# ConcurrentMap - 并发映射

## 1.概述

ConcurrentMap及其子类是JDK1.5提供的一套用于对应高并发的映射机制

ConcurrentMap在并发的时候还能比较好的保证线程安全

## 2.ConcurrentHashMap - 并发哈希映射

### 2.1.ConcurrentHashMap关于容器的剖析

ConcurrentHashMap底层是基于数组+链表来存储，数组的默认容量是16(表示有16个桶)，默认加载因子是0.75，默认扩容是增加一倍的桶数。ConcurretHashMap最多允许存在230个桶。

ConcurrentHashMap底层对应的常量：

初始桶数：

*/\*\**

*\* The default initial table capacity. Must be a power of 2  
 \* (i.e., at least 1) and at most MAXIMUM\_CAPACITY.  
 \*/*private static final int *DEFAULT\_CAPACITY* = 16;

加载因子：

*/\*\**

*\* The load factor for this table. Overrides of this value in  
 \* constructors affect only the initial table capacity. The  
 \* actual floating point value isn't normally used -- it is  
 \* simpler to use expressions such as {****@code*** *n - (n >>> 2)} for  
 \* the associated resizing threshold.  
 \*/*private static final float *LOAD\_FACTOR* = 0.75f;

最大桶数：

*/\*\**

*\* The largest possible table capacity. This value must be  
 \* exactly 1<<30 to stay within Java array allocation and indexing  
 \* bounds for power of two table sizes, and is further required  
 \* because the top two bits of 32bit hash fields are used for  
 \* control purposes.  
 \*/*private static final int *MAXIMUM\_CAPACITY* = 1 << 30;

#### 2.1.1.ConcurrentHashMap容量计算剖析

在ConcurrentHashMap中，允许指定容量。这个指定的容量需要经过计算，最终结果一定是2n的形式区间。

如果外界指定的容量值大于最大容量值则直接选择最大容量，如果小于将会先做一步计算，然后调用本来中的tablesSizeFor()方法来进行进一步的计算。

公式基本可以理解为：(n + n/2 + 1)

*/\*\**

*\* Creates a new, empty map with an initial table size  
 \* accommodating the specified number of elements without the need  
 \* to dynamically resize.  
 \*  
 \** ***@param*** *initialCapacity The implementation performs internal  
 \* sizing to accommodate this many elements.  
 \** ***@throws*** *IllegalArgumentException if the initial capacity of  
 \* elements is negative  
 \*/*public ConcurrentHashMap(int initialCapacity) {  
 if (initialCapacity < 0)  
 throw new IllegalArgumentException();  
 int cap = ((initialCapacity >= (*MAXIMUM\_CAPACITY* >>> 1)) ?  
 *MAXIMUM\_CAPACITY* :  
 *tableSizeFor*(initialCapacity + (initialCapacity >>> 1) + 1));  
 this.sizeCtl = cap;  
}

计算最终容量的方法：

在方法里进行第一次的容量计算，之后就会在此方法里面进行再一次的计算，经过了这个方法的计算之后会将原值放在2n的位置作为结果输出。

*/\*\**

*\* Returns a power of two table size for the given desired capacity.  
 \* See Hackers Delight, sec 3.2  
 \*/*private static final int tableSizeFor(int c) {  
 int n = c - 1;  
 n |= n >>> 1;  
 n |= n >>> 2;  
 n |= n >>> 4;  
 n |= n >>> 8;  
 n |= n >>> 16;  
 return (n < 0) ? 1 : (n >= *MAXIMUM\_CAPACITY*) ? *MAXIMUM\_CAPACITY* : n + 1;  
}

#### 2.1.2.ConcurrentHashMap容量计算案例

例子：

假设当前在外界构造ConcurrentHashMap的时候指定的长度为48,那么最终计算出来的长度为128。通过以下的计算可以判断输入的值经过第一次的计算后必定或落在2n<73<2x+1右边的位置，现在是73那么必定会落在64<73<128的位置，并且一定是取128。

计算公式：

构造方法：

(48 + 48/2 + 1)=73

tablesSizeFor()：

73 – 1 = 72

72 | = 72>>>1 = 108

108 | = 108>>>2 = 127

127 | = 127>>>4 = 127

127 | = 127>>>8 = 127

127 | = 127>>>16 = 127

127 + 1 = 128

#### 2.1.3.ConcurrentHashMap与HashMap计算区别

两者最主要的区别在于构造方法里调用tablesSizeFor()方法之前的计算方式不同，可以通过查看以下两段代码看出，HashMap调用方法之前并不会进行计算而是直接传入。

ConcurrentHashMap:

*/\*\**

*\* Creates a new, empty map with an initial table size  
 \* accommodating the specified number of elements without the need  
 \* to dynamically resize.  
 \*  
 \** ***@param*** *initialCapacity The implementation performs internal  
 \* sizing to accommodate this many elements.  
 \** ***@throws*** *IllegalArgumentException if the initial capacity of  
 \* elements is negative  
 \*/*public ConcurrentHashMap(int initialCapacity) {  
 if (initialCapacity < 0)  
 throw new IllegalArgumentException();  
 int cap = ((initialCapacity >= (*MAXIMUM\_CAPACITY* >>> 1)) ?  
 *MAXIMUM\_CAPACITY* :  
 *tableSizeFor*(initialCapacity + (initialCapacity >>> 1) + 1));  
 this.sizeCtl = cap;  
}

HashMap:

HashMap由于输入的值是直接调用计算方法所以在这里假设输入的是19，那么最终计算出来的容量必定为16<19<32，取32。

*/\*\**

*\* Constructs an empty <tt>HashMap</tt> with the specified initial  
 \* capacity and load factor.  
 \*  
 \** ***@param*** *initialCapacity the initial capacity  
 \** ***@param*** *loadFactor the load factor  
 \** ***@throws*** *IllegalArgumentException if the initial capacity is negative  
 \* or the load factor is nonpositive  
 \*/*public HashMap(int initialCapacity, float loadFactor) {  
 if (initialCapacity < 0)  
 throw new IllegalArgumentException("Illegal initial capacity: " +  
 initialCapacity);  
 if (initialCapacity > *MAXIMUM\_CAPACITY*)  
 initialCapacity = *MAXIMUM\_CAPACITY*;  
 if (loadFactor <= 0 || Float.*isNaN*(loadFactor))  
 throw new IllegalArgumentException("Illegal load factor: " +  
 loadFactor);  
 this.loadFactor = loadFactor;  
 this.threshold = *tableSizeFor*(initialCapacity);  
}

#### 2.1.4.ConcurrentHashMap的面试题

创建一个ConcurrentHashMap对象，定义容量为100，存入了100个元素在极端情况下是否需要进行扩容？

答案：

不需要

原因：

经过基层计算容量为100+50+1=151所在范围128<151<256，取256，这个时候当前容量确定为256，经过计算0.75的加载因子计算后确定为临界值为192，最终确认存入的元素在极端情况下占用了193个桶才会进行扩容。

如果是HashMap的话由于在记性最终容量计算前不会做任何计算，那么可以直接判断容量为64<100<128取128，经过0.75的加载因子计算确认扩容临界值为75，这么在这个时候存入了100个元素，在计算情况下必定会进行双倍的扩容。

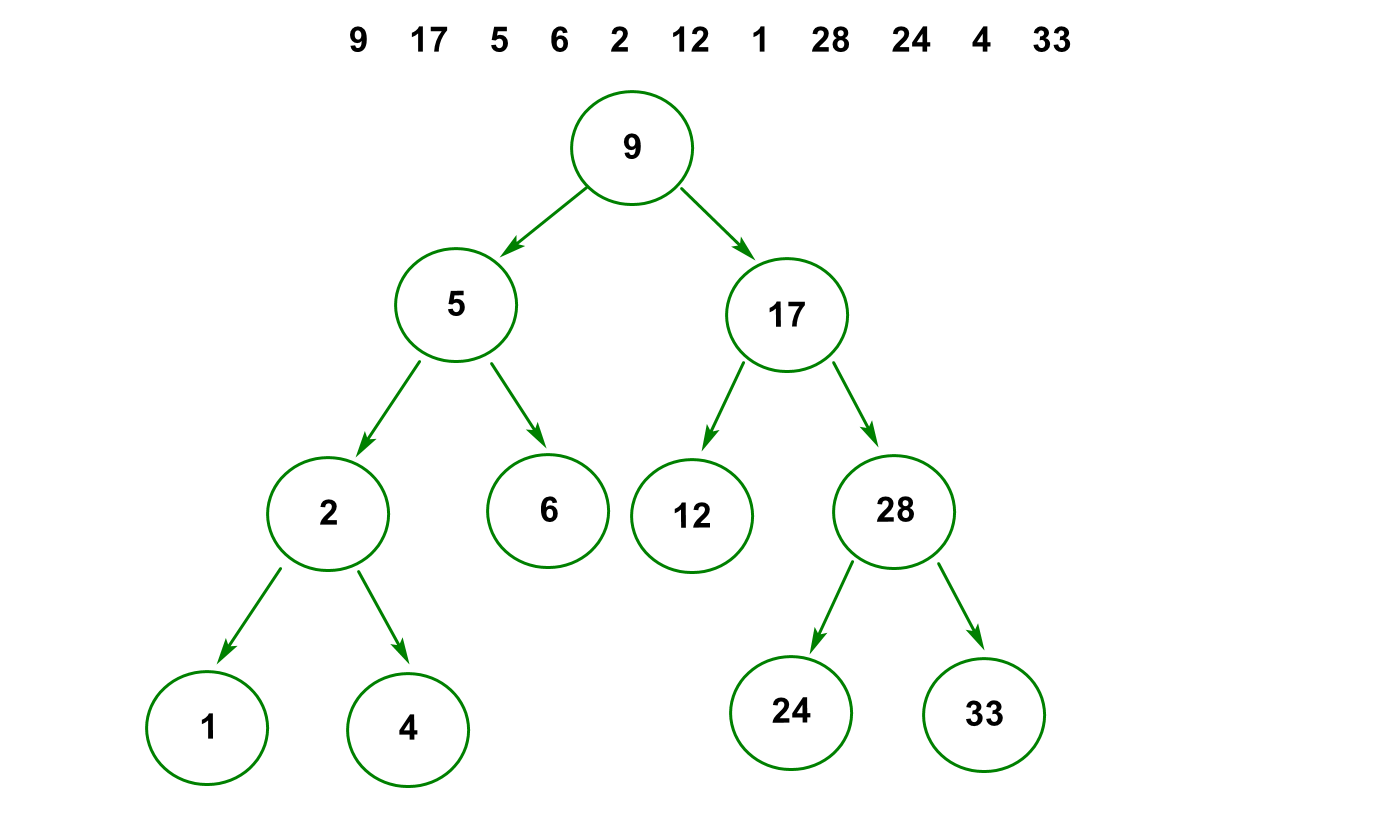
### 2.2. ConcurrentHashMap的红黑树

在JDK1.8中，为了提高效率引入了红黑树机制。当桶中的元素个数超过8个的时候，这个桶中的链表扭转成一棵红黑树；如果红黑树的节点个数不足7个的时候，将红黑树扭转回链表。扭转成红黑树的前提：桶的数量>=64，那么也就意味着如果桶数<64，那么桶中的元素个数无论是多少都不会变成红黑树。

#### 2.2.1.二叉树

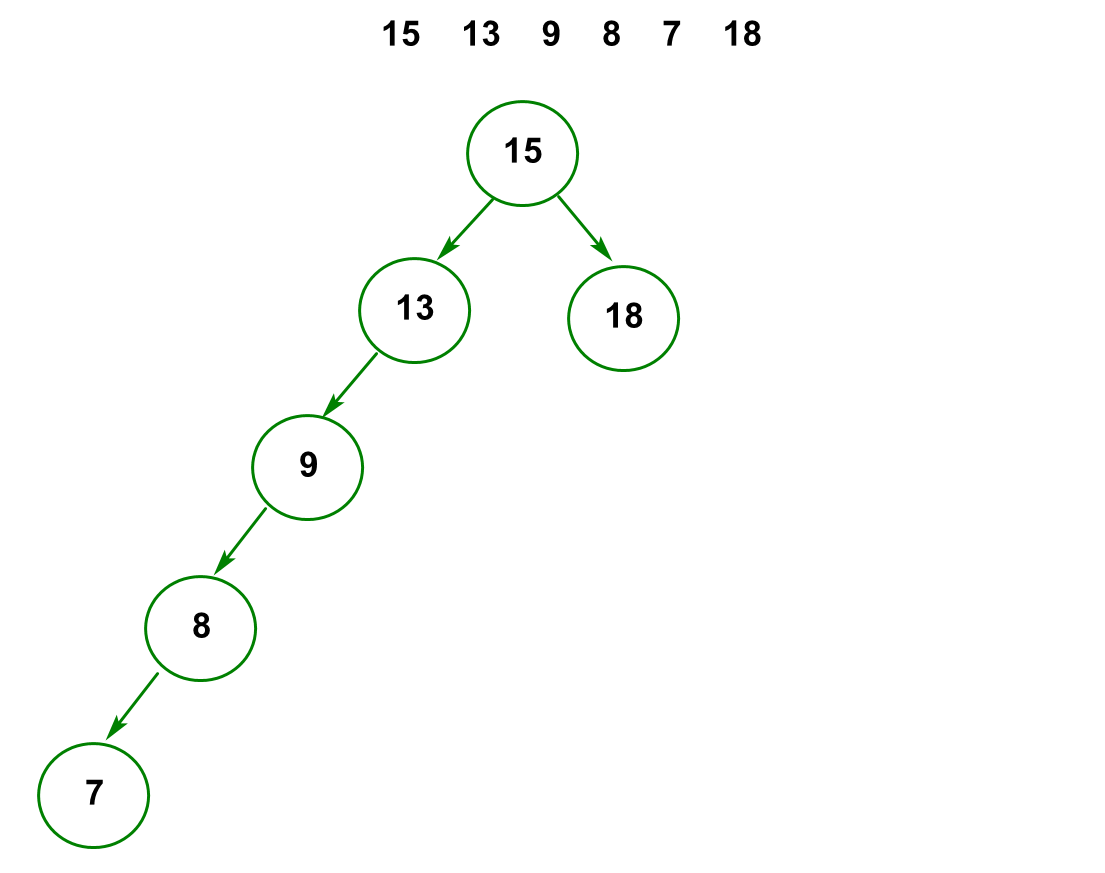
二叉查找树(BST)：在二叉树的基础上加入了排序机制，使得左子树一定是小于根，右子树一定是大于根。

理想情况下的二叉树：



非理想情况下的二叉树：

左右挂载阶段倾斜很大以这种结构反而使用链表的进行来查找的效率更改，叫法也有很多种，有“瘸腿二叉树”，“非平衡二叉树”等。



#### 2.2.2.红黑树概述及特征

本质上是一棵自平衡二叉查找树，他解决了普通的查找二叉树中会出现阶段挂载不平均的情况。

1.所有节点非红即黑

2.根节点为黑

3.红节点的子节点一定是黑的

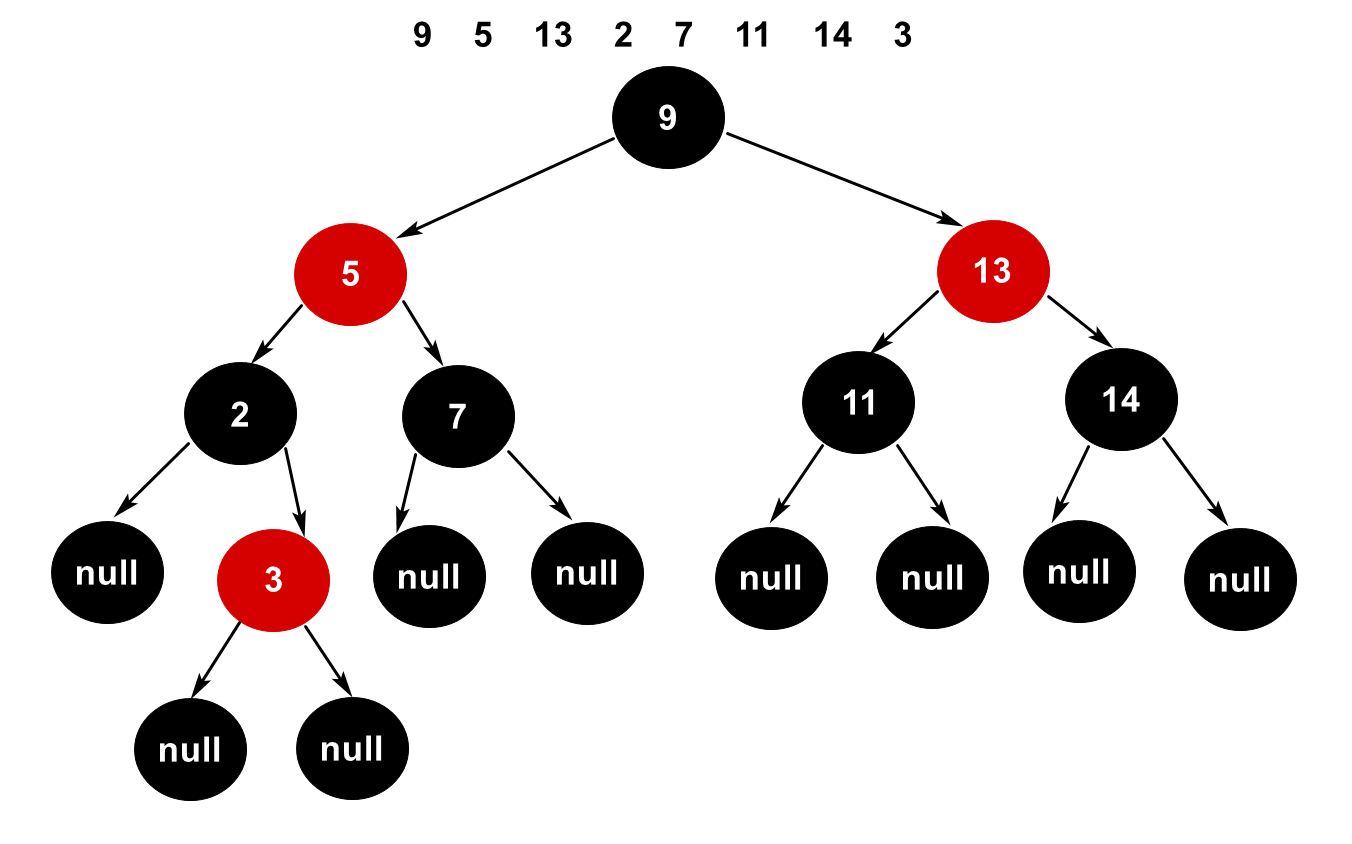
4.最底下的叶子节点是黑色的空节点

5.从根节点到叶子节点经过的路径中的黑色节点个数一致，即黑节点的高度是一致的

6.新添节点的颜色一定是红色的

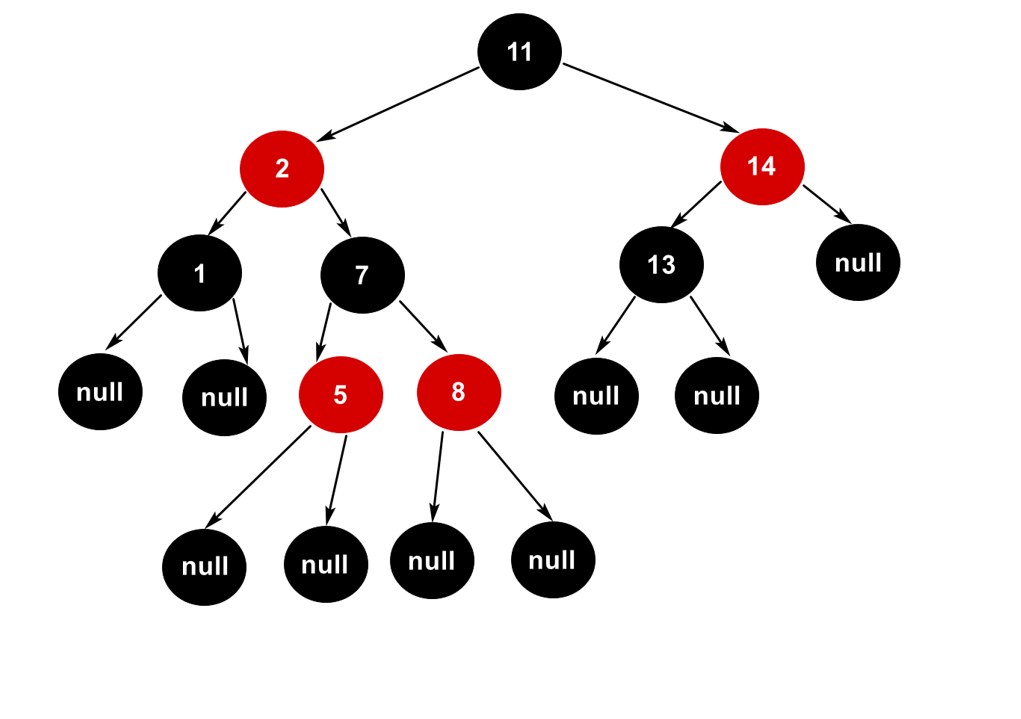
7.红黑树在构建过程中，每次添加一个新的节点都需要考虑是否需要修正：涂色、左旋、右旋。

##### 红黑树示意图

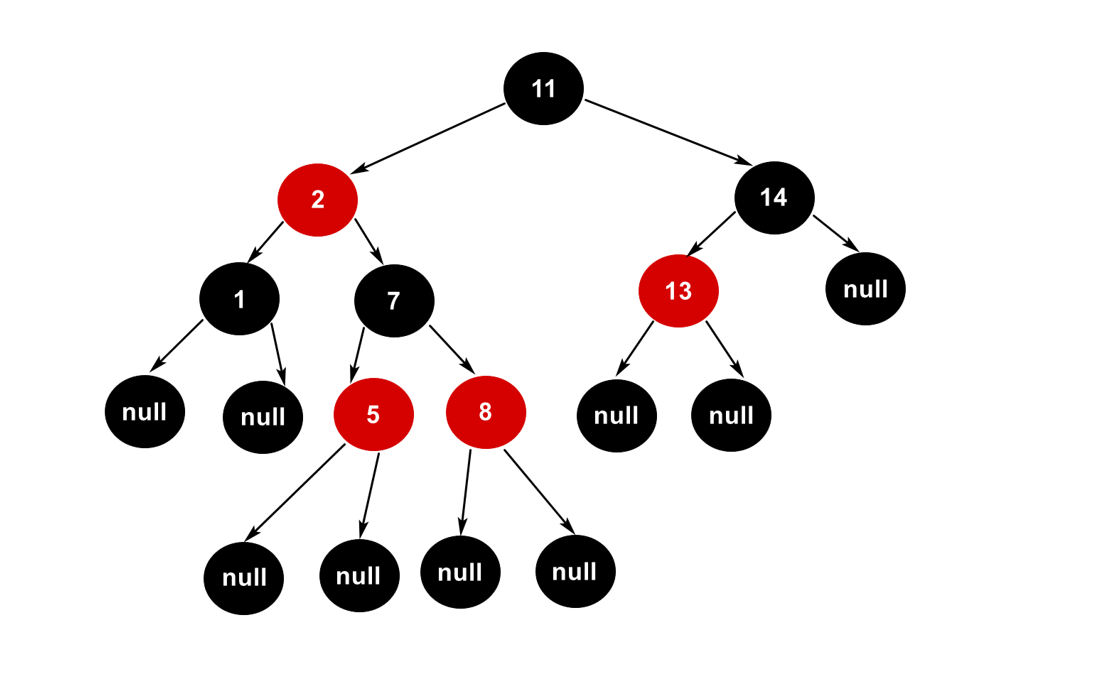


##### 红黑树涂色修正-01

从下列的红黑树图中可以看到，根节点到每一个最底下的叶节点经过的黑色节点左右不一致，违反了红黑树的规则。

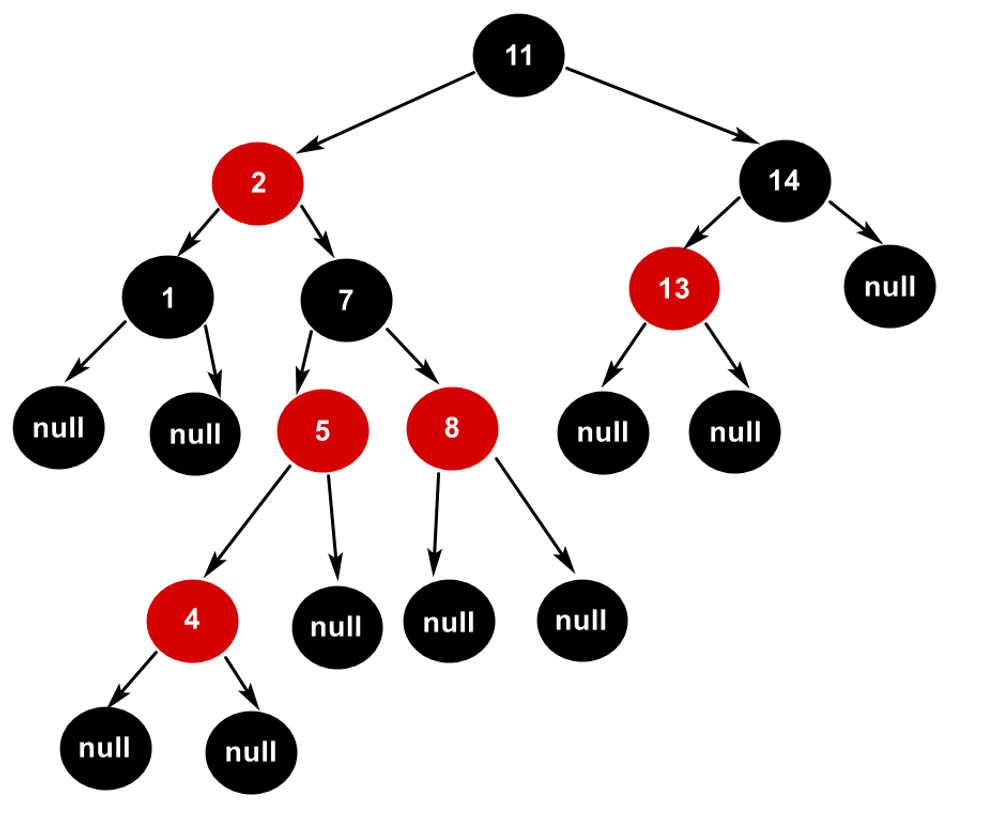


经过涂色修正后将右边的黑节点与左边的的黑节点保持一致。

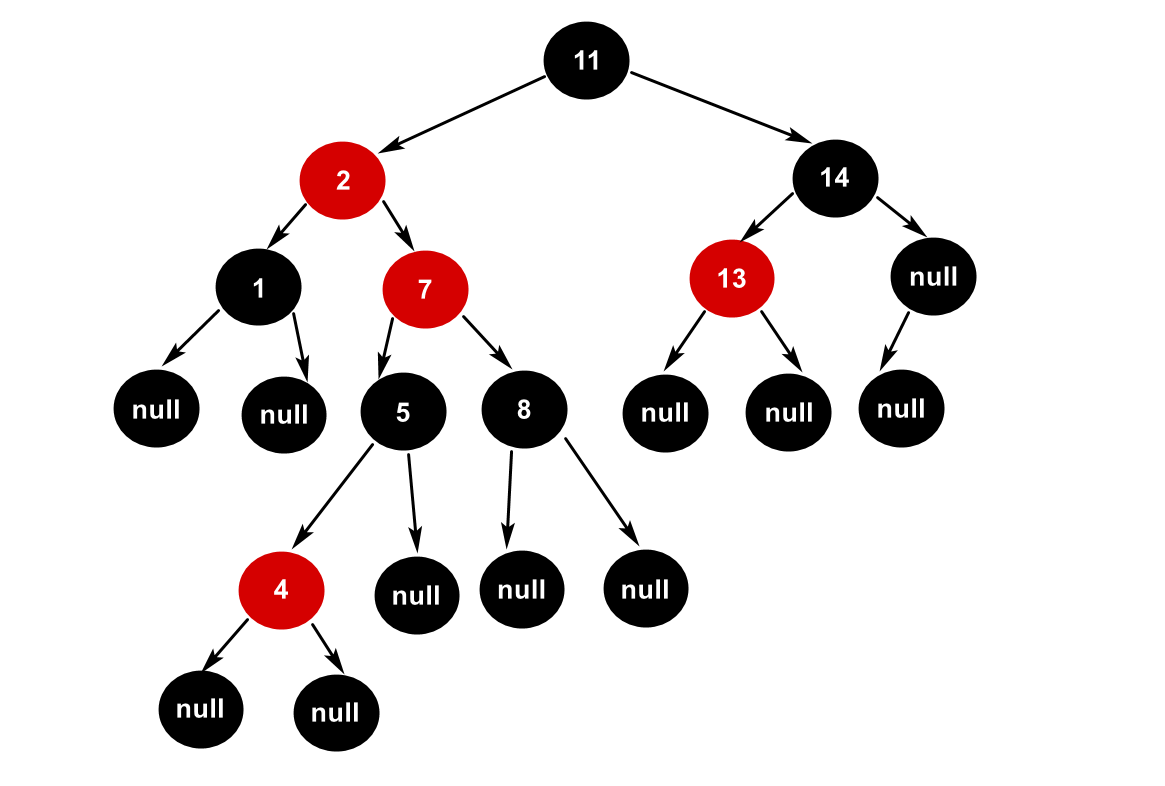


##### 红黑树涂色修正-02

在红黑树新增节点的时候一定是红色的，那么很有可能就会出现以下情况，两个红色节点在一起这样就破坏了红黑树的特征。



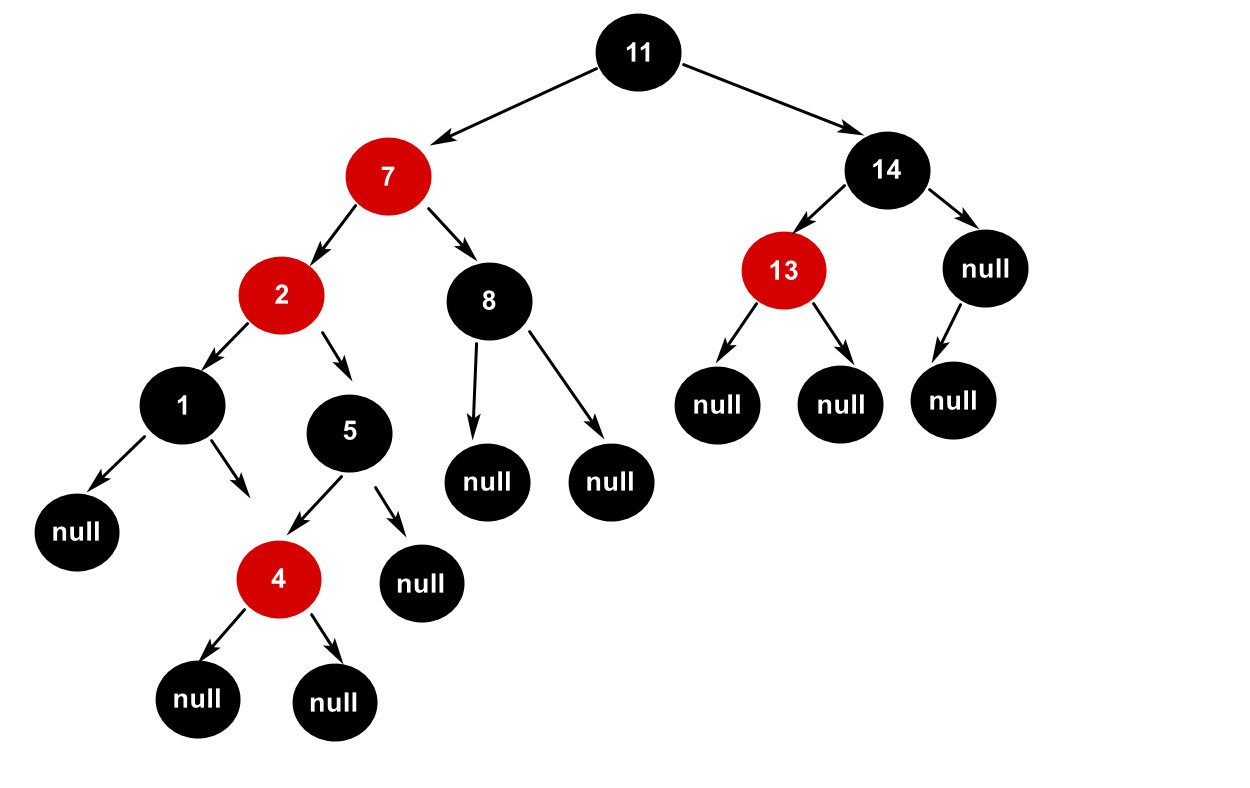
在这种情况下如果当前红色节点的父节点及叔父节点都为红色，那么将叔父节点变为黑色再将祖父节点变为红色。修正之后发现其实还是有问题，2与7节点还是两个红色并且现在7的叔父节点为黑色，不能再次使用涂色来进行修正



##### 红黑树左旋修正

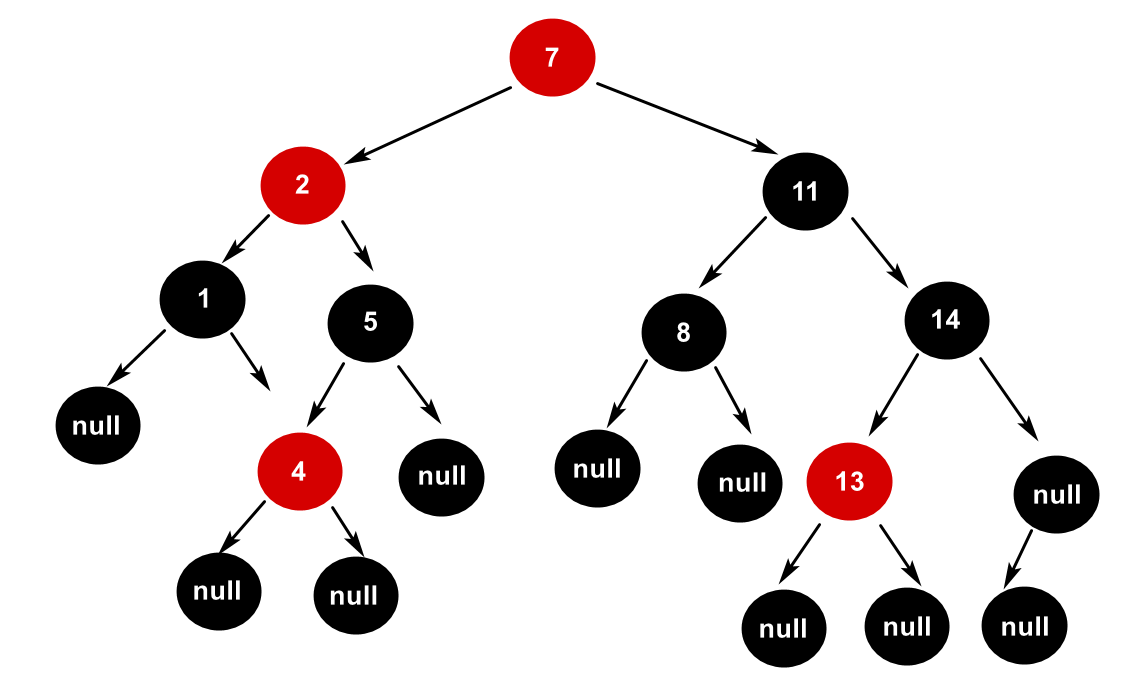
如果父子节点均为红色，叔父节点为黑色，且当前节点为右子叶，则以当前节点为轴进行左旋。将7往上提，2往左下推，推动过程中7左边所有节点将会挂载在2的节点下，由于2右边的节点数字一定比2大，所以直接挂载在2的右边。

在这一次修正之后发现还是没有完全的修正完全，所以在目前的情况下还需要再一次进行修正，这次修正称之为右旋。



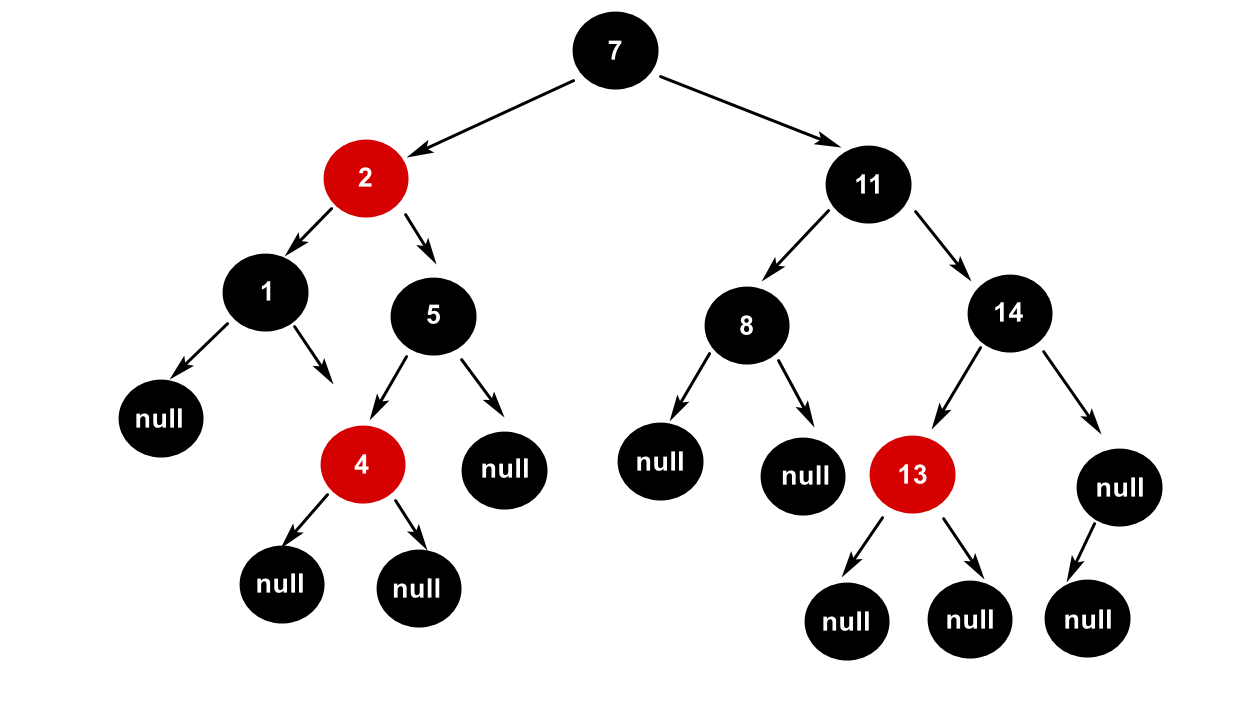
##### 红黑树右旋修正

如果父子节点均为红色，叔父节点为黑色，且当前节点为左子夜，则以父节点为轴进行右旋。这一次修整完成之后看到7的节点变成了红色，按照红黑树的特性，根节点一定为黑色，所以这里需要将7节点变成黑色。

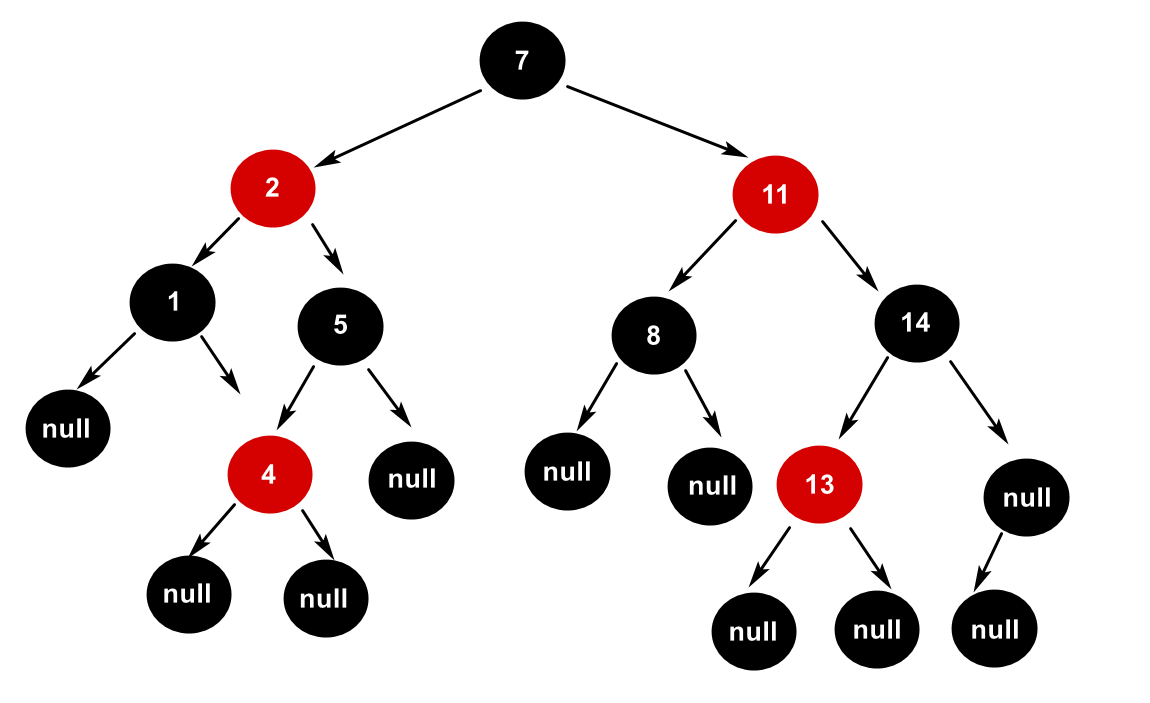


##### 案例最终的修正

将7节点变成黑色之后发现左右的节点高度又不均衡了，这种情况一般会将红节点少的一方进行涂红，即这个位置需要将11的节点变成红色。



最终修正结果。



红黑树的时间复杂度是O(logn)

### 2.3.浅谈时间复杂度

#### 2.3.1.时间复杂度概述

在一段逻辑中，将某一段会必然/重复执行的逻辑的执行时间看作是单位1，考虑随着元素个数变化，单位1的执行次数 -> 这个映射关系就是时间复杂度。在时间复杂度中用O()表示。

例如：

for(int i = 0; i < arr.length; i++){

System.out.println(arr[i]); // 将这行代码的执行时间看作单位1

}

随着arr的元素个数增多，单位1的执行次数也在增多，映射关系： f(n) = n -> 时间复杂度O(n)

时间复杂度中，如果出现多项式和系数，那么除非系数能够大到影响变化趋势，否则只考虑最高阶且忽略系数

#### 2.3.2.冒泡排序的时间复杂度

冒泡排序的代码结构：

for(){

for(){

if(){} // 将if结构的执行时间看作为1

}

}

推导过程：

第1轮，比较n-1次

第2轮，比较n-2次

第3轮，比较n-3次

...

第n-1轮，比较1次

f(n) = (n-1) + (n-2) + (n-3) + ... + 1

= (n-1+1)(n-1)/2

= n\*(n-1)/2

-> n(n-1)

-> n^2 - n -> n^2

此时时间复杂度就是O(n^2)

#### 2.3.3.二分查抄的时间复杂度

如果在时间复杂度中出现对数，对数的底数默认为2

二分查找代码结构：

while(){

...

}

假设一个有序数组有n个元素，利用二分查找去定位元素位置：

第1次查找，没有找到，剩余n\*1/2=n/(2^1)

第2次查找，没有找到，剩余n\*1/2\*1/2=n/(2^2)

第3次查找，没有找到，剩余n\*1/2\*1/2\*1/2=n/(2^3)

...

第x次查找，找到元素，剩余1=n/(2^x)

找到次数和元素个数的映射关系

-> n = 2^x -> f(n) = logn -> 时间复杂度是O(logn)

### 4.分段锁读写锁以及CSA

#### 4.1.分段锁

ConcurrentHashMap是一个异步线程安全的映射：采用了分段/桶锁机制：

意思是对当前操作的桶进行加锁，有线程在使用当前桶的时候不允许其他线程进行操作，相比对象锁来说效率搞了。

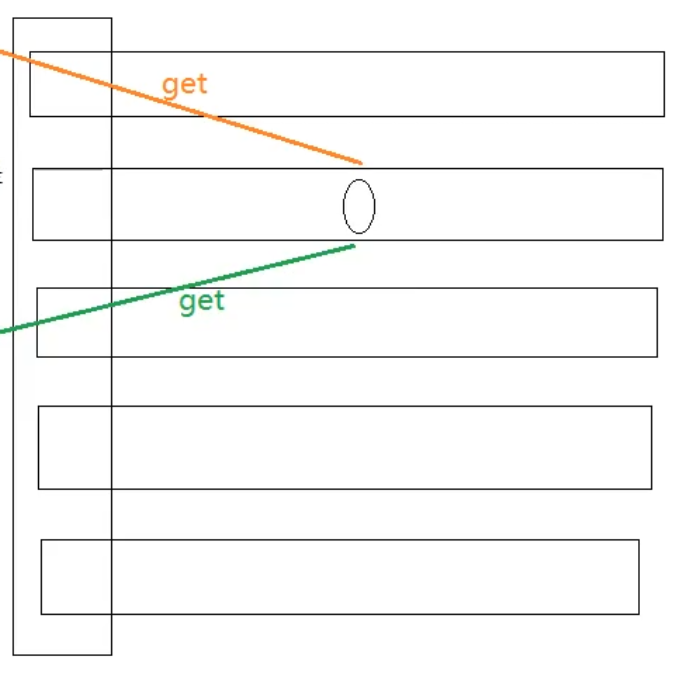
#### 4.2.读写锁

在后续版本中，ConcurrentHashMap在分段锁基础上引入了读写锁机制：

读锁：允许多个线程同时读，不允许线程写

写锁：只允许一个线程写，不允许线程读

ConcurrentHashMap利用锁保证线程安全，但是在使用锁的时候，造成CPU的资源浪费(例如线程调度，线程上下文切换等)。



#### 4.3.CSA比较和交换

在JDK1.8中，考虑到锁所带来的开销，引入了无锁算法CAS(Compare And Swap，比较和交换)。因为CAS涉及到线程的重新调度问题，所以需要结合具体的CPU内核架构来设计，因此Java中的CAS底层是依靠C语言实现的。目前市面上，服务器中流行的内核架构都是支持CAS的。

计算机生成了可选文字:
CAS 的 语文： 我 认 为 V 的 值 应 该 是 A 如 果 是 ， 那 么 将 V 的 值 更 新 为 B 
诉 V 的 值 实 际 为 多 少 
否 则 不 修 改 并 告 
B ． 新 的 预 期 值 
内 存 值 
A ： 旧 的 预 期 值 
V ： i 的 内 存 地 址 
5 
V ： i 的 内 存 地 址 
A ： \ 4 
CAS 的 过 程 一 旦 被 打 断 ， 月 仫 就 会 重 新 开 始 

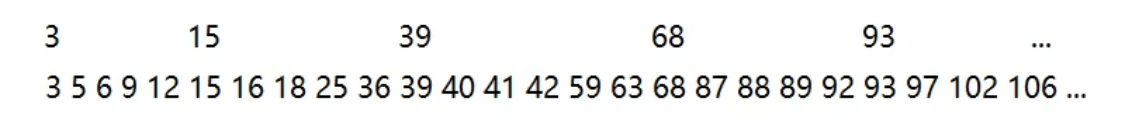
### 5.ConcurrentNavigableMap - 并发导航映射

提供了用于截取子映射的方法，元素自动排序：

ConcurrentNavigableMap是一个接口，所以使用的是它的实现类ConcurrentSkipListMap - 并发跳跃表映射 - 底层是基于跳跃表实现的

#### 5.1.跳跃表特性

在原来的的列表上进行跳跃提取元素，之后进行元素查询的时候会根据提取出来的表范围进行查询，所以ConcurrentNavigableMap适合查询多增删少的操作。



1.针对有序列表进行操作

2.适用于查询多而增删少的场景

3.典型的"以空间换时间"的产物

4.跳跃表允许进行多层提取，最上层跳跃表的元素个数不能少于2个

5.如果在跳跃表中新添了元素，那么新添的元素是否要提取到上层的跳跃表中遵循"抛硬币"原则

跳跃表的时间复杂度是O(logn)

#### 5.2.测试代码

package cn.tedu.concurrentmap;

import java.util.concurrent.ConcurrentNavigableMap;  
import java.util.concurrent.ConcurrentSkipListMap;  
  
public class ConcurrentNavigableMapDemo {  
  
 public static void main(String[] args) {  
  
 // 提供了用于截取子映射的方法  
 ConcurrentNavigableMap<String, Integer> map =  
 new ConcurrentSkipListMap<>();  
 map.put("d", 5);  
 map.put("e", 3);  
 map.put("a", 4);  
 map.put("y", 6);  
 map.put("u", 5);  
 map.put("o", 9);  
 map.put("m", 0);  
 System.*out*.println(map);  
 // 从头开始截取到指定的位置  
 System.*out*.println(map.headMap("e"));  
 // 从指定位置开始截取到末尾  
 System.*out*.println(map.tailMap("e"));  
 // 截取指定范围内的元素  
 System.*out*.println(map.subMap("e", "u"));  
  
 }  
  
}

# CAS算法

## 1.锁的代价

锁是用来做并发最简单的方式，当然其代价也是最高的，内核态的锁的时候需要操作系统进行一次上下文切换，加锁、释放锁会导致比较多的上下文切换和调度延时，等待锁的线程会被挂起直至锁释放。

在上下文切换的时候，cpu之前缓存的指令和数据都将失效，对性能有很大的损失，用户态的锁虽然避免了这些问题，但是其实它们只是在没有真实的竞争时才有效。

Java在JDK1.5之前都是靠synchronized关键字保证同步的，这种通过使用一致的锁定协议来协调对共享状态的访问，可以确保无论哪个线程持有守护变量的锁，都采用独占的方式来访问这些变量。

如果出现多个线程同时访问锁，那未抢到锁的线程将被挂起，当线程恢复执行时，必须等待其它线程执行完他们的时间片以后才能被调度执行，线程在挂起和恢复执行过程中存在着很大的开销。

当一个线程正在等待锁时，它不能做任何事，如果一个线程在持有锁的情况下被延迟执行，那么所有需要这个锁的线程都无法执行下去，如果被阻塞的线程优先级高，而持有锁的线程优先级低，将会导致优先级反转(Priority Inversion)。

## 2.独占锁

独占锁是一种悲观锁，synchronized就是一种独占锁，它是假设最坏的情况，并且只有在确保其它线程不会造成干扰的情况下执行，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。

## 3.CAS

CAS的本意为Compare And Swap，是一种无锁算法。

CAS需要结合CPU指令实现，现在大多数处理器架构都是支持CAS算法的。

CAS的语义：我认为V的值应该是A，如果是，那么将V的值更新为B，否则不修改并告诉V的值实际为多少。

CAS是一种乐观锁技术，即当多个线程尝试使用CAS同时更新同一个变量的时候，只有其中一个线程能更新变量的值，而其他线程失败。失败的线程并不会被挂起，而是会被告知在本次竞争中失败，并且可以再次尝试。

CAS有3个参数：内存值V，旧的预期值A，要修改的值B，当且仅当预期值A和内存值V相同时，将内存值修改为B，否则什么都不做。

### 3.1.C语言实现CAS

**int** compare\_and\_swap (**int**\* reg, **int** oldval, **int** newval) {

    ATOMIC();

**int** old\_reg\_val = \*reg;

**if** (old\_reg\_val == oldval)

        \*reg = newval; END\_ATOMIC();

**return** old\_reg\_val;

}

# ExecutorService - 执行器服务

## 1.概述

ExecutorService本质上是一个线程池。意义：减少服务器端的线程的创建和销毁，来提高线程资源的利用率，线程池刚创建的时候是空的，每过来一个请求，就会在线程池中创建一个核心线程来处理这个请求。核心线程的数量在定义线程池的时候需要指定。

## 2.线程池的工作原理

### 2.1.核心线程

核心线程用完之后不会被销毁而是等待下一个请求，只要核心线程没有达到指定的数量，那么每一个请求都会触发创建一个新的核心线程处理。

### 2.2.工作队列

如果核心线程被全部占用，那么新来的请求会放到工作队列中进行排队等待。工作队列本质上是一个阻塞式队列。

### 2.3.临时线程

如果工作队列被全部占用，那么新来的请求会交给临时线程来处理。临时线程的数量在定义线程池的时候需要指定。

临时线程用完之后会存活一段时间，如果在这段时间内没有接收到新的任务那么就会被销毁，工作队列中的任务不会被临时线程执行：尽量缩短临时线程的存活时间，尽量提高核心线程的利用率。

如果临时线程被全部占用，那么新来的请求会交给拒绝执行处理器来处理。

计算机生成了可选文字:
c 0 r e t h r e a d 5 
temporary 
hread 5 
work queue 7 
RejectedExecutionHand1er 

## 3.测试代码

 package cn.tedu.threadpool;

import java.util.concurrent.\*;  
  
public class ExecutorServiceDemo {  
  
 public static void main(String[] args) {  
  
 // corePoolSize - 核心线程数量  
 // maximumPoolSize - 总线程数量 = 核心线程数 + 临时线程数  
 // keepAliveTime - 临时线程的存活时间  
 // unit - 时间单位  
 // workQueue - 工作队列  
 // handler - 拒绝执行处理器，可以不指定  
 ExecutorService es = new ThreadPoolExecutor(  
 5, // 5个核心线程  
 12, // 7个临时线程  
 5, TimeUnit.*SECONDS*, // 临时线程用完之后能够存活5s  
 new ArrayBlockingQueue<Runnable>(5),  
 new RejectedExecutionHandler() {  
 @Override  
 public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {  
 // 如果有拒绝流程，那么需要将拒绝流程写到这个方法中  
 // 例如：日志记录(请求、时间、IP等)，跳转页面(努力加载中，失败了 - 秒杀等)等  
 System.*out*.println(r + "被拒绝了~~~");  
 }  
 }  
 );  
  
 // new Thread(new ESThread()).start();  
 // 核心 - 5 临时 - 7 队列 - 5  
 // 12个start，3个拒绝 -> 3s之后再有5个start  
 for (int i = 0; i < 20; i++) {  
 es.execute(new ESThread());  
 }  
 // 如果线程池用完，可以关闭线程池  
 // 实际开发中，线程池开启之后一般是不关的  
 es.shutdown();  
  
 }  
  
}  
  
class ESThread implements Runnable {  
 @Override  
 public void run() {  
 try {  
 System.*out*.println("start");  
 Thread.*sleep*(3000);  
 //System.out.println("finish");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}