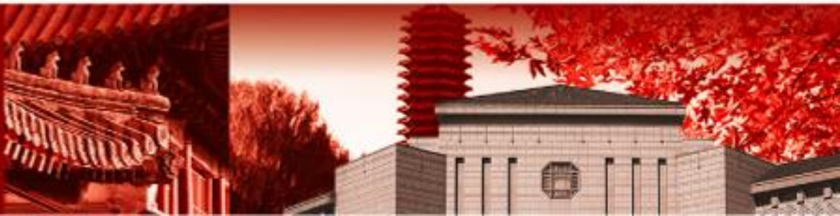
数据结构与算法B

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

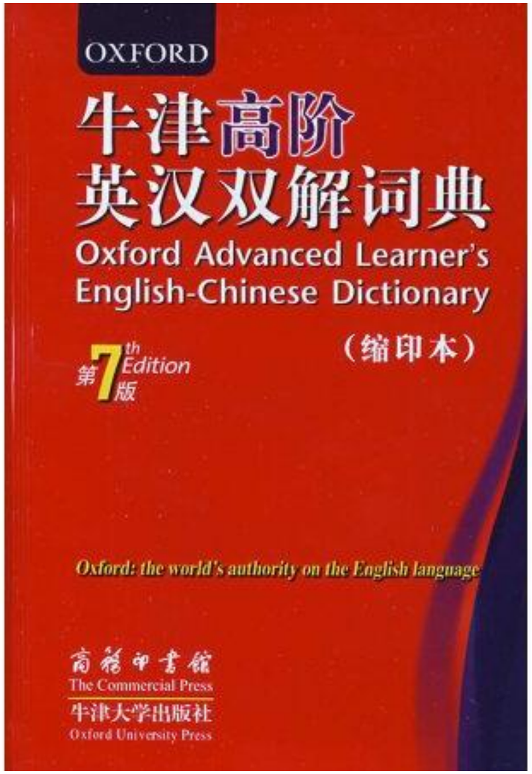
- 字典(Dictionary)是元素的有穷集合，其主要的操作为对元素的检索。
- 字典中的每个元素由两部分组成，分别称为元素的关键码(key)和属性 (attribute)。
  - 关键码本质上是一个特殊的属性
  - 必须保证字典中的每个元素具有唯一的关键码
    - 比如：学号就是每个同学的关键码，身份证号就是每个公民的关键码。
    - 可以通过关键码来查询元素的其他属性
    - 如果有两个元素关键码相同，我们就无法区分这两个元素

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，有时也要考虑字典的插入和删除操作
- 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：

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

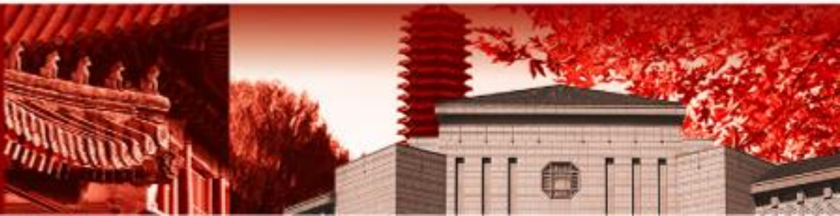
  - n是元素的个数；p_i是查找第i个元素的概率；c_i是算法为了找到第i个元素所需的比较次数。
  - 除非特别声明，一般假定各元素的检索概率相等，即p_i=1/n

北京大学

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 (3)　↑ high (4)

↑ mid (3)

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 ↑（位置 3） high ↑（位置 4）

low

high

mid

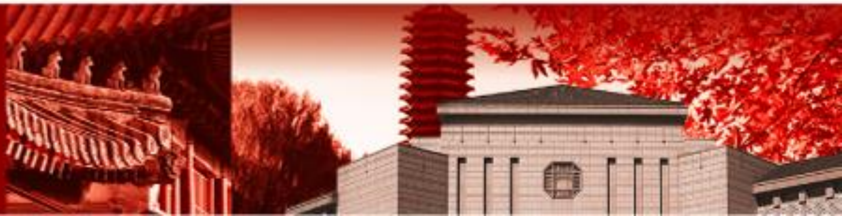
25 = 25

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　　high

mid

$73 < 76$

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

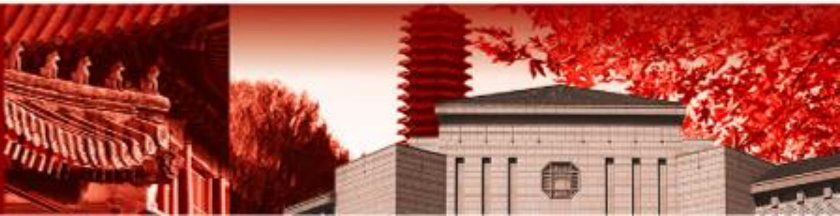
二分查找

```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))
```

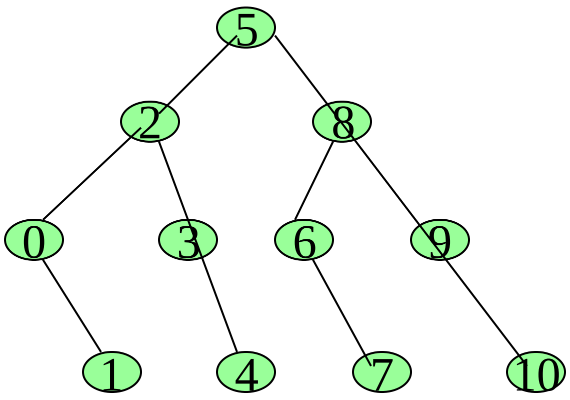


二分查找

- 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个记录的判定树。如要检索第0、3、6、9个记录需要3次比较；检索第1、4、7、10个记录需要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。
  - 假定总记录数$n=2^h - 1$， 即$h=\log_2(n+1)$，则描述折半检索的判定树是一棵深度为h-1的满二叉树。
  - 除第 h-1 层外，第 i 层的结点数量为 2^i，查询需要的比较次数为 i+1
    - 根节点为第0层
  - 假定各个节点的检索概率相等

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

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

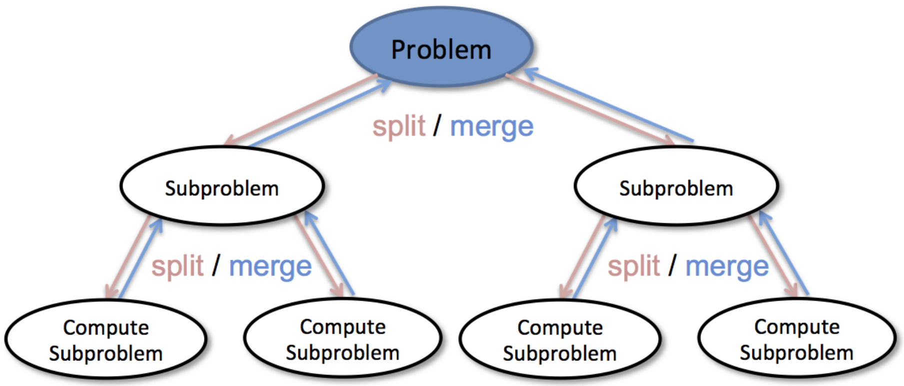
  - 当n很大时，得到：$\mathrm{ASL} \approx \log_2(n+1)-1 = O(\log n)$

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)
```