Data Structures and Algorithms [1]

BITS-Pilani K. K. Birla Goa Campus

[1] Material for the presentation taken from Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms, Third Edition*;

Part II Sorting and Order Statistics

- Record : Collection of data
- Key : Value to be sorted
- Satellite data
- If satellite data is large, we permute an array of pointers to the records.

Sorting algorithms

- *In place* sorting : If at any time only a constant number of elements are stored outside the array.

Sorting algorithms

- *In place* sorting : If at any time only a constant number of elements are stored outside the array.

  Auxilliary space complexity is

Sorting algorithms

- *In place* sorting : If at any time only a constant number of elements are stored outside the array.

  Auxilliary space complexity is $\Theta(1)$

Sorting algorithms

- *In place* sorting : If at any time only a constant number of elements are stored outside the array.

  Auxilliary space complexity is $\Theta(1)$
- MERGE procedure does not operate in place.

Sorting algorithms

- *In place* sorting : If at any time only a constant number of elements are stored outside the array.

  Auxilliary space complexity is $\Theta(1)$
- MERGE procedure does not operate in place.
- Ch 6 : Heapsort that uses a data structure called heap.

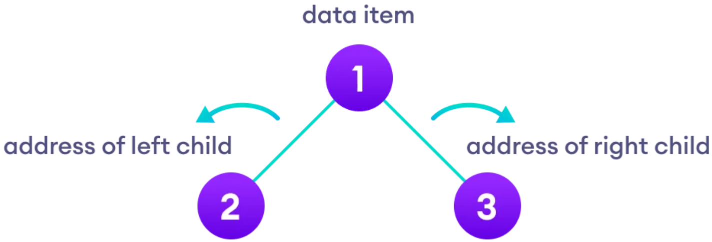
Sorting algorithms

- *In place* sorting : If at any time only a constant number of elements are stored outside the array.

  Auxilliary space complexity is $\Theta(1)$
- MERGE procedure does not operate in place.
- Ch 6 : Heapsort that uses a data structure called heap.
- Heapsort sorts n elements *in place* in $O(n \lg n)$ time.

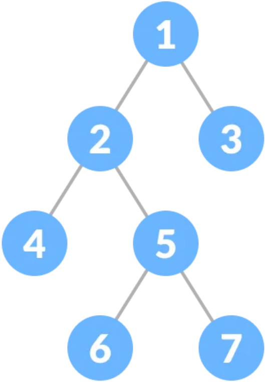
Binary tree

- Binary tree is a tree data structure where each node can have at most two child nodes : left child node and right child node.



Full Binary tree

- Each node is either a leaf node or has two child nodes.



Complete Binary tree

- A **complete binary tree** is a binary tree in which all levels are completely filled except possibly the last level, which is filled from left to right.

Complete Binary tree

- A **complete binary tree** is a binary tree in which all levels are completely filled except possibly the last level, which is filled from left to right.

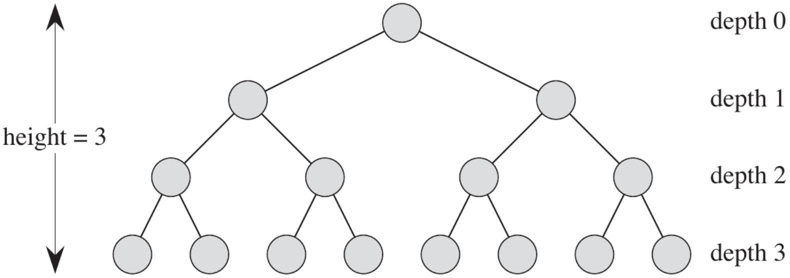
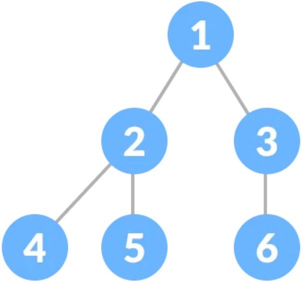
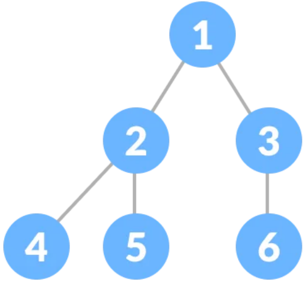


Figure B.8 A complete binary tree of height 3 with 8 leaves and 7 internal nodes.

Heap data structure : a complete binary tree



Heap data structure : a complete binary tree



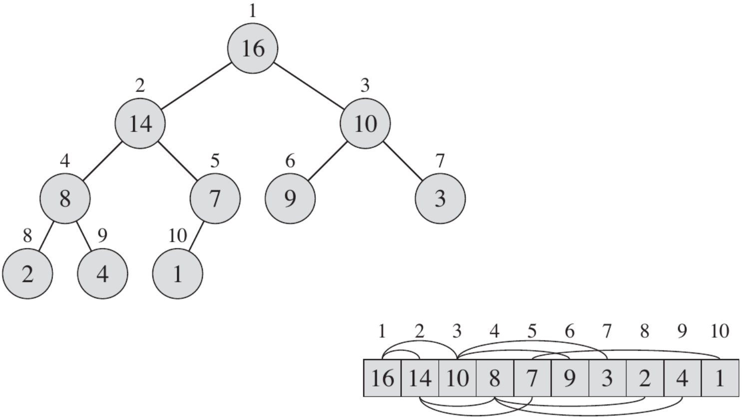
- The last level is not completely filled.

Heap : nearly a complete binary tree

- Each node of the heap corresponds to an element of the array.

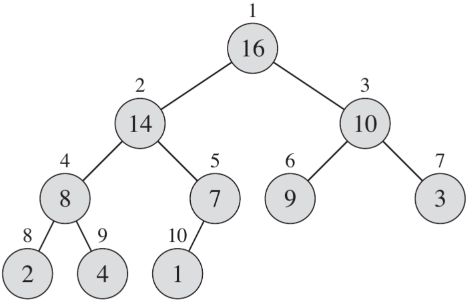
Heap : nearly a complete binary tree

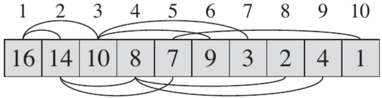
- Each node of the heap corresponds to an element of the array.



Heap : nearly a complete binary tree

- Each node of the heap corresponds to an element of the array.

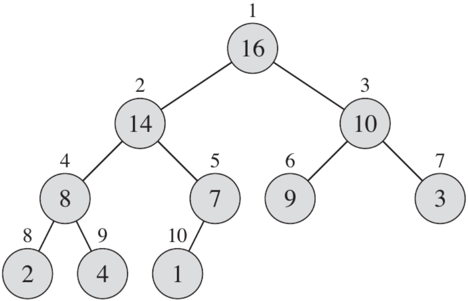


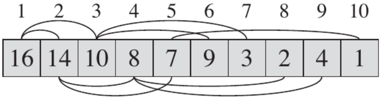


- *A.length*,

Heap : nearly a complete binary tree

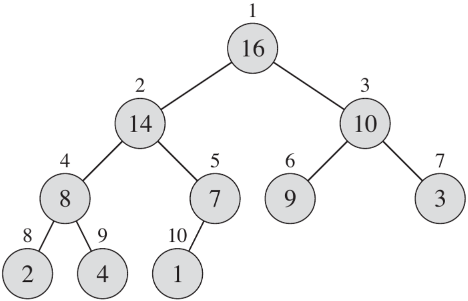
- Each node of the heap corresponds to an element of the array.

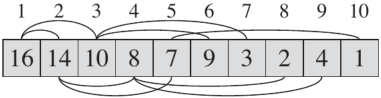




- *A.length*, *A.heap-size*,

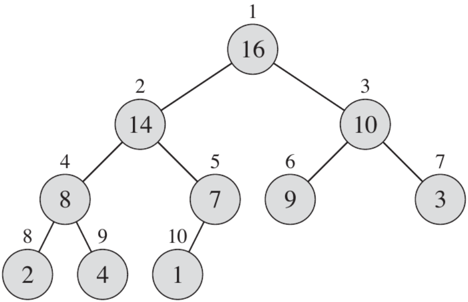
- Each node of the heap corresponds to an element of the array.





- *A.length*, *A.heap-size*, Root : *A[1]*

Heap : parent, left child, right child



PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i + 1$

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

Binary Heaps

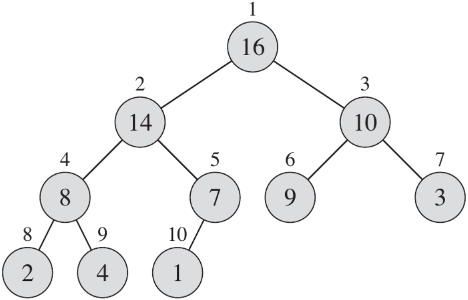
- Either max-heaps or min-heaps

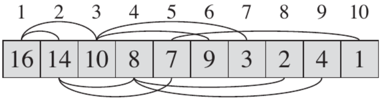
Binary Heaps

- Either max-heaps or min-heaps
- Max-heap property: $A[PARENT(i)] \geq A[i]$

Binary Heaps

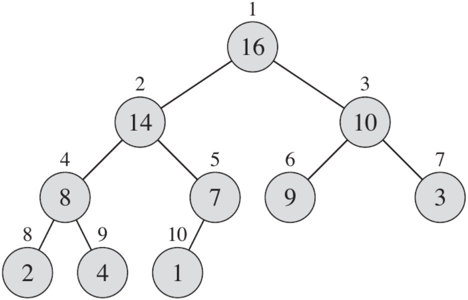
- Either max-heaps or min-heaps
- Max-heap property: $A[PARENT(i)] \geq A[i]$

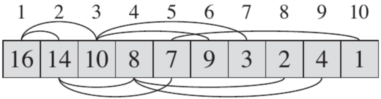




Binary Heaps

- Either max-heaps or min-heaps
- Max-heap property: $A[PARENT(i)] \geq A[i]$





- $A[1]$ contains the maximum element

Binary Heaps

- Min heap

Binary Heaps

- Min heap
- Min-heap property: $A[PARENT(i)] \leq A[i]$

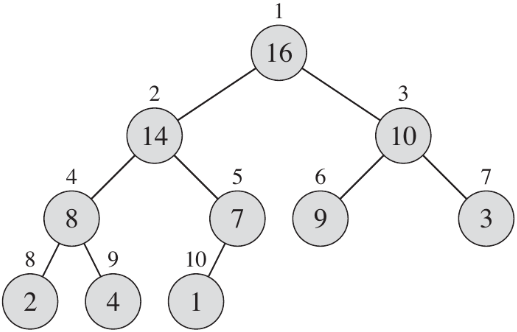
Binary Heaps

- Min heap
- Min-heap property: $A[PARENT(i)] \leq A[i]$
- $A[1]$ will contain the smallest element

Heap

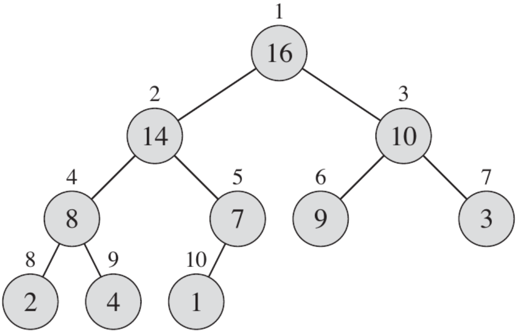
- **Height** of a node : number of edges on the *longest* simple downward path from the node to a leaf.

- **Height** of a node : number of edges on the *longest* simple downward path from the node to a leaf.



Heap

- **Height** of a node : number of edges on the *longest* simple downward path from the node to a leaf.



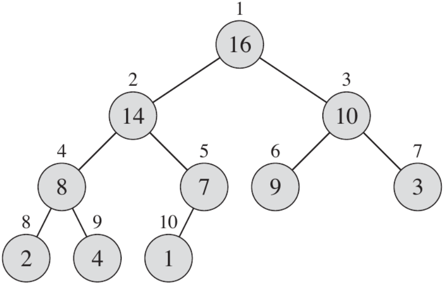
- **Height** of a heap is the height of its root.

Height of a heap

- Suppose a heap has n elements and has a height of h.

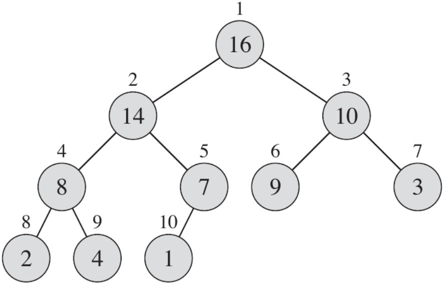
Height of a heap

- Suppose a heap has n elements and has a height of h.



Height of a heap

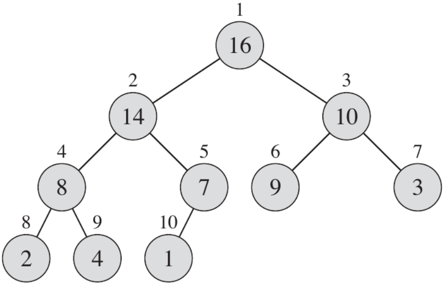
- Suppose a heap has n elements and has a height of h.



$$2^h$$

Height of a heap

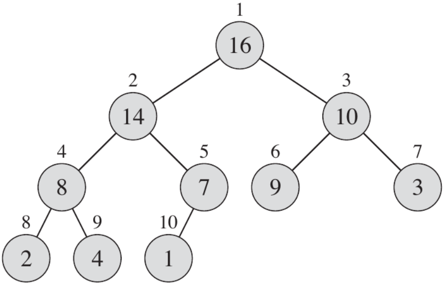
- Suppose a heap has n elements and has a height of h.



$$2^h \leq n$$

Height of a heap

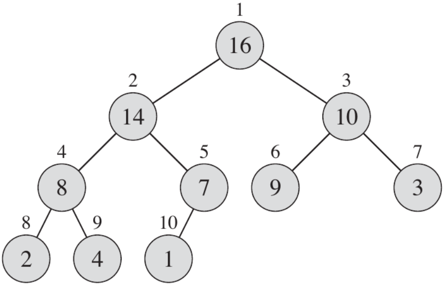
- Suppose a heap has n elements and has a height of h.



$$2^h \leq n \leq 2^{h+1} - 1$$

Height of a heap

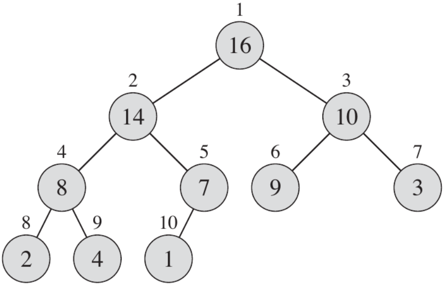
- Suppose a heap has n elements and has a height of h.



$$2^h \leq n \leq 2^{h+1} - 1$$
$$2^h \leq n < 2^{h+1}$$

Height of a heap

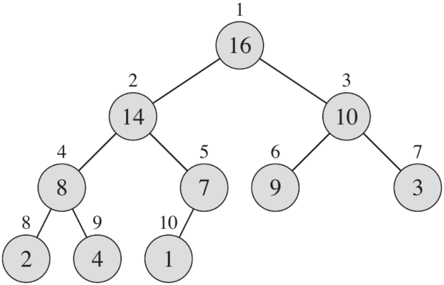
- Suppose a heap has n elements and has a height of h.



$$2^h \leq n \leq 2^{h+1} - 1$$
$$2^h \leq n < 2^{h+1}$$
$$h \leq \lg n < (h+1)$$

Height of a heap

- Suppose a heap has n elements and has a height of h.

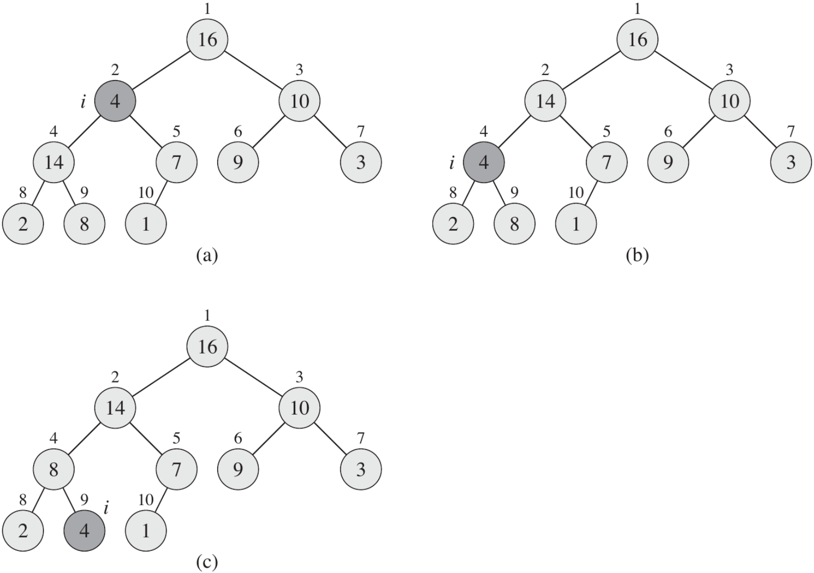


$$2^h \leq n \leq 2^{h+1} - 1$$
$$2^h \leq n < 2^{h+1}$$
$$h \leq \lg n < (h+1)$$
$$h = \lfloor \lg n \rfloor$$

MAX-HEAPIFY: Maintaining the heap property

- MAX-HEAPIFY(A,i) : binary trees rooted at LEFT(i) and RIGHT(i) satisfy max-heap property.

MAX-HEAPIFY: Maintaining the heap property



MAX-HEAPIFY: Maintaining the heap property

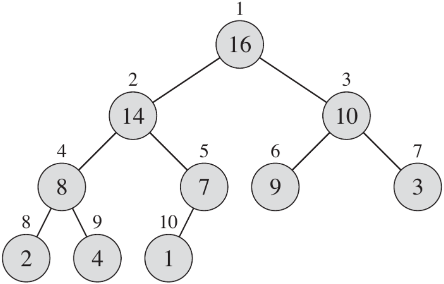
```
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

MAX-HEAPIFY: Finding recurrence

- Let the tree rooted at i have n nodes. The child subtree will have a size at most 2n/3

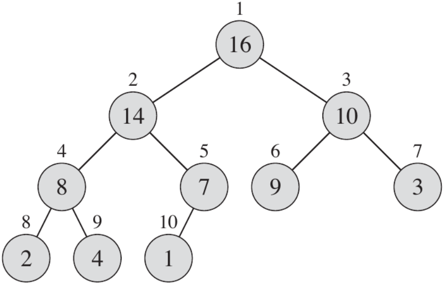
MAX-HEAPIFY: Finding recurrence

- Let the tree rooted at i have n nodes. The child subtree will have a size at most 2n/3



MAX-HEAPIFY: Finding recurrence

- Let the tree rooted at i have n nodes. The child subtree will have a size at most $2n/3$

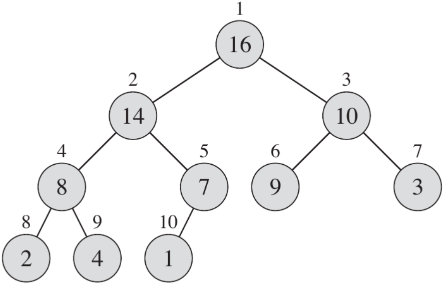


$$h = \text{ height of node } i$$

$$\text{Maximum size of child subtree } = 2^h - 1$$

MAX-HEAPIFY: Finding recurrence

- Let the tree rooted at i have n nodes. The child subtree will have a size at most 2n/3



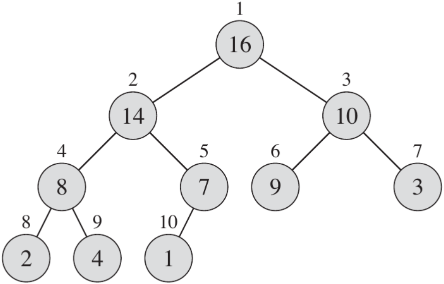
$$h = \text{ height of node } i$$

$$\text{Maximum size of child subtree } = 2^h - 1$$

$$\text{Maximum fraction of nodes } =$$

MAX-HEAPIFY: Finding recurrence

- Let the tree rooted at i have n nodes. The child subtree will have a size at most 2n/3



$$h = \text{ height of node } i$$
$$\text{Maximum size of child subtree } = 2^h - 1$$
$$\text{Maximum fraction of nodes } = \frac{2^h - 1}{2^h - 1 + 2^{h-1}}$$

MAX-HEAPIFY: Finding recurrence

$$\text{Maximum fraction of nodes } = \frac{2^h - 1}{2^h - 1 + 2^{h-1}}$$

MAX-HEAPIFY: Finding recurrence

$$\text{Maximum fraction of nodes } = \frac{2^h - 1}{2^h - 1 + 2^{h-1}}$$

$$\text{For } 0 < a < b \text{ ,} \qquad \frac{a}{b} < \frac{a+1}{b+1}$$

MAX-HEAPIFY: Finding recurrence

$$\text{Maximum fraction of nodes} \;=\; \frac{2^h - 1}{2^h - 1 + 2^{h-1}}$$

$$\text{For } 0 < a < b\;, \qquad \frac{a}{b} < \frac{a+1}{b+1}$$

$$\frac{2^h - 1}{2^h - 1 + 2^{h-1}} <$$

MAX-HEAPIFY: Finding recurrence

$$\text{Maximum fraction of nodes } = \frac{2^h - 1}{2^h - 1 + 2^{h-1}}$$

$$\text{For } 0 < a < b \text{ ,} \qquad \frac{a}{b} < \frac{a+1}{b+1}$$

$$\frac{2^h - 1}{2^h - 1 + 2^{h-1}} < \frac{2^h}{2^h + 2^{h-1}}$$

MAX-HEAPIFY: Finding recurrence

$$\text{Maximum fraction of nodes} = \frac{2^h - 1}{2^h - 1 + 2^{h-1}}$$

$$\text{For } 0 < a < b \text{ ,} \qquad \frac{a}{b} < \frac{a+1}{b+1}$$

$$\frac{2^h - 1}{2^h - 1 + 2^{h-1}} < \frac{2^h}{2^h + 2^{h-1}}$$

$$= \frac{2 \times 2^{h-1}}{(2+1) \times 2^{h-1}}$$

MAX-HEAPIFY: Finding recurrence

$$\text{Maximum fraction of nodes} = \frac{2^h - 1}{2^h - 1 + 2^{h-1}}$$

$$\text{For } 0 < a < b \text{ ,} \qquad \frac{a}{b} < \frac{a+1}{b+1}$$

$$\frac{2^h - 1}{2^h - 1 + 2^{h-1}} < \frac{2^h}{2^h + 2^{h-1}}$$

$$= \frac{2 \times 2^{h-1}}{(2+1) \times 2^{h-1}} = \frac{2}{3}$$

MAX-HEAPIFY: Finding recurrence

$$\text{Maximum fraction of nodes} = \frac{2^h - 1}{2^h - 1 + 2^{h-1}}$$

$$\text{For } 0 < a < b\ , \qquad \frac{a}{b} < \frac{a+1}{b+1}$$

$$\frac{2^h - 1}{2^h - 1 + 2^{h-1}} < \frac{2^h}{2^h + 2^{h-1}}$$

$$= \frac{2 \times 2^{h-1}}{(2+1) \times 2^{h-1}} = \frac{2}{3}$$

$$\text{Maximum size of child subtree} < \frac{2n}{3}$$

MAX-HEAPIFY: Maintaining the heap property

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

MAX-HEAPIFY: Running time

- $T(n) \leq T(2n/3) + c$

MAX-HEAPIFY: Running time

- $T(n) \leq T(2n/3) + c$

$$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = 1$$

MAX-HEAPIFY: Running time

- $T(n) \leq T(2n/3) + c$

$$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = 1$$
$$c = \Theta(n^{\log_b a} \lg^k n)$$

MAX-HEAPIFY: Running time

- $T(n) \leq T(2n/3) + c$

$$
\begin{aligned}
n^{\log_b a} &= n^{\log_{\frac{3}{2}} 1} = 1 \\
c &= \Theta(n^{\log_b a} \lg^k n) \\
&= \Theta(1) \text{ , for } k = 0
\end{aligned}
$$

MAX-HEAPIFY: Running time

- $T(n) \leq T(2n/3) + c$

$$
\begin{aligned}
n^{\log_b a} &= n^{\log_{\frac{3}{2}} 1} = 1 \\
c &= \Theta(n^{\log_b a} \lg^k n) \\
&= \Theta(1) \text{ , for } k = 0 \\
\text{Soln. } & \Theta(n^{\log_b a} \lg^{k+1} n)
\end{aligned}
$$

MAX-HEAPIFY: Running time

- $T(n) \leq T(2n/3) + c$

$$
\begin{aligned}
n^{\log_b a} &= n^{\log_{\frac{3}{2}} 1} = 1 \\
c &= \Theta(n^{\log_b a} \lg^k n) \\
&= \Theta(1)\ \text{, for } k = 0 \\
\text{Soln. } &\Theta(n^{\log_b a} \lg^{k+1} n) = \Theta(\lg n)
\end{aligned}
$$

MAX-HEAPIFY: Running time

- $T(n) \leq T(2n/3) + c$

$$
\begin{aligned}
n^{\log_b a} &= n^{\log_{\frac{3}{2}} 1} = 1 \\
c &= \Theta(n^{\log_b a} \lg^k n) \\
&= \Theta(1) \text{ , for } k = 0 \\
\text{Soln. } & \Theta(n^{\log_b a} \lg^{k+1} n) = \Theta(\lg n) \\
T(n) &= O(\lg n)
\end{aligned}
$$

BUILD-MAX-HEAP : Building a heap

- Convert an unordered array into a max-heap using MAX-HEAPIFY

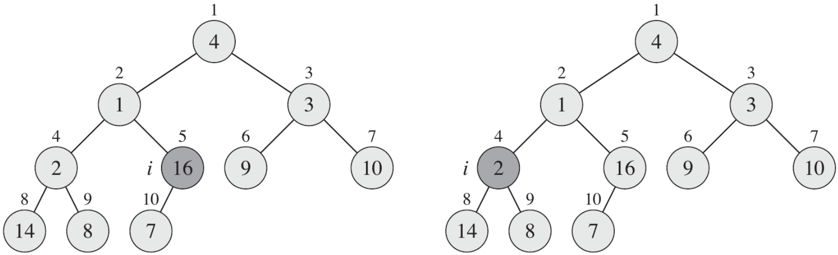
BUILD-MAX-HEAP : Building a heap

- Convert an unordered array into a max-heap using MAX-HEAPIFY

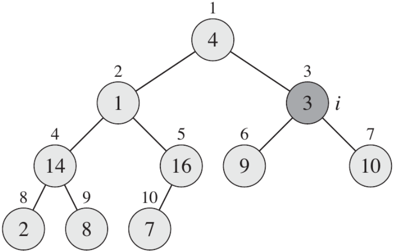
```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

BUILD-MAX-HEAP : Building a heap

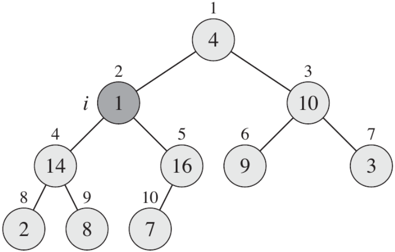
A									
4	1	3	2	16	9	10	14	8	7



BUILD-MAX-HEAP : Building a heap

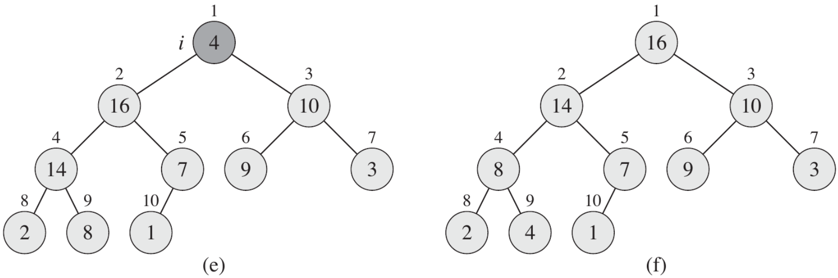


(c)



(d)

BUILD-MAX-HEAP : Building a heap



BUILD-MAX-HEAP : Running time

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

BUILD-MAX-HEAP : Running time

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

Running time:

BUILD-MAX-HEAP : Running time

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

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

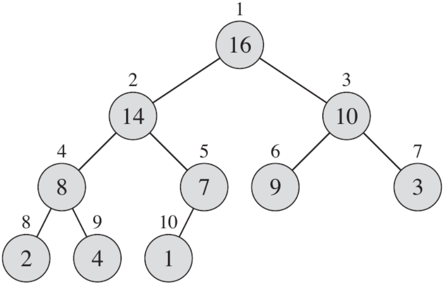
BUILD-MAX-HEAP : Running time

BUILD-MAX-HEAP(A)

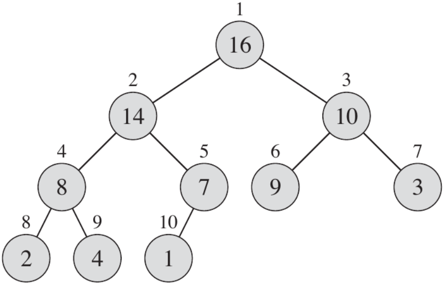
```
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

Running time: $O(n \lg n)$ { Not asymptotically tight }

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

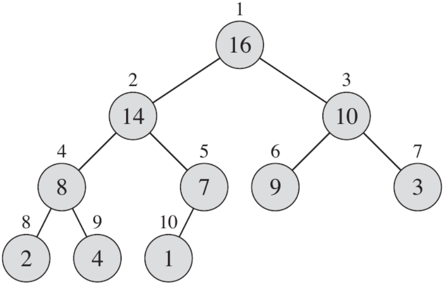


Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$



- There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes having a height h.

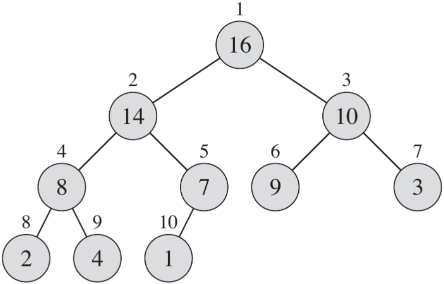
Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$



- There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes having a height h.

$h = 0,$

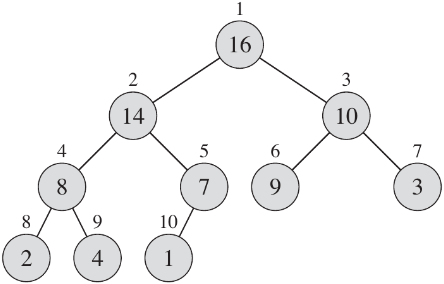
Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$



- There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes having a height h.

$$h = 0, \qquad \left\lceil \frac{10}{2^{0+1}} \right\rceil$$

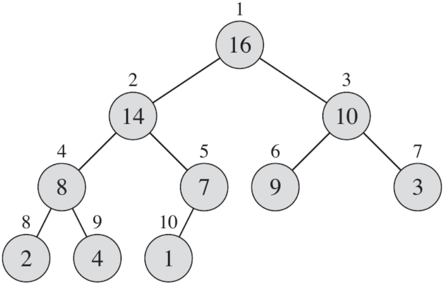
Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$



- There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes having a height h.

$$h = 0, \qquad \left\lceil \frac{10}{2^{0+1}} \right\rceil = 5;$$

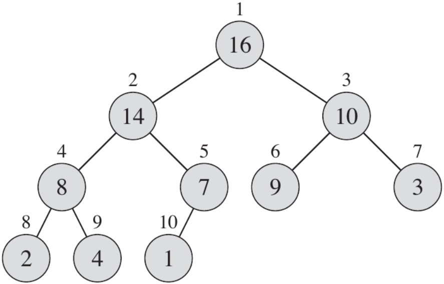
Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$



- There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes having a height h.

$$h = 0, \qquad \left\lceil \frac{10}{2^{0+1}} \right\rceil = 5; \qquad h = 1,$$

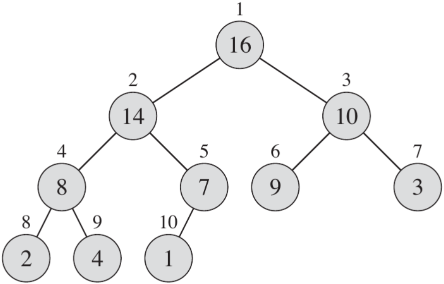
Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$



- There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes having a height h.

$$h = 0, \qquad \left\lceil \frac{10}{2^{0+1}} \right\rceil = 5; \qquad\qquad h = 1, \qquad \left\lceil \frac{10}{2^{1+1}} \right\rceil = 3$$

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

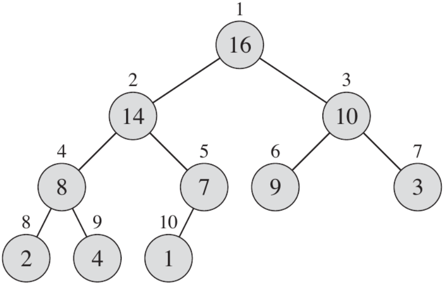


- There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes having a height h.

$h = 0, \qquad \left\lceil \frac{10}{2^{0+1}} \right\rceil = 5; \qquad h = 1, \qquad \left\lceil \frac{10}{2^{1+1}} \right\rceil = 3$

$h = 2,$

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

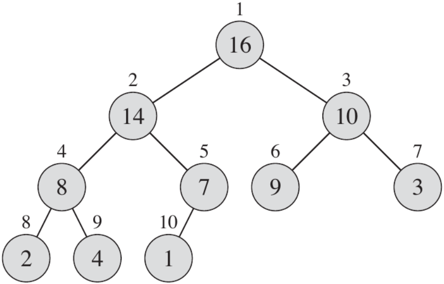


- There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes having a height h.

$$h = 0, \quad \left\lceil \frac{10}{2^{0+1}} \right\rceil = 5; \qquad h = 1, \quad \left\lceil \frac{10}{2^{1+1}} \right\rceil = 3$$

$$h = 2, \quad \left\lceil \frac{10}{2^{2+1}} \right\rceil = 2;$$

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

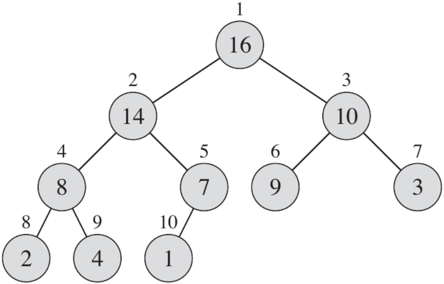


- There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes having a height h.

$$h = 0, \quad \left\lceil \frac{10}{2^{0+1}} \right\rceil = 5; \qquad h = 1, \quad \left\lceil \frac{10}{2^{1+1}} \right\rceil = 3$$

$$h = 2, \quad \left\lceil \frac{10}{2^{2+1}} \right\rceil = 2; \qquad h = 3,$$

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$



- There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes having a height h.

$$h = 0, \quad \left\lceil \frac{10}{2^{0+1}} \right\rceil = 5; \qquad h = 1, \quad \left\lceil \frac{10}{2^{1+1}} \right\rceil = 3$$

$$h = 2, \quad \left\lceil \frac{10}{2^{2+1}} \right\rceil = 2; \qquad h = 3, \quad \left\lceil \frac{10}{2^{3+1}} \right\rceil = 1$$

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

- Proof by Induction

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

- Proof by Induction
- Base case: $h = 0$

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

- Proof by Induction
- Base case: $h = 0$
  Number of nodes having zero height $= \left\lceil \frac{n}{2} \right\rceil$

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

- Proof by Induction
- Base case: $h = 0$
  Number of nodes having zero height $= \left\lceil \frac{n}{2} \right\rceil \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

- Proof by Induction
- Base case: $h = 0$
  Number of nodes having zero height $= \left\lceil \frac{n}{2} \right\rceil \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Inductive step:
  Assume that there are at most $\left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil$ number of nodes of height $h - 1$.

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

- Proof by Induction
- Base case: $h = 0$
  Number of nodes having zero height $= \left\lceil \frac{n}{2} \right\rceil \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Inductive step:
  Assume that there are at most $\left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil$ number of nodes of height $h - 1$.
  Let k_h be the number of nodes of height h in the binary heap T.

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

- Proof by Induction
- Base case: $h = 0$
  Number of nodes having zero height $= \left\lceil \frac{n}{2} \right\rceil \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Inductive step:
  Assume that there are at most $\left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil$ number of nodes of height $h-1$.
  Let k_h be the number of nodes of height h in the binary heap T.
  Let us construct a new heap T' by removing all leaf nodes from T

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

- Proof by Induction
- Base case: $h = 0$
  Number of nodes having zero height $= \left\lceil \frac{n}{2} \right\rceil \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Inductive step:
  Assume that there are at most $\left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil$ number of nodes of height $h - 1$.
  Let k_h be the number of nodes of height h in the binary heap T.
  Let us construct a new heap T' by removing all leaf nodes from T

$$k_h = k'_{h-1}$$

Nodes with height $h \le \left\lceil \frac{n}{2^{h+1}} \right\rceil$

- Proof by Induction
- Base case: $h = 0$
  Number of nodes having zero height $= \left\lceil \frac{n}{2} \right\rceil \le \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Inductive step:
  Assume that there are at most $\left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil$ number of nodes of height $h - 1$.
  Let k_h be the number of nodes of height h in the binary heap T.
  Let us construct a new heap T' by removing all leaf nodes from T

$$k_h = k'_{h-1} \le \left\lceil \frac{n'}{2^{(h-1)+1}} \right\rceil$$

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

- Proof by Induction
- Base case: $h = 0$

  Number of nodes having zero height $= \left\lceil \frac{n}{2} \right\rceil \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Inductive step:

  Assume that there are at most $\left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil$ number of nodes of height $h-1$.

  Let k_h be the number of nodes of height h in the binary heap T.

  Let us construct a new heap T' by removing all leaf nodes from T

$$k_h = k'_{h-1} \leq \left\lceil \frac{n'}{2^{(h-1)+1}} \right\rceil = \left\lceil \frac{\lfloor (n/2) \rfloor}{2^{(h-1)+1}} \right\rceil$$

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

- Proof by Induction
- Base case: $h = 0$
  Number of nodes having zero height $= \left\lceil \frac{n}{2} \right\rceil \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Inductive step:
  Assume that there are at most $\left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil$ number of nodes of height $h - 1$.
  Let k_h be the number of nodes of height h in the binary heap T.
  Let us construct a new heap T' by removing all leaf nodes from T

$$k_h = k'_{h-1} \leq \left\lceil \frac{n'}{2^{(h-1)+1}} \right\rceil = \left\lceil \frac{\lfloor (n/2) \rfloor}{2^{(h-1)+1}} \right\rceil$$
$$\leq \left\lceil \frac{(n/2)}{2^h} \right\rceil$$

Nodes with height $h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

- Proof by Induction
- Base case: $h = 0$

  Number of nodes having zero height $= \left\lceil \frac{n}{2} \right\rceil \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Inductive step:

  Assume that there are at most $\left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil$ number of nodes of height $h-1$.

  Let k_h be the number of nodes of height h in the binary heap T.

  Let us construct a new heap T' by removing all leaf nodes from T

$$
\begin{aligned}
k_h = k'_{h-1} &\leq \left\lceil \frac{n'}{2^{(h-1)+1}} \right\rceil = \left\lceil \frac{\lfloor (n/2) \rfloor}{2^{(h-1)+1}} \right\rceil \\
&\leq \left\lceil \frac{(n/2)}{2^h} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil
\end{aligned}
$$

BUILD-MAX-HEAP : Running time

```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

BUILD-MAX-HEAP : Running time

BUILD-MAX-HEAP(A)

```
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

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

BUILD-MAX-HEAP : Running time

```
BUILD-MAX-HEAP(A)
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

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

BUILD-MAX-HEAP : Running time

BUILD-MAX-HEAP(A)

1 $A.heap\text{-}size = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3 MAX-HEAPIFY(A, i)

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

$$\sum_{k=0}^{\infty} kx^k$$

BUILD-MAX-HEAP : Running time

BUILD-MAX-HEAP(A)

```
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

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

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

BUILD-MAX-HEAP : Running time

```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

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

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

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

BUILD-MAX-HEAP : Running time

BUILD-MAX-HEAP(A)

```
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

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

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

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

BUILD-MAX-HEAP(A)

```
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

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

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

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

BUILD-MAX-HEAP : Running time

BUILD-MAX-HEAP(A)

1 $A.heap\text{-}size = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3 MAX-HEAPIFY(A, i)

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

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

BUILD-MAX-HEAP : Running time

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

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

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= O(2n)$$

BUILD-MAX-HEAP : Running time

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

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

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= O(2n) = O(n)$$

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

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

- $0 < x < 1$, $\quad 1 + x + x^2 + \dots$

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

- $0 < x < 1$, $\quad 1 + x + x^2 + \ldots = \dfrac{1}{1-x}$

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

- $0 < x < 1$, $\qquad 1 + x + x^2 + \ldots = \dfrac{1}{1-x}$
- $\displaystyle\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$

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

- $0 < x < 1$, $\qquad 1 + x + x^2 + \ldots = \dfrac{1}{1-x}$
- $\displaystyle\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$
- Differentiating both sides w.r.t x

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

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

- $0 < x < 1$, $\quad 1 + x + x^2 + \ldots = \dfrac{1}{1-x}$
- $\displaystyle\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$
- Differentiating both sides w.r.t x

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

- Multiplying x on both sides

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

Heap

- We can construct a max heap (or a min heap) from an unordered array in $O(n)$ time.

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
```

Heapsort

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

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

Heapsort

HEAPSORT(A)

```
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

- Running time : $O(n \lg n)$
- Operation (P. 161)

Priority Queues

- Heap data structure can be used to construct efficient priority queue

Priority Queues

- Heap data structure can be used to construct efficient priority queue
- Job scheduling

Priority Queues

- Heap data structure can be used to construct efficient priority queue
- Job scheduling
- Two forms : max-priority queue and min priority-queue

Priority Queues

- Heap data structure can be used to construct efficient priority queue
- Job scheduling
- Two forms : max-priority queue and min priority-queue
- Max-priority queue operations:

Priority Queues

- Heap data structure can be used to construct efficient priority queue
- Job scheduling
- Two forms : max-priority queue and min priority-queue
- Max-priority queue operations:
  1. Insert(S,x)

Priority Queues

- Heap data structure can be used to construct efficient priority queue
- Job scheduling
- Two forms : max-priority queue and min priority-queue
- Max-priority queue operations:
    1. Insert(S,x)
    2. Maximum(S)

Priority Queues

- Heap data structure can be used to construct efficient priority queue
- Job scheduling
- Two forms : max-priority queue and min priority-queue
- Max-priority queue operations:
  1. Insert(S,x)
  2. Maximum(S)
  3. Extract-Max(S)

Priority Queues

- Heap data structure can be used to construct efficient priority queue
- Job scheduling
- Two forms : max-priority queue and min priority-queue
- Max-priority queue operations:
    1. Insert(S,x)
    2. Maximum(S)
    3. Extract-Max(S)
    4. Increase-Key(S,x,k)

Priority Queues

HEAP-MAXIMUM(A)

1 **return** $A[1]$

Priority Queues

```
HEAP-MAXIMUM(A)
    return A[1]
```

```
HEAP-EXTRACT-MAX(A)
    if A.heap-size < 1
        error "heap underflow"
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
    return max
```

Priority Queues

```
HEAP-MAXIMUM(A)
return A[1]
```

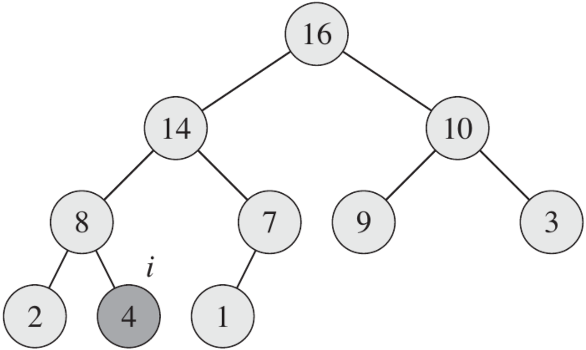
```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

Running time of Heap-Extract-Max : $O(\lg n)$

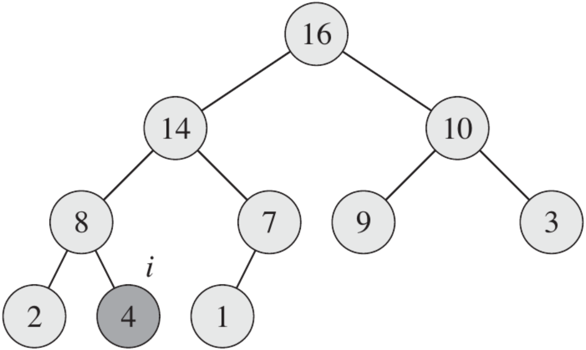
Heap-Increase-Key

```
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)
```

Heap-Increase-Key

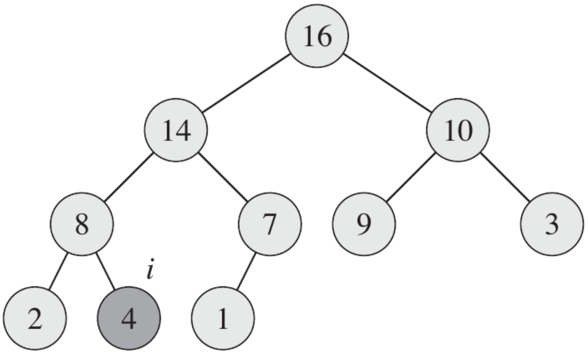


Heap-Increase-Key



- Suppose we increase the value at index i to 15.

Heap-Increase-Key



- Suppose we increase the value at index i to 15.
- Running time of Heap-Increase-Key : $O(\lg n)$

Max-Heap-Insert

MAX-HEAP-INSERT(A, key)

$A.heap\text{-}size = A.heap\text{-}size + 1$

$A[A.heap\text{-}size] = -\infty$

HEAP-INCREASE-KEY(A, $A.heap\text{-}size$, key)

Max-Heap-Insert

MAX-HEAP-INSERT(A, key)

1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$

Running time of Heap-Increase-Key : $O(\lg n)$

Priority Queue

- Using a heap, all the basic operations can be performed in $O(\lg n)$ time.