Heaps with priorities

We previously studied the max-heap, which implements a bag of integers with insertion of a value and extraction (polling) of a maximum value in O(log `n`) time for a heap of size `n`. Min-heaps are similar.

We now extend the idea to implement a min-heap of a set of distinct values each with a *priority*. The priority will be a **double** value. The separation of value from priority is needed in several different situations. Here are two:

- Consider finding the shortest route from Ithaca, NY, to Washington, D.C. A heap will contain points along the way, and the priorities will be the shortest distances found thus far from Ithaca to those points.
- In a discrete event simulation, pending events can be kept in a heap, with the priority being the time at which an event is to occur. Keeping the set in a min-heap makes it easy to extract the next event to process.

We will define all the methods needed, but we won't show many of the method bodies, since they are fairly easy to write based on the earlier discussion of max- and min-heaps.

Class Heap and inner class Item

We will implement a generic min-heap with priorities. The start of class `Heap` appears to the right. An object of this class implements a heap of elements of type `E`. For example, `E` could be a point on a map or an event to be simulated.

Because there are now two items to maintain, a value and its priority, class `Heap` will contain a private inner class, `Item`, shown to the right. We omit comments to save space, since is all simple. The type `E` of field `val` is the type parameter with which class `Heap` is declared. The fields are private, but they can and should still be referenced in methods in class `Heap`.

Class `Item` has the obvious constructor. Method `toString` may be helpful in debugging the methods of class `Heap`.

```
/** A heap of elements of type E. */
public class Heap<E> {

   private class Item {
      private E val;
      private double priority;

      public Item(E v, double p) {
         val= v; priority= p;
      }

      public String toString() {
         return "(" + val + ", " + priority + ")";
      }
   }
```

The fields and the class invariant

To the right, we give the declaration of the two fields needed, `b` and `size`, and their definition in the class invariant.

Elements `b[size..]` are not mentioned. We don't care about them and never reference them. We are interested only in `b[0..size-1]`, the items in the heap.

```
/** b[0..size-1] is a min-heap, i.e.
   * 1. Each Item in b[0..size-1] contains a value and its priority.
   * 2. The children of each b[k] are b[2k+1] and b[2k+2].
   * 3. The parent of each b[k] is b[(k-1)/2].
   * 4. For each k, (priority of b[k]'s parent) <= (priority of b[k]). */
private Item[] b= (Item[]) Array.newInstance(Item.class, 1500);
private int size;
```

Array `b` is initialized to contain an array of 1500 elements. The righthand side of the assignment is indeed weird. We are forced to create the array like this way because generics and arrays don't mix too well in Java —note that the type of field `val` in class `Item` is type parameter `E`. Method `Array.newInstance` creates the array. The first argument of the call indicates the type of array element, the second gives the size of the array. The array has to be cast to `Item[]` to be stored in `b`.

Methods constructor, size, swap

To the right are the constructor, method `size`, and a method that is used to swap two elements of array `b`. Method `swap` is not completely necessary, but we will see later that its use in moving array elements around during bubbling up and down and elsewhere will make some changes easier.

```
/** Constructor: an empty heap. */
public HeapPriority() {}

/** Return the size of the heap. */
public int size() { return size; }

/** Swap b[h] and b[k]. */
private void swap(int h, int k) {
   Item t= b[h]; b[h]= b[k]; b[k]= t;
}
```

©David Gries, 2018

Methods add, bubbleUp, bubbleDown

Method `add` is similar to that in the original heap of integers except that an `Item` object is stored in array `b` instead of an **int**.

Methods `bubbleUp` and `bubbleDown` can be written based on the ones written in the **int** heap implementation. Just remember that the priority of an element `b`[`k`] is in `b`[`k`].`priority`, and field priority can be referenced in class `Heap`.

As mentioned earlier, it's best to move items in `b` around only by calling method `swap`. We'll see why later.

Remember that `bubbleUp` and `bubbleDown` require that all parts of the class invariant be true except that `b`[`k`] may be out of place. Therefore, don't call one of these methods until everything else needed to make the class invariant true has been done.

```
/** Add e with priority p to the heap.
  * Takes time O(log size-of-heap).
  * Precondition: size of heap is < 1500. */
public void add(E e, double p) { … }

/** Bubble b[k] up in heap to its right place.
  * Pre: Every b[i] ≥ its parent except
  *       perhaps for b[k] */
public void bubbleUp(int k) { … }

/** Bubble b[k] down in heap to its place.
  * Pre: Every b[i] ≤ its children except
  *       perhaps for b[k]. */
private void bubbleDown(int k) { … }
```

Peek and poll

Method `peek` returns the smallest value in the heap. It's a simple method, since the smallest value is in `b`[0]. We have written its body.

Method `poll` also returns the smallest value, but in addition it removes it from the heap. Write it based on the one in the original **int** heap implementation.

```
/** Return the smallest value in the heap.
  * This operation takes constant time.
  * Precondition: the heap is not empty. */
public E peek() { return b[0].val; }

/** Remove, return smallest value in heap.
  * Worst-case time: O(log size-of-heap).
  * Precondition: the heap is not empty. */
public E poll() { … }
```

The need for a new data structure, a HashMap

In some applications that use this class `Heap<E>`, it is necessary to change the priority of a value. Therefore, we introduce method `updatePriority`, shown to the right.

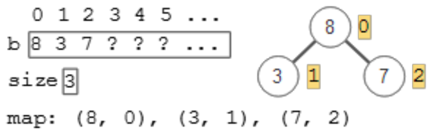
```
/** Change the priority of v in the heap to p.
  * Precondition: v is in the heap. */
public void updatePriority(E v, double p)
```

The first task of this method is to find `v` in the heap. Because of the nature of the heap, the only way to do this is to search `b`[0..`size`-1] from beginning to end, looking for `v`. But such a search has expected and worst-case time O(`size`)! That's too slow. So, we add a data structure to make the search more efficient.

A hash table has expected constant time and worst-case linear time to search for a value. So let's use a hash table. We actually need a Java `HashMap`: given a value `v`, it will return the index in `b` where `v` is. Thus, if it returns `k`, we know that `b`[`k`].`val` = `v`. The declaration of `map` is given to the right above, along with its definition.

```
/** 5. For each item b[k] in the heap,
  *    map contains the pair (b[k].val, k).
  * 6. (size of heap) = (size of map) */
HashMap<E, Integer> map= new HashMap<>();
```

To the right, we give an example. The tree has three values. You see the contents of `b` (without priorities!) and `size`. The new field `map` has three (*key*, *value*) pairs; it indicates that 8 is in `b`[0], 3 is in `b`[1], and 7 is in `b`[2].



The first step of method `updatePriority` is to get the index of `v` in `b`, using the statement shown to the right. If `v` is not in the heap, `k` will be **null**, which is an error.

```
Integer k= map.get(v);
```

Retruthifying the loop invariant

Before writing method `updatePriority`, it is necessary to make changes in all methods to make sure that the new parts of the heap invariant, concerning `map`, are maintained. (For example, `swap` has to update `map` to show that `b`[`h`] and `b`[`k`] were swapped.) This will require changes in at least methods `add`, `swap`, and `poll`. If `poll` was used as the only way to change the heap during bubbling, the bubbling methods don't need changing.

©David Gries, 2018