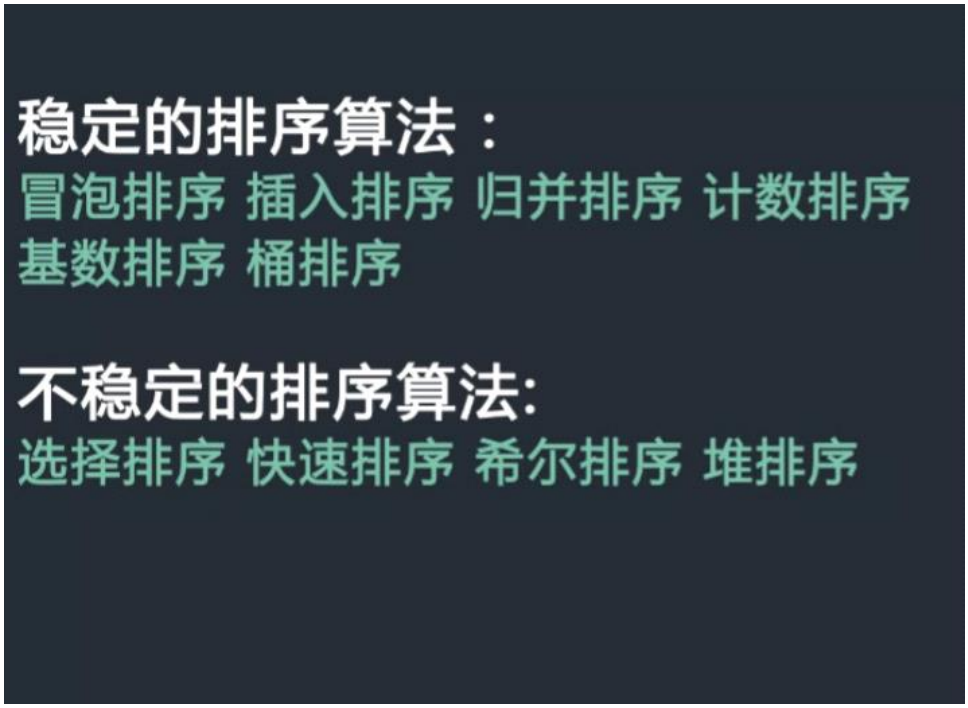
排序

2017年3月26日 10:30

目录

概况

排序法	平均时间	最差情形	稳定度	额外空间	备注
冒泡	$O(n^2)$	$O(n^2)$	稳定	O(1)	n小时较好
交换	$O(n^2)$	$O(n^2)$	不稳定	O(1)	n小时较好
选择	$O(n^2)$	$O(n^2)$	不稳定	O(1)	n小时较好
插入	$O(n^2)$	$O(n^2)$	稳定	O(1)	大部分已排序时较好
基数	$O(\log_R B)$	$O(\log_R B)$	稳定	O(n)	B是真数(0-9)， R是基数(个十百)
Shell	O(nlogn)	$O(n^s)$ 1<s<2	不稳定	O(1)	s是所选分组
快速	O(nlogn)	$O(n^2)$	不稳定	O(nlogn)	n大时较好
归并	O(nlogn)	O(nlogn)	稳定	O(1)	n大时较好
堆	O(nlogn)	O(nlogn)	不稳定	O(1)	n大时较好



什么叫不稳定性

拿选择排序距离如下：

这是一个2、2、2、1序列，最后的1和第一个2交换后，第一个2就变成最后一位了，这就破坏了稳定性，即破坏了相同元素原本的顺序

各类排序适用范围

总体原则：

1. n较小时（如n≤50），用**插入排序**或**选择排序**。
2. 若数据初始状态基本有序（正序），用**插入排序**。
3. 若n较大，用**快速排序**或者**堆排序**。若数组无序，快速排序的平均用时最短。
4. 若n较大，且要稳定排序，就只能用**归并排序**了。

快速排序，很多重复数字如何优化

返回基准元素位置时返回基准元素的最左和最有索引，减少排序次数。

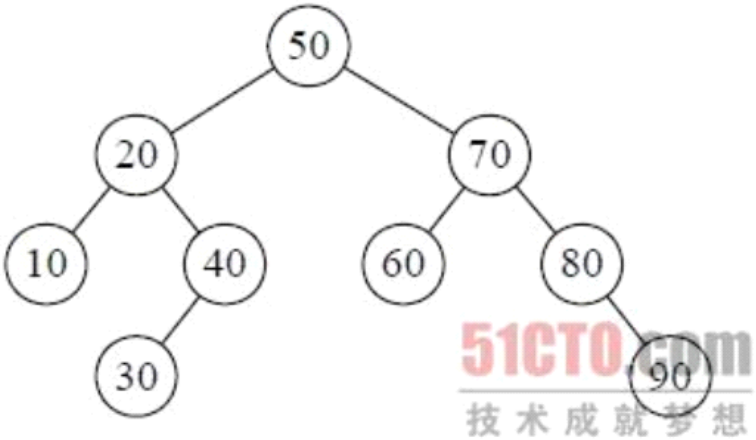
快排什么情况下最差?

初始序列基本有序时！！！最差为O(n*n)

快速排序的时间性能取决于快速排序递归的深度，在最坏的情况下，待排序的序列为正序或者逆序，每次划分只得到一个比上一次划分少一个记录的子序列，注意另一个为空。如果递归树画出来，它就是一棵斜树。此时需要执行n-1次递归调用，且第i次划分需要经过n-i次关键字的比较才能找到第i个记录，比较次数达到(n(n-1))/2，最终其时间复杂度为O(n*n)

快排什么情况下最优

快速排序的时间性能取决于快速排序递归的深度，当递归树的深度最小即树是平衡树的时候，此时最优。如下：它是{50,10,90,30, 70,40,80,60,20}在快速排序过程中的递归过程，即我们第一个随机选的数是50，然后在左边随机选的数是中间的20或者30，在右边随机选的数是中间的70或者80，这样递归下来数就是平衡的。



快速排序、归并排序，过程和优缺点

快速排序：先选定一个基准元素，按照这个基准元素将数组划分，再在被划分的数组上重复

上过程，最后可以得到排序结果。

归并排序：将数组不断细分成最小的单位，然后每个单位分别排序，排序完以后合并，重复这个过程就得到了排序结果

优缺点：归并排序稳定且最高最低时间复杂度都是nlogn，但是占用额外空间；快排不稳定，最高时间复杂度n2，最低时间复杂度nlgn，不占用额外空间

希尔排序（描述），为什么这么操作

改良的插入排序。插入排序在数组基本有序的时候可大大降低时间复杂度，希尔排序通过将数组分块后对每块数组进行插入排序，每次排序完成，块数减少一倍，数组也相对变得有序，知道最后对整个数组进行插入排序，则排序完成

O(N)

- 计数排序：适用于序列最大值和最小值相差不大的情况
- 基数排序：适用非负数且要知道最大值的位数

O(N方)

- 冒泡排序、选择排序：不管原始序列是什么顺序，时间复杂度都是严格的O(N方)
- 插入排序：n较小时（如n≤50），插入排序的过程与原始顺序有关，每个元素移动距离不超过K，K是元素要移动的最大间距，时间复杂度为O(N*K)

O(NlogN)

- 快速排序：快速排序是目前基于比较的内部排序中被认为是最好的方法，当初始序列无序时，快速排序的平均时间最短。初始序列有序或者基本有序时，快速排序时间复杂度降为O(n*n)
- 归并排序：与数组原始顺序无关
- 堆排序：与数组原始顺序有关，适合几乎有序的数组（即把数组排好顺序的话，每个元素移动的距离不超过k)

例题：

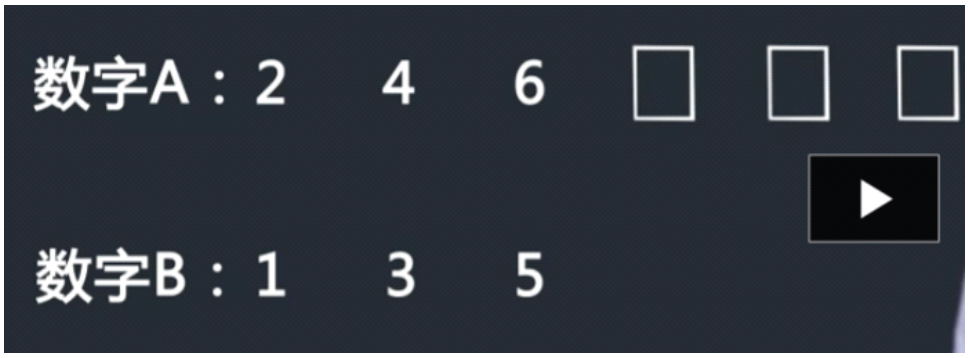
给定一个数组，判断数组中是否有重复值，必须保证额外空间复杂度为O(1)

解答：如果没有额外空间复杂度的要求，应该用哈希表来实现。如果加上了空间复杂度的要求，则应该先把数组排序，然后判断。用非递归版本的堆排序。（希尔排序、冒泡、插入排序都是空间复杂度为O(1)）

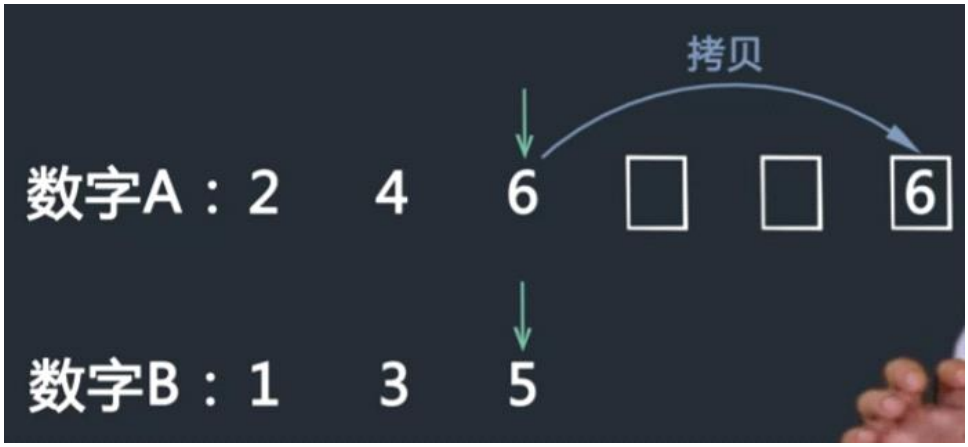
例题：

把两个有序数组合并为一个数组，第一个数组空间正好可以容纳两个数组的元素。

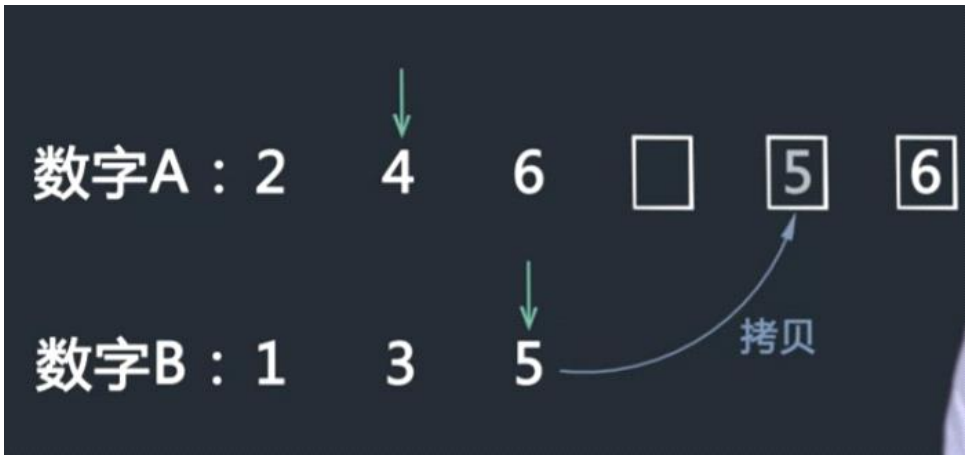
解答：



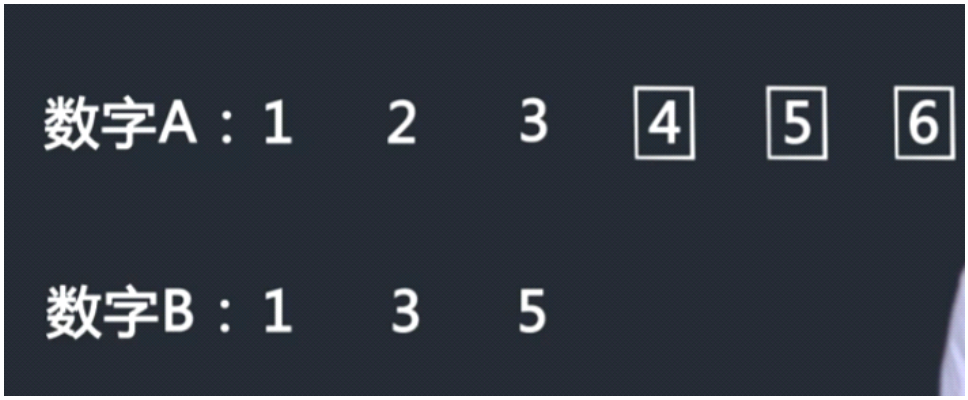
有序数组A和B，A后面有3个空位



首先6和5比较，6大，则6拷贝到数组最后的位置上



接下来4和5进行比较，5比较大，则5拷贝到数组第二后的位置上



依次比较所有数，直到有序数组B完全拷进到有序数组A，则搞定

关键在于从后往前覆盖A，这样保证A有用的部分不会被覆盖掉

布隆过滤器

2017年12月13日 9:35

布隆过滤器

它可以判断出某个元素，肯定不在集合里，或者可能在集合里。布隆过滤器相对于直接使用数据库的优势就是**占用空间小**。

基本原理

一个空的布隆过滤器是一个m位的位数组，所有位的值都为0。定义了k个不同的符合均匀随机分布的哈希函数，每个函数把集合元素映射到位数组的m位中的某一位。

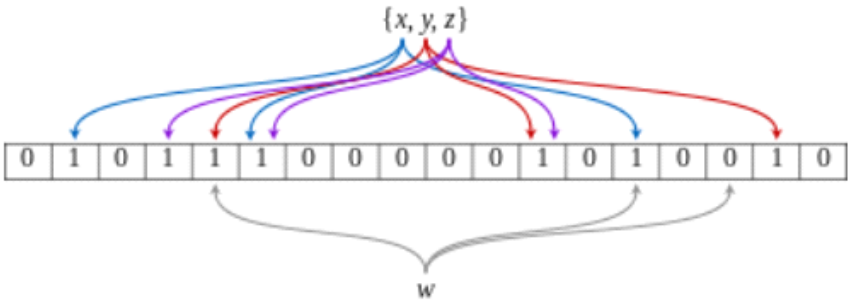
添加一个元素：

先把这个元素作为k个哈希函数的输入，拿到k个数组位置，然后把所有的这些位置置为1。

查询一个元素（测试这个元素是否在集合里）：

把这个元素作为k个哈希函数的输入，得到k个数组位置。这些位置中只要有任意一个是0，元素肯定不在这个集合里。如果元素在集合里，那么这些位置在插入这个元素时都被置为1了。如果这些位置都是1，那么要么元素在集合里，要么所有这些位置是在其他元素插入过程中被偶然置为1了，导致了一次“误报”。

例图：



https://www.nowcoder.com/study/vod/1/8/1

网页黑名单系统
垃圾邮件过滤系统
爬虫的网址判断重复系统
容忍一定程度的失误率
对空间要求较严格

⟹ 布隆过滤器

布隆过滤器可精确的代表一个集合

可精确判断某一元素是否在此集合中

精确程度由用户的具体设计决定

做到100%的精确即正确是不可能的

布隆过滤器的优势在于，利用很少的空间可以做到精确率较高。

布隆过滤器的bitarray大小如何确定？

大小为m，样本数量为n，失误率为p。

n=100亿，p=0.01%

单个样本大小不影响布隆过滤器大小，只影响了哈希函数的实现细节。

$$m=-\frac{n\times\ln p}{(\ln 2)^2}$$

求得m=19.19n，向上取整为20n。

2000亿bit，约为25G。

$$k=\ln 2\times\frac{m}{n}=0.7\times\frac{m}{n}$$

求得k=14

总结生成布隆过滤器的过程：

1、注意到题目允许有一定程度的失误率。

2、根据样本个数n，和允许的失误率p，
结合以下公式：

$$m=-\frac{n\times\ln p}{(\ln 2)^2}$$

求出m.

3、根据已经求得的m，以及以下公式：

$$k=\ln 2\times\frac{m}{n}=0.7\times\frac{m}{n}$$

求得哈希函数个数k.

4、根据向上取整后的m，n，k，根据以下公式：

$$(1-e^{-\frac{nk}{m}})^k$$

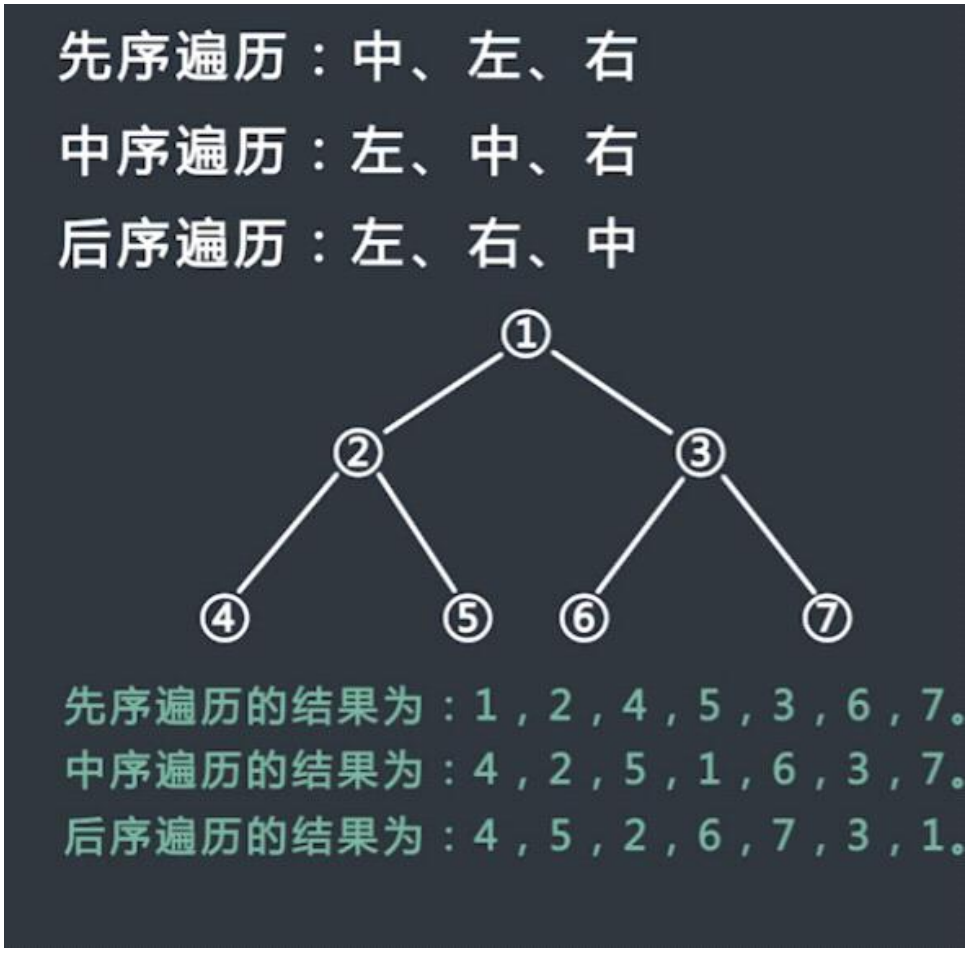
求得真实失误率p.

二叉树

2017年12月6日 15:04

目录

先序遍历：中、左、右
中序遍历：左、中、右
后序遍历：左、右、中



打印二叉树

不管是下面的递归方法还是非递归方法，遍历整棵树的时间复杂度都是O(N)，N为二叉树的节点数，额外空间复杂度为O(L)，L是二叉树的层数。

先序

递归方式实现先序遍历

```
public void preOrderRecur(Node head) {
        if (head == null) {
            return;
        }
        System.out.print(head.value + " ");
        preOrderRecur(head.left);
        preOrderRecur(head.right);
}
```

非递归方式实现先序遍历

具体过程：

1、首先申请一个新的栈，记为stack。
2、然后将头节点head压入stack中。
3、每次从stack中弹出栈顶节点，记为cur，然后打印cur节点的值。如果cur右孩子不为空的话，将cur的右孩子先压入stack中。最后如果cur的左孩子不为空的话，将cur的左孩子压入stack中。
4、不断重复步骤3，直到stack为空，全部过程结束。

中序

递归方法实现中序遍历

```
public void inOrderRecur(Node head) {
        if (head == null) {
                return;
        }
        inOrderRecur(head.left);
        System.out.print(head.value + " ");
        inOrderRecur(head.right);
}
```

非递归方法实现中序遍历

具体过程：

1、申请一个新的栈，记为stack，申请一个变量cur，初始时令cur等于头节点。

2、先把cur节点压入栈中，对以cur节点为头的整棵子树来说，依次把整棵树的左边界压入栈中，即不断令cur=cur.left，然后重复步骤2。

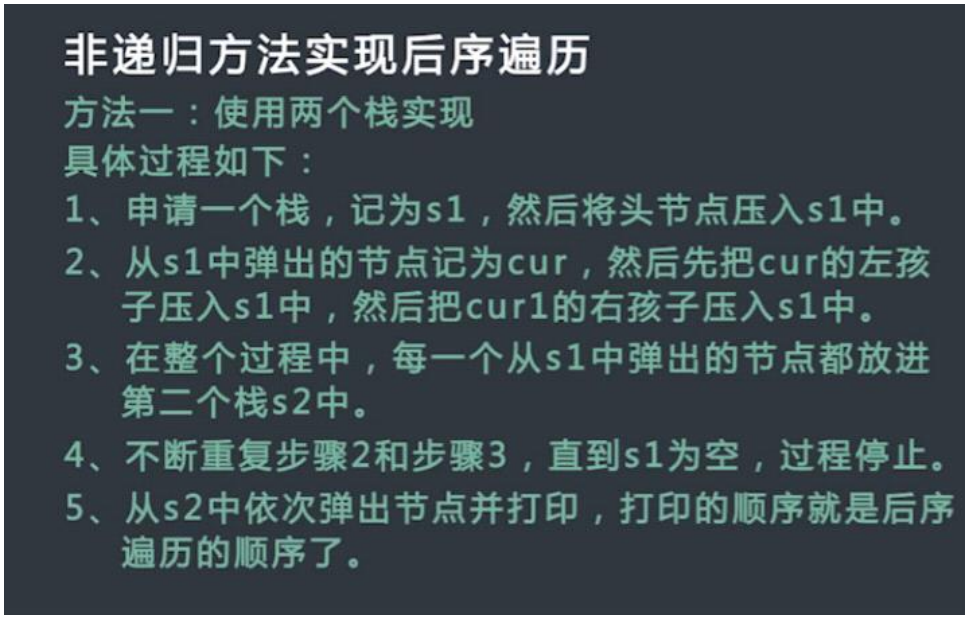
3、不断重复步骤2，直到发现cur为空，此时从stack中弹出一个节点，记为node。打印node的值，并让cur=node.right，然后继续重复步骤2。

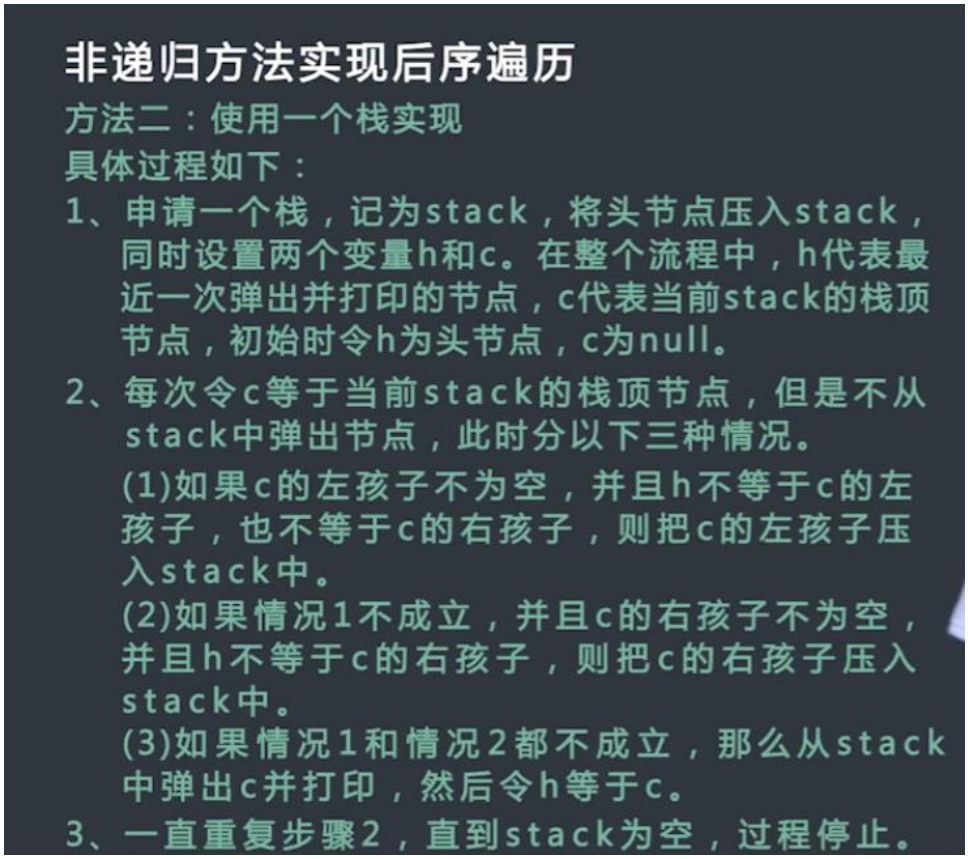
4、当stack为空并且cur为空时，整个过程结束。

后序

递归方法实现后序遍历

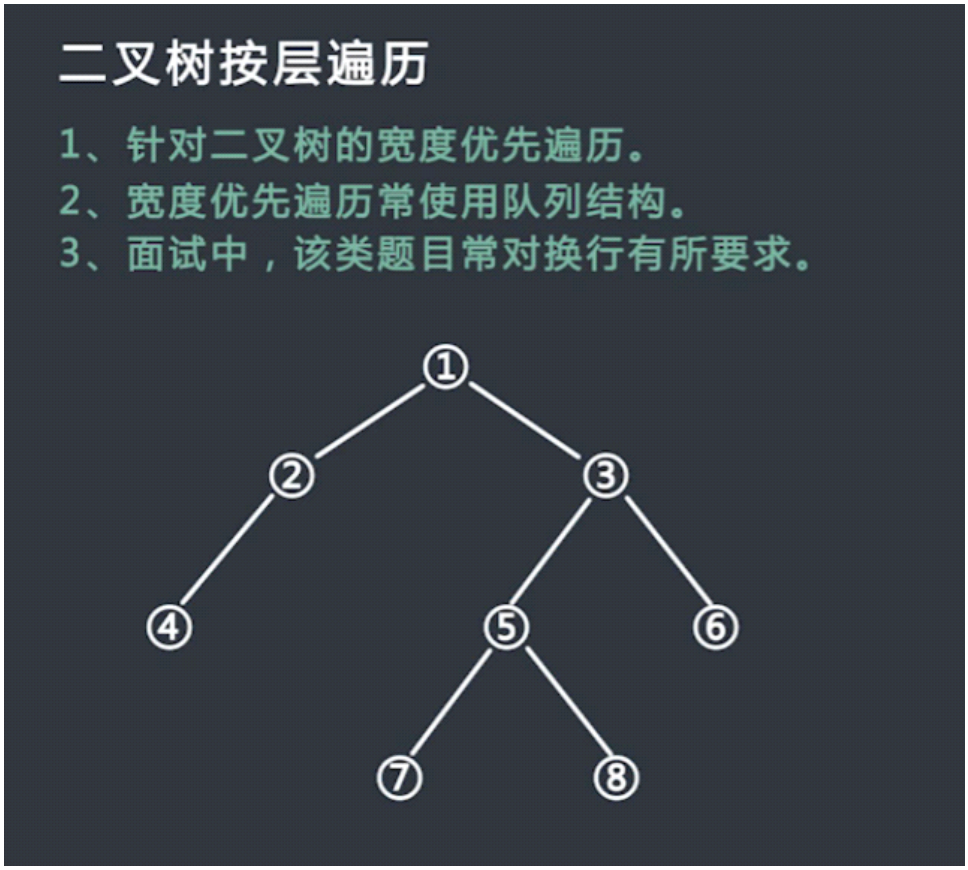
```
public void posOrderRecur(Node head) {
        if (head == null) {
                return;
        }
        posOrderRecur(head.left);
        posOrderRecur(head.right);
        System.out.print(head.value + " ");
}
```



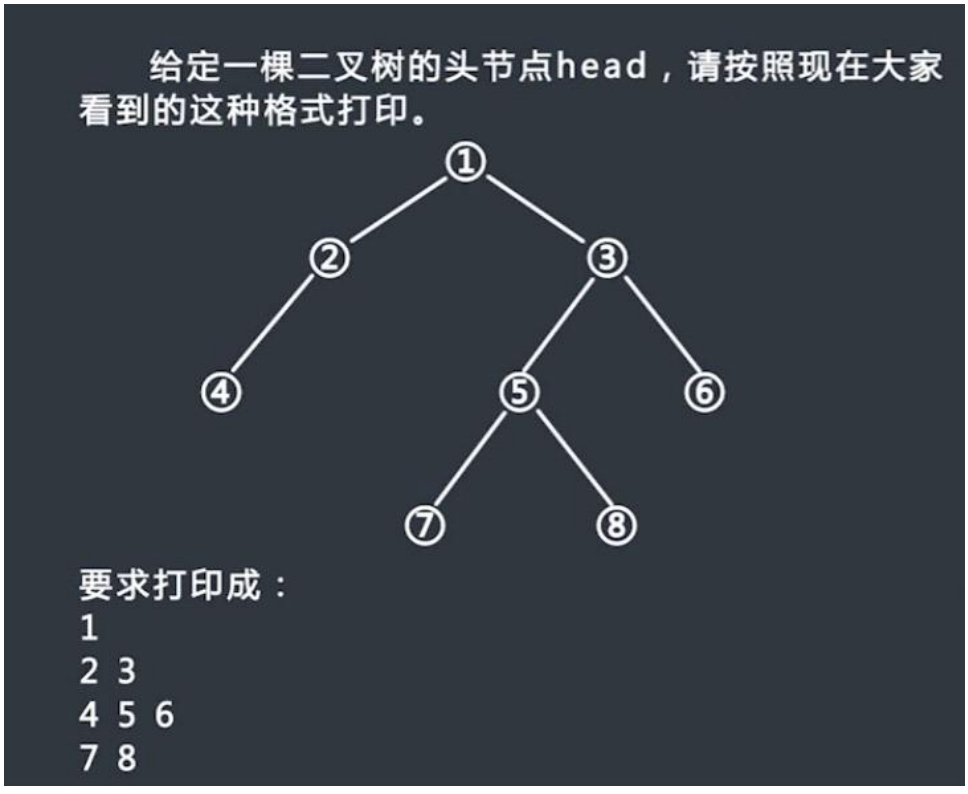


按层遍历二叉树

https://www.nowcoder.com/study/vod/1/7/4



例如：



二叉树序列化和反序列化

https://www.nowcoder.com/study/vod/1/7/4

二叉树的序列化和反序列化

1、二叉树→字符串（序列化）
2、字符串→二叉树（反序列化）

序列化的方式：

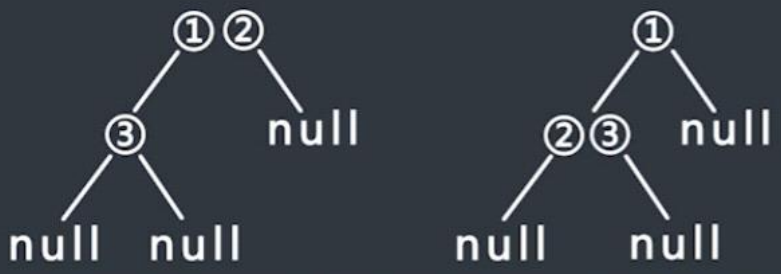
1、根据先序遍历序列化
2、根据中序遍历序列化
3、根据后序遍历序列化
4、按层序列化

先序遍历对二叉树进行序列化

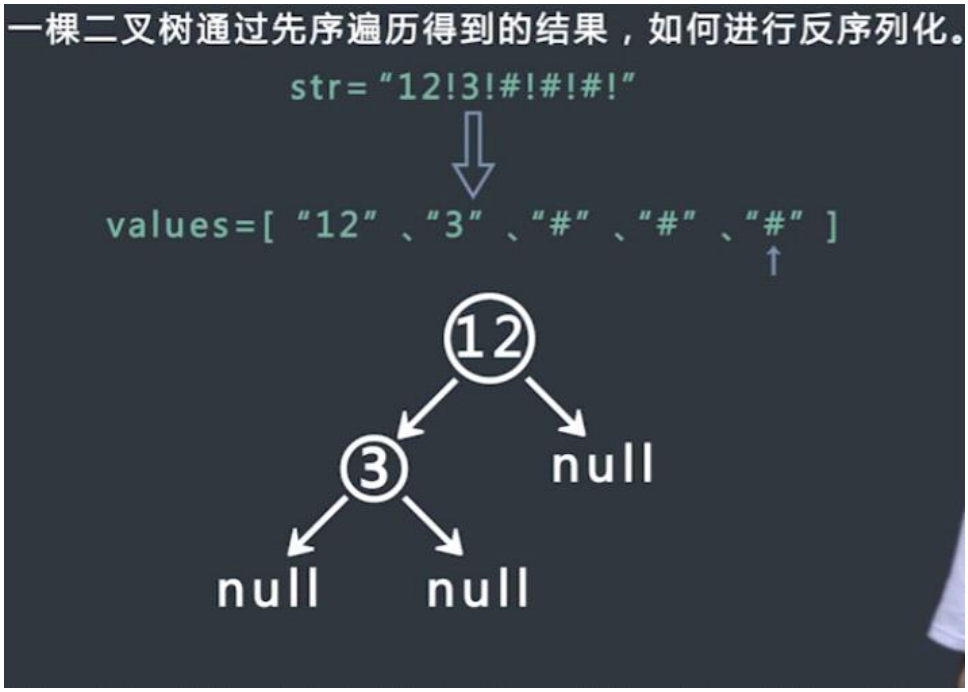
1、假设序列化结果为str，初始时str为空字符串。
2、先序遍历二叉树时如果遇到空节点，在str末尾加上“#!”。
3、如果遇到不为空的节点，假设节点值为3，就在str的末尾加上“3!”。

str=①②!③!#!#!#!

用一个特殊字符表示一个二叉树节点值的结束的意义。

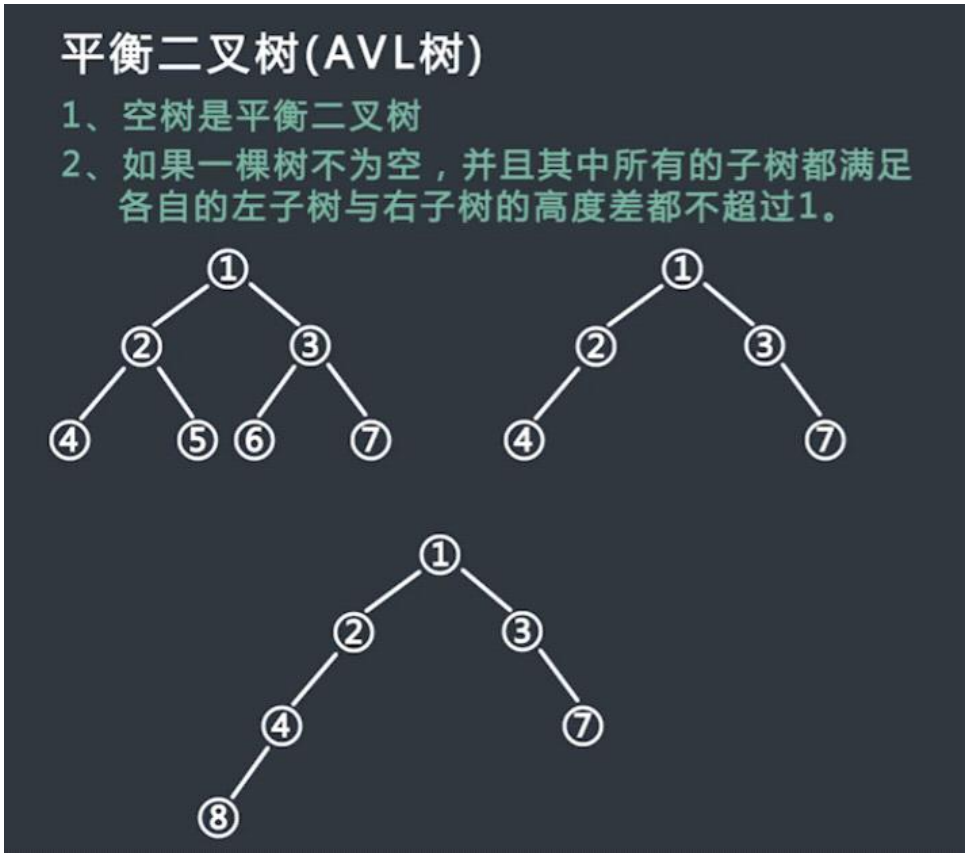


如果不用特殊符号表示值的结束，则这两棵树的序列化结果为：123###，说明不用特殊字符表示节点值结束的话，会产生歧义。

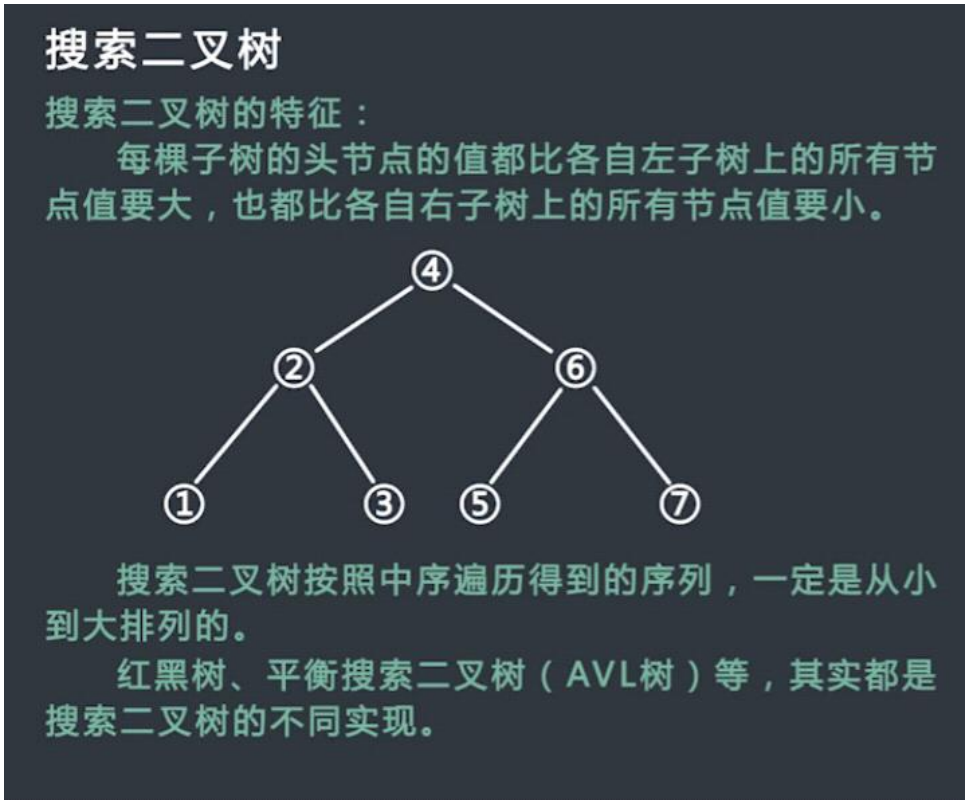


平衡二叉树

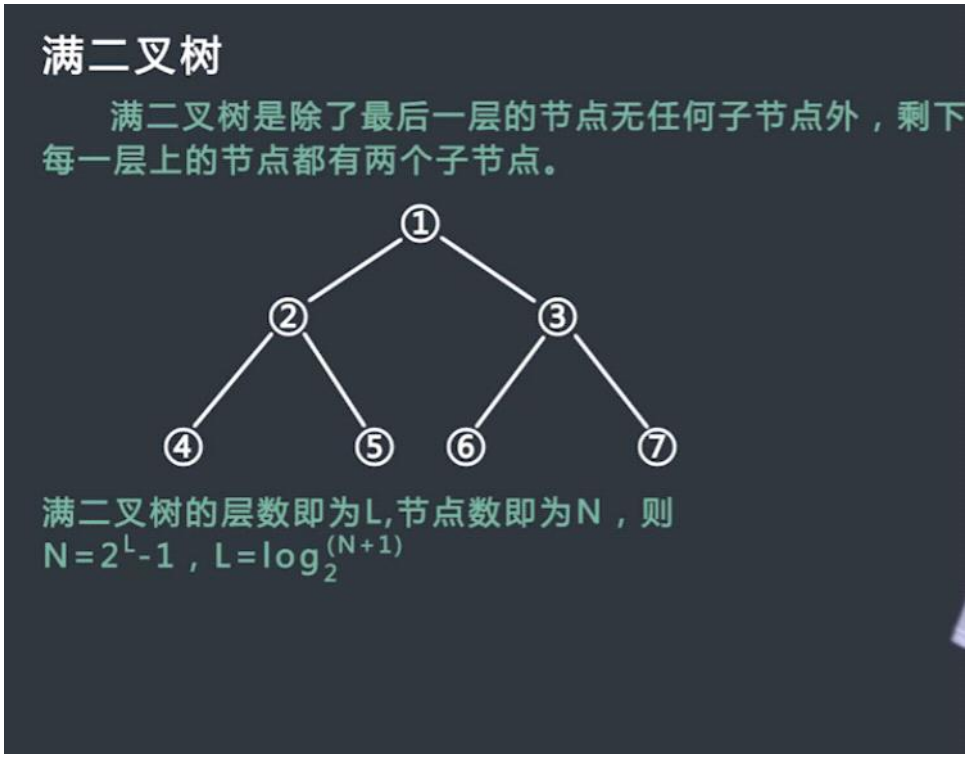
最下面那颗就不是平衡二叉树



搜索二叉树

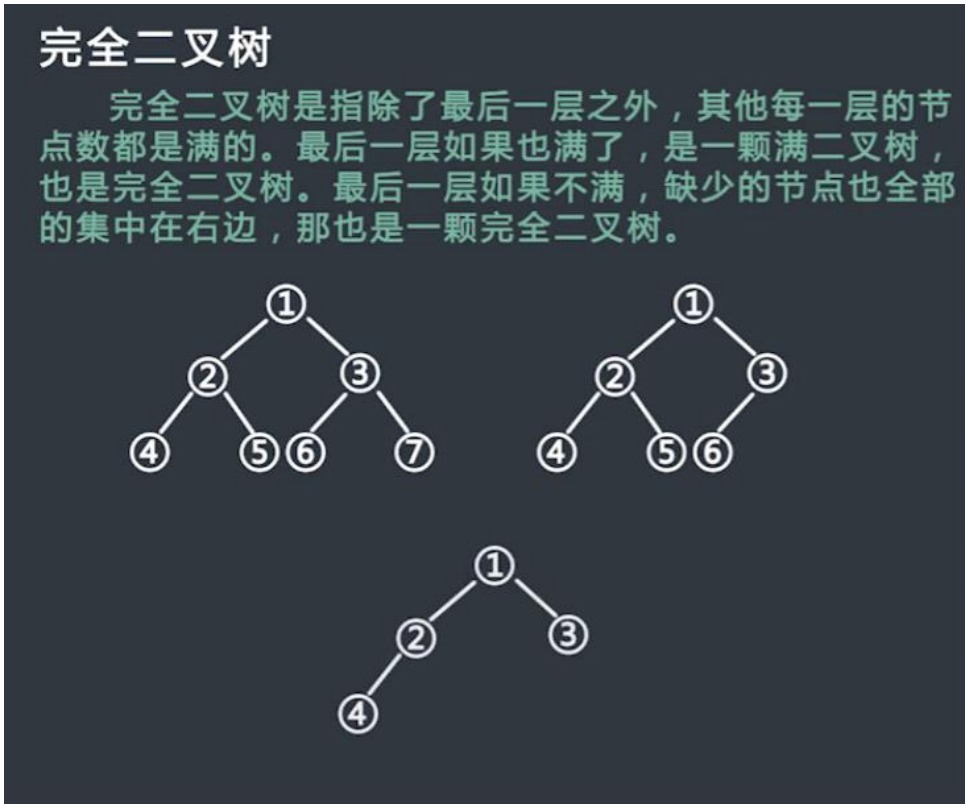


满二叉树



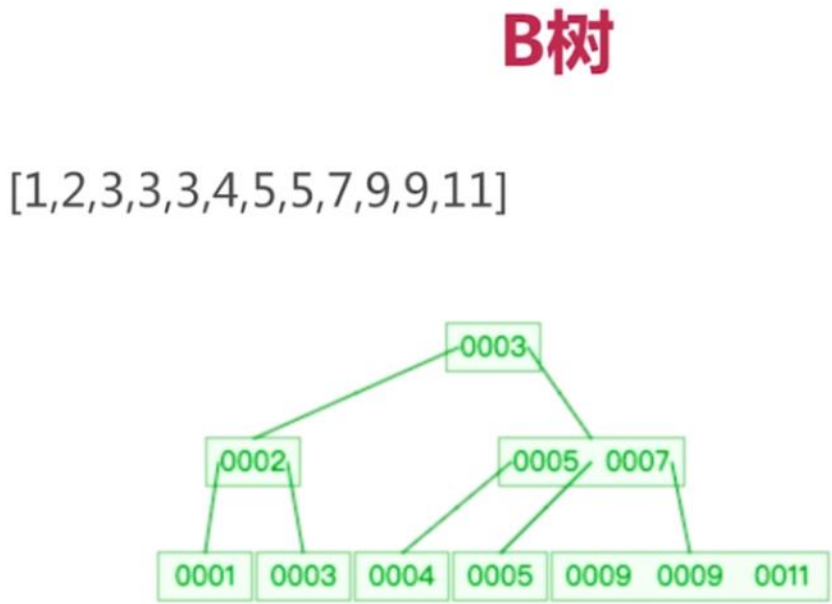
完全二叉树

第一个既是满二叉树也是完全二叉树，后面两个都是完全二叉树


完全二叉树
完全二叉树是指除了最后一层之外，其他每一层的节点数都是满的。最后一层如果也满了，是一颗满二叉树，也是完全二叉树。最后一层如果不满，缺少的节点也全部的集中在右边，那也是一颗完全二叉树。
1
2
3
4
5
6
7
1
2
3
4
5
6
1
2
3
4

B树

B树

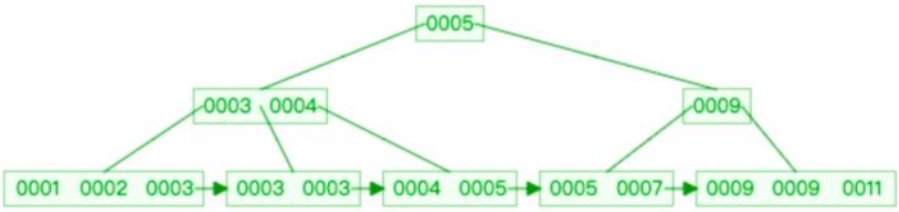

B树
[1,2,3,3,3,4,5,5,7,9,9,11]
0003
0002
0005
0007
0001
0003
0004
0005
0009
0009
0011

B+树

B+树

B+树

◆ [1,2,3,3,3,4,5,5,7,9,9,11]



优化后就是B+树。

1、跟B树的区别，所有原始的数值最终都会出现在叶子结点上，并且串起来就是原始数据的顺序。

2、根节点和中间的节点都是用来帮助索引的节点

红黑树

时间复杂度：

- 红黑树的操作时间跟二叉查找树的时间复杂度是一样的，执行查找、插入、删除等操作的时间复杂度为O（logn）
- 一棵含有n个节点的红黑树的高度至多为2log(n+1)

红黑树需要满足的五条性质：

性质一：节点是红色或者是黑色；

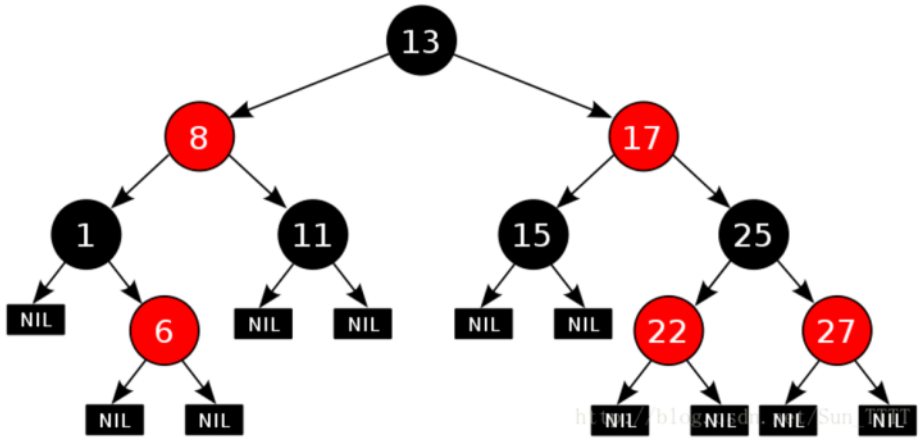
在树里面的节点不是红色的就是黑色的，没有其他颜色，要不怎么叫红黑树呢，是吧。

性质二：根节点是黑色；

根节点总是黑色的。它不能为红。

性质三：每个叶节点（即NIL节点，指空节点）是黑色；

这个可能有点理解困难，可以看图：



这个图片就是一个红黑树，NIL节点是个空节点，并且是黑色的。

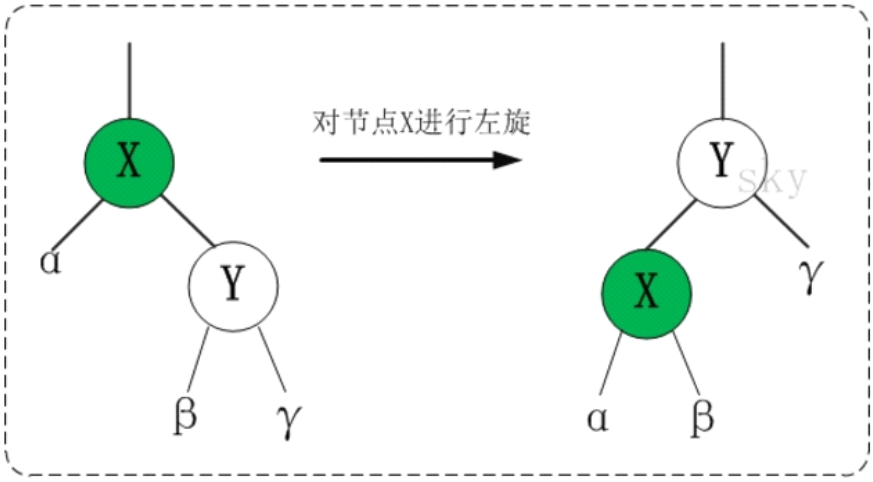
性质四：每个红色节点的两个子节点都是黑色的（也就是说不存在两个连续的红色节点）；

性质五：从任一节点到其每个叶节点的所有路径都包含相同数目的黑色节点；确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树

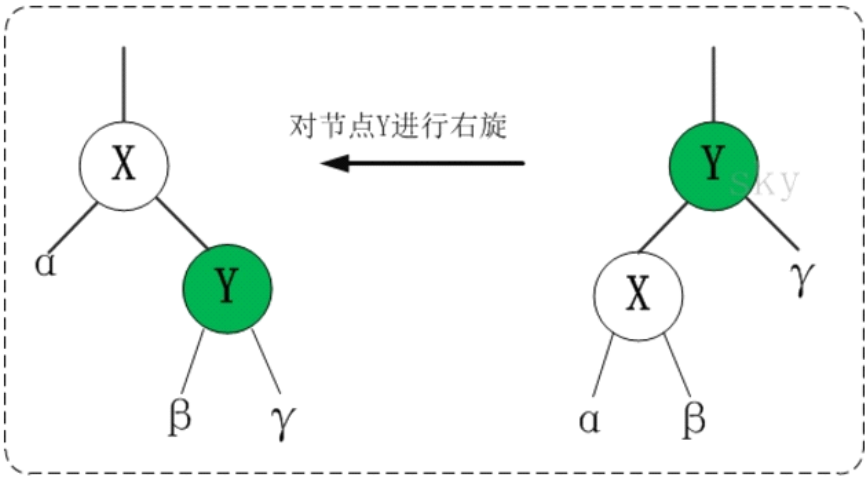
红黑树的基本操作：左旋、右旋

无论是左旋还是右旋，被旋转的树，在旋转前是二叉查找树，并且旋转之后仍然是一颗二叉查找树

1. 左旋



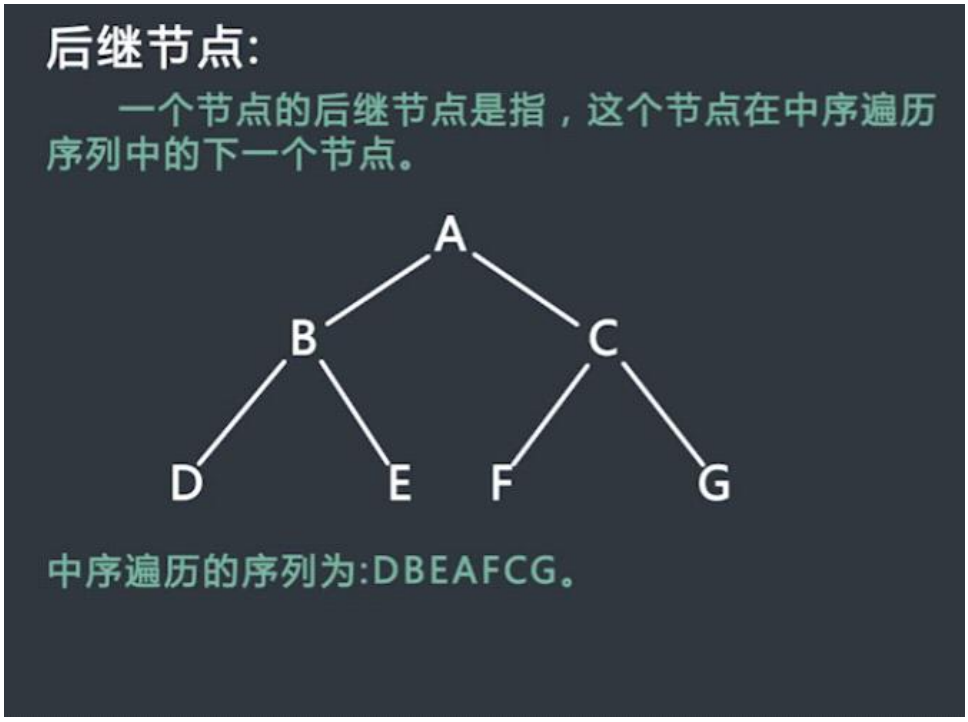
2. 右旋



红黑树的添加、删除操作：

- 添加：首先，将红黑树当作一颗二叉查找树，将节点插入；然后，将节点着色为红色；最后，通过旋转和重新着色等方法来修正该树，使之重新成为一颗红黑树。
    - 为什么插入的节点是红色？ 将插入的节点着色为红色，不会违背"特性(5)"！少违背一条特性，就意味着我们需要处理的情况越少。
- 删除：首先，将红黑树当作一颗二叉查找树，将该节点从二叉查找树中删除；然后，通过"旋转和重新着色"等一系列来修正该树，使之重新成为一棵红黑树。

后继节点和前驱节点



B树、B+树、B*树

2018年3月5日 11:19

https://www.cnblogs.com/George1994/p/7008732.html

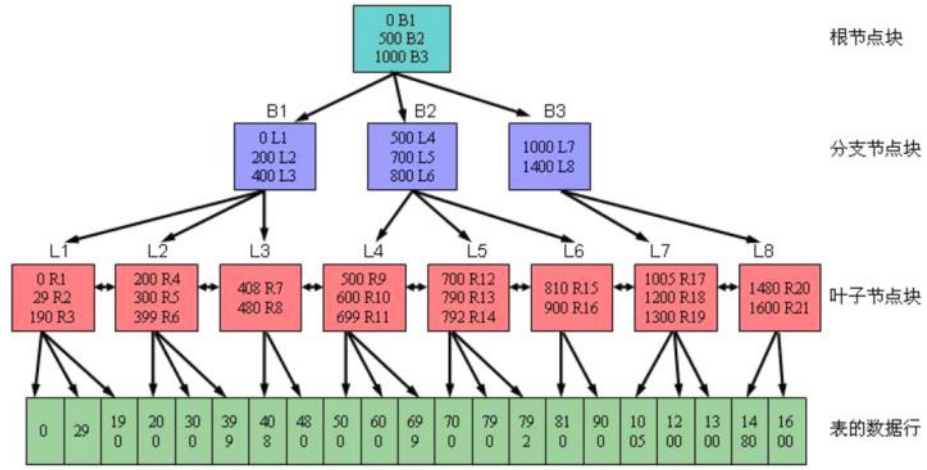
B树

为什么要B树?

1、当大规模数据存储到磁盘中的时候，显然定位是一个非常花费时间的过程。我们可以通过B树进行优化，提高磁盘读取时定位的效率。

2、为什么B类树可以进行优化呢？我们可以根据B类树的特点，构造一个多阶的B类树，然后在尽量多的在结点上存储相关的信息，保证层数尽量的少，以便后面我们可以更快的找到信息，磁盘的I/O操作也少一些，而且B类树是平衡树，每个结点到叶子结点的高度都是相同，这也保证了每个查询是稳定的。

举个栗子：



B+树

为什么要B+树?

由于B+树的数据都存储在叶子结点中，分支结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引，而B树则常用于文件索引。

B树和B+树的区别

这都是由于B+树和B具有这不同的存储结构所造成的区别，以一个m阶树为例。

1、关键字的数量不同；B+树中分支结点有m个关键字，其叶子结点也有m个，其关键字只是起到了一个索引的作用，但是B树虽然也有m个子结点，但是其只拥有m-1个关键字。

2、存储的位置不同；B+树中的数据都存储在叶子结点上，也就是其所有叶子结点的数据组合起来就是完整的数据，但是B树的数据存储在每一个结点中，并不仅仅存储在叶子结点上。

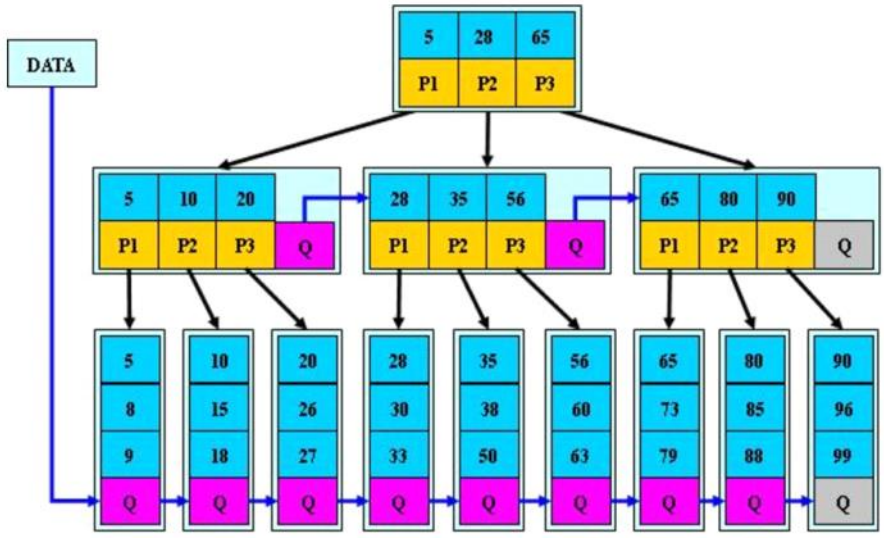
3、分支结点的构造不同；B+树的分支结点仅仅存储着关键字信息和儿子的指针（这里的指针指的是磁盘块的偏移量），也就是说内部结点仅仅包含着索引信息。

4、查询不同；B树在找到具体的数值以后，则结束，而B+树则需要通过索引找到叶子结点中的数据才结束，也就是说B+树的搜索过程中走了一条从根结点到叶子结点的路径。

B*树

是B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟的指针；

看下图的中间的Q就是指向兄弟的指针。



B*树

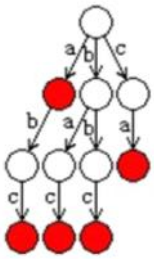
Trie树（字典树）

2018年3月18日 14:48

Trie树（字典树）概念

Trie树就是字典树，Trie树核心是空间换时间，利用字符串的公共前缀来降低查询的时间开销，从而达到提高效率的目的。

问答项目中前缀树算法中的前缀树就是Trie树



Trie树的基本性质

1. 查找、插入的复杂度均为O(len),len为字符串长度
2. 根结点不包含字符，其他的每一个节点只包含一个字符
3. 从根结点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串
4. 每个节点的所有子节点包含的字符都不相同
5. 建立+查询在Trie树中是可以同时执行的，建立的过程也就可以成为查询的过程，所以总的复杂度为O(n*len)，实际查询的复杂度只是O(len)。n是总共有多少个字（节点）。

红黑树

2018年3月19日 9:43

红黑树

时间复杂度：

- 红黑树的操作时间跟二叉查找树的时间复杂度是一样的，执行查找、插入、删除等操作的时间复杂度为O（logn）
- 一棵含有n个节点的红黑树的高度至多为2log(n+1)

红黑树需要满足的五条性质：

性质一：节点是红色或者是黑色；

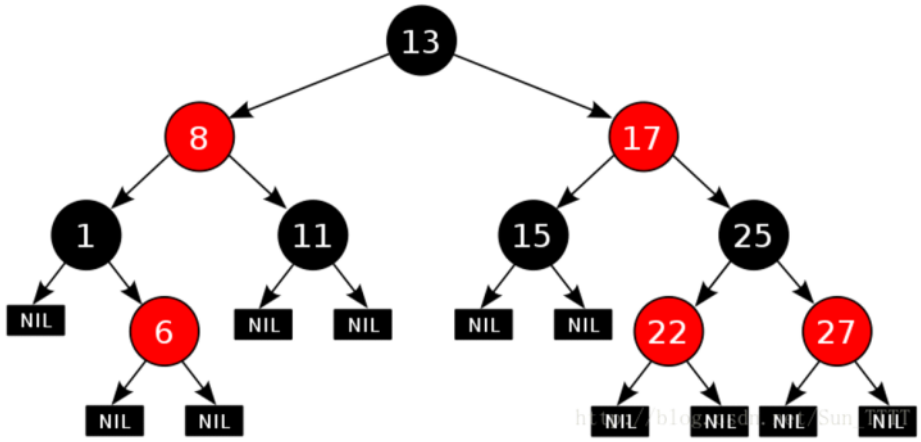
在树里面的节点不是红色的就是黑色的，没有其他颜色，要不怎么叫红黑树呢，是吧。

性质二：根节点是黑色；

根节点总是黑色的。它不能为红。

性质三：每个叶节点（即NIL节点，指空节点）是黑色；

这个可能有点理解困难，可以看图：



这个图片就是一个红黑树，NIL节点是个空节点，并且是黑色的。

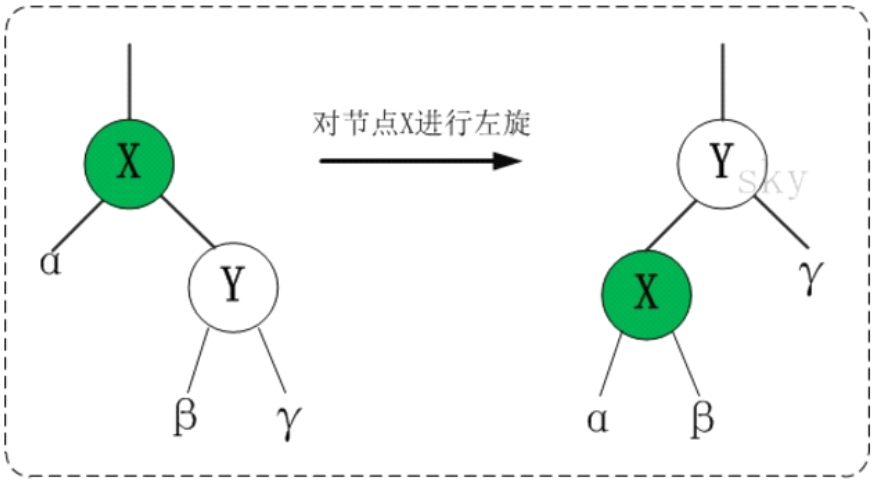
性质四：每个红色节点的两个子节点都是黑色的（也就是说不存在两个连续的红色节点）；

性质五：从任一节点到其每个叶节点的所有路径都包含相同数目的黑色节点；确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树

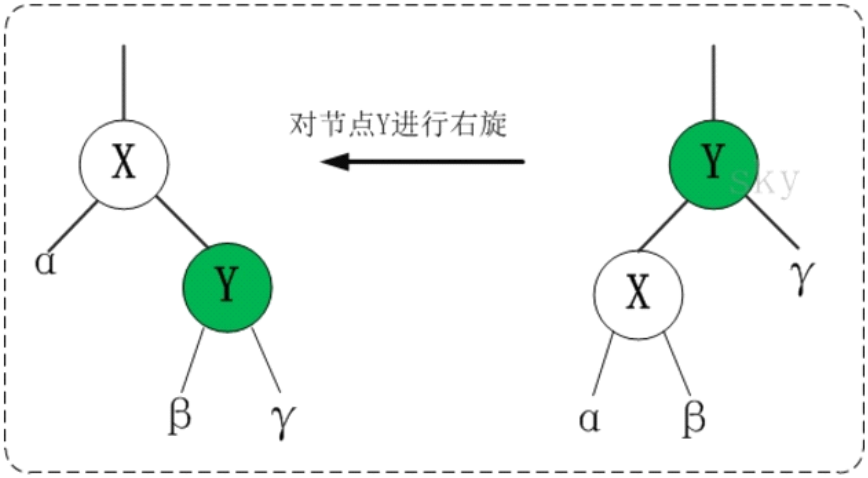
红黑树的基本操作：左旋、右旋

无论是左旋还是右旋，被旋转的树，在旋转前是二叉查找树，并且旋转之后仍然是一颗二叉查找树

1. 左旋



2. 右旋



红黑树的添加、删除操作：

- 添加：首先，将红黑树当作一颗二叉查找树，将节点插入；然后，将节点着色为红色；最后，通过旋转和重新着色等方法来修正该树，使之重新成为一颗红黑树。
  - 为什么插入的节点是红色？ 将插入的节点着色为红色，不会违背"特性(5)"！少违背一条特性，就意味着我们需要处理的情况越少。
- 删除：首先，将红黑树当作一颗二叉查找树，将该节点从二叉查找树中删除；然后，通过"旋转和重新着色"等一系列来修正该树，使之重新成为一棵红黑树。

为什么有红、黑两色?

1. 红黑树是平衡二叉树的变形，红黑树的颜色是保证红黑树查找速度的一种方式，目的是利用颜色值作为二叉树的平衡对称性的检查，在插入删除过程中只要满足红黑树定义要求，就能满足二叉树的相对平衡。
2. 从任意的节点开始到叶节点的路径，黑节点的个数是相同的，这就能保证搜索路径的最大长度不超过搜索路径的最短长度的2倍。

二分查找

2018年3月22日 11:15

适用场景

- 往一个有序的队列里插入数字
- 或者在一个有序队列中查找我们想要的数字
- 或者100个数插入100万个数的有序队列中
- 用二分查找的时间复杂度是O（logn）；n表示区间长度。

参考
http://blog.csdn.net/mengxiang000000/article/details/52751310

什么是二分查找?

1、我们首先引入这样一个问题：如果规定某一科目成绩分数范围：[0,100]，现在小明知道自己的成绩，他让你猜他的成绩，如果猜的高了或者低了都会告诉你，用最少的次数猜出他的成绩，你会如何设定方案？（排除运气成分和你对小明平时成绩的了解程度）

①最笨的方法当然就是从0开始猜，一直猜到100分，考虑这样来猜的最少次数：1（运气嘎嘎好），100（运气嘎嘎背）；

②其实在我们根本不知道对方水平的条件下，我们每一次的猜测都想尽量将不需要猜的部分去除掉，而又对小明不了解，不知道其水平到底如何，那么我们考虑将分数均分，

将分数区间一分为2，我们第一次猜的分数将会是50，当回答是低了的时候，我们将其分数区域从【0,100】确定到【51,100】；当回答高了的时候，我们将分数区域确定到【0,49】。这样一下子就减少了多余的50次猜想（从0数到49）（或者是从51到100）。

③那么我们假设当猜完50分之后答案是低了，那么我们需要在【51,100】分的区间内继续猜小明的分数，同理，我们继续折半，第二次我们将猜75分，当回答是低了的时候，我们将其分数区域从【51,100】确定到【76,100】；当回答高了的时候，我们将分数区域确定到【51,74】。这样一下子就减少了多余的猜想（从51数到74）（或者是从76到100）。

④就此继续下去，直到回复是正确为止，这样考虑显然是最优的、

位图BitMap

2018年3月24日 15:39

位图干嘛的

一句话：用来判断十亿、百亿海量数据中某数据是否存在

位图定义

https://blog.csdn.net/wenqiang1208/article/details/76724338

Java中的位图类，java.util.BitSet

https://blog.csdn.net/yaoweijq/article/details/5982265

BitSet类只支持Integer型，也就是10亿数据，利用两个BitSet处理Long型数据

https://blog.csdn.net/rishengcsdn/article/details/25621691

常见海量数据面试题

https://blog.csdn.net/LLZK_/article/details/53106697

例题1：给定100亿个整数，设计算法找到只出现1次的整数。可用内存为1GB。

遇到筛选出出现次数不超过N的整数（N不能太大），并且不对具体次数是多少做出要求时，一般考虑用位图。节省空间，且效率高。

1GB = 1073741824个Byte = 8589934592个 bit（位）。

题目的要求是找出只出现一次的数字，所以我们需要2个位来表示一个数的状态。00不存在，01出现1次，10出现多次，11无意义。

所以这80多亿个位刚好可以表示所有的整数（int最大可以表示40亿）。

即第0和第1位表示数字1的状态，第2位和第3位表示数字2的状态，以此类推。

通过这个方法，我们可以快速筛选出只出现一次的整数。

例题2：给一个100G大小的log file, log中存着IP地址, 设计算法找到出现次数最多的IP地址？

利用Hash切分，把100G切成100份，那么相同的IP地址必然在同一份里，之后对每一份（大小1G，可放内存里）利用哈希表或者Map来计数，最后统计。

例题3：给两个文件，分别有100亿个整数，我们只有1G内存，如何找到两个文件交集。

两个文件各自Hash取模切分成200份，每份500M，拿取模相同的文件分别做位图处理，用0表示数字存在，用1表示数字不存在，然后同时遍历两个位图，取都是1的，其对应的数就存在。

Tips：1GB内存=80亿bit位，int最大可以表示40亿。

例题4：给两个文件，分别有100亿个URL，我们只有1G内存，如何找到两个文件交集。

两个文件各自Hash取模切分成200份，每份500M，拿取模相同的文件，第一个做遍历并放入HashMap，另一个遍历去查HashMap，如果有值就是交集。

例题5：查找一个元素是否在一个集合里，近似算法，节省空间

布隆过滤器 布隆过滤器

数组

2018年4月4日　14:52

前缀和

中心思想：任意一个子数组和都能表示为两个前缀和之差

一维前缀和

这个优化主要是用来在O（1）时间内求出一个序列a中,a[i]+a[i+1]+……+a[j]的和。

具体原理十分简单：用sum[i]表示（a[1]+a[2]+……+a[i]），其中sum[0]=0，**则（a[i]+a[i+1]+……+a[j]）即等于sum[j]-sum[i-1]。**

二维前缀和

同理，有一维就有二维。对于一个矩阵a，我们也能在O（1）时间内求出子矩阵[x1~x2][y1~y2]的和。

设sum[i][j]为子矩阵[1~i][1~j]的和。则由容斥原理得：

sum[0][j]=sum[i][0]=0

a[x1~x2][y1~y2]=sum[x2][y2]-sum[x1-1][y2]-sum[x2][y1-1]+sum[x1-1][y1-1]

应用问题

核心就两个字：**降维。**
面对许多高维问题，往往前缀和是最先想到的降维方法。
这样在降维的基础上，许多更进一步的优化才能实现。

算法题中：**解决SubArray问题的杀手锏。**

基础概念

2018年4月4日 14:58