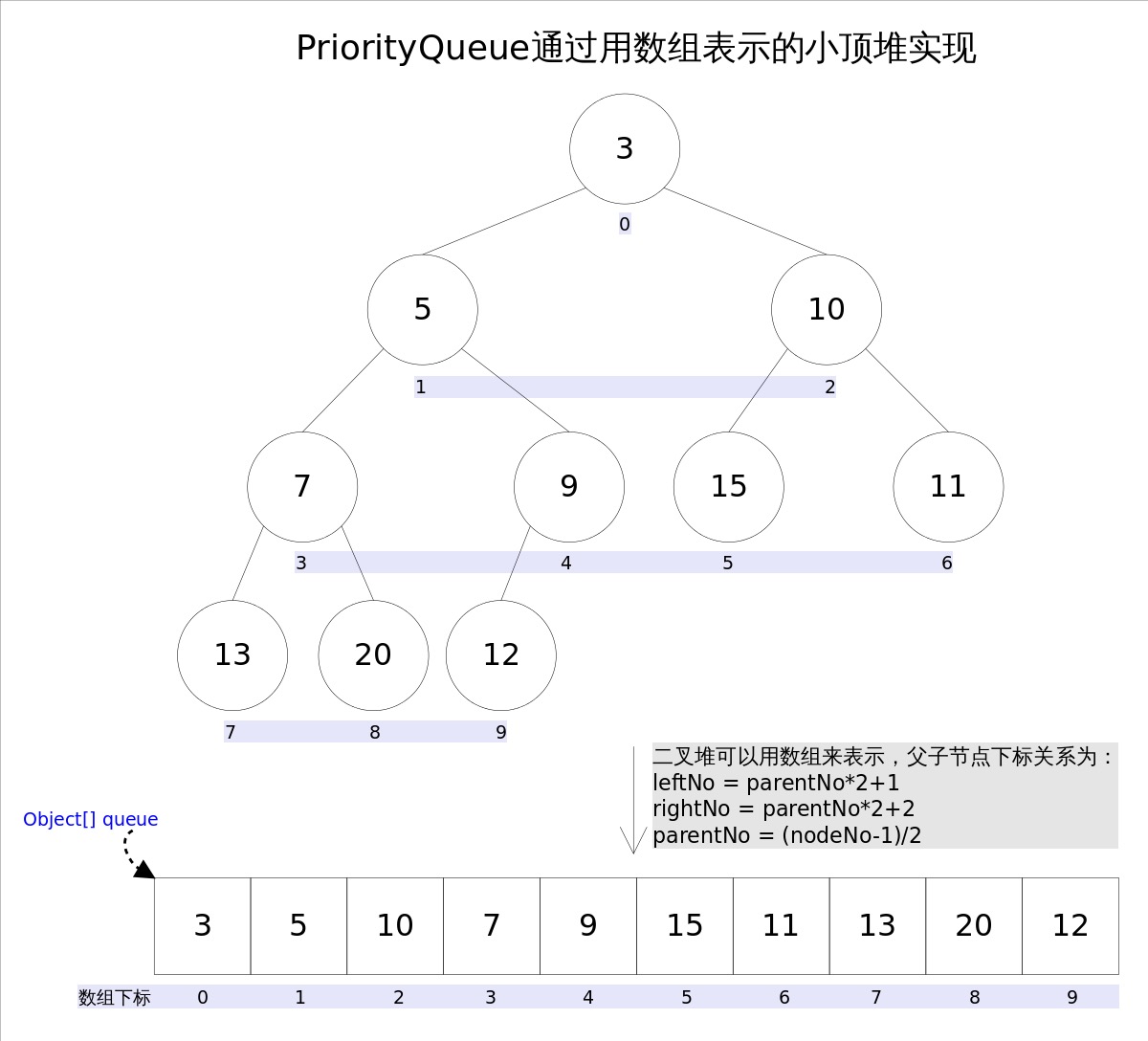
**java集合框架之PriorityQueue 深度解析（一）**

**摘要：**

Java中PriorityQueue通过二叉小顶堆实现，可以用一棵完全二叉树表示。本文从Queue接口函数出发，结合生动的图解，深入浅出地分析PriorityQueue每个操作的具体过程和时间复杂度，将让读者建立对PriorityQueue建立清晰而深入的认识。  
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。

**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）。



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)。

**API**

**1.内部属性**

*//默认用于存储节点信息的数组的大小*

private static final int DEFAULT\_INITIAL\_CAPACITY = 11;

*//用于存储节点信息的数组*

transient Object[] queue;

*//数组中实际存放元素的个数*

private int size = 0;

*//Comparator比较器*

private final Comparator<? super E> comparator;

*//用于记录修改次数的变量*

transient int modCount = 0;

我们知道，堆这种数据结构主要分类有两种，大根堆和小根堆。而我们每次的调整结构都是不断按照某种规则比较两个元素的值大小，然后调整结构，这里就需要用到我们的比较器。所以构建一个PriorityQueue实例主要还是初始化数组容量和comparator比较器，而在PriorityQueue主要有以下几种构造器：

**2.构造函数**

*/\*\**

*\* Creates a {@code PriorityQueue} with the default initial*

*\* capacity (11) that orders its elements according to their*

*\* {@linkplain Comparable natural ordering}.*

*\*/*

public PriorityQueue() {

this(DEFAULT\_INITIAL\_CAPACITY, null);

}

*/\*\**

*\* Creates a {@code PriorityQueue} with the specified initial*

*\* capacity that orders its elements according to their*

*\* {@linkplain Comparable natural ordering}.*

*\**

*\* @param initialCapacity the initial capacity for this priority queue*

*\* @throws IllegalArgumentException if {@code initialCapacity} is less*

*\* than 1*

*\*/*

public PriorityQueue(int initialCapacity) {

this(initialCapacity, null);

}

*/\*\**

*\* Creates a {@code PriorityQueue} with the default initial capacity and*

*\* whose elements are ordered according to the specified comparator.*

*\**

*\* @param comparator the comparator that will be used to order this*

*\* priority queue. If {@code null}, the {@linkplain Comparable*

*\* natural ordering} of the elements will be used.*

*\* @since 1.8*

*\*/*

public PriorityQueue(Comparator<? super E> comparator) {

this(DEFAULT\_INITIAL\_CAPACITY, comparator);

}

*/\*\**

*\* Creates a {@code PriorityQueue} with the specified initial capacity*

*\* that orders its elements according to the specified comparator.*

*\**

*\* @param initialCapacity the initial capacity for this priority queue*

*\* @param comparator the comparator that will be used to order this*

*\* priority queue. If {@code null}, the {@linkplain Comparable*

*\* natural ordering} of the elements will be used.*

*\* @throws IllegalArgumentException if {@code initialCapacity} is*

*\* less than 1*

*\*/*

public PriorityQueue(int initialCapacity,

Comparator<? super E> comparator) {

*// Note: This restriction of at least one is not actually needed,*

*// but continues for 1.5 compatibility*

if (initialCapacity < 1)

throw new IllegalArgumentException();

this.queue = new Object[initialCapacity];

this.comparator = comparator;

}

*/\*\**

*\* Creates a {@code PriorityQueue} containing the elements in the*

*\* specified collection. If the specified collection is an instance of*

*\* a {@link SortedSet} or is another {@code PriorityQueue}, this*

*\* priority queue will be ordered according to the same ordering.*

*\* Otherwise, this priority queue will be ordered according to the*

*\* {@linkplain Comparable natural ordering} of its elements.*

*\**

*\* @param c the collection whose elements are to be placed*

*\* into this priority queue*

*\* @throws ClassCastException if elements of the specified collection*

*\* cannot be compared to one another according to the priority*

*\* queue's ordering*

*\* @throws NullPointerException if the specified collection or any*

*\* of its elements are null*

*\*/*

@SuppressWarnings("unchecked")

public PriorityQueue(Collection<? extends E> c) {

if (c instanceof SortedSet<?>) {

SortedSet<? extends E> ss = (SortedSet<? extends E>) c;

this.comparator = (Comparator<? super E>) ss.comparator();

initElementsFromCollection(ss);

}

else if (c instanceof PriorityQueue<?>) {

PriorityQueue<? extends E> pq = (PriorityQueue<? extends E>) c;

this.comparator = (Comparator<? super E>) pq.comparator();

initFromPriorityQueue(pq);

}

else {

this.comparator = null;

initFromCollection(c);

}

}

*/\*\**

*\* Creates a {@code PriorityQueue} containing the elements in the*

*\* specified priority queue. This priority queue will be*

*\* ordered according to the same ordering as the given priority*

*\* queue.*

*\**

*\* @param c the priority queue whose elements are to be placed*

*\* into this priority queue*

*\* @throws ClassCastException if elements of {@code c} cannot be*

*\* compared to one another according to {@code c}'s*

*\* ordering*

*\* @throws NullPointerException if the specified priority queue or any*

*\* of its elements are null*

*\*/*

@SuppressWarnings("unchecked")

public PriorityQueue(PriorityQueue<? extends E> c) {

this.comparator = (Comparator<? super E>) c.comparator();

initFromPriorityQueue(c);

}

*/\*\**

*\* Creates a {@code PriorityQueue} containing the elements in the*

*\* specified sorted set. This priority queue will be ordered*

*\* according to the same ordering as the given sorted set.*

*\**

*\* @param c the sorted set whose elements are to be placed*

*\* into this priority queue*

*\* @throws ClassCastException if elements of the specified sorted*

*\* set cannot be compared to one another according to the*

*\* sorted set's ordering*

*\* @throws NullPointerException if the specified sorted set or any*

*\* of its elements are null*

*\*/*

@SuppressWarnings("unchecked")

public PriorityQueue(SortedSet<? extends E> c) {

this.comparator = (Comparator<? super E>) c.comparator();

initElementsFromCollection(c);

}

主要构造器有上述前面四种，前三种在内部会调用最后一个构造器。两个参数，一个指定要初始化数组的容量，一个则用于初始化一个比较器。如果没有显式指定他们的值，则对于容量则默认为DEFAULT\_INITIAL\_CAPACITY（11），comparator则为null。下面我们看获取到PriorityQueue实例之后，如何向其中添加和删除节点却一样保持原堆结构不变。

**3.常用功能函数**



**方法剖析**

**boolean add(E e) 和boolean offer(E e)**

*/\*\**

*\* Inserts the specified element into this priority queue.*

*\**

*\* @return {@code true} (as specified by {@link Collection#add})*

*\* @throws ClassCastException if the specified element cannot be*

*\* compared with elements currently in this priority queue*

*\* according to the priority queue's ordering*

*\* @throws NullPointerException if the specified element is null*

*\*/*

public boolean add(E e) {

return offer(e);

}

*/\*\**

*\* Inserts the specified element into this priority queue.*

*\**

*\* @return {@code true} (as specified by {@link Queue#offer})*

*\* @throws ClassCastException if the specified element cannot be*

*\* compared with elements currently in this priority queue*

*\* according to the priority queue's ordering*

*\* @throws NullPointerException if the specified element is null*

*\*/*

public boolean offer(E e) {

if (e == null)

throw new NullPointerException();

modCount++;

int i = size;

if (i >= queue.length)

grow(i + 1);

size = i + 1;

if (i == 0)

queue[0] = e;

else

siftUp(i, e);

return true;

}

实际上add方法的内部调用的还是offer方法，所以我们主要看看offer是如何实现添加一个元素到堆结构中并维持这种结构不被破坏的。首先该方法定义了一 变量获取queue中实际存放的元素个数，紧接着一个if判断，如果该数组已经被完全使用了（没有可用空间了），会调用grow方法进行扩容，grow方法会根据具体情况判断，如果原数组较大则会扩大两倍+2，否则增加50%容量，由于具体代码比较清晰，此处不再赘述。

*/\*\**

*\* Increases the capacity of the array.*

*\**

*\* @param minCapacity the desired minimum capacity*

*\*/*

private void grow(int minCapacity) {

int oldCapacity = queue.length;

*// Double size if small; else grow by 50%*

int newCapacity = oldCapacity + ((oldCapacity < 64) ?

(oldCapacity + 2) :

(oldCapacity >> 1));

*// overflow-conscious code*

if (newCapacity - MAX\_ARRAY\_SIZE > 0)

newCapacity = hugeCapacity(minCapacity);

queue = Arrays.copyOf(queue, newCapacity);

}

接着判断该完全二叉树是否为空，如果没有任何节点，那么直接将新增节点作为根节即可，否则会调用siftUp添加新元素并调整结构，所以该方法是重点。

*/\*\**

*\* Inserts item x at position k, maintaining heap invariant by*

*\* promoting x up the tree until it is greater than or equal to*

*\* its parent, or is the root.*

*\**

*\* To simplify and speed up coercions and comparisons. the*

*\* Comparable and Comparator versions are separated into different*

*\* methods that are otherwise identical. (Similarly for siftDown.)*

*\**

*\* @param k the position to fill*

*\* @param x the item to insert*

*\*/*

private void siftUp(int k, E x) {

if (comparator != null)

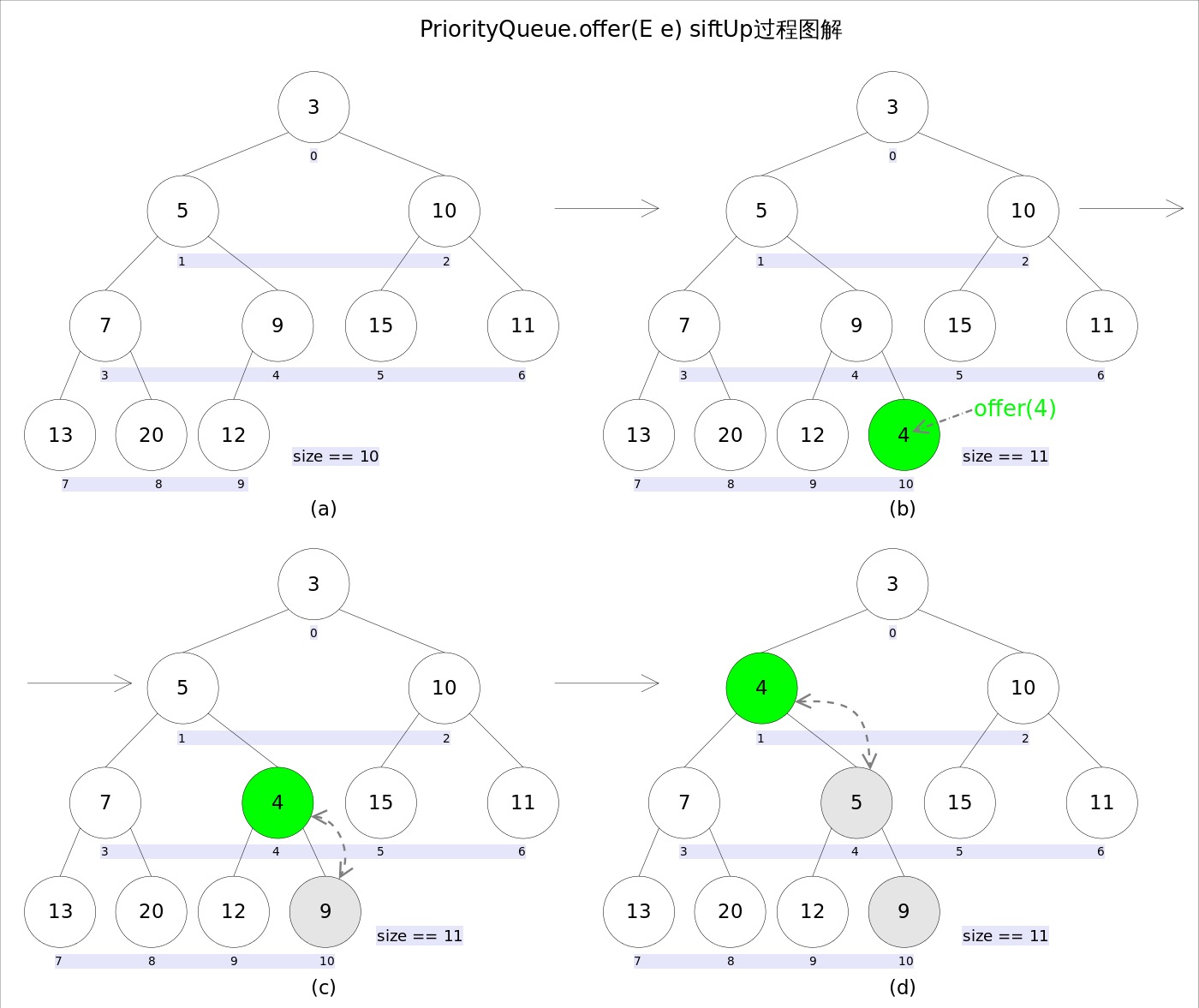
siftUpUsingComparator(k, x);

else

siftUpComparable(k, x);

}

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。  
此处会根据调用者是否传入比较器而分为两种情况，代码类似.而且比较复杂，结合图解释最好，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。

*/\*\**

*\* Retrieves, but does not remove, the head of this queue. This method*

*\* differs from {@link #peek peek} only in that it throws an exception if*

*\* this queue is empty.*

*\**

*\* <p>This implementation returns the result of <tt>peek</tt>*

*\* unless the queue is empty.*

*\**

*\* @return the head of this queue*

*\* @throws NoSuchElementException if this queue is empty*

*\*/*

public E element() {

E x = peek();

if (x != null)

return x;

else

throw new NoSuchElementException();

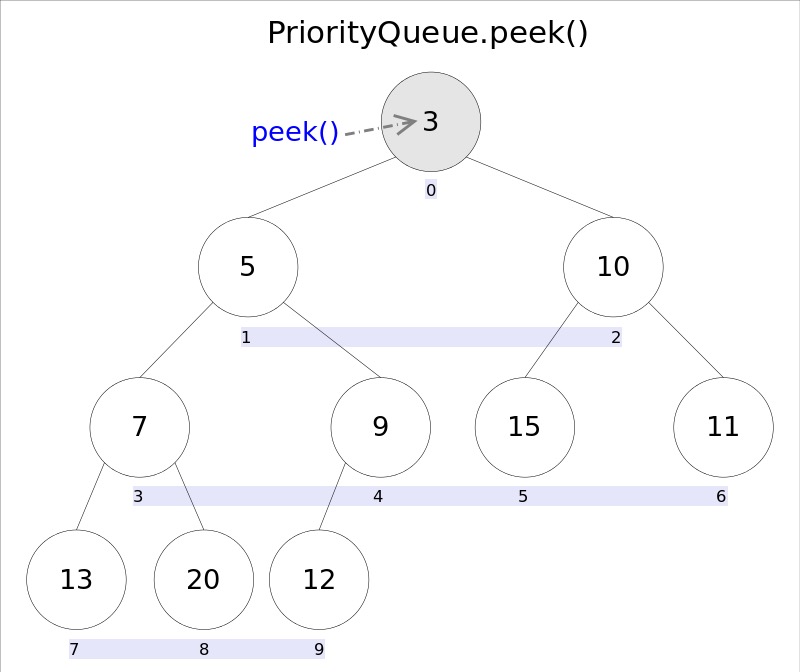
}

@SuppressWarnings("unchecked")

public E peek() {

return (size == 0) ? null : (E) queue[0];

}



**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。

*/\*\**

*\* Retrieves and removes the head of this queue. This method differs*

*\* from {@link #poll poll} only in that it throws an exception if this*

*\* queue is empty.*

*\**

*\* <p>This implementation returns the result of <tt>poll</tt>*

*\* unless the queue is empty.*

*\**

*\* @return the head of this queue*

*\* @throws NoSuchElementException if this queue is empty*

*\*/*

public E remove() {

E x = poll();

if (x != null)

return x;

else

throw new NoSuchElementException();

}

*/\*\**

*\* Retrieves and removes the head of the queue represented by this deque*

*\* (in other words, the first element of this deque), or returns*

*\* {@code null} if this deque is empty.*

*\**

*\* <p>This method is equivalent to {@link #pollFirst}.*

*\**

*\* @return the head of the queue represented by this deque, or*

*\* {@code null} if this deque is empty*

*\*/*

public E poll() {

return pollFirst();

}

public E pollFirst() {

int h = head;

@SuppressWarnings("unchecked")

E result = (E) elements[h];

*// Element is null if deque empty*

if (result == null)

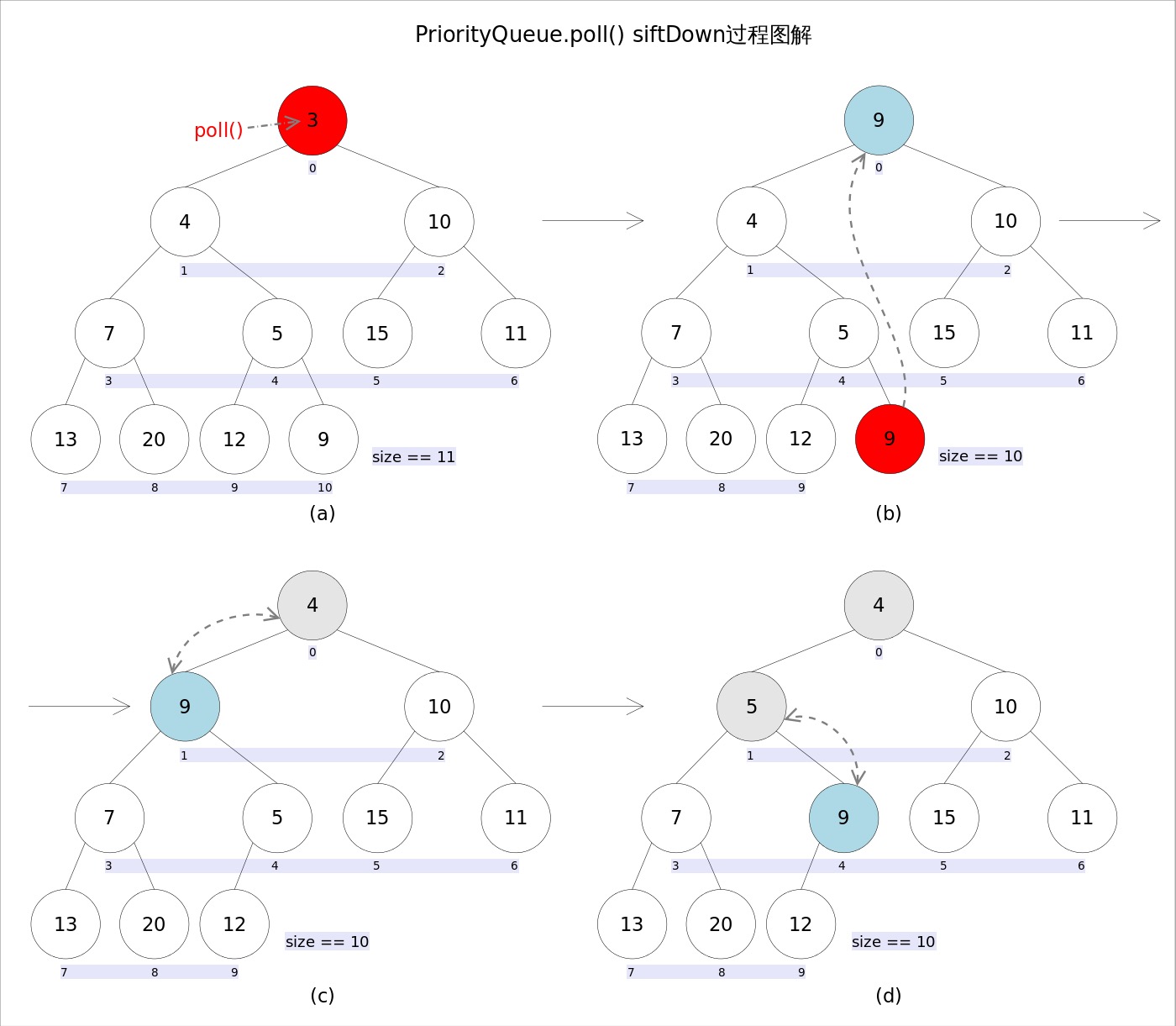
return null;

elements[h] = null; *// Must null out slot*

head = (h + 1) & (elements.length - 1);

return result;

}



**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.

*/\*\**

*\* Removes a single instance of the specified element from this queue,*

*\* if it is present. More formally, removes an element {@code e} such*

*\* that {@code o.equals(e)}, if this queue contains one or more such*

*\* elements. Returns {@code true} if and only if this queue contained*

*\* the specified element (or equivalently, if this queue changed as a*

*\* result of the call).*

*\**

*\* @param o element to be removed from this queue, if present*

*\* @return {@code true} if this queue changed as a result of the call*

*\*/*

public boolean remove(Object o) {

int i = indexOf(o);

if (i == -1)

return false;

else {

removeAt(i);

return true;

}

}

如果删除索引不是最后一个位置，那么首先会获取到最后一个节点的值并将其删除，紧接着将最后一个节点的值覆盖掉待删位置的节点值并调整结构，调整完成之后，会判断queue[i] == moved，如果为true表示新增元素之后并没有调整结构（满足堆结构），那么就会向上调整结构。（如果向下调整过结构自然是不需要再向上调整了），如果queue[i] != moved值为true表示向上调整过结构，那么将会返回moved。（至于为什么要在向上调整结构之后返回moved，主要是用于迭代器使用，此处暂时不会介绍）。

*/\*\**

*\* Removes the ith element from queue.*

*\**

*\* Normally this method leaves the elements at up to i-1,*

*\* inclusive, untouched. Under these circumstances, it returns*

*\* null. Occasionally, in order to maintain the heap invariant,*

*\* it must swap a later element of the list with one earlier than*

*\* i. Under these circumstances, this method returns the element*

*\* that was previously at the end of the list and is now at some*

*\* position before i. This fact is used by iterator.remove so as to*

*\* avoid missing traversing elements.*

*\*/*

@SuppressWarnings("unchecked")

private E removeAt(int i) {

*// assert i >= 0 && i < size;*

modCount++;

int s = --size;

if (s == i) *// removed last element*

queue[i] = null;

else {

E moved = (E) queue[s];

queue[s] = null;

siftDown(i, moved);

if (queue[i] == moved) {

siftUp(i, moved);

if (queue[i] != moved)

return moved;

}

}

return null;

}

**重点**

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。

*/\*\**

*\* Inserts item x at position k, maintaining heap invariant by*

*\* demoting x down the tree repeatedly until it is less than or*

*\* equal to its children or is a leaf.*

*\**

*\* @param k the position to fill*

*\* @param x the item to insert*

*\*/*

private void siftDown(int k, E x) {

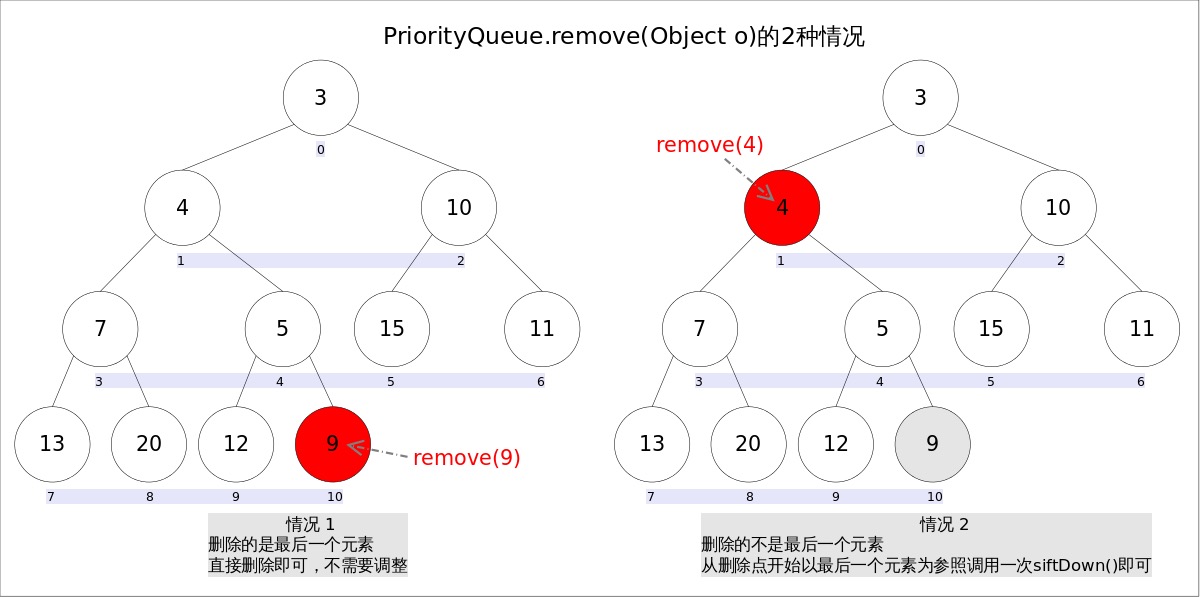
if (comparator != null)

siftDownUsingComparator(k, x);

else

siftDownComparable(k, x);

}



这里就是删除一个节点的大致过程，该方法还是比较底层的，其实PriorityQueue中是有一些其他删除节点的方法的，但是他们内部调用的几乎都是removeAt这个方法。

**总结：**

这个优先级队列的方法和使用比较繁琐了，相对来说需要理解，这里只列出了一部分日常的使用方法，很多也没有和前面的一样进行详细的讲解，内部调用方法篇幅太多了，不好一一举例，只好让大家自己去看看源码，这篇文章相当于给大家介绍了一下。