Map集合、散列表、红黑树介绍

> 声明，本文用得是jdk1.8

前面已经讲了Collection的总览和剖析List集合：

- Collection总览
- List集合就这么简单【源码剖析】

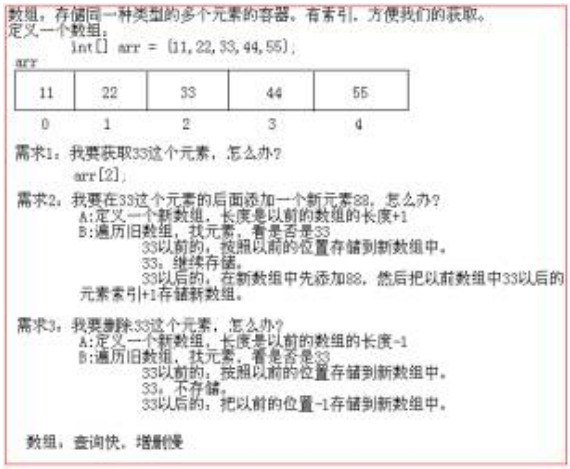
原本我是打算继续将Collection下的Set集合的，结果看了源码发现：**Set集合实际上就是HashMap来构建的**！

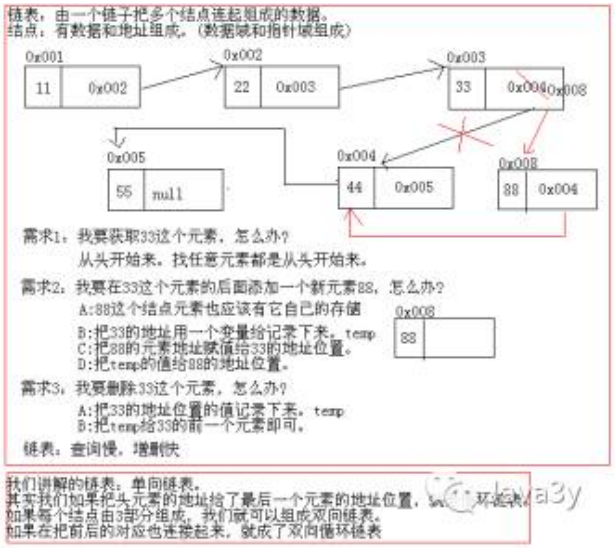
```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{
    static final long serialVersionUID = -5024744406713321676L;

    private transient HashMap<E,Object> map;

    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();

    /**
     * Constructs a new, empty set; the backing <tt>HashMap</tt> instance has
     * default initial capacity (16) and load factor (0.75).
     */
    public HashSet() {
        map = new HashMap<>();
    }
```


所以，就**先介绍Map集合、散列表和红黑树吧**！

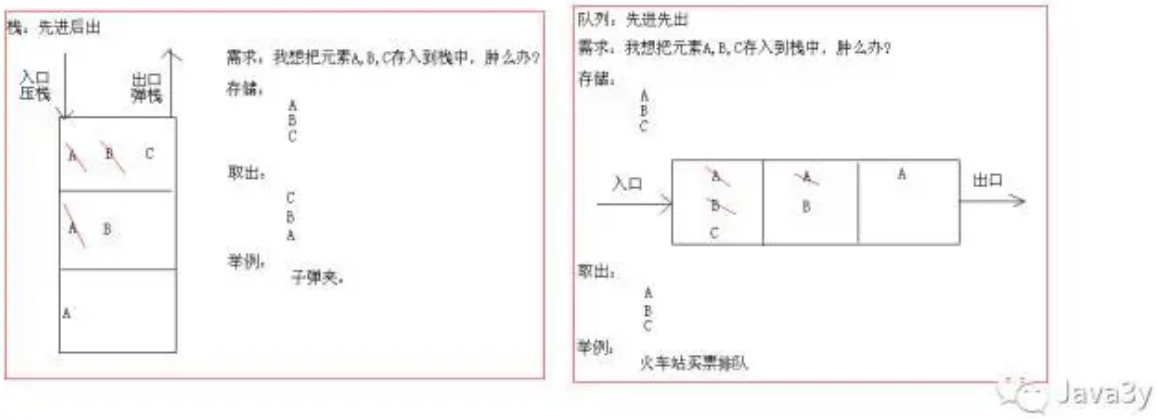
看这篇文章之前最好是有点数据结构的基础：

- Java实现单向链表
- 栈和队列就是这么简单
- 二叉树就这么简单





数据结构，数据的组织方式。
面试题，常见的数据结构的优缺点？
（数据结构+算法）



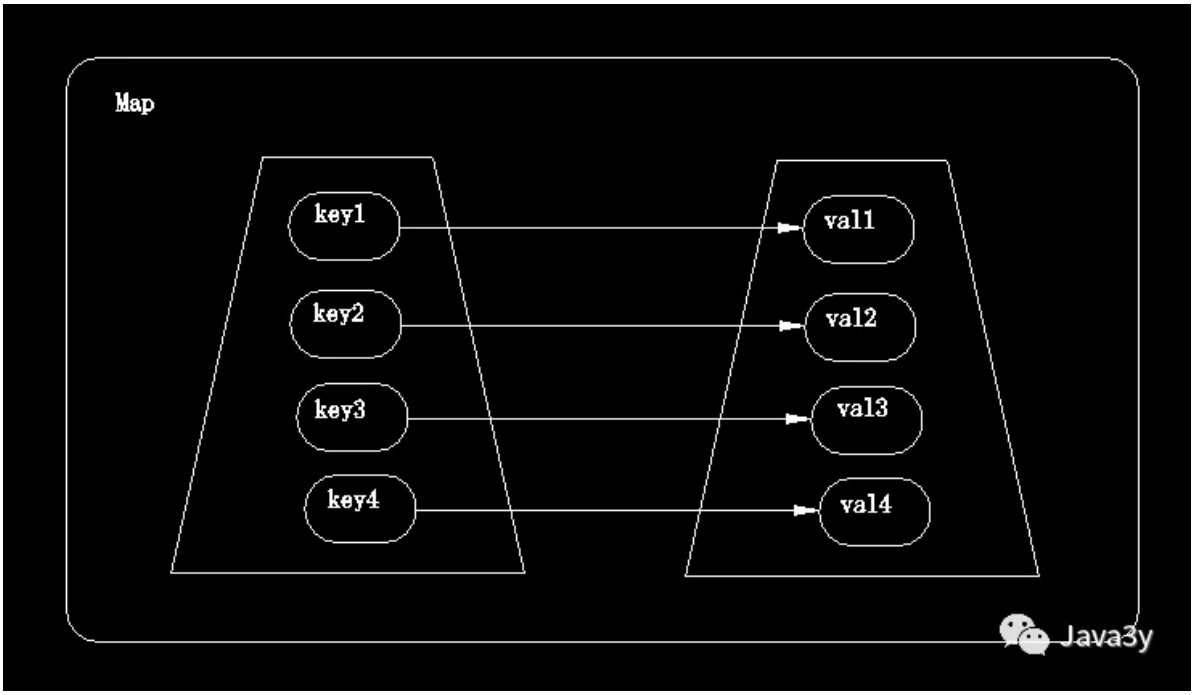
一、Map介绍

1.1为什么需要Map

前面我们学习的Collection叫做集合，它可以快速查找现有的元素。

而Map在《Core Java》中称之为-->映射..

映射的模型图是这样的：



那为什么我们需要这种数据存储结构呢？？？举个例子

- 作为学生来说，我们是根据学号来区分不同的学生。**只要我们知道学号，就可以获取对应的学生信息。**这就是Map映射的作用！

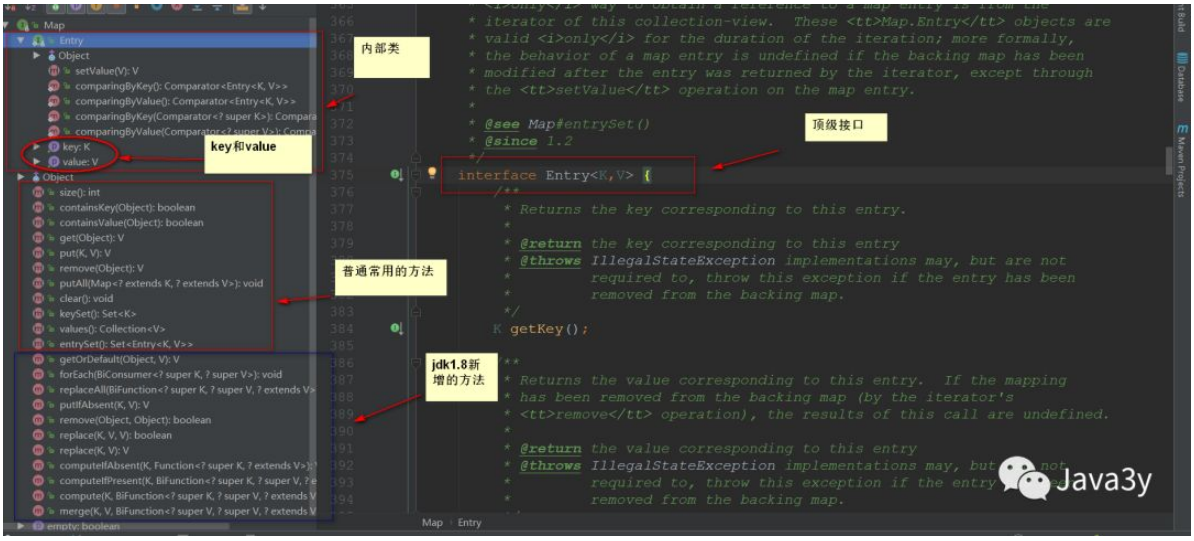
生活中还有很多这样的例子：**只要你掏出身份证(key)，那就可以证明是你自己(value)**

1.2Map与Collection的区别

```
/*
    Map集合的特点:
        将键映射到值的对象,一个映射不能包含重复的键,每个键最多只能映射到一个值

    Map和Collection集合的区别:
        1:Map集合存储元素是成对出现的,Map的键是唯一的,值是可以重复的.
        2:Collection集合存储元素是单独出现的,Collection的儿子Set是唯一的,List是可重复的

    要点:
        1:Map集合的数据结构针对键有效,跟值无关
        2:Collection集合的数据结构针对元素有效
 */
```


1.3Map的功能

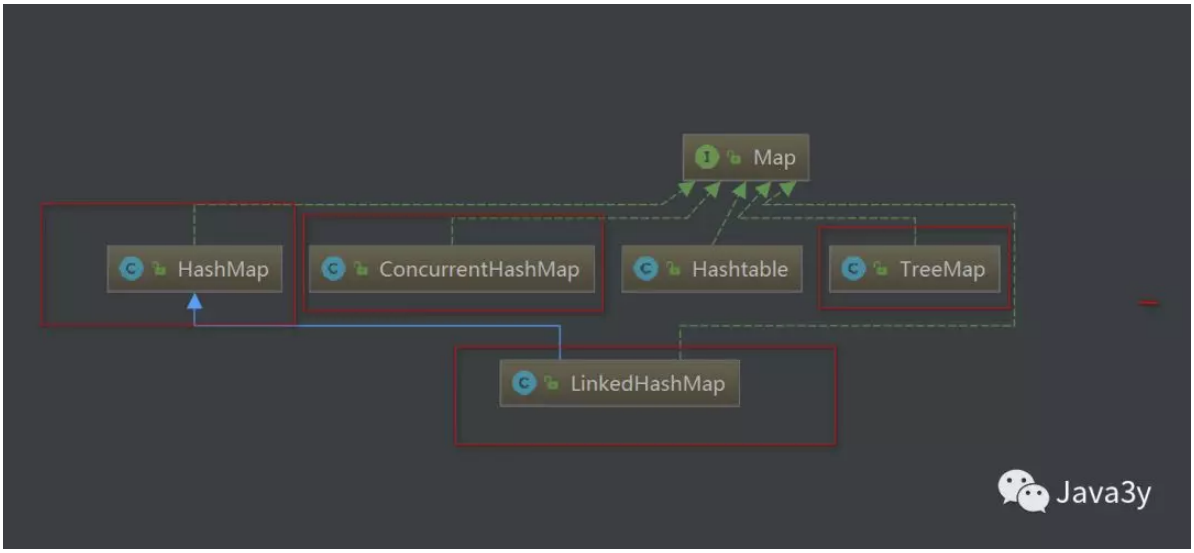
下面我们来看看Map的源码:



简单常用的Map功能有这么一些:

```
/**
 * Map集合的功能概述:
 *     1:添加功能
 *       V put (K key , V value):添加元素
 *           如果键是第一次存储，就直接存储元素，返回null
 *           如果键不是第一次存在，就用值把以前的值替换掉，返回以前的值
 *
 *     2:删除功能
 *       void clear():移除所有的键值对元素
 *       V remove(Object key):根据键删除值,并把值返回
 *
 *     3:判断功能
 *       boolean containsKey(Object key):判断集合是否包含指定的键
 *       boolean containsValue(Object value):判断集合是否包含指定的值
 *       boolean isEmpty():判断集合是否为空
 *
 *     4:获取功能
 *       Set<Map.Entry<K key, V value> entrySet():返回的是键值对对象的集合
 *       V get(Object key):根据键获取值
 *       Set<K> keySet():获取集合中所有的键的集合
 *       Collection<V> values():获取集合中所有值的集合
 *
 *     5:长度功能
 *       int size():返回集合中键值对的对数
 *
 */
```


下面用红色框框圈住的就是**Map值得关注的子类**:



二、散列表介绍

无论是Set还是Map，我们会发现都会有对应的-->**Hash**Set,**Hash**Map

首先我们也先得**回顾一下数据和链表**：

- 链表和数组都可以按照人们的意愿来排列元素的次序，他们可以说是**有序**的(存储的顺序和取出的顺序是一致的)
- 但同时，这会带来缺点：**想要获取某个元素，就要访问所有的元素，直到找到为止。**
- 这会让我们消耗很多的时间在里边，遍历访问元素~

而还有另外的一些存储结构：**不在意元素的顺序，能够快速的查找元素的数据**

- 其中就有一种非常常见的：**散列表**

2.1散列表工作原理

散列表**为每个对象计算出一个整数，称为散列码。根据**这些计算出来的**整数(散列码)保存在对应的位置上**!

在Java中，散列表用的是链表数组实现的，**每个列表称之为桶。**【之前也写过桶排序就这么简单，可以回顾回顾】

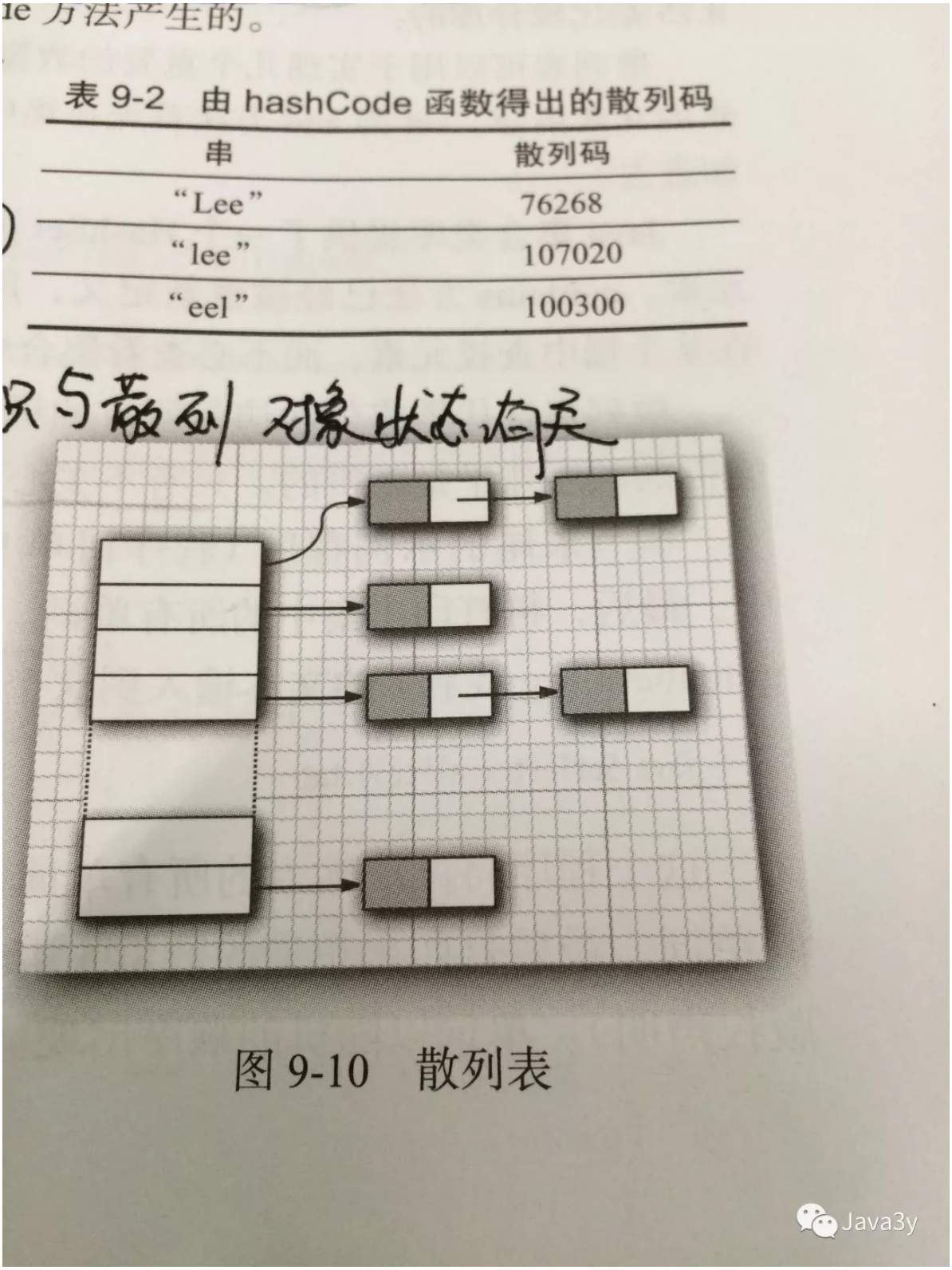


表 9-2　由 hashCode 函数得出的散列码

串	散列码
“Lee”	76268
“lee”	107020
“eel”	100300

图 9-10　散列表

一个桶上可能会**遇到被占用的情况(hashCode散列码相同，就存储在同一个位置上)**，这种情况是无法避免的，这种现象称之为：**散列冲突**

- 此时需要**用该对象与桶上的对象进行比较，看看该对象是否存在桶子上了**~如果存在，就不添加了，如果不存在则添加到桶子上
- 当然了，如果hashcode函数设计得足够好，桶的数目也足够，这种比较是很少的~
- 在**JDK1.8**中，**桶满时**会从**链表变成平衡二叉树**

如果散列表太满，**是需要对散列表再散列，创建一个桶数更多的散列表，并将原有的元素插入到新表中，丢弃原来的表**~

- 装填因子(load factor)**决定了何时**对散列表再散列~

- 装填因子默认为0.75，如果表中**超过了75%的位置**已经填入了元素，那么这个表就会用**双倍的桶数**自动进行再散列

当然了， 在后面阅读源码的时候会继续说明的，现在简单了解一下即可~

扩展阅读：

- https://www.cnblogs.com/s-b-b/p/6208565.html
- https://www.cnblogs.com/chinajava/p/5808416.html

三、红黑树介绍

上面散列表中已经提过了：如果桶数满的时候，JDK8是将**链表转成红黑树**的~。并且，我们的TreeSet、TreeMap底层都是红黑树来实现的。

所以，在这里学习一波红黑树到底是啥玩意。

之前涉及过二叉树的文章：

- 二叉树就这么简单
- 堆排序就这么简单

在未学习之前，我们可能是听过红黑树这么一个数据结构类型的，还有其他什么B/B+树等等，反正是**比较复杂的数据结构了**~~~

各种常见的树的用途：

AVL树，红黑树，B树，B+树，Trie树都分别应用在哪些现实场景...

Bill Utada

535 人赞同了该回答

AVL树: 最早的平衡二叉树之一。应用相对其他数据结构比较少。windows对进程地址空间的管理用到了AVL树。

红黑树: 平衡二叉树，广泛用在C++的STL中。如map和set都是用红黑树实现的。

B/B+树: 用在磁盘文件组织 数据索引和数据库索引。

Trie树(字典树): 用在统计和排序大量字符串，如自动机。

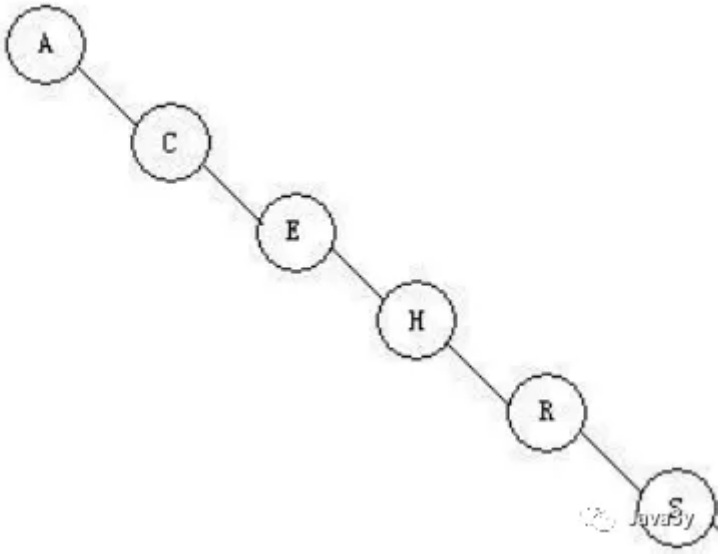
来源：
https://www.zhihu.com/question/30527705/answer/52527887

3.1回顾二叉查找树

首先我们来回顾一下：利用二叉查找树的特性，我们一般来说可以很快地查找出对应的元素。

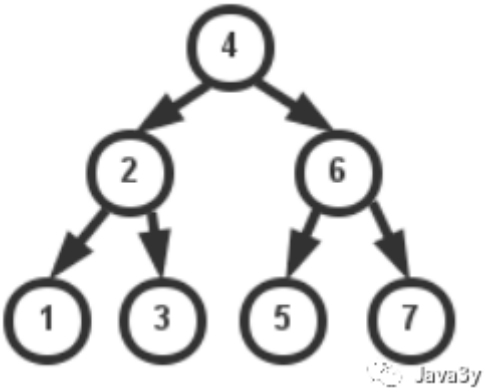
- 二叉树就这么简单

可是二叉查找树也有**个例(最坏)**的情况(线性)：



上面符合二叉树的特性，但是它是线性的，完全没树的用处~

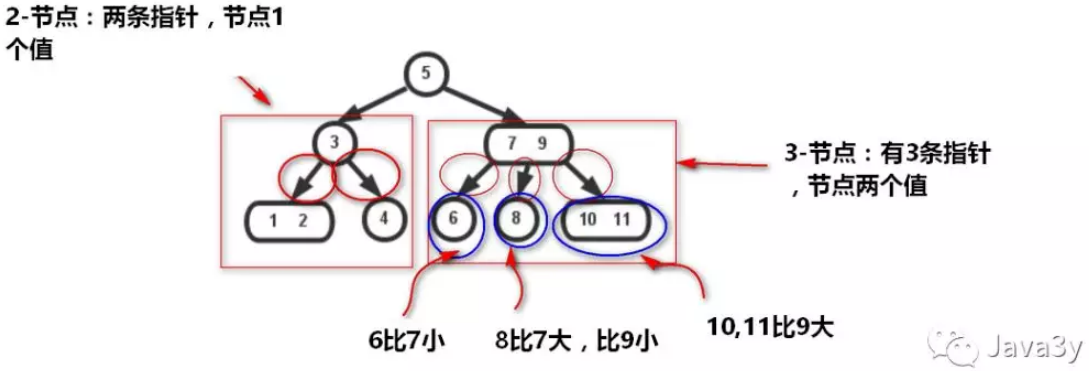
树是要**“均衡”**才能将它的优点展示出来的~，比如下面这种：



因此，就有了**平衡树**这么一个概念~红黑树就是一种平衡树，它可以**保证二叉树基本符合矮矮胖胖(均衡)的结构**

3.2知新2-3树

讲到了平衡树就不得不说**最基础**的2-3树，2-3树**长的是这个样子：**



在二叉查找树上，我们插入节点的过程是这样的：小于节点值往右继续与左子节点比，大于则继续与右子节点比，直到某节点左或右子节点为空，把值插入进去。**这样无法避免偏向问题**

而2-3树不一样：**它插入的时候可以保持树的平衡**！

在2-3树插入的时可以简单总结为两个操作：

- **合并2-节点为3-节点，扩充将3-节点扩充为一个4-节点**
- **分解4-节点为3-节点，节点3-节点为2-节点**
- ……**至使得树平衡**~

合并分解的操作还是**比较复杂的，要分好几种情况**，代码量很大~这里我就不介绍了，因为要学起来是一大堆的，很麻烦~

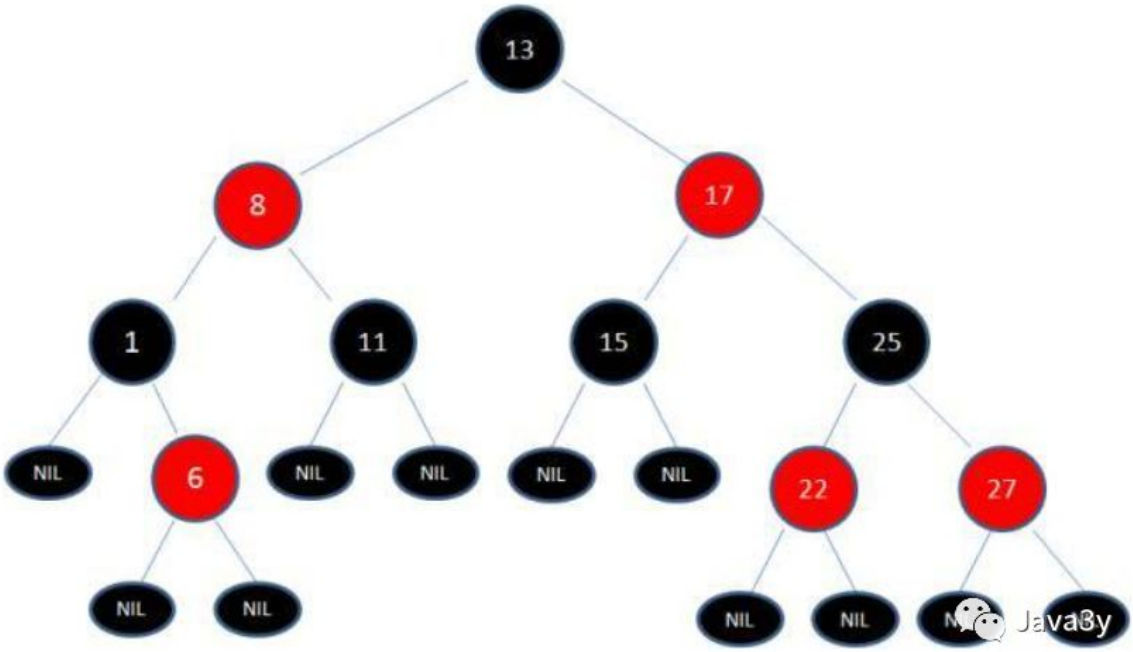
3.3从2-3树到红黑树

由于2-3树为了保持平衡性，在维护的时候是需要大量的节点交换的！这些变换在实际代码中是很复杂的，大佬们**在2-3树的理论基础上发明了红黑树**(2-3-4树也是同样的道理，只是2-3树是最简单的一种情况，所以我就不说2-3-4树了)。

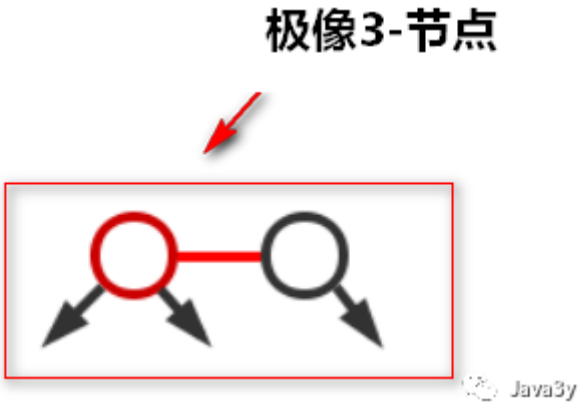
- 红黑树是对2-3查找树的改进，它能用一种**统一的方式完成所有变换。**

红黑树是一种**平衡二叉树**,因此它没有3-节点。那红黑树是怎么将3-节点来改进成全都是二叉树呢?

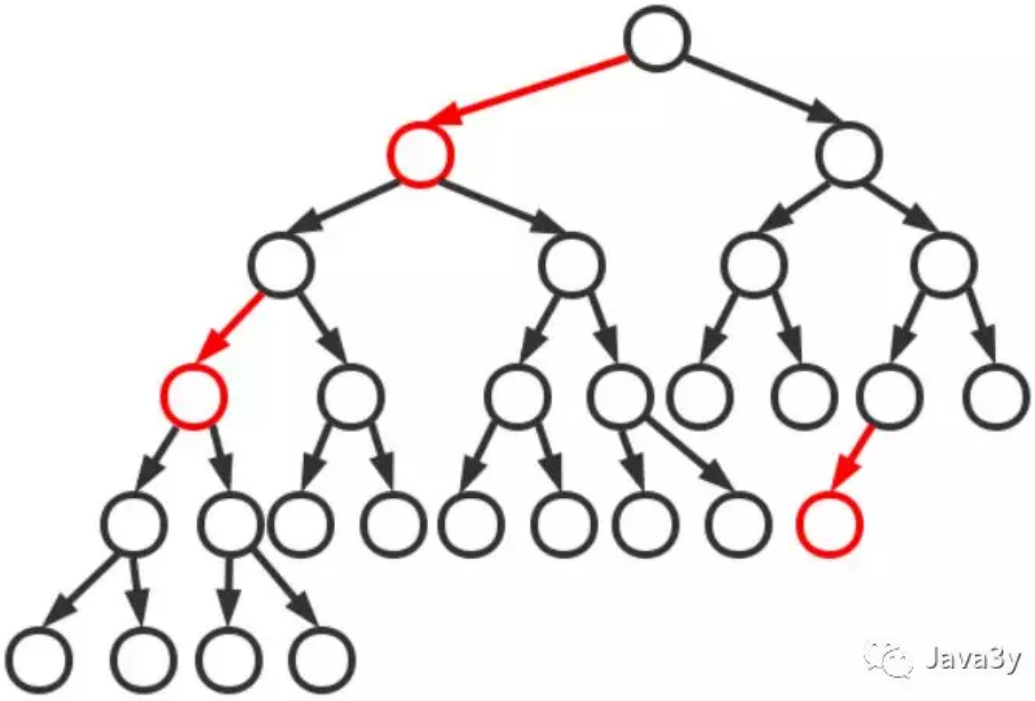
红黑树就字面上的意思，**有红色的节点，有黑色的节点**:



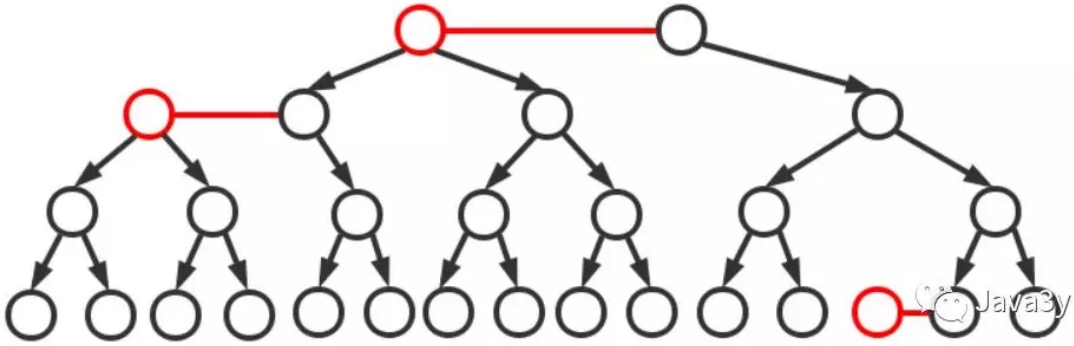
我们可以将红色节点的**左链接**画平看看:



一颗典型的二叉树：



将红色节点的左链接画平之后：得到2-3平衡树：



3.4红黑树基础知识

前面已经说了，红黑树是在2-3的基础上实现的一种树，它能够用统一的方式完成所有变换。很好理解：红黑树也是平衡树的一种，在插入元素的时候它也得保持树的平衡，那红黑树是以什么的方式来保持树的平衡的呢?

红黑树用的是也是两种方式来替代2-3树不断的节点交换操作：

- **旋转**：顺时针旋转和逆时针旋转
- **反色**：交换红黑的颜色
- 这个两个实现比2-3树交换的节点(合并，分解)要方便一些

红黑树为了保持平衡，还有制定一些约束，遵守这些约束的才能叫做红黑树：

1. 红黑树是二叉搜索树。
2. **根节点是黑色。**
3. **每个叶子节点都是黑色的空节点（NIL节点）。**
4. **每个红色节点的两个子节点都是黑色。(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路径都包含相同数目的黑色节点(每一条树链上的黑色节点数量（称之为“黑高”）必须相等)。**

3.5红黑树总结

红黑树可以说是十分复杂的，我在学习的时候并没有去认真细看当中的处理细节，只是大概的过了一遍，知道了整体~

有了前辈很多优质的资料，相信要等到想要理解其中的细节，花点力气和时间还是可以掌握一二的。

红黑树参考资料：

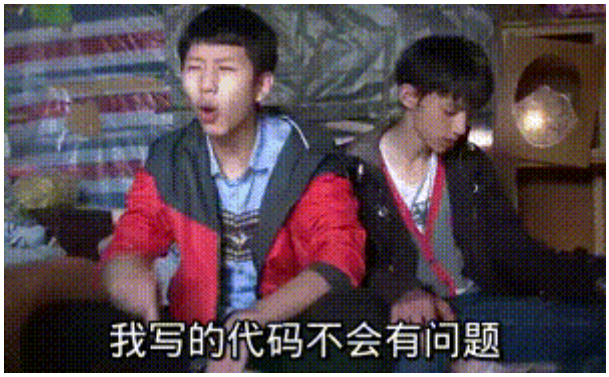
- https://blog.csdn.net/chen_zhang_yu/article/details/52415077
- https://riteme.github.io/blog/2016-3-12/2-3-tree-and-red-black-tree.html#fn:red-is-left
- http://www.sohu.com/a/201923614_466939
- https://www.jianshu.com/p/37c845a5add6
- https://www.cnblogs.com/nullzx/p/6111175.html
- https://blog.csdn.net/fei33423/article/details/79132930

四、总结

这篇主要介绍了Map集合的基础知识，了解Map的常用子类~

简单介绍了散列表和红黑树，他俩作为Hashxxx和Treexxx的底层，了解其整体思想和相关基础在后续看源码也不至于那么懵~

后续会去看Map常用子类的源码，文章敬请期待~~~~



文章的目录导航：https://zhongfucheng.bitcron.com/post/shou-ji/gong-zhong-hao-wen-zhang-zheng-li