- 红黑树相关

红黑树规则

- 根节点必须为黑色
- 新添加节点为红色
- 根节点到任何一个叶子节点, 拥有相同数量的黑色节点
- 不能出现连续的红色节点

红黑树添加

1. 添加检测连续的红色节点.

如果出现连续红色节点, 检测父节点的兄弟节点颜色

1. 黑色旋转
2. 红色染色.

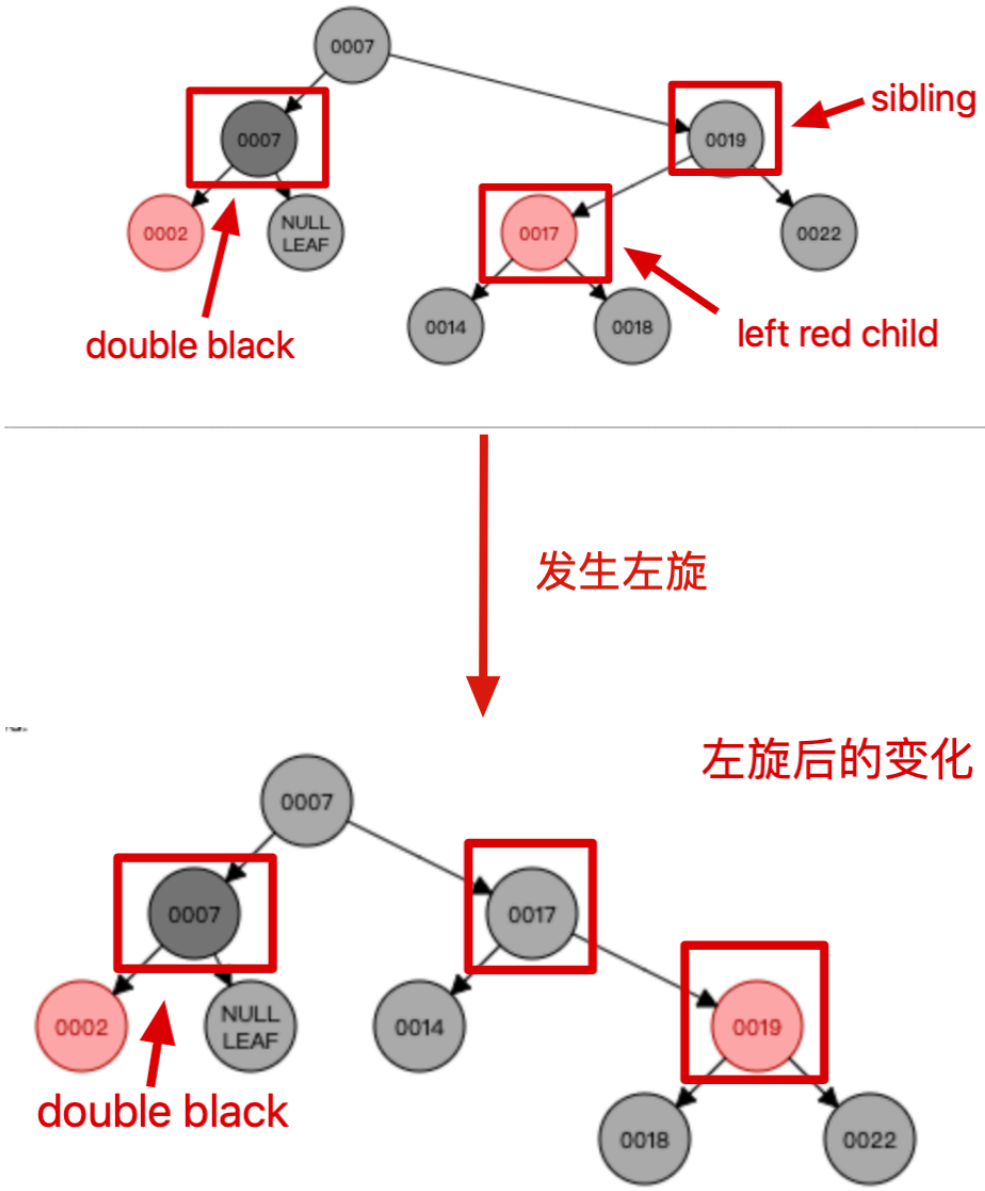
旋转与染色对象都是 grant
旋转完, 确保新的父节点是黑色, 子节点红色.

红黑色删除

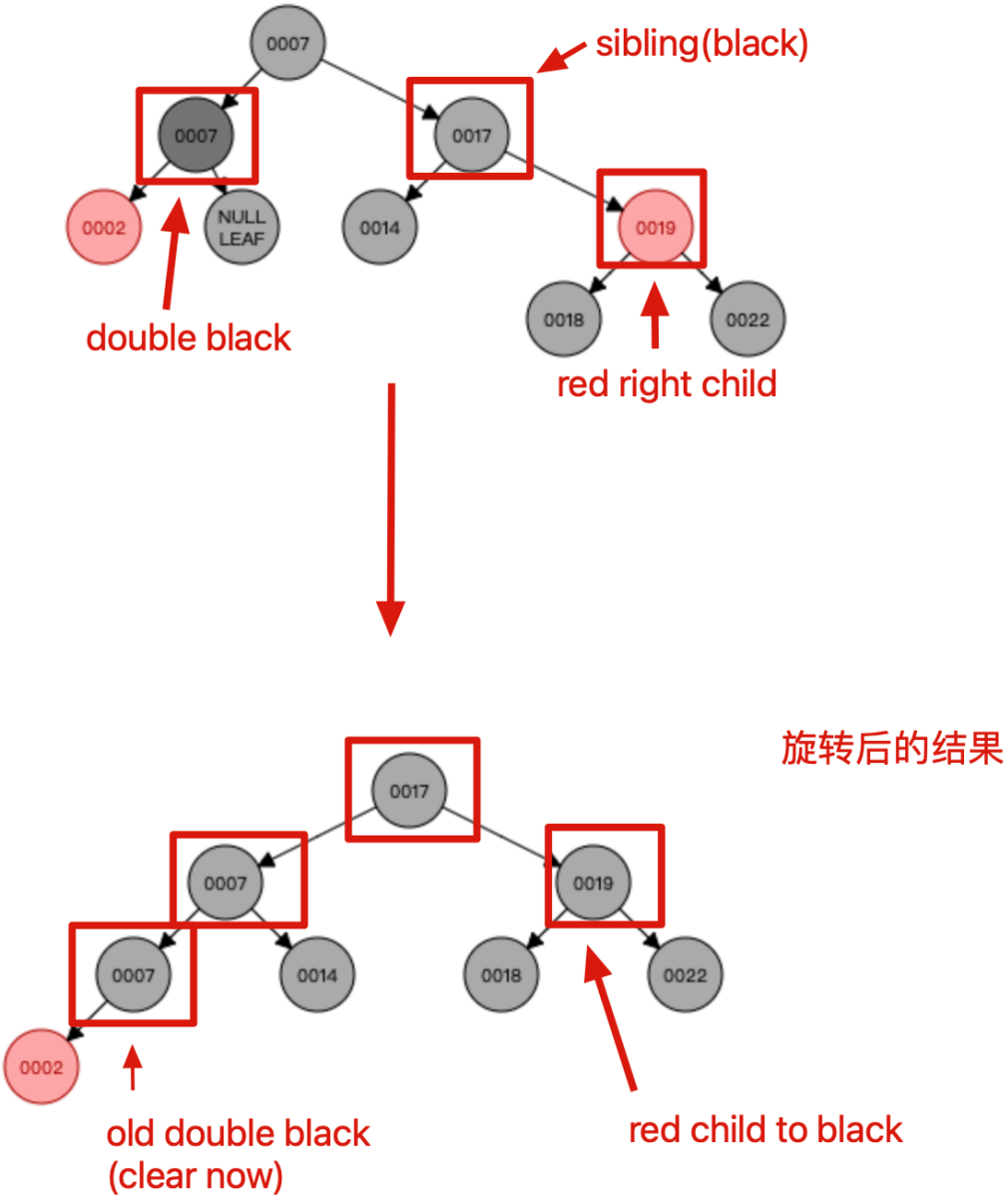
1. 节点是红色,正常删除即可
2. 检测删除节点是否为黑色.
    - 删除节点为叶子节点(没有左右子节点), 如果是黑色, 执行双黑修正. double black issue , 如果是红色, 直接删除
    - 删除节点, 如果有左右子树. 找到后继节点, 然后交互值, 删除后继节点
    - 仅有左子树, 或者右子树. 一种是, 如果 删除节点是黑色, 后继节点也是黑色, 需要修正 double black issue, 如果是一个红色, 一个黑色. 交互色值, 删除节点, 并让后续节点为黑色

Fix double black issue.

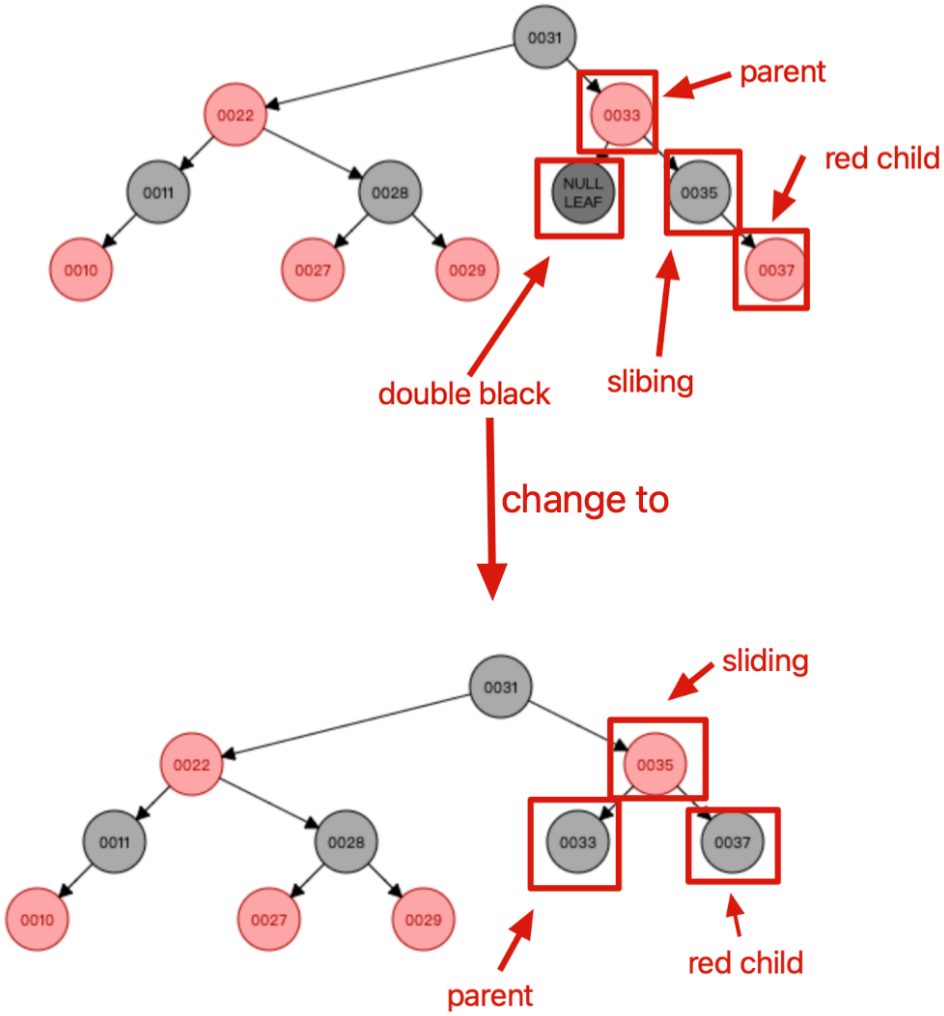
1. 如果 sibling 是 null, parent变成新的双黑节点 , 然后递归修正 node 的 double black issue
2. 如果 sibling 是红色, 旋转sibling的父节点,进行中序排序 , siling染黑色和sibling父节点染红色, 双黑节点保持不变, 然后递归修正 node 的 double black issue.
3. 如果 sibling 是黑色, 且只有一个红色节点, 这里也有两个情况:
    - 红色节点在左边,旋转sibling和sibling的红色节点(红色节点变成sliding的父节点,sliding变成了其右节点), 然后红色节点变成黑色,sibling变成红色,然后继续递归修正 node 的 double black issue



- 红色节点在右边 , 那么sibling ,sibling父节点,sibling的红色节点进行中序遍历来拉高, 然后sibling子节点染成黑色,end



4. 如果 sibling 是黑色，sibling有红色child和红色parent, 那么slibing和parent进行旋转，slibing升高，parent变成slibing的left child，然后 slibing以及它的新child们都重新染色，end：



5. 如果 sibling 是黑色，且子节也是黑色(空节点也是黑色的), 那么sibling染红色, 然后针对parent有三种情况
    - 如果parent是红色, 那么 parent 染黑色, end
    - 如果parent是根节点,parent 保持黑色即可, end
    - 如果parent是黑色，parent变成新的双黑节点 , 然后递归修正 node 的 double black issue

添加的演示:
从1 添加到10,包括10, 讲解添加的每个步骤

- 根节点必须为黑色
- 新添加节点为红色
- 不能出现连续的红色节点

如果出现连续红色节点, 检测父节点的兄弟节点颜色

1. 黑色旋转
2. 红色染色.

旋转与染色对象都是 grant
旋转完, 确保新的父节点是黑色, 子节点红色.

红黑色删除

2. 检测删除节点是否为黑色.

2.1 删除节点为叶子节点(没有左右子节点), 如果是黑色, 执行双黑修正. double black issue , 如果是红色, 直接删除
2.2 删除节点, 如果有左右子树. 找到后继节点, 然后交互值, 删除后继节点,
2.3 仅有左子树, 或者右子树. 一种是, 如果 删除节点是黑色, 后继节点也是黑色, 需要修正 double black issue, 如果是一个红色, 一个黑色. 交互值, 删除节点, 并让后续节点为黑色

Fix double black issue.

1. 如果 sRedBlackTree记录.mdibling 是 null, 递归为父节点修正平衡
2. 如果 sibling 是红色, 为siling染色 和父节点染色, 并旋转父节点, 然后递归修正 node 的 double black issue.
3. 如果 sibling 是黑色, 且, 有一个红色节点, 旋转父节点, 并且保证父节点始终是黑色
4. 如果 sibling 是黑色, 且子节也是黑色, 染色, 如果 parent 节点是黑色, 修复 double black issue.

remove: 04

1. 找到后继节点3, 交换值, 我们需要删除的节点就变成了一个叶子节点:04
2. 04叶子节点为黑色. 需要执行, fix double black issue. sibling : 001, 为黑色. 且没有红色节点, 执行染色.
3. 对04的父节点 02进行double black issue fix. 兄弟节点为6, 需要执行条件3的修正.

remove: 06

1. 有左右子树, 后继节点为: 05, 交换值.
2. 需要删除后继节点: 06, sibling: 02, 对 parent: 03 执行右旋操作.

remove: 05

1. 有左右子树, 后继节点为: 03, 交的值, 去递归删除后继节点03
2. Sibling为: 01, 它是一个叶子节点, 执行条件4, 首先对sibling 进行染色, 然后因为parent: 02是父节点, 执行修正逻辑.
3. 递归向上, 为节点02执行修正, 节点02的 sibling 节点为08而且是黑色, 且没有红色子节点. 首先执行染色. 检测父节点05,

remove: 03

1. 有左右子树, 后继节点为: 02, 交换值, 去递归删除后继节点02, 但是, 后继节点, 有左侧节点:01
   删除节点, 有一个红色节点, 让父容器对接子节点. 让子节节点为黑色

删除节点: 02

1. 有左右子树, 后继节点为: 01, 交换值, 去递归删除后继节点02.
2. 后继节点为黑色, 需要执行双黑修正, 我们的 sibling节点为红色, 值为8, 将 sibling 染色为红色, 父节点01, 染色为黑, 并为父节点执行左旋
3. 旋转完之后, 对 node 节点, 继续执行双黑修正, 这个时候, 他的新的 sibling 是07, 他是一个黑色叶子节点. 将染色为红色, 他的 parent 节点为红色, 将其他染色为黑色.

删除节点: 08

1. 找到继节点: 07, 交换值, 递归删除删除08
2. 检测到删除节点是红色, 直接删除.

删除节点: 07

1. 找到继节点: 01, 交换值, 递归删除删除07
2. 删除节点为黑色, 需要执行双黑修正, sibing 有一个红色子节点. 需要执行染色, 并旋转父节点.

删除节点: 09

1. 找到继节点: 01, 交换值, 递归删除删除09
2. sibling 节点为: 10, 是黑色, 且没有子节点. 执行染色.将自己染色成红色, 检测父节点是否为黑色, 如果是递归执行双黑 fix. 但是因为父节点是根节点, 退出.

删除节点:01

1. 只有右子树. 且右子树为红色. 将10与新的父节点关系, 检测到删除节点为根节点, 不管因为根节点, 还是因为删除节点的后继有红色子节点, 都需要将后继节点置为黑色.

删除:10
为根节点, 直接返回.

学习的看法.

1. 笨人, 总想走捷径, 并且. 认为有效. 很少有自己思考, 以及实践. 他的大部分想法, 与资料, 可能是, 21天精通 Java. 缺额改良, 理解, 深入的过程.
2. 第二类人, 可能强一点, 他有执行力, 但是不愿思考. 有执行力是指工作之外, 还肯学, 还能学. 我们在工作中, 或者在要找工作, 在某些时刻, 我们会很努力去学习. 但是, 要么学完就完事了, 要么业余可能达到某个水平, 理解, 或者看完了,就完事了. 这个过程中, 属于比较浅的学习. 写简单的自定义控件, 写了五年, 我就是不去学学, RecyclerView 怎么实现. 比如动画怎么实现的. 但是, 我天天用动画, 天天实现动画, 我天天写 RecyclerView. RedBlackTree_副本.md
3. 第三类人, 可能就是我这类人, 尝试去做这类事, 很多原理, 真正的原理也理解不了, 但是会尝试去理解, 去学习, 本着学一点是一点. 多理解一点, 可能未来就能搞出来了, 所以算是在长期的学习. 在别人眼中, 也是在往深入钻.
   比如红黑树, 比如多线程, 其实我尝试着用汇编去理解内存保护, 我还是没学会. 红黑树, 但是之前的资料留的太好了, 包括代码.; 所以我可以快速的去继续思考, 验证, 以及再次基于之前的理解, 去学习.
4. 天赋, 背景, 人还聪明. 还肯努力.