Analysis of Algorithms

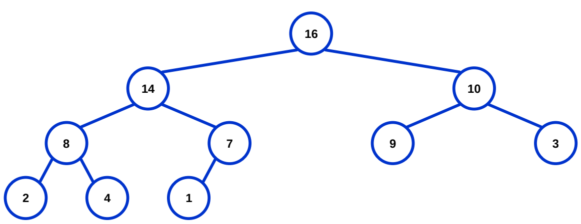
Heapsort

Sorting Revisited

- So far we've talked about two algorithms to sort an array of numbers
  - What is the advantage of merge sort?
    - Answer: O(n lg n) worst-case running time
  - What is the advantage of insertion sort?
    - Answer: sorts in place
    - Also: When array "nearly sorted", runs fast in practice
- Next on the agenda: *Heapsort*
  - Combines advantages of both previous algorithms

Heaps

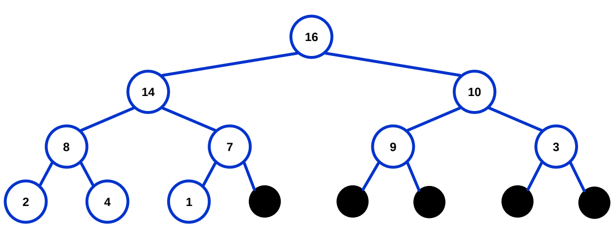
- A *heap* can be seen as a complete binary tree:



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

Heaps

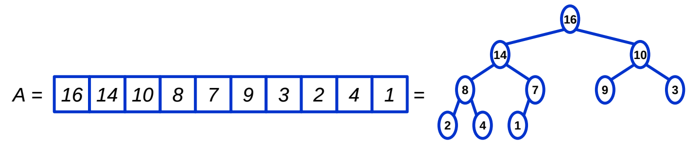
- A *heap* can be seen as a complete binary tree:



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

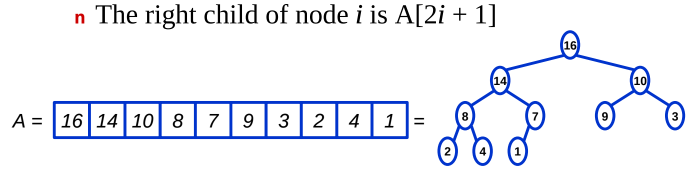
Heaps

- In practice, heaps are usually implemented as arrays:



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]



Referencing Heap Elements

- So…

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

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

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?*

Heap Height

- 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
- *What is the height of an n-element heap? Why?*

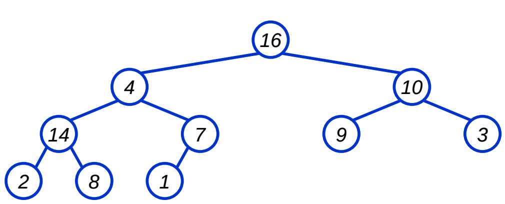
Heap Operations: Heapify()

- **`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
  - Action: let the value of the parent node “float down” so subtree at *i* satisfies the heap property

Heap Operations: Heapify()

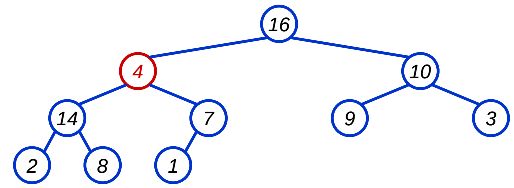
```
Heapify(A, i)
{
   l = Left(i); r = Right(i);
   if (l <= heap_size(A) && A[l] > A[i])
       largest = l;
   else
       largest = i;
   if (r <= heap_size(A) && A[r] > A[largest])
       largest = r;
   if (largest != i)
       Swap(A, i, largest);
       Heapify(A, largest);
}
```


Heapify() Example



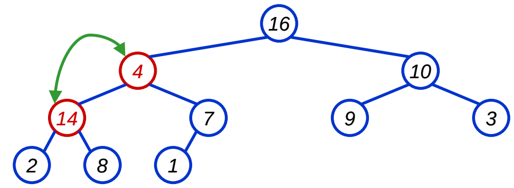
A =										
	16	4	10	14	7	9	3	2	8	1

Heapify() Example



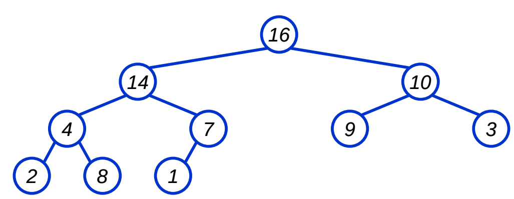
A =										
	16	4	10	14	7	9	3	2	8	1

Heapify() Example



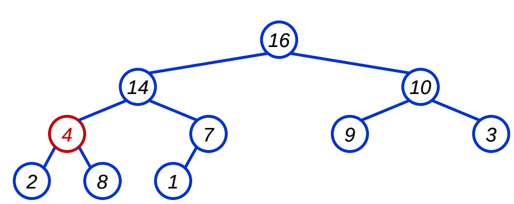
A =										
	16	4	10	14	7	9	3	2	8	1

Heapify() Example



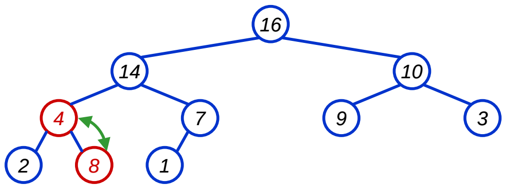
A =										
	16	14	10	4	7	9	3	2	8	1

Heapify() Example



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

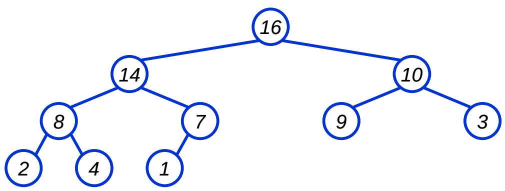
Heapify() Example



A =

16	14	10	4	7	9	3	2	8	1

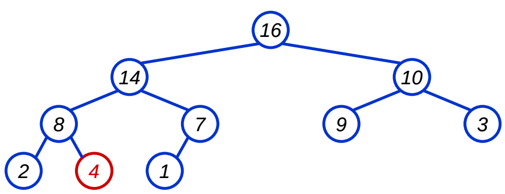
Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

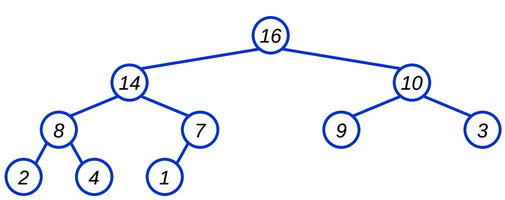
Heapify() Example



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

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

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 logarithmic time

Heap Operations: BuildHeap()

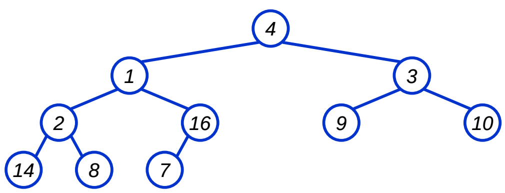
- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Fact: for array of length n, all elements in range $A[\lfloor n/2 \rfloor + 1 .. n]$ are heaps (*Why?*)
  - So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.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

- A data structure for maintaining a set *S* of elements, each with an associated value called a *key*.
- Applications: scheduling jobs on a shared computer, prioritizing events to be processed based on their predicted time of occurrence.
- Heap can be used to implement a priority queue.

Basic Operations

- Insert(*S*, *x*) - inserts the element *x* into the set *S*, i.e. $S \rightarrow S \cup \{x\}$
- 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

Heap-Extract-Max(*A*)

```
if heap-size[A] < 1
   then error "heap underflow"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] - 1
Heapify(A, 1)
return max
```


Heap-Insert(*A*, *key*)

```
heap-size[A] ← heap-size[A] + 1
i ← heap-size[A]
while i > 1 and A[Parent(i)] < key
     do A[i] ← A[Parent(i)]
          i ← Parent(i)
A[i] ← key
```


Running Time

- Running time of Heap-Extract-Max is $O(\lg n)$.
  - Performs only a constant amount of work on top of Heapify, which takes $O(\lg n)$ time
- Running time of Heap-Insert is $O(\lg n)$.
  - The path traced from the new leaf to the root has length $O(\lg n)$.

Examples

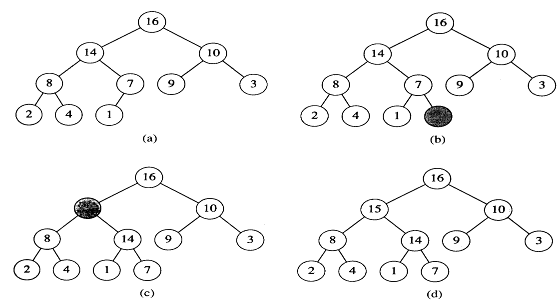


Figure 7.5 The operation of HEAP-INSERT. **(a)** The heap of Figure 7.4(a) before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** The key 15 is inserted.