

数据结构课程设计

**2022/2023(1)**



课设题目 用户登录系统模拟

学生姓名 沈铖域

学生学号 202105070118

学生班级 计算机科学与技术2103

任课教师 潘翔

提交日期 2023.2.20

目录

**一、实验题目和要求 3**

**二、设计思路 4**

2.1 系统总体设计

2.2 系统功能设计

2.3 类的设计

2.4 主程序的设计

三、调试分析

3.1 技术难点分析

3.2 调试错误分析

四、测试结果分析

4.1 调试测试情况

4.2 总体分析

**五、实验总结**

参考文献（算法及相关实现如有参考，请列出）

1. **用户登录系统的模拟**

**【问题描述】**在登录服务器系统时，都需要验证用户名和密码，如telnet远程登录服务器。用户输入用户名和密码后，服务器程序会首先验证用户信息的合法性。由于用户信息的验证频率很高，系统有必要有效地组织这些用户信息，从而快速查找和验证用户。另外，系统也会经常会添加新用户、删除老用户和更新用户密码等操作，因此，系统必须采用动态结构，在添加、删除或更新后，依然能保证验证过程的快速。请采用相应的数据结构模拟用户登录系统，其功能要求包括用户登录、用户密码更新、用户添加和用户删除等。

**【基本要求】**

1. 要求自己编程实现二叉树结构及其相关功能，以存储用户信息，**不允许使用标准模板类的二叉树结构和函数**。同时要求根据二叉树的变化情况，进行相应的平衡操作，即AVL平衡树操作，**四种平衡操作都必须考虑**。测试时，各种情况都需要测试，并附上测试截图；
2. 要求采用类的设计思路，不允许出现类以外的函数定义，但允许友元函数。主函数中只能出现类的成员函数的调用，不允许出现对其它函数的调用。
3. 要求采用多文件方式：.h文件存储类的声明，.cpp文件存储类的实现，主函数main存储在另外一个单独的cpp文件中。如果采用类模板，则类的声明和实现都放在.h文件中。
4. **不强制要求采用类模板；不强制要求采用可视化窗口，但若采用可视化窗口可适当提高考核分数；要求源程序中有相应注释；**
5. 要求测试例子要比较详尽，各种极限情况也要考虑到，测试的输出信息要详细易懂，表明各个功能的执行正确；
6. 建议采用Visual C++ 6.0及以上版本进行调试；

【**实现提示**】

1. 用户信息(即用户名和密码)可以存储在文件中，当程序启动时，从文件中读取所有的用户信息，并建立合适的查找二叉树；
2. 验证过程时，需要根据登录的用户名，检索整个二叉树，找到匹配的用户名，进行验证；更新用户密码时，也需要检索二叉树，找到匹配项后进行更新，同时更新文件中存储的用户密码。
3. 添加用户时，不仅需要在文件中添加，也需要在二叉树中添加相应的节点；删除用户时，也是如此；
4. **用户登录系统可以联系实际，扩展相应功能。**

**【运行结果要求】**要求能够实现用户登录验证、添加用户、删除用户和更新用户密码功能，实验报告要求有详细的功能测试截图。

**【考核要求】**要求程序能正常运行，全面完成题目要求。

**【题目难度】 难，成绩等级高**

**【咨询教师】 如有问题，可直接咨询任课教师**

1. **设计思路**

分析问题，用户信息的验证频率很高，说明搜索查询的次数较多，考虑二叉树需要进行自平衡以提高查询的效率；另外，系统经常会进行添加新用户、删除老用户和更新用户密码等操作，说明数据操作的次数也比较多，在典型的自平衡二叉树，AVL树和RB树中，由于AVL的数据操作效率相对低于红黑树，从综合场景来看，红黑树更加适合。以红黑树作为主要数据结构。

**2.1系统总体设计**

主要采用红黑树来实现用户数据的查询以及添加新用户、删除老用户和更新用户密码等操作。

定义一个 user\_data 头文件用结构体存储用户信息，用户信息为用户名和密码。再添加一个 NODE类用于定义数据节点的基本属性：数据本身，数据的左右孩子；以及数据的初始化和用户登录系统功能的基本函数声明。

定义一个RB\_tree头文件用于描述红黑树的基本框架以及其函数的实现声明，包括查询、插入、删除、更新等基本操作。

**2.2系统功能设计**

**用户登录验证功能**

具体封装为以下的结构体。

struct user {

string name;

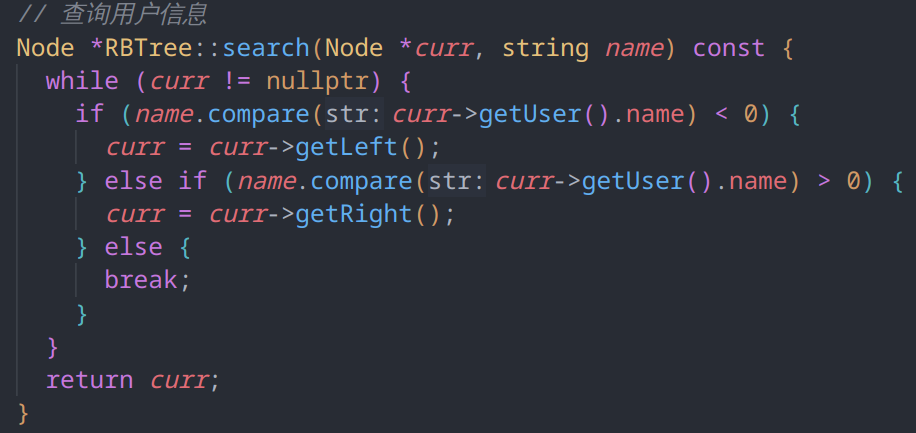
string password;

};

将数据抽象为一个节点NODE ，封装USER、color、父节点以及左右子节点到 node 中。

用户登录的具体实现，查询用户，使用了一个公有函数和一个私有函数。

私有函数通过name查询到用户信息，并返回用户信息的对应节点，如果用户不存在则返回的是空节点。



公有函数通过私有函数提供的节点进行密码的验证，认证成功则返回对应的节点，否则返回空节点。



**添加用户功能**

添加用户功能的具体实现，提供了一个公有函数和私有函数

私有函数，考虑红黑树插入节点的各种情景，执行具体的操作

插入情景1：红黑树为空树

操作处理：把插入节点作为根节点，颜色设为黑色。

插入情景2：插入节点的父节点为黑节点

由于插入节点默认设为红色（除了根节点），因此插入红节点不会影响红黑树的平衡，因此不需要自平衡操作

操作处理：直接插入节点。

插入情景3：插入节点的父节点为红节点

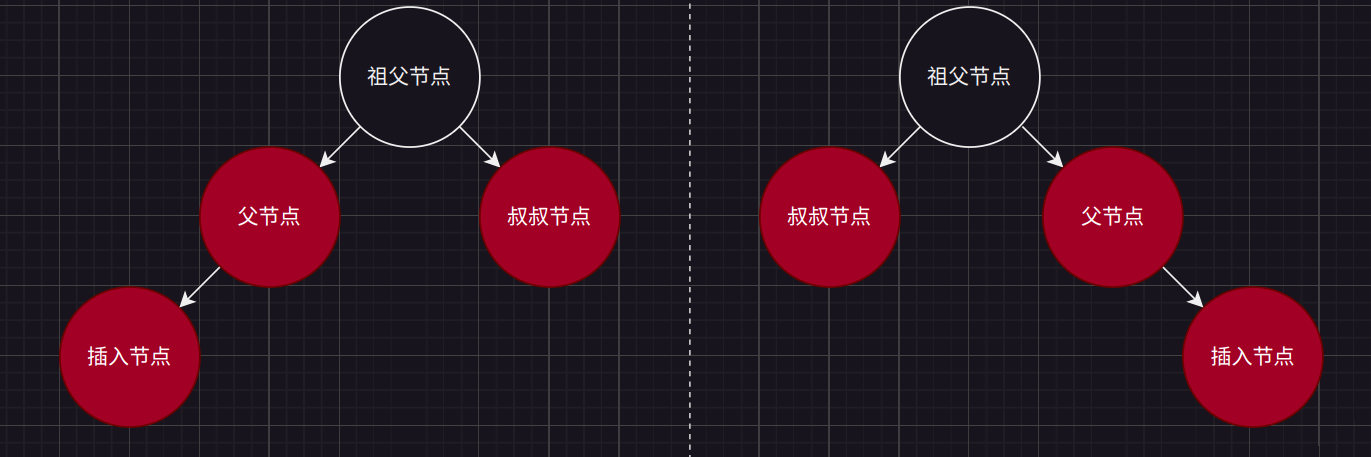
如果插入节点的父节点为红节点，则父节点肯定不为根节点，因此插入节点一定存在祖父节点。祖父节点需要参与后续的操作。

插入情景3进一步分情况考虑叔叔节点，叔叔节点即父节点的兄弟节点

插入情景3.1：叔叔节点存在且为红节点

由红黑树性质，祖父节点一定为黑节点，红节点不可能与红节点相连。此时红黑树层数的情况是：黑红红，更改红黑红。

操作处理：将父节点和兄弟节点设为黑色，将祖父节点设为红色，将祖父节点设为当前插入节点，继续找到祖父节点的父节点，如果祖父节点的父节点为黑色，则不需要再操作，如果祖父节点的父节点为红色，则以祖父节点作为新的插入节点，执行自平衡操作。如果祖父节点刚好为根节点则还需要设置回黑色。



插入情景3.1

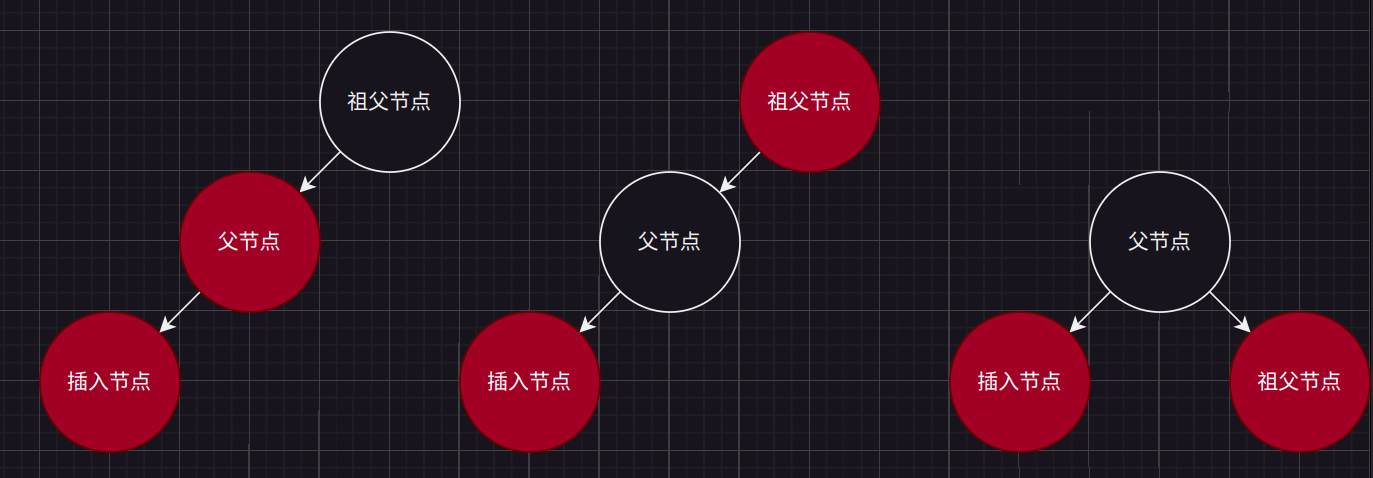
插入情景3.2：叔叔节点不存在或为黑节点，且插入节点的父节点是祖父节点的左子节点

这里叔叔节点为黑节点，其子节点只能为或不存在该节点（Nil），否则违反了红黑树的性质。

进一步考虑插入节点与父节点的关系

插入情景3.2.1：插入节点是父节点的左子节点

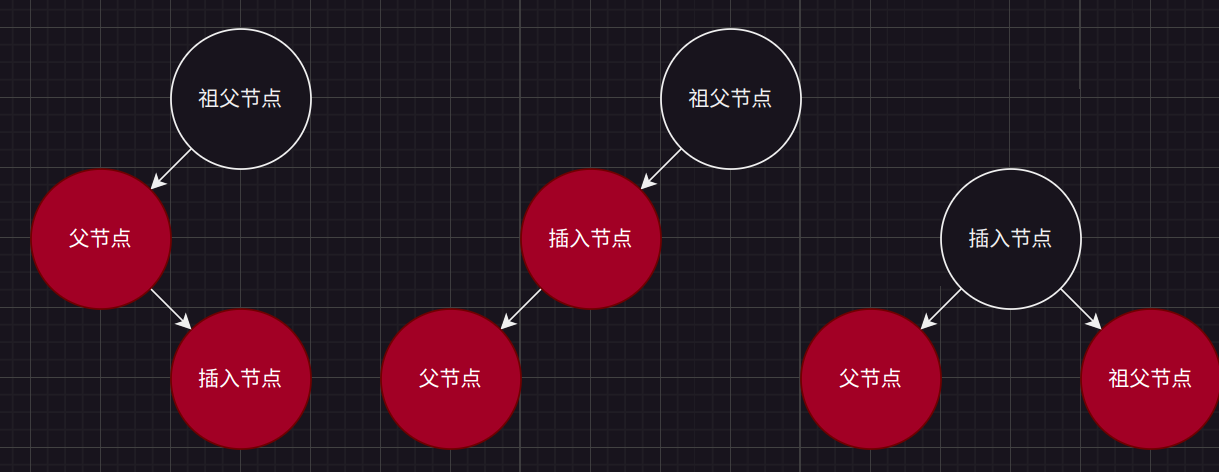
操作处理：将父节点设为黑色，将祖父节点设为红色，对祖父节点进行右旋操作。



插入情景3.2.1及其调整过程

插入情景3.2.2：插入节点是父节点的右子节点

操作处理：对父节点左旋，将父节点设为插入节点，即转化为插入情景3.2.1,进行3.2.1的操作即可。



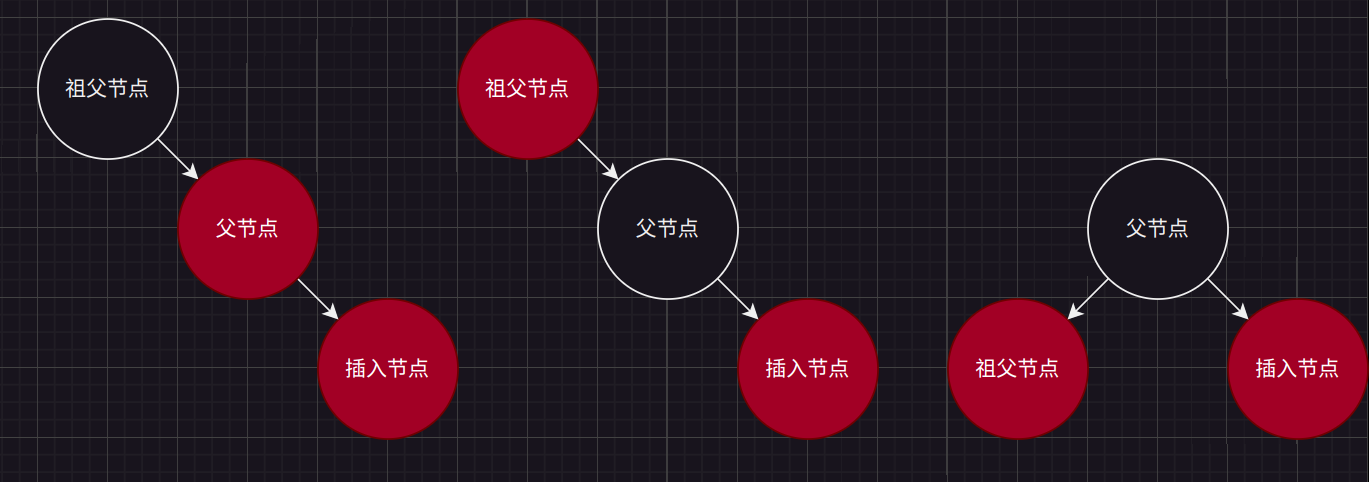
插入情景3.2.2

插入情景3.3：叔叔节点不存在或为黑节点，且插入节点的父节点是祖父节点的右子节点

进一步考虑插入节点与父节点的关系

插入情景3.3.1：插入节点是父节点的右子节点

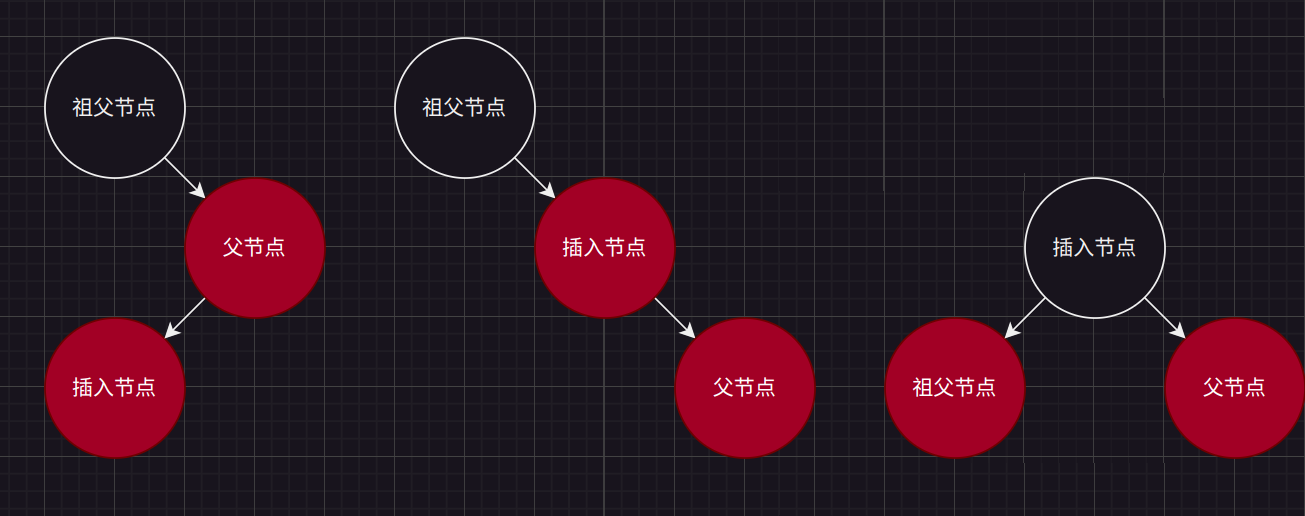
操作处理：将父节点设为黑色，将祖父节点设为红色，对祖父节点进行左旋操作。



插入情景3.3.1及其调整过程

插入情景3.3.2：插入节点是父节点的左子节点

操作处理：对父节点右旋，将父节点设为插入节点，即转化为插入情景3.3.1,进行3.3.1的操作即可。



插入情景3.3.2

公有函数，

TODO

**删除用户功能**

删除用户功能的具体实现，提供了一个公有函数和私有函数

私有函数，考虑红黑树删除节点的各种情景，执行具体的操作

红黑树删除节点，需要先寻找替代节点，再在删除之后进行自平衡。因此这里可以复用前面的私有查询函数，实现对用户信息的直接查询，再删除掉替代节点，并进行自平衡操作。

寻找替代节点的3种情景，替代节点的深度一定大于删除节点

情景1：删除节点无子节点，直接删除

情景2：删除节点只有一个子节点，用子节点替换删除节点

情景3：删除节点有两个子节点，用后继节点（大于删除节点的最小节点）替换删除节点

思路：删除节点被替代节点替换之后，不考虑节点的具体数据，对于红黑树来说相当于删除了替代节点。

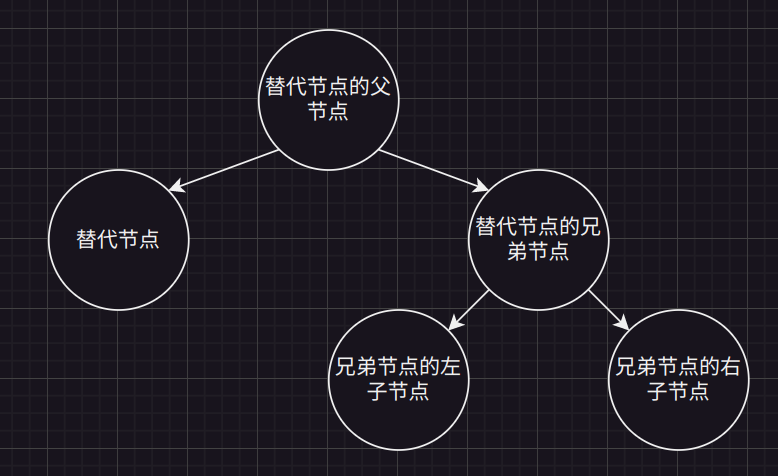
基于以上思路，再分析前面的3种情景

情景2：删除用其唯一的子节点替换，子节点替换为删除节点后，相当于删除了子节点，若子节点又存在两个子节点，则转换情景3，一直自顶向下转换，直到转换为情景1（对于红黑树来说，只存在一个子节点的节点肯定在树末的位置）。

情景3：删除节点用后继节点替代，如果后继节点有右子节点，则转换为情景2，否则转换为情景1。

总之，情景2、3最后都能转换情景1。实际上删除节点就相当于是寻找前继节点或者后继节点来替换删除节点，并删除原来的替换节点。

关于替代节点的一些约定及关系



替代节点的约定及关系

删除情景1：替代节点是红节点

删除红节点，不影响红黑树的平衡，只需要将替代节点的颜色设为删除节点的颜色，即可维持红黑树平衡

操作处理：用替代节点替换掉删除节点，并将替代节点的颜色变为删除节点的颜色

删除情景2：替代节点是黑节点

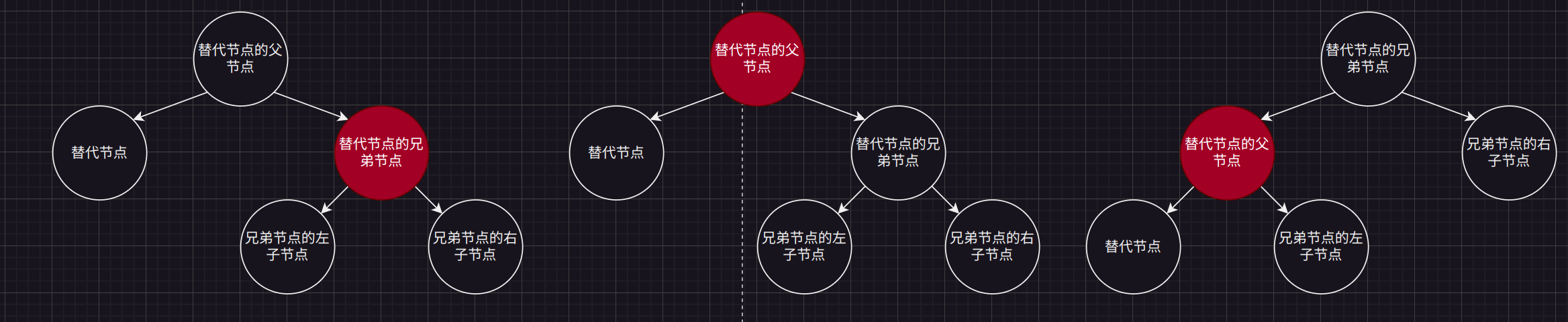
当替代节点是黑节点时，就必须进行自平衡操作。需要判断替代节点是其父节点的左子节点还是右子节点，来执行对应的旋转操作，使红黑树重新平衡。

删除情景2.1：替代节点是其父节点的左子节点

删除情景2.1.1：替代节点的兄弟节点是红节点

兄弟节点为红节点，则说明兄弟节点的父节点和子节点肯定为黑节点，不会出现其他情况。

操作处理：将兄弟节点设为黑色，将父节点设为红色，对父节点进行左旋操作，得到情景2.1.2.3（后续提到），执行操作。



删除情景2.1.1

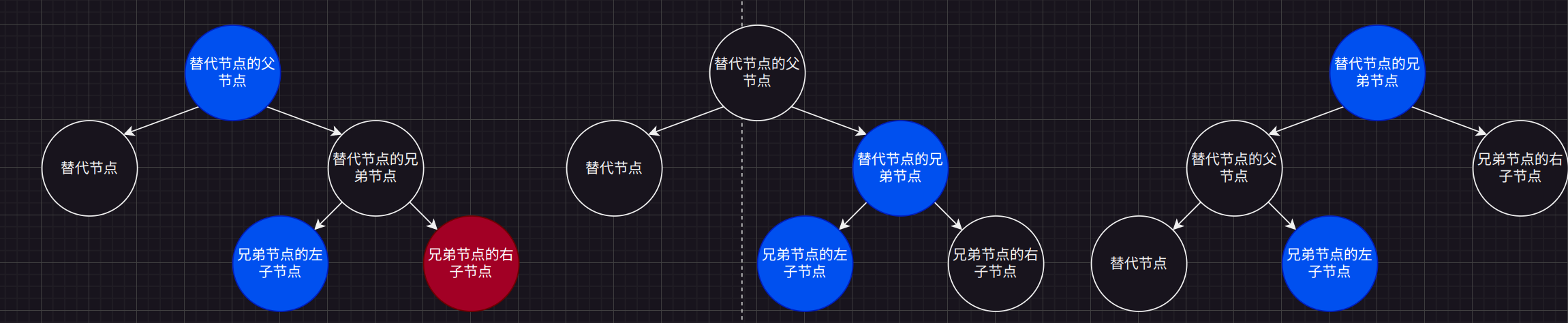
删除情景2.1.2：替代节点的兄弟节点是黑节点

兄弟节点为黑色时，无法确定其父节点和子节点的颜色，需要进一步考虑兄弟节点的子节点情况。

删除情景2.1.2.1：替代节点的兄弟节点的右子节点是红节点，左子节点任意颜色

即将删除左子树的黑色替代节点，则左子树的黑节点减少1，右子树有红色节点，向右子树借节点来补充黑节点。由于红黑树性质，兄弟节点的左子节点肯定为红色或者不存在。

操作处理：将兄弟节点的颜色设为父节点的颜色，将父节点设为黑色，将兄弟节点的右子节点设为黑色，对父节点进行左旋操作。

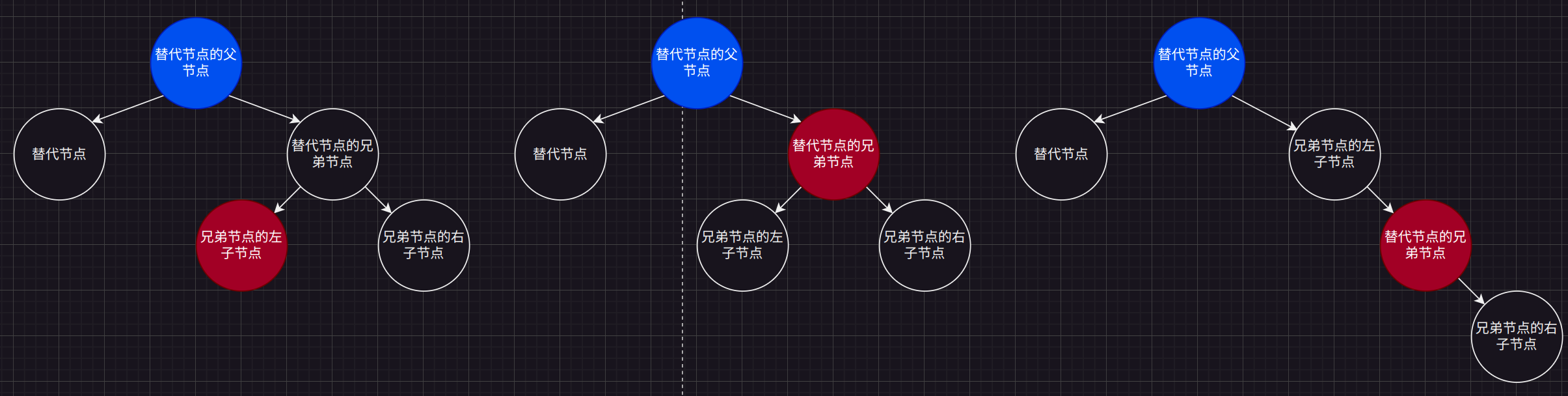


删除情景2.1.2.1（其中蓝色表示节点颜色不确定或者节点不存在）

删除情景2.1.2.2：替代节点的兄弟节点的右子节点是黑节点，左子节点为红节点

兄弟节点所在的子树拥有红节点，可以转化为情景2.1.2.1处理

操作处理：将兄弟节点设为红色，将兄弟节点的左子节点设为黑色，对兄弟节点进行右旋操作，转换为情景2.1.2.1，执行操作。

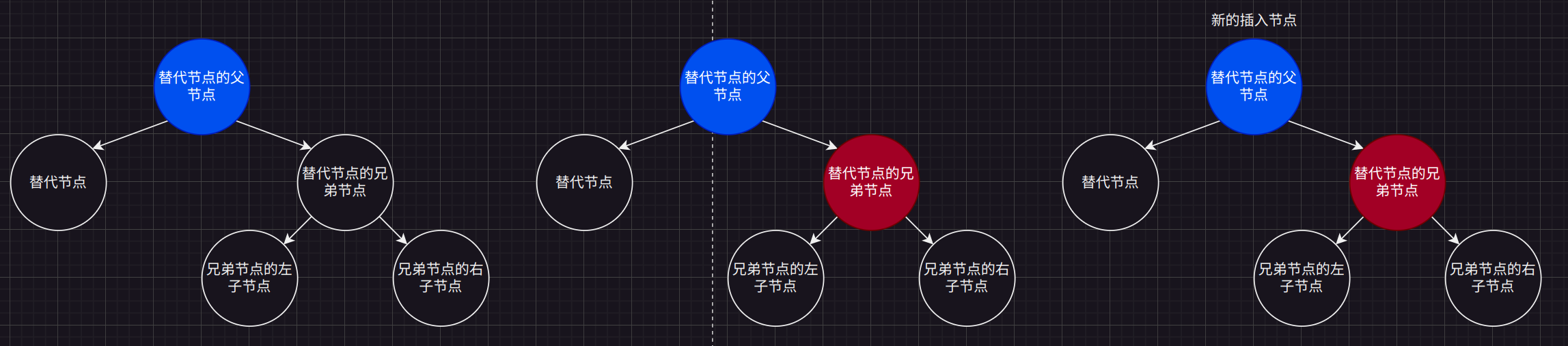


删除情景2.1.2.2（其中蓝色表示节点颜色不确定或者节点不存在）

删除情景2.1.2.3：替代节点的兄弟节点的子节点全为黑色

兄弟子树没有多余的节点可以借，往上找父子树借，将父节点作为替代节点，将兄弟节点设为红色以保证在替代节点删除后，父节点所在子树的平衡。

操作处理：将兄弟节点设为红色，将父节点作为新的替代节点，重新进行删除节点的情景处理。

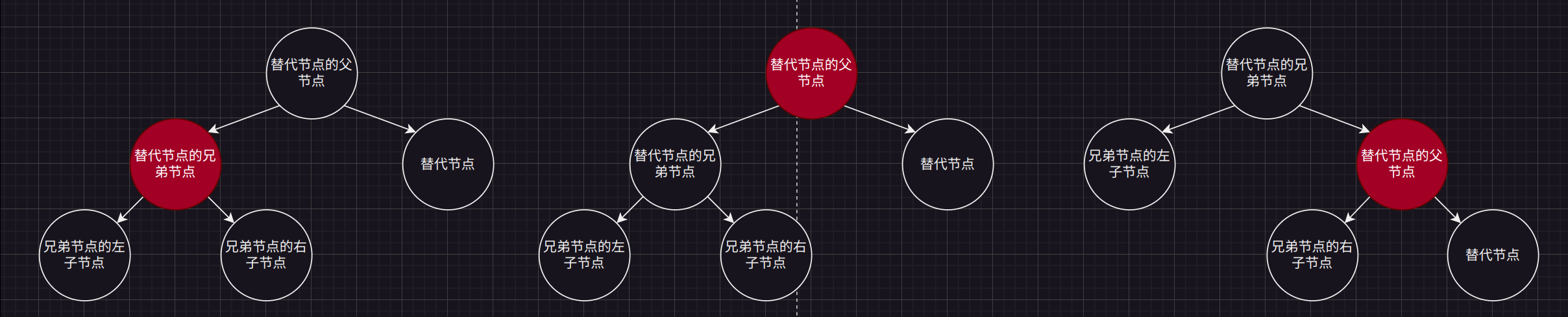


删除情景2.1.2.3（其中蓝色表示节点颜色不确定或者节点不存在）

删除情景2.2：替代节点是其父节点的右子节点

删除情景2.2.1：替代节点的兄弟节点是红节点

操作处理：将兄弟节点设为黑色，将父节点设为红色，对父节点进行右旋操作，转换为情景2.2.2.3处理。

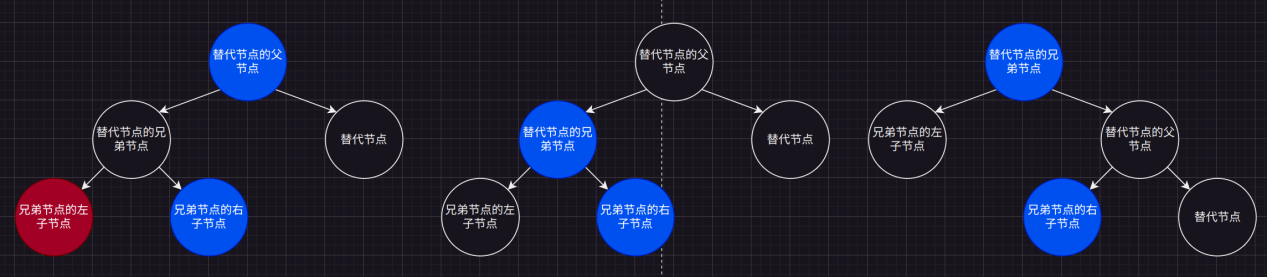


删除情景2.2.1

删除情景2.2.2：替代节点的兄弟节点是黑节点

删除情景2.2.2.1：替代节点的兄弟节点的左子节点是红节点，右子节点任意颜色

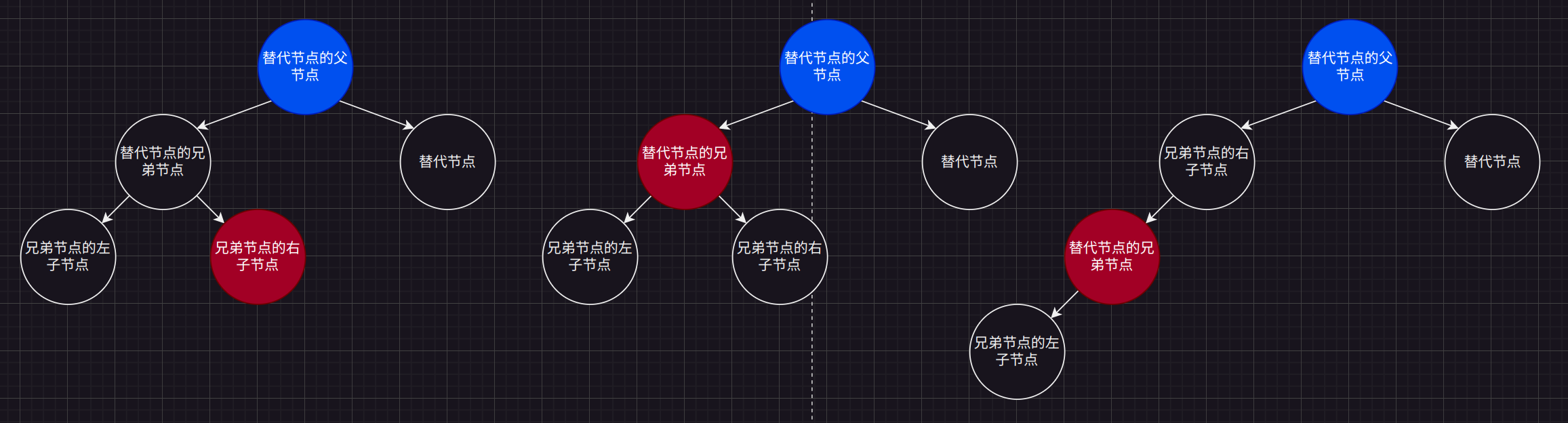
操作处理：将兄弟节点的颜色设为父节点的颜色，将父节点的颜色设为黑色，将兄弟节点的左子节点设为黑色，对父节点进行右旋操作。



删除情景2.2.2.1（其中蓝色表示节点颜色不确定或者节点不存在）

删除情景2.2.2.2：替代节点的兄弟节点的左子节点是黑节点，右子节点是红节点

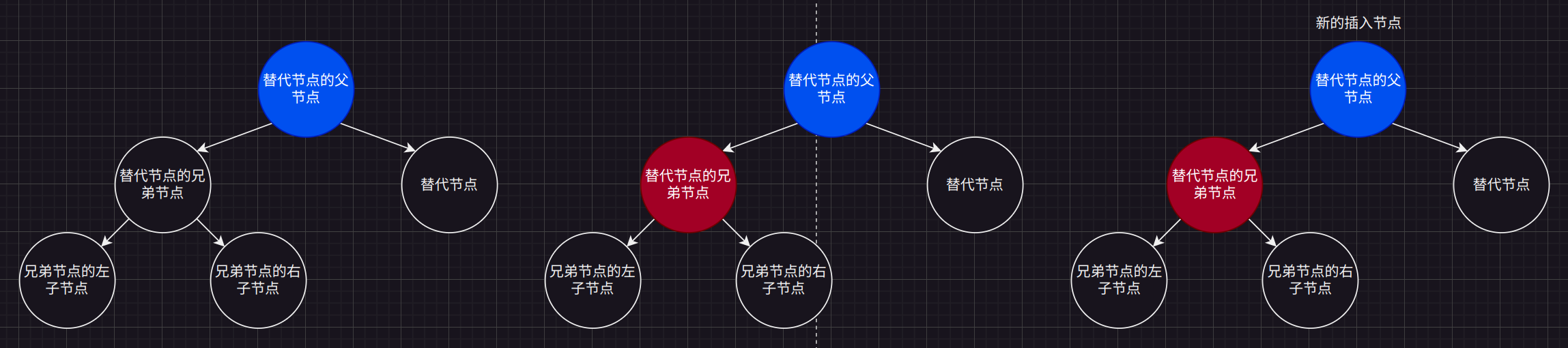
操作处理：将兄弟节点设为红色，将兄弟节点的右子节点设为黑色，对兄弟节点进行左旋操作，转换为情景2.2.2.1处理。



删除情景2.2.2.2（其中蓝色表示节点颜色不确定或者节点不存在）

删除情景2.2.2.3：替代节点的兄弟节点的子节点都是黑节点

操作处理：将兄弟节点设为红色，将父节点作为新的替代节点，重新进行删除节点的情景处理。



删除情景2.2.2.3（其中蓝色表示节点颜色不确定或者节点不存在）

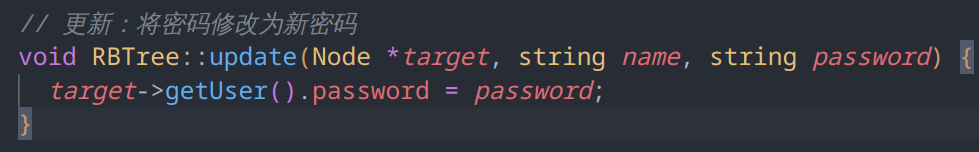
公有函数

TODO

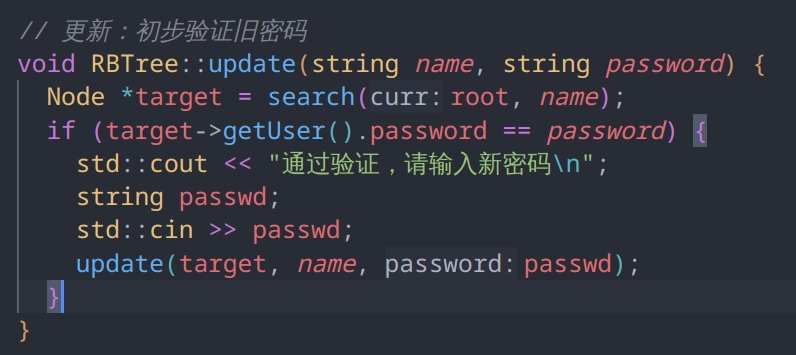
**更新用户密码功能**

更新用户密码功能的具体实现，提供了一个公有函数和私有函数

私有函数接收需要修改的数据节点，直接设置用户的密码即可。



公有函数，直接复用查询函数找到需要修改密码的数据节点，在修改密码之前需要先验证旧密码。



**2.3 类的设计**

用户数据部分



功能实现部分



**2.4 主程序的设计**

1. **调试分析**

**3.1 技术难点分析**

主要是红黑树各种插入、删除情景考虑，以及传递参数列表的设计

**3.2 调试错误分析**

**错误情况1：插入情景下叔叔节点定位错误**

当前节点curr：用于寻找插入节点的位置，主要是为了将数据操作与红黑树关联起来（curr的作用域能够修改红黑树的数据结构），以及寻找插入情景下，需要提供的父节点以及叔叔节点信息。

父节点parent：由当前节点curr使用getParent方法获得。

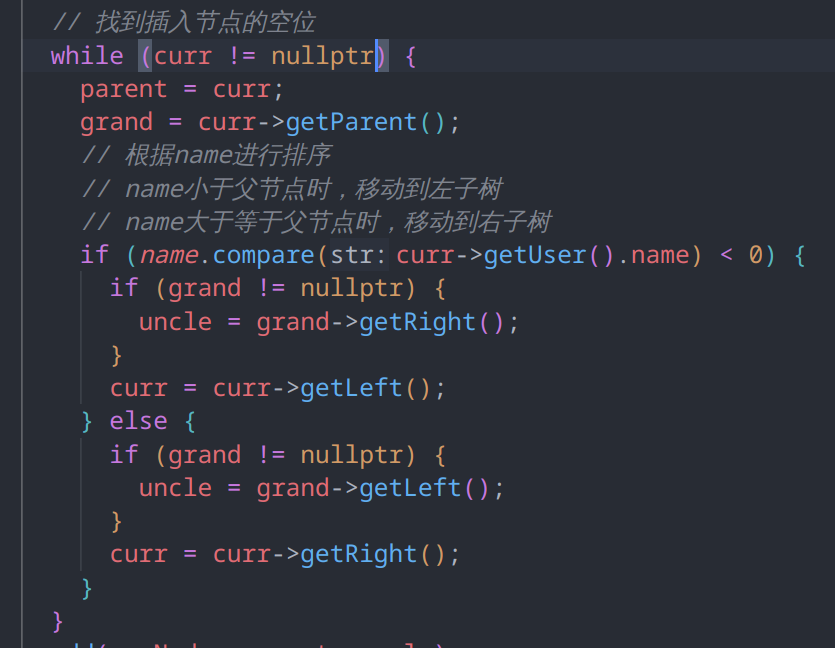
祖父节点grand：获取方法同父节点。（祖父节点可以为空）

叔叔节点uncle：由祖父节点grand在curr进行操作的同时

在寻找红黑树的父节点，以及叔叔节点时，根据当前节点在比较已有节点来寻找插入节点位置的同时，寻找叔叔节点。（叔叔节点可以为空）

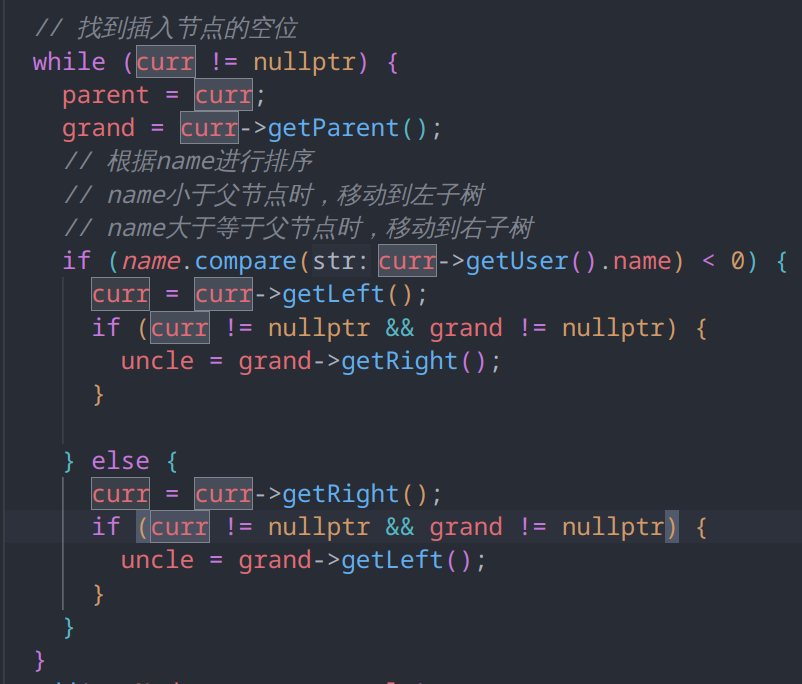
错误代码复现

错误版本一



主要问题在于curr节点在最后找到空节点时，在之前uncle节点就被更新了一次，实际上在curr节点已经找到了插入节点所在的位置时（此时插入节点还未插入，寻找到的是一个空节点，主要是为了确定parent节点和uncle节点的位置），uncle节点就不需要再进行更新了。上面的代码uncle节点在curr节点之前更新会导致uncle节点有概率指向parent节点的位置。

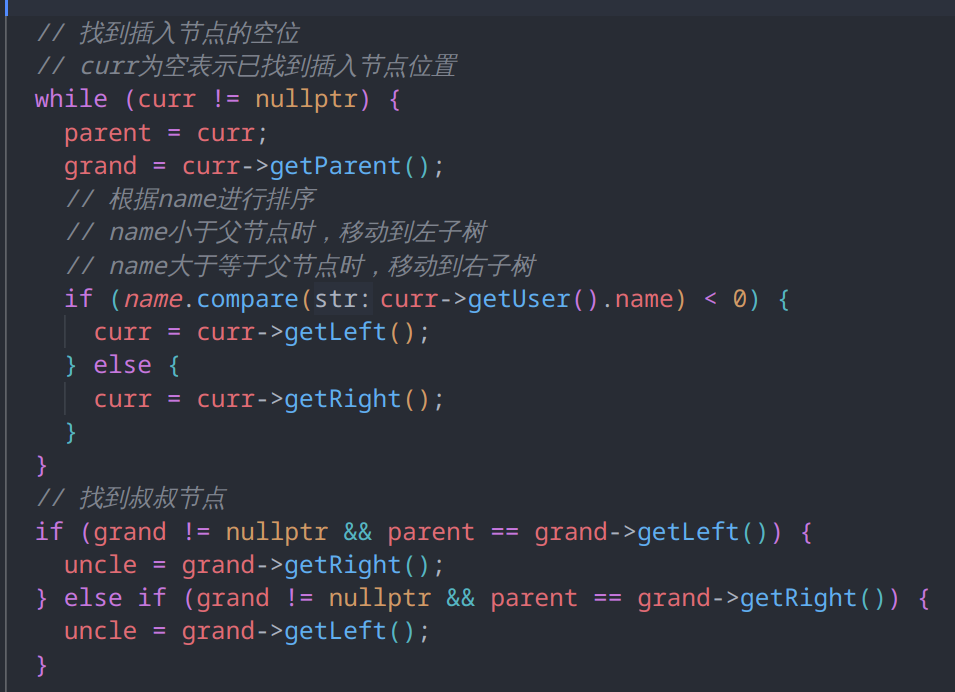
错误版本二



将curr节点更新的代码提到uncle节点更新之前，并附加curr节点是否为空的判断，如果为空则不需要再进行更新。这样就确定了插入节点的叔叔节点uncle节点的位置。

这里改进了错误版本一uncle节点定位被多更新一次的而导致错位的问题。但是当处理左右旋或者右左旋情况时，又会因为在寻找到插入节点的前一个位置时，由于uncle节点的寻找路径被设置为与寻找插入节点的左右方向相反，则在左右旋和右左旋的结构下，会因为延后curr节点更新而导致uncle节点又指向了parent节点。

最终代码修正如下



将uncle节点的寻找，直接放在后面处理，根据grand节点与parent节点的关系，准确找到uncle节点，由于grand节点和parent节点是能够准确定位的，因此在仅需要考虑grand节点和parent节点，而不需要考虑curr节点的情况下，uncle节点也能够准确定位。

**错误情况2：左旋、右旋、左右旋、右左旋处理问题**

以插入节点为基准定义相关节点

插入节点node：作为一个左子节点或右子节点的指针参数传递过来。

父节点parent：作为一个父节点的指针参数传递过来。

祖父节点grand：通过parent的getParent获取。（祖父节点可以为空）

左旋和右旋分别封装成单独的方法，左右旋和右左旋通过组合使用这两个方法实现。

在左旋和右旋情况下，传递的参数是插入节点和其父节点。

错误代码复现（左旋部分，右旋部分同理）



以上代码在处理左右旋和右左旋时，由于最后在将结果同步到红黑树上时只考虑了父节点与祖父节点之间关系的一种情况，因此处理左右旋时，在第一次左旋时，旋转结果就同步到了与预期结果相反的位置上（本来是在左子树上，结果变成了在右子树上）。处理右左旋时，也是类似的情况。

代码修正（左旋部分，右旋部分同理）



以上代码增加了对左右旋和右左旋情况的考虑，在左右旋和右左旋情况下，第一次传递的参数是插入节点和其父节点，第二次传递的参数是父节点和祖父节点。由于父节点和祖父节点是能准确定位的，且脱离左旋和右旋的单一情况以及左右旋和右左旋的混合情况，直接考虑祖父节点与父节点的关系，实现在任意情况下，旋转结果都能正确地同步到红黑树上。

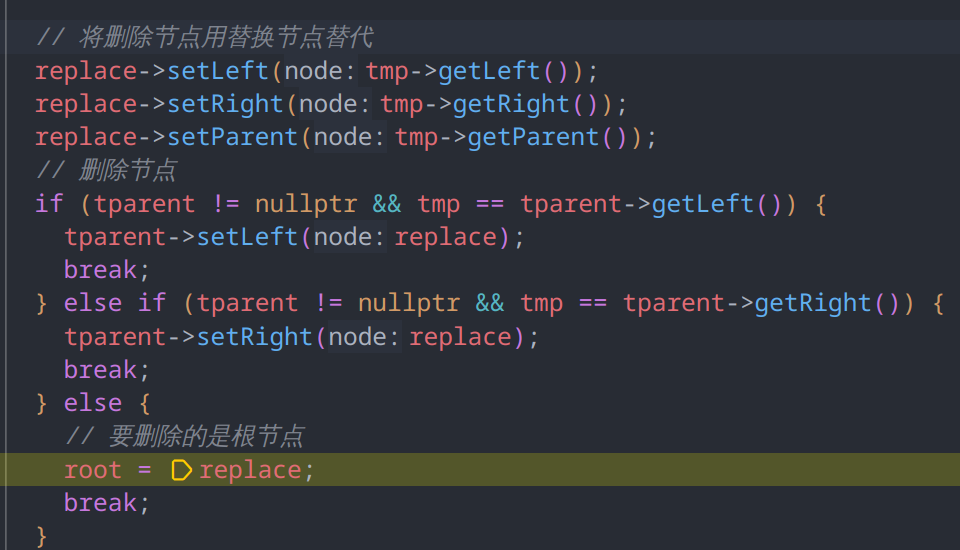
**错误情况3：删除情景下节点死循环**

删除节点：需要从红黑树结构中移除的节点

替换节点：原本就在红黑树结构存在的节点，通常它是删除节点的前继节点或后继节点。

删除方法的实现步骤涉及到通过查询到的节点来找到红黑树结构中对应的删除节点，寻找删除节点的前继节点或后继节点作为替换节点实现删除（替换节点可以为空，直接删除的情况），用替换节点替换掉删除节点后的自平衡处理 等主要三个部分。

错误代码复现（删除节点部分）



以上代码在处理的数据节点较少时容易出现节点死循环 （一个节点的指针不断地指向自身，无限循环下去）。主要就是在删除节点与替换节点是直接相连的情况下，替代节点为了替换删除节点而更新其左子节点指针和右子节点指针时，误将其中一个指针指向了自身而导致的问题。在替换节点与删除节点不直接连接的情况下是没有问题的，这里缺少了对“替换节点与删除节点直接连接”的情况的考虑和处理。

代码修正



以上代码增加了对删除节点周边节点的检测，避免了出现死循环的边界条件。另外，由于替换节点的深度一定大于删除节点，因此对于父节点更新不需要做判断，因为替换节点不可能是删除节点的父节点或者是深度更小的节点。

1. **测试结果分析**

**4.1 调试测试情况**

**4.2 总体分析**

1. **实验总结**

**参考资料**

**<https://www.jianshu.com/p/e136ec79235c/>**

**<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>**

**<https://visualgo.net/zh/bst>**

**<https://zhuanlan.zhihu.com/p/91960960>**