|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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 № 5** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Сбалансированные деревья поиска (СДП) и их применение для поиска данных в файле»** | |
|  | |
| Выполнил студент группы ИКБО-11-22 | Гришин А. В. |
| Принял преподаватель | Скворцова Л. А. |

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

Москва 2023

**Содержание**

[1. Цель работы 3](#_Toc153309148)

[2.1 Задание 1 3](#_Toc153309149)

[2.2 Задание 2 4](#_Toc153309150)

[2.3 Задание 3 5](#_Toc153309151)

[3 Реализация задания 1 6](#_Toc153309152)

[3.1 Реализация управления двоичным файлом 6](#_Toc153309153)

[3.1.1 Реализация структуры записи 6](#_Toc153309154)

[3.1.2 Функция 1 7](#_Toc153309155)

[3.1.3 Функция №2 8](#_Toc153309156)

[3.1.4 Функция №3 9](#_Toc153309157)

[3.2 Реализация бинарного дерева поиска 10](#_Toc153309158)

[3.2.1 Реализация структуры узла бинарного дерева поиска 10](#_Toc153309159)

[3.2.2 Функция №3 10](#_Toc153309160)

[3.2.3 Функция №4 11](#_Toc153309161)

[3.2.4 Функция №5 12](#_Toc153309162)

[3.2.5 Функция №6 13](#_Toc153309163)

[3.2.6 Функция №7 14](#_Toc153309164)

[4 Задание 2 15](#_Toc153309165)

[4.1 Реализация управления сбалансированным деревом поиска 15](#_Toc153309166)

[4.1.1 Реализация структуры записи 15](#_Toc153309167)

[4.1.2 Функция №8 15](#_Toc153309168)

[4.1.3. Функция №9 16](#_Toc153309169)

[4.1.4 Функция №10 16](#_Toc153309170)

[4.1.5 Функция №11 17](#_Toc153309171)

[4.1.6 Функция №12 17](#_Toc153309172)

[4.1.7 Функция №13 18](#_Toc153309173)

[4.1.8 Функция 14. 19](#_Toc153309174)

[4.2 Количество поворотов 20](#_Toc153309175)

[5 Задание 3. Анализ алгоритма поиска для 100, 1000, 10000 ключей 20](#_Toc153309176)

[6 Код программы 21](#_Toc153309177)

[7 Вывод 32](#_Toc153309178)

# Цель работы

- получить навыки в разработки и реализации алгоритмов управления бинарным деревом поиска и сбалансированными бинарными деревьями поиска (АВЛ – деревьями);

- получить навыки в применении файловых потоков прямого доступа к данным файла;

- получить навыки в применении сбалансированного дерева поиска для прямого доступа к записям файла.

## 2.1 Задание 1

Разработать приложение, которое использует бинарное дерево поиска (БДП) для поиска записи с ключом в файле, структура которого представлена в задании 2 вашего варианта.

1. Разработать класс (или библиотеку функций) «Бинарное дерево поиска». Тип информационной части узла дерева: ключ и ссылка на запись в файле (как в практическом задании 2). Методы: включение элемента в дерево, поиск ключа в дереве, удаление ключа из дерева, отображение дерева.

2. Разработать класс (библиотеку функций) управления файлом (если не создали в практическом задании 2). Включить методы: создание двоичного файла записей фиксированной длины из заранее подготовленных данных в текстовом файле; поиск записи в файле с использованием БДП; остальные методы по вашему усмотрению.

3. Разработать и протестировать приложение.

4. Подготовить отчет

## 2.2 Задание 2

Разработать приложение, которое использует сбалансированное дерево поиска, предложенное в варианте, для доступа к записям файла.

1. Разработать класс СДП с учетом дерева варианта. 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).

2. Разработать приложение, которое создает и управляет СДП в соответствии с заданием.

3. Выполнить тестирование.

4. Определить среднее число выполненных поворотов (число поворотов на общее число вставленных ключей) при включении ключей в дерево при формировании дерева из двоичного файла.

5. Оформить отчет

**Индивидуальное задание:**

|  |  |
| --- | --- |
| Сбалансированное дерево поиска (СДП) | Структура элемента множества (ключ – подчеркнутое поле) остальные поля представляют данные элемента |
| АВЛ | Дни рождения друзей: дата рождения, имя |

## 2.3 Задание 3

Выполнить анализ алгоритма поиска записи с заданным ключом при применении структур данных: - хеш – таблица; - бинарное дерево поиска; - СДП Требования по выполнению задания:

1. Протестировать на данных: а) небольшого объема; б) большого объема.

2. Построить хеш-таблицу из чисел файла.

3. Осуществить поиск введенного целого числа в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Оформить таблицу результатов по форме.

4. Провести анализ алгоритма поиска ключа на исследованных поисковых структурах на основе данных, представленных в таблице.

5. Оформить отчет.

# 3 Реализация задания 1

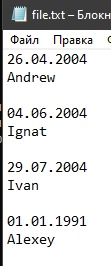


Рисунок 1 – Пример записей в текстовом файле

## 3.1 Реализация управления двоичным файлом

### 3.1.1 Реализация структуры записи

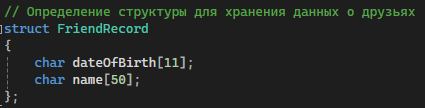


Рисунок 2 – Структура записи двоичного файла

### 3.1.2 Функция 1

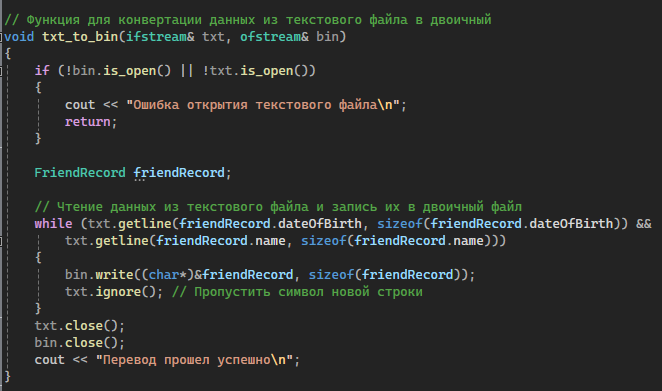


Рисунок 3 – Функция №1

Функция txt\_to\_bin:

* Прототип void txt\_to\_bin(ifstream& txt, ofstream& bin)
* Предусловие: txt – ссылка на объект потока чтения файла, bin – объект потока записи в файл.
* Постусловие: Получен двоичный файл.
* Описание: Функция txt\_to\_bin заполняет двоичный файл записями фиксированной длины с текстового файла с заранее подготовленными данными.

### 3.1.3 Функция №2

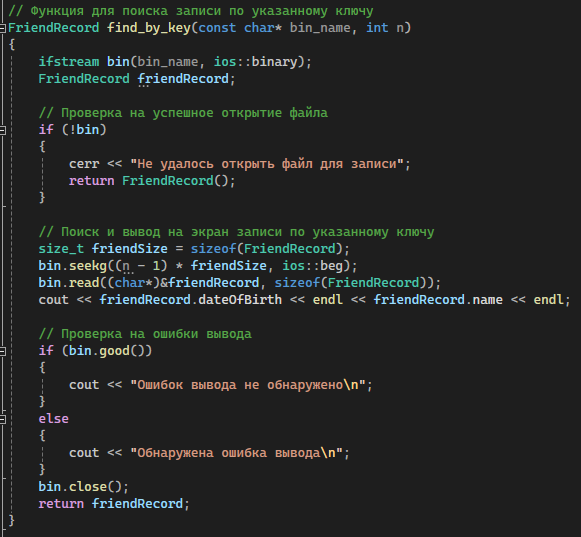


Рисунок 4 – Функция №2

Функция find\_by\_key:

* Прототип: FriendRecord find\_by\_key(const char\* bin\_name, int n)
* Предусловие: bin\_name – название двоичного файла, n – порядковый номер записи в двоичном файле.
* Постусловие: Получена искомая структура.
* Описание: Функция find\_by\_key ищет запись в двоичном файле под названием bin\_name, находит его, выводит поля на экран и возвращает его.

### 3.1.4 Функция №3

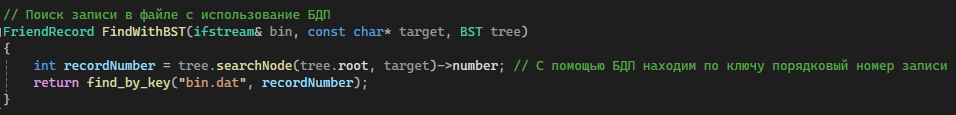


Рисунок 5 – Функция №3

Функция FindWithBST:

* Прототип: FriendRecord FindWithBST(ifstream& bin, const char\* target, BST tree)
* Предусловие: bin – объект потока чтения файла, target – ключ записи, tree – бинарное дерево поиска.
* Постусловие: Найдена искомая запись.
* Описание: Функция FindWithBST ищет в БДП узел по ключу, возврашает порядковый номер записи, затем ищет по порядковому номеру в двоичном файле и возвращает узел.

## 3.2 Реализация бинарного дерева поиска

### 3.2.1 Реализация структуры узла бинарного дерева поиска

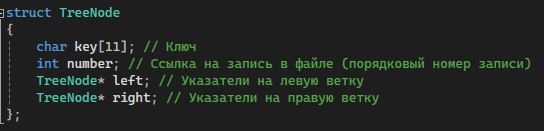


Рисунок 5 – Структура узла бинарного дерева поиска

### 3.2.2 Функция №3

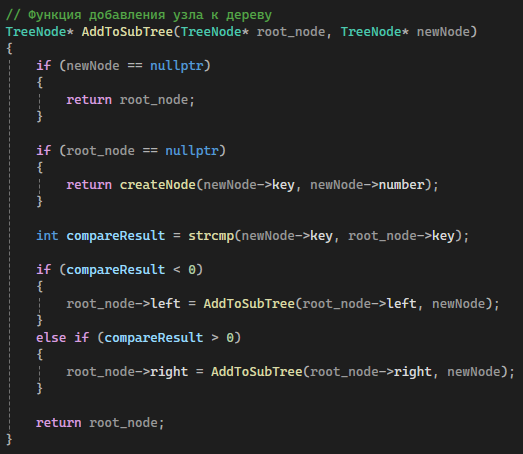


Рисунок 6 – Функция №3

Функция AddToSubTree:

* Прототип: TreeNode\* AddToSubTree(TreeNode\* root\_node, Treenode\* new Node)
* Предусловие: root\_node и newNode должны быть корректными указателями TreeNode\*.
* Постусловие: Узел newNode добавлен в бинарное дерево `tree`. Возвращает root\_node.
* Описание: Функция AddToSubTree рекурсивно добавляет узел newNode к бинарному дереву tree.

### 3.2.3 Функция №4

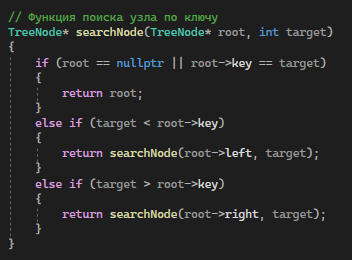


Рисунок 7 – Функция №4

Функция searchNode:

* Прототип: TreeNode\* searchNode(TreeNode\* root, const char\* target)
* Предусловие: root должен быть корректным указателем на узел дерева TreeNode\*, const char\* target – ключ искомого узла.
* Постусловие: найдено искомое слово или возвращен root.
* Описание: Функция searchNode принимает корневой узел дерева, проверяет его на наличие, далее запускает себя рекурсивно для левой и правой ветки дерева. При нахождении возвращает искомый узел, при не

нахождении – root.

* Описание: Функция AddToSubTree рекурсивно добавляет узел newNode к бинарному дереву tree.

### 3.2.4 Функция №5



Рисунок 8 – Функция №5

Функция deleteNode:

* Прототип: TreeNode\* deleteNode(TreeNode\* root, const char\* target)
* Предусловие: root должен быть корректным указателем на узел TreeNode\*. target — ключ удаляемого узла.
* Постусловие: Удачно удален узел или не удален.
* Описание: Функция deleteNode удаляет узел с словом данным в параметре target. Не удаляет если данного узла нет или дерево пустое. Делится на 2 случая: узел без потомков или один потомок; узел с двумя потомками. Возвращает root.

### 3.2.5 Функция №6

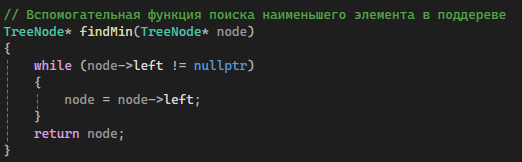


Рисунок 9 – Функция №6

Функция findMin:

* Прототип: TreeNode\* findMin(TreeNode\* node)
* Предусловие: node должен быть корректным указателем на узел TreeNode\*.
* Постусловие: Находит самый «нижний» узел в поддереве.
* Описание: Вспомогательная функция findMin проходит до самого «нижнего» узла поддерева и возвращает его. Используется в функции deleteNode при случае наличия двух потомков у удаляемого узла.

### 3.2.6 Функция №7

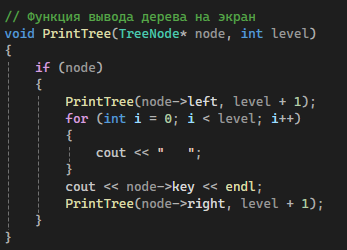


Рисунок 10 – Функции №7

Функция PrintTree:

* Прототип: void PrintTree(TreeNode\* node, int level)
* Предусловие: node должен быть корректным указателем на узел TreeNode\*, level – уровень глубины, изначально 0.
* Постусловие: Бинарное дерево поиска выведено на экран.
* Описание: Функция PrintTree рекурсивно вызывает сама себя сначала для левой ветки дерева, затем для правой. Выводит на экран «повернутое на бок» дерево.

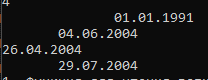


Рисунок 11 – Тестирование вывода дерева

# 4 Задание 2

## 4.1 Реализация управления сбалансированным деревом поиска

### 4.1.1 Реализация структуры записи

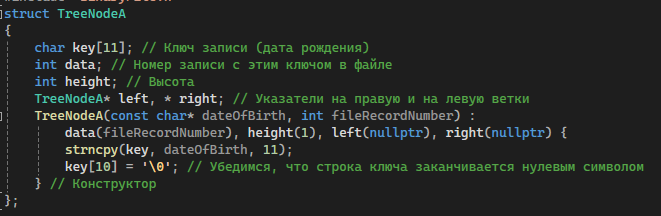


Рисунок 12 – Структура записи АВЛ дерева

### 4.1.2 Функция №8

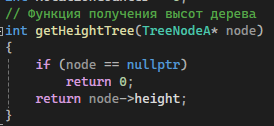


Рисунок 13 – Функция №8

Функция getHeightTree:

* + - Прототип: int getHeightTree(TreeNodeA\* node)
    - Предусловие: node - указатель на узел типа TreeNodeA.
    - Постусловие: Возвращает высоту AVL-дерева, корень которого находится в узле node. Если node равен nullptr, возвращает 0.
    - Описание: Данная функция вычисляет и возвращает высоту AVL дерева, корень которого задан узлом node.

### 4.1.3. Функция №9

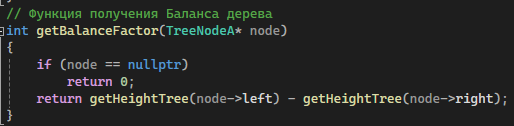


Рисунок 14 – Функция №9

Функция getBalanceFactor:

* + - Прототип: int getBalanceFactor(TreeNodeA\* node)
    - Предусловие: node - указатель на узел типа TreeNodeA.
    - Постусловие: Возвращает фактор баланса (разницу высот между левым и правым поддеревьями) AVL-дерева, корень которого находится в узле node. Если node равен nullptr, возвращает 0.
    - Описание: Данная функция вычисляет и возвращает баланс AVL-дерева, корень которого задан узлом node.

### 4.1.4 Функция №10

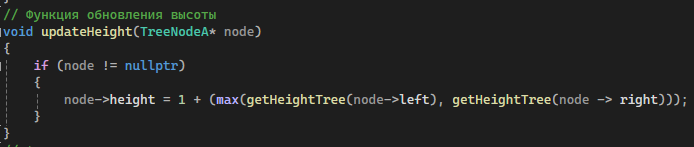


Рисунок 15 – Функция №10

Функция updateHeight:

* + - Прототип: void updateHeight(TreeNodeA\* node)
    - Предусловие: node - указатель на узел типа TreeNodeA.
    - Постусловие: Высота AVL-дерева, корень которого находится в узле node, обновляется на основе высот его левого и правого поддеревьев.
    - Описание: Данная функция обновляет высоту AVL-дерева, корень которого задан узлом node.

### 4.1.5 Функция №11

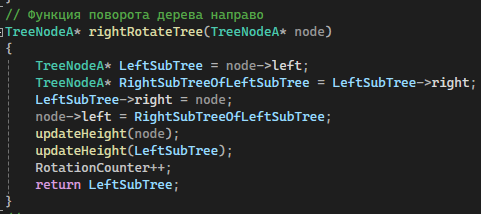


Рисунок 16 – Функция №11

Функция rightRotateTree:

* + - Прототип: TreeNodeA\* rightRotateTree(TreeNodeA\* node)
    - Предусловие: node - указатель на узел типа TreeNodeA. • Постусловие: Возвращает указатель на новый корень повернутого AVLдерева. Счетчик поворотов (RotationCounter) увеличивается.
    - Описание: Данная функция выполняет правый поворот AVL-дерева, корень которого задан узлом node.

### 4.1.6 Функция №12

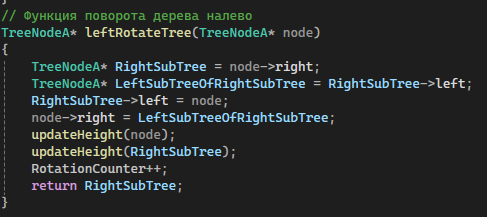


Рисунок 17 – Функция №12

Функция leftRotateTree:

* + - Прототип: TreeNodeA\* leftRotateTree(TreeNodeA\* node)
    - Предусловие: node - указатель на узел типа TreeNodeA.
    - Постусловие: Возвращает указатель на новый корень повернутого AVL дерева. Счетчик поворотов (RotationCounter) увеличивается.
    - Описание: Данная функция выполняет левый поворот AVL-дерева, корень которого задан узлом node.

### 4.1.7 Функция №13



Рисунок 18 – Функции №13

Функция AddToSubAVLTree

* + - Прототип: TreeNodeA\* AddToSubAVLTree(TreeNodeA\* node, TreeNodeA\* newnode)
    - Предусловие: node - указатель на узел типа TreeNodeA. newnode - указатель на новый узел для добавления.
    - Постусловие: Возвращает указатель на корень обновленного поддерева AVL.
    - Описание: Данная функция добавляет новый узел в поддерево AVL, корень которого задан узлом node.

### 4.1.8 Функция 14.



Рисунок 19 – Функции №14

Функция DeleteNodeFromAVL:

* + - Прототип: TreeNodeA\* DeleteNodeFromAVL(TreeNodeA\* root, const char\* key)
    - Предусловие: root - указатель на корень AVL-дерева. key - ключ узла для удаления.
    - Постусловие: Возвращает указатель на корень обновленного AVL дерева после удаления.
    - Описание: Данная функция удаляет узел с заданным ключом из AVL дерева.

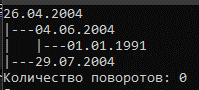


Рисунок 20 – Тестирование вывода рандомизированного дерева

## 4.2 Количество поворотов

Число поворотов на общее число вставленных ключей при файле с 4-мя записями равно 0.

# 5 Задание 3. Анализ алгоритма поиска для 100, 1000, 10000 ключей

|  |  |  |  |
| --- | --- | --- | --- |
| Вид поисковой структуры | Количество элементов, загруженных в структуру в момент выполнения поиска | Емкостная сложность: объем памяти для структуры | Количество выполненных сравнений, время на поиск ключа в структуре (микросекунды) |
| БДП | 100 | 1000 | 10000 | 0(n) | 15, 1234 | 18,  2203 | 24, 2435 |
| АВЛ | 100 | 1000 | 10000 | 0(n) | 21, 9480 |
| Хеш-таблица | 100 | 1000 | 10000 | 0(n) | 1, 1023 |

# 6 Код программы

*Листинг BinaryFile.h*

|  |
| --- |
| #ifndef \_3\_BINARYFILE\_H  #define \_3\_BINARYFILE\_H  #include <iostream>  #include <fstream>  #include <string>  using namespace std;  // Определение структуры для хранения данных о друзьях  struct FriendRecord  {  char dateOfBirth[11];  char name[50];  };  // Функция для конвертации данных из текстового файла в двоичный  void txt\_to\_bin(ifstream& txt, ofstream& bin)  {  if (!bin.is\_open() || !txt.is\_open())  {  cout << "Ошибка открытия текстового файла\n";  return;  }  FriendRecord friendRecord;  // Чтение данных из текстового файла и запись их в двоичный файл  while (txt.getline(friendRecord.dateOfBirth, sizeof(friendRecord.dateOfBirth)) &&  txt.getline(friendRecord.name, sizeof(friendRecord.name)))  {  bin.write((char\*)&friendRecord, sizeof(friendRecord));  txt.ignore(); // Пропустить символ новой строки  }  txt.close();  bin.close();  cout << "Перевод прошел успешно\n";  }  // Функция для конвертации данных из двоичного файла в текстовый  void bin\_to\_txt(ifstream& binn, ofstream& txtt)  {  FriendRecord friendRecord;  if (!txtt.is\_open())  {  cout << "Ошибка открытия текстового файла\n";  return;  }  // Чтение данных из двоичного файла и запись их в текстовый файл  while (binn.read((char\*)&friendRecord, sizeof(FriendRecord)))  {  txtt << friendRecord.dateOfBirth << endl << friendRecord.name << endl;  }  // Проверка на ошибки чтения из двоичного файла  if (!binn.eof() && binn.fail())  {  cout << "Ошибка чтения из двоичного файла" << endl;  }  cout << "Перевод прошел успешно\n";  }  // Функция для вывода данных из двоичного файла на экран  void print\_from\_bin(ifstream& bin)  {  FriendRecord friendRecord;  bin.read((char\*)&friendRecord, sizeof(FriendRecord));  // Вывод данных из двоичного файла на экран  while (!bin.eof())  {  cout << friendRecord.dateOfBirth << endl << friendRecord.name << endl;  bin.read((char\*)&friendRecord, sizeof(FriendRecord));  }  bin.close();  }  // Функция для удаления записи по указанному ключу  void delete\_by\_key(const char\* bin\_name, int n)  {  fstream bin(bin\_name, ios::binary | ios::in);  // Проверка на успешное открытие файла  if (!bin.is\_open())  {  cerr << "Ошибка открытия файла\n";  return;  }  fstream temp("temp\_file.dat", ios::binary | ios::out);  // Проверка на успешное открытие временного файла  if (!temp.is\_open())  {  cerr << "Ошибка открытия временного файла\n";  bin.close();  return;  }  FriendRecord friendRecord;  int key = 1;  // Копирование данных из исходного файла во временный, исключая запись с указанным ключом  while (bin.read((char\*)&friendRecord, sizeof(FriendRecord)))  {  if (key != n)  {  temp.write((char\*)&friendRecord, sizeof(FriendRecord));  }  key++;  }  bin.close();  temp.close();  // Удаление исходного файла и переименование временного файла  remove(bin\_name);  rename("temp\_file.dat", bin\_name);  }  // Функция для поиска записи по указанному ключу  FriendRecord find\_by\_key(const char\* bin\_name, int n)  {  ifstream bin(bin\_name, ios::binary);  FriendRecord friendRecord;  // Проверка на успешное открытие файла  if (!bin)  {  cerr << "Не удалось открыть файл для записи";  return FriendRecord();  }  // Поиск и вывод на экран записи по указанному ключу  size\_t friendSize = sizeof(FriendRecord);  bin.seekg((n - 1) \* friendSize, ios::beg);  bin.read((char\*)&friendRecord, sizeof(FriendRecord));  cout << friendRecord.dateOfBirth << endl << friendRecord.name << endl;  // Проверка на ошибки вывода  if (bin.good())  {  cout << "Ошибок вывода не обнаружено\n";  }  else  {  cout << "Обнаружена ошибка вывода\n";  }  bin.close();  return friendRecord;  }  #endif //\_3\_BINARYFILE\_H |

*Листинг BST.h*

|  |
| --- |
| #pragma once  #include <iostream>  #include "BinaryFile.h"  using namespace std;  struct TreeNode  {  char key[11]; // Ключ  int number; // Ссылка на запись в файле (порядковый номер записи)  TreeNode\* left; // Указатели на левую ветку  TreeNode\* right; // Указатели на правую ветку  };  struct BST  {  TreeNode\* root = nullptr;  // Функция создания узла  TreeNode\* createNode(const char\* new\_key, int new\_number)  {  TreeNode\* node = new TreeNode;  strncpy(node->key, new\_key, sizeof(node->key) - 1);  node->key[sizeof(node->key) - 1] = '\0';  node->number = new\_number;  node->left = nullptr;  node->right = nullptr;  return node;  }  // Функция добавления узла к дереву  TreeNode\* AddToSubTree(TreeNode\* root\_node, TreeNode\* newNode)  {  if (newNode == nullptr)  {  return root\_node;  }  if (root\_node == nullptr)  {  return createNode(newNode->key, newNode->number);  }  int compareResult = strcmp(newNode->key, root\_node->key);  if (compareResult < 0)  {  root\_node->left = AddToSubTree(root\_node->left, newNode);  }  else if (compareResult > 0)  {  root\_node->right = AddToSubTree(root\_node->right, newNode);  }  return root\_node;  }  //Функция добавления поддерева к дереву  void AddSubTreeToTree(BST& tree, TreeNode\* subtreeRoot)  {  if (subtreeRoot == nullptr) {  return;  }  tree.root = AddToSubTree(tree.root, subtreeRoot);  }  // Функция поиска узла по ключу  TreeNode\* searchNode(TreeNode\* root, const char\* target)  {  if (root == nullptr || strcmp(root->key, target) == 0)  {  return root;  }  else if (strcmp(target, root->key) < 0)  {  return searchNode(root->left, target);  }  else if (strcmp(target, root->key) > 0)  {  return searchNode(root->right, target);  }  return nullptr;  }  // Функция удаления узла по ключу  TreeNode\* deleteNode(TreeNode\* root, const char\* target)  {  if (root == nullptr)  {  return root;  }  int compareResult = strcmp(target, root->key);  if (compareResult < 0)  {  root->left = deleteNode(root->left, target);  }  else if (compareResult > 0)  {  root->right = deleteNode(root->right, target);  }  else  {  // Случай 1: Узел без потомков или с одним потомком  if (root->left == nullptr) {  TreeNode\* temp = root->right;  delete root;  return temp;  }  else if (root->right == nullptr) {  TreeNode\* temp = root->left;  delete root;  return temp;  }  // Случай 2: Узел с двумя потомками  // Найти наименьший элемент в правом поддереве (минимум в правом поддереве)  TreeNode\* temp = findMin(root->right);  // Скопировать значение минимального элемента в текущий узел  strcpy(root->key, temp->key);  // Удалить минимальный элемент из правого поддерева  root->right = deleteNode(root->right, temp->key);  }  return root;  }  // Вспомогательная функция поиска наименьшего элемента в поддереве  TreeNode\* findMin(TreeNode\* node)  {  while (node->left != nullptr)  {  node = node->left;  }  return node;  }  // Функция вывода дерева на экран  void PrintTree(TreeNode\* node, int level)  {  if (node)  {  PrintTree(node->left, level + 1);  for (int i = 0; i < level; i++)  {  cout << " ";  }  cout << node->key << endl;  PrintTree(node->right, level + 1);  }  }  // Функция построения дерева из записей двоичного файла  void BuildFromeBinFile(const char\* binfile, BST& tree)  {  ifstream bin(binfile, ios::binary);  if (bin.is\_open())  {  FriendRecord record;  int recordNumber = 0;  while (bin.read(reinterpret\_cast<char\*>(&record), sizeof(FriendRecord)))  {  TreeNode\* newNode = tree.createNode(record.dateOfBirth, recordNumber);  tree.AddSubTreeToTree(tree, newNode);  recordNumber++;  }  bin.close();  }  else  {  cerr << "Ошибка в открытии двоичного файла\n";  }  }  }; |

*Листинг FileWithBST.h*

|  |
| --- |
| #pragma once  #include "BinaryFile.h"  #include "BST.h"  // Поиск записи в файле с использование БДП  FriendRecord FindWithBST(ifstream& bin, const char\* target, BST tree)  {  int recordNumber = tree.searchNode(tree.root, target)->number; // С помощью БДП находим по ключу порядковый номер записи  return find\_by\_key("bin.dat", recordNumber);  } |

*Листинг AVL.h*

|  |
| --- |
| #pragma once  #include <iostream>  #include <cstringt.h>  #include <regex>  #include <string>  #include <fstream>  #include "BinaryFile.h"  struct TreeNodeA  {  char key[11]; // Ключ записи (дата рождения)  int data; // Номер записи с этим ключом в файле  int height; // Высота  TreeNodeA\* left, \* right; // Указатели на правую и на левую ветки  TreeNodeA(const char\* dateOfBirth, int fileRecordNumber) :  data(fileRecordNumber), height(1), left(nullptr), right(nullptr) {  strncpy(key, dateOfBirth, 11);  key[10] = '\0'; // Убедимся, что строка ключа заканчивается нулевым символом  } // Конструктор  };  extern int RotationCounter;  struct AVL  {  TreeNodeA\* root = nullptr;  void deleteTree() {  // Вызываем вспомогательную функцию для удаления всех узлов  deleteTreeHelper(root);  root = nullptr; // Устанавливаем корень в nullptr  }  // Рекурсивная функция для удаления узлов дерева  void deleteTreeHelper(TreeNodeA\* node) {  if (node == nullptr) {  return;  }  // Сначала рекурсивно удаляем левое и правое поддерево  deleteTreeHelper(node->left);  deleteTreeHelper(node->right);  // Затем удаляем текущий узел  delete node;  }  };  int RotationCounter = 0;  // Функция получения высот дерева  int getHeightTree(TreeNodeA\* node)  {  if (node == nullptr)  return 0;  return node->height;  }  // Функция получения Баланса дерева  int getBalanceFactor(TreeNodeA\* node)  {  if (node == nullptr)  return 0;  return getHeightTree(node->left) - getHeightTree(node->right);  }  // Функция обновления высоты  void updateHeight(TreeNodeA\* node)  {  if (node != nullptr)  {  node->height = 1 + (max(getHeightTree(node->left), getHeightTree(node -> right)));  }  }  // Функция поворота дерева направо  TreeNodeA\* rightRotateTree(TreeNodeA\* node)  {  TreeNodeA\* LeftSubTree = node->left;  TreeNodeA\* RightSubTreeOfLeftSubTree = LeftSubTree->right;  LeftSubTree->right = node;  node->left = RightSubTreeOfLeftSubTree;  updateHeight(node);  updateHeight(LeftSubTree);  RotationCounter++;  return LeftSubTree;  }  // Функция поворота дерева налево  TreeNodeA\* leftRotateTree(TreeNodeA\* node)  {  TreeNodeA\* RightSubTree = node->right;  TreeNodeA\* LeftSubTreeOfRightSubTree = RightSubTree->left;  RightSubTree->left = node;  node->right = LeftSubTreeOfRightSubTree;  updateHeight(node);  updateHeight(RightSubTree);  RotationCounter++;  return RightSubTree;  }  TreeNodeA\* AddToSubAVLTree(TreeNodeA\* node, TreeNodeA\* newNode);  //Функция добавления узла в АВЛ дерево  void AddToAVLTree(AVL& tree, const char\* k, int d)  {  TreeNodeA\* node = new TreeNodeA(k, d);  tree.root = AddToSubAVLTree(tree.root, node);  }  // Функция добавления узла в АВЛ поддерево  TreeNodeA\* AddToSubAVLTree(TreeNodeA\* node, TreeNodeA\* newnode)  {  if (node == nullptr)  {  return newnode;  }  if (newnode->key < node->key)  node->left = AddToSubAVLTree(node->left, newnode);  else if (newnode->key > node->key)  node->right = AddToSubAVLTree(node->right, newnode);  else  {  delete newnode;  return node;  }  updateHeight(node);  int BalanceFactor = getBalanceFactor(node);  if (BalanceFactor > 1)  {  if (newnode->key < node->left->key)  return rightRotateTree(node);  if (newnode->key > node->left->key)  {  node->left = leftRotateTree(node->left);  return rightRotateTree(node);  }  }  if (BalanceFactor < -1)  {  if (newnode->key > node->right->key)  return leftRotateTree(node);  if (newnode->key < node->right->key)  {  node->right = rightRotateTree(node->right);  return leftRotateTree(node);  }  }  return node;  }  // Функция вывода структурированного АВЛ поддерева(алгоритв КЛП)  void PrintSubAVLTreeStructure(TreeNodeA\* node, int depth, bool isRoot = false)  {  if (node == nullptr) return;  if (isRoot) {  printf("%s(%i,%i)\n", node->key, node->data, getBalanceFactor(node));  }  else  {  for (int i = 0; i < depth; i++)  printf("| ");  printf("\b\b\b---%s(%i,%i)\n", node->key, node->data, getBalanceFactor(node));  }  PrintSubAVLTreeStructure(node->left, depth + 1);  PrintSubAVLTreeStructure(node->right, depth + 1);  }  // Функция вывода структурированного АВЛ дерева(алгоритм КЛП)  void PrintAVLTreeStructure(AVL tree)  {  PrintSubAVLTreeStructure(tree.root, 0, true);  }  // Функция поиска в АВЛ поддереве  int SearchInAVLSubTree(TreeNodeA\* node, const char\* key) {  if (node == nullptr)  return -1;  int compareResult = strcmp(key, node->key);  if (compareResult == 0)  return node->data;  else if (compareResult < 0)  return SearchInAVLSubTree(node->left, key);  else  return SearchInAVLSubTree(node->right, key);  }  // Функция поиска в АВЛ дереве  int SearchInAVLTree(AVL tree, const char\* key)  {  return SearchInAVLSubTree(tree.root, key);  }  // Функция для поиска замещающего узла  TreeNodeA\* SubstituteNodeSearch(TreeNodeA\* node) {  if (node->left == nullptr) {  return node;  }  return SubstituteNodeSearch(node->left);  }  // Функция для удаления узла из АВЛ дерева  TreeNodeA\* DeleteNodeFromAVL(TreeNodeA\* root, const char\* key) {  if (root == nullptr) return root;  int compareResult = strcmp(key, root->key);  if (compareResult < 0) root->left = DeleteNodeFromAVL(root->left, key);  else if (compareResult > 0) root->right = DeleteNodeFromAVL(root->right, key);  else {  if (root->left == nullptr || root->right == nullptr) {  TreeNodeA\* temp;  if (root->left != nullptr)  temp = root->left;  else  temp = root->right;  if (temp == nullptr) {  temp = root;  root = nullptr;  }  else {  \*root = \*temp;  }  delete temp;  }  else {  TreeNodeA\* temp = SubstituteNodeSearch(root->right);  strncpy(root->key, temp->key, 11);  root->data = temp->data;  root->right = DeleteNodeFromAVL(root->right, temp->key);  }  }  if (root == nullptr) return root;  updateHeight(root);  int BalanceFactor = getBalanceFactor(root);  if (BalanceFactor > 1) {  if (getBalanceFactor(root->left) >= 0) return rightRotateTree(root);  else {  root->left = leftRotateTree(root->left);  return rightRotateTree(root);  }  }  if (BalanceFactor < -1) {  if (getBalanceFactor(root->right) <= 0) return leftRotateTree(root);  else {  root->right = rightRotateTree(root->right);  return leftRotateTree(root);  }  }  return root;  }  void BuildAVLFromBinFile(AVL& avl, const char\* binfile) {  ifstream bin(binfile, ios::binary);  if (!bin.is\_open()) {  cerr << "Ошибка при открытии бинарного файла" << endl;  return;  }  FriendRecord record;  int recordNumber = 0;  while (bin.read(reinterpret\_cast<char\*>(&record), sizeof(FriendRecord))) {  AddToAVLTree(avl, record.dateOfBirth, recordNumber);  }  bin.close();  } |

# 7 Вывод

Мы получили навыки в разработке и реализации алгоритмов управления бинарным деревом поиска и AVL-деревом, получили навыки применения файловых потоков прямого доступа к данным файла, получили навыки в применении сбалансированного дерева поиска для прямого доступа к записям в файле.