搜索和排序

1 搜索

1.1 顺序搜索、二分搜索

1.2 散列 (Hash)

1.2.1 Hash table

- It is an abstract data type (ADT) that **maps keys to values**. A hash table uses a **hash function** to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found.

  Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ **an imperfect hash function**, which might cause **hash collisions** where the hash function generates the same index for more than one key.

  *Hashing is an example of a **space-time tradeoff.***

- **Load Factor (载荷因子)**

  $$\text{load factor} \quad \lambda = \frac{n}{m},$$

  where n is the the number of keys, m is the number of buckets. $0.6 < \lambda < 0.75$ is acceptable.

1.2.2 Hash function

- A hash function may be considered to perform three functions:

  - Convert variable-length keys into **fixed length** (usually machine word length or less) values, by folding them by words or other units using a parity-preserving operator like ADD or XOR.
  - Scramble the bits of the key so that the resulting values are **uniformly distributed** over the keyspace.
  - Map the key values into ones **less than or equal to the size** of the table.

  A good hash function satisfies two basic properties: 1) it should be very fast to compute; 2) it should minimize duplication of output values (collisions).

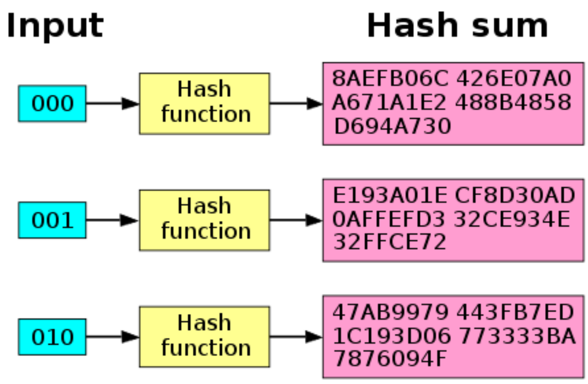
- **Testing and measurement**

  - **Chi-Squared Test (卡方检验)**

$\dfrac{\sum_{j=0}^{m-1}(b_j)(b_j+1)/2}{(n/2m)(n+2m-1)} \in (0.95, 1.05)$ indicates the hash function has an expected **uniform distribution**, where b_j is the number of items in bucket j.

- **Strict Avalanche Criterion (严格雪崩准则)**

  Whenever **a single input bit** is complemented, **each of the output bits** changes with a **50%** probability.



1.2.3 Hash collision

- when two pieces of data in a hash table share the same hash value
- Collision Resolution
  - **Open addressing (开放定址法)**

    Cells in the hash table are assigned one of three states in this method – occupied, empty, or deleted. If a hash collision occurs, the table will **be probed to move the record to an alternate cell that is stated as empty.**

    - linear probing (线性探测), double hashing (双散列), quadratic probing (平方探测)

    采用线性探测策略，搜索成功的平均比较次数为 $\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$，搜索失败的平均比较次数为 $\frac{1}{2}\left[1+\left(\frac{1}{1-\lambda}\right)^2\right]$.

  - **Separate chaining (分离链接法)**

    If two records are being directed to the same cell, both would go into that cell **as a linked list.**

    搜索成功的平均比较次数为 $1+\frac{\lambda}{2}$，搜索失败的平均比较次数为 λ.

1.2.4 Implement ADT Map using Hash table 用哈希表实现映射

- 实现 `key` 为整数，`Hash function` 为取余函数的简单情形
- `HashTable` 类的实现

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

    def put(self, key, data): # 处理冲突时，采用线性探测法
        hashvalue = self.hashfunction(key)

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

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

    def hashfunction(self, key): # 采用简单的取余函数
        return key % self.size

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

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

        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)
                if position == startslot:
                    stop = True
        return data

    # 重载 __getitem__ 和 __setitem__ ，以通过 [] 进行访问
    def __getitem__(self, key):
        return self.get(key)

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

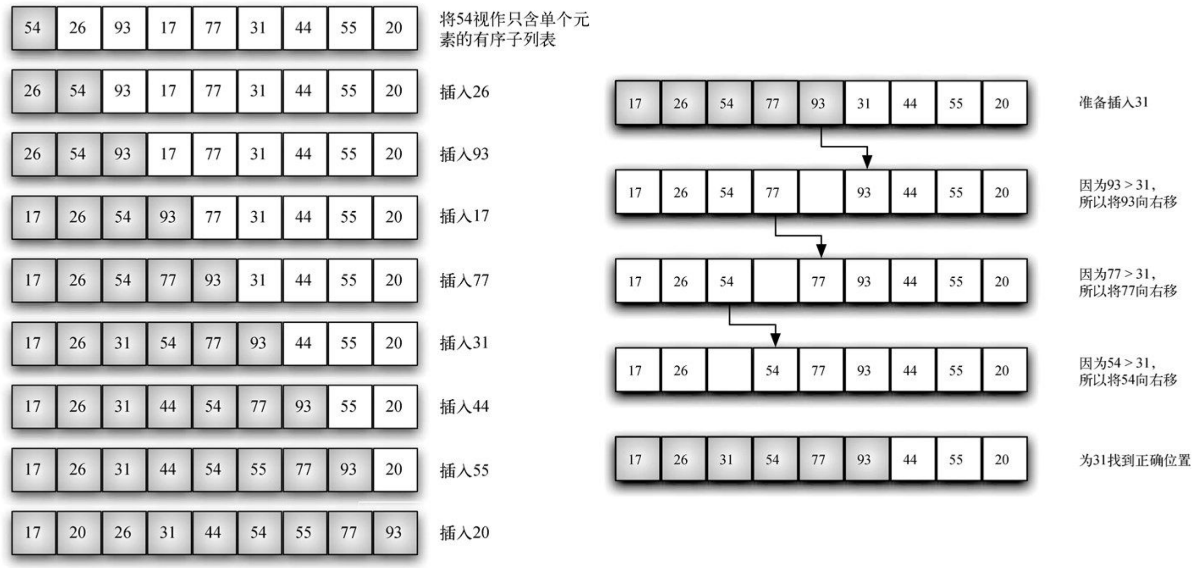
2 排序

2.1 冒泡排序、选择排序

2.2 插入排序、希尔排序 (Shell sort)

2.2.1 插入排序

- 图示



2.2.2 希尔排序

- 希尔排序也称“递减增量排序”，它对插入排序做了改进，将列表分成数个子列表，**并对每一个子列表应用插入排序。如何切分列表**是希尔排序的关键——并不是连续切分，而是使用增量 i 选取所有间隔为 i 的元素组成子列表。

  增量 i 递减，最后一步 $i=1$，即为基本的插入排序（但已不需要多次比较或移动）。

- 先为 $n/2$ 个子列表排序，再对 $n/4$ 个子列表排序，…，采用这种增量的 Python 实现：

```
def shellSort(alist):
    sublistcount = len(alist) // 2
    while sublistcount > 0:
        for startposition in range(sublistcount):
            gapInsertionSort(alist, startposition, sublistcount)
        sublistcount = sublistcount // 2

def gapInsertionSort(alist, start, gap):
    for i in range(start+gap, len(alist), gap):
        currentvalue = alist[i]
        position = i
        while position >= gap and \
                alist[position-gap] > currentvalue:
```

```
        alist[position] = alist[position-gap]
        position = position-gap
    alist[position] = currentvalue
```

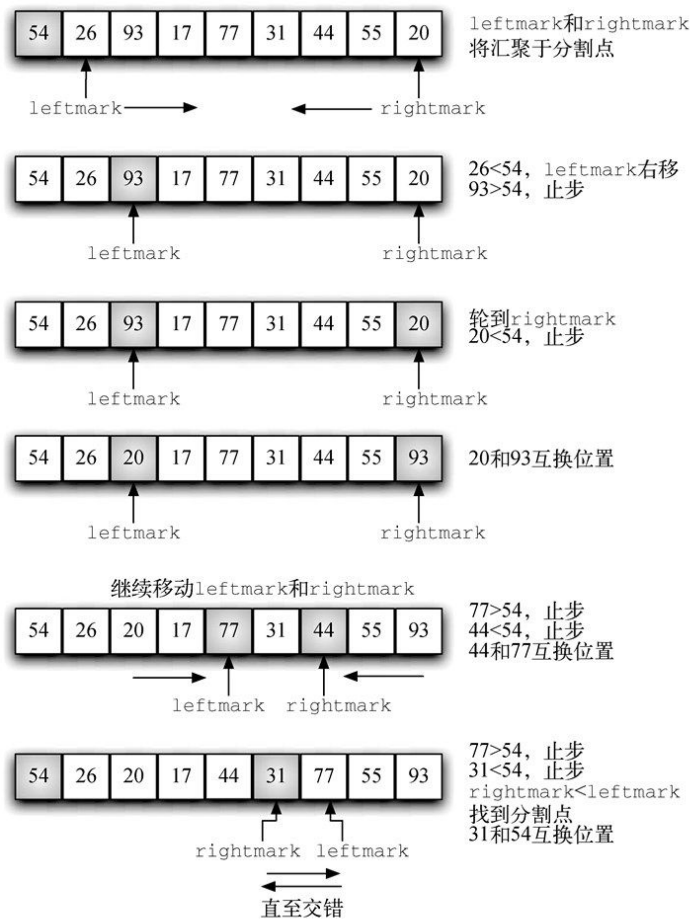
- most proposed gap sequences

OEIS	General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
A000225	$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Hibbard, 1963[9]
A083318	$2^k + 1$, prefixed with 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Papernov & Stasevich, 1965[10]
A003586	Successive numbers of the form $2^p 3^q$ (3-smooth numbers)	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta\left(N \log^2 N\right)$	Pratt, 1971[1]
A003462	$\frac{3^k - 1}{2}$, not greater than $\left\lceil \frac{N}{3} \right\rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Knuth, 1973,[3] based on Pratt, 1971[1]
A036569	$\prod_I a_q$, where $a_0 = 3$ $a_q = \min\left\{n \in \mathbb{N} \colon n \geq \left(\frac{5}{2}\right)^{q+1}, \forall p \colon 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1\right\}$ $I = \left\{0 \leq q < r \mid q \neq \frac{1}{2}\left(r^2 + r\right) - k\right\}$ $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$	$1, 3, 7, 21, 48, 112, \ldots$	$O\left(N^{1+\sqrt{\frac{8 \ln(5/2)}{\ln(N)}}}\right)$	Incerpi & Sedgewick, 1985,[11] Knuth[3]
A036562	$4^k + 3 \cdot 2^{k-1} + 1$, prefixed with 1	$1, 8, 23, 77, 281, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1982[6]
A033622	$\begin{cases} 9\left(2^k - 2^{\frac{k}{2}}\right) + 1 & k \text{ even}, \\ 8 \cdot 2^k - 6 \cdot 2^{(k+1)/2} + 1 & k \text{ odd} \end{cases}$	$1, 5, 19, 41, 109, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1986[12]

2.3 归并排序、快速排序

2.3.1 快速排序

- 图示



- 时间复杂度：$O(n\log n)$，最坏情况 $O(n^2)$
    - 原因：分割点偏向某一端，导致切分不均匀；例如待排序列部分有序时
    - 优化：**选择基准值**，不再始终选择头部元素/尾部元素
        - 随机选取 `rand()`
        - 三数取中法（考虑头元素、中间元素和尾元素）
- 进一步优化：处理重复数组时，时间复杂度仍是 $O(n^2)$
    - 优化方案：
        - 当待排序序列的长度分割到较小后，使用插入排序
        - 在一次分割结束后，可以把与基准值 `key` 相等的元素聚在一起，继续下次分割时，不用再对与 `key` 相等的元素分割
    - 实现

```python
def quickSort(lst, low, high): # low, high 相向移动，按key分割
    first, last = low, high # first, last 固定不动，指向片段头尾
    left, right = low, high # left, right 指向=key的元素的两个边界
    leftLen, rightLen = 0, 0 # 记录两端各有多少=key的元素，实际即为 leftLen = left - first, rightLen = last - right
```

```
    if high - low + 1 < 10: # 短片段进行插排
        insertionSort(lst, low, high)
        return

    key = selectPivotMedianOfThree(lst, low, high) # 三数取中法

    while low < high: # 正常快排
        while high > low and lst[high] >= key:
            if lst[high] == key:
                lst[right], lst[high] = lst[high], lst[right]
                right -= 1
                rightLen += 1
            high -= 1
        lst[low] = lst[high]
        while high > low and lst[low] <= key:
            if lst[low] == key:
                lst[left], lst[low] = lst[low], lst[left]
                left += 1
                leftLen += 1
            low += 1
        lst[high] = lst[low]
    lst[low] = key

    # 将两端=key的元素移到中间key的左右
    i, j = low - 1, first
    while j < left and lst[i] != key:
        lst[i], lst[j] = lst[j], lst[i]
        i -= 1
        j += 1
    i, j = low + 1, last
    while j > right and lst[i] != key:
        lst[i], lst[j] = lst[j], lst[i]
        i += 1
        j -= 1

    # 中间一列=key的片段不再参与排序
    quickSort(lst, first, low - 1 - leftLen)
    quickSort(lst, low + 1 + rightLen, last)

def selectPivotMedianOfThree(lst, low, high):
    mid = (low + high) // 2
    if lst[mid] > lst[high]:
        lst[mid], lst[high] = lst[high], lst[mid]
    if lst[low] > lst[high]:
        lst[low], lst[high] = lst[high], lst[low]
    if lst[mid] > lst[low]:
```

```python
        lst[mid], lst[low] = lst[low], lst[mid]
    return lst[low]

def insertionSort(lst, low, high):
    for i in range(low + 1, high + 1):
        key = lst[i]
        j = i - 1
        while j >= low and lst[j] > key:
            lst[j + 1] = lst[j]
            j -= 1
        lst[j + 1] = key
```

- 优化效果（C++; 数组大小 100 万）

算法	随机数组	升序数组	降序数组	重复数组
固定基准值	133 ms	745125 ms	644360 ms	755422 ms
随机基准值	218 ms	235 ms	187 ms	701813 ms
三数取中	141 ms	63 ms	250 ms	705110 ms
三数取中+插入排序	131 ms	63 ms	250 ms	699516 ms
三数取中+插排+聚集相等元素	110 ms	32 ms	31 ms	10 ms
STL 中的 Sort 函数	125 ms	27 ms	31 ms	8 ms