SQLite中B-tree、B+tree初步探秘（欢迎指正，共同进步）

最近半年实验室一直在fedora下用Qt做ARM平台的火灾自动报警方面的开发，用的是SQLite数据库。作为一个嵌入式的数据库，确实有好多过人之处，个人蛮喜欢。于是找来

《The Definitive Guide to SQLite》深入探究一下，这本书1/3将怎么契合SQL使用，1/3讲C API接口实现，剩下的一直在讲述SQLite内部实现机制。从前到后，一直强调数据库文件格式：表用B-tree，索引用B+tree，本科数据结构（严蔚敏版）讲到B tree，但是只是理论说教，屁用没有。不废话了，直入主题

在进一步探究SQLite之前，先预热一下，说说B tree、B-tree和B+tree。

参考：http://blog.csdn.net/daliaojie/article/details/8662312

（1）B tree

即二叉搜索树：

1.所有非叶子结点至多拥有两个儿子（Left和Right）；

2.所有结点存储一个关键字；

3.非叶子结点的左指针指向小于其关键字的子树，右指针指向大于其关键字的子树；

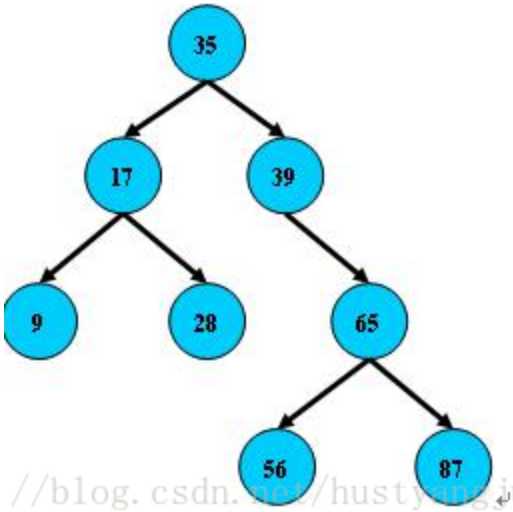
B树的搜索，从根结点开始，如果查询的关键字与结点的关键字相等，那么就命中；

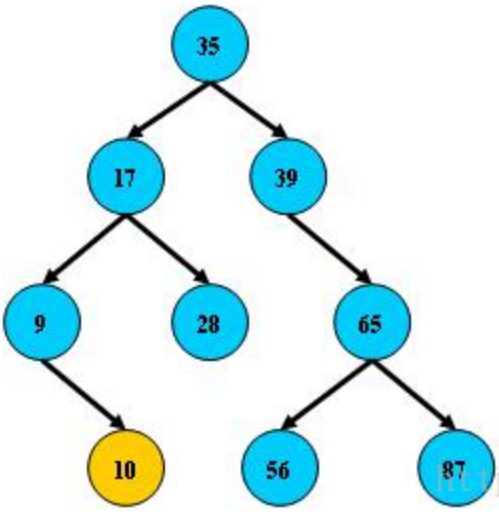
否则，如果查询关键字比结点关键字小，就进入左儿子；如果比结点关键字大，就进入

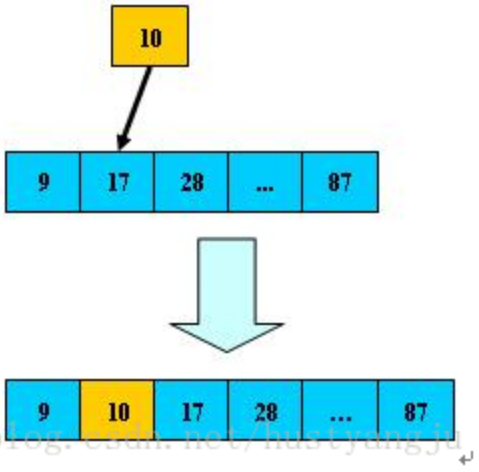
右儿子；如果左儿子或右儿子的指针为空，则报告找不到相应的关键字；

如果B树的所有非叶子结点的左右子树的结点数目均保持差不多（平衡），那么B树的搜索性能逼近二分查找；但它比连续内存空间的二分查找的优点是，改变B树结构（插入与删除结点）不需要移动大段的内存数据，甚至通常是常数开销；

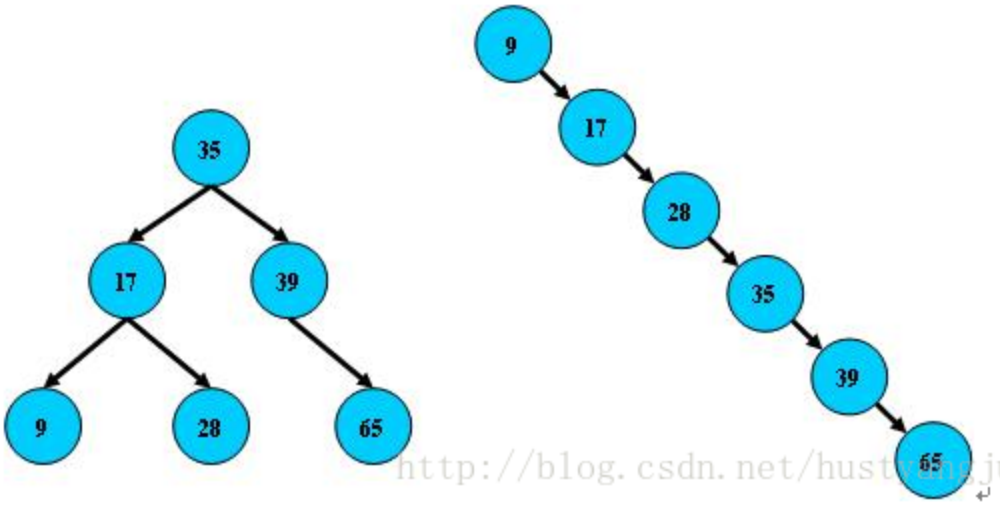
如：







但B树在经过多次插入与删除后，有可能导致不同的结构：



右边也是一个B树，但它的搜索性能已经是线性的了；同样的关键字集合有可能导致不同的

树结构索引；所以，使用B树还要考虑尽可能让B树保持左图的结构，和避免右图的结构，也就是所谓的“平衡”问题；

实际使用的B树都是在原B树的基础上加上平衡算法，即“平衡二叉树”（binary balanced tree，又称AVL树）；如何保持B树结点分布均匀的平衡算法是平衡二叉树的关键；平衡算法是一种在B树中插入和删除结点的策略；

（2）B-tree

是一种多路搜索树（并不是二叉的）：

1.定义任意非叶子结点最多只有M个儿子；且M>2；

2.根结点的儿子数为[2, M]；

3.除根结点以外的非叶子结点的儿子数为[M/2, M]；

4.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）

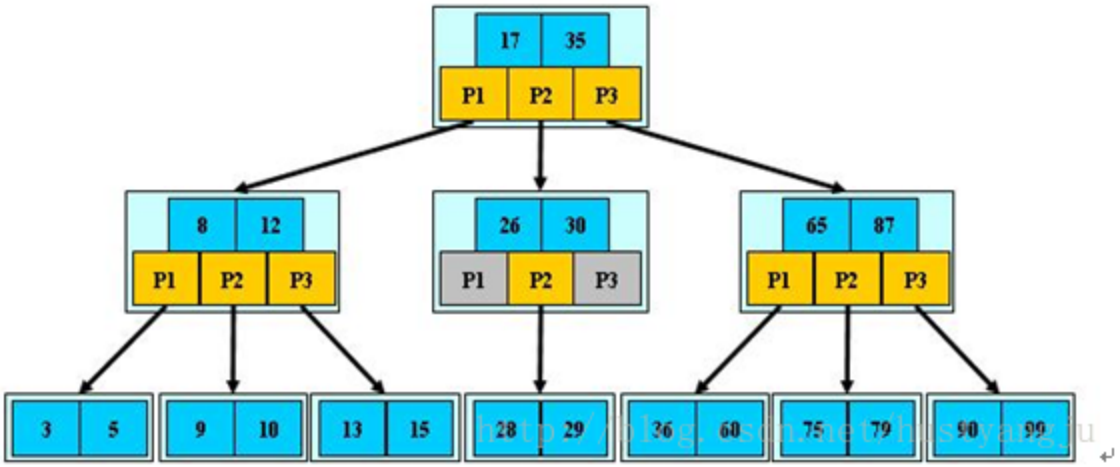
5.非叶子结点的关键字个数=指向儿子的指针个数-1；

6.非叶子结点的关键字：K[1], K[2], …, K[M-1]；且K[i] < K[i+1]；

7.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；

8.所有叶子结点位于同一层；



B-树的搜索，从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为空，或已经是叶子结点；

B-树的特性：

1.关键字集合分布在整颗树中；

2.任何一个关键字出现且只出现在一个结点中；

3.搜索有可能在非叶子结点结束；

4.其搜索性能等价于在关键字全集内做一次二分查找；

5.自动层次控制；

由于限制了除根结点以外的非叶子结点，至少含有M/2个儿子，确保了结点的至少利用率，其最底搜索性能为：

$$
\begin{aligned}
&O_{Min} \\
&= O[\log_2(\lceil \tfrac{M}{2} - 1 \rceil) \times \log_{\frac{M}{2}}(\tfrac{N}{\lceil \frac{M}{2}-1 \rceil})] \\
&= O[\log_2(\tfrac{M}{2})] \times O[\log_{\frac{M}{2}}(\tfrac{N}{\frac{M}{2}})] \\
&= O[\log_2(\tfrac{M}{2}) \times (\log_{\frac{M}{2}} N - 1)] \\
&= O[\log_2 N - \log_2(\tfrac{M}{2})] \\
&= O[\log_2 N] - O[C] \\
&= O[\log_2 N]
\end{aligned}
$$

其中，M为设定的非叶子结点最多子树个数，N为关键字总数；

所以B-树的性能总是等价于二分查找（与M值无关），也就没有B树平衡的问题；

由于M/2的限制，在插入结点时，如果结点已满，需要将结点分裂为两个各占M/2的结点；删除结点时，需将两个不足M/2的兄弟结点合并；

（3）B+tree

B+树是B-树的变体，也是一种多路搜索树：

1.其定义基本与B-树同，除了：

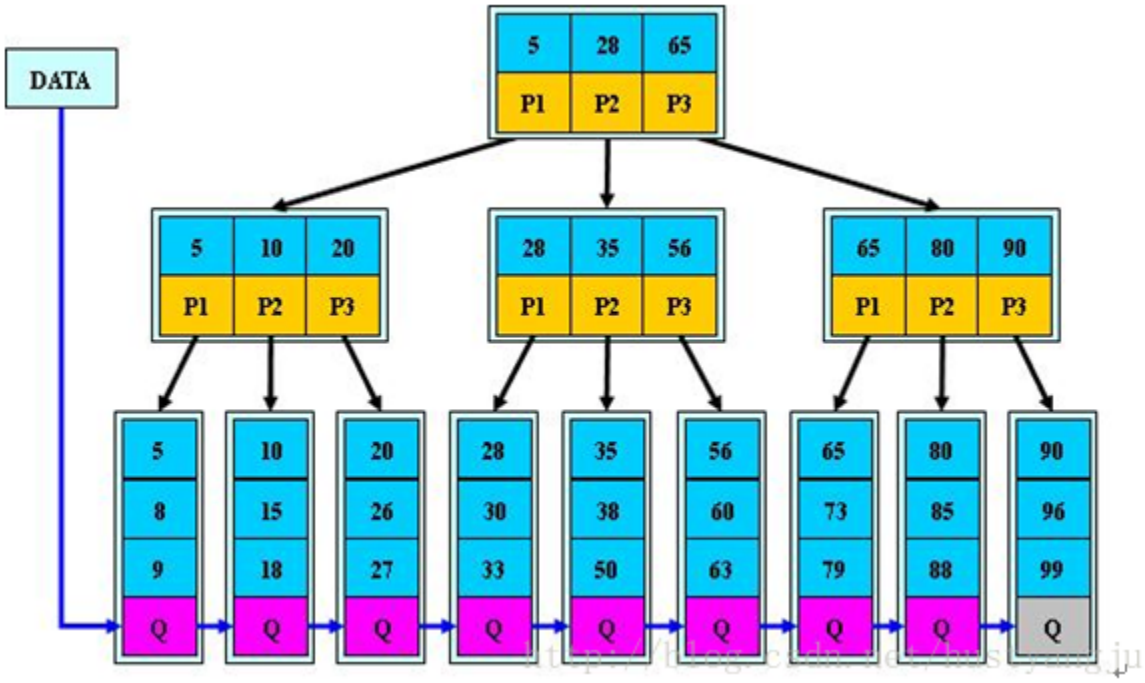
2.非叶子结点的子树指针与关键字个数相同；

3.非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；

5.为所有叶子结点增加一个链指针；

6.所有关键字都在叶子结点出现；

如：（M=3）



B+的搜索与B-树也基本相同，区别是B+树只有达到叶子结点才命中（B-树可以在非叶子结点命中），其性能也等价于在关键字全集做一次二分查找；

B+的特性：

1.所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；

2.不可能在非叶子结点命中；

3.非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；

4.更适合文件索引系统；

进入主题：

由于操作系统内存按照分页机制管理，大一点数据库文件不可能全部存放在内存中，只能用一部风从硬盘中调取一部分放到内存中。所以数据库的存储和索引也是分页的，即paper。

在SQLite中，每一个表（含多字段）都用一个唯一的B-tree存储，数据库有多个表就有多个B-tree。前面说到SQLite数据库是分页存储的，对，SQLite把一个paper当做B-tree的一个结点（包含根结点、中间结点和叶子结点），中间结点和叶子结点在B-tree中不作区分，一个表的

根结点最特殊：根结点是一张表的第一个paper分页，其前100 Byte包含了用以说明库版本、模式版本、页大小、编码方式和是否启用自动清理等信息的头文件（该头文件在btree.c中定义）。

《The Definitive Guide to SQLite》中提到，B-tree中的paper（即B-tree结点）由一系列的b-treepayload（有效载荷）组成。怎么理解这个b-tree payload呢，我的理解是：一个paper中包含至少M/2-1（取上整）和至多M-1个关键字（至少2个关键字），B-tree在建立时首先对关键字排序，具体规则是在新建表时约束的，每一个关键字（即ROWID或主键）都用一个B-tree来存储关键字相同的数据。也就是说，这里的b-tree payload 也是b-tree，只不过它的键值域是相同的，而数值域是可以存放任意格式的数据，在SQLite中有一种针对所有存储格式数据排序的算法（中文版《The Definitive Guide to SQLite》116页），这样b-treepayload的数值域也可按照一定算法构成b-tree。

综上，每一个paper多个关键字就对应多个b-tree payload；paper大小事固定的，而b-tree payload是受数据域大小决定的，所以会有页面溢出的问题出现。针对这种情况，B-tree会根据需要创建一到多个溢出页。

可以理解SQLite B-tree机制了吧，(*^__^*) 嘻嘻……

对于B+tree，简单一些：

SQLite是根据关键值建立索引表的，在创建时会按照一定顺序排序。B+tree的根结点、中间结点和叶子结点也和B-tree一样，对应一个paper，只不过根结点和中间结点的键值域已经排好序，而且数值域不再存储数据啦，而是存储指向下一层paper的指针。这样就会出现这种情况：较早出现的关键字会按照指针的指向一直到达叶子结点，叶子结点的关键字也是排好序的。最后，所有数据都保存在叶子结点的数据域。叶子结点的数据域中的数据由VDBE控制，采用一种特殊的格式记录数据：

VDBE

逻辑头段　　数据段

hsize	T1	T2	T3	TN		D1	D2	D3	DN

hsize： 记录逻辑头段大小，一般为64 bit的整数倍的整数

T1、T2...：描述对应D1、D2...的存储类型和长度（类型+长度可以确定D1、D2...的内容啦），一般为64bit整数倍大小的数组。

注意：书上画的图显示，叶子结点的每一个关键字就对应一个VDBE，有一点疑问：怎么处理类似页溢出的问题呢？