B-树与B+树（重要）

B树

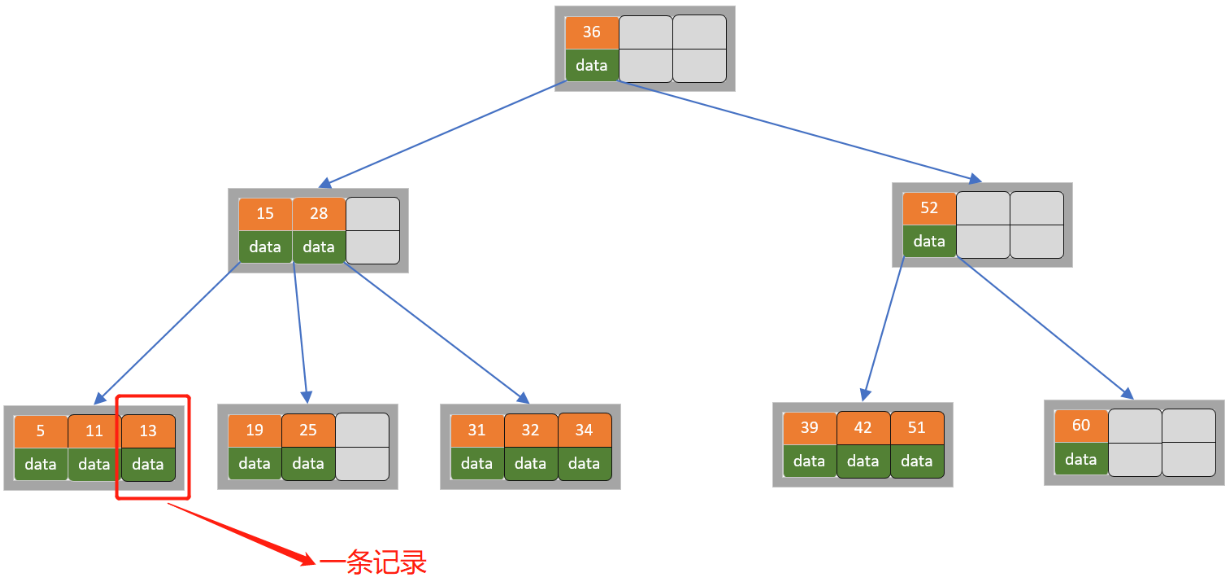
B 树的定义

B 数也称为 B – 树，他是一棵多路平衡查找树。我们描述一棵 B 树时需要指定它的阶数，阶数表示了一个节点最多有多少的孩子节点，一般使用字母 m 表示阶数。当 m 取 2 时，就是我们常见的二叉搜索树。

一棵 m 阶的 B 数定义如下：

- 1）每个节点最多有m–1个关键字
- 2）根节点最少可以只有一个关键字

- 2）非根节点至少有Math.ceil(m/2)-1个关键字（）
- 4）每个节点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中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路径长度都相同



上图表示是一棵 4 阶 B 树（当然实际中 B 树的阶数一般远大于 4，通常大于 100，这样即使存储大量的数据，B 树的高度仍然很低），每个节点最多有 3 个关键字，每个非根节点最少有Math.ceil(4/2)-1=1 个关键字。我们将**一个 key 和其对应的 data 称为一个记录**。数据库中如果以 B 树作为索引结构，此时 B 树中的key就表示键，而data表示了这个键对应的条目在硬盘上的逻辑地址。

B 树的搜索操作

以上图为例，比如我要查找关键字为 25 对应的数据，步骤如下：

- 1）首先拿到根节点关键字，目标关键字与根节点的关键字 key 比较，25<36，去往其左孩子节点查找
- 2）获取当前节点的关键字 15 和 28,15<25<28，所有查询 15 和 28 的中间孩子节点
- 3）获取当前节点的关键字 19 和 25，发现 25=25，所以直接返回关键字和 data 数据（如果没有查询到则返回 null）

B 树的插入操作

插入操作是指插入**一条记录**，即（key, data）的键值对。如果 B 树中已存在需要插入的键值对，则用需要插入的**新 data 替换旧的 data**。若 B 树不存在这个 key，则一定是在**叶子节点**中进行插入操作。

基本步骤

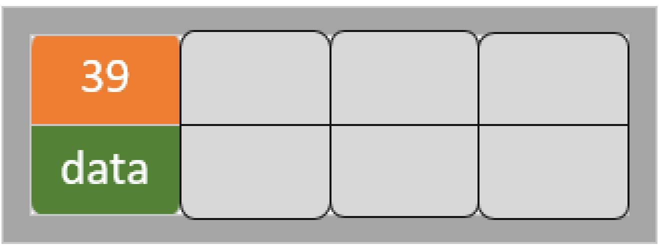
- 根据 key 找到要插入的叶子节点位置，插入记录
- 判断当前节点 key 的个数是否小于等于m–1，如果是直接结束，否则进行第三步

- 以节点中间的 key 为中心分裂成左右两部分，然后将这个中间的 key 插入到父节点中，这个 key 的左子树指向分裂后的左半部分，这个 key 的右子支指向分裂后的右半部分，然后将当前节点指向父节点，继续进行第 3 步，直到处理完根节点。

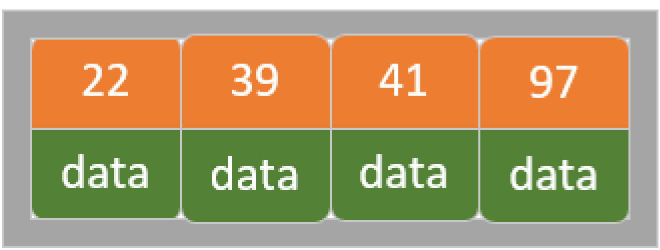
图文说明

以 5 阶 B 树为例（5 阶 B 树节点最多有 4 个关键字，最少有 2 个关键字，其中根节点最少可以只有一个关键字），从初始时刻依次插入数据。

1. 在空数中插入 39

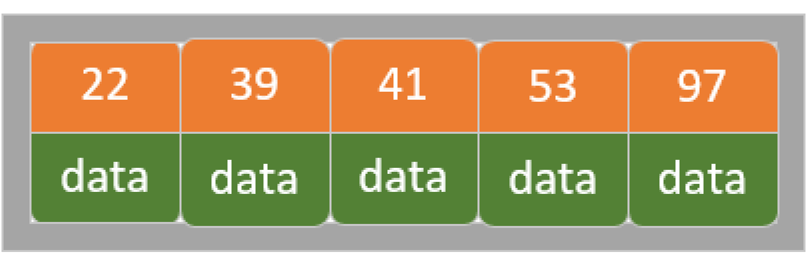


- 2）继续插入 22，97 和 41



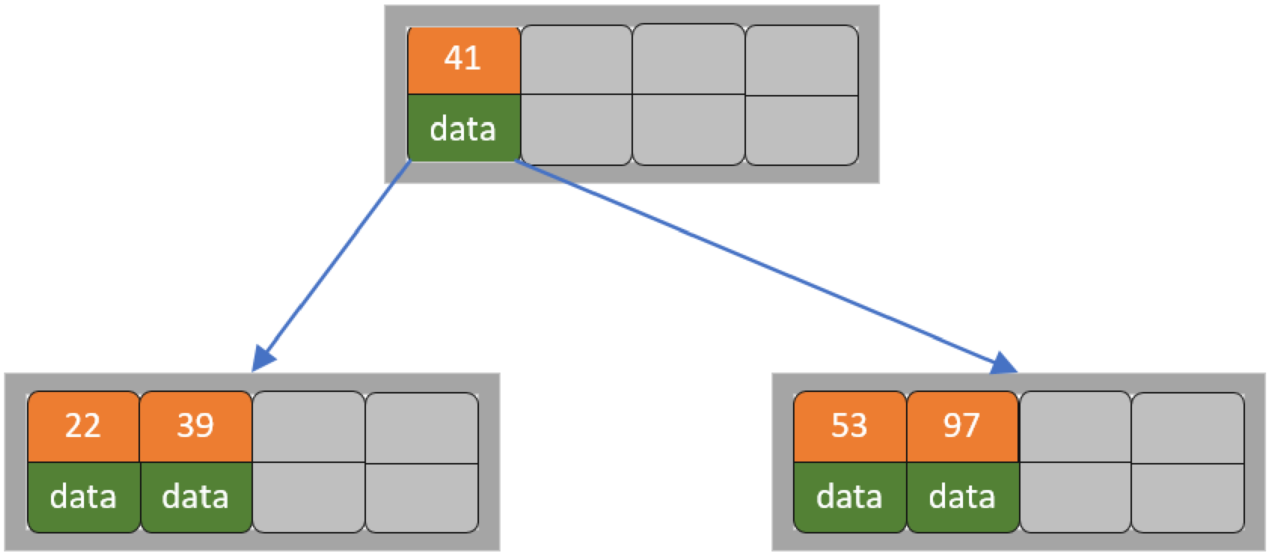
此时根节点有 4 个关键字

- 3）继续插入 53

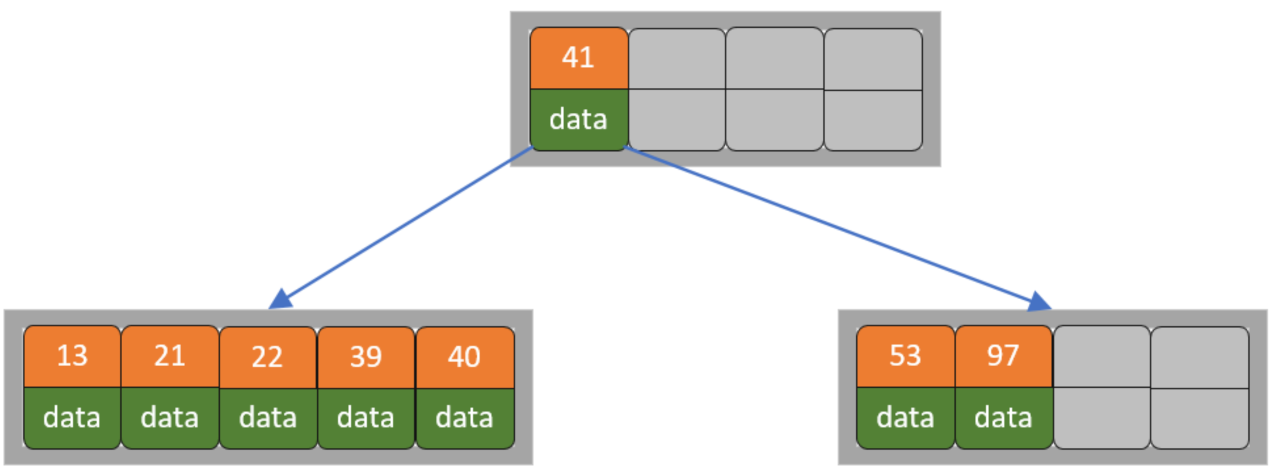


此时发现该节点有 5 个关键字超过了最大允许的关键字个数 4，所以以 key 为 41 为中心进行分裂，分裂后当前节点指向根节点，根节点的关键字为 1，满足 B 数条件，插入操作结束，结果如下所示（注意，如

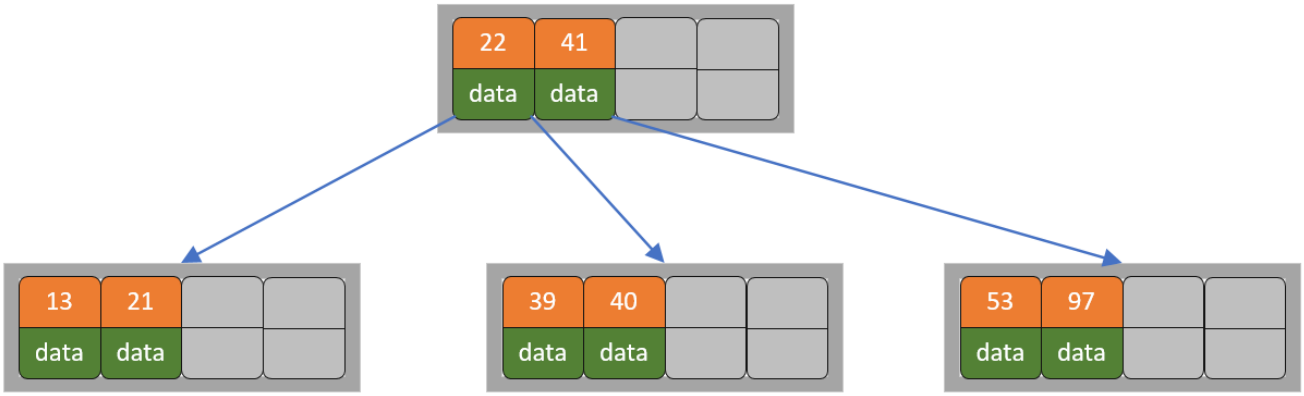
果阶数是偶数，分裂时就不存在排序恰好在中间的 key，那么我们选择中间位置的前一个 key 或中间位置的后一个 key 为中心进行分裂即可）



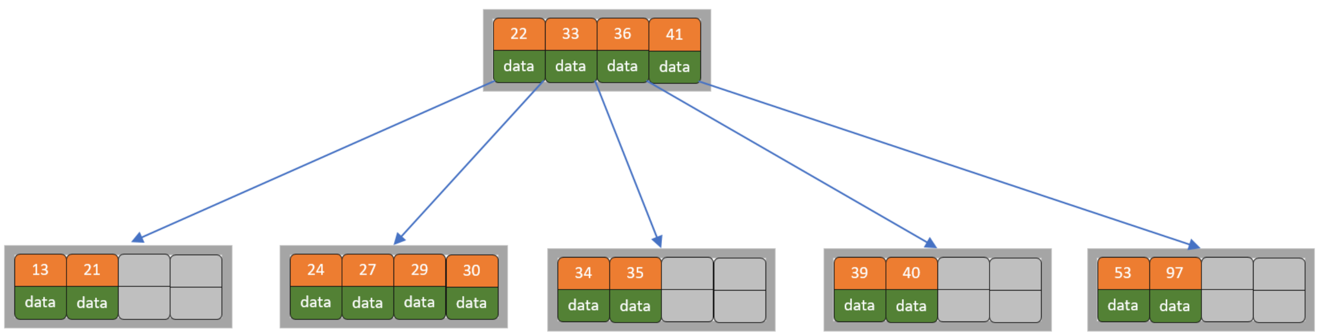
- 4）插入 13，21，40



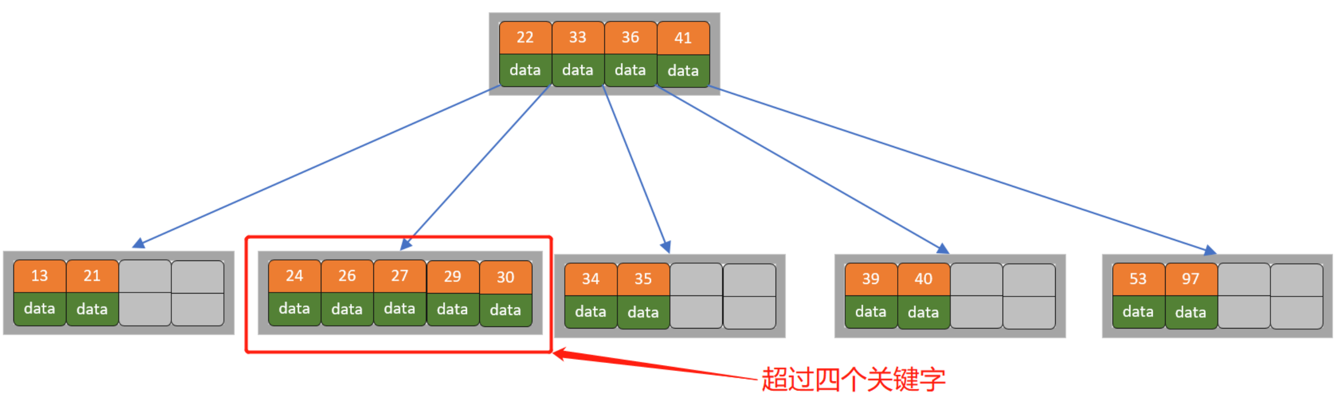
此时当前节点 5 个关键字，需要分裂，则以 22 为中心，22 节点插入到其父节点中，分裂后当前节点指向根节点，根节点的关键字为 2，满足 B 数条件，插入操作结束，结果如下所示



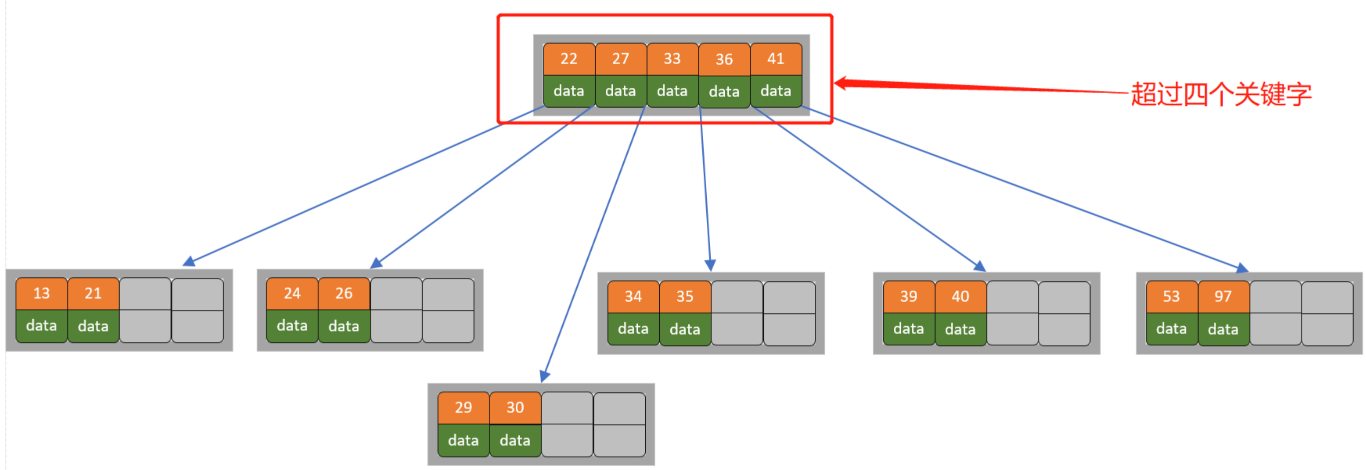
- 5）同理依次输入 30，27, 33 ，36，35，34 ，24，29，结果如下所示



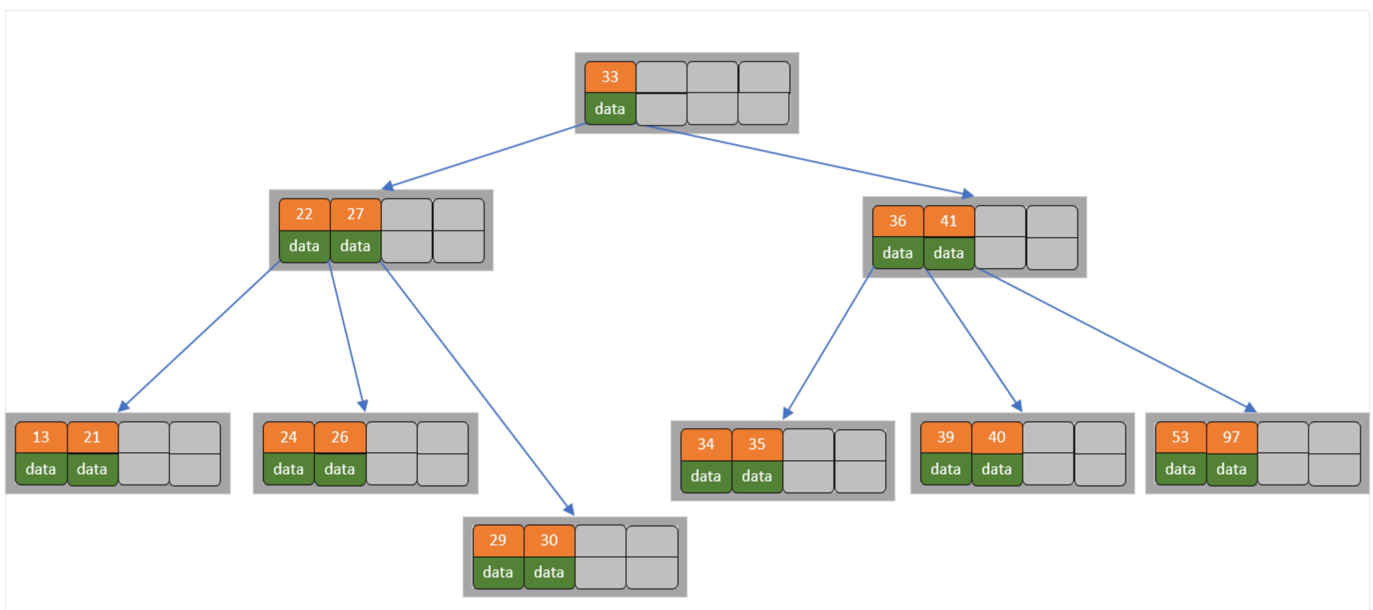
- 6）继续插入 26



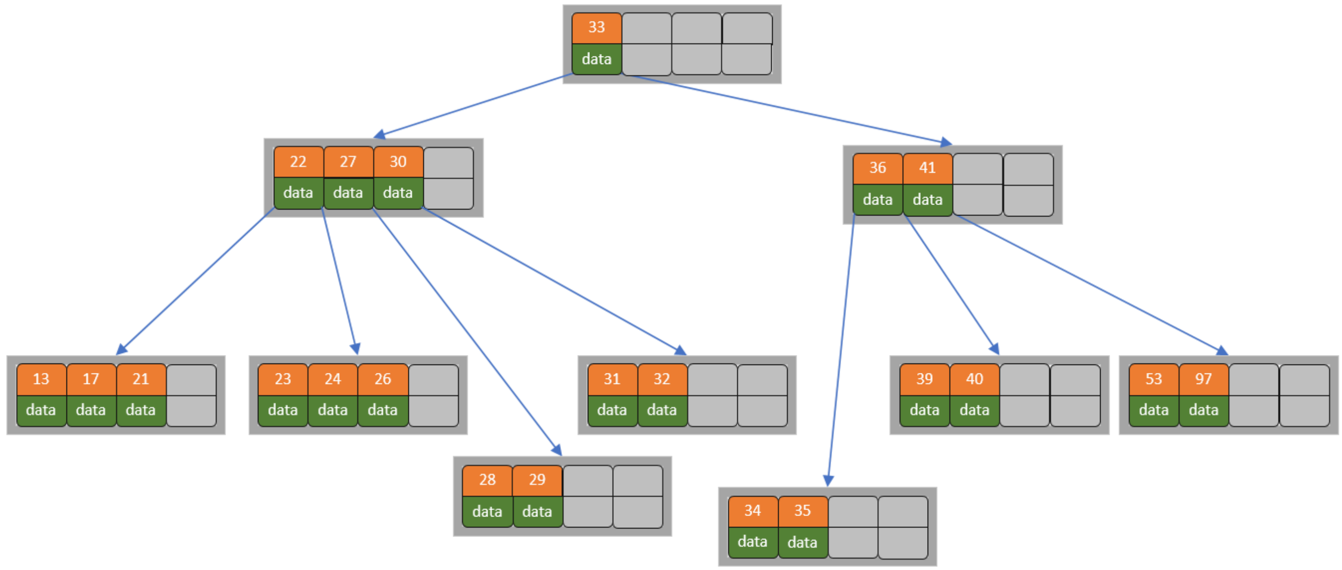
此时节点关键字等于 5，以 27 为中心分裂，并将 27 插入到父节点中，分裂后当前节点指向根节点，如下所示



此时 27 的进位导致当前节点也需要分裂，则以 33 为中心进行分裂，结果如下



- 7）同理最后再依次插入 17，28，29，31，32，结果如下图所示



总结

一般来说，对于确定的 m 和确定类型的记录，节点大小是固定的，无论它实际存储了多少个记录。但是分配固定节点大小的方法会**存在浪费的情况**，比如 key 为 28 和 29 所在的节点，还有 2 个 key 的位置没有使用，但是已经不可能继续在插入任何值了，因为这个节点的前序 key 是 27，后继 key 是 30，所有整数值都用完了。所以如果记录先按 key 的大小排好序，再插入到 B 树中，节点的使用率就会很低，**最差情况下使用率仅为 50%**。

B树的删除操作

删除操作是指根据 key 删除记录，如果 B 树中的记录中不存对应 key 的记录，则删除失败。

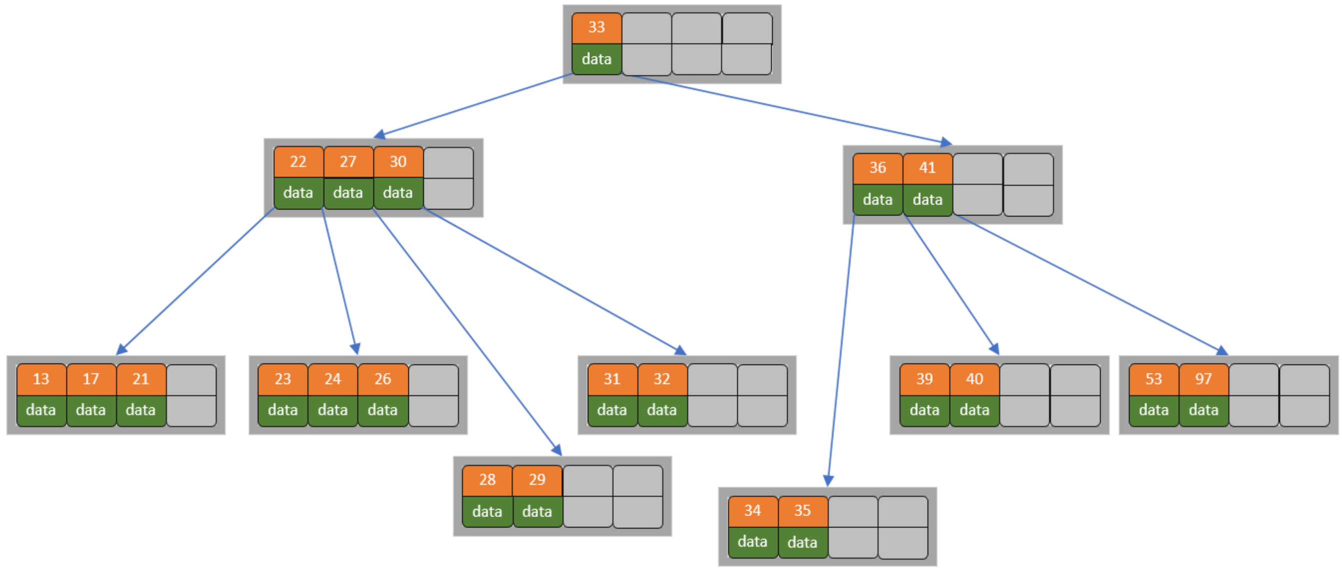
基本步骤

- 如果当前需要删除的 key 位于非叶子节点上，则用后继 key（这里的后继 key 均指后继记录的意思）覆盖要删除的 key，然后在后继 key 所在的子支中删除该后继 key。此时后继 key 一定位于叶子节点上，这个过程和二叉搜索树删除节点的方式类似。删除这个记录后执行第 2 步
- 该节点 key 个数大于等于Math.ceil(m/2)–1，结束删除操作，否则执行第 3 步。
- 如果兄弟节点 key 个数大于Math.ceil(m/2)–1，则父节点中的 key 下移到该节点，兄弟节点中的一个 key 上移，删除操作结束。
- 否则，将**父节点中的 key 下移与当前节点及它的兄弟节点中的 key 合并**，形成一个新的节点。原父节点中的 key 的两个孩子指针就变成了一个孩子指针，指向这个新节点。然后当前节点的指针指向父节点，重复上第 2 步。（有些节点它可能即有左兄弟，又有右兄弟，那么我们任意选择一个兄弟节点进行操作即可）

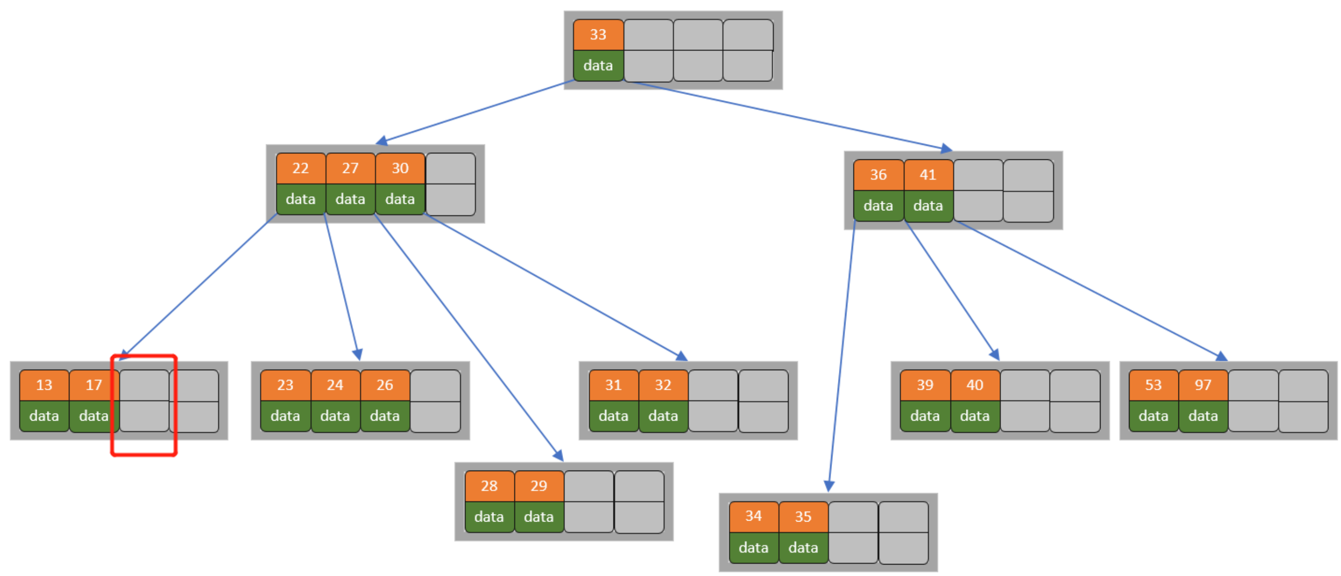
图文说明

以 5 阶 B 树为例（5 阶 B 树节点最多有 4 个关键字，最少有 2 个关键字，其中根节点最少可以只有一个关键字）。初始时刻以上述插入操作的最终状态为例。

- 1）初始状态

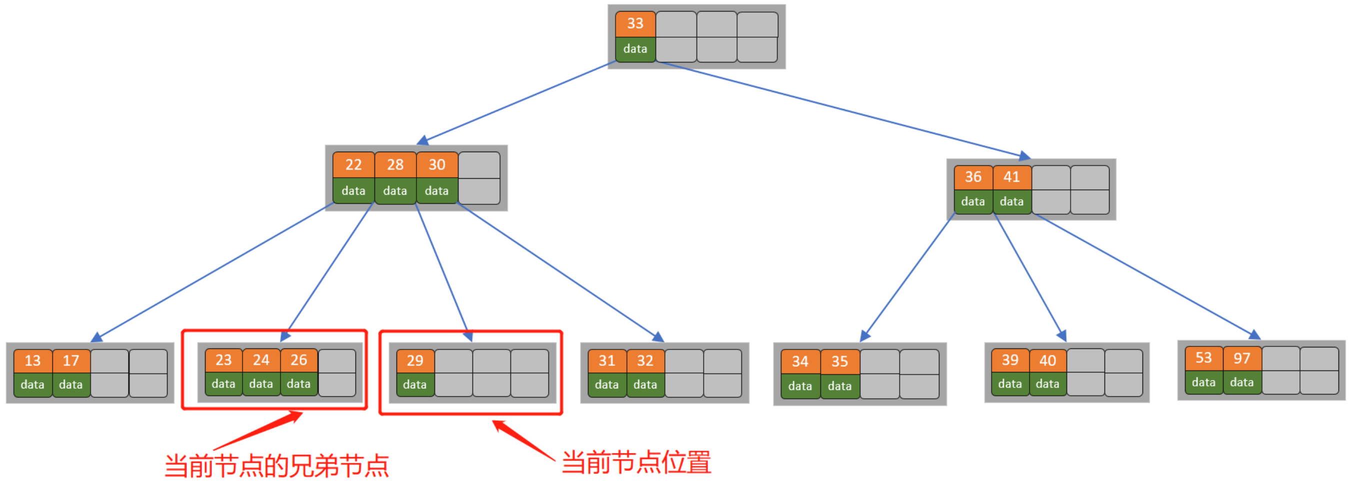


- 2）删除节点 21

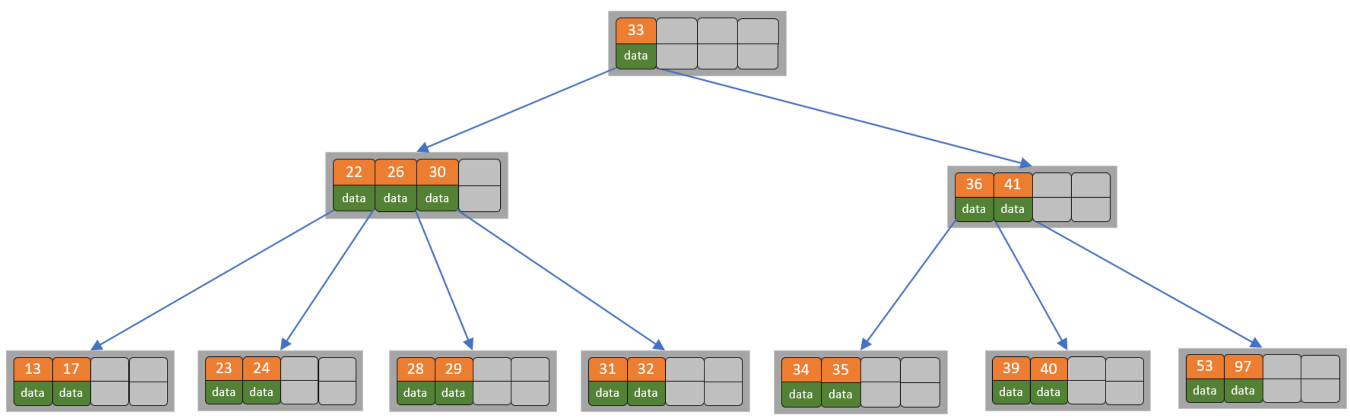


删除后节点中的关键字个数仍然大于等 2，所以删除结束。

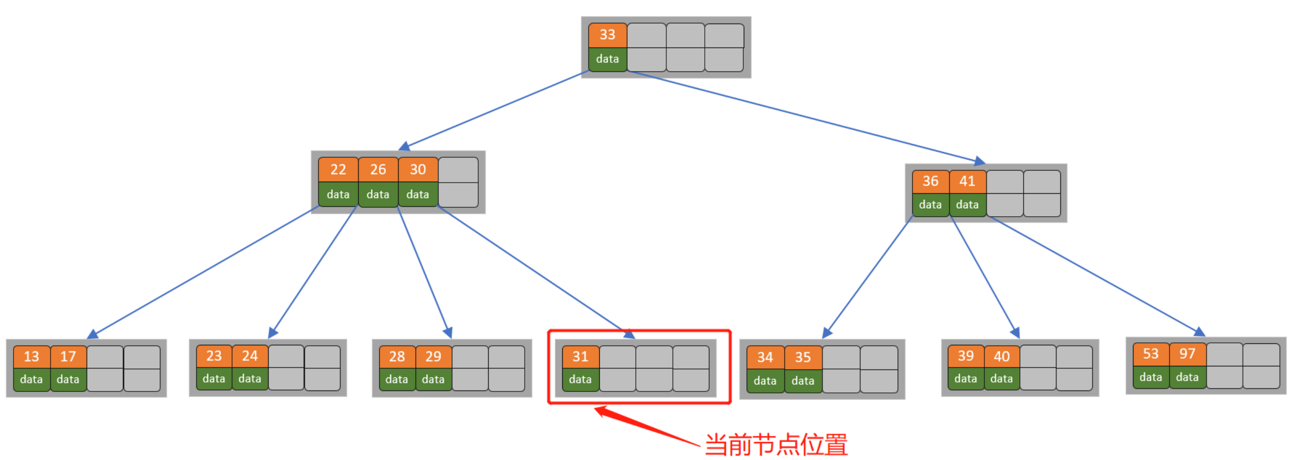
- 3）继续删除 27，此时 27 由于是非叶子节点，则由它的后继节点 28 替换 27，再删除 28，结果如下所示



此时发现叶子节点的个数小于 2，而它的兄弟节点中有 3 个记录（当前节点还有一个右兄弟，选择右兄弟就会出现合并节点的情况，不论选哪一个都行，只是最后 B 树的形态会不一样而已），我们可以从兄弟节点中借取一个 key。所以父节点中的 28 下移，兄弟节点中的 26 上移, 删除结束。结果如下图所示

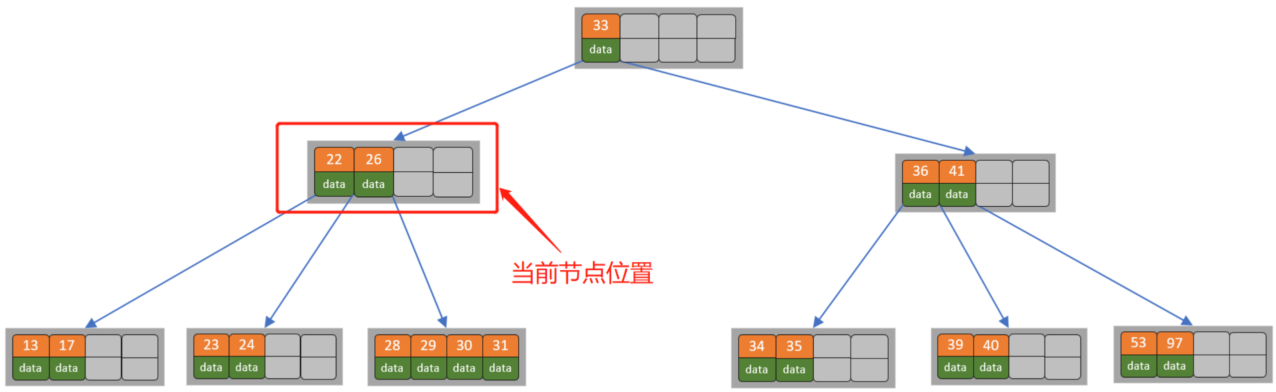


- 4）删除 32，结果如下图所示



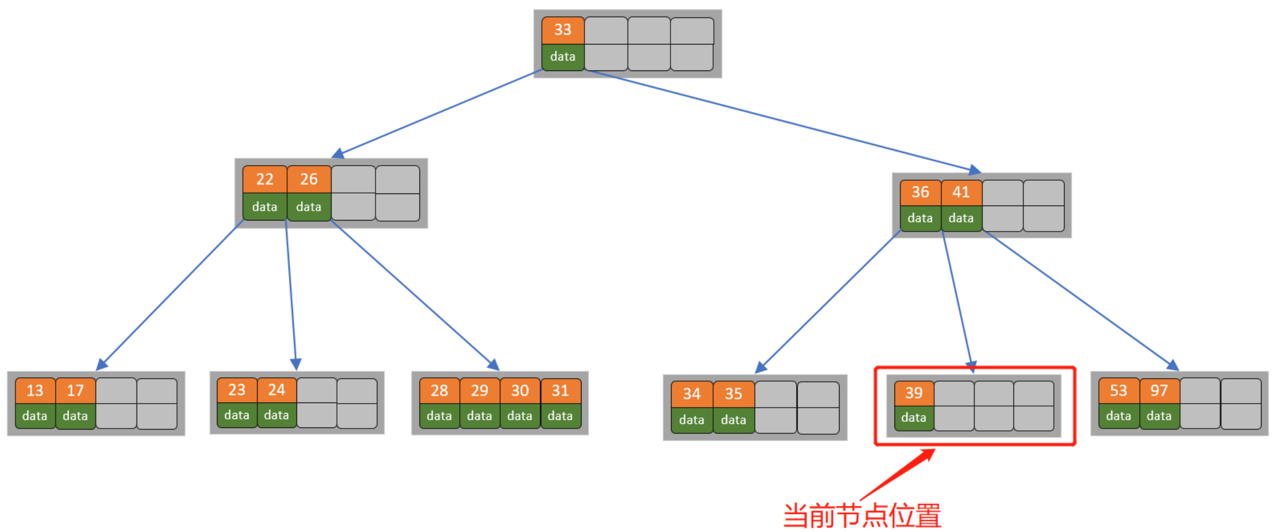
- 当前节点中只有一个 key，而兄弟节点中也仅有 2 个 key。所以只能让**父节点中的 30 下移和这个两个孩**

子节点中的 key 合并，成为一个新的节点，当前节点的指针指向父节点。结果如下图所示

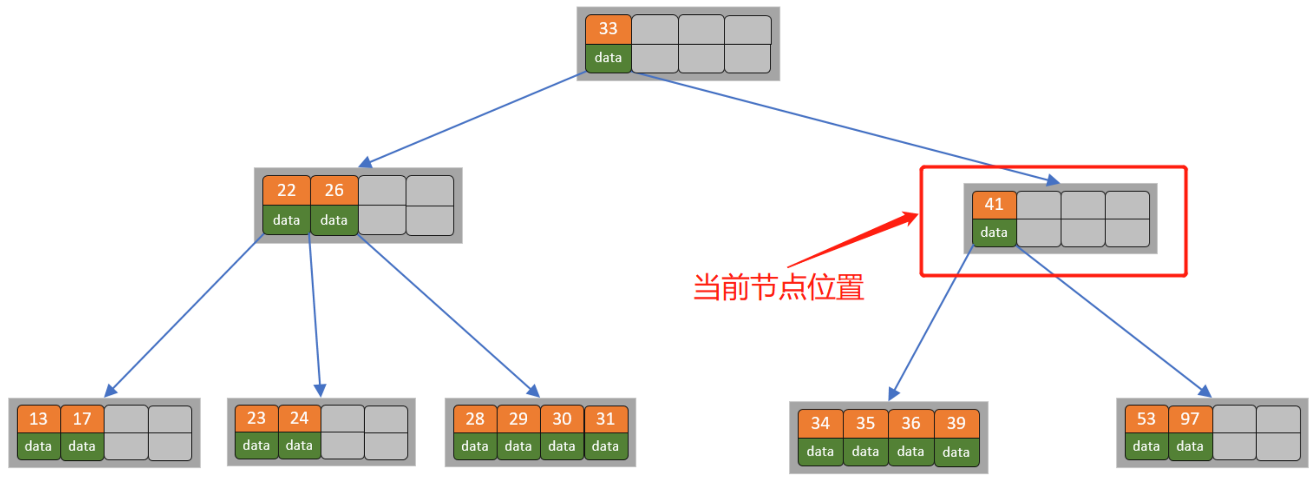


当前节点 key 的个数满足条件，故删除结束

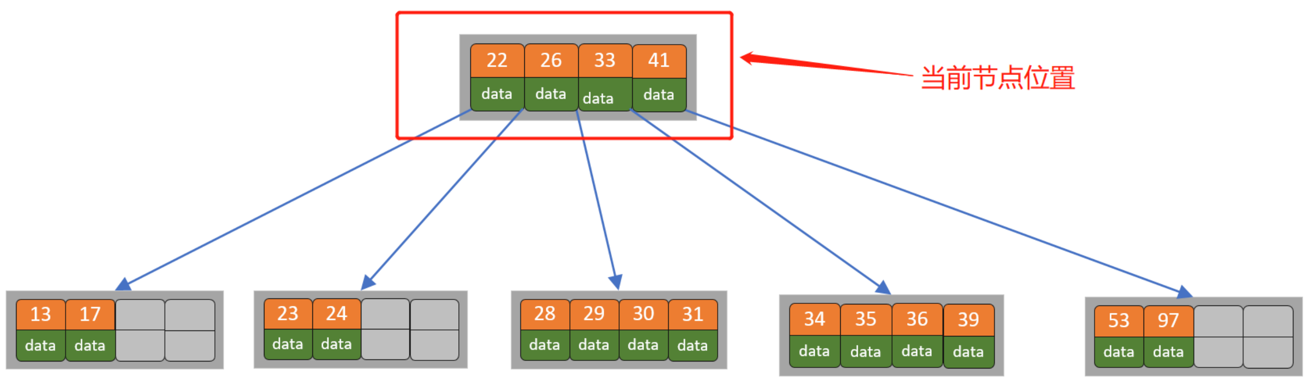
- 5）接着删除 key 为 40 的记录，删除后结果如下图所示



同理，当前节点的记录数小于 2，兄弟节点中没有多余 key，所以父节点中的 key 下移，和兄弟（这里我们选择左兄弟，选择右兄弟也可以）节点合并，合并后的指向当前节点的指针就指向了父节点。如下图所示



同理，对于当前节点而言只能继续合并了，最后结果如下所示



合并后节点当前节点满足条件，删除结束。

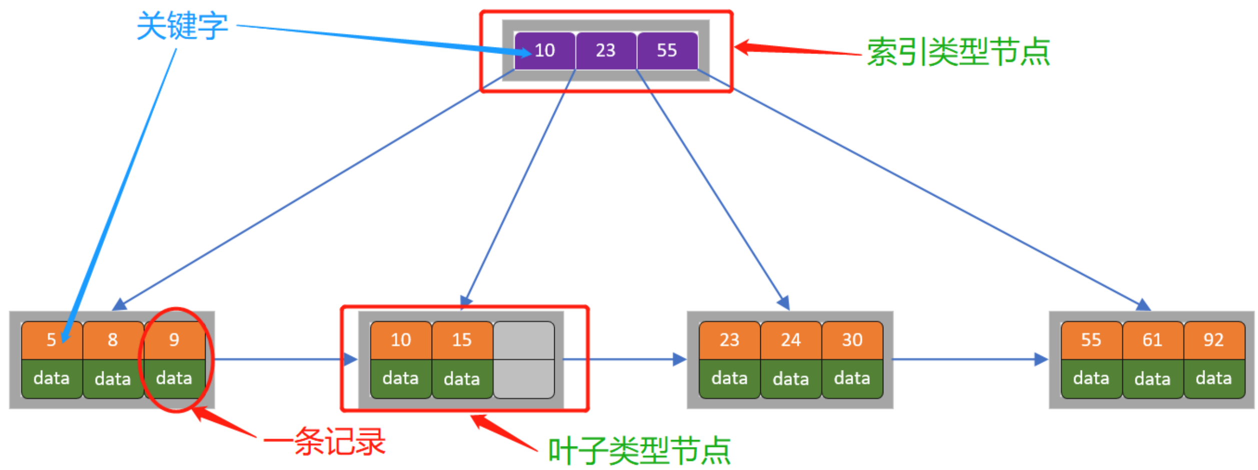
B+树

B + 树的定义

B + 树是 B 树的一种变形形式。**网上各种资料上 B + 树的定义各有不同，一种定义方式是关键字个数和孩子节点个数相同。这里我们采取维基百科上所定义的方式，即关键字个数比孩子节点个数小 1，这种方式是和 B 树基本等价的**。除了 B 树的性质，B + 树还包括以下要求：

- 1）B + 树包含 2 种类型的节点：**内部节点（也称索引节点）**和**叶子节点**。根节点本身即可以是内部节点，也可以是叶子节点。根节点的关键字个数最少可以只有 1 个。
- 2）B + 树与 B 树最大的不同是**内部节点不保存数据**，**只用于索引**，所有数据（或者说记录）都**保存在叶子节点中**。
- 3） m 阶 B + 树表示了**内部节点最多有 m–1 个关键字**（或者说内部节点最多有 m 个子树），阶数 m 同时限制了**叶子节点最多存储 m–1 个记录**。

- 4）内部节点中的 key 都按照**从小到大**的顺序排列，对于内部节点中的一个 key，左树中的所有 key 都**小于**它，右子树中的 key 都**大于等于**它。叶子节点中的记录也按照 key 的大小排列。
- 5）每个**叶子节点都存有相邻叶子节点的指针**，叶子节点本身依关键字的大小自小而大顺序链接。



上图是一棵阶数为 4 的 B + 树

B + 树的搜索操作

操作流程同 B 树的搜索流程，只不过如果要找的关键字匹配上了索引节点的关键字，需要继续往下找，因为索引节点不存储数据，所有的数据都存储在叶子节点上。

B + 树的插入操作

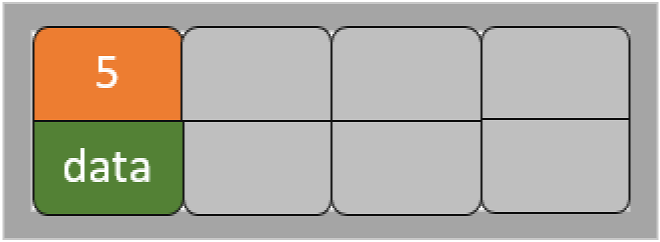
基本步骤

- 1）若为空树，创建一个叶子节点，然后将记录插入其中，此时这个叶子节点也是根节点，插入操作结束。
- 2）针对**叶子类型**节点：根据 key 值找到叶子节点，向这个叶子节点插入记录。插入后，若当前节点 key 的个数小于等于 m–1，则插入结束。否则将这个叶子节点分裂成左右两个叶子节点，左叶子节点包含前 m/2 个记录，右节点包含剩下的记录，将第 m/2+1 个记录的 key 进位到父节点中（父节点一定是索引类型节点），进位到父节点的 key 左孩子指针向左节点, 右孩子指针向右节点。将当前节点的指针指向父节点，然后执行第 3 步。
- 3）针对**索引类型**节点：若当前节点 key 的个数小于等于 m–1，则插入结束。否则，将这个索引类型节点分裂成两个索引节点，左索引节点包含前 (m–1)/2 个 key，右节点包含 m–(m–1)/2 个 key，将第 m/2 个 key 进位到父节点中，进位到父节点的 key 左孩子指向左节点, 进位到父节点的 key 右孩子指向右节点。将当前节点的指针指向父节点，然后重复第 3 步。

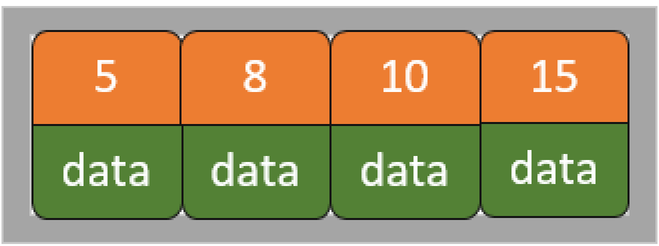
图文说明

以 5 阶 B + 树为例（5 阶 B + 树节点最多有 4 个关键字，最少有 2 个关键字，其中根节点最少可以只有一个关键字），从初始时刻依次插入数据。

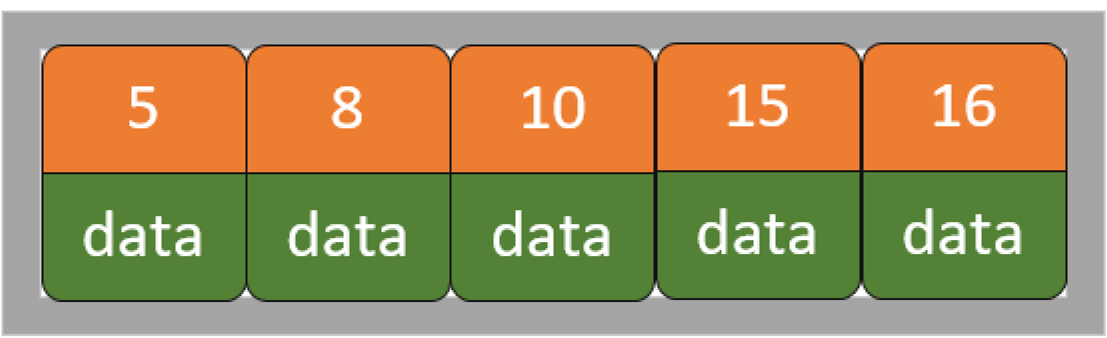
- 1）在空树插入 5



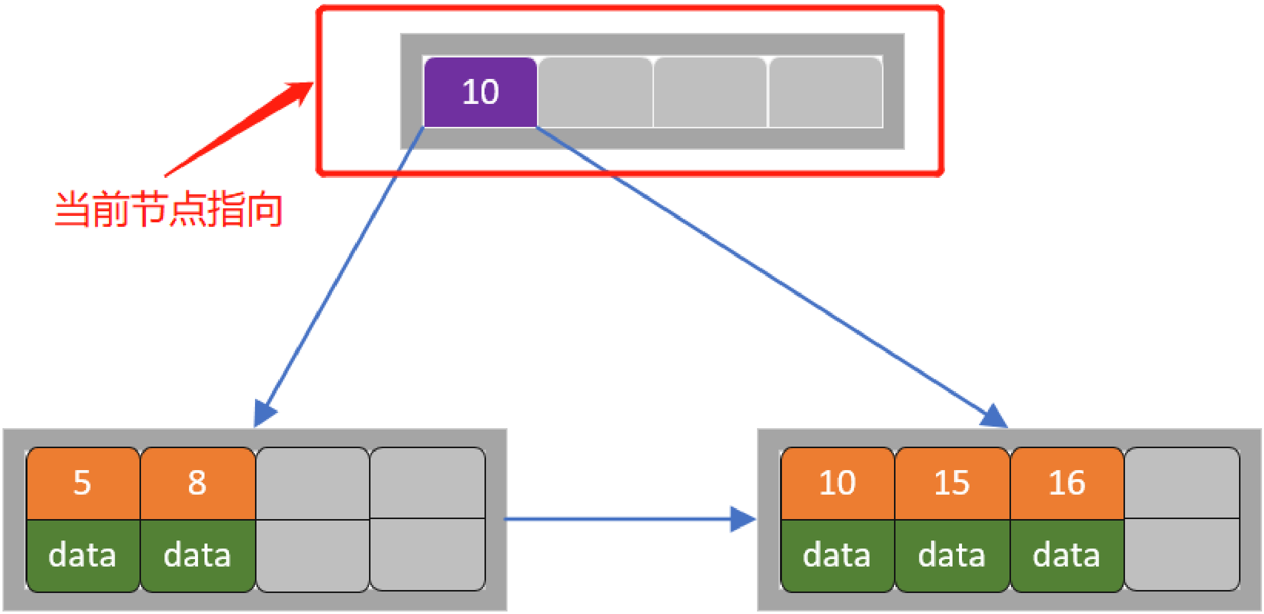
- 2）依次插入 8,10,15



- 3）插入 16

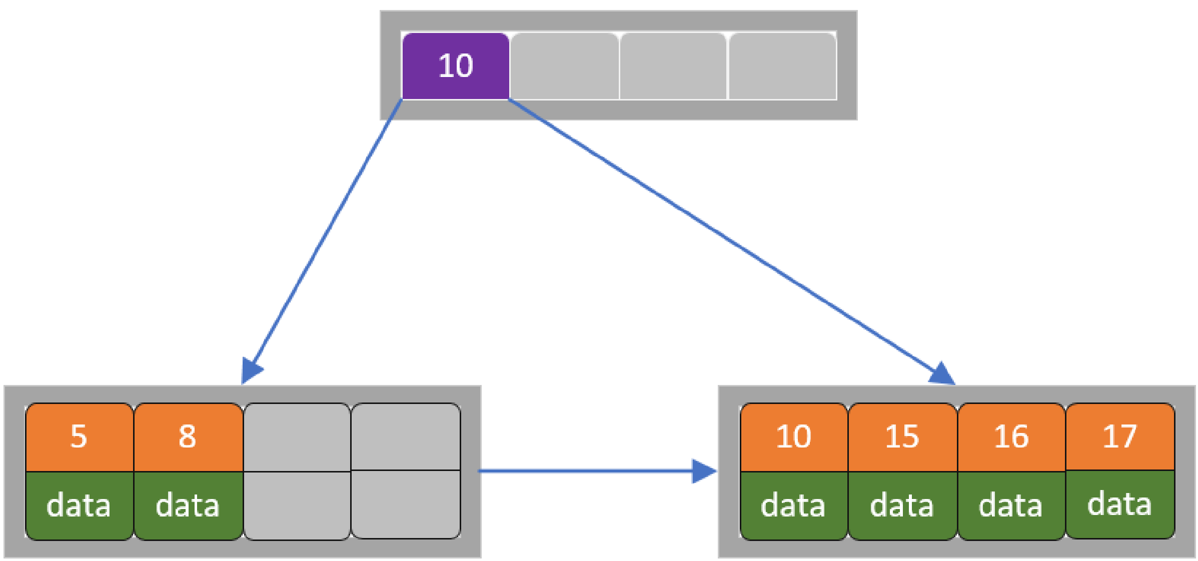


此时节点超过关键字的个数，所以需要进行分裂。由于该节点为叶子节点，所以可以分裂出来左节点 2 个记录，右边 3 个记录，中间 key 成为索引节点中的 key（也可以左节点 3 个记录，右节点 2 个记录），分裂后当前节点指向了父节点（根节点）。结果如下图所示

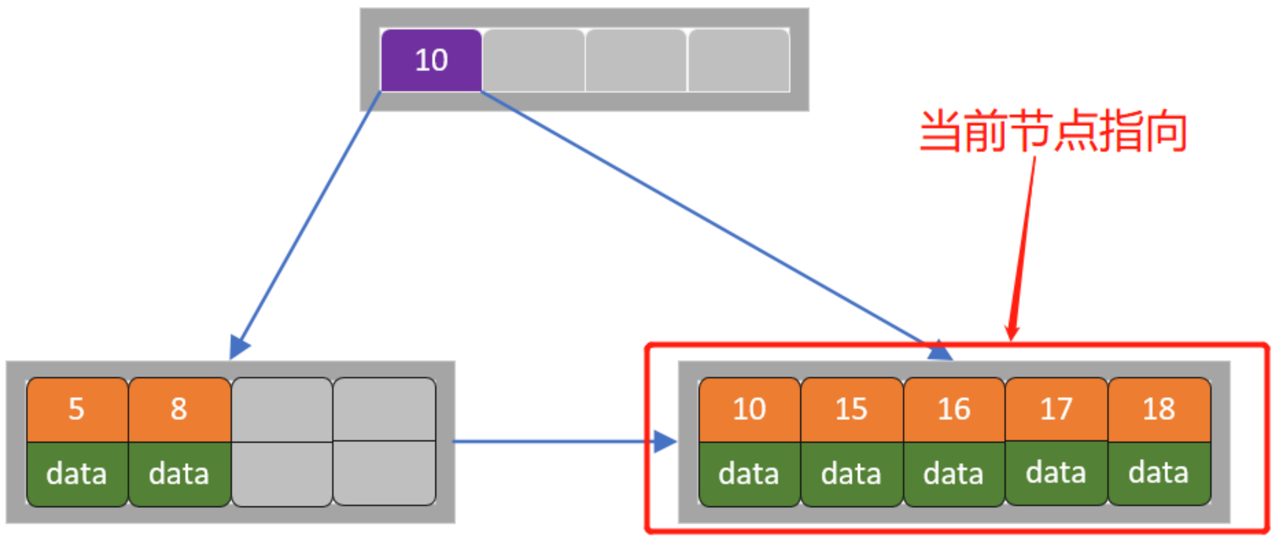


当前节点的关键字个数满足条件，插入结束

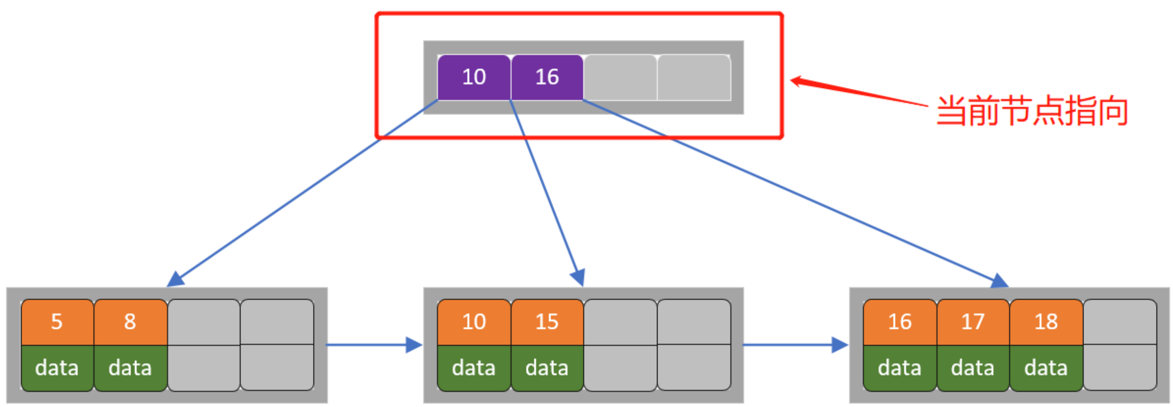
- 4）插入 17



- 5）插入 18

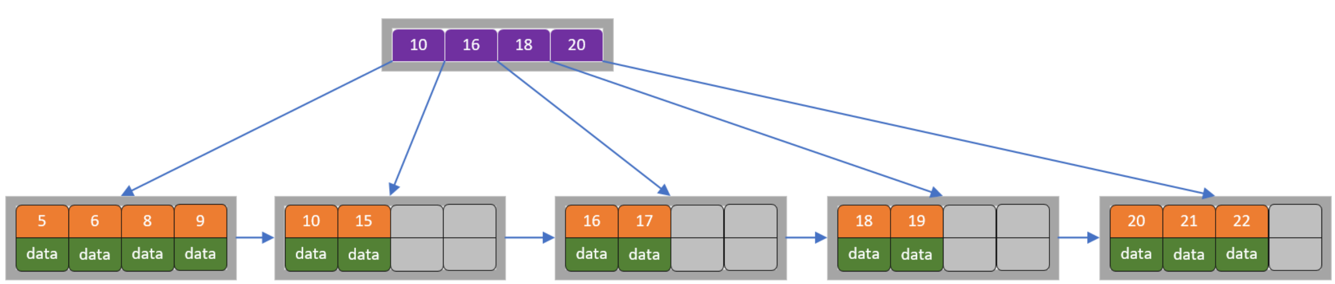


当前节点超过关键字的个数，进行分裂。由于是叶子节点，分裂成两个节点，左节点 2 个记录，右节点 3 个记录，关键字 16 进位到父节点（索引类型）中，将当前节点的指针指向父节点，如下图所示

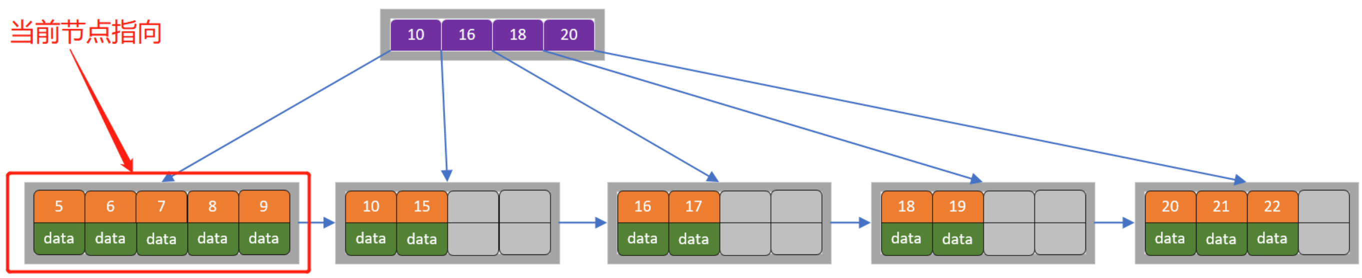


当前节点的关键字个数满足条件，插入结束

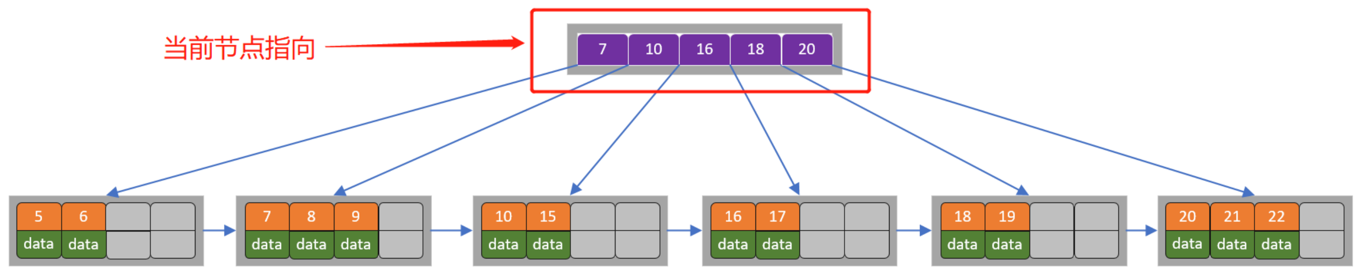
- 6）同理继续插入 6,9,19，细节不再描述



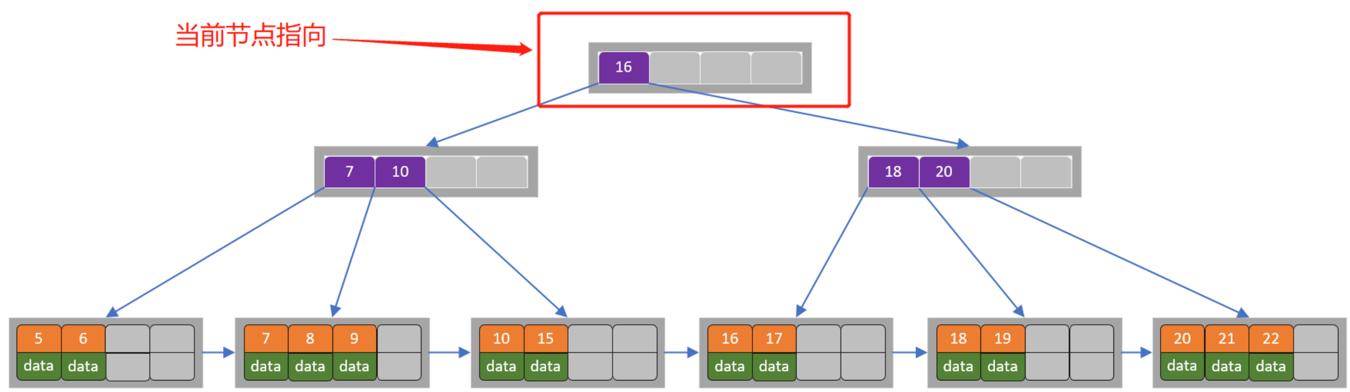
- 7）继续插入 7



当前节点超过关键字的个数，进行分裂。由于是叶子节点，分裂成两个节点，左节点 2 个记录，右节点 3 个记录，关键字 7 进位到父节点（索引类型）中，将当前节点的指针指向父节点，如下图所示



当前节点超过关键字的个数，进行分裂。由于是索引节点，左节点 2 个关键字，右节点 2 个关键字，关键字 16 进入到父节点中，将当前节点指向父节点，如下图所示



当前节点的关键字个数满足条件，插入结束

B + 树的删除操作

基本步骤

如果叶子节点中没有相应的 key，则删除失败。否则执行下面的步骤：

- 1）删除叶子节点中对应的 key。删除后若节点的 key 的个数大于等于 Math.ceil(m–1)/2 – 1，删除操作结束，否则执行第 2 步。
- 2）若兄弟节点 key 有富余（大于 Math.ceil(m–1)/2 – 1），向兄弟节点借一个记录，同时用借到的 key 替换父结（指当前节点和兄弟节点共同的父节点）点中的 key，删除结束。否则执行第 3 步。

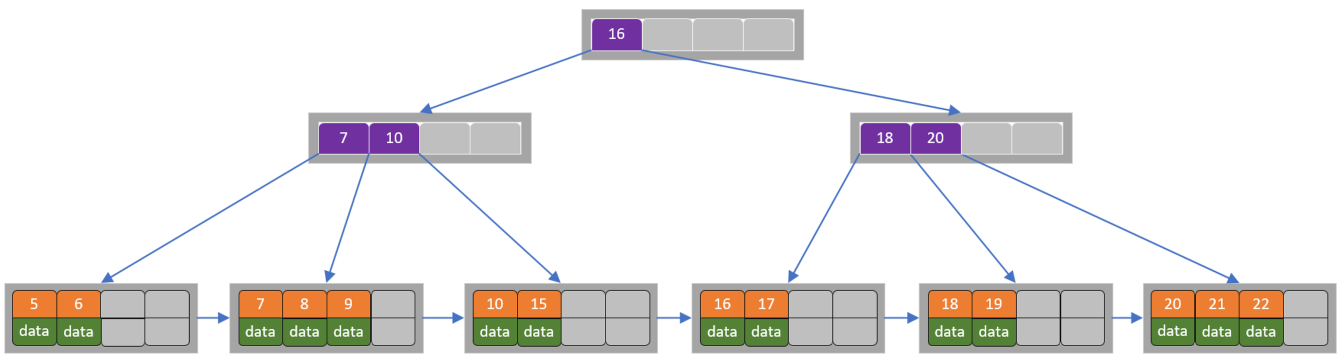
- 3）若兄弟节点中没有富余的 key, 则当前节点和兄弟节点合并成一个新的叶子节点，并删除父节点中的 key（父节点中的这个 key 两边的孩子指针就变成了一个指针，正好指向这个新的叶子节点），将当前节点指向父节点（必为索引节点），执行第 4 步（第 4 步以后的操作和 B 树就完全一样了，主要是为了更新索引节点）。
- 4）若索引节点的 key 的个数大于等于 Math.ceil(m–1)/2 – 1，则删除操作结束。否则执行第 5 步
- 5）若兄弟节点有富余，父节点 key 下移，兄弟节点 key 上移，删除结束。否则执行第 6 步
- 6）当前节点和兄弟节点及父节点下移 key 合并成一个新的节点。将当前节点指向父节点，重复第 4 步。

注意，通过 B + 树的删除操作后，索引节点中存在的 key，不一定在叶子节点中存在对应的记录。

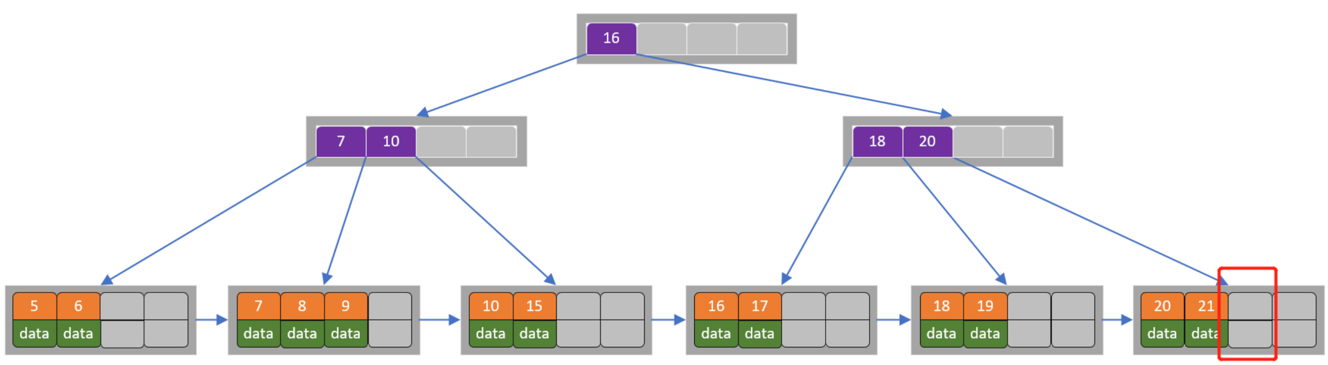
图文说明

以 5 阶 B 树为例（5 阶 B 树节点最多有 4 个关键字，最少有 2 个关键字，其中根节点最少可以只有一个关键字）。初始时刻以上述插入操作的最终状态为例。

- 1）初始状态

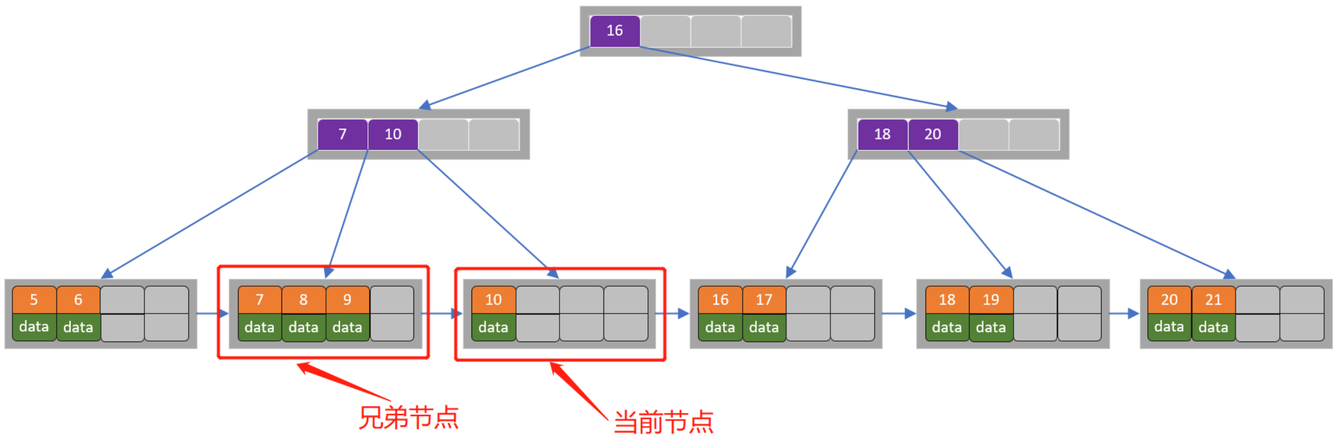


- 2）删除 22

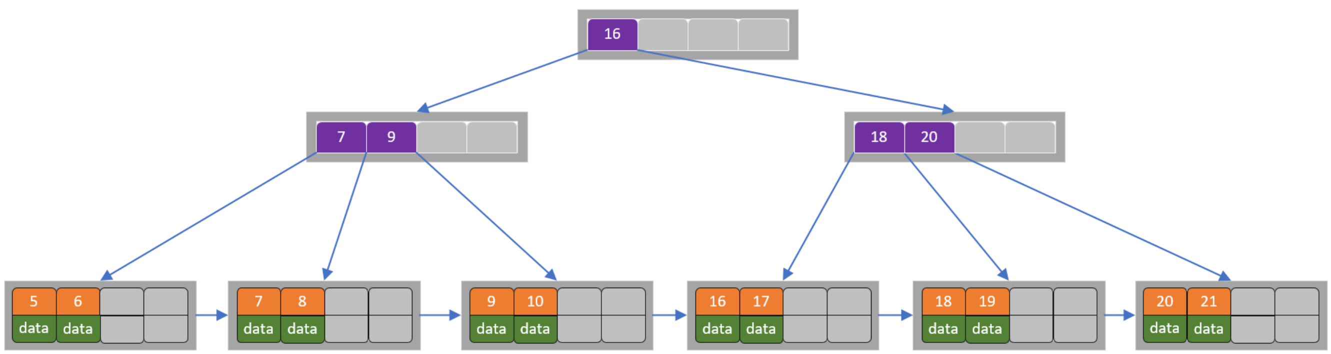


删除后叶子节点中 key 的个数大于等于 2，删除结束

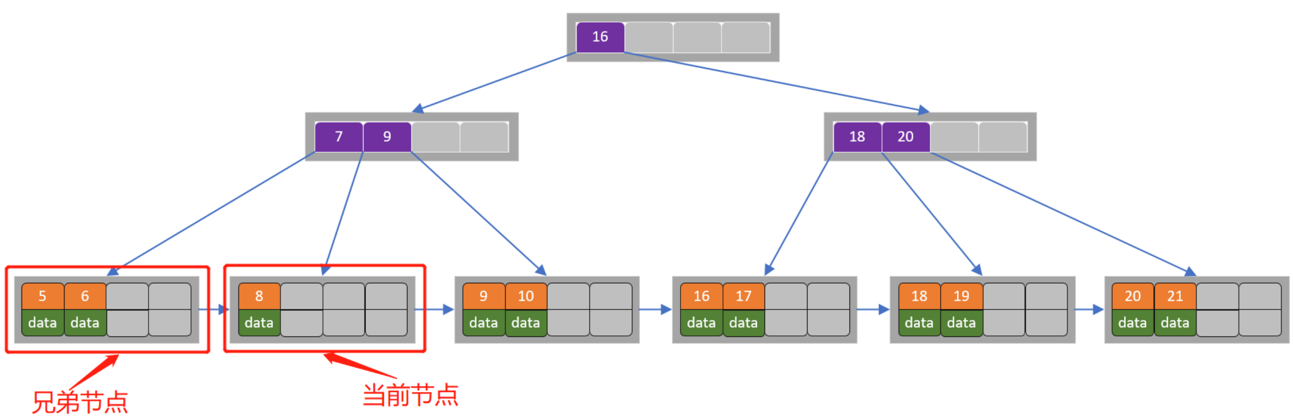
- 3）删除 15



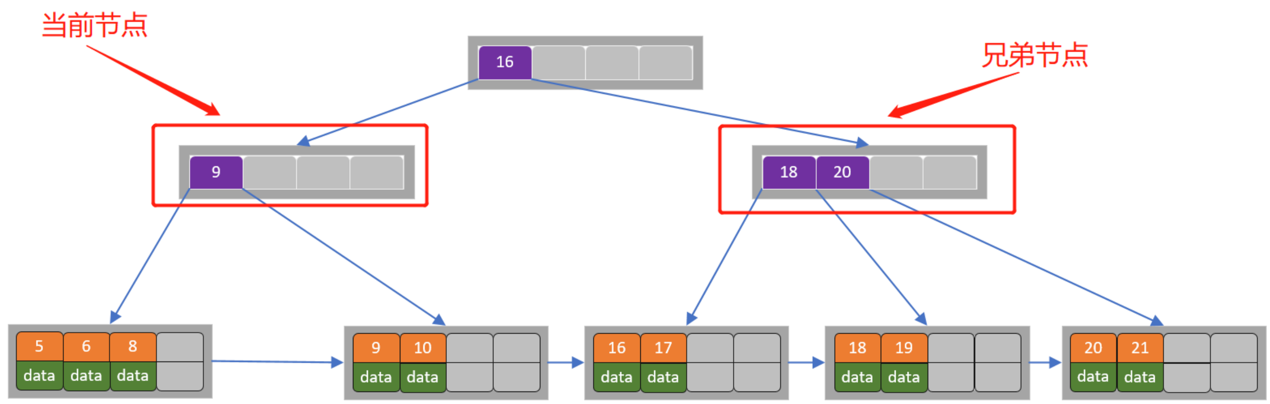
当前节点只有一个 key，不满足条件，而兄弟节点有三个 key，可以从兄弟节点借一个关键字为 9 的记录，同时更新将父节点中的关键字由 10 也变为 9，删除结束。



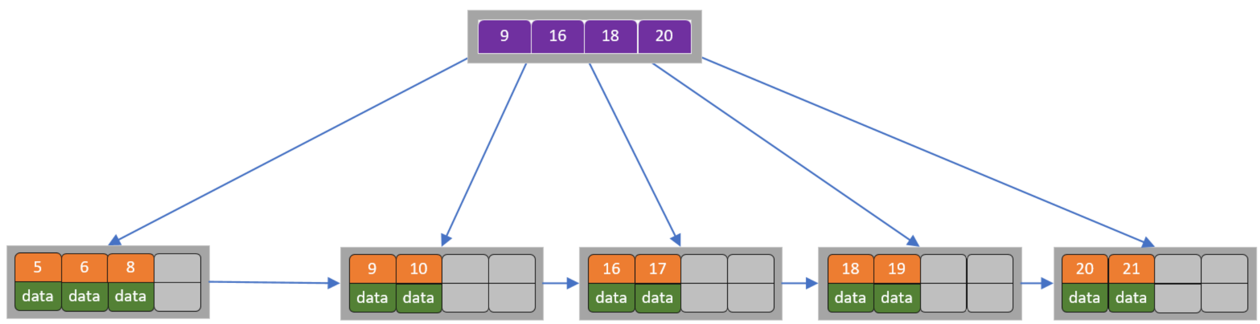
- 4）删除 7

- 

  当前节点关键字个数小于 2，（左）兄弟节点中的也没有富余的关键字（当前节点还有个右兄弟，不过选择任意一个进行分析就可以了，这里我们选择了左边的），所以当前节点和兄弟节点合并，并删除父节点中的 key，当前节点指向父节点。



此时当前节点的关键字个数小于 2，兄弟节点的关键字也没有富余，所以父节点中的关键字下移，和两个孩子节点合并，结果如下图所示。



删除结束。

原博文地址：http://xianzilei.cn/blog/31

其余总结

B树（balance tree）和B+树应用在数据库索引，可以认为是m叉的多路平衡查找树，但是从理论上讲，二叉搜索树查找速度和比较次数都是最小的，为什么不用二叉搜索树呢？ **因为我们要考虑磁盘IO的影响，它相对于内存来说是很慢的。**
数据库索引是存储在磁盘上的，当数据量大时，就不能把整个索引全部加载到内存了，只能逐一加载每一个磁盘页（对应索引树的节点）。所以我们要减少IO次数，**对于树来说，IO次数就是树的高度，而“矮胖”就是b树的特征之一**，它的每个节点最多包含m个孩子，m称为B树的阶，m的大小取决于磁盘页的大小。

B+树主要应用场景是MySQL的索引（范围查询快，稳定，IO更少），B−树的应用场景是MangoDB的索引（热点数据查询快）

- B−树中间节点也存在卫星数据，B+树只有叶子存在卫星数据，如果是非聚簇索引，叶子节点存在的是指向卫星数据的指针
- 由于B+树中间节点不存放卫星数据，所以可以存放更多数据，所以B+树更扁平（IO更少）
- B−树查找不稳定，可能在中间节点查找到，也可能在叶子节点查找到。B+树稳定（更稳定）
- 范围查找，B+可以通过叶子节点的指针直接遍历，而B−只能通过中序遍历，将数据取出来，效率不高，且IO更大，（适合范围查找）

B树的优点是：如果经常访问的数据离根节点很近，而B树的非叶子节点本身存放数据，所以这种情况下的数据检索会比B+树快。