B树、B+树索引算法原理（上）

codedump.info/post/20200609-btree-1

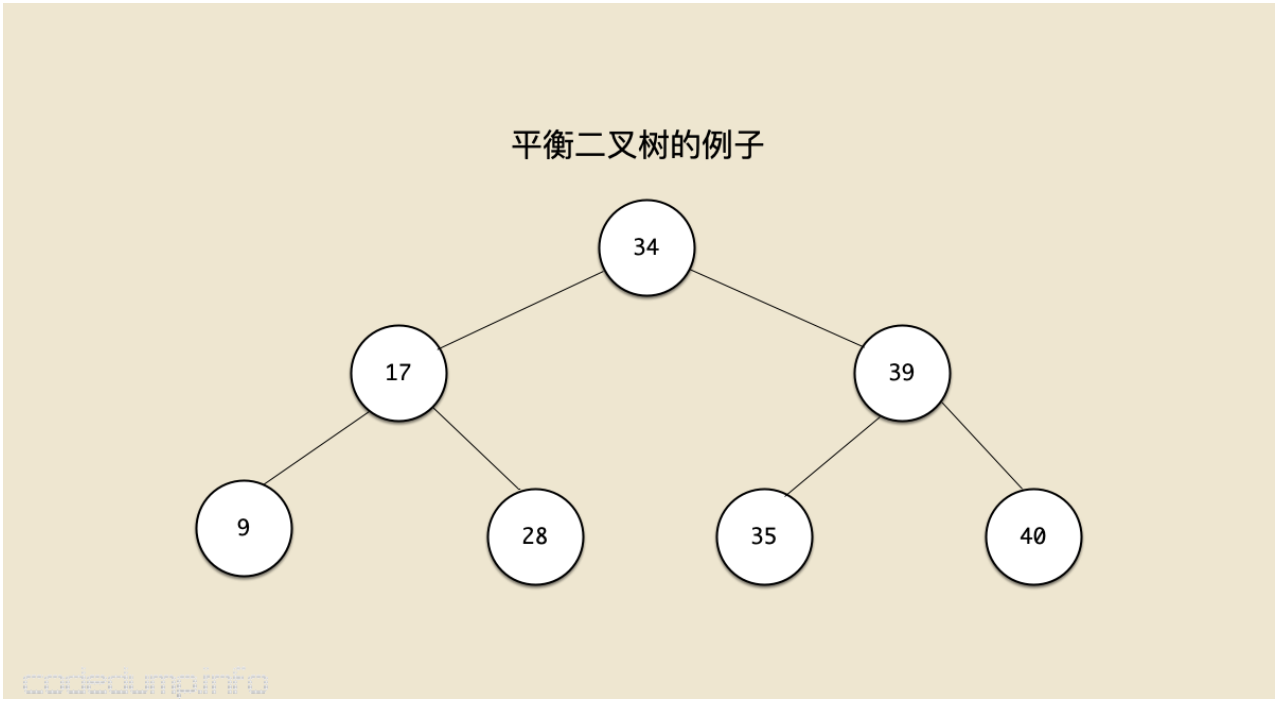
2020年6月9日

2020-06-09
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。

B树数据库索引原理

在一堆数据中查找一个数据时，常用的数据结构有二叉查找树（binary search tree，简称BST）、哈希桶等。以BST为例，常见的实现有AVT、红黑树等，由于这类型的树是平衡的，每次比较操作都会去掉当前数据量一半的数据，因此查找的时间复杂度为`O(log2n)`。



但是这类型数据结构的问题在于，由于每个节点只能容纳一个数据，导致树的高度很高，逻辑上挨着的节点数据可能离的很远。如果是在内存中操作数据的话，这样问题并不大。

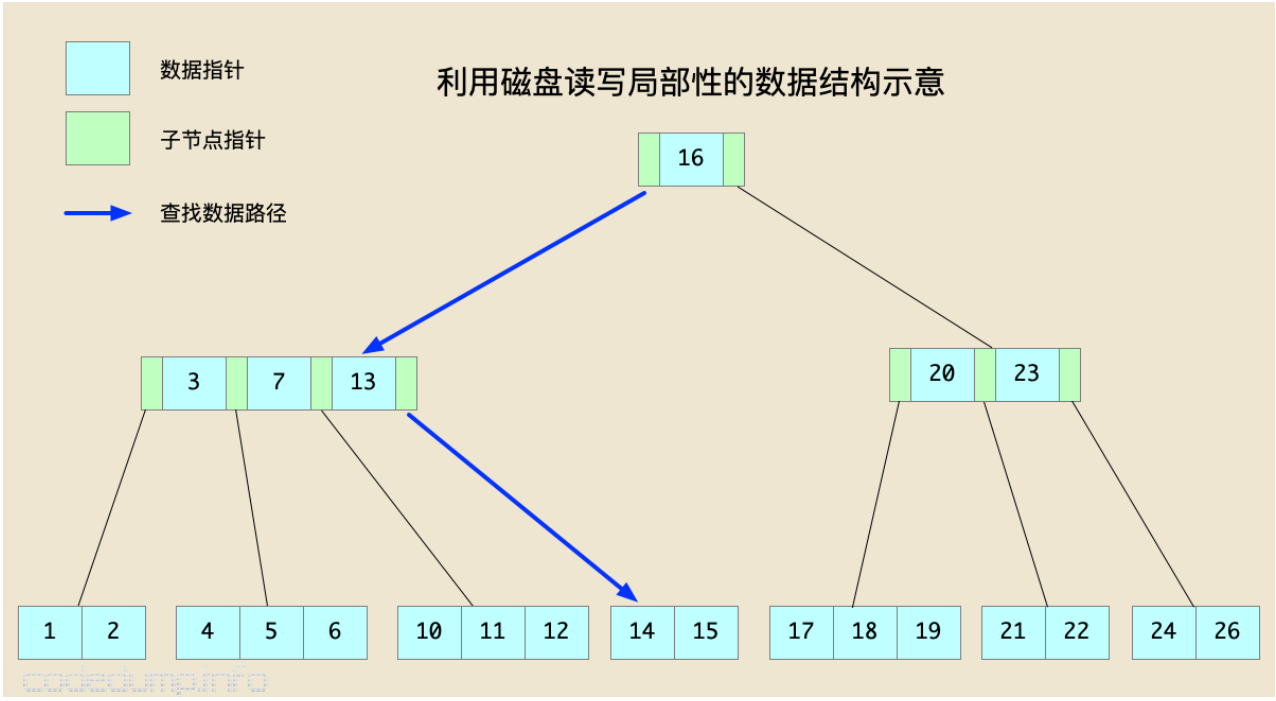
考虑在磁盘中存储数据的情况，与内存相比，读写磁盘有以下不同点：

- 读写磁盘的速度相比内存读写慢很多。
- 因为上面的原因，因此每次读写磁盘的单位要比读写内存的最小单位大很多。

因为读写磁盘的这个特点，因此对应的数据结构应该尽量的满足“局部性原理”：“当一个数据被用到时，其附近的数据也通常会马上被使用”，为了满足局部性原理，应该：**将逻辑上相邻的数据在物理上也尽量存储在一起**。这样才能减少读写磁盘的数量。

所以，对比起一个节点只能存储一个数据的BST类数据结构来，要求这种数据结构在形状上更“胖”、更加“扁平”，即：每个节点能容纳更多的数据，这样就能降低树的高度，同时让逻辑上相邻的数据都能尽量的存储在物理上也相邻的硬盘空间上，减少磁盘读写。

以下图为例：



图中从根节点出发，查找数据14的过程中，经过的第二个节点中有键值`[3,7,13]`，这三个值在“逻辑”上是相邻的，如果它们在磁盘上的存储也能做到在“物理”上相邻，那么只需要一次读操作就能把这个节点的数据从磁盘上加载到内存中进行数据比较，这样整个查找过程就只需要两次磁盘读操作。

在这里，一个节点越“胖”，意味着扇出（fanout）越大，同时高度越低，这两个性质决定了：

- 高扇出：邻近键值的数据局部性更好。
- 低高度：遍历期间的寻道次数更少。

可以证明，查找数据的次数（searchnum）与degree、以及数据总量有以下关系：

$$\log_t num = searchnum$$

B树和B+树就是两种利用磁盘局部性原理进行优化的树结构，B+树基于B树做了一些改进，这里首先将介绍B树的原理。本系列将用两篇文章讲解这两种数据结构的原理，并且提供Python实现代码。

B树的定义及性质

在B树中，分为两种节点：

- 内部节点（internal node）：存储了数据以及指向其子节点的指针。

- 叶子节点（leaf node）：与内部节点不同的是，叶子节点只存储数据，并没有子节点。

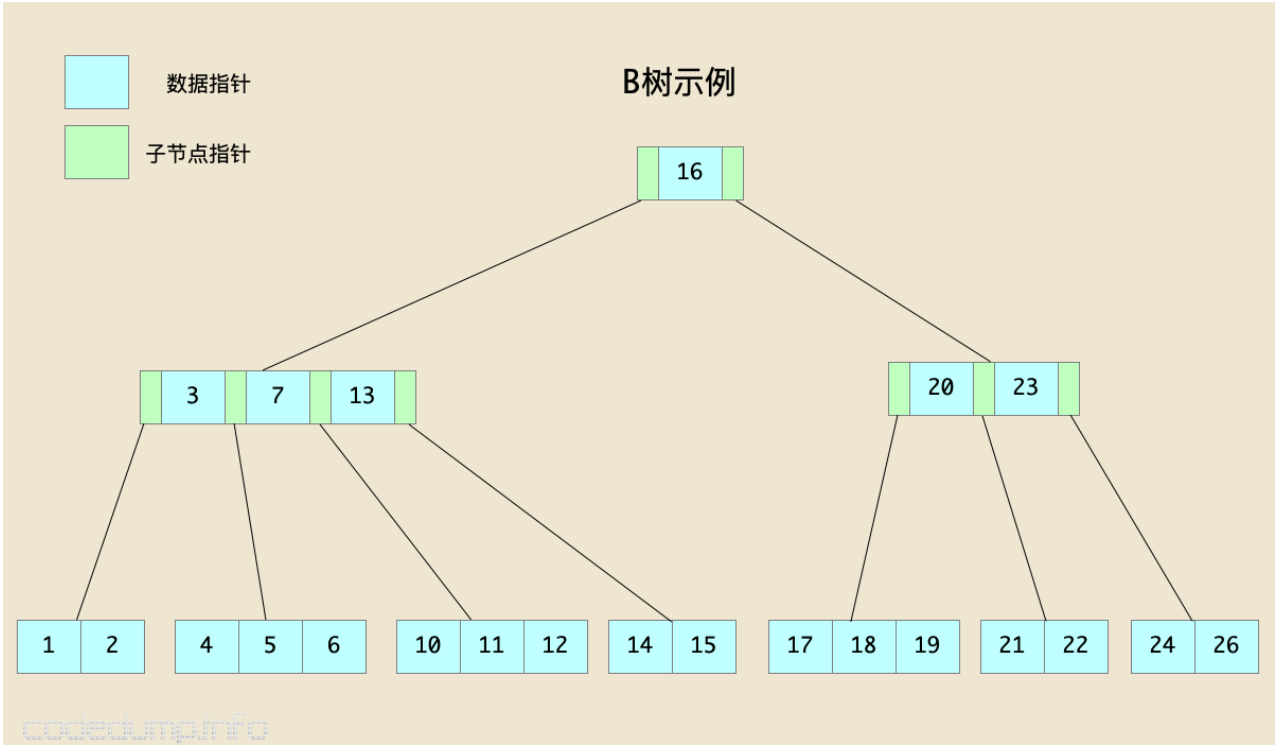
一个数据，既可能存在内部节点上，也可能存在叶子节点上，这一点是与后面讲到的B+树最大的不同，后者只会将数据存储在叶子节点上。

创建B树时，需要输入一个degree参数（以下简写为t），该参数决定了每个节点上数据量的多少，即节点的“胖”、“瘦”程度，而节点的胖瘦程度又会影响整棵树的高度，因为越胖的节点树高度就会越矮。

为了维持B树的平衡性，需要满足以下的属性：

- 在每个节点上的键值，以递增顺序排列，即 `node.keys[i] <= node.keys[i+1]` 。
- 在一个键值左边的子树，其键值大于该键值右边子树的所有键值，即 `node.keys[i] > max(node.child[i]的所有键值)` ；同时，在一个键值右边的子树，其键值的最小值都不小于该键值，即 `node.keys[i] <= min(node.child[i + 1]的所有键值)` 。具体情况可以在下面的图中进行说明。
- 在内部节点中，指向子节点的指针数量总是存储数据节点的数量+1，即： `num(node.child) = num(node.keys) + 1` 。
- 所有叶子节点的高度一致。
- 无论是内部节点还是叶子节点，其存储的键值数量在 `[t-1,2t-1]` 之间，如果数量不满足此条件，需要做重平衡操作。如果少于 `t-1` ，需要借用或合并数据；反之，如果数据量大于 `2t-1` ，则需要分裂成两个节点。

我们来看下面的图示，该图中的B树，t参数的值为2（**需要特别说明的是，一棵树中每个存储数据的地方，应该既有键值（key）也有数据（value），本文中为了简单起见，存储的数据只有键值。**）：



在上图中：

- 由于 `t=2` ，所有所有节点的键值数量在 `[1,3]` 之间。
- 所有叶子节点的高度相同。
- 以左边的内部节点为例，其第一个键值为3，即该节点的 `keys[0]=3` ，而该键值的左边子树的键值为 `[1,2]` ，都小于3，即 `keys[0]>max(child[0]的所有键值)` ；而其右边子树的键值为 `[4,5,6]` ，都不小于3，即 `keys[0]<=min(child[1]的所有键值)` 。

B树算法原理

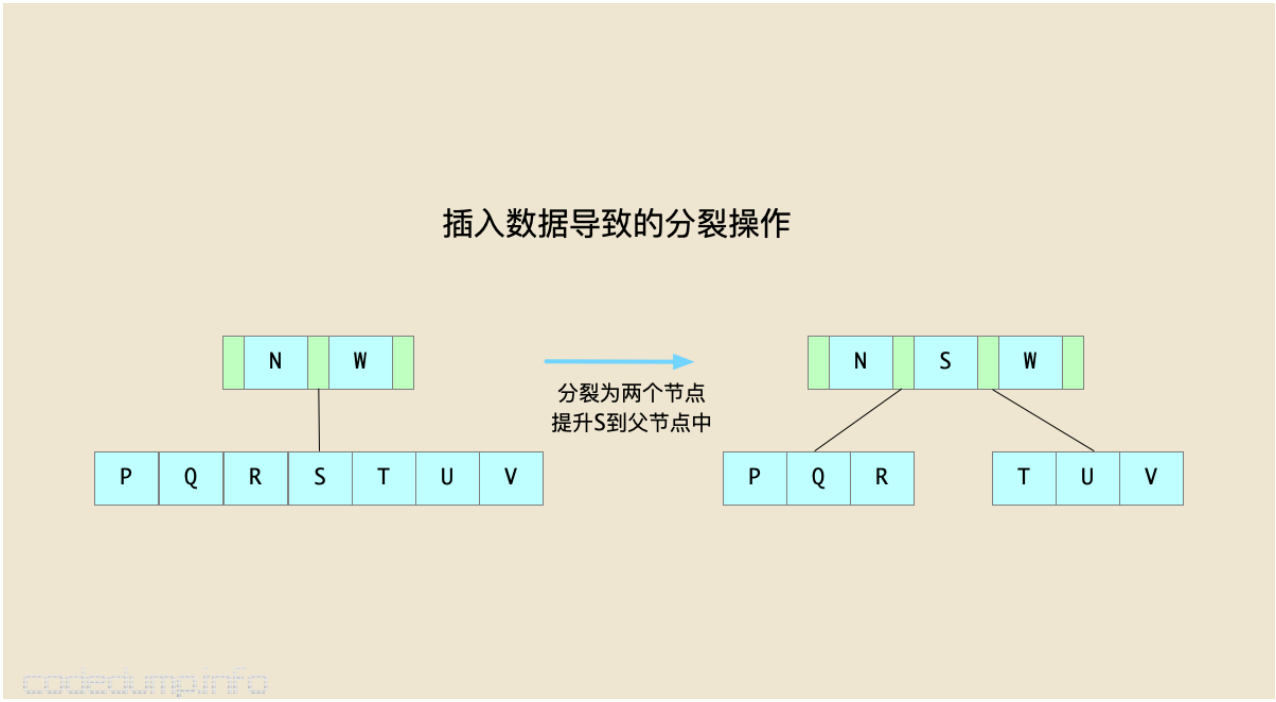
了解了B树的性质，下面讨论B树中的两个核心操作：插入及删除。这两个操作的核心，都是在操作如果破坏了B树的平衡性之后，进行重新平衡以满足B树的性质。

插入数据

向B树中插入一个数据，可能会导致节点的数据变满，即不满足上面提到的节点数据数量在 `[t,2t-1]` 这个性质。此时需要对节点进行分裂节点操作：

- 将数据变满（即节点数据量为 `2t` ）的节点，分为左右两个数据量分别为 `t-1` 的节点，同时将中间的数据提升到父节点的合适位置上。
- 如果父节点由于新增了这个被提升的数据导致了变满，就继续上面的分裂节点操作。
- 沿着树向上一直执行该操作，直到不再变满为止。

分裂操作的示意图如下：



在上图中，由于插入数据，导致节点 `[P,Q,R,S,T,U,V]` 数据量不满足平衡性要求，这时将其分裂为两个节点，同时将中间的节点S提升到父节点中形成 `[N,S,W]` ，同时修改子树指针。

因此，向B树中插入一个数据的大体流程如下：

```
向B树中插入数据：

    找到插入数据所在的最合
适节点

    如果该节点的数据量已
满：

        进行分裂操作

        插入数据

    否则：

        插入数据
```

由于《算法导论》中详细给出了插入流程的伪代码，这里就不再一并列出。

删除数据

与插入操作相同，删除操作也可能会破坏B树的性质，需要进行重新平衡操作。

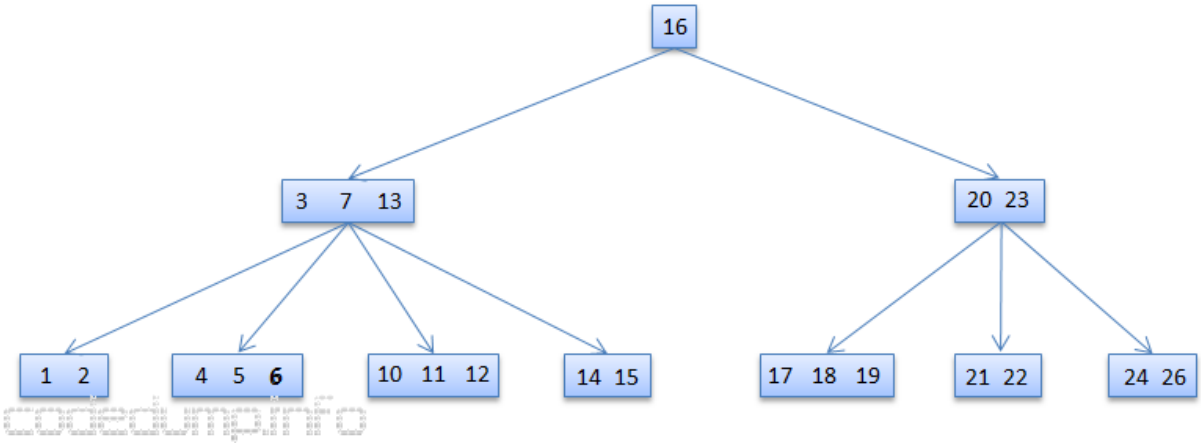
B树的删除算法，在沿着树向下查找待删除节点的流程中，依次看沿路的节点是否不满足至少有 `t` 个节点的条件，如果不满足这个条件就首先对这个节点进行平衡操作。由于在一开始就让沿路节点都至少有 `t` 个节点，这样在删除数据之后也至少能有 `t-1` 个数据满足平衡条件，这样就不用在删除之后还要回溯到祖先节点进行重平衡操作了。

我认为这是B树删除算法最核心的地方：**如果已知一个回溯操作不可避免，又无法预测到底在哪里发生，就在路上提前解决掉可能需要进行回溯操作的节点。**

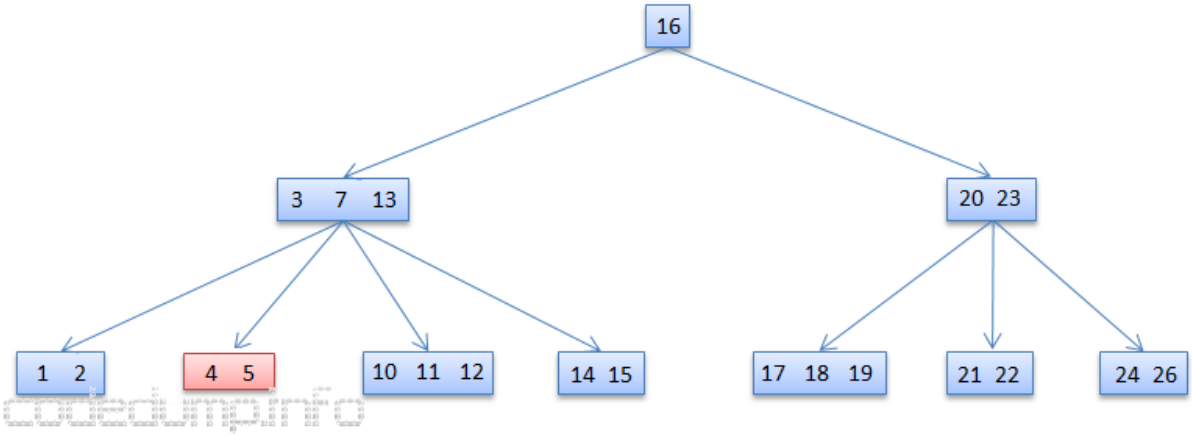
下面将在B树中删除一个数据分为以下几种情况进行讨论。

情况1：数据存在于叶子节点

在这种情况下，只需简单的从叶子节点中删除数据即可，如下图中，从节点 `[4,5,6]` 中删除数据6，最开始的图如下：



下图演示了删除数据6之后的情况：



上图只是最简单的情况，因为 `[4,5,6]` 节点数据量即使在删除一个数据之后仍然满足平衡条件，不满足平衡条件的将在下面展开说明。

情况2：数据存在于内部节点

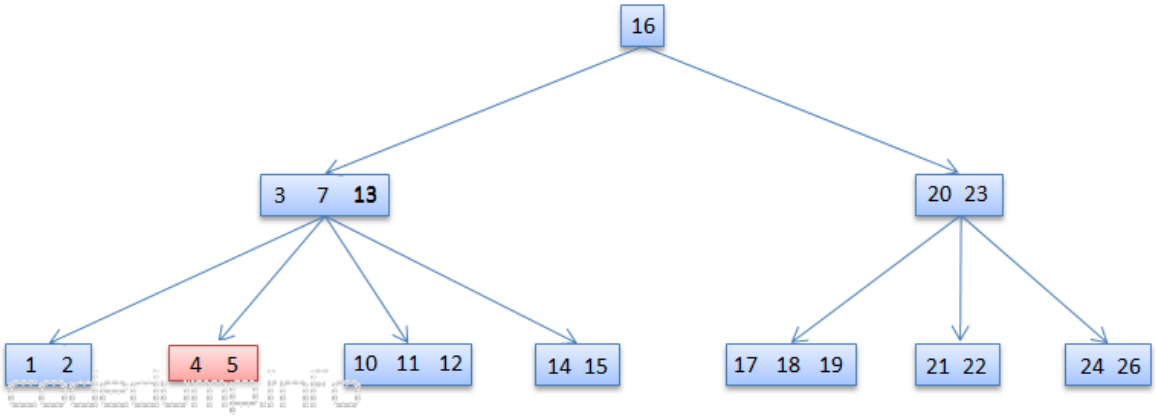
这种情况下，如果该内部节点的数据数量不大于 `t` ，则需要做重新平衡操作，这里区分成两种情况：

- 如果左右子树节点中有一个数据量至少有 `t` ，则可以从相邻子树中借用数据。
- 否则，如果左右两边相邻的子树节点数据也都不够，则进行合并操作。

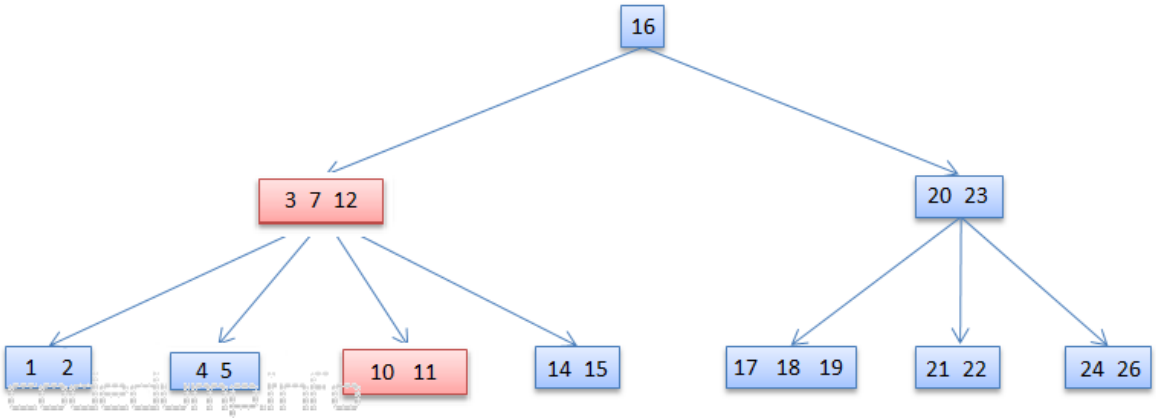
下面分这两种情况进行演示。

情况2-a：借用相邻左右子树节点

首先演示从兄弟节点中借用数据的情况，这是从 `[3,7,13]` 的节点在删除13之前的示意图：



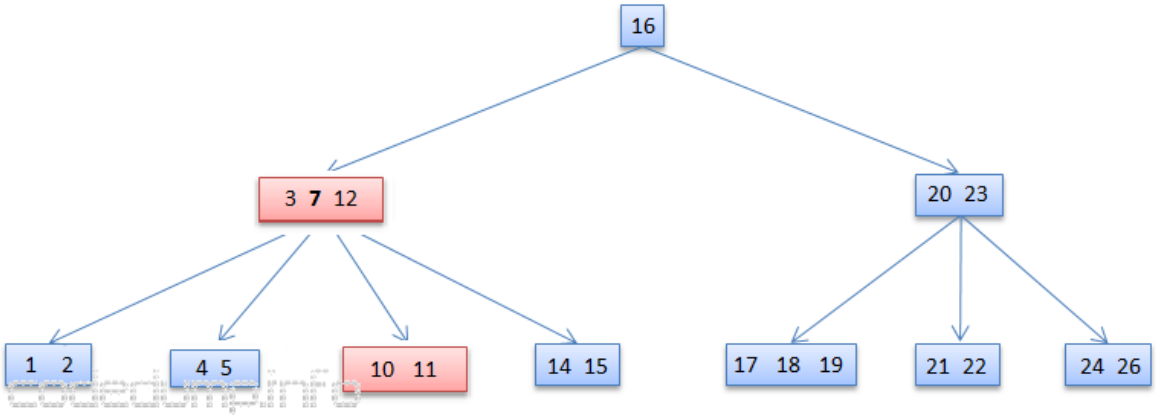
含有键值 `[3,7,13]` 的节点在删除13之后，由于其相邻子树 `[10,11,12]` 有足够的数据数量，因此可以从旁边的子树抽调了数据12，提升到该节点中替换了13，于是变成了 `[3,7,12]` ，而子树的数据也从 `[10,11,12]` 变成了 `[10,11]` ：



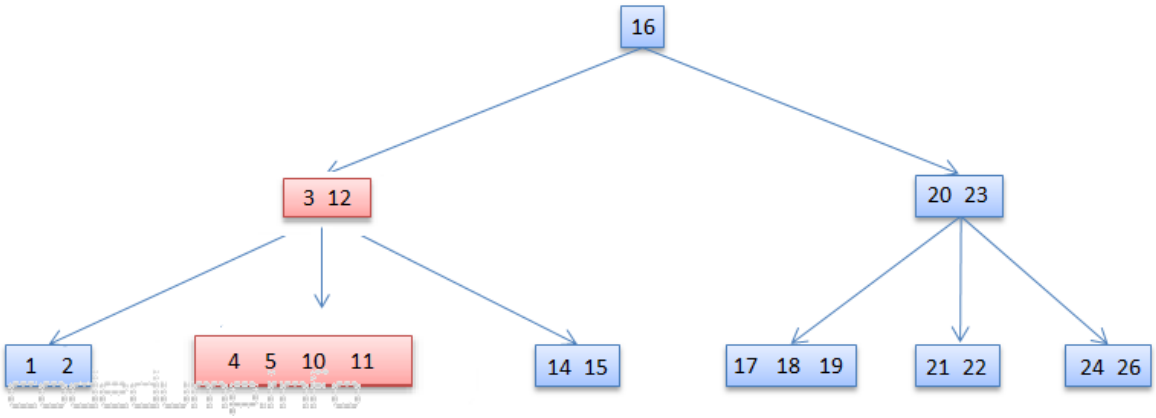
情况2-b：合并左右相邻子树节点

除了上面两种从旁边子树借用数据的操作之外，如果左右两边的子树数据量都不满足不少于 `t` 的情况，那么只能将两者进行合并了，如下图所示。

在从节点 `[3,7,12]` 中删除键值7之前的图示如下图，这时该节点的左右两边的子树 `[4,5]` 和 `[10,11]` 都不满足借用条件：



在上图的基础上，从节点`[3,7,12]`中删除键值7，由于该键值左右两个子树`[4,5]`和]`[10,11]`都不满足数据量不少于`t`的条件，所以只能将两者合并成一个节点：



情况3：数据不直接存在于某个内部节点的情况

上面两种情况，是键值直接存在于特定（叶子、内部）节点时的操作，最后一种情况则是针对它们的父节点进行操作。

在这种情况中，如果待删除数据并不存在于节点X中，那么找到必然包含该数据的X的子节点X.child[i]（假设为节点Y），如果这时候Y节点数据数量不满足t这个条件，则需要用下面两种方式进行重平衡。

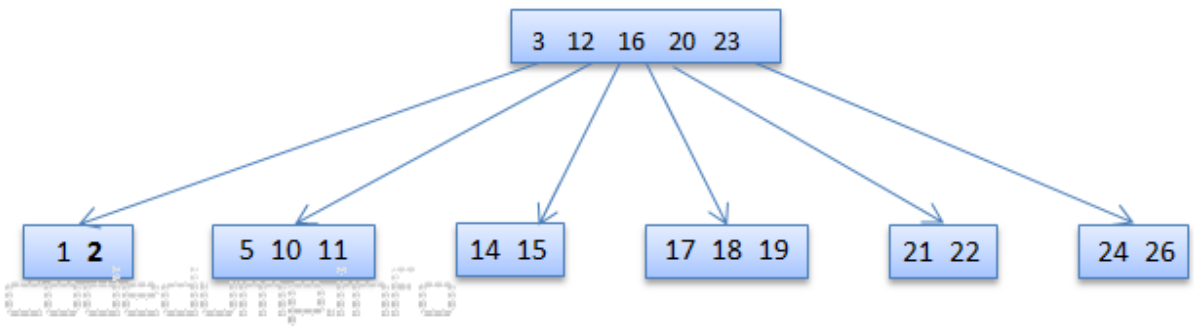
需要特别注意的是，这里的节点X、Y并不是**直接**包含待删除数据的节点，可能是待删除节点的父节点或者祖先节点，第三种情况要做的是：在查找待删除数据的路径上，发现哪些不满足数据量是`t`的节点，就地进行重平衡操作，而不用等待回溯。

情况3-a：借用兄弟节点

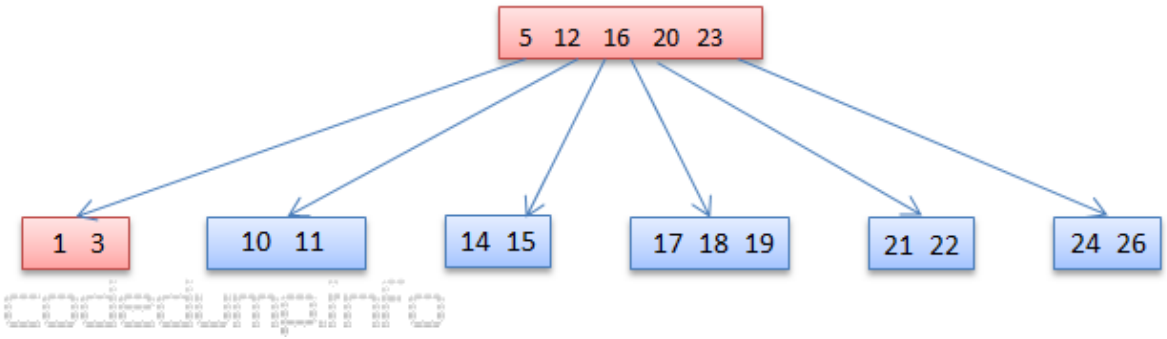
在这种情况中，如果Y节点其键值数量不满足不少于`t`的条件，同时其兄弟节点够数量借用给它，那么：

- 从父节点X中下降一个数据到Y中。
- 提升兄弟节点的一个数据到父节点X中。

假设删除数据2之前的图示如下：



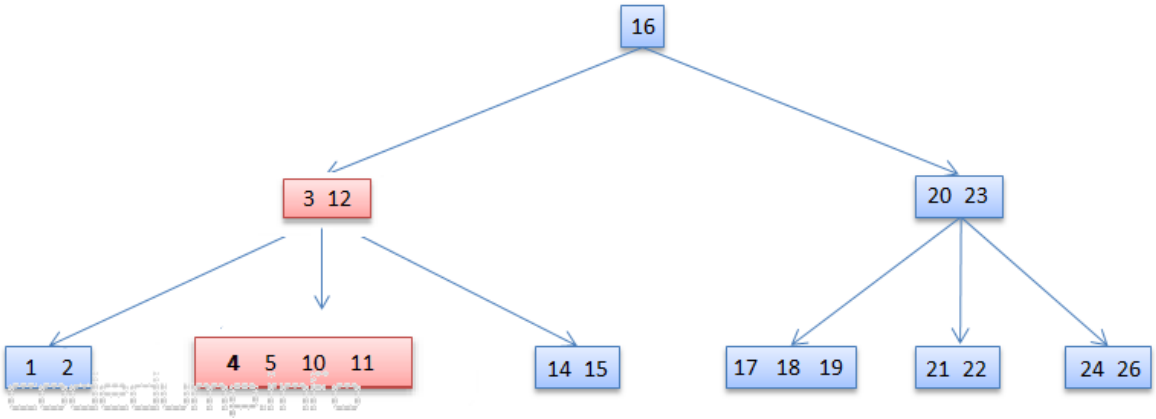
因为节点 `[1,2]` 在删除数据2之后不满足平衡条件，所以需要重平衡。其兄弟节点 `[5,10,11]` 够数量，因此可以父节点对应的数据3下降去填补删除2之后的空缺，而将数据5提升到父节点。



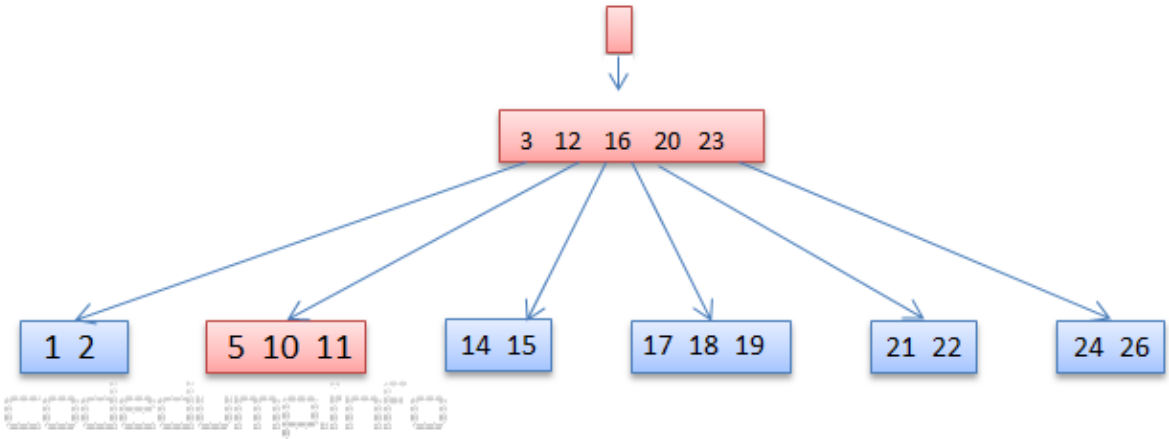
情况3-b：与兄弟节点合并数据

在这种情况下，如果节点Y和其兄弟节点的数据量都只有 `t-1` ，那么将进行合并操作。

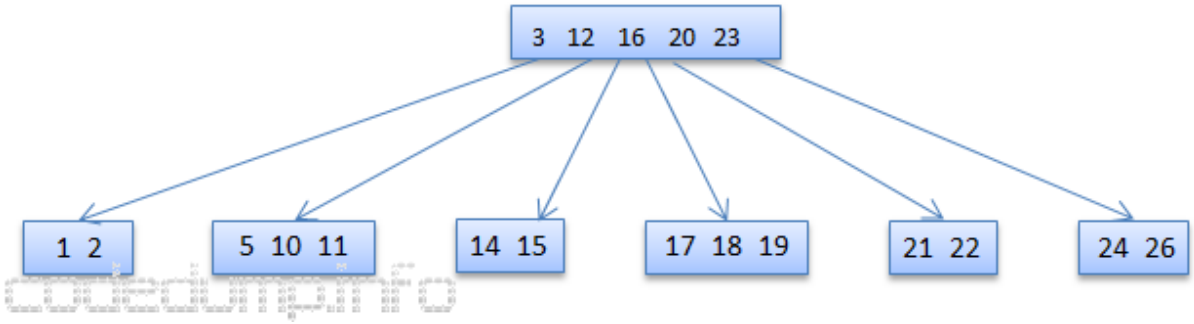
在删除数据4之前如下图所示：



包含数据4的节点 `[4,5,10,11]` ，其父节点 `[3,12]` 只有2个数据，因此需要做重平衡操作。但是其兄弟节点 `[20,23]` 也是只有2个数据，因此只能将两者进行合并：



由于合并数据，导致了树的高度缩减，因此需要修改根节点，从`[16]`这个节点修改为`[3,12,16,2023]`：



情况2和情况3有一些类似，以至于初学的时候会发生混淆，这里将我学习总结的区别列举一下：

- 情况2操作针对的是**直接**包含数据的节点，而情况3操作针对的是它们的祖先节点。
- 情况2的重平衡操作对象，都是对其相邻左右子树节点进行的。
- 情况3的重平衡操作对象，都是对其相邻左右节点进行的，而并不是和其子树节点进行合并和借用。

B树删除伪代码

在《算法导论》中，B树的删除代码并没有直接给出，而是作为章后的问题留给了学习者，这里贴出来，可以对比上面几种情况的分析，或者结合最后给出的可运行的Python代码进行阅读加深理解，出自Deletion in B-Tree：

```
1. B-Tree-Delete-Key(x, k)
2.      if not leaf[x] then
3.              y ← Preceding-Child(x)
4.              z ← Successor-Child(x)
5.              if n[y] > t − 1 then
6.                      k' ← Find-Predecessor-Key(k, x)
7.                      Move-Key(k', y, x)
8.                      Move-Key(k, x, z)
9.                      B-Tree-Delete-Key(k, z)
10.             else if n[z] > t − 1 then
11.                     k' ← Find-Successor-Key(k, x)
12.                     Move-Key(k', z, x)
13.                     Move-Key(k, x, y)
14.                     B-Tree-Delete-Key(k, y)
15.             else
16.                     Move-Key(k, x, y)
17.                     Merge-Nodes(y, z)
18.                     B-Tree-Delete-Key(k, y)
19.             else (leaf node)
20.              y ← Preceding-Child(x)
21.              z ← Successor-Child(x)
22.              w ← root(x)
23.              v ← RootKey(x)
24.                     if n[x] > t − 1 then Remove-Key(k, x)
25.                     else if n[y] > t − 1 then
26.                             k' ← Find-Predecessor-Key(w, v)
27.                             Move-Key(k', y,w)
28.                             k' ← Find-Successor-Key(w, v)
29.                             Move-Key(k',w, x)
30.                             B-Tree-Delete-Key(k, x)
31.                     else if n[w] > t − 1 then
32.                             k' ← Find-Successor-Key(w, v)
```

```
33.                             Move-Key(k', z,w)
34.                             k' ← Find-Predecessor-Key(w, v)
35.                             Move-Key(k',w, x)
36.                             B-Tree-Delete-Key(k, x)
37.                     else
38.                             s ← Find-Sibling(w)
39.                             w' ← root(w)
40.                                     if n[w'] = t − 1 then
41.                                             Merge-Nodes(w',w)
42.                                             Merge-Nodes(w, s)
43.                                             B-Tree-Delete-
Key(k, x)
44.                                     else
45.                                             Move-Key(v,w, x)
46.                                             B-Tree-Delete-
Key(k, x)
```

中序遍历

根据B树的特点，对一颗B树进行中序遍历时，即：

```
in-order-visit-tree(node):

  # 中序遍历左子树

  in-order-visit-
tree(node.left)

  # 输出节点数据

  output(node)

  # 中序遍历右子树

  in-order-visit-
tree(node.right)
```

其输出应该是一个递增的序列，我的Python演示代码就是用该特征来对操作之后的B树进行测试。

Python演示代码

我将B树的算法，使用Python写了一个简单的实现，附带测试用例，如果不满足于上述伪代码，想手写B树算法的，可以参考：btree.py

参考资料

Author codedump

LastMod 2020-06-09

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）
- **2020-06-15:** B树、B+树索引算法原理（下）
- **2019-02-15:** Leveldb代码阅读笔记

邮件订阅

微信公众号