|  |  |
| --- | --- |
| **Российский университет транспорта (МИИТ) Институт транспортной техники и систем управления Кафедра «Управление и защита информации»** | |
| **Отчет по практическому заданию по теме «Структуры данных» по дисциплине «Системы управления базами данных»** | |
|  | Выполнил:  Студенты группы ТКИ-441  Порхун Д.Д.  Мацко А.А. |
|  | Проверил:  Доцент кафедры УиЗИ к.т.н., доцент  Васильева М. А. |
| Москва 2023 | |

Оглавление

[1. Условие задачи 3](#_Toc153553501)

[2. Текст программы на языке С++ 3](#_Toc153553502)

[2.1. Node.h 3](#_Toc153553506)

[2.2. Tree.h 4](#_Toc153553507)

[2.3. Main.cpp 13](#_Toc153553508)

[2.4. Test.cpp 14](#_Toc153553509)

[3. Результат работы программы 17](#_Toc153553510)

[4. UML диаграмма классов 18](#_Toc153553511)

1. Условие задачи

**AVL дерево**

Определить класс «Node», включающий следующую информацию:

* Данные любого типа;
* Указатели на поддеревья и родителя;

Определить класс «BST», включающий следующую информацию:

* Указатель на корень дерева;

В программе предусмотреть:

* создание БДП;
* вывод на экран;
* удаление элемента по ключу;
* вставка элемента по ключу;
* поиск данных элемента по ключу;
* поиск минимального и максимального элемента.

1. Текст программы на языке С++

4. 1. Node.h

#pragma once

template<typename T>

struct Node {

T data;

Node<T>\* left;

Node<T>\* right;

Node<T>\* parent;

int height;

Node(T data) : data(data), left(nullptr), right(nullptr), parent(nullptr), height(0) {};

};

* 1. Tree.h

#pragma once

#include <iostream>

#include <sstream>

#include "Node.h"

template<typename T>

class Tree {

private:

Node<T>\* root;

Node<T>\* insert(T x, Node<T>\* t){

if (t == nullptr)

t = new Node<T>(x);

else if (x < t->data) {

t->left = insert(x, t->left);

if (height(t->left) - height(t->right) == 2) {

if (x < t->left->data)

t = singleRightRotate(t);

else

t = doubleRightRotate(t);

}

}

else if (x > t->data) {

t->right = insert(x, t->right);

if (height(t->right) - height(t->left) == 2) {

if (x > t->right->data)

t = singleLeftRotate(t);

else

t = doubleLeftRotate(t);

}

}

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

return t;

};

Node<T>\* singleRightRotate(Node<T>\*& t){

Node<T>\* u = t->left;

t->left = u->right;

u->right = t;

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

u->height = std::max(height(u->left), t->height) + 1;

return u;

};

Node<T>\* singleLeftRotate(Node<T>\*& t){

Node<T>\* u = t->right;

t->right = u->left;

u->left = t;

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

u->height = std::max(height(t->right), t->height) + 1;

return u;

};

Node<T>\* doubleLeftRotate(Node<T>\*& t){

t->right = singleRightRotate(t->right);

return singleLeftRotate(t);

};

Node<T>\* doubleRightRotate(Node<T>\*& t){

t->left = singleLeftRotate(t->left);

return singleRightRotate(t);

};

Node<T>\* findMin(Node<T>\* t){

if (t == nullptr)

return nullptr;

else if (t->left == nullptr)

return t;

else

return findMin(t->left);

};

Node<T>\* findMax(Node<T>\* t){

if (t == nullptr)

return nullptr;

else if (t->right == nullptr)

return t;

else

return findMax(t->right);

};

Node<T>\* remove(T x, Node<T>\* t){

Node<T>\* temp;

if (t == nullptr)

return nullptr;

else if (x < t->data)

t->left = remove(x, t->left);

else if (x > t->data)

t->right = remove(x, t->right);

else if (t->left && t->right) {

temp = findMin(t->right);

t->data = temp->data;

t->right = remove(t->data, t->right);

}

else {

temp = t;

if (t->left == nullptr)

t = t->right;

else if (t->right == nullptr)

t = t->left;

delete temp;

}

if (t == nullptr)

return t;

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

if (height(t->left) - height(t->right) == 2) {

if (height(t->left->left) - height(t->left->right) == 1)

return singleLeftRotate(t);

else

return doubleLeftRotate(t);

}

else if (height(t->right) - height(t->left) == 2) {

if (height(t->right->right) - height(t->right->left) == 1)

return singleRightRotate(t);

else

return doubleRightRotate(t);

}

return t;

};

bool find(T x, Node<T>\* t){

if (t == nullptr)

return false;

else if (x == t->data)

return true;

else if (x <= t->data)

find(x, root->left);

else

find(x, root->right);

};

int height(Node<T>\* t){

return (t == nullptr ? -1 : t->height);

};

int getBalance(Node<T>\* t){

if (t == nullptr)

return 0;

else

return height(t->left) - height(t->right);

};

std::string inorder(Node<T>\* t, std::ostringstream& out){

if (t != nullptr) {

inorder(t->left, out);

out << t->data << " ";

inorder(t->right, out);

}

return out.str();

};

public:

Tree(){ root = nullptr; };

void insert(T x){ root = insert(x, root); };

void remove(T x){ root = remove(x, root); };

std::string display(){

std::ostringstream buffer{};

inorder(root, buffer);

return buffer.str();

};

bool find(T x){ return find(x, root); };

};

* 1. Main.cpp

#include "Tree.h"

int main(){

return 0;

}

* 1. Test.cpp

#include "pch.h"

#include "..\..\..\..\Desktop\avl\Tree.h"

#include <string>

TEST(test, Testing\_TEST) {

EXPECT\_EQ(1, 1);

EXPECT\_TRUE(true);

}

TEST(test\_INT, InsertTest\_INT) {

Tree<int>\* tree = new Tree<int>();

tree->insert(3);

ASSERT\_TRUE(tree->find(3));

}

TEST(test\_INT, RemoveTest\_INT) {

Tree<int>\* tree = new Tree<int>();

tree->insert(3);

tree->remove(3);

ASSERT\_FALSE(tree->find(3));

}

TEST(test\_INT, Size\_Int\_Success) {

Tree<int>\* tree = new Tree<int>();

tree->insert(3);

tree->insert(1);

ASSERT\_EQ( tree->height() , 1);

}

TEST(test\_INT, FindTest\_INT\_True) {

Tree<int>\* tree = new Tree<int>();

tree->insert(3);

tree->insert(20);

tree->insert(1);

tree->insert(23);

tree->insert(50);

ASSERT\_TRUE(tree->find(23));

}

TEST(test\_INT, DisplayTest\_INT) {

Tree<int>\* tree = new Tree<int>();

tree->insert(3);

tree->insert(23);

tree->insert(50);

ASSERT\_EQ(tree->display(),"3 23 50 ");

}

TEST(test\_DOBUBLE, InsertTest\_DOBUBLE) {

Tree<double>\* tree = new Tree<double>();

tree->insert(3.4);

ASSERT\_TRUE(tree->find(3.4));

}

TEST(test\_DOBUBLE, RemoveTest\_DOBUBLE) {

Tree<double>\* tree = new Tree<double>();

tree->insert(2.1);

tree->remove(2.1);

ASSERT\_FALSE(tree->find(2.1));

}

TEST(test\_DOBUBLE, Size\_Double\_Success) {

Tree<double>\* tree = new Tree<double>();

tree->insert(3.4);

tree->insert(2.1);

ASSERT\_EQ(tree->height(), 1);

}

TEST(test\_DOBUBLE, FindTest\_DOBUBLE\_True) {

Tree<double>\* tree = new Tree<double>();

tree->insert(3.1);

tree->insert(2.3);

tree->insert(12.7);

tree->insert(23.0);

tree->insert(50.2);

ASSERT\_TRUE(tree->find(3.1));

}

TEST(test\_DOBUBLE, DisplayTest\_DOBUBLE) {

Tree<double>\* tree = new Tree<double>();

tree->insert(3.4);

tree->insert(12.7);

tree->insert(23.2);

ASSERT\_EQ(tree->display(), "3.4 12.7 23.2 ");

}

TEST(test\_STRING, InsertTest\_STRING) {

Tree<std::string>\* tree = new Tree<std::string>();

tree->insert("3.4");

ASSERT\_TRUE(tree->find("3.4"));

}

TEST(test\_STRING, RemoveTest\_STRING) {

Tree<std::string>\* tree = new Tree<std::string>();

tree->insert("2");

tree->remove("2");

ASSERT\_FALSE(tree->find("2"));

}

TEST(test\_STRING, Size\_String\_Success) {

Tree<std::string>\* tree = new Tree<std::string>();

tree->insert("3.4");

tree->insert("2");

ASSERT\_EQ(tree->height(), 1);

}

TEST(test\_STRING, FindTest\_STRING\_True) {

Tree<std::string>\* tree = new Tree<std::string>();

tree->insert("3.1");

tree->insert("2.3");

tree->insert("12.7");

tree->insert("23.0");

tree->insert("50.2");

ASSERT\_TRUE(tree->find("12.7"));

}

TEST(test\_STRING, DisplayTest\_STRING) {

Tree<std::string>\* tree = new Tree<std::string>();

tree->insert("3.4");

tree->insert("3");

tree->insert("2.3");

ASSERT\_EQ(tree->display(), "2.3 3 3.4 ");

}

1. Результат работы программы

|  |
| --- |
|  |
| 1. Результат тестов |

1. UML диаграмма классов

|  |
| --- |
|  |
| 1. Uml диаграмма классов |