# 第一部分 常见算法

## 1. 布隆过滤器

布隆过滤器是一个很长的二进制向量和一系列随机映射函数。

布隆过滤器可以检测一个元素是否在一个集合中，不会漏报。

它的优点是空间效率和查询时间都远远超过一般的算法。

它的缺点是有一定的误识别率和删除困难。

* 算法描述：

一个empty bloom filter是一个有m bites的bit array，每一个bit都初始化为0.并且定义有k个不同的hash function，每个都以uniform random distribution将元素hash到m个不同位置中的一个；

添加元素时，用k个hash function将它得到的bloom filter中的k个bit位置1；

查询时，用k个hash function得到k个bit位，若k bits全为1，则我们认为它在集合中，否则不在。

另外，该算法不允许remove元素

## 2. 缓存算法

### 2.1 基本概念

命中：当客户发起一个请求，如果在缓存中，就成为缓存命中

Cache Miss：如果还有缓存空间，没有命中的就会被存储到缓存中；如果缓存满了，而又没命中缓存，那么久会按照缓存算法，用新对象替换旧对象。

存储成本：将数据放到缓存所需要的时间和空间

失效：当存在缓存中的数据需要更新时，缓存中的这个数据就失效了

adoptive to access模式：

### 2.2 缓存算法

Least Frequently Used (LFU):

根据每个缓存对象被使用的频率，将最不常用的缓存对象移除

Least Recently Used (LRU):

将最近最少使用的对象移除，最近使用的对象会被放到缓存的顶部，当缓存达到容量上限时，将底部的对象移除。

Least Recently Used 2 (LRU2)：

将被两次访问过的对象放入缓存池，当缓存池满了，会移除两次最少使用的缓存对象。因为要跟踪对象两次，访问负载就会随着缓存池的增加而增加，所以不能用于大容量的缓存池。

Two Queues (2Q)：

将被访问的数据放到LRU的缓存中，如果这个对象再一次被访问，就将它转移到更大的LRU缓存中。移除对象是为了保持第一个缓存池是第二个缓存池的1/3，当缓存访问负载是固定的时候，把LRU换成LRU2，比增加缓存容量更好，是adoptive to access模式

Adaptive Replacement Cache (ARC):

介于LRU和LFU之间。由两个LRU组成，第一个L1，包含的条目是最近值被使用过一次的，而L2，包含的是最近被使用过两次的数据。L1放的是新对象，L2放的是常用对象。

Most Recently Used (MRU):

移除最近最多被使用的对象。每当一次缓存记录的使用，就会被放到栈顶，当栈满了，将栈顶的对象移除。

First In First Out (FIFO):

先进先出，低负载的算法，通过队列跟踪所有的缓存对象，最近最常用的对象放在后边，当缓存容量满的时，会移除前边缓存的更早的对象。很快，但是不适用

# 第二部分 算法基础

### 1. 插入排序

对于插入排序，将其伪代码命名为INSERTION-SORT，输入是一个数组A[1...n],该算法是原址排序：在排序过程中，最多只有常数个数字存储在数组外。

**时间复杂度：Θ(n2)**

|  |
| --- |
| INSERTION**-**SORT**(**A**)**  1 **for** j **=** 2 to A**.**length  2 key **=** A**[**j**]**  3 // Insert A[j] into the sorted sequence A[1..j-1].  4 i **=** j **-** 1  5 **while** i**>**0 and A**[**i**]>**key  6 A**[**i**+**1**]** **=** A**[**i**]**  7 i **=** i **-** 1  8 A**[**i**+**1**]** **=** key |

循环不变式主要用来帮助我们理解算法的正确性：

**初始化**：循环的第一次迭代之前，它为真

**保持**：如果循环的某次迭代之前它为真，那么下次迭代之前它仍为真。

**终止**：当循环终止时，不变式为我们提供了有用的性质，有助于证明算法的正确性

### 2. 分治法（递归）

分治模式在每层递归时有三个步骤：

**分解**原问题为若干子问题，这些子问题是原问题的规模较小的实例

**解决**这些子问题，递归地求解各子问题。若子问题较小，则直接求解

**合并**这些子问题的解成原问题的解

### 3. 归并排序

归并排序完全遵循分治模式，算法的时间复杂度是**Θ(nlgn)**：

分解：分解待排序的n个元素的序列成各含有n/2个元素的子序列

解决：使用归并排序递归排序两个子序列

合并：合并两个已排序的子序列以产生已排序的答案

|  |
| --- |
| MERGE**(**A**,** p**,** q**,** r**)**  1 m **=** q **-** p **+** 1  2 n **=** r **-** q  3 let L**[**1..m**+**1**]** and R**[**1..n**+**1**]** be **new** arrays  4 **for** i**=**1 to m  5 L**[**i**]** **=** A**[**p**+**i**-**1**]**  6 **for** j**=**1 to n  7 R**[**j**]** **=** A**[**q**+**j**]**  8 L**[**m**+**1**]** **=** ∞ // 哨兵，包含一个特殊的值，简化代码  9 R**[**n**+**1**]** **=** ∞  10 i **=** 1  11 j **=** 1  12 **for** k**=**p to r  13 **if** L**[**i**]**≤R**[**j**]**  14 A**[**k**]** **=** L**[**i**]**  15 i **=** i **+** 1  16 **else** A**[**k**]** **=** R**[**j**]**  17 j **=** j **+** 1  MERGE**-**SORT**(**A**,** p**,** r**)**  1 **if** p**<**r  2 q **=** **(**p**+**r**)%**2 **\*** 2  3 MERGE**-**SORT**(**A**,** p**,** q**)**  4 MERGE**-**SORT**(**A**,** q**+**1**,** r**)**  5 MERGE**(**A**,** p**,** q**,** r**)** |

## 3. 分治策略

递归技术的一些细节：边界条件时我们常忽略的细节

重要的思想：**问题转换**

### 1. 最大子数组问题（数组中包含负数才有意义）

股票问题->最大子数组问题->利用之前计算出的子数据的和来计算当前子数组降低事件复杂度。

问题描述：给出股票每日的价格，找出获取最大收益的方法

问题变换：将每日的价格，转换为每日价格的变化，第i天的价格变化定义为第i天和第i-1天的价格差。那么问题就转化为寻找数组中最大的非空连续子数组。

解决：A[low..high]的任何连续子数组A[i..j]所处的位置必然是以下三种情况：完全位于子数组A[low..mid]；完全位于子数组A[mid+1..hig]中；跨越了中点。

|  |
| --- |
| **寻找从中点开始最大的子数组** |
| FIND**-**MAX**-**CROSSING**-**SUBARRAY**(**A**,** low**,** mid**,** high**)**  1 left**-**sum **=** **-**∞  2 sum **=** 0  3 **for** i**=**mid downto low  4 sum **=** sum **+** A**[**i**]**  5 **if** sum**>**left**-**sum  6 left**-**sum **=** sum  7 max**-**left **=** i  8 right**-**sum **=** **-**∞  9 sum **=** 0  10 **for** j **=**mid**+**1 to hight  11 sum **=** sum **+** A**[**j**]**  12 **if** sum**>**right**-**sum  13 right**-**sum **=** sum  14 max**-**right **=** j  15 **return** **(**max**-**left**,** max**-**right**,** left**-**sum **+** right**-**sum**)** |

时间复杂度为**Θ(nlgn)**的最大子数组算法

|  |
| --- |
| **最大子数组算法** |
| FIND**-**MAXIMUM**-**SUBARRAY**(**A**,** low**,** high**)**  1 **if** high **==** low  2 **return** **(**low**,** high**,** A**[**low**])** //base case: 只有一个元素  3 **else**  4 mid**=(**low**+**high**)/**2 // 小于或等于(low+high)/2的最大整数  5 **(**left**-**low**,** left**-**high**,** left**-**sum**)** **=**  6 FIND**-**MAXIMUM**-**SUBARRAY**(**A**,** low**,** mid**)**  7 **(**right**-**low**,** right**-**high**,** right**-**sum**)** **=**  8 FIND**-**MAXIMUM**-**SUBARRAY**(**A**,** mid**+**1**,** high**)**  9 **(**cross**-**low**,** cross**-**high**,** cross**-**sum**)** **=**  10 FIND**-**MAX**-**CROSSING**-**SUBARRAY**(**A**,** low**,** mid**,** high**)**  11 **if** left**-**sum≥right**-**sum and left**-**sum≥cross**-**sum  12 **return** **(**left**-**low**,** left**-**high**,** left**-**sum**)**  13 **else** **if** right**-**sum≥left**-**sum and right**-**sum≥cross**-**sum  14 **return** **(**right**-**low**,** right**-**high**,** right**-**sum**)**  15 **else**  16 **return** **(**cross**-**low**,** cross**-**high**,** cross**-**sum**)** |

# 第三部分：排序和顺序统计量

**数据结构**：实际中，待排数据都是带有卫星数据的记录，我们通常是根据关键字重排记录的指针数组。

**原址排序**：输入数组中仅有常数个元素需要在排序过程中存储在数组之外。

各排序运行时间分析：

|  |  |  |  |
| --- | --- | --- | --- |
| 算法 | 最坏情况运行时间 | 平均情况/期望运行时间 | 是否为原址排序 |
| 插入排序 | **Θ(n2)** | **Θ(n2)** | 原址排序 |
| 归并排序 | **Θ(nlgn)** | **Θ(nlgn)** | 非原址排序 |
| 堆排序 | **O(nlgn)** | - | 原址排序 |
| 快速排序 | **Θ(n2)** | **Θ(nlgn) （期望）** | 原址排序 |
| 计数排序 | **Θ(k+n)** | **Θ(k+n)** | 非原址排序 |
| 基数排序 | **Θ(d(k+n))** | **Θ(d(k+n))** | 非原址排序 |
| 桶排序 | **Θ(n2)** | **Θ(n) （平均情况）** | 非原址排序 |

## 1. 堆排序

（二叉）堆是一个数据，它可以被看成一个近似的完全二叉树，不仅可以用来排序，还可以构造一种有效的优先队列.

**最大堆**：最大堆性质是指除了根节点以外的所有节点i都要满足：

因此堆中最大的元素存放在根节点。

在堆排序算法中，我们使用最大堆，最小堆常用于构造优先队列。

|  |
| --- |
| **MAX-HEAPIFY用于维护最大堆，将A[i]放到正确的位置，（时间复杂度O(h)）** |
| MAX**-**HEAPIFY**(**A**,** i**)**  1 l **=** LEFT**(**i**)**  2 r **=** RIGHT**(**i**)**  3 **if** l≤A**.**heap**-**size and A**[**l**]>**A**[**i**]**  4 largest **=** l  5 **else** largest **=** i  6 **if** r≤A**.**heap**-**size and A**[**r**]>**A**[**largest**]**  7 largest **=** r  8 **if** largest≠i // 如果A[i]最大，程序结束，否则递归调用函数  9 exchange A**[**i**]** with A**[**largest**]**  10 MAX**-**HEAPIFY**(**A**,** largest**)** |

使用自底向上的方法利用MAX-HEAPIFY把一个大小为n=A.length的数组转换成最大堆。最大堆的**下标是从1开始**的，可以对其left和right方法进行处理转换。 left=2i+1,right=2i+2

|  |
| --- |
| **构造最大堆** |
| BUILD**-**MAX**-**HEAP**(**A**)**  1 A**.**heap**-**size **=** A**.**length  2 **for** i**=**A**.**length**/**2 downto 1 // 5/2=2  3 MAX**-**HEAPIFY**(**A**,** i**)** |

堆排序算法利用BUILD**-**MAX**-**HEAP构造最大堆，因为数组中最大元素总在根节点A[1]中，通过把它与A[n]进行互换，可以将元素放到正确的位置

|  |
| --- |
| **堆排序算法** |
| HEAPSORT**(**A**)**  1 BUILD**-**MAX**-**HEAP**(**A**)**  2 **for** i**=**A**.**length downto 2  3 exchange A**[**1**]** with A**[**i**]**  4 A**.**heap**-**size **=** A**.**heap**-**size**-**1  5 MAX**-**HEAPIFY**(**A**,**1**)** |

## 2.优先队列

定义：

优先队列是一种用来维护由一组元素构成的集合S的数据结构，其中的每一个元素都有一个相关值，称为关键字。

一个最大优先队列支持：插入元素INSERT(S,x)，获取最大键元素MAXIMUM(S)，去掉最大键字元素EXTRACT-MAX(S)，将元素x的关键字值增加到k INCRETSE-KEY(S,x,k)

最小优先队列支持的操作有：INSERT，MINIMUM, EXTRACT-MAX,DECREASE-KEY.

基于堆的最大优先队列实现

|  |
| --- |
| **获取最大元素** |
| HEAP**-**MAXIMUM**(**A**)**  1 **return** A**[**1**]** |
| **去掉最大元素 O(lgn)** |
| HEAP**-**EXTRACT**-**MAX**(**A**)**  1 **if** A**.**heap**-**size **<** 1  2 **throw** "heap underflow"  3 max **=** A**[**1**]**  4 A**[**1**]** **=** A**[**A**.**heap**-**size**]**  5 A**.**heap**-**size **=** A**.**heap**-**size **-** 1  6 MAX**-**HEAPIFY**(**A**,** 1**)**  7 **return** max |
| **将x的关键字值增加到k O(lgn)** |
| HEAP**-**INCREASE**-**KEY**(**A**,** i**,** key**)**  1 **if** key **<** A**[**i**]**  2 **throw** "new key is smaller than current key"  3 A**[**i**]** **=** key  4 **while** i**>**1 and A**[**PARENT**(**i**)]<**A**[**i**]**  5 exchange A**[**i**]** with A**[**PARENT**(**i**)]**  6 i **=** PARENT**(**i**)** |
| **将x插入到队列中 O(lgn)** |
| MAX**-**HEAP**-**INSERT**(**A**,** key**)**  1 A**.**heap**-**size **=** A**.**heap**-**size **+** 1  2 A**[**A**.**heap**-**size**]** **=** **-**∞  3 HEAP**-**INCREASE**-**KEY**(**A**,** A**.**heap**-**size**,** key**)** |

## 3. 快速排序

与归并排序一样，快速排序也使用了分治思想：

* **分解**：数组A[p..r]被划分为两个子数组A[p..q-1]和A[q+1..r]，使得A[p..q-1]中每个元素都小于A[q]，而A[q+1..r]中的每个元素都大于A[q]
* **解决：**通过递归调用快速排序，对子数组A[p..q-1]和A[q+1..r]进行排序
* **合并：**子数组为原址排序，不需要合并

|  |
| --- |
| **分解** |
| QUICKSORT**(**A**,** p**,** r**)**  1 **if** p**<**r  2 q **=** PARTITION**(**A**,** p**,** r**)**  3 QUICKSORT**(**A**,** p**,** q**-**1**)**  4 QUICKSORT**(**A**,** q**+**1**,** r**)** |
| **随机化版本** |
| RANDOMZED**-**PARTITION**(**A**,** p**,** r**)**  1 i **=** RANDOM**(**p**,** r**)**  2 exchange A**[**r**]** with A**[**i**]**  3 **return** PARTITION**(**A**,** p**,** r**)** |
| **合并** |
| PARTITION**(**A**,** p**,** r**)**  1 x **=** A**[**r**]**  2 i **=** p **-** 1  3 **for** j**=**p to r**-**1  4 **if** A**[**j**]** ≤x  5 i **=** i **+** 1  6 exchange A**[**i**]** with A**[**j**]**  7 exchange A**[**i**+**1**]** with A**[**r**]**  8 **return** i**+**1 |

## 4. 线性时间排序

使用比较来确定排序顺序的，称之为比较排序。任何比较排序在最坏情况下都要经过Ω(nlgn)次比较。

### 4.1 计数排序

计数排序是假设n个输入元素中的每一个都是在0到k区间内的一个整数，其中k为整数

基本思想：对每一个输入元素x，确定小于x的元素个数，利用这一信息，就可以直接把x放到它在输出数组中的位置了

|  |
| --- |
| **计数排序** |
| // B[1..n] 存放的是排好序的数组  COUNTIG**-**SORT**(**A**,** B**,** k**)**  1 let C**[**0..k**]** be a **new** array  2 **for** i**=**0 to k  3 C**[**i**]=**0  4 **for** j**=**1 to A**.**length  5 C**[**A**[**j**]]** **=** C**[**A**[**j**]]** **+** 1  6 // C[i] now contains the number of elements equal to i  7 **for** i**=**1 to k  8 C**[**i**]** **=** C**[**i**]** **+** C**[**i**-**1**]**  9 // C[i] now contains the number of elements less than or equal to i  10 **for** j**=**A**.**length downto 1  11 B**[**C**[**A**[**j**]]]** **=** A**[**j**]**  12 C**[**A**[**j**]]** **=** C**[**A**[**j**]]** **-** 1 |

### 4.2 基数排序

基数排序是先按最低有效位进行排序，然后递归地进行排序

|  |
| --- |
| **Θ(d(n+k))** |
| RADIX**-**SORT**(**A**,** d**)**  1 **for** i**=**1 to d  2 use a stable to sort array A on digit i |

因为计数排序算法不是原址的，而且计数排序的常数项因子可能比较大，所以在主存容量比较宝贵时，我们更倾向于快速排序这样的原址排序

### 4.3 桶排序

桶排序假设输入的数据服从均匀分布，平均情况下它的时间代价为O(n)。桶排序假设输入是由一个随机过程产生，该过程将元素均匀、独立地分布在[0,1)的区间上

|  |
| --- |
| **期望运行时间为Θ(n)** |
| BUCKET**-**SORT**(**A**)**  1 n **=** A**.**length  2 let B**[**0..n**-**1**]** be a **new** array  3 **for** i**=**0 to n**-**1  4 make B**[**i**]** an empty list  5 **for** i**=**1 to n  6 insert A**[**i**]** into list B**[**nA**[**i**]]**  7 **for** i**=**0 to n**-**1  8 sort list B**[**i**]** with insertion sort  9 concatenate the list B**[**0**],** B**[**1**],..,**B**[**n**-**1**]** together in order |

## 5. 中位数与顺序统计量

在一个由n个元素组成的集合中，第i个顺序统计量是该集合中第i小的元素。中位数是它所设计和的“中点元素”。当n为奇数时，i=(n+1)/2; 当n为偶数时，i=n/2和i=n/2+1.

输入：包含n个（互异的）数的集合A和一个整数i， 1≤i≤n

输出：A中恰好有i-1个元素小于该元素。

### 5.1 找出最大值和最小值

对输入元素进行成对的比较，然后把较小的元素与当前最小值比较，较大的元素同当前最大值比较，比较次数只需要原来的3/4。

1. 判断n是奇数还是偶数；

2. 当n为奇数时，将最大值和最小值设为第一个元素，然后成对地处理余下元素；

3. 当n为偶数时，对前两个元素进行比较，以决定最小值和最大值的初值。

### 5.2 期望时间为线性时间的选择算法

|  |
| --- |
| **基于选择排序，期望时间为线性** |
| RANDOMIZED**-**SELECT**(**A**,** p**,** r**,** i**)**  1 **if** p **==** r  2 **return** A**[**p**]**  3 q **=** RANDOMIZED**-**PARTITION**(**A**,** p**,** r**)**  4 k **=** q **-** p **+** 1  5 **if** i **==** k  6 **return** A**[**q**]**  7 **else** **if** i **<** k  8 **return** RANDOMIZED**-**SELECT**(**A**,** p**,** q**-**1**,** i**)**  9 **else**  10 **return** RANDOMIZED**-**SELECT**(**A**,** q**+**1**,** r**,** i**-**k**)** |

# 第四部分 数据结构

实现动态集合的关键取决于必须支持的一些集合操作（暴露的接口）。（数据结构决定算法）

下面列出一些具体操作，任何应用通常只需要实现这些操作的若干个就可以实现：

* SEARCH(S, k): 在S集合中查找关键字为key的元素，如果没有返回NIL
* INSERT(S, x): 修改操作，将x指向的元素插入到集合S中
* DELETE(s, x): 修改操作，从S中删除x指向的元素
* MINIMUM(S): 查询操作，在全序集S上返回一个指向S中具有最小关键字的元素
* MAXIMUX(S): 查询操作，在全序集S上返回一个指向S中具有最大关键字的元素
* SUCCESSOR(S, x): 查询操作，返回S中比x大的下一个元素，若x最大，返回NIL
* PREDECESSOR(S, x): 查询操作，返回S中比x小的下一个元素，若x最小，返回NIL

## 1. 基本数据结构

### 1.1 栈和队列

栈和队列都是动态集合，而且在其上进行DELETE操作所移除的元素都是预先设定的。栈实现的是一种后进先出（LIFO）策略。队列中实现的是一种先进先出（FIFO）策略。

**栈描述**：用一个数组S[1..n]来实现一个最多容纳n个元素的栈，该数组有一个属性S.top指向最新插入的元素。栈中包含的元素为S[1..S.top].

|  |
| --- |
| STACK**-**EMPTY**(**S**)** // 是否为空  1 **if** S**.**top **==** 0  2 **return** **true**  3 **else**  4 **return** **false** |
| PUSH**(**S**,** x**)** // 向栈中push元素  1 S**.**top **=** S**.**top **+** 1  2 S**[**S**.**top**]** **=** x |
| POP**(**S**)** // 从栈中取出元素  1 **if** STACK**-**EMPTY**(**S**)**  2 **throw** 'underflow'  3 **else**  4 S**.**top **=** S**.**top **-** 1  5 **return** S**[**S**.**top **+** 1**]** |

**队列描述**：用数组Q[1..n]来实现一个最多容纳n-1个元素的队列，Q.head指向队头元素，Q.tail指向下一个元素将要插入的位置。队列中的元素存放在Q.head, Q.head\_1,...Q.tail-1中。 Q.head=Q.tail时，队列为空；Q.head=Q.tail+1时，对列是满的

|  |
| --- |
| ENQUEUE**(**Q**,** x**)**  1 Q**[**Q**.**tail**]** **=** x  2 **if** Q**.**tail **==** Q**.**head **+** 1  3 **throw** 'full queue'  4 **if** Q**.**tail **==** Q**.**length  5 Q**.**tail **=** 1  6 **else**  7 Q**.**tail **=** Q**.**tail **+** 1 |
| DEQUEUE**(**Q**)**  1 **if** Q**.**tail **==** Q**.**head  2 **throw** 'empty queue'  3 x **=** Q**[**Q**.**head**]**  4 **if** Q**.**head **==** Q**.**length  5 Q**.**head **=** 1  6 **else**  7 Q**.**head **=** Q**.**head **+** 1  8 **return** x |