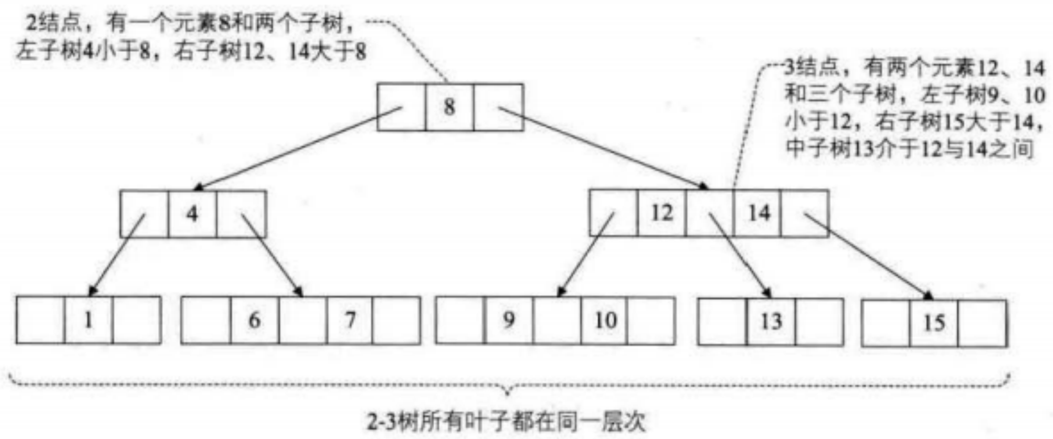
2-3树（3阶B树）

定义

2-3树是一棵自平衡的多路查找树，具有如下特点：

1. 每个节点一定是一个 2节点 或者 3节点
2. 2节点： 有 1 个数据项，2 个或者 0 个子节点，如果包含 2 个子节点，那么与二叉树类似，左子节点的数据项小于当节点数据项，右子节点数据项大于当前节点数据项
3. 3节点： 有 2 个数据项，3 个或者 0 个子节点，如果包含 3 个，那么左子节点的数据项较小，右子节点的数据项较大，中间子节点的数据项介于当前节点数据项之间



节点插入过程

分为三种情况，分别是在空树中插入、在二节点上插入和在三节点上插入

在空树中插入

对于空树，插入一个2节点即可

在二节点上插入

插入节点到一个2节点的叶子上。由于本身就只有一个元素，所以只需要将其升级为3节点即可

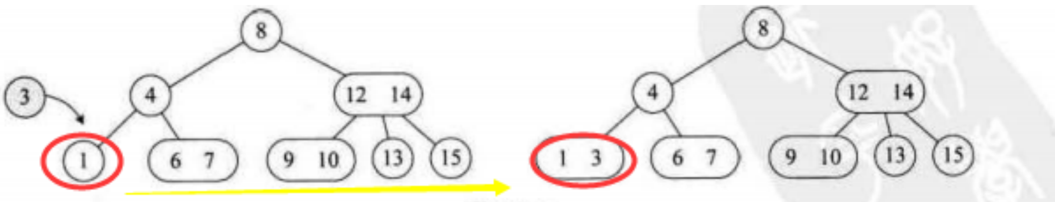
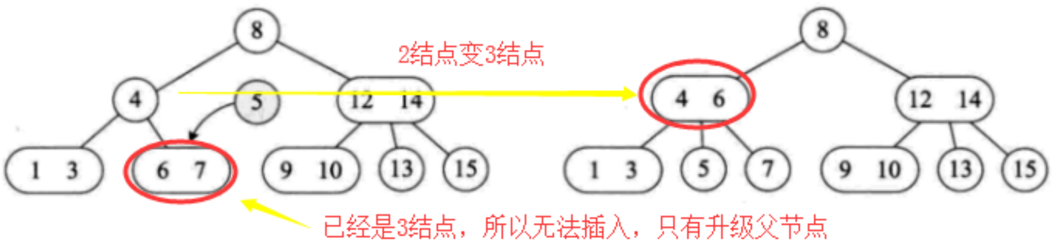


图 8-8-3

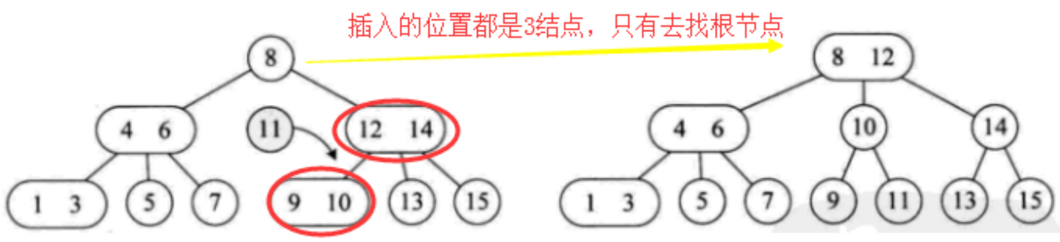
在三节点上插入

插入节点到一个3节点的叶子上。因为3节点本身最大容量，因此需要拆分，且将树中两元素或者插入元素的三者中选择其一向上移动一层，一共有三种情况

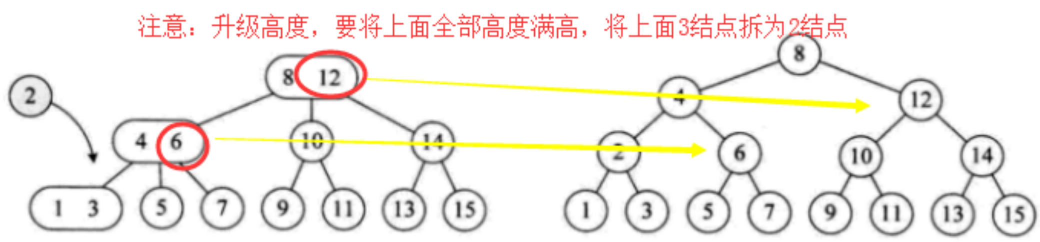
- 升级父节点



- 升级根节点 （一条路上的父节点都变为3结点，只有去找根）



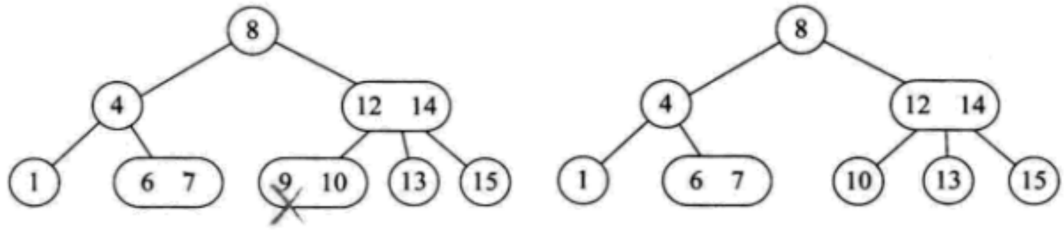
- 增加树高度（当我们根节点也变为3结点后，我们就要去升级树的高度）



节点删除过程

在为叶子节点的三节点上删除

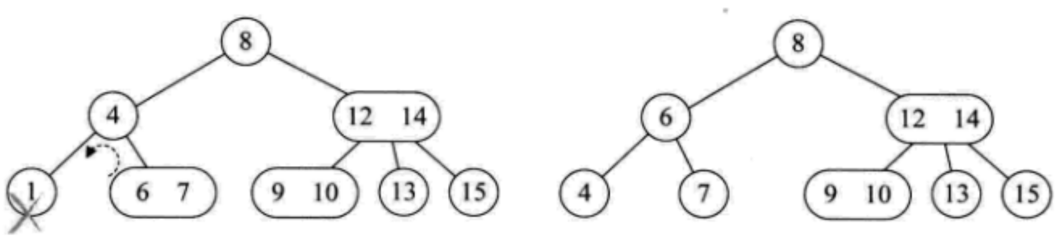
所删元素位于一个3节点的叶子节点上，直接删除，不会影响树结构



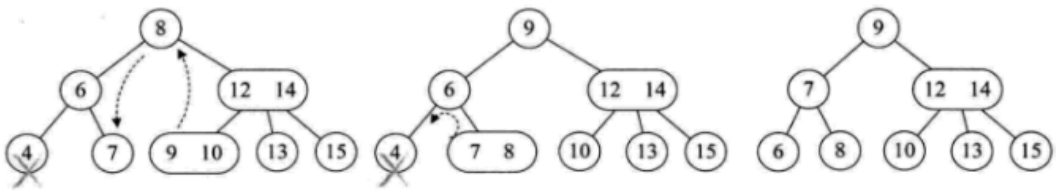
在为叶子节点的二节点上删除

所删元素位于一个2节点上，直接删除，破坏树结构，分为四种情况：

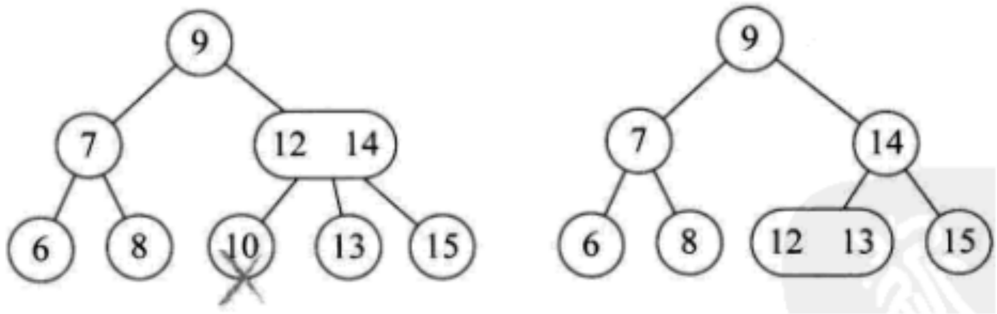
- 此节点双亲也是2节点，此节点的兄弟节点是一个 3节点



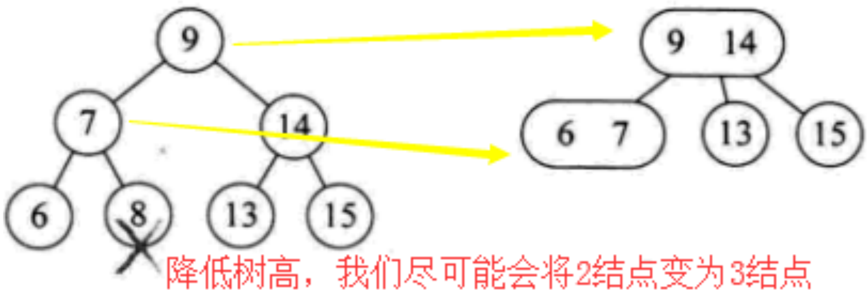
- 此节点双亲也是2节点，此节点的兄弟节点也是一个 2节点



- 此节点双亲也是3节点



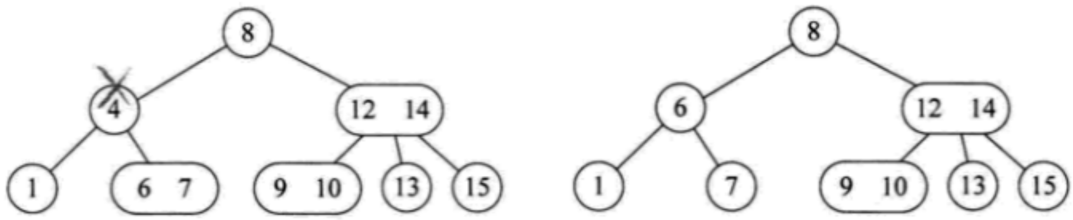
- 当前树是一个满二叉树，降低树高



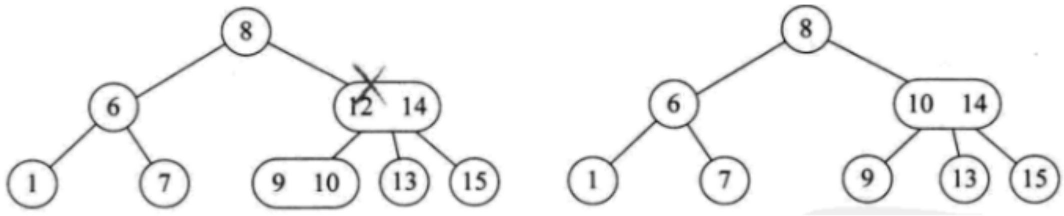
在为非叶子节点的节点上删除

此时按树中序遍历得到此元素的前驱或后续元素，补位，分为两种情况：

- 分支节点是2节点



- 分支节点是3节点



2-3-4树（4阶B树）

定义

2-3-4树是 2-3树的扩展，区别在于多了一个4节点。在2-3树中，每个节点一定是 2 节点或者 3 节点。而在2-3-4树中 每个节点一定是 2节点或者 3 节点或者 4 节点

4节点：包含小中大三个元素和四个孩子（或没有孩子）

节点插入过程

1. 分别插入7,1,2时的结果图，因为3个元素正好满足2-3-4树的单个4结点定义，因此不需要拆分

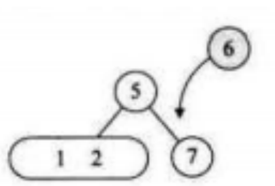


图2

2. 插入元素5，因为已经超过了4结点的定义，所以要进行拆分，因为要满足结点要么没有孩子，要么满子，所以我们不能选择7来作为根，最好选择5来作为拆分后的根

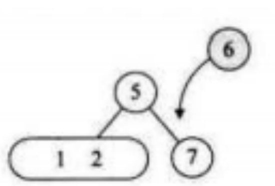


图2

3. 插入元素6，安装排序树方法找到7，发现可以扩展，直接将2结点扩展为3结点，存放6

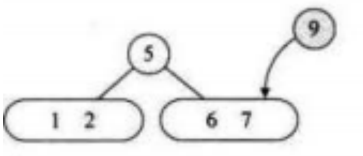
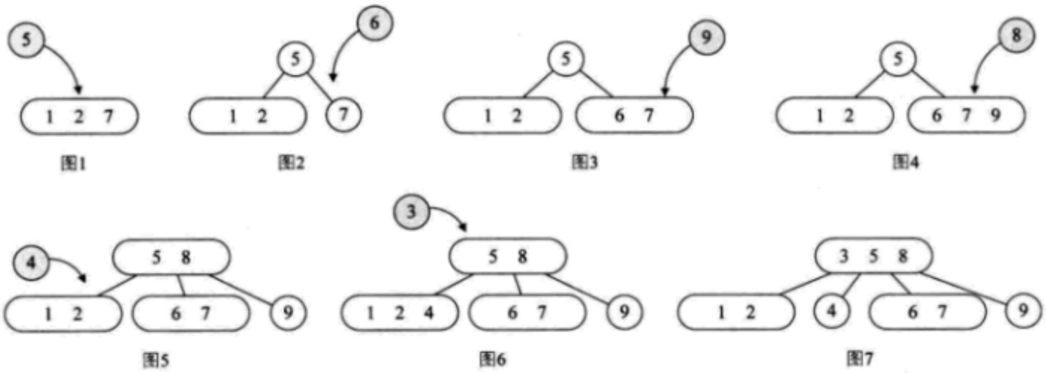


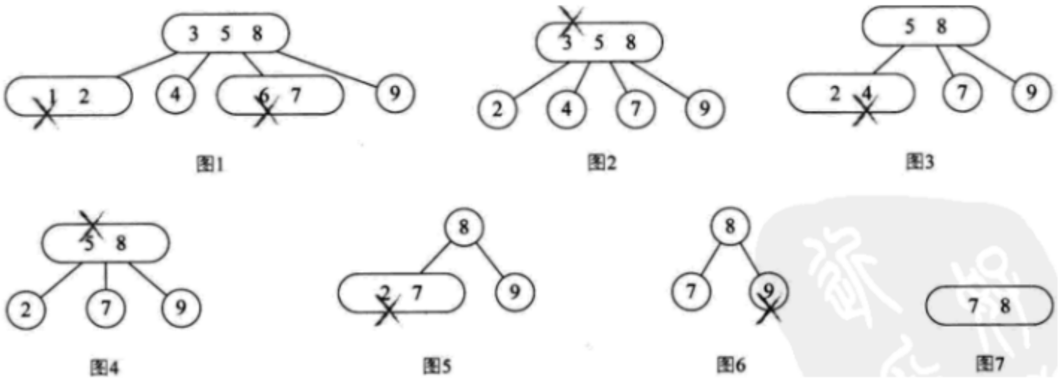
图3

4. 完整过程



节点删除过程

删除顺序：1、6、3、4、5、2、9



B树

性质

如果前面的2-3树与2-3-4树理解了，B树也就理解了，因为2-3树就是3阶的B树，2-3-4树就是4阶的B树。所以，对于B树的性质，根据2-3-4树都可以推导出来了，即：

一颗m阶的B树（B-tree） 定义如下：

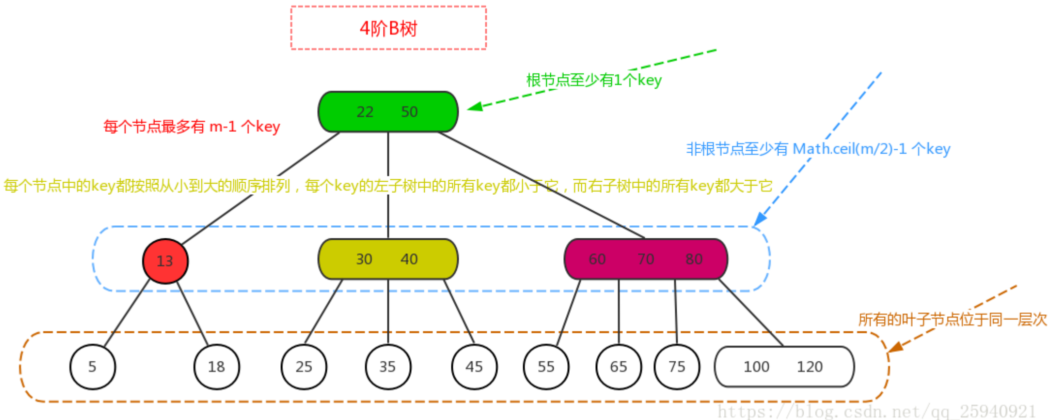
（1）每个节点最多有 m-1 个key；

（2）根节点至少有1个key；

（3）非根节点至少有 Math.ceil(m/2)-1 个key；

（4）每个节点中的key都按照从小到大的顺序排列，每个key的左子树中的所有key都小于它，而右子树中的所有key都大于它；

（5）所有叶子节点都位于同一层，即根节点到每个叶子节点的长度都相同



B树对比二叉排序树的优势

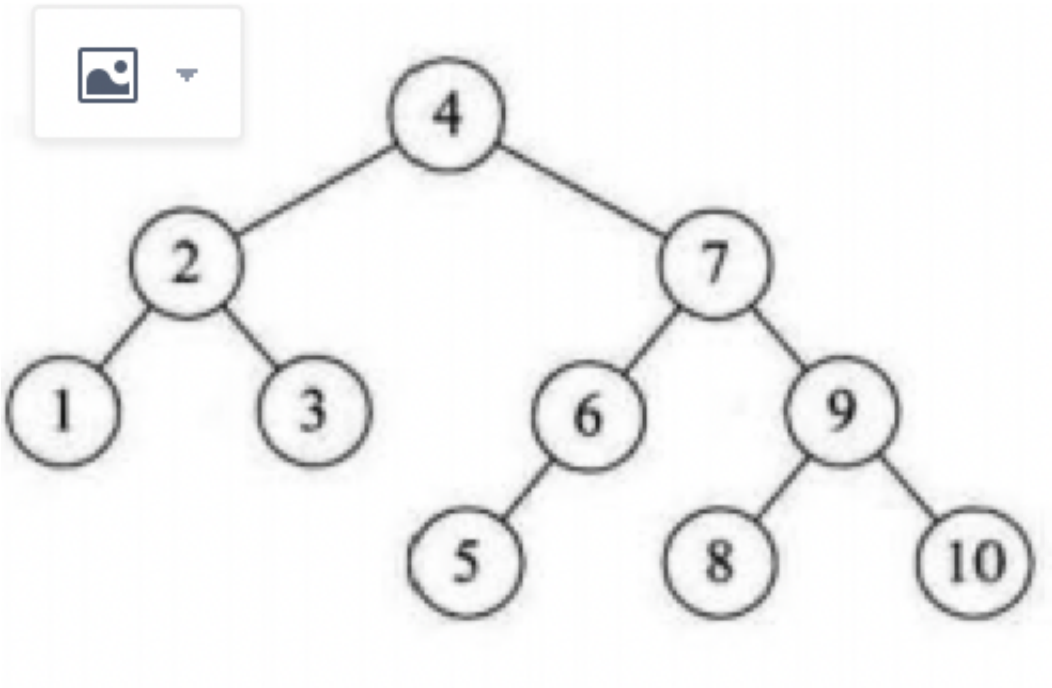
查询时间复杂度都是 log2(n)

B树的结构与二叉搜索树类似，查询效率也是 log2(n)，但是数据往往不是存储在内存中，当数据太多，内存不能全部存储。

B树的优势

将根节点存在内存中，其他节点存放在硬盘上 ，查找过程说明

- 二叉树



可以看到高度有4层，如果查找8，在内存中查找根节点 4 ， 4的子节点存放在硬盘上就要去硬盘找到7，然后继续硬盘查找9，最后硬盘查找得到8

- B树

而B树的高度比 二叉树低的多，阶数越大，高度越低。可以极大减少硬盘查找的次数

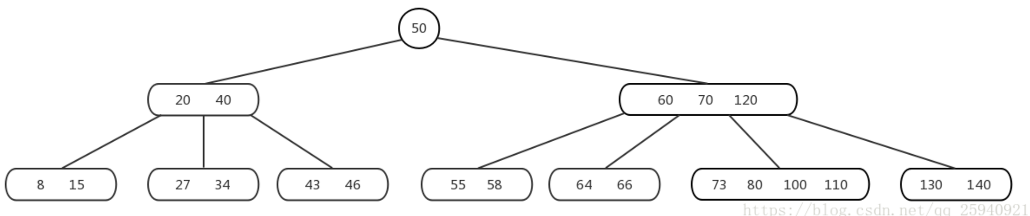
打个比方，以2-3树为例，树高为3的时候，一棵2-3树可以保存2+3x2+3x2x2=20个key，若当B树的阶数达到1001阶，即一个节点可以放1000个key，然后树高还是3，即 1000+1000x1001+1000x1001x1000 ，零头不算了，即至少可以放10个亿的key，此时我们只要让根节点读取到内存中，把子节点及子孙节点持久化到硬盘中，那么在这棵树上，寻找某一个key至多需要2次硬盘的读取即可

ps：key表示数据项

B+树

B树的小瑕疵

虽然B树这种数据结构，应用在内外存交互，可以极大的减少磁盘的IO次数，但还是有些小瑕疵，如下5阶的B树图，若我需要读取key为“66”与“73”的数据，则此时从根节点“50”开始，“66”大于“50”，找右孩子，即到“60 70 120”的节点，再锁定到“64 66”的节点，找到key为“66”的数据，然后读“73”的数据，再重新从根开始往下寻找key为“73”的数据，如果需要查询的数据量一多，性能就很糟糕。还有一点，就是B树的每个节点都包含key及其value数据，这样的话，我每次读取叶子节点的数据时，在经过路径上的非叶子节点也会被读出，但实际上这部分数据我是不需要的，这样又占用了没有必要的内存空间



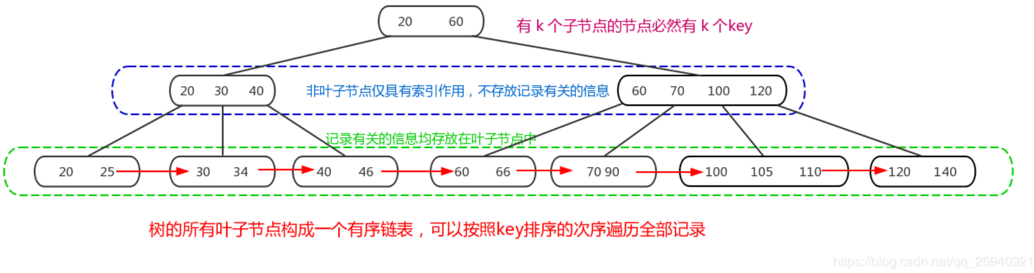
B+树的特性

B+树在B树的基础上做了优化，它与B树的差异在于：

（1）有 k 个子节点的节点必然有 k 个key；

（2）非叶子节点仅具有索引作用，跟记录有关的信息均存放在叶子节点中。

（3）树的所有叶子节点构成一个有序链表，可以按照key排序的次序遍历全部记录



B树和B+树的区别

B+树的非叶子结点只包含导航信息，不包含实际的值，所有的叶子结点和相连的节点使用链表相连，便于区间查找和遍历

B树的优点

由于B树的每一个节点都包含key和value，因此经常访问的元素可能离根节点更近，因此访问也更迅速

B+树的优点

1. 由于B+树在内部节点上不包含数据信息，因此在内存页中能够存放更多的key
2. B+树的叶子结点都是相链的，因此对整棵树的便利只需要一次线性遍历叶子结点即可，便于区间范围的 查找和搜索