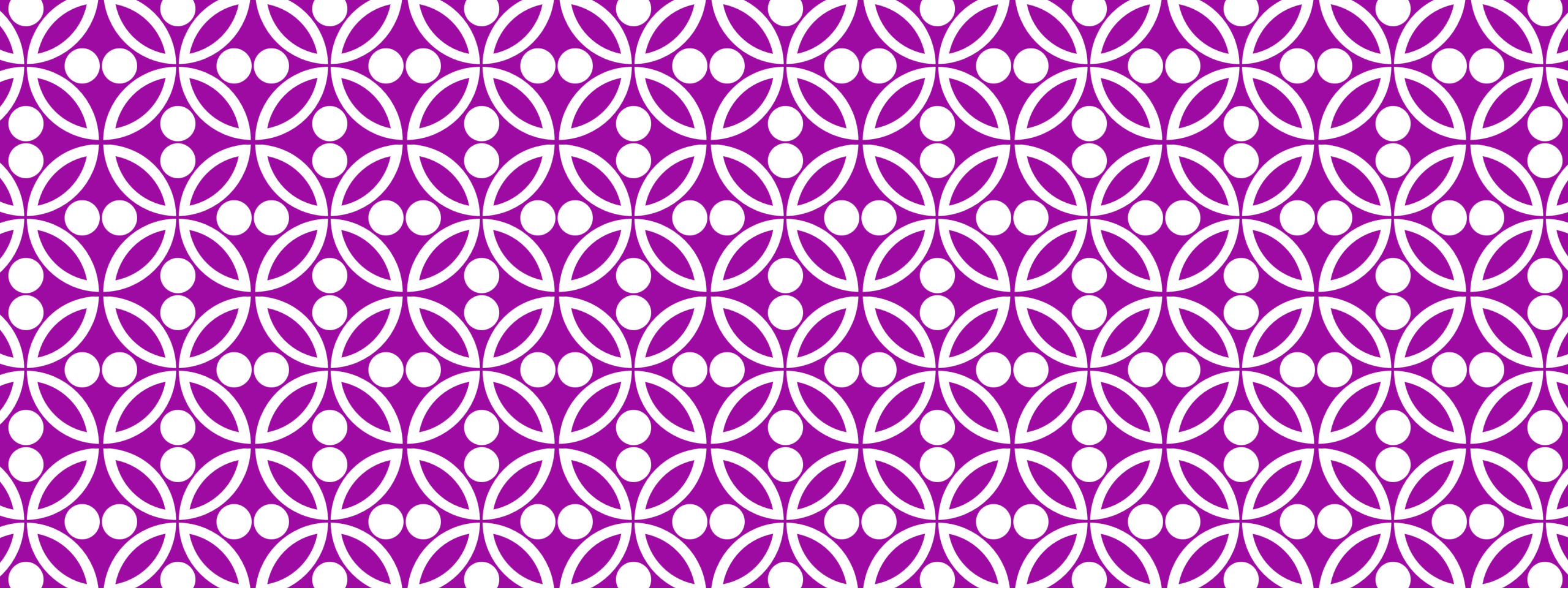


توابع درهم سازی
HASH TABLES

فرشته دهقانی

سرفصل مطالب

- مقدمه
- معرفی توابع درهم سازی
- برخی توابع درهم سازی
- برخورد و روش های مقابله با آن

شرح مسئله

فرض کنید مجموعه از اعداد در بازه صفر تا صد داریم و می خواهیم داده ساختاری برای نگهداری آنها طراحی کنیم که امکان انجام عملیات‌های درج، حذف و پیدا کردن از مرتبه O(1) باشد.

- از یک آرایه به طول صد استفاده می کنیم. اگر خانه **i**ام آرایه مقدار صف داشته باشد، یعنی عدد **i** در مجموعه وجود ندارد و اگر مقدار یک باشد، یعنی عدد **i** در مجموعه وجود دارد. به طور مشابه حذف اعداد با صفر کردن خانه متناظر و اضافه کردن اعداد با یک کردن آن انجام می شود.
- **یک مسئله سخت‌تر:** فرض کنید این‌بار اعداد در بازه صفر تا 10^{12} است اما میدانیم حداکثر 10^{3} عدد در مجموعه وجود خواهد داشته باشند. آیا راه قبلی قابل استفاده است؟

معرفی تابع درهمسازی

تابعی مانند $h: D \rightarrow L$: اعضای مجموعه D (با اندازه دلخواه) را به مجموعه با اندازه ثابت L نگاشت می دهد

معمولا اندازه D بسیار بزرگتر از اندازه L است (مزایا و معایب)

مثال: فرض کنید در مساله قسمت قبل تابعی داشته باشیم که اعداد بین ۰ و 10^{12} را به اعداد بین ۰ و 10^3 نگاشت کند، با این تضمین که احتمال نگاشت شدن دو عدد متفاوت به یک عدد در این بازه قابل صرف نظر باشد(همچنین فرض کنید هزینه نگاشت کردن کم و قابل صرف نظر باشد که معمولا همینطور است).

حال با کمک این تابع، می‌توانیم ایده مساله اول را به مساله بزرگتر تعمیم دهیم به گونه‌ای که برای درج، یافتن و حذف یک عنصر مانند x به خانه h(x) با 10^3عنصر مراجعه شود

کاربرد

داده ساختارهایی چون جدول درهمسازی

سیستم‌های رمزنگاری

مبادلات اینترنتی

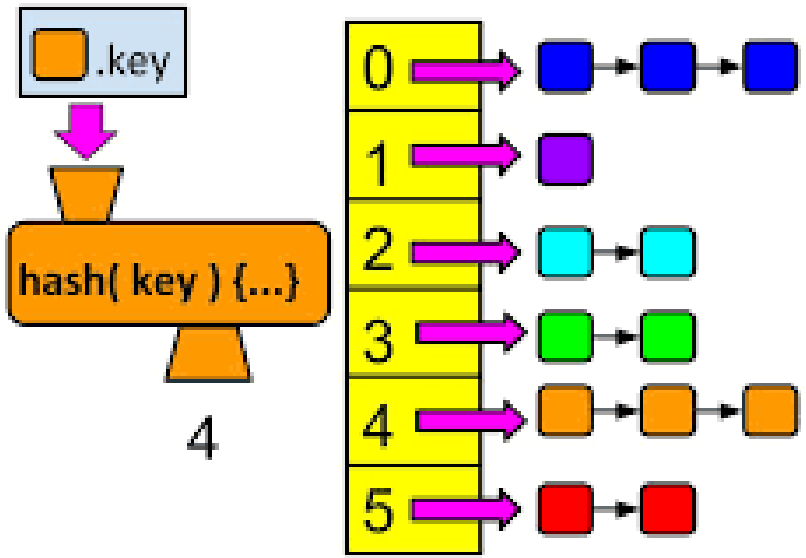
سیستم‌های تعیین هویت دیجیتال مانند تشخیص اثر انگشت

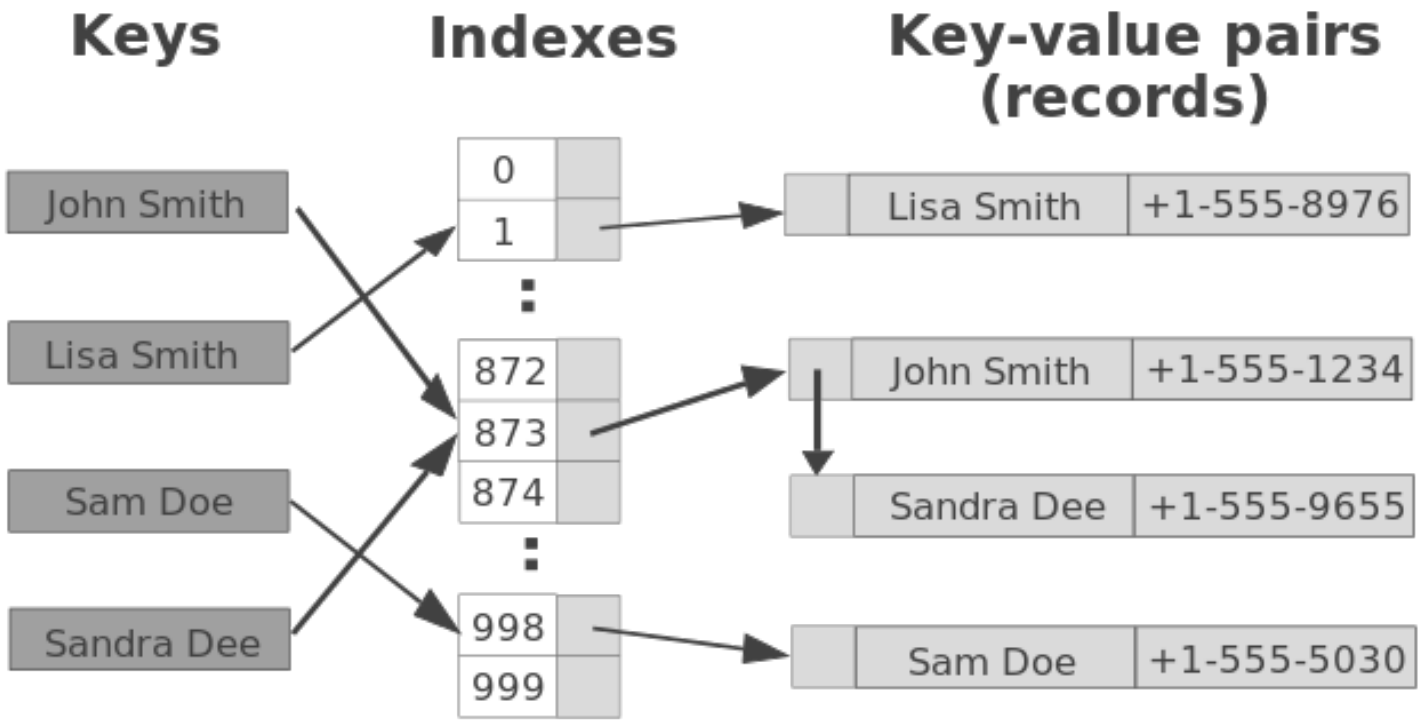
بررسی صحت فایل‌ها

یک کاربرد: جدول درهمسازی

- داده ساختاری به اسم جدول درهمسازی: تعمیمی از مثال زده شده در قسمت قبل
- این داده ساختار از یک آرایه عادی برای نگهداری عناصر قرار داده شده در آرایه استفاده می‌کند
- اندیسهای مورد نظر را با استفاده از یک تابع درهمسازی به دست می‌آورد.
- به عبارت دیگر در تعریف تابع درهمسازی، D همان مجموعه‌ایست که داده ها از آن می‌آیند و L مجموعه اندیس های آرایه است

جدول درهمسازی





توابع درهم سازی
روش باقیمانده تقسیم

در این روش به هر عدد باقی مانده تقسیمش بر یک عدد ثابت مانند m را نسبت می‌دهیم:

$$H(x) = x \; mode \; m$$

انتخاب مقدار مناسب برای m از اهمیت ویژه ای برخوردار است. اگر m توانی از دو باشد، خروجی تابع تنها log(m) بیت کم ارزش x است و بقیه بیت های آن نقشی ندارند

بهتر است m یک عدد اول و دور از توان دو باشد

Assume a table with 8 slots:

Hash key = key % table size

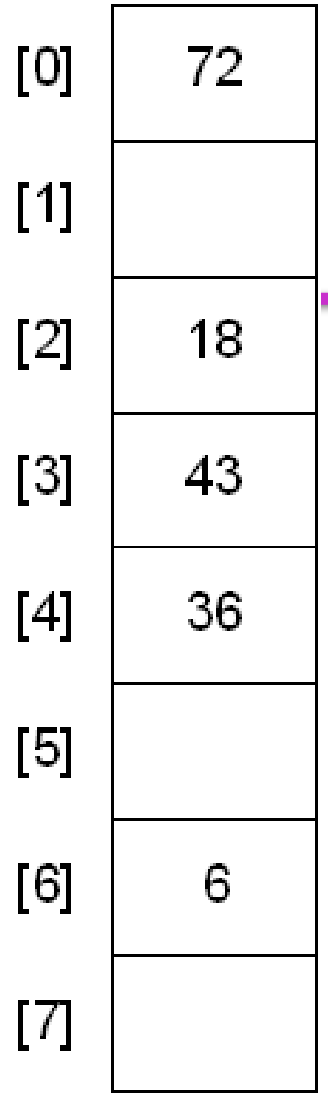
4 = 36 % 8

2 = 18 % 8

0 = 72 % 8

3 = 43 % 8

6 = 6 % 8



```python
hash_size = 10**4

def hash_function(x):
    return x % hash_size

class my_set:
    table = [0] * hash_size

    def insert(self, x):
        self.table[hash_function(x)] = 1

    def exists(self, x):
        return self.table[hash_function(x)] == 1

    def remove(self, x):
        self.table[hash_function(x)] = 0
```

```python
s = my_set()
s.insert(10)
s.insert(245)
print(s.exists(124))
s.insert(8653)
s.insert(12543252)
print(s.exists(10))
s.remove(10)
print(s.exists(10))
print(s.exists(12543252))
print(s.exists(12543251))
print(s.exists(3252))
```

```
False
True
False
True
False
True
```

توابع درهم سازی
روش ضرب

یک استراتژی دیگر روش ضرب است که دو مرحله دارد. ابتدا داده‌ی ورودی ضرب در یک عدد ثابت A (0<A<1.) ضرب می شود و سپس قسمت اعشار آن گرفته میشود. سپس این قسمت اعشاری در عدد صحیح m ضرب شده و قسمت صحیح آن گرفته می‌شود.

$$h(x) = \lfloor m(xA \ mode \ 1) \rfloor$$

- اهمیت انتخاب m کمتر
- معمولا m توانی از دو و A عددی گویای کمتر از $s/2^w$ $(0 < s < 2^w)$

برخورد (COLLISION)

- عدم وجود نگاشت یک به یک از یک مجموعه با کاردینالیتی C1 به مجموعه ای با کاردینالیتی C2 که C1>C2
- بنابراین در تابع درهمساز h به منظور نگاشت D به L که |L|<|D| است، حتما عناصر متمایزی از D هستند که به یک عنصر در L نگاشت می شوند. به این اتفاق **برخورد** گفته می شود.
- در عمل، ما دوست داریم از تابع درهمسازی‌ای استفاده کنیم که تعداد برخوردهای آن در سناریوهای مدنظر ما کمینه باشد.

Hash key = key % table size

4 = 36 % 8
2 = 18 % 8
0 = 72 % 8
3 = 43 % 8
6 = 6 % 8
2 = 10 % 8
5 = 5 % 8
7 = 15 % 8

روش مقابله با برخورد

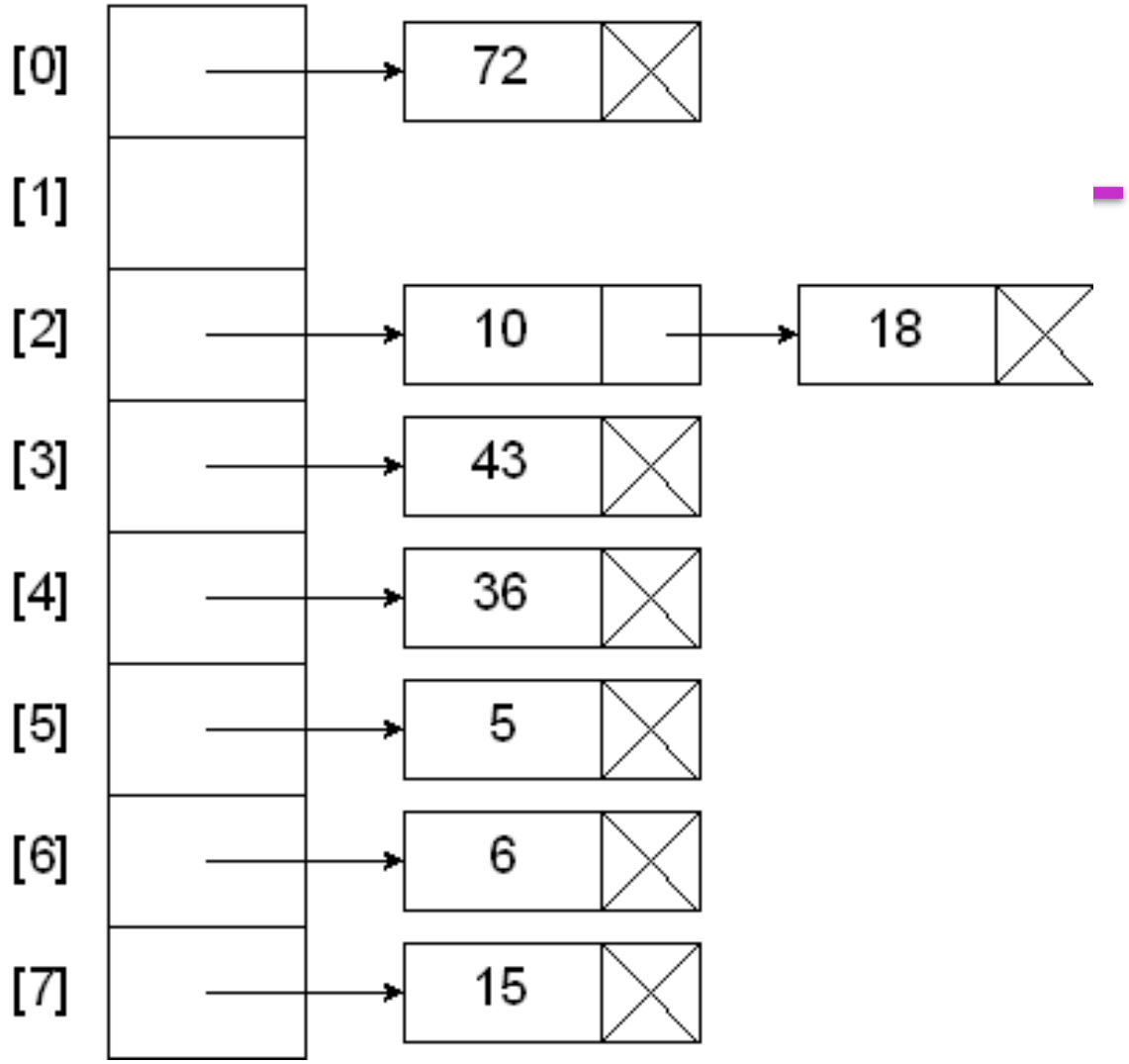
- زنجیره سازی
- روش آدرس دهی باز
- کاوش خطی
- کاوش مربعی

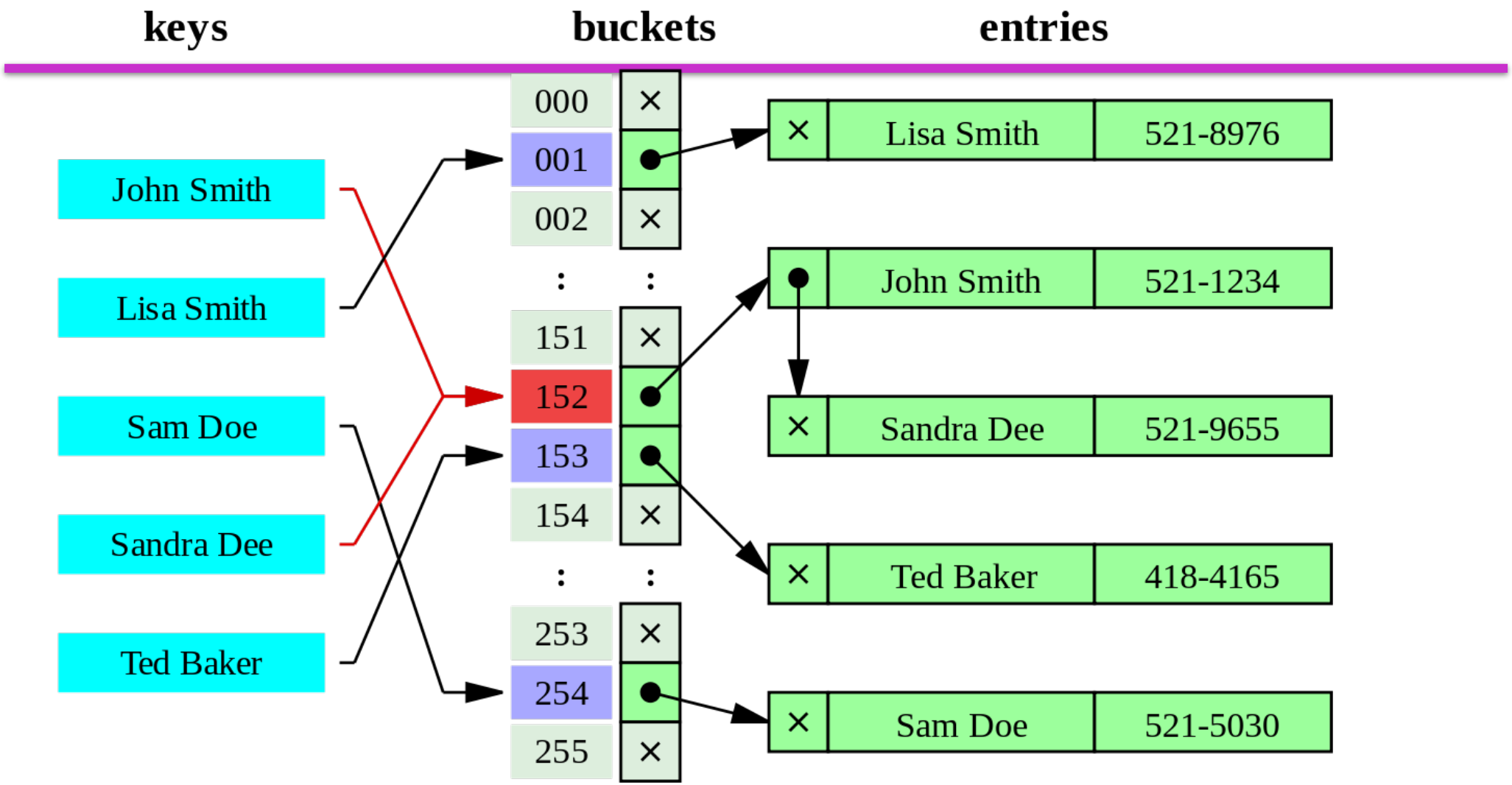
زنجیره‌سازی

در این روش، هر خانه‌ی جدول را یک لیست پیوندی می‌گیریم. حال هر عدد را که می‌خواهیم به خانه‌ای اضافه کنیم، خود عدد را در انتهای لیست آن خانه اضافه می‌کنیم. برای چک کردن وجود هم باید کل لیست آن خانه را بگردیم تا این که یا لیست تمام شود یا عدد مورد نظر را پیدا کنیم.

- Hash key = key % table size

```
4      = 36 % 8
2      = 18 % 8
0      = 72 % 8
3      = 43 % 8
6      =  6 % 8
2      = 10 % 8
5      =  5 % 8
7      = 15 % 8
```




keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
151
152
153
154
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

مزایا و معایب

- مشکل این روش داینامیک بودن سایز داده ساختار است.
- اگر طول یک لیست پیوندی خیلی زیاد شد می‌توانیم از هش و یا درخت دودویی جست‌وجو به جای لیست پیوندی استفاده کنیم.
- این روش شاید به نظر کارا نیاید اما دقت کنید که فرض کرده بودیم قرار نیست تعداد زیادی برخورد داشته باشیم.

```
class my_chained_set:
    table = [[] for _ in range(hash_size)]

    def insert(self, x):
        if not self.exists(x):
            self.table[hash_function(x)].append(x) # O(1)

    def exists(self, x):
        return x in self.table[hash_function(x)] # O(size of list)

    def remove(self, x):
        self.table[hash_function(x)].remove(x)  # O(size of list)
```

```
s = my_chained_set()
s.insert(10)
s.insert(245)
print(s.exists(10))

s.insert(10987)
print(s.exists(987))  # right answer!
print(s.exists(10987))
```

```
True
False
True
```