

# Разделы

1. Идеальное хеширование ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#perfect](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#perfect))
  - A. Цель ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#intro](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#intro))
  - B. Идея алгоритма идеального хеширования ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#idea](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#idea))
  - C. Доказательство ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#proof](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#proof))
  - D. Краткий итог ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#summary](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#summary))
2. Хеш-таблицы в Java ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#java](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#java))
3. Еще о dict в Python ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#python](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#python))
4. C++ ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#cpp](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#cpp))

5. Robin Hood hashing ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#robinhood](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#robinhood)).
6. 2-choice hashing ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#2choice](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#2choice)).
7. Домашняя работа ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#homework](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#homework)).

## Идеальное хеширование

### Цель

Если заранее известно статическое (неизменное) множество ключей, которое будет необходимо сохранять в хеш-таблицу, можно заранее подобрать хеш-функцию таким образом, чтобы избежать коллизий.

Это позволяет обеспечить производительность  $O(1)$  в худшем, а не только в среднем случае.

# Идея алгоритма идеального хеширования

Мы рассмотрим алгоритм, использующий модифицированный метод цепочек и подбор универсальной хеш-функции.

Первичная хеш-функция  $h \in \mathcal{H}$  используется для вычисления хешей и отправки записей в "корзины".

Однако в корзинах находятся не списки, а *вторичные хеш-таблицы*. У вторичной хеш-таблицы в ячейке  $j$  должны быть

- подобранная хеш-функция  $h_j$  из семейства  $\mathcal{H}$
- достаточно большой размер, чтобы избежать коллизий
- В сумме первичная таблица и все вторичные хеш-таблицы должны занимать линейное количество памяти

## Подбор вторичных хеш-функций

Первичная хеш-функция выбирается из класса  $\mathcal{H}_{pm}$ ,  $h_{ab} = ((a \cdot k + b) \bmod p) \bmod m$ .

Со вторичными все сложнее: подобрать их можно, только проверив на конкретном множестве ключей, как они хешируются в хеш-таблицу, обеспечив, при этом достаточно большой размер вторичных хеш-таблиц, чтобы сделать такой подбор возможным и достаточно быстрым.

размер вторичной таблицы  $S_j$  равен квадрату количества ключей, которые попадают в нее:  $m_j = n_j^2$

$$h(k) = ((3k + 7) \bmod 17) \bmod 8$$

0	-	/	/	9	/	/	3	/	/	6
1	-									
2	-	/	/	1	4					
3	-	7								
4	-									
5	-	2	5	/	/					
6	-	8								
7	-	0								

$$h(k) = ((k + 2) \bmod 17) \bmod 9$$

$$h(k) = ((9k + 5) \bmod 17) \bmod 4$$

$$h(k) = 1$$

$$h(k) = ((7k + 12) \bmod 17) \bmod 4$$

$$h(k) = 1$$

$$h(k) = 1$$

# Доказательство

При выборе случайной хеш-функции из класса  $\mathfrak{H}_{pm}$  вероятность коллизии не более  $1/2$

- Количество пар ключей, приводящих к коллизии:  $\binom{n_j}{2} = \frac{n_j!}{2!(n_j - 2)!}$  - количество сочетаний, или количество всех подмножеств размера 2 из множества размера  $n$ .
- Вероятность коллизий для случайно выбранной хеш-функции из семейства  $\mathfrak{H}_{pm}$  равна  $1/m$ ;  $m = n^2$ , и

$$E[x] = \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n^2 - n}{2} \cdot \frac{1}{n^2} < 1/2$$

Вероятность того, что коллизий не будет, меньше вероятности того, что они случатся. Подбором (не очень продолжительным, благодаря этому правилу) можно найти подходящую вторичную хеш-функцию для каждой хеш-таблицы.  
**Общий размер хеш-таблиц**

На первом уровне все ключи хешируются в таблицу размером  $m = n$ , где  $n$  - размер статического множества ключей.

Если в ячейку хешируется 0 или 1 ключ - ситуация тривиальная, но каким будет расход памяти при коллизиях не первом уровне?

Поскольку процесс выбора хеш-функций - случайный, будем искать матожидание размера хеш-таблицы  $E[\sum_{j=0}^{m-1} m_j] = E[\sum_{j=0}^{m-1} n_j^2]$ .

- $n^2 = n + 2\binom{n}{2}$ , тогда  
 $E[\sum_{j=0}^{m-1} n_j^2] = E[\sum_{j=0}^{m-1} n_j] + 2E[\sum_{j=0}^{m-1} \binom{n_j}{2}] = n + 2E[\sum_{j=0}^{m-1} \binom{n_j}{2}]$
- $E[\sum_{j=0}^{m-1} \binom{n_j}{2}]$  - это общее число коллизий. При универсальном хешировании вероятность коллизии  $1/m = 1/n$ , тогда

$$\binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{n} = \frac{n-1}{2},$$

поскольку  $m = n$  для внешней таблицы

- Следовательно,  $E[\sum_{j=0}^{m-1} m_j] \leq n + 2 \frac{2n-1}{2} = 2n-1 < 2n$ . В итоге, матожидание размера всех хеш-таблиц будет асимптотически  $O(n)$
- Отсюда следует, что вероятность того, что размер итоговой хеш-таблицы будет больше  $4n$ , равен  $\frac{E[\sum_{j=0}^{m-1} m_j]}{4n} < \frac{2n}{4n} = 1/2$  - и вероятность будет падать с увеличением размера хеш-таблицы.

# Краткий итог

## Для создания идеальной хеш-таблицы нужно:

- Наличие статического множества ключей размера  $n$
- Хеш-таблица "первого уровня" размера  $m = n$
- Универсального семейства хеш-функций  $\mathfrak{H}_{pm}$ , из которого выбирается хеш-функция первого уровня и хеш-функции второго уровня

## Алгоритм:

- Выбрать размер хеш-таблицы  $m$  по количеству ключей
- Выбрать из множества  $\mathfrak{H}_{pm}$  хеш-функцию первого уровня  $h_1$
- рассчитать для всех ключей значения хешей
- для каждой ячейки  $j$ , куда попало больше 1 элемента, повторять следующие действия:
  - Выбрать случайную хеш-функцию  $h_j$  из семейства  $\mathfrak{H}_{pm}$
  - Проверить, есть ли коллизии, если коллизий нет, перейти к следующей ячейке

## Свойства:

- Время на все операции  $O(1)$  в худшем случае,  $O(n)$  для последовательности из  $n$  любых операций
- Ожидаемый расход памяти  $O(n)$

в начало ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#index](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#index)).

# Хеш-таблицы в Java

Кроме `HashTable`, в Java есть еще `HashMap`, который обсуждался на первом занятии. Между ними есть разница; во-первых,

While still supported, these classes (`HashTable`) were made obsolete by the JDK1.2 collection classes, and should probably not be used in new development.

Кроме того, есть разница в реализации: (<https://stackoverflow.com/questions/40471/differences-between-hashmap-and-hashtable>). Более современной версией `HashTable` является `ConcurrentHashMap`.

В качестве примера разбирается `HashTable`.

Итак, `HashTable`: Ценный комментарий есть здесь (<http://developer.classpath.org/doc/java/util/Hashtable-source.html>):

```
55:  * This implementation of Hashtable uses a hash-bucket approach. Tha
t is:
56:  * linear probing and rehashing is avoided; instead, each hashed val
ue maps
57:  * to a simple linked-list which, in the best case, only has one nod
e.
58:  * Assuming a large enough table, low enough load factor, and / or w
ell
59:  * implemented hashCode() methods, Hashtable should provide O(1)
60:  * insertion, deletion, and searching of keys. Hashtable is O(n) in
61:  * the worst case for all of these (if all keys hash to the same buc
ket).
```

## Важные атрибуты класса:

```
257:  public Hashtable(int initialCapacity, float loadFactor)
258:  {
259:      if (initialCapacity < 0)
260:          throw new IllegalArgumentException("Illegal Capacity: "
261:                                             + initialCapacity);
262:      if (! (loadFactor > 0)) // check for NaN too
263:          throw new IllegalArgumentException("Illegal Load: " + loadFa
ctor);
264:
265:      if (initialCapacity == 0)
266:          initialCapacity = 1;
267:      buckets = (HashEntry[]) new HashEntry[initialCapacity];
268:      this.loadFactor = loadFactor;
269:      threshold = (int) (initialCapacity * loadFactor);
270:  }
```

## Хеш-функция:

```

814: private int hash(Object key)
815: {
816:     // Note: Inline Math.abs here, for less method overhead, and t
o avoid
817:     // a bootstrap dependency, since Math relies on native method
s.
818:     int hash = key.hashCode() % buckets.length;
819:     return hash < 0 ? -hash : hash;
820: }

```

### Пример .hashCode() для String:

```

1065: public int hashCode()
1066: {
1067:     if (cachedHashCode != 0)
1068:         return cachedHashCode;
1069:
1070:     // Compute the hash code using a local variable to be reentran
t.
1071:     int hashCode = 0;
1072:     int limit = count + offset;
1073:     for (int i = offset; i < limit; i++)
1074:         hashCode = hashCode * 31 + value[i];
1075:     return cachedHashCode = hashCode;
1076: }

```

### Поиск в хеш-таблице по ключу:

```

390: /**
391:  * Return the value in this Hashtable associated with the suppli
ed key,
392:  * or null if the key maps to nothing.
393:  *
394:  * @param key the key for which to fetch an associated value
395:  * @return what the key maps to, if present
396:  * @throws NullPointerException if key is null
397:  * @see #put(Object, Object)
398:  * @see #containsKey(Object)
399:  */
400: public synchronized V get(Object key)
401: {
402:     int idx = hash(key);
403:     HashEntry e = buckets[idx];
404:     while (e != null)
405:     {
406:         if (e.key.equals(key))
407:             return e.value;
408:         e = e.next;
409:     }
410:     return null;
411: }

```

### Ресайзинг и перезапись хешей:

```

874:    /**
875:     * Increases the size of the Hashtable and rehashes all keys to
new array
876:     * indices; this is called when the addition of a new value woul
d cause
877:     * size() > threshold. Note that the existing Entry objects are
reused in
878:     * the new hash table.
879:     *

880:     *
881:     * This is not specified, but the new size is twice the current
size plus
882:     * one; this number is not always prime, unfortunately. This imp
lementation
883:     * is not synchronized, as it is only invoked from synchronized
methods.
884:     */
885:     protected void rehash()
886:     {
887:         HashEntry[] oldBuckets = buckets;
888:
889:         int newcapacity = (buckets.length * 2) + 1;
890:         threshold = (int) (newcapacity * loadFactor);
891:         buckets = (HashEntry[]) new HashEntry[newcapacity];
892:
893:         for (int i = oldBuckets.length - 1; i >= 0; i--)
894:         {
895:             HashEntry e = oldBuckets[i];
896:             while (e != null)
897:             {
898:                 int idx = hash(e.key);
899:                 HashEntry dest = buckets[idx];
900:
901:                 if (dest != null)
902:                 {
903:                     HashEntry next = dest.next;
904:                     while (next != null)
905:                     {
906:                         dest = next;
907:                         next = dest.next;
908:                     }
909:                     dest.next = e;
910:                 }
911:                 else
912:                 {
913:                     buckets[idx] = e;
914:                 }
915:
916:                 HashEntry next = e.next;
917:                 e.next = null;
918:                 e = next;
919:             }
920:         }
921:     }

```

</font>

в начало ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#index](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#index))



## Еще о dict Python

`dict` - хеш-таблица общего назначения, активно используемая, в том числе, в ООП и базовых активностях языка.

Основана на открытой адресации.

Для получения значений хешей используется `Py_hash_t PyObject_Hash(PyObject *v)`.

```

Py_hash_t
PyObject_Hash(PyObject *v)
{
    PyTypeObject *tp = Py_TYPE(v);
    if (tp->tp_hash != NULL)
        return (*tp->tp_hash)(v);
    /* To keep to the general practice that inheriting
     * solely from object in C code should work without
     * an explicit call to PyType_Ready, we implicitly call
     * PyType_Ready here and then check the tp_hash slot again
     */
    if (tp->tp_dict == NULL) {
        if (PyType_Ready(tp) < 0)
            return -1;
        if (tp->tp_hash != NULL)
            return (*tp->tp_hash)(v);
    }
    /* Otherwise, the object can't be hashed */
    return PyObject_HashNotImplemented(v);
}

```

Для получения хеша используются различные функции, например, str :

```

static long
string_hash(PyStringObject *a)
{
    register Py_ssize_t len;
    register unsigned char *p;
    register long x;

    if (a->ob_shash != -1)
        return a->ob_shash;
    len = Py_SIZE(a);
    p = (unsigned char *) a->ob_sval;
    x = *p << 7;
    while (--len >= 0)
        x = (1000003*x) ^ *p++;
    x ^= Py_SIZE(a);
    if (x == -1)
        x = -2;
    a->ob_shash = x;
    return x;
}

```

Для целых чисел int :

В python нет ограничения на размеры числа int . Информация о том, сколько чисел long в памяти занимает данный int , получается при помощи Py\_SIZE() .

Это проверяется в операторе switch, если число по модулю меньше, чем long , используется тривиальное хеширование. Иначе определяется знак числа, и далее в цикле выполняется операция хеширования по каждому из значений v->ob\_digit[i] .

```

static Py_hash_t
long_hash(PyLongObject *v)
{
    Py_uhash_t x;
    Py_ssize_t i;
    int sign;

    i = Py_SIZE(v);
    switch(i) {
    case -1: return v->ob_digit[0]==1 ? -2 : -(sdigit)v->ob_digit[0];
    case 0: return 0;
    case 1: return v->ob_digit[0];
    }
    sign = 1;
    x = 0;
    if (i < 0) {
        sign = -1;
        i = -(i);
    }
    while (--i >= 0) {
        x = ((x << PyLong_SHIFT) & _PyHASH_MODULUS) |
            (x >> (_PyHASH_BITS - PyLong_SHIFT));
        x += v->ob_digit[i];
        if (x >= _PyHASH_MODULUS)
            x -= _PyHASH_MODULUS;
    }
    x = x * sign;
    if (x == (Py_uhash_t)-1)
        x = (Py_uhash_t)-2;
    return (Py_hash_t)x;
}

```

## C++

std::unordered\_map использует открытую адресацию. Реализация std (на мой взгляд) крайне непрозрачна, так что я не думаю, что это хорошая идея - разбирать ее в рамках семинара, по крайней мере, мной.

Извините :) Boost я тоже посмотрел, но там все также довольно запутано.

</font>

в начало ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#index](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#index)).

# Robin Hood Hashing

Статья отсюда (<https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/>) про реализацию хеш-таблицы.

Сама реализация: [ссылка на github](https://github.com/skarupke/flat_hash_map/blob/master/flat_hash_map.hpp) ([https://github.com/skarupke/flat\\_hash\\_map/blob/master/flat\\_hash\\_map.hpp](https://github.com/skarupke/flat_hash_map/blob/master/flat_hash_map.hpp)).

Это усовершенствование алгоритма прямой адресации. В данной хеш-таблице применяется техника Robin Hood (отбирай у богатых, отдавай бедным). Обеспечивает  $O(\ln n)$  итераций во время пробинга.

- "Бедные" - это ключи, которые находятся далеко от "своей" позиции правильного хеша.
- "Богатые" - это ключи, расположенные близко к своему хешу.

$$h(k) = k \% m$$

Хеш	Ключ	Шаги		Хеш	Ключ	Шаги
0	20	0	20	0	20 → 18	0 → 2
1	21	1	1	1	21	1
2	/	/	2	2	20	2
3	23	0		3	23	0
4	/	/		4	/	/
5	5	0	26	5	5	0
6	25	1	0	6	25	1
7	15	2	1	7	15	2
8	18	0	2	8	18 → 26	0 → 2
9	28	1		9	28	1

Плюсы подхода

- Снижается расстояние при пробинге
- Снижается *разброс* (лучше сказать, variance) количества шагов при пробинге
- Таблица работает быстрее при простом алгоритме
- Можно делать *load factor* гораздо больше - т.е. экономится память

## Минусы

- Чтобы Robin Hood хорошо работал с отсутствующими ключами, надо исхитриться дополнительно

</font>

[в начало \(https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#index\)](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#index)

## 2-choice hashing

Вместо одной хеш-функции в методе цепочек используются целых две.

При операции вставки хеш-функции указывают на разные цепочки, вставка осуществляется в наименьшую.

</font>

[в начало \(https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#index\)](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#index)

# Домашняя работа

- Для тех, кто делал хеш-таблицы с обработкой ошибок методом цепочек, рекомендуется идеальное хеширование: реализовать алгоритм
- Для тех, кто выбрал открытую адресацию, проще будет сделать Robin Hood hashing, так как у вас уже есть наработки. Но вы можете выбрать и первый вариант.
- Дополнительные задачи (можно решать любое количество; присылайте решенные ДЗ в LaTeX'e, если хотите)
  - Решите парадокс близнецов: рассчитайте, для какого количества людей вероятность совпадения дней рождения в один день составляет 0.5; 0.9.
  - Решите ту же задачу для совпадения дней рождений у трех людей. То есть какого размера должен быть коллектив, чтобы вероятность того, что три человека родились в один день, превышала 0.5; 0.9.
  - Рассчитайте ожидаемое количество шагов при линейном пробинге длиной в один шаг для таблицы с открытой адресацией при load factor'e 0.5; 0.75; 0.9.
  - Найдите ожидаемую длину цепочки в 2-choice hashing'e

</font>

в начало ([https://otus.ru/media-private/08/66/perfect\\_hashing\\_hash\\_tables\\_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#index](https://otus.ru/media-private/08/66/perfect_hashing_hash_tables_implementation-31272-0866a8.html?hash=AZsX0SGYvL9BgfSU57YqhQ&expires=1593465693#index)).