Course 7：散列表

（一）散列表的基本概念

在之前所讲的线性结构、树表结构的查找方法中，都是以关键字的比较为基础的。因此，如果我们在查找过程中只考虑各元素关键字之间的相对大小，则查找的时间和表的长度有关。当结点个数很多时，要大量地与无效结点的关键字进行比较，致使查找速度很慢。

如果我们能在元素的存储位置和其关键字之间建立某种直接关系，那么在进行查找时，就无需做比较或做很少次的比较。直接由关键字找到相应的记录，这就是**散列查找法**（Hash Search）的思想

散列查找通过对元素的关键字值进行某种运算，直接求出元素的地址，即使用了关键字到地址的直接转换方法，而不需要反复比较

散列法中的常用术语：

1.散列函数和散列地址

在记录的存储位置p和其关键字key之间建立一个确定的对应关系H，使得p = H（key），称这个对应关系H为散列函数，p为散列地址

2.散列表

一个有限连续的地址空间，用以存储按散列函数计算得到相应散列地址的数据记录。通常，散列表的存储空间是一个一维数组，散列地址是数组的下标。

3.冲突和同义词

对不同的关键字，可能得到相同的散列地址，即：H(key1) = H(key2),where key1 ≠ key2,这种现象称为冲突。具有相同函数值的关键字对该散列函数来说称作同义词，key1与key2互称为同义词。

通常，散列函数是一个多对一的映射，冲突往往不可避免。因此问题归结为：如何构造散列函数；如何处理冲突

（二）散列函数的构造方法

构造散列函数的方法很多，应根据具体问题选用不同的散列函数。通常考虑：散列表的长度；关键字的长度；关键字的分布情况；计算散列函数所需的时间；记录的查找频率

一个好的散列函数，应该遵循：（1）函数计算要简单，每一关键字只能有一个散列地址与之对应；函数的值域需要在表长的范围内，计算出的散列地址的分布应均匀，尽可能减少冲突

1.数字分析法

如果事先知道关键字集合，且每个关键字的位数（n）比散列表的地址码位数多，则可以从关键字中提取数字分布比较均匀的若干位作为散列地址

eg 有80个关键字为8位十进制数的记录，散列表的表长为100，则可取两位分布均匀、尽量避免冲突的十进制数组成散列地址

2.平方取中法

一个数平方后的中间几位数和数的每一位都相关，如果取关键字平方后的中间几位或其组合作为散列地址，则使随机分布的关键字得到的散列地址也是随机的

3.折叠法

关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为散列地址。

根据数位叠加的方式，可以把折叠法分为移位叠加和边界叠加两种。移位叠加是将分割后每一部分的最低位对齐，然后相加;边界叠加是将两个相邻的部分沿边界来回折看，然后对齐相加。

适用情况：适合于散列地址的位数较少，而关键字的位数较多，且难于直接从关键字中找到数值较分散的几位

4.除留余数法

假设散列表表长为m，选择一个不大于m的数p，用p去除关键字，除后所得余数为散列地址：H(key) = key % p

这个方法的关键是选取适当的p，一般情况下，可以选p为小于表长的最大质数

该方法应用较广。它不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模，这样能够保证散列地址一定落在散列表的地址空间中

（三）处理冲突的方法

在实际应用中，很难完全避免发生冲突，所以需要一个有效的处理冲突的方法。创造散列表和查找散列表都会遇到冲突，两种情况下处理冲突的方法应该一致。下面以创建散列表为例说明处理冲突的方法

处理冲突的方法与散列表本身的组织形式有关。按组织形式的不同，通常分为“开放地址法和链地址法

1.开放地址法chaining

把记录都存储在散列表数组中，当某一记录关键字key的初始散列地址H0 = H(key)发生冲突时，以H0为基础，采取合适方法计算得到另一个地址H1，如果H1仍然发生冲突，以之为基础再求下一个H2...以此类推，直到Hk不冲突为止，并将Hk作为该记录在表中的散列地址

通常把寻找”下一个“空位的过程称为”探测“：

H_i = (H(key)+di)%m m为散列表表长，d为增量序列。根据d取值的不同，可以分为以下3种探测方法：

（1）线性探测法linear probing

d_i = 1，2，3，...m-1

这种探测方法可以将散列表假想成一个循环表，发生冲突就从冲突地址的下一单元顺序寻找空单元

（2）二次探测法quadratic probing

d_i = 1,-1,4,-4,.....k^2,-k^2，k<=m/2

（3）伪随机探测法

d_1 = 伪随机数序列

（4）双重散列double hashing

如果发生冲突，就使用第二个散列函数来计算下一个槽位的位置，直到找到一个空槽位或者达到散列表的末尾

步骤：使用第一个散列函数计算关键字key的初始散列值。如果槽位为空，则将关键字key插入到该槽位中；如果槽位不为空，表示发生冲突，这时采用第二个散列函数计算关键字key的步长step。然后，我们将在原槽位的基础上跳过step个槽位，继续在散列表中查找下一个槽位，重复该过程直到找到一个空槽位。

2.链地址法

链地址法的基本思想是：把具有相同散列地址的记录放在同一个单链表中，称为同义词链表。有m个散列地址就有m个单链表，同时用数组HT存放各个链表的头指针。凡是散列地址为i的记录都以结点方式插入到以HT中的对应单链表中

3.再散列rehashing

当散列表的装载因子（load factor，填入表中元素个数比散列表的大小）超过一定阈值时，进行扩容操作，重新调整散列函数和散列桶的数量，以减少冲突的概率

4.建立公共溢出区public overflow area

将冲突的元素存储在一个公共的溢出区域，而不是在散列桶中。在进行查找时，需要遍历溢出区域

（三）程序实现：HashTable类（字典类）

1.构建简单的散列函数

采用字符位置作为权重因子，以ord方法生成散列函数值

```
def hash(a_string, table_size):
    sum = 0
    for pos in range(len(a_string)):
        sum = sum + (pos+1) * ord(a_string[pos])

    return sum%table_size
```

2.总程序逻辑

使用两个列表创建HashTable类，以此实现抽象数据类型：slots存储键，data存储值。两个列表中的键与值一一对应

hashfunction采用简单的取余函数。处理冲突时，采用+1的线性探测法。

HashTable类的最后两个方法提供了额外的字典功能：getitem与setitem。创建HashTable类之后，可以使用熟悉的索引运算符

程序实现：

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size

    def put(self,key,data):
        hashvalue = self.hashfunction(key,len(self.slots))

        if self.slots[hashvalue] == None:
            self.slots[hashvalue] = key
            self.data[hashvalue] = data
        else:
            if self.slots[hashvalue] == key:
                self.data[hashvalue] = data #replace
            else:
                nextslot = self.rehash(hashvalue,len(self.slots))
                while self.slots[nextslot] != None and self.slots[nextslot] != key:
                    nextslot = self.rehash(nextslot,len(self.slots))

                if self.slots[nextslot] == None:
                    self.slots[nextslot] = key
                    self.data[nextslot] = data
                else:
                    self.data[nextslot] = data #replace

    def hashfunction(self,key,size):
        return key%size

    def rehash(self,oldhash,size):
        return (oldhash+1)%size

    def get(self,key):
        startslot = self.hashfunction(key,len(self.slots))

        data = None
        stop = False
        found = False
        position = startslot
        while self.slots[position] != None and not found and not stop:
                if self.slots[position] == key:
                    found = True
                    data = self.data[position]
                else:
                    position=self.rehash(position,len(self.slots))
                    if position == startslot:
                        stop = True
```

```
        return data

    def __getitem__(self,key):
        return self.get(key)

    def __setitem__(self,key,data):
        self.put(key,data)
```

注：python中字典的实现原理就是散列表

（二）神奇的dict

dict的value如果是一层list，则是邻接表（最传统的邻接表是列表套列表，但是在查找上时间复杂度不高。因此也可以将两层列表改为dict）

1.邻接表：

在图论中，邻接表是一种表示图的常见方式之一。如果使用字典（ `dict` ）来表示图的邻接关系，并且将每个顶点的邻居顶点存储为列表（ `list` ），那么就构成了邻接表。例如：

```
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D'],
    'D': ['B', 'C']
}
```

2.字典树（前缀树，Trie）

字典树是一种用字典嵌套表示树形数据结构的方式，用于高效存储和检索字符串数据集中的键。

```
trie = {
    'a': {
        'p': {
            'p': {
                'l': {
                    'e': {'is_end': True}
                }
            }
        }
    },
    'b': {
        'a': {
            'l': {
                'l': {'is_end': True}
            }
        }
    },
    'c': {
        'a': {
```

```
            't': {'is_end': True}
        }
    }
}
```

其中，根结点不包含字符，除根结点外每一个结点都只包含一个字符；从根结点到某一结点，路径上经过的字符连接起来，为该结点对应的字符串；每个结点的所有子结点包含的字符都不相同。