1 remove 函数实现的阐述

1. 首先，若对象是空树，则直接 return。其次，若对象比当前节点小，则继续向左找，比当前大，向右找。找到对象后，如果该待删除对象只有一侧子树甚至没有子树，都很容易实现，按照原版代码即可（直接把子树接管权交给父辈，然后删除该节点即可）。我们只关心当待删除对象的左右子树均非空时的情况。

2. 此时，需要先把它的继任者找出来，这个找继任者的操作就在 detachMin 里完成。detachMin 负责传回将继任者找到并且无痕地取出来，由于继任者要求是右子树的最小者，一路往左就行。找到继任者后，要将继任者从树上剥离下来，但继任者可能也有子辈，于是将其子辈接回断枝，即：继任者子辈和继任者父辈相接。此外，别忘了先找个指针盯着这个继任者（否则就弄丢了），该指针值也作为参数传出去，被 remove 函数中的变量接收。

3. 现在 remove 函数拿到了剥离下来的继任者，只需要做善后工作：把删除对象的左枝右枝告诉继任者，把指向删除对象的指针改成指向继任者，然后把删除对象 delete 即可。

4. 这些结束后，在**序关系**层面已经全对了，但是在树层面还不一定平衡，下面讨论如何使树平衡。

5. 分析：由于仅删除了一个节点，而删除后，将会由其右子树的最小元素来替补它。显然，修改后的树的节点里，height 参数需要变更的那些节点，必仅存在于被删除节点以下。CASE1：继任者没有子节点，此时，仅需继任者改变 height 参数；CASE2：继任者有子节点，根据继任者的定义，仅有右子树，而右子树是作为一个整体接到继任者本来的父辈身上的，因此，右子树中的所有节点的 height 参数无需改变，只用考察继任者本身开始往上，那些 height 参数需改变，因此，在 detachMin 递归的途中，调用 balance 即可。

6. 在 remove 函数处调用 balance 函数，传入的参数是当前节点的指针引用，目的是把 remove 后的该层进行 balance，因为该层以下的 balance 问题，已经在 detachMin 函数中解决了。

2 代码测试

测试环境：12th Gen Intel(R) Core(TM) i5-12500H 2.50 GHz，在 WSL 中进行测试
首先需要**接除限制**：ulimit -s unlimited
可以直接执行 make run：（即沿袭上次作业，执行 g++ test.cpp -o test -O2 以及 time ./test）
也可以分开执行，然后 time ./test 的输出如下（省略前面打印树的很长的输出）：

```
------------------------------
Empty tree


real    0m0.942s
user    0m0.932s
sys     0m0.010s
```

我用 valgrind –leak-check=full ./test 进行测试，发现没有发生内存泄露。