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 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.

**Class Heap and inner class Item**

/\*\* 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 + ")";

}

}

# We show how to 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.

# The first component of Heap that shown is a private inner class. We omit comments to save space, since we are commenting here. Each object of class Item contains a value val and a priority. Note the type E of field val is the generic type 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.

**The fields and the class invariant**

/\*\* 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;

# To the right, we give the declaration of the two fields needed, array b and int variable size, and the class invariant.

# We don’t mention b[size..1499]. We don’t care about those array elements and never reference them. We are interested only in b[0..size-1].

# Array b is initialized to contain an array of 1500 elements. The righthand side of the assignment is indeed strange. This way of creating an array of Items is needed because generics and arrays don’t mix too well in Java —note that the type of field val in class Item is the generic type parameter, E. If you ever encounter a problem creating an array where generics are used, look at method Array.newInstance.

**Methods constructor, size, add**

/\*\* Constructor: an empty heap. \*/

**public** HeapPriority() {}

/\*\* Return the size of the heap. \*/

**public** **int** size() { **return** size; }

/\*\* Add e with priority p to the heap.

\* This operation takes time O(log size-of-heap).

\* Precondition: the size of the heap is < 1500. \*/

**public** **void** add(E e, double p) {

b[size]= **new** Item(e, p);

size= size + 1;

bubbleUp(size - 1);

}

# To the right is the constructor, method size, and method add. Method add is similar to that in the original heap of integers except that an object of class Item, instead of an int has to be created and stored in array b.

# We don’t show method bubbleUp. You can figure out how to write it based on the one given in the discussion if an int max-heap. Just remember that the priority of an element b[k] is in b[k].priority, and it can be referenced even though field priority is private.

**Methods peek and poll**

/\*\* 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 and return the smallest value in the heap.

\* The worst-case time is O(log size-of-heap).

\* Precondition: the heap is not empty. \*/

**public** E poll() {

**if** (size == 1) { size= 0; **return** b[0].val; }

// At least 2 elements in heap

E val= b[0].val;

size= size - 1;

b[0]= b[size];

bubbleDown(0);

**return** val;

}

# Method peek returns the smallest value in the heap. It’s a simple method, since the smallest value is in b[0].

# Method poll also returns the smallest value, but it removes it from the heap. If the heap size is 1, it’s easy; set the size to 0 and return b[0].val.

# If the heap contains at least two elements, there is more work to do. It’s like method poll in an int heap:

# (1) Save the value to return.

# (2) Move the last value in the heap to b[0] and decrement the size of the heap.

# (3) b[0] is now large; bubble it down.

# (4) Return the value.

# Here’s an important point. Methods bubbleDown and bubbleUp require that the class invariant be true except perhaps that the value at the node to be bubbled may be out of place. Therefore, don’t bubble up or down until this is true.

**Changing the priority of an element**

# We introduce another operationIn order to make this operation logarithmic in the size of the heap, we will have to introduce another data structure, complicating matters quite a bit. But this operation is needed in some situations, such as Dijkstra’s shortest path algorithm.

/\*\* Change the priority of v in the heap to p.  
 \* Precondition: v is in the heap. \*/

**public** **void** updatePriority(E v, **double** p)

# We want to be able to change the priority of a value v on the heap, say with a call updatePriority(E v, double p). This requires us to be find v in the heap, and the only way to do it, really, is to search the heap for it. But that is an expected linear-time operation, because there is no ordering of the values in array b. Therefore, we introduce another field map

/\*\* 5. map contains an entry for each value in the  
 \* heap, and no other entries.

\* 6. For each item b[k] in the heap,  
 \* map contains the pair (b[k].val, k). \*/

HashMap<E, Integer> map= new HashMap<>();