ГУАП

КАФЕДРА № 43

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ассистент |  |  |  | М. А. Мурашова |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ |
| АВЛ – ДЕРЕВЬЯ ПОИСКА |
| по курсу: СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4134К |  |  |  | Иванов И.В. |
|  |  |  | подпись, дата |  | инициалы, фамилия |

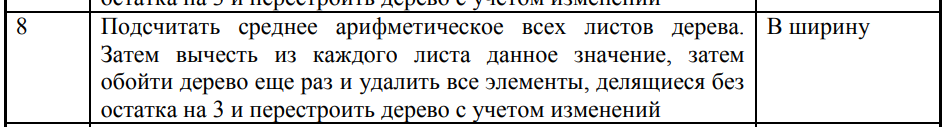
Санкт-Петербург 2022

**Цель работы**: Целью работы является изучение деревьев поиска и получение практических навыков их использования.

**Задание на лабораторную работу:** Разработать на языке программирования высокого уровня программу, которая должна выполнять следующие функции: осуществлять операцию заданную по варианту, добавлять элементы в сбалансированное дерево, удалять элементы из сбалансированного дерева, искать элементы в дереве поиска с выводом количества шагов, за которое осуществляется поиск и тд.

**Ход работы:**

1. В соответствии с формулой рассчитываем свой вариант



1. Разработать алгоритм и запрограммировать его

**Листинг программы**

#include<iostream>

#include <string>

#include <vector>

using namespace std;

struct node //вершина дерева

{

int value;

node\* left;

node\* right;

int height;

};

struct Elem //для списка. Нужен для очереди (используется в обходе)

{

node\* value;

Elem\* next;

};

class List {

int count = 0;

Elem\* head, \* tail;

Elem\* get\_elem(int position, Elem\*& prev\_elem) {

Elem\* elem = head;

prev\_elem = nullptr;

int i = 0;

if (position == 0)

prev\_elem = tail;

while (i < position)

{

if (i == position - 1)

prev\_elem = elem;

elem = elem->next;

i++;

}

return elem;

}

public:

int get\_len() {

return count;

}

void add\_tail(node\* value) {

Elem\* tmp = new Elem;

if (count == 0) {

head = tail = tmp;

tmp->next = tmp;

}

else {

tail->next = tmp;

tail = tmp;

tmp->next = head;

}

tmp->value = value;

count++;

}

node\* get\_elem\_value(int position) {

Elem\* tmp = head;

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

tmp = tmp->next;

}

return tmp->value;

}

void del\_elem(int position) {

Elem\* prevDel = 0;

Elem\* del = get\_elem(position, prevDel);

if (count != 1) {

Elem\* afterDel = del->next;

prevDel->next = afterDel;

if ((count + position) % count == 0)

head = afterDel;

else if (position % (count - 1) == 0)

tail = prevDel;

}

else {

head = 0;

tail = 0;

}

delete del;

count--;

}

void del\_list() {

while (count != 0)

del\_elem(0);

}

};

class Queue { //Очередь

private:

List queue;

public:

void push(node\* t) {

queue.add\_tail(t);

}

node\* pop() {

node\* tmp = queue.get\_elem\_value(0);

queue.del\_elem(0);

return tmp;

}

node\* get() {

if (queue.get\_len() > 0)

return queue.get\_elem\_value(0);

}

bool is\_empty() {

return queue.get\_len() == 0;

}

};

class AVL\_tree

{

node\* root = NULL;

void del\_tree(node\* element) //чистит все дерево

{

vector<int> v;

order(root, v);

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

remove(v[i]);

}

node\* insert(int value, node\* element) //вставляет элемент и балансирует дерево

//если такой элемент был в дереве - не добавляет

{

if (element == NULL)

{

element = new node;

element->value = value;

element->height = 0;

element->left = element->right = NULL;

}

else if (value < element->value)

{

element->left = insert(value, element->left);

if (height(element->left) - height(element->right) == 2)

{

if (value < element->left->value)

element = rightRotate(element);

else

element = bigRightRotate(element);

}

}

else if (value > element->value)

{

element->right = insert(value, element->right);

if (height(element->right) - height(element->left) == 2)

{

if (value > element->right->value)

element = leftRotate(element);

else

element = bigLeftRotate(element);

}

}

element->height = max(height(element->left), height(element->right)) + 1;

return element;

}

node\* rightRotate(node\*& element) //4 метода поворотов, большие реализованны как правый, левый и левый, правый (по 2 малых делает)

{

node\* u = element->left;

element->left = u->right;

u->right = element;

element->height = max(height(element->left), height(element->right)) + 1;

u->height = max(height(u->left), element->height) + 1;

return u;

}

node\* leftRotate(node\*& element)

{

node\* u = element->right;

element->right = u->left;

u->left = element;

element->height = max(height(element->left), height(element->right)) + 1;

u->height = max(height(element->right), element->height) + 1;

return u;

}

node\* bigLeftRotate(node\*& element)

{

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

return leftRotate(element);

}

node\* bigRightRotate(node\*& element)

{

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

return rightRotate(element);

}

node\* findMin(node\* element) //минимальный элемент в поддереве

{

if (element == NULL)

return NULL;

else if (element->left == NULL)

return element;

else

return findMin(element->left);

}

int getBalance(node\* element) //балансировка в поддереве

{

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

}

void fixheight(node\* element)//восстановить значение высоты заданного узла

{

element->height = max(height(element->left), height(element->right)) + 1;

}

node\* balance(node\* p) // балансировка узла p

{

fixheight(p);

if (getBalance(p) == 2)

{

if (getBalance(p->right) < 0)

p->right = rightRotate(p->right);

return leftRotate(p);

}

if (getBalance(p) == -2)

{

if (getBalance(p->left) > 0)

p->left = leftRotate(p->left);

return rightRotate(p);

}

return p; // балансировка не нужна

}

node\* removeMin(node\* p) // удаление узла с минимальным ключом из поддерева p

{

if (p->left == 0)

return p->right;

p->left = removeMin(p->left);

return balance(p);

}

node\* remove(node\* p, int k) // удаление ключа k из дерева p

{

if (!p) return 0;

if (k < p->value)

p->left = remove(p->left, k);

else if (k > p->value)

p->right = remove(p->right, k);

else // k == p->key

{

node\* q = p->left;

node\* r = p->right;

delete p;

if (!r) return q;

node\* min = findMin(r);

min->right = removeMin(r);

min->left = q;

return balance(min);

}

return balance(p);

}

int height(node\* element) //высота узла

{

if (element == NULL)

return -1;

else

return element->height;

}

node\* search(node\*& element, int val, int k = 0) { //есть ли элемент в дереве

if (val == element->value) { //элемент найден

cout << "Поиск за: " << k++ << endl;

return element;

}

k++;

if (val < element->value) { //элемент меньше - идем влево

if (element->left == NULL) {

return NULL;

}

search(element->left, val, k);

}

else { //больше - вправо

if (element->right == NULL) {

return NULL;

}

search(element->right, val, k);

}

}

void printBT(const std::string& prefix, const node\* node, bool isLeft) //вывод дерева

{

if (node != NULL)

{

cout << prefix;

cout << (isLeft ? "|==" : "L==");

cout << node->value << std::endl;

printBT(prefix + (isLeft ? "| " : " "), node->left, true);

printBT(prefix + (isLeft ? "| " : " "), node->right, false);

}

}

void order(node\* root, vector<int>& v) { //обход в ширину от вершины

Queue q;

//Для начала поместим в очередь корень

q.push(root);

while (!q.is\_empty()) {

node\* tmp = q.pop();

v.push\_back(tmp->value); //в векторе хранится обход, каждый раз добавляем посещенную вершину

//Если есть левый наследник, то помещаем его в очередь для дальнейшей обработки

if (tmp->left) {

q.push(tmp->left);

}

//Если есть правый наследник, то помещаем его в очередь для дальнейшей обработки

if (tmp->right) {

q.push(tmp->right);

}

}

}

int avg() { //подсчет среднего арифмитического. Т.к инт - с некоторым округлением

vector<int> v;

order(root, v); //просто получаем список обхода в вектор

int sum = 0;

for (int i = 0; i < v.size(); i++) //собираем сумму в векторе

sum += v[i];

return sum / v.size(); //и возврат сумма на количество элементов

}

void minus(int value) {

vector<int> v;

order(root, v); //также получаем список обхода

for (int i = 0; i < v.size(); i++) //вычитаем из каждого элемента значение

v[i] -= value;

del\_tree(root); //удаляем дерево(чистим)

for (int i = 0; i < v.size(); i++) //и добавляем все по новой. Т.к обход в ширину дерево сохранит свой вид

insert(v[i]);

}

void del\_mod3() {

vector<int> v;

order(root, v); //также получаем список обхода

for (int i = 0; i < v.size(); i++) //делится на 3 - удаляем

if (v[i] % 3 == 0)

remove(v[i]); //после удалений дерево будет оставаться сбалансированным

}

void task(node\* root) { //само задание

int a = avg();

cout << "\nСреденее зачение = " << a << endl;

minus(a);

cout << "\nДерево после вычитаний\n";

print();

cout << endl;

del\_mod3();

cout << "\nДерево после удалений кратных 3\n";

print();

cout << endl;

}

public:

node\* getroot() {

return root;

}

AVL\_tree()//конструктор

{

root = NULL;

}

void task() { //задание

task(root);

}

void insert(int value) //вставка от корня

{

root = insert(value, root);

}

void remove(int value) //удаление по значению от корня

{

root = remove(root, value);

}

void print() //вывод дерева от корня

{

printBT("", root, false);

}

void order() { //обход дерева в ширину, по умолчанию от корня дерева

vector<int> v;

order(root, v);

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

cout << v[i] << " ";

cout << endl;

}

bool search(int value) { //поиск по значению

return search(root, value);

}

};

struct Trunk

{

Trunk\* prev;

string str;

Trunk(Trunk\* prev, string str)

{

this->prev = prev;

this->str = str;

}

};

// Вспомогательная функция для печати ветвей бинарного дерева

void showTrunks(Trunk\* p)

{

if (p == nullptr) {

return;

}

showTrunks(p->prev);

cout << p->str;

}

void showTree(node\* root, Trunk\* prev, bool isLeft)

{

if (root == nullptr) {

return;

}

string prev\_str = " ";

Trunk\* trunk = new Trunk(prev, prev\_str);

showTree(root->right, trunk, true);

if (!prev) {

trunk->str = "---";

}

else if (isLeft)

{

trunk->str = ".---";

prev\_str = " |";

}

else {

trunk->str = "'---";

prev->str = prev\_str;

}

showTrunks(trunk);

cout << " " << root->value << endl;

if (prev) {

prev->str = prev\_str;

}

trunk->str = " |";

showTree(root->left, trunk, false);

};

int main()

{

setlocale(0, "");

node Node;

AVL\_tree element;

srand(time(NULL)); //чтобы рандом работал

int choise = 96;

for (;;) {

cout << "1 Добавить элемент\n";

cout << "2 Показать дерево\n";

cout << "3 Найти элемент\n";

cout << "4 Удалить элемент\n";

cout << "5 Обойти дерево(в ширину)\n";

cout << "6 Начать задание\n";

cout << "7 Добавить рандомные элементы\n";

cout << "0 Выход\n";

cout << "\n";

cout << "Выбираем пункт: ";

cin >> choise;

cout << "##################################\n";

cout << "\n";

switch (choise) {

case 1: {

cout << "Ввод элемента для добавления в дерево, добавлются только уникальные\n";

int value;

cin >> value;

element.insert(value);

break;

}

case 2: {

node\* ku = element.getroot();

showTree(ku, nullptr, false);

break;

}

case 3: {

cout << "Поиск элемента. Он есть - вывод кол-ва шагов, нет - ничего\n";

int value;

cin >> value;

element.search(value);

break;

}

case 4: {

cout << "Ввод элемента для удаления из дерева\n";

int value;

cin >> value;

element.remove(value);

break;

}

case 5: {

element.order();

break;

}

case 6: {

element.task();

break;

}

case 7: {

srand(time(0));

system("cls");

int i;

int n;

cout << "Сколько элементов добавить?" << endl;

cin >> i;

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

n = rand() % 100;

element.insert(n);

}

break;

}

case 0: {

return 0;

}

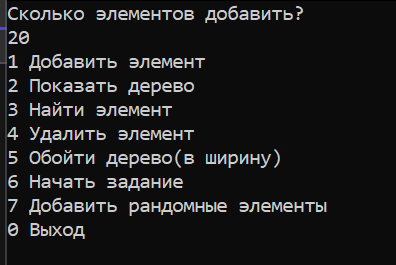
}

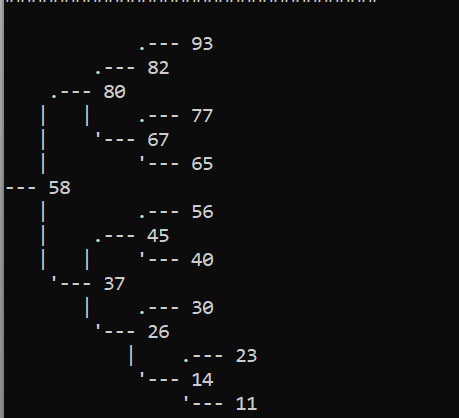
}

}

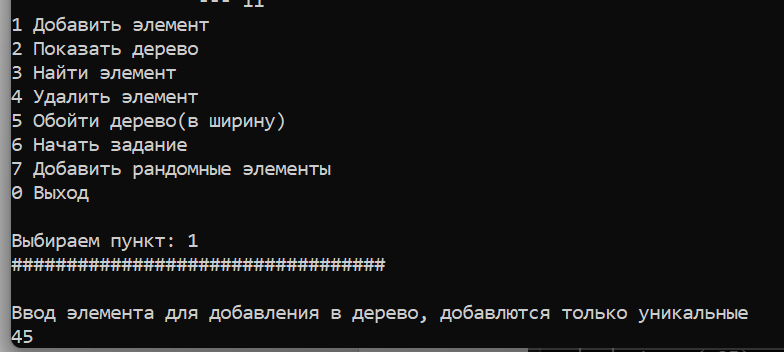
**Работа программы:**

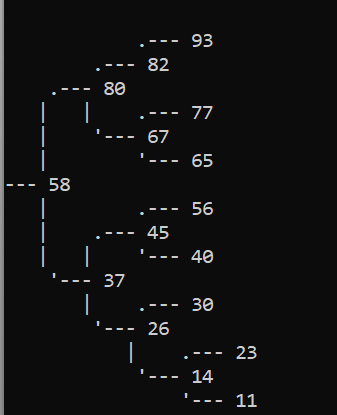
**Добавление случайных значений**

****

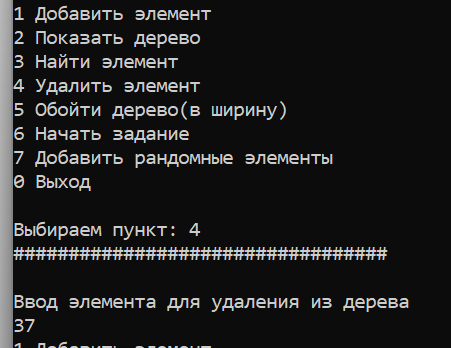
****

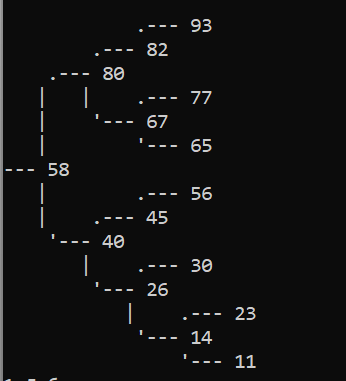
**Добавление ключа в дерево**

****

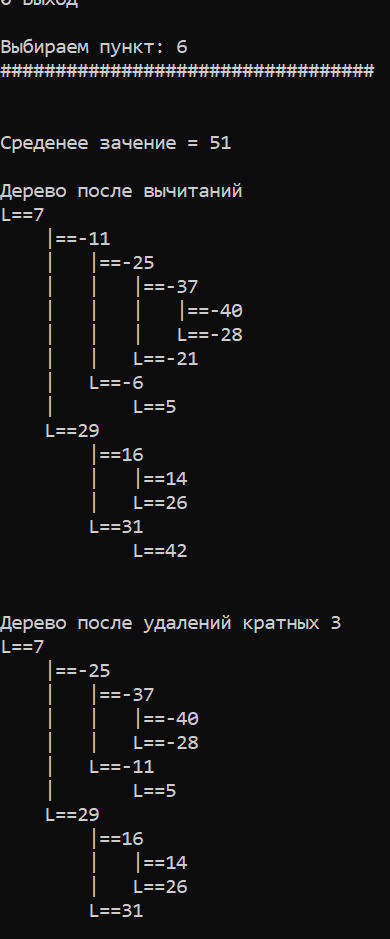
****

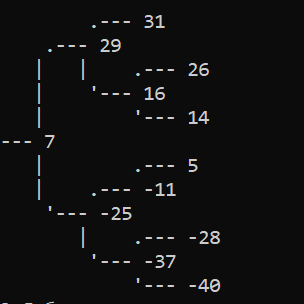
**Удаление ключа**

****

****

**Выполнение задания согласно варианту**

****

****

**Вывод:**

Я изучил деревья поиска и получил практические навыки по их использованию.