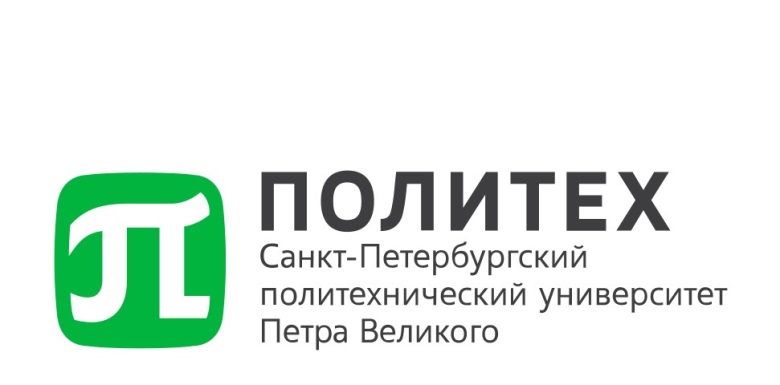
**Санкт-Петербургский политехнический университет Петра Великого**

**Институт компьютерных наук и кибербезопасности**

**Высшая школа программной инженерии**

****

**КУРСОВАЯ РАБОТА**

**Алгоритмы работы со словарями**

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

Выполнил

студент гр. 5130904/30002 Севостьянова А.В.

Руководитель Череповский Д.К.

« 05 » мая 2024г.

Санкт-Петербург

2024 г

Оглавление

[Введение. Общая постановка задачи 3](#_Toc166407061)

[Основная часть работы 4](#_Toc166407062)

[1. Описание алгоритма решения и используемых структур данных 4](#_Toc166407063)

[2. Анализ алгоритма 7](#_Toc166407064)

[3. Описание спецификации программы (детальные требования) 8](#_Toc166407065)

[4. Описание программы (структура программы, формат входных и выходных данных) 9](#_Toc166407066)

[Заключение 12](#_Toc166407067)

[Список использованных источников 13](#_Toc166407068)

[Приложение 1. Текст программы 14](#_Toc166407069)

[Приложение 2. Протоколы отладки 23](#_Toc166407070)

# 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 1.3.2***

***Частотный словарь. АВЛ-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 АВЛ-дерево.

Узел АВЛ-дерева должен содержать:

• Ключ – слово

• Показатель (фактор) сбалансированности

• Информационная часть – количество слов

# Основная часть работы

1. **Описание алгоритма решения и используемых структур данных**

*АВЛ-дерево* – сбалансированное по высоте двоичное дерево поиска, для каждой из вершин которого разность высот двух поддеревьев не превышает 1.

*Частотный словарь* – набор слов, вместе с частотой их встречаемости.

Определим понятие «слово» для данного словаря. *Слово – слово естественного языка, представленное в формате данных string.* Ключом данного дерева является определенное нами понятие «слова». Построение дерева будет происходить на основе алфавитного порядка.

Узел дерева – структура Node, содержащая следующие поля:

* key\_ (значение ключа)
* cnt\_ (количество повторений данного слова в тексте)
* balance\_ (фактор сбалансированности данного узла)
* right\_ (указатель на правое поддерево)
* left\_ (указатель на левое поддерево)

Дерево строится на основе класса AVL\_Tree, содержащего указатель на его вершину (root), начальное значение которого равно нулевому указателю.

Дерево поддерживает 5 методов:

1. *search (const std::string word)* – поиск записи с указанным значением

Данный метод основывается на методе searchIterative(const std::string word), который возвращает указатель на узел, если запись с указанным значение присутствует в структуре или нулевой указатель, если такая запись отсутствует.

Результат работы метода search – вывод указанного значения с информацией о его встречаемости, если слово находится в словаре, иначе сообщение о том, что запись не найдена. При попытке поиска в пустом словаре возникнет исключение *ERROR\_IS\_EMPTY\_TREE* (пустой словарь).

1. *insert( const std::string word)* – добавление записи с указанным ключом

Выполняется на основе метода insertRecoursive(Node\*& node, const std::string word), который сравнивает значение ключа текущего узла со вставляемым значением. Если значения совпадают, то количество повторений (cnt\_) увеличивается на единицу, иначе если в алфавитном порядке добавляемое значение идет раньше значения ключа, то метод рекурсивной вставки вызывается для левого поддерева, в противном случае для правого поддерева. Если при обходе всего дерева данное значение не было найдено, то создается новый узел и происходит перебалансировка дерева.

При перебалансировке дерева setBalance(Node\*& node) для данного узла с помощью метода getHightOfSubTree(Node\* node), который определяет высоту поддерева, подсчитывается разность левого и правого поддеревьев. Если данное значение превосходит единицу, то выполняется поворот для данного узла (левый, правый, поворот в лево-право, поворот в право-лево), в зависимости от положения добавляемого узла.

В итоге, метод insert добавляет в дерево новый узел, если переданное в качестве параметра значение не было найдено или увеличивает частоту встречаемости для данного значения.

1. *remove(const std::string word)* – удаление узла с заданным значением

Работает на основе метода removeRecoursive(Node\*& node, const std::string word), который удаляет узел дерева в зависимости от наличия у него потомков (лист, правое поддерево, левое поддерево, и правое, и левое поддерево). После удаления происходит перебалансировка дерева с помощью метода setBalanceAfterRemove(), для реализации которого была написана структура данных *stack*, на основе указателей (элементы стека – указатели типа AVL\_Tree::Node).

Метод remove удаляет узел с заданным значением, если оно было найдено, или выводит сообщение о невозможности удалении элемента, так как он отсутствует в дереве. При попытке удаления из пустого дерева вызывается исключение *ERROR\_IS\_EMPTY\_TREE* (пустой словарь).

1. *output ()* – вывод дерева по уровням

Работает на основе метода outputRecoursive(Node\*& node, level), который осуществляет вывод каждого уровня дерева. Для подсчета уровней дерева используется результат работы метода getHightOfSubTree, увеличенный на единицу.

Метод output выводит уровни дерева на консоль. Если происходит вывод пустого дерева, то выводится сообщение о том, что дерево пустое.

1. *findMax()* – вывод трех наиболее часто встречающихся слов

Работает на основе метода findMaxRecoursive(Node\* node, Node\* arr[SIZE]), который осуществляет обход дерева и добавляет указатели на узлы с наиболее часто встречающимися значениями ключа в массив указателей, состоящий из трех элементов.

Метод findMax находит 3 наиболее часто встречающихся значения ключа и производит их вывод на консоль с информацией о частоте повторений данного значения в тексте. Если количество узлов дерева меньше 3, то возникает ошибка *ERROR\_IN\_THE\_NUMBER\_OF\_ELEMENTS* (количество элементов меньше 3).

1. **Анализ алгоритма**

*Время выполнения:*

• Поиск слова в АВЛ-дереве: O(log n)

• Вставка слова в АВЛ-дерево: O(log n)

• Увеличение частоты слова: O(1)

Поскольку алгоритм выполняет эти операции для каждого слова в тексте, а длина текста в среднем составляет n слов, общая сложность алгоритма равна O(n log n).

*Память:*

Алгоритм хранит АВЛ-дерево, которое содержит слова и их частоты в тексте. Размер АВЛ-дерева зависит от количества уникальных слов в тексте. В среднем он составляет O(n), где n — количество уникальных слов.

1. **Описание спецификации программы (детальные требования)**
2. Поиск возможен только в непустом словаре

* Если дерево пустое, сообщение об ошибке, завершение работы программы.
* Иначе вывод информации о наличии/отсутствие значения в словаре.

1. Удаление элементов возможно исключительно из непустого словаря

* Если дерево пустое, сообщение об ошибке, завершение работы программы.
* Иначе удаление узла с данным значение из словаря или сообщение об отсутствии записи.

1. Поиск 3 наиболее часто встречающихся слов возможен только тогда, когда количество элементов в словаре больше 3

* Если количество элементов меньше 3, сообщение об ошибки, завершение работы программы.
* Иначе поиск 3 наиболее часто встречающихся значений ключа, их вывод на консоль.

1. Выбор пользователя должен быть корректен

* Если выбор некорректен, сообщение об ошибке, завершение работы программы.
* Иначе продолжение работы программы.

1. **Описание программы (структура программы, формат входных и выходных данных)**

Подключение библиотек и заголовочных файлов

#include<exception>

#include<fstream>

#include<sstream>

#include “Structs.h”

Тестирование программы происходит с помощью функции *testWithExample()* с небольшим количеством слов для более наглядного результата работы методов (тесты для всех методов) и сгенерированного файла со словами *test.txt* (тестирование методов insert, findMax, так как корректность работы других методов сложно отследить, их корректность проверяем при тестировании с помощью функции testWithExample).

В начале работы программы происходит тестирование всех написанных методов с помощью функции, затем подключается файл с текстом, содержащим большее количество слов.

*Входные данные:*

* Выбор пользователя тестирования с корректным/некорректным результатом работы программы (строка типа string)
* Последовательности символов из сгенерированного файла (string).

Выходные данные:

* Информация о возникшей ошибки при тестировании для некорректных случаев (строка типа string)
* Информация о записях словаря (значение – строка типа string, информация о встречаемости – число типа int, фактор сбалансированности –число [-1;1] типа int)
* Информация о трех наиболее часто встречающихся словах (значение – строка типа string, частота встречаемости – число типа int)

***Тест план***

|  |  |  |  |
| --- | --- | --- | --- |
| ***Требования*** | ***Детальные требования*** | ***Данные*** | ***Ожидаемый результат*** |
| 1. Открытие файла с текстом для тестирования должно быть корректно. | Файл не открыт или не найден  Открытие файла произошло корректно | -  - | Сообщение:  “File not open”  Аварийное завершение работы программы.  Продолжение работы программы |
| 2. Выбор пользователя должен быть корректен. | Выбор некорректен  Корректный выбор | Т  Y(y, N, n) | Сообщение:  “Ircorrect input”  Аварийное завершение работы программы.  Продолжение работы программы. |
| 3. Поиск возможен только в непустом дереве. | Дерево не содержит элементов  В дереве есть элементы   1. Элемент с заданным значение найден 2. Элемент с заданным значением не найдет | “word”  “word”  “banana” | Сообщение:  “Vocabulary is empty”  Аварийное завершение программы.   1. Сообщение:   “word : 1”  Продолжение работы программы   1. Сообщение:   “banana is not found in vocabulary”  Продолжение работы программы |
| 4. Удаление элементов возможно исключительно из непустого дерева. | Дерево не содержит элементов  В дереве есть элементы   1. Элемент с заданным значение найден 2. Элемент с заданным значением не найдет | “word”  “watch”  “banana” | Сообщение:  “Vocabulary is empty”  Аварийное завершение программы.   1. Сообщение:   “word(1,0)”  Продолжение работы программы   1. Сообщение:   “banana is not found”  Продолжение работы программы |
| 5. Поиск 3 наиболее часто встречающихся слов возможен только тогда, когда количество элементов в дереве больше 3. | В дереве меньше 3 элементов  В дереве больше 3 элементов | -  - | Сообщение:  “The number of elements is less than 3”  Аварийное завершение программы.  Сообщение:  “3 most common words:  apple: 2  word: 1  banana: 1”  Продолжение работы программы |

# Заключение

В ходе работы были изучено сбалансированное дерево поиска (АВЛ-дерево), основы его построения, а также понятие частотного словаря. Был разработан частотный словарь, реализованный на основе АВЛ-дерева, написаны методы вставки, поиска, удаления заданного значения в словарь, вывода дерева по уровням, поиска 3 наиболее часто встречающихся слов. Написана структура данных стек (на основе указателей), хранящая значения ссылок на узел дерева.

# Список использованных источников

1. Никлаус Вирт: Алгоритмы и структуры данных. Новая версия для Оберона + CD / Пер. с англ. Ткачев Ф. В. – М.: ДМК Пресс, 2010 – 272с
2. Сенюкова О.В. : Сбалансированные деревья поиска: Учебно-методическое пособие. – М.: Издательский отдел факультета ВМиК МГУ имени М.В. Ломоносова (лицензия ИД N 05899 от 24.09.2001 г.); МАКС Пресс, 2014 – 68 c..: ил.

# Приложение 1. Текст программы

***Main.cpp***

#include <iostream>

#include<exception>

#include<fstream>

#include<sstream>

#include"AVL-TREE.h"

const std::string IRCORRECT\_INPUT = "Ircorrect input\n";

const std::string ERROR\_OF\_OPENING = "File not open\n";

void testWithExample();

int main()

{

try

{

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_trial testing\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";

testWithExample();

std::cout << "\n\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_the main work of the program\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";

std::ifstream file("test.txt");

if (!file.is\_open()) throw std::runtime\_error(ERROR\_OF\_OPENING);

AVL\_Tree tree;

std::string line;

std::string word;

while (std::getline(file,line))

{

std::istringstream iss (line);

while(iss >> word)

tree.insert(word);

}

tree.findMax();

}

catch (const std::exception& ex)

{

std::cerr << ex.what() << std::endl;

return EXIT\_FAILURE;

}

return EXIT\_SUCCESS;

}

void testWithExample()

{

AVL\_Tree tree;

std::cout << "Do you want to see a negative search option?\n(y/n):\t";

std::string choice;

std::cin >> choice;

if (choice.length() > 1) throw std::runtime\_error(IRCORRECT\_INPUT);

switch (choice[0])

{

case 'y': case 'Y':

tree.search("word");

break;

case 'n': case 'N':

tree.insert("word");

tree.output();

tree.search("word");

tree.search("banana");

tree.remove("word");

break;

default:

throw std::runtime\_error(IRCORRECT\_INPUT);

}

std::cout << "\nVocabulary before insert elements:\n\t";

tree.output();

std::cout << "Do you want to see a negative remove option?\n(y/n):\t";

std::cin >> choice;

if (choice.length() > 1) throw std::runtime\_error(IRCORRECT\_INPUT);

switch (choice[0])

{

case 'y': case 'Y':

tree.remove("word");

break;

case 'n': case 'N':

tree.insert("word");

tree.insert("watch");

tree.output();

tree.remove("watch");

tree.output();

tree.remove("banana");

break;

default:

throw std::runtime\_error(IRCORRECT\_INPUT);

}

tree.insert("apple");

tree.insert("banana");

tree.insert("apple");

std::cout << "\nVocabulary after insert elements:\n\t";

tree.output();

std::cout << "Do you want to see a negative search option for the 3 maximum elements?\n(y/n):\t";

std::cin >> choice;

if (choice.length() > 1) throw std::runtime\_error(IRCORRECT\_INPUT);

switch (choice[0])

{

case 'y': case 'Y':

tree.remove("banana");

std::cout << "Vocabulary after deleting the element:\n\t";

tree.output();

tree.findMax();

break;

case 'n': case 'N':

tree.findMax();

break;

default:

throw std::runtime\_error(IRCORRECT\_INPUT);

}

}

***Structs.h***

#ifndef AVL\_TREE\_H

#define AVL\_TREE\_H

#include<iostream>

#include<string>

const int SIZE = 3;

const std::string ERROR\_IS\_EMPTY\_TREE = "Vocabulary is empty\n";

const std::string ERROR\_IN\_THE\_NUMBER\_OF\_ELEMENTS = "The number of elements is less than 3\n";

const std::string ERROR\_STACK\_UNDERFLOW = "Srack underflow\n";

class AVL\_Tree

{

friend class Stack;

struct Node

{

Node(std::string key) :

key\_(key), cnt\_(1), balance\_(0), right\_(nullptr), left\_(nullptr) {};

std::string key\_;

int cnt\_;

int balance\_;

Node\* right\_;

Node\* left\_;

};

public:

AVL\_Tree() : root(nullptr) {};

~AVL\_Tree();

void search(const std::string& word) const;

void insert(const std::string& word);

void remove(const std::string& word);

void output();

void findMax();

private:

Node\* root;

void deleteNodes(Node\* node);

Node\* searchIterative(const std::string& word)const;

int getHightOfSubTree(Node\* node)const;

void setBalance(Node\*& node)const;

void turnLeft(Node\*& node);

void turnRigth(Node\*& node);

void insretRecoursive(Node\*& node, const std::string& word);

void removeRecoursive(Node\*& node, const std::string& word)const;

void setBalanceAfterRemove(Node\* node);

void outputRecoursive(Node\*& node, int level)const;

int countOfNodes(Node\* node)const;

void findMaxRecoursive(Node\* node, Node\* arr[SIZE])const;

};

class Stack

{

struct NodeStack

{

AVL\_Tree::Node\* key\_;

NodeStack\* next\_;

NodeStack(AVL\_Tree::Node\* key, NodeStack\* next = nullptr) : key\_(key), next\_(next) {};

};

public:

Stack() : top(nullptr) {};

~Stack();

bool isEmpty() const;

void push(AVL\_Tree::Node\* key);

AVL\_Tree::Node\* getTop() const;

void pop();

private:

NodeStack\* top;

};

#endif

***AVL-TREE.cpp***

#include "Structs.h"

AVL\_Tree::~AVL\_Tree() { deleteNodes(root); }

void AVL\_Tree::deleteNodes(Node\* node)

{

if (!node) return;

else

{

deleteNodes(node->left\_);

deleteNodes(node->right\_);

delete node;

}

}

void AVL\_Tree::search(const std::string& word)const

{

if (!root) throw std::runtime\_error(ERROR\_IS\_EMPTY\_TREE);

else

if (!searchIterative(word)) std::cout << word <<" is not found in vocabulary\n";

else std::cout << word << " : " << searchIterative(word)->cnt\_<<"\n";

}

AVL\_Tree::Node\* AVL\_Tree::searchIterative(const std::string& word)const

{

if (!root) return nullptr;

else

{

Node\* ptr = root;

while (ptr && ptr->key\_!=word)

{

if (ptr->key\_ > word) ptr = ptr->left\_;

else ptr = ptr->right\_;

}

return ptr;

}

}

int AVL\_Tree::getHightOfSubTree(Node\* node)const

{

if (!node) return 0;

else return 1 + std::max(getHightOfSubTree(node->left\_), getHightOfSubTree(node->right\_));

}

void AVL\_Tree::setBalance(Node\*& node)const

{

node->balance\_ = getHightOfSubTree(node->right\_) - getHightOfSubTree(node->left\_);

}

void AVL\_Tree::turnLeft(Node\*& node)

{

Node\* rightSubTree = node->right\_;

Node\* rightSubTreeLeftSubTree = rightSubTree->left\_;

rightSubTree->left\_ = node;

node->right\_ = rightSubTreeLeftSubTree;

node = rightSubTree;

setBalance(node->left\_);

setBalance(node);

}

void AVL\_Tree::turnRigth(Node\*& node)

{

Node\* leftSubTree = node->left\_;

Node\* leftSubTreeRightSubTree = leftSubTree->right\_;

leftSubTree->right\_ = node;

node->left\_ = leftSubTreeRightSubTree;

node = leftSubTree;

setBalance(node->right\_);

setBalance(node);

}

void AVL\_Tree::insert(const std::string& word) { insretRecoursive(root, word); }

void AVL\_Tree::insretRecoursive(Node\*& node,const std::string& word)

{

if (!node)

node = new Node(word);

else

{

if (searchIterative(word)) searchIterative(word)->cnt\_++;

else

{

if (node->key\_ < word)

{

insretRecoursive(node->right\_, word);

if (getHightOfSubTree(node->right\_) - getHightOfSubTree(node->left\_) > 1)

{

if (getHightOfSubTree(node->right\_->right\_) < getHightOfSubTree(node->right\_->left\_))

turnRigth(node->right\_);

turnLeft(node);

}

}

else

{

insretRecoursive(node->left\_, word);

if (getHightOfSubTree(node->left\_) - getHightOfSubTree(node->right\_) > 1)

{

if (getHightOfSubTree(node->left\_->left\_) < getHightOfSubTree(node->left\_->right\_))

turnLeft(node->left\_);

turnRigth(node);

}

}

setBalance(node);

}

}

}

void AVL\_Tree::remove(const std::string& word)

{

if (!root) throw std::runtime\_error(ERROR\_IS\_EMPTY\_TREE);

if (!searchIterative(word)) std::cerr << word << " is not found\n";

else

{

removeRecoursive(root, word);

if (root) setBalanceAfterRemove(root);

}

}

void AVL\_Tree::removeRecoursive(Node\*& node,const std::string& word) const

{

if (!node) return;

else

{

if (node->key\_ < word) removeRecoursive(node->right\_, word);

else

if (node->key\_ > word) removeRecoursive(node->left\_, word);

else

{

if (!node->left\_ && !node->right\_)

{

delete node;

node = nullptr;

}

else if (!node->right\_)

{

Node\* ptr = node;

node = node->left\_;

delete ptr;

}

else if (!node->left\_)

{

Node\* ptr = node;

node = node->right\_;

delete ptr;

}

else

{

Node\* minNode = node->right\_;

while (minNode->left\_) minNode = minNode->left\_;

node->key\_ = minNode->key\_;

removeRecoursive(node->right\_, minNode->key\_);

}

}

if (node)setBalance(node);

}

}

void AVL\_Tree::setBalanceAfterRemove(Node\* node)

{

Stack stack;

Node\* ptr = root;

while (node->key\_!= ptr->key\_)

{

stack.push(ptr);

if (ptr->key\_ < node->key\_) ptr = ptr->right\_;

else ptr = ptr->left\_;

}

while (ptr)

{

Node\* change = nullptr;

if (abs(ptr->balance\_) > 1)

{

if (ptr->balance\_ > 1)

{

if (ptr == root) turnLeft(root);

else

{

turnLeft(ptr);

change = ptr;

}

}

else

{

if (ptr == root) turnRigth(root);

else

{

turnRigth(ptr);

change = ptr;

}

}

}

if (stack.isEmpty()) break;

else

{

ptr = stack.getTop();

if (change && ptr->key\_ > change->key\_ && ptr->left\_ != change) ptr->left\_ = change;

if(change && ptr->key\_ < change->key\_ && ptr->right\_ != change) ptr->right\_ = change;

setBalance(ptr);

stack.pop();

}

}

if (node->right\_)setBalanceAfterRemove(node->right\_);

if (node->left\_)setBalanceAfterRemove(node->left\_);

}

int AVL\_Tree::countOfNodes(Node\* node)const

{

if (!node) return 0;

return countOfNodes(node->right\_) + countOfNodes(node->left\_) + 1;

}

void AVL\_Tree::outputRecoursive( Node\*& node, int level) const

{

if (!node) return;

else

{

if (level == 1)

std::cout << node->key\_ <<"("<<node->cnt\_<<","<<node->balance\_<< ") ";

else if (level > 1)

{

outputRecoursive(node->left\_, level - 1);

outputRecoursive(node->right\_, level - 1);

}

}

}

void AVL\_Tree::output()

{

if (!root) std::cerr << ERROR\_IS\_EMPTY\_TREE << "\n";

else

{

int height = getHightOfSubTree(root) + 1;

for (int level = 1; level <= height; level++)

{

outputRecoursive(root, level);

std::cout << std::endl;

}

}

}

void AVL\_Tree::findMax()

{

if (countOfNodes(root) < 3) throw std::runtime\_error(ERROR\_IN\_THE\_NUMBER\_OF\_ELEMENTS);

Node\* arr[SIZE]{ nullptr };

findMaxRecoursive(root, arr);

std::cout << "3 most common words:\n";

for (auto el : arr)

std::cout << el->key\_ << ": " << el->cnt\_ << "\n";

}

void AVL\_Tree::findMaxRecoursive(Node\* node, Node\* arr[SIZE])const

{

if (!node) return;

else

{

if (!arr[0]) arr[0] = node;

else

if (arr[0]->cnt\_ <= node->cnt\_)

{

for (int i = SIZE - 1; i > 0; --i)

arr[i] = arr[i - 1];

arr[0] = node;

}

findMaxRecoursive(node->right\_, arr);

findMaxRecoursive(node->left\_, arr);

}

}

***Stack.cpp***

#include"Structs.h"

Stack::~Stack()

{

if (top)

{

NodeStack\* previous = top;

NodeStack\* ptr = top->next\_;

while (ptr)

{

delete previous;

previous = ptr;

ptr = ptr->next\_;

}

delete previous;

}

}

bool Stack::isEmpty() const { return !top ? true : false; }

void Stack::push(AVL\_Tree::Node\* key)

{

NodeStack\* new\_el = new NodeStack(key);

if (!top) top = new\_el;

else

{

NodeStack\* ptr = top;

while (ptr->next\_) ptr = ptr->next\_;

ptr->next\_ = new\_el;

}

}

AVL\_Tree::Node\* Stack::getTop() const

{

if (top)

{

NodeStack\* ptr = top;

while (ptr->next\_) ptr = ptr->next\_;

return ptr->key\_;

}

else throw std::runtime\_error(ERROR\_STACK\_UNDERFLOW);

}

void Stack::pop()

{

if (top)

{

NodeStack\* previous = top;

if (previous->next\_)

{

NodeStack\* ptr = top->next\_;

while (ptr->next\_)

{

previous = previous->next\_;

ptr = ptr->next\_;

}

previous->next\_ = nullptr;

delete ptr;

}

else

{

top = nullptr;

delete previous;

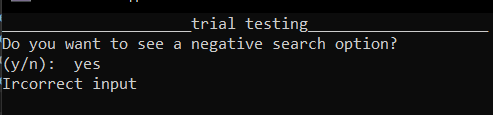
}

}

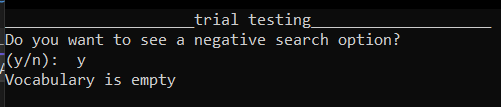
else throw std::runtime\_error(ERROR\_STACK\_UNDERFLOW);

}

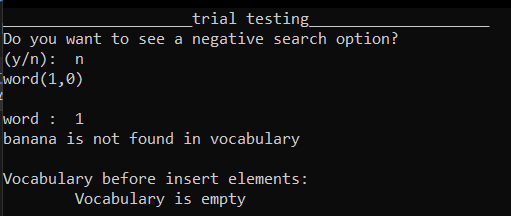
# Приложение 2. Протоколы отладки

****

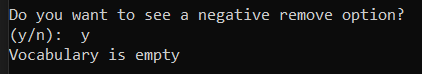
*Рис1.Некорректный пользовательский ввод*

****

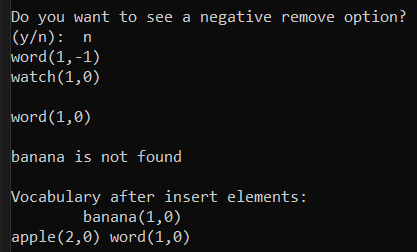
*Рис2. Поиск в пустом словаре*

****

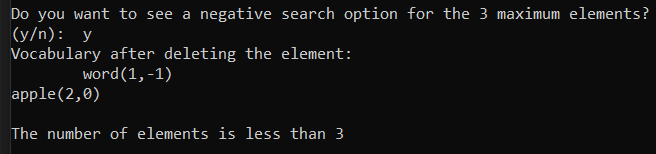
*Рис3. Корректный поиск словаре*

****

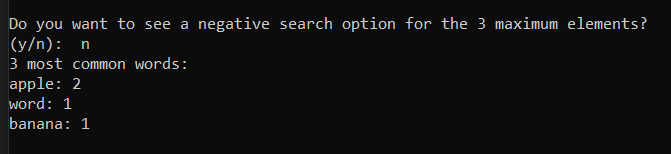
*Рис4. Удаление из пустого словаря*

****

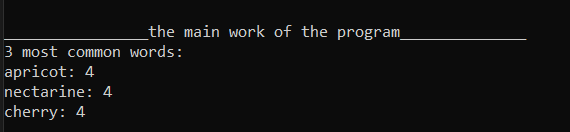
*Рис5.Корректное удаление из словаря*

****

*Рис 6. Поиск наиболее часто встречающихся слов, когда в словаре меньше 3 элементов*

****

*Рис7.Корректный поиск наиболее часто встречающихся слов*

****

*Рис8.Тестирование с данными из файла*