Правительство Российской Федерации

Федеральное государственное автономное образовательное

учреждение высшего образования

«Национальный исследовательский университет

«Высшая школа экономики»

Факультет компьютерных наук

Департамент программной инженерии

Дисциплина **«Алгоритмы и структуры данных»** (часть 2)

Учебный год 2015-2016, модуль 4

**Домашнее задание** (4 части)

Отчёт о выполненной работе

**Студент:** Ярных Роман Вячеславович

**Группа:** БПИ141 (2)

**Статус представляемой работы:**

Часть 1: выполнено полностью

Часть 2: выполнено полностью

Часть 3: выполнено полностью

Часть 4: не выполнена

Москва 2016

**СОДЕРЖАНИЕ**

1. ЧАСТЬ №1 3

1.1 Постановка задачи 3

1.2 Краткий анализ задачи 3

1.3 Детализации реализации 4

1.4 Дополнительная функциональность 20

1.5 Заключение 20

2. ЧАСТЬ №2 21

2.1 Постановка задачи 21

2.2 Краткий анализ задачи 21

2.3 Детализации реализации 22

2.4 Дополнительная функциональность 29

2.5 Заключение 29

3. ЧАСТЬ №3 30

3.1 Постановка задачи 30

3.2 Краткий анализ задачи 30

3.3 Детализации реализации 31

3.4 Дополнительная функциональность 36

3.5 Заключение 36

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 37

# 1. ЧАСТЬ №1

## 1.1 Постановка задачи

Цель части №1 заключается в создании программы для поиска тех слов, прочитанных из файла, которые есть в исходном словаре, и тех, которые отсутствуют с использованием различных структур данных и алгоритмов и анализом эффективности их использования.

## 1.2 Краткий анализ задачи

Задача поиска схожих строк очень актуальна в области построения поисковых систем, анализа ДНК, нахождения грамматических ошибок и перевода текстов. Так как анализ текстов требует много времени и ресурсов для осуществления поиска строк, то необходимо изучить всевозможные алгоритмы и оценить их эффективность. В рамках данной части было реализовано четыре алгоритмы для поиска схожих строк: поиск на базе префиксных деревьев [2], поиск на базе PATRICIA деревьев [8], поиск на базе ассоциативного массива и поиск на базе хэш-таблиц [1].

Первый подход заключается в использовании хэш-таблиц [1]. Хэш-таблицы [1] представляют собой разновидность ассоциативного массива, в котором ключами являются значения хэш-функции, а в качестве значений может выступать любой тип, для которого возможно вычисление строки. Главное преимущество использования хэш-таблиц [1] заключается в том, что поиск строки в словаре занимает в среднем O(1). Библиотека STL [6] содержит реализацию контейнера на основе хэш-таблиц в виде класса unsorted\_set [4]. Класс unsorted\_set [4] использует закрытую адресацию для вставки, поиска и удаления значения из словаря. Однако стоит помнить, что в худшем случае сложность по времени составит O(n), если появятся коллизии. Вероятность же коллизии достаточно мала и оценивает как 1.0/std::numeric\_limits<size\_t>::max() [4], то есть для 32-х битных машин вероятность равна около , а для 64-х битных машин – .

Второй подход заключается в использовании ассоциативного массива, который реализуется через красно-черное дерево [7]. Красно-черные деревья [7] являются одним из видов двоичных деревьев, которые поддерживают самобалансировку. Поиск в среднем занимает O(log(n)), что хуже, чем при использовании хэш-таблиц, однако красно-черные деревья выигрывают в худшем случае, так сложность составляет O(log(n)) против O(n) у хэш-таблиц. Библиотека STL [1] содержит реализацию контейнера на базе КЧ-деревьев [7] в виде класса set.

Третий подход заключается в использовании префиксных деревьев. Префиксное дерево состоит из узлов, которые не содержат ключи, а являются метками в дереве. Поэтому любая строка по своей сути является набором меток, по котором можно пройти от корневого узла до листа. Каждый лист является пустым и служит только для того, чтобы сигнализировать о конце поиска. Сложность поиска, вставки и удаления в среднем, лучшем и худшем случае занимает O(k), где k – длина строки. Главное преимущество использования префиксных деревьев заключается в том, поиск производится достаточно быстро, причем скорость поиска не проседает на больших данных. Также важное отличие от других типов данных связано с особенностью потребления памяти. Так как префиксные деревья не хранят ключи, то по сравнению с хэш-таблицами и красно-черными деревьями они выигрывают по потреблению памяти у последних.

Четвертый подход является улучшенной версией третьего. По своей сути PATRICIA деревья [8] входят в разновидность сжатых префиксных деревьев. Сжатие достигается за счет использования строк в качестве ключей, а не отдельных символов. Данный подход позволяет уменьшить потребление памяти и ускорить процесс поиска.

Запишем сложность каждого из методов в таблицу:

|  |  |  |  |
| --- | --- | --- | --- |
| Метод | В лучшем случае | В худшем случае | В среднем |
| Хэш-таблица | O(1) | O(n) | O(1) |
| Ассоциативный массива на базе деревьев | O(log(n)) | O(log(n)) | O(log(n)) |
| Префиксные деревья | O(k), где k – длина слова | O(k), где k – длина слова | O(k), где k – длина слова |
| PATRICIA деревья | O(k), где k – длина слова | O(k), где k – длина слова | O(k), где k – длина слова |

## 1.3 Детализации реализации

Программа состоит из нескольких программных модулей:

1. main.cpp – чтение из файла, выполнение предложенного алгоритма поиска и вывод результата в консоль или файл;
2. AbstractDictionary.h – абстрактный класс, предоставляющий интерфейс для работы с словарем;
3. HashDictionary.h – класс, реализующий словарь на базе хэш-таблиц;
4. SetDictionary.h – класс, реализующий словарь на базе ассоциативного массива;
5. TrieDictionary.h – класс. реализующий словарь на базе префиксного дерева;
6. PTrieDictionary.h – класс, реализующий словарь на базе PATRICIA дерева.

Класс AbstractDictionary и его наследники отвечают за загрузку в память словаря, поиска слов, которые присутствуют в исходном словаре, из текста, прочитанного из файла. Опишем методы и поля классов:

|  |  |
| --- | --- |
| Dictionary | |
| Методы | void loadFromFile(istream&) – загружает словарь из входящего потока |
| void addWord(string word) – добавляет слово в словарь |
| pair<vector<string>, vector<string>> findOccurrences(istream&) – ищет все слова в файле, которые встречаются в словаре и те, которые нет |
| bool isIncluded(string word) – проверяет, есть ли слово в словаре |
| HashDictionary | |
| Поля | unsorted\_set<string> dict\_ – словарь на базе хэш-таблицы |
| Методы | void loadFromFile(istream&) – загружает словарь из входящего потока |
| void addWord(string word) – добавляет слово в словарь |
| pair<vector<string>, vector<string>> findOccurrences(istream&) – ищет все слова в файле, которые встречаются в словаре и те, которые нет |
| bool isIncluded(string word) – проверяет, есть ли слово в словаре |
| HashDictionary() – конструктор |
| ~HashDictionary() – дескруктор |
| SetDictionary | |
| Поля | set<string> dict\_ – словарь на базе ассоциативного массива |
| Методы | void loadFromFile(istream&) – загружает словарь из входящего потока |
| void addWord(string word) – добавляет слово в словарь |
| pair<vector<string>, vector<string>> findOccurrences(istream&) – ищет все слова в файле, которые встречаются в словаре и те, которые нет |
| bool isIncluded(string word) – проверяет, есть ли слово в словаре |
| SetDictionary() – конструктор |
| ~SetDictionary() – дескруктор |
| TrieDictionary | |
| Поля | Node head\_ – корневой узел префиксного дерева |
| Методы | void loadFromFile(istream&) – загружает словарь из входящего потока |
| void addWord(string word) – добавляет слово в словарь |
| pair<vector<string>, vector<string>> findOccurrences(istream&) – ищет все слова в файле, которые встречаются в словаре и те, которые нет |
| bool isIncluded(string word) – проверяет, есть ли слово в словаре |
| TrieDictionary() – конструктор |
| ~TrieDictionary() – дескруктор |
| PTrieDictionary | |
| Поля | Node head\_ – корневой узел PATRICIA дерева |
| Методы | void loadFromFile(istream&) – загружает словарь из входящего потока |
| void addWord(string word) – добавляет слово в словарь |
| pair<vector<string>, vector<string>> findOccurrences(istream&) – ищет все слова в файле, которые встречаются в словаре и те, которые нет |
| bool isIncluded(string word) – проверяет, есть ли слово в словаре |
| PTrieDictionary() – конструктор |
| ~PTrieDictionary() – дескруктор |

Покажем взаимосвязь классов в программе с помощью UML диаграммы:



Приведем исходные коды реализации. Исходный код класса Dictionary:

#pragma once  
#ifndef **CHW\_2\_ABSTRACTDICTIONARY\_H**#define **CHW\_2\_ABSTRACTDICTIONARY\_H**#include **<iostream>**#include **<vector>  
  
using namespace** std;  
  
**class** Dictionary {  
**public**:  
 **virtual void** loadDictFromFile(istream& input);  
 **virtual void** addWord(string word) = 0;  
 **virtual** pair<vector<string>, vector<string>> findOccurrences(istream& input);  
 **virtual bool** isIncluded(string word) = 0;  
 **virtual** ~Dictionary() = 0;  
};  
  
**void** Dictionary::loadDictFromFile(istream &input) {  
 string temp\_;  
 **while**(getline(input, temp\_)){  
 addWord(temp\_);  
 }  
}  
  
pair<vector<string>, vector<string>> Dictionary::findOccurrences(istream &input) {  
 string temp\_;  
 vector<string> occurrences\_;  
 vector<string> notIncluded\_;  
 **while**(input >> temp\_){  
 stringstream ss;  
 **bool** lastNonAlpha = **false**;  
 **for**(string::iterator it = temp\_.begin(); it != temp\_.end(); it++){  
 **if**((\*it < **'a'** || \*it > **'z'**) && !lastNonAlpha) {  
 string word\_ = ss.str();  
 **if** (isIncluded(word\_)) {  
 occurrences\_.push\_back(word\_);  
 }  
 **else**{  
 notIncluded\_.push\_back(word\_);  
 }  
 ss.str(**""**);  
 lastNonAlpha = **true**;  
 }  
 **else if**(isalpha(\*it)){  
 lastNonAlpha = **false**;  
 ss << (\*it);  
 }  
 }  
 **if**(ss.str().size() > 0){  
 string word\_ = ss.str();  
 **if** (isIncluded(word\_)) {  
 occurrences\_.push\_back(word\_);  
 }  
 **else**{  
 notIncluded\_.push\_back(word\_);  
 }  
 }  
 }  
 **return** pair<vector<string>, vector<string>>(occurrences\_, notIncluded\_);  
}  
  
Dictionary::~Dictionary(){  
  
}  
  
#endif *//CHW\_2\_ABSTRACTDICTIONARY\_H*

Исходный код класса HashDictionary:

#pragma once  
  
#ifndef **CHW\_2\_HASHDICTIONARY\_H**#define **CHW\_2\_HASHDICTIONARY\_H**#include **<unordered\_set>**#include **<sstream>**#include **<set>**#include **<map>**#include **"IDictionary.h"***/\*  
 \* Dictionary based on hash tables  
 \*/***class** HashDictionary : **public** IDictionary {  
**public**:  
 **void** addWord(string word);  
 **bool** isIncluded(string word);  
 set<string> getWords();  
 HashDictionary();  
 **virtual** ~HashDictionary();  
**private**:  
 unordered\_set<string> \*dict\_;  
};  
  
*/\*  
 \* Add word to dictionary  
 \*/***void** HashDictionary::addWord(string word) {  
 dict\_->insert(word);  
}  
  
*/\*  
 \* Check if word exists in dictionary  
 \*/***bool** HashDictionary::isIncluded(string word){  
 **return** dict\_->find(word) != dict\_->end();  
}  
  
*/\*  
 \* Retrieve all words from dictionary  
 \*/*set<string> HashDictionary::getWords() {  
 set<string> words\_;  
 **for**(unordered\_set<string>::iterator it = dict\_->begin(); it != dict\_->end(); it++)  
 words\_.insert(\*it);  
 **return** words\_;  
}  
  
*/\*  
 \* Construct an instance of dictionary  
 \*/*HashDictionary::HashDictionary() {  
 dict\_ = **new** unordered\_set<string>();  
}  
  
*/\*  
 \* Delete instance of dictionary  
 \*/*HashDictionary::~HashDictionary() {  
 **delete** dict\_;  
}  
  
#endif *//CHW\_2\_HASHDICTIONARY\_H*

Исходный код класса SetDictionary:

#pragma once  
  
#ifndef **CHW\_2\_SETDICTIONARY\_H**#define **CHW\_2\_SETDICTIONARY\_H**#include **<set>**#include **"IDictionary.h"***/\*  
 \* Dictionary based on red-n-black trees  
 \*/***class** SetDictionary : **public** IDictionary {  
  
**public**:  
  
 **virtual void** addWord(string word);  
  
 **virtual bool** isIncluded(string word);  
  
 **virtual** set<string> getWords();  
  
 SetDictionary();  
  
 **virtual** ~SetDictionary();  
  
**private**:  
  
 set<string> \*dict\_;  
};  
  
*/\*  
 \* Add word to dictionary  
 \*/***void** SetDictionary::addWord(string word) {  
 dict\_->insert(word);  
}  
  
*/\*  
 \* Check if word exists in dictionary  
 \*/***bool** SetDictionary::isIncluded(string word) {  
 **return** dict\_->find(word) != dict\_->end();  
}  
  
*/\*  
 \* Retrieve all words from dictionary  
 \*/*set<string> SetDictionary::getWords() {  
 **return** set<string>(\*dict\_);  
}  
  
*/\*  
 \* Construct an instance of dictionary  
 \*/*SetDictionary::SetDictionary() {  
 dict\_ = **new** set<string>;  
}  
  
*/\*  
 \* Delete instance of dictionary  
 \*/*SetDictionary::~SetDictionary() {  
 **delete** dict\_;  
}  
  
#endif *//CHW\_2\_SETDICTIONARY\_H*

Исходный код класса TrieDictionary:

#pragma once  
  
#ifndef **CHW\_2\_TRIEDICTIONARY\_H**#define **CHW\_2\_TRIEDICTIONARY\_H**#include **"IDictionary.h"**#include **"../tree/Trie.h"***/\*  
 \* Dictionary based on prefix trees  
 \*/***class** TrieDictionary : **public** IDictionary {  
  
**public**:  
 **virtual** set<string> getWords();  
  
 **void** addWord(string word);  
  
 **bool** isIncluded(string word);  
  
 TrieDictionary();  
  
 **virtual** ~TrieDictionary();  
  
**private**:  
 Trie \*trie\_;  
};  
  
*/\*  
 \* Add word to dictionary  
 \*/***void** TrieDictionary::addWord(string word) {  
 trie\_->add(word);  
}  
  
*/\*  
 \* Check if word exists in dictionary  
 \*/***bool** TrieDictionary::isIncluded(string word) {  
 **return** trie\_->contains(word);  
}  
  
*/\*  
 \* Retrieve all words from dictionary  
 \*/*set<string> TrieDictionary::getWords() {  
 **return** set<string>();  
}  
  
*/\*  
 \* Construct an instance of dictionary  
 \*/*TrieDictionary::TrieDictionary() {  
 trie\_ = **new** Trie;  
}  
  
*/\*  
 \* Delete instance  
 \*/*TrieDictionary::~TrieDictionary() {  
 **delete** trie\_;  
}  
  
#endif *//CHW\_2\_TRIEDICTIONARY\_H*

Исходный код класса Trie:

#pragma once  
  
#ifndef **CHW\_2\_TRIE\_H**#define **CHW\_2\_TRIE\_H**#define **ALPH\_N** 26  
  
#include **<set>**#include **"ITree.h"***/\*  
 \* Prefix tree (trie)  
 \*/***class** Trie : **public** ITree<string> {  
**private**:  
 **struct** Node{  
 **char** key;  
 Node\* children[**ALPH\_N**];  
 };  
 Node \*head\_;  
  
**public**:  
 **void** add(string key);  
 **bool** contains(string key);  
 Trie();  
 **virtual** ~Trie();  
  
**private**:  
 **void** \_clear(Node\* node);  
};  
  
*/\*  
 \* Construct an instance of trie  
 \*/*Trie::Trie() {  
 head\_ = **new** Node;  
}  
  
*/\*  
 \* Add key into tree  
 \*/***void** Trie::add(string key) {  
 Node\* temp\_ = head\_;  
 **for**(string::iterator it = key.begin(); it != key.end(); it++){  
 Node\* child\_ = temp\_->children[\*it - **'a'**];  
 **if**(!child\_){  
 child\_ = **new** Node;  
 child\_->key = \*it;  
 temp\_->children[\*it - **'a'**] = child\_;  
 }  
 temp\_ = child\_;  
 }  
}  
  
*/\*  
 \* Check if key exists in tree  
 \*/***bool** Trie::contains(string key) {  
 **bool** inc\_ = **true**;  
 Node \*temp\_ = head\_;  
 **for**(string::iterator it = key.begin(); it != key.end(); it++){  
 **if**(!temp\_->children[\*it - **'a'**]){  
 inc\_ = **false**;  
 **break**;  
 }  
 **else**{  
 temp\_ = temp\_->children[\*it - **'a'**];  
 }  
 }  
 **return** inc\_;  
}  
  
*/\*  
 \* Clear tree  
 \*/***void** Trie::\_clear(Node\* node){  
 **if**(!node)  
 **return**;  
 **for**(**int** i = 0; i < **ALPH\_N**; i++){  
 **if**(!node->children[i]) {  
 \_clear(node->children[i]);  
 **delete** node->children[i];  
 }  
 }  
}  
  
*/\*  
 \* Delete instance  
 \*/*Trie::~Trie() {  
 Node\* temp\_ = head\_;  
 \_clear(temp\_);  
}  
  
#endif *//CHW\_2\_TRIE\_H*

Исходный код класса PTrie:

#pragma once  
  
#ifndef **CHW\_2\_PTREEDICTIONARY\_H**#define **CHW\_2\_PTREEDICTIONARY\_H**#include **<set>**#include **"IDictionary.h"**#include **"../tree/PTrie.h"***/\*  
 \* Compressed radix (prefix) tree  
 \*/***class** PTrieDictionary : **public** IDictionary {  
  
**public**:  
  
 **virtual** set<string> getWords();  
  
 PTrieDictionary();  
 **virtual void** addWord(string word);  
  
 **virtual bool** isIncluded(string word);  
  
 **virtual** ~PTrieDictionary();  
  
**private**:  
 PTrie \*trie\_;  
};  
  
*/\*  
 \* Construct an instance of PATRICIA trie  
 \*/*PTrieDictionary::PTrieDictionary() {  
 trie\_ = **new** PTrie;  
}  
  
*/\*  
 \* Add word to dictionary  
 \*/***void** PTrieDictionary::addWord(string word) {  
 trie\_->add(word);  
}  
  
*/\*  
 \* Check if word exists in dictionary  
 \*/***bool** PTrieDictionary::isIncluded(string word) {  
 **return** trie\_->contains(word);  
}  
  
*/\*  
 \* Retrieve all words from dictionary  
 \*/*set<string> PTrieDictionary::getWords() {  
 **return** set<string>();  
}  
  
*/\*  
 \* Delete instance  
 \*/*PTrieDictionary::~PTrieDictionary() {  
 **delete** trie\_;  
}  
  
#endif *//CHW\_2\_PTREEDICTIONARY\_H*

Исходный код класса PTrie:

#pragma once  
  
#ifndef **CHW\_2\_PTRIE\_H**#define **CHW\_2\_PTRIE\_H**#include **"ITree.h"**#include **"PTNode.h"***/\*  
 \* PATRICIA trie  
 \*/***class** PTrie : **public** ITree<string>{  
**private**:  
 PTNode \*head\_;  
  
**public**:  
 **virtual void** add(string key);  
 **virtual bool** contains(string key);  
 **virtual** ~PTrie();  
  
**private**:  
 **void** \_clear(PTNode \* node);  
 PTNode \* \_find(PTNode \* node, string str);  
 **unsigned long** \_prefixLen(string s1, string s2);  
 PTNode \* \_insert(PTNode \* node, string str);  
 **void** \_cutNode(PTNode \* node, **unsigned long** index);  
};  
  
*/\*  
 \* Add key into tree  
 \*/***void** PTrie::add(string key) {  
 head\_ = \_insert(head\_, key);  
}  
  
*/\*  
 \* Check if key exists in tree  
 \*/***bool** PTrie::contains(string key) {  
 **return** \_find(head\_, key) != **nullptr**;  
}  
  
*/\*  
 \* Clear tree  
 \*/***void** PTrie::\_clear(PTNode \*node) {  
 **if**(!node)  
 **return**;  
 PTNode \* olderChild = node->olderChild;  
 \_clear(olderChild);  
 PTNode \* sister = node->sister;  
 **delete** node;  
 \_clear(sister);  
}  
  
*/\*  
 \* Find node with specific key  
 \*/*PTNode \* PTrie::\_find(PTNode \*node, string str){  
 **if**(!node)  
 **return nullptr**;  
 **unsigned long** pfl\_ = \_prefixLen(str, node->key);  
 **if**(pfl\_ == str.length())  
 **return** node;  
 **else if**(!pfl\_)  
 **return** \_find(node->sister, str);  
 **else if**(pfl\_ == node->key.length())  
 **return** \_find(node->olderChild, str.substr(pfl\_, str.length() - pfl\_));  
 **return nullptr**;  
}  
  
*/\*  
 \* Compute length of largest common prefix  
 \*/***unsigned long** PTrie::\_prefixLen(string s1, string s2) {  
 **for**(**unsigned long** k = 0; k < s1.length(); k++)  
 **if**(k == s2.length() || (s1[k] != s2[k]))  
 **return** k;  
 **return** s1.length();  
}  
  
*/\*  
 \* Insert a node into tree  
 \*/*PTNode \* PTrie::\_insert(PTNode \*node, string str) {  
 **if**(!node)  
 **return new** PTNode(str, **nullptr**, **nullptr**);  
 **unsigned long** pfl\_ = \_prefixLen(str, node->key);  
 **if**(pfl\_ == 0)  
 node->sister = \_insert(node->sister, str);  
 **else if**(pfl\_ < str.length()){  
 **if**(pfl\_ < node->key.length())  
 \_cutNode(node, pfl\_);  
 node->olderChild = \_insert(node->olderChild, str.substr(pfl\_, str.length() - pfl\_));  
 }  
 **return** node;  
}  
  
*/\*  
 \* Split node into two new ones  
 \*/***void** PTrie::\_cutNode(PTNode \*node, **unsigned long** index) {  
 PTNode \*newChild\_ = **new** PTNode(node->key.substr(index, node->key.length() - index), **nullptr**, node->olderChild);  
 node->key = node->key.substr(0, index);  
 node->olderChild = newChild\_;  
}  
  
*/\*  
 \* Delete instance  
 \*/*PTrie::~PTrie() {  
 \_clear(head\_);  
}  
  
#endif *//CHW\_2\_PTRIE\_H*

## 1.4 Дополнительная функциональность

Дополнительная функциональность реализации отсутствует.

## 1.5 Заключение

В рамках первой части нами было проведена оценка эффективности работы предложенных методов с помощью теоретического математического аппарата теории алгоритмов и разработана программа, решающая поставленные задачи с использованием этих методов.

Было выявлено, что все три метода, а именно префиксные деревья, красно-черные деревья и PATRICIA деревья, работают во всех случаях за одно время. В противоположенность этому хэш-таблицы ввиду ненулевой вероятности возникновения коллизий могут привести к ухудшению сложности. Однако вероятность возникновения коллизии достаточно мала. Согласно официальной документации по unordered\_set вероятность коллизии составляет 1.0/std::numeric\_limits<size\_t>::max(), то есть для 32-х битных машин вероятность равна около , а для 64-х битных машин – .

# 2. ЧАСТЬ №2

## 2.1 Постановка задачи

Цель части №2 заключается в написании программного кода для оценки эффективности использования различных комбинаций контейнеров и алгоритмов для поиска слов, отличных на один символ и имеющих один и тот же размер.

## 2.2 Краткий анализ задачи

В соответствии с целью были поставлены следующие задачи:

1. разработать алгоритмы (методы) для поиска слов, отличающихся на один символ;
2. оценить эффективность использованных алгоритмов и контейнеров;
3. сделать выводы по поводу использования данных алгоритмов на практике.

Для решения поставленных задач было разработано три метода поиска слов:

1. наивный поиск с использованием std::map<string, vector<string>>;
2. улучшенный поиск с использованием std::map<string, vector<string>> и предварительной группировкой слов по их длине;
3. поиск с предварительной группировкой слов по длине, а также по общему префиксу и суффиксу.

Первый метод заключается в наивном поиске слов, путем прохода по всему массиву и сравниванию одного слова с другим. В случае, если одно слово может быть получено путем изменения буквы в другом, то для первого в список заносится второе. Данный метод имеет сложность в лучшем худшем и среднем случаях , что является, несомненно, самым плохим вариантом поиска.

Второй метод имеет асимптотически такую же сложность, что и первый, но при его использовании улучшается коэффициент при за счет уменьшения числа сравнений путем группировки слов по длине. При таком методе слова сравниваются только в своей группе и имеют одну и ту же длину, что позволяет не производить заведомо лишние сравнения. В общем, сложность второго подхода равна , где – время группировки слов по длине. Также важно отметить, что для второго метода требуется дополнительной памяти.

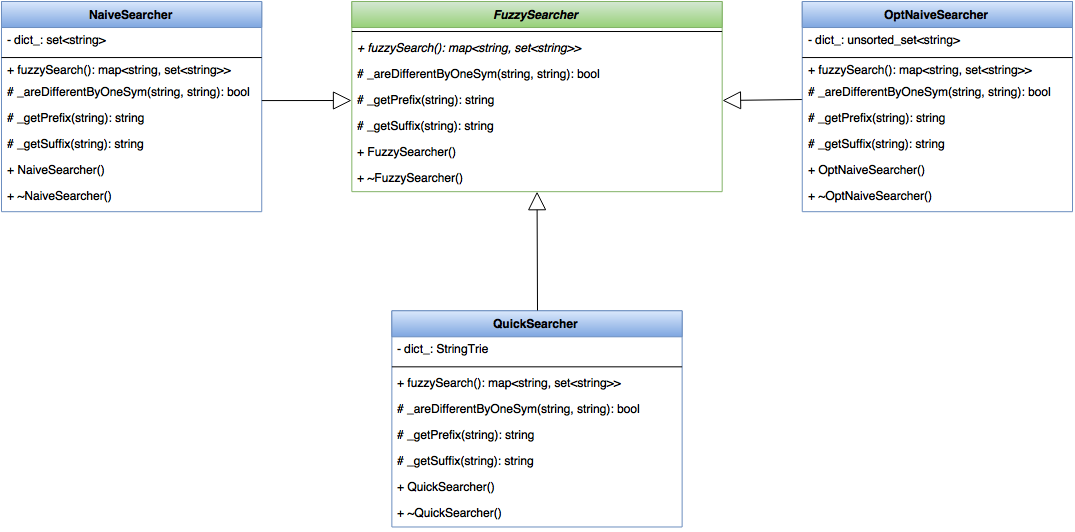
Третий вариант улучшает изначальный наивный поиск с помощью группировки слов не только по размеру, но и общему префиксу и суффиксу. Если для группировки слов с общим префиксом и суффиксом используется std::set<string>, то сложность по времени в худшем, лучшем и среднем случае составит , где – размер префиксного или суффиксного массива. Заметим, что если все слова имеют один префикс или суффикс, при условии , то , следовательно, сложность метода составит . При использовании std::unordered\_set<string> сложность метода в лучшем и среднем случае составит , в худшем – . Как и во втором методе, потребуется дополнительной памяти.

## 2.3 Детализации реализации

Было решено перенести функциональность поиска схожих строк в новые классы, не входящие в иерархию классов, ответственных за реализацию работы словарей. С этой целью были разработаны классы FuzzySearcher, NaiveSearcher, OptNaiveSearcher и QuickSearcher. Вершиной иерархии выступает FuzzySearcher, определяющий интерфейс методов для конечного клиента. Классы NaiveSearcher, OptNaiveSearcher и QuickSearcher реализует каждый из предложенных 3 методов. Опишем структуру классов:

|  |  |
| --- | --- |
| FuzzySearcher | |
| Поля | set<string> \*dict\_ – исходный словарь |
| Методы | FuzzySearcher() – конструктор по умолчанию |
| FuzzySearcher(set<string>) – конструктор, принимающий в качестве аргумента исходный словарь |
| ~FuzzySearcher() – деструктор |
| map<string, set<string>> fuzzySearch() – поиск строк, отличных на один символ |
| bool \_areDifferentByOneSym(string, string) – сравнение строк |
| string \_getPrefix(string) – получение префикса строки |
| string \_getSuffix(string) – получение суффикса строки |
| NaiveSearcher | |
| Поля | set<string> \*dict\_ – исходный словарь |
| Методы | NaiveSearcher() – конструктор по умолчанию |
| NaiveSearcher(set<string>) – конструктор, принимающий в качестве аргумента исходный словарь |
| ~NaiveSearcher() – деструктор |
| map<string, set<string>> fuzzySearch() – поиск строк, отличных на один символ |
| bool \_areDifferentByOneSym(string, string) – сравнение строк |
| string \_getPrefix(string) – получение префикса строки |
| string \_getSuffix(string) – получение суффикса строки |
| OptNaiveSearcher | |
| Поля | set<string> \*dict\_ – исходный словарь |
| Методы | OptNaiveSearcher() – конструктор по умолчанию |
| OptNaiveSearcher(set<string>) – конструктор, принимающий в качестве аргумента исходный словарь |
| ~OptNaiveSearcher() – деструктор |
| map<string, set<string>> fuzzySearch() – поиск строк, отличных на один символ |
| bool \_areDifferentByOneSym(string, string) – сравнение строк |
| string \_getPrefix(string) – получение префикса строки |
| string \_getSuffix(string) – получение суффикса строки |
| QuickSearcher | |
| Поля | set<string> \*dict\_ – исходный словарь |
| Методы | QuickSearcher() – конструктор по умолчанию |
| QuickSearcher(set<string>) – конструктор, принимающий в качестве аргумента исходный словарь |
| ~QuickSearcher() – деструктор |
| map<string, set<string>> fuzzySearch() – поиск строк, отличных на один символ |
| bool \_areDifferentByOneSym(string, string) – сравнение строк |
| string \_getPrefix(string) – получение префикса строки |
| string \_getSuffix(string) – получение суффикса строки |

Покажем отношения между классами с помощью UML диаграмм:



Приведем исходные коды классов. Исходный код класса FuzzySearcher:

#pragma once  
#ifndef **CHW\_2\_FUZZYSEARCHER\_H**#define **CHW\_2\_FUZZYSEARCHER\_H**#include **<iostream>**#include **<string>**#include **<set>  
  
using namespace** std;  
  
**class** FuzzySearcher {  
**public**:  
 FuzzySearcher();  
 FuzzySearcher(set<string> ws);  
 **virtual** ~FuzzySearcher();  
 **virtual** map<string, set<string>> fuzzySearch() = 0;  
  
**protected**:  
 **int** \_hamming\_distance(string a, string b);  
 **bool** \_areDifferentByOneSym(string a, string b);  
 string \_getPrefix(string s);  
 string \_getSuffix(string s);  
 set<string> \*dict\_;  
};  
  
FuzzySearcher::FuzzySearcher() {  
  
}  
  
FuzzySearcher::FuzzySearcher(set<string> ws) {  
 dict\_ = **new** set<string>(move(ws));  
}  
  
FuzzySearcher::~FuzzySearcher() {  
 **delete** dict\_;  
}  
  
**int** FuzzySearcher::\_hamming\_distance(string a, string b) {  
 **const char**\* a\_ = a.c\_str();  
 **const char**\* b\_ = b.c\_str();  
 **int** d\_ = 0;  
 **for**(**int** i = 0; i < a.length(); i++){  
 d\_ += ((a\_[i] - b\_[i]) == 0 ? 0 : 1);  
 }  
 **return** d\_;  
}  
  
**bool** FuzzySearcher::\_areDifferentByOneSym(string a, string b) {  
 **bool** dif\_ = **false**;  
 **for**(**int** i = 0; i < a.length(); i++){  
 **if**(a[i] != b[i]){  
 **if**(dif\_) {  
 dif\_ = **false**;  
 **break**;  
 }  
 **else** dif\_ = **true**;  
 }  
 }  
 **return** dif\_;  
}  
  
string FuzzySearcher::\_getPrefix(string s) {  
 **int** prefixEnd\_ = (**int**)floor(s.length() / 2);  
 **return** s.substr(0, (**unsigned long**)prefixEnd\_ + 1);  
}  
  
string FuzzySearcher::\_getSuffix(string s) {  
 **int** prefixEnd\_ = (**int**)floor(s.length() / 2);  
 **return** s.substr((**unsigned long**)prefixEnd\_ + 1, s.length() - prefixEnd\_ - 1);  
}  
  
  
#endif *//CHW\_2\_FUZZYSEARCHER\_H*

Исходный код NaiveSearcher:

#ifndef **CHW\_2\_NAIVYSEARCHER\_H**#define **CHW\_2\_NAIVYSEARCHER\_H**#include **"FuzzySearcher.h"**#include **<map>  
  
class** NaiveSearcher : **public** FuzzySearcher{  
  
**public**:  
 NaiveSearcher(set<string> s) : FuzzySearcher(s){}  
 **virtual** map<string, set<string>> fuzzySearch();  
};  
  
  
#endif *//CHW\_2\_NAIVYSEARCHER\_H*map<string, set<string>> NaiveSearcher::fuzzySearch() {  
 map<string, set<string>> fsmw\_;  
 **for**(**auto** it = dict\_->begin(); it != dict\_->end(); it++){  
 set<string> mw\_;  
 string current\_ = \*it;  
 **for**(**auto** it1 = dict\_->begin(); it1 != dict\_->end(); it1++){  
 string other\_ = \*it1;  
 **if**(other\_.length() != current\_.length())  
 **continue**;  
 **if**(\_areDifferentByOneSym(current\_, other\_))  
 mw\_.insert(other\_);  
 }  
 fsmw\_.insert(pair<string, set<string>>(current\_, mw\_));  
 }  
 **return** fsmw\_;  
}

Исходный код OptNaiveSearcher:

#ifndef **CHW\_2\_OPTNAIVESEARCHER\_H**#define **CHW\_2\_OPTNAIVESEARCHER\_H**#include **"FuzzySearcher.h"**#include **<map>  
  
class** OptNaiveSearcher : **public** FuzzySearcher{  
  
**public**:  
 OptNaiveSearcher(set<string> s) : FuzzySearcher(s){}  
 **virtual** map<string, set<string>> fuzzySearch();  
};  
  
  
#endif *//CHW\_2\_OPTNAIVESEARCHER\_H*map<string, set<string>> OptNaiveSearcher::fuzzySearch() {  
 map<string, set<string>> fsmw\_;  
 map<**unsigned long**, set<string>> lg\_;  
 **for**(**auto** it = dict\_->begin(); it != dict\_->end(); it++){  
 **if**(lg\_.find(it->length()) != lg\_.end()){  
 lg\_[it->length()].insert(\*it);  
 }  
 **else**{  
 set<string> clws\_;  
 clws\_.insert(\*it);  
 lg\_.insert(pair<**unsigned long**, set<string>>(it->length(), clws\_));  
 }  
 }  
 **for**(**auto** lg\_it = lg\_.begin(); lg\_it != lg\_.end(); lg\_it++){  
 **for**(set<string>::iterator word\_ = lg\_it->second.begin(); word\_ != lg\_it->second.end(); word\_++){  
 set<string> mw\_;  
 **for**(set<string>::iterator oword\_ = lg\_it->second.begin(); oword\_ != lg\_it->second.end(); oword\_++){  
 **if**(\_areDifferentByOneSym(\*word\_, \*oword\_))  
 mw\_.insert(\*oword\_);  
 }  
 fsmw\_.insert(pair<string, set<string>>(\*word\_, mw\_));  
 }  
 }  
 **return** fsmw\_;  
}

Исходный код QuickSearcher:

#ifndef **CHW\_2\_QUICKSEARCHER\_H**#define **CHW\_2\_QUICKSEARCHER\_H**#include **"FuzzySearcher.h"**#include **<map>**#include **<unordered\_set>**#include **<unordered\_map>  
  
class** QuickSearcher : **public** FuzzySearcher{  
  
**public**:  
 QuickSearcher(set<string> s) : FuzzySearcher(s) {}  
 **virtual** map<string, set<string>> fuzzySearch();  
};  
  
  
#endif *//CHW\_2\_QUICKSEARCHER\_H*map<string, set<string>> QuickSearcher::fuzzySearch() {  
 map<string, set<string>> fsmw\_;  
 map<**unsigned long**, set<string>> lg\_;  
 **for**(**auto** it = dict\_->begin(); it != dict\_->end(); it++){  
 **if**(lg\_.find(it->length()) != lg\_.end()){  
 lg\_[it->length()].insert(\*it);  
 }  
 **else**{  
 set<string> clws\_;  
 clws\_.insert(\*it);  
 lg\_.insert(pair<**unsigned long**, set<string>>(it->length(), clws\_));  
 }  
 }  
 **for**(**auto** lg\_it = lg\_.begin(); lg\_it != lg\_.end(); lg\_it++){  
 unordered\_map<string, unordered\_set<string>> prefixMap\_;  
 unordered\_map<string, unordered\_set<string>> suffixMap\_;  
 **for**(**auto** word\_ = lg\_it->second.begin(); word\_ != lg\_it->second.end(); word\_++){  
 string prefix\_ = \_getPrefix(\*word\_);  
 string suffix\_ = \_getSuffix(\*word\_);  
 **if**(prefixMap\_.find(prefix\_) != prefixMap\_.end()){  
 prefixMap\_[prefix\_].insert(suffix\_);  
 }  
 **else**{  
 prefixMap\_[prefix\_] = unordered\_set<string>();  
 }  
  
 **if**(suffixMap\_.find(suffix\_) != suffixMap\_.end()){  
 suffixMap\_[suffix\_].insert(prefix\_);  
 }  
 **else**{  
 suffixMap\_[suffix\_] = unordered\_set<string>();  
 }  
 }  
 **for**(**auto** word\_ = lg\_it->second.begin(); word\_ != lg\_it->second.end(); word\_++){  
 set<string> mw\_;  
 string ownPrefix\_ = \_getPrefix(\*word\_);  
 string ownSuffix\_ = \_getSuffix(\*word\_);  
 unordered\_set<string> sufs\_ = prefixMap\_[ownPrefix\_];  
 unordered\_set<string> prefs\_ = suffixMap\_[ownSuffix\_];  
 **for**(**auto** suf\_ = sufs\_.begin(); suf\_ != sufs\_.end(); suf\_++){  
 **if**(\_areDifferentByOneSym(ownSuffix\_, \*suf\_))  
 mw\_.insert(ownPrefix\_ + (\*suf\_));  
 }  
 **for**(**auto** pref\_ = prefs\_.begin(); pref\_ != prefs\_.end(); pref\_++){  
 **if**(\_areDifferentByOneSym(ownPrefix\_, \*pref\_))  
 mw\_.insert((\*pref\_) + ownSuffix\_);  
 }  
 fsmw\_.insert(pair<string, set<string>>(\*word\_, mw\_));  
 }  
 }  
 **return** fsmw\_;  
}

## 2.4 Дополнительная функциональность

Дополнительная функциональность реализации отсутствует.

## 2.5 Заключение

Во второй части нами было иследовано влияния комбинации различных контейнеров для решения поставленной задачи на эффективность поиска. Первый метод оказался наиболее неэффективным в плане скорости работы, так как время работы является квадратичной, поэтому его целесообразно использовать на словарях меньшего размера. Другой метод является логическим следствием первого, но с некоторыми улучшениями, однако данные улучшения не меняют асимптотическую сложность метода. Третий подход оказался самым эффективным и его асимптотическая сложность стремится к , что позволяет использовать данный метод на практике. За ускорение по времени необходимо платить дополнительной памятью.

# 3. ЧАСТЬ №3

## 3.1 Постановка задачи

Цель части №3 разработать методы для поиска ошибочных слов и вариантов их замены путем удаления одного символа, вставки одного символа и транспозиции букв в строке, разработать программу для реализации методов и оценить их эффективность.

## 3.2 Краткий анализ задачи

Поиск слов, написанных с ошибкой, и вариантов их исправления является неотъемлемой частью системы проверки орфография современных программ для редактирования документов. В соответствии с предложенной проблемой были поставлены следующие задачи:

1. разработать методы для поиска ошибочных слов в тексте и вариантов их исправления путем преобразований через вставку или удаления одного символа, либо транспозицию букв внутри строки;
2. написать программу, реализующие данные методы;
3. оценить эффективность использования данных методов;
4. сделать выводы по поводу использования данных методов на практике.

Перед тем, как находить ошибки в тексте, необходимо составить словарь, включающие эталонные слова для проверки орфографии. Словарь целесообразно хранить в тех структурах данных, которые позволяют сократить время поиска до максимума. В нашей реализации было решено использовать хэш-таблицы, так как они обеспечивают в большинстве случаев линейную сложность поиска. В редких случаях возможно возникновение коллизий, которые приводят к тому, что время поиска уже не линейно, а квадратично. Но в среднем время поиска в хэш-таблице лучшем, чем в красно-черных деревьях, чья сложность поиска составляет .

При сканировании текста, программа ищет слово в словаре и если его не находит, что фиксирует это слово как ошибочное, сохраняет его номер, а затем производит поиск вариантов замены. Замена может быть произведена либо через удаление одного символа из исходного слова или вставки в него символа, либо через транспозицию букв.

Первый метод производит поиск все тех слов в словаре, которые могут быть получены путем перестановки (транспозиции) букв в исходном слове. Наивный поиск даст нам сложность , что не приемлемо для работы с большим текстом и словарем, поэтому необходимо снизить сложность до приемлемого уровня. Снизить сложность можно путем предварительной группировки слов по длине и общему набору символов, входящих в эти слова. Транспозиция обладает таким свойством, что слова одинаковой длины и отличные лишь перестановкой символов имеют одинаковый набор символов, входящих в них. Распишем последовательность шагов выбранного метода:

1. загружаем словарь из исходного файла;
2. группируем слова по длине строки;
3. в каждой группе разбиваем слова на новые группы (ассоциативными массивами), ключами которых являются отсортированная последовательность символов исходных строк, а значения – сами строки;
4. для проверяемого слова вычисляем длину строки;
5. длина строки позволит нам найти нужную группу слов, в которых мы и будем выбирать варианты замены;
6. в найденной группе находим слова, которые имеют тот же набор символов, что и проверяемое слово;
7. эти слова и будут вариантами замены.

Оценим сложность данного метода. Группировка слов по длине занимает . Группировка по набору символов занимает также . Поиск вариантов замены имеет сложность в лучшем и среднем случае , если используются хэш-таблицы, в худшем случае – . Если же используются красно-черные деревья [6], то сложность составляет в лучшем, среднем и худшем случае .

## 3.3 Детализации реализации

Для реализации разработанных методов был написан класс SpellSearcher, который отвечает за проверку слова на предмет орфографической ошибки и вариантов замены. Опишем структуру класса:

|  |  |
| --- | --- |
| SpellSearcher | |
| Поля | set<string> \*dict\_ – исходный словарь |
| unordered\_map<unsigned long, unordered\_map<string, unordered\_ser<string>>> ltg\_ – группа слов с одинаковой длиной и набором букв |
| Методы | SpellChecker() – конструктор |
| ~SpellChecker() – деструктор |
| void loadDictionary(istream&) – загрузка словаря из входного потока |
| set<string> options\_T(string) – поиск вариантов замены через транспозицию |
| set<string> options\_D(string) – поиск вариантов замены через удаление символа |
| set<string> options\_I(string) – поиск замены через вставку символа |
| bool spellCheck(string) – проверка на предмет наличия орфографической ошибки |

Приведем исходный код класса SpellChecker:

#pragma once  
  
#ifndef **CHW\_2\_SPELLCHECKER\_H**#define **CHW\_2\_SPELLCHECKER\_H**#include **<string>**#include **<set>**#include **<unordered\_map>**#include **<unordered\_set>  
  
using namespace** std;  
  
**typedef** unordered\_map<string, unordered\_set<string>> PREFIX\_MAP;  
**typedef** unordered\_map<string, unordered\_set<string>> SUFFIX\_MAP;  
**typedef unsigned long** ULONG;  
**typedef** unordered\_set<string> STRING\_SET;  
**typedef** unordered\_set<string>::iterator SSET\_ITERATOR;  
  
*/\*\*  
 \* Implementation of spell checking  
 \* with search options for replace wrong words by right ones  
 \*/***class** SpellChecker {  
**public**:  
 SpellChecker();  
 **virtual** ~SpellChecker();  
 **void** loadDictionary(istream& input);  
 set<string> options\_T(string s);  
 set<string> options\_D(string s);  
 set<string> options\_I(string s);  
 **bool** spellCheck(string s);  
**private**:  
 set<string> \*dict\_;  
 unordered\_map<ULONG, unordered\_map<string, unordered\_set<string>>> ltg\_;  
 unordered\_map<ULONG, pair<PREFIX\_MAP, SUFFIX\_MAP>> lpg\_;  
 **void** \_initLTG();  
 **void** \_initLPSG();  
 string \_getPrefix(string s);  
 string \_getSuffix(string s);  
};  
  
*/\*  
 \* Construct an instance of dictionary  
 \*/*SpellChecker::SpellChecker(){  
 dict\_ = **new** set<string>();  
}  
  
*/\*  
 \* Delete instance  
 \*/*SpellChecker::~SpellChecker() {  
 **delete** dict\_;  
}  
  
*/\*  
 \* Load dictionary from input stream (file)  
 \*/***void** SpellChecker::loadDictionary(istream &input) {  
 string temp\_;  
 **while**(getline(input, temp\_)){  
 dict\_->insert(temp\_);  
 }  
 \_initLTG();  
 \_initLPSG();  
}  
  
*/\*  
 \* Check if word is wrong  
 \*/***bool** SpellChecker::spellCheck(string s) {  
 **return** dict\_->find(s) != dict\_->end();  
}  
  
*/\*  
 \* Find all options for replace wrong word by right ones  
 \* which can be transformed into verifiable word through transposition  
 \* neighborhood in source word  
 \*/*set<string> SpellChecker::options\_T(string s) {  
 sort(s.begin(), s.end());  
 **return** set<string>(ltg\_[s.length()][s].begin(), ltg\_[s.length()][s].end());  
}  
  
*/\*  
 \* Find all options for replace wrong word by right ones  
 \* which can be transformed into verifiable word through deletion one symbol  
 \* from source word  
 \*/*set<string> SpellChecker::options\_D(string s) {  
 set<string> cw\_;  
 string prefix\_ = \_getPrefix(s);  
 string suffix\_ = \_getSuffix(s);  
 PREFIX\_MAP prefix\_map = lpg\_[s.length() + 1].first;  
 SUFFIX\_MAP suffix\_map = lpg\_[s.length() + 1].second;  
 STRING\_SET suffixes\_ = prefix\_map[prefix\_];  
 STRING\_SET prefixes\_ = suffix\_map[suffix\_];  
 **for**(**auto** s\_ = suffixes\_.begin(); s\_ != suffixes\_.end(); s\_++){  
 cw\_.insert(prefix\_ + \*s\_);  
 }  
 **for**(**auto** p\_ = prefixes\_.begin(); p\_ != prefixes\_.end(); p\_++){  
 cw\_.insert(\*p\_ + suffix\_);  
 }  
 **return** cw\_;  
}  
  
*/\*  
 \* Find all options for replace wrong word by right ones  
 \* which can be transformed into verifiable word through insertion one symbol  
 \* into source word  
 \*/*set<string> SpellChecker::options\_I(string s) {  
 set<string> cw\_;  
 string prefix\_ = \_getPrefix(s);  
 string suffix\_ = \_getSuffix(s);  
 PREFIX\_MAP prefix\_map = lpg\_[s.length() - 1].first;  
 SUFFIX\_MAP suffix\_map = lpg\_[s.length() - 1].second;  
 STRING\_SET suffixes\_ = prefix\_map[prefix\_];  
 STRING\_SET prefixes\_ = suffix\_map[suffix\_];  
 **for**(**auto** s\_ = suffixes\_.begin(); s\_ != suffixes\_.end(); s\_++){  
 cw\_.insert(prefix\_ + \*s\_);  
 }  
 **for**(**auto** p\_ = prefixes\_.begin(); p\_ != prefixes\_.end(); p\_++){  
 cw\_.insert(\*p\_ + suffix\_);  
 }  
 **return** cw\_;  
}  
  
  
**void** SpellChecker::\_initLTG(){  
 **for**(**auto** w\_ = dict\_->begin(); w\_ != dict\_->end(); w\_++){  
 **if**(ltg\_.find(w\_->length()) == ltg\_.end()){  
 ltg\_[w\_->length()] = unordered\_map<string, unordered\_set<string>>();  
 }  
 string temp\_(\*w\_);  
 sort(temp\_.begin(), temp\_.end());  
 **if**(ltg\_[w\_->length()].find(temp\_) == ltg\_[w\_->length()].end()){  
 ltg\_[w\_->length()][temp\_] = unordered\_set<string>();  
 }  
 ltg\_[w\_->length()][temp\_].insert(\*w\_);  
 }  
}  
  
**void** SpellChecker::\_initLPSG() {  
 **for**(**auto** w\_ = dict\_->begin(); w\_ != dict\_->end(); w\_++){  
 **if**(lpg\_.find(w\_->length()) == lpg\_.end()){  
 lpg\_[w\_->length()] = pair<PREFIX\_MAP, SUFFIX\_MAP>();  
 }  
 string prefix\_ = \_getPrefix(\*w\_);  
 string suffix\_ = \_getSuffix(\*w\_);  
 **if**(w\_->length() % 2){  
 **if**(lpg\_[w\_->length()].first.find(prefix\_) == lpg\_[w\_->length()].first.end())  
 lpg\_[w\_->length()].first[prefix\_] = unordered\_set<string>();  
 **if**(lpg\_[w\_->length()].second.find(suffix\_) == lpg\_[w\_->length()].second.end())  
 lpg\_[w\_->length()].second[suffix\_] = unordered\_set<string>();  
 lpg\_[w\_->length()].first[prefix\_].insert(suffix\_);  
 lpg\_[w\_->length()].second[suffix\_].insert(prefix\_);  
 }  
 **else**{  
 ULONG plen\_ = (ULONG)floor(w\_->length() / 2);  
 string pkey\_ = prefix\_;  
 **if**(w\_->length() != 1)  
 prefix\_ += w\_->**operator**[](plen\_ + 1);  
 **if**(lpg\_[w\_->length()].first.find(pkey\_) == lpg\_[w\_->length()].first.end())  
 lpg\_[w\_->length()].first[pkey\_] = unordered\_set<string>();  
 **if**(lpg\_[w\_->length()].second.find(suffix\_) == lpg\_[w\_->length()].second.end())  
 lpg\_[w\_->length()].second[suffix\_] = unordered\_set<string>();  
 lpg\_[w\_->length()].first[pkey\_].insert(suffix\_);  
 lpg\_[w\_->length()].second[suffix\_].insert(prefix\_);  
 }  
 }  
}  
  
string SpellChecker::\_getPrefix(string s) {  
 **if**(s.length() < 2)  
 **return** s;  
 ULONG plen\_ = (ULONG)floor(s.length() / 2);  
 **return** s.substr(0, plen\_);  
}  
  
string SpellChecker::\_getSuffix(string s) {  
 **if**(s.length() < 2)  
 **return** s;  
 ULONG plen\_ = (ULONG)floor(s.length() / 2);  
 **if**(s.length() % 2)  
 **return** s.substr(plen\_, s.length() - plen\_);  
 **return** s.substr(plen\_ + 1, s.length() - plen\_ - 1);  
}  
  
#endif *//CHW\_2\_SPELLCHECKER\_H*

## 3.4 Дополнительная функциональность

Дополнительная функциональность реализации отсутствует.

## 3.5 Заключение

В третьей части нами было исследовано влияние эффективности использования разных видов контейнеров для поиска замены ошибочных слов и написана программа, решающая данную задачу. Было выяснено, что красно-черные деревья, реализованные внутри set, имеют одну и ту же сложность по времени в лучшем, худшем и среднем случае. Хэш-таблицы, реализованные в unordered\_set, имеют сложность по времени лучшем, чем красно-черные деревья, при условии отсутствия возникших коллизий. Но как было уже выявлено в заключении к первой части, вероятность коллизий достаточно мала и стремится к нулю.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Wikipedia. Hash table [Электронный ресурс] //URL: https://en.wikipedia.org/wiki/Hash\_table (Дата обращения: 11.05.2016, режим доступа: свободный).
2. Habrahabr. Trie, или нагруженное дерево [Электронный ресурс] //URL: https://habrahabr.ru/post/111874/ (Дата обращения: 11.05.2016, режим доступа: свободный).
3. C++ Reference. Set [Электронный ресурс] //URL: http://www.cplusplus.com/reference/set/set/ (Дата обращения: 11.05.2016, режим доступа: свободный).
4. C++ Reference. Unordered set [Электронный ресурс] //URL: http://www.cplusplus.com/reference/unordered\_set/unordered\_set/ (Дата обращения: 11.05.2016, режим доступа: свободный).
5. C++ Reference. Unordered map [Электронный ресурс] //URL: http://www.cplusplus.com/reference/unordered\_map/unordered\_map/ (Дата обращения: 11.05.2016, режим доступа: свободный).
6. Wikipedia. Standatd Template Library [Электронный ресурс] //URL: https://en.wikipedia.org/wiki/Standard\_Template\_Library (Дата обращения: 11.05.2016, режим доступа: свободный).
7. Wikipedia. Red-black tree [Электронный ресурс] //URL: https://en.wikipedia.org/wiki/Red–black\_tree (Дата обращения: 11.05.2016, режим доступа: свободный).
8. Wikipedia. PATRICIA tree [Электронный ресурс] //URL: https://en.wikipedia.org/wiki/Radix\_tree (Дата обращения: 11.05.2016, режим доступа: свободный).

Я, Ярных Роман Вячеславович, подтверждаю, что предоставляемое на проверку Домашнее задание по дисциплине «Алгоритмы и структуры данных» (модуль 4, 2015-2016 учебного года) выполнено мною самостоятельно. Данный отчет не содержит объяснений (пояснительного текста), фрагментов псевдокода и/или кода программ, взятых из других работ (электронных или печатных), без указания в тексте явных ссылок на эти источники.

Я проинформирован и полностью осознаю, что при обнаружении в отчёте факта списывания, подлога, плагиата и пр., моей оценкой за домашнее задание будет 0 (нуль) и ко мне будут применены меры, изложенные в документе *«Порядок применения дисциплинарных взысканий при нарушениях академических норм в написании письменных учебных работ в НИУ-ВШЭ»* (<http://www.hse.ru/studyspravka/loc>).