Мини-задача #27 (1 балл)

Реализуйте стек, операция pop которого выталкивает наиболее часто встречаемый в стеке элемент (если есть ничья, выбирается элемент, который ближе к вершине стека)

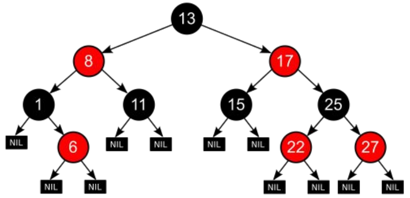
Можно использовать структуры данных из стандартной библиотеки (включая словари).

https://leetcode.com/problems/maximum-frequency-stack

Алгоритмы и структуры данных

Хеш-таблицы

Сбалансированные деревья поиска



Операции:

1. find(value) -> O(logN)
2. select(i) -> O(logN)
3. min/max -> O(logN)
4. pred/succ(ptr) -> O(logN)
5. rank(value) -> O(logN)
6. вывод в пор. возрастания -> O(N)

---

7. insert(value) -> O(logN)
8. remove(value) -> O(logN)

Деревья поиска дают самый широкий набор операций

Пирамиды лучше здесь (по константам или даже по асимптотике)

Спойлер: хеш-таблицы лучше здесь (при правильной реализации дадут O(1))

Хеш-множества

Операции:

1. find(value)   -> O(1)

----

7. insert(value)   -> O(1)
8. remove(value)   -> O(1)

Спойлер: хеш-таблицы лучше здесь (при правильной реализации дадут O(1))

Хеш-множества

Операции:

```
1.  find(value)     -> O(1)
2.  insert(value)  -> O(1)
3.  remove(value)  -> O(1)
---------------------------
```

Хеш-множества

Операции:

```
1.  find(value)     -> O(1)
2.  insert(value)   -> O(1)
3.  remove(value)   -> O(1)
--------------------------
4.  iterate         -> O(N)
```

Перебрать все элементы, но без гарантии какого-либо порядка

Хеш-множества

Операции:

```
1.  find(value)     -> O(1)
2.  insert(value)   -> O(1)
3.  remove(value)   -> O(1)
--------------------------
4.  iterate         -> O(N)
```

Перебрать все элементы, но без гарантии какого-либо порядка

Использования:

1. Быстрый ответ на вопрос: встречали ли мы уже этот элемент

Хеш-множества

Операции:

```
1.  find(value)     -> O(1)
2.  insert(value)   -> O(1)
3.  remove(value)   -> O(1)
-----------------------------
4.  iterate         -> O(N)
```

Перебрать все элементы, но без гарантии какого-либо порядка

Использования:

1. Быстрый ответ на вопрос: встречали ли мы уже этот элемент
2. Динамическое изменение структуры

Хеш-множества. Пример использования: 2-SUM

Задача: дан неупорядоченный массив целых чисел. Понять, есть ли в нем значения x и y, такие, что x + y = t

Хеш-множества. Пример использования: 2-SUM

Задача: дан неупорядоченный массив целых чисел. Понять, есть ли в нем значения x и y, такие, что x + y = t

Наивное решение: двойной цикл, работает за O(N^2)

Хеш-множества. Пример использования: 2-SUM

Задача: дан неупорядоченный массив целых чисел. Понять, есть ли в нем значения x и y, такие, что x + y = t

Наивное решение: двойной цикл, работает за O(N^2)

Чуть более хорошее решение: отсортировать массив => пройтись по каждому элементу x и искать бинарным поиском в массиве элемент t - x

Хеш-множества. Пример использования: 2-SUM

Задача: дан неупорядоченный массив целых чисел. Понять, есть ли в нем значения x и y, такие, что x + y = t

Наивное решение: двойной цикл, работает за O(N^2)

Чуть более хорошее решение: отсортировать массив => пройтись по каждому элементу x и искать бинарным поиском в массиве элемент t - x

Работает за O(N*logN) - сортировка + N бинарных поисков.

Хеш-множества. Пример использования: 2-SUM

Задача: дан неупорядоченный массив целых чисел. Понять, есть ли в нем значения x и y, такие, что x + y = t

Наивное решение: двойной цикл, работает за O(N^2)

Чуть более хорошее решение: отсортировать массив => пройтись по каждому элементу x и искать бинарным поиском в массиве элемент t - x

Работает за O(N*logN) - сортировка + N бинарных поисков.

Хорошее решение: использовать hashset => добавлять туда очередной элемент x и искать в нем элемент t - x.

Хеш-множества. Пример использования: 2-SUM

Задача: дан неупорядоченный массив целых чисел. Понять, есть ли в нем значения x и y, такие, что x + y = t

Наивное решение: двойной цикл, работает за O(N^2)

Чуть более хорошее решение: отсортировать массив => пройтись по каждому элементу x и искать бинарным поиском в массиве элемент t - x

Работает за O(N*logN) - сортировка + N бинарных поисков.

Хорошее решение: использовать hashset => добавлять туда очередной элемент x и искать в нем элемент t - x.

Сработает за O(N) - т.к. N поисков и N добавлений

Хеш-множества. Пример использования: 2-SUM

Задача: дан неупорядоченный массив целых чисел. Понять, есть ли в нем значения x и y, такие, что x + y = t

Наивное решение: двойной цикл, работает за O(N^2)

Чуть более хорошее решение: отсортировать массив => пройтись по каждому элементу x и искать бинарным поиском в массиве элемент t - x

Работает за O(N*logN) - сортировка + N бинарных поисков.

Хорошее решение: использовать hashset => добавлять туда очередной элемент x и искать в нем элемент t - x.

Сработает за O(N) - т.к. N поисков и N добавлений
(вот только что это за сложность?)

Мини-задача #27 (1 балл)

Реализуйте стек, операция pop которого выталкивает наиболее часто встречаемый в стеке элемент (если есть ничья, выбирается элемент, который ближе к вершине стека)

Можно использовать структуры данных из стандартной библиотеки (включая словари).

https://leetcode.com/problems/maximum-frequency-stack

Хеш-таблицы

Операции:

```
1.  find(key) -> value | None        -> O(1)
2.  insert(key, value)               -> O(1)
3.  remove(key)                      -> O(1)
 ----------------------------------------
4.  iterate                          -> O(N)
```

Хеш-таблицы

Операции:

```
1.  find(key) -> value | None      -> O(1)
2.  insert(key, value)             -> O(1)
3.  remove(key)                    -> O(1)
-----------------------------------------
4.  iterate                        -> O(N)
```

Использования:

1. Быстрое получение данных, если уже встречали элемент
2. Динамическое изменение структуры

Хеш-таблицы

Операции:

```
1.  find(key) -> value | None       -> O(1)
2.  insert(key, value)              -> O(1)
3.  remove(key)                     -> O(1)
 -----------------------------------------
4.  iterate                         -> O(N)
```

Использования:

1. Быстрое получение данных, если уже встречали элемент
2. Динамическое изменение структуры

Часто используют, как словарь (изначально в компиляторах)

Хеш-таблицы

Идея: воспользуемся свойствами массивов, а именно доступом к i-ому элементу за O(1).

Хеш-таблицы

Идея: воспользуемся свойствами массивов, а именно доступом к i-ому элементу за O(1).

Пусть элементы/ключи - это натуральные числа,

{1, 2, 3, 42, 13}

…	1	2	3	…	…	…	13	…	…	…	…	…	…	…	…	…	42	…	…

Заведем массив arr, будем хранить значение k по индексу k.
Доступ arr[k].

Хеш-таблицы

Идея: воспользуемся свойствами массивов, а именно доступом к i-ому элементу за O(1).

Пусть элементы/ключи - это натуральные числа,

{1, 2, 3, 42, 13}

...	1	2	3	...	...	...	13	...	...	...	...	...	...	...	...	...	42	...	...

Заведем массив arr, будем хранить значение k по индексу k.
Доступ arr[k] за O(1), добавление и удаление - аналогично.

Хеш-таблицы

Идея: воспользуемся свойствами массивов, а именно доступом к i-ому элементу за O(1).

Пусть элементы/ключи - это натуральные числа,

{1, 2, 3, 42, 13}

…	1	2	3	…	…	…	13	…	…	…	…	…	…	…	…	…	42	…	…

Заведем массив arr, будем хранить значение k по индексу k.
Доступ arr[k] за O(1), добавление и удаление - аналогично.

Проблемы:

1. ключи не всегда натуральные числа

Хеш-таблицы

Идея: воспользуемся свойствами массивов, а именно доступом к i-ому элементу за O(1).

Пусть элементы/ключи - это натуральные числа,

{1, 2, 3, 42, 13}

…	1	2	3	…	…	…	13	…	…	…	…	…	…	…	…	…	42	…	…

Заведем массив arr, будем хранить значение k по индексу k.
Доступ arr[k] за O(1), добавление и удаление - аналогично.

Проблемы:

1. ключи не всегда натуральные числа
2. память конечна => а множество ключей - счетно

Хеш-таблицы

…	1	2	3	…	…	…	13	…	…	…	…	42	…	…

Пусть у нас есть массив размера m.
И пусть U - множество ключей, |U| ⩾ m.

Хеш-таблицы

…	1	2	3	…	…	…	13	…	…	…	…	42	…	…

Пусть у нас есть массив размера m.
И пусть U - множество ключей, |U| ⩾ m.

Тогда функцию $h : U \to \{0, 1, 2, \ldots, m - 1\}$ назовем хеш-функцией.

Хеш-таблицы

…	1	2	3	…	…	…	13	…	…	…	…	42	…	…

Пусть у нас есть массив размера m.
И пусть U - множество ключей, |U| ⩾ m.

Тогда функцию $h : U \to \{0, 1, 2, \ldots, m-1\}$ назовем хеш-функцией.

Обычно:

$$\boxed{\mathrm{U}} \longrightarrow \boxed{\mathbb{Z}} \longrightarrow \boxed{\{0, 1, 2, \ldots, m-1\}}$$

Хеш-таблицы

…	1	2	3	…	…	…	13	…	…	…	…	42	…	…

Пусть у нас есть массив размера m.
И пусть U - множество ключей, |U| ⩾ m.

Тогда функцию $h : U \to \{0, 1, 2, \ldots, m-1\}$ назовем хеш-функцией.

Обычно:

$$\mathrm{U} \xrightarrow[\texttt{hashCode}]{} \mathbb{Z} \xrightarrow[\text{функция сжатия}]{} \{0, 1, 2, \ldots, m-1\}$$

Хеш-таблицы

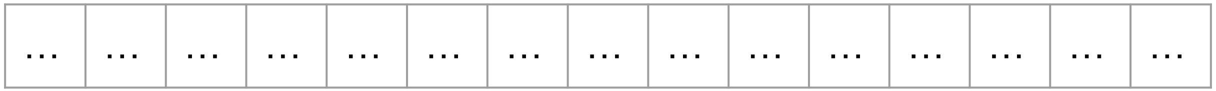
…	1	2	3	…	…	…	13	…	…	…	…	42	…	…

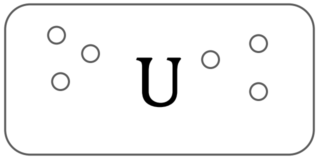
Пусть у нас есть массив размера m.
И пусть U - множество ключей, |U| ⩾ m.

Тогда функцию $h : U \to \{0, 1, 2, \ldots, m-1\}$ назовем хеш-функцией.

Также потребуем, чтобы хеш-функция работала за O(1), т.е. по любому ключу мы могли бы за константное время получить индекс в массиве.

Хеш-таблицы

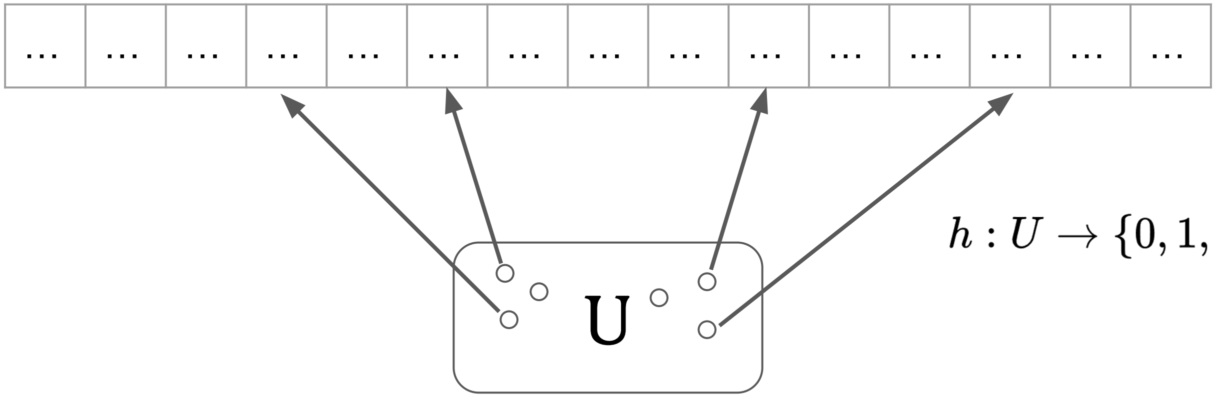




Пусть у нас есть массив размера m.
И пусть U - множество ключей, |U| ⩾ m.

Тогда функцию $h : U \to \{0, 1, 2, \ldots, m-1\}$ назовем хеш-функцией.

Хеш-таблицы

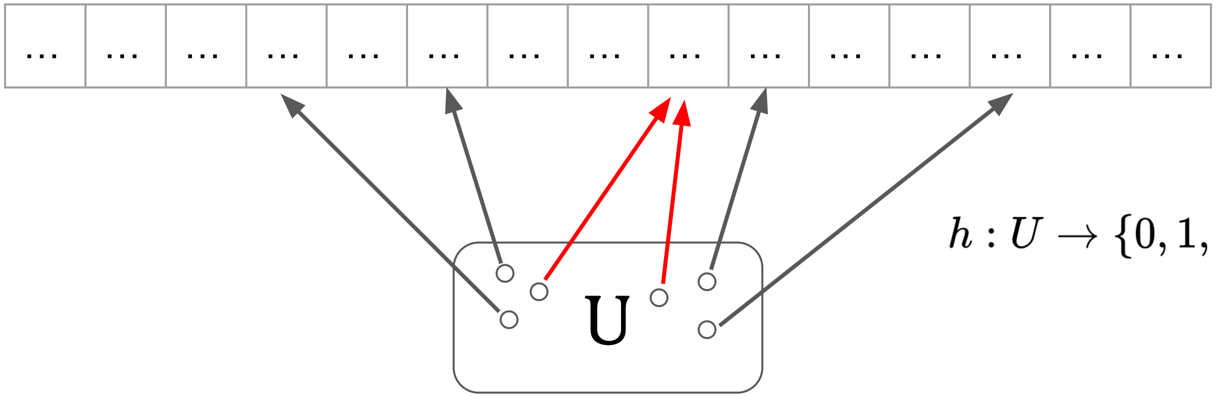


$$h : U \to \{0, 1, 2, \ldots, m-1\}$$

Пусть у нас есть массив размера m.
И пусть U - множество ключей, |U| ⩾ m.

Тогда функцию $h : U \to \{0, 1, 2, \ldots, m-1\}$ назовем хеш-функцией.

Хеш-таблицы

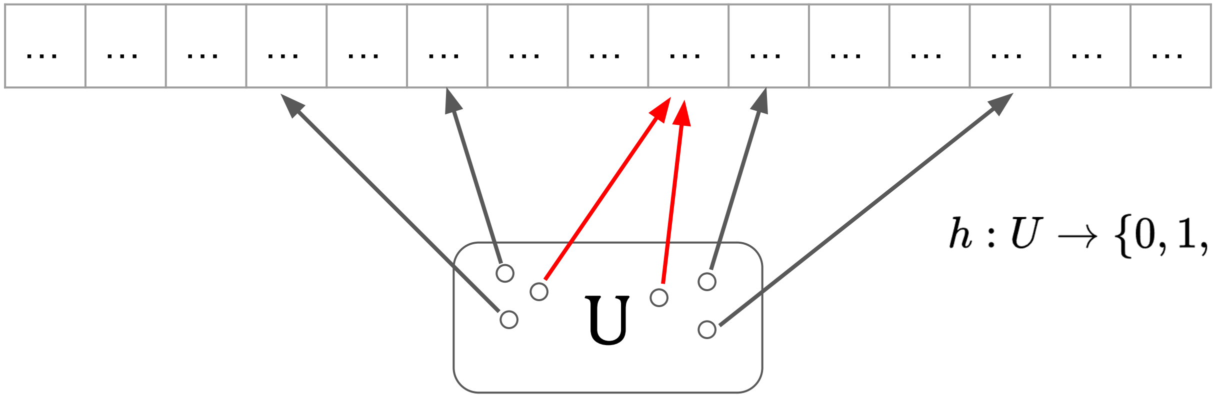


$h : U \to \{0, 1, 2, \ldots, m-1\}$

Пусть у нас есть массив размера m.
И пусть U - множество ключей, |U| ⩾ m.

Тогда функцию $h : U \to \{0, 1, 2, \ldots, m-1\}$ назовем хеш-функцией.

Хеш-таблицы

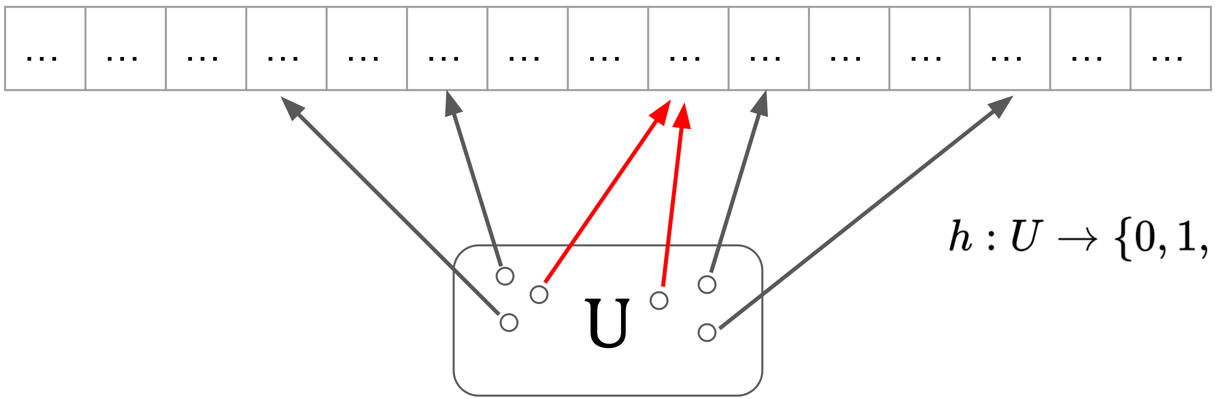


$h : U \to \{0, 1, 2, \ldots, m-1\}$

Пусть у нас есть массив размера m.
И пусть U - множество ключей, |U| ⩾ m.

Тогда функцию $h : U \to \{0, 1, 2, \ldots, m-1\}$ назовем хеш-функцией.
Ситуацию: $x, y \in U : h(x) = h(y)$ будем называть коллизией.

Хеш-таблицы



$h : U \to \{0, 1, 2, \ldots, m-1\}$

Что делать в такой ситуации?

Пусть у нас есть массив размера m.
И пусть U - множество ключей, |U| ⩾ m.

Тогда функцию $h : U \to \{0, 1, 2, \ldots, m-1\}$ назовем хеш-функцией.
Ситуацию: $x, y \in U : h(x) = h(y)$ будем называть коллизией.

Хеш-таблицы: метод цепочек

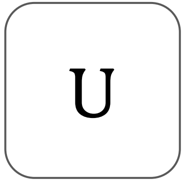
U

U - строки, фамилии

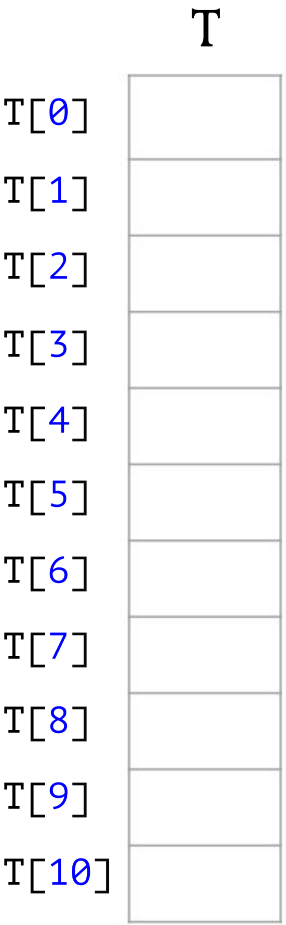
T

T
T[0]
T[1]
T[2]
T[3]
T[4]
T[5]
T[6]
T[7]
T[8]
T[9]
T[10]

Хеш-таблицы: метод цепочек



U - строки, фамилии



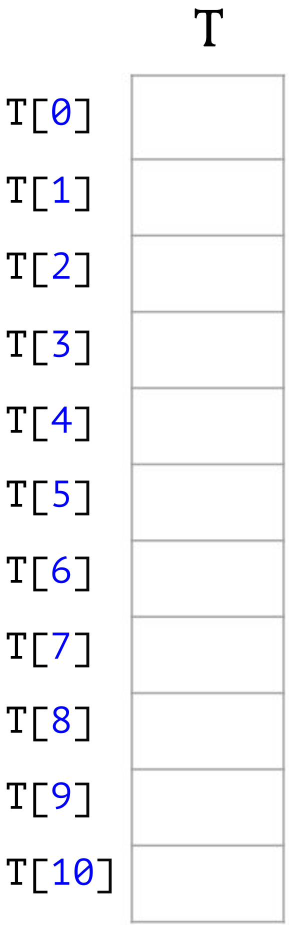
Давайте хранить в массиве не элементы, а головы связных списков

Хеш-таблицы: метод цепочек

U

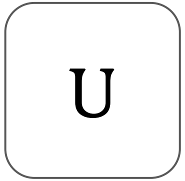
U - строки, фамилии

insert("Иванов")



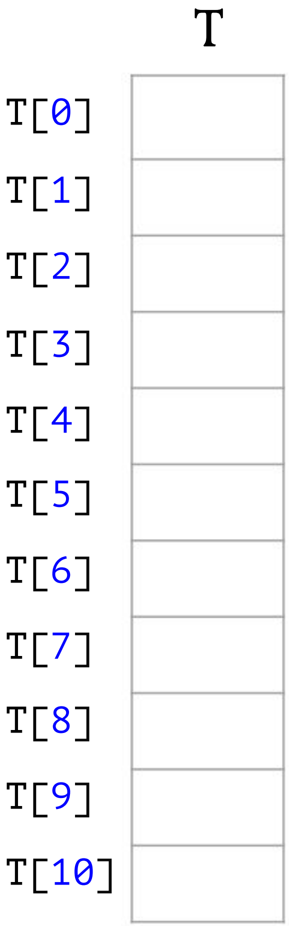
Давайте хранить в массиве не элементы, а головы связных списков

Хеш-таблицы: метод цепочек



U - строки, фамилии

```
insert("Иванов")
h("Иванов") = 6
```

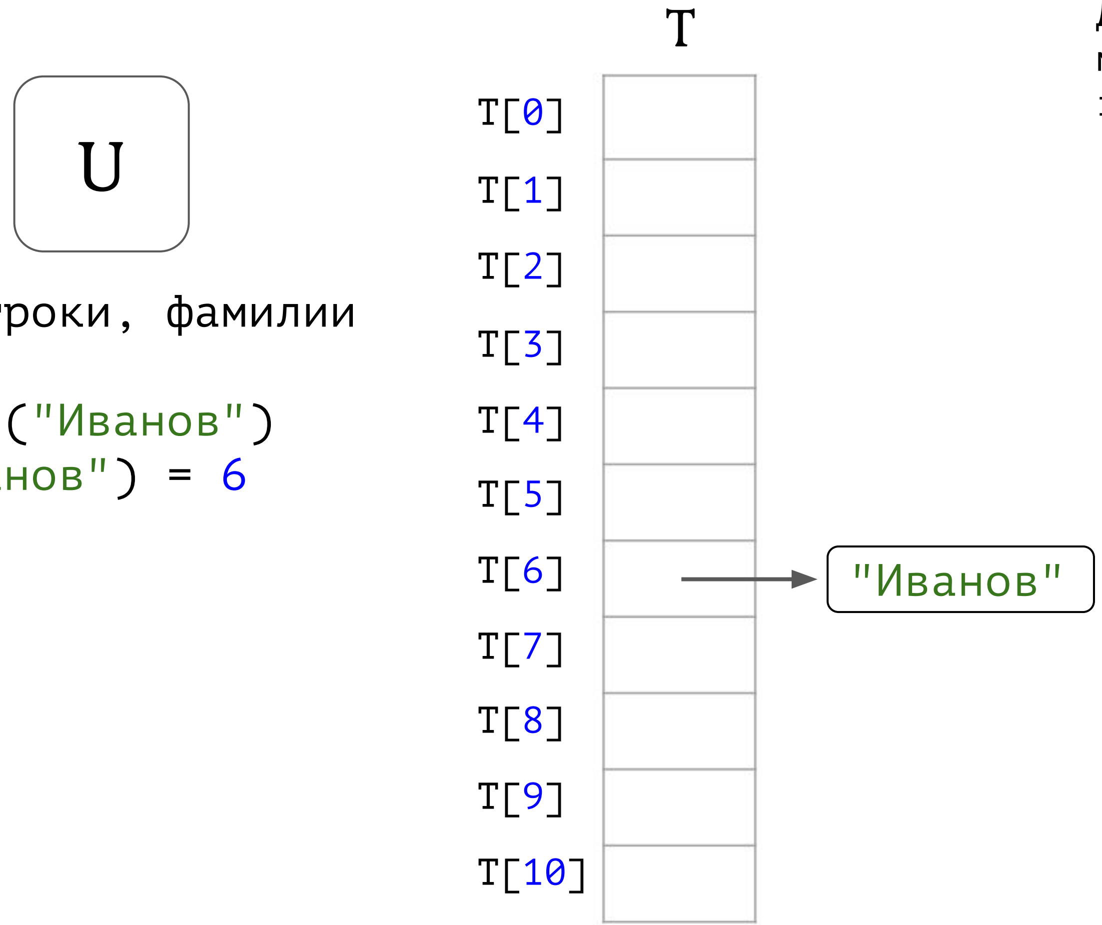


Давайте хранить в массиве не элементы, а головы связных списков

Хеш-таблицы: метод цепочек

U - строки, фамилии

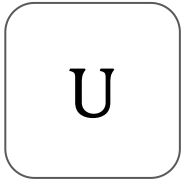
```
insert("Иванов")
h("Иванов") = 6
```



Давайте хранить в массиве не элементы, а головы связных списков

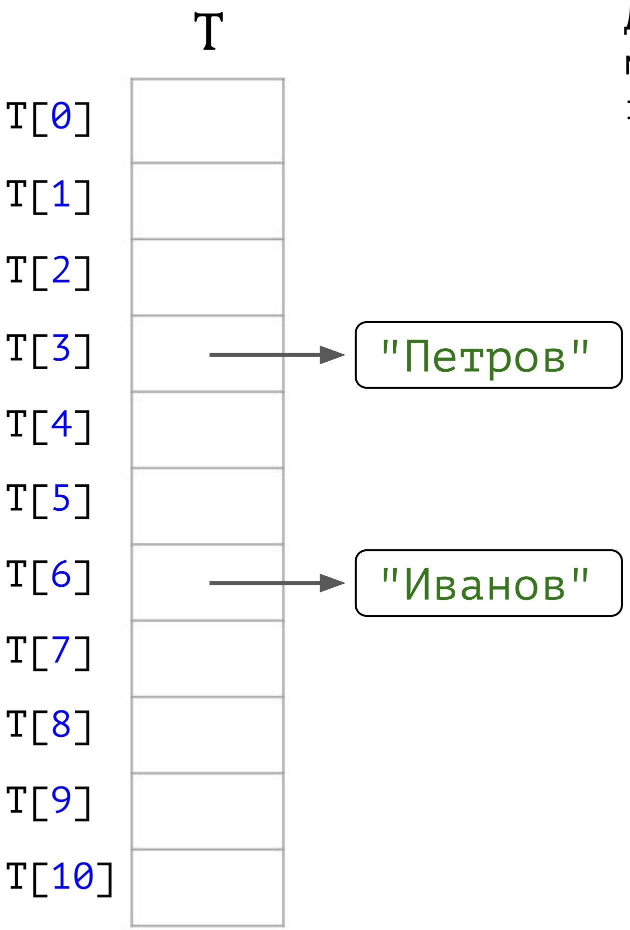
Хеш-таблицы: метод цепочек

Давайте хранить в массиве не элементы, а головы связных списков



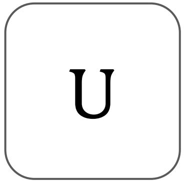
U - строки, фамилии

```
insert("Петров")
h("Петров") = 3
```



Хеш-таблицы: метод цепочек

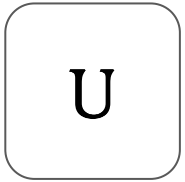
Давайте хранить в массиве не элементы, а головы связных списков



U - строки, фамилии

```
insert("Сидорова")
h("Сидорова") = 8
```

Хеш-таблицы: метод цепочек



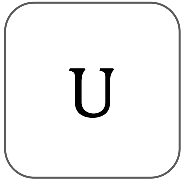
U - строки, фамилии

```
insert("Кузнецов")
h("Кузнецов") = 6
```

Произошла коллизия!

Давайте хранить в массиве не элементы, а головы связных списков

Хеш-таблицы: метод цепочек

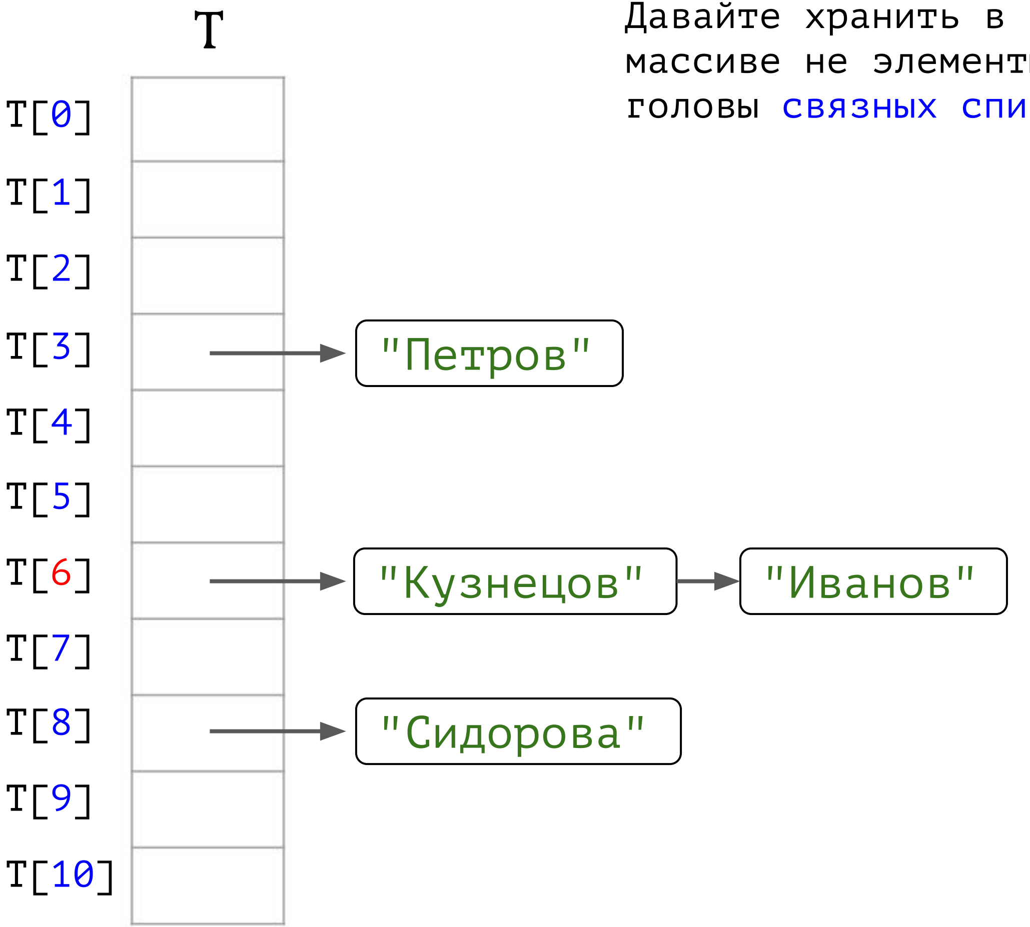


U - строки, фамилии

insert("Кузнецов")
h("Кузнецов") = 6

Произошла коллизия!

Добавляем в голову связного списка из соответствующей ячейки.



Давайте хранить в массиве не элементы, а головы связных списков

Хеш-таблицы: метод цепочек

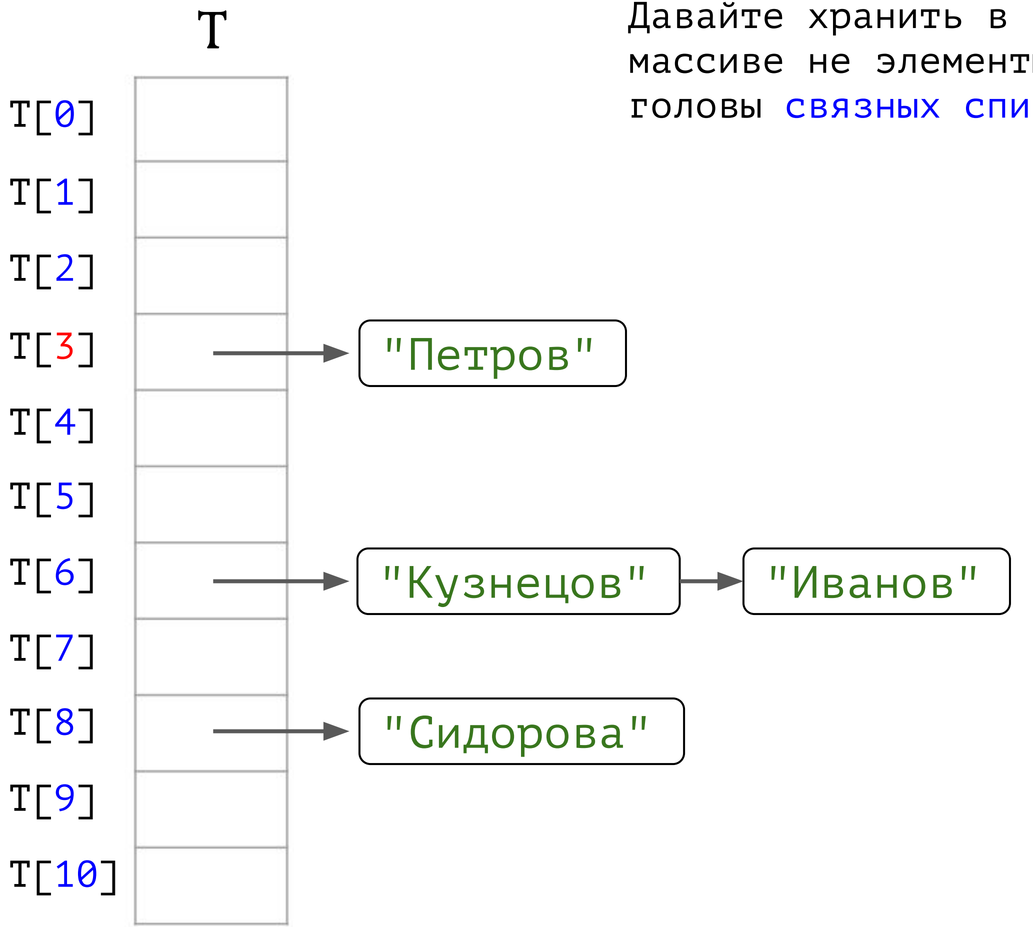
Давайте хранить в массиве не элементы, а головы связных списков

U

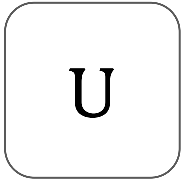
U - строки, фамилии

insert("Ким")
h("Ким") = 3

Произошла коллизия!



Хеш-таблицы: метод цепочек

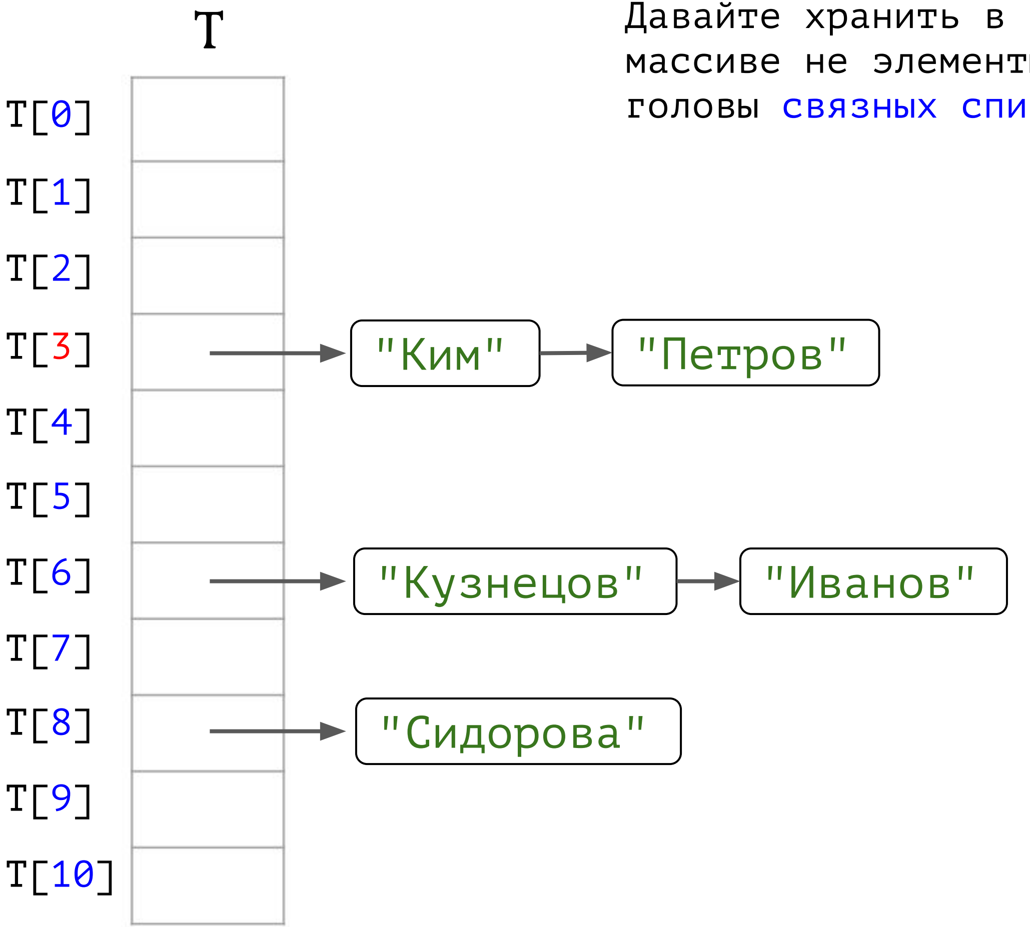


U - строки, фамилии

insert("Ким")
h("Ким") = 3

Произошла коллизия!

Добавляем в голову связного списка из соответствующей ячейки.



Давайте хранить в массиве не элементы, а головы связных списков

Хеш-таблицы: метод цепочек

U

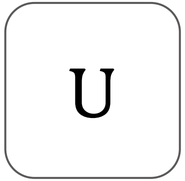
U - строки, фамилии

Как теперь искать?

find("Иванов")

Давайте хранить в массиве не элементы, а головы связных списков

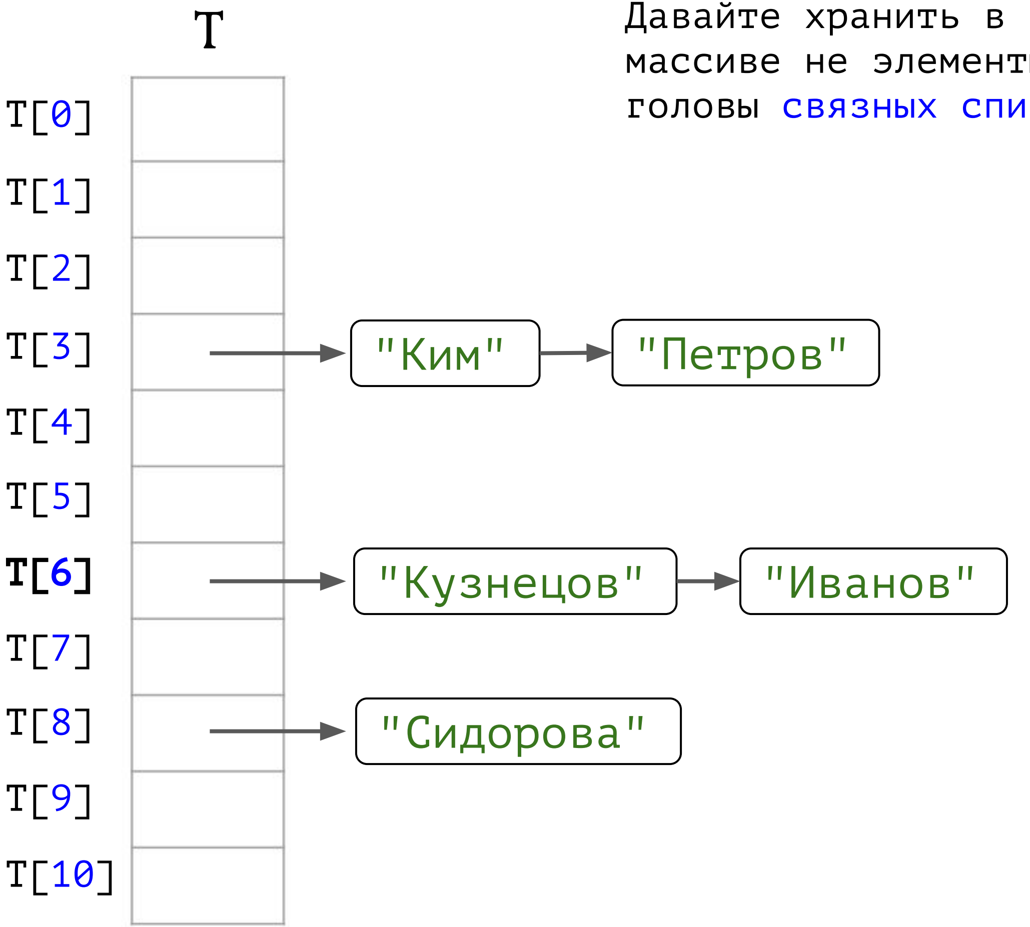
Хеш-таблицы: метод цепочек



U - строки, фамилии

Как теперь искать?

find("Иванов")
h("Иванов") = 6



Давайте хранить в массиве не элементы, а головы связных списков

Хеш-таблицы: метод цепочек

Давайте хранить в массиве не элементы, а головы связных списков

U

U - строки, фамилии

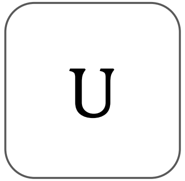
Как теперь искать?

find("Иванов")
h("Иванов") = 6

Дальше ищем по списку, пока не найдем значение или он не закончится.

T

Хеш-таблицы: метод цепочек



U - строки, фамилии

Как теперь удалять?

remove("Иванов")

Давайте хранить в массиве не элементы, а головы связных списков

Хеш-таблицы: метод цепочек

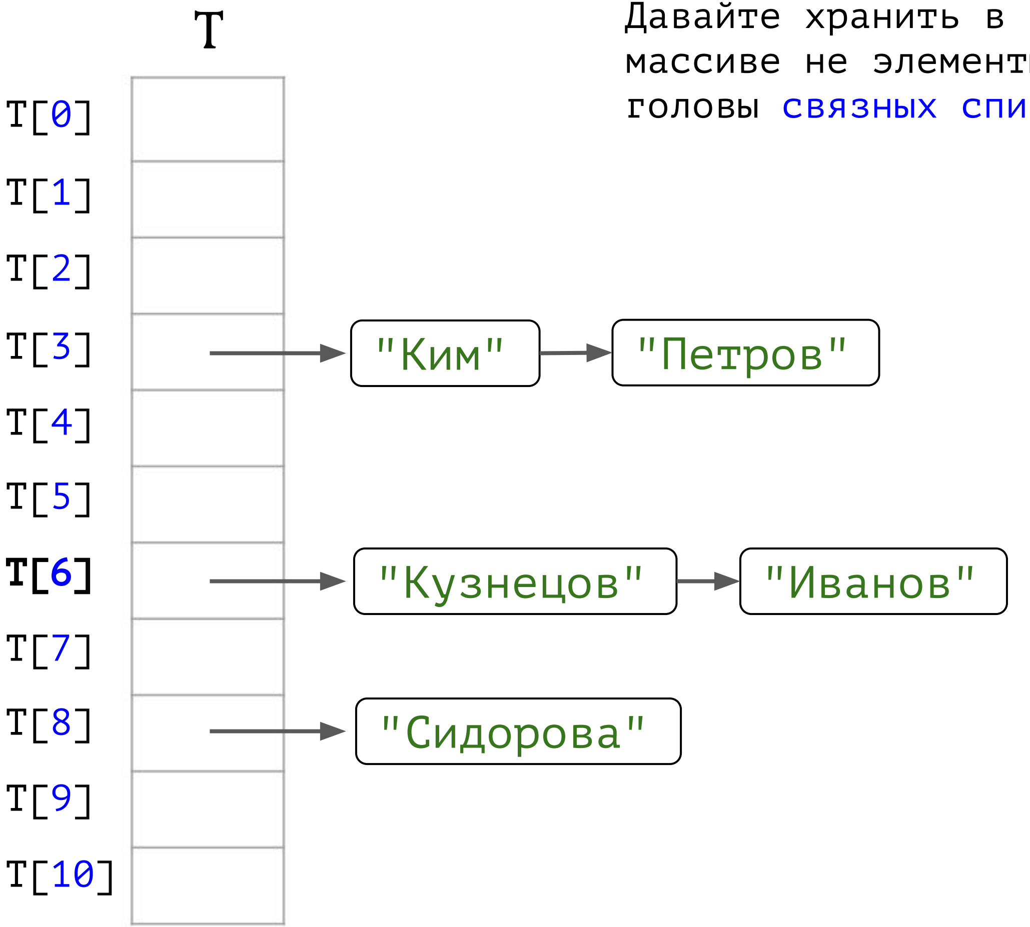
Давайте хранить в массиве не элементы, а головы связных списков

U

U - строки, фамилии

Как теперь удалять?

```
remove("Иванов")
h("Иванов") = 6
```



Хеш-таблицы: метод цепочек

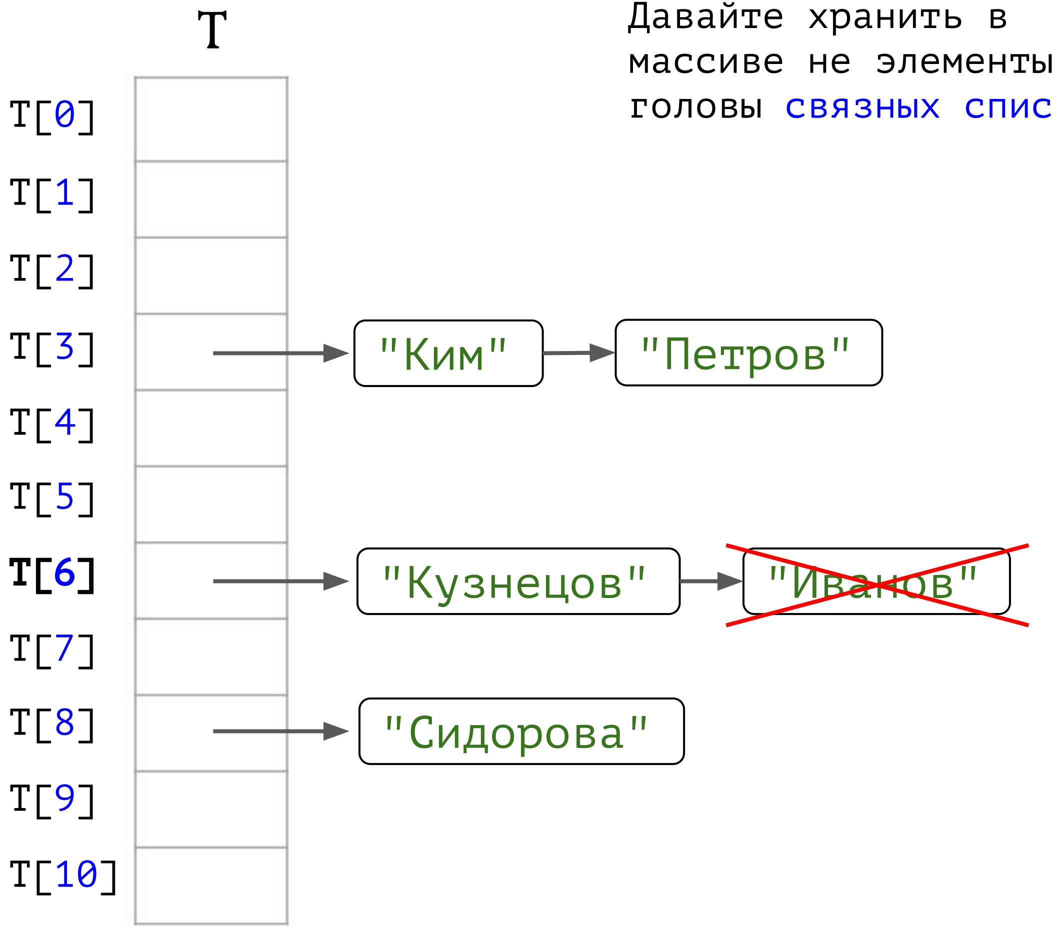
U

U - строки, фамилии

Как теперь удалять?

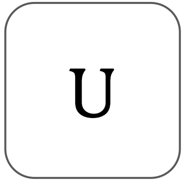
remove("Иванов")
h("Иванов") = 6

Снова ищем по списку, потом удаляем.



Давайте хранить в массиве не элементы, а головы связных списков

Хеш-таблицы: метод цепочек

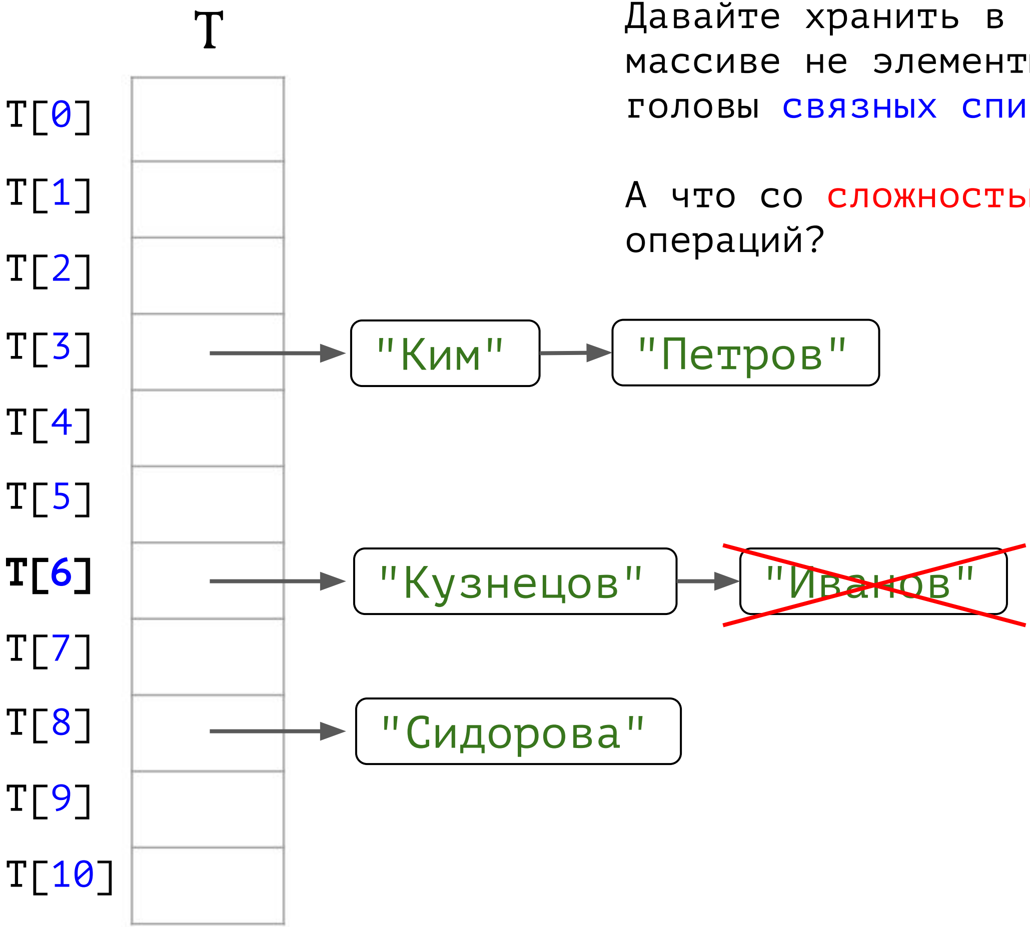


U - строки, фамилии

Как теперь удалять?

remove("Иванов")
h("Иванов") = 6

Снова ищем по списку, потом удаляем.



Давайте хранить в массиве не элементы, а головы связных списков

А что со сложностью операций?

Хеш-таблицы

Операции:

```
1.  find(key) -> value | None       -> O(1)
2.  insert(key, value)              -> O(1)
3.  remove(key)                     -> O(1)
 -----------------------------------------
4.  iterate                         -> O(N)
```

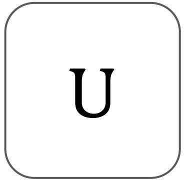
Использования:

1. Быстрое получение данных, если уже встречали элемент
2. Динамическое изменение структуры

Часто используют, как словарь (изначально в компиляторах)

Хеш-таблицы: метод цепочек

Давайте хранить в массиве не элементы, а головы связных списков

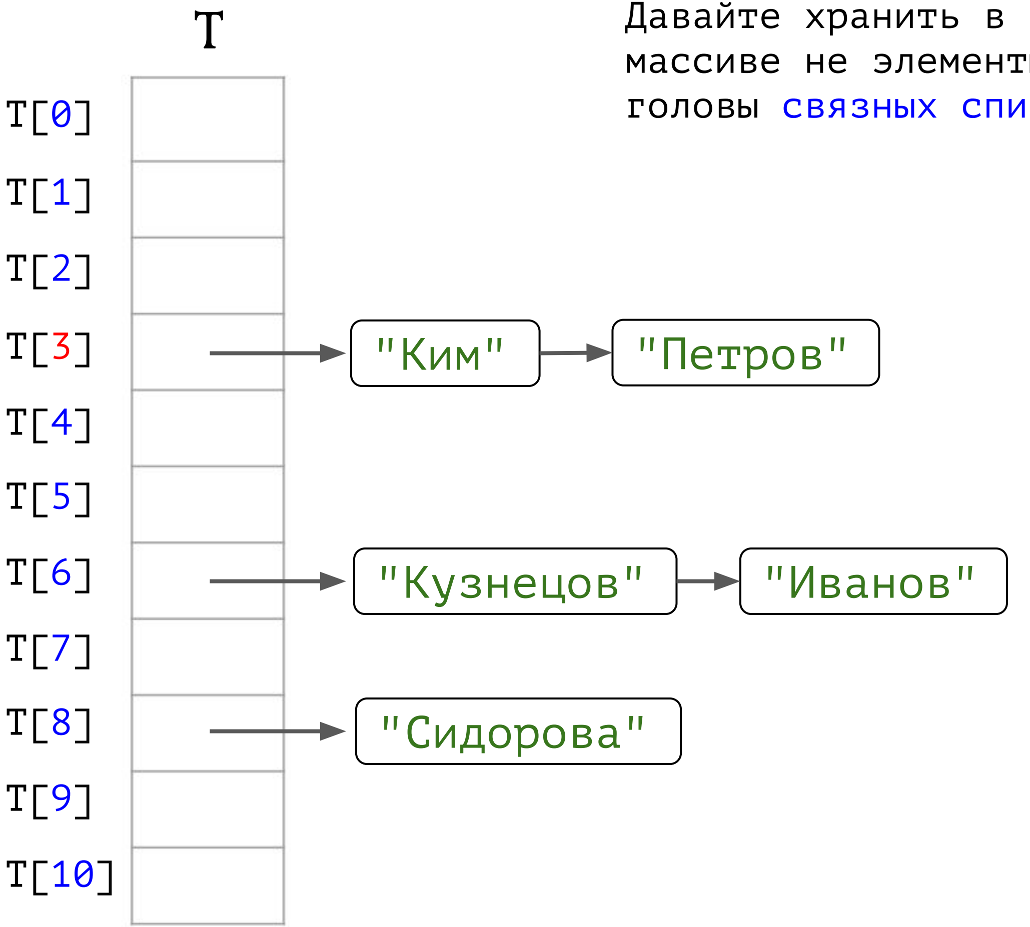


U - строки, фамилии

```
insert("Ким")
h("Ким") = 3
```

Произошла коллизия!

Добавляем в голову связного списка из соответствующей ячейки.



Хеш-таблицы: метод цепочек

U

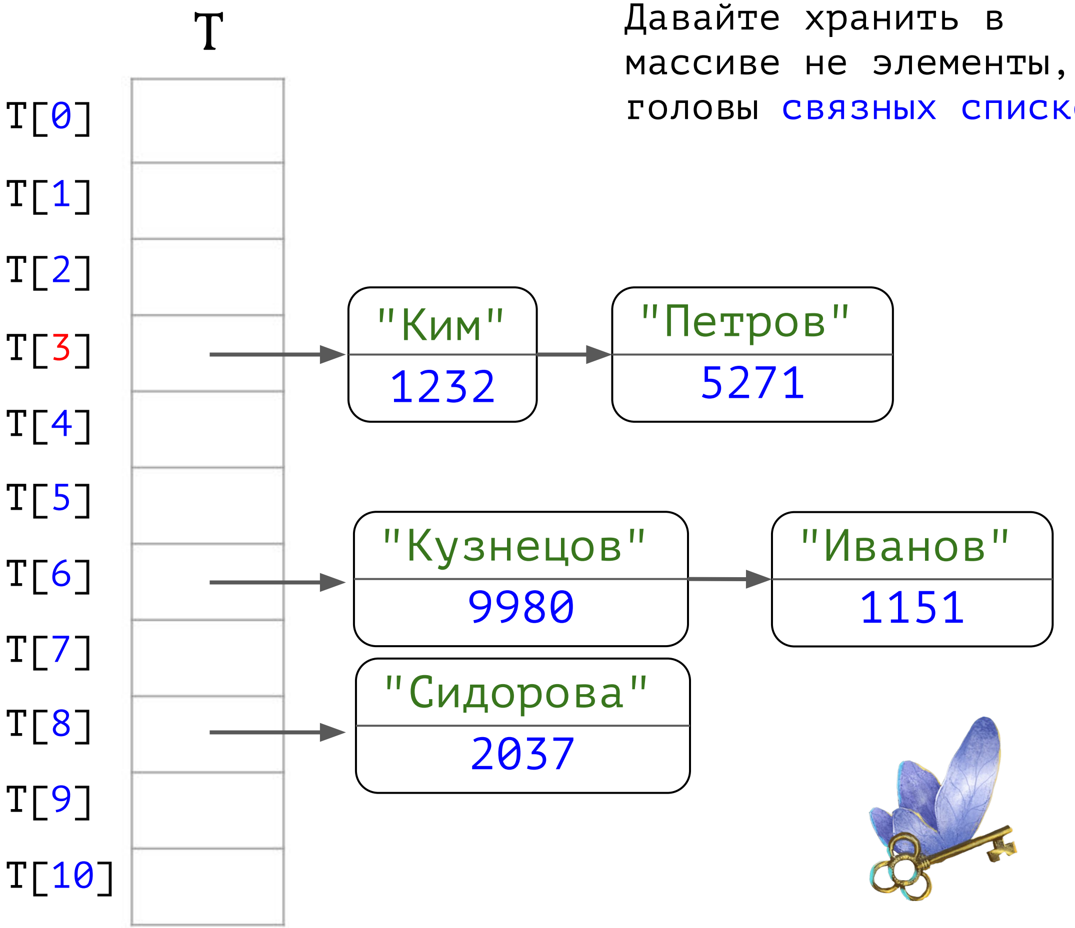
U - строки, фамилии

```
insert("Ким", 1232)
h("Ким") = 3
```

Произошла коллизия!

Добавляем в голову связного списка из соответствующей ячейки.

Давайте хранить в массиве не элементы, а головы связных списков



Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Все очень хорошо со вставкой, там сложность ...

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Все очень хорошо со вставкой, там сложность O(1)!

Хеш-таблицы: метод цепочек

Давайте хранить в массиве не элементы, а головы связных списков

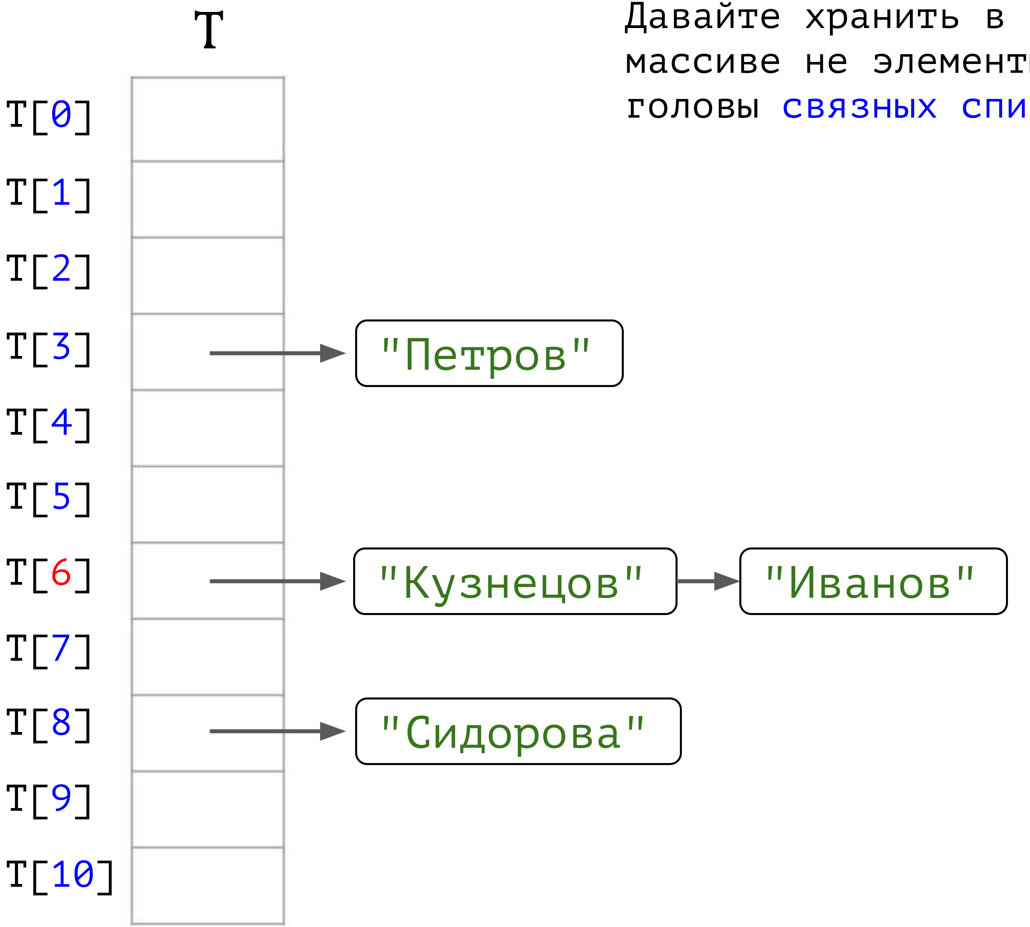
U

U - строки, фамилии

```
insert("Кузнецов")
h("Кузнецов") = 6
```

Произошла коллизия!

Добавляем в голову связного списка из соответствующей ячейки.



Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Все очень хорошо со вставкой, там сложность O(1)!

Т.е. договорились, что хеш-функция работает за O(1), добавление в голову списка - тоже O(1)

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Соображения (про поиск и удаление):

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Соображения (про поиск и удаление):

1. Все сильно зависит от хеш-функции h(x)

   Пусть $h : U \to \{0, 1, 2, \ldots, m-1\}$: $\forall x \in U \Rightarrow h(x) = 0$

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Соображения (про поиск и удаление):

1. Все сильно зависит от хеш-функции h(x)

   Пусть $h : U \to \{0, 1, 2, \dots, m-1\}$: $\forall x \in U \Rightarrow h(x) = 0$

   Тогда шанс коллизии - 100% => таблица вырождается в связный список, а сложность операций - O(N)

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Соображения (про поиск и удаление):

1. Все сильно зависит от хеш-функции $h(x)$
2. Даже если хеш-функция минимизирует количество коллизий, не забываем, что $|U| \geqslant m$

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Соображения (про поиск и удаление):

1. Все сильно зависит от хеш-функции h(x)
2. Даже если хеш-функция минимизирует количество коллизий, не забываем, что $|U| \gg m$

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Соображения (про поиск и удаление):

1. Все сильно зависит от хеш-функции h(x)

2. Даже если хеш-функция минимизирует количество коллизий, не забываем, что $|U| \gg m$

   Это значит, что всегда есть хотя бы один bucket (элемент массива), в который хешируются как минимум $\frac{|U|}{m}$ элементов.

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Соображения (про поиск и удаление):

1. Все сильно зависит от хеш-функции h(x)

2. Даже если хеш-функция минимизирует количество коллизий, не забываем, что $|U| \gg m$

   Это значит, что всегда есть хотя бы один bucket (элемент массива), в который хешируются как минимум $\frac{|U|}{m}$ элементов.

   Если мы начнем добавлять именно такие элементы, то как бы ни была хороша хеш-функция, мы получим множество коллизий => сложность в худшем O(N).

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Соображения (про поиск и удаление):

1. Все сильно зависит от хеш-функции h(x)

2. Даже если хеш-функция минимизирует количество коллизий, не забываем, что $|U| \gg m$

   Это значит, что всегда есть хотя бы один bucket (элемент массива), в который хешируются как минимум $\frac{|U|}{m}$ элементов.

   Если мы начнем добавлять именно такие элементы, то как бы ни была хороша хеш-функция, мы получим множество коллизий => сложность в худшем O(N) (pathological dataset)

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Соображения (про поиск и удаление):

1. Все сильно зависит от хеш-функции h(x)

2. Даже если хеш-функция минимизирует количество коллизий, не забываем, что $|U| \gg m$

3. Еще все сильно зависит от количество buckets, т.е. размера массива. Чем больше ячеек, тем все лучше будет работать.

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Соображения (про поиск и удаление):

1. Все сильно зависит от хеш-функции h(x)
2. Даже если хеш-функция минимизирует количество коллизий, не забываем, что $|U| \gg m$
3. Еще все сильно зависит от количество buckets, т.е. размера массива. Чем больше ячеек, тем все лучше будет работать.

   (в крайнем случае мы таки получили количество элементов массива равное |U|, тогда имеем худший случай O(1), а такая ситуация называется идеальным хешированием).

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Соображения (про поиск и удаление):

1. Все сильно зависит от хеш-функции $h(x)$
2. Даже если хеш-функция минимизирует количество коллизий, не забываем, что $|U| \gg m$
3. Еще все сильно зависит от количество buckets, т.е. размера массива. Чем больше ячеек, тем все лучше будет работать.

   (в крайнем случае мы таки получили количество элементов массива равное $|U|$, тогда имеем худший случай $O(1)$, а такая ситуация называется идеальным хешированием).

Хеш-таблицы: метод цепочек

Вопрос: а что со сложностью операций в такой структуре данных?

Соображения (про поиск и удаление):

Раз со сложностью в худшем и так все понятно (и плохо), то давайте оценим время работы в среднем.

Хеш-таблицы: метод цепочек

Пусть хеш-функция h распределяет ключи по ячейкам равномерно и независимо. Назовем свойство простым равномерным хешированием.

Хеш-таблицы: метод цепочек

Пусть хеш-функция h распределяет ключи по ячейкам равномерно и независимо. Назовем свойство простым равномерным хешированием.

Введем обозначения:

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Хеш-таблицы: метод цепочек

Пусть хеш-функция h распределяет ключи по ячейкам равномерно и независимо. Назовем свойство простым равномерным хешированием.

Введем обозначения:

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] =$ **?**

Хеш-таблицы: метод цепочек

Пусть хеш-функция h распределяет ключи по ячейкам равномерно и независимо. Назовем свойство простым равномерным хешированием.

Введем обозначения:

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] =$?

Линейность мат. ожидания

Пример #12: (балансировка нагрузки) пусть есть N серверов и N процессов. Нужно как-то распределить процессы по серверам.

Ленивое решение: каждый процесс будем отправлять на случайный сервер

Насколько это будет плохо? Какое ожидаемое количество процессов на каждом сервере?

Линейность мат. ожидания

Ленивое решение: каждый процесс будем отправлять на случайный сервер

$$\Omega = \{\text{варианты назначений}\}$$

Линейность мат. ожидания

Ленивое решение: каждый процесс будем отправлять на случайный сервер

$$\Omega = \{\text{варианты назначений}\};\ |\Omega| = N^N; \forall i \in \Omega : p(i) = \frac{1}{N^N}$$

Линейность мат. ожидания

Ленивое решение: каждый процесс будем отправлять на случайный сервер

$$\Omega = \{\text{варианты назначений}\};\ |\Omega| = N^N; \forall i \in \Omega : p(i) = \frac{1}{N^N}$$

Пусть Y – количество процессов на первом* сервере

т.к. рассуждение для всех серверов одинаковые, то можем взять любой

Линейность мат. ожидания

Ленивое решение: каждый процесс будем отправлять на случайный сервер

$\Omega = \{\text{варианты назначений}\};\ |\Omega| = N^N; \forall i \in \Omega : p(i) = \frac{1}{N^N}$

Пусть $Y-$ количество процессов на первом* сервере

$\mathbb{E}[Y] = ?$

т.к. рассуждение для всех серверов одинаковые, то можем взять любой

Линейность мат. ожидания

Ленивое решение: каждый процесс будем отправлять на случайный сервер

$$\Omega = \{\text{варианты назначений}\};\ |\Omega| = N^N; \forall i \in \Omega : p(i) = \frac{1}{N^N}$$

Пусть Y – количество процессов на первом* сервере

$$\mathbb{E}[Y] = ?$$

Пусть $X_j = \begin{cases} 1, \text{если процесс } j \text{ на первом сервере} \\ 0, \text{ иначе} \end{cases}$

т.к. рассуждение для всех серверов одинаковые, то можем взять любой

Линейность мат. ожидания

Ленивое решение: каждый процесс будем отправлять на случайный сервер

$$\Omega = \{\text{варианты назначений}\};\ |\Omega| = N^N; \forall i \in \Omega : p(i) = \frac{1}{N^N}$$

Пусть $Y-$ количество процессов на первом* сервере

$\mathbb{E}[Y] =?$

Пусть $X_j = \begin{cases} 1, \text{если процесс } j \text{ на первом сервере} \\ 0, \ \text{иначе} \end{cases}$

такие случайные величины называют индикаторными и не редко используют в доказательствах

---

* т.к. рассуждение для всех серверов одинаковые, то можем взять любой

Линейность мат. ожидания

Ленивое решение: каждый процесс будем отправлять на **случайный** сервер

Пусть Y – количество процессов на первом* сервере

$\mathbb{E}[Y] = ?$

Пусть $X_j = \begin{cases} 1, \text{если процесс } j \text{ на первом сервере} \\ 0, \text{ иначе} \end{cases}$

Заметим, что $Y = \sum_{j=1}^{N} X_j$.

Линейность мат. ожидания

Ленивое решение: каждый процесс будем отправлять на случайный сервер

Пусть Y – количество процессов на первом* сервере

$\mathbb{E}[Y] = ?$

Пусть $X_j = \begin{cases} 1, \text{если процесс } j \text{ на первом сервере} \\ 0, \text{ иначе} \end{cases}$

Заметим, что $Y = \sum_{j=1}^{N} X_j$. Тогда $\mathbb{E}[Y] = \mathbb{E}[\sum_{j=1}^{N} X_j] = \sum_{j=1}^{N} \mathbb{E}[X_j]$

Линейность мат. ожидания

Ленивое решение: каждый процесс будем отправлять на случайный сервер

Пусть Y – количество процессов на первом* сервере

$\mathbb{E}[Y] = ?$

Пусть $X_j = \begin{cases} 1, \text{если процесс } j \text{ на первом сервере} \\ 0, \text{ иначе} \end{cases}$

Заметим, что $Y = \sum_{j=1}^{N} X_j$. Тогда $\mathbb{E}[Y] = \mathbb{E}[\sum_{j=1}^{N} X_j] = \sum_{j=1}^{N} \mathbb{E}[X_j]$

$$= \sum_{j=1}^{N} (Pr[X_j = 0] * 0 + Pr[X_j = 1] * 1)$$

Линейность мат. ожидания

Ленивое решение: каждый процесс будем отправлять на **случайный** сервер

Пусть Y – количество процессов на первом* сервере

$\mathbb{E}[Y] = ?$

Пусть $X_j = \begin{cases} 1, \text{если процесс } j \text{ на первом сервере} \\ 0, \text{ иначе} \end{cases}$

Заметим, что $Y = \sum_{j=1}^{N} X_j$. Тогда $\mathbb{E}[Y] = \mathbb{E}[\sum_{j=1}^{N} X_j] = \sum_{j=1}^{N} \mathbb{E}[X_j]$

$$= \sum_{j=1}^{N} (\cancel{Pr[X_j = 0] * 0} + \underbrace{\boxed{Pr[X_j = 1]}}_{\frac{1}{N}} * 1)$$

Линейность мат. ожидания

Ленивое решение: каждый процесс будем отправлять на случайный сервер

Пусть Y – количество процессов на первом* сервере

$\mathbb{E}[Y] = ?$

Пусть $X_j = \begin{cases} 1, \text{если процесс } j \text{ на первом сервере} \\ 0, \text{ иначе} \end{cases}$

$\frac{1}{N}$

Заметим, что $Y = \sum_{j=1}^{N} X_j$. Тогда $\mathbb{E}[Y] = \mathbb{E}[\sum_{j=1}^{N} X_j] = \sum_{j=1}^{N} \mathbb{E}[X_j]$

$$= \sum_{j=1}^{N} (\cancel{Pr[X_j = 0] * 0} + \boxed{Pr[X_j = 1]} * 1)$$

$$= \sum_{j=1}^{N} \frac{1}{N} = 1$$

Линейность мат. ожидания

Ленивое решение: каждый процесс будем отправлять на случайный сервер

Пусть Y – количество процессов на первом* сервере

$\mathbb{E}[Y] = ?$

Пусть $X_j = \begin{cases} 1, \text{если процесс } j \text{ на первом сервере} \\ 0, \text{ иначе} \end{cases}$

Заметим, что $Y = \sum_{j=1}^{N} X_j$. Тогда $\mathbb{E}[Y] = \mathbb{E}[\sum_{j=1}^{N} X_j] = \sum_{j=1}^{N} \mathbb{E}[X_j]$

$$= \sum_{j=1}^{N} (\cancel{Pr[X_j = 0] * 0} + \underbrace{\boxed{Pr[X_j = 1]}}_{\frac{1}{N}} * 1)$$

$$= \sum_{j=1}^{N} \frac{1}{N} = 1$$

Получается, что не такой уж и плохой алгоритм! В среднем все сбалансированно 😊

Линейность мат. ожидания

Ленивое решение: каждый ~~процесс~~ элемент будем отправлять ~~на~~ в ~~случайный сервер~~ некоторый bucket с равной вероятностью

Пусть Y – количество процессов на первом* сервере

$\mathbb{E}[Y] = ?$

Пусть $X_j = \begin{cases} 1, \text{если процесс } j \text{ на первом сервере} \\ 0, \text{ иначе} \end{cases}$

Заметим, что $Y = \sum_{j=1}^{N} X_j$. Тогда $\mathbb{E}[Y] = \mathbb{E}[\sum_{j=1}^{N} X_j] = \sum_{j=1}^{N} \mathbb{E}[X_j]$

$$= \sum_{j=1}^{N} (\cancel{Pr[X_j = 0] * 0} + \boxed{Pr[X_j = 1]} * 1)$$

$\frac{1}{N}$

$$= \sum_{j=1}^{N} \frac{1}{N} = 1$$

Получается, что не такой уж и плохой алгоритм! В среднем все сбалансированно 😊

Линейность мат. ожидания

Ленивое решение: каждый ~~процесс~~ элемент будем отправлять ~~на~~ в ~~случайный сервер~~ некоторый bucket с равной вероятностью

Пусть Y – количество ~~процессов на первом* сервере~~ элементов в первой цепочке

$\mathbb{E}[Y] = ?$

Пусть $X_j = \begin{cases} 1, \text{если } \cancel{\text{процесс } j \text{ на первом сервере}} \text{ j-ый элемент в первой цепочке} \\ 0, \text{ иначе} \end{cases}$

Заметим, что $Y = \sum_{j=1}^{N} X_j$. Тогда $\mathbb{E}[Y] = \mathbb{E}[\sum_{j=1}^{N} X_j] = \sum_{j=1}^{N} \mathbb{E}[X_j]$

$$= \sum_{j=1}^{N} (\cancel{Pr[X_j = 0] * 0} + \boxed{Pr[X_j = 1]} * 1)$$

$$\frac{1}{N}$$

$$= \sum_{j=1}^{N} \frac{1}{N} = 1$$

Получается, что не такой уж и плохой алгоритм! В среднем все сбалансированно 😊

Линейность мат. ожидания

Ленивое решение: каждый ~~процесс~~ элемент будем отправлять ~~на случайный сервер~~ в некоторый bucket с равной вероятностью

Пусть Y – количество ~~процессов на первом* сервере~~ элементов в первой цепочке

$\mathbb{E}[Y] = ?$

Пусть $X_j = \begin{cases} 1, \text{если } \cancel{\text{процесс } j \text{ на первом сервере}} \text{ j-ый элемент в первой цепочке} \\ 0, \text{ иначе} \end{cases}$

Заметим, что $Y = \sum_{j=1}^{N} X_j$. Тогда $\mathbb{E}[Y] = \mathbb{E}[\sum_{j=1}^{N} X_j] = \sum_{j=1}^{N} \mathbb{E}[X_j]$

$$= \sum_{j=1}^{N} (\cancel{Pr[X_j = 0] * 0} + \boxed{Pr[X_j = 1]} * 1)$$

$\frac{1}{M}$

$$= \sum_{j=1}^{N} \frac{1}{M} = \frac{N}{M}$$

Хеш-таблицы: метод цепочек

Пусть хеш-функция h распределяет ключи по ячейкам равномерно и независимо. Назовем свойство простым равномерным хешированием.

Введем обозначения:

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] =$ **?**

Хеш-таблицы: метод цепочек

Пусть хеш-функция h распределяет ключи по ячейкам равномерно и независимо. Назовем свойство простым равномерным хешированием.

Введем обозначения:

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \frac{n}{m} = \alpha$

Хеш-таблицы: метод цепочек

Пусть хеш-функция h распределяет ключи по ячейкам равномерно и независимо. Назовем свойство простым равномерным хешированием.

Введем обозначения:

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \frac{n}{m} = \alpha$

Теперь выразим через α время работы в среднем операций поиска и удаления элемента. Рассмотрим два случая: удачный поиск и неудачный.

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \frac{n}{m} = \alpha$

Неудачный поиск: ищем элемент, которого в таблице нет.

h(key) отправляет в некоторый bucket, после чего перебираем все элементы в соответствующем связном списке.

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \frac{n}{m} = \alpha$

Неудачный поиск: ищем элемент, которого в таблице нет.

h(key) отправляет в некоторый bucket, после чего перебираем все элементы в соответствующем связном списке.

Тогда среднее время работы будет: время на хеширование O(1) + средняя длина цепочки $E[n_j] = \alpha$.

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \frac{n}{m} = \alpha$

Неудачный поиск: ищем элемент, которого в таблице нет.

h(key) отправляет в некоторый bucket, после чего перебираем все элементы в соответствующем связном списке.

Тогда среднее время работы будет: время на хеширование O(1) + средняя длина цепочки $E[n_j] = \alpha$.

Т.е. среднее время работы поиска и удаления: $O(1 + \alpha)$

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \frac{n}{m} = \alpha$

Неудачный поиск: ищем элемент, которого в таблице нет.

Т.е. среднее время работы поиска и удаления: $O(1 + \alpha)$

Важно учитывать оба слагаемых, т.к. во время своей работы структура данных может и будет меняться: элементы будут добавляться и удаляться, количество бакетов m тоже может меняться!

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \frac{n}{m} = \alpha$

Неудачный поиск: ищем элемент, которого в таблице нет.

Т.е. среднее время работы поиска и удаления: $O(1 + \alpha)$

Важно учитывать оба слагаемых, т.к. во время своей работы структура данных может и будет меняться: элементы будут добавляться и удаляться, количество бакетов m тоже может меняться!

В зависимости n/m вклад второго слагаемого влияет или нет.

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \frac{n}{m} = \alpha$

Удачный поиск: ищем элемент, который в таблице есть.

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \frac{n}{m} = \alpha$

Удачный поиск: ищем элемент, который в таблице есть.

Это значит, а) что элемент находится в каком-то из списков, б) все элементы в списке до него добавлены позже (т.к. добавляем в голову)

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \frac{n}{m} = \alpha$

Удачный поиск: ищем элемент, который в таблице есть.

Это значит, а) что элемент находится в каком-то из списков, б) все элементы в списке до него добавлены позже (т.к. добавляем в голову)

Нужно оценить количество элементов, добавленных позже того, который ищем (в среднем).

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Это значит, а) что элемент находится в каком-то из списков, б) все элементы в списке до него добавлены позже (т.к. добавляем в голову)

Нужно оценить количество элементов, добавленных позже того, который ищем (в среднем).

Пусть x_i - i-ый добавленный по порядку элемент, $i \in (1, 2, 3, \ldots, n)$

Введем индикаторную случайную величину: $X_{ij} = \begin{cases} 1, \text{если } h(x_i) = h(x_j) \\ 0, \text{иначе} \end{cases}$

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Это значит, а) что элемент находится в каком-то из списков, б) все элементы в списке до него добавлены позже (т.к. добавляем в голову)

Нужно оценить количество элементов, добавленных позже того, который ищем (в среднем).

Пусть x_i - i-ый добавленный по порядку элемент, $i \in (1, 2, 3, \ldots, n)$

Введем индикаторную случайную величину: $X_{ij} = \begin{cases} 1, \text{если } h(x_i) = h(x_j) \\ 0, \text{иначе} \end{cases}$

В предположении равномерного хеширования:

$$Pr\{h(x_i) = h(x_j)\} = \frac{1}{m} \Rightarrow E[X_{ij}] = \frac{1}{m}$$

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Посчитаем мат. ожидание количества проверяемых элементов.

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Посчитаем мат. ожидание количества проверяемых элементов:

Разобьем это на два вида случайных событий: сначала зафиксируем уже построенную таблицу и получим ожидаемое время поиска при выборе одного конкретного элемента

Немного теории вероятностей #4

Пусть $X : \Omega \to R$ – случайная величина

Тогда: математическое ожидание случайной величины X —

$$\mathbb{E}[X] = \sum_{i \in \Omega} X(i) * p(i)$$

Физический смысл - среднее значение случайной величины; то, что мы "скорее всего" получим

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Посчитаем мат. ожидание количества проверяемых элементов:

$$\frac{1}{n}\sum_{i=1}^{n}(\qquad\qquad\qquad\qquad)$$

Разобьем это на два вида случайных событий: сначала зафиксируем уже построенную таблицу и получим ожидаемое время поиска при выборе одного конкретного элемента

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Посчитаем мат. ожидание количества проверяемых элементов:

$$\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})$$

Разобьем это на два вида случайных событий: сначала зафиксируем уже построенную таблицу и получим ожидаемое время поиска при выборе одного конкретного элемента, считаем длину цепочки.

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Посчитаем мат. ожидание количества проверяемых элементов:

$$\frac{1}{n}\sum_{i=1}^{n}(1 + \sum_{j=i+1}^{n} X_{ij})$$

Теперь вспоминаем, что при построении всей таблицы тоже были выборы бакетов => это тоже результат случайных событий

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Посчитаем мат. ожидание количества проверяемых элементов:

$$E[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})] =$$

Теперь вспоминаем, что при построении всей таблицы тоже были выборы бакетов => это тоже результат случайных событий. Тогда снова считаем мат.ожидание

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Посчитаем мат. ожидание количества проверяемых элементов:

$$E[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})] = \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E[X_{ij}]) =$$

По линейности
мат. ожидания

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Посчитаем мат. ожидание количества проверяемых элементов:

$$E[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})] = \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E[X_{ij}]) =$$

$$\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m}) =$$

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Посчитаем мат. ожидание количества проверяемых элементов:

$$E[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})] = \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E[X_{ij}]) =$$

$$\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m}) = 1+\frac{1}{nm}\sum_{i=1}^{n}(\sum_{j=i+1}^{n}1) = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) =$$

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Посчитаем мат. ожидание количества проверяемых элементов:

$$E[\tfrac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})] = \tfrac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E[X_{ij}]) =$$

$$\tfrac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\tfrac{1}{m}) = 1+\tfrac{1}{nm}\sum_{i=1}^{n}(\sum_{j=i+1}^{n}1) = 1+\tfrac{1}{nm}\sum_{i=1}^{n}(n-i) =$$

$$1+\tfrac{1}{nm}(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i) = 1+\tfrac{1}{nm}(n^2-\tfrac{n*(n+1)}{2}) = 1+\tfrac{n-1}{2m} =$$

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Посчитаем мат. ожидание количества проверяемых элементов:

$$E[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})] = \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E[X_{ij}]) =$$

$$\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m}) = 1+\frac{1}{nm}\sum_{i=1}^{n}(\sum_{j=i+1}^{n}1) = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) =$$

$$1+\frac{1}{nm}(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i) = 1+\frac{1}{nm}(n^2-\frac{n*(n+1)}{2}) = 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

Хеш-таблицы: метод цепочек

Удачный поиск: ищем элемент, который в таблице есть.

Посчитаем мат. ожидание количества проверяемых элементов:

$$E[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})] = \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E[X_{ij}]) =$$

$$\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m}) = 1+\frac{1}{nm}\sum_{i=1}^{n}(\sum_{j=i+1}^{n}1) = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) =$$

$$1+\frac{1}{nm}(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i) = 1+\frac{1}{nm}(n^2-\frac{n*(n+1)}{2}) = 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

Не забудем добавить 1 за вычисление хеш-функции и получаем среднее время работы:

$$O(2+\frac{\alpha}{2}-\frac{\alpha}{2n}) = O(1+\alpha) \quad \square$$

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \boxed{\frac{n}{m} = \alpha}$

Сложность операций:

1. Добавление -> $O(1)$
2. Поиск -> $O(1 + \alpha)$ в среднем
3. Удаление -> $O(1 + \alpha)$ в среднем

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \boxed{\frac{n}{m} = \alpha}$

Сложность операций:

1. Добавление -> $O(1)$
2. Поиск -> $O(1 + \alpha)$ в среднем
3. Удаление -> $O(1 + \alpha)$ в среднем

Тогда, если хотя бы:

$n = O(m)$

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \boxed{\frac{n}{m} = \alpha}$

Сложность операций:

1. Добавление -> $O(1)$
2. Поиск -> $O(1+\alpha)$ в среднем
3. Удаление -> $O(1+\alpha)$ в среднем

Тогда, если хотя бы:

$$n = O(m) \Rightarrow \alpha = \frac{O(m)}{m} = O(1)$$

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \boxed{\frac{n}{m} = \alpha}$

Сложность операций:

1. Добавление -> $O(1)$
2. Поиск -> $O(1)$ в среднем
3. Удаление -> $O(1)$ в среднем

Тогда, если хотя бы:

$$n = O(m) \Rightarrow \alpha = \frac{O(m)}{m} = O(1)$$

А все оценки становятся константными.

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \boxed{\frac{n}{m} = \alpha}$

Сложность операций:

1. Добавление -> $O(1)$
2. Поиск -> $O(1)$ в среднем
3. Удаление -> $O(1)$ в среднем

Тогда, если хотя бы:

$$n = O(m) \Rightarrow \alpha = \frac{O(m)}{m} = O(1)$$

А все оценки становятся константными.

Вывод?

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \boxed{\frac{n}{m} = \alpha}$

Сложность операций:

1. Добавление -> $O(1)$
2. Поиск -> $O(1)$ в среднем
3. Удаление -> $O(1)$ в среднем

Надо поддерживать
такой инвариант!

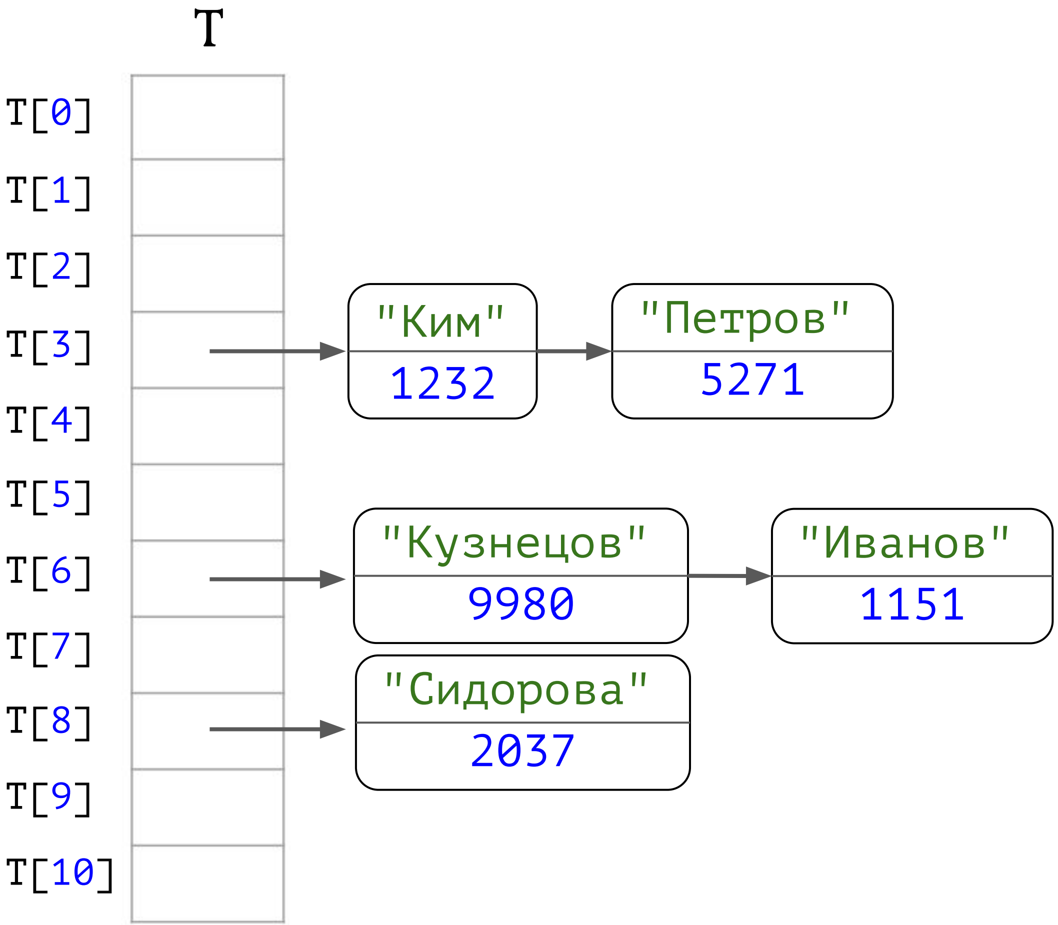
Тогда, если хотя бы:

$$n = O(m) \Rightarrow \alpha = \frac{O(m)}{m} = O(1)$$

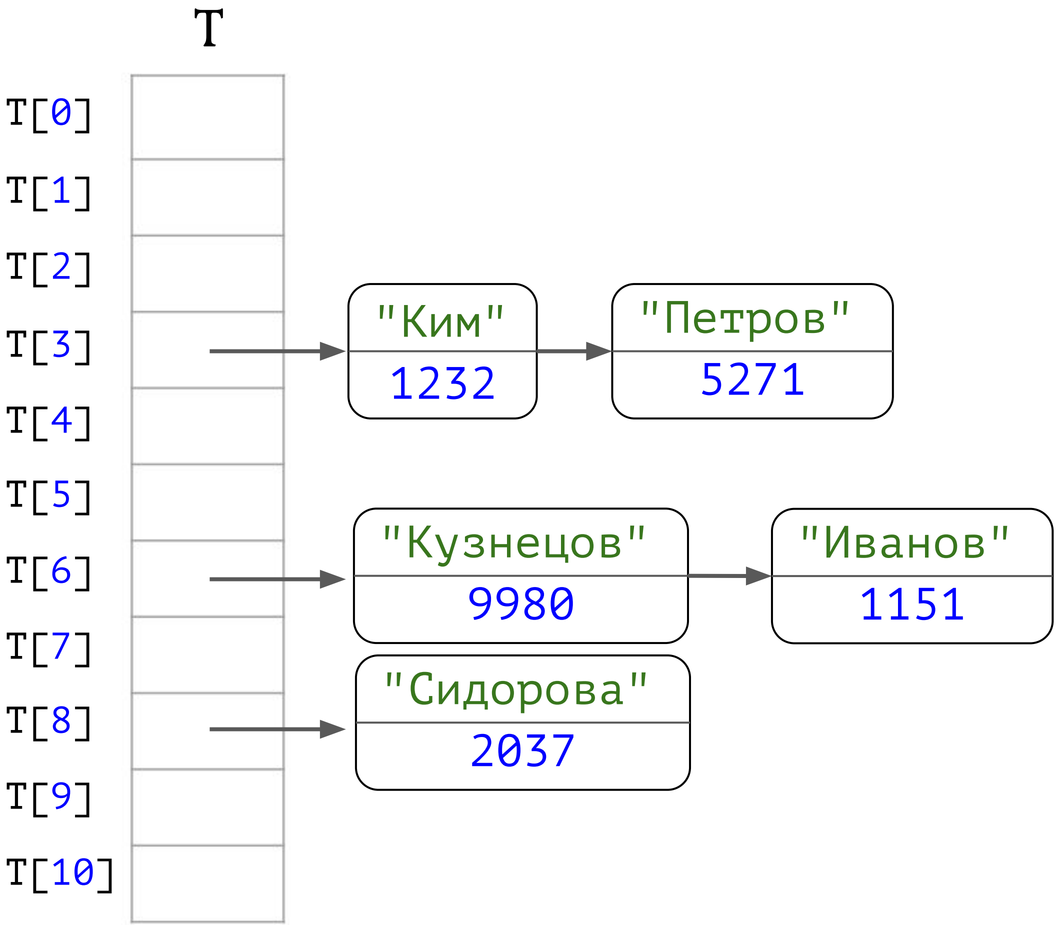
А все оценки становятся константными.

Вывод?

Хеш-таблицы: перехеширование



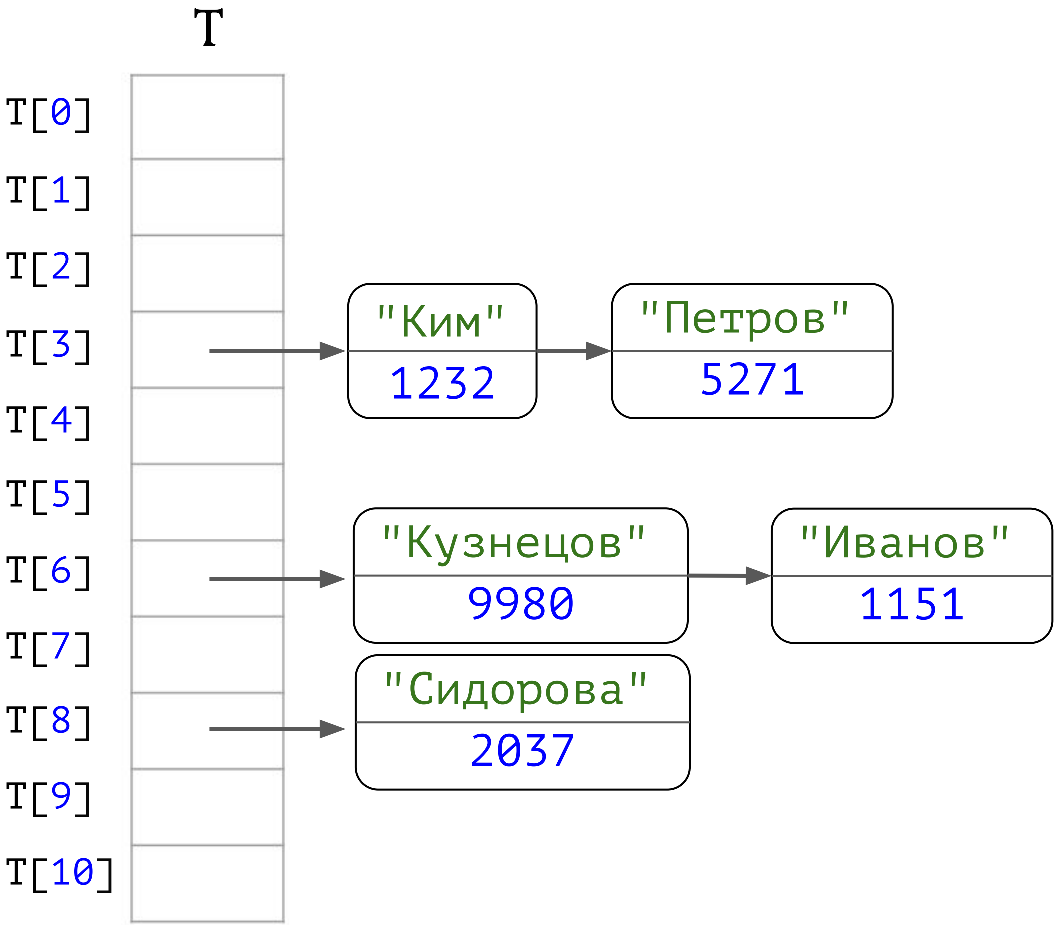
Хеш-таблицы: перехеширование



При добавлении очередного элемента замечаем, что пропорция n:m превысила некоторый порог (load factor).

По умолчанию в некоторых языках - 0.75, но обычно это настраивается.

Хеш-таблицы: перехеширование

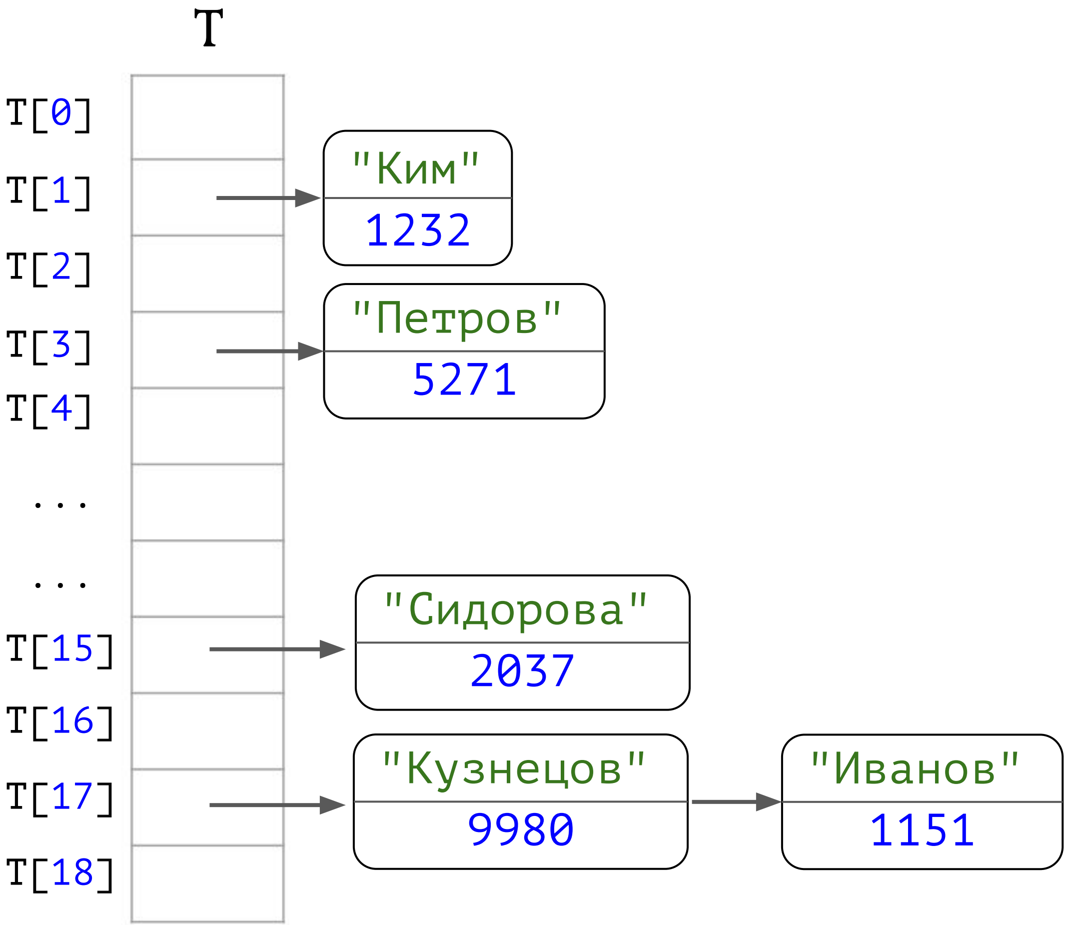


При добавлении очередного элемента замечаем, что пропорция n:m превысила некоторый порог (load factor).

По умолчанию в некоторых языках - 0.75, но обычно это настраивается.

Тогда запускаем процедуру rehash: увеличиваем массив, передобавляем туда старые элементы

Хеш-таблицы: перехеширование

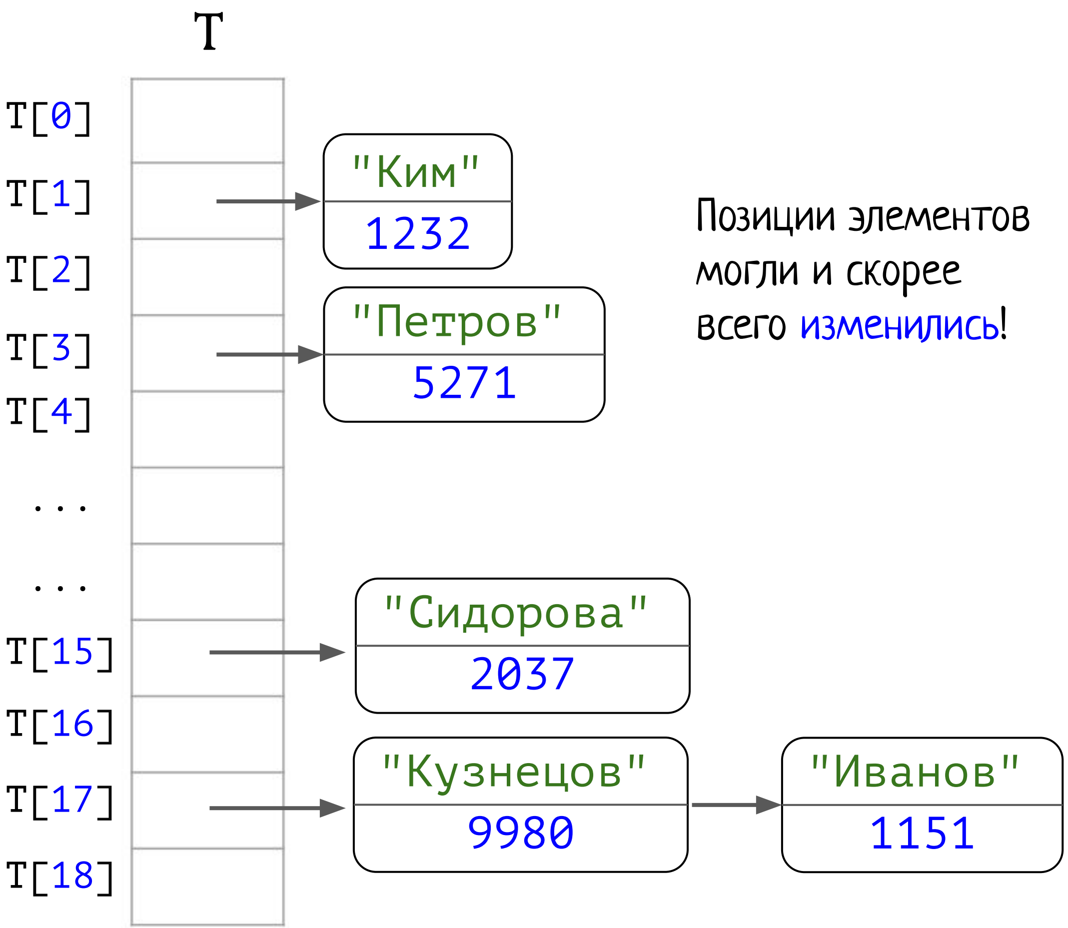


При добавлении очередного элемента замечаем, что пропорция n:m превысила некоторый порог (load factor).

По умолчанию в некоторых языках - 0.75, но обычно это настраивается.

Тогда запускаем процедуру rehash: увеличиваем массив, передобавляем туда старые элементы

Хеш-таблицы: перехеширование



Позиции элементов могли и скорее всего изменились!

При добавлении очередного элемента замечаем, что пропорция n:m превысила некоторый порог (load factor).

По умолчанию в некоторых языках - 0.75, но обычно это настраивается.

Тогда запускаем процедуру rehash: увеличиваем массив, передобавляем туда старые элементы

Хеш-таблицы

…	1	2	3	…	…	…	13	…	…	…	…	42	…	…

Пусть у нас есть массив размера m.
И пусть U - множество ключей, |U| ⩾ m.

Тогда функцию $h : U \to \{0, 1, 2, \ldots, m-1\}$ назовем хеш-функцией.

Обычно:

$$\mathrm{U} \xrightarrow{\text{hashCode}} \mathbb{Z} \xrightarrow{\text{функция сжатия}} \{0, 1, 2, \ldots, m-1\}$$

Хеш-таблицы

…	1	2	3	…	…	…	13	…	…	…	…	42	…	…

Пусть у нас есть массив размера m.
И пусть U - множество ключей, |U| ⩾ m.

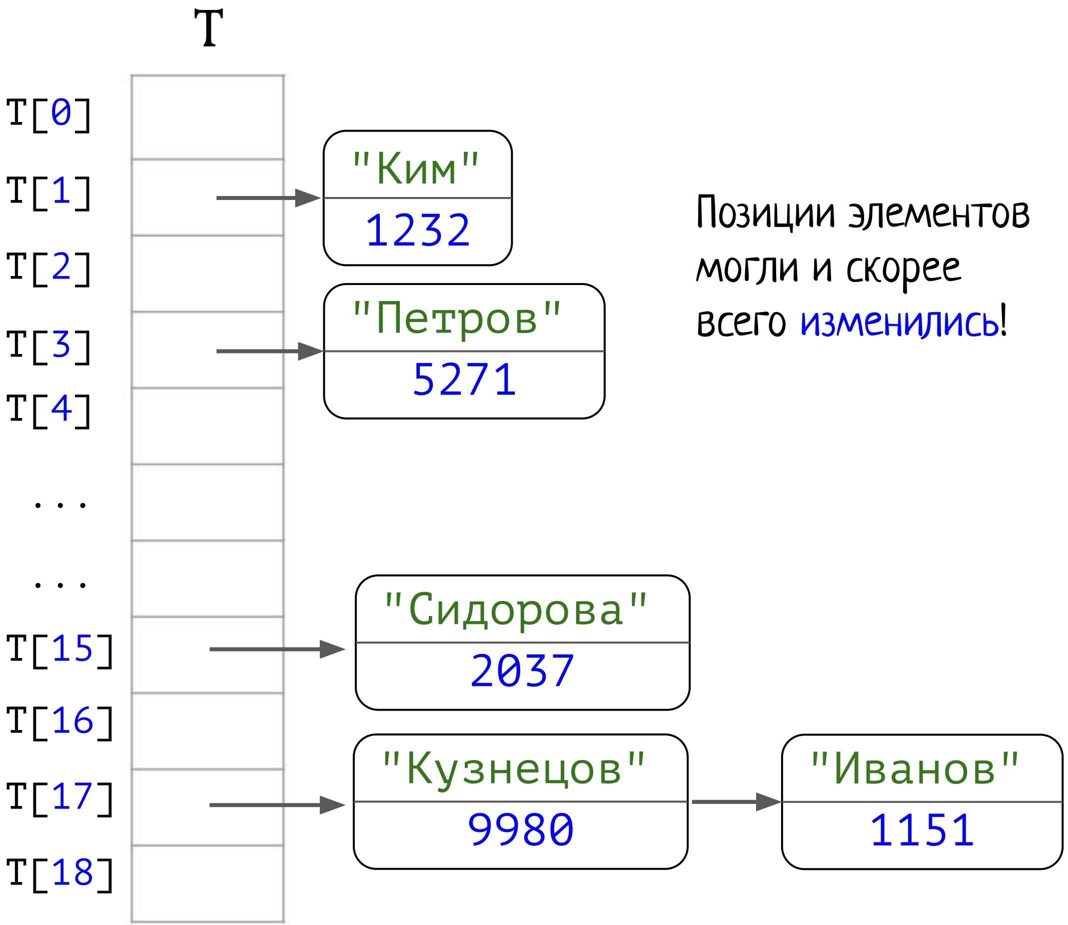
Тогда функцию $h: U \to \{0, 1, 2, \ldots, m-1\}$ назовем хеш-функцией.

Раз m изменилось, то и позиция может изменится

Обычно:

$$\boxed{\mathrm{U}} \xrightarrow{\text{hashCode}} \boxed{\mathbb{Z}} \xrightarrow{\text{функция сжатия}} \boxed{\{0, 1, 2, \ldots, m-1\}}$$

Хеш-таблицы: перехеширование



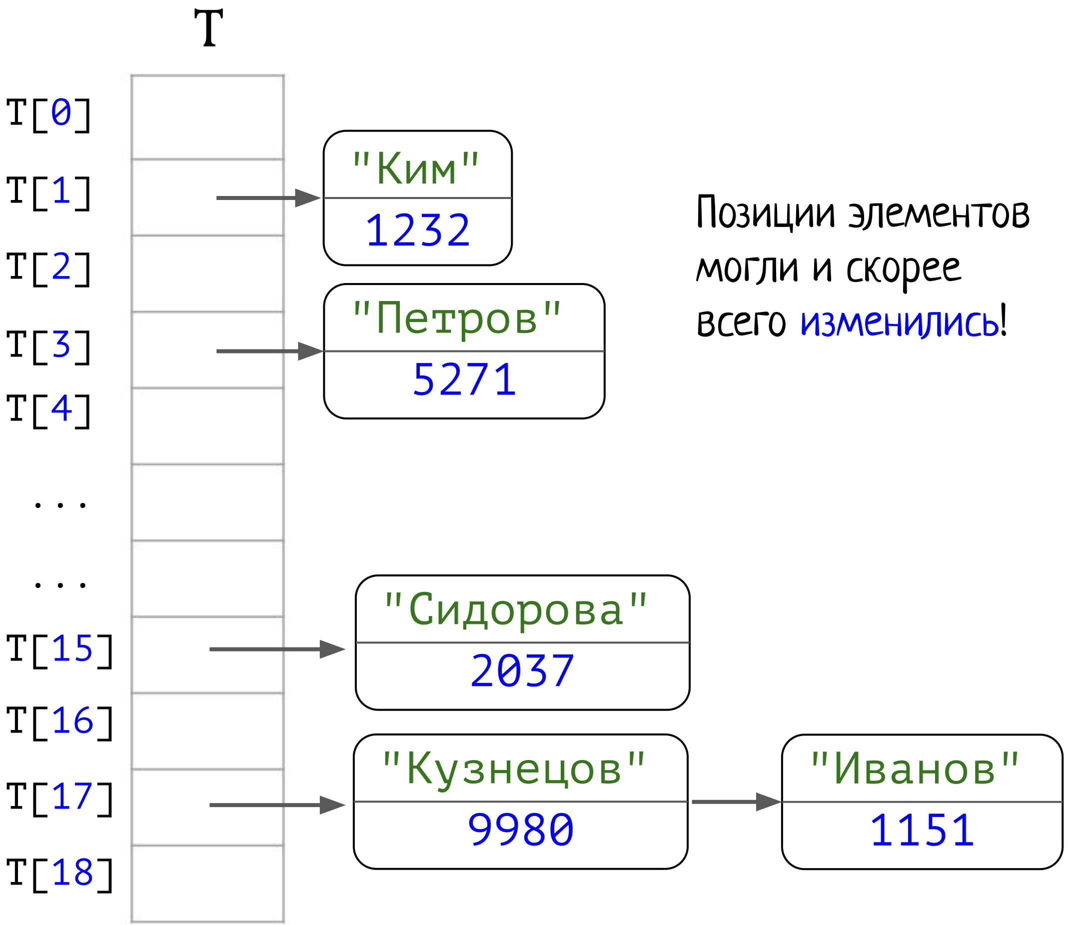
Позиции элементов могли и скорее всего изменились!

При добавлении очередного элемента замечаем, что пропорция n:m превысила некоторый порог (load factor).

Тогда запускаем процедуру rehash: увеличиваем массив, передобавляем туда старые элементы

Как это повлияет на сложность?

Хеш-таблицы: перехеширование



Позиции элементов могли и скорее всего изменились!

При добавлении очередного элемента замечаем, что пропорция n:m превысила некоторый порог (load factor).

Тогда запускаем процедуру rehash: увеличиваем массив, передобавляем туда старые элементы

Как это повлияет на сложность?

Плохо, но знаем, что делать: считаем амортизационную сложность

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \boxed{\frac{n}{m} = \alpha}$

Сложность операций:

1. Добавление -> $O(1)$
2. Поиск -> $O(1)$ в среднем
3. Удаление -> $O(1)$ в среднем

Тогда, если хотя бы:

$$n = O(m) \Rightarrow \alpha = \frac{O(m)}{m} = O(1)$$

А все оценки становятся константными.

Хеш-таблицы: метод цепочек

n - общее количество элементов в таблице, m - количество buckets.

Тогда $\alpha = \frac{n}{m}$ - коэффициент заполнения таблицы.

Обозначим через n_j длину цепочки из j-ого бакета.
Тогда $n = n_0 + n_1 + \ldots + n_{m-1}$, а чему равно $E[n_j] = \boxed{\frac{n}{m} = \alpha}$

Сложность операций:

1. Добавление -> $O^*(1)$
2. Поиск -> $O(1)$ в среднем
3. Удаление -> $O(1)$ в среднем

Тогда, если хотя бы:

$$n = O(m) \Rightarrow \alpha = \frac{O(m)}{m} = O(1)$$

А все оценки становятся константными.

(добавление - амортизационная)

А как еще можно бороться с коллизиями?

Хеш-таблицы: метод открытой адресации

Хеш-таблицы: метод цепочек

Давайте хранить в массиве не элементы, а головы связных списков

U

U - строки, фамилии

```
insert("Ким")
h("Ким") = 3
```

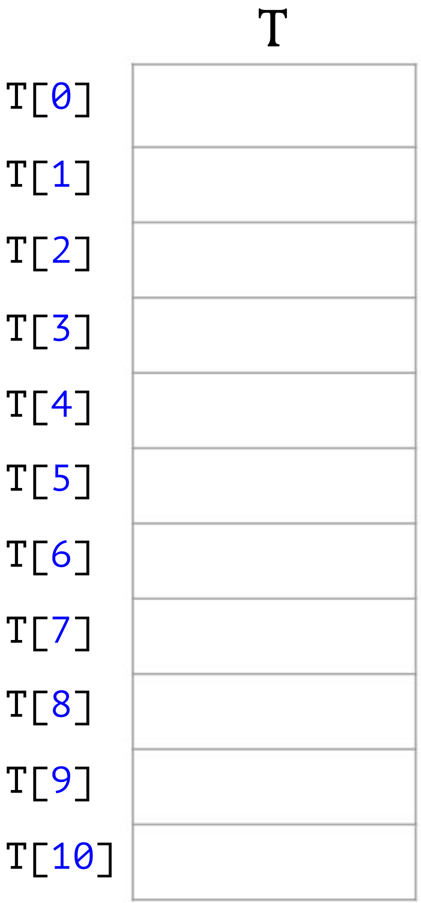
Произошла коллизия!

Добавляем в голову связного списка из соответствующей ячейки.

Хеш-таблицы: метод открытой адресации

Давайте хранить в массиве все-таки именно что элементы

Хеш-таблицы: метод открытой адресации



Давайте хранить в массиве все-таки именно что элементы

Хеш-таблицы: метод открытой адресации

```
insert("Иванов")
h("Иванов") = 6
```

T

T[0]	
T[1]	
T[2]	
T[3]	
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	
T[8]	
T[9]	
T[10]	

Давайте хранить в массиве все-таки именно что элементы

Хеш-таблицы: метод открытой адресации

```
insert("Петров")
h("Петров") = 3
```

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	
T[8]	
T[9]	
T[10]	

Давайте хранить в массиве все-таки именно что элементы

Хеш-таблицы: метод открытой адресации

```
insert("Сидорова")
h("Сидорова") = 8
```

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	
T[8]	"Сидорова"
T[9]	
T[10]	

Давайте хранить в массиве все-таки именно что элементы

Хеш-таблицы: метод открытой адресации

```
insert("Кузнецов")
h("Кузнецов") = 6
```

Коллизия!

Что делать?

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	
T[8]	"Сидорова"
T[9]	
T[10]	

Давайте хранить в массиве все-таки именно что элементы

Хеш-таблицы: метод открытой адресации

```
insert("Кузнецов")
h("Кузнецов") = 6
```

Коллизия!

Что делать?

В массиве занято -> ищем следующее свободное место по некоторому правилу.

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	
T[8]	"Сидорова"
T[9]	
T[10]	

Давайте хранить в массиве все-таки именно что элементы

Хеш-таблицы: метод открытой адресации

```
insert("Кузнецов")
h("Кузнецов") = 6
```

Коллизия!

Что делать?

В массиве занято -> ищем следующее свободное место по некоторому правилу.

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	
T[8]	"Сидорова"
T[9]	
T[10]	

Давайте хранить в массиве все-таки именно что элементы

Линейное исследование: ищем просто следующую пустую ячейку.

(есть и другие варианты)

Хеш-таблицы: метод открытой адресации

```
insert("Кузнецов")
h("Кузнецов") = 6
```

Коллизия!

Что делать?

В массиве занято -> ищем следующее свободное место по некоторому правилу.

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	
T[10]	

Давайте хранить в массиве все-таки именно что элементы

Линейное исследование: ищем просто следующую пустую ячейку.

(есть и другие варианты)

Хеш-таблицы: метод открытой адресации

```
insert("Ким")
h("Ким") = 7
```

Коллизии в старом понимании нет, ведь ни для какого ключа функция h не возвращала 7

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	
T[10]	

Давайте хранить в массиве все-таки именно что элементы

Линейное исследование: ищем просто следующую пустую ячейку.

(есть и другие варианты)

Хеш-таблицы: метод открытой адресации

```
insert("Ким")
h("Ким") = 7
```

Коллизии в старом понимании нет, ведь ни для какого ключа функция h не возвращала 7

Но на деле коллизия есть, т.к. там занято!

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	
T[10]	

Давайте хранить в массиве все-таки именно что элементы

Линейное исследование: ищем просто следующую пустую ячейку.

(есть и другие варианты)

Хеш-таблицы: метод открытой адресации

insert("Ким")
h("Ким") = 7

Коллизии в старом понимании нет, ведь ни для какого ключа функция h не возвращала 7

Но на деле коллизия есть, т.к. там занято!

Снова ищем пустую.

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	
T[10]	

Давайте хранить в массиве все-таки именно что элементы

Линейное исследование: ищем просто следующую пустую ячейку.

(есть и другие варианты)

Хеш-таблицы: метод открытой адресации

```
insert("Ким")
h("Ким") = 7
```

Коллизии в старом понимании нет, ведь ни для какого ключа функция h не возвращала 7

Но на деле коллизия есть, т.к. там занято!

Снова ищем пустую.

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Давайте хранить в массиве все-таки именно что элементы

Линейное исследование: ищем просто следующую пустую ячейку.

(есть и другие варианты)

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Хеш-таблицы: метод открытой адресации

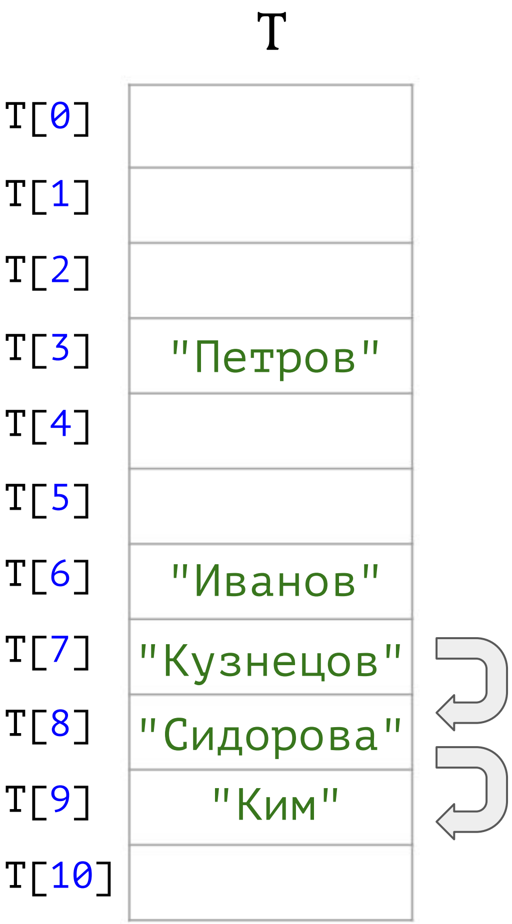
	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Поиск:

1. Ищем с помощью исследования,
2. Проверяем по ключу, нашли или нет

Хеш-таблицы: метод открытой адресации



Как при этом меняются остальные операции?

Поиск:

1. Ищем с помощью исследования,
2. Проверяем по ключу, нашли или нет

```
find("Ким")
h("Ким") = 7
```

Хеш-таблицы: метод открытой адресации

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Поиск:

1. Ищем с помощью исследования,
2. Проверяем по ключу, нашли или нет

```
find("Ким")
h("Ким") = 7
```

Это был пример успешного поиска,
а когда поиск пора признавать неуспешным?

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Поиск:

1. Ищем с помощью исследования,
2. Проверяем по ключу, нашли или нет

```
find("Сергеев")
h("Сергеев") = 7
```

Это был пример успешного поиска,
а когда поиск пора признавать неуспешным?

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Поиск:

1. Ищем с помощью исследования,
2. Проверяем по ключу, нашли или нет

```
find("Сергеев")
h("Сергеев") = 7
```

Это был пример успешного поиска,
а когда поиск пора признавать неуспешным?

Дошли до пустоты, значит не нашли.

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?

```
remove("Сидорова")
h("Сидорова") = 8
```

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?

```
remove("Сидорова")
h("Сидорова") = 8
```

Допустим, удалим, оставим пустоту.

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?

```
remove("Сидорова")
h("Сидорова") = 8
```

Допустим, удалим, оставим пустоту.

Хеш-таблицы: метод открытой адресации

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?

```
remove("Сидорова")
h("Сидорова") = 8
```

Допустим, удалим, оставим пустоту.

```
find("Ким")
h("Ким") = 7
```

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?

```
remove("Сидорова")
h("Сидорова") = 8
```

Допустим, удалим, оставим пустоту.

```
find("Ким")
h("Ким") = 7
```

Дошли до пустоты =>
нет такого элемента!

Хеш-таблицы: метод открытой адресации

T	
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?
   a. Просто удалять нельзя

```
remove("Сидорова")
h("Сидорова") = 8
```

Хеш-таблицы: метод открытой адресации

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	DELETED
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?
   a. Просто удалять нельзя
   b. Можно помечать, как DELETED

```
remove("Сидорова")
h("Сидорова") = 8
```

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	DELETED
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?
    a. Просто удалять нельзя
    b. Можно помечать, как DELETED

    Поиск продолжает искать "сквозь" эту ячейку, а вот вставка может сюда добавить новый элемент.

```
remove("Сидорова")
h("Сидорова") = 8
```

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	DELETED
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?
   a. Просто удалять нельзя
   b. Можно помечать, как DELETED

      Поиск продолжает искать "сквозь" эту ячейку, а вот вставка может сюда добавить новый элемент.

      Плохо повлияет на сложность поиска, подробности позже.

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?
   a. Просто удалять нельзя
   b. Можно помечать, как DELETED

   c. Можно продолжить поиск, все, что найдем до пустоты - перехешировать.

Хеш-таблицы: метод открытой адресации

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Сидорова"
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?
   a. Просто удалять нельзя
   b. Можно помечать, как DELETED

   c. Можно продолжить поиск, все, что найдем до пустоты - перехешировать.

Хеш-таблицы: метод открытой адресации

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?
   a. Просто удалять нельзя
   b. Можно помечать, как DELETED

   c. Можно продолжить поиск, все, что найдем до пустоты - перехешировать.

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	
T[9]	"Ким"
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?
   a. Просто удалять нельзя
   b. Можно помечать, как DELETED

   c. Можно продолжить поиск, все, что найдем до пустоты - перехешировать.

Хеш-таблицы: метод открытой адресации

	T
T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Ким"
T[9]	
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?
    a. Просто удалять нельзя
    b. Можно помечать, как DELETED

    c. Можно продолжить поиск, все, что найдем до пустоты - перехешировать.

Хеш-таблицы: метод открытой адресации

T

T[0]	
T[1]	
T[2]	
T[3]	"Петров"
T[4]	
T[5]	
T[6]	"Иванов"
T[7]	"Кузнецов"
T[8]	"Ким"
T[9]	
T[10]	

Как при этом меняются остальные операции?

Удаление:

1. Ищем с помощью исследования,
2. А что дальше?
   a. Просто удалять нельзя
   b. Можно помечать, как DELETED

   c. Можно продолжить поиск, все, что найдем до пустоты - перехешировать.

   d. Не использовать открытую адресацию с удалением

Хеш-таблицы: метод открытой адресации

А что по поводу сложности операций?

Хеш-таблицы: метод открытой адресации

А что по поводу сложности операций?

Худший случай любой операции понятен: O(N)

Хеш-таблицы: метод открытой адресации

А что по поводу сложности операций?

Худший случай любой операции понятен: O(N)

Утверждение #1: мат. ожидание количества исследований при неуспешном поиске в хеш-таблице с открытой адресацией и коэффициентом заполненности $\alpha = \frac{n}{m} < 1$ в предположении равномерного хеширования не превышает $\frac{1}{1-a}$

Хеш-таблицы: метод открытой адресации

А что по поводу сложности операций?

Худший случай любой операции понятен: O(N)

Утверждение #1: мат. ожидание количества исследований при неуспешном поиске в хеш-таблице с открытой адресацией и коэффициентом заполненности $\alpha = \frac{n}{m} < 1$ в предположении равномерного хеширования не превышает $\frac{1}{1-a}$

А значит вставка в среднем работает за $O(1 + \frac{1}{1-a})$

Хеш-таблицы: метод открытой адресации

А что по поводу сложности операций?

Худший случай любой операции понятен: O(N)

Утверждение #1: мат. ожидание количества исследований при неуспешном поиске в хеш-таблице с открытой адресацией и коэффициентом заполненности $\alpha = \frac{n}{m} < 1$
в предположении равномерного хеширования не превышает $\frac{1}{1-a}$

А значит вставка в среднем работает за $O(1 + \frac{1}{1-a})$

Утверждение #2: мат. ожидание … при успешном поиске … не превышает $\frac{1}{a} ln(\frac{1}{1-a})$

Хеш-таблицы: метод открытой адресации

А что по поводу сложности операций?

Про сложность удаления в открытой адресации:

https://upcommons.upc.edu/bitstream/handle/2117/122770/Jimenez.pdf

Краткие выводы: удаление можно сделать в среднем за величину, зависящую от коэффициента заполняемости, а значит можно и подобрать размер таблицы такой, чтобы все работало в среднем хорошо.

Хеш-таблицы: метод открытой адресации

Важно помнить про открытую адресацию:

1. Всегда нужно держать коэффициент заполненности строго меньше 1, иначе мы просто не сможем вставить элемент.

Хеш-таблицы: метод открытой адресации

Важно помнить про открытую адресацию:

1. Всегда нужно держать коэффициент заполненности строго меньше 1, иначе мы просто не сможем вставить элемент.

2. Исследования бывают не только линейными, про это в следующий раз.

Хеш-таблицы: метод открытой адресации

Важно помнить про открытую адресацию:

1. Всегда нужно держать коэффициент заполненности строго меньше 1, иначе мы просто не сможем вставить элемент.
2. Исследования бывают не только линейными, про это в следующий раз.

Зачем это все надо, чем цепочки то не угодили?

Хеш-таблицы: метод открытой адресации

Важно помнить про открытую адресацию:

1. Всегда нужно держать коэффициент заполненности строго меньше 1, иначе мы просто не сможем вставить элемент.

2. Исследования бывают не только линейными, про это в следующий раз.

Зачем это все надо, чем цепочки то не угодили?

1. Сильно экономим память на указателях в списках => можем ее использовать для расширения массива

Хеш-таблицы: метод открытой адресации

Важно помнить про открытую адресацию:

1. Всегда нужно держать коэффициент заполненности строго меньше 1, иначе мы просто не сможем вставить элемент.

2. Исследования бывают не только линейными, про это в следующий раз.

Зачем это все надо, чем цепочки то не угодили?

1. Сильно экономим память на указателях в списках => можем ее использовать для расширения массива

2. Нет аллокаций (кроме как при рехеше).

Хеш-таблицы: метод открытой адресации

Важно помнить про открытую адресацию:

1. Всегда нужно держать коэффициент заполненности строго меньше 1, иначе мы просто не сможем вставить элемент.

2. Исследования бывают не только линейными, про это в следующий раз.

Зачем это все надо, чем цепочки то не угодили?

1. Сильно экономим память на указателях в списках => можем ее использовать для расширения массива

2. Нет аллокаций (кроме как при рехеше).

3. Без удаления - проще написать.

Хеш-таблицы: осталось обсудить

1. Примеры хороших, плохих, ~~злых~~ хеш-функций
2. Как приблизиться к равномерному хешированию?

Хеш-таблицы: осталось обсудить

1. Примеры хороших, плохих, ~~злых~~ хеш-функций
2. Как приблизиться к равномерному хешированию?
3. Сколько бакетов иметь и добавлять при рехеше?

Хеш-таблицы: осталось обсудить

1. Примеры хороших, плохих, ~~злых~~ хеш-функций
2. Как приблизиться к равномерному хешированию?
3. Сколько бакетов иметь и добавлять при рехеше?
4. Как выработать устойчивость к pathological dataset?