|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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-07-21 | Дамарад Д.В. |
| Принял преподаватель | Скворцова Л.А. |

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

**СОДЕРЖАНИЕ**

[**ЦЕЛЬ РАБОТЫ 3**](#_Toc120644248)

[**ПОСТАНОВКА ЗАДАЧИ 3**](#_Toc120644249)

[**РЕШЕНИЕ 5**](#_Toc120644250)

[**Задание 1 5**](#_Toc120644251)

[**Задание 2 12**](#_Toc120644252)

[**Интерфейс 19**](#_Toc120644253)

[**Тестирование 20**](#_Toc120644254)

[**Задание 3 30**](#_Toc120644255)

[**ЗАКЛЮЧЕНИЕ 31**](#_Toc120644256)

[**ПОЛНЫЙ КОД ПРОГРАММЫ 32**](#_Toc120644257)

# **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.

# **ПОСТАНОВКА ЗАДАЧИ**

**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.

**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.

|  |  |
| --- | --- |
| Сбалансированное дерево поиска (СДП) | Структура элемента множества (ключ – подчеркнутое поле) остальные поля представляют данные элемента |
| АВЛ | Товар: название, код – шестиразрядное число, завод изготовитель, цена, страна (название). |

**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.

1. Построить хеш-таблицу из чисел файла.
2. Осуществить поиск введенного целого числа в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице.
3. Провести анализ алгоритма поиска ключа на исследованных поисковых структурах на основе данных, представленных в таблице.

# **РЕШЕНИЕ**

# **Задание 1**

Структура элемента множества:

struct Product {

char name[20] = "";

char manufacturer[20] = "";

char country[20] = "";

int price = 0;

int code = 0;

};

Функция printRecord() выводит содержимое переданной структуры Record в консоль:

void printRecord(Product product) {

cout << product.name << " " << product.code << " " << product.manufacturer << " " << product.price << " " << product.country << endl;

}

Функция ConvertTextToBin() генерирует бинарный файл из текстового. Для перевода данных из текстового файла в бинарный необходимо открыть поток на чтение из текстового файла и поток на запись в бинарный файл. Затем в цикле, пока поток чтения не достигнет конца, считываем значения из текстового файла и записываем с помощью метода write данные в бинарный файл:

void ConvertTextToBin(string bin\_file\_name, string txt\_file\_name) {

ifstream fin(txt\_file\_name, ios::in);

if (fin.is\_open()) {

ofstream fout(bin\_file\_name, ios::out | ios::binary);

Product product;

while (!fin.eof()) {

fin >> product.name >> product.code >> product.manufacturer >> product.price >> product.country;

fout.write((char\*)&product, sizeof(Product));

}

fout.close();

}

else {

cout << "File not found" << endl;

}

fin.close();

}

Функция OutputBinFile() выводит данные бинарного файла. Для вывода данных из бинарного файла откроем поток для чтения и будем считывать, пока это возможно, значения в переменную типа структуры варианта, затем ее выводить:

void OutputBinFile(string bin\_file\_name) {

ifstream fin(bin\_file\_name, ios::in | ios::binary);

Product product;

if (fin.is\_open()) {

while (fin.read((char\*)&product, sizeof(Product))) {

cout << product.name << " " << product.code << " " << product.manufacturer << " " << product.price << " " << product.country << "\n";

}

}

else {

cout << "File not found" << endl;

}

fin.close();

}

Функция DirectAccess() принимает на вход имя файла и номер записи, осуществляет прямой доступ к записи файла по номеру при помощи функций seekg (используется для установки указателя на необходимую строку), read (используется для чтения данных и записи их в структуру):

Product DirectAccess(string bin\_file\_name, int number) {

ifstream binFile(bin\_file\_name, ios::binary | std::ios::in);

if (!binFile.is\_open()) {

cout << "Файла с таким именем не существует" << endl;

}

binFile.seekg(0, ios::end);

long long size = binFile.tellg();

binFile.seekg(0, std::ios::beg);

if ((PRODUCT\_SIZE \* sizeof(char)) \* number > size) {

binFile.close();

return {};

}

binFile.seekg((PRODUCT\_SIZE)\*number);

Product product;

binFile.read((char\*)&product, sizeof(product));

binFile.close();

return product;

}

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

Для управления бинарным деревом используется класс BinaryTree:

class BinaryTree {

private:

struct Node {

Product value;

int key = -1;

int index = -1;

Node\* left = nullptr;

Node\* right = nullptr;

};

Node\* head;

void print(Node\* node, string space = "");

Node\* findMinNode(Node\* node, Node\* parent, int left);

public:

BinaryTree();

bool createTree(string bin\_file\_name);

void addNode(Product product, int index);

int findNode(int key);

bool removeNode(int key);

void printTree();

};

Конструктор BinaryTree инициализирует корень дерева:

BinaryTree::BinaryTree() {

head = new Node();

}

Функция createTree() строит бинарное дерево по бинарному файлу при помощи итеративного добавления элементов в дерево. Для создания дерева открываем поток на чтение бинарного файла. Если файл открыт, то пробегаемся по всем его элементам, одновременно сохраняя ключ и позицию каждой записи и передаем эти значение в метод addNode:

bool BinaryTree::createTree(string bin\_file\_name) {

ifstream fin;

fin.open(bin\_file\_name, ios::binary | ios::in);

Product product;

if (fin.is\_open()) {

int index = 0;

while (fin.read((char\*)&product, PRODUCT\_SIZE)) {

addNode(product, index);

index++;

}

fin.close();

return true;

}

return false;

}

Метод addNode() добавляет элемент в дерево. Для добавления узла в бесконечном цикле обрабатываем временную переменную, отвечающую за текущую вершину. Каждую итерацию проверяем отсутствия значения у вершины в случае, если вершина пуста, добавляем в нее запись. Если вершина не пуста, то сравниваем значение текущей вершины со значением записи, если значение вершины больше значения в записи, то придаем временной переменной значение левого потомка, иначе – правого:

void BinaryTree::addNode(Product product, int index) {

Node\* temp = head;

while (true) {

if (temp->value.code==0) {

temp->value = product;

temp->index = index;

break;

}

else if (temp->value.code < product.code) {

if (temp->right == nullptr) {

temp->right = new Node;

}

temp = temp->right;

}

else if (temp->value.code > product.code) {

if (temp->left == nullptr) {

temp->left = new Node;

}

temp = temp->left;

}

}

}

Метод findNode() ищет запись по ключу в дереве. Для нахождения записи по ключу в дереве в бесконечном цикле первым делом проверяем узел на null, если он указывает на null, то возвращаем -1 позицию. Если узел не равен null, то проверяем значение словаря из узла на сходство с ключом, если они совпадают, то возвращаем индекс этой записи, в противном случае в зависимости от значения ключа переходим к левому или правому потомку:

int BinaryTree::findNode(int key) {

Node\* temp = head;

while (true) {

if (temp == nullptr) {

return -1;

}

else if (temp->value.code == key) {

return temp->index;

}

else if (temp->value.code < key) {

temp = temp->right;

}

else if (temp->value.code > key) {

temp = temp->left;

}

}

return -1;

}

Функция findMinNode() находит наименьший узел дерева и возвращает указатель на него. Для нахождения минимальной вершины каждый раз переходим в левую вершину, если она последняя, то стираем у родителя ее адрес и возвращаем эту вершину:

BinaryTree::Node\* BinaryTree::findMinNode(Node\* node, Node\* parent, int left) {

if (node->left == nullptr) {

parent->left = nullptr;

return node;

}

return findMinNode(node->left, node, true);

}

Функция removeNode() удаляет узел из дерева. Для удаления записи из дерева первым делом проверяем присутствует ли эта запись в дереве, если присутствует, то добираемся до нее. После ее нахождения проверяем, если это лист, то удаляем его без каких либо проблем, если это вершина содержащая одного потомка, то меняем ее на этого потомка, если это вершина с двумя потомками, то находим минимальную вершину со стороны правого потомка с помощью метода findMinNode и присваиваем ему потомков удаляемой вершины, удаляем саму вершину и присваиваем новую вершину в соответствующую позицию предку:

bool BinaryTree::removeNode(int key) {

Node\* temp = head;

Node\* parent = head;

int flag = 0;

if (findNode(key) == -1) {

return false;

}

while (true) {

if (temp->value.code == key) {

if (flag) parent->right = nullptr;

else parent->left = nullptr;

if (temp->left == nullptr && temp->right == nullptr) {

delete temp;

}

else if (temp->left == nullptr) {

parent->right = temp->right;

delete temp;

}

else if (temp->right == nullptr) {

parent->left = temp->left;

delete temp;

}

else {

Node\* res = findMinNode(temp->right, temp, true);

res->left = temp->left;

res->right = temp->right;

delete temp;

if (flag) parent->right = res;

else parent->left = res;

}

return true;

}

else if (temp->value.code < key) {

flag = 1;

parent = temp;

temp = temp->right;

}

else {

flag = 0;

parent = temp;

temp = temp->left;

}

}

return false;

}

Метод print() использует прямой обход и выводит информацию о записи в узле:

void BinaryTree::print(Node\* node, string space) {

if (node->right != nullptr) {

print(node->right, space + " ");

}

cout << space << node->value.name << " " << node->value.code << " " << node->value.manufacturer << " " << node->value.price << " " << node->value.country << endl;

if (node->left != nullptr) {

print(node->left, space + " ");

}

}

Метод printTree() выводит дерево. Для вывода дерева ключей и индексов их в бинарном файле используем метод printTree() доступный для пользователя, который в свою очередь только вызывает метод print, недоступный для пользователя:

void BinaryTree::printTree() {

print(head, "");

}

# **Задание 2**

Сбалансированное бинарное дерево поиска — это бинарное дерево поиска с логарифмической высотой. Данное определение скорее идейное, чем строгое. Строгое определение оперирует разницей глубины самого глубокого и самого неглубокого листа (в AVL-деревьях) или отношением глубины самого глубокого и самого неглубокого листа (в красно-черных деревьях). В сбалансированном бинарном дереве поиска операции поиска, вставки и удаления выполняются за логарифмическое время (так как путь к любому листу от корня не более логарифма). В вырожденном случае несбалансированного бинарного дерева поиска, например, когда в пустое дерево вставлялась отсортированная последовательность, дерево превратится в линейный список, и операции поиска, вставки и удаления будут выполняться за линейное время. Поэтому балансировка дерева крайне важна. Технически балансировка осуществляется поворотами частей дерева при вставке нового элемента, если вставка данного элемента нарушила условие сбалансированности.

АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. Узел дерева хранит ключ, значение, указатели на поддеревья, а также разницу высот левого и правого поддерева. Когда происходит разбалансировка дерева, выполняется специальный алгоритм – левый или правый поворот, чтобы сбалансировать дерево, не потеряв свойств соединенных элементов.

Для управления бинарным деревом используется класс AVLtree:

class AVLtree {

private:

struct Node {

Product value;

int key = -1;

int index = -1;

Node\* left = nullptr;

Node\* right = nullptr;

int height = -1;

};

int size = -1;

int height(Node\* node);

int height\_diff(Node\* node);

void fixheight(Node\* node);

Node\* rotateright(Node\* p);

Node\* rotateleft(Node\* q);

Node\* balance(Node\* node);

Node\* findMinNode(Node\* node);

Node\* removeMinNode(Node\* node);

void print(Node\* node, string space);

public:

Node\* head = nullptr;

int rotations = 0;

bool createAVLTree(string bin\_file\_name);

Node\* addNode(Node\* p, Product pr, int index);

Node\* removeNode(Node\* p, int k);

int findNode(int k);

void printTree();

};

Функция createAVLTree() строит бинарное дерево по бинарному файлу при помощи итеративного добавления элементов в дерево. Для создания дерева открываем поток на чтение бинарного файла. Если файл открыт, то пробегаемся по всем его элементам, одновременно сохраняя ключ и позицию каждой записи и передаем эти значение в метод addNode:

bool AVLtree::createAVLTree(string bin\_file\_name) {

ifstream fin;

fin.open(bin\_file\_name, ios::binary | ios::in);

Product product;

int index = 0;

if (fin.is\_open()) {

int index = 0;

while (fin.read((char\*)&product, PRODUCT\_SIZE)) {

head = addNode(head, product, index);

index++;

}

fin.close();

return true;

}

return false;

}

Метод height() определяет высоту дерева:

int AVLtree::height(Node\* node) {

if (node) {

return node->height;

}

else {

return 0;

}

}

Метод height\_diff() возвращает разницу высот дочерних поддеревьев узла, переданного в метод:

int AVLtree::height\_diff(Node\* node) {

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

}

Метод fixheight() определяет правильное значение поля height, используя вспомогательный метод для избежания обращения к нулевым указателям:

void AVLtree::fixheight(Node\* node) {

int hl = height(node->left);

int hr = height(node->right);

if (hl > hr) {

node->height = hl + 1;

}

else {

node->height = hr + 1;

}

}

Метод rotateright() принимает на вход корень дерева и выполняет правый поворот, обновляя высоты узлов:

AVLtree::Node\* AVLtree::rotateright(Node\* p) {

Node\* q = p->left;

p->left = q->right;

q->right = p;

fixheight(p);

fixheight(q);

rotations++;

return q;

}

Метод rotateleft() принимает на вход корень дерева и выполняет левый поворот, обновляя высоты узлов :

AVLtree::Node\* AVLtree::rotateleft(Node\* q) {

Node\* p = q->right;

q->right = p->left;

p->left = q;

fixheight(q);

fixheight(p);

rotations++;

return p;

}

Метод balance() принимает на вход корень дерева и при помощи определения разницы высот выполняет один из больших поворотов дерева: левый или правый. В результате дерево балансируется и возвращается указатель:

AVLtree::Node\* AVLtree::balance(Node\* node) {

fixheight(node);

if (height\_diff(node) == 2) {

if (height\_diff(node->right) < 0) {

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

}

return rotateleft(node);

}

if (height\_diff(node) == -2) {

if (height\_diff(node->left) > 0) {

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

}

return rotateright(node);

}

return node;

}

Метод addNode() принимает на вход указатель на корень, ключ и значение. Если указатель не существует, то инициализируем его, иначе обойдем дерево в глубину, сравнивая ключи, добавляем элемент и вернем сбалансированную версию дерева:

AVLtree::Node\* AVLtree::addNode(Node\* node, Product product, int index) {

if (!node) {

node = new Node;

node->value = product;

node->key = product.code;

node->index = index;

node->left = node->right = nullptr;

node->height = 1;

size++;

return node;

}

if (product.code < node->key) {

node->left = addNode(node->left, product, index);

}

else {

node->right = addNode(node->right, product, index);

}

return balance(node);

}

Метод findMinNode() принимает на вход корень дерева, находит наименьший узел дерева и возвращает указатель на него:

AVLtree::Node\* AVLtree::findMinNode(Node\* node) {

if (node->left) {

return findMinNode(node->left);

}

else {

return node;

}

}

Метод removeMinNode() принимает на вход корень дерева, находит наименьший узел дерева, удаляет его и балансирует дерево:

AVLtree::Node\* AVLtree::removeMinNode(Node\* node) {

if (node->left == 0) {

return node->right;

}

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

return balance(node);

}

Метод removeNode() принимает на вход корень дерева, ключ. Находит узел p с заданным ключом k (если не находим, то делать ничего не надо), в поддереве находит узел min с наименьшим ключом и заменяет удаляемый узел p на найденный узел min, возвращает сбалансированное дерево:

AVLtree::Node\* AVLtree::removeNode(Node\* node, int k) {

if (!node) {

return nullptr;

}

if (k < node->key) {

node->left = removeNode(node->left, k);

}

else if (k > node->key) {

node->right = removeNode(node->right, k);

}

else {

Node\* q = node->left;

Node\* r = node->right;

delete node;

if (!r) {

return q;

}

Node\* min = findMinNode(r);

min->right = removeMinNode(r);

min->left = q;

return balance(min);

}

return balance(node);

}

Метод findNode() ищет запись по ключу в дереве. Для нахождения записи по ключу в дереве в бесконечном цикле первым делом проверяем узел на null, если он указывает на null, то возвращаем -1 позицию. Если узел не равен null, то проверяем значение словаря из узла на сходство с ключом, если они совпадают, то возвращаем индекс этой записи, в противном случае в зависимости от значения ключа переходим к левому или правому потомку:

int AVLtree::findNode(int k) {

Node\* node = head;

while (true) {

if (node == nullptr) {

return -1;

}

if (k < node->key) {

node = node->left;

}

else if (k > node->key) {

node = node->right;

}

else {

return node->index;

}

}

return -1;

}

Метод print() использует прямой обход и выводит информацию о записи в узле:

void AVLtree::print(Node\* node, string space) {

if (node->right != nullptr) {

print(node->right, space + " ");

}

cout << space << node->value.name << " " << node->value.code << " " << node->value.manufacturer << " " << node->value.price << " " << node->value.country << " " << node->value.country << "(" << node->index << ")" << endl;

//cout << space << node->key << " " << node->index << endl;

if (node->left != nullptr) {

print(node->left, space + " ");

}

}

Метод printTree() выводит дерево. Для вывода дерева ключей и индексов их в бинарном файле используем метод printTree() доступный для пользователя, который в свою очередь только вызывает метод print, недоступный для пользователя:

void AVLtree::printTree() {

print(head, "");

}

# **Интерфейс**

При запуске программы пользователю предоставляется выбор одного из предложенных заданий (Рисунок 1), указанных в постановке задачи. Любой другой введенный символ или число не входящее в диапазон [0;14] спровоцирует повторный ввод.

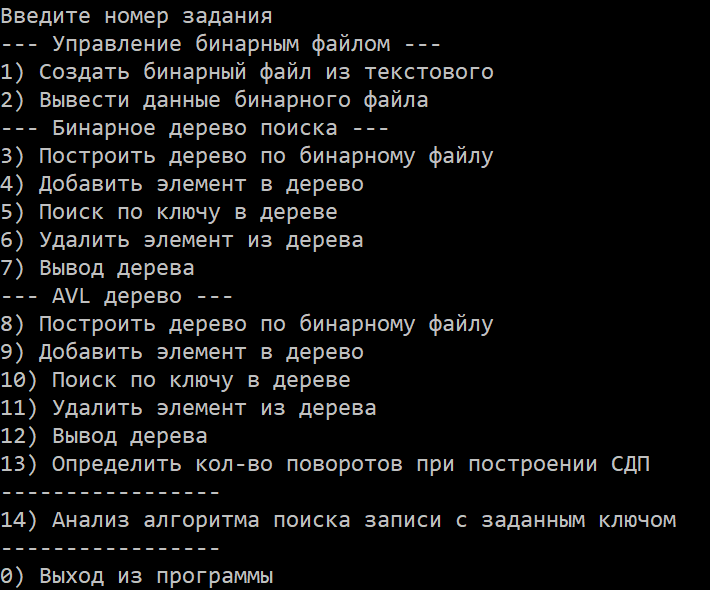


Рисунок 1 - Интерфейс программы

# **Тестирование**

Проведем тестирование всех функций, представленных в меню интерфейса программы.

Пункт 1 отвечает за конвертацию файла из текста в бинарный файл, используя заранее разработанную структуру. Выведем сгенерированный файл, заодно и протестировав пункт 2 (Рисунок 2). Операция выполняется корректно.

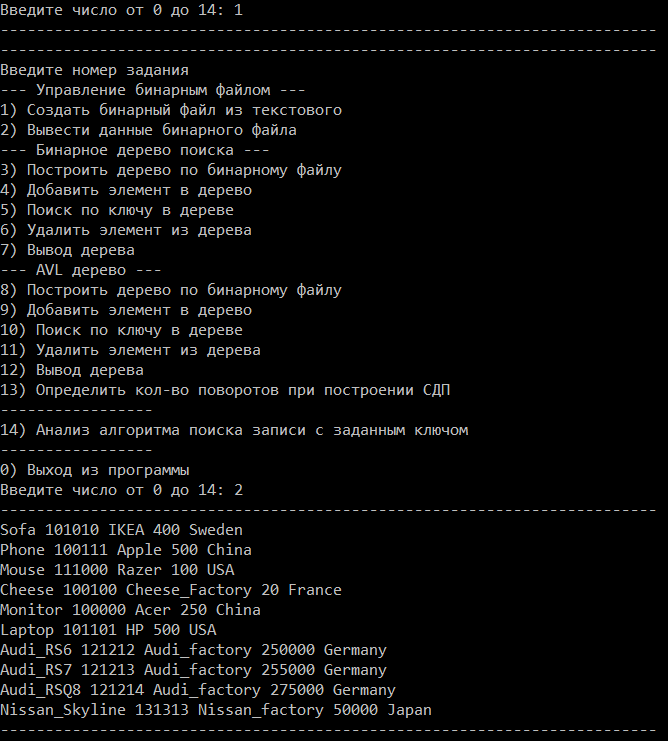


Рисунок 2 - Тестирование 1 и 2 функций

Пункт 3 строит бинарное дерево поиска по бинарному файлу. Выведем созданное дерево, заодно и протестировав пункт 7 (Рисунок 3). Операция выполняется корректно.

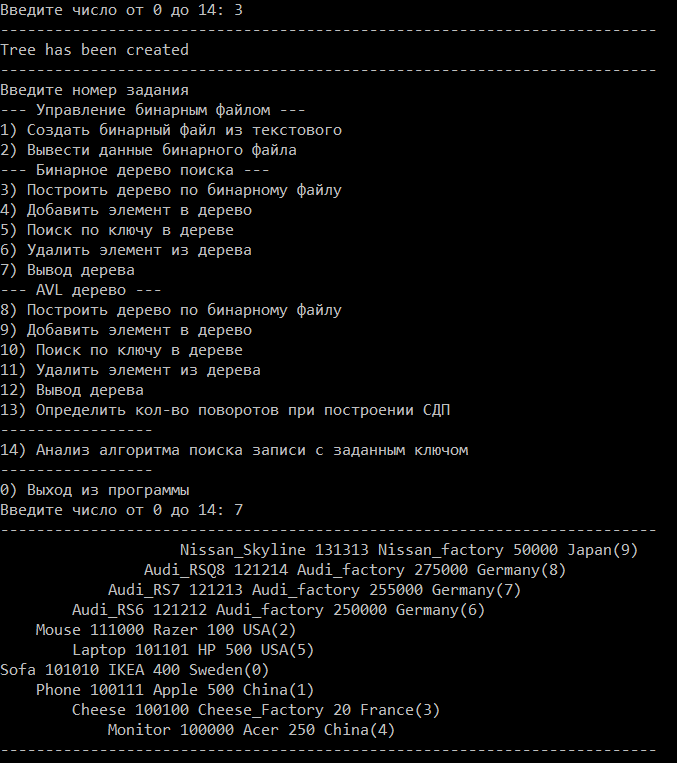


Рисунок 3 - Тестирование 3 и 7 функций

Пункт 4 добавляет новый элемент дерево. Пользователю предлагается ввести данные объекта, а затем созданный объект добавляется в дерево (Рисунок 4). Операция выполняется корректно.

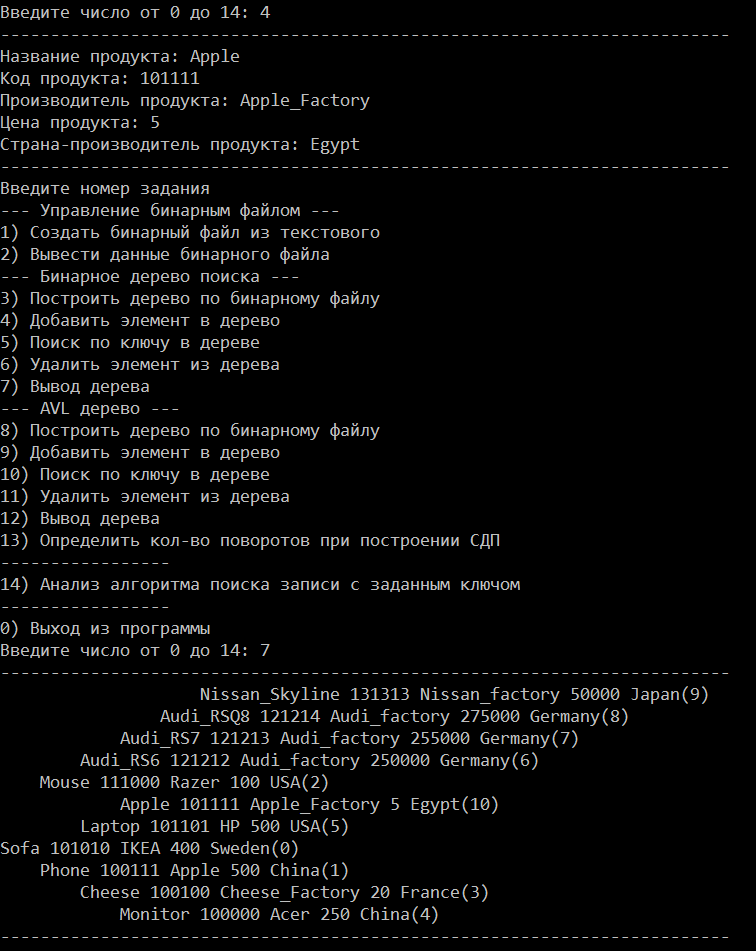


Рисунок 4 - Тестирование 4 функции

Пункт меню под номером 5 выполняет поиск в дереве по ключу, обходя его в глубину. Пользователю предлагается ввести ключ, а затем в дереве ищется элемент по введенному ключу. Операция выполняется корректно (Рисунок 5).

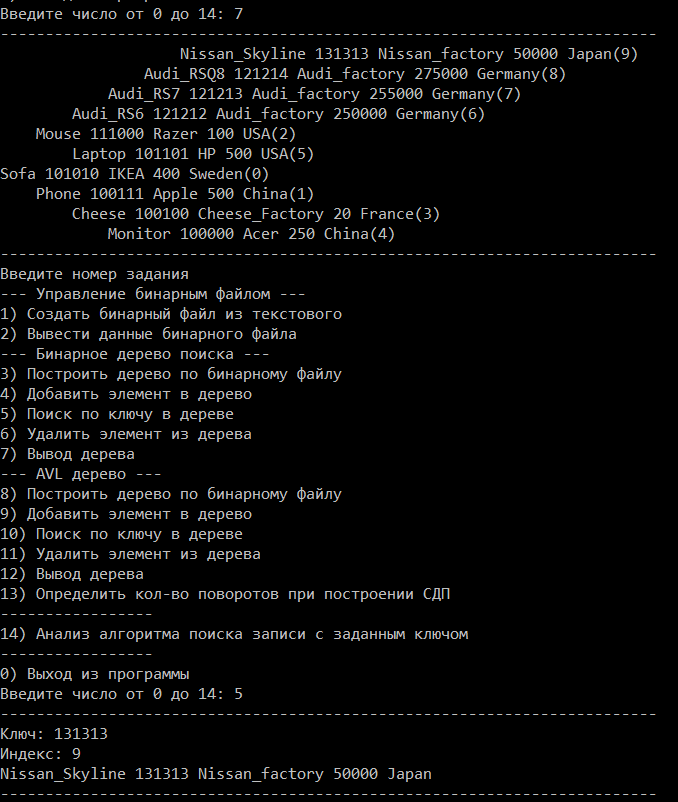


Рисунок 5 - Тестирование 5 функции

Пункт 6 удаляет элемент из дерева по ключу. Пользователю предлагается ввести ключ, а затем элемент по введенному ключу удаляется из дерева. Операция выполняется корректно (Рисунок 6).

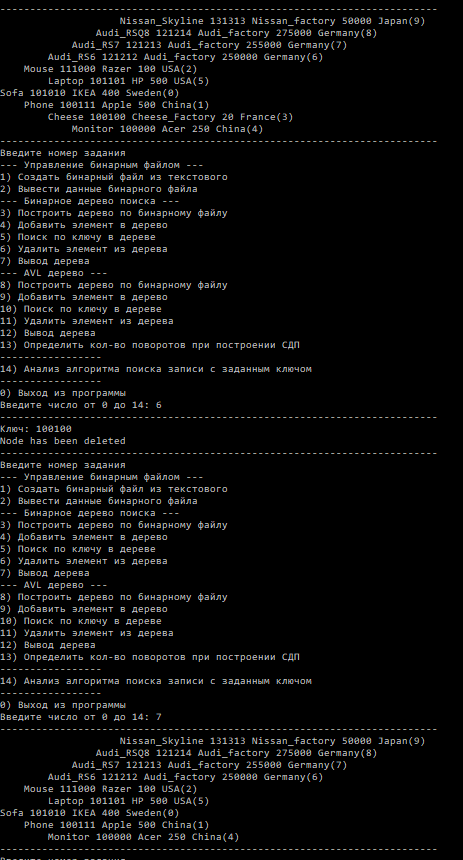


Рисунок 6 - Тестирование 6 функции

Пункт 8 строит АВЛ-дерево по бинарному файлу. Выведем созданное дерево, заодно и протестировав пункт 12 (Рисунок 7). Операция выполняется корректно.

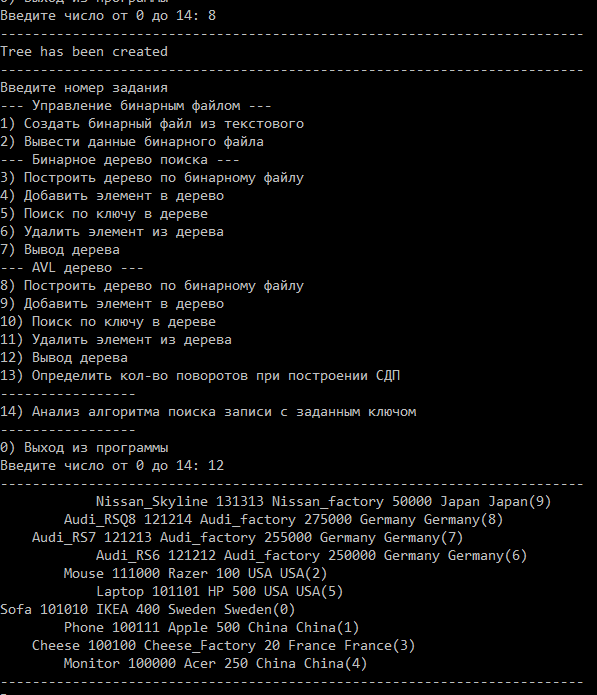


Рисунок 7 - Тестирование 8 и 12 функций

Пункт 9 добавляет новый элемент в дерево. Пользователю предлагается ввести данные объекта, а затем созданный объект добавляется в дерево (Рисунок 8). Операция выполняется корректно.

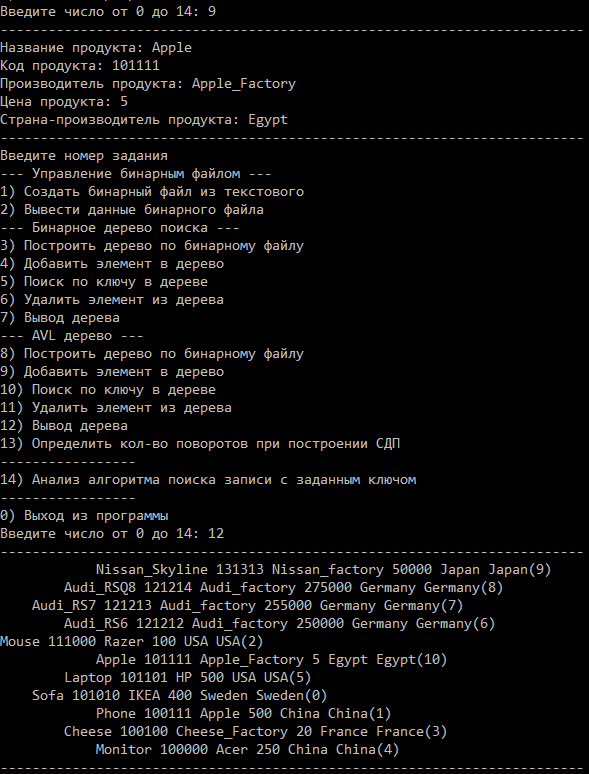


Рисунок 8 - Тестирование 9 функции

Пункт меню под номером 10 выполняет поиск в дереве по ключу, обходя его в глубину. Пользователю предлагается ввести ключ, а затем в дереве ищется элемент по введенному ключу. Операция выполняется корректно (Рисунок 9).

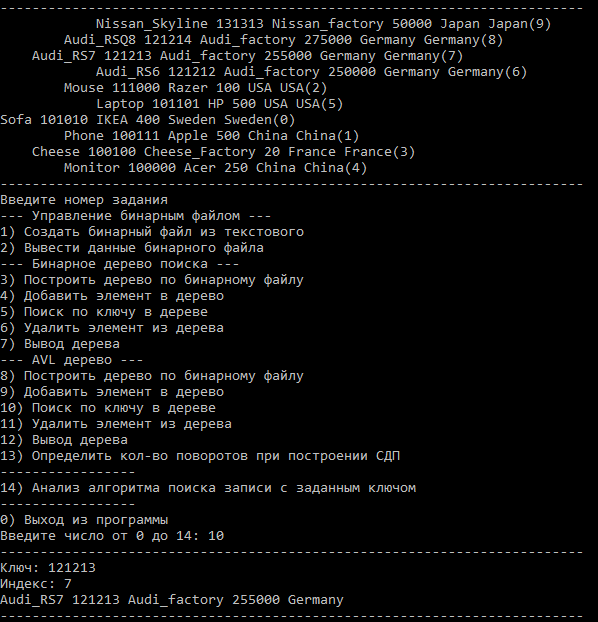


Рисунок 9 - Тестирование 10 функции

Пункт 11 удаляет элемент из дерева по ключу. Пользователю предлагается ввести ключ, а затем элемент по введенному ключу удаляется из дерева. Операция выполняется корректно (Рисунок 10).

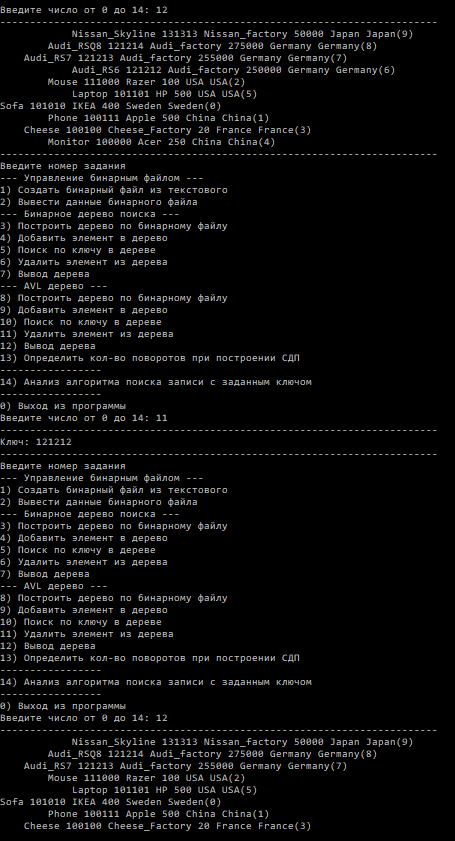


Рисунок 10 - Тестирование 11 функции

Пункт 13 определяет кол-во поворотов при построении СДП. Определим число поворотов при построении СДП из 100 000 элементов (Рисунок 11). В результате можно сделать вывод, что среднее число поворотов приблизительно 0,69 на вставленный элемент.

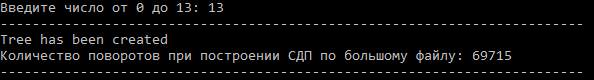


Рисунок 11 - Тестирование 13 функции

# **Задание 3**

Проведем анализ алгоритма поиска записи с заданным ключом при применении структур данных:

* хеш – таблица;
* бинарное дерево поиска;
* СДП

Результаты анализа представлены в таблице ниже

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |
| хеш – таблица | 10 / 100000 | 68 байт | 1 мс / 84 мс |
| бинарное дерево поиска | 10 / 100000 | 68 байт | 1 мс / 324 мс |
| СДП | 10 / 100000 | 68 байт | 1 мс / 151 мс |

Из таблицы видно, что хэш-таблицы работают гораздо быстрее, чем деревья. Поиск в бинарном дереве самый медленный из представленных структур данных из-за проблемы неоптимизированной высоты дерева, которая в AVL дереве технически верно решена и как следствие – более быстрое время поиска.

# **ЗАКЛЮЧЕНИЕ**

В результате выполнения работы были:

* получены навыки в разработки и реализации алгоритмов управления бинарным деревом поиска и сбалансированными бинарными деревьями поиска (АВЛ – деревьями);
* получены навыки в применении файловых потоков прямого доступа к данным файла;
* получены навыки в применении сбалансированного дерева поиска для прямого доступа к записям файла.

# **ПОЛНЫЙ КОД ПРОГРАММЫ**

Product.h

#pragma once

#ifndef PRODUCT\_H

#define PRODUCT\_H

struct Product {

char name[20] = "";

char manufacturer[20] = "";

char country[20] = "";

int price = 0;

int code = 0;

};

static int PRODUCT\_SIZE = sizeof(Product);

void printRecord(Product product);

#endif // !1

Product.cpp

#include "Product.h"

#include <iostream>

using namespace std;

void printRecord(Product product) {

cout << product.name << " " << product.code << " " << product.manufacturer << " " << product.price << " " << product.country << endl;

}

binfile.h

#pragma once

#ifndef BINFILE\_H

#define BINFILE\_H

#include "Product.h"

#include <fstream>

#include <iostream>

using namespace std;

void ConvertTextToBin(string bin\_file\_name, string txt\_file\_name);

void OutputBinFile(string bin\_file\_name);

Product DirectAccess(string bin\_file\_name, int number);

#endif

binfile.cpp

#include "binfile.h"

void ConvertTextToBin(string bin\_file\_name, string txt\_file\_name) {

ifstream fin(txt\_file\_name, ios::in);

if (fin.is\_open()) {

ofstream fout(bin\_file\_name, ios::out | ios::binary);

Product product;

while (!fin.eof()) {

fin >> product.name >> product.code >> product.manufacturer >> product.price >> product.country;

fout.write((char\*)&product, sizeof(Product));

}

fout.close();

}

else {

cout << "File not found" << endl;

}

fin.close();

}

void OutputBinFile(string bin\_file\_name) {

ifstream fin(bin\_file\_name, ios::in | ios::binary);

Product product;

if (fin.is\_open()) {

while (fin.read((char\*)&product, sizeof(Product))) {

cout << product.name << " " << product.code << " " << product.manufacturer << " " << product.price << " " << product.country << "\n";

}

}

else {

cout << "File not found" << endl;

}

fin.close();

}

Product DirectAccess(string bin\_file\_name, int number) {

ifstream binFile(bin\_file\_name, ios::binary | std::ios::in);

if (!binFile.is\_open()) {

cout << "Файла с таким именем не существует" << endl;

}

binFile.seekg(0, ios::end);

long long size = binFile.tellg();

binFile.seekg(0, std::ios::beg);

if ((PRODUCT\_SIZE \* sizeof(char)) \* number > size) {

binFile.close();

return {};

}

binFile.seekg((PRODUCT\_SIZE)\*number);

Product product;

binFile.read((char\*)&product, sizeof(product));

binFile.close();

return product;

}

bintree.h

#pragma once

#include "Product.h"

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

class BinaryTree {

private:

struct Node {

Product value;

int key = -1;

int index = -1;

Node\* left = nullptr;

Node\* right = nullptr;

};

Node\* head;

void print(Node\* node, string space = "");

Node\* findMinNode(Node\* node, Node\* parent, int left);

public:

BinaryTree();

bool createTree(string bin\_file\_name);

void addNode(Product product, int index);

int findNode(int key);

bool removeNode(int key);

void printTree();

};

bintree.cpp

#include "bintree.h"

BinaryTree::BinaryTree() {

head = new Node();

}

bool BinaryTree::createTree(string bin\_file\_name) {

ifstream fin;

fin.open(bin\_file\_name, ios::binary | ios::in);

Product product;

if (fin.is\_open()) {

int index = 0;

while (fin.read((char\*)&product, PRODUCT\_SIZE)) {

addNode(product, index);

index++;

}

fin.close();

return true;

}

return false;

}

void BinaryTree::addNode(Product product, int index) {

Node\* temp = head;

while (true) {

if (temp->value.code==0) {

temp->value = product;

temp->key = product.code;

temp->index = index;

break;

}

else if (temp->value.code < product.code) {

if (temp->right == nullptr) {

temp->right = new Node;

}

temp = temp->right;

}

else if (temp->value.code > product.code) {

if (temp->left == nullptr) {

temp->left = new Node;

}

temp = temp->left;

}

}

}

int BinaryTree::findNode(int key) {

Node\* temp = head;

while (true) {

if (temp == nullptr) {

return -1;

}

else if (temp->value.code == key) {

return temp->index;

}

else if (temp->value.code < key) {

temp = temp->right;

}

else if (temp->value.code > key) {

temp = temp->left;

}

}

return -1;

}

BinaryTree::Node\* BinaryTree::findMinNode(Node\* node, Node\* parent, int left) {

if (node->left == nullptr) {

parent->left = nullptr;

return node;

}

return findMinNode(node->left, node, true);

}

bool BinaryTree::removeNode(int key) {

Node\* temp = head;

Node\* parent = head;

int flag = 0;

if (findNode(key) == -1) {

return false;

}

while (true) {

if (temp->value.code == key) {

if (flag) parent->right = nullptr;

else parent->left = nullptr;

if (temp->left == nullptr && temp->right == nullptr) {

delete temp;

}

else if (temp->left == nullptr) {

parent->right = temp->right;

delete temp;

}

else if (temp->right == nullptr) {

parent->left = temp->left;

delete temp;

}

else {

Node\* res = findMinNode(temp->right, temp, true);

res->left = temp->left;

res->right = temp->right;

delete temp;

if (flag) parent->right = res;

else parent->left = res;

}

return true;

}

else if (temp->value.code < key) {

flag = 1;

parent = temp;

temp = temp->right;

}

else {

flag = 0;

parent = temp;

temp = temp->left;

}

}

return false;

}

void BinaryTree::print(Node\* node, string space) {

if (node->right != nullptr) {

print(node->right, space + " ");

}

cout << space << node->value.name << " " << node->value.code << " " << node->value.manufacturer << " " << node->value.price << " " << node->value.country << "(" << node->index << ")" << endl;

if (node->left != nullptr) {

print(node->left, space + " ");

}

}

void BinaryTree::printTree() {

print(head, "");

}

AVLtree.h

#pragma once

#include "Product.h"

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

class AVLtree {

private:

struct Node {

Product value;

int key = -1;

int index = -1;

Node\* left = nullptr;

Node\* right = nullptr;

int height = -1;

};

int size = -1;

int height(Node\* node);

int height\_diff(Node\* node);

void fixheight(Node\* node);

Node\* rotateright(Node\* p);

Node\* rotateleft(Node\* q);

Node\* balance(Node\* node);

Node\* findMinNode(Node\* node);

Node\* removeMinNode(Node\* node);

void print(Node\* node, string space);

public:

Node\* head = nullptr;

int rotations = 0;

bool createAVLTree(string bin\_file\_name);

Node\* addNode(Node\* p, Product pr, int index);

Node\* removeNode(Node\* p, int k);

int findNode(int k);

void printTree();

};

AVLtree.cpp

#include "AVLtree.h"

bool AVLtree::createAVLTree(string bin\_file\_name) {

ifstream fin;

fin.open(bin\_file\_name, ios::binary | ios::in);

Product product;

int index = 0;

if (fin.is\_open()) {

int index = 0;

while (fin.read((char\*)&product, PRODUCT\_SIZE)) {

head = addNode(head, product, index);

index++;

}

fin.close();

return true;

}

return false;

}

int AVLtree::height(Node\* node) {

if (node) {

return node->height;

}

else {

return 0;

}

}

int AVLtree::height\_diff(Node\* node) {

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

}

void AVLtree::fixheight(Node\* node) {

int hl = height(node->left);

int hr = height(node->right);

if (hl > hr) {

node->height = hl + 1;

}

else {

node->height = hr + 1;

}

}

AVLtree::Node\* AVLtree::rotateright(Node\* p) {

Node\* q = p->left;

p->left = q->right;

q->right = p;

fixheight(p);

fixheight(q);

rotations++;

return q;

}

AVLtree::Node\* AVLtree::rotateleft(Node\* q) {

Node\* p = q->right;

q->right = p->left;

p->left = q;

fixheight(q);

fixheight(p);

rotations++;

return p;

}

AVLtree::Node\* AVLtree::balance(Node\* node) {

fixheight(node);

if (height\_diff(node) == 2) {

if (height\_diff(node->right) < 0) {

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

}

return rotateleft(node);

}

if (height\_diff(node) == -2) {

if (height\_diff(node->left) > 0) {

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

}

return rotateright(node);

}

return node;

}

AVLtree::Node\* AVLtree::addNode(Node\* node, Product product, int index) {

if (!node) {

node = new Node;

node->value = product;

node->key = product.code;

node->index = index;

node->left = node->right = nullptr;

node->height = 1;

size++;

return node;

}

if (product.code < node->key) {

node->left = addNode(node->left, product, index);

}

else {

node->right = addNode(node->right, product, index);

}

return balance(node);

}

AVLtree::Node\* AVLtree::findMinNode(Node\* node) {

if (node->left) {

return findMinNode(node->left);

}

else {

return node;

}

}

AVLtree::Node\* AVLtree::removeMinNode(Node\* node) {

if (node->left == 0) {

return node->right;

}

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

return balance(node);

}

AVLtree::Node\* AVLtree::removeNode(Node\* node, int k) {

if (!node) {

return nullptr;

}

if (k < node->key) {

node->left = removeNode(node->left, k);

}

else if (k > node->key) {

node->right = removeNode(node->right, k);

}

else {

Node\* q = node->left;

Node\* r = node->right;

delete node;

if (!r) {

return q;

}

Node\* min = findMinNode(r);

min->right = removeMinNode(r);

min->left = q;

return balance(min);

}

return balance(node);

}

int AVLtree::findNode(int k) {

Node\* node = head;

while (true) {

if (node == nullptr) {

return -1;

}

if (k < node->key) {

node = node->left;

}

else if (k > node->key) {

node = node->right;

}

else {

return node->index;

}

}

return -1;

}

void AVLtree::print(Node\* node, string space) {

if (node->right != nullptr) {

print(node->right, space + " ");

}

cout << space << node->value.name << " " << node->value.code << " " << node->value.manufacturer << " " << node->value.price << " " << node->value.country << " " << node->value.country << "(" << node->index << ")" << endl;

//cout << space << node->key << " " << node->index << endl;

if (node->left != nullptr) {

print(node->left, space + " ");

}

}

void AVLtree::printTree() {

print(head, "");

}

siaod5 2part.cpp

#include <iostream>

#include <chrono>

#include "Product.h"

#include "binfile.h"

#include "bintree.h"

#include "AVLtree.h"

using namespace std;

int inputMenu(int& number);

int rotations();

int main() {

system("chcp 1251 > null");

int menu;

BinaryTree bintree;

AVLtree avltree;

while (true) {

inputMenu(menu);

if (menu == 0) {

cout << "Задание не выбрано... Выход из программы" << endl;

break;

}

switch (menu) {

case 1: {

cout << "-------------------------------------------------------------------------" << endl;

ConvertTextToBin("file.bin", "file.txt");

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 2: {

cout << "-------------------------------------------------------------------------" << endl;

OutputBinFile("file.bin");

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 3: {

cout << "-------------------------------------------------------------------------" << endl;

if (bintree.createTree("file.bin")) {

cout << "Tree has been created" << endl;

}

else {

cout << "Error..." << endl;

}

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 4: {

cout << "-------------------------------------------------------------------------" << endl;

Product pr;

cout << "Название продукта: "; cin >> pr.name; cin.ignore(); cin.clear();

cout << "Код продукта: "; cin >> pr.code; cin.ignore(); cin.clear();

cout << "Производитель продукта: "; cin >> pr.manufacturer; cin.ignore(); cin.clear();

cout << "Цена продукта: "; cin >> pr.price; cin.ignore(); cin.clear();

cout << "Страна-производитель продукта: "; cin >> pr.country; cin.ignore(); cin.clear();

bintree.addNode(pr, 10);

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 5: {

cout << "-------------------------------------------------------------------------" << endl;

int key, index=-1;

cout << "Key: "; cin >> key; cin.ignore(); cin.clear();

index = bintree.findNode(key);

cout << "Index: " << index << endl;

printRecord(DirectAccess("file.bin", index));

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 6: {

cout << "-------------------------------------------------------------------------" << endl;

int key;

cout << "Key: "; cin >> key; cin.ignore(); cin.clear();

if (bintree.removeNode(key)) {

cout << "Node has been deleted" << endl;

}

else {

cout << "Error..." << endl;

}

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 7: {

cout << "-------------------------------------------------------------------------" << endl;

bintree.printTree();

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 8: {

cout << "-------------------------------------------------------------------------" << endl;

if (avltree.createAVLTree("file.bin")) {

cout << "Tree has been created" << endl;

}

else {

cout << "Error..." << endl;

}

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 9: {

cout << "-------------------------------------------------------------------------" << endl;

Product pr;

cout << "Название продукта: "; cin >> pr.name; cin.ignore(); cin.clear();

cout << "Код продукта: "; cin >> pr.code; cin.ignore(); cin.clear();

cout << "Производитель продукта: "; cin >> pr.manufacturer; cin.ignore(); cin.clear();

cout << "Цена продукта: "; cin >> pr.price; cin.ignore(); cin.clear();

cout << "Страна-производитель продукта: "; cin >> pr.country; cin.ignore(); cin.clear();

avltree.head = avltree.addNode(avltree.head, pr, 10);

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 10: {

cout << "-------------------------------------------------------------------------" << endl;

int key, index=-1;

cout << "Key: "; cin >> key; cin.ignore(); cin.clear();

index = avltree.findNode(key);

cout << "Index: " << index << endl;

printRecord(DirectAccess("file.bin", index));

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 11: {

cout << "-------------------------------------------------------------------------" << endl;

int key;

cout << "Ключ: "; cin >> key; cin.ignore(); cin.clear();

avltree.head = avltree.removeNode(avltree.head, key);

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 12: {

cout << "-------------------------------------------------------------------------" << endl;

avltree.printTree();

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 13: {

cout << "-------------------------------------------------------------------------" << endl;

cout << "Количество поворотов при построении СДП по большому файлу: "<< rotations() << endl;

cout << "-------------------------------------------------------------------------" << endl;

break;

}

default: {

break;

}

}

}

}

int inputMenu(int& number) {

cout << "Введите номер задания" << endl;

cout << "--- Управление бинарным файлом ---" << endl;

cout << "1) Создать бинарный файл из текстового" << endl;

cout << "2) Вывести данные бинарного файла" << endl;

cout << "--- Бинарное дерево поиска ---" << endl;

cout << "3) Построить дерево по бинарному файлу" << endl;

cout << "4) Добавить элемент в дерево" << endl;

cout << "5) Поиск по ключу в дереве" << endl;

cout << "6) Удалить элемент из дерева" << endl;

cout << "7) Вывод дерева" << endl;

cout << "--- AVL дерево ---" << endl;

cout << "8) Построить дерево по бинарному файлу" << endl;

cout << "9) Добавить элемент в дерево" << endl;

cout << "10) Поиск по ключу в дереве" << endl;

cout << "11) Удалить элемент из дерева" << endl;

cout << "12) Вывод дерева" << endl;

cout << "13) Определить кол-во поворотов при построении СДП" << endl;

cout << "-----------------" << endl;

cout << "0) Выход из программы" << endl;

while (1) {

cout << "Введите число от 0 до 13: ";

if ((cin >> number).good() && (number >= 0) && (number <= 13)) {

return number;

}

if (cin.fail()) {

cin.clear();

cout << endl << "Неверный ввод, повторите." << endl << endl;

}

else {

cout << endl << "Число вне допустимого диапазона значений. Повторите ввод." << endl << endl;

}

cin.ignore(100, '\n');

}

}

int rotations() {

AVLtree tree;

if (tree.createAVLTree("bigfile.bin")) {

cout << "Tree has been created" << endl;

}

else {

cout << "Error..." << endl;

}

return tree.rotations;

}

//Apple

//101111

//Apple\_Factory

//5

//Egypt