In [1]:
```
from hash_tables import *
```

Содержание

Обзор (других) структур данных для быстрого поиска информации

Список

- Список вообще не очень хорош для поиска в нем данных
- Поиск за $O(n)$, просмотр всех элементов подряд

Отсортированный список/массив

- Все еще простейшая структура данных
- Нужно поддерживать в отсортированном виде - $O(log\ n)$ на вставку, если это список
- Поиск за $O(log\ n)$

Heap

- $O(1)$ на извлечение минимального/максимального элемента (используется в основном в этих целях)
- За $O(log\ n)$ элементы добавляются и удаляются
- Искать определенный элемент по ключу неудобно (обход heap)

Binary search tree

- Поиск элемента, вставка, удаление из BST: $O(h)$
- $O(h)$ не всегда равно $O(log\ n)$, так как дерево не сбалансировано
- Есть возможность делать запросы "меньше/больше, чем..."

Red-Black tree

- Сбалансированное дерево с возможностью выполнять базовые операции за $O(log\ n)$
- Есть возможность делать запросы "меньше/больше, чем..."

И другие

...

Поиск, удаление и вставка за O(1)

- Иногда запросы формата "больше/меньше" не требуются
- Скорость $O(log\ n)$ недостаточно хороша

Таблицы с прямой адресацией

Ассоциативный массив

Ассоциативный массив - абстрактный тип данных, позволяющий хранить пары "ключ-значение"И, реаслизующий операции

- `Insert(key, value)`
- `Find(key)`
- `Remove(key)`

Абстрактный - в данном случае значит, что конкретная реализация неизвестна, и мы сейчас рассмотрим несколько вариантов.
Две пары значений с одинаковым ключом храниться не могут.

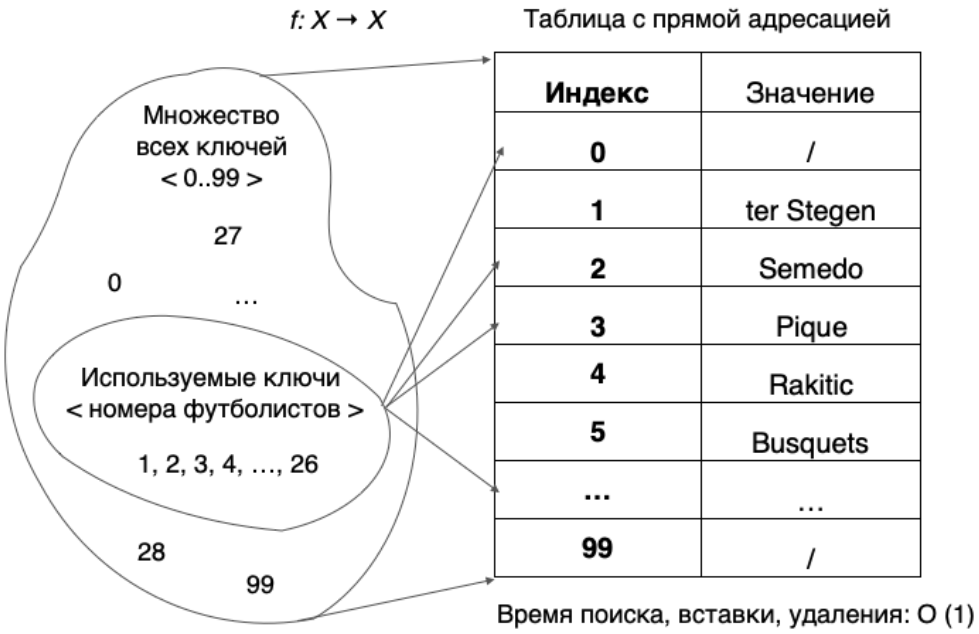
Простейший случай

Ключ однозначно указывает на ячейку памяти (например, ключ - это какое-то число).
За $O(1)$ можно обратиться к этой ячейке памяти и получить ассоциированное с ключом значение.
Сколько всего доступно памяти?

Адресация в массиве

- Размер адресуемой памяти у 32-битной архитектуры: $2^{32} = 4294967296$ (~4 Гб)
- У 64-битной: $2^{64} = 18446744073709551616$ (16 эксабайт, очень-очень много)

Представление таблицы с прямой адресацией



Плюсы

- Простота реализации
- Практически нет дополнительных расходов на чтение, удаление и запись

Проблемы?

- Перерасход памяти: многие ячейки пусты
- Ограничение по количеству возможных типов ключей

в начало (https://otus.ru/media-private/37/b6/hash_tables_intro-31272-37b61e.html?

In [2]:
```
barcelona_table = fill_team_table(barcelona)
barcelona_table.iloc[10:20]
```

Out[2]:

idx	player
10	Messi
11	Dembele
12	Rafinha
13	Cillessen
14	Malcom
15	Lenglet
16	Samper
17	
18	Alba
19	el Haddadi

In [3]:
```
# <font size="4">Какой же в нашей таблице процент заполнения?</font>
round(filling_percentage(barcelona_table), 2)
```

Out[3]:
```
0.23
```

Доступные операции

- Вставка: `DirectAddressInsert(Table, key, value)` - O(1)
- Удаление: `DirectAddressDelete(Table, key)` - O(1)
- Поиск: `DirectAddressSearch(Table, key)` - O(1)
- Псевдокод операций:

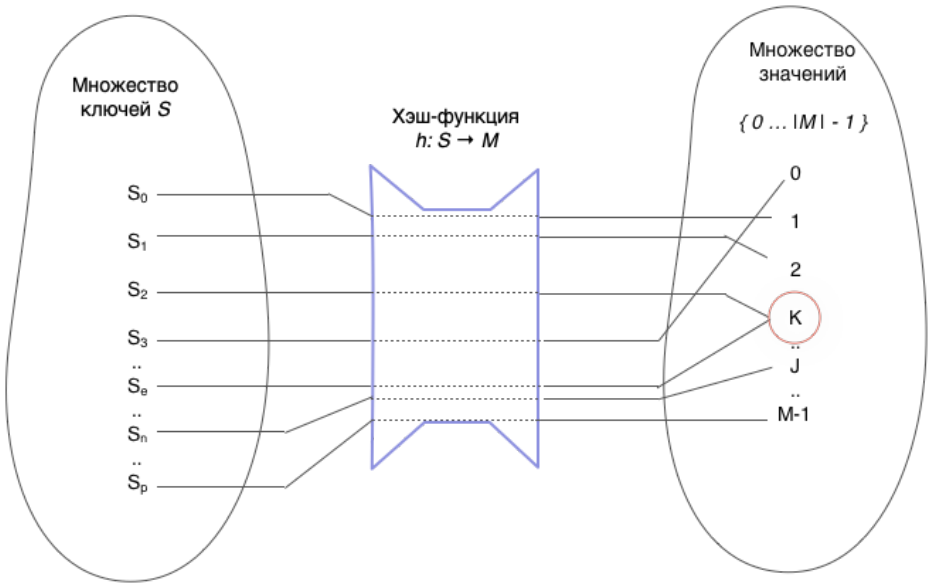
```
DirectAddressInsert(Table, key, value):
    Table[key] = value

DirectAddressDelete(Table, key):
    Table[key] = NIL

DirectAddressSearch(Table, key):
    return Table[key]
```


Хэш-функция

Общие характеристики

Хэш-функция используется для отображения множетсва ключей в множетсво хэшей. Их может

- $h: S \rightarrow M$
- $|S| > |M|$
- Отображение сюръективно



Требования

- Всегда одинаковые значения для одного ключа
- Желательно, чтобы ключ с равной вероятностью хэшировался в случайную ячейку
- Для допустимого ключа на выходе должна давать натруральное число ($+0$) в заданном диапазоне

Простейшая хэш-функция

- Хэш-функция через деление с остатком
- $hash(k) = k \mod M$, где M - размер таблицы
- Далее мы разберем другие хеш-функции, в том числе и для работы со строками и последовательностями

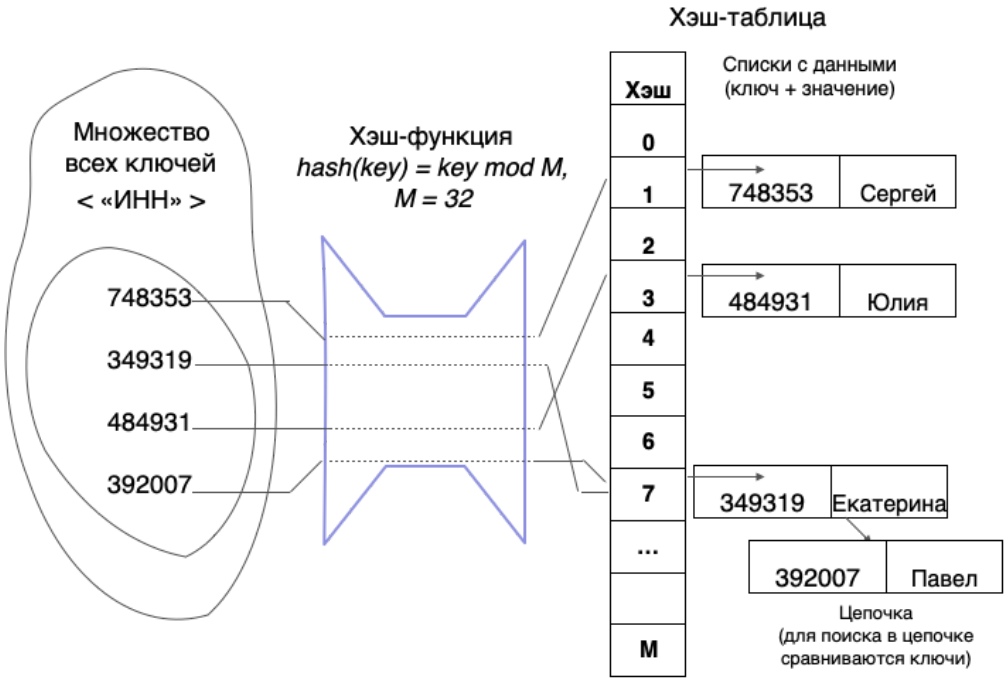
Такая хэш-функция хорошо подходит для работы с ключами, которые приводятся к целым числам, и распределены более-менее равномерно.

Коллизии

- Коллизия - это ситуация, когда два ключа в результате применения одной хэш-функции дают одно значение хэша.
- В случае с функцией $hash(k) = k \mod M$ это одинаковые остатки от деления на M.

Разрешение коллизий

Метод цепочек



Сложность операций

- Вставка: `ChainingInsert(Table, key, value)` - O(1)
- Удаление: `ChainingDelete(Table, key)` - O(1) в среднем, O(n) в худшем
- Поиск: `ChainingSearch(Table, key)` - O(1) в среднем, O(n) в худшем
- Псевдокод операций:

```
M - размер таблицы
hash: key → index ∈ {0..M-1}


ChainingInsert(Table, key, value):
    insert (key, value) at the head of list Table[hash(key)]


ChainingDelete(Table, key):
    delete (key, value) from the listTable[hash(key)]


ChainingSearch(Table, key):
    search for an element with key "key" in list Table[hash(key)]
```

Математическое ожидание частоты коллизий и сложность операций

Данный расчет верен для *простого равномерного хеширования*

- Значения хешей распределяются равномерно
- Хеши вычисляются независимо (т.е. значение хеша не зависит от вычисленных ранее значений)

У хеш-таблиц есть важный параметр α, коэффициент заполнения таблицы $\alpha = n/M$, где M - это количество цепочек, а n - количество записанных значений.

Длины списков в ячейках в сумме $= n$, так как:

- Всего записей в таблице: $n = n_0 + n_1 + n_2 + \ .. + n_{M-1}$. Матожидание количества записей в цепочке: $E[n_j] = n/M = \alpha$
- Как правило, $n > M$, следовательно, $\alpha < 1$.

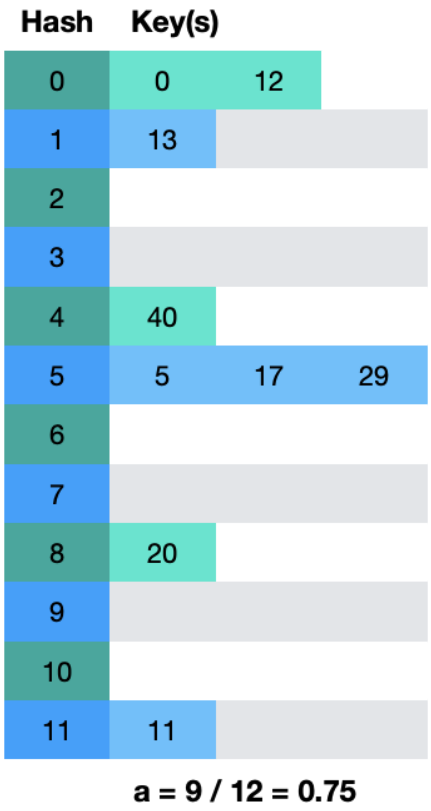
Далее, если операция вычисления хеша и поиска нужной цепочки (корзины) занимает $O(1)$, то отсается найти ожидаемое время поиска элемента в цепочке.

- Элемента нет в списке: поскольку ожидаемая длина равна α, то на промотр цепочки в среднем потребуется α шагов, и $+1$ на остальные действия, итого $O(1 + \alpha)$.
- Элемент есть в списке: ожидаемое количество элементов, которые необходимо просмотреть, так же равно $O(1 + \alpha)$.

Если $n = O(m)$ (т.е. количество элементов пропорционально количеству цепочек), то $\alpha = n/m = O(m)/m = O(1)$, и поиск (а вместе с ним и удаление) занимают $O(1)$.

Пример

На изображении используется таблица с $M = 12,\ n = 9,\ hash(k) = k \mod M$



в начало (https://otus.ru/media-private/37/b6/hash_tables_intro-31272-37b61e.html?hash=okvx7MOSeqh5taAlDqylLg&expires=1593465693#index)

Демонстрация работы метода цепочек

In [4]:
```
demonstrator = demonstrator_gen(barcelona)
```

In [5]:

```
try:
    demo = next(demonstrator)
    print(demo) if demo is not None else ""
except StopIteration:
    print("Just finished")
```

```
0: [(20, 'Roberto')]
1: [(1, 'ter Stegen')]
2: [(2, 'Semedo'), (22, 'Vidal')]
3: [(3, 'Pique'), (23, 'Umtiti')]
4: [(24, 'Vermaelen'), (4, 'Rakitic')]
5: [(5, 'Buskuets')]
6: [(6, 'Denis Suarez'), (26, 'Alena')]
7: [(7, 'Coutinho')]
8: [(8, 'Arthur')]
9: [(9, 'Luis Suarez')]
10: [(10, 'Messi')]
11: [(11, 'Dembele')]
12: [(12, 'Rafinha')]
13: [(13, 'Cillessen')]
14: [(14, 'Malcom')]
15: [(15, 'Lenglet')]
16: [(16, 'Samper')]
17: []
18: [(18, 'Alba')]
19: [(19, 'el Haddadi')]
```

Псевдокод операций

- Вставка: `ChainingInsert(Table, key, value)` - O(1)
- Удаление: `ChainingDelete(Table, key)` - O(1) в среднем, O(n) в худшем
- Поиск: `ChainingSearch(Table, key)` - O(1) в среднем, O(n) в худшем
- Псевдокод операций:

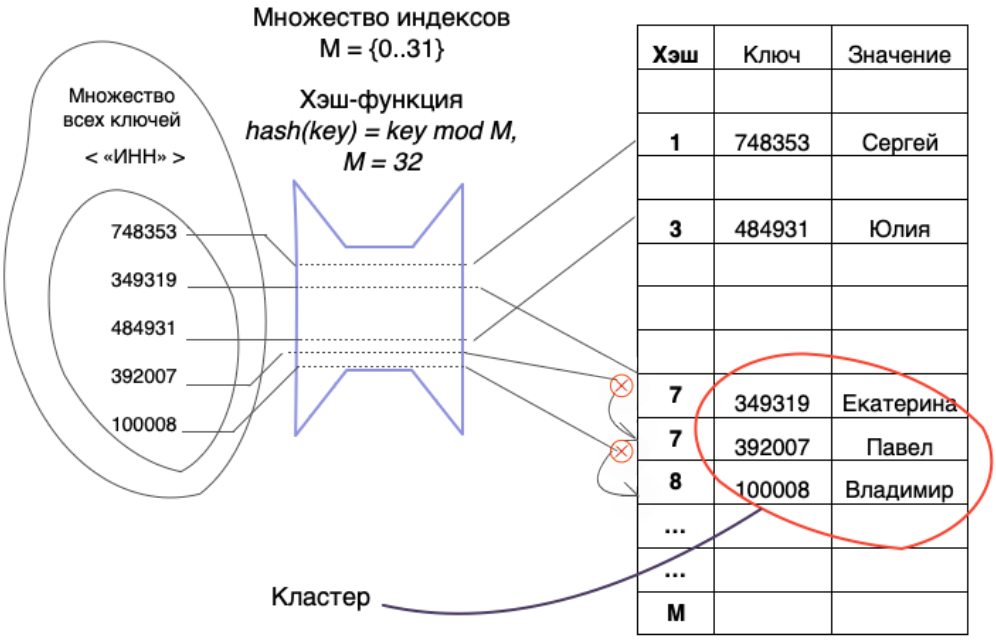
```
M - размер таблицы
hash: key → index ∈ {0..M-1}


ChainingInsert(Table, key, value):
    insert (key, value) at the head of list Table[hash(key)]


ChainingDelete(Table, key):
    delete (key, value) from the Table[hash(key)]


ChainingSearch(Table, key):
    search for an element with key "key" in list Table[hash(key)]
```

в начало (https://otus.ru/media-private/37/b6/hash_tables_intro-31272-37b61e.html?hash=okvx7MOSeqh5taAlDqylLg&expires=1593465693#index)

Открытая адресация

Основной принцип

- Идея: выбрать алгоритм (probing), по которому будет выбираться следующая ячейка, если произошла коллизия
- Искать записи при помощи того же алгоритма
- Раширять таблицу, когда она заполнится достаточно сильно (load factor)



Сложности

- Выбор хэш-функции, пробинга
- Образование кластеров
- Операция удаления

Виды пробинга

Есть несколько основных видов пробинга, от сосвсем простых до более сложных. Это, например,

- Линейный пробинг
- Квадратичный пробинг
- Двойное хэширование

Они различаются количеством вариантов *обхода* при поиске и *"равномерностью"* распределения данных по массиву.

Для того, чтобы понять, как работают таблицы, мы рассмотрим *линейный пробинг*, а потом перейдем к другим видам пробинга и сравним их.

Линейный пробинг

- $hash(k, i) = (hash'(k) + i) \mod M$
- $hash'(k, i)$ - вспомогательная хэш-функция (например, остаток от деления)
- Неравномерно, склонно к образованию кластеров
- Чем больше кластер, тем быстрее он растет: $(s + 1)/M$, где s - размер кластера

На иллюстрации используется хеш-таблица с разщмером $M = 12$

- желтые квадраты - *"сдвинутые"* ключи

- зеленые - ключи на *"своих"* местах
- $i = 1$

На иллюстрации ошибка: на самом деле, вероятность увеличения кластеров = 1/3, 5/12, 1/6!

Псевдокод операций

- Вставка: `OpenAddressInsert(Table, key, value)` - O(1) в среднем, O(n) в худшем
- Поиск: `OpenAddressSearch(Table, key)` - O(1) в среднем, O(n) в худшем
- Удаление: `OpenAddressDelete(Table, key)` - O(1) в среднем, O(n) в худшем
- Псевдокод операций (шаг $i = 1$):

```
M - размер таблицы
hash: key → index ∈ {0..M-1}


OpenAddressInsert(Table, key, value)
    i = 0
    repeat
        j = hash(key, i)
        if Table[j] == NIL
            return j
        else:
            i = i + 1
    until i == m
    return "overflow"


OpenAddressSearch(Table, key)
    i = 0
    repeat
        j = hash(key, i)
        if Table[j] == key
            return j
        i = i + 1
    until T[j] == NIL or i == m
    return NIL



</pre></font>
```

Удаление?

- Создавать дополнительный массив - "данные удалены" (замедляет поиск, "портит" _load factor_)
- Пропускаем ячеки с другим хэшем; значение с первым совпадающим ключом копируем в текущую ячейку; удаляем его. Однако на практике такой подход не используется из-за его высокой сложности и не слишком высокой эффективнсоти. Кроме того, он плозо работает со сложными схемами пробинга.

Разбор удаления при открытой адресации

Удаление с Sentinel'ом / Tombstone'ом

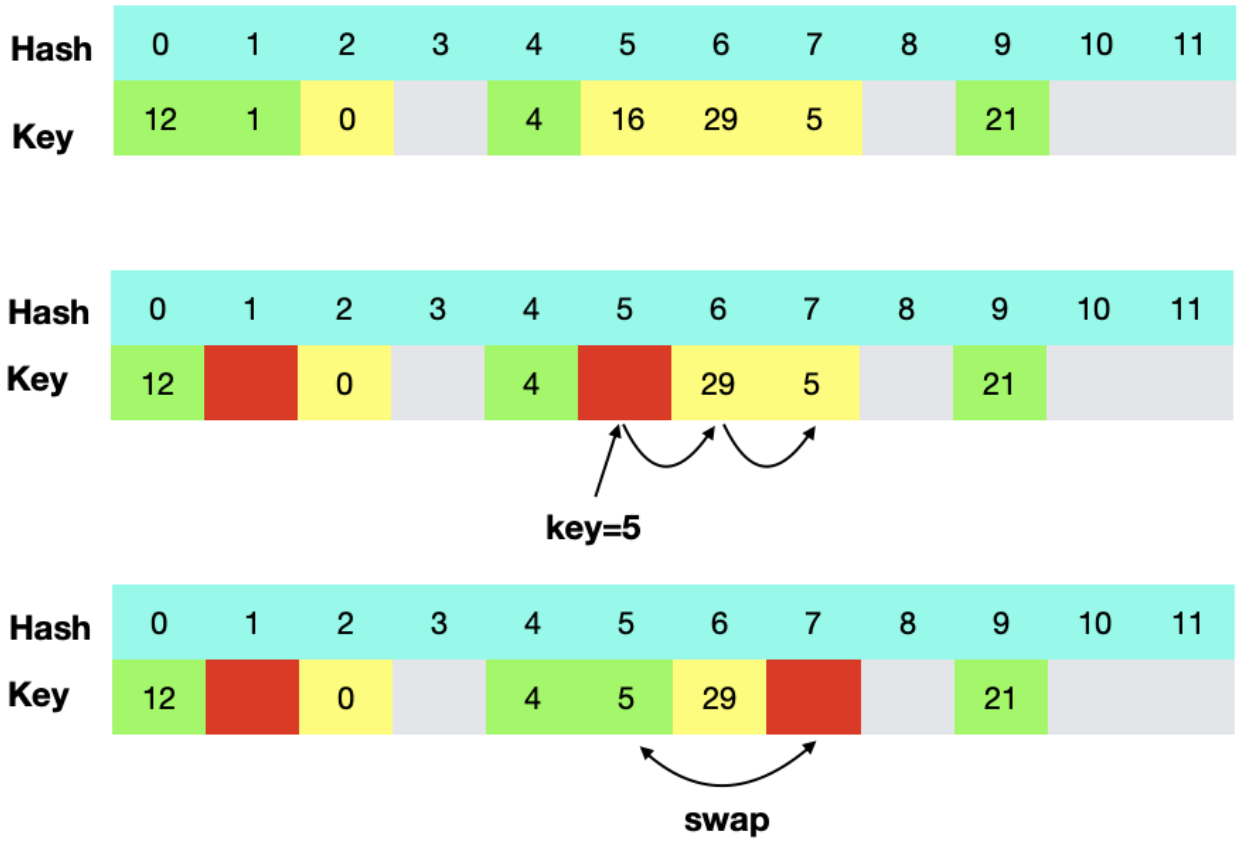
```
M - размер таблицы
hash: key → index ∈ {0..M-1}


OpenAddressInsert(Table, key, value)
    i = 0
    repeat
        j = hash(key, i)
        if Table[j] == NIL or Table[j] == DELETED:
            return j
        else:
            i = i + 1
    until i == m
    return "overflow"


OpenAddressSearch(Table, key)
    i = 0
    repeat
        j = hash(key, i)
        if Table[j] == key
            return j
        i = i + 1
    until T[j] == NIL or i == m
    return NIL


OpenAddressDeleted(Table, key)
    i = 0
    repeat
        j = hash(key, i)
        if Table[j] == DELETED
            return j
        i = i + 1
```

```
35      until T[j] == NIL or i == m
36      return NIL
```

"Ленивое" удаление

- Во время поиска элемента запомнить первый индекс, где попалось DELETED (если такой встретился)
- Поменять первый найденный DELETED с найденным элементом местами - это в будущем ускорит поиск
- Я не очень понимаю, почему такая схема называется "ленивой", с ленивыми вычислениями тут мало общего

Используется практически та же таблица, что и на прошлых изображениях. Красные квадраты - удаленные данные.



Примеры использования открытой адресации

Рассмотрим удаление на примере практической реализации общего назначнения - dictobject.c языка Python.

Sentinel для удаленных значений - DKIX_DUMMY:

```
    Dummy.  index == DKIX_DUMMY  (combined only)
       Previously held an active (key, value) pair, but that was dele
    ted and an
       active pair has not yet overwritten the slot.  Dummy can trans
    ition to
       Active upon key insertion.  Dummy slots cannot be made Unused
     again
       else the probe sequence in case of collision would have no way
    to know
       they were once active.
```

Код метода dict_popitem (dict.pop(key)), который удаляет и возвращает элемент по ключу:

```
0  static PyObject *
1  dict_popitem(PyDictObject *mp, PyObject *Py_UNUSED(ignored))
2  {
3      Py_ssize_t i, j;
4      PyDictKeyEntry *ep0, *ep;
5      PyObject *res;
6
7      res = PyTuple_New(2);
8      if (res == NULL)
9          return NULL;
10     if (mp->ma_used == 0) {
11        Py_DECREF(res);
12        PyErr_SetString(PyExc_KeyError,
13                        "popitem(): dictionary is empty");
14        return NULL;
15    }
16    /* Convert split table to combined table */
17    if (mp->ma_keys->dk_lookup == lookdict_split) {
18        if (dictresize(mp, DK_SIZE(mp->ma_keys))) {
19            Py_DECREF(res);
20            return NULL;
21        }
22    }
23    ENSURE_ALLOWS_DELETIONS(mp);
24
25    /* Pop last item */
26    ep0 = DK_ENTRIES(mp->ma_keys);
27    i = mp->ma_keys->dk_nentries - 1;
28    while (i >= 0 && ep0[i].me_value == NULL) {
29        i--;
30    }
31    assert(i >= 0);
32
33    ep = &ep0[i];
34    j = lookdict_index(mp->ma_keys, ep->me_hash, i);
```

```
    35    assert(j >= 0);
    36    assert(dictkeys_get_index(mp->ma_keys, j) == i);
    37    dictkeys_set_index(mp->ma_keys, j, DKIX_DUMMY);
    38
    39    PyTuple_SET_ITEM(res, 0, ep->me_key);
    40    PyTuple_SET_ITEM(res, 1, ep->me_value);
    41    ep->me_key = NULL;
    42    ep->me_value = NULL;
    43    /* We can't dk_usable++ since there is DKIX_DUMMY in indice
    s */
    44    mp->ma_keys->dk_nentries = i;
    45    mp->ma_used--;
    46    mp->ma_version_tag = DICT_NEXT_VERSION();
    47    assert(_PyDict_CheckConsistency(mp));
    48    return res;
    49 }
```

А вот так выглядит Search:

```
    1  static Py_ssize_t
    2  lookdict_index(PyDictKeysObject *k, Py_hash_t hash, Py_ssize_t
    index)
    3  {
    4      size_t mask = DK_MASK(k);
    5      size_t perturb = (size_t)hash;
    6      size_t i = (size_t)hash & mask;
    7
    8      for (;;) {
    9          Py_ssize_t ix = dictkeys_get_index(k, i);
    10         if (ix == index) {
    11             return i;
    12         }
    13         if (ix == DKIX_EMPTY) {
    14             return DKIX_EMPTY;
    15         }
    16         perturb >>= PERTURB_SHIFT;
    17         i = mask & (i*5 + perturb + 1);
    18     }
    19     Py_UNREACHABLE();
    20 }
```

В принципе, ничего особенного, кроме схемы сдвига, зедсь нет - это практич
ески псевдокод, переведенный на C.

[в начало (https://otus.ru/media-private/37/b6/hash_tables_intro-31272-37b6
1e.html?hash=okvx7MOSeqh5taAlDqyILg&expires=1593465693#index)](https://otus.ru/media-private/37/b6/hash_tables_intro-31272-37b61e.html?hash=okvx7MOSeqh5taAlDqyILg&expires=1593465693#index)

Виды пробинга

Линейное исследование

Было разобрано выше (https://otus.ru/media-private/37/b6/hash_tables_intro-31272-37b61e.html?hash=okvx7MOSeqh5taAlDqyILg&expires=1593465693#problin)

Квадратичное исследование

- $hash(k,i) = (hash'(k) + c_1 i + c_2 i^2) \mod M$.
- Лучше линейного, но нужно подбирать c_1, c_2, M.

Несколько популярных вариантов выбора констант

- $hash(k) = (hash'(k) + i^2) \mod M$, где $c_1 = 0, c_2 = 1, M -$ простое > 3, фактор заполнения $\alpha < 1/2$
- $hash(k) = (hash'(k) + (i + i^2)\,/\,2) \mod M$, где $c_1 = c_2 = 1/2, M = 2^k$
- $hash(k) = (hash'(k) + -1^i \cdot i^2) \mod M$, где $M \equiv 3\ mod\ 4$

Почему $\alpha < 1/2$? Пусть есть x и y, указывающие на одну локацию, но $x \neq y$, и $0 \leq x, y \leq M/2$.

$$hash(k) + x^2 \equiv hash(k) + y^2 mod M$$
$$x^2 \equiv y^2 mod M$$
$$x^2 - y^2 \equiv mod M$$
$$(x - y) \cdot (x + y) \equiv mod M$$

$x - y \neq 0, x + y \neq 0$ - значит, существует M/2 различных мест для записи.

Двойное хэширование

- $hash(k, i) = (hash_1(k) + i \times hash_2(k)) \mod M$.
- Дает m^2 возможных последовательностей, более равномерно
- Значение $hash_2(k)$ всегда должно быть взаимно простым с M (для обхода всей таблицы)
  - как вариант, для достаточно большого M:
  - $h_1(k) = k \mod M$
  - $h_2(k) = 1 + (k \mod (M - 1))$

в начало (https://otus.ru/media-private/37/b6/hash_tables_intro-31272-37b61e.html?hash=okvx7MOSeqh5taAlDqylLg&expires=1593465693#index)

Домашняя работа

- Домашняя работа одна на всю неделю. Это *первая половина*, которую можно начать выполнять сейчас.
- Вторая половина будет после занятия в среду.

1. Реализовать хеш-таблицу, использующую метод цепочек
   - дополнительно: для хранения внутри цепочек при достижении значительного числа элементов (~32) заменять их на BST
2. *Или*: реализовать хеш-таблицу с открытой адресацией
   - дополнительно: реализовать "ленивое" удаление
   - реализовать квадратичный пробинг

Краткая справка

C++

- `std::unordered_map`
- Метод цепочек - для универсальноси
- https://bit.ly/2RNxPnD (в другом разделе ссылка дублируется)
- Можно "легко" сделать под свои нужды

Java 8

- `java.util.HashMap`
- Метод цепочек
- Используются деревья внтури одной корзины

C#

- `System.Collections.Hashtable` и `System.Collections.Generic.Dictionary`
- Hashtable: открытая адресация, двойное хэширование
- Dictionary: метод цепочек
- https://bit.ly/2C0ryPW (в другом разделе ссылка дублируется)

в начало (https://otus.ru/media-private/37/b6/hash_tables_intro-31272-37b61e.html?hash=okvx7MOSeqh5taAlDqylLg&expires=1593465693#index)

Ссылки

Python

- ссылка на код [dictobject.c] (https://hg.python.org/cpython/file/52f68c95e025/Objects/dictobject.c#l33)
- [как выбиралась реализация] (https://github.com/python/cpython/blob/master/Objects/dictnotes.txt)
- [Просто хорошая статья на английском с пояснениями исходного кода dict()] (https://www.laurentluce.com/posts/python-dictionary-implementation/)

C++

- Тут стоит обратить внимание на комментарии
  - [std::unordered_map](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/unordered__map-source.html)
  - [hashtable](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/hashtable-source.html)
- Почему именно так - [по ссылке](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1456.html). Стена текста!

Java

- Подробности про реализацию корзин в `java.utils.HashMap` [тут] (http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/HashM (комментарии к исходнику)

C#

- Повторение [ссылки](https://bit.ly/2C0ryPW) выше - объяснение реализации HashTable и Dictionary
- Исходник [HashTable] (https://referencesource.microsoft.com/#mscorlib/system/collections/hashtable.cs)
- Исходник [Dictionary] (https://referencesource.microsoft.com/#mscorlib/system/collections/generic/dictionary.cs)

в начало (https://otus.ru/media-private/37/b6/hash_tables_intro-31272-37b61e.html?

In [ ]:

In [ ]:

In [ ]:

In [ ]: