r);

insertCounter += tree->getOperations();

r = longRandom();

}

}

cout << "Теоретическая трудоемкость для операций: " << size / 2 << endl;

cout << "Вставка: " << (insertCounter / (size / 2)) << endl;

cout << "Удаление: " << deleteCounter / (size / 2) << endl;

cout << "Поиск: " << searchCounter / (size / 2) << endl;

cout << "Размер: " << tree->getSize() << endl << endl;

tree->clear();

break;

case 15:

return;

}

}

system("pause");

}