搜索树数据结构性能比较实验报告

张圣朗 PB22000242

March 30th, 2025

1 实验目的

1. 熟悉二叉搜索树 (BST) 及常见平衡树数据结构的基本原理；
2. 比较 BST、AVL 树、红黑树和 5 阶 B 树在插入、删除和搜索操作上的性能差异；
3. 通过实验数据对不同搜索树的时间复杂度建立直观认识。

2 实验项目

1. 随机生成包含 n 个整数的数据集（取值范围 1-10000），分别在 BST、AVL 树、红黑树和 5 阶 B 树上进行相同节点的插入操作，记录构建时间；
2. 对已构建的树执行搜索操作（查询固定 key），记录搜索时间；
3. 对已构建的树执行删除操作（删除固定 key），记录删除时间；
4. 改变数据规模 n（5 万、10 万、20 万、50 万、100 万、200 万），重复上述实验。

3 实验原理

1. 二叉搜索树是基于二分查找思想设计的经典数据结构。作为最基本的搜索树结构，它通过维护” 左子树键值小于根节点，右子树键值大于根节点” 的有序性质实现高效查找。当插入新节点时，算法会沿着搜

索路径找到合适的叶子位置进行插入；删除操作则需要处理叶子节点、单子节点和双子节点三种不同情况。虽然二叉搜索树在随机数据下平均表现良好，但其最坏情况下会退化成链表结构，导致时间复杂度恶化至 $O(n)$，这使得其在实际应用中需要更稳定的变种。

2. AVL 树是最早的自平衡二叉搜索树，其核心原理是通过平衡因子维护严格的平衡条件。平衡因子定义为左子树高度减去右子树高度，要求每个节点的平衡因子绝对值不超过 1。当插入或删除节点破坏平衡时，AVL 树通过旋转操作恢复平衡。

   具体操作步骤：

   1. **插入操作**：

   (a) 按照二叉搜索树规则找到插入位置，创建新节点

   (b) 从插入点向上回溯，更新祖先节点的高度信息

   (c) 检查每个祖先节点的平衡因子：

   - 若发现平衡因子数之差大于 1，根据不平衡类型执行旋转：
     - LL 型：在失衡节点右旋
     - RR 型：在失衡节点左旋
     - LR 型：先左旋左子树，再右旋失衡节点
     - RL 型：先右旋右子树，再左旋失衡节点

   (d) 继续回溯直到根节点

   2. **删除操作**：

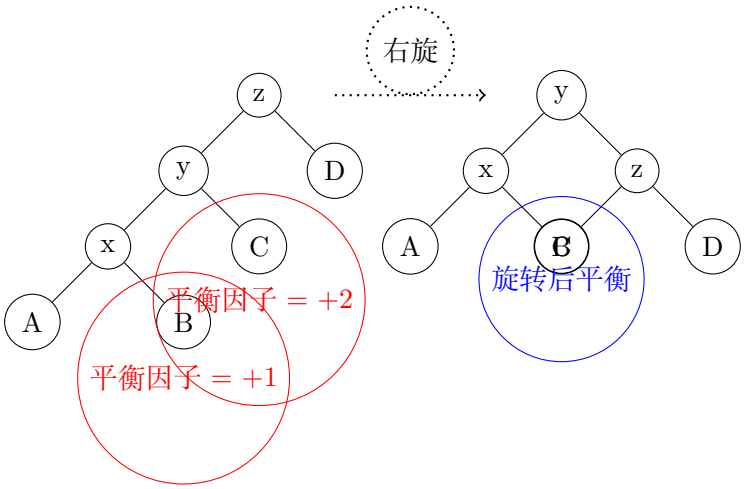
   (a) 定位待删除节点，按二叉搜索树规则执行删除

   (b) 从删除位置向上回溯，更新高度信息

   (c) 检查每个祖先节点的平衡因子，发现失衡时执行旋转（同插入操作）

   (d) 可能需要多次旋转直至根节点

   示例：当节点左子树高度比右子树大 2，且左子树的左子树导致不平衡时：



3. 红黑树通过颜色约束维持近似平衡，满足以下性质：

   - 每个节点非红即黑
   - 根节点和 NIL 叶子为黑
   - 红色节点的子节点必须为黑
   - 从任一节点到其叶子的所有路径包含相同数目的黑色节点（黑高相同）

   具体操作步骤：

   1. **插入操作**：

   (a) 按二叉搜索树规则插入新节点 z，初始设为红色

   (b) 若父节点 p[z] 为黑，无需调整

   (c) 若 p[z] 为红，则根据叔节点颜色处理：

   - Case1：叔节点为红
     i. 将父节点和叔节点变黑
     ii. 祖父节点变红
     iii. z 上移至祖父节点继续调整
   - Case2：叔节点为黑且 z 为右孩子
     i. z 上移至父节点
     ii. 对 z 执行左旋

iii. 转入 Case3 处理

- Case3：叔节点为黑且 z 为左孩子
  i. 父节点变黑，祖父节点变红
  ii. 对祖父节点执行右旋

2. **删除操作**：

(a) 执行标准二叉搜索树删除，记 x 为实际删除节点

(b) 若 x 为红，直接删除；若 x 为黑需要调整：

- Case1：x 的兄弟 w 为红
  i. w 变黑，p[x] 变红
  ii. 对 p[x] 左旋
- Case2：w 为黑且 w 的两个孩子为黑
  i. w 变红
  ii. x 上移至 p[x]
- Case3：w 为黑且 w 的左孩子红、右孩子黑
  i. w 的左孩子变黑
  ii. w 变红
  iii. 对 w 右旋
- Case4：w 为黑且 w 的右孩子红
  i. w 继承 p[x] 颜色
  ii. p[x] 变黑，w 的右孩子变黑
  iii. 对 p[x] 左旋

4. B 树是为磁盘存储设计的多路平衡树，m 阶 B 树满足：

- 根节点至少有 2 个子节点（除非为叶子）
- 内部节点有 $\lceil m/2 \rceil$ 到 m 个子节点
- 所有叶子节点位于同一层

具体操作步骤：

1. **插入操作**：

(a) 搜索到合适的叶子节点插入键值

(b) 若叶子未满（键数 $< m-1$），直接插入

(c) 若叶子已满：

- 将节点分裂为两个，各含 $\lfloor m/2 \rfloor$ 个键
- 中间键上移至父节点
- 若父节点满，递归分裂直至根节点

2. **删除操作**：

(a) 定位待删除键：

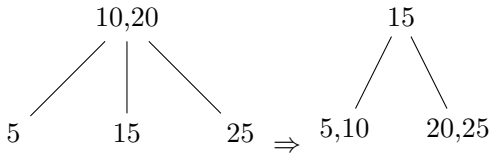
- 若在内部节点，用后继键替换后删除后继

(b) 从叶子节点删除键：

- 若键数 $\geq \lceil m/2 \rceil$，直接删除
- 否则向兄弟节点借键：
  - 若兄弟有富余键，通过父节点转移
  - 否则与兄弟合并，调整父节点

(c) 若合并导致父节点下溢，递归向上调整

示例：3 阶 B 树插入键 15 导致分裂：



4 实验结果与分析

4.1 实验环境

- 编译器：g++ 14.2.0
- 处理器：Intel(R) Core(TM) 5 220H
- 操作系统：Windows 11
- 内存：16GB DDR4

4.2 实验结果

见图 1等和表 1等

数据规模	BST	AVL 树	红黑树	5 阶 B 树
5 万	35	50	17	59
10 万	89	158	43	149
20 万	244	273	73	323
50 万	2236	822	139	862
100 万	10580	3305	470	3321
200 万	55787	8927	1144	9637

表 1: 不同规模下四种树结构的建立时间比较（单位：ms）

数据规模	BST	AVL 树	红黑树	5 阶 B 树
5 万	2	2	1	3
10 万	4	3	4	3
20 万	4	12	4	3
50 万	4	3	3	3
100 万	2	2	6	8
200 万	3	3	10	4

表 2: 不同规模下四种树结构的搜索时间比较（单位：μs）

4.3 实验分析与结论

1. **插入性能**：从表 1与图 1可见：

   - 红黑树在所有规模下插入性能最优，特别是在 $n = 2 \times 10^6$ 时比 AVL 快 7.8 倍，比 BST 快 48.8 倍
   - BST 在小数据量 ($n = 5 \times 10^4$) 时表现尚可，但随着数据量增大性能急剧下降，$n = 2 \times 10^6$ 时比红黑树慢 48.8 倍
   - AVL 树和 B 树的插入性能相近，但都不及红黑树
   - 插入 n 个数据的时间复杂度相当于对 $O(\log i)$ 从 $i = 1$ 到 $i = n$ 求和，这接近 $O(n \log n)$，与所得结果相符。

数据规模	BST	AVL 树	红黑树	5 阶 B 树
5 万	3	17	1	53
10 万	11	11	2	23
20 万	46	17	3	40
50 万	19	14	1	48
100 万	89	13	2	71
200 万	139	25	2	43

表 3: 不同规模下四种树结构的删除时间比较（单位：μs）

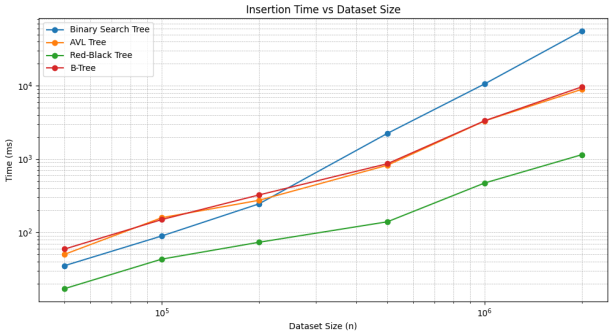


图 1: 四种树结构的建立时间比较示意图

图 2: 四种树结构的查找时间比较示意图

图 3: 四种树结构的删除时间比较示意图

2. **搜索性能**：从表 2与图 2可见：

   - 所有树结构的搜索时间都保持微秒级别，验证了 $O(\log n)$ 的时间复杂度
   - BST 和 AVL 树的搜索性能最优，红黑树和 B 树在大数据量时搜索时间略有增加
   - 在 $n = 1 \times 10^6$ 时，红黑树搜索时间 (6μs) 比 AVL 树 (2μs) 慢 3 倍
   - 搜索的时间取决于节点深度，具有较强的随机性，因此所得实验数据波动较大；若想获得稳定的实验结果，需要数十次重复实验，取平均值。

3. **删除性能**：从表 3与图 3可见：

   - 红黑树的删除性能显著优于其他结构，在 $n = 2 \times 10^6$ 时仅需 2μs
   - B 树的删除时间相对较长，与其复杂的节点调整操作有关

4. **综合比较**：

   - 红黑树在插入和删除操作上表现最优，搜索性能稍逊于 AVL 树
   - AVL 树搜索性能最优，但插入删除开销较大
   - B 树适合大规模数据，能有效减小树深，减少读取外存的 IO 次数和访问开销
   - BST 仅适用于小规模随机数据