CSE250 Week 12: Heaps---for more than just Priority Queues

A **heap** is a full binary tree, minus some subset of leaves from right-to-left, whose comparable keys obey the invariant that if u is a child of v, then u.key $\leqslant$ v.key.

This defines a **max-heap** and means that `root.key` is always the largest value in the heap (not necessarily uniquely). There is a corresponding notion of **min-heap** where the root holds the least value and children compare $\geqslant$ their parents. Unlike in a BST:

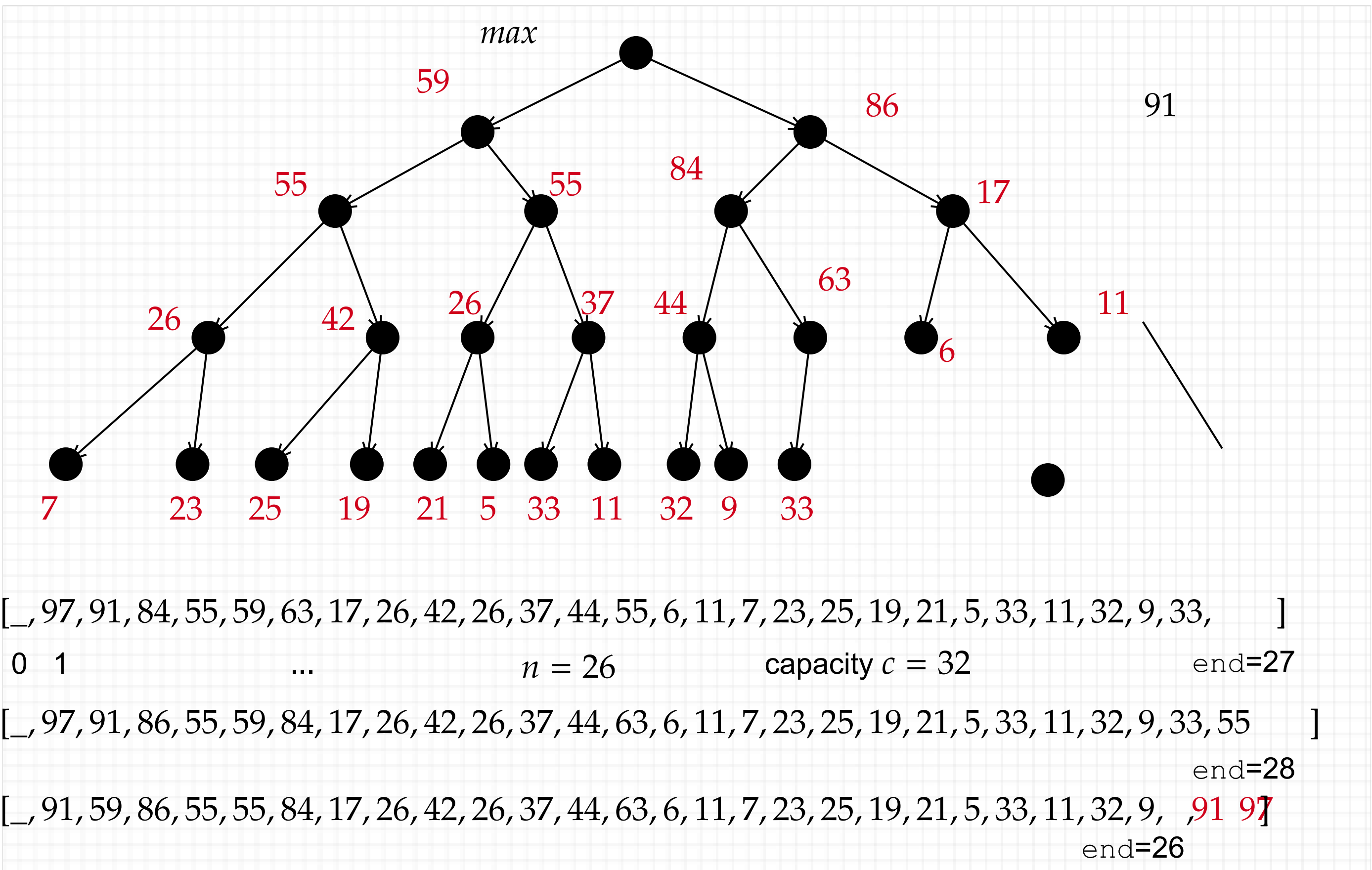
(a) the left-to-right layout does not matter to the heap's operation or meaning;
(b) the up/down sortedness is too weak to help you `find` a key in less than linear time.

By (a), we may as well suppose the tree is laid out so that the leaves in the bottom level fill in left-to-right. The text uses the standard term **complete tree** for this, but I prefer "raster tree" to connote that the tree fills in level-by-level going down and left-to-right in each level.

The signature implementation of a heap is an array with n elements. Happily, the text defines the version that does not use cell 0 of the array. In ancient times, there was no special support for the `swap` operation, so it was further convenient to use cell 0 as swap space. Then:

- Nodes $u, v, \dots$ are identified with indices in $[1 \dots n]$, n treated inclusively in an array of size $n+1$, and 1 being the root;
- The parent of a non-root node v is always $u = v/2$ (integer division).
- The left child of a non-leaf node u has index $2u$, while the right-child (if present) has $2u+1$.
- To facilitate adding and removing elements---and optimize memory usage generally---the array has a fixed size c (so n can be up to $c-1$) with the current $n+1$ maintained as a field or variable `end` apart from the array.

Example (different from the text's)



- Insert 86: it "bubbles up" to the position currently occupied by 84..
- Then pop 97. The right-hand 55 key which got swapped into the last position is first lifted to the root. Then it "sinks down" (like a stone, says the text), always swapping with the larger child, so that 91 is the new max and it settles where 59 was.

Note that keys can jump around after a pop. But the time is never worse than proportional to the height of the tree, which by its enforced balance is $O(\log n)$.

If we add 5 more elements, we will make a **full** binary tree and (since $c = 32$) also fill the capacity of the array. If we add one more element, we will have to double the size of the array and recopy the elements. The ethic with heaps, in my (ancient) experience, is to right-size them to begin with, because the idea is that they use memory **in-place** without using temporary storage for copying (like QuickSort versus MergeSort). [Important note: this meaning of heap is not related to that of the "system heap", whose ethic is to be expandable without the layout regularity needed for values on the "system stack" (which really is organized like a stack).]

Coding Heaps

This time I'll use several more helper functions than the text does. Scala seems not to have low-level support for fast `swap` like C/C++ implementations do, so we'll use `heap(0)` like the text does. (What follows is not real code inside a `Heap` or `PriorityQueue` class.)

```
def swap(heap,i,j): Unit = {      //REQ: 1 <= i,j <= n < end
   heap(0) = heap(i)
   heap(i) = heap(j)
   heap(j) = heap(0)
}
```

Some texts define `left(u) = 2u` and `right(u) = 2u+1` and `parent(u) = u/2`, but this does not much matter. Here are the two most important helpers, not used as-such in the text.

```
def fixDown(u: Int): Unit = {
   var st = u          //"stone" in text.  Note var needed in Scala for val params
   while (st < (end+1)/2) {
      val gc = if (keyComp(heap(2*st),heap(2*st+1)) > 0 || 2*st+1 >= end)
                     2*st else 2*st+1
      if (keyComp(heap(st), heap(gc)) < 0) {
         swap(st,gc)
         st = gc
      } else {
         return
      }
   }     //must allow for leftChild in-bounds but rightChild >= end where
}        //it could have a garbage value

def fixUp(u): Unit
   var bubble = u        //note bubble > = 2 means parent exists and is bubble/2
   while (bubble >= 2 && keyComp(heap(bubble),heap(bubble/2)) > 0) {
      swap(bubble,bubble/2)
      bubble = bubble / 2        //integer division
   }
}
```

Side note: If one could store a "plus infinity" value in cell 0 as a **sentinel value**, then one could marginally simplify and speed the while loop by removing the `bubble >= 2` check. Since `1 / 2 = 0` with integer division, the loop would stop at the root anyway. But modern branched execution mostly negates the supposed gain, which was dubious even in ancient times.

Now we can quickly code the main operations. Because both `fixDown` and `fixUp` run in $O(\log n)$

time, so do these operations.

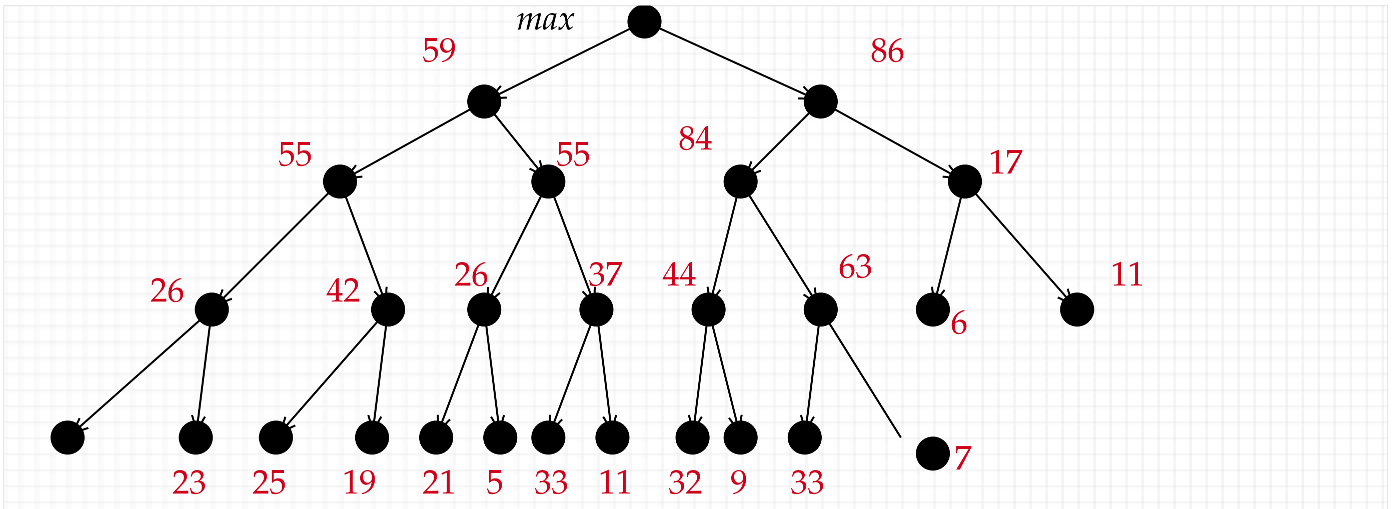
```
def enqueue(item): Unit = {
   if (end >= capacity) { ... }  //blah-blah...
   heap(end) = item
   fixUp(end)
   end += 1
}

def pop(): A = {          //dequeue in text and Scala generally
   val ret = heap(1)
   swap(1,end-1)          //sometimes it is useful to put max there
   end -= 1               //must be done first
   fixDown(1)
   return ret
}
```

Making and Sorting Heaps (not all in the text)

To create a heap from an array `arr` of items in any order, one can create an empty heap and `enqueue` the items one by one. That would take $O(n \log n)$ time on the whole, using repeated calls to `fixUp`. There is a quicker way to make a heap directly from `arr`, assuming there is at least one unused cell n at the end. (Well, not having to make this assumption is the main reason other authorities define heaps with the root in cell 0. The version with `u <- end/2` was OK too.)

```
def makeHeap(arr: Array[A], n: Int): Unit = {  //REQ: arr(n) exists and is empty
   arr(n) = arr(0)
   end = n+1                             //keep end with this value
   for (u <- (end-1)/2 to 1 by -1) {     //INV: Sub-heaps rooted at v > u are valid
      fixDown(u)                         //now INV holds for u too
   }
}
```



Theorem (not in the text): `makeHeap` runs in $O(n)$ time.

Proof. Consider the full binary tree case $n = 2^{h+1} - 1$. The worst possible thing that can happen is that `fixDown` has to swap once for every internal node just above the leaves, twice for each node in the level above, three times for the next level, up to h times for the root. Putting $m = 2^{h-1} \approx n/4$, this worst-case time adds up to

$$m*1 + \frac{m}{2}*2 + \frac{m}{4}*3 + \frac{m}{8}*4 + \cdots + 1*h \ .$$

This is order-of $2m$ times the sum $\sum_{i=1}^{h} \frac{i}{2^i}$. This sum converges even when h is allowed to go to infinity, so the time is $O(m) = O(n)$. ☒

After building the heap, the max element can be read off immediately (hence in $O(1)$ time) but the next elements may not be so immediate. Still, we can state:

Corollary: A priority queue from n unsorted items can be built in $O(n)$ time and operated in $O(\log n)$ time per operation.

Corollary: For any k, the top k items in an arbitrary unsorted array of n items can be found in $O(n + k \log n)$ time.

When $k = o(n)$, this beats the $\Theta(n \log n)$ time for sorting the array and outputting the top k elements. Often k is regarded as a constant, such as for generating "Top Ten" lists.

The case $k = n$ gives us the whole sorted array. We can get it neatly by:

```
makeHeap(arr, n)
while(!isEmpty) {
   pop()
}
```

By the behavior of `pop()` always putting the popped element past the end, the result of iterating it is a fully sorted array, done entirely **in-place**. This is **HeapSort**. It, too, runs in $\Theta(n \log n)$ time. The observed downside compared to QuickSort is that it tends to make more swaps on randomly generated cases of arrays. For HeapSort, one does *not* need `makeHeap` to run in $O(n)$ time (so the text is OK to ignore that detail)---but it helps.

Heaps With External Updates (not in the text)

The real usefulness of coding `fixDown` and `fixUp` as helpers is to model **dynamic** updates. For example, priorities of jobs can change suddenly owing to external events. We want to quickly re-constitute a legal heap with the changed key value(s). When the changes are one-at-a-time, the operations we need are simple:

- If the key of item u increases, we need a call to `heap.fixUp(u)`.
- If the key of u decreases, we need to do `heap.fixDown(u)`.

Thus, an individual update can be handled in $O(\log n)$ time at worst---and often much better, when no actual swap is needed or maybe just one. This is good enough to call the updates "on-the-fly."

The main challenge is the following: *how to communicate the change to the item in the heap that needs it?*

- We cannot say the item is "node u" because the update will change the index value of u.
- We cannot count on finding the item associatively, because in a heap that is $\Theta(n)$ time.
- Even if we make the heap array merely store a reference to the main object which we keep in an associative container (so we can find it by a key such as its name which may be different from its priority value), we still have to notify the heap to do a corresponding update.
- If we try to keep an iterator on the item, we'll have to update u inside it whenever a swap is made.
- One idea is that the heap can store references to items that are maintained elsewhere.
- We still have to communicate the results of a swap to those items, so they need to keep the actual index in the heap. See:

https://stackoverflow.com/questions/46996064/how-to-update-elements-within-a-heap-priority-queue