实验二: 平衡树搜索性能比较

PB20061372 朱云沁 May 3, 2023

1 实验说明

1.1 实验目的

熟悉二叉搜索树以及常见的平衡树数据结构, 对不同树搜索过程的时间复杂度有直观认识.

1.2 实验内容

1. 给定一个包含 n 个元素的实数数组, 分别构建二叉搜索树, AVL 树, 红黑树, 5 阶 B 树, 5 阶 B+ 树, 并实现节点的插入, 删除和搜索过程;
2. 随机生成一个包含 n 个元素的实数数组, 比较在二叉搜索树, AVL 树, 红黑树, 5 阶 B 树, 5 阶 B+ 树上进行相同节点插入, 删除和搜索三个过程时间复杂度.

2 时间复杂度分析

2.1 二叉搜索树

基本性质:

1. 若任意节点的左子树不空, 则左子树上所有节点的值均小于它根节点的值;
2. 若任意节点的右子树不空, 则右子树上所有节点的值均大于或等于它根节点的值.

由此可知, 二叉搜索树中任意节点的左右子树均为二叉搜索树. 此外, 中序遍历二叉搜索树得到的序列为递增序列.

一颗包含 10 个元素的二叉搜索树如下图所示.

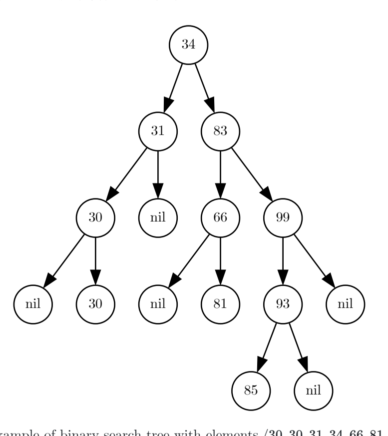

Figure 1. An example of binary search tree with elements $\langle 30, 30, 31, 34, 66, 81, 83, 85, 93, 99\rangle$.

搜索操作:

在二叉搜索树 T 中搜索元素 x, 分如下 4 种情况:

1. 若 T 为空, 则搜索失败.
2. 若 x 等于根节点, 则搜索成功;
3. 若 x 小于根节点, 则在 T 左子树中递归搜索;
4. 若 x 大于等于根节点, 则在 T 右子树中递归搜索;

固定树高 h, 考虑单次搜索操作. 最坏情况下, 元素不存在或在树的最底层, 故时间复杂度为 $O(h)$, 只需求出 h 的渐进上界.

固定元素个数 n, 改变树的形状, 依次查询各个元素. 最坏情况下, 树的高度为 n, 可视为链表上的顺序搜索, 平均搜索长度为 $\frac{n+1}{2}$, 时间复杂度为 $O(n)$; 最好情况下, 树的高度为 $O(\log n)$, 可视为二分搜索, 平均搜索长度为 $O(\log n)$, 时间复杂度为 $O(\log n)$.

可以证明, 一棵随机构建的二叉搜索树的期望高度为 $O(\log n)$, 故单次搜索操作的平均情况时间复杂度为 $O(\log n)$.

插入操作:

向二叉搜索树 T 中插入元素 x, 分以下 3 种情况:

1. 若 T 为空, 则 x 作为根节点插入;
2. 若 x 小于根节点, 则递归插入到 T 的左子树中;
3. 若 x 大于等于根节点, 则递归插入到 T 的右子树中.

显然, 若 T 中已有 n 个元素, 则插入操作的时间复杂度取决于搜索操作的时间复杂度, 即 $O(\mathbb{E}(h)) = O(\log n)$.

给定长度为 n 的数组, 依次插入元素, 构建二叉搜索树. 该过程的平均情况时间复杂度为 $\sum_{i=1}^{n} O(\log i) = O(n \log n)$, 最坏情况时间复杂度为 $\sum_{i=1}^{n} O(i) = O(n^2)$.

删除操作:

从二叉搜索树 T 中删除元素 x, 首先搜索元素 x, 然后分以下 3 种情况:

1. 若 x 为叶子节点, 则直接删除;
2. 若 x 有且仅有一个子节点, 则将该子树移植到 x 的位置;
3. 若 x 有两个子节点, 则搜索 x 的后继 y, 然后分以下 2 种情况:
   (1) y 是 x 的右孩子, 则将 y 对应子树移植到 x 的位置;
   (2) y 是 x 右子树中的左孩子, 则将 y 的右子树移植到 y 的位置, 用 y 替换 x.

显然, 插入操作的时间复杂度取决于搜索操作的时间复杂度, 即 $O(\mathbb{E}(h)) = O(\log n)$. 最坏情况下, 单次删除的时间复杂度为 $O(n)$.

2.2 AVL 树

基本性质:

定义每个节点的平衡因子为其左子树的高度减去右子树的高度. 则 AVL 树:

1. 是一颗二叉搜索树.
2. 每个节点的平衡因子的绝对值不超过 1.

带有平衡因子 1, 0 或 -1 的节点被认为是平衡的, 否则被认为是不平衡的. 显然, AVL 树的左右子树仍为 AVL 树. 易知, 对于高度为 h 的 AVL 树, 其节点个数至少为 $F_h - 1$, 其中 $\{F_i\}_{i=1}^{\infty}$ 为 Fibonacci 数列. 因此, AVL 树的高度为 $O(\log n)$.

一颗包含 10 个元素的 AVL 树如下图所示.

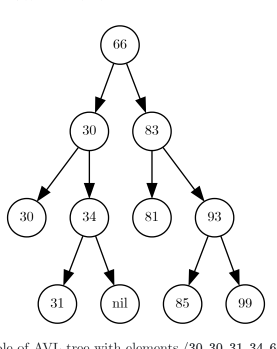

Figure 2. An example of AVL tree with elements $\langle 30, 30, 31, 34, 66, 81, 83, 85, 93, 99\rangle$.

搜索操作:

同二叉搜索树, 时间复杂度为 $O(h) = O(\log n)$.

插入操作:

首先, 调用二叉搜索树的插入操作. 然后, 从被插入节点 x 的父节点开始, 逐层向上更新 x 的祖先的平衡因子. 若发现祖先节点 y 不平衡, 则分以下 4 种情况:

1. 若 y 的平衡因子为 $+2$, y 的左孩子的平衡因子为 $+1$, 则以 y 为根进行右旋;
2. 若 y 的平衡因子为 $+2$, y 的左孩子的平衡因子为 -1, 则对 y 的左子树进行左旋, 转向情况 1;
3. 若 y 的平衡因子为 -2, y 的右孩子的平衡因子为 -1, 则以 y 为根进行左旋;
4. 若 y 的平衡因子为 -2, y 的右孩子的平衡因子为 $+1$, 则对 y 的右子树进行右旋, 转向情况 3.

搜索及重新平衡的时间复杂度均为 $O(h)$, 故单次插入操作的总时间复杂度为 $O(h) = O(\log n)$. 此外, 单次插入操作至多需要两次旋转.

构建二叉搜索树的时间复杂度为 $\sum_{i=1}^{n} O(\log i) = O(n \log n)$.

删除操作:

首先, 调用二叉搜索树的删除操作, 记被删除节点为 x. 分以下 2 种情况:

1. 若 x 被其左右孩子替换, 则从替换后的节点开始, 逐层向上重新平衡;
2. 若 x 被其后继 y 替换, 且 y 不是 x 的右孩子, 则从 y 的原父节点开始, 逐层向上重新平衡.

搜索及重新平衡的时间复杂度均为 $O(h)$, 故单次删除操作的总时间复杂度为 $O(h) = O(\log n)$. 此外, 单次删除操作可能需要多于两次的旋转.

2.3 红黑树

基本性质:

1. 每个节点或是红色的, 或是黑色的;
2. 根节点为黑色;
3. nil 节点为黑色;
4. 红色节点的孩子为黑色;
5. 从节点 x 到其后代叶子节点的所有简单路径, 包含的黑色节点数目相等, 称为 x 的黑高.

记以 x 为根的子树的节点数为 $n(x)$, 高度为 $h(x)$, 黑高为 $bh(x)$. 由性质 4, 有 $bh(x) \geq h(x)/2$.

当 $bh(x) = 0$, 即 $x =$ nil 时, 有 $n(x) = 0$; 假设当 $bh(x) = k$ 时, $n(x) \geq 2^{bh(x)} - 1$, 则当 $bh(x) = k + 1$ 时,

$$
\begin{aligned}
n(x) &= n(x.left) + n(x.right) + 1 \\
&= 2^{bh(x.left)} + 2^{bh(x.left)} - 1 \\
&\geq 2^{bh(x)-1} + 2^{bh(x)-1} - 1 \\
&= 2^{bh(x)} - 1
\end{aligned}
$$

归纳得, $n(x) \geq 2^{bh(x)} - 1$, 即 $bh(x) \leq \log(n(x) + 1)$. 因此 $h \leq 2\log(n + 1)$.

一颗包含 10 个元素的红黑树如下图所示.

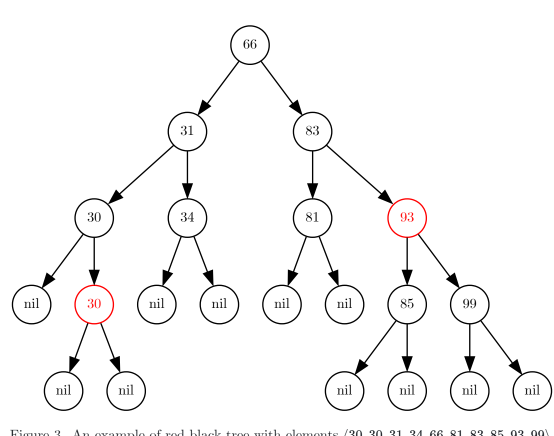

Figure 3. An example of red-black tree with elements $\langle 30, 30, 31, 34, 66, 81, 83, 85, 93, 99\rangle$.

搜索操作:

同二叉搜索树. 时间复杂度为 $O(h) = O(\log n)$.

插入操作:

首先, 调用二叉搜索树的插入操作, 将被插入节点 x 染红. 然后, 分 3 种情况:

1. 若 x 为根, 终止;
2. 若 x 的父节点 p 为黑, 终止;
3. 若 x 的父节点 p 为红, 逐层向上递归修复红黑树性质. 记 x 的祖父节点为 gp, 叔叔节点为 u. 分 6 种情况:
   (1) p 为左孩子, u 为右孩子. 若 p 与 u 均为红. 则将 p 和 u 染黑, gp 染红, 对 gp 递归;
   (2) p 为左孩子, u 为右孩子. 若 p 为红, u 为黑, 且 x 为右孩子, 则以 p 为根进行左旋, 对 p 递归, 转向情况 (3);
   (3) p 为左孩子, u 为右孩子. 若 p 为红, u 为黑, 且 x 为左孩子, 则将 p 染黑, gp 染红, 以 gp 为根右旋, 终止;
   (4) p 为右孩子, u 为左孩子. 若 p 与 u 均为红. 则将 p 和 u 染黑, gp 染红, 对 gp 递归;
   (5) p 为右孩子, u 为左孩子. 若 p 为红, u 为黑, 且 x 为左孩子, 则以 p 为根进行右旋, 对 p 递归, 转向情况 (6);
   (6) p 为右孩子, u 为左孩子. 若 p 为红, u 为黑, 且 x 为右孩子, 则将 p 染黑, gp 染红, 以 gp 为根左旋, 终止.

最后, 将根节点染黑.

搜索与修复的时间复杂度均为 $O(h)$, 故单次插入操作的总时间复杂度为 $O(\log n)$.

单次插入操作至多需要两次旋转.

删除操作:

首先, 调用二叉搜索树的删除操作, 记被删除节点为 x. 分 2 种情况:

1. 若 x 被其孩子替换, 记该孩子为 z;
2. 若 x 被其后继 y 替换, 且 y 不是 x 的右孩子, 则等价于交换 x 与 y 的值后删除 y. 由于 y 被其右孩子替换, 记该孩子为 z.

若被 z 替换的节点为红, 终止; 否则, 分 3 种情况:

1. 若 z 为根, 终止;
2. 若 z 为红, 终止;
3. 若 z 为黑, 逐层向上递归修复红黑树性质. 记 z 的父节点为 p, 兄弟节点为 s. 分 8 种情况:
   (1) z 为左孩子, s 为右孩子. 若 z 为黑, s 为红, 则将 s 染黑, p 染红, 以 p 为根左旋, 对 z 递归, 转向情况 (2), (3), (4).
   (2) z 为左孩子, s 为右孩子. 若 z 与 s 均为黑, 且 s 的左右孩子均为黑, 则将 s 染红, 对 p 递归.
   (3) z 为左孩子, s 为右孩子. 若 z 与 s 均为黑, 且 s 的左孩子为红, 右孩子为黑, 则将 s 染红, s 的左孩子染黑, 以 s 为根右旋, 对 z 递归, 转向情况 (4).
   (4) z 为左孩子, s 为右孩子. 若 z 与 s 均为黑, 且 s 的右孩子为红, 则将 s 染成 p 的颜色, p 染黑, s 的右孩子染黑, 以 p 为根左旋, 终止.
   (5) z 为右孩子, s 为左孩子. 若 z 为黑, s 为红, 则将 s 染黑, p 染红, 以 p 为根右旋, 对 z 递归, 转向情况 (6), (7), (8).
   (6) z 为右孩子, s 为左孩子. 若 z 与 s 均为黑, 且 s 的左右孩子均为黑, 则将 s 染红, 对 p 递归.
   (7) z 为右孩子, s 为左孩子. 若 z 与 s 均为黑, 且 s 的右孩子为红, 左孩子为黑, 则将 s 染红, s 的右孩子染黑, 以 s 为根左旋, 对 z 递归, 转向情况 (8).
   (8) z 为右孩子, s 为左孩子. 若 z 与 s 均为黑, 且 s 的左孩子为红, 则将 s 染成 p 的颜色, p 染黑, s 的左孩子染黑, 以 p 为根右旋, 终止.

最后, 将 z 染黑.

搜索与修复的时间复杂度均为 $O(h)$, 故单次删除操作的总时间复杂度为 $O(\log n)$.

单次删除操作至多需要三次旋转, 优于 AVL 树.

2.4 5 阶 B 树

基本性质:

1. 叶子节点位于同一层.
2. 根结点的关键字个数不超过 4.
3. 每个非根叶子节点包含 $[1, 4]$ 个关键字;
4. 每个非根内节点包含 $[2, 5]$ 个孩子;
5. 每个内节点若包含 m 个孩子, 则包含 $m - 1$ 个关键字;
6. 每个内节点的第 i 个关键字大于等于其第 i 个孩子中的关键字, 小于等于其第 $i + 1$ 个孩子中的关键字.
7. 每个节点中的关键字按升序排列.

易知, n 个关键字的 5 阶 B 树的高度为 $O(\log n)$.

一颗包含 10 个元素的 5 阶 B 树如下图所示.

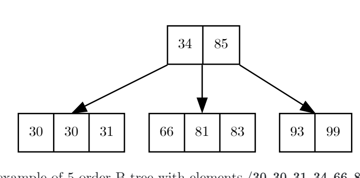

Figure 4. An example of 5-order B tree with elements $\langle 30, 30, 31, 34, 66, 81, 83, 85, 93, 99\rangle$.

搜索操作:

在根节点中二分搜索关键字 k, 得到第一个大于等于 k 的关键字 k_i 及其下标 i. 若 $k_i = k$, 则搜索成功; 否则, 递归搜索第 i 个孩子.

由于阶数固定为 5, 单次二分搜索的时间复杂度为 $O(\log M) = O(1)$, 故单次搜索操作的时间复杂度为 $O(h) = O(\log n)$.

插入操作:

首先搜索被插入元素所对应的叶子结点 x. 若 x 未满, 即 x 关键字个数小于 4, 则直接插入; 否则, 将 x 分裂为两个节点 y 和 z, 步骤如下:

1. 从叶子节点的关键字和新的关键字中选择出中位数;
2. 小于中位数的元素放入 y, 大于中位数的元素放入 z, 中位数作为分隔值;
3. 若 x 为根节点, 则新建根节点 r, 将分隔值与其相邻孩子 y 与 z 插入父节点 r, 终止;
4. 否则, 向上递归, 将分隔值与其相邻孩子 y 与 z 插入父节点.

最后, 逐层向上更新索引关键字.

由于阶数固定, 搜索与递归分裂的时间复杂度均为 $O(h) = O(\log n)$, 故单次插入操作的时间复杂度为 $O(\log n)$.

删除操作:

首先搜索被删除元素所对应的结点 x. 若 x 为内节点, 将其与前驱或后继交换, 进而递归删除叶子结点中的前驱或后继. 对叶子结点 x 执行以下步骤:

1. x 关键字个数大于 1, 则直接删除;
2. 否则, 尝试从 x 富裕的兄弟节点中借取一个关键字, 插入 x 中;
3. 若失败, 即 x 的兄弟节点不存在或关键字个数均为 1, 则将 x 与某个兄弟节点合并, 将父节点中的关键字下移至合并后的节点中. 此时, 需要递归删除 x 的父节点中的关键字.
4. 若递归删除至根结点, 使得根结点为空, 则将其唯一孩子作为新的根结点.

最后, 逐层向上更新索引关键字.

由于阶数固定, 搜索与递归借取/合并的时间复杂度均为 $O(h) = O(\log n)$, 故单次删除操作的时间复杂度为 $O(\log n)$.

2.5 5 阶 B+ 树

基本性质:

1. 元素存储于叶子结点中.
2. 叶子节点位于同一层.
3. 根结点的关键字个数不超过 5;
4. 每个非根叶子节点包含 $[2, 5]$ 个关键字;
5. 每个内节点包含 $[2, 5]$ 个孩子;
6. 每个内节点若包含 m 个孩子, 则包含 m 个索引关键字;
7. 每个内节点的第 i 个索引关键字为第 i 个孩子的最大关键字;
8. 每个节点中的关键字按升序排列.

易知, n 个关键字的 5 阶 B+ 树的高度为 $O(\log n)$.

一颗包含 10 个元素的 5 阶 B+ 树如下图所示.

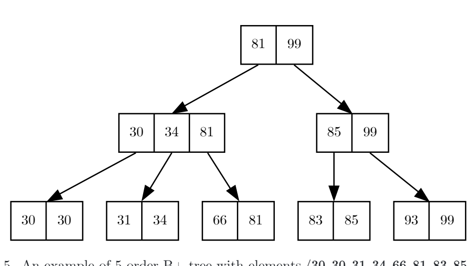

Figure 5. An example of 5-order B+ tree with elements $\langle 30, 30, 31, 34, 66, 81, 83, 85, 93, 99\rangle$.

搜索操作:

在根节点中二分搜索关键字 k, 得到第一个大于等于 k 的关键字 k_i 及其下标 i. 若下标越界, 则搜索失败; 否则, 递归搜索第 i 个孩子, 直到在叶子结点中搜索到 $k_i = k$.

由于阶数固定为 5, 单次二分搜索的时间复杂度为 $O(\log M) = O(1)$, 故单次搜索操作的时间复杂度为 $O(h) = O(\log n)$.

插入操作:

首先搜索被插入元素所对应的叶子结点 x. 若 x 未满, 即 x 关键字个数小于 5, 则直接插入; 否则, 将 x 分裂为两个节点 y 和 z, 步骤如下:

1. 将 x 的关键字和新的关键字中较小的 3 个放入 y, 较大的 3 个的关键字放入 z;
2. 若 x 为根节点, 则新建根节点 r, 将 y 与 z 和相应索引插入 r, 终止;
3. 否则, 向上递归, 将 y 与 z 和相应索引插入父节点.

最后, 逐层向上更新索引关键字.

由于阶数固定, 搜索与递归分裂的时间复杂度均为 $O(h) = O(\log n)$, 故单次插入操作的时间复杂度为 $O(\log n)$.

删除操作:

首先搜索被删除元素所对应的叶子结点 x. 执行以下步骤:

1. x 关键字个数大于 2, 则直接删除;
2. 否则, 尝试从 x 富裕的兄弟节点中借取一个关键字, 插入 x 中;
3. 若失败, 即 x 的兄弟节点关键字个数均为 2, 则将 x 与某个兄弟节点合并. 此时, 需要递归删除 x 的父节点中的关键字.
4. 若递归删除至根结点, 使得根结点中关键字个数为 1, 则将其唯一孩子作为新的根结点.

最后, 逐层向上更新索引关键字.

由于阶数固定, 搜索与递归借取/合并的时间复杂度均为 $O(h) = O(\log n)$, 故单次删除操作的时间复杂度为 $O(\log n)$.

3 运行时间比较

二叉搜索树, AVL 树, 红黑树, 5 阶 B 树, 5 阶 B+ 树的时间复杂度比较如下表所示:

Table 1. Comparison of space and time complexity of different search trees.

Tree	Search Average	Insert Average	Delete Average	Search Worst	Insert Worst	Delete Worst
BST	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
RB	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
B	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
B+	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

对于 $n \in \{1 \times 10^4, 2 \times 10^4, 3 \times 10^4, 4 \times 10^4, 5 \times 10^4, 6 \times 10^4, 7 \times 10^4, 8 \times 10^4, 9 \times 10^4, 10 \times 10^4\}$, 分别随机生成包含 n 个元素的 float 型数组, 分别记录在二叉搜索树, AVL 树, 红黑树, 5 阶 B 树, 5 阶 B+ 树上随机插入所有元素, 随机搜索所有元素, 随机删除所有元素所花费的运行时间.

对于每一组 n 和搜索树, 取 5 个随机种子, 重复 5 次实验, 求出插入所有元素, 搜索所有元素, 删除所有元素三种操作所需平均时间, 如表 2, 3, 4 所示.

以 $n \log n$ 为横坐标, 绘制 5 次重复实验得到插入所有元素, 搜索所有元素, 删除所有元素三种操作所需平均时间三种操作运行时间与数组规模 n 的关系图, 如图 6, 7, 8 所示.

3.1 插入操作

Table 2. Time cost of inserting all elements into different search trees with different array sizes. For each cell, the first number in parentheses is the mean value of 5 runs, and the second number is the standard deviation.

n	Binary Search Tree	AVL Tree	Red-Black Tree
10000	$(1.387 \pm 0.018) \times 10^{-3}$	$(4.153 \pm 0.143) \times 10^{-3}$	$(1.703 \pm 0.061) \times 10^{-3}$
20000	$(3.358 \pm 0.729) \times 10^{-3}$	$(1.151 \pm 0.454) \times 10^{-2}$	$(3.657 \pm 0.130) \times 10^{-3}$
30000	$(5.945 \pm 1.256) \times 10^{-3}$	$(1.603 \pm 0.222) \times 10^{-2}$	$(5.847 \pm 0.160) \times 10^{-3}$
40000	$(6.979 \pm 0.228) \times 10^{-3}$	$(1.944 \pm 0.060) \times 10^{-2}$	$(8.343 \pm 0.583) \times 10^{-3}$
50000	$(8.958 \pm 0.333) \times 10^{-3}$	$(2.496 \pm 0.055) \times 10^{-2}$	$(1.072 \pm 0.053) \times 10^{-2}$
60000	$(1.154 \pm 0.086) \times 10^{-2}$	$(3.109 \pm 0.086) \times 10^{-2}$	$(1.306 \pm 0.054) \times 10^{-2}$
70000	$(1.320 \pm 0.048) \times 10^{-2}$	$(3.678 \pm 0.084) \times 10^{-2}$	$(1.647 \pm 0.157) \times 10^{-2}$
80000	$(1.641 \pm 0.167) \times 10^{-2}$	$(4.290 \pm 0.078) \times 10^{-2}$	$(1.804 \pm 0.070) \times 10^{-2}$
90000	$(1.821 \pm 0.066) \times 10^{-2}$	$(4.883 \pm 0.130) \times 10^{-2}$	$(2.306 \pm 0.261) \times 10^{-2}$
100000	$(2.095 \pm 0.054) \times 10^{-2}$	$(5.680 \pm 0.219) \times 10^{-2}$	$(2.398 \pm 0.044) \times 10^{-2}$

n	5-Order B Tree	5-Order B+ Tree
10000	$(2.602 \pm 0.306) \times 10^{-3}$	$(2.862 \pm 0.204) \times 10^{-3}$
20000	$(5.113 \pm 0.050) \times 10^{-3}$	$(6.466 \pm 0.866) \times 10^{-3}$
30000	$(8.133 \pm 0.402) \times 10^{-3}$	$(9.718 \pm 1.264) \times 10^{-3}$
40000	$(1.089 \pm 0.031) \times 10^{-2}$	$(1.272 \pm 0.046) \times 10^{-2}$
50000	$(1.399 \pm 0.051) \times 10^{-2}$	$(1.619 \pm 0.049) \times 10^{-2}$
60000	$(1.718 \pm 0.018) \times 10^{-2}$	$(2.041 \pm 0.106) \times 10^{-2}$
70000	$(2.017 \pm 0.035) \times 10^{-2}$	$(2.349 \pm 0.039) \times 10^{-2}$
80000	$(2.362 \pm 0.032) \times 10^{-2}$	$(2.776 \pm 0.070) \times 10^{-2}$
90000	$(2.704 \pm 0.069) \times 10^{-2}$	$(3.233 \pm 0.301) \times 10^{-2}$
100000	$(3.025 \pm 0.050) \times 10^{-2}$	$(3.557 \pm 0.109) \times 10^{-2}$

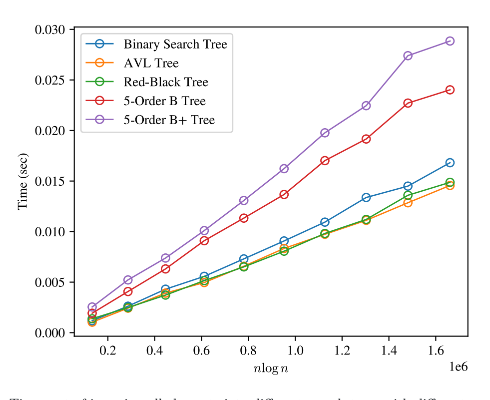

Figure 6. Time cost of inserting all elements into different search trees with different array sizes.

由表 2 和 图 6 可知, 给定规模为 n 的实数数组, 构建 5 种搜索树所花费的时间均与 $n \log n$ 大致呈线性关系, 与时间复杂度分析结果相符, 验证了 5 种搜索树的插入操作平均时间复杂度均为 $O(\log n)$.

3.2 搜索操作

Table 3. Time cost of searching all elements in different search trees with different array sizes.

n	Binary Search Tree	AVL Tree	Red-Black Tree
10000	$(1.387 \pm 0.018) \times 10^{-3}$	$(4.153 \pm 0.143) \times 10^{-3}$	$(1.703 \pm 0.061) \times 10^{-3}$
20000	$(3.358 \pm 0.729) \times 10^{-3}$	$(1.151 \pm 0.454) \times 10^{-2}$	$(3.657 \pm 0.130) \times 10^{-3}$
30000	$(5.945 \pm 1.256) \times 10^{-3}$	$(1.603 \pm 0.222) \times 10^{-2}$	$(5.847 \pm 0.160) \times 10^{-3}$
40000	$(6.979 \pm 0.228) \times 10^{-3}$	$(1.944 \pm 0.060) \times 10^{-2}$	$(8.343 \pm 0.583) \times 10^{-3}$
50000	$(8.958 \pm 0.333) \times 10^{-3}$	$(2.496 \pm 0.055) \times 10^{-2}$	$(1.072 \pm 0.053) \times 10^{-2}$
60000	$(1.154 \pm 0.086) \times 10^{-2}$	$(3.109 \pm 0.086) \times 10^{-2}$	$(1.306 \pm 0.054) \times 10^{-2}$
70000	$(1.320 \pm 0.048) \times 10^{-2}$	$(3.678 \pm 0.084) \times 10^{-2}$	$(1.647 \pm 0.157) \times 10^{-2}$
80000	$(1.641 \pm 0.167) \times 10^{-2}$	$(4.290 \pm 0.078) \times 10^{-2}$	$(1.804 \pm 0.070) \times 10^{-2}$
90000	$(1.821 \pm 0.066) \times 10^{-2}$	$(4.883 \pm 0.130) \times 10^{-2}$	$(2.306 \pm 0.261) \times 10^{-2}$
100000	$(2.095 \pm 0.054) \times 10^{-2}$	$(5.680 \pm 0.219) \times 10^{-2}$	$(2.398 \pm 0.044) \times 10^{-2}$

n	5-Order B Tree	5-Order B+ Tree
10000	$(2.602 \pm 0.306) \times 10^{-3}$	$(2.862 \pm 0.204) \times 10^{-3}$
20000	$(5.113 \pm 0.050) \times 10^{-3}$	$(6.466 \pm 0.866) \times 10^{-3}$
30000	$(8.133 \pm 0.402) \times 10^{-3}$	$(9.718 \pm 1.264) \times 10^{-3}$
40000	$(1.089 \pm 0.031) \times 10^{-2}$	$(1.272 \pm 0.046) \times 10^{-2}$
50000	$(1.399 \pm 0.051) \times 10^{-2}$	$(1.619 \pm 0.049) \times 10^{-2}$
60000	$(1.718 \pm 0.018) \times 10^{-2}$	$(2.041 \pm 0.106) \times 10^{-2}$
70000	$(2.017 \pm 0.035) \times 10^{-2}$	$(2.349 \pm 0.039) \times 10^{-2}$
80000	$(2.362 \pm 0.032) \times 10^{-2}$	$(2.776 \pm 0.070) \times 10^{-2}$
90000	$(2.704 \pm 0.069) \times 10^{-2}$	$(3.233 \pm 0.301) \times 10^{-2}$
100000	$(3.025 \pm 0.050) \times 10^{-2}$	$(3.557 \pm 0.109) \times 10^{-2}$

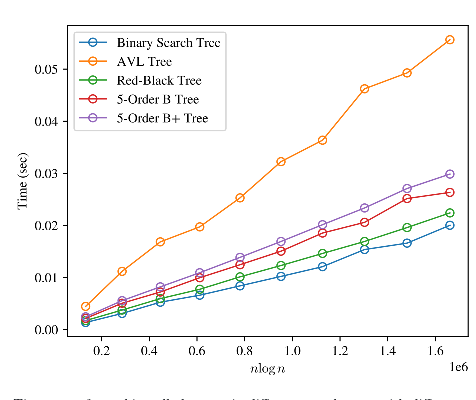

Figure 7. Time cost of searching all elements in different search trees with different array sizes.

由表 3 和 图 7 可知, 在元素数目为 n 的 5 种搜索树中逐个搜索所有元素所花费的时间均与 $n \log n$ 大致呈线性关系, 与时间复杂度分析结果相符, 验证了 5 种搜索树的搜索操作平均时间复杂度均为 $O(\log n)$.

3.3 删除操作

Table 4. Time cost of deleting all elements from different search trees with different array sizes.

n	Binary Search Tree	AVL Tree	Red-Black Tree
10000	$(1.387 \pm 0.018) \times 10^{-3}$	$(4.153 \pm 0.143) \times 10^{-3}$	$(1.703 \pm 0.061) \times 10^{-3}$
20000	$(3.358 \pm 0.729) \times 10^{-3}$	$(1.151 \pm 0.454) \times 10^{-2}$	$(3.657 \pm 0.130) \times 10^{-3}$
30000	$(5.945 \pm 1.256) \times 10^{-3}$	$(1.603 \pm 0.222) \times 10^{-2}$	$(5.847 \pm 0.160) \times 10^{-3}$
40000	$(6.979 \pm 0.228) \times 10^{-3}$	$(1.944 \pm 0.060) \times 10^{-2}$	$(8.343 \pm 0.583) \times 10^{-3}$
50000	$(8.958 \pm 0.333) \times 10^{-3}$	$(2.496 \pm 0.055) \times 10^{-2}$	$(1.072 \pm 0.053) \times 10^{-2}$
60000	$(1.154 \pm 0.086) \times 10^{-2}$	$(3.109 \pm 0.086) \times 10^{-2}$	$(1.306 \pm 0.054) \times 10^{-2}$
70000	$(1.320 \pm 0.048) \times 10^{-2}$	$(3.678 \pm 0.084) \times 10^{-2}$	$(1.647 \pm 0.157) \times 10^{-2}$
80000	$(1.641 \pm 0.167) \times 10^{-2}$	$(4.290 \pm 0.078) \times 10^{-2}$	$(1.804 \pm 0.070) \times 10^{-2}$
90000	$(1.821 \pm 0.066) \times 10^{-2}$	$(4.883 \pm 0.130) \times 10^{-2}$	$(2.306 \pm 0.261) \times 10^{-2}$
100000	$(2.095 \pm 0.054) \times 10^{-2}$	$(5.680 \pm 0.219) \times 10^{-2}$	$(2.398 \pm 0.044) \times 10^{-2}$

n	5-Order B Tree	5-Order B+ Tree
10000	$(2.602 \pm 0.306) \times 10^{-3}$	$(2.862 \pm 0.204) \times 10^{-3}$
20000	$(5.113 \pm 0.050) \times 10^{-3}$	$(6.466 \pm 0.866) \times 10^{-3}$
30000	$(8.133 \pm 0.402) \times 10^{-3}$	$(9.718 \pm 1.264) \times 10^{-3}$
40000	$(1.089 \pm 0.031) \times 10^{-2}$	$(1.272 \pm 0.046) \times 10^{-2}$
50000	$(1.399 \pm 0.051) \times 10^{-2}$	$(1.619 \pm 0.049) \times 10^{-2}$
60000	$(1.718 \pm 0.018) \times 10^{-2}$	$(2.041 \pm 0.106) \times 10^{-2}$
70000	$(2.017 \pm 0.035) \times 10^{-2}$	$(2.349 \pm 0.039) \times 10^{-2}$
80000	$(2.362 \pm 0.032) \times 10^{-2}$	$(2.776 \pm 0.070) \times 10^{-2}$
90000	$(2.704 \pm 0.069) \times 10^{-2}$	$(3.233 \pm 0.301) \times 10^{-2}$
100000	$(3.025 \pm 0.050) \times 10^{-2}$	$(3.557 \pm 0.109) \times 10^{-2}$

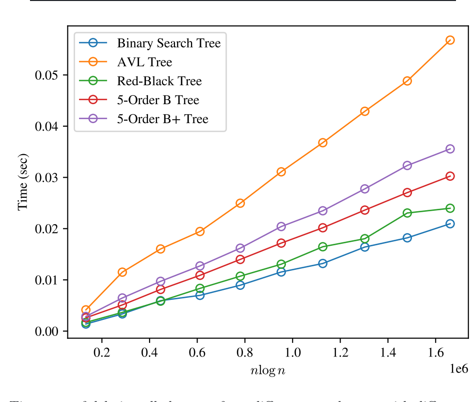

Figure 8. Time cost of deleting all elements from different search trees with different array sizes.

由表 4 和 图 8 可知, 从元素数目为 n 的 5 种搜索树中逐个删除所有元素所花费的时间均与 $n \log n$ 大致呈线性关系, 与时间复杂度分析结果相符, 验证了 5 种搜索树的删除操作平均时间复杂度均为 $O(\log n)$.

4 结论

1. 最坏情况下, 二叉搜索树的插入、搜索和删除操作的时间复杂度均为 $O(n)$; 4 种平衡树由于其高度为 $O(\log n)$, 插入、搜索和删除操作的时间复杂度均为 $O(\log n)$.
2. 本实验中, 由于数组随机生成, 数据分布均匀, 测得运行时间均接近于平均情况. 实验结果验证了 5 种搜索树的插入、搜索和删除操作的平均时间复杂度均为 $O(\log n)$.
3. 平衡树的效率极大依赖于具体实现. 本实验中, 可作如下观察:
   (1) 插入操作中, AVL 树与红黑树两种平衡树优于二叉搜索树, 说明了平衡的重要性.
   (2) 二叉搜索树在搜索与删除操作中表现最优, 可能是由于其算法简单, 操作数较少;
   (3) B 树与 B+ 树在三种操作中均花费较多时间, 且 B+ 树的运行时间高于 B 树, 可能是由于内存操作花费了较多时间;
   (4) B+ 树搜索操作花费时间最为稳定, 这是由于元素均存储在叶子结点中, 与其原理相符.
   (5) 本实验采用面向对象的实现方式, AVL 树与红黑树的搜索操作直接继承自二叉搜索树, 然而 AVL 树在搜索与删除操作中表现较差, 猜测其原理可能存在一定的缺陷, 也可能是部分函数的实现较为低效.