红黑树插入算法实现原理分析

ziwenxie.site/2017/03/20/algorithm-red-black-tree/

引言

红黑树是在实际工程中被广泛应用的一种数据结构，比如Linux中的线程调度就是使用的红黑树来管理进程控制块，而Nginx中也是使用红黑树来管理的timer，Java中的TreeMap和TreeSet也是基于红黑树来实现的。红黑树相比普通二叉查找树的一个优势就是它的树高为~lgN，所以不管是查找/插入/删除操作它均能保证能够在对数时间之内完成。本文我们就先来了解一下红黑树插入算法的实现。

红黑树的定义

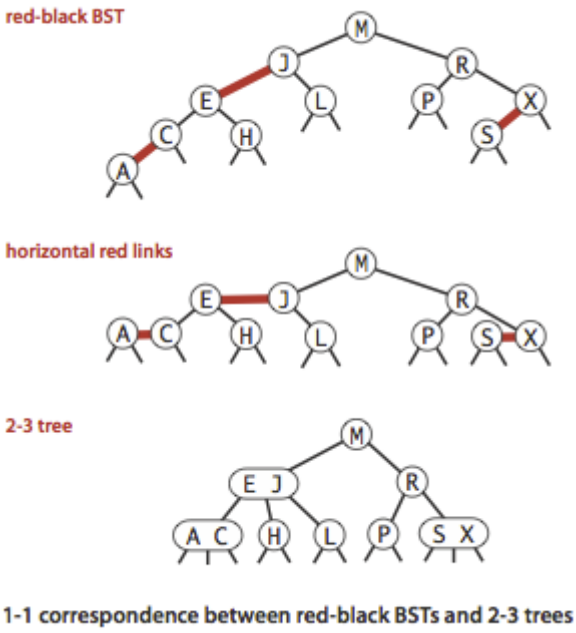
红黑树可以定义成含有红黑链接并且满足下列条件的二叉查找树：

1. 红链接均为左链接。
2. 没有任何一个节点同时和两条红链接相连。
3. 任意空链接到根节点的路径上的黑链接数目相同。p.s: 我们将指向一棵空树的链接称为空链接。

比如下图就是一棵典型的红黑树，如果之前了解过2-3树的话(不了解也没有关系，我们下面的内容并不会涉及到2-3树)，可以发现如果将红黑树中的红色结点画平，实际上它就是2-3树的一种变形，不过相比2-3树，红黑树的查找操作要简单的多，但它同时也结合了2-3树中高效的插入操作，所以说红黑树其实是普通的二叉查找树和2-3树两种数据结构优点的结合。

红黑树的定义

在实现红黑树之前我们先来定义一棵红黑树：

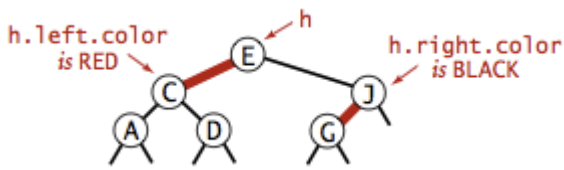


1-1 correspondence between red-black BSTs and 2-3 trees

```
public class RedBlackLiteBST<Key extends
Comparable<Key>, Value> {

    private static final boolean RED   =
true;

    private static final boolean BLACK =
false;

    private Node root;

    private int n;


    private class Node {

        private Key key;

        private Value val;

        private Node left, right;

        private boolean color;

        public Node(Key key, Value val,
boolean color) {

            this.key = key;

            this.val = val;

            this.color = color;

        }

    }

}
```

在上面的代码中，我们将链接的颜色保存在表示该结点的Node对象中的color变量中。如果指向它的链接是红色的，那么该变量为true，黑色则为false，我们规定空链接都为黑色。如下图所示，指向结点C的链接是红色，那么我们就将h.left.color设置为红色，指向结点J的链接是黑色的，我们就将h.right.color设置为黑色。

红黑树的颜色表示



红黑树的几种基本操作

红黑树相比普通的二叉查找树的一个重要优势就是插入的高效性，但是正因为如此红黑树的插入操作的算法实现相比普通的二叉树要复杂一些。在正式实现插入算法之前，我们有必要先了解一下对于红黑树的几种基本操作。

左旋转

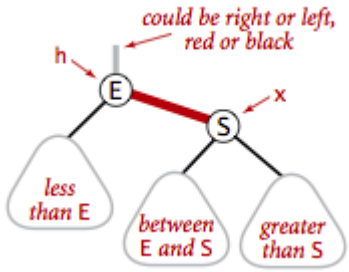
如下图所示，现在我们有一条红色的右链接，现在我们想要将这条红色右链接转换为左链接，这个操作过程就叫做左旋转：

左旋转之前

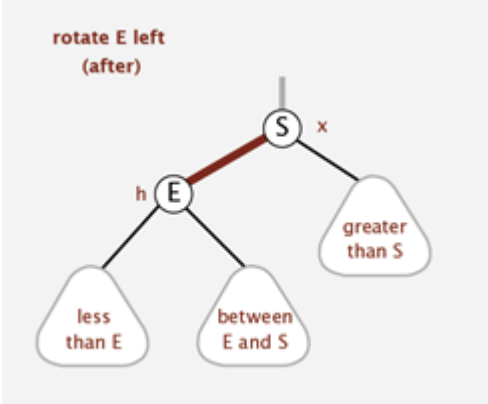
我们要做的就是在保持红黑树平衡性的同时，将上图的结构变为下面这样：

左旋转之后

仔细观察可以发现，我们要实现的其实就是将红色链接关联的两个节点中由较小的节点E作为根节点转换成由较大的节点S作为根节点。同时在这个过程中为了保持二叉树中左子树都要小于根节点，右子树都要大于根节点，所以如果S节点还存在的话左链接我们还需要将它变成E节点的右链接。具体的实现代码如下：




```
private Node rotateLeft(Node h) {
    assert (h != null) && isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```



右旋转

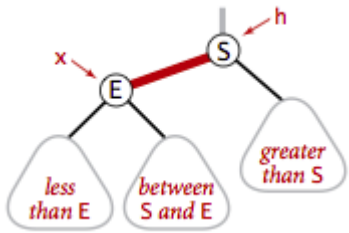
右旋转的实现和左旋转的实现是类似的，就是将一个红色左链接转换成一个红色右链接：

右旋转之前

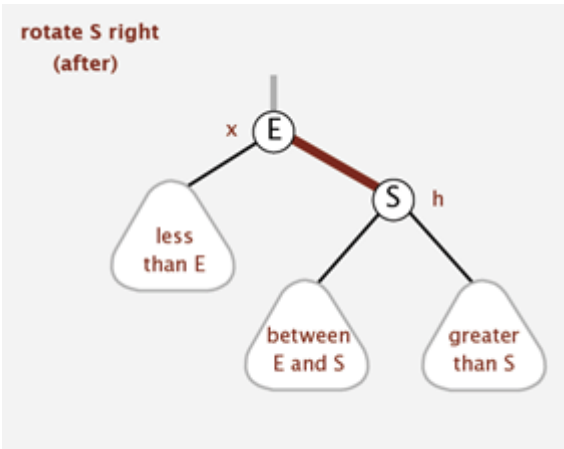
与左旋转的时候相反，我们要做的其实就是红色链接关联的两个节点中较大的节点S作为根节点转换成由较小的节点E作为根节点：

右旋转之后

所以转换成具体的代码，我们只需要将left和right相互转换就行了：




```
private Node rotateRight(Node h) {
    assert (h != null) && isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

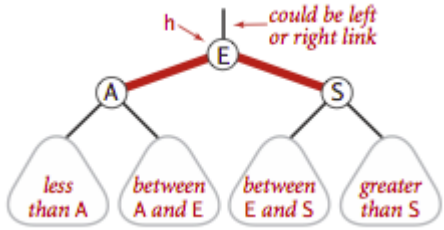


颜色转换

上面我们提到红黑树的一个重要特性就是红链接均为做左链接，所以对于下面这种情形，如果一个结点的两个子结点均为红色链接，我们就将子结点的颜色全部由红色转换成黑色
，同时将父结点的颜色由黑变红。

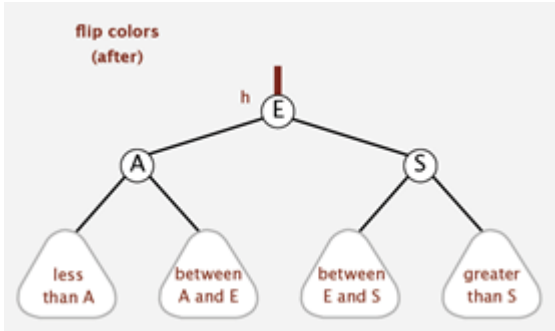
颜色转换之前

颜色转换之后



具体的实现代码非常简单：

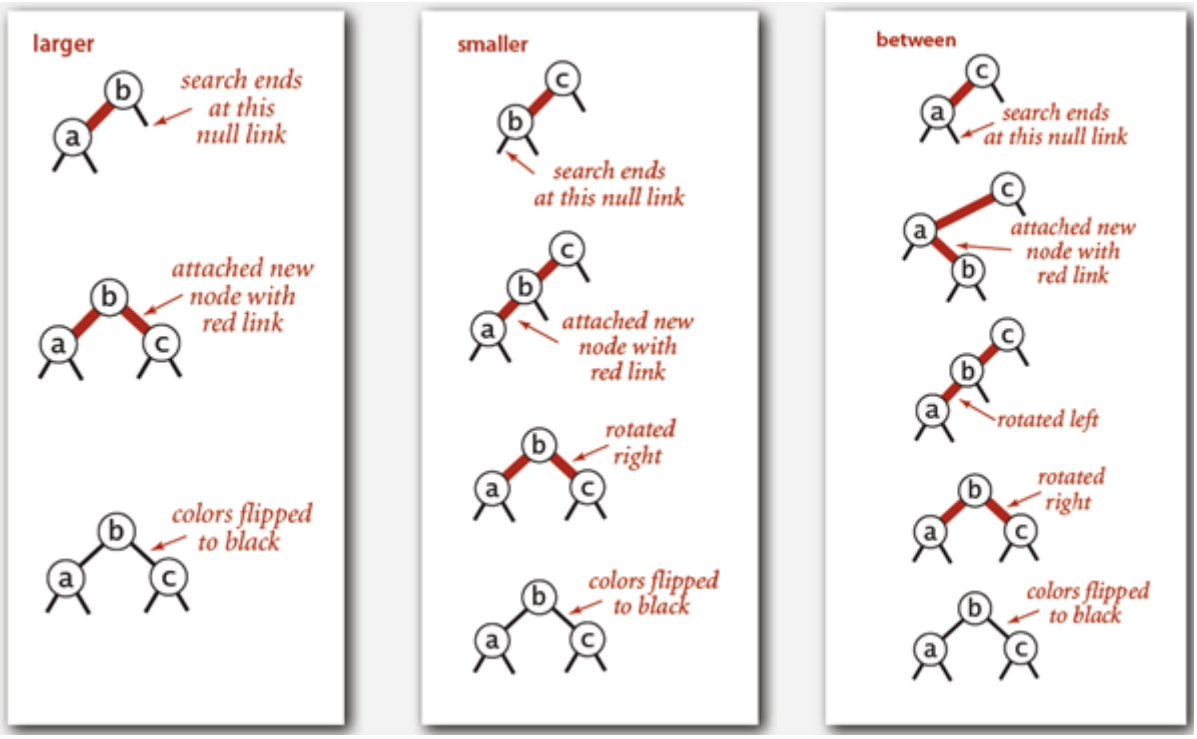
```
private void flipColors(Node h) {
    assert !isRed(h) && isRed(h.left) &&
isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```



插入操作的实现

上面说了这么多，其实都是为接下来的插入操作做的预热。接下来结合下图，我们来分析红黑树插入操作过程我们会遇到的三种情形：

1. 插入的新结点c大于树中现存的两个键，所以我们要将它连接到b结点的右链接。因为这个时候b的两条链接都是红链接，所以我们要进行flipColors。接下来可以发现，我们下面的两种情况都会转换成这种情形。
2. 插入的新结点a要比树中现存的两个键都要小，所以我们先将它连接到结点b的左链接，前面我们提到红黑树的一个特性就是没有任何一个节点可以同时和两个红色链接相连，而现在b结点却违背了这一原则，所以我们要进行右旋转操作，接下来情形1是一模一样的了。
3. 插入的新结点b在树中现存的两个键之间，所以我们先将它连接到结点a的右链接，前面我们提到红黑树中红色链接都是左链接，所以我们首先要进行左旋转操作，接下来就和情形2一模一样了。



插入操作的3种情形

插入操作的具体实现代码如下，下面的代码直到`// fix-up any right-leaning right`之前我们做的都是为了找到合适的插入位置，而之后的3个`if`语句实际上就是对上图中情形3的一种总结。

```
public void put(Key key, Value val) {
    root = insert(root, key, val);
    root.color = BLACK;

}
private Node insert(Node h, Key key, Value val) {
    if (h == null) {
        n++;
        return new Node(key, val, RED);
    }
    int cmp = key.compareTo(h.key);
    if      (cmp < 0) h.left  = insert(h.left,  key, val);
    else if (cmp > 0) h.right = insert(h.right, key, val);
    else              h.val   = val;

    if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
    if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);
    return h;
}
```

`isRed`的实现非常简单，我就不解释了：

```
private boolean isRed(Node x) {
    if (x == null) return false;
    return x.color == RED;
}
```

References

ALGORITHM 4TH