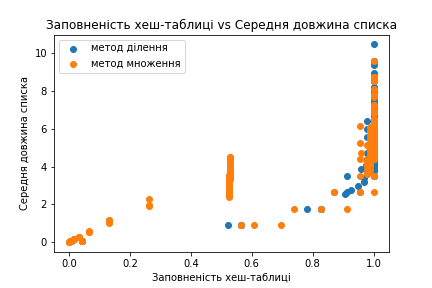
Вибір хеш-функцій

Для виконання лабораторної роботи було створено 2 способи генерації хеш-функцій: методом ділення і методом множення. Для методу ділення було використано основу 23, для метода множення - M = 23, C = 0.69. Після цього для кожного із способів було створено хеш-таблицю та проведено маніпуляці із таблицею, а саме додавання елементів, пошук, видалення і перебудова при потребі (вирішення колізій). З метою вирішення колізій при переповненні максимально допустимого розміру ланцюжка збільшується розмір хеш-таблиці вдвічі плюс 1 (щоб отримати непарне число) і змінюється функція хешування.

Аналіз методів хешування:

Для аналізу було побудовано набір даних із 1000 елементів, які додавались до хеш-таблиць. При цьому на кожному 20 кроці рахувалось відсоток заповненості хеш-таблиці та середня довжина списків. Усі результати записувались у файл. Такі дії було проведено 3 рази. Числові результати можемо побачити у файлах div.txt, mul.txt. Після цього побудовано наступний графік:



Бачимо, що при методі множення кількість колізій є вищою при однакових значеннях заповненості хеш-таблиці.

Також на екран виводиться розмір таблиць і їх заповненість, коли виникає реструктиризація. Приклад отриманих даних можна знайти у файлах div2.txt, mul2.txt.

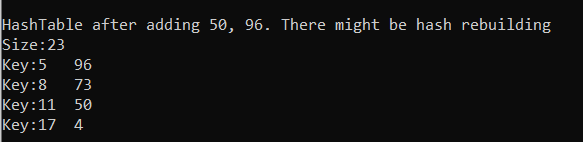
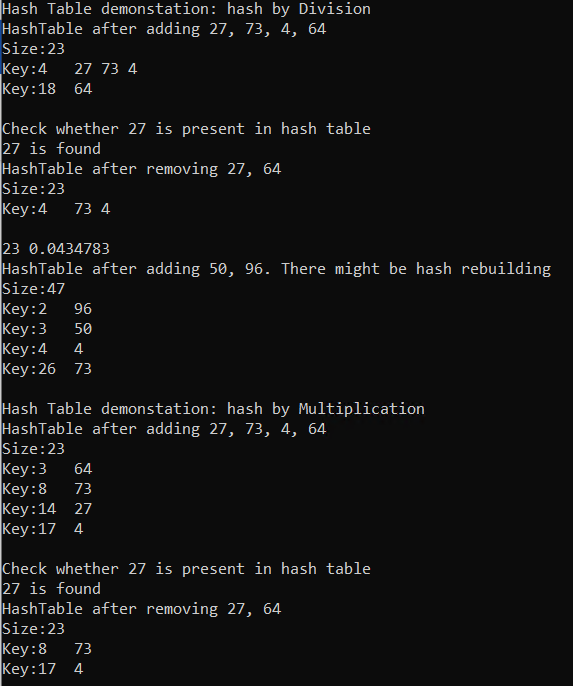
За даними можна помітити, що при методі ділення кількість колізій виникає при заповненості близькій до 1, тоді як для метода множення - вона виникає при значно меншій заповненості, що у деяких випадках навіть може призвести до переповнення розміру цілого числа (розміру хеш-таблиці).

Висновки:

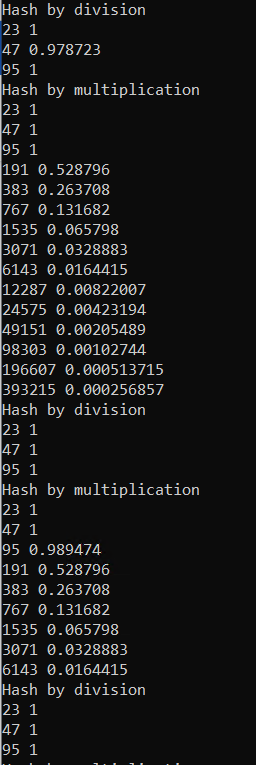
Таким чином автор вважає, що при рівномірному розподілі згенерованих чисел кращим методом хешування є хешування діленням, адже воно призводить до меншої кількості колізій, а тому відповідно є ефективішим.

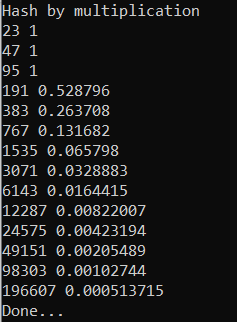
Скріншоти роботи програми:

Спершу перевіряємо роботу хеш-таблиць вручну



Виведення результатів реструктиризації хеш-таблиць





Лістинг

#include <iostream>

#include <fstream>

#include <functional>

#include <vector>

#include <utility>

using namespace std;

int hash1(int x, int M) // M should be prime

{

return x % M;

}

std::function<int(int)> hashByDivision(int M)

{

return [=](int x) {

return x % M;

};

}

int hash2(int x, double C, int M) // C in [0,1]

{

//int M = 13; double C = 0.618033;

int h = M \* fmod(x \* C, 1);

return h;

}

std::function<int(int)> hashByMultiplication(double C, int M)

{

return [=](int x) {

return int(M \* fmod(x \* C, 1));

};

}

std::function<int(int)> hashByMultiplicationMethod(int M)

{

double C = 0.69;

return [=](int x) {

return int(M \* fmod(x \* C, 1));

};

}

int getIndex(vector<int> v, int x)

{

auto it = find(v.begin(), v.end(), x);

if (it != v.end())

return (it - v.begin());

else

return -1;

}

class HashTable {

public:

int hash\_size;

vector<int>\* hash\_table; // array with pointers to list of elements

int H; // max length in list

function<int(int)> hash; // hash function

function<function<int(int)>(int)> hash\_method; // hash function generator

HashTable(int \_hash\_size, int \_H, function<int(int)> \_hash, function<function<int(int)>(int)> \_hash\_method) {

hash\_size = \_hash\_size;

H = \_H;

hash\_table = new vector<int>[hash\_size];

hash = \_hash;

hash\_method = \_hash\_method;

}

void add(int x)

{

int key = hash(x);

hash\_table[key].push\_back(x);

if (hash\_table[key].size() > H)

{

rebuild();

}

}

void rebuild()

{

cout << hash\_size << " " << keys\_with\_elements\_frequency() << endl;

int M = hash\_size \* 2 + 1;

hash = hash\_method(M); // generate new hash function

vector<int>\* h;

h = new vector<int>[M];

for (int i = 0; i < hash\_size; i++)

{

vector<int> v = hash\_table[i];

for (int j = 0; j < v.size(); j++)

{

int key = hash(v[j]);

h[key].push\_back(v[j]);

}

}

hash\_table = h;

hash\_size = M;

}

int find(int x) // return idx in relative list

{

int key = hash(x);

vector<int> v = hash\_table[key];

return getIndex(v, x);

}

void remove(int x) // remove first appearance of x

{

int key = hash(x);

vector<int>\* v = &hash\_table[key];

(\*v).erase((\*v).begin() + getIndex(\*v, x));

}

void print\_full()

{

for (int i = 0; i < hash\_size; i++)

{

vector<int> v = hash\_table[i];

cout << "Key:" << i << "\t";

for (int j = 0; j < v.size(); j++)

cout << v[j] << " ";

cout << endl;

}

}

void print()

{

cout << "Size:" << hash\_size << endl;

// Print only lists with values

for (int i = 0; i < hash\_size; i++)

{

vector<int> v = hash\_table[i];

if (v.size() > 0)

{

cout << "Key:" << i << "\t";

for (int j = 0; j < v.size(); j++)

cout << v[j] << " ";

cout << endl;

}

}

}

// functions for analysis

int keys\_with\_elements\_count() // used keys count

{

int res = 0;

for (int i = 0; i < hash\_size; i++)

{

vector<int> v = hash\_table[i];

if(v.size() > 0)

res ++;

}

return res;

}

double keys\_with\_elements\_frequency()

{

return (1.0 \* keys\_with\_elements\_count()) / hash\_size;

}

double average\_list\_length()

{

int res = 0;

for (int i = 0; i < hash\_size; i++)

{

vector<int> v = hash\_table[i];

res += v.size();

}

return (1.0 \* res) / hash\_size;

}

};

void demo(function<int(int)> h, function<function<int(int)>(int)> hash\_method)

{

//cout << "Hash Table demonstation: hash by Division" << endl;

//function<int(int)> h = hashByDivision(23);

HashTable hash = HashTable(23, 3, h, hash\_method);

// Add some elements

hash.add(27);

hash.add(73);

hash.add(4);

hash.add(64);

cout << "HashTable after adding 27, 73, 4, 64" << endl;

hash.print();

cout << endl;

cout << "Check whether 27 is present in hash table" << endl;

if (hash.find(27) != -1)

cout << "27 is found" << endl;

hash.remove(27);

hash.remove(64);

cout << "HashTable after removing 27, 64" << endl;

hash.print();

cout << endl;

hash.add(50);

hash.add(96);

cout << "HashTable after adding 50, 96. There might be hash rebuilding" << endl;

hash.print();

cout << endl;

}

vector<int> generate(int N)

{

vector<int> res;

for (int i = 0; i < N; i++)

res.push\_back(rand());

return res;

}

vector<vector<double>> demo1(vector<int> v, int N, function<int(int)> h, function<function<int(int)>(int)> hash\_method)

{

HashTable hash = HashTable(23, 15, h, hash\_method);

vector<double> x, y, size;

for (int i = 0; i < N; i++)

{

hash.add(v[i]);

if (i % 20 == 0)

{

x.push\_back(hash.keys\_with\_elements\_frequency());

y.push\_back(hash.average\_list\_length());

size.push\_back(hash.hash\_size);

}

}

vector<vector<double>> res;

res.push\_back(x); res.push\_back(y); res.push\_back(size);

return res;

}

void demo2()

{

srand(static\_cast<unsigned int>(time(0)));

function<int(int)> h1 = hashByDivision(23);

function<int(int)> h2 = hashByMultiplication(0.69, 23);

vector<double> x1, y1, x2, y2, s1, s2;

vector<vector<double>> res;

int N = 1000;

for (int i = 0; i < 3; i++)

{

vector<int> v = generate(N);

cout << "Hash by division" << endl;

res = demo1(v, N, h1, hashByDivision);

x1.insert(x1.end(), res[0].begin(), res[0].end());

y1.insert(y1.end(), res[1].begin(), res[1].end());

s1.insert(s1.end(), res[2].begin(), res[2].end());

cout << "Hash by multiplication" << endl;

res = demo1(v, N, h2, hashByMultiplicationMethod);

x2.insert(x2.end(), res[0].begin(), res[0].end());

y2.insert(y2.end(), res[1].begin(), res[1].end());

s2.insert(s2.end(), res[2].begin(), res[2].end());

}

ofstream f1("div.txt");

for(int i = 0; i < x1.size(); i++)

f1 << x1[i] << "," << y1[i] << "," << s1[i] << endl;

f1.close();

ofstream f2("mul.txt");

for (int i = 0; i < x1.size(); i++)

f2 << x2[i] << "," << y2[i] << "," << s2[i] << endl;

f2.close();

}

int main()

{

cout << "Hash Table demonstation: hash by Division" << endl;

function<int(int)> h = hashByDivision(23);

demo(h, hashByDivision);

cout << "Hash Table demonstation: hash by Multiplication" << endl;

function<int(int)> h2 = hashByMultiplication(0.69, 23);

demo(h2, hashByMultiplicationMethod);

demo2();

cout << "Done...";

return 0;

}