Algorithms

Solving Recurrences Continued

The Master Theorem

Introduction to heapsort

Review: Merge Sort

```
MergeSort(A, left, right) {
   if (left < right) {
      mid = floor((left + right) / 2);
      MergeSort(A, left, mid);
      MergeSort(A, mid+1, right);
      Merge(A, left, mid, right);
   }
}
// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A.
// Code for this is in the book.  It requires O(n)
// time, and *does* require allocating O(n) space
```

Review: Analysis of Merge Sort

Statement	Effort
`MergeSort(A, left, right) {`	T(n)
`    if (left < right) {`	Θ(1)
`        mid = floor((left + right) / 2);`	Θ(1)
`        MergeSort(A, left, mid);`	T(n/2)
`        MergeSort(A, mid+1, right);`	T(n/2)
`        Merge(A, left, mid, right);`	Θ(n)
`    }`	
`}`	

- So $T(n) = \quad \Theta(1)$ when $n = 1$, and

  $2T(n/2) + \Theta(n)$ when $n > 1$
- This expression is a *recurrence*

Review: Solving Recurrences

- Substitution method
- Iteration method
- Master method

Review: Solving Recurrences

- The substitution method
  - A.k.a. the "making a good guess method"
  - Guess the form of the answer, then use induction to find the constants and show that solution works
  - *Run an example*: merge sort
    - $T(n) = 2T(n/2) + cn$
    - We guess that the answer is $O(n \lg n)$
    - Prove it by induction
  - Can similarly show $T(n) = \Omega(n \lg n)$, thus $\Theta(n \lg n)$

Review: Solving Recurrences

- The "iteration method"
  - Expand the recurrence
  - Work some algebra to express as a summation
  - Evaluate the summation
- We showed several examples, were in the middle of:

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\dfrac{n}{b}\right) + cn & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- $T(n) =$

  $aT(n/b) + cn$

  $a(aT(n/b/b) + cn/b) + cn$

  $a^2T(n/b^2) + cna/b + cn$

  $a^2T(n/b^2) + cn(a/b + 1)$

  $a^2(aT(n/b^2/b) + cn/b^2) + cn(a/b + 1)$

  $a^3T(n/b^3) + cn(a^2/b^2) + cn(a/b + 1)$

  $a^3T(n/b^3) + cn(a^2/b^2 + a/b + 1)$

  …

  $a^kT(n/b^k) + cn(a^{k-1}/b^{k-1} + a^{k-2}/b^{k-2} + \ldots + a^2/b^2 + a/b + 1)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So we have
  - $T(n) = a^k T(n/b^k) + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$
- For $k = \log_b n$
  - $n = b^k$
  - $T(n) = a^k T(1) + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$

    $= a^k c + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$

    $= ca^k + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$

    $= cna^k / b^k + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$

    $= cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a = b$?
  - $T(n) = cn(k + 1)$

    $= cn(\log_b n + 1)$

    $= \Theta(n \log n)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a < b$?

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a < b$?
  - Recall that $\Sigma(x^k + x^{k-1} + \ldots + x + 1) = (x^{k+1} - 1)/(x-1)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a < b$?
  - Recall that $(x^k + x^{k-1} + \ldots + x + 1) = (x^{k+1} - 1)/(x - 1)$
  - So:

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 = \frac{(a/b)^{k+1} - 1}{(a/b) - 1} = \frac{1 - (a/b)^{k+1}}{1 - (a/b)} < \frac{1}{1 - a/b}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a < b$?
  - Recall that $\Sigma(x^k + x^{k-1} + \ldots + x + 1) = (x^{k+1} - 1)/(x-1)$
  - So:

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 = \frac{(a/b)^{k+1} - 1}{(a/b) - 1} = \frac{1 - (a/b)^{k+1}}{1 - (a/b)} < \frac{1}{1 - a/b}$$

  - $T(n) = cn \cdot \Theta(1) = \Theta(n)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a > b$?

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if a > b?

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 \quad = \quad \frac{(a/b)^{k+1} - 1}{(a/b) - 1} \quad = \quad \Theta\left((a/b)^k\right)$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if a > b?

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 \quad = \quad \frac{(a/b)^{k+1} - 1}{(a/b) - 1} \quad = \quad \Theta\left((a/b)^k\right)$$

  - $T(n) = cn \cdot \Theta(a^k / b^k)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if a > b?

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 = \frac{(a/b)^{k+1} - 1}{(a/b) - 1} = \Theta\left((a/b)^k\right)$$

  - $T(n) = cn \cdot \Theta(a^k / b^k)$

$$= cn \cdot \Theta(a^{\log n} / b^{\log n}) = cn \cdot \Theta(a^{\log n} / n)$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a > b$?

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 \quad = \quad \frac{(a/b)^{k+1} - 1}{(a/b) - 1} \quad = \quad \Theta\left((a/b)^k\right)$$

  - $T(n) = cn \cdot \Theta(a^k / b^k)$

$$= cn \cdot \Theta(a^{\log n} / b^{\log n}) = cn \cdot \Theta(a^{\log n} / n)$$

recall logarithm fact: $a^{\log n} = n^{\log a}$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a > b$?

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 \quad = \quad \frac{(a/b)^{k+1} - 1}{(a/b) - 1} \quad = \quad \Theta\left((a/b)^k\right)$$

  - $T(n) = cn \cdot \Theta(a^k / b^k)$

$$= cn \cdot \Theta(a^{\log n} / b^{\log n}) = cn \cdot \Theta(a^{\log n} / n)$$

recall logarithm fact: $a^{\log n} = n^{\log a}$

$$= cn \cdot \Theta(n^{\log a} / n) = \Theta(cn \cdot n^{\log a} / n)$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a > b$?

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 = \frac{(a/b)^{k+1} - 1}{(a/b) - 1} = \Theta\left((a/b)^k\right)$$

  - $T(n) = cn \cdot \Theta(a^k / b^k)$

$$= cn \cdot \Theta(a^{\log n} / b^{\log n}) = cn \cdot \Theta(a^{\log n} / n)$$

recall logarithm fact: $a^{\log n} = n^{\log a}$

$$= cn \cdot \Theta(n^{\log a} / n) = \Theta(cn \cdot n^{\log a} / n)$$

$$= \Theta(n^{\log a})$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So…

$$T(n) = \begin{cases} \Theta(n) & a < b \\ \Theta(n \log_b n) & a = b \\ \Theta\left(n^{\log_b a}\right) & a > b \end{cases}$$

The Master Theorem

- Given: a *divide and conquer* algorithm
  - An algorithm that divides the problem of size n into a subproblems, each of size n/b
  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function $f(n)$
- Then, the Master Theorem gives us a cookbook for the algorithm’s running time:

The Master Theorem

- if T(n) = aT(n/b) + f(n) then

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{cases} \quad \begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}$$

Using The Master Method

- $T(n) = 9T(n/3) + n$
  - $a=9$, $b=3$, $f(n) = n$
  - $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$
  - Since $f(n) = O(n^{\log_3 9 - \varepsilon})$, where $\varepsilon=1$, case 1 applies:

$$T(n) = \Theta\left(n^{\log_b a}\right) \text{when } f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$

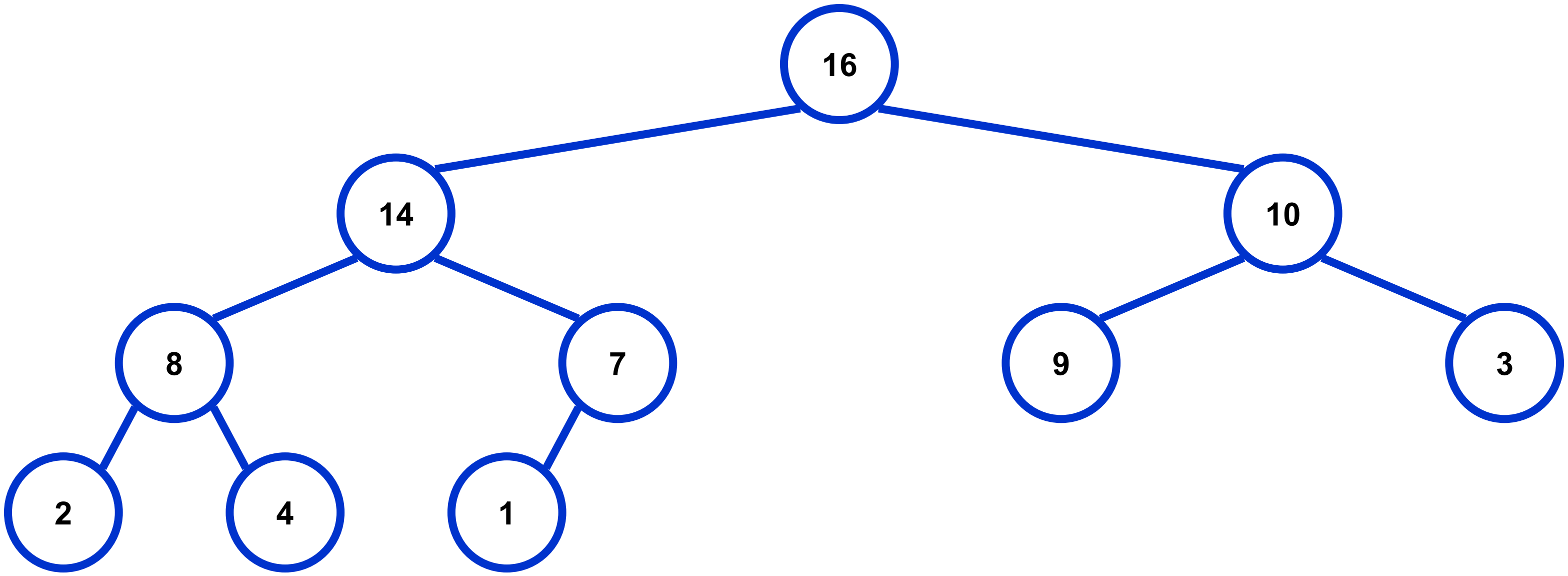
  - Thus the solution is $T(n) = \Theta(n^2)$

Sorting Revisited

- So far we've talked about two algorithms to sort an array of numbers
  - What is the advantage of merge sort?
  - What is the advantage of insertion sort?
- Next on the agenda: *Heapsort*
  - Combines advantages of both previous algorithms

Heaps

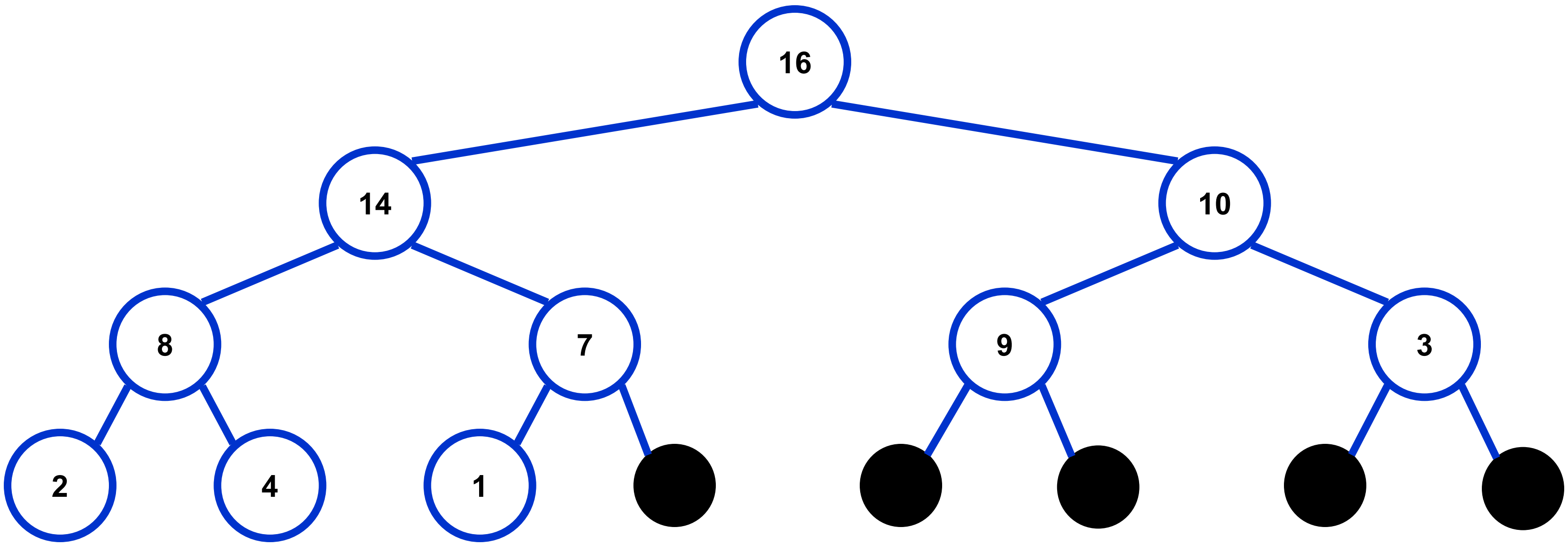
- A *heap* can be seen as a complete binary tree:



  - *What makes a binary tree complete?*
  - *Is the example above complete?*

Heaps

- A *heap* can be seen as a complete binary tree:



  - The book calls them "nearly complete" binary trees; can think of unfilled slots as null pointers

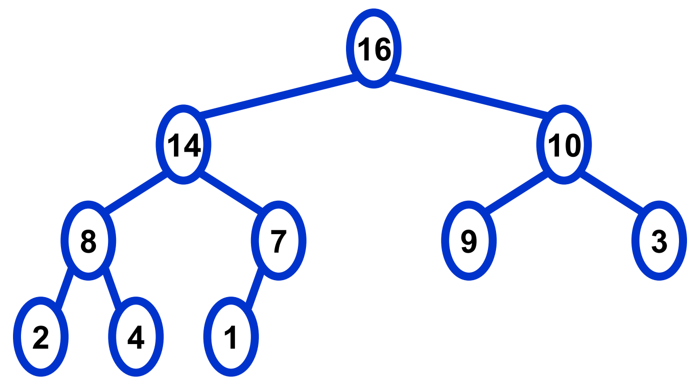
Heaps

- In practice, heaps are usually implemented as arrays:

A =

16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

=



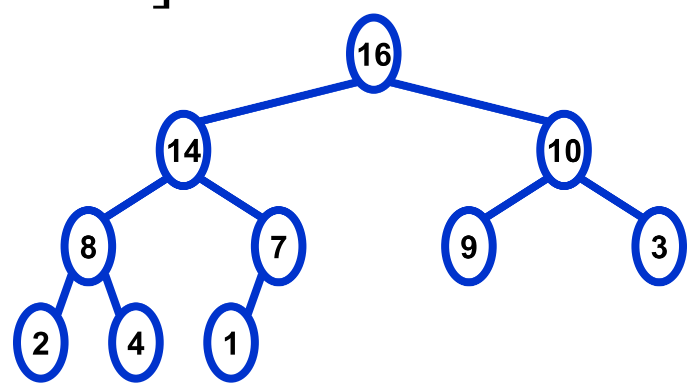
Heaps

- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node *i* is A[*i*]
  - The parent of node *i* is A[*i*/2] (note: integer divide)
  - The left child of node *i* is A[2*i*]
  - The right child of node *i* is A[2*i* + 1]

A =

16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

=



Referencing Heap Elements

- So…

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

- An aside: *How would you implement this most efficiently?*
- Another aside: *Really?*

The Heap Property

- Heaps also satisfy the *heap property*:

  A[*Parent*(i)] ≥ A[i]   for all nodes $i > 1$

  - In other words, the value of a node is at most the value of its parent
  - *Where is the largest element in a heap stored?*
- Definitions:
  - The *height* of a node in the tree = the number of edges on the longest downward path to a leaf
  - The height of a tree = the height of its root

Heap Height

- *What is the height of an n-element heap? Why?*
- This is nice: basic heap operations take at most time proportional to the height of the heap

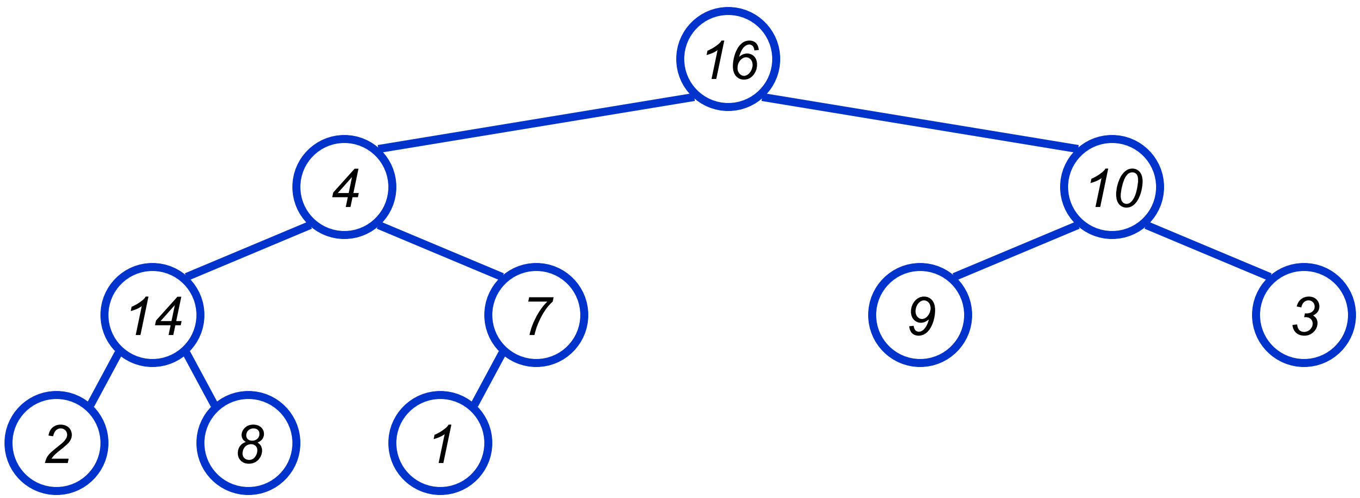
Heap Operations: Heapify()

- **`Heapify()`**: maintain the heap property
  - Given: a node *i* in the heap with children *l* and *r*
  - Given: two subtrees rooted at *l* and *r*, assumed to be heaps
  - Problem: The subtree rooted at *i* may violate the heap property (*How?*)
  - Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property
    - *What do you suppose will be the basic operation between i, l, and r?*

Heap Operations: Heapify()

```
Heapify(A, i)
{
   l = Left(i); r = Right(i);
   if (l <= heap_size(A) && A[l] > A[i])
        largest = l;
   else
        largest = i;
   if (r <= heap_size(A) && A[r] > A[largest])
        largest = r;
   if (largest != i)
        Swap(A, i, largest);
        Heapify(A, largest);
}
```

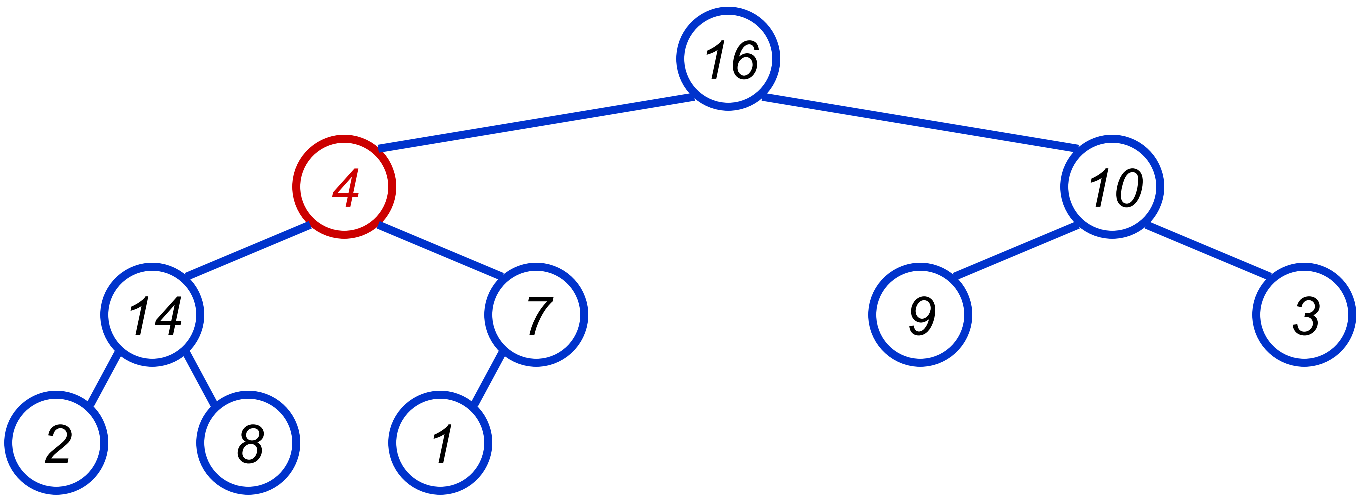
Heapify() Example



A =

16	*4*	*10*	*14*	*7*	*9*	*3*	*2*	*8*	*1*

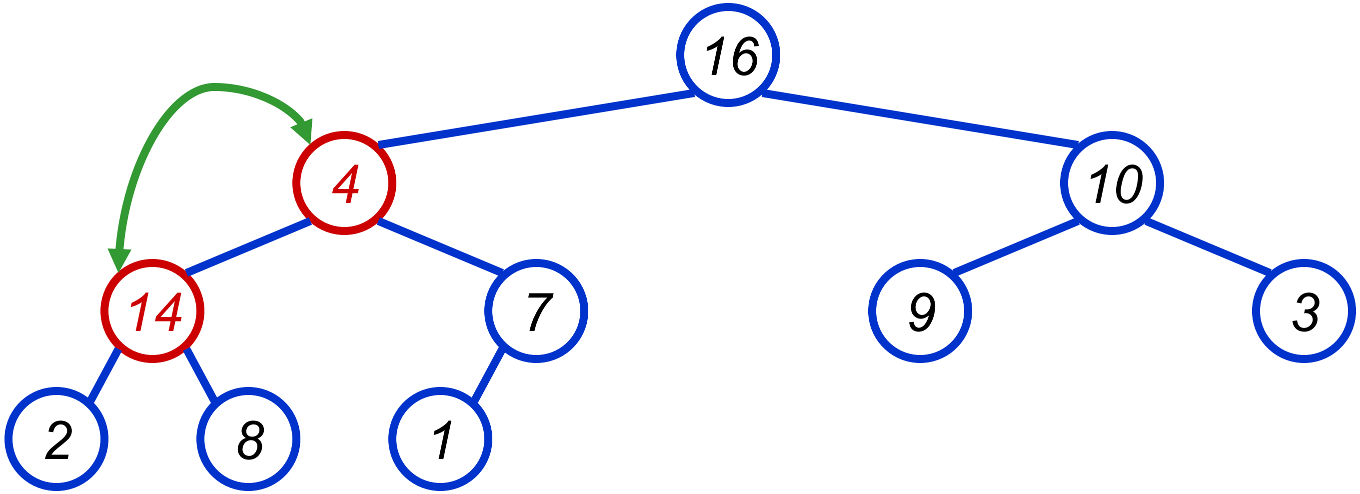
Heapify() Example

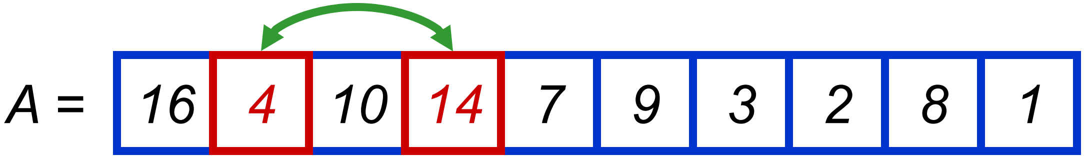


A =

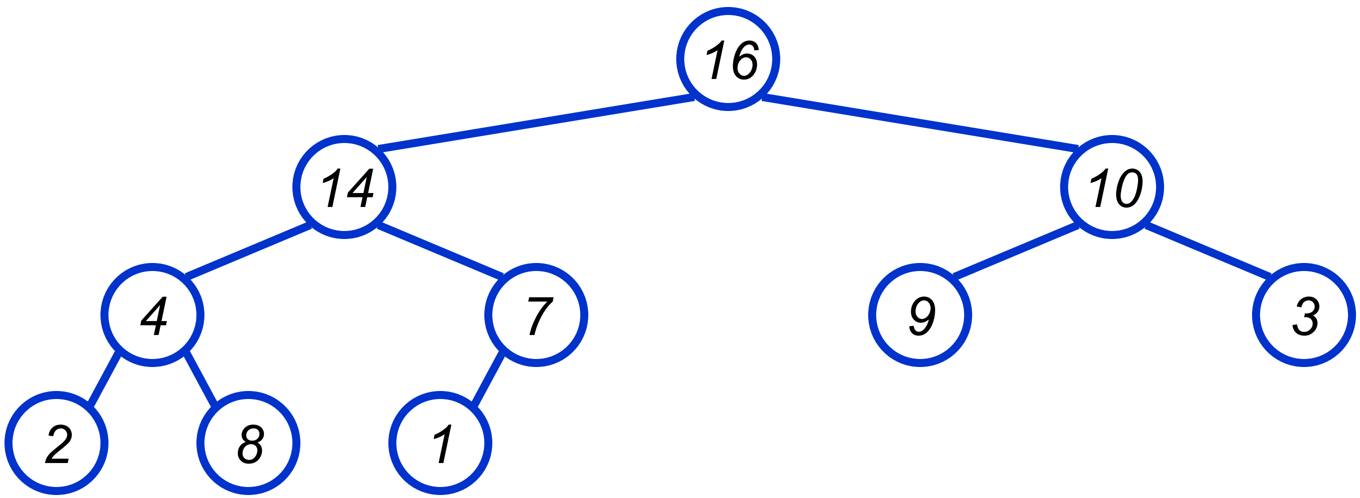
16	4	10	14	7	9	3	2	8	1

Heapify() Example





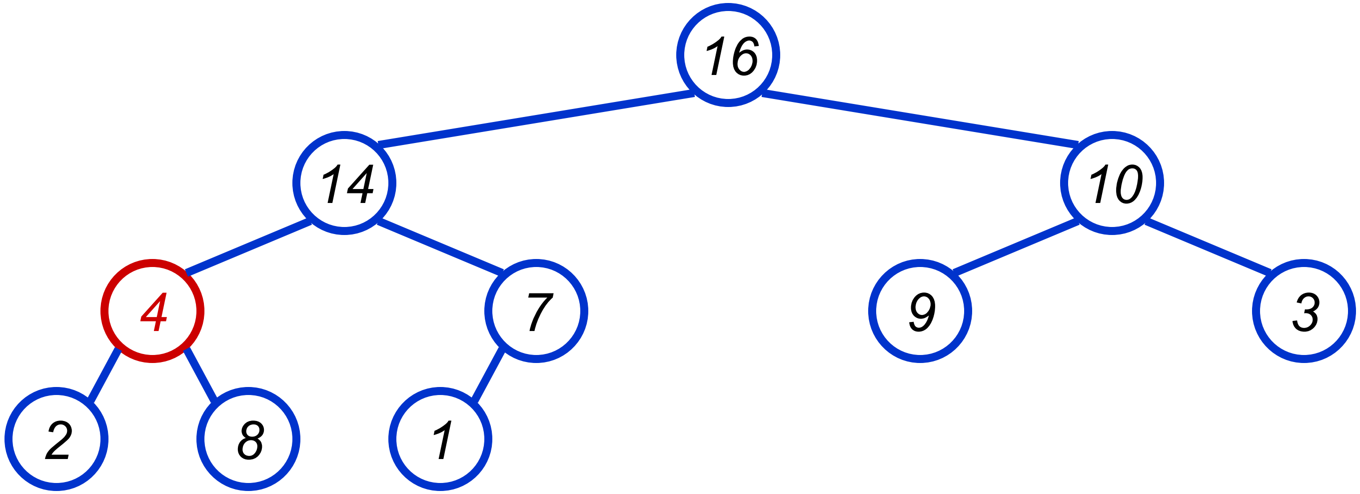
Heapify() Example



A =

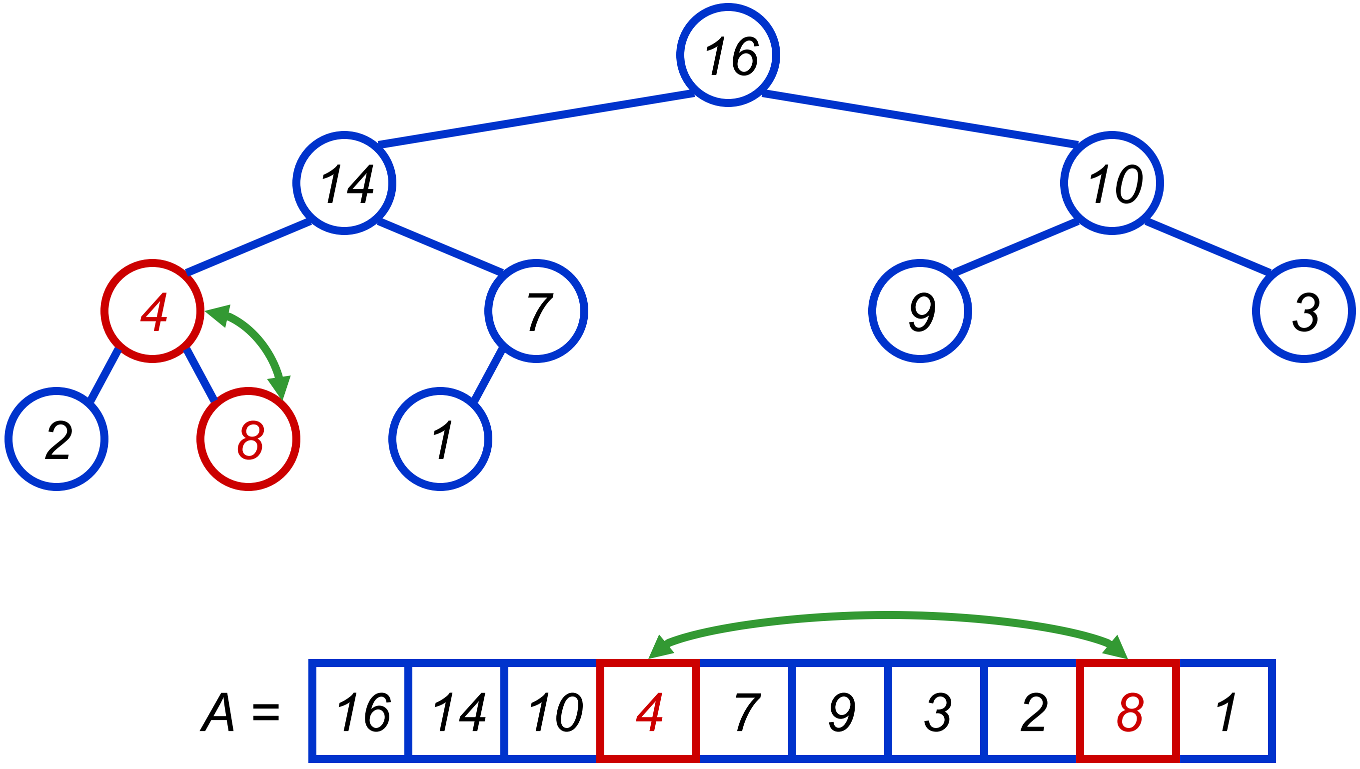
16	14	10	4	7	9	3	2	8	1

Heapify() Example

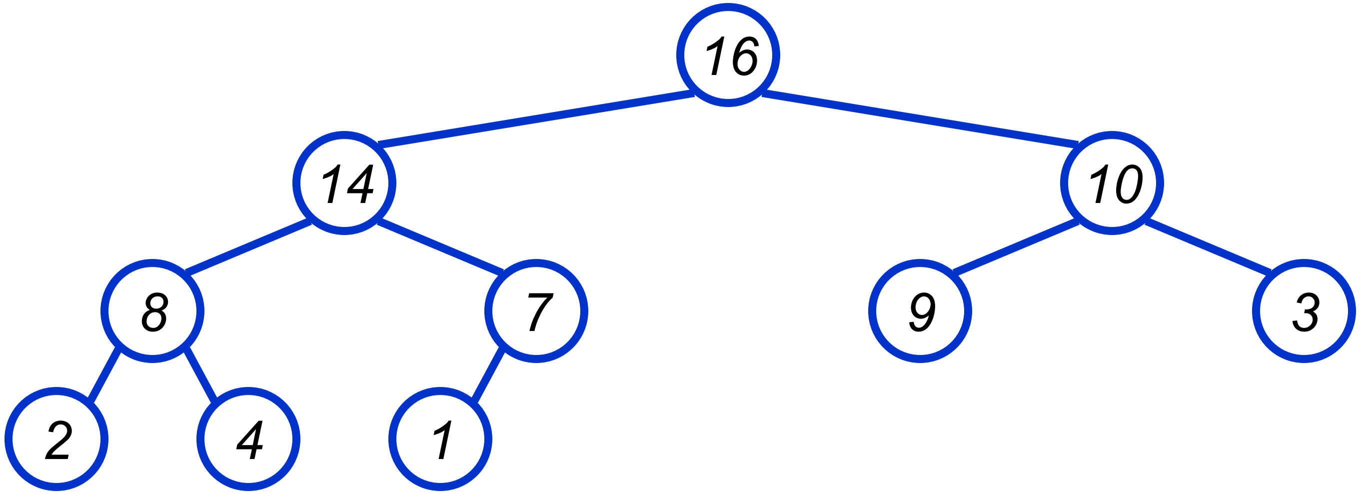


A =										
	16	14	10	4	7	9	3	2	8	1

Heapify() Example

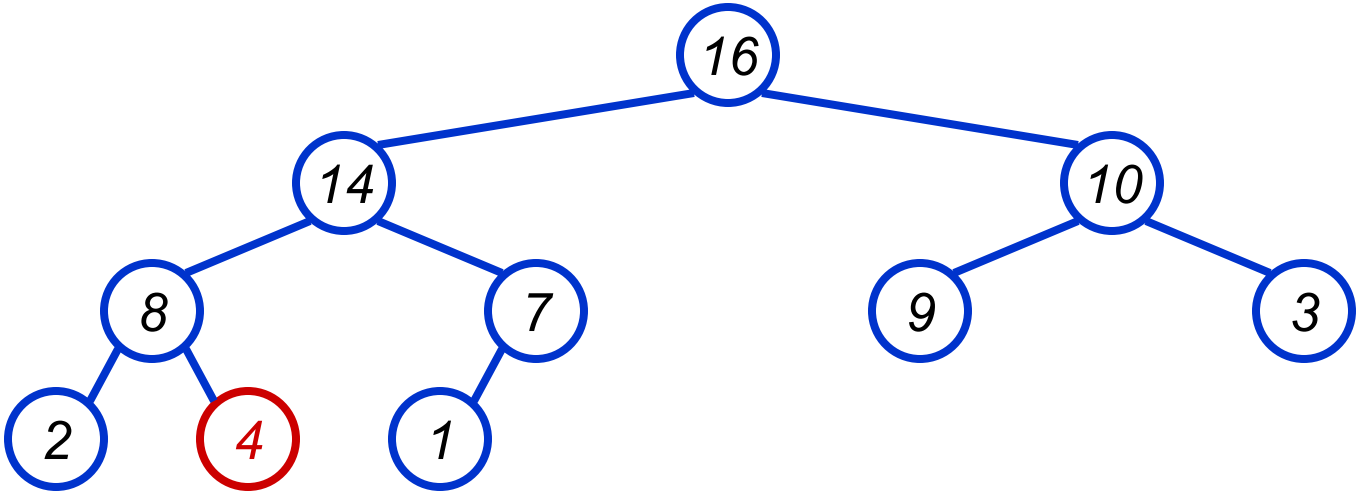


Heapify() Example



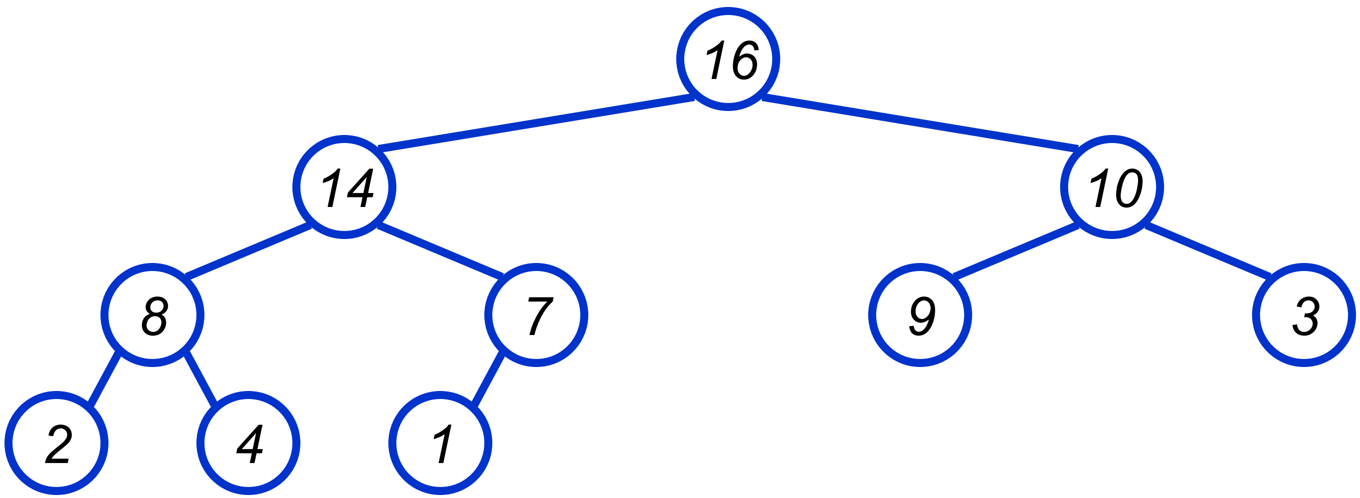
A =	16	14	10	8	7	9	3	2	4	1

Heapify() Example



A =	16	14	10	8	7	9	3	2	4	1

Heapify() Example



A =	16	14	10	8	7	9	3	2	4	1

Analyzing Heapify(): Informal

- *Aside from the recursive call, what is the running time of* **`Heapify()`***?*
- *How many times can* **`Heapify()`** *recursively call itself?*
- *What is the worst-case running time of* **`Heapify()`** *on a heap of size n?*

Analyzing Heapify(): Formal

- Fixing up relationships between i, l, and r takes $\Theta(1)$ time
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*
  - Draw it
- Answer: $2n/3$ (worst case: bottom row 1/2 full)
- So time taken by **`Heapify()`** is given by

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

Analyzing Heapify(): Formal

- So we have

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

- By case 2 of the Master Theorem,

$$T(n) = O(\lg n)$$

- Thus, **`Heapify()`** takes linear time

Heap Operations: BuildHeap()

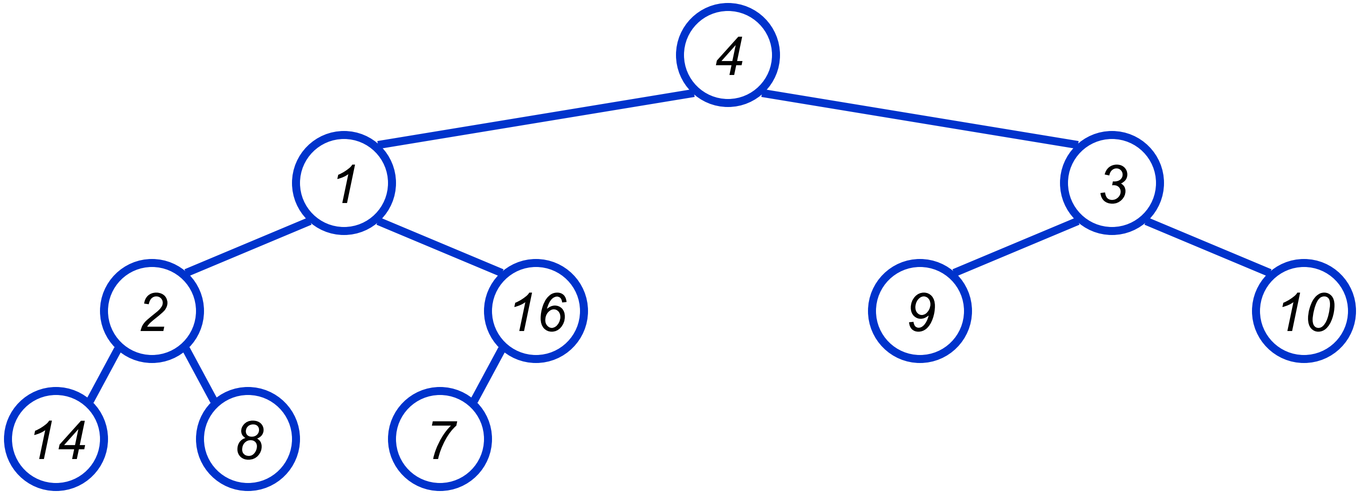
- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Fact: for array of length *n*, all elements in range A[$\lfloor n/2 \rfloor$ + 1 .. n] are heaps (*Why?*)
  - So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

BuildHeap()

```
// given an unsorted array A, make A a heap
BuildHeap(A)
{
  heap_size(A) = length(A);
  for (i = ⌊length[A]/2⌋ downto 1)
     Heapify(A, i);
}
```

BuildHeap() Example

- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



Analyzing BuildHeap()

- Each call to **`Heapify()`** takes O(lg *n*) time
- There are O(*n*) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(*n* lg *n*)
  - *Is this a correct asymptotic upper bound?*
  - *Is this an asymptotically tight bound?*
- A tighter bound is O(*n*)
  - *How can this be? Is there a flaw in the above reasoning?*

Analyzing BuildHeap(): Tight

- To **`Heapify()`** a subtree takes O(*h*) time where *h* is the height of the subtree
  - $h = O(\lg m)$, m = # nodes in subtree
  - The height of most subtrees is small
- Fact: an *n*-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height *h*
- CLR 7.3 uses this fact to prove that **`BuildHeap()`** takes O(*n*) time

Heapsort

- Given **`BuildHeap()`**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **`Heapify()`**
  - Repeat, always swapping A[1] for A[heap_size(A)]

Heapsort

```
Heapsort(A)
{
      BuildHeap(A);
      for (i = length(A) downto 2)
      {
            Swap(A[1], A[i]);
            heap_size(A) -= 1;
            Heapify(A, 1);
      }
}
```

Analyzing Heapsort

- The call to **`BuildHeap()`** takes $O(n)$ time
- Each of the $n - 1$ calls to **`Heapify()`** takes $O(\lg n)$ time
- Thus the total time taken by **`HeapSort()`**
  $= O(n) + (n - 1)\, O(\lg n)$
  $= O(n) + O(n \lg n)$
  $= O(n \lg n)$

Priority Queues

- Heapsort is a nice algorithm, but in practice Quicksort (coming up) usually wins
- But the heap data structure is incredibly useful for implementing *priority queues*
  - A data structure for maintaining a set *S* of elements, each with an associated value or *key*
  - Supports the operations **`Insert()`**, **`Maximum()`**, and **`ExtractMax()`**
  - ***What might a priority queue be useful for?***

Priority Queue Operations

- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key
- *How could we implement these operations using a heap?*