

```

1 INSERT(key):
2     ind = hash(key) mod M
3     while table[ind] != NULL
4         if table[ind] == key
5             return
6     ind = (ind + 1) mod M
7     table[ind] = key

```

```

1 DELETE(key):
2     ind = hash(key) mod M
3     while table[ind] != NULL
4         if table[ind] == key
5             table[ind] = ERASED
6             return
7     ind = (ind + 1) mod M

```

```

1 SEARCH(key):
2     ind = hash(key) mod M
3     while table[ind] != NULL
4         if table[ind] == key
5             return TRUE
6     ind = (ind + 1) mod M
7     return FALSE

```

БОО предположим, что функция `hash` для целочисленных типов реализована так же, как `call operator` в функторе `std::hash` стандартной библиотеки языка C++, т.е. для целочисленных типов их хэш равен их значению, проинтерпретируемому как `std::size_t`.

## Первый пример

Первым примером последовательности, для которой операции будут работать *долго*, может быть такая последовательность:

```

INSERT(0)
DELETE(0)
INSERT(1)
DELETE(1)
INSERT(2)
DELETE(2)
...
INSERT(n)
DELETE(n)

```

для некоторого  $n < M$

Т.к. при удалении ключей они помечаются **ERASED**, и хэши по модулю  $M$  вставленных ключей  $0, \dots, n-1$  - это целые числа от  $0$  до  $n-1$ , то во время выполнения `INSERT(n)` и `DELETE(n)` массив `table` будет заполнен значениям **ERASED** во всех ячейках с индексами  $0, \dots, n-1$ , поэтому вставка и последующее удаление ключа  $n$  произойдёт за линейное относительно размера таблицы время.

Возможной доработкой для данного случая может быть изменение функции `INSERT` - возможность вставки кдюча в ячейку, которая равна не только **NULL**, но и **ERASED**

```

1 INSERT(key):
2     ind = hash(key) mod M
3     while table[ind] != NULL and table[ind] != ERASED
4         if table[ind] == key
5             return
6     ind = (ind + 1) mod M
7     table[ind] = key

```

Эта небольшая доработка не испортит корректность структуры данных, т.к. до доработки роль элементов, сохраняющих целостность кластера при удалении, играли только **ERASED**, а после неё на их место стали вставляться действительные ключи, которые также подходят для сохранения кластера, т.к. непосредственно являются его элементами.

## Второй пример

Вторым примером последовательности, операции над которой приведут к *долгой* работе (за линейное время относительно размера таблицы), может быть:

```
INSERT(0)
INSERT(M)
INSERT(2 · M)
INSERT(3 · M)
INSERT(4 · M)
...
INSERT(n · M)
SEARCH(n · M)
DELETE(n · M)
```


для некоторого  $n < M$

На момент вызова операции  $\text{INSERT}(n \cdot M)$ , в хэш-таблице будет  $n$  элементов, при этом, т.к. у всех вставленных ключей хэш по модулю  $M$  равен 0, то они при вставке последовательно вставлялись в массив `table` по индексам  $0, 1, \dots, n - 1$ , поэтому вставка ключа  $n \cdot M$  займёт линейное время относительно размера таблицы (мы переберём все индексы от 0 до  $n - 1$ , пока не найдём первое свободное место по индексу  $n$ )


Аналогично, для операций  $\text{SEARCH}(n \cdot M)$  и  $\text{DELETE}(n \cdot M)$  время их выполнения будет пропорционально размеру хэш-таблицы, т.е. поиск и удаление выполнятся за линейное время относительно количества элементов в таблице.


Для решения данной проблемы можно использовать квадратичное пробирование и параллельный поиск свободных значений при помощи векторных инструкций как, например, в:

<https://doc.rust-lang.org/std/collections/struct.HashMap.html>

**Struct** `std::collections::HashMap`  1.0.0 · [source](#) · [-]

```
pub struct HashMap<K, V, S = RandomState> { /* private fields */ }
```

 A **hash map** implemented with quadratic probing and SIMD lookup.

The hash table implementation is a Rust port of Google's [SwissTable](#). The original C++ version of SwissTable can be found [here](#), and this [CppCon talk](#)  gives an overview of how the algorithm works.