Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и кибербезопасности
Высшая школа программной инженерии

РАСЧЁТНО-ГРАФИЧЕСКАЯ РАБОТА

по дисциплине **«Технологии программирования»**

Выполнил
студент гр. 5130904/40003 Николаев А.Д.

Руководитель Череповский Д.К.

« 08 » мая 2025г.

Санкт-Петербург
2025 г

Содержание

Введение. Общая постановка задачи

Целью расчетно-графической работы по дисциплине «Технология программирования» является разработка программы, реализующей функциональность англо-русского словаря с использованием средств стандартной библиотеки C++ (STL). Задание основано на аналогичной задаче из РГР по дисциплине «Алгоритмы и структуры данных», но вместо собственных структур данных применяются контейнеры, умные указатели и алгоритмы стандартной библиотеки.

Программа должна предоставлять консольный интерфейс для взаимодействия с пользователем посредством ввода команд. Признаком завершения ввода служит сигнал EOF (Ctrl+Z на Windows или Ctrl+D на Linux). Поддерживаемые команды согласованы с преподавателем и включают следующие операции:

- Добавление нового английского слова и его русских переводов.
- Удаление слова из словаря.
- Добавление нового перевода к существующему слову.
- Удаление перевода для слова.
- Поиск всех переводов по английскому слову.
- Поиск английского слова по русскому переводу.
- Автоматический перевод текста с определением языка ввода (английский или русский).
- Вывод всех слов и их переводов из словаря.

В процессе разработки требуется:

Использовать контейнеры STL **(std::map, std::list и др.)** для хранения словаря. Применять умные указатели (**std::unique_ptr)** для управления ресурсами. Использовать алгоритмы STL.

Реализовать обработку ошибок через исключения, включая проверку корректности ввода. Результатом работы является программа

Описание получившейся программы и совершаемых в ней действий

Алгоритмы, используемые в файлах MyJson.cpp и Dict.cpp

Обзор структуры и алгоритмов

Класс Dict реализует англо-русский словарь, использующий красно-черное дерево (std::map) для хранения пар "слово-переводы". Ключом является английское слово (std::string), а значением — отсортированный список русских переводов (std::list<std::string>). Дополнительно хранится путь к JSON-файлу (dataPath_), используемому для персистентного хранения данных. Класс поддерживает операции добавления, удаления, поиска слов и переводов, а также автоматический перевод. Валидация ввода, управление файлами и интерактивный интерфейс обеспечивают удобство работы. Класс MyJson отвечает за парсинг и сериализацию данных в формате JSON. Основные алгоритмы и их характеристики описаны ниже.

1. Алгоритмы в MyJson.cpp

Файл MyJson.cpp реализует функциональность для парсинга JSON-данных и их преобразования в строку. Используются следующие алгоритмы:

1.1. Пропуск пробельных символов (skipWhitespace)

- **Описание**: Рекурсивная функция, пропускающая пробельные символы (пробелы, табуляции, переносы строк) в JSON-строке.
- **Алгоритм**: Проверяет текущий символ через isspace. Рекурсивно вызывает себя для следующей позиции, если символ пробельный.
- **Сложность**: O(n), где n — длина оставшейся части строки.

1.2. Парсинг строк (parseString)

- **Описание**: Извлекает строку в кавычках, обрабатывая экранированные символы (например, \").
- **Алгоритм**: Использует std::find_if для поиска закрывающей кавычки (").
- **Сложность**: O(n), где n — длина строки до закрывающей кавычки.

1.3. Парсинг массивов (parseArray)

- **Описание:** Разбирает JSON-массив строк вида ["a", "b"], сохраняя элементы в вектор Translations.

- **Алгоритм**:
  - Пропуск пробельных символов.
  - Проверка открывающей скобки [.
  - Рекурсивно извлекает строки через parseString.
  - Обработка запятых и закрывающей скобки].
- **Сложность**: O(n), где n — длина массива в строке JSON.

1.4. Парсинг JSON-объекта (parse)

- **Описание**: Преобразует JSON-строку в словарь std::map<std::string, std::list<std::string>>..
- **Алгоритм:**
- Пропускает пробелы и проверяет открывающую скобку {.
- Для каждой пары ключ-значение: Извлекает ключ (parseString). Проверяет двоеточие. Извлекает массив значений (parseArray). Сортирует и удаляет дубликаты переводов (sort() + unique()).
- Завершает парсинг при закрывающей скобке }.
- **Сложность**:
  - Парсинг: O(n), где n — длина JSON-строки.
  - Сортировка: O(m log m) для каждого ключа, где m — количество переводов.
  - Удаление дубликатов: O(m).
  - Итоговая сложность зависит от числа ключей и переводов.

1.5. Преобразование словаря в JSON-строку (convertToString)

- **Описание**: Преобразует словарь (std::map) в JSON-строку.
- **Алгоритм**: Итерирует по словарю через std::for_each
- **Сложность**: O(n), где n — суммарная длина всех ключей и переводов.

2. Алгоритмы в Dict.cpp

Файл Dict.cpp реализует функциональность англо-русского словаря с использованием красно-черного дерева (std::map). Основные алгоритмы описаны ниже.

2.1. Конструктор и загрузка данных (Dict, loadFromFile)

- **Описание**: Конструктор Dict(const std::string& filename) инициализирует словарь, загружая данные из JSON-файла, и сохраняет путь в dataPath_. Метод loadFromFile считывает файл и парсит его.
- **Алгоритм**:
  - Чтение файла в строку с использованием std::istreambuf_iterator.
  - Вызов MyJson::parse для заполнения tree_.
  - Если файл отсутствует, словарь остаётся пустым.
- **Сложность**: O(n) для чтения файла + сложность парсинга JSON (см. выше).
- **Входные данные**: Путь к файлу (std::string).
- **Выходные данные**: Нет (модифицирует tree_ и dataPath_).

2.2. Деструктор (~Dict)

- **Описание**: Освобождает ресурсы словаря.
- **Алгоритм**: Полагается на автоматическое управление памятью STL (std::map, std::string). Дополнительных действий не выполняется.
- **Сложность**: O(1).
- **Входные/Выходные данные**: Нет.

2.3. Запись данных в файл (writeToFile, removeFromFile)

- **Описание**: writeToFile сохраняет словарь в JSON-файл. removeFromFile удаляет слово или перевод и обновляет файл.
- **Алгоритм**:
  - writeToFile: Преобразование tree_ в JSON-строку через MyJson::convertToString и запись в файл по dataPath_.
  - removeFromFile: Удаление слова (tree_.erase) или перевода (std::remove) из tree_, затем вызов writeToFile.
  - При ошибке открытия файла выбрасывается FailOfMemoryAllocation.
- **Сложность**:
  - writeToFile: O(n) для преобразования и записи, где n — суммарная длина данных.
  - removeFromFile: O(log n) для удаления слова + O(k) для удаления перевода, где k — длина списка переводов.

- **Входные данные** (removeFromFile): Слово (std::string), перевод (std::string* или nullptr).

- **Выходные данные**: Нет.

2.4. Вставка в отсортированный список (insertSorted)

- **Описание**: Вставляет перевод в отсортированный список, избегая дубликатов.
- **Алгоритм**: Использование std::lower_bound для поиска позиции вставки. Если перевода нет, он вставляется в список.
- **Сложность**: O(log k), где k — длина списка переводов.
- **Входные данные**: Список переводов (std::liststd::string&), перевод (std::string).
- **Выходные данные**: Нет (модифицирует список).

2.5. Добавление слова и переводов (insert)

- **Описание**: Добавляет новое слово и его переводы в словарь.
- **Алгоритм**:
  - Проверка на пустое слово и дубликаты.
  - Разделение строки переводов (translationsStr) на элементы с разделителем ; с помощью std::istringstream и рекурсивной лямбда-функции.
  - Валидация переводов (isRussianWord), приведение к нижнему регистру (toLowerCaseR).
  - Вставка переводов в отсортированный список (insertSorted).
  - Добавление пары в tree_ и обновление файла.
- **Сложность**:
  - Разделение строки: O(n), где n — длина строки переводов.
  - Валидация и преобразование: O(m) для каждого перевода, где m — длина перевода.
  - Вставка: O(log k) для каждого перевода, где k — длина списка.
  - Итоговая сложность: O(n + m log k).
- **Входные данные**: Слово (std::string), переводы (std::string).
- **Выходные данные**: Нет (модифицирует tree_).

2.6. Добавление перевода (addTranslation)

- **Описание**: Добавляет новый перевод для существующего слова.
- **Алгоритм**:
  - Поиск слова в tree_ (std::map::find, O(log n)).
  - Приведение перевода к нижнему регистру (toLowerCaseR).
  - Проверка уникальности перевода (std::lower_bound).
  - Вставка перевода в отсортированный список.
  - Обновление файла.
- **Сложность**: O(log n + log k), где n — число слов, k — число переводов.
- **Входные данные**: Слово (std::string), перевод (std::string&).
- **Выходные данные**: Нет (модифицирует tree_).

2.7. Удаление перевода или слова (removeTranslation, remove)

- **Описание**: removeTranslation удаляет перевод слова, remove удаляет слово целиком.
- **Алгоритм**:
  - Поиск слова (std::map::find, O(log n)).
  - Для removeTranslation: Поиск перевода (std::lower_bound, O(log k)), удаление перевода, удаление слова при пустом списке переводов.
  - Для remove: Удаление слова из tree_ (std::map::erase).
  - Обновление файла через removeFromFile.
- **Сложность**:
  - removeTranslation: O(log n + log k).
  - remove: O(log n).
  - Запись в файл: O(n).
- **Входные данные**: Слово (std::string), перевод (std::string) для removeTranslation.
- **Выходные данные**: Нет (модифицирует tree_).

2.8. Поиск переводов по слову (findTranslationByWord)

- **Описание**: Возвращает список переводов для заданного слова.
- **Алгоритм**: Поиск ключа в std::map с помощью find.
- **Сложность**: O(log n), где n — число слов.

- **Входные данные**: Слово (std::string).
- **Выходные данные**: Список переводов (const std::liststd::string&).

2.9. Поиск слова по переводу (findWordByTranslation)

- **Описание**: Находит слово по заданному переводу.
- **Алгоритм**:
  - Приведение перевода к нижнему регистру (toLowerCaseR).
  - Линейный проход по словарю (std::find_if) с использованием std::binary_search для проверки переводов.
- **Сложность**: O(n log k), где n — число слов, k — средняя длина списка переводов.
- **Входные данные**: Перевод (std::string).
- **Выходные данные**: Слово (std::string).

2.10. Автоматический перевод (autoTranslate)

- **Описание**: Определяет язык ввода и выполняет перевод.
- **Алгоритм**:
  - Проверка языка (isEnglishWord или isRussianWord).
  - Вызов findTranslationByWord (для английского) или findWordByTranslation (для русского).
  - Форматирование результата с объединением переводов через ;.
- **Сложность**: O(log n) для английского слова или O(n log k) для русского перевода.
- **Входные данные**: Текст (std::string&).
- **Выходные данные**: Результат перевода (std::string).

2.11. Ввод команды (getCommandInput)

- **Описание**: Считывает команду пользователя.
- **Алгоритм**: Считывание строки, удаление пробелов (trimSpaces), приведение к верхнему регистру (toUpperCaseE).
- **Сложность**: O(n), где n — длина строки.
- **Входные данные**: Нет.
- **Выходные данные**: Команда (std::string).

2.12. Вывод приветствия (printWelcome)

- **Описание**: Выводит приветственное сообщение и список команд.
- **Алгоритм**: Вывод статического текста с описанием операций (INSERT, REMOVE, ADDTRANSLATION и др.).
- **Сложность**: O(1).
- **Входные/Выходные данные**: Нет.

2.13. Запуск интерактивного режима (run)

- **Описание**: Реализует интерактивный интерфейс словаря.
- **Алгоритм**:
  - Вывод приветствия (printWelcome).
  - Рекурсивная лямбда-функция для обработки команд (getCommandInput).
  - Вызов соответствующих методов (insert, remove, addTranslation, и др.).
  - Обработка исключений (std::logic_error, std::runtime_error) с возвратом к вводу.
- **Сложность**: Зависит от вызываемых методов.
- **Входные/Выходные данные**: Нет.

2.14. Проверка корректности слов (isEnglishWord, isRussianWord)

- **Описание**: Проверяют, является ли строка английским или русским словом.
- **Алгоритм**:
  - isEnglishWord: Проверка символов на принадлежность к ASCII (буквы, апостроф, дефис, запятая, пробел) с помощью std::all_of.
  - isRussianWord: Рекурсивная проверка байтов UTF-8 для русских букв.
- **Сложность**: O(n), где n — длина строки.
- **Входные данные**: Строка (std::string).
- **Выходные данные**: bool.

2.15. Преобразование регистра (toUpperCaseE, toLowerCaseE, toLowerCaseR)

- **Описание**: Преобразуют строку в верхний/нижний регистр для английских или русских букв.
- **Алгоритм**:
  - toUpperCaseE, toLowerCaseE: Использование std::transform с std::toupper или std::tolower для ASCII.
  - toLowerCaseR: Рекурсивная обработка UTF-8 байтов для русских букв.

- **Сложность**: O(n), где n — длина строки.
- **Входные данные**: Строка (std::string&).
- **Выходные данные**: Нет (модифицирует строку).

2.16. Удаление пробелов (trimSpaces)

- **Описание**: Удаляет начальные и конечные пробельные символы.
- **Алгоритм**: Использование std::find_if_not для поиска непробельных границ и обрезки строки.
- **Сложность**: O(n), где n — длина строки.
- **Входные данные**: Строка (std::string&).
- **Выходные данные**: Нет (модифицирует строку).

2.17. Вывод словаря (print)

- **Описание**: Выводит содержимое словаря в консоль.
- **Алгоритм**: Итерация по tree_ с использованием std::for_each, вывод слов и переводов, объединённых через ;.
- **Сложность**: O(n), где n — суммарная длина всех слов и переводов.
- **Входные/Выходные данные**: Нет.

3. Используемые структуры данных

- **Красно-черное дерево (std::map)**: Хранит пары "слово-переводы". Обеспечивает O(log n) для поиска, вставки и удаления, где n — число слов.
- **std::list**: Хранит отсортированные списки переводов, поддерживает эффективную вставку и удаление (O(1) при известной позиции).
- **std::vector (в MyJson)**: Временное хранение переводов при парсинге.
- **std::ostringstream, std::istringstream**: Для форматирования и разбора строк.

4. Основные алгоритмические подходы

- **Красно-черное дерево**: Обеспечивает логарифмическую сложность (O(log n)) для операций с ключами.
- **Бинарный поиск**: Используется для поиска переводов (std::lower_bound, std::binary_search, O(log k)).
- **Сортировка**: Применяется для поддержания порядка переводов (std::sort, O(m log m)).

- **Рекурсия**: Используется в isRussianWord, toLowerCaseR, и для обработки команд в run.
- **Линейная обработка**: Для парсинга JSON, чтения/записи файлов и обработки строк.
- **Валидация**: Проверка ASCII для английских слов и UTF-8 для русских.

5. Общая характеристика

Класс Dict использует красно-черное дерево для эффективного хранения и доступа к данным (O(log n)). Переводы хранятся в отсортированных списках, что обеспечивает быстрый поиск и вставку (O(log k)). JSON-сериализация через MyJson имеет линейную сложность (O(n)). Интерактивный интерфейс реализован с рекурсивной обработкой команд, что упрощает добавление новых функций. Валидация ввода (ASCII для английского, UTF-8 для русского) обеспечивает корректность данных. Класс предоставляет надёжный и удобный интерфейс для работы со словарём.

Описание тестирования

При запуске программы автоматически прогоняются тесты. Пользователю виден результат теста и что тестировалось. После этого у пользователя есть возможность самостоятельно работать со словарём.

Автоматическое тестирование	
Добавление в словарь	Тестирует корректность добавления слова с несколькими переводами, проверяя количество и порядок добавленных переводов для слова "hello".
Дублирование слова	Проверяет, что при попытке добавить уже существующее слово "hello" выбрасывается исключение DuplicateWord.
Добавление перевода	Тестирует добавление нового перевода "хеллоу" к существующему слову "hello", проверяя наличие перевода и общее количество переводов.
Дублирование перевода	Проверяет, что при попытке добавить уже существующий перевод "привет" для слова "hello" выбрасывается исключение DuplicateTranslation.
Удаление перевода	Тестирует удаление перевода "хеллоу" из слова "hello", проверяя отсутствие перевода и корректное количество оставшихся переводов.
Удаление последнего перевода	Проверяет корректность удаления всех переводов слова "hello" (последних двух), а также автоматическое удаление слова из словаря, что подтверждается исключением NoFoundWord при поиске.
Удаление слова	Тестирует удаление слова "world" из словаря, проверяя, что слово больше не находится (выбрасывается исключение NoFoundWord).
Поиск переводов по слову	Проверяет корректность поиска и возврата всех переводов для слова "test", включая их количество и порядок.
Поиск слова по переводу	Тестирует поиск слова "test" по его переводу "тест", проверяя корректность возвращаемого значения.

Автоматический перевод английский	Проверяет работу функции автоматического перевода для слова "test", сравнивая результат с ожидаемой строкой "проверка; тест".

Результатом успешного прохождения будет **«[PASSED] :-D»** около каждого теста.

Исходный код

main.cpp

```
1 | #include "headers/Dict.h"
2 | #include "headers/Exceptions.h"
3 | #include <iostream>
4 | #include <windows.h>
5 |
6 | void runTests();
7 | const std::string DATAPATH = "../data/DictionaryT.json";
8 |
9 | int main() {
10|     SetConsoleOutputCP(CP_UTF8);
11|     SetConsoleCP(CP_UTF8);
12|
13|     try {
14|         std::cout << "\t\tЗапуск тестов\n" << std::endl;
15|         runTests();
16|         std::cout << "\n\t\tВсе тесты прошли!\n\n" << std::endl;
17|
18|         Dict dictionary(DATAPATH);
19|         dictionary.run();
20|     } catch (const std::exception& error) {
21|         std::cerr << "Error: " << error.what() << std::endl;
22|         system("pause");
23|         return EXIT_FAILURE;
24|     }
25|
26|     system("pause");
27|     return EXIT_SUCCESS;
28| }
29|
```

headers/Dict.h

```
1 | #ifndef DICT_H
2 | #define DICT_H
3 |
4 | #include <string>
5 | #include <list>
6 | #include <map>
7 |
8 | class Dict {
9 | public:
10|     explicit Dict(const std::string& filename);
11|     ~Dict();
12|
13|     std::string dataPath_;
14|
```

```
15|     static void insertSorted(std::list<std::string>& list, const
std::string & value);
16|     void insert(const std::string& word, const std::string&
translationsStr);
17|     void addTranslation(const std::string& word, std::string&
newTranslation);
18|     void removeTranslation(const std::string& word, const
std::string& translation);
19|     const std::list<std::string>& findTranslationByWord(const
std::string& word) const;
20|     std::string findWordByTranslation(const std::string&
translation) const;
21|     void remove(const std::string& word);
22|     void print() const;
23|     std::string autoTranslate(std::string& input) const;
24|
25|     static void printWelcome();
26|     void run();
27| private:
28|     std::map<std::string, std::list<std::string>> tree_;
29|
30|     void loadFromFile(const std::string& filename);
31|     void writeToFile() const;
32|     void removeFromFile(const std::string& word, const
std::string* translation = nullptr);
33|     static std::string getCommandInput();
34|
35|     static bool isEnglishWord(const std::string& word);
36|     static bool isRussianWord(const std::string& word);
37|     static void toUpperCaseE(std::string& str);
38|     static void toLowerCaseE(std::string& str);
39|     static void toLowerCaseR(std::string& str);
40|     static void trimSpaces(std::string& str);
41| };
42|
43| #endif // DICT_H
```

src/Dict.cpp

```
1 | #include <algorithm>
2 | #include <cctype>
3 | #include <fstream>
4 | #include <functional>
5 | #include <iostream>
6 | #include <iterator>
7 | #include <list>
8 | #include <map>
9 | #include <sstream>
10|
11| #include "../headers/Dict.h"
12| #include "../headers/Exceptions.h"
```

```
13| #include "../headers/MyJson.h"
14|
15| Dict::Dict(const std::string& filename) {
16|     loadFromFile(filename);
17|     dataPath_ = filename;
18| }
19|
20| Dict::~Dict() = default;
21|
22| void Dict::loadFromFile(const std::string& filename) {
23|     std::ifstream file(filename);
24|     if (!file.is_open()) {
25|         return;
26|     }
27|
28|     const std::string
content((std::istreambuf_iterator<char>(file)),
29|                          std::istreambuf_iterator<char>());
30|     file.close();
31|
32|     MyJson::parse(content, tree_);
33| }
34|
35| void Dict::writeToFile() const {
36|     std::ofstream fout(dataPath_);
37|     if (!fout) {
38|         throw FailOfMemoryAllocation("Dictionary.json file");
39|     }
40|     fout << MyJson::convertToString(tree_);
41| }
42|
43| void Dict::removeFromFile(const std::string& word, const
std::string* translation) {
44|     if (translation == nullptr) {
45|         tree_.erase(word);
46|     } else if (tree_.find(word) != tree_.end()) {
47|         auto& translations = tree_[word];
48|         translations.erase(std::remove(translations.begin(),
translations.end(), *translation),
49|                            translations.end());
50|         if (translations.empty()) {
51|             tree_.erase(word);
52|         }
53|     }
54|
55|     std::ofstream fout(dataPath_);
56|     if (!fout) {
57|         throw FailOfMemoryAllocation("Dictionary.json file");
58|     }
59|     fout << MyJson::convertToString(tree_);
```

```
60| }
61|
62| void Dict::insertSorted(std::list<std::string>& list, const
std::string& value) {
63|     const auto it = std::lower_bound(list.begin(), list.end(),
value);
64|     if (it == list.end() || *it != value)
65|         list.insert(it, value);
66| }
67|
68| void Dict::insert(const std::string& word, const std::string&
translationsStr) {
69|     if (word.empty())
70|         throw InvalidWordLanguage();
71|
72|     if (tree_.find(word) != tree_.end())
73|         throw DuplicateWord();
74|
75|     if (translationsStr.empty())
76|         throw EmptyValue();
77|
78|     std::list<std::string> translationsList;
79|     std::istringstream iss(translationsStr);
80|
81|     std::function<void()> processTranslations = [&] {
82|         std::string translation;
83|         if (!std::getline(iss, translation, ';'))
84|             return;
85|         trimSpaces(translation);
86|         if (!translation.empty()) {
87|             if (!isRussianWord(translation))
88|                 throw InvalidTranslationLanguage();
89|             toLowerCaseR(translation);
90|             insertSorted(translationsList, translation);
91|         }
92|         processTranslations();
93|     };
94|     processTranslations();
95|
96|     if (translationsList.empty())
97|         throw EmptyValue();
98|
99|     tree_[word] = translationsList;
100|     writeToFile();
101| }
102|
103| void Dict::addTranslation(const std::string& word, std::string&
newTranslation) {
104|     const auto it = tree_.find(word);
105|     if (it == tree_.end())
```

```
106|         throw NoFoundWord();
107|
108|     toLowerCaseR(newTranslation);
109|
110|     auto& translations = it->second;
111|     const auto pos = std::lower_bound(translations.begin(),
translations.end(), newTranslation);
112|     if (pos != translations.end() && *pos == newTranslation)
113|         throw DuplicateTranslation();
114|
115|     translations.insert(pos, newTranslation);
116|     writeToFile();
117| }
118|
119| void Dict::removeTranslation(const std::string& word, const
std::string& translation) {
120|     const auto it = tree_.find(word);
121|     if (it == tree_.end())
122|         throw NoFoundWord();
123|
124|     std::string lowerTranslation = translation;
125|     toLowerCaseR(lowerTranslation);
126|
127|     auto& translations = it->second;
128|     const auto pos = std::lower_bound(translations.begin(),
translations.end(), lowerTranslation);
129|
130|     if (pos == translations.end() || *pos != lowerTranslation)
131|         throw TranslationsNotFound();
132|
133|     translations.erase(pos);
134|
135|     if (translations.empty()) {
136|         tree_.erase(it);
137|         removeFromFile(word);
138|     } else
139|         removeFromFile(word, &lowerTranslation);
140| }
141|
142| const std::list<std::string>& Dict::findTranslationByWord(const
std::string& word) const {
143|     const auto it = tree_.find(word);
144|     if (it == tree_.end())
145|         throw NoFoundWord();
146|     return it->second;
147| }
148|
149| std::string Dict::findWordByTranslation(const std::string&
translation) const {
150|     std::string lowerTranslation = translation;
```

```
151|     toLowerCaseR(lowerTranslation);
152|
153|     const auto it = std::find_if(tree_.begin(), tree_.end(),
154|                                  [&lowerTranslation](const auto& pair)
{
155|                                      return
std::binary_search(pair.second.begin(), pair.second.end(),
lowerTranslation);
156|                                  });
157|     if (it != tree_.end())
158|         return it->first;
159|     throw NoFoundWord();
160| }
161|
162| void Dict::remove(const std::string& word) {
163|     if (word.empty())
164|         throw InvalidWordLanguage();
165|
166|     if (tree_.erase(word))
167|         removeFromFile(word);
168|     else
169|         throw NoFoundWord();
170| }
171|
172| void Dict::print() const {
173|     std::for_each(tree_.begin(), tree_.end(),
174|                   [](const auto& pair) {
175|                       std::cout << pair.first << " - ";
176|                       std::ostringstream oss;
177|                       std::copy(pair.second.begin(),
pair.second.end(),
178|                                 std::ostream_iterator<std::string
>(oss, "; "));
179|                       std::string result = oss.str();
180|                       if (!result.empty())
181|                           result.resize(result.size() - 2);
182|                       std::cout << result << std::endl;
183|                   });
184| }
185|
186| std::string Dict::autoTranslate(std::string& input) const {
187|     trimSpaces(input);
188|
189|     if (isEnglishWord(input)) {
190|         toLowerCaseE(input);
191|         const auto& translations = findTranslationByWord(input);
192|         std::ostringstream oss;
193|         std::copy(translations.begin(), translations.end(),
194|                   std::ostream_iterator<std::string>(oss, "; "));
195|         std::string result = oss.str();
```

```
196|         if (!result.empty())
197|             result.resize(result.size() - 2);
198|         return result;
199|     }
200|     toLowerCaseR(input);
201|     return findWordByTranslation(input);
202| }
203|
204| std::string Dict::getCommandInput() {
205|     std::cout << "\nВведите команду: ";
206|     std::string command;
207|     if (!std::getline(std::cin, command)) {
208|         std::cout << "\nЗавершение работы. Все данные сохранены в
файле\n";
209|         throw std::runtime_error("");
210|     }
211|     trimSpaces(command);
212|     toUpperCaseE(command);
213|     return command;
214| }
215|
216| void Dict::printWelcome() {
217|     std::cout << "\n\t\tАнгло-русский словарь\n\n";
218|     std::cout << "\nДоступные команды:\n"
219|               << "1. INSERT - Добавить новое слово и его переводы
в словарь.\n"
220|               << "2. REMOVE - Удалить слово из словаря.\n"
221|               << "3. ADDTRANSLATION - Добавить новый перевод для
существующего слова.\n"
222|               << "4. REMOVETRANSLATION - Удалить перевод для
существующего слова.\n"
223|               << "5. FINDTRANSLATION - Найти все переводы для
слова.\n"
224|               << "6. FINDWORD - Найти слово по его переводу.\n"
225|               << "7. AUTOTRANSLATE - Автоматический перевод
текста (определяется язык ввода).\n"
226|               << "8. PRINT - Вывести все слова и их переводы из
словаря.\n\n"
227|               << "Для выхода используйте Ctrl+Z (Windows) или
Ctrl+D (Linux)\n\n";
228| }
229|
230| void Dict::run()
231| {
232|     printWelcome();
233|     std::function<void()> runRecursive = [&]
234|     {
235|         try
236|         {
237|             const std::string command = getCommandInput();
```

```
238|             if (command.empty())
239|             {
240|                 runRecursive();
241|                 return;
242|             }
243|
244|             if (command == "INSERT")
245|             {
246|                 std::string word;
247|                 std::cout << "Введите английское слово: ";
248|                 if (!std::getline(std::cin, word))
249|                     return;
250|
251|                 trimSpaces(word);
252|                 toLowerCaseE(word);
253|                 if (!isEnglishWord(word))
254|                     throw InvalidWordLanguage();
255|
256|                 std::string translations;
257|                 std::cout << "Введите перевод(ы) через ';': ";
258|                 if (!std::getline(std::cin, translations))
259|                     return;
260|
261|                 insert(word, translations);
262|                 std::cout << "Добавлено: " << word << "\n";
263|             }
264|             else if (command == "REMOVE")
265|             {
266|                 std::string word;
267|                 std::cout << "Введите слово для удаления: ";
268|                 if (!std::getline(std::cin, word))
269|                     return;
270|
271|                 trimSpaces(word);
272|                 toLowerCaseE(word);
273|                 if (!isEnglishWord(word))
274|                     throw InvalidWordLanguage();
275|
276|                 remove(word);
277|                 std::cout << "Удалено успешно!\n";
278|             }
279|             else if (command == "ADDTRANSLATION")
280|             {
281|                 std::string word;
282|                 std::cout << "Введите слово: ";
283|                 if (!std::getline(std::cin, word))
284|                     return;
285|
286|                 trimSpaces(word);
287|                 toLowerCaseE(word);
```

```
288|                 if (!isEnglishWord(word))
289|                     throw InvalidWordLanguage();
290|                 if (tree_.find(word) == tree_.end())
291|                     throw NoFoundWord();
292|
293|                 std::string translation;
294|                 std::cout << "Введите новый перевод: ";
295|                 if (!std::getline(std::cin, translation))
296|                     return;
297|
298|                 trimSpaces(translation);
299|                 toLowerCaseR(translation);
300|                 if (!isRussianWord(translation))
301|                     throw InvalidTranslationLanguage();
302|
303|                 addTranslation(word, translation);
304|                 std::cout << "Добавлено успешно!\n";
305|             }
306|             else if (command == "REMOVETRANSLATION")
307|             {
308|                 std::string word;
309|                 std::cout << "Введите слово: ";
310|                 if (!std::getline(std::cin, word))
311|                     return;
312|
313|                 trimSpaces(word);
314|                 toLowerCaseE(word);
315|                 if (!isEnglishWord(word))
316|                     throw InvalidWordLanguage();
317|
318|                 std::string translation;
319|                 std::cout << "Введите перевод для удаления: ";
320|                 if (!std::getline(std::cin, translation))
321|                     return;
322|
323|                 trimSpaces(translation);
324|                 toLowerCaseR(translation);
325|                 if (!isRussianWord(translation))
326|                     throw InvalidTranslationLanguage();
327|
328|                 removeTranslation(word, translation);
329|                 std::cout << (tree_.find(word) == tree_.end() ?
"Перевод и слово удалены!\n" : "Перевод удалён!\n");
330|             }
331|             else if (command == "FINDTRANSLATION")
332|             {
333|                 std::string word;
334|                 std::cout << "Введите слово: ";
335|                 if (!std::getline(std::cin, word))
336|                     return;
```

```
337|
338|                 trimSpaces(word);
339|                 toLowerCaseE(word);
340|                 if (!isEnglishWord(word))
341|                     throw InvalidWordLanguage();
342|
343|                 const auto& translations =
findTranslationByWord(word);
344|                 std::cout << "Переводы: ";
345|                 std::function<void(std::list<std::string>::const_
iterator)>
346|                 printTranslations = [&](const
std::list<std::string>::const_iterator it)
347|                 {
348|                     if (it == translations.end())
349|                     {
350|                         std::cout << "\n";
351|                         return;
352|                     }
353|                     if (it != translations.begin())
354|                         std::cout << "; ";
355|                     std::cout << *it;
356|                     printTranslations(std::next(it));
357|                 };
358|                 printTranslations(translations.begin());
359|             }
360|             else if (command == "FINDWORD")
361|             {
362|                 std::string translation;
363|                 std::cout << "Введите перевод: ";
364|                 if (!std::getline(std::cin, translation))
365|                     return;
366|
367|                 trimSpaces(translation);
368|                 toLowerCaseR(translation);
369|                 if (!isRussianWord(translation))
370|                     throw InvalidTranslationLanguage();
371|
372|                 std::cout << "Найдено слово: " <<
findWordByTranslation(translation) << "\n";
373|             }
374|             else if (command == "AUTOTRANSLATE")
375|             {
376|                 std::string input;
377|                 std::cout << "Введите текст: ";
378|                 if (!std::getline(std::cin, input))
379|                     return;
380|
381|                 std::cout << "Результат: " <<
autoTranslate(input) << "\n";
```

```
382|             }
383|             else if (command == "PRINT")
384|                 print();
385|             else
386|                 std::cout << "Неизвестная команда. Попробуйте
снова.\n";
387|
388|             runRecursive();
389|         }
390|         catch (const std::logic_error& e)
391|         {
392|             std::cout << e.what() << "\n";
393|             runRecursive();
394|         }
395|         catch (const std::runtime_error& e)
396|         {
397|             std::cerr << e.what() << "\n";
398|         }
399|     };
400|     runRecursive();
401| }
402|
403| bool Dict::isEnglishWord(const std::string& word) {
404|     return std::all_of(word.begin(), word.end(),
405|                        [](const unsigned char c) {
406|                            return (c >= 'A' && c <= 'Z') ||
407|                                   (c >= 'a' && c <= 'z') ||
408|                                   c == '\'' || c == '-' ||
409|                                   c == ',' || c == ' ';
410|                        });
411| }
412|
413| bool Dict::isRussianWord(const std::string& word) {
414|     if (word.empty())
415|         return false;
416|
417|     std::function<bool(size_t)> checkRussian = [&](const size_t
i) -> bool {
418|         if (i >= word.size())
419|             return true;
420|         const unsigned char c = word[i];
421|
422|         if (c == ' ' || c == ',')
423|             return checkRussian(i + 1);
424|
425|         if ((c == 0xD0 || c == 0xD1) && i + 1 < word.size()) {
426|             const unsigned char next = word[i + 1];
427|             const bool isRussian = (c == 0xD0 && next >= 0x90 &&
next <= 0xBF) ||
428|                                    (c == 0xD1 && next >= 0x80 && next
```

```
<= 0x8F) ||
429|                                (c == 0xD0 && next == 0x81) ||
430|                                (c == 0xD1 && next == 0x91);
431|             return isRussian ? checkRussian(i + 2) : false;
432|         }
433|         return false;
434|     };
435|
436|     return checkRussian(0);
437| }
438|
439| void Dict::toUpperCaseE(std::string& str) {
440|     std::transform(str.begin(), str.end(), str.begin(),
441|                    [](const unsigned char c) { return
std::toupper(c); });
442| }
443|
444| void Dict::trimSpaces(std::string& str) {
445|     const auto first =
446|         std::find_if_not(str.begin(), str.end(), [](const int c)
{ return std::isspace(c); });
447|     const auto last =
448|         std::find_if_not(str.rbegin(), str.rend(), [](const int
c) { return std::isspace(c); }).base();
449|     str = first < last ? std::string(first, last) : "";
450| }
451|
452| void Dict::toLowerCaseE(std::string& str) {
453|     std::transform(str.begin(), str.end(), str.begin(),
454|                    [](const unsigned char c) { return
std::tolower(c); });
455| }
456|
457| void Dict::toLowerCaseR(std::string& str) {
458|     std::function<void(size_t)> processRussian = [&](const size_t
i) {
459|         if (i >= str.size())
460|             return;
461|         const unsigned char c = str[i];
462|
463|         if ((c == 0xD0 || c == 0xD1) && i + 1 < str.size()) {
464|             const unsigned char next = str[i + 1];
465|             if (c == 0xD0) {
466|                 if (next >= 0x90 && next <= 0x9F)
467|                     str[i + 1] = next + 0x20;
468|                 else if (next == 0x81) {
469|                     str[i] = 0xD1;
470|                     str[i + 1] = 0x91;
471|                 } else if (next >= 0xA0 && next <= 0xAF) {
472|                     str[i] = 0xD1;
```

```
473|                     str[i + 1] = next - 0x20;
474|                 }
475|             }
476|             processRussian(i + 2);
477|         } else
478|             processRussian(i + 1);
479|     };
480|
481|     processRussian(0);
482| }
```

headers/MyJson.h

```
1 | #ifndef MYJSON_H
2 | #define MYJSON_H
3 |
4 | #include <string>
5 | #include <vector>
6 | #include <list>
7 | #include <map>
8 |
9 | class MyJson {
10| public:
11|     using Translations = std::vector<std::string>;
12|     static void parse(const std::string& text,
std::map<std::string, std::list<std::string>>& dictionary);
13|     static std::string convertToString(const std::map<std::string,
std::list<std::string>>& dictionary);
14|
15| private:
16|     static void skipWhitespace(const std::string& text, size_t&
pos);
17|     static std::string parseString(const std::string& text,
size_t& pos);
18|     static Translations parseArray(const std::string& text,
size_t& pos);
19| };
20|
21| #endif // MYJSON_H
```

src/MyJson.cpp

```
1 | #include "../headers/MyJson.h"
2 | #include <algorithm>
3 | #include <sstream>
4 |
5 | void MyJson::skipWhitespace(const std::string& text, size_t& pos)
{
6 |     while (pos < text.size() && isspace(text[pos])) {
7 |         ++pos;
8 |     }
9 | }
```

```
10|
11| std::string MyJson::parseString(const std::string& text, size_t&
pos) {
12|     if (text[pos] != '"') return "";
13|
14|     std::string result;
15|     bool escape = false;
16|
17|     for (++pos; pos < text.size(); ++pos) {
18|         if (escape) {
19|             escape = false;
20|             result += text[pos];
21|         } else if (text[pos] == '\\') {
22|             escape = true;
23|         } else if (text[pos] == '"') {
24|             ++pos;
25|             break;
26|         } else {
27|             result += text[pos];
28|         }
29|     }
30|
31|     return result;
32| }
33|
34| MyJson::Translations MyJson::parseArray(const std::string& text,
size_t& pos) {
35|     Translations result;
36|
37|     skipWhitespace(text, pos);
38|     if (text[pos] != '[') return result;
39|     ++pos;
40|
41|     while (pos < text.size()) {
42|         skipWhitespace(text, pos);
43|         if (text[pos] == ']') {
44|             ++pos;
45|             break;
46|         }
47|
48|         std::string translation = parseString(text, pos);
49|         if (!translation.empty()) {
50|             result.push_back(std::move(translation));
51|         }
52|
53|         skipWhitespace(text, pos);
54|         if (text[pos] == ',') {
55|             ++pos;
56|         }
57|     }
```

```
58|
59|     return result;
60| }
61|
62| void MyJson::parse(const std::string& text, std::map<std::string,
std::list<std::string>>& dictionary) {
63|     size_t pos = 0;
64|
65|     skipWhitespace(text, pos);
66|     if (text[pos] != '{') return;
67|     ++pos;
68|
69|     while (pos < text.size()) {
70|         skipWhitespace(text, pos);
71|         if (text[pos] == '}') {
72|             ++pos;
73|             break;
74|         }
75|
76|         std::string key = parseString(text, pos);
77|         if (key.empty()) break;
78|
79|         skipWhitespace(text, pos);
80|         if (text[pos] != ':') break;
81|         ++pos;
82|
83|         Translations translations = parseArray(text, pos);
84|         if (!translations.empty()) {
85|             std::list<std::string>
translationsList(translations.begin(), translations.end());
86|             translationsList.sort();
87|             translationsList.unique();
88|             dictionary[std::move(key)] =
std::move(translationsList);
89|         }
90|
91|         skipWhitespace(text, pos);
92|         if (text[pos] == ',') {
93|             ++pos;
94|         }
95|     }
96| }
97|
98| std::string MyJson::convertToString(const std::map<std::string,
std::list<std::string>>& dictionary) {
99|     std::ostringstream oss;
100|      oss << "{";
101|
102|      bool first = true;
103|      for (const auto & it : dictionary) {
```

```
104|         if (!first) oss << ",";
105|         first = false;
106|
107|         oss << "\n  \"" << it.first << "\": [";
108|         bool firstTr = true;
109|         for (const auto& tr : it.second) {
110|             if (!firstTr) oss << ",";
111|             firstTr = false;
112|             oss << "\"" << tr << "\"";
113|         }
114|         oss << "]";
115|     }
116|
117|     oss << "\n}";
118|     return oss.str();
119| }
120|
```

headers/Exceptions.h

```
1 | #ifndef EXCEPTIONS_H
2 | #define EXCEPTIONS_H
3 |
4 | #include <stdexcept>
5 |
6 | class FailOfMemoryAllocation final : public std::runtime_error {
7 | public:
8 |     explicit FailOfMemoryAllocation(const std::string &
objectType)
9 |         : std::runtime_error("ERROR: Memory allocation failed for
" + objectType + "\n") {
10|     }
11| };
12|
13| class DuplicateTranslation final : public std::logic_error {
14| public:
15|     explicit DuplicateTranslation()
16|         : std::logic_error("WARNING: Translation already
exists!\n") {
17|     }
18| };
19|
20| class DuplicateWord final : public std::logic_error {
21| public:
22|     explicit DuplicateWord()
23|         : std::logic_error("WARNING: Word already exists!\n") {
24|     }
25| };
26|
27| class TranslationsNotFound final : public std::logic_error {
28| public:
```

```
29|     TranslationsNotFound() : std::logic_error("WARNING:
Translations not found\n") {
30|     }
31| };
32|
33| class NoFoundWord final : public std::logic_error {
34| public:
35|     explicit NoFoundWord()
36|         : std::logic_error("WARNING: Word not found\n") {
37|     }
38| };
39|
40| class EmptyValue final : public std::logic_error {
41| public:
42|     explicit EmptyValue()
43|         : std::logic_error("WARNING: Cannot insert an empty string
value.\n") {
44|     }
45| };
46|
47| class InvalidWordLanguage final : public std::logic_error {
48| public:
49|     InvalidWordLanguage() : std::logic_error("WARNING: Word should
be in English.\n") {
50|     }
51| };
52|
53| class InvalidTranslationLanguage final : public std::logic_error {
54| public:
55|     InvalidTranslationLanguage() : std::logic_error("WARNING:
Translation should be in Russian.\n") {
56|     }
57| };
58|
59| class ErrorInFile final : public std::logic_error {
60| public:
61|     ErrorInFile() : std::logic_error("ERROR: something went wrong
with file\n") {
62|     }
63| };
64|
65|
66| #endif //EXCEPTIONS_H
```

src/Tests.cpp

```
1 | #include "../headers/Dict.h"
2 | #include "../headers/Exceptions.h"
3 | #include <iostream>
4 | #include <fstream>
5 | #include <stdexcept>
```

```
6 | #include <algorithm>
7 |
8 | void runTests() {
9 |
10|     const std::string testFile = "temp.json";
11|     std::remove(testFile.c_str());
12|
13|     Dict dict(testFile);
14|
15|     try {
16|         dict.insert("hello", "привет;здравствуй");
17|         const auto& translations =
dict.findTranslationByWord("hello");
18|         if (translations.size() == 2 &&
19|             translations.front() == "здравствуй" &&
20|             translations.back() == "привет") {
21|             std::cout << "[PASSED] :-D Добавление в словарь:
Успешно добавлено слово 'hello'" << std::endl;
22|         } else {
23|             throw std::runtime_error("Incorrect translations for
'hello'");
24|         }
25|     } catch (const std::exception& e) {
26|         std::cout << "[FAILED] :-( Добавление в словарь: Ошибка: "
<< e.what() << std::endl;
27|     }
28|
29|     try {
30|         dict.insert("hello", "привет");
31|         std::cout << "[FAILED] :-( Проверка дубликата слова:
Ошибка: Ожидалось исключение DuplicateWord" << std::endl;
32|     } catch (const DuplicateWord&) {
33|         std::cout << "[PASSED] :-D Проверка дубликата слова:
Корректно выброшено исключение, тк существует 'hello'" << std::endl;
34|     } catch (const std::exception& e) {
35|         std::cout << "[FAILED] :-( Проверка дубликата слова:
Ошибка с неожиданным исключением: " << e.what() << std::endl;
36|     }
37|
38|     try {
39|         std::string newTranslation = "хеллоу";
40|         dict.addTranslation("hello", newTranslation);
41|         const auto& translations =
dict.findTranslationByWord("hello");
42|         if (translations.size() == 3 &&
43|             std::find(translations.begin(), translations.end(),
"хеллоу") != translations.end()) {
44|             std::cout << "[PASSED] :-D Добавление перевода:
Успешно добавлен перевод" << std::endl;
45|         } else {
```

```
46|             throw std::runtime_error("Translation 'хеллоу' not
found or incorrect size");
47|         }
48|     } catch (const std::exception& e) {
49|         std::cout << "[FAILED] :-( Добавление перевода: Ошибка: "
<< e.what() << std::endl;
50|     }
51|
52|     try {
53|         std::string dupTranslation = "привет";
54|         dict.addTranslation("hello", dupTranslation);
55|         std::cout << "[FAILED] :-( Проверка дубликата перевода:
Ошибка: Ожидалось исключение DuplicateTranslation" << std::endl;
56|     } catch (const DuplicateTranslation&) {
57|         std::cout << "[PASSED] :-D Проверка дубликата перевода:
Корректно выброшено исключение для существующего перевода" <<
std::endl;
58|     } catch (const std::exception& e) {
59|         std::cout << "[FAILED] :-( Проверка дубликата перевода:
Ошибка с неожиданным исключением: " << e.what() << std::endl;
60|     }
61|
62|     try {
63|         dict.removeTranslation("hello", "хеллоу");
64|         const auto& translations =
dict.findTranslationByWord("hello");
65|         if (translations.size() == 2 &&
66|             std::find(translations.begin(), translations.end(),
"хеллоу") == translations.end()) {
67|             std::cout << "[PASSED] :-D Удаление перевода: Успешно
удален перевод" << std::endl;
68|         } else {
69|             throw std::runtime_error("Translation 'хеллоу' still
present or incorrect size");
70|         }
71|     } catch (const std::exception& e) {
72|         std::cout << "[FAILED] :-( Удаление перевода: Ошибка: " <<
e.what() << std::endl;
73|     }
74|
75|     try {
76|         dict.removeTranslation("hello", "привет");
77|         dict.removeTranslation("hello", "здравствуй");
78|         try {
79|             dict.findTranslationByWord("hello");
80|             std::cout << "[FAILED] :-( Удаление последнего
перевода: Ошибка: Ожидалось исключение NoFoundWord" << std::endl;
81|         } catch (const NoFoundWord&) {
82|             std::cout << "[PASSED] :-D Удаление последнего
перевода: Корректно удалены последние переводы и слово" << std::endl;
```

```
83|         }
84|     } catch (const std::exception& e) {
85|         std::cout << "[FAILED] :-( Удаление последнего перевода:
Ошибка: " << e.what() << std::endl;
86|     }
87|
88|     try {
89|         dict.insert("world", "мир");
90|         dict.remove("world");
91|         try {
92|             dict.findTranslationByWord("world");
93|             std::cout << "[FAILED] :-( Удаление слова: Ошибка:
Ожидалось исключение NoFoundWord" << std::endl;
94|         } catch (const NoFoundWord&) {
95|             std::cout << "[PASSED] :-D Удаление слова: Успешно
удалено слово" << std::endl;
96|         }
97|     } catch (const std::exception& e) {
98|         std::cout << "[FAILED] :-( Удаление слова: Ошибка: " <<
e.what() << std::endl;
99|     }
100|
101|     try {
102|         dict.insert("test", "тест;проверка");
103|         const auto& translations =
dict.findTranslationByWord("test");
104|         if (translations.size() == 2 &&
105|             translations.front() == "проверка" &&
106|             translations.back() == "тест") {
107|             std::cout << "[PASSED] :-D Поиск переводов по слову:
Успешно найдены переводы" << std::endl;
108|         } else {
109|             throw std::runtime_error("Incorrect translations for
'test'");
110|         }
111|     } catch (const std::exception& e) {
112|         std::cout << "[FAILED] :-( Поиск переводов по слову:
Ошибка: " << e.what() << std::endl;
113|     }
114|
115|     try {
116|         const std::string translation = "тест";
117|         const std::string word =
dict.findWordByTranslation(translation);
118|         if (word == "test") {
119|             std::cout << "[PASSED] :-D Поиск слова по переводу:
Успешно найдено слово 'test' для перевода 'тест'" << std::endl;
120|         } else {
121|             throw std::runtime_error("Incorrect word for
'тест'");
```

```
122|         }
123|     } catch (const std::exception& e) {
124|         std::cout << "[FAILED] :-( Поиск слова по переводу:
Ошибка: " << e.what() << std::endl;
125|     }
126|
127|     try {
128|         std::string input = "test";
129|         const std::string result = dict.autoTranslate(input);
130|         if (result == "проверка; тест") {
131|             std::cout << "[PASSED] :-D Автоматический перевод
английский: Успешно переведено" << std::endl;
132|         } else {
133|             throw std::runtime_error("Incorrect translation for
'test'");
134|         }
135|     } catch (const std::exception& e) {
136|         std::cout << "[FAILED] :-( Автоматический перевод
английский: Ошибка: " << e.what() << std::endl;
137|     }
138|
139|     std::remove(testFile.c_str());
140| }
141|
```