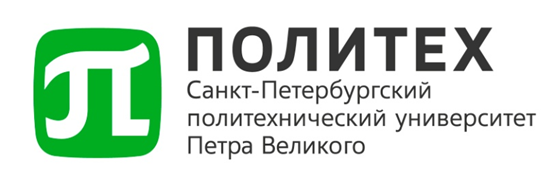
****

**РАСЧЕТНО-ГРАФИЧЕСКАЯ РАБОТА**

**Алгоритмы работы со словарями**

по дисциплине «**Практикум по программированию**»

Выполнил

студент гр. 3530904/00006 Смирнов Е. А.

Руководитель  Павлов Е. А.

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2021

Содержание

[Введение. Общая постановка задачи 3](#_Toc72172792)

[Основная часть работы 4](#_Toc72172793)

[1. Описание алгоритма решения и используемых структур данных 4](#_Toc72172794)

[2. Анализ алгоритма 11](#_Toc72172795)

[3. Описание спецификации программы (детальные требования) 14](#_Toc72172796)

[4. Описание программы (структура программы, форматы входных и выходных данных) 16](#_Toc72172797)

[Заключение. 18](#_Toc72172798)

[Список использованных источников 19](#_Toc72172799)

[Приложение 1. Текст программы 20](#_Toc72172800)

[Приложение 2. Протоколы отладки 21](#_Toc72172801)

# Введение. Общая постановка задачи

***Тема: Алгоритмы работы со словарями***

1. Для разрабатываемого словаря реализовать основные операции:

* INSERT (ключ, значение) – добавить запись с указанным ключом и значением
* SEARCH (ключ)- найти запись с указанным ключом
* DELETE (ключ)- удалить запись с указанным ключом

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

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

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

***Вариант 1.1.4.***

***Англо-русский словарь. Красно-черное дерево.***

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

Узел дерева должен содержать:

• Ключ – английское слово,

• Информационная часть – ссылка на список, содержащий переводы английского слова, отсортированные по алфавиту (переводов слова может быть несколько.

# Основная часть работы

## Описание алгоритма решения и используемых структур данных

Для разрабатываемого словаря реализованы две структуры данных.

**Красно-чёрное дерево** (англ. Red-black tree, RB-Tree) — самобалансирующееся двоичное дерево поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и позволяющее быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность - дополнительный атрибут узла дерева — «цвета» - бит цвета. Одно из двух возможных значений — «чёрный» или «красный».

Красно-черные деревья относятся к сбалансированным бинарным деревьям поиска. Поэтому красно-черное дерево обладает свойствами бинарного и сбалансированного дерева – свойства:

1. Оба поддерева являются бинарными деревьями поиска.
2. Для каждого узла с ключом k выполняется критерий упорядочения: “ключи всех левых потомков” < k < “ключи всех правых потомков”
3. Дерево, в котором заполнены все уровни, возможно кроме последнего.

roma

london

california

N

I

L

N

I

L

africa

N

I

L

N

I

L

belgrad

oslo

N

I

L

N

I

L

madrid

N

I

L

N

I

L

prague

ufa

talin

N

I

L

N

I

L

spb

N

I

L

N

I

L

tula

warsawa

N

I

L

N

I

L

viena

N

I

L

N

I

L

zagreb

Представление красно-черного дерева

Свойства красно-черных деревьев:

1. Узел может быть либо красным, либо чёрным и имеет двух потомков
2. Корень — как правило чёрные.
3. Все листья (так называемые NULL-узлы) — чёрные и не содержат данных
4. Оба потомка каждого красного узла — чёрные
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов. (Чёрные высоты всех листьев равны)

Высота дерева, измеряемая только по чёрным нодам, называется **чёрной высотой***.* В красно-черном дереве с черной высотой hb количество внутренних вершин не менее .

Красно-чёрное дерево с N ключами имеет высоту h=O(logN).

Благодаря этим свойствам, путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего. *Базовые операции*: вставка, удаление и поиск - требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.

Вставка

Добавление нового узла в красно-чёрное дерево происходит следующим образом: находится место для вставки, узел окрашивается в красный цвет и к нему прикрепляется два nill листа. (Nill листья – абстракция, служат лишь для удобства последующих операции). Как только произошла вставка, возможно произошло нарушение свойств красно-черного дерева, так как узел после вставки окрашивается в красный цвет.

Возможны три случая. При вставке узла может нарушиться свойство 2, которое требует, чтобы корень красно-черного дерева был черным, и другое свойство 4, согласно которому красный узел не может иметь красного потомка. Свойство 5 (Все пути от любого узла до листовых узлов содержат одинаковое число чёрных узлов) может нарушиться только при добавлении чёрного узла, при перекрашивании красного узла в чёрный (или наоборот), или при повороте.

Удаление

Удаление оказывается несколько более сложной задачей, чем вставка.

Процесс удаления также, как и вставка предлагается разделить на 2 последовательных процесса.

Первым делом удаляется узел как в обычном бинарном дереве:

* если у удаляемого узла нет детей, то просто его удаляем
* если у удаляемого узла 1 ребенок, меняем указатель на родителя на указатель на ребенка
* если у удаляемого узла 2 ребенка, то ищем самый левый узел в правом поддереве и меняем его значения на значения удаляемого, и затем удаляем.

Естественно, после таких изменений в красно-черном дере нужна балансировка. Если узел был красным, при его удалении узла «красно-черные свойства» сохраняются по следующим причинам.

* ни одна черная высота в дереве не меняется
* никакие красные узлы не делаются смежными

Если удаляемый узел был черным, то свойства красно-черного дерева могут быть нарушены - одно или несколько. Во-первых, если удаленный узел был корнем (новый корень - красный потомок), нарушается свойство 2. Во-вторых, если у удаленного узла был красный родитель, то нарушается свойство 4. В-третьих, перемещение узлов в дереве приводит к тому, что любой простой путь, ранее содержавший одинаковое количество черных вершин, может таковым не быть. Таким образом, нарушается свойство 5.

Поиск

Для поиска элемента в черно-красном дереве поиска можно воспользоваться поиском, как в бинарном дереве. Для каждого узла сравниваем значение его ключа с искомым ключом. Если ключи одинаковы, узел найден, в противном случае рекурсивно смотрим для левого или правого поддерева, пока не найдем нужный узел, если он есть в дереве.

Повороты

Операции удаления и вставки над деревом поиска, «ломают» свойства красно-черного дерева. Для их восстановления создается балансировка - необходимо изменить цвета некоторых узлов дерева, а также расположение узлов (указателей). Если цвет узла изменить – довольно легкая задача, то изменить в структуру указателей в поддереве – нелегкая задача. Для этого придуманы левый и правый поворот – операции, сохраняющие свойство бинарного дерева поиска и помогающие восстановить эти свойства. Иллюстрации работы левого и правого поворота изображены ниже:

Меньше, чем X

Меньше, чем Y

Больше, чем Y

Больше, чем Y

Меньше, чем X

Больше, чем X, меньше, чем Y

Левый поворот

Правый поворот

Повороты

Важно уточнить, что, если узел «ребенок» - *X принимает цвет своего родителя – Y*, то *цвет родителя всегда определяется как красный, независимо от предыдущего цвета потомка.*

Структура **Node** для описания элемента дерева:

enum Color { RED, BLACK }; - перечисление для обозначения цвета узла

struct Node

{

Node(std::string&, List&); конструктор для создания узла

std::string key\_; английское слово

List\* translation\_; список, содержащий переводы слова

Node\* left\_; указатель на левое поддерево

Node\* right\_; указатель на правое поддерево

Node\* parent\_; указатель на родителя

RBTree::Color color\_; цвет узла

};

Поле:

Node\* root\_; - корень дерева

Методы

public:

RBTree(); - конструктор по умолчанию

~RBTree(); - деструктор

void insertKey(std::string&, List&); - вставка узла

void deleteKey(std::string&); - удаления узла по ключу

void searchKey(std::string&); - поиск информации узла и ее вывод

void print(std::ostream&); - демонстрация хранения данных в дереве

private:

Node\* insertNode(Node\* root, Node\* node) - вставка узла в дерево, как в бинарном дереве поиске

void deleteSubtree(Node\* node); - удаляет поддерево

Node\* findNode(std::string& key); - ищет узел по ключу

void printNode(std::ostream& out, Node\* root); - выводит узел

void fixInsert(Node\*); балансировка дерева при вставке

void rotateLeft(Node\*); - левый поворот

void rotateRight(Node\*); - правый поворот

Node\* findUncle(Node\* node); - возвращает указатель на дядю

void deleteNode(Node\* node); - удаление узла из дерева

void fixDelete(Node\*); - балансировка дерева при удалении

void swapValues(Node\* node1, Node\* node2); - меняет значения узлов

Node\* findSuccessor(Node\* node); ); - находит узел, след. по значению

Node\* findSibling(Node\* node); - возвращает указатель, на брата

Node\* findReplacement(Node\* node); - находит узел-замену при удалении

**List** - собственный класс для работы с упорядоченной последовательностью элементов. Реализован, как односвязный список, состоящий из узлов, каждый из которых содержит собственно данные, и одну ссылку на следующий узел списка. Таким образом, набор произвольных адресов памяти объединяется в «цепочку».

Структура **Node** для описания элемента списка:

struct Node

{

Node\* next; - указатель на след. элемент списка

std::string value; - значение элемента

Node(std::string value, Node\* next = nullptr) - конструктор для создания нового элемента

};

**Поля класса List:**

Node\* head\_; - первый элемент списка

int size\_; - количество элементов

**Методы:**

List() noexcept; - конструктор по умолчанию

~List(); - деструктор

std::string& operator[](const int); - оператор индексирования

const std::string& operator[] (const int) const; - оператор индексирования

List& operator+ (List&); - оператор ‘+’ складывает два списка, в результате получится список из неповторяющихся слов

bool findWord(std::string&); - если слово есть, то возвращается true

int getSize() noexcept; - возвращает размер списка

void clear(); - очищает список

void popFront(); - удаляет первый элемент

void popBack(); - удаляет последний элемент

void pushBack(const std::string&); - добавляет в конец

void pushFront(const std::string&); - добавляет в начало

void insert(const std::string&, int); - вставляет в нужное место

void insertOrder(const std::string&); - вставляет в отсортированном порядке

void deleteNode(int index); - удаление заданного узла

friend std::ostream& operator<< (std::ostream&, List&); - перегруженный оператор сдвига (для вывода в поток

## Анализ алгоритма

**Анализ** **алгоритма** работы структуры данных « **RB-Tree**»

При вставке существуют 3 различных случая, в зависимости от которых определяется дальнейший алгоритм выполнения балансировки:

Случай 1. “Дядя” узла — красный;

Случай 2. “Дядя” узла черный, и узел — правый потомок;

Случай 3. “Дядя” узла черный, и узел — левый потомок.

Поскольку высота красно-черного дерева с n узлами равна O(lgn), выполнение процедуры insert требует времени O(lgn). В процедуре балансировки цикл while повторно выполняется только в случае 1, и указатель узла при этом перемещается вверх по дереву на два уровня. Таким образом, общее количество возможных выполнений тела цикла while равно O(lgn). Следовательно, общее время работы процедуры insert равно O(lgn). Интересно, что в ней никогда не выполняется больше двух поворотов, поскольку цикл while в случаях 2 и 3 завершает работу.

При удалении возникают 4 различных случая:

Случай 1. Брат узла— красный;

Случай 2. Узел — черный, оба его дочерних узла — черные;

Случай 3. Брат узла — черный, левый дочерний узел узла — красный, а правый — черный;

Случай 4. Брат узла черный, а правый дочерний узел узла красный.

Поскольку высота дерева с п узлами равна O(lgn), общая стоимость процедуры без вызова вспомогательной процедуры балансировки равна O(lgn). В процедуре балансировки в случаях 1, 3 и 4 завершение работы происходит после выполнения фиксированного числа изменений цвета и не более трех поворотов. Случай 2 — единственный, после которого возможно выполнение очередной итерации цикла while, причем указатель узла перемещается вверх по дереву не более чем O(logn) раз и никакие повороты при этом не выполняются. Таким образом, время работы процедуры балансировки составляет O(logn), причем она выполняет не более трех поворотов. Общее время работы процедуры delete, само собой разумеется, также равно O(logn).

|  |  |  |
| --- | --- | --- |
| Метод | Временная сложность | Сложность по память |
| Node(std::string&, List&); | O(N) |  |
| RBTree(); | O(N) |  |
| RBTree(RBTree&&) = delete; | O(N) |  |
| RBTree(const RBTree&) = delete; | O(N) |  |
| ~RBTree(); | O(N) |  |
| RBTree& operator=(const RBTree&) = delete; | O(N) |  |
| RBTree& operator=(RBTree&&) = delete; | O(N) |  |
| void insertKey(std::string&, List&); | O(N) |  |
| void deleteKey(std::string&); | O(N) |  |
| void searchKey(std::string&); | O(N) |  |
| void print(std::ostream&); | O(N) |  |
| RBTree(); | O(N) |  |
| RBTree(RBTree&&) = delete; | O(N) |  |
| RBTree(const RBTree&) = delete; | O(N) |  |
| ~RBTree(); | O(N) |  |
| Node\* insertNode(Node\* root, Node\* node) | O(N) |  |
| void deleteSubtree(Node\* node); | O(N) |  |
| Node\* findNode(std::string& key); | O(N) |  |
| void printNode(std::ostream& out, Node\* root); | O(N) |  |
| void fixInsert(Node\*); | O(N) |  |
| void rotateLeft(Node\*); | O(N) |  |
| void rotateRight(Node\*); | O(N) |  |
| Node\* findUncle(Node\* node); | O(N) |  |
| void deleteNode(Node\* node); | O(N) |  |
| void fixDelete(Node\*); | O(N) |  |
| void swapValues(Node\* node1, Node\* node2); | O(N) |  |
| Node\* findSuccessor(Node\* node); | O(N) |  |
| Node\* findSibling(Node\* node); | O(N) |  |
| Node\* findReplacement(Node\* node); | O(N) |  |

**Анализ** **алгоритма** работы структуры данных «**List**»

Временная сложность списка, в котором хранятся переводы не так велика: так добавление в конец или в начало и удаление из начала или конца имеют временную сложность таких операций О(1) и O(N). А значит время работы данных методов выполняется за константное и за линейное время соответственно. Метод InsertOrder выполняется за квадратичное время только в самом «худшем случае» ( при добавлении в отсортированный лист в конец), однако в среднем, время работы данного метода также происходит за линейное время».

|  |  |  |
| --- | --- | --- |
| Метод | Временная сложность | Сложность по память |
| List() | O(1) |  |
| ~List() | O(N) |  |
| std::string& operator[](const int) | O(N) |  |
| int getSize() | O(1) |  |
| void clear() | O(N) |  |
| void popFront() | O(1) |  |
| void popBack() | O(N) |  |
| void pushFront(const std::string&) | O(1) |  |
| void pushBack(const std::string&) | O(N) |  |
| void insert(const std::string&, int) | O(N) |  |
| void insertOrder(const std::string&) | О |  |
| void deleteNode(int) | O(N) |  |
| friend std::ostream& operator<< (std::ostream&, List&) | O(N) |  |
| Node(std::string value, Node\* next) | O(1) |  |
| List& operator+ (List&) | О |  |
| bool findWord(std::string&) | О |  |
| List::List(List&&) noexcept | O(1) |  |

## Описание спецификации программы (детальные требования)

1. Требования

* Переводы слова не могут повторяться.
* Слова заданы корректно.
* Слово вводится в английской раскладке, переводы – в русской. Допускается наличие дефисов, пробелов и только букв нижнего регистра.
* Текстовый файл для первоначального ввода данных в словарь должен существовать
* Текстовый файл для первоначального ввода данных в словарь должен быть не пустым.
* Английское слово для поиска и удаления вводится корректно
* Номер команды выполнения задания должен быть целым числом

1. Спецификация
2. Переводы слова не могут повторяться. В случае, если такой перевод уже есть в словаре, то к уже существующим переводам добавляются неповторяющиеся переводы. Вводится сообщение: “*Слово уже есть в словаре, дополнительный перевод добавлен!*”.
3. Слова заданы корректно.
   1. Ключ вводится в английской раскладке. Если ключ содержит нежелательные символы (кроме дефиса и пробела), включая буквы другой раскладки, сообщение " В слове "СЛОВО" грамматическая ошибка (запрещенный символ), слово не добавлено!"
   2. Информационная часть вводится в русской раскладке. Если информация содержит нежелательные символы (кроме дефиса и пробела), включая буквы другой раскладки, сообщение " В слове "СЛОВО," грамматическая ошибка (запрещенный символ), перевод не добавлен!"
4. Текстовый файл для первоначального ввода данных в словарь должен существовать. В случае, если файл не открыт, сообщение: *“Ошибка! Файла не существует”.*
5. Текстовый файл для первоначального ввода данных не пуст. В случае, если файл пуст, сообщение: *“Ошибка! Файл пуст.”*
6. Ключ для поиска перевода или удаления вводится пользователем с консоли. Если слово есть в словаре, то оно успешно удалиться и будет сообщено “*Слово успешно удалено*!”,( или для поиска – просто выводятся переводы), иначе *“Ошибка! В словаре нет такого слова!”.*
7. Номер команды, вводимый пользователем должен быть целым числом из набора [1, 2, 3, 4, 5, 6]. В случае если введено другое значение – программа выводит сообщение *“Некорректная команда!”* и программа завершается.

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

Работа программы начинается с выбора пользователем номера команды, которую он хочет выполнить. Ввод происходит в консоль. Предлагается выбрать:

«Выберите команду:

1. Прочитать из файла

2. Вывести словарь в файл

3. Добавить слово в словарь

4. Удалить слово из словаря

5. Найти перевод слова

6. Закончить программу»

Для 1 команды. – Прочитать из файла

Пользователь вводит имя файла в консоль. Ввод данных в словарь осуществляется из этого файла. Формат входных данных:

*«’английское слово’ ‘–‘ ‘перевод1, перевод2, перевод3, …’»*

или допустимо:

*«’английская фраза’ ’ –‘ ‘перевод1, перевод2, перевод3, …’»*

Такой принцип ввода данных позволяет вводить сразу несколько переводов для одного слова, что является очень удобно, так как часто слово имеют не один перевод, а несколько.

Для 2 команды. - Вывести словарь в файл

Вывод происходит в файл: «output.txt». Дополнительных действий от пользователя не требуется.

Для 3 команды. - Добавить слово в словарь

Ввод данных осуществляется из консоли. Формат входных данных имеет такую же специфику, как и для команды 1.

При добавлении программа выводит сообщение: "Слово «слово» успешно добавлено в словарь”

Для 4 команды. - Удалить слово из словаря

Ввод слова осуществляется из консоли. Формат входных данных:

*«’английское слово’» или «’английская фраза’».*

При удалении программа выводит сообщение:

"Слово «слово» успешно удалено из словаря”

или

"Слово «слово» невозможно удалить из словаря, так как такового нет в нет. ”

Для 5 команды. - Найти перевод слова

Ввод слова осуществляется из консоли. Формат входных данных:

*«’английское слово’» или «’английская фраза’».*

При нахождении слова программа выводит список переводов:

*«’английское слово’» или «’английская фраза’» - «перевод1, перевод2,…»*

Если слово в словаре отсутствует, то программа об этом сообщает.

Для 6 команды. - Закончить программу

Программа заканчивает свою работу. Дополнительных действий от пользователя не требуется.

# Заключение.

В результате выполнения задания были получены знания о красно-черном дереве, о его особенностях, как о разновидности AVL дерева, и преимуществах перед бинарным деревом поиска. Была оценена алгоритмическая сложность изученного алгоритма, которая показывает особенности красно-черного дерева. В результате, мной был разработан собственный алгоритм красно-черного дерева для хранения английских слов и списка переводов, отсортированных по алфавиту, с использованием собственных структур данных: строк типа String, односвязного списка и иерархии исключений.

# Список использованных источников

# Приложение 1. Текст программы

# Приложение 2. Протоколы отладки