**Лабораторная работа**

# **База данных (access, SQL server, PostgreSQL,…) - хранилище данных программного средства (чтение, запись и временное хранение данных программного средства.**

# **Цель**

Изучить основы работы с различными системами управления базами данных (Microsoft Access, SQL Server, PostgreSQL) как хранилищами данных программного средства, освоить методы чтения, записи и временного хранения данных на языке C++.

# **Теория**

## **2.1. Зачем нужны БД ?**

Представьте, что у вас есть куча информации о программах, которые вы используете: их названия, версии, даты установки — всё, что нужно для работы. Теперь подумайте, как это хранить? Можно, конечно, записать всё в блокнот: "Программа 1, версия 1.0", "Программа 2, версия 2.5". Но что, если программ станет сто или тысяча? А если нужно быстро найти программу по имени или изменить версию? Листать блокнот и искать вручную — это долго и неудобно, как искать иголку в стоге сена.

Вот тут и приходят на помощь базы данных (БД) — это как умный шкаф с ящиками, где всё аккуратно разложено по полочкам. Например, у нас есть Microsoft Access, SQL Server и PostgreSQL — это разные виды таких "шкафов". Они помогают организовать данные так, чтобы:

* Быстро находить нужное (например, "покажи все программы с версией выше 1.0").
* Легко добавлять новое (например, "добавь новую программу App1").
* Менять данные (например, "обнови версию программы с ID 1").
* Удалять ненужное (например, "удали программу, которую больше не используем").

Простыми словами, база данных — это место, где данные хранятся в виде таблиц, как в Excel, только гораздо мощнее. Но зачем нам три разные базы — Access, SQL Server и PostgreSQL? Они как разные шкафы для разных задач:

* **Microsoft Access** — это как небольшой шкаф у вас дома: удобно для маленьких проектов, всё хранится в одном файле (например, .accdb), и его легко настроить.
* **SQL Server** — это как огромный шкаф в офисе: он мощный, может хранить данные для целой компании, но требует сервера и сложнее в настройке.
* **PostgreSQL** — это как универсальный шкаф с открытым доступом: он бесплатный, мощный, и его любят разработчики за гибкость.

## **2.2.** **Основы работы с базами данных?**

Базы данных — это реляционные системы управления базами данных (СУБД), которые хранят данные в виде таблиц. Каждая таблица состоит из строк и столбцов, где строки — это записи, а столбцы — это поля (например, ID, Name, Version).

### **2.2.1. Основные СУБД, используемые в работе**

* **Microsoft Access**: Файловая СУБД, хранит данные в файлах .accdb. Подходит для небольших проектов.
* **SQL Server**: Серверная СУБД от Microsoft, предназначена для крупных приложений.
* **PostgreSQL**: Открытая серверная СУБД, популярна среди разработчиков за гибкость и производительность.

### **2.2.2. ODBC (Open Database Connectivity)**

ODBC — это стандартный API, который позволяет программам на C++ подключаться к разным базам данных (Access, SQL Server, PostgreSQL) через специальные драйверы. Для Access используется драйвер Microsoft Access Driver (\*.mdb, \*.accdb).

### **2.1.3. SQL (Structured Query Language)**

SQL — это язык запросов для работы с данными в базах:

* **INSERT**: Добавление новой записи.
* **SELECT**: Извлечение данных.
* **UPDATE**: Изменение данных.
* **DELETE**: Удаление данных.

### **2.1.4. Временное хранение данных**

Временное хранение данных в программе на C++ можно реализовать с помощью контейнеров, таких как std::vector. Данные сначала сохраняются в память, а затем могут быть записаны в базу данных. Это ускоряет работу приложения, так как доступ к памяти быстрее, чем к базе данных.

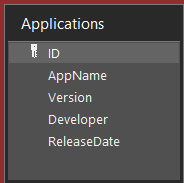
# **Типовой пример**

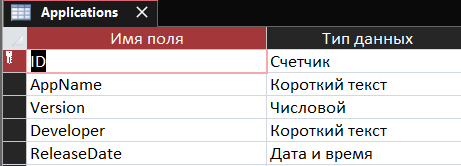
## **3.1. Microsoft Access**

Для начала давайте сделаем БД и свяжем её с нашим проектом в VS. Для начала переходим в Microsoft Access и создаём новый файл

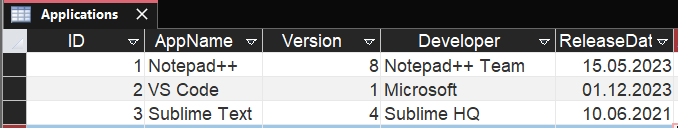


с такой структурой (таблицу называем Applications):

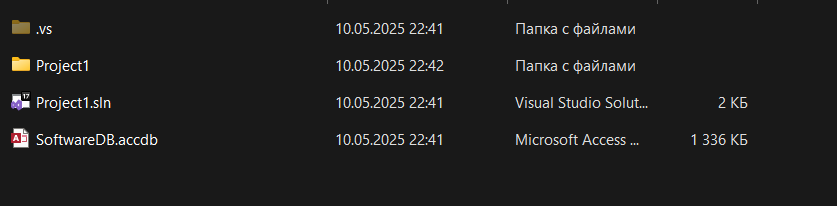




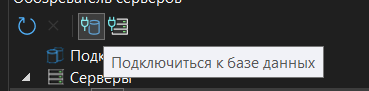
Далее заполняем все поля такой информацией



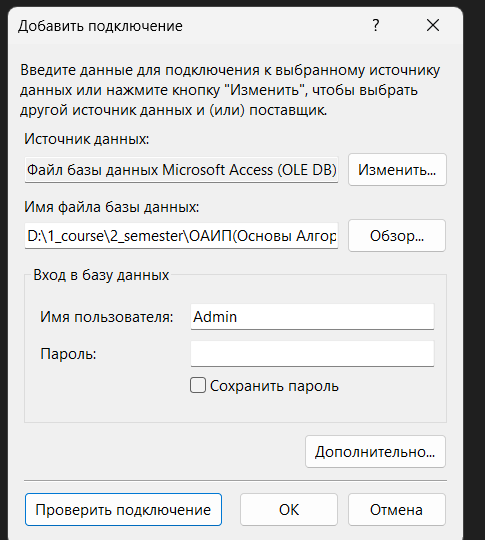
Теперь давайте создадим проект в VS ( я проект назвал Project1 ). После этого переносим нашу БД в папку к проекту и получается такая структура:

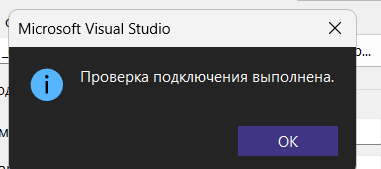


Создаём как обычно пустой проект Нажимаем Вид -> Обозреватель серверов



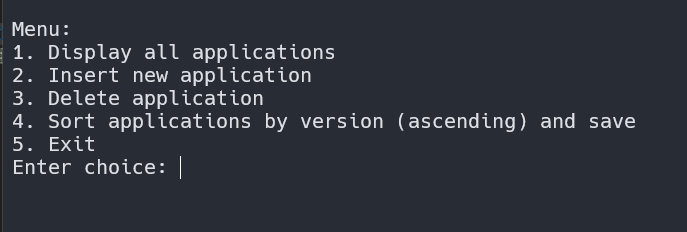
Нажимаем подключиться к базе данных. Дальше в этом окошке в строке имя файла указываем путь к нашей БД.



И теперь нажимаем Проверить подключение. Если отобразилось 

Значит всё хорошо. Теперь напишем main.cpp

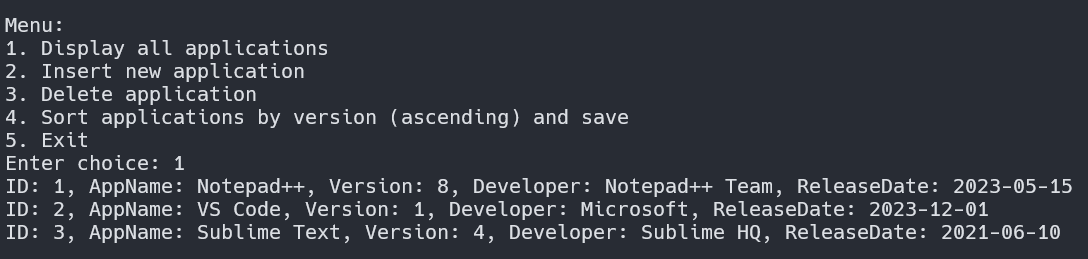
|  |
| --- |
| #include <windows.h>  #include <sqlext.h>  #include <iostream>  #include <vector>  #include <algorithm>  #include <string>  #include <iomanip>  // Функция для проверки формата даты YYYY-MM-DD  // Проверяет, что введённая дата корректна (например, 2023-05-15)  // Возвращает true, если дата правильная, и false, если есть ошибки  bool validateDate(const std::wstring& date) {  // Проверяем длину строки и наличие символов '-' на нужных позициях  if (date.length() != 10 || date[4] != L'-' || date[7] != L'-') {  return false;  }  try {  // Извлекаем год, месяц и день из строки и преобразуем их в числа  int year = std::stoi(date.substr(0, 4)); // Первые 4 символа — год  int month = std::stoi(date.substr(5, 2)); // Символы 5-6 — месяц  int day = std::stoi(date.substr(8, 2)); // Символы 8-9 — день  // Проверяем, что месяц от 1 до 12  if (month < 1 || month > 12) return false;  // Проверяем, что день от 1 до 31  if (day < 1 || day > 31) return false;  // Проверяем, что год в разумных пределах (1900–9999)  if (year < 1900 || year > 9999) return false;  // Проверяем количество дней в месяце (простая проверка, без учёта високосных годов)  if (month == 2 && day > 29) return false; // Февраль не может иметь больше 29 дней  // Апрель, июнь, сентябрь и ноябрь имеют 30 дней  if ((month == 4 || month == 6 || month == 9 || month == 11) && day > 30) return false;  return true; // Дата корректна  }  catch (...) {  return false;  }  }  // Функция для отображения ошибок ODBC  // Если что-то идёт не так при работе с базой данных, эта функция покажет, что именно пошло не так  void showSQLError(SQLSMALLINT handleType, SQLHANDLE handle) {  SQLWCHAR sqlState[1024]; // Буфер для кода состояния ошибки  SQLWCHAR message[2048]; // Буфер для текста ошибки  SQLINTEGER nativeError; // Код ошибки  SQLSMALLINT textLength; // Длина текста ошибки  // Получаем информацию об ошибке  SQLRETURN ret = SQLGetDiagRecW(handleType, handle, 1, sqlState, &nativeError, message, 2048, &textLength);  if (ret == SQL\_SUCCESS || ret == SQL\_SUCCESS\_WITH\_INFO) {  // Выводим текст ошибки и код состояния  std::wcout << L"ODBC Error: " << message << L" [State: " << sqlState << L"]\n";  }  }  // Структура для хранения данных одной записи из таблицы Applications  struct Application {  SQLINTEGER id; // ID записи (автоинкрементное поле в базе)  std::wstring appName; // Название приложения (например, Notepad++)  double version; // Версия приложения (например, 8.5)  std::wstring developer; // Разработчик (например, Notepad++ Team)  std::wstring releaseDate; // Дата выпуска (например, 2023-05-15)  };  // Функция для подключения к базе данных Access  // Устанавливает соединение с файлом SoftwareDB.accdb  bool connectToDB(SQLHENV& hEnv, SQLHDBC& hDbc) {  // Создаём окружение (environment) для работы с ODBC  SQLAllocHandle(SQL\_HANDLE\_ENV, SQL\_NULL\_HANDLE, &hEnv);  // Устанавливаем версию ODBC (используем версию 3)  SQLSetEnvAttr(hEnv, SQL\_ATTR\_ODBC\_VERSION, (void\*)SQL\_OV\_ODBC3, 0);  // Создаём дескриптор соединения (connection handle)  SQLAllocHandle(SQL\_HANDLE\_DBC, hEnv, &hDbc);  // Строка подключения: указываем драйвер и путь к файлу базы данных  SQLWCHAR connStr[] = L"Driver={Microsoft Access Driver (\*.mdb, \*.accdb)};DBQ=D:\\1\_course\\2\_semester\\ОАИП(Основы Алгоритмизации и Программирования)\\лабы\_белодед\\код ко второй\\для бд\\SoftwareDB.accdb;";  SQLWCHAR outConnStr[1024]; // Буфер для строки подключения после выполнения  SQLSMALLINT outConnStrLen; // Длина строки подключения  // Выполняем подключение к базе данных  SQLRETURN ret = SQLDriverConnectW(hDbc, NULL, connStr, SQL\_NTS, outConnStr, 1024, &outConnStrLen, SQL\_DRIVER\_COMPLETE);  if (!SQL\_SUCCEEDED(ret)) {  // Если подключение не удалось, показываем ошибку  showSQLError(SQL\_HANDLE\_DBC, hDbc);  return false;  }  return true; // Подключение успешно  }  // Функция для чтения всех данных из таблицы Applications  // Возвращает вектор записей (Application), которые хранятся в базе  std::vector<Application> readData(SQLHDBC hDbc) {  std::vector<Application> apps; // Вектор для хранения всех записей  SQLHSTMT hStmt; // Дескриптор запроса (statement handle)  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt); // Создаём дескриптор запроса  // Выполняем SQL-запрос: выбираем все столбцы из таблицы Applications  SQLRETURN ret = SQLExecDirectW(hStmt, (SQLWCHAR\*)L"SELECT ID, AppName, Version, Developer, ReleaseDate FROM Applications;", SQL\_NTS);  if (SQL\_SUCCEEDED(ret)) {  // Переменные для хранения данных из столбцов  SQLINTEGER id;  SQLWCHAR appName[101], developer[101], releaseDate[50];  SQLDOUBLE version;  SQLLEN idLen, appNameLen, versionLen, developerLen, releaseDateLen;  // Читаем строки из таблицы по одной  while (SQLFetch(hStmt) == SQL\_SUCCESS) {  Application app = { 0 }; // Создаём пустую структуру для текущей записи  // Читаем ID (целочисленное значение)  ret = SQLGetData(hStmt, 1, SQL\_C\_LONG, &id, 0, &idLen);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error fetching ID\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  break;  }  app.id = id;  // Читаем AppName (строку)  ret = SQLGetData(hStmt, 2, SQL\_C\_WCHAR, appName, sizeof(appName), &appNameLen);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error fetching AppName\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  break;  }  app.appName = appName;  // Читаем Version (число с плавающей точкой)  ret = SQLGetData(hStmt, 3, SQL\_C\_DOUBLE, &version, 0, &versionLen);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error fetching Version\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  break;  }  app.version = version;  // Читаем Developer (строку)  ret = SQLGetData(hStmt, 4, SQL\_C\_WCHAR, developer, sizeof(developer), &developerLen);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error fetching Developer\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  break;  }  app.developer = developer;  // Читаем ReleaseDate (строку)  ret = SQLGetData(hStmt, 5, SQL\_C\_WCHAR, releaseDate, sizeof(releaseDate), &releaseDateLen);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error fetching ReleaseDate\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  app.releaseDate = L"Invalid Date";  }  else {  // Если дата содержит время (например, 2023-05-15 00:00:00), обрезаем его  std::wstring dateStr(releaseDate);  size\_t spacePos = dateStr.find(L" ");  if (spacePos != std::wstring::npos) {  app.releaseDate = dateStr.substr(0, spacePos);  }  else {  app.releaseDate = dateStr;  }  }  apps.push\_back(app); // Добавляем запись в вектор  }  }  else {  showSQLError(SQL\_HANDLE\_STMT, hStmt); // Показываем ошибку, если запрос не удался  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt); // Освобождаем дескриптор запроса  return apps; // Возвращаем вектор записей  }  // Функция для вывода данных на экран  // Принимает вектор записей и печатает их в консоль  void displayData(const std::vector<Application>& apps) {  for (const auto& app : apps) {  std::wcout << L"ID: " << app.id << L", AppName: " << app.appName  << L", Version: " << app.version  << L", Developer: " << app.developer  << L", ReleaseDate: " << app.releaseDate << L"\n";  }  }  // Функция для вставки новой записи в таблицу Applications  void insertData(SQLHDBC hDbc, const std::wstring& appName, double version, const std::wstring& developer, const std::wstring& releaseDate) {  // Проверяем, что дата корректна  if (!validateDate(releaseDate)) {  std::wcout << L"Invalid date format. Please use YYYY-MM-DD with valid values (e.g., 2023-05-15).\n";  return;  }  SQLHSTMT hStmt; // Создаём дескриптор для SQL-запроса  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  // Формируем SQL-запрос для вставки новой записи  // Для даты в Access нужно использовать формат #YYYY-MM-DD#  SQLWCHAR query[512];  swprintf\_s(query, 512, L"INSERT INTO Applications (AppName, Version, Developer, ReleaseDate) VALUES ('%s', %f, '%s', #%s#);",  appName.c\_str(), version, developer.c\_str(), releaseDate.c\_str());  // Выполняем запрос  SQLRETURN ret = SQLExecDirectW(hStmt, query, SQL\_NTS);  if (SQL\_SUCCEEDED(ret)) {  std::wcout << L"Record inserted successfully.\n"; // Сообщаем об успешной вставке  std::wcout << L"\nUpdated database:\n";  displayData(readData(hDbc)); // Показываем обновлённую таблицу  }  else {  showSQLError(SQL\_HANDLE\_STMT, hStmt); // Показываем ошибку, если запрос не удался  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt); // Освобождаем дескриптор  }  // Функция для удаления записи из таблицы по ID  void deleteData(SQLHDBC hDbc, int id) {  SQLHSTMT hStmt; // Создаём дескриптор для SQL-запроса  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  // Формируем SQL-запрос для удаления записи с заданным ID  SQLWCHAR query[256];  swprintf\_s(query, 256, L"DELETE FROM Applications WHERE ID=%d;", id);  // Выполняем запрос  SQLRETURN ret = SQLExecDirectW(hStmt, query, SQL\_NTS);  if (SQL\_SUCCEEDED(ret)) {  std::wcout << L"Record deleted successfully.\n"; // Сообщаем об успешном удалении  std::wcout << L"\nUpdated database:\n";  displayData(readData(hDbc)); // Показываем обновлённую таблицу  }  else {  showSQLError(SQL\_HANDLE\_STMT, hStmt); // Показываем ошибку, если запрос не удался  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt); // Освобождаем дескриптор  }  // Функция для сортировки записей по версии и сохранения их в базу  void sortByVersionAndSave(SQLHDBC hDbc) {  // 1. Читаем все записи из базы  std::vector<Application> apps = readData(hDbc);  // 2. Сортируем записи в памяти по возрастанию версии  std::sort(apps.begin(), apps.end(), [](const Application& a, const Application& b) {  return a.version < b.version;  });  // 3. Очищаем таблицу Applications перед вставкой отсортированных данных  SQLHSTMT hStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  SQLRETURN ret = SQLExecDirectW(hStmt, (SQLWCHAR\*)L"DELETE FROM Applications;", SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error clearing table before sorting\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  return;  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  // 4. Вставляем отсортированные записи обратно в таблицу  for (const auto& app : apps) {  SQLHSTMT insertStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &insertStmt);  // Формируем запрос для вставки каждой записи  SQLWCHAR query[512];  swprintf\_s(query, 512, L"INSERT INTO Applications (AppName, Version, Developer, ReleaseDate) VALUES ('%s', %f, '%s', #%s#);",  app.appName.c\_str(), app.version, app.developer.c\_str(), app.releaseDate.c\_str());  ret = SQLExecDirectW(insertStmt, query, SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error inserting sorted record\n";  showSQLError(SQL\_HANDLE\_STMT, insertStmt);  SQLFreeHandle(SQL\_HANDLE\_STMT, insertStmt);  return;  }  SQLFreeHandle(SQL\_HANDLE\_STMT, insertStmt);  }  // 5. Показываем результат сортировки  std::wcout << L"Applications sorted and saved successfully.\n";  std::wcout << L"\nUpdated database:\n";  displayData(readData(hDbc));  }  // Главная функция программы  int main() {  SetConsoleOutputCP(CP\_UTF8); // Устанавливаем кодировку консоли для поддержки Unicode  SQLHENV hEnv = nullptr; // Переменная для окружения ODBC  SQLHDBC hDbc = nullptr; // Переменная для соединения с базой данных  // Подключаемся к базе данных  if (!connectToDB(hEnv, hDbc)) {  return 1; // Если подключение не удалось, завершаем программу  }  int choice; // Переменная для выбора опции меню  std::wstring appName, developer, releaseDate; // Переменные для ввода данных  double version;  int id;  // Основной цикл программы  while (true) {  // Выводим меню с доступными опциями  std::wcout << L"\nMenu:\n";  std::wcout << L"1. Display all applications\n"; // Показать все записи  std::wcout << L"2. Insert new application\n"; // Добавить новую запись  std::wcout << L"3. Delete application\n"; // Удалить запись  std::wcout << L"4. Sort applications by version (ascending) and save\n"; // Сортировать и сохранить  std::wcout << L"5. Exit\n"; // Выход из программы  std::wcout << L"Enter choice: ";  std::cin >> choice; // Считываем выбор пользователя  std::cin.ignore(); // Очищаем буфер ввода  if (choice == 5) break; // Если выбрана опция 5, выходим из цикла  // Обрабатываем выбор пользователя  switch (choice) {  case 1: // Показать все записи  displayData(readData(hDbc));  break;  case 2: // Добавить новую запись  std::wcout << L"Enter AppName: ";  std::getline(std::wcin, appName); // Считываем название приложения  std::wcout << L"Enter Version: ";  std::cin >> version; // Считываем версию  std::cin.ignore(); // Очищаем буфер  std::wcout << L"Enter Developer: ";  std::getline(std::wcin, developer); // Считываем разработчика  std::wcout << L"Enter ReleaseDate (YYYY-MM-DD): ";  std::getline(std::wcin, releaseDate); // Считываем дату выпуска  insertData(hDbc, appName, version, developer, releaseDate); // Добавляем запись  break;  case 3: // Удалить запись  std::wcout << L"Enter ID to delete: ";  std::cin >> id; // Считываем ID записи для удаления  deleteData(hDbc, id); // Удаляем запись  break;  case 4: // Сортировать и сохранить  sortByVersionAndSave(hDbc);  break;  default:  std::wcout << L"Invalid choice.\n"; // Сообщаем о некорректном выборе  }  }  // Закрываем соединение с базой данных перед завершением  if (hDbc) {  SQLDisconnect(hDbc); // Отключаемся от базы  SQLFreeHandle(SQL\_HANDLE\_DBC, hDbc); // Освобождаем дескриптор соединения  }  if (hEnv) SQLFreeHandle(SQL\_HANDLE\_ENV, hEnv); // Освобождаем окружение  return 0; // Завершаем программу  } |

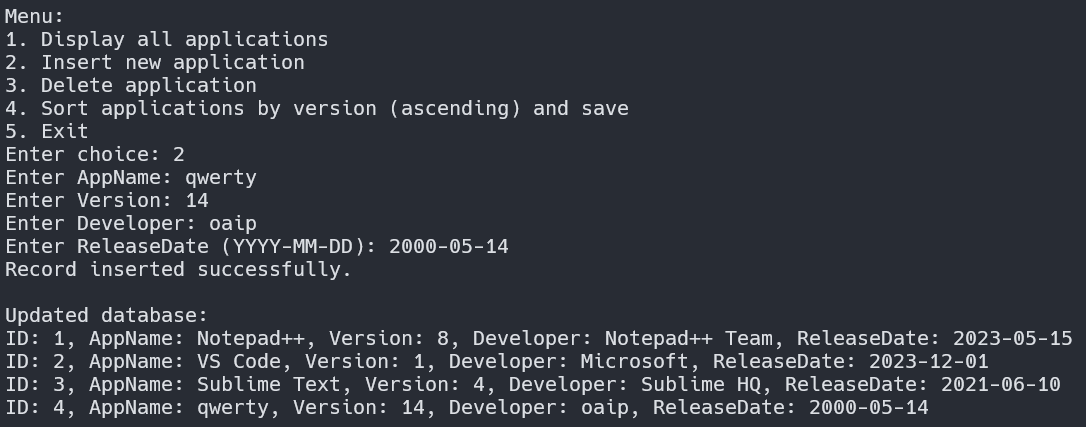
Если всё запустилось , то у вас после запуска выведется меню 

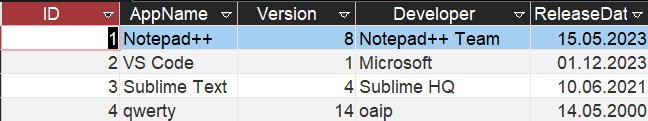
Программа реализует основные задачи управления данными в таблице Applications:

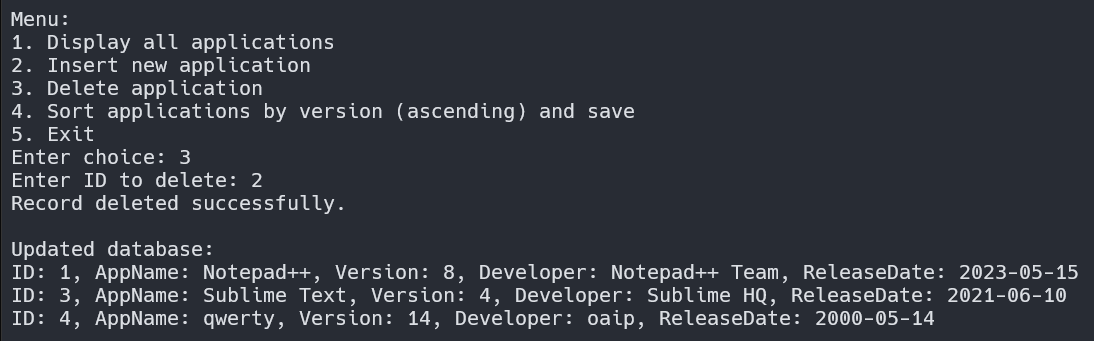
* Отображение всех приложений из базы данных.
* Вставка нового приложения в таблицу.
* Удаление приложения по ID.
* Сортировка приложений по версии (по возрастанию) с сохранением в базу.

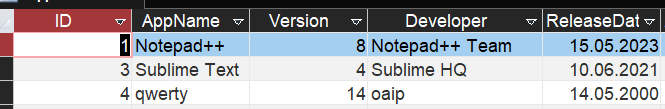
Если мы добавляем/сортируем/удаляем нам нужно закрыть и открыть нащу БД на компьютере, т.к. в онлайне данные не меняются. Пример работы программы:

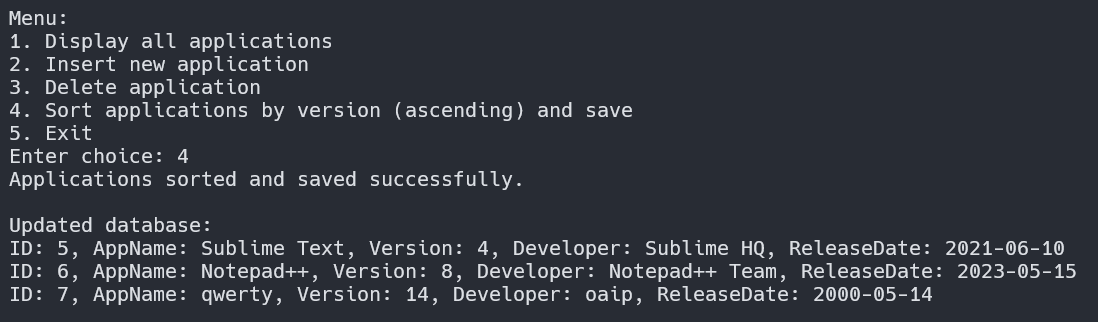














## **3.2. SQL Server**

## **3.3. PostgreSQL**

# **Задачи по теме**

|  |  |
| --- | --- |
| **№ варианта** | **Условие задачи** |
| 1 | Создайте базу данных StudentsDB.accdb в Microsoft Access с таблицей Students, имеющей следующую структуру:   * ID — уникальный идентификатор студента. * FullName— полное имя студента. * AverageScore— средний балл (от 0 до 100). * Major— специальность (например, "Computer Science"). * EnrollmentDate — дата поступления.   Добавьте записи по своему усмотрению.  **Задача**:   1. Подключитесь к базе данных через ODBC. 2. Загружайте данные в std::vector<Student> для временного хранения. 3. Реализуйте программу с меню:  * Отобразить всех студентов. * Сортировать студентов по среднему баллу (по убыванию) и сохранить в базу. * Выход.  1. После сортировки обновите базу данных и выведите результат. |
| 2 |  |
| 3 |  |
| 4 | Создайте базу данных LibraryDB.accdb с таблицей Books, имеющей следующую структуру:   * ID — уникальный идентификатор книги. * Title — название книги. * Author — автор книги. * Genre — жанр (например, "Fiction", "Science"). * YearPublished — год публикации.   **Задача**:   1. Подключитесь к базе данных через ODBC. 2. Загружайте данные в std::vector<Book> для временного хранения. 3. Реализуйте программу с меню:  * Отобразить все книги. * Фильтровать книги по жанру (пользователь вводит жанр, например, "Fantasy"). * Выход.  1. После фильтрации выведите только те книги, которые соответствуют введённому жанру. |
| 5 |  |
| 6 |  |
| 7 | Создайте базу данных StoreDB.accdb с таблицей Products, имеющей следующую структуру:   * ID — уникальный идентификатор продукта. * ProductName — название продукта. * Price — цена продукта (в долларах). * Category — категория (например, "Electronics"). * InStock — наличие на складе (Yes = 1, No = 0).   **Задача**:   1. Подключитесь к базе данных через ODBC. 2. Загружайте данные в std::vector<Product> для временного хранения. 3. Реализуйте программу с меню:  * Отобразить все продукты. * Найти продукты в заданном ценовом диапазоне (пользователь вводит минимальную и максимальную цену). * Выход.  1. После поиска выведите только те продукты, цена которых попадает в указанный диапазон. |
| 8 |  |
| 9 |  |
| 10 | Создайте базу данных EmployeesDB.accdb с таблицей Employees, имеющей следующую структуру:   * ID — уникальный идентификатор сотрудника. * Name — имя сотрудника. * Salary — зарплата (в долларах). * Department — отдел (например, "IT"). * HireDate — дата приёма на работу.   **Задача**:   1. Подключитесь к базе данных через ODBC. 2. Загружайте данные в std::vector<Employee> для временного хранения. 3. Реализуйте программу с меню:  * Отобразить всех сотрудников. * Сортировать сотрудников сначала по стажу (от большего к меньшему, на основе HireDate), а затем по зарплате (по убыванию), и сохранить в базу. * Выход.  1. После сортировки обновите базу данных и выведите результат. |
| 11 |  |
| 12 |  |
| 13 | Создайте базу данных MoviesDB.accdb с таблицей Movies, имеющей следующую структуру:   * ID — уникальный идентификатор фильма. * Title — название фильма. * Director — режиссёр. * Genre — жанр (например, "Action"). * Rating — рейтинг (от 0 до 10).   **Задача**:   1. Подключитесь к базе данных через ODBC. 2. Загружайте данные в std::vector<Movie> для временного хранения. 3. Реализуйте программу с меню:  * Отобразить все фильмы. * Подсчитать средний рейтинг фильмов для заданного жанра (пользователь вводит жанр, например, "Sci-Fi"). * Выход.  1. После ввода жанра выведите средний рейтинг фильмов этого жанра. |
| 14 |  |
| 15 |  |

# **Дополнительные задачи**

## **6.1 Анализатор временных рядов датчиков**

## **6.2 Умный подбор товаров с ограничением по весу**

## **6.3 Система мониторинга движущихся объектов с прогнозированием столкновений**

## **6.4 Многопользовательская система управления транзакциями с MVCC**

## **6.5 Интеллектуальный текстовый индекс**

# **Контрольные вопросы**

**1. Что такое красно-чёрное дерево и почему оно называется именно так?**

**2. Перечислите основные правила красно-чёрного дерева.**

**3. Почему красно-чёрное дерево подходит для индексирования в базах данных?**

**4. Какой цвет имеет новый узел при вставке в красно-чёрное дерево и почему?**

**5. Опишите случай балансировки при вставке, когда у нового узла красный родитель и красный дядя.**

**6. Какой случай балансировки при удалении возникает, если брат удаляемого узла красный?**

**7. Что такое чёрная высота и почему она важна для красно-чёрного дерева?**

**8. Как красно-чёрное дерево обеспечивает логарифмическую сложность операций?**

**9. В чём преимущество красно-чёрного дерева перед обычным бинарным деревом поиска (BST) при использовании в базах данных?**

**10. Какие ограничения красно-чёрного дерева для индексирования в базах данных?**

**11. Какой метод используется в программе для поиска студента по ID? Опишите его работу.**

**12. Как в программе реализовано удаление студента из базы данных?**

**13. Почему в программе используется вектор students в дополнение к красно-чёрному дереву?**

**14. Как красно-чёрное дерево может быть использовано для диапазонных запросов?**

**15. Сравните красно-чёрное дерево с B-деревом для задач индексирования.**

# **8. Решения к задачам повышенной сложности**

|  |  |
| --- | --- |
| **№** | **Условие задачи** |
| 1 | Реализуйте систему индексирования временных рядов с использованием красно-чёрного дерева. Каждый временной ряд принадлежит датчику (ID, Название датчика) и содержит временные метки с измерениями (Timestamp, Value). Постройте красно-чёрное дерево по ID датчика. Реализуйте метод, который для заданного датчика прогнозирует следующее значение, используя линейную регрессию (на основе последних N измерений). Также реализуйте метод для поиска аномалий (значений, которые отклоняются от среднего более чем на 2 стандартных отклонения).  **Входные данные** Файл devices.txt:  ID SensorName  101 TempSensor  102 PressureSensor  Файл measurements.txt:  SensorID Timestamp Value  101 2025-04-12 10:00 20.5  101 2025-04-12 10:05 21.0  101 2025-04-12 10:10 22.0  101 2025-04-12 10:15 30.0  102 2025-04-12 10:00 1013.0  102 2025-04-12 10:05 1012.5  102 2025-04-12 10:10 1014.0  Прогнозируйте следующее значение для датчика с ID 101 (используя последние 3 измерения) и найдите аномалии для этого датчика.  **Ожидаемый результат**  Прогноз для устройства 101: 2025  Тренд устройства 101: Стабильность  Устройства с аномальным трендом:    **Схема базы данных** Добавлена таблица Measurements для хранения измерений и таблица Predictions для хранения прогнозов.  #include <iostream>  #include <fstream>  #include <string>  #include <sstream>  #include <vector>  #include <cmath>  using namespace *std*;  // Структура для хранения одного измерения  struct Measurement {  *string* timestamp; // Временная метка измерения  double value; // Значение измерения  };  // Структура для устройства с датчиками  struct Device {  int id; // Уникальный идентификатор устройства  *string* name; // Название устройства  *vector*<Measurement> measurements; // Вектор всех измерений устройства  // Конструктор устройства  Device(int \_id, *string* \_name) {  id = \_id;  name = \_name;  }  };  // Перечисление для цветов узлов красно-черного дерева  enum Color { RED, BLACK };  // Структура узла красно-черного дерева  struct RBNode {  int key; // Ключ узла (id устройства)  Device\* data; // Указатель на данные устройства  RBNode\* left; // Указатель на левого потомка  RBNode\* right; // Указатель на правого потомка  RBNode\* parent; // Указатель на родителя  Color color; // Цвет узла  // Конструктор узла  RBNode(int k, Device\* d) {  key = k;  data = d;  left = nullptr;  right = nullptr;  parent = nullptr;  color = RED; // Новые узлы всегда красные  }  };  // Структура красно-черного дерева  struct RBTree {  RBNode\* root; // Корень дерева  RBNode\* nil; // Специальный nil-узел (листья дерева)  // Конструктор дерева  RBTree() {  nil = new RBNode(0, nullptr);  nil->color = BLACK; // Nil-узлы всегда черные  nil->*left* = nil;  nil->*right* = nil;  root = nil;  }  // Левый поворот вокруг узла x  void leftRotate(RBNode\* x) {  RBNode\* y = x->*right*;  x->*right* = y->*left*;  if (y->*left* != nil) y->*left*->parent = x;  y->parent = x->parent;  if (x->parent == nullptr) root = y;  else if (x == x->parent->*left*) x->parent->*left* = y;  else x->parent->*right* = y;  y->*left* = x;  x->parent = y;  }  // Правый поворот вокруг узла y  void rightRotate(RBNode\* y) {  RBNode\* x = y->*left*;  y->*left* = x->*right*;  if (x->*right* != nil) x->*right*->parent = y;  x->parent = y->parent;  if (y->parent == nullptr) root = x;  else if (y == y->parent->*right*) y->parent->*right* = x;  else y->parent->*left* = x;  x->*right* = y;  y->parent = x;  }  // Восстановление свойств дерева после вставки  void fixInsert(RBNode\* z) {  while (z->parent != nullptr && z->parent->color == RED) {  if (z->parent == z->parent->parent->*left*) {  RBNode\* y = z->parent->parent->*right*;  if (y->color == RED) {  // Случай 1: дядя красный  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->*right*) {  // Случай 2: дядя черный, z - правый потомок  z = z->parent;  leftRotate(z);  }  // Случай 3: дядя черный, z - левый потомок  z->parent->color = BLACK;  z->parent->parent->color = RED;  rightRotate(z->parent->parent);  }  }  else {  // Симметричные случаи для правого поддерева  RBNode\* y = z->parent->parent->*left*;  if (y->color == RED) {  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->*left*) {  z = z->parent;  rightRotate(z);  }  z->parent->color = BLACK;  z->parent->parent->color = RED;  leftRotate(z->parent->parent);  }  }  }  root->color = BLACK; // Корень всегда черный  }  // Вставка нового узла в дерево  void insert(int key, Device\* data) {  RBNode\* z = new RBNode(key, data);  z->*left* = nil;  z->*right* = nil;  RBNode\* y = nullptr;  RBNode\* x = root;  // Поиск места для вставки  while (x != nil) {  y = x;  if (z->key < x->key) x = x->*left*;  else x = x->*right*;  }  // Установка родителя  z->parent = y;  if (y == nullptr) root = z;  else if (z->key < y->key) y->*left* = z;  else y->*right* = z;  // Балансировка дерева  fixInsert(z);  }  // Поиск устройства по id  Device\* search(int key) {  RBNode\* current = root;  while (current != nil) {  if (key == current->key) return current->*data*;  else if (key < current->key) current = current->*left*;  else current = current->*right*;  }  return nullptr; // Устройство не найдено  }  };  // Класс для работы с базой данных устройств  struct DeviceDB {  RBTree index; // Красно-черное дерево для быстрого поиска  *vector*<Device\*> devices; // Все устройства  *string* devicesFile; // Файл с данными устройств  *string* measurementsFile; // Файл с измерениями  // Конструктор  DeviceDB(*string* dFile, *string* mFile) {  devicesFile = dFile;  measurementsFile = mFile;  loadDevices(); // Загрузка устройств при создании  loadMeasurements(); // Загрузка измерений при создании  }  // Загрузка устройств из файла  void loadDevices() {  *ifstream* file(devicesFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << devicesFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  int id;  *string* name;  ss >> id >> name;  // Создание нового устройства и добавление в базу  Device\* device = new Device(id, name);  devices.*push\_back*(device);  index.insert(id, device); // Вставка в индексное дерево  }  file.*close*();  }  // Загрузка измерений из файла  void loadMeasurements() {  *ifstream* file(measurementsFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << measurementsFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  int deviceID;  *string* timestamp;  double value;  ss >> deviceID;  ss.*ignore*(1);  *getline*(ss, timestamp, ' ');  ss >> value;  // Поиск устройства и добавление измерения  Device\* device = index.search(deviceID);  if (device) {  Measurement m;  m.timestamp = timestamp;  m.value = value;  device->measurements.*push\_back*(m);  }  }  file.*close*();  }  // Прогнозирование следующего значения с помощью экспоненциального сглаживания  double predictNextValue(int deviceID) {  Device\* device = index.search(deviceID);  if (!device || device->measurements.*size*() < 2) return -1;  double alpha = 0.3; // Коэффициент сглаживания  double forecast = device->measurements[0].value;  // Применение формулы экспоненциального сглаживания  for (*size\_t* i = 1; i < device->measurements.*size*(); i++) {  forecast = alpha \* device->measurements[i].value + (1 - alpha) \* forecast;  }  return forecast;  }  // Определение тренда на основе скользящего среднего  *string* determineTrend(Device\* device) {  if (device->measurements.*size*() < 3) return "Недостаточно данных";  *vector*<double> movingAverages;  // Расчет скользящего среднего по 3 точкам  for (*size\_t* i = 2; i < device->measurements.*size*(); i++) {  double avg = (device->measurements[i].value +  device->measurements[i - 1].value +  device->measurements[i - 2].value) / 3.0;  movingAverages.*push\_back*(avg);  }  if (movingAverages.*size*() < 2) return "Недостаточно данных";  // Анализ изменения скользящего среднего  double diff = movingAverages.*back*() - movingAverages[movingAverages.*size*() - 2];  if (diff > 0.5) return "Рост";  else if (diff < -0.5) return "Спад";  else return "Стабильность";  }  // Поиск аномальных изменений тренда  void findAnomalousTrends() {  *cout* << "Устройства с аномальным трендом:\n";  for (Device\* device : devices) {  if (device->measurements.*size*() < 4) continue;  *vector*<*string*> trends;  // Анализ тренда на разных временных отрезках  for (*size\_t* i = 3; i < device->measurements.*size*(); i++) {  Device tempDevice(device->id, device->name);  for (*size\_t* j = 0; j <= i; j++) {  tempDevice.measurements.*push\_back*(device->measurements[j]);  }  trends.*push\_back*(determineTrend(&tempDevice));  }  // Выявление резкого роста после стабильности  if (trends.*size*() >= 2 &&  trends[trends.*size*() - 2] == "Стабильность" &&  trends.*back*() == "Рост") {  *cout* << device->name << " (резкий рост после стабильности)\n";  }  }  }  };  int *main*() {  // Создание и инициализация базы данных  DeviceDB db("devices.txt", "measurements.txt");  // Пример использования функционала  double forecast = db.predictNextValue(101);  *cout* << "Прогноз для устройства 101: " << forecast << "\n";  Device\* device = db.index.search(101);  *cout* << "Тренд устройства 101: " << db.determineTrend(device) << "\n";  // Поиск аномальных трендов  db.findAnomalousTrends();  return 0;  } |
| 2 | Реализуйте систему индексирования товаров с использованием красно-чёрного дерева. Каждый товар имеет поля: ID, Название, Цена, Вес, Категория. Постройте красно-чёрное дерево по ID товара. Реализуйте метод, который выполняет многомерный запрос: находит товары в заданном диапазоне цен, веса и из заданной категории. Добавьте метод, который оптимизирует выбор товаров для доставки (максимизирует суммарную стоимость при ограничении на вес).  **Входные данные** Файл products.txt:  ID ProductName Price Weight Category  301 ProductA 100 2 Electronics  302 ProductB 150 3 Electronics  303 ProductC 80 1 Clothing  Найдите товары с ценой 90-160, весом 1-3, из категории Electronics. Оптимизируйте выбор товаров для доставки с максимальным весом 4.  **Ожидаемый результат**  Товары (цена 90-160, вес 1-3, категория Electronics):  ID 301 ProductA, Price: 100, Weight: 2  ID 302 ProductB, Price: 150, Weight: 3  Оптимальный выбор для доставки (макс. вес 4):  ID 303 ProductC, Price: 80, Weight: 1  ID 301 ProductA, Price: 100, Weight: 2  Общая стоимость: 180    #include <iostream>  #include <fstream>  #include <string>  #include <sstream>  #include <vector>  #include <algorithm>  using namespace *std*;  // Структура для хранения информации о товаре  struct Product {  int id; // Уникальный идентификатор товара  *string* name; // Название товара  int price; // Цена товара  int weight; // Вес товара  *string* category; // Категория товара  // Конструктор товара  Product(int \_id, *string* \_name, int \_price, int \_weight, *string* \_cat) {  id = \_id;  name = \_name;  price = \_price;  weight = \_weight;  category = \_cat;  }  };  // Перечисление цветов для красно-черного дерева  enum Color { RED, BLACK };  // Структура узла красно-черного дерева  struct RBNode {  int key; // Ключ (id товара)  Product\* data; // Указатель на данные товара  RBNode\* left; // Левый потомок  RBNode\* right; // Правый потомок  RBNode\* parent; // Родительский узел  Color color; // Цвет узла  // Конструктор узла  RBNode(int k, Product\* d) {  key = k;  data = d;  left = nullptr;  right = nullptr;  parent = nullptr;  color = RED; // Новые узлы всегда красные  }  };  // Структура красно-черного дерева  struct RBTree {  RBNode\* root; // Корень дерева  RBNode\* nil; // Специальный nil-узел (листья дерева)  // Конструктор дерева  RBTree() {  nil = new RBNode(0, nullptr);  nil->color = BLACK; // Nil-узлы всегда черные  nil->*left* = nil;  nil->*right* = nil;  root = nil;  }  // Левый поворот вокруг узла x  void leftRotate(RBNode\* x) {  RBNode\* y = x->*right*; // y становится правым потомком x  x->*right* = y->*left*; // Левый потомок y становится правым потомком x  if (y->*left* != nil)  y->*left*->parent = x;  y->parent = x->parent; // Перенос родителя  if (x->parent == nullptr)  root = y;  else if (x == x->parent->*left*)  x->parent->*left* = y;  else  x->parent->*right* = y;  y->*left* = x; // x становится левым потомком y  x->parent = y;  }  // Правый поворот вокруг узла y  void rightRotate(RBNode\* y) {  RBNode\* x = y->*left*; // x становится левым потомком y  y->*left* = x->*right*; // Правый потомок x становится левым потомком y  if (x->*right* != nil)  x->*right*->parent = y;  x->parent = y->parent; // Перенос родителя  if (y->parent == nullptr)  root = x;  else if (y == y->parent->*right*)  y->parent->*right* = x;  else  y->parent->*left* = x;  x->*right* = y; // y становится правым потомком x  y->parent = x;  }  // Восстановление свойств дерева после вставки  void fixInsert(RBNode\* z) {  // Пока родитель красный (нарушение свойства красно-черного дерева)  while (z->parent != nullptr && z->parent->color == RED) {  if (z->parent == z->parent->parent->*left*) {  RBNode\* y = z->parent->parent->*right*; // Дядя узла z  if (y->color == RED) {  // Случай 1: дядя красный  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->*right*) {  // Случай 2: дядя черный, z - правый потомок  z = z->parent;  leftRotate(z);  }  // Случай 3: дядя черный, z - левый потомок  z->parent->color = BLACK;  z->parent->parent->color = RED;  rightRotate(z->parent->parent);  }  }  else {  // Симметричные случаи для правого поддерева  RBNode\* y = z->parent->parent->*left*;  if (y->color == RED) {  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->*left*) {  z = z->parent;  rightRotate(z);  }  z->parent->color = BLACK;  z->parent->parent->color = RED;  leftRotate(z->parent->parent);  }  }  }  root->color = BLACK; // Корень всегда черный  }  // Вставка нового узла в дерево  void insert(int key, Product\* data) {  RBNode\* z = new RBNode(key, data);  z->*left* = nil;  z->*right* = nil;  RBNode\* y = nullptr;  RBNode\* x = root;  // Поиск места для вставки  while (x != nil) {  y = x;  if (z->key < x->key)  x = x->*left*;  else  x = x->*right*;  }  // Установка родителя  z->parent = y;  if (y == nullptr)  root = z;  else if (z->key < y->key)  y->*left* = z;  else  y->*right* = z;  // Балансировка дерева  fixInsert(z);  }  // Рекурсивный поиск товаров в заданном диапазоне  void findInRange(RBNode\* node, int minPrice, int maxPrice,  int minWeight, int maxWeight, *string* category) {  if (node == nil) return;  // Сначала идем в левое поддерево (обход в порядке возрастания)  findInRange(node->*left*, minPrice, maxPrice, minWeight, maxWeight, category);  Product\* product = node->*data*;  // Проверка соответствия критериям поиска  if (product->price >= minPrice && product->price <= maxPrice &&  product->weight >= minWeight && product->weight <= maxWeight &&  product->category == category) {  // Вывод информации о найденном товаре  *cout* << "ID " << product->id << " " << product->name  << ", Price: " << product->price  << ", Weight: " << product->weight << "\n";  }  // Затем идем в правое поддерево  findInRange(node->*right*, minPrice, maxPrice, minWeight, maxWeight, category);  }  };  // Класс для работы с базой данных товаров  struct ProductDB {  RBTree index; // Красно-черное дерево для индексации  *vector*<Product\*> products; // Все товары  *string* inputFile; // Имя входного файла  // Конструктор базы данных  ProductDB(*string* file) {  inputFile = file;  loadFromFile(); // Загрузка данных при создании  }  // Загрузка данных из файла  void loadFromFile() {  *ifstream* file(inputFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << inputFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  int id, price, weight;  *string* name, category;  // Чтение данных о товаре из строки  ss >> id >> name >> price >> weight >> category;  // Создание нового товара и добавление в базу  Product\* product = new Product(id, name, price, weight, category);  products.*push\_back*(product);  index.insert(id, product); // Вставка в индексное дерево  }  file.*close*();  }  // Поиск товаров в заданном диапазоне параметров  void findInRange(int minPrice, int maxPrice,  int minWeight, int maxWeight, *string* category) {  *cout* << "Товары (цена " << minPrice << "-" << maxPrice  << ", вес " << minWeight << "-" << maxWeight  << ", категория " << category << "):\n";  index.findInRange(index.root, minPrice, maxPrice, minWeight, maxWeight, category);  }  // Оптимизация доставки (жадный алгоритм)  void optimizeDelivery(int maxWeight) {  // Сортировка товаров по убыванию отношения цена/вес  *vector*<Product\*> sortedProducts = products;  *sort*(sortedProducts.*begin*(), sortedProducts.*end*(), [](Product\* a, Product\* b) {  return (double)a->price / a->weight > (double)b->price / b->weight;  });  *vector*<Product\*> selected; // Выбранные товары  int totalWeight = 0; // Общий вес  int totalPrice = 0; // Общая стоимость  // Выбор товаров с максимальным отношением цена/вес  for (Product\* product : sortedProducts) {  if (totalWeight + product->weight <= maxWeight) {  selected.*push\_back*(product);  totalWeight += product->weight;  totalPrice += product->price;  }  }  // Вывод результатов оптимизации  *cout* << "\nОптимальный выбор для доставки (макс. вес " << maxWeight << "):\n";  for (Product\* product : selected) {  *cout* << "ID " << product->id << " " << product->name  << ", Price: " << product->price  << ", Weight: " << product->weight << "\n";  }  *cout* << "Общая стоимость: " << totalPrice << "\n";  }  };  int *main*() {  *setlocale*(*LC\_ALL*, "ru"); // Настройка локали для вывода русских символов  // Создание и инициализация базы данных  ProductDB db("products.txt");  // Пример 1: Поиск товаров в заданном диапазоне  db.findInRange(90, 160, 1, 3, "Electronics");  // Пример 2: Оптимизация доставки  db.optimizeDelivery(4);  return 0;  } |
| 3 | Реализуйте систему индексирования движущихся объектов с использованием красно-чёрного дерева. Каждый объект имеет поля: ID, Название, Траектория (временные метки с координатами: Timestamp, Latitude, Longitude). Постройте красно-чёрное дерево по ID объекта. Реализуйте метод, который прогнозирует будущую позицию объекта через заданное время (линейная интерполяция). Реализуйте метод, который находит потенциальные столкновения (объекты, которые окажутся ближе 1 км друг к другу через заданное время).  **Входные данные** Файл objects.txt:  ID ObjectName  401 ObjectA  402 ObjectB  Файл trajectories.txt:  ObjectID Timestamp Latitude Longitude  401 2025-04-12 10:00 52.0 13.0  401 2025-04-12 10:05 52.1 13.1  402 2025-04-12 10:00 52.2 13.2  402 2025-04-12 10:05 52.1 13.1  Прогнозируйте позицию объектов через 5 минут (на 2025-04-12 10:10). Найдите потенциальные столкновения.  **Ожидаемый результат**  Прогноз на 2025-04-12 10:10:  ObjectA: (52.2, 13.2)  ObjectB: (52.0, 13.0)  Потенциальные столкновения:  ObjectA и ObjectB (расстояние: 0 км)     |  | | --- | | #include <iostream>  #include <fstream>  #include <string>  #include <sstream>  #include <vector>  #include <cmath>  using namespace *std*;  // Структура для хранения точки траектории с временной меткой  struct Point {  *string* timestamp; // Временная метка (формат ГГГГ-ММ-ДД ЧЧ:ММ:СС)  double latitude; // Широта в градусах  double longitude; // Долгота в градусах  };  // Структура для объекта с траекторией движения  struct Object {  int id; // Уникальный идентификатор объекта  *string* name; // Название объекта  *vector*<Point> trajectory; // Вектор точек траектории  // Конструктор объекта  Object(int \_id, *string* \_name) {  id = \_id;  name = \_name;  }  };  // Цвета узлов для красно-черного дерева  enum Color { RED, BLACK };  // Узел красно-черного дерева  struct RBNode {  int key; // Ключ (id объекта)  Object\* data; // Указатель на данные объекта  RBNode\* left; // Левый потомок  RBNode\* right; // Правый потомок  RBNode\* parent; // Родительский узел  Color color; // Цвет узла  // Конструктор узла  RBNode(int k, Object\* d) {  key = k;  data = d;  left = nullptr;  right = nullptr;  parent = nullptr;  color = RED; // Новые узлы всегда красные  }  };  // Красно-черное дерево для хранения и быстрого поиска объектов  struct RBTree {  RBNode\* root; // Корень дерева  RBNode\* nil; // Специальный nil-узел (листья дерева)  // Конструктор дерева  RBTree() {  nil = new RBNode(0, nullptr);  nil->color = BLACK; // Nil-узлы всегда черные  nil->*left* = nil;  nil->*right* = nil;  root = nil;  }  // Левый поворот вокруг узла x  void leftRotate(RBNode\* x) {  RBNode\* y = x->*right*; // y становится правым потомком x  x->*right* = y->*left*; // Левый потомок y становится правым потомком x  if (y->*left* != nil)  y->*left*->parent = x;  y->parent = x->parent; // Перенос родителя  if (x->parent == nullptr)  root = y;  else if (x == x->parent->*left*)  x->parent->*left* = y;  else  x->parent->*right* = y;  y->*left* = x; // x становится левым потомком y  x->parent = y;  }  // Правый поворот вокруг узла y  void rightRotate(RBNode\* y) {  RBNode\* x = y->*left*; // x становится левым потомком y  y->*left* = x->*right*; // Правый потомок x становится левым потомком y  if (x->*right* != nil)  x->*right*->parent = y;  x->parent = y->parent; // Перенос родителя  if (y->parent == nullptr)  root = x;  else if (y == y->parent->*right*)  y->parent->*right* = x;  else  y->parent->*left* = x;  x->*right* = y; // y становится правым потомком x  y->parent = x;  }  // Восстановление свойств дерева после вставки  void fixInsert(RBNode\* z) {  // Пока родитель красный (нарушение свойства красно-черного дерева)  while (z->parent != nullptr && z->parent->color == RED) {  if (z->parent == z->parent->parent->*left*) {  RBNode\* y = z->parent->parent->*right*; // Дядя узла z  if (y->color == RED) {  // Случай 1: дядя красный  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->*right*) {  // Случай 2: дядя черный, z - правый потомок  z = z->parent;  leftRotate(z);  }  // Случай 3: дядя черный, z - левый потомок  z->parent->color = BLACK;  z->parent->parent->color = RED;  rightRotate(z->parent->parent);  }  }  else {  // Симметричные случаи для правого поддерева  RBNode\* y = z->parent->parent->*left*;  if (y->color == RED) {  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->*left*) {  z = z->parent;  rightRotate(z);  }  z->parent->color = BLACK;  z->parent->parent->color = RED;  leftRotate(z->parent->parent);  }  }  }  root->color = BLACK; // Корень всегда черный  }  // Вставка нового узла в дерево  void insert(int key, Object\* data) {  RBNode\* z = new RBNode(key, data);  z->*left* = nil;  z->*right* = nil;  RBNode\* y = nullptr;  RBNode\* x = root;  // Поиск места для вставки  while (x != nil) {  y = x;  if (z->key < x->key)  x = x->*left*;  else  x = x->*right*;  }  // Установка родителя  z->parent = y;  if (y == nullptr)  root = z;  else if (z->key < y->key)  y->*left* = z;  else  y->*right* = z;  // Балансировка дерева  fixInsert(z);  }  // Поиск объекта по id  Object\* search(int key) {  RBNode\* current = root;  while (current != nil) {  if (key == current->key)  return current->*data*;  else if (key < current->key)  current = current->*left*;  else  current = current->*right*;  }  return nullptr; // Объект не найден  }  };  // Класс для работы с базой данных объектов  struct ObjectDB {  RBTree index; // Красно-черное дерево для быстрого поиска  *vector*<Object\*> objects; // Все объекты  *string* objectsFile; // Файл с данными объектов  *string* trajectoriesFile; // Файл с траекториями  // Конструктор базы данных  ObjectDB(*string* oFile, *string* tFile) {  objectsFile = oFile;  trajectoriesFile = tFile;  loadObjects(); // Загрузка объектов при создании  loadTrajectories(); // Загрузка траекторий при создании  }  // Загрузка объектов из файла  void loadObjects() {  *ifstream* file(objectsFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << objectsFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  int id;  *string* name;  // Чтение данных об объекте из строки  ss >> id >> name;  // Создание нового объекта и добавление в базу  Object\* object = new Object(id, name);  objects.*push\_back*(object);  index.insert(id, object); // Вставка в индексное дерево  }  file.*close*();  }  // Загрузка траекторий из файла  void loadTrajectories() {  *ifstream* file(trajectoriesFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << trajectoriesFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  int objectID;  *string* timestamp;  double lat, lon;  ss >> objectID;  ss.*ignore*(1);  *getline*(ss, timestamp, ' ');  ss >> lat >> lon;  // Поиск объекта и добавление точки траектории  Object\* object = index.*search*(objectID);  if (object) {  Point p;  p.timestamp = timestamp;  p.latitude = lat;  p.longitude = lon;  object->trajectory.*push\_back*(p);  }  }  file.*close*();  }  // Прогнозирование позиции объекта через указанное количество минут  *pair*<double, double> predictPosition(Object\* object, int minutesAhead) {  // Необходимо минимум 2 точки для расчета скорости  if (object->trajectory.*size*() < 2)  return { 0, 0 };  // Берем две последние точки траектории  Point p1 = object->trajectory[object->trajectory.*size*() - 2];  Point p2 = object->trajectory[object->trajectory.*size*() - 1];  // Расчет скорости изменения координат (предполагаем интервал 5 минут между точками)  double speedLat = (p2.latitude - p1.latitude) / 5.0; // Скорость по широте (градусов/мин)  double speedLon = (p2.longitude - p1.longitude) / 5.0; // Скорость по долготе (градусов/мин)  // Расчет прогнозируемой позиции  double newLat = p2.latitude + speedLat \* minutesAhead;  double newLon = p2.longitude + speedLon \* minutesAhead;  return { newLat, newLon };  }  // Проверка потенциальных столкновений между объектами  void findCollisions(int minutesAhead) {  *vector*<*pair*<Object\*, *pair*<double, double>>> predictions;  // Прогнозирование позиций для всех объектов  for (Object\* object : objects) {  auto pos = predictPosition(object, minutesAhead);  predictions.*push\_back*({ object, pos });  }  // Вывод прогнозируемых позиций  *cout* << "Прогноз на 2025-04-12 10:10:\n";  for (auto& pred : predictions) {  *cout* << pred.*first*->name << ": (" << pred.*second*.*first* << ", " << pred.*second*.*second* << ")\n";  }  // Поиск потенциальных столкновений  *cout* << "\nПотенциальные столкновения:\n";  for (*size\_t* i = 0; i < predictions.*size*(); i++) {  for (*size\_t* j = i + 1; j < predictions.*size*(); j++) {  // Расчет расстояния между объектами (1° ≈ 111 км)  double dist = *sqrt*(*pow*(predictions[i].*second*.*first* - predictions[j].*second*.*first*, 2) +  *pow*(predictions[i].*second*.*second* - predictions[j].*second*.*second*, 2)) \* 111;  // Если расстояние меньше 1 км - потенциальное столкновение  if (dist < 1) {  *cout* << predictions[i].*first*->name << " и " << predictions[j].*first*->name  << " (расстояние: " << dist << " км)\n";  }  }  }  }  };  int *main*() {  *setlocale*(*LC\_ALL*, "ru"); // Настройка локали для вывода русских символов  // Создание и инициализация базы данных  ObjectDB db("objects.txt", "trajectories.txt");  // Поиск потенциальных столкновений через 5 минут  db.findCollisions(5);  return 0;  } | |
| 4 | Реализуйте систему индексирования транзакций с использованием красно-чёрного дерева. Каждая транзакция имеет поля: ID, Сумма, Статус (Active, Committed, Aborted), Время. Постройте красно-чёрное дерево по ID транзакции. Реализуйте систему MVCC (многоверсионное управление конкурентным доступом): каждая операция (вставка, обновление) создаёт новую версию записи. Реализуйте метод, который обеспечивает согласованность (проверяет, что сумма всех активных транзакций не превышает заданный лимит). Реализуйте метод отката до заданного времени.  **Входные данные** Файл transactions.txt:  ID Amount Status Timestamp  501 100 Committed 2025-04-12 10:00  502 200 Committed 2025-04-12 10:05  Добавьте транзакции: (503, 300, Active, 2025-04-12 10:10), (504, 400, Active, 2025-04-12 10:15). Проверьте согласованность (лимит 600). Откатите до 2025-04-12 10:05.  **Ожидаемый результат**  Сумма активных транзакций: 700  Нарушение лимита (600)!  После отката до 2025-04-12 10:05:  ID 501 Amount 100  ID 502 Amount 200     |  | | --- | | #include <iostream>  #include <fstream>  #include <string>  #include <sstream>  #include <vector>  using namespace *std*;  // Структура для хранения версии транзакции  struct Version {  int amount; // Сумма транзакции  *string* status; // Статус ("Active", "Cancelled" и т.д.)  *string* timestamp; // Временная метка (формат ГГГГ-ММ-ДД ЧЧ:ММ)  };  // Структура для транзакции с историей версий  struct Transaction {  int id; // Уникальный идентификатор транзакции  *vector*<Version> versions; // История изменений транзакции  // Конструктор транзакции (создает первую версию)  Transaction(int \_id, int \_amount, *string* \_status, *string* \_ts) {  id = \_id;  Version v;  v.amount = \_amount;  v.status = \_status;  v.timestamp = \_ts;  versions.*push\_back*(v); // Добавляем начальную версию  }  };  // Цвета узлов для красно-черного дерева  enum Color { RED, BLACK };  // Узел красно-черного дерева  struct RBNode {  int key; // Ключ (id транзакции)  Transaction\* data; // Указатель на данные транзакции  RBNode\* left; // Левый потомок  RBNode\* right; // Правый потомок  RBNode\* parent; // Родительский узел  Color color; // Цвет узла  // Конструктор узла  RBNode(int k, Transaction\* d) {  key = k;  data = d;  left = nullptr;  right = nullptr;  parent = nullptr;  color = RED; // Новые узлы всегда красные  }  };  // Красно-черное дерево для хранения транзакций  struct RBTree {  RBNode\* root; // Корень дерева  RBNode\* nil; // Специальный nil-узел (листья дерева)  // Конструктор дерева  RBTree() {  nil = new RBNode(0, nullptr);  nil->color = BLACK; // Nil-узлы всегда черные  nil->*left* = nil;  nil->*right* = nil;  root = nil;  }  // Левый поворот вокруг узла x  void leftRotate(RBNode\* x) {  RBNode\* y = x->*right*; // y становится правым потомком x  x->*right* = y->*left*; // Левый потомок y становится правым потомком x  if (y->*left* != nil)  y->*left*->parent = x;  y->parent = x->parent; // Перенос родителя  if (x->parent == nullptr)  root = y;  else if (x == x->parent->*left*)  x->parent->*left* = y;  else  x->parent->*right* = y;  y->*left* = x; // x становится левым потомком y  x->parent = y;  }  // Правый поворот вокруг узла y  void rightRotate(RBNode\* y) {  RBNode\* x = y->*left*; // x становится левым потомком y  y->*left* = x->*right*; // Правый потомок x становится левым потомком y  if (x->*right* != nil)  x->*right*->parent = y;  x->parent = y->parent; // Перенос родителя  if (y->parent == nullptr)  root = x;  else if (y == y->parent->*right*)  y->parent->*right* = x;  else  y->parent->*left* = x;  x->*right* = y; // y становится правым потомком x  y->parent = x;  }  // Восстановление свойств дерева после вставки  void fixInsert(RBNode\* z) {  // Пока родитель красный (нарушение свойства красно-черного дерева)  while (z->parent != nullptr && z->parent->color == RED) {  if (z->parent == z->parent->parent->*left*) {  RBNode\* y = z->parent->parent->*right*; // Дядя узла z  if (y->color == RED) {  // Случай 1: дядя красный  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->*right*) {  // Случай 2: дядя черный, z - правый потомок  z = z->parent;  leftRotate(z);  }  // Случай 3: дядя черный, z - левый потомок  z->parent->color = BLACK;  z->parent->parent->color = RED;  rightRotate(z->parent->parent);  }  }  else {  // Симметричные случаи для правого поддерева  RBNode\* y = z->parent->parent->*left*;  if (y->color == RED) {  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->*left*) {  z = z->parent;  rightRotate(z);  }  z->parent->color = BLACK;  z->parent->parent->color = RED;  leftRotate(z->parent->parent);  }  }  }  root->color = BLACK; // Корень всегда черный  }  // Вставка новой транзакции в дерево  void insert(int key, Transaction\* data) {  RBNode\* z = new RBNode(key, data);  z->*left* = nil;  z->*right* = nil;  RBNode\* y = nullptr;  RBNode\* x = root;  // Поиск места для вставки  while (x != nil) {  y = x;  if (z->key < x->key)  x = x->*left*;  else  x = x->*right*;  }  // Установка родителя  z->parent = y;  if (y == nullptr)  root = z;  else if (z->key < y->key)  y->*left* = z;  else  y->*right* = z;  // Балансировка дерева  fixInsert(z);  }  // Рекурсивный вывод транзакций на момент указанного времени  void printInOrder(RBNode\* node, *string* maxTimestamp) {  if (node == nil) return;  printInOrder(node->*left*, maxTimestamp);  Transaction\* trans = node->*data*;  // Ищем последнюю версию на момент maxTimestamp  for (int i = trans->versions.*size*() - 1; i >= 0; i--) {  if (trans->versions[i].timestamp <= maxTimestamp) {  *cout* << "ID " << trans->id << " Amount " << trans->versions[i].amount << "\n";  break;  }  }  printInOrder(node->*right*, maxTimestamp);  }  // Рекурсивный подсчет суммы активных транзакций  int sumActive(RBNode\* node) {  if (node == nil) return 0;  int sum = sumActive(node->*left*);  Transaction\* trans = node->*data*;  // Проверяем последнюю версию транзакции  if (trans->versions.*back*().status == "Active") {  sum += trans->versions.*back*().amount;  }  sum += sumActive(node->*right*);  return sum;  }  };  // Класс для работы с базой данных транзакций  struct TransactionDB {  RBTree index; // Красно-черное дерево для индексации  *vector*<Transaction\*> transactions; // Все транзакции  *string* inputFile; // Имя входного файла  // Конструктор базы данных  TransactionDB(*string* file) {  inputFile = file;  loadFromFile(); // Загрузка данных при создании  }  // Загрузка транзакций из файла  void loadFromFile() {  *ifstream* file(inputFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << inputFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  int id, amount;  *string* status, timestamp;  ss >> id >> amount >> status;  ss.*ignore*(1);  *getline*(ss, timestamp);  // Создание новой транзакции и добавление в базу  Transaction\* trans = new Transaction(id, amount, status, timestamp);  transactions.*push\_back*(trans);  index.insert(id, trans); // Вставка в индексное дерево  }  file.*close*();  }  // Добавление новой транзакции  void insert(int id, int amount, *string* status, *string* timestamp) {  Transaction\* trans = new Transaction(id, amount, status, timestamp);  transactions.*push\_back*(trans);  index.insert(id, trans);  }  // Проверка соблюдения лимита активных транзакций  void checkConsistency(int limit) {  int total = index.sumActive(index.root);  *cout* << "Сумма активных транзакций: " << total << "\n";  if (total > limit) {  *cout* << "Нарушение лимита (" << limit << ")!\n";  }  else {  *cout* << "Лимит соблюдён.\n";  }  }  // Откат системы до указанного времени  void rollbackTo(*string* timestamp) {  *cout* << "После отката до " << timestamp << ":\n";  index.printInOrder(index.root, timestamp);  }  };  int *main*() {  *setlocale*(*LC\_ALL*, "ru"); // Настройка локали для вывода русских символов  // Создание и инициализация базы данных  TransactionDB db("transactions.txt");  // Добавление тестовых транзакций  db.insert(503, 300, "Active", "2025-04-12 10:10");  db.insert(504, 400, "Active", "2025-04-12 10:15");  // Проверка лимита (макс. 600)  db.checkConsistency(600);  // Откат системы до указанного времени  db.rollbackTo("2025-04-12 10:05");  return 0;  } | |
| 5 | Реализуйте систему индексирования слов из текстового корпуса с использованием красно-чёрного дерева. Каждое слово имеет поля: само слово (уникальный ключ), частота встречаемости, список документов (ID документа, где слово встречается), список связанных слов (слова, которые встречаются в одном предложении с данным словом). Постройте красно-чёрное дерево по строковому ключу (слову). Реализуйте следующие функции:   1. **Анализ связей слов**: Найдите компоненты связности в графе слов, где рёбра соединяют слова, которые встречаются в одном предложении. 2. **Рекомендация слов**: Для заданного слова найдите слова, которые часто встречаются с ним в одном предложении, и отсортируйте их по частоте совместного появления. 3. **Анализ редких слов**: Найдите слова, которые встречаются реже заданного порога (например, частота < 3), но имеют высокую степень связности (много связанных слов). 4. **Оптимизация корпуса**: Удалите из корпуса слова, которые встречаются только в одном документе и имеют низкую частоту (например, < 2), и обновите связи между оставшимися словами.   **Входные данные** Файл words.txt:  Word Frequency DocIDs  apple 5 1,2,3  banana 3 1,2  cherry 2 1  date 1 3  Файл cooccurrences.txt:  Word1 Word2  apple banana  apple cherry  banana cherry  **Действия**:   1. Найдите компоненты связности в графе слов. 2. Для слова "apple" найдите рекомендации (слова, с которыми оно часто встречается). 3. Найдите редкие слова с высокой степенью связности (порог частоты: 3). 4. Оптимизируйте корпус, удалив слова с частотой < 2 и встречающиеся только в одном документе, и обновите связи.   **Ожидаемый результат**  Компоненты связности:  Компонента 1: apple, banana, cherry  Компонента 2: date  Рекомендации для слова "apple":  banana (совместно: 1), cherry (совместно: 1)  Редкие слова с высокой степенью связности (порог 3):  cherry (частота: 2, связанных слов: 2)  После оптимизации корпуса:  Удалено слово: date  Оставшиеся слова: apple, banana, cherry  Обновлённые связи:  apple: banana, cherry  banana: apple, cherry  cherry: apple, banana     |  | | --- | | #include <iostream>  #include <fstream>  #include <string>  #include <sstream>  #include <vector>  #include <map>  #include <set>  using namespace *std*;  // Структура для хранения информации о слове  struct Word {  *string* word; // Само слово  int frequency; // Частота встречаемости  *vector*<int> docIDs; // ID документов, где встречается слово  *vector*<*string*> relatedWords; // Связанные слова (встречаются в одном контексте)  *map*<*string*, int> cooccurrenceCount; // Частота совместного появления с другими словами  // Конструктор слова  Word(*string* \_word, int \_freq) {  word = \_word;  frequency = \_freq;  }  };  // Цвета узлов для красно-черного дерева  enum Color { RED, BLACK };  // Узел красно-черного дерева  struct RBNode {  *string* key; // Ключ (слово)  Word\* data; // Указатель на данные слова  RBNode\* left; // Левый потомок  RBNode\* right; // Правый потомок  RBNode\* parent; // Родительский узел  Color color; // Цвет узла  // Конструктор узла  RBNode(*string* k, Word\* d) {  key = k;  data = d;  left = nullptr;  right = nullptr;  parent = nullptr;  color = RED; // Новые узлы всегда красные  }  };  // Красно-черное дерево для хранения слов  struct RBTree {  RBNode\* root; // Корень дерева  RBNode\* nil; // Специальный nil-узел (листья дерева)  // Конструктор дерева  RBTree() {  nil = new RBNode("", nullptr);  nil->color = BLACK; // Nil-узлы всегда черные  nil->*left* = nil;  nil->*right* = nil;  root = nil;  }  // Деструктор дерева (рекурсивное удаление узлов)  ~RBTree() {  deleteTree(root);  delete nil;  }  // Рекурсивное удаление узлов дерева  void deleteTree(RBNode\* node) {  if (node == nil) return;  deleteTree(node->*left*);  deleteTree(node->*right*);  delete node->*data*; // Удаляем данные слова  delete node; // Удаляем сам узел  }  // Левый поворот вокруг узла x  void leftRotate(RBNode\* x) {  RBNode\* y = x->right; // y становится правым потомком x  x->right = y->left; // Левый потомок y становится правым потомком x  if (y->left != nil)  y->left->parent = x;  y->parent = x->parent; // Перенос родителя  if (x->parent == nullptr)  root = y;  else if (x == x->parent->left)  x->parent->left = y;  else  x->parent->right = y;  y->left = x; // x становится левым потомком y  x->parent = y;  }  // Правый поворот вокруг узла y  void rightRotate(RBNode\* y) {  RBNode\* x = y->left; // x становится левым потомком y  y->left = x->right; // Правый потомок x становится левым потомком y  if (x->right != nil)  x->right->parent = y;  x->parent = y->parent; // Перенос родителя  if (y->parent == nullptr)  root = x;  else if (y == y->parent->right)  y->parent->right = x;  else  y->parent->left = x;  x->right = y; // y становится правым потомком x  y->parent = x;  }  // Восстановление свойств дерева после вставки  void fixInsert(RBNode\* z) {  // Пока родитель красный (нарушение свойства красно-черного дерева)  while (z->parent != nullptr && z->parent->color == RED) {  if (z->parent == z->parent->parent->left) {  RBNode\* y = z->parent->parent->right; // Дядя узла z  if (y->color == RED) {  // Случай 1: дядя красный  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->right) {  // Случай 2: дядя черный, z - правый потомок  z = z->parent;  leftRotate(z);  }  // Случай 3: дядя черный, z - левый потомок  z->parent->color = BLACK;  z->parent->parent->color = RED;  rightRotate(z->parent->parent);  }  }  else {  // Симметричные случаи для правого поддерева  RBNode\* y = z->parent->parent->left;  if (y->color == RED) {  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->left) {  z = z->parent;  rightRotate(z);  }  z->parent->color = BLACK;  z->parent->parent->color = RED;  leftRotate(z->parent->parent);  }  }  }  root->color = BLACK; // Корень всегда черный  }  // Вставка нового слова в дерево  void insert(*string* key, Word\* data) {  RBNode\* z = new RBNode(key, data);  z->left = nil;  z->right = nil;  RBNode\* y = nullptr;  RBNode\* x = root;  // Поиск места для вставки  while (x != nil) {  y = x;  if (z->key < x->key)  x = x->left;  else  x = x->right;  }  // Установка родителя  z->parent = y;  if (y == nullptr)  root = z;  else if (z->key < y->key)  y->left = z;  else  y->right = z;  // Балансировка дерева  fixInsert(z);  }  // Поиск слова в дереве  Word\* search(*string* key) {  RBNode\* current = root;  while (current != nil) {  if (key == current->key)  return current->data;  else if (key < current->key)  current = current->left;  else  current = current->right;  }  return nullptr; // Слово не найдено  }  // Удаление слова из дерева  void remove(*string* key) {  RBNode\* z = root;  while (z != nil) {  if (key == z->key) break;  else if (key < z->key) z = z->left;  else z = z->right;  }  if (z == nil) return;  RBNode\* y = z;  Color yOriginalColor = y->color;  RBNode\* x;  if (z->left == nil) {  x = z->right;  transplant(z, z->right);  }  else if (z->right == nil) {  x = z->left;  transplant(z, z->left);  }  else {  y = minimum(z->right);  yOriginalColor = y->color;  x = y->right;  if (y->parent == z) {  x->parent = y;  }  else {  transplant(y, y->right);  y->right = z->right;  y->right->parent = y;  }  transplant(z, y);  y->left = z->left;  y->left->parent = y;  y->color = z->color;  }  if (yOriginalColor == BLACK) {  fixDelete(x);  }  delete z->data; // Удаляем данные слова  delete z; // Удаляем узел  }  // Замена поддерева  void transplant(RBNode\* u, RBNode\* v) {  if (u->parent == nullptr)  root = v;  else if (u == u->parent->left)  u->parent->left = v;  else  u->parent->right = v;  v->parent = u->parent;  }  // Поиск минимального узла в поддереве  RBNode\* minimum(RBNode\* node) {  while (node->*left* != nil) {  node = node->*left*;  }  return node;  }  // Восстановление свойств дерева после удаления  void fixDelete(RBNode\* x) {  while (x != root && x->color == BLACK) {  if (x == x->parent->left) {  RBNode\* w = x->parent->right;  if (w->color == RED) {  w->color = BLACK;  x->parent->color = RED;  leftRotate(x->parent);  w = x->parent->right;  }  if (w->left->color == BLACK && w->right->color == BLACK) {  w->color = RED;  x = x->parent;  }  else {  if (w->right->color == BLACK) {  w->left->color = BLACK;  w->color = RED;  rightRotate(w);  w = x->parent->right;  }  w->color = x->parent->color;  x->parent->color = BLACK;  w->right->color = BLACK;  leftRotate(x->parent);  x = root;  }  }  else {  RBNode\* w = x->parent->left;  if (w->color == RED) {  w->color = BLACK;  x->parent->color = RED;  rightRotate(x->parent);  w = x->parent->left;  }  if (w->right->color == BLACK && w->left->color == BLACK) {  w->color = RED;  x = x->parent;  }  else {  if (w->left->color == BLACK) {  w->right->color = BLACK;  w->color = RED;  leftRotate(w);  w = x->parent->left;  }  w->color = x->parent->color;  x->parent->color = BLACK;  w->left->color = BLACK;  rightRotate(x->parent);  x = root;  }  }  }  x->color = BLACK;  }  // Обход дерева в порядке возрастания (для получения всех слов)  void inorderTraversal(RBNode\* node, *vector*<*string*>& words) {  if (node == nil) return;  inorderTraversal(node->*left*, words);  words.*push\_back*(node->key);  inorderTraversal(node->*right*, words);  }  // Получение списка всех слов в дереве  *vector*<*string*> getAllWords() {  *vector*<*string*> words;  inorderTraversal(root, words);  return words;  }  };  // Класс для анализа текстовых данных  struct TextAnalyzer {  RBTree index; // Красно-черное дерево для индексации слов  *vector*<Word\*> words; // Все слова  *string* wordsFile; // Файл со словами  *string* cooccurrencesFile; // Файл с совместными появлениями  // Конструктор анализатора  TextAnalyzer(*string* wFile, *string* cFile) {  wordsFile = wFile;  cooccurrencesFile = cFile;  loadWords(); // Загрузка слов при создании  loadCooccurrences(); // Загрузка совместных появлений  }  // Деструктор (освобождение памяти)  ~TextAnalyzer() {  for (Word\* word : words) {  delete word;  }  }  // Загрузка слов из файла  void loadWords() {  *ifstream* file(wordsFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << wordsFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  *string* word;  int freq;  *string* docIDsStr;  if (!(ss >> word >> freq >> docIDsStr)) {  *cout* << "Ошибка: Некорректный формат строки в файле " << wordsFile << ": " << line << "\n";  continue;  }  // Создание нового слова и добавление в базу  Word\* w = new Word(word, freq);  *stringstream* docSS(docIDsStr);  *string* docID;  while (*getline*(docSS, docID, ',')) {  w->docIDs.*push\_back*(*stoi*(docID));  }  words.*push\_back*(w);  index.insert(word, w); // Вставка в индексное дерево  }  file.*close*();  }  // Загрузка информации о совместных появлениях слов  void loadCooccurrences() {  *ifstream* file(cooccurrencesFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << cooccurrencesFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  *string* word1, word2;  if (!(ss >> word1 >> word2)) {  *cout* << "Ошибка: Некорректный формат строки в файле " << cooccurrencesFile << ": " << line << "\n";  continue;  }  // Поиск слов в индексе  Word\* w1 = index.search(word1);  Word\* w2 = index.search(word2);  if (w1 && w2) {  // Добавление информации о совместном появлении  w1->relatedWords.*push\_back*(word2);  w1->cooccurrenceCount[word2]++;  w2->relatedWords.*push\_back*(word1);  w2->cooccurrenceCount[word1]++;  }  else {  *cout* << "Ошибка: Слово " << word1 << " или " << word2 << " не найдено!\n";  }  }  file.*close*();  }  // Поиск компонент связности графа слов (алгоритм DFS)  void findConnectedComponents() {  if (words.*empty*()) {  *cout* << "Компоненты связности:\nНет слов.\n";  return;  }  *set*<*string*> visited; // Множество посещенных слов  *vector*<*vector*<*string*>> components; // Найденные компоненты связности  for (Word\* word : words) {  if (!visited.*count*(word->word)) {  *vector*<*string*> component;  dfs(word->word, visited, component); // Обход в глубину  components.*push\_back*(component);  }  }  // Вывод результатов  *cout* << "Компоненты связности:\n";  for (*size\_t* i = 0; i < components.*size*(); i++) {  *cout* << "Компонента " << (i + 1) << ": ";  for (*size\_t* j = 0; j < components[i].*size*(); j++) {  *cout* << components[i][j];  if (j < components[i].*size*() - 1) *cout* << ", ";  }  *cout* << "\n";  }  }  // Обход в глубину для поиска компонент связности  void dfs(*string* word, *set*<*string*>& visited, *vector*<*string*>& component) {  Word\* w = index.search(word);  if (!w) return;  visited.*insert*(word);  component.*push\_back*(word);  // Рекурсивный обход всех связанных слов  for (const *string*& related : w->relatedWords) {  if (!visited.*count*(related)) {  dfs(related, visited, component);  }  }  }  // Рекомендация слов на основе совместных появлений  void recommendWords(*string* targetWord) {  Word\* w = index.search(targetWord);  if (!w) {  *cout* << "Слово " << targetWord << " не найдено.\n";  return;  }  *cout* << "Рекомендации для слова \"" << targetWord << "\":\n";  *vector*<pair<*string*, int>> recommendations;  // Сбор рекомендаций  for (const auto& pair : w->cooccurrenceCount) {  recommendations.*push\_back*(pair);  }  // Сортировка по частоте совместного появления (по убыванию)  for (*size\_t* i = 0; i < recommendations.*size*(); i++) {  for (*size\_t* j = i + 1; j < recommendations.*size*(); j++) {  if (recommendations[i].*second* < recommendations[j].*second*) {  *swap*(recommendations[i], recommendations[j]);  }  }  }  // Вывод результатов  if (recommendations.*empty*()) {  *cout* << "Нет рекомендаций.\n";  }  else {  for (const auto& rec : recommendations) {  *cout* << rec.*first* << " (совместно: " << rec.*second* << ")";  if (&rec != &recommendations.*back*()) *cout* << ", ";  }  *cout* << "\n";  }  }  // Поиск редких слов с высокой степенью связности  void findRareConnectedWords(int freqThreshold) {  *cout* << "Редкие слова с высокой степенью связности (порог " << freqThreshold << "):\n";  bool found = false;  for (Word\* w : words) {  if (w->frequency < freqThreshold && w->relatedWords.*size*() >= 2) {  *cout* << w->word << " (частота: " << w->frequency << ", связанных слов: " << w->relatedWords.*size*() << ")\n";  found = true;  }  }  if (!found) *cout* << "Нет подходящих слов.\n";  }  // Оптимизация корпуса (удаление редких и малоиспользуемых слов)  void optimizeCorpus(int freqThreshold) {  *vector*<*string*> wordsToRemove;  // Поиск слов для удаления  for (Word\* w : words) {  if (w->frequency < freqThreshold && w->docIDs.*size*() <= 1) {  wordsToRemove.*push\_back*(w->word);  }  }  // Удаление слов из индекса  for (const *string*& word : wordsToRemove) {  *cout* << "Удалено слово: " << word << "\n";  index.remove(word);  }  // Обновление списка слов  *vector*<Word\*> newWords;  for (Word\* w : words) {  if (*find*(wordsToRemove.*begin*(), wordsToRemove.*end*(), w->word) == wordsToRemove.*end*()) {  newWords.*push\_back*(w);  }  else {  delete w; // Освобождение памяти  }  }  words = newWords;  // Обновление связей между словами  for (Word\* w : words) {  *vector*<*string*> newRelatedWords;  *map*<*string*, int> newCooccurrenceCount;  // Фильтрация связанных слов  for (const *string*& related : w->relatedWords) {  if (index.search(related)) {  newRelatedWords.*push\_back*(related);  newCooccurrenceCount[related] = w->cooccurrenceCount[related];  }  }  w->relatedWords = newRelatedWords;  w->cooccurrenceCount = newCooccurrenceCount;  }  // Вывод результатов оптимизации  *cout* << "Оставшиеся слова: ";  for (*size\_t* i = 0; i < words.*size*(); i++) {  *cout* << words[i]->word;  if (i < words.*size*() - 1) *cout* << ", ";  }  *cout* << "\nОбновлённые связи:\n";  for (Word\* w : words) {  *cout* << w->word << ": ";  for (*size\_t* i = 0; i < w->relatedWords.*size*(); i++) {  *cout* << w->relatedWords[i];  if (i < w->relatedWords.*size*() - 1) *cout* << ", ";  }  *cout* << "\n";  }  }  };  int *main*() {  *setlocale*(*LC\_ALL*, "ru"); // Настройка локали для русских символов  // Создание и инициализация анализатора  TextAnalyzer analyzer("words.txt", "cooccurrences.txt");  // Анализ компонент связности  analyzer.findConnectedComponents();  // Получение рекомендаций для слова "apple"  analyzer.recommendWords("apple");  // Поиск редких, но связанных слов  analyzer.findRareConnectedWords(3);  // Оптимизация корпуса (удаление редких слов)  analyzer.optimizeCorpus(2);  return 0;  } | |

1. **Ответы к контрольным вопросам**

**1. Что такое красно-чёрное дерево и почему оно называется именно так?**

**Ответ:**  
Красно-чёрное дерево — это сбалансированное бинарное дерево поиска, в котором каждый узел имеет цвет (красный или чёрный). Название происходит от этой особенности: цвета узлов используются для поддержания баланса, чтобы дерево не вырождалось в список, обеспечивая сложность операций O(log n).

**2. Перечислите основные правила красно-чёрного дерева.**

**Ответ:**

1. Каждый узел — либо красный, либо чёрный.
2. Корень всегда чёрный.
3. Все листья (NULL-узлы) — чёрные.
4. У красного узла не может быть красных детей (нет двух красных узлов подряд).
5. Чёрная высота (количество чёрных узлов на пути от узла до листа) одинакова для всех путей.

**3. Почему красно-чёрное дерево подходит для индексирования в базах данных?**

**Ответ:**  
Красно-чёрное дерево подходит для индексирования, потому что:

* Обеспечивает логарифмическую сложность операций (поиск, вставка, удаление — O(log n)).
* Автоматически балансируется, предотвращая вырождение в список.
* Эффективно обновляется при добавлении или удалении данных.
* Поддерживает сравнение ключей любого типа (например, ID, строки).

**4. Какой цвет имеет новый узел при вставке в красно-чёрное дерево и почему?**

**Ответ:**  
Новый узел при вставке всегда красный. Это связано с тем, что добавление красного узла меньше нарушает правила красно-чёрного дерева (например, чёрную высоту), чем добавление чёрного, и требует меньше операций для восстановления баланса.

**5. Опишите случай балансировки при вставке, когда у нового узла красный родитель и красный дядя.**

**Ответ:**  
Если у нового узла красный родитель и красный дядя:

* Родитель и дядя перекрашиваются в чёрный.
* Дед становится красным.
* Проверяется дед: если его родитель красный, процесс балансировки продолжается выше по дереву.  
  Этот случай встречается с вероятностью около 25%.

**6. Какой случай балансировки при удалении возникает, если брат удаляемого узла красный?**

**Ответ:**  
Если брат удаляемого узла красный:

* Брат становится чёрным.
* Родитель становится красным.
* Выполняется левый поворот вокруг родителя.
* После этого случай сводится к другим сценариям (например, брат становится чёрным).  
  Этот случай встречается с вероятностью около 50% при удалении.

**7. Что такое чёрная высота и почему она важна для красно-чёрного дерева?**

**Ответ:**  
Чёрная высота — это количество чёрных узлов на пути от данного узла до листа (NULL), не включая сам узел, но включая NULL. Она важна, потому что правило одинаковой чёрной высоты на всех путях гарантирует сбалансированность дерева, обеспечивая, что самый длинный путь не более чем в 2 раза длиннее самого короткого.

**8. Как красно-чёрное дерево обеспечивает логарифмическую сложность операций?**

**Ответ:**  
Красно-чёрное дерево поддерживает баланс благодаря своим правилам (например, нет двух красных узлов подряд, одинаковая чёрная высота). Это гарантирует, что высота дерева не превышает 2 \* log(n + 1), где n — число узлов, что обеспечивает сложность операций (поиск, вставка, удаление) O(log n).

**9. В чём преимущество красно-чёрного дерева перед обычным бинарным деревом поиска (BST) при использовании в базах данных?**

**Ответ:**  
Преимущество красно-чёрного дерева перед BST:

* Оно автоматически балансируется, не вырождаясь в список, в отличие от BST, где в худшем случае операции могут иметь сложность O(n).
* Гарантирует O(log n) для операций, что критично для баз данных с большим количеством записей.

**10. Какие ограничения красно-чёрного дерева для индексирования в базах данных?**

**Ответ:**  
Ограничения красно-чёрного дерева:

* Сложность реализации: алгоритмы вставки и удаления сложнее из-за балансировки.
* Менее эффективно для дисковых операций по сравнению с B-деревьями, так как хранит только один ключ в узле.
* Диапазонные запросы менее оптимизированы, чем в B+-деревьях, где ключи хранятся в листьях.

**11. Какой метод используется в программе для поиска студента по ID? Опишите его работу.**

**Ответ:**  
Метод search в структуре RBTree используется для поиска студента по ID:

* Начинается с корня дерева.
* Сравнивает ключ (ID) с текущим узлом: если ключ равен, возвращает данные узла; если меньше — идёт влево, если больше — вправо.
* Повторяет, пока не найдёт узел или не дойдёт до листа (nil).  
  Сложность — O(log n) благодаря сбалансированности дерева.

**12. Как в программе реализовано удаление студента из базы данных?**

**Ответ:**  
Удаление студента в программе:

1. Метод deleteStudent вызывает index.remove(id) для удаления узла из красно-чёрного дерева.
2. В дереве:
   * Находит узел с заданным ID.
   * Если у узла один ребёнок или нет детей, заменяет его на ребёнка.
   * Если два ребёнка, находит преемника (минимальный узел в правом поддереве), заменяет удаляемый узел на преемника и корректирует дерево.
   * Выполняет балансировку (fixDelete), если удалённый узел был чёрным.
3. Удаляет студента из вектора students и переписывает файл.

**13. Почему в программе используется вектор students в дополнение к красно-чёрному дереву?**

**Ответ:**  
Вектор students используется для:

* Хранения всех записей о студентах в порядке загрузки из файла.
* Упрощения операций, таких как переписывание файла после удаления или вставки.
* Обеспечения доступа к данным без необходимости обхода дерева, если порядок важен.  
  Красно-чёрное дерево же используется для быстрого поиска по ID.

**14. Как красно-чёрное дерево может быть использовано для диапазонных запросов?**

**Ответ:**  
Для диапазонных запросов (например, найти студентов с ID от 1531 до 1535):

* Выполняется обход дерева (например, инфиксный).
* Проверяется каждый узел: если его ключ попадает в диапазон, он включается в результат.
* Поддеревья, которые заведомо не содержат нужных ключей (например, левое поддерево, если ключ меньше минимального значения диапазона), пропускаются.  
  Это менее эффективно, чем в B+-деревьях, но всё же возможно.

**15. Сравните красно-чёрное дерево с B-деревом для задач индексирования.**

**Ответ:**

* **Красно-чёрное дерево:**
  + Плюсы: проще реализация, эффективно для операций в памяти, O(log n) сложность.
  + Минусы: не оптимизировано для дисковых операций (много узлов — много обращений к диску).
* **B-дерево:**
  + Плюсы: оптимизировано для дисков (хранит больше ключей в узле, меньше обращений), лучше для диапазонных запросов.
  + Минусы: сложнее реализация, больше памяти на узел.  
    Красно-чёрное дерево лучше для задач в оперативной памяти, B-дерево — для больших баз данных на диске.