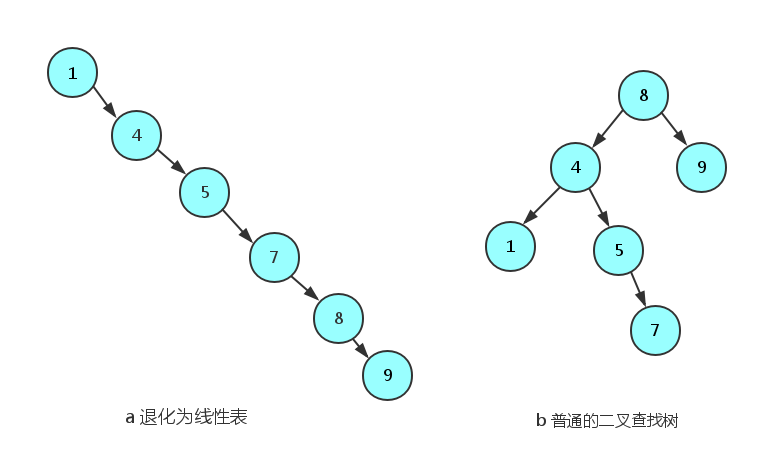
**二叉查找树**

**简介**

二叉查找树也称为有序二叉查找树,满足二叉查找树的一般性质,是指一棵空树具有如下性质:

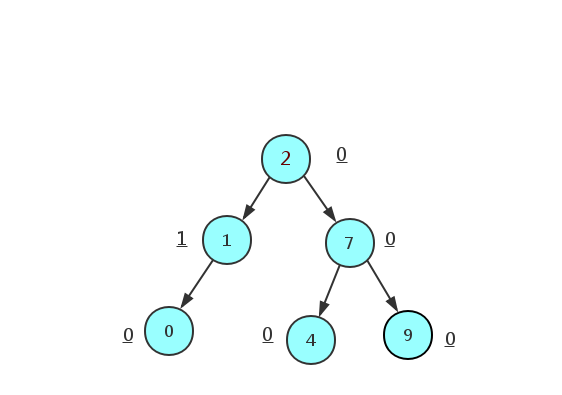
* **任意节点左子树不为空,则左子树的值均小于根节点的值.**
* **任意节点右子树不为空,则右子树的值均大于于根节点的值.**
* **任意节点的左右子树也分别是二叉查找树.**
* **没有键值相等的节点.**

**局限性及应用**

一个二叉查找树是由n个节点**随机构成**,所以，对于某些情况,**二叉查找树会退化成一个有n个节点的线性链.如下图:**   
b图为一个普通的二叉查找树,大家看a图,如果我们的根节点选择是最小或者最大的数,那么二叉查找树就完全退化成了线性结构,因此,在**二叉查找树的基础上,又出现了AVL树,红黑树**,它们两个都是基于二叉查找树,只是在二叉查找树的基础上又对其做了限制.

**AVL树**

**简介**

AVL树是带有平衡条件的二叉查找树,一般是用平衡因子差值判断是否平衡并通过旋转来实现平衡,**左右子树树高不超过1,和红黑树相比,它是严格的平衡二叉树,平衡条件必须满足(所有节点的左右子树高度差不超过1).**不管我们是执行插入还是删除操作,只要不满足上面的条件,就要**通过旋转来保持平衡,而旋转是非常耗时的,由此我们可以知道AVL树适合用于插入删除次数比较少，但查找多的情况。**   
从上面这张图我们可以看出,**任意节点的左右子树的平衡因子差值都不会大于1.**

**局限性**

由于**维护这种高度平衡所付出的代价比从中获得的效率收益还大,故而实际的应用不多，**更多的地方是用**追求局部而不是非常严格整体平衡的红黑树.**当然,如果应用场景中对插入删除不频繁,只是对查找要求较高,那么AVL还是较优于红黑树.

**应用**

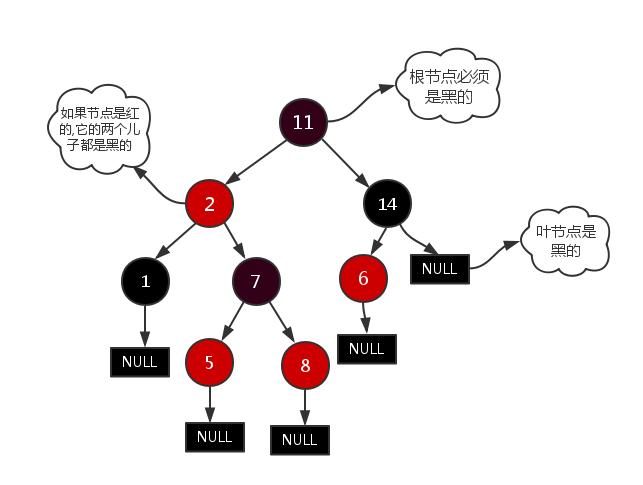
**Windows NT内核中广泛存在.**

**红黑树**

**简介**

一种二叉查找树,但在每个节点增加一个存储位表示节点的颜色,可以是**red或black.** 通过对任何一条从根到叶子的路径上各个节点着色的方式的限制,红黑树确保没有一条路径会比其它路径长出两倍.它是一种**弱平衡二叉树**(**由于是若平衡,可以推出,相同的节点情况下,AVL树的高度低于红黑树**),相对于要求严格的AVL树来说,它的旋转次数变少,所以对于搜索,插入,删除操作多的情况下,我们就用红黑树.

**性质**

* **每个节点非红即黑.**
* **根节点是黑的。**
* **每个叶节点(叶节点即树尾端NUL指针或NULL节点)都是黑的.**
* **如果一个节点是红的,那么它的两儿子都是黑的.**
* **对于任意节点而言,其到叶子点树NIL指针的每条路径都包含相同数目的黑节点.**

**应用**

* 广泛用于C++的STL中,map和set都是用红黑树实现的.
* 著名的linux进程调度[Completely Fair Scheduler](https://en.wikipedia.org/wiki/Completely_Fair_Scheduler" \t "_blank),用红黑树管理进程控制块,进程的虚拟内存区域都存储在一颗红黑树上,每个虚拟地址区域都对应红黑树的一个节点,左指针指向相邻的地址虚拟存储区域,右指针指向相邻的高地址虚拟地址空间.
* **IO多路复用epoll的实现采用红黑树组织管理sockfd，以支持快速的增删改查.**
* **ngnix中,用红黑树管理timer,因为红黑树是有序的,可以很快的得到距离当前最小的定时器.**
* **java中TreeMap的实现.**

红黑树的查询性能略微逊色于AVL树，但是，红黑树在插入和删除上完爆avl树，avl树每次插入删除会进行大量的平衡度计算，而红黑树为了维持红黑性质所做的红黑变换和旋转的开销，相较于avl树为了维持平衡的开销要小得多.

最多旋转3次。

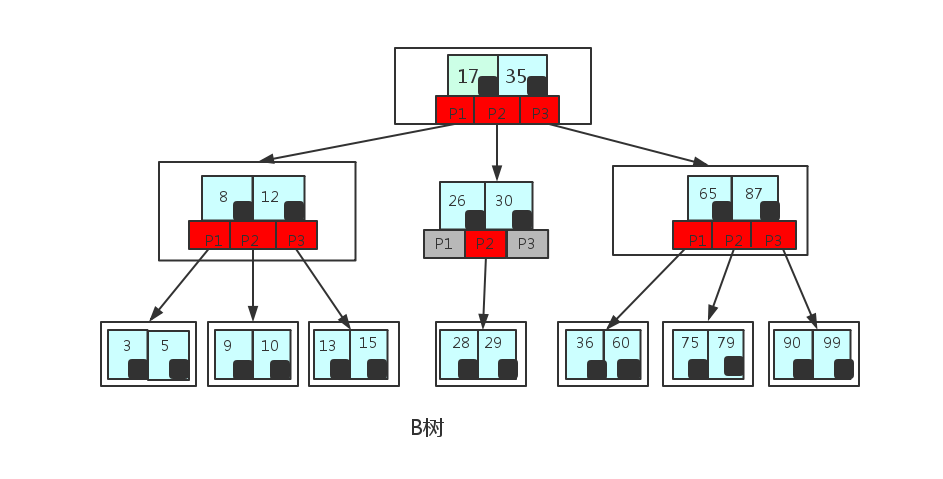
**B/B+树**

注意B-树就是B树,-只是一个符号.

**简介**

B/B+树是为了磁盘或其它存储设备而设计的一种**平衡多路查找树**(相对于二叉,B树每个内节点有多个分支),与红黑树相比,在相同的的节点的情况下,**一颗B/B+树的高度远远小于红黑树的高度(在下面B/B+树的性能分析中会提到).**B/B+树上操作的时间通常由存取磁盘的时间和CPU计算时间这两部分构成,而CPU的速度非常快,所以**B树的操作效率取决于访问磁盘的次数**,**关键字总数相同的情况下B树的高度越小，**磁盘I/O所花的时间越少.

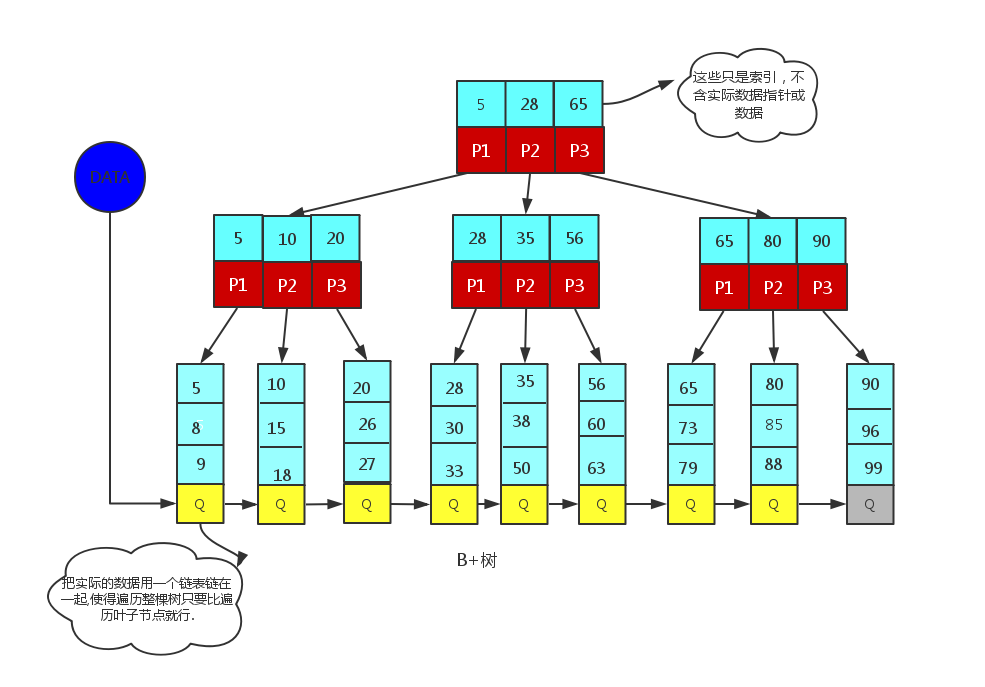
**B树的性质**

* 定义任意非叶子结点最多只有M个儿子；**且M>2；**
* 根结点的儿子数为[2, M]；
* 除根结点以外的非叶子结点的儿子数为[M/2, M]；
* 每个结点存放至少M/2-1（取上整）和至多M-1个关键字；（至少2个关键字）
* 非叶子结点的关键字个数=指向儿子的指针个数-1；
* 非叶子结点的关键字：K[1], K[2], …, K[M-1]；且K[i] < K[i+1]；
* 非叶子结点的指针：P[1], P[2], …, P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指向关键字大于K[M-1]的子树，其它P[i]指向关键字属于(K[i-1], K[i])的子树；
* **所有叶子结点位于同一层；**  
    
  这里只是一个简单的B树,在实际中B树节点中关键字很多的.上面的图中比如35节点,**35代表一个key(索引)，而小黑块代表的是这个key所指向的内容在内存中实际的存储位置.是一个指针.（多了B+tree这个东西）**

**B+树**

B+树是应文件系统所需而产生的一种B树的变形树**(文件的目录一级一级索引,只有最底层的叶子节点(文件)保存数据.),**非叶子节点**只保存索引,不保存实际的数据,**数据都保存在叶子节点中.这不就是文件系统文件的查找吗?我们就举个文件查找的例子:**有3个文件夹,a,b,c, a包含b,b包含c,**一个文件yang.c, a,b,c就是索引(存储在非叶子节点), a,b,c**只是要找到的yang.c的key,而实际的数据yang.c存储在叶子节点上.   
所有的非叶子节点都可以看成索引部分**

**B+树的性质(下面提到的都是和B树不相同的性质)**

* **非叶子节点的子树指针与关键字个数相同;**
* **非叶子节点的子树指针p[i],指向关键字值属于[k[i],k[i+1]]的子树.(B树是开区间,也就是说B树不允许关键字重复,B+树允许重复**)；
* **为所有叶子节点增加一个链指针.**
* **所有关键字都在叶子节点出现(稠密索引). (且链表中的关键字恰好是有序的);**
* **非叶子节点相当于是叶子节点的索引(稀疏索引),叶子节点相当于是存储(关键字)数据的数据层.**
* 更适合于文件系统;   
  看下图:   
    
   **非叶子节点(比如5,28,65)只是一个key(索引),实际的数据存在叶子节点上(5,8,9)才是真正的数据或指向真实数据的指针.**

**应用**

B和B+树主要用在文件系统以及数据库做索引.比如Mysql;

**B/B+树性能分析**

* n个节点的平衡二叉树的高度为H(即logn),而n个节点的B/B+树的高度为logt((n+1)/2)+1;
* 若要作为内存中的查找表,B树却不一定比平衡二叉树好,尤其当m较大时更是如此.因为查找操作CPU的时间在B-树上是O(mlogtn)=O(lgn(m/lgt)),而m/lgt>1;所以m较大时O(mlogtn)比平衡二叉树的操作时间大得多. 因此在内存中使用B树必须取较小的m.（通常取最小值m=3，此时B-树中每个内部结点可以有2或3个孩子，这种3阶的B-树称为2-3树）。

**为什么说B+tree比B树更适合实际应用中操作系统的文件索引和数据索引.**

* **B+-tree的内部节点并没有指向关键字具体信息的指针,因此其内部节点相对B树更小,**如果把所有同一内部节点的关键字存放在同一盘块中,那么盘块所能容纳的关键字数量也越多,一次性读入内存的需要查找的关键字也就越多,相对IO读写次数就降低了.
* 由于**非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引**。**所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率一样。**ps:我在知乎上看到有人是这样说的,我感觉说的也挺有道理的:   
  他们认为数据库索引采用B+树的主要原因是**:B树在提高了IO性能的同时并没有解决元素遍历的我效率低下的问题,正是为了解决这个问题,B+树应用而生.B+树只需要去遍历叶子节点就可以实现整棵树的遍历.**而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作（或者说效率太低）.