Было бы странно в курсе от Яндекса и не попросить вас сделать свою поисковую систему. В финальной задаче «Красного пояса» мы именно этим и займёмся :)

В этой задаче вам надо поработать с классом SearchServer, который позволяет выполнять поиск в базе документов:

class SearchServer {

public:

  SearchServer() = default;

  explicit SearchServer(istream& document\_input);

  void UpdateDocumentBase(istream& document\_input);

  void AddQueriesStream(istream& query\_input, ostream& search\_results\_output);

};

Рассмотрим его интерфейс.

**Конструктор**

Конструктор класса SearchServer принимает поток ввода, содержащий базу документов. При этом

* один документ — это одна строка входного потока;
* документы состоят из слов, разделённых одним или несколькими пробелами;
* слова состоят из строчных латинских букв. Например, код, приведённый ниже, загружает в объект класса SearchServer базу из трёх документов:

string docs;

docs += "london is the capital of great britain\n";

    // документ содержит 7 слов

docs += "i am travelling down the river\n";

    // документ содержит 6 слов

docs += "  here     we    go             ";

    // документ содержит 3 слова

istringstream document\_input(docs);

SearchServer srv(document\_input);

### Метод AddQueriesStream(istream& query\_input, ostream& search\_results\_output)

Метод AddQueriesStream выполняет собственно поиск. Он принимает входной поток поисковых запросов и выходной поток для записи результатов поиска. При этом

* один запрос — это одна строка в потоке query\_input
* каждый поисковый запрос состоит из слов, разделённых одним или несколькими пробелами
* так же, как и в документах, слова в запросах состоят из строчных латинских букв

Результатом обработки поискового запроса является набор из **максимум пяти** наиболее релевантных документов. В реальных поисковых системах метрика релевантности устроена довольно сложно. В рамках нашей задачи в качестве метрики релевантности мы будем использовать суммарное количество вхождений всех слов запроса в документ. Например, допустим, у нас есть поисковая база из трёх документов: "london is the capital of great britain", "moscow is the capital of the russian federation", "paris is the capital of france", — и поисковый запрос "the best capital". Тогда метрика релевантности у наших документов будет такой:

* london is the capital of great britain — 2 (слово "the" входит в документ 1 раз, слово "best" — ни разу, слово "capital" — 1 раз)
* moscow is the capital of the russian federation — 3 (слово "the" входит в документ 2 раза, слово "best" — ни разу, слово "capital" — 1 раз)
* paris is the capital of france — 2 ("the" — 1, "best" — 0, "capital" — 1)

В итоге получается, что документ "moscow is the capital of the russian federation" оказывается наиболее релевантным запросу "the best capital".

Для каждого поискового запроса метод AddQueriesStream должен вывести в поток search\_results\_output одну строку в формате [текст запроса]: {docid: <значение>, hitcount: <значение>} {docid: <значение>, hitcount: <значение>} ..., где docid — идентификатор документа (см. ниже), а hitcount — значение метрики релевантности для данного документа (то есть суммарное количество вхождений всех слов запроса в данный документ).

Два важных замечания:

* Добавлять в результаты поиска документы, hitcount которых равен нулю, не нужно.
* при подсчёте hitcount нужно учитывать только слова целиком, то есть слово «there» не является вхождением слова «the»

### Метод UpdateDocumentBase(istream& document\_input)

Метод UpdateDocumentBase заменяет текущую базу документов на новую, которая содержится в потоке document\_input. При этом документ из первой строки этого потока будет иметь идентификатор (docid) 0, документ из второй строки — идентификатор 1 и т.д. Точно так же должен назначать идентификаторы документам и конструктор класса SearchServer. Например, код

const string doc1 = "london is the capital of great britain";

const string doc2 = "moscow is the capital of the russian federation";

istringstream doc\_input1(doc1 + '\n' + doc2);

SearchServer srv(doc\_input1);

const string query = "the capital";

istringstream query\_input1(query);

srv.AddQueriesStream(query\_input1, cout);

istringstream doc\_input2(doc2 + '\n' + doc1);

srv.UpdateDocumentBase(doc\_input2);

istringstream query\_input2(query);

srv.AddQueriesStream(query\_input2, cout);

должен выводить

the capital: {docid: 1, hitcount: 3} {docid: 0, hitcount: 2}

the capital: {docid: 0, hitcount: 3} {docid: 1, hitcount: 2}

### Первая часть задачи

Эта задача состоит из двух частей. В первой части вам дана корректная реализация класса SearchServer, которая работает недостаточно быстро. Вам нужно найти и устранить узкие места в её реализации. В тестирующую систему нужно сдать cpp-файл или архив из нескольких исходных файлов, содержащий вашу ускоренную реализацию.

Тестирование реализации будет проводиться на производительность и на целостность.

При тестировании на производительность, Ваша реализация будет тестироваться вот такой функцией (объект класса SearchServer будет создан один раз, и у него один раз будет вызван метод AddQueriesStream):

void TestSearchServer(istream& document\_input, istream& query\_input,

                      ostream& search\_results\_output) {

  SearchServer srv(document\_input);

  srv.AddQueriesStream(query\_input, search\_results\_output);

}

При этом:

* document\_input содержит не более 50000 документов
* каждый документ содержит не более 1000 слов
* общее число различных слов во всех документах не превосходит 15000
* максимальная длина одного слова — 100 символов, слова состоят из строчных латинских букв и разделены одним или несколькими пробелами
* query\_input содержит не более 500 000 запросов, каждый запрос содержит от 1 до 10 слов.

В отличие от тестирования на производительность, при тестировании на целостность и корректность работы исходного функционала сервера допускается вызов UpdateDocumentBase в том числе и у уже созданного и проинициализированного объекта класса SearchServer.

### Вторая часть задачи

В этой части мы смоделируем работу web-сервера, который одновременно обрабатывает несколько потоков поисковых запросов, а также обновления базы документов. Ваша реализация будет тестироваться так:

void TestSearchServer(vector<pair<istream, ostream\*>> streams) {

  // IteratorRange — шаблон из задачи Paginator

  // random\_time() — функция, которая возвращает случайный

  // промежуток времени

  LOG\_DURATION("Total");

  SearchServer srv(streams.front().first);

  for (auto& [input, output] :

       IteratorRange(begin(streams) + 1, end(streams))) {

    this\_thread::sleep\_for(random\_time());

    if (!output) {

      srv.UpdateDocumentBase(input);

    } else {

      srv.AddQueriesStream(input, \*output);

    }

  }

}

Как видите, с помощью макроса LOG\_DURATION мы замеряем общее время работы тестирующей функции. Ваша реализация должна работать быстрее, чем простое последовательное применение методов UpdateDocumentBase и AddQueriesStream.

На практике такой web-сервер бо̀льшую часть времени обрабатывает поисковые запросы. База документов обновляется не очень часто с определённой периодичностью, например, один раз в сутки. Обновление базы может занимать значительное время, поэтому сервер не должен прекращать обработку запросов, пока оно выполняется. Ваше решение второй части задачи должно это учитывать.

1. Метод AddQueriesStream должен быть готов к тому, что параллельно с ним будет выполняться метод UpdateDocumentBase и наоборот.
2. Метод AddQueriesStream не обязан обрабатывать все запросы с той версией базы документов, которая была актуальна во время его запуска. То есть, если во время работы метода AddQueriesStream база документов была обновлена, он может использовать новую версию базы для обработки оставшихся запросов.