CSC 611: Analysis of Algorithms

Lecture 8

Heaps, Priority Queues, and HeapSort

A Job Scheduling Application

- Job scheduling
  - The key is the priority of the jobs in the queue
  - The job with the highest priority needs to be executed next
- Operations
  - Insert, remove maximum
- Data structures
  - **Priority queues**
  - Ordered array/list, unordered array/list

PQ Implementations & Cost

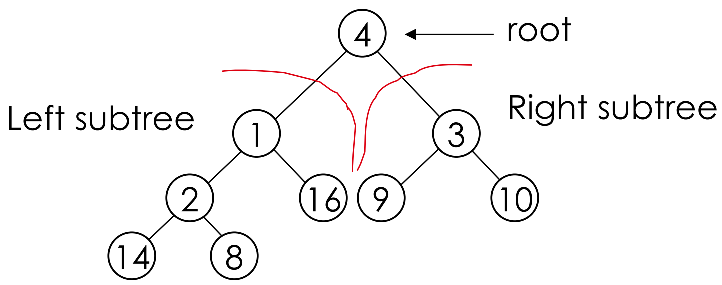
Worst-case asymptotic costs for a PQ with N items

	Insert	Remove max
ordered array	N	1
ordered list	N	1
unordered array	1	N
unordered list	1	N

Can we implement both operations efficiently?

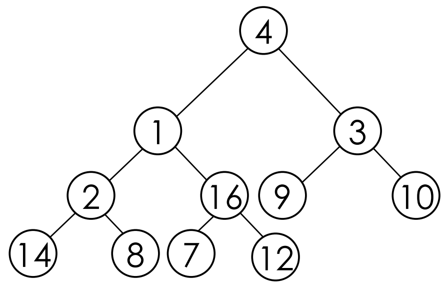
Background on Trees

- *Def:* Binary tree = structure composed of a finite set of nodes that either:
  - Contains no nodes, or
  - Is composed of three disjoint sets of nodes: a **root** node, a **left subtree** and a **right subtree**



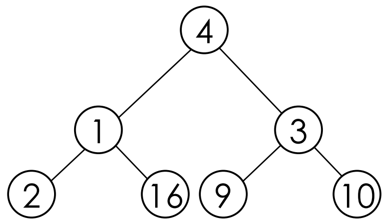
Special Types of Trees

- *Def:* **Full binary tree** = a binary tree in which each node is either a leaf or has degree (number of children) exactly 2.



Full binary tree

- *Def:* **Complete binary tree** = a binary tree in which all leaves have the same depth and all internal nodes have degree 2.

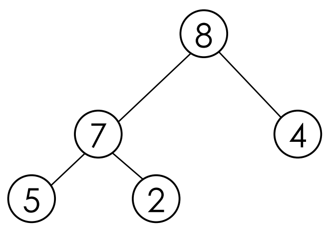


Complete binary tree

The Heap Data Structure

- *Def:* A **heap** is a nearly complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node **x**

$$\text{Parent}(x) \geq x$$

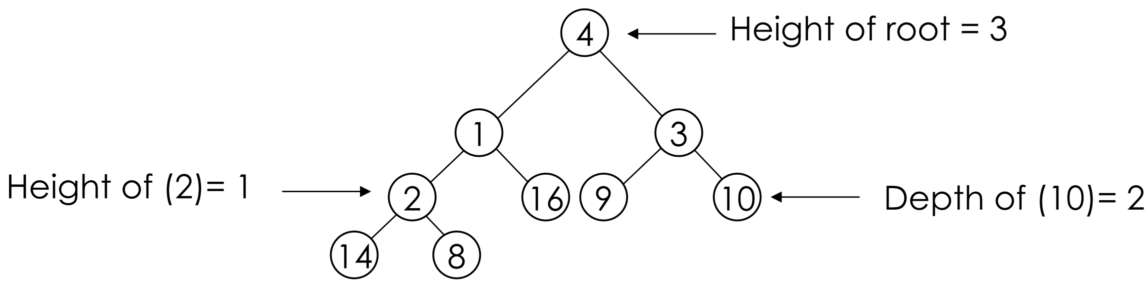


Heap

It doesn't matter that 4 in level 1 is smaller than 5 in level 2

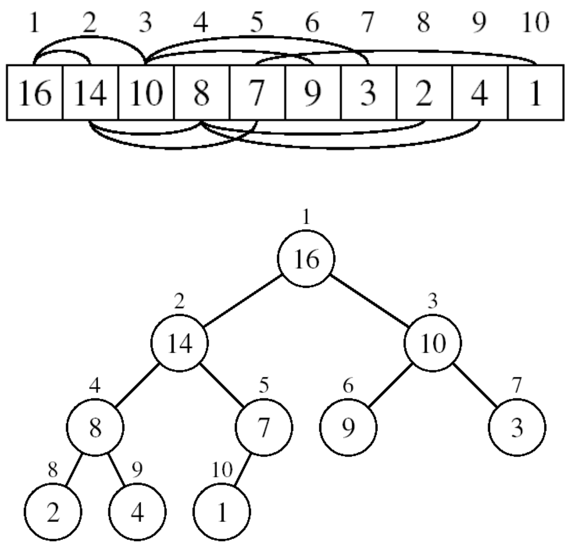
Definitions

- **Height** of a node = the number of edges on a longest simple path from the node down to a leaf
- **Depth** of a node = the length of a path from the root to the node
- **Height** of tree = height of root node
  = $\lfloor \lg n \rfloor$, for a heap of **n** elements



Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - Heapsize[A] ≤ length[A]
- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are leaves
- The root is the maximum element of the heap



Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property:*
  - for all nodes i, excluding the root:

    $A[\text{PARENT}(i)] \geq A[i]$

- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

    $A[\text{PARENT}(i)] \leq A[i]$

Operations on Heaps

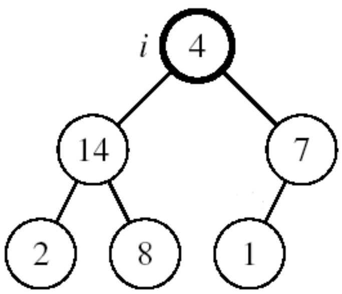
- Maintain the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT
- Priority queue operations

Operations on Priority Queues

- Max-priority queues support the following operations:
  - INSERT(S, x): inserts element x into set S
  - EXTRACT-MAX(S): removes and returns element of S with largest key
  - MAXIMUM(S): returns element of S with largest key
  - INCREASE-KEY(S, x, k): increases value of element x's key to k (assume k ≥ current key value at x)

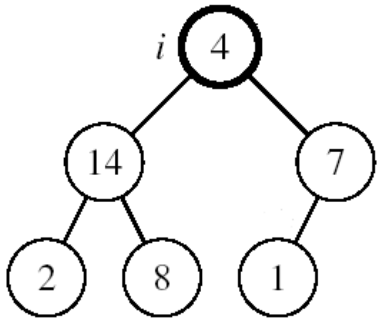
Maintaining the Heap Property

- Suppose a node is smaller than a child
  - Left and Right subtrees of i are max-heaps
- Invariant:
  - the heap condition is violated only at that node
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children



Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - A[i] may be smaller than its children

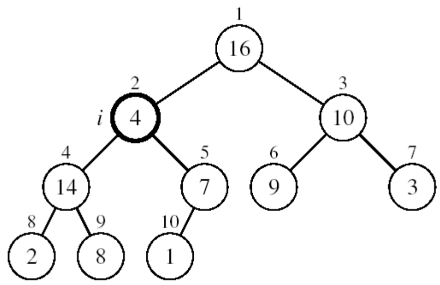


Alg: MAX-HEAPIFY(*A*, i, n)

```
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i]
   then largest ←l
   else largest ←i
if r ≤ n and A[r] > A[largest]
   then largest ←r
if largest ≠ i
   then exchange A[i] ⇔ A[largest]
          MAX-HEAPIFY(A, largest, n)
```

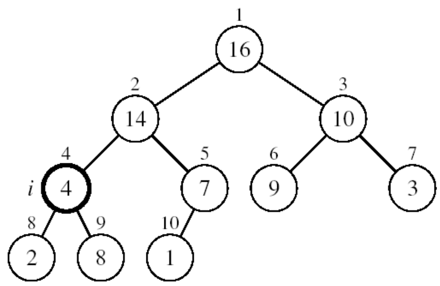

Example

MAX-HEAPIFY(A, 2, 10)



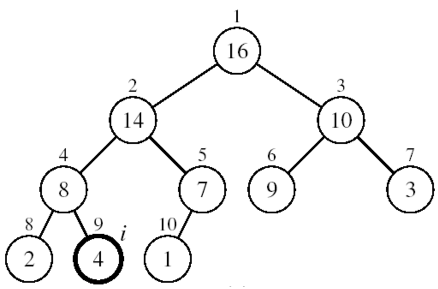
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

A[4] ↔ A[9]



Heap property restored

MAX-HEAPIFY Running Time

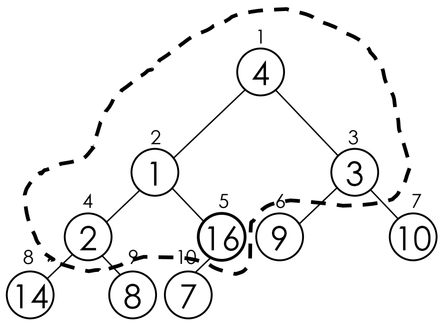
- Intuitively:
  - A heap is an almost complete binary tree ⇒ must process ***O*(lgn)** levels, with constant work at each level
- Running time of MAX-HEAPIFY is ***O*(lgn)**
- Can be written in terms of the height of the heap, as being ***O*(h)**
  - Since the height of the heap is $\lfloor lgn \rfloor$

Building a Heap

- Convert an array ***A*[1 … n]** into a max-heap (**n = length[*A*]**)
- The elements in the subarray ***A*[($\lfloor n/2 \rfloor$+1) .. n]** are leaves
- Apply MAX-HEAPIFY on elements between **1** and $\lfloor n/2 \rfloor$

Alg: BUILD-MAX-HEAP(*A*)

1. **n** = length[A]
2. **for** i ← $\lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(*A*, i, n)



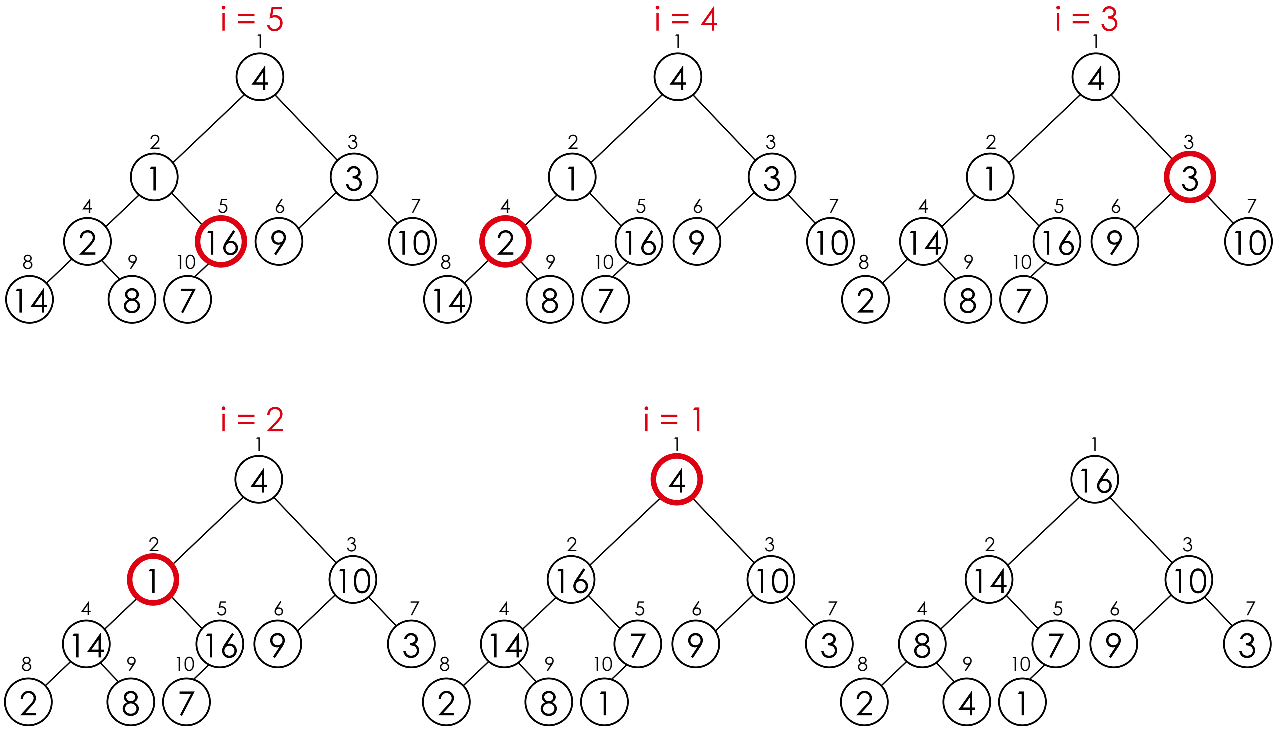
A:

4	1	3	2	16	9	10	14	8	7

Example:

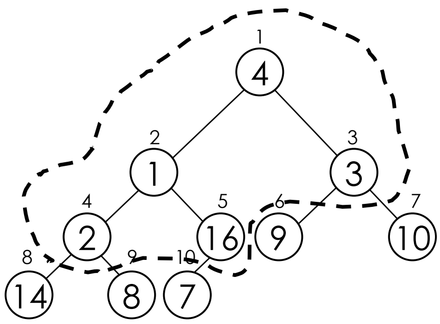
A

4	1	3	2	16	9	10	14	8	7



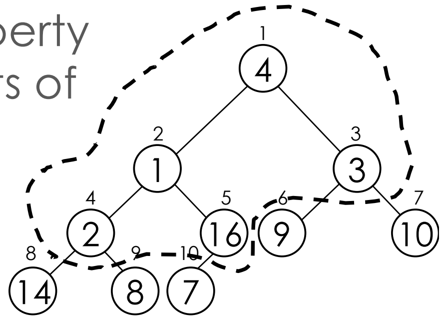
Correctness of BUILD-MAX-HEAP

- **Loop invariant:**
  - At the start of each iteration of the **for** loop, each node **i + 1**, **i + 2**,..., **n** is the root of a max-heap
- **Initialization:**
  - $i = \lfloor n/2 \rfloor$: Nodes $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, ..., n are leaves ⇒ they are the root of trivial max-heaps



Correctness of BUILD-MAX-HEAP

- **Maintenance:**
  - MAX-HEAPIFY makes node **i** a max-heap root and preserves the property that nodes **i + 1**, **i + 2**, …, **n** are roots of max-heaps
  - Decrementing **i** in the for loop reestablishes the loop invariant



- **Termination:**
  - i = 0 ⇒ each node 1, 2, …, n is the root of a max-heap (by the loop invariant)

Running Time of BUILD MAX HEAP

Alg: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n) — O(lgn)

(lines 2–3: O(n))

⇒ It would seem that running time is **O(nlgn)**

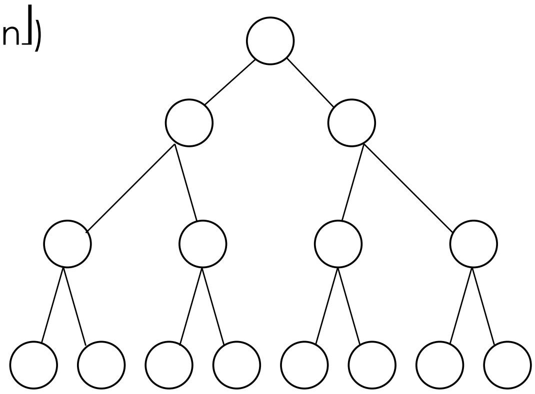
- This is not an asymptotically tight upper bound

Running Time of BUILD MAX HEAP

- HEAPIFY takes **O(h)** ⇒ the cost of HEAPIFY on a node **i** is proportional to the height of the node **i** in the tree

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h-i) = O(n)$$

Height	Level	No. of nodes
$h_0 = 3$ ($\lfloor \lg n \rfloor$)	i = 0	2^0
$h_1 = 2$	i = 1	2^1
$h_2 = 1$	i = 2	2^2
$h_3 = 0$	i = 3 ($\lfloor \lg n \rfloor$)	2^3



$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

Running Time of BUILD MAX HEAP

$$T(n) = \sum_{i=0}^{h} n_i h_i$$ Cost of HEAPIFY at level i × number of nodes at that level

$$= \sum_{i=0}^{h} 2^i (h-i)$$ Replace the values of n_i and h_i computed before

$$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$ Multiply by 2^h both at the numerator and denominator and write 2^i as $\frac{1}{2^{-i}}$

$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$ Change variables: k = h - i

$$\le n \sum_{k=0}^{\infty} \frac{k}{2^k}$$ The sum above is smaller than the sum of all elements to ∞ and h = lgn

$$= O(n)$$ The sum above is smaller than 2

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$
for $|x| < 1$.

Running time of BUILD-MAX-HEAP: T(n) = O(n)

Operations on Priority Queues

- Max-priority queues support the following operations:
  - INSERT(*S*, **x**): inserts element **x** into set *S*
  - EXTRACT-MAX(*S*): removes and returns element of *S* with largest key
  - MAXIMUM(*S*): returns element of *S* with largest key
  - INCREASE-KEY(*S*, **x**, **k**): increases value of element **x**'s key to **k** (assume **k** ≥ current key value at **x**)

HEAP-MAXIMUM

Goal:

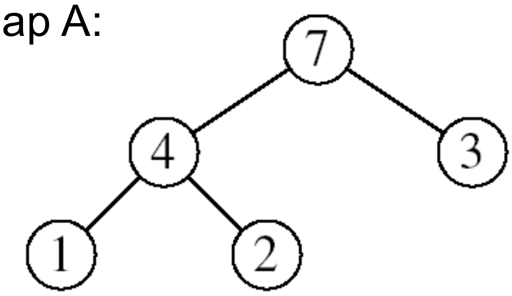
– Return the largest element of the heap

Alg: HEAP-MAXIMUM(*A*)

1. **return** *A*[1]

Running time: *O*(1)

Heap A:



Heap-Maximum(A) returns 7

HEAP-EXTRACT-MAX

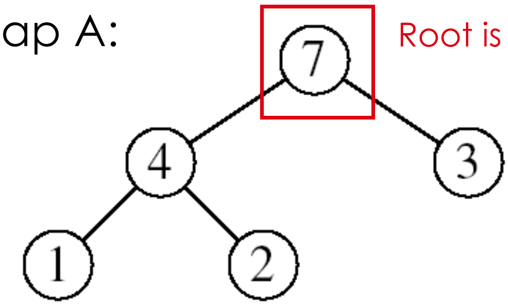
Goal:

- Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap

Idea:

- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size n-1

Heap A:

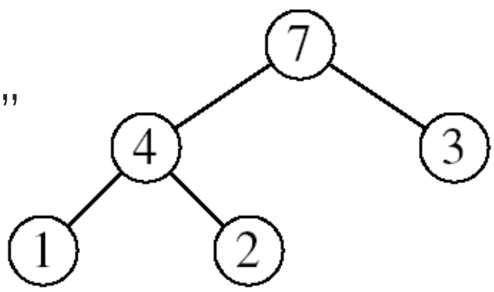


Root is the largest element

HEAP-EXTRACT-MAX

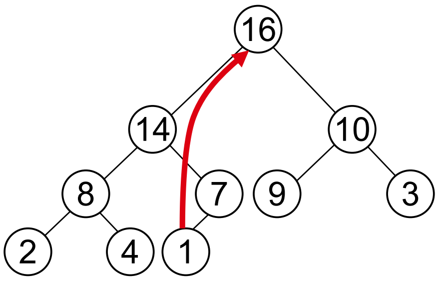
Alg: HEAP-EXTRACT-MAX(*A*, n)

1. **if** n < 1
2. **then error** "heap underflow"
3. max ← *A*[1]
4. *A*[1] ← *A*[n]
5. MAX-HEAPIFY(*A*, 1, n-1) ▷ remakes heap
6. **return** max

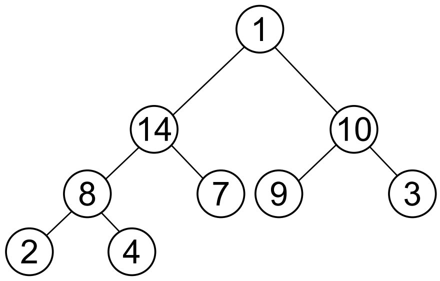


Running time: *O*(lgn)

Example: HEAP-EXTRACT-MAX

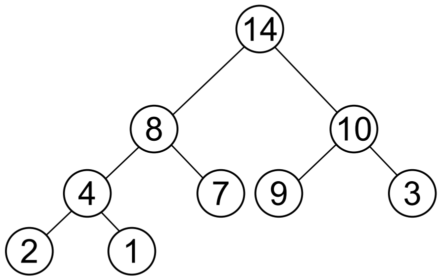


max = 16



Heap size decreased with 1

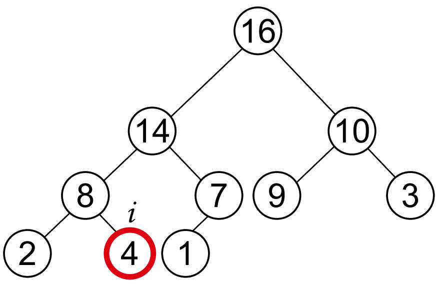
Call MAX-HEAPIFY(*A*, 1, n-1)



HEAP-INCREASE-KEY

- Goal:
  - Increases the key of an element i in the heap
- Idea:
  - Increment the key of **A[i]** to its new value
  - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key

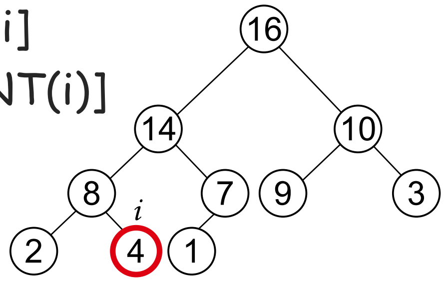
Key [i] ← 15



HEAP-INCREASE-KEY

Alg: HEAP-INCREASE-KEY(A, i, key)

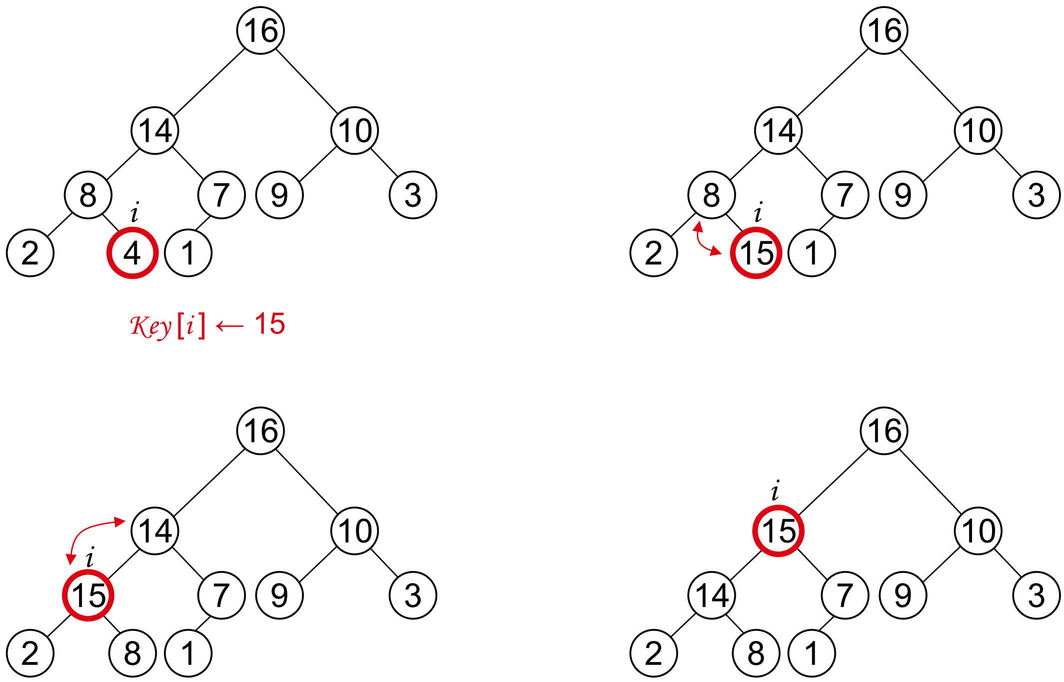
```
if key < A[i]
   then error "new key is smaller than current key"
A[i] ← key
while i > 1 and A[PARENT(i)] < A[i]
    do exchange A[i] ⇔ A[PARENT(i)]
       i ← PARENT(i)
```



Key [i] ← 15

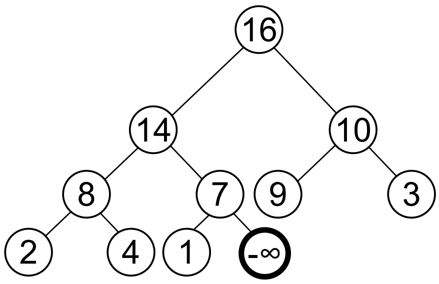
- Running time: $O(\lg n)$

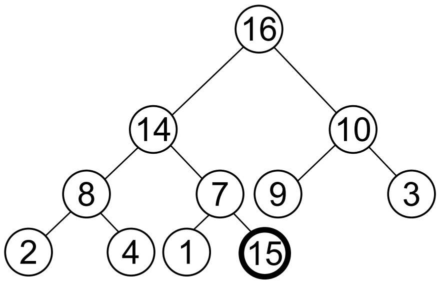
Example: HEAP-INCREASE-KEY



MAX-HEAP-INSERT

- Goal:
  - Inserts a new element into a max-heap
- Idea:
  - Expand the max-heap with a new element whose key is -∞
  - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property

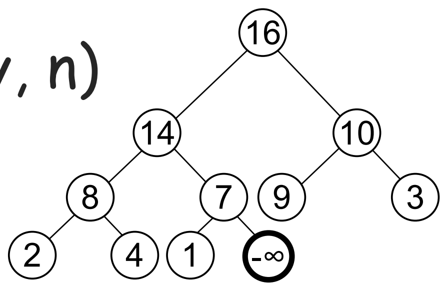




MAX-HEAP-INSERT

Alg: MAX-HEAP-INSERT(*A*, **key**, **n**)

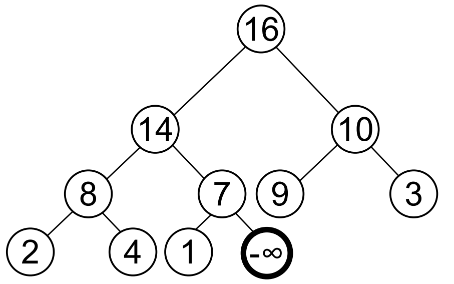
1. **heap-size[*A*]** ← **n + 1**
2. ***A*[n + 1]** ← **- ∞**
3. HEAP-INCREASE-KEY(***A*, n + 1, key**)



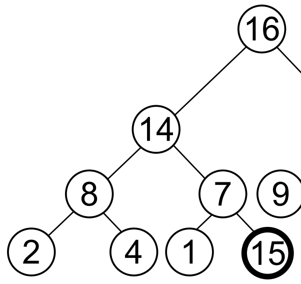
Running time: ***O*(lgn)**

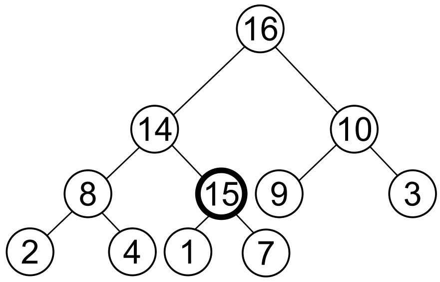
Example: MAX-HEAP-INSERT

Insert value 15:
- Start by inserting -∞

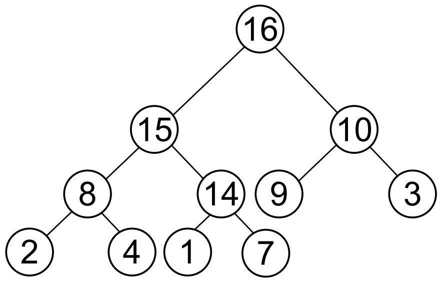


Increase the key to 15
Call HEAP-INCREASE-KEY on A[11] = 15





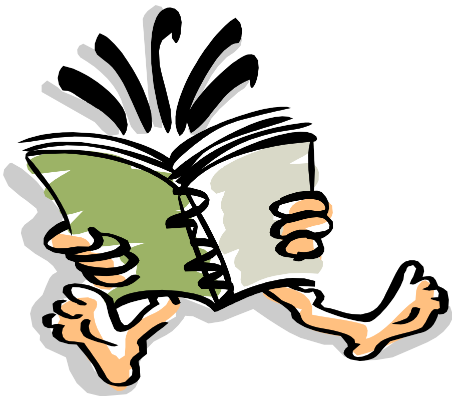
The restored heap containing the newly added element



Summary

- We can perform the following operations on heaps:
  - MAX-HEAPIFY $O(\lg n)$
  - BUILD-MAX-HEAP $O(n)$
  - HEAP-SORT $O(n\lg n)$
  - MAX-HEAP-INSERT $O(\lg n)$
  - HEAP-EXTRACT-MAX $O(\lg n)$
  - HEAP-INCREASE-KEY $O(\lg n)$
  - HEAP-MAXIMUM $O(1)$

Readings



- Chapters 14