Introduction to Algorithms

Merge Sort

Solving Recurrences

The Master Theorem

Review: Asymptotic Notation

- Upper Bound Notation:
  - f(n) is O(g(n)) if there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
  - Formally, $O(g(n)) = \{ f(n): \exists$ positive constants c and n_0 such that $f(n) \leq c \cdot g(n) \; \forall \; n \geq n_0$
- Big O fact:
  - A polynomial of degree k is $O(n^k)$

Review: Asymptotic Notation

- Asymptotic lower bound:
  - f(n) is *$\Omega(g(n))$* if $\exists$ positive constants c and n_0 such that $0 \le c \cdot g(n) \le f(n) \quad \forall\, n \ge n_0$
- Asymptotic tight bound:
  - f(n) is *$\Theta(g(n))$* if $\exists$ positive constants c_1, c_2, and n_0 such that $c_1\, g(n) \le f(n) \le c_2\, g(n) \ \forall\, n \ge n_0$
  - $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ AND $f(n) = \Omega(g(n))$

Other Asymptotic Notations

- A function f(n) is o(g(n)) if $\exists$ positive constants c and n_0 such that

  $$f(n) < c\, g(n) \;\; \forall\; n \geq n_0$$

- A function f(n) is ω(g(n)) if $\exists$ positive constants c and n_0 such that

  $$c\, g(n) < f(n) \;\; \forall\; n \geq n_0$$

- Intuitively,
  - o() is like <
  - ω() is like >
  - Θ() is like =
  - O() is like ≤
  - Ω() is like ≥

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
//    (how long should this take?)
```

Merge Sort: Example

- Show MergeSort() running on the array

```
A = {10, 5, 7, 6, 1, 4, 8, 3, 2, 9};
```

Analysis of Merge Sort

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

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

  $2T(n/2) + \Theta(n)$ when $n > 1$
- So what (more succinctly) is $T(n)$?

Recurrences

- The expression:

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

is a *recurrence*.

  - Recurrence: an equation that describes a function in terms of its value on smaller functions

Recurrence Examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

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

Solving Recurrences

- Substitution method
- Iteration method
- Master method

Solving Recurrences

- The substitution method (CLR 4.1)
  - 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
  - Examples:
    - $T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n \Rightarrow$???

Solving Recurrences

- The substitution method (CLR 4.1)
  - 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
  - Examples:
    - $T(n) = 2T(n/2) + \Theta(n) \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n \rightarrow$???

Solving Recurrences

- The substitution method (CLR 4.1)
  - 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
  - Examples:
    - $T(n) = 2T(n/2) + \Theta(n) \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n \rightarrow \Theta(n \lg n)$

Solving Recurrences

- Another option is what the book calls the "iteration method"
  - Expand the recurrence
  - Work some algebra to express as a summation
  - Evaluate the summation
- We will show several examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

- s(n) =

  c + s(n-1)

  c + c + s(n-2)

  2c + s(n-2)

  2c + c + s(n-3)

  3c + s(n-3)

  …

  kc + s(n-k) = ck + s(n-k)

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

- So far for n >= k we have
  - s(n) = ck + s(n-k)
- What if k = n?
  - s(n) = cn + s(0) = cn

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

- So far for n >= k we have
  - s(n) = ck + s(n-k)
- What if k = n?
  - s(n) = cn + s(0) = cn
- So
$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$
- Thus in general
  - s(n) = cn

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- s(n)

= n + s(n-1)

= n + n-1 + s(n-2)

= n + n-1 + n-2 + s(n-3)

= n + n-1 + n-2 + n-3 + s(n-4)

= …

= n + n-1 + n-2 + n-3 + … + n-(k-1) + s(n-k)

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- s(n)

= n + s(n-1)

= n + n-1 + s(n-2)

= n + n-1 + n-2 + s(n-3)

= n + n-1 + n-2 + n-3 + s(n-4)

= …

= n + n-1 + n-2 + n-3 + … + n-(k-1) + s(n-k)

$$= \sum_{i=n-k+1}^{n} i \quad + \quad s(n-k)$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- So far for n >= k we have

$$\sum_{i=n-k+1}^{n} i \quad + \quad s(n-k)$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- So far for n >= k we have

$$\sum_{i=n-k+1}^{n} i \quad + \quad s(n-k)$$

- What if k = n?

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- So far for n >= k we have

$$\sum_{i=n-k+1}^{n} i \quad + \quad s(n-k)$$

- What if k = n?

$$\sum_{i=1}^{n} i \quad + \quad s(0) \quad = \quad \sum_{i=1}^{n} i \quad + \quad 0 \quad = \quad n\frac{n+1}{2}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- So far for n >= k we have

$$\sum_{i=n-k+1}^{n} i \quad + \quad s(n-k)$$

- What if k = n?

$$\sum_{i=1}^{n} i \quad + \quad s(0) \quad = \quad \sum_{i=1}^{n} i \quad + \quad 0 \quad = \quad n\frac{n+1}{2}$$

- Thus in general

$$s(n) \quad = \quad n\frac{n+1}{2}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

- $T(n) =$

  $2T(n/2) + c$

  $2(2T(n/2/2) + c) + c$

  $2^2T(n/2^2) + 2c + c$

  $2^2(2T(n/2^2/2) + c) + 3c$

  $2^3T(n/2^3) + 4c + 3c$

  $2^3T(n/2^3) + 7c$

  $2^3(2T(n/2^3/2) + c) + 7c$

  $2^4T(n/2^4) + 15c$

  …

  $2^kT(n/2^k) + (2^k - 1)c$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\dfrac{n}{2}\right) + c & n > 1 \end{cases}$$

- So far for n > 2k we have
  - $T(n) = 2^k T(n/2^k) + (2^k - 1)c$
- What if k = lg n?
  - $T(n) = 2^{\lg n}\, T(n/2^{\lg n}) + (2^{\lg n} - 1)c$

    $= n\, T(n/n) + (n - 1)c$

    $= n\, T(1) + (n-1)c$

    $= nc + (n-1)c = (2n - 1)c$

Master Theorem

- Let T(n) be <u>a monotonically increasing</u> function that satisfies

$$T(n) = a\,T(n/b) + f(n)$$
$$T(1) = c$$

where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Pitfalls

- You cannot use the Master Theorem if
  - T(n) is not monotone, e.g. T(n) = sin(x)
  - f(n) is not a polynomial, e.g., $T(n)=2T(n/2)+2^n$
  - b cannot be expressed as a constant, e.g.

$$T(n) = T(\sqrt{n})$$

- Note that the Master Theorem does not solve the recurrence equation
- Does the base case remain a concern?

Master Theorem: Example 1

- Let $T(n) = T(n/2) + \frac{1}{2} n^2 + n$. What are the parameters?

  a = *1*

  b = *2*

  d = *2*

  Therefore, which condition applies?

  $1 < 2^2$, case 1 applies

- *We conclude that*

$$T(n) \in \Theta(n^d) = \Theta(n^2)$$

Master Theorem: Example 2

- Let T(n)= 2 T(n/4) + √n + 42. What are the parameters?

  a = *2*

  b = *4*

  d = *1/2*

  Therefore, which condition applies?

  2 = $4^{1/2}$, case 2 applies

- *We conclude that*

$$T(n) \in \Theta(n^d \log n) = \Theta(\log n \sqrt{n})$$

Master Theorem: Example 3

- Let T(n)= 3 T(n/2) + 3/4n + 1. What are the parameters?

  a = *3*

  b = *2*

  d = *1*

  Therefore, which condition applies?

  $3 > 2^1$, case 3 applies

- *We conclude that*

$$T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$$

- *Note that $\log_2 3 \approx 1.584\ldots$, can we say that $T(n) \in \Theta(n^{1.584})$*

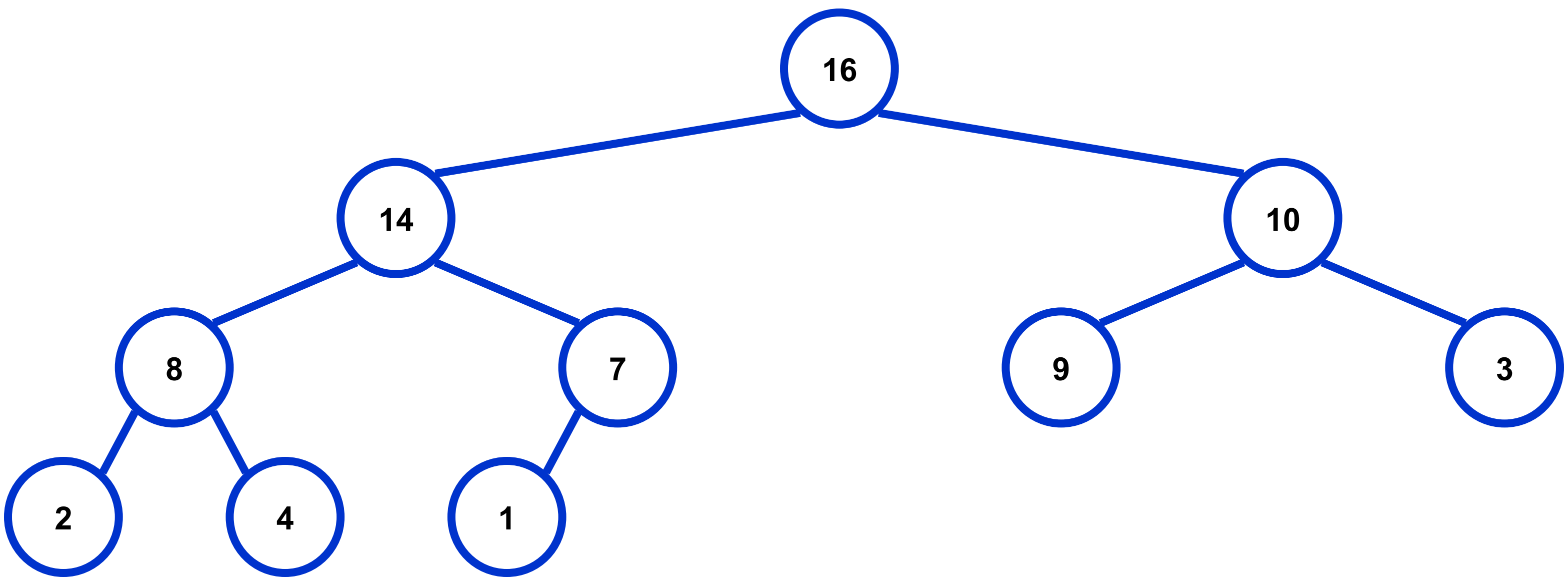
  No, because $\log_2 3 \approx 1.5849\ldots$ and $n^{1.584} \notin \Theta(n^{1.5849})$

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

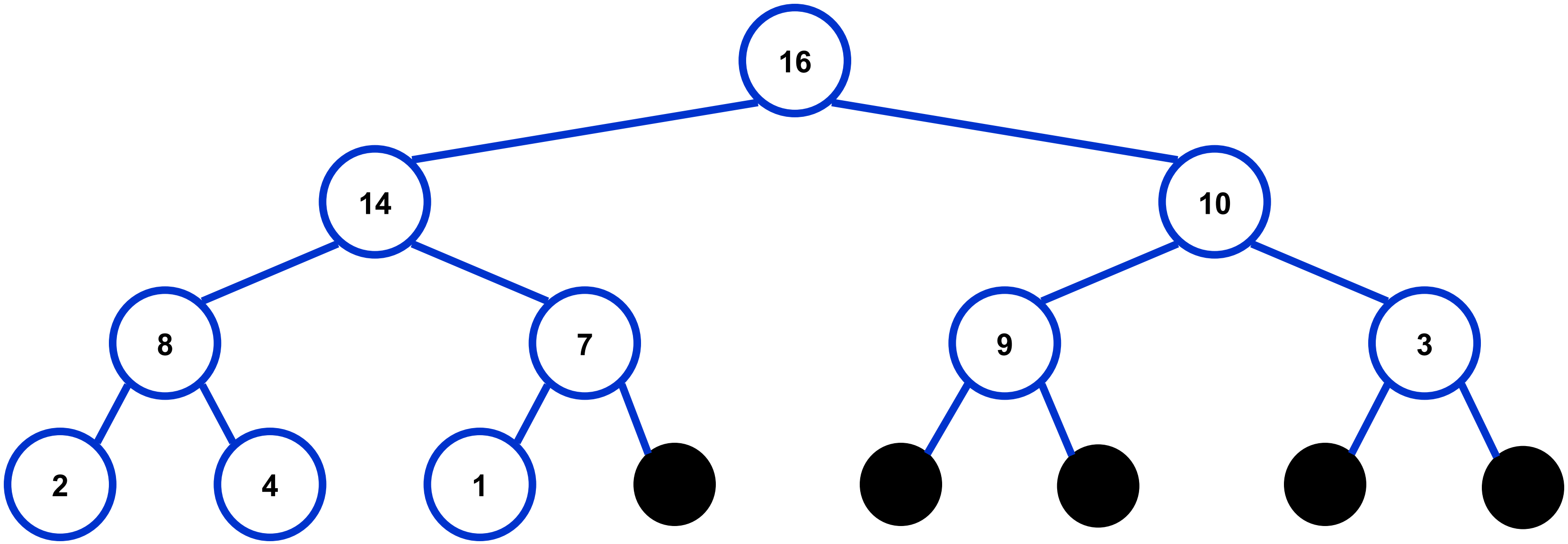
- A *heap* can be seen as a complete binary tree:



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

Heaps

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

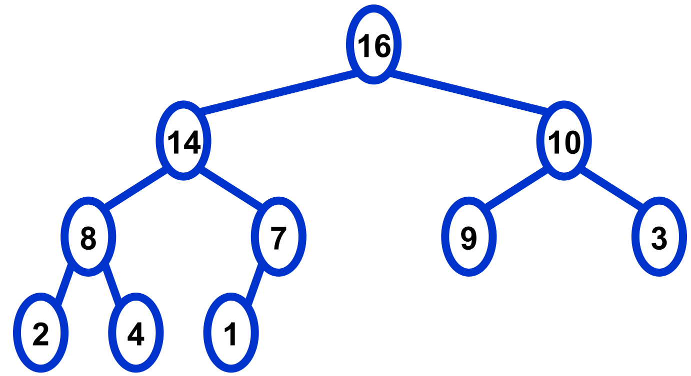


  - The book calls them "nearly complete" binary trees; can think of unfilled slots as null pointers

Heaps

- In practice, heaps are usually implemented as arrays:

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



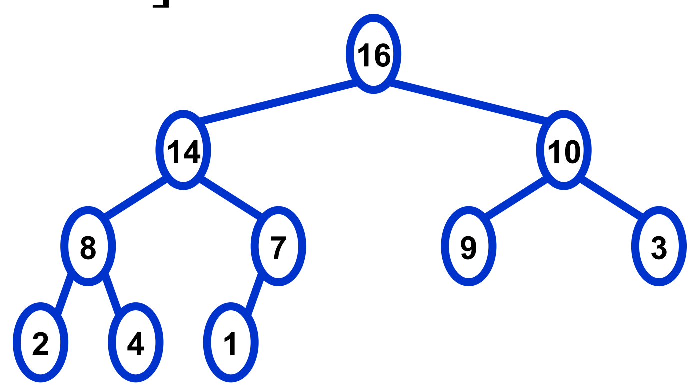
Heaps

- 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[2i]
  - The right child of node i is A[2i + 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)] $\geq$ 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

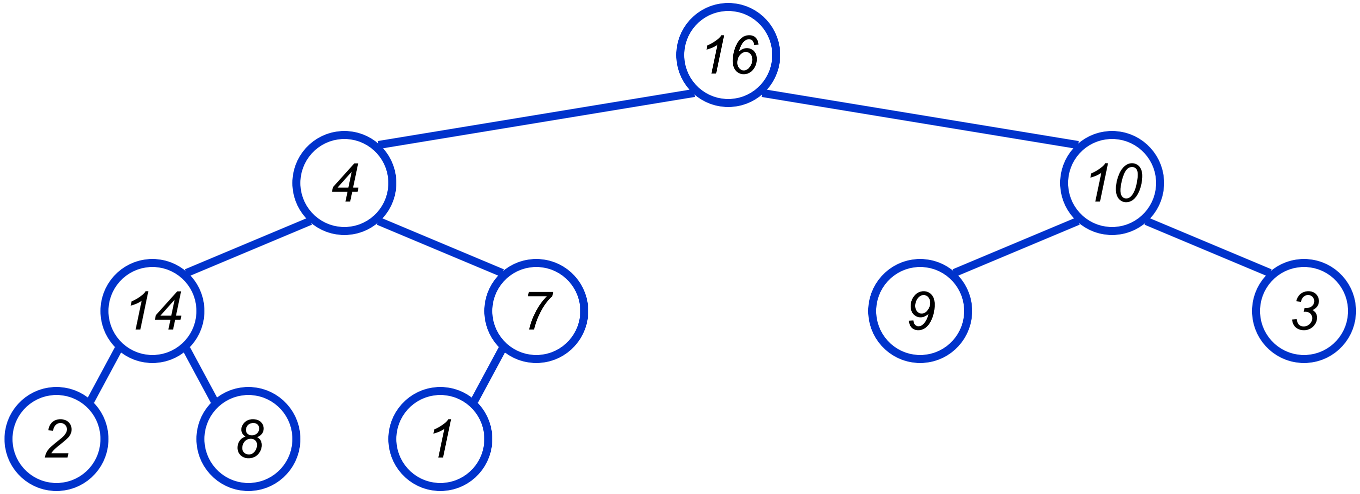
Heap Operations: Heapify()

- **`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()

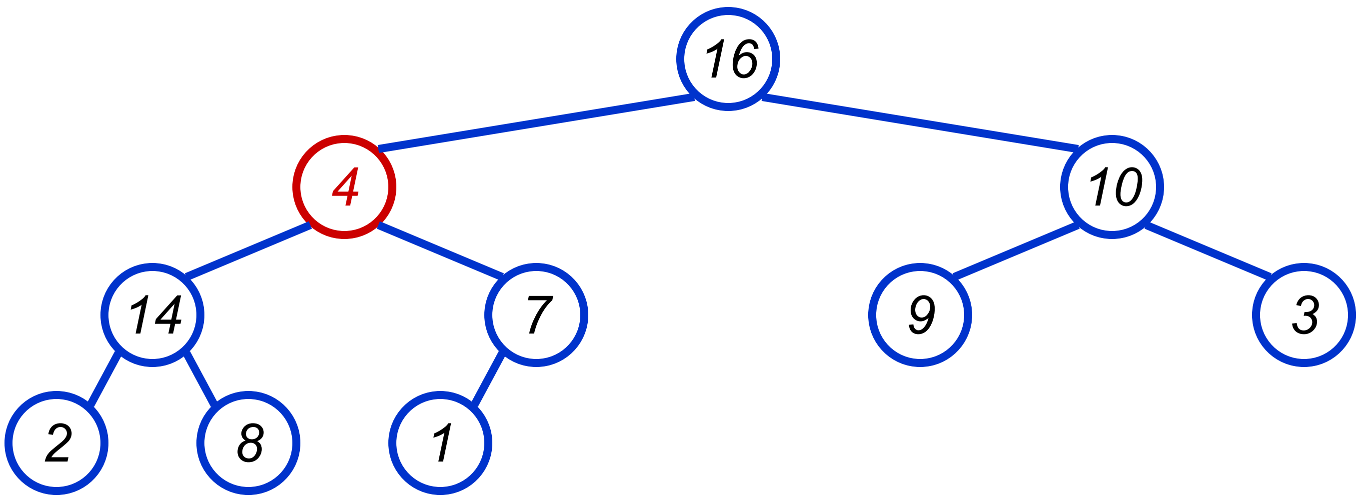
```
Heapify(A, i)
{
  l = Left(i); r = Right(i);
  if (l <= heap_size(A) && A[l] > A[i])
      largest = l;
  else
      largest = i;
  if (r <= heap_size(A) && A[r] > A[largest])
      largest = r;
  if (largest != i)
      Swap(A, i, largest);
      Heapify(A, largest);
}
```

Heapify() Example



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

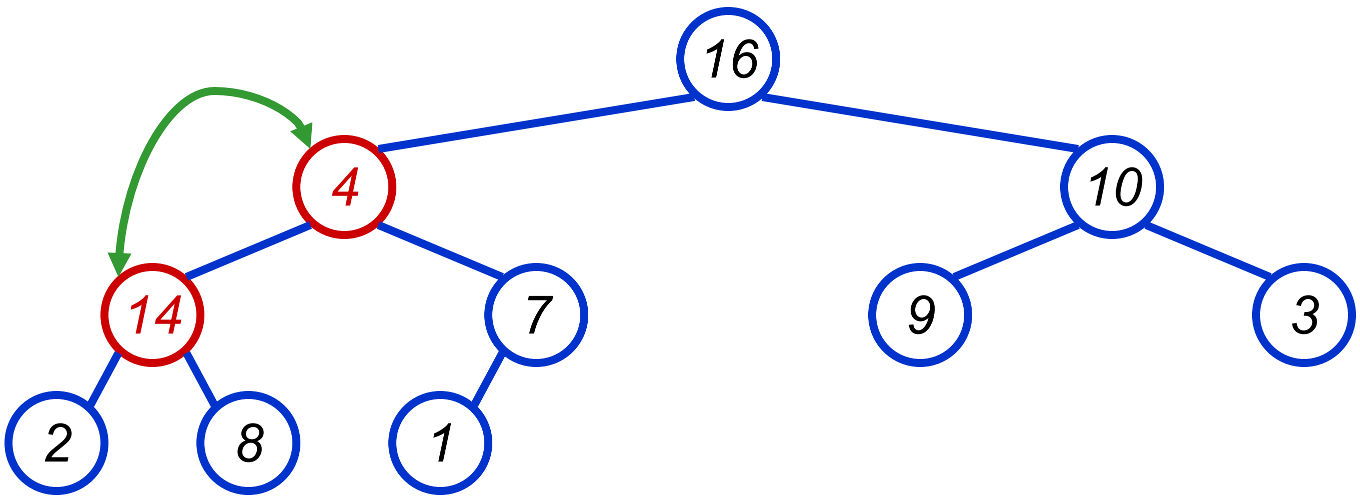
Heapify() Example



A =

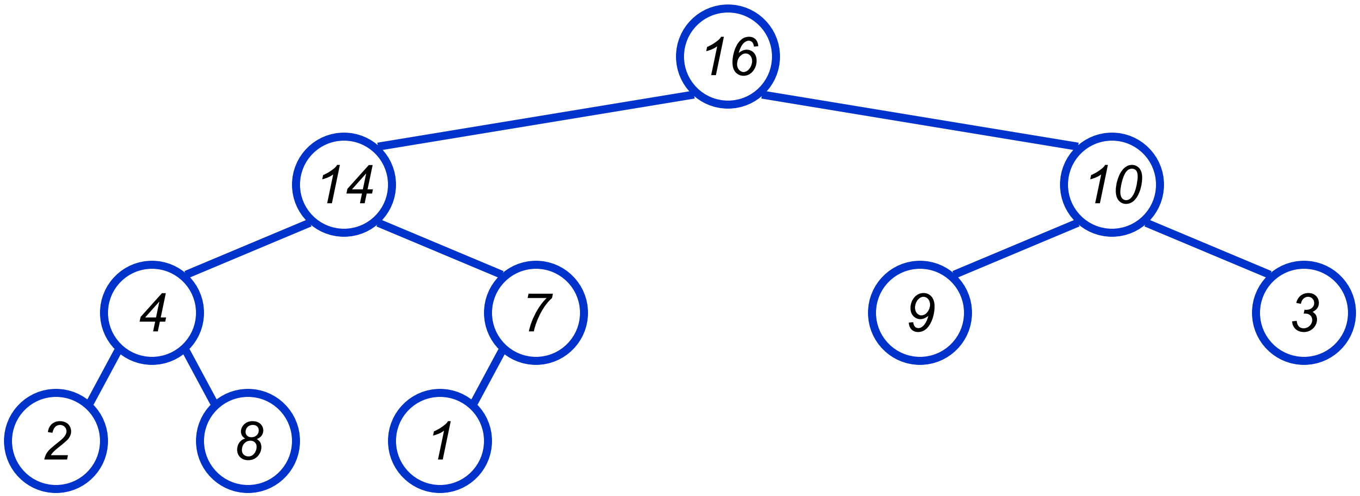
16	4	10	14	7	9	3	2	8	1

Heapify() Example



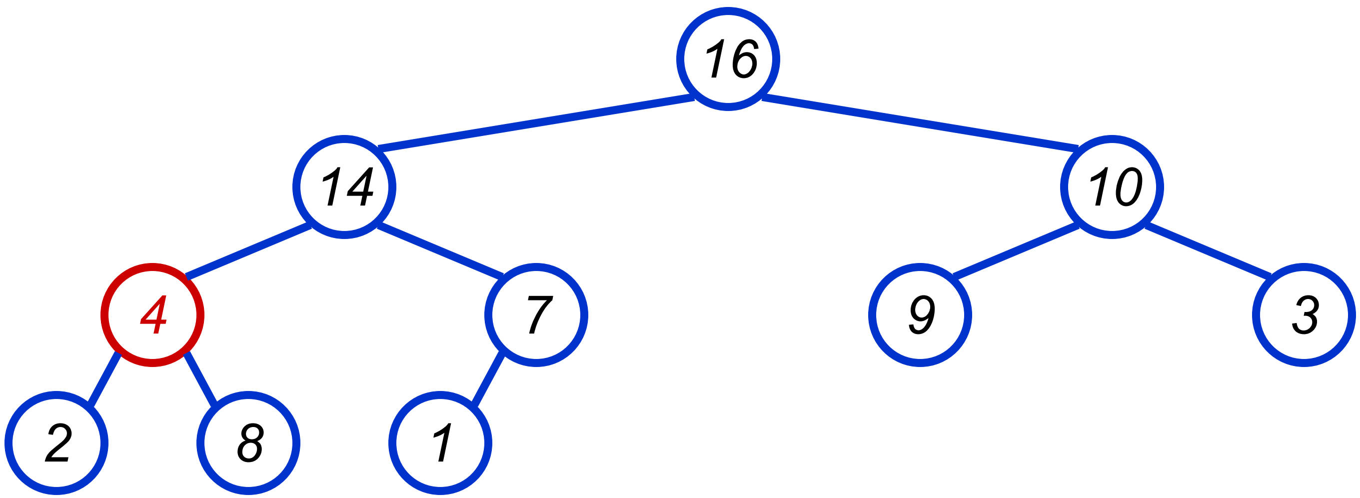
A =	16	4	10	14	7	9	3	2	8	1

Heapify() Example



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

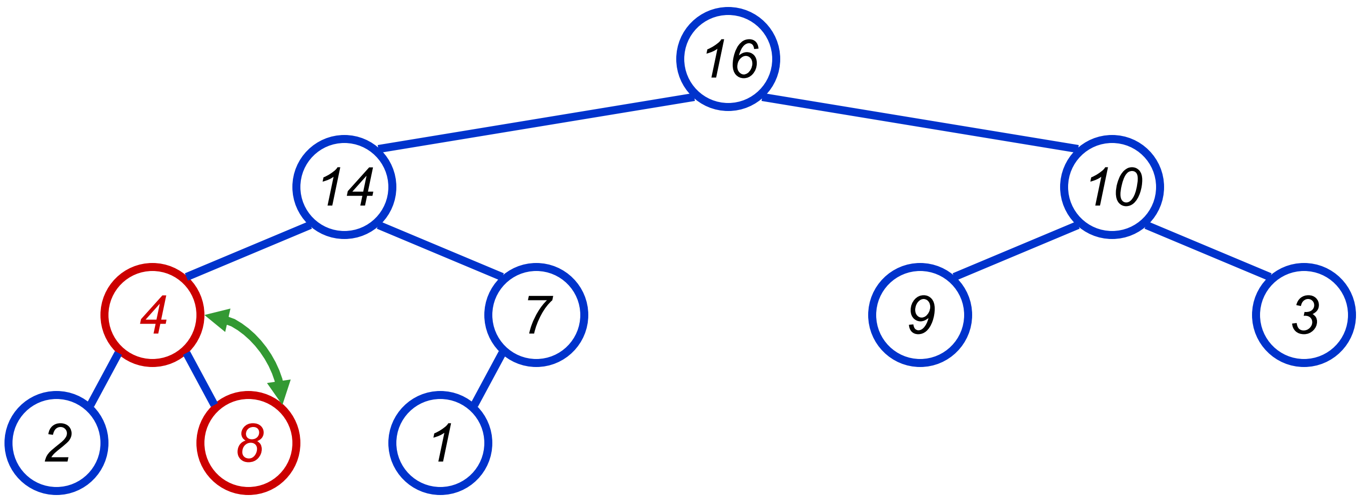
Heapify() Example



A =

16	14	10	4	7	9	3	2	8	1

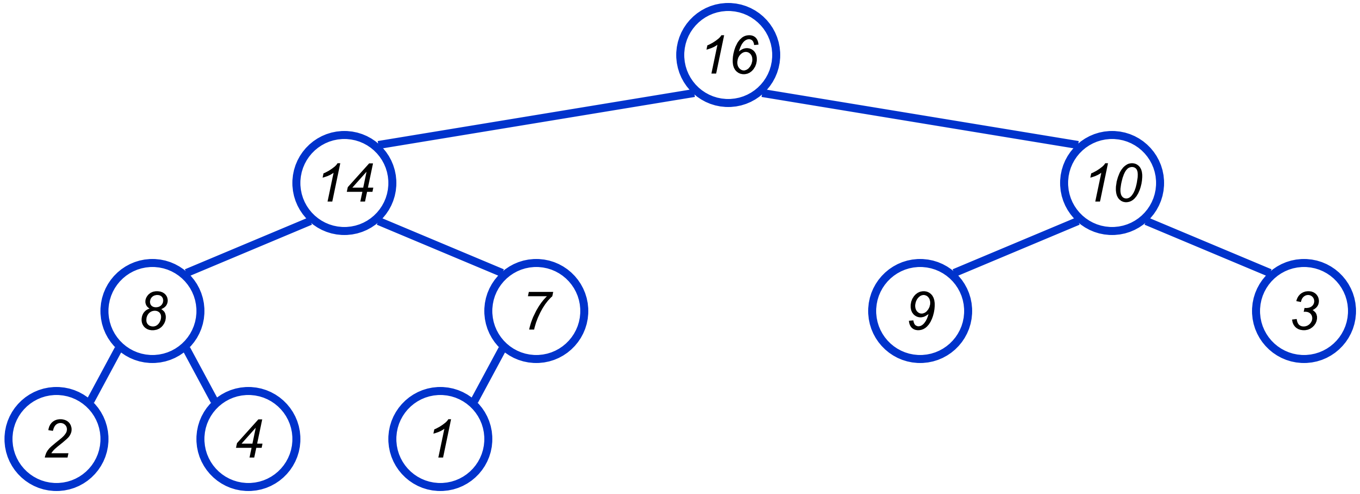
Heapify() Example



A =

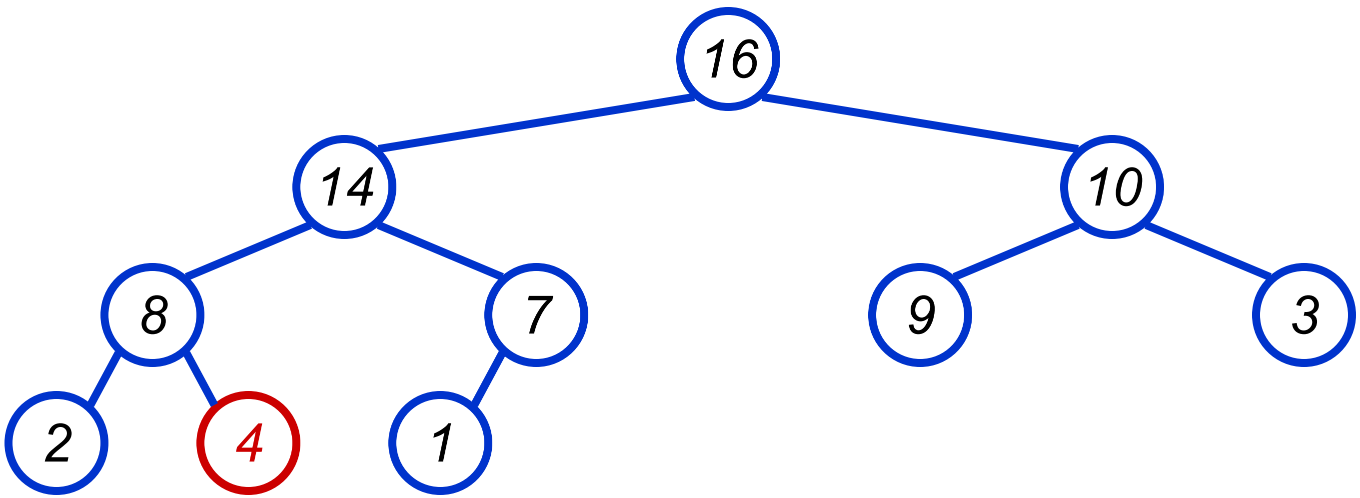
16	14	10	4	7	9	3	2	8	1

Heapify() Example



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

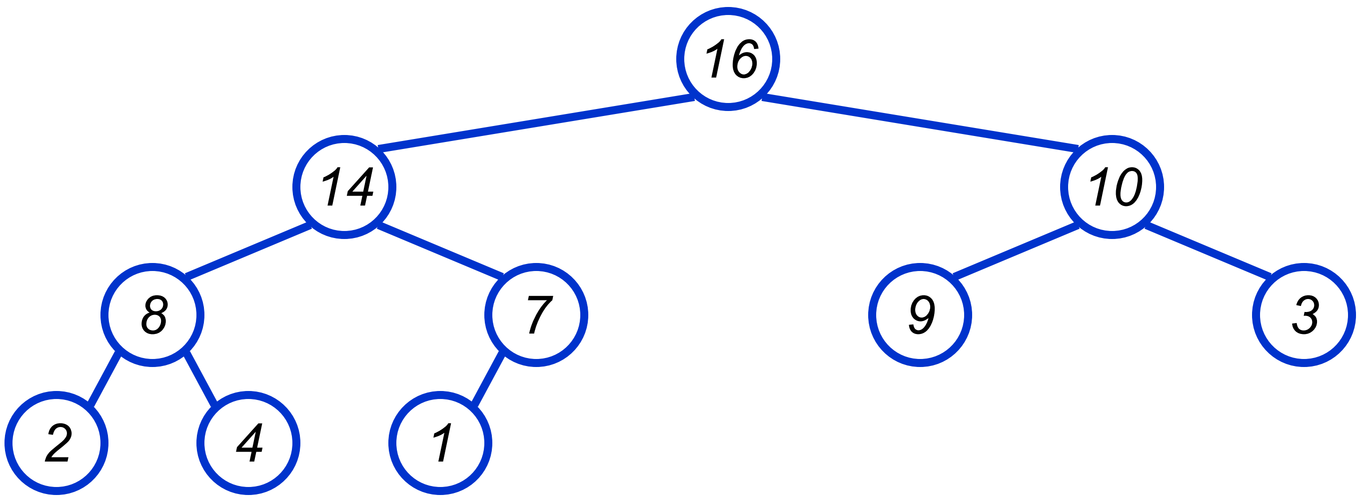
Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

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

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

- By case 2 of the Master Theorem,

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

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

Heap Operations: BuildHeap()

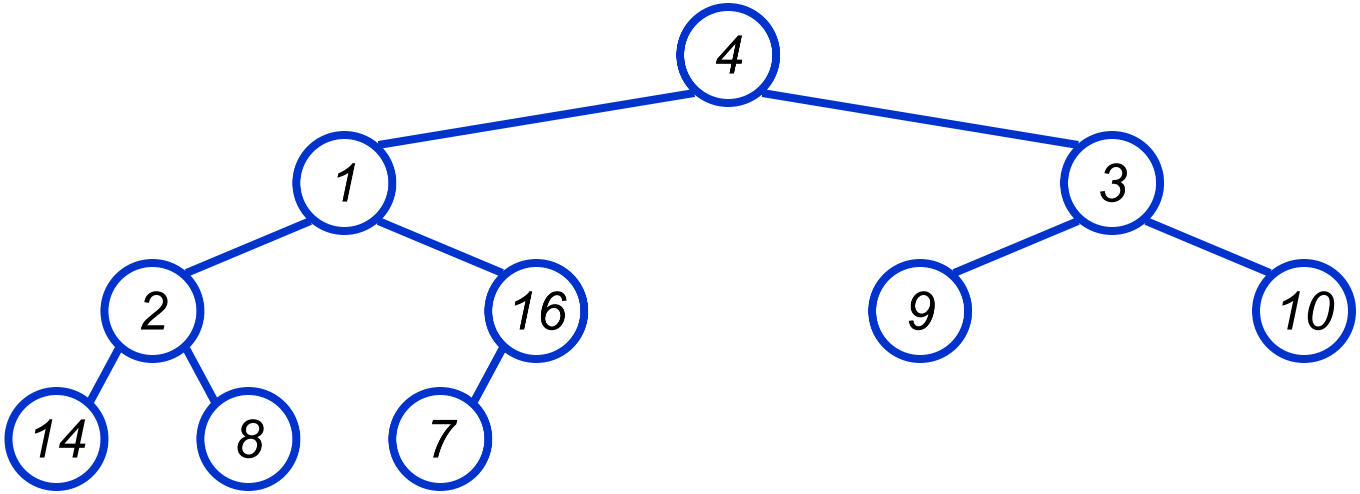
- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Fact: for array of length *n*, all elements in range $A[\lfloor n/2 \rfloor + 1 .. n]$ are heaps (*Why?*)
  - So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
    - 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 6.3 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?*