1、排序二叉树

**排序二叉树：**

若它的左子树不空，则左子树上所有节点的值均小于它的根节点的值

若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值

它的左、右子树也分别为排序二叉树

**排序二叉树的插入操作：**

（1）若二叉排序树是空树，则key成为二叉排序树的根；

（2）若二叉排序树非空，则将key与二叉排序树的根进行比较。如果key的值等于根结点的值，则停止插入；如果key的值小于根结点的值，则将key插入左子树，如果key的值大于根结点的值，则将key插入右子树。

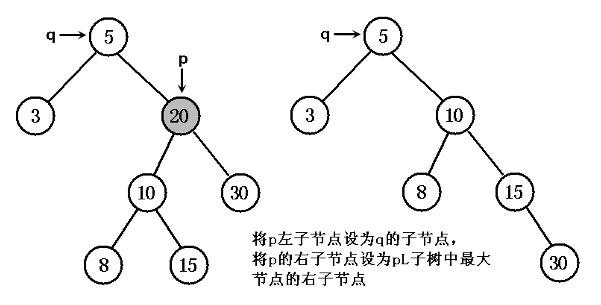
（3）重复步骤2，直到找到合适的插入位置。

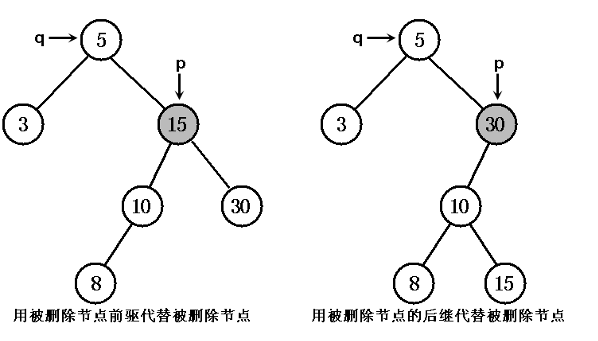
**排序二叉树删除操作：**

（1）被删除的节点是叶子节点，则只需将它从其父节点中删除即可。

（2）如果待删除节点左子树存在右子树不存在，或者左子树不存在右子树存在。直接将其子树中存在的一边候补上来即可。

（3）若左右子树都不为空，有两种方式维护;





TreeMap删除节点采用第二种方案，也就是用被删除节点的右子树中最小节点与被删除的节点交换的方式进行维护。

**排序二叉树之查找操作：**

从二叉排序树中进行查找时，根据树的性质，节点的左子树必定小于根节点，右子树必定大于根结点。如果查找的节点值小于根节点，则进入左子树，大于进入右子树，重复这个比较步骤直到找到这个节点或者这个节点不存在。

**总结:**

**排序二叉树可以快速检索**，但在最坏的情况下，如果插入的节点集本身就是有序的 ，要么是由小到大排列，要么是由大到小的排列，那么最后得到的排序二叉树将变成链表，所有节点只有左节点（如果插入节点集本身是大到小排列）；或者所有节点只有右节点（如果插入节点集是由小到大排列的），在这种情况下，排序二叉树就变成了普通的链表，其检索效率就会很差。

2、平衡二叉树（AVL）

**平衡因子**：某结点的左子树与右子树的高度差即为该结点的平衡因子。

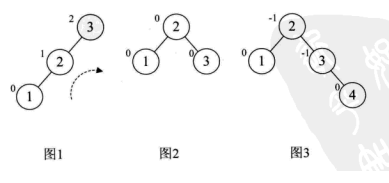
AVL树中任何两个结点的高度差值最大不查过1。

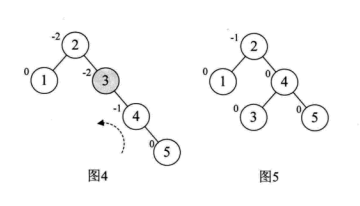
平衡二叉树构建的基本思想就是在在构建平衡二叉树的过程中，每当插入一个结点，先检查是否因插入而破坏了树的平衡性，若是，则找出最小平衡子树，在保持二叉排序树特性的前提下，调整关系。

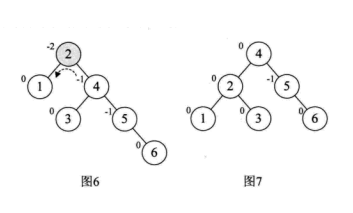
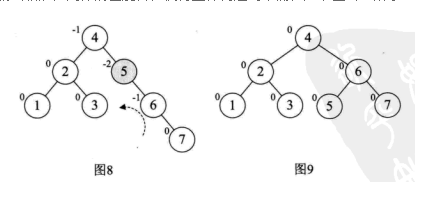
平衡因子为2,右旋，平衡因子为-2,左旋，只有符号统一才能旋转（插入结点后，最小不平衡子树的BF与它的子树的BF符号相反时，

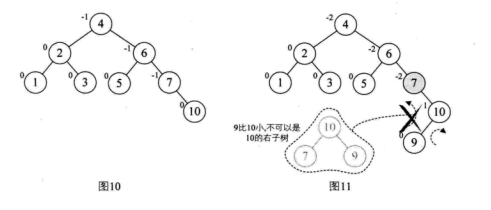
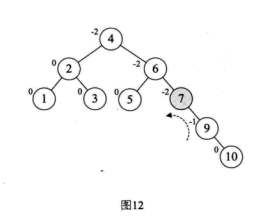
后，再反向旋转一次才能完成平衡操作），

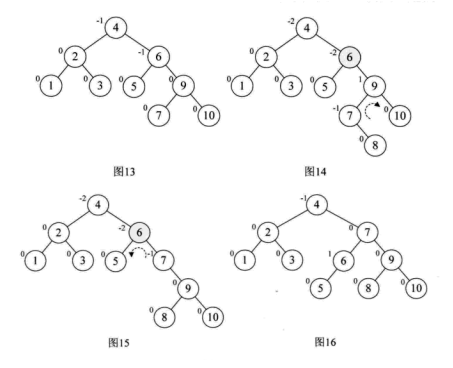
a[10] = {3, 2, 1, 4, 5, 6, 7, 10, 9, 8}需要构建二叉排序树







由于平衡二叉树也是排序二叉树，所以使用二叉树的插入、删除、和查找操作即可，只是在操作完成后，**为了下次能够保持一个好的检索效率，也为了防止这个链表退化为普通的链表，则需要对树进行旋转，以期再次达到平衡。**

3、红黑树

红黑树是一个更高效的检索二叉树，因此常常用来实现关联数组，JDK提供的集合类**TreeMap本身就是一个红黑树的实现**。**在JDK8后，HashMap开始用红黑树代替链表，查找由O（n）变成了O（Logn）**

**特性**：

**性质 1：**每个节点要么是红色，要么是黑色。

**性质 2：**根节点永远是黑色的。

**性质 3：**所有的叶节点都是空节点（即 null），并且是黑色的。

**性质 4：**每个红色节点的两个子节点都是黑色。（**从每个叶子到根的路径上不会有两个连续的红色节点**）

**性质 5：**从任一节点到其子树中每个叶子节点的路径都包含相同数量的黑色节点。

对于给定的黑色高度为 N 的红黑树，从根到叶子节点的最短路径长度为 N-1，（路径全黑）最长路径长度为 2 \* (N-1)（路径黑红交替）。

**读取操作**：

由于红黑树只是一个特殊的排序二叉树，因此对红黑树上的只读操作与普通排序二叉树的只读操作完全相同，只是红黑树保持了大致平衡，因此检索性能比排序二叉树要好很多，但在红黑树上进行插入操作和删除操作会导致树不再符合红黑树的特征，

为了每次插入删除结点时，需要进行结点颜色变换和进行旋转变换。

**插入结点后修复：**

1、以排序二叉树的方法插入新结点，并将它设为红色。

2、进行颜色调换和树旋转。

在介绍中，我们把新插入的节点定义为 N 节点，N 节点的父节点定义为 P 节点，P 节点的兄弟节点定义为 U 节点，P 节点父节点定义为 G 节点

**情形 1：新节点 N 是树的根节点，没有父节点**

在这种情形下，直接将它设置为黑色以满足性质 2。

**情形 2：新节点的父节点 P 是黑色**

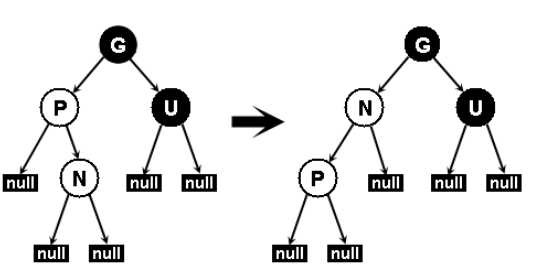
在这种情况下，新插入的结点是红色的，而且新结点N有两个黑色的叶子结点

**情形 3：如果父节点 P 和父节点的兄弟节点 U 都是红色**

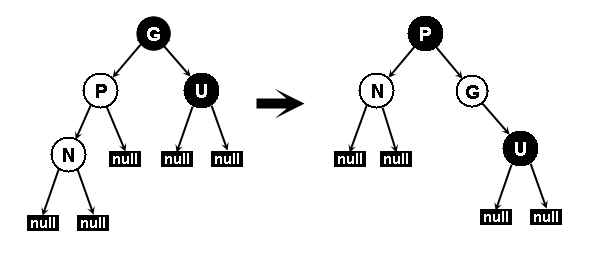
在这种情况下，程序应该将P结点,U结点都设置为黑色，并将P结点的父节点G设置为红色，然后把G当成是新插入的结点进行处理即可

**情形 4：父节点 P 是红色、而其兄弟节点 U 是黑色或缺少；且新节点 N 是父节点 P 的右子节点，而父节点 P 又是其父节点 G 的左子节点。**

在这种情况下，需要对新结点和其父结点进行左旋转操作，接着按照情形5处理以前的父节点P



**情形 4：父节点 P 是红色、而其兄弟节点 U 是黑色或缺少；且新节点 N 是父节点 P 的左子节点，而父节点 P 又是其父节点 G 的左子节点。**



在这种情形下，需要对结点G进行一次右旋，在旋转的过程中，以前的父节点P现在是新结点N和结点G的父节点，由于以前的结点G 是黑色，我们切换P和G的颜色，

**红黑树的优势：红黑树能够以O(log2(N))的时间复杂度进行搜索、插入、删除操作。此外,任何不平衡都会在3次旋转之内解决。这一点是AVL所不具备的。**

4、 B树/B+树

B树;(m阶)

1、每个结点至多有m棵子树。

2、根结点至少拥有两棵子树（存在子树的情况下）

3、除了根结点以外，其余每个分支结点至少拥有m/2棵子树

4、所有叶子结点都在同一层上

5、有K棵子树的分支结点

5、树的区别

AVL树和红黑树都是二叉搜索树的变体，AVL树是严格维持平衡的，红黑树是黑平衡的。但是维持平衡又需要额外的操作，这也加大了数据结构的时间复杂度，所以红黑树可以看做是二叉搜索树和AVL树的一个折中，可以尽量维持树的平衡，又不用话过多的时间来维持数据结构的性质。

AVL树：平衡二叉树，一般是用平衡因子差值决定并通过旋转来实现，左右子树树高差不超过1，

红黑树：平衡二叉树，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，确保没有一条路径会比其他路径长2倍，

B树：多路搜索树，关键字集合分布在整棵树中，搜索有可能在非叶子结点结束。

B+树：非叶子节点只保存索引 ，不保存数据，数据都保存在叶子结点中。

B树B+树特点：分支多层数少，一般用于数据库做索引，减少磁盘IO的次数

B+树相对B树磁盘读写代价更低：因为B+树非叶子结点只存储键值，单个节点占空间小，索引块能够存储更多的节点，从磁盘读索引时所需的索引块更少，而且B+Tree在叶子节点存放的记录以链表的形式链接，范围查找或遍历效率更高。Mysql InnoDB用的就是B+Tree索引。

6、排序算法



**归并排序 O(nlogn) O(nlogn) O(nlogn) 稳定**

**Arrays.sort()采用的是一种名为TimSort的排序算法，就是归并排序的优化版本。**

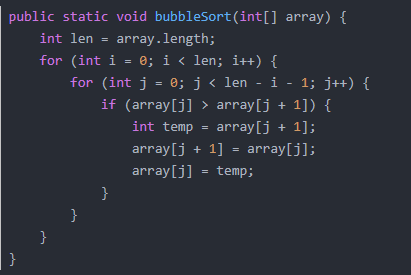
冒泡排序：

**算法原理：**

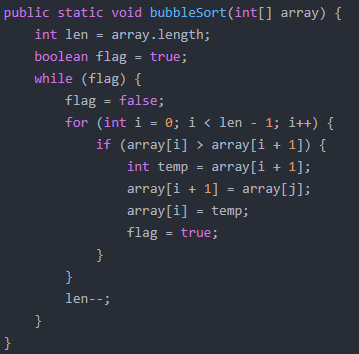
S1：从待排序序列的起始位置开始，从前往后依次比较各个位置和其后一位置的大小并执行S2。

S2：如果当前位置的值大于其后一位置的值，就把他俩的值交换（完成一次全序比较之后，最大值就在尾部了）

S3：将序列的最后位置从待排序序列中移除。若移除后的待排序序列不为空则继续执行S1，否则冒泡结束。

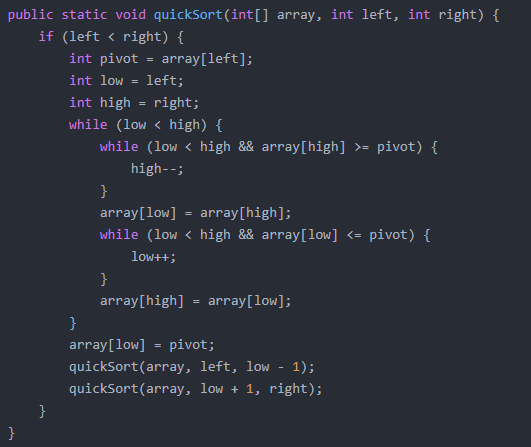


算法改进：若一趟排序之后，未发生任何交换，则排序结束



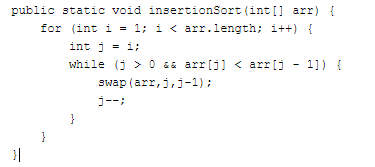
快速排序：

快排是对冒泡排序的改进，通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此实现整个数据变成有序序列。



插入排序：

每步将一个待排序序列按数据大小插到前面已经排序的序列中的适当位置，直到全部数据插入完毕为止



L

堆排序：

堆指的是二叉堆，二叉堆是一个完全二叉树

**二叉堆特性：**

1、父结点的键值总是大于或等于（小于或等于）任何一个子节点的键值。

2、每个结点的左子树和右子树都是一个二叉树。

**堆的存储结构**

一般用数组来表示堆，i 结点的父结点的下标为(i-1)/2 它的左右结点下标分别为2i+1和2i+2

**堆插入：**

将结点插入最底层最右边，然后依次上浮

**堆删除：**

将最底层最右侧的结点放到删除位置，然后依次下沉

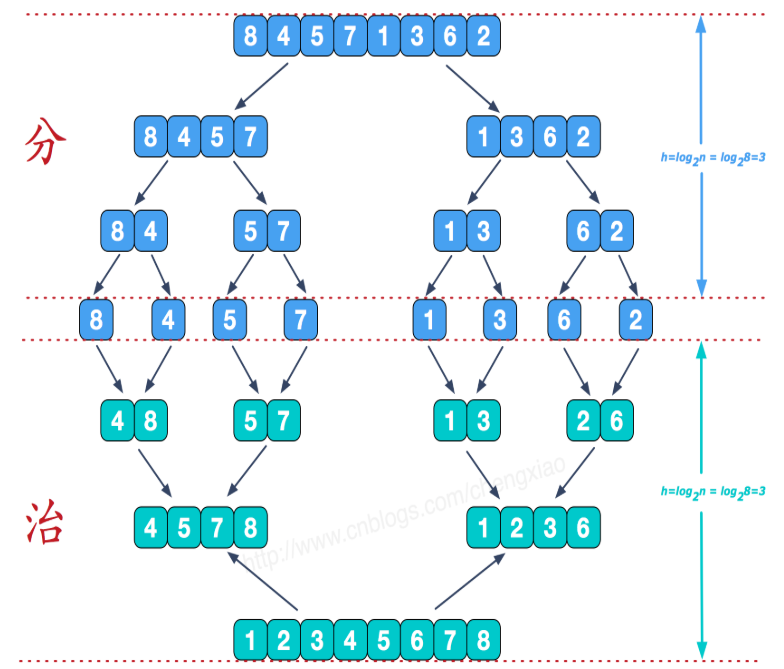
**堆排序**

创建堆，交换根结点与最底层最右边结点，然后输出最底层最右侧结点并删除该结点，然后重新调整堆

大堆根输出升序序列，小堆根输出降序序列

归并排序：

基本思想：分治思想



合并相邻有序子序列

