数据结构与算法B

10-字典与检索

目录

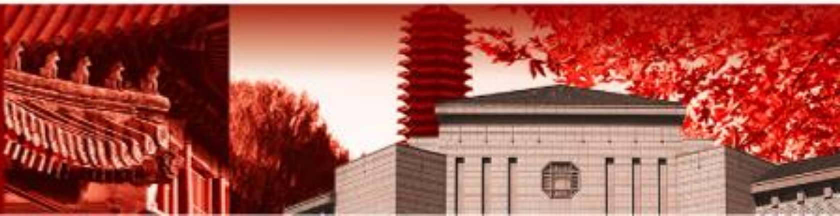
10.1 检索问题的基本概念

检索问题

- 检索：在一个数据结构中查找关键码值等于给定值的元素。
  - 数据结构中的元素可能包含不止一个属性，检索过程只需要针对其中的个别属性，称为检索的关键码
  - 检索也称为查找，在这一章二者是同义的
- 检索的结果：
  - 如果找到，则检索成功；
  - 否则应该报告检索失败，即数据结构中不存在符合要求的元素

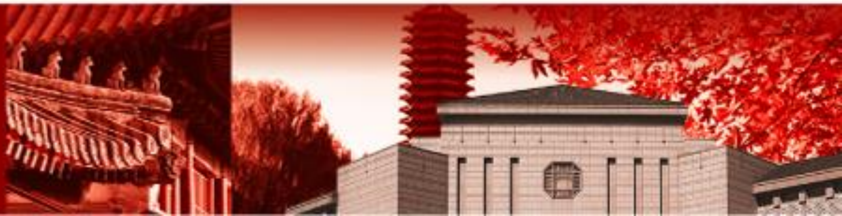
检索问题

- 不同种类的检索
  - **精确匹配查询**（exact-matching query）：在数据结构中查找关键码值与查询值相等的所有元素。
  - **范围查询**（range query）：在数据结构中查找关键码值属于某个指定范围内的所有元素。
- 在本章中，假定关键码都为正整数，各数据记录（即数据结构中存储的元素）类型相同，因此各元素可以按照关键码排序。



检索与字典

- 字典(Dictionary)是元素的有穷集合，其主要的操作为对元素的检索。
- 字典中的每个元素由两部分组成，分别称为元素的关键码(key)和属性 (attribute)。
  - 关键码本质上是一个特殊的属性
  - 必须保证字典中的每个元素具有唯一的关键码
    - 比如：学号就是每个同学的关键码，身份证号就是每个公民的关键码。
    - 可以通过关键码来查询元素的其他属性
    - 如果有两个元素关键码相同，我们就无法区分这两个元素

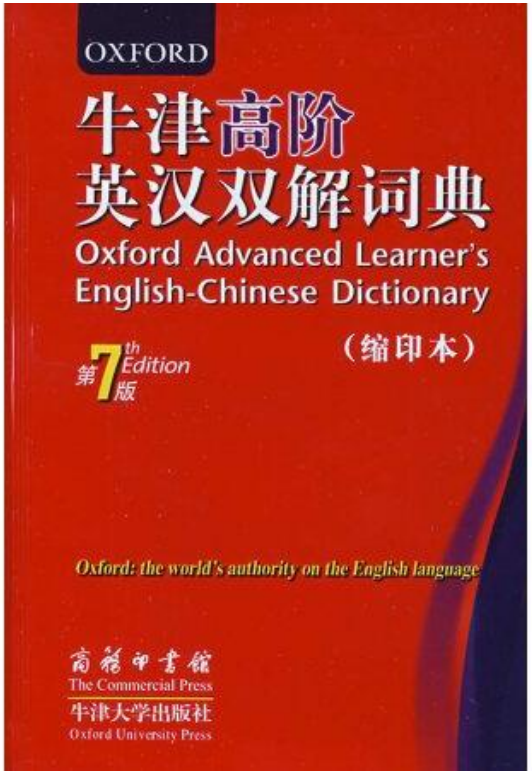


检索与字典

- 静态(static)字典：字典一经建立就基本固定不变，主要的操作就是字典元素的检索。
  - 为静态字典选择存储方法主要考虑检索效率、检索运算的简单性
  - 在实际应用中，有时要考虑字典的插入和删除操作，所以静态字典无法满足要求。
- 动态(dynamic)字典：经常需要改动的字典。
  - 对于动态字典，存储方法的选择不仅要考虑检索效率，还要考虑字典元素的插入、删除运算是否简便。

例：生活中的字典

- 字典中的每个元素由两部分组成，分别称为元素的关键码(key)和属性(attribute)。
  - 英汉字典中，每个词条是一个元素
  - 词条中的英文单词可看作是该元素的关键码
  - 对该英文单词的解释可看作是元素的属性。
- 关键在于，如何提高检索的效率
  - 思考：如何在字典中快速检索指定单词?

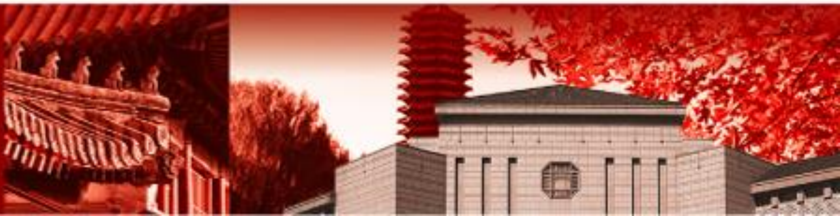


检索算法的效率

- 如何评价一个检索算法的效率？
  - 检索过程中关注的基本操作是关键码的比较
  - 衡量检索算法效率的主要标准是：检索过程中关键码的平均比较次数，即平均检索长度 (Average Search Length, ASL)，定义为：

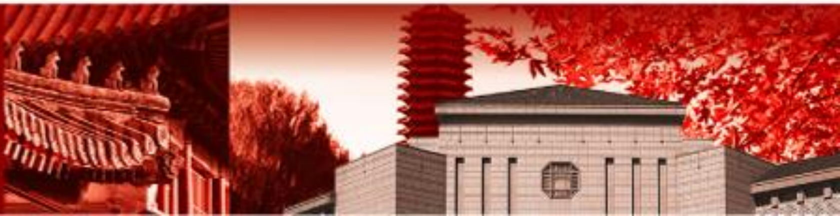
$$ASL = \sum_{i=1}^{n} p_i c_i$$

  - n是元素的个数；p_i是查找第i个元素的概率；c_i是算法为了找到第i个元素所需的比较次数。
  - 除非特别声明，一般假定各元素的检索概率相等，即p_i=1/n

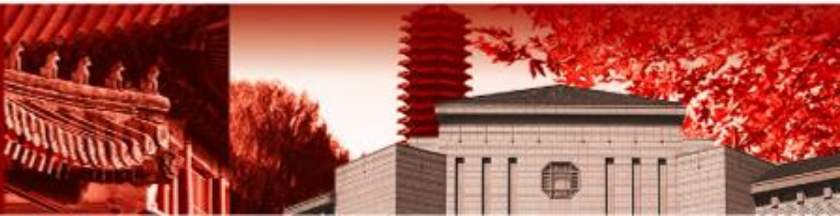


检索算法

- 检索算法的分类
  - 基于线性表的方法：顺序检索、二分检索
  - 基于散列表的方法：散列检索
  - 树索引法：在原有数据结构之外，将所有关键码额外组织为树结构，称为索引，用来加快增删改查操作
    - 通常用于动态增删的场景
    - 包括二叉搜索树、B树、B+树、红黑树、字典树等



10.1 顺序检索



顺序检索

- 基本思想：
  – 顺序检索是基于线性表的检索方法
  – 从线性表的一端开始顺序扫描，将元素的关键码和给定值比较，如果相等，则检索成功；
  – 当扫描结束时，还未找到关键码等于给定值的元素，则检索失败。

顺序检索

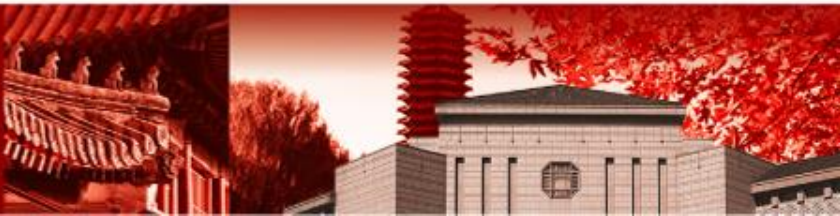
- 平均检索长度ASL：
  - 若找到的是第i个元素，则比较次数为$c_i=i$。因此

$$ASL = 1 \times P_1 + 2 \times P_2 + \ldots + n \times P_n$$

  - 假设每个元素的检索概率相等，即$P_i=1/n$，则平均检索长度为：

$$ASL = \sum_{i=1}^{n} p_i c_i = \sum_{i=1}^{n} i/n = (n+1)/2$$

  - 因此，成功检索的平均比较次数约为字典长度的一半；若字典中不存在关键码为key的元素，则需进行n次比较。
  - 总之，顺序检索的平均检索长度为ASL=O(n)

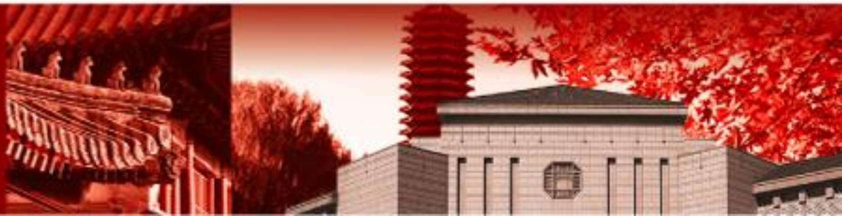


顺序检索的实现

```python
def sequentialSearch(alist, item):
    pos = 0
    found = False

    while pos < len(alist) and not found:
        if alist[pos] == item:
            found = True
        else:
            pos = pos+1

    return found

testlist = [1, 2, 32, 8, 17, 19, 42, 13, 0]
print(sequentialSearch(testlist, 3))
print(sequentialSearch(testlist, 13))
```

顺序检索

- 顺序检索优点：
  - 算法实现简单，是最基础的检索方法
  - 顺序检索不要求字典中的元素的有序性，适用场景更广
- 顺序检索缺点：
  - 平均检索长度较大，复杂度为O(n)
  - 特别是当n很大时，检索效率较低。



顺序检索

- 顺序检索的改进：
  - 当$P_1 \geq P_2 \ldots \geq P_n$时，顺序检索需要使ASL最小，应该保持概率最大的元素在最前面，概率最小的元素在最后面。
- 改进1：把最近检索的元素放到第一个位置。
- 改进2：如果无法预先知道各个元素的查找概率，则可以用检索成功次数代替查找概率。
  - 当检索元素成功时，其检索成功次数加1
  - 保持检索成功次数最大的元素在前面，检索成功次数最小的元素在最后面。

10.2 二分检索

二分查找

- 基本思想：
  - 要求线性表已经按照关键码顺序排序。
  - 将字典中间位置上元素的关键码 key’ 和给定值 key 比较，若
    - key’==key，则检索成功；
    - key’ > key，在字典前半部分中继续进行二分法检索；
    - key’ < key，在字典后半部分中继续进行二分法检索。
  - 二分检索的实质是逐步缩小查找区间。

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为25的元素（成功检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
↑										↑
low										high

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为25的元素（成功检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
↑					↑					↑
low					mid					high

32 > 25

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为25的元素（成功检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
low				high						

二分查找

- 算法演示：
  – 分别用low和high表示当前查找区间的下界和上界
  – 每次取 mid = (low + high) // 2
  – 例如： 在下列待检索有序表中检索关键码为25的元素（成功检索情况）

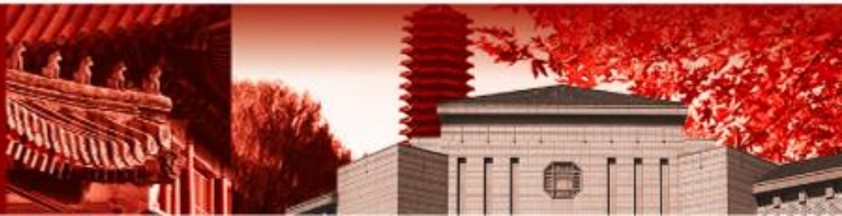
0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
low		mid		high						

18 < 25

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为25的元素（成功检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
			↑ low	↑ high						



二分查找

- 算法演示：
  – 分别用low和high表示当前查找区间的下界和上界
  – 每次取 mid = (low + high) // 2
  – 例如： 在下列待检索有序表中检索关键码为25的元素（成功检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99

low
high

mid
25 = 25

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如：在下列待检索有序表中检索关键码为25的元素（成功检索情况）

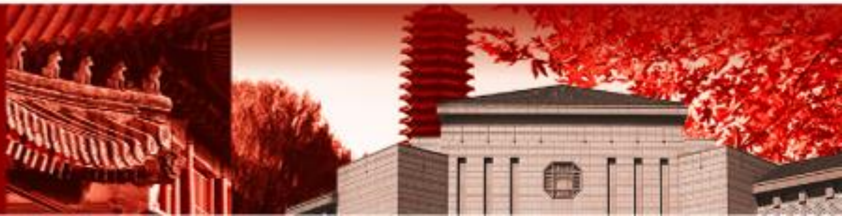
0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99

low（指向 3）　high（指向 4）

mid（指向 3）

25 = 25

Arr[mid]=25，检索成功



二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为76的元素（失败检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
low										high

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为76的元素（失败检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
low					mid					high

32 < 76

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为76的元素（失败检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
						↑				↑
						low				high

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为76的元素（失败检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
						↑		↑		↑
						low		mid		high

68 < 76

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为76的元素（失败检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
									low	high

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为76的元素（失败检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
									↑ low ↑ mid	↑ high

73 < 76

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为76的元素（失败检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99

high

low

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为76的元素（失败检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99

high

low

mid

99 >76

二分查找

- 算法演示：
  – 分别用low和high表示当前查找区间的下界和上界
  – 每次取 mid = (low + high) // 2
  – 例如： 在下列待检索有序表中检索关键码为76的元素（失败检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
									high	low

二分查找

- 算法演示：
  - 分别用low和high表示当前查找区间的下界和上界
  - 每次取 mid = (low + high) // 2
  - 例如： 在下列待检索有序表中检索关键码为76的元素（失败检索情况）

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
									high	low

high < low，检索失败

二分查找

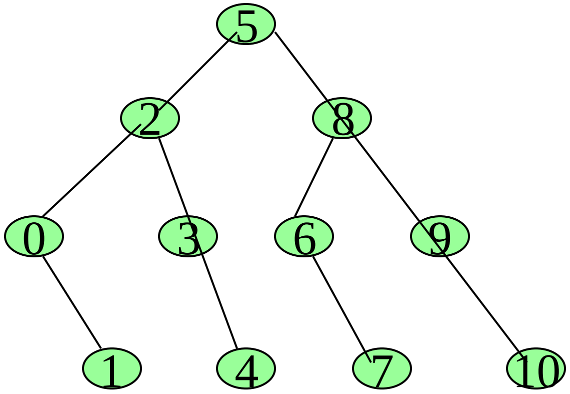
```python
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1

    return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

二分查找

- 时间复杂度分析：
  - 每比较一次缩小一半的查找区间。
  - 查找过程可用二叉树来描述。树中结点数字表示结点在有序表中的位置，通常称这个描述查找过程的二叉树为判定树。

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99

右图为11个元素的判定树。如要检索第0、3、6、9个元素需要3次比较；检索第1、4、7、10个元素需要4次比较。



二分查找

- 二分检索的过程恰好是在判定树中从根到检索结点的路径，关键码的比较次数取决于该结点在二叉树中的层数。
  - 假定总结点数$n=2^h - 1$，即层数$h=\log_2(n+1)$，则描述折半检索的判定树是一棵高度为h-1的满二叉树。
    - 根结点为第0层
    - 第 i 层的结点数量为 2^i，查询需要的比较次数为 i+1
  - 假定各个结点的检索概率相等

  其中h_i表示第i个结点的层数。

  - 当n很大时，得到：$ASL \approx \log_2(n+1)-1 = O(\log n)$

二分查找

- 二分检索的过程恰好是在判定树中从根到检索结点的路径，关键码的比较次数取决于该结点在二叉树中的层数。
  - 假定总结点数$n=2^h - 1$，即层数$h=\log_2(n+1)$，则描述折半检索的判定树是一棵高度为h-1的满二叉树。
    - 根结点为第0层
    - 第m层的结点数量为 2^m，查询需要的比较次数为 m+1
  - 假定各个结点的检索概率相等

$$\mathrm{ASL} = \sum_{i=1}^{n} p_i c_i = \frac{1}{n}\sum_{i=1}^{n}(h_i+1) = \frac{1}{n}\sum_{m=0}^{h-1}\sum_{j=1}^{2^m}(m+1) = \frac{1}{n}\sum_{m=0}^{h-1}(m+1)\cdot 2^m$$

$$= \frac{1}{n}\left(h\cdot 2^h - 2^h + 1\right) = \frac{1}{n}\left(h\cdot(n+1) - n\right) = \frac{n+1}{n}\log_2(n+1) - 1$$

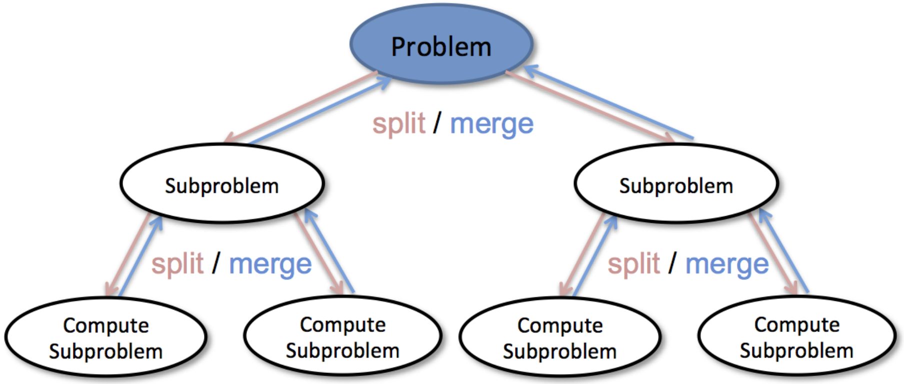
    - 其中h_i表示第i个结点的层数，即m
  - 当n很大时，得到：$\mathrm{ASL} \approx \log_2(n+1)-1 = O(\log n)$

二分查找

- 二分法检索的优点
  - 二分检索的效率要高于顺序检索。比较次数少，检索速度快。
  - 如n=1000时，顺序检索ASL≈500，而二分检索ASL≈9。
- 二分法检索缺点
  - 要求待检索按关键码排序，且只适用于顺序存储结构；
  - 对于大型的表，排序一次的计算成本十分昂贵。需要依据检索操作的频繁程度来衡量进行额外的排序是否值得。
  - 在动态场景，即字典的插入删除操作频繁的场景下，维护顺序表的有序性的成本也较高。
    - 此时可以采取二叉搜索树等基于树索引的检索方法。

二分查找

- 二分查找算法实际上体现了解决问题的一个典型策略：分而治之（Divide and Conquer）
  - 将问题分为若干更小规模的部分；通过解决每一个小规模部分问题，并将结果汇总得到原问题的解
  - 二分查找法也适合用递归方式实现




```python
def binarySearch(alist, item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist)//2
        if alist[midpoint]==item:
          return True
        else:
          if item<alist[midpoint]:
            return binarySearch(alist[:midpoint],item)
          else:
            return binarySearch(alist[midpoint+1:],item)
```

分块查找

- 基本思想：“按块有序”，即在块的级别上有序
  - 设线性表中共有 n 个数据元素，将表分成 b 块
  - 额外设置一个索引表，记录每一块的最小或最大的关键码
  - 前一块最大关键码必须小于后一块最小关键码
  - 每一块中的关键码不一定有序
- 顺序与二分法的折衷
  - 先在索引表中二分查找所在的块，然后在块中顺序查找目标元素
  - 既有较快的检索，又有较灵活的更改

分块查找

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
22	12	13	9	8		33	42	44	24	48		60	80	74	49	86	53

- 块起始位置
- 块内最大关键码
- 块内有效元素个数

link:	0	6	12
Key:	22	48	86
count:	5	5	6

分块查找

- 分块检索为两级检索
  - 先在索引表中确定待查元素所在的块，ASL_b
  - 然后在块内检索待查的元素，ASL_w
- 假设在索引表使用二分检索，在块内使用顺序检索
  - n为字典中的元素总数，b为分块的数量，s为块的最大长度

$$
\begin{aligned}
ASL &= ASL_b + ASL_w \\
&\approx \log_2 (b+1)-1 + (s+1)/2 \\
&\approx \log_2(1+n / s) + s/2
\end{aligned}
$$

分块查找

- 分块检索为两级检索
  - 先在索引表中确定待查元素所在的块，ASL_b
  - 然后在块内检索待查的元素，ASL_w
- 假设在索引表中用顺序检索，在块内也用顺序检索
  - 当 $s = \sqrt{n}$ 时，ASL 取最小值。
  - $ASL = \sqrt{n} + 1 \approx \sqrt{n}$

$$ASL_b = \frac{b+1}{2} \qquad ASL_w = \frac{s+1}{2}$$

$$ASL = \frac{b+1}{2} + \frac{s+1}{2} = \frac{b+s}{2} + 1$$

$$= \frac{n+s^2}{2s} + 1$$

注：

为了找到最小值，我们对 $f(s)$ 关于 s 求导，并令导数等于零：

$$f(s) = \frac{n}{2s} + \frac{s}{2}$$

$$f'(s) = -\frac{n}{2s^2} + \frac{1}{2}$$

令 $f'(s) = 0$：

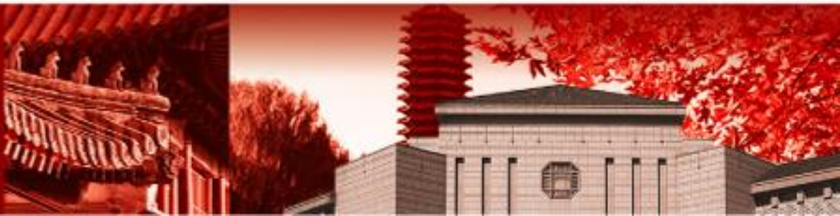
$$-\frac{n}{2s^2} + \frac{1}{2} = 0$$

$$\frac{n}{2s^2} = \frac{1}{2}$$

$$\frac{n}{s^2} = 1$$

$$s^2 = n$$

$$s = \sqrt{n}$$

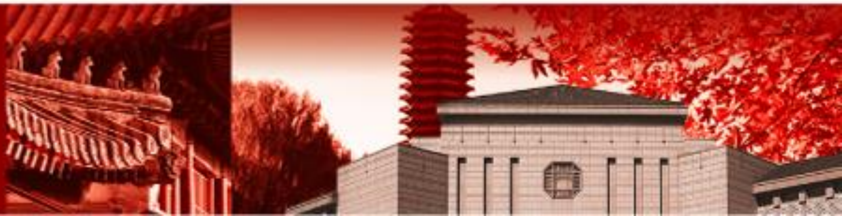


分块查找

- 优点：
  - 插入、删除相对较易
  - 没有大量记录移动
- 缺点：
  - 增加一个辅助数组的存储空间
  - 初始线性表分块排序同样需要代价
  - 当大量插入/删除时，或结点分布不均匀时，速度下降

比较：顺序检索 & 二分检索

- 虽然二分查找在时间复杂度上优于顺序查找，但也要考虑到对数据项进行排序的开销
  - 如果一次排序后可以进行多次查找，那么排序的开销就可以摊薄。
  - 但如果数据集经常变动，查找次数相对较少，那么可能还是直接用无序表加上顺序查找来得经济
- 所以，在算法选择的问题上，光看时间复杂度的优劣是不够的，还需要考虑到实际应用的情况。



10.4 散列检索

散列检索

- 散列法(hashing)的基本思想：
  - 直接根据记录的内容得到其存储位置
  - 插入关键码为key的字典元素时，按一个确定的散列函数 h 计算哈希值 h(key)，并把 h(key) 作为该元素的存储地址，即散列地址。
  - 检索时，同样根据 h(key) 得到关键码所在元素的存储地址。
- 散列法也称哈希法、杂凑法。
- 散列表/哈希表：用散列法表示的字典。
  - 散列表中的每一个存储位置，称为槽（slot），可以用来保存数据项。

散列检索

- 散列函数示例：
  - 有如下关键码的数据项：54，26，93，17，77，31
  - 最常见的散列函数之一是“求余数”，将数据项除以散列表的大小，得到的余数作为槽号。
    - h(item)= item % 11
  - 例如，如果需要查找关键码26，计算出其哈希值4，直接访问下标为4的元素即可。

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

散列地址	0	1	2	3	4	5	6	7	8	9	10	11
关键码	77				26	93	17			31	54	

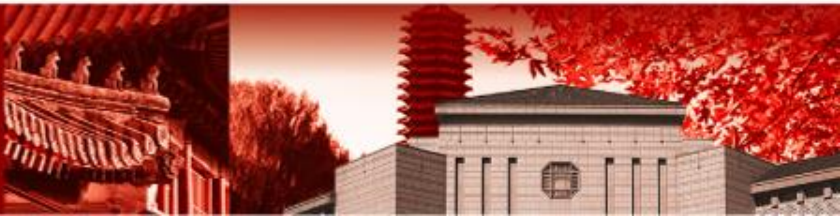
散列检索

- 不过，这个方案有很明显的缺陷。
  - 例如，如果要插入新元素44，其哈希值为0，就会被分配到已有元素77所在的槽中。
- 这种现象称为碰撞(Collision)，发生碰撞的两个(或多个)关键码称为同义词。
  - 即对于两个不相等的key1, key2，散列函数使得 h(key1) = h(key2)
- h(key)的值域所对应的地址空间称为基本区域。
- 发生碰撞时，同义词可以存放在基本区域中未被占用的单元，也可以放到基本区域以外另开辟的区域（溢出区）。

散列检索

- 散列表的构造和检索中，需要解决的主要问题：

  1. 定义散列表和散列函数
     - 散列函数：散列表如何检索？
     - 检索效率如何？检索效率与哪些因素有关？
  2. 解决碰撞
     - 对于任意的散列函数，都可能出现“碰撞”现象
     - 碰撞发生时如何处理？



散列函数

- 散列函数的选择标准
  - 散列函数应该将关键字均匀映射到到整个地址空间中，从而尽可能减少碰撞。即落在任意一个槽中的概率应该均等。
  - 散列函数本身的计算应该尽可能简单。
  - 散列函数应该使得任意关键字的哈希值都落在表长范围内。

散列函数

- 完美散列函数
  - 给定一组数据项，如果一个散列函数能把每个数据项映射到不同的槽中，那么这个散列函数就可以称为“完美散列函数”
  - 对于固定的一组数据，总是能想办法设计出完美散列函数
- 但如果数据项经常性的变动，很难有一个系统性的方法来设计对应的完美散列函数。

散列函数

- 常见的散列函数
  - 直接定址法
  - 数字分析法
  - 平方取中法
  - 折叠法
  - 除留余数法
  - 基数转换法

散列函数

- 散列函数：1. 直接定址法
  - 关键字的线性函数，h(key) = a * key + b
  - 例：1~100岁人口数字统计表（a=1,b=0, key=年龄）

地址	01	02	03	...	25	26	27	...	100
年龄	1	2	3	...	25	26	27	...	100
人数	3000	2000	5000	...	1050	...	...	...	...

  - 适用于关键码分布基本连续情况。若关键码分布不连续，容易造成空单元，存储空间浪费。

散列函数

- 散列函数：2. 数字分析法
  - 关键码的位数远多于存储区的地址码位数，这时可以对各位进行分析，丢掉分布不均匀的位而留下均匀的位作为地址。
  - 例1：关键码为 395003、395010、395012、395085、395097
    - 前四位相同，后两位随机分布。故可以取后两位构成散列函数，得到的散列地址为（03，10，12，85，97）
  - 例2：大学学生学号
    - 前缀编码了年级、院系等信息，分布较不均匀。后几位通常是随机均匀分布的，可以作为散列地址。
  - 通常用于已知记录关键码，并且关键码各位分布已经知道的情况，适合于静态的字典。

散列函数

- 散列函数：3. 平方取中法
  - 先求出关键码的平方，然后取中间若干位构成散列函数。
  - 例如：key = 4731，key^2 = 22382361，如地址码为3位，则可以取382为散列地址，即 h(4731) = 382
  - 这是一种较常用的构造散列函数的方法。设计思想是，一个数平方的中间几位数和每一位都相关，由此使均匀分布的关键码得到的散列地址也是比较均匀的。
  - 取的位数由表长决定。

散列函数

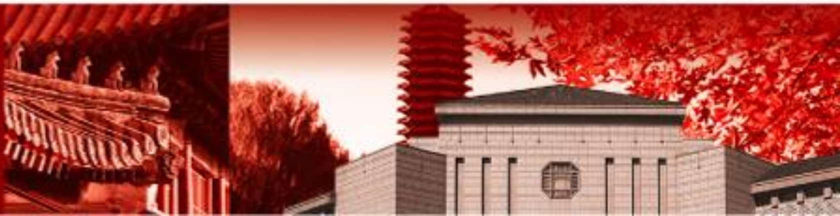
- 散列函数：4. 折叠法
  - 如果关键码的位数多于地址位数，且关键码中每一位分布均匀时，此时可以将关键码分成若干部分，其中一部分的长度等于地址位数。各部分相加，舍弃进位，最后的和作为散列地址。
  - 例如：key = 05 8242 2241，地址位数为4。
  - 移位相加： (2241 + 8242 + 05) % 10000 = 0488
  - 间界叠加（隔数反转）：(2241 + 2428 + 05) % 10000 = 4674
    - 隔数反转，即相加时每隔一个数就将数字反转
    - 虽然隔数反转从理论上看来毫无必要，但这个步骤确实为折叠法得到散列函数提供了一种常用的微调手段。

散列函数

- 散列函数：5.余数法
  - h(key) = key % p
  - p的选择非常重要，通常选择小于等于散列表长度m的某个最大素数。
  - 散列表长设置为m时，通常取的p值：

m=	8	16	32	64	128	256	512	1024
p=	7	13	31	61	127	251	503	1019

- 思考：为什么要求p是素数？

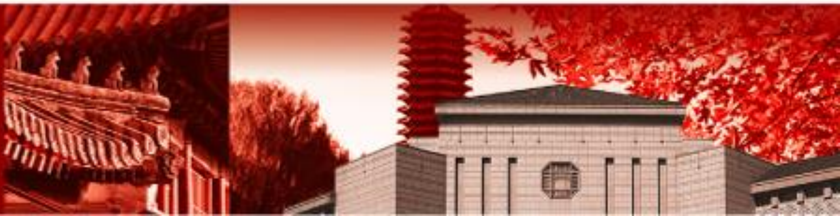


散列函数

- 散列函数：6.基数转换法
  - 将关键码首先看作是另一进制的表示，然后再转换为原来的进制数，并用数字分析法取若干位作为散列地址。
  - 一般转换基数大于原基数，且最好二者互素。
  - 例如： key = $(236075)_{10}$，把它看作13进制数 $(236075)_{13}$,
    - 再转换为10进制：$(236075)_{13} = 2*13^5+3*13^4+6*13^3+7*13^1+5 = (841547)_{10}$
    - 假设地址位数为4，选择 841547 的2~5位
    - 得到散列地址 h(236075) = 4154

碰撞处理

- 无论选择何种散列函数，碰撞都可能发生。换句话讲，合适的散列函数可以减少碰撞发生的几率，但不能保证不发生碰撞。碰撞发生时如何处理？
- 碰撞处理
  - 开地址法（探查法）：为冲突的数据项再找一个开放的空槽
  - 拉链法：将容纳单个数据项的槽扩展为容纳数据项集合（或者对数据项链表的引用）



碰撞处理：开地址法

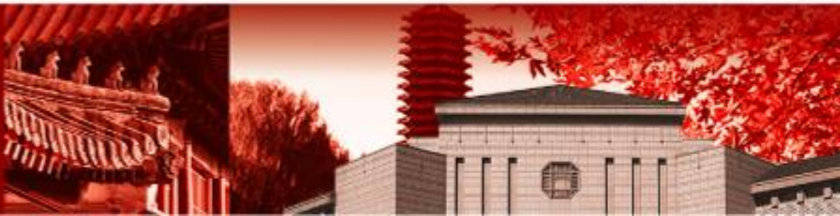
- 开地址法基本思想：
  - d_0 = h(hey) 称为关键码key的基地址
  - 当碰撞发生时，用某种方法在基本区域内确定一个探查序列。

$$d_i = d_0 + p(key, i)$$

  - p称为探查函数，d_i称为后继散列地址
- 按照确定探查序列的方式分为：
  - 线性探查法
  - 双散列函数法

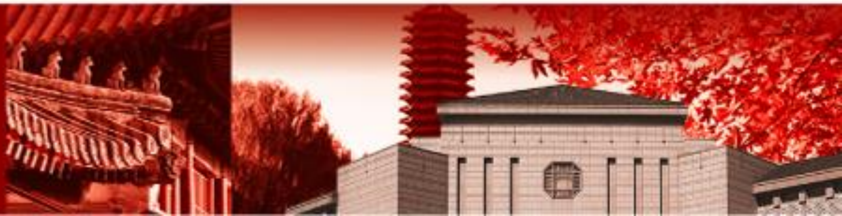
碰撞处理：开地址法

- 插入关键码key时，若发生了碰撞：
  - 则按照探查函数生成的探查序列一次查找
  - 将找到的第一个空闲位置di作为key的存储位置
  - 若后续散列地址都不空闲，说明散列表已满，报告溢出
- 检索关键码key时：
  - 依然首先计算基地址，以及相应的探查序列
  - 在探查序列中依次遍历查找关键码key
  - 如果遇到了空闲位置，表示探查序列结束，检索失败
- 插入和检索时都需要考虑表满的情况
  - 探查序列可能会进入一个无限循环中
  - 可以限制探查序列的长度



线性探查法

- 线性探查法（Linear Probing）
  - 若在地址为d (d=h(key)) 的单元发生碰撞：
  - 则探查序列为：d+1,d+2,…,m-1,0,1,…,d-1 (m为基本存储区的长度)
    - 相当于将基本存贮区看作一个循环表，进行顺序遍历。
  - 如果从单元d开始探查，查找一遍后又回到地址d，则表示基本存贮区已经溢出。



线性探查法

- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13 ；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码													

线性探查法

- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码						18							

线性探查法

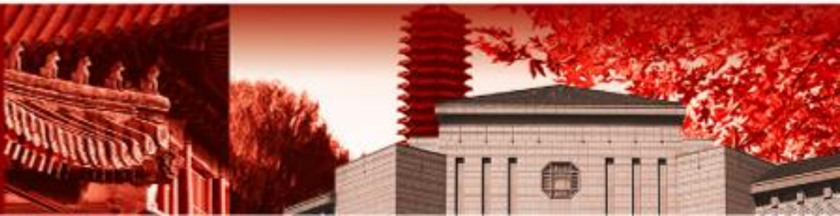
- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码						18			73				

线性探查法

- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码						18			73		10		



线性探查法

- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码						18			73		10		

线性探查法

- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码						18	05		73		10		

线性探查法

- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码				68		18	05		73		10		

线性探查法

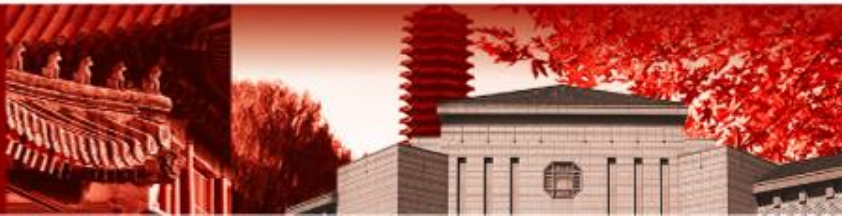
- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码				68		18	05		73		10		

线性探查法

- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

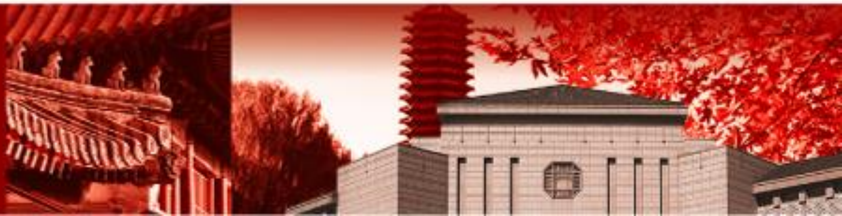
散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码				68		18	05		73	99	10		



线性探查法

- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27		68		18	05		73	99	10		



线性探查法

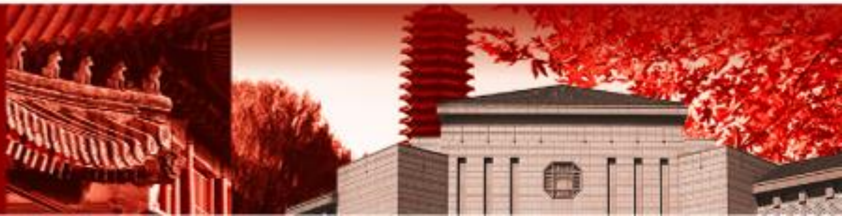
- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27	41	68		18	05		73	99	10		

线性探查法

- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27	41	68		18	05		73	99	10		51



线性探查法

- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13 ；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27	41	68		18	05		73	99	10		51

  - 注意：虽然关键码为32与05的元素发生碰撞，但二者并非同义词。在插入05时也发生了碰撞，这导致05插入到了散列地址为6的槽中

线性探查法

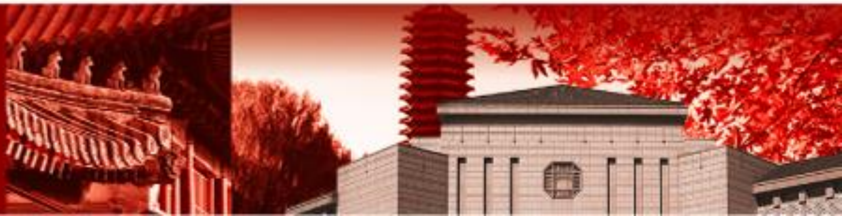
- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27	41	68		18	05	32	73	99	10		51

线性探查法

- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

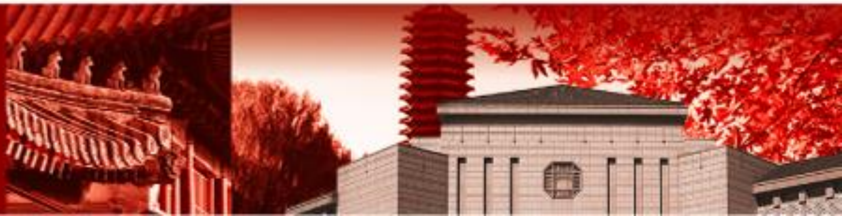
散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27	41	68		18	05	32	73	99	10		51



线性探查法

- 线性探查示例：依次插入元素
  - 设关键码集合为： K = {18，73，10，05，68，99，27，41，51，32，25}
  - 散列表长度为13，散列函数为h(key)=key % 13；
  - 计算基地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查。
  - 查找元素时进行相同的线性探查过程即可。
  - 哈希值：h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码	25	27	41	68		18	05	32	73	99	10		51



线性探查法：聚集问题

- 聚集：哈希值不同的关键码产生的探查序列相互重叠，在同一个地址序列上堆积，导致很长的探查序列。
  - 可以改进一下探查序列，例如由线性函数改为二次函数，减少重叠
- 二次聚集：哈希值相同关键码产生的探查序列总是相同的的，插入哈希值相同的元素（同义词），不可避免地导致元素堆积在一起，也使得探查序列变长。
  - 根本原因在于，探查序列完全由哈希值确定，与关键码值本身无关。
- 为了减少二次聚集的产生，可以改进线性探查方法
  - 令探查序列不仅依赖于哈希值，也依赖于关键码值
  - 使得同义词也能够产生不同的探查序列，减少聚集

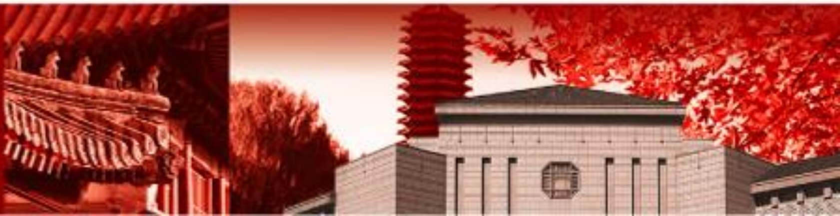
双散列函数法

- 双散列函数法选用两个散列函数h_1和h_2
  - 假设m为散列表长
  - h_1产生一个0到m-1之间的数作为地址。
  - h_2产生一个1到m-1之间的数作为探查序列的间隔
- 例如，$h_1(key)=key\%m$，$h_2(key) = key \% (m-1) + 1$。
  - 如果$d=h_1(key)$发生碰撞，则再计算$h_2(key)$，
  - 产生的探查序列为：$(d+h_2(key))\%m$, $(d+2*h_2(key))\%m$, …
  - 结果是，探查序列既依赖于哈希值，也依赖于关键码值
- h_2产生的间隔必须与m互素
  - 为了保证，如果表中存在空闲位置，探查序列总能遍历到它
  - 实际中，将 m 设置为素数就可以保证这一点

双散列函数法

- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：$h_1(key)=key\%13$，$h_2(key)=key\%11+1$
  - 计算基地址$d=h_1(key)$，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：$h_1(18)=5$, $h_1(73)=8$, $h_1(10)=10$, $h_1(05)=5$, $h_1(68)=3$, $h_1(99)=8$, $h_1(27)=1$, $h_1(41)=2$, $h_1(51)=12$, $h_1(32)=6$, $h_1(25)=12$

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码													



双散列函数法

- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：h_1(key)=key%13，h_2(key)=key%11+1
  - 计算基地址d=h_1(key)，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：h_1 (18)=5, h_1 (73)=8, h_1 (10)=10, h_1 (05)=5, h_1 (68)=3, h_1 (99)=8, h_1 (27)=1, h_1 (41)=2, h_1 (51)=12, h_1 (32)=6, h_1 (25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码						18							

双散列函数法

- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：$h_1(key)=key\%13$，$h_2(key)=key\%11+1$
  - 计算基地址$d=h_1(key)$，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：$h_1(18)=5$, $h_1(73)=8$, $h_1(10)=10$, $h_1(05)=5$, $h_1(68)=3$, $h_1(99)=8$, $h_1(27)=1$, $h_1(41)=2$, $h_1(51)=12$, $h_1(32)=6$, $h_1(25)=12$

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码						18			73				

双散列函数法

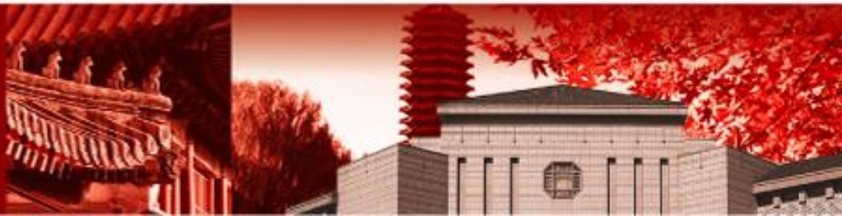
- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：h_1(key)=key%13，h_2(key)=key%11+1
  - 计算基地址d=h_1(key)，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：h_1 (18)=5, h_1 (73)=8, h_1 (10)=10, h_1 (05)=5, h_1 (68)=3, h_1 (99)=8, h_1 (27)=1, h_1 (41)=2, h_1 (51)=12, h_1 (32)=6, h_1 (25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码						18			73		10		

双散列函数法

- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：$h_1(key)=key\%13$，$h_2(key)=key\%11+1$
  - 计算基地址$d=h_1(key)$，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：$h_1(18)=5$, $h_1(73)=8$, $h_1(10)=10$, $h_1(05)=5$, $h_1(68)=3$, $h_1(99)=8$, $h_1(27)=1$, $h_1(41)=2$, $h_1(51)=12$, $h_1(32)=6$, $h_1(25)=12$
  - 关键码为05的元素发生碰撞，计算$h_2(5)=6$

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码						18			73		10		



双散列函数法

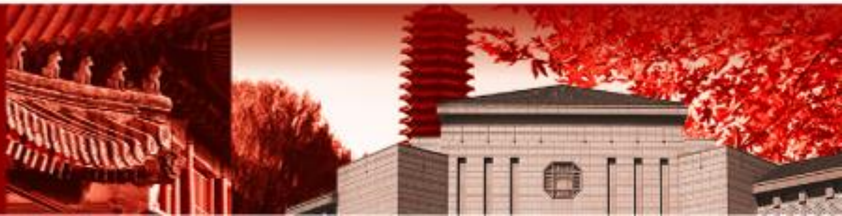
- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：$h_1(key)=key\%13$，$h_2(key)=key\%11+1$
  - 计算基地址$d=h_1(key)$，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：$h_1(18)=5$, $h_1(73)=8$, $h_1(10)=10$, $h_1(05)=5$, $h_1(68)=3$, $h_1(99)=8$, $h_1(27)=1$, $h_1(41)=2$, $h_1(51)=12$, $h_1(32)=6$, $h_1(25)=12$
  - 关键码为05的元素发生碰撞，计算$h_2(5)=6$

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码						18			73		10	05	

双散列函数法

- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：$h_1(key)=key\%13$，$h_2(key)=key\%11+1$
  - 计算基地址$d=h_1(key)$，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：$h_1(18)=5$, $h_1(73)=8$, $h_1(10)=10$, $h_1(05)=5$, $h_1(68)=3$, $h_1(99)=8$, $h_1(27)=1$, $h_1(41)=2$, $h_1(51)=12$, $h_1(32)=6$, $h_1(25)=12$

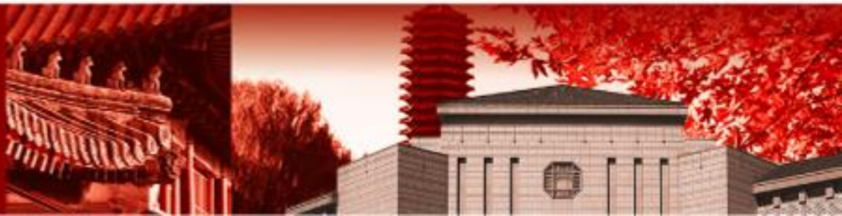
散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码				68		18			73		10	05	



双散列函数法

- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：h_1(key)=key%13，h_2(key)=key%11+1
  - 计算基地址d=h_1(key)，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：h_1 (18)=5, h_1 (73)=8, h_1 (10)=10, h_1 (05)=5, h_1 (68)=3, h_1 (99)=8, h_1 (27)=1, h_1 (41)=2, h_1 (51)=12, h_1 (32)=6, h_1 (25)=12
  - 关键码为99的元素发生碰撞，计算h_2(99)=1

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码				68		18			73		10	05	



双散列函数法

- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：$h_1(key)=key\%13$，$h_2(key)=key\%11+1$
  - 计算基地址$d=h_1(key)$，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：$h_1(18)=5$, $h_1(73)=8$, $h_1(10)=10$, $h_1(05)=5$, $h_1(68)=3$, $h_1(99)=8$, $h_1(27)=1$, $h_1(41)=2$, $h_1(51)=12$, $h_1(32)=6$, $h_1(25)=12$
  - 关键码为99的元素发生碰撞，计算$h_2(99)=1$

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码				68		18			73	99	10	05	

双散列函数法

- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：$h_1(key)=key\%13$，$h_2(key)=key\%11+1$
  - 计算基地址$d=h_1(key)$，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：$h_1(18)=5$, $h_1(73)=8$, $h_1(10)=10$, $h_1(05)=5$, $h_1(68)=3$, $h_1(99)=8$, $h_1(27)=1$, $h_1(41)=2$, $h_1(51)=12$, $h_1(32)=6$, $h_1(25)=12$

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27		68		18			73	99	10	05	

双散列函数法

- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：h_1(key)=key%13，h_2(key)=key%11+1
  - 计算基地址d=h_1(key)，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：h_1 (18)=5, h_1 (73)=8, h_1 (10)=10, h_1 (05)=5, h_1 (68)=3, h_1 (99)=8, h_1 (27)=1, h_1 (41)=2, h_1 (51)=12, h_1 (32)=6, h_1 (25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27	41	68		18			73	99	10	05	

双散列函数法

- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：h_1(key)=key%13，h_2(key)=key%11+1
  - 计算基地址d=h_1(key)，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：h_1 (18)=5, h_1 (73)=8, h_1 (10)=10, h_1 (05)=5, h_1 (68)=3, h_1 (99)=8, h_1 (27)=1, h_1 (41)=2, h_1 (51)=12, h_1 (32)=6, h_1 (25)=12

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27	41	68		18			73	99	10	05	51

双散列函数法

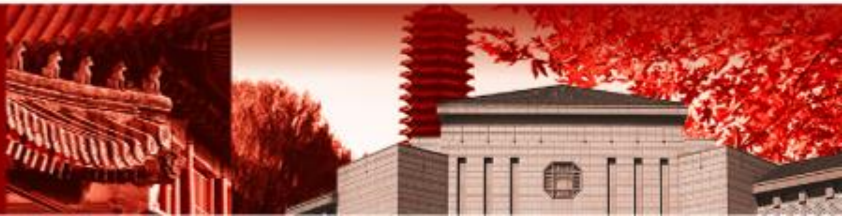
- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：$h_1(key)=key\%13$，$h_2(key)=key\%11+1$
  - 计算基地址$d=h_1(key)$，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：$h_1(18)=5$, $h_1(73)=8$, $h_1(10)=10$, $h_1(05)=5$, $h_1(68)=3$, $h_1(99)=8$, $h_1(27)=1$, $h_1(41)=2$, $h_1(51)=12$, $h_1(32)=6$, $h_1(25)=12$

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27	41	68		18	32		73	99	10	05	51

双散列函数法

- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：$h_1(key)=key\%13$，$h_2(key)=key\%11+1$
  - 计算基地址$d=h_1(key)$，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：$h_1(18)=5$, $h_1(73)=8$, $h_1(10)=10$, $h_1(05)=5$, $h_1(68)=3$, $h_1(99)=8$, $h_1(27)=1$, $h_1(41)=2$, $h_1(51)=12$, $h_1(32)=6$, $h_1(25)=12$
  - 关键码为25的元素发生碰撞，计算$h_2(25)=4$

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27	41	68		18	32		73	99	10	05	51



双散列函数法

- 双散列函数法示例：依次插入元素
  - K={18，73，10，05，68，99，27，41，51，32，25}
  - m=13，h1、h2函数分别为：$h_1(key)=key\%13$，$h_2(key)=key\%11+1$
  - 计算基地址$d=h_1(key)$，若地址未被占用，则插入新结点；否则用h2函数计算探查间隔，直到找到未被占用的地址。
  - 哈希值：$h_1(18)=5$, $h_1(73)=8$, $h_1(10)=10$, $h_1(05)=5$, $h_1(68)=3$, $h_1(99)=8$, $h_1(27)=1$, $h_1(41)=2$, $h_1(51)=12$, $h_1(32)=6$, $h_1(25)=12$
  - 关键码为25的元素发生碰撞，计算$h_2(25)=4$

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27	41	68		18	32	25	73	99	10	05	51

思考

- 插入元素的过程中，如果在探查序列中发现了关键码相同的元素，应该如何操作？

思考

- 插入元素的过程中，如果在探查序列中发现了关键码相同的元素，应该如何操作？
  - 应该直接返回插入失败
  - 回忆：必须保证每个元素的关键码在字典中是唯一的
  - 关键码是我们区分不同元素的唯一途径。如果关键码有重复，我们就无法区分这些元素。

开地址法：删除元素

- 在使用开地址法的哈希表中删除元素时，需要关注两点：
  - 释放的位置要能够为将来所用
  - 删除元素不能影响后续的检索
- 思考：将元素直接简单删除有什么问题？

删除元素带来的问题

- 考虑长度m=13的如下哈希表：
  - 对于关键码 K1, K2，以及它们的同义词(K1', K2')，h(K1)=2, h(K2)=6
  - 进一步，采取对应的探查序列为（假设探查序列仅依赖于哈希值，并非线性探查）
    - K1: 2, 3, 1, 6, 11, ……
    - K2: 6, 7, 5, 10, 2, ……
  - (1) 插入K2，以及其所有的同义词K2'

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码			K2'			K2'	K2	K2'			K2'		

删除元素带来的问题

- 考虑长度m=13的如下哈希表：
  - 对于关键码 K1, K2， 以及它们的同义词(K1’, K2’)，h(K1)=2, h(K2)=6
  - 进一步， 采取对应的探查序列为（假设探查序列仅依赖于哈希值，并非线性探查）
    - K1: 2, 3, 1, 6, 11, ……
    - K2: 6, 7, 5, 10, 2, ……
  - (2) 插入K1， 以及其所有的同义词K1’

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		K1’	K2’	K1		K2’	K2	K2’			K2’		

删除元素带来的问题

- 考虑长度m=13的如下哈希表：
  - 对于关键码 K1, K2， 以及它们的同义词(K1’, K2’)，h(K1)=2, h(K2)=6
  - 进一步，采取对应的探查序列为（假设探查序列仅依赖于哈希值，并非线性探查）
    - K1: 2, 3, 1, 6, 11, ……
    - K2: 6, 7, 5, 10, 2, ……
  - (3) 删除位置6上的元素
    - 如果直接删除，K2的所有同义词K2’就无法被查找到

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		K1’	K2’	K1		K2’		K2’			K2’		

删除元素带来的问题

- 考虑长度m=13的如下哈希表：
  - 对于关键码 K1, K2，以及它们的同义词(K1’, K2’)，h(K1)=2, h(K2)=6
  - 进一步，采取对应的探查序列为（假设探查序列仅依赖于哈希值，并非线性探查）
    - K1: 2, 3, 1, 6, 11, ……
    - K2: 6, 7, 5, 10, 2, ……
  - (3) 删除位置6上的元素
    - 为了保证K2’的同义词仍然能够被正常找到，我们或许可以这样做：把K2的探查序列最后位置（即位置2）上的元素替换到这里来
    - 但就会影响K1的检索：K1及其同义词将查不到

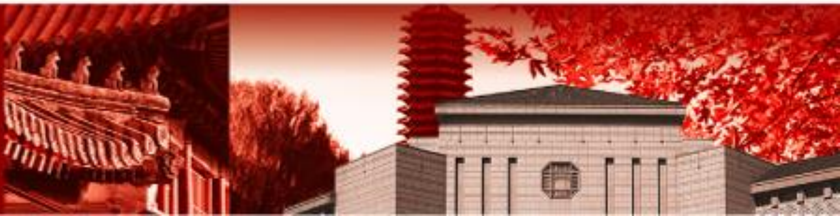
散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		K1’		K1		K2’	K2’	K2’			K2’		

开地址法：删除元素

- 对于上述例子中的6号位置，我们需要一个特殊的标记位
  - 表示该位置上不存在元素，但是探查过程不应该停止在这里
- 称这个被删除的标记为“墓碑”(Tombstone)
- 删除元素时：
  - 首先沿着探查序列定位到待删除位置
  - 直接将该位置的值设置为 TOMB 值（特殊标记）

开地址法：删除元素

- 对于上述例子中的6号位置，我们需要一个特殊的标记位
  – 表示该位置上不存在元素，但是探查过程不应该停止在这里
- 称这个被删除的标记为“墓碑”(Tombstone)
- 插入元素时：
  – 仍沿着探查序列遍历。遇到墓碑，应当视为已被占用的位置继续遍历
  – 遇到墓碑直接占用是错误的：插入过程需要确保表中的关键码不重复，必须遍历整个探查序列直到遇到空位。
  – 如果遇到重复元素，应返回插入失败；
  – 遇到空位后，即可确保表中无重复元素，可以插入关键码
    - 如果未曾遇到墓碑，正常插入空位置即可
    - 如果遇到过墓碑，就可以将关键码插入到第一个墓碑的位置



开地址法：删除元素

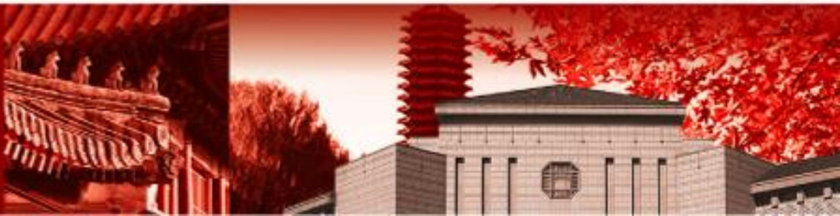
- 对于上述例子中的6号位置，我们需要一个特殊的标记位
  - 表示该位置上不存在元素，但是探查过程不应该停止在这里
- 称这个被删除的标记为“墓碑”(Tombstone)
- 检索元素时：
  - 沿着探查序列遍历，直到遍历到空位置或者找到目标关键码
  - 遇到墓碑，应当视为已被占用的位置继续遍历

开地址法：检索效率分析

- 衡量标准：检索操作的ASL
  - 删除操作时必须先找到元素，代价相当于进行一次成功的检索
  - 插入元素时无论是否考虑墓碑，都必须先找到探查序列的尾部，代价相当于进行一次失败检索
- 散列效率与负载因子(Load Factor) α 有关

$$负载因子\, \alpha = \frac{散列表中的已占用的位置数}{散列表长}$$

  - α较小时，散列表比较空，各位置上的探查序列都比较短
  - α较大时，插入时的碰撞更频繁，探查序列较长
  - 如果散列表中具有墓碑标记位，应该视为已占用的位置



开地址法：检索效率分析

- 例：散列表长m=15，关键码为 (26, 36, 41, 38, 44, 15, 68, 12, 06, 51, 25)，散列函数为h(key)=key%13。使用线性探查法解决冲突。插入全部关键码后的结果如下：

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
关键码	26	25	41	15	68	44	6				36		38	12	51
探查序列中的位置	1	5	1	2	2	1	1				1		1	2	3

- 成功检索的ASL（删除元素的代价）：
  - 假设：表中每个关键码被检索的概率均等
  - 但每个元素在探查序列中的位置不同
  - $ASL_{\mathrm{succ}} = \frac{1}{11}(1+5+1+2+2+1+1+1+1+2+3) = \frac{20}{11}$

开地址法：检索效率分析

- 例：散列表长m=15，关键码为 (26, 36, 41, 38, 44, 15, 68, 12, 06, 51, 25)，散列函数为h(key)=key%13。使用线性探查法解决冲突。插入全部关键码后的结果如下：

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
关键码	26	25	41	15	68	44	6				36		38	12	51
探查序列中的位置	1	5	1	2	2	1	1				1		1	2	3

- 失败检索的ASL（插入元素的代价）：
  - 假设：检索的key均匀分布，即目标基地址落在0-12槽的概率均等
  - 每个槽的探查序列中包含的关键码数量不同
  - $ASL_{\text{unsucc}} = \frac{1}{13}(8+7+6+5+4+3+2+1+1+1+2+1+11) = \frac{52}{13} = 4$

开地址法：检索效率分析

- 结论1：给定散列函数，散列表的检索代价不依赖于表长，只依赖于负载因子。随着负载因子的增加，预期的代价也会增加
  - 实践中，通常可以设定一个负载因子的临界值，例如 0.5、0.75、2/3
  - 大于这个临界值，散列表的性能就会急剧下降
  - 如果：(1) 散列函数产生的哈希值在哈希表中分布均匀；(2) 负载因子不大（低于经验值），则可以认为插入、删除、检索操作的代价都是O(1)的。

开地址法：检索效率分析

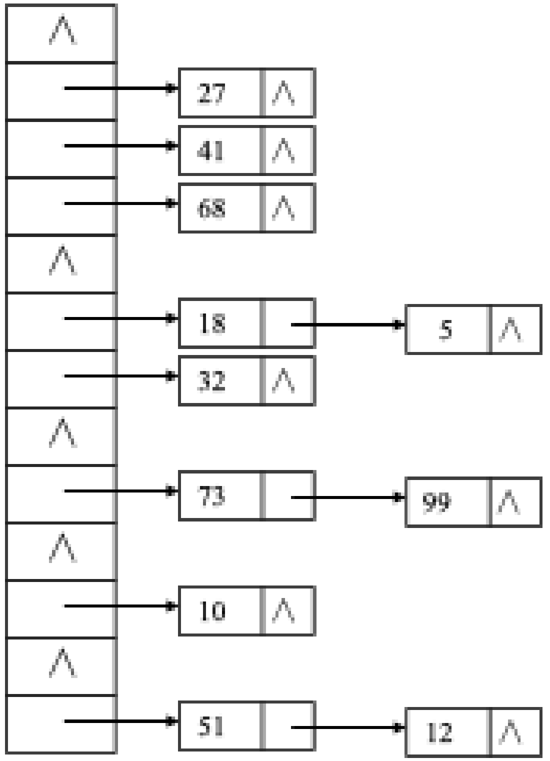
- 结论2：散列表的插入和删除操作如果很频繁，将降低散列表的检索效率
  - 大量的插入操作将增加负载因子，使同义词的数量增加，也使平均检索长度ASL增加
  - 大量的删除操作将增加墓碑的数量，增加元素本身到基地址的平均长度
- 应用中，对于插入和删除操作比较频繁的散列表，可以定期对表进行重新散列
  - 将所有记录插入至新的表中
  - 清除所有墓碑
  - 把最频繁访问的记录放到基地址处

碰撞处理：拉链法

- 拉链法基本思想：
  - 假设哈希表的基本区域长度为m，表中不存储关键码值，每个元素是一个对关键码链表的引用。
  - 插入元素时，直接插入到对应槽中的关键码链表末尾。
  - 检索元素时，沿着对应槽中的关键码链表遍历比较。

拉链法

- 拉链法示例：
  - K={18，73，10，05，68，99，27，41，51，32，12}
  - 散列函数为h(key)=key%13，插入新结点时，将结点插入到链表中
- 有时把存放散列函数值相同的记录（同义词）的结构称为桶，则整个散列结构由一个指向桶的指针数组和若干个桶组成，称为桶散列。



拉链法

- 检索关键码值为key的元素
  - 计算基地址h(key)
  - 在基地址对应的同义词链表中顺序查找该关键码
  - 如果同义词链表为空，或不包含目标关键码，则检索失败
- 插入关键码值为key的元素
  - 首先执行对该元素的查询
  - 如果检索成功，表明关键码重复，插入失败
  - 如果检索失败，就向同义词链表追加关键码值即可
- 删除关键码值为key的元素
  - 首先执行对该元素的查询
  - 如果检索失败，表明表中不存在该元素，删除失败
  - 如果检索成功，在同义词链表中删除该元素即可

拉链法

- 拉链法的优势：
  - 由于各链表上的结点空间是动态申请的，因此更适应于构造表前无法确定表长的情况；
  - 避免了聚集问题，哈希值不同的关键码的检索过程不会重叠
  - 结点的删除操作较方便，不需要设置墓碑作为特殊标记位
- 拉链法散列表的负载因子为同义词链表的平均长度 n/m。
  - n为总关键码数量，m为散列表长度
  - 一般设计 n/m 大于1小于10
  - 负载因子越高，意味着同义词链表的平均长度越长，插入删除检索的代价越高

实例：Python中的dict, set

- Python内置了dict 类型，存储一系列 key-value 对，支持按照关键码 key 的高效检索。
  - dict的实现基于散列表（哈希表），并使用了线性探查的开地址法
  - set类型的实现采取了相同方法，只不过set只需要存储 key，无需存储 value
- 哈希函数为 h(key) = hash(key) % m，m为哈希表大小
  - Python内置了hash函数，接受不可变对象（数值，字符串，元组）作为关键码，返回整数哈希值
  - key必须是不可变类型，因为必须保证同一元素的哈希值不会改变
  - 自定义的类型实现__hash__方法，之后也可以作为hash函数的参数

实例：Python中的dict, set

- dict，set不定长，同样具有动态扩容机制
- 插入元素时，如果散列表的负载因子超过临界值2/3，就会触发扩容
  - 扩容时，表长会扩大至2倍或4倍，并进行重新散列，将旧表中的所有元素插入至新哈希表中
- 删除元素则不会触发扩容，因为删除操作不改变散列表的负载因子

课堂练习

- 考虑关键字集合K={19，1，23，34，20，84，27，53，11，56，29}，设计一个散列表：
  – 固定表长为13，地址空间为0~12
  – 采用哈希函数：h(key)=key%13，
- (1)使用线性探查法解决碰撞。写出依次插入上述关键码后的散列表结果
- (2)使用双散列函数法解决碰撞，且$h_2(key)=key\%11+1$。写出依次插入上述关键码后的散列表结果