САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. Петра Великого
Институт Компьютерных Наук и Кибербезопасности
02.03.01 - Математика и Компьютерные Науки
Высшая школа технологий искусственного интеллекта

Отчет по лабораторной работе №6 по дисциплине «Теория Графов»

Словарь на основе красно-черного дерева. Словарь на основе хэш-таблицы.

Студент гр. 5130201/20002

Салимли А.

_____

Преподаватель: Востров А.В.

_____

«___»_____ 20__г.

Санкт-Петербург

Весна 2024.

Содержание

Введение

В отчете содержится описание лабораторной работы по дисциплине «Теория графов». Лабораторная работа включает реализацию словаря на основе красно-черного дерева, а также реализацию словаря на основе хэш-таблицы. Работа была выполнена в среде разработки Xcode на языке программирования C++.

1 Постановка задачи

Требуется:

- Реализовать словарь на основе красно-черного дерева и хэш-таблицы без использования контейнеров и шаблонов из библиотеки STL
- Реализовать методы добавления, поиска и удаления элементов из словарей
- Реализовать функцию заполнения словаря из текстового файла
- Реализовать метод полной очистки словаря
- Реализовать хэш-функцию, генерирующую id для слова, добавляемого в словарь-таблицу

2 Математическое описание

2.1 Красно-черные деревья

2.1.1 Определение красно-чёрного дерева

Красно-черные деревья - это самобалансирующееся дерево поиска. Гарантирует логарифмический рост высоты дерева в зависимости от количества узлов. Бинарное дерево, баланс которого достигается за счет поддержания раскраски вершин в два цвета. Правила окрашивания: корень окрашен в черный цвет; каждый узел покрашен либо в черный, либо в красный цвет; листьями объявляются NIL-узлы. Считается, что листья покрашены в чёрный цвет; если узел красный, то оба его поломка черные. Для реализации этого вида сбалансированных деревьев, нужно в каждой вершине хранить дополнительно 1 бит информации (цвет)

2.1.2 Вставка

- Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня до тех пор, пока указатель на следующего потомка не станет NIL (т.е. этот потомок — лист).
- Вставляем вместо него новый элемент с NIL-потомками и красным цветом.
- Проверяем балансировку.

При вставке нового узла в красно-чёрное дерево используется перекраска узлов, балансировка дерева и поворот.

Сложность алгоритма O(logN).

2.1.3 Балансировка при добавлении узла

При добавлении в словарь новому узлу присваивается красный цвет.

Происходит проверка соседних узлов. Возможны 5 случаев:

1. Если текущий узел - корень дерева, то узер добавляемый узел перекрашивается в черный цвет, чтобы соблюдать условия красно-черныых деревьев.

2. Если «родитель» текущего узла черный, тогда свойства не нарушаются. Узел можно не перекрашивать.
3. Если «родитель» и «дядя» красные, в таком случае перекрасим «родителя» и «дядю» в чёрный цвет, а «дедушку» — в красный. При этом число чёрных вер- шин на любом пути от корня к листьям остаётся прежним. Теперь у текущего красного узла «родитель» черного цвета. Нарушение свойств красно-черного дерева возможно лишь в одном месте: вершина «дедушка» может иметь красного «родителя». Чтобы этого не произошло рекурсивно выполняется процедура первого случая. В остальных случаях начинает применяться поворот дерева вправо или влево.
4. Если «дядя» — чёрная вершина, «родитель» - красная и текущий узел — правый потомок, то добавленный узел является левым потомком красной вершины, которая, в свою очередь, является левым потомком своего родителя, правым потомком которой является «дядя». В этом случае достаточно произвести правое вращение и перекрасить две вершины. Процесс перекраски окончится, так как вершина родитель будет после этого чёрной.
5. Если «дядя» — чёрная вершина, «родитель» - красная, текущий узел — левый потомок, то добавленный узел является правым потомком красной верши- ны, которая , в свою очередь, является левым потомком своего родителя, правым потомком которой является «дядя». В этом случае производится левое вращение, которое сводит это случай к случаю 2, когда добавляемый узел является потом- ком своего родителя. После вращения глубина, измеренная в чёрных узлах от корня к листьям, остаётся прежней.

2.1.4 Удаление

В процессе удаления вершины могут возникнуть 3 случая в зависимости от количества её потомков:

1. Если у вершины нет потомков, то изменяем указатель на неё у родителя на NIL.

2. Если у неё только один потомок, то делаем у родителя ссылку на него вместо этой вершины.
3. Если имеются оба потомка, находим вершину со следующим значением клю- ча. У такой вершины нет левого потомка. Удаляем уже эту вершину, описан- ным во 2 пункте способом, скопировав её ключ в изначальную вершину. Для балансировки дерева также используются 2 действия: • Перекраска узлов. • Вращение (вправо, влево). Сложность алгоритма O(logN).

2.1.5 Балансировка при удалении узла

При удалении красной вершины свойства дерева не нарушаются. Балансировка потребуется только при удалении чёрной.

1. Если «брат» этого потомка красный, то делаем вращение вокруг ребра между «отцом» и «братом», тогда «брат» становится родителем «отца». Красим его в чёрный, а «отца» - в красный цвет.
2. Если «брат» текущей вершины был чёрным, то требуется рассмотреть еще 3 случая:

- Оба ребёрка у «брата» чёрные. Красим «брата» в красный цвет и рассматриваем далее «отца» вершины.
- Если у «брата» правый «ребёнок» черный, а левый красный, то перекрашиваем брата и его левого «сына» и делаем вращение.
- Если у брата правый «ребёнок» красный, то перекрашиваем «брата» в цвет отца, его «ребёнка» и «отца» - в чёрный, делаем вращение и выходим из алгоритма.

2.1.6 Поиск

Поиск в красно-чёрном дереве происходит как и в бинарном дереве. Алгоритм стартует от корня дерево, на каждый итерации происходит проверка, соответсвует ли ключ рассматриваемого узла ключу, который мы ищем.

- Если да, то возвращается узел в качестве ответа.

- Если нет, то происходит сравнение текущего значения ключа и искомого. В зависимости от того больше или меньше переходим в правое или левое поддерево.Алгоритм повторяет вышеперечисленное до тех пор, пока не дойдет до листа NIL.
- Если ответ не найден, возвращаем NULL.

Сложность алгоритма O(logN).

2.1.7 Пример словаря на основе красно-черного дерева

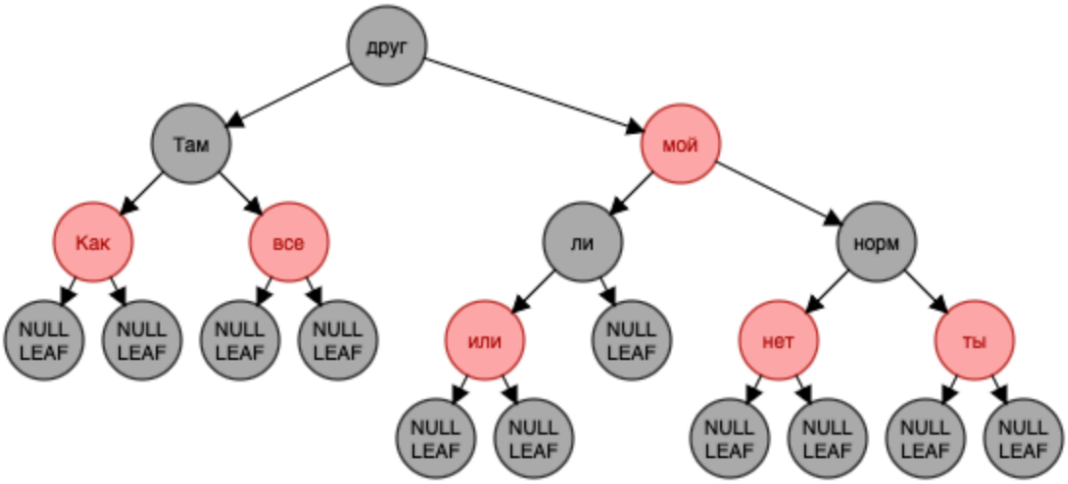

Рис.1 Словарь на основе к-ч дерева

2.2 Хэш-таблицы

2.2.1 Определение хэш-таблицы

Хэш-таблица - это структура данных, похожая на ассоциативный массив, т.е хранит пары ключ-значение, где в качестве ключа выступает любой объект, для которого можно вычислить хеш-код. Интерфейс хеш-таблицы предоставляет нам следующие операции: добавление новой пары ключ-значение, поиск значения по ключу, удаление пары ключ-значение по ключу.

2.2.2 Хэш-функция

Хэш-функция f находит остаток от деления суммы кодов всех элементов строки и размера корзины. Результат хэш-функции является номером ячейки объекта в корзине T[0,..,m-1]. Разрешение коллизий хэш функции применяется путем, двойного хэширования методом FNV-1a, в нашем случае используется версия для 32 битного типа integer (целочисленное) - uint32_t далее с помощью XOR статически преобразовывает тип в uint8_t.

$$f : T \to \{0, 1, .., m-1\}$$

$$f(x) = (\sum_{i=0}^{k} n_i) mod\ m$$

Функция FNV:

$h = x_{n+1}$,
$x_{i+1} = x_i p \oplus d_i \pmod{2^{32}}$,
$x_0 = 2166136261$,
$p = 16777619$ — простое число,
d_i — входная последовательность двоичных слов.

Модифицированная функция FNV:

$h = x_{n+1}$,
$x_{i+1} = (x_i \oplus d_i)\, p \pmod{2^{32}}$.

,где m - кол.во элементов в корзине, к - длина слова, n_i - код символа в слове.

2.2.3 Функции хэш-таблицы

Добавление элемента - Когда приходит новый объект, вычисляется хэш-код ключа, с помощью функции хэширования, ему присваивается индекс. Функция хэширования может быть реализована любым способом, единственное условие корректности хэш функции - её возвращаемое значение должно не превосходить размер массива(корзины).

Удаление элемента - Находится id элемента, который требуется удалить, если это единственный элемент с найденным id, он удаляется из корзины, в ином случае, он удаляется из цепочки по найденному id.

Поиск элемента - Находится id элемента, который требуется найти, если это единственный элемент с найденным id, элемент найден, в ином случае, происходит поиск этого элемента в цепочки по найденому id.
Сложность всех операций составляет O(1).

2.2.4 Разрешение коллизий

При добавлении нового элемента, хэш-функция может выдать такой идентификатор, по которому уже присутствует ключ. В данной реализации эта проблема решается методом двойного хэширование и методом цепочек.
Каждая ячейка корзины является указателем на цепочку пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента.
Операции поиска или удаления элемента требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом.
Для добавления элемента нужно добавить элемент в конец цепочки.
При предположении, что каждый элемент может попасть в любую позицию корзины с равной вероятностью и независимо от того, куда попал любой другой элемент, среднее время работы операции поиска элемента составляет θ(1+a), где a - коэффициент заполнения корзины.

2.2.5 Пример словаря на основе хэш-таблицы

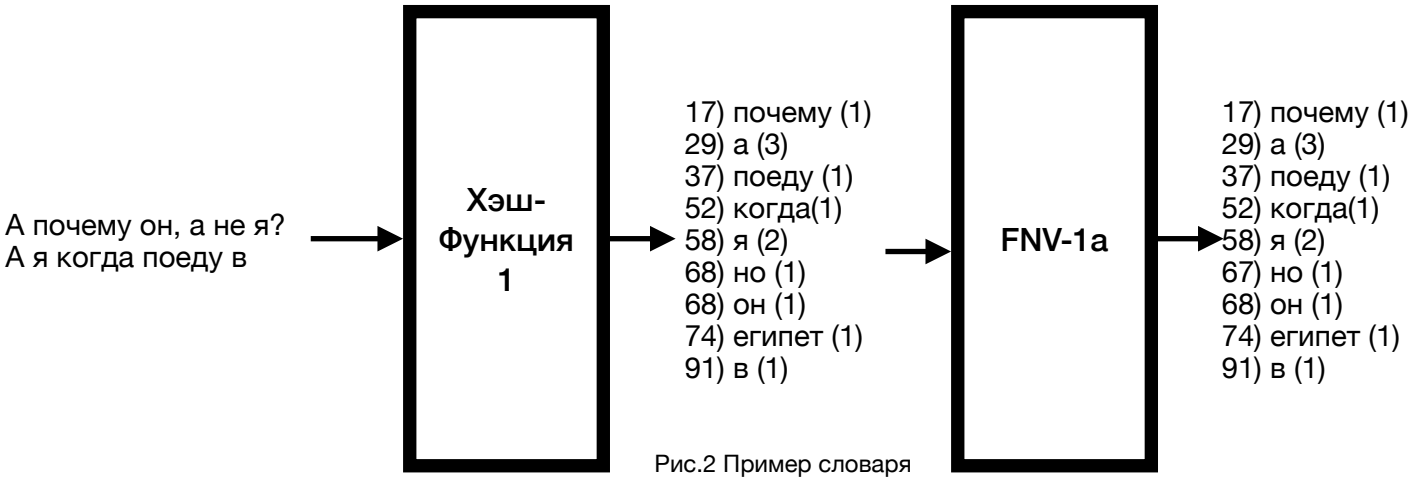

Рис.2 Пример словаря на основе хэш-таблицы

3 Особености реализации

3.1 Красно-черное дерево

Дерево реализовано и хранится в памяти с помощью структуры treeNode и класса TreeRedBlack. Структура Node содержит информацию об одном узле: его ключ, индекса, цвет, указатель на правого и левого потомка, указатель на родите- ля. TreeRedBlack - непосредственно чёрно-красное дерево, TreeRedBlack содержит одно поле - корень дерева.

```
struct

treeNode {
string key;
int index = 0;
treeColor color;
treeNode* left;
treeNode* right;
treeNode* parent;
treeNode(string k, treeColor c, treeNode* p, treeNode* l, treeNode* r) :
key(k), color(c), parent(p), left(l), right(r) { };
};
class

TreeRedBlack {
treeNode* root;
void RotateLeft(treeNode*& root, treeNode* x);
void RotateRight(treeNode*& root, treeNode* y);
void DeleteTree(treeNode*& node);
treeNode* Find(treeNode* node, string key) const;
void PrintTree(treeNode* node)const;
public:
TreeRedBlack();
~TreeRedBlack();
void insert(string key);
void insertFromFile(string file);
void DeleteElement(string key);
treeNode* Find(string key);
void PrintTree();
void Clear();
};
```

---

insert() - метод добавления элемента в словарь-дерево. Принимает строку, ничего не возвращает. Если дерево ещё не содержит элементов, то добавленный элемент перекрашивается в чёрный цвет и становится корнем дерева. Если элемент с такой строкой уже есть в словаре, индекс этого элемента инкрементируется. Если дерево не пустое, то находится потенциальный родитель для добавленного элемента, далее, исходя из его значения, помещаем принятое значение слева или справа. С помощью дополнительных функций поворота и перекраски узлов, сохраняются свойства красно-чёрных деревьев.

```
void TreeRedBlack::insert(string word) {
treeNode* word = this->Find(word);
if (word) {
word->index++;
```

```
return;
}
treeNode* z = new treeNode(word, Red, NULL, NULL, NULL);
treeNode* x = root;
treeNode* y = NULL;
while (x != NULL) {
y = x;
if (z->word > x->word)
x = x->right;
else
x = x->left;
}
z->parent = y;
if (y != NULL) {
if (z->word > y->word)
y->right = z;
else
y->left = z;
}
else
root = z;
z->color = Red;
treeNode* parent;
parent = z->parent;
while (z != root && parent->color == Red) {
treeNode* gparent = parent->parent;
if (gparent->left == parent) {
treeNode* uncle = gparent->right;
if (uncle != NULL && uncle->color == Red) {
parent->color = Black;
uncle->color = Black;
gparent->color = Red;
z = gparent;
parent = z->parent;
}
else {
if (parent->right == z) {
RotateLeft(root, parent);
swap(z, parent);
}
RotateRight(root, gparent);
gparent->color = Red;
parent->color = Black;
break;
}
}
else {
treeNode* uncle = gparent->left;
if (uncle != NULL && uncle->color == Red) {
gparent->color = Red;
parent->color = Black;
uncle->color = Black;
z = gparent;
parent = z->parent;
}
else {
if (parent->left == z) {
RotateRight(root, parent);
swap(parent, z);
}
RotateLeft(root, gparent);
parent->color = Black;
gparent->color = Red;
break;
}
}
}
root->color = Black;
};
```

insertFromFile() - метод типа void заполняет словарь из файла. Принемает строку-путь файла. Посимвольно считывает содержимое файла в вектор text, заменяя символы новой строки ('\n') на пробелы (' ').
Выполняет действия по чтению символов из файла и вставке строк методом insert в красно-черное дерево. Закрывает файл.

```
void TreeRedBlack::insertFromFile(string name) {
vector<char> text;
char symb;
ifstream myfile(name);
if (myfile.is_open()) {
while (myfile.get(symb)) {
if (symb !='\n')
text.push_back(symb);
else
text.push_back(' ');
}
myfile.close();
}
string res;
for (int i = 0; i < text.size(); i++) {
if ((text[i] <=-1 && text[i] >= -64) || text[i] ==-72 || text[i]==-88)
res += text[i];
else {
if (res.size()) {
this->insert(res);
res.clear();
}
}
if (i == text.size() - 1)
this->insert(res);}
}
```

Find() - метод поиска элемента в словаре. Имеет две перегрузки.
Первая: принимает в качестве параметра строку, возвращает найденный узел. Эта функция вызывается в main. В ней вызывается вторая перегрузка Find().
Вторая: это рекурсивная функция. Принимает в качестве параметра указатель на просматриваемый узел и строку-ключ.

```
treeNode* TreeRedBlack::Find(string key) {
return Find(root, key);
}
treeNode* TreeRedBlack::Find(treeNode* node, string key) const {
if (node == NULL || node->key == key)
return node;
else
if (key > node->key)
return Find(node->right, key);
else
return Find(node->left, key);
}
```

DeleteElement() - метод удаления элемента из словаря. Принимает строку, ничего не возвращает. Удаление включает в себя перестановки узлов, поворот дерева. После удаления сохраняется свойство красно-черных деревьев.

```
void TreeRedBlack::DeleteElement(string key) {
```

```
treeNode* delEL = Find(root, key);
if (delEL == root && delEL->left == NULL && delEL->right == NULL) {
root = NULL;
return;
}
if (delEL != NULL) {
treeNode* child, * parent;
treeColor color;
if (delEL->left != NULL && delEL->right != NULL) {
treeNode* replace = delEL;
replace = delEL->right;
while (replace->left != NULL)
replace = replace->left;
if (delEL->parent != NULL)
{
if (delEL->parent->left == delEL)
delEL->parent->left = replace;
else
delEL->parent->right = replace;
}
else
root = replace;
child = replace->right;
parent = replace->parent;
color = replace->color;
if (parent == delEL)
parent = replace;
else {
if (child != NULL)
child->parent = parent;
parent->left = child;
replace->right = delEL->right;
delEL->right->parent = replace;
}
replace->parent = delEL->parent;
replace->color = delEL->color;
replace->left = delEL->left;
delEL->left->parent = replace;
if (color == Black) {
treeNode* othernode;
while ((!child) || child->color == Black && child != TreeRedBlack::root) {
if (parent->left == child) {
othernode = parent->right;
if (othernode->color == Red) {
othernode->color = Black;
parent->color = Red;
RotateLeft(root, parent);
othernode = parent->right;
}
else {
if (!(othernode->right) || othernode->right->color == Black) {
othernode->left->color = Black;
othernode->color = Red;
RotateRight(root, othernode);
othernode = parent->right;
}
othernode->color = parent->color;
parent->color = Black;
othernode->right->color = Black;
RotateLeft(root, parent);
child = root;
break;
}
}
else {
othernode = parent->left;
if (othernode->color == Red) {
othernode->color = Black;
parent->color = Red;
RotateRight(root, parent);
othernode = parent->left;
}
```

```
if ((!othernode->left || othernode->left->color == Black) &&
(!othernode->right || othernode->right->color == Black)) {
othernode->color = Red;
child = parent;
parent = child->parent;
}
else {
if (!(othernode->left) || othernode->left->color == Black) {
othernode->right->color = Black;
othernode->color = Red;
RotateLeft(root, othernode);
othernode = parent->left;
}
othernode->color = parent->color;
parent->color = Black;
othernode->left->color = Black;
RotateRight(root, parent);
child = root;
break;
}
}
}
if (child)
child->color = Black;
}
delete delEL;
return;
}
if (delEL->left != NULL)
child = delEL->left;
else
child = delEL->right;
parent = delEL->parent;
color = delEL->color;
if (child)
child->parent = parent;
if (parent) {
if (delEL == parent->left)
parent->left = child;
else
parent->right = child;
}
else
TreeRedBlack::root = child;
if (color == Black) {
treeNode* othernode;
while ((!child) || child->color == Black && child != TreeRedBlack::root) {
if (parent->left == child) {
othernode = parent->right;
if (othernode->color == Red){
othernode->color = Black;
parent->color = Red;
RotateLeft(root, parent);
othernode = parent->right;
}
else {
if (!(othernode->right) || othernode->right->color == Black){
othernode->left->color = Black;
othernode->color = Red;
RotateRight(root, othernode);
othernode = parent->right;
}
othernode->color = parent->color;
parent->color = Black;
othernode->right->color = Black;
RotateLeft(root, parent);
child = root;
break;
}
}
else{
othernode = parent->left;
```

```
if (othernode->color == Red){
othernode->color = Black;
parent->color = Red;
RotateRight(root, parent);
othernode = parent->left;
}
if ((!othernode->left || othernode->left->color == Black) &&
(!othernode->right || othernode->right->color == Black)){
othernode->color = Red;
child = parent;
parent = child->parent;
}
else{
if (!(othernode->left) || othernode->left->color == Black){
othernode->right->color = Black;
othernode->color = Red;
RotateLeft(root, othernode);
othernode = parent->left;
}
othernode->color = parent->color;
parent->color = Black;
othernode->left->color = Black;
RotateRight(root, parent);
child = root;
break;
}
}
}
if (child)
child->color = Black;
}
delete delEL;
}
}
```

Clear() - метод очистки словаря. Вызывается в main. Ничего не принимает, ничего не возвращает. В методе вызывается метод DeleteTree(), которому в качестве параметра передается корень дерева. Это рекурсивный метод, который удаляет каждый узел.

```
void TreeRedBlack::Clear() {
DeleteTree(root);
}
void TreeRedBlack::DeleteTree(treeNode*& node) {
if (node == NULL)
return;
DeleteTree(node->left);
DeleteTree(node->right);
delete node;
node = nullptr;
}
```

3.2 Хэш-таблица

Хэш-таблица реализована и хранится в памяти с помощью структуры element и класса HashTable. Структура element содержит информацию об одном элементе: его ключ, индекс, цвет, указатель на следующий элемент со схожим id. HashTable - непосредственно хэш-таблица, HashTable содержит два поля - корзина data и size - размер корзины.

```
struct element {
string data;
```

```
int index = 1;
element* ref;
element(string d = "", element* r = NULL) { data = d; ref = r;}


};
class HashTable {
int size;
public:
element* data;
~HashTable();
HashTable();
int HashFunc(string el);
void insert(string word);
void insertFromFile(string file);
element* find(string word);
void DeleteElement(string word);
void PrintTable();
};
```

HashF() - это метод, реализующий хэш-функцию для определения id элемента. Принимает строку, возвращает целочисленное значение в диапазоне от 0 до 100. Вычисление происходит суммированием значений ASCII кодов символов слова и вычислением остатка от деления этой суммы на размер корзины.

```
int HashTable::HashF(string e) {
int sum=0;
for (int i = 0; i < e.size(); i++)
sum += -1*e[i];
return (sum%size);
}
```

FNV1aHash() - это метод, для разрешений коллизий для того что бы id элементов не совпадали. Это вторая функция хэширования. Так как двойное хэширование обеспечивает правильную индексацию. Работает по принципу мат.модели приведенной ниже:

Функция FNV:

$$
\begin{aligned}
&h = x_{n+1}, \\
&x_{i+1} = x_i p \oplus d_i \pmod{2^{32}}, \\
&x_0 = 2166136261, \\
&p = 16777619 \text{ — простое число}, \\
&d_i \text{ — входная последовательность двоичных слов.}
\end{aligned}
$$

Модифицированная функция FNV:

$$
\begin{aligned}
&h = x_{n+1}, \\
&x_{i+1} = (x_i \oplus d_i)\, p \pmod{2^{32}}.
\end{aligned}
$$

Мат.модель работы второго хэша FNV-1a

```
uint32_t HashTable::FNV1aHash(const string& key){

const uint32_t prime = 0x01000193;

uint32_t hash = 0x811c9dc5;

for(char c: key) {
hash ^= static_cast<uint8_t>(c);
hash *= prime;
};
return (hash);
    }
}
```

---

insert() - метод добавления элемента в словарь-таблицу. Принимает строку, ничего не возвращает. Если элемент с такой строкой уже есть в словаре, индекс этого элемента инкрементируется. Если элементов с найденным id не существует в словаре, на место id в корзину добавляется элемент с ключом, переданным в качестве параметра. Если в словаре существуют элементы с подобным id, в конец цепочки добавляется элемент с ключом, переданным в качестве параметра

```
void HashTable::insert(string word) {
if (this->find(word) == NULL) {
int id = HashFunc(word);
element d;
d.data = word;
d.ref = NULL;
if (data[id].data == "")
data[id] = d;
else {
element* current = &data[id];
while (current->ref) {
current = current->ref;
}
current->ref = new element;
current->ref->data = word;
current->ref->ref = NULL;
}
}
else {
if (this->find(word)->data == word)
this->find(word)->index++;
else
this->find(word)->ref->index++;
}
}
```

---

insertFromFile() - метод типа void заполняет словарь из файла. Принемает строку-путь файла, открывает файл.
Посимвольно считывает содержимое файла в вектор text, заменяя символы новой строки ('\n') на пробелы (' '). Затем закрывает файл.

```
void HashTable::insertFromFile(string name) {
vector<char> text;
char symbol;
ifstream myfile(name);
if (myfile.is_open()) {
while (myfile.get(symbol)) {
if (symbol !=
'
\n
'
```

```
) {
text.push_back(symbol);
}
else {
text.push_back(
' '
);
}
}
myfile.close();
}
string res;
for (int i = 0; i < text.size(); i++) {
if ((text[i] <= -1 && text[i] >= -64) || text[i] == -72 || text[i] == -88)
res += text[i];
else {
if (res.size()) {
this->insert(res);
res.clear();
}
}
if (i == text.size() - 1)
if (res.size())
this->insert(res);
}
}
```

---

find() - метод поиска элемента в словаре. Принимает строку-ключ. Возвращает указатель на элемент, если он один в цепи. Если элементов в цепи несколько, возвращает указатель на предшествующий в цепи элемент

```
element* HashTable::find(string word) {
int id = HashFunc(word);
if (data[id].data == word) {
return &data[id];
}
else {
element* current = &data[id];
while (current->ref != nullptr) {
if (current->ref->data == word) {
return current;
}
current = current->ref;
}
}
return NULL;
}
```

DeleteElement() - метод удаления элемента из словаря. Принимает строку, ничего не возвращает. Если элементов с таким ключом несколько, их количество уменьшается на 1. Если элемент не единственный в цепочке, то он удаляется, а цепочка перестраивается

```
void HashTable::DeleteElement(string word) {
element* removing = find(word);
if (removing != NULL) {
if (removing->index > 1 && removing->data == word) {
removing->index--;
return;
}
if (removing->ref != NULL) {
```

```
if (removing->data != word) {
if (removing->ref->index > 1) {
removing->ref->index--;
return;
}
if (removing->ref->data == word) {
if (!removing->ref->ref) {
delete removing->ref;
removing->ref = NULL;
}
else {
element* del = removing->ref;
removing->ref->data = removing->ref->ref->data;
removing->ref->index = removing->ref->ref->index;
removing->ref->ref = removing->ref->ref->ref;
delete del;
}
}
}
else {
element* del = removing->ref;
removing->data = removing->ref->data;
removing->index = removing->ref->index;
removing->ref = removing->ref->ref;
delete del;
}
}
else {
removing->data = "";


removing->ref = NULL;
}
}
}
```

Clear() - метод очистки словаря. Вызывается в main. Ничего не принимает, ничего не возвращает. Вызывает метод DeleteElement() для слов, содержащихся в словаре. Предварительно уменьшает их количество до единицы. Также сначала удаляет слова, находящиеся в цепочке не на первом месте.

```
void Clear() {
bool flagEmpty = true;
for (int i = 0; i < size; i++) {
if (data[i].data != "") {
if (data[i].index > 1)
data[i].index = 1;
while (data[i].ref != NULL)
DeleteElement(data[i].ref->data);
DeleteElement(data[i].data);
flagEmpty = false;
}
}
if (flagEmpty) cout << "Hash-table if empty !" << endl;
else cout << "Cleaning successful !" << endl;
}
```

---

4 Результаты программы

4.1 К-Ч дерево

```
В меню
1) Записать элемент.
2) Запись из файла.
3) Удалить элемент из дерева.
4) Удалить К-Ч дерево.
5) Вывести К-Ч дерево.
6) Выйти.
Введите цифру меню: 1

Введите элемент: привет
Готово!
Введите цифру меню: 1

Введите элемент: как
Готово!
Введите цифру меню: 1

Введите элемент: дела
Готово!
Введите цифру меню: 5

как(Черный) – Это КОРЕНЬ
дела(Красный) – Левый как
привет(Красный) – Правый как
Введите цифру меню: 2

Готово!
Введите цифру меню: 5

не(Черный) – Это КОРЕНЬ
как(Черный) – Левый не
в(Красный) – Левый как
а(Черный) – Левый в
дела(Черный) – Правый в
египет(Красный) – Правый дела
когда(Черный) – Правый как
почему(Черный) – Правый не
он(Черный) – Левый почему
поеду(Красный) – Правый он
привет(Черный) – Правый почему
я(Красный) – Правый привет
Введите цифру меню:
```

Рис 3. Пример вставки и вставки из файла

```
Введите цифру меню: 4

Удалено дерево!
Введите цифру меню: 5

Дерево пустое(
Введите цифру меню:
```

Рис.4 Пример удаления дерева

```
Введите цифру меню: 3

Введите элемент: когда
Элемент когда удален!
Введите цифру меню: 5

он(Черный) – Это КОРЕНЬ
в(Красный) – Левый он
а(Черный) – Левый в
не(Черный) – Правый в
египет(Красный) – Левый не
почему(Черный) – Правый он
поеду(Красный) – Левый почему
я(Красный) – Правый почему
Введите цифру меню:
```

Рис.5 Пример удаления элемента

4.2 Хэш-таблица

```
В Меню
1) Записать элемент.
2) Запись из файла.
3) Удалить элемент из словаря.
4) Вывести словарь.
5) Поиск.
6) Очистить весь словарь.
7) Выйти.
Введите цифру меню: 2

Готово!
В Меню
1) Записать элемент.
2) Запись из файла.
3) Удалить элемент из словаря.
4) Вывести словарь.
5) Поиск.
6) Очистить весь словарь.
7) Выйти.
Введите цифру меню: 4

17)почему (1)
29)а (3)
37)поеду (1)
52)когда (1)
58)я (2)
61)не (1)
68)он (1)
74)египет (1)
91)в (1)
В Меню
1) Записать элемент.
2) Запись из файла.
3) Удалить элемент из словаря.
4) Вывести словарь.
5) Поиск.
6) Очистить весь словарь.
7) Выйти.
Введите цифру меню:
```

Рис.6 Пример вставки из файла в хэш-таблицу

```
Введите цифру меню: 3

Введите элемент: египет
Готово!
В Меню
1) Записать элемент.
2) Запись из файла.
3) Удалить элемент из словаря.
4) Вывести словарь.
5) Поиск.
6) Очистить весь словарь.
7) Выйти.
Введите цифру меню: 4

17)почему (1)
29)а (3)
37)поеду (1)
52)когда (1)
58)я (2)
61)не (1)
68)он (1)
91)в (1)
В Меню
1) Записать элемент.
2) Запись из файла.
3) Удалить элемент из словаря.
4) Вывести словарь.
5) Поиск.
6) Очистить весь словарь.
7) Выйти.
```

Рис. 7 Пример удаления элемента из хэш-таблицы

```
Введите цифру меню: 6

Словарь удален
В Меню
1) Записать элемент.
2) Запись из файла.
3) Удалить элемент из словаря.
4) Вывести словарь.
5) Поиск.
6) Очистить весь словарь.
7) Выйти.
Введите цифру меню: 4

Словарь пуст!(.
В Меню
1) Записать элемент.
2) Запись из файла.
3) Удалить элемент из словаря.
4) Вывести словарь.
5) Поиск.
6) Очистить весь словарь.
7) Выйти.
Введите цифру меню:
```

Рис.8 Пример очистки хэщ-словаря

Заключение

В результате работы на языке программирования C++ были реализованы: словарь на основе красно-черного дерева, словарь на основе хэш-таблицы, операции добавления элемента, поиска элемента и удаления элемента в обоих словарях. Выполнение всех функций сохраняет свойства красно-черного дерева. Также реализованы функция полной очистки словаря на основе красно-черного дерева и функции заполнения словарей из файла.

Достоинства -
- Доступ к полной информации об элементе словаря-дерева (его цвет, родитель и потомки) из-за реализации с помощью класса treeNode.
- Высокая скорость выполнения операций со словарем на основе хэш-функции. Каждый метод выполняется за O(1).
- Подсчет количества одинаковых слов с словаре на основе хэш-таблицы.

Недостатки -
- Обязательное выполнение балансировки красно-черного дерева после каждого добавления нового элемента или после удаления элемента, чтобы не нарушить свойств красно-черных деревьев. (недостаток самой структуры)
- Обязательное хранение дополнительной информации об элементе - его цвет в дереве.
- Большой объем памяти, занимаемый словарем на основе хэш-таблицы.
- Намеренное приведение букв к малому регистру и исключение символов.

Масштабирование -
- Возможность поиска однокоренных слов.
- Добавление выбора метода хэш-функции, для словаря.
- Добавление выбора вида дерева для словаря.

Список источников

1. Новиков Ф.А. Дискретная математика для программистов. 3-е издание. СПб.:Питер,2009. 384 с.
2. Востров А.В. Лекция «Теория графов. Информационные деревья» СПб., 2022. 54 с. https://tema.spbstu.ru/tgraph_lect/ (дата обращения 20.05.2024г.)