

Структуры данных

Данные можно хранить по-разному

- list
- tuple
- dict
- pd.DataFrame

Структура данных

- позволяет хранить связанные по смыслу данные
- позволяет модифицировать эти данные: добавлять, искать, изменять, удалять
- позволяет эффективно решать определённый класс задач

Массив

- Структура данных, которая хранит фиксированное количество последовательно хранящихся значений.
- За счет этого обеспечивается быстрый доступ к элементу по его индексу

Массив

- Структура данных, которая хранит фиксированное количество последовательно хранящихся значений.
- За счет этого обеспечивается быстрый доступ к элементу по его индексу
- Если `start` – адрес начала массива в памяти, то i -й элемент находится по адресу `start + i * sizeof(array[0])`, где `sizeof(array[0])` – количество памяти, которое занимает один элемент массива.
- Доступ к произвольному элементу всегда за $O(1)$. Массив нельзя увеличить, но можно изменять значения элементов за $O(1)$.

А если мы не знаем заранее, сколько элементов нам понадобится хранить?

А если мы не знаем заранее, сколько элементов нам понадобится хранить?

- Можно расширять массив по ходу добавления данных – динамический массив (вектор)
- При создании нового вектора создадим пустой массив
- При добавлении нового элемента будем переписывать массив, увеличивая его размер на 1

А если мы не знаем заранее, сколько элементов нам понадобится хранить?

- Можно расширять массив по ходу добавления данных – динамический массив (вектор)
- При создании нового вектора создадим пустой массив
- При добавлении нового элемента будем переписывать массив, увеличивая его размер на 1. Получим сложность добавления элемента по времени $O(n)$, где n – текущий размер массива.

Можно ли быстрее?

А если мы не знаем заранее, сколько элементов нам понадобится хранить?

- Можно расширять массив по ходу добавления данных – динамический массив (вектор)
- При создании нового вектора создадим пустой массив
- При добавлении нового элемента будем переписывать массив, увеличивая его размер **в 2 раза** (или в некоторое другое постоянное число раз). Получаем сложность добавления элемента по времени $O(n)$, если место во «внутреннем» массиве закончилось, и $O(1)$, если оно ещё есть

Какая на самом деле сложность вставки?


- Большинство вставок лёгкие. Трудоемкие вставки происходят все реже с ростом длины массива, но и сложность «трудной» вставки растёт (линейно)

Какая на самом деле сложность вставки?

- Большинство вставок лёгкие. Трудоемкие вставки происходят все реже с ростом длины массива, но и сложность «трудной» вставки растёт (линейно)
- Будем считать среднюю (*амортизационную*) сложность вставки.

2

2 7

 2 7 1

2 7 1 3

 2 7 1 3 8

2 7 1 3 8 4

└──────────┘

Logical size

└──────────────────────────┘

Capacity

Амортизационный анализ вставки в динамический массив

- Пусть в пустой вектор выполнено N вставок и N – степень двойки (для удобства). Трудоемкие вставки происходят $\log_2 N$ раз и занимают время, пропорциональное текущему размеру списка, все остальные – лёгкие и занимают константное время. Тогда средняя стоимость одной вставки:

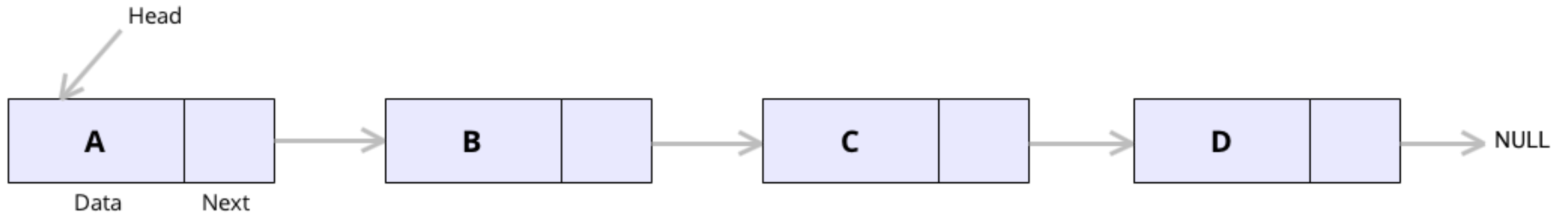
$$\frac{\sum_{i=1}^{\log_2 N} O(1) \times 2^i + (N - \log_2 N) \times O(1)}{N} =$$
$$= O(1) \times \frac{2^{\log_2 N + 1} - 2}{N} + O(1) = O(1) \times \frac{2N - 2}{N} = O(1)$$

Итого

- list в Python – это динамический массив
- В Java ArrayList, в C++ std::vector
- Можно реализовать дополнительные операции: pop, shrink_to_fit

Программируем динамический массив

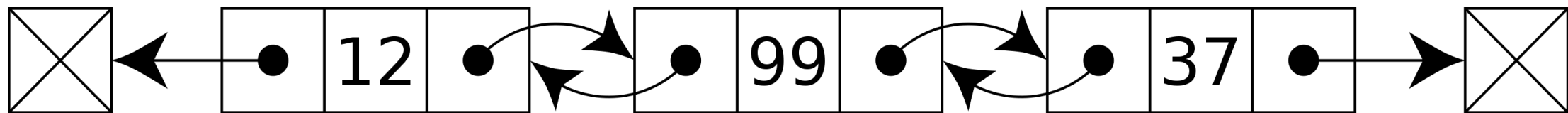
Связный список



СВЯЗНЫЙ СПИСОК

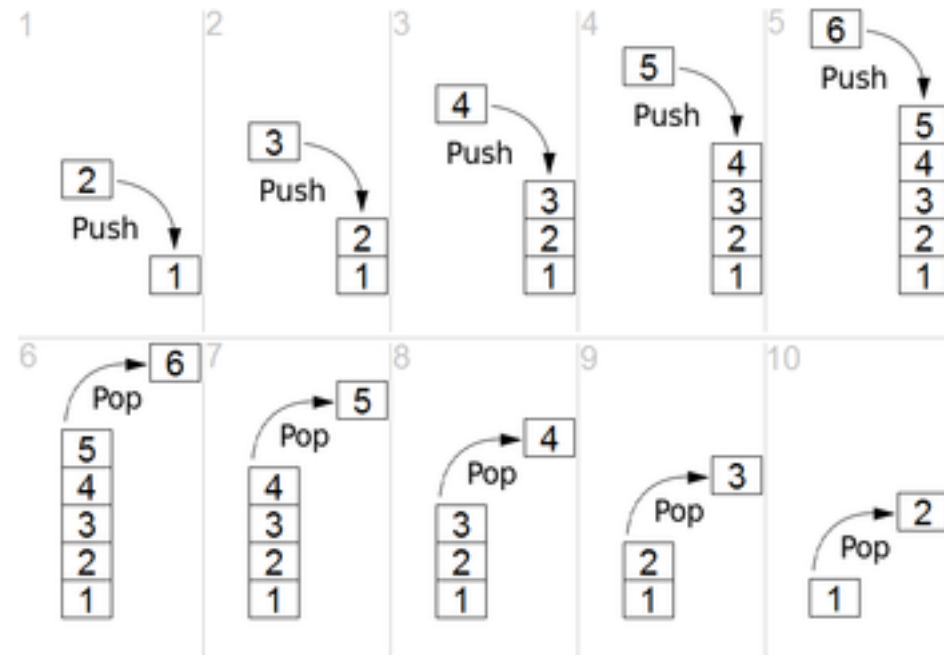
- Позволяет хранить разные элементы списка в разных частях памяти (в отличие от массива)
- Занимает больше совокупного места в памяти из-за наличия связей между элементами
- Позволяет вставлять данные в любое место списка за $O(1)$
- Нет быстрого случайного доступа

Двусвязный список



Стек

- Структура данных, поддерживающая метод push (добавить элемент) и pop (вернуть последний добавленный элемент и удалить его из списка)

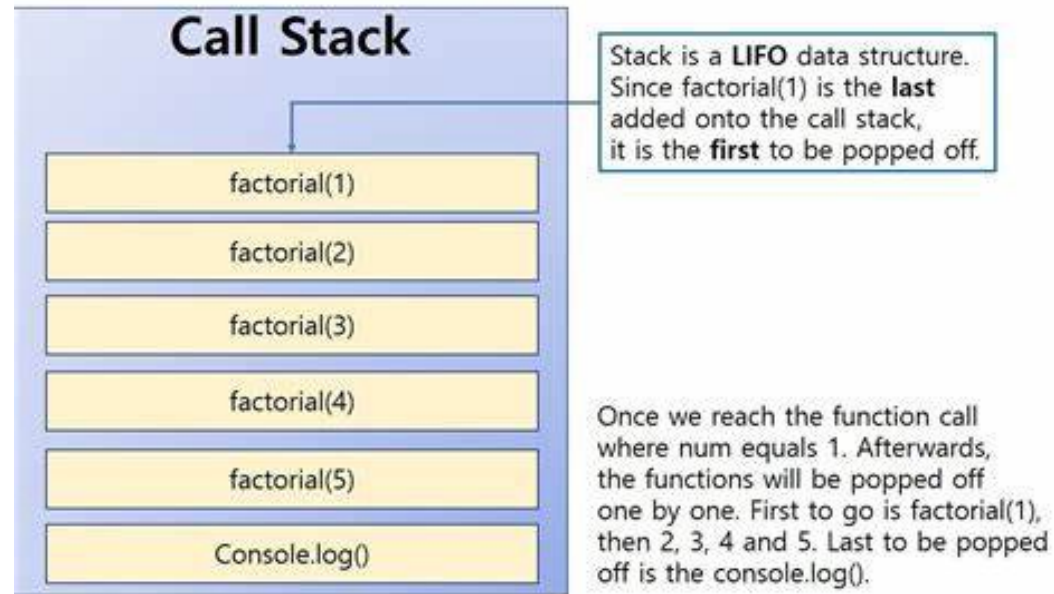


Стек

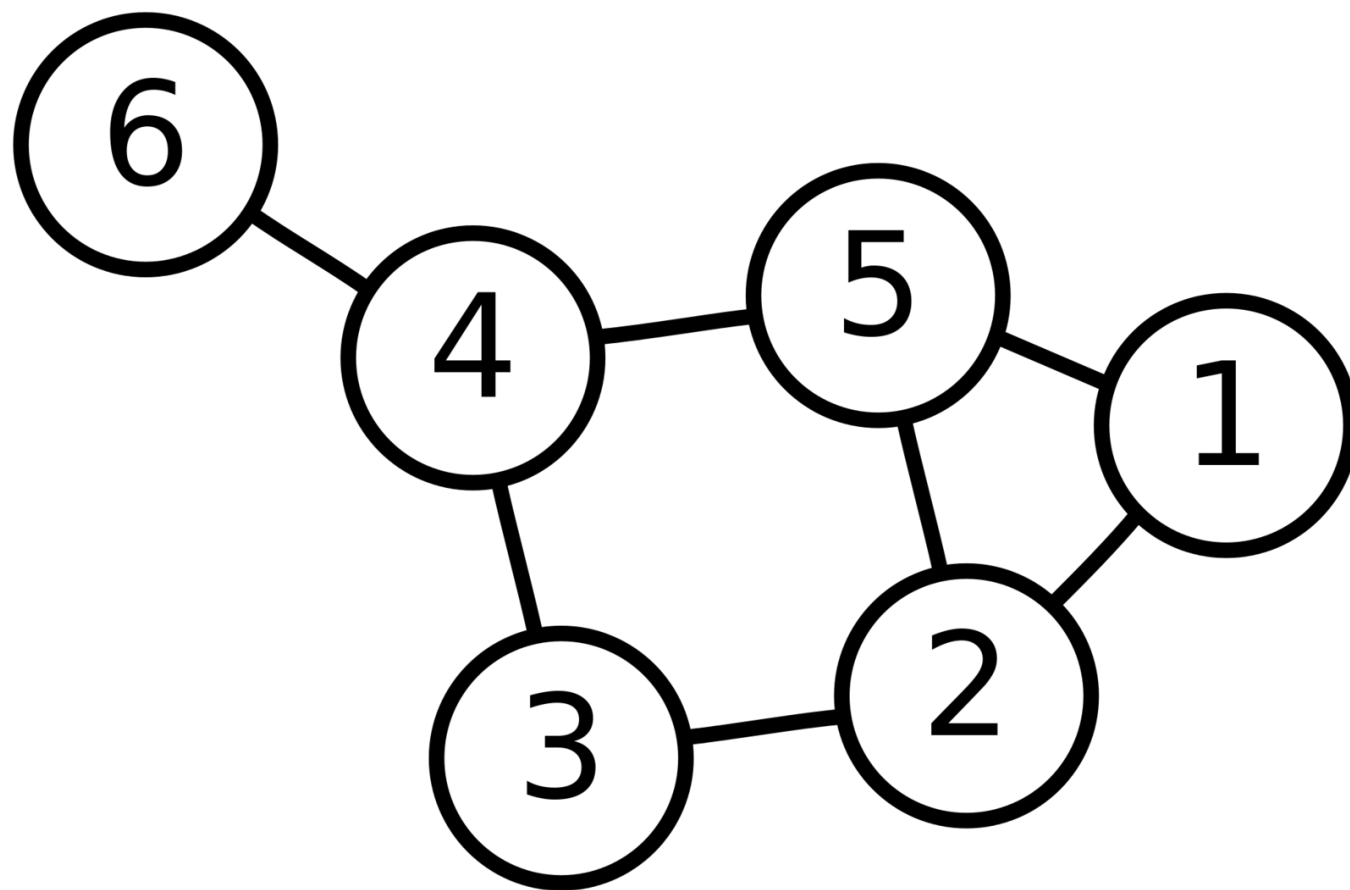
- Структура данных, поддерживающая метод push (добавить элемент) и pop (вернуть последний добавленный элемент и удалить его из списка)



Применение стека



Применение стека



Применение стека

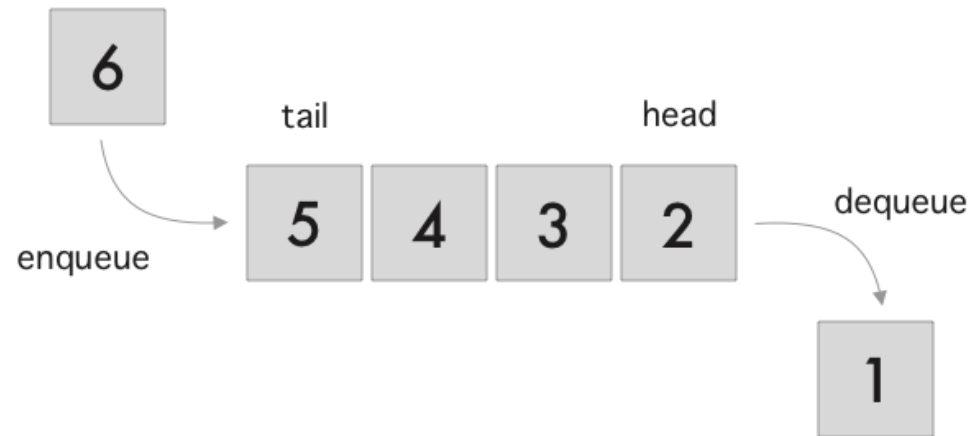
```
{
  "orders": [
    {
      "orderno": "748745375",
      "date": "June 30, 2088 1:54:23 AM",
      "trackingno": "TN0039291",
      "custid": "11045",
      "customer": [
        {
          "custid": "11045",
          "fname": "Sue",
          "lname": "Hatfield",
          "address": "1409 Silver Street",
          "city": "Ashland",
          "state": "NE",
          "zip": "68003"
        }
      ]
    }
  ]
}
```

Как реализовать стек?

- Связный список
- Динамический массив

Очередь

- Структура данных, поддерживающая метод `enqueue` (добавить элемент в конец очереди) и `dequeue` (вернуть элемент, стоящий в начале очереди, и удалить его)



Как реализовать очередь?

- (Динамический) массив
- Связный список

Как реализовать очередь?

- (Динамический) массив
- Связный список
- Два стека

FIFO LIFO

- Очередь – first in, first out
- Стек – last in, first out

Программируем связный список и учимся
пользоваться дебаггером в PyCharm

Далее

- Очередь с приоритетом, heap sort
- Хеш-таблицы, словари

Очередь с приоритетом

- Структура данных, поддерживающая метод `enqueue(e1, priority)` (добавить элемент `e1` в очередь с приоритетом `priority`) и `dequeue` (вернуть элемент с минимальным приоритетом и удалить его)
- Это как обычная очередь, только элементы будут удаляться из очереди в порядке, определяемом приоритетом

А как реализовать?

	Вставка	Извлечение
Связный список	$O(1)$	$O(N)$, N – длина очереди
Сортированный связный список	$O(N)$	$O(1)$
Массив	$O(1)$	$O(N)$
Сортированный массив	$O(N)$	$O(1)$

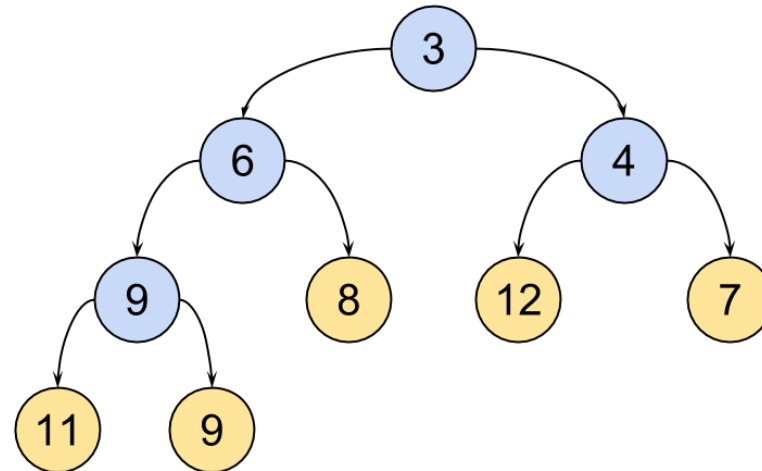
А как реализовать?

	Вставка	Извлечение	В среднем на 1 элемент
Связный список	$O(1)$	$O(N)$, N – длина очереди	$O(N)$
Сортированный связный список	$O(N)$	$O(1)$	$O(N)$
Массив	$O(1)$	$O(N)$	$O(N)$
Сортированный массив	$O(N)$	$O(1)$	$O(N)$

Двоичная куча

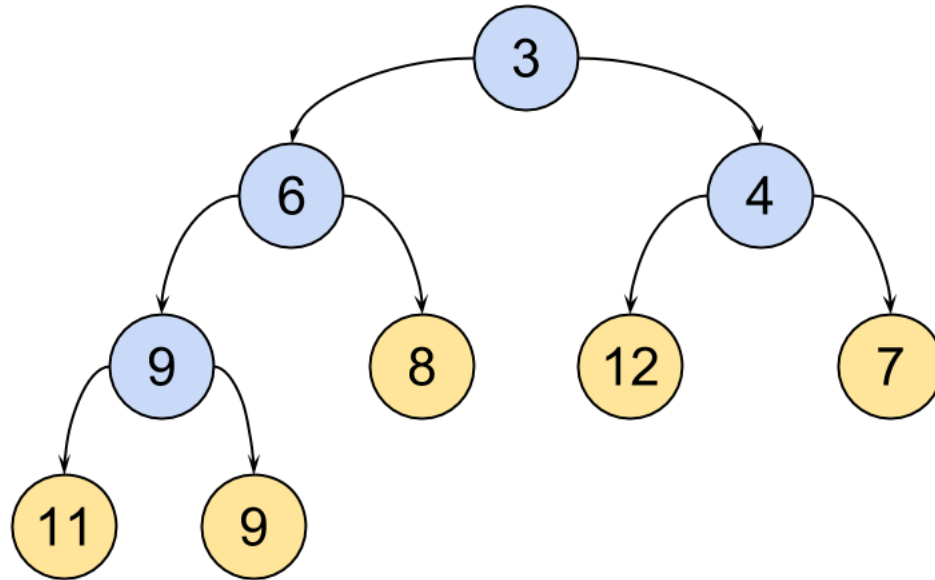
Дерево, для которого справедливы условия:

- Значения в любой вершине не больше, чем значения её потомков.
- На i -ом слое 2^i вершин, кроме последнего. Слои нумеруются с нуля.
- Последний слой заполнен слева направо



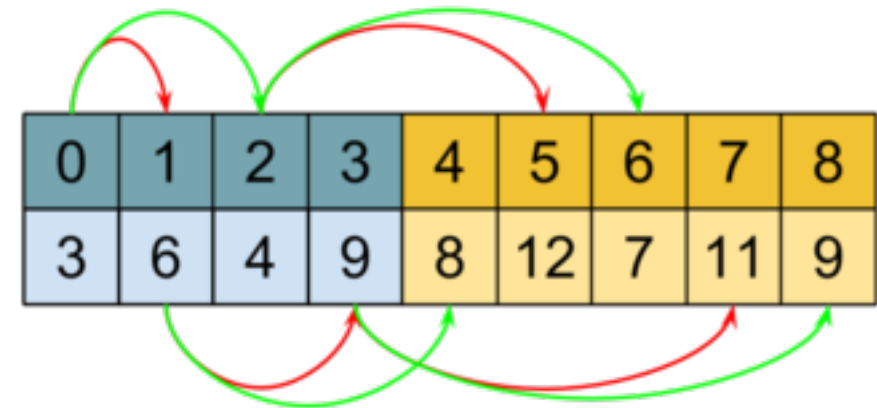
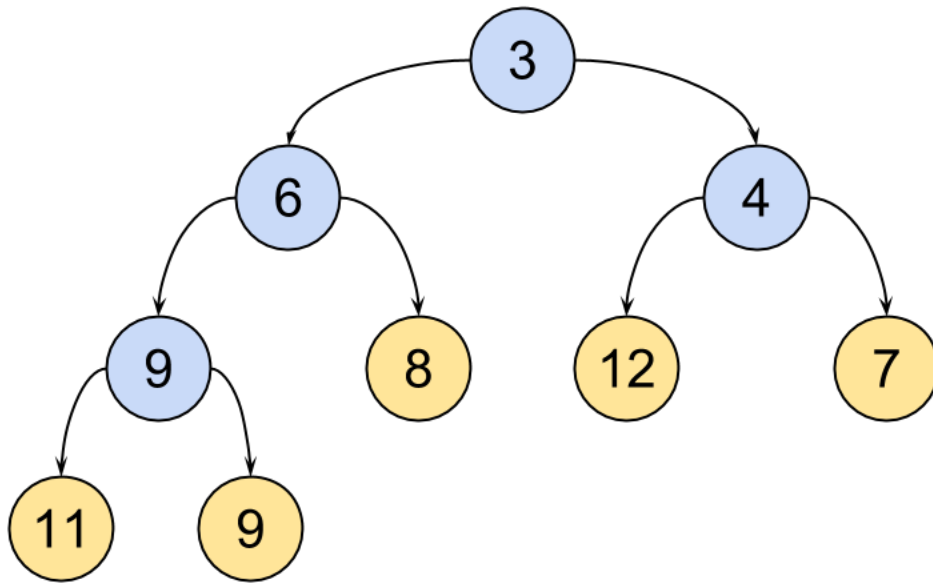
Представление кучи в памяти

- Можно хранить в виде массива, где нулевой элемент – корень, а $a[2i + 1]$ и $a[2i + 2]$ – потомки элемента $a[i]$



Представление кучи в памяти

- Можно хранить в виде массива, где нулевой элемент – корень, а $a[2i + 1]$ и $a[2i + 2]$ – потомки элемента $a[i]$



Просеивание кучи

- Хотим удалить минимальный элемент из кучи
- Первым делом заменим его последним элементом, чтобы не переписывать всю кучу. Затем нужно восстановить свойство упорядоченности кучи
- Если корень дерева \leq своим двум потомкам, всё хорошо. Если нет, поменяем его местами с меньшим потомком, а затем запустим ту же процедуру для поддерева, для которого этот потомок является корнем

Просеивание кучи

- Хотим удалить минимальный элемент из кучи
- Первым делом заменим его последним элементом, чтобы не переписывать всю кучу. Затем нужно восстановить свойство упорядоченности кучи
- Если корень дерева \leq своим двум потомкам, всё хорошо. Если нет, поменяем его местами с меньшим потомком, а затем запустим ту же процедуру для поддерева, для которого этот потомок является корнем
- $O(\log N)$ по времени

Просеивание кучи

- Хотим добавить элемент в кучу
- Допишем его в конец кучи. Затем нужно восстановить свойство упорядоченности.
- Если элемент в новой вершине больше значения в родительской вершине, всё хорошо. Если нет, поменяем его местами с родителем, а затем запустим ту же процедуру для родителя.

Просеивание кучи

- Хотим добавить элемент в кучу
- Допишем его в конец кучи. Затем нужно восстановить свойство упорядоченности.
- Если элемент в новой вершине больше значения в родительской вершине, всё хорошо. Если нет, поменяем его местами с родителем, а затем запустим ту же процедуру для родителя.
- $O(\log N)$ по времени

Очередь с приоритетом

	Вставка	Извлечение	В среднем на 1 элемент
Связный список	$O(1)$	$O(N)$, N – длина очереди	$O(N)$
Сортированный связный список	$O(N)$	$O(1)$	$O(N)$
Массив	$O(1)$	$O(N)$	$O(N)$
Сортированный массив	$O(N)$	$O(1)$	$O(N)$
Двоичная куча	$O(\log N)$	$O(\log N)$	$O(\log N)$

Хеш-таблица

- (Динамический) массив позволяет осуществлять мгновенный доступ к элементам по индексу (0, 1, ..., N-1)
- Хотим такой же быстрый доступ к элементу по ключу (произвольному числу, строке, объекту etc)

Хеш-таблица

Структура данных, позволяющая:

1. Получать элемент по ключу ($a[key]$)
2. Добавлять пару «ключ-значение» ($a[key] = value$)
3. Удалять пару по ключу ($del\ a[key]$)

Хеш-таблица

Структура данных, позволяющая:

1. Получать элемент по ключу (`a[key]`)
2. Добавлять пару «ключ-значение» (`a[key] = value`)
3. Удалять пару по ключу (`del a[key]`)

Словари и множества в Python – это хеш-таблицы

Хеш-таблица с целочисленными ключами

- Хотим хранить в хеш-таблице не более N элементов
- Заведём массив $a[0..N-1]$ длины N
- $f: \mathbb{Z} \rightarrow \{0, 1, \dots, N-1\}$ – функция, сопоставляющая ключу его индекс в массиве
- При добавлении элемента key будем обращаться к элементу массива $a[f(key)]$

Хеш-таблица с произвольными ключами

- Хотим хранить в хеш-таблице не более N элементов
- Заведём массив $a[0..N-1]$ длины N
- $f: U \rightarrow \{0, 1, \dots, N-1\}$ – функция, сопоставляющая ключу его индекс в массиве (U – множество всех возможных ключей)
- При добавлении элемента key будем обращаться к элементу массива $a[f(key)]$

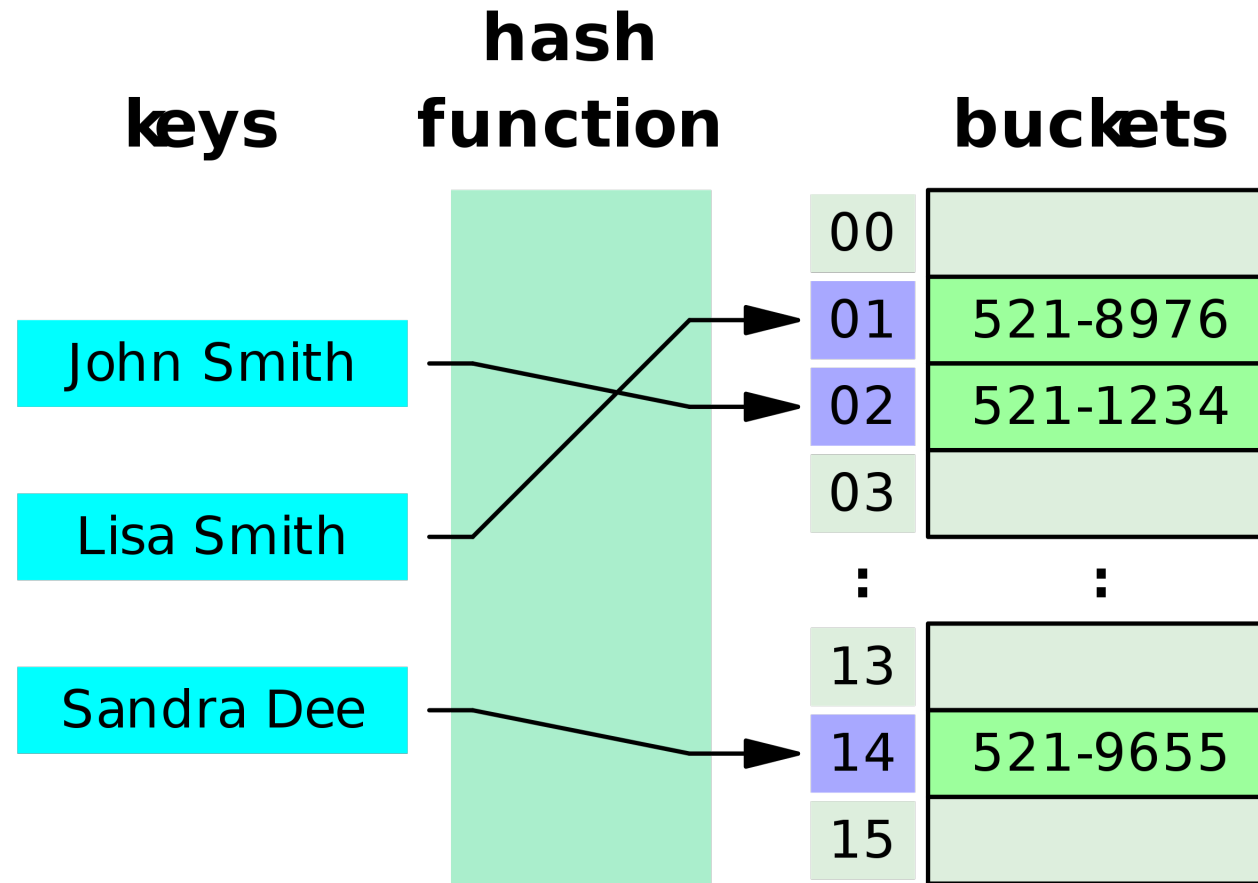
Хеш-таблица с произвольными ключами

- Хотим хранить в хеш-таблице не более N элементов
- Заведём массив $a[0..N-1]$ длины N
- $f: U \rightarrow \{0, 1, \dots, N-1\}$ – функция, сопоставляющая ключу его индекс в массиве (U – множество всех возможных ключей)
- При добавлении элемента key будем обращаться к элементу массива $a[f(key)]$
- Сложность всех операций – $O(f(key))$

Хеш-таблица с произвольными ключами

- Хотим хранить в хеш-таблице не более N элементов
- Заведём массив $a[0..N-1]$ длины N
- $f: U \rightarrow \{0, 1, \dots, N-1\}$ – функция, сопоставляющая ключу его индекс в массиве (U – множество всех возможных ключей) (*хеш-функция*)
- При добавлении элемента key будем обращаться к элементу массива $a[f(key)]$
- Сложность всех операций – $O(1)$

Хеш-таблица с произвольными ключами



Задача

- В группе n студентов. Какова вероятность, что хотя бы у двух человек совпадут день и месяц рождения?

Задача

- В группе n студентов. Какова вероятность, что хотя бы у двух человек совпадут день и месяц рождения?
- Вероятность, что у конкретной пары студентов совпадут дни рождения, равна $\frac{1}{365}$.

Задача

- В группе n студентов. Какова вероятность, что хотя бы у двух человек совпадут день и месяц рождения?
- Вероятность, что у конкретной пары студентов совпадут дни рождения, равна $\frac{1}{365}$. Всего студентов n , значит, по формуле сложения вероятностей искомая вероятность равна $\frac{n}{365}$ (для $n = 20$, например, 5.5%)

Задача

- В группе n студентов. Какова вероятность того, что у двух человек совпадут день и месяц рождения?
- Вероятность, что у конкретной пары студентов совпадут дни рождения, равна $\frac{1}{365}$. Всего студентов n , значит, по формуле сложения вероятностей комбинаторная вероятность равна $\frac{n}{365}$ (для $n = 20$, например, 5.5%).

Решение

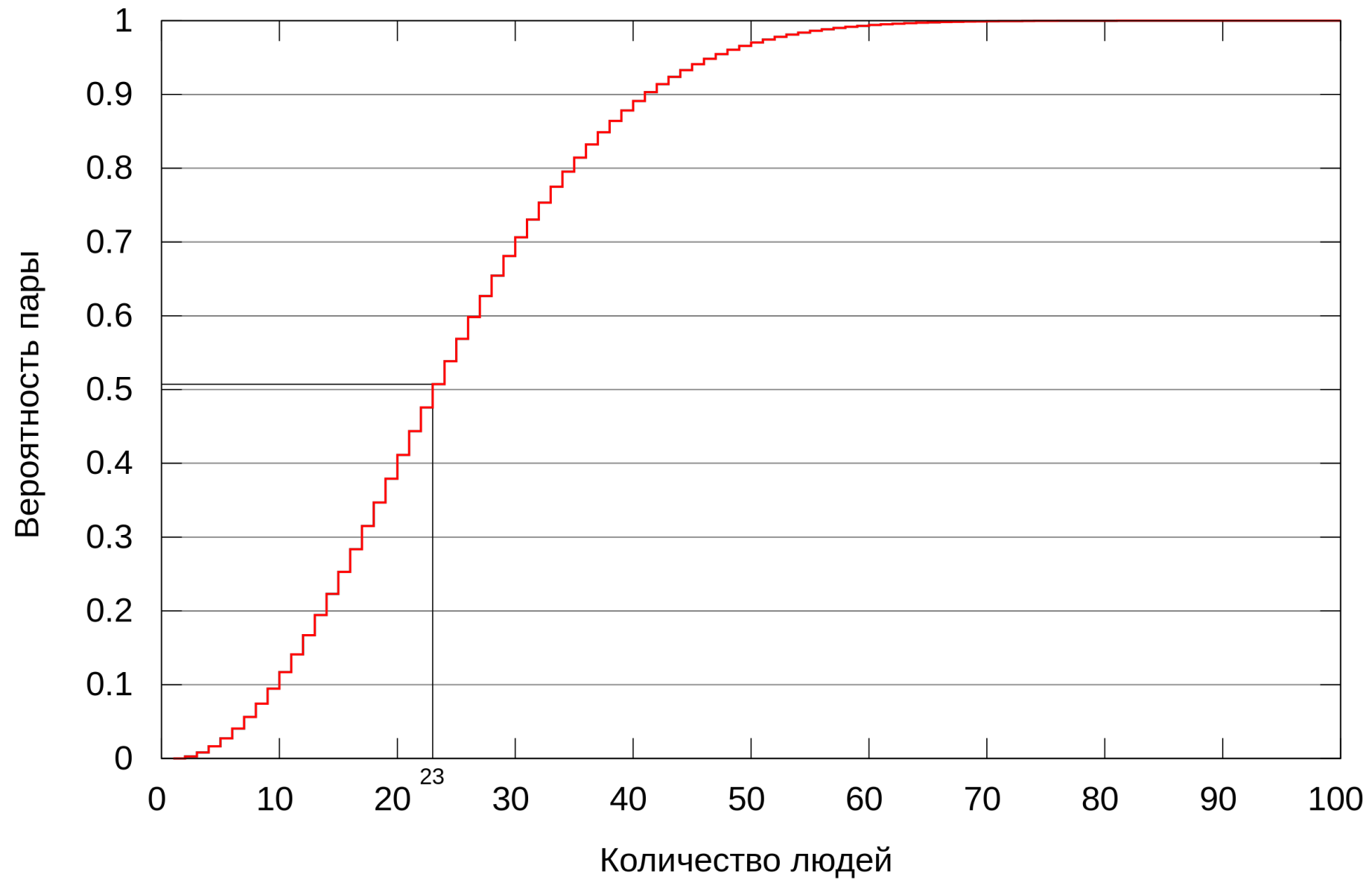
- Найдем вероятность обратного события – дни рождения у всех разные. Пусть они равны a_1, a_2, \dots, a_n , тогда
- $P(a_2 \neq a_1) = \frac{364}{365} = 1 - \frac{1}{365}$
- $P(a_3 \neq a_2 \ \& \ a_3 \neq a_1) = 1 - \frac{2}{365}$
- $P(a_4 \neq a_3 \ \& \ a_4 \neq a_2 \ \& \ a_4 \neq a_1) = 1 - \frac{3}{365}$
- ...
- $P(a_n \neq a_{n-1} \ \& \ \dots \ \& \ a_n \neq a_1) = 1 - \frac{n-1}{365}$

Решение

Вероятность обратного события равна произведению всех вероятностей, ибо события независимые:

$$P_o = \left(1 - \frac{1}{365}\right) \left(1 - \frac{2}{365}\right) \dots \left(1 - \frac{n-1}{365}\right) = \frac{364 \times 363 \times \dots \times (365 - n + 1)}{365^n}$$
$$= \frac{365!}{365^n (365 - n)!}$$

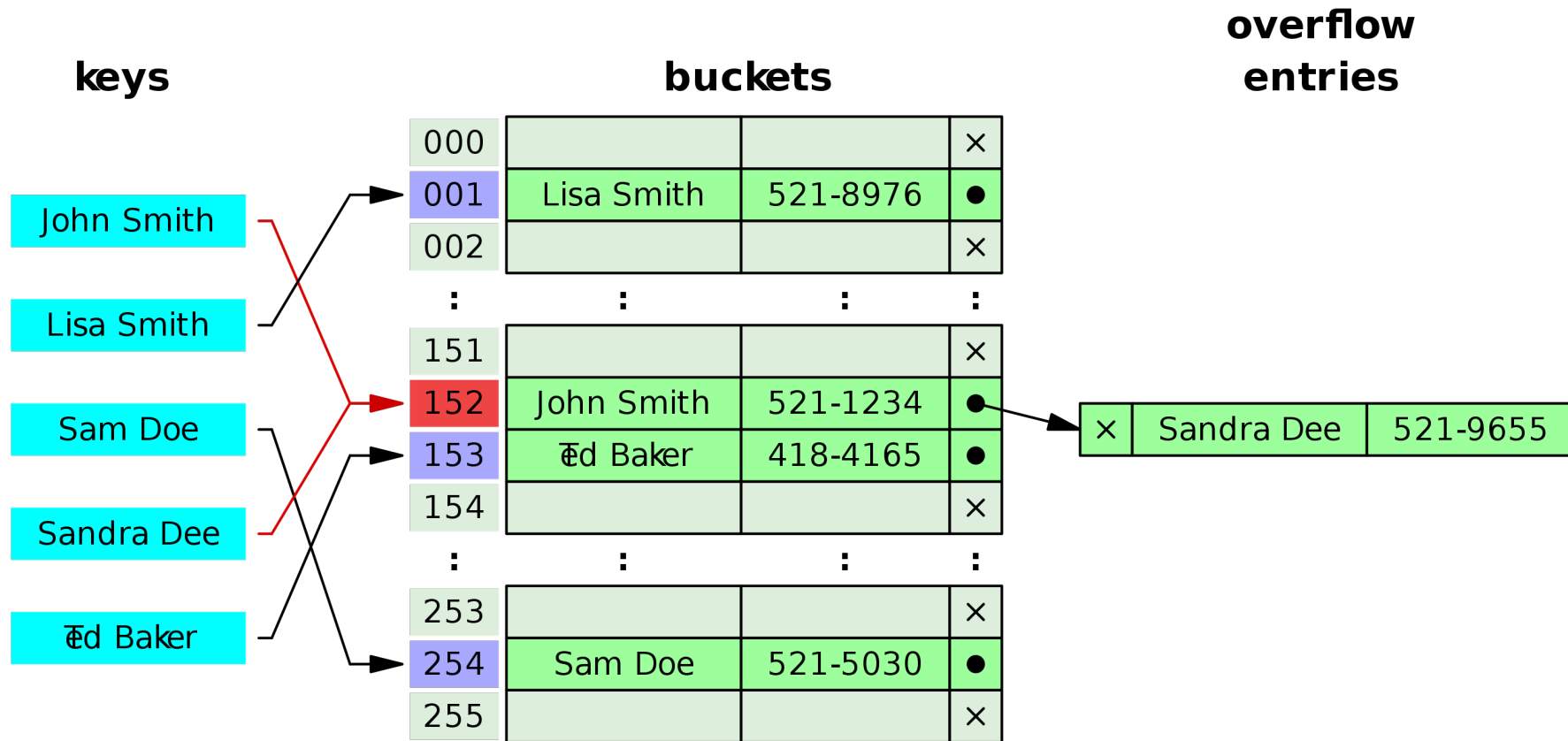
Искомая вероятность равна $1 - P_o$.



Разрешение коллизий

- Вероятность коллизии в хеш-таблице (совпадение результатов хеш-функций) очень велика и увеличивается с ростом хеш-таблицы.

Разрешение коллизий методом цепочек



Сложность случайного доступа

- Сложность доступа по ключу key : $O(1 + \text{chainlen})$, где chainlen – длина связного списка по индексу $f(key)$
- Если мы используем массив длины N и сейчас в хеш-таблице хранится K элементов, то матожидание длины цепочки равно $\frac{K}{N}$
- Соответственно, сложность доступа, удаления и присвоения в хеш-таблице: $O(1 + \alpha)$, где $\alpha = \frac{K}{N}$ - коэффициент заполнения таблицы (load factor)

Сложность случайного доступа

- Сложность доступа по ключу key : $O(1 + \text{chainlen})$, где chainlen – длина связного списка по индексу $f(key)$
- Если мы используем массив длины N и сейчас в хеш-таблице хранится K элементов, то матожидание длины цепочки равно $\frac{K}{N}$
- Соответственно, сложность доступа, удаления и присвоения в хеш-таблице: $O(1 + \alpha)$, где $\alpha = \frac{K}{N}$ - коэффициент заполнения таблицы (load factor)
- При достижении определённого значения α стоит расширять хеш-таблицу (так же, как динамический массив)

Вспоминаем хеш-таблицы

- Хеш-таблицы позволяют вставлять и изменять данные по ключу за $O(1)$
- Коллизии – частое явление в хеш-таблицах, с ними можно бороться, храня значения с одинаковыми хешами в связанных списках

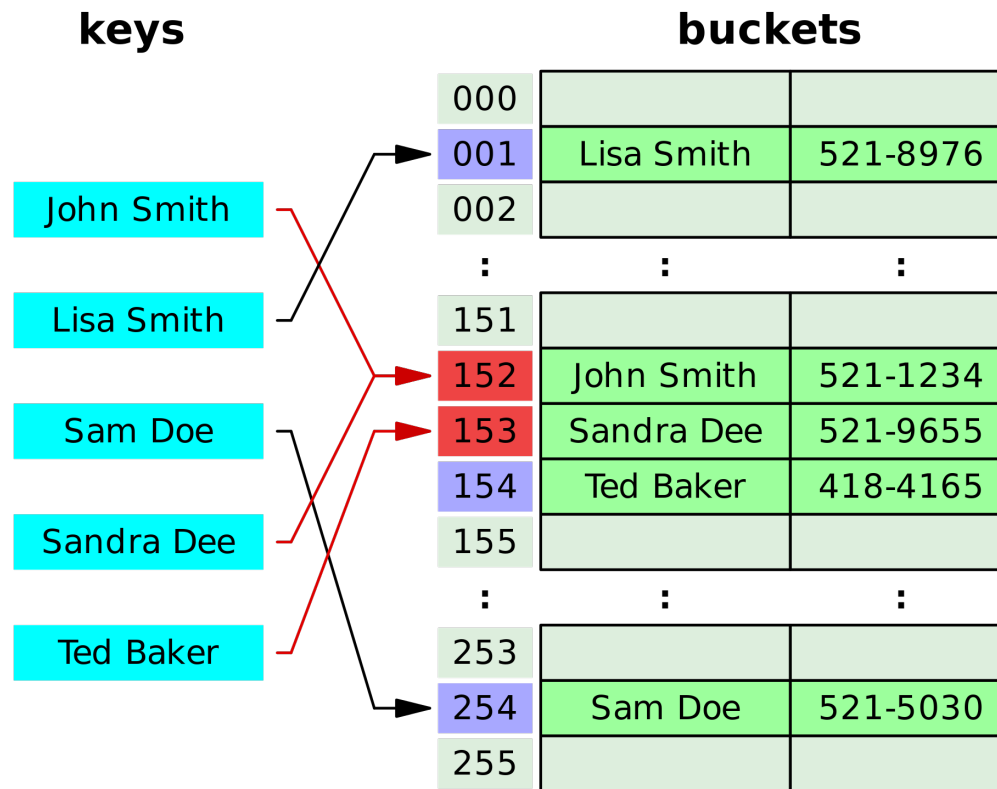
Хеш-таблицы с открытой адресацией

- Будем хранить в массиве сами ключи и значения
- При вставке будем проверять ячейки массива в определённом порядке $h_0(x), h_1(x), \dots, h_{n-1}(x)$ (просматривая в худшем случае все ячейки).
- $h_i(x)$ – последовательность проб.

Хеш-таблицы с открытой адресацией: линейное пробирование

- Будем просматривать ячейки друг за другом
- i -й элемент последовательности проб равен $(f(x) + i) \% N$, где $f(x)$ – хеш вставляемого элемента

Хеш-таблицы с открытой адресацией: линейное пробирование



Хеш-таблицы с открытой адресацией: квадратичное пробирование

- Будем просматривать ячейки, увеличивая расстояние между просматриваемыми ячейками
- i -й элемент последовательности проб равен $(f(x) + i^2) \% N$, где $f(x)$ – хеш вставляемого элемента
- Это позволяет частично решить проблему кластеризации

А как удалять элементы в таких хеш-таблицах?

- Нельзя просто удалять элемент с того места, где он хранится
- Заведём отдельный массив и будем помечать элементы как удалённые – это позволит не разрывать цепочки

Сложность доступа по ключу в хеш-таблицах с открытой адресацией

- $O(\frac{1}{1-\alpha})$ в среднем, где α – доля занятых ячеек в массиве
- $O(1)$, если не допускать превышения некоторого порога α , а затем экспоненциально расширять массив (как в динамическом массиве) и перехешировать элементы

Хеш-таблицы: итог

- Позволяют хранить пары ключ-значение с доступом, записью и удалением по ключу в среднем за $O(1)$, в худшем случае за $O(\text{текущее кол-во элементов})$.
- Существуют различные способы разрешения коллизий в зависимости от того, что более ценно – память или время
- На основе хеш-таблицы можно легко реализовать структуру «множество».

Хеш-функция

- Должна быть детерминированной
- Должна быть быстро вычислима
- Должна быть равномерно распределена по множеству значений

Хеширование целых чисел

- $f(x) = 1$
- $f(x) = x \% k$

Хеширование целых чисел

- $f(x) = 1$
- $f(x) = x \% k$
- Более продвинутые алгоритмы

Хеширование строк

- $f(x) = s_0 + s_1p + s_2p^2 + s_3p^3 + \dots + s_np^n$, p обычно выбирают простым
https://e-maxx.ru/algo/string_hashes

Хеширование в Python

- Хеши объекта вычисляется с помощью функции `hash(your_object)`
- У неизменяемых объектов (`int`, `str`) есть хеш
- У изменяемых (`list`, `dict`, ...) - нет

Хеширование в Python

- Хеши объекта вычисляется с помощью функции `hash(your_object)`
- У неизменяемых объектов (`int`, `str`) есть хеш
- У изменяемых (`list`, `dict`, ...) – нет
- Для своих классов можно реализовать функцию `__hash__`

Хеширование в Python

- Хеш объекта вычисляется с помощью функции `hash(your_object)`
- У неизменяемых объектов (`int`, `str`) есть хеш
- У изменяемых (`list`, `dict`, ...) – нет
- Для своих классов можно реализовать функцию `__hash__`
- Более корректный способ:

```
def __hash__(self):  
    return hash((self.name, self.nick, self.color))
```

Хеширование в Python

```
[In [4]: hash(1)
```

```
Out[4]: 1
```

```
[In [5]: hash('Hello world')
```

```
Out[5]: 15810118034299150
```

```
[In [6]: hash(23435)
```

```
Out[6]: 23435
```

```
[In [7]: hash((1,2,3))
```

```
Out[7]: 529344067295497451
```

```
[In [8]: hash([1, 2, 3])
```

```
-----  
TypeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-8-84d65be9aa35> in <module>
```

```
----> 1 hash([1, 2, 3])
```

```
TypeError: unhashable type: 'list'
```

```
In [9]: █
```

Про хранение паролей в базах данных

- У нас есть сервис, где можно регистрироваться пользователям
- Почему хранить пароли в открытом виде (plain text) в базе данных плохо?

Про хранение паролей в базах данных

- У нас есть сервис, где можно регистрироваться пользователям
- Почему хранить пароли в открытом виде (plain text) в базе данных плохо?
- Можно хранить хеши паролей и сравнивать хеши

Про хранение паролей в базах данных

- У нас есть сервис, где можно регистрироваться пользователям
- Почему хранить пароли в открытом виде (plain text) в базе данных плохо?
- Можно хранить хеши (например, md5) паролей и сравнивать хеши, но это тоже не очень безопасно

Про хранение паролей в базах данных

- Хороший способ хранить пароли – генерировать случайную строку (соль) для каждого пользователя и хранить в базе данных соль и значение `hash(password + salt)`

Про хранение паролей в базах данных

- Хороший способ хранить пароли – генерировать случайную строку (соль) для каждого пользователя и хранить в базе данных соль и значение `hash(password + salt)`
- Это позволяет избежать взлом хеша с помощью перебора по словарям

Про хранение паролей в базах данных

- Хороший способ хранить пароли – генерировать случайную строку (соль) для каждого пользователя и хранить в базе данных соль и значение `hash(password + salt)`
- Это позволяет избежать взлом хеша с помощью перебора по словарям
- К *криптографическим* хеш-функциям есть отдельные требования