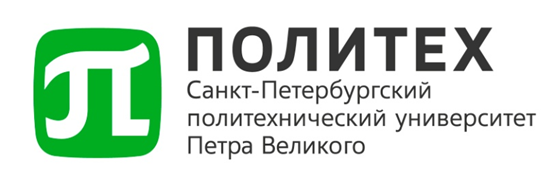
**Санкт-Петербургский политехнический университет Петра Великого**

**Институт компьютерных наук и технологий**

**Высшая школа программной инженерии**

****

**КУРСОВАЯ РАБОТА**

**Частотный словарь. 2-3 дерево.**

по дисциплине «**Практикум по программированию**»

Выполнил

студент гр.

3530904/90006 *<подпись>* Ковалева А.П

Руководитель *<подпись>* Толстолес А.А

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2020

Санкт-Петербург

2020 г

Содержание

[Введение. Общая постановка задачи 3](#_Toc42162842)

[Основная часть работы 4](#_Toc42162843)

[Описание алгоритма решения и используемых структур данных 4](#_Toc42162844)

[Анализ алгоритма 8](#_Toc42162845)

[Описание спецификации программы (детальные требования) 9](#_Toc42162846)

[Описание программы 10](#_Toc42162847)

[Заключение 11](#_Toc42162848)

[Список литературы 12](#_Toc42162849)

[Приложение 1 13](#_Toc42162850)

[text.txt 13](#_Toc42162851)

[main.cpp 13](#_Toc42162852)

[FrequencyDictionary.h 14](#_Toc42162853)

[FrequencyDictionary.cpp 15](#_Toc42162854)

[Приложение 2 39](#_Toc42162855)

# 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 1.4.2.**

**Частотный словарь. 2-3 дерево***.*

Разработать и реализовать алгоритм формирования частотного словаря:

* определить понятие слово,
* прочитать текст и сформировать набор слов вместе с информацией о частоте их встречаемости
* определить три чаще всего встречающихся слова

Для реализации задания использовать **2-3 дерево,**

Внутренний узел 2-3 дерева должен содержать только ключи, листья дерева - ключ и информационную часть:

* Ключ – слово
* Информационная часть – количество слов

Лабораторная работа предназначена для закрепления теоретических знаний и практических навыков. Данная лабораторная работа предполагает изучение 2-3 деревьев и частотных словарей и реализацию частотного словарь на основе 2-3 дерева.

Частотный словарь – это набор слов данного языка вместе с информацией о частоте их встречаемости. Он может быть отсортирован как по частоте встречаемости, так и по алфавиту. В моей реализации частотный словарь отсортирован по частоте встречаемости.

# Основная часть работы

## Описание алгоритма решения и используемых структур данных

Определение понятия «слово»:

Слово – набор английских букв любого размера. Слово может быть составным, т.е. состоять из нескольких слов разделенных 1 дефисом («-»). Перед и после дефиса («-») должна находиться хотя бы 1 буква. Также слово может содержать апостроф («’»), он должен находиться на предпоследнем месте, до апострофа («’») и после него («-») должна находиться хотя бы 1 буква.

Примеры:

“I”, “someone’s”, “well-trained”, “bla-bla-bla”, “ffgge” – слова.

“2”, “d>f”, “ff’ff”, “fs’gg’s”, “ds--ff”, “da2da” – не слова.

Описание алгоритма:

Из файла по одному считываются входные наборы символов. Если в начале набора символов стоит скобка или/и кавычка, она/они отбрасываются. Если набор символов оканчивается скобкой, или/и кавычкой, или/и знаком препинания, этот/эти символы отбрасываются. Если в слове есть большие буквы, то они заменяются на маленькие. Дальше проверяется является ли этот набор символов словом. При положительном результате вносим слово в 2-3 дерево, при отрицательном – сообщаем что такой набор символов содержит недопустимые символы и считываем следующий набор символов. Если данное слово уже есть в 2-3 дереве, то счетчик частоты встречаемости этого слова увеличится на 1. Если этого слова еще нет в 2-3 дереве, то оно добавляется в нужное место. При добавлении в дерево, слово сравнивается с тремя самыми часто встречающимися словами в словаре га данный момент и если частота встречаемости нового слова больше, то оно встает в топ 3.

В конце в консоль будут выведены 3 самых часто встречающихся слова.

Пример входных данных:

One night's - it was on the somethin'g's bla-bla-bla hi--hi twentieth "of" ("as"), March, 1888 - I was retUrning from a journey to a patient (for I had now returned to civil practice), when my way led me through Baker Street. As I passed well-remembered door, which must always: be; associated in my mind with my wooing, with the dark incidents of the Study in Scarlet, I was seized with a keen desire to see Holmes again, and to know how he was employing his extraordinary powers. His rooms were brilliantly lit, and, even as I looked up, I saw his tall, spare his figure pass twice in a dark silhouette against the blind. He was pacing the room swiftly, eagerly, with his head sunk upon his chest and his hands clasped behind him. To me, who knew his every mood and habit, his attitude and manner told their own story. He was at work again. He had risen out of his drug-created dreams and was hot upon the scent of some new problem. I rang the bell and was shown up to the chamber which had was formerly been in part my own.

Данный пример содержит как допустимые значение, так и недопустимые.

Вывод:

3 most frequent words:

his : 10

was : 9

the : 8

Используемые структуры данных:

**2-3 дерево**

2-3 дерево – это структура данных, представляющая собой сбалансированное дерево поиска, такое что из глубина всех листьев одинакова. Каждая вершина 2-3 дерева может иметь 2 ключа и 3 сына или 1 ключ и 2 сына.

Свойства:

* Все нелистовые вершины содержат 1 ключ и 2 сына или 2 ключа и 3 сына.
* Все листовые вершины находятся на одном уровне (на нижнем уровне) и содержат 1 или 2 поля.
* Нелистовые вершины содержат один или два ключа, указывающих на диапазон значений в их поддеревьях. Значение первого ключа строго больше наибольшего значения в левом поддереве и меньше или равно наименьшему значению в правом поддереве (или в центральном поддереве, если это 3-вершина); аналогично, значение второго поля (если оно есть) строго больше наибольшего значения в центральном поддереве и меньше или равно, чем наименьшее значение в правом поддереве.

Тип данных 2-3 дерева – Data.

В структуре данных Data хранятся слова и количество их повторений

Типы полей:

struct Node

{

public:

int size; //размер вершины, т.е сколько ключей в нем находится

Data key[3]; //ключи

Node \*first; // указатели на потомков

Node \*second;

Node \*third;

Node \*fourth;

Node \*parent; // указатель на родителя

Node(string k);

Node();

void sort(); // сортирует вершины по возрастанию

void insertToNode(string k); //добавление в вершину слова

void insertToNode(string k, int timesAdded); //добавление в вершину слова с

заданной частотой встречаемости

void insertToNode(Data k); //добавление в вершину словаа

void insertToNode(Data k, int timesAdded); //добавление в вершину слова с заданной

частотой встречаемости

bool isLeaf(); // возвращает true если вершина является листом, т.е не имеет

потомков

bool find(Data k); // возвращает true если в вершине имеется ключ к

void removeFromNode(Data k); // удаляет ключ из узла

void removeFromNode(Data k, int timesAdded); // удаляет ключ из узла заданное

количество раз

};

struct Data

{

string word; // слово

int timesAdded = 0; // сколько раз это слово встретилось

Data(string k);

Data();

int compare(string word1, string word2); // сравнивает 2 слова

bool operator < (Data &other);

friend bool operator< (string &str, Data &data);

bool operator > (Data &other);

friend bool operator > (string &str, Data &data);

bool operator <= (string other);

bool operator >= (string other);

bool operator == (string other);

friend bool operator == (Data &dataL, Data &dataR);

friend bool operator == (string &other, Data& data);

friend std::ostream& operator<<(std::ostream & out, Data &data);

};

Прототипы методов:

public:

FrequencyDictionary();

void insert(string k); //добавление слова

void INSEART(string k, int timesAdded); //добавление слова

void split(Node \*node); //разделение вершины

void printTree(Node \*node, int space); //выводит словарь в форме 2-3 дерева

void printTree();

void printDict(Node \*node); //выводит словарь в виде списка

void printDict();

void printThreeMostFrequentWords(); //выводит 3 самых часто встречающихся слова

Node \*SEARCH(string key); //поиск слова в словаре

void DELETE(string key); // удаление заданного слова

Node \*findMin(Node \*node);

Node \*fix(Node \*node); // исправление дерева после удаления

Node \*redestribute(Node \*node); // перераспределение

Node \*merge(Node \*node); //склеивание

void deleteTree(Node \*node); // удаление дерева

void deleteTree();

bool isWord(string &k); // проверка является ли k словом

void addToDicrionary(Node \*node, string key); //добавление в словарь

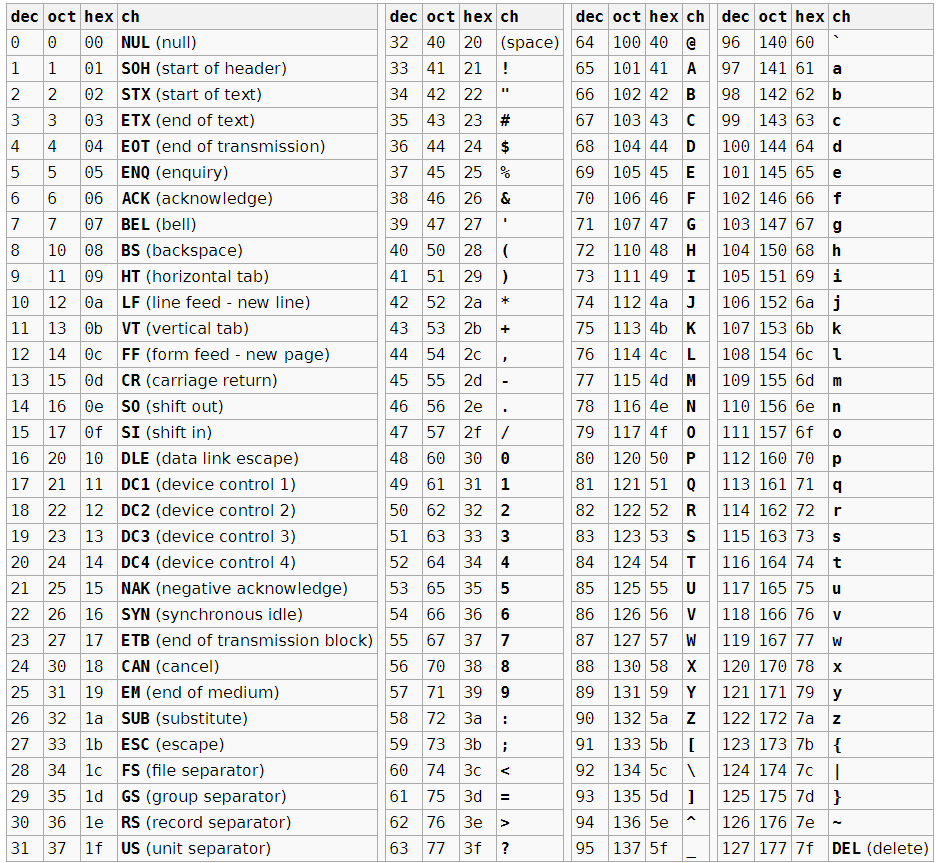
void removeFromDictionary(Node \*node, string key);// удаление из словаря

~FrequencyDictionary();

Операции сравнения ключей в дереве основаны на посимвольном сравнении слов основываясь на коде символов из ASCII таблицы.

Например слово «on» больше слова «it», т.к первые символы отличаются друг от друга и «o» > «i» (код символа «o» = 111, а код символа «i» = 105, соответственно 111 > 105). Или слово «twenty» > «the», т.к первые отличающиеся символы – «w» и «h». Код символа «w» = 119, код символа «h» = 104, 119 > 104 и значит «w» > «h», отсюда следует, что «twenty» > «the».

ASCII таблица:



## Анализ алгоритма

Удаление и добавление элемента в 2-3 дерево как в лучшем, так и в худшем и среднем случае займет время = О(log h), где h – высота дерева. Поиск элемента также занимает О(log h).

## Описание спецификации программы (детальные требования)

Узел 2-3 дерева состоит из двух ключей типа Data. Ключи хранят в себе слово и информацию о частоте его встречаемости.

Слово – набор английских букв любого размера. Слово может быть составным, т.е. состоять из нескольких слов разделенных 1 дефисом («-»). Перед и после дефиса («-») должна находиться хотя бы 1 буква. Также слово может содержать апостроф («’»), он должен находиться на предпоследнем месте, до апострофа («’») и после него («-») должна находиться хотя бы 1 буква.

Примеры:

“I”, “someone’s”, “well-trained”, “bla-bla-bla”, “ffgge” – слова.

“2”, “d>f”, “ff’ff”, “fs’gg’s”, “ds--ff”, “da2da” – не слова.

Если перед словом и после него стоят скобки или кавычки – они будут отброшены. Знаки пунктуации в конце слова также будут отброшены.

## Описание программы

Программа состоит из 3 файлов:

* FrequensyDictionary.h и FrequensyDictionary.cpp – реализация частотного словаря на основе 2-3 дерева
* Main.cpp – тестовая main функция

Входные данные – файл “text.txt” с текстом, из которого будут считываться слова и заноситься в частотный словарь.

Выходные данные – 3 самых часто встречающихся слова и весь частотный словарь.

# Заключение

Цель лабораторной работы, поставленная во введении, выполнена. При написании лабораторной работы я познакомилась с 2-3 деревьями и реализовала частотный словарь на основе 2-3 дерева.

При написании лабораторной работы мною были изучены статьи, посвященные данному вопросу, описаны алгоритмы реализации и теоретические аспекты.

# Список литературы

* 2-3 дерево <https://neerc.ifmo.ru/wiki/index.php?title=2-3_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE>
* Частотный словарь <https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%81%D1%82%D0%BE%D1%82%D0%BD%D1%8B%D0%B9_%D1%81%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C>
* 2-3 дерево <https://habr.com/ru/post/303374/>
* 2-3 дерево <https://www.geeksforgeeks.org/2-3-trees-search-and-insert/?ref=lbp>

# Приложение 1

## text.txt

One night's - it was on the somethin'g's bla-bla-bla hi--hi twentieth "of" ("as"), March, 1888 - I was retUrning from a journey to a patient (for I had now returned to civil practice),

when my way led me through Baker Street. As I passed well-remembered door, which must always: be; associated in my mind with my wooing, with the dark incidents of the Study in Scarlet,

I was seized with a keen desire to see Holmes again, and to know how he was employing his extraordinary powers. His rooms were brilliantly lit, and, even as I looked up, I saw his tall,

spare his figure pass twice in a dark silhouette against the blind. He was pacing the room swiftly, eagerly, with his head sunk upon his chest and his hands clasped behind him. To me, who knew his

every mood and habit, his attitude and manner told their own story. He was at work again. He had risen out of his drug-created dreams and was hot upon the scent of some new problem. I rang the bell

and was shown up to the chamber which had was formerly been in part my own.

## main.cpp

#include <iostream>

#include <fstream>

#include <algorithm>

#include "FrequencyDictionary.h"

using namespace std;

int main()

{

FrequencyDictionary dict;

ifstream file;

file.open("text.txt");

string word;

while (file >> word)

{

dict.insert(word); //чтение из файла всех слов и занесение их в словарь

}

file.close();

cout << "\n";

dict.printThreeMostFrequentWords(); // вывод 3x самых часто встречающихся слов

cout << "\nWhole dictionary: \n";

dict.printDict(); // вывод всего частотного словаря отсортированного по частоте встречаемости слов

cout << "\nInserting words with set frequency\n";

dict.INSEART("Wa4gg", 20); // добавляем неправильное слово

cout << "Inserting Watson\n";

dict.INSEART("Watson", 20); // длбавляем слово с частотой встречаемости

dict.printThreeMostFrequentWords(); // вывозим 3 самых часто встречающихся слова

cout << "\ndeleteng xoxo\n";

dict.DELETE("xoxo"); // слово не удалится так как такого слова в словаре нет

dict.printDict();

cout << "\ndeleteng was\n";

dict.DELETE("was");

dict.printThreeMostFrequentWords();

cout << "whole dictionary after deleting was\n";

dict.printDict();

return 0;

}

## FrequencyDictionary.h

#pragma once

#include <iostream>

#include <string>

#include <vector>

using std::string;

using std::vector;

struct Data

{

string word; // слово

int timesAdded = 0; // сколько раз это слово встретилось

Data(string k);

Data();

int compare(string word1, string word2); // сравнивает 2 слова

bool operator < (Data &other);

friend bool operator< (string &str, Data &data);

bool operator > (Data &other);

friend bool operator > (string &str, Data &data);

bool operator <= (string other);

bool operator >= (string other);

bool operator == (string other);

friend bool operator == (Data &dataL, Data &dataR);

friend bool operator == (string &other, Data& data);

friend std::ostream& operator<<(std::ostream & out, Data &data);

};

struct Node

{

public:

int size; //размер вершины, т.е сколько ключей в нем находится

Data key[3]; //ключи

Node \*first; // указатели на потомков

Node \*second;

Node \*third;

Node \*fourth;

Node \*parent; // указатель на родителя

Node(string k);

Node();

void sort(); // сортирует вершины по возрастанию

void insertToNode(string k); //добавление в вершину слова

void insertToNode(string k, int timesAdded); //добавление в вершину слова с заданной частотой встречаемости

void insertToNode(Data k); //добавление в вершину словаа

void insertToNode(Data k, int timesAdded); //добавление в вершину слова с заданной частотой встречаемости

bool isLeaf(); // возвращает true если вершина является листом, т.е не имеет потомков

bool find(Data k); // возвращает true если в вершине имеется ключ к

void removeFromNode(Data k); // удаляет ключ из узла

void removeFromNode(Data k, int timesAdded); // удаляет ключ из узла заданное количество раз

};

class FrequencyDictionary

{

public:

FrequencyDictionary();

void insert(string k); //добавление слова

void INSEART(string k, int timesAdded); //добавление слова

void split(Node \*node); //разделение вершины

void printTree(Node \*node, int space); //выводит словарь в форме 2-3 дерева

void printTree();

void printDict(Node \*node); //выводит словарь в виде списка

void printDict();

void printThreeMostFrequentWords(); //выводит 3 самых часто встречающихся слова

Node \*SEARCH(string key); //поиск слова в словаре

void DELETE(string key); // удаление заданного слова

Node \*findMin(Node \*node);

Node \*fix(Node \*node); // исправление дерева после удаления

Node \*redestribute(Node \*node); // перераспределение

Node \*merge(Node \*node); //склеивание

void deleteTree(Node \*node); // удаление дерева

void deleteTree();

bool isWord(string &k); // проверка является ли k словом

void addToDicrionary(Node \*node, string key); //добавление в словарь

void removeFromDictionary(Node \*node, string key);// удаление из словаря

~FrequencyDictionary();

private:

Node \*root\_; //корень дерева

vector<Data> frequensyDict\_; //частотный словарь

};

## FrequencyDictionary.cpp

#pragma once

#include "FrequencyDictionary.h"

#include <iostream>

#include <string>

#include <algorithm>

using std::string;

using std::cout;

using std::vector;

FrequencyDictionary::FrequencyDictionary() :

root\_(nullptr),

frequensyDict\_()

{}

void FrequencyDictionary::insert(string k)

{

if (!isWord(k)) // проверяем является ли словом

{

cout << "Cant add '" << k << "' becuse it contains illegal characters\n";

return;

}

if (root\_ == nullptr) // случай когда дерево пустое

{

root\_ = new Node(k);

addToDicrionary(root\_, k);

}

else

{

Node \*check = SEARCH(k);

if (check != nullptr)

{

check->insertToNode(k); // случай когда в дереве уже есть ключ k

addToDicrionary(check, k);

}

else

{

Node \*current = root\_; // случай когда k - новое слово. Ищем подходящее место для вставки

if (current->isLeaf()) // корень - лист

{

current->insertToNode(k);

split(current);

addToDicrionary(SEARCH(k), k);

}

else

{

while (!current->isLeaf())

{

if (current->key[0] >= k)

{

current = current->first;

}

else

{

if ((current->size == 1) || ((current->size == 2) && (current->key[1]) >= k))

{

current = current->second;

}

else

{

current = current->third;

}

}

}

current->insertToNode(k);

split(current);

addToDicrionary(SEARCH(k), k);

}

}

}

}

void FrequencyDictionary::INSEART(string k, int timesAdded)

{

if (!isWord(k))

{

cout << "Cant add '" << k << "' becuse it contains illegal characters\n";

return;

}

if (root\_ == nullptr)

{

root\_ = new Node();

root\_->insertToNode(k, timesAdded);

addToDicrionary(root\_, k);

}

else

{

Node \*check = SEARCH(k);

if (check != nullptr)

{

check->insertToNode(k, timesAdded);

addToDicrionary(check, k);

}

else

{

Node \*current = root\_;

if (current->isLeaf())

{

current->insertToNode(k, timesAdded);

split(current);

addToDicrionary(SEARCH(k), k);

}

else

{

while (!current->isLeaf())

{

if (current->key[0] >= k)

{

current = current->first;

}

else

{

if ((current->size == 1) || ((current->size == 2) && (current->key[1]) >= k))

{

current = current->second;

}

else

{

current = current->third;

}

}

}

current->insertToNode(k, timesAdded);

split(current);

addToDicrionary(SEARCH(k), k);

}

}

}

}

void FrequencyDictionary::split(Node \* node) // пункт1

{

if (node->size >= 3) // пункт1

{

Node \*x = new Node; // пункт2

x->insertToNode(node->key[0]);

x->first = node->first;

x->second = node->second;

x->parent = node->parent;

if (x->first != nullptr)

{

x->first->parent = x;

}

if (x->second != nullptr)

{

x->second->parent = x;

}

Node \*y = new Node;

y->insertToNode(node->key[2]);

y->first = node->third;

y->second = node->fourth;

y->parent = node->parent;

if (y->first != nullptr)

{

y->first->parent = y;

}

if (y->second != nullptr)

{

y->second->parent = y;

}

if (node->parent != nullptr) // пункт3

{

node->parent->insertToNode(node->key[1]);

if (node->parent->first == node)

{

node->parent->first = nullptr;

}

else

{

if (node->parent->second == node)

{

node->parent->second = nullptr;

}

else

{

if (node->parent->third == node)

{

node->parent->third = nullptr;

}

}

}

if (node->parent->first == nullptr) // пункт 3.1

{

node->parent->fourth = node->parent->third;

node->parent->third = node->parent->second;

node->parent->second = y;

node->parent->first = x;

}

else

{

if (node->parent->second == nullptr) // пункт 3.2

{

node->parent->fourth = node->parent->third;

node->parent->third = y;

node->parent->second = x;

}

else

{

node->parent->fourth = y; // пункт 3.3

node->parent->third = x;

}

}

if (node->parent->size == 3)

{

split(node->parent); // пункт 3.4

}

delete node;

}

else

{

Node \*temp = new Node; // пуикт 4

temp->insertToNode(node->key[1]);

if (node == root\_)

{

root\_ = temp;

}

else

{

temp->parent = node->parent;

}

delete node;

temp->first = x;

temp->second = y;

x->parent = temp;

y->parent = temp;

}

}

}

void FrequencyDictionary::printTree(Node \* node, int space)

{

if (node != nullptr)

{

space += 15;

printTree(node->first, space);

cout << std::endl;

for (int i = 15; i < space; i++)

{

cout << " ";

}

if (node == root\_)

{

cout << "r-> ";

}

if (node->size == 2)

{

cout << node->key[0] << "(" << node->key[0].timesAdded << ");\n";

for (int i = 15; i < space; i++)

{

cout << " ";

}

cout << node->key[1] << "(" << node->key[1].timesAdded << ")\n";

}

if (node->size == 1)

{

cout << node->key[0] << "(" << node->key[0].timesAdded << ")" << std::endl;

}

printTree(node->second, space);

printTree(node->third, space);

}

else

{

return;

}

}

void FrequencyDictionary::printTree()

{

if (root\_ == nullptr)

{

cout << "Tree is emty \n";

}

else

{

printTree(root\_, 0);

}

}

void FrequencyDictionary::printDict()

{

if (root\_ == nullptr)

{

cout << "Dictionary is emty \n";

}

else

{

vector<Data>::iterator it = frequensyDict\_.begin();

for (int i = 0; i < frequensyDict\_.size(); i++)

{

cout << frequensyDict\_.at(i).word << "(" << frequensyDict\_.at(i).timesAdded << ")" << std::endl;

}

}

}

void FrequencyDictionary::printThreeMostFrequentWords()

{

cout << "3 most frequent words:\n";

for (int i = 0; i < 3; i++)

{

cout << frequensyDict\_.at(i).word << " : " << frequensyDict\_.at(i).timesAdded << "\n";

}

}

Node \* FrequencyDictionary::SEARCH(string key)

{

if (root\_ == nullptr) // пункт 1

{

cout << "Tree is empty.\n";

return nullptr;

}

Node \*current = root\_;

while (current != nullptr)

{

if (current->find(key))// пункт 2

{

return current;

}

if (key < current->key[0])// пункт 3

{

current = current->first;

}

else

{

if (((current->size == 2) && (key < current->key[1])) || (current->size == 1))

{

current = current->second;

}

else

{

if (current->size == 2)

{

current = current->third;

}

}

}

}

return nullptr;

}

void FrequencyDictionary::DELETE(string key)

{

Node \*node = SEARCH(key);

if (node == nullptr) // пункт 1

{

cout << "No such node in tree\n";

}

else

{

removeFromDictionary(node, key);

Node \*min = nullptr;

if (node->key[0] == key) // пункт 2

{

min = findMin(node->second);

}

else

{

min = findMin(node->third);

}

if (min != nullptr) // пункт 3

{

if (node->key[0] == key)

{

node->key[0] = min->key[0];

}

else

{

node->key[1] = min->key[0];

}

min->removeFromNode(min->key[0], min->key[0].timesAdded);

fix(min);

}

else

{

if (node->key[0] == key) // пункт 4

{

node->removeFromNode(node->key[0], node->key[0].timesAdded);

}

else

{

node->removeFromNode(node->key[1], min->key[1].timesAdded);

}

fix(node);

}

}

}

Node \* FrequencyDictionary::findMin(Node \*node)

{

if (node == nullptr)

{

return node;

}

if (node->first == nullptr)

{

return node;

}

else

{

return findMin(node->first);

}

}

Node \* FrequencyDictionary::fix(Node \* node)

{

if (node->size == 0 && node->parent == nullptr)

{

delete node;

return nullptr;

}

if (node->size != 0)

{

if (node->parent != nullptr)

{

return fix(node->parent);

}

else

{

return node;

}

}

Node \*parent = node->parent;

if ((parent->first->size == 2) || (parent->second->size == 2) || (parent->size == 2))

{

node = redestribute(node);

}

else

{

if (parent->size == 2 && parent->third->size == 2)

{

node = redestribute(node);

}

else

{

node = merge(node);

}

}

return fix(node);

}

Node \* FrequencyDictionary::redestribute(Node\* node)

{

Node \*parent = node->parent;

Node \*first = parent->first;

Node \*second = parent->second;

Node \*third = parent->third;

if ((parent->size == 2) && (first->size < 2) && (second->size < 2) && (third->size < 2)) { // пункт 1

if (first == node) { // пункт 1.1

parent->first = parent->second;

parent->second = parent->third;

parent->third = nullptr;

parent->first->insertToNode(parent->key[0], parent->key[0].timesAdded);

parent->first->third = parent->first->second;

parent->first->second = parent->first->first;

if (node->first != nullptr) parent->first->first = node->first;

else if (node->second != nullptr) parent->first->first = node->second;

if (parent->first->first != nullptr) parent->first->first->parent = parent->first;

parent->removeFromNode(parent->key[0], parent->key[0].timesAdded);

delete first;

}

else if (second == node) { // пункт 1.2

first->insertToNode(parent->key[0], parent->key[0].timesAdded);

parent->removeFromNode(parent->key[0], parent->key[0].timesAdded);

if (node->first != nullptr) first->third = node->first;

else if (node->second != nullptr) first->third = node->second;

if (first->third != nullptr) first->third->parent = first;

parent->second = parent->third;

parent->third = nullptr;

delete second;

}

else if (third == node) {// пункт 1.3

second->insertToNode(parent->key[1], parent->key[1].timesAdded);

parent->third = nullptr;

parent->removeFromNode(parent->key[1], parent->key[1].timesAdded);

if (node->first != nullptr) second->third = node->first;

else if (node->second != nullptr) second->third = node->second;

if (second->third != nullptr) second->third->parent = second;

delete third;

}

}

else if ((parent->size == 2) && ((first->size == 2) || (second->size == 2) || (third->size == 2))) { // пункт 2

if (third == node) { // пункт 2.1

if (node->first != nullptr) {

node->second = node->first;

node->first = nullptr;

}

node->insertToNode(parent->key[1], parent->key[1].timesAdded);

if (second->size == 2) {

parent->key[1] = second->key[1];

second->removeFromNode(second->key[1], second->key[1].timesAdded);

node->first = second->third;

second->third = nullptr;

if (node->first != nullptr) node->first->parent = node;

}

else if (first->size == 2) {

parent->key[1] = second->key[0];

node->first = second->second;

second->second = second->first;

if (node->first != nullptr) node->first->parent = node;

second->key[0] = parent->key[0];

parent->key[0] = first->key[1];

first->removeFromNode(first->key[1], first->key[1].timesAdded);

second->first = first->third;

if (second->first != nullptr) second->first->parent = second;

first->third = nullptr;

}

}

else if (second == node) { // пункт 2.2

if (third->size == 2) {

if (node->first == nullptr) {

node->first = node->second;

node->second = nullptr;

}

second->insertToNode(parent->key[1], parent->key[1].timesAdded);

parent->key[1] = third->key[0];

third->removeFromNode(third->key[0], third->key[0].timesAdded);

second->second = third->first;

if (second->second != nullptr) second->second->parent = second;

third->first = third->second;

third->second = third->third;

third->third = nullptr;

}

else if (first->size == 2) {

if (node->second == nullptr) {

node->second = node->first;

node->first = nullptr;

}

second->insertToNode(parent->key[0], parent->key[0].timesAdded);

parent->key[0] = first->key[1];

first->removeFromNode(first->key[1], first->key[1].timesAdded);

second->first = first->third;

if (second->first != nullptr) second->first->parent = second;

first->third = nullptr;

}

}

else if (first == node) { // пункт 2.3

if (node->first == nullptr) {

node->first = node->second;

node->second = nullptr;

}

first->insertToNode(parent->key[0], parent->key[0].timesAdded);

if (second->size == 2) {

parent->key[0] = second->key[0];

second->removeFromNode(second->key[0], second->key[0].timesAdded);

first->second = second->first;

if (first->second != nullptr) first->second->parent = first;

second->first = second->second;

second->second = second->third;

second->third = nullptr;

}

else if (third->size == 2) {

parent->key[0] = second->key[0];

second->key[0] = parent->key[1];

parent->key[1] = third->key[0];

third->removeFromNode(third->key[0], third->key[0].timesAdded);

first->second = second->first;

if (first->second != nullptr) first->second->parent = first;

second->first = second->second;

second->second = third->first;

if (second->second != nullptr) second->second->parent = second;

third->first = third->second;

third->second = third->third;

third->third = nullptr;

}

}

}

else if (parent->size == 1) { // пункт 3

node->insertToNode(parent->key[0]);

if (first == node && second->size == 2) { // пункт 3.1

parent->key[0] = second->key[0];

second->removeFromNode(second->key[0], second->key[0].timesAdded);

if (node->first == nullptr) node->first = node->second;

node->second = second->first;

second->first = second->second;

second->second = second->third;

second->third = nullptr;

if (node->second != nullptr) node->second->parent = node;

}

else if (second == node && first->size == 2) { // пункт 3.2

parent->key[0] = first->key[1];

first->removeFromNode(first->key[1], first->key[1].timesAdded);

if (node->second == nullptr) node->second = node->first;

node->first = first->third;

first->third = nullptr;

if (node->first != nullptr) node->first->parent = node;

}

}

return parent;

}

Node \* FrequencyDictionary::merge(Node \* node)

{

Node \*parent = node->parent;

if (parent->first == node) { // пункт 1

parent->second->insertToNode(parent->key[0], parent->key[0].timesAdded);

parent->second->third = parent->second->second;

parent->second->second = parent->second->first;

if (node->first != nullptr) parent->second->first = node->first;

else if (node->second != nullptr) parent->second->first = node->second;

if (parent->second->first != nullptr) parent->second->first->parent = parent->second;

parent->removeFromNode(parent->key[0], parent->key[0].timesAdded);

delete parent->first;

parent->first = nullptr;

}

else if (parent->second == node) { // пункт 2

parent->first->insertToNode(parent->key[0], parent->key[0].timesAdded);

if (node->first != nullptr) parent->first->third = node->first;

else if (node->second != nullptr) parent->first->third = node->second;

if (parent->first->third != nullptr) parent->first->third->parent = parent->first;

parent->removeFromNode(parent->key[0], parent->key[0].timesAdded);

delete parent->second;

parent->second = nullptr;

}

if (parent->parent == nullptr) { // пункт 3

Node \*temp = nullptr;

if (parent->first != nullptr) temp = parent->first;

else temp = parent->second;

temp->parent = nullptr;

delete parent;

root\_ = temp;

return temp;

}

return parent;

}

void FrequencyDictionary::deleteTree(Node \* node)

{

if (node->first != nullptr)

{

deleteTree(node->first);

}

if (node->second != nullptr)

{

deleteTree(node->second);

}

if (node->third != nullptr)

{

deleteTree(node->third);

}

if (node->size == 2)

{

cout << "deleted " << node->key[0] << ";" << node->key[1] << std::endl;

delete node;

}

else

{

if (node->size == 1)

{

cout << "deleted " << node->key[0] << std::endl;

delete node;

}

}

}

void FrequencyDictionary::deleteTree()

{

deleteTree(root\_);

}

bool FrequencyDictionary::isWord(string& k)

{

if (((\*k.begin() == '(') || (\*k.begin() == '"')) && (k.size() > 1))

{

k.erase(k.begin());

}

if ((\*k.begin() == '"') && (k.size() > 1))

{

k.erase(k.begin());

}

if (((k[k.size() - 1] == ')') || (k[k.size() - 1] == ',') || (k[k.size() - 1] == '.') || (k[k.size() - 1] == '!') || (k[k.size() - 1] == '?') || (k[k.size() - 1] == ':') || (k[k.size() - 1] == ';') || (k[k.size() - 1] == '"')) && (k.size() > 1))

{

k.erase(k.size() - 1);

}

if (((k[k.size() - 1] == ')') || (k[k.size() - 1] == '"')) && (k.size() > 1))

{

k.erase(k.size() - 1);

}

if ((k[k.size() - 1] == '"') && (k.size() > 1))

{

k.erase(k.size() - 1);

}

transform(k.begin(), k.end(), k.begin(), ::tolower);

int sign = 0;

for (int i = 0; k[i] != '\0'; i++)

{

if (isalpha(k[i]) == 0)

{

if ((k[i] == '-'))

{

if (!((i != 0) && (isalpha(k[i - 1]) != 0) && (isalpha(k[i + 1]) != 0)))

{

return false;

}

}

else

{

if (k[i] == 39 && (sign == 0)) // 39 = '

{

sign++;

if (!((i == (k.size() - 2)) && (isalpha(k[i - 1]) != 0) && (isalpha(k[i + 1]) != 0)))

{

return false;

}

}

else

{

return false;

}

}

}

}

return true;

}

void FrequencyDictionary::addToDicrionary(Node \* node, string key)

{

if (node != nullptr)

{

vector<Data>::iterator it = frequensyDict\_.begin();

bool found = false;

int j = 0;

if ((node->size == 2) && (node->key[1] == key))

{

j = 1;

}

for (int i = 0 ; it != frequensyDict\_.end(); it ++, i++)

{

if (frequensyDict\_.at(i) == node->key[j])

{

frequensyDict\_.at(i) = node->key[j];

found = true;

break;

}

}

if (!found)

{

frequensyDict\_.push\_back(node->key[j]);

}

struct greater\_than\_key

{

inline bool operator() (const Data& data1, const Data& data2)

{

return (data1.timesAdded > data2.timesAdded);

}

};

std::sort(frequensyDict\_.begin(), frequensyDict\_.end(), greater\_than\_key());

}

}

void FrequencyDictionary::removeFromDictionary(Node \* node, string key)

{

if (node != nullptr)

{

vector<Data>::iterator it = frequensyDict\_.begin();

bool found = false;

int j = 0;

if ((node->size == 2) && (node->key[1] == key))

{

j = 1;

}

for (int i = 0; it != frequensyDict\_.end(); it++, i++)

{

if (frequensyDict\_.at(i) == node->key[j])

{

frequensyDict\_.at(i) = node->key[j];

found = true;

break;

}

}

if (found)

{

frequensyDict\_.erase(it);

}

struct greater\_than\_key

{

inline bool operator() (const Data& data1, const Data& data2)

{

return (data1.timesAdded > data2.timesAdded);

}

};

std::sort(frequensyDict\_.begin(), frequensyDict\_.end(), greater\_than\_key());

}

}

FrequencyDictionary::~FrequencyDictionary()

{

if (root\_ != nullptr)

{

deleteTree();

}

}

Node::Node(string k) :

size(1),

key{ {k}, {}, {} },

first(nullptr),

second(nullptr),

third(nullptr),

fourth(nullptr),

parent(nullptr)

{}

Node::Node() :

size(0),

key{},

first(nullptr),

second(nullptr),

third(nullptr),

fourth(nullptr),

parent(nullptr)

{}

void Node::sort()

{

if (size == 1)

{

return;

}

if (size == 2)

{

if (key[0] > key[1])

{

std::swap(key[0], key[1]);

}

}

if (size == 3)

{

if (key[0] > key[1])

{

std::swap(key[0], key[1]);

}

if (key[0] > key[2])

{

std::swap(key[0], key[2]);

}

if (key[1] > key[2])

{

std::swap(key[1], key[2]);

}

}

}

void Node::insertToNode(string k)

{

if (find(k))

{

if (k == key[0])

{

key[0].timesAdded++;

}

if (k == key[1])

{

key[1].timesAdded++;

}

}

else

{

key[size] = k;

size++;

sort();

}

}

void Node::insertToNode(string k, int timesAdded)

{

Data newData(k);

if (find(k))

{

if (k == key[0])

{

key[0].timesAdded = timesAdded + 1;

}

if (k == key[1])

{

key[1].timesAdded = timesAdded + 1;

}

}

else

{

key[size] = k;

key[size].timesAdded = timesAdded;

size++;

sort();

}

}

void Node::insertToNode(Data k)

{

if (find(k))

{

if (k == key[0])

{

key[0].timesAdded++;

}

if (k == key[1])

{

key[1].timesAdded++;

}

}

else

{

key[size] = k;

size++;

sort();

}

}

void Node::insertToNode(Data k, int timesAdded)

{

if (find(k))

{

if (k == key[0])

{

key[0].timesAdded = timesAdded + 1;

}

if (k == key[1])

{

key[1].timesAdded = timesAdded + 1;

}

}

else

{

key[size] = k;

key[size].timesAdded = timesAdded;

size++;

sort();

}

}

bool Node::isLeaf()

{

return (first == nullptr) && (second == nullptr) && (fourth == nullptr);

}

bool Node::find(Data k)

{

for (int i = 0; i < size; i++)

{

if (key[i] == k)

{

return true;

}

}

return false;

}

void Node::removeFromNode(Data k)

{

if ((k == key[0] && key[0].timesAdded == 1) || (k == key[1] && key[1].timesAdded == 1) || (k == key[2] && key[2].timesAdded == 1))

{

if (k == key[0])

{

key[0].timesAdded--;

}

if (k == key[1])

{

key[1].timesAdded--;

}

if (k == key[2])

{

key[2].timesAdded--;

}

if (size >= 1 && key[0] == k) {

key[0] = key[1];

key[1] = key[2];

size--;

}

else if (size == 2 && key[1] == k) {

key[1] = key[2];

size--;

}

}

else

{

if (k == key[0])

{

key[0].timesAdded--;

}

if (k == key[1])

{

key[1].timesAdded--;

}

if (k == key[2])

{

key[2].timesAdded--;

}

}

}

void Node::removeFromNode(Data k, int timesAdded)

{

if (k == key[0])

{

key[0].timesAdded -= timesAdded;

}

if (k == key[1])

{

key[1].timesAdded -= timesAdded;

}

if (k == key[2])

{

key[2].timesAdded -= timesAdded;

}

if (size >= 1 && key[0] == k) {

key[0] = key[1];

key[1] = key[2];

size--;

}

else if (size == 2 && key[1] == k) {

key[1] = key[2];

size--;

}

}

Data::Data(string k) :

word(k),

timesAdded(1)

{}

Data::Data() :

word(""),

timesAdded(0)

{}

int Data::compare(string word1, string word2)

{

if (word1 == word2)

{

return 0;

}

for (int i = 0; word1[i] != '\0' || word2[i] != '\0'; i++)

{

if (word1[i] < word2[i])

{

return -1;

}

if (word1[i] > word2[i])

{

return 1;

}

}

if (word1.size() > word2.size())

{

return 1;

}

else

{

return -1;

}

}

bool Data::operator < (Data &other)

{

if (compare(word, other.word) < 0)

{

return true;

}

else

{

return false;

}

}

bool operator < (string &str, Data &data)

{

if (data.compare(str, data.word) < 0)

{

return true;

}

else

{

return false;

}

}

bool Data::operator > (Data &other)

{

if (compare(word, other.word) > 0)

{

return true;

}

else

{

return false;

}

}

bool operator > (string &str, Data &data)

{

if (data.compare(str, data.word) > 0)

{

return true;

}

else

{

return false;

}

}

bool Data::operator <= (string other)

{

if (compare(word, other) <= 0)

{

return true;

}

else

{

return false;

}

}

bool Data::operator >= (string other)

{

if (compare(word, other) >= 0)

{

return true;

}

else

{

return false;

}

}

bool Data::operator == (string other)

{

return (word == other);

}

bool operator == (Data &dataL, Data &dataR)

{

return (dataL.word == dataR.word);

}

bool operator == (string &other, Data& data)

{

return (other == data.word);

}

std::ostream& operator<<(std::ostream & out, Data &data)

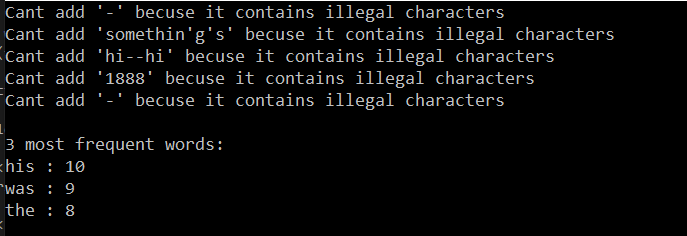
{

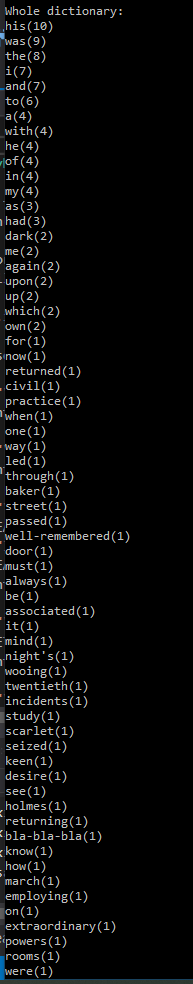
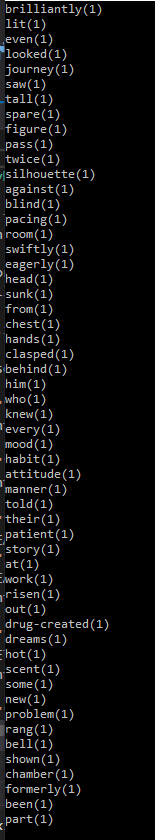
out << data.word;

return out;

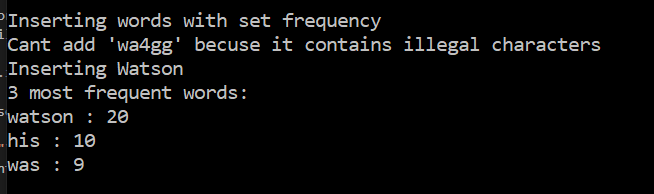
}

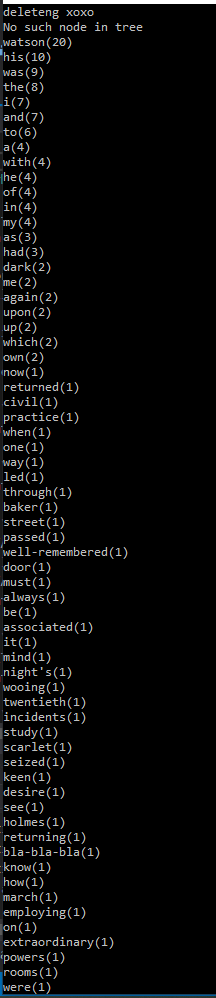
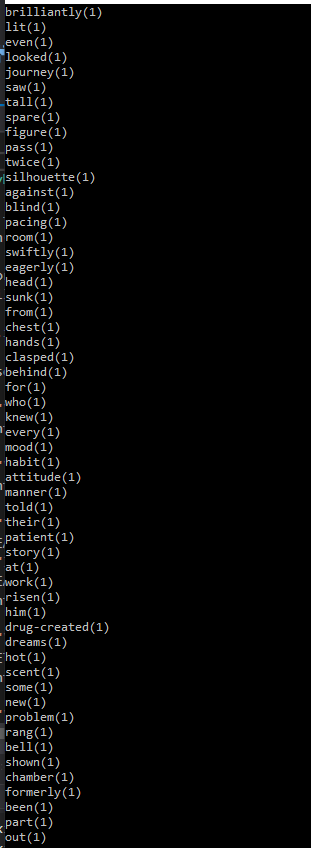
# Приложение 2

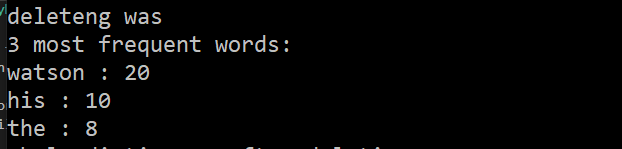
После чтения файла:

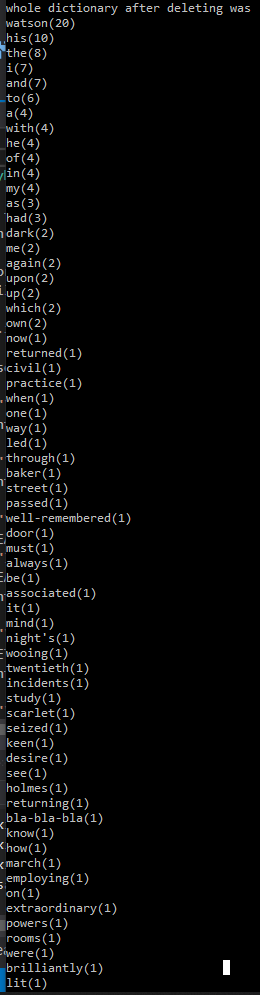
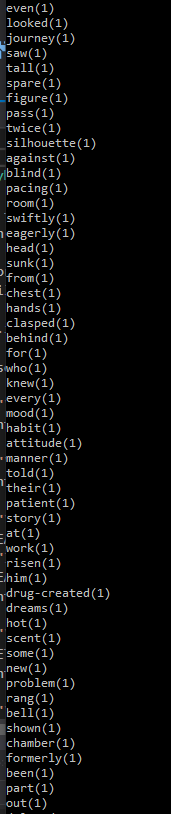
Весь словарь после чтения файла:

После добавления слов с заданной частотой встречаемости:



Удаление несуществующего слова и вывод словаря, который не изменился:

Удаление существующего слова:

Весь словарь после удаления was:

Текст2 для демонстрации 2-3 дерева:

One night's - it was on the somethin'g's bla-bla-bla hi--hi twentieth "of" ("as"), March, 1888 - I was retUrning

A screen shot of a smart phone

Description automatically generated