数据结构课程项目
线性表

杜雨轩 19307130196

2020.12.20

“线性表”

- std::deque在绝大多数要求上均有很好的时空复杂度优势，但是在insert及erase两项表现很不好，最坏、平均情况均是O（n）的时间复杂度。
- std::list在这两项的表现十分出色，都是O（1）的时空复杂度，但是，在随机访问方面，最坏、平均情况却是O（n），这也是我们我们难以接受的。
- 有没有什么东西能把两者的优势结合起来，可以在至少均摊常数的情况下完成随机访问，自由插入与删除？

我不知道（或许不存在吧）

但是我们可以用O(log n)的数据结构

- 平衡树
  - AVL
  - RBT
  - SPLAY
  - TREAP
  - ……
- 线段树
  - ……
- 跳表
  - ……

我选择RBT（红黑树）

为什么是RBT?

- ~~代码量大，有成就感~~
- ~~难以调试，惊险刺激，锻炼心态~~
- ~~pre可以做得长一些~~

- std::map多是RBT，有相当权威~~且晦涩~~的参考资料，特别是迭代器的设计问题
- RBT的常数相对较少，跑起来会快一些
- ~~我一时糊涂~~

以下正文

节点设计

RBT首先是一种二叉树搜索树，所以节点与BST的节点设计类似。

- 节点拥有的三根指针分别指向父母，左右子女
- val是该节点的值，size是该节点之下的节点个数（包括该节点自身）

- color代表节点颜色，是RBT特有的节点附加域
- 在RBT平衡在起到关键作用

RBT的数据结构

- 数据段只有两个，一个节点指针与容器的大小。
- 其余是各种typedef及函数定义
- 下面谈谈为什么只有一个header

header

- header是一个巧妙的设计，我是在std::map中了解到这种写法的。

```
link_type& root() const { return (link_type&)header->father; }
link_type& leftmost() const { return (link_type&)header->children[0]; }
link_type& rightmost() const { return (link_type&)header->children[1]; }
```

- 可以看出，header储存了根节点指针，最小节点，最大节点
- 同时，header还充当了end节点的功能
- header是红节点，与Root（）恒为黑节点形成区别

- 至此，一些简单的函数已经可以开始写了
- （关于header的维护详见后文）

以下函数

我觉得你们不需要看具体代码吧,
所以我把原本的代码图片全删了

构造函数 析构函数

- 默认构造函数
- 复制构造函数
- 等号重载
- 移动构造函数

- 析构函数

- 比较细节一点的是等号重载（比复制构造函数有更多细节）

等号重载

- 有一个重要的细节，检查等号左右是否是同一个容器，没有检查的话会有潜在的严重错误：）

- 然后分类讨论，更新header
- 关键:
- 深拷贝部分由
- 函数 **__copy** 完成

__copy

- 可以暴力递归
- 也可以沿左子女链，递归右子女进行深拷贝。
- 或许可以模拟一个非递归

begin end front back

- 常数复杂度
- header的左子女就是front（begin），右子女就是back
- header本身就是end
- 此处可以看见header的设计，使得数据结构的数据域相当的精简

BST基础操作

- RBT首先是BST，所以支持BST的基础操作
- 众所周知，有一道“普通平衡树”的板题，告诉我们平衡树可以支持
- 插入
- 删除
- 查排名 （暂无对应）
- 第k大 （对应 [ ],随机访问）
- 前驱（对应 ++iterator）
- 后继（对应 --iterator）
- 注：红字对应操作没有足够的泛用性，依数据结构而异

伪·随机访问

- 时间复杂度O（log n）,与O（1）有差距但不会太大
- 利用的是BST的kth，根据size域判断目标位置，从根节点出发至目标节点

前驱与后继

- 迭代器相关
- 迭代器设计先按下不表
- 根据对称性，前驱与后继的实现也是对称的（end（）的前驱需要特判）

- 主要思路：先在相应子女找，如果没有就往父母域找
- 因为遍历RBT为O（n），因此均摊复杂度为常数
- 表内数据足够多，RBT的高度是log n级的
- 因此单次最坏情况为O（log n）

RBT的插入与删除

相比TREAP、SPLAY，RBT的插入与删除的逻辑相当严谨

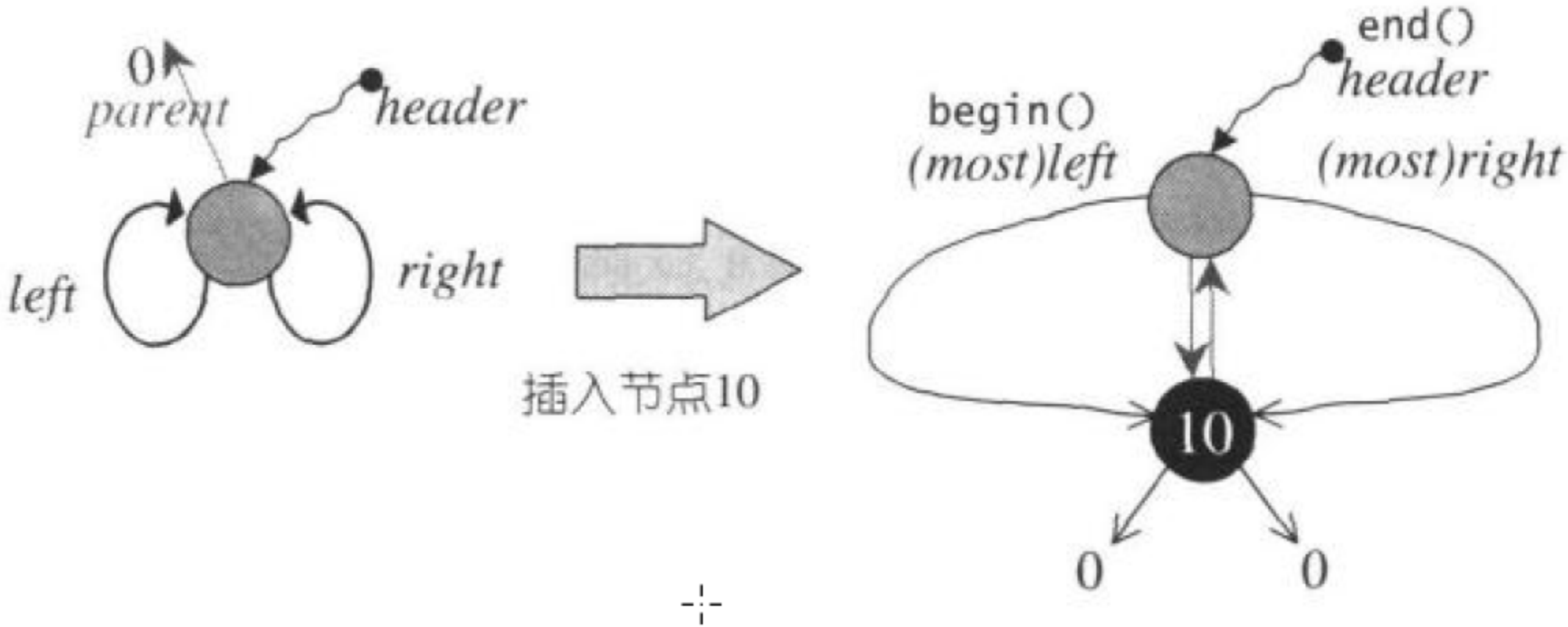
- 重要的原则
- A.叶子节点是黑节点
- Ps.一般的操作是，增加1层黑色子女节点，即 nullptr 为黑色节点
- B.叶子节点到根节点的路径上黑色节点个数相等（称为“黑高度”）
- C.红色节点的子女节点为黑节点
- B，C保证了树的高度不超过黑高度的2倍，保证平衡性

RBT的插入——寻找插入点

- 针对数据插入空表特判一下

- 然后是确定插入位置
- 标准RBT的是根据compare的比较确定位置的
- 这个不一样，我现在逐一讲
- push_back:应该沿右子女链走到底
- push_front:应该沿左儿子链走到底
- insert，由于插入在给定迭代器前，所以检查是否有左子女，有就插入在左子女处，否则插入在左儿子的右儿子链尽头

插入空表——header的妙处



注意，header和 root 互为对方的父节点，这是一种实现技巧

一帆风顺的插入？

- 很好，新建节点，写值，连好父母、子女，更新size，就连全局的leftmost、rightmost都更新好了，完成了？
- 等等RBT的红黑呢？
- 我们的插入好像会造成RBT原则的破坏
- 所以我们需要插入的修正“双红修正”

InsertFix——Rotate操作

- 关键方法是旋转 左旋/右旋
- 具体操作类似课上AVL的选择操作
- 值得注意的是我们要维护父母指针，左右子女指针

- 因此需要理好逻辑

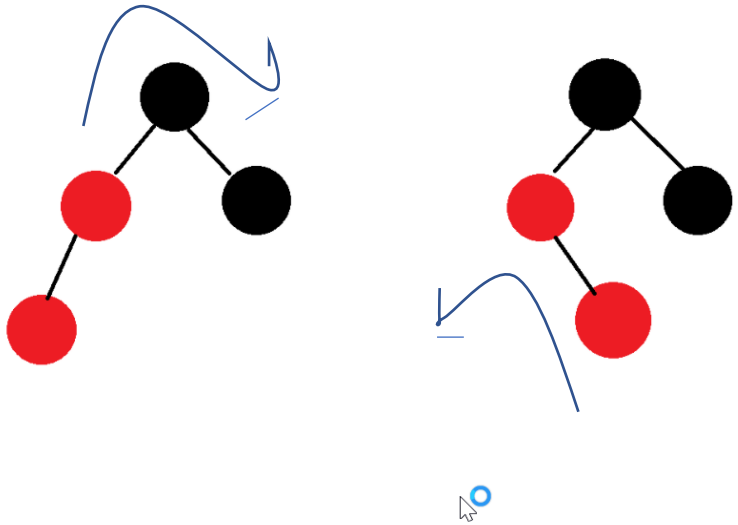
- 由于对称性，我们只需要控制一个布尔变量就可以实现左右旋转

InsertFix——分类讨论

- 常规操作将插入节点初定为红节点
- 插入点的父亲是黑节点——插入后不影响黑高度
- 否则，根据叔父节点的颜色进行讨论

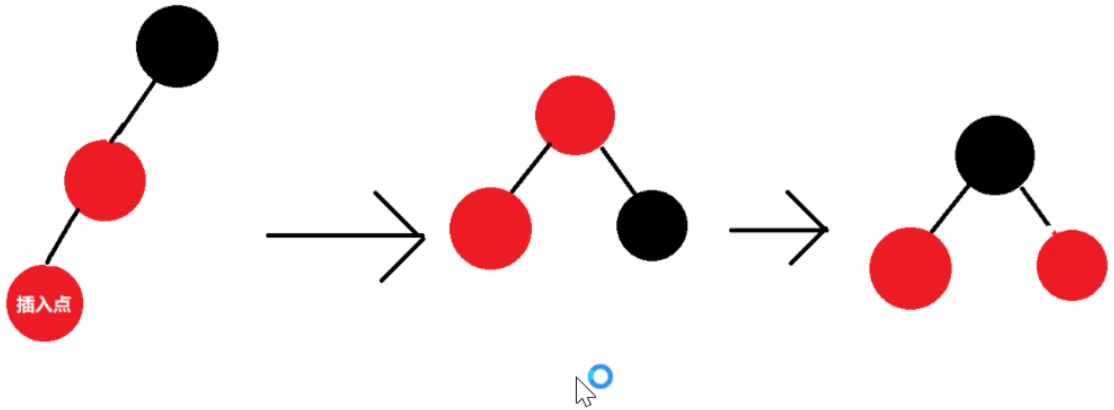
叔父节点为黑节点

- 有两种情况
- 第三层为新插入点
- 右图只需要左旋就可以变成左图



- 左图可以通过先右旋再变色

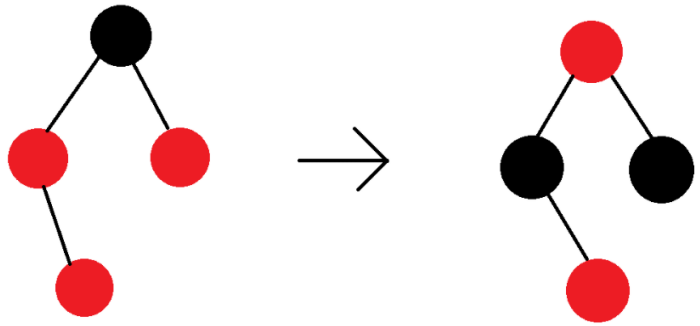
实现平衡



叔父节点为红节点

- 通过变色进行向上传递
- 相当于原本的祖父节点变成了新插入的节点

- 递归/循环



- 终止条件：到达根节点/父母是黑色节点

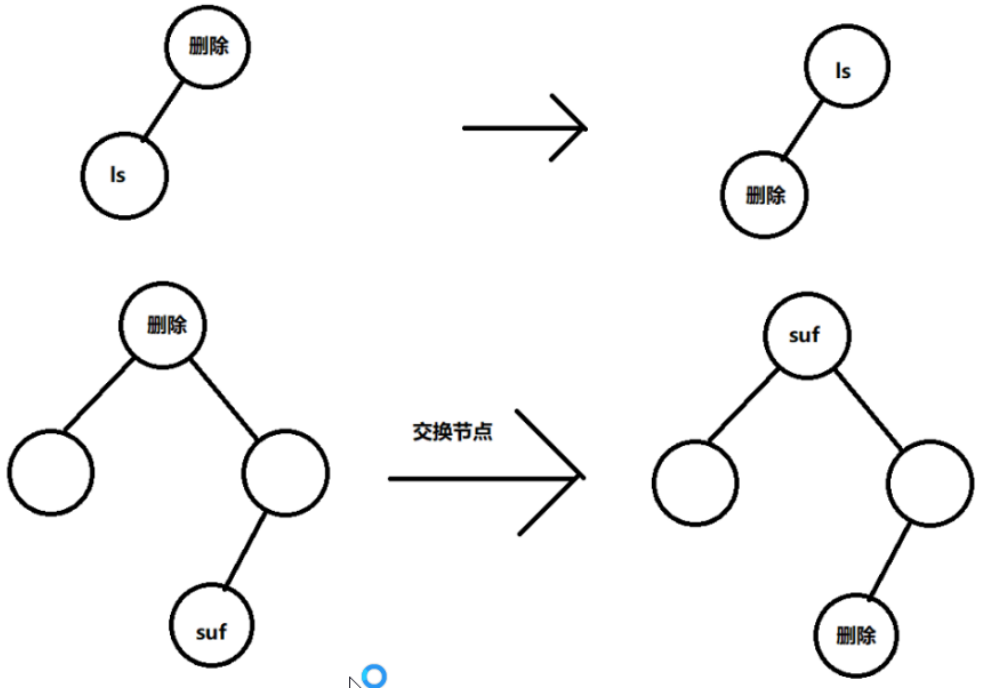
路阻且长的删除

- 删除节点比插入要更复杂
- 要删除的节点不一定是叶子

- 我们可以通过一些交换或连接使要删除的节点变成叶子节点
- 这样子可能会破坏RBT的一些原则，但是我们可以后续修正

寻找“补位节点”

- 具体方法是，如果只有一个子女节点，那么就把子女节点接到要删除的节点的位置；否则，我们也可以在右子树中找最小值与要删除的节点（即后继）交换
- 交换时记得处理好父母，子女域的指针关系，特别要注意要将后继的右子树接到后继的父母节点的左子女处
- 以及更新leftmost及rightmost，size



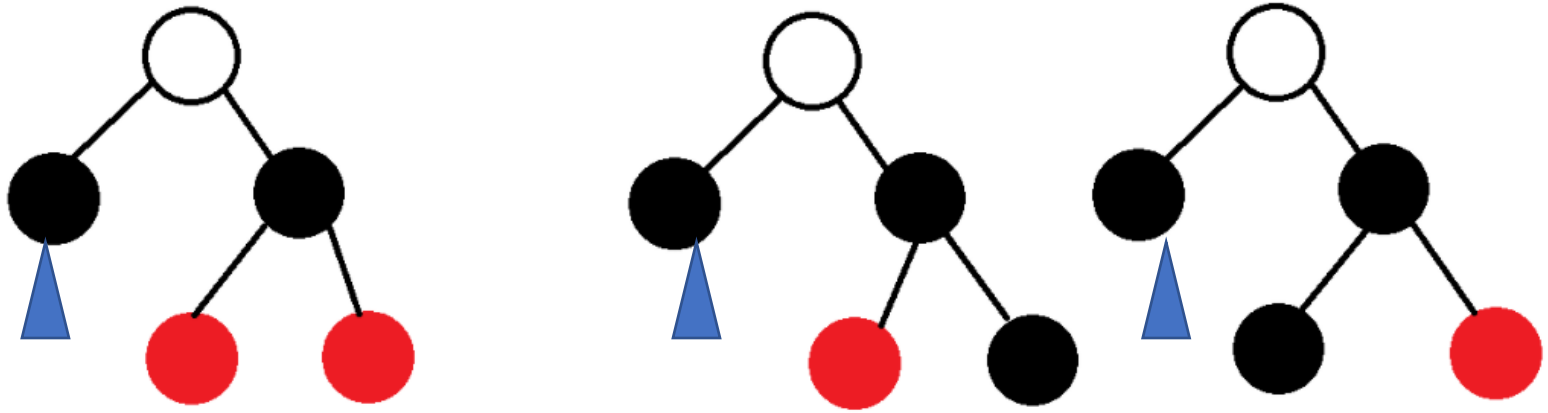
至于删除位置

- front和back和插入同理
- erase就是需要直接提供需要删除的迭代器的嘛

DeleteFix——分类讨论

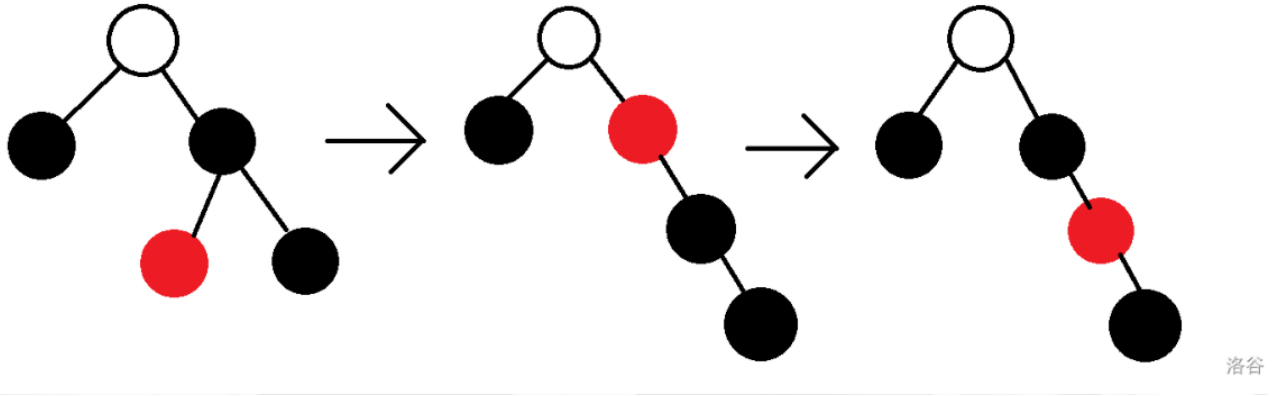
- 删红节点是不用Fix的，好耶！

- 情况一，被删节点父亲是黑色节点，叔父节点是黑色节点且有红色子女节点（除图二外，叔父的右子女都是红节点！）

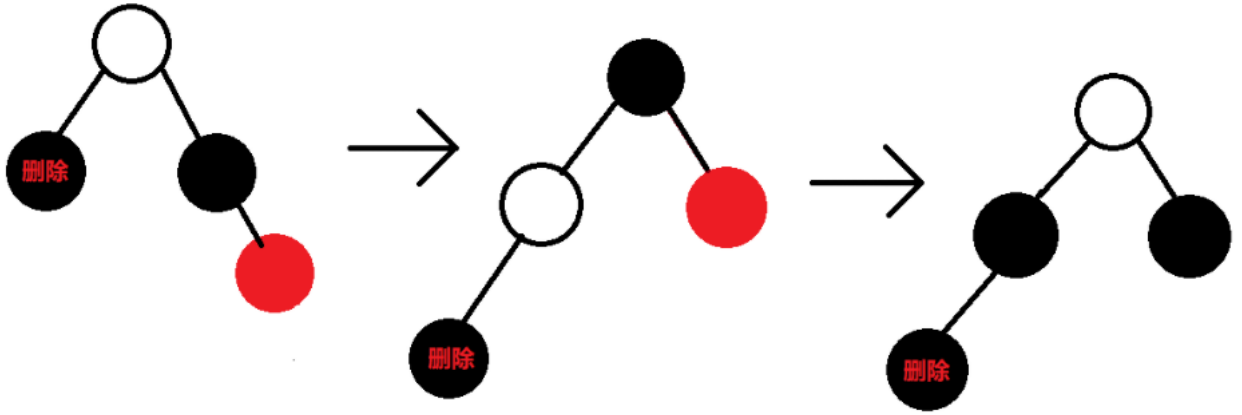


情况一·续

- 对于图二，我们可以通过右旋加变色来调整

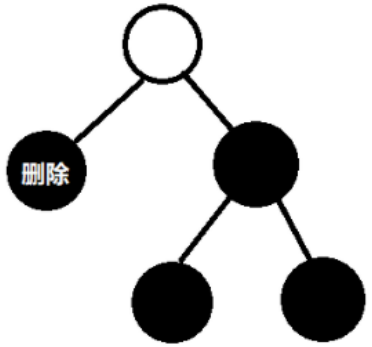


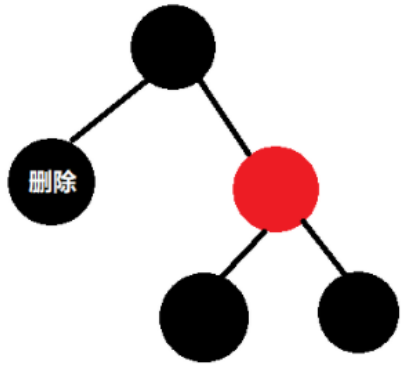
- 调整完后，我们的叔父的右儿子是红色节点了
- “删除”指被删节点在该子树上



情况二

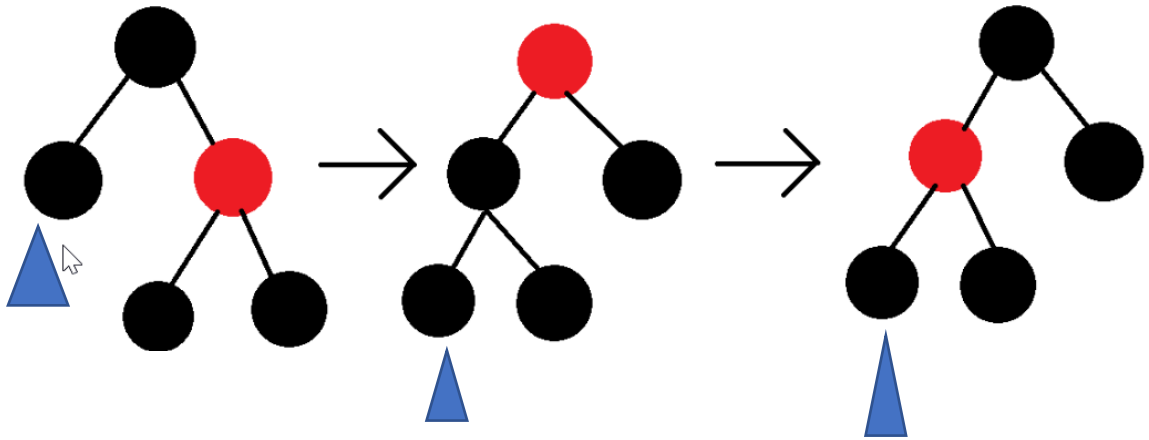
- 若叔父节点的子女全为黑色节点（nullptr也是黑色节点），有2种小情况。





- 对于左图，叔父节点变成红色即可
- 对于右图，可以通过旋转变色将问题向下传递

这时叔父节点是黑色了



简单函数一笔带过

- size
- empty
- swap——交换数据域的指针
- clear——dfs清空

迭代器

似是而非

伪·随机访问迭代器

- 我的设计支持了

遗留随机访问迭代器(LegacyRandomAccessIterator)

的所有表达式操作，包括

+=(int) / (int)+ / +(int) / -=(int)/ - / [] / < / <= / > / >= / == / !=

- 遗憾的是，前5项复杂度为log n

更大的遗憾

- 由于现代STL的复杂性，模仿stl的行为需要大量的铺垫准备工作，包括但不限于准备std::iterator_traits<It>。
- 我无法胜任这个任务，因此，我无法实现
- std::sort(a.begin(),a.end());
- 这样子的语句。

- 这项任务或许可以交给精通C++的同学完成。

常值容器

- 常值容器实现的关键在于const_iterator的设计与其支持的函数的const标记

- 这些都是相当基础的设计

- 理论上有更合适的数据结构去实现常量容器（std::array或许可以）
- 但是这个设计实现有难度
- 希望精通C++的同学可以完成这个设计。

拓展——最值、有序容器

朴素做法是最优解吗？

最值相关

- 全局：每次询问遍历一次，O（n）
- 区间k大：取出区间至vector，跑kth，O(dis(it1,it2))
- 区间内第区间大：取出区间至vector，跑sort，记dis(it1,it2)=m
- O（mlogm）

有序容器

- sort，取出所有元素至vector后调用std::sort,填回原容器
- sorted_insert,RBT常规操作
- sorted_merge,沿begin向end遍历，遇到合适位置插入

为什么要朴素做法？

- 好实现
- 稳定

- 规避维护数据的失效问题

内存池

- 这个代码使用了内存池设计
- 可以避免频繁的new和delete
- 同时实现了单个容器的可变长

参考资料

- stl_tree.h, STL, Cygnus C++ 2.91 for windows
- 《STL源码剖析》by 侯捷
- 《浅谈红黑树》by bf，https://www.luogu.com.cn/blog/bfqaq/qian-tan-hong-hei-shu

以上

谢谢