21120491：高级数据结构与算法分析 **2023-2024 学年春夏学期**

Lecture 2: 红黑树与 B+ 树

教师：刘金飞，助教：吴一航 日期: 2024 年 3 月 5 日

2.1 红黑树

2.1.1 红黑树的引入与定义

为了实现平衡搜索树，AVL 树用一种非常简单暴力的思想，即保持严格平衡（任一结点左右子树高度差绝对值最多为 1）然后用旋转维持平衡。红黑树在某种程度上是希望放松这一假设，从另一角度——给结点染色，定义其它平衡因子——也给出了一种解决方案。我们首先来看红黑树的五条性质，或者说满足这五条性质的搜索树称为红黑树：

定义 2.1 一棵红黑树是满足下面性质的二叉搜索树：

1. 每个结点或者是红色的，或者是黑色的；
2. 根结点是黑色的；
3. 每个叶结点（*NIL*）是黑色的；
4. 如果一个结点是红色的，那么它的两个子结点都是黑色的；
5. 对每个结点，从该结点到其所有后代叶结点的简单路径上，均包含相同数目的黑色结点。

之后的讨论我们会经常使用上面的序号代表该序号对应的性质。PPT 第 2 页的右上角图示给出了红黑树结点的形式，除了常规的父子以及键值信息之外，红黑树还有一个颜色的属性，并且当左右子结点为空时，必须是黑色的 NIL 结点（它没有键值），这与一般的 NULL 不一样。下面我们来逐条解读以深入理解定义：

1. 第一条的言外之意是：只有这两种颜色，这一点或许很显然，但在之后讨论红黑树删除的时候有用处。
2. 对于第二条，之后我们会经常遇到在插入、删除调整时，根结点变为红色的情况，这时候如果直接将根结点染成黑色，对于其它四个条件是完全没有影响的！因此我们后面其实并不会太关心根结点的颜色问题，因为不是黑色直接染成黑色完全不影响其它性质。

3. 这一点或许有些迷惑，但既然定义如此必定有其作用，其作用将会在我们讨论删除的时候体现。但的确有时候 NIL 对于做题和复杂度分析并无影响，比如判断是否是红黑树，实际上把所有的 NIL 都去掉也完全不影响判断。

对于最后两条，它们的意义在于可以直接决定红黑树的树高是 $\log n$ 级别的，因此是平衡搜索树。在介绍定理之前，我们需要先引入两个定义：

定义 2.2 *1. 我们称 NIL 为外部结点，其余有键值的结点为内部结点。*

2. 从某个结点 X 出发到达一个叶子结点（NIL）的任意一条简单路径上的黑色结点个数（不含 X 本身）称为 X 的黑高，记为 $bh(X)$。根据定义第五条，这一定义是合理的，因为从 X 出发出发到达一个叶子结点的任意一条简单路径上的黑色结点个数相同。除此之外，我们定义整棵红黑树的黑高为其根结点的黑高。

然后我们可以引入如下定理：

定理 2.3 *一棵有 n 个内部结点的红黑树的高度至多为 $2\log(n+1)$。*

具体证明这里不再重复，这里想说的是，这一定理事实上无需证明就能非常直观地得到。因为如果我们仔细观察定义第五条，就会明白，黑高实际上就是红黑树最关键的平衡因子：如果我们舍去全部的红色结点，剩下的树的结点个数一定大于等于高度为 $bh(root)$ 的完全平衡二叉树的结点个数，大于的情况看 PPT 第二页的例子就可以知道，因为删掉红色结点后可能不是二叉，这就是证明的第一个步骤：证明以 X 为子结点的子树至少有 $2^{bh(x)}-1$ 个内部结点。而根据第四条，任何路径上黑色结点必定占到至少一半的数量，因为红色不能是父子关系，所以有了证明的第二步，树高最多为黑高 2 倍。

总结而言，对于红黑树，**黑高是绝对严格的平衡要求，而红色结点则是少量不平衡的因素，并且定义控制了红色结点的个数，也就控制了不平衡因素的影响**，因此红黑树还是可以保持一定程度的平衡的。有了这一结论，至少现在可以知道对于搜索操作，红黑树的时间复杂度是 $O(\log n)$ 的，至于插入和删除，还需要下面进一步讨论。

【讨论 1】证明：在一棵红黑树中，从某结点 X 到其后代叶结点的所有简单路径中，最长的一条路径的长度至多是最短一条的 2 倍。

实际上这个问题和上述定理证明基于同样的思想，因为所有简单路径的黑色结点数量都是相同的，最短的路径就是全黑路径，最长的就是黑红相间的路径（因为红色不能是父子关系）。

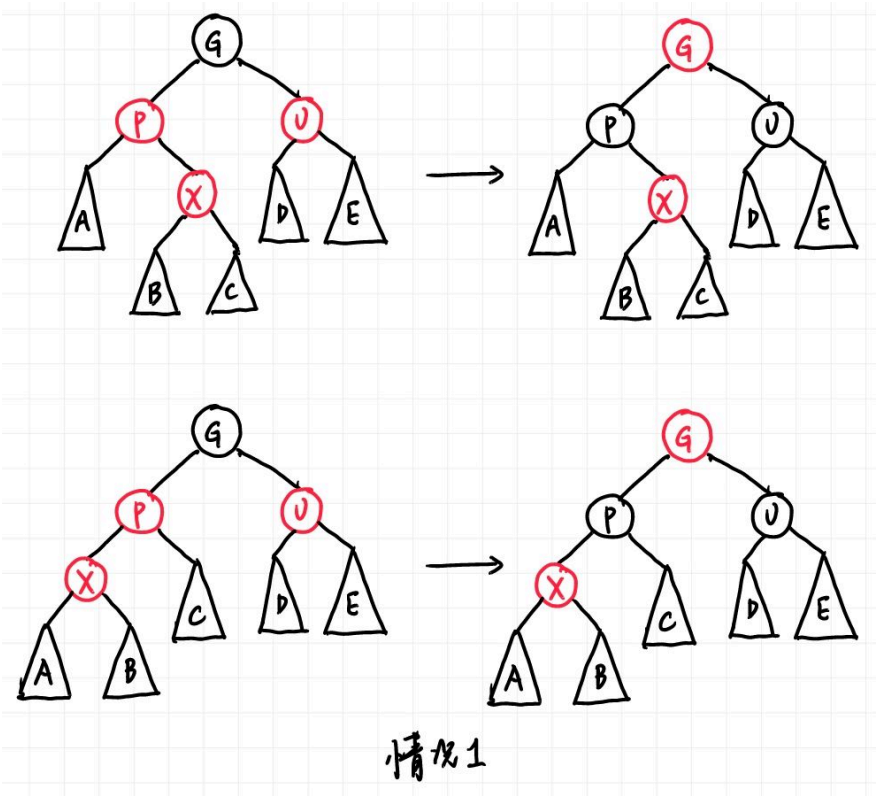
2.1.2 红黑树的插入

与 AVL 树类似，插入结点时有可能出现平衡性质被破坏，因此需要一定的调整。那么很自然地，我们面临的第一个问题就是：插入应该插入黑色结点还是红色结点？假设我们每次都插入黑色结点，那么定

义第五条一定会被破坏，因此每次插入都必须调整（除了从空树插入的那次）；但是如果插入红色结点，定义第二条（仅在插入空树时）和定义第四条仅仅是有可能被破坏，因此直观来看插入红色结点会比插入黑色结点需要的调整更少。除此之外，在学完红黑树删除后，红黑树删除黑色结点时的复杂情况也暗示了插入红色应当是更好的选择。

于是接下来我们开始介绍红黑树的插入操作，根据前面的讨论，我们决定插入的结点首先都是红色的。最简单的情况是插入在黑色结点下面，这时无需任何调整，因为没有破坏任何性质；其次是插入空树时，这时直接把结点染黑即可恢复定义第二条。因此下面我们只讨论最复杂的情况，即插入到了红色结点的子结点的位置。此时唯一被违反的就是定义第四条。因此如果要解决这一问题，我们自然的想法就是，在不影响其它平衡性质（主要是第五条）的前提下，通过一些染色和旋转使得没有父子都是红色。我们主要分为以下三种情况讨论（假设插入的结点为 X，我们先讨论 X 插入在祖父 G 左侧的情况）：

情况 1：X 的叔叔（即父亲的兄弟）是红色的，X 无论左右孩子都是该情况 需要注意的是，这里所有



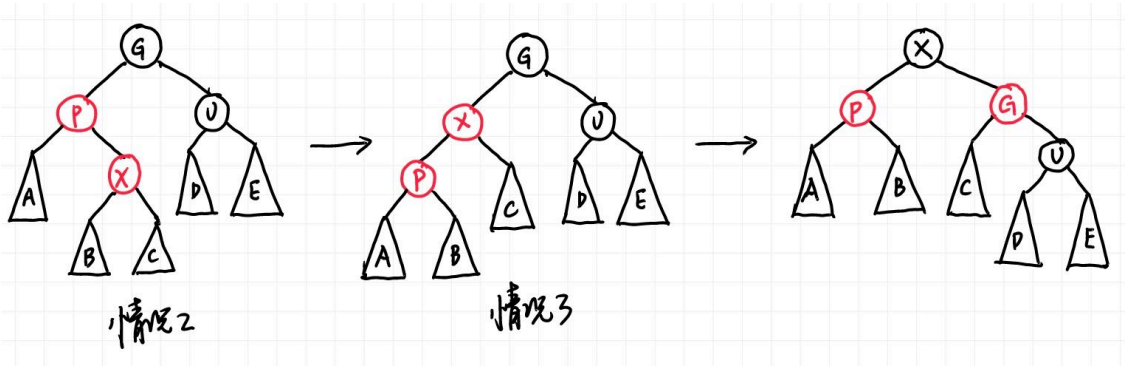
结点都带子树，一方面至少有 NIL 结点，另一方面这可以不仅仅表示刚刚插入的情况，也可以表示经过几次调整后还在被这种情况困扰，因此更具一般性。注意 G 一定是黑色，因为 P 是红色，插入前它们就在红黑树中，因此不可能违背定义第四条。

下面开始解释解决方案，事实上此时解决方案非常直白：它的父亲和叔叔都是红色，如果把红色看成一种 debuff，X 都求不能把这个 debuff 甩给他们，只能求助于祖父，于是直接把 X 的父亲和叔叔染黑，祖父染红。此时完全不影响黑高性质，但问题并没有解决，因为祖父的父亲可能还是红色！但是，问题

被往上推了，最差的情况也是一直上推到把问题交给根结点，那么根结点此时只需要直接染黑就可以解决问题。当然也可能问题上推后变为下面两种情况，接下来我们进行解释：

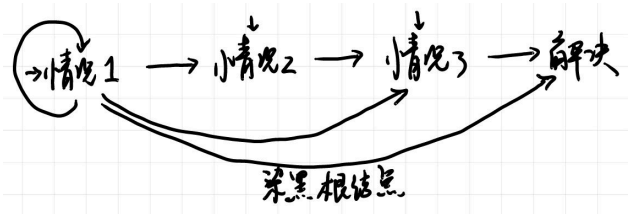
情况 2：X 的叔叔（即父亲的兄弟）是黑色的，且 X 是右孩子

情况 3：X 的叔叔（即父亲的兄弟）是黑色的，且 X 是左孩子



这里我们直接借用 AVL 树的想法，把红色视为 trouble，根据 trouble 是 LR 还是 LL 按 AVL 树的调整方式旋转，染色时，局部情况是根黑色，第二层红色。

当然我们还有 X 插入在祖父 G 右侧的情况，这是完全对称的，因此不再重复。总结而言，就是第一种情况叔叔红，直接改颜色，问题推给祖父，第二、三种情况叔叔黑且 LR、LL（对称的有 RL、RR），用 AVL 的旋转，最后局部情况是根黑色，第二层红色，大致流程如下：



即如果插入后直接落入情况三，只需要一次旋转染色即可解决，直接落入情况二，一次旋转进入情况三，再一次旋转染色即可解决，但如果落入情况一，一次调整后可能还在情况一，可能直到最后都是通过情况一加上染黑根结点解决，也可能几次调整后进入情况二或三后解决。根据这一流程我们知道，红黑树插入最多可能的旋转次数为 2（因为只有情况 2 和 3 会要旋转进入情况 2 后 1 次旋转必定进入情况 3，进入情况 3 后 1 次旋转必定解决），然后更改颜色最多是 $O(\log n)$ 次，因为进入情况 2 或 3 只需要一次染色，在情况 1 最差也是每两层染一次色，而我们已经证明红黑树的最大高度是 $O(\log n)$ 的。因此插入操作包括 $O(\log n)$ 的搜索时间，加常数的旋转，加 $O(\log n)$ 的染色，因此还是 $O(\log n)$ 的时间复杂度，总结而言我们有如下结论：

定理 2.4 一棵有 n 个内部结点的红黑树插入一个结点的时间复杂度为 $O(\log n)$。

当然，上面的描述都是原理层面的讲解，如果读者希望学习具体代码实现加深印象，可以参考《算法导论》的伪代码。

【讨论 2】考虑从空树开始连续插入 $n(n > 1)$ 个结点得到一棵红黑树（每一步插入都要保证红黑树性质），试问这棵树一定会有红色结点吗？若是，请给出清晰的证明；若不是，请举出反例。

答案是一定会，可以使用数学归纳法证明：$n = 2$ 时显然正确，根下面插入的结点一定是红色且无需调整；此后如果不需要调整，因为我们插入的是红色结点，因此红色结点只可能变多；如果需要调整，则根据三种情况的讨论，我们发现无论哪一种情况，在调整之后一定还保留着红色结点。有同学可能会质疑，情况 1 如果 G 到了根结点，则需要被染黑，但要注意的是，此时的 X 还是红色的，因此不管什么情况都是会保留红色结点。

2.1.3 红黑树的删除

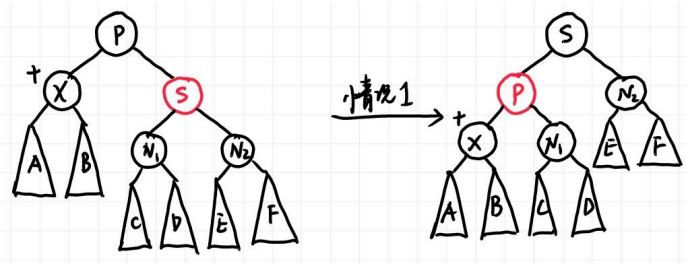
红黑树的删除是一个并不简单的操作，这其中有太多可能的情况需要不同的调整策略，我们接下来尽可能地清晰列举。在讲解红黑树删除之前，我们首先回顾一下普通平衡搜索树的删除操作，假设被删除的结点为 X（如果不熟悉了可以回顾去年的 PPT 的例子）：

1. 如果 X 没有孩子，直接删除就好，没有任何后顾之忧；
2. 如果 X 只有一个孩子，那就让孩子接替 X 的位置；
3. 如果 X 有两个孩子，那就让 X 与其左子树的最大结点（或右子树最小结点）交换，然后删除 X（这时 X 所在的位置一定只有一个子节点，因为左子树最大结点不可能有右孩子，右子树最小结点不可能有左孩子）。

事实上红黑树的删除是基于这些操作的，需要注意的是第三种情况，X **和与其交换的结点只交换键值，不交换颜色**，否则如果两者颜色不同，在交换的时候就可能破坏第五条性质，这是很难令人满意的。总而言之，**第三种情况可以通过一步交换直接转化为第一或第二种情况**，因此我们只需要关心第一和第二种情况。在第一种情况中，接替被删除结点所在位置的结点是 NIL，第二种则是被删除结点的子结点。如果被删除的结点是红色，事实上无事发生，没有任何性质被破坏；如果被删除的是黑色，如果接替上来的结点是红色的，直接染黑也不会破坏任何性质。接下来就是问题的关键，如果接替的是 NIL 或是黑色结点应该怎么办?

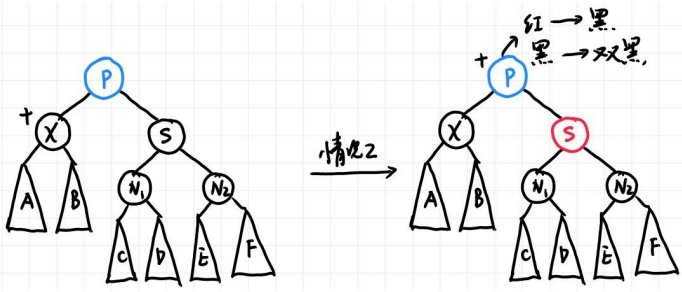
这个办法很聪明，我们直接给黑色结点或者 NIL（其实也是黑色结点）再加一重黑色，于是它的颜色变成了 “双黑”。此时第五条性质没有被破坏，但是，第一条性质被破坏了！这里出现了非红也非黑的颜色！于是我们的想法就是：把这个黑色的 debuff 扔给一个红色结点，或者一步一步往上扔给根结点，事实上根结点把双黑直接变成黑色完全不影响其它性质。我们可以将删除的情况分成以下四类（与插入相同，这里用子树表示更一般的情况，X 在此处则表示双黑结点，图中用黑色圆圈和圆旁边的加号表示双黑，蓝色表示颜色无所谓，可红可黑。注意这里 X 都是父亲的左孩子，右孩子情况对称）：

情况 1：X 的兄弟是红色的



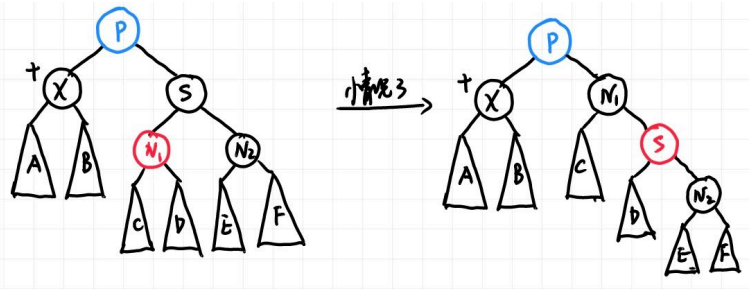
由于原先的树满足红黑树定义第四条，因此此时父结点一定是黑色。我们的想法很简单，兄弟是红色，那就希望兄弟能两肋插刀，把兄弟转上去，为了保持红黑树性质，很可惜只能把父亲染红，，自己还承受双黑 debuff。但是好处在于，这个问题转化为了接下来的情况二三四中的一种，我们来看如何解决。

情况 2：X 的兄弟是黑色的，且兄弟的两个孩子（根据距离划分为近、远侄子，用远近而不用左右是为了对称情况不混淆左右）都是黑色的



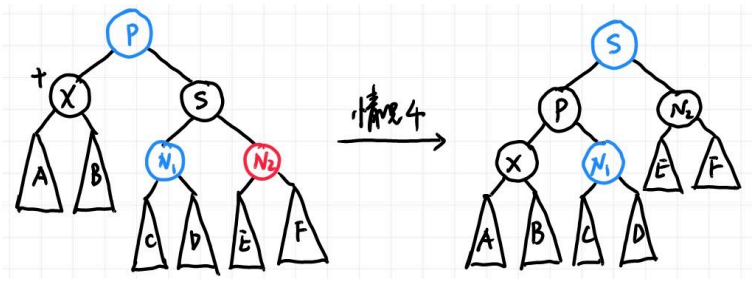
这时候没人是红色可以救了，那就想起根结点再怎么样都能救，所以把双黑往上推给父亲，为了保证红黑树性质，兄弟也要从黑变红，总而言之就是 X 这一层的黑色全部往上推。如果父亲原本是红色，那就染黑，问题解决；如果父亲原本是黑色，那父亲就变成双黑，让问题向根结点靠近。需要特别注意的是，如果是从情况 1 变成情况 2 的，父亲一定是红色，所以如果是 1 变为 2，则问题会马上解决。

情况 3：X 的兄弟是黑色的，且近侄子是红色的，远侄子是黑色的

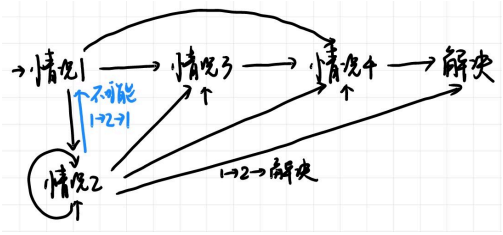


这时我们借用 AVL 树的想法，红色在父亲 P 的 RL 位置，因此做 double rotation：single rotation 后会变成情况 4 的 RR 的情况（也就意味着红色要给到 RR 的位置，这里有一个颜色的变化，用 RR 记忆很方便）。

情况 4：X 的兄弟是黑色的，且远侄子是红色的，近侄子颜色任意



此时对应 AVL 树的 RR，于是再一次 single rotation 即可把双黑的一重黑丢给红色远侄子（即 X 和 N_2 都变成黑色），但要注意为了保证红黑树性质的颜色变化，即 P 和 S 还要交换颜色，此时问题解决。



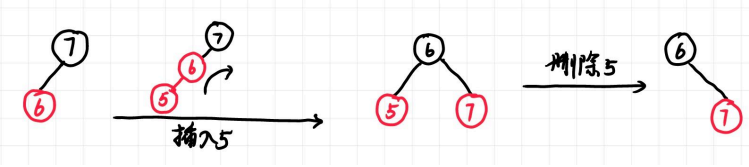
此时我们可以总结并计算出删除操作的时间复杂度。首先我们最多用 $O(\log n)$ 的时间找到删除结点，最多 1 次交换和 1 个删除的操作。接下来如果删除后没有问题则到此结束；否则根据分析，情况 1、3 和 4 在问题解决前最多进去一次，因为 4 可以直接解决，3 直接进入 4 然后解决，1 如果进入 3 和 4 也可以马上解决，进入 2 后也因为父结点是红色可以马上解决。因此关键在于情况 2 可能出现很多次，但最多也只是树高 $O(\log n)$ 次，因为每次都会上推 1 格。总而言之，因为情况 1、3 和 4 在问题解决前最多进去一次，所以最多 3 次旋转加上 $O(\log n)$ 次颜色调整可以解决问题，因此我们有如下结论：

定理 2.5 *一棵有 n 个内部结点的红黑树删除一个结点的时间复杂度为 $O(\log n)$。*

总而言之，这里的情况的确非常复杂，但相信上面的解释会给出一些直观。考试的时候删除不会太难，很多时候依靠保证搜索树性质（左小右大）和红黑性质这两点就能猜出来操作是什么，当然能准确记住是更好。如果希望学习具体代码实现，同样可以参考《算法导论》的伪代码，尽管有些难看懂，但可以了解一下如何在不扩充红黑树结点颜色的可取值范围的前提下表示双黑这种颜色。除此之外，这里的讲法与 PPT 不完全一致但实际上是一样的，实际上不同老师的讲法都可能略有区别，但各位可以过一遍 PPT 上的例子就会发现这些方法都是一样的。

【讨论 3】考虑将一个结点 X 插入红黑树 T_0，得到红黑树 T_1，然后紧接着下一步操作又立刻将 X 从 T_1 删除得到 T_2，请问 T_0 和 T_2 是否一定一样？若是，请给出清晰的证明；若不是，请举出反例。

显然不是；我们考虑下面这一非常简单的例子即可：



2.1.4 再论 AVL 树和红黑树的区别

在开头我们已经提到，AVL 树的平衡条件太严苛，因此更新树（即插入和删除）操作会更频繁，所以我们希望有一个条件更松的平衡要求但也能保证树高被控制在 $O(\log n)$ 的量级。除此之外，AVL 树和红黑树似乎都是通过旋转恢复平衡，没有很大的差别。但其实有一个很有趣的现象，又非常多的库函数在选择平衡搜索树实现功能的时候，会更常用红黑树，例如大家最熟悉的 C++ 的 `std::map`，以及 Java 8 开始的 HashMap 和 Microsoft .NET 框架的部分代码，甚至 Linux 内核中内存管理也使用了红黑树（可以参考这个 GitHub 上的 Linux 文档）。那这其中的原因可能是什么呢？

事实上这一问题应当是没有标准答案的，毕竟是当年工程师的多方面考虑综合后的选择，但我们可以通过这个问题看一看 AVL 树和红黑树的一些更细致的区别：

1. 我们都知道，AVL 树平衡条件更严格，推导 AVL 树高的时候我们用到了斐波那契数列，实际上，可以验证的是 AVL 树最差高度大约为 $1.44\log n$，红黑树最差则可以达到 $2\log n$，事实上讨论题 1 隐含了这一点，从这一层面来看，如果对一棵树的查询操作居多，那么 AVL 树会是更好的选择；
2. 但上一节我们提到，AVL 树虽然插入只需要常数次旋转即可，但在删除时可能需要 $O(\log n)$ 次旋转，而红黑树插入和删除都是常数次，有人提到在代码实现时旋转是插入和删除最耗时的操作，因此如果插入删除操作多，AVL 树不如红黑树快速，而我们知道使用 `std::map` 时的确可能遇到较多插入删除操作；
3. AVL 树需要维护树高或者 balance factor 属性，这是一个整数的大小，而红黑树只需要 1 个 bit 存储颜色即可，因此更省空间；
4. 红黑树是可持久化的数据结构，因此在函数式编程中容易实现；并且红黑树也可以支持分裂、合并等操作，这使得它可以做批量并行的插入、删除操作（实际上这与讲义最后红黑树与 B 树的关联是相关的），具体已经超出课程范畴，不再详细讨论。

上面提到的很多内容都可以在维基百科或者StackOverflow 的讨论上看到更详细的说明。事实上还有人将这一问题归结于《算法导论》这一著作没有写 AVL 树因此很多人不了解，因为有些测试表明 AVL

树的效率更高。但这些测试很难有完美的说服力，实际上上面的很多理由也并不完全有说服力，或许这就是工程上的选择的特点吧，并非一两句话可以严谨解释的。

2.2 B+ 树

B+ 树的定义非常简单，这里不再重复，只需要记住孩子个数和结点中存储的键值个数的限制（为什么会有 $\lceil M/2 \rceil$ 呢？是因为插入到爆炸的时候就是分裂到这个数量，为什么根结点最少是 2 个子结点呢？因为最开始插入到根结点爆炸的时候只能分裂成两个孩子），以及非叶结点中存储的键值的含义（实际上就是在向下搜索时区分小于键值和大于等于键值从而决定应该去第几个孩子结点）即可，然后需要理解 2-3-4 树或者 2-3 树这种简称的含义，实际上就是每个结点可能含有的子结点个数（内部可能的键值数则是可能的子结点数减 1）。需要注意的是，这里我们主要介绍 B+ 树的操作和意义。

2.2.1 B+ 树的操作

B+ 树的操作相比于红黑树直观很多，因为无需复杂的染色和旋转，也没有很多复杂的分类讨论。

1. 搜索：根据 B+ 树定义，需要在非叶结点层逐层和存储的键值比较从而确定去哪一个孩子结点。因此时间复杂度有两个重要因素：一个是树的高度，另一个是每一层搜索需要的时间。树的高度非常好计算，最差的情况也是每个结点都存 $\lceil M/2 \rceil$ 个结点，因此最大高度是 $O(\log_{\lceil M/2 \rceil} N)$ 的。然后每一层因为键值是排好序的，因此用二分查找找到要去哪个孩子结点，复杂度为 $O(\log_2 M)$，综合可得搜索的时间复杂度为 $O(\log_2 M \cdot \log_{\lceil M/2 \rceil} N) = (\log_2 M/2 + 1) \cdot \dfrac{\log_2 N}{\log_2 M/2} = O(\log N)$，注意推导中使用了换底公式.

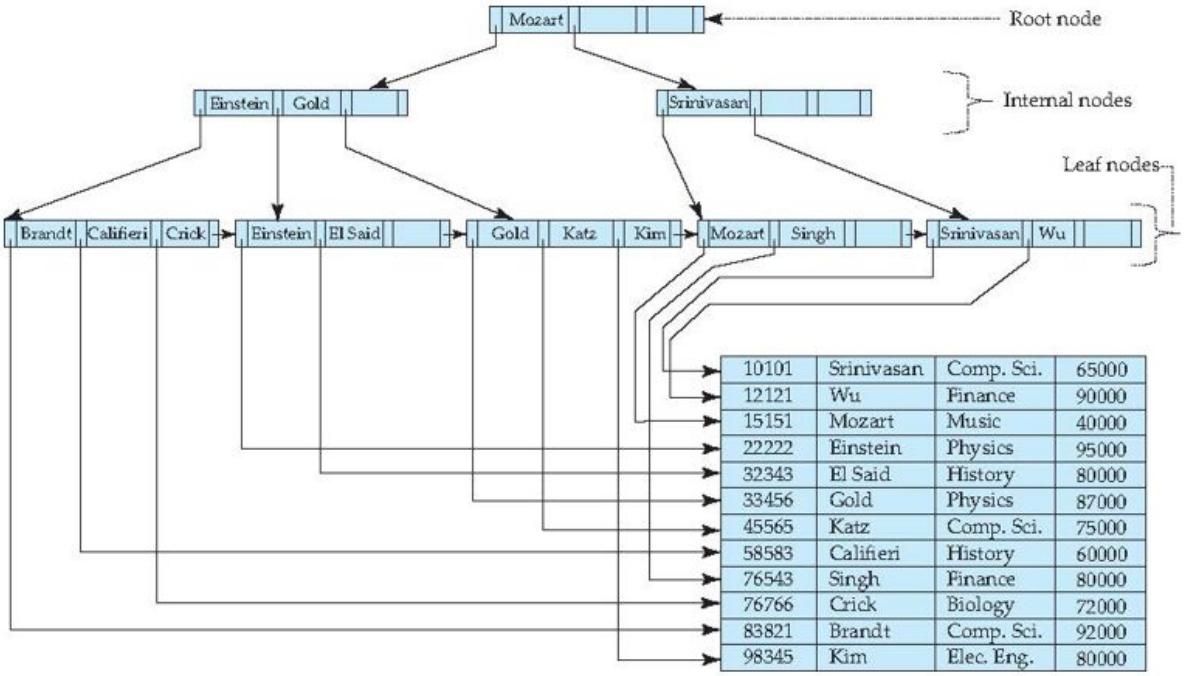
2. 插入：PPT 上的伪代码已经十分清楚，就是找到插入的位置，然后插入看结点是否放得下，放不下就分裂，如果分裂后子结点个数也过多则继续向上一层分裂，直到根结点孩子爆满则将根结点分裂并生成新的根结点，当然还要注意即使不分裂也可能需要按 B+ 树定义更新上层结点。我们知道树有 $O(\log_{\lceil M/2 \rceil} N)$ 层，每层操作最多是 $O(M)$ 的（如更新结点或者分裂，无非就是更改 $O(M)$ 个键值以及修改 $O(M)$ 个父子指针），因此整体时间复杂度为 $O(M \cdot \log_{\lceil M/2 \rceil} N) = O(\dfrac{M}{\log M} \log N)$。

3. 删除：PPT 没有要求，但想法很简单，因为只需把插入时分裂结点改为合并键值或孩子数量少的结点，当然需要注意的是，为了确保合并后键值数量不会超过 M 且减少合并次数，可以先看看兄弟结点是不是键值还很多，多的话拿一个过来即可，事实上整体时间复杂度和插入分析类似，也为 $O(\dfrac{M}{\log M} \log N)$。

【讨论 4】总结 AVL 树、红黑树和 B+ 树的搜索、插入和删除的时间复杂度，绘制成表格。

2.2.2 B+ 树的意义

B+ 树与之前学过的搜索树的目的不完全一致，事实上这学期选择了数据库系统课程的同学之后会学到 B+ 树在数据库索引中的应用。的确，B+ 树（或者 B 树族）自诞生之日起就注定是为数据库系统（或者文件系统）服务的。对于在内存中暂时存储的数据，我们通过一般的平衡二叉搜索树即可，因为根据前面的分析，插入和查询前面会带 $M/\log M$ 的倍数，M 较大时这是一个比较大的常数，因此比二叉树耗时多，并且逐层分裂更新结点也是非常复杂的操作，所以在数据量很小的时候，B+ 树看起来并不是一个很好的选择。然而，当数据量非常大，不再存储在主存而是存储在机械硬盘时，情况恰好相反。假设有 1000000 条数据存在硬盘中，这时我们需要存储在硬盘中的搜索树（因为显然这棵树也太大了，而且这棵树也需要作为硬盘索引永久存储）。如果使用二叉树，则大约有 $\log_2 1000000 = 20$ 层，因此需要经过约 20 个结点才能逐步下推到叶子结点找到真正的数据。然而，记住这棵树在磁盘里，因此每次从一个结点下降到其孩子时，磁盘都需要重新寻道、旋转，取出下孩子结点放在内存才能进行运算比较决定下一个结点，这里的时间是非常长的，可以达到毫秒级别，比内存中需要的时间要高出 5 个甚至更多数量级。所以现在时间瓶颈不在每一层的比较、重建的时间，因为这是在内存中的操作，相比于磁盘操作完全可以忽略，现在的瓶颈在在磁盘上重新定位孩子结点，因此在这种情况下，最好的选择应该是让树的高度越小越好，比如取 $M = 100$（真实的情况一般在 50-2000 之间），则大约需要 $\log_{100} 1000000 = 3$ 次磁盘操作就可以了，因此比二叉树效率高很多。

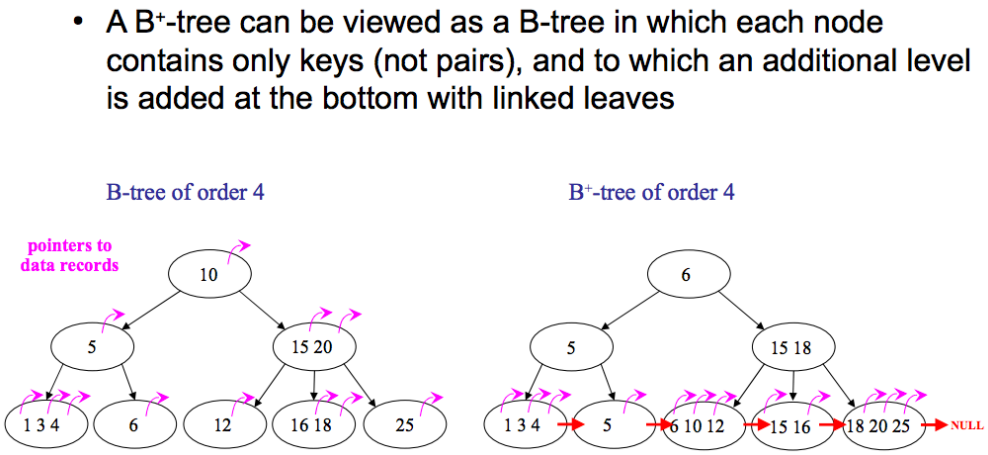


一个真实的数据库索引 B+ 树如上图所示，事实上从图中我们可以明确看出，本课程学习的 B+ 树并非真正应用的 B+ 树，一方面真正的 B+ 树叶子结点并非存的是真实的数据，而是数据表的主键（索引值，也就是寻找这条记录的依据）和指向主键对应的记录的指针，并非把真正的数据记录存储在叶子

结点；另一方面，真正的 B+ 树的叶子结点是互相连接的，这里的原因可以回顾删除操作，我们需要看兄弟结点是不是有盈余，所以叶子之间的指针可以方便直接找到兄弟结点。

2.2.3 B 树族

事实上，B+ 树这个名字就会让我们想到是不是有 B 树，的确如此，B+ 树是 B 树改进后的版本。如下图所示，我们发现 B 树叶子结点之间没有指针，B+ 树对这一点做了改进；另一方面 B 树的非叶结点也有指向真实数据记录的指针，而中间结点有的，在叶结点就不需要再出现了，因为搜索的时候搜到非叶结点中存了主键即可直接通过指针找到数据记录，无需像 B+ 树那样搜索到底。但是为什么 B+ 树放弃了部分记录的搜索便利呢？因为在非叶结点还要存储指向数据记录的指针是很耗空间的事情，假设每个结点都是一个 page 的大小，磁盘每次读写也是 1 个 page 为单位，那么这一个 page 里面如果还要存储指向数据记录的指针就会大大减少指向孩子的指针个数，显然在分叉数非常大的时候孩子中键值的数目远远多于这一层存的键值，因此这一个 page 最终能索引到的数据记录会少很多，并且回忆磁盘为了取到这一个 page 要花很多的时间，因此在非叶结点存储指向数据记录的指针是因小失大。



B* 树则是另一种 B 树族中的树，这种树要求每个结点中键值数目至少为 $2/3M$，比 B+ 树更满，因此直观理解插入时应当是相邻两个都插满了后分裂成三个。如何做到保证同时插满两个结点呢？这就需要在插满一个结点后看一看旁边的兄弟有没有满，如果没满直接给兄弟即可，满了就可以分裂。当然还有所谓的 B*+ 树，感兴趣的同学可以自行搜索，这里不再展开。

当然，一个有趣的问题是为什将这类树命名为 B 树族，发明者并未给出明确解释，它可能是 Boeing（发明者所在实验室名），balanced, between, broad, bushy 中的任何可能含义，事实上发明者之一 Edward M. McCreight 曾说 “the more you think about what the B in B-trees means, the better you understand B-trees”，因此这其中的奥秘还需同学们自行理解体会。

2.2.4 红黑树与 B 树的关联

我想很多同学都会怀疑，红黑树这种复杂的结构是怎么就想到的呢？一个有趣的事情是，红黑树的提出时间是 1978 年，比 B 树的时间 1972 年晚了六年，并且其中都有 Rudolf Bayer 参与设计。事实上，红黑树的思想正是源于想法更加直观的 B 树。实际上，2-3-4 树和 B 树的操作多有相似之处，由于文档可视化比较麻烦，读者可以直接参考这个网页或者这个网页，实际上非常直观（但请时刻记住 B 和 B+ 树是不一样的，因此很多操作也不完全一致，不要弄混了，考试都是 B+ 树）。当然这也给我们一个启示，假如我们用分叉更多的 B 树，实际上我们也可以设计彩色（多于两种颜色）树。当然有趣的是据作者所说，他们决定设计红黑树是因为红色是作者在 Xerox PARC 工作期间可用的彩色激光打印机打印的最好看的颜色，另一种说法是使用红色和黑色是因为他们可以用红笔和黑笔来画树。