数据结构和算法
(Python描述)

郭 炜

微信公众号

微博：http://weibo.com/guoweiofpku

学会程序和算法，走遍天下都不怕!

讲义照片均为郭炜拍摄

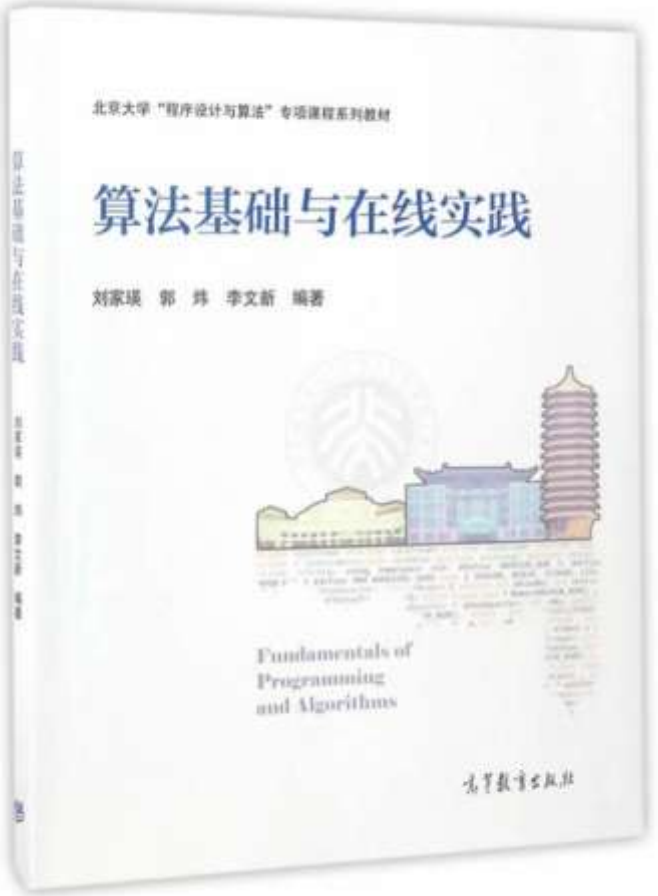
配套教材：

高等教育出版社

《算法基础与在线实践》

刘家瑛 郭炜 李文新 编著

本讲义中所有例题，根据题目名称在
http://openjudge.cn
“百练”组进行搜索即可提交



散列表（哈希表）

散列表的概念

散列表要解决的问题

如何尽量在O(1)的时间内，对数据集进行查询，插入、删除

- 如果关键字是非负整数，将记录存放在列表L中，关键字为n的记录，就存放在L[n]
- n太大时浪费空间。比如身份证号是18位整数，而全国只有不超过15亿人
- 设计一个散列函数（哈希函数)h, 关键字为key的记录，存放在L[h(key)]。h(key)是非负整数。比如 h(key) = key % 2,000,000,000 用来处理身份证号。h(key)的值称为散列值。

散列表要解决的问题

如何尽量在O(1)的时间内，通过关键字查询到想要的记录

- 即便关键字key不是整数，是字符串或其他对象，也可以定义散列函数将其映射到一个整数

- 散列函数h是取值范围大的集合到取值范围小的集合的映射，必然会有不同key有相同h(key)值导致的冲突需要处理（如果事先知道所有需要散列的元素，则有可能做到没有冲突）

散列函数（哈希函数）

散列函数的应用

- 为文件计算一个散列值。检查文件是否被破坏，即可以通过计算其散列值来进行
- 用来做数据加密。用户名和密码在后台不是直接存放，而是存放其散列函数值。验证用户名和密码，就是验证其散列值。
- 用于各种网络传输和安全协议
- Python的dict和set都是散列表

散列函数的设计要求

- 1) 函数he(key)简单，计算速度快
- 2) h(key)的结果尽量均匀分布
- 3) h(key)的结果尽量减少冲突
- 4) h(key)的结果可以覆盖整个存储区，避免有些存储单元被浪费

h(key) 的结果越没有规律越能满足后3条。没规律：相似的key的h(key)不相似。key中的信息被h(key)用到越多（比如每个二进制bit都有用）,越能满足后3条。

常见散列函数设计思路

- 数字分析法

如果关键字是一些已知的整数，可以抽取其中的若干位作为散列函数值，可能做到不冲突。

key	h(key)
00013445	34
00074842	44
88472742	24
77298402	80

常见散列函数设计思路

- 折叠法

将较长关键字切成几段，然后合并。例如对10位整数，每三位分为一段，四段相加去掉进位的结果就是散列函数值(位于[0,999])。

key	h(key)
2145673873	2+145+673+873 = 1693

常见散列函数设计思路

- 平方取中法

取整数关键字的平方中的若干位，作为散列函数值

key	h(key)
2145673873	2145673873^2 = 4603916369274820129

用计算机做运算，应当取二进制形式的若干位，而不是十进制形式的若干位，运算速度更快。平方取中法的随机性比较突出。

常见散列函数设计思路

- 除余法

  适用于关键字 key 是整数。
  散列表长度m取值为2的幂
  找到不大于m的最大质数P， h(key) = key % P

  也可以取更大的P, h(key)取 key % P 结果去掉最低若干个二进制位后的结果

计算速度快，但是有规律（连续整数除余结果连续，容易冲突）

常见散列函数设计思路

- 基数转换法

将整数的十进制表示形式，看作是一个r进制数（r是质数），然后再转换成十进制形式

Key：335647　h(key)：1211620

$(335647)_{13} = (1211620)_{10}$

然后再除余，或者折叠

常见散列函数设计思路

- key为字符串的情况

一个字符看作一个整数（编码值），通过基数转换法把字符串转换成整数：

```
def stringHash(s):
        h = 0
        for c in s:
                h += h * 31 + ord(c)
        return h
```

然后再对整数除余或折叠

冲突消解

冲突消解

- 难以避免不同key的哈希值相同
- 发生上述现象时的处理，即为冲突消解
- 冲突消解法分为两类：

1) 内消解：只使用散列表基本存储区（一个列表）解决冲突
2) 外消解：用散列表基本存储区之外的额外空间解决冲突

冲突内消解

- 开地址法和探查序列

在散列表中插入关键字为key的数据，发现冲突，即h(key)的位置已经被占据，则要为待插入数据找一个空的位置存放。找空位置的策略，称为探查方法。空位置的候选序列，称为探查序列。

定义整数序列：$D = d_0,d_1,d2.....$ $d_0 = 0$

则探查序列为：

$H_i = (h(key) + d_i) \bmod P$ P为不超过表长度的最大质数

H_i 若已经被占据，就考虑放在H_{i+1}（i从0开始）

冲突内消解

- 线性探查

$H_i = (h(key) + d_i) \bmod P$　　P为不超过表长度的最大质数

D = 0,1,2,…. 称为线性探查，即插入时发现冲突，则顺序往后看找第一个空闲的列表单元存放待插入的数据。

冲突内消解

- 双散列探查

设计另外一个散列函数g,称为再散列函数。

令 $d_i = i * g(key)$　　例如：$g(key) = key \% 5 + 1$

跳着探查，比线性探查更能避免元素聚集

此时，$H_i = (h(key) + i*g(key)) \bmod P$　　(i从0开始)

散列表检索指定key的元素

- 1) 求出h(key)，作为第一个探查位置
- 2) 若探查位置为空槽，则元素不存在，检索失败
- 3) 若探查位置为满槽或闲槽元素，则看该元素的key是否一致。如果一致，则检索成功
- 4) 若3）尚未检索成功，则按照探查序列找到下一个探查位置，转2)

- 槽分为三类：

空：从来没放过元素
闲：曾经放过元素后来又删了，现在没放元素
满：放着一个元素

散列表中删除指定key的元素

- 1) 按照检索办法找到关键字为key的待删除元素
- 2) 在该元素的位置做删除标记，使之成为闲槽（不可将其标记为空，因为标记为空可能切断了探查序列，导致后序检索和key有相同哈希值的元素失败)

检索时，在探查序列碰到闲槽，则也要继续找探查序列的下一个元素

插入元素时，如果按探查序列找到了带删除标记的元素，则将新元素插入在此

冲突外消解

- 溢出区方法

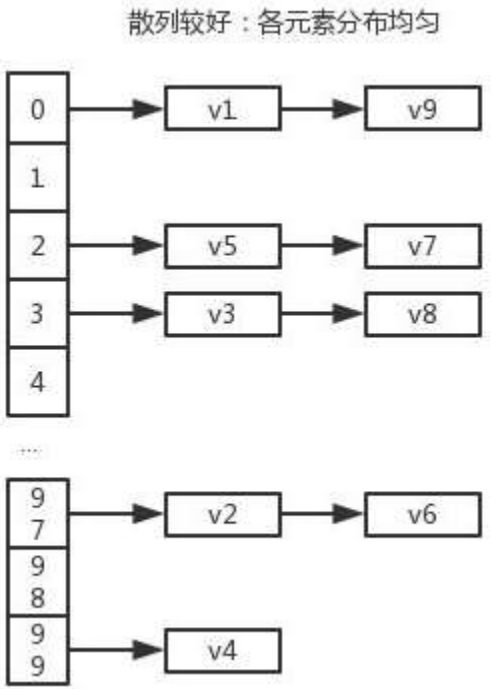
用另外一个列表顺序存放冲突的数据。查询时先看散列值的位置，没找到则到溢出区顺序查找。

冲突多时时间接近线性。

冲突外消解

- 桶散列

基本存储区不放元素，而是放链表的开头指针。所有散列值相同的元素，都放在同一个链表(或列表)，顺序查找。



散列表的装载因子

$$\text{装载因子 } a = \frac{\text{满槽数目+闲槽数目}}{\text{总槽数}}$$

装载因子会影响冲突频率

装载因子小于0.75时，散列表查找、插入、删除性能基本是O(1)

Python的散列表选择了冲突内消解，而且装载因子大于2/3 就会重新分配空间。

Python中的散列表

对象作为字典的键或集合的元素

- 默认情况下，自定义类的对象，可以作为集合元素或字典的键，被作为集合元素或字典键的，是对象的id，因此意义不大

```
class A:
    def __init__(self,x):
        self.x = x

a,b = A(5),A(5)          #两个A(5)不是同一个，因此a和b的id不同
dt = {a:20,A(5):30,b:40}          #三个元素的键id不同，因此在不同槽里
print(len(dt),dt[a],dt[b])       #>>3 20 40
print(dt[A(5)])                   #runtime error
```

对象作为字典的键或集合的元素

- 集合和字典都是哈希表，可哈希的类的对象才可以作为集合的元素或者字典的键
- 一个类，有__hash__方法，即为可哈希。自定义类的默认__hash__方法根据对象id算哈希值，哈希值是个整数
- __hash__函数返回值相同的两个对象a,b，在集合中放在同一个槽（单元）里。若a==b成立，则只能保留一个（字典的键类似处理）；若不成立，可以都保留。一个槽可能放多个对象，查找的时候，根据被查找元素的哈希值O(1)时间找到槽，然后再到槽里用==顺序查找。哈希值不同的对象，放在不同的槽里面。
- 如果为自定义类重写__eq__方法，则其__hash__方法会被Python自动变成None，其变成不可哈希。也可以同时重写__eq__和 __hash__

对象作为字典的键或集合的元素

- 类的__hash__方法示例

```
x = 23.1
print(x.__hash__(),23.1.__hash__())
#>>2305843009213726 95 2305843009213726 95
x = 23
print(x.__hash__(),hash(23))    #>>23 23
x = (1,2)
print(x.__hash__(),(1,2).__hash__(),hash(x))
#>>3713081631934410656 3713081631934410656 3713081631934410656
x = "ok"
print(x.__hash__(), "ok".__hash__())
#>>-4237608756544806 03 -4237608756544806 03
```

对象作为字典的键或集合的元素

- 为自定义类重写__eq__和__hash__方法，可以做到用对象的值作为集合元素或字典的键

```
class  A:
    def __init__(self,x):
        self.x = x
    def __eq__(self,other):
        if isinstance(other,A): #判断other是不是类A的对象
            return self.x == other.x
        elif isinstance(other,int): #如果other是整数
            return self.x == other
        else:
            return False
    def __hash__(self):
        return self.x
```

对象作为字典的键或集合的元素

```
a = A(3)
print(3 == a)                          #>>True
b = A(3)
d = {A(5):10,A(3):20,a:30}
print(len(d),d[a],d[b],d[3])           #>>2 30 30 30
```