# Introduction to Algorithms

**George Voutsadakis**[1]

[1]Mathematics and Computer Science
Lake Superior State University

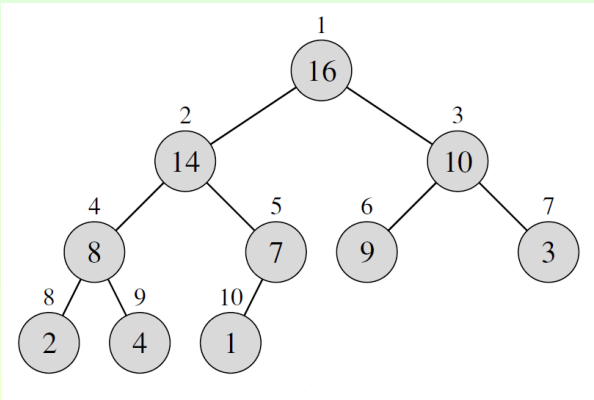
LSSU Math 400

Subsection 1

# Heaps

# Heap

- The (**binary**) **heap** data structure is an array object that we can view as a nearly complete binary tree:

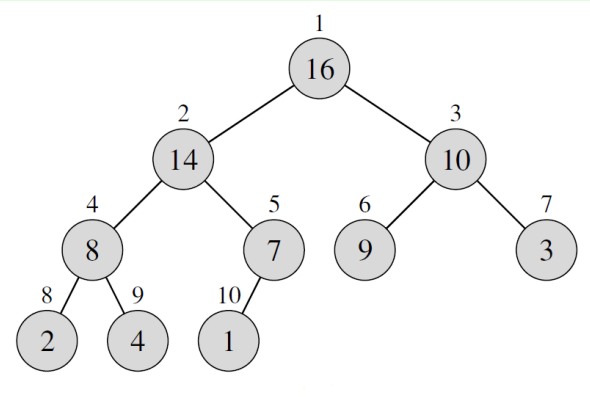


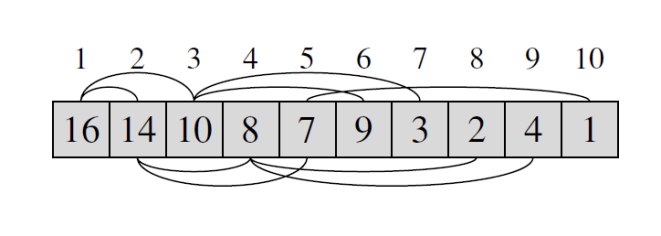

Each node of the tree corresponds to an element of the array.
The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

- An array A that represents a heap is an object with two attributes:
  - A.length, which gives the number of elements in the array;
  - A.heap-size, which represents how many elements in the heap are stored within array A.
- I.e., although A[1 . . . A.length] may contain numbers, only the elements in A[1 . . . A.heap-size], where $0 \leq$ A.heap-size $\leq$ A.length, are valid elements of the heap.

# Navigating the Heap

- The root of the tree is A[1].




- Given the index $i$ of a node, we can easily compute the indices of its parent, left child, and right child:

| PARENT($i$) | LEFT($i$) | RIGHT($i$) |
|---|---|---|
| `return` $\lfloor i/2 \rfloor$ | `return` $2i$ | `return` $2i + 1$ |

# Max Heaps and Min Heaps

- There are two kinds of binary heaps: max heaps and min heaps.
- They both satisfy a specific heap property:
  - **Max Heap Property**: For every node $i$ other than the root, $A[PARENT(i)] \geq A[i]$, i.e., the value of a node is at most the value of its parent.
    It follows that the largest element in a max heap is stored at the root and the subtree rooted at a node contains values no larger than than contained at the node itself.
  - **Min Heap Property**: For every node $i$ other than the root, $A[PARENT(i)] \leq A[i]$.
    The smallest element in a min-heap is at the root.
- For the heapsort algorithm, we use max-heaps.
- Min-heaps commonly implement priority queues.
- When properties apply to either max-heaps or min-heaps, we just use the term "heap".

# Height of a Heap

- Viewing a heap as a tree, we define the **height** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf.
- We define the **height** of the heap to be the height of its root.
- Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\log n)$.
- We shall see that the basic operations on heaps run in time at most proportional to the height of the tree and, thus, take $O(\log n)$ time.

# Heap Procedures

- The look at the following basic procedures and how they are used in a sorting algorithm and a priority-queue data structure:
  - The MAX-HEAPIFY procedure, which runs in $O(\log n)$ time, is the key to maintaining the max-heap property.
  - The BUILD-MAX-HEAP procedure, which runs in linear time, produces a maxheap from an unordered input array.
  - The HEAPSORT procedure, which runs in $O(n \log n)$ time, sorts an array in place.
  - The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM procedures, which run in $O(\log n)$ time, allow the heap data structure to implement a priority queue.

Subsection 2

# Maintaining the Heap Property

# Description of the Procedure MAX-HEAPIFY

- In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY.
- Its inputs are an array A and an index i into the array.
- When called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT($i$) and RIGHT($i$) are maxheaps, but that A[$i$] might be smaller than its children, thus violating the max-heap property.
- MAX-HEAPIFY lets the value at A[$i$] "float down" in the max-heap so that the subtree rooted at index $i$ obeys the max-heap property.

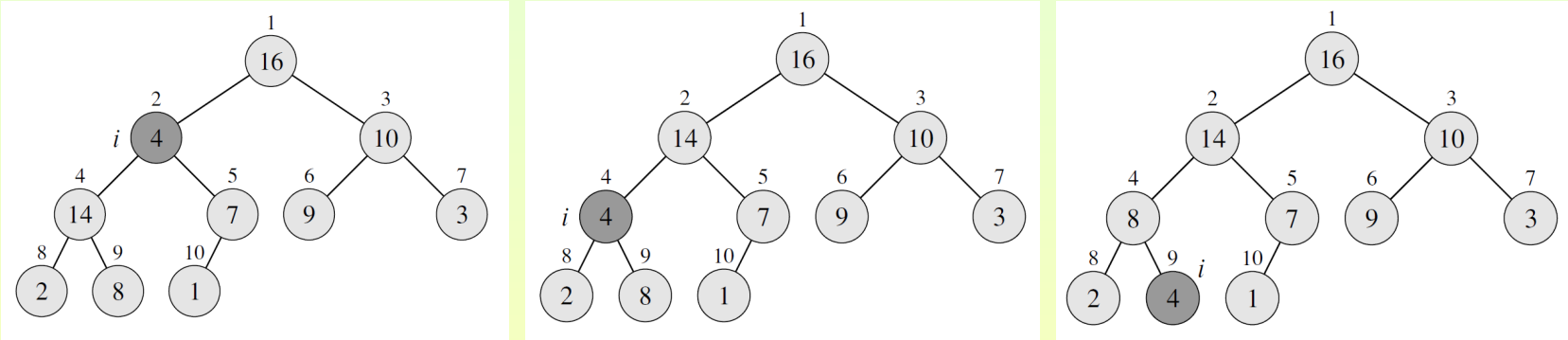
# The Procedure MAX-HEAPIFY

## MAX-HEAPIFY(A, $i$)

1. $\ell = \text{LEFT}(i)$
2. $r = \text{RIGHT}(i)$
3. `if` $\ell \leq$ A.heap-size `and` A[$\ell$] $>$ A[$i$]
4. largest $= \ell$
5. `else` largest $= i$
6. `if` $r \leq$ A.heap-size `and` A[$r$] $>$ A[largest]
7. largest $= r$
8. `if` largest $\neq i$
9. exchange A[$i$] with A[largest]
10. MAX-HEAPIFY(A, largest)

# Illustration of MAX-HEAPIFY

- At each step, the largest of the elements A[$i$], A[LEFT($i$)] and A[RIGHT($i$)] is determined, and its index is stored in largest.
  - If A[$i$] is largest, then the subtree rooted at node $i$ is already a max-heap and the procedure terminates.
  - Otherwise, one of the two children has the largest element, and A[$i$] is swapped with A[largest], which causes node $i$ and its children to satisfy the max-heap property.




The node indexed by largest has the original value A[$i$] and, hence, the subtree rooted at largest might violate the max-heap property. So MAX-HEAPIFY is called recursively on that subtree.

# The Running Time of MAX-HEAPIFY

- The running time of MAX-HEAPIFY on a subtree of size $n$ rooted at given node $i$ consists of
  - the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$,
  - plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node $i$.
    The children's subtrees each have size at most $\frac{2n}{3}$. The worst case occurs when the last row of the tree is exactly half full.

  The running time of MAX-HEAPIFY can therefore be described by the recurrence

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1).$$

  The solution to this recurrence is $T(n) = O(\log n)$.
- Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height $h$ as $O(h)$.

Subsection 3

# Building a Heap

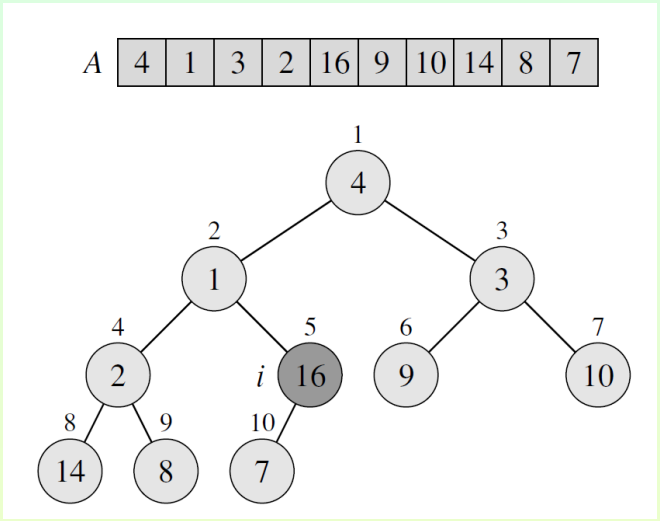
# Building a Max Heap

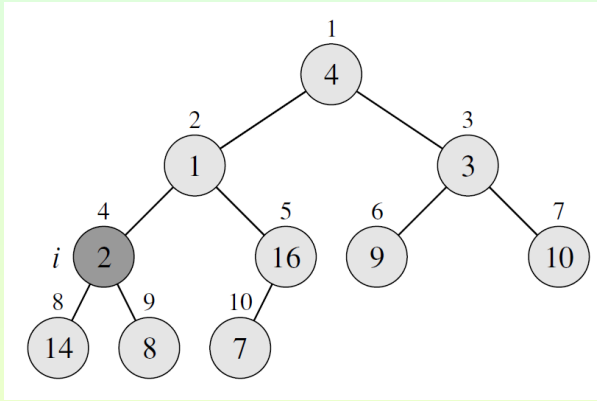
- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array $A[1 \ldots n]$, where $n = \text{length}[A]$, into a max-heap.
- The elements in the subarray $A[(\lfloor \frac{n}{2} \rfloor + 1) \ldots n]$ are all leaves of the tree, and so each is a 1-element heap to begin with.
- The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one:

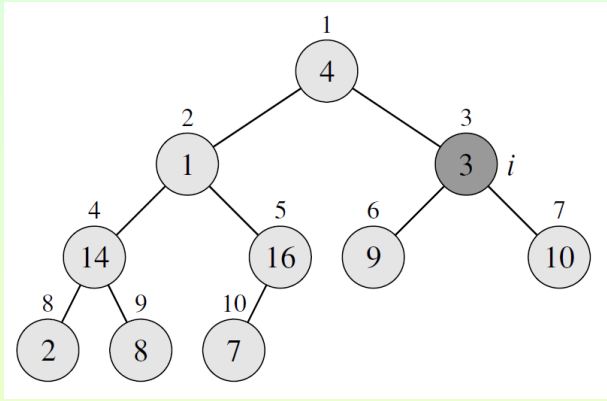
## BUILD-MAX-HEAP(A)

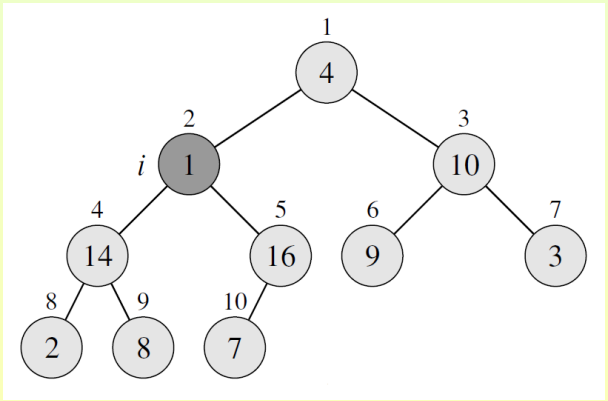
1. heap-size[A] $\leftarrow$ length[A]
2. `for` $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ `downto` 1
3. `do` MAX-HEAPIFY$(A, i)$

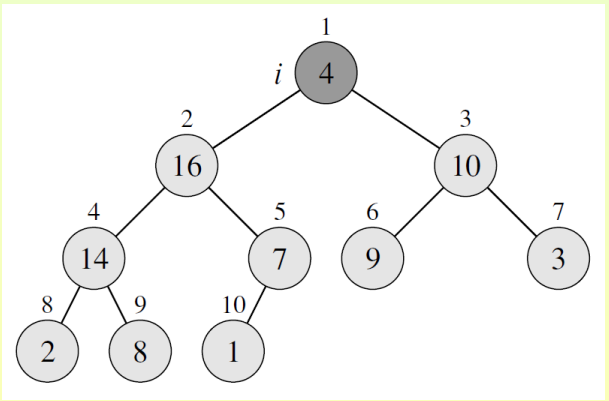
# Example of the Action of BUILD-MAX-HEAP

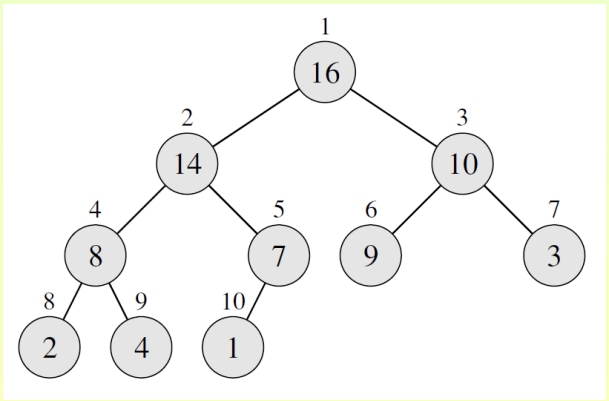










# Correctness of BUILD-MAX-HEAP(A)

- To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

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

  We need to show that:

  - This invariant is true prior to the first loop iteration.
  - Each iteration of the loop maintains the invariant.
  - The invariant provides a useful property to show correctness when the loop terminates.

# Correctness of BUILD-MAX-HEAP(A) (Cont'd)

- Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$.

  Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf.

  It is, thus, the root of a trivial max-heap.
- Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node $i$ are numbered higher than $i$. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY$(A, i)$ to make node $i$ a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1, i + 2, \ldots, n$ are all roots of max-heaps. Decrementing $i$ in the for loop update reestablishes the loop invariant for the next iteration.
- Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.

# Running Time of BUILD-MAX-HEAP(A)

- We can compute a simple upper bound on the running time of BUILD-MAX-HEAP:

  Each call to MAX-HEAPIFY costs $O(\log n)$ time, and there are $O(n)$ such calls. Thus, the running time is $O(n \log n)$.
- This upper bound is not asymptotically tight. To derive a tighter bound, observe that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Note, also, that an $n$-element heap has:
  - height $\lfloor \log n \rfloor$;
  - at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of any height $h$.

  The time required by MAX-HEAPIFY when called on a node of height $h$ is $O(h)$. Thus, we can express the total cost of BUILD-MAX-HEAP as $\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$. But $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$. So, we obtain the bound $O(n)$. Thus a max-heap can be built from an unordered array in linear time.

Subsection 4

## The Heapsort Algorithm

# Description of the Heapsort Algorithm

- The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array $A[1 \ldots n]$, where $n = \text{length}[A]$.
- Since the maximum element of the array is stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$.
- If we now "discard" node $n$ from the heap (by decrementing heap-size[A]), we observe that $A[1 \ldots (n-1)]$ can easily be made into a max-heap:

  The children of the root remain max-heaps, but the new root element may violate the max-heap property. All that is needed to restore the max-heap property, however, is one call to MAX-HEAPIFY$(A, 1)$, which leaves a max-heap in $A[1 \ldots (n-1)]$.
- The heapsort algorithm then repeats this process for the maxheap of size $n-1$ down to a heap of size 2.

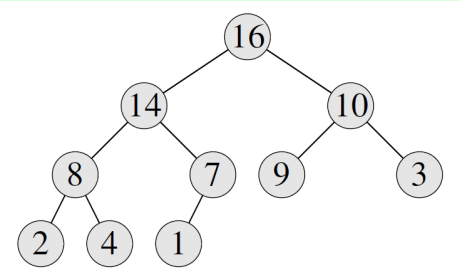
# The Heapsort Algorithm

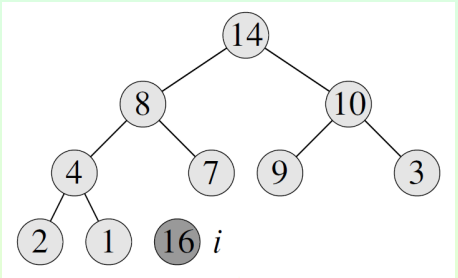
## HeapSort(A)

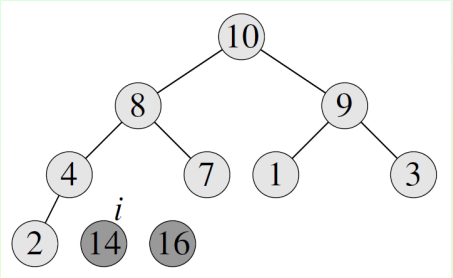
1. Build-Max-Heap(A)
2. `for` $i \leftarrow$ length[A] `downto` 2
3. `do` exchange A[1] $\leftrightarrow$ A[$i$]
4. heap-size[A] $\leftarrow$ heap-size[A] $-$ 1
5. Max-Heapify(A, 1)

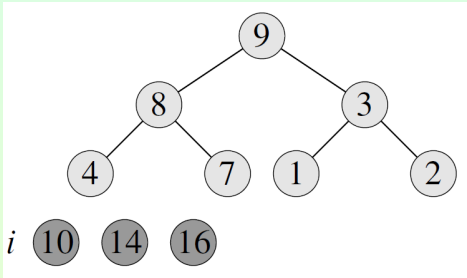
Complexity: The HeapSort procedure takes time $O(n \log n)$, since the call to Build-Max-Heap takes time $O(n)$ and each of the $n-1$ calls to Max-Heapify takes time $O(\log n)$.

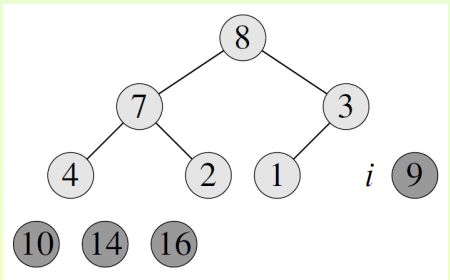
# Illustration of Heap Sort

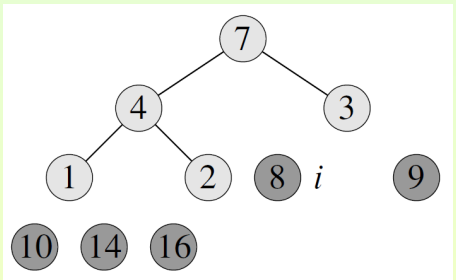


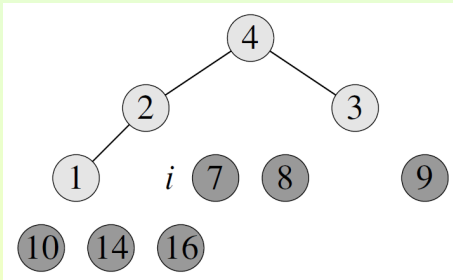



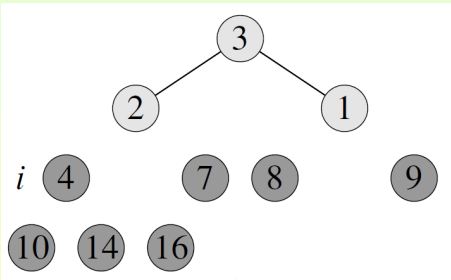



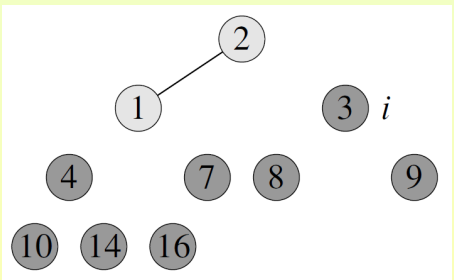



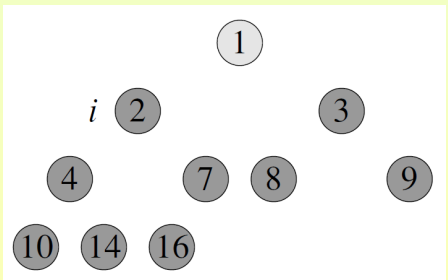




| $A$ | 1 | 2 | 3 | 4 | 7 | 8 | 9 | 10 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|

Subsection 5

# Priority Queues

# Priority Queues

- One of the most popular applications of a heap is as an efficient priority queue.
- As with heaps, priority queues come in two forms: max-priority queues and min-priority queues.

  We focus on max-priority queues, which are based on maxheaps;
- A **priority queue** is a data structure for maintaining a set $S$ of elements, each with an associated value called a **key**.
- A **max-priority queue** supports the following operations:
  - $\textsc{Insert}(S, x)$ inserts the element $x$ into the set $S$, which is equivalent to the operation $S = S \cup \{x\}$.
  - $\textsc{Maximum}(S)$ returns the element of $S$ with the largest key.
  - $\textsc{Extract-Max}(S)$ removes and returns the element of $S$ with the largest key.
  - $\textsc{Increase-Key}(S, x, k)$ increases the value of element $x$'s key to the new value $k$, which is assumed to be at least as large as $x$'s current key value.

# Sample Application: Scheduling

- We can use max-priority queues to schedule jobs on a shared computer.
  - The max-priority queue keeps track of the jobs to be performed and their relative priorities.
  - When a job is finished or interrupted, the scheduler selects the highest-priority pending job by calling EXTRACT-MAX.
  - The scheduler can add a new job to the queue by calling INSERT.

# Sample Application: Event-Driven Simulator

- A min-priority queue supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY.
- A min-priority queue can be used in an event-driven simulator.
  - The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key.
    They must be simulated in order of time of occurrence, because the simulation of an event can cause other events to be simulated in the future.
  - The simulation program calls EXTRACT-MIN at each step to choose the next event to simulate.
  - As new events are produced, the simulator inserts them into the min-priority queue by calling INSERT.

# Heap-Maximum

- The procedure Heap-Maximum implements the Maximum operation in $\Theta(1)$ time.

## Heap-Maximum(A)

1. `return` A[1]

# Heap-Extract-Max

- The procedure Heap-Extract-Max implements the Extract-Max operation.

## Heap-Extract-Max(A)

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

- The running time of Heap-Extract-Max is $O(\log n)$, since it performs only a constant amount of work on top of the $O(\log n)$ time for Max-Heapify.

# Description of Heap-Increase-Key

- The procedure Heap-Increase-Key implements the Increase-Key operation.
- An index $i$ into the array identifies the priority-queue element whose key we wish to increase.
  - The procedure first updates the key of element $A[i]$ to its new value.
  - Because increasing the key of $A[i]$ might violate the max-heap property, the procedure then traverses a simple path from this node toward the root to find a proper place for the newly increased key.
    As Heap-Increase-Key traverses this path, it repeatedly compares an element to its parent:
    - If the element's key is larger, it exchanges their keys and continues;
    - If the element's key is smaller, it terminates, since the max-heap property now holds.

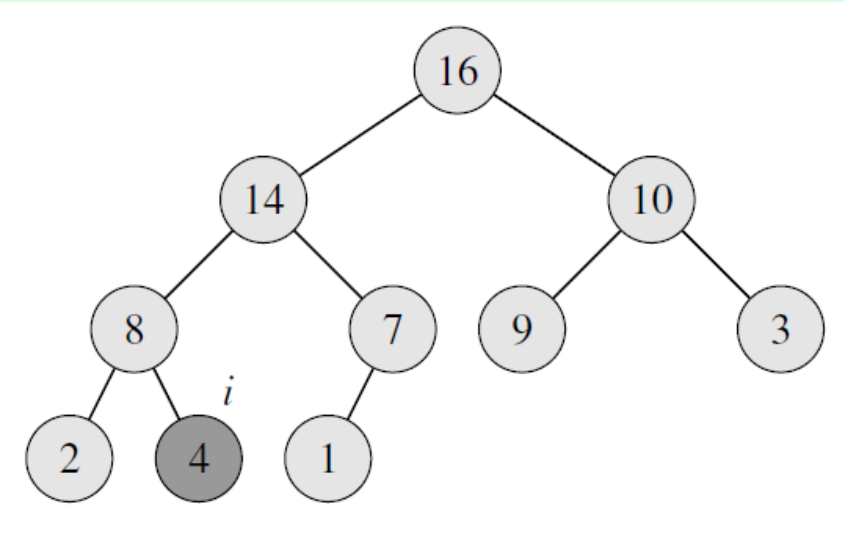
# Heap-Increase-Key

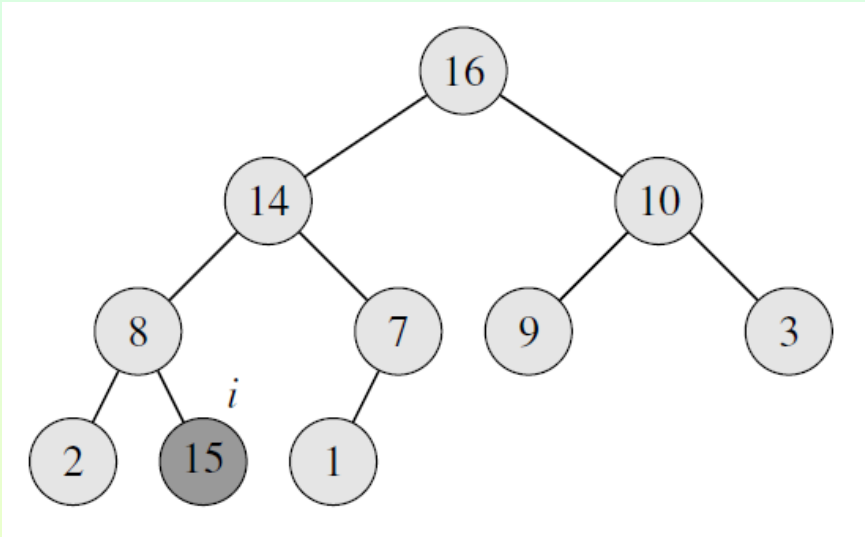
## Heap-Increase-Key(A, $i$, key)

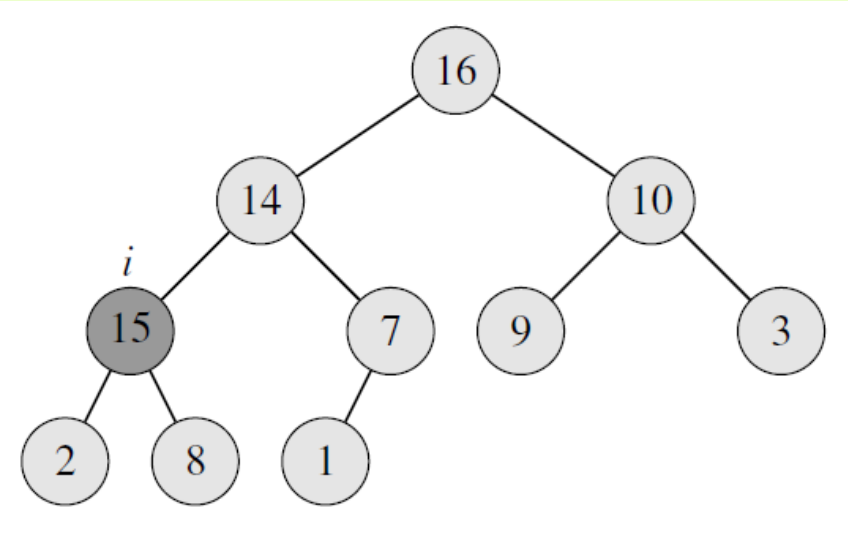
1. `if` key $<$ A[$i$]
2. `error` "new key is smaller than current key"
3. A[$i$] $=$ key
4. `while` $i > 1$ `and` A[Parent($i$)] $<$ A[$i$]
5. exchange A[$i$] with A[Parent($i$)]
6. $i =$ Parent($i$)

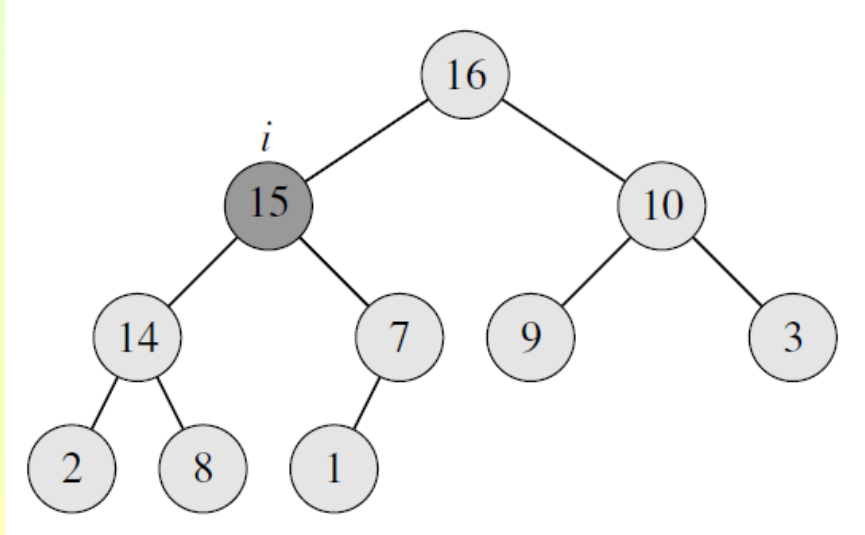
- The running time of Heap-Increase-Key on an $n$-element heap is $O(\log n)$, since the path traced from the node updated in line 3 to the root has length $O(\log n)$.

# Illustration of HEAP-INCREASE-KEY



# Max-Heap-Insert

- The procedure Max-Heap-Insert implements the Insert operation.
- It takes as an input the key of the new element to be inserted into max-heap A.
  - It first expands the max-heap by adding to the tree a new leaf whose key is $-\infty$.
  - Then it calls Heap-Increase-Key to set the key of this new node to its correct value and maintain the max-heap property.

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

1. A.heap-size $=$ A.heap-size $+ 1$
2. A[A.heap-size] $= -\infty$
3. Heap-Increase-Key(A, A.heap-size, key)

- The running time of Max-Heap-Insert on an $n$-element heap is $O(\log n)$.