B-Tree 和 B+Tree

Dezeming Family

2023 年 4 月 9 日

DezemingFamily 系列文章和电子书**全部都有免费公开的电子版**，可以很方便地进行修改和重新发布。如果您获得了 DezemingFamily 的系列电子书，可以从我们的网站 [https://dezeming.top/] 找到最新的版本。对文章的内容建议和出现的错误也欢迎在网站留言。

目录

一　B-Tree

B-Tree 读作"B Tree"，中间是横杠，不是减号。这是一种最开始为磁盘搜索而优化的搜索树，但我们从一般化的应用角度来讲解 B-Tree 的相关原理。

当涉及到存储和搜索大量数据时，传统的二进搜索树由于性能差和内存使用率高而变得不符合实际需求。B 树，也称为平衡树 (Balanced Tree)，是一种专门为克服这些限制而设计的自平衡树。

与传统的二进搜索树不同，B 树的特点是它们可以在单个节点中存储大量的 keys，这就是为什么它们也被称为"large keys"树。B 树中的每个节点都可以包含多个 keys，这使得树具有更大的分支，从而具有更浅的高度。这种较浅的高度会减少磁盘 I/O（后面会再详细描述其原因），从而加快搜索和插入操作。B 树特别适合于具有缓慢、庞大数据访问的存储系统，如硬盘驱动器、闪存和 CD-ROM。

B 树通过确保每个节点都有最小数量的 keys 来保持平衡，因此树总是平衡的。这种平衡保证了插入、删除和搜索等操作的时间复杂性始终为 $O(\log N)$，而与树的初始形状无关。

平衡二叉搜索树 (Balanced Binary Search Tree) 的插入、删除和搜索时间复杂度在 $O(\log N)$ 量级，而且需要保证一开始树就要建立为平衡的。它的效率确实很高，但是，对于文件目录存储这种内存开销很大的数据，比如现代主机或者数据库中的文件索引结构的存储，都不可能在内存中建立查找结构，而是需要在磁盘中存储。

如果要在磁盘中读取文件，需要先在磁盘中查找结构，从一个节点指向另一个节点时，需要先读取到内存中，然后再比较，从而再去读取下一个节点，直到找到数据。这样的频繁的 I/O 操作的效率非常低，由此，B 树就诞生了。

准确来说，B 树主要用于数据库系统；还有为了文件系统改良的 B+ 树。

1 1　B 树的一些特点

B 树要求所有的叶节点都在同一层级上，也就是都在最底层。但这不是说指向数据的节点都在最底层，每一层非空节点都有指向数据。

先假设一个 t 值。除了根节点，所有的其他节点都至少有 $t-1$ 个 keys（一般根节点只有一个 keys）；包括根节点在内所有节点最多有 $2t-1$ 的 keys。

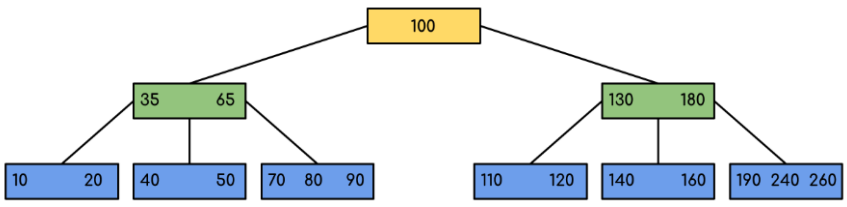
keys 可以理解为数据的用于比较大小来划分的值。

每个节点的子节点数等于该节点的 keys+1（后面会详细解释）。

一个节点的所有 keys 都是按照递增的顺序来排序的，新节点的插入只会发生在叶节点上（后面会详细解释）。

1 2　一个简单的示例

以下图为例：



可以看到，所有的叶节点都被放在了同一层上，也就是最底层。

除了叶节点，其他节点都是描述的一个范围，比如第二层左边那个绿色节点，每个节点有 2 个 keys，因此子节点有 3 个，其中第一个子节点都是小于 35 的数据，第二个子节点是 35 到 65 之间的数据；第三个子节点是大于 65 的数据（但是又得小于父节点 100）。

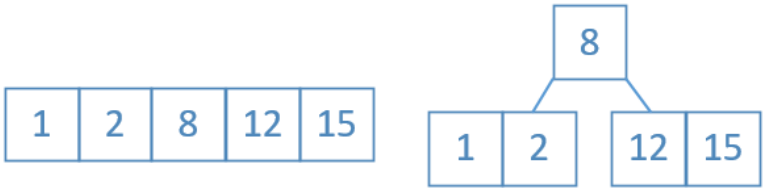
1 3　B 树的构建、增加元素

对于不同的应用，有不同的 B 树构建方式。我们讲解一下最常见的构建方式，要求每个节点的 keys 数为 2 到 4 个。

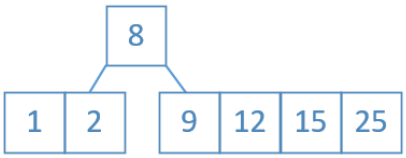
设我们构建 B-Tree 的数据：[1,15,8,12,2,9,25,13,63,51,19,7,5]。

插入前四个以后，得到的结构是：

再插入 2 以后，由于 keys 数大于 4，需要把中间的元素 8 挤上去：

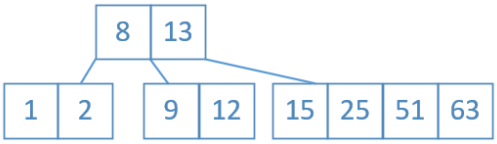


再插入 9 和 25 以后，都还不用变：



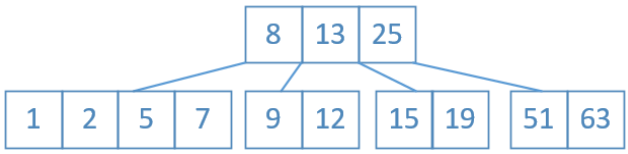
插入 13 以后，右子树节点 keys 又大于 4 了，因此再把 13 挤上去：

再插入 63 和 51，都没有变化：



再插入 19 以后，右子树节点 keys 大于 4，于是把中间的 25 挤上去：

插入 7 和 5：

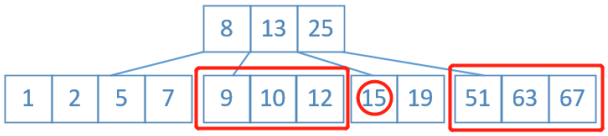


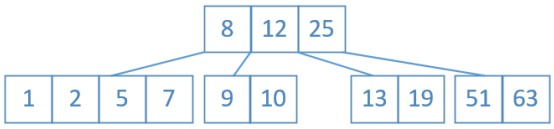
1 4　B 树删除元素

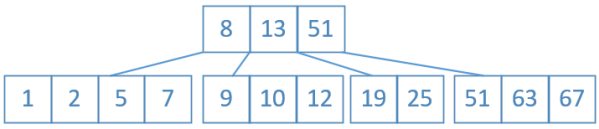
删除会复杂一些，因为会牵扯到如何填补的问题。

比如上面构建的 B-Tree，如果我们删除 5 或者 7，树的结构是不需要再调整的，因为满足“每个节点的 keys 为 2 到 4 个”的条件。但是如果删除 15，情况会变复杂。

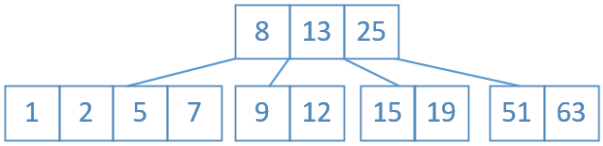
如果一开始是下面的结构，那么可以从左边或者右边“借”一个元素下来，从左边借和从右边借的结果分别表示在下图的第二行：



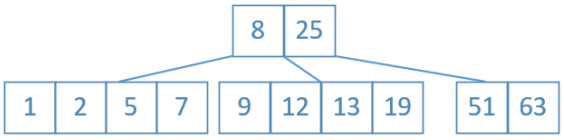


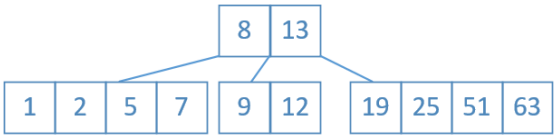


但是如果左右两边都没有元素可以借：

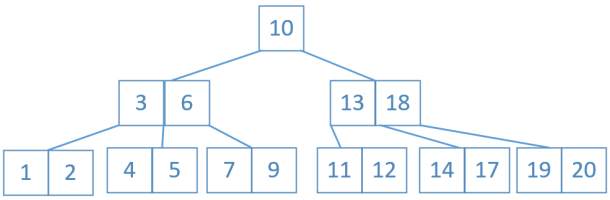


此时可以将 15 所在子树与左边子树合并或者与右边子树合并，分别见下图左图和右图：

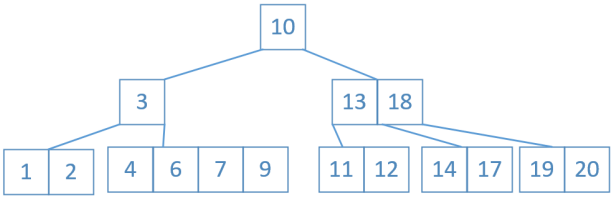




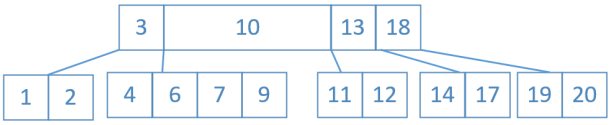
对于下面的情况，删除会更复杂一些，比如要删除 5：



此时需要与旁边的子树合并，但是合并完以后，父节点又不满足“每个节点的 keys 为 2 到 4 个”这一条件了：



继续向上合并：



二　B+Tree

B+Tree 是 B 树的提升版，对于文件系统查找而言效率更高。

2 1　与 B-Tree 的区别

我们介绍一下它与 B-Tree 的区别：

（1）非叶子结点的子树指针与 key 个数相同；

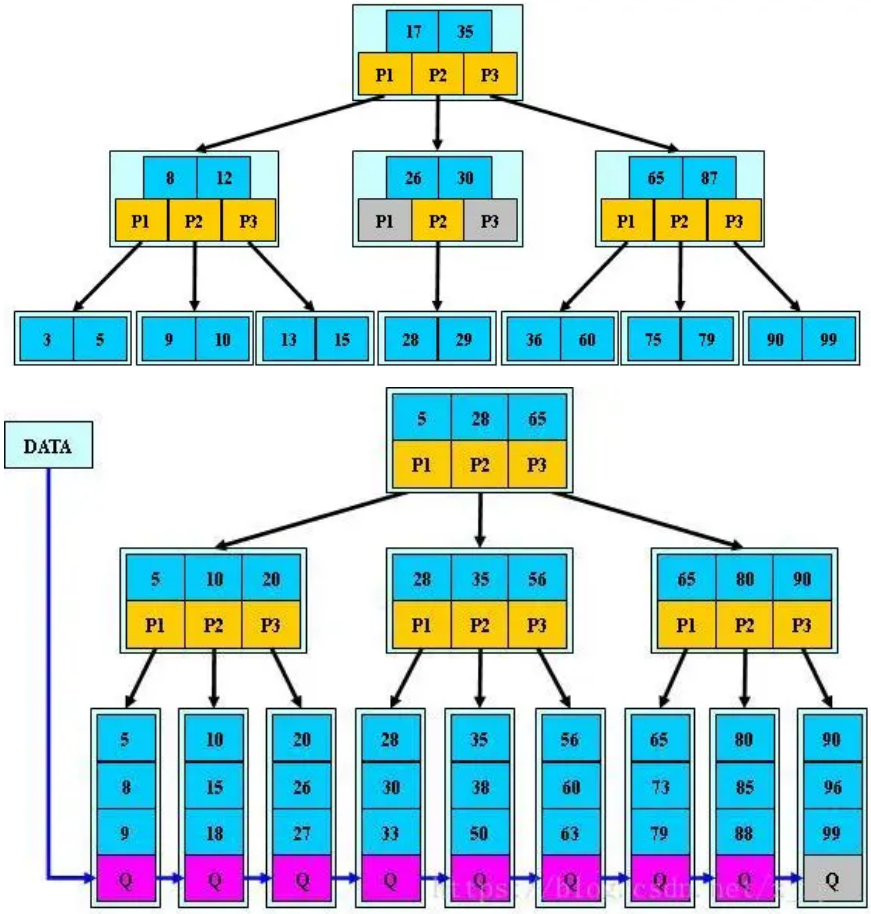
（2）非叶子结点的子树指针 P[i]，指向 key 值属于 $[K[i], K[i+1])$ 的子树（B 树是 $(K[i], K[i+1])$）；

（3）所有叶子结点都有一个链指针；

（4）所有 keys 都在叶子结点出现；

（5）数据只在叶子节点上存在，在中间节点上只有指针和 keys，没有数据。

对比一下 B 树（下图第一行）和 B+ 树（下图第二行），就能认清它们的区别：



这样我们就可以很容易地看出，B+ 树只有达到叶子结点才命中（B 树可以在非叶子结点命中）。非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层。

参考文献

[1] https://www.geeksforgeeks.org/introduction-of-b-tree-2/

[2] https://blog.csdn.net/z702143700/article/details/49079107

[3] https://www.jianshu.com/p/d281604aee5d