1 Heaps

A heap is a type of data structure. One of the interesting things about heaps is that they allow you to find the largest element in the heap in $O(1)$ time. (Recall that in certain other data structures, like arrays, this operation takes $O(n)$ time.) Furthermore, extracting the largest element from the heap (i.e. finding and removing it) takes $O(\log n)$ time. These properties make heaps very useful for implementing a "priority queue," which we'll get to later. They also give rise to an $O(n \log n)$ sorting algorithm, "heapsort," which works by repeatedly extracting the largest element until we have emptied the heap.

1.1 The heap data structure

1.1.1 Intuitive view of the heap

You can view a max heap as a binary tree, where each node has two (or fewer) children, and the key of each node (i.e. the number inside the node) is greater than the keys of its child nodes. (From now on I'm going to say node i is larger than node j when I really mean that the key of i is larger than the key of j.) There are also min heaps, where each node is smaller than its child nodes, but here we will talk about max heaps, with the understanding that the algorithms for min heaps are analogous.

For example, the root of a max heap is the largest element in the heap. However, note that it's possible for some nodes on level 3 to be smaller than nodes on level 4 (if they're in different branches of the tree).

The heap is an almost complete binary tree, in that all levels of the tree have to be completely filled except for the level at the bottom. The bottom level gets filled from left to right.

Definition: We define the "height" of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf. The height of a heap is the height of its root.

Fact: A heap of n nodes has a height of $\lfloor \log n \rfloor$. (Why? Hint: if a heap has height h, what are the minimum and maximum possible number of elements in it? Answer: $2^h \leq n \leq 2^{h+1} - 1$)

Definition: The "max-heap property" (of a heap A) is the property we were talking about, where for every node i other than the root, $A[parent[i]] \geq A[i]$. The "min-heap property" is defined analogously.

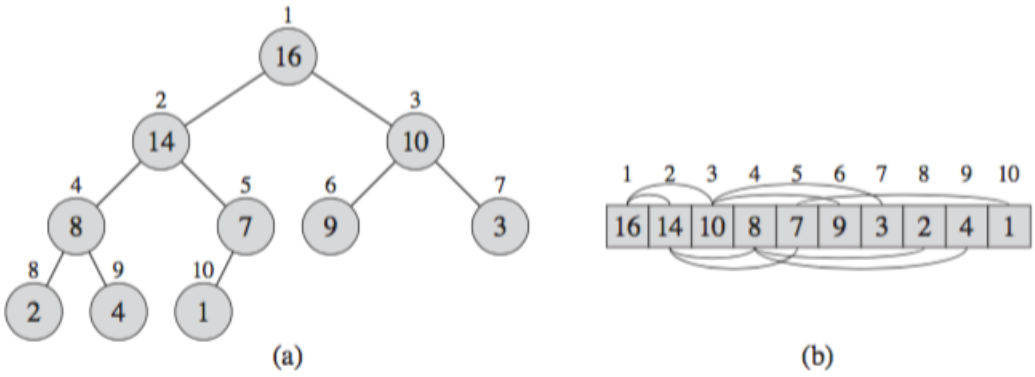

Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

1.1.2 Implementation of the heap

You can implement a heap as an array. This array is essentially populated by "reading off" the numbers in the tree, from left to right and from top to bottom.

The root is stored at index 1, and if a node is at index i, then

- Its left child has index $2i$
- Its right child has index $2i + 1$
- Its parent has index $\lfloor i/2 \rfloor$

Furthermore, for the heap array A, we also store two properties: $A.length$, which is the number of elements in the array, and $A.heapsize$, which is the number of array elements that are actually part of the heap. Even though the array A is filled with numbers, only the elements in $A[1..A.heapsize]$ are actually part of the heap.

Fact: The leaves of the heap are the nodes indexed by $\lfloor n/2 \rfloor + 1, \ldots, n$.

1.2 Maintaining the heap property

Heaps use this one weird trick to maintain the max-heap property. Unlike most of the algorithms we've seen so far, MaxHeapify assumes a very specific type of input. Given a heap A and a node i inside that heap, MaxHeapify attempts to "heapify" the subtree rooted at i, i.e., it changes the order of nodes in A so that the subtree at i satisfies the max-heap property. However, it assumes that the left subtree of i and the right subtree of i are both max-heaps in and of themselves, so the only possible violation is that i might be smaller than its immediate children.

In order to fix this violation, the algorithm finds which node is largest: i, i's left child, or i's right child. If i is largest, the max-heap property is already satisfied. Otherwise, it swaps i with the largest node, which causes the top of the heap to be fine. But this might mess up the subtree that used to be under the largest node, because i might be smaller than some of the nodes in that subtree. So now we call MaxHeapify on that subtree (because the only possible violation of the max-heap property occurs at the top of the subtree).

MaxHeapify runs in time $O(h)$, where h is the height of the node.

Exercise: Show that the worst-case running time of MaxHeapify on a heap of size n is $\Omega(\log n)$.

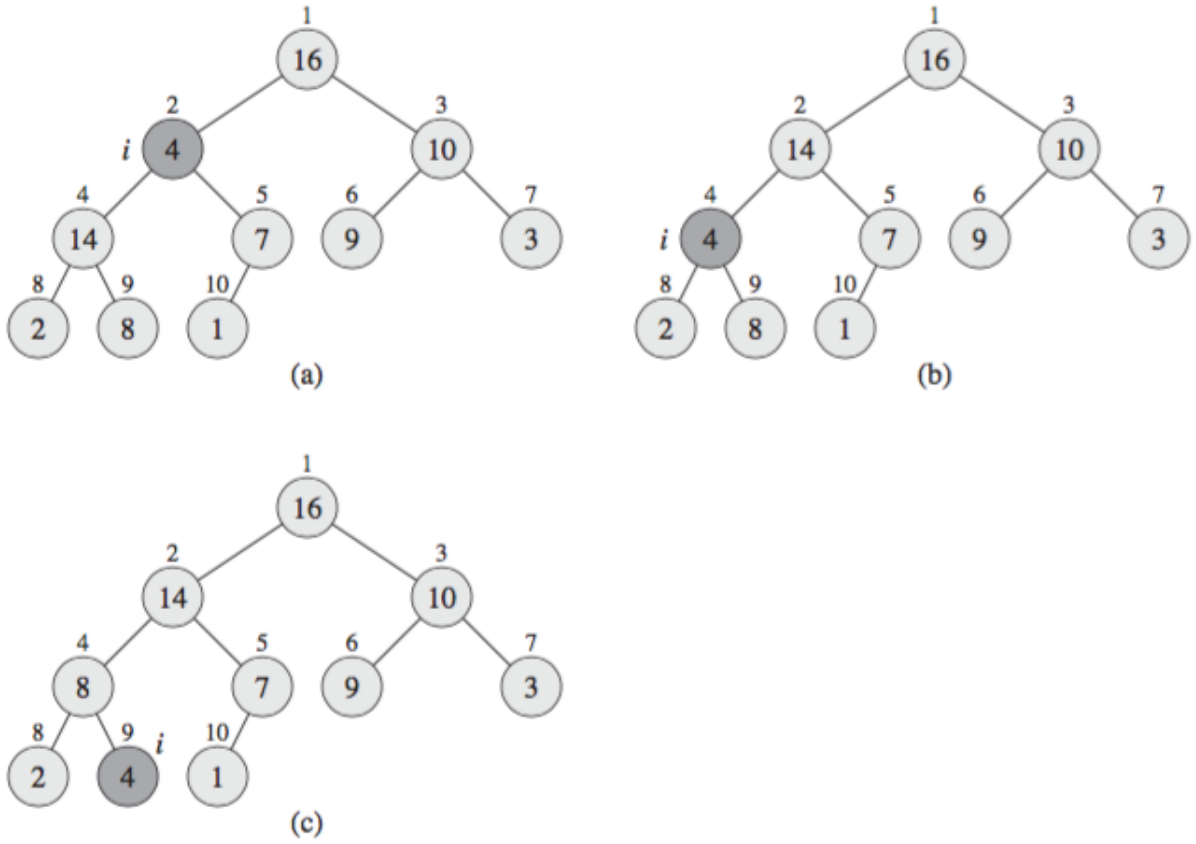

Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

Recall that we are implementing heaps as arrays, so the actual code of MaxHeapify takes as input an array A and an index i into that array.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

1.3 Building a heap

The BuildMaxHeap procedure runs MaxHeapify on all the nodes in the heap, starting at the nodes right above the leaves and moving towards the root. We start at the bottom because in order to run MaxHeapify on a node, we need the subtrees of that node to already be heaps.

Recall that the leaves of the heap are the nodes indexed by $\lfloor n/2 \rfloor + 1, \ldots, n$, so when we use our array implementation we just start at the index $\lfloor A.length/2 \rfloor$ and move towards 1.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

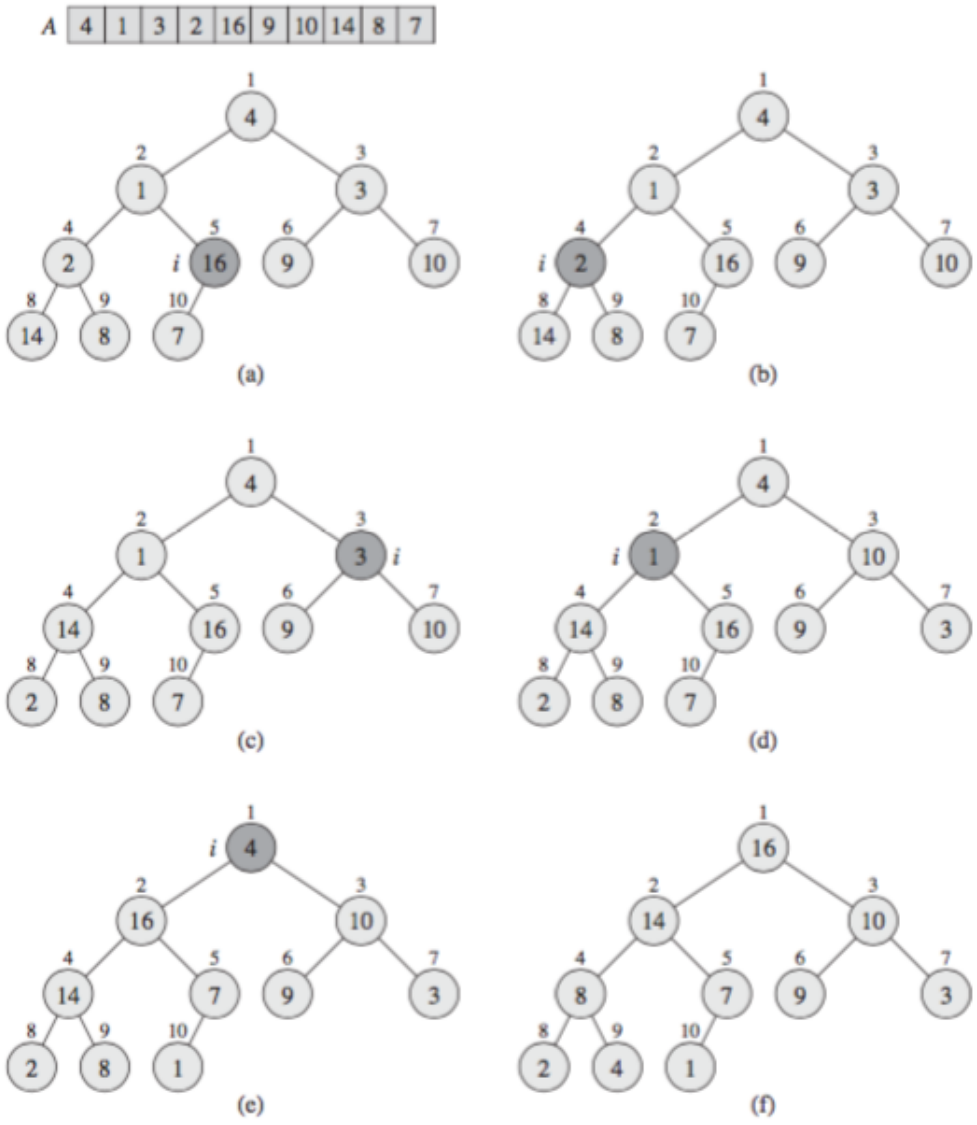

Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

1.3.1 Correctness

Loop invariant: At the start of each iteration of the for loop, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.

(Initialization) Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node past that is a leaf and is thus the root of a trivial max-heap.

(Maintenance) To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the inductive hypothesis, they are both roots of max-heaps. This is precisely the condition required for MaxHeapify to make node i the root of a max heap. Moreover, the MaxHeapify call preserves the property that nodes $i+1, i+2, \ldots, n$ are all roots of max-heaps. Decrementing i in the for loop update reestablishes the loop invariant for the next iteration.

(Termination) At termination, $i = 0$. By the loop invariant, each node $1, \ldots, n$ is the root of a max heap. In particular, node 1 is.

1.3.2 Runtime

Basic upper bound: There are $O(n)$ calls to MaxHeapify, each of which takes $O(\log n)$ time. So the running time is $O(n \log n)$.

Better upper bound: Recall that an n-element heap has height $\lfloor \log n \rfloor$. In addition, an n element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of any height h (this is exercise 6.3-3 in the book, and we won't solve it here, but you can prove it by induction).

So the runtime is

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) \leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(2n) = O(n)$$

Proof of summation formula: Recall that the sum of an infinite geometric series is $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ when $|x| < 1$. Differentiating this formula, we get $\sum_{k=0}^{\infty} k x^{k-1} = \frac{1}{(1-x)^2}$, and multiplying by x, we get

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}.$$

Letting $x = 1/2$ makes

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1/2)^2} = 2.$$

1.4 Heapsort

Heapsort is a way of sorting arrays. First we use BuildMaxHeap to turn the array into a max-heap. Now we can extract the maximum (i.e. the root) from the heap, swapping it with the last element in the array and then shrinking the size of the heap so we never operate on the max element again. At this point, the heap property is violated because the root may be smaller than other elements, so we call MaxHeapify on the root, which restores the max heap property. Now we can keep repeating this until the whole array is sorted.

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Note that each call to MaxHeapify takes $O(\log n)$ because the height of the heap is $O(\log n)$, so heapsort takes $O(n \log n)$ overall.

1.5 Priority queues

Earlier we said that heaps could be used to implement priority queues. A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A max-priority queue supports the following operations:

- Insert(S, x) inserts the element x into the set S.
- Maximum(S) returns the element of S with the largest key.
- ExtractMax(S) removes and returns the element with the largest key.
- IncreaseKey(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

In addition, there are min-priority queues, which support the analogous operations Minimum, ExtractMin, and DecreaseKey.

A max-priority queue can be used to schedule jobs on a shared computer, where each job has a priority level. Every time a job is finished, we pick the highest-priority job to run next, and we do this using ExtractMax. Furthermore, we can add new jobs to the queue by calling Insert.

In addition, min-priority queues are useful in Dijkstra's algorithm for finding shortest paths in graphs, which we will see later in class.

1.5.1 Implementation using heaps

To implement Maximum(A), we simply return $A[1]$.

To implement ExtractMax, we do the same thing we did in heapsort, where we swapped the first element with the last element of the array, decreased the size of the heap by 1 and then called MaxHeapify.

To implement IncreaseKey, we increase the key of the node, then "push" the node up through the heap until it is greater than all its children. We do this by repeatedly swapping the node with its parent.

To implement Insert, we expand the heap by one element, setting the temporary key of that element to $-\infty$, and then call IncreaseKey.

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2       error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

2 If we have extra time

2.1 Change of variables (substitution method)

$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log n$.

For now don't worry about rounding off values to be integers.

Let $m = \log n$, i.e. $n = 2^m$. Then $T(2^m) = 2T(2^{m/2}) + m$.

Now let $S(m) = T(2^m)$. Then $S(m) = 2S(m/2) + m$.

This recurrence has the solution $S(m) = O(m \log m)$.

So $T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n)$.