Введение

Вы почти закончили обучение и дошли до завершающей части курса, большая часть теории и практики осталась позади. Мы гордимся вашим трудолюбием и упорством! Теперь осталось совсем немного: выполнить финальную работу.

Финальная работа — это тренировка знаний и дополнительный кейс в портфолио. Кейсы интересуют работодателей в первую очередь. Результат финальной работы пригодится вам на собеседовании или станет дополнительным преимуществом при росте внутри компании.

Тема проекта — «Реализация системы поиска по корпоративному порталу компании „АйТиБокс“». Вы разработаете поисковый движок для корпоративного портала компании — аналог поисковой строки Яндекса или Google. Научитесь строить поисковые индексы, работать с конфигурационными файлами и попробуете написать набор тестов для проверки решения. Напишете формулу релевантности для выдачи результатов поиска.

Будет интересно!

Разработка локального поискового движка по файлам

Перед вами подробное описание будущего проекта, который станет отличным дополнением вашего портфолио. Здесь есть всё, что вам нужно, чтобы справиться с поставленной задачей.

Описание задачи

Представьте, что вы пришли в отдел разработки программного обеспечения недавно созданного информационно-новостного портала, на котором каждый день выходят новости о событиях в мире и статьи разных авторов. Руководитель поручил вам реализацию собственного поискового движка, который поможет осуществлять поиск среди набора документов.

Существующие поисковые движки ваши коллеги уже попробовали. Лучшим из них был ресурс от Яндекса, который можно было установить на своём сервере и использовать для поиска по своему набору документов. Но он перестал существовать как отдельный сервис. Ваше руководство решило реализовать собственный поиск, алгоритм и принципы работы которого при необходимости можно менять и развивать.

Поисковый движок должен представлять из себя консольное приложение (исполняемый файл, запускаемый на любом сервере или компьютере), осуществляющее поиск и имеющее возможность настройки через файлы формата JSON. Применённые в нём решения можно впоследствии встроить в поисковый движок работающий на веб.

Принципы работы поискового движка должны быть следующими:

1. В конфигурационном файле перед запуском приложения задаются названия файлов, по которым движок будет осуществлять поиск.

2. Поисковый движок должен самостоятельно обходить все файлы и индексировать их так, чтобы потом по любому поисковому запросу находить наиболее релевантные документы.

3. Пользователь задаёт запрос через JSON-файл requests.json. Запрос — это набор слов, по которым нужно найти документы.

4. Запрос трансформируется в список слов.

5. В индексе ищутся те документы, на которых встречаются все эти слова.

6. Результаты поиска ранжируются, сортируются и отдаются пользователю, максимальное количество возможных документов в ответе задаётся в конфигурационном файле.

7. В конце программа формирует файл answers.json, в который записывает результаты поиска.

Ниже вы найдёте все технические подробности реализации поискового движка, которые помогут вам создать работающее приложение. Они разбиты на несколько небольших этапов. По каждому этапу подробно расписано, что необходимо сделать и как проверить конечный результат.
Если у вас появятся вопросы или вы найдёте ошибку в техническом задании, смело обращайтесь к вашему куратору.

Этап 1. Подготовка

Цель
Подготовить необходимое программное обеспечение для программирования поискового движка.

Что нужно сделать

1. Установите на свой компьютер компилятор MinGW и IDE Clion, если их ещё нет.
2. В IDE Clion создайте проект search_engine и дополните его библиотекой сериализации JSON, как это описано в модуле 32.2.

Этап 2. Разработка класса для взаимодействия с файлами JSON в описанном формате

Цель
Добавить в созданный проект search_engine классы ConverterJSON для работы с файлами формата JSON.

Что нужно сделать
Напишите класс, который будет работать с данными в формате JSON. Класс должен выполнять следующие функции:

- считывать конфигурационные данные из JSON,
- преобразовывать запросы в формате JSON,
- формировать ответы в заданном формате JSON.

Для этого приложения конфигурационные данные будут храниться в файле с названием config.json, пользовательские запросы — в файле requests.json, а ответы — в answers.json.
Все три файла по умолчанию находятся в директории с проектом, поэтому отдельный путь для них задавать не нужно. Рассмотрим каждый из них подробнее:

1. Файл конфигурации config.json.

Без него запуск приложения невозможен. Он содержит название поискового движка, его версию, время обновления базы (с какой периодичностью необходимо делать переиндексирование базы, заново подгружать файлы и обсчитывать их поисковый рейтинг), максимальное количество вариантов в ответе (если не указано, то значение выбирается равным пяти).

Пример описания файла config.json:

```
{
	"config": {

		"name": "SkillboxSearchEngine",
		"version": "0.1",
		"max_responses": 5
	},
	"files": [
		"../resources/file001.txt",
		"../resources/file002.txt",
		"../resources/file003.txt",
		"../resources/file004.txt",
		…
	]
}
```

Подробнее разберём каждое поле файла config.json:

- *config* — общая информация, без которой приложение не запускается. Если это поле отсутствует, то при старте программа должна выбросить исключение и вывести в текстовую консоль ошибку config file is empty. Если отсутствует сам файл config.json, то необходимо выбросить исключение и вывести ошибку config file is missing.
- *name* — поле с названием поискового движка. Оно может быть любым: вы можете придумать название для поискового движка самостоятельно. Информацию из данного поля необходимо отображать при старте приложения Starting.
- *version* — поле с номером версии поискового движка. Впоследствии можно сделать проверку. Если поле version не совпадает с версией самого приложения, то необходимо выдавать ошибку config.json has incorrect file version.

- *max_responses* — поле, определяющее максимальное количество ответов на один запрос.
- *files* содержит пути к файлам, по которым необходимо осуществлять поиск. Внутри списка files лежат пути к файлам.
- <путь к файлу>(“../resorces/file001.txt”) — это путь к файлу. Впоследствии по его содержимому необходимо совершить поиск. Если по этому пути файл не существует, то на экран выводится соответствующая ошибка, но выполнение программы не прекращается. При этом каждый документ содержит не более 1000 слов с максимальной длиной каждого в 100 символов. Слова состоят из строчных латинских букв и разделены одним или несколькими пробелами.

2. Файл с запросами requests.json.

Он содержит запросы, которые необходимо обработать поисковому движку.
Пример описания файла requests.json:

```
{
	"requests": [
		"some words..",
		"some words..",
		"some words..",
		"some words..",
		…
	]
}
```

Разберём каждого поле файла config.json подробнее.

- *requests* состоит из списка запросов, которые необходимо обработать поисковым движком. Поле содержит не более 1000 запросов, каждый из которых включает от одного до десяти слов.
- <содержимое запроса>(“some words”) — поисковый запрос, набор слов, разделённых одним или несколькими пробелами. По ним необходимо осуществить поиск. Все слова состоят из строчных латинских букв.

3. Файл с ответами на запросы answers.json.

В него записываются результаты работы поискового движка. Если при старте приложения в директории с проектом не существует этого файла, то его необходимо создать. Если файл уже существует, то необходимо стереть всё его содержимое.
Пример описания файла answers.json:

```
{
	"answers": {
		"request001": {
			"result": "true",
```

```
                "relevance": {
                        "docid": 0, “rank” : 0.989,
                        "docid": 1, “rank” : 0.897,
                        "docid": 2, “rank” : 0.750,
                        "docid": 3, “rank” : 0.670,
                        "docid": 4, “rank” : 0.561
                }
        },
        "request002": {
                "result": "true",
                "docid": 0, “rank” : 0.769
        },
        "request003": {
                "result": "false"
        }
    }
}
```

Рассмотрим каждого поле файла answers.json:

- *answers* — базовое поле в этом файле, которое содержит ответы на запросы.
- *request001 … 003* — идентификатор запроса, по которому сформирован ответ. Идентификатор запроса формируется автоматически по порядку, в котором находятся запросы в поле requests файла requests.json. Например:
  "requests": [
  "some words..", для данной строки id запроса будет равен “request001”
  "some words..", для данной строки id запроса будет равен “request002”
  "some words..", для данной строки id запроса будет равен “request003”
  "some words..", для данной строки id запроса будет равен “request004”
  …
  ]

- *result* – результат поиска запроса. Если он принимает значение true, значит по данному запросу найден хотя бы один документ. Если результат имеет значение false, значит ни одного документа не найдено. Тогда других полей в ответе на этот запрос нет.
- *relevance* включается в файл answers.json, если на этот запрос удалось найти более одного документа.

Далее идут соответствия рейтинга ответа и названия id документа, в котором осуществлялся поиск:

- <Идентификатор документа>("docid") — идентификатор документа, в котором найден ответ на запрос. Он формируется автоматически при индексации всех документов исходя из порядка, в котором документы расположены в поле files в файле config.json. Например, если в поле config.json поле files содержит:
  "files": [
  "../resources/file001.txt", для данного файла docid будет равен 0

"../resources/file002.txt", для данного файла docid будет равен 1
"../resources/file003.txt", для данного файла docid будет равен 2
"../resources/file004.txt", для данного файла docid будет равен 3
…
]

- <ранг ответа>(“rank”) — ранг или поисковый рейтинг. Это число показывает, насколько документ подходит для заданного запроса. В ответе id документов располагаются в порядке уменьшения поискового рейтинга.

Для работы со всеми JSON-файлами, описанными выше, необходимо разработать класс ConverterJSON со следующим интерфейсом:

```
/**
*  Класс для работы с JSON-файлами
*/
class ConverterJSON {
public:
    ConverterJSON() = default;

    /**
     * Метод получения содержимого файлов
     * @return Возвращает список с содержимым файлов перечисленных
     *         в config.json
     */
    std::vector<std::string> GetTextDocuments();

    /**
     * Метод считывает поле max_responses для определения предельного
     *  количества ответов на один запрос
     * @return
     */
    int GetResponsesLimit();

    /**
     * Метод получения запросов из файла requests.json
     * @return возвращает список запросов из файла requests.json
     */
    std::vector<std::string> GetRequests();

    /**
     * Положить в файл answers.json результаты поисковых запросов
     */
    void putAnswers(std::vector<std::vector<std::pair<int, float>>>
answers);
}
```

Этап 3. Подключение системы тестирования к проекту.

Описание

Чтобы убедится, что приложение работает, как задумывалось, и упростить документирование и дальнейшее сопровождение проекта, применяется система юнит-тестирования (англ. unit testing). Модульное тестирование или юнит-тестирование — процесс в программировании, позволяющий проверить отдельные модули исходного кода программы на корректность.
Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже протестированных местах программы, а также облегчает обнаружение и устранение таких недочётов.
Для тестирования приложения выберем систему GTest. Вы можете выбрать любую другую (QtTest, Boost.Test) с возможностью автоматического запуска и поддержкой набора проверок, достаточных для отслеживания корректности работы приложения.

Цель
Подключить к проекту search_engine библиотеку Google Test для проведения автоматических модульных тестов. Для этого в файле проекта search_engine CMakeLists.txt, необходимо подключить библиотеку Google Test. Механизм подключения похож на то, как в прошедших модулях подключалась библиотека cpr.

Что нужно сделать

1. Библиотека Google Test используется напрямую из GitHub, поэтому в файле CMakeLists.txt необходимо добавить строки:

```
include(FetchContent)
FetchContent_Declare(
        googletest
        URL
https://github.com/google/googletest/archive/609281088cfefc76f9d0ce82e1ff6c30cc3591e5.zip
)
```

2. Добавьте include_directories:

```
include_directories(${gtest_SOURCE_DIR}/include ${gtest_SOURCE_DIR})
```

3. Если вы используете компилятор MinGW, то для корректной работы библиотеки необходимо добавить:

```
set(gtest_disable_pthreads on)
```

4. Выставите динамическое подключение библиотеки к проекту:

```
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
```

5. Сделайте все настройки доступными для проекта:

```
FetchContent_MakeAvailable(googletest)
```

6. Укажите, что в проекте будут применятся тесты с помощью строки:

```
enable_testing()
```

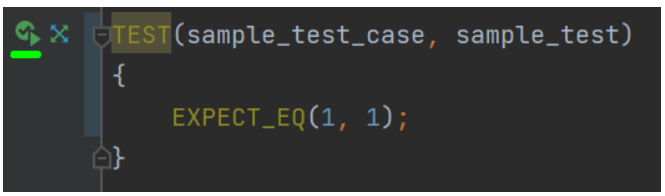
7. В конце добавьте библиотеку gtest_main:

```
target_link_libraries(search_engine PRIVATE gtest_main)

include(GoogleTest)
gtest_discover_tests(search_engine)
```

8. Проверьте работу при помощи короткого теста в проекте:

```
#include "gtest/gtest.h"

TEST(sample_test_case, sample_test)
{
    EXPECT_EQ(1, 1);
}
```

IDE Clion, как и многие другие среды разработки, может распознавать тесты в проектах. Поэтому слева от запускаемого теста должен появится значок:



Теперь, нажав правую кнопку мыши на этом значке, можно запустить этот тест отдельно (Run). При этом на нижней части CLion появится результат прохождения теста:

Это означает, что тест пройден успешно. Также отобразиться время, необходимое для его прохождения.

Дополнительные ресурсы

- Подробнее о библиотеке GTest можно узнать здесь.
- О том, как важно применять тесты для проектов, можно прочитать здесь.

Этап 4. Создание инвертированного индекса для документов

Описание

Скорость поиска по поисковому индексу в любых поисковых системах обычно занимает доли секунды. Это немного по сравнению с обычным перебором по всему массиву информации. Вспомните разницу в скорости поиска перебором в простом массиве и бинарного поиска в отсортированном. Это достигается за счёт заранее подготовленной информации.

Цель

Реализовать инвертированную индексацию документов — систему, которая позволит подсчитывать встречающиеся в документах слова.

Что нужно сделать

Необходимо написать класс InvertedIndex, который будет хранить и индексировать слова. Класс InvertedIndex будет принимать текстовые блоки и формировать из них инвертированный индекс. Инвертированный индекс (inverted index) — структура данных. В ней для каждого слова коллекции документов в соответствующем списке перечислены все документы в коллекции, в которых оно встретилось. Инвертированный индекс используется для поиска по текстам.

Класс InvertedIndex должен хранить внутри себя *идентификаторы документов*, списки уникальных слов к каждому из документов, а также посчитанные поисковые индексы.

Для хранения данных внутри класса InvertedIndex рекомендуется использовать следующие коллекции:

- *std::vector<std::string> docs* — коллекция для хранения текстов документов, в которой номер элемента в векторе определяет doc_id для формирования результата запроса;
- *std::map<std::string, std:vector<Entry>> freq_dictionary* или частотный словарь — это коллекция для хранения частоты слов, встречаемых в тексте. Entry представляет собой структуру:

```
struct Entry {
   size_t doc_id, count;
};
```

В коллекции *freq_dictionary* ключом служат слова из загруженных текстов, а значением — вектор из полей doc_id и count. Size_t — тип, используемый при задании размеров и индексации коллекций, обычно — это беззнаковый int, но правильнее писать именно size_t.

- *doc_id* — номер элемента в векторе docs, по которому хранится текст;
  count — число, которое обозначает, сколько раз ключевое слово встретилось в документе doc_id.

Например, по документам необходимо осуществить поиск с таким содержимым:

file001.txt:
milk sugar salt

file002.txt:
milk a milk b milk c milk d

Необходимо открыть файлы и загрузить содержимое в поле docs:

docs[0] = “milk sugar salt”;
docs[1] = “milk a milk b milk c milk d”;

Затем нужно разбить все тексты на слова и заполнить коллекцию *freq_dictionary* следующим образом:

index[“a”] = {1, 1};
index[“b”] = {1, 1};
index[“c”] = {1, 1};
index[“d”] = {1, 1};
index[“milk”] = {0, 1}, {1, 4};
index[“salt”] = {0, 1};
index[“sugar”] = {0, 1};

Пояснение: слово milk встречается в двух документах. В первом(file001.txt) — один раз, а во втором — четыре раза (file002.txt). Остальные слова встречаются лишь по одному разу.

Индексация, процесс подсчёта слов в тексте, должна запускаться каждый раз после вызова UpdateDocumentBase.

При запуске индексации поисковый движок должен выполнять следующие операции:

1. В отдельных потоках запускать индексацию каждого из файлов, перечисленных в конфигурационном файле.

2. Разбивать полученные текстовые блоки из класса ConverterJSON на отдельные слова.

3. Собирать все уникальные слова для документа и считать их количество.

4. Если слово отсутствует в базе, добавлять его в коллекцию *freq_dictionary* со значением count, равным единице. Если присутствует, то увеличивать число count на единицу. Число count должно соответствовать количеству этого слова в документе.

5. Добавлять связку слова и документа, на которой она встречается, в коллекцию *freq_dictionary* со значением count, равным количеству упоминаний в документе.

После этого этапа получится класс InvertedIndex и структура Entry с таким интерфейсом:

```
struct Entry {
    size_t doc_id, count;

    // Данный оператор необходим для проведения тестовых сценариев
    bool operator ==(const Entry& other) const {
        return (doc_id == other.doc_id &&
                count == other.count);
    }
};



class InvertedIndex {
public:
    InvertedIndex() = default;

    /**
     * Обновить или заполнить базу документов, по которой будем совершать
поиск
     * @param texts_input содержимое документов
     */
    void UpdateDocumentBase(std::vector<std::string> input_docs);

    /**
```

```
    * Метод определяет количество вхождений слова word в загруженной базе документов
    * @param word слово, частоту вхождений которого необходимо определить
    * @return возвращает подготовленный список с частотой слов
    */
   std::vector<Entry> GetWordCount(const std::string& word);

private:


   std::vector<std::string> docs; // список содержимого документов
   std::map<std::string, std::vector<Entry>> freq_dictionary; // частотный словарь
};
```

Чтобы убедиться, что класс InvertedIndex правильно заполняет коллекцию freq_dicitonary, предлагается использовать набор подготовленных тестов в системе Google Test:

```
using namespace std;

void TestInvertedIndexFunctionality(
       const vector<string>& docs,
       const vector<string>& requests,
       const std::vector<vector<Entry>>& expected
) {
   std::vector<std::vector<Entry>> result;
   InvertedIndex idx;

   idx.UpdateDocumentBase(docs);

   for(auto& request : requests) {
       std::vector<Entry> word_count = idx.GetWordCount(request);
       result.push_back(word_count);
   }

   ASSERT_EQ(result, expected);
}

TEST(TestCaseInvertedIndex, TestBasic) {
   const vector<string> docs = {
           "london is the capital of great britain",
           "big ben is the nickname for the Great bell of the striking clock"
   };
   const vector<string> requests = {"london", "the"};
   const vector<vector<Entry>> expected = {
                                          {
                                              {0, 1}
                                           }, {
                                              {0, 1}, {1, 3}
                                           }
```

```
                                                };
    TestInvertedIndexFunctionality(docs, requests, expected);
}

TEST(TestCaseInvertedIndex, TestBasic2) {
    const vector<string> docs = {
            "milk milk milk milk water water water",
            "milk water water",
            "milk milk milk milk milk water water water water water",
            "Americano Cappuccino"
    };
    const vector<string> requests = {"milk", "water", "cappuchino"};
    const vector<vector<Entry>> expected = {
            {
                    {0, 4}, {1, 1}, {2, 5}
            }, {
                    {0, 2}, {1, 2}, {2, 5}
            }, {
                    {3, 1}
            }
    };
    TestInvertedIndexFunctionality(docs, requests, expected);
}

TEST(TestCaseInvertedIndex, TestInvertedIndexMissingWord) {
    const vector<string> docs = {
            "a  b  c  d  e  f  g  h  i  j  k  l",
            "statement"
    };
    const vector<string> requests = {"m", "statement"};
    const vector<vector<Entry>> expected = {
            {
                    {}
            }, {
                    {1, 1}
            }
    };
    TestInvertedIndexFunctionality(docs, requests, expected);
}
```

Этап 5. Система индексации документов

Описание

Индексация — это процесс формирования поискового индекса по некоторому объёму информации, определения релевантности ответов для заданных запросов.

Цель

Реализовать систему определения релевантности поискового запроса.

Что нужно сделать

Разработайте основной класс SearchServer, который позволит определять наиболее релевантные, соответствующие поисковому запросу документы по прочитанным из файла requests.json поисковым запросам.

Алгоритм выдачи результатов поиска

Если индексирование документов завершилось, то по нему можно осуществлять поиск. В этом случае поисковый движок считывает запросы из файла requests.json и выполняет следующие операции:

1. Разбивает поисковый запрос на отдельные слова.
2. Формирует из них список уникальных.
3. Сортирует слова в порядке увеличения частоты встречаемости: от самых редких до самых частых. По возрастанию значения поля count поля freq_dictionary.
4. По первому, самому редкому слову из списка находит все документы, в которых встречается слово.
5. Далее ищет соответствия следующего слова и этого списка документов. Так по каждому следующему слову. Список документов на каждой итерации должен уменьшаться или, по крайней мере, не увеличиваться.
6. Если в итоге не осталось ни одного документа, то выводит количество найденных документов, равное 0. В результат ответа записывает false.
7. Если документы найдены, рассчитывает по каждому из них релевантность и выводит её в поле rank в ответе на запрос. Для этого для каждой страницы рассчитывается абсолютная релевантность — сумма всех count всех найденных в документе слов из коллекции freq_dictionary, которая делится на максимальное значение абсолютной релевантности для всех найденных.

Пример расчёта:

Документ (doc_id)	Frequency слова «лошадь»	Frequency слова «бегает»	Абсолютная релевантность	Относительная релевантность
1	4	3	7	0,7
2	1	2	3	0,3
3	5	5	10	1

Пример расчёта абсолютной релевантности для первого документа:

$$R_{abs} = 4 + 3 = 7$$

Относительную релевантность можно получить делением абсолютной для конкретного документа на максимальную абсолютную релевантность среди всех документов для данной поисковой выдачи:

$$R_{rel} = 7 / 10 = 0,7$$

8. Сортирует страницы по убыванию релевантности: от большей к меньшей.

9. Записывает результат работы в файл answers.json в соответствии с форматом.

Обработка поискового запроса будет происходить в классе SearchServer с таким интерфейсом:

```
struct RelativeIndex{
   size_t doc_id;
   float rank;

   bool operator ==(const RelativeIndex& other) const {
       return (doc_id == other.doc_id && rank == other.rank);
   }
};

class SearchServer {
public:
   /**
    * @param idx в конструктор класса передаётся ссылка на класс
InvertedIndex,
    *      чтобы SearchServer мог узнать частоту слов встречаемых в
запросе
    */
    SearchServer(InvertedIndex& idx) : _index(idx){ };
    /**
     * Метод обработки поисковых запросов
     * @param queries_input поисковые запросы взятые из файла
requests.json
     * @return возвращает отсортированный список релевантных ответов для
заданных запросов
     */
   std::vector<std::vector<RelativeIndex>> search(const
std::vector<std::string>& queries_input);
private:
   InvertedIndex _index;
};
```

Чтобы убедиться, что приложение правильно рассчитывает абсолютную и относительную релевантность, предлагается использовать набор подготовленных тестов в системе Google Test:

```
TEST(TestCaseSearchServer, TestSimple) {
    const vector<string> docs = {
            "milk milk milk milk water water water",
            "milk water water",
            "milk milk milk milk milk water water water water water",
            "Americano Cappuccino"
    };

    const vector<string> request = {"milk water", "sugar"};
    const std::vector<vector<RelativeIndex>> expected = {
            {
                    {2, 1},
                    {0, 0.7},
                    {1, 0.3}
            },
            {
                    {}
            }
    };

    InvertedIndex idx;
    idx.UpdateDocumentBase(docs);

    SearchServer srv(idx);

    std::vector<vector<RelativeIndex>> result = srv.search(request);

    ASSERT_EQ(result, expected);
}

TEST(TestCaseSearchServer, TestTop5) {
    const vector<string> docs = {
            "london is the capital of great britain",
            "paris is the capital of france",
            "berlin is the capital of germany",
            "rome is the capital of italy",
            "madrid is the capital of spain",
            "lisboa is the capital of portugal",
            "bern is the capital of switzerland",
            "moscow is the capital of russia",
            "kiev is the capital of ukraine",
            "minsk is the capital of belarus",
            "astana is the capital of kazakhstan",
            "beijing is the capital of china",
            "tokyo is the capital of japan",
            "bangkok is the capital of thailand",
            "welcome to moscow the capital of russia the third rome",
            "amsterdam is the capital of netherlands",
            "helsinki is the capital of finland",
            "oslo is the capital of norway",
```

```
            "stockholm is the capital of sweden",
            "riga is the capital of latvia",
            "tallinn is the capital of estonia",
            "warsaw is the capital of poland",
    };

    const vector<string> request = {"moscow is the capital of russia"};
    const std::vector<vector<RelativeIndex>> expected = {
            {
                    {7, 1},
                    {14, 1},
                    {0, 0.4},
                    {1, 0.4},
                    {2, 0.4}
            }
    };

    InvertedIndex idx;
    idx.UpdateDocumentBase(docs);

    SearchServer srv(idx);

    std::vector<vector<RelativeIndex>> result = srv.search(request);

    ASSERT_EQ(result, expected);
}
```

Этап 6. Размещение в GitHub

Цель

Научиться размещать проекты в публичном доступе для демонстрации при презентации проектов и при общении с потенциальными работодателями.

Что нужно сделать

1. Разместите исходные коды вашего приложения в публичном доступе в своём GitHub. Создайте в корне репозитория файл README.md. В него добавьте:

    - описание проекта;
    - описание стека используемых технологий;
    - краткую инструкцию, как запустить проект локально: команды, действия, переменные среды, которые обязательно требуется задать.

О том как создать репозиторий на GitHub и разместить на нём свой проект вам поможет руководство:
«Загружаем проект в удалённый репозиторий через GitHub Desktop»

Как составить красивый и понятный README.md вам поможет руководство «Как написать красивый и информативный README.md».

Этап 7. Подготовка к сдаче проекта

Цель
Подготовить проект к сдаче на проверку куратору.

Что нужно сделать
Проверьте проект по чек-листу, описанному ниже. Оцените каждый пункт чек-листа отдельно и убедитесь, что проект выполняет все требования. Если все пункты чек-листа полностью выполнены, значит работа готова к сдаче. Работа сдаётся на проверку в виде ссылки на репозиторий GitHub.

Чек-лист проекта

- ☐ Реализованы все классы, необходимые в проекте: ConverterJSON, InvertedIndex, SearchServer.

- ☐ Все функции, переменные, классы имеют корректные названия:
  - не используют латиницу;
  - не используют аббревиатуры, если это не общепринятые аббревиатуры или не распространённые названия, которые используются внутри проекта;
  - используют имена существительные для названия переменных и классов;
  - используют глаголы для названий методов классов и функций;
  - используют единый стиль именования snake_case или camelCase;
  - используют короткие, читаемые, легко произносимые названия;
  - не используют длинные названия howMuchEachApartmentWillPay или сокращения hmeap;
  - отражают роль объекта в программе, а не его содержимое.

- ☐ Классы успешно проходят все модульные тесты.

- ☐ Индексация документов запускается в отдельных потоках.

- ☐ Для слов в документе рассчитывается их ранг по формуле.

- ☐ Полностью реализован метод search, результат помещается в файл answers.json в корректном формате.

- ☐ Для каждой страницы в результате рассчитана её релевантность по формуле.

- ☐ Поиск возвращает результаты в порядке их релевантности.
- ☐ Метод поиска возвращает общее количество результатов.
- ☐ Проект выложен на GitHub со всеми необходимыми файлами для сборки.
- ☐ В README описана команда, по которой необходимо запускать приложение после сборки.

Критерии оценки финальной работы

Готовый проект мы будем проверять по критериям:

1. Проект компилируется, собирается и запускается.
2. Все JSON-файлы, используемые приложением, соответствуют формату описанному во втором этапе и являются валидными.
3. Все пункты из чек-листа проекта выполнены.
4. В бизнес-логике нет ошибок.

Проект возвращается на доработку по этим критериям:

1. Куратор не может запустить проект локально.
2. Отсутствует или не работает базовая функциональность: индексация, поиск.
3. Формат результирующего JSON-файла имеет структуру, отличную от описанной на втором этапе.
4. Хотя бы один из пунктов чек-листа выполнен частично.

Рекомендации по работе над проектом

Мы рекомендуем составить план работы по датам. Исходя из нашего опыта, продуктивнее выделять на проект по два-три часа несколько дней в неделю, чем делать этот же объём за один подход. Лучше придерживаться такого графика и обязательно выделять время для отдыха.

Отмечайте свой прогресс по мере выполнения плана. Это полезно по нескольким причинам:

- вы будете держать ритм,
- сможете контролировать ситуацию.

И самое главное: каждый выполненный этап — это ваша маленькая победа. Чем больше таких побед вы будете замечать, тем больше удовольствия получите от выполненного проекта.