Algorithms

Heaps and priority queues

Emanuele Rodolà
rodola@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

2nd semester a.y. 2019/2020 · March 26, 2020

Heap

Data structures allow us to organize data for efficient use.

$$(1, 2, 3, 4, 9, 10, 7, 16, 8, 14) \quad \overset{\text{build}}{\Longrightarrow}$$

16	14	10	8	7	9	3	2	4	1

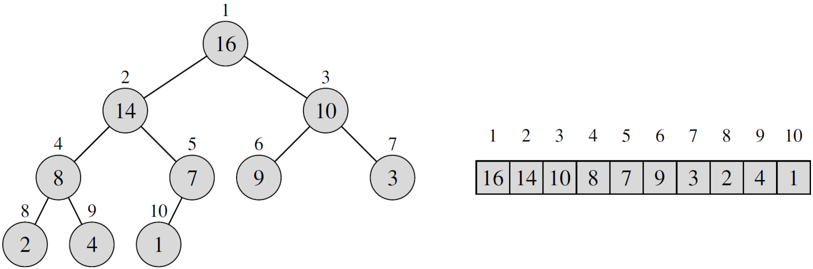
Heap

Data structures allow us to organize data for efficient use.

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

Heap

Data structures allow us to organize data for efficient use.

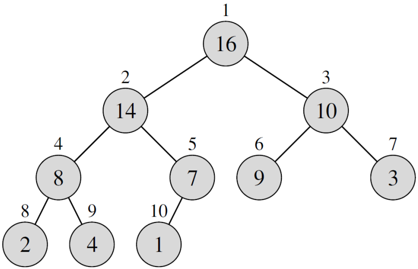


(a) (b)

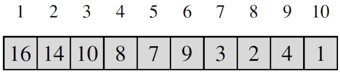
- $\text{left}(i) = 2i$
- $\text{right}(i) = 2i + 1$

Heap

Data structures allow us to organize data for efficient use.



(a)

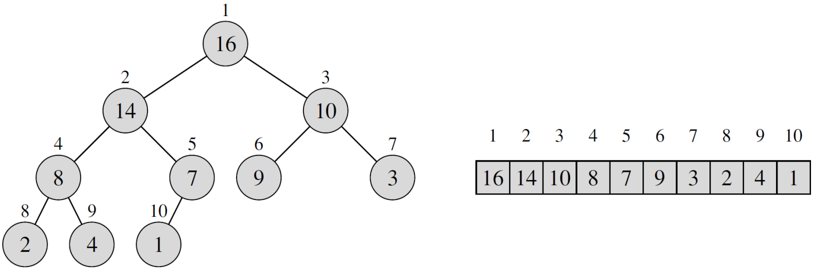


(b)

- $\text{left}(i) = 2i$
- $\text{right}(i) = 2i + 1$
- $\text{parent}(i) = \lfloor i/2 \rfloor$

Heap

Data structures allow us to organize data for efficient use.



(a) (b)

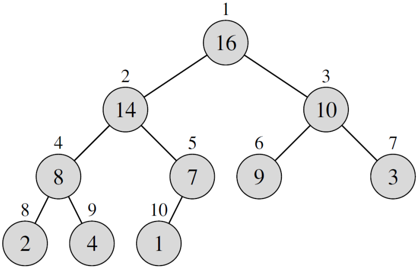
max-heap property:

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

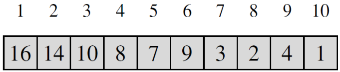
which means that root = largest element.

Heap

Data structures allow us to organize data for efficient use.



(a)



(b)

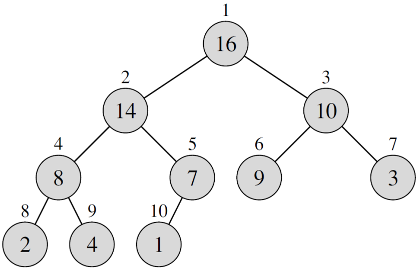
min-heap property:

$$A[\mathrm{parent}(i)] \leq A[i]$$

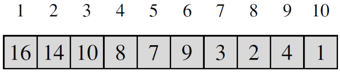
which means that root = smallest element.

Heap

Data structures allow us to organize data for efficient use.



(a)

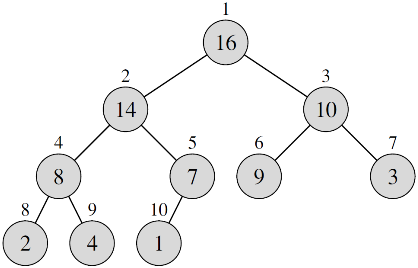


(b)

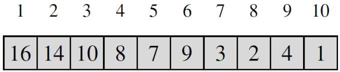
height of a node: # edges of the longest path to a leaf

Heap

Data structures allow us to organize data for efficient use.



(a)



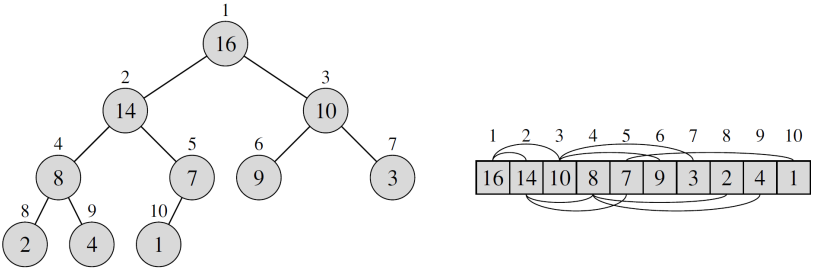
(b)

height of a node: # edges of the longest path to a leaf

height of the heap = height of the root = $\Theta(\lg n)$

Heap

Data structures allow us to organize data for efficient use.

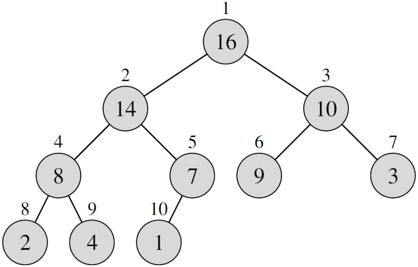


(a) (b)

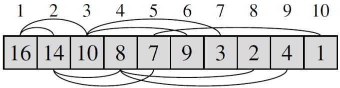
The heap can be seen as a binary tree or as an array.

Heap

Data structures allow us to organize data for efficient use.



(a)



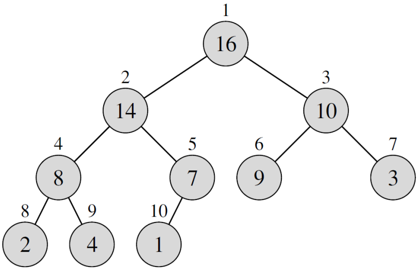
(b)

The heap can be seen as a binary tree or as an array.

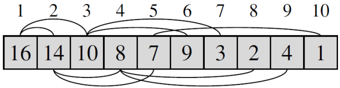
Each level (except the last one) must be filled completely left-to-right.

Heap

Data structures allow us to organize data for efficient use.



(a)



(b)

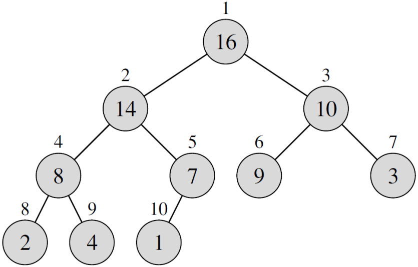
The heap can be seen as a binary tree or as an array.

Each level (except the last one) must be filled completely left-to-right.

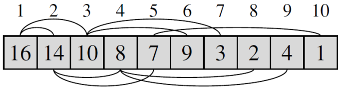
The number of leaves is $n - \lfloor \frac{n}{2} \rfloor$.

Heap

Data structures allow us to organize data for efficient use.



(a)



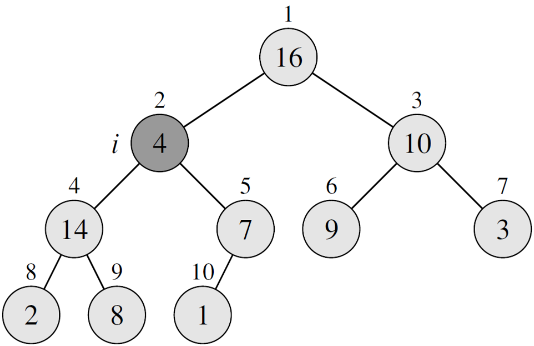
(b)

We will see the following operations:

- Build a (max-)heap from some input array.
- Maintain the (max-)heap property.
- Construct a priority queue on top of a heap.

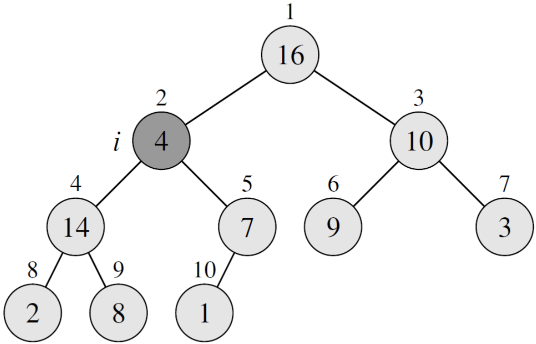
"Heapify": Maintaining the heap property

We are given as input the root index i of the sub-tree to check.



"Heapify": Maintaining the heap property

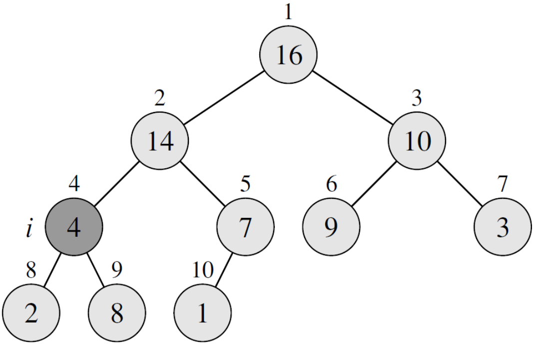
We are given as input the root index i of the sub-tree to check.



Assumption: $\text{left}(i)$ and $\text{right}(i)$ are roots of valid heaps.

"Heapify": Maintaining the heap property

We are given as input the root index i of the sub-tree to check.

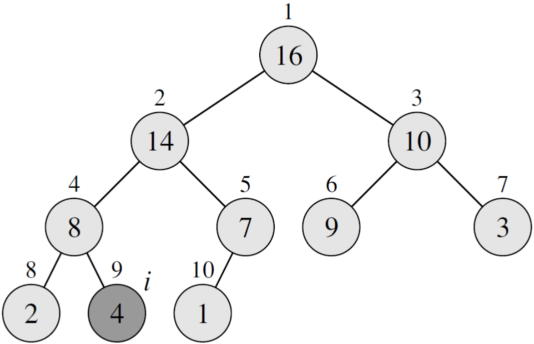


Assumption: $\mathrm{left}(i)$ and $\mathrm{right}(i)$ are roots of valid heaps.

Recursively exchange the violating node with its largest child.

"Heapify": Maintaining the heap property

We are given as input the root index i of the sub-tree to check.



Assumption: $\mathrm{left}(i)$ and $\mathrm{right}(i)$ are roots of valid heaps.

Recursively exchange the violating node with its largest child.

Maintaining the heap property

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

If $A[i]$ is the largest, then the sub-tree is already a max heap.

Maintaining the heap property

MAX-HEAPIFY(A, i)

$\Theta(1)$

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

Maintaining the heap property

```
MAX-HEAPIFY(A, i)
       l ← LEFT(i)
       r ← RIGHT(i)
       if l ≤ heap-size[A] and A[l] > A[i]
          then largest ← l
Θ(1)      else largest ← i
       if r ≤ heap-size[A] and A[r] > A[largest]
          then largest ← r
       if largest ≠ i
          then exchange A[i] ↔ A[largest]
T(2n/3) →       MAX-HEAPIFY(A, largest)
```

Maintaining the heap property

MAX-HEAPIFY(A, i)

$\Theta(1)$

$l \leftarrow$ LEFT(i)
$r \leftarrow$ RIGHT(i)
if $l \leq$ *heap-size*$[A]$ and $A[l] > A[i]$
    then *largest* $\leftarrow l$
    else *largest* $\leftarrow i$
if $r \leq$ *heap-size*$[A]$ and $A[r] > A[$*largest*$]$
    then *largest* $\leftarrow r$
if *largest* $\neq i$
    then exchange $A[i] \leftrightarrow A[$*largest*$]$

$T(2n/3) \rightarrow$ MAX-HEAPIFY$(A,$ *largest*$)$

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

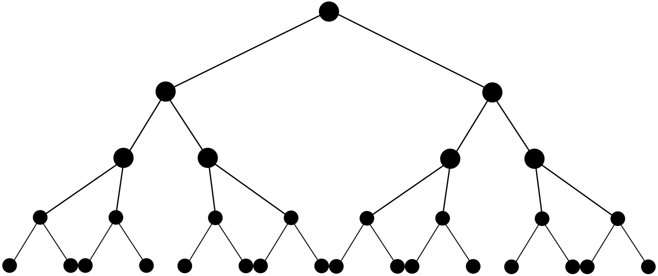
Maintaining the heap property

```
         MAX-HEAPIFY(A, i)
         1  l ← LEFT(i)
         2  r ← RIGHT(i)
         3  if l ≤ heap-size[A] and A[l] > A[i]
         4     then largest ← l
Θ(1)     5     else largest ← i
         6  if r ≤ heap-size[A] and A[r] > A[largest]
         7     then largest ← r
         8  if largest ≠ i
         9     then exchange A[i] ↔ A[largest]
T(2n/3) → 10         MAX-HEAPIFY(A, largest)
```

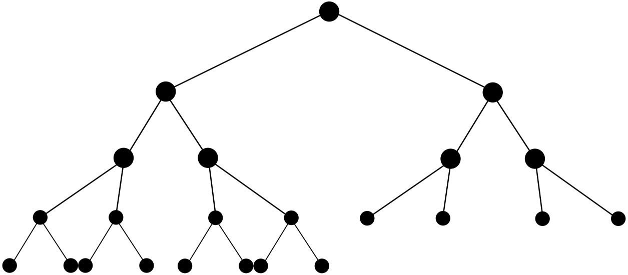
$$T(n) = O(\lg n)$$

by case (2) of the master theorem.

Maximum size of sub-tree in a heap

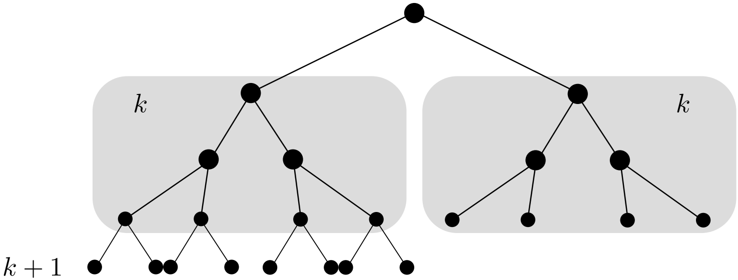


Maximum size of sub-tree in a heap



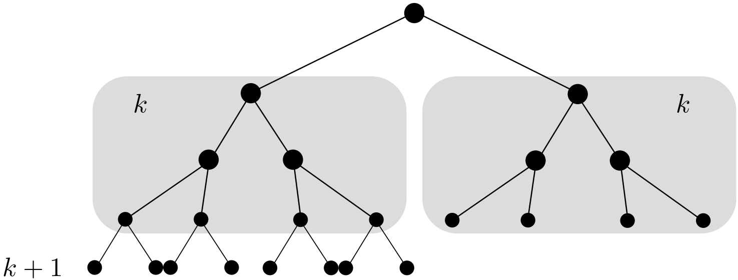
- A valid heap tree **must** be completely filled at each level.

Maximum size of sub-tree in a heap



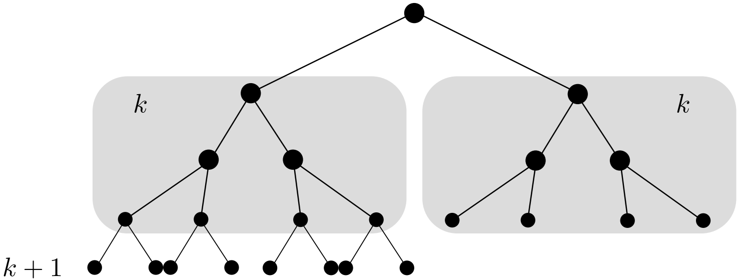
- A valid heap tree must be completely filled at each level.
- The largest sub-tree has $\leq k + (k+1)$ nodes.

Maximum size of sub-tree in a heap



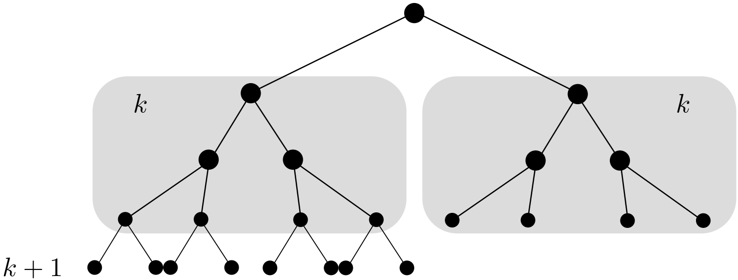
- A valid heap tree must be completely filled at each level.
- The largest sub-tree has $\leq 2k+1$ nodes.

Maximum size of sub-tree in a heap



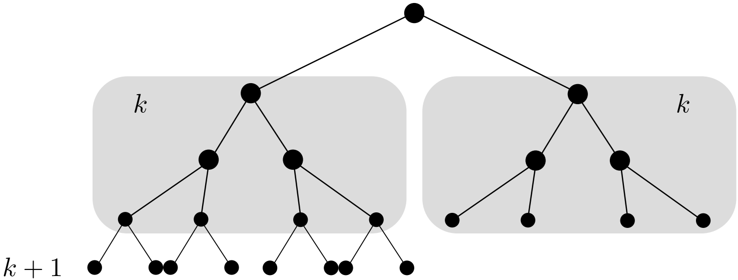
- A valid heap tree must be completely filled at each level.
- The largest sub-tree has $\leq 2k+1$ nodes.
- The entire tree has $n = 1 + 2k + (k+1) = 3k+2$ nodes.

Maximum size of sub-tree in a heap



- A valid heap tree must be completely filled at each level.
- The largest sub-tree has $\leq 2k+1$ nodes.
- The entire tree has $n = 1 + 2k + (k+1) = 3k+2$ nodes.
- Thus, the largest sub-tree has $\leq \frac{2k+1}{3k+2} n$ nodes.

Maximum size of sub-tree in a heap



- A valid heap tree must be completely filled at each level.
- The largest sub-tree has $\leq 2k+1$ nodes.
- The entire tree has $n = 1 + 2k + (k+1) = 3k+2$ nodes.
- Thus, the largest sub-tree has $\leq \frac{2k+1}{3k+2}n$ nodes.
- For arbitrary k, we get $\lim_{k\to\infty} \frac{2k+1}{3k+2}n = \frac{2}{3}n$.

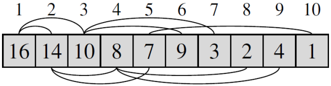
Building a heap

How to convert a given input array into a (max-)heap?

Building a heap

How to convert a given input array into a (max-)heap?

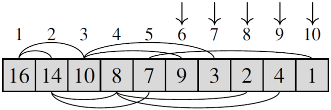
The leaf indices are known a priori: $(\lfloor n/2 \rfloor + 1) \cdots n$



Building a heap

How to convert a given input array into a (max-)heap?

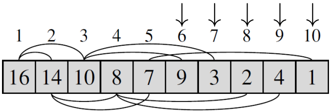
The leaf indices are known a priori: $(\lfloor n/2 \rfloor + 1) \cdots n$



Building a heap

How to convert a given input array into a (max-)heap?

The leaf indices are known a priori: $(\lfloor n/2 \rfloor + 1) \cdots n$

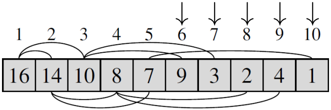


General idea: run max-heapify on the internal nodes.

Building a heap

How to convert a given input array into a (max-)heap?

The leaf indices are known a priori: $(\lfloor n/2 \rfloor + 1) \cdots n$



General idea: run max-heapify on the internal nodes.

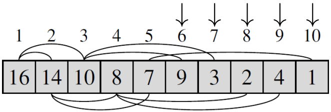
$\textsc{Build-Max-Heap}(A)$

$\quad heap\text{-}size[A] \leftarrow length[A]$
$\quad \textbf{for } i \leftarrow \lfloor length[A]/2 \rfloor \textbf{ downto } 1$
$\quad\quad \textbf{do } \textsc{Max-Heapify}(A, i)$

Building a heap

How to convert a given input array into a (max-)heap?

The leaf indices are known a priori: $(\lfloor n/2 \rfloor + 1) \cdots n$



General idea: run max-heapify on the internal nodes.

BUILD-MAX-HEAP(A)

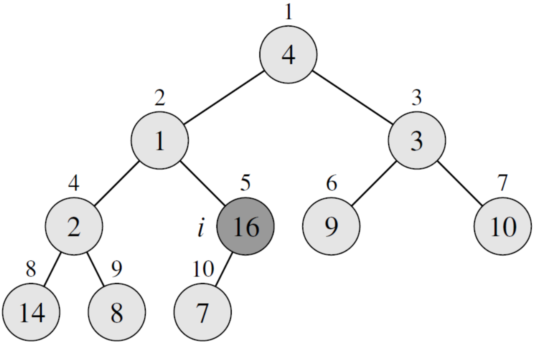
$heap\text{-}size[A] \leftarrow length[A]$
for $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
    do MAX-HEAPIFY(A, i)

$O(n)$ calls to a $O(\lg n)$ algorithm: $T(n) = O(n \lg n)$.

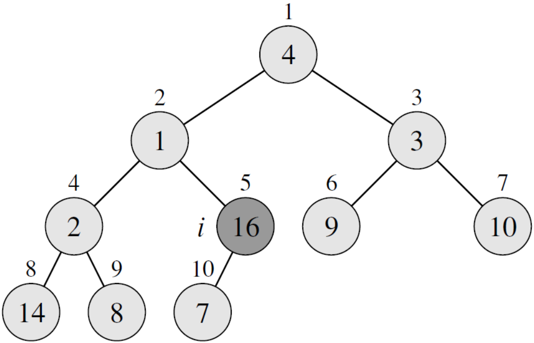
Building a heap

A	4	1	3	2	16	9	10	14	8	7

Building a heap

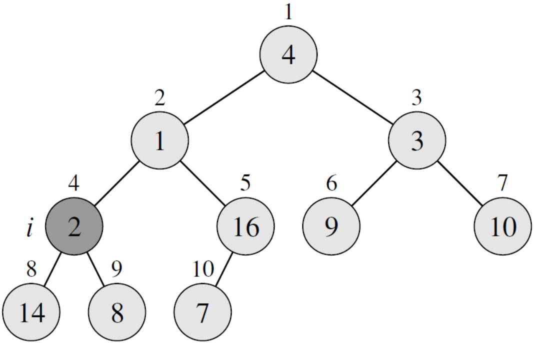


Building a heap

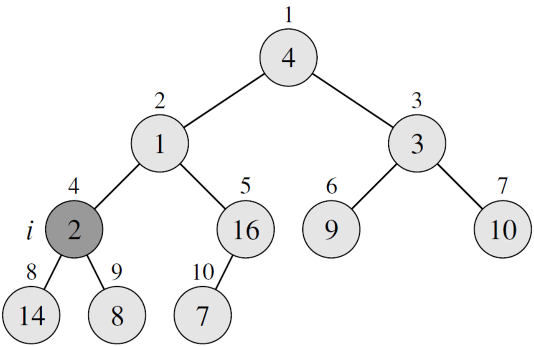


The sub-tree is already a heap.

Building a heap

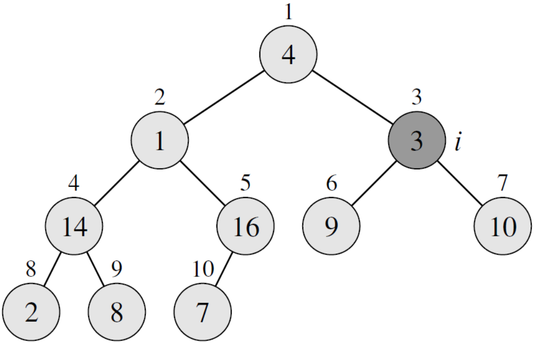


Building a heap

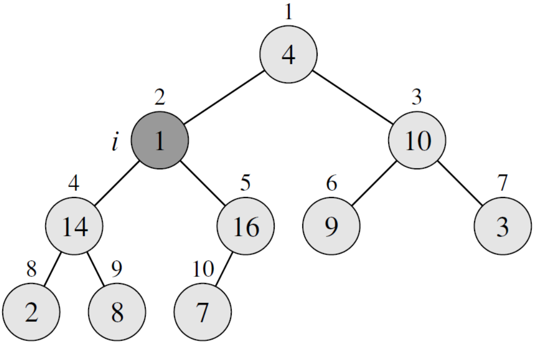


The sub-tree does not satisfy the heap property – run heapify.

Building a heap



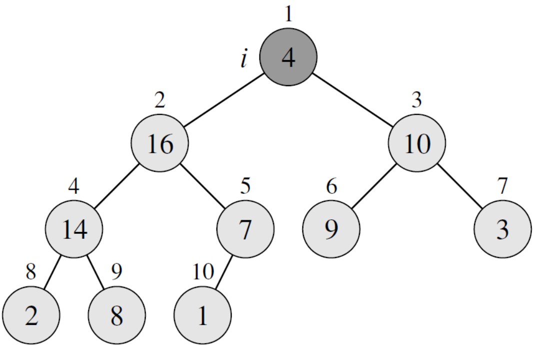
Building a heap



By proceeding bottom-up, we can always run heapify, which assumes that the sub-trees rooted at $\text{left}(i)$ and $\text{right}(i)$ are valid.

This way, after heapify we can guarantee that i is the root of a heap.

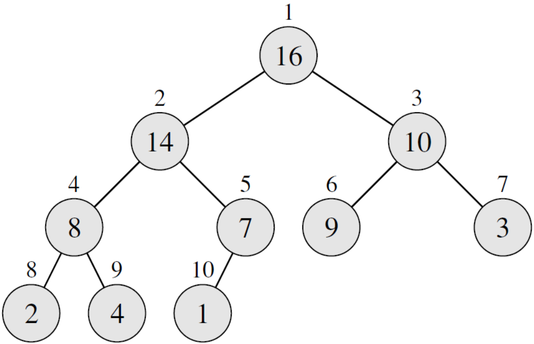
Building a heap



By proceeding bottom-up, we can always run heapify, which assumes that the sub-trees rooted at $\mathrm{left}(i)$ and $\mathrm{right}(i)$ are valid.

This way, after heapify we can guarantee that i is the root of a heap.

Building a heap



By proceeding bottom-up, we can always run heapify, which assumes that the sub-trees rooted at $\mathrm{left}(i)$ and $\mathrm{right}(i)$ are valid.

This way, after heapify we can guarantee that i is the root of a heap.

Priority queues

Example:

Let us be given a set of tasks, each with a priority.

When a task is done, pass to the next with highest priority.

Priority queues

Example:

Let us be given a set of tasks, each with a priority.

When a task is done, pass to the next with highest priority.

A priority queue is an efficient data structure to do this.

Priority queues

Example:

Let us be given a set of tasks, each with a priority.

When a task is done, pass to the next with highest priority.

A priority queue is an efficient data structure to do this.

We need to support the following operations:

- Insert a new element + priority.
- Find the highest-priority element.
- Extract the highest-priority element.
- Increase the priority of a given element.

Priority queues

Example:

Let us be given a set of tasks, each with a priority.

When a task is done, pass to the next with highest priority.

A priority queue is an efficient data structure to do this.

We need to support the following operations:

- Insert a new element + priority.
- Find the highest-priority element.
- Extract the highest-priority element.
- Increase the priority of a given element.

We will implement the priority queue on top of a heap.

Priority queues

The heap stores the priority values of each element.

In practice the element itself is also attached, but we ignore it here.

Priority queues

The heap stores the priority values of each element.

In practice the element itself is also attached, but we ignore it here.

Find the highest priority element in $\Theta(1)$ time:

HEAP-MAXIMUM(A)

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

Priority queues

The heap stores the priority values of each element.

In practice the element itself is also attached, but we ignore it here.

Find the highest priority element in $\Theta(1)$ time:

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

Extract the highest-priority element in $O(\lg n)$ time:

```
HEAP-EXTRACT-MAX(A)
    if heap-size[A] < 1
        then error "heap underflow"
    max ← A[1]
discard    A[1] ← A[heap-size[A]]
the root   heap-size[A] ← heap-size[A] − 1
    MAX-HEAPIFY(A, 1)
    return max
```

Priority queues

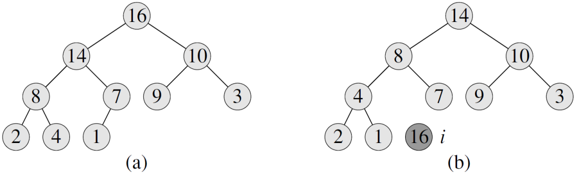
The heap stores the priority values of each element.

In practice the element itself is also attached, but we ignore it here.

Find the highest priority element in $\Theta(1)$ time:

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

Extract the highest-priority element in $O(\lg n)$ time:



discarding the root

Priority queues

Increase the priority of a given element in $O(\lg n)$ time:

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
   then error "new key is smaller than current key"
A[i] ← key
while i > 1 and A[PARENT(i)] < A[i]
     do exchange A[i] ↔ A[PARENT(i)]
        i ← PARENT(i)
```

Priority queues

Increase the priority of a given element in $O(\lg n)$ time:

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2     then error "new key is smaller than current key"
3  A[i] ← key
4  while i > 1 and A[PARENT(i)] < A[i]
5       do exchange A[i] ↔ A[PARENT(i)]
6          i ← PARENT(i)
```

Line 3 can violate the max-heap property, depending on the value of key.

Priority queues

Increase the priority of a given element in $O(\lg n)$ time:

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      then error "new key is smaller than current key"
3  A[i] ← key
4  while i > 1 and A[PARENT(i)] < A[i]
5       do exchange A[i] ↔ A[PARENT(i)]
6          i ← PARENT(i)
```

Line 3 can violate the max-heap property, depending on the value of key.

If so, the correct index for key must be up in the tree.

Lines 4-6 find the correct index by moving the value in an upward path.

Priority queues

Insert a new element + priority in $O(\lg n)$ time:

MAX-HEAP-INSERT(A, key)

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

Priority queues

Insert a new element + priority in $O(\lg n)$ time:

```
MAX-HEAP-INSERT(A, key)
  heap-size[A] ← heap-size[A] + 1
  A[heap-size[A]] ← −∞
  HEAP-INCREASE-KEY(A, heap-size[A], key)
```

The value of $-\infty$ ensures that line 1 of increase is false.

Priority queues

Insert a new element + priority in $O(\lg n)$ time:

```
MAX-HEAP-INSERT(A, key)
    heap-size[A] ← heap-size[A] + 1
    A[heap-size[A]] ← −∞
    HEAP-INCREASE-KEY(A, heap-size[A], key)
```

The value of $-\infty$ ensures that line 1 of increase is false.

To summarize:

operation	**cost**
find	$\Theta(1)$
extract	$O(\lg n)$
increase	$O(\lg n)$
insert	$O(\lg n)$

Suggested reading

Chapters 6.1, 6.2, 6.3 and 6.5 of:

"Introduction to Algorithms – 2nd Ed.", Cormen et al.