
Table of Contents

Introduction	1.1
概述	1.2
ArrayList	1.3
LinkedList	1.4
Stack and Queue	1.5
TreeSet and TreeMap	1.6
HashSet and HashMap	1.7
LinkedHashSet and LinkedHashMap	1.8
PriorityQueue	1.9
WeakHashMap	1.10

深入理解 Java集合类

Java Collections Framework Internals

Authors

Name	Weibo Id	Blog	Mail
李豪	@计算所的小鼠标	CarpenterLee	hooleeucas@163.com

Introduction

关于C++标准模板库(*Standard Template Library, STL*)的书籍和资料有很多，关于Java集合框架(*Java Collections Framework, JCF*)的资料却很少，甚至很难找到一本专门介绍它的书籍，这给Java学习者们带来不小的麻烦。我深深的不解其中的原因。虽然JCF设计参考了STL，但其定位不是Java版的STL，而是要实现一个精简紧凑的容器框架，对STL的介绍自然不能替代对JCF的介绍。

本系列文章主要从数据结构和算法层面分析JCF中List, Set, Map, Stack, Queue等典型容器，结合生动图解和源代码，帮助读者对Java集合框架建立清晰而深入的理解。本文并不特意介绍Java的语言特性，但会在需要的时候做出简洁的解释。

Contents

具体内容安排如下：

1. [Overview](#) 对Java Collections Framework, 以及Java语言特性做出基本介绍。
2. [ArrayList](#) 结合源码对ArrayList进行讲解。
3. [LinkedList](#) 结合源码对LinkedList进行讲解。
4. [Stack and Queue](#) 以AarrayDeque为例讲解Stack和Queue。
5. [TreeSet and TreeMap](#) 结合源码对TreeSet和TreeMap进行讲解。
6. [HashSet and HashMap](#) 结合源码对HashSet和HashMap进行讲解。
7. [LinkedHashSet and LinkedHashMap](#) 结合源码对LinkedHashSet和LinkedHashMap进行讲解。
8. [PriorityQueue](#) 结合源码对PriorityQueue进行讲解。
9. [WeakHashMap](#) 对WeakHashMap做出基本介绍。

概览

容器，就是可以容纳其他Java对象的对象。Java Collections Framework (JCF) 为Java开发者提供了通用的容器，其始于JDK 1.2，优点是：

- 降低编程难度
- 提高程序性能
- 提高API间的互操作性
- 降低学习难度
- 降低设计和实现相关API的难度
- 增加程序的重用性

Java容器里只能放对象，对于基本类型 (int, long, float, double等)，需要将其包装成对象类型后 (Integer, Long, Float, Double等) 才能放到容器里。很多时候拆包装和解包装能够自动完成。这虽然会导致额外的性能和空间开销，但简化了设计和编程。

泛型 (Generics)

Java容器能够容纳任何类型的对象，这一点表面上是通过泛型机制完成，Java泛型不是什么神奇的东西，只是编译器为我们提供的一个“语法糖”，泛型本身并不需要Java虚拟机的支持，只需要在编译阶段做一下简单的字符串替换即可。实质上Java的单继承机制才是保证这一特性的根本，因为所有的对象都是Object的子类，容器里只要能够存放Object对象就行了。事实上，所有容器的内部存放的都是Object对象，泛型机制只是简化了编程，由编译器自动帮我们完成了强制类型转换而已。JDK 1.4以及之前版本不支持泛型，类型转换需要程序员显式完成。

```
//JDK 1.4 or before
ArrayList list = new ArrayList();
list.add(new String("Monday"));
list.add(new String("Tuesday"));
list.add(new String("Wensday"));
for(int i = 0; i < list.size(); i++){
    String weekday = (String)list.get(i); //显式类型转换
    System.out.println(weekday.toUpperCase());
}
```

```
//JDK 1.5 or latter
ArrayList<String> list = new ArrayList<String>(); //参数化类型
list.add(new String("Monday"));
list.add(new String("Tuesday"));
list.add(new String("Wensday"));
for(int i = 0; i < list.size(); i++){
    String weekday = list.get(i); //隐式类型转换，编译器自动完成
    System.out.println(weekday.toUpperCase());
}
```

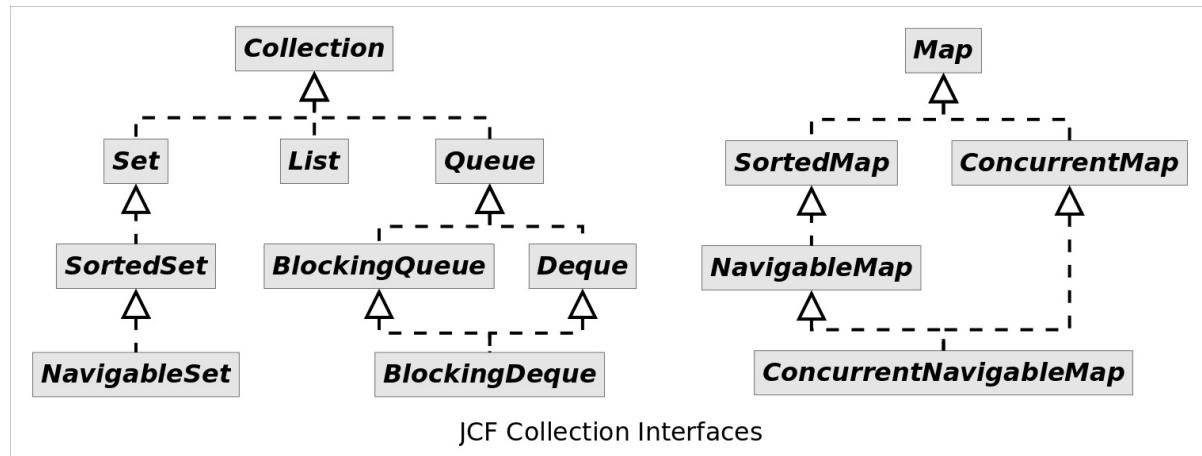
内存管理

跟C++复杂的内存管理机制不同，Java GC自动包揽了一切，Java程序并不需要处理令人头疼的内存问题，因此JCF并不像C++ STL那样需要专门的空间适配器（allocor）。另外，由于Java里对象都在堆上，且对象只能通过引用（reference，跟C++中的引用不是同一个概念，可以理解成经过包装后的指针）访问，容器里放的其实是对象的引用而不是对象本身，也就不存在C++容器的复制拷贝问题。

接口和实现（Interfaces and Implementations）

接口

为了规范容器的行为，统一设计，JCF定义了14种容器接口（collection interfaces），它们的关系如下图所示：



*Map*接口没有继承自*Collection*接口，因为*Map*表示的是关联式容器而不是集合。但Java为我们提供了从*Map*转换到*Collection*的方法，可以方便的将*Map*切换到集合视图。上图中提供了*Queue*接口，却没有*Stack*，这是因为*Stack*的功能已被JDK 1.6引入的*Deque*取代。

实现

上述接口的通用实现见下表：

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Deque		ArrayDeque		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

接下来的篇幅，会逐个介绍上表中容器的数据结构以及用到的算法。

迭代器 (Iterator)

跟C++ STL一样，JCF的迭代器 (Iterator) 为我们提供了遍历容器中元素的方法。只有容器本身清楚容器里元素的组织方式，因此迭代器只能通过容器本身得到。每个容器都会通过内部类的形式实现自己的迭代器。相比STL的迭代器，JCF的迭代器更容易使用。

```
//visit a list with iterator
ArrayList<String> list = new ArrayList<String>();
list.add(new String("Monday"));
list.add(new String("Tuesday"));
list.add(new String("Wensday"));
Iterator<String> it = list.iterator(); //得到迭代器
while(it.hasNext()){
    String weekday = it.next(); //访问元素
    System.out.println(weekday.toUpperCase());
}
```

JDK 1.5 引入了增强的for循环，简化了迭代容器时的写法。

```
//使用增强for迭代
ArrayList<String> list = new ArrayList<String>();
list.add(new String("Monday"));
list.add(new String("Tuesday"));
list.add(new String("Wensday"));
for(String weekday : list){ //enhanced for statement
    System.out.println(weekday.toUpperCase());
}
```

源代码

JDK安装目录下的src.zip包含了Java core API的源代码，本文采用的是JDK 1.7u79的源码，[下载地址](#)。[这里复制了一份](#)。

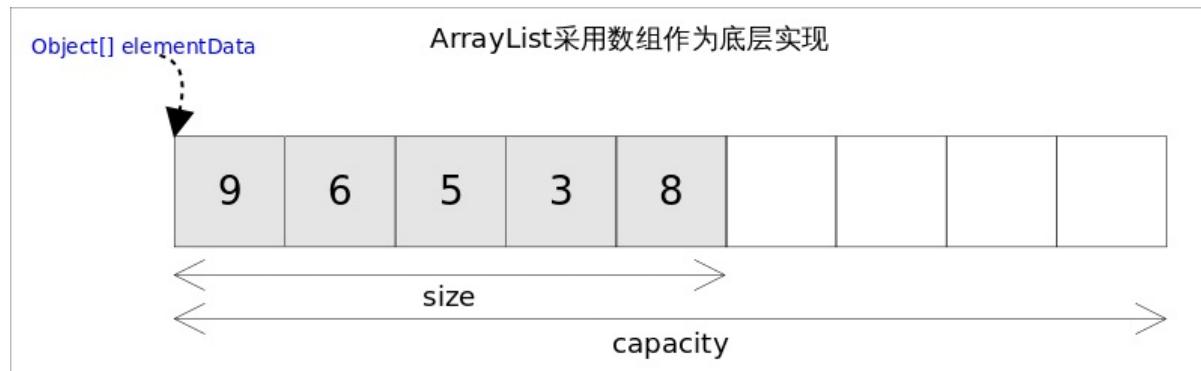
参考文献

- [Collections Framework Overview](#)
- [The For-Each Loop](#)

ArrayList

总体介绍

*ArrayList*实现了*List*接口，是顺序容器，即元素存放的数据与放进去的顺序相同，允许放入 `null` 元素，底层通过数组实现。除该类未实现同步外，其余跟*Vector*大致相同。每个*ArrayList*都有一个容量（`capacity`），表示底层数组的实际大小，容器内存储元素的个数不能多于当前容量。当向容器中添加元素时，如果容量不足，容器会自动增大底层数组的大小。前面已经提过，Java泛型只是编译器提供的语法糖，所以这里的数组是一个Object数组，以便能够容纳任何类型的对象。



`size()`, `isEmpty()`, `get()`, `set()`方法均能在常数时间内完成，`add()`方法的时间开销跟插入位置有关，`addAll()`方法的时间开销跟添加元素的个数成正比。其余方法大都是线性时间。

为追求效率，*ArrayList*没有实现同步（`synchronized`），如果需要多个线程并发访问，用户可以手动同步，也可使用*Vector*替代。

方法剖析

set()

既然底层是一个数组*ArrayList*的 `set()` 方法也就变得非常简单，直接对数组的指定位置赋值即可。

```
public E set(int index, E element) {
    rangeCheck(index); //下标越界检查
    E oldValue = elementData(index);
    elementData[index] = element; //赋值到指定位置，复制的仅仅是引用
    return oldValue;
}
```

get()

`get()` 方法同样很简单，唯一要注意的是由于底层数组是Object[], 得到元素后需要进行类型转换。

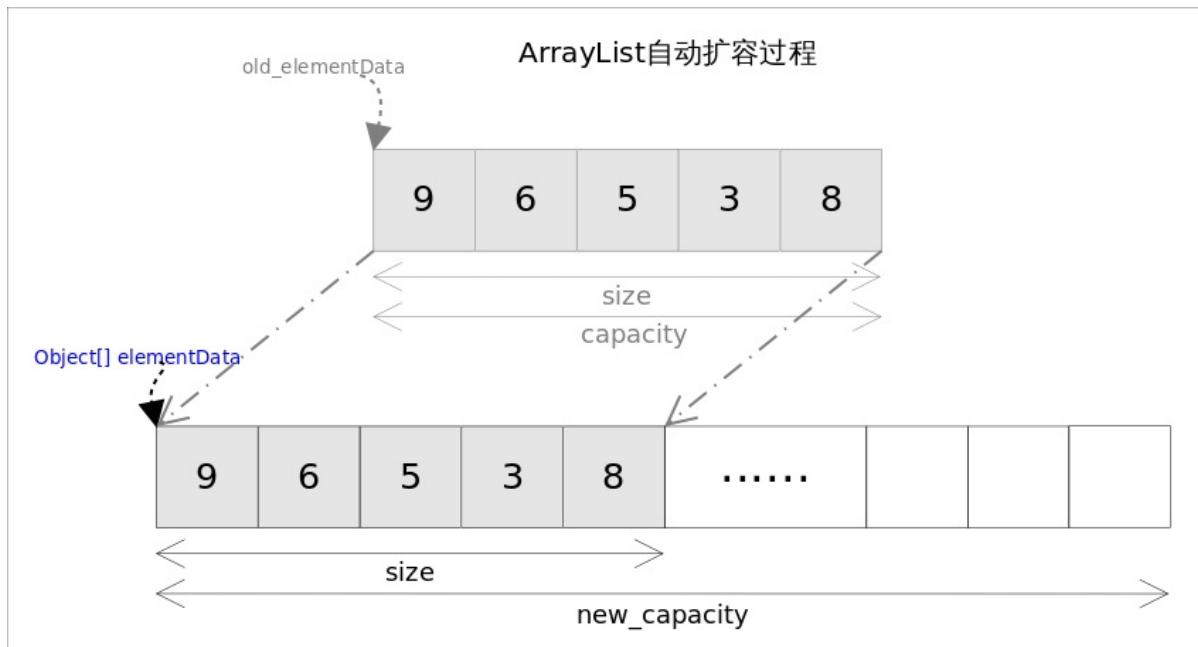
```
public E get(int index) {
    rangeCheck(index);
    return (E) elementData[index];//注意类型转换
}
```

add()

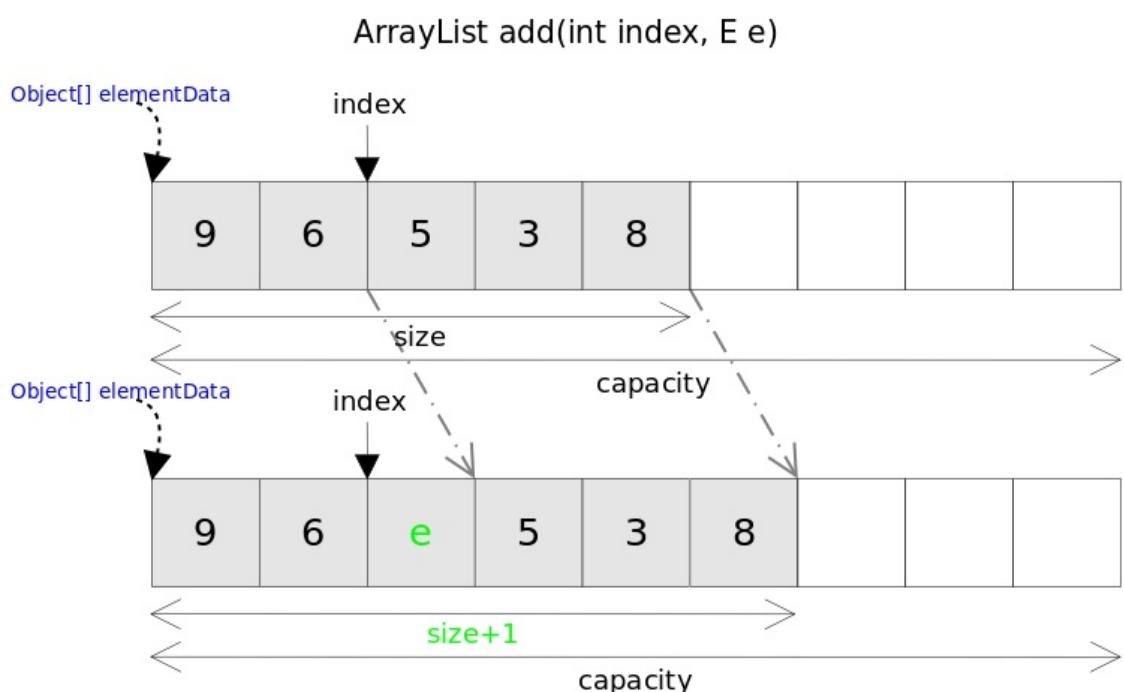
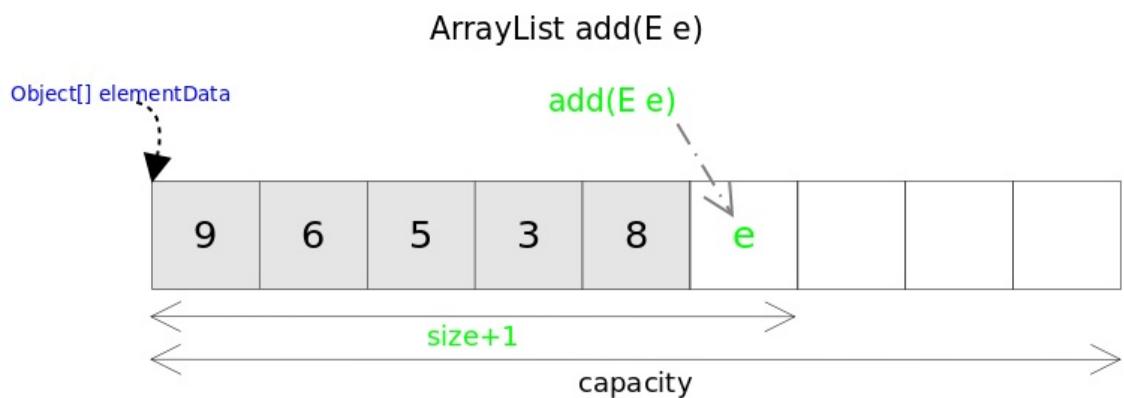
跟C++ 的`vector`不同，`ArrayList`没有 `push_back()` 方法，对应的方法是 `add(E e)`，`ArrayList`也没有 `insert()` 方法，对应的方法是 `add(int index, E e)`。这两个方法都是向容器中添加新元素，这可能会导致`capacity`不足，因此在添加元素之前，都需要进行剩余空间检查，如果需要则自动扩容。扩容操作最终是通过 `grow()` 方法完成的。

```
private void grow(int minCapacity) {
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1); //原来的1.5倍
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    elementData = Arrays.copyOf(elementData, newCapacity); //扩展空间并复制
}
```

由于Java GC自动管理了内存，这里也就不需要考虑源数组释放的问题。



空间的问题解决后，插入过程就显得非常简单。



`add(int index, E e)` 需要先对元素进行移动，然后完成插入操作，也就意味着该方法有着线性的时间复杂度。

addAll()

`addAll()` 方法能够一次添加多个元素，根据位置不同也有两个版本，一个是在末尾添加的 `addAll(Collection<? extends E> c)` 方法，一个是从指定位置开始插入的 `addAll(int index, Collection<? extends E> c)` 方法。跟 `add()` 方法类似，在插入之前也需要进行空间检查，如果需要则自动扩容；如果从指定位置插入，也会存在移动元素的情况。`addAll()` 的时间复杂度不仅跟插入元素的多少有关，也跟插入的位置相关。

remove()

`remove()` 方法也有两个版本，一个是 `remove(int index)` 删除指定位置的元素，另一个是 `remove(Object o)` 删除第一个满足 `o.equals(elementData[index])` 的元素。删除操作是 `add()` 操作的逆过程，需要将删除点之后的元素向前移动一个位置。需要注意的是为了让GC起作用，必须显式的为最后一个位置赋 `null` 值。

```
public E remove(int index) {  
    rangeCheck(index);  
    modCount++;  
    E oldValue = elementData(index);  
    int numMoved = size - index - 1;  
    if (numMoved > 0)  
        System.arraycopy(elementData, index+1, elementData, index, numMoved);  
    elementData[--size] = null; //清除该位置的引用，让GC起作用  
    return oldValue;  
}
```

关于Java GC这里需要特别说明一下，有了垃圾收集器并不意味着一定不会有内存泄漏。对象能否被GC的依据是是否还有引用指向它，上面代码中如果不手动赋 `null` 值，除非对应的位置被其他元素覆盖，否则原来的对象就一直不会被回收。

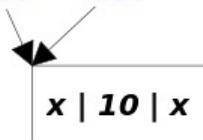
LinkedList

总体介绍

*LinkedList*同时实现了*List*接口和*Deque*接口，也就是说它既可以看作一个顺序容器，又可以看作一个队列（*Queue*），同时又可以看作一个栈（*Stack*）。这样看来，*LinkedList*简直就是个全能冠军。当你需要使用栈或者队列时，可以考虑使用*LinkedList*，一方面是因为Java官方已经声明不建议使用*Stack*类，更遗憾的是，Java里根本没有一个叫做*Queue*的类（它是个接口名字）。关于栈或队列，现在的首选是*ArrayDeque*，它有着比*LinkedList*（当作栈或队列使用时）有着更好的性能。

只有一个元素的*LinkedList*

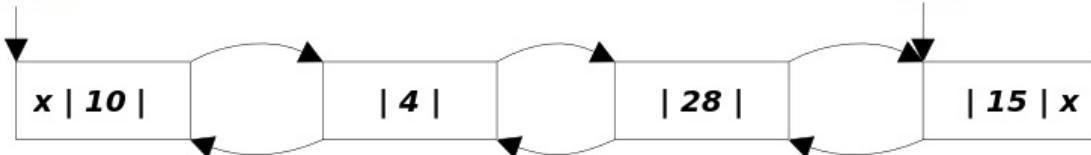
first *last*



包含四个元素的*LinkedList*

first

last



*LinkedList*底层通过双向链表实现，本节将着重讲解插入和删除元素时双向链表的维护过程，也即是直接跟*List*接口相关的函数，而将*Queue*和*Stack*以及*Deque*相关的知识放在下一节讲。双向链表的每个节点用内部类*Node*表示。*LinkedList*通过 *first* 和 *last* 引用分别指向链表的第一个和最后一个元素。注意这里没有所谓的哑元，当链表为空的时候 *first* 和 *last* 都指向 `null`。

```

//Node内部类
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
  
```

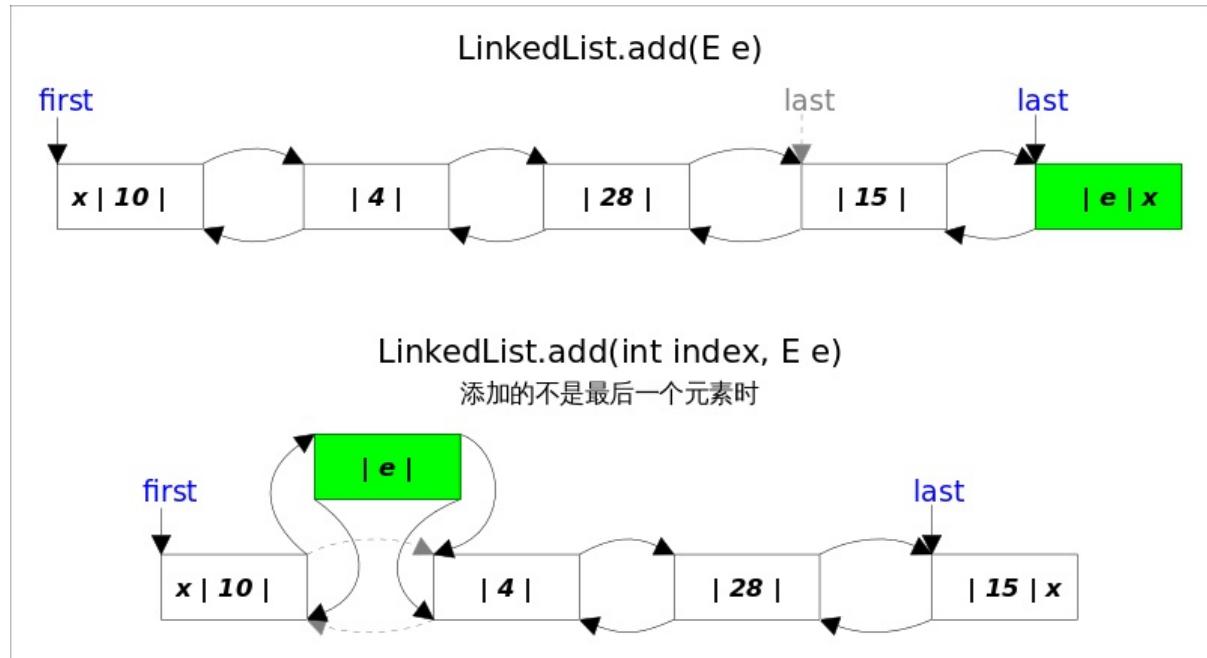
```
}
```

*LinkedList*的实现方式决定了所有跟下标相关的操作都是线性时间，而在首段或者末尾删除元素只需要常数时间。为追求效率*LinkedList*没有实现同步（synchronized），如果需要多个线程并发访问，可以先采用 `Collections.synchronizedList()` 方法对其进行包装。

方法剖析

add()

`add()`方法有两个版本，一个是 `add(E e)`，该方法在*LinkedList*的末尾插入元素，因为有 `last` 指向链表末尾，在末尾插入元素的花费是常数时间。只需要简单修改几个相关引用即可；另一个是 `add(int index, E element)`，该方法是在指定下表处插入元素，需要先通过线性查找找到具体位置，然后修改相关引用完成插入操作。



结合上图，可以看出 `add(E e)` 的逻辑非常简单。

```
//add(E e)
public boolean add(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<E>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode; //原来链表为空，这是插入的第一个元素
    else
        l.next = newNode;
    size++;
    return true;
}
```

`add(int index, E element)` 的逻辑稍显复杂，可以分成两部分，1.先根据index找到要插入的位置；2.修改引用，完成插入操作。

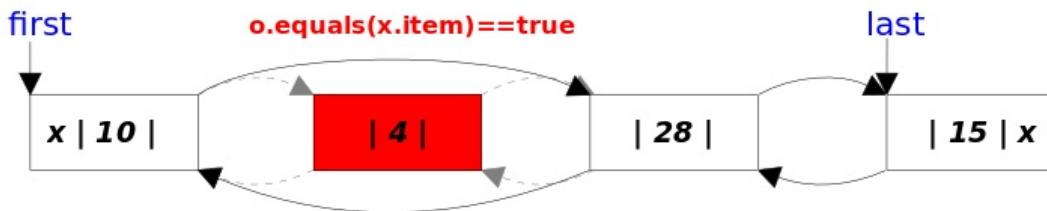
```
//add(int index, E element)
public void add(int index, E element) {
    checkPositionIndex(index); //index >= 0 && index <= size;
    if (index == size) //插入位置是末尾，包括列表为空的情况
        add(element);
    else{
        Node<E> succ = node(index); //1.先根据index找到要插入的位置
        //2.修改引用，完成插入操作。
        final Node<E> pred = succ.prev;
        final Node<E> newNode = new Node<E>(pred, e, succ);
        succ.prev = newNode;
        if (pred == null) //插入位置为0
            first = newNode;
        else
            pred.next = newNode;
        size++;
    }
}
```

上面代码中的 `node(int index)` 函数有一点小小的trick，因为链表双向的，可以从开始往后找，也可以从结尾往前找，具体朝那个方向找取决于条件 `index < (size >> 1)`，也即是index是靠近前端还是后端。

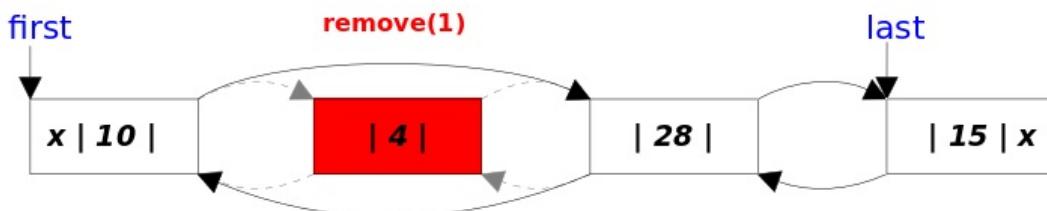
remove()

`remove()` 方法也有两个版本，一个是删除跟指定元素相等的第一个元素 `remove(Object o)`，另一个是删除指定下标处的元素 `remove(int index)`。

LinkedList.remove(Object o)



LinkedList.remove(int index)



删除之前下标	0	1	2	3
删除之后下标	0		1	2

两个删除操作都要1.先找到要删除元素的引用，2.修改相关引用，完成删除操作。在寻找被删元素引用的时候 `remove(Object o)` 调用的是元素的 `equals` 方法，而 `remove(int index)` 使用的是下标计数，两种方式都是线性时间复杂度。在步骤2中，两个 `remove()` 方法都是通过 `unlink(Node<E> x)` 方法完成的。这里需要考虑删除元素是第一个或者最后一个时的边界情况。

```
//unlink(Node<E> x), 删除一个Node
E unlink(Node<E> x) {
    final E element = x.item;
    final Node<E> next = x.next;
    final Node<E> prev = x.prev;
    if (prev == null) {//删除的是第一个元素
        first = next;
    } else {
        prev.next = next;
        x.prev = null;
    }
    if (next == null) {//删除的是最后一个元素
        last = prev;
    } else {
        next.prev = prev;
        x.next = null;
    }
    x.item = null;//let GC work
    size--;
    return element;
}
```

get()

`get(int index)` 得到指定下标处元素的引用，通过调用上文中提到的 `node(int index)` 方法实现。

```
public E get(int index) {
    checkElementIndex(index); //index >= 0 && index < size;
    return node(index).item;
}
```

set()

`set(int index, E element)` 方法将指定下标处的元素修改成指定值，也是先通过 `node(int index)` 找到对应下表元素的引用，然后修改 `Node` 中 `item` 的值。

```
public E set(int index, E element) {
    checkElementIndex(index);
    Node<E> x = node(index);
    E oldVal = x.item;
    x.item = element; //替换新值
    return oldVal;
}
```

Stack and Queue

前言

Java里有一个叫做`Stack`的类，却没有叫做`Queue`的类（它是个接口名字）。当需要使用栈时，Java已不推荐使用`Stack`，而是推荐使用更高效的`ArrayDeque`；既然`Queue`只是一个接口，当需要使用队列时也就首选`ArrayDeque`了（次选是`LinkedList`）。

总体介绍

要讲栈和队列，首先要讲`Deque`接口。`Deque`的含义是“double ended queue”，即双端队列，它既可以当作栈使用，也可以当作队列使用。下表列出了`Deque`与`Queue`相对应的接口：

Queue Method	Equivalent Deque Method	说明
<code>add(e)</code>	<code>addLast(e)</code>	向队尾插入元素，失败则抛出异常
<code>offer(e)</code>	<code>offerLast(e)</code>	向队尾插入元素，失败则返回 <code>false</code>
<code>remove()</code>	<code>removeFirst()</code>	获取并删除队首元素，失败则抛出异常
<code>poll()</code>	<code>pollFirst()</code>	获取并删除队首元素，失败则返回 <code>null</code>
<code>element()</code>	<code>getFirst()</code>	获取但不删除队首元素，失败则抛出异常
<code>peek()</code>	<code>peekFirst()</code>	获取但不删除队首元素，失败则返回 <code>null</code>

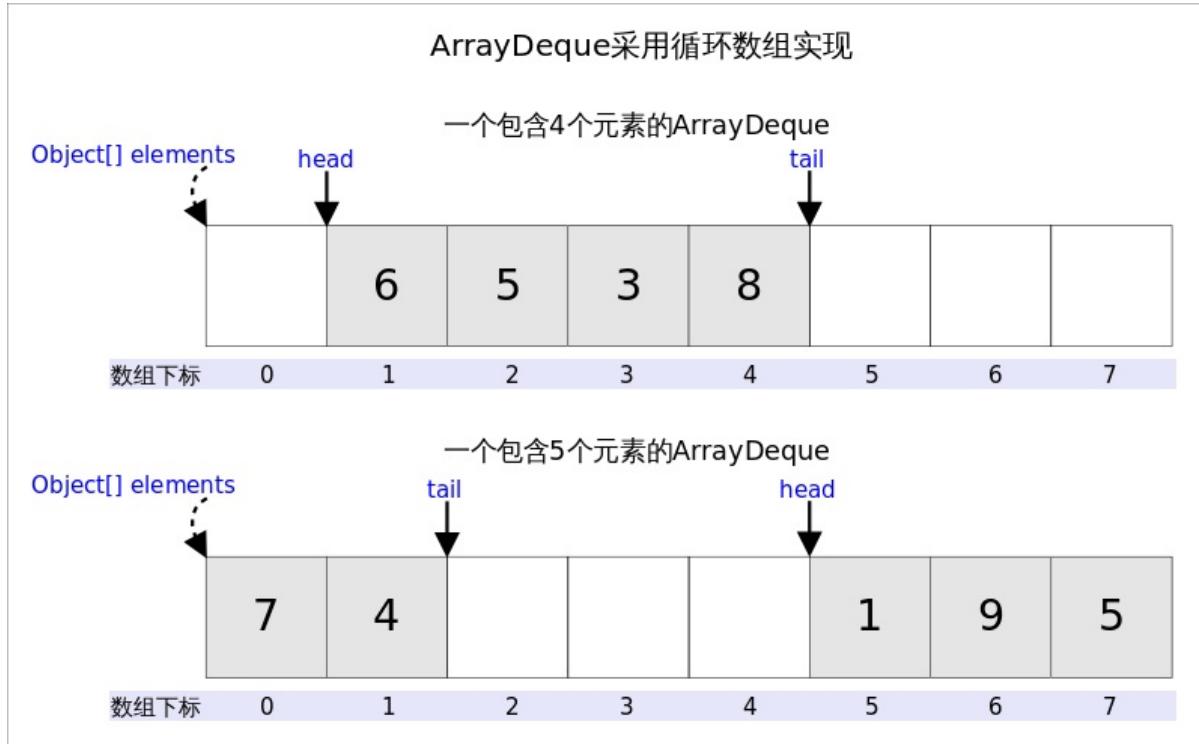
下表列出了`Deque`与`Stack`对应的接口：

Stack Method	Equivalent Deque Method	说明
<code>push(e)</code>	<code>addFirst(e)</code>	向栈顶插入元素，失败则抛出异常
无	<code>offerFirst(e)</code>	向栈顶插入元素，失败则返回 <code>false</code>
<code>pop()</code>	<code>removeFirst()</code>	获取并删除栈顶元素，失败则抛出异常
无	<code>pollFirst()</code>	获取并删除栈顶元素，失败则返回 <code>null</code>
<code>peek()</code>	<code>peekFirst()</code>	获取但不删除栈顶元素，失败则抛出异常
无	<code>peekFirst()</code>	获取但不删除栈顶元素，失败则返回 <code>null</code>

上面两个表共定义了`Deque`的12个接口。添加，删除，取值都有两套接口，它们功能相同，区别是对失败情况的处理不同。一套接口遇到失败就会抛出异常，另一套遇到失败会返回特殊值（`false`或`null`）。除非某种实现对容量有限制，大多数情况下，添加操作是不会失败的。虽然`Deque`的接口有12个之多，但无非就是对容器的两端进行操作，或添加，或删除，或查看。明白了这一点讲解起来就会非常简单。

`ArrayDeque`和`LinkedList`是`Deque`的两个通用实现，由于官方更推荐使用`ArrayDeque`用作栈和队列，加之上一篇已经讲解过`LinkedList`，本文将着重讲解`ArrayDeque`的具体实现。

从名字可以看出 `ArrayDeque` 底层通过数组实现，为了满足可以同时在数组两端插入或删除元素的需求，该数组还必须是循环的，即循环数组（circular array），也就是说数组的任何一点都可能被看作起点或者终点。`ArrayDeque` 是非线程安全的（not thread-safe），当多个线程同时使用的时候，需要程序员手动同步；另外，该容器不允许放入 `null` 元素。

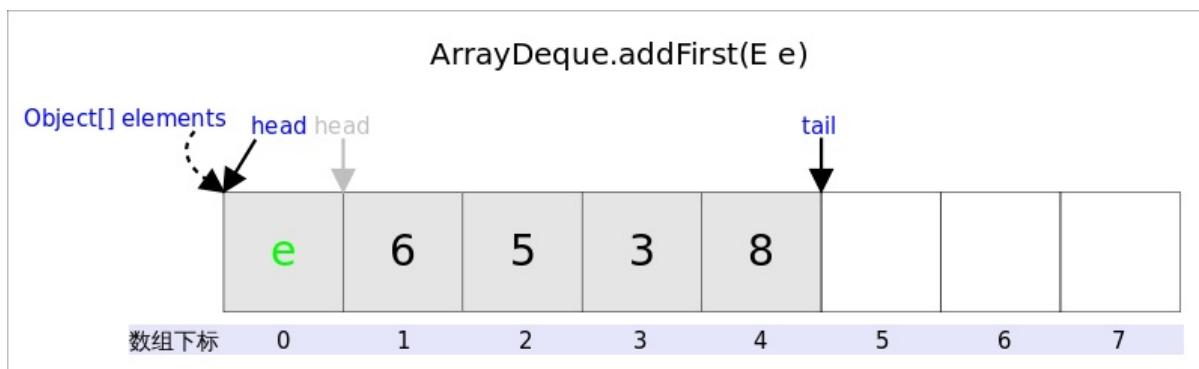


上图中我们看到，`head` 指向首端第一个有效元素，`tail` 指向尾端第一个可以插入元素的空位。因为是循环数组，所以 `head` 不一定总等于0，`tail` 也不一定总是比 `head` 大。

方法剖析

`addFirst()`

`addFirst(E e)` 的作用是在 `Deque` 的首端插入元素，也就是在 `head` 的前面插入元素，在空间足够且下标没有越界的情况下，只需要将 `elements[--head] = e` 即可。



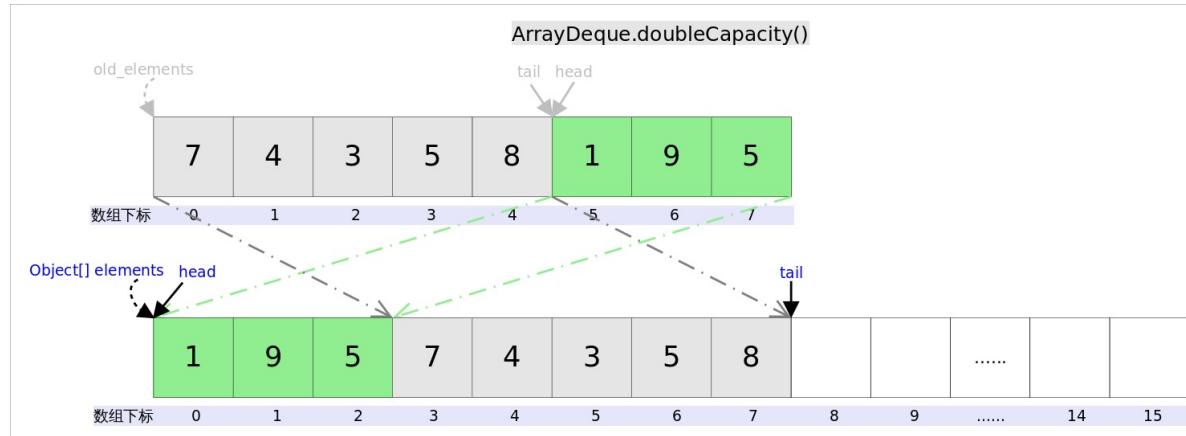
实际需要考虑：1.空间是否够用，以及2.下标是否越界的问题。上图中，如果 `head` 为 0 之后接着调用 `addFirst()`，虽然空余空间还够用，但 `head` 为 -1，下标越界了。下列代码很好的解决了这两个问题。

```
//addFirst(E e)
public void addFirst(E e) {
    if (e == null)//不允许放入null
        throw new NullPointerException();
    elements[head = (head - 1) & (elements.length - 1)] = e;//2.下标是否越界
    if (head == tail)//1.空间是否够用
        doubleCapacity();//扩容
}
```

上述代码我们看到，空间问题是在插入之后解决的，因为 `tail` 总是指向下一个可插入的空位，也就意味着 `elements` 数组至少有一个空位，所以插入元素的时候不用考虑空间问题。

下标越界的处理解决起来非常简单，`head = (head - 1) & (elements.length - 1)` 就可以了，这段代码相当于取余，同时解决了 `head` 为负值的情况。因为 `elements.length` 必需是 2 的指数倍，`elements - 1` 就是二进制低位全 1，跟 `head - 1` 相与之后就起到了取模的作用，如果 `head - 1` 为负数（其实只可能是-1），则相当于对其取相对于 `elements.length` 的补码。

下面再说说扩容函数 `doubleCapacity()`，其逻辑是申请一个更大的数组（原数组的两倍），然后将原数组复制过去。过程如下图所示：



图中我们看到，复制分两次进行，第一次复制 `head` 右边的元素，第二次复制 `head` 左边的元素。

```
//doubleCapacity()
private void doubleCapacity() {
    assert head == tail;
    int p = head;
    int n = elements.length;
    int r = n - p; // head右边元素的个数
    int newCapacity = n << 1;//原空间的2倍
    if (newCapacity < 0)
        throw new IllegalStateException("Sorry, deque too big");
    Object[] a = new Object[newCapacity];
```

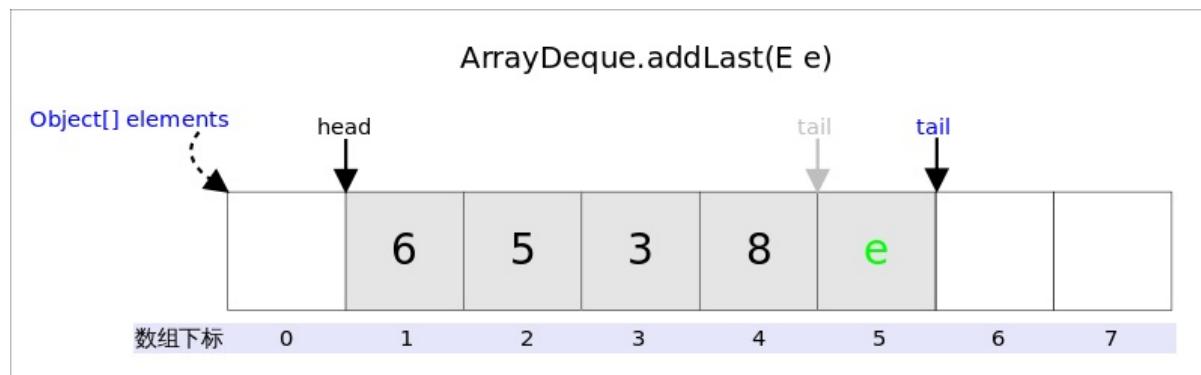
```

System.arraycopy(elements, p, a, 0, r); //复制右半部分, 对应上图中绿色部分
System.arraycopy(elements, 0, a, r, p); //复制左半部分, 对应上图中灰色部分
elements = (E[])a;
head = 0;
tail = n;
}

```

addLast()

`addLast(E e)` 的作用是在 `Deque` 的尾端插入元素，也就是在 `tail` 的位置插入元素，由于 `tail` 总是指向下一个可以插入的空位，因此只需要 `elements[tail] = e;` 即可。插入完成后再检查空间，如果空间已经用光，则调用 `doubleCapacity()` 进行扩容。



```

public void addLast(E e) {
    if (e == null) //不允许放入null
        throw new NullPointerException();
    elements[tail] = e; //赋值
    if ((tail = (tail + 1) & (elements.length - 1)) == head) //下标越界处理
        doubleCapacity(); //扩容
}

```

下标越界处理方式 `addFirst()` 中已经讲过，不再赘述。

pollFirst()

`pollFirst()` 的作用是删除并返回 `Deque` 首端元素，也即是 `head` 位置处的元素。如果容器不空，只需要直接返回 `elements[head]` 即可，当然还需要处理下标的问题。由于 `ArrayDeque` 中不允许放入 `null`，当 `elements[head] == null` 时，意味着容器为空。

```

public E pollFirst() {
    E result = elements[head];
    if (result == null) //null值意味着deque为空
        return null;
    elements[h] = null; //let GC work
    head = (head + 1) & (elements.length - 1); //下标越界处理
    return result;
}

```

```
}
```

pollLast()

`pollLast()` 的作用是删除并返回 `Deque` 尾端元素，也即是 `tail` 位置前面的那个元素。

```
public E pollLast() {
    int t = (tail - 1) & (elements.length - 1); // tail的上一个位置是最后一个元素
    E result = elements[t];
    if (result == null) // null值意味着 deque 为空
        return null;
    elements[t] = null; // let GC work
    tail = t;
    return result;
}
```

peekFirst()

`peekFirst()` 的作用是返回但不删除 `Deque` 首端元素，也即是 `head` 位置处的元素，直接返回 `elements[head]` 即可。

```
public E peekFirst() {
    return elements[head]; // elements[head] is null if deque empty
}
```

peekLast()

`peekLast()` 的作用是返回但不删除 `Deque` 尾端元素，也即是 `tail` 位置前面的那个元素。

```
public E peekLast() {
    return elements[(tail - 1) & (elements.length - 1)];
}
```

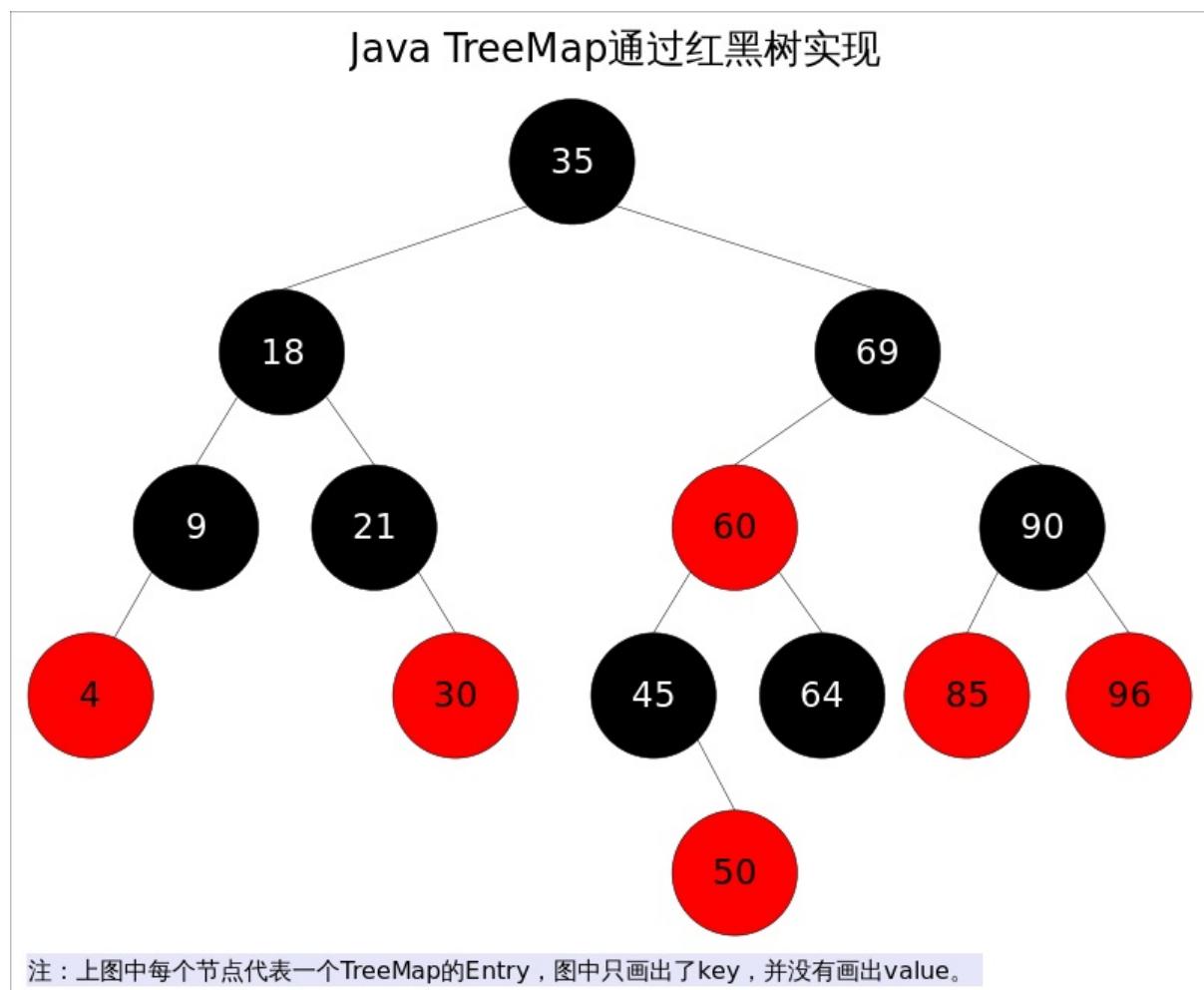
TreeSet and TreeMap

总体介绍

之所以把 *TreeSet* 和 *TreeMap* 放在一起讲解，是因为二者在 Java 里有着相同的实现，前者仅仅是对后者做了一层包装，也就是说 *TreeSet* 里面有一个 *TreeMap*（适配器模式）。因此本文将重点分析 *TreeMap*。

Java *TreeMap* 实现了 *SortedMap* 接口，也就是说会按照 `key` 的大小顺序对 *Map* 中的元素进行排序，`key` 大小的评判可以通过其本身的自然顺序（natural ordering），也可以通过构造时传入的比较器（Comparator）。

TreeMap 底层通过红黑树（Red-Black tree）实现，也就意味着 `containsKey()`, `get()`, `put()`, `remove()` 都有着 $\log(n)$ 的时间复杂度。其具体算法实现参照了《算法导论》。



出于性能原因，*TreeMap* 是非同步的（not synchronized），如果需要在多线程环境使用，需要程序员手动同步；或者通过如下方式将 *TreeMap* 包装成（wrapped）同步的：

```
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));
```

红黑树是一种近似平衡的二叉查找树，它能够确保任何一个节点的左右子树的高度差不会超过二者中较低那个的一倍。具体来说，红黑树是满足如下条件的二叉查找树（binary search tree）：

1. 每个节点要么是红色，要么是黑色。
2. 根节点必须是黑色
3. 红色节点不能连续（也即是，红色节点的孩子和父亲都不能是红色）。
4. 对于每个节点，从该点至 `null`（树尾端）的任何路径，都含有相同个数的黑色节点。

在树的结构发生改变时（插入或者删除操作），往往会破坏上述条件3或条件4，需要通过调整使得查找树重新满足红黑树的约束条件。

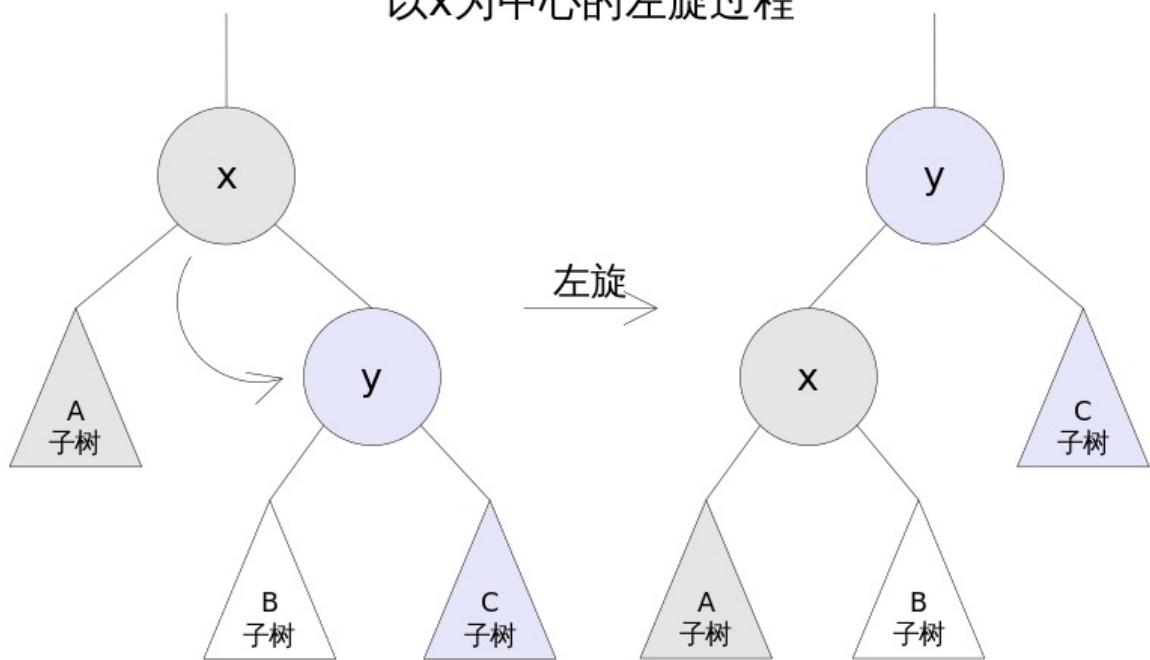
预备知识

前文说到当查找树的结构发生改变时，红黑树的约束条件可能被破坏，需要通过调整使得查找树重新满足红黑树的约束条件。调整可以分为两类：一类是颜色调整，即改变某个节点的颜色；另一类是结构调整，即改变检索树的结构关系。结构调整过程包含两个基本操作：左旋（Rotate Left），右旋（Rotate Right）。

左旋

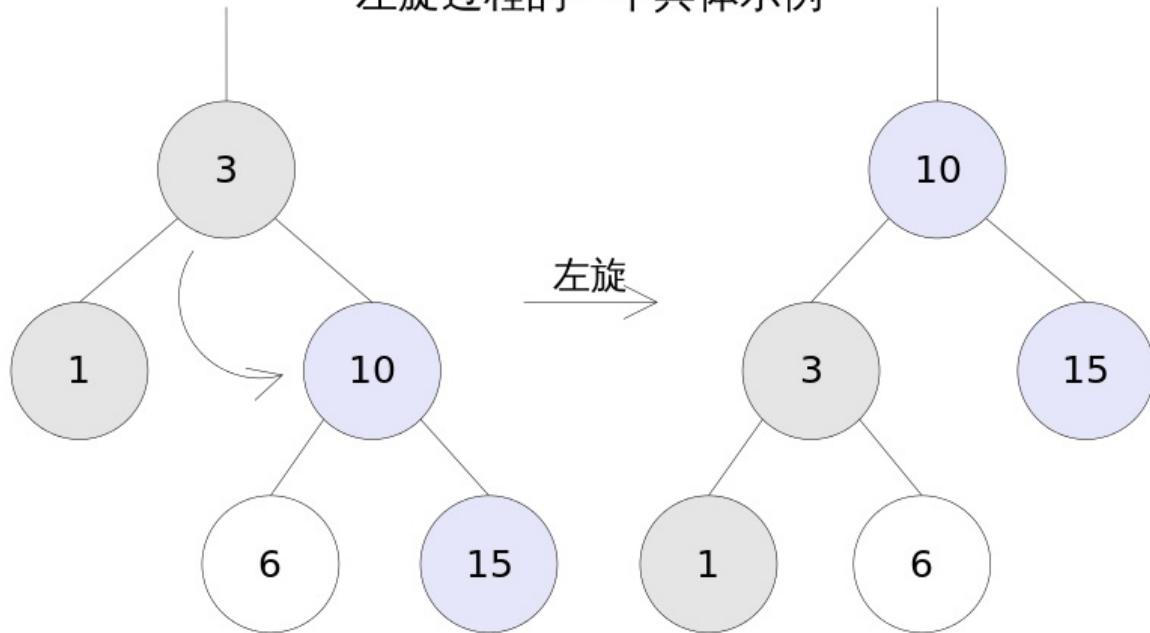
左旋的过程是将 x 的右子树绕 x 逆时针旋转，使得 x 的右子树成为 x 的父亲，同时修改相关节点的引用。旋转之后，二叉查找树的属性仍然满足。

以x为中心的左旋过程



注：上图中各子树可以是多个节点构成的子树，也可以是一个具体节点，也可是null。

左旋过程的一个具体示例



TreeMap中左旋代码如下：

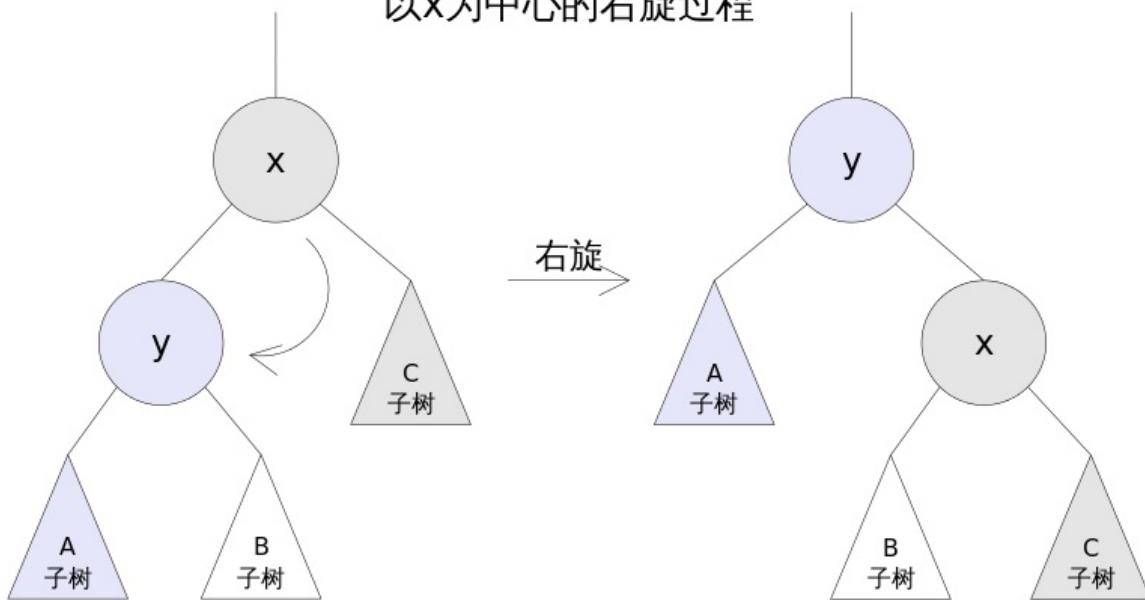
```
//Rotate Left
private void rotateLeft(Entry<K,V> p) {
    if (p != null) {
        Entry<K,V> r = p.right;
        p.right = r.left;
        if (r.left != null)
```

```
    r.left.parent = p;
    r.parent = p.parent;
    if (p.parent == null)
        root = r;
    else if (p.parent.left == p)
        p.parent.left = r;
    else
        p.parent.right = r;
    r.left = p;
    p.parent = r;
}
}
```

右旋

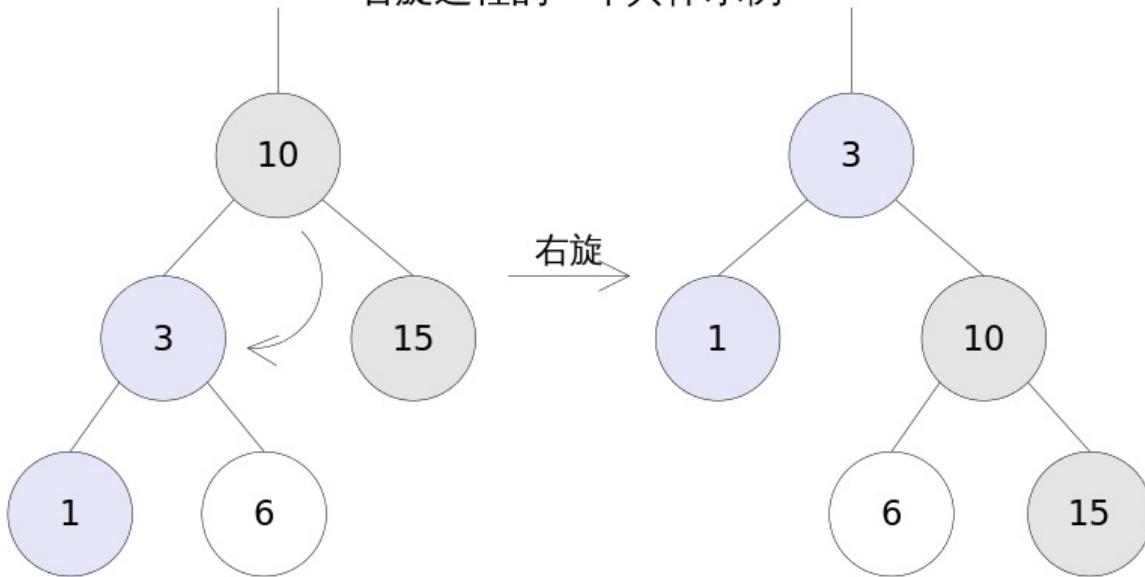
右旋的过程是将 x 的左子树绕 x 顺时针旋转，使得 x 的左子树成为 x 的父亲，同时修改相关节点的引用。旋转之后，二叉查找树的属性仍然满足。

以x为中心的右旋过程



注：上图中各子树可以是多个节点构成的子树，也可以是一个具体节点，也可是null。

右旋过程的一个具体示例



TreeMap中右旋代码如下：

```
//Rotate Right
private void rotateRight(Entry<K,V> p) {
    if (p != null) {
        Entry<K,V> l = p.left;
        p.left = l.right;
        if (l.right != null) l.right.parent = p;
        l.parent = p.parent;
        if (p.parent == null)
            root = l;
        else if (p.parent.right == p)
```

```

        p.parent.right = l;
    else p.parent.left = l;
    l.right = p;
    p.parent = l;
}
}

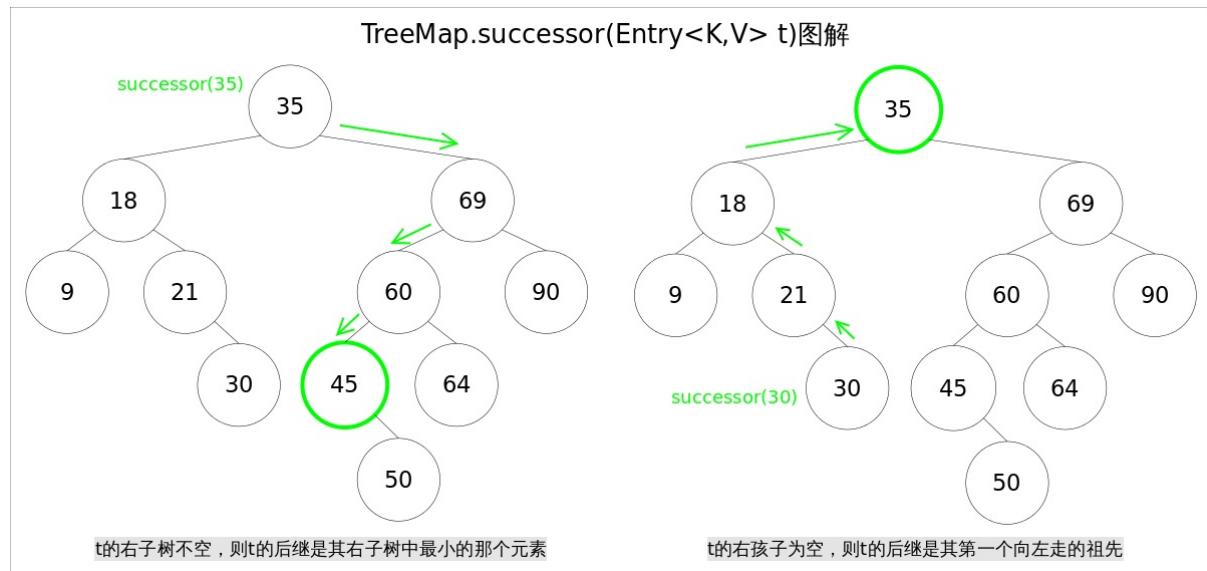
```

寻找节点后继

对于一棵二叉查找树，给定节点t，其后继（树中比大于t的最小的那个元素）可以通过如下方式找到：

1. t的右子树不空，则t的后继是其右子树中最小的那个元素。
2. t的右孩子为空，则t的后继是其第一个向左走的祖先。

后继节点在红黑树的删除操作中将会用到。



TreeMap中寻找节点后继的代码如下：

```

// 寻找节点后继函数successor()
static <K,V> TreeMap.Entry<K,V> successor(TreeMap.Entry<K,V> t) {
    if (t == null)
        return null;
    else if (t.right != null) {// 1. t的右子树不空，则t的后继是其右子树中最小的那个元素
        Entry<K,V> p = t.right;
        while (p.left != null)
            p = p.left;
        return p;
    } else { // 2. t的右孩子为空，则t的后继是其第一个向左走的祖先
        Entry<K,V> p = t.parent;
        Entry<K,V> ch = t;
        while (p != null && ch == p.right) {
            ch = p;
            p = p.parent;
        }
        return p;
    }
}

```

```

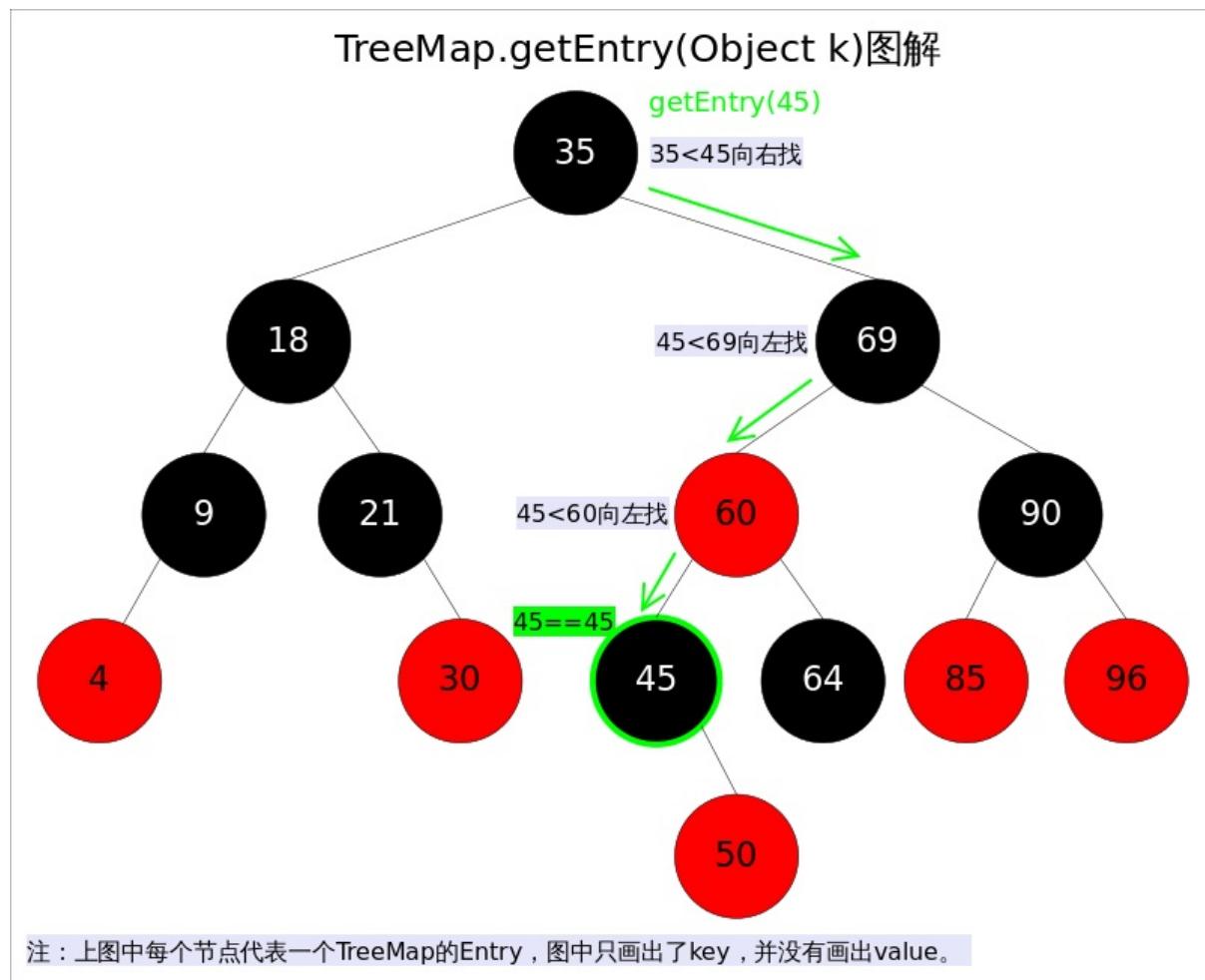
    }
    return p;
}
}

```

方法剖析

get()

`get(Object key)` 方法根据指定的 `key` 值返回对应的 `value`，该方法调用了 `getEntry(Object key)` 得到相应的 `entry`，然后返回 `entry.value`。因此 `getEntry()` 是算法的核心。算法思想是根据 `key` 的自然顺序（或者比较器顺序）对二叉查找树进行查找，直到找到满足 `k.compareTo(p.key) == 0` 的 `entry`。



具体代码如下：

```

//getEntry()方法
final Entry<K,V> getEntry(Object key) {
    .....
    if (key == null)//不允许key值为null

```

```

        throw new NullPointerException();
Comparable<? super K> k = (Comparable<? super K>) key;//使用元素的自然顺序
Entry<K,V> p = root;
while (p != null) {
    int cmp = k.compareTo(p.key);
    if (cmp < 0)//向左找
        p = p.left;
    else if (cmp > 0)//向右找
        p = p.right;
    else
        return p;
}
return null;
}

```

put()

`put(K key, V value)` 方法是将指定的 `key` , `value` 对添加到 `map` 里。该方法首先会对 `map` 做一次查找，看是否包含该元组，如果已经包含则直接返回，查找过程类似于 `getEntry()` 方法；如果没有找到则会在红黑树中插入新的 `entry` ，如果插入之后破坏了红黑树的约束条件，还需要进行调整（旋转，改变某些节点的颜色）。

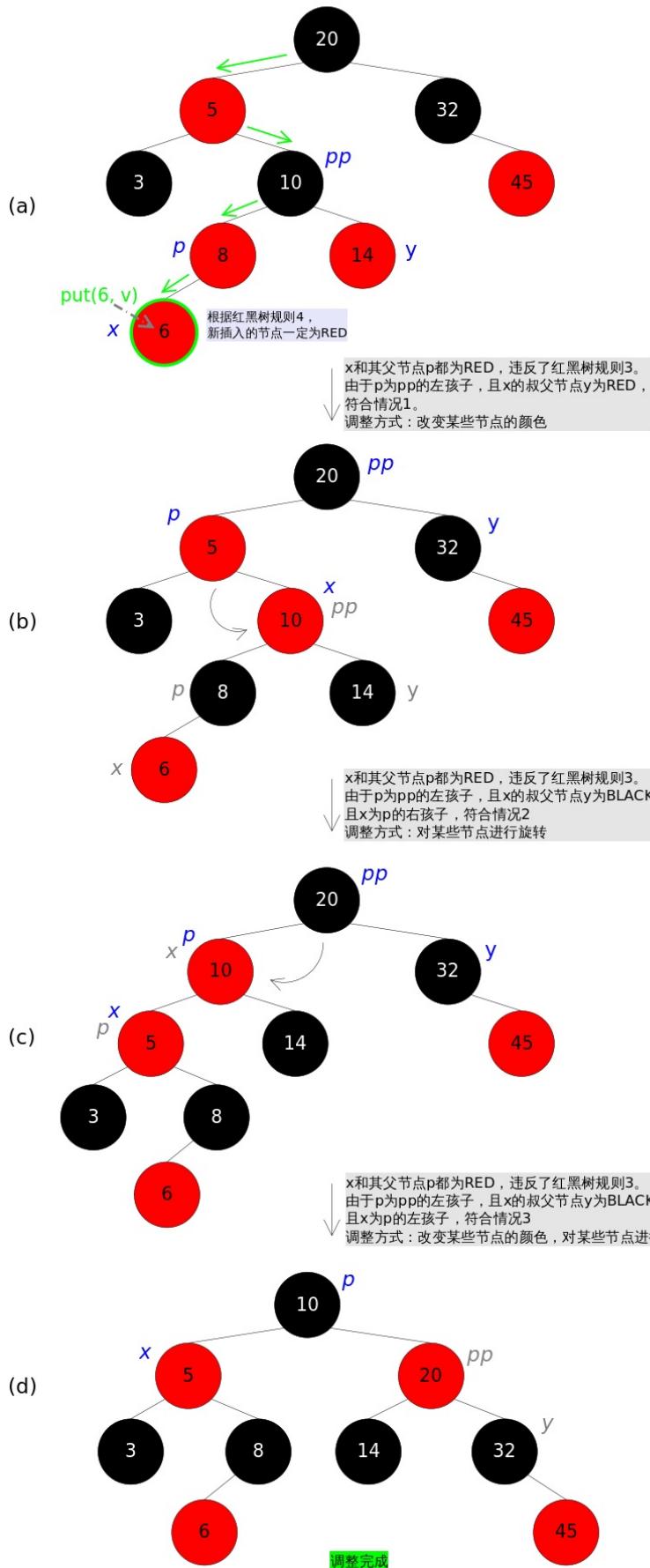
```

public V put(K key, V value) {
    .....
    int cmp;
    Entry<K,V> parent;
    if (key == null)
        throw new NullPointerException();
Comparable<? super K> k = (Comparable<? super K>) key;//使用元素的自然顺序
do {
    parent = t;
    cmp = k.compareTo(t.key);
    if (cmp < 0) t = t.left;//向左找
    else if (cmp > 0) t = t.right;//向右找
    else return t.setValue(value);
} while (t != null);
Entry<K,V> e = new Entry<>(key, value, parent);//创建并插入新的entry
if (cmp < 0) parent.left = e;
else parent.right = e;
fixAfterInsertion(e);//调整
size++;
return null;
}

```

上述代码的插入部分并不难理解：首先在红黑树上找到合适的位置，然后创建新的 `entry` 并插入（当然，新插入的节点一定是树的叶子）。难点是调整函数 `fixAfterInsertion()`，前面已经说过，调整往往需要1.改变某些节点的颜色，2.对某些节点进行旋转。

TreeMap.put(K key, V value)和调整过程图解



调整函数 `fixAfterInsertion()` 的具体代码如下，其中用到了上文中提到的 `rotateLeft()` 和 `rotateRight()` 函数。通过代码我们能够看到，情况2其实是落在情况3内的。情况4~情况6跟前三种情况是对称的，因此图解中并没有画出后三种情况，读者可以参考代码自行理解。

```
//红黑树调整函数fixAfterInsertion()
private void fixAfterInsertion(Entry<K,V> x) {
    x.color = RED;
    while (x != null && x != root && x.parent.color == RED) {
        if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {
            Entry<K,V> y = rightOf(parentOf(parentOf(x)));
            if (colorOf(y) == RED) {
                setColor(parentOf(x), BLACK);           // 情况1
                setColor(y, BLACK);                   // 情况1
                setColor(parentOf(parentOf(x)), RED);   // 情况1
                x = parentOf(parentOf(x));           // 情况1
            } else {
                if (x == rightOf(parentOf(x))) {
                    x = parentOf(x);                 // 情况2
                    rotateLeft(x);                  // 情况2
                }
                setColor(parentOf(x), BLACK);       // 情况3
                setColor(parentOf(parentOf(x)), RED); // 情况3
                rotateRight(parentOf(parentOf(x))); // 情况3
            }
        } else {
            Entry<K,V> y = leftOf(parentOf(parentOf(x)));
            if (colorOf(y) == RED) {
                setColor(parentOf(x), BLACK);       // 情况4
                setColor(y, BLACK);               // 情况4
                setColor(parentOf(parentOf(x)), RED); // 情况4
                x = parentOf(parentOf(x));       // 情况4
            } else {
                if (x == leftOf(parentOf(x))) {
                    x = parentOf(x);                 // 情况5
                    rotateRight(x);                // 情况5
                }
                setColor(parentOf(x), BLACK);       // 情况6
                setColor(parentOf(parentOf(x)), RED); // 情况6
                rotateLeft(parentOf(parentOf(x))); // 情况6
            }
        }
    }
    root.color = BLACK;
}
```

remove()

`remove(Object key)` 的作用是删除 `key` 值对应的 `entry`，该方法首先通过上文中提到的 `getEntry(Object key)` 方法找到 `key` 值对应的 `entry`，然后调用 `deleteEntry(Entry<K,V> entry)` 删除对应的 `entry`。由于删除操作会改变红黑树的结构，有可能破坏红黑树的约束条件，因此有可能要进行调整。

`getEntry()` 函数前面已经讲解过，这里重点放 `deleteEntry()` 上，该函数删除指定的 `entry` 并在红黑树的约束被破坏时进行调用 `fixAfterDeletion(Entry<K,V> x)` 进行调整。

由于红黑树是一棵增强版的二叉查找树，红黑树的删除操作跟普通二叉查找树的删除操作也就非常相似，唯一的区别是红黑树在节点删除之后可能需要进行调整。现在考虑一棵普通二叉查找树的删除过程，可以简单分为两种情况：

1. 删除点p的左右子树都为空，或者只有一棵子树非空。
2. 删除点p的左右子树都非空。

对于上述情况1，处理起来比较简单，直接将p删除（左右子树都为空时），或者用非空子树替代p（只有一棵子树非空时）；对于情况2，可以用p的后继s（树中大于x的最小的那个元素）代替p，然后使用情况1删除s（此时s一定满足情况1.可以画画看）。

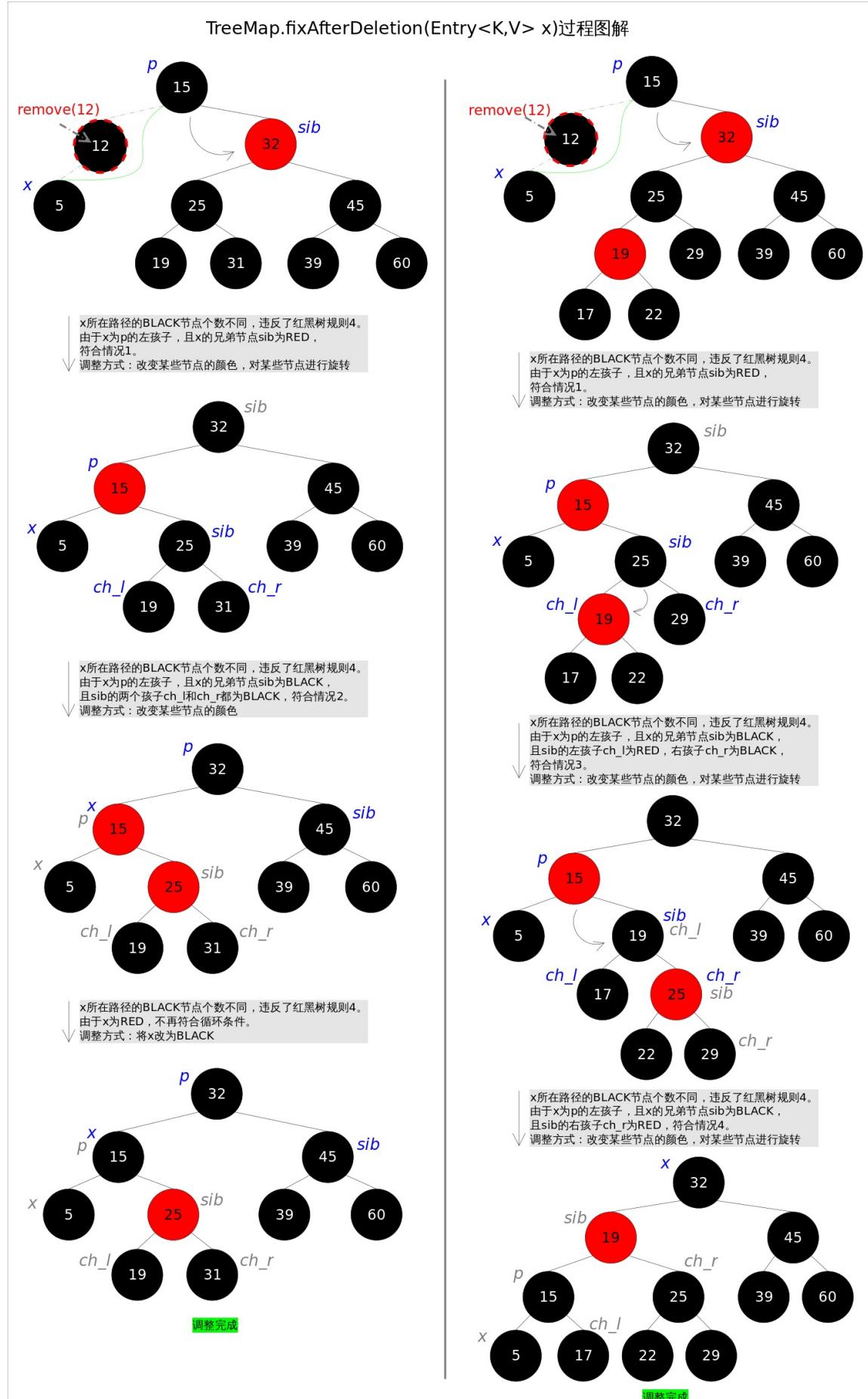
基于以上逻辑，红黑树的节点删除函数 `deleteEntry()` 代码如下：

```
// 红黑树entry删除函数deleteEntry()
private void deleteEntry(Entry<K,V> p) {
    modCount++;
    size--;
    if (p.left != null && p.right != null) { // 2. 删除点p的左右子树都非空。
        Entry<K,V> s = successor(p); // 后继
        p.key = s.key;
        p.value = s.value;
        p = s;
    }
    Entry<K,V> replacement = (p.left != null ? p.left : p.right);
    if (replacement != null) { // 1. 删除点p只有一棵子树非空。
        replacement.parent = p.parent;
        if (p.parent == null)
            root = replacement;
        else if (p == p.parent.left)
            p.parent.left = replacement;
        else
            p.parent.right = replacement;
        p.left = p.right = p.parent = null;
        if (p.color == BLACK)
            fixAfterDeletion(replacement); // 调整
    } else if (p.parent == null) {
        root = null;
    } else { // 1. 删除点p的左右子树都为空
        if (p.color == BLACK)
            fixAfterDeletion(p); // 调整
        if (p.parent != null) {
            if (p == p.parent.left)
```

```
        p.parent.left = null;
    else if (p == p.parent.right)
        p.parent.right = null;
    p.parent = null;
}
}
}
```

上述代码中占据大量代码行的，是用来修改父子节点间引用关系的代码，其逻辑并不难理解。下面着重讲解删除后调整函数 `fixAfterDeletion()`。首先请思考一下，删除了哪些点才会导致调整？只有删除点是BLACK的时候，才会触发调整函数，因为删除RED节点不会破坏红黑树的任何约束，而删除BLACK节点会破坏规则4。

跟上文中讲过的 `fixAfterInsertion()` 函数一样，这里也要分成若干种情况。记住，无论有多少情况，具体的调整操作只有两种：1.改变某些节点的颜色，2.对某些节点进行旋转。



上述图解的总体思想是：将情况1首先转换成情况2，或者转换成情况3和情况4。当然，该图解并不意味着调整过程一定是从情况1开始。通过后续代码我们还会发现几个有趣的规则：a).如果是由情况1之后紧接着进入的情况2，那么情况2之后一定会退出循环（因为x为红色）；b).一旦进入情况3和情况4，一定会退出循环（因为x为root）。

删除后调整函数 `fixAfterDeletion()` 的具体代码如下，其中用到了上文中提到的 `rotateLeft()` 和 `rotateRight()` 函数。通过代码我们能够看到，情况3其实是落在情况4内的。情况5~情况8跟前四种情况是对称的，因此图解中并没有画出后四种情况，读者可以参考代码自行理解。

```

private void fixAfterDeletion(Entry<K,V> x) {
    while (x != root && colorOf(x) == BLACK) {
        if (x == leftOf(parentOf(x))) {
            Entry<K,V> sib = rightOf(parentOf(x));
            if (colorOf(sib) == RED) {
                setColor(sib, BLACK); // 情况1
                setColor(parentOf(x), RED); // 情况1
                rotateLeft(parentOf(x)); // 情况1
                sib = rightOf(parentOf(x)); // 情况1
            }
            if (colorOf(leftOf(sib)) == BLACK &&
                colorOf(rightOf(sib)) == BLACK) {
                setColor(sib, RED); // 情况2
                x = parentOf(x); // 情况2
            } else {
                if (colorOf(rightOf(sib)) == BLACK) {
                    setColor(leftOf(sib), BLACK); // 情况3
                    setColor(sib, RED); // 情况3
                    rotateRight(sib); // 情况3
                    sib = rightOf(parentOf(x)); // 情况3
                }
                setColor(sib, colorOf(parentOf(x))); // 情况4
                setColor(parentOf(x), BLACK); // 情况4
                setColor(rightOf(sib), BLACK); // 情况4
                rotateLeft(parentOf(x)); // 情况4
                x = root; // 情况4
            }
        } else { // 跟前四种情况对称
            Entry<K,V> sib = leftOf(parentOf(x));
            if (colorOf(sib) == RED) {
                setColor(sib, BLACK); // 情况5
                setColor(parentOf(x), RED); // 情况5
                rotateRight(parentOf(x)); // 情况5
                sib = leftOf(parentOf(x)); // 情况5
            }
            if (colorOf(rightOf(sib)) == BLACK &&
                colorOf(leftOf(sib)) == BLACK) {
                setColor(sib, RED); // 情况6
                x = parentOf(x); // 情况6
            }
        }
    }
}

```

```

        } else {
            if (colorOf(leftOf(sib)) == BLACK) {
                setColor(rightOf(sib), BLACK);           // 情况7
                setColor(sib, RED);                     // 情况7
                rotateLeft(sib);                      // 情况7
                sib = leftOf(parentOf(x));             // 情况7
            }
            setColor(sib, colorOf(parentOf(x)));      // 情况8
            setColor(parentOf(x), BLACK);            // 情况8
            setColor(leftOf(sib), BLACK);            // 情况8
            rotateRight(parentOf(x));               // 情况8
            x = root;                            // 情况8
        }
    }
}
setColor(x, BLACK);
}

```

TreeSet

前面已经说过 TreeSet 是对 TreeMap 的简单包装，对 TreeSet 的函数调用都会转换成合适的 TreeMap 方法，因此 TreeSet 的实现非常简单。这里不再赘述。

```

// TreeSet是对TreeMap的简单包装
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
{
    .....
    private transient NavigableMap<E, Object> m;
    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();
    public TreeSet() {
        this.m = new TreeMap<E, Object>(); // TreeSet里面有一个TreeMap
    }
    .....
    public boolean add(E e) {
        return m.put(e, PRESENT)==null;
    }
    .....
}

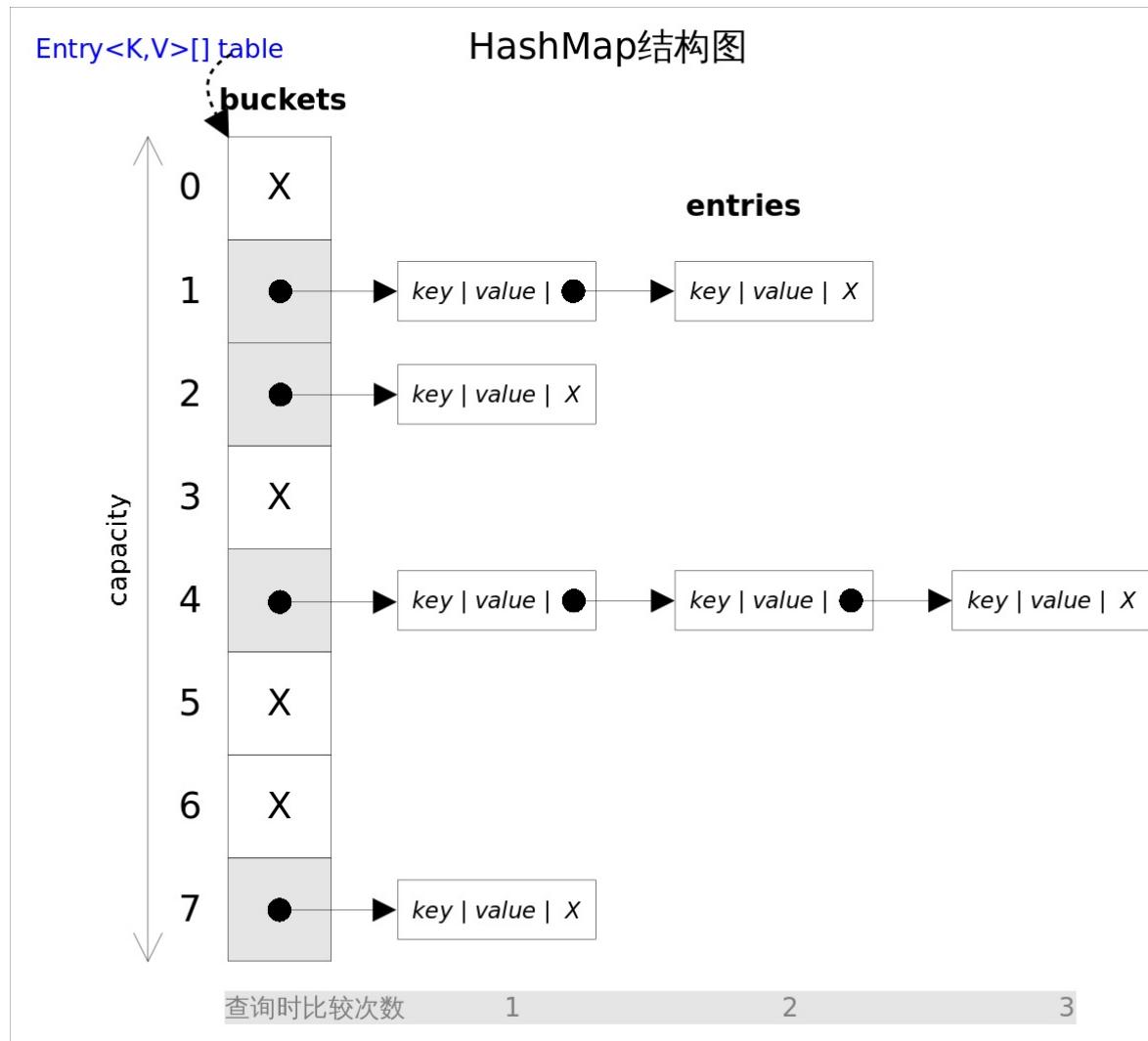
```

HashSet and HashMap

总体介绍

之所以把`HashSet`和`HashMap`放在一起讲解，是因为二者在Java里有着相同的实现，前者仅仅是对后者做了一层包装，也就是说`HashSet`里面有一个`HashMap`（适配器模式）。因此本文将重点分析`HashMap`。

`HashMap`实现了`Map`接口，即允许放入 `key` 为 `null` 的元素，也允许插入 `value` 为 `null` 的元素；除该类未实现同步外，其余跟 `Hashtable` 大致相同；跟 `TreeMap`不同，该容器不保证元素顺序，根据需要该容器可能会对元素重新哈希，元素的顺序也会被重新打散，因此不同时间迭代同一个`HashMap`的顺序可能会不同。根据对冲突的处理方式不同，哈希表有两种实现方式，一种开放地址方式（Open addressing），另一种是冲突链表方式（Separate chaining with linked lists）。Java `HashMap`采用的是冲突链表方式。



从上图容易看出，如果选择合适的哈希函数，`put()` 和 `get()` 方法可以在常数时间内完成。但在对 `HashMap` 进行迭代时，需要遍历整个 `table` 以及后面跟的冲突链表。因此对于迭代比较频繁的场景，不宜将 `HashMap` 的初始大小设的过大。

有两个参数可以影响 `HashMap` 的性能：初始容量（initial capacity）和负载系数（load factor）。初始容量指定了初始 `table` 的大小，负载系数用来指定自动扩容的临界值。当 `entry` 的数量超过 `capacity * load_factor` 时，容器将自动扩容并重新哈希。对于插入元素较多的场景，将初始容量设大可以减少重新哈希的次数。

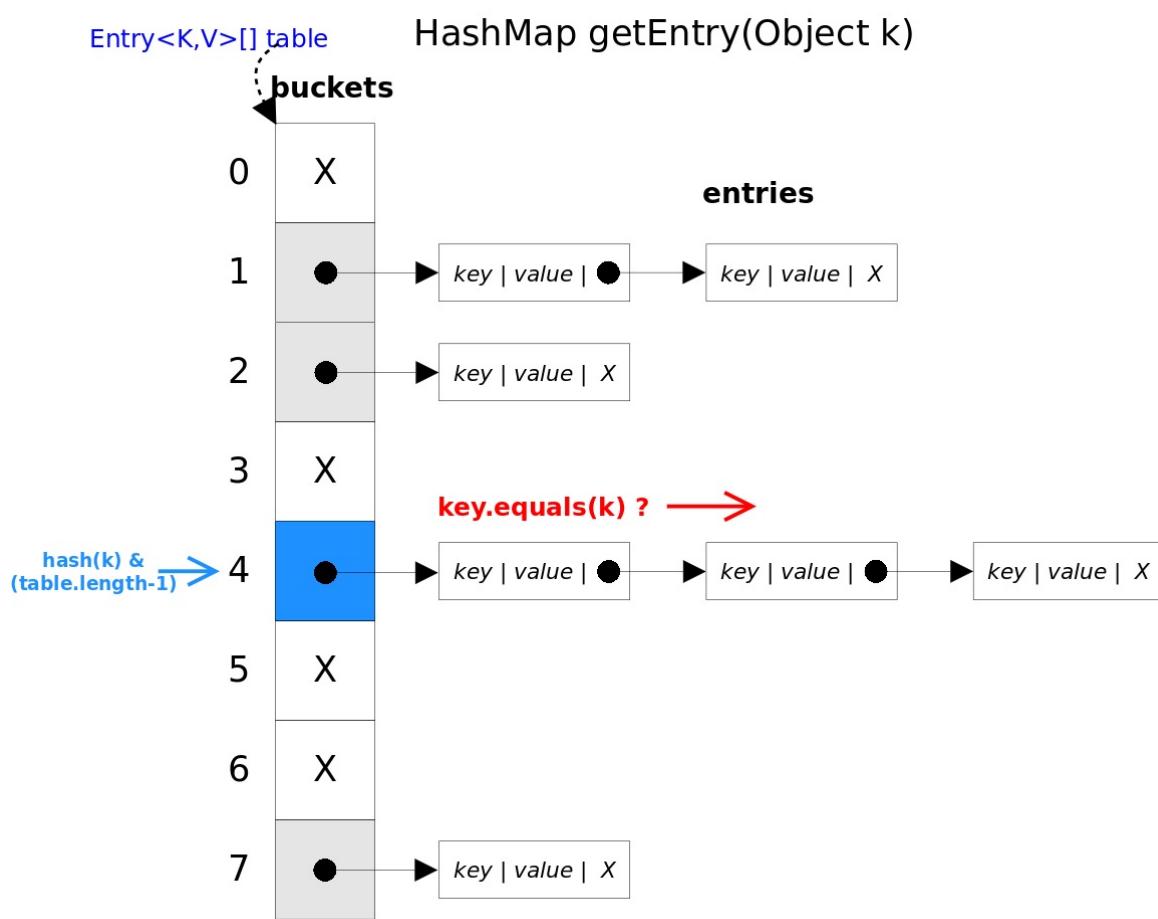
将对象放入到 `HashMap` 或 `HashSet` 中时，有两个方法需要特别关

心：`hashCode()` 和 `equals()`。`hashCode()` 方法决定了对象会被放到哪个 `bucket` 里，当多个对象的哈希值冲突时，`equals()` 方法决定了这些对象是否是“同一个对象”。所以，如果要将自定义的对象放入到 `HashMap` 或 `HashSet` 中，需要 `@Override hashCode()` 和 `equals()` 方法。

方法剖析

get()

`get(Object key)` 方法根据指定的 `key` 值返回对应的 `value`，该方法调用了 `getEntry(Object key)` 得到相应的 `entry`，然后返回 `entry.getValue()`。因此 `getEntry()` 是算法的核心。算法思想是首先通过 `hash()` 函数得到对应 `bucket` 的下标，然后依次遍历冲突链表，通过 `key.equals(k)` 方法来判断是否是要找的那个 `entry`。

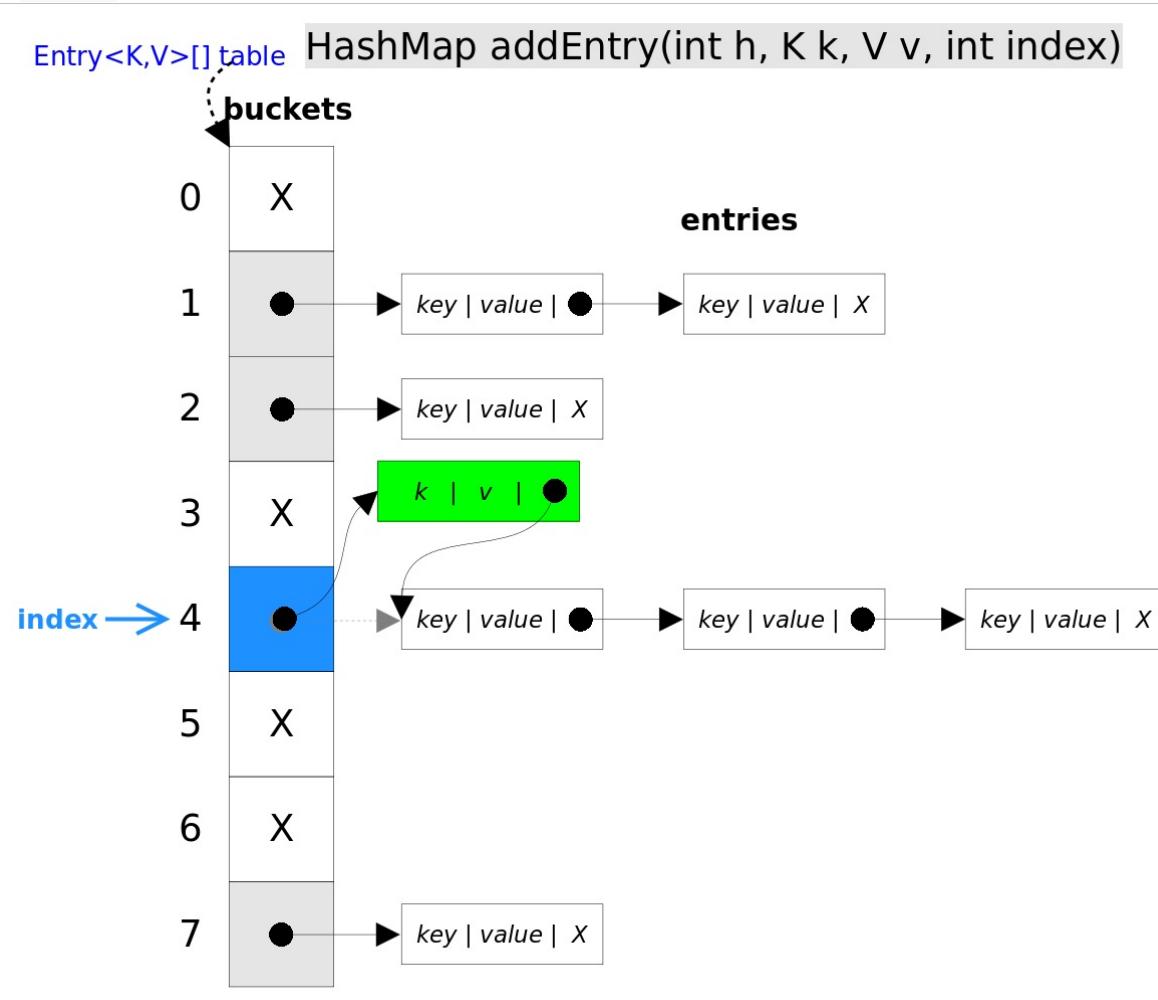


上图中 `hash(k)&(table.length-1)` 等价于 `hash(k)%table.length`，原因是`HashMap`要求 `table.length` 必须是2的指数，因此 `table.length-1` 就是二进制低位全是1，跟 `hash(k)` 相与会将哈希值的高位全抹掉，剩下的就是余数了。

```
//getEntry()方法
final Entry<K,V> getEntry(Object key) {
    .....
    int hash = (key == null) ? 0 : hash(key);
    for (Entry<K,V> e = table[hash&(table.length-1)];//得到冲突链表
        e != null; e = e.next) {//依次遍历冲突链表中的每个entry
        Object k;
        //依据equals()方法判断是否相等
        if (e.hash == hash &&
            ((k = e.key) == key || (key != null && key.equals(k))))
            return e;
    }
    return null;
}
```

put()

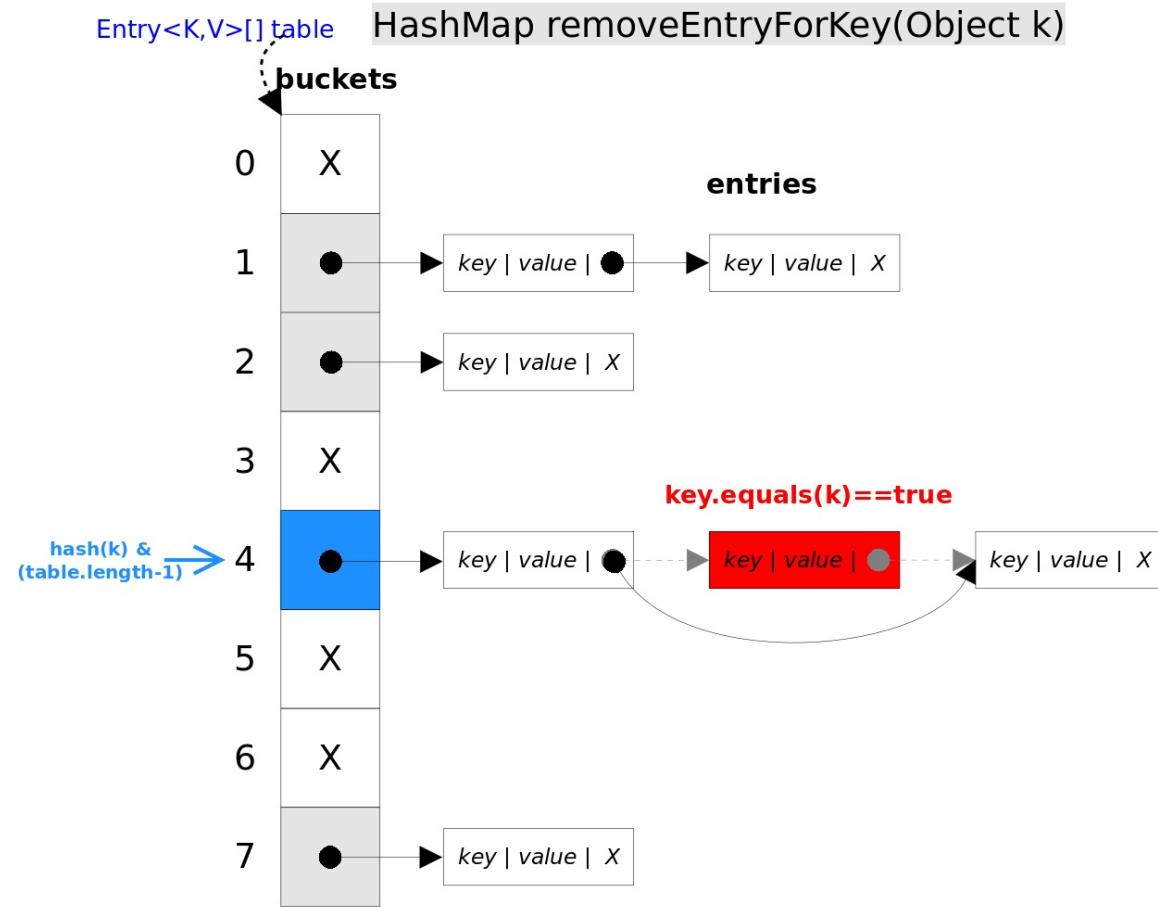
`put(K key, V value)` 方法是将指定的 `key, value` 对添加到 `map` 里。该方法首先会对 `map` 做一次查找，看是否包含该元组，如果已经包含则直接返回，查找过程类似于 `getEntry()` 方法；如果没有找到，则会通过 `addEntry(int hash, K key, V value, int bucketIndex)` 方法插入新的 entry，插入方式为头插法。



```
//addEntry()
void addEntry(int hash, K key, V value, int bucketIndex) {
    if ((size >= threshold) && (null != table[bucketIndex])) {
        resize(2 * table.length); //自动扩容，并重新哈希
        hash = (null != key) ? hash(key) : 0;
        bucketIndex = hash & (table.length-1); //hash%table.length
    }
    //在冲突链表头部插入新的entry
    Entry<K,V> e = table[bucketIndex];
    table[bucketIndex] = new Entry<>(hash, key, value, e);
    size++;
}
```

`remove()`

`remove(Object key)` 的作用是删除 `key` 值对应的 `entry`，该方法的具体逻辑是在 `removeEntryForKey(Object key)` 里实现的。`removeEntryForKey()` 方法会首先找到 `key` 值对应的 `entry`，然后删除该 `entry`（修改链表的相应引用）。查找过程跟 `getEntry()` 过程类似。



```
//removeEntryForKey()
final Entry<K,V> removeEntryForKey(Object key) {
    .....
    int hash = (key == null) ? 0 : hash(key);
    int i = indexFor(hash, table.length); //hash&(table.length-1)
    Entry<K,V> prev = table[i]; //得到冲突链表
    Entry<K,V> e = prev;
    while (e != null) { //遍历冲突链表
        Entry<K,V> next = e.next;
        Object k;
        if (e.hash == hash &&
            ((k = e.key) == key || (key != null && key.equals(k)))) { //找到要删除的entry
try {
            modCount++; size--;
            if (prev == e) table[i] = next; //删除的是冲突链表的第一个entry
            else prev.next = next;
            return e;
        }
        prev = e; e = next;
    }
}
```

```
    return e;
}
```

HashSet

前面已经说过`HashSet`是对`HashMap`的简单包装，对`HashSet`的函数调用都会转换成合适的`HashMap`方法，因此`HashSet`的实现非常简单，只有不到300行代码。这里不再赘述。

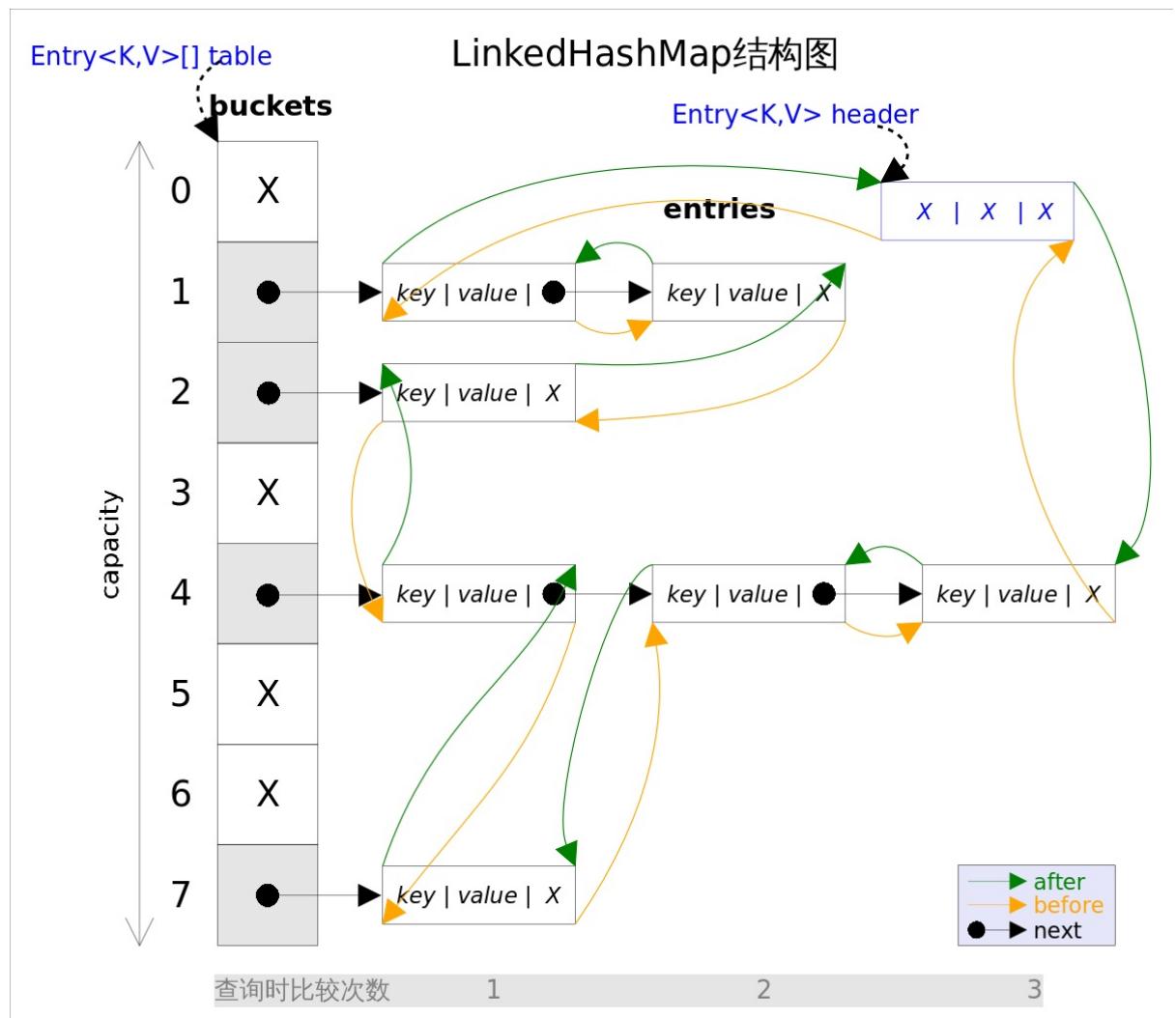
```
//HashSet是对HashMap的简单包装
public class HashSet<E>
{
    .....
    private transient HashMap<E, Object> map;//HashSet里面有一个HashMap
    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();
    public HashSet() {
        map = new HashMap<>();
    }
    .....
    public boolean add(E e) { //简单的方法转换
        return map.put(e, PRESENT)==null;
    }
    .....
}
```

LinkedHashSet and LinkedHashMap

总体介绍

如果你已看过前面关于 *HashSet* 和 *HashMap*, 以及 *TreeSet* 和 *TreeMap* 的讲解, 一定能够想到本文将要讲解的 *LinkedHashSet* 和 *LinkedHashMap* 其实也是一回事。 *LinkedHashSet* 和 *LinkedHashMap* 在 Java 里也有着相同的实现, 前者仅仅是对后者做了一层包装, 也就是说 *LinkedHashSet* 里面有一个 *LinkedHashMap* (适配器模式)。因此本文将重点分析 *LinkedHashMap*。

LinkedHashMap 实现了 *Map* 接口, 即允许放入 *key* 为 *null* 的元素, 也允许插入 *value* 为 *null* 的元素。从名字上可以看出该容器是 *linked list* 和 *HashMap* 的混合体, 也就是说它同时满足 *HashMap* 和 *linked list* 的某些特性。可将 *LinkedHashMap* 看作采用 *linked list* 增强的 *HashMap*。



事实上 *LinkedHashMap* 是 *HashMap* 的直接子类, 二者唯一的区别是 *LinkedHashMap* 在 *HashMap* 的基础上, 采用双向链表 (doubly-linked list) 的形式将所有 entry 连接起来, 这样是为保证元素的迭代顺序跟插入顺序相同。上图给出了 *LinkedHashMap* 的结构图, 主体部分跟 *HashMap* 完全一样, 多了一个 *header* 指向双向链表的头部 (是一个哑元), 该双向链表的迭代顺序就是 entry 的插入顺序。

除了可以保证迭代顺序，这种结构还有一个好处：迭代 `LinkedHashMap` 时不需要像 `HashMap` 那样遍历整个 `table`，而只需要直接遍历 `header` 指向的双向链表即可，也就是说 `LinkedHashMap` 的迭代时间就只跟 `entry` 的个数相关，而跟 `table` 的大小无关。

有两个参数可以影响 `LinkedHashMap` 的性能：初始容量（initial capacity）和负载系数（load factor）。初始容量指定了初始 `table` 的大小，负载系数用来指定自动扩容的临界值。当 `entry` 的数量超过 `capacity * load_factor` 时，容器将自动扩容并重新哈希。对于插入元素较多的场景，将初始容量设大可以减少重新哈希的次数。

将对象放入到 `LinkedHashMap` 或 `LinkedHashSet` 中时，有两个方法需要特别关心：`hashCode()` 和 `equals()`。`hashCode()` 方法决定了对象会被放到哪个 `bucket` 里，当多个对象的哈希值冲突时，`equals()` 方法决定了这些对象是否是“同一个对象”。所以，如果要将自定义的对象放入到 `LinkedHashMap` 或 `LinkedHashSet` 中，需要 `@Override hashCode()` 和 `equals()` 方法。

通过如下方式可以得到一个跟源 `Map` 迭代顺序一样的 `LinkedHashMap`：

```
void foo(Map m) {
    Map copy = new LinkedHashMap(m);
    ...
}
```

出于性能原因，`LinkedHashMap` 是非同步的（not synchronized），如果需要在多线程环境使用，需要程序员手动同步；或者通过如下方式将 `LinkedHashMap` 包装成（wrapped）同步的：

```
Map m = Collections.synchronizedMap(new LinkedHashMap(...));
```

方法剖析

get()

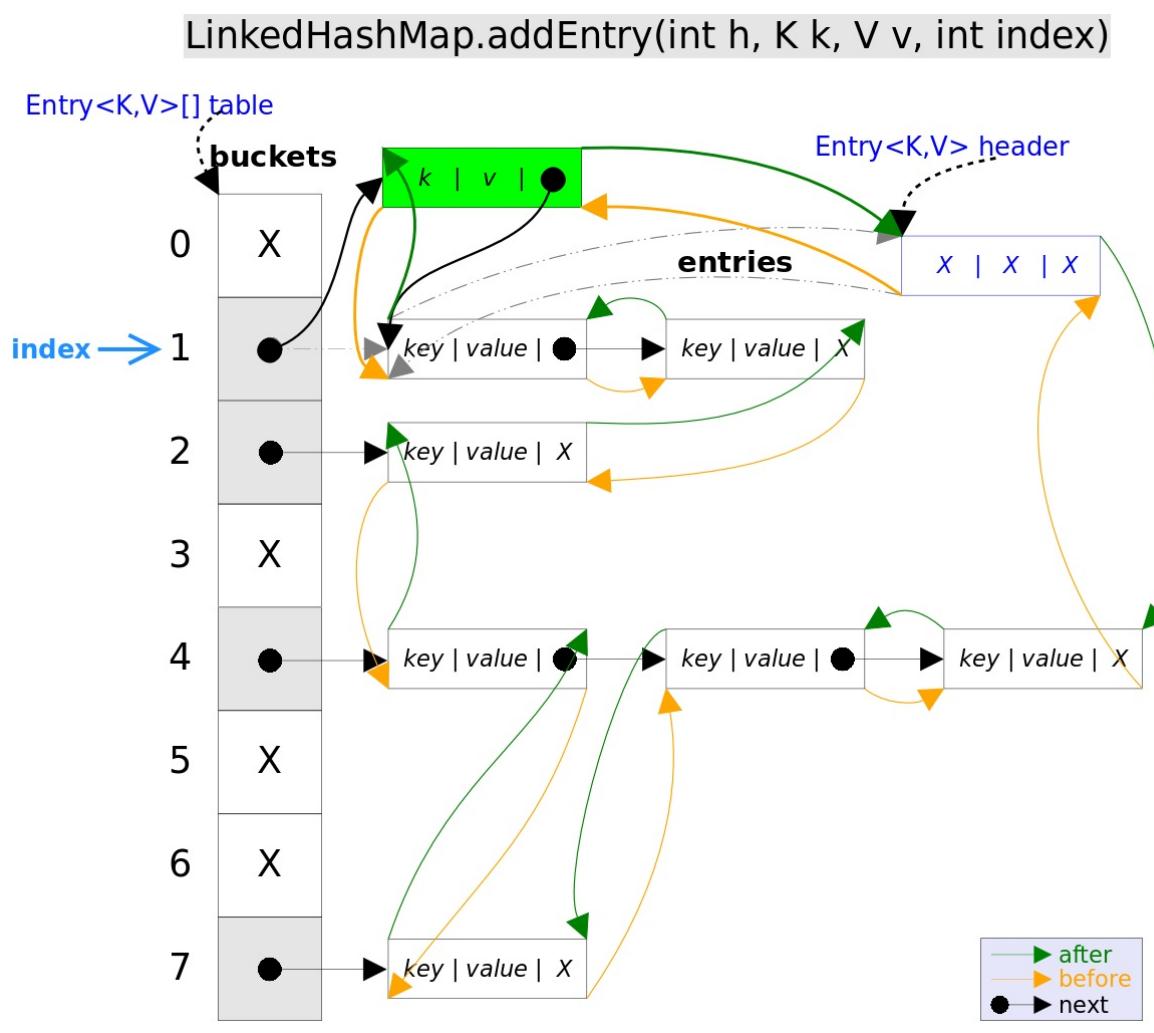
`get(Object key)` 方法根据指定的 `key` 值返回对应的 `value`。该方法跟 `HashMap.get()` 方法的流程几乎完全一样，读者可自行[参考前文](#)，这里不再赘述。

put()

`put(K key, V value)` 方法是将指定的 `key, value` 对添加到 `map` 里。该方法首先会对 `map` 做一次查找，看是否包含该元组，如果已经包含则直接返回，查找过程类似于 `get()` 方法；如果没有找到，则会通过 `addEntry(int hash, K key, V value, int bucketIndex)` 方法插入新的 `entry`。

注意，这里的插入有两重含义：

1. 从 `table` 的角度看，新的 `entry` 需要插入到对应的 `bucket` 里，当有哈希冲突时，采用头插法将新的 `entry` 插入到冲突链表的头部。
2. 从 `header` 的角度看，新的 `entry` 需要插入到双向链表的尾部。



`addEntry()` 代码如下：

```
// LinkedHashMap.addEntry()
void addEntry(int hash, K key, V value, int bucketIndex) {
    if ((size >= threshold) && (null != table[bucketIndex])) {
        resize(2 * table.length); // 自动扩容，并重新哈希
        hash = (null != key) ? hash(key) : 0;
        bucketIndex = hash & (table.length-1); // hash%table.length
    }
    // 1. 在冲突链表头部插入新的entry
    HashMap.Entry<K,V> old = table[bucketIndex];
    Entry<K,V> e = new Entry<>(hash, key, value, old);
    table[bucketIndex] = e;
    // 2. 在双向链表的尾部插入新的entry
    e.addBefore(header);
    size++;
}
```

上述代码中用到了 `addBefore()` 方法将新 `entry e` 插入到双向链表头引用 `header` 的前面，这样 `e` 就成为双向链表中的最后一个元素。`addBefore()` 的代码如下：

```
// LinkedHashMap.Entry.addBefore(), 将this插入到existingEntry的前面
private void addBefore(Entry<K,V> existingEntry) {
    after = existingEntry;
    before = existingEntry.before;
    before.after = this;
    after.before = this;
}
```

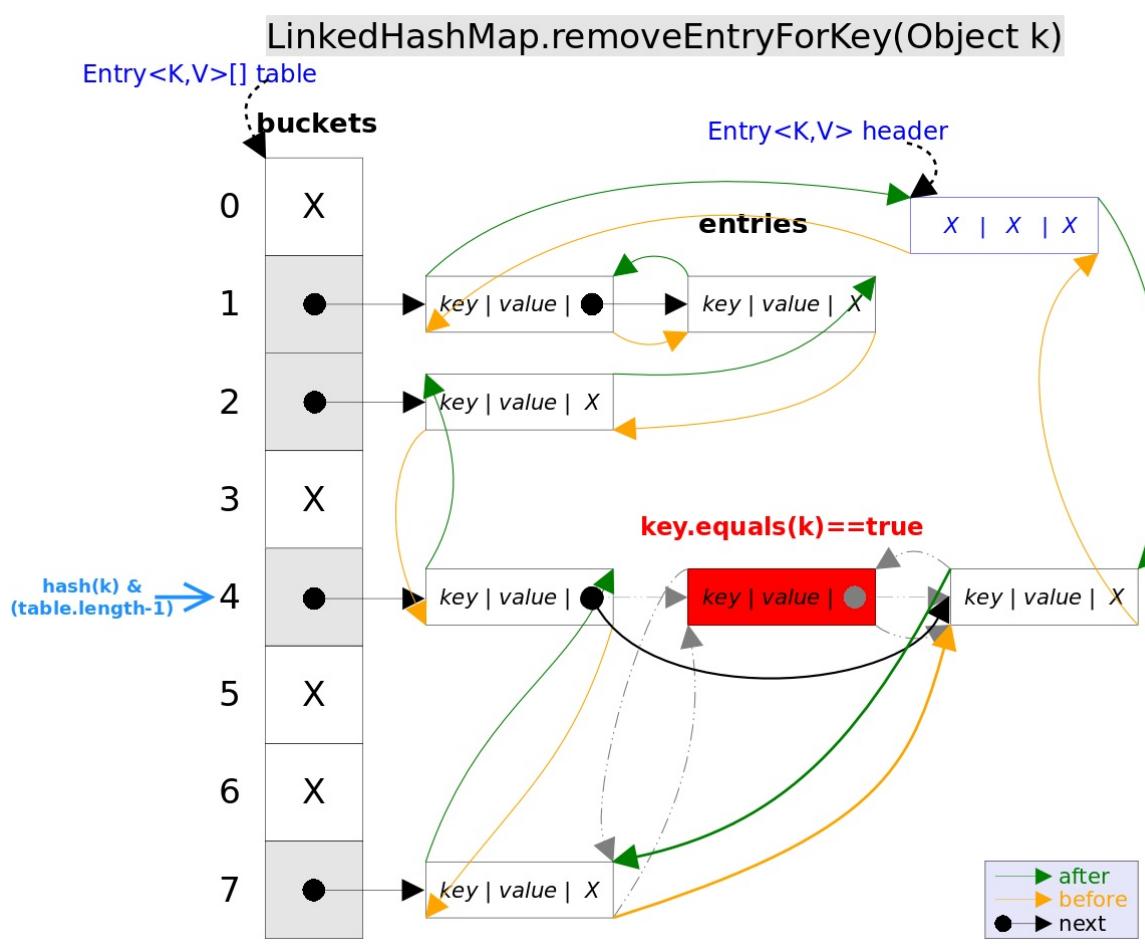
上述代码只是简单修改相关 entry 的引用而已。

remove()

`remove(Object key)` 的作用是删除 `key` 值对应的 `entry`，该方法的具体逻辑是在 `removeEntryForKey(Object key)` 里实现的。`removeEntryForKey()` 方法会首先找到 `key` 值对应的 `entry`，然后删除该 `entry`（修改链表的相应引用）。查找过程跟 `get()` 方法类似。

注意，这里的删除也有两重含义：

1. 从 `table` 的角度看，需要将该 `entry` 从对应的 `bucket` 里删除，如果对应的冲突链表不空，需要修改冲突链表的相应引用。
2. 从 `header` 的角度来看，需要将该 `entry` 从双向链表中删除，同时修改链表中前面以及后面元素的相应引用。



`removeEntryForKey()` 对应的代码如下：

```
// LinkedHashMap.removeEntryForKey(), 删除key值对应的entry
final Entry<K,V> removeEntryForKey(Object key) {
    .....
    int hash = (key == null) ? 0 : hash(key);
    int i = indexFor(hash, table.length); // hash&(table.length-1)
    Entry<K,V> prev = table[i]; // 得到冲突链表
    Entry<K,V> e = prev;
    while (e != null) { // 遍历冲突链表
        Entry<K,V> next = e.next;
        Object k;
        if (e.hash == hash &&
            ((k = e.key) == key || (key != null && key.equals(k)))) { // 找到要删除的entry
            modCount++; size--;
            // 1. 将e从对应bucket的冲突链表中删除
            if (prev == e) table[i] = next;
            else prev.next = next;
            // 2. 将e从双向链表中删除
            e.before.after = e.after;
            e.after.before = e.before;
            return e;
        }
    }
}
```

```

        }
        prev = e; e = next;
    }
    return e;
}

```

LinkedHashSet

前面已经说过`LinkedHashSet`是对`LinkedHashMap`的简单包装，对`LinkedHashSet`的函数调用都会转换成合适的`LinkedHashMap`方法，因此`LinkedHashSet`的实现非常简单，这里不再赘述。

```

public class LinkedHashSet<E>
    extends HashSet<E>
    implements Set<E>, Cloneable, java.io.Serializable {
    .....
    // LinkedHashSet里面有一个LinkedHashMap
    public LinkedHashSet(int initialCapacity, float loadFactor) {
        map = new LinkedHashMap<E>(initialCapacity, loadFactor);
    }
    .....
    public boolean add(E e) { //简单的方法转换
        return map.put(e, PRESENT)==null;
    }
    .....
}

```

LinkedHashMap经典用法

`LinkedHashMap`除了可以保证迭代顺序外，还有一个非常有用的用法：可以轻松实现一个采用了FIFO替换策略的缓存。具体说来，`LinkedHashMap`有一个子类方法 `protected boolean removeEldestEntry(Map.Entry<K, V> eldest)`，该方法的作用是告诉Map是否要删除“最老”的Entry，所谓最老就是当前Map中最早插入的Entry，如果该方法返回 `true`，最老的那个元素就会被删除。在每次插入新元素之后`LinkedHashMap`会自动询问`removeEldestEntry()`是否要删除最老的元素。这样只需要在子类中重载该方法，当元素个数超过一定数量时让`removeEldestEntry()`返回`true`，就能够实现一个固定大小的FIFO策略的缓存。示例代码如下：

```

/** 一个固定大小的FIFO替换策略的缓存 */
class FIFOCache<K, V> extends LinkedHashMap<K, V>{
    private final int cacheSize;
    public FIFOCache(int cacheSize){
        this.cacheSize = cacheSize;
    }

    // 当Entry个数超过cacheSize时，删除最老的Entry
    @Override

```

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {  
    return size() > cacheSize;  
}  
}
```

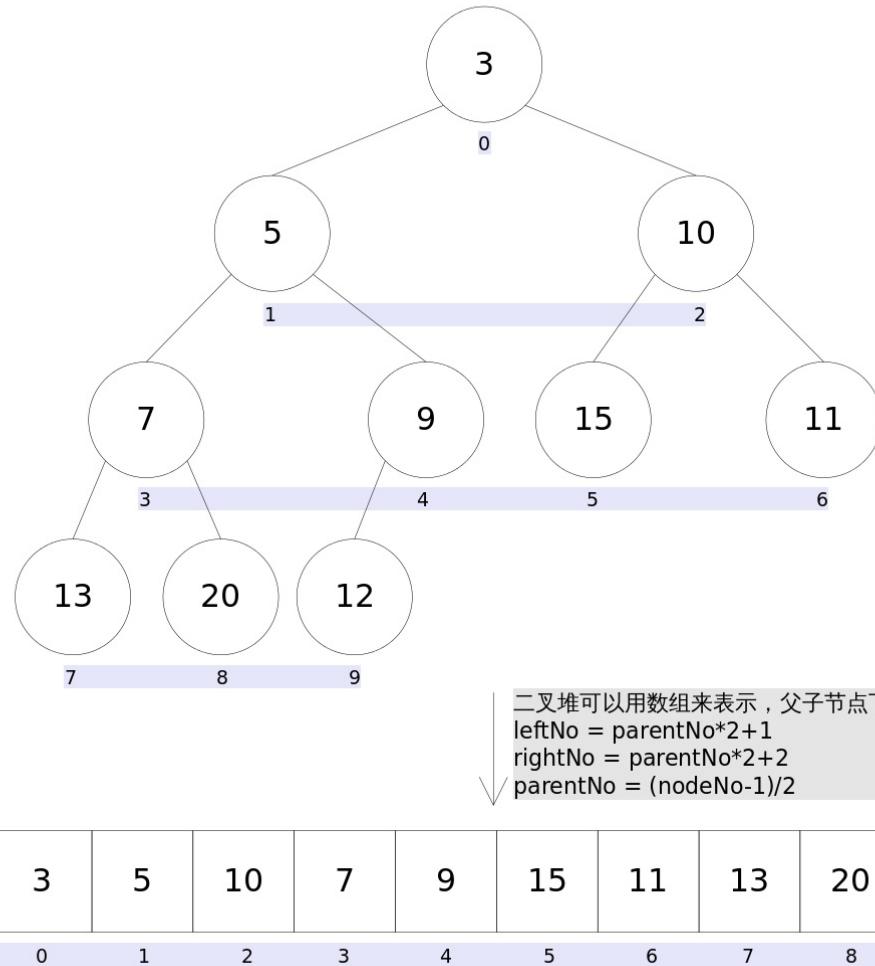
PriorityQueue

总体介绍

前面以Java `ArrayDeque`为例讲解了 `Stack` 和 `Queue`，其实还有一种特殊的队列叫做 `PriorityQueue`，即优先队列。优先队列的作用是能保证每次取出的元素都是队列中权值最小的（Java的优先队列每次取最小元素，C++的优先队列每次取最大元素）。这里牵涉到了大小关系，元素大小的评判可以通过元素本身的自然顺序 (*natural ordering*)，也可以通过构造时传入的比较器 (*Comparator*, 类似于C++的仿函数)。

Java中 `PriorityQueue` 实现了 `Queue` 接口，不允许放入 `null` 元素；其通过堆实现，具体说是通过完全二叉树 (*complete binary tree*) 实现的小顶堆（任意一个非叶子节点的权值，都不大于其左右子节点的权值），也就意味着可以通过数组来作为 `PriorityQueue` 的底层实现。

PriorityQueue通过用数组表示的小顶堆实现



上图中我们给每个元素按照层序遍历的方式进行了编号，如果你足够细心，会发现父节点和子节点的编号是有联系的，更确切的说父子节点的编号之间有如下关系：

```

leftNo = parentNo*2+1
rightNo = parentNo*2+2
parentNo = (nodeNo-1)/2

```

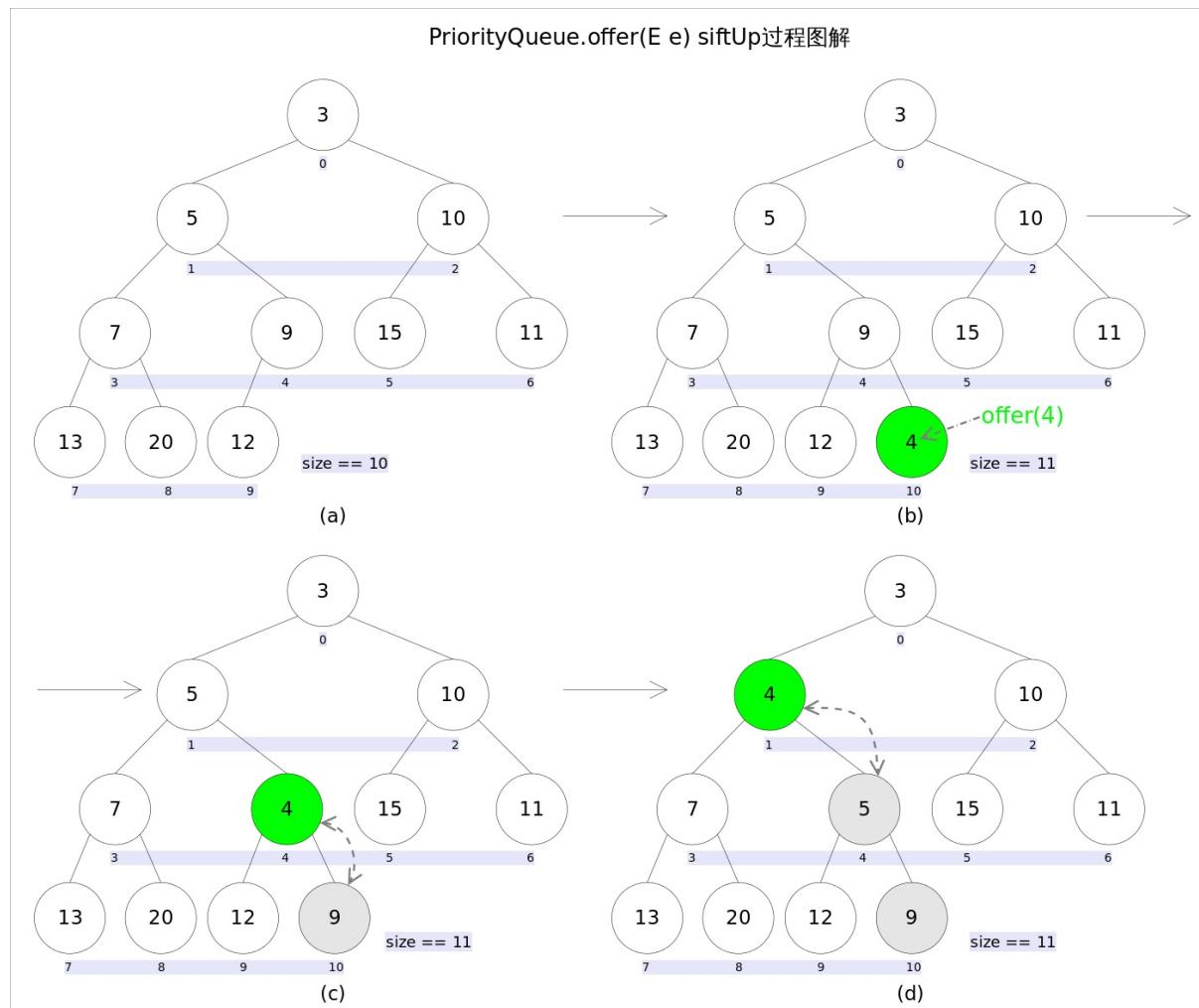
通过上述三个公式，可以轻易计算出某个节点的父节点以及子节点的下标。这也就是为什么可以直接用数组来存储堆的原因。

PriorityQueue 的 `peek()` 和 `element` 操作是常数时间，`add()`，`offer()`，无参数的 `remove()` 以及 `poll()` 方法的时间复杂度都是 $\log(N)$ 。

方法剖析

add()和offer()

`add(E e)` 和 `offer(E e)` 的语义相同，都是向优先队列中插入元素，只是 `Queue` 接口规定二者对插入失败时的处理不同，前者在插入失败时抛出异常，后则则会返回 `false`。对于 *PriorityQueue* 这两个方法其实没什么差别。



新加入的元素可能会破坏小顶堆的性质，因此需要进行必要的调整。

```
//offer(E e)
public boolean offer(E e) {
    if (e == null)//不允许放入null元素
        throw new NullPointerException();
    modCount++;
    int i = size;
    if (i >= queue.length)
        grow(i + 1); //自动扩容
    size = i + 1;
    if (i == 0)//队列原来为空，这是插入的第一个元素
        queue[0] = e;
    else
        siftUp(i, e); //调整
    return true;
}
```

上述代码中，扩容函数 `grow()` 类似于 `ArrayList` 里的 `grow()` 函数，就是再申请一个更大的数组，并将原数组的元素复制过去，这里不再赘述。需要注意的是 `siftUp(int k, E x)` 方法，该方法用于插入元素 `x` 并维持堆的特性。

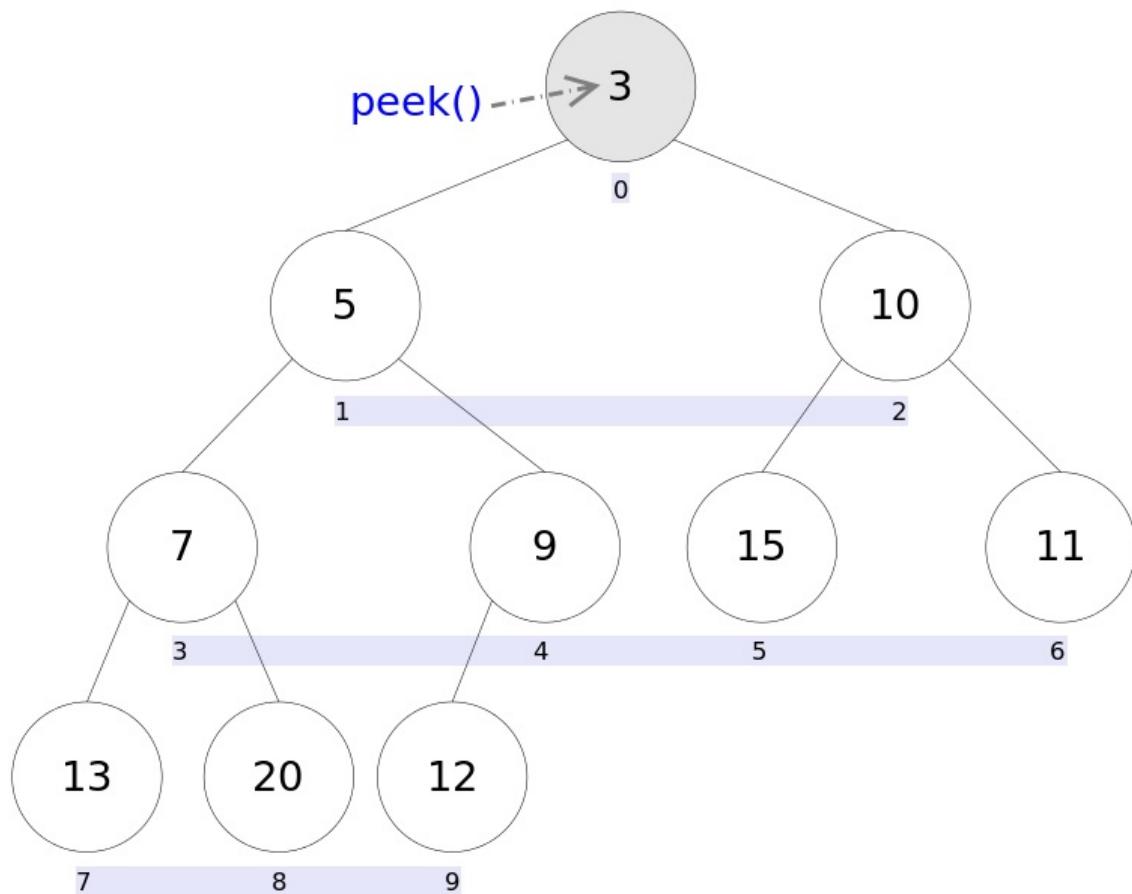
```
//siftUp()
private void siftUp(int k, E x) {
    while (k > 0) {
        int parent = (k - 1) >>> 1; //parentNo = (nodeNo-1)/2
        Object e = queue[parent];
        if (comparator.compare(x, (E) e) >= 0) //调用比较器的比较方法
            break;
        queue[k] = e;
        k = parent;
    }
    queue[k] = x;
}
```

新加入的元素 `x` 可能会破坏小顶堆的性质，因此需要进行调整。调整的过程为：从 `k` 指定的位置开始，将 `x` 逐层与当前点的 `parent` 进行比较并交换，直到满足 `x >= queue[parent]` 为止。注意这里的比较可以是元素的自然顺序，也可以是依靠比较器的顺序。

element()和peek()

`element()` 和 `peek()` 的语义完全相同，都是获取但不删除队首元素，也就是队列中权值最小的那个元素，二者唯一的区别是当方法失败时前者抛出异常，后者返回 `null`。根据小顶堆的性质，堆顶那个元素就是全局最小的那个；由于堆用数组表示，根据下标关系，`0` 下标处的那个元素既是堆顶元素。所以直接返回数组 `0` 下标处的那个元素即可。

PriorityQueue.peek()



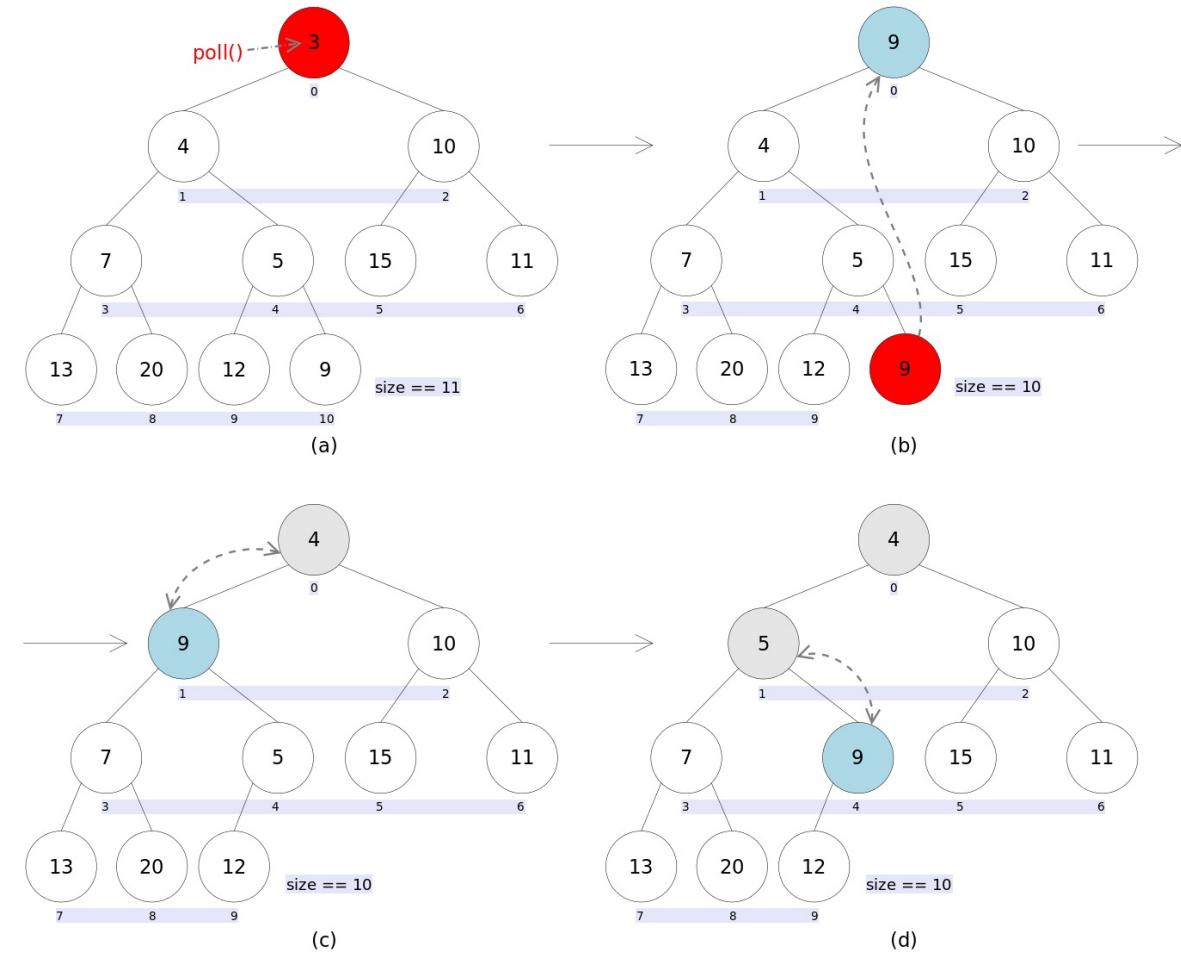
代码也就非常简洁：

```
//peek()
public E peek() {
    if (size == 0)
        return null;
    return (E) queue[0]; //0下标处的那个元素就是最小的那个
}
```

remove()和poll()

`remove()` 和 `poll()` 方法的语义也完全相同，都是获取并删除队首元素，区别是当方法失败时前者抛出异常，后者返回 `null`。由于删除操作会改变队列的结构，为维护小顶堆的性质，需要进行必要的调整。

PriorityQueue.poll() siftDown过程图解



代码如下：

```

public E poll() {
    if (size == 0)
        return null;
    int s = --size;
    modCount++;
    E result = (E) queue[0];//0下标处的那个元素就是最小的那个
    E x = (E) queue[s];
    queue[s] = null;
    if (s != 0)
        siftDown(0, x);//调整
    return result;
}

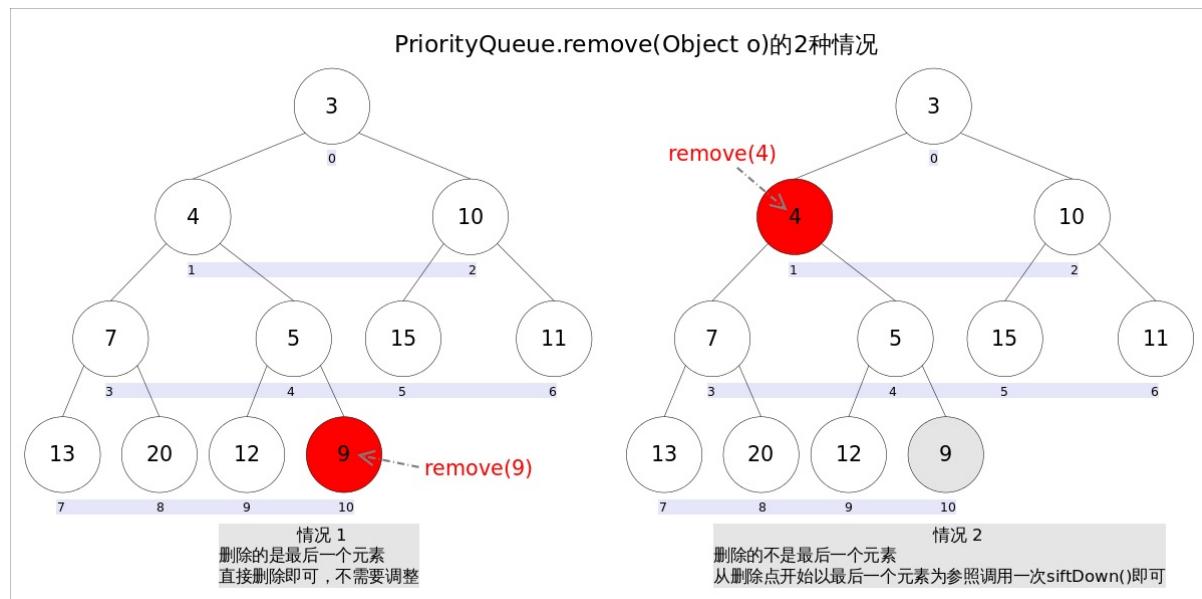
```

上述代码首先记录 0 下标处的元素，并用最后一个元素替换 0 下标位置的元素，之后调用 `siftDown()` 方法对堆进行调整，最后返回原来 0 下标处的那个元素（也就是最小的那个元素）。重点是 `siftDown(int k, E x)` 方法，该方法的作用是从 `k` 指定的位置开始，将 `x` 逐层向下与当前点的左右孩子中较小的那个交换，直到 `x` 小于或等于左右孩子中的任何一个为止。

```
//siftDown()
private void siftDown(int k, E x) {
    int half = size >>> 1;
    while (k < half) {
        //首先找到左右孩子中较小的那个，记录到c里，并用child记录其下标
        int child = (k << 1) + 1;//leftNo = parentNo*2+1
        Object c = queue[child];
        int right = child + 1;
        if (right < size &&
            comparator.compare((E) c, (E) queue[right]) > 0)
            c = queue[child = right];
        if (comparator.compare(x, (E) c) <= 0)
            break;
        queue[k] = c;//然后用c取代原来的值
        k = child;
    }
    queue[k] = x;
}
```

remove(Object o)

`remove(Object o)` 方法用于删除队列中跟 `o` 相等的某一个元素（如果有多个相等，只删除一个），该方法不是`Queue`接口内的方法，而是`Collection`接口的方法。由于删除操作会改变队列结构，所以要进行调整；又由于删除元素的位置可能是任意的，所以调整过程比其它函数稍加繁琐。具体来说，`remove(Object o)` 可以分为2种情况：1. 删除的是最后一个元素。直接删除即可，不需要调整。2. 删除的不是最后一个元素，从删除点开始以最后一个元素为参照调用一次 `siftDown()` 即可。此处不再赘述。



具体代码如下：

```
//remove(Object o)
```

```
public boolean remove(Object o) {  
    //通过遍历数组的方式找到第一个满足o.equals(queue[i])元素的下标  
    int i = indexOf(o);  
    if (i == -1)  
        return false;  
    int s = --size;  
    if (s == i) //情况1  
        queue[i] = null;  
    else {  
        E moved = (E) queue[s];  
        queue[s] = null;  
        siftDown(i, moved); //情况2  
        .....  
    }  
    return true;  
}
```

WeakHashMap

总体介绍

在Java集合框架系列文章的最后，笔者打算介绍一个特殊的成员：*WeakHashMap*，从名字可以看出它是某种 *Map*。它的特殊之处在于 *WeakHashMap* 里的 *entry* 可能会被GC自动删除，即使程序员没有调用 *remove()* 或者 *clear()* 方法。

更直观的说，当使用 *WeakHashMap* 时，即使没有显示的添加或删除任何元素，也可能发生如下情况：

- 调用两次 *size()* 方法返回不同的值；
- 两次调用 *isEmpty()* 方法，第一次返回 *false*，第二次返回 *true*；
- 两次调用 *containsKey()* 方法，第一次返回 *true*，第二次返回 *false*，尽管两次使用的是同一个 *key*；
- 两次调用 *get()* 方法，第一次返回一个 *value*，第二次返回 *null*，尽管两次使用的是同一个对象。

遇到这么奇葩的现象，你是不是觉得使用者一定会疯掉？其实不然，*WeakHashMap* 的这个特点特别适用于需要缓存的场景。在缓存场景下，由于内存是有限的，不能缓存所有对象；对象缓存命中可以提高系统效率，但缓存MISS也不会造成错误，因为可以通过计算重新得到。

要明白 *WeakHashMap* 的工作原理，还需要引入一个概念：弱引用（*WeakReference*）。我们都知道 Java中内存是通过GC自动管理的，GC会在程序运行过程中自动判断哪些对象是可以被回收的，并在合适的时机进行内存释放。GC判断某个对象是否可被回收的依据是，是否有有效的引用指向该对象。如果没有有效引用指向该对象（基本意味着不存在访问该对象的方式），那么该对象就是可回收的。这里的“有效引用”并不包括弱引用。也就是说，虽然弱引用可以用来访问对象，但进行垃圾回收时弱引用并不会被考虑在内，仅有弱引用指向的对象仍然会被GC回收。

WeakHashMap 内部是通过弱引用管理 *entry* 的，弱引用的特性对应到 *WeakHashMap* 上意味着什么呢？将一对 *key, value* 放入到 *WeakHashMap* 里并不能避免该 *key* 值被GC回收，除非在 *WeakHashMap* 之外还有对该 *key* 的强引用。

关于强引用，弱引用等概念以后再具体讲解，这里只需要知道Java中引用也是分种类的，并且不同种类的引用对GC的影响不同就够了。

具体实现

*WeakHashMap*的存储结构类似于*HashMap*，读者可自行[参考前文](#)，这里不再赘述。

关于强弱引用的管理方式，博主将会另开专题单独讲解。

Weak HashSet?

如果你看过前几篇关于 *Map* 和 *Set* 的讲解，一定会问：既然有 *WeakHashMap*，是否有 *WeakHashSet* 呢？答案是没有：(。不过 Java *Collections* 工具类给出了解决方案，`Collections.newSetFromMap(Map<E, Boolean> map)` 方法可以将任何 *Map* 包装成一个 *Set*。通过如下方式可以快速得到一个 *Weak HashSet*：

```
// 将WeakHashMap包装成一个Set
Set<Object> weakHashSet = Collections.newSetFromMap(
    new WeakHashMap<Object, Boolean>());
```

不出你所料，`newSetFromMap()` 方法只是对传入的 *Map* 做了简单包装：

```
// Collections.newSetFromMap()用于将任何Map包装成一个Set
public static <E> Set<E> newSetFromMap(Map<E, Boolean> map) {
    return new SetFromMap<E>(map);
}

private static class SetFromMap<E> extends AbstractSet<E>
    implements Set<E>, Serializable
{
    private final Map<E, Boolean> m; // The backing map
    private transient Set<E> s; // Its keySet
    SetFromMap(Map<E, Boolean> map) {
        if (!map.isEmpty())
            throw new IllegalArgumentException("Map is non-empty");
        m = map;
        s = map.keySet();
    }
    public void clear() { m.clear(); }
    public int size() { return m.size(); }
    public boolean isEmpty() { return m.isEmpty(); }
    public boolean contains(Object o) { return m.containsKey(o); }
    public boolean remove(Object o) { return m.remove(o) != null; }
    public boolean add(E e) { return m.put(e, Boolean.TRUE) == null; }
    public Iterator<E> iterator() { return s.iterator(); }
    public Object[] toArray() { return s.toArray(); }
    public <T> T[] toArray(T[] a) { return s.toArray(a); }
    public String toString() { return s.toString(); }
    public int hashCode() { return s.hashCode(); }
    public boolean equals(Object o) { return o == this || s.equals(o); }
    public boolean containsAll(Collection<?> c) { return s.containsAll(c); }
    public boolean removeAll(Collection<?> c) { return s.removeAll(c); }
    public boolean retainAll(Collection<?> c) { return s.retainAll(c); }
    // addAll is the only inherited implementation
    ...
}
```

结语

至此*Java Collections Framework Internals*系列已经全部讲解完毕，希望这几篇简短的博文能够帮助各位读者对Java容器框架建立基本的理解。通过这里可以返回[本系列文章目录](#)

如果对各位有哪怕些微的帮助，博主将感到非常高兴！如果博文中有任何的纰漏和谬误，欢迎各位博友指正。