

# 关联式容器

关联式容器概念：每个元素都有一个“键（key）”和一个值（value）；

关联式容器底层实现：RB-tree 或 hash – table ;

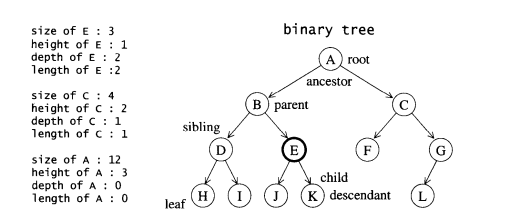
STL 标准的关联式容器：set（集合）及 map（映射表或字典）两大类；Set 的键值就是实值 ，map的键值和实值分开；

# 树的导览

树的应用：

* + 1. 操作系统将文件存放在树状结构里；
    2. 编译器实现表达式树(expression tree)
    3. 文件压缩使用哈夫曼算法（需要用到树—哈夫曼编码树）
    4. 数据库使用B-tree

STL使用的RB-tree是可提供良好的搜寻效率的树状结构；



树由节点（nodes）和边（edges）构成；

树的术语：

根节点：整棵树最上端的节点；

边：用来和其他节点相连；

父节点（parent）：

子节点（child）：

叶子节点（leaf）：

兄弟节点（sibling）：不同节点拥有相同父节点；

路径长度（length）：根节点至任何节点之间有唯一路径（path），路径所经过的边数即路径长度（length）；

节点深度（depth）：根节点至任意节点的路径长度，即该节点深度；根节点深度永远为0；

节点高度（height）：某节点至其最深子节点（叶节点）的路径长度。

树的高度：以根节点的高度代表；

前提：节点A->B之间如果存在唯一一条路径；

祖代：那么A称为B的祖代（ancestor）

子代：那么B称为A的子代（dascendant）

节点大小：其所有子代（包括自己）的节点总数；

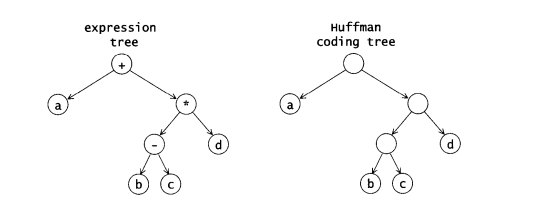
子节点可存在多个，如果最多只允许两个子节点，即所谓二叉树；

## 二叉搜索树（binary search tree）

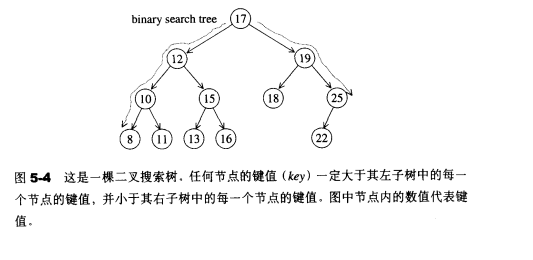
二叉树：任何节点最多只允许两个子节点；这两个子节点称为“左子节点”和“右子节点”；

以递归方式定义二叉树：“一个二叉树如果不为空，便是由一个根节点和左右两子树构成；左右子树都可能为空；”

二叉树运用：编译器表达式树和哈夫曼编码树；



**二叉搜索树：**可提供对数时间的元素插入和访问，任何节点的键值一定大于其左子树中的每一个节点的键值、并小于其右子树中的每一个节点的键值（**节点放置规则**）；



因此，从根节点一直往左走，直至无路可走，即得最小元素；从根节点一直往右走，直至无路可走，即得最大元素；

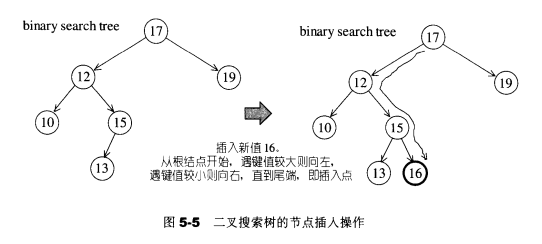
二叉搜索树：

查找最大最小元素及其简单（同上规则）：

插入和移除比较麻烦；

插入元素时：

可以从根节点开始，遇到键值较大者就向左，遇键值较小者就向右；

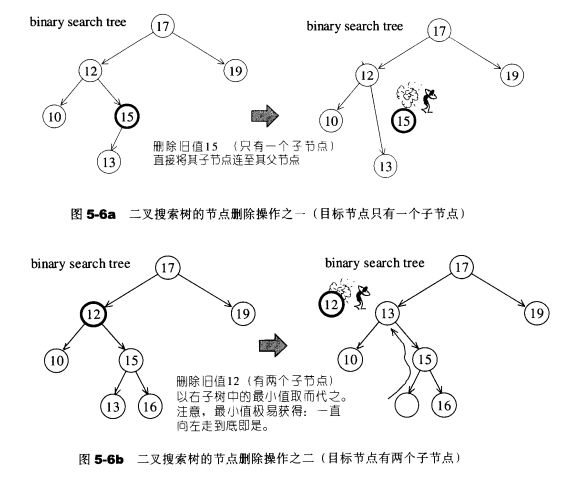


移除元素时：

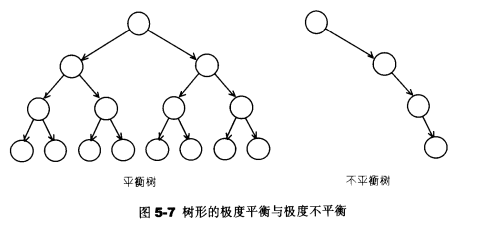
删除键值为12或15的节点；

情况分两种：

1. 15节点只有一个子节点，我们就直接将15的子节点连至15的父节点，并将15删除；
2. 12节点有两个子节点，我们就以右子树内最小节点（即13节点）取代12节点；右子树的最小节点极易获得：从12节点开始，一直向左走到底即可；



**二叉搜索树的弊端：**输入值不够随机或经过某些插入或删除操作；二叉搜索树可能会失去平衡，造成搜寻效率低落的情况；



## 平衡二叉搜索树（balanced binary search tree）

平衡二叉搜索树，所谓的树形平衡并没有一个绝对的测量标准，“平衡”的大致意义是：没有任何一个节点过深（深度过大）。

不同的平衡条件，造就出不同的效率表现，以及不同的实现复杂度；有数种特殊结构如：” AVL-tree / RB-tree / AA-tree ”，均可实现出平衡二叉搜索树 ； 它们可以避免极难应付的最坏（高度不平衡）情况，而且由于总是保持某种程度的平衡，所以元素访问时间平均而言也就比较少，一般而言其搜寻时间可节省25%左右；

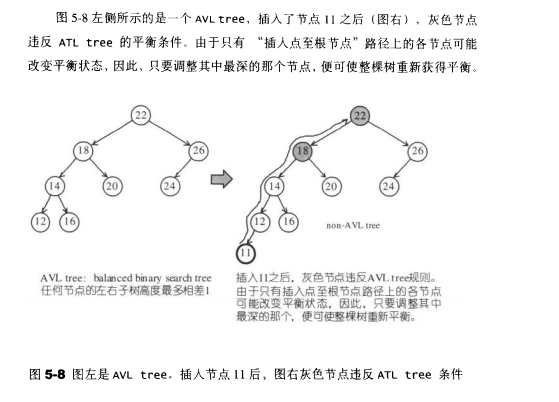
## AVL–tree

AVL-tree 是一个 “加上了额外平衡条件”的二叉搜索树；平衡条件确保了整棵树的深度（ O(logN) ）；

**额外平衡条件：**

最佳平衡条件是每个节点的左右子树有着相同的高度（过于苛刻）；

**AVL-tree退而求其次**：要求任何节点的左右子树高度相差最多为1；这是一个较弱的条件，但仍能保证“对数深度（logarithmic depth）”平衡状态；

****

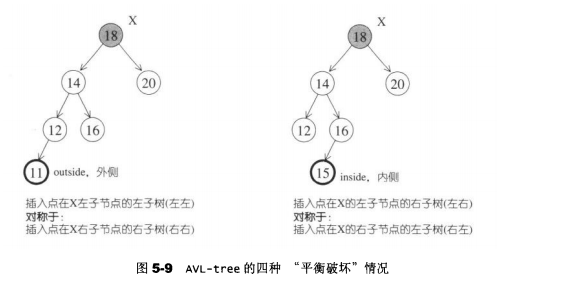
**平衡被破坏的4种情况：**

意味着X的左右两颗子树的高度相差2；

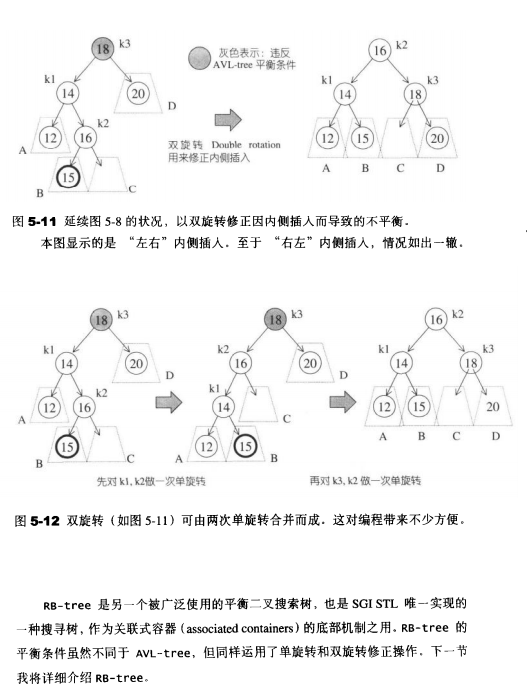
1. 插入点位于X的左子节点的左子树 --- 左左
2. 插入点位于X的左子节点的右子树 --- 左右
3. 插入点位于X的右子节点的左子树 --- 右左
4. 插入点位于X的右子节点的右子树 --- 右右

1） 、 4）称为外侧插入，采用**单旋转**操作解决 ；

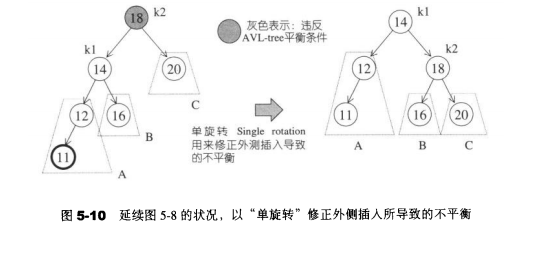
2） 、 3）称为内侧插入，采用**双旋转**操作解决 ；



## 双旋转（Double Rotation）



## 单旋转（Single Rotation）



左左情况：单右旋转；

K1向上提起，K2自然下滑，将B子树挂在K2左侧。

二叉搜索树规定：k1的右子树（B子树）必须位于k1与k2之间；

右右情况：单左旋转；

情况同上；

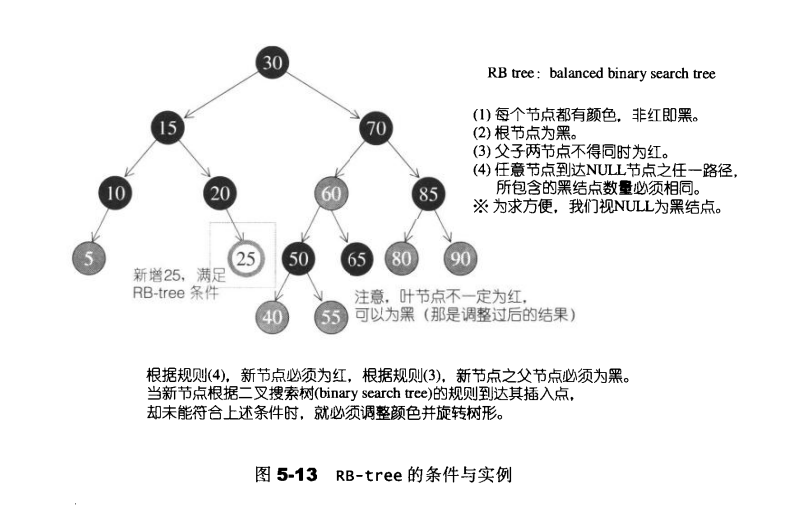
# RB-tree树（红黑树）

红黑树也是基于二叉搜索树规则的,只是额外多出了平衡条件；

红黑树实现：位于#include<stl\_tree.h>中

RB-tree树规则：

1. 每个节点不是红色就是黑色；（深色底纹：代表黑色，浅色底纹：代表红色）
2. 根节点为黑色；
3. 节点为红，子节点必须为黑（也就是父子节点不能同时为红）；
4. 任一节点至NULL（树尾端）的任何路径，所含之黑色节点数必须相同（也就是说插入的节点必须为红）；



## 插入节点

X：新增节点

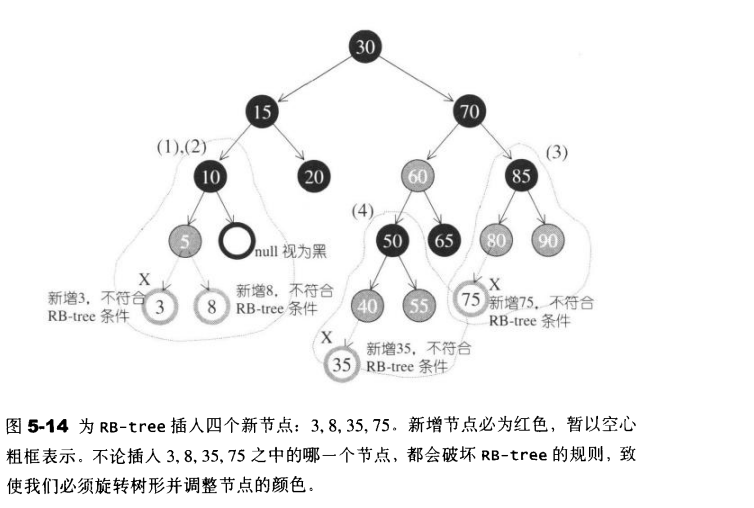
P：父节点·

S: 伯父节点（P的兄弟节点）

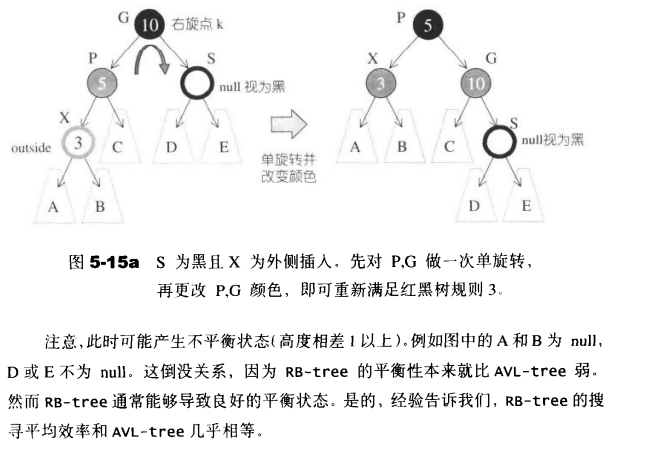
G: 祖父节点

GG：曾祖父节点

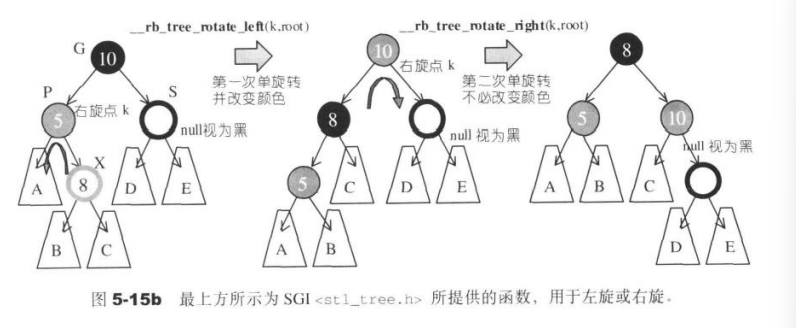
以为RB-tree插入3，8，35，75为例：



状况1：新增3节点（S为黑，X为外侧插入），外侧插入，对P、G做一次单旋转并改变颜色

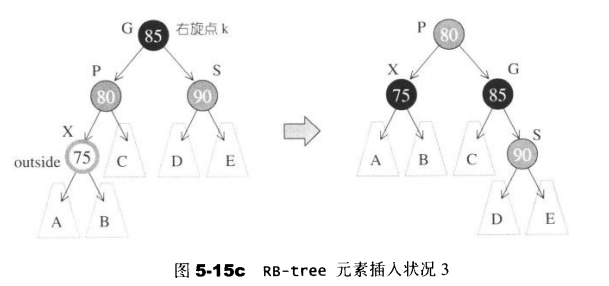


状况2：新增8节点（S为黑，X为内侧插入），内侧插入，对P、G做一次单旋转并改变颜色；对P,X做一次单旋转并改变G、X颜色，再将结果对G做一次旋转；

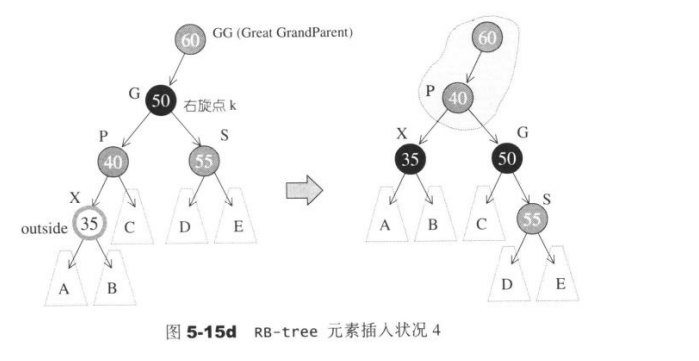


状况3：新增75节点（S为红 , X为外侧插入），对P和G做一次单旋转，并改变X颜色；

此时GG为黑一切搞定，但GG为红问题就大了，见状况4；



状况4：新增35节点（S为红 , X为外侧插入），对P和G做一次单旋转，并改变X颜色；此时GG如果为红，还得持续往上做，直到父子不在有连续为红的情况；



SGI-STL中红黑树为了”应付状况4”的情况；提供了一个由上而下的情况；