AVLTree 和 SplayTree 排序效率比较设计思路和测试说明

November 14, 2021

1. 项目设计思路:

    1. BalanceTree.h的实现:

        I BalanceTree.h内包含AVLTree和SplayTree两个模板类。其中AVLTree的框架摘自课本4.4节的AVL树的代码。此外我改写了printTree()函数，使得main.cpp调用该函数后即可将排序完成的数据通过中序遍历存储在vector中。**为了排序时能更好的控制变量**，我将SplayTree类的框架大致与AVLTree类保持一致。二者主要的区别是AVLTree和SplayTree分别使用了balance和splay函数。其中splay函数实现了自底向上伸展的过程。与书中思路略有不同的是，本函数是从祖父的角度对各种情形进行考虑的，即从路径上自底向上对存在子节点和孙节点的所有节点进行类似于书上的判断。此外，insert函数在递归插入数据的同时也间接给出了插入路径。这些细节帮助我既可以避免通过堆栈或者新增一个记录父节点的变量来回溯，也防止了指针复杂运算的情况发生。

    2. main.cpp的实现:

        I **数据的打乱**：用户输入进行排序的元素个数N后，程序将自动生成一个大小为N的 vector<TN> DATA。由于本程序使用的具体的模板类型是Int,因此，为了方便验证排序的结果,打乱前后的DATA将恰好含有[0,N-1]中的全部整数(这样排序好的数据恰好是0,1, ...N-1 ，**可以方便我们的检验**)。具体的实现方式是，对于[0, N-1]中的全部整数，给其一个随机的下标p。如果vector[p]已经被赋值，则向下寻找至第一个未赋值的单元并给其赋值。

        II **Randomized_BT_sort()和排序效率比较的实现**：根据项目要求，我们首先对DATA中的数据进行多次打乱。然后，将每次打乱的数据分别用AVLTree和SplayTree两种数据结构进行BST排

序。并分别记录两种数据结构排序所用的总时间，从而进行比较。

2. 测试说明：

   1. 测试输入：

      I 运行 make 命令，自动编译 main.cpp 并生成可执行文件 test；

      II 运行 bash run 命令，**在同一行后输入一个正整数N，表示进行排序的元素的个数**。

      III **注意事项：建议N的值不要太小也不要太大**。如果N的值太小，排序的偶然性会较大，不宜体现平均水平；如果N的值过大，程序运行会十分缓慢。**建议的N的值是50-50000。**

   2. 测试输出：

      I 测试输出分别用AVLTree和SplayTree进行排序后的数组和排序所用总时间。具体在输出中都有详细的说明。

3. 测试结论：

   1. 经验证，无论N取什么值，输出的排序后的DATA都是0,1,...N-1，这说明**用AVLTree和SplayTree进行排序的算法都是正确的**。

   2. 对于两种排序所用的总时间T，经多次测试得如下平均结果(部分值)：

N	T(AVLTree)/ms	T(SplayTree)/ms
100	1.7	2.4
1000	35	52
10000	395	578
100000	6100	8950

   纵向比较，我们测试出发现当N较小(<500)或者过大(>10000)时，T(N)的增长速率要略大于(N log N)。**对此在上一次作业中已经有了详细的解释，是printTree()中的push_back()操作导致的。**

   我们更关心的是横向比较。结果表明，虽然两种数据结构排序用时增长速率是一致的，这也符合书中对此的分析，但是**对于相同的N，SplayTree的用时总是AVLTree的约1.5倍**。对此的解释是，针对此排序算法，每次进行insert操作时，AVLTree只会对自底向上第一个不平衡的节点进行旋转操作，而SplayTree为了把当前插入的数据自底向上旋转至根节点，会对路径上每一个节点都进行旋转。因此，SplayTree的展开操作总是比AVLTree的平衡操作更耗时。**空间方面**，在本程序尽可能控制变量的基础上，两种数据结

构的空间上的区别是，AVLTree需要对节点增加一个变量来记录录高度，这当数据量相当庞大的时候也是一个不小的开销。综上，可以得出结论：**就本排序算法而言，时间效率AVLTree占优，空间效率SplayTree占优。**