B树、B+树索引算法原理（下）

codedump.info/post/20200615-btree-2

2020年6月15日

2020-06-15
存储 存储引擎 算法与数据结构

> 这一段时间由于在阅读boltdb代码的缘故，找机会学习了B树及B+树的算法原理，这个系列会花两个篇幅分别介绍这两种数据结构的实现，其用于数据库索引中的基本原理。

在上一篇文章中，介绍了数据库索引的简单概念，以及B树的结构及核心算法，这一篇将继续介绍B树的变形B+树。

B+树的定义及性质

B+树之于B树，最大的不同在于：

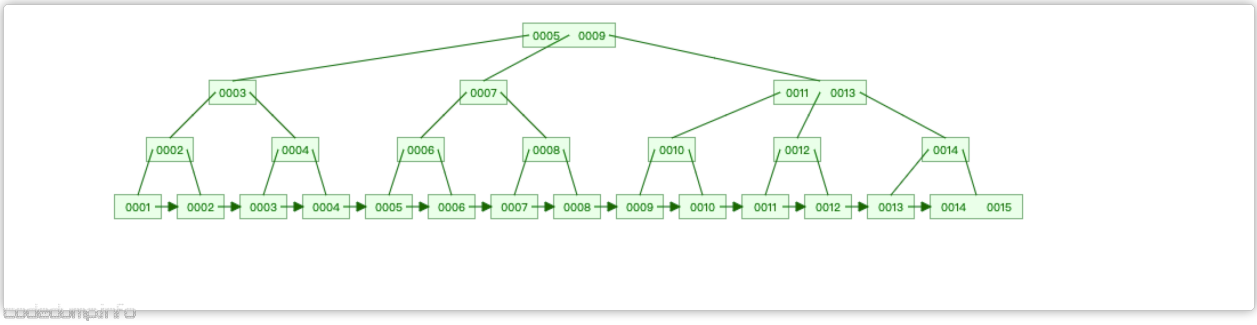
- B树的数据可以存储在内部节点上，也可以存储在叶子节点上。
- 而在B+树中，内部节点上仅存放数据的索引，数据只存储在叶子节点上。在内部节点中的键值，被称为“索引”，由于是数据索引，因此可能出现同一个键值，既出现在内部节点，也出现在叶子节点中的情况。

内部节点的“索引”，应该满足以下条件：

- 大于左边子树的最大键值；
- 小于等于右边子树的最小键值。

同时，B+树为了方便范围查询，叶子节点之间还用指针串联起来。

以下是一颗B+树的典型结构：



由于采用了这样的结构，B+树对比B树有以下优点：

- 索引节点上由于只有索引而没有数据，所以索引节点上能存储比B树更多的索引，这样树的高度就会更矮。按照我们上一篇中介绍数据库索引的内容，这种面向磁盘的数据结构，树的高度越矮，磁盘寻道的次数就会越少。

- 因为数据都集中在叶子节点了，而所有叶子节点的高度相同，那么可以在叶子节点中增加前后指针，指向同一个父节点的相邻兄弟节点，给范围查询提供遍历。比如这样的SQL语句：`select * from tbl where t > 10`，如果使用B+树存储数据的话，可以首先定位到数据为10的节点，再沿着它的next指针一路找到所有在该叶子节点右边的叶子节点数据返回。而如果使用B树结构，由于数据既可以存储在内部节点也可以存储在叶子节点，范围查询可想而知是很繁琐的。

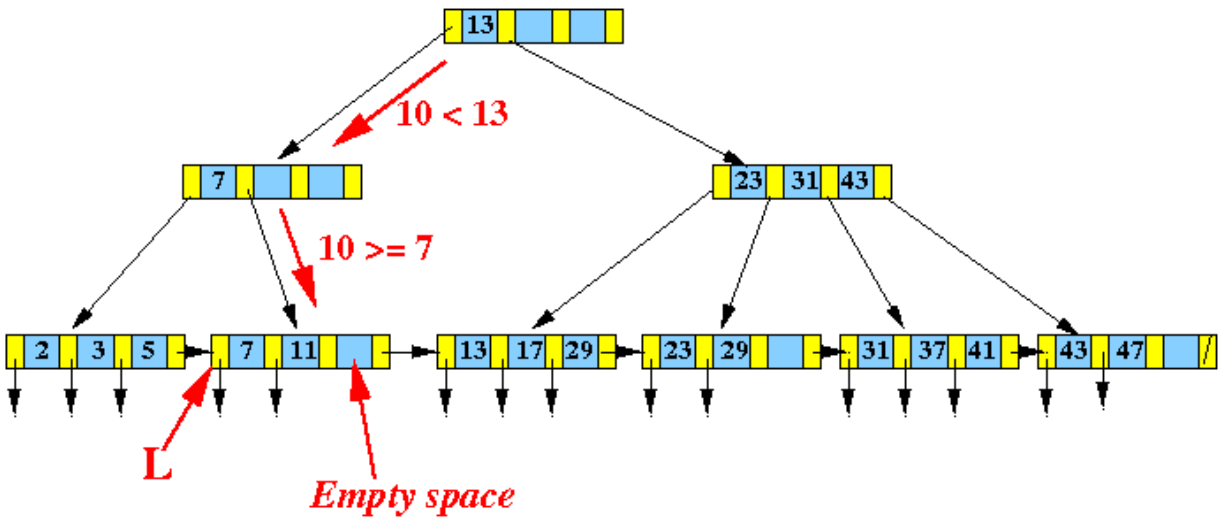
核心算法

插入算法

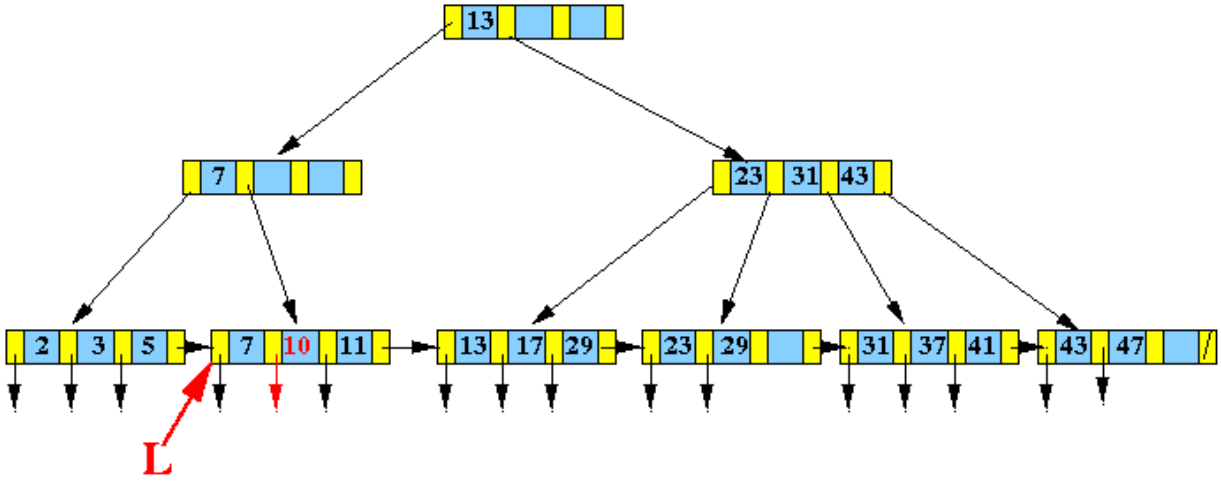
B+树的插入算法与B树的很相近，都是：

- 首先判断待插入数据节点是否已经溢出，如果是就首先拆分成两个节点，然后再插入数据。
- 由于内部节点上的数据是索引，所以在插入完成之后调整父节点指针。

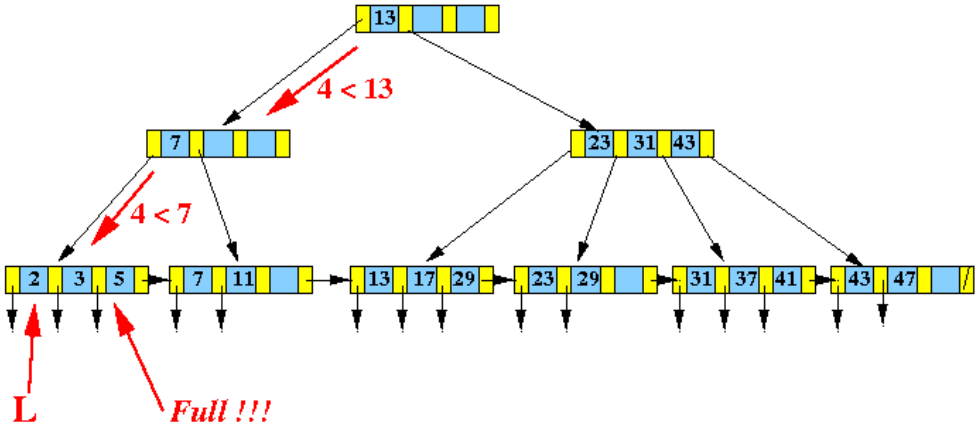
比如在下图的B+树中，向这里插入新的数据`10`：



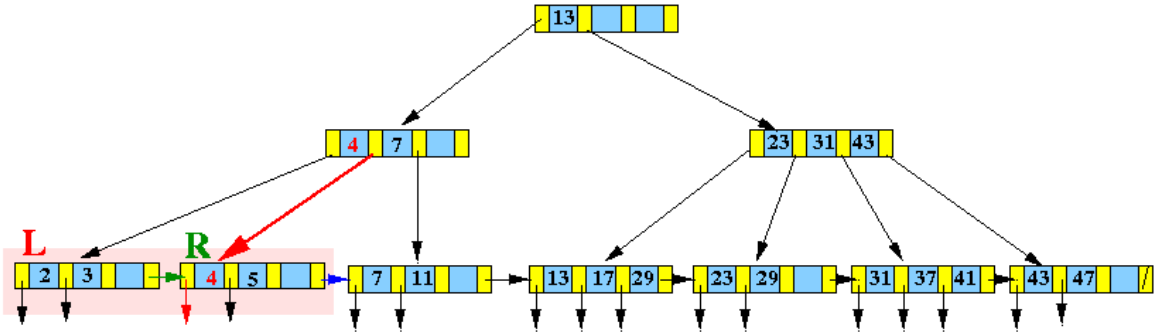
由于插入节点`[7,11]`在插入之后并没有溢出，所以可以直接变成`[7,10,11]`：



而如下图的B+树中，插入数据`4`：



由于所在节点`[2,3,5]`在插入之后数据溢出，因此需要分裂为两个新的节点，同时调整父节点的索引数据：



`[2,3,4,5]` 分裂成了 `[2,3]` 和 `[4,5]` ，因此需要在这两个节点之间新增一个索引值，这个值应该满足：

- 大于左子树的最大值。
- 小于等于右子树的最小值。

综上，需要在父节点中新增索引 `4` 和两个指向新节点的指针。

删除算法

B+树的删除算法，与B树类似，分为以下几步：

- 首先查询到键值所在的叶子节点，删除该叶子节点的数据。
- 如果删除叶子节点之后的数据数量，满足B+树的平衡条件，则直接返回不用往下走了。
- 否则，就需要做平衡操作：
  - 如果该叶子节点的左右兄弟节点的数据量可以借用，就借用过来满足平衡条件。
  - 否则，就只能与相邻的兄弟节点合并成一个新的子节点了。
- 在上面平衡操作中，如果是进行了合并操作，就需要向上修正父节点的指针：删除被合并节点的键值以及指针。由于做了删除操作，可能父节点也会不平衡，那么就按照前面的步骤也对父节点进行重新平衡操作，这样一直到某个节点平衡为止。

下面结合B-tree=delete1、B-tree=delete2 的图示对删除算法展开具体的分析。

从叶子节点中删除数据

从叶子节点中删除数据分为三种情况：

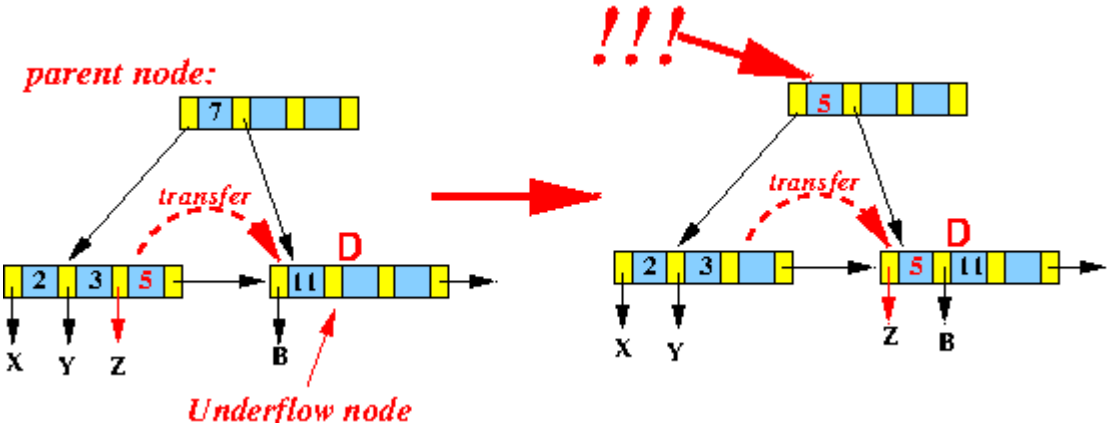
- 删除之后的数据量足够，不需要进行重平衡操作；
- 删除之后的数据量不够，但是可以从兄弟节点那里借用数据，重新达到平衡；

- 删除之后的数据量不够，兄弟节点的数据也不够，那么需要合并成一个新的节点，同时在父节点中删除索引和指针。

以下针对后面两种需要做重平衡的操作展开分析。

借用兄弟节点数据进行重平衡操作

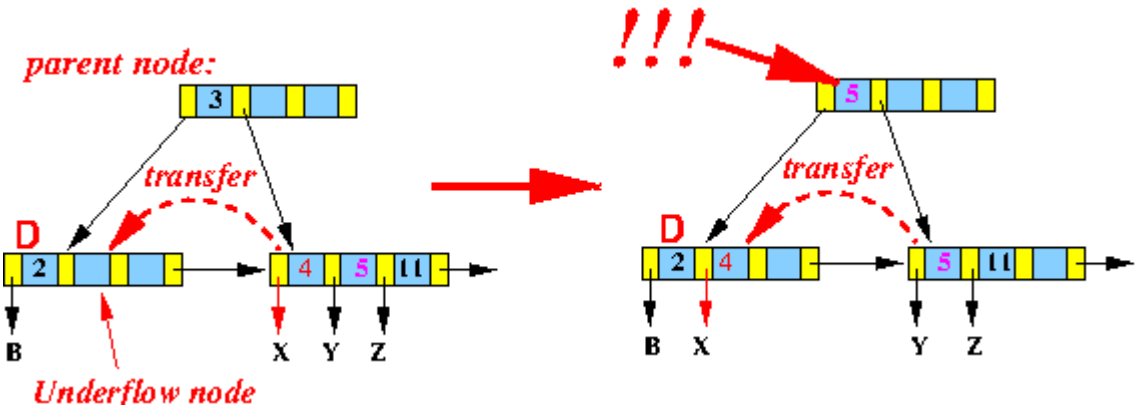
在下图中，从叶子节点中删除数据之后，只剩下数据 `[11]` ：



由于此时其左兄弟节点 `[2,3,5]` 有足够的数据可以借用，于是：

- 将数据 `5` 移动到 `[11]` 中，成为新的叶子节点 `[5,11]` 。
- 由于该叶子节点数据发生了变化，因此需要同时修正父节点中的索引数据 `7` 为 `5` 。为什么是 `5` ？因为修改的是该索引的右边子树数据，所以要取右子树数据中的最小值 `5` 。

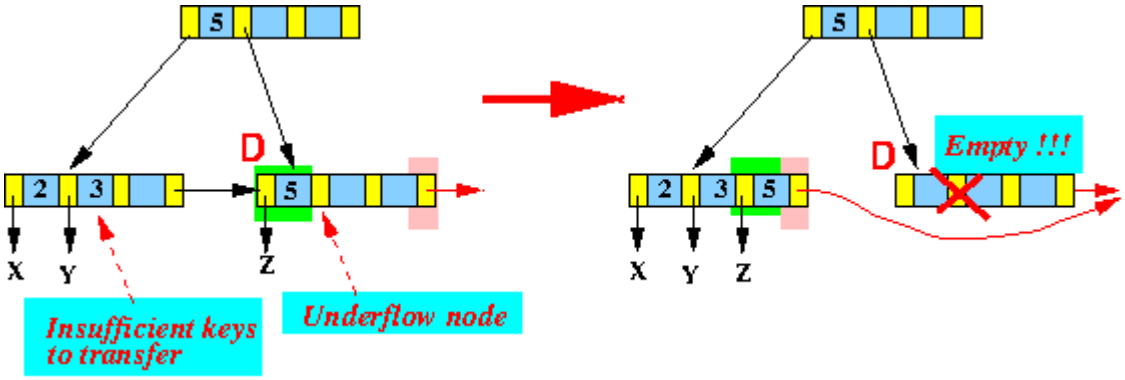
类似的，也有从右边兄弟节点借用数据的情况，如下：



与兄弟节点进行合并

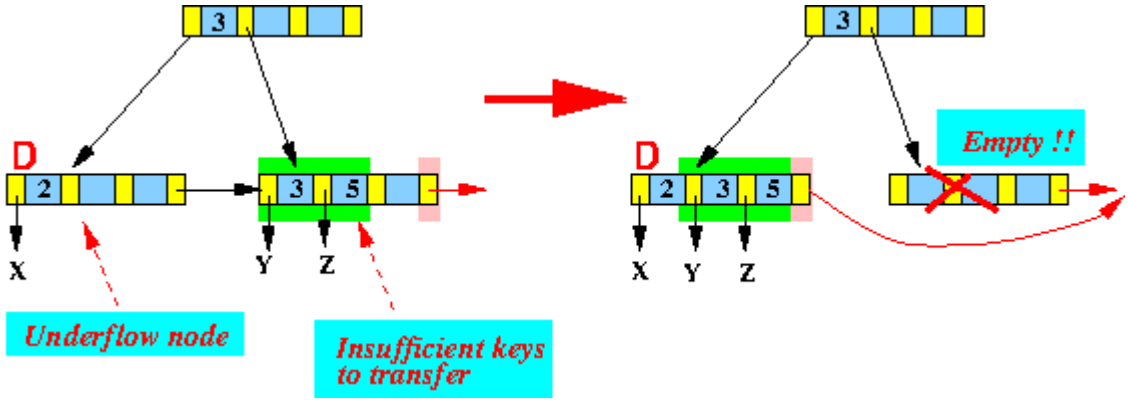
当左右兄弟节点数据都不够借用的时候，那就只能进行合并，此时会有一个节点要从父节点中删除其索引数据以及子节点指针。

在下图中，从叶子节点中删除数据之后，只剩下数据 `[5]` ：



而左边兄弟节点 `[2,3]` 的数据也不够用，于是两个节点进行了合并，形成新的节点 `[2,3,5]` ，这样节点 `[5]` 就要在父节点中被删除。

类似的，也有合并到右边的情况：



上面从叶子节点中删除数据的操作，一共分为以下三种情况：

- 被删除节点的数据量足够，不需要做重新平衡操作。
- 被删除节点的数据量不够，但是可以借用兄弟节点数据达到重平衡。
- 被删除节点的数据量不够，兄弟节点的数据也不够，只能把两者合并成新的节点。

这三种情况中，前面两种由于并没有调整父节点，删除其中的索引和子节点指针，因此不需要继续在父节点中做重平衡操作，而第三种情况由于合并节点导致父节点需要删除数据，所以需要进一步在父节点中进行重平衡操作。以下继续以例子展开说明。

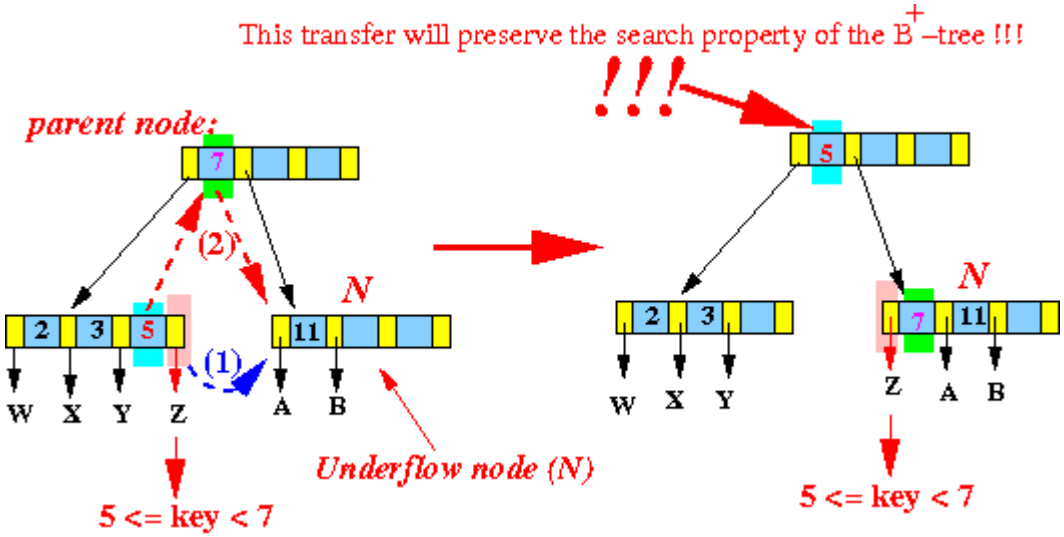
内部节点节点的重平衡

在删除父节点中的索引和子节点指针之后，如果父节点中的数据足够，同样也是不需要进行调整的，下面讨论的是内部节点需要进行重平衡的两种情况。

借用兄弟节点数据进行重平衡操作

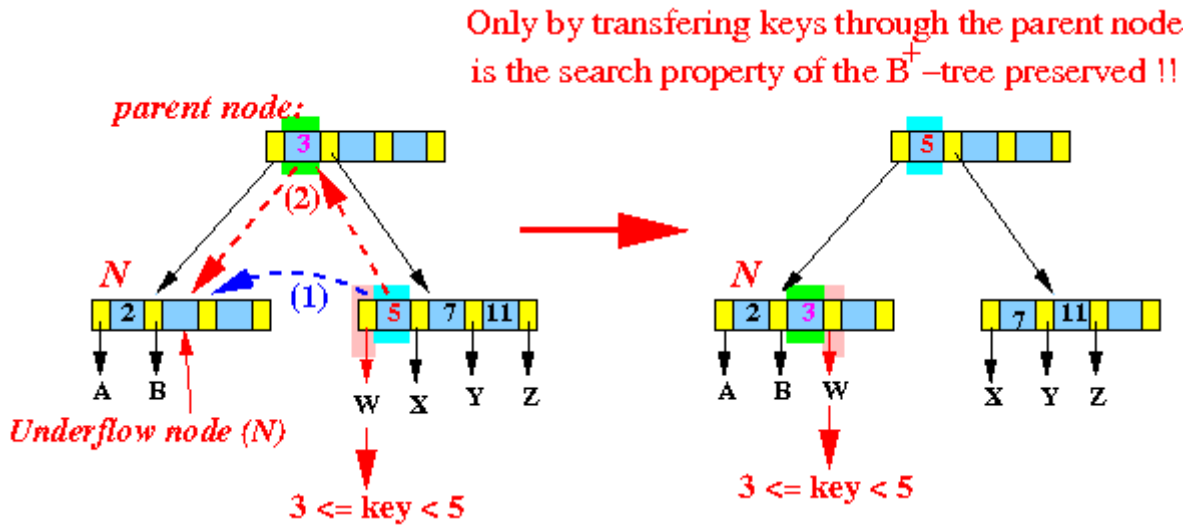
如果兄弟节点数据足够，那么同样可以从兄弟节点借用数据进行重平衡操作。

以下图为例，假设内部节点 `[11]` 在删除索引和指针之后，需要从兄弟节点 `[2,3,5]` 借用数据：



大体的流程其实与叶子节点的借用数据挑战差不多，只是内部节点有指向子节点的指针，也要随着一起调整。

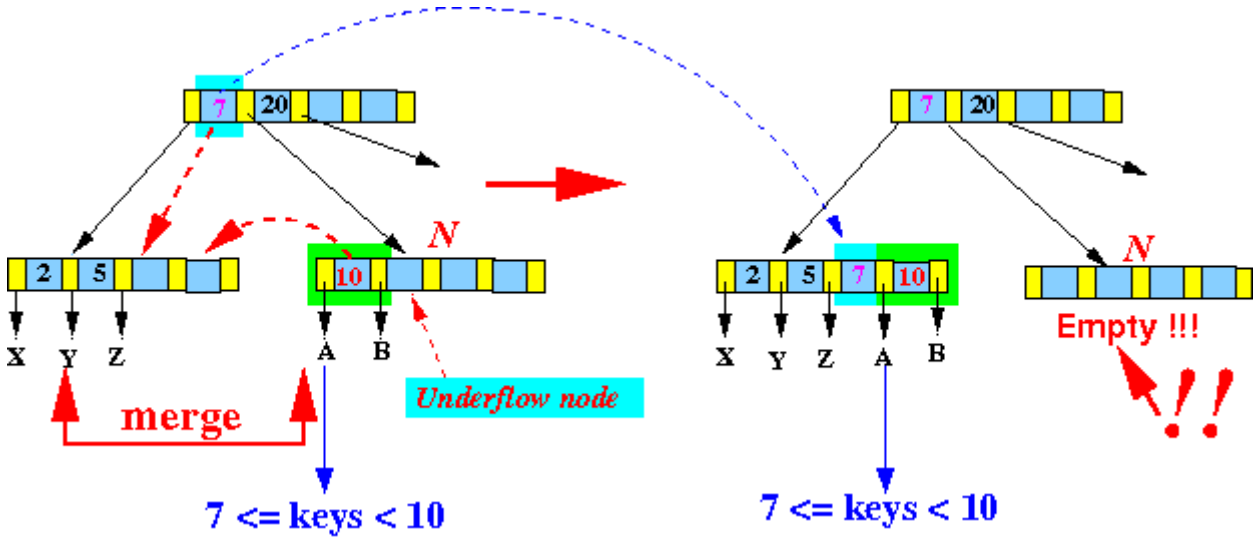
同样的，也有从右边兄弟节点借用数据的情况：



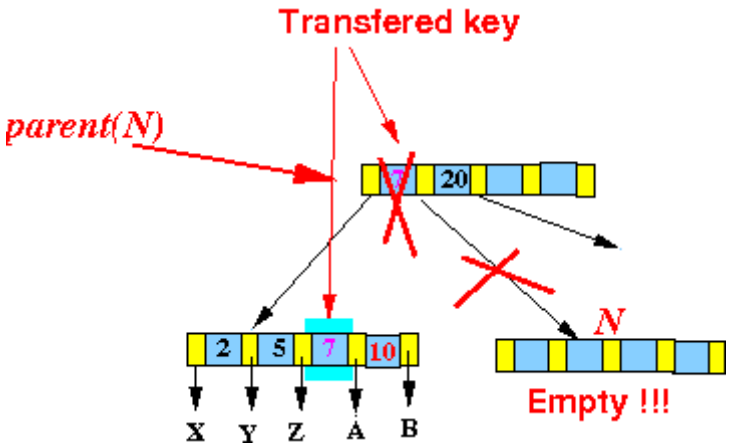
与兄弟节点进行合并

当兄弟节点数据不足时，内部节点也是进行合并操作。

下图中，节点 `[10]` 的数据量不足，而兄弟节点 `[2,5]` 也不够数据借用，只能将两者合并，同时调整父节点指针：



合并之后的图示：



总结

相比B树的算法，B+树算法更简单一些，删除节点后的节点分为三种情况：

- 数据量足够，不需要进行重平衡操作，这种情况不用继续到父节点中进行调整；
- 数据量不够，但是可以从兄弟节点借用数据重新满足平衡条件，这种情况不用继续到父节点中进行调整；
- 数据量不够，同时兄弟节点的数据也不够，此时只能将两者合并成一个新的节点，而由于两个节点变成了一个节点，所以要对应的从父节点中删除索引和子节点指针，这样继续往上看看是否需要对父节点进行重平衡操作，一直到满足平衡条件位置。

我同样将B+树算法写了一个python的实现，参见：bplustree.py

参考资料

emory大学的B+树算法讲解，是我能找到的最好的B+树算法讲解：

- delete1
- delete2
- delete3

在这里能找到该课程更多关于B+树的课件索引。

B+树的动画演示

Author codedump

LastMod 2020-06-15

相关文章

- **2022-02-01:** sqlite3.36版本 btree实现（五）- Btree的实现
- **2022-01-06:** sqlite3.36版本 btree实现（四）- WAL的实现
- **2021-12-22:** sqlite3.36版本 btree实现（三）- journal文件备份机制
- **2021-12-18:** sqlite3.36版本 btree实现（二）- 并发控制框架
- **2021-12-17:** sqlite3.36版本 btree实现（一）- 管理页面缓存
- **2021-12-17:** sqlite3.36版本 btree实现（零）- 起步及概述
- **2020-07-26:** boltdb 1.3.0实现分析（四）
- **2020-07-25:** boltdb 1.3.0实现分析（三）
- **2020-07-11:** boltdb 1.3.0实现分析（二）
- **2020-06-25:** boltdb 1.3.0实现分析（一）
- **2020-06-09:** B树、B+树索引算法原理（上）
- **2019-02-15:** Leveldb代码阅读笔记

邮件订阅

微信公众号