BBST实验报告

2021201645 赵宸

本次lab，我基本实现了既定的基本功能和附加题目，在luogu上找到对拍程序完成若干次对拍之后，没有发现问题。对拍实况如下图所示：

3369.bat

```
@echo off
:loop1
    3369rand.exe > data.in
    AVL.exe < data.in > AVL.out
    3369model.exe < data.in > std.out
    fc AVL.out std.out
if not errorlevel 1 goto loop2
```

问题 输出 调试控制台 终端 Batch Runner Terminal

```
正在比较文件 RBtree.out 和 STD.OUT
FC: 找不到差异

正在比较文件 AVL.out 和 STD.OUT
FC: 找不到差异

正在比较文件 RBtree.out 和 STD.OUT
FC: 找不到差异

正在比较文件 AVL.out 和 STD.OUT
FC: 找不到差异

正在比较文件 RBtree.out 和 STD.OUT
FC: 找不到差异
```

行 8，列 6 空格: 4 UTF-8 CRLF Batch

3391.bat

```
@echo off
:loop
    3391rand.exe > data.in
    splay.exe < data.in > splay.out
    3391model.exe < data.in > std.out
    fc splay.out std.out
if not errorlevel 1 goto loop
```

问题 输出 调试控制台 终端 Batch Runner Terminal

```
FC: 找不到差异

正在比较文件 splay.out 和 STD.OUT
FC: 找不到差异

正在比较文件 splay.out 和 STD.OUT
FC: 找不到差异

正在比较文件 splay.out 和 STD.OUT
FC: 找不到差异

正在比较文件 splay.out 和 STD.OUT
FC: 找不到差异

正在比较文件 splay.out 和 STD.OUT
```

行 4，列 36 空格: 4 UTF-8 CRLF Batch

第一题我挑选了AVL树和RBtree实现,第二题则采用了splay+treap的思想完成.

AVL树

由于非常反感邓俊辉的代码风格,因此我没有采用课本上的代码,而是重新在CSDN上找了个完整的板子,结果发现...板子抄完题目就做的差不多了(?).然后又加了一点点特殊输入的特判,就构成了我的这段代码.
由于是AVL树,代码的核心逻辑还是落在balance这棵树上.而balance的方法又分为左旋和右旋.对于每一个节点,如果左子树的高度高于右子树+1,那就左旋,反之则右旋.由于每次插入删除都检查,因此实际差异不会超过2.
至于插入和删除,就和二叉搜索树基本一致,不同的是要加一个沿着父亲向上的平衡过程.因此这里就要涉及zig_zag和zag_zig的问题,这里就不多赘述了,其实把这个过程拆成两步走也完全可行,只不过是追求代码简洁性而已.
在完成模板树的构建之后,题目就变得非常简单了.3.4就是和二叉搜索树一模一样的search过程,而五六就是一个非常典型的查找前驱后继的问题,这些东西在二叉平衡树甚至二叉树那里都曾经实现过,这里就不展开了.

RBtree

红黑树,上课的时候就感觉这个东西不简单,一动笔他还真是个大哥.难度最大的部分就是貌似很简单的双红和双黑修正问题.容我后边慢慢道来.
关于这个数据结构,我们的课本也和没有一样,完全没有任何参考价值,没有办法,我有在网上开始找红黑树的模板.您猜怎么着,我找了一圈,发现写的最好的竟然还是邓俊辉的代码.然后我就把他的模板搞下来,自己添加了一个双黑修正,最终...也能用.
红黑树的单纯插入删除操作比AVL树要简单,因为他不需要考虑旋转,红黑的特性保证了他的树结构是局部正则的.但要维护这种局部正则性,我们就必须时刻修正红黑的关系.插入和删除基本同上,下边步入正题.
当我们插入的时候,为了保证总树高不变,我们必须插入红节点,但两个红节点不能相邻,因此我们就需要对双红的情况进行修正.由于插入之前,树是符合RBtree的规则的,因此红parent就意味着它必然有一个黑grandparent.但uncle的情况是不确定的.倘若他的uncle是红的,那我们直接把黑色特征从grandparent那里继承下来,然后把grandparent染红即可.但此时,grandparent又有可能触发他和她parent的双红问题,因此需要递归判断;倘若uncle是黑色的,那我们就把parent转到grandparents的位置,然后让grandparent下来和x当兄弟.此时调整随即完成.可见双红修正条件少,旋转简单,还是比较好处理的.
最难的还属后边这位:双黑修正.当我们删除一个节点的时候,如果他是个"顶梁柱"黑节点,那删除他就意味着整体高度要-1,除非他父亲是红的可以染黑.倘若他父亲也是黑的,那这个问题就需要向上延伸了.首先如果x的uncle是黑色的,那我们就讨论他孩子的情况.倘若孩子有红色,那就对uncle的孩子\uncle\grandparent做两次旋转,让他们仨挑大梁;否则那就要再依托于他的父亲了,如果父亲是红的,那我们直接染黑父亲染红uncle和parent,这样就能光明正大删掉啦;否则那就是全黑的情况,既然没有回旋的余地那就只能减高度了,强行把他的uncle变红,然后告诉grandparent的parent,我们高度减1啦,你们看着办!然后压力来到grandparent这边(不是).到这儿,才讨论了一半,但是是困难的一半.剩下的就是uncle是红的情况,既然uncle为红,那我直接右旋,让uncle当grandparent,grandparent下来当parent,层层递减补缺口.此时无论如何,新树的x的parent一定是红色的了,如果新parent接手的uncle有红孩子,那就转!如果没有,那就染!这里就和前边的讨论完全重合了.
就这样,红黑树的修正就讨论完了,剩下的问题也就非常简单了.比AVL的情况还要简单(?).

附加题——splay

这个题既然没有方法限制,那自然是选择代码量最小的splay啦.splay的简单之处在于不必维护树的完全性,只需要额外实现一个splay函数用于把刚刚访问的点转到顶点即可.这个函数我采用了课本上讲的双层旋转,这样能够保证程序的稳定性.另外,由于上边两题debug过于复杂,这次我就采用了一个没有用指针的板子,转而用数字+数组索引的方式(后来才知道这原来叫treap,感觉写起来要麻烦,但debug容易)
关于splay函数,核心思想是每次search一个点,我都找到这个点的父亲先旋转,然后他再转,直到它或者他父亲登顶为止.剩下的完全就是一个普通的二叉树思想,至于旋转的过程,我采用的是从上到下传递信息的方式,由于起点已

经登顶了,而终点一定在下边,起点到终点的所有点都要做一次类似冒泡排序的左右孩子交换,这样可以把每次信息传递的成本降到logn.其他的也就没有什么好说的了.

写在最后

首先谴责一下助教,这个lab真的没有助教师兄说的那么"水",BBST的debug真的非常非常非常麻烦!有时候一个细节错了(比如if前边漏一个else)就能de一天,尤其是后期打对拍,好几万的数据点找到底是哪里出了问题...怪不得我的某个信竞同学说他对树PTSD了().

work cites

- AVL树模板
- RBtree模拟
- RBtree模板
- splay模板