|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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-33-22 | Шило Ю.С. |
| Принял преподаватель | Филатов А.С. |

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

Москва 2023

# **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.

1. **Постановка задачи**
2. Разработать класс управления файлом. Включить методы: создание двоичного файла записей фиксированной длины из заранее подготовленных данных в текстовом файле; поиск и вывод записи в файле с помощью линейного поиска; вывод записи по известному адресу в файле, добавление записи в файл. Структура записи файла определена индивидуальным вариантом задания.
3. Разработать класс «Бинарное дерево поиска». Структура информационной части узла дерева включает ключ и ссылку на запись в файле. Разработать основные методы: построение дерева по существующему файлу; добавление элемента в дерево; поиск по ключу в дереве; удаление элемента из дерева; вывод дерева в форме дерева.
4. Разработать класс «Сбалансированное дерево поиска». Структура информационной части узла дерева включает ключ и ссылку на запись в файле. Разработать основные методы: построение дерева по существующему файлу; добавление элемента в дерево; поиск по ключу в дереве; удаление элемента из дерева; вывод дерева в форме дерева.
5. Разработать приложение, демонстрирующее выполнение всех методов и поиск записей в файле с помощью БДП, СДП и линейного поиска.
6. Провести анализ времени выполнения поиска записей, находящихся в начале, середине и конце большого файла тремя способами.
7. Определить среднее число выполненных поворотов (число поворотов на общее число вставленных ключей) при включении ключей в СДП при его формировании из двоичного файла.
8. Выполнить тестирование.
9. Составить отчет, отобразив в нем описание выполнения всех этапов разработки, тестирования и код всей программы со скриншотами результатов тестирования.

|  |  |  |
| --- | --- | --- |
| № | Тип балансированного  дерева поиска (СДП) | Структура записи (ключ – подчеркнутое поле) |
| 4 | Рандомизированное | Владельцев автомобилей. номер машины, марка, сведения о владельце. |

# **Решение**

Сбалансированные деревья поиска (СДП) - это структуры данных, которые обеспечивают эффективное хранение и поиск элементов. Они характеризуются тем, что высота поддеревьев в каждом узле различается не более чем на один уровень. Это позволяет достичь быстрого времени выполнения операций поиска, вставки и удаления элементов в дереве.

СДП обеспечивают логарифмическую сложность операций поиска, вставки и удаления, что делает их очень эффективными для работы с большими объемами данных.

Сбалансированные деревья поиска находят широкое применение в различных областях, включая базы данных, поисковые системы, компиляторы и другие. Их гибкость и эффективность делают их незаменимыми инструментами при работе с динамическими наборами данных, где требуется эффективное выполнение операций поиска и модификации.

**Задание**

1. Структура CarRecord которая будет находится в файле. Данная структура содержит три поля одно поле целочисленное, а два остальных символьный массив.

|  |
| --- |
| struct CarRecord{  int key;  char brand[64];  char owner[64]; }; |

1. Для создания бинарного файла был написан метод createFile. Данный метод преобразует обычный текстовый файл в бинарный.

|  |
| --- |
| void createFile(const string& textFileName, const string& binaryFileName) {  ifstream textFile(textFileName);  file.open(binaryFileName, ios::out | ios::binary);  if (!textFile || !file) {  cout << "Failed to open files." << endl;  return;  }  CarRecord record;  while (textFile >> record.key >> record.brand >> record.owner) {  file.write(reinterpret\_cast<const char\*>(&record), sizeof(CarRecord));  }  textFile.close();  file.close(); } |

1. Для линейного поиска объекта в бинарном файле был написан метод linearSearch. Данный метод ищет объект с нужным ключом и выводит информацию о данном ключе.

|  |
| --- |
| void linearSearch(const string& binaryFileName, int carNumber) {  file.open(binaryFileName, ios::in | ios::binary);  if (!file) {  std::cout << "Failed to open file." << std::endl;  return;  }  CarRecord record;  bool found = false;  while (file.read(reinterpret\_cast<char\*>(&record), sizeof(CarRecord))) {  if (record.key == carNumber) {  cout << "Record found." << endl;  cout << "KEY | Brand\t| Owner Info" << endl;  cout << record.key << " | " << record.brand << "\t| " << record.owner << endl;  found = true;  break;  }  }  if (!found) {  cout << "Record not found." << endl;  }  file.close(); } |

1. Для вывода записей из файла был написан метод printRecords. Данный метод выводит все объекты, которые содержаться в бинарном файле.

|  |
| --- |
| void printRecords(const string& binaryFileName){  file.open(binaryFileName, ios::in | ios::binary);  CarRecord record;  cout << "KEY | Brand\t| Owner Info" << endl;  while (file.read(reinterpret\_cast<char\*>(&record), sizeof(CarRecord))) {  cout << record.key << " | " << record.brand << "\t| " << record.owner << endl;  }  file.close(); } |

1. Что бы добавить объект в конец бинарного файла был написан метод addRecord.

|  |
| --- |
| void addRecord(const string& binaryFileName, CarRecord newRecord) {  file.open(binaryFileName, ios::out | ios::app | ios::binary);  if (!file) {  cout << "Failed to open file." << endl;  return;  }  file.write(reinterpret\_cast<const char\*>(&newRecord), sizeof(CarRecord));   file.close(); } |

1. Для поиска элемента внутри бинарного дерева поиска был написан метод searchRecursive.

|  |
| --- |
| NodeBin\* searchRecursive(NodeBin\* node, int key) {  if (node == nullptr || node->key == key){  return node;  }   if (key < node->key) {  return searchRecursive(node->left, key);  }  else {  return searchRecursive(node->right, key);  } } |

1. Для удаления элемента внутри бинарного дерева поиска был написан метод deleteRecursive.

|  |
| --- |
| NodeBin\* deleteRecursive(NodeBin\* node, int key) {  if (node == nullptr) {  return node;  }   if (key < node->key) {  node->left = deleteRecursive(node->left, key);  } else if (key > node->key) {  node->right = deleteRecursive(node->right, key);  } else {  if (node->left == nullptr) {  NodeBin\* temp = node->right;  delete node;  return temp;  } else if (node->right == nullptr) {  NodeBin\* temp = node->left;  delete node;  return temp;  }   NodeBin\* minValueNode = getMinValueNode(node->right);  node->key = minValueNode->key;  node->ptr = minValueNode->ptr;  node->right = deleteRecursive(node->right, minValueNode->key);  }   return node; } |

1. Для вывода бинарного дерева поиска был написан метод printRecursive.

|  |
| --- |
| void printTreeRecursive(NodeBin\* node, int level) {  if (node != nullptr) {  printTreeRecursive(node->right, level + 1);  for (int i = 0; i < level; i++) {  std::cout << " ";  }  cout << level;  std::cout << "-> " << node->key << "\\" << node->ptr->brand << std::endl;  printTreeRecursive(node->left, level + 1);  }  } |

1. Для поворота рандомизированного древа было написано два метода rotateLeft и rotateRight.

|  |
| --- |
| Node\* rotateRight(Node\* node) {  Node\* newRoot = node->left;  node->left = newRoot->right;  newRoot->right = node;  return newRoot;  }  Node\* rotateLeft(Node\* node) {  Node\* newRoot = node->right;  node->right = newRoot->left;  newRoot->left = node;  return newRoot;  } |

1. Для добавления элемента в рандомизированном древе поиска был написан метод insertNode.

|  |
| --- |
| Node\* insertNode(Node\* node, int key, CarRecord\* ptr) {  if (node == nullptr)  return new Node(key, ptr);   if (key < node->key) {  node->left = insertNode(node->left, key, ptr);  if (node->left->priority > node->priority)  node = rotateRight(node);  } else {  node->right = insertNode(node->right, key, ptr);  if (node->right->priority > node->priority)  node = rotateLeft(node);  }   return node; } |

1. Для удаления элемента в рандомизированном древе поиска был написан метод deleteNode.

|  |
| --- |
| Node\* deleteNode(Node\* node, int key) {  if (node == nullptr)  return nullptr;   if (key < node->key)  node->left = deleteNode(node->left, key);  else if (key > node->key)  node->right = deleteNode(node->right, key);  else {  if (node->left == nullptr) {  Node\* temp = node->right;  delete node;  node = temp;  } else if (node->right == nullptr) {  Node\* temp = node->left;  delete node;  node = temp;  } else {  if (node->left->priority > node->right->priority) {  node = rotateRight(node);  node->right = deleteNode(node->right, key);  } else {  node = rotateLeft(node);  node->left = deleteNode(node->left, key);  }  }  }   return node; } |

1. Для поиска элемента внутри рандомизированного древа написан метод searchRecursive.

|  |
| --- |
| Node\* searchRecursive(Node\* node, int key) {  if (node == nullptr || node->key == key){  return node;  }   if (key < node->key) {  return searchRecursive(node->left, key);  }  else {  return searchRecursive(node->right, key);  } } |

1. Для вывода рандомизированного древа был написан метод printTreeRecursive.

|  |
| --- |
| void printTreeRecursive(Node\* node, int level) {  if (node != nullptr) {  printTreeRecursive(node->right, level + 1);  for (int i = 0; i < level; i++) {  std::cout << " ";  }  cout << level;  std::cout << "-> " << node->key << "\\" << node->ptr->key << "\\" << node->ptr->brand << std::endl;  printTreeRecursive(node->left, level + 1);  } |

1. Для вывода рандомизированного древа был написан метод printTreeRecursive.

|  |
| --- |
| void printTreeRecursive(Node\* node, int level) {  if (node != nullptr) {  printTreeRecursive(node->right, level + 1);  for (int i = 0; i < level; i++) {  std::cout << " ";  }  cout << level;  std::cout << "-> " << node->key << "\\" << node->ptr->key << "\\" << node->ptr->brand << std::endl;  printTreeRecursive(node->left, level + 1);  } |

1. Для проверки бинарного древа поиска была написана функция testFM.

|  |
| --- |
| void testBST(){  string binaryFileName = "binary-test.txt";   BinarySearchTree bst;  bst.Load(binaryFileName);   cout << "===[ BinarySearchTree ]===" << endl;  bst.printTree();  cout << endl;   cout << "Delete elem with key 538398" << endl;  bst.remove(538398);   cout << "===[ BinarySearchTree ]===" << endl;  bst.printTree();  cout << endl;   cout << "Add elem with key 400000" << endl;  CarRecord\* temp = new CarRecord (400000, "BRAND-ADDED", "OWNER-ADDED");  bst.insert\_file(400000, temp, binaryFileName);   cout << "===[ BinarySearchTree ]===" << endl;  bst.printTree();  cout << endl;   cout << "Search elem with key 910542" << endl;  bst.search(910542); } |

1. Для проверки рандомизированного древа поиска была написана функция testRBST.

|  |
| --- |
| void testRBST(){  string binaryFileName = "binary-test.txt";   RandomizedBST rbst;  rbst.Load(binaryFileName);   cout << "===[ RandomizedBST ]===" << endl;  rbst.printTree();  cout << endl;   cout << "Delete elem with key 538398" << endl;  rbst.remove(538398);   cout << "===[ RandomizedBST ]===" << endl;  rbst.printTree();  cout << endl;   cout << "Add elem with key 910000" << endl;  CarRecord\* temp = new CarRecord (910000, "BRAND-ADDED", "OWNER-ADDED");  rbst.insert(910000, temp);    cout << "===[ RandomizedBST ]===" << endl;  rbst.printTree();  cout << endl;   cout << "Search elem with key 910542" << endl;  rbst.search(910542); } |

1. **Тестирование**

Для тестирования кода создадим текстовый файл и введем следующие значения. Данный текстовый файл предоставлен на рисунке 1.

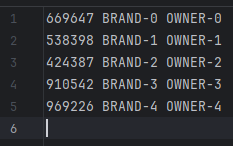


Рисунок 1. Содержание тестового файла

Вызовем функцию, которая проверит работоспособность класса для работы с бинарным файлом. Отработка данной функции предоставлена на рисунке 2.

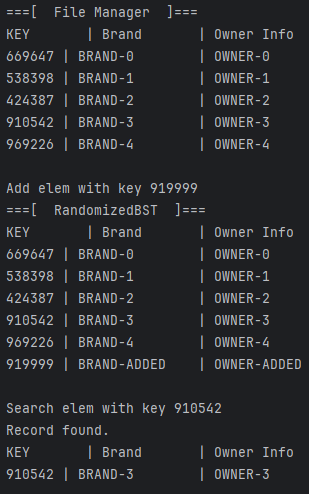


Рисунок 2. Результат, полученный после вызова функции для работы с текстом

Затем вызовем функцию, которая отвечает за бинарное дерево поиска и поможет нам проверить его работоспособность. Отработка данной функции предоставлена на рисунке 3.

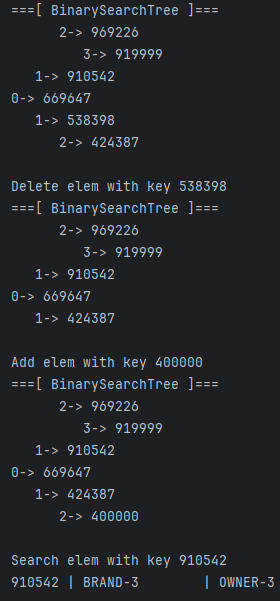


Рисунок 3. Результат отработки функции для работы с бинарным деревом поиска

И наконец вызовем функцию, которая отвечает за рандомизированное дерево поиска и поможет нам проверить его работоспособность. Отработка данной функции предоставлена на рисунке 4.

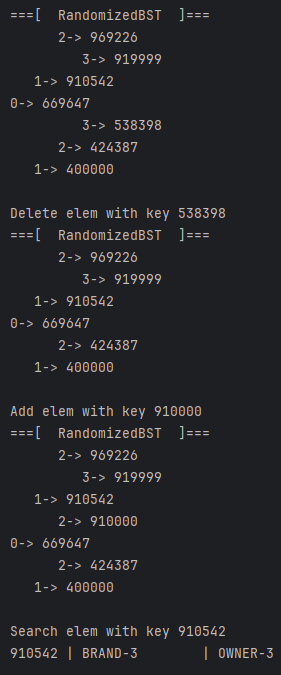


Рисунок 4. Результат отработки функции для работы с рандомизированным деревом поиска

# **Вывод**

В процессе выполнения данной практической работы я успешно овладел навыками работы с файловыми потоками в языке программирования C++ для управления текстовыми и двоичными файлами, также были получены навыки по работе с бинарным древом поиска и рандомизированным древом поиска.

# **Исходный код программы**

Исходный код файла - main.cpp

|  |
| --- |
| #include "FileManager.h"  #include "BinSearchTree.h"  #include "RandomizedSearchTree.h"  using namespace std;  void testFM(){  string binaryFileName = "binary-test.txt", textFileName = "text-test.txt";  FileManager fileManager;  fileManager.createFile(textFileName, binaryFileName);  cout << "===[ File Manager ]===" << endl;  fileManager.printRecords(binaryFileName);  cout << endl;  cout << "Add elem with key 919999" << endl;  CarRecord temp = CarRecord (919999, "BRAND-ADDED", "OWNER-ADDED");  fileManager.addRecord(binaryFileName, temp);  cout << "===[ RandomizedBST ]===" << endl;  fileManager.printRecords(binaryFileName);  cout << endl;  cout << "Search elem with key 910542" << endl;  fileManager.linearSearch(binaryFileName, 910542);  }  void testBST(){  string binaryFileName = "binary-test.txt";  BinarySearchTree bst;  bst.Load(binaryFileName);  cout << "===[ BinarySearchTree ]===" << endl;  bst.printTree();  cout << endl;  cout << "Delete elem with key 538398" << endl;  bst.remove(538398);  cout << "===[ BinarySearchTree ]===" << endl;  bst.printTree();  cout << endl;  cout << "Add elem with key 400000" << endl;  CarRecord\* temp = new CarRecord (400000, "BRAND-ADDED", "OWNER-ADDED");  bst.insert\_file(400000, temp, binaryFileName);  cout << "===[ BinarySearchTree ]===" << endl;  bst.printTree();  cout << endl;  cout << "Search elem with key 910542" << endl;  bst.search(910542);  }  void testRBST(){  string binaryFileName = "binary-test.txt";  RandomizedBST rbst;  rbst.Load(binaryFileName);  cout << "===[ RandomizedBST ]===" << endl;  rbst.printTree();  cout << endl;  cout << "Delete elem with key 538398" << endl;  rbst.remove(538398);  cout << "===[ RandomizedBST ]===" << endl;  rbst.printTree();  cout << endl;  cout << "Add elem with key 910000" << endl;  CarRecord\* temp = new CarRecord (910000, "BRAND-ADDED", "OWNER-ADDED");  rbst.insert(910000, temp);  cout << "===[ RandomizedBST ]===" << endl;  rbst.printTree();  cout << endl;  cout << "Search elem with key 910542" << endl;  rbst.search(910542);  }  int main(){  testFM();  cout << "\n\n\n";  testBST();  cout << "\n\n\n";  testRBST();  return 0;  } |

Исходный код файла - FileManager.h

|  |
| --- |
| #ifndef PRACTICE\_5\_\_FILEMANAGER\_H  #define PRACTICE\_5\_\_FILEMANAGER\_H  #include "random"  #include "iostream"  #include "fstream"  using namespace std;  int Random\_Func(int min, int max){  random\_device rd;  mt19937 gen(rd());  uniform\_int\_distribution<> RANDOM(min, max);  return RANDOM(gen);  }  void createText(const string& textFileName, int size){  ofstream out;  out.open(textFileName);  for (int i = 0; i < size; i++) {  out << Random\_Func(100000, 1000000) << " BRAND-" << i << " OWNER-" << i << endl;  }  out.close();  }  struct CarRecord{  int key;  char brand[64];  char owner[64];  };  class FileManager {  private:  fstream file;  public:  void createFile(const string& textFileName, const string& binaryFileName) {  ifstream textFile(textFileName);  file.open(binaryFileName, ios::out | ios::binary);  if (!textFile || !file) {  cout << "Failed to open files." << endl;  return;  }  CarRecord record;  while (textFile >> record.key >> record.brand >> record.owner) {  file.write(reinterpret\_cast<const char\*>(&record), sizeof(CarRecord));  }  textFile.close();  file.close();  }  void linearSearch(const string& binaryFileName, int carNumber) {  file.open(binaryFileName, ios::in | ios::binary);  if (!file) {  std::cout << "Failed to open file." << std::endl;  return;  }  CarRecord record;  bool found = false;  while (file.read(reinterpret\_cast<char\*>(&record), sizeof(CarRecord))) {  if (record.key == carNumber) {  cout << "Record found." << endl;  cout << "KEY | Brand\t| Owner Info" << endl;  cout << record.key << " | " << record.brand << "\t| " << record.owner << endl;  found = true;  break;  }  }  if (!found) {  cout << "Record not found." << endl;  }  file.close();  }  void printRecord(const string& binaryFileName, int address) {  file.open(binaryFileName, std::ios::in | std::ios::binary);  if (!file) {  cout << "Failed to open file." << endl;  return;  }  CarRecord record;  file.seekg(address \* sizeof(CarRecord));  file.read(reinterpret\_cast<char\*>(&record), sizeof(CarRecord));  cout << "ADDRESS\t| KEY | Brand\t| Owner Info" << endl;  cout << address << "\t| " << record.key << " | " << record.brand << "\t| " << record.owner << endl;  file.close();  }  void printRecords(const string& binaryFileName){  file.open(binaryFileName, ios::in | ios::binary);  CarRecord record;  cout << "KEY | Brand\t| Owner Info" << endl;  while (file.read(reinterpret\_cast<char\*>(&record), sizeof(CarRecord))) {  cout << record.key << " | " << record.brand << "\t| " << record.owner << endl;  }  file.close();  }  void addRecord(const string& binaryFileName, CarRecord newRecord) {  file.open(binaryFileName, ios::out | ios::app | ios::binary);  if (!file) {  cout << "Failed to open file." << endl;  return;  }  file.write(reinterpret\_cast<const char\*>(&newRecord), sizeof(CarRecord));  file.close();  }  };  #endif |

Исходный код файла - BinSearchTree.h

|  |
| --- |
| #ifndef PRACTICE\_5\_\_BINSEARCHTREE\_H  #define PRACTICE\_5\_\_BINSEARCHTREE\_H  #include "FileManager.h"  struct NodeBin {  int key;  CarRecord\* ptr;  NodeBin\* left;  NodeBin\* right;  NodeBin(int key, CarRecord\* ptr) {  this->key = key;  this->ptr = ptr;  left = nullptr;  right = nullptr;  }  };  class BinarySearchTree {  private:  NodeBin\* root;  NodeBin\* insertRecursive(NodeBin\* node, int key, CarRecord\* ptr) {  if (node == nullptr) {  return new NodeBin(key, ptr);  }  if (key < node->key) {  node->left = insertRecursive(node->left, key, ptr);  } else if (key > node->key) {  node->right = insertRecursive(node->right, key, ptr);  }  return node;  }  NodeBin\* searchRecursive(NodeBin\* node, int key) {  if (node == nullptr || node->key == key){  return node;  }  if (key < node->key) {  return searchRecursive(node->left, key);  }  else {  return searchRecursive(node->right, key);  }  }  NodeBin\* deleteRecursive(NodeBin\* node, int key) {  if (node == nullptr) {  return node;  }  if (key < node->key) {  node->left = deleteRecursive(node->left, key);  } else if (key > node->key) {  node->right = deleteRecursive(node->right, key);  } else {  if (node->left == nullptr) {  NodeBin\* temp = node->right;  delete node;  return temp;  } else if (node->right == nullptr) {  NodeBin\* temp = node->left;  delete node;  return temp;  }  NodeBin\* minValueNode = getMinValueNode(node->right);  node->key = minValueNode->key;  node->ptr = minValueNode->ptr;  node->right = deleteRecursive(node->right, minValueNode->key);  }  return node;  }  NodeBin\* getMinValueNode(NodeBin\* node) {  NodeBin\* current = node;  while (current && current->left != nullptr) {  current = current->left;  }  return current;  }  void printTreeRecursive(NodeBin\* node, int level) {  if (node != nullptr) {  printTreeRecursive(node->right, level + 1);  for (int i = 0; i < level; i++) {  std::cout << " ";  }  cout << level;  std::cout << "-> " << node->key << "\\" << node->ptr->brand << std::endl;  printTreeRecursive(node->left, level + 1);  }  }  public:  BinarySearchTree() {  root = nullptr;  }  void insert(int key, CarRecord\* ptr) {  root = insertRecursive(root, key, ptr);  }  void insert\_file(int key, CarRecord\* ptr, string binFileName){  FileManager fileManager;  fileManager.addRecord(binFileName, \*ptr);  root = insertRecursive(root, key, ptr);  }  void search(int key) {  NodeBin\* Found = searchRecursive(root, key);  if (Found != nullptr)  cout << Found->ptr->key << " | " << Found->ptr->brand << "\t| " << Found->ptr->owner << endl;  else {  cout << "The element with this key does not exist." << endl;  }  }  void remove(int key) {  root = deleteRecursive(root, key);  }  void printTree() {  printTreeRecursive(root, 0);  }  void Load(const string& binaryFileName){  fstream file;  file.open(binaryFileName, ios::in | ios::binary);  CarRecord record;  while (file.read(reinterpret\_cast<char\*>(&record), sizeof(CarRecord))) {  CarRecord \*temp = new CarRecord(record);  insert(record.key, temp);  }  file.close();  }  };  #endif |

Исходный код файла - RandomizedSearchTree.h

|  |
| --- |
| #ifndef PRACTICE\_5\_\_RANDOMIZEDSEARCHTREE\_H  #define PRACTICE\_5\_\_RANDOMIZEDSEARCHTREE\_H  #include "FileManager.h"  struct Node {  int key;  CarRecord\* ptr;  int priority;  Node\* left;  Node\* right;  Node(int key, CarRecord\* ptr) {  this->key = key;  this->ptr = ptr;  priority = Random\_Func(0, 1000);  left = nullptr;  right = nullptr;  }  };  class RandomizedBST {  private:  Node\* root;  Node\* rotateRight(Node\* node) {  Node\* newRoot = node->left;  node->left = newRoot->right;  newRoot->right = node;  return newRoot;  }  Node\* rotateLeft(Node\* node) {  Node\* newRoot = node->right;  node->right = newRoot->left;  newRoot->left = node;  return newRoot;  }  Node\* insertNode(Node\* node, int key, CarRecord\* ptr) {  if (node == nullptr)  return new Node(key, ptr);  if (key < node->key) {  node->left = insertNode(node->left, key, ptr);  if (node->left->priority > node->priority)  node = rotateRight(node);  } else {  node->right = insertNode(node->right, key, ptr);  if (node->right->priority > node->priority)  node = rotateLeft(node);  }  return node;  }  Node\* deleteNode(Node\* node, int key) {  if (node == nullptr)  return nullptr;  if (key < node->key)  node->left = deleteNode(node->left, key);  else if (key > node->key)  node->right = deleteNode(node->right, key);  else {  if (node->left == nullptr) {  Node\* temp = node->right;  delete node;  node = temp;  } else if (node->right == nullptr) {  Node\* temp = node->left;  delete node;  node = temp;  } else {  if (node->left->priority > node->right->priority) {  node = rotateRight(node);  node->right = deleteNode(node->right, key);  } else {  node = rotateLeft(node);  node->left = deleteNode(node->left, key);  }  }  }  return node;  }  Node\* searchRecursive(Node\* node, int key) {  if (node == nullptr || node->key == key){  return node;  }  if (key < node->key) {  return searchRecursive(node->left, key);  }  else {  return searchRecursive(node->right, key);  }  }  void printTreeRecursive(Node\* node, int level) {  if (node != nullptr) {  printTreeRecursive(node->right, level + 1);  for (int i = 0; i < level; i++) {  std::cout << " ";  }  cout << level;  std::cout << "-> " << node->key << "\\" << node->ptr->key << "\\" << node->ptr->brand << std::endl;  printTreeRecursive(node->left, level + 1);  }  }  public:  void insert(int key, CarRecord\* ptr) {  root = insertNode(root, key, ptr);  }  void remove(int key) {  root = deleteNode(root, key);  }  void printTree() {  printTreeRecursive(root, 0);  }  void search(int key) {  Node\* Found = searchRecursive(root, key);  if (Found != nullptr)  cout << Found->ptr->key << " | " << Found->ptr->brand << "\t| " << Found->ptr->owner << endl;  else {  cout << "The element with this key does not exist." << endl;  }  }  void Load(const string& binaryFileName){  fstream file;  file.open(binaryFileName, ios::in | ios::binary);  CarRecord record;  while (file.read(reinterpret\_cast<char\*>(&record), sizeof(CarRecord))) {  CarRecord\* temp = new CarRecord(record);  insert(record.key, temp);  }  file.close();  }  };  #endif |