

La interface Comparable<T> en Colas de prioridades

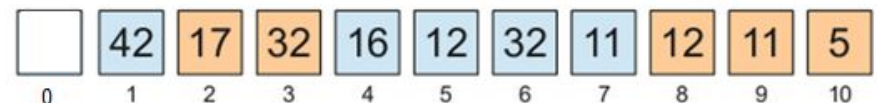
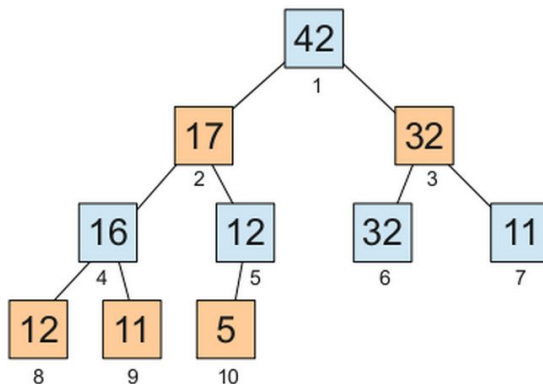
El código de esta presentación es a nivel informativo.
No se implementarán estas estructuras en la práctica.

Colas de Prioridad

En las *colas de prioridad*, el *orden lógico* de sus elementos está determinado por la prioridad de los mismos. Los elementos de mayor prioridad están en el frente de la cola y los de menor prioridad están al final. De esta manera, cuando se encola un elemento, puede suceder que éste se mueva hasta el comienzo de la cola.

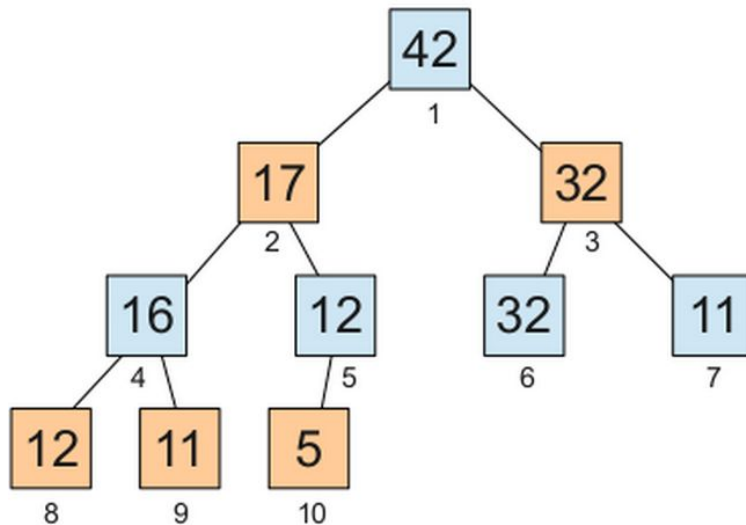
Hay varias implementaciones de cola de prioridad (listas ordenadas, listas desordenadas, ABB, etc.) pero la manera clásica es utilizar una **heap binaria**. La heap binaria implementa la cola usando un árbol binario que permite encolar y desencolar con un $O(\log n)$ y debe cumplir dos propiedades:

- **propiedad estructural:** ser un árbol binario completo de altura h , es decir, un árbol binario lleno de altura $h-1$ y en el nivel h , los nodos se completan de izquierda a derecha.
- **propiedad de orden:** El elemento máximo (en MaxHeap) está almacenado en la raíz y para cada nodo, el valor almacenado es mayor o igual al de sus hijos (en MinHeap es inverso).

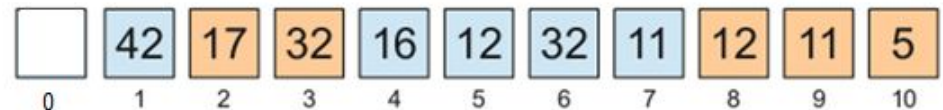


Colas de Prioridad

Usando un arreglo para almacenar los elementos, podemos usar algunas propiedades para determinar, dado un elemento, el lugar donde están sus hijos o su padre.



La raíz está almacenada en la posición 1



El hijo izquierdo está en la posición $2*i$

El hijo derecho está en la posición $2*i + 1$

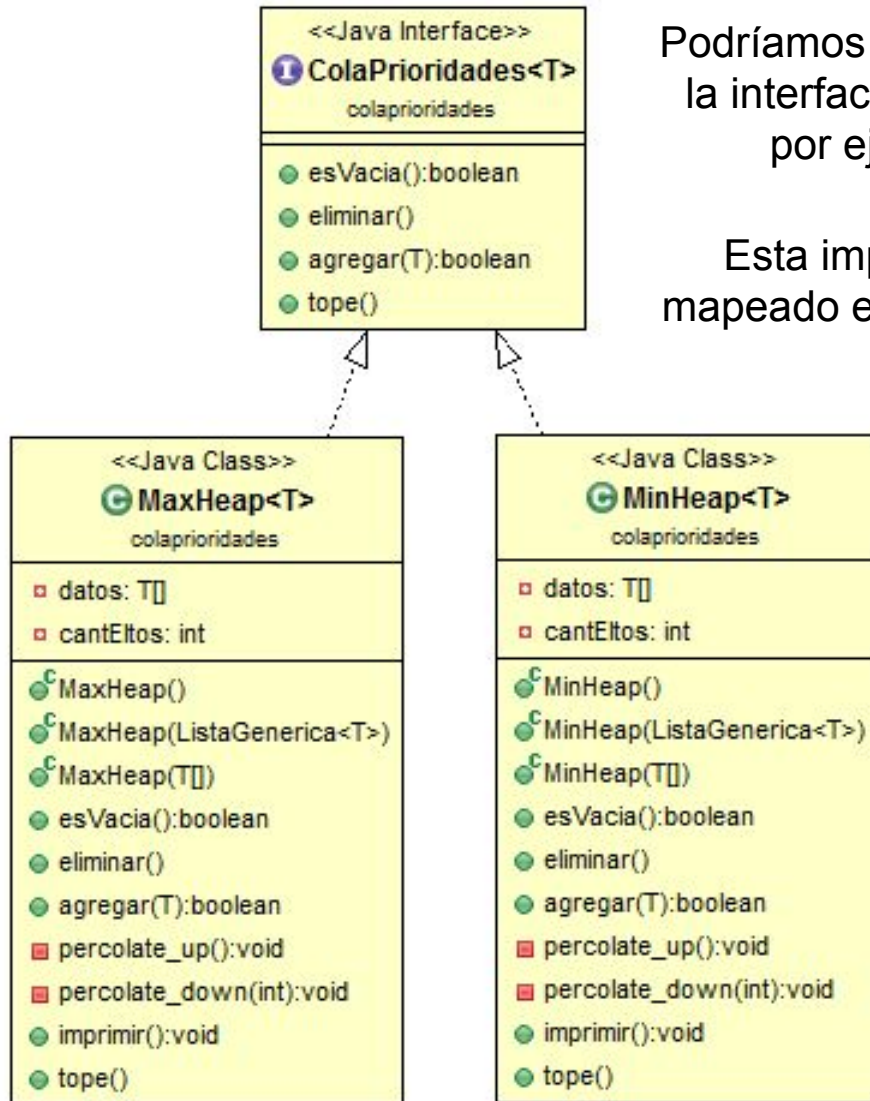
El padre está en la posición $[i/2]$

Hay dos tipos de heaps: **MinHeap** y **MaxHeap**.

MinHeap: el valor de la raíz es menor que el valor de sus hijos y éstos son raíces de minHeaps

MaxHeap: el valor de la raíz es mayor que el valor de sus hijos y éstos son raíces de maxHeaps.

Cola de Prioridades - HEAPs



Podríamos tener múltiples implementaciones de la interface `ColaPrioridades<T>` usando, por ejemplo, listas ordenadas, listas desordenadas, ABB, etc.

Esta implementación usa un árbol binario mapeado en un arreglo para implementar Colas de Prioridades -> HEAPs

MaxHeap

Constructores

```
package heap;

public class MaxHeap<T extends Comparable<T>>
    implements ColaPrioridades<T> {
    private T[] datos = (T[]) new Comparable[100];
    private int cantEltos = 0;

    public MaxHeap() {}

    public MaxHeap(ListaGenerica<T> lista) {
        lista.comenzar();
        while(!lista.fin()) {
            this.agregar(lista.proximo());
        }

        public MaxHeap(T[] elementos) {
            for (int i=0; i<elementos.length; i++) {
                cantEltos++;
                datos[cantEltos] = elementos[i];
            }
            for (int i=cantEltos/2; i>0; i--)
                this.percolate_down(i);
            . . .
        }
    }
```

En java no se puede crear un arreglo de elementos T:

```
private T[] datos = new T[100];
```

Una opción es crear un arreglo de Comparable y castearlo:

```
private T[] datos=(T[]) new Comparable[100];
```

$O(n \log n)$

En este constructor se recibe la lista, se recorre y para cada elemento se agrega y se filtra. El agregar es el método de la **HEAP**. El agregar() invoca al **percolate_up()**.

Orden lineal

En este constructor, después de agregar todos los elementos en la heap, en el orden en que vienen en el arreglo enviado por parámetro, restaura la propiedad de orden intercambiando el dato de cada nodo hacia abajo a lo largo del camino que contiene los hijos máximos invocando al método **percolate_down()**.

HEAP

Insertar/Agregar un elemento

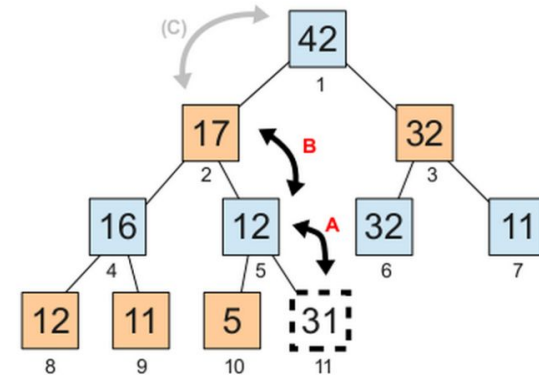
```
package heap;

public class MaxHeap<T extends Comparable<T>>
    implements ColaPrioridades<T> {
    private T[] datos = (T[]) new Comparable[100];
    private int cantEltos = 0;

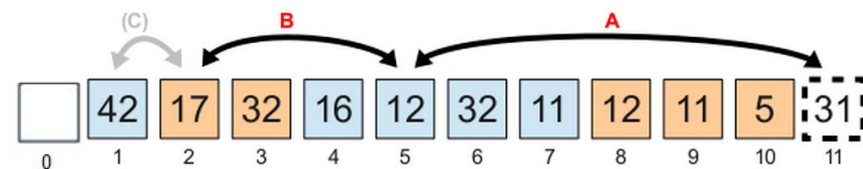
    public boolean agregar(T elemento) {
        this.cantEltos++;
        this.datos[cantEltos] = elemento;
        this.percolate_up(cantEltos);
        return true;
    }

    private void percolate_up(int indice) {
        T temporal = datos[indice];
        while (indice/2>0 &&
            datos[indice/2].compareTo(temporal)<0) {
            datos[indice] = datos[indice/2];
            indice = indice/2;
        }
        datos[indice] = temporal;
    }
    . . .
}
```

El dato se inserta como último ítem en la heap. La propiedad de orden de la heap se puede romper. Se debe hacer un filtrado hacia arriba para restaurar la propiedad de orden.



Hay que intercambiar el 31 con el 12 y después con el 17 porque 31 es mayor que ambos. El último intercambio (c), no se realiza porque 31 es menor que 42.



El filtrado hacia arriba restaura la propiedad de orden intercambiando el *elemento insertado* a lo largo del camino hacia arriba desde el lugar de inserción

HEAP

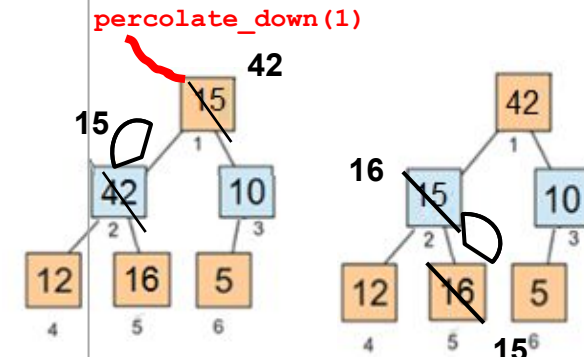
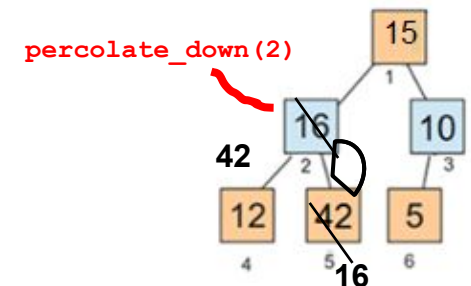
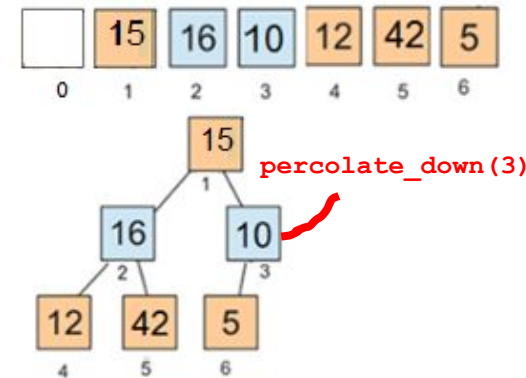
Filtrado hacia abajo: *percolate_down*

Para filtrar: se elige el mayor de los hijos y se lo compara con el padre.

```
package heap;

public class MaxHeap<T extends Comparable<T>>
    implements ColaPrioridades<T> {

    private T[] datos = (T[]) new Comparable[100];
    private int cantEltos = 0;
    . . .
    private void percolate_down(int posicion) {
        T candidato = datos[posicion];
        boolean detener_percolate = false;
        while (2 * posicion <= cantEltos && !detener_percolate) {
            //buscar el hijo maximo de candidato (hijo_máximo es el indice)
            int hijo_maximo = 2 * posicion;
            if (hijo_maximo != this.cantEltos){ //hay+ eltos,tiene hdercho
                if (datos[hijo_maximo + 1].compareTo(datos[hijo_maximo]) > 0) {
                    hijo_maximo++;
                }
            }
            if (candidato.compareTo(datos[hijo_maximo]) < 0) { //padre<hijo
                datos[posicion] = datos[hijo_maximo];
                posicion = hijo_maximo;
            } else {
                detener_percolate = true;
            }
        }
        this.datos[posicion] = candidato;
    }
    . . .
}
```



HEAP

Eliminar máximo – tope()

```
package heap;

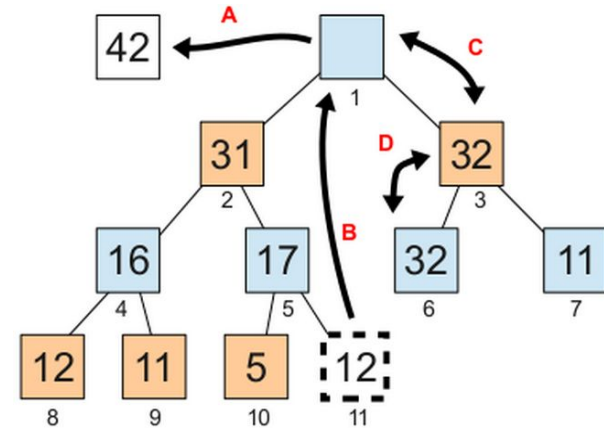
public class Heap<T extends Comparable<T>>
    implements ColaPrioridades<T>{
    private T[] datos = (T[]) new Comparable[100];
    private int cantEltos = 0;

    public T eliminar() {
        if (this.cantElto > 0) {
            T elemento = this.datos[1];
            this.datos[1] = this.datos[this.cantEltos];
            this.cantEltos--;
            this.percolate_down(1);
            return elemento;
        }
        return null;
    }

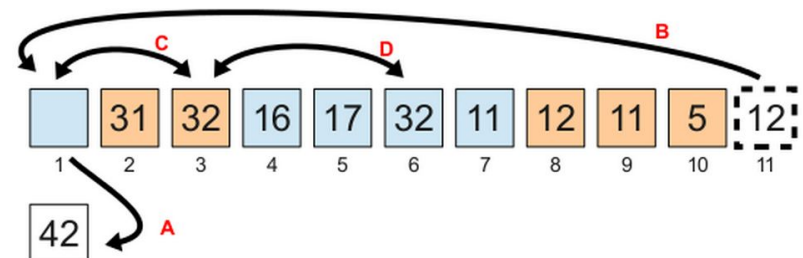
    public T tope() {
        return this.datos[1];
    }

    public boolean esVacia() {
        if (this.cantEltos>0) {
            return false;
        }
        return true;
    }
    . . .
}
```

Para extraer un elemento de la heap hay que moverlo a una variable temporal, mover el último elemento de la heap al hueco, y hacerlo bajar mediante intercambios hasta restablecer la propiedad de orden (cada nodo debe ser mayor o igual que sus descendientes).



En el arreglo se ve así:



HEAP

Problema: Encontrar la Valencia Total



El Sr. White ha encontrado una manera de maximizar la pureza de los cristales basados en ciertos compuestos químicos. Ha observado que cada compuesto está hecho de moléculas que están unidas entre sí siguiendo la estructura de un árbol binario completo.

Cada nodo del árbol almacena la valencia de una molécula y se representa como un número entero. El Sr. White utiliza un microscopio electrónico que descarga la estructura de la molécula como un stream de números enteros y le gustaría tener su ayuda para obtener automáticamente la valencia total de las hojas del árbol dado.

Cada línea de entrada comienza con un entero N ($1 \leq N \leq 1000000$), seguido de N números enteros V_i que representan las valencias de cada molécula separadas por espacios en blanco ($0 \leq V_i \leq 100$). El final de la entrada se indica mediante un caso de prueba con $N = 0$.

Ejemplo

Input:

6 4 3 2 6 0 3

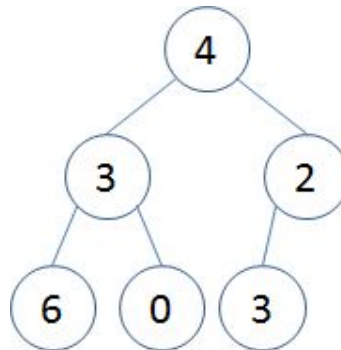
7 1 1 1 2 1 2 1

0

Output:

9

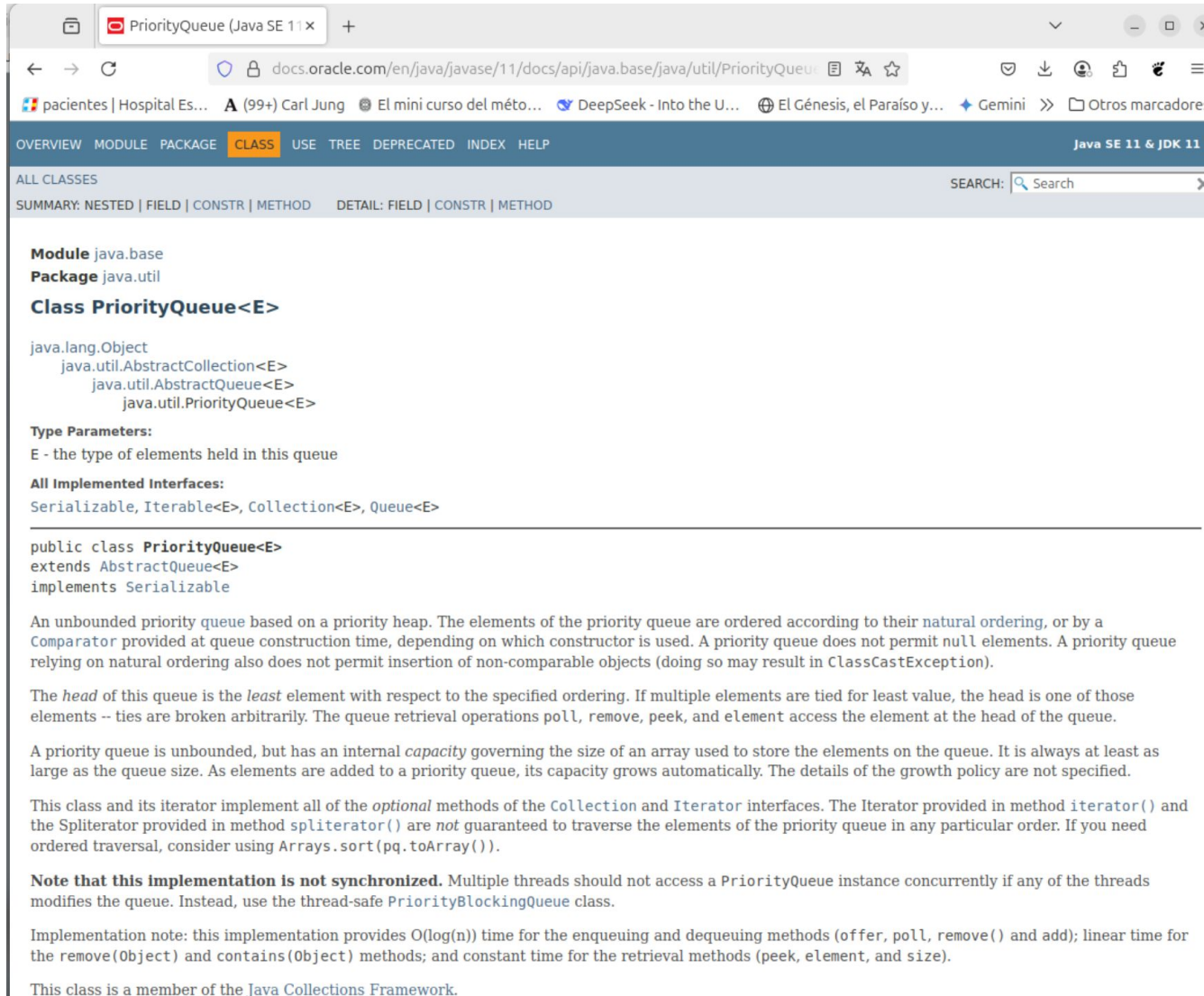
6



Se les ocurre otra solución?

HEAP

Implementación en el framework de colecciones



The screenshot shows the Oracle Java SE 11 API documentation for the `PriorityQueue` class. The browser address bar shows the URL `docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/PriorityQueue`. The page has a navigation bar with tabs for OVERVIEW, MODULE, PACKAGE, CLASS (selected), USE, TREE, DEPRECATED, INDEX, and HELP. Below the navigation bar, there's a search bar and a summary section. The main content area shows the class hierarchy: `java.lang.Object` → `java.util.AbstractCollection<E>` → `java.util.AbstractQueue<E>` → `java.util.PriorityQueue<E>`. It also lists the type parameter `E` as the type of elements held in this queue. The "All Implemented Interfaces" section lists `Serializable`, `Iterable<E>`, `Collection<E>`, and `Queue<E>`. The "public class" declaration is shown: `public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable`. The description explains that it's an unbounded priority queue based on a priority heap, ordered by natural ordering or a `Comparator`. It also mentions the `head` of the queue and the `capacity` governing the size of the internal array. The class implements `Collection` and `Iterator` interfaces. A note states that this implementation is not synchronized. An implementation note provides time complexity details: $O(\log(n))$ for enqueueing and dequeueing methods, linear time for `remove(Object)` and `contains(Object)`, and constant time for retrieval methods. Finally, it states that this class is a member of the Java Collections Framework.

Module `java.base`
Package `java.util`
Class `PriorityQueue<E>`

`java.lang.Object`
 `java.util.AbstractCollection<E>`
 `java.util.AbstractQueue<E>`
 `java.util.PriorityQueue<E>`

Type Parameters:
`E` - the type of elements held in this queue

All Implemented Interfaces:
`Serializable`, `Iterable<E>`, `Collection<E>`, `Queue<E>`

```
public class PriorityQueue<E>
    extends AbstractQueue<E>
    implements Serializable
```

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a `Comparator` provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in `ClassCastException`).

The *head* of this queue is the *least* element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations `poll`, `remove`, `peek`, and `element` access the element at the head of the queue.

A priority queue is unbounded, but has an internal *capacity* governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.

This class and its iterator implement all of the *optional* methods of the `Collection` and `Iterator` interfaces. The `Iterator` provided in method `iterator()` and the `Splitter` provided in method `spliterator()` are *not* guaranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using `Arrays.sort(pq.toArray())`.

Note that this implementation is not synchronized. Multiple threads should not access a `PriorityQueue` instance concurrently if any of the threads modifies the queue. Instead, use the thread-safe `PriorityBlockingQueue` class.

Implementation note: this implementation provides $O(\log(n))$ time for the enqueueing and dequeueing methods (`offer`, `poll`, `remove()` and `add`); linear time for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods (`peek`, `element`, and `size`).

This class is a member of the Java Collections Framework.