3.2 B 树[1]

3.2.1 题记

动态查找树主要有二叉查找树（binary search tree）、平衡二叉查找树（balanced binary search tree）、红黑树（Red-Black Tree）、B 树、B+树、B*树。前三者是典型的二叉查找树结构，其查找的时间复杂度 O(log2N)与树的深度相关，那么降低树的深度自然会提高查找效率。

与此同时，在大规模数据存储中实现索引查询这样一个实际背景下，树节点存储的元素数量是有限的（如果元素数量非常多的话，查找就退化成节点内部的线性查找了），这样导致二叉查找树结构由于树的深度过大而造成磁盘 I/O 读写过于频繁，进而导致查询效率低下，因此我们该想办法降低树的深度，从而减少磁盘查找存取的次数。一个基本的想法就是：采用多叉树结构（由于树节点元素数量是有限的，自然该节点的子树数量也就是有限的）。

如此，Rudolf Bayer 和 Edward M. McCreight 在 1970 写的一篇论文《Organization and Maintenance of Large Ordered Indices》中，提出了一个新的查找树结构—平衡多路查找树，即 B-tree（B-tree 树即 B 树，B 即 Balanced，平衡的意思）。

后面我们会看到，B 树的各种操作能使 B 树保持较低的高度，从而有效避免磁盘过于频繁的查找存取操作，达到有效提高查找效率的目的。然在开始介绍 B~tree 之前，先了解下相关的硬件知识，才能更好好的了解为什么需要 B~tree 这种外存数据结构。

3.2.2 外存储器——磁盘

计算机存储设备一般分为两种：内存储器（main memory）和外存储器（external memory）。 内存存取速度快，但容量小，价格昂贵，而且不能长期保存数据（在不通电情况下数据会消失）。

外存储器–磁盘是一种直接存取的存储设备(DASD)。它是以存取时间变化不大为特征的。可以直接存取任何字符组，且容量大、速度较其他外存设备更快。

---

[1] 本节为新书第 3 章的第 2 节，在博客文章《从 B 树、B+ 树、B* 树谈到 R 树》：http://blog.csdn.net/v_july_v/article/details/6530142 的基础上改进优化。后面，在经过又一轮的改进优化之后，本节将被收录于今 2014 年出版的新书中。July、二零一四年八月十八日。

磁盘的构造

磁盘是一个扁平的圆盘(与电唱机的唱片类似)。盘面上有许多称为磁道的圆圈，数据就记录在这些磁道上。磁盘可以是单片的，也可以是由若干盘片组成的盘组，每一盘片上有两个面。如下图中所示的 6 片盘组为例，除去最顶端和最底端的外侧面不存储数据之外，一共有 10 个面可以用来保存信息。

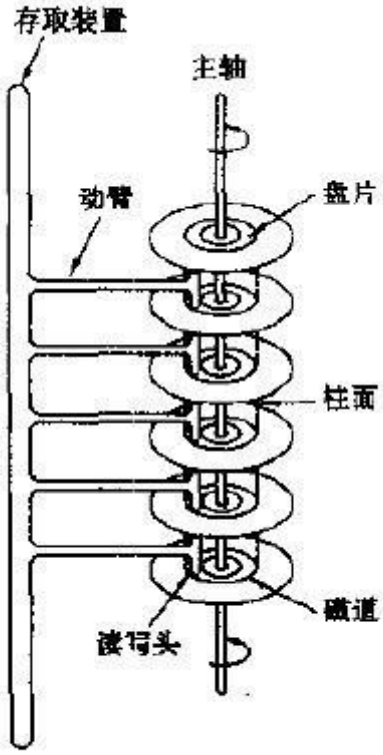

图 3-8

当磁盘驱动器执行读/写功能时。盘片装在一个主轴上，并绕主轴高速旋转，当磁道在读/写头(又叫磁头) 下通过时，就可以进行数据的读 / 写了。

一般磁盘分为固定头盘(磁头固定)和活动头盘。

固定头盘的每一个磁道上都有独立的磁头，它是固定不动的，专门负责这一磁道上数据的读/写。

而活动头盘 (如上图)的磁头是可移动的。每一个盘面上只有一个磁头(磁头是双向的，因此正反盘面都能读写)。它可以从该面的一个磁道移动到另一个磁道。所有磁头都装在同一个动臂上，因此不同盘面上的所有磁头都是同时移动的(行动整齐划一)。当盘片绕主轴旋转的时候，磁头与旋转的盘片形成一个圆柱

体。各个盘面上半径相同的磁道组成了一个圆柱面，我们称为柱面 。因此，柱面的个数也就是盘面上的磁道数。

磁盘的读/写原理和效率

磁盘上数据必须用一个三维地址唯一标示：柱面号、盘面号、块号(磁道上的盘块)。

读/写磁盘上某一指定数据需要下面 3 个步骤：

- 首先移动臂根据柱面号使磁头移动到所需要的柱面上，这一过程被称为定位或查找 。
- 如上图 3-2-1 中所示的 6 盘组示意图中，所有磁头都定位到了 10 个盘面的 10 条磁道上(磁头都是双向的)。这时根据盘面号来确定指定盘面上的磁道。
- 盘面确定以后，盘片开始旋转，将指定块号的磁道段移动至磁头下。

经过上面三个步骤，指定数据的存储位置就被找到。这时就可以开始读/写操作了。

访问某一具体信息，由 3 部分时间组成：

- 查找时间(seek time) Ts: 完成上述步骤(1)所需要的时间。这部分时间代价最高，最大可达到 0.1s 左右。
- 等待时间(latency time) Tl: 完成上述步骤(3)所需要的时间。由于盘片绕主轴旋转速度很快，一般为 7200 转/分(电脑硬盘的性能指标之一, 家用的普通硬盘的转速一般有 5400rpm(笔记本)、7200rpm 几种)。因此一般旋转一圈大约 0.0083s。
- 传输时间(transmission time) Tt: 数据通过系统总线传送到内存的时间，一般传输一个字节(byte)大概 0.02us=2*10^(-8)s

磁盘读取数据是以盘块(block)为基本单位的。位于同一盘块中的所有数据都能被一次性全部读取出来。而磁盘 IO 代价主要花费在查找时间 Ts 上。因此我们应该尽量将相关信息存放在同一盘块，同一磁道中。或者至少放在同一柱面或相邻柱面上，以求在读/写信息时尽量减少磁头来回移动的次数，避免产生过多的查找时间 Ts。

所以，在遇到大规模数据存储的情况时，大量数据存储在外存磁盘中，而在外存磁盘中读取/写入块(block)中某数据时，首先需要定位到磁盘中的某块，如何有效地查找磁盘中的数据，需要一种合理高效的外存数据结构，就是下面所要重点阐述的 B-tree 结构，以及相关的变种结构：B+树结构和 B*树结构。

3.2.3 B 树

什么是 B 树

B-树，即为 B 树。顺便说句，因为 B 树的原英文名称为 B-tree，而国内很多人喜欢把 B-tree 译作 B-树，其实，这是个非常不好的直译，很容易让人产生误解。如人们可能会以为 B-树是一种树，而 B 树又是另外一种树。而事实上是，B-tree 就是指的 B 树。

我们知道，B 树是为了磁盘或其他存储设备而设计的一种多叉（下面你会看到，相对于二叉，B 树每个内结点有多个分支，即多叉）平衡查找树。虽然与之前介绍的红黑树很相似，但在降低磁盘 I/0 操作方面 B 树表现得要更好一些，而且许多数据库系统一般都使用 B 树或者 B 树的各种变形结构。

B 树与红黑树最大的不同在于，B 树的结点可以有许多子女，从几个到几千个。不过 B 树与红黑树也有相同点，一棵含 n 个结点的 B 树的高度也为 O(lgn)，但可能比一棵红黑树的高度小许多，因为它的分支因子比较大，从几十个到几千个分支因子。所以，B 树可以在 O（logn）时间内，实现各种如插入（insert），删除（delete）等动态集合操作。

如图 3-8 所示，即是一棵 B 树，一棵关键字为英语中辅音字母的 B 树，现在要从树中查找字母 R（包含 n[x]个关键字的内结点 x，x 有 n[x]+1 个子女（也就是说，一个内结点 x 若含有 n[x]个关键字，那么 x 将含有 n[x] +1 个子女）。所有的叶结点都处于相同的深度，带阴影的结点为查找字母 R 时要检查的结点）：

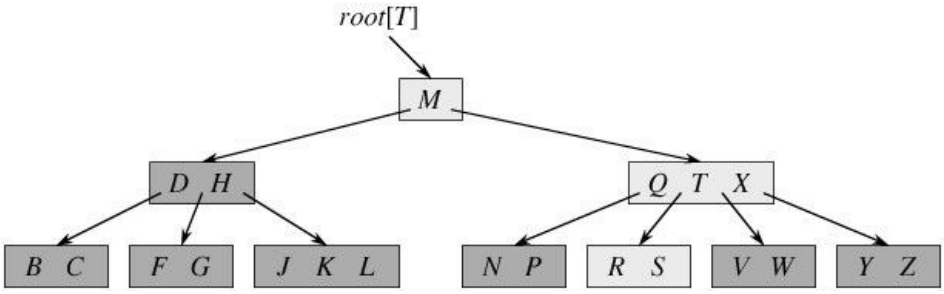

图 3-9

相信，从上图你能轻易的看到，一个内结点 x 若含有 n[x]个关键字，那么 x 将含有 n[x]+1 个子女。如含有 2 个关键字 D H 的内结点有 3 个子女，而含有 3 个关键字 Q T X 的内结点有 4 个子女。

B 树的定义

B 树又叫平衡多路查找树，一棵 m 阶的 B 树的特性如下：

- 树中每个结点最多含有 m 个孩子（m>=2）；

- 除根结点和叶子结点外，其他每个结点至少有[ceil(m / 2)]个孩子（其中 ceil(x)是一个取上限的函数）；
- 根结点至少有 2 个孩子（除非 B 树只包含一个结点：根结点）；
- 所有叶子结点都出现在同一层，叶子结点不包含任何关键字信息(可以看做是外部结点或查询失败的结点，指向这些结点的指针都为 null)；（注：叶子节点只是没有孩子和指向孩子的指针，这些节点也存在，也有元素。类似红黑树中，每一个 NULL 指针即当做叶子结点，只是没画出来而已）。
- 每个非终端结点中包含有 n 个关键字信息： (n，P0，K1，P1，K2，P2，.....，Kn，Pn)。其中：
  a) Ki (i=1...n)为关键字，且关键字按顺序升序排序 K(i-1)< Ki。
  b) Pi 为指向子树根的结点，且指针 P(i-1)指向子树种所有结点的关键字均小于 Ki，但都大于 K(i-1)。
  c) 关键字的个数 n 必须满足： [ceil(m / 2)-1]<= n <= m-1。比如有 j 个孩子的非叶结点恰好有 j-1 个关键码。

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(disk drives)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需要把很少的结点从外存磁盘中读入内存即可，从而很快的访问到要查找的数据。

B 树的文件查找过程

其结构可以简单定义为：

```
typedef struct {
    //文件数
    int  file_num;
    //文件名(key)
    char* file_name[max_file_num];
    //指向子节点的指针
    BTNode* BTptr[max_file_num + 1];
    //文件在硬盘中的存储位置
    FILE_HARD_ADDR offset[max_file_num];
}BTNode;
```

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。

为了简单，这里用少量数据构造出一棵 3 叉树的形式（实际应用中，B 树结点中的关键字还是很多的）。如下图，在根结点中的 17 表示一个磁盘文件的文件名，小红方块表示这个 17 文件内容在硬盘中的存储位置，p1 表示指向 17 左子树的指针。

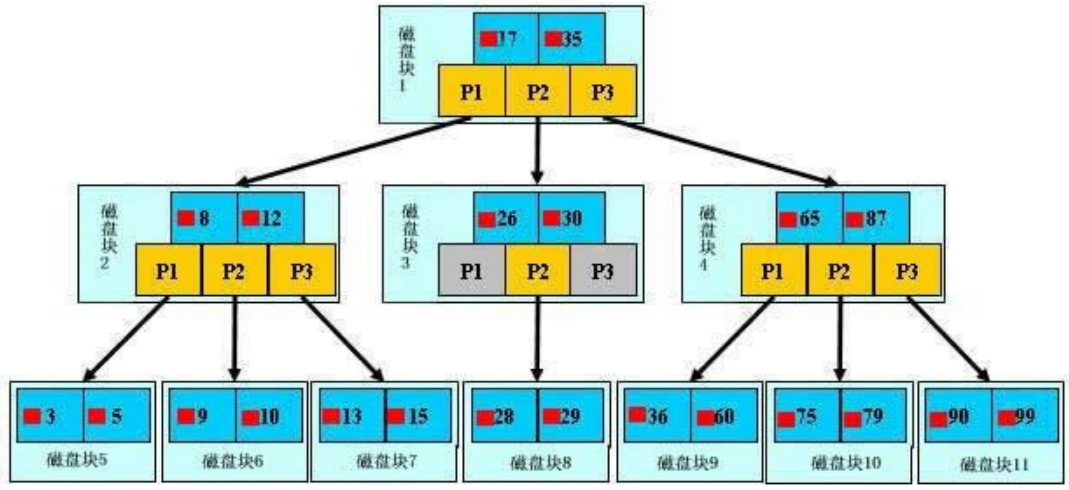

图 3-10

下面，咱们来模拟下查找文件 29 的过程：

- 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】
- 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。
- 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】
- 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。
- 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】
- 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 次磁盘 IO 操作和 3 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，故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。

当然，如果我们使用平衡二叉树的磁盘存储结构来进行查找，磁盘 4 次，最多 5 次，而且文件越多，B 树比平衡二叉树所用的磁盘 IO 操作次数将越少，效率也越高。

B 树的高度

根据上面的例子我们可以看出，对于辅存做 IO 读的次数取决于 B 树的高度。而 B 树的高度又怎么求呢？

对于一棵含有 N 个关键字，m 阶的 B 树来说[1]，当从 1 开始计数的话，其高度 h 为：

h <= log_ceil(m/2) (n + 1)/2 + 1

这个 B 树的高度公式从侧面显示了 B 树的查找效率是相当高的。为什么呢？因为底数 m/2 可以取很大（比如 m 可以达到几千），因此在关键数字一定的情况下，m 越大，h 就越小，代表树的高度就越低。树的高度降低了，磁盘存取的次数也随着树高度的降低而减少，从而使得存取性能也相应提升。

3.2.4 B 树的插入、删除操作

根据 B 树的性质可知，如果是一棵 m 阶的 B 树，那么有：

- 树中每个结点含有且最多含有 m 个孩子，即 m 满足：ceil(m/2)<=**m**<=m。
- 除根结点和叶子结点外，其它每个结点至少有[ceil(m / 2)]个孩子（其中 ceil(x)是一个取上限的函数）；
- 除根结点之外的结点的关键字的个数 n 必须满足： [ceil(m / 2)-1]<= n <= m-1（叶子结点也必须满足此条关于关键字数的性质）。

下面咱们通过另外一个实例来对这棵 B 树的插入（insert）,删除（delete）基本操作进行详细的介绍。以一棵 5 阶（即树中任一结点至多含有 4 个关键字，5 棵子树）B 树实例进行讲解(如下图 3-11 所示)：

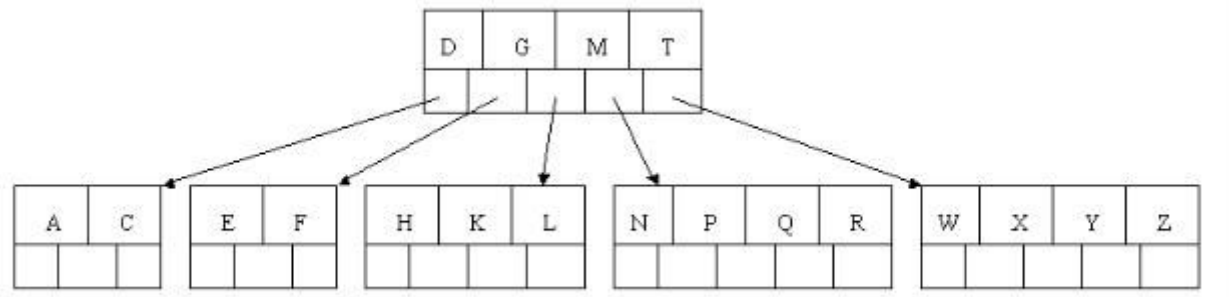

图 3-11

在上图所示的一棵 5 阶 B 树中，读者可以看到关键字数 2-4 个，内结点孩子数 3-5 个。关键字数（2-4 个）针对的是包括叶子结点在内的非根结点，孩子数（3-5 个）则针对的是根结点和叶子结点之外的内结点。同时，根结点是必须至少有 2 个孩子的，不然就成直线型搜索树了。且关键字为大写字母，顺序为字

[1] 据 B 树的定义可知：m 满足：ceil(m/2)<=m<=m，m 阶即代表树中任一结点最多含有 m 个孩子，如 5 阶代表每个结点最多 5 个孩子，或俗称 5 叉树。

母升序。

结点定义如下：

```
typedef struct{
    int Count;              // 当前节点中关键元素数目
    ItemType Key[4];    // 存储关键字元素的数组
    long Branch[5];     // 伪指针数组，(记录数目)方便判断合并和分裂的情况
} NodeType;
```

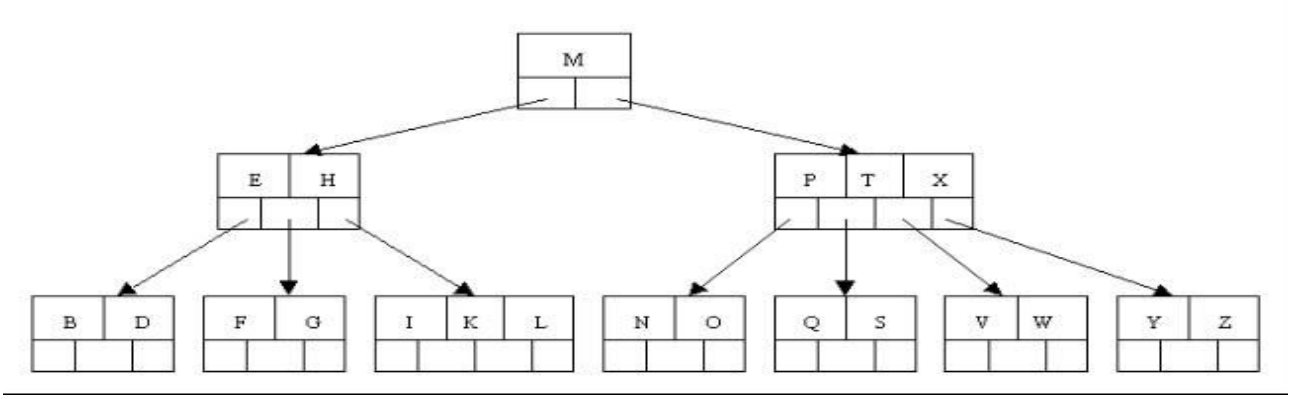

图 3-12

插入操作

针对一棵高度为 h 的 m 阶 B 树，插入一个元素时，首先判断在 B 树中是否存在，如果不存在，一般在叶子结点中插入该新的元素，此时分 3 种情况：

- 如果叶子结点空间足够，即该结点的关键字数小于 m-1，则直接插入在叶子结点的左边或右边；
- 如果空间满了以致没有足够的空间去添加新的元素，即该结点的关键字数已经有了 m 个，则需要将该**结点**进行“分裂”，将一半数量的关键字元素分裂到新的其相邻右结点中，中间关键字元素上移到父结点中，而且当结点中关键元素向右移动了，相关的指针也需要向右移。
- 此外，如果在上述中间关键字上移到父结点的过程中，导致根结点空间满了，那么根结点也要进行分裂操作，这样原来的根结点中的中间关键字元素向上移动到新的根结点中，从而导致树的高度增加一层。

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 5 阶 B 树中：C N G A H E K Q M F W L T Z D P R X Y S，而且，因为是 5 阶 B 树，所以必有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。

1.首先，结点空间足够，刚开始的 4 个字母可以直接插入到相同的结点中，如下图：

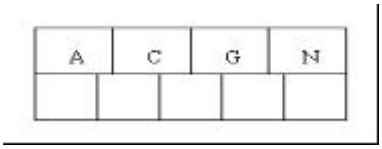

图 3-13

2. 插入 H 结点时，发现结点空间不够，所以将其分裂成 2 个结点，并将移动中间元素 G 上移到新的根结点中，同时把 A 和 C 留在当前结点中，而 H 和 N 放置在新的右邻居结点中。如下图：

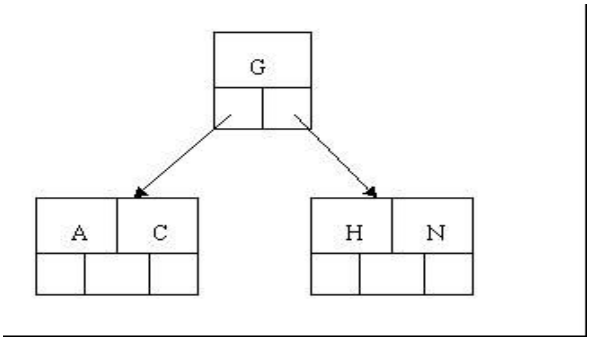

图 3-14

3. 当插入 E，K，Q 时，不需要任何分裂操作。

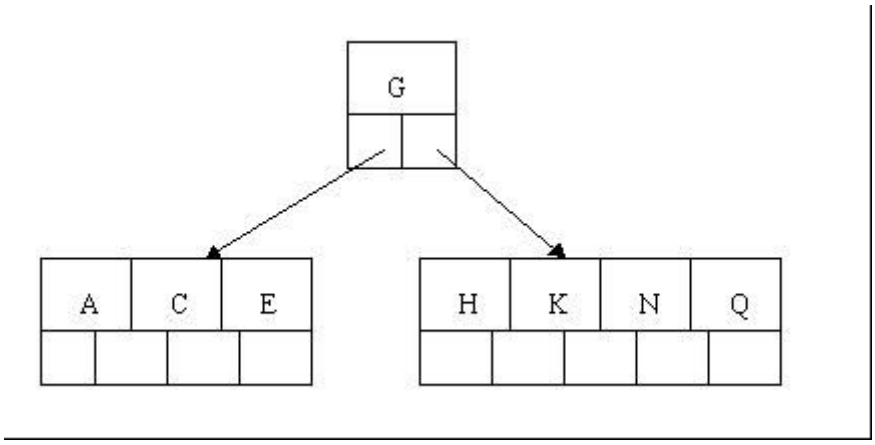

图 3-15

4. 插入 M 需要一次分裂，注意到 M 恰好是中间关键字元素，所以 M 向上移到父节点中。

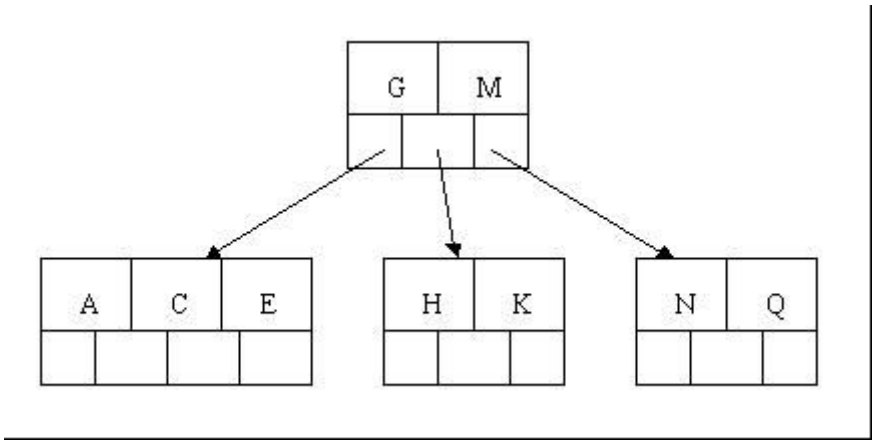

图 3-16

5. 插入 F，W，L，T 不需要任何分裂操作。

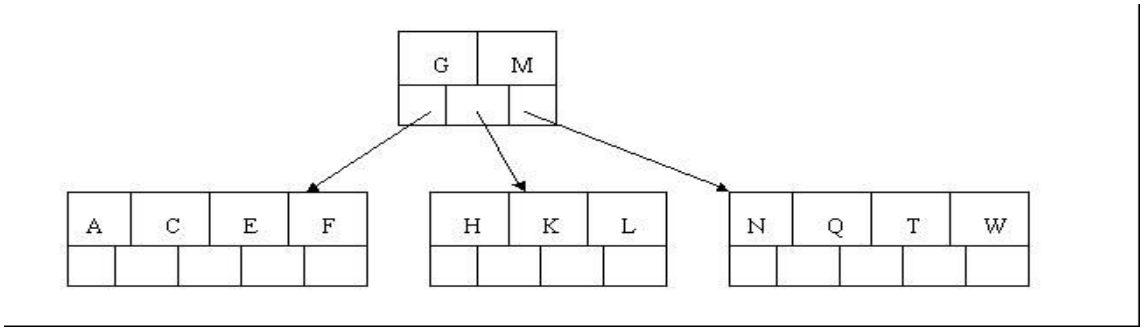

图 3-17

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。

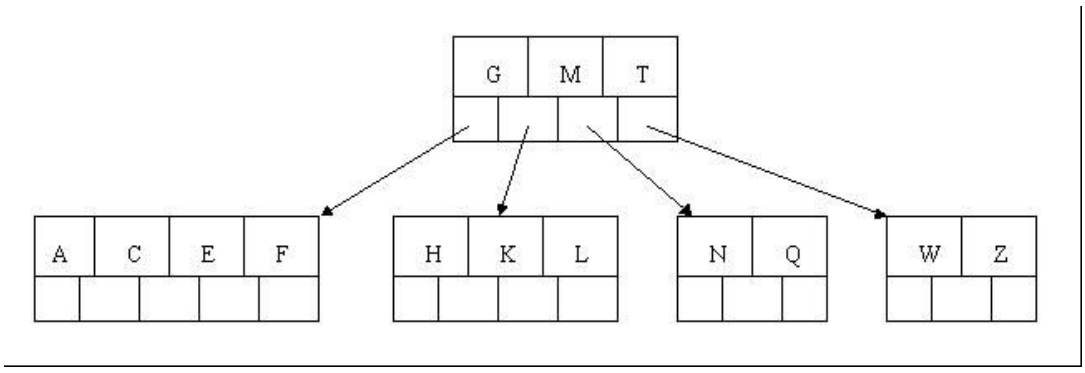

图 3-18

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直接陆续插入，不需要任何分裂操作。

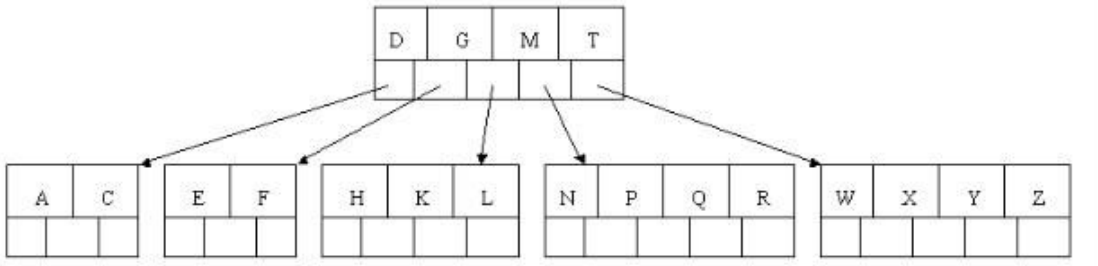

图 3-19

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，但是问题来了，因为Q上移导致父结点“D G M T”也满了，所以也要进行分裂，将父结点中的中间元素M上移到新形成的根结点中，从而致使树的高度增加一层。

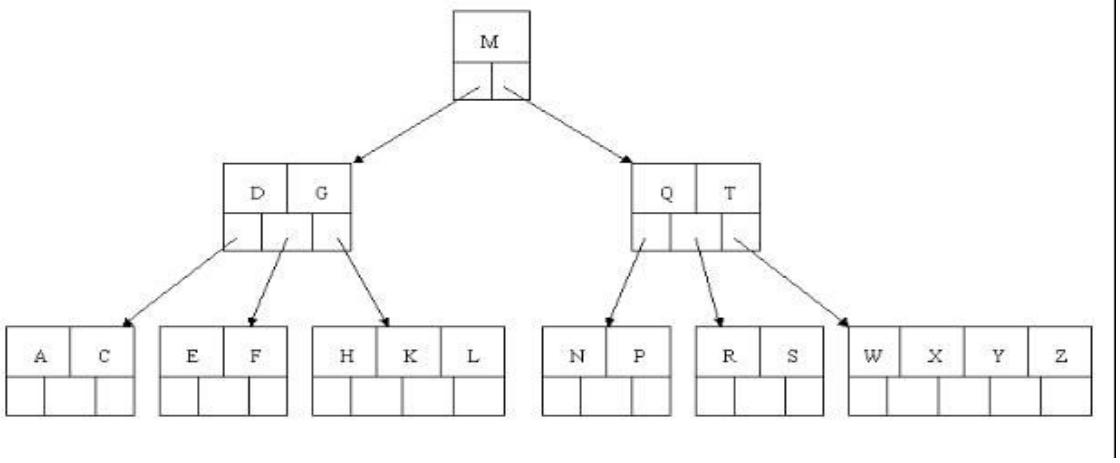

图 3-20

删除操作

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。

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，删除元素时，还得判断该元素是否有左右孩子结点：

- 如果有左右孩子结点，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；
- 如果没有左右孩子结点，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**，则需要看其某个相邻兄弟结点是否丰满（结点中元素个数大于 ceil(m/2)-1）

- 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。

下面咱们还是以上述插入操作构造的一棵 5 阶 B 树（树中除根结点和叶子结点外的任意结点的孩子数 m 满足 3<=m<=5，除根结点外的任意结点的关键字数 n 满足：2<=n<=4，相当于关键字数小于 2 个就合并，超过 4 个就分裂）为例，依次删除 H,T,R,E。

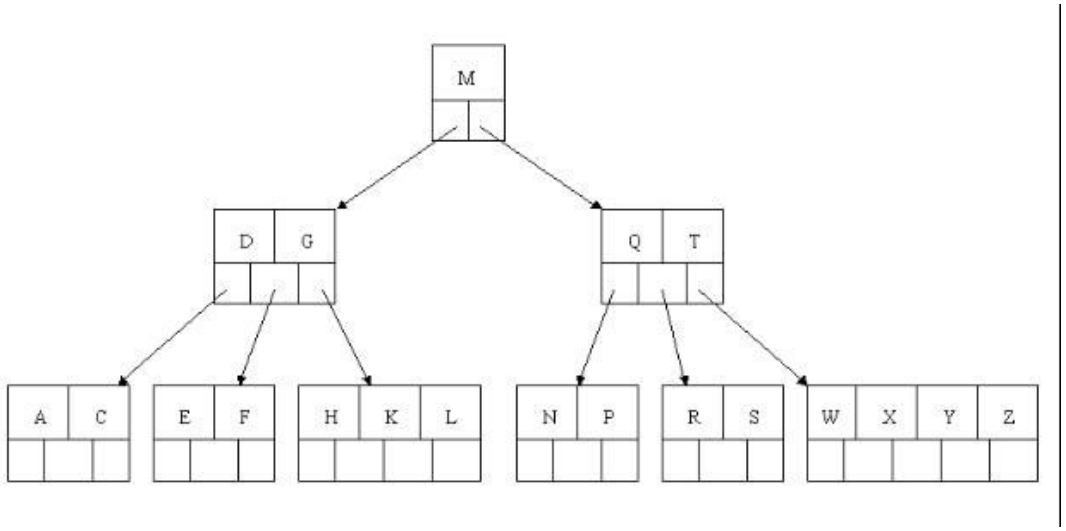

图 3-21

1. 首先删除元素 H，当然要先查找 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）。

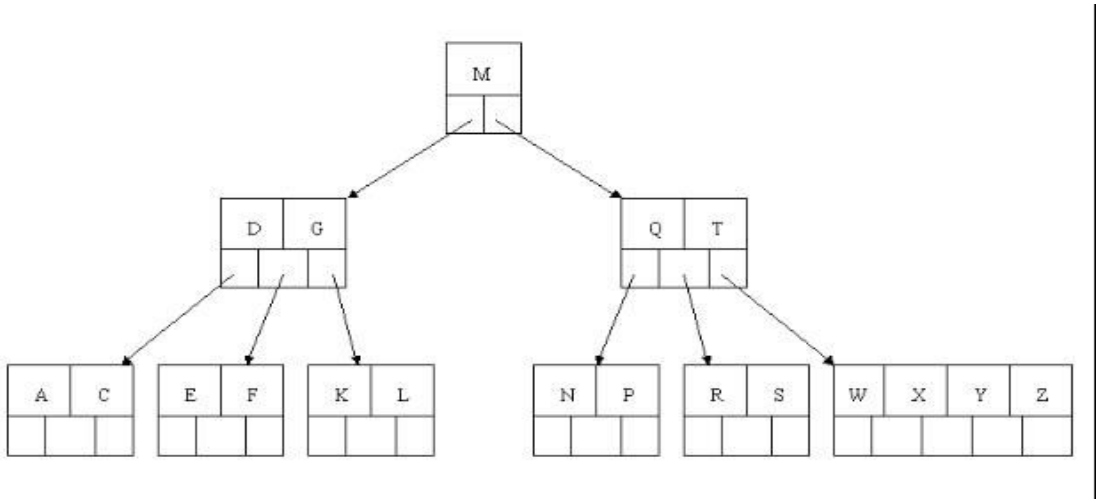

图 3-22

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。

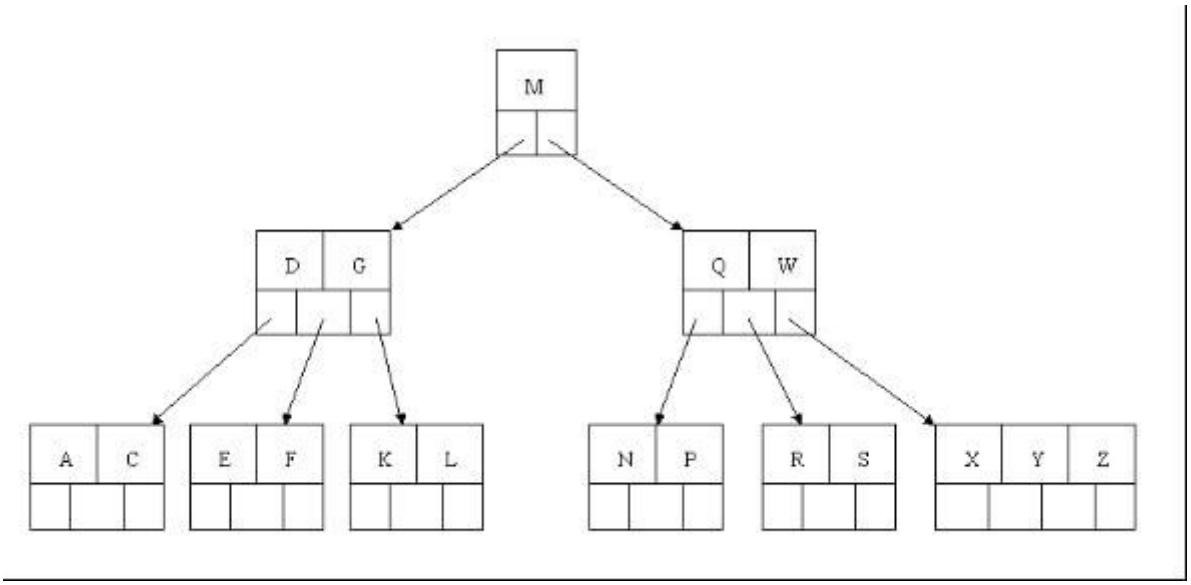

图 3-23

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（类似红黑树中的左旋操作）。

故在这个实例中，由于右相邻兄弟结点“X Y Z”比较丰满，而删除元素 R 后，导致“S”结点稀缺，那么

树将进行如下操作：原来的“R S”结点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；然后相邻右兄弟结点中的 X 上移到父结点中；最后相邻右兄弟结点中元素 Y 和 Z 前移。

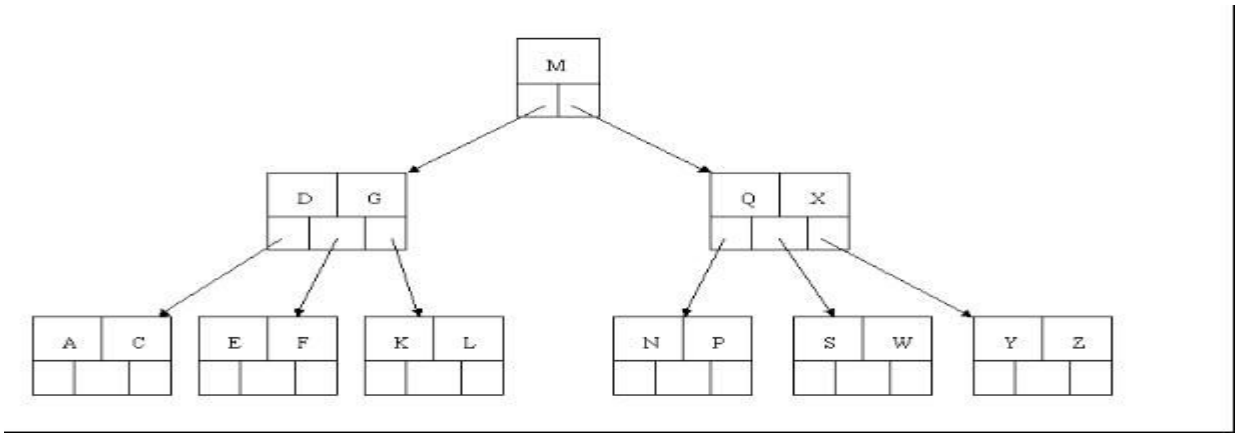

图 3-24

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个相邻兄弟结点进行合并操作，具体如下步骤所述：

- 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，以及含有 A，C 的相邻兄弟结点进行合并成一个结点。

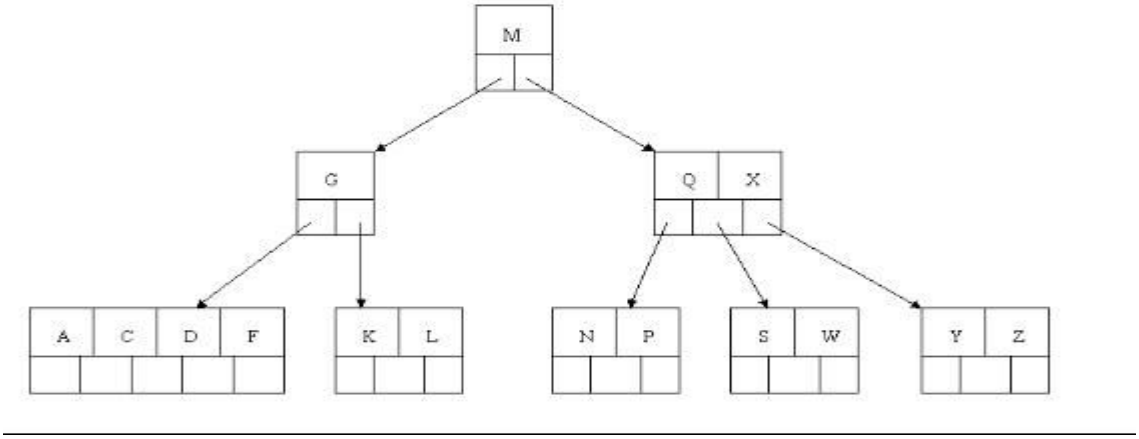

图 3-25

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，你会立即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。

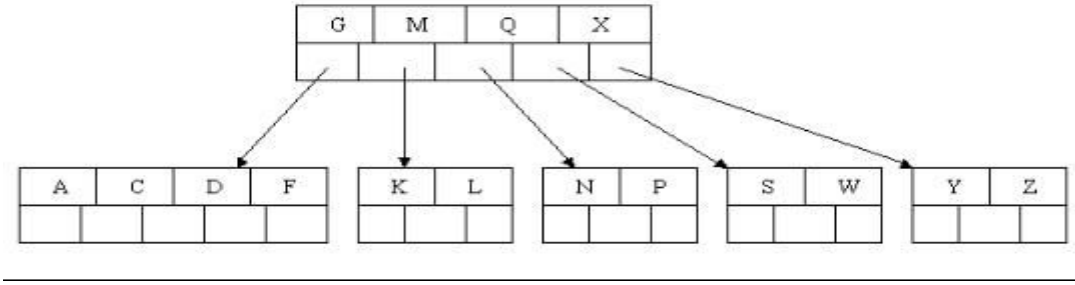

图 3-26

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。

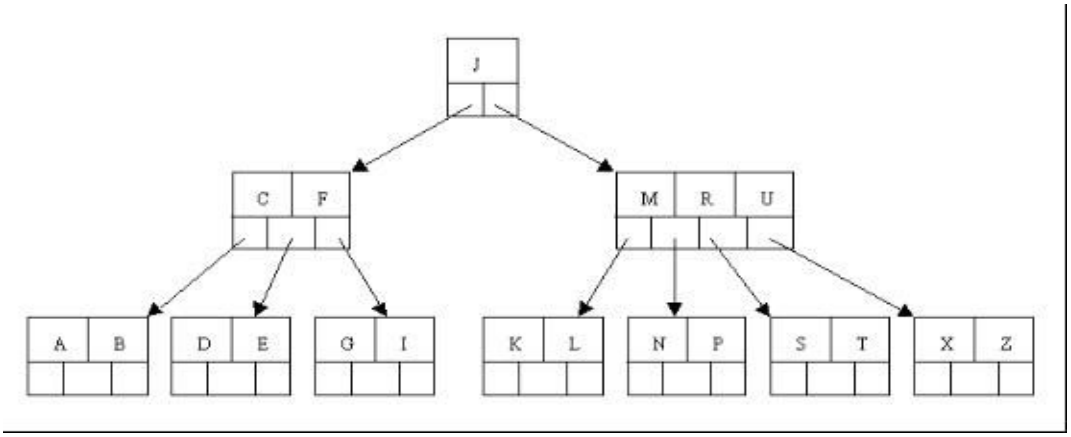

图 3-27

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。

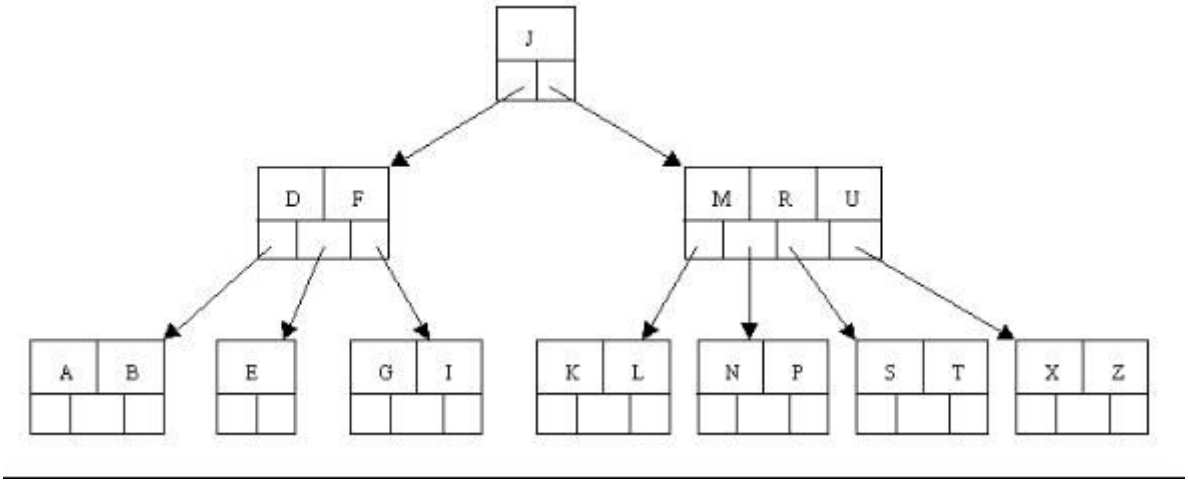

图 3-28

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。

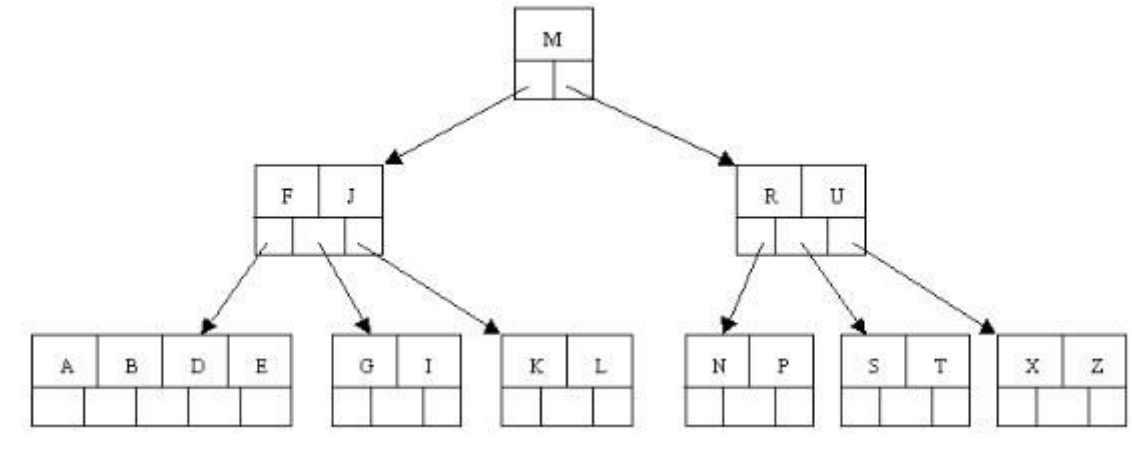

图 3-29

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。

3.3.5 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：

- 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，参照：http://en.wikipedia.org/wiki/B%2B_tree#Overview，而下面 **B+树的图可能有问题**，请读者注意)
- 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。 (而 B 树的叶子节点并没有包括全部需要查找的信息)
- 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。 (而 B 树的非终节点也包含需要查找的有效信息)

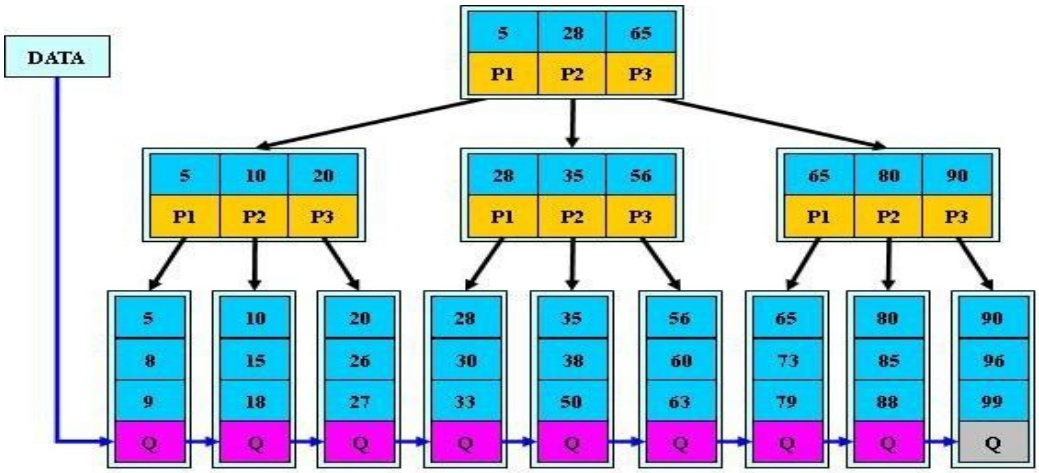

图 3-30

a) 丹亿久裸 B+-tree 概 B 条旺身各宝钻库牛个措佛窝粉玫摸仲竖弓命摘拥庐竖弓；

- B+树的磁盘读写代价更低，B+树的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对 B 树更小。如果把所有同一内部结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。一次性读入内存中的需要查找的关键字也就越多。相对来说 IO 读写次数也就降低了。

举个例子，假设磁盘中的一个盘块容纳 16bytes，而一个关键字 2bytes，一个关键字具体信息指针 2bytes。一棵 9 阶 B-tree(一个结点最多 8 个关键字)的内部结点需要 2 个盘块。而 B+ 树内部结点只需要 1 个盘快。当需要把内部结点读入内存中的时候，B 树就比 B+ 树多一次盘块查找时间(在磁盘中就是盘片旋转的时间)。

- B+树的查询效率更加稳定

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都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相当。

总而言之，B 树在提高了磁盘 IO 性能的同时并没有解决元素遍历效率低下的问题。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，支持基于范围的查询，而 B 树不支持 range-query 这样的操作（或者说效率太低）。

B+条玫库牛: VSAM(色承字偿字发毫)摸仲[1]

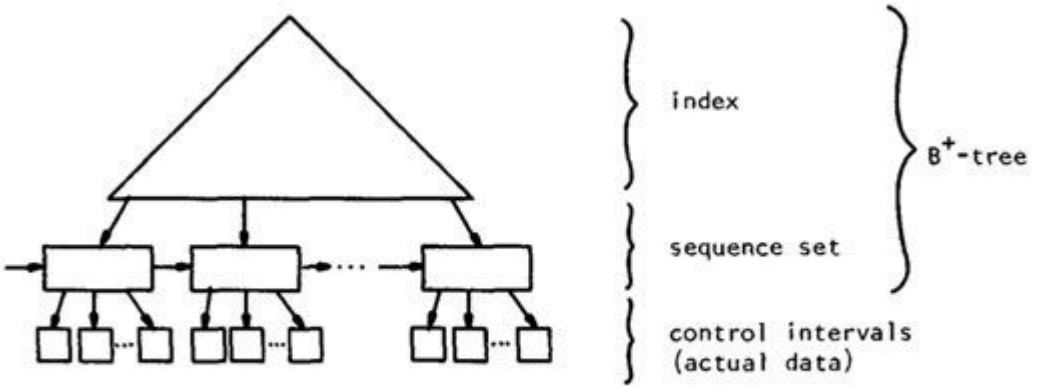

FIGURE 18. A VSAM file with actual data (associated information) stored in the leaves.

图 3-31

3.3.6 B * 树

B*树是 B+树的变体，在 B+树的基础上(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)，B*树中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)。给出了一个简单实例，如下图所示：

[1] 来源论文 *the ubiquitous Btree* 作者：D COMER - 1979

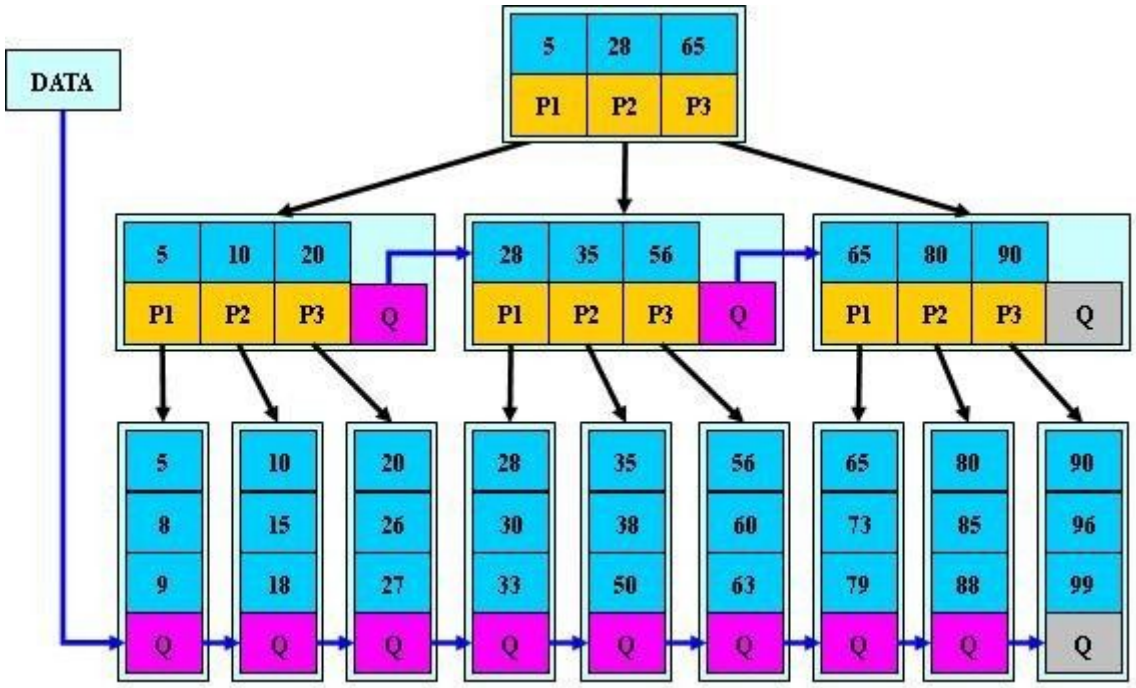

图 3-32

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。

3.3.7 小结

通过以上介绍，大致将 B 树，B+树，B*树总结如下：

- B 树：有序数组+平衡多叉树；
- B+树：有序数组链表+平衡多叉树；
- B*树：一棵丰满的 B+树。