```
package mypackage;

import java.util.ArrayList;
import java.util.Comparator;

// An implementation of a priority queue using an array-based heap.
public class HeapPriorityQueue<K1,K2,V> extends AbstractPriorityQueue<K1,K2,V> {

    //primary collection of priority queue entries
    protected ArrayList<Entry<K1,K2,V>> heap = new ArrayList<>( );

    //Creates an empty priority queue based on the natural ordering of its keys.
    public HeapPriorityQueue( ) { super( ); }

    //Creates an empty priority queue using the given comparator to order keys.
    public HeapPriorityQueue(Comparator<K1> comp, Comparator<K2> comp2) { super(comp, comp2); }

    // protected utilities
    protected int parent(int j) { return (j-1) / 2; } // truncating division
    protected int left(int j) { return 2*j + 1; }
    protected int right(int j) { return 2*j + 2; }
    protected boolean hasLeft(int j) { return left(j) < heap.size( ); }
    protected boolean hasRight(int j) { return right(j) < heap.size( ); }

    //Exchanges the entries at indices i and j of the array list.
    protected void swap(int i, int j) {
        Entry<K1,K2,V> temp = heap.get(i);
        heap.set(i, heap.get(j));
        heap.set(j, temp);
}

//Moves the entry at index j higher, if necessary, to restore the heap property.
protected void upheap(int j) {
    while (j > 0) { // continue until reaching root (or break statement)
        int p = parent(j);
        if (compare(heap.get(j), heap.get(p)) >= 0) break; // heap property verified
        swap(j, p);
        j = p; // continue from the parent's location
    }
}

// Moves the entry at index j lower, if necessary, to restore the heap property.
protected void downheap(int j) {
    while (hasLeft(j)) { // continue to bottom (or break statement)
    int leftIndex = left(j);
    int smallChildIndex = leftIndex; // although right may be smaller
    if (hasRight(j)) {
        int rightIndex = right(j);
        if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
            smallChildIndex = rightIndex; // right child is smaller
    }
    if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
        break; // heap property has been restored
    swap(j, smallChildIndex);
    j = smallChildIndex; // continue at position of the child
    }
}
```

```
// public methods

    //Returns the number of items in the priority queue.
    public int size( ) { return heap.size( ); }

    //Returns (but does not remove) an entry with minimal key (if any).
    public Entry<K1,K2,V> min( ) {
        if (heap.isEmpty( )) return null;
        return heap.get(0);
    }

    //Inserts a key-value pair and returns the entry created.
    public Entry<K1,K2,V> insert(K1 key1, K2 key2, V value) throws IllegalArgumentException {
        checkKey1(key1); // auxiliary key-checking method (could throw exception)
        checkKey2(key2); // checking key2
        Entry<K1,K2,V> newest = new PQEntry<>(key1, key2, value);
        heap.add(newest); // add to the end of the list
        upheap(heap.size( ) - 1); // upheap newly added entry
        return newest;
    }

    //Removes and returns an entry with minimal key (if any).
    public Entry<K1,K2,V> removeMin( ) {
        if (heap.isEmpty( )) return null;
        Entry<K1,K2,V> answer = heap.get(0);
        swap(0, heap.size( ) - 1); // put minimum item at the end
        heap.remove(heap.size( ) - 1); // and remove it from the list;
        downheap(0); // then fix new root
        return answer;
    }
}
```