数据结构之树

# 二叉查找树

二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：

1. 若左子树不空，则左子树上所有结点的值均小于或等于它的根结点的值；
2. 若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值；
3. 左、右子树也分别为二叉排序树；

**注**：treap tree(可能是最简单的一种二叉查找树)

# AVL树

## 定义

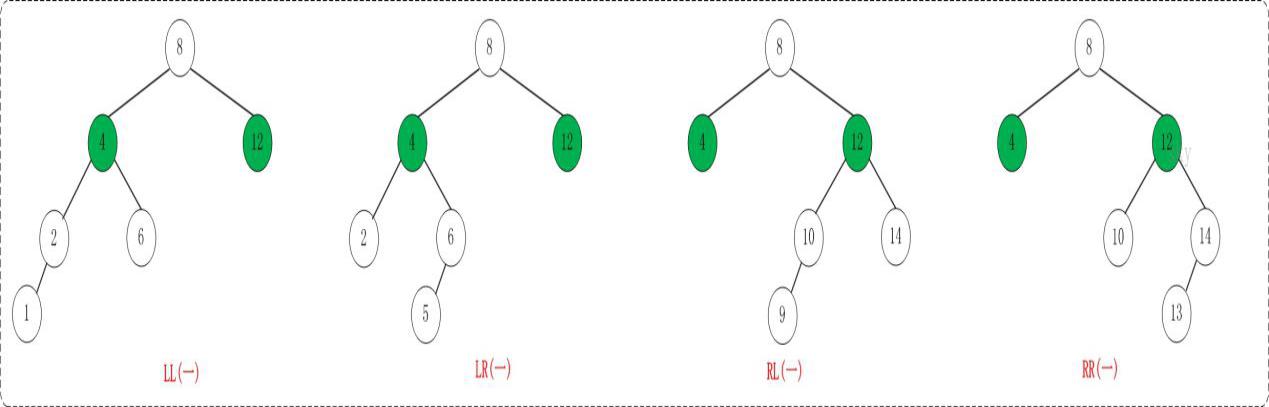
AVL树本质上还是一棵二叉搜索树，它的特点是：

1. 本身首先是一棵二叉搜索树；
2. 带有平衡条件：每个结点的左右子树的高度之差的绝对值（平衡因子）最多为1；

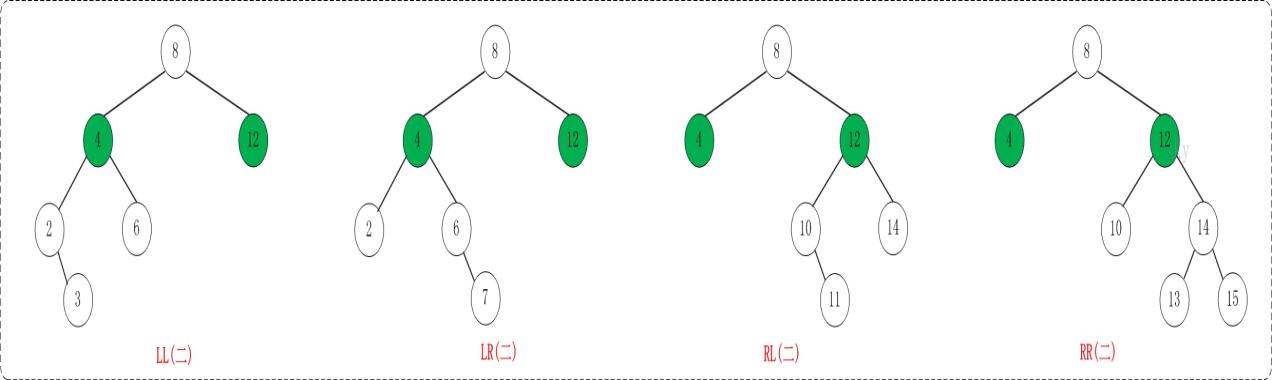
也就是说，AVL树，本质上是带了平衡功能的二叉查找树（二叉排序树，二叉搜索树）。

## 失衡

失衡图一：LL（右旋）、LR（先左旋后右旋）、RL（先右旋后左旋）、RR（左旋）



失衡图二：LL（右旋）、LR（先左旋后右旋）、RL（先右旋后左旋）、RR（左旋）



# 2-3-4树

1. 每个节点每个节点有1、2或3个key，分别称为2（孩子）节点，3（孩子）节点，4（孩子）节点。
2. 所有叶子节点到根节点的长度一致（也就是说叶子节点都在同一层）。
3. 每个节点的key从左到右保持了从小到大的顺序，两个key之间的子树中所有的key一定大于它的父节点的左key，小于父节点的右key。

**注**：着色的二叉查找树、AVL树的一个变种；红黑树和2-3-4树是完全等价的，由于绝大多数编程语言直接实现2-3-4树会非常繁琐，所以一般是通过实现红黑树来实现替代2-3-4树，而红黑树本也同样保证在O(lgn)的时间内完成查找、插入和删除操作。

# 红黑树

## 定义

红黑树是每个节点都带有颜色属性的二叉查找树，颜色或红或黑。在二叉查找树强制一般要求以外，增加了如下的额外要求：<http://www.cnblogs.com/nullzx/p/6128416.html>

1. 节点是红色或黑色；
2. 根节点是黑色；
3. 每个叶节点（NIL节点，空节点）是黑色的；
4. 每个红色节点的两个子节点都是黑色；
5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点；

**注：首先是个二叉查找树！**

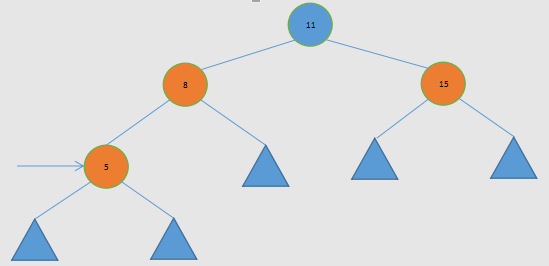
## 插入

### 父黑

直接插入

### 父红叔红

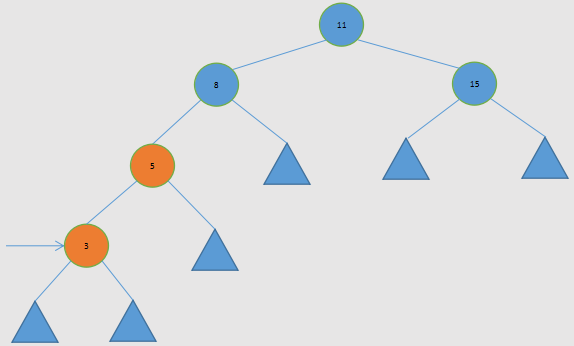
1. 将父节点设为黑色；
2. 将叔叔节点设为黑色；
3. 将祖父节点设为红色；
4. 将当**前节点指向祖父节点**继续循环判断是否平衡



### 父红叔黑

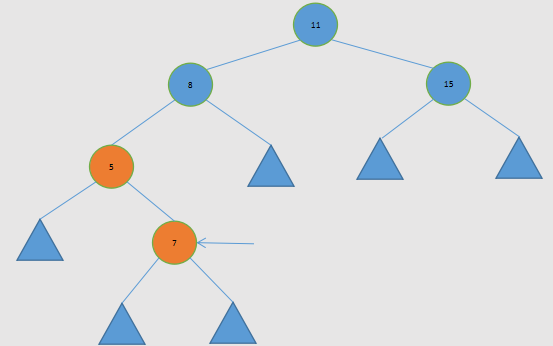
#### 父红叔黑-LL

1. 以祖父节点为支点进行右旋；
2. 将父节点置为黑色；
3. 将祖父节点置为红色；



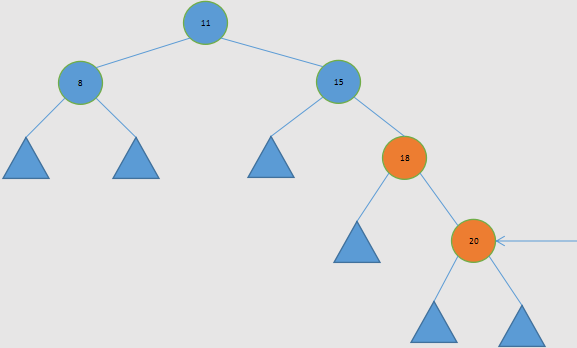
#### 父红叔黑-LR

1. 以父节点为支点进行左旋；
2. 以祖父节点为支点进行右旋；
3. 将当前节点置为黑色；
4. 将祖父节点置为红色；



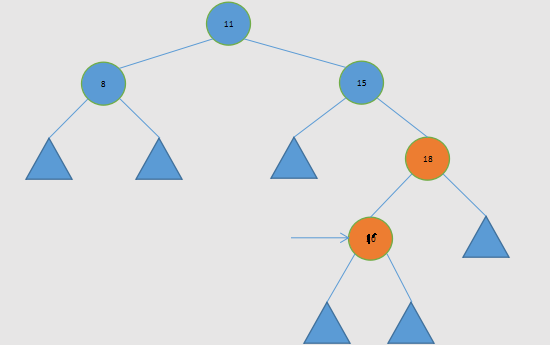
#### 父红叔黑-RR

1. 以祖父节点为支点进行左旋；
2. 将父节点置为黑色；
3. 将祖父节点置为红色；



#### 父红叔黑-RL

1. 以父节点为支点进行右旋；
2. 以祖父节点为支点进行左旋；
3. 将当前节点置为黑色；
4. 将祖父节点置为红色；



**注：调整完结束循环！**

### 调整根节点

1. 根节点置为黑色；
2. 旋转有可能导致根节点变更；

## 删除

### 当前节点有两个子节点

1. 遍历左右子树，分别找到左子树的最大节点和右子树的最小节点；**（节点肯定不会有两个子树）**
2. 若上述两个节点存在红色节点，则将此节点的值赋给要删除的节点，并删除此节点；
3. 若上述两个节点都非红色节点，则随机选一个节点，将此节点的值赋给要删除的节点，并删除此节点；

### 当前节点有且只有一个子节点

1. 将子节点的值赋值给当前节点；
2. 将子节点删除；

**注：子节点一定为红色，当前节点一定为黑色！**

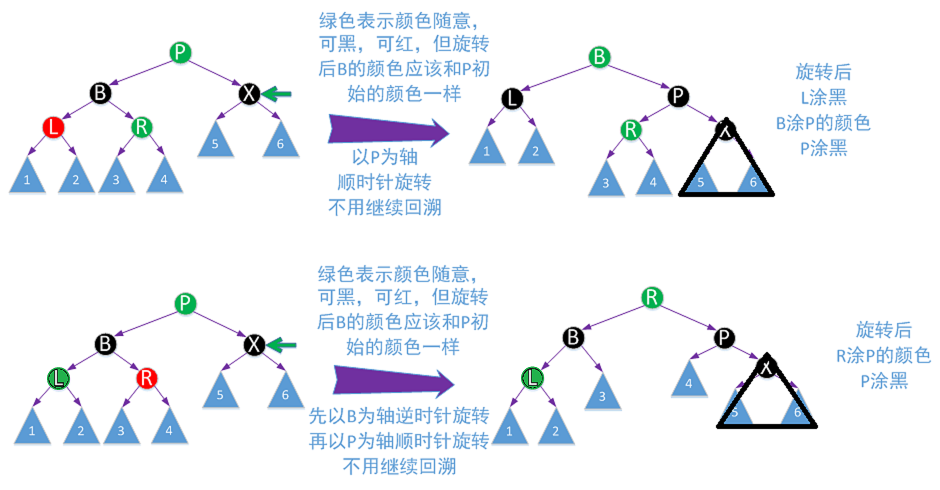
### 当前节点没有子节点

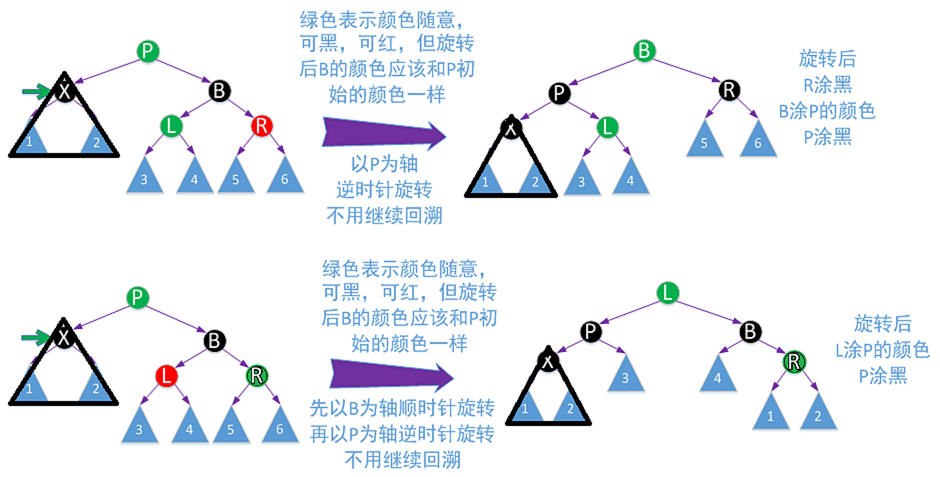
#### 当前节点为红色

直接删除即可

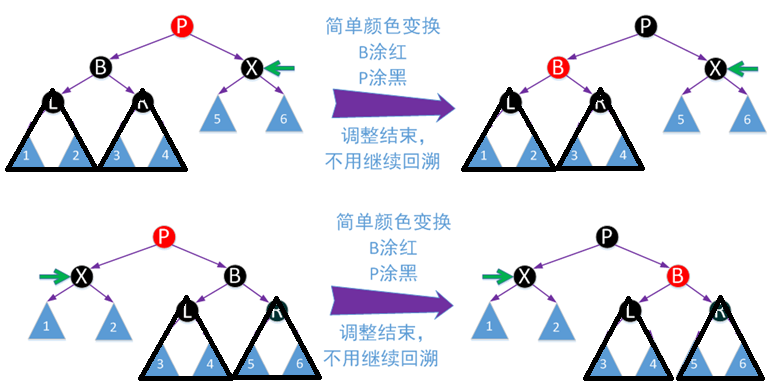
#### 当前节点为黑色

##### 黑兄红侄

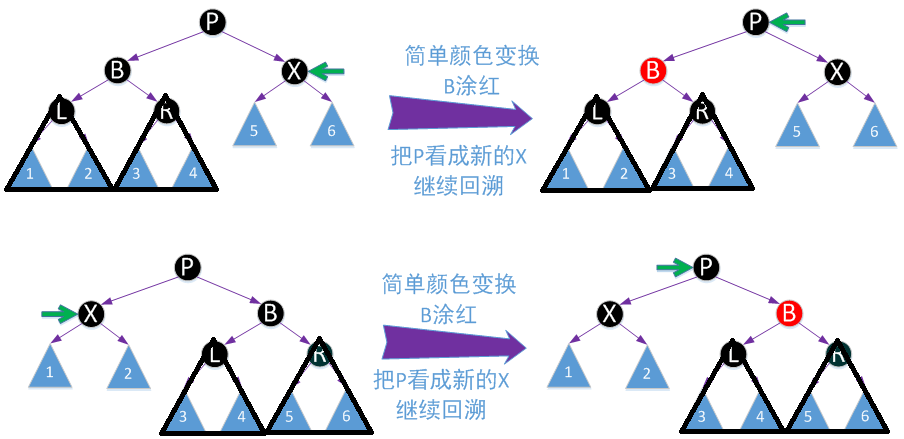




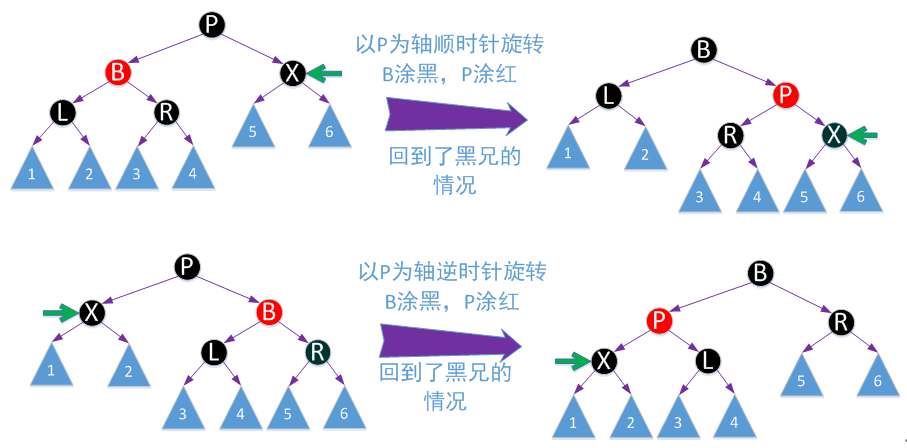
##### 黑兄黑侄红父



##### 黑兄黑侄黑父



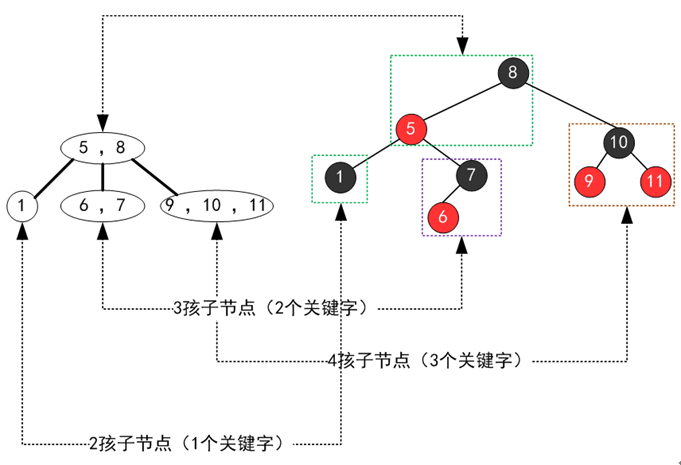
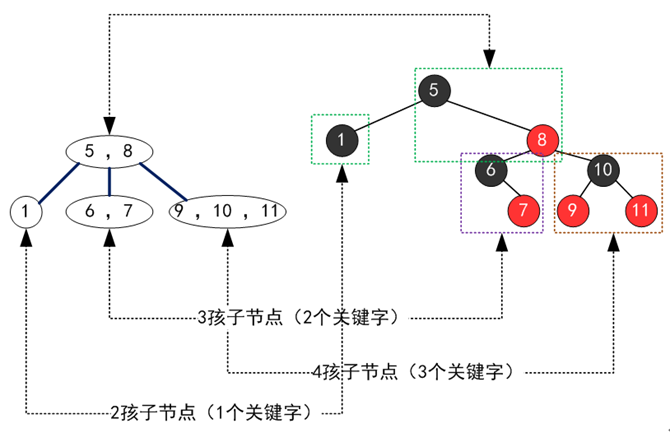
##### 红兄（黑侄黑父）



## 问答

**问：**红黑树和2-3-4树的关系？

**答：**2-3-4树和红黑树是完全等价的，由于绝大多数编程语言直接实现2-3-4树会非常繁琐，所以一般是通过实现红黑树来实现替代2-3-4树，而红黑树本也同样保证在O(lgn)的时间内完成查找、插入和删除操作。



**注：**

1. **把红结点收缩到其父结点，就变成了2-3-4树；**
2. **我们还需要明白的是，一棵红黑树对应唯一形态的2-3-4树，但是一颗2-3-4树可以对应多种形态的红黑树（主要是3节点可以对应两种不同的红黑树形态），如图。红黑树的删除操作中就会利用这种情况。**

**问：**增、删、查时间复杂度？

**答：**最坏的情况也是O(logn)。

**问：**构建时间复杂度？

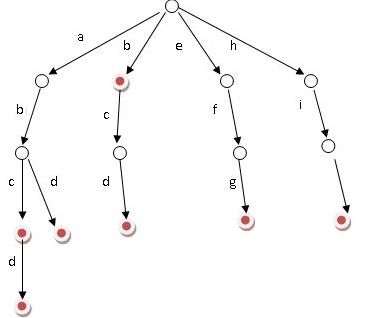
**答：**最坏的情况也是nO(logn)。

**问：**为什么要将新插入的节点着为红色？

**答：**依据性质4和性质5着为红色，可最大限度的保持平衡。

# SB 树（SizeBalancedTree）

# Trie树



## 基本性质

* 根节点不包含字符，除根节点外的每一个子节点都包含一个字符
* 从根节点到某一节点。路径上经过的字符连接起来，就是该节点对应的字符串
* 每个节点的所有子节点包含的字符都不相同

## 应用场景

典型应用是用于统计，排序和保存大量的字符串(不仅限于字符串)，经常被搜索引擎系统用于文本词频统计。

**注：利用字符串的公共前缀来减少查询时间，最大限度的减少无谓的字符串比较，查询效率高。以空间换时间。**

# B树 / B+树

b-tree/b+tree - mysql一种索引的数据结果，很复杂，能谈其性质和相关的特点即可。

**注**：特别想研究的时候在研究。

# 树型数组

## 二叉堆（最小堆和最大堆）

1. 由于堆是一棵完全二叉树，用数组实现即可（下标从0开始）；
2. 父节点下标[i]和子节点下标[jl,jr]的关系
   1. i = (j - 1) >>> 1
   2. jl = ((i + 1) << 1) - 1
   3. jr = (i + 1) << 1

### 最小堆

最小堆，是一种经过排序的完全二叉树，其中任一非终端节点的数据值均不大于其左子节点和右子节点的值。

1. 一棵完全二叉树
2. 父节点小于等于子节点

### 最大堆

最大堆，是一种经过排序的完全二叉树，其中任一非终端节点的数据值均不小于其左子节点和右子节点的值。

1. 一棵完全二叉树
2. 父节点大于等于子节点

## 线段树

## IndexTree