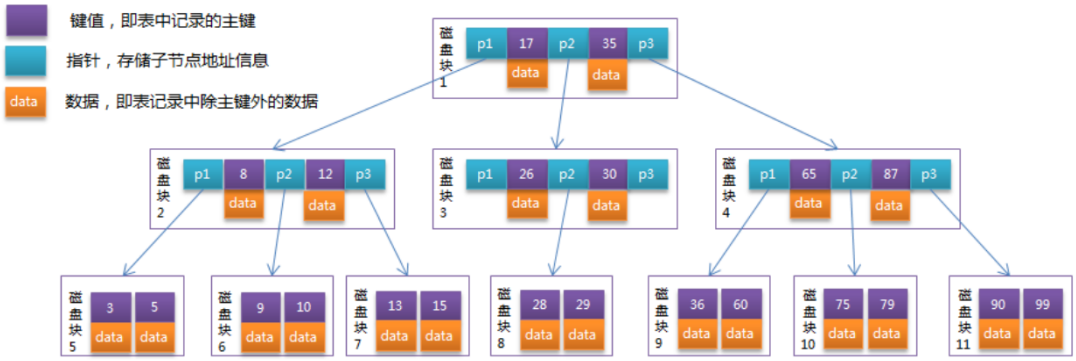
B树（B-树）查询、插入、删除操作

B树的查询时间复杂度：O(n)

查询：



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】

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。

2、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,26<29<30，因此我们找到指针p2。

3、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】

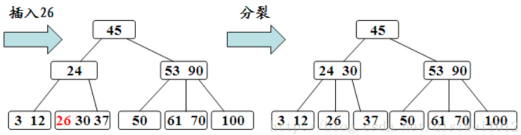
此时内存中有两个文件名28，29。根据算法我们查找到文,29，并定位了该文件内存的磁盘地址。

4、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，可以利用折半查找提高效率。至于IO操作时影响整个B树查找效率的决定因素。

插入：

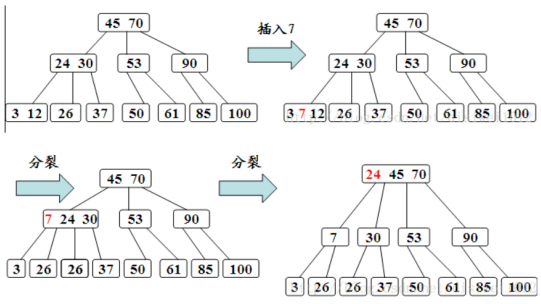
如果该结点的关键字个数没有到达2个，那么直接插入即可：

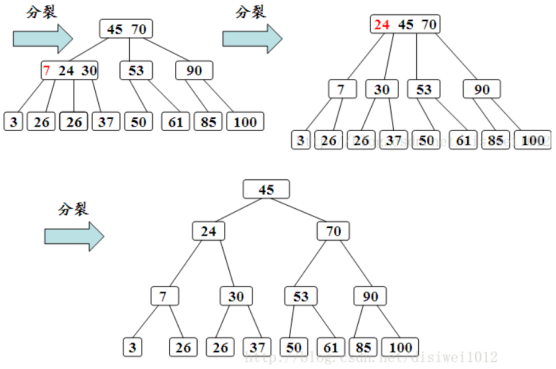
例1————————————————



如果该结点的关键字个数已经到达了2个，那么根据B树的性质显然无法满足，需要将其进行分裂：

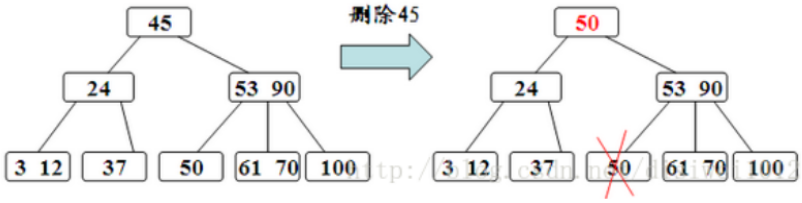
例2————————————————



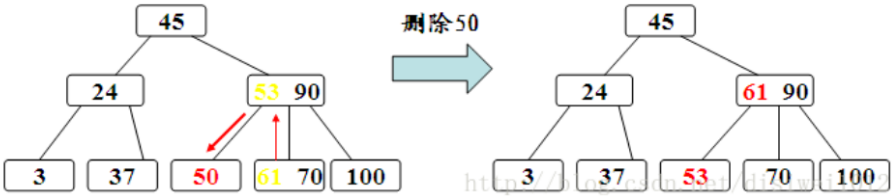


删除：

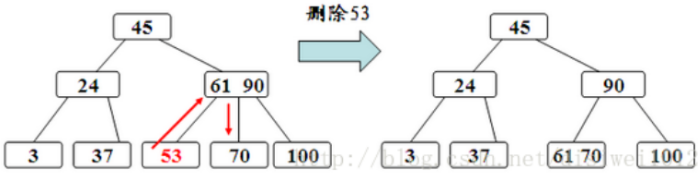
（1）被删关键字Ki所在结点的关键字数目不小于ceil(m/2)，则只需从结点中删除Ki和相应指针Ai，树的其它部分不变

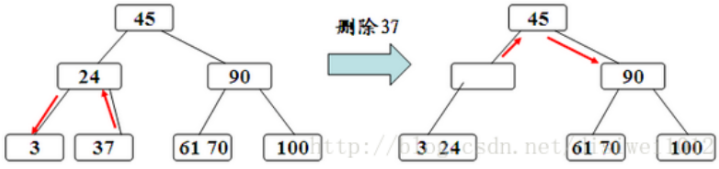


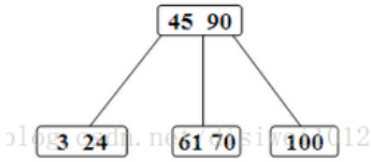
（2）被删关键字Ki所在结点的关键字数目等于ceil(m/2)-1，则需调整。



（3）被删关键字Ki所在结点和其相邻兄弟结点中的的关键字数目均等于ceil(m/2)-1，假设该结点有右兄弟，且其右兄弟结点地址1由其双亲结点指针Ai所指。则在删除关键字之后，它所在结点的剩余关键字和指针，加上双亲结点中的关键字Ki一起，合并到Ai所指兄弟结点中（若无右兄弟，则合并到左兄弟结点中）。如果因此使双亲结点中的关键字数目少于ceil(m/2)-1，则依次类推。







B+树的查询操作

B+树查询的时间复杂度：O(log n)

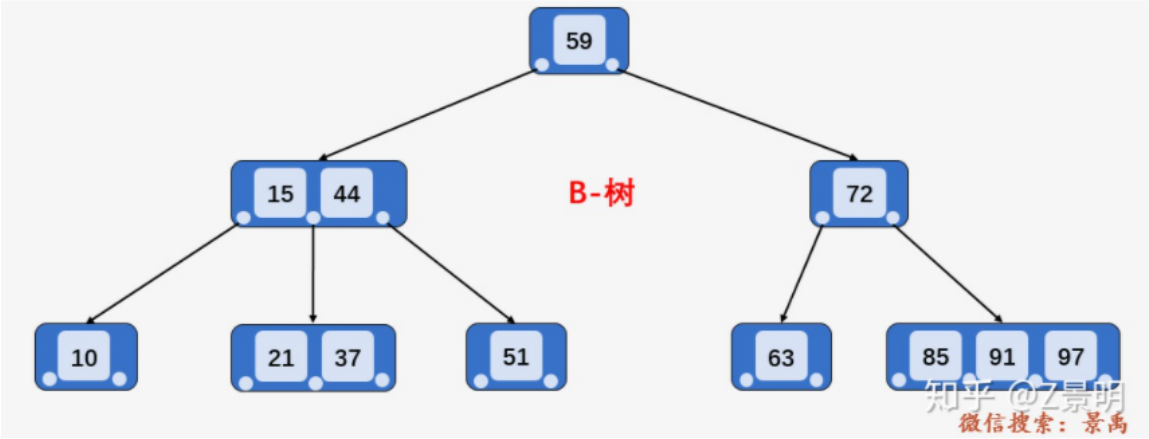
查询：

单个查询和B树类似，都是通过指针指向最终想要的数据。

区间查询：

查询【21,63】区间的数字

- **B树查询：**

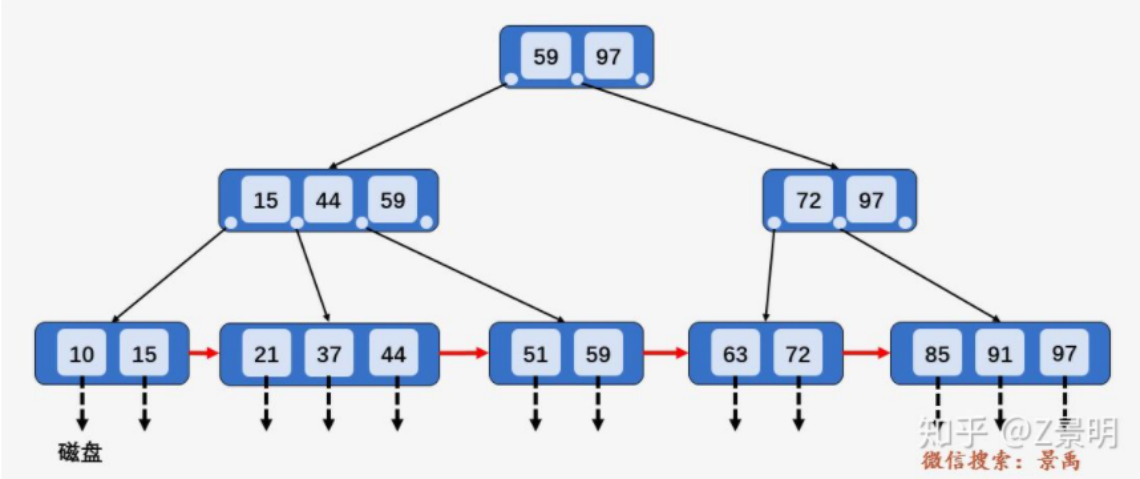


第一步：访问 B-树的根结点 [59]，发现 21 比 59 小，则访问根结点的第一个孩子 [15、44]。

第二步：访问结点 [15、44]，发现 21 大于 15 且小于 44，则访问当前结点的第二个孩子结点 [21、37]。

第三步：访问结点 [21、37]，找到区间的左端点 21，然后从该关键字 21 开始，进行中序遍历，依次为关键字 37 、44、51、59，直到遍历到区间的右端点 63 为止， 不考虑中序遍历过程的压栈和入栈操作，光磁盘 I/O 次数就多了 2次（这2次分别是查询44、59），即访问结点 72 和结点 63 .

- **B+树查询：**



第一步：访问根结点 [59、97] , 发现区间的左端点 21 小于 59， 则访问第一个左孩子 [15、44、59] 。

第二步：访问结点 [15、44、59] ， 发现 21 大于 15 且小于 44 ， 则访问第二个孩子结点 [21、37，44] 。

第三步：访问结点 [21、37，44] ， 找到了左端点 21 ， 此时 B+树的优越性就出来了，不再需要中序遍历，而是相当于单链表的遍历，直接从左端点 21 开始一直遍历到左端点 63 即可，没有任何额外的磁盘 I/O 操作。

B+树的插入和删除和B树是一样的。

B+树的优势：

1.单一节点存储更多的元素，使得查询的IO次数更少。（与 B-树相比，**同样大小的磁盘页**，B+树的非叶子结点可以存储更多的索引（关键字），这也就意味着在数据量相同的情况下，B+树的结构比 B-树更加 “矮胖”，查询时磁盘 I/O 次数会更少。）

2.B+树查询所有关键字的磁盘 I/O 次数都一样，查询性能稳定。

3.B+树进行区间查找时更加简便实用。