1 remove 程序的设计思路

具体代码实现参见 BST.h 中的 line 183-249 部分，篇幅所限，这里不附代码。

- 首先，作业要求中 “一个没有左右儿子的节点高度为 1”，即树叶高度为 1；实际和课本定义下，所有树叶的高都是 0 有所不同。因此对 height 函数略做修改，参 Line 254 部分。
- 为了实现 remove 函数，需要 detachMin 函数用于找到右子树的最小节点。如果使用 while 语句，则没法在 detachMin 的同时，对树进行平衡操作，尤其因为需要进行从树叶到根的平衡操作，会比较复杂。因此使用递归实现 detachMin。
- 通过递归调用 detachMin 函数，并在调用前进行 balance 操作，即相当于，在 remove 有两个子节点的节点情况下，可以在 remove 的同时，实现对树进行平衡操作。与课本代码的逻辑是一致的，相对也比较简洁。
- 在 remove 函数中，也采用递归查找的策略，并在递归后进行 balance 操作。就相当于对从被删除节点到树根，进行了回溯的平衡。
- 另外，在 detachMin 中和 remove 中，采用了例如t=t->right;//直接用右子节点替换，进行指针的替换，可以避免对于该节点的父节点的考虑，更加简洁便捷。

2 测试结果

测试结果正常。应该由于随机数生成的影响，运行时间在 2.2-2.5s 之间有所波动，但均符合要求且未出错。同时，经 valgrind 检测，未发现内存泄漏。报告如图 1所示。

```
==17551== HEAP SUMMARY:
==17551==     in use at exit: 0 bytes in 0 blocks
==17551==   total heap usage: 227 allocs, 227 frees, 21,159 bytes allocated
==17551==
==17551== All heap blocks were freed -- no leaks are possible
```

图 1: valgrind 检测结果

3 补充

我原本撰写了另一份的实现代码，参见 BST 复杂分情况讨论版.h 的 Line 210-452 部分。

- detachMin 函数实际借用 findMin 函数，同时用寻亲操作进行善后。
- remove 函数中分情况讨论，同时不用递归策略，利用自定义的 find 函数直接查找节点。
- 由于上述两种操作，因此无法直接用伴随递归的平衡操作。因此需要进行高度维护和回溯平衡。
- 因此，自己抽象了 upgradeHeight 的高度修改函数，目的是删除节点后，从根到树叶进行逐个高度修改。
- 以及回溯性的平衡函数 retrospective_balance,从树叶到根逐个进行平衡,回溯的过程调用自己集成的 find_parent 寻亲函数，逐层向上。
- 在此基础上，在 remove 函数的实现中，对被删节点的各种情况分类讨论，并具体进行不同的高度维护、平衡操作。

但出现了复杂的段错误，且始终未能解决。应当还是分情况讨论的策略过于复杂，同时必须采用大量自定义的指针操作函数。因此，一方面水平不足，同时时间有限，只能放弃这种实现策略。
全面覆写程序，在递归的同时自动进行高度的维护以及平衡操作，实现明显简单了很多。