|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |   Институт Информационных технологий | |
|  | |
| Кафедра Математического обеспечения и стандартизации информационных технологий | |
|  | |
|  | |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 4** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Сбалансированные деревья поиска (СДП) и их применение для поиска данных в файле.»** | |
|  | |
| Выполнил студент группы ИКБО-02-21 | Семянников Н.С. |
| Принял преподаватель | Сорокин А.В. |

|  |  |  |
| --- | --- | --- |
| Лабораторная работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2022

[1. Отчёт по заданию 1 4](#_Toc119287714)

[1.1. Условие задачи 4](#_Toc119287715)

[1.2. Подход к решению 4](#_Toc119287716)

[1.3. Алгоритмы операций 5](#_Toc119287717)

[1.3.1. Функция создания узла 5](#_Toc119287718)

[1.3.2. Функция добавления узла 5](#_Toc119287719)

[1.3.3. Функция удаления элемента 6](#_Toc119287720)

[1.3.4. Функция поиска элемента 7](#_Toc119287721)

[1.3.5. Функция создания дерева с помощью случайных значений 7](#_Toc119287722)

[1.3.6. Создание дерева с помощью бинарного файла 8](#_Toc119287723)

[1.3.7. Функция заполнения бинарного файла данными из текстового файла 8](#_Toc119287724)

[1.3.8. Функция вывода дерева 9](#_Toc119287725)

[1.4. Содержание файла 9](#_Toc119287726)

[1.5. Тестирование 9](#_Toc119287727)

[2. Отчёт по заданию 2 11](#_Toc119287728)

[2.1. Условие задачи 11](#_Toc119287729)

[2.2. Подход к решению 11](#_Toc119287730)

[2.3. Алгоритмы операций 12](#_Toc119287731)

[2.3.1. Функция создания узла 12](#_Toc119287732)

[2.3.2. Функция добавления узла 12](#_Toc119287733)

[2.3.3. Функция поворота влево 13](#_Toc119287734)

[2.3.4. Функция поворота вправо 13](#_Toc119287735)

[2.3.5. Функция удаления элемента 14](#_Toc119287736)

[2.3.6. Функция поиска элемента 15](#_Toc119287737)

[2.3.7. Функция создания дерева с помощью случайных значений 15](#_Toc119287738)

[2.3.8. Создание дерева с помощью бинарного файла 15](#_Toc119287739)

[2.3.9. Функция заполнения бинарного файла данными из текстового файла 16](#_Toc119287740)

[2.3.10. Функция вывода дерева 16](#_Toc119287741)

[2.4. Код программы 17](#_Toc119287742)

[2.5. Тестирование 22](#_Toc119287743)

[2.6. Количество поворотов при балансировке СДП 24](#_Toc119287744)

[3. Отчёт по заданию 3 25](#_Toc119287745)

[4. Вывод 26](#_Toc119287746)

[5. Список литературы 27](#_Toc119287747)

# Отчёт по заданию 1

## Условие задачи

Разработать приложение, которое использует бинарное дерево поиска (БДП) для поиска записи с ключом в бинарном файле. Разработать класс «Бинарное дерево поиска». Тип информационной части узла ключ и ссылка на запись в файле. Методы: включение элемента в дерево, поиск ключа в дереве, удаление ключа из дерева, отображение дерева.

**Дано**:

Файл двоичный с записями фиксированной длины.

Бинарное дерево поиска

Структура записи файла:

Код специальности, название вуза.

**Результат**: приложение, которое использует БДП для организации прямого доступа к записям файла и работы с ними.

Условие варианта №7(таблица 1):

Таблица 1 - Условие варианта №7

|  |  |  |
| --- | --- | --- |
| №  варианта | Сбалансированное дерево поиска (СДП) | Операции варианта |
| 7 | АВЛ | Специализация вуза: код специальности, название вуза |

## Подход к решению

Структура узла имеет следующие поля: key – ключ к хранящимся данным в файле, data – значения для добавления его в файл, а также два указателя на левый и правый узел дерева.

Операции по управлению БДП: вставка, удаление и поиск элемента, создание БДП с помощью случайно сгенерированных данных, создание БДП с помощью бинарного файла.

Структура записи файла: ключ, длина строки, строка.

Операции по управлению файлом: считывание/запись из/в файл, заполнение бинарного файла данными из текстового файла.

## Алгоритмы операций

### Функция создания узла

Принимает: ключ и значение узла.

Возвращает: узел.

Описание: создаёт новый узел с предоставленным ключом и значением.

Код:

Node\* newNode(int key, string data) {

Node\* node = new Node();

node->key = key;

node->data = data;

node->left = NULL;

node->right = NULL;

node->height = 1;

return (node);

}

### Функция добавления узла

Принимает: узел, ключ, значение и режим работы.

Возвращает узел.

Описание: рекурсивно ищет место для вставки нового узла, после нахождения узла возвращает головной узел. Режим работы необходим для выбора дерева (БДП или СДП).

Код:

Node\* insertNode(Node\* node, int key, string data, int mode) {

if (node == NULL)

return (newNode(key, data));

if (key < node->key)

node->left = insertNode(node->left, key, data, mode);

else if (key > node->key)

node->right = insertNode(node->right, key, data, mode);

else

return node;

if (mode == 1)

{

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

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1) {

if (key < node->left->key) {

rotates++;

return rightRotate(node);

}

else if (key > node->left->key) {

rotates++;

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

rotates++;

return rightRotate(node);

}

}

if (balanceFactor < -1) {

if (key > node->right->key) {

rotates++;

return leftRotate(node);

}

else if (key < node->right->key) {

rotates++;

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

rotates++;

return leftRotate(node);

}

}

}

return node;

}

### Функция удаления элемента

Принимает: узел, ключ, режим.

Возвращает: узел.

Описание: рекурсивно находит узел, который надо удалить. После этого в зависимости от количества потомков делает следующее:

0 потомков – удаляет узел.

1 потомок – удаляемый узел заменяется своим потомком.

2 потомка – рекурсивно находится минимальное значение в поддереве правого потомка, после чего ключ и значение удаляемого узла заменяются ключом и значение найденного узла при этом найденный узел удаляется.

Режим необходим для выбора дерева (БДП и СДП)

Код:

Node\* deleteNode(Node\* root, int key, int mode) {

if (root == NULL)

return root;

if (key < root->key)

root->left = deleteNode(root->left, key, mode);

else if (key > root->key)

root->right = deleteNode(root->right, key, mode);

else {

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

Node\* temp = root->left ? root->left : root->right;

if (temp == NULL) {

temp = root;

root = NULL;

}

else

\*root = \*temp;

free(temp);

}

else {

Node\* temp = nodeWithMinimumValue(root->right);

root->key = temp->key;

root->data = temp->data;

root->right = deleteNode(root->right, temp->key, mode);

}

}

if (root == NULL)

return root;

if (mode == 1) {

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

int balanceFactor = getBalanceFactor(root);

if (balanceFactor > 1) {

if (getBalanceFactor(root->left) >= 0) {

return rightRotate(root);

}

else {

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

return rightRotate(root);

}

}

if (balanceFactor < -1) {

if (getBalanceFactor(root->right) <= 0) {

return leftRotate(root);

}

else {

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

return leftRotate(root);

}

}

}

return root;

}

### Функция поиска элемента

Принимает: узел, ключ.

Возвращает: узел.

Описание: рекурсивно находит узел с переданным ключом.

Код:

Node\* find(Node\* root, int key) {

checks++;

if (root == nullptr || key == root->key)

{

return root;

}

else if (key < root->key) {

find(root->left, key);

}

else {

find(root->right, key);

}

}

### Функция создания дерева с помощью случайных значений

Принимает: узел, количество узлов, режим.

Возвращает: узел.

Описание: циклично добавляет новые узлы в дерево.

Код:

Node\* createNew(Node\* root, int size, int mode) {

int key;

string s[5] = { "МИРЭА", "МАИ", "МЭИ", "РЭУ", "МГУ" };

random\_device rd;

mt19937 rmt(rd());

uniform\_int\_distribution<int> random(100000, 999999);

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

{

key = random(rmt);

root = insertNode(root, key, s[rand() % 5], mode);

}

return root;

}

### Создание дерева с помощью бинарного файла

Принимает: узел, файл.

Возвращает: узел.

Описание: считывает бинарный файл и добавляет узел с полученными данными.

Код:

Node\* readFile(fstream& f, Node\* root) {

map<int, string> univers;

rotates = 0;

string s = "";

char\* endLiner;

int k = 0;

while (f.good() && !f.eof())

{

f.read((char\*)&k, sizeof(int));

f.read((char\*)&endLiner, sizeof('\n'));

getline(f, s, '\0');

univers.insert({ k, s });

}

for (auto& kv : univers)

{

root = insertNode(root, kv.first, kv.second, 1);

}

cout << "Среднее количество поворотов = " << ((double)rotates / univers.size()) << endl;

return root;

}

### Функция заполнения бинарного файла данными из текстового файла

Описание: считывает текстовый файл и заполняет полученными данными бинарный файл.

Код:

void textToBin() {

int key;

string str, info;

fstream txt("input.bin", ios::in);

fstream bin("university", ios::out | ios::binary);

while (getline(txt, info)) {

regex rgx("\\s+");

sregex\_token\_iterator iter(info.begin(), info.end(), rgx, -1);

key = stoi(\*iter);

str = info.substr(to\_string(key).size() + 1);

bin.write(reinterpret\_cast<char\*>(&key), sizeof(int));

bin.write("\0", sizeof(char));

bin.write(str.c\_str(), str.size());

bin.write("\0", sizeof(char));

}

txt.close();

bin.close();

}

### Функция вывода дерева

Принимает: узел.

Описание: рекурсивно выводит дерево.

Код:

void printBinTree(Node\* root, int level, int L) {

if (root != nullptr) {

printBinTree(root->right, level + 1, L);

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

cout << " ";

};

cout << root->key << endl;

printBinTree(root->left, level + 1, L);

}

}

## Содержание файла

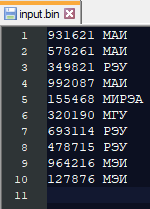


Рисунок 1 - Содержание текстового файла

## Тестирование

|  |  |  |  |
| --- | --- | --- | --- |
| № теста | Вводимые значения | Полученные данные | Тест пройден/не пройден |
| 1 | n = 5  root = {13, 17, 22, 3, 8}  find1 = 2  find2 = 17  del = 3 | root = {13, 17, 22, 3, 8}  find1 – не найден  find2 – найден  root = {13, 17, 22, 8} | пройден |

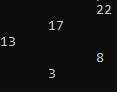


Рисунок 2 - Вывод дерева



Рисунок 3 - Поиск элемента

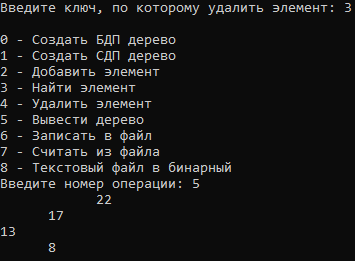


Рисунок 4 - Удаление элемента

# Отчёт по заданию 2

## Условие задачи

Разработать приложение, которое использует сбалансированное дерево поиска для доступа к записям файла. Разработать класс СДП. Структура информационной части узла дерева включает ключ и ссылку на запись в файле (адрес места размещения). Основные методы: включение элемента в дерево; поиск ключа в дереве с возвратом ссылки; удаление ключа из дерева; вывод дерева в форме дерева (с отображением структуры дерева). Определить среднее число выполненных поворотов при включении ключей в дерево при формировании дерева из двоичного файла

**Дано**:

Файл двоичный с записями фиксированной длины.

АВЛ сбалансированное дерево поиска (СДП)

Структура записи файла:

Код специальности, название вуза.

**Результат**:

Приложение, которое использует АВЛ СДП для организации прямого доступа к записям файла и работы с ними.

Условие варианта №7(таблица 2):

Таблица 2 - Условие варианта №7

|  |  |  |
| --- | --- | --- |
| №  варианта | Сбалансированное дерево поиска (СДП) | Операции варианта |
| 7 | АВЛ | Специализация вуза: код специальности, название вуза |

## Подход к решению

Структура узла имеет следующие поля: key – ключ к хранящимся данным в файле, data – значения для добавления его в файл, а также два указателя на левый и правый узел дерева.

Операции по управлению СДП: вставка, удаление и поиск элемента, создание СДП с помощью случайно сгенерированных данных, поворот влево, поворот вправо, создание СДП с помощью бинарного файла.

Структура записи файла: ключ, длина строки, строка.

Операции по управлению файлом: считывание/запись из/в файл, заполнение бинарного файла данными из текстового файла.

## Алгоритмы операций

### Функция создания узла

Принимает: ключ и значение узла.

Возвращает: узел.

Описание: создаёт новый узел с предоставленным ключом и значением.

Код:

Node\* newNode(int key, string data) {

Node\* node = new Node();

node->key = key;

node->data = data;

node->left = NULL;

node->right = NULL;

node->height = 1;

return (node);

}

### Функция добавления узла

Принимает: узел, ключ, значение и режим работы.

Возвращает узел.

Описание: рекурсивно ищет место для вставки нового узла, после нахождения узла возвращает головной узел. Режим работы необходим для выбора дерева (БДП или СДП).

Код:

Node\* insertNode(Node\* node, int key, string data, int mode) {

if (node == NULL)

return (newNode(key, data));

if (key < node->key)

node->left = insertNode(node->left, key, data, mode);

else if (key > node->key)

node->right = insertNode(node->right, key, data, mode);

else

return node;

if (mode == 1)

{

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

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1) {

if (key < node->left->key) {

rotates++;

return rightRotate(node);

}

else if (key > node->left->key) {

rotates++;

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

rotates++;

return rightRotate(node);

}

}

if (balanceFactor < -1) {

if (key > node->right->key) {

rotates++;

return leftRotate(node);

}

else if (key < node->right->key) {

rotates++;

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

rotates++;

return leftRotate(node);

}

}

}

return node;

}

### Функция поворота влево

Принимает: узел.

Возвращает: узел.

Описание: делает родительский узел дочерним своего правого поддерева.

Код:

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

### Функция поворота вправо

Принимает: узел.

Возвращает: узел.

Описание: делает родительский узел дочерним своего левого поддерева.

Код:

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

### Функция удаления элемента

Принимает: узел, ключ, режим.

Возвращает: узел.

Описание: рекурсивно находит узел, который надо удалить. После этого в зависимости от количества потомков делает следующее:

0 потомков – удаляет узел.

1 потомок – удаляемый узел заменяется своим потомком.

2 потомка – рекурсивно находится минимальное значение в поддереве правого потомка, после чего ключ и значение удаляемого узла заменяются ключом и значение найденного узла при этом найденный узел удаляется.

Режим необходим для выбора дерева (БДП и СДП)

Код:

Node\* deleteNode(Node\* root, int key, int mode) {

if (root == NULL)

return root;

if (key < root->key)

root->left = deleteNode(root->left, key, mode);

else if (key > root->key)

root->right = deleteNode(root->right, key, mode);

else {

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

Node\* temp = root->left ? root->left : root->right;

if (temp == NULL) {

temp = root;

root = NULL;

}

else

\*root = \*temp;

free(temp);

}

else {

Node\* temp = nodeWithMinimumValue(root->right);

root->key = temp->key;

root->data = temp->data;

root->right = deleteNode(root->right, temp->key, mode);

}

}

if (root == NULL)

return root;

if (mode == 1) {

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

int balanceFactor = getBalanceFactor(root);

if (balanceFactor > 1) {

if (getBalanceFactor(root->left) >= 0) {

return rightRotate(root);

}

else {

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

return rightRotate(root);

}

}

if (balanceFactor < -1) {

if (getBalanceFactor(root->right) <= 0) {

return leftRotate(root);

}

else {

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

return leftRotate(root);

}

}

}

return root;

}

### Функция поиска элемента

Принимает: узел, ключ.

Возвращает: узел.

Описание: рекурсивно находит узел с переданным ключом.

Код:

Node\* find(Node\* root, int key) {

checks++;

if (root == nullptr || key == root->key)

{

return root;

}

else if (key < root->key) {

find(root->left, key);

}

else {

find(root->right, key);

}

}

### Функция создания дерева с помощью случайных значений

Принимает: узел, количество узлов, режим.

Возвращает: узел.

Описание: циклично добавляет новые узлы в дерево.

Код:

Node\* createNew(Node\* root, int size, int mode) {

int key;

string s[5] = { "МИРЭА", "МАИ", "МЭИ", "РЭУ", "МГУ" };

random\_device rd;

mt19937 rmt(rd());

uniform\_int\_distribution<int> random(100000, 999999);

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

{

key = random(rmt);

root = insertNode(root, key, s[rand() % 5], mode);

}

return root;

}

### Создание дерева с помощью бинарного файла

Принимает: узел, файл.

Возвращает: узел.

Описание: считывает бинарный файл и добавляет узел с полученными данными.

Код:

Node\* readFile(fstream& f, Node\* root) {

map<int, string> univers;

rotates = 0;

string s = "";

char\* endLiner;

int k = 0;

while (f.good() && !f.eof())

{

f.read((char\*)&k, sizeof(int));

f.read((char\*)&endLiner, sizeof('\n'));

getline(f, s, '\0');

univers.insert({ k, s });

}

for (auto& kv : univers)

{

root = insertNode(root, kv.first, kv.second, 1);

}

cout << "Среднее количество поворотов = " << ((double)rotates / univers.size()) << endl;

return root;

}

### Функция заполнения бинарного файла данными из текстового файла

Описание: считывает текстовый файл и заполняет полученными данными бинарный файл.

Код:

void textToBin() {

int key;

string str, info;

fstream txt("input.bin", ios::in);

fstream bin("university", ios::out | ios::binary);

while (getline(txt, info)) {

regex rgx("\\s+");

sregex\_token\_iterator iter(info.begin(), info.end(), rgx, -1);

key = stoi(\*iter);

str = info.substr(to\_string(key).size() + 1);

bin.write(reinterpret\_cast<char\*>(&key), sizeof(int));

bin.write("\0", sizeof(char));

bin.write(str.c\_str(), str.size());

bin.write("\0", sizeof(char));

}

txt.close();

bin.close();

}

### Функция вывода дерева

Принимает: узел.

Описание: рекурсивно выводит дерево.

Код:

void printBinTree(Node\* root, int level, int L) {

if (root != nullptr) {

printBinTree(root->right, level + 1, L);

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

cout << " ";

};

cout << root->key << endl;

printBinTree(root->left, level + 1, L);

}

}

## Код программы

#include <iostream>

#include <random>

#include <string>

#include <fstream>

#include <map>

#include <chrono>

#include <regex>

using namespace std;

using namespace chrono;

int rotates, checks;

class Node {

public:

int key;

string data;

Node\* left;

Node\* right;

int height;

};

int height(Node\* N) {

if (N == NULL)

return 0;

return N->height;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

//Создание узла

Node\* newNode(int key, string data) {

Node\* node = new Node();

node->key = key;

node->data = data;

node->left = NULL;

node->right = NULL;

node->height = 1;

return (node);

}

//Поворот вправо

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

//Поворот влево

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

//Определить фактор сбалансированности

int getBalanceFactor(Node\* N) {

if (N == NULL)

return 0;

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

}

//Добавить узел

Node\* insertNode(Node\* node, int key, string data, int mode) {

if (node == NULL)

return (newNode(key, data));

if (key < node->key)

node->left = insertNode(node->left, key, data, mode);

else if (key > node->key)

node->right = insertNode(node->right, key, data, mode);

else

return node;

if (mode == 1)

{

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

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1) {

if (key < node->left->key) {

rotates++;

return rightRotate(node);

}

else if (key > node->left->key) {

rotates++;

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

rotates++;

return rightRotate(node);

}

}

if (balanceFactor < -1) {

if (key > node->right->key) {

rotates++;

return leftRotate(node);

}

else if (key < node->right->key) {

rotates++;

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

rotates++;

return leftRotate(node);

}

}

}

return node;

}

Node\* createNew(Node\* root, int size, int mode) {

int key;

string s[5] = { "МИРЭА", "МАИ", "МЭИ", "РЭУ", "МГУ" };

random\_device rd;

mt19937 rmt(rd());

uniform\_int\_distribution<int> random(100000, 999999);

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

{

key = random(rmt);

root = insertNode(root, key, s[rand() % 5], mode);

}

return root;

}

//Найти узел с наименьшим значением

Node\* nodeWithMinimumValue(Node\* node) {

Node\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

//Удалить узел

Node\* deleteNode(Node\* root, int key, int mode) {

if (root == NULL)

return root;

if (key < root->key)

root->left = deleteNode(root->left, key, mode);

else if (key > root->key)

root->right = deleteNode(root->right, key, mode);

else {

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

Node\* temp = root->left ? root->left : root->right;

if (temp == NULL) {

temp = root;

root = NULL;

}

else

\*root = \*temp;

free(temp);

}

else {

Node\* temp = nodeWithMinimumValue(root->right);

root->key = temp->key;

root->data = temp->data;

root->right = deleteNode(root->right, temp->key, mode);

}

}

if (root == NULL)

return root;

if (mode == 1) {

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

int balanceFactor = getBalanceFactor(root);

if (balanceFactor > 1) {

if (getBalanceFactor(root->left) >= 0) {

return rightRotate(root);

}

else {

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

return rightRotate(root);

}

}

if (balanceFactor < -1) {

if (getBalanceFactor(root->right) <= 0) {

return leftRotate(root);

}

else {

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

return leftRotate(root);

}

}

}

return root;

}

void printBinTree(Node\* root, int level, int L) {

if (root != nullptr) {

printBinTree(root->right, level + 1, L);

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

cout << " ";

};

cout << root->key << endl;

printBinTree(root->left, level + 1, L);

}

}

Node\* find(Node\* root, int key) {

checks++;

if (root == nullptr || key == root->key)

{

return root;

}

else if (key < root->key) {

find(root->left, key);

}

else {

find(root->right, key);

}

}

//Считать из файла

Node\* readFile(fstream& f, Node\* root) {

map<int, string> univers;

rotates = 0;

string s = "";

char\* endLiner;

int k = 0;

while (f.good() && !f.eof())

{

f.read((char\*)&k, sizeof(int));

f.read((char\*)&endLiner, sizeof('\n'));

getline(f, s, '\0');

univers.insert({ k, s });

}

for (auto& kv : univers)

{

root = insertNode(root, kv.first, kv.second, 1);

}

cout << "Среднее количество поворотов = " << ((double)rotates / univers.size()) << endl;

return root;

}

//Записать в файл

void writeFile(fstream& f, Node\* root) {

if (root != nullptr)

{

writeFile(f, root->left);

writeFile(f, root->right);

f.write(reinterpret\_cast<char\*>(&root->key), sizeof(int));

f.write("\0", sizeof(char));

f.write(root->data.c\_str(), root->data.size());

f.write("\0", sizeof(char));

}

}

//Удалить дерево

void deleteTree(Node\* root)

{

if (root == NULL) return;

deleteTree(root->left);

deleteTree(root->right);

delete root;

}

//Текстовый в бинарный

void textToBin() {

int key;

string str, info;

fstream txt("input.bin", ios::in);

fstream bin("university", ios::out | ios::binary);

while (getline(txt, info)) {

regex rgx("\\s+");

sregex\_token\_iterator iter(info.begin(), info.end(), rgx, -1);

key = stoi(\*iter);

str = info.substr(to\_string(key).size() + 1);

bin.write(reinterpret\_cast<char\*>(&key), sizeof(int));

bin.write("\0", sizeof(char));

bin.write(str.c\_str(), str.size());

bin.write("\0", sizeof(char));

}

txt.close();

bin.close();

}

int main() {

setlocale(0, "");

system("chcp 1251");

Node\* root = NULL, \* tmp = NULL;

string value;

int key = 0, a = 0, count = 0, mode = 1;

while (true)

{

cout << "0 - Создать БДП дерево\n";

cout << "1 - Создать СДП дерево\n";

cout << "2 - Добавить элемент\n";

cout << "3 - Найти элемент\n";

cout << "4 - Удалить элемент\n";

cout << "5 - Вывести дерево\n";

cout << "6 - Записать в файл\n";

cout << "7 - Считать из файла\n";

cout << "8 - Текстовый файл в бинарный\n";

cout << "Введите номер операции: ";

cin >> a;

switch (a)

{

case 0: {

mode = 0;

cout << "Введите размер дерева: ";

cin >> count;

deleteTree(root);

root = NULL;

root = createNew(root, count, mode);

break;

}

case 1: {

mode = 1;

cout << "Введите размер дерева: ";

cin >> count;

deleteTree(root);

root = NULL;

root = createNew(root, count, mode);

break;

}

case 2: {

cout << "Введите ключ: ";

cin >> key;

cout << "Введите значение: ";

cin >> value;

root = insertNode(root, key, value, mode);

break;

}

case 3: {

checks = 0;

cout << "Введите ключ для поиска: ";

cin >> key;

auto start = high\_resolution\_clock::now();

tmp = find(root, key);

auto stop = high\_resolution\_clock::now();

if (tmp != nullptr) {

cout << "Узел " << key << " имеет адрес " << &tmp->data << endl;

}

cout << "Проверок: " << checks << endl;

cout << "Время поиска: " << duration\_cast<nanoseconds>(stop - start).count() << endl;

break;

}

case 4: {

cout << "Введите ключ, по которому удалить элемент: ";

cin >> key;

deleteNode(root, key, mode);

break;

}

case 5: {

//printTree(root, "", true);

printBinTree(root, 0, 6);

break;

}

case 6: {

fstream f("university", ios::out | ios::binary);

writeFile(f, root);

f.close();

break;

}

case 7: {

deleteTree(root);

root = NULL;

fstream f("university", ios::in | ios::binary);

root = readFile(f, root);

f.close();

break;

}

case 8: {

textToBin();

break;

}

default:

exit(0);

break;

}

cout << endl;

}

}

## Тестирование

|  |  |  |  |
| --- | --- | --- | --- |
| № теста | Вводимые значения | Полученные данные | Тест пройден/не пройден |
| 1 | n = 5  root = {13, 17, 22, 3, 8}  find1 = 2  find2 = 22  del = 8 | root = {17, 8, 3, 13, 22}  find1 – не найден  find2 – найден  root = {17, 13, 3, 22} | пройден |

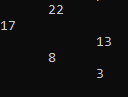


Рисунок 5 - Вывод дерева

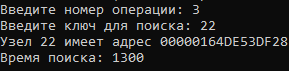


Рисунок 6 - Поиск элемента

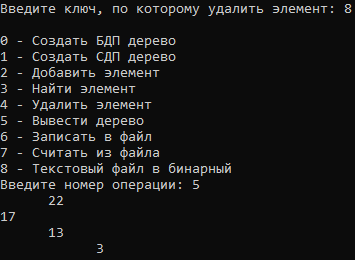


Рисунок 7 - Удаление элемента

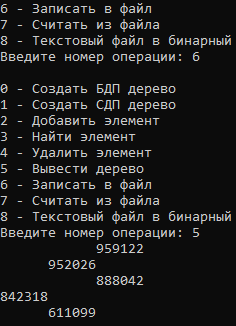


Рисунок 8 - Запись дерева в файл

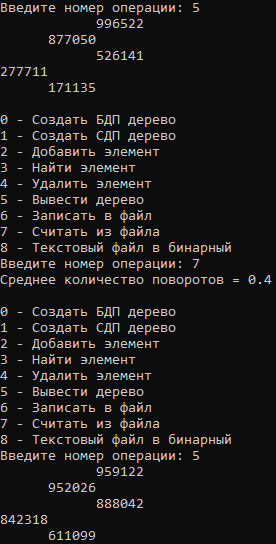


Рисунок 9 - Считывание дерева из файла

## Количество поворотов при балансировке СДП

Среднее количество поворотов при формировании дерева из двоичного файла, состоящего из 1000 узлов = 0.98999/узел.

# Отчёт по заданию 3

Результаты анализа скорости работы алгоритмов поиска на небольшом объёме данных (таблица 3).

Таблица 3 - Анализ скорости работы алгоритмов поиска

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Вид структуры | Количество элементов | Ёмкостная сложность: объем памяти для структуры | Количество выполненных сравнений | Время на поиск ключа (нс) |
| Хеш-таблица (Цепное хэширование) | 100 | Лучший: O(n)  Худший: O(2\*n) | Лучший: 1  Средний: 1  Худший: 100 | Лучший: 8400  Средний: 8400  Худший: 58300 |
| Бинарное дерево поиска | 100 | O(n) | Лучший: 1  Средний: 15  Худший: 100 | Лучший: 500  Средний: 1200  Худший: 4800 |
| Сбалансированное дерево поиска (АВЛ) | 100 | O(n) | Лучший: 7  Средний: 7  Худший: 7 | Лучший: 700  Средний: 700  Худший: 900 |

Результаты анализа скорости работы алгоритмов поиска на большом объёме данных (таблица 4).

Таблица 4 - Анализ скорости работы алгоритмов поиска

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Вид структуры | Количество элементов | Ёмкостная сложность: объем памяти для структуры | Количество выполненных сравнений | Время на поиск ключа (нс) |
| Хеш-таблица (Цепное хэширование) | 5000 | Лучший: O(n)  Худший: O(2\*n) | Лучший: 1  Средний: 2  Худший: 5000 | Лучший: 8200  Средний: 9000  Худший: 2607000 |
| Бинарное дерево поиска | 5000 | O(n) | Лучший: 1  Средний: 14  Худший: 5000 | Лучший: 1000  Средний: 1500  Худший: 250000 |
| Сбалансированное дерево поиска (АВЛ) | 5000 | O(n) | Лучший: 13  Средний: 13  Худший: 13 | Лучший: 1500  Средний: 1500  Худший: 1500 |

Вывод:

В ходе анализа скорости работы алгоритмов поиска в хеш-таблице, бинарном дереве поиска и в сбалансированном дереве поиска были построены таблицы результатов. Из таблиц результатов видно, что самым быстрым видом структуры поиска является СДП, так как у неё нет необходимости находить с помощью хэш-функции индекс элемента и отсутствует такая глубина деревьев, какая есть у БДП. Но при этом в лучшем и среднем случаях хеш-таблица выигрывает по количеству выполняемых сравнений.

# Вывод

В ходе выполнения практической работы были приобретены знания по работе с бинарным деревом поиска и сбалансированным деревом поиска, а также с бинарными файлами.

Была разработана и протестирована программа для задачи варианта №7, а также были созданы таблицы результатов работы деревьев.

# Список литературы

1. Система вопросов и ответов о программировании: <https://stackoverflow.com/>
2. Веб ресурс: <https://ru.wikipedia.org/wiki/Двоичное_дерево_поиска>