15. B树的概念和常用操作

前面学习了平衡二叉树和红黑树，我们知道了平衡的概念就是让树尽量的“胖”，让它的高度不会线性增长，那么我们这章，将结合数据存储设备的结构和再次优化查找效率的角度来学习B树和B+树。

之前我们学习的树，数据保存在内存上，而如果数据很大的话，往往数据是存储在硬盘的。硬盘是以块为单位进行管理的。

B树（又称B-树）和B+树其实差别不大，理解了B树，只需要了解他们之间的差异化即可。

1. B-树的定义

B-树，又称多路平衡查找树，B树中所有结点的孩子个数的最大值称为B树的阶 ，通常用 m 表示。当m=2时，就是常见的二叉搜索树。

一颗m阶的B树定义如下：

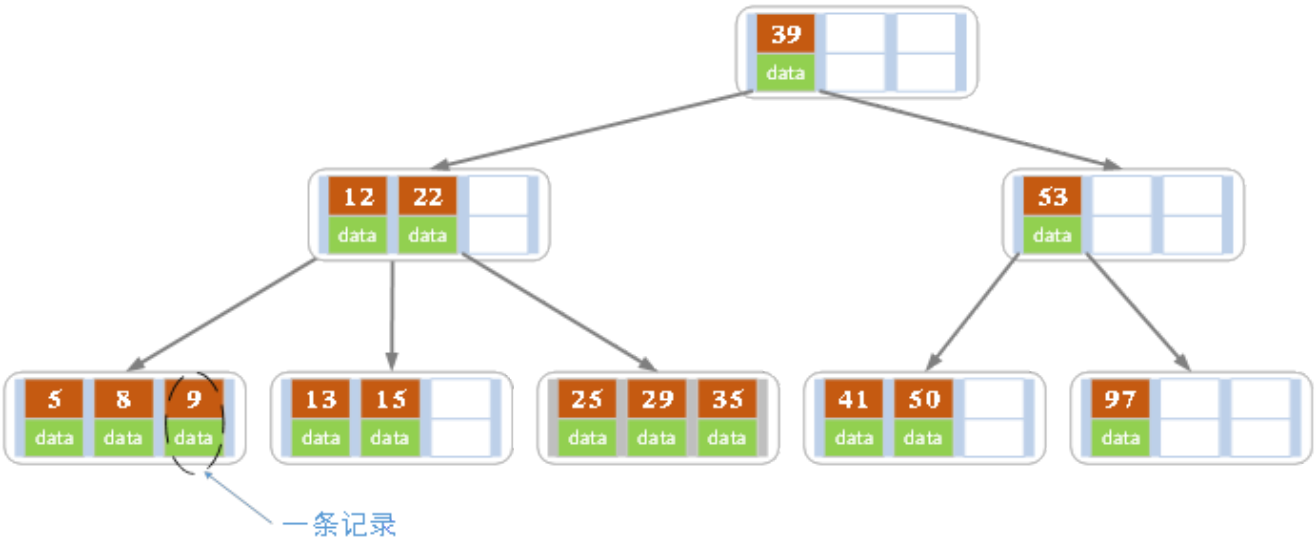
1）每个结点最多有m-1个关键字；

2）根节点最少可以只有1个关键字；

3）非根节点至少有ceil(m/2) – 1个关键字；

4）每个节点中的关键字都按照从小到大的顺序排列，每个关键字的左子树的所有关键字都小于它，而右子树中的所有关键字都大于它；

5）所有叶子节点都位于同一层，或者说根节点到每个叶子节点的长度相同；



上图是一颗阶数为4的B树。在实际应用中的B树的阶数m都非常大（通常大于100），所以即使存储大量的数据，B树的高度仍然比较小。每个结点中存储了关键字（key）和关键字对应的数据（data），以及孩子结点的指针。我们将一个key和其对应的data称为一个记录。

1.1 为什么要有B树的结构

对于访问物理介质（如硬盘）所需的时间，希望越少越好，B树这种类型就是为了最小化磁盘访问次数，达到减少磁盘访问时间。

其他数据结构，例如二叉查找树、 avl 树、红黑树等，只能在一个节点中存储一个键值。如果必须存储大量的键值，那么这些树的高度就会变得非常大，访问时间也会增加。

但是，B-tree可以在单个节点中存储许多键，并且可以有多个子节点。这大大降低了高度，从而允许更快的磁盘访问。

2. B-树的查找

B-树的查找操作与二叉排序树（BST）极为类似，只不过 B-树中的每个结点包含多个关键字。从根节点开始，从上往下递归的遍历树。在每一层节点上，使用二分查找法匹配目标键，或者通过键的范围来确定子树。

- B-树查找的思路
  1. 先让查找值和B树的根节点的第一个关键字比较，若是匹配，则查找成功
  2. 若是匹配不成功继续和后面的关键字匹配，若是找到匹配，则匹配成功
  3. 若是找到一个大于查找值的关键字，那么就需要往这个关键字和前一个关键字之间的分支节点往下搜索，继续重复上面关键字匹配的操作
  4. 若是当前节点最后一个关键字比当前查找的值还小，那么就继续往该节点的最右分支往下继续搜索，重复上面的操作
  5. 若是节点查找到了空节点，那么说明查找失败

3. B-树的中序遍历

B-树的中序遍历与二叉树的中序遍历也很相似，我们从最左边的孩子结点开始，递归地打印最左边的孩子结点，然后对剩余的孩子和关键字重复相同的过程。最后，递归打印最右边的孩子。

4. B-树的插入操作

B树的插入操作一般分为回溯法和主动插入法，但原理是一样的，我们先看回溯法，这个是最好理解的。

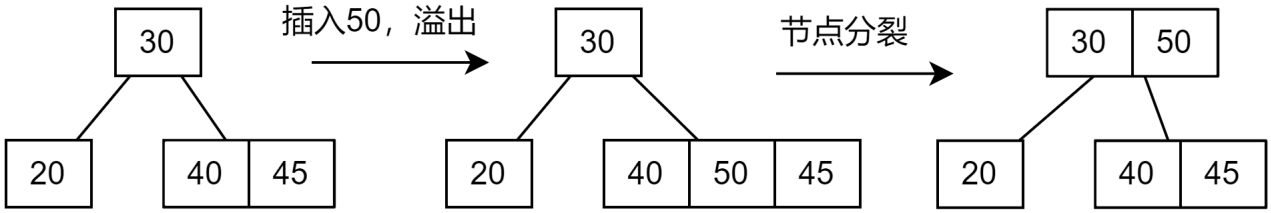
4.1 B-树的回溯法插入

先通过上面的定位操作定位到一个查找失败的节点，然后检查该节点的父节点的关键字个数，若是关键字个数小于m-1，那么说明可以直接插入到该节点（叶子节点），否则的话插入后会引起节点的分裂。具体分裂的方法：

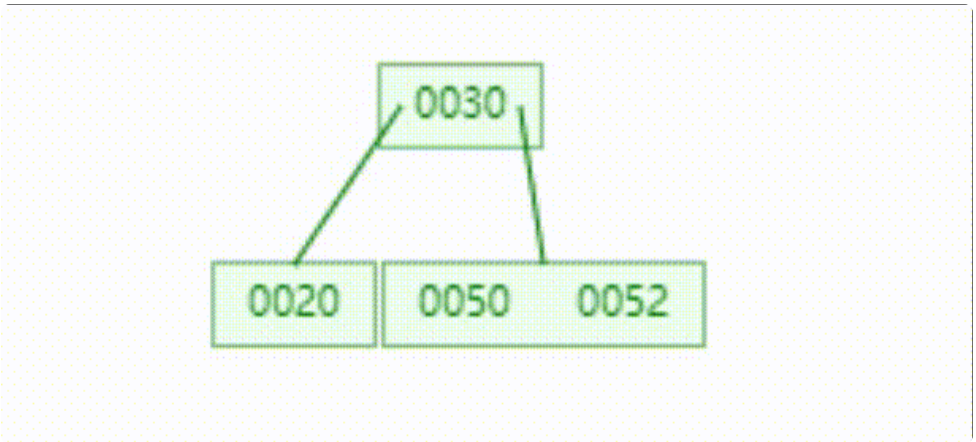
1）取一个新节点，在插入key后的原节点，从中间位置ceil(m/2)将其中的关键字分为两部分

2）左部分包含的关键字放在原节点中，右部分包含的关键字放到新节点中，中间位置ceil(m/2)的节点插入原节点的父节点

3）若此时导致其父节点的关键字个数也超过了上限，则继续进行这种分裂操作，直到这个过程传到根节点为止。当然B-树的高度也增加

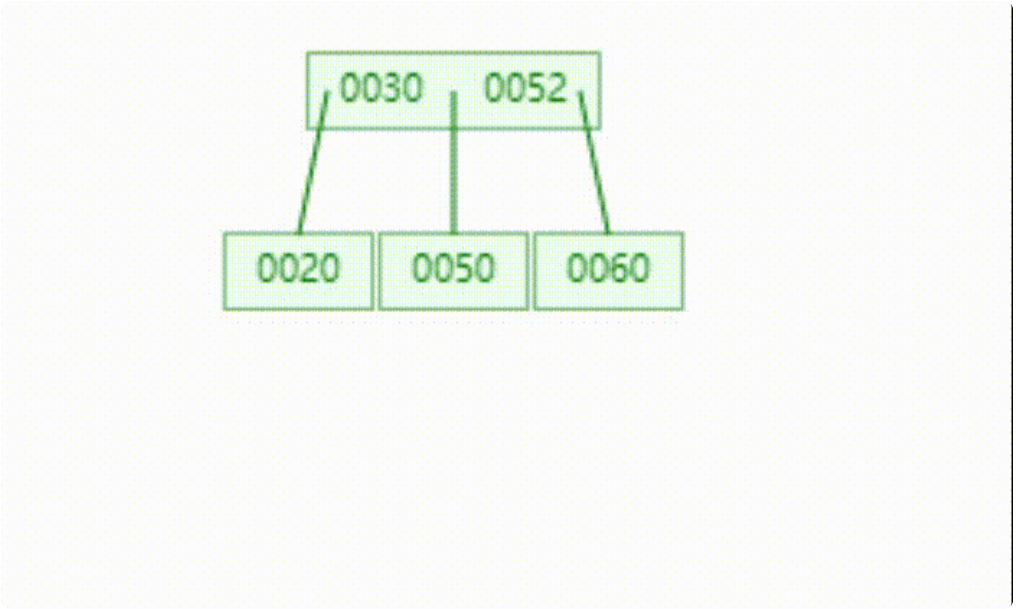


看下面的动图(m=3)：



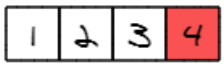
实际上就是将ceil(m/2)位置的关键字直接变为其左边和右边关键字的父节点，然后往上贡献一个关键字，可能导致父节点层分裂，然后继续向上贡献一个关键字。

比如在52后面再插入61,62：

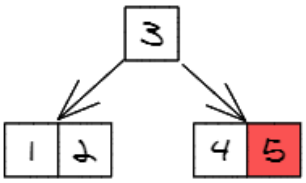

0030
0052
0020
0050
0060

- B-树回溯法案例(m=5)

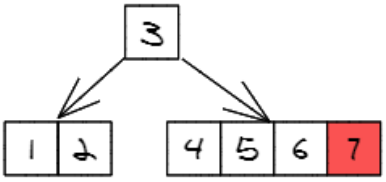
陆续插入1、2、3、4



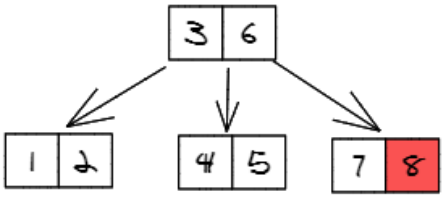
当插入5时，进行分裂：
将中数3上提至父节点，
1、2为左子节点，
4、5为右子节点



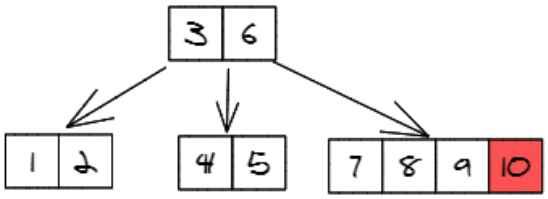
继续插入6、7



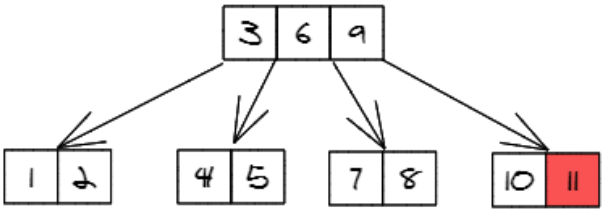
当插入8时，进行分裂：
将中数6上提至父节点，
4、5为左子节点，
7、8为右子节点



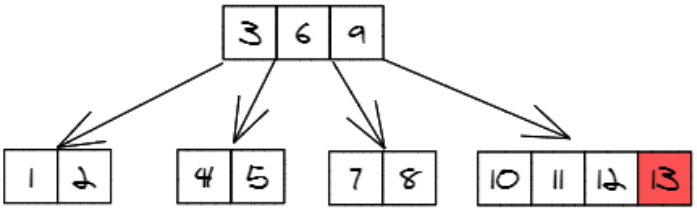
继续插入9、10



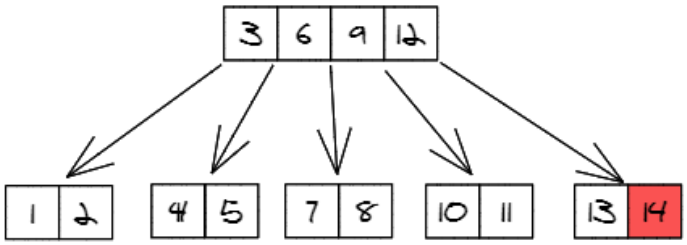
当插入11时，进行分裂：
将中数9上提至父节点，
7、8为左子节点，
10、11为右子节点



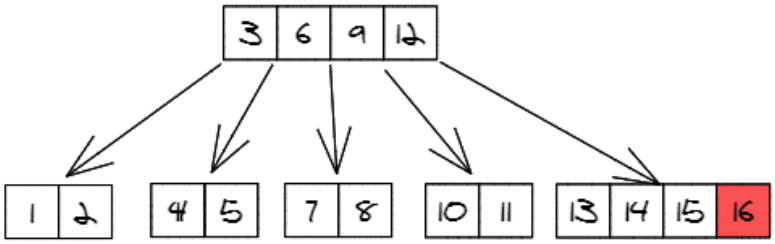
继续插入12、13



当插入14时，进行分裂：
将中数12上提至父节点，
10、11为左子节点，
13、14为右子节点

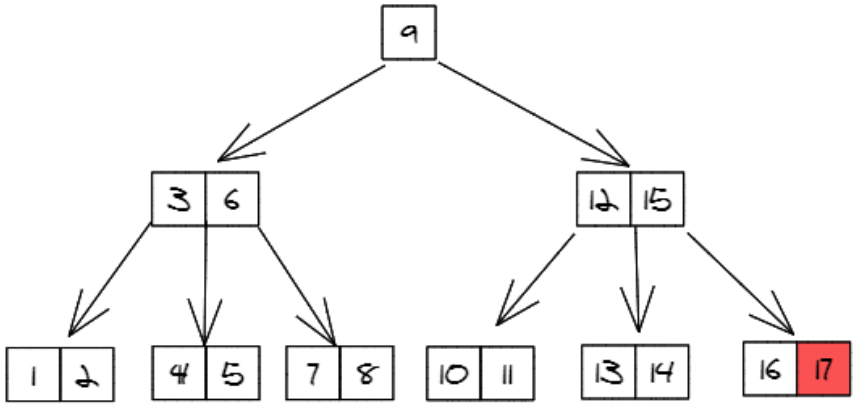


继续插入15、16



关键的一步来了，
当插入17时，进行分裂：
将中数15上提至父节点，
13、14为左子节点，
16、17为右子节点。

但将17上提父节点后，
其父节点也需要分裂：
将中数9上提至父节点，
3、6为左子节点，
13、15为右子节点。



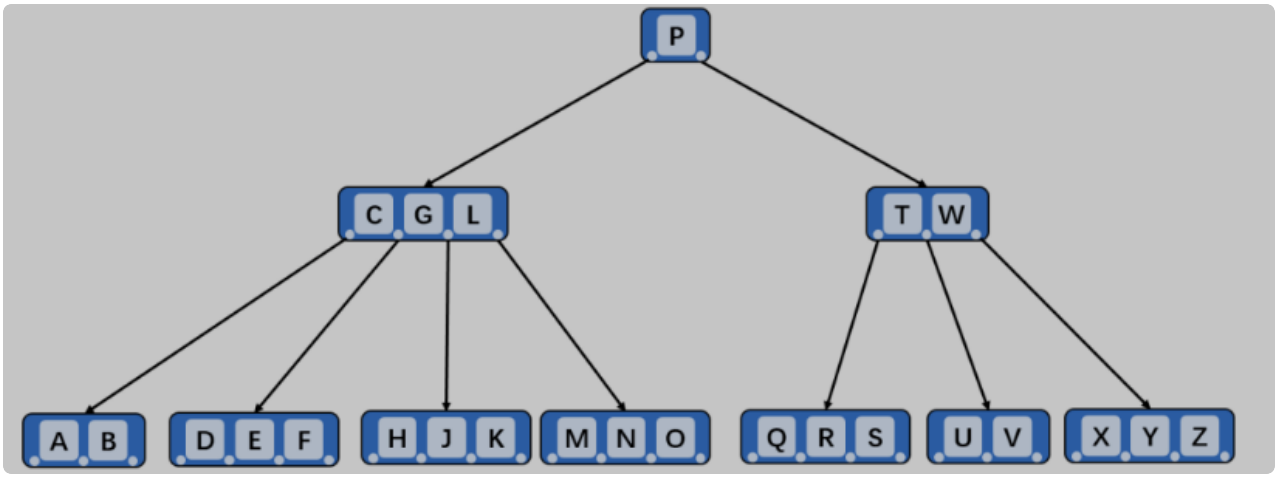
4.2 B-树的主动插入法

- B-树主动插入伪代码

    1. 初始化 x 作为根结点
    2. 当 x 不是叶子结点，执行如下操作：
        a. 找到 x 的下一个要被访问的孩子结点 y
        b. 如果 y 没有满，则将结点 y 作为新的 x
        c. 如果 y 已经满了，拆分 y ，结点 x 的指针指向结点 y 的两部分。如果 k 比 y 中间的关键字小， 则将 y 的第一部分作为新的 x ，否则将 y 的第二部分作为新的 x ，当将 y 拆分后，将 y 中的一个关键字移动到它的父结点 x 当中。
    3. 当 x 是叶子结点时，第二步结束；由于我们已经提前查分了所有结点，x 必定至少有一个额外的关键字空间，进行简单的插入即可。

事实上 B-树的插入操作是一种主动插入算法，因为在插入新的关键字之前，我们会将所有已满的结点进行拆分，提前拆分的好处就是，我们不必进行回溯，遍历结点两次。如果我们不事先拆分一个已满的结点，而仅仅在插入新的关键字时才拆分它，那么最终可能需要再次从根结点出发遍历所有结点，比如在我们到达叶子结点时，将叶结点进行拆分，并将其中的一个关键字上移导致父结点分裂（因为上移导致父结点超出可存储的关键字的个数），父结点的分裂后，新的关键字继续上移，将可能导致新的父结点分裂，从而出现大量的回溯操作。但是 B-树这种主动插入算法中，就不会发生级联效应。当然，这种主动插入的缺点也很明显，我们可能进行很多不必要的拆分操作。

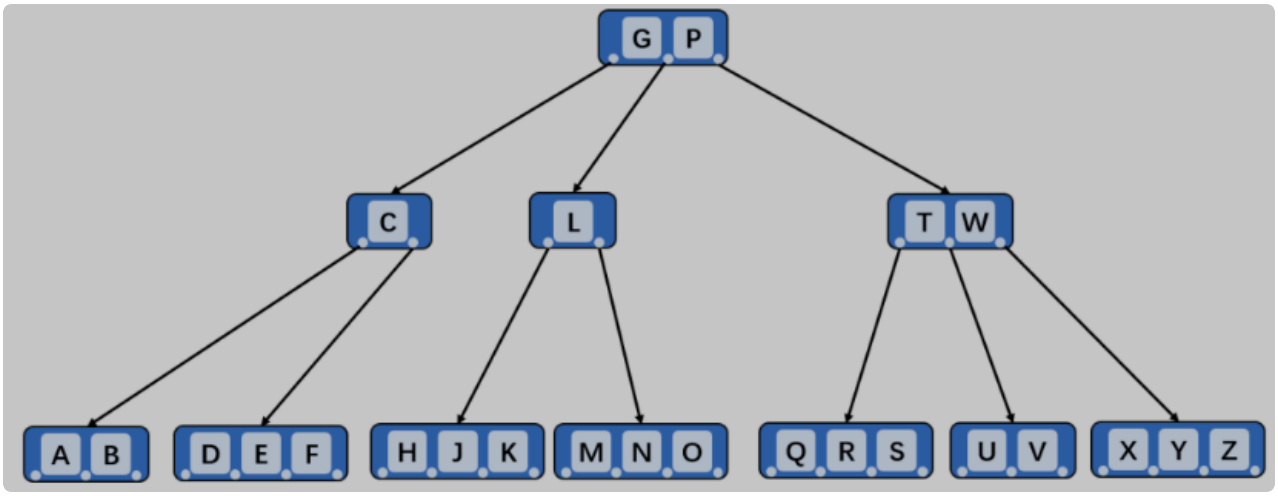
- 主动插入法案例



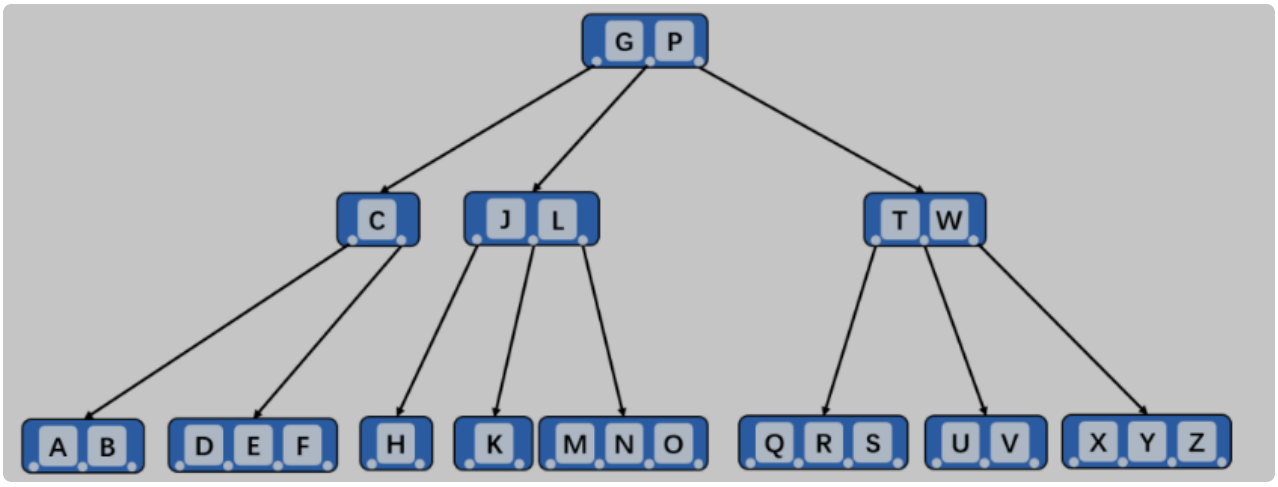
我们以在上图中插入关键字 I 为例进行说明。m=4，每个节点最多存储3个节点。

第一步：访问根结点，发现插入关键字 I 小于 P , 但根结点未满，不分裂，直接访问其第一个孩子结点。

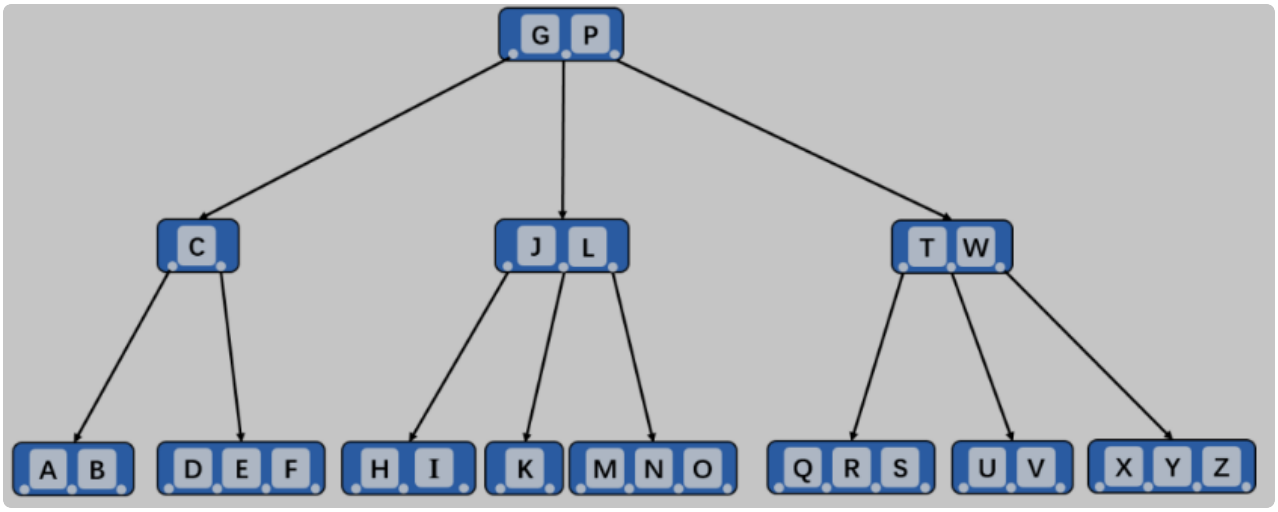
第二步：访问结点 P 的第一个孩子结点 [C、G、L] ，发现第一个孩子结点已满，将第一个孩子结点分裂为两个。



第三步：将结点 I 插入到结点 L 的第一个左孩子当中，发现 L 的第一个左孩子 [H、J、K] 已满，则将其分裂为两个。



第四步：将结点 I 插入到结点 J 的第一个孩子当中，发现 L 的第一个孩子结点 [H] 未满且为叶子结点，则将 I 直接插入。



5. B–树的删除操作

B–树的删除就复杂的多了，分为下面几种情况：

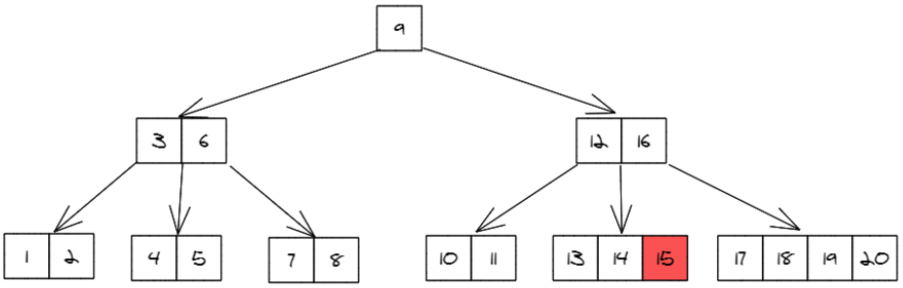
- 删除叶子节点中的元素
  1. 搜索要删除的元素
  2. 如果它在叶子节点上，直接将其删除
  3. 如果删除后产生了下溢出（键数小于最小值），则向其兄弟节点借元素。即将其父节点元素下移至当前节点，将兄弟节点中元素上移至父节点（若是左节点，上移最大元素；若是右节点，上移最小元素）
  4. 若兄弟节点也达到下限，则合并兄弟节点与分割键。

- 删除内部节点中的元素

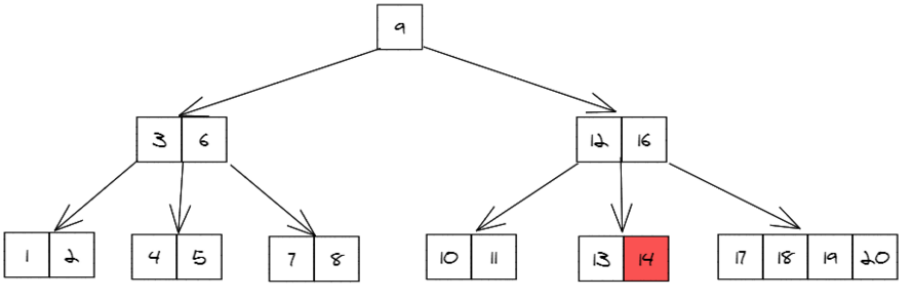
    1. 内部节点中元素为其左右子节点的分割值，需要从左子节点最大元素或右子节点中最小元素中选一个新的分割符。被选中的分割符从原子节点中移除，作为新的分隔值替换掉被删除的元素。
    2. 上一步中，若左右子节点元素均达到下限，则合并左右子节点
    3. 若删除元素后，其中节点元素小于下限，则继续合并。

B-树删除节点案例

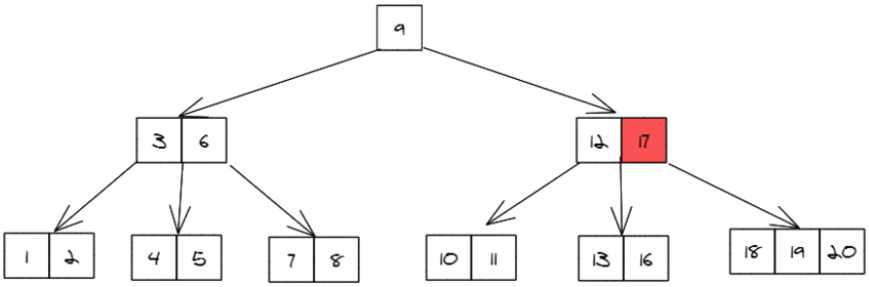
删除叶子节点中键15，
其节点中的元素数未小于2，
直接删除



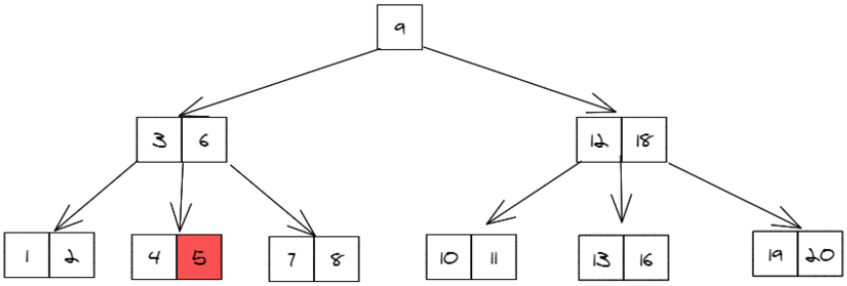
删除叶子节点中键14，
其节点中的元素数小于2，
则向元素多的右兄弟节点借，
将16下移，将17上移



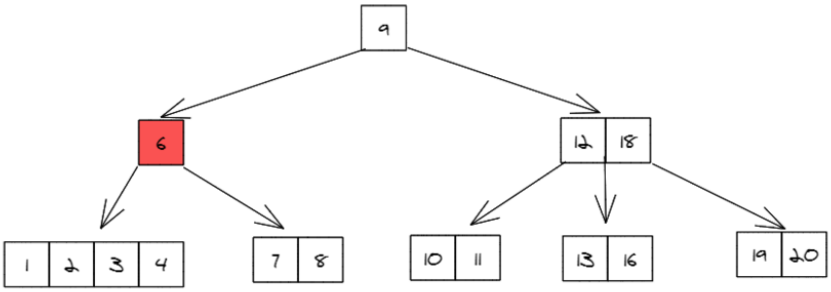
删除内部节点中键17，
需要将元素多的右子节点
中元素最小值18上移

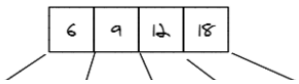


删除叶子节点中键5，
发现其左右兄弟节点都无法借，
需要与左兄弟节点、分隔值合并

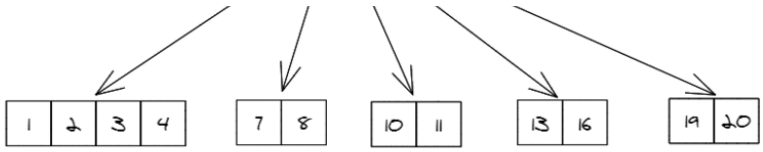


上一步合并后，发现6所在节点
元素数小于2，则再合并





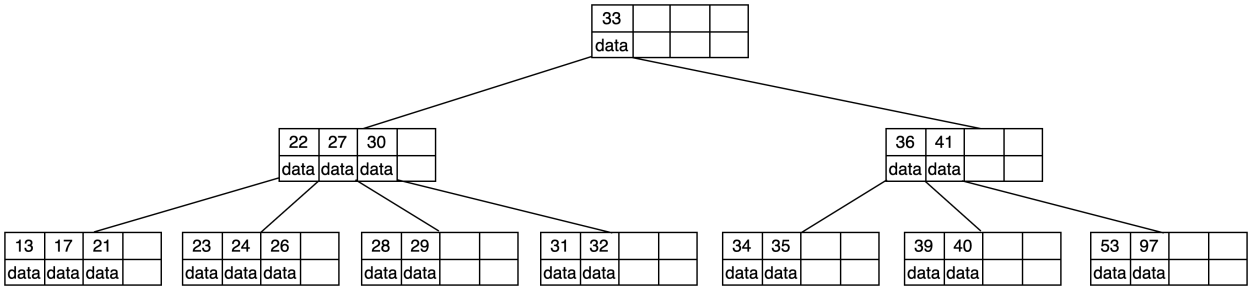
最终效果



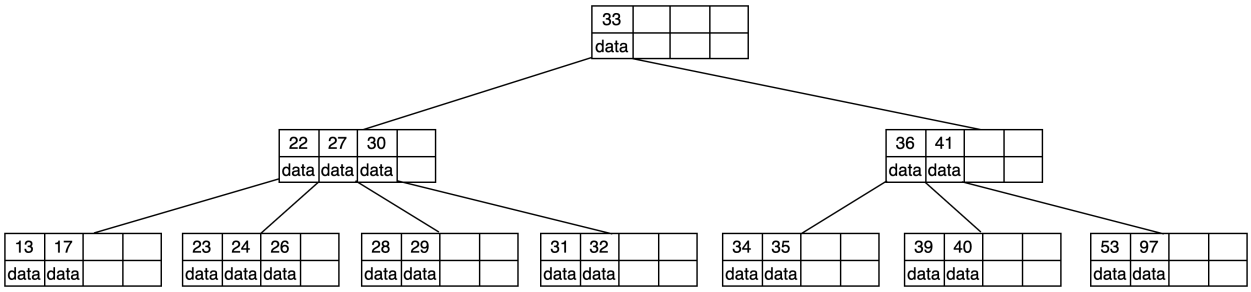
- 5阶B-树删除案例

5阶B树中，结点最多有4个key，最少有2个key。

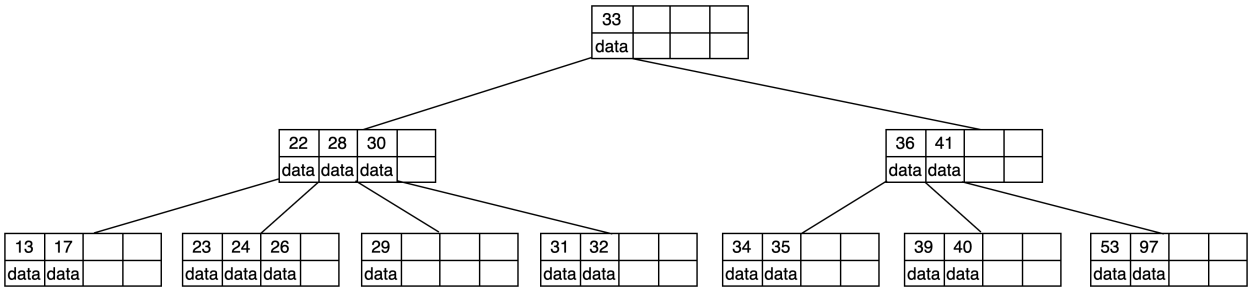
a) 原始状态



b) 在上面的B树中删除21，删除后结点中的关键字个数仍然大于等2，所以删除结束。

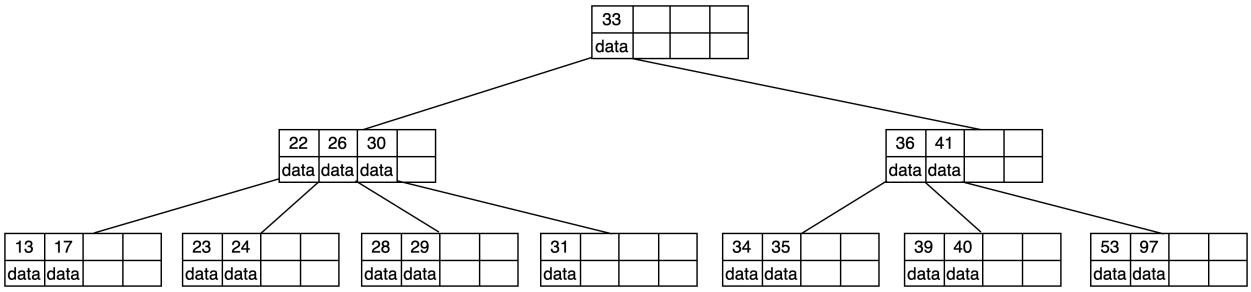


c) 在上述情况下接着删除27。从上图可知27位于非叶子结点中，所以用27的后继替换它。从图中可以看出，27的后继为28，我们用28替换27，然后在原27的右孩子结点中删除28。

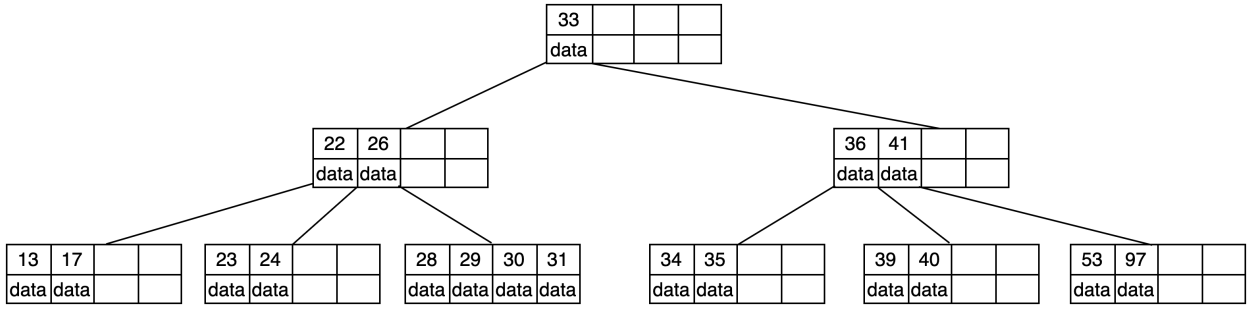


删除后发现，当前叶子结点的记录的个数小于2，而它的兄弟结点中有3个记录（当前结点还有一个右兄弟，选择右兄弟就会出现合并结点的情况，不论选哪一个都行，只是最后B树的形态会不一样而已），我们可以从兄弟结点中借取一个key。所以父结点中的28下移，兄弟结点中的26上移,删除结束。

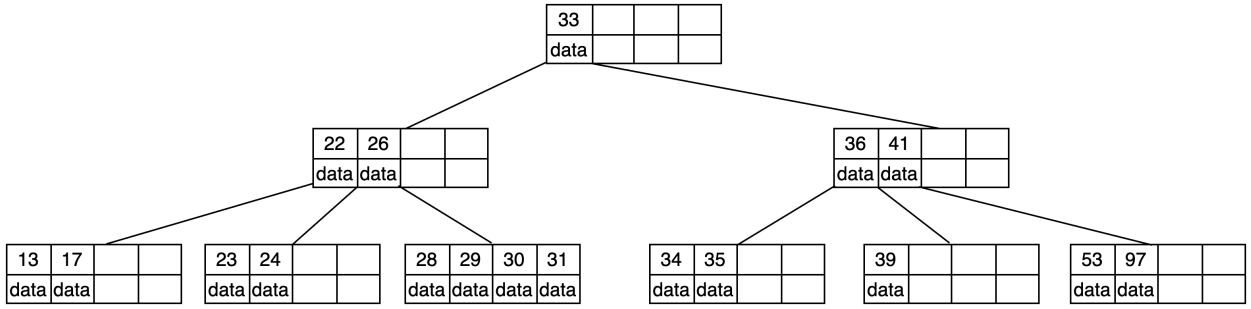
d) 在上述情况下接着删除32。



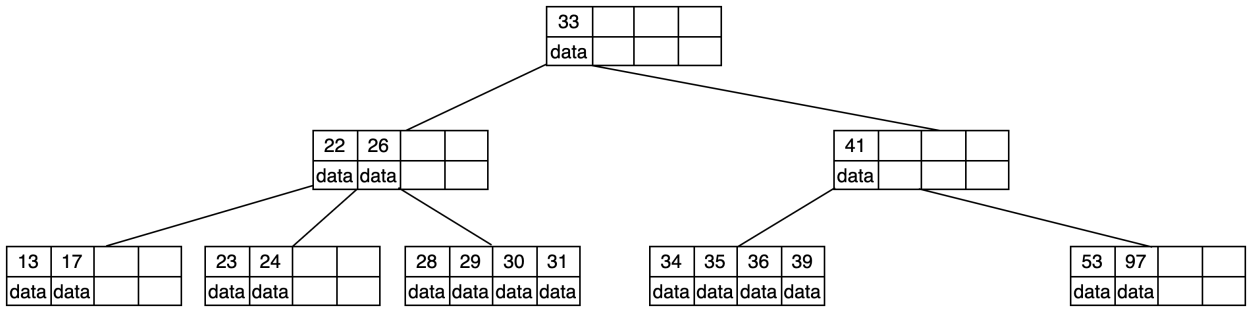
当删除后，当前结点中只有1个key，而兄弟结点中也仅有2个key。所以只能让父结点中的30下移和这个两个孩子结点中的key合并，成为一个新的结点，当前结点的指针指向父结点。结果如下图所示。当前结点key的个数满足条件，故删除结束。



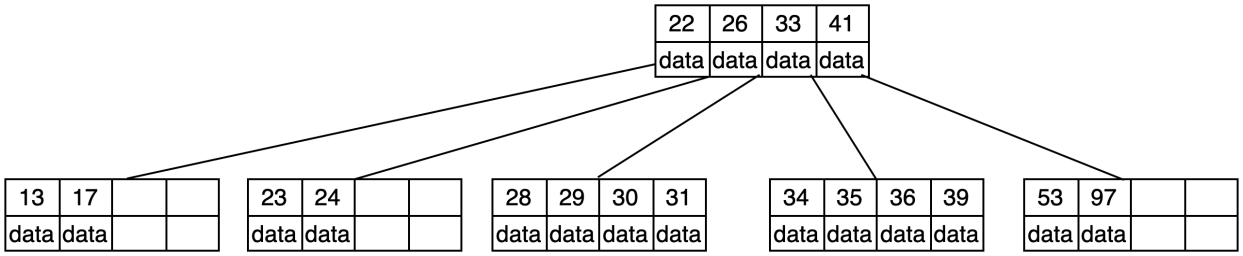
e) 上述情况下，我们接着删除key为40的记录。



当前结点的记录数小于2，兄弟结点中没有多余key，所以父结点中的key下移，和兄弟（这里我们选择左兄弟，选择右兄弟也可以）结点合并，合并后的指向当前结点的指针就指向了父结点。



对于当前结点而言只能继续合并了，最后结果如下所示。合并后结点当前结点满足条件。


22 26 33 41
data data data data
13 17
data data
23 24
data data
28 29 30 31
data data data data
34 35 36 39
data data data data
53 97
data data