算法分析与设计 树专题

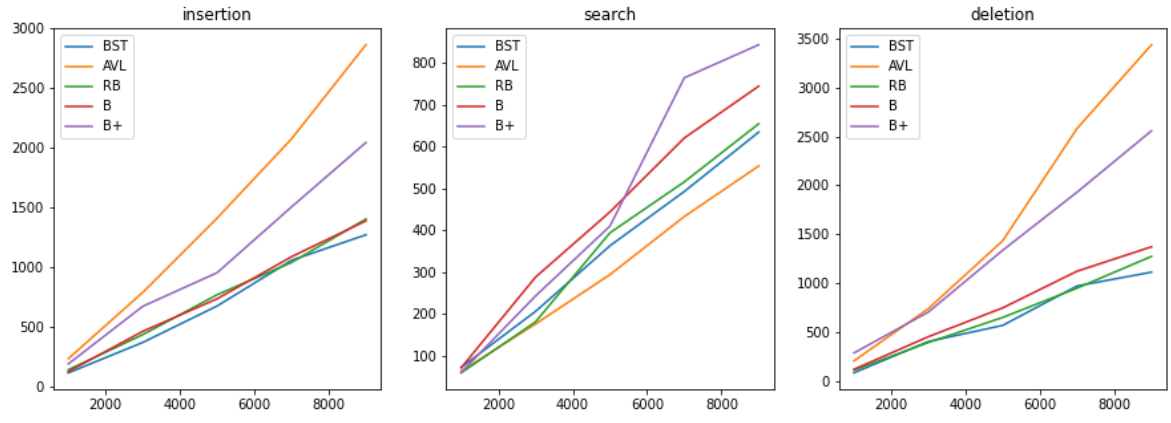
PB20061210 赖永凡

实验内容

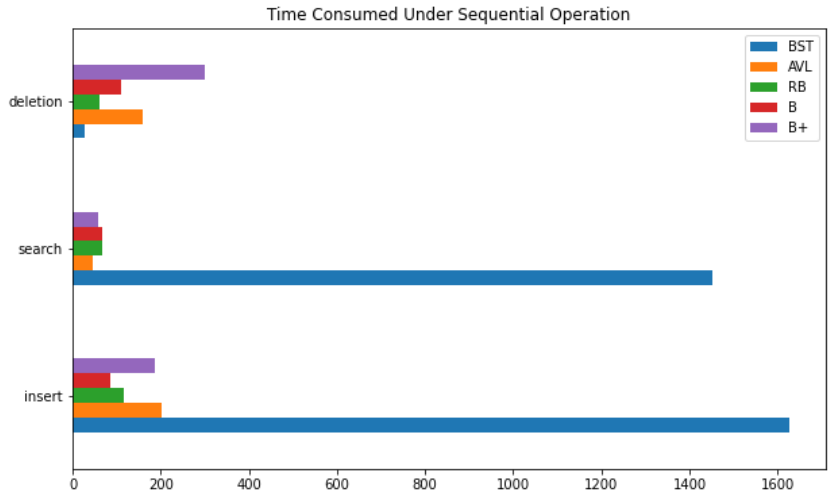
随机生成一个包含n个元素的数组，请比较在二叉搜索树、AVL树、红黑树、5阶B树、5阶B+树上进行相同节点的插入、搜索、删除三个过程的时间复杂度。

实验细节与分析

随机生成长度分别为1000，3000，5000，7000，9000的数组，分别对上述五种树进行节点的插入、搜索与删除，并记录所消耗的时间，绘制出以下折线图。



此外，我还使用长度为1000的顺序序列来对各种树进行插入、查找和删除，结果如图所示。



以下将结合实验结果对每一种树的时间复杂度进行分析。

二叉搜索树

在二叉搜索树中，查找的路径就是一条从根节点开始到待查节点的路径，此外，无论是插入还是删除操作，都是对孩子数少于二的节点进行物理上的插入和删除。因此，各操作的复杂度都是$O(h)$，其中h是树的深度。对于理想状态下的二叉搜索树，也即其是近似平衡的时候，$h = \log n$。而当随机生成元素并插入到树中时，可以认为二叉搜索树是相对平衡的，又由于其不需要对树做任何维护，因此花费的时间在各种树中最短。
但是，当进行顺序插入时，二叉搜索树的弊端也就显现了出来，插入和查找的时间复杂度都退化成了$O(n)$。注意到，由于之后的删除也是按原序进行的，因此每次删除的都是根节点，复杂度为$O(1)$，实验情况与理论相符。

AVL树

AVL树是一种高度平衡的二叉树，其定义的平衡因子始终需要保持在正负一之间，因此任意节点的左右子树高度差都在一以内。由于其高度平衡的特性，因此AVL树的搜索用时是最短的，从实验情况来看也符合分析。此外，在数学上可以通过斐波那契数列相关知识证明含n个节点的AVL树的最大深度 $h = \log_{\phi}(\sqrt{5}(n+1)) - 2$，因此其等概率查找的复杂度可以近似为 $O(\log n)$ 量级，且十分稳定。
AVL树通过旋转来维护树的形态，每次在物理上插入节点后，都需要分情况地对最小子树根进行旋转来使得树重新恢复平衡，所花费的时间是查询的 $O(h)$ 加上维护的 $O(1)$ ，最终为 $O(h)$。但在删除时，由于不能确定哪些节点失衡，因此需要从当前节点开始，不断向根节点回溯，针对每个失衡节点采用不同的旋转方式，故所需的时间为查找的 $O(c_1h)$ 加上维护的 $O(c_2h)$，最终为 $O(h)$。在具体代码中，我采用了递归调用的方式来实现，对于每个节点都在删除操作结束后判断自身的平衡性是否被破坏，这样代码简洁、便于理解，唯一的缺点就是递归调用会损耗部分的性能。

红黑树

作为一种近似平衡的二叉树，红黑树牺牲了部分平衡性，以此换取在在维护上的更好性能。红黑树舍弃了平衡因子，取而代之引入了颜色的概念，由于从根节点到叶子结点的黑高都是相同的且两个红色节点不能相连，因此根节点到叶子节点的最长路径不会超过最短路径的两倍，由此确保了数的近似平衡。理论上可以进一步证明，一个含n个内节点的红黑树其高度h最多为 $2\log(n+1)$，因此搜索的时间复杂度可以稳定在 $O(\log n)$。
红黑树在插入节点x后，首先将其染成红色，此后根据x叔叔节点的颜色以及他们在各自父节点下的位置，可分为六种情况，但前三种和后三种情况是对称的。对于情况一，仅需简单的重新涂色操作即可向祖先节点进行上溯，而情况二和情况三则能通过至多两次旋转操作使得树满足红黑的性质。因此红黑树插入的复杂度与AVL树相当，为 $O(h)$。在节点的删除时，红黑树不像AVL树需要逐节点地进行多次旋转，而同样分成前后对称的八种情况进行处理，至多仅需三次的旋转即可维护树的形态，因此其旋转操作的复杂度始终为 $O(1)$ 量级。虽然总的删除复杂度为 $O(h)$，但从实验情况来看，红黑树的插入和删除性能均优于AVL树。
此外，红黑树叶子结点（nil）的设计也有讲究，我在实现时让所有内节点共用同一个叶子节点。这样的好处在于大大节省了内存，但由于红黑树中的 nil 节点与通常的 `nullptr` 存在区别，因此在边界条件的判别时要多加注意，以避免出错。

B树

B树最主要的变化在于抛弃了传统搜索树只有两个孩子指针的限制，而是在一个节点中允许存放更多的键与指针，从而减少在磁盘中频繁访问不同的扇区或磁道。由于B树对于非根节点的最大和最小容量都做出了限制，因此可以做到近似平衡，因而树深 $h = O(\log_m n)$ 其中 m 表示树的最小度数，也即非根节点至少包含的孩子节点数（在本次实验中为3）。虽然高度得以减小，但查询时每个叶子节点的遍历复杂度都为 $O(m)$。对于实验时选取的一万个节点而言，B树高度上的优势很小，因此搜索的用时要高于二叉树。
对于插入和删除，B树主要的操作在于节点的分裂与合并，其中还涉及到和兄弟节点、父节点的互动。无论是分裂还是合并，其一次操作的复杂度均为 $O(m)$，但由于从当前节点到根节点均需要维护，因此总的操作复杂度为 $O(m\log_m n)$。从实验中效果看来，也仅仅优于AVL树，毕竟实现中数据的规模仍然较小，而且是储存在内存中，发挥不了B树在磁盘访问中的优势。

B+树

B+树是对B树的稍微改良，但结构、操作的内核没有任何改变，因此其理论上和B树的操作复杂度完全一致。B+树最大的区别就是所有的元素都储存在叶子节点中，因此所有的访问都需要下探相同的深度，所以在时间消耗上更加稳定。此外，B+树支持相邻的叶子节点直接访问，因此在顺序查找上更有优势。
由于B+树的节点定义中未设置父节点指针域，因此在插入和删除时经常需要调用从根节点开始的递归函数来查找当前节点的父节点，所以造成了B+树在插入和删除的用时上高于B树的实验结果。

实验总结感想

本次实验我亲手写了各种搜索树的算法，对各种数据结构的维护操作有了新的理解与体会。其中令我印象最深刻的当属B树和B+树，我在实现它们时花费了较多的时间，之后又在让他们能够处理重复元素这一问题上下了一番功夫。虽然最终B+树的实现存在一些瑕疵，但对于B树我个人还是很满意且自信的。
此外，本次实验我尝试了使用C++的类来构建节点与搜索树，同时让AVL树和红黑树来继承二叉搜索树的类，这样避免了重复编写搜索、中序遍历等代码，节省了一定的时间。