**1、平衡二叉树**

* **概念**

平衡二叉树是基于二分法的策略提高数据的查找速度的二叉树的数据结构；

* **特点：**

平衡二叉树是采用二分法思维把数据按规则组装成一个树形结构的数据，用这个树形结构的数据减少无关数据的检索，大大的提升了数据检索的速度；平衡二叉树的数据结构组装过程有以下规则：

（1）非叶子节点只能允许最多两个子节点存在。

（2）每一个非叶子节点数据分布规则为左边的子节点小当前节点的值，右边的子节点大于当前节点的值(这里值是基于自己的算法规则而定的，比如hash值)；

平衡树的层级结构：因为平衡二叉树查询性能和树的层级（h高度）成反比，h值越小查询越快、为了保证树的结构左右两端数据大致平衡降低二叉树的查询难度一般会采用一种算法机制实现节点数据结构的平衡，实现了这种算法的有比如[Treap](https://link.zhihu.com/?target=http%3A//baike.baidu.com/item/Treap" \t "_blank)、红黑树，使用平衡二叉树能保证数据的左右两边的节点层级相差不会大于1.，通过这样避免树形结构由于删除增加变成线性链表影响查询效率，保证数据平衡的情况下查找数据的速度近于二分法查找；

总结平衡二叉树特点：

（1）非叶子节点最多拥有两个子节点；

（2）非叶子节值大于左边子节点、小于右边子节点；

（3）树的左右两边的层级数相差不会大于1;

（4）没有值相等重复的节点;

**2、B树(B-tree)**

注意:之前有看到有很多文章把B树和B-tree理解成了两种不同类别的树，其实这两个是同一种树;

* **概念：**

B树和平衡二叉树稍有不同的是B树属于多叉树又名平衡多路查找树（查找路径不只两个），数据库索引技术里大量使用者B树和B+树的数据结构，让我们来看看他有什么特点;

* **规则：**

（1）排序方式：所有节点关键字是按递增次序排列，并遵循左小右大原则；

（2）子节点数：非叶节点的子节点数>1，且<=M ，且M>=2，空树除外（注：M阶代表一个树节点最多有多少个查找路径，M=M路,当M=2则是2叉树,M=3则是3叉）；

（3）关键字数：枝节点的关键字数量大于等于ceil(m/2)-1个且小于等于M-1个（注：ceil()是个朝正无穷方向取整的函数 如ceil(1.1)结果为2);

（4）所有叶子节点均在同一层、叶子节点除了包含了关键字和关键字记录的指针外也有指向其子节点的指针只不过其指针地址都为null对应下图最后一层节点的空格子;

最后我们用一个图和一个实际的例子来理解B树（这里为了理解方便我就直接用实际字母的大小来排列C>B>A）

* **B树的查询流程：**

如上图我要从上图中找到E字母，查找流程如下

（1）获取根节点的关键字进行比较，当前根节点关键字为M，E<M（26个字母顺序），所以往找到指向左边的子节点（二分法规则，左小右大，左边放小于当前节点值的子节点、右边放大于当前节点值的子节点）；

（2）拿到关键字D和G，D<E<G 所以直接找到D和G中间的节点；

（3）拿到E和F，因为E=E 所以直接返回关键字和指针信息（如果树结构里面没有包含所要查找的节点则返回null）；

* **B树的插入节点流程**

定义一个5阶树（平衡5路查找树;），现在我们要把3、8、31、11、23、29、50、28 这些数字构建出一个5阶树出来;

遵循规则：

（1）节点拆分规则：当前是要组成一个5路查找树，那么此时m=5,关键字数必须<=5-1（这里关键字数>4就要进行节点拆分）；

（2）排序规则：满足节点本身比左边节点大，比右边节点小的排序规则;

先插入 3、8、31、11

再插入23、29

再插入50、28

* **B树节点的删除**

**规则：**

（1）节点合并规则：当前是要组成一个5路查找树，那么此时m=5,关键字数必须大于等于ceil（5/2）（这里关键字数<2就要进行节点合并）；

（2）满足节点本身比左边节点大，比右边节点小的排序规则;

（3）关键字数小于二时先从子节点取，子节点没有符合条件时就向向父节点取，取中间值往父节点放；

**特点：**

B树相对于平衡二叉树的不同是，每个节点包含的关键字增多了，特别是在B树应用到数据库中的时候，数据库充分利用了磁盘块的原理（磁盘数据存储是采用块的形式存储的，每个块的大小为4K，每次IO进行数据读取时，同一个磁盘块的数据可以一次性读取出来）把节点大小限制和充分使用在磁盘快大小范围；把树的节点关键字增多后树的层级比原来的二叉树少了，减少数据查找的次数和复杂度;

**3、B+树**

* **概念**

B+树是B树的一个升级版，相对于B树来说B+树更充分的利用了节点的空间，让查询速度更加稳定，其速度完全接近于二分法查找。为什么说B+树查找的效率要比B树更高、更稳定；我们先看看两者的区别

* **规则**

（1）B+跟B树不同B+树的**非叶子**节点不保存关键字记录的指针，只进行数据索引，这样使得B+树每个**非叶子**节点所能保存的关键字大大增加；

（2）B+树**叶子**节点保存了父节点的所有关键字记录的指针，所有数据地址必须要到叶子节点才能获取到。所以每次数据查询的次数都一样；

（3）B+树叶子节点的关键字从小到大有序排列，左边结尾数据都会保存右边节点开始数据的指针。

（4）非叶子节点的子节点数=关键字数（来源百度百科）（根据各种资料 这里有两种算法的实现方式，另一种为非叶节点的关键字数=子节点数-1（来源维基百科)，虽然他们数据排列结构不一样，但其原理还是一样的Mysql 的B+树是用第一种方式实现）;

**（百度百科算法结构示意图）**

**（维基百科算法结构示意图）**

* **特点**

1、B+**树的层级更少**：相较于B树B+每个**非叶子**节点存储的关键字数更多，树的层级更少所以查询数据更快；

2、B+**树查询速度更稳定**：B+所有关键字数据地址都存在**叶子**节点上，所以每次查找的次数都相同所以查询速度要比B树更稳定;

3、B+**树天然具备排序功能：**B+树所有的**叶子**节点数据构成了一个有序链表，在查询大小区间的数据时候更方便，数据紧密性很高，缓存的命中率也会比B树高。

4、B+**树全节点遍历更快：**B+树遍历整棵树只需要遍历所有的**叶子**节点即可，，而不需要像B树一样需要对每一层进行遍历，这有利于数据库做全表扫描。

**B树**相对于**B+树**的优点是，如果经常访问的数据离根节点很近，而**B树**的**非叶子**节点本身存有关键字其数据的地址，所以这种数据检索的时候会要比**B+树**快。

**4、B\*树**

* **规则**

B\*树是B+树的变种，相对于B+树他们的不同之处如下：

（1）首先是关键字个数限制问题，B+树初始化的关键字初始化个数是cei(m/2)，b\*树的初始化个数为（cei(2/3\*m)）

（2）B+树节点满时就会分裂，而B\*树节点满时会检查兄弟节点是否满（因为每个节点都有指向兄弟的指针），如果兄弟节点未满则向兄弟节点转移关键字，如果兄弟节点已满，则从当前节点和兄弟节点各拿出1/3的数据创建一个新的节点出来；

* **特点**

在B+树的基础上因其初始化的容量变大，使得节点空间使用率更高，而又存有兄弟节点的指针，可以向兄弟节点转移关键字的特性使得B\*树额分解次数变得更少；

**5、 总结**

**1、相同思想和策略**

从平衡二叉树、B树、B+树、B\*树总体来看它们的贯彻的思想是相同的，都是采用二分法和数据平衡策略来提升查找数据的速度；

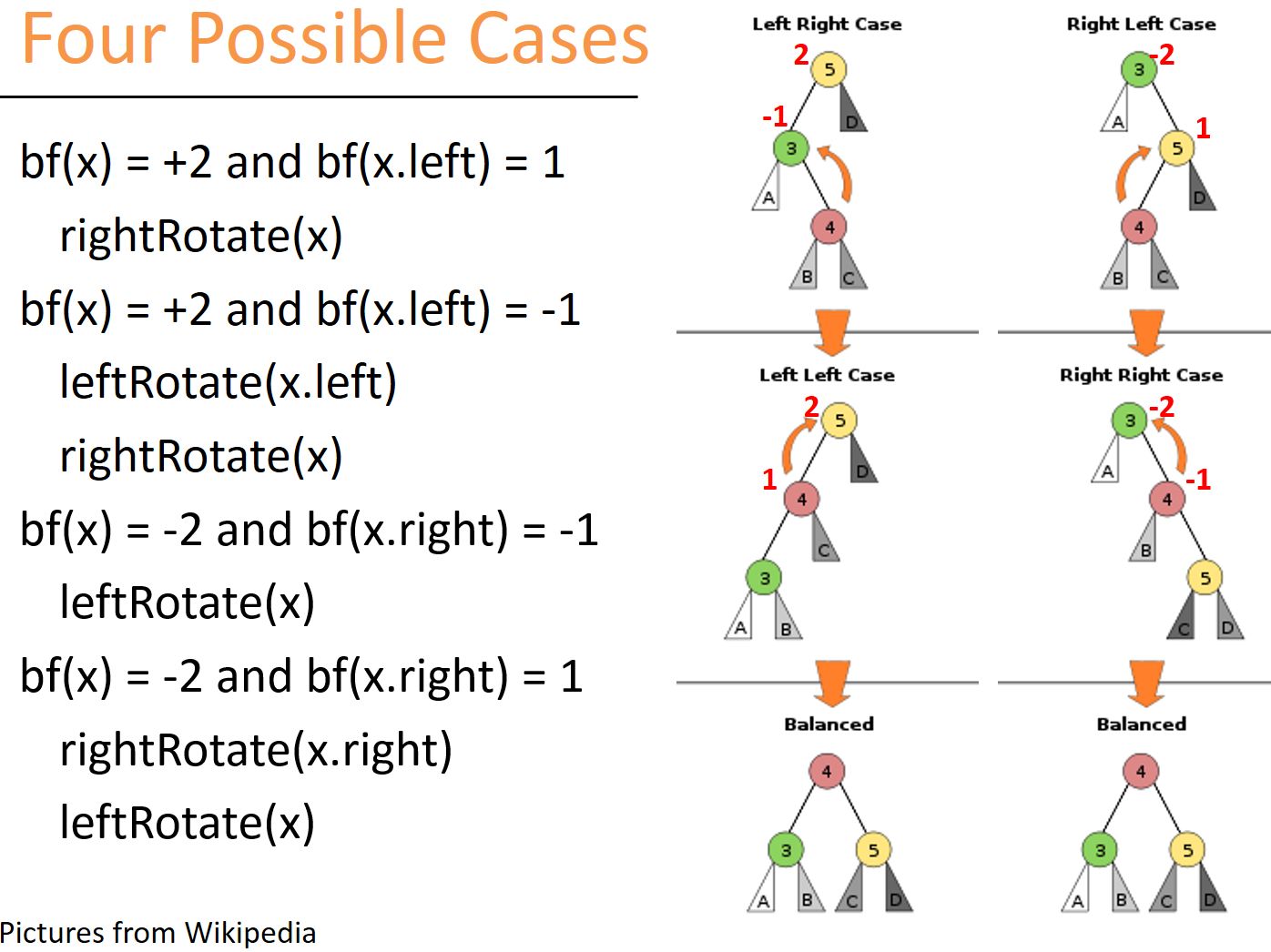
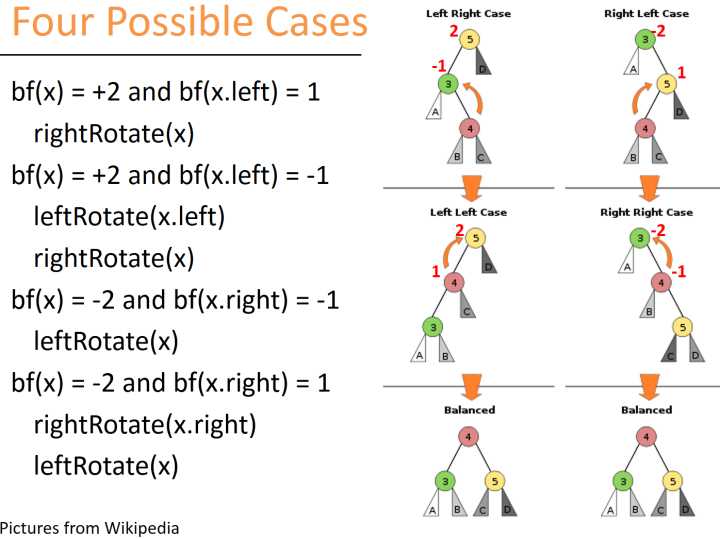
**2、不同的方式的磁盘空间利用**

不同点是他们一个一个在演变的过程中通过IO从磁盘读取数据的原理进行一步步的演变，每一次演变都是为了让节点的空间更合理的运用起来，从而使树的层级减少达到快速查找数据的目的；

应用场景：

作者：北南  
链接：https://www.zhihu.com/question/30527705/answer/260005525  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

AVL树是平衡二叉搜索树的鼻祖，它的平衡度也最好，左右高度差可以保证在「-1，0，1」，基于它的平衡性，它的查询时间复杂度可以保证是O（log n）。但每个节点要额外保存一个平衡值，或者说是高度差。这种树是二叉树的经典应用，现在最主要是出现在教科书中。AVL的平衡算法比较麻烦，需要左右两种rotate交替使用，需要分四种情况，是数据结构课的最理想课后作业之一。**这里要重点强调，AVL插入新节点所需要的最大旋转次数是常数，再说一遍，是常数，不要为这个再来回复或者私信我了。这个也有证明，不需要一直旋转到根节点。**



红黑树一样也是平衡二叉搜索树，也是工业界最主要使用的二叉搜索平衡树。但平衡度红黑树没AVL那么好。也就是说，如果从高度差来说，红黑树是大于AVL的，其实也就代表着它的实际查询时间(最坏情况)略逊于AVL的。数学证明红黑树的最大深度是 , 其实最差情况它从根到叶子的最长路可以是最短路的两倍，但也不是很差，所以它的查询时间复杂度也是O（log n）。从实现角度来说，保存红黑状态，每个节点只需要一位二进制，也就是一个bit（有些做法，可以把这个bit塞到其他地方，就可以不占用额外空间了）。可AVL每个节点需要额外存储一个平衡值（数）（按照评论中大神韦易笑的说法，avl的平衡值可以用两个bit来存储，然后塞到指针里，惊才绝艳的搞法。当然指针少两位会有什么问题，我就不多想了，你去找他，他有个很好的avl实现）。所以（这里不能用所以，因为谁知道真实原因是啥），一般工业界把红黑树作为一种更通用的平衡搜索数来用，Java用它来实现TreeMap, C++的std::set/map/multimap等等。

以上都是二叉（binary）树，其实没有太多实质差别。他们的实现都比较复杂，如果做得不好会带来性能的隐患，所以强烈建议入行不足10年的朋友尽量使用系统中的已有默认实现，不管是avl还是红黑树，这两个有什么就用什么，千万别自己写代码，能写也别写，留个坑给别人不合适。

二叉平衡搜索树的问题在于每次插入和删除都有很大可能需要进行重新平衡，数据就要不停的搬来搬去，在内存中这问题不是特别大，可如果在磁盘中，这个开销可能就大了。

有兴趣的朋友还可以看看2-3-4 tree，它等价于红黑树，可以互相转换。它每个节点可以最多有四个孩子，重新平衡操作的次数稍稍少了一点点。以上这些树绝大多数应用都是作为内存中的数据结构。

B/B+ 树就是N叉（N-ary）平衡树了，每个节点可以有更多的孩子，新的值可以插在已有的节点里，而不需要改变树的高度，从而大量减少重新平衡和数据迁移的次数，这非常适合做数据库索引这种需要持久化在磁盘，同时需要大量查询和插入操作的应用。

以上几种树都是有序的，如果你采用合适的算法遍历整个数，可以得到一个有序的列表。这也是为什么如果有数据库索引的情况下，你order by你索引的值，就会速度特别快，因为它并没有给你真的排序，只是遍历树而已。

Trie并不是平衡树，也不一定非要有序。它主要用于前缀匹配，比如字符串，比如说ip地址，如果字符串长度是固定或者说有限的，那么Trie的深度是可控制的，你可以得到很好的搜索效果，而且插入新数据后不用平衡。不过Trie不像B-tree通用性那么强，你需要针对你自己的实际应用来设计你自己的Trie，比如说你做个字典应用，是用26个字母，还是用unicode来前缀匹配？如果是ip地址搜索，是用二进制来前缀拼配，还是八进制来匹配？

**AVL树**: 最早的平衡二叉树之一。应用相对其他数据结构比较少。windows对进程地址空间的管理用到了AVL树。

**红黑树**: 平衡二叉树，广泛用在C++的STL中。如map和set都是用红黑树实现的。

**B/B+树:**用在磁盘文件组织 数据索引和数据库索引。

**Trie树(字典树):**用在统计和排序大量字符串，如自动机。