7，字典和集合-I

- 数据检索和字典，数据规模和检索，索引和字典
- 基于表和排序表实现字典和检索，二分检索
- 散列的概念，散列表和字典，散列函数和冲突消解，实现
- 集合数据结构，集合的实现，位向量实现
- Python 的字典和集合
- 二叉排序树的概念和实现
- 最佳二叉排序树，等概率和不等概率情况，构造算法
- 支持动态操作的排序树：AVL 树
- 支持字典实现的其他树形结构简介

数据存储、检索和字典

- 本章不是介绍一种结构，而是介绍计算中最重要的一类问题的许多不同解决方式，以及与之相关的问题和性质
- 存储和检索是计算中最重要最基本的工作，也是各种计算机应用和信息处理的基础。数据需要存储和使用，因此需要检索
- 本章研究的问题就是数据的存储和检索（查询），实际中的例子：
  - 电子字典，基本功能就是基于算法的数据检索
  - 图书馆编目目录和检索系统，支持读者检索书籍资料的有关信息
  - 规模巨大的有机物库，需要基于结构或光谱等参数进行检索
  - 多元多项式乘法，做出一个因子乘积后应合并同类项，需要检索
- 本章讨论的是基于关键码的数据存储和检索
  - 关键码指数据项的某种（可能具有唯一性的）特征，可以是数据内容的组成部分，也可以是专门为数据检索建立的标签
  - 支持这种操作的数据结构，通常称为字典、查找表或映射

字典

- 字典就是实现存储和检索的结构。需要存储和检索的信息和环境有许多具体情况，因此要考虑各种不同的字典实现技术
- 字典的实现可以用到前面讨论过的许多想法和结构。包括
  - 各种线性结构、树性结构及其各种组合
  - 涉及到在这些结构上操作的许多算法
  - 组织方法很多，下面讨论顺序、散列、二叉树和其他树形结构等
  - 这里的基本问题是空间利用率和操作效率
- 字典的最主要也是使用最频繁的操作是检索（**search**，也称查找）
  - 检索效率是字典实现中最重要的考虑因素
  - 由于规模不同，检索效率的重要性也可能不同
- 更一般的问题是需要根据某些线索找出相关数据，可能需要做更复杂的匹配，或者“模糊”的匹配，基于内容的匹配。例如网络上的检索。这些都是字典概念的发展

字典

- 字典可以分为两类：
  - 静态字典：建立后保持不变，只做检索，实现只需考虑检索效率
  - 动态字典：内容经常动态变动的字典。除检索外，基本操作还包括插入和删除等，实现时就必须考虑插入删除操作的效率
- 动态字典的插入删除操作可能导致字典结构的变化。要支持长期使用，还需要考虑字典在动态变化中能否保持良好的结构，能否保证良好的检索效率？（字典的性能不应该随着反复操作而逐渐恶化）

检索效率的评价标准是检索过程中关键码的平均比较次数，称为平均检索长度 **ASL**（**Average Search Length**），定义（**n** 为字典的项数）：

$$ASL = \sum_{i=0}^{n-1} p_i c_i$$

c_i 和 p_i 分别为数据元素 **i** 的检索长度和概率。如果各元素检索概率相等，就有 $p_i=1/n$，$ASL = 1/n \sum c_i$。还可能需要考虑找不到的情况

字典：抽象模型

设有关键码集合KEY和值（或称属性）集合$VALUE$

关联 **(Association)** 是二元组 $(k, v) \in KEY \times VALUE$

字典：以关联 为元素的有穷集合

$$dic \subseteq KEY \times VALUE$$

$$i \neq j \Rightarrow k_i \neq k_j$$

关键码到值的有穷函数： $dic : KEY \rightarrow VALUE$

所有字典的集合： $DIC \subseteq \mathcal{P}(KEY \times VALUE)$

主要字典操作：

- 检索 $search : DIC \times KEY \rightarrow VALUE$
- 插入元素 $insert : DIC \times KEY \times VALUE \rightarrow DIC$
- 删除元素 $delete : DIC \times KEY \rightarrow DIC$

字典和索引

- 字典是两种功能的统一：
  - 既作为数据的存储结构，支持数据的存储
  - 也提供支持数据检索的功能，维护着关键码到所存数据的联系
- 索引是字典的附属结构，它只提供检索功能
  - 索引可能提供与基本字典不同的查找方式，例如另一套关键码
  - 基于关键码的索引，实现的是关键码到数据存储位置的映射
  - 一个字典可以没有另外的索引，只有自身提供的检索方式；也可以附有一个或多个索引，支持多种不同方式的检索

  考虑现实中的《新华字典》，它有基本部分和若干种索引（部首等）
- 下面讨论的各种技术，既可用于实现字典也可用于实现索引
  - 实现字典时，关键码关联于实际数据，数据保存在字典里
  - 实现索引时，关键码关联于数据在相关字典里保存的位置

字典元素：关联

- 字典的元素是关联，定义一个类：

```
class Assoc:
  def __init__(self, key, value):
      self.key = key
      self.value = value
   def __lt__(self, other): # 有时（有些操作）可能需要考虑序
      return self.key < other.key
   def __le__(self, other):
      return self.key < other.key or self.key == other.key
   def __str__(self): # 定义字符串表示形式便于输出和交互
      return "Assoc({0},{1})".format(self.key, self.value)
```

- 假定下面讨论的字典都以 **Assoc** 对象为元素

  如果 **x** 的值是关联，**x.key** 取得其关键码，**x.value** 取得其关联值

  定义 **<** 和 **<=** 是因为有时可能需要比较数据项，如使用 **sorted**

字典的实现

- 从最基本的存储需求看，字典也就是关联的汇集
  - 前面讨论过的各种数据汇集结构都可用作字典的实现基础
  - 例如线性表，是元素的线性的顺序汇集。如果以关联作为元素，就可以看作是字典了。下面首先考虑这种实现
  - 作为字典实现，最重要的问题是字典操作的实现。由于字典可能有一定规模，需要频繁执行查询等操作，操作的效率非常重要
- 下面将讨论一系列字典实现技术
  - 首先是线性表，特别是连续表
  - 而后是另一种特别的技术：散列表
  - 最后是基于树形结构的各种技术

线性表表示

- 线性表里可以存储信息，因此可以作为字典的实现基础
  - 关联在线性表里顺序排列，形成关联的序列
  - 检索就是在线性表里查找具有特定关键码的数据项，数据项的插入删除都是普通的线性表操作
- 在 **Python**，顺序字典可用 **list** 实现
  - 关联可以用 **Assoc** 对象，也可以用二元的 **tuple** 或 **list** 实现
  - 检索就是在用关键码在表中查找（顺序查找）。遇到关键码 **key** 相同的字典项就是检索成功，返回相应的 **value**；检查完表中所有的项但没遇到要找的关键码，就是检索失败
  - 插入新关联用 **append** 实现；删除可以在定位后用 **del** 实现，或者基于要删除项的内容，用 **remove** 操作实现
- 还可以考虑以 **list** 作为内部表示，自己定义一个字典类，该类的对象就是具体字典，字典操作实现为类里的对象方法。这些都留着自我练习

简单线性表字典的性质

- 插入的元素放在最后，**O(1)** 时间复杂性
- 删除元素时需要先检索，确定了元素的位置后删除（表中删除元素）
- 主要操作是检索（删除依赖于检索），分析其复杂性，考虑比较次数：

$$\begin{aligned} ASL &= 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n \\ &= (1 + 2 + \ldots + n)/n \qquad \text{when } p_i = 1/n \\ &= (n+1)/2 = O(n) \end{aligned}$$

- 基于线性表的字典实现，优点和缺点都很明显：
  - 优点：数据结构和算法简单，适用于任何关键码集合
  - 缺点：平均检索长度大，表长度 **n** 较大时检索耗时太长
  - 删除的效率也比较低，因此不太适合频繁变动的字典
  - 在字典的动态变化中，各种操作的效率不变（因为都已经是效率很低的操作了）

有序线性表和二分检索

- 要想提高字典的操作效率，就需要把字典里的数据项更好地组织起来，使之具有可利用的结构，从而提高检索的效率

  内部结构更复杂的字典，可能支持更有效的检索

- 如果关键码取自一个有序集（关键码集合有某种序，例如整数的小于，字符串的字典序），就可以将字典了的项按关键码排列（从小到大或从大到小）。由于数据项排列有序，可以采用二分法实现快速检索
- 二分法检索通过按比例缩小检索范围的方式，快速逼近要检索的数据。其基本过程是（假设元素按关键码的升序排列）：
  1. 初始时，所考虑的范围是整个字典（一个顺序表）
  2. 取所考虑范围里位于中间位置的数据项，比较该项的关键码与检索关键码。如果它们相等则检索结束；否则
  3. 如果检索关键码较大，把检索范围改为中间项之后的一半；如果检索关键码较小，把检索范围改为中间项之前的一半
  4. 如果范围里已经没有数据，检索失败结束，否则回到 **2** 继续

有序线性表和二分检索

- 二分检索的实现：

```
def bisearch(lst, key):
    low, high = 0, len(lst)-1
    while low <= high: # 范围内还有元素
        mid = (low + high)//2
        if key == lst[mid].key:
            return lst[mid].value
        if key < lst[mid].key:
            high = mid - 1 # 在低半区继续
        else:
            low = mid + 1  # 在高半区继续
```

- 向排序表里插入数据时需要保序，因此是 **O(n)** 操作；删除时可以用二分法检索，但实际删除后需要移动数据项，所以也是 **O(n)** 操作
- 插入的一个特殊情况是被查关键码存在，可修改关联值或插入新项（即允许关键码重复），删除时有删除一项还是所有关键码相同项的选择

有序线性表和二分检索

- 二分法检索的具体示例：

  以下面 **11** 个数的检索为例：

  关键码： **5 13 19 21 37 56 64 75 80 88 92**

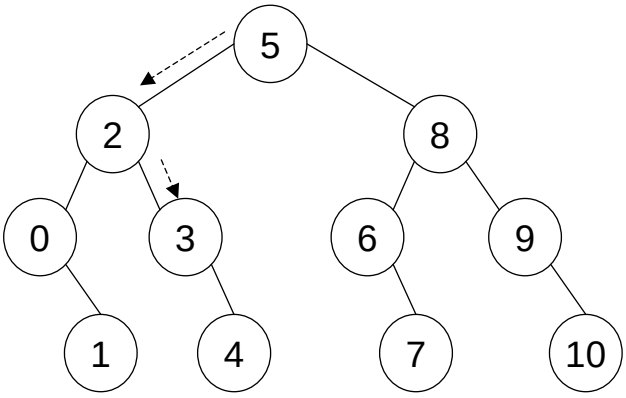
  位置： **0 1 2 3 4 5 6 7 8 9 10**

  采用二分法检索：找到位置 **5** 的数需比较 **1** 次，找到位置 **2, 8** 需比较 **2** 次，找到位置 **0, 3, 6, 9** 需 **3** 次，找到另外 **4** 个位置需比较 **4** 次

检索过程可用二叉树表示。树结点所标数字是数据的位置

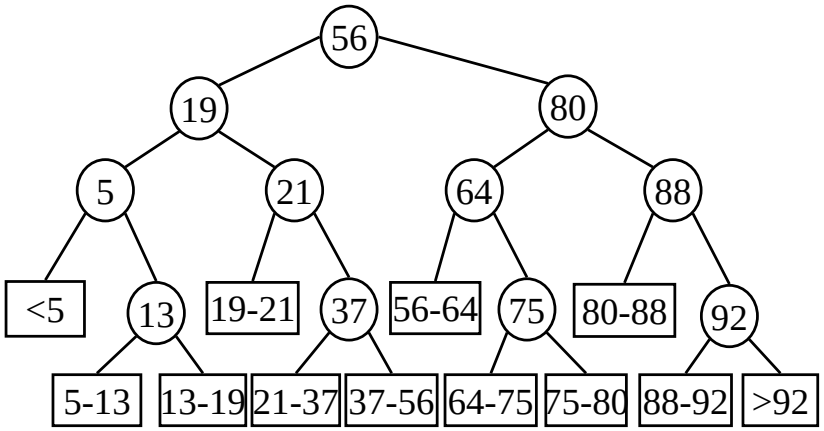
通过检索找到某结点的比较次数等于该结点的层数加 **1**

检索结点的过程沿着从根结点到需要检索的结点的路径，在每个位置做了一次比较



有序线性表和二分检索

- 检索成功时所做比较次数不超过树的深度。**n** 个结点二分判定树的高度不超过 $\lfloor \log_2 n \rfloor$。二分法检索成功时的比较次数不超过 $\lfloor \log_2 n \rfloor + 1$
- 包含检索成功和不成功情况的判定树如下。方框表示检索不成功（是扩充二叉树的外部结点），例如下图中标着 **13-19** 的方框表示被检索关键码值在 **13** 和 **19** 之间。检索到达方框就是失败



由此可见，检索不成功时，最大比较次数也是 $\lfloor \log_2 n \rfloor + 1$

这是基于检索中的判定操作形成的判定树的分析

有序线性表和二分检索

- 每次比较范围缩小一半，第 i 次需比较的元素个数如下表

比较次数	可能比较的元素个数
1	$1=2^0$
2	$2=2^1$
⋮	⋮
k	2^{k-1}

如果元素个数 n 恰好为

$2^0 + 2^1 + \ldots + 2^{k-1} = 2^k - 1$

则最大检索长度为 k；对于一般的 $2^k-1 < n \leqslant 2^{k+1}-1$，最大检索长度为 k+1

- 所以，n 元字典的二分法检索，最大检索长度为 $\lceil \log_2(n+1) \rceil$

平均检索长度（设 $n = 2^j - 1$，j 是搜索的“深度”）：

$$ASL = \frac{1}{n} * (\sum_{i=1}^{j} i * 2^{i-1})$$

$$= \frac{1}{n} * \sum_{i=1}^{j} \sum_{m=i}^{j} 2^{m-1}$$

$$= \frac{1}{n} * \sum_{i=1}^{j} (2^j - 2^{i-1})$$

$$= \frac{1}{n} * (j * 2^j - \sum_{i=1}^{j} 2^{i-1})$$

$$= \frac{1}{n} * (j * 2^j - 2^j + 1)$$

$$= \frac{n+1}{n} * \log_2(n+1) - 1$$

有序线性表实现的字典

- 有序线性表和二分法检索（排序顺序字典）：
  - 优点：检索速度快，O(log n)
  - 插入删除时需要维护数据项顺序，O(n) 操作（检索插入或删除的位置可以用二分法，O(log n)，但完成操作仍需要 O(n) 时间）
  - 只能用于数据项按关键码排序的字典，只适合顺序存储结构；不适合用于实现大的动态字典
- 也可以考虑用单链表或双链表实现字典，有关情况：
  - 如果字典里的数据项任意排列：插入时可以简单插入在表头，O(1) 操作；检索和删除需要顺序扫描检查，O(n) 操作
  - 如果数据项按关键码升序或者降序排列，插入需要检索位置，O(n) 操作；检索和删除需要顺序扫描检查，平均查半个表，O(n) 操作
  - 易见：用链接表实现字典没有任何优势，实际中很少采用。具体的实现技术很简单，不需要再进一步讨论

字典的操作效率

- 采用线性表技术实现字典，常常不能满足实际的需要
- 实际中需要存储和检索的数据集的数据量常常很大，内容动态变化
  - 采用简单连续表或链接表，顺序检索的效率太低，不能满足存储和检索大规模的数据集合的需要
  - 采用排序连续表和二分检索，检索速度大大提高，但仍有两大问题
    1. 不能很好支持数据的变化（数据插入和删除）
    2. 必须采用连续方式表示。如果数据集很大（例如几百 **M** 或者几个 **G** 或更大的数据集），连续实现方式就很难接受了

  要支持在大且变动的数据集合里高效检索，必须考虑其他组织结构
- 实际中人们考虑了另一些结构，主要分为两类：
  - 基于散列（**hash**）的思想开发的散列表（哈希表）
  - 基于树形结构的数据存储和检索技术（利用树结构的特性，即，在不大的深度范围内可以容纳巨大数量的结点）

散列表（Hash Table，哈希表，杂凑表）

- 什么情况下检索元素的速度最快？

  如果关键码是连续表的下标，就可以直接找到元素！（常量时间）
- 但是，一般而言，关键码可能不是整数（不能作为下标）

  即使是整数，也可能取值范围太大，不好直接作为下标

  例如，北大学生学号有 **10** 位数字，取值范围达到 **100** 亿
- 散列表的基本思想就是把基于关键码的检索变为基于整数下标的访问
  - 方法：选定一个整数下标范围（通常以 **0** 或 **1** 开始）建立一个连续表；选定一个从关键码集合到该下标范围的适当的映射 ***h***
  - 存入关键码为 ***key*** 的数据时，将其存入表中第 ***h*(*key*)** 个位置
  - 以 ***key*** 为关键码检索数据时，直接去查第 ***h*(*key*)** 个位置的元素
- 这个 ***h*** 就称为散列函数，又称哈希（**hash**）函数或杂凑函数，是从可能的关键码到一个整数区间里（下标）的映射。其类型是：

$$h : KEY \rightarrow INDEX$$

散列技术（Hashing）

- 散列的思想是计算领域逐渐发展起来的一种极其重要的思想。所谓散列，就是以某种精心设计的方式，从可能很长的一组数据生成出一段很短（常为固定长度）的信息串（整数/字符串，都是二进制序列）
- 散列技术在计算机和信息领域很有价值，应用极广，可能用在各种数据处理、存储、检索工作中。例如：
  - 文件完整性检查（定义一个散列函数从软件的所有文件映射到一个数或一个字符串，需要时检查实际文件的散列值是否与其匹配）

    软件安装时说的“检查文件的完整性”，就是基于这种技术检查文件。这显然是一种概率性的检查，但出错的概率极低
  - 互联网技术中到处都使用和依靠散列函数
  - 计算机安全领域中大量使用散列技术，例如各种安全协议
- 将散列技术用于存储和检索，就得到了散列表。基本做法如前所述

  实现散列表，需要选择合适的关键码映射（散列函数），还要考虑由于用这种映射确定存储和检索位置而带来的问题

散列技术：设计和性质

- 对于散列表，通常都有 $|KEY| >> |INDEX|$

  h 把一个大集合的元素映射一个小集合里，显然不可能是单射，必然会出现多个关键码对应到同一个位置的情况，即出现

  $$key_1 \neq key_2 \quad 但 \quad h(key_1) = h(key_2)$$

  此时就说出现冲突（碰撞），也称 key_1 和 key_2 为同义词

  冲突是必然会出现的情况，散列表的实现必须考虑和解决冲突问题
- 显然，对规模固定的散列表，一般而言，表中元素越多，出现冲突的可能性也就越大，人们提出“负载因子”作为一种有用的度量
- 负载因子 α 是考察散列表性质的一个重要参数。定义是：

  $$\alpha = \frac{\text{散列表中实际元素个数}}{\text{基本存储区能容纳的元素个数}}$$
- 这样定义的负载因子总小于等于 1，其大小与冲突出现的可能性密切相关：负载因子越大，出现冲突的可能性也越大

散列表

- 增大存储空间（增加可能存储位置）可以减小负载因子，减小出现冲突的概率。但负载因子小，空闲空间的比例就大。这里也有得失权衡

  人们在散列表的设计中提出了许多冲突处理技术。不同的处理方式形成了不同的多种散列表实现结构

- 下面讨论散列表设计实现的两大问题：散列函数设计，冲突消解机制

  首先考虑散列函数的设计问题

- 关键码和位置区间是事先确定的，作为散列函数的基础集合（参数域和值域）。散列函数就是两个有限集之间的函数，可以任意选择，但其选择可能影响出现冲突的概率（很显然）。最重要的考虑：
  - 尽可能把关键码映射到不同值，映射到值域中尽可能大的部分
  - 关键码的散列值在下标区间里均匀分布，有可能减少冲突的出现。当然，实际情况还与真实数据里不同关键码值出现的分布有关
  - 计算比较简单（使用散列表的本意就是要提高效率）

散列函数

- 散列函数的基本想法就是其映射关系越乱越好，越没有规律越好。在此基本考虑下，人们提出了许多设计散列函数的方法
- 一些方法依赖于对实际数据集的分析，实际中很难使用

  如果事先知道需要存储的数据及其分布，有可能设计出一个散列函数取得最佳存储效果，甚至可能保证不出现冲突

  一些数据结构教科书里介绍了一批散列函数设计，例如张老师和清华的数据结构教材。如数字分析法，折叠法，中平方法等

- 常见情况是需要存储和使用的数据不能在设计字典之前确定，具体使用的关键码和分布情况事先未知，上述分析方法就没用了

  只能用一些通用方法，基于对数据集（关键码）的均匀分布假设

- 下面只介绍两种散列函数
  - 除余法，用于整数关键码的散列
  - 基数转换法，用于整数或字符串关键码的散列

散列函数

- 除余法：关键码是整数。以关键码除以一个不大于散列表长度 m 的整数 p 得到的余数（或者余数加 1）作为散列地址

$$h(key) \hat{=} key \bmod p$$

  m 经常取 2 的某个幂值，此时 p 可以取小于 m 的最大素数（如果连续表的下标从 1 开始，可以用 $key \bmod p + 1$）

  例如，当 m 取 128，256，512，1024 时，p 可以分别取 127，251，503，1023

  除余法使用最广，还常用于将其他散列函数的结果归入所需区间

- 设计散列函数的最基本想法是使得到的结果尽可能没有规律
  - 采用除余法法时，如果用偶数作为除数，就会出现偶数关键码得到偶数散列值，奇数关键码得到奇数散列值的情况
  - 如果关键码集合的数字位数较多，可考虑采用较大的除数，然后去掉最低位（或去掉最低的一个或几个二进制位），排除最低位的规律性。还可以考虑其他方法，目标是去掉规律性

散列函数

- 基数转换法：针对整数关键码。取一个正整数 r，把关键码看作基数为 r 的数，将其转换为十进制或二进制数。通常 r 取一个素数

  例，取 r = 13。对关键码 335647，可以计算出：

$$(335647)_{13} = 3 * 13^5 + 3 * 13^4 + 5 * 13^3 + 6 * 13^2 + 4 * 13 + 7$$

$$= (6758172)_{10}$$

  可以考虑用除余法或删除几位数字的方法将其归入所需范围

- 对非整数关键码，常用的是先设计一种方法把它转换到整数，而后再用整数散列的方法。通常最后用除余法把关键码归入所需范围
- 字符串也常作为关键码。常见方法是把一个字符看作一个整数（例如用字符的编码），把一个字符串看作以某个整数为基数的“数”
  - 常以 29 或 31 为基数，用基数转换法把字符串转换为整数
  - 再用整数散列法（如除余法），把结果归入散列表的下标范围

冲突的消解

- 冲突是散列表必然会出现的事件
  - 大集合到小集合的全函数，必然会出现两个元素函数值相同的情况
  - 设计散列表时必须确定一种冲突消解方案
- 人们提出了一些冲突消解方法，可分为
  - 内消解方法（在基本存储区内解决元素冲突）
  - 外消解方法（在基本存储区之外解决元素冲突）
- 用散列函数根据关键码为要插入的数据项算出存储位置，但却发现那里已经有关键码不同的项，就知道出现了冲突，必须处理
- 对冲突处理技术的基本要求：
  - 保证存入当前数据项的工作能够完成
  - 保证字典的基本存储性质：在任何时候，从任何以前存入字典而后未删除的关键码出发，都能找到对应的数据项

散列表：开地址法和探查序列

- 开地址法（内消解法）：
  - 出现冲突时设法在连续表里为要插入的数据项另安排一个位置
  - 需要设计一种系统的且易于计算的位置安排方式（称为探查方式）
- 常用方法是为散列表定义一个探查位置序列：

  $H_i = (h(key) + d_i) \bmod m$

  其中 m 为不超过表长的数，d_i 为一个增量序列，设 $d_0 = 0$

  插入数据项时，如果 $h(key)$ 空闲就直接存入（相当于使用 d_0）

  否则就逐个试探位置 H_i，直至找到第一个空位时把数据项存入
- 增量序列有许多可能取法，例如

  取 $d_i = 0, 1, 2, 3, 4, \ldots$，称为线性探查

  设计另一散列函数 h_2，令 $d_i = i * h_2(key)$，称为双散列探查

散列表：开地址法示例

- 例：设 key 为整数，取 h(key) = key mod 13。设有关键码集合

  KEY = {18, 73, 10, 5, 68, 99, 22, 32, 46, 58, 25}

  h(key) = { 5, 8, 10, 5, 3, 8, 9, 6, 7, 6, 12}，用线性探查

位置	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码	58	25		68		18	5	32	73	99	10	22	46

  使用线性探查法容易出现堆积现象，即在处理同义词时又引进非同义词之间冲突。其他探查法也可能出现这种情况，但可能稍好

- 同样实例，采用双散列探查法

  h(key) = key mod 13；h_2(key) = key mod 5 + 1

位置	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码	99	58	25	68		18	5	46	73	22	10		32

散列表：开地址技术下的检索与删除

- 在开地址法散列表上做检索，工作过程与插入类似。对给定的 key：
  - 用散列函数求出对应的散列地址
  - 如果该位置无数据项，就说明散列表里无此数据，检索失败结束
  - 否则比较数据项的关键码，如果与 key 相等则检索成功结束
  - 否则根据散列表的探查序列找到“下一地址”并回到步骤 2

  为判定“找不到”，还需要为“无值”确定一种表示方式

- 开地址法的一个问题是元素删除。在需要删除一个数据项时，如果在找到对应项后简单地将其删掉，有可能影响后面的检索操作
  - 如果被删除元素位于同一散列值或不同散列值的检索链上，直接删除就会导致检索链的破坏
  - 解决办法：在被删除元素处放一个（与空位标记不同的）特殊标记。检索操作将这个标记看作有元素，还将继续进行探查；而插入操作工作中则把这种标记看作空位

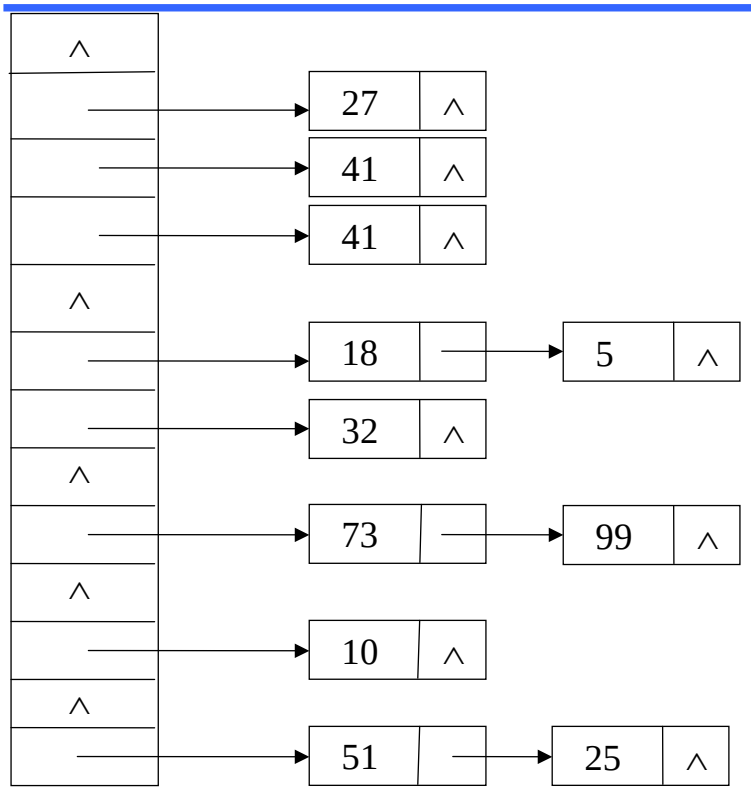
散列表：溢出区

- 外消解冲突的一种方法是在基本存储区之外，另外设置一个公共溢出区，把所有关键码冲突的数据项都保存在这个溢出表里
- 例如，用一个线性表作为公共的溢出区
  - 如果插入数据时遇到冲突，就将数据项放入公共溢出区里下一空位
  - 检索时先通过散列函数找到对应位置
    - 如果关键码匹配就返回相应的数据
    - 如果关键码不匹配，就转到公共溢出区去顺序检索。注意，这种检索是在线性表里检索，时间复杂性是线性的
- 很容易看出
  - 如果散列表里的数据项很多，溢出区就可能变得很大
  - 随着数据项的增加，插入和检索的工作量都会趋于线性
- 采用了公共溢出区的散列表，散列表的负载因子有可能超过 **1**

散列表：拉链法

- 如果数据项不存放在散列表里，而是另外存储。在散列表里保存对数据项的引用，可以发现很多可能的设计。最简单的技术是“拉链法”
- 拉链法的基本思想：数据项存在散列表的基本连续表之外，每个关键码建立一个链接表，所有关键字为同义词的数据项存在同一个链表里
- 这样，所有数据项都可以统一处理（无论是否为冲突项）
  - 允许任意的负载因子
  - 插入操作的最简单实现是把新元素插在链接表头，如果要防止出现重复关键码，就需要检查整个链表
- 例： **h(*key*) = *key* mod 13**
  - 关键码集合：**{18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25}**
  - 算出的位置：**{ 5, 8, 10, 5, 3, 8, 9, 6, 7, 6, 12}**
  - 把同一个散列值的数据项收集到一起：**{ { }, {{27}, {41}, {68}, { }, {05, 18}, {32}, { }, {99, 73}, { }, {10}, { }, {25, 51} }**

散列表：拉链法



K = {18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25}

h(*key*) = *key* mod 13

拉链法散列表

散列表的实现

- 在实际应用中，拉链法还可以推广
  - “同义词表”也可以采用顺序表或散列表，还可以采用其他结构
  - 人们有时把存放同义词的表称为“桶”，称这种结构为“桶散列”
  - 这种结构可以用于存储大型字典，或者用于组织大量文件
- 无论采用哪种消解方法

  随着元素增加，负载因子增大，出现冲突的可能性会明显增大

  在开地址法中，最终导致存储区溢出；在溢出区方法里是溢出区越来越大，检索效率趋于线性；在拉链法中表现为链的平均长度增加
- 可以考虑在负载因子达到一定程度时下扩大基本存储表
  - 如何扩大存储的策略和性质，在前面介绍连续表时已经讨论过了
  - 扩大存储区后，把字典里已有数据项重新散列到新的存储区里
  - 这时要付出重新分配存储区的代价和再散列装入数据项的代价

散列表：扩大存储

- 这是明显的时间与空间交换（计算机科学技术中的一条基本原理）
  - 采用内部消解时的一般情况

    经验说明负载因子 $\alpha \leqslant 0.7 \sim 0.75$ 时平均检索长度接近常数
  - 采用桶散列，负载因子就是桶的平均大小（平均长度）

    因此可以容忍任意大的负载因子

    随着负载因子变大，检索时间趋于线性（桶长 ＝ 数据项数/桶数）
- 散列表字典的许多性质都是概率性的，不是确定性

  基本假设是
  - 实际存入字典的数据（关键码）的散列函数值分布均匀
  - 字典散列表的负载因子不太高（试验证明应在 0.7 以下）

  在上述假设下，字典检索、插入、删除操作的时间开销近乎常量

散列表字典的性质

- 散列表字典在概率上极为高效，但也有另一面，包括一些必然情况：
  - 常量时间操作代价是平均，不是每次操作的实际情况
  - 冲突必然会发生，不同操作的代价可能差异很大，且不能事先确知
  - 不断插入导致负载因子增大，为保证操作效率就需要扩大存储，造成高代价的插入。无法预计这种高代价操作什么时候出现
  - 基于内消解机制的字典，长期使用性能会变差（很多长的探查序列）
  - 字典内部项的排列顺序无法预知，也没有任何保证
- 还有一些可能出现的情况：
  - 实际数据的散列值相对集中，导致一些操作的性能恶化

    极端情况也可能出现很长的探查序列，使一些操作非常低效
  - 长期使用的大散列表里（采用内消解技术），有可能出现很长的已删除元素序列，影响很多操作的效率

散列表字典的性质

- 一些可以考虑的技术：
  - 给用户提供检查负载因子和主动扩大散列表存储区的操作

    用户可在一段效率要求高的计算前根据需要设定足够大的存储
  - 记录或检查被删除项的量或比例，在一定情况下自动整理

    最简单的方法就是把有效数据项重新散列到另一块存储区

    重新散列时可以消去开地址散列表里所有已删除项的空位
- 散列表字典使用广泛，人们已经积累了许多设计和实现的经验
  - 许多编程语言或标准库提供了基于散列表的字典。常被称为 **map**，或者 **table**，或者 **dictionary**
  - 很多软件以散列表作为基本实现技术，例如 **Maple** 等
  - **Python** 的 **dict** 就是基于散列表实现的，其性质下面就能看清楚

    Python 系统里的很多地方也使用了散列表，后面简单介绍