ICS 311 #09: Heaps, Heapsort, and Priority Queues

Outline

1. Heaps and their Properties
2. Building and Maintaining Heaps
3. Application to Sorting
4. Application to Priority Queues

Readings and Screencasts

- CLRS 3rd Ed. Chapter 6 (all)
- Screencasts 9A, 9B, 9C, 9D (also in Laulima and iTunesU)

Heaps and their Properties

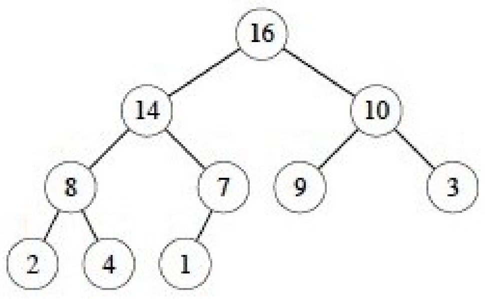
Heaps are a useful data structure with applications to sorting and priority queues.

They are *nearly complete binary trees* that satisfy a *heap property* that organizes data under a partial ordering of their keys, enabling access to elements with maximum (or minimum) keys without having to pay the cost of fully sorting the keys.

Heaps are not to be confused with garbage collected storage (a heap of garbage)!

Heaps as Nearly Complete Binary Trees

Conceptually, heaps are **nearly complete binary trees**: they have leaves on at most two adjacent levels *l-1* and *l* and in which the leaves at the bottommost level *l* lie in the leftmost positions of *l*:



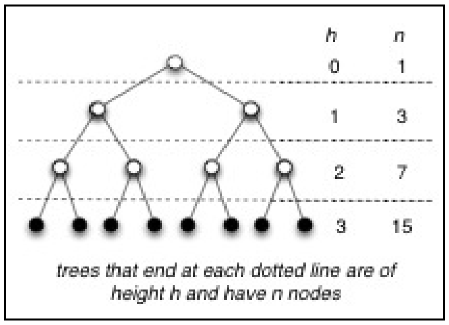
These quantitative properties concerning full and nearly complete binary trees will be useful:

Number of elements in nearly complete binary trees of height *h* (6.1-1)

As discussed in Topic 8, a **complete binary tree** has at most $2^{h+1} - 1$ nodes (vertices). We can see this by adding up the number of elements at each level: $2^0 + 2^1 + ... + 2^h$ for a complete binary tree of height h.

Then apply formula A.5 with x=2 and n=h:

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$


h
n
0
1
1
3
2
7
3
15
trees that end at each dotted line are of height h and have n nodes

You get $(2^{h+1} - 1) / (2 - 1) = 2^{h+1} - 1$.

So, a *nearly* complete binary tree has *at most* $2^{h+1} - 1$ elements (if it is complete, as analyzed above). The *fewest* number of elements it can have at height h is when the last level has just 1 element and the level before it is complete. So do the math for a complete binary tree of height h−1: there are exactly $2^h - 1$ elements in levels 1 to l−1 and one more element in the lth level, for a total of 2^h elements.

Height of an *n*-element nearly complete binary tree (6.1-2)

Given an n-element nearly complete binary tree of height h, from 6.1-1:

$$2^h \le n \le 2^{h+1} - 1 < 2^{h+1}$$

Taking the log of the first, second and last terms,

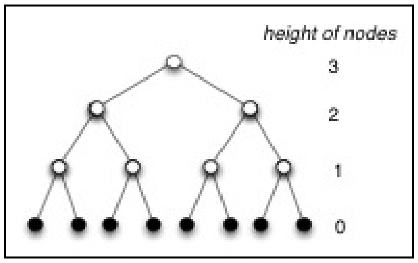
$$h \le \lg n < h + 1$$

Since h is an integer, $h = \lfloor \lg n \rfloor$ *(Notice the "floor" notation.)*

Number of leaves

An n-element nearly complete binary tree has $\lceil n/2 \rceil$ leaves. *(Notice the "ceiling" notation. Left as exercise.)*

Nodes of height *h* in a nearly complete binary tree (6.3-3)



The **height of a node** is the number of edges on the longest downward path from the node to a leaf.

There are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in a nearly complete binary tree. (A proof by contradiction is possible.) For example, in the tree shown there are $\lceil 15/2^{2+1} \rceil = \lceil 15/8 \rceil = 2$ nodes of height 2.

The Heap Property

Depending on whether it is a *max heap* or a *min heap*, to be a heap the binary tree must also satisfy a heap property:

Max Heap Property:

For all nodes i, excluding the root, key(parent(i)) ≥ key(i).

By induction and transitivity of ≥, the max heap property guarantees that the maximum element of a max-heap is at the root.

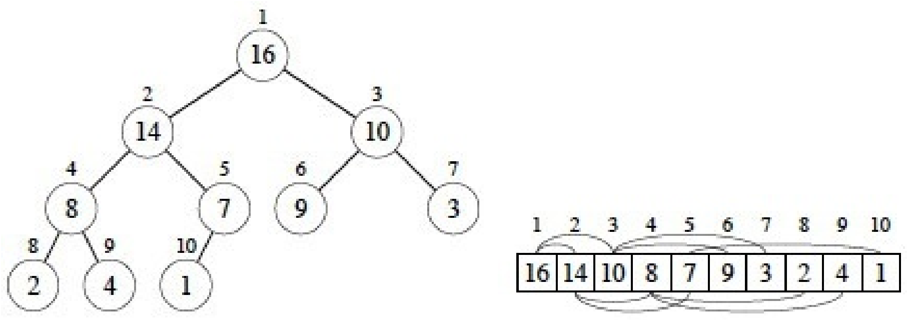
Min Heap Property:

For all nodes *i*, excluding the root, key(parent(*i*)) ≤ key(*i*).

By induction and transitivity of ≤, the min heap property guarantees that the minimum element of a min heap is at the root.

Array Representation

Heaps are usually represented using arrays, following the mapping shown by the indices in the tree:



The fact that we can see a heap both as a binary tree and as an array is an example of a powerful idea in computer science: mapping between an implementation representation that has efficient computational properties and a conceptual representation that fits how we think about a problem.

If a heap is stored in array `A`, then movement in the tree is easy:

- Root of the tree is `A[1]`
- Parent of `A[i]` is `A[⌊i/2⌋]` *(Notice we are taking the floor of i/2).*
- Left Child of `A[i]` is `A[2i]`
- Right Child of `A[i]` is `A[2i+1]`
- Index operations are fast in binary (left and right shifts and setting the low order bit).

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

Indices of leaves (6.1-7)

By the number of leaves fact, when an *n*-element heap is stored in the array representation, the leaves are the nodes indexed by ⌊n/2⌋ + 1, ⌊n/2⌋ + 2, ..., *n*. (Left as exercise.)

This fact will be used in algorithms that only need to process either the leaves or the internal nodes of a heap.

Building and Maintaining Heaps

Maintaining the Heap Property

MAX-HEAPIFY is used to maintain the max-heap property by addressing a possible violation at node `A[i]`:

- MAX-HEAPIFY assumes that the left and right subtrees of *i* are max-heaps.
- When called, `A[i]` may (or may not) be smaller than its children, violating the max-heap property

if it is.

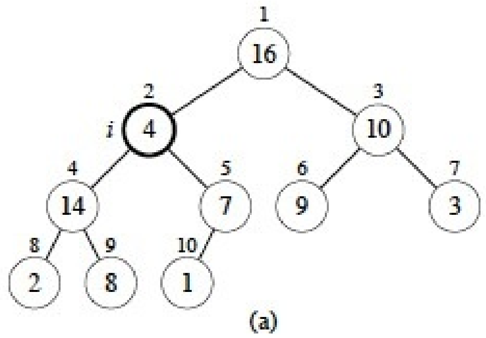
- After MAX-HEAPIFY, the subtree rooted at *i* will be a heap.

```
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)
```

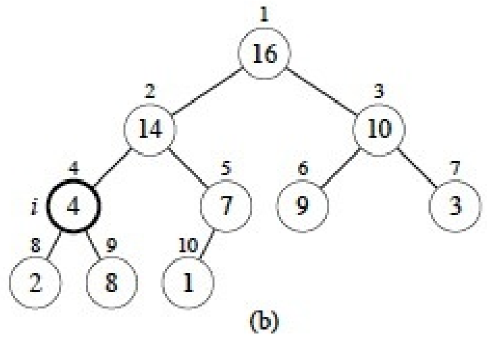
It works by comparing `A[i]` with its left and right children (lines 3-7), and if necessary swapping `A[i]` with the larger of the two children to preserve the heap property (lines 8-9). *Tail recursion* after the swap propagates this change until the subtree is a heap (line 10).

Example

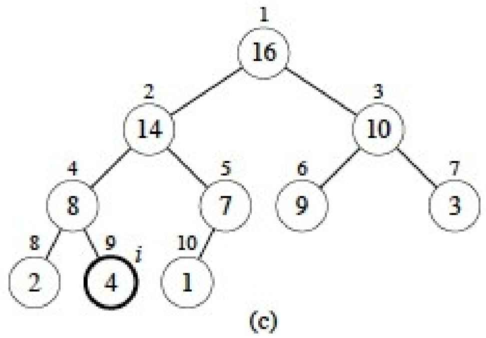
Max-Heapify from the node at index 2 (containing 4):



(a)



(b)



(c)

Analysis

It is easy to see that the body of each call before recursion is O(1), and the recursion repeats this for at most O(lg *n*) nodes on the path from the root to the leaves.

More formally, the worst case is when the bottom level is exactly half full, and in this case, the *children's subtrees* can have at most 2*n*/3 nodes. So, adding the cost to recurse on these subtrees plus Θ(1) cost for comparisons at a given node, we get the recurrence relation:

$$T(n) \le T(2n/3) + \Theta(1).$$

This fits case 2 of the Master Theorem ($a = 1$, $b = 3/2$ since $1/(3/2) = 2/3$, and $f(n) = 1 = O(n^{\log_{3/2} 1}) = O(n^0)$), giving $\Theta(\lg n)$.

Building a Heap

Suppose we load some keys into an array in arbitrary order from left to right, creating an almost complete binary tree that may not satisfy the heap property.

Each leaf of the corresponding tree is trivially a heap. If we call MAX-HEAPIFY on the parents of the leaves, the assumption that the right and left subtrees are heaps is met. Once MAX-HEAPIFY returns, the parents are roots of heaps too, so we call it on *their* parents.

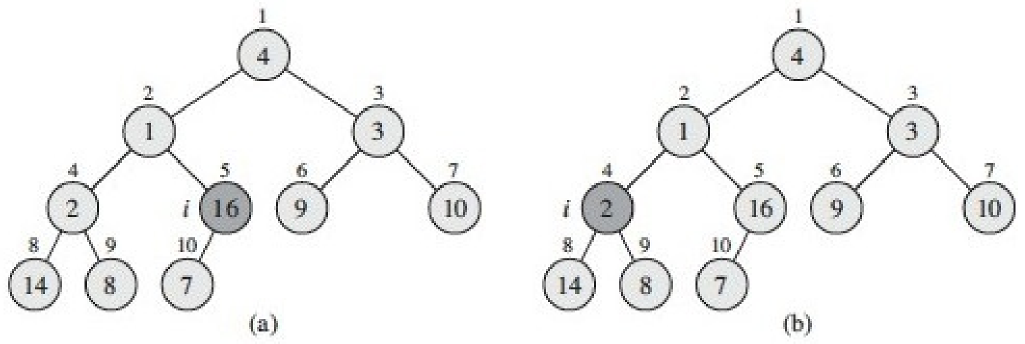
Using the previously established result that the leaves begin in the array at index $\lfloor n/2 \rfloor + 1$, so the last non-leaf node is at $\lfloor n/2 \rfloor$, the implementation is trivial:

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

Example

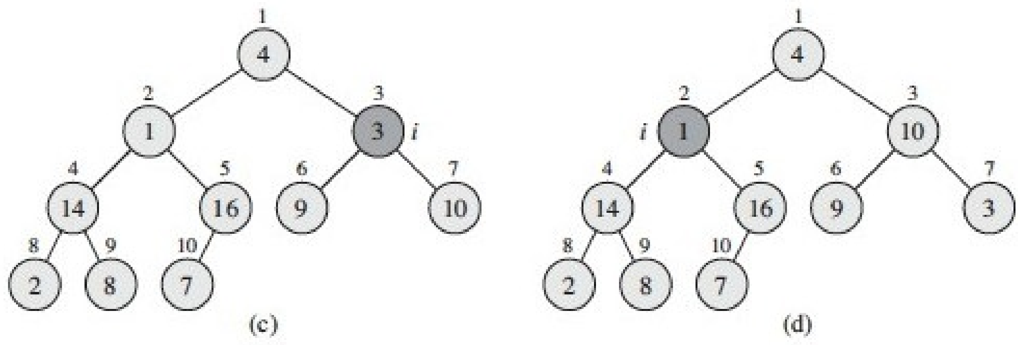
Let's trace this on an array of size 10, for i = 5 downto 1:

A	4	1	3	2	16	9	10	14	8	7



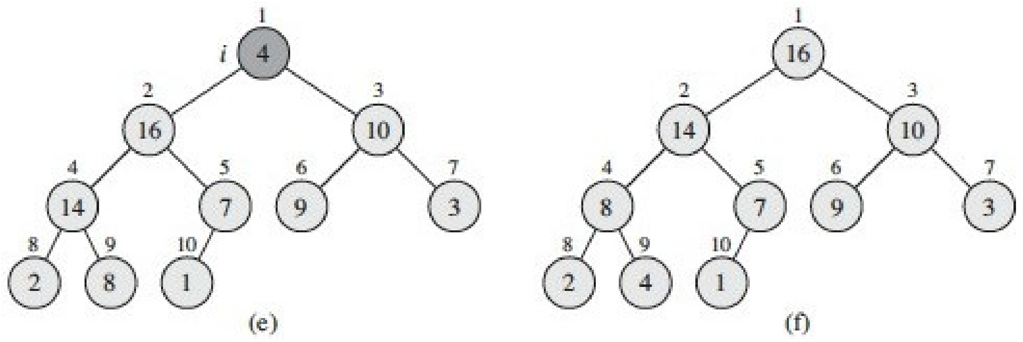
(a) The heap rooted at vertex or array index 5 is already a max heap: no change is made.

(b) The heap rooted at index 4 is not a max heap: the value 2 is smaller than its children. We restore the max heap property by swapping 2 with the larger child key, 14 (see next figure for result). If we had swapped with 8, it would not be a max heap: this is why we always swap with the larger child.



(c) Decrementing *i* to 3, there is another violation of the max heap property, and we swap value 3 at index 3 with value 10 at index 7 (the larger child).

(d) The heap at index 2 violates the max heap property: we must propagate the value 1 down by swapping with 16, and then with 7 in a recursive call to Max-Heapify (see next figure).



(e) Finally, checking the value at index 1 (value 4) against its children, we find we need to swap it with value 16 at index 2, and then with value 14 at index 4 and value 8 at index 9 in two recursive calls to Max-Heapify. (f) shows the resulting max heap.

Correctness

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

Loop Invariant:

At the start of every iteration of the `for` loop, each node *i*+1, *i*+2, ..., *n* is a root of a max heap.

Initialization:

By Exercise 6.1-7, each node $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, ..., *n* is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the `for` loop, the invariant is initially true.

Maintenance:

Children of node *i* are indexed higher than *i*, so by the loop invariant, they are both roots of max-heaps. Thus the assumption of MAX-HEAPIFY is met, enabling it to make node *i* a max-heap root. Decrementing *i* reestablishes the loop invariant at each iteration.

Termination:

The loop terminates when *i* = 0. By the loop invariant, each node including the root indexed by 1 is the root of a max-heap.

Analysis

Sometimes a good approach is to prove an easy bound and then tighten it.

It is easy to see that there are O(*n*) (about *n*/2) calls to MAX-HEAPIFY, and we already know that MAX-HEAPIFY on a tree of height O(lg *n*) is O(lg *n*). Therefore an upper bound is O(*n* lg *n*).

However, only the root node and those near it are at height O(lg *n*). Many nodes are close to the leaves and we don't even process half of them. So let's try for a tighter bound ...

There are no more than $\lceil n/2^{h+1} \rceil$ nodes of height *h* (Exercise 6.3-3), and the heap is $\lfloor \lg n \rfloor$ high (Exercise 6.1-2). MAX-HEAPIFY called on a node of height *h* is O(*h*), so we need to sum this cost times the number of nodes at each *h* for all relevant *h*:

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

We can simplify this as follows:

1. Wrap big-O around the whole thing, leaving h behind.
2. Remove the ceiling (which does not affect big-O analysis).
3. Rewrite the resulting $nh/2^{h+1}$ as $(n/2)(h/2^h)$.
4. Move $n/2$ out of the summation, as it does not involve h.
5. Eliminate the constant 1/2, as we are inside the magical world of big-O!

Tricky, huh? Now maybe you can see why the text authors write that as:

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

The above summation runs up to $\lfloor \lg n \rfloor$, but we would like to use a convenient formula A-8, shown below, which runs up to ∞:

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

for $|x| < 1$.

Since big-O implies an inequality (upper bound), we can go ahead and run the summation to ∞ instead of $\lfloor \lg n \rfloor$, because all of the additional terms are positive (and also very small), so the inequality will be maintained. Then, if we let $x = 1/2$ (since $h/2^h = h(1^h/2^h)$ can be written as $h(1/2)^h$), we get:

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

Thus our big-O expression simplifies to O(n*2) = O(n), which is a tighter bound than O(n lg n). The same analysis appliles to the min-heap version.

(You might wonder why we can build a heap in O(n) time when sorting takes O(n lg n), as will be proven in Topic 10. This is because a heap is only a partial order, so less work needs to be done to guarantee the heap property.)

Application to Sorting

Suppose `A[1..n]` contains keys to be sorted. If we call BUILD-MAX-HEAP on this array, the maximum element will be at `A[1]`. We can swap it with the item at `A[n]`, then repeat on `A[1..n-1]` (reducing the size of the heap by 1 each iteration) until this reaches size 1.

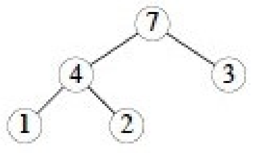
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for $i = A.length$ downto 2
3      exchange $A[1]$ with $A[i]$
4      $A.heap\text{-}size = A.heap\text{-}size - 1$
5      MAX-HEAPIFY(A, 1)
```

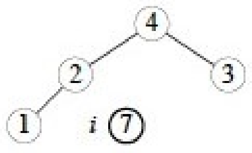
Analysis:

BUILD-MAX-HEAP is O(*n*) (by analysis above). The `for` loop executes *n*-1 times, with O(1) exchange each iteration and a call to O(lg *n*) MAX-HEAPIFY. Thus heapsort is O(*n* lg *n*).

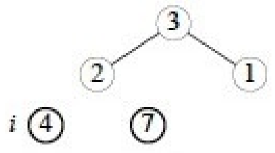
Example:

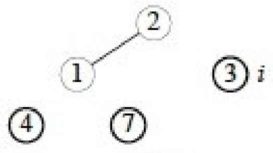
Suppose we have an array A with five integers. First, BUILD-MAX-HEAP is called on it, resulting in the array A = [7, 4, 3, 1, 2] shown as the tree in (a) below.


(a)

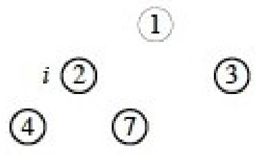

(b)

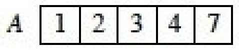
Then the loop of HEAPSORT successively takes out the maximum from the first index by swapping it with the last element in the heap, and calls MAX-HEAPIFY. So, 7 is swapped with 2, and then the heap (now one smaller) is reconstructed, resulting in the heap shown in (b): A = [4, 2, 3, 1, 7], with the first four elements being the heap.


(c)


(d)

The maximum element 4 (from b) was swapped with the minimum element 1 (removing 4 from the heap) and the heap restored, resulting in (c) A = [3, 2, 1, 4, 7] with the first three elements being the heap. Then in (d), the max element 3 was swapped with 1 and the heap restored by percolating 1 down: A = [2, 1, 3, 4, 7] with the heap being the first two elements.


(e)



(e) Finally, the maximum element 2 is removed by swapping with the only remaining element 1, resulting in the sorted array shown.

Here is a playing card demonstration of heap sort, in case it helps. This demonstration is using a ***min-heap*** to sort the cards with the card of ***maximum*** value ending up at the top of the stack of cards.

Application to Priority Queues

An important application of heaps is implementing **priority queues**. There are *min* and *max* versions.

A **max-priority queue** is an ADT with the following operations:

INSERT(S,*x*)
    $S \leftarrow S \cup \{x\}$

MAXIMUM(S)
    Returns the element of S with the largest key.

EXTRACT-MAX(S)
    Removes and returns the element of S with the largest key.

INCREASE-KEY(S,*x*,*k*)
    Increases the value of *x*'s key to the new value $k \geq$ current key(x).

A **min-priority queue** has corresponding operations MINIMUM, EXTRACT-MIN, and DECREASE-KEY.

Max-priority queues can be used in job scheduling: the highest priority job is always run next, but job priority can be increased as a job ages in the queue, or for other reasons.

Min-priority queues will be very important in graph algorithms we cover later in the semester: efficient implementations of EXTRACT-MIN and DECREASE-KEY will be especially important.

Min-priority queues also used in event-driven simulations, where an event may generate future events, and we need to simulate the events in chronological order.

Accessing Maximums

In the array representation, MAXIMUM is trival to implement in O(1) by returning the first element of the array. However, if we EXTRACT-MAX we need to restore the heap property afterwards.

HEAP-EXTRACT-MAX takes the root out, replaces it with the last element in the heap *(stop and think: why this element?)*, and then uses MAX-HEAPIFY to propagate that element (which probably has a small key) down to its proper place:

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

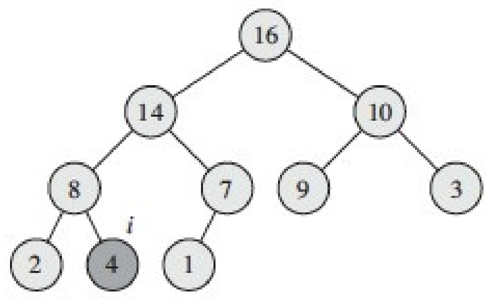
HEAP-EXTRACT-MAX is O(lg *n*) since there is only constant work added to the O(lg *n*) MAX-HEAPIFY.

Increasing keys

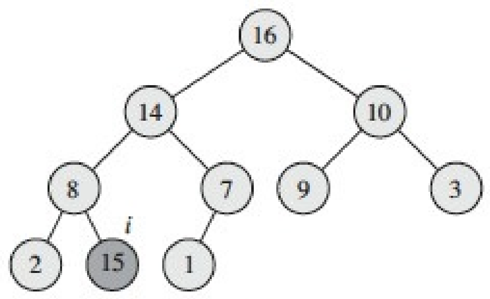
An increase to the key may require propagating the element *up* the tree (the opposite direction as compared to MAX-HEAPIFY):

```
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)
```

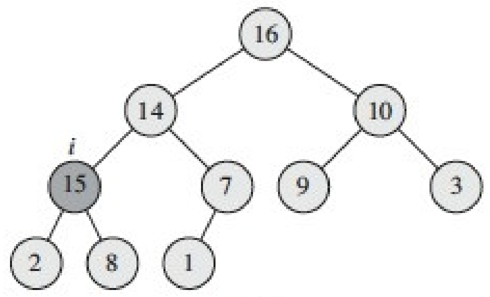
This is clearly O(lg *n*) due to following a simple path up the tree. Let's work this example, where the element at *i* has its key increased from 4 to 15, and then it is propagated up:



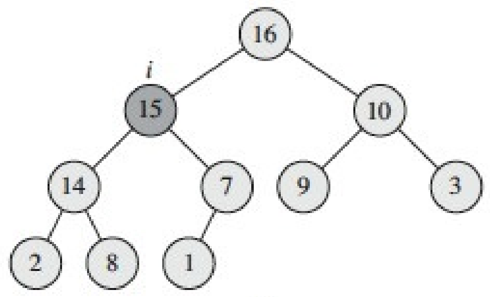
(a)



(b)



(c)



(d)

This propagation follows the "Peter Principle": the claim that persons in a hierarchical organization are promoted through the ranks of management until they reach their level of incompetency!!!

Inserting New Elements

When inserting, we are going to have to make the heap bigger, so let's add the element at the end and propagate it up to where it belongs.

HEAP-INCREASE-KEY already has the code for this propagation, so if we set the key to the smallest possible value and then try to increase it with HEAP-INCREASE-KEY, it will end up in the right place:

```
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)
```

Again, this is O(lg *n*).

Next

In Topic 10 we wrap up our examination of sort algorithms with Quicksort, a practical sort that performs well in practice and also illustrates the value of probabilistic analysis and random algorithms.

We will return to other kinds of trees, in particular special kinds of binary search trees that are kept balanced to guarantee O(lg *n*) performance, in Topic 11.

Dan Suthers
Last modified: Mon Sep 14 21:55:10 HST 2015
Images are from the instructor's material for Cormen et al. Introduction to Algorithms, Third Edition.