# 排序算法

http://www.codeceo.com/article/8-java-sort.html



## 直接插入排序

基本思想：

在要排序的数组中，假设前面(n-1)[n>=2]个数已经是排好顺序的，现在要把第n个元素插入到前面的有序元素中，使得这n个元素也是排好顺序的。如此反复循环，直到全部排好顺序。

步骤：

循环不变式：将i位置的元素插入到的前面已排序的数组A[0,… i-1]中，并使i++，保持循环不变式。

1）初始：1=2，A[1]已排序；

2）保持：在迭代开始前，A[1... i-1]已排序，而循环的目的是将A[i]插入A[1... i-1]中，使得A[1... i]排序。此时i++，并保持循环不变式；

3）终止：最后i=n+1，并且A[1... n]已排序，而A[1... n]就是整个数组，算法完成。

最佳运行时间（数组已排序）：O(n)，而快排O(n^2)

最坏运行时间（数组倒序）：O(n^2)

适用：少量元素的数组

实例：



代码实现：

insertSort(a) {

for(int i = 1; i < a.length; i++) {

int j = i-1;

int temp = a[i];

for(; j>=0&&temp<a[j]; j--) {

a[j+1] = a[j]; //将大于temp的值整体后移一位

}

a[j+1] = temp;

}

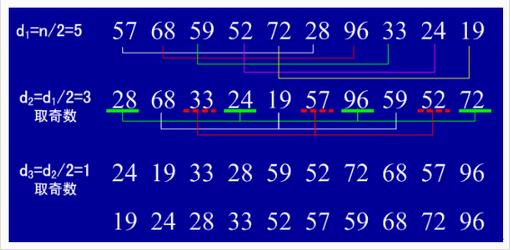
}

## 希尔排序（最小增量排序）

基本思想：

将要排序的数组按某个增量d（n/2，n为数组个数）分成若干组，每组中记录的下标相差d。对每组中全部元素进行直接查询排序，然后再用一个较小的增量（d/2）对它进行分组，在每组中再进行直接插入排序。当增量减到1时，进行插入排序后，排序完成。

实例：



代码实现：

shellSort(a) {

double d1 = a.length;

while(true) {

d1 = Math.ceil(d1/2);

int d = (int)d1;

for(int x = 0; x < d; x++) {

for(int i = x+d; i < a.length; i += d) {

int j = i-d;

int temp = a[i];

for(; j>=0&&temp<a[j]; j-=d) {

a[j+d] = a[j];

}

a[j+d] = temp;

}

}

if(d == 1)

break;

}

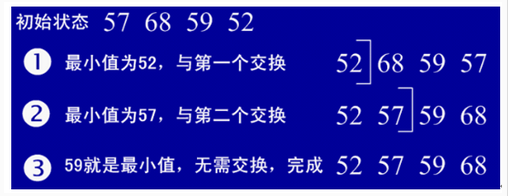
}

## 简单选择排序

基本思想：

在要排序的数组中，选出最小的一个元素与第一个位置的元素交换；然后在剩下的数中再找出最小的与第二个位置的数交换；一直重复循环，直到倒数第二个数和最后一个数比较为止。

实例：



源代码：

selectSort(a) {

int position = 0;

for(int i = 0; i < a.length; i++) {

int j = i+1;

position = i;

int temp = a[i];

for(; i < a.length; j++) {

if(a[j] < temp) {

temp = a[j];

position = j;

}

a[position] = a[i];

a[i] = temp;

}

}

}

## 堆排序

基本思想：

堆排序是一种树形选择排序，是对直接选择排序的有效改进。

堆的定义如下：

具有n个元素的序列（h1,h2,…,hn），当且仅当满足（hi>=h2i,hi>2i+1）或（hi<=h2i,hi<=2i+1）（i=1,2,…,n/2）时称之为堆。在这里只讨论满足前者条件的堆。由堆的定义可以看出，堆顶元素（即第一个元素）必为最大项（大顶堆）。完全二叉树可以很直观地表示堆的结构。堆顶为根，其它为左子树、右子树。初始时把要排序的数序列看作是一棵顺序存储的二叉树，调整它们的存储序，使之成为一个堆，这时堆的根节点的数最大。然后将根节点与堆的最后一个节点交换。然后对前面（n-1）个数重新调整使之成为堆。以此类推，直到只有两个节点的堆，并对它们做交换，最后得到n个节点的有序序列。从算法描述来看，堆排序需要两个过程。一是建立堆，二是堆顶与堆的最后一个元素交换

--未完成

## 冒泡排序

思想：通过两两交换，小的先冒出来，大的后冒出来。

最坏运行时间：O(n^2)

最佳运行时间：O(n^2)（当然也可以进行改进使得最佳运行时间为O(n)）

伪代码：

BubbleSort(A){

for i to n

for j to n-1

if(A[j] > a[j+1])

swap(a[j], a[j+1])

}

## 选择排序

思想：每次从未排序的元素中找到最小值。

最佳运行时间：O(n^2)

最坏运行时间：O(n^2)

算法：

循环不变式：数组A[0... i-1]已排序，从数组[i … n]中找到最小值放到i位置，i++，保持循环不变。

伪代码：

selectSort(A){

for i=0 to n

min = i;

for j = i+1 to n

if(A[min] > A[j])

min = j;

swap(A[min], A[i]);

}

## 快速排序

思想：挑一个“基准”，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面，该基准是它的最后位置。

最佳运行时间：O(nlgn)

最坏运行时间：O(n^2)

源代码：

public void quickSort(int[] num, int left, int right){

if(left < right){

int base = num[left]; //第一个基准值

int i=left, j=right;

do{

while(num[i]<base && i<right) i++;

while(num[j]>base && j>left) j--;

if(i <= j) swap(num[i], num[j])

}while(i <= j);

if(left < j) quickSort(num, left, j);

if(right > i) quickSort(num, i, right);

}

}

## 二分插入排序

思想：将右边未排序的元素插入到左边已排序的队列中

public void erfenSort(int[] num){

int left,right,mid,temp;

for(int i = 0; i < num.length; i++){

left = 0; right=i-1; temp = num[i];

while(left <= right){

mid = (left+right)/2;

if(num[mid] > temp) right = mid-1;

else left = mid+1;

}

for(int j = i+1; j > right; j--) num[j+1] = num[j];

num[right+1] = temp;

}

}

# 二叉查找树

## 满二叉树、完全二叉树、平衡二叉树

若设二叉树的深度为h，出第h层外，其它各层（1~h-1）的节点数都达到最大个数，第h层所有的节点都连续集中在最左边，这就是完全二叉树。

问题1：{1,2,3,4,5,6,7}按顺序构建一棵平衡二叉树的过程

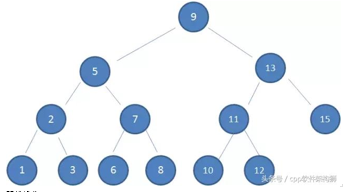
问题2：左旋和右旋怎么实现，过程是怎样的

## 特征

-1）值大小：左子树所有的节点 <= 根节点 <= 右子树所有的节点

-2）左右子树也一定分别为二叉查找树

图例：



查找操作：

--查找到值为10的节点步骤：

-1）查找到根节点9；

-2）由于10>9，所以查找到右孩子13；

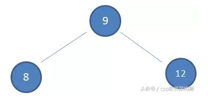
-3）由于10<13，所以查找到左孩子11；

-4）由于10<11，所以查找到左孩子10。

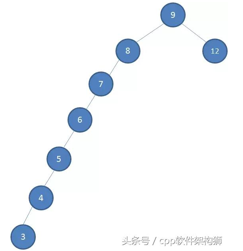
-5）由于正好发现是10，恰好是所寻找的值。

插入操作：

--1）如图，初始的二叉查找树只有三个节点



--2）依次插入5个节点：7，6，5，4，3后



缺点：如果根节点足够大，左子树会变得特别长。解决这种多次插入新节点而导致的不平衡？

解决：引出红黑树

# 红黑树

## 概念

红黑树是一种自平衡的二叉查找树，是一种高效的查找树。其具有良好的效率，它可在O(logN)时间内完成查找、增加、删除等操作。

这里的平衡指的是它不会变成“瘸子”，左腿特别长或者右腿特别长。

特征：

-1）节点是红色或黑色

-2）根节点是黑色

-3）每个叶子节点（NULL节点/空节点）是黑色

-4）每个红色节点的两个孩子节点必须是黑色。（从叶子到根的所有路径上不能有两个连续的红色节点）

-5）从任意节点到其叶子节点的所有路径都包含相同数目的黑色节点

正是因为这些规则，才能保证红黑树的自平衡。最长路径不超过最短路径的2倍。

## 旋转

当树的结构发生改变时（添加/删除元素），红黑树的五个特征可能会被打破，需要通过调整结构和颜色使树重新满足红黑树的特征。

调整分类：

-1）颜色调整：改变节点的颜色

-2）结构调整：左旋+右旋

### 左旋

左旋步骤：

-1）将旋转节点的右节点与旋转节点的右节点的左节点进行关联

-2）将旋转节点的父节点与旋转节点的右节点进行关联

-3）将旋转节点与旋转节点的右节点进行关联

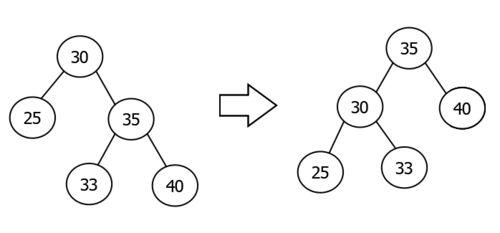
例如：

对节点30进行左旋的过程如下:

-1）将33与30的右节点进行关联

-2）将35的父节点指向30的父节点

-3）将30与35的左节点进行关联



参考TreeMap的左旋代码：

private void rotateLeft(Entry<K, V> p) {

if (p != null) {

// 获取p的右节点r，临时存储

Entry<K, V> r = p.right;

// 步骤1

// 1.1 p的右节点指向r的左节点

p.right = r.left;

// 1.2 r.left的父节点指向p

if (r.left != null)

r.left.parent = p;

// 步骤2

// 2.1 r的父节点指向p的父节点

r.parent = p.parent;

// 2.2 p的父节点为空，则根节点为r

if (p.parent == null)

root = r;

// 2.3 p的父节点不为空，判断p是父节点的左节点还是右节点

else if (p.parent.left == p) //p是父节点的左节点

p.parent.left = r;

lese //p是父节点的右节点

p.parent.right = r;

// 步骤3

r.left = p;

p.parent = r;

}

}

### 右旋

右旋步骤：

-1）将旋转节点的左节点的右节点关联到旋转节点的左节点上

-2）将旋转节点的父节点与旋转节点的左节点进行关联

-3）将旋转节点与旋转节点的左节点进行关联

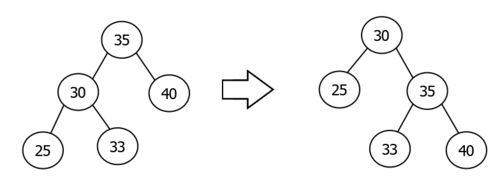
例如：

对节点35进行右旋的过程如下:

-1）将33与35的左节点进行关联

-2）将30的父节点指向35的父节点

-3）将35与30的右节点进行关联



参考TreeMap的右旋代码：

private void rotateRight(Entry<K, V> p) {

if (p != null) {

// 临时存储p的左节点

Entry<K, V> l = p.left;

// 步骤1

p.left = l.right;

if (l.right != null)

l.right.parent = p;

// 步骤2

l.parent = p.parent;

if (p.parent == null)

root = l;

else if (p.parent.right == p)

p.parent.right = l;

else

p.parent.left = l;

// 步骤3

l.right = p;

p.parent = l;

}

}

## 寻找节点的后继

当删除一个节点时，需要找一个后继节点/前驱节点，接替删除节点的位置。

参考TreeMap的寻找后继代码：

static <K, V> TreeMap.Entry<K, V> successor(Entry<K, V> t) {

if (t == null)

return null;

else if (t.right != null) { //右节点非空

// 循环寻找右节点的左节点的左节点...，直到左节点的左节点为空，返回

Entry<K, V> p = t.right;

while (p.left != null)

p = p.left;

return p;

} else { //右节点为空

Entry<K, V> p = t.parent; //父节点

Entry<K, V> ch = t; //当前节点

while (p != null && ch == p.right) { //当前节点是否是父节点的右节点

ch = p; //获取父节点的引用

p = p.parent; //父节点为祖父节点

}

// 如果当前节点不是父节点的右节点, 返回当前节点

return p;

}

}

当然TreeMap中还有寻找节点的前驱的方法: Entry<K,V> predecessor(Entry<K,V> t).

实际上前驱后继就是二叉树中序遍历时待删除节点的前驱后继.

## 插入？？？

插入时会出现以下四种情况：

-1）情况1：新节点（当前节点）为根节点

-2）情况2：新节点（当前节点）的父节点为黑色

-3）情况3：新节点（当前节点）的父节点为红色&叔叔节点为红色

-4）情况4：新节点（当前节点）的父节点为红色&叔叔节点为黑色

情况1：新节点（当前节点）为根节点

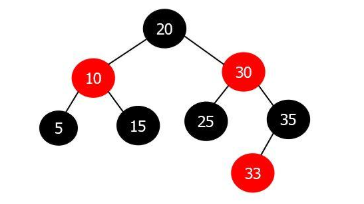
直接将新节点染成黑色即可。

情况2：新节点（当前节点）的父节点为黑色

父节点为黑色，添加一个红色孩子节点并不会影响红黑树的性质，不需要调整。

例如：

在一棵红黑树中插入节点33, 因为父节点是黑色, 所以不需要进行调整即可。



情况3：新节点（当前节点）的父节点为红色&叔叔节点为红色

祖父节点一定为黑色

处理步骤：将父节点和叔叔节点染为黑色，将祖父节点染为红色，将指针指向祖父节点

该情况与当前节点是父节点的左孩子还是右孩子无关，因为不涉及旋转。

这时新节点的颜色还是红色，可能出现四种情况：

-1）情况1：新节点为根节点

-2）情况2：新节点的父节点为黑色

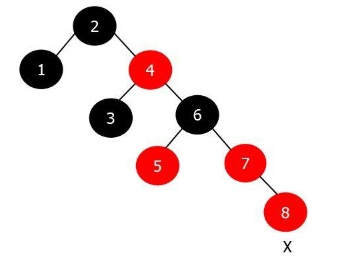
-3）情况3：新节点的叔叔节点为红色

-4）情况4：新节点的叔叔节点为黑色

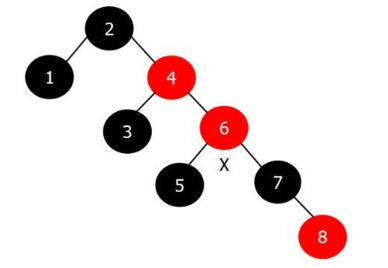
然后再进入对应情况的处理方案中处理。

例如：

在红黑树中插入节点8(X)，插入之后的红黑树如下：



很明显违反了红黑树的性质5，需要进行调整，按照情况3的处理步骤进行调整。



然后将当前节点(X)指向祖父节点，继续进行其它情况的调整。

情况4：新节点（当前节点）的父节点为红色&叔叔节点为黑色

处理步骤(父节点是祖父节点的左节点)：

-1）判断新节点（当前节点）是否是父节点的右孩子节点（将当前节点调整为父节点的左孩子节点）。是，则新节点（当前节点）指向父节点，然后对当前节点进行左旋；否，则不处理

-2）将父节点染为黑色

-3）将祖父节点染为红色

-4）对祖父节点进行右旋

处理步骤(父节点是祖父节点的右节点)：

-1）判断新节点（当前节点）是否是父节点的左孩子节点的（将当前节点调整为父节点的右孩子节点）。是，则新节点（当前节点）指向父节点，然后对当前节点进行右旋；否，则不处理

-2）将父节点染为黑色

-3）将祖父节点染为红色

-4）对祖父节点进行左旋

以当前节点是父节点的左节点为例，将当前节点指向父节点，然后对当前节点进行右旋，就变成当前节点是父节点的左孩子节点，并且叔叔节点是黑色。如果当前节点本就是父节点的左孩子节点，则不进行处理，直接进入步骤2。

这时新节点的颜色还是红色，兄弟节点的颜色为红色，父节点为黑色，可能出现四种情况：

-1）情况1：新节点（当前节点）为根节点

-2）情况2：新节点（当前节点）的父节点为黑色

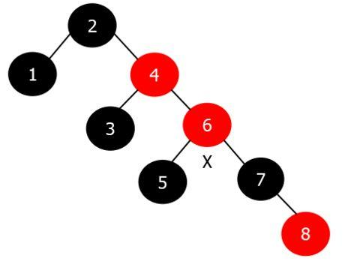
-3）情况3：新节点（当前节点）的叔叔节点为红色

-4）情况4：新节点（当前节点）的叔叔节点为黑色

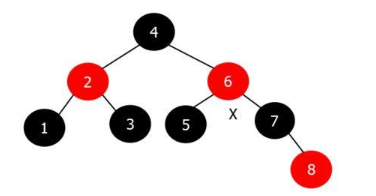
然后再进入对应情况的处理方案处理。

实例图：

继续调整情况3中的红黑树：



按照情况4进行调整之后如下：



调整完成。

### 插入总结

当新插入一个元素时，先按照二叉排序树的方法进行元素的插入，之后将新元素的颜色染为红色，然后对树进行调整，使其重新成为红黑树。

参考TreeMap的插入调整代码：

/\*\* from CLR \*/

private void fixAfterInsertion(Entry<K, V> x) {

// 默认新节点的颜色为红色

x.color = RED;

// 父节点为黑色时，增加一个红色节点并不会影响红黑树

while(x != null && x != root && x.parent.color == RED) {

// 父节点为祖父节点的左节点

if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {

//获取叔叔节点

Entry<K,V> y = rightOf(parentOf(parentOf(x)));

if (colorOf(y) == RED) { //叔叔节点为红色时

// 父节点和叔叔节点染为黑色

setColor(parentOf(x), BLACK);

setColor(y, BLACK);

// 祖父节点染为红色

setColor(parentOf(parentOf(x)), RED);

// 当前节点指向为祖父节点

x = parentOf(parentOf(x));

} else { // 叔叔节点为黑色

// 判断当前节点的左右

// 将当前节点调整为父节点的左节点，并进行左旋

if (x == rightOf(parentOf(x))) {

x = parentOf(x);

rotateLeft(x);

}

// 父节点染为黑色

setColor(parentOf(x), BLACK);

// 祖父节点染为红色，并进行右旋

setColor(parentOf(parentOf(x)), RED);

rotateRight(parentOf(parentOf(x)));

}

} else {

Entry<K,V> y = leftOf(parentOf(parentOf(x)));

if (colorOf(y) == RED) {

setColor(parentOf(x), BLACK);

setColor(y, BLACK);

setColor(parentOf(parentOf(x)), RED);

x = parentOf(parentOf(x));

} else {

if (x == leftOf(parentOf(x))) {

x = parentOf(x);

rotateRight(x);

}

setColor(parentOf(x), BLACK);

setColor(parentOf(parentOf(x)), RED);

rotateLeft(parentOf(parentOf(x)));

}

}

}

// 最后将根节点染为黑色?

root.color = BLACK;

}

## 删除

相对于插入, 红黑树的删除操作要复杂的多, 不过我们拆解分析, 就简单了, 把复杂问题拆解为小问题。

对于一颗红黑树，其删除节点的情况可以分为3种：

-1）情况1：节点既有左子树又有右子树

-2）情况2：节点只有左子树或右子树

-3）情况3：节点既没有左子树又没有右子树(叶子节点)

对于情况1，我们首先要找到该节点的前驱或后继节点，使用前驱或后继节点的值覆盖待删除节点的值，然后将前驱或后继节点按照情况2或情况3进行删除即可。前驱或者后继节点顶多有一个子节点。

所以, 对于红黑树来说, 实际删除节点的情况只有两种(情况2和情况3)。