WIKIPEDIA

# B树

维基百科，自由的百科全书

在计算机科学中，**B**树（英语：B-tree）是一种自平衡的树，能够保持数据有序。这种数据结构能够让查找数据、顺序访问、插入数据及删除的动作，都在对数时间内完成。B树，概括来说是一个一般化的二叉查找树（binary search tree）一个节点可以拥有2个以上的子节点。与自平衡二叉查找树不同，B树适用于读写相对大的数据块的存储系统，例如磁盘。B树减少定位记录时所经历的中间过程，从而加快存取速度。B树这种数据结构可以用来描述外部存储。这种数据结构常被应用在数据库和文件系统的实现上。

| **B**树 | | |
|---|---|---|
| 类型 | 树 | |
| 发明时间 | 1972 | |
| 发明者 | Rudolf Bayer, Edward M. McCreight | |
| 大**O**符号的时间复杂度 | | |
| 算法 | 平均 | 最差 |
| 空间 | O(*n*) | O(*n*) |
| 搜索 | O(log *n*) | O(log *n*) |
| 插入 | O(log *n*) | O(log *n*) |
| 删除 | O(log *n*) | O(log *n*) |

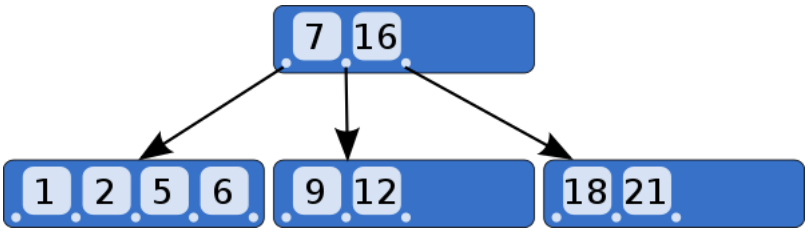
## 目录



## 概述

在B树中，内部（非叶子）节点可以拥有可变数量的子节点（数量范围预先定义好）。当数据被插入或从一个节点中移除，它的子节点数量发生变化。为了维持在预先设定的数量范围内，内部节点可能会被合并或者分离。因为子节点数量有一定的允许范围，所以B树不需要像其他自平衡查找树那样频繁地重新保持平衡，但是由于节点没有被完全填充，可能浪费了一些空间。子节点数量的上界和下界依特定的实现而设置。例如，在一个2-3 B树（通常简称2-3树），每一个内部节点只能有2或3个子节点。

B树中每一个内部节点会包含一定数量的键，键将节点的子树分开。例如，如果一个内部节点有3个子节点（子树），那么它就必须有两个键： $a_1$ 和 $a_2$ 。左边子树的所有值都必须小于 $a_1$ ，中间子树的所有值都必须在 $a_1$ 和 $a_2$ 之间，右边子树的所有值都必须大于 $a_2$ 。



B树 （Bayer & McCreight 1972） (order为5) （Knuth 1998）.

通常，键的数量被选定在 $d$ 和 $2d$ 之间。其中 $d$ 是键的最小数量，$d+1$ 是树最小的 度 或 分支因子 。在实际中，键值占用了节点中大部分的空间。因数2将保证节点可以被拆分或组合。如果一个内部节点有 $2d$ 个键，那么要添加一个键值给此节点，只需要拆分这 $2d+1$ 个键为2个 拥有 $d$ 个键的节点，并把中间值节点移动到父节点。每一个拆分的节点需要拥有足够数目的键。相似地，如果一个内部节点和他的邻居两者都有 $d$ 个键，那么将通过它与邻居的合并来删除一个键。删除此键将导致此节点拥有 $d-1$ 个键;与邻居的合并则加上 $d$ 个键，再加上从邻居节点的父节点移来的一个键值。结果为完全填充的 $2d$ 个键。

一个节点的分支（或子节点）的数量会比存储在节点内部键值的数量大1。在 2-3 B树中，内部节点将会存储1个键值（带有2个子节点）或2个键值（带有3个子节点）。一个B树有时会被描述为 $(d+1)-(2d+1)$ 或简单地使用最高分支 $(2d+1)$ 。

一个Ｂ树通过约束所有叶子节点在相同深度来保持平衡。深度在元素添加至树的过程中缓慢增长，而整体深度极少地增长，并导致所有叶子节点与根节点距离加1。

在存取节点数据所耗时间远超过处理节点数据所耗时间的情况下，Ｂ树在可选的实现中拥有很多优势，因为存取节点的开销被分摊到里层节点的多次操作上。这通常出现在当节点存储在二级存储器如硬盘存储器上。通过最大化内部里层节点的子节点的数量，树的高度减小，存取节点的开销被缩减。另外，重新平衡树的动作也更少出现。子节点的最大数量取决于，每个子节点必需存储的信息量，和完整磁盘块的大小或者二次存储器中类似的容量。虽然 2-3 树更易于解释，实际运用中，Ｂ树使用二级存储器，需要大量数目的子节点来提升效率。

## 变体

术语Ｂ树可以指一个特定的方案，也可以指大体上一类方案。狭义上，一个Ｂ树在它内部节点中存储键值，但不需在叶子节点上存储这些键值的记录。大体上的一类包含一些变体，如B+树和Ｂ*树。

在Ｂ+树，这些键值的拷贝被存储在内部节点；键值和记录存储在叶子节点；另外，一个叶子节点可以包含一个指针，指向另一个叶子节点以加速顺序存取。

B*树分支出更多的内部邻居节点以保持内部节点更密集地填充。此变体要求非根节点至少2/3填充，而不是1/2。为了维持这样的结构，当一个节点填满之后将不会再立即分割节点，而是将它的键值与下一个节点共享。当两个节点都填满之后，分割成3个节点。

计数B树存储，每一树都带有一个指针和其指向子树的节点数目。这就允许了以键值为序快速查找第N笔记录，或是统计2笔记录之间的记录数目，还有其他很多相关的操作。

## 名字取义

Rudolf Bayer 和 Ed McCreight 于1972年，在Boeing Research Labs 工作时发明了B 树，但是他们没有解释B 代表什么意义（如果有的话）。Douglas Comer 解释说: 两位作者从来都没解释过B树的原始意义。正如我们所见，“balanced”，“broad” 或 “bushy” 可能适合。其他人建议字母“B”代表 Boeing。源自于他的赞助，不过，看起来把B树当作“Bayer”树更合适些

Donald Knuth 在他1980年5月发表的题为“CS144C classroom lecture about disk storage and B-trees”的论文中推测了B树的名字取义，提出“B”可能意味Boeing 或者Bayer 的名字。

# 数据库的问题

## 已排序文件的查找时间

通常，排序和查找算法会被通过大O符号，刻画为比较级别的数值。对一个有N笔记录的已排序表进行二叉查找，打个比方说，可以在O（$\log_2 N$）比较级完成。如果表有1,000,000笔记录,那么定位其中一笔记录，将在20个比较级内完成。$\log_2(1{,}000{,}000) = 19.931...$

大数据库一直以来被存储在磁盘。从磁盘上读取一笔记录，与之后的比较键值操作相比，在花费的运行时间上前者处于支配地位。从磁盘读取记录的时间涉及到一个 寻道时间 和 旋转延迟。寻道时间可能是从0到20或者更多毫秒，旋转延迟平均下来约是旋转周期的一半。对于一个7,200转每分钟的磁盘，旋转周期大约是8.33毫秒。像希捷ST3500320NS这样的磁盘,磁道至磁道的寻道时间为 0.8毫秒，平均读取寻道时间为8.5毫秒。为了简化，假设从磁盘读取花费10毫秒。

乐观来说，如此，在一百万中定位一笔记录将会话花费20次磁盘读取乘上10毫秒每次读取时间，总共是0.2秒。

时间花费没有那么糟糕的原因是，独立的记录被成组地记录在磁盘块上。一个磁盘块可能为16 千字节。如果每笔记录大小为160 字节，那么一个块可以存储100 笔记录。上面假设的磁盘读取时间确切地说是读取一个完整块的时间。一旦磁头到达位置，一个或者更多的磁盘块可以以较小的延迟来完成读取。对于100笔记录每块，最后差不多6个比较级是不需要任何磁盘读取的——都在上次读取操作中完成了。

为进一步加速查找，开始的13或14个比较级（每个需要一次磁盘访问）必须要提速。

## 提升查找的索引

较大程度上的提升是通过索引来做到的。在上面的例子中，初始磁盘读取从2个因素限制了查找范围。这基本上可以通过创建一个辅助索引来改善，这个索引包含每块磁盘块上的首笔记录（有时称为稀疏索引）。这个辅助索引可能只有原始数据库的1%大小，但是它可以更快速地被检索。在辅助索引中查找入口可以告诉我们在主数据库中要读取哪一块;查找辅助索引之后，我们只需要读取主数据库中的特定的某一个磁盘分块——通过一次磁盘读取开销。索引可以提供10,000入口，所以，这样最多需要14个比较级。就像主数据库，辅助索引中最后6个左右的比较级可能在相同的磁盘分块上。索引可以在大约8次磁盘读取中完成查找，目标记录会在9次磁盘读取后获得。

创建辅助索引的窍门是可以重复地给辅助索引创建辅助索引。那样可以实现一个只拥有100 入口，能填满一整个磁盘块的辅助-辅助索引。

要找到想要的记录，我们只需要读取3次磁盘分块，而不是14次。读取和查找辅助-辅助索引中第一个（而且是唯一的）块，标记了相应的辅助索引中的分块。读取和查找辅助索引的分块，标记了主数据库中相应的分块。我们只需要30毫秒，而不是150毫秒就能获取记录。

辅助的索引，使得查找问题从约为log2N 磁盘读取开销的二分查找，变成logbN 磁盘读取开销的查找，其中b为分块因素（每分块的入口数目：b = 100 入口每分块;logb1,000,000 = 3 次读取）。

在实际中，如果主数据库被频繁查找，辅助-辅助索引和大部分的辅助索引可能会存储在磁盘缓存中，所以它们不会产生磁盘读取。

## 插入和删除带来的麻烦

如果数据库不会改变，那么编制索引就很简单，而且索引永远不需要改变。如果他们会改变，那么管理数据库及其索引就变得非常麻烦。

从数据库中删除记录不会引起太大问题。索引可以保持不变，记录只需要标记为已删除。数据库仍然保持有序状态。如果会有很多删除，之后查找和存储就不再那么高效了。

在一个有序文件中进行插入将是个灾难，因为需要给插入的记录制造空间。在文件中第一笔记录后插入记录需要把所有记录向后偏移一个位置。如此的操作在实际中实在太过昂贵。

一种做法是预留一些空间给插入操作。磁盘块有一些空闲空间允许后来的插入，而不是高密度地填充。这些记录可以被标记为像是已删除的记录。

现在，只要块中存在空间，插入和删除都可以很快速。如果一个插入操作在一个块上找不到合适的空间，就在临近的块中寻找，且要调整辅助索引。期望是临近存在足够的空间，以免重新调整大量的块。作为可选方案，可以使用一些非排序的块。

### B树运用的理念

B树使用了以上所有的想法。特别是：

- 保持键值有序，以顺序遍历
- 使用层次化的索引来最小化磁盘读取
- 使用不完全填充的块来加速插入和删除
- 通过优雅的遍历算法来保持索引平衡

另外，B树通过保证内部节点至少半满来最小化空间浪费。一棵B树可以处理任意数目的插入和删除。

### B树的弊端

- 除非完全重建数据库，否则无法改变键值的最大长度。这使得许多数据库系统将人名截断到70字符之内。

（其他关联数组的实现，例如三元搜索树或者开散列哈希表，可以动态适应任意长度的键值）。

## 术语和定义

### 术语

文献中B树的术语并不统一 （Folk & Zoellick 1992，p.362）。

Bayer & McCreight （1972），Comer （1979）等人将B树的 *阶* 定义为非根节点拥有键的最小数量。Folk & Zoellick（1992） 指出这一术语是模糊不清的。一个 3 阶B树键的最大数量可能为 6 或 7。 Knuth（1998, p. 483） 通过将 *阶* 定义为最大数量的子节点（比最大数量的键大1）来避免这一问题。

术语 *叶子* 的定义也不一致。 Bayer & McCreight（1972） 认为叶子层是最下面一层的键，但是 Knuth 认为叶子层是最下面一层键之下的一层 （Folk & Zoellick 1992，p.363）。可能的实现有许多。在一些设计中，叶子可能保存了完整的数据记录；在另一些设计中，叶子可能只保存了指向数据记录的指针。

为了简化，许多作者假定一个节点能够容纳固定数量的键。基础的假设是键和节点的大小都是固定的。事实上，可变长度的键可能会被使用 （Folk & Zoellick 1992，p.379）。

### 定义

根据 Knuth 的定义，一个 $m$ 阶的B树是一个有以下属性的树：

1. 每一个节点最多有 $m$ 个子节点
2. 每一个非叶子节点（除根节点）最少有 $\lceil m/2 \rceil$ 个子节点
3. 如果根节点不是叶子节点，那么它至少有两个子节点
4. 有 $k$ 个子节点的非叶子节点拥有 $k - 1$ 个键

5. 所有的叶子节点都在同一层

每一个内部节点的键将节点的子树分开。例如，如果一个内部节点有3个子节点（子树），那么它就必须有两个键： $a_1$ 和 $a_2$ 。左边子树的所有值都必须小于 $a_1$ ，中间子树的所有值都必须在 $a_1$ 和 $a_2$ 之间，右边子树的所有值都必须大于 $a_2$ 。

内部节点

内部节点是除叶子节点和根节点之外的所有节点。它们通常被表示为一组有序的元素和指向子节点的指针。每一个内部节点拥有最多 *U* 个，最少 *L* 个子节点。元素的数量总是比子节点指针的数量少一（元素的数量在 *L*-1 和 *U*-1 之间）。*U* 必须等于 2*L* 或者 2*L*-1;因此，每一个内部节点都至少是半满的。*U* 和 *L* 之间的关系意味着两个半满的节点可以合并成一个合法的节点，一个全满的节点可以被分裂成两个合法的节点（如果父节点有空间容纳移来的一个元素）。这些特性使得在B树中删除或插入新的值时可以调整树来保持B树的性质。

根节点

根节点拥有的子节点数量的上限和内部节点相同，但是没有下限。例如，当整个树中的元素数量小于 *L*-1 时，根节点是唯一的节点并且没有任何子节点。

叶子节点

叶子节点对元素的数量有相同的限制，但是没有子节点，也没有指向子节点的指针。

一个深度为$n$+1 的B树可以容纳的元素数量大约是深度为 $n$ 的B树的 $U$ 倍，但是搜索、插入和删除操作的开销也会增加。和其他的平衡树一样，这一开销增加的速度远远慢于元素数量的增加。

一些平衡树只在叶子节点中存储值，而且叶子节点和内部节点使用不同的结构。B树在每一个节点中都存储值，所有的节点有着相同的结构。然而，因为叶子节点没有子节点，所以可以通过使用专门的结构来提高B树的性能。

# 算法

## 搜索

B树的搜索和二叉搜索树类似。从根节点开始，从上到下递归的遍历树。在每一层上，搜索的范围被减小到包含了搜索值的子树中。子树值的范围被它的父节点的键确定。

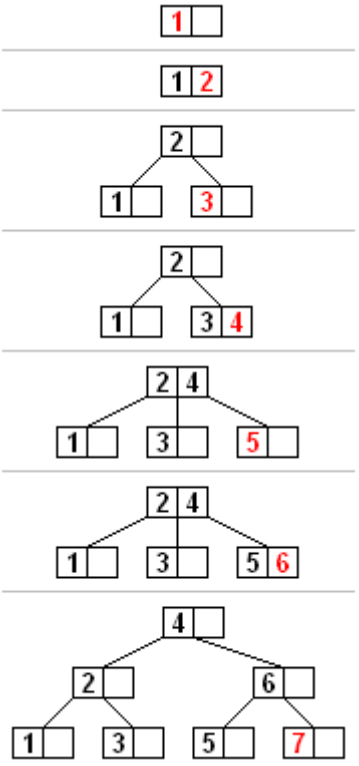
## 插入

所有的插入都从根节点开始。要插入一个新的元素，首先搜索这棵树找到新元素应该被添加到的叶子节点。将新元素插入到这一节点中的步骤如下：

1. 如果节点拥有的元素数量小于最大值，那么有空间容纳新的元素。将新元素插入到这一节点，且保持节点中元素有序。
2. 否则的话这一节点已经满了，将它平均地分裂成两个节点：
    1. 从叶子节点的元素和新的元素中选择出中位数
    2. 小于这一中位数的元素放入左边节点，大于这一中位数的元素放入右边节点，中位数作为分隔值。
    3. 分隔值被插入到父节点中，这可能会造成父节点分裂，分裂父节点时可能又会使它的父节点分裂，以此类推。如果没有父节点（这一节点是根节点），就创建一个新的根节点（增加了树的高度）。

如果分裂一直上升到根节点，那么一个新的根节点会被创建，它有一个分隔值和两个子节点。这就是根节点并不像内部节点一样有最少子节点数量限制的原因。每个节点中元素的最大数量是 $U$-1。当一个节点分裂时，一个元素被移动到它的父节点，但是一个新的元素增加了进来。所以最大的元素数量 $U$-1 必须能够被分成两个合法的节点。如果 $U$-1 是奇

数，那么 $U=2L$ ，总共有 $2L$-1 个元素，一个新的节点有 $L$-1 个元素，另外一个有 $L$ 个元素，都是合法的节点。如果 $U$-1 是偶数，那么 $U=2L$-1,总共有 $2L$-2 个元素。一半是 $L$-1，正好是节点允许的最小元素数量。



B树插入的例子。 节点最多有3个孩子 (Knuth 阶为 3).

## 删除

有两种常用的删除策略

1. 定位并删除元素，然后调整树使它满足约束条件； 或者
2. 从上到下处理这棵树，在进入一个节点之前，调整树使得之后一旦遇到了要删除的键，它可以被直接删除而不需要再进行调整

以下的算法使用了前一种策略。

删除一个元素时有以下两种特殊情况

1. 这个元素用于分隔一个内部节点的子节点
2. 删除元素会导致它所在的节点的元素或子节点数量小于最低值

下面分别是这些情况的处理过程

删除叶子节点中的元素

1. 搜索要删除的元素
2. 如果它在叶子节点，将它从中删除
3. 如果发生了下溢出，按照后边 “删除后重新平衡”部分的描述重新调整树

删除内部节点中的元素

内部节点中的每一个元素都作为分隔两颗子树的分隔值，因此我们需要重新划分。值得注意的是左子树中最大的元素仍然小于分隔值。同样的，右子树中最小的元素仍然大于分隔值。这两个元素都在叶子节点中，并且任何一个都可以作为两颗子树的新分隔值。算法的描述如下：

1. 选择一个新的分隔符（左子树中最大的元素或右子树中最小的元素），将它从叶子节点中移除，替换掉被删除的元素作为新的分隔值。
2. 前一步删除了一个叶子节点中的元素。如果这个叶子节点拥有的元素数量小于最低要求，那么从这一叶子节点开始重新进行平衡。

删除后的重新平衡

重新平衡从叶子节点开始向根节点进行，直到树重新平衡。如果删除节点中的一个元素使该节点的元素数量低于最小值，那么一些元素必须被重新分配。通常，移动一个元素数量大于最小值的兄弟节点中的元素。如果兄弟节点都没有多余的元素，那么缺少元素的节点就必须要和他的兄弟节点 合并。合并可能导致父节点失去了分隔值，所以父节点可能缺少元素并需要重新平衡。合并和重新平衡可能一直进行到根节点，根节点变成惟一缺少元素的节点。重新平衡树的算法如下：

- 如果缺少元素节点的右兄弟存在且拥有多余的元素，那么向左旋转
    1. 将父节点的分隔值复制到缺少元素节点的最后（分隔值被移下来；缺少元素的节点现在有最小数量的元素）
    2. 将父节点的分隔值替换为右兄弟的第一个元素（右兄弟失去了一个节点但仍然拥有最小数量的元素）
    3. 树又重新平衡
- 否则，如果缺少元素节点的左兄弟存在且拥有多余的元素，那么向右旋转

1. 将父节点的分隔值复制到缺少元素节点的第一个节点（分隔值被移下来；缺少元素的节点现在有最小数量的元素）
2. 将父节点的分隔值替换为左兄弟的最后一个元素（左兄弟失去了一个节点但仍然拥有最小数量的元素）
3. 树又重新平衡

- 否则，如果它的两个直接兄弟节点都只有最小数量的元素，那么将它与一个直接兄弟节点以及父节点中它们的分隔值合并

1. 将分隔值复制到左边的节点（左边的节点可以是缺少元素的节点或者拥有最小数量元素的兄弟节点）
2. 将右边节点中所有的元素移动到左边节点（左边节点现在拥有最大数量的元素，右边节点为空）
3. 将父节点中的分隔值和空的右子树移除（父节点失去了一个元素）
    - 如果父节点是根节点并且没有元素了，那么释放它并且让合并之后的节点成为新的根节点（树的深度减小）
    - 否则，如果父节点的元素数量小于最小值，重新平衡父节点

# 相关条目

B+树