Министерство образования и науки РФ

Санкт-Петербургский Политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа искусственного интеллекта

Направление «Математика и компьютерные науки»

Отчёт по лабораторным работе 6 по теории графов

Студент: Кулыгин Е. А. гр. 3530201/00001

Руководитель: Востров Алексей Владимирович

Санкт – Петербург

2022

Содержание

Элементы оглавления не найдены.

# Введение

Красно-чёрное дерево - это одно из самобалансирующихся деревьев поиска, гаран- тирующих логарифмический рост высоты дерева от числа узлов и быстровыполняющие основные операции: добавление, удаление, поиск узла.

Хеш-таблица — это структура данных, реализующая интерфейс ассоциативного мас- сива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Цель данной лабораторной работы:

1. Построить словарь на основе красно-чёрного дерева и хэш-таблицы.
2. Реализовать функции добавления, удаления и поиска слова в словарях.
3. Добавить функцию полной очистки словая на основе красно-чёрного дерева и до- полнения словаря из текстового файла.

# 1 Математическое описание

## Красно – чёрное дерево

### 1.1.1 Определение красно – чёрного дерева

Красно-черное дерево — бинарное дерево, баланс которого достигается за счет под- держания раскраски всршин в два цвета, подчиняющийся следующим правилам:

1. Корень окрашен в черный цвет.

2. Каждый узел покрашен либо в черный, либо в красный цвет.

3. Листьями объявляются NIL-узлы. Листья покрашены в чёрный цвет.

4. Если узел красный, то оба его поломка черные.

5. На всех ветвях дерева, ведущих от его корня к листьям, число черных узлов

Для реализации этого вида сбалансированных деревьев, нужно в каждой вершине хранить дополнительно 1 бит информации (цвет).

### 1.1.2 Вставка

• Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня дo 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.

• Теперь проверяем балансировку.

При вставке нового узла в красно-чёрное дерево используется:

1. Перекраска узлов

2. Вращение

Сложность алгоритма O(logN).

### 1.1.3 Балансировка при добавлении узла

Балансировка при добавлении слова в словарь:

• Новодобавленному углу присваивается красный цвет.

• Просматриваются цвета близлежащих узлов. Далее отдельно рассматрива ются 5 случаев:

1. Если текущий узел - корень дерева.

В этом случае перекрашиваем узел в черный цвет, чтобы не нарушить свойство (1) красно-чёрных деревьев.

2. Если «родитель» текущего узла черный.

Тогда свойства не нарушаются. Узел можно не перекрашивать.

3. Если «родитель» и «дядя» красные.

Перекрасим «родителя» и «дядю» в чёрный цвет, а «дедушку» — в красный. При этом число чёрных вершин на любом пути от корня к листьям остаётся прежним. Теперь у текущего красного yзлa чёрный «родитель». 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 остав- шихся случаях начинает применяться поворот дерева вправо или влево.

4. В случае, если «дядя» — чёрная вершина, «родитель» - красная, теку- щий узел — правый потомок.

Добавленный узел (красная вершина) является левым потомком красной вершины, которая, в свою очередь, является левым потомком своего родителя, правым потомком которой является «дядя». В этом случае дocтaтoчнo произвести правое вращение и пере- красить две вершины. Процесс перекраски окончится, так как вершина родитель бyдeт после этого чёрной.

5. В случае, если «дядя» — чёрная вершина, «родитель» - красная, теку- щий узел — левый потомок.

Добавленный узел (красная вершина) является правым потомком красной вершины, 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 потомком своего родителя. После вращения глубина, измеренная в чёрных узлах от корня к листьям, остаётся прежней.

### 1.1.4 Удаление

При удалении вершины могут возникнуть 3 случая в зависимсоти от кличества 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 ключа. У та- кой вершины нет левого потомка. Удаляем уже эту вершину, описанным во 2 пункте способом, скопировав её ключ в изначальную вершину.

Для балансировки дерева также используются 2 действия:

1. Перекраска узлов.
2. Вращение (вправо, влево). Сложность алгоритма O(logN).

### 1.1.5 Балансировка при удалении узла

Проверим балансировку дерева.

Так как при удалении красной вершины свойства дерева не нарушаются, то восста- новление балансировки потребуется только при удалении чёрной.

Рассмотрим несколько вариантов балансировки после удалении узла:

•Если «брат» этого потомка красный, то делаем вращение вокруг ребра между «от- цом» и «братом», тогда «брат» становится родителем «отца». Красим его в чёрный, а «отца» - в красный цвет.

•Если «брат» текущей вершины был чёрным, то получаем 3 случая:

1. Оба ребёрка у «брата» чёрные. Красим «брата» в красный цвет и рассматриваем далее «отца» вершины.

2. Если у «брата» правый «ребёнок» черный, а левый красный, то перекрашиваем брата и его левого «сына» и делаем вращение.

3. Если у брата правый «ребёнок» красный, то перекрашиваем «брата» в цвет отца, его «ребёнка» и «отца» - в чёрный, делаем вращение и выходим из алго- ритма.

### 1.1.6 Поиск

Поиск в красно-чёрном дереве происходит как и в бинарном дереве.

Алгоритм стартует от вершины на каждый итерации происходит проверка, соответ- свует ли ключ нашего узла ключу, который мы ищем.

• Если да, то возвращается узел в качестве ответа.

• Если нет, то происходит сравнение текущего значения ключа и искомого. В зависи- мости 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).

1.2 Хэш-таблица  
  
1.2.1 Определение хэш-таблицы

Хэш-таблица представляет собой обобщение обычного массива. Ключом массива мо- жет быть 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.

### 1.2.2 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].

Формула:

m - количество элементов в корзине, k - длина слова, - код очередного символа в слове.

### 1.2.3 Функции хэш-таблицы

Добавление элемента. Когда приходит новый объект, вычисляется хэш-код ключа, с помощью функции хэширования, ему присвивается индекс. Функция хэширования мо- жет быть реализована любым способом, единственное условие корректности хэш функции

- её возвращаемое значение должно не превосходить размер массива(корзины).

Сложность алгоритма O(1).

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

Сложность алгоритма O(1).

Поиск элемента. Находится id элемента, который требуется найти, если это един- ственный элемент с найденным id, элемент найден, в ином случае, происходит поиск этого элемента в цепочки по найденемму id.

Сложность алгоритма O(1).

### 1.2.4 Решение коллизий

В данной реализации используется метод цепочек для разрешения коллизий.

Каждая ячейка корзины является указателем на цепочку пар ключ-значение, соот- ветствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента. В данной реализации коллизии определены с помощью структуры красно – чёрного дерева RBTNode. Которая уже включает в себя указатели на следующий элемент.

Операции поиска или удаления элемента требуют просмотра всех элементов соответ- ствующей 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, сред- нее время работы операции поиска элемента составляет (1 + a), где a — коэффициент заполнения корзины.

# 2 Особенности реализации

## 2.1 Красно-чёрное дерево

### Реализация дерева

Дерево реализовано и хранится в памяти с помощью структуры RbrNode и класса RbtMap. Структура RbrNode содержит информацию об одном узле: его ключ, индекса, цвет, указатель на правого и левого потомка. RbtMap - непосредственно чёрно-красное дерево, RbtMap содержит два поля - корень дерева типа RbtNode и размер дерева size.

struct RbtNode {

string\* key;

int val;

unsigned char col;

RbtNode\* heirs[2];

RbtNode(string k = string(), unsigned char c = RED, RbtNode\* left = nullptr, RbtNode\* right = nullptr, int v = 0);

~RbtNode();

};

class RbtMap {

RbtNode\* root = nullptr;

int size = 0;

int Assert(RbtNode\* node);

void PrintNode(ostream& os, RbtNode\* node, RbtNode\* parent);

RbtNode\* Search(RbtNode\* node, string key, RbtNode\*& parent);

RbtNode\* RotateSingle(RbtNode\* node, bool dir);

RbtNode\* RotateDouble(RbtNode\* node, bool dir);

RbtNode\* InsertRecur(RbtNode\* node, string key);

void ClearRecur(RbtNode\*& node);

public:

~RbtMap();

int GetSize() const;

void Print(ostream& os);

void Save(string file = RBT\_OUTPUT\_TXT);

void Load(string file = RBT\_INPUT\_TXT);

RbtNode\* Find(string key, RbtNode\*& parent);

bool Insert(string key);

bool Remove(string key);

void Clear();

};

### 2.1.2 Функция добавления элемента

Вход: Cтрока key.

Выход: Словарь с добавленной строкой key.

Алгоритм:

1. Если элемент с такой строкой уже есть в словаре, индекс этого элемента инкремен- тируется.

2. Если дерево ещё не содержит элементов, то добавленный элемент перекрашивается в чёрный цвет и становится корнем дерева.

3. Находится потенциальный родитель для добавленного элемента, далее, исходя из его значения, помещаем его слева или справа.

4. С помощью дополнительных функций поворота и перекраски узлов, сохраняются свойства красно-чёрных деревьев.  
  
В решении используется рекурсивная функция вставки InsertRecur

bool RbtMap::Insert(string key) {

transform(key.begin(), key.end(), key.begin(), tolower);

if (root == nullptr) {

root = new RbtNode(key, BLACK);

size++;

return true;

}

RbtNode \*parent = nullptr, \*found = Find(key, parent);

if (found) {

(found->val)++;

return false;

}

root = InsertRecur(root, key);

root->col = BLACK;

size++;

return true;

}

RbtNode\* RbtMap::InsertRecur(RbtNode\* node, string key) {

if (node == nullptr) {

node = new RbtNode(key);

return node;

}

bool dir = (\*(node->key)) < key;

node->heirs[dir] = InsertRecur(node->heirs[dir], key);

if (IS\_RED(node->heirs[dir])) {

if (IS\_RED(node->heirs[!dir])) {

node->col = RED;

node->heirs[LEFT]->col = node->heirs[RIGHT]->col = BLACK;

}

else {

if (IS\_RED(node->heirs[dir]->heirs[dir])) {

node = RotateSingle(node, !dir);

}

else if (IS\_RED(node->heirs[dir]->heirs[!dir])) {

node = RotateDouble(node, !dir);

}

}

}

return node;

}

Для вращения используются функции Rotate, реализующие алгоритм вращения.

### 2.1.3 Функция удаления элемента

Вход: строка key.

Выход: Словарь с удаленной строкой key. Алгоритм:

1. Проверка на существование элемента в словаре.

2. Если элемент со строкой key - корень дерева, дерево становится пустым.

3. Если у узла нет потомков, просто удаляем элемент

4. Если узел имеет один потомок, то меняем местами значение узлов и удаляем тот, который остался без потомков.

5. Если узел имеет двух потомков, тогда выбираем подходящий по значению узел и меняем их местами, второй потомок станет потомком первого.

6. Сохраняем свойства черно-красного дерева с помощью поворота и покраски узлов.

bool RbtMap::Remove(string key) {

if (root == nullptr) {

return false;

}

transform(key.begin(), key.end(), key.begin(), tolower);

RbtNode falseRoot;

RbtNode \*it = &falseRoot, \*parent = nullptr, \*gParent = nullptr, \*found = nullptr, \*tmp = nullptr;

int dir = RIGHT, tmpDir, last;

bool res = false;

it->heirs[RIGHT] = root;

while (it->heirs[dir] != nullptr) {

last = dir;

gParent = parent;

parent = it;

it = it->heirs[dir];

dir = \*(it->key) < key;

if (\*(it->key) == key) {

found = it;

}

if (!IS\_RED(it) && !IS\_RED(it->heirs[dir])) {

if (IS\_RED(it->heirs[!dir])) {

parent = parent->heirs[last] = RotateSingle(it, dir);

}

else if (!IS\_RED(it->heirs[!dir])) {

tmp = parent->heirs[!last];

if (tmp != nullptr) {

if (!IS\_RED(tmp->heirs[!last]) && !IS\_RED(it->heirs[last])) {

parent->col = BLACK;

tmp->col = RED;

it->col = RED;

}

else {

tmpDir = (gParent->heirs[RIGHT] == parent);

if (IS\_RED(tmp->heirs[last])) {

gParent->heirs[tmpDir] = RotateDouble(parent, last);

}

else if (IS\_RED(tmp->heirs[!last])) {

gParent->heirs[tmpDir] = RotateSingle(parent, last);

}

it->col = gParent->heirs[tmpDir]->col = RED;

gParent->heirs[tmpDir]->heirs[LEFT]->col = BLACK;

gParent->heirs[tmpDir]->heirs[RIGHT]->col = BLACK;

}

}

}

}

}

if (found) {

std::swap(found->key, it->key);

parent->heirs[parent->heirs[RIGHT] == it] = it->heirs[it->heirs[LEFT] == nullptr];

delete it;

size--;

res = true;

}

root = falseRoot.heirs[RIGHT];

if (root != nullptr) {

root->col = BLACK;

}

return res;

}

### 2.1.4 Функция поиска элемента

Функция реализована с помощью рекурсии.

Вход: строка key.

Выход: Найденный узел со строкой key.

Алгоритм:

1. Указываем на корень дерева.
2. Далее последовательно сравниваем значение элемента , который мы ищем, со значением элемента, на который указывает указатель.
3. Исходя из результатов сравнения перемещаемся вправо или влево по дереву, пока не дой- дём до нужного элемента.

RbtNode\* RbtMap::Search(RbtNode\* node, string key, RbtNode\*& parent) {

if ((node == nullptr) || (\*(node->key) == key)) {

return node;

}

int dir = \*(node->key) < key;

parent = node;

return Search(node->heirs[dir], key, parent);

}

### 2.1.5 Функция полной очистки словаря

Обнуляется каждый элемент словаря, пока он не станет пустым.

Вход: словарь

Выход: пустой словарь

Функции повторяет удаление list из библиотеки STL.

void RbtMap::ClearRecur(RbtNode\*& node) {

if (node == nullptr) {

return;

}

ClearRecur(node->heirs[LEFT]);

ClearRecur(node->heirs[RIGHT]);

delete node;

node = nullptr;

size--;

}

### 2.1.6 Функция заполнения словаря из файла

В словарь добавляется каждое слово, находящееся в файле.

Вход: словарь

Выход: словарь, заполненный словами из файла

void RbtMap::Load(string file) {

std::ifstream ifs(file);

if (ifs.is\_open()) {

string s;

boost::basic\_regex<char, boost::cpp\_regex\_traits<char>> expr;

expr.imbue(std::locale{"russian"});

expr = RX\_WORDS\_RU;

while (ifs >> s) {

if (boost::regex\_match(s, expr)) {

Insert(s);

}

}

ifs.close();

}

### 2.1.7 Функция вывода словаря в файл

Вход: словарь

Выход: файл с записанным в него словарем

void RbtMap::Save(string file) {

std::ofstream ofs(file);

if (ofs.is\_open()) {

Print(ofs);

ofs.close();

}

}

void RbtMap::Print(ostream& os) {

PrintNode(os, root, nullptr);

Assert(root);

}

void RbtMap::PrintNode(ostream& os, RbtNode\* node, RbtNode\* parent) {

if (node == nullptr) {

return;

}

os << setw(15) << \*(node->key) << setw(3) << (node->val)+1 << " -- ";

HANDLE console = GetStdHandle(STD\_OUTPUT\_HANDLE);

if (node->col == RED) {

SetConsoleTextAttribute(console, TEXT\_COLOR\_RED);

}

else {

SetConsoleTextAttribute(console, TEXT\_COLOR\_GREEN);

}

os << setw(7) << kColors[node->col];

SetConsoleTextAttribute(console, TEXT\_COLOR\_DEFAULT);

if (node != root) {

os << " " << setw(6) << kDirections[GET\_DIRECTION(node, parent)] << " потомок " << \*(parent->key);

}

else {

os << " корень дерева";

}

os << '\n';

PrintNode(os, node->heirs[LEFT], node);

PrintNode(os, node->heirs[RIGHT], node);

}

int RbtMap::GetSize() const {

return size;

## Хэш-таблица

### Реализация хэш-таблицы

Хэш-таблица реализовано и хранится в памяти с помощью структуры RbtNode, реализованной в работе с красно – чёрными деревьями и класса HashMap. Структура RbtNode содержит информацию об одном элементе: его ключ, индекс, цвет, указатель на следующий элемент со схожим id. HashMap - непосредственно хэш-таблица, HashMap содержит два поля - корзина data состоящая из вектора указателей на RbtMap и size - размер корзины.

class HashMap {

vector<RbtMap\*> data = vector<RbtMap\*>();

int size = 0;

public:

~HashMap();

unsigned int GetHashValue(string key);

int GetSize() const;

RbtNode\* Find(string key, RbtNode\*& parent);

void Print(ostream& os);

bool Insert(string key);

bool Remove(string key);

void Clear();

void Load(string file = HASH\_INPUT\_TXT);

void Save(string file = HASH\_OUTPUT\_TXT);

};

### Хэш-функция

Хэш-функция заключается в деления суммы кодов всех элементов на 15. В классическом варианте, деление совершается на размер корзины. Но, для того, чтобы постоянно не применять resize на корзину, был выбран именно этот вариант.

unsigned int HashMap::GetHashValue(string key) {

int sum=0;

for (auto symbol : key) {

sum += symbol;

}

return abs((sum)/15);

}

### Функция добавления элемента

Вход: Строка key.

Выход: Словарь с добавленной строкой key. Алгоритм:

1. Если элемент с такой строкой уже есть в словаре, индекс этого элемента инкремен- тируется.

2. Если элементов с найденным id не существует в словаре, на место id в корзину добавляется элемент с ключом key.

3. Если в словаре существуют элементы с подобным id, в конец цепочки, добавляется элемент с ключом key.

bool HashMap::Insert(string key) {

transform(key.begin(), key.end(), key.begin(), tolower);

RbtNode \*found = nullptr, \*parent = nullptr;

found = Find(key, parent);

if (found) {

(found->val)++;

return false;

}

unsigned int hashVal = GetHashValue(key);

if (hashVal >= data.size()) {

data.resize(hashVal + 1, nullptr);

}

if (data[hashVal] == nullptr) {

data[hashVal] = new RbtMap();

}

data[hashVal]->Insert(key);

size++;

return true;

}

### Функция удаления элемента

Вход: строка key.

Выход: Словарь с удаленной строкой key. Алгоритм:

1. Проверка на существования элемента со строкой key.

2. Если индекс удаляемого элемента больше 1, индекс удаляемого элемента декремен- тируется.

3. Если удаляемый элемент - единственный элемент в цепочке, элемент удаляется

4. Если удаляемый элемент - один из элементов цепочки, элемент удаляется из цепочки, цепочка перестраивается.

bool HashMap::Remove(string key) {

if (size != 0) {

transform(key.begin(), key.end(), key.begin(), tolower);

RbtNode\* found = nullptr, \* parent = nullptr;

found = Find(key, parent);

if (found) {

int hashVal = GetHashValue(key);

data[hashVal]->Remove(key);

if (data[hashVal]->GetSize() == 0) {

delete data[hashVal];

data[hashVal] = nullptr;

}

size--;

return true;

}

}

return false;

}

### Функция поиска элемента

Вход: строка key и соответствующий на нее указатель.

Выход: Предок элемента со строкой key.

Алгоритм:

1. Если в словаре 1 элемент с найденным id, возвращается адрес этого элемент

2. Если удаляемый элемент - элемент в цепочке, возвращается адрес предка этого эле- мента.

3. Если элемента с такой строкой не существует, возвращается значение NULL

RbtNode\* HashMap::Find(string key, RbtNode\*& parent) {

if (size != 0) {

transform(key.begin(), key.end(), key.begin(), tolower);

unsigned int hashVal = GetHashValue(key);

RbtNode\* found = nullptr;

if (hashVal < data.size()) {

if (data[hashVal] != nullptr) {

found = data[hashVal]->Find(key, parent);

}

}

return found;

}

return nullptr;

}

### Функция заполнения словаря из файла

В словарь добавляется каждое слово, находящееся в файле.

Вход: словарь

Выход: словарь, заполненный словами из файла

void HashMap::Load(string file) {

std::ifstream ifs(file);

if (ifs.is\_open()) {

string s;

boost::basic\_regex<char, boost::cpp\_regex\_traits<char>> expr;

expr.imbue(std::locale{ "russian" });

expr = RX\_WORDS\_RU;

while (ifs >> s) {

if (boost::regex\_match(s, expr)) {

Insert(s);

}

}

ifs.close();

}

}

### Функция вывода словаря в файл

void HashMap::Save(string file) {

std::ofstream ofs(file);

if (ofs.is\_open()) {

Print(ofs);

ofs.close();

}

}