БУ ВО Ханты-Мансийского автономного округа – Югры

«Сургутский государственный университет»

Политехнический институт

Кафедра автоматики и компьютерных систем

ОТЧЁТ

по лабораторной работе №5

по дисциплине: «Алгоритмы и структуры данных»

Выполнил: студент(ка) группы №609-31,

Гаврилов Е.Е.

Принял: старший преподаватель кафедры АиКС

Назаров Е.В.

Сургут

2025г.

**Цель работы:** изучить базовые алгоритмы работы с деревьями: построение, обход, поиска элемента, удаление элемента, подсчет количества узлов, нахождение высоты дерева, исследовать свойства деревьев, закрепить навыки структурного программирования.

**Задание:**

1. Реализовать функции вставки, поиска и удаления узла, обхода дерева, вставки в корень, вывода дерева на экран, нахождения высоты дерева и количества узлов, а также функцию вставки, строящую рандомизированное дерево.

2. Построить зависимости высоты деревьев (обычного бинарного поиска и рандомизированного) от количества ключей (ключи – случайные и упорядоченные величины), полагая, что ключи целые числа.

3. Реализовать заданную функцию (в соответствии с вариантом: T – тип ключей, D – диапазон изменения значений ключей).

4. Составить отчет, в котором привести графики полученных зависимостей, анализ свойств алгоритмов, листинги функций вставки, поиска, удаления узла с комментариями и выводы по работе.

**Вариант 3.**

Тип ключа: int;

Диапазон изменений значений ключей: [0; 1000];

Функция для подсчета количества нечетных чисел в узлах, имеющих ровно 2 поддерева.

Ход работы:

1. Реализовать класс Node, представляющий собой узел дерева, хранящий данные, имеющий указатели на два дочерних узла и один родительский и приватный набор методов для добавления, удаления, поиска узла, поиска количества дочерних узлов, поиска минимального в правом поддереве, добавления в корень, случайного добавления.

2. Реализовать класс BinaryTree, содержащий указатель на корень дерева и предоставляющий интерфейс для взаимодействия с деревом.

3. Реализовать функцию по варианту.

**Исследование работы алгоритмов**

Таблица 1. Высота обычного бинарного поиска от количества ключей (для случайных и упорядоченных величин);

|  |  |  |
| --- | --- | --- |
| Количество ключей | Высота дерева | |
| Случайные величины | Упорядоченные величины |
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 10 | 6 | 10 |
| 100 | 13 | 100 |
| 1000 | 23 | 1000 |
| 2000 | 24 | 2000 |
| 3000 | 26 | 3000 |
| 4000 | 27 | 4000 |
| 5000 | 29 | 5000 |

Таблица 2. Высота рандомизированного дерева поиска от количества ключей (для случайных и упорядоченных величин).

|  |  |  |
| --- | --- | --- |
| Количество ключей | Высота дерева | |
| Случайные величины | Упорядоченные величины |
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 10 | 5 | 6 |
| 100 | 14 | 11 |
| 1000 | 20 | 19 |
| 2000 | 25 | 23 |
| 3000 | 26 | 24 |
| 4000 | 28 | 25 |
| 5000 | 31 | 25 |

Рис. 1. Графики зависимости высоты дерева от количества ключей для обычного и рандомизированного дерева (ключи — случайные целые числа в диапазоне от 0 до 1999).

Рис. 2. График зависимости высоты дерева от количества ключей для обычного дерева (ключи — возрастающая последовательность целых чисел).

Рис. 3. График зависимости высоты дерева от количества ключей для рарндомизированного дерева (ключи — возрастающая последовательность целых чисел).

**Анализ**

На при вставке случайных чисел оба дерева показывают схожую высоту дерева. Небольшой разброс объясняется взятием для вставки случайных значений.

При вставке возрастающей последовательности высота обычного дерева росла линейна, так как все новые значения добавлялись только в одно поддерево, дерево по сути представляло собой связный список. Высота рандомизированного дерева была схожей с высотой при добавлении случайных значений.

**Вывод:** была разработана система классов для реализации бинарного дерева поиска и рандомизированного бинарного дерева поиска; функции для взаимодействием с деревом; исследованы свойства деревьев.

Листинг 1. binarytree.h

#pragma once

#include <cstdio>

#include <format>

#include <iostream>

#include <random>

template <typename T> class BinaryTree;

template <typename T> class Node {

friend class BinaryTree<T>;

private:

T data;

Node \*left;

Node \*right;

Node \*parent;

Node(T data, Node \*left, Node \*right);

Node(T data);

~Node();

void Add(Node<T> \*new\_node);

void Clear();

int Higth();

void Print();

void FPrint(int layer);

Node<T> \*Find(T element);

int NumberOfChildren();

Node<T> \*FindRightMin();

void AddRoot(Node<T> \*new\_node, BinaryTree<T> \*tree);

void AddRandom(Node<T> \*new\_node, BinaryTree<T> \*tree);

int Fun();

};

template <typename T> class BinaryTree {

public:

Node<T> \*root;

BinaryTree();

BinaryTree(Node<T> \*);

~BinaryTree();

void Insert(Node<T> \*new\_node);

void Insert(T new\_element);

void FPrint();

void Clear();

int Higth();

Node<T> \*Find(T element);

int NumberOfNodes();

int Fun();

void Delete(Node<T> \*node);

void Delete(T element);

void LNRTravel();

void LeftRotate(Node<T> \*);

void RightRotate(Node<T> \*);

void InsertRoot(Node<T> \*new\_node);

void InsertRoot(T new\_element);

void InsertRandom(Node<T> \*new\_node);

void InsertRandom(T new\_element);

};

#include "binarytree.tpp"

Листинг 2. bimarytree.tpp

#include "binarytree.h"

using namespace std;

template <typename T>

Node<T>::Node(T data, Node<T> \*left, Node<T> \*right)

: data(data), left(left), right(right), parent(nullptr) {}

template <typename T> Node<T>::Node(T data) : Node<T>(data, nullptr, nullptr) {}

template <typename T> Node<T>::~Node() {}

template <typename T> void Node<T>::Add(Node<T> \*new\_node) {

if (nullptr == new\_node)

return;

new\_node->parent = this;

if (new\_node->data < data) {

if (nullptr == left)

left = new\_node;

else

left->Add(new\_node);

} else {

if (nullptr == right)

right = new\_node;

else

right->Add(new\_node);

}

}

template <typename T> void Node<T>::Clear() {

if (nullptr != left) {

left->Clear();

delete (left);

left = nullptr;

}

if (nullptr != right) {

right->Clear();

delete (right);

right = nullptr;

}

}

template <typename T> int Node<T>::Higth() {

int left\_hight = 1, right\_hight = 1;

if (nullptr != left) {

left\_hight += left->Higth();

}

if (nullptr != right) {

right\_hight += right->Higth();

}

return max(left\_hight, right\_hight);

}

template <typename T> void Node<T>::FPrint(int layer) {

if (nullptr == right)

cout << data;

else

cout << std::format("{:-<5}", data);

if (nullptr != right) {

cout << "---";

right->FPrint(layer + 1);

}

if (nullptr != left) {

cout << endl;

for (int i = 0; i < layer; i++) {

for (int j = 0; j < 8; j++) {

// if (j==layer) cout<<"|";

cout << " ";

}

}

cout << "`-------";

left->FPrint(layer + 1);

}

}

template <typename T> void Node<T>::Print() {

if (nullptr != left)

left->Print();

cout << data << " ";

if (nullptr != right)

right->Print();

}

template <typename T> void BinaryTree<T>::LNRTravel() {

if (nullptr == root)

return;

root->Print();

}

template <typename T>

BinaryTree<T>::BinaryTree(Node<T> \*root) : root(root){}

template <typename T> BinaryTree<T>::BinaryTree() : root(nullptr) {}

template <typename T> BinaryTree<T>::~BinaryTree() { Clear(); }

template <typename T> void BinaryTree<T>::Insert(Node<T> \*new\_node) {

if (nullptr == root) {

root = new\_node;

root->parent = nullptr;

} else

root->Add(new\_node);

}

template <typename T> void BinaryTree<T>::Insert(T new\_element) {

Insert(new Node<T>(new\_element));

}

template <typename T> void BinaryTree<T>::FPrint() {

if (nullptr != root) {

root->FPrint(0);

} else {

cout << "Empty tree" << endl;

}

}

template <typename T> void BinaryTree<T>::Clear() {

if (nullptr != root) {

root->Clear();

delete root;

root = nullptr;

}

}

template <typename T> int BinaryTree<T>::Higth() {

int result = 0;

if (nullptr != root) {

result += root->Higth();

}

return result;

}

template <typename T> Node<T> \*Node<T>::Find(T element) {

Node<T> \*result = nullptr;

if (data == element)

result = this;

else if (element < data) {

if (nullptr != left)

result = left->Find(element);

} else if (nullptr != right)

result = right->Find(element);

return result;

}

template <typename T> Node<T> \*BinaryTree<T>::Find(T element) {

if (nullptr != root)

return root->Find(element);

else

return nullptr;

}

template <typename T> int Node<T>::NumberOfChildren() {

int result = 0;

if (nullptr != left) {

result += left->NumberOfChildren() + 1;

}

if (nullptr != right) {

result += right->NumberOfChildren() + 1;

}

return result;

}

template <typename T> int BinaryTree<T>::NumberOfNodes() {

if (nullptr != root) {

return root->NumberOfChildren() + 1;

} else

return 0;

}

template <typename T> Node<T> \*Node<T>::FindRightMin() {

Node<T> \*node\_ptr = right;

while (node\_ptr->left)

node\_ptr = node\_ptr->left;

return node\_ptr;

}

template <typename T> void BinaryTree<T>::Delete(Node<T> \*node) {

if (nullptr != node->left && nullptr != node->right) {

Node<T> \*right\_min = node->FindRightMin();

right\_min->parent = node->parent;

if (root == node)

root = right\_min;

if (nullptr != node->parent)

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

node->parent->left = right\_min;

else

node->parent->right = right\_min;

right\_min->left = node->left;

if (nullptr != right\_min->left)

right\_min->left->parent = right\_min;

} else {

Node<T> \*child = nullptr;

if (nullptr != node->left)

child = node->left;

else if (nullptr != node->right)

child = node->right;

if (nullptr != child)

child->parent = node->parent;

if (nullptr != node->parent) {

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

node->parent->left = child;

else {

node->parent->right = child;

}

}

}

delete (node);

}

template <typename T> void BinaryTree<T>::Delete(T element) {

Node<T> \*node\_to\_delete = Find(element);

if (nullptr != node\_to\_delete)

Delete(node\_to\_delete);

}

template <typename T> void BinaryTree<T>::LeftRotate(Node<T> \*node) {

if (nullptr == node->right)

return;

Node<T> \*parent\_ptr = node->parent;

Node<T> \*right\_ptr = node->right;

Node<T> \*mid = right\_ptr->left;

Node<T> \*temp;

if (nullptr == parent\_ptr) {

root = right\_ptr;

} else {

right\_ptr->parent = parent\_ptr;

if (parent\_ptr->right == node)

parent\_ptr->right = right\_ptr;

else

parent\_ptr->left = right\_ptr;

}

node->right = mid;

if (nullptr != mid)

mid->parent = node;

right\_ptr->parent = parent\_ptr;

right\_ptr->left = node;

node->parent = right\_ptr;

}

template <typename T> void BinaryTree<T>::RightRotate(Node<T> \*node) {

if (nullptr == node->left)

return;

Node<T> \*parent\_ptr = node->parent;

Node<T> \*left\_ptr = node->left;

Node<T> \*mid = left\_ptr->right;

Node<T> \*temp;

if (nullptr == parent\_ptr) {

root = left\_ptr;

} else {

left\_ptr->parent = parent\_ptr;

if (parent\_ptr->left == node)

parent\_ptr->left = left\_ptr;

else

parent\_ptr->right = left\_ptr;

}

node->left = mid;

if (nullptr != mid)

mid->parent = node;

left\_ptr->parent = parent\_ptr;

left\_ptr->right = node;

node->parent = left\_ptr;

}

template <typename T> void BinaryTree<T>::InsertRoot(Node<T> \*new\_node) {

if (nullptr == root) {

root = new\_node;

root->parent = nullptr;

} else

root->AddRoot(new\_node, this);

}

template <typename T> void BinaryTree<T>::InsertRoot(T new\_element) {

InsertRoot(new Node<T>(new\_element));

}

template <typename T>

void Node<T>::AddRoot(Node<T> \*new\_node, BinaryTree<T> \*tree) {

if (nullptr == new\_node)

return;

new\_node->parent = this;

if (new\_node->data < data) {

if (nullptr == left)

left = new\_node;

else

left->AddRoot(new\_node, tree);

tree->RightRotate(new\_node->parent);

} else {

if (nullptr == right)

right = new\_node;

else

right->AddRoot(new\_node, tree);

tree->LeftRotate(new\_node->parent);

}

}

template <typename T>

void Node<T>::AddRandom(Node<T> \*new\_node, BinaryTree<T> \*tree) {

if (nullptr == new\_node || nullptr == tree)

return;

if (0 == rand() % (NumberOfChildren() + 1)) {

AddRoot(new\_node, tree);

} else {

if (new\_node->data < data) {

if (nullptr == left)

Add(new\_node);

else

left->AddRandom(new\_node, tree);

} else {

if (nullptr == right)

Add(new\_node);

else

right->AddRandom(new\_node, tree);

}

}

}

template <typename T> void BinaryTree<T>::InsertRandom(Node<T> \*new\_node) {

if (nullptr == root) {

root = new\_node;

root->parent = nullptr;

} else {

root->AddRandom(new\_node, this);

}

}

template <typename T> void BinaryTree<T>::InsertRandom(T new\_element) {

InsertRandom(new Node<T>(new\_element));

}

Листинг 3. lab05.cpp

#include "binarytree.h"

#include <ctime>

#include <iostream>

#include <vector>

#include <fstream>

using namespace std;

// class Node;

// class BinaryTree;

template <> inline int Node<int>::Fun() {

int result = 0;

if(nullptr!= left) result += left->Fun();

if(nullptr != right ) result += right->Fun();

if(data%2 == 1 && nullptr!= left && nullptr != right) result+=1;

return result;

}

template <> inline int BinaryTree<int>::Fun() {

if(nullptr == root) return 0;

return root->Fun();

}

int main() {

srand(time(0));

ofstream outs1("standart\_out.txt");

ofstream outs2("random\_out.txt");

ofstream outs3("st\_line\_out.txt");

ofstream outs4("rnd\_line\_out.txt");

auto tree = new BinaryTree<int>();

for(int i = 0; i<5000;i++){

tree->Insert(random()%10000);

outs1<<i+1<<"|"<<tree->Higth()<<endl;

}

delete tree;

tree = new BinaryTree<int>();

for(int i = 0; i<5000;i++){

tree->InsertRandom(random()%10000);

outs2<<i+1<<"|"<<tree->Higth()<<endl;

}

delete tree;

tree = new BinaryTree<int>();

for(int i = 0; i<5000;i++){

tree->Insert(i);

outs3<<i+1<<"|"<<tree->Higth()<<endl;

}

delete tree;

tree = new BinaryTree<int>();

for(int i = 0; i<5000;i++){

tree->InsertRandom(i);

outs4<<i+1<<"|"<<tree->Higth()<<endl;

}

delete tree;

return 0;

}