红黑树

维基百科，自由的百科全书

红黑树（英語：Red–black tree）是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。它在1972年由鲁道夫·贝尔发明，被称为"**对称二叉B树**"，它现代的名字源于Leo J. Guibas和Robert Sedgewick于1978年写的一篇论文。红黑树的结构复杂，但它的操作有着良好的最坏情况运行时间，并且在实践中高效：它可以在$O(\log n)$时间内完成查找、插入和删除，这里的n是树中元素的数目。

目录

用途和好处

红黑树和AVL树一样都对插入时间、删除时间和查找时间提供了最好可能的最坏情况担保。这不只是使它们在时间敏感的应用，如实时应用（real time application）中有价值，而且使它们有在提供最坏情况担保的其他数据结构中作为基础模板的价值；例如，在计算几何中使用的很多数据结构都可以基于红黑树实现。

红黑树在函数式编程中也特别有用，在这里它们是最常用的持久数据结构（persistent data structure）之一，它们用来构造关联数组和集合，每次插入、删除之后它们能保持为以前的版本。除了$O(\log n)$的时间之外，红黑树的持久版本对每次插入或删除需要$O(\log n)$的空间。

红黑树是2-3-4树的一种等同。换句话说，对于每个2-3-4树，都存在至少一个数据元素是同样次序的红黑树。在2-3-4树上的插入和删除操作也等同于在红黑树中颜色翻转和旋转。这使得2-3-4树成为理解红黑树背后的逻辑的重要工具，这也是很多介绍算法的教科书在红黑树之前介绍2-3-4树的原因，尽管2-3-4树在实践中不经常使用。

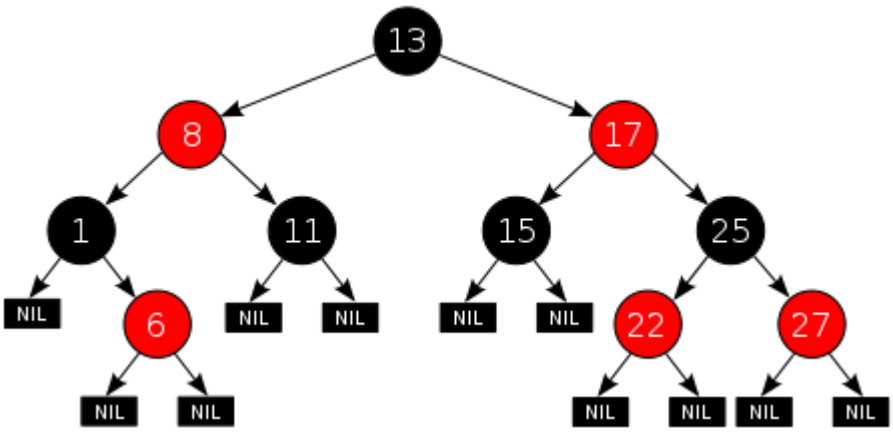
红黑树相对于AVL树来说，牺牲了部分平衡性以换取插入/删除操作时少量的旋转操作，整体来说性能要优于AVL树。

性质

红黑树是每个节点都带有*颜色*属性的二叉查找树，颜色为*红色*或*黑色*。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求：

1. 节点是红色或黑色。
2. 根是黑色。
3. 所有叶子都是黑色（叶子是NIL节点）。
4. 每个红色节点必须有两个黑色的子节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）
5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

下面是一个具体的红黑树的图例：



这些约束确保了红黑树的关键特性：从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

要知道为什么这些性质确保了这个结果，注意到性质4导致了路径不能有两个毗连的红色节点就足够了。最短的可能路径都是黑色节点，最长的可能路径有交替的红色和黑色节点。因为根据性质5所有最长的路径都有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。

在很多树数据结构的表示中，一个节点有可能只有一个子节点，而叶子节点包含数据。用这种范例表示红黑树是可能的，但是这会改变一些性质并使算法复杂。为此，本文中我们使用"nil叶子"或"空（null）叶子"，如上图所示，它不包含数据而只充当树在此结束的指示。这些节点在绘图中经常被省略，导致了这些树好像同上述原则相矛盾，而实际上不是这样。与此有关的结论是所有节点都有两个子节点，尽管其中的一个或两个可能是空叶子。

操作

因为每一个红黑树也是一个特化的二叉查找树，因此红黑树上的只读操作与普通二叉查找树上的只读操作相同。然而，在红黑树上进行插入操作和删除操作会导致不再符合红黑树的性质。恢复红黑树的性质需要少量（$O(\log n)$）的颜色变更（实际是非常快速的）和不超过三次树旋转（对于插入操作是两次）。虽然插入和删除很复杂，但操作时间仍可以保持为$O(\log n)$次。

插入

我们首先以二叉查找树的方法增加节点并标记它为红色。（如果设为黑色，就会导致根到叶子的路径上有一条路上，多一个额外的黑节点，这个是很难调整的。但是设为红色节点后，可能会导致出现两个连续红色节点的冲突，那么可以通过颜色调换（color flips）和树旋转来调整。）下面要进行什么操作取决于其他临近节点的颜色。同人类的家族树中一样，我们将使用术语*叔父节点*来指一个节点的父节点的兄弟节点。注意：

- 性质1和性质3总是保持着。
- 性质4只在增加红色节点、重绘黑色节点为红色，或做旋转时受到威胁。
- 性质5只在增加黑色节点、重绘红色节点为黑色，或做旋转时受到威胁。

在下面的示意图中，将要插入的节点标为**N**，N的父节点标为**P**，N的祖父节点标为**G**，N的叔父节点标为**U**。在图中展示的任何颜色要么是由它所处情形这些所作的假定，要么是假定所暗含（imply）的。

对于每一种情形，我们将使用C示例代码来展示。通过下列函数，可以找到一个节点的叔父和祖父节点：

```
node* grandparent(node *n){
    return n->parent->parent;
}

node* uncle(node *n){
    if(n->parent == grandparent(n)->left)
        return grandparent (n)->right;
    else
        return grandparent (n)->left;
}
```

情形1:新节点N位于树的根上，没有父节点。在这种情形下，我们把它重绘为黑色以满足性质2。因为它在每个路径上对黑节点数目增加一，性质5符合。

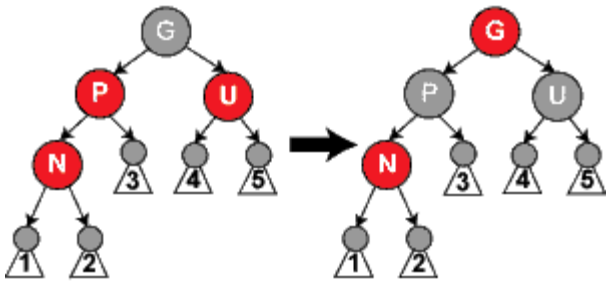
```
void insert_case1(node *n){
    if(n->parent == NULL)
        n->color = BLACK;
    else
        insert_case2 (n);
}
```

情形2:新节点的父节点P是黑色，所以性质4没有失效（新节点是红色的）。在这种情形下，树仍是有效的。性质5也未受到威胁，尽管新节点N有两个黑色叶子子节点；但由于新节点N是红色，通过它的每个子节点的路径就都有同通过它所取代的黑色的叶子的路径同样数目的黑色节点，所以依然满足这个性质。

```
void insert_case2(node *n){
    if(n->parent->color == BLACK)
        return; /* 树仍旧有效*/
    else
        insert_case3 (n);
}
```

注意：在下列情形下我们假定新节点的父节点为红色，所以它有祖父节点；因为如果父节点是根节点，那父节点就应当是黑色。所以新节点总有一个叔父节点，尽管在情形4和5下它可能是叶子节点。

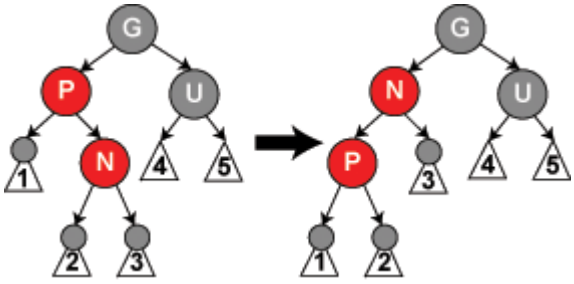
情形3:如果父节点P和叔父节点U二者都是红色，（此时新插入节点N做为P的左子节点或右子节点都属于情形3，这里右图仅显示N做为P左子的情形）则我们可以将它们两个重绘为黑色并重绘祖父节点G为红色（用来保持性质5）。现在我们的新节点N有了一个黑色的父节点P。因为通过父节点P或叔父节点U的任何路径都必定通过祖父节点G，在这些路径上的黑节点数目没有改变。但是，红色的祖父节点G可能是根节点，这就违反了性质2，也有可能祖父节点G的父节点是红色的，这就违反了性质4。为了解决这个问题，我们在祖父节点G上递归地进行**情形1**的整个过程。（把G当成是新加入的节点进行各种情形的检查）



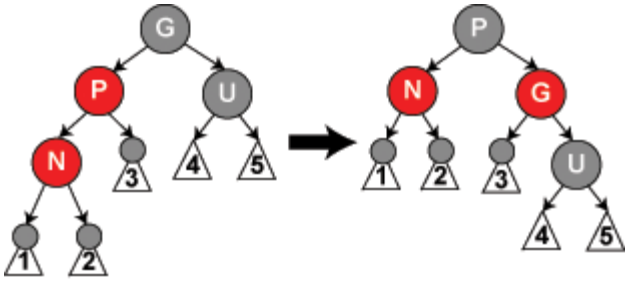

```
void insert_case3(node *n){
    if(uncle(n) != NULL && uncle (n)->color == RED) {
        n->parent->color = BLACK;
        uncle (n)->color = BLACK;
        grandparent (n)->color = RED;
        insert_case1(grandparent(n));
    }
    else
        insert_case4 (n);
}
```

注意：在余下的情形下，我们假定父节点P是其祖父G的左子节点。如果它是右子节点，情形4和情形5中的*左*和*右*应当对调。

情形4:父节点P是红色而叔父节点U是黑色或缺少，并且新节点N是其父节点P的右子节点而父节点P又是其父节点的左子节点。在这种情形下，我们进行一次左旋转调换新节点和其父节点的角色;接着，我们按**情形5**处理以前的父节点P以解决仍然失效的性质4。注意这个改变会导致某些路径通过它们以前不通过的新节点N（比如图中1号叶子节点）或不通过节点P（比如图中3号叶子节点），但由于这两个节点都是红色的，所以性质5仍有效。




```
void insert_case4(node *n){
    if(n == n->parent->right && n->parent == grandparent(n)->left) {
        rotate_left(n);
        n = n->left;
    } else if(n == n->parent->left && n->parent == grandparent(n)->right) {
        rotate_right(n);
        n = n->right;
    }
    insert_case5 (n);
}
```



情形5：父节点P是红色而叔父节点U是黑色或缺少，新节点N是其父节点的左子节点，而父节点P又是其父节点G的左子节点。在这种情形下，我们进行针对祖父节点G的一次右旋转；在旋转产生的树中，以前的父节点P现在是新节点N和以前的祖父节点G的父节点。我们知道以前的祖父节点G是黑色，否则父节点P就不可能是红色（如果P和G都是紅色就違反了性質4，所以G必須是黑色）。我们切换以前的父节点P和祖父节点G的颜色，结果的树满足性质4。性质5也仍然保持满足，因为通过这三个节点中任何一个的所有路径以前都通过祖父节点G，现在它们都通过以前的父节点P。在各自的情形下，这都是三个节点中唯一的黑色节点。

```
void insert_case5(node *n){
    n->parent->color = BLACK;
    grandparent (n)->color = RED;
    if(n == n->parent->left && n->parent == grandparent(n)->left) {
        rotate_right(n->parent);
    } else {
        /* Here, n == n->parent->right && n->parent == grandparent (n)->right */
        rotate_left(n->parent);
    }
}
```

注意插入实际上是原地算法，因为上述所有调用都使用了尾部递归。

删除

如果需要删除的节点有两个儿子，那么问题可以被转化成删除另一个只有一个儿子的节点的问题（为了表述方便，这里所指的儿子，为非叶子节点的儿子）。对于二叉查找树，在删除带有两个非叶子儿子的节点的时候，我们要么找到它左子树中的最大元素、要么找到它右子树中的最小元素，并把它的值转移到要删除的节点中（如在这里所展示的那样）。我们接着删除我们从中复制出值的那个节点，它必定有少于两个非叶子的儿子。因为只是复制了一个值（没有复制颜色），不违反任何性质，这就把问题简化为如何删除最多有一个儿子的节点的问题。它不关心这个节点是最初要删除的节点还是我们从中复制出值的那个节点。

在本文余下的部分中，**我们只需要讨论删除只有一个儿子的节点**（如果它两个儿子都为空，即均为叶子，我们任意将其中一个看作它的儿子）。如果我们删除一个红色节点（此时该节点的儿子将都为叶子节点），它的父亲和儿子一定是黑色的。所以我们可以简单的用它的黑色儿子替换它，并不会破坏性质3和性质4。通过被删除节点的所有路径只是少了一个红色节点，这样可以继续保证性质5。另一种简单情况是在被删除节点是黑色而它的儿子是红色的时候。如果只是去除这个黑色节点，用它的红色儿子顶替上来的话，会破坏性质5，但是如果我们重绘它的儿子为黑色，则曾经通过它的所有路径将通过它的黑色儿子，这样可以继续保持性质5。

需要进一步讨论的是在要删除的节点和它的儿子二者都是黑色的时候，这是一种复杂的情况（这种情况下该结点的两个儿子都是叶子结点，否则若其中一个儿子是黑色非叶子结点，另一个儿子是叶子结点，那么从该结点通过非叶子结点儿子的路径上的黑色结点数最小为2，而从该结点到另一个叶子结点儿子的路径上的黑色结点数为1，违反了性质5）。我们首先把要删除的节点替换为它的儿子。出于方便，称呼这个儿子为**N**（在新的位置上），称呼它的兄弟（它父亲的另一个儿子）为**S**。在下面的示意图中，我们还是使用**P**称呼N的父亲，$\mathbf{S_L}$称呼S的左儿子，$\mathbf{S_R}$称呼S的右儿子。我们将使用下述函数找到兄弟节点：

```
struct node *
sibling(struct node *n)
{
        if(n == n->parent->left)
                return n->parent->right;
        else
                return n->parent->left;
}
```

我们可以使用下列代码进行上述的概要步骤，这里的函数`replace_node`替换`child`到n在树中的位置。出于方便，在本章节中的代码将假定空叶子被用不是NULL的实际节点对象来表示（在*插入*章节中的代码可以同任何一种表示一起工作）。

```
void
delete_one_child(struct node *n)
{
        /*
         * Precondition: n has at most one non-null child.
         */
        struct node *child = is_leaf(n->right)? n->left : n->right;

        replace_node(n, child);
        if(n->color == BLACK){
                if(child->color == RED)
                        child->color = BLACK;
                else
                        delete_case1 (child);
        }
        free (n);
}
```

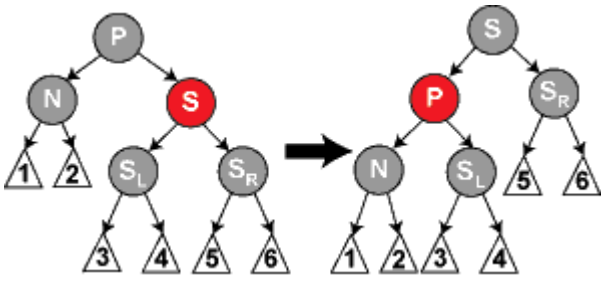
如果N和它初始的父亲是黑色，则删除它的父亲导致通过N的路径都比不通过它的路径少了一个黑色节点。因为这违反了性质5，树需要被重新平衡。有几种情形需要考虑：

情形1: N是新的根。在这种情形下，我们就做完了。我们从所有路径去除了一个黑色节点，而新根是黑色的，所以性质都保持着。

```
void
delete_case1(struct node *n)
{
        if(n->parent != NULL)
                delete_case2 (n);
}
```

注意：在情形2、5和6下，我们假定N是它父亲的左儿子。如果它是右儿子，则在这些情形下的*左*和*右*应当对调。

情形2： S是红色。在这种情形下我们在N的父亲上做左旋转，把红色兄弟转换成N的祖父，我们接着对调N的父亲和祖父的颜色。完成这两个操作后，尽管所有路径上黑色节点的数目没有改变，但现在N有了一个黑色的兄弟和一个红色的父亲（它的新兄弟是黑色因为它是红色S的一个儿子），所以我们可以接下去按**情形4**、**情形5**或**情形6**来处理。

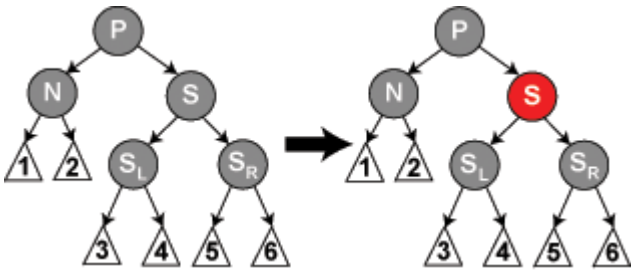


（注意：这里的图中没有显示出来，N是删除了黑色节点后替换上来的子节点，所以这个过程中由P->X->N变成了P->N，实际上是少了一个黑色节点，也可以理解为Parent(Black)和Silbing(Red)那么他们的孩子黑色节点的数目肯定不等，让他们做新兄弟肯定是不平衡的，还需后面继续处理。这里看英文版本的[1] (https://en.wikipedia.org/wiki/Red%E2%80%93black_tree)比较的明了）

```
void
delete_case2(struct node *n)
{
        struct node *s = sibling (n);

        if(s->color == RED){
                n->parent->color = RED;
                s->color = BLACK;
                if(n == n->parent->left)
                        rotate_left(n->parent);
                else
                        rotate_right(n->parent);
        }
        delete_case3 (n);
}
```

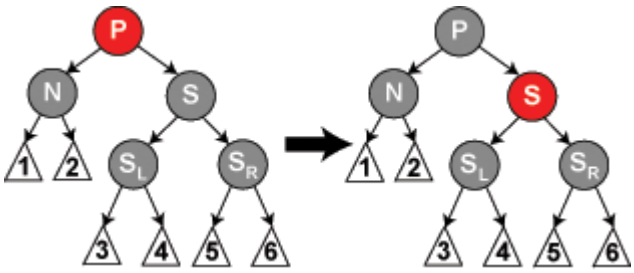
情形3： N的父亲、S和S的儿子都是黑色的。在这种情形下，我们简单的重绘S为红色。结果是通过S的所有路径，它们就是以前不通过N的那些路径，都少了一个黑色节点。因为删除N的初始的父亲使通过N的所有路径少了一个黑色节点，这使事情都平衡了起来。但是，通过P的所有路径现在比不通过P的路径少了一个黑色节点，所以仍然违反性质5。要修正这个问题，我们要从**情形1**开始，在P上做重新平衡处理。




```
void
delete_case3(struct node *n)
{
        struct node *s = sibling (n);

        if((n->parent->color == BLACK)&&
(s->color == BLACK)&&
(s->left->color == BLACK)&&
(s->right->color == BLACK)) {
                s->color = RED;
                delete_case1(n->parent);
        } else
                delete_case4 (n);
}
```

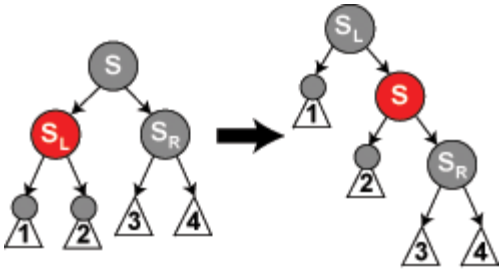
情形4： S和S的儿子都是黑色，但是N的父亲是红色。在这种情形下，我们简单的交换N的兄弟和父亲的颜色。这不影响不通过N的路径的黑色节点的数目，但是它在通过N的路径上对黑色节点数目增加了一，添补了在这些路径上删除的黑色节点。




```
void
delete_case4(struct node *n)
{
        struct node *s = sibling (n);

        if ((n->parent->color == RED)&&
(s->color == BLACK)&&
(s->left->color == BLACK)&&
(s->right->color == BLACK)) {
                s->color = RED;
                n->parent->color = BLACK;
        } else
                delete_case5 (n);
}
```

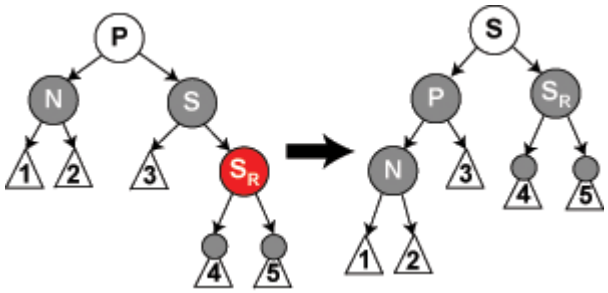
情形5： S是黑色，S的左儿子是红色，S的右儿子是黑色，而N是它父亲的左儿子。在这种情形下我们在S上做右旋转，这样S的左儿子成为S的父亲和N的新兄弟。我们接着交换S和它的新父亲的颜色。所有路径仍有同样数目的黑色节点，但是现在N有了一个黑色兄弟，他的右儿子是红色的，所以我们进入了**情形6**。N和它的父亲都不受这个变换的影响。




```
void
delete_case5(struct node *n)
{
        struct node *s = sibling (n);

        if (s->color == BLACK){ /* this if statement is trivial,
due to Case 2(even though Case two changed the sibling to a sibling's child,
the sibling's child can't be red, since no red parent can have a red child). */
// the following statements just force the red to be on the left of the left of the parent,
// or right of the right, so case six will rotate correctly.
                if((n == n->parent->left)&&
(s->right->color == BLACK)&&
(s->left->color == RED)) { // this last test is trivial too due to cases 2-4.
                        s->color = RED;
                        s->left->color = BLACK;
                        rotate_right (s);
                } else if((n == n->parent->right)&&
(s->left->color == BLACK)&&
(s->right->color == RED)) {// this last test is trivial too due to cases 2-4.
                        s->color = RED;
                        s->right->color = BLACK;
                        rotate_left (s);
                }
        }
        delete_case6 (n);
}
```

情形6： S是黑色，S的右儿子是红色，而N是它父亲的左儿子。在这种情形下我们在N的父亲上做左旋转，这样S成为N的父亲（P）和S的右儿子的父亲。我们接着交换N的父亲和S的颜色，并使S的右儿子为黑色。子树在它的根上的仍是同样的颜色，所以性质3没有被违反。但是，N现在增加了一个黑色祖先：要么N的父亲变成黑色，要么它是黑色而S被增加为一个黑色祖父。所以，通过N的路径都增加了一个黑色节点。



此时，如果一个路径不通过N，则有两种可能性：

- 它通过N的新兄弟。那么它以前和现在都必定通过S和N的父亲，而它们只是交换了颜色。所以路径保持了同样数目的黑色节点。
- 它通过N的新叔父，S的右儿子。那么它以前通过S、S的父亲和S的右儿子，但是现在只通过S，它被假定为它以前的父亲的颜色，和S的右儿子，它被从红色改变为黑色。合成效果是这个路径通过了同样数目的黑色节点。

在任何情况下，在这些路径上的黑色节点数目都没有改变。所以我们恢复了性质4。在示意图中的白色节点可以是红色或黑色，但是在变换前后都必须指定相同的颜色。

```
void
delete_case6(struct node *n)
{
        struct node *s = sibling (n);

        s->color = n->parent->color;
        n->parent->color = BLACK;

        if(n == n->parent->left){
                s->right->color = BLACK;
                rotate_left(n->parent);
        } else {
                s->left->color = BLACK;
                rotate_right(n->parent);
        }
}
```

同样的，函数调用都使用了尾部递归，所以算法是原地算法。此外，在旋转之后不再做递归调用，所以进行了恒定数目（最多3次）的旋转。

C++示例代码

```
#define BLACK 1
#define RED 0
#include <iostream>

using namespace std;

class bst {
private:

    struct Node {
        int value;
        bool color;
        Node *leftTree, *rightTree, *parent;

        Node() : value(0), color(RED), leftTree(NULL), rightTree(NULL), parent(NULL) { }

        Node* grandparent() {
            if(parent == NULL){
                return NULL;
            }
            return parent->parent;
        }

        Node* uncle() {
            if(grandparent() == NULL) {
                return NULL;
            }
            if(parent == grandparent()->rightTree)
                return grandparent()->leftTree;
            else
                return grandparent()->rightTree;
        }

        Node* sibling() {
            if(parent->leftTree == this)
                return parent->rightTree;
            else
                return parent->leftTree;
        }
    };

    void rotate_right(Node *p){
        Node *gp = p->grandparent();
        Node *fa = p->parent;
        Node *y = p->rightTree;

        fa->leftTree = y;

        if(y != NIL)
            y->parent = fa;
        p->rightTree = fa;
        fa->parent = p;

        if(root == fa)
            root = p;
        p->parent = gp;

        if(gp != NULL){
            if(gp->leftTree == fa)
                gp->leftTree = p;
            else
                gp->rightTree = p;
        }

    }

    void rotate_left(Node *p){
        if(p->parent == NULL){
            root = p;
            return;
        }
        Node *gp = p->grandparent();
        Node *fa = p->parent;
        Node *y = p->leftTree;

        fa->rightTree = y;

        if(y != NIL)
            y->parent = fa;
        p->leftTree = fa;
        fa->parent = p;
```

```
    if(root == fa)
        root = p;
    p->parent = gp;

    if(gp != NULL){
        if(gp->leftTree == fa)
            gp->leftTree = p;
        else
            gp->rightTree = p;
    }
}

void inorder(Node *p){
    if(p == NIL)
        return;

    if(p->leftTree)
        inorder(p->leftTree);

    cout << p->value << " ";

    if(p->rightTree)
        inorder(p->rightTree);
}

string outputColor (bool color) {
    return color ? "BLACK" : "RED";
}

Node* getSmallestChild(Node *p){
    if(p->leftTree == NIL)
        return p;
    return getSmallestChild(p->leftTree);
}

bool delete_child(Node *p, int data){
    if(p->value > data){
        if(p->leftTree == NIL){
            return false;
        }
        return delete_child(p->leftTree, data);
    } else if(p->value < data){
        if(p->rightTree == NIL){
            return false;
        }
        return delete_child(p->rightTree, data);
    } else if(p->value == data){
        if(p->rightTree == NIL){
            delete_one_child (p);
            return true;
        }
        Node *smallest = getSmallestChild(p->rightTree);
        swap(p->value, smallest->value);
        delete_one_child (smallest);

        return true;
    }else{
       return false;
     }
}

void delete_one_child(Node *p){
    Node *child = p->leftTree == NIL ? p->rightTree : p->leftTree;
    if(p->parent == NULL && p->leftTree == NIL && p->rightTree == NIL){
        p = NULL;
        root = p;
        return;
    }

    if(p->parent == NULL){
        delete  p;
        child->parent = NULL;
        root = child;
        root->color = BLACK;
        return;
    }

    if(p->parent->leftTree == p){
        p->parent->leftTree = child;
    } else {
        p->parent->rightTree = child;
    }
    child->parent = p->parent;
```

```
        if(p->color == BLACK){
            if(child->color == RED){
                child->color = BLACK;
            } else
                delete_case (child);
        }

        delete p;
    }

    void delete_case(Node *p){
        if(p->parent == NULL){
            p->color = BLACK;
            return;
        }
        if(p->sibling()->color == RED) {
            p->parent->color = RED;
            p->sibling()->color = BLACK;
            if(p == p->parent->leftTree)
                //rotate_left(p->sibling());
                rotate_left(p->parent);
            else
                //rotate_right(p->sibling());
                rotate_right(p->parent);
        }
        if(p->parent->color == BLACK && p->sibling()->color == BLACK
                && p->sibling()->leftTree->color == BLACK && p->sibling()->rightTree->color == BLACK)
{
            p->sibling()->color = RED;
            delete_case(p->parent);
        } else if(p->parent->color == RED && p->sibling()->color == BLACK
                && p->sibling()->leftTree->color == BLACK && p->sibling()->rightTree->color == BLACK)
{
            p->sibling()->color = RED;
            p->parent->color = BLACK;
        } else {
            if(p->sibling()->color == BLACK) {
                if(p == p->parent->leftTree && p->sibling()->leftTree->color == RED
                        && p->sibling()->rightTree->color == BLACK) {
                    p->sibling()->color = RED;
                    p->sibling()->leftTree->color = BLACK;
                    rotate_right(p->sibling()->leftTree);
                } else if(p == p->parent->rightTree && p->sibling()->leftTree->color == BLACK
                        && p->sibling()->rightTree->color == RED) {
                    p->sibling()->color = RED;
                    p->sibling()->rightTree->color = BLACK;
                    rotate_left(p->sibling()->rightTree);
                }
            }
            p->sibling()->color = p->parent->color;
            p->parent->color = BLACK;
            if(p == p->parent->leftTree){
                p->sibling()->rightTree->color = BLACK;
                rotate_left(p->sibling());
            } else {
                p->sibling()->leftTree->color = BLACK;
                rotate_right(p->sibling());
            }
        }
    }

    void insert(Node *p, int data){
        if(p->value >= data){
            if(p->leftTree != NIL)
                insert(p->leftTree, data);
            else {
                Node *tmp = new Node();
                tmp->value = data;
                tmp->leftTree = tmp->rightTree = NIL;
                tmp->parent = p;
                p->leftTree = tmp;
                insert_case (tmp);
            }
        } else {
            if(p->rightTree != NIL)
                insert(p->rightTree, data);
            else {
                Node *tmp = new Node();
                tmp->value = data;
                tmp->leftTree = tmp->rightTree = NIL;
                tmp->parent = p;
                p->rightTree = tmp;
```

```
                insert_case (tmp);
            }
        }
    }

    void insert_case(Node *p){
        if(p->parent == NULL){
            root = p;
            p->color = BLACK;
            return;
        }
        if(p->parent->color == RED){
            if(p->uncle()->color == RED) {
                p->parent->color = p->uncle()->color = BLACK;
                p->grandparent()->color = RED;
                insert_case(p->grandparent());
            } else {
                if(p->parent->rightTree == p && p->grandparent()->leftTree == p->parent) {
                    rotate_left(p);
                    p->color = BLACK;
                    p->leftTree->color = p->rightTree->color = RED;
                } else if(p->parent->leftTree == p && p->grandparent()->rightTree == p->parent) {
                    rotate_right(p);
                    p->color = BLACK;
                    p->leftTree->color = p->rightTree->color = RED;
                } else if(p->parent->leftTree == p && p->grandparent()->leftTree == p->parent) {
                    p->parent->color = BLACK;
                    p->grandparent()->color = RED;
                    rotate_right(p->parent);
                } else if(p->parent->rightTree == p && p->grandparent()->rightTree == p->parent) {
                    p->parent->color = BLACK;
                    p->grandparent()->color = RED;
                    rotate_left(p->parent);
                }
            }
        }
    }

    void DeleteTree(Node *p){
        if(!p || p == NIL){
            return;
        }
        DeleteTree(p->leftTree);
        DeleteTree(p->rightTree);
        delete p;
    }
public:

    bst() {
        NIL = new Node();
        NIL->color = BLACK;
        root = NULL;
    }

    ~bst() {
        if (root)
            DeleteTree (root);
        delete NIL;
    }

    void inorder() {
        if(root == NULL)
            return;
        inorder (root);
        cout << endl;
    }

    void insert (int x) {
        if(root == NULL){
            root = new Node();
            root->color = BLACK;
            root->leftTree = root->rightTree = NIL;
            root->value = x;
        } else {
            insert(root, x);
        }
    }

    bool delete_value (int data) {
        return delete_child(root, data);
    }
private:
```

```
    Node *root, *NIL;
};
```

渐近边界的证明

包含n个内部节点的红黑树的高度是$\mathrm{O}(\log n)$。

定义：

- $h(v)$表示以节点v为根的子树的高度。
- $bh(v)$表示从v到子树中任何叶子的黑色节点的数目（如果v是黑色则不计数它，也叫做黑色高度）。

引理：以节点v为根的子树有至少$2^{bh(v)}-1$个内部节点。

引理的证明（通过归纳高度）：

基础：$h(v)=0$

如果v的高度是零则它必定是NIL，因此$bh(v)=0$。所以：

$$2^{bh(v)}-1=2^0-1=1-1=0$$

归纳假设：$h(v)=k$的v有$2^{bh(v)-1}-1$个内部节点暗示了h(v') = k+1的v'有$2^{bh(v')}-1$个内部节点。

因为v'有h(v')> 0所以它是个内部节点。同样的它有黑色高度要么是bh(v')要么是bh(v')-1（依据v'是红色还是黑色）的两个儿子。通过归纳假设每个儿子都有至少$2^{bh(v')-1}-1$个内部接点，所以v'有：

$$2^{bh(v')-1}-1+2^{bh(v')-1}-1+1=2^{bh(v')}-1$$

个内部节点。

使用这个引理我们现在可以展示出树的高度是对数性的。因为在从根到叶子的任何路径上至少有一半的节点是黑色（根据红黑树性质4），根的黑色高度至少是$\frac{h(root)}{2}$。通过引理我们得到：

$$n\geqslant 2^{\frac{h(root)}{2}}-1\leftrightarrow\ \log(n+1)\geqslant\frac{h(root)}{2}\leftrightarrow\ h(root)\leqslant 2\log(n+1)$$

因此根的高度是$\mathrm{O}(\log n)$。

参见

- AVL樹
- B樹
- dancing tree
- 伸展樹
- 2-3-4树
- Treap

引用

- Mathworld: Red-Black Tree (http://mathworld.wolfram.com/Red-BlackTree.html)
- San Diego State University: CS 660: Red-Black tree notes (http://www.eli.sdsu.edu/courses/fall95/cs660/notes/RedBlackTree/RedBlack.html#RTFToC2), by Roger Whitney
- Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms.* Massachusetts: The MIT Press, 2002. pp273-77. ISBN 0-07-013151-1

外部链接

- An applet + quick explanation (http://www.ibr.cs.tu-bs.de/lehre/ss98/audii/applets/BST/RedBlackTree-Example.html)
- Red/Black Tree Demonstration (http://gauss.ececs.uc.edu/RedBlack/redblack.html)
- An example (https://web.archive.org/web/20050706052539/http://www.aisee.com/anim/maple.htm)（animated GIF, 200KB）
- An example (https://web.archive.org/web/20050703001958/http://www.aisee.com/gallery/graph7.htm)（static picture）
- Another explanation (https://web.archive.org/web/20050621072749/http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/red_black.html)（pictures, source code, and Java interactive animation）
- Red-Black Tree Demonstration (https://www.webcitation.org/query?id=1256547034536341&url=geocities.com/dmh2000/articles/code/red-blacktree.html) by David M. Howard
- RBT: A SmallEiffel Red-Black Tree Library (http://efsa.sourceforge.net/archive/durian/red_black_tree.htm)
- libredblack: A C Red-Black Tree Library (http://libredblack.sourceforge.net/)
- Red-Black Tree C++ Code (https://web.archive.org/web/20081025221615/http://web.mit.edu/~emin/www/source_code/cpp_trees/index.html)
- Red-Black Trees (https://web.archive.org/web/20050719170536/http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/niemann/s_rbt.htm) by Thomas Niemann
- 红黑树的介绍和实现 (https://web.archive.org/web/20120320021445/http://saturnman.blog.163.com/blog/static/557611201096942038 3/)
