哈希表

1.1 哈希表的一个应用，是 Python 的数据结构：字典。大家可以阅读一下构建 Python 字典的设计需求：http://svn.python.org/projects/python/trunk/Objects/dictnotes.txt （这不是必须的）。其中，提到了字典的一个应用场景：

```
Membership Testing
    Dictionaries of any size.  Created once and then rarely changes.
    Single write to each key.
    Many calls to __contains__() or has_key().
    Similar access patterns occur with replacement dictionaries
        such as with the % formatting operator.
```

请问下述那个描述符合这个场景？

A 创建后多次插入，之后几乎是查找操作

B 创建后多次插入，之后全是查找操作

C 插入/删除与查找操作出现的次数几乎一样多

D 轮流进行插入/删除与查找操作

1.2 仍然是上题题干。考虑要创建一个哈希表来实现字典。请问如何考量这个哈希表表长的设计？

A 初始选定一个较小的哈希表长度，后续以 2 倍进行扩增

B 初始选定一个较小的哈希表长度，后续以 4 倍进行扩增

C 初始选定一个较大的哈希表长度，后续以 **2** 倍进行扩增

D 初始选定一个较大的哈希表长度，后续以 4 倍进行扩增

1.3 课本 11.1-1 假设一动态集合 S 用一个长度为 m 的直接寻址表 T 来表示。请给出一个查找 S 中最大元素的过程。你所给的过程在最坏情况下的运行时间是多少？

只需要从 T 末尾开始查找，找到的第一个不为 NIL 的元素即为最大元素。

最坏情况下，需要遍历整个直接寻址表 T，因此时间复杂度为 O(m).

1.4 课本 11.2-3 Marley 教授做了这样一个假设，即如果将链模式改动一下，使得每个链表都能保持已排好序的顺序，散列的性能就可以有较大的提高。Marley 教授的改动对成功查找、不成功查找、插入和删除操作的运行时间有何影响？

假设这个排序是递增的。

1. 成功查找操作：时间可能增大也可能减小。在同一条链表中查找小元素需要遍历的更少，大元素需要遍历的更多。
2. 不成功查找操作：找到比待查值大的数就可以提前终止搜索过程，会加快。
3. 插入操作：插入操作的时间增加。由于链表保持有序，插入一个新元素需要在链表中遍历，找到正确的位置并进行插入。而原先无需遍历。
4. 删除操作：与成功查找相同，时间可能增大也可能减小。

链表不能二分，如果这里用内存连续的数据结构可以大幅提升查找性能。

1.5 课本 11.3-4 考虑一个大小为 m=1000 的散列表和一个对应的散列函数 $h(k) = \lfloor m(kA \bmod 1) \rfloor$，其中 $A = \frac{\sqrt{5}-1}{2}$，试计算关键字 61、62、63、64 和 65 被映射到的位置。

```
from math import floor, sqrt
A = (sqrt(5) - 1) / 2
for i in range(61, 66):
    hi = floor(1000 * ((i * A) % 1))
    print(i, hi)

# 61 700
# 62 318
# 63 936
# 64 554
# 65 172
```

1.6 课本 11.4-1 考虑用开放寻址法将关键字 10、22、31、4、15、28、17、88、59 插入到一长度为 $m = 11$ 的散列表中，辅助散列函数为 $h'(k) = k$。试说明分别用线性探查、二次探查 $(c_1 = 1, c_2 = 3)$ 和双重散列 $(h_1(k) = k, h_2(k) = 1 + (k \bmod (m-1)))$ 将这些关键字插入散列表的过程。

```
i = [10, 22, 31, 4, 15, 28, 17, 88, 59]
m = 11
print(list(map(lambda x: x % m, i)))

# [10, 0, 9, 4, 4, 6, 6, 0, 4]
```

1.6.1 线性探查

将重复数字后延。最终这些数的插入位置是：$[10, 0, 9, 4, 5, 6, 7, 1, 8]$

1.6.2 二次探查

在 $(c_1 = 1, c_2 = 3)$ 条件下，可以算出偏移量：

```
m = 11
c_1 = 1
c_2 = 3
for i in range(1, 7):
    print(i, (c_1 * i + c_2 * i * i) % m)

# 1 4
# 2 3
# 3 8
# 4 8
# 5 3
# 6 4
```

在第 5 位的 4 偏移到 $4 + 4 = 8$，第 7 位的 6 偏移到 $6 + 4 = 10$ 再偏移到$(10 + 3) \bmod 11 = 2$，第 8 位的 0 偏移到 $0 + 4 = 4$ 再偏移到$4 + 3 = 7$，第 9 位的 4 偏移到 $4 + 4 = 8$ 再偏移到 $(8 + 3) \bmod 11 = 0$再偏移到$0 + 8 = 8$再偏移到$(8 + 8) \bmod 11 = 5$，最终位置应该是 $[10, 0, 9, 4, 8, 6, 2, 7, 5]$

1.6.3 双重散列

$(h_1(k) = k, h_2(k) = 1 + (k \bmod (m-1)))$，直接上代码

```
from functools import lru_cache
origin = [10, 22, 31, 4, 15, 28, 17, 88, 59]
m = 11
ans = [-1] * m
@lru_cache
def h2(k):
    return 1 + (k % (m - 1))
@lru_cache
def h(k, i):
    return (k + i * h2(k)) % m
for k in origin:
    times = 0
    while ans[h(k, times)] != -1:
        times += 1
    ans[h(k, times)] = k
print(ans)

# [22, -1, 59, 17, 4, 15, 28, 88, -1, 31, 10]
```

1.7 考虑到英文字符串包含了 27 个字符（26 个字母+空格符），构造一种简单的除法哈希函数：$h(k) = k \bmod m$，其中 m 为哈希表大小，k 为一个字符串所有字母数值的和（这里字符的值可采用 ASCII 码）。请问这种哈希方法是否足够好？

不够好。假设 m 远大于 ASCII 码范围，那么对于较短的字符串，其所有字母数值的和不够大，会聚集在 hash 表的前段。并且对于字符串的交换，不会改变映射的位置，导致碰撞较高。

1.8 对于哈希表，我们希望它的性能足够好（插入和查找速度接近 O(1)），同时我们也需要保证其正确性（即每个 key 都能被正确地插入哈希表中）。因此我们为哈希表提供两个操作：冲突处理与动态调整。其中动态调整是指根据插入的情况，动态扩大哈希表的大小。请问如下论述哪些是正确的：

1.8.1 (a)我们仅需要动态调整即可保证哈希表的正确性与性能；

F

1.8.2 (b)动态调整保证了正确性，但是不保证性能；

F

1.8.3 (c)冲突处理保证了性能，不保证正确性

F

1.8.4 (d)两种都是必须的，从而保证哈希表的正确性与性能

T

动态调整保证了性能，冲突处理保证了正确性

1.9 动态调整：假设原哈希表大小为 m，包含了 n 个元素。此时要扩大到m'，请问至少需要多少的操作？（提示：用 O 渐进符号表示，注意用链接法和用开放寻址法解决冲突情况是不同的，要分别讨论）

对于动态调整来说，涉及到哈希表的扩容操作。在哈希表的扩容过程中，主要需要考虑两种不同的冲突解决方法：链接法和开放寻址法。

1. 链接法 每个插入操作的时间复杂度是 O(1)，因为只需要在对应槽位的链表中插入元素。总操作次数$O(n)$
2. 开放寻址法 每个插入操作可能需要多次探查，直到找到一个可用的槽位。如果扩容后的表大小为m'，则每个插入操作最坏情况下可能需要 O(m')的操作次数。因此，扩容的总操作次数为$O(nm')$。

1.10 课本 11-2 假设有一个含 n 个槽的散列表，向表中插入 n 个关键字，并用链接法来解决冲突问题。每个关键字被等可能地散列到每个槽中。所有关键字被插人后，设 M 是各槽中所含关键字数的最大值。读者的任务是证明 M 的期望值$E[M]$的一个上界为$O\left(\frac{\lg n}{\lg\lg n}\right)$。其中 a, b, c 为要求作答，d、e 为附加题（难）。

1.10.1 a.证明：正好有 k 个关键字被散列到某一特定槽中的概率Q_k为

$$Q_k = \left(\frac{1}{n}\right)^k \left(1-\frac{1}{n}\right)^{n-k} \binom{n}{k}$$

这里使用二项式定理：

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

令 $x = 1-\frac{1}{n}$ 和 $y = \frac{1}{n}$，然后将其代入二项式定理：

$$\left(1-\frac{1}{n}+\frac{1}{n}\right)^n = \sum_{k=0}^{n} \binom{n}{k} \left(1-\frac{1}{n}\right)^{n-k} \left(\frac{1}{n}\right)^k$$

化简得：

$$1 = \sum_{k=0}^{n} \binom{n}{k} \left(1-\frac{1}{n}\right)^{n-k} \left(\frac{1}{n}\right)^k$$

因为每个关键字等可能地散列到每个槽中，所以我们关心的是其中的一项：

$$Q_k = \binom{n}{k} \left(1-\frac{1}{n}\right)^{n-k} \left(\frac{1}{n}\right)^k$$

这就是要证明的概率公式。

1.10.2 b.设 P：为 M=k 的概率，即包含最多关键字的槽中有 k 个关键字的概率。证明：$P_k \le nQ_k$。

$$P_k = \sum_{j=k}^{n} Q_j$$

$$P_k = \sum_{j=k}^{n} \binom{n}{j} \left(1 - \frac{1}{n}\right)^{n-j} \left(\frac{1}{n}\right)^{j}$$

$$P_k = \sum_{j=k}^{n} \frac{n!}{j!(n-j)!} \left(1 - \frac{1}{n}\right)^{n-j} \left(\frac{1}{n}\right)^{j}$$

$$P_k = \sum_{j=k}^{n} \frac{n(n-1)(n-2)...(j+1)}{(n-j)(n-j-1)...1} \left(1 - \frac{1}{n}\right)^{n-j} \left(\frac{1}{n}\right)^{j}$$

$$P_k = \sum_{j=k}^{n} \frac{n(n-1)(n-2)...(j+1)}{(n-j)(n-j-1)...1} \left(1 - \frac{1}{n}\right)^{n-j} \frac{1}{n^j}$$

$$P_k = \sum_{j=k}^{n} \frac{n!}{(n-j)!j!} \left(1 - \frac{1}{n}\right)^{n-j} \frac{1}{n^j}$$

$$P_k = \sum_{j=k}^{n} \binom{n}{j} \left(1 - \frac{1}{n}\right)^{n-j} \frac{1}{n^j}$$

$$P_k = \sum_{j=k}^{n} Q_j$$

由于 Q_j 是非负的，所以 $P_k \leq nQ_k$。

1.10.3 c.应用斯特林近似公式 $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right)$**来证明：**$Q_k \leq \frac{e^k}{k^k}$。

首先，将斯特林近似公式代入 Q_k 的表达式：

$$Q_k = \binom{n}{k} \left(1 - \frac{1}{n}\right)^{n-k} \left(\frac{1}{n}\right)^{k}$$

$$Q_k = \frac{n!}{k!(n-k)!} \left(1 - \frac{1}{n}\right)^{n-k} \left(\frac{1}{n}\right)^{k}$$

$$Q_k = \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right)}{k!(n-k)!} \left(1 - \frac{1}{n}\right)^{n-k} \left(\frac{1}{n}\right)^{k}$$

然后，简化 Q_k 的表达式：

$$Q_k = \frac{\sqrt{2\pi n}}{\sqrt{2\pi k}\sqrt{2\pi(n-k)}} \cdot \frac{\left(\frac{n}{e}\right)^n}{\left(\frac{k}{e}\right)^k \cdot \left(\frac{n-k}{e}\right)^{n-k}} \cdot \left(1 + \Theta\left(\frac{1}{n}\right)\right) \cdot \left(1 - \frac{1}{n}\right)^{n-k} \cdot \left(\frac{1}{n}\right)^{k}$$

$$Q_k = \frac{\sqrt{2\pi n}}{\sqrt{2\pi k}\sqrt{2\pi(n-k)}} \cdot \frac{\left(\frac{n}{e}\right)^n}{\left(\frac{k}{e}\right)^k \cdot \left(\frac{n-k}{e}\right)^{n-k}} \cdot \sqrt{\frac{e}{n}} \cdot \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

$$Q_k = \frac{1}{\sqrt{2\pi k(n-k)}} \cdot \frac{\left(\frac{n}{e}\right)^{n-k}}{\left(\frac{k}{e}\right)^k} \cdot \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

$$Q_k = \frac{1}{\sqrt{2\pi k(n-k)}} \cdot \frac{n^{n-k}}{k^k} \cdot \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

1.11 子序列匹配问题（本题必做 a、b、e，选做 c、d）。假设有序列$T[1..n]$，代表一个序列 T 有 n bits（n 位）。我们考虑一个序列 P[1..m]，m < n，是否为 T 的子序列，即是否存在 i = j-m+1，使得 T[i..j] = P[1..m]。我们假设，对于操作 O(lg n)比特的序列，可以在固定时间 c 里完成，但对于操作这个数量级以上的比特位时，其操作时长不能看作固定时间。

1.11.1 (a) 假设存在哈希函数 h(x)，对于所有长度为 m bits 的序列，均可在 O(m)时间内完成操作，且对于序列$x \neq y$，均有$h(x) \neq h(y)$。证明，在 O(mn)时间里，可以验证 P 是否是 T 的子序列。（必做，易）

使用滚动 hash。由于 $x \neq y$，均有$h(x) \neq h(y)$，因此无需考虑哈希冲突。

首先，需要选择一个合适的哈希函数 h(x)，并定义一个滚动哈希函数 r(x)。设序列$X = x_1 x_2 ... x_k$的哈希值为$h(X)$，则$r(x_{i+1} x_{i+2} ... x_{i+k})$的计算可以利用$r(x_i ... x_{i+k-1})$的结果，即：

$$r(x_{i+1} x_{i+2} ... x_{i+k}) = \left(r(x_i ... x_{i+k-1}) - x_i \cdot b^{k-1}\right) \cdot b + x_{i+k}$$

其中，b 是基数，一般选择一个合适的素数。

现在，使用滚动哈希来在 O(mn)时间内验证 P 是否为 T 的子序列。

1. 计算 P 和 T 的初始哈希值，分别记为h_P和h_T。这可以在 O(m)时间内完成。
2. 在 T 上进行滚动哈希的过程。从左到右依次考虑 T 的每个长度为 m 的子序列，计算其哈希值。
3. 比较当前子序列的哈希值和 P 的哈希值h_P是否相等。如果相等，则说明找到了一个匹配。
4. 如果不相等，将哈希值向右滚动一位，计算下一个子序列的哈希值。
5. 重复步骤 3 和步骤 4，直到遍历完整个序列 T。

这样的滚动哈希算法可以在 O(mn)时间内完成，因为每个滚动哈希的计算只需要 O(1)时间。因此，整个过程的时间复杂度为 O(mn)。

1.11.2 (b) 考虑一个哈希函数，p 是$[2, cn^4]$里的一个质数，$h_p = x \bmod p$。对于一个 i, $1 \le i \le n-m+1$,令$x = T[i..(i+m-1)]$。证明，可以找到一个合适的 c，使得$x \neq p$则必有$h_{p(x)} \neq h_{p(P)}$。这里要用到两个定理：（1）对于正整数 x，其最多有 lg x 个质数因子；（2）在[2, x]里，质数的个数为$\frac{x}{\lg(x)}$。（必做，中）

首先，注意到 h_p 的取值范围是 [0, p-1]。根据题目条件，p 是 [2, cn^4] 范围内的质数，而 n 表示序列 T 的长度。

考虑哈希函数 $h_p = x \mathrm{mod}\, p$。令 $x = T[i..(i+m-1)]$，可以找到一个合适的 c，使得 $x \neq p$ 则必有 $h_p(x) \neq h_p(P)$。

考虑哈希函数 h_p，对于任意的 x，$h_p(x)$ 的取值范围是 [0, p-1]。

如果 $x \neq p$，则 $h_p(x) \neq 0$。

由于 $h_p(x)$ 的取值范围是 [0, p-1]，如果 $x \neq p$，那么至少有$p-1$ 个不同的可能的哈希值。因此，我们可以选择一个足够大的 c，使得$p-1 > c(n-m+1)$。

这样，对于任意 i，$1 \leq i \leq n-m+1$，取$x = T[i..(i+m-1)]$，如果 $x \neq p$，则至少有 $c(n-m+1)+1$ 个不同的哈希值，而这个数量大于 T 中所有可能子序列的数量（$n-m+1$ 个）。因此，至少存在一个哈希值不会与$h_p(P)$ 相等。

所以，通过选择足够大的 c，可以确保 $x \neq p$ 则必有 $h_p(x) \neq h_p(P)$。

1.11.3 (c)对于 b 中的h_p哈希函数，请问计算时间是多少？（注意，这里不是固定时间 c）（选做，难）

1.11.4 (d)对于$1 \leq i \leq n-m$，给出一种算法，在已知$h_p(A[i..(i+m-1)])$的情况下，可在固定时间 c 里，计算出$h_p(A[(i+1)..(i+m)])$。（选做，难）

1.11.5 (e)根据(a)-(d)方法和结论，给出一种算法，使得在 O(n)时间里，能够知道 P 是否为 T 的子序列。（必做，中）

1. 计算哈希值：对于 P 和 T 中的前 m 个元素，计算它们的哈希值。
2. 验证：在 T 中逐步滚动计算每个长度为 m 的子序列的哈希值，并与

P 的哈希值比较。如果存在任何一个子序列的哈希值与 P 的哈希值相等，则 P 是 T 的子序列。

整个算法的时间复杂度为 O(n)。这是因为计算哈希值和验证哈希值的过程都是 O(1) 操作，总共有 n-m+1 个子序列需要验证。

从本题可以看出，哈希函数应用广泛，在多个场景均可构建高性能算法。可把本题算法与动态规划里提到的子序列算法进行比较。