rr

**МIНIСТЕРСТВО ОСВIТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**

**ІМЕНІ ІГОРЯ СІКОРСЬКОГО”**

**Факультет прикладної математики**

**Кафедра програмного забезпечення комп’ютерних систем**

**Лабораторна робота № 8**

**з дисципліни “ Основи програмування”**

**тема “МАТЕМАТИЧНІ ВИРАЗИ. ОБЧИСЛЕННЯ МАТЕМАТИЧНИХ ФОРМУЛ”**

|  |  |  |
| --- | --- | --- |
| **Виконав**  **студент I курсу**  **групи КП-83**  **Мортіков Владислав Євгенович**  **(прізвище, ім’я, по батькові)**  **варіант № 16** |  | **Перевірив**  **“\_\_\_\_” “\_\_\_\_\_\_\_\_\_\_\_\_” 20\_\_\_ р.**  **викладач**  **Гадиняк Руслан Анатолійович**  **(прізвище, ім’я, по батькові)** |

**Київ 2019**

**Мета роботи**

Навчитись працювати з реляційними базами даних.  
Реалізувати модуль для доступу до SQLite бази даних.

### **Загальні вимоги до завдання**

Визначення та позначки:

1. **SQLite клієнт -** програма для керування SQLite базами даних.

Кроки виконання:

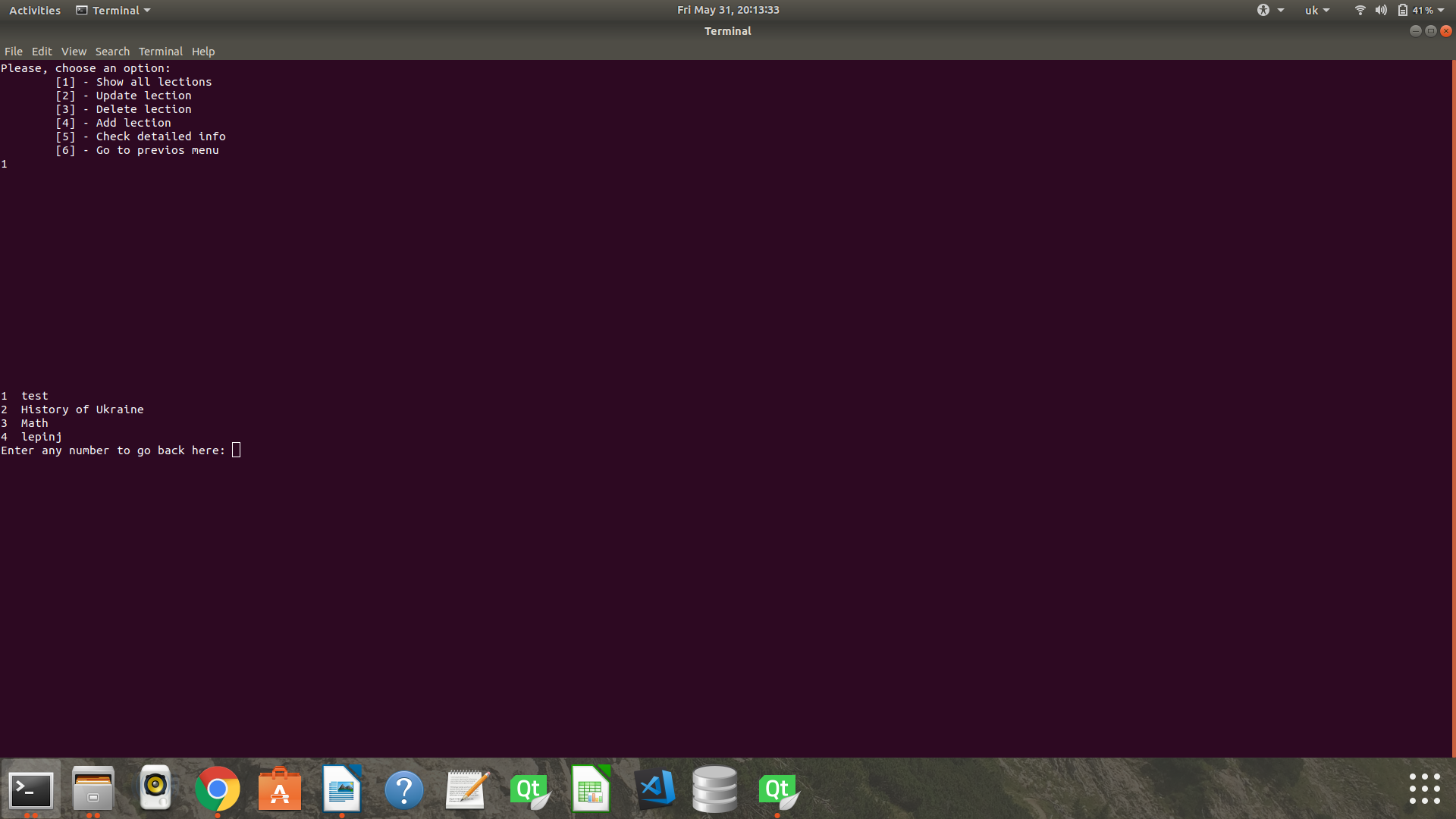
1. Скопіювати у проект даного завдання ваше CUI рішення [лабораторної роботи №6](https://progbase.herokuapp.com/modules/progbase2/tasks/lab6).
2. Встановити будь-який графічний (або консольний) SQLite клієнт.
3. Сховище даних SQLite:
   1. Створити за допомогою SQLite клієнта файл бази даних data.sqlite у директорії data/sqlite(**UPD-1**):
      1. Створити у базі даних дві таблиці, що відповідатимуть за основний та додатковий тип даних. Назвати таблиці аналогічно назвам файлів з даними (наприклад, students, courses).  
         (**UPD-2**) Для цього можна також імпортувати CSV файли з даними і відредагувати типи і обмеження атрибутів створених таблиць.
      2. Кожна таблиця має мати відповідні своєму типу даних типізовані стовпці.
      3. Назви таблиць і їх стовпців мають задовольняти правила іменування змінних мови С (і бути в [snake case](https://en.wikipedia.org/wiki/Snake_case)).
      4. Кожна таблиця обов'язково має мати стовпець id, як автоінкрементний цілочисельний ключ.
      5. Заповнити кожну таблицю відповідними записами з даними (можна взяти з CSV файлів).
   2. Створити модуль sqlite\_storage з класом SqliteStorage, що публічно наслідується від абстрактного класу Storage.
   3. Реалізувати SqliteStorage з використанням [Qt SQL класів](https://doc.qt.io/qt-5/qtsql-cachedtable-example.html):
      1. Додати у файл налаштувань проекту (.pro) QT += sql як перший рядок .
      2. Підключити у модулі sqlite\_storage заголовочний файл #include <QtSql>
      3. (**UPD-2**) Ідентифікатори нових записів (для автоінкрементних атрибутів таблиць) генеруються базою даних. При додаванні нового запису в БД ідентифікатор не задається. Його можна отримати після виконання цього запиту через метод lastInsertId() (див. **Додаток А**).
      4. (**UPD-2**) Особливістю даного сховища буде те, що всі зміни сутностей в його методах одразу змінюватимуть базу даних.
   4. При створенні об'єкта сховища замінити його тип на SqliteStorage, а шлях до файлу на ../lab8/data/sqlite (**UPD-1**).

**Тексти коду програм**

|  |
| --- |
| **sqlite\_storage.hpp** |
| #ifndef SQLITE\_STORAGE\_HPP  #define SQLITE\_STORAGE\_HPP  #include "storage.hpp"  #include <QtSql>  #include <QSqlDatabase>  class SQlitestorage : public Storage  {  const string dir\_name\_;  QSqlDatabase db\_;  public:  SQlitestorage(const string & dir\_name) : dir\_name\_(dir\_name)  {  db\_ = QSqlDatabase::addDatabase("QSQLITE");  }  bool ***open***();  bool ***close***();  // lections  vector<Lection> ***getAllLections***(void);  optional<Lection> ***getLectionById***(int student\_id);  bool ***updateLection***(const Lection &lection);  bool ***removeLection***(int lection\_id);  int ***insertLection***(const Lection &lection);  // courses  vector<Topic> ***getAllTopics***(void);  optional<Topic> ***getTopicById***(int topic\_id);  bool ***updateTopic***(const Topic &topic);  bool ***removeTopic***(int topic\_id);  int ***insertTopic***(const Topic &topic);  };  #endif // SQLITE\_STORAGE\_HPP |

|  |
| --- |
| **sqlite\_storage.cpp** |
| #include "sqlite\_storage.hpp"  #include <QtSql>  #include <iostream>  #include <QDebug>  //SQlitestorage::SqliteStorage(const string &dir\_name): dir\_name\_(dir\_name)  //{  // db\_ = QSqlDatabase::addDatabase("QSQLITE");  //}  bool SQlitestorage::***open***()  {  QString path = QString::fromStdString(this->dir\_name\_) + "/data.sqlite";  db\_.setDatabaseName(path);  bool connected = db\_.open();  if (!connected) {return false;}  return true;  }  bool SQlitestorage::***close***()  {  db\_.close();  return true;  }  vector<Lection> SQlitestorage::***getAllLections***(void)  {  vector<Lection> lections;  QSqlQuery query("SELECT \* FROM lections");  if (!query.exec()) {  qDebug() << "getAlllections error:" << query.lastError().text();  }  while (query.next())  {  Lection newLection;  newLection.id = query.value("id").toInt();  newLection.title = query.value("title").toString().toStdString();  newLection.lectorer = query.value("lectorer").toString().toStdString();  newLection.students = query.value("students").toInt();  lections.push\_back(newLection);  }  return lections;  }  optional<Lection> SQlitestorage::***getLectionById***(int lection\_id)  {  QSqlQuery query;  query.prepare("SELECT \* FROM lections WHERE id = :id");  query.bindValue(":id", lection\_id);  if (!query.exec())  {  qDebug() << "getLections error:" << query.lastError();  return nullopt;  }  if (query.next())  {  Lection newLection;  newLection.id = query.value("id").toInt();  newLection.title = query.value("title").toString().toStdString();  newLection.lectorer = query.value("lectorer").toString().toStdString();  newLection.students = query.value("students").toInt();  return newLection;  }  else  {  return nullopt;  }  }  bool SQlitestorage::***updateLection***(const Lection &lection)  {  QSqlQuery query;  query.prepare("UPDATE lections SET title = :title, lectorer = :lectorer, students = :students WHERE id = :id");  query.bindValue(":title", QString::fromStdString(lection.title));  query.bindValue(":lectorer", QString::fromStdString(lection.lectorer));  query.bindValue(":students", lection.students);  query.bindValue(":id", lection.id);  if (!query.exec()){  qDebug() << "updateLection error:" << query.lastError();  return false;  }  return true;  }  bool SQlitestorage::***removeLection***(int lection\_id)  {  QSqlQuery query;  query.prepare("DELETE FROM lection WHERE id = :id");  query.bindValue(":id", lection\_id);  if (!query.exec()){  qDebug() << "removeLection error:" << query.lastError();  return false;  }  if(query.numRowsAffected() == 0)  {  return false;  }  return true;  }  int SQlitestorage::***insertLection***(const Lection &lection)  {  QSqlQuery query;  query.prepare("INSERT INTO lections (title, lectorer, students)"  "VALUES (:title, :lectorer, :students)");  query.bindValue(":title", QString::fromStdString(lection.title));  query.bindValue(":lectorer", QString::fromStdString(lection.lectorer));  query.bindValue(":students", lection.students);  if (!query.exec()){  qDebug() << "insertLection error:" << query.lastError();  return 0;  }  QVariant var = query.lastInsertId();  return var.toInt();  }  vector<Topic> SQlitestorage::***getAllTopics***(void)  {  vector<Topic> topics;  QSqlQuery query("SELECT \* FROM topics");  if (!query.exec()) {  qDebug() << "getAllTopics error:" << query.lastError().text();  }  while (query.next())  {  Topic newTopic;  newTopic.id = query.value("id").toInt();  newTopic.data = query.value("data").toString().toStdString();  newTopic.topic = query.value("topic").toString().toStdString();  newTopic.students = query.value("students").toInt();  topics.push\_back(newTopic);  }  return topics;  }  optional<Topic> SQlitestorage::***getTopicById***(int topic\_id)  {  QSqlQuery query;  query.prepare("SELECT \* FROM topics WHERE id = :id");  query.bindValue(":id", topic\_id);  if (!query.exec())  {  qDebug() << "getTopicById error:" << query.lastError();  return nullopt;  }  if (query.next())  {  Topic newTopic;  newTopic.id = query.value("id").toInt();  newTopic.data = query.value("data").toString().toStdString();  newTopic.topic = query.value("topic").toString().toStdString();  newTopic.students = query.value("students").toInt();  return newTopic;  }  else  {  return nullopt;  }  }  bool SQlitestorage::***updateTopic***(const Topic &topic)  {  QSqlQuery query;  query.prepare("UPDATE topics SET data = :data, topic = :topic, students = :students WHERE id = :id");  query.bindValue(":data", QString::fromStdString(topic.data));  query.bindValue(":topic", QString::fromStdString(topic.topic));  query.bindValue(":students",topic.students);  query.bindValue(":id", topic.id);  if (!query.exec()){  qDebug() << "updateTopic error:" << query.lastError();  return false;  }  return true;  }  bool SQlitestorage::***removeTopic***(int topic\_id)  {  QSqlQuery query;  query.prepare("DELETE FROM topics WHERE id = :id");  query.bindValue(":id", topic\_id);  if (!query.exec()){  qDebug() << "removeTopic error:" << query.lastError();  return false;  }  if(query.numRowsAffected() == 0)  {  return false;  }  return true;  }  int SQlitestorage::***insertTopic***(const Topic &topic)  {  QSqlQuery query;  query.prepare("INSERT INTO topics (data, topic, students) VALUES (:data, :topic, :students)");  query.bindValue(":data", QString::fromStdString(topic.data));  query.bindValue(":topic", QString::fromStdString(topic.topic));  query.bindValue(":students", topic.students);  if (!query.exec()){  qDebug() << "insertTopic error:" << query.lastError();  return 0;  }  QVariant var = query.lastInsertId();  return var.toInt();  } |

**￼Приклади результатів**



**Висновки**

Виконавши дану лабораторну роботу познайомилися із реляційними базами даних.

**База даних** - упорядкований набір логічно взаємопов'язаних даних, що використовується спільно, та призначений для задоволення інформаційних потреб користувачів. У технічному розумінні включно й система керування БД.

**Система керування базами даних** - спеціалізована програма (частіше комплекс програм), призначена для організації і ведення бази даних. Для створення і управління інформаційною системою СУБД необхідна в тій же мірі, як для розробки програми на алгоритмічній мові необхідний транслятор. СУБД дозволяє зосередитися на роботі з даними, абстрагувавшись від їх фізичного розміщення, а також бере на себе роботу, щодо ефективного їх збереження і вибірки.

**Реляційна база даних** – це база даних, побудована на основі реляційної моделі. У реляційній базі кожен об'єкт задається відповідним записом (рядком) в таблиці. Фактично реляційна база даних це тіло зв'язаної інформації, що зберігається в двовимірних таблицях.

**Особливість бази РБД** у тому, що зв'язок між таблицями може знаходити своє відображення в структурі даних, а може тільки матися на увазі, тобто бути присутнім на неформалізованому рівні. Реляційні бази даних надають більш простий доступ і складається зі звітів (зазвичай SQL) і забезпечують підвищену надійність і цілісність даних завдяки відсутності надлишкової інформації.

**Основними операторами SQL є:**

* **Оператори опису** даних: CREATE, DROP, ALTER
* **Оператори** маніпуляції даними: INSERT, DELETE, SELECT, UPDATE.
* **Оператори надання** прав доступу у базі данних: GRANT / REVOKE, LOCK / UNLOCK , SET LOCK MODE.