A2. Кубическое пробирование

Хеш-таблицы с ***открытой адресацией*** используют различные методы пробирования для разрешения коллизий, к основным из которых можно отнести:

1. ***Линейное*** пробирование, при котором последовательно проверяются ячейки хеш-таблицы с индексами $hash(key), hash(key)+1, hash(key)+2, \ldots$
2. ***Квадратичное*** пробирование $hash(key,i)=hash(key)+c_1 \cdot i+c_2 \cdot i^2$, при котором:
   - в простом варианте при $c1=c2=1$ последовательно проверяются ячейки:

     $$hash(key), hash(key)+1, \ hash(key)+2, \ hash(key)+6 \ \ldots$$

   - для хеш-таблицы размера $M=2^m$ при $c1=c2=\frac{1}{2}$ последовательно проверяются ячейки:

     $$hash(key), hash(key)+1, \ hash(key)+3, \ hash(key)+6 \ \ldots$$

Мы решили пойти дальше и рассмотреть ***кубическое*** пробирование, при котором проверка ячеек в хеш-таблице выполняются по следующему правилу: $hash(key,i)=hash(key)+c1 \cdot i+c2 \cdot i2+c3 \cdot i3$.

Оцените, будет ли кубическое пробирование выполнять распределение ключей по хеш-таблице лучше (более равномерно), чем квадратичное, с точки зрения образования кластеров и возникновения коллизий. Подкрепите свои рассуждения ***программными экспериментами*** с хеш-таблицами различных размеров, а также приложите код. Ограничений на используемые языки программирования в этом задании нет.

Предварительный анализ

Фундаментальная задача пробирования - создать равномерное распределение по всей таблице, т. е. не допустить создания слишком больших кластеров и использовать все ячейки таблицы.

функция нашего кубического пробирования будет иметь вид:

$$hashcub(key, i) = hash(key) + ic_1 + i^2c_2 + i^3c_3$$

A2.py

```
p = 16 # меняемый размер массива
m = [0 for _ in range(p)]
for c1 in range(5):
    for c2 in range(5):
        for c3 in range(5):
            m = [0 for _ in range(p)]
            for i in range(100): # менять кол-во вставок в зависимости от
```

```
p
                if((c1*i + c2*i*i + c3*i**3)%1!=0):
                    break
                m[(c1*i + c2*i*i + c3*i**3)%p] += 1
            if(all(x!=0 for x in m) and c3!=0):
                print(m,c1, c2, c3)
```

был написан такой скрипт для нахождения хороших коэффициентов. В начале для размера массива пробовал использовать простые числа, но для них закономерностей не нашлось. Потом решил попробовать посмотреть как в квадратичном плобировании степени двойки в качестве длин массивов, и это оказалось хорошей стратегией.

Появились коэффициенты которые выдавали достаточно равномерно распределенное распространение на каждой длине (среди степеней двоек)

они оказались такими

```
c1  c2  c3
----------
1   0   2
1   0   4
1   2   2
1   2   4
1   4   2
1   4   4
3   0   2
3   0   4
3   2   2
3   2   4
3   4   2
3   4   4
```

для удобства будем использовать первую тройку. Теперь наша функция имеет вид

$$hashcub(key, i) = hash(key) + i + 2i^3$$

После был написан такой код для сравнения видов пробирования

A2.cpp

```
#include <iostream>
#include <vector>

std::hash<int> hasher{};

size_t hash1(int a, size_t i){
    return hasher(a) + i/2 + i*i/2;
}
```

```
size_t hash2(int a, size_t i){
    return hasher(a) + i + 2*i*i*i;
}

inline void coutvec(std::vector<int> &v){
    for(size_t i = 0; i < v.size(); ++i){
        std::cout<<v[i]<<' ';
    }
    std::cout<<'\n';
}

int main(){
    size_t M = 32;
    std::vector<int> v1(M, 0);
    std::vector<int> v2(M, 0);

    for(size_t i = 0; i < 10; ++i){
        v1[hash1(1, i) % M] += 1;
        v2[hash2(1, i) % M] += 1;
    }
    std::cout<<"quadro:\n";
    coutvec(v1);
    std::cout<<"cubo:\n";
    coutvec(v2);

    return 0;
}
```

В нем я вставляю элемент с одним и тем же хешем i раз, как будто каждый раз место занято и он движется на следующий шаг.

Этот тест показал, что кубическое пробирование даже лучше справляется с задачей - элемент ни разу не попал в одно место. Хотя сформировалось больше маленьких кластеров в два элемента.

```
quadro:
0 2 0 0 1 1 1 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0
cubo:
1 1 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 0 1 0 0 0 0
```

Теперь попробуем их протестировать "по-взрослому": запустим симуляцию плобирования

A2-sim.cpp

```
#include <iostream>
#include <vector>

std::hash<int> hasher{};

size_t hash1(int a, size_t i){
```

```
    return hasher(a) + i/2 + i*i/2;
}

size_t hash2(int a, size_t i){
    return hasher(a) + i + 2*i*i*i;
}

inline void coutvec(std::vector<int> &v){
    for(size_t i = 0; i < v.size(); ++i){
        std::cout<<v[i]<<' ';
    }
    std::cout<<'\n';
}

void insert_to_table(int key, std::vector<int> &table, bool iscubo){
    int ind  = key;
    int i = 1;
    while(table[ind] != 0){
        if(iscubo){
            ind = hash2(key, i) % table.size();
        }
        else{
            ind = hash1(key, i) % table.size();
        }
        ++i;
    }
    table[ind] = key;
}

int main(){
    size_t M = 16;
    std::vector<int> v1(M, 0);
    std::vector<int> v2(M, 0);

    std::vector<int> keys{1,3,2,4,2,3,1,2,3,2};

    for(size_t i = 0; i < keys.size(); ++i){
        insert_to_table(keys[i], v1, false);
        insert_to_table(keys[i], v2, true);
    }
    std::cout<<"quadro:\n";
    coutvec(v1);
    std::cout<<"cubo:\n";
    coutvec(v2);

    return 0;
}
```

Который показал вот такие результаты:

```
quadro:
0 1 2 3 4 2 3 2 3 0 0 1 2 0 0 0
```

```
cubo:
0 1 2 3 4 2 3 0 2 0 1 2 3 0 0 0
```

Кубический хеш сделал большой кластер меньше, но тем самым увеличил маленький. Здесь кубический выигрывает

Попробуем увеличить массив и количество вставляемых элементов

```
quadro:
0 1 2 3 4 2 3 2 3 4 0 1 2 3 0 1 2 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0
cubo:
0 1 2 3 4 2 3 4 0 0 0 0 0 0 0 0 0 0 0 1 2 3 0 0 2 3 1 2 3 0 0 0
```

Здесь результаты такие же.

Для проверки на коллизии была изменена функция вставки - появилась проверка в конце с выводом

```
void insert_to_table(int key, std::vector<int> &table, bool iscubo){
    int ind  = key;
    int i = 1;
    while(table[ind] != 0){
        if(iscubo){
            ind = hash2(key, i) % table.size();
        }
        else{
            ind = hash1(key, i) % table.size();
        }
        ++i;
        if(table[ind] != 0 && table[ind] != key)
            std::cout<<table[ind]<<' '<<key<<' '<<iscubo<<'\n';
    }
    table[ind] = key;

}
```

которая показала вотт такие результаты

```
4 1 0
3 1 0
4 1 1
2 4 0
3 2 1
1 2 1
4 3 1
2 3 1
4 1 0
```

```
3 1 0
4 1 1
```

Получается, 5 против 6 коллизии встречаются примерно одинаково.

Итог - кубическое пробирование имеет те же качества что и квадратичная и не имеет строгих отличий. В целом проще использовать квадратичное пробирование, потому что оно требует меньше вычислений (особенно в случае $c_1 = c_2 = 1/2$)