

CSE 221: Algorithms

Heapsort

Mumit Khan
Fatema Tuz Zohora

Computer Science and Engineering
BRAC University

References

- 1 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

Last modified: February 12, 2013



This work is licensed under the *Creative Commons Attribution-NonCommercial-Share Alike 3.0 Unported License*.

Contents

1 Heapsort

- Introduction
- Heap data structure
- Heap algorithms
- Heapsort algorithm
- Priority queue
- Conclusion

Contents

1 Heapsort

- Introduction
- Heap data structure
- Heap algorithms
- Heapsort algorithm
- Priority queue
- Conclusion

Heapsort

- $O(n \lg n)$ in the worst case – like *merge sort*.

Heapsort

- $O(n \lg n)$ in the worst case – like *merge sort*.
- Sorts *in place* – like *insertion sort*.

Heapsort

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

Heapsort

- $O(n \lg n)$ in the worst case – like *merge sort*.
- Sorts *in place* – like *insertion sort*.
- Combines the best of both algorithms.
- Uses a data structure called the **heap**, which is also extensively used in other applications.

Contents

1 Heapsort

- Introduction
- **Heap data structure**
- Heap algorithms
- Heapsort algorithm
- Priority queue
- Conclusion

Heap data structure

- A data structure that provides worst-case $O(1)$ time access to the largest (**max heap**) or smallest (**min heap**) element.

Heap data structure

- A data structure that provides worst-case $O(1)$ time access to the largest (**max heap**) or smallest (**min heap**) element.
- A data structure that provides worst-case $\Theta(\lg n)$ time extract the largest (**max heap**) or smallest (**min heap**) element.

Heap data structure

- A data structure that provides worst-case $O(1)$ time access to the largest (**max heap**) or smallest (**min heap**) element.
- A data structure that provides worst-case $\Theta(\lg n)$ time extract the largest (**max heap**) or smallest (**min heap**) element.
- **Priority queue** is a prototypical application, where the keys are retrieved by priority.

Heap data structure

- A data structure that provides worst-case $O(1)$ time access to the largest (**max heap**) or smallest (**min heap**) element.
- A data structure that provides worst-case $\Theta(\lg n)$ time extract the largest (**max heap**) or smallest (**min heap**) element.
- **Priority queue** is a prototypical application, where the keys are retrieved by priority.
- **Heapsort** is an another application, where the keys can be sorted by repeatedly extracting the largest from the heap.

Heap data structure

- A data structure that provides worst-case $O(1)$ time access to the largest (**max heap**) or smallest (**min heap**) element.
- A data structure that provides worst-case $\Theta(\lg n)$ time extract the largest (**max heap**) or smallest (**min heap**) element.
- **Priority queue** is a prototypical application, where the keys are retrieved by priority.
- **Heapsort** is an another application, where the keys can be sorted by repeatedly extracting the largest from the heap.

Max vs. Min Heap

Unless explicitly stated as **max heap** or **min heap**, **heap** means **max heap** in this course.

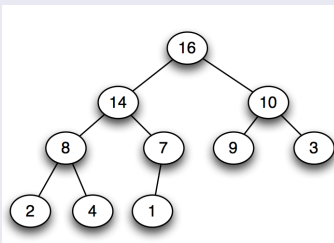
Heap-ordered tree

Definition

A binary tree is heap-ordered if:

- 1 the value at a node is \geq the value at each of its children.

Example of (max) heap



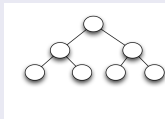
Heap-ordered tree

Definition

A binary tree is heap-ordered if:

- 1 the value at a node is \geq the value at each of its children.
- 2 the tree is **almost-complete**.

Example of complete tree (or *not*)



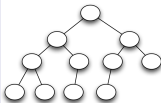
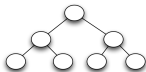
Heap-ordered tree

Definition

A binary tree is heap-ordered if:

- 1 the value at a node is \geq the value at each of its children.
- 2 the tree is **almost-complete**.

Example of complete tree (or *not*)



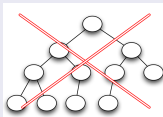
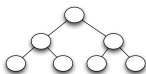
Heap-ordered tree

Definition

A binary tree is heap-ordered if:

- 1 the value at a node is \geq the value at each of its children.
- 2 the tree is **almost-complete**.

Example of complete tree (or *not*)



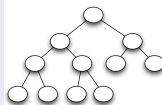
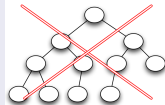
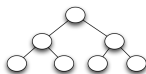
Heap-ordered tree

Definition

A binary tree is heap-ordered if:

- 1 the value at a node is \geq the value at each of its children.
- 2 the tree is **almost-complete**.

Example of complete tree (or *not*)



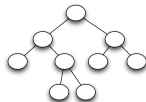
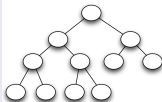
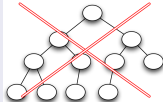
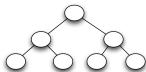
Heap-ordered tree

Definition

A binary tree is heap-ordered if:

- 1 the value at a node is \geq the value at each of its children.
- 2 the tree is **almost-complete**.

Example of complete tree (or *not*)



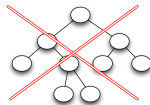
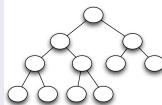
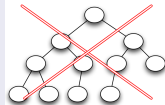
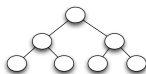
Heap-ordered tree

Definition

A binary tree is heap-ordered if:

- 1 the value at a node is \geq the value at each of its children.
- 2 the tree is **almost-complete**.

Example of complete tree (or *not*)



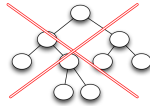
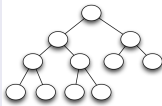
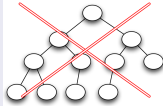
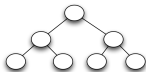
Heap-ordered tree

Definition

A binary tree is heap-ordered if:

- 1 the value at a node is \geq the value at each of its children.
- 2 the tree is **almost-complete**. Height of tree is $\Theta(\lg n)$.

Example of complete tree (or *not*)



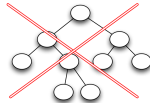
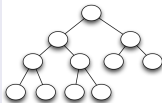
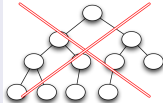
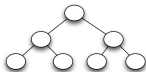
Heap-ordered tree

Definition

A binary tree is heap-ordered if:

- 1 the value at a node is \geq the value at each of its children.
- 2 the tree is **almost-complete**. Height of tree is $\Theta(\lg n)$. Why?

Example of complete tree (or *not*)



Height of a heap-ordered tree

- Height h of a tree is the maximum distance of any leaf node to the root.

Height of a heap-ordered tree

- Height h of a tree is the maximum distance of any leaf node to the root.
- A heap of height h has the most number of elements if the tree is complete, so n equals the sum of nodes at each level.

$$\begin{aligned} n &\leq 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^h \\ &= \frac{2^{h+1} - 1}{2 - 1} \\ &= 2^{h+1} - 1. \end{aligned}$$

Height of a heap-ordered tree

- Height h of a tree is the maximum distance of any leaf node to the root.
- A heap of height h has the most number of elements if the tree is complete, so n equals the sum of nodes at each level.

$$\begin{aligned}n &\leq 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^h \\&= \frac{2^{h+1} - 1}{2 - 1} \\&= 2^{h+1} - 1.\end{aligned}$$

- It has the least number of elements if the lowest level has a single element and all higher levels are complete, so $n \geq 2^h - 1 + 1 = 2^h$.

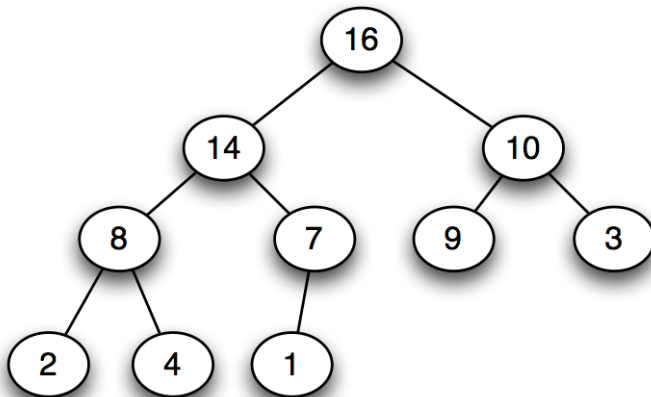
Height of a heap-ordered tree

- Height h of a tree is the maximum distance of any leaf node to the root.
- A heap of height h has the most number of elements if the tree is complete, so n equals the sum of nodes at each level.

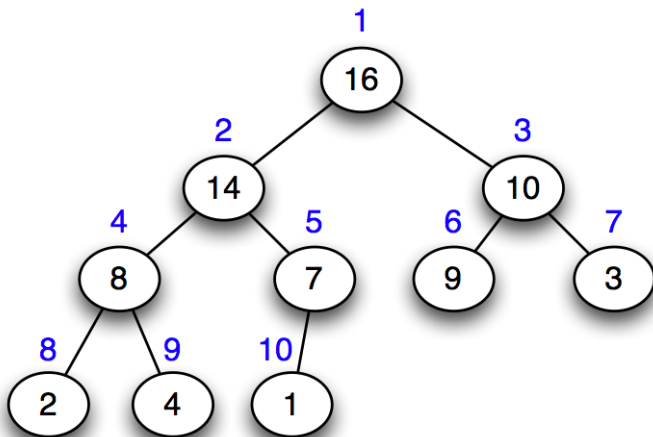
$$\begin{aligned} n &\leq 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^h \\ &= \frac{2^{h+1} - 1}{2 - 1} \\ &= 2^{h+1} - 1. \end{aligned}$$

- It has the least number of elements if the lowest level has a single element and all higher levels are complete, so $n \geq 2^h - 1 + 1 = 2^h$.
- $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1} \Rightarrow h \leq \lg n < h + 1$. Since h is an integer, $h = \lfloor \lg n \rfloor = \Theta(\lg n)$.

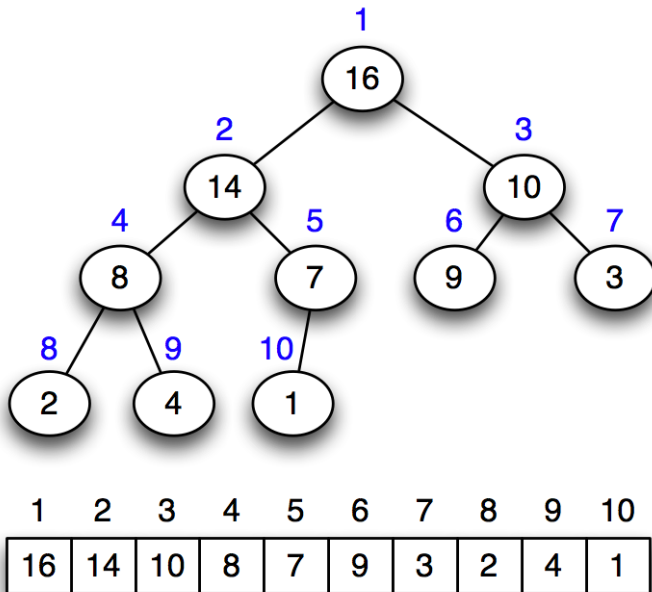
Heap – array representation of heap-ordered tree



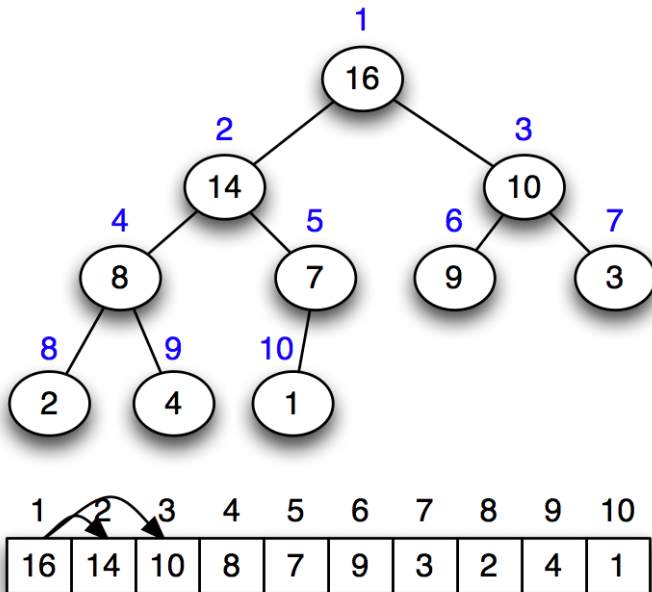
Heap – array representation of heap-ordered tree



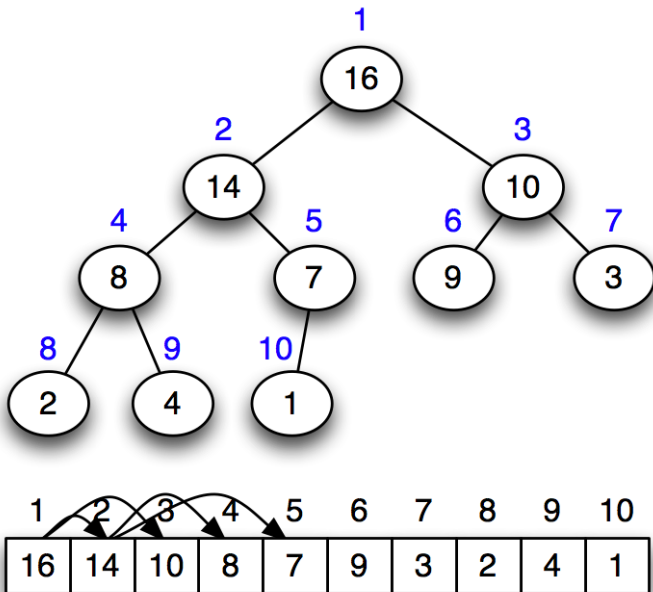
Heap – array representation of heap-ordered tree



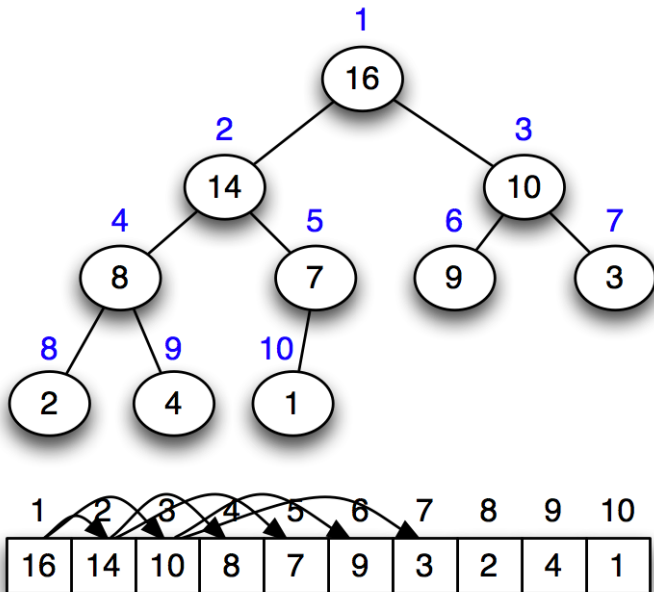
Heap – array representation of heap-ordered tree



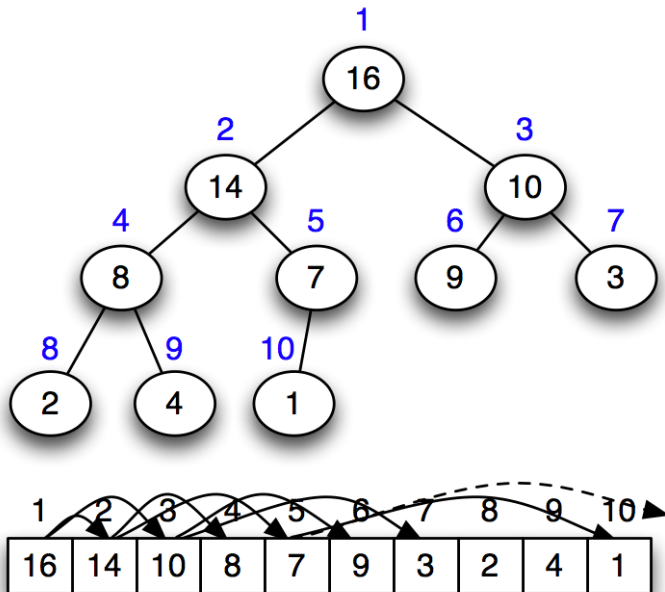
Heap – array representation of heap-ordered tree



Heap – array representation of heap-ordered tree

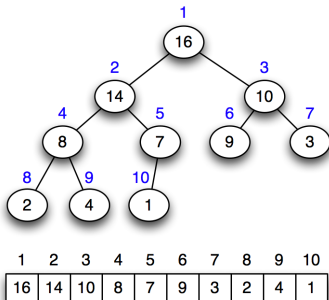


Heap – array representation of heap-ordered tree

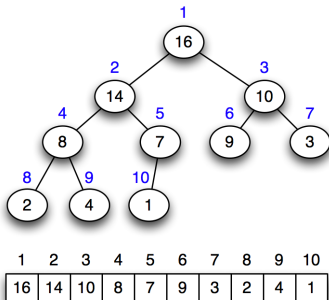


Heap – accessing parent and children

MAXIMUM(A)
return $A[1]$



Heap – accessing parent and children



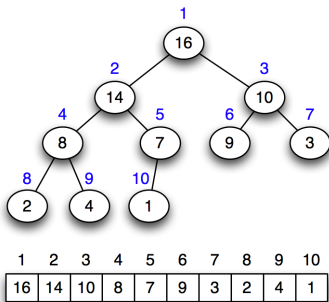
MAXIMUM(A)

return $A[1]$

PARENT(i)

return $\lfloor i/2 \rfloor$

Heap – accessing parent and children



MAXIMUM(A)

return $A[1]$

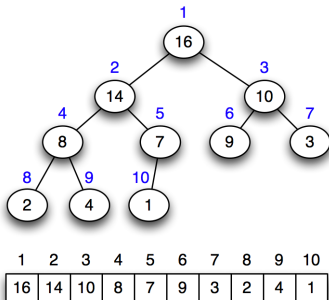
PARENT(i)

return $\lfloor i/2 \rfloor$

Question

What if $\text{PARENT}(i) < 1$?

Heap – accessing parent and children



MAXIMUM(A)

return $A[1]$

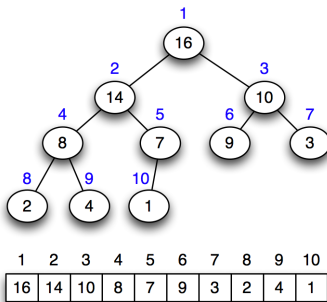
PARENT(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

Heap – accessing parent and children



MAXIMUM(A)

return $A[1]$

PARENT(i)

return $\lfloor i/2 \rfloor$

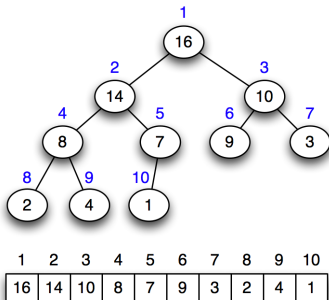
LEFT(i)

return $2i$

Question

What if LEFT(i) $> n$?

Heap – accessing parent and children



MAXIMUM(A)

return $A[1]$

PARENT(i)

return $\lfloor i/2 \rfloor$

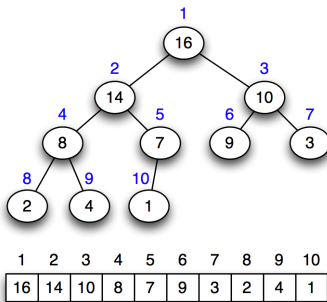
LEFT(i)

return $2i$

RIGHT(i)

return $2i + 1$

Heap – accessing parent and children



MAXIMUM(A)

return $A[1]$

PARENT(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

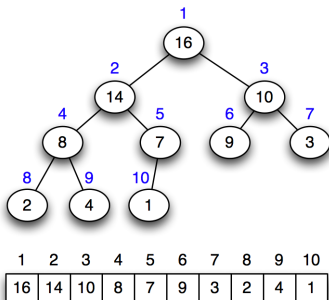
RIGHT(i)

return $2i + 1$

Question

What if RIGHT(i) $> n$?

Heap – accessing parent and children



MAXIMUM(A)

return $A[1]$

PARENT(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

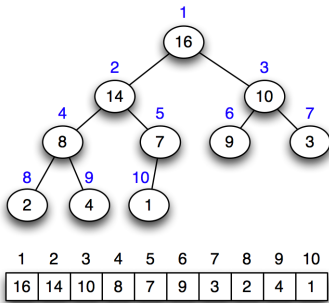
RIGHT(i)

return $2i + 1$

Lemma

All nodes $i > \lfloor \text{length}[A]/2 \rfloor$ (or equivalently, $i > \lfloor \text{heap-size}[A]/2 \rfloor$) are leaf nodes.

Heap – accessing parent and children



MAXIMUM(A)

return $A[1]$

PARENT(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

RIGHT(i)

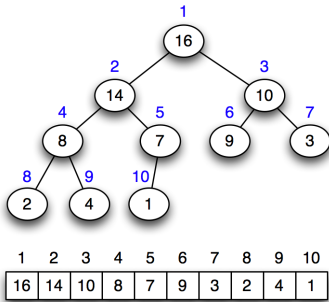
return $2i + 1$

Definition (Heap property)

Heap property: For every node i other than the root,

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

Heap – accessing parent and children



MAXIMUM(A)

return $A[1]$

PARENT(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

RIGHT(i)

return $2i + 1$

Question

Why do we insist that a heap-ordered tree be a **complete** binary tree? (Hint: draw the array representation of a tree that is not complete and see the *gaps*).

Contents

1 Heapsort

- Introduction
- Heap data structure
- **Heap algorithms**
- Heapsort algorithm
- Priority queue
- Conclusion

Operations on heap

- 1 $\text{MAX-HEAPIFY}(A, i)$ – Ensure the heap property of A starting at node i . Also known as “sink” operation since it sinks the lighter elements down the tree.

Operations on heap

- 1 $\text{MAX-HEAPIFY}(A, i)$ – Ensure the heap property of A starting at node i . Also known as “sink” operation since it sinks the lighter elements down the tree.
- 2 $\text{MAX-HEAP-INSERT}(A, \text{key})$ – Insert key in the heap A , maintaining A 's heap property.

Operations on heap

- 1 $\text{MAX-HEAPIFY}(A, i)$ – Ensure the heap property of A starting at node i . Also known as “sink” operation since it sinks the lighter elements down the tree.
- 2 $\text{MAX-HEAP-INSERT}(A, \text{key})$ – Insert key in the heap A , maintaining A 's heap property.
- 3 $\text{BUILD-MAX-HEAP}(A)$ – Build a **max heap** given an array A .

Operations on heap

- 1 $\text{MAX-HEAPIFY}(A, i)$ – Ensure the heap property of A starting at node i . Also known as “sink” operation since it sinks the lighter elements down the tree.
- 2 $\text{MAX-HEAP-INSERT}(A, \text{key})$ – Insert key in the heap A , maintaining A 's heap property.
- 3 $\text{BUILD-MAX-HEAP}(A)$ – Build a **max heap** given an array A .
- 4 $\text{HEAPSORT}(A)$ – Sort the elements in array A using the heap operations.

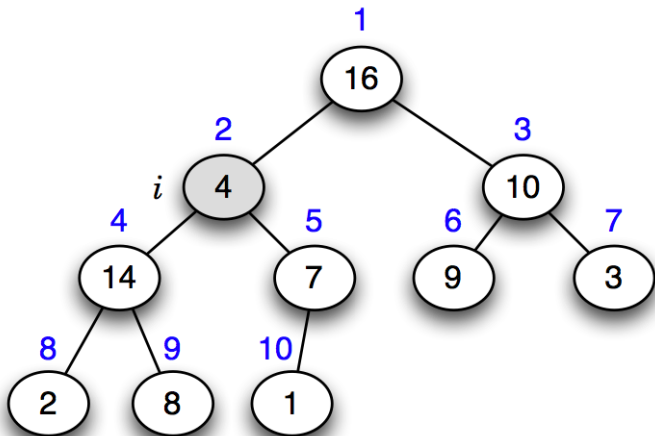
Operations on heap

- 1 $\text{MAX-HEAPIFY}(A, i)$ – Ensure the heap property of A starting at node i . Also known as “sink” operation since it sinks the lighter elements down the tree.
- 2 $\text{MAX-HEAP-INSERT}(A, \text{key})$ – Insert key in the heap A , maintaining A 's heap property.
- 3 $\text{BUILD-MAX-HEAP}(A)$ – Build a **max heap** given an array A .
- 4 $\text{HEAPSORT}(A)$ – Sort the elements in array A using the heap operations.
- 5 $\text{HEAP-INCREASE-KEY}(A, i, \text{key})$ – Increase the value of element at node i to key , and ensure the heap property of A by moving larger elements upwards. Also known as “swim” operation as it moves larger elements upwards.

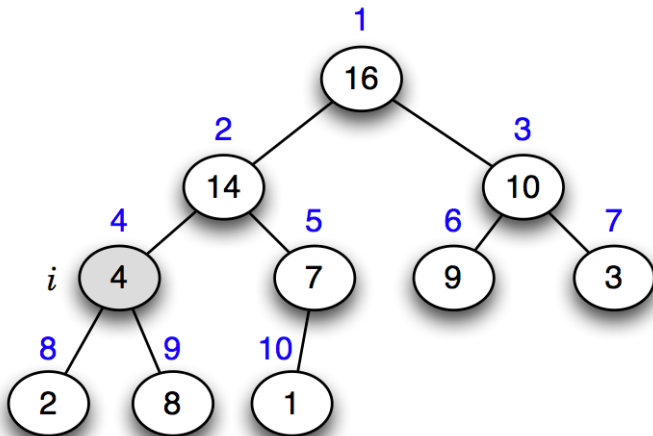
Operations on heap

- ❶ $\text{MAX-HEAPIFY}(A, i)$ – Ensure the heap property of A starting at node i . Also known as “sink” operation since it sinks the lighter elements down the tree.
- ❷ $\text{MAX-HEAP-INSERT}(A, \text{key})$ – Insert key in the heap A , maintaining A 's heap property.
- ❸ $\text{BUILD-MAX-HEAP}(A)$ – Build a **max heap** given an array A .
- ❹ $\text{HEAPSORT}(A)$ – Sort the elements in array A using the heap operations.
- ❺ $\text{HEAP-INCREASE-KEY}(A, i, \text{key})$ – Increase the value of element at node i to key , and ensure the heap property of A by moving larger elements upwards. Also known as “swim” operation as it moves larger elements upwards.
- ❻ $\text{HEAP-EXTRACT-MAX}(A)$ – Extract the largest element from heap A .

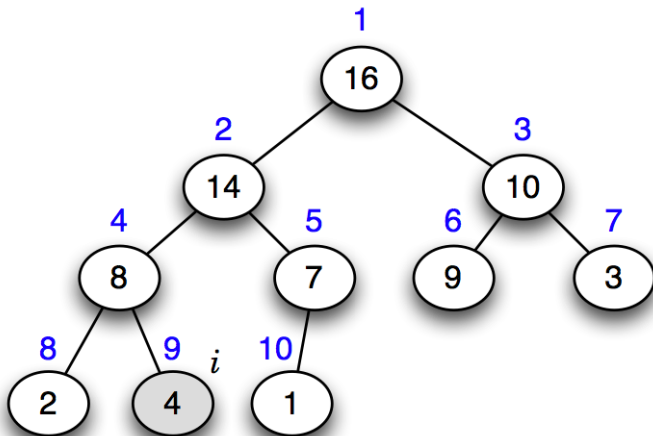
Example of MAX-HEAPIFY ("sink") operation



Example of MAX-HEAPIFY ("sink") operation



Example of MAX-HEAPIFY ("sink") operation



MAX-HEAPIFY algorithm

MAX-HEAPIFY(A, i)

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

MAX-HEAPIFY algorithm

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

Analysis – first way

Since the children's subtrees each have at most size of $2n/3$ (when the last row is exactly half full), we have

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

According to case 2 of the Master theorem, $T(n) = O(\lg n)$.

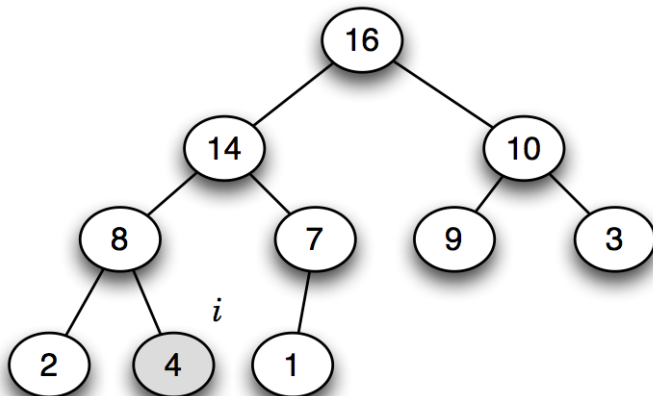
MAX-HEAPIFY algorithm

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

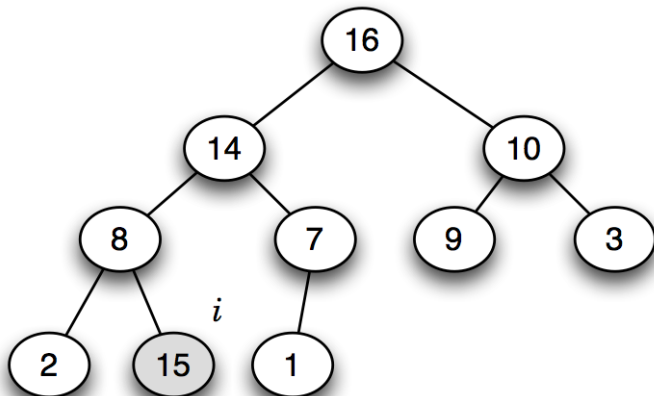
Analysis – second way

The running time of MAX-HEAPIFY on a node of height h is
 $T(n) = O(h) = O(\lg n)$.

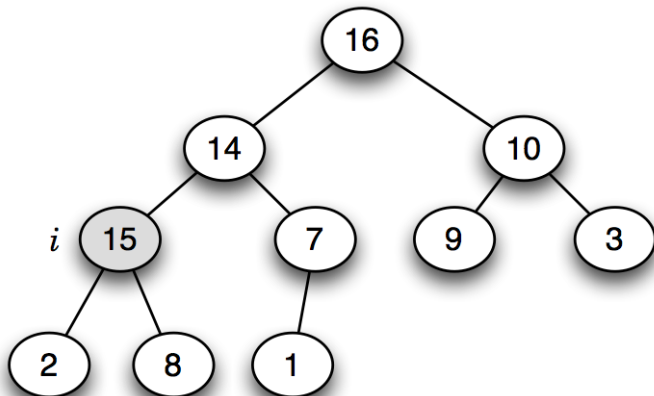
Example of HEAP-INCREASE-KEY (“swim”) operation



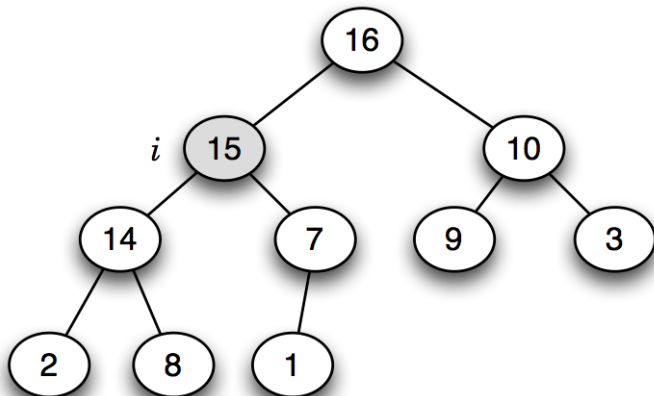
Example of HEAP-INCREASE-KEY (“swim”) operation



Example of HEAP-INCREASE-KEY (“swim”) operation



Example of HEAP-INCREASE-KEY (“swim”) operation



HEAP-INCREASE-KEY algorithm

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

```
1  if  $\text{key} < A[i]$ 
2      then error "new key is smaller than current key"
3   $A[i] \leftarrow \text{key}$ 
4  while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$ 
5      do exchange  $A[i] \leftrightarrow A[\text{parent}(i)]$ 
6       $i \leftarrow \text{PARENT}(i)$ 
```

HEAP-INCREASE-KEY algorithm

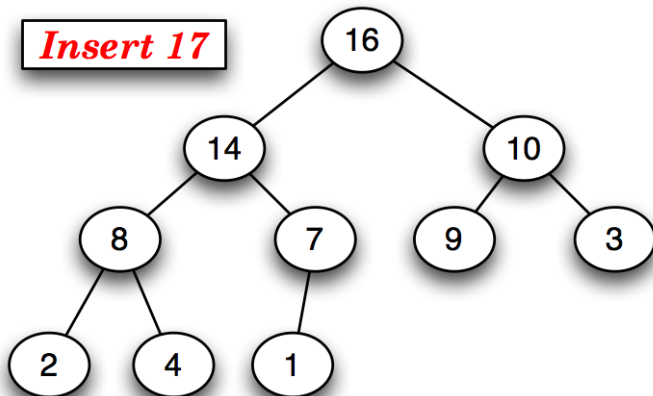
HEAP-INCREASE-KEY(A, i, key)

```
1  if  $\text{key} < A[i]$ 
2      then error "new key is smaller than current key"
3   $A[i] \leftarrow \text{key}$ 
4  while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$ 
5      do exchange  $A[i] \leftrightarrow A[\text{parent}(i)]$ 
6       $i \leftarrow \text{PARENT}(i)$ 
```

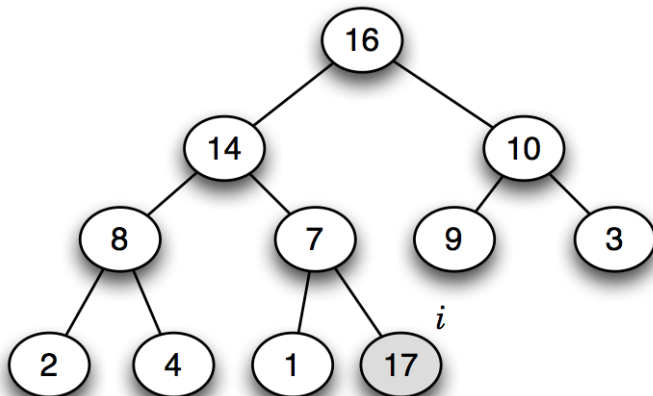
Analysis

A node may move all the way from a leaf node to the root because of increased value, so $T(n) = O(h) = O(\lg n)$.

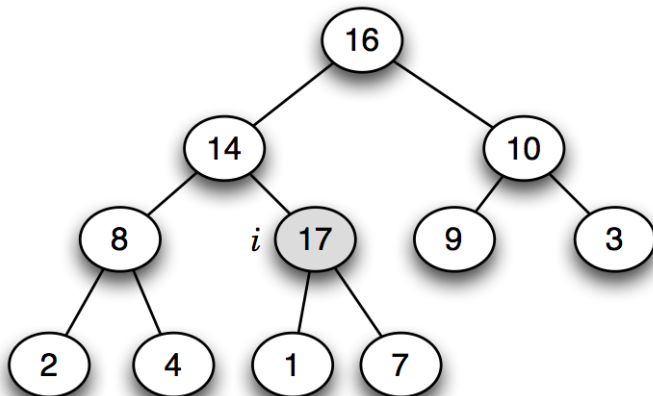
Example of MAX-HEAP-INSERT operation



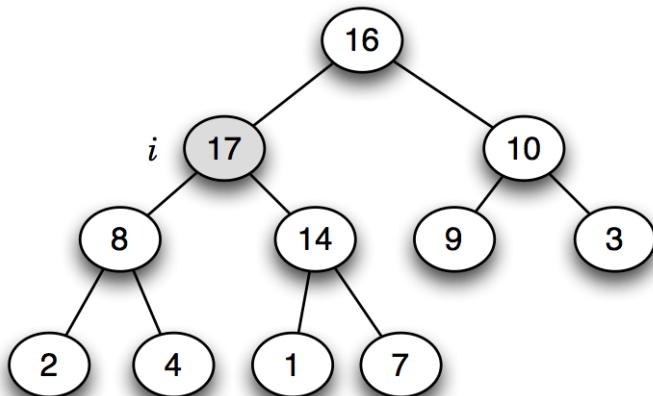
Example of MAX-HEAP-INSERT operation



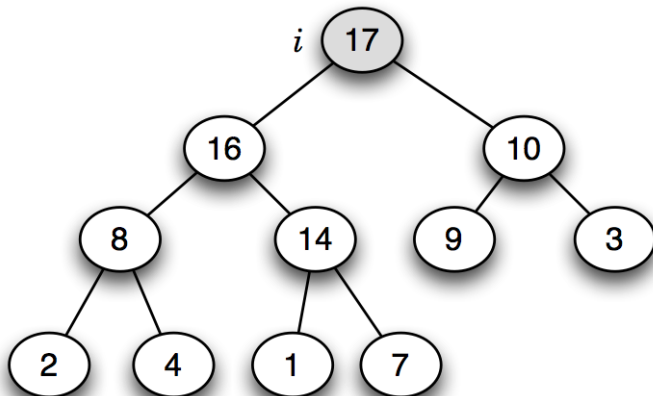
Example of MAX-HEAP-INSERT operation



Example of MAX-HEAP-INSERT operation



Example of MAX-HEAP-INSERT operation



MAX-HEAP-INSERT algorithm

MAX-HEAP-INSERT(A, key)

- 1 $heap-size[A] \leftarrow heap-size[A] + 1$
- 2 $A[heap-size[A]] \leftarrow key$
- 3 $i \leftarrow heap-size[A]$
- 4 **while** $i > 1$ and $A[PARENT(i)] < A[i]$
- 5 **do** exchange $A[i] \leftrightarrow A[parent(i)]$
- 6 $i \leftarrow PARENT(i)$

MAX-HEAP-INSERT algorithm

MAX-HEAP-INSERT(A, key)

- 1 $heap-size[A] \leftarrow heap-size[A] + 1$
- 2 $A[heap-size[A]] \leftarrow key$
- 3 $i \leftarrow heap-size[A]$
- 4 **while** $i > 1$ and $A[PARENT(i)] < A[i]$
- 5 **do** exchange $A[i] \leftrightarrow A[parent(i)]$
- 6 $i \leftarrow PARENT(i)$

Can also be done using HEAP-INCREASE-KEY.

MAX-HEAP-INSERT(A, key)

- 1 $heap-size[A] \leftarrow heap-size[A] + 1$
- 2 $A[heap-size[A]] \leftarrow -\infty$
- 3 HEAP-INCREASE-KEY($A, heap-size[A], key$)

MAX-HEAP-INSERT algorithm

MAX-HEAP-INSERT(A, key)

- 1 $heap-size[A] \leftarrow heap-size[A] + 1$
- 2 $A[heap-size[A]] \leftarrow key$
- 3 $i \leftarrow heap-size[A]$
- 4 **while** $i > 1$ and $A[PARENT(i)] < A[i]$
- 5 **do** exchange $A[i] \leftrightarrow A[parent(i)]$
- 6 $i \leftarrow PARENT(i)$

Can also be done using HEAP-INCREASE-KEY.

MAX-HEAP-INSERT(A, key)

- 1 $heap-size[A] \leftarrow heap-size[A] + 1$
- 2 $A[heap-size[A]] \leftarrow -\infty$
- 3 HEAP-INCREASE-KEY($A, heap-size[A], key$)

Analysis

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

Simple BUILD-MAX-HEAP algorithm

BUILD-MAX-HEAP'(A)

1 *heap-size*[A] \leftarrow 1

2 **for** $i \leftarrow 2$ **to** *length*[A]

3 **do** MAX-HEAP-INSERT(A, A[i])

Simple BUILD-MAX-HEAP algorithm

BUILD-MAX-HEAP'(A)

1 *heap-size*[A] \leftarrow 1

2 **for** $i \leftarrow 2$ **to** *length*[A]

3 **do** MAX-HEAP-INSERT(A, A[i])

Analysis

There are $n - 1$ calls to MAX-HEAP-INSERT, each taking $O(\lg n)$ time, so $T(n) = O(n \lg n)$.

Simple BUILD-MAX-HEAP algorithm

BUILD-MAX-HEAP'(A)

```
1  heap-size[A]  $\leftarrow$  1
2  for  $i \leftarrow 2$  to length[A]
3      do MAX-HEAP-INSERT(A, A[i])
```

Analysis

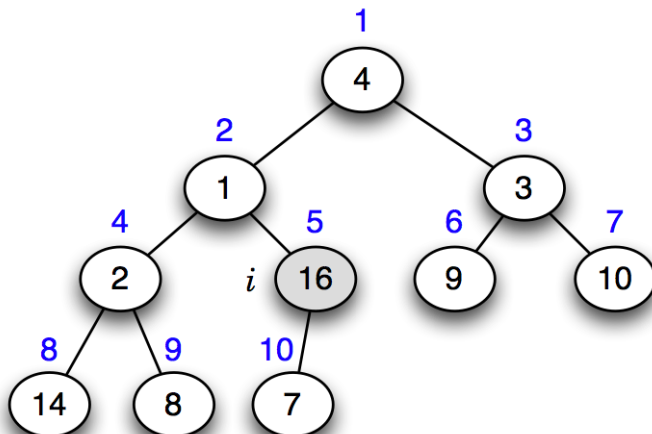
There are $n - 1$ calls to MAX-HEAP-INSERT, each taking $O(\lg n)$ time, so $T(n) = O(n \lg n)$.

Better way?

A better way is to build up the heap from the smaller trees. See next.

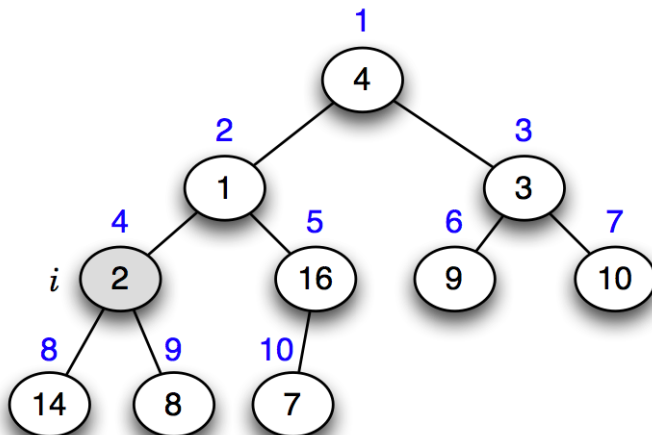
Example of BUILD-MAX-HEAP (“heapify”) operation

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



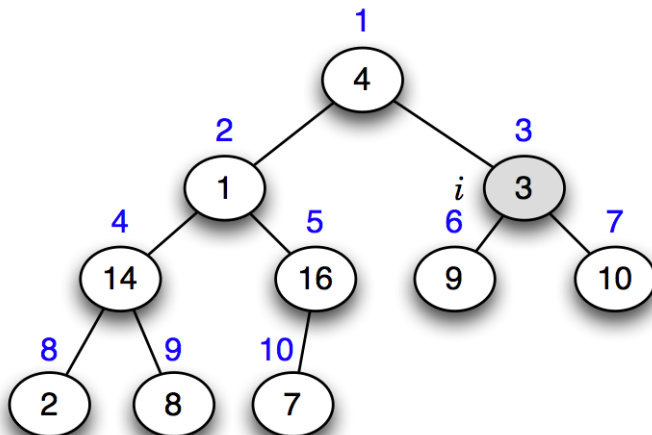
Example of BUILD-MAX-HEAP (“heapify”) operation

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



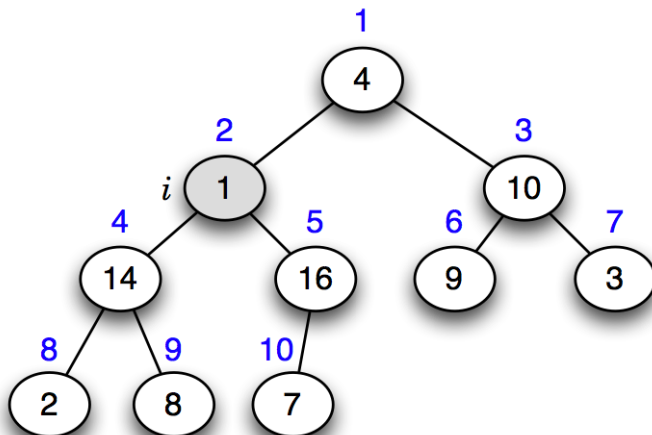
Example of BUILD-MAX-HEAP (“heapify”) operation

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



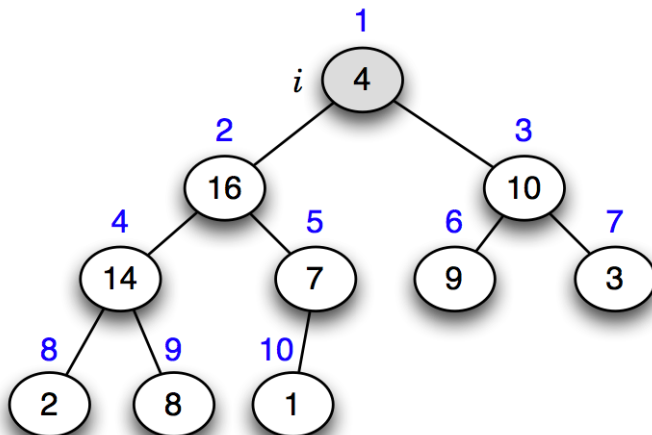
Example of BUILD-MAX-HEAP (“heapify”) operation

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



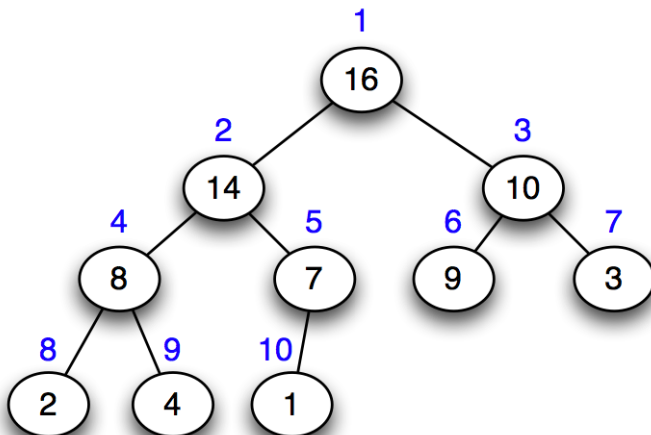
Example of BUILD-MAX-HEAP (“heapify”) operation

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



Example of BUILD-MAX-HEAP (“heapify”) operation

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



BUILD-MAX-HEAP algorithm

BUILD-MAX-HEAP(A)

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


BUILD-MAX-HEAP algorithm

BUILD-MAX-HEAP(A)

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

Analysis

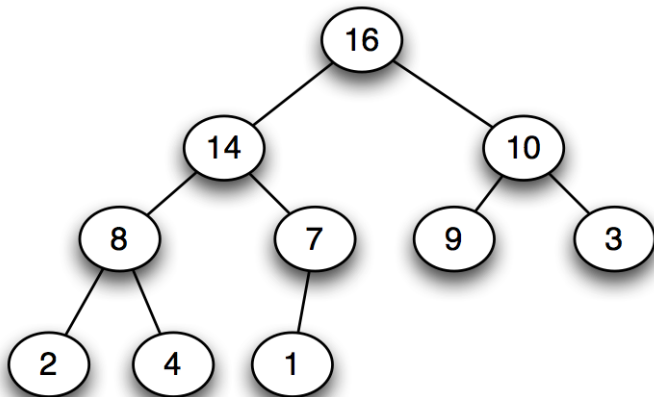
$T(n) = O(n)$ (see textbook for details)

Contents

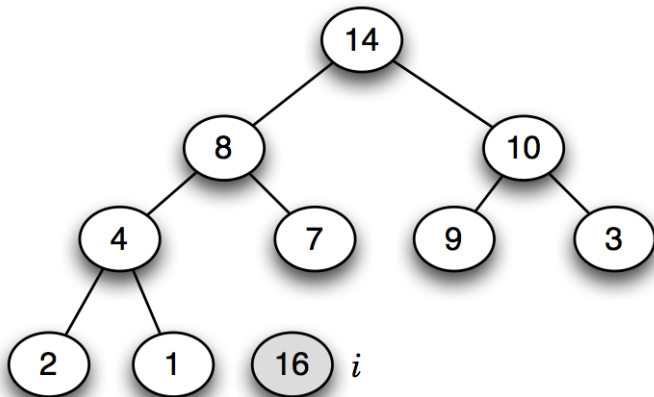
1 Heapsort

- Introduction
- Heap data structure
- Heap algorithms
- **Heapsort algorithm**
- Priority queue
- Conclusion

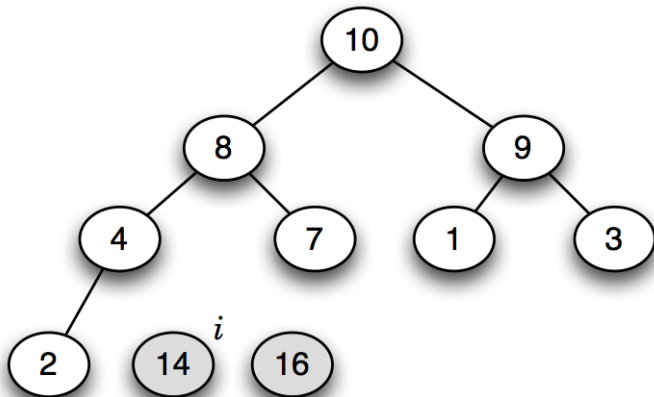
Heap use - heapsort



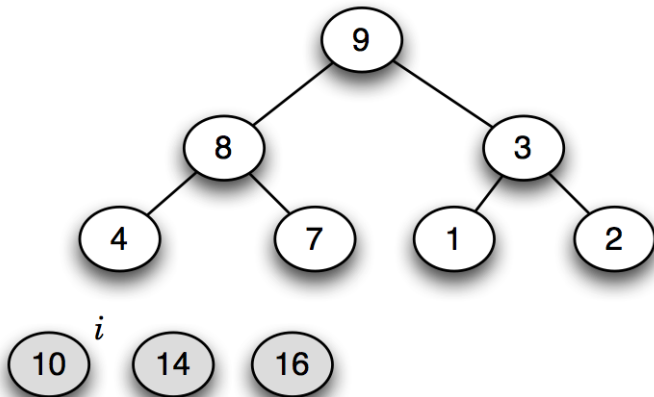
Heap use - heapsort



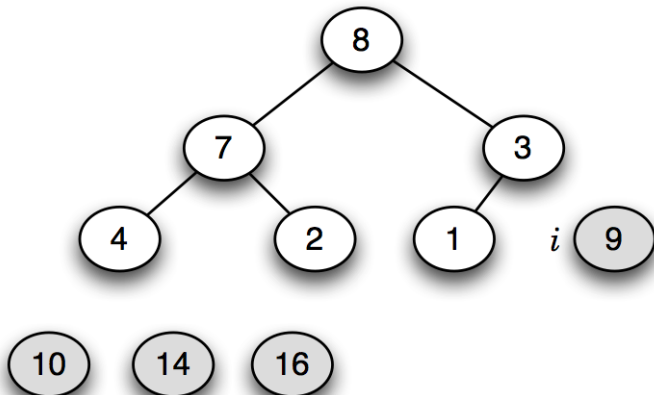
Heap use - heapsort



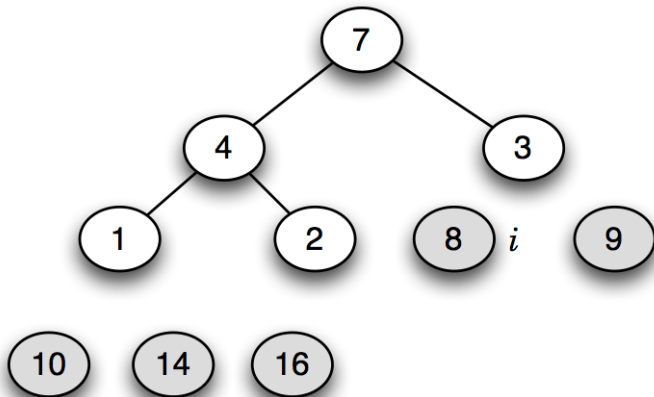
Heap use - heapsort



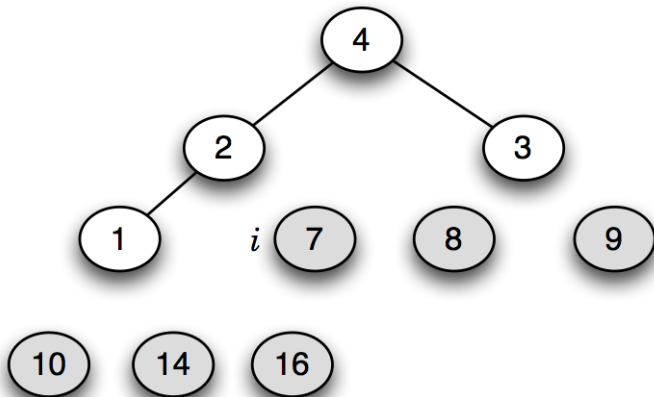
Heap use - heapsort



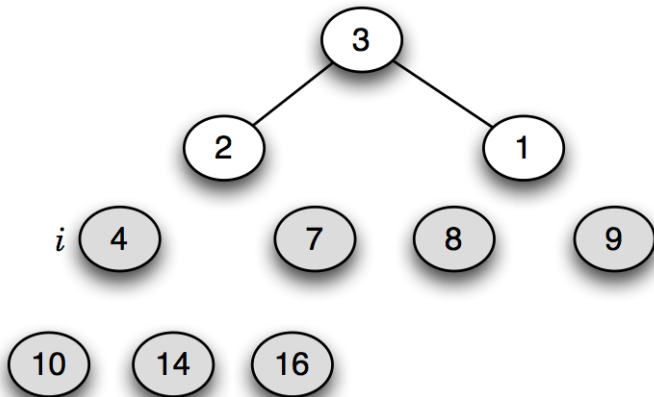
Heap use - heapsort



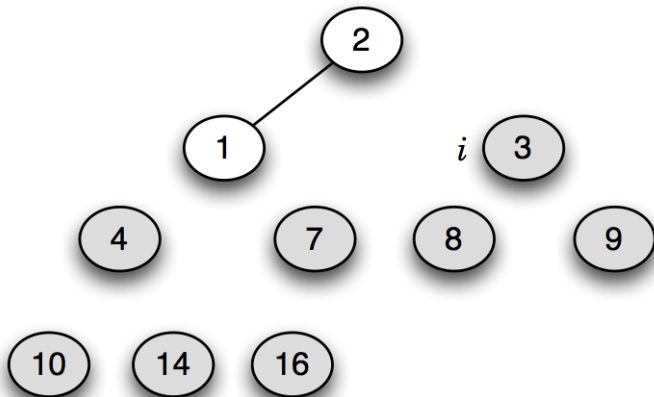
Heap use - heapsort



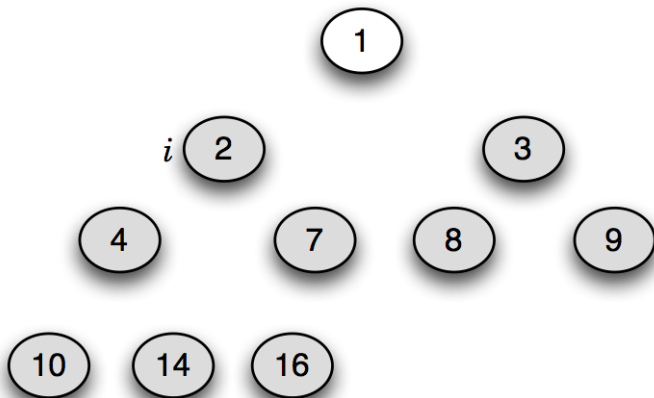
Heap use - heapsort



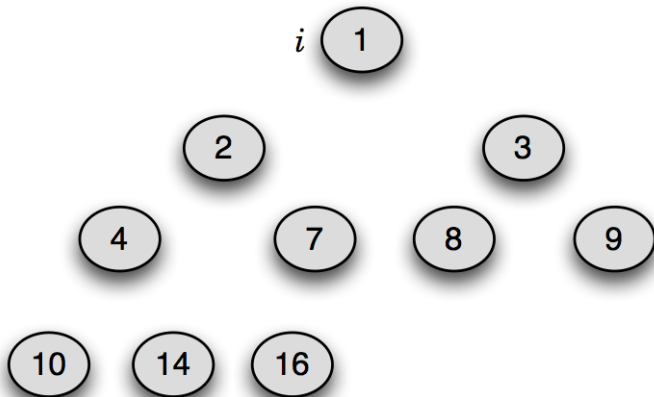
Heap use - heapsort



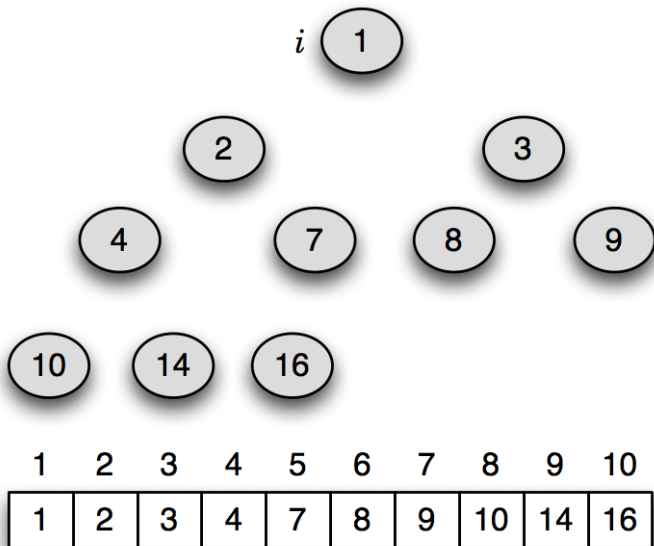
Heap use - heapsort



Heap use - heapsort



Heap use - heapsort



HEAPSORT algorithm

HEAPSORT(A)

	<i>cost</i>	<i>times</i>
1 BUILD-MAX-HEAP(A)	$\Theta(n)$	1
2 for $i \leftarrow \text{length}[A]$ downto 2	$\Theta(1)$	n
3 do exchange $A[1] \leftrightarrow A[i]$	$\Theta(1)$	$n - 1$
4 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	$n - 1$
5 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	$n - 1$

HEAPSORT algorithm

HEAPSORT(A)

	<i>cost</i>	<i>times</i>
1 BUILD-MAX-HEAP(A)	$\Theta(n)$	1
2 for $i \leftarrow \text{length}[A]$ downto 2	$\Theta(1)$	n
3 do exchange $A[1] \leftrightarrow A[i]$	$\Theta(1)$	$n - 1$
4 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	$n - 1$
5 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	$n - 1$

HEAPSORT algorithm

HEAPSORT(A)

	<i>cost</i>	<i>times</i>
1 BUILD-MAX-HEAP(A)	$\Theta(n)$	1
2 for $i \leftarrow \text{length}[A]$ downto 2	$\Theta(1)$	n
3 do exchange $A[1] \leftrightarrow A[i]$	$\Theta(1)$	$n - 1$
4 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	$n - 1$
5 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	$n - 1$

HEAPSORT algorithm

HEAPSORT(A)

	<i>cost</i>	<i>times</i>
1 BUILD-MAX-HEAP(A)	$\Theta(n)$	1
2 for $i \leftarrow \text{length}[A]$ downto 2	$\Theta(1)$	n
3 do exchange $A[1] \leftrightarrow A[i]$	$\Theta(1)$	$n - 1$
4 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	$n - 1$
5 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	$n - 1$

HEAPSORT algorithm

HEAPSORT(A)

	<i>cost</i>	<i>times</i>
1 BUILD-MAX-HEAP(A)	$\Theta(n)$	1
2 for $i \leftarrow \text{length}[A]$ downto 2	$\Theta(1)$	n
3 do exchange $A[1] \leftrightarrow A[i]$	$\Theta(1)$	$n - 1$
4 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	$n - 1$
5 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	$n - 1$

HEAPSORT algorithm

HEAPSORT(A)

	<i>cost</i>	<i>times</i>
1 BUILD-MAX-HEAP(A)	$\Theta(n)$	1
2 for $i \leftarrow \text{length}[A]$ downto 2	$\Theta(1)$	n
3 do exchange $A[1] \leftrightarrow A[i]$	$\Theta(1)$	$n - 1$
4 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	$n - 1$
5 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	$n - 1$

HEAPSORT algorithm

HEAPSORT(A)

	<i>cost</i>	<i>times</i>
1 BUILD-MAX-HEAP(A)	$\Theta(n)$	1
2 for $i \leftarrow \text{length}[A]$ downto 2	$\Theta(1)$	n
3 do exchange $A[1] \leftrightarrow A[i]$	$\Theta(1)$	$n - 1$
4 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	$n - 1$
5 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	$n - 1$

Worst-case analysis

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

Contents

1 Heapsort

- Introduction
- Heap data structure
- Heap algorithms
- Heapsort algorithm
- **Priority queue**
- Conclusion

Heap use - priority queue

Definition (Priority Queue)

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.

- 1 **INSERT**(S, x) – inserts the element x into the set S . This operation could be written as $S \leftarrow S \cup \{x\}$.
- 2 **MAXIMUM**(S) – returns the element of S with the largest key.
- 3 **EXTRACT-MAX**(S) – removes and returns the element of S with the largest key.
- 4 **INCREASE-KEY**(S, x, k) – increases the value of element x 's key to k . Assume $k \geq x$'s current value.

Heap use - priority queue

Definition (Priority Queue)

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.

- ❶ **INSERT**(S, x) – inserts the element x into the set S . This operation could be written as $S \leftarrow S \cup \{x\}$.
 - ❷ **MAXIMUM**(S) – returns the element of S with the largest key.
 - ❸ **EXTRACT-MAX**(S) – removes and returns the element of S with the largest key.
 - ❹ **INCREASE-KEY**(S, x, k) – increases the value of element x 's key to k . Assume $k \geq x$'s current value.
- Used in many scheduling applications where jobs or tasks are scheduled according to priority.

Heap use - priority queue

Definition (Priority Queue)

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.

- ❶ **INSERT**(S, x) – inserts the element x into the set S . This operation could be written as $S \leftarrow S \cup \{x\}$.
 - ❷ **MAXIMUM**(S) – returns the element of S with the largest key.
 - ❸ **EXTRACT-MAX**(S) – removes and returns the element of S with the largest key.
 - ❹ **INCREASE-KEY**(S, x, k) – increases the value of element x 's key to k . Assume $k \geq x$'s current value.
- Used in many scheduling applications where jobs or tasks are scheduled according to priority.
 - A FIFO queue is a priority queue where the priority is inversely proportional to time of arrival.

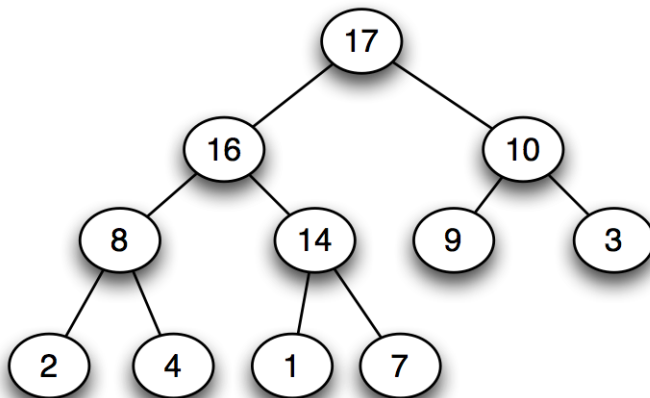
Heap use - priority queue

Definition (Priority Queue)

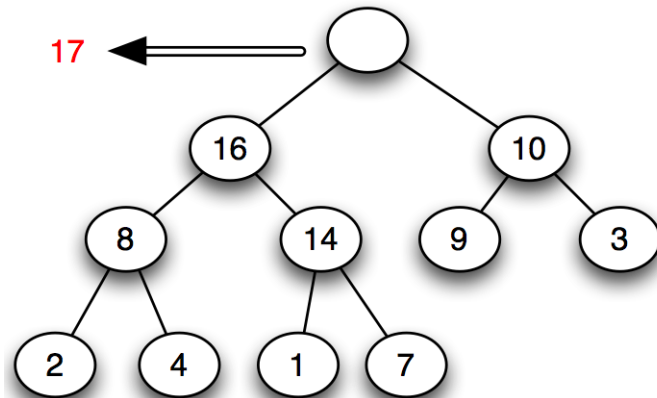
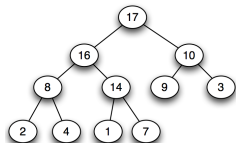
A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a *key*. A **max-priority queue** supports the following operations.

- ❶ **INSERT**(S, x) – inserts the element x into the set S . This operation could be written as $S \leftarrow S \cup \{x\}$.
 - ❷ **MAXIMUM**(S) – returns the element of S with the largest key.
 - ❸ **EXTRACT-MAX**(S) – removes and returns the element of S with the largest key.
 - ❹ **INCREASE-KEY**(S, x, k) – increases the value of element x 's key to k . Assume $k \geq x$'s current value.
- Used in many scheduling applications where jobs or tasks are scheduled according to priority.
 - A FIFO queue is a priority queue where the priority is inversely proportional to time of arrival.
 - A LIFO stack is a priority queue where the priority is proportional to time of arrival.

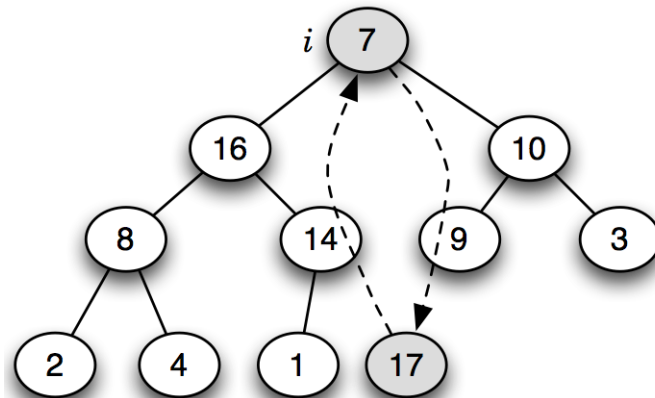
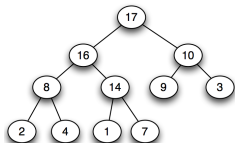
Example of HEAP-EXTRACT-MAX operation



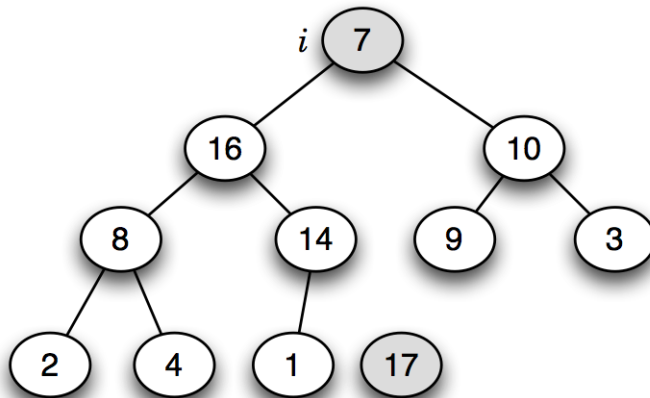
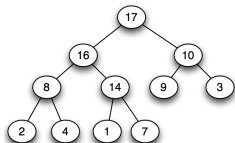
Example of HEAP-EXTRACT-MAX operation



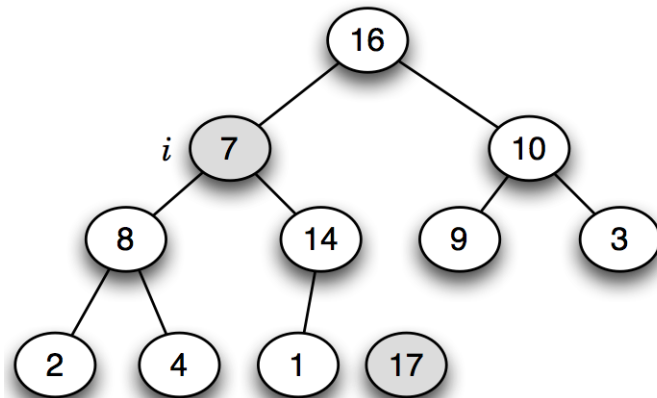
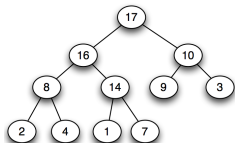
Example of HEAP-EXTRACT-MAX operation



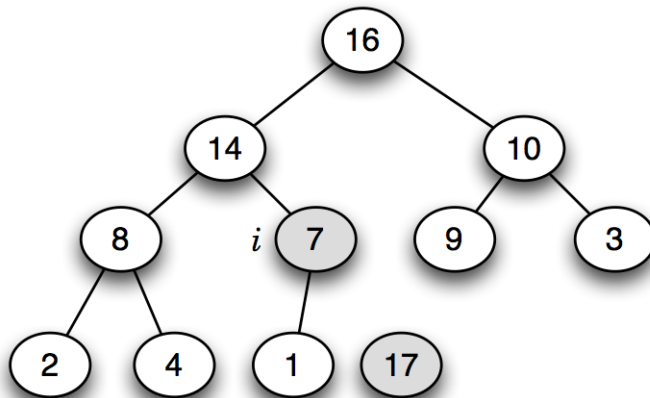
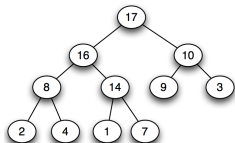
Example of HEAP-EXTRACT-MAX operation



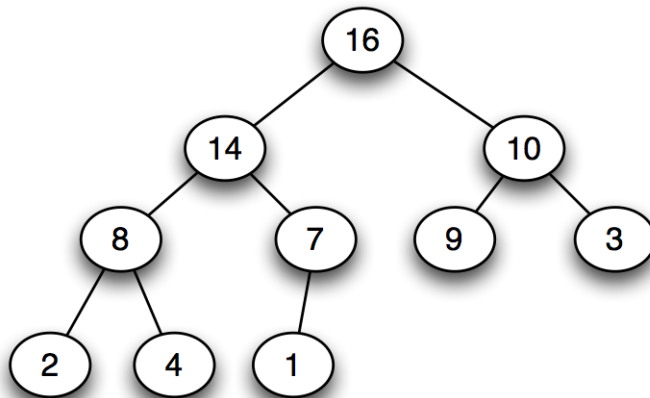
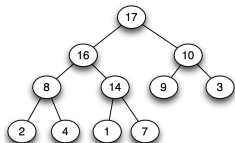
Example of HEAP-EXTRACT-MAX operation



Example of HEAP-EXTRACT-MAX operation



Example of HEAP-EXTRACT-MAX operation



HEAP-EXTRACT-MAX algorithm

EXTRACT-MAX(A)

	<i>cost</i>	<i>times</i>
1 if $\text{heap-size}[A] < 1$	$\Theta(1)$	1
2 then error “heap underflow”	$\Theta(1)$	1
3 $\text{max} \leftarrow A[1]$	$\Theta(1)$	1
4 $A[1] \leftarrow A[\text{heap-size}[A]]$	$\Theta(1)$	1
5 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	1
6 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	1

HEAP-EXTRACT-MAX algorithm

EXTRACT-MAX(A)

	<i>cost</i>	<i>times</i>
1 if $\text{heap-size}[A] < 1$	$\Theta(1)$	1
2 then error “heap underflow”	$\Theta(1)$	1
3 $\text{max} \leftarrow A[1]$	$\Theta(1)$	1
4 $A[1] \leftarrow A[\text{heap-size}[A]]$	$\Theta(1)$	1
5 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	1
6 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	1

HEAP-EXTRACT-MAX algorithm

EXTRACT-MAX(A)

	<i>cost</i>	<i>times</i>
1 if $\text{heap-size}[A] < 1$	$\Theta(1)$	1
2 then error "heap underflow"	$\Theta(1)$	1
3 $\text{max} \leftarrow A[1]$	$\Theta(1)$	1
4 $A[1] \leftarrow A[\text{heap-size}[A]]$	$\Theta(1)$	1
5 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	1
6 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	1

HEAP-EXTRACT-MAX algorithm

EXTRACT-MAX(A)

	<i>cost</i>	<i>times</i>
1 if $\text{heap-size}[A] < 1$	$\Theta(1)$	1
2 then error "heap underflow"	$\Theta(1)$	1
3 $\text{max} \leftarrow A[1]$	$\Theta(1)$	1
4 $A[1] \leftarrow A[\text{heap-size}[A]]$	$\Theta(1)$	1
5 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	1
6 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	1

HEAP-EXTRACT-MAX algorithm

EXTRACT-MAX(A)

	<i>cost</i>	<i>times</i>
1 if $\text{heap-size}[A] < 1$	$\Theta(1)$	1
2 then error "heap underflow"	$\Theta(1)$	1
3 $\text{max} \leftarrow A[1]$	$\Theta(1)$	1
4 $A[1] \leftarrow A[\text{heap-size}[A]]$	$\Theta(1)$	1
5 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	1
6 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	1

HEAP-EXTRACT-MAX algorithm

EXTRACT-MAX(A)

	<i>cost</i>	<i>times</i>
1 if $\text{heap-size}[A] < 1$	$\Theta(1)$	1
2 then error “heap underflow”	$\Theta(1)$	1
3 $\text{max} \leftarrow A[1]$	$\Theta(1)$	1
4 $A[1] \leftarrow A[\text{heap-size}[A]]$	$\Theta(1)$	1
5 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	1
6 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	1

HEAP-EXTRACT-MAX algorithm

EXTRACT-MAX(A)

	<i>cost</i>	<i>times</i>
1 if $\text{heap-size}[A] < 1$	$\Theta(1)$	1
2 then error "heap underflow"	$\Theta(1)$	1
3 $\text{max} \leftarrow A[1]$	$\Theta(1)$	1
4 $A[1] \leftarrow A[\text{heap-size}[A]]$	$\Theta(1)$	1
5 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$\Theta(1)$	1
6 MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	1

Worst-case analysis

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

Conclusion

- **Heap** plays a very important role in many algorithms, either used directly or as part of a priority queue implementation.
- If the size of a queue is known in advance, then an array representation (using a fixed size array) provides compact storage coupled with fast operations.
- Even if the size of the heap is not known in advance, “intelligent” resizing can still provide good benefits.
- Heapsort is a natural application of Heap with two very important properties – $\Theta(n \lg n)$ complexity, and in-place sorting.