为什么 MySQL 使用 B+Tree?

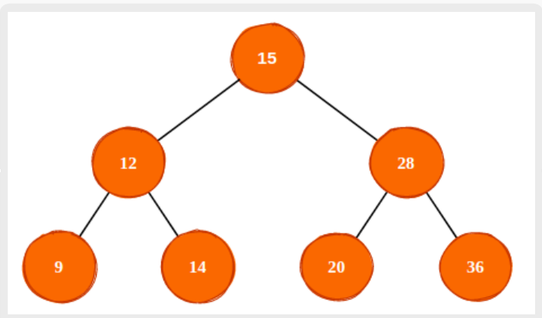
需求分析

- 查询
  - 对于一个数据库而言，最普遍的查询就是两种: 精确查询与范围查询。精确查询是指通过一个具体 key 来找到一条或者多条数据，范围查询则查询 key 在某一个范围内的所有数据
  - 如果仅仅考虑这两个需求的话，有序数组就能够满足我们的要求，二分搜索就好了，C++ 标准库还提供了 std::upper_bound 和 std::lower_bound 等好用的函数
- 插入
  - 这时候有序数组实现的 DB 在插入时效率就会很差，因为我们需要保证元素的有效性，所以平均时间复杂度为 O(n)。如果我们有 4G 数据的话，每一次的插入都可能需要几秒甚至几十秒来完成

有序数组还有另外一个很致命的缺陷，就是在数据量超过内存时，我们就要考虑怎么把一部分数据暂时存放在硬盘上，并且，我们还需要考虑数据的增删改查。对于连续排列的数据结构来说，做这件事情非常困难，并且效率很差

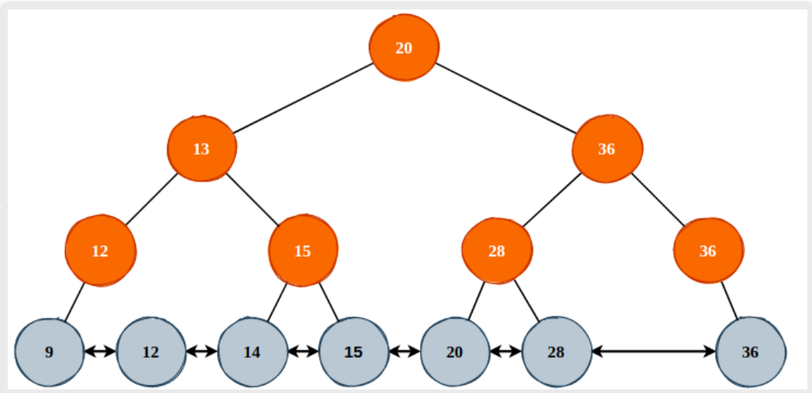
二分搜索树与多叉树

既然顺序数组在内存和硬盘之间的转换上有困难，那么我们可以尝试使用二分搜索树来做这件事情。当数据需要保存在磁盘时，可以使用文件偏移量来表示左、右指针



- 这时候元素的查询、插入以及更新均可以在 O(logn) 的时间内完成了，并且即使是放到硬盘上，也只需要数次 I/O
- 比如说我们要查询 36 这条数据，首先一次 I/O 操作取到根节点，判断大小，而后取右子树的文件偏移量，再一次 I/O 操作找到节点 28，最终找到目标节点
- 也就是说，我们需要 logn 次 I/O 操作才能把数据从硬盘中找出

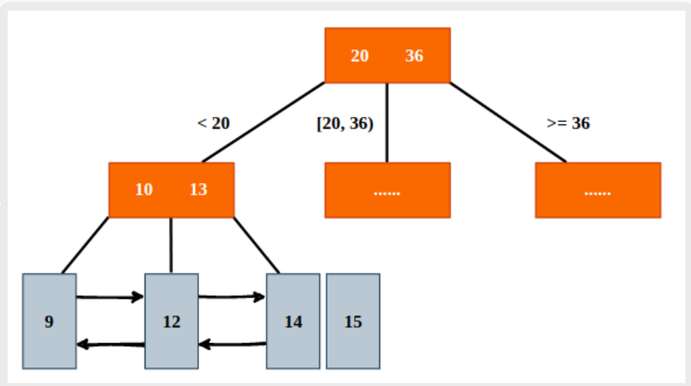
二分搜索树实现的 DB 存在一个问题: 范围查询会很慢。我们需要不断地从根节点出发，然后往下遍历。所以我们稍微改造下，数据只保存在叶子节点上，并且用双向链表进行连接



- 这么一来不管是精确查询还是范围查询都能工作了，并且效率不会太差
- 这东西看起来其实非常像一个跳跃表，不过有意思的是，skiplist 出现时间要晚于 B+Tree

对于二分搜索树实现的 DB 而言，其查询效率是与树高有关的。假设我们有 2000 万条数据，我们大概需要一棵树高为 25 的 BST 才能装下所有数据和索引。也就是说，我们至少需要 25 次硬盘 I/O 才能取出一条数据

所以，现在我们的优化方向就是降低树高，使我们的树结构变得更加"矮胖"，其实也就是把二叉搜索树变成"多叉搜索树"



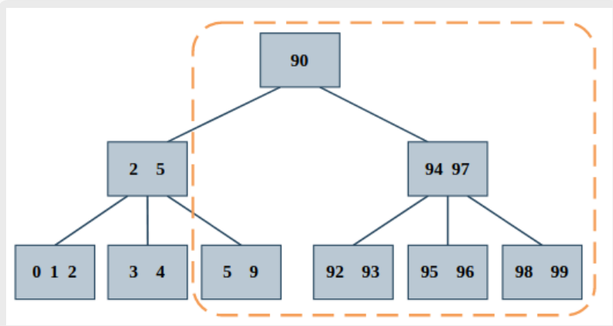
- 最后，我们得到了一个"多叉搜索树"，这玩意儿其实就是 B+Tree
- B+Tree 的特点就是只有叶子节点会存储数据，并且使用双向链表将所有的叶子节点连接在一起，以优化范围查询

B-Tree 与 B+Tree

B-Tree 其实就是一棵多叉平衡搜索树，并且每一个节点都可以保存数据，并不像 B+Tree 那样，仅在叶子节点保存数据，而在非叶子节点保存索引

B-Tree 和 B+Tree 的最大区别其实在于范围查询时的过程与性能

- 对于 B-Tree 而言，因为节点和节点之间仅通过父、子节点指针连接，那么进行范围查询时，就必须不断的进行"回溯"过程，会带来较多的随机 I/O，其范围查询过程可以参考线段树的查询过程
- B+Tree 就简单很多了，通过叶子几点的 prev、next 指针往前或者是往后遍历即可取出所有数据



如图所示，假如我们需要在该棵 B-Tree 中查询范围在 [5, 100] 之间的数据的话，我们就需要递归多次，并且需要在左右子树中同时进行。如果数据量很大的话，每一个获取左、右孩子节点都是一个随机 I/O

当然，B-Tree 在少量数据时的查询性能要优于 B+Tree，因为 B+Tree 必须要走到叶子节点才可取出数据，而 B-Tree 则在节点中即保存数据，那么不可能每一次查询都遍历到叶子节点。etcd 中的 MVCC 就是综合使用 IndexTree 这棵 B-Tree 实现的