CSE 221: Algorithms

Heapsort

Mumit Khan

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: May 14, 2010

Contents

Contents

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

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.

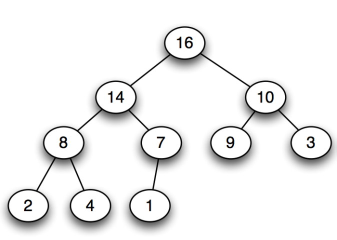
Heap-ordered tree

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



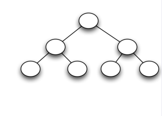
Heap-ordered tree

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*)



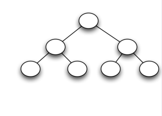
Heap-ordered tree

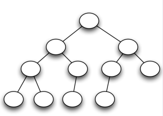
Definition

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*)





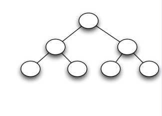
Heap-ordered tree

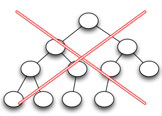
Definition

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*)





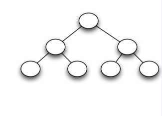
Heap-ordered tree

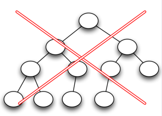
Definition

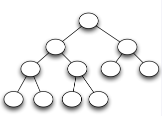
A binary tree is heap-ordered if:

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*)







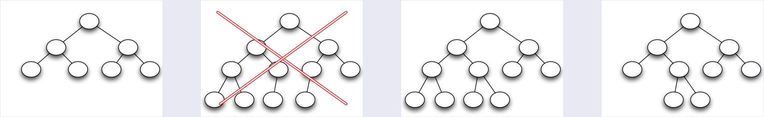
Heap-ordered tree

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*)



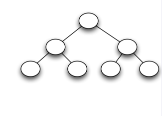
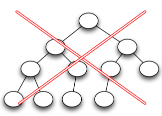
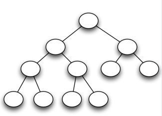
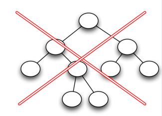
Heap-ordered tree

Definition

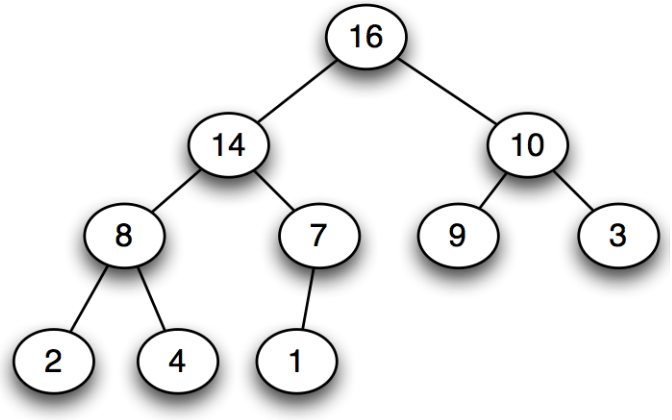
A binary tree is heap-ordered if:

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

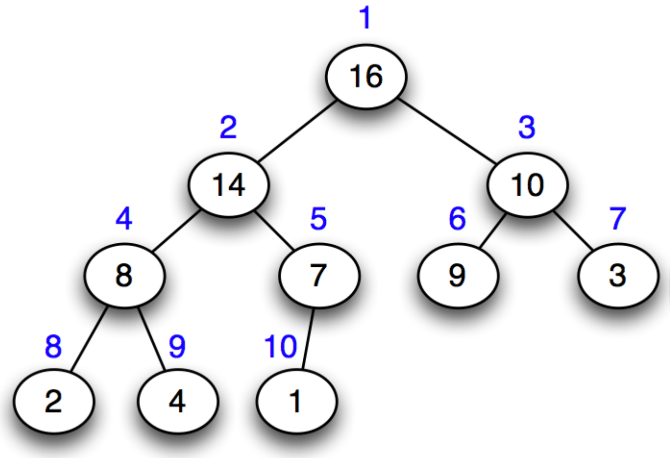
Example of complete tree (or *not*)

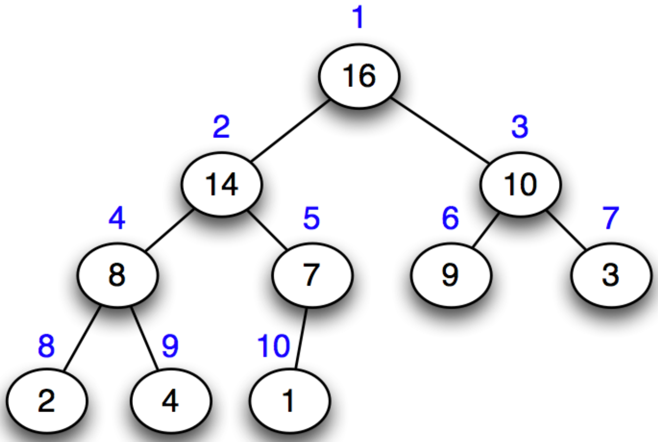
Heap – array representation of heap-ordered tree



Heap – array representation of heap-ordered tree

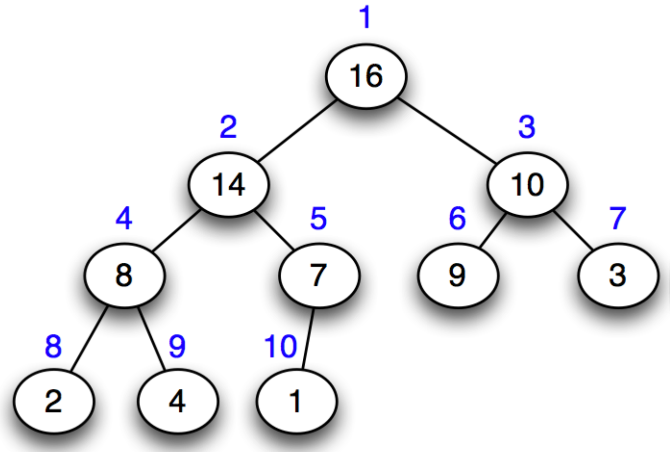


Heap – array representation of heap-ordered tree



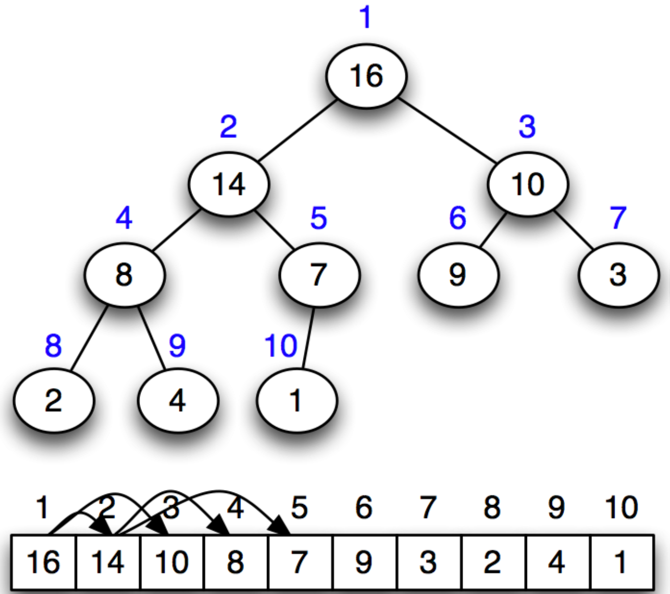
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap – array representation of heap-ordered tree

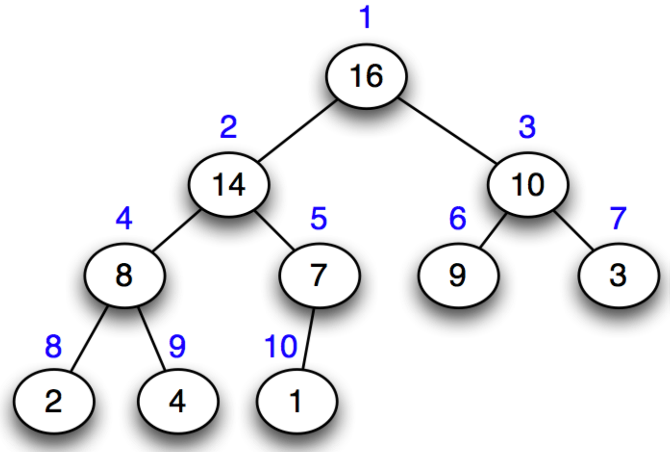


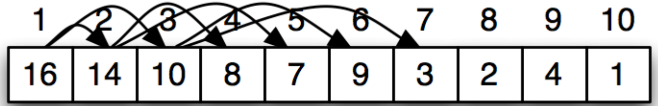
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap – array representation of heap-ordered tree

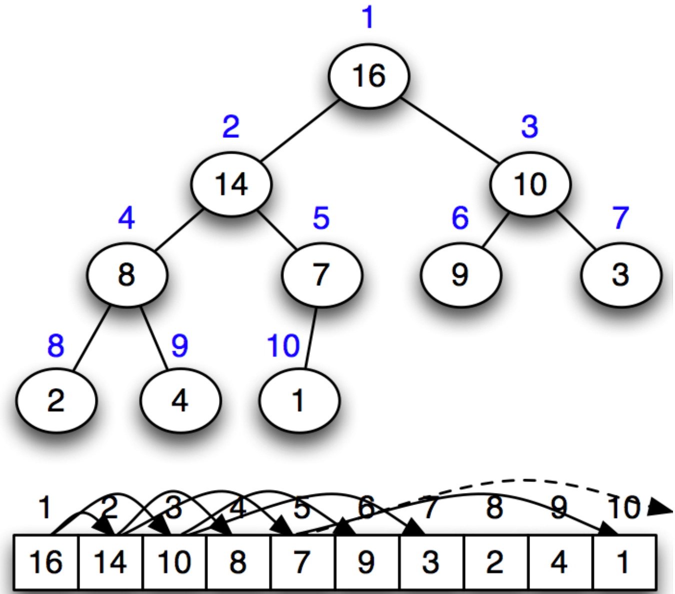


Heap – array representation of heap-ordered tree

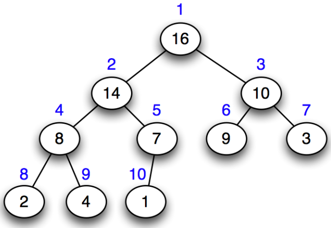




Heap – array representation of heap-ordered tree



Heap – accessing parent and children

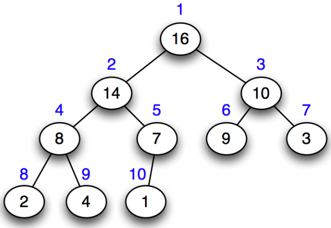


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

MAXIMUM(A)

  return $A[1]$

Heap – accessing parent and children



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

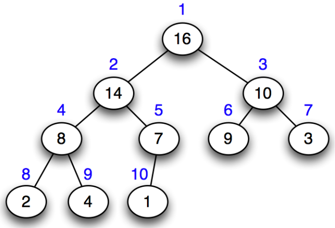
$\text{MAXIMUM}(A)$

return $A[1]$

$\text{PARENT}(i)$

return $\lfloor i/2 \rfloor$

Heap – accessing parent and children



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

MAXIMUM(A)

return $A[1]$

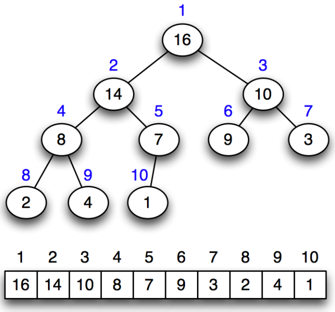
PARENT(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

Heap – accessing parent and children



MAXIMUM(A)

return $A[1]$

PARENT(i)

return $\lfloor i/2 \rfloor$

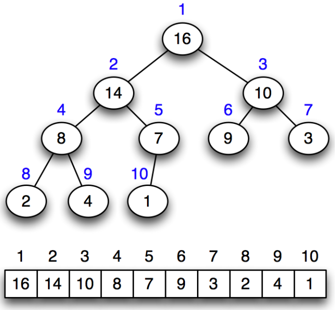
LEFT(i)

return $2i$

Question

What if LEFT$(i) > n$?

Heap – accessing parent and children



MAXIMUM(A)

return $A[1]$

PARENT(i)

return $\lfloor i/2 \rfloor$

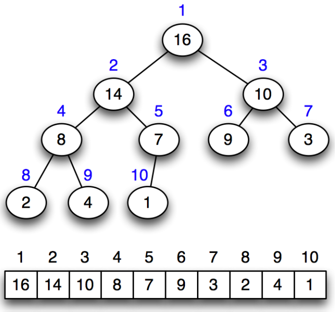
LEFT(i)

return $2i$

RIGHT(i)

return $2i + 1$

Heap – accessing parent and children



$\text{MAXIMUM}(A)$
return $A[1]$

$\text{PARENT}(i)$
return $\lfloor i/2 \rfloor$

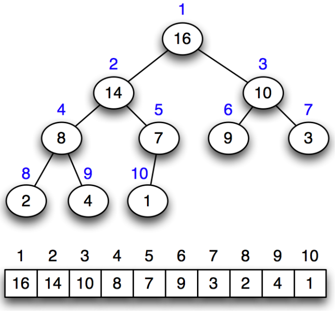
$\text{LEFT}(i)$
return $2i$

$\text{RIGHT}(i)$
return $2i + 1$

Question

What if $\text{RIGHT}(i) > n$?

Heap – accessing parent and children



$\text{MAXIMUM}(A)$

return $A[1]$

$\text{PARENT}(i)$

return $\lfloor i/2 \rfloor$

$\text{LEFT}(i)$

return $2i$

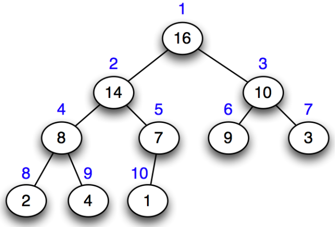
$\text{RIGHT}(i)$

return $2i + 1$

Lemma

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

Heap – accessing parent and children



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

$\text{MAXIMUM}(A)$

return $A[1]$

$\text{PARENT}(i)$

return $\lfloor i/2 \rfloor$

$\text{LEFT}(i)$

return $2i$

$\text{RIGHT}(i)$

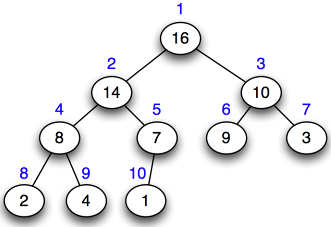
return $2i + 1$

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



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

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

Operations on heap

1. 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. 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. MAX-HEAP-INSERT(A, key) – Insert key in the heap A, maintaining A's heap property.

Operations on heap

1. 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. MAX-HEAP-INSERT(A, key) – Insert key in the heap A, maintaining A's heap property.
3. BUILD-MAX-HEAP(A) – Build a max heap given an array A.

Operations on heap

1. 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. MAX-HEAP-INSERT(A, key) – Insert key in the heap A, maintaining A's heap property.
3. BUILD-MAX-HEAP(A) – Build a max heap given an array A.
4. HEAPSORT(A) – Sort the elements in array A using the heap operations.

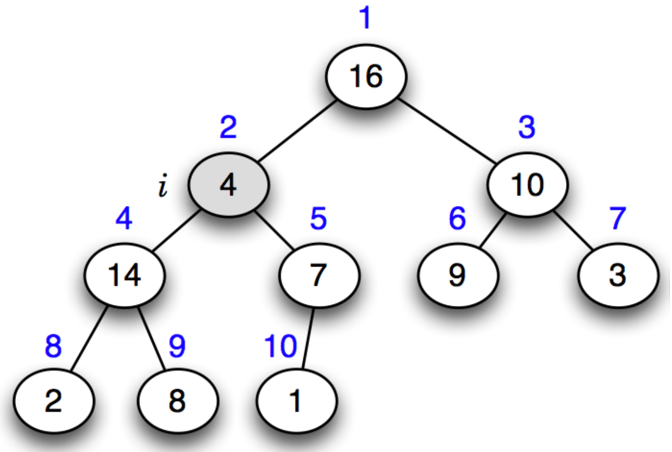
Operations on heap

1. 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. MAX-HEAP-INSERT(A, key) – Insert key in the heap A, maintaining A's heap property.
3. BUILD-MAX-HEAP(A) – Build a max heap given an array A.
4. HEAPSORT(A) – Sort the elements in array A using the heap operations.
5. HEAP-INCREASE-KEY(A, i, 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.

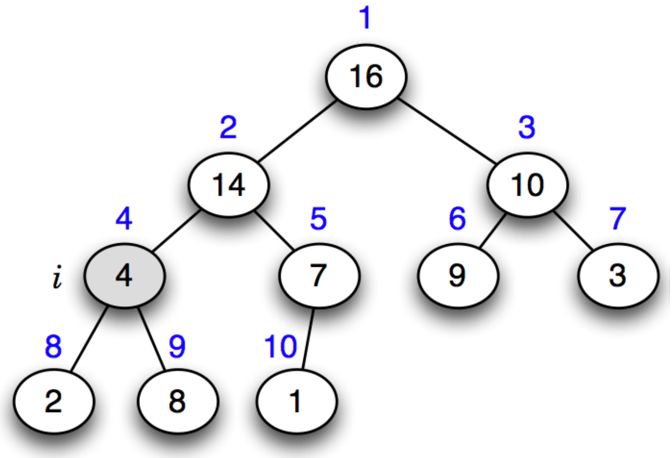
Operations on heap

1. 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. MAX-HEAP-INSERT(A, key) – Insert key in the heap A, maintaining A's heap property.
3. BUILD-MAX-HEAP(A) – Build a max heap given an array A.
4. HEAPSORT(A) – Sort the elements in array A using the heap operations.
5. HEAP-INCREASE-KEY(A, i, 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.
6. HEAP-EXTRACT-MAX(A) – Extract the largest element from heap A.

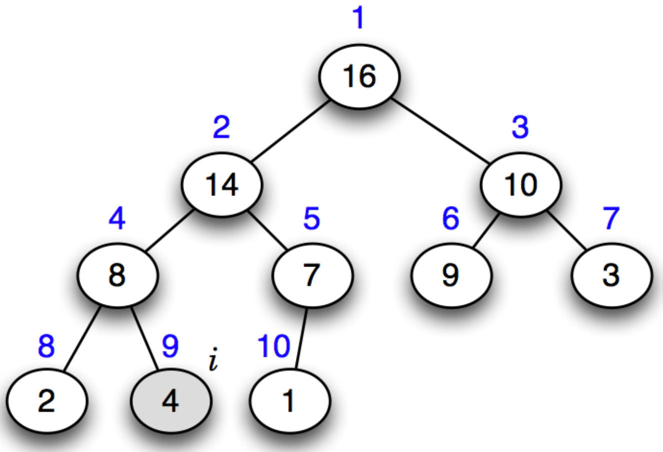
Example of MAX-HEAPIFY ("sink") operation



Example of MAX-HEAPIFY ("sink") operation



Example of MAX-HEAPIFY ("sink") operation



MAX-HEAPIFY algorithm

MAX-HEAPIFY(A, i)

$l \leftarrow left(i)$
$r \leftarrow right(i)$
if $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
  then $largest \leftarrow l$
  else $largest \leftarrow i$
if $r \leq heap\text{-}size[A]$ and $A[r] > A[largest]$
  then $largest \leftarrow r$
if $largest \neq i$
  then exchange $A[i] \leftrightarrow A[largest]$
    MAX-HEAPIFY$(A, largest)$

MAX-HEAPIFY algorithm

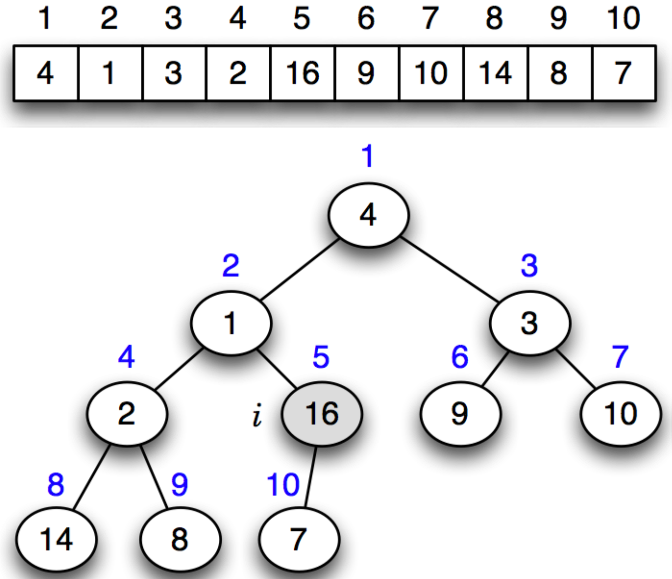
MAX-HEAPIFY(A, i)

$l \leftarrow left(i)$
$r \leftarrow right(i)$
if $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
  then $largest \leftarrow l$
  else $largest \leftarrow i$
if $r \leq heap\text{-}size[A]$ and $A[r] > A[largest]$
  then $largest \leftarrow r$
if $largest \neq i$
  then exchange $A[i] \leftrightarrow A[largest]$
    MAX-HEAPIFY$(A, largest)$

Analysis – second way

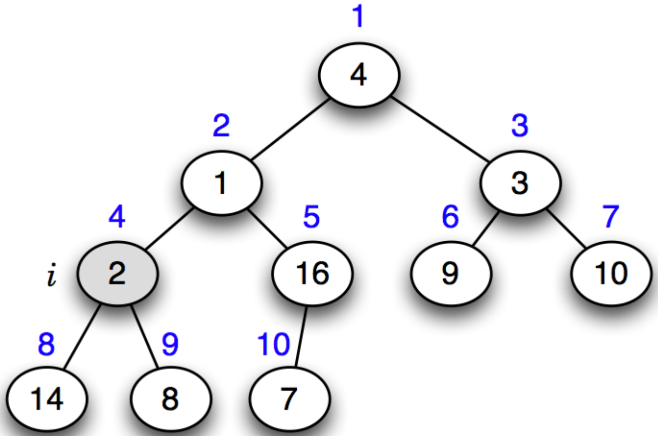
The running time of MAX-HEAPIFY on a node of height h is $T(n) = O(h) = O(\lg n)$.

Example of BUILD-MAX-HEAP ("heapify") operation



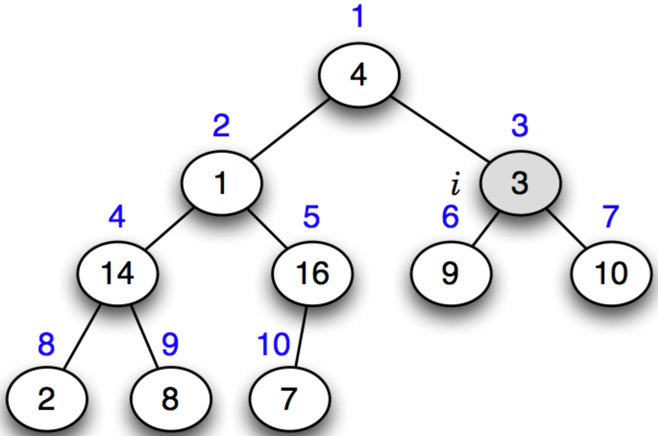
Example of BUILD-MAX-HEAP ("heapify") operation

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



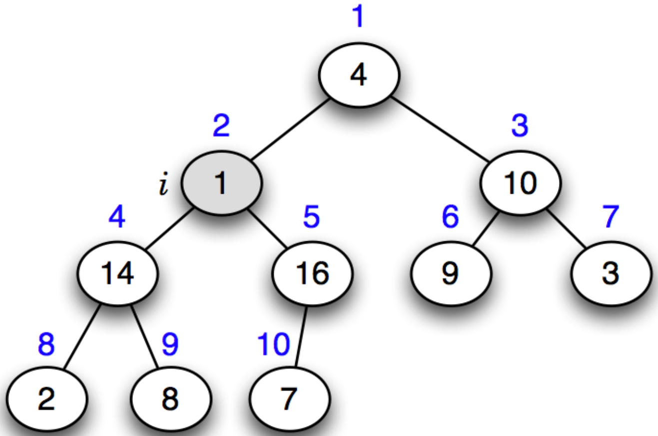
Example of BUILD-MAX-HEAP ("heapify") operation

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



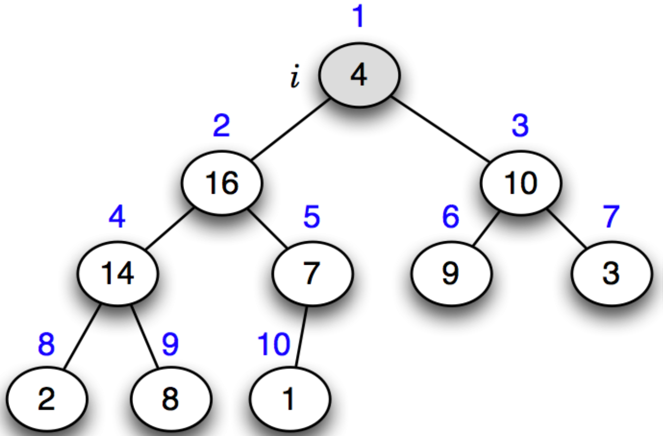
Example of BUILD-MAX-HEAP (“heapify”) operation

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



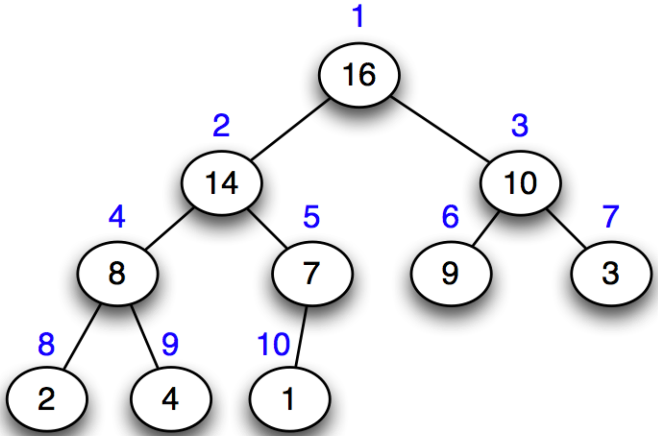
Example of BUILD-MAX-HEAP ("heapify") operation

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)

$heap\text{-}size[A]] \leftarrow length[A]$

for $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1

  do MAX-HEAPIFY(A, i)

BUILD-MAX-HEAP algorithm

BUILD-MAX-HEAP(A)

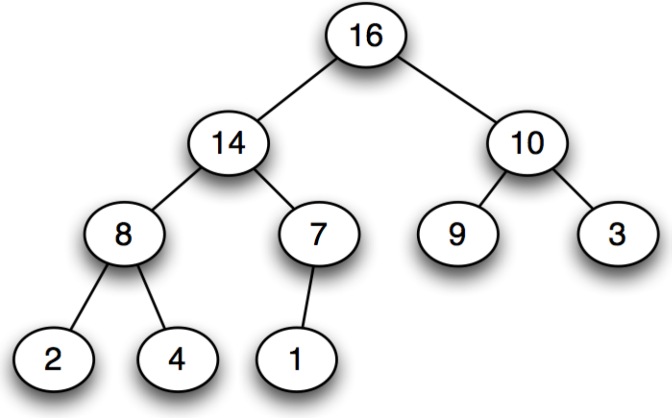
1 $heap\text{-}size[A]] \leftarrow length[A]$
2 **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A, i)

Analysis

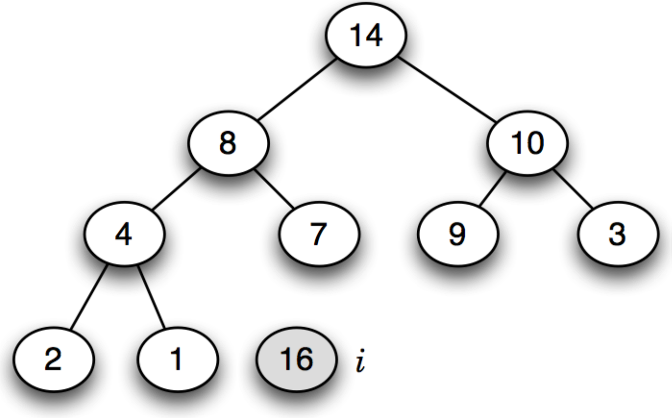
$T(n) = O(n)$ (see textbook for details)

Contents

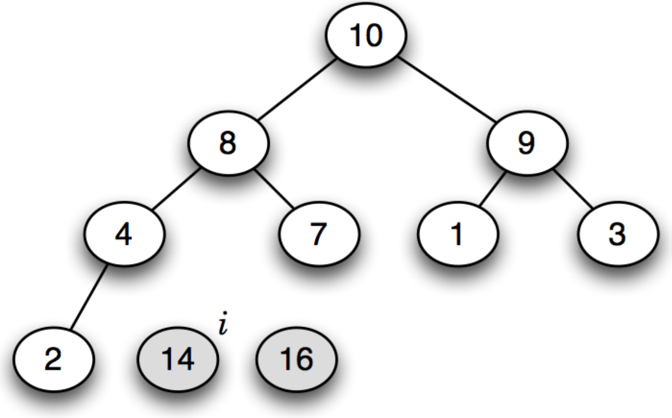
Heap use - heapsort



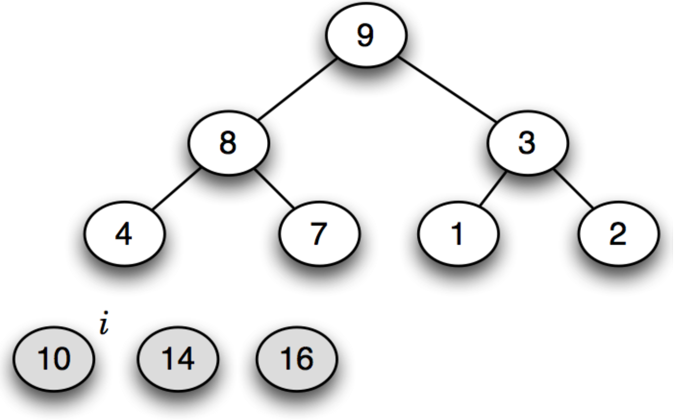
Heap use - heapsort



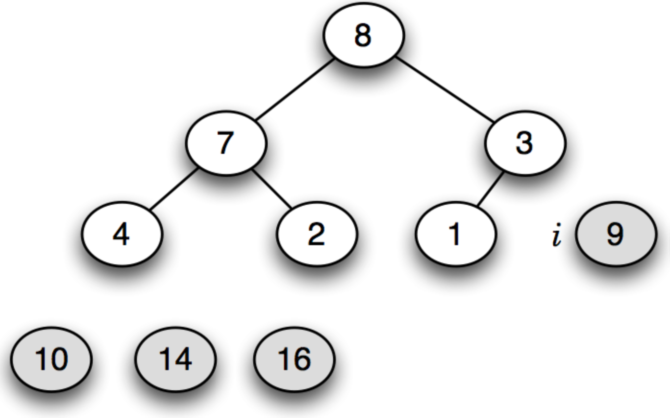
Heap use - heapsort



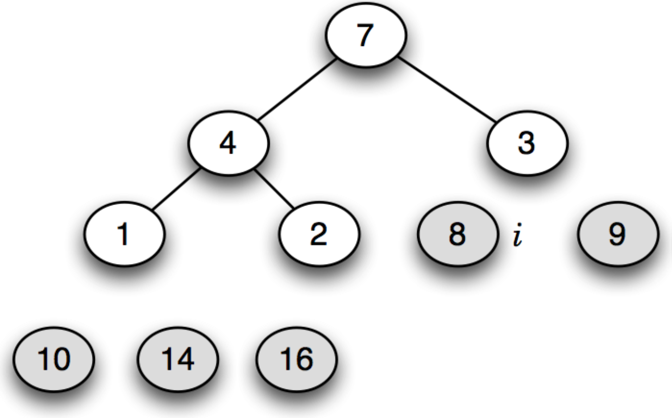
Heap use - heapsort



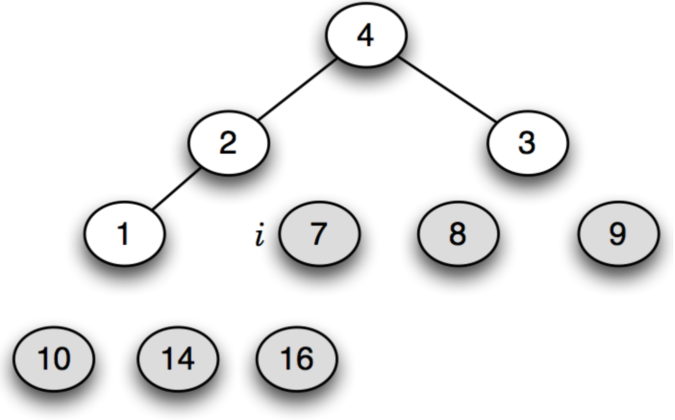
Heap use - heapsort



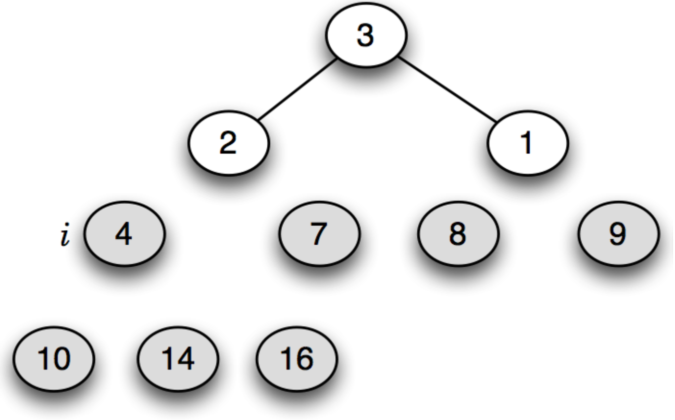
Heap use - heapsort



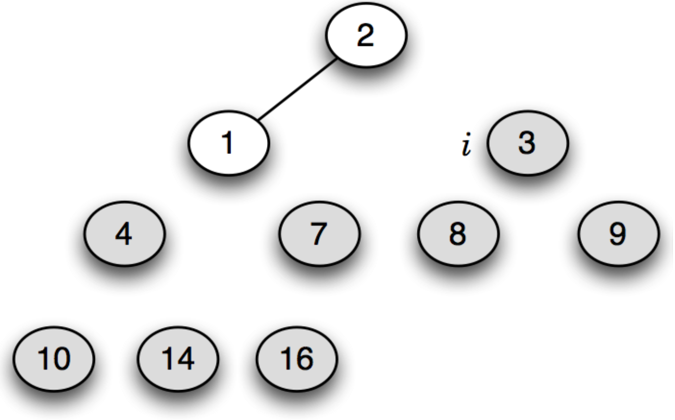
Heap use - heapsort



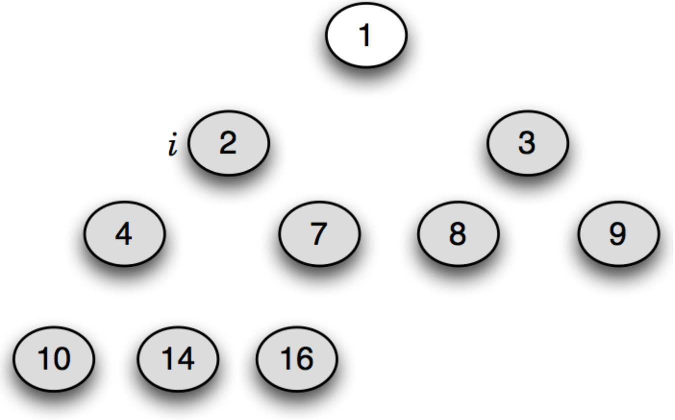
Heap use - heapsort



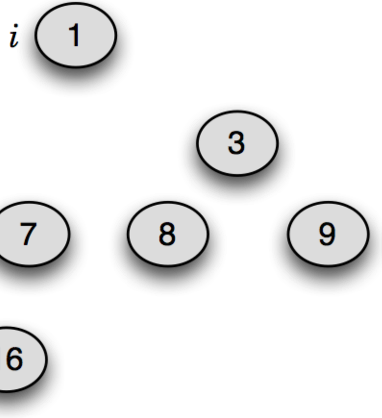
Heap use - heapsort



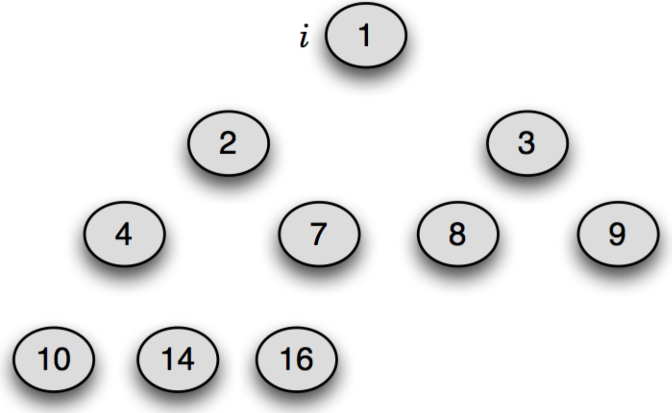
Heap use - heapsort



Heap use - heapsort



Heap use - heapsort



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

HEAPSORT algorithm

HEAPSORT(A)

	cost	*times*
BUILD-MAX-HEAP(A)	$\Theta(n)$	1
for $i \leftarrow$ *length*$[A]$ **downto** 2	$\Theta(1)$	n
do exchange $A[1] \leftrightarrow A[i]$	$\Theta(1)$	$n-1$
heap-size$[A] \leftarrow$ *heap-size*$[A] - 1$	$\Theta(1)$	$n-1$
MAX-HEAPIFY$(A, 1)$	$\Theta(\lg n)$	$n-1$

HEAPSORT algorithm

HEAPSORT(A)

	cost	*times*
BUILD-MAX-HEAP(A)	$\Theta(n)$	1
for $i \leftarrow$ *length*$[A]$ **downto** 2	$\Theta(1)$	n
do exchange $A[1] \leftrightarrow A[i]$	$\Theta(1)$	$n-1$
heap-size$[A] \leftarrow$ *heap-size*$[A] - 1$	$\Theta(1)$	$n-1$
MAX-HEAPIFY$(A, 1)$	$\Theta(\lg n)$	$n-1$

HEAPSORT algorithm

HEAPSORT(A)

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

HEAPSORT algorithm

HEAPSORT(A)

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

HEAPSORT algorithm

HEAPSORT(A)

	cost	*times*
BUILD-MAX-HEAP(A)	$\Theta(n)$	1
for $i \leftarrow$ *length*$[A]$ **downto** 2	$\Theta(1)$	n
do exchange $A[1] \leftrightarrow A[i]$	$\Theta(1)$	$n-1$
heap-size$[A] \leftarrow$ *heap-size*$[A] - 1$	$\Theta(1)$	$n-1$
MAX-HEAPIFY($A, 1$)	$\Theta(\lg n)$	$n-1$

HEAPSORT algorithm

HEAPSORT(A)

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

HEAPSORT algorithm

HEAPSORT(A)

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

Worst-case analysis

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