

# CSC 378 Lecture 9

July 12, 2001

For another reference, see chapter 7 of the textbook (CLR).

## 1 Priority Queues

A *priority queue* is an abstract data type (ADT) that operates on a set  $S$  where each element  $x \in S$  has a priority  $p(x)$  which comes from a well-ordered universe (usually the natural numbers). There are three operations on this set:

- **INSERT**( $S, x$ ): insert an element  $x$  in the set  $S$ .
- **MAXIMUM**( $S$ ): return an element  $x \in S$  with highest priority.
- **EXTRACT-MAX**( $S$ ): remove and return an element  $x \in S$  with highest priority.

Priority queues are very useful. Some of their applications are:

- Job scheduling in operating systems
- Printer queues
- Event-driven simulation algorithms
- Greedy algorithms

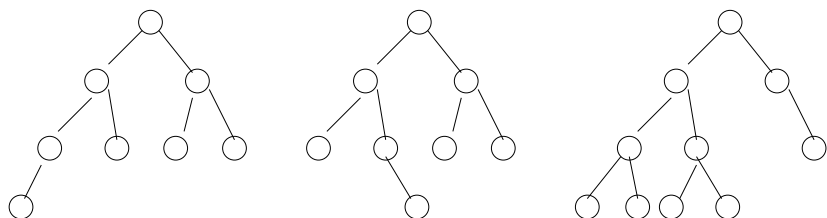
There are several possible data structures for implementing priority queues:

- Unsorted list: takes time  $\Theta(n)$  for **EXTRACT-MAX** in the worst-case.
- Sorted list (by priorities): takes time  $\Theta(n)$  for **INSERT** in worst-case.
- Red-Black tree (key-values are priorities): each operation takes time  $\Theta(\log n)$ . If we augment the tree to contain a pointer to the maximum priority element, then we can support **MAXIMUM** in time  $\Theta(1)$  (**EXTRACT-MAX** will still take  $\Theta(\log n)$  because of the deletion).
- Direct addressing: if the universe  $U$  of priorities is small and the priorities are all distinct, then we can store an element with priority  $k$  in the  $k$ th cell of an array. **INSERT** takes time  $\Theta(1)$ . **MAXIMUM** and **EXTRACT-MAX** require time  $\Theta(|U|)$  in the worst-case (have to look at each location to find the first nonempty one).

## 2 Heaps

We will look at one particular data structure for priority queues in depth. They are called *heaps* and are defined as follows: a heap is a binary tree  $T$  of elements with priorities such that

1.  $T$  is *complete*: this means that every level of the tree is full except perhaps the bottom one, which fills up from left to right. For example:



Complete

Not complete

Not complete

2. For each node  $x$  in  $T$ , if  $x$  has a left-child, then  $p(x) \geq p(\text{left}(x))$  and if  $x$  has a right-child, then  $p(x) \geq p(\text{right}(x))$ .

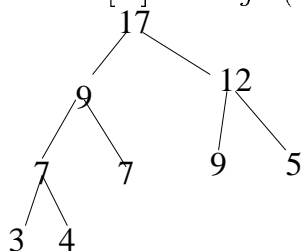
We can conclude a few immediate facts about heaps from the definition. First of all, the root has maximum priority. Secondly, every subtree of a heap is also a heap (in particular, an empty tree is a heap). Finally, since heaps are complete, if a heap contains  $n$  nodes, then its height  $h$  is  $\Theta(\log n)$ . To see this, consider a heap  $T$  with  $n$  nodes. Every level of  $T$  is full except perhaps the bottom level. We can count the number of nodes above the bottom level with the sum

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1.$$

Then  $2^h - 1 < n$ , so  $h < \log(n + 1) = O(\log n)$ . To show that  $h = \Omega(\log n)$ , just count the number of nodes in a tree of height  $h$  where every level is full, including the bottom level. This will be  $\geq n$ .

### 2.1 Storing heaps

Traditionally, a heap is stored by using an array  $A$  together with an integer *heapsize* that stores the number of elements currently in the heap (or the number of nonempty entries in  $A$ ). The following conventions are used to store the nodes of the tree in the array: the root of the tree is stored at  $A[1]$ , the two children of the root are stored at  $A[2]$  and  $A[3]$ , the four grandchildren of the root are stored at  $A[4]$ ,  $A[5]$ ,  $A[6]$ ,  $A[7]$ , etc. In general, if element  $x$  is stored at  $A[i]$ , then  $\text{left}(x)$  is stored at  $A[2i]$  and  $\text{right}(x)$  is stored at  $A[2i + 1]$ .



$A = [17, 9, 12, 7, 7, 9, 5, 3, 4]$

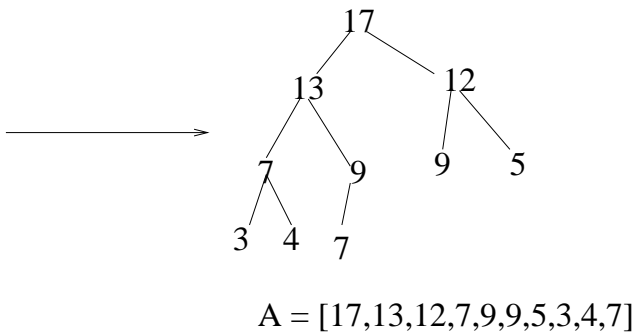
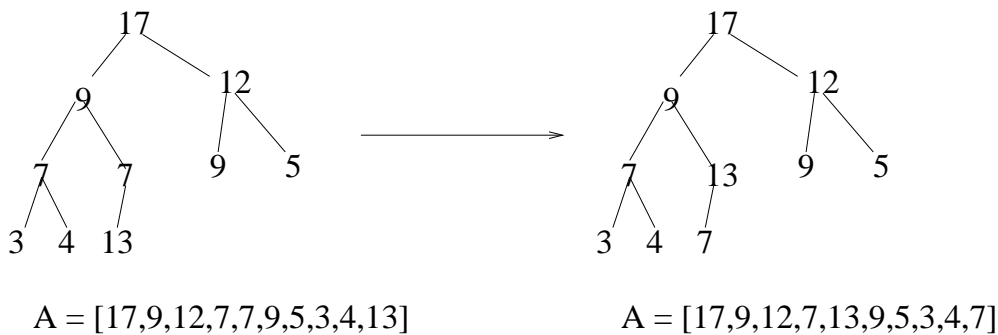
If the size of the array is close to the number of elements in the heap, then this data structure is extremely space-efficient because we don't have to store any pointers. We can use a dynamic array to ensure that this is true (recall that the amortized cost of managing a dynamic array is small).

## 2.2 Implementing priority queues

We can perform the priority queue operations on a heap as follows:

- **INSERT:** Increment *heapsize* and add the new element at the end of the array. The result might violate the heap property, so "percolate" the element up (exchanging it with its parent) until its priority is no greater than the priority of its parent.

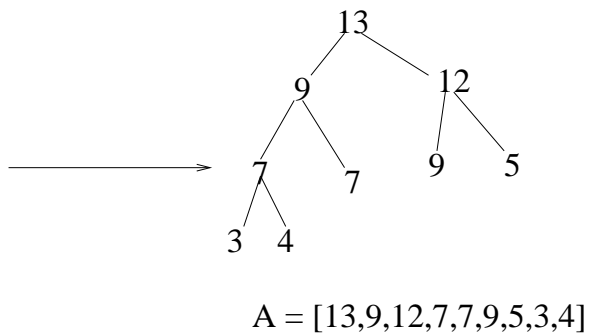
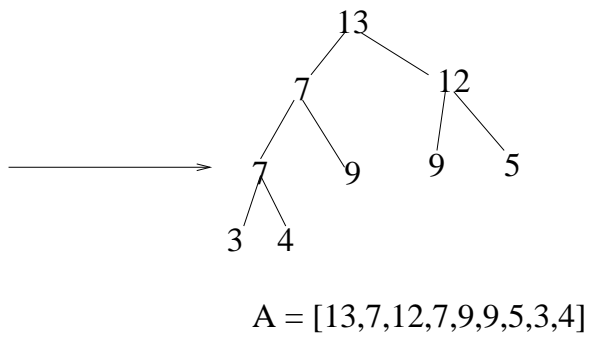
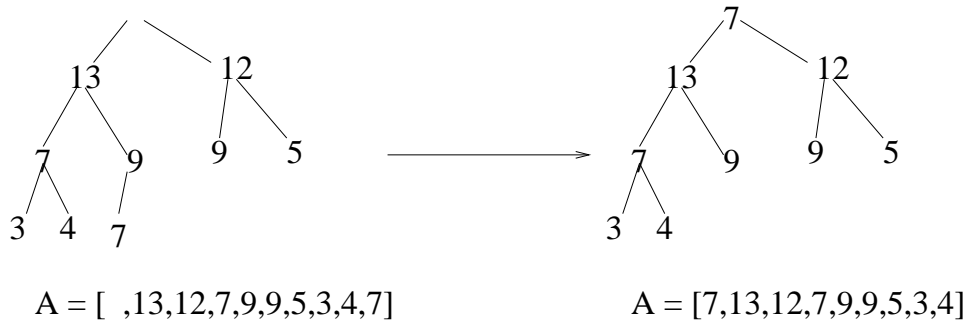
For example, if we perform **INSERT(13)** on the previous heap, we get the following result (showing both the tree and the array for each step of the operation):



In the worst-case, we will have to move the new element all the way to the root, which takes time  $\Theta(\text{height of heap}) = \Theta(\log n)$ .

- **MAXIMUM:** Simply return  $A[1]$  (if *heapsize*  $\geq 1$ ).  $\Theta(1)$  time.
- **EXTRACT-MAX:** Decrement *heapsize* and remove the first element of the array. In order to be left with a valid heap, move the last element in the array to the first position (so the heap now has the right "shape"), and percolate this element down until its priority is no smaller than the priorities of both its children. Do this by exchanging the element with its child of highest priority at every step.

For example, if we perform **EXTRACT-MAX** on the previous heap, we get the following result (showing both the tree and the array for each step of the operation):



As with INSERT, we may wind up moving the last element from the root all the way down to a leaf, which takes  $\Theta(\text{height of heap}) = \Theta(\log n)$  in the worst-case.

The "percolating down" of an element that we just described for EXTRACT-MAX is a very useful operation for heaps. In fact, it's so useful that it already has a name: If  $x$  is the element initially stored at  $A[i]$ , and assuming that the left and right subtrees of  $x$  are heaps. Then HEAPIFY( $A, i$ ) percolates  $x$  downwards until the subtree of the element now stored at  $A[i]$  is a heap.

```

HEAPIFY(A, i)
  largest := i;
  if ( 2i <= heapsize and A[2i] > A[i] ) then
    largest := 2i;
  endif
  if ( 2i+1 <= heapsize and A[2i+1] > A[largest] ) then
    largest := 2i+1;
  endif
  if ( largest != i ) then
    swap A[i] and A[largest];
    HEAPIFY(A, largest);
  endif

```

```

endif
END

```

The running time, as with `EXTRACT-MAX`, is  $\Theta(\log n)$ .

### 2.3 Building heaps

If we start with an array  $A$  of elements with priorities, whose only empty slots are at the far right, then we can immediately view  $A$  as a complete binary tree.  $A$ , however, is not necessarily a heap unless the elements are ordered in a certain way. There are several options for making  $A$  into a heap:

1. Sort  $A$  from highest priority element to lowest. Clearly  $A$  will now obey part 2 of the heap definition (actually, every sorted array is a heap, but every heap is not necessarily a sorted array). This takes time  $\Theta(n \log n)$  if we use, say, the guaranteed fast version of quicksort from homework 1.
2. We can simply make a new array  $B$  and go through every element of  $A$  and `INSERT` it into  $B$ . Since `INSERT` takes time  $\Theta(\log n)$  and we do it for each of the  $n$  elements of  $A$ , the whole thing takes time  $\Theta(n \log n)$ .
3. The most efficient way is to use `HEAPIFY`: notice that every item in the second half of  $A$  corresponds to a leaf in the tree represented by  $A$ , so starting at the "middle" element (i.e., the first nonleaf node in the tree represented by  $A$ ), we simply call `HEAPIFY` on each position of the array, working back towards position 1.

```

BUILD-HEAP(A)
  heapsize := size(A);
  for i := floor(heapsize/2) downto 1 do
    HEAPIFY(A,i);
  end for
END

```

Because each item in the second half of the array is already a heap (it's a leaf), the preconditions for `HEAPIFY` are always satisfied before each call. For example, if  $A = [1,5,7,6,2,9,4,8]$ , then `BUILD-HEAP(A)` makes the following sequence of calls to `HEAPIFY` (you can check the result of each one by tracing it):

```

HEAPIFY( [1,5,7,6,2,9,4,8], 4 ) = [1,5,7,8,2,9,4,6]
HEAPIFY( [1,5,7,8,2,9,4,6], 3 ) = [1,5,9,8,2,7,4,6]
HEAPIFY( [1,5,9,8,2,7,4,6], 2 ) = [1,8,9,6,2,7,4,5]
HEAPIFY( [1,8,9,6,2,7,4,5], 1 ) = [9,8,7,6,2,1,4,5]

```

Since we make  $O(n)$  calls to `HEAPIFY` and since each one takes  $O(\log n)$  time, we immediately get a bound of  $O(n \log n)$ . But in fact, we can do better by analyzing more carefully: basically, we call `HEAPIFY` on each subtree of height  $\geq 1$  and `HEAPIFY` runs in time proportional to the height of that subtree. So we can estimate the total running time as follows:

$$O\left(\sum_{h=1}^{\log n} h \times \text{number of subtrees of height } h\right).$$

The sum goes to  $\log n$  because the height of the whole tree is  $\Theta(\log n)$ . A tree with  $n$  nodes contains at most  $\lceil n/2^{h+1} \rceil$  nodes of height  $h$  (why?), so it contains at most the same number of subtrees of height  $h$ . Therefore, the running time is:

$$O\left(\sum_{h=1}^{\log n} h \times \lceil n/2^{h+1} \rceil\right) = O\left(n \sum_{h=1}^{\infty} h/2^h\right) = O(n).$$

The last equation comes from the fact that  $\sum_{h=1}^{\infty} h/2^h \leq 2$  (p. 44 of CLR). So BUILD-HEAP runs in time  $O(n)$ .