Notes on Heapsort

Geoffrey Matthews

April 26, 2016

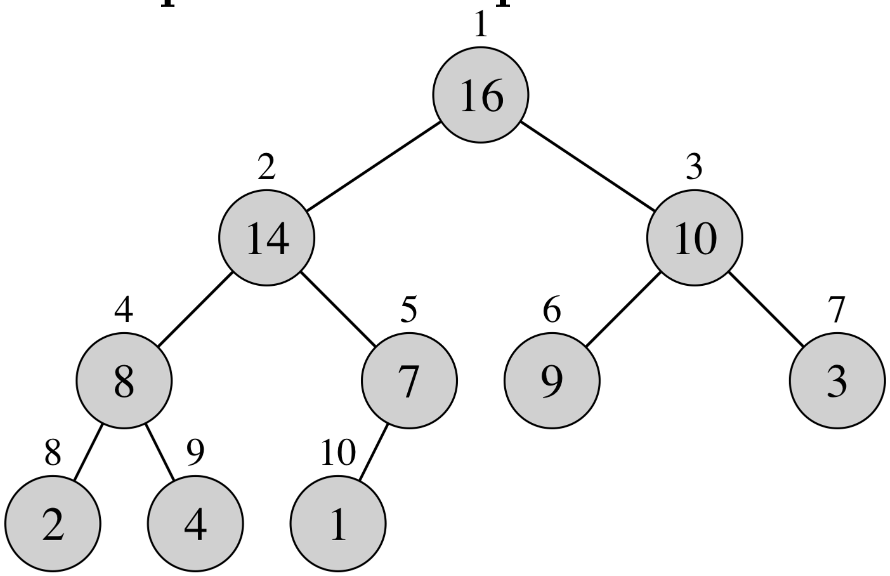
Heapsort

- $O(n \lg n)$ worst case — like merge sort
- Sorts in place — like insertion sort
- Combines best of both algorithms

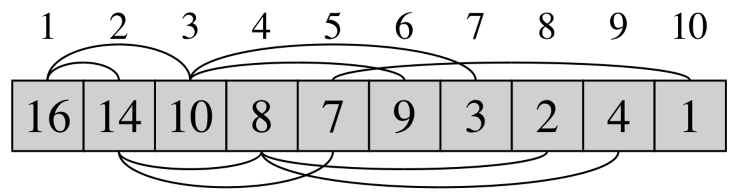
Heaps

- A nearly complete binary tree.
- **Height:** number of edges on longest path from node to leaf
- Stored as an array A
  - Root at $A[1]$
  - Left child at $A[2i]$
  - Right child at $A[2i+1]$
  - Parent of $A[i]$ at $A[\lfloor i/2 \rfloor]$
- Computing very fast

Example Max-heap



(a)



(b)

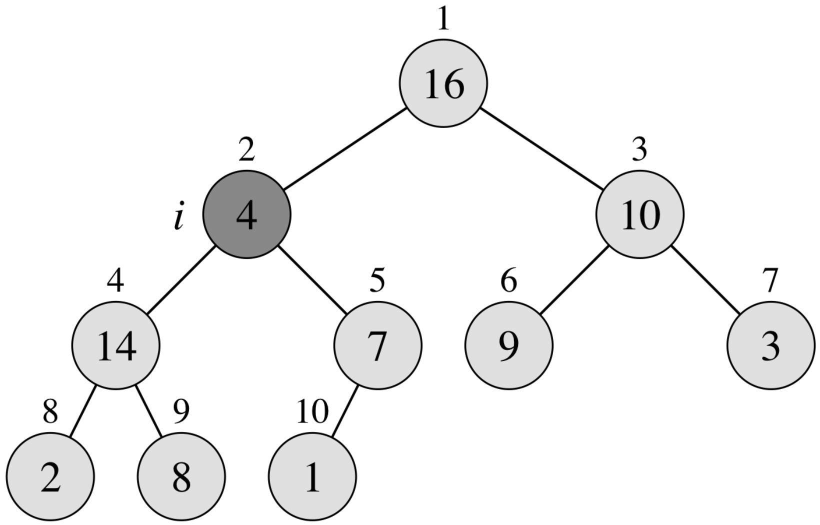
- For max-heaps: $A[\textsc{Parent}(i)] \geq A[i]$
- Induction can prove largest element is at root.

Max-Heapify

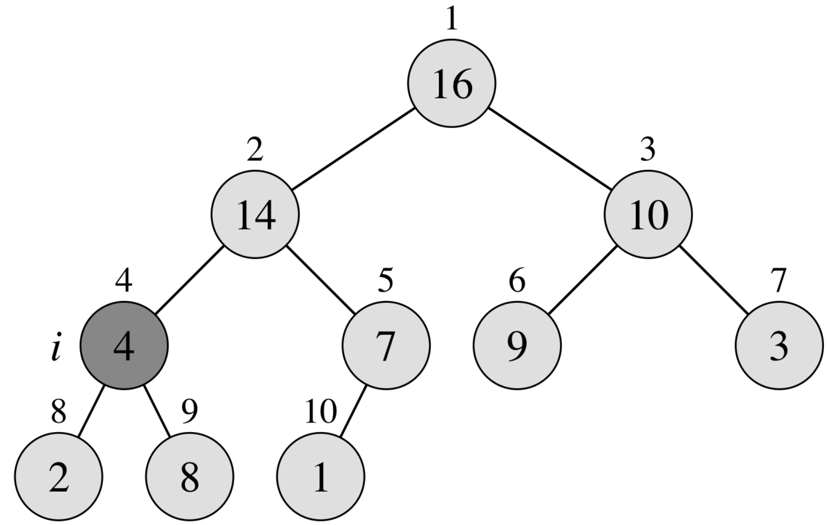
- Preconditions:
  - $A[i]$ may be smaller than its children.
  - Left and right subtrees of i are max-heaps.
- Postcondition: subtree rooted at i is a max-heap.

Max-Heapify(A, i, n)

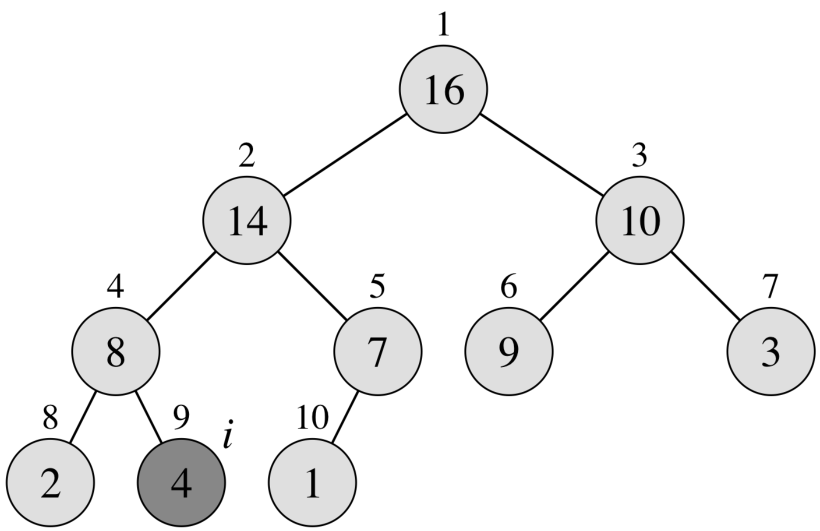
$l = $ Left(i)
$r = $ Right(i)
if $l \leq n$ and $A[l] > A[i]$
    $largest = l$
else $largest = i$
if $r \leq n$ and $A[r] > A[largest]$
    $largest = r$
if $largest \neq i$
    exchange $A[i]$ with $A[largest]$
    Max-Heapify$(A, largest, n)$


1
16
2
i
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1

(a)


1
16
2
14
3
10
4
i
4
5
7
6
9
7
3
8
2
9
8
10
1

(b)


1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
i
4
10
1

(c)

Time for `Max-Heapify`

- $O(\lg n)$
- Why?

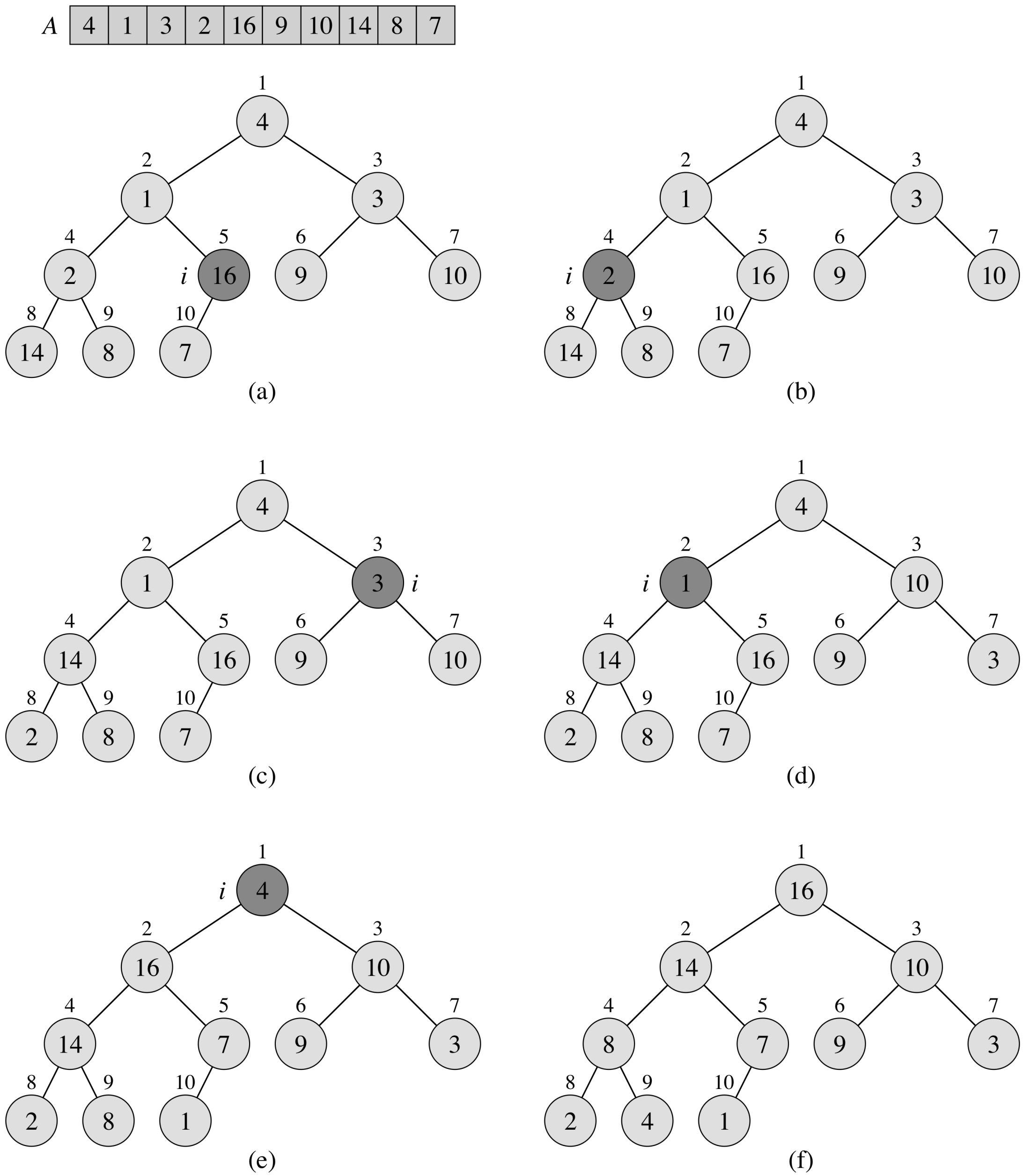
Build-Max-Heap

- If A is not a max-heap, this will make it one.

Build-Max-Heap(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

  Max-Heapify(A, i, n)


A
4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)
(e)
(f)

Loop invariant for Build-Max-Heap

- At start of every iteration of **for** loop, each node $i+1, i+2, \ldots, n$ is root of a max-heap.
  - Initialization?
  - Maintenance?
  - Termination?

Build-Max-Heap(A)

```
for i = ⌊n/2⌋ downto 1
        Max-Heapify(A, i, n)
```

Running time of BUILD-MAX-HEAP

- Loose bound: $O(n)$ calls to MAX-HEAPIFY, which is $O(\lg n)$, gives $O(n \lg n)$.
- We can get a tighter bound.
  - n element heap has height $\lfloor \lg n \rfloor$
  - n element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h
  - Time for MAX-HEAPIFY on a node of height h is $O(h)$
  - Total time for BUILD-MAX-HEAP:

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

Note:

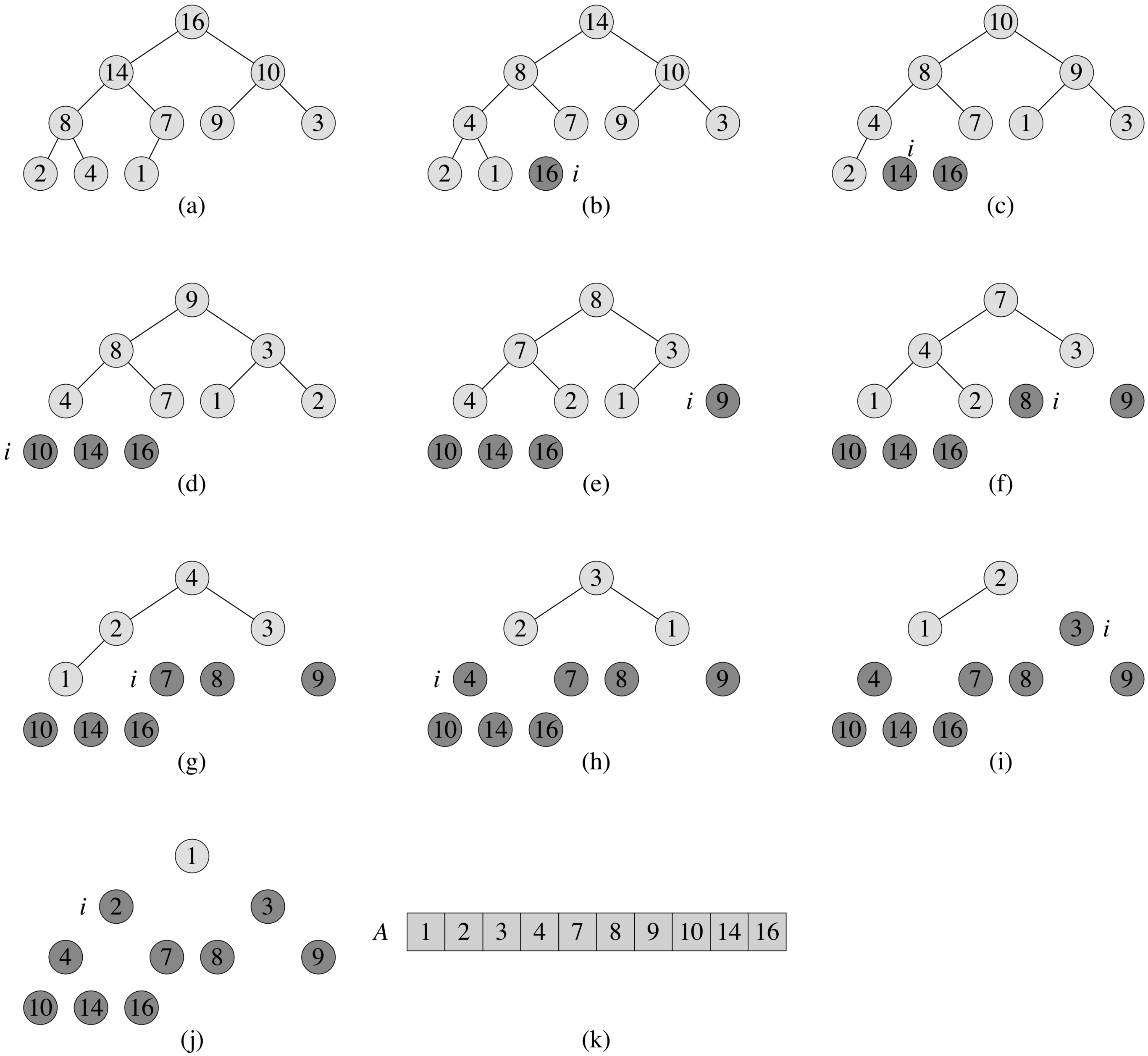
$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \tag{A.8}$$

$$\sum_{k=0}^{\infty} k(1/2)^k = \frac{1/2}{(1-1/2)^2} = 2$$

Heapsort

- Builds max-heap in the array.
- Swaps the root (the maximum) with the element at the end.
- Heapifies the result, with one less element.
- Repeat until only one element left.

```
Heapsort(A)
  Build-Max-Heap(A)
  for i = n downto 2
        exchange A[1] with A[i]
        Max-Heapify(A, 1, i − 1)
```


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
A 1 2 3 4 7 8 9 10 14 16
(k)

Heapsort

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = n downto 2
        exchange A[1] with A[i]
        MAX-HEAPIFY(A, 1, i − 1)
```

- Analysis: $O(n \lg n)$

Priority queue

- Heaps efficiently implement priority queues.
- Maintains a dynamic set S of elements.
- Each element has a *key*
- Operations:
  - $\textsc{Insert}(S, x)$
  - $\textsc{Maximum}(S)$
  - $\textsc{Extract-Max}(S)$
  - $\textsc{Increase-Key}(S, x, k)$
- Min priority queue similar

Heap-Maximum

- Trivial
- Should probably check for empty heap

Heap-Maximum(A)

return $A[1]$

- $O(1)$

Heap-Extract-Max

- Make sure heap is not empty.
- Copy the max element (root)
- Make the last node the new root.
- Decrement the size.
- Re-heapify starting at the root.
- Return the max element copy.

Heap-Extract-Max(A)

if $n < 1$
    error "heap underflow"
$max = A[1]$
$A[1] = A[n]$
$n = n - 1$
Max-Heapify($A, 1, n$)
return max

- $O(\lg n)$

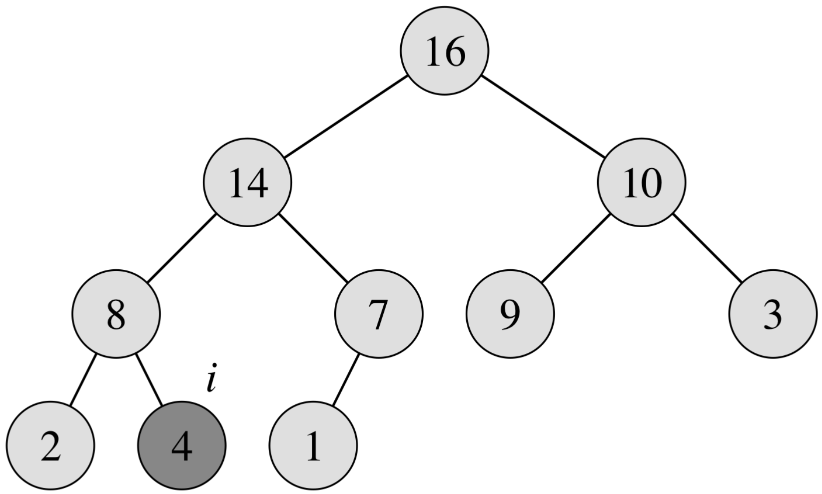
Heap-Increase-Key

- Make sure $k \geq x$'s current key
- Update x's key to k
- Traverse tree upward, swapping keys if necessary.

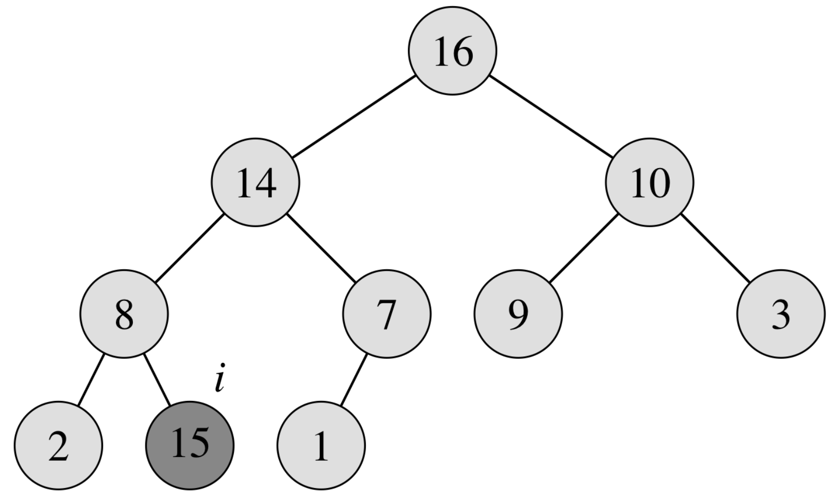
Heap-Increase-Key(A, i, key)

```
if key < A[i]
        error "new key is smaller than current key"
A[i] = key
while i > 1 and A[Parent(i)] < A[i]
        exchange A[i] with A[Parent(i)]
        i = Parent(i)
```

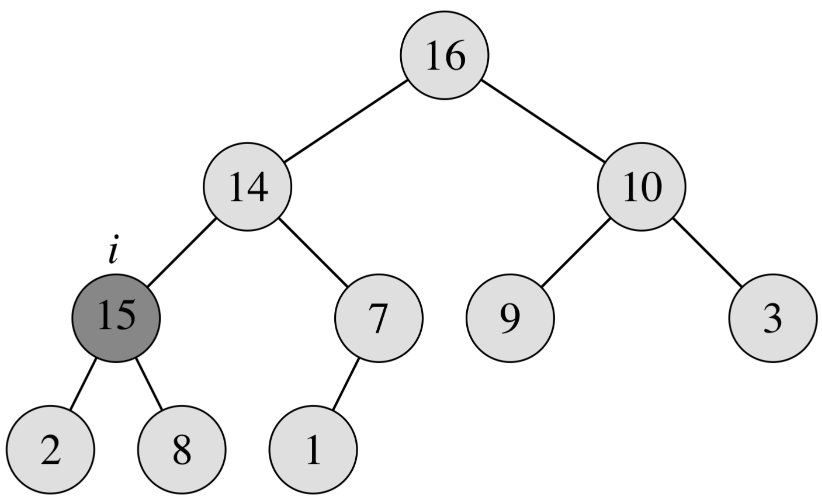
- $O(\lg n)$


16
14
10
8
7
9
3
i
2
4
1

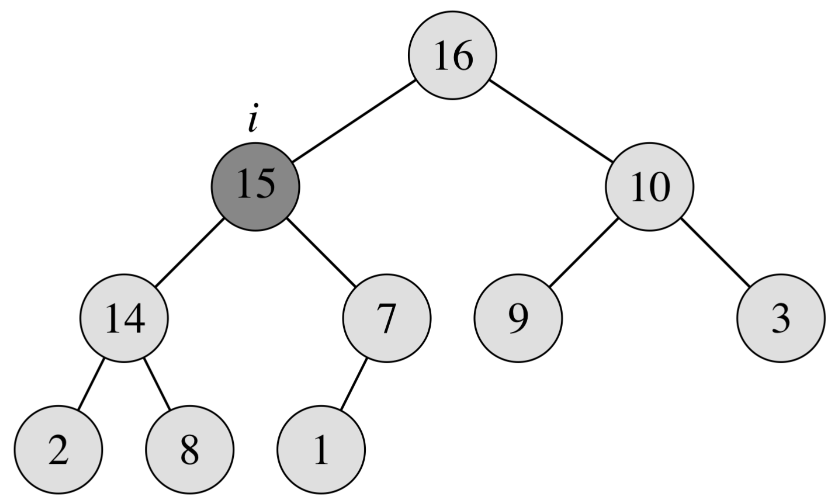
(a)


16
14
10
8
7
9
3
i
2
15
1

(b)


16
14
10
i
15
7
9
3
2
8
1

(c)


16
i
15
10
14
7
9
3
2
8
1

(d)

Why can't we `Heap-Decrease-Key` on a max-heap?

Max-Heap-Insert

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

```
n = n + 1
A[n] = -∞
Heap-Increase-Key(A, n, key)
```

- $O(\lg n)$