

一篇文章搞懂红黑树的原理及实现



六尺帐篷 (/u/f8e9b1c246f1)
2017.07.13 00:58 字数 4017 阅读 669 评论 0 喜欢 8
(/u/f8e9b1c246f1)

[编辑文章 \(/writer#/notebooks/14296477/notes/14472679\)](#)

2-3-4 Tree(2-3-4树)

二叉查找树 (Binary Search Tree, 简称BST) 是一棵二叉树, 它的左子节点的值比父节点的值要小, 右节点的值要比父节点的值大。它的高度决定了它的查找效率。
我们知道二叉查找树。每个节点只可以有有一个key, 而2-3-4树就是将节点的key的数量增加, 可以有多个key, 并且2-3-4树可以保持完美平衡 (Perfect balance. Every path from root to leaf has same length)

什么是完美平衡 (Perfect balance) ? 实际上就是每条从根节点到叶节点的路径的高度都是一样的 (Every path from root to leaf has same length)

2-3-4树的名字是根据子节点数来确定的。
我们看2-3-4树的key的种类。

- 2-node: one key, two children.一个key值, 两个儿子节点
- 3-node: two keys, three children。两个key值, 三个儿子节点
- 4-node: three keys, four children.三个key值, 四个儿子节点

其中2-node的左孩子就代表比key小, 右孩子就代表比key大,
3-node的左孩子代表比第一个key小, 中间的孩子代表值位于第一个key和第二个key之间, 右孩子代表值大于第二个key
4-node同理

我们看一棵2-3-4树的例子

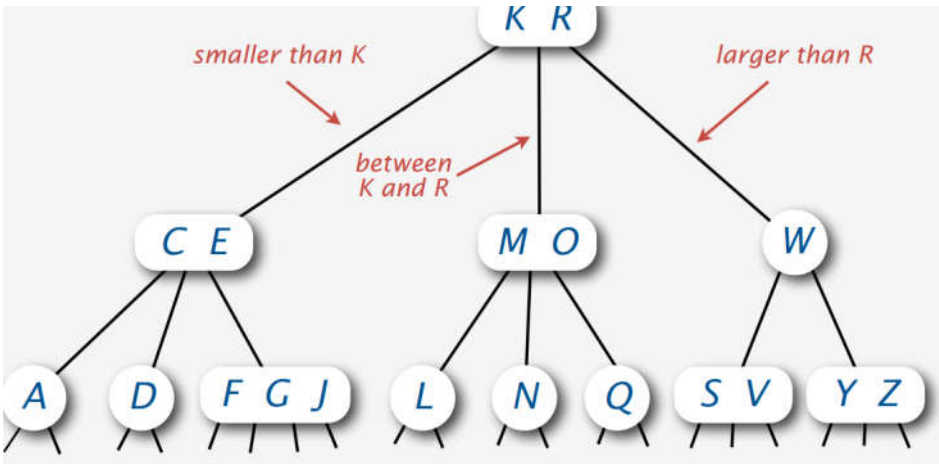


image.png

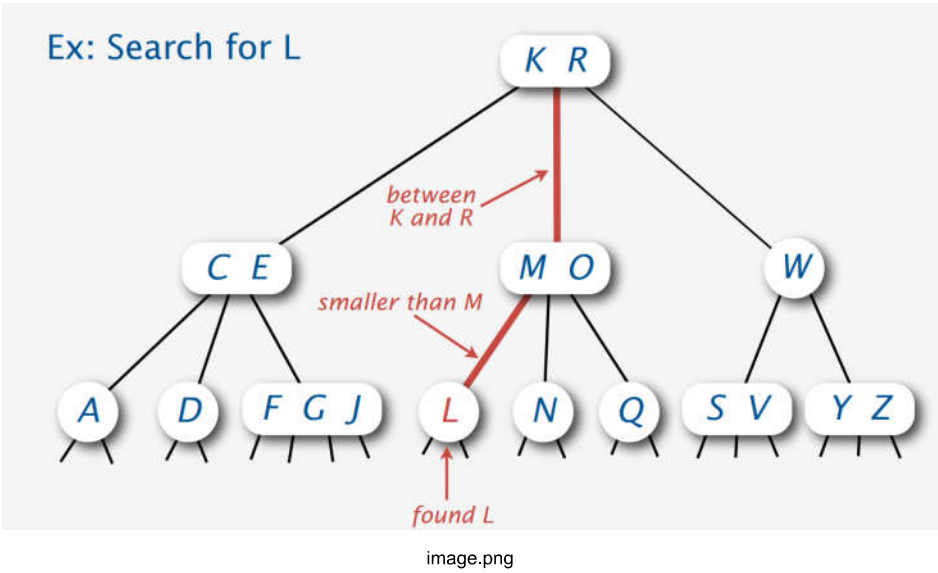
2-3-4树的查找 (Search in a 2-3-4 Tree)

2-3-4的查找类似于BST的查找, 只要递归找到所在的子树就可以。

- 比较要查找的key与当前节点中的key值
- 根据key值选择要查找的key所存在的子树区间



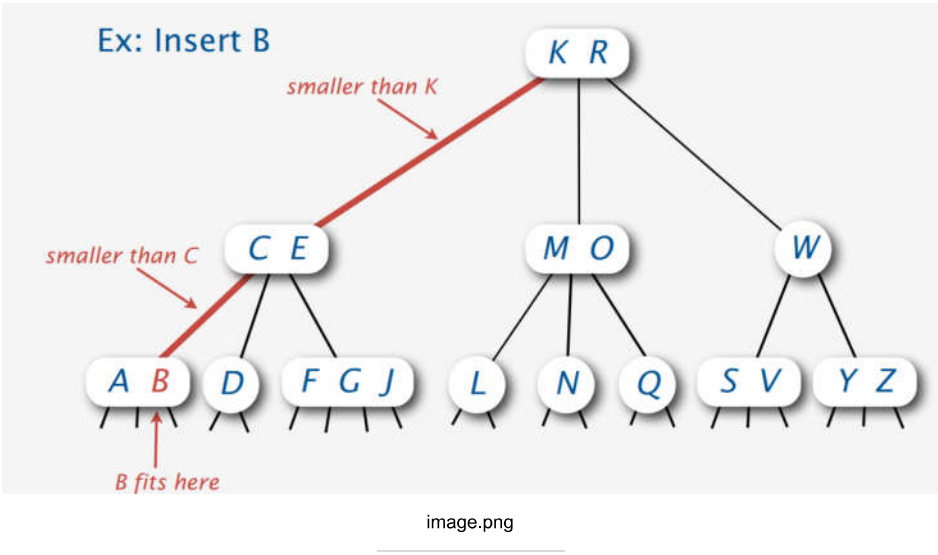
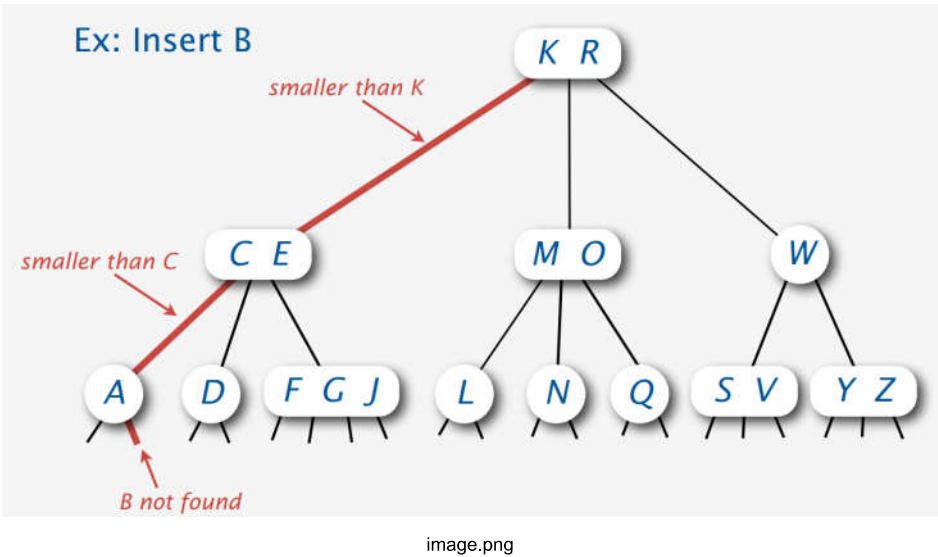
- 重复上述步骤（递归实现），直到查找到key



2-3-4的插入 (Insertion in a 2-3-4 Tree)

2-3-4的插入有几种情况，下面我们会一一的讨论。

首先，如果是向一个2-node插入节点的话，那么直接将它转换为3-node就可以了。



📄

🔖

🔗

如果我们是向3-node插入，就将3-node变成4-node即可

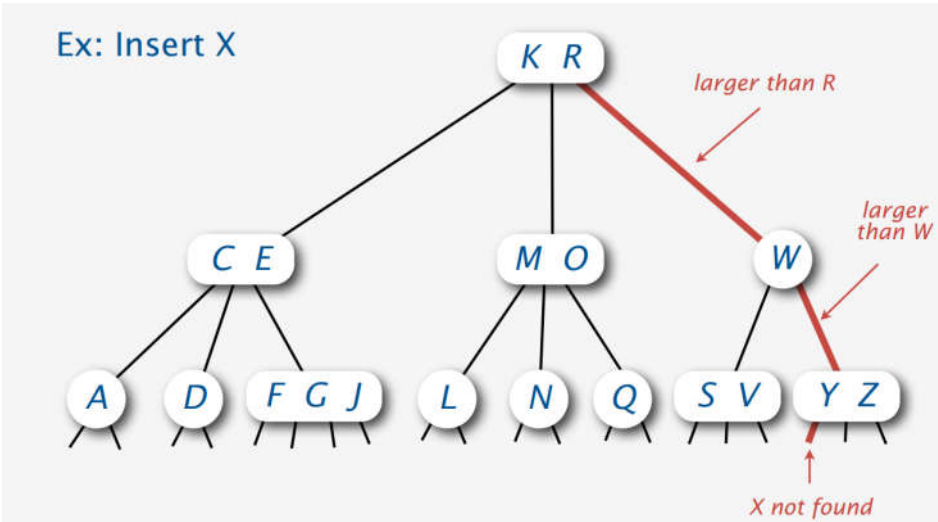


image.png

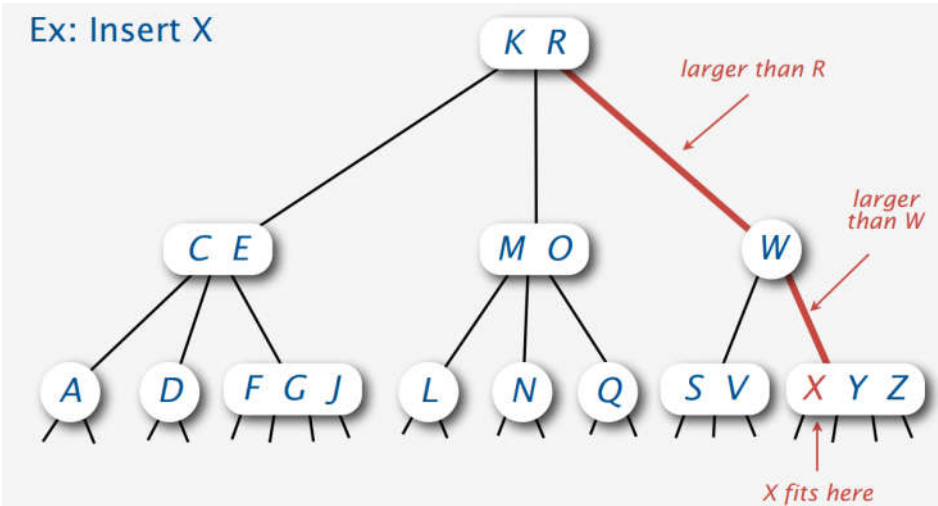


image.png

那么问题就来了，如果我们向4-node插入呢？
显然我们无法直接插入，因为4-node已经是最大的节点了。
这时，我们就需要对节点进行一些变换

最常用的方法就是将4-node的中间节点向上移动，移动到父节点中。这样就可以插入
了。

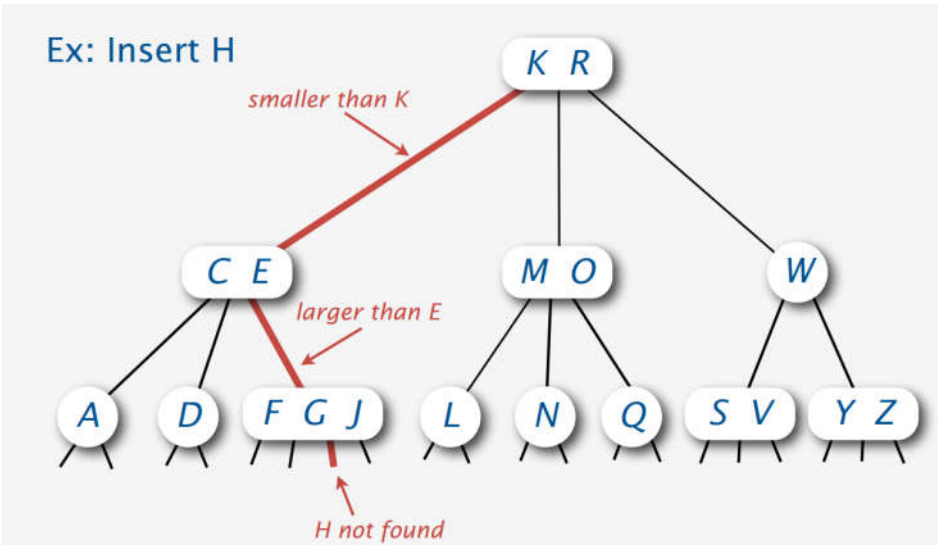


image.png

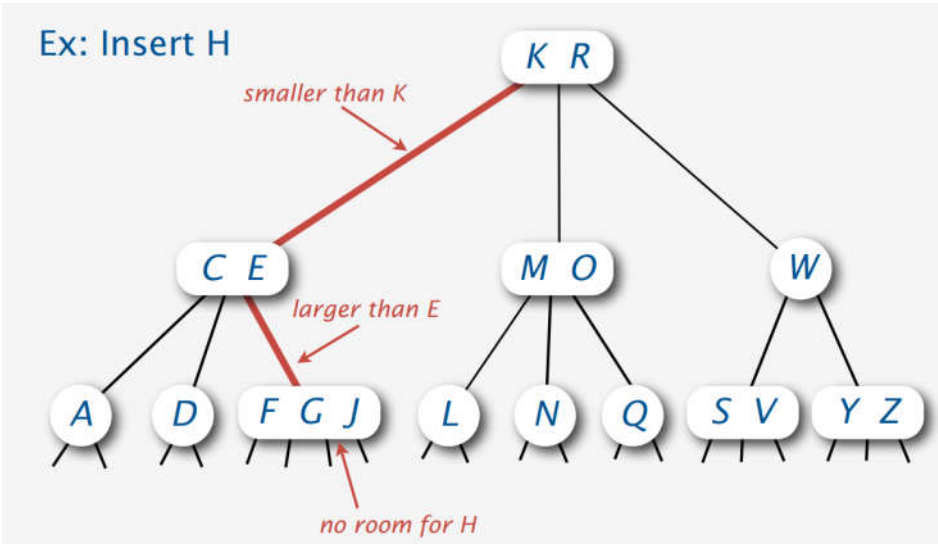


image.png

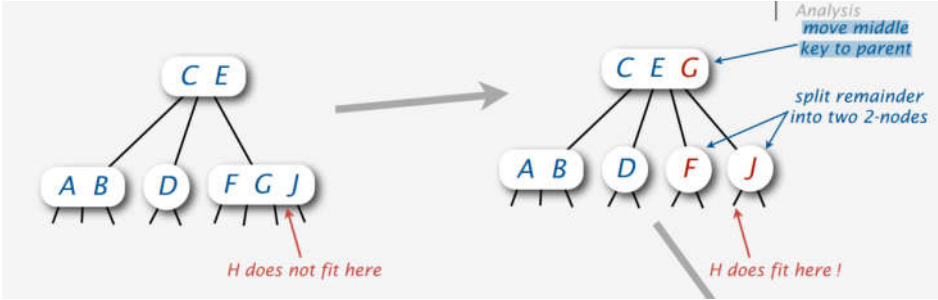


image.png



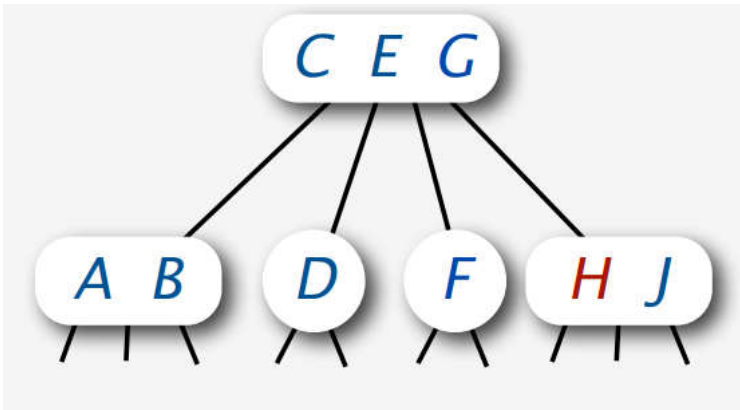


image.png

但这个方法有一个问题，就是如果父节点是4-node，那么就无法切分中间节点了。

一种解决的方法就是ensures that the “current” node is not a 4-node，我们做出保证父节点永远不会是4-node，就是说4-node不会出现在最后一层以外的层上。

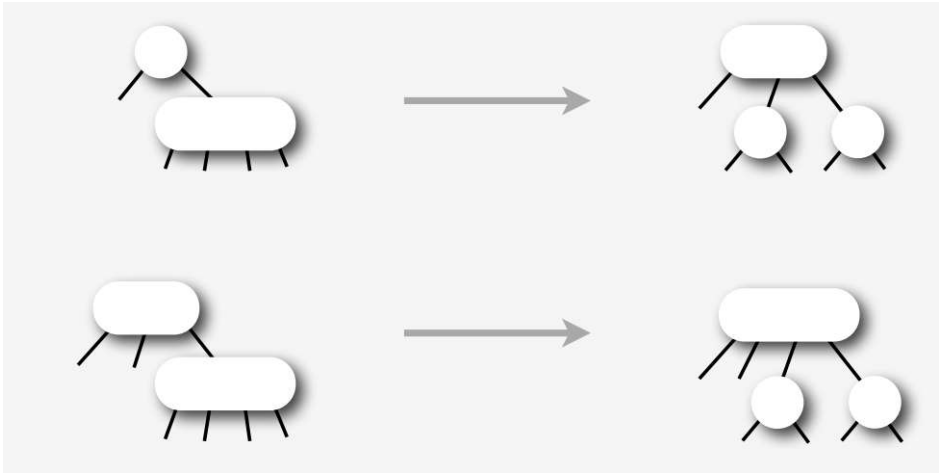


image.png

所以我们需要维持一个不变的条件，当前的父节点永远不会是4-node

也就是，4-node的子节点中又有一个子节点是不可能的
最底部的节点中不可能出现4-node，只可能出现2-node和3-node

下面分别分析切分4-node的几种不同情况

首先，4-node 的父节点是2-node 的情况，同样是把中间节点向上移动



image.png

📄

🔖

🔗

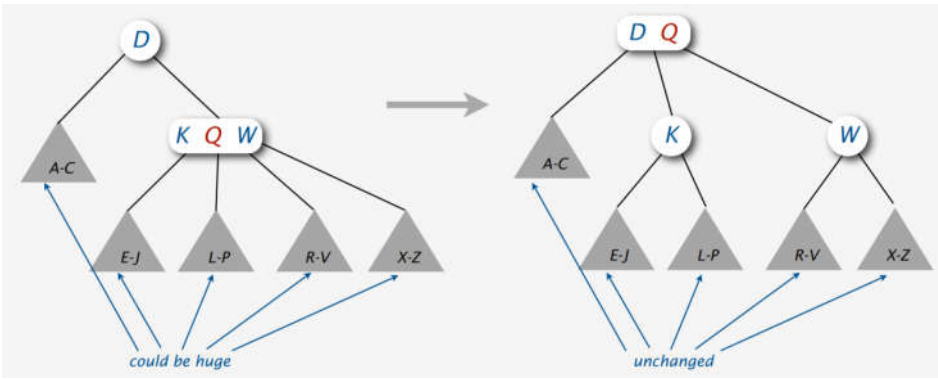


image.png

对于4-node的父节点是3-node的情况，也是同样处理，将middle中间节点向上移动



image.png

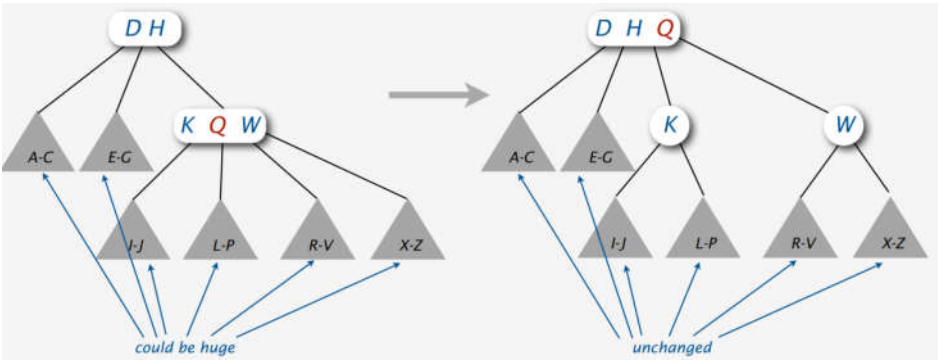


image.png

我们看一下如何从空开始插入建立一个2-3-4树

下面，我们通过动态添加一个完整的2-3-4的过程，说明2-3-4树的插入和构建过程

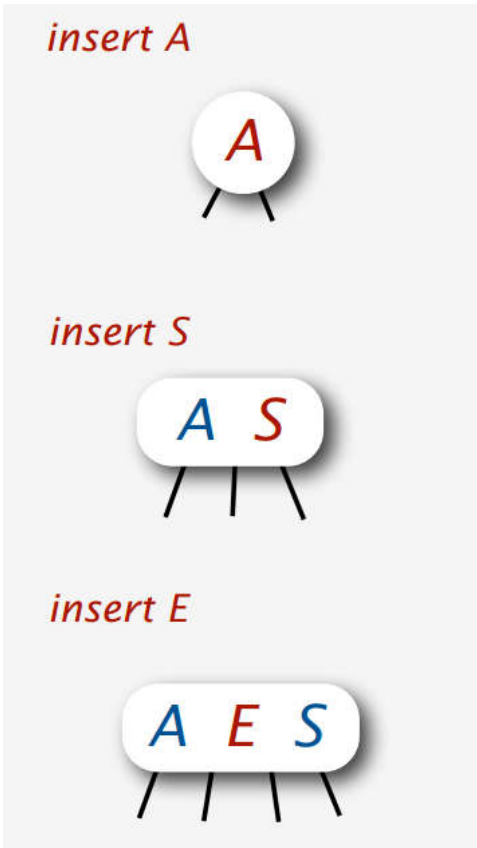


image.png

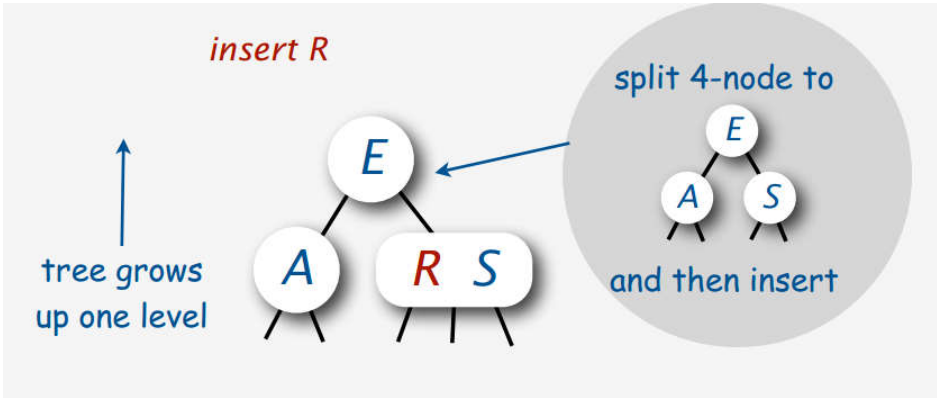


image.png

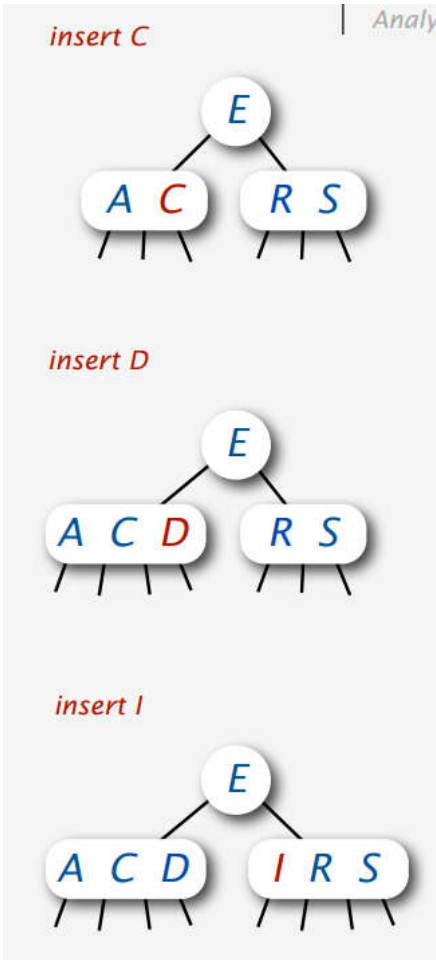


image.png

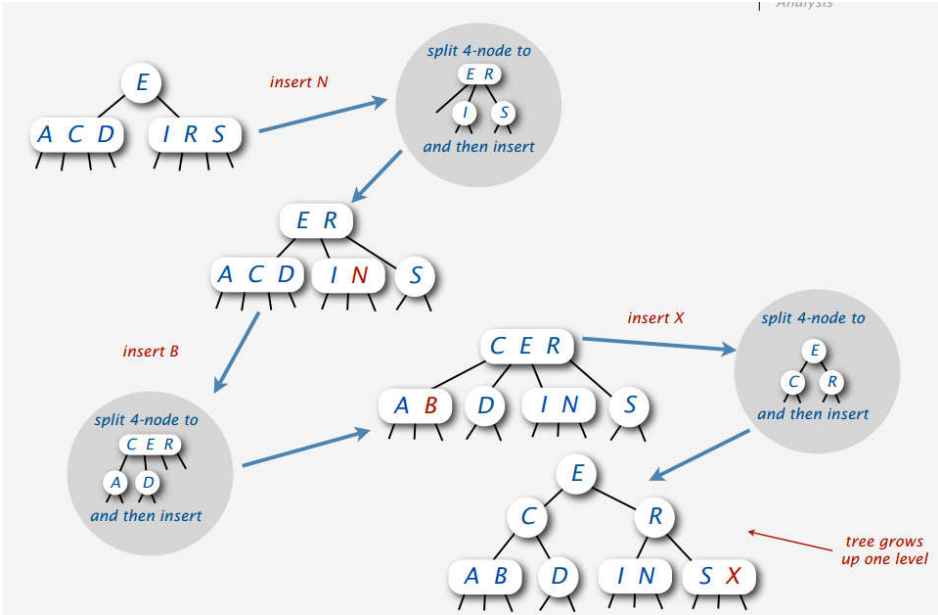


image.png

我们发现，2-3-4树所有叶子节点都在同一个高度上。

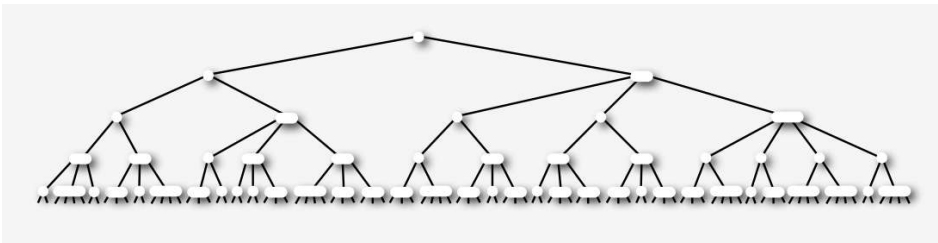


image.png

我们分析2-3-4树的效率：

- 2-3-4的高度的最坏情况（全是2-node），也就相当于演变成了平衡二叉树：相当于平衡二叉树 $\lg N$
- 2-3-4树高度的最好情况（全是4-node）， $\log_4 N = 1/2 \lg N$ （但我们知道这种情况是不可能出现的，因为我们要求4-node的父节点或者子节点不能是4-node）
- 对于100万个节点，2-3-4树的高度会在10~20之间
- 对于10亿的节点，2-3-4树的高度会在15~30之间

由此来看，2-3-4树的效率比平衡二叉树要好，但是问题在于，2-3-4树并不好实现

- 首先，我们需要用三种不同类型的节点代表2-3-4node
- 然后，在插入节点的时候，我们可能需要进行大量的切分4-node的工作
- 我们可能也需要频繁的在三种节点之间进行转换

一个简单的伪码实现：

```
private void insert(Key key, Val val)
{
    Node x = root;
    while (x.getTheCorrectChild(key) != null)
    {
        x = x.getTheCorrectChild(key);
        if (x.is4Node()) x.split();
    }
    if (x.is2Node()) x.make3Node(key, val);
    else if (x.is3Node()) x.make4Node(key, val);
    return x;
}
```

为了更好的利用2-3-4树平衡高度的特点，同时又更好的便于实现，我们就引入了红黑树。

红黑树

- 我们用最常见的平衡二叉树来代表2-3-4树
- 我们通过给节点区分红色和黑色来区分三种不同的节点。（红边指向下的节点为红节点）

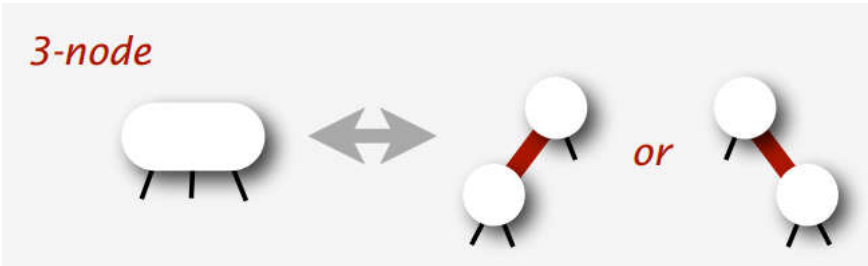


image.png

📄

🔖

🔗



image.png

这样我们就可以用BST来代表2-3-4树了。看下面的例子

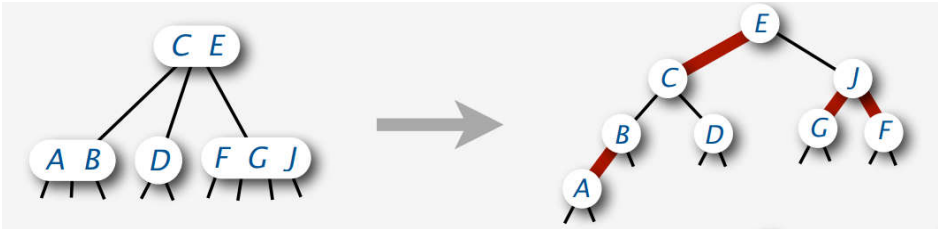


image.png

但是存在一个问题，就是对于一棵2-3-4树可能有多种不同的表示，这是在于对于3-node的表示，红色的边可以向左倾，也可以向右倾。

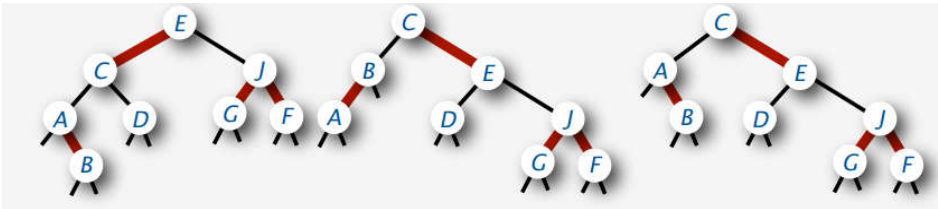


image.png

所以就要考虑很多情况。

但我们在此只考虑左倾的情况，所以这种树也叫做左倾红黑树
这样，对于任何一棵2-3-4树，我们都可以得到一棵唯一对应的左倾红黑树

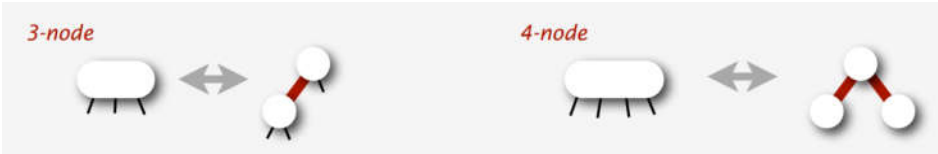


image.png

对于左倾红黑树，我们还有以下要求，就是不能以下情况的节点情况：

- 首先，由于是左倾的，就不能出现右倾的3-node

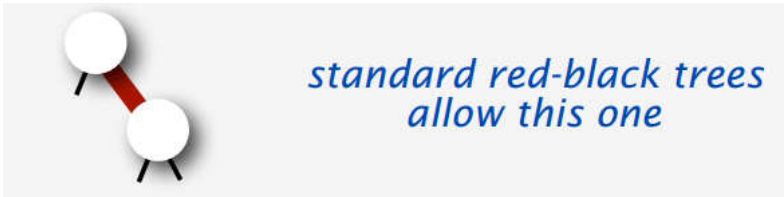


image.png

- 其次，不允许出现两个红边连在一起的情况，变成2-3-4树的情况就是不允许两个3-node相互连接

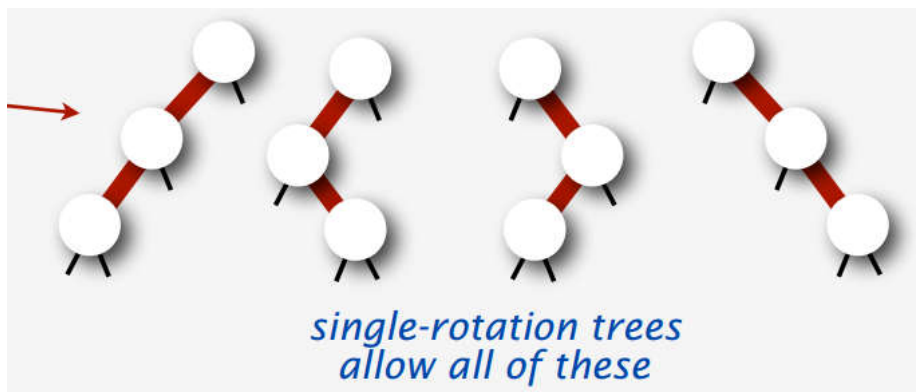


image.png

左倾红黑树的ADT

```
public class BST<Key extends Comparable<Key>, Value>
{
    private static final boolean RED = true;
    private static final boolean BLACK = false;
    private Node root;
    private class Node
    {
        Key key;
        Value val;
        Node left, right;
        boolean color;
        Node(Key key, Value val, boolean color)
        {
            this.key = key;
            this.val = val;
            this.color = color;
        }
    }
    public Value get(Key key)
    // Search method.
    public void put(Key key, Value val)
    // Insert method.
    }

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

红黑树的get方法和BST是一样的

```
public Value get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if (cmp == 0) return x.val;
        else if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
    }
    return null;
}

public Key min()
{
    Node x = root;
    while (x != null) x = x.left;
    if (x == null) return null;
    else return x.key;
}
```

左倾红黑树的插入

插入操作是红黑树中最复杂的操作之一。因为不仅要插入还要维持红黑颜色的。



首先，我们先介绍如何对红黑树的一些节点进行转换操作

- 左旋操作
左旋操作就是将右倾的3-node变成左倾的3-node

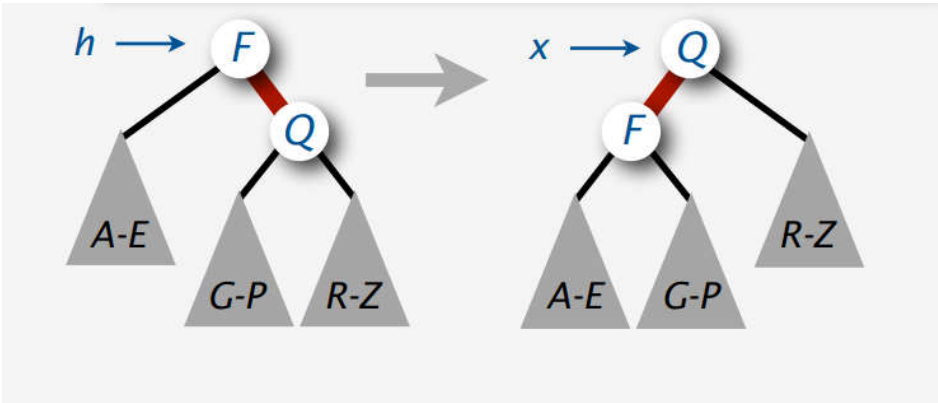


image.png

```
private Node rotateLeft(Node h)
{
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = x.left.color;
    x.left.color = RED;
    return x;
}
```

- 右旋操作
就是与左旋操作相反

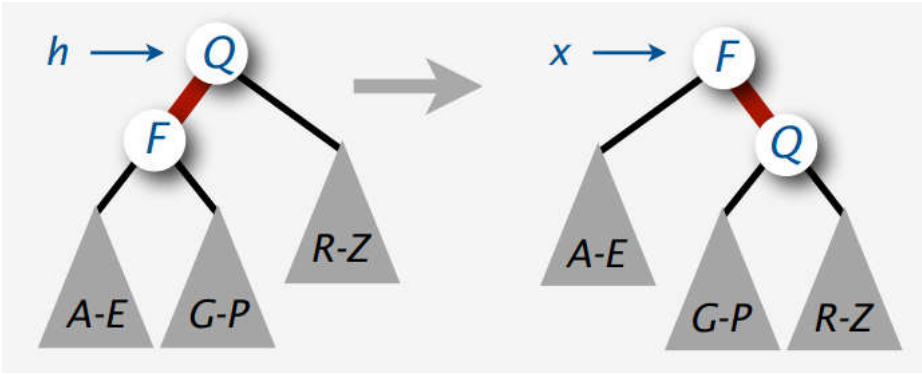


image.png

```
private Node rotateRight(Node h)
{
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = x.right.color;
    x.right.color = RED;
    return x;
}
```

下面我们来具体分析插入操作

当我们要向红黑树的底部插入一个节点的时候，就可能出现多种情况

如果我们向2-node的节点插入的话，有两种情况，如果插入左孩子，那么直接插入就可以，但如果插入的是右孩子，为了保持左倾，插入之后，我们需要进行一个左旋操作

📄

🔖

🔗



image.png

我们可以看到这种情况对应于2-3-4树就是想2-node插入变成3-node

下面一种情况，就是我们向3-node插入一个节点，那么我们就需要将它变成2-3-4树中对应的树节点

这也是为什么我们之前定义的不允许的情况中的第二种，不允许两条红边连在一起，也就是不允许两个红节点互为父子节点，因为插入的节点一定是红节点。

向3-node插入有三种情况：

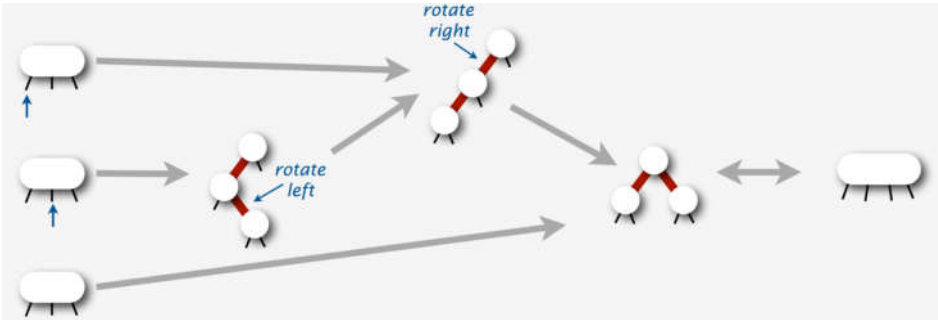


image.png

向4-node插入：

根据我们之前在2-3-4树中学习的可以知道，我们需要对4-node进行切分，切分的方法就是将4-node的中间节点向上移动到父节点中。

首先，当父节点是2-node时候：

有两种情况

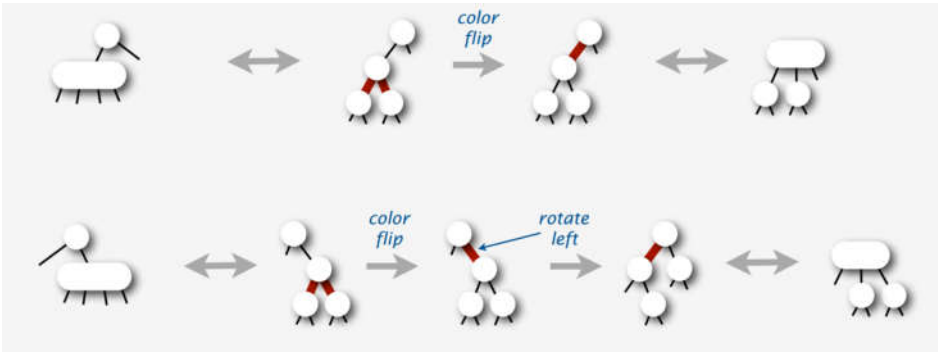


image.png

我们发现在红黑树中进行切分工作很简单，只要将两个红节点变成黑，然后父节点变成红就可以了。这个变换的过程，我们叫做 color flip。

代码如下：

📄

🔖

🔗

```
private Node colorFlip(Node h)
{
    x.color = !x.color;
    x.left.color = !x.left.color;
    x.right.color = !x.right.color;
    return x;
}
```

过程如下图

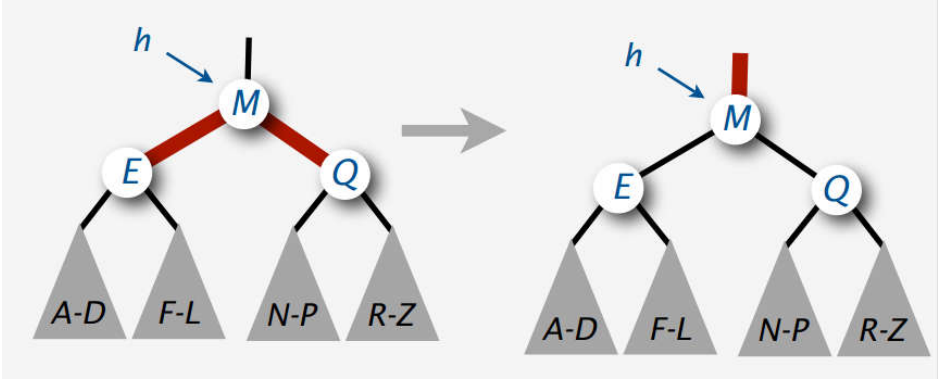


image.png

对于父节点为3-node的情况:

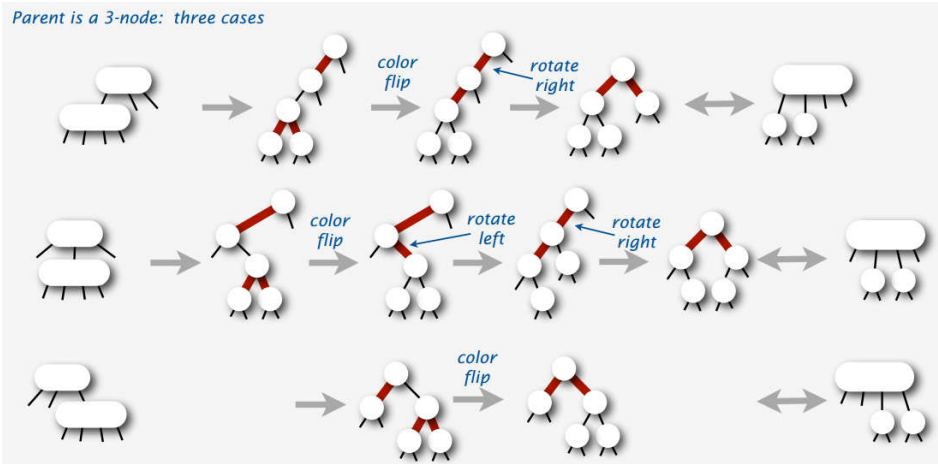


image.png

观察这五种情况，我们发现首先都是先惊醒color flip操作，然后就变成了之前的操作，左旋和右旋。

我们可以把上面这些插入操作总结，然后实现一个统一适用的插入算法

- 首先，向空节点插入一个节点，一定为红节点

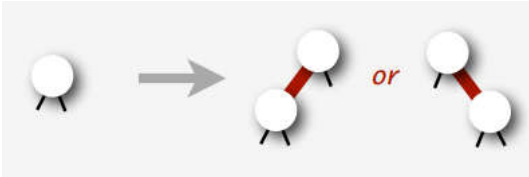


image.png

```
if (h == null)
    return new Node(key, value, RED);
```

- 如果出现了4-node的情况，我们我们就进行color flip

📄

🔖

🔗

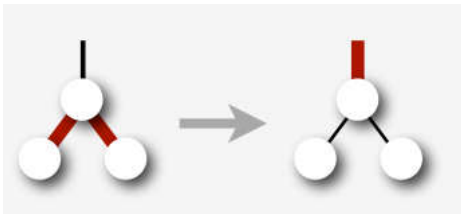


image.png

```
if (isRed(h.left) && isRed(h.right))
    colorFlip(h);
```

- 调整右倾的节点



image.png

```
if (isRed(h.right))
    h = rotateLeft(h);
```

- 对连续的两个红节点进行转换



image.png

```
if (isRed(h.left) && isRed(h.left.left))
    h = rotateRight(h);
```

左倾红黑树插入算法的实现

```
if (h == null)
    return new Node(key, val, RED);
if (isRed(h.left) && isRed(h.right))
    colorFlip(h);
int cmp = key.compareTo(h.key);
if (cmp == 0) h.val = val;
else if (cmp < 0)
    h.left = insert(h.left, key, val);
else
    h.right = insert(h.right, key, val);
if (isRed(h.right))
    h = rotateLeft(h);
if (isRed(h.left) && isRed(h.left.left))
    h = rotateRight(h);
return h;
```

这里代码的执行顺序是很重要的。

比如如果我们把colorflip移到最后，那么会出现什么情况？

📄

🔖

🔗

由于每次在最后都将4-node 进行color flip了，那么自然红黑树中不存在4-node了，所以就变成了2-3树的红黑树

我们可以对比普通红黑树的插入算法的实现

```
private Node insert(Node x, Key key, Value val, boolean sw)
{
    if (x == null)
        return new Node(key, value, RED);
    int cmp = key.compareTo(x.key);
    if (isRed(x.left) && isRed(x.right))
    {
        x.color = RED;
        x.left.color = BLACK;
        x.right.color = BLACK;
    }
    if (cmp == 0) x.val = val;
    else if (cmp < 0)
    {
        x.left = insert(x.left, key, val, false);
        if (isRed(x) && isRed(x.left) && sw)
            x = rotR(x);
        if (isRed(x.left) && isRed(x.left.left))
        {
            x = rotR(x);
            x.color = BLACK; x.right.color = RED;
        }
    }
    else // if (cmp > 0)
    {
        x.right = insert(x.right, key, val, true);
        if (isRed(h) && isRed(x.right) && !sw)
            x = rotL(x);
        if (isRed(h.right) && isRed(h.right.right))
        {
            x = rotL(x);
            x.color = BLACK; x.left.color = RED;
        }
    }
    return x;
}
```

左倾红黑树的删除操作

首先我们介绍一下，删除完成之后，如何调整红黑树为左倾的红黑树？
这里有一个方法，主要就是进行三个调整的步骤

```
private Node fixUp(Node h)
{
    if (isRed(h.right))
        h = rotateLeft(h);
    if (isRed(h.left) && isRed(h.left.left))
        h = rotateRight(h);
    if (isRed(h.left) && isRed(h.right))
        colorFlip(h);
    return h;
}
```

删除操作的原则

- 删除的当前节点不能是2-node
- 如果有必要可以变换成4-node
- 从底部删除节点
- 向上的fix过程中，消除4-node

红黑树的删除操作与插入操作一样，极其复杂，所以先从相对容易的情况开始考虑

删除最大节点

显然最大节点一定是在最右边



如果我们删除的节点在3-node或者4-node中，我们直接删除掉就可以了。



image.png

最复杂的情况，是我们要删除的节点是2-node，如果我们直接删除就会破坏红黑树的平衡，所以我们再删除之前，要进行一定的变换，变成3-node或者4-node，也就是借一个或者两个节点过来。

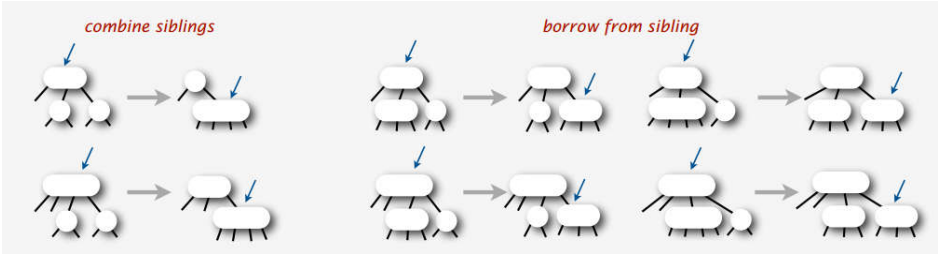


image.png

根据父节点的不同。3-node或者4-node和兄弟节点的不同可以分为六种情况，但其中又可以分为两类

- 第一种处理方法就是兄弟节点不是2-node，就可以直接从兄弟节点借一个节点过来
- 第二种处理方法兄弟节点是2-node，则从父节点中借一个过来，然后和兄弟节点合并成一个4-node

这六种情况的条件根据2-3-4树转换成红黑树，就是h.right和h.right.left均为黑色。但其中有需要分为两种

对于上述提到的第二种处理方法，处理比较简单，直接color flip即可

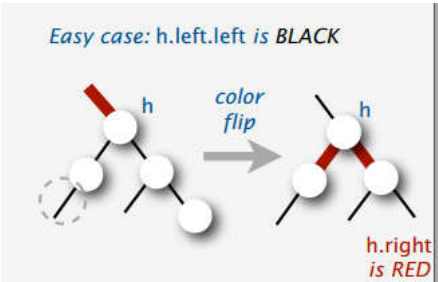


image.png

其中这种情况的条件就是左子节点为2-node，也就是h.left.left为黑。

对于h.left.left为红的情况，就对应上述的第一种处理方法，首先color flip，然后还要借一个节点过来



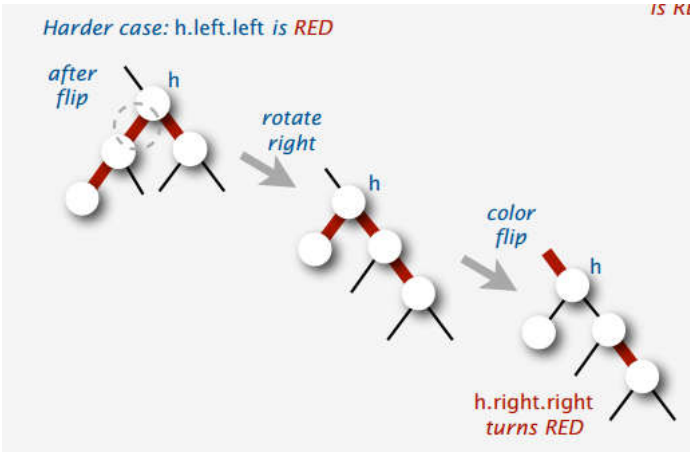


image.png

所以将上面两种方法合并：

```
private Node moveRedRight(Node h)
{
    colorFlip(h);
    if (isRed(h.left.left))
    {
        h = rotateRight(h);
        colorFlip(h);
    }
    return h;
}
```

然后我们就可以得到删除最大节点的算法：

```
public void deleteMax()
{
    root = deleteMax(root);
    root.color = BLACK;
}

private Node deleteMax(Node h)
{
    if (isRed(h.left))
    {
        h = rotateRight(h);
        if (h.right == null)
            return null;
        if (!isRed(h.right) && !isRed(h.right.left))
        {
            h = moveRedRight(h);
            h.left = deleteMax(h.left);
            return fixUp(h);
        }
    }
}
```

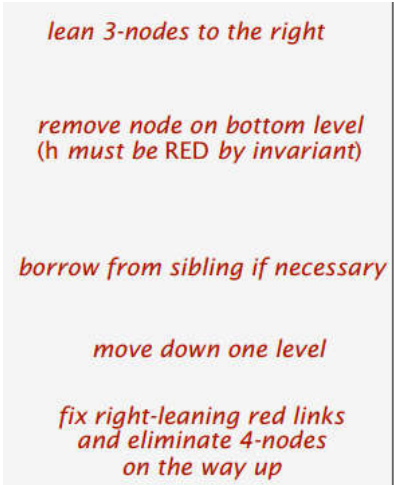


image.png

流程就是：

- 首先如果左旋则变为右旋，因为找最大节点在最右边
- 如果，已经到了最底部，那么直接移除就行，移除的要求是最底部的节点一定是red
- 如果遇到了2-node就借一个节点
- 继续往下递归查找
- 删除完毕，就恢复红黑树

我们下面看两个例子

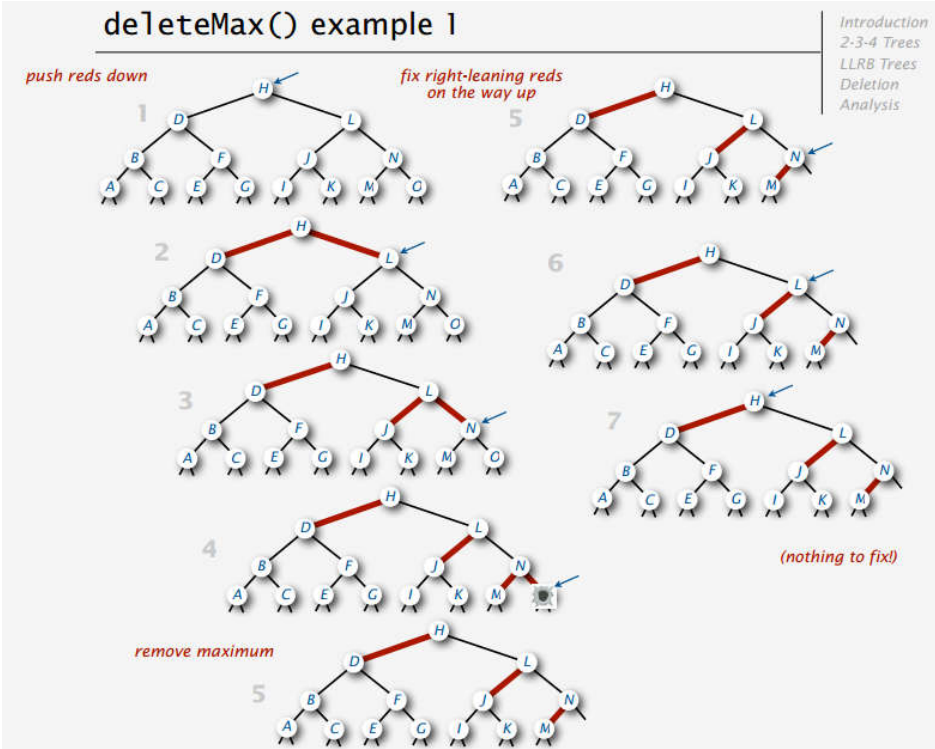


image.png

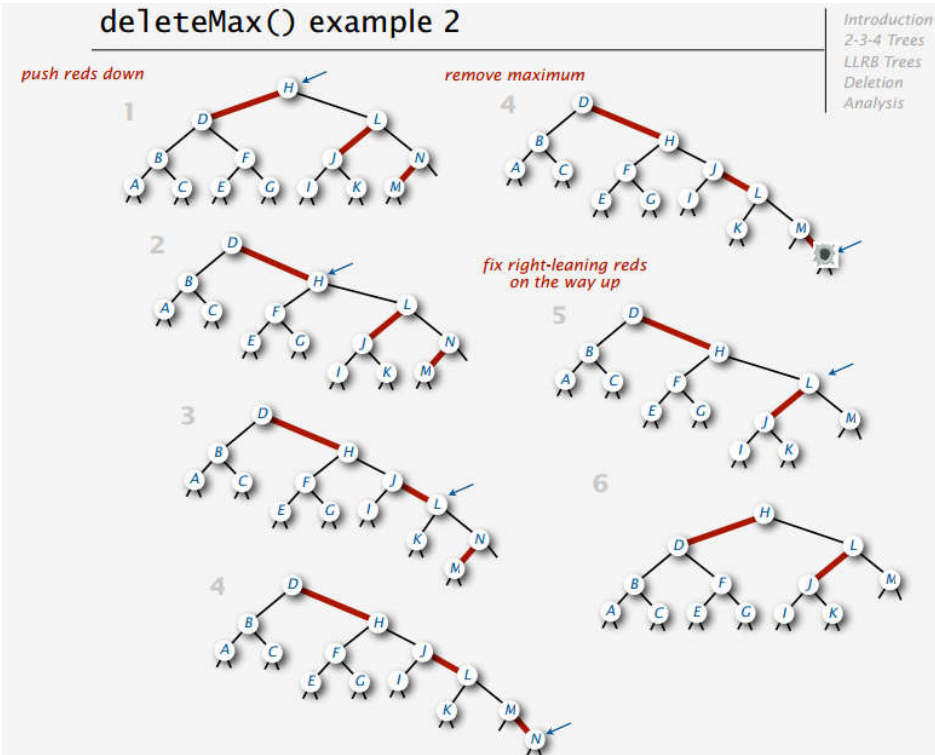


image.png

删除红黑树最小节点

最小节点的方法与最大节点的类似，只不过是从最右边变成了最左边

思想还是一样的

首先，不变量，就是h或者h的left一定是红色的。遇到底部的红节点，就直接删除了。

然后就是对于2-node需要从兄弟节点中借一个节点变成3-node或者4-node

2-node的条件就是，h.left和h.left.left均为黑色的。

然后其中又有两种情况，如果h.right.left为黑，则说明兄弟节点也是2-node，就从父节点借节点，直接color flip即可

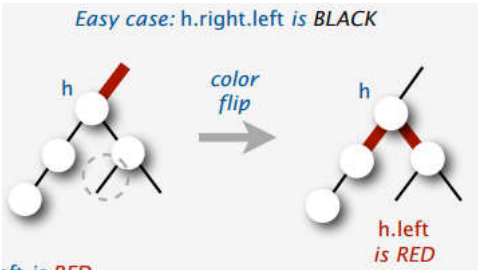


image.png

如果h.right.left为红，则可以直接从兄弟节点借一个节点过来。

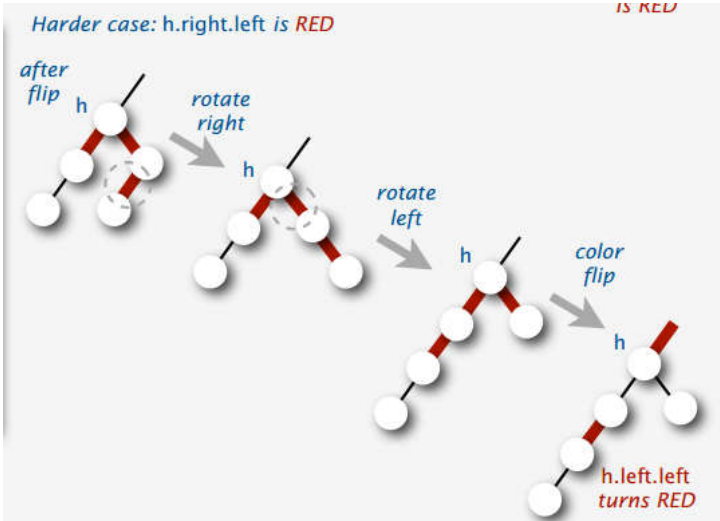


image.png

代码是

```
private Node moveRedLeft(Node h)
{
    colorFlip(h);
    if (isRed(h.right.left))
    {
        h.right = rotateRight(h.right);
        h = rotateLeft(h);
        colorFlip(h);
    }
    return h;
}
```

最后归纳得到删除最小节点的代码

📄

🔖

🔗

```
public void deleteMin()
{
    root = deleteMin(root);
    root.color = BLACK;
}
private Node deleteMin(Node h)
{
    if (h.left == null)
        return null;
    if (!isRed(h.left) && !isRed(h.left.left))
        h = moveRedLeft(h);
    h.left = deleteMin(h.left);
    return fixUp(h);
}
```

流程如下

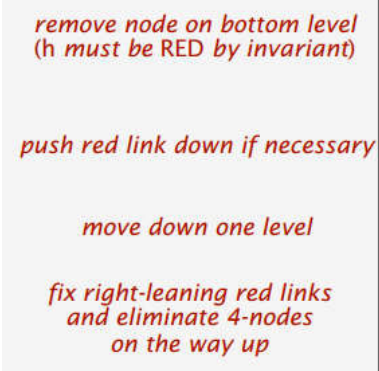


image.png

两个删除最小节点的例子：

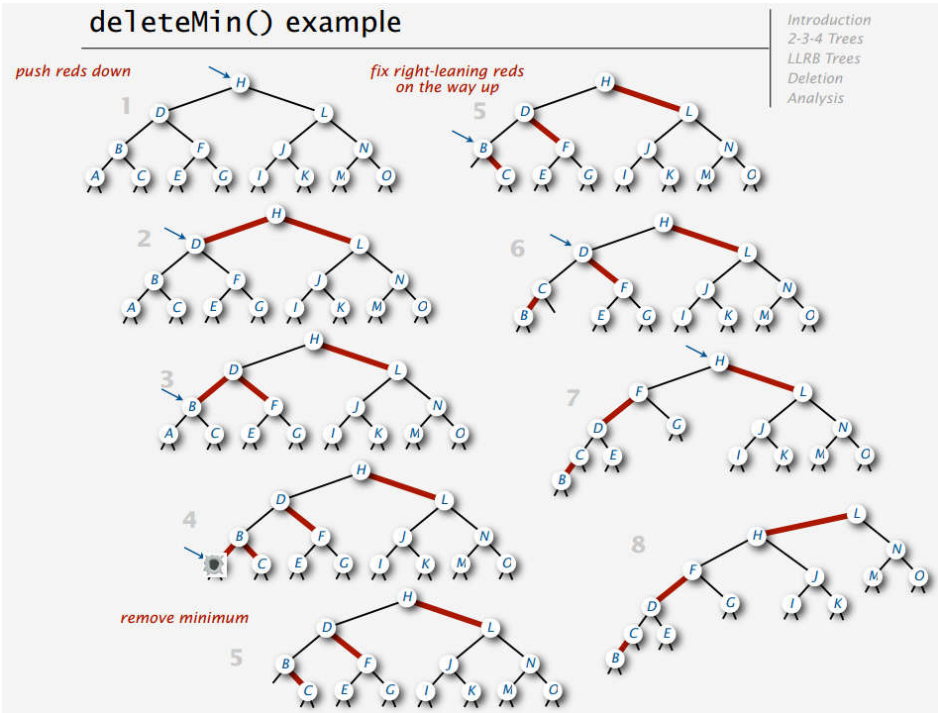


image.png

删除任意节点

我们学习了怎么删除最大节点和最小节点，下面我们开始研究最复杂的情况，就是删除任意节点
其实思路是一样的，如果所要删除的节点在3-node或者4-node中，根据2-3-4树的性质直接删除就可以了。
最复杂的情况是如果是2-node，那么删除就会引起不平衡。所以就得从兄弟节点中借一

📄

🔖

🔗

个节点，但是由于是任意节点，不像删除最大最小的情况，确定是左边或者右边，而是有很多情况。

根据理论研究， $9 \times 6 + 27 \times 9 + 81 \times 12 = 1269$ 多种情况（我也不知道咋算的哈哈，有兴趣去看论文）

所以单纯的这种思路是不行的。

我们变换想法，类似于堆，我们如果要删除一个节点，把要删除的那个节点和最底部的节点交换，然后就变成删除最底部的节点，就可以转换成删除最大节点或者最小节点了。这也就是我们为什么要先讲最大节点和最小节点。同时这样也把问题简化了，因为删除最大和最小节点的方法我们已经分析出来了。

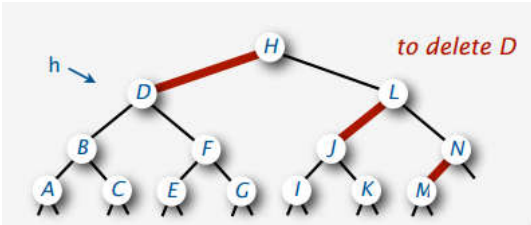


image.png

如果我们要删除D节点，我们可以选择用D节点左子树的最大节点或者右子树的最小节点来替换D的值，然后再删除最大节点或者最小节点就可以了。

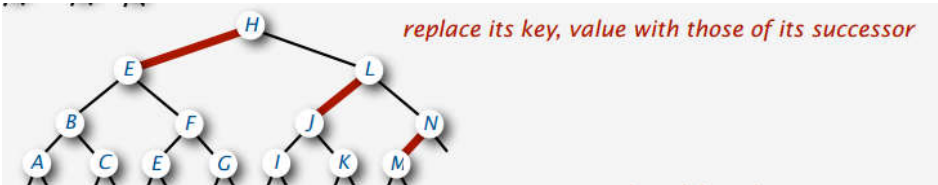


image.png

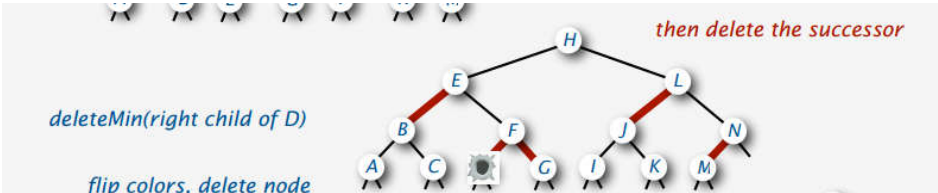


image.png



image.png

代码如下：

```
h.key = min(h.right);
h.value = get(h.right, h.key);
h.right = deleteMin(h.right);
```

对于删除的节点在最底部的情况，则我们可以直接利用前面最大节点和最小节点的方法往下搜索就行了，如果所删除的节点比当前节点大，就往右搜索，采取删除最大节点的搜索路径，反之，就往左搜索。

红黑树删除任意节点的代码

```
private Node delete(Node h, Key key)
{
    int cmp = key.compareTo(h.key);
    if (cmp < 0)
    {
        if (!isRed(h.left) && !isRed(h.left.left))
            h = moveRedLeft(h);
        h.left = delete(h.left, key);
    }
    else
    {
        if (isRed(h.left)) h = leanRight(h);
        if (cmp == 0 && (h.right == null))
            return null;
        if (!isRed(h.right) && !isRed(h.right.left))
            h = moveRedRight(h);
        if (cmp == 0)
        {
            h.key = min(h.right);
            h.value = get(h.right, h.key);
            h.right = deleteMin(h.right);
        }
        else h.right = delete(h.right, key);
    }
    return fixUp(h);
}
```

流程图如下：

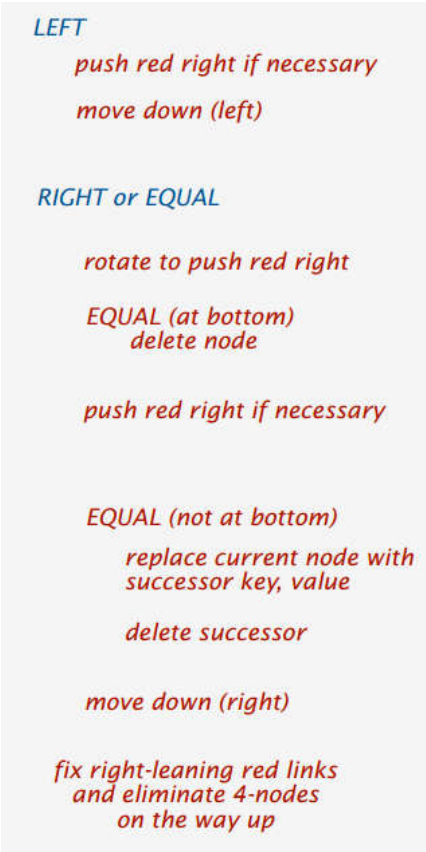


image.png

总结

至此，我们就基本讲完了红黑树的基本原理和实现。
我们首先从2-3-4树开始讲起，然后引出红黑树其实就是2-3-4树的BST的表示。接着介绍插入和删除算法。
很少需要我们自己手动实现红黑树，但我们需要对红黑树的基本原理，作用，算法的思路有一个基本的了解。这篇文章的目的就在此。

📄

🔖

🔗

本文主要基于<http://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf>
(<http://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf>)
这篇文章来讲解的，有兴趣的可以参考

📖 数据结构与算法 (/nb/14296477)


© 著作权归作者所有



六尺帐篷 (/u/f8e9b1c246f1)

写了 245418 字，被 15240 人关注，获得了 1429 个喜欢
(/u/f8e9b1c246f1)

❤ 喜欢 8





更多分享


(<http://cwb.assets.jianshu.io/notes/images/1447267>)


被以下专题收入，发现更多相似内容


⚙ 投稿管理


- + 收入我的专题
-  IT 共论 (/c/1113b792c0dc?utm_source=desktop&utm_medium=notes-included-collection)

 程序员 (/c/NEt52a?utm_source=desktop&utm_medium=notes-included-collection)

 Android 知识 (/c/3fde3b545a35?utm_source=desktop&utm_medium=notes-included-collection)

 Android... (/c/5139d555c94d?utm_source=desktop&utm_medium=notes-included-collection)

 iOS Dev... (/c/3233d1a249ca?utm_source=desktop&utm_medium=notes-included-collection)

 数据结构 (/c/a93492d8d125?utm_source=desktop&utm_medium=notes-included-collection)





