МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ ФБГОУ ВО «ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра «Программная инженерия»

**Отчёт**

По лабораторной работе №5

**«Анализ методов разрешения коллизий при хешировании»**

**Выполнили:**

Студенты группы 92-ПГ

Погосян Ж.В.

**Проверили:**

Ужаринский А.Ю.

Орёл — 2021

**Код программы:**

#include <cmath>

#include <map>

#include <string>

#include <utility>

#include <vector>

#include <algorithm>

#include <fstream>

#include <iostream>

#include <chrono>

#include <random>

#define ATTEMPT 10000

using namespace std;

const int r[6] = { 257, 269, 277, 389, 359, 491 };

int xor\_devide\_hash(int sum, int M);

int xor\_multiply\_hash(int sum, int M);

void generate\_keys(int num, std::vector<std::string>& result);

static int string\_xor\_sum(std::string key);

int find\_nearby\_prime(int);

bool is\_prime(int);

enum class HashType { XOR\_DEVIDE, XOR\_MULTIPLY };

class LinearProbing {

public:

LinearProbing(int \_capacity, double \_load\_factor, HashType hash\_type)

: capacity(abs(\_capacity)),

load\_factor(fabs(\_load\_factor)),

real\_capacity(capacity / load\_factor),

size(0) {

c = real\_capacity / 5 + 1;

while (real\_capacity % c == 0) c++;

table = new std::string \* [real\_capacity];

if (hash\_type == HashType::XOR\_DEVIDE)

hash = xor\_devide\_hash;

else

hash = xor\_multiply\_hash;

}

~LinearProbing() { drop(); }

bool insert(std::string value);

std::pair<int, int> find(std::string value);

bool del(std::string value);

void drop();

private:

const double load\_factor;

const int capacity;

const int real\_capacity;

int size;

int c;

int (\*hash)(int, int);

std::string\*\* table;

};

class QudraticProbing {

public:

QudraticProbing(int min\_capacity, HashType hash\_type)

: load\_factor(0.5),

size(0),

capacity(min\_capacity),

c(1),

d(1),

real\_capacity(find\_nearby\_prime(capacity / load\_factor)) {

table = new std::string \* [real\_capacity];

if (hash\_type == HashType::XOR\_DEVIDE)

hash = xor\_devide\_hash;

else

hash = xor\_multiply\_hash;

}

~QudraticProbing() { drop(); }

bool insert(std::string value);

std::pair<int, int> find(std::string value);

bool del(std::string value);

void drop();

private:

const double load\_factor;

int capacity;

int real\_capacity;

int size;

int c;

int d;

int (\*hash)(int, int);

std::string\*\* table;

};

//хеш-значение ключа

int xor\_devide\_hash(int sum, int M) {

int hashed\_key = sum % M;

return hashed\_key;

}

int xor\_multiply\_hash(int sum, int M) {

const double A = (sqrt(5) - 1) / 2;

int hashed\_key =

static\_cast<int>(M \* ((sum \* A) - static\_cast<int>(sum \* A)));

return hashed\_key;

}

int find\_nearby\_prime(int x) {

while (!is\_prime(x)) x++;

return x;

}

bool is\_prime(int x) {

for (int i = 2; i <= x / 2; i++) {

if (x % 2 == 0) return false;

}

return true;

}

//сумма элементов сортировки

int string\_xor\_sum(string value) {

int sum = 0;

int counter = 0;

for (auto symbol : value)

sum += (static\_cast<int>(symbol)) ^ r[counter++];

return sum;

}

//генерация ключей

void generate\_keys(int num, vector<string>& result) {

result.clear();

chrono::system\_clock::time\_point seed = chrono::system\_clock::now();

default\_random\_engine engine(seed.time\_since\_epoch().count());

uniform\_int\_distribution<int> distrib(48, 122);

int random;

result.resize(num);

for (int i = 0; i < num; i++) {

for (int k = 0; k < 6; k++) {

random = distrib(engine);

if ((random >= 48 && random <= 57) ||

(random >= 65 && random <= 90) ||

(random >= 97 && random <= 122))

result.at(i).push\_back(

static\_cast<char>(random));

else

--k;

}

}

}

//вставка(линейное)

bool LinearProbing::insert(string value) {

if (size >= capacity || find(value).second != -1) return false;

int key = string\_xor\_sum(value);

int hashed\_key = hash(key, real\_capacity);

while (table[hashed\_key] && \*table[hashed\_key] != "deleted" &&

\*table[hashed\_key] != value) {

hashed\_key = (hashed\_key + c) % real\_capacity;

}

if (!table[hashed\_key]) {

table[hashed\_key] = new string(value);

size++;

}

else if (\*table[hashed\_key] == "deleted") {

\*table[hashed\_key] = value;

size++;

}

else if (\*table[hashed\_key] == value)

return false;

return true;

}

//удаление(линейное)

bool LinearProbing::del(string value) {

int hashed\_key = find(value).second;

if (hashed\_key != -1) {

(\*table[hashed\_key]) = "deleted";

return true;

}

return false;

}

//поиск(линейное)

pair<int, int> LinearProbing::find(string value) {

int key = string\_xor\_sum(value);

int hashed\_key = hash(key, real\_capacity);

int i = 0;

while (table[hashed\_key] && \*table[hashed\_key] != value) {

hashed\_key = (hashed\_key + c) % real\_capacity;

i++;

}

if (!table[hashed\_key]) return make\_pair(-1, -1);

return make\_pair(i, hashed\_key);

}

//очистка таблицы(линейное)

void LinearProbing::drop() {

for (int i = 0; i < real\_capacity; i++)

if (table[i] != nullptr) {

delete table[i];

table[i] = nullptr;

}

size = 0;

}

//вставка(линейное)

bool QudraticProbing::insert(string value) {

if (size >= capacity || find(value).second != -1) return false;

int key = string\_xor\_sum(value);

int hashed\_key = hash(key, real\_capacity);

int tmp\_key = hashed\_key;

int i = 0;

while (table[tmp\_key] && \*table[tmp\_key] != "deleted" &&

\*table[tmp\_key] != value) {

tmp\_key =

(hashed\_key + c \* i + static\_cast<int>(d \* pow(i, 2))) %

real\_capacity;

i++;

}

if (!table[tmp\_key]) {

table[tmp\_key] = new string(value);

size++;

}

else if (\*table[tmp\_key] == "deleted") {

\*table[tmp\_key] = value;

size++;

}

else if (\*table[tmp\_key] == value)

return false;

return true;

}

//удаление(линейное)

bool QudraticProbing::del(string value) {

int hashed\_key = find(value).second;

if (hashed\_key != -1) {

(\*table[hashed\_key]) = "deleted";

return true;

}

return false;

}

//очистка таблицы(линейное)

void QudraticProbing::drop() {

for (int i = 0; i < real\_capacity; i++)

if (table[i] != nullptr) {

delete table[i];

table[i] = nullptr;

}

size = 0;

}

//поиск(линейное)

pair<int, int> QudraticProbing::find(string value) {

int key = string\_xor\_sum(value);

int hashed\_key = hash(key, real\_capacity);

int tmp\_key = hashed\_key;

int i = 0;

while (table[tmp\_key] && \*table[tmp\_key] != value) {

tmp\_key =

(hashed\_key + c \* i + static\_cast<int>(d \* pow(i, 2))) %

real\_capacity;

i++;

}

if (!table[tmp\_key]) return make\_pair(-1, -1);

return make\_pair(i, tmp\_key);

}

int sum(string key) {

int sum = 0;

int counter = 0;

for (auto symbol : key) sum += static\_cast<int>(symbol) ^ r[counter++];

return sum;

}

int main() {

int table\_size;

int keys\_num;

int experiments;

vector<string> keys;

cout << "Enter table size: ";

cin >> table\_size;

cout << "Enter keys amount(mod 10): ";

cin >> keys\_num;

if (keys\_num % 10 != 0) {

cout << "Enter more keys!";

return 1;

}

LinearProbing lnProbingDev(table\_size, 0.5, HashType::XOR\_DEVIDE);

LinearProbing lnProbingMul(table\_size, 0.5, HashType::XOR\_MULTIPLY);

QudraticProbing qdProbingDev(table\_size, HashType::XOR\_DEVIDE);

QudraticProbing qdProbingMul(table\_size, HashType::XOR\_MULTIPLY);

int sum = 0;

experiments = keys\_num / 10;

cout << "Linear probing (devide): " << endl;

while (experiments) {

for (int i = 0; i < ATTEMPT; i++) {

lnProbingDev.drop();

generate\_keys(experiments \* 10, keys);

for (auto key : keys) lnProbingDev.insert(key);

for (auto key : keys)

sum += lnProbingDev.find(key).first;

}

cout << "\tFor " << experiments \* 10 << " keys average length is " << (static\_cast<double>(sum) / (ATTEMPT \* experiments \* 10)) << endl;

sum = 0;

experiments--;

}

experiments = keys\_num / 10;

cout << "Linear probing (multiply): " << endl;

while (experiments) {

for (int i = 0; i < ATTEMPT; i++) {

lnProbingMul.drop();

generate\_keys(experiments \* 10, keys);

for (auto key : keys) lnProbingMul.insert(key);

for (auto key : keys)

sum += lnProbingMul.find(key).first;

}

cout << "\tFor " << experiments \* 10

<< " keys average length is "

<< (static\_cast<double>(sum) /

(ATTEMPT \* experiments \* 10))

<< endl;

sum = 0;

experiments--;

}

experiments = keys\_num / 10;

cout << "Qudratic probing (devide): " << endl;

while (experiments) {

for (int i = 0; i < ATTEMPT; i++) {

qdProbingDev.drop();

generate\_keys(experiments \* 10, keys);

for (auto key : keys) qdProbingDev.insert(key);

for (auto key : keys)

sum += qdProbingDev.find(key).first;

}

cout << "\tFor " << experiments \* 10

<< " keys average length is "

<< (static\_cast<double>(sum) /

(ATTEMPT \* experiments \* 10))

<< endl;

sum = 0;

experiments--;

}

experiments = keys\_num / 10;

cout << "Qudratic probing (multiply): " << endl;

while (experiments) {

for (int i = 0; i < ATTEMPT; i++) {

qdProbingMul.drop();

generate\_keys(experiments \* 10, keys);

for (auto key : keys) qdProbingMul.insert(key);

for (auto key : keys)

sum += qdProbingMul.find(key).first;

}

cout << "\tFor" << experiments \* 10

<< " keys average length is "

<< (static\_cast<double>(sum) /

(ATTEMPT \* experiments \* 10))

<< endl;

sum = 0;

experiments--;

}}

**Результаты:**

Рисунок 1 - линейное опробование делением

Рисунок 2 - линейное опробование умножением

Рисунок 3 - квадратичное опробование делением

Рисунок 4 - квадратичное опробование умножением

Рисунок 5 - сравнение всех методов

**Вывод:**

Исходя из результатов, представленных на рисунках 1 - 5, становится понятно, что использование метода линейного опробования для решения коллизий позволяет значительно сократить длину поиска элемента в хеш – таблице почти в 2 раза. Однако квадратичное опробование автоматически решает проблему скучиванья, благодаря избеганию срастания цепочек из-за «нелинейного» шага.