# Hashing

## The searching problem

### 直接地址法

例子：IPv4的32位地址，对于一个学校

- 可用地址数量(n)相对于总量(m)是非常小的($n \ll m$)，所以可以优化

### Hashing

- 通过一种映射，映射到特别小的空间，即开一个哈希表，在这个表里，进行查找
- 会出现冲突情况，所以要处理

**Load Factor/负载因子** 定义为 $\alpha = \frac{n}{m}$

## Collision Handling for Hashing

### Closed Address

- Each address is a linked list - 接链表
- 头插法效率更高，所以默认用头插法进行时间复杂度分析
- $\alpha = \frac{n}{m}$ （n可以大于m）

### Open Address

- 所有元素都存在哈希表里，不开辟新的空间
  - 无链表
  - $\alpha = \frac{n}{m} < 1$
- 冲突处理靠"rehashing"
- 探查(probing)序列可以看成1, 2, ..., m-1的一个排列

α很接近1的时候效率很低

- 常见探查函数
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
- Equally Likely Permutations
  - Each key is equally likely to have any of the m! Permutations of (1,2,...,m) as its probe sequence
  - Both linear and quadratic probing have only m distinct probe sequence, as determined by the first probe

## Cost Analysis of Hashing

## Closed Address

### Assumption - simple uniform hashing

> 需要做一个假设，假设你知道整体哈希表的情况，比方说大小m，元素n
>
> 只有定好场景之后，才能分析时间复杂度

假设等概率分布到任意地址

如果两个元素在同一个地址，会被链起来

链表平均长度有多长？$n/m$

### The average cost for an unsuccessful search

不存在的元素，认定会被哈希函数等概率映射到m个地址空间，所以查找情况都是链表遍历，对应平均长度，平均时间复杂度为$\Theta(1+n/m)$

**For successful search** (Assuming that $x_i$ is the $i^{th}$ element inserted into the table, $i=1,2,\cdots,n$)

- 对每个i，$x_i$被查找的概率是$\frac{1}{n}$
- 对给定$x_i$，成功搜索需要查看t+1次元素（t是在$x_i$之后被插入同一个哈希表的元素数量）

从而successful search的平均时间复杂度为

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

（注：在某个结点之后在同一位置插入的结点在其之前，假设有$t$个新结点，再包含本身，就是$t+1$）

计算上式，需要处理t，由于t是与被查找元素同一个地址的元素，基于等概率映射到每个地址的假设，有$t=\frac{1}{m}$，从而

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

（其中“1”是计算哈希值）

展开计算

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m}) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)\\ &= 1+\frac{1}{nm}\cdot\frac{n(n-1)}{2}\\ &= 1+\frac{n-1}{2m}\\ &= \Theta(1+\alpha)\end{aligned}$$

哈希表就只能考复杂度分析，所以一定要会算

## Open Address

**The average number of probes in an unsuccessful search is at most** $\frac{1}{1-\alpha}(\alpha = \frac{n}{m} < 1)$

- Assuming uniform hashing

第一次查找发现被占的概率是$\frac{n}{m}$

第二次查找要基于第一次查找的情况，只有第一次查找没找到才会进行第二次查找，会查找一个新的地址，相当于排除一个地址，因此，概率是$\frac{n-1}{m-1}$

第j$^{th}$(j > 1)查找发现位置被占的概率是$\frac{n-i+1}{m-i+1}$

所以探查次数不少于i的概率为（推到第$i-1$次）

$$\frac{n}{m}\cdot\frac{n-1}{m-1}\cdot\frac{n-2}{m-2}\cdot\cdots\cdot\frac{n-i+2}{m-i+2}\leq\left(\frac{n}{m}\right)^{i-1}=\alpha^{i-1}$$

The average number of probe is:（近似方法，无穷级数求和要注意有开地址哈希前提$\alpha < 1$）

$$\sum_{i=1}^{\infty}\alpha^{i-1}=\sum_{i=0}^{\infty}\alpha^{i}=\frac{1}{1-\alpha}$$

> *Introduction to Algorithms* p.1199

> *C.3 Discrete random variables* *1199*
>
> When a random variable $X$ takes on values from the set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$, we have a nice formula for its expectation:
>
> $$\begin{aligned} \mathrm{E}[X] &= \sum_{i=0}^{\infty} i \cdot \Pr\{X = i\} \\ &= \sum_{i=0}^{\infty} i(\Pr\{X \geq i\} - \Pr\{X \geq i+1\}) \\ &= \sum_{i=1}^{\infty} \Pr\{X \geq i\}\ , \end{aligned} \qquad \text{(C.25)}$$
>
> since each term $\Pr\{X \geq i\}$ is added in $i$ times and subtracted out $i-1$ times (except $\Pr\{X \geq 0\}$, which is added in 0 times and not subtracted out at all).

**The average cost of probes in an successful search is at most** $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}(\alpha = \frac{n}{m} < 1)$

- Assume uniform hashing

查找第$(i+1)^{th}$个被插入的元素的开销等同于在哈希表里只有$i$个元素时插入它（考虑查找过程，和插入过程完全一致），等价于在有$i$元素$m$这么大的哈希表里面查找失败（因为没有这个元素才要插入），此时$\alpha = \frac{i}{m}$，可以直接使用上面unsuccessful search的cost计算公式，所以cost是$\frac{1}{1-\frac{i}{m}} = \frac{m}{m-i}$.

所以成功查找的平均开销（即求期望）为

$$
\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{m-n+1}^{m}\frac{1}{i} \\
&\le \frac{1}{\alpha}\int_{m-n}^{m}\frac{dx}{x} \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}
\end{aligned}
$$

## Hash Function

- A good hash function satisfies the assumption of *simple uniform hashing*
  - Heuristic hashing functions
    - The division method: $h(k) = k \mod m$
    - The multiplication method: $h(k) = \lfloor m(kA \mod 1)\rfloor (0 < A < 1)$
  - No single function can avoid the worst case $\Theta(n)$
    - So "universal hashing" is preposed.
  - Rich resource about hashing function
    - Gonnet and Baeza-Yates: Handbook of Algorithms and Data Structures, Addison-Wesley, 1991.

# Amortized Analysis（平摊分析）

## Array Doubling - An Example

```
hashingInsert(HASHTABLE H, ITEM x)
    integer size = 0, num = 0;
    if size = 0 then
        allocate a block of 2 * size;
        move all item into new table;
        size = 2 * size;
    insert x into the table;
    num = num + 1;
return
```

（std::vector的实现方式）

## Worst-case Analysis

单次操作Worst-case是$O(n)$，但是因此就认为n次操作是$O(n^2)$就不合理，这个界太宽了

一个操作序列有n次操作，要看整体的开销。

$$c_i = \begin{cases} i, & \text{if } i-1 \text{ is exactly the power of 2} \\ 1, & \text{otherwise} \end{cases}$$

So the total cost is（直接求和放缩）：

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j < n + 2n = 3n$$

总体操作中有简单操作也有复杂操作，看平摊开销，让这个界紧一点。

## Amortized Analysis - Why?

- **Unusually expensive operations**
  - E.g., Insert-with-array-doubling
- **Relationship between expensive operations and ordinary ones**
  - Each piece of the doubling cost corresponds to some previous insert

## Amortized Analysis - How?

- **Amortized equation:**
  - amortized cost = actual cost + accounting cost

> “未雨绸缪，没等你开销先帮你记下来”

accounting - 会计

- **Design goal for accounting cost**
  - In any legal sequence of operations, the sum of the accounting costs is nonnegative

    $\sum \text{accounting} \ge 0$
  - The amortized cost of each operation is fairly regular, inspite of the wide fluctuate possible for the actual cost of individual operations

    尽管但看各个操作cost会有波动，整体平摊下来的cost很稳定规整。

## 例子

### Array Doubling

- Why non-negative accounting cost?
  - For any possible sequence of operations?

| | $Amortized$ | $Actual$ | $Accounting$ |
|---|---|---|---|
| $\text{Insert}(\text{normal})$ | $3$ | $1$ | $2$ |
| $\text{Insert}(\text{doubling})$ | $3$ | $k+1$ | $-k+2$ |

Insert(normal)的actual cost为1代表直接插入；Accounting cost的2代表为了之后扩张成2k的新空间准备自己摊到的那一份“储蓄”。为什么是2呢，因为这k个元素已经消耗掉了以前存储cost（前一半元素为后一半存储，一个动态过程），所以后面扩展的时候不光要存储自己这一半的cost还要存储之前替自己存储的前一半的cost（因为都需要被拷贝）

Insert(doubling)的actual cost为k+1代表开辟2k空间之后把原k个元素搬进去再加上新元素插入；Accounting cost中的-k代表消耗先前的Insert(normal)提前准备的储蓄，2则是作为未来扩展成4k的时候做的储备中平摊到它一个元素的那份。

## Multi-pop Stack

多次压栈，一次全部出栈

| | $Amortized$ | $Actual$ | $Accounting$ |
|---|---|---|---|
| Push | 2 | 1 | 1 |
| Multi-pop | 0 | $k$ | $-k$ |

## Binary Counter

一次操作：位翻转(bit flip)

0翻到1，下一次一定是1翻到0

| | $Amortized$ | $Actual$ | $Accounting$ |
|---|---|---|---|
| Set 1 | 2 | 1 | 1 |
| Set 0 | 0 | 1 | $-1$ |