写在前面的话：

在解决和分析算法问题时，切记对一种想法钻牛角尖。首先要分析问题所包含的特性，这些特性包含问题本身和其具有的数学特征。

# 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 |

### 1.2 链表

链表中的各对象按线性顺序排列，数组的线性顺序是由数组下标决定的，链表的顺序是由各个对象里的指针决定的。

双向链表的每个元素都是一个对象，每个对象有一个关键字key和两个指针:next和prev，还可以包含其他卫星数据。

链表分类：单向链表，双向链表，有序链表，循环链表

|  |
| --- |
| **链表搜索（未排序，单向链表）** |
| LIST**-**SEARCH**(**L**,** k**)**  1 x **=** L**.**head  2 **while** x ≠ NIL and x**.**key ≠ key  3 x **=** x**.**next  4 **return** x |
| **链表插入（未排序，单向链表）** |
| LIST**-**INSERT**(**L**,** x**)**  1 x**.**next **=** L**.**head  2 **if** L**.**head ≠ NIL  3 L**.**head**.**prev **=** x  4 L**.**head **=** x  5 x**.**prev **=** NIL |
| **链表的删除（未排序，单向链表）** |
| LIST**-**DELETE**(**L**,** x**)**  1 **if** x**.**prev ≠ NIL  2 x**.**prev**.**next **=** x**.**next  3 **else**  4 L**.**head **=** x**.**next  5 **if** x**.**next ≠ NIL  6 x**.**next**.**prev **=** x**.**prev |

哨兵是一个哑对象，作用是简化边界条件的处理。在链表L中设置一个对象L.nil，该对象代表NIL，但也具有和其他对象相同的属性，对于代码中出现的每一处对NIL的引用，都代以对哨兵L.nil的引用，L.nil.next指向表头，L.nil.prev指向表位。

### 1.3有根树的表示

二叉树：利用属性p、left和right存放指向父节点、左孩子和右孩子的指针；

分支无限树：使用左孩子右兄弟表示法表示：每个节点都包含一个父节点指针p，和两个指针，x.left-child指向节点x最左边的孩子节点；x.right-sibling指向x右侧相邻的兄弟节点。

## 2. 散列表

散列表使用一个长度与实际存储的关键字数目成比例的数组来存储。适用于实际存储的关键字数目比全部的可能关键字总数要小，通过链接方法解决冲突。

在直接寻址方式下，具有关键字k的元素被存放在槽k中；在散列方式下，该元素存放在槽h(k)中，即利用散列函数h，由关键字k计算出槽的位置。

当两个关键字被映射到一个槽上时，称之为冲突。一方面通过设计散列函数来减少冲突，另一方面要有解决冲突的办法：链接法，开放寻址法。

链接法：把散列到同一槽中的所有元素都放在一个链表中。

给定一个能存放n个元素的、具有m个槽的散列表T，定义T的装载因子为 n/m，即一个链的平均存储元素数。

在简单均匀散列的情况下，查找、删除、添加都需要常数时间内完成。

### 2.1 散列函数

多数散列函数都假定关键字的全域为自然数集N={0,1,2,...}。因此，如果所给关键字不是自然数，就需要将其转换成自然数。

### 2.2 除法散列

通过取k除以m的余数，将关键字k映射到m个槽中的某一个上，即散列函数为：

h(k) = k mod m

一个不太接近2的整数幂的素数，常常是m的较好的选择

### 2.3 乘法散列

构造乘法散列有两步：1. 用关键字k乘以常数A(0<A<1)，并提取kA的小数部分；2. 用m乘以该值，再向下取整，散列函数为：

h(k) = [m(kA mod 1)]

m一般选择为2的整数次幂

### 2.4全域散列

全域散列是从一组有限的散列函数中，随机地选取一个作为散列函数。全域函数的选取见P150

### 2.5 开放寻址法

在开放寻址法中，所有的元素都存放在散列表里，当查找某个元素时，要系统地检查所有的表项，直到找出所需元素，或最终查明该元素不在表中。在该方法中，散列表可能会被填满，因此装载因子不能超过1.

开放寻址法的好处在于它不用指针，而是计算出要存取的槽序列。不用存储指针而节省空间，潜在地减少了冲突，提高了检索速度。

### 2.6 线性探查

给定一个普通散列函数h’:U->{0,1,...,m-1},线性探查方法采用的散列函数为

h(k,i) = (h’(k) + i)mod m, i=0,1,...,m-1

线性探查会从探查位置开始，探查这个序列，知道找到元素，或nil

随着连续被占用的槽不断增加，平均查找时间也会越来越大（一次集群）

### 2.7 二次探查

二次探查采用如下散列函数：

h(k,i) = (h’(k) +c1i +c2i2)mod m, i=0,1,...m-1

### 2.8 双重散列

双重散列是用于开放寻址最好的方法之一，它产生的排列具有随机选择排列的许多特性。散列函数为：

h(k,i)=(h1(k) +ih2(k)mod m

为了能查找整个散列表，h2(k)必须要与表的大小m互素

h1(k) = k mod m, h2(k) = 1 + (k mod m’). 其中m’略小于m

### 2.9 完全散列

采用两级散列方法来设计完全散列，在每级上都使用全域散列。

第一级：利用从某一全域散列函数簇中选出的散列函数h，将n个关键字散列到m个槽中；

第二级：采用一个较小的二次散列表，及相关散列函数，利用精心选择的函数，可以确保在第二级上不出现冲突。

## 3. 二叉搜索树

二叉搜索树上的基本操作所花费的时间与这棵树的高度成正比，可以使用搜索树作为一个字典又可以作为一个优先队列。

二叉搜索树关键字的性质：设x是二叉搜索树中的一个结点。如果y是x左子树中的一个结点，那么y.key<=x.key;如果y是x的右子树中的一个结点，那么y.key>=x.key

中序遍历：输出子树根的关键字位于其左子树的关键字和右子树关键字之间

先序遍历：输出的根的关键字在其左右子树的关键字之前

后序遍历：输出的根的关键字在其左右子树的关键字之后

|  |
| --- |
| **中序遍历 O(n)** |
| INORDER**-**TREE**-**WALK**(**x**)**  1 **if** x ≠ NIL  2 INORDER**-**TREE**-**WALK**(**x**.**left**)**  3 print x**.**key  4 INORDER**-**TREE**-**WALK**(**x**.**right**)** |

### 3.1 查询二叉搜索树

所有的查询操作都可以在O(h)时间内执行完毕

|  |
| --- |
| **查找（输入指向树根的指针和关键字k）** |
| TREE**-**SEARCH**(**x**,** k**)** // 递归方式  1 **if** x **==** NIL or k **==** x**.**key  2 **return** x  3 **if** k **<** x**.**key  4 **return** TREE**-**SEARCH**(**x**.**left**,** k**)**  5 **else**  6 **return** TREE**-**SEARCH**(**x**.**right**,** k**)** |
| TREE**-**SEARCH**(**x**,** k**)** // 迭代方式  1 **while** x ≠ NIL or k ≠ x**.**key  2 **if** k **<** x**.**key  3 x **=** x**.**left  4 **else**  5 x **=** x**.**right  6 **return** x |
| **查找最小值** |
| TREE**-**MINIMUM**(**x**)**  1 **while** x**.**left ≠ NIL  2 x **=** x**.**left  3 **return** x |
| **查找最大值** |
| TREE\_MAXIMUM**(**x**)**  1 **while** x**.**right ≠ NIL  2 x **=** x**.**right  3 **return** x |
| **查找后继节点** |
| TREE**-**SUCCESSOR**(**x**)**  1 **if** x**.**right ≠ NIL  2 **return** TREE**-**MINIMUM**(**x**)**  3 y **=** x**.**p  4 **while** y ≠ NIL and x **==** y**.**right  5 x **=** y  6 y **=** y**.**p  7 **return** y |
| **插入节点：插入为叶子节点** |
| TREE**-**INSERT**(**T**,** z**)**  1 y **=** NIL  2 x **=** T**.**root  3 **while** x ≠ NIL  4 y **=** x  5 **if** z**.**key **<** x**.**key  6 x **=** x**.**left  7 **else** x **=** x**.**right  8 z**.**p **=** y  9 **if** y **==** NIL  10 T**.**root **=** z  11 elseif z**.**key **<** y**.**key  12 y**.**left **=** z  13 **else** y**.**right **=** z |

### 3.2 从二叉搜索树中删除节点

从一个二叉搜索树中删除一个节点z的整个策略分为三种情况：

* 如果z没有孩子节点，那么只是简单地将它删除，并修改它的父节点，用NIL作为孩子来替换z
* 如果z只有一个孩子，那么将孩子提升到树z的位置，并修改z的父节点，用z的孩子替换z
* 如果z有两个孩子，那么找到z的后继y（一定在z的右子树中），并让y占据z的位置。z的原来右子树部分称为y的新右子树，并且z的左子树称为y的新左子树

|  |
| --- |
| **二叉搜索树节点的删除** |
| INORDER**-**TREE**-**WALK**(**x**)**  1 **if** x ≠ NIL  2 INORDER**-**TREE**-**WALK**(**x**.**left**)**  3 print x**.**key  4 INORDER**-**TREE**-**WALK**(**x**.**right**)**  INORDER**-**TREE**-**WALK**(**x**)**  1 **if** x ≠ NIL  2 INORDER**-**TREE**-**WALK**(**x**.**left**)**  3 print x**.**key  4 INORDER**-**TREE**-**WALK**(**x**.**right**)** |

## 4. 红黑树（一种平衡搜索树）

红黑树在每个节点上增加了一个存储位来表示节点的颜色（RED、BLACK），通过对任何一条从根到叶子的简单路径上各个节点的颜色进行约束，红黑树确保没有一条路径会比其他路径长出2倍，因而近乎平衡。

树中的节点有五个属性：color、key、left、right、parent。为了便于处理红黑树代码中的边界条件，使用一个哨兵来代表NIL，红黑树满足如下性质：

* 每个节点或是红色的，或是黑色的
* 根节点是黑色的
* 每个叶节点(NIL)是黑色的
* 如果一个节点是红色的，则它的两个子节点都是黑色的
* 对每个节点，从该节点到掐所有后代节点的单路径上，均包含相同数目的黑色节点

黑高(bh(x)): 从某个节点x出发（不含该节点）到达一个叶节点的任一条简单路径上的黑色节点的个数。从该节点出发的所有下降到其叶节点的简单路径的黑节点个数相同。

一棵有n个内部节点的红黑树的高度至多为2lg(n+1)。

代码等见书或者java代码

## 5. 数据结构的扩张

扩张一个数据结构可以分为4个步骤：

1. 选择一种基础的数据结构

2. 确定基础数据结构中药维护的附加信息

3. 检验基础数据结构上的基本修改操作能否维护附加信息

4. 设计一些新的操作

# 第五部分 高级设计和分析技术

这一部分介绍了设计和分析高效算法的三种重要技术：动态规划、贪心算法和摊还分析。

动态规划通常用来解决最优化问题，在这类问题中，我们通过做出一组选择来达到最优解。其关键技术是对每个子问题保存其解，当其重复出现时既可避免重复求解。

贪心算法也通常用于最优化问题，其思想就是每步选择都是局部最优解。

摊还分析执行一组相似的操作组成的序列，通过分析序列整体的实际代价的界

## 1. 动态规划

分治方法将问题划分为互不相交的子问题，递归地求解子问题，再将它们的解组合起来，求出原问题的解。

动态规划应用于子问题重叠的情况，即不同的子问题具有公共的子子问题。这种情况下，分治算法会反复求解公共子问题，而动态规划通过仔细安排求解顺序，对每个子问题只求解一次，并将结果保存下来，如果随后再次需要此子问题的解，只需要找保存的结果。因此动态规划通过额外的内存空间来节省计算时间。

通常按如下4个步骤来设计一个动态规划算法：

1. 刻画一个最优解的结构特征

2. 递归地定义最优解的值，找寻递归算法（如何将一个问题归纳成数学表达式）

3. 计算最优解的值，通常采用自底向上的方法

4. 利用计算出的信息构造一个最优解。

### 1.1 动态规划原理

适合应用动态规划方法求解的最优化问题应该具备的两个要素：最优子结构和子问题重叠。

如果一个问题的最优解包含其子问题的最优解，我们就称此问题具有最优子结构性质。

在发掘最优子结构性质的过程中，实际上遵循了如下通用模式：

* 证明问题最优解的第一个组成部分是做出一个选择，做出的这个选择会产生一个或者多个待解的子问题
* 对于一个给定的问题，在其可能的第一步选择中，假定已经知道哪种选择才会得到最优解。
* 给定可获得最优解的选择后，你确定这次选择会产生哪些子问题，以及如何最好地刻画子问题空间（保持子问题空间尽可能简单）。

可以用子问题的总数和每个子问题需要考察多少种选择这两个因素的乘积来粗略分析动态规划算法的运行时间。对于钢条切割问题，共有Θ(n)个子问题，每个子问题最多需要考察n种选择，因此运行时间为O(n2)

子问题之间必须是独立的，不相互依赖的。

重叠子问题是问题的递归算法会反复地求解相同的子问题，而不是一直生成新的子问题。

凡是一个问题的自然递归算法的递归调用树中反复出现相同的子问题，而不同子问题的总数很少时，动态规划方法都能提高效率。

将一个问题归纳成数学表达式，最重要

### 1.2 最优二叉搜索树

其形式化定义为：给定一个n个不同关键字的已排序的序列K=<k1, k2, ..., kn>,每个关键字都有一个概率pi 表示其搜索频率，构造一棵二叉树使得所有搜索操作访问的节点总数最少。用伪节点做叶节点，表示搜索的(n+1)个关键字不在树中，概率用q表示

下面伪代码接受概率列表p[1...n]和q[0...n]以及规模n

|  |
| --- |
| OPTIMAL**-**BST**(**p**,** q**,** n**)**  1 let e**[**1...n**+**1**,**0...n**],** w**[**1...n**+**1**,**0...n**],** root**[**1...n**,**1...n**]** be **new** tables  2 **for** i**=**1 to n**+**1  3 e**[**i**,**i**-**1**]** **=** q**[**i**-**1**]**  4 w**[**i**,**i**-**1**]** **=** q**[**i**-**1**]**  5 **for** l**=**1 to n  6 **for** i**=**1 to n**-**l**+**1  7 j **=** i**+**l**-**1  8 e**[**i**,**j**]=**∞  9 w**[**i**,**j**]** **=** w**[**i**,**j**-**1**]** **+** p**[**j**]** **+** q**[**j**]**  10 **for** r**=**i to j**-**1  11 t **=** e**[**i**,**r**-**1**]** **+** e**[**r**+**1**,**j**]** **+** w**[**i**,**j**]**  12 **if** t**<**e**[**i**,**j**]**  13 e**[**i**,**j**]** **=** t  14 root**[**i**,**j**]=**r  15 **return** e and root |

## 2. 贪心算法

设计贪心算法的步骤：

1. 将最优化未提转化为这样的形式：对其做出一次选择后，只剩下一个子问题需要求解；

2. 证明做出贪心选择后，原问题总是存在最优解，即贪心选择总是安全的

3. 证明做出贪心选择后，剩余的子问题满足性质：其最优解与贪心选择组合即可得到原问题的最优解，这样就得到了最优子结构

贪心选择性质的第一要素是：我们可以通过做出局部最优选择来构造全局最优解。贪心算法进行选择时可能依赖之前做出的选择，但不依赖任何将来的选择或是子问题的解。

证明每个步骤做出贪心选择能生成全局最优解：首先考察某个子问题的最优解，然后用贪心选择替换某个其他选择来修改此解，从而得到一个相似但更小的子问题。

通过对输入进行预处理或者使用合适的数据结构（通常是优先队列），通常可以使贪心选择更快。

一般来说，贪心算法是将问题分解为最优子结构之后，剩余一个最优子问题。

### 2.1 郝夫曼编码

郝夫曼编码可以有效地压缩数据。郝夫曼编码采用变长编码，将高频字符赋予短码字，低频字符赋予长码字

## 3. 摊还分析

摊还分析通过赋予对象费用来进行分析。通过做摊还分析可以获得某种特定数据结构的认识，有助于优化设计。

### 3.1 聚合分析

对于所有n，一个n个操作的序列最坏情况下花费的总时间为T(n)，因此，在最坏情况下，每个操作的平均代价，或摊还分析为T(n)/n.

聚合分析中的摊还代价是适用于每个操作的，即使序列中有多个操作类型。而核算法和势能算法对不同类型的操作可能赋予不同的摊还代价。

### 3.2 势能法

势能法摊还分析将“势能”作为数据结构中特定对象的信用。

# 第六部分 高级数据结构

B树，是为磁盘存储而设计的一类平衡搜索树。度量B树的性能，不仅要考虑动态集合操作消耗多少计算时间，而且还要考虑这些操作执行了多少次磁盘存取。

## 1. B树

B树是为磁盘或其他存取的辅助存储设备而设计的一种平衡搜索树。B树的节点可以有很多孩子。

由于在大多数系统中，B树算法的运行时间主要由它所执行的DISK-READ和DISK-WIRTE操作的次数决定，所以我们希望这些操作能够读或写尽可能多的信息。因此一个B树节点通常和一个完成磁盘页一样大，并且磁盘页的大小限制了一个B树结点可以包含孩子的个数。

### 1.1 B树的定义

实际上，为每个关键字存放一个指针，这个指针指向存放该关键字的卫星数据的磁盘页。伪代码中都隐含地假设了当一个关键字从一个结点移动到另一个结点时，关键字及其卫星数据都是一块移动的。

一种常见的B树变种—-B+树，它把所有的卫星数据都存储在叶节点中，内部节点只存放关键字和孩子指针，最大化了内部结点的分支因子。

一颗B树T具有如下性质：

1. 每个节点x有如下属性：
   1. x.n，当前存储在节点x中的关键字的个数
   2. x.n个关键字本身x.key1,x.key2,...,x.keyn,以非降序存放。
2. 每个内部结点x还包含x.n+1个指向其他孩子的指针.
3. 每个叶结点具有相同的深度
4. 每个结点所包含的关键字个数有上界和下界，用一个B树的最小度数（t≥2）来表示
   1. 除了根结点意外的每个结点都必须至少有t-1个关键字
   2. 每个结点之多可包含2t-1个关键字，因此，一个内部结点之多可有2t个孩子。当一个结点恰有2t-1个关键字时，该节点是满的

B树的高度：对任意一棵包含n(n≥1)个关键字、高度为h、最小度数t≥2的B树有：

### 1.2 B树上的基本操作：搜索、创建、插入

搜索一棵B树和搜索一棵二叉树相似

|  |
| --- |
| **搜索B树** |
| B**-**TREE**-**SEARCH**(**x**,** k**)**  i **=** 1  **while** i ≤ x**.**n and k **>** x**.**key**[**i**]**  i **=** i **+** 1  **if** i ≤ x**.**n and k **==** x**.**key**[**i**]**  **return** **(**x**,** i**)**  **else** **if** x**.**leaf // 如果x是叶结点，说明没有找到  **return** NIL  **else** DISK**-**READ**(**x**,** c**)**  **return** B**-**TREE**-**SEARCH**(**x**.**c**[**i**],** k**)** |
| **创建空的B树** |
| B**-**TREE**-**CREATE**(**T**)**  1 x **=** ALLOCATE**-**NODE**()**  2 x**.**leaf **=** TURE  3 x**.**n **=** 0  4 DISK**-**WRITE**(**x**)**  5 T**.**root **=** x |

**向B树中插入一个关键字**

**将一个**关键字插入一个已经存在的叶节点上，由于不能讲关键字插入一个满的叶节点，故引入一个操作，将一个满(2t-1)的结点y按其中间关键字分裂成为两个各含t-1个关键字的结点。中间关键字被提升到y的父**节点，以标识两颗新树的划分点。但如果y的父节点也是满的，就必须在插入新的关键字之前将其分裂，最终满结点会沿着树向上传播。**

**与二叉搜索树不同，当沿着树往下查找新的关键字位置时，就分裂沿途遇到的每个满的节点（包括叶结点本身）。**

**对根进行分裂是增加B树高度的唯一途径**

|  |
| --- |
| B**-**TREE**-**SPLIT**-**CHILD**(**x**,** i**)**  1 z **=** ALLOCATE**-**NODE**()**  2 y **=** x**.**c**[**i**]**  3 z**.**leaf **=** y**.**leaf  4 z**.**n **=** t **-** 1  5 **for** j**=**1 to t**-**1  6 z**.**key**[**i**]** **=** y**.**key**[**j**+**t**]**  7 **if** not y**.**leaf  8 **for** j**=**1 to t  9 z**.**c**[**j**]** **=** y**.**c**[**j**+**t**]**  10 y**.**n **=** t**-**1  11 **for** j**=**x**.**n**+**1 downto i**+**1  12 x**.**c**[**j**+**1**]** **=** x**.**c**[**j**]**  13 x**.**c**[**i**+**1**]** **=** z  14 **for** j**=**x**.**n downto i  15 x**.**key**[**j**+**1**]** **=** x**.**key**[**i**]**  16 x**.**key**[**i**]** **=** y**.**key**[**t**]**  17 x**.**n **=** x**.**n **+** 1  18 DISK**-**WRITE**(**y**)**  19 DISK**-**WRITE**(**z**)**  20 DISK**-**WRITE**(**x**)** |

**以沿树单程下行的方式向B树插入关键字**

**在一棵高度为h的B树T中，以沿树单程下行的方式插入一个关键字k的操作需要O(h)次磁盘存取。**

|  |
| --- |
| B**-**TREE**-**INSERT**(**T**,** k**)**  1 r **=** T**.**root  2 **if** r**.**n **==** 2t **-** 1  3 s **=** ALLOCATE**-**NODE**()**  4 T**.**root **=** s  5 s**.**leaf **=** FALSE  6 s**.**n **=** 0  7 s**.**c**[**1**]** **=** r\  8 B**-**TREE**-**SPLIT**-**CHILD**(**s**,** 1**)**  9 B**-**TREE**-**INSERT**-**NONFULL**(**s**,** k**)**  10 **else** B**-**TREE**-**INSERT**-**NONFULL**(**r**,** k**)** |

**辅助递归过程B-TREE-INSERT-NONFULL将关键字插入结点x，要求假定在调用该过程时x是非满的。**

|  |
| --- |
| B**-**TREE**-**INSERT**-**NONFULL**(**x**,** k**)**  1 i **=** x**.**n  2 **if** x**.**leaf  3 **while** i ≥ 1 and k **<** x**.**key  4 x**.**key**[**i**+**1**]** **=** x**.**key**[**i**]**  5 i **=** i **-** 1  6 x**.**key**[**i**+**1**]** **=** k  7 x**.**n **=** x**.**n **+** 1  8 DISK**-**WIRTE**(**x**)**  9 **else**  10 **while** i ≥ 1 and k **<** x**.**key**[**i**]**  11 i **=** i **-** 1  12 i **=** i **+** 1  13 DISK**-**READ**(**x**.**c**[**i**])**  14 **if** x**.**c**[**i**].**n **==** 2t **-** 1  15 B**-**TREE**-**SPLIT**-**CHILD**(**x**,** i**)**  16 **if** k **>** x**,**key**[**i**]**  17 i **=** i **+** 1  18 B**-**TREE**-**INSERT**-**NONFULL**(**x**.**c**[**i**],** k**)** |

### ****1.3 从B树中删除关键字****

B树中删除关键字与插入操作类似，必须防止因删除操作而导致树的结构违反B树的性质：除根结点外，其他结点允许最少关键字数t-1。

删除操作分一下情况：

* 如果关键字k在节点x中，并且x是叶结点，则从x中删除k。
* 如果关键字k在x中，并且x是内部结点，则做以下操作：
* 如果结点x中前于k的子结点y至少包含t个关键字，则找出k在以y为根的子树中的前驱k1，并在x中用k1代替k（找到k1并删除它可在沿树下降的单过程中完成）
* 对称地，如果y有少于t的关键字，，则检查结点x中后于k的子节点z，如果z至少有t个关键字，则找出k在以z为根的子树中的后继k1，递归地删除k1，并在x中用k1代替k
* 否则，如果y和z都只含有t-1个关键字，则将k和z的全部合并进y，这样x就失去了k和执行z的指针，情切y现在包含2t-1个关键字。然后释放z并递归地从y中删除k
* 如果关键字k当前不在内部结点x中，则确定必包含k的子树的根x.c[i]，如果x.c[i]只有t-1个关键字，必须执行3a或3b来保证降至一个至少包含t个关键字的结点。
* 如果x.c[i]只含有t-1个关键字，但是它的一个相邻的兄弟至少包含t个关键字，则将x中的某一个关键字降至x.c[i]中，将x.c[i]的相邻左兄弟或右兄弟的一个关键字升至x，将该兄弟中相应的孩子指针移到x.c[i]中，这样就是得x.c[i]增加了一个额外的关键字
* 如果x.c[i]以及x.c[i]相邻的所有兄弟都只包含t-1个关键字，则将x.c[i]与一个兄弟合并，即将x的一个关键字一直新合并的结点，使之成为该结点的中间关键字。

## 2. 斐波那契堆

斐波那契堆有两种用途：支持一系列操作，这些操作构成了“可合并堆”；斐波那契堆的一些操作可以在常数摊还时间内完成，非常适合于需要频繁调用这些操作的应用

**可合并堆**：支持一下五中操作：

* MAKE-HEAP(): 创建和返回一个新的不含任何元素的堆
* INSERT(H,x): 将一个已填入关键字的元素x插入堆H中
* MINIMUM(H): 返回一个指向堆H中具有最小关键字元素的指针
* EXTRACT-MIN(H): 从堆H中删除最小关键字的元素，并返回一个指向该元素的指针
* UMION(H1，H2): 创建并返回一个包含堆H1和堆H2中所有元素的心堆，堆H1和H2销毁

斐波那契堆还支持：

* DECREASE-KEY(H, x, k): 将堆H中的元素x的关键字赋予新值k（k不大于当前的关键字）
* DELETE(H,x): 从堆H中删除x

斐波那契堆是一系列具有最小堆序的有根树（非二叉树）的集合：每个结点的关键字大于或等于它的父节点的关键字。

每个结点包含一个指向它父结点的指针x.p和一个指向它的某一个孩子的指针x.child。x的所有孩子被链接成一个环形的双向链表。

通过指针H.min来访问一个给定的斐波那契堆H，该指针指向具有最小关键字的树的根结点，称该节点为H的最小结点。所有树的根都用其left和right指针链成一个环形的双链表，称之为根链表。