查找（二）

散列表

散列表是普通数组概念的推广。由于对普通数组可以直接寻址，使得能在O(1)时间内访问数组中的任意位置。在散列表中，不是直接把关键字作为数组的下标，而是根据关键字计算出相应的下标。
使用散列的查找算法分为两步。第一步是用散列函数将被查找的键转化为数组的一个索引。我们需要面对两个或多个键都会散列到相同的索引值的情况。因此，第二步就是一个处理碰撞冲突的过程，由两种经典解决碰撞的方法：拉链法和线性探测法。

散列表是算法在时间和空间上作出权衡的经典例子。
如果没有内存限制，我们可以直接将键作为（可能是一个超大的）数组的索引，那么所有查找操作只需要访问内存一次即可完成。但这种情况不会经常出现，因此当键很多时需要的内存太大。
另一方面，如果没有时间限制，我们可以使用无序数组并进行顺序查找，这样就只需要很少的内存。而散列表则使用了适度的空间和时间并在这两个极端之间找到了一种平衡。

●散列函数

我们面对的第一个问题就是散列函数的计算，这个过程会将键转化为数组的索引。我们要找的散列函数应该易于计算并且能够均匀分布所有的键。
散列函数和键的类型有关，对于每种类型的键我们都需要一个与之对应的散列函数。

正整数
将整数散列最常用的方法就是**除留余数法**。我们选择大小为**素数M**的数组，对于任意正整数k，计算k除以M的余数。（如果M不是素数，我们可能无法利用键中包含的所有信息，这可能导致我们无法均匀地散列值。）

浮点数

将键表示为二进制数，然后再使用除留余数法。（让浮点数的各个位都起作用）（Java就是这么做的）

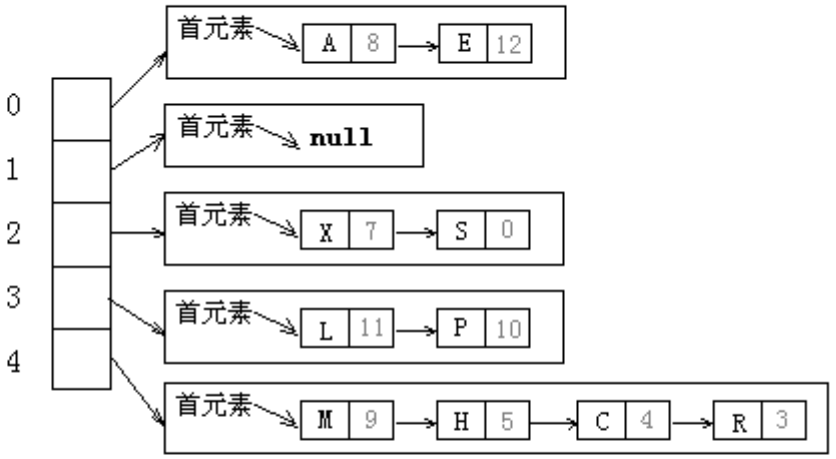
字符串
除留余数法也可以处理较长的键，例如字符串，我们只需将它们当做大整数即可。即相当于**将字符串当做一个N位的R进制值，将它除以M并取余。**

……软缓存
如果散列值的计算很耗时，那么我们或许可以将每个键的散列值缓存起来，即在每个键中使用一个hash变量来保存它的hashCode()返回值。

●基于拉链法的散列表

一个散列函数能够将键转化为数组索引。散列算法的第二步是碰撞处理，也就是处理两个或多个键的散列值相同的情况。
拉链法：将大小为M的数组中的每个元素指向一条链表，链表中的每个结点都存储了散列值为该元素的索引的键值对。
查找分两步：首先根据散列值找到对应的链表，然后沿着链表顺序查找相应的键。

键	值	散列值
S	0	2
E	1	0
A	2	0
R	3	4
C	4	4
H	5	4
E	6	0
X	7	2
A	8	0
M	9	4
P	10	3
L	11	3
E	12	0



拉链法在实际情况中很有用，因为每条链表确实都大约含有N/M个键值对。
基于拉链法的散列表的实现简单。在键的顺序并不重要的应用中，它可能是最快的（也是使用最广泛的）符号表实现。

●基于线性探测法的散列表

实现散列表的另一种方式就是用大小为M的数组保存N个键值对，其中M>N。我们需要依靠数组中的空位解决碰撞冲突。基于这种策略的所有方法被统称为开放地址散列表。
开放地址散列表中最简单的方法叫做**线性探测法**：当碰撞发生时，我们直接检查散列表中的下一个位置（将索引值加1），如果不同则继续查找，直到找到该键或遇到一个空元素。

（开放地址类的散列表的核心思想是：与其将内存用作链表，不如将它们作为在散列表的空元素。这些空元素可以作为查找结束的标志。）

特点：散列最主要的目的在于均匀地将键散布开来，因此在计算散列后键的顺序信息就丢失了，如果你需要快速找到最大或最小的键，或是查找某个范围内的键，散列表都不是合适的选择。

【应用举例】

海量处理

给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url？

答：

可以估计每个文件安的大小为5G×64=320G，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

分而治之/hash映射：

遍历文件a，对每个url求取，然后根据所取得的值将url分别存储到1000个小文件（记为，这里漏写个了a1）中。这样每个小文件的大约为300M。遍历文件b，采取和a相同的方式将url分别存储到1000小文件中（记为）。这样处理后，所有可能相同的url都在对应的小文件（）中，不对应的小文件不可能有相同的url。然后我们只要求出1000对小文件中相同的url即可。

hash_set统计：

求每对小文件中相同的url时，可以把其中一个小文件的url存储到hash_set中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash_set中，如果是，那么就是共同的url，存到文件里面就可以了。

（此题来源于v_July_v的博客）

B树（多向平衡查找树）

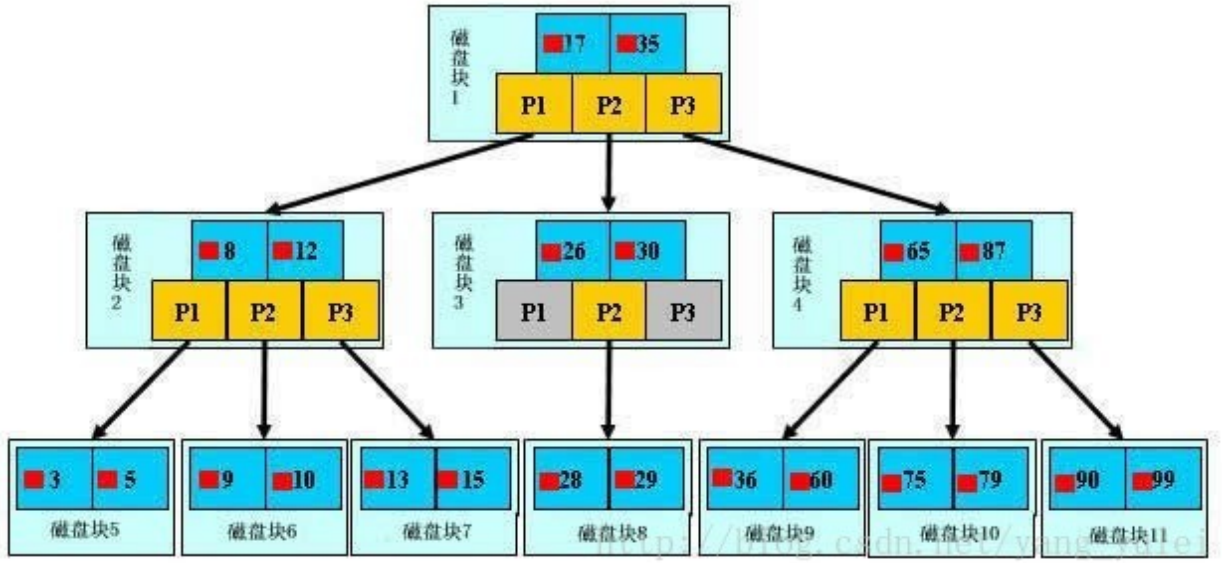
B-树是对2-3树数据结构的扩展。它支持对保存在磁盘或者网络上的符号表进行外部查找，这些文件可能比我们以前考虑的输入要大的多（以前的输入能够保存在内存中）。

（B树和B+树是实现数据库的数据结构，一般程序员用不到它。）

和2-3树一样，我们限制了每个结点中能够含有的“键-链接”对的上下数量界限：一个M阶的B-树，每个结点最多含有M-1对键-链接（假设M足够小，使得每个M向结点都能够存放在一个页中），最少含有M/2对键-链接，但也不能少于2对。

(B树是用于存储海量数据的，一般其一个结点就占用磁盘一个块的大小。)

【注】以下B树部分参考自July的博客，尤其是插入及删除示图，为了省力直接Copy自July。



B树中的结点存放的是键-值对。图中红色方块即为键对应值的指针。

B树中的每个结点根据实际情况可以包含大量的关键字信息和分支(当然是不能超过磁盘块的大小，根据磁盘驱动(diskdrives)的不同，一般块的大小在1k~4k左右)；这样树的深度降低了，这就意味着查找一个元素只要很少结点从外存磁盘中读入内存，很快访问到要查找的数据。

查找

假如每个盘块可以正好存放一个B树的结点（正好存放2个文件名）。那么一个BTNODE结点就代表一个盘块，而子树指针就是存放另外一个盘块的地址。

下面，咱们来模拟下查找文件29的过程：

1. 根据根结点指针找到文件目录的根磁盘块1，将其中的信息导入内存。【磁盘IO操作1次】
2. 此时内存中有两个文件名17、35和三个存储其他磁盘页面地址的数据。根据算法我们发现：17<29<35，因此我们找到指针p2。
3. 根据p2指针，我们定位到磁盘块3，并将其中的信息导入内存。【磁盘IO操作 2次】
4. 此时内存中有两个文件名26，30和三个存储其他磁盘页面地址的数据。根据算法我们发现：26<29<30，因此我们找到指针p2。
5. 根据p2指针，我们定位到磁盘块8，并将其中的信息导入内存。【磁盘IO操作 3次】

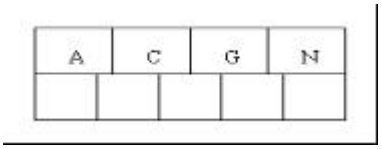
6. 此时内存中有两个文件名28，29。根据算法我们查找到文件名29，并定位了该文件内存的磁盘地址。分析上面的过程，发现需要3 3次磁盘IO操作和次磁盘IO操作和3次内存查找次内存查找操作。关于内存中的文件名查找，由于是一个有序表结构，可以利用折半查找提高效率。至于IO操作是影响整个B树查找效率的决定因素。

插入

想想2-3树的插入。2-3树结点的最大容量是2个元素，故当插入操作造成超出容量之后，就得分裂。同样m-阶B树规定的结点的最大容量是m-1个元素，故当插入操作造成超出容量之后也得分裂，其分裂成两个结点每个结点分m/2个元素。（副作用是在其父结点中要插入一个中间元素，用于分隔这两结点。和2-3树一样，再向父结点插入一个元素也可能会造成父结点的分裂，逐级向上操作，直到不再造成分裂为止。）
向某结点中插入一个元素使其分裂，可能会造成连锁反应，使其之上的结点也可能造成分裂。

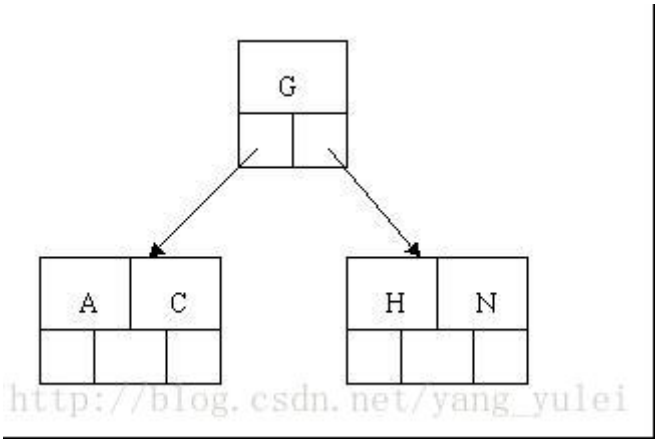
总结：在B树中插入关键码key的思路：
对高度为h的m阶B树，新结点一般是插在第h层。通过检索可以确定关键码应插入的结点位置。然后分两种情况讨论：
1、 若该结点中关键码个数小于m-1，则直接插入即可。
2、 若该结点中关键码个数等于m-1，则将引起结点的分裂。以中间关键码为界将结点一分为二，产生一个新结点，并把中间关键码插入到父结点(h-1层)中
重复上述工作，最坏情况一直分裂到根结点，建立一个新的根结点，整个B树增加一层。

【例】
1、下面咱们通过一个实例来逐步讲解下。插入以下字符字母到一棵空的B 树中（非根结点**关键字数**小了（小于2个）就合并，大了（超过4个）就分裂）：
C N G A H E K Q M F W L T Z D P R X Y S，首先，结点空间足够，4个字母插入相同的结点中，如下图：

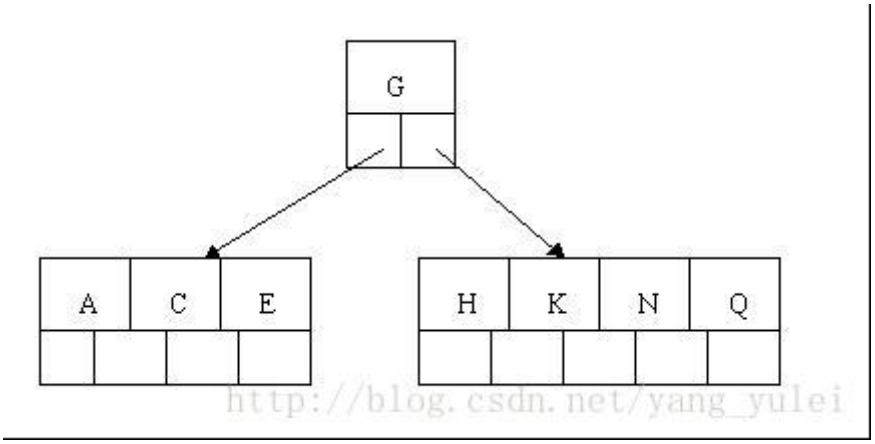


2、当咱们试着插入H时，结点发现空间不够，以致将其分裂成2个结点，移动中间元素G上移到新的根结点中，在实现过程中，咱们把A和C留在当前结点中，而H和N放置新的其右邻

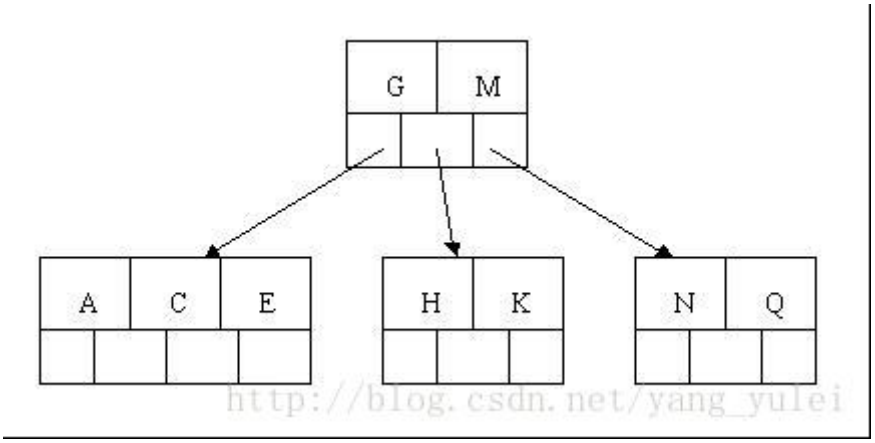
居结点中。如下图：



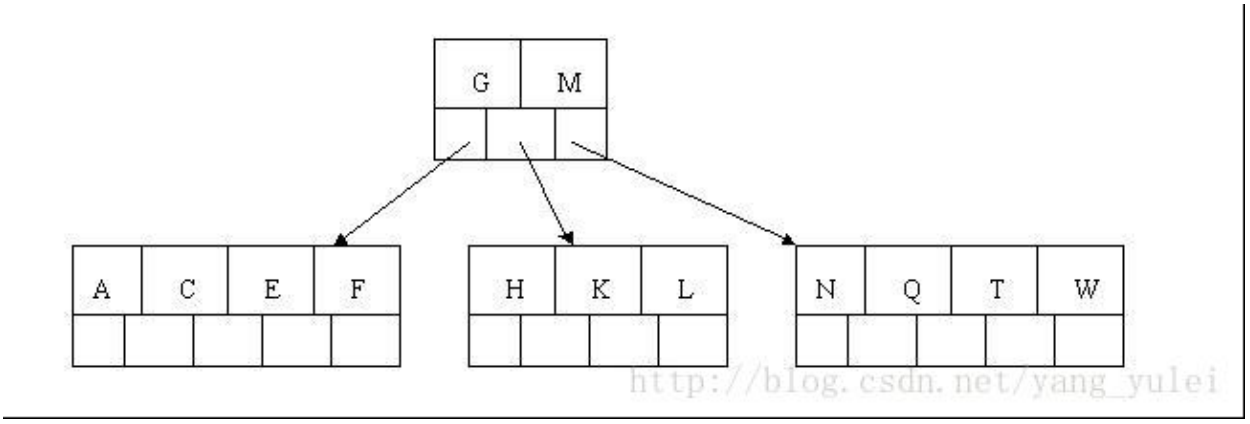
3、当咱们插入E,K,Q时，不需要任何分裂操作



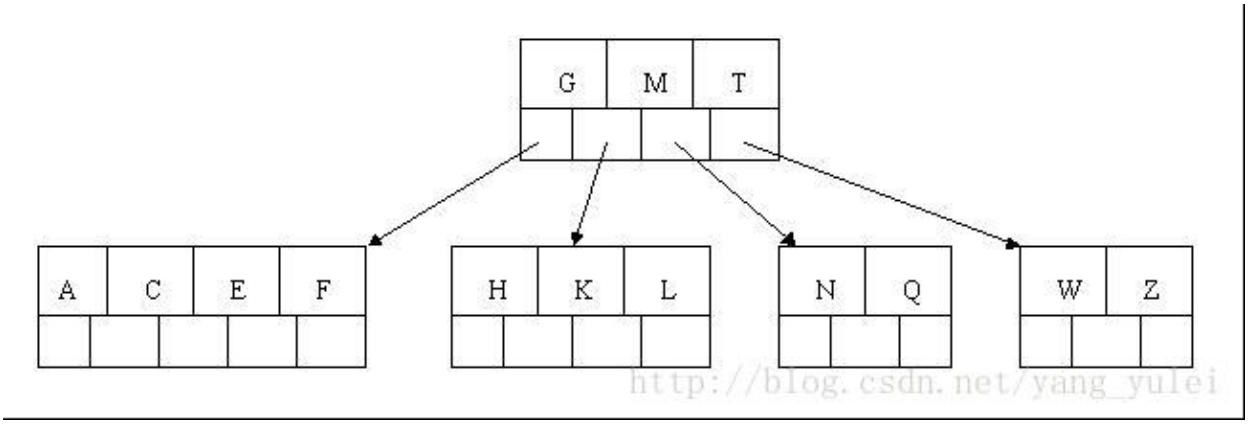
4、插入M需要一次分裂，注意M恰好是中间关键字元素，以致向上移到父节点中



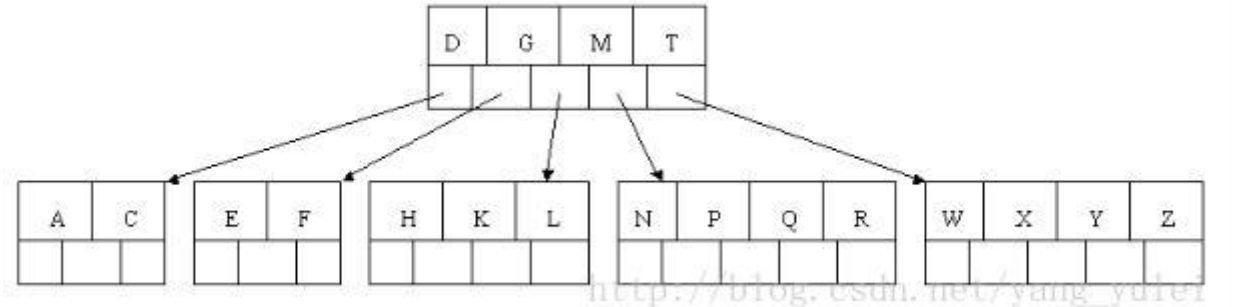
5、插入F,W,L,T不需要任何分裂操作



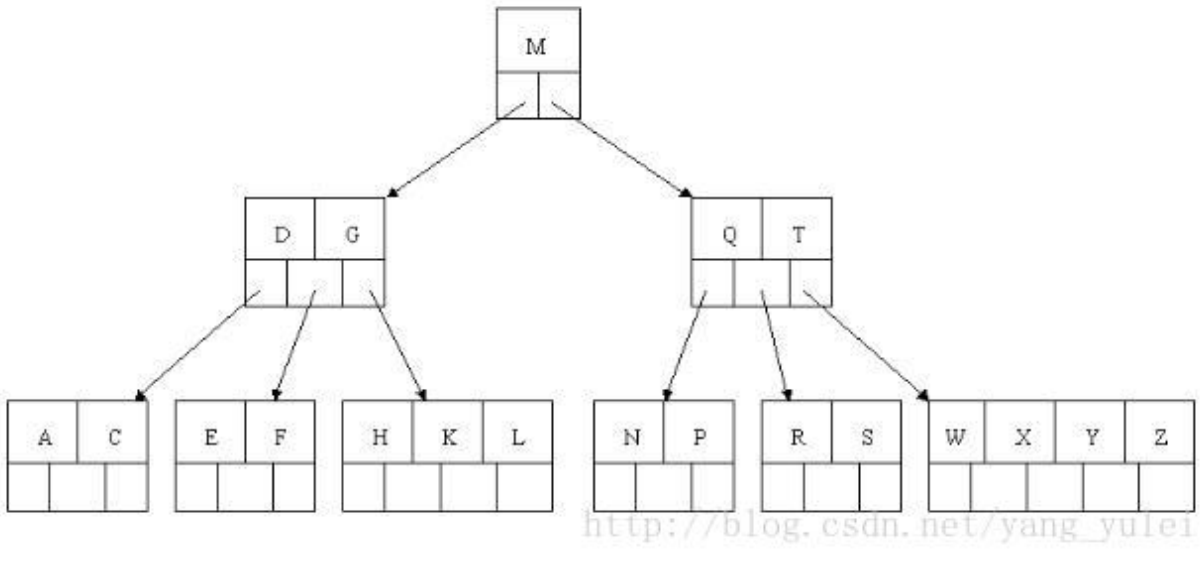
6、插入Z时，最右的叶子结点空间满了，需要进行分裂操作，中间元素T上移到父节点中，注意通过上移中间元素，树最终还是保持平衡，分裂结果的结点存在2个关键字元素。



7、插入D时，导致最左边的叶子结点被分裂，D恰好也是中间元素，上移到父节点中，然后字母P,R,X,Y陆续插入不需要任何分裂操作（别忘了，树中至多5个孩子）。



8、最后，当插入S时，含有N,P,Q,R的结点需要分裂，把中间元素Q上移到父节点中，但是情况来了，父节点中空间已经满了，所以也要进行分裂，将父节点中的中间元素M上移到新形成的根结点中，注意以前在父节点中的第三个指针在修改后包括D和G节点中。这样具体插入操作的完成，下面介绍删除操作，删除操作相对于插入操作要考虑的情况多点。



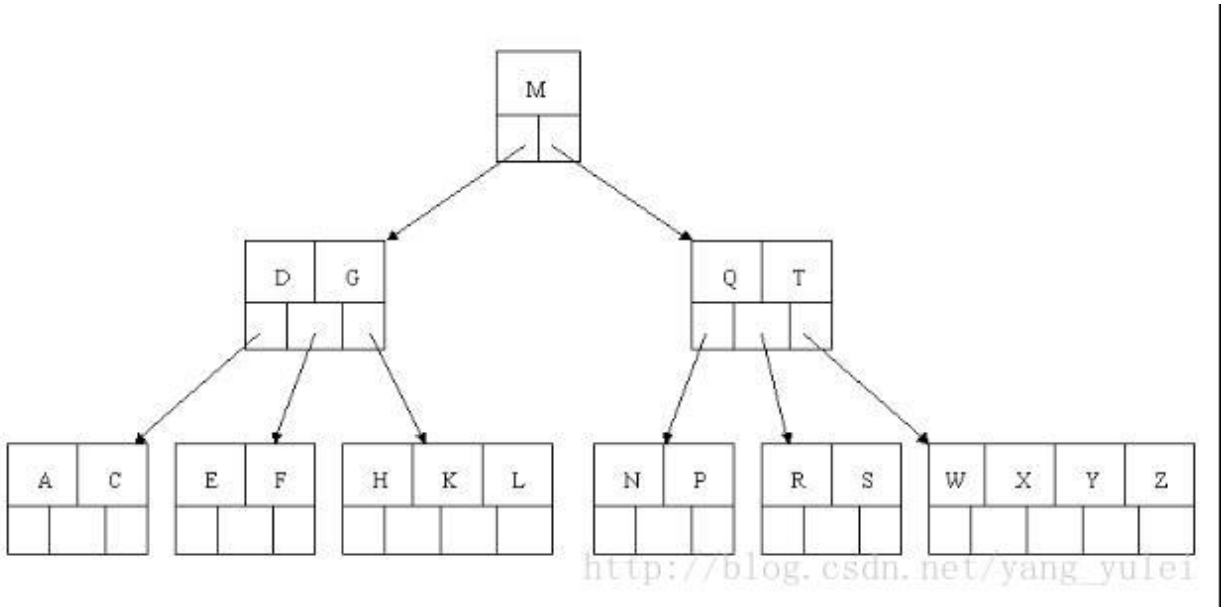
删除(delete)操作

首先查找B树中需删除的元素,如果该元素在B树中存在，则将该元素在其结点中进行删除，如果删除该元素后，首先判断该元素是否有左右孩子结点，如果有，则上移孩子结点中的某相近元素(“左孩子最右边的节点”或“右孩子最左边的节点”)到父节点中，然后是移动之后的情况；如果没有，直接删除后，移动之后的情况。

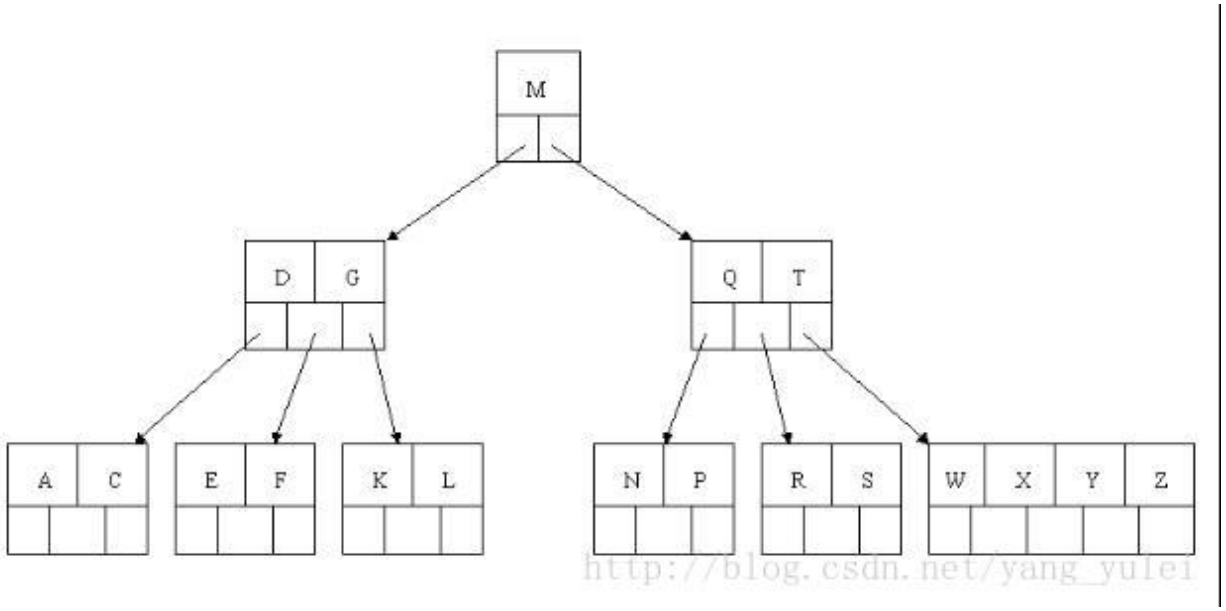
删除元素，移动相应元素之后，如果某结点中元素数目（即关键字数）小于ceil(m/2)-1，则需要看其某相邻兄弟结点是否丰满（结点中元素个数大于ceil(m/2)-1）（**还记得第一节中关于B树的第5个特性中的c点么？： c)除根结点之外的结点（包括叶子结点）的关键字的个数n必须满足：（ceil(m / 2)-1）<= n <=m-1。m表示最多含有m个孩子，n表示关键字数。在本小节中举的一颗B树的示例中，关键字数n满足：2<=n<=4**），如果丰满，则向父节点借一个元素来满足条件；如果其相邻兄弟都刚脱贫，即借了之后其结点数目小于ceil(m/2)-1，则该结点与其相邻的某一兄弟结点进行“合并”成一个结点，以此来满足条件。那咱们通过下面实例来详细了解吧。

以上述插入操作构造的一棵5阶B树（树中最多含有m（m=5）个孩子，因此关键字数最小为**ceil(m/ 2)-1=2。**还是这句话，**关键字数**小了（小于2个）就合并，大了（超过4个）就分裂）为例，依次删除H,T,R,E。

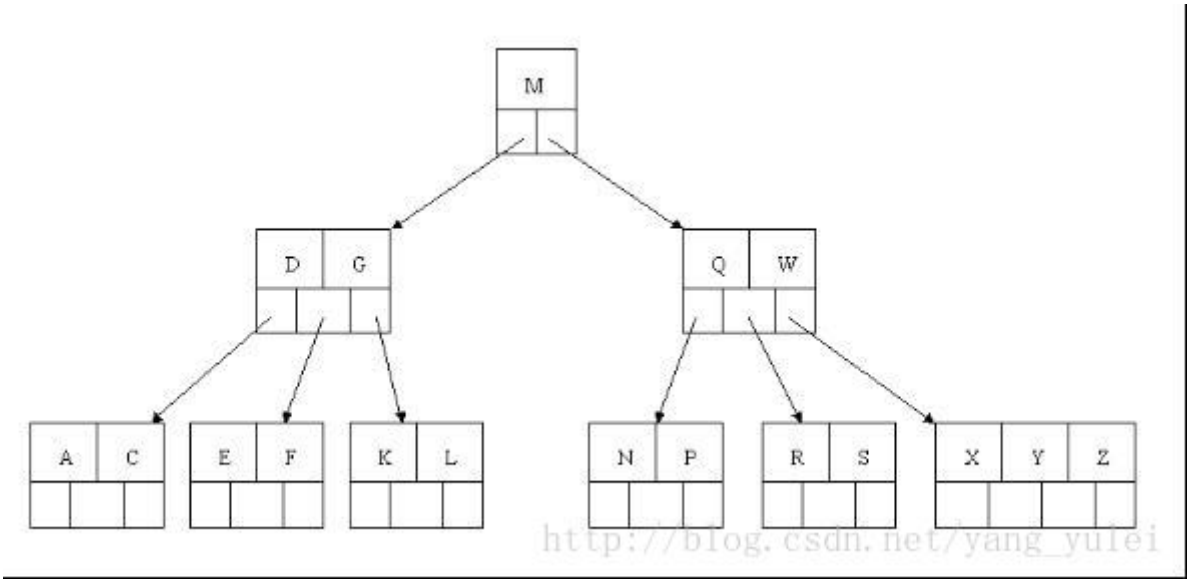
1、首先删除元素H，当然首先查找H，H在一个叶子结点中，且该叶子结点元素数目3大于最小元素数目ceil(m/2)-1=2，则操作很简单，咱们只需要移动K至原来H的位置，移动L至K的位置（也就是结点中删除元素后面的元素向前移动）



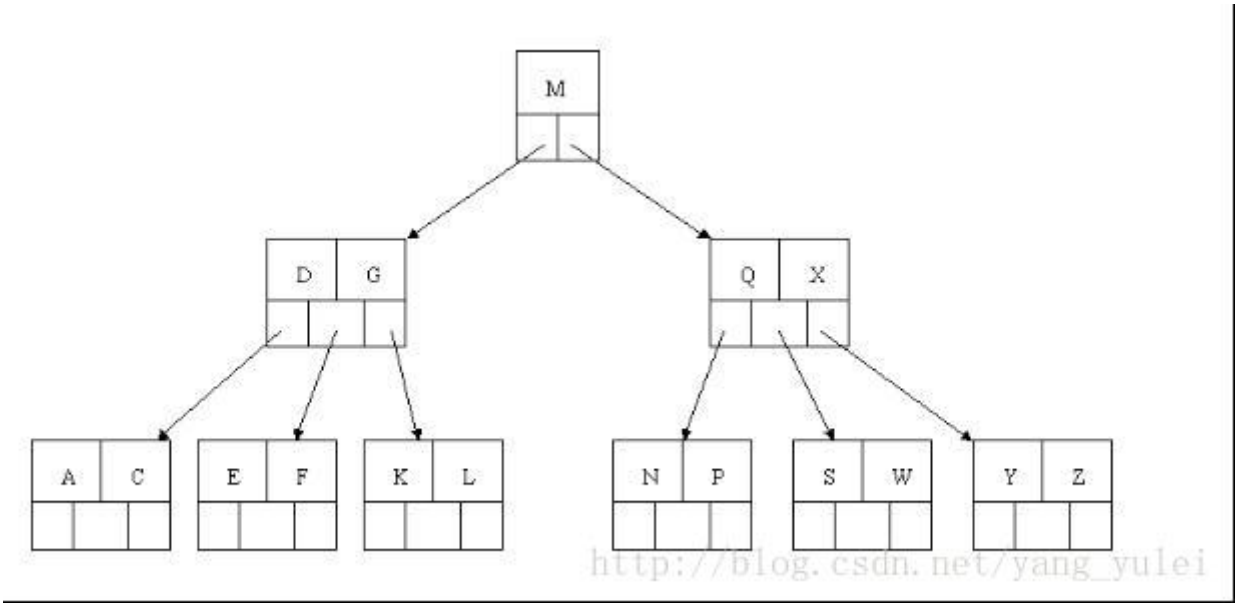
2、下一步，删除T,因为T没有在叶子结点中，而是在中间结点中找到，咱们发现他的继承者W(字母升序的下个元素)，将W上移到T的位置，然后将原包含W的孩子结点中的W进行删除，这里恰好删除W后，该孩子结点中元素个数大于2，无需进行合并操作。



3、下一步删除R，R在叶子结点中,但是该结点中元素数目为2，删除导致只有1个元素，已经小于最小元素数目ceil(5/2)-1=2,而由前面我们已经知道：如果其某个相邻兄弟结点中比较丰满（元素个数大于ceil(5/2)-1=2），则可以向父结点借一个元素，然后将最丰满的相邻兄弟结点中上移最后或最前一个元素到父节点中（有没有看到红黑树中左旋操作的影子?），在这个实例中，右相邻兄弟结点中比较丰满（3个元素大于2），所以先向父节点借一个元素W下移到该叶子结点中，代替原来S的位置，S前移；然后X在相邻右兄弟结点中上移到父结点中，最后在相邻右兄弟结点中删除X，后面元素前移。

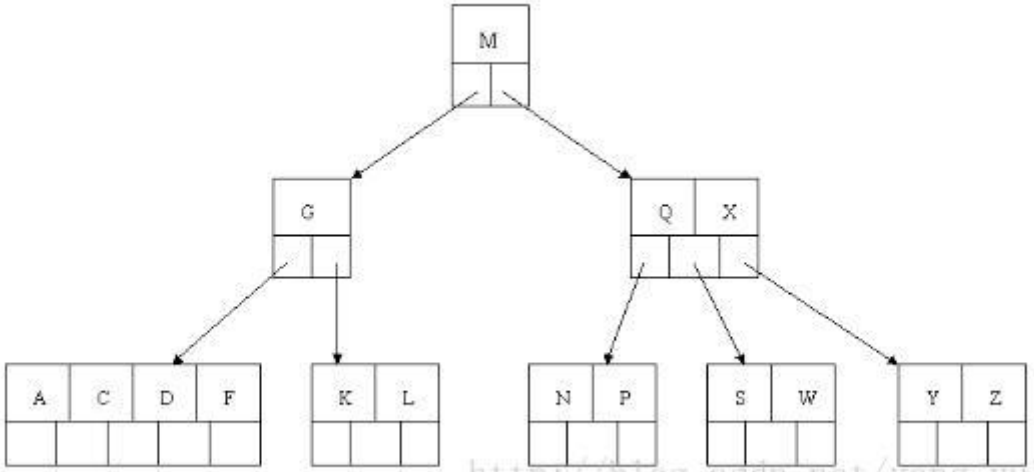


4、最后一步删除E， 删除后会导致很多问题，因为E所在的结点数目刚好达标，刚好满足最小元素个数（ceil(5/2)-1=2）,而相邻的兄弟结点也是同样的情况，删除一个元素都不能满足条件，所以需要该节点与某相邻兄弟结点进行合并操作；首先移动父结点中的元素（该元素在两个需要合并的两个结点元素之间）下移到其子结点中，然后将这两个结点进行合并成一个结点。所以在该实例中，咱们首先将父节点中的元素D下移到已经删除E而只有F的结点中，然后将含有D和F的结点和含有A,C的相邻兄弟结点进行合并成一个结点。

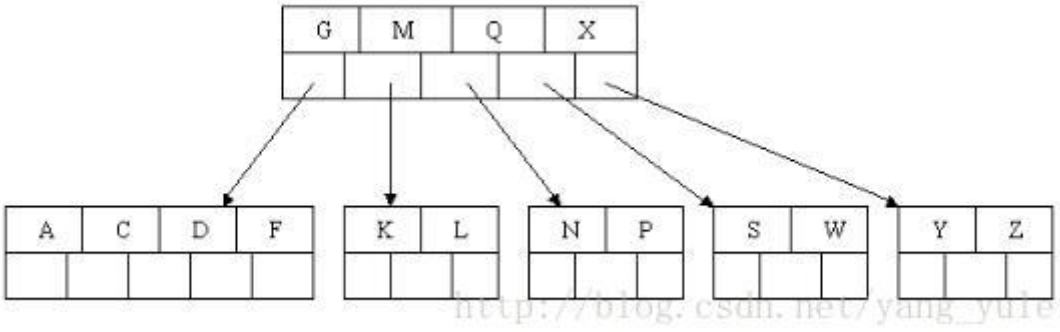


5、也许你认为这样删除操作已经结束了，其实不然，在看看上图，对于这种特殊情况，你立即会发现父节点只包含一个元素G，没达标（因为非根节点包括叶子结点的关键字数n必须满足于2=<n<=4，而此处的n=1），这是不能够接受的。如果这个问题结点的相邻兄弟比较丰满，则可以向父结点借一个元素。假设这时右兄弟结点（含有Q,X）有一个以上的元素

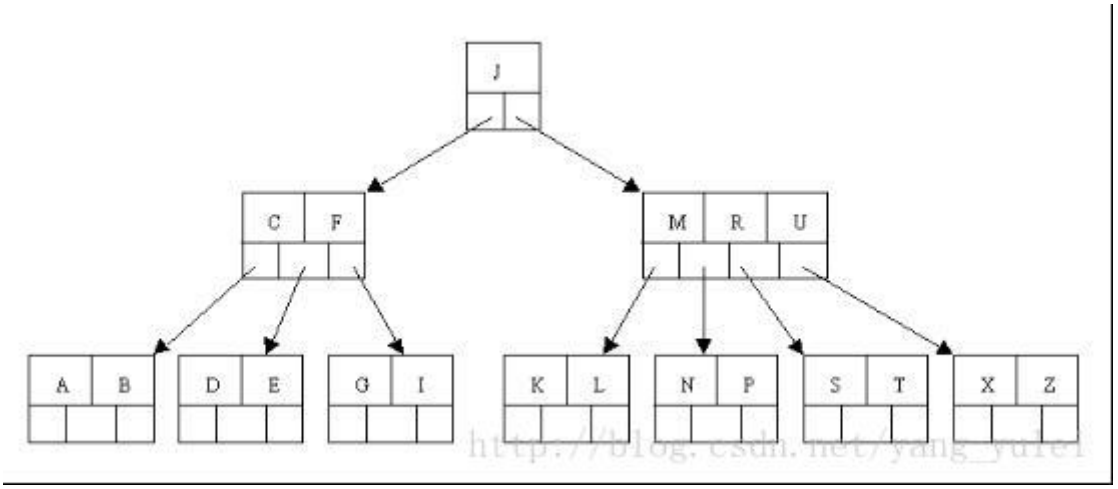
（Q右边还有元素），然后咱们将M下移到元素很少的子结点中，将Q上移到M的位置，这时，Q的左子树将变成M的右子树，也就是含有N，P结点被依附在M的右指针上。所以在这个实例中，咱们没有办法去借一个元素，只能与兄弟结点进行合并成一个结点，而根结点中的唯一元素M下移到子结点，这样，树的高度减少一层。



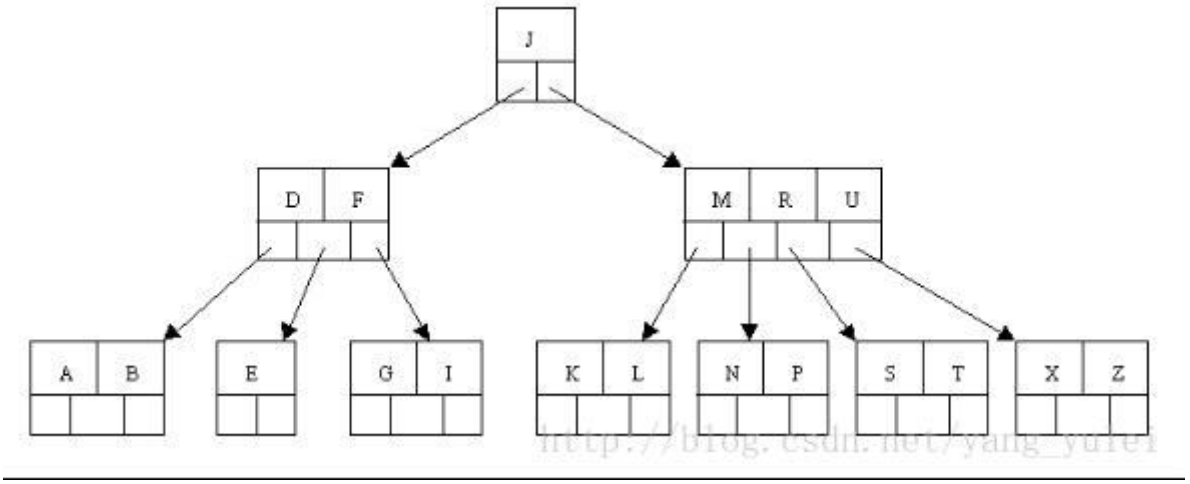
为了进一步详细讨论删除的情况，**再举另外一个实例**：
这里是一棵不同的5序B树，那咱们试着删除C



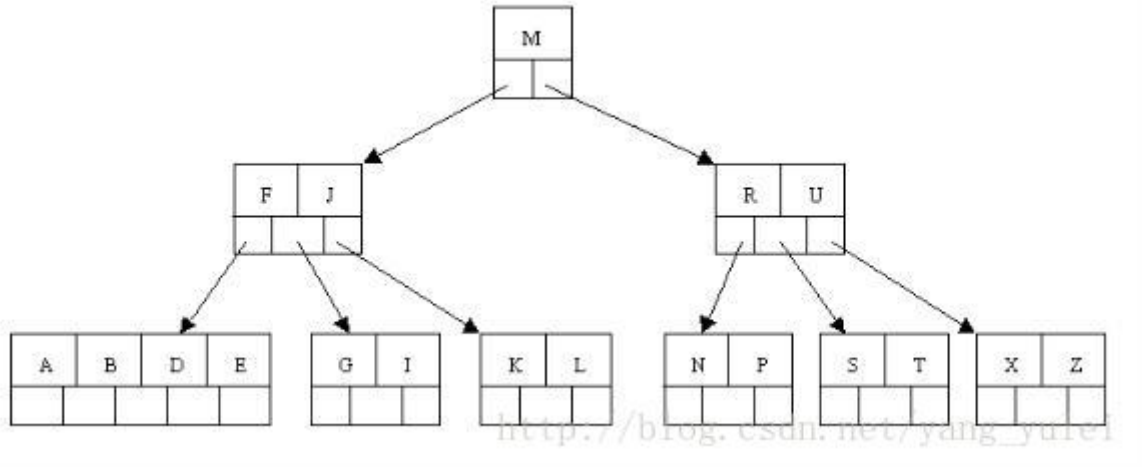
于是将删除元素C的右子结点中的D元素上移到C的位置，但是出现上移元素后，只有一个元素的结点的情况。
又因为含有E的结点，其相邻兄弟结点才刚脱贫（最少元素个数为2），不可能向父节点借元素，所以只能进行合并操作，于是这里将含有A,B的左兄弟结点和含有E的结点进行合并成一个结点。



这样又出现只含有一个元素F结点的情况，这时，其相邻的兄弟结点是丰满的（元素个数为3>最小元素个数2），这样就可以想父结点借元素了，把父结点中的J下移到该结点中，相应的如果结点中J后有元素则前移，然后相邻兄弟结点中的第一个元素（或者最后一个元素）上移到父节点中，后面的元素（或者前面的元素）前移（或者后移）；注意含有K，L的结点以前依附在M的左边，现在变为依附在J的右边。这样每个结点都满足B树结构性质。



从以上操作可看出：除根结点之外的结点（包括叶子结点）的关键字的个数n满足：（ceil(m / 2)-1）<= n <= m-1，即2<=n<=4。这也佐证了咱们之前的观点。删除操作完。



（我思：）

（1、　　关于B树中指针的表示。指针就是线索，是为了指示你找到目标。在内存中用内存的线性地址表示，在磁盘上，用磁盘的柱面和磁道号表示。

（2、　　B树也是一种文件组织形式。它与OS文件系统的区别是，文件系统是面向磁盘上各种应用的文件的，所有文件的索引都被组织在一个系统文件表中。这样，一个相关应用的文件之间就没有体现有序性，我们对某组相关的文件进行查找，效率就会较低。 而B树是专门对某组相关的文件进行组织，使其之间相对有序，提高查找效率。 --尤其是对于需要频繁查找访问文件的操作。

例如： 对10亿个有序数，其分布在1000个文件中。普通的查找（类2分查找），和构造一个B树，普通的二分查找不仅需要多次访问文件，且其通过OS的文件系统通过文件名来访问文件，这样效率低——OS需要在整张系统文件表中通过文件名查找文件。 而B树，其是多叉树，树的深度比二分树要小很多，需要查找的文件比二分查找需要的少。且其通过自己建立的B树来索引文件（每次查找文件都通过该B树得到文件在磁盘上的位置）。B树是独立于OS的文件系统的，它中的每个文件都有相应的磁盘位置，而不仅是文件名。

B+树

B+ tree：是应文件系统所需而产生的一种B-tree的变形树。

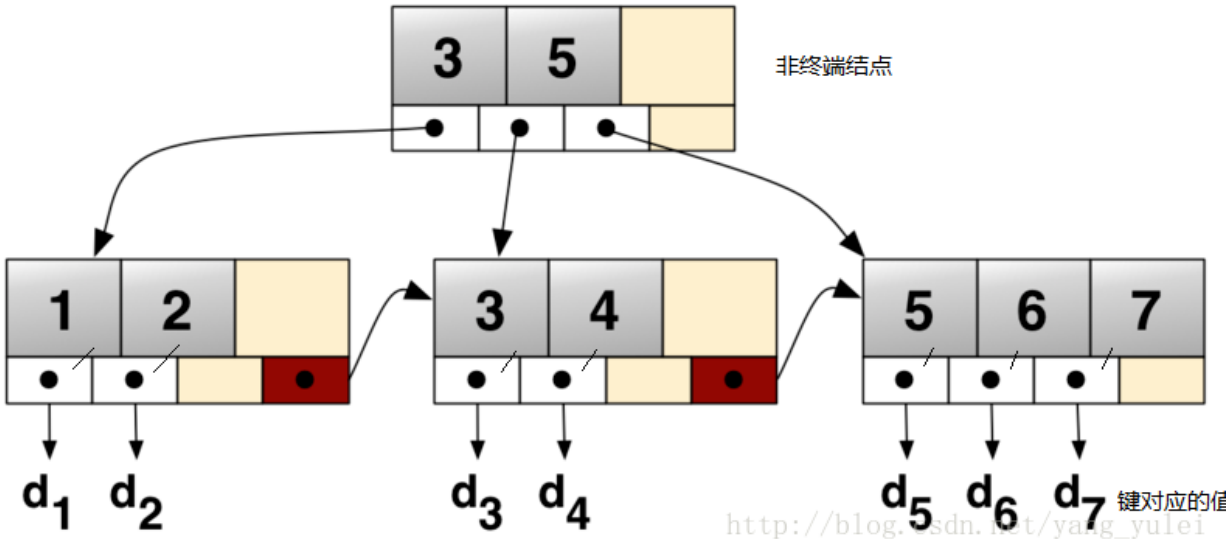
一棵m阶的B+树和m阶的B树的异同点在于：

1、有n棵子树的结点中含有n-1 个关键字； (与B 树n棵子树有n-1个关键字 保持一致，)

2、所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。

3、所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。

【总结：最大的区别在于，B树是像2-3树那样把数据分散到所有的结点中，而B+树的数据都集中在叶结点，上层结点只是数据的索引，并不包含数据信息】



【应用举例】

1、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库索引？

数据库索引采用B+树的主要原因是 B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。正是为了解决这个问题，B+树应运而生。

B+树只要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树需要遍历整棵树，效率太低。

2、B+-tree的应用: VSAM(虚拟存储存取法)文件

B树与B+树

走进搜索引擎的作者梁斌老师针对B树、B+树给出了他的意见（来源于July）：

“B+树还有一个最大的好处，方便扫库，B树必须用中序遍历的方法按序扫库，而B+树直接从叶子结点挨个扫一遍就完了，B+树支持range-query非常方便，而B树不支持。这是数据库选用B+树的最主要原因。

比如要查 5-10之间的，B+树一把到5这个标记，再一把到10，然后串起来就行了，B树就非常麻烦。B树的好处，就是成功查询特别有利，因为树的高度总体要比B+树矮。不成功的情况下，B树也比B+树稍稍占一点点便宜。B树比如你的例子中查，17的话，一把就得到结果了。

有很多基于频率的搜索是选用B树，越频繁query的结点越往根上走，前提是需要对query做统计，而且要对key做一些变化。

另外B树也好B+树也好，根或者上面几层因为被反复query，所以这几块基本都在内存中，不会出现读磁盘IO，一般已启动的时候，就会主动换入内存。”

"MySQL 底层存储是用B+树实现的，因为在内存中B+树是没有优势的，但是一到磁盘，B+树的威力就出来了"。

B+树是B树的变形，它把所有的附属数据都放在叶子结点中，只将关键字和子女指针保存于内结点，**内结点完全是索引的功能**，最大化了内结点的分支因子。不过是n个关键字对应着n个子女，子女中含有父辈的结点信息，叶子结点包含所有信息（内结点包含在叶子结点中，内结点没有指向“附属数据”的指针必须索引到叶子结点）。这样的话还有一个好处就是对于每个结点所需的索引次数都是相等的，保证了稳定性。

【B*树】

B*树是B+树的变体，**在B+树非根和非叶子结点再增加指向兄弟的指针**；B*树定义了**非叶子结点关键字个数至少为(2/3)*M**，即块的最低使用率为2/3（代替B+树的1/2）。

B+树的分裂：当一个结点满时，分配一个新的结点，并将原结点中1/2的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针；

B*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制1/3的数据到新结点，最后在父结点增加新结点的指针；

所以，B*树分配新结点的概率比B+树要低，空间使用率更高；

在数据库中的应用及性能分析

一般关系型数据库使用B+树来做索引，NoSQL数据库用哈希来做索引。例如MySQL就普遍使用B+Tree实现其索引结构。

上文说过，红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用B/B+Tree作为索引结构。
因为索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：
当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。
预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

【下面分析B/B+Tree索引的性能】
我们使用磁盘I/O次数评价索引结构的优劣。先从B Tree分析，根据B Tree的定义，可知检索一次最多需要访问h个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现中B-Tree在每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。
B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)。
一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。
综上所述，用B-Tree作为索引结构效率是非常高的。
而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

B+Tree更适合外存索引，原因和内节点出度d有关。从上面分析可以看到，d越大索引的性能越好，而出度的上限取决于节点内key和data的大小，由于B+Tree内节点去掉了data域，因此可以拥有更大的出度，拥有更好的性能。

我应该使用符号表的哪种实现

对于典型的应用程序，应该在散列表和二叉查找树之间进行选择。
相对于二叉查找树，散列表的优点在于代码更简单，且查找时间最优（常数级别）。二叉查找树相对于散列表的优点在于抽象结构更简单（不需要设计散列函数），红黑树可以保证最坏情况下的性能且它能够支持的操作更多（如排名、选择和范围查找）。
大多数程序员的第一选择都是散列表，在其他因素更重要时才会选择红黑树。（”第一选择”的例外：当键都是长字符串时，我们可以构造出比红黑树更灵活而又比散列表更高效的数据结构 Trie树）

==========字符串的查找==========

单词查找树(Trie树)

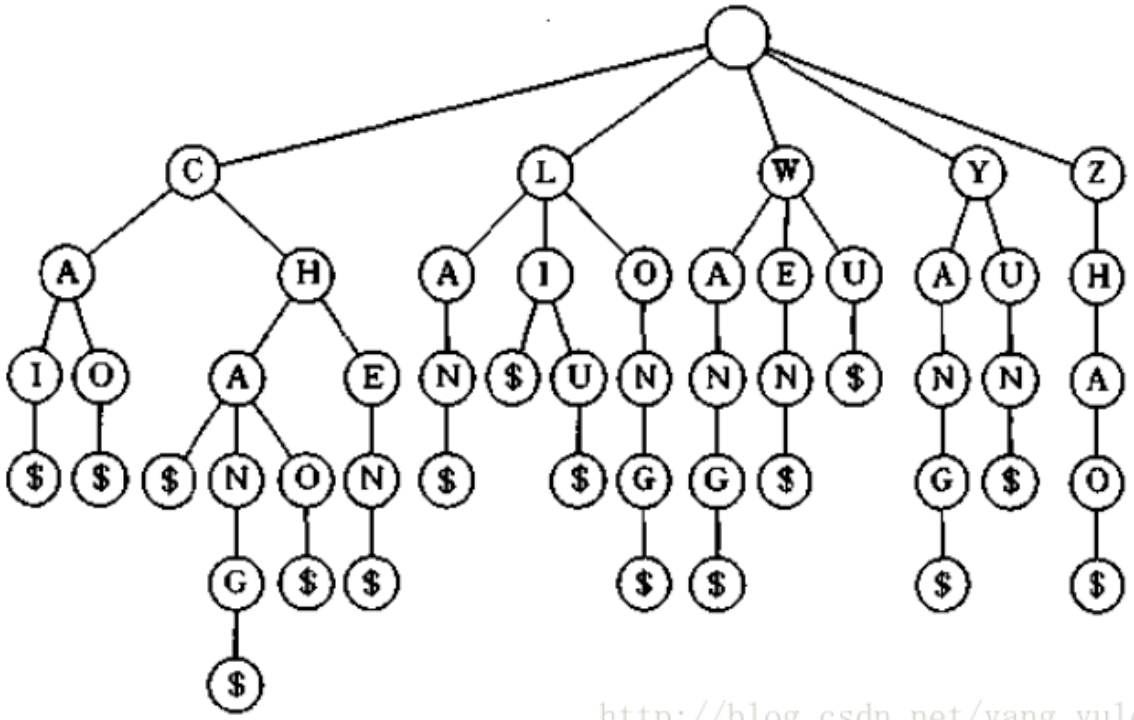
单词查找树的英文单词trie来自于E.Fredkin在1960年玩的一个文字游戏，因为这个数据结构的作用是取出(retrieval)数据，但发音为try是为了避免与tree相混淆。

基本性质：
每个结点都含有R条链接，其中R为字母表的大小。（单词查找树一般都含有大量的空链接，因此在绘制一颗单词查找树时一般会忽略空链接。）

树中的每个结点中不是包含一个或几个关键字，而是只含有组成关键字的符号。例如，若关键字是数值，则结点中只包含一个数位；若关键字是单词，则结点中只包含一个字母字符。
我们将每个键所关联的值保存在该键的最后一个字母所对应的结点中。
（这种树会给某种类型关键字的表的查找带来方便。）

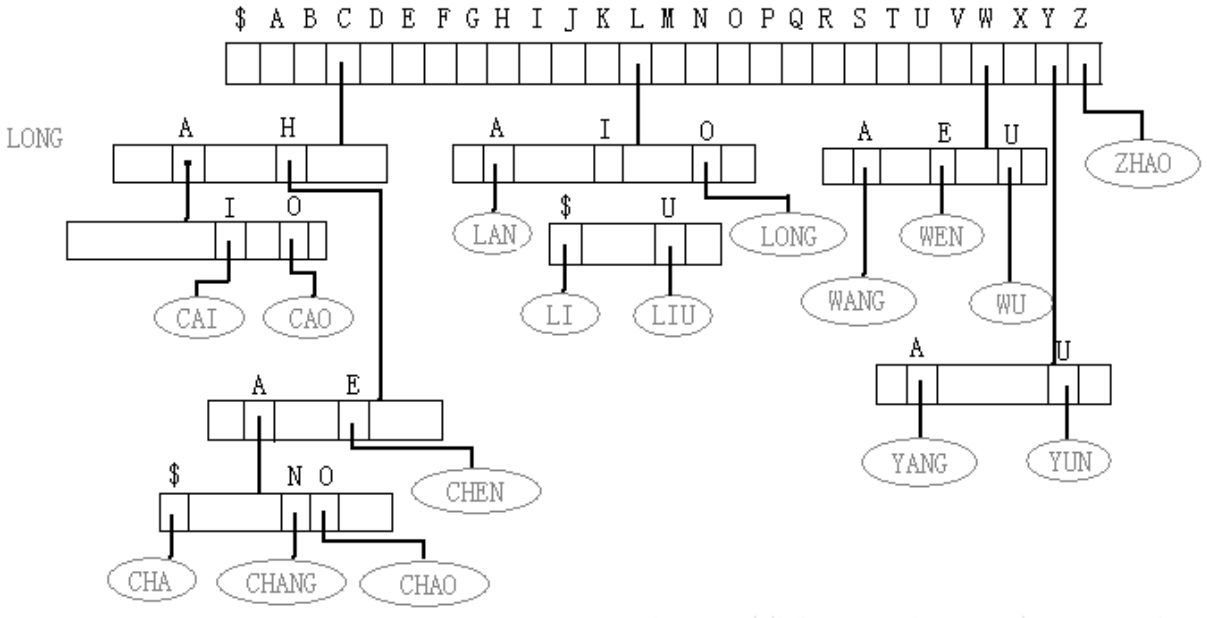
假设有如下关键字的集合

{ CAI、CAO、LI、LAN、CHA、CHANG、WEN、CHAO、YUN、YANG、LONG、WANG、ZHAO、LIU、WU、CHEN }



若以树的多重链表来表示Trie树，则树的每个结点中应含有d个指针域。

若从Trie树中某个结点到叶子结点的路径上每个结点都只有一个孩子，则可将该路径上所有结点压缩成一个“叶子结点”，且在该叶子结点中存储关键字及指向记录的指针等信息。



在Trie树中有两种结点：
分支结点：含有d个指针域和一个指示该结点中非空指针域的个数的整数域。（分支结点所表示的字符是由其指向子树指针的索引位置决定的）
叶子结点：含有关键字域和指向记录的指针域。

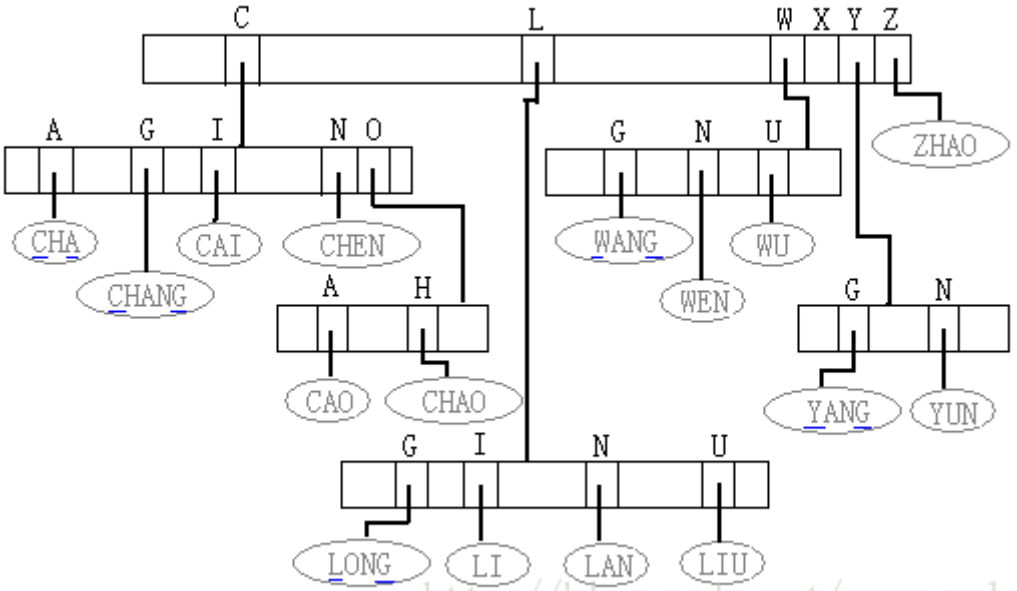
```
typedef structTrieNode
{
   NodeKind kind ;
   union {
      struct {KeyType K;  Record *infoptr} lf ;   //叶子结点
      struct {TrieNode *ptr[27];  int num} bh ; //分支结点
   } ;
} TrieNode,*TrieTree ;
```

查找

在Trie树上进行查找的过程为：从根结点出发，沿给定值相应的指针逐层向下，直至叶子结点，若叶子结点中的关键字和给定值相等，则查找成功。若分支结点中和给定值相应的指针为空，或叶结点中的关键字和给定值不相等，则查找不成功。

分割

查找操作的时间依赖于树的深度。
我们可以对关键字集选择一种合适的分割，以缩减Trie树的深度。
例如：先按首字符不同分成多个子集之后，然后按最后一个字符不同分割每个子集，再按第二个字符……，前后交叉分割。
如下图：在该树上，除两个叶子结点在第四层上外，其余叶子结点均在第三层上。
若分割的合适，则可使每个叶子结点中只含有少数几个同义词。



插入和删除

在Trie树上易于进行插入和删除，只是需要相应地增加和删除一些分支结点。

把沿途分支结点中相应的指针域置空，再把其分支结点中的num-1，然后删除叶子结点。当分支结点中num域的值减为1时，便可删除。