Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и кибербезопасности
Высшая школа программной инженерии

КУРСОВАЯ РАБОТА

Алгоритмы работы со словарями

по дисциплине **«Алгоритмы и структуры данных»**

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

Руководитель Череповский Д.К.

« 08 » мая 2025г.

Санкт-Петербург
2025 г

Содержание

Введение. Общая постановка задачи

Тема: Алгоритмы работы со словарями

1. Для разрабатываемого словаря реализовать основные операции:

• INSERT (ключ, значение) – добавить запись с указанным ключом и значением

• SEARCH (ключ)- найти запись с указанным ключом

• DELETE (ключ)- удалить запись с указанным ключом

2. Предусмотреть обработку и инициализацию исключительных ситуаций, связанных, например, с проверкой значения полей перед инициализацией и присваиванием.

3. Программа должна быть написана в соответствии со стилем программирования: C++ Programming Style Guidelines (http://geosoft.no/development/cppstyle.html).

4. Тесты должны учитывать как допустимые, так и не допустимые последовательности входных данных.

Вариант 1.1.3. Англо-русский словарь. Красно-черное дерево.

Разработать и реализовать алгоритм работы с англо-русским словарем, реализованным как красно-черное дерево.

Узел дерева должен содержать:

- Ключ – английское слово
- Цвет узла
- Информационная часть – ссылка на список, содержащий переводы английского слова, отсортированные по алфавиту (переводов слова может быть несколько).

Основная часть работы

Описание алгоритма решения и используемых структур данных

Англо-русский словарь – тип словаря, при котором в качестве слова выступает строго английское слово, а в качестве перевода – список слов/предложений строго на русском языке.

Слово в данной работе – это слово естественного языка строго на английском языке тип std::string.

Переводы в данной работе – список предложений/словосочетаний строго на русском языке, хранящийся в односвязном списке.

Красно-чёрное дерево (КЧД, КЧ-дерево) — это самобалансирующееся двоичное дерево поиска, гарантирующее логарифмический рост высоты дерева от числа узлов. Отличительные свойства:

- Узел имеет цвет: красный или чёрный
- Корень всегда чёрный.
- Все листья (nullptr-узлы) чёрные.
- Красный узел не может иметь красного потомка (не может быть двух красных узлов подряд).
- На любом пути от узла до листа одинаковое количество чёрных узлов.

Благодаря этим ограничениям, путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего и дерево примерно сбалансировано. Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.

Балансировка в красно-чёрном дереве основана на строгих правилах, регулирующих цвет узлов и их расположение. При добавлении нового

элемента он всегда окрашивается в красный, чтобы не нарушить баланс чёрных узлов. Однако если родитель нового узла тоже красный, возникает нарушение — тогда запускается процедура корректировки с перекраской и, при необходимости, поворотами.

Перекраска применяется, если дядя нового узла тоже красный: тогда и родитель, и дядя становятся чёрными, а дедушка — красным. Этот процесс может рекурсивно подниматься вверх по дереву. Если же дядя чёрный, выполняется поворот (влево или вправо в зависимости от конфигурации), чтобы вернуть дерево к допустимому виду. Повороты меняют местами узлы, сохраняя порядок, но восстанавливая правила.

При удалении узла балансировка сложнее. Если удаляется чёрный узел, может нарушиться правило равенства числа чёрных узлов на всех путях. Тогда дерево проводит серию операций: перекраску и повороты, распространяя «недостающий чёрный» вверх, пока не будет восстановлен баланс. Эти механизмы делают структуру устойчивой к нарушениям и гарантируют логарифмическую высоту.

Сложность алгоритма составляет:

	В среднем	В лучшем случае
Память	O(n)	O(n)
Поиск	O(log*n)	O(log*n)
Вставка	O(log*n)	O(log*n)
Удаление	O(log*n)	O(log*n)

Для корректной работы и соблюдения условий задания дополнительно были реализованы следующие структуры:

Перечисление (enum Color) – тип данных, предназначенный для задания констант, в данном случае цветов узла дерева.

В качестве данных, которые хранятся в дереве выступает структура - пара (Pair):

- Используется для хранения пары слово-переводы.
- Поля:
  1. std::string word_ - хранит слово на английском языке.
  2. List translations_ - список переводов слова на русском языке
- Перегружены следующие методы:
  1. Операторы сравнения: больше, меньше и равенства. Лексикографически сравнивают строки (слова)
  2. Оператор вывода данных.

Так как переводов у слова может быть несколько, то они хранятся в структуре односвязный список (List):

- Используется для упорядоченного хранения (в алфавитном порядке) переводов слов.
- Каждый элемент представлен структурой NodeList, имеющей поля:
  1. std::string value_ - в данной реализации – перевод слова (строка или фраза)
  2. NodeList* next_ - указатель на следующий элемент
- Поля List:
  1. NodeList* head_ - указатель на голову (начало) списка
  2. NodeList* tail_ - указатель на хвост (конец) списка
  3. size_t size_ - количество элементов в списке (в данном случае переводов)
- Реализованы следующие методы:
  1. NodeList* getHead() – получение указателя на начало списка
  2. size_t getSize() – количества переводов
  3. bool isEmpty() – проверка пустой ли список

4. void push() – метод для вставки элемента в список и упорядочивания его по алфавиту

5. void remove () – удаление перевода слова

6. std::string convertTranslationsToString() – метод превращающий список переводов в строку вида «слово – перевод1; переводX»

7. void clear() – приватный метод для отчистки списка при вызове деструктора

Красно-чёрное дерево

- Хранит слова-переводы
- Каждый элемент представлен структурой Node, реализованы поля:

1. Pair data_ - хранит пару слово-переводы
2. Color color_ - хранит цвет узла
3. Указатели Node* на левого, правого потомка и на родителя

- Node* root_ - поле хранит указатель на корень дерева
- Реализованы следующие методы:

1. void rotateLeft() – выполняет левый поворот вокруг узла для балансировки дерева
2. void rotateRight() - выполняет правый поворот вокруг узла для балансировки дерева
3. void fixInsert() – метод для балансировки дерева после вставки
4. void fixRemove() - метод для балансировки дерева после удаления
5. void clear() – метод для очистки дерева при вызове деструктора
6. Node* search() – метод для поиска элемента в дереве
7. Color getColor() – метод для получения цвета узла
8. void setColor() - метод для изменения цвета узла
9. Node* getRoot() – метод для получения указателя на узел дерева
10. void insert() – метод вставки элемента в дерево

11. void transplant() – вспомогательный метод для замены одного узла на другой в дереве

12. Node* findMaximum() – вспомогательный метод для поиска минимального элемента в лево поддереве

13. void remove() – метод удаления элемента в дерево

14. std::string makeTreeToString() – возвращает строку-строковое представление структуры дерева

15. void makeTreeToStringRecursive() – метод формирует рекурсивно строковое представление дерева

16. void loadFromFile() – метод для автоматической вставки слов-переводов из JSON файла

С целью отделения логики работы с данными от взаимодействия с пользователем был реализован класс Dict, обеспечивающий удобный доступ к функциональности красно-чёрного дерева и позволяющий интегрировать его в другие части программы. Dict:

- Хранит указатель на дерево
- Реализованы методы:

1. void printRecursive() — рекурсивно выводит данные.
2. void writeToFile() — сохраняет новую пару в файл.
3. void removeFromFile() — удаляет строку из файла.
4. RBT getTree() — возвращает копию дерева.
5. void insert() — добавляет новую пару с запросом ввода от пользователя.
6. void remove() — удаляет слово и все его переводы из словаря и файла.
7. void addTranslation() — добавляет перевод к существующему слову.
8. void removeTranslation() — удаляет один перевод у слова.

9. List& findTranslationByWord()— находит переводы по заданному английскому слову.

10. void printWordWithTranslations() — выводит слово и все его переводы.

11. std::string findWordByTranslation() — ищет английское слово по русскому переводу.

12. Node* findInTreeByTranslation() —ищет слово по переводу.

13. std::string autotranslate() — автоматически определяет язык ввода и переводит слово.

14. void print() — выводит дерево с переводами («слово – переводы»).

15. void run(); — запускает интерфейс команд для взаимодействия с пользователем.

16.

Для минимизации ошибок и упрощения работы с пользователем были введены следующие функции, которые приводят команду, слово и переводы в необходимый вид:

1. bool isEnglishWord() — проверяет, является ли слово английским.
2. bool isRussianWord() — проверяет, является ли слово русским.
3. void toUpperCaseE() — переводит английское слово в верхний регистр.
4. void trimSpaces() — удаляет лишние пробелы в строке.
5. void toLowerCaseE() — переводит английское слово в нижний регистр.
6. void toLowerCaseR() — переводит русское слово в нижний регистр.

Также для работы с файлом был написан класс MyJson необходимый для парсинга и чтения данных из файла .json формата. Приводить его в отчёте излишне. Краткое описание находится в Приложении 3.

Анализ алгоритма

Чтобы определить сильные и слабые стороны написанной программы, проведем анализ эффективности выполнения трёх основных операций: вставки, поиска и удаления элементов.

Для односвязного списка сложность операций:

- Поиск (search): операция выполняется за время O(n), так как поиск элемента требует последовательного просмотра каждого узла, начиная с головы, до нахождения нужного значения или достижения конца списка. Нет возможности перейти сразу к середине или к нужному элементу.
- Вставка (insert): O(n), чтобы вставить элемент в алфавитном порядке, нужно пройти список до нужной позиции, что занимает O(n).
- Удаление (remove):. O(n), чтобы удалить узел, нужно сначала его найти, затем переназначить указатель предыдущего элемента.

Для КЧД сложность операций:

- Поиск (search): операция выполняется за время O(log*n), поскольку красно-чёрное дерево сбалансировано и его высота ограничена O(log*n). Поиск сравнивает ключи по пути от корня до листа.
- Вставка (insert): также O(log*n) – требуется найти место вставки (что занимает O(log*n)), затем может потребоваться до O(log*n) операций вращения и перекраски для восстановления баланса (число таких операций ограничено глубиной дерева).
- Удаление (remove): аналогично работает за O(log*n) – поиск узла на удаление стоит O(log*n) а восстановление свойств дерева (fixDelete) тоже выполняется за время, пропорциональное высоте дерева.

В целом, красно-чёрное дерево обеспечивает гарантированное время выполнения основных операций O(log*n) в худшем случае.

Описание спецификации программы (детальные требования)

В программе используется следующая логика ошибок:

- Если сообщение начинается с “WARNING”, то это не критическая ошибка и работа продолжается
- Если сообщение начинается с “ERROR”, то ошибка критическая (файл поврежден или ошибка с памятью) и работа программы завершается.

Требования:

1. При запуске программы пользователю выводится сообщение с возможными командами и предлагается 3 варианта:
   1.1 **“USER”** – запуск интерфейса для работы с пользователем (при запуске работы с пользователем в словаре уже находится >5000 слов).
   1.2 **“TEST”** – запуск тестов, в консоль отображается только результат прохождения (PASS или FAIL). После тестов программа завершается.
   1.3 **“EXIT”** – завершение работы.
2. При выборе пользовательского интерфейса пользователю будет доступно на выбор 10 команд для их понимания описание каждой будет выведено на экран.
   2.1 “**INSERT**” – добавление нового слова с минимум одним переводом
      2.1.1 Выводится приглашение для ввода
         2.1.1.1 При отсутствии ввода выводится сообщение об ошибке.
         2.1.1.2 При введении слова не на английском языке выводится сообщение об ошибке
      2.1.2 При вводе корректного слова (здесь и далее корректное слово – слово на английском языке в любом регистре) выводится приглашение для ввода переводов для слова. Если пользователь хочет ввести несколько переводов их необходимо вводить через точку с запятой.

2.1.2.1 При отсутствии ввода выводится сообщение об ошибке

2.1.2.2 При введении слова не на русском языке выводится сообщение об ошибке

2.1.3 При корректном вводе и переводов (здесь и далее корректные перевод/переводы – слово/фраза/предложение на русском языке в любом регистре) и слова выводится сообщение об успешном добавлении

2.1.4 Если слово или перевод (один из переводов) уже есть в словаре – выводится сообщение об ошибке

2.2 “**REMOVE**” – удаление слова и всех его переводов из словаря

2.2.1 Выводится приглашение для ввода

2.2.1.1 При отсутствии ввода выводится сообщение об ошибке.

2.2.1.2 При введении слова не на английском языке выводится сообщение об ошибке

2.2.1.3 При введении слова, которого нет в словаре выводится сообщение об ошибке

2.2.2 Если введено корректное слово, то выводится сообщение об успешном удалении

2.3 “**ADDTRANSLATION” –** добавление перевода к существующему слову

2.3.1 Выводится приглашение для ввода

2.3.1.1 При отсутствии ввода выводится сообщение об ошибке.

2.3.1.2 При введении слова не на английском языке выводится сообщение об ошибке

2.3.2 При вводе корректного слова выводится приглашение для ввода перевода для слова.

2.3.2.1 При отсутствии ввода выводится сообщение об ошибке

2.3.2.2 При введении слова не на русском языке выводится сообщение об ошибке

2.3.3 При корректном вводе и переводов и слова выводится сообщение об успешном добавлении

2.3.4 Если слово или перевод уже есть в словаре – выводится сообщение об ошибке

2.4 “**REMOVETRANSLATION” –** удаление перевода

2.4.1 Выводится приглашение для ввода

2.4.1.1 При отсутствии ввода выводится сообщение об ошибке.

2.4.1.2 При введении слова не на английском языке выводится сообщение об ошибке

2.4.2 При вводе корректного слова выводится приглашение для ввода перевода для слова.

2.4.2.1 При отсутствии ввода выводится сообщение об ошибке

2.4.2.2 При введении слова не на русском языке выводится сообщение об ошибке

2.4.3 При корректном вводе и переводов и слова выводится сообщение об удалении добавлении. Если у слова был 1 перевод, то слово полностью удаляется и выводится об этом сообщение.

2.4.4 Если слова или перевода нет в словаре – выводится сообщение об ошибке

2.5 “**FINDTRANSLATION**” – поиск переводов по слову

2.5.1 Выводится приглашение для ввода

2.5.1.1 При отсутствии ввода выводится сообщение об ошибке.

2.5.1.2 При введении слова не на английском языке выводится сообщение об ошибке

2.5.2 При вводе корректного слова выводятся все переводы

2.5.2.1 Если слова или перевода нет в словаре – выводится сообщение об ошибке

2.6 “**FINDWORD**” – поиск слова по переводу

2.6.1 Выводится приглашение для ввода

2.6.1.1 При отсутствии ввода выводится сообщение об ошибке.

2.6.1.2 При введении перевода не на русском языке выводится сообщение об ошибке

2.6.2 При вводе корректного перевода выводится слово

2.6.2.1 Если слова или перевода нет в словаре – выводится сообщение об ошибке

2.7 “**AUTOTRANSLATE” –** вывод либо слова, либо перевода в зависимости от языка

2.7.1 Выводится приглашение для ввода

2.7.1.1 При отсутствии ввода выводится сообщение об ошибке.

2.7.2 При вводе корректного слова или перевода выводятся переводы или слово, соответственно

2.7.2.1 Если слова или перевода нет в словаре – выводится сообщение об ошибке

2.8 “**PRINT**” – вывод данных в словаре в виде «слово – переводы»

2.9 “**TREE**” – вывод данных в словаре с соблюдением структуры и цвета дерева в виде ASCII-графики.

2.10 **“EXIT” –** завершение работы

Описание программы (структура программы, форматы входных и выходных данных)

Структуру программы можно разделить так:

- ***main.cpp*** – точка входа в программу
- ***Structures.h и Structures.cpp*** – вспомогательные структуры
- ***RBT.h и RBT.cpp*** – красно-чёрное дерево
- ***Dict.h и Dict.cpp*** – оболочка для пользовательской работы со словарём
- ***MyJson.h и MyJson.cpp*** – реализация для работы с файлами ***json***, необходимыми для хранения слов-переводов
- ***Tests.h и Tests.cpp*** – тестирование

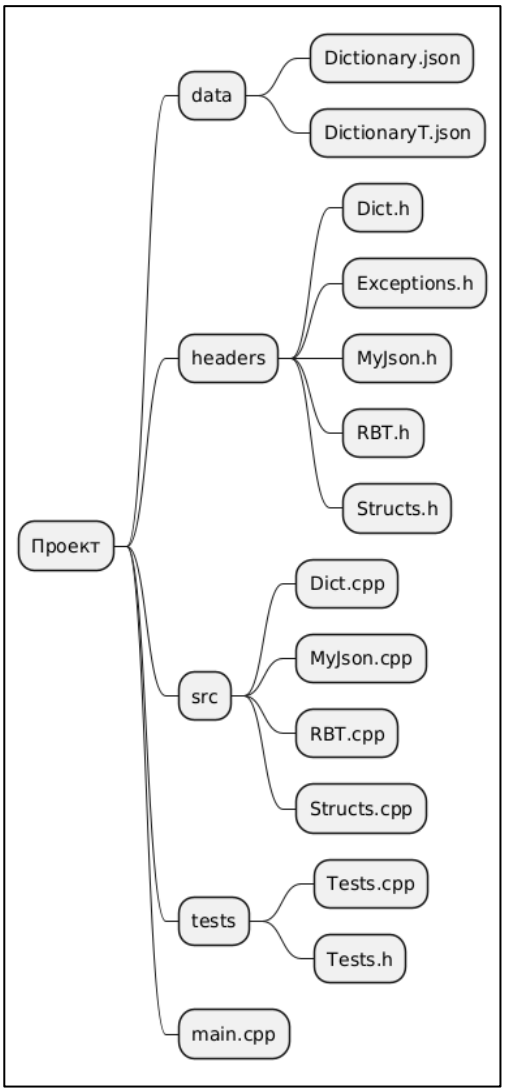

Рисунок 1. Файловая структура проекта

Проблема использования и кодировки русского языка

- ***SetConsoleOutputCP(CP_UTF8)*** устанавливает кодировку вывода консоли в ***UTF-8***, что позволяет корректно отображать Unicode-символы, включая кириллицу.

- ***SetConsoleCP(CP_UTF8)*** настраивает кодировку ввода консоли на ***UTF-8***, обеспечивая правильное считывание символов, введённых пользователем.

Входные данные:

- Файл json, в котором хранятся слова и их переводы, для корректной работы словаря
- Пользовательский ввод команд, слов и переводов (std::string)

Выходные данные:

- Информация о возникшей ошибки при тестировании для некорректных случаев (строка типа std::string)
- Информация о данных в словаре (строка типа std::string)

Ключевые функции и их взаимодействие

- Добавление перевода **(Dict::addTranslation(word, trans)):** вызывается пользователем или другим кодом. Dict выполняет **RBTree::search(word).** Если узел с таким словом найден, вызывается **List::insertSorted(trans)** для добавления перевода; иначе создаётся новый объект **Pair(word),** в него добавляется перевод через List, и затем **RBTree::insert()** вставляет новый узел.

- Удаление слова **(Dict::deleteWord(word)):** Dict вызывает **RBTree::remove(word).** Внутри выполняется удаление узла и восстановление баланса через fixDelete.

- Удаление перевода **(Dict::deleteTranslation(word, trans)):** Dict сначала ищет узел через **RBTree::search(word).** Если найден, из списка **Pair.translations_** вызывается **remove(trans).** После удаления проверяется пустота списка: если списки переводов больше нет, удаляется и сам узел **(RBTree::remove).**

- Поиск переводов **(Dict::find(word)):** вызывает **RBTree::search(word).** Если узел найден, возвращает translations_; иначе сообщает об отсутствии слова.

- Загрузка из файла **(Dict::loadFromFile(filename)):** метод читает JSON-файл перебирает пары «слово – массив переводов». Для каждой такой пары создаётся Pair с загруженными переводами и вызывается **RBTree::insert(pair),** восстанавливая дерево.

- Сохранение в файл **(Dict::saveToFile(filename)):** выполняется симметричный обход всего дерева При посещении каждого узла берутся word_ и все translations_, формируется JSON-структура (объект вида слово: [переводы]). После обхода эта JSON-структура записывается в файл.

- Вывод словаря **(Dict::printAll()):** по аналогии с сохранением, выполняется симметричный обход дерева и вывод каждой пары в текстовом виде.

Заключение

В рамках курса «Алгоритмы и структуры данных» мной были получены фундаментальные теоретические знания и практические навыки, необходимые для реализации сложных структур данных. В ходе курсового проекта мной был разработан англо-русский словарь на основе красно-чёрного дерева, в котором каждый узел хранит английское слово и список его переводов, реализованный с помощью собственного двусвязного списка. Все структуры данных — включая дерево, список, строку и файловый ввод-вывод — были реализованы без использования стандартной библиотеки STL.

Работа включала в себя глубокое изучение принципов самобалансирующихся деревьев, алгоритмов вставки, удаления и поиска с сохранением баланса. Были предусмотрены и тщательно протестированы исключительные ситуации, что обеспечило надёжность программы.

Подводя итог, можно сказать, что красно-чёрное дерево доказало свою эффективность как основа для ассоциативного контейнера с быстрым поиском и упорядоченным хранением данных. Поставленная задача была выполнена полностью, проект прошёл отладку и тестирование, а также готов к реальному использованию в качестве словарного приложения.

Список использованной литературы

1. Бьярне Страуструп. Программирование: принципы и практика использования C++, исправленное издание = Programming: Principles and Practice Using C++. — М.: «Вильямс», 2011. — С. 1248.

2. Алгоритмы: построение и анализ: / Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. - М.: Вильямс, 2011.- 1296 с.

3. Балансировка красно-чёрных деревьев — Три случая // Хабр URL: https://habr.com/ru/companies/otus/articles/472040/ (дата обращения: 02.05.2025).

4. Понимаем красно-черное дерево. Часть 2. Балансировка и вставка // Хабр URL: https://habr.com/ru/articles/557328/ (дата обращения: 02.05.2025).

Приложения

Приложение 1. Текст программы

main.cpp

```
#include "headers/Dict.h"
#include "headers/Exceptions.h"
#include "tests/Tests.h"
#include <iostream>

void runProgram();
void runForUser();
void runForTests();

int main() {
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);

    runProgram();

    return EXIT_SUCCESS;
}

void runProgram() {
    std::cout << "\n\t\tАнгло-русский словарь. Красно-чёрное дерево\n";
    std::cout << "\nДоступные команды:\n"
              << "1. TEST - Для запуска автоматического тестирования программы.\n"
              << "2. USER- Добавить новое слово и его переводы в словарь.\n"
              << "3. EXIT - Завершение работы.\n";
    std::cout << "Ваш выбор: ";

    while (true) {
        std::string command;
        std::getline(std::cin, command);
        trimSpaces(command);
        toUpperCaseE(command);

        if (command == "TEST") {
            try {
                DATAPATH = R"(..\data\DictionaryT.json)";
                runForTests();
            } catch (const std::exception& e) {
                std::cerr << e.what() << '\n';
                exit(EXIT_FAILURE);
            }
        } else if (command == "USER"){
            try {
                DATAPATH = R"(..\data\DictionaryT.json)";
                runForUser();
            } catch (const std::exception& e) {
                std::cerr << e.what() << '\n';
                exit(EXIT_FAILURE);
            }
        } else if (command == "EXIT")
            exit(EXIT_SUCCESS);
        else {
            std::cout << "Неизвестная команда, попробуйте ещё раз\n";
```

```
53|             std::cout << "Ваш выбор:   ";
54|         }
55|     }
56| }
57|
58| void runForUser() {
59|     Dict dictionary;
60|     dictionary.run();
61| }
62|
63| void runForTests() {
64|     runAllTests();
65|     exit(EXIT_SUCCESS);
66| }
67|
```

Exceptions.h

```
1 | #ifndef EXCEPTIONS_H
2 | #define EXCEPTIONS_H
3 |
4 | #include <stdexcept>
5 |
6 | class NullPointerDeletion final : public std::logic_error {
7 | public:
8 |     NullPointerDeletion() : std::logic_error("WARNING: List Attempted to delete
null pointer\n") {
9 |     }
10| };
11|
12| class FailOfMemoryAllocation final : public std::runtime_error {
13| public:
14|     explicit FailOfMemoryAllocation(const std::string & objectType)
15|         : std::runtime_error("ERROR: Memory allocation failed for " + objectType +
"\n") {
16|     }
17| };
18|
19| class FailOfObjectDeletion final : public std::runtime_error {
20| public:
21|     explicit FailOfObjectDeletion(const std::string & objectType)
22|         : std::runtime_error("ERROR: Deletion failed for object of type: " +
objectType + "\n") {
23|     }
24| };
25|
26| class DuplicateTranslation final : public std::logic_error {
27| public:
28|     explicit DuplicateTranslation()
29|         : std::logic_error("WARNING: Translation already exists!\n") {
30|     }
31| };
32|
33| class NoTranslations final : public std::logic_error {
34| public:
35|     explicit NoTranslations()
36|         : std::logic_error("WARNING: There isn`t any translations!\n") {
37|     }
38| };
39|
40| class DuplicateWord final : public std::logic_error {
41| public:
```

```
42|     explicit DuplicateWord()
43|         : std::logic_error("WARNING: Word already exists!\n") {
44|     }
45| };
46|
47| class InvalidListState final : public std::runtime_error {
48| public:
49|     explicit InvalidListState()
50|         : std::runtime_error("ERROR: List Invalid list state\n") {
51|     }
52| };
53|
54| class NodeNotFoundInList final : public std::logic_error {
55| public:
56|     NodeNotFoundInList() : std::logic_error("WARNING: List Node not found in
list\n") {
57|     }
58| };
59|
60| class EmptyValueInNode final : public std::logic_error {
61| public:
62|     EmptyValueInNode() : std::logic_error("WARNING: List Node contains an empty
value\n") {
63|     }
64| };
65|
66| class WrongColor final : public std::runtime_error {
67| public:
68|     WrongColor() : std::runtime_error("ERROR: Invalid color value specified\n") {
69|     }
70| };
71|
72| class RotationErrorNullptr final : public std::runtime_error {
73| public:
74|     explicit RotationErrorNullptr(const std::string & side)
75|         : std::runtime_error("ERROR:" + side + "Cannot rotate around a null
node.\n") {
76|     }
77| };
78|
79| class LeftRotationError final : public std::runtime_error {
80| public:
81|     explicit LeftRotationError()
82|         : std::runtime_error("ERROR: Cannot rotate left; the node's right child is
null.\n") {
83|     }
84| };
85|
86| class RightRotationError final : public std::runtime_error {
87| public:
88|     explicit RightRotationError()
89|         : std::runtime_error("ERROR: Cannot rotate right; the node's left child is
null.\n") {
90|     }
91| };
92|
93| class EmptyValue final : public std::logic_error {
94| public:
95|     explicit EmptyValue()
96|         : std::logic_error("WARNING: Cannot insert an empty string value.\n") {
97|     }
98| };
```

```
99|
100| class NullInsertionNode final : public std::logic_error {
101| public:
102|     NullInsertionNode()
103|         : std::logic_error("WARNING: Initial node for fixup cannot be null.\n") {
104|     }
105| };
106|
107| class MissingGrandparent final : public std::runtime_error {
108| public:
109|     MissingGrandparent()
110|         : std::runtime_error("ERROR: RED parent node has no grandparent.\n") {
111|     }
112| };
113|
114| class NodeNotFound final : public std::logic_error {
115| public:
116|     explicit NodeNotFound()
117|         : std::logic_error("WARNING: Word not found in the tree\n") {
118|     }
119| };
120|
121| class CorruptedTreeException final : public std::runtime_error {
122| public:
123|     CorruptedTreeException() : std::runtime_error("ERROR: Corrupted tree
structure.\n") {
124|     }
125| };
126|
127| class InvalidColorException final : public std::runtime_error {
128| public:
129|     InvalidColorException() : std::runtime_error("ERROR: Invalid color state.\n")
{
130|     }
131| };
132|
133| class RotationException final : public std::runtime_error {
134| public:
135|     RotationException() : std::runtime_error("ERROR: Rotation error in tree.\n")
{
136|     }
137| };
138|
139| class InvalidWordLanguage final : public std::logic_error {
140| public:
141|     InvalidWordLanguage() : std::logic_error("WARNING: Word should be in
English.\n") {
142|     }
143| };
144|
145| class InvalidTranslationLanguage final : public std::logic_error {
146| public:
147|     InvalidTranslationLanguage() : std::logic_error("WARNING: Translation should
be in Russian.\n") {
148|     }
149| };
150|
151| class NoFoundWord final : public std::logic_error {
152| public:
153|     explicit NoFoundWord()
154|         : std::logic_error("WARNING: Word not found in dictionary.\n") {
155|     }
```

```
156| };
157| #endif //EXCEPTIONS_H
158|
```

Structs.h

```
1 | #ifndef STRUCTS_H
2 | #define STRUCTS_H
3 |
4 | #include <string>
5 |
6 | struct NodeList {
7 |     std::string value_;
8 |     NodeList* next_;
9 |
10|     explicit NodeList(const std::string& value);
11| };
12|
13| class List {
14| private:
15|     NodeList* head_;
16|     NodeList* tail_;
17|     size_t size_;
18|
19|     void clear();
20|
21| public:
22|     List();
23|     ~List();
24|     List& operator=(const List& other);
25|     List(const List& other);
26|
27|     NodeList* getHead() const;
28|     void push(const std::string& value);
29|
30|     void remove(NodeList* nodeToRemove);
31|
32|     std::string convertTranslationsToString() const;
33|     bool isEmpty() const;
34|     size_t getSize() const;
35| };
36|
37| struct Pair {
38|     std::string word_;
39|     List translation_;
40|
41|     Pair();
42|     Pair(const std::string& eng, const List& rus);
43|
44|     bool operator<(const Pair& other) const;
45|     bool operator>(const Pair& other) const;
46|     bool operator==(const Pair& other) const;
47|     friend std::ostream& operator<<(std::ostream& output, const Pair& pair);
48| };
49|
50| #endif // STRUCTS_H
51|
```

Structs.cpp

```
1 |
2 | #include "../headers/Structs.h"
3 | #include "../headers/Exceptions.h"
```

```
4 |
5 | #include <sstream>
6 | #include <string>
7 | #include <iostream>
8 |
9 | NodeList::NodeList(const std::string& value) {
10|     try {
11|         value_ = value;
12|     } catch (const std::bad_alloc&) {
13|         throw FailOfMemoryAllocation("NodeList");
14|     }
15|     next_ = nullptr;
16| }
17|
18| void List::clear() {
19|     while (head_ != nullptr) {
20|         NodeList* temp = head_;
21|         head_ = head_->next_;
22|         delete temp;
23|     }
24|     tail_ = nullptr;
25|     size_ = 0;
26| }
27|
28| List::List() {
29|     head_ = nullptr;
30|     tail_ = nullptr;
31|     size_ = 0;
32| }
33|
34| List::~List() {
35|     clear();
36| }
37|
38| List& List::operator=(const List& other) {
39|     if (this != &other) {
40|         clear();
41|
42|         const NodeList* current = other.head_;
43|         while (current) {
44|             push(current->value_);
45|             current = current->next_;
46|         }
47|     }
48|     return *this;
49| }
50|
51| List::List(const List& other) : head_(nullptr), tail_(nullptr), size_(0) {
52|     const NodeList* current = other.head_;
53|     while (current != nullptr) {
54|         push(current->value_);
55|         current = current->next_;
56|     }
57| }
58|
59| NodeList* List::getHead() const {
60|     return head_;
61| }
62|
63| void List::push(const std::string & value) {
64|     NodeList* check = head_;
65|     while (check != nullptr) {
```

```
66|         if (check->value_ == value) {
67|             throw DuplicateTranslation();
68|         }
69|         check = check->next_;
70|     }
71|
72|     NodeList* newNode = nullptr;
73|     try {
74|         newNode = new NodeList(value);
75|     } catch (const std::bad_alloc &) {
76|         throw FailOfMemoryAllocation("NodeList");
77|     }
78|
79|     if (head_ == nullptr) {
80|         head_ = tail_ = newNode;
81|     }
82|     else if (value < head_->value_) {
83|         newNode->next_ = head_;
84|         head_ = newNode;
85|     }
86|     else if (value > tail_->value_) {
87|         tail_->next_ = newNode;
88|         tail_ = newNode;
89|     }
90|     else {
91|         NodeList* current = head_;
92|         while (current->next_ != nullptr && current->next_->value_ < value) {
93|             current = current->next_;
94|         }
95|         newNode->next_ = current->next_;
96|         current->next_ = newNode;
97|
98|         if (newNode->next_ == nullptr) {
99|             tail_ = newNode;
100|         }
101|     }
102|
103|     ++size_;
104| }
105|
106| void List::remove(NodeList* nodeToRemove) {
107|     if (nodeToRemove == nullptr || head_ == nullptr) {
108|         throw NullPointerDeletion();
109|     }
110|
111|     NodeList* current = head_;
112|     NodeList* prev = nullptr;
113|
114|     while (current != nullptr && current != nodeToRemove) {
115|         prev = current;
116|         current = current->next_;
117|     }
118|
119|     if (current == nullptr) {
120|         throw NodeNotFoundInList();
121|     }
122|
123|     if (prev == nullptr) {
124|         head_ = nodeToRemove->next_;
125|         if (head_ == nullptr) {
126|             tail_ = nullptr;
127|         }
```

```
128|     }
129|     else {
130|         prev->next_ = nodeToRemove->next_;
131|         if (nodeToRemove == tail_) {
132|             tail_ = prev;
133|         }
134|     }
135|
136|     delete nodeToRemove;
137|     --size_;
138| }
139|
140| std::string List::convertTranslationsToString() const {
141|     std::string result;
142|     const NodeList* current = head_;
143|
144|     while (current) {
145|         if (current->value_.empty()) {
146|             throw EmptyValueInNode();
147|         }
148|
149|         result += current->value_;
150|         if (current->next_) {
151|             result += "; ";
152|         }
153|         current = current->next_;
154|     }
155|
156|     return result;
157| }
158|
159| bool List::isEmpty() const {
160|     return head_ == nullptr;
161| }
162|
163| size_t List::getSize() const {
164|     return size_;
165| }
166|
167| Pair::Pair() = default;
168|
169| Pair::Pair(const std::string& eng, const List& rus) {
170|     try {
171|         word_ = eng;
172|         translation_ = rus;
173|     } catch (const std::bad_alloc&) {
174|         throw FailOfMemoryAllocation("Pair");
175|     }
176| }
177|
178| bool Pair::operator<(const Pair& other) const {
179|     return word_ < other.word_;
180| }
181|
182| bool Pair::operator>(const Pair& other) const {
183|     return word_ > other.word_;
184| }
185|
186| bool Pair::operator==(const Pair& other) const {
187|     return word_ == other.word_;
188| }
189|
```

```
190| std::ostream& operator<<(std::ostream& output, const Pair& pair) {
191|     if (pair.translation_.isEmpty()) {
192|         throw NoTranslations();
193|     }
194|
195|     output << pair.word_ << " - " <<
pair.translation_.convertTranslationsToString();
196|     return output;
197| }
```

RBT.h

```
1 |
2 | #ifndef RBT_H
3 | #define RBT_H
4 |
5 | #include <fstream>
6 | #include <windows.h>
7 | #include <string>
8 | #include <iostream>
9 |
10| #include "Structs.h"
11|
12| inline std::string DATAPATH;
13|
14| enum Color { RED, BLACK };
15|
16| struct Node {
17|     Pair data_;
18|     Color color_;
19|     Node *left_;
20|     Node *right_;
21|     Node *parent_;
22|
23|     Node(const std::string& word, const List& translation);
24|     explicit Node(const Pair& pair);
25| };
26|
27| class RBT {
28| private:
29|     Node* root_ = nullptr;
30|
31|     void loadFromFile(const std::string& filename);
32|
33|     static void destroyTree(Node* node);
34|
35|     void rotateLeft(Node* node);
36|     void rotateRight(Node* node);
37|
38|     void fixInsert(Node*& current);
39|     void fixRemove(Node* current, Node* parentNode);
40| public:
41|     RBT();
42|
43|     ~RBT();
44|     void clear();
45|
46|     static Node *search(Node* node, const std::string& key);
47|
48|     static Color getColor(const Node* node);
49|     static void setColor(Node* node, const Color & color);
50|     Node* getRoot() const;
```

```
51|
52|     void insert(const Pair& value);
53|
54|     void transplant(Node* nodeToReplace, Node* replacementNode);
55|     static Node* findMaximum(Node* node);
56|     void remove(const std::string& word);
57|
58|     std::string makeTreeToString() const;
59|     static void makeTreeToStringRecursive(const Node* node, const std::string&
prefix, bool isTail, std::string& result);
60| };
61|
62| #endif //RBT_H
63|
```

RBT.cpp

```
1 |
2 | #include "../headers/RBT.h"
3 | #include "../headers/Exceptions.h"
4 | #include "../headers/MyJson.h"
5 |
6 | Node::Node(const std::string& word, const List& translation) {
7 |     try {
8 |         this->data_ = Pair(word, translation);
9 |     } catch (const std::bad_alloc&) {
10|         throw FailOfMemoryAllocation("NodeRBT");
11|     }
12|     this->color_ = RED;
13|     this->left_ = nullptr;
14|     this->right_ = nullptr;
15|     this->parent_ = nullptr;
16| }
17|
18| Node::Node(const Pair& pair) {
19|     try {
20|         this->data_ = pair;
21|     } catch (const std::bad_alloc&) {
22|         throw FailOfMemoryAllocation("NodeRBT");
23|     }
24|     this->color_ = RED;
25|     this->left_ = nullptr;
26|     this->right_ = nullptr;
27|     this->parent_ = nullptr;
28| }
29|
30| void RBT::loadFromFile(const std::string& filename) {
31|     std::ifstream file(filename);
32|     if (!file.is_open())
33|         return;
34|
35|     std::string content;
36|     std::string line;
37|     while (std::getline(file, line))
38|         content += line + '\n';
39|
40|     file.close();
41|
42|     size_t i = 0;
43|     while (i < content.size() && (content[i] == ' ' || content[i] == '\t'
44|         || content[i] == '\n' || content[i] == '\r'))
45|         ++i;
```

```
46|
47|
48|     if (i >= content.size() || content[i] != '{') {
49|         std::ofstream fout(filename);
50|         fout << "{\n}\n";
51|         fout.close();
52|         return;
53|     }
54|
55|     MyJson dict;
56|     dict.parse(content);
57|
58|     for (auto it = dict.begin(); it != dict.end(); ++it) {
59|         const std::string& word = it.key();
60|         MyJsonArray translationsArray = it.value();
61|
62|         List translations;
63|         for (auto tr = translationsArray.begin(); tr != translationsArray.end();
++tr)
64|             translations.push(tr.value());
65|
66|         insert(Pair(word, translations));
67|     }
68| }
69|
70| void RBT::destroyTree(Node* node) {
71|     if (node != nullptr) {
72|         destroyTree(node->left_);
73|         destroyTree(node->right_);
74|         delete node;
75|     }
76| }
77|
78| Color RBT::getColor(const Node* node) {
79|     if (node != nullptr)
80|         return node->color_;
81|     return BLACK;
82| }
83|
84| void RBT::setColor(Node* node, const Color& color) {
85|     if (color != RED && color != BLACK)
86|         throw WrongColor();
87|
88|     if (node != nullptr)
89|         node->color_ = color;
90| }
91|
92| Node *RBT::getRoot() const {
93|     return root_;
94| }
95|
96| void RBT::rotateLeft(Node* node) {
97|     if (node == nullptr)
98|         throw RotationErrorNullptr("Left");
99|
100|     if (node->right_ == nullptr)
101|         throw LeftRotationError();
102|
103|     Node* rightChild = node->right_;
104|
105|     node->right_ = rightChild->left_;
106|     if (rightChild->left_ != nullptr)
```

```
107|         rightChild->left_->parent_ = node;
108|
109|     rightChild->parent_ = node->parent_;
110|     if (node->parent_ == nullptr)
111|         root_ = rightChild;
112|     else if (node == node->parent_->left_)
113|         node->parent_->left_ = rightChild;
114|     else
115|         node->parent_->right_ = rightChild;
116|
117|     rightChild->left_ = node;
118|     node->parent_ = rightChild;
119| }
120|
121| void RBT::rotateRight(Node* node) {
122|     if (node == nullptr)
123|         throw RotationErrorNullptr("Right");
124|
125|     if (node->left_ == nullptr)
126|         throw RightRotationError();
127|
128|     Node* leftChild = node->left_;
129|
130|     node->left_ = leftChild->right_;
131|     if (leftChild->right_ != nullptr)
132|         leftChild->right_->parent_ = node;
133|
134|     leftChild->parent_ = node->parent_;
135|     if (node->parent_ == nullptr)
136|         root_ = leftChild;
137|     else if (node == node->parent_->left_)
138|         node->parent_->left_ = leftChild;
139|     else
140|         node->parent_->right_ = leftChild;
141|
142|     leftChild->right_ = node;
143|     node->parent_ = leftChild;
144| }
145|
146| void RBT::fixInsert(Node*& current) {
147|     if (current == nullptr)
148|         throw NullInsertionNode();
149|
150|     while (current != root_ && getColor(current) == RED &&
151|            current->parent_ && getColor(current->parent_) == RED) {
152|         Node* parent = current->parent_;
153|         Node* grandparent = parent->parent_;
154|
155|         if (grandparent == nullptr)
156|             throw MissingGrandparent();
157|
158|         const bool isParentLeftChild = parent == grandparent->left_;
159|         Node* uncle = isParentLeftChild ? grandparent->right_ : grandparent-
>left_;
160|
161|         if (getColor(uncle) == RED) {
162|             setColor(parent, BLACK);
163|             setColor(uncle, BLACK);
164|             setColor(grandparent, RED);
165|             current = grandparent;
166|         } else {
167|             if (isParentLeftChild && current == parent->right_) {
```

```
168|                 rotateLeft(parent);
169|                 current = parent;
170|             } else if (!isParentLeftChild && current == parent->left_) {
171|                 rotateRight(parent);
172|                 current = parent;
173|             }
174|
175|             setColor(current->parent_, BLACK);
176|             setColor(grandparent, RED);
177|             if (isParentLeftChild)
178|                 rotateRight(grandparent);
179|             else
180|                 rotateLeft(grandparent);
181|         }
182|     }
183|
184|     if (root_ != nullptr)
185|         root_->color_ = BLACK;
186| }
187|
188| void RBT::fixRemove(Node* current, Node* parentNode) {
189|     while (current != root_ && getColor(current) == BLACK) {
190|         if (parentNode == nullptr)
191|             break;
192|
193|         bool isLeftChild = current == parentNode->left_ || current == nullptr &&
parentNode->left_ == nullptr;
194|         Node* sibling = isLeftChild ? parentNode->right_ : parentNode->left_;
195|
196|         if (sibling == nullptr)
197|             break;
198|
199|         if (getColor(sibling) == RED) {
200|             setColor(sibling, BLACK);
201|             setColor(parentNode, RED);
202|             if (isLeftChild)
203|                 rotateLeft(parentNode);
204|             else
205|                 rotateRight(parentNode);
206|             sibling = isLeftChild ? parentNode->right_ : parentNode->left_;
207|
208|             if (sibling == nullptr)
209|                 break;
210|         }
211|
212|         const bool leftBlack = sibling->left_ == nullptr || getColor(sibling-
>left_) == BLACK;
213|         bool rightBlack = sibling->right_ == nullptr || getColor(sibling->right_)
== BLACK;
214|
215|         if (leftBlack && rightBlack) {
216|             setColor(sibling, RED);
217|             current = parentNode;
218|             parentNode = current->parent_;
219|         } else {
220|             if (isLeftChild) {
221|                 if (sibling->right_ == nullptr || getColor(sibling->right_) ==
BLACK) {
222|                     if (sibling->left_ != nullptr)
223|                         setColor(sibling->left_, BLACK);
224|                     setColor(sibling, RED);
225|                     rotateRight(sibling);
```

```
226|                     sibling = parentNode->right_;
227|                 }
228|                 setColor(sibling, getColor(parentNode));
229|                 setColor(parentNode, BLACK);
230|                 if (sibling->right_ != nullptr)
231|                     setColor(sibling->right_, BLACK);
232|                 rotateLeft(parentNode);
233|             } else {
234|                 if (sibling->left_ == nullptr || getColor(sibling->left_) ==
BLACK) {
235|                     if (sibling->right_ != nullptr)
236|                         setColor(sibling->right_, BLACK);
237|                     setColor(sibling, RED);
238|                     rotateLeft(sibling);
239|                     sibling = parentNode->left_;
240|                 }
241|                 setColor(sibling, getColor(parentNode));
242|                 setColor(parentNode, BLACK);
243|                 if (sibling->left_ != nullptr)
244|                     setColor(sibling->left_, BLACK);
245|                 rotateRight(parentNode);
246|             }
247|             current = root_;
248|             break;
249|         }
250|     }
251|
252|     if (current != nullptr)
253|         setColor(current, BLACK);
254| }
255|
256|
257|
258| Node *RBT::search(Node* node, const std::string & key) {
259|     while (node != nullptr) {
260|         if (key == node->data_.word_)
261|             return node;
262|
263|         if (key < node->data_.word_)
264|             node = node->left_;
265|         else
266|             node = node->right_;
267|     }
268|     return nullptr;
269| }
270|
271| RBT::RBT() {
272|     root_ = nullptr;
273|     loadFromFile(DATAPATH);
274| }
275|
276| RBT::~RBT() {
277|     destroyTree(root_);
278|     root_ = nullptr;
279| }
280|
281| void RBT::clear() {
282|     destroyTree(root_);
283|     root_ = nullptr;
284| }
285|
286| void RBT::insert(const Pair & value) {
```

```
287|     if (search(root_, value.word_) != nullptr)
288|         throw DuplicateWord();
289|
290|     Node** insertionPoint = &root_;
291|     Node* parent = nullptr;
292|
293|     while (*insertionPoint != nullptr) {
294|         parent = *insertionPoint;
295|         if (value < (*insertionPoint)->data_)
296|             insertionPoint = &(*insertionPoint)->left_;
297|         else
298|             insertionPoint = &(*insertionPoint)->right_;
299|     }
300|
301|     Node* newNode;
302|     try {
303|         newNode = new Node(value);
304|     } catch (const std::bad_alloc &) {
305|         throw FailOfMemoryAllocation("Node");
306|     }
307|
308|     newNode->parent_ = parent;
309|     *insertionPoint = newNode;
310|
311|     fixInsert(newNode);
312| }
313|
314|
315| void RBT::transplant(Node* nodeToReplace, Node* replacementNode) {
316|     if (nodeToReplace->parent_ == nullptr)
317|         root_ = replacementNode;
318|     else if (nodeToReplace == nodeToReplace->parent_->left_)
319|         nodeToReplace->parent_->left_ = replacementNode;
320|     else
321|         nodeToReplace->parent_->right_ = replacementNode;
322|
323|     if (replacementNode != nullptr)
324|         replacementNode->parent_ = nodeToReplace->parent_;
325| }
326|
327| Node *RBT::findMaximum(Node* node) {
328|     while (node->right_ != nullptr)
329|         node = node->right_;
330|     return node;
331| }
332|
333| void RBT::remove(const std::string & word) {
334|     Node* nodeToDelete = search(root_, word);
335|     if (nodeToDelete == nullptr)
336|         return;
337|
338|     Node* replacementNode = nodeToDelete;
339|     Node* replacementChild = nullptr;
340|     Node* replacementChildParent = nullptr;
341|     Color originalColor = replacementNode->color_;
342|
343|     if (nodeToDelete->left_ == nullptr) {
344|         replacementChild = nodeToDelete->right_;
345|         replacementChildParent = nodeToDelete->parent_;
346|         transplant(nodeToDelete, nodeToDelete->right_);
347|     } else if (nodeToDelete->right_ == nullptr) {
348|         replacementChild = nodeToDelete->left_;
```

```
349|         replacementChildParent = nodeToDelete->parent_;
350|         transplant(nodeToDelete, nodeToDelete->left_);
351|     } else {
352|         replacementNode = findMaximum(nodeToDelete->left_);
353|         originalColor = replacementNode->color_;
354|         replacementChild = replacementNode->left_;
355|         replacementChildParent = replacementNode;
356|
357|         if (replacementNode->parent_ != nodeToDelete) {
358|             transplant(replacementNode, replacementNode->left_);
359|             replacementNode->left_ = nodeToDelete->left_;
360|             replacementNode->left_->parent_ = replacementNode;
361|         }
362|
363|
364|         transplant(nodeToDelete, replacementNode);
365|         replacementNode->right_ = nodeToDelete->right_;
366|         replacementNode->right_->parent_ = replacementNode;
367|         replacementNode->color_ = nodeToDelete->color_;
368|     }
369|
370|     delete nodeToDelete;
371|
372|     if (originalColor == BLACK)
373|         fixRemove(replacementChild, replacementChildParent);
374| }
375|
376| std::string RBT::makeTreeToString() const {
377|     std::string result;
378|     makeTreeToStringRecursive(root_, "", true, result);
379|     return result;
380| }
381|
382| void RBT::makeTreeToStringRecursive(const Node* node, const std::string & prefix,
const bool isTail,
383|                                     std::string & result) {
384|     if (node == nullptr)
385|         return;
386|
387|     result += prefix + (isTail ? "└── " : "├── ");
388|
389|     result += (node->color_ == RED ? "[R] " : "[B] ") +
390|             node->data_.word_ + ": " +
391|             node->data_.translation_.convertTranslationsToString() + "\n";
392|
393|     const std::string newPrefix = prefix + (isTail ? "    " : "│   ");
394|
395|     if (node->left_ || node->right_) {
396|         if (node->left_)
397|             makeTreeToStringRecursive(node->left_, newPrefix, false, result);
398|         if (node->right_)
399|             makeTreeToStringRecursive(node->right_, newPrefix, true, result);
400|     }
401| }
402|
```

Dict.h

```
1 |
2 | #ifndef DICT_H
3 | #define DICT_H
4 |
```

```
5 | #include "RBT.h"
6 | #include "Structs.h"
7 |
8 | #include <string>
9 |
10| class Dict {
11| private:
12|     RBT tree_;
13|
14|     static void printRecursive(const Node* node);
15|
16|     static void writeToFile(const Pair & value);
17|     static void removeFromFile(const std::string & word);
18|
19| public:
20|     Dict();
21|     ~Dict();
22|
23|     RBT getTree();
24|
25|     void insert(std::string& word, std::string& translationsStr);
26|     void remove(const std::string& word);
27|
28|     void addTranslation(const std::string& word, const std::string&
newTranslation);
29|     void removeTranslation(const std::string& word, const std::string&
translation);
30|
31|     List& findTranslationByWord(const std::string& word) const;
32|     static void printWordWithTranslations(const std::string& word, const List&
translations);
33|
34|     std::string findWordByTranslation(const std::string& translation) const;
35|
36|     static Node* findInTreeByTranslation(Node* node, const std::string&
translation);
37|
38|     std::string autotranslate(std::string input) const;
39|
40|     void print() const;
41|
42|     void run();
43| };
44|
45| bool isEnglishWord(const std::string& word);
46| bool isRussianWord(const std::string& word);
47|
48| void toUpperCaseE(std::string& str);
49| void trimSpaces(std::string& str);
50| void toLowerCaseE(std::string& str);
51| void toLowerCaseR(std::string& str);
52|
53| #endif //DICT_H
54|
```

Dict.cpp

```
1 |
2 | #include "../headers/Dict.h"
3 | #include "../headers/Exceptions.h"
4 | #include "../headers/MyJson.h"
5 |
```

```
6 | void Dict::printRecursive(const Node* node) {
7 |     if (!node)
8 |         return;
9 |
10|     printRecursive(node->left_);
11|     std::cout << node->data_ << std::endl;
12|     printRecursive(node->right_);
13| }
14|
15| void Dict::writeToFile(const Pair& value) {
16|     std::ifstream fin(DATAPATH);
17|     if (!fin) {
18|         std::ofstream fout(DATAPATH);
19|         fout << "{\n}\n";
20|         fout.close();
21|         fin.open(DATAPATH);
22|         if (!fin)
23|             throw FailOfMemoryAllocation("Dictionary.json file");
24|     }
25|
26|     std::string content;
27|     char ch;
28|     while (fin.get(ch))
29|         content += ch;
30|     fin.close();
31|
32|     MyJson jsonData;
33|     jsonData.parse(content);
34|
35|     std::string newContent = "{";
36|     bool inserted = false;
37|
38|     for (size_t i = 0; i < jsonData.getSize(); ++i) {
39|         const auto& entry = jsonData.getEntry(i);
40|
41|         if (entry.key == value.word_) {
42|             if (i > 0)
43|                 newContent += ",";
44|             newContent += "\n  \"" + value.word_ + "\": [";
45|             bool first = true;
46|             for (NodeList* cur = value.translation_.getHead(); cur; cur = cur-
>next_) {
47|                 if (!first)
48|                     newContent += ",";
49|                 newContent += "\"" + cur->value_ + "\"";
50|                 first = false;
51|             }
52|             newContent += "]";
53|             inserted = true;
54|         } else {
55|             if (i > 0 || inserted)
56|                 newContent += ",";
57|             newContent += "\n  \"" + entry.key + "\": [";
58|             for (size_t j = 0; j < entry.translationsCount; ++j) {
59|                 if (j > 0)
60|                     newContent += ",";
61|                 newContent += "\"" + entry.translations[j] + "\"";
62|             }
63|             newContent += "]";
64|         }
65|     }
66|
```

```
67|     if (!inserted) {
68|         if (jsonData.getSize() > 0)
69|             newContent += ",";
70|
71|         newContent += "\n  \"" + value.word_ + "\": [";
72|         bool first = true;
73|         for (NodeList* cur = value.translation_.getHead(); cur; cur = cur->next_)
{
74|             if (!first)
75|                 newContent += ",";
76|
77|             newContent += "\"" + cur->value_ + "\"";
78|             first = false;
79|         }
80|         newContent += "]";
81|     }
82|
83|     newContent += "\n}\n";
84|
85|     std::ofstream fout(DATAPATH);
86|     if (!fout)
87|         throw FailOfMemoryAllocation("Dictionary.json file");
88|
89|     fout << newContent;
90|     fout.close();
91| }
92|
93| void Dict::removeFromFile(const std::string & word) {
94|     std::ifstream fin(DATAPATH);
95|     if (!fin)
96|         throw NodeNotFound();
97|
98|     std::string content;
99|     char ch;
100|     while (fin.get(ch))
101|         content += ch;
102|     fin.close();
103|
104|     MyJson jsonData;
105|     jsonData.parse(content);
106|
107|     bool found = false;
108|     size_t indexToRemove = 0;
109|     for (size_t i = 0; i < jsonData.getSize(); ++i) {
110|         if (jsonData.getEntry(i).key == word) {
111|             found = true;
112|             indexToRemove = i;
113|             break;
114|         }
115|     }
116|
117|     if (!found)
118|         throw NodeNotFound();
119|
120|     std::string newContent = "{";
121|     bool firstEntry = true;
122|
123|     for (size_t i = 0; i < jsonData.getSize(); ++i) {
124|         if (i == indexToRemove)
125|             continue;
126|
127|         if (!firstEntry)
```

```
128|             newContent += ",";
129|         firstEntry = false;
130|
131|         const auto& entry = jsonData.getEntry(i);
132|         newContent += "\n  \"" + entry.key + "\": [";
133|
134|         for (size_t j = 0; j < entry.translationsCount; ++j) {
135|             if (j > 0)
136|                 newContent += ",";
137|             newContent += "\"" + entry.translations[j] + "\"";
138|         }
139|         newContent += "]";
140|     }
141|
142|     newContent += "\n}\n";
143|
144|     std::ofstream fout(DATAPATH);
145|     if (!fout)
146|         throw FailOfMemoryAllocation("Dictionary.json file");
147|     fout << newContent;
148|     fout.close();
149| }
150|
151| Dict::Dict() = default;
152|
153| Dict::~Dict() {
154|     tree_.clear();
155| }
156|
157| RBT Dict::getTree() {
158|     return tree_;
159| }
160|
161| void Dict::insert(std::string& word, std::string& translationsStr) {
162|     trimSpaces(word);
163|     if (word.empty() || !isEnglishWord(word))
164|         throw InvalidWordLanguage();
165|
166|     toLowerCaseE(word);
167|
168|     if (translationsStr.empty())
169|         throw EmptyValue();
170|
171|     List listTranslations;
172|     size_t start = 0;
173|     size_t end = translationsStr.find(';');
174|
175|     while (end != std::string::npos) {
176|         std::string translation = translationsStr.substr(start, end - start);
177|         trimSpaces(translation);
178|         if (!translation.empty()) {
179|             if (!isRussianWord(translation))
180|                 throw InvalidTranslationLanguage();
181|
182|             toLowerCaseR(translation);
183|             listTranslations.push(translation);
184|         }
185|         start = end + 1;
186|         end = translationsStr.find(';', start);
187|     }
188|
189|     std::string lastTranslation = translationsStr.substr(start);
```

```
190|     trimSpaces(lastTranslation);
191|     if (!lastTranslation.empty()) {
192|         if (!isRussianWord(lastTranslation))
193|             throw InvalidTranslationLanguage();
194|
195|         toLowerCaseR(lastTranslation);
196|         listTranslations.push(lastTranslation);
197|     }
198|
199|     if (listTranslations.isEmpty())
200|         throw EmptyValue();
201|
202|     tree_.insert(Pair(word, listTranslations));
203|     writeToFile(Pair(word, listTranslations));
204| }
205|
206| void Dict::addTranslation(const std::string & word, const std::string &
newTranslation) {
207|     Node* node = tree_.search(tree_.getRoot(), word);
208|     if (!node)
209|         throw NodeNotFound();
210|
211|     const NodeList* current = node->data_.translation_.getHead();
212|     while (current != nullptr) {
213|         if (current->value_ == newTranslation)
214|             throw DuplicateTranslation();
215|         current = current->next_;
216|     }
217|
218|     node->data_.translation_.push(newTranslation);
219|     writeToFile(node->data_);
220| }
221|
222| void Dict::removeTranslation(const std::string & word, const std::string &
translation) {
223|     Node* node = RBT::search(tree_.getRoot(), word);
224|     if (!node)
225|         throw NodeNotFound();
226|
227|     NodeList* cur = node->data_.translation_.getHead();
228|     while (cur && cur->value_ != translation)
229|         cur = cur->next_;
230|
231|     if (!cur)
232|         throw NodeNotFoundInList();
233|
234|     node->data_.translation_.remove(cur);
235|
236|     if (node->data_.translation_.isEmpty()) {
237|         tree_.remove(word);
238|         removeFromFile(word);
239|     } else {
240|         writeToFile(node->data_);
241|     }
242| }
243|
244| List &Dict::findTranslationByWord(const std::string & word) const {
245|     Node* node = RBT::search(tree_.getRoot(), word);
246|
247|     if (node == nullptr)
248|         throw NoFoundWord();
249|
```

```
250|     return node->data_.translation_;
251| }
252|
253| void Dict::printWordWithTranslations(const std::string & word, const List &
translations) {
254|     std::cout << "\nСлово: " << word << "\n";
255|     std::cout << "Переводы: " << translations.convertTranslationsToString() <<
"\n\n";
256| }
257|
258| std::string Dict::findWordByTranslation(const std::string & translation) const {
259|     Node* result = findInTreeByTranslation(tree_.getRoot(), translation);
260|     if (!result)
261|         throw NodeNotFoundInList();
262|
263|     return result->data_.word_;
264| }
265|
266| Node *Dict::findInTreeByTranslation(Node* node, const std::string & translation)
{
267|     if (!node)
268|         return nullptr;
269|
270|     Node* leftResult = findInTreeByTranslation(node->left_, translation);
271|     if (leftResult)
272|         return leftResult;
273|
274|     const NodeList* current = node->data_.translation_.getHead();
275|     while (current) {
276|         if (current->value_ == translation)
277|             return node;
278|         current = current->next_;
279|     }
280|
281|     return findInTreeByTranslation(node->right_, translation);
282| }
283|
284| std::string Dict::autotranslate(std::string input) const {
285|     trimSpaces(input);
286|
287|     if (isEnglishWord(input)) {
288|         toLowerCaseE(input);
289|         return findTranslationByWord(input).convertTranslationsToString();
290|     }
291|     toLowerCaseR(input);
292|     return findWordByTranslation(input);
293| }
294|
295|
296| void Dict::remove(const std::string & word) {
297|     tree_.remove(word);
298|     removeFromFile(word);
299| }
300|
301| void Dict::print() const {
302|     printRecursive(tree_.getRoot());
303| }
304|
305| void Dict::run() {
306|     std::cout << "Выбрана пользовательская работа со словарём\n";
307|     std::cout << "\nДоступные команды:\n"
308|                     << "1. INSERT - Добавить новое слово и его переводы в
```

```
словарь.\n"
309|                     << "2. REMOVE - Удалить слово из словаря.\n"
310|                     << "3. ADDTRANSLATION - Добавить новый перевод для
существующего слова.\n"
311|                     << "4. REMOVETRANSLATION - Удалить перевод для существующего
слова.\n"
312|                     << "5. FINDTRANSLATION - Найти все переводы для слова.\n"
313|                     << "6. FINDWORD - Найти слово по его переводу.\n"
314|                     << "7. AUTOTRANSLATE - Автоматический перевод текста
(определяется язык ввода).\n"
315|                     << "8. PRINT - Вывести все слова и их переводы из словаря.\n"
316|                     << "9. TREE - Печать дерева словаря в текстовом виде.\n"
317|                     << "10. EXIT - Завершить работу программы.\n";
318|     while (true) {
319|         std::cout << "Введите команду: ";
320|         try {
321|             std::string command;
322|             std::getline(std::cin, command);
323|             trimSpaces(command);
324|             toUpperCaseE(command);
325|
326|             if (command == "INSERT") {
327|                 std::string word;
328|                 std::cout << "Введите английское слово: ";
329|                 std::getline(std::cin, word);
330|
331|                 std::string translations;
332|                 std::cout << "Введите перевод(ы), разделяя их точкой с запятой
';': ";
333|                 std::getline(std::cin, translations);
334|
335|                 try {
336|                     insert(word, translations);
337|                     std::cout << "Добавлено: " << word << "\n";
338|                 } catch (const std::exception& e) {
339|                     std::cout << "Ошибка: " << e.what() << "\n";
340|                 }
341|             } else if (command == "REMOVE") {
342|                 std::string word;
343|                 std::cout << "Введите слово для удаления: ";
344|                 std::getline(std::cin, word);
345|                 trimSpaces(word);
346|                 toLowerCaseE(word);
347|
348|                 if (!isEnglishWord(word))
349|                     throw InvalidWordLanguage();
350|
351|                 remove(word);
352|                 std::cout << "Удалено успешно!\n";
353|             } else if (command == "ADDTRANSLATION") {
354|                 std::string word;
355|                 std::cout << "Введите слово для добавления перевода: ";
356|                 std::getline(std::cin, word);
357|                 trimSpaces(word);
358|                 toLowerCaseE(word);
359|
360|                 if (!isEnglishWord(word))
361|                     throw InvalidWordLanguage();
362|
363|                 if (!tree_.search(tree_.getRoot(), word))
364|                     throw NodeNotFound();
365|
```

```
366|                 std::string translation;
367|                 std::cout << "Введите новый перевод: ";
368|                 std::getline(std::cin, translation);
369|                 trimSpaces(translation);
370|                 toLowerCaseR(translation);
371|
372|                 if (!isRussianWord(translation))
373|                     throw InvalidTranslationLanguage();
374|
375|                 addTranslation(word, translation);
376|                 std::cout << "Добавлено успешно!\n";
377|             } else if (command == "REMOVETRANSLATION") {
378|                 std::string word;
379|                 std::cout << "Введите слово для удаления перевода: ";
380|                 std::getline(std::cin, word);
381|                 trimSpaces(word);
382|                 toLowerCaseE(word);
383|
384|                 if (!isEnglishWord(word))
385|                     throw InvalidWordLanguage();
386|
387|                 if (!tree_.search(tree_.getRoot(), word))
388|                     throw NodeNotFound();
389|
390|                 std::string translation;
391|                 std::cout << "Введите перевод для удаления: ";
392|                 std::getline(std::cin, translation);
393|                 trimSpaces(translation);
394|                 toLowerCaseR(translation);
395|
396|                 if (!isRussianWord(translation)) throw
InvalidTranslationLanguage();
397|
398|                 removeTranslation(word, translation);
399|
400|                 if (!tree_.search(tree_.getRoot(), word)) {
401|                     std::cout << "Перевод удален, и слово полностью удалено из
словаря!\n";
402|                 } else {
403|                     std::cout << "Перевод удален!\n";
404|                 }
405|             } else if (command == "FINDTRANSLATION") {
406|                 std::string word;
407|                 std::cout << "Введите слово для поиска переводов: ";
408|                 std::getline(std::cin, word);
409|                 trimSpaces(word);
410|                 toLowerCaseE(word);
411|
412|                 if (!isEnglishWord(word)) throw InvalidWordLanguage();
413|
414|                 Node* node = tree_.search(tree_.getRoot(), word);
415|                 if (!node)
416|                     throw NoFoundWord();
417|
418|                 std::cout << "Найдено: " << node->data_ << "\n";
419|             } else if (command == "FINDWORD") {
420|                 std::string translation;
421|                 std::cout << "Введите перевод для поиска слова: ";
422|                 std::getline(std::cin, translation);
423|                 trimSpaces(translation);
424|                 toLowerCaseR(translation);
425|
```

```
426|                 if (!isRussianWord(translation)) throw
InvalidTranslationLanguage();
427|
428|                 std::string word = findWordByTranslation(translation);
429|                 if (!word.empty()) {
430|                     std::cout << "Найдено слово: " << word << '\n';
431|                 } else {
432|                     throw NoFoundWord();
433|                 }
434|             } else if (command == "PRINT") {
435|                 print();
436|             } else if (command == "TREE") {
437|                 std::cout << tree_.makeTreeToString() << "\n";
438|             } else if (command == "EXIT") {
439|                 std::cout << "Выход из программы.\n";
440|                 break;
441|             } else if (command == "AUTOTRANSLATE") {
442|                 std::string input;
443|                 std::cout << "Введите текст для автоперевода (язык определяется
автоматически): ";
444|                 std::getline(std::cin, input);
445|
446|                 std::cout << "Результат перевода: " << autotranslate(input) <<
"\n";
447|             } else {
448|                 std::cout << "Неизвестная команда. Попробуйте снова.\n";
449|             }
450|         } catch (const std::logic_error & e) {
451|             std::cout << e.what() << "\nПопробуйте снова.\n\n";
452|         } catch (const std::runtime_error & e) {
453|             std::cerr << e.what();
454|             std::cout << "Ошибка критическая, завершение работы!!!\n";
455|             exit(EXIT_FAILURE);
456|         }
457|     }
458| }
459|
460| bool isEnglishWord(const std::string & word) {
461|     for (const unsigned char c: word) {
462|         if (!((c >= 'A' && c <= 'Z') ||
463|               (c >= 'a' && c <= 'z') ||
464|               c == '\'' || c == '-' ||
465|               c == ',' || c == ' ')) {
466|             return false;
467|         }
468|     }
469|     return true;
470| }
471|
472| bool isRussianWord(const std::string & word) {
473|     for (size_t i = 0; i < word.size();) {
474|         const unsigned char c = word[i];
475|
476|         if (c == ' ' || c == ',') {
477|             i++;
478|             continue;
479|         }
480|
481|         if ((c == 0xD0 || c == 0xD1) && i + 1 < word.size()) {
482|             const unsigned char next = word[i + 1];
483|
484|             const bool isRussian = (c == 0xD0 && next >= 0x90 && next <= 0xBF) ||
```

```
                                (c == 0xD1 && next >= 0x80 && next <= 0x8F) ||
                                (c == 0xD0 && next == 0x81) ||
                                (c == 0xD1 && next == 0x91);

            if (!isRussian) return false;
            i += 2;
        } else {
            return false;
        }
    }
    return !word.empty();
}

void toUpperCaseE(std::string & str) {
    for (char & c: str) {
        if (c >= 'a' && c <= 'z') {
            c = c - ('a' - 'A');
        }
        else if (c >= 'a' && c <= 'я') {
            c = c - ('a' - 'A');
        } else if (c == 'ё') {
            c = 'Ё';
        }
    }
}

void trimSpaces(std::string& str) {
    if (str.empty()) return;

    size_t start = 0;
    while (start < str.size() && str[start] == ' ')
        ++start;

    if (start == str.size()) {
        str.clear();
        return;
    }

    size_t end = str.size() - 1;
    while (end > start && str[end] == ' ')
        --end;

    str = str.substr(start, end - start + 1);
}


void toLowerCaseE(std::string & str) {
    for (char & c: str) {
        if (c >= 'A' && c <= 'Z')
            c = c + ('a' - 'A');
    }
}

void toLowerCaseR(std::string & str) {
    for (size_t i = 0; i < str.size();) {
        unsigned char c = str[i];

        if ((c == 0xD0 || c == 0xD1) && i + 1 < str.size()) {
            const unsigned char next = str[i + 1];

            if (c == 0xD0) {
                if (next >= 0x90 && next <= 0x9F) {
```

```
547|                     str[i + 1] = next + 0x20;
548|                 } else if (next == 0x81) {
549|                     str[i] = 0xD1;
550|                     str[i + 1] = 0x91;
551|                 } else if (next >= 0xA0 && next <= 0xAF) {
552|                     str[i] = 0xD1;
553|                     str[i + 1] = next - 0x20;
554|                 }
555|             }
556|             i += 2;
557|         } else {
558|             i++;
559|         }
560|     }
561| }
562|
```

MyJson.h

```
1 |
2 | #ifndef MYJSON_H
3 | #define MYJSON_H
4 |
5 | #include <string>
6 |
7 | class MyJsonArrayIterator {
8 | public:
9 |     MyJsonArrayIterator(const std::string* data, size_t index);
10|     MyJsonArrayIterator& operator++();
11|     bool operator!=(const MyJsonArrayIterator& other) const;
12|     std::string value() const;
13|
14| private:
15|     const std::string* data_;
16|     size_t index_;
17| };
18|
19| class MyJsonArray {
20| public:
21|     MyJsonArray(const std::string* data, size_t size);
22|     MyJsonArrayIterator begin() const;
23|     MyJsonArrayIterator end() const;
24|
25| private:
26|     const std::string* data_;
27|     size_t size_;
28| };
29|
30| class MyJsonIterator;
31|
32| class MyJson {
33| public:
34|     struct Entry {
35|         std::string key;
36|         std::string* translations;
37|         size_t translationsCount;
38|
39|         Entry();
40|         ~Entry();
41|
42|         Entry(const Entry& other);
43|
```

```
44|         Entry& operator=(const Entry& other);
45|     };
46|
47|     MyJson();
48|     ~MyJson();
49|
50|     void parse(const std::string& text);
51|     size_t getSize() const;
52|     const Entry& getEntry(size_t index) const;
53|
54|     MyJsonIterator begin() const;
55|     MyJsonIterator end() const;
56|
57| private:
58|     static void skipWhitespace(const std::string& text, size_t& pos);
59|     static std::string parseString(const std::string& text, size_t& pos);
60|     void grow();
61|
62|     Entry* entries_;
63|     size_t size_;
64|     size_t capacity_;
65| };
66|
67| class MyJsonIterator {
68| public:
69|     MyJsonIterator(const MyJson* json, size_t index);
70|     MyJsonIterator& operator++();
71|     bool operator!=(const MyJsonIterator& other) const;
72|     std::string key() const;
73|     MyJsonArray value() const;
74|
75| private:
76|     const MyJson* json_;
77|     size_t index_;
78| };
79|
80| #endif // MYJSON_H
81|
```

MyJson.cpp

```
1 |
2 | #include "../headers/MyJson.h"
3 |
4 | void MyJson::skipWhitespace(const std::string& text, size_t& pos) {
5 |     while (pos < text.size() && (text[pos] == ' ' || text[pos]
6 |         == '\n' || text[pos] == '\t' || text[pos] == '\r')) {
7 |         ++pos;
8 |     }
9 | }
10|
11| std::string MyJson::parseString(const std::string& text, size_t& pos) {
12|     if (text[pos] != '"')
13|         return "";
14|     ++pos;
15|     std::string result;
16|     while (pos < text.size() && text[pos] != '"')
17|         result += text[pos++];
18|
19|     if (pos < text.size() && text[pos] == '"')
20|         ++pos;
21|     return result;
```

```
22| }
23|
24| MyJsonArrayIterator::MyJsonArrayIterator(const std::string* data, size_t index)
25|     : data_(data), index_(index) {}
26|
27| MyJsonArrayIterator& MyJsonArrayIterator::operator++() {
28|     ++index_;
29|     return *this;
30| }
31|
32| bool MyJsonArrayIterator::operator!=(const MyJsonArrayIterator& other) const {
33|     return index_ != other.index_;
34| }
35|
36| std::string MyJsonArrayIterator::value() const {
37|     return data_[index_];
38| }
39|
40| MyJsonArray::MyJsonArray(const std::string* data, size_t size)
41|     : data_(data), size_(size) {}
42|
43| MyJson::Entry::Entry() : translations(new std::string[32]), translationsCount(0)
{}
44|
45| MyJson::Entry::~Entry() {
46|     delete[] translations;
47| }
48|
49| MyJson::Entry::Entry(const Entry & other) {
50|     key = other.key;
51|     translationsCount = other.translationsCount;
52|     translations = new std::string[32];
53|     for (size_t i = 0; i < translationsCount; ++i)
54|         translations[i] = other.translations[i];
55| }
56|
57| MyJson::Entry & MyJson::Entry::operator=(const Entry & other) {
58|     if (this != &other) {
59|         delete[] translations;
60|         key = other.key;
61|         translationsCount = other.translationsCount;
62|         translations = new std::string[32];
63|         for (size_t i = 0; i < translationsCount; ++i)
64|             translations[i] = other.translations[i];
65|     }
66|     return *this;
67| }
68|
69| MyJson::MyJson() : size_(0), capacity_(4) {
70|     entries_ = new Entry[capacity_];
71| }
72|
73| MyJson::~MyJson() {
74|     delete[] entries_;
75| }
76|
77| void MyJson::grow() {
78|     capacity_ *= 2;
79|     auto* newEntries = new Entry[capacity_];
80|     for (size_t i = 0; i < size_; ++i)
81|         newEntries[i] = entries_[i];
82|
```

```
83|     delete[] entries_;
84|     entries_ = newEntries;
85| }
86|
87| void MyJson::parse(const std::string& text) {
88|     size_ = 0;
89|     size_t pos = 0;
90|     skipWhitespace(text, pos);
91|     if (text[pos] != '{')
92|         return;
93|     ++pos;
94|
95|     while (pos < text.size()) {
96|         skipWhitespace(text, pos);
97|         if (text[pos] == '}')
98|             break;
99|
100|         std::string key = parseString(text, pos);
101|         skipWhitespace(text, pos);
102|         if (text[pos] != ':')
103|             return;
104|         ++pos;
105|         skipWhitespace(text, pos);
106|         if (text[pos] != '[')
107|             return;
108|         ++pos;
109|
110|         if (size_ >= capacity_)
111|             grow();
112|
113|         entries_[size_].key = key;
114|         entries_[size_].translationsCount = 0;
115|
116|         while (pos < text.size()) {
117|             skipWhitespace(text, pos);
118|             if (text[pos] == ']') {
119|                 ++pos;
120|                 break;
121|             }
122|
123|             std::string translation = parseString(text, pos);
124|             if (entries_[size_].translationsCount < 32)
125|                 entries_[size_].translations[entries_[size_].translationsCount++]
= translation;
126|
127|             skipWhitespace(text, pos);
128|             if (text[pos] == ',')
129|                 ++pos;
130|         }
131|
132|         skipWhitespace(text, pos);
133|         if (text[pos] == ',')
134|             ++pos;
135|         ++size_;
136|     }
137| }
138|
139| const MyJson::Entry& MyJson::getEntry(size_t index) const {
140|     return entries_[index];
141| }
142|
143| size_t MyJson::getSize() const {
```

```
144|     return size_;
145| }
146|
147| MyJsonIterator::MyJsonIterator(const MyJson* json, size_t index)
148|     : json_(json), index_(index) {}
149|
150| MyJsonIterator& MyJsonIterator::operator++() {
151|     ++index_;
152|     return *this;
153| }
154|
155| bool MyJsonIterator::operator!=(const MyJsonIterator& other) const {
156|     return index_ != other.index_;
157| }
158|
159| std::string MyJsonIterator::key() const {
160|     return json_->getEntry(index_).key;
161| }
162|
163| MyJsonArray MyJsonIterator::value() const {
164|     const auto& entry = json_->getEntry(index_);
165|     return MyJsonArray(entry.translations, entry.translationsCount);
166| }
167|
168| MyJsonArrayIterator MyJsonArray::begin() const {
169|     return {data_, 0};
170| }
171|
172| MyJsonArrayIterator MyJsonArray::end() const {
173|     return {data_, size_};
174| }
175|
176| MyJsonIterator MyJson::begin() const {
177|     return {this, 0};
178| }
179|
180| MyJsonIterator MyJson::end() const {
181|     return {this, size_};
182| }
```

Test.h

```
1 |
2 | #ifndef TESTS_H
3 | #define TESTS_H
4 |
5 | #include "../headers/Dict.h"
6 | #include "../headers/RBT.h"
7 | #include "../headers/Structs.h"
8 | #include "../headers/Exceptions.h"
9 | #include <string>
10| #include <iostream>
11| #include <sstream>
12|
13|
14| void printTestResult(const std::string& testName, bool passed);
15| void initializeEmptyJsonFile(const std::string& filename);
16|
17| void testListConstructors();
18| void testListPush();
19| void testListRemove();
20| void testListToString();
```

```
21|
22| void testPairConstructors();
23| void testPairOperators();
24| void testPairOutputOperator();
25|
26| void testRBTConstructors();
27| void testRBTInsert();
28| void testRBTRemove();
29| void testRBTSearch();
30| void testRBTToString();
31| void testRBTExceptions();
32|
33| void testDictInsert();
34| void testDictExceptions();
35|
36| void runAllTests();
37|
38| #endif // TESTS_H
39|
```

Test.cpp

```
1 |
2 | #include "Tests.h"
3 |
4 | void printTestResult(const std::string & testName, bool passed) {
5 |     std::cout << (passed ? "[PASSED] :-D " : "[FAILED] (¬_¬ ) ") << testName <<
"\n";
6 | }
7 |
8 | void initializeEmptyJsonFile(const std::string & filename) {
9 |     std::ofstream file(filename);
10|     if (!file.is_open()) {
11|         throw std::runtime_error("Failed to open file: " + filename);
12|     }
13|
14|     file << "{"
15|             "\n"
16|             "}";
17|
18|     file.close();
19| }
20|
21| void testListConstructors() {
22|     bool passed = true;
23|     try {
24|         List list1;
25|         if (list1.getHead() != nullptr || !list1.isEmpty())
26|             passed = false;
27|
28|         List list2;
29|         list2.push("test");
30|         List list3 = list2;
31|         if (list3.getHead() == nullptr || list3.isEmpty() || list3.getHead()-
>value_ != "test")
32|             passed = false;
33|     } catch (...) {
34|         passed = false;
35|     }
36|     printTestResult("List Constructors", passed);
37| }
38|
```

```
39| void testListPush() {
40|     bool passed = true;
41|     try {
42|         List list;
43|         list.push("b");
44|         list.push("a");
45|         list.push("c");
46|
47|         NodeList* head = list.getHead();
48|         if (head == nullptr || head->value_ != "a" ||
49|             head->next_ == nullptr || head->next_->value_ != "b" ||
50|             head->next_->next_ == nullptr || head->next_->next_->value_ != "c") {
51|             passed = false;
52|         }
53|     } catch (...) {
54|         passed = false;
55|     }
56|     printTestResult("List Push", passed);
57| }
58|
59| void testListRemove() {
60|     bool passed = true;
61|     try {
62|         List list;
63|         list.push("a");
64|         list.push("b");
65|         list.push("c");
66|
67|         NodeList* node = list.getHead()->next_;
68|         list.remove(node);
69|         if (list.getSize() != 2 || list.getHead()->value_ != "a" ||
list.getHead()->next_->value_ != "c")
70|             passed = false;
71|
72|         list.remove(list.getHead());
73|         if (list.getSize() != 1 || list.getHead()->value_ != "c")
74|             passed = false;
75|
76|         list.remove(list.getHead());
77|         if (!list.isEmpty())
78|             passed = false;
79|     } catch (...) {
80|         passed = false;
81|     }
82|     printTestResult("List Remove", passed);
83| }
84|
85| void testListToString() {
86|     bool passed = true;
87|     try {
88|         List list;
89|         list.push("а");
90|         list.push("б");
91|         list.push("в");
92|
93|         std::string result = list.convertTranslationsToString();
94|         if (result != "а; б; в")
95|             passed = false;
96|     } catch (...) {
97|         passed = false;
98|     }
99|     printTestResult("List ToString", passed);
```

```
100| }
101|
102| void testPairConstructors() {
103|     bool passed = true;
104|     try {
105|         Pair pair1;
106|         if (!pair1.word_.empty() || !pair1.translation_.isEmpty())
107|             passed = false;
108|
109|         List translations;
110|         translations.push("trans1");
111|         Pair pair2("word", translations);
112|         if (pair2.word_ != "word" || pair2.translation_.getHead()->value_ !=
"trans1")
113|             passed = false;
114|     } catch (...) {
115|         passed = false;
116|     }
117|     printTestResult("Pair Constructors", passed);
118| }
119|
120| void testPairOperators() {
121|     bool passed = true;
122|     try {
123|         List listA; listA.push("a");
124|         List listB; listB.push("b");
125|         List listC; listC.push("c");
126|
127|         const Pair p1("apple", listA);
128|         const Pair p2("banana", listB);
129|         const Pair p3("apple", listC);
130|
131|         if (!(p1 < p2) || p2 < p1)
132|             passed = false;
133|         if (!(p2 > p1) || p1 > p2)
134|             passed = false;
135|         if (!(p1 == p3) || p1 == p2)
136|             passed = false;
137|     } catch (...) {
138|         passed = false;
139|     }
140|     printTestResult("Pair Operators", passed);
141| }
142|
143| void testPairOutputOperator() {
144|     bool passed = true;
145|     try {
146|         List list;
147|         list.push("trans1");
148|         Pair pair("word", list);
149|
150|         std::ostringstream oss;
151|         oss << pair;
152|         if (oss.str() != "word - trans1")
153|             passed = false;
154|
155|         Pair emptyPair("empty", List());
156|         try {
157|             oss << emptyPair;
158|             passed = false;
159|         } catch (const NoTranslations&) {
160|             // Ожидаемое исключение
```

```
161|         } catch (...) {
162|             passed = false;
163|         }
164|     } catch (...) {
165|         passed = false;
166|     }
167|     printTestResult("Pair Output Operator", passed);
168| }
169|
170| void testRBTConstructors() {
171|     bool passed = true;
172|     try {
173|         RBT tree;
174|         if (tree.getRoot() != nullptr)
175|             passed = false;
176|     } catch (...) {
177|         passed = false;
178|     }
179|     printTestResult("RBT Constructors", passed);
180| }
181|
182| void testRBTInsert() {
183|     bool passed = true;
184|     try {
185|         RBT tree;
186|         List list1; list1.push("банан");
187|         List list2; list2.push("яблоко");
188|         List list3; list3.push("вишня");
189|
190|         tree.insert(Pair("banana", list1));
191|         tree.insert(Pair("apple", list2));
192|         tree.insert(Pair("cherry", list3));
193|
194|         if (tree.search(tree.getRoot(), "apple") == nullptr ||
195|             tree.search(tree.getRoot(), "banana") == nullptr ||
196|             tree.search(tree.getRoot(), "cherry") == nullptr) {
197|             passed = false;
198|         }
199|     } catch (...) {
200|         passed = false;
201|     }
202|     printTestResult("RBT Insert", passed);
203| }
204|
205| void testRBTRemove() {
206|     bool passed = true;
207|     try {
208|         RBT tree;
209|         List list1; list1.push("банан");
210|         List list2; list2.push("яблоко");
211|         List list3; list3.push("вишня");
212|
213|         tree.insert(Pair("banana", list1));
214|         tree.insert(Pair("apple", list2));
215|         tree.insert(Pair("cherry", list3));
216|
217|         tree.remove("apple");
218|         if (tree.search(tree.getRoot(), "apple") != nullptr)
219|             passed = false;
220|
221|         tree.remove("banana");
222|         if (tree.search(tree.getRoot(), "banana") != nullptr)
```

```
223|             passed = false;
224|     } catch (...) {
225|         passed = false;
226|     }
227|     printTestResult("RBT Remove", passed);
228| }
229|
230| void testRBTSearch() {
231|     bool passed = true;
232|     try {
233|         RBT tree;
234|         List list1; list1.push("банан");
235|         List list2; list2.push("яблоко");
236|
237|         tree.insert(Pair("banana", list1));
238|         tree.insert(Pair("apple", list2));
239|
240|         if (tree.search(tree.getRoot(), "apple") == nullptr ||
241|             tree.search(tree.getRoot(), "banana") == nullptr ||
242|             tree.search(tree.getRoot(), "nonexistent") != nullptr) {
243|             passed = false;
244|         }
245|     } catch (...) {
246|         passed = false;
247|     }
248|     printTestResult("RBT Search", passed);
249| }
250|
251| void testRBTToString() {
252|     bool passed = true;
253|     try {
254|         RBT tree;
255|         List list1; list1.push("банан");
256|         List list2; list2.push("яблоко");
257|         List list3; list3.push("вишня");
258|
259|         tree.insert(Pair("banana", list1));
260|         tree.insert(Pair("apple", list2));
261|         tree.insert(Pair("cherry", list3));
262|
263|         std::string result = tree.makeTreeToString();
264|         if (result.empty() ||
265|             result.find("apple") == std::string::npos ||
266|             result.find("banana") == std::string::npos ||
267|             result.find("cherry") == std::string::npos) {
268|             passed = false;
269|         }
270|     } catch (...) {
271|         passed = false;
272|     }
273|     printTestResult("RBT ToString", passed);
274| }
275|
276| void testRBTExceptions() {
277|     bool passed = true;
278|     try {
279|         RBT tree;
280|
281|         List list; list.push("trans");
282|         tree.insert(Pair("word", list));
283|         try {
284|             tree.insert(Pair("word", list));
```

```
285|             passed = false;
286|         } catch (const DuplicateWord&) {
287|             // Ожидаемое исключение
288|         } catch (...) {
289|             passed = false;
290|         }
291|
292|         Node* node = tree.search(tree.getRoot(), "word");
293|         try {
294|             tree.setColor(node, static_cast<Color>(2));
295|             passed = false;
296|         } catch (const WrongColor&) {
297|             // Ожидаемое исключение
298|         } catch (...) {
299|             passed = false;
300|         }
301|     } catch (...) {
302|         passed = false;
303|     }
304|     printTestResult("RBT Exceptions", passed);
305| }
306|
307| void testDictInsert() {
308|     bool passed = true;
309|     std::string testE = "apple";
310|     std::string testR = "яблоко";
311|     std::string testRR = "книга; литература";
312|     try {
313|         Dict dict;
314|
315|         dict.insert(testE, testR);
316|         if (dict.findTranslationByWord("apple").convertTranslationsToString() !=
"яблоко")
317|             passed = false;
318|
319|         try {
320|             dict.insert(testE, testR);
321|             passed = false;
322|         } catch (const DuplicateWord&) {
323|             // Ожидаемое исключение
324|         } catch (...) {
325|             passed = false;
326|         }
327|
328|     } catch (...) {
329|         passed = false;
330|     }
331|     printTestResult("Dict Insert", passed);
332| }
333|
334| void testDictExceptions() {
335|     bool passed = true;
336|     std::string testE = "apple";
337|     std::string testR = "яблоко";
338|     std::string testEmpty = "";
339|     std::string testStrange = ";;;";
340|     try {
341|         Dict dict;
342|
343|         try {
344|             dict.insert(testR, testE);
345|             passed = false;
```

```
346|         } catch (const InvalidWordLanguage&) {
347|             // Ожидаемое исключение
348|         } catch (...) {
349|             passed = false;
350|         }
351|
352|         try {
353|             dict.insert(testE, testE);
354|             passed = false;
355|         } catch (const InvalidTranslationLanguage&) {
356|             // Ожидаемое исключение
357|         } catch (...) {
358|             passed = false;
359|         }
360|
361|         try {
362|             dict.insert(testEmpty, testR);
363|             passed = false;
364|         } catch (const InvalidWordLanguage&) {
365|             // Ожидаемое исключение
366|         } catch (...) {
367|             passed = false;
368|         }
369|
370|         try {
371|             dict.insert(testE, testEmpty);
372|             passed = false;
373|         } catch (const EmptyValue&) {
374|             // Ожидаемое исключение
375|         } catch (...) {
376|             passed = false;
377|         }
378|
379|         try {
380|             dict.insert(testE, testStrange);
381|             passed = false;
382|         } catch (const EmptyValue&) {
383|             // Ожидаемое исключение
384|         } catch (...) {
385|             passed = false;
386|         }
387|
388|     } catch (...) {
389|         passed = false;
390|     }
391|     printTestResult("Dict Exceptions", passed);
392| }
393|
394| void runAllTests() {
395|     initializeEmptyJsonFile(DATAPATH);
396|
397|     testListConstructors();
398|     testListPush();
399|     testListRemove();
400|     testListToString();
401|
402|     testPairConstructors();
403|     testPairOperators();
404|     testPairOutputOperator();
405|
406|     testRBTConstructors();
407|     testRBTInsert();
```

```
408|     testRBTRemove();
409|     testRBTSearch();
410|     testRBTToString();
411|     testRBTExceptions();
412|
413|     testDictInsert();
414|     testDictExceptions();
415|
416|     initializeEmptyJsonFile(DATAPATH);
417| }
418|
```

Dictionary.json (часть)

```
1 |
2 | {
3 |     "access": ["доступ"],
4 |     "cat": ["кот", "котэ", "кошка"],
5 |     "personal": ["персональный", "личностный", "личной"],
6 |     "perspective": ["перспектива", "перспективный"],
7 |     "sip": ["глоток"]
8 | }
9 |
```

Приложение 2. Протокол отладки

```
Доступные команды:
1. TEST - Для запуска автоматического тестирования программы.
2. USER- Добавить новое слово и его переводы в словарь.
3. EXIT - Завершение работы.
Ваш выбор:test

[PASSED] :-D List Constructors
[PASSED] :-D List Push
[PASSED] :-D List Remove
[PASSED] :-D List ToString
[PASSED] :-D Pair Constructors
[PASSED] :-D Pair Operators
[PASSED] :-D Pair Output Operator
[PASSED] :-D RBT Constructors
[PASSED] :-D RBT Insert
[PASSED] :-D RBT Remove
[PASSED] :-D RBT Search
[PASSED] :-D RBT ToString
[PASSED] :-D RBT Exceptions
[PASSED] :-D Dict Insert
[PASSED] :-D Dict Exceptions
```

Рисунок 2 Работа автотестов

```
Введите команду:insert

Введите английское слово:слово

Введите перевод(ы), разделяя их точкой с запятой ';':translation

Ошибка: WARNING: Word should be in English.
```

Рисунок 3. Слово на русском языке

```
Введите команду:insert

Введите английское слово:cat

Введите перевод(ы), разделяя их точкой с запятой ';':кот

Ошибка: WARNING: Word already exists!
```

Рисунок 4. Слово существует

```
Введите команду:remove

Введите слово для удаления:papaia

WARNING: Word not found in the tree

Попробуйте снова.
```

Рисунок 5. Слова не существует

```
Введите команду:addtranslation

Введите слово для добавления перевода:cat

Введите новый перевод:kiiten

WARNING: Translation should be in Russian.

Попробуйте снова.
```

Рисунок 6. Перевод не на русском

```
Введите команду:aUtOtRaNsLaTe

Введите текст для автоперевода (язык определяется автоматически):cat

Результат перевода: кот
```

Рисунок 7. Работа изменения к нижнему регистру

```
Введите команду:tree

└── [B] cat: кот
    └── [R] test: автотест; тест; тестирование

Введите команду:print

cat - кот
test - автотест; тест; тестирование
```

Рисунок 8. Вывод словаря

```
Введите команду:print

cat - кот
test - автотест; тест; тестирование
Введите команду:print

cat - кот
test - автотест; тест; тестирование
Введите команду:removetranslation

Введите слово для удаления перевода:cat

Введите перевод для удаления:кот

Перевод удален, и слово полностью удалено из словаря!
```

Рисунок 9. Удаления одного перевода

Приложение 3. Использование JSON формата

MyJson разработан для организации чтения и записи словарных данных в формате, аналогичном JSON, без использования сторонних библиотек и стандартных контейнеров STL. Он обеспечивает базовую сериализацию и десериализацию словаря, представленного в виде пар "английское слово — список русских переводов", в текстовом файле.

Алгоритм разбора данных

Метод parse() выполняет последовательный синтаксический анализ текстового содержимого:

1. Пропуск пробельных символов (skipWhitespace);
2. Проверка корректного открытия ({) и итерация по ключам;
3. Чтение строкового ключа (parseString);
4. Разбор массива переводов [...] с добавлением в соответствующую структуру;
5. Обработка разделителей (запятых);
6. При необходимости — автоматическое увеличение ёмкости массива (grow()).