

Chapter 6

Heapsort

Algorithm Analysis

School of CSEE

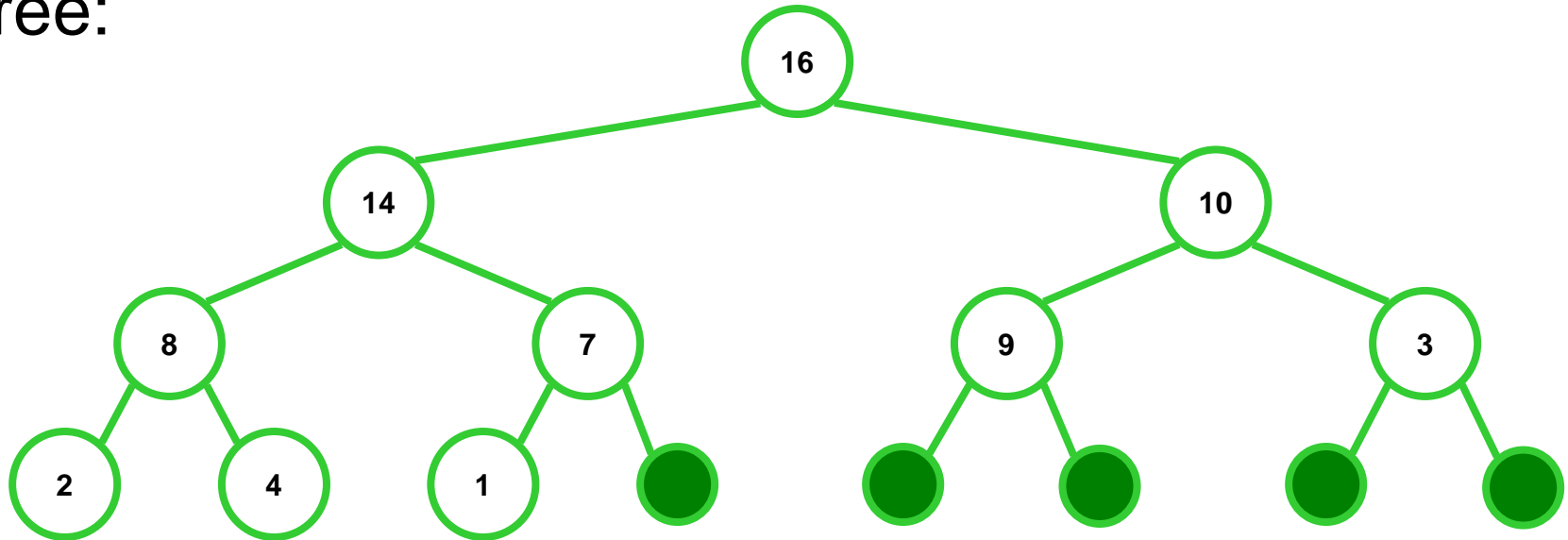
Sorting Revisited

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

Sort in place : Only a constant number of array elements are stored outside the input array at any time.

Heaps

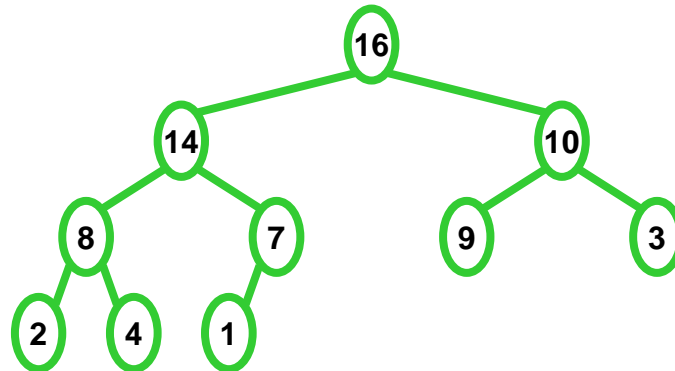
- A *heap* can be seen as a nearly complete binary tree:



– We can think of unfilled slots as null pointers

Heaps

- In practice, heaps are usually implemented as arrays:



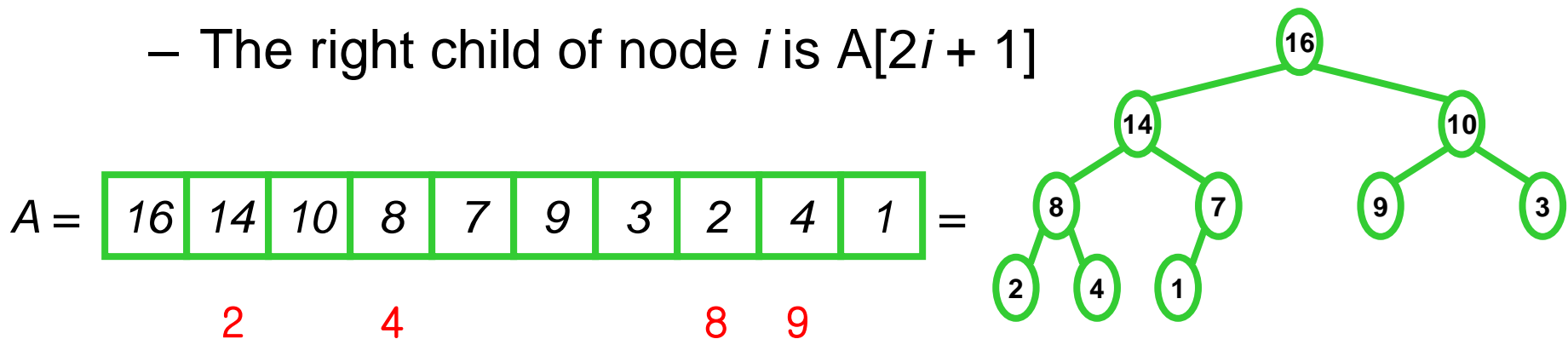
=

$A =$

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

Heaps

- To represent a nearly 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 **max tree** is a tree in which the key value in each node is no smaller than the key values in its children.
- A **min tree** is a tree in which the key value in each node is no larger than the key values in its children.
- A **max-heap** is a nearly complete binary tree that is also a max tree.
- A **min-heap** is a nearly complete binary tree that is also a min tree.

The Heap Property

- Heaps satisfy the *heap property*:

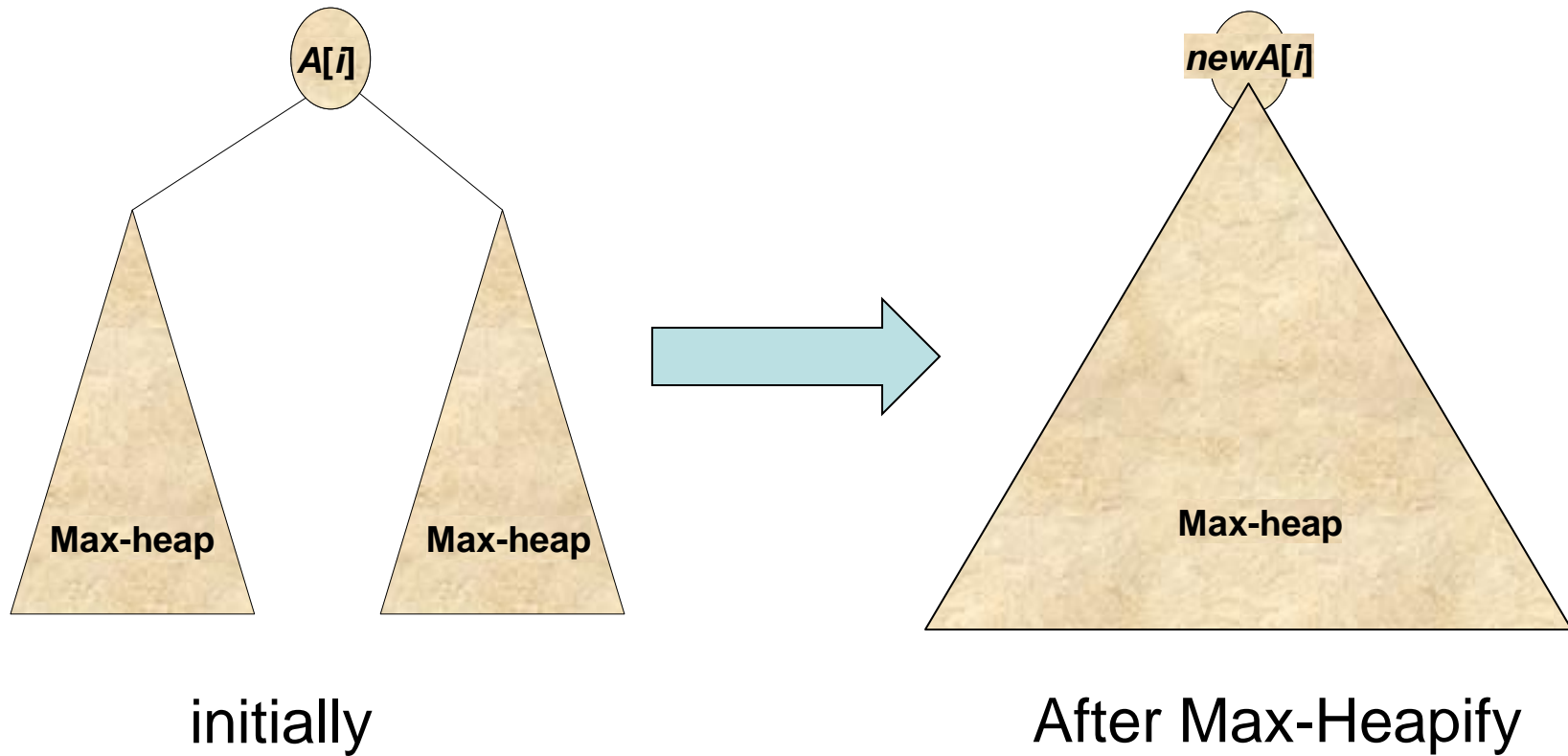
$$A[\text{Parent}(i)] \geq A[i] \text{ for all nodes } i > 1$$

- 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 = $\Theta(\lg n)$.

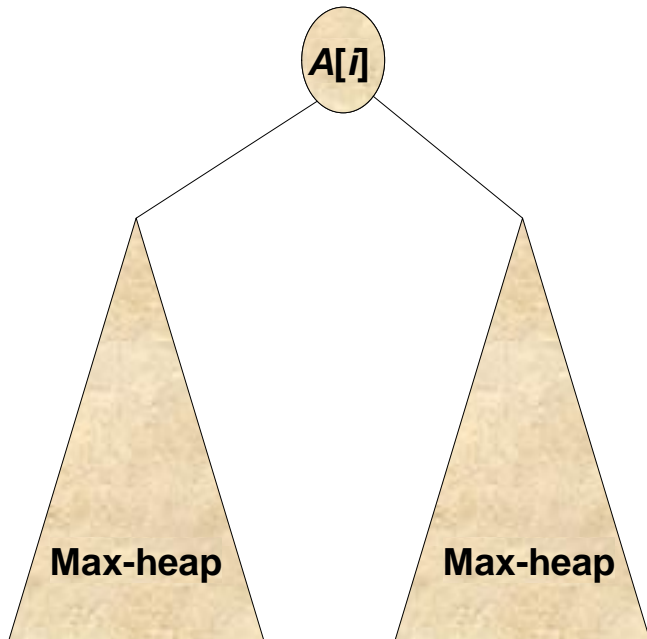
- **MAX-HEAPIFY** maintains the max-heap property.
 - Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
 - Assume left and right subtrees of i are max-heaps.
 - After MAX-HEAPIFY, subtree rooted at i is a max-heap.

MAX-HEAPIFY lets the value at $A[i]$ “float down” in the max-heap so that the subtree rooted at index i becomes a max-heap.

Maintaining the Max-Heap



Maintaining the Max-Heap

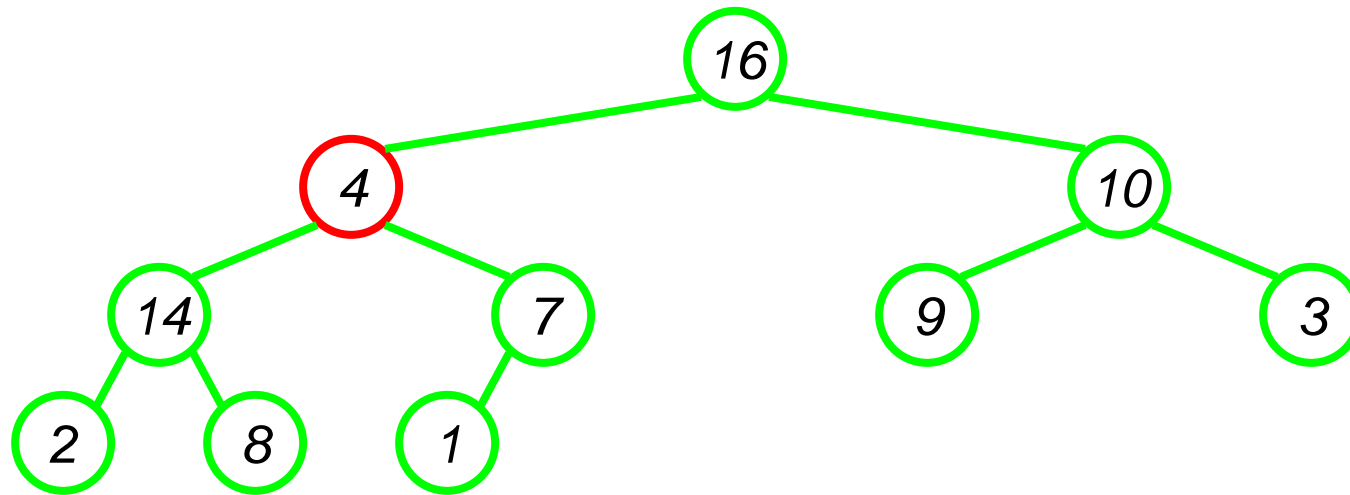


Max-Heapify(A, i)

```

1   $l = \text{left}(i)$ 
2   $r = \text{right}(i)$ 
3  if  $l \leq \text{heap-size}[A]$  and  $A[l] > A[i]$ 
4    then  $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq \text{heap-size}[A]$  and  $A[r] > A[\text{largest}]$ 
7    then  $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9    then exchange  $A[i]$  with  $A[\text{largest}]$ 
10   Max-Heapify( $A, \text{largest}$ )
  
```

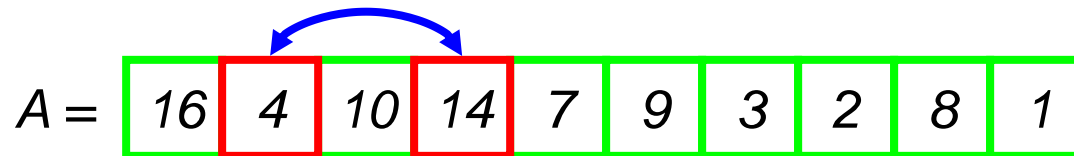
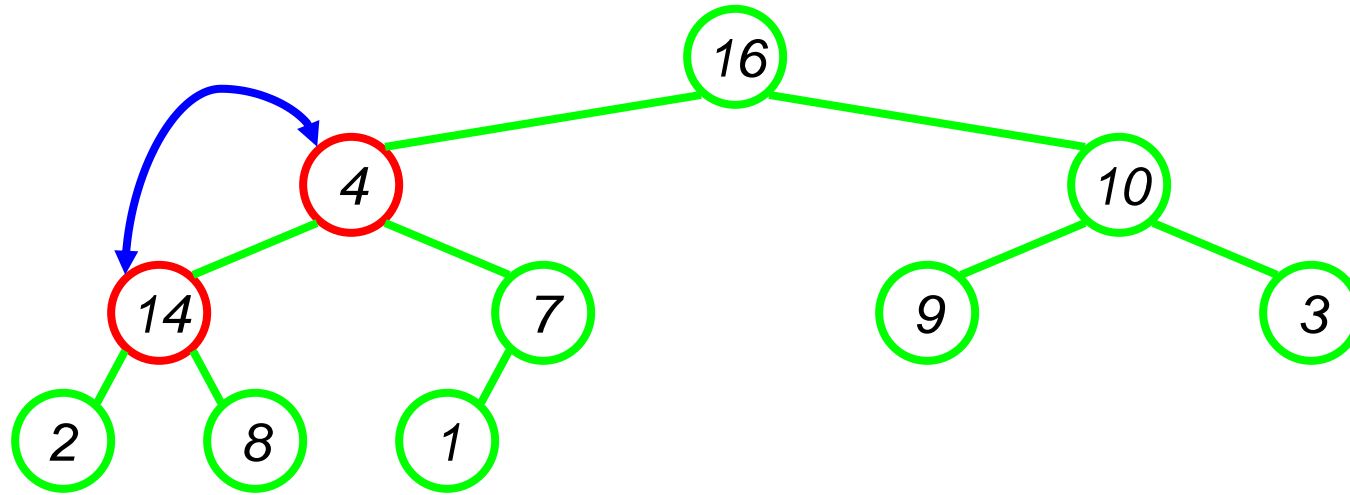
Max-Heapify() Example



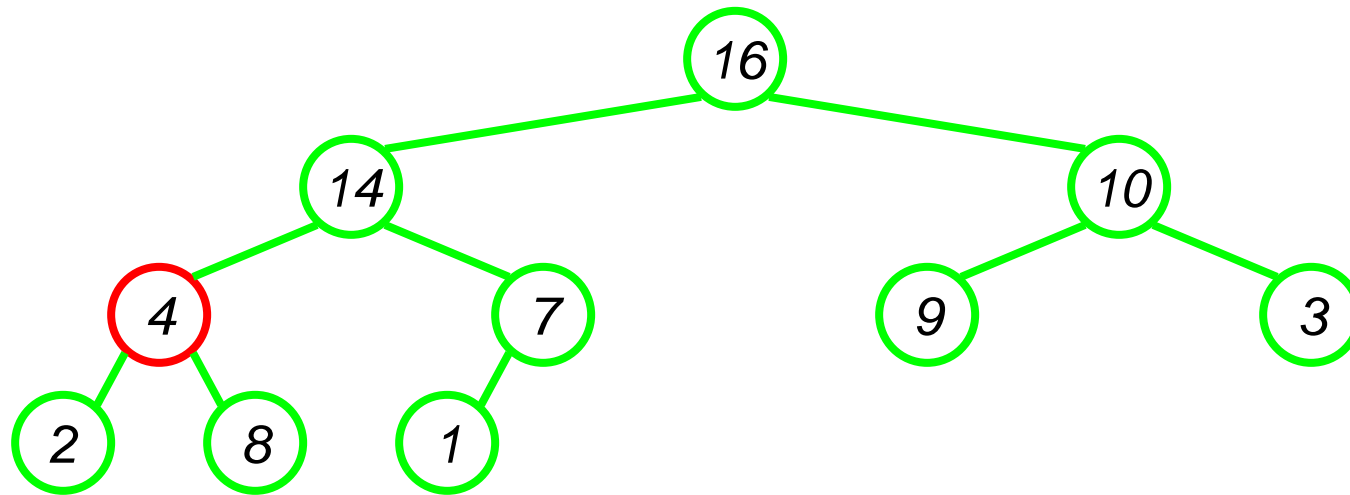
$A =$

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

Max-Heapify() Example



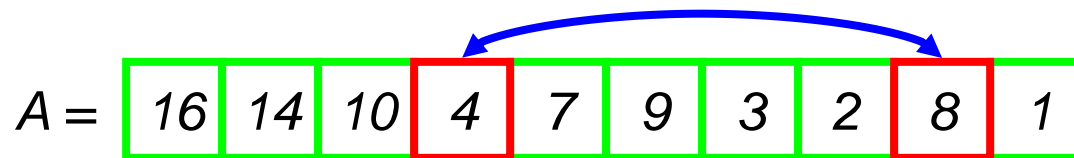
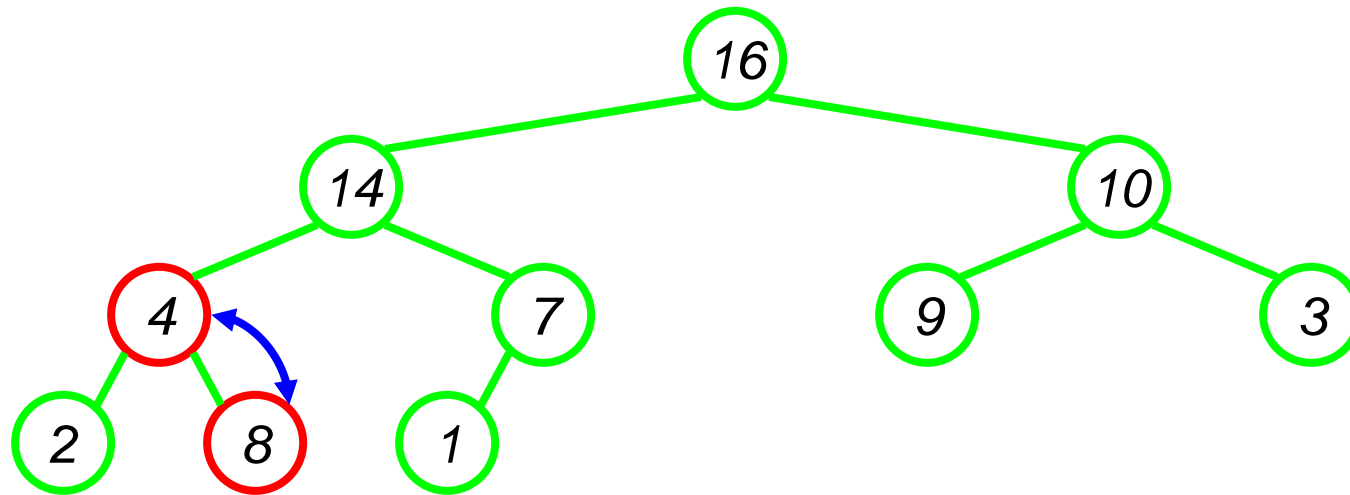
Max-Heapify() Example



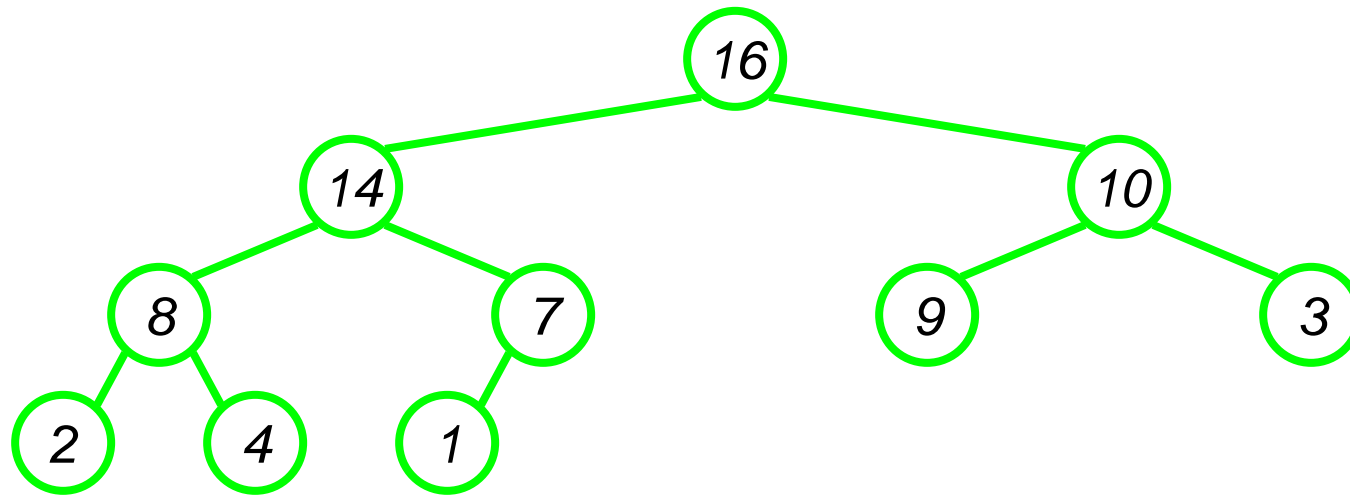
$A =$

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

Max-Heapify() Example



Max-Heapify() Example



$A =$

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

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?*

- Worst case = Most unbalanced case

$2n/3$ on left subtree and $n/3$ on right subtree

(bottom row 1/2 full)

- So time taken by **Max-Heapify()** is given by

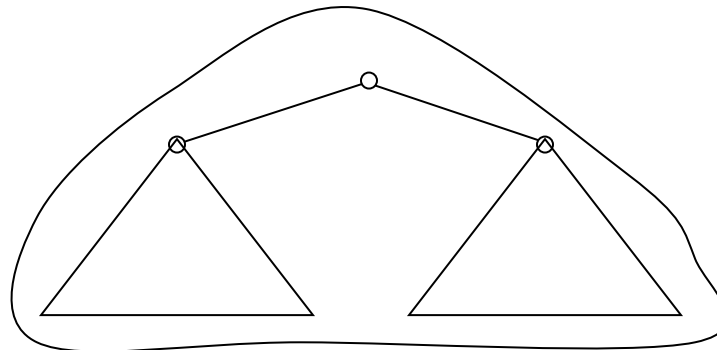
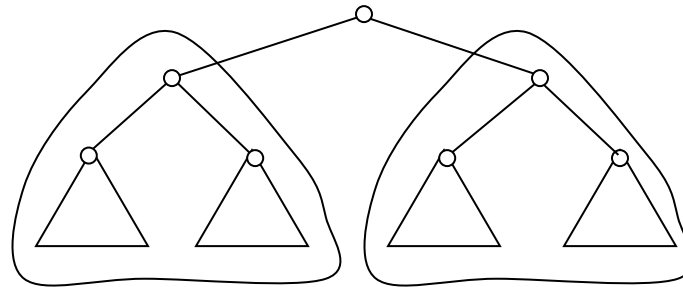
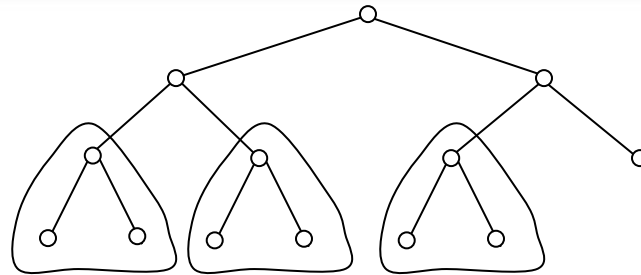
$$T(n) \leq T(2n/3) + \Theta(1)$$

By case 2 of the Master Theorem,

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

- We can build a heap in a bottom-up manner by running **Max-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 **Max-Heapify()** on each node.
 - Order of processing guarantees that the children of node i are heaps when i is processed

How to create a heap?



Building a Heap

Build-Max-Heap(A)

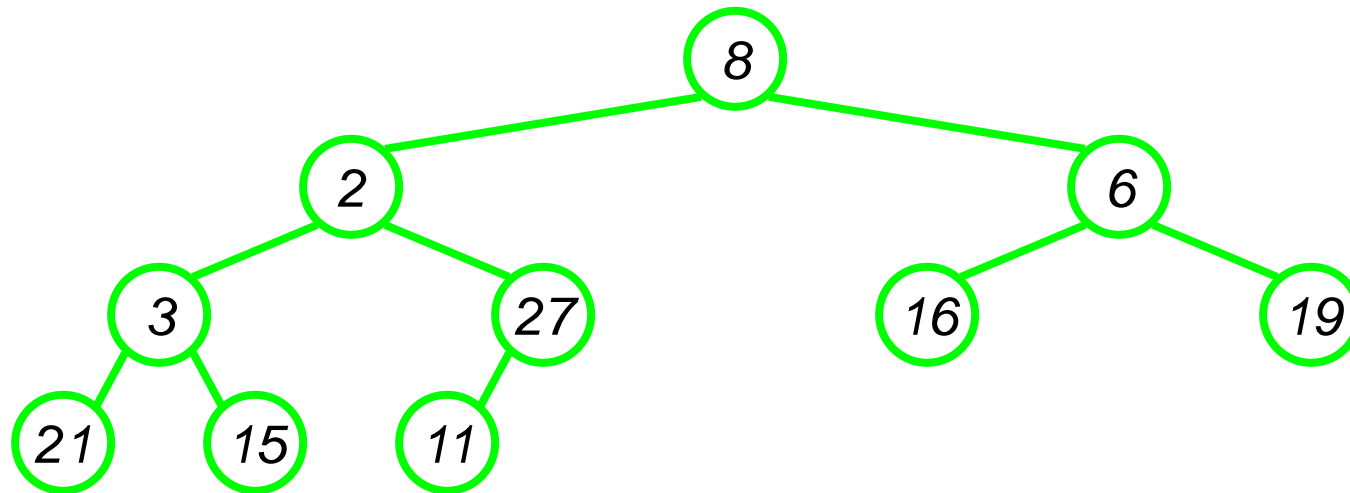
```
1  heap-size[ $A$ ] = length[ $A$ ]  
2  for  $i = \left\lfloor \frac{\text{length}[A]}{2} \right\rfloor$  downto 1  
3      do Max-Heapify( $A, i$ )
```

At the start of each iteration of the “**for**” loop of lines 2-3, each node $i+1, i+2, \dots, n$ is the root of a max-heap.

Build-Max-Heap() Example

- Work through example


$A = \{8, 2, 6, 3, 27, 16, 19, 21, 15, 11\}$



Analyzing Build-Max-Heap()

- Each call to **Max-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?*

- To **Max-Heapify()** a subtree takes $O(h)$ time where h is the height of the subtree
 - $h = O(\lg n)$, $n = \#$ 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 .

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


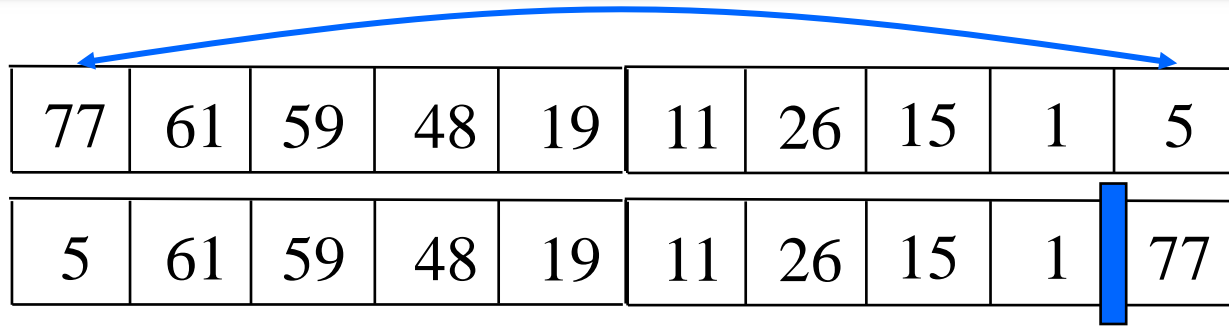
Page 1148 (A.8)

Heapsort

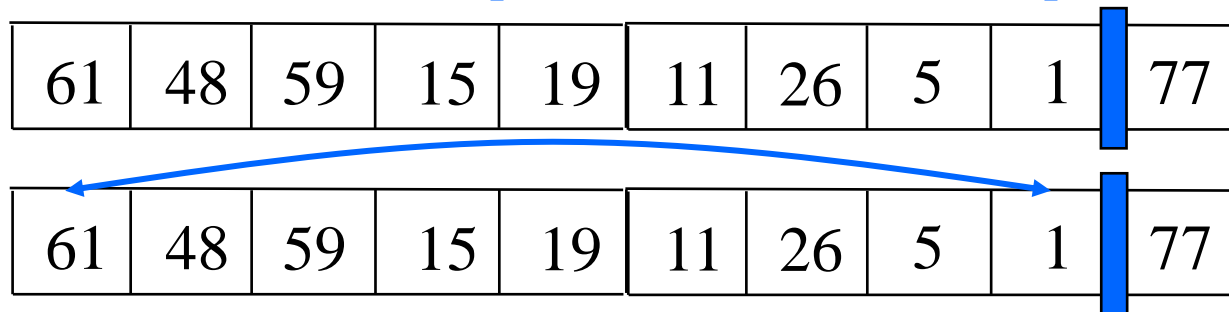
- Given **Build-Max-Heap** () , an in-place sorting algorithm is easily constructed:
 - Maximum element is at $A[1]$
 - Discard by swapping with element at $A[n]$
 - Decrement $\text{heap_size}[A]$
 - $A[n]$ now contains correct value
 - Restore heap property at $A[1]$ by calling **Max-Heapify** ()
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

Heapsort

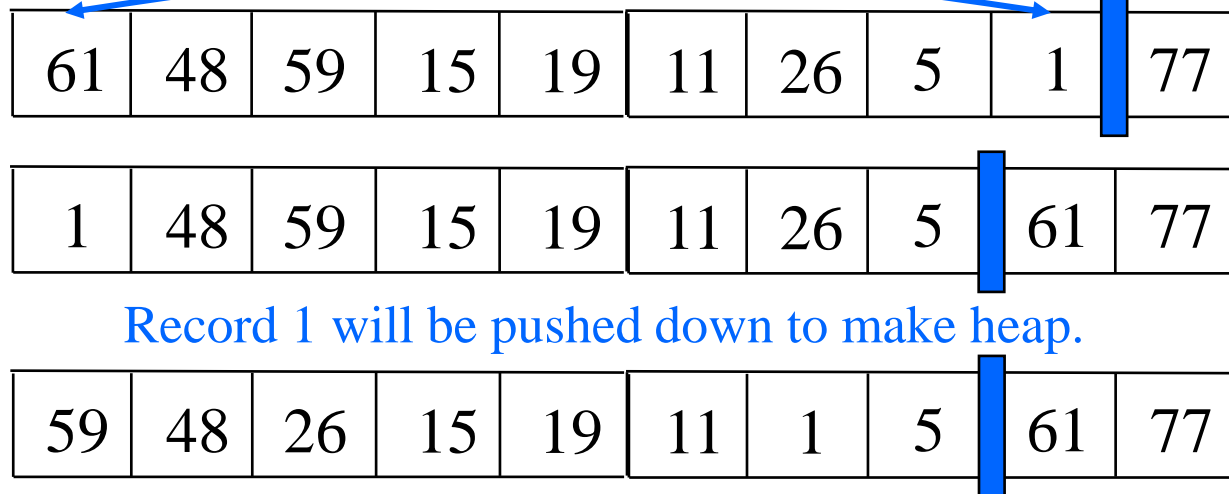
Pass 1



Record 5 will be pushed down to make heap.



Pass 2



Record 1 will be pushed down to make heap.

Heapsort

Heapsort(A)

```
1  Build-Max-Heap( $A$ )
2  for  $i = \text{length}[A]$  downto 2
3      do exchange  $A[1]$  with  $A[i]$ 
4          heap-size[ $A$ ] = heap-size[ $A$ ] - 1
5          Max-Heapify( $A, 1$ )
```

Steps 1 : $O(n)$

Step 3-5: $O(\lg n)$

$$T(n) = O(n) + (n - 1)O(\lg n) = O(n \lg n)$$

Heapsort

Heapsort(A)

```

1  Build-Max-Heap( $A$ )
2  for  $i = \text{length}[A]$  downto 2
3    do exchange  $A[1]$  with  $A[i]$ 
4      heap-size[ $A$ ] = heap-size[ $A$ ] - 1
5      Max-Heapify( $A, 1$ )
  
```

Build-Max-Heap(A)

```

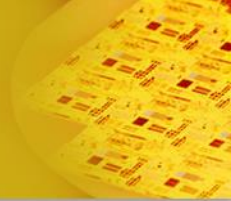
1  heap-size[ $A$ ] = length[ $A$ ]
2  for  $i = \left\lfloor \frac{\text{length}[A]}{2} \right\rfloor$  downto 1
3    do Max-Heapify( $A, i$ )
  
```

Max-Heapify(A, i)

```

1   $l = \text{left}(i)$ 
2   $r = \text{right}(i)$ 
3  if  $l \leq \text{heap-size}[A]$  and  $A[l] > A[i]$ 
4    then  $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq \text{heap-size}[A]$  and
    $A[r] > A[\text{largest}]$ 
7    then  $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9    then exchange  $A[i]$  /  $A[\text{largest}]$ 
10   Max-Heapify( $A, \text{largest}$ )
  
```

Priority Queues



- A well-implemented quicksort usually beats heapsort in practice.
- But the heap data structure is incredibly useful for implementing *priority queues*. The element to be deleted is the one with highest (or lowest) priority.

- Priority queue
 - Maintains a dynamic set S of elements.
 - Each set element has a key.
 - Example of max-priority queue : schedule jobs on shared computers.
 - Example of min-priority queue : Utilizing the service machine time. Job with smallest service time is selected.

Dynamic Set : The set that can grow, shrink, or otherwise change over time.

Priority Queue - Insertion

(A,1)					
-------	--	--	--	--	--

Insert (B,3)

(B,3)	(A,1)				
-------	-------	--	--	--	--

Insert (C,2)

(B,3)	(A,1)	(C,2)			
-------	-------	-------	--	--	--

Insert (D,1)

(B,3)	(A,1)	(C,2)	(D,1)		
-------	-------	-------	-------	--	--

Insert (E,3)

(B,3)	(E,3)	(C,2)	(D,1)	(A,1)	
-------	-------	-------	-------	-------	--

Priority Queue - Deletion

(B,3)	(E,3)	(C,2)	(D,1)	(A,1)	
-------	-------	-------	-------	-------	--

Delete (B,3)

(E,3)	(A,1)	(C,2)	(D,1)		
-------	-------	-------	-------	--	--

Delete (E,3)

(C,2)	(A,1)	(D,1)			
-------	-------	-------	--	--	--

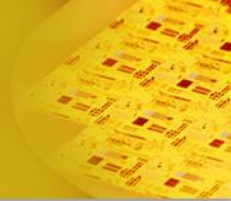
Delete (C,2)

(A,1)	(D,1)				
-------	-------	--	--	--	--

Delete(A,1)

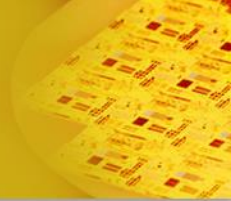
(D,1)					
-------	--	--	--	--	--

Priority queues



- Max-priority queue supports dynamic-set operations :
 - INSERT(S, x) : inserts element x into set S .
 - MAXIMUM(S) : returns element of S with largest key.
 - EXTRACT-MAX(S) : removes and returns element of S with largest key.
 - INCREASE-KEY(S, x, k) : increases value of element x 's key to k . (Assume $k \geq x$'s current key value).
- Min-priority queue supports similar operations.

Maximum operation



Heap-Maximum(A)

1 Return $A(1)$

$O(1)$ running time

Extracting the maximum

Heap-Extract-Max(A)

```
1  if heap-size[ $A$ ] < 1
2    then error "heap underflow"
3   $max = A[1]$ 
4   $A[1] = A[heapsize[A]]$ 
5   $Heapsize[A] = Heapsize[A] - 1$ 
6  Max-Heapify( $A, 1$ )
7  return  $max$ 
```

$O(\lg n)$ running time

Increase-Key operation

Heap-Increase-Key(A, i, key)

```
1  if  $key < A[i]$ 
2      then error "new key is smaller than current key"
3   $A[i] = key$ 
4  while  $i > 1$  and  $A[parent(i)] < A[i]$ 
5      do exchange  $A[i]$  with  $A[parent(i)]$ 
6       $i = parent(i)$ 
```

$O(\lg n)$ running time

Insert operation

Max-Heap-Insert(A, key)

1 $heapsize[A] = heapsize[A] + 1$

2 $A[heapsize[A]] = -\infty$

3 Heap-Increase-Key($A, heapsize[A], key$)

$O(\lg n)$ running time