CMPSC 465: LECTURE IX

Heap Operations & HeapSort

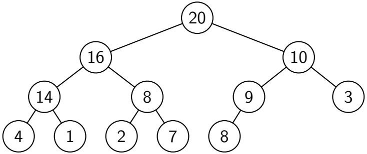
Ke Chen

September 17, 2025

Binary max-heap

Recall that a binary max-heap is a complete binary tree stored in an array that satisfies the Max-Heap Property: $H[i] \leq H[parent(i)]$.

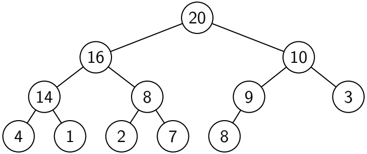
Example: $H[1..12] = 20, 16, 10, 14, 8, 9, 3, 4, 1, 2, 7, 8$.



Binary max-heap

Recall that a binary max-heap is a complete binary tree stored in an array that satisfies the Max-Heap Property: $H[i] \leq H[parent(i)]$.

Example: $H[1..12] = 20, 16, 10, 14, 8, 9, 3, 4, 1, 2, 7, 8.$

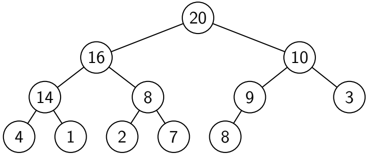


- A completely filled binary tree of height h (equivalently $h+1$ layers) has $2^0 + 2^1 + \cdots + 2^h =$

Binary max-heap

Recall that a binary max-heap is a complete binary tree stored in an array that satisfies the Max-Heap Property: $H[i] \leq H[parent(i)]$.

Example: $H[1..12] = 20, 16, 10, 14, 8, 9, 3, 4, 1, 2, 7, 8$.

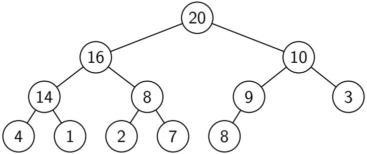


- A completely filled binary tree of height h (equivalently $h+1$ layers) has $2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$ nodes.

Binary max-heap

Recall that a binary max-heap is a complete binary tree stored in an array that satisfies the Max-Heap Property: $H[i] \leq H[parent(i)]$.

Example: $H[1..12] = 20, 16, 10, 14, 8, 9, 3, 4, 1, 2, 7, 8.$

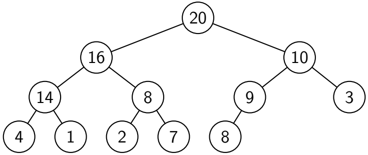


- A completely filled binary tree of height h (equivalently $h+1$ layers) has $2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$ nodes.
- So the number of elements n in a heap of height h satisfies $2^h \leq n \leq 2^{h+1} - 1$,

Binary max-heap

Recall that a binary max-heap is a complete binary tree stored in an array that satisfies the Max-Heap Property: $H[i] \leq H[parent(i)]$.

Example: $H[1..12] = 20, 16, 10, 14, 8, 9, 3, 4, 1, 2, 7, 8$.



- A completely filled binary tree of height h (equivalently $h + 1$ layers) has $2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$ nodes.
- So the number of elements n in a heap of height h satisfies $2^h \leq n \leq 2^{h+1} - 1$, or $h = O(\log n)$.

Heap operations

Suppose $H[1..n]$ is a binary max-heap.

GetMax Takes $O(1)$, all we have to do is return $H[1]$.

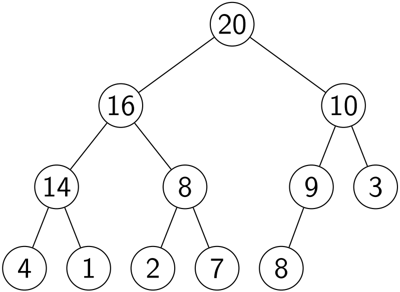
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Insertion

- Insert in the first available position $H[n+1]$.

Example Suppose we want to insert 18 in:



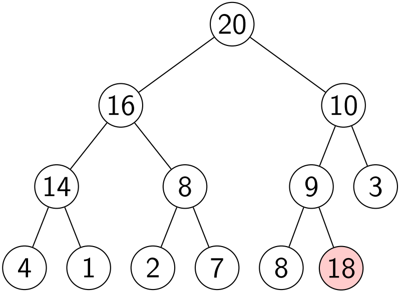
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Insertion

- Insert in the first available position $H[n+1]$.

Example Suppose we want to insert 18 in:



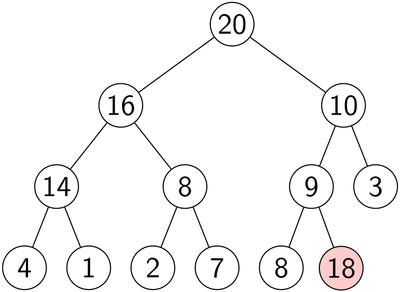
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Insertion

- Insert in the first available position $H[n+1]$.

Example Suppose we want to insert 18 in:



Too large, violate the max-heap property!

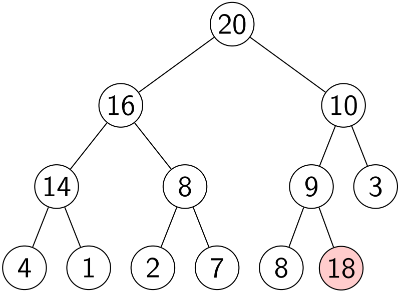
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Insertion

- Insert in the first available position $H[n+1]$.
- Idea Let the new element "bubble up" by swapping until it finds the right position.

Example Suppose we want to insert 18 in:



Too large, violate the max-heap property!

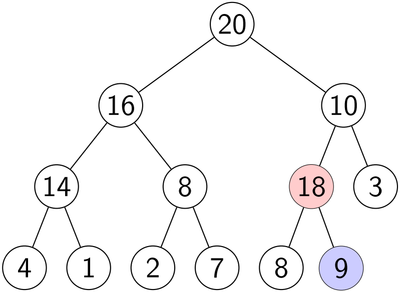
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Insertion

- Insert in the first available position $H[n+1]$.
- Idea Let the new element "bubble up" by swapping until it finds the right position.

Example Suppose we want to insert 18 in:



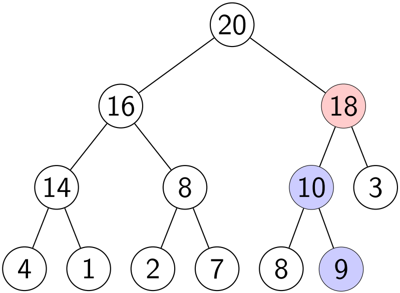
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Insertion

- Insert in the first available position $H[n+1]$.
- Idea Let the new element "bubble up" by swapping until it finds the right position.

Example Suppose we want to insert 18 in:



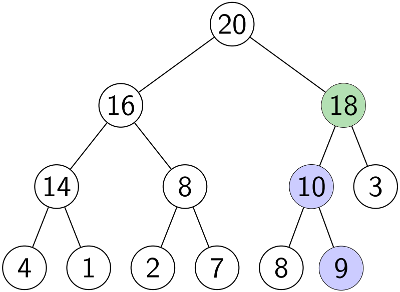
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Insertion

- Insert in the first available position $H[n+1]$.
- Idea Let the new element "bubble up" by swapping until it finds the right position.

Example Suppose we want to insert 18 in:



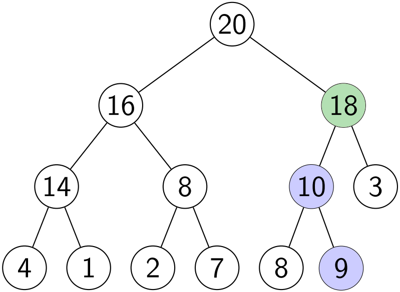
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Insertion

- Insert in the first available position $H[n+1]$.
- Idea Let the new element "bubble up" by swapping until it finds the right position.

Example Suppose we want to insert 18 in:



This process is called HeapifyUp.

Heap operations

Insert($H[1..n]$, key)

```
// Assume the array H still has available space
H[n + 1] = key
HeapifyUp(H, n + 1)
n = n + 1
```

HeapifyUp(H, i)

```
while i > 1 and H[i] > H[parent(i)] do
    swap H[i] and H[parent(i)]
    i = parent(i)
```

Correctness?

Heap operations

Insert($H[1..n]$, key)

```
// Assume the array H still has available space
H[n+1] = key
HeapifyUp(H, n+1)
n = n+1
```

HeapifyUp(H, i)

while $i > 1$ **and** $H[i] > H[parent(i)]$ **do**

  swap $H[i]$ and $H[parent(i)]$

  $i = parent(i)$

Correctness?

Time complexity?

HeapifyUp takes $O(h) = O(\log n)$ time, so does Insert.

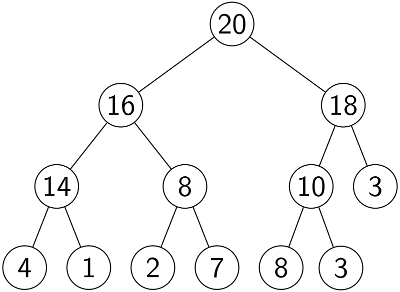
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Deletion at index i

- Replace the element at index i by the last element.

Example Suppose we want to remove the element at index 2:



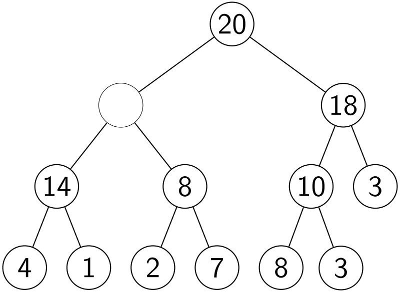
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Deletion at index i

- Replace the element at index i by the last element.

Example Suppose we want to remove the element at index 2:



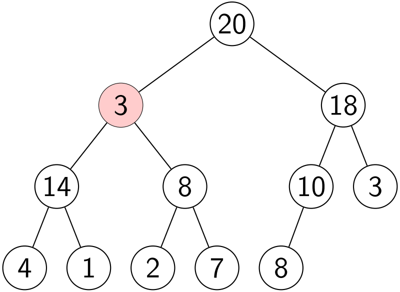
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Deletion at index i

- Replace the element at index i by the last element.

Example Suppose we want to remove the element at index 2:



Heap operations

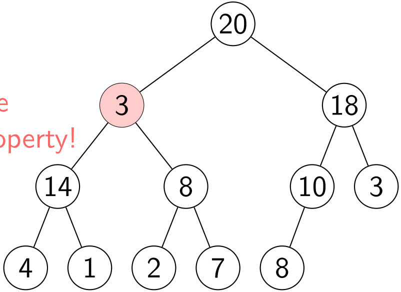
Suppose $H[1..n]$ is a binary max-heap.

Deletion at index i

- Replace the element at index i by the last element.

Example Suppose we want to remove the element at index 2:

Too small, violate
the max-heap property!



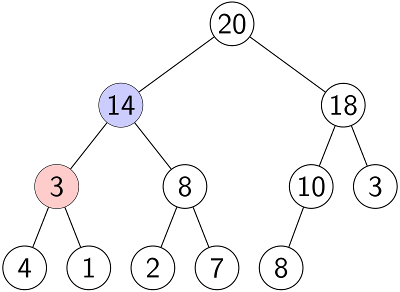
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Deletion at index i

- Replace the element at index i by the last element.
- Idea Fix the violation by swapping until it finds the right position.

Example Suppose we want to remove the element at index 2:



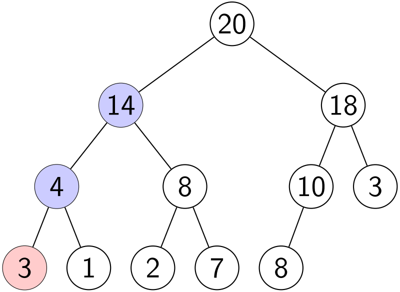
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Deletion at index i

- Replace the element at index i by the last element.
- Idea Fix the violation by swapping until it finds the right position.

Example Suppose we want to remove the element at index 2:



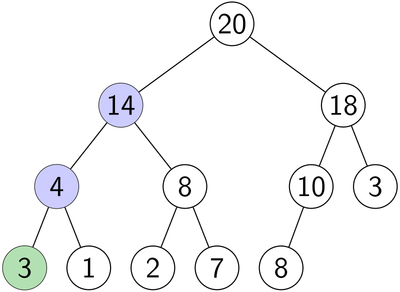
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Deletion at index i

- Replace the element at index i by the last element.
- Idea Fix the violation by swapping until it finds the right position.

Example Suppose we want to remove the element at index 2:



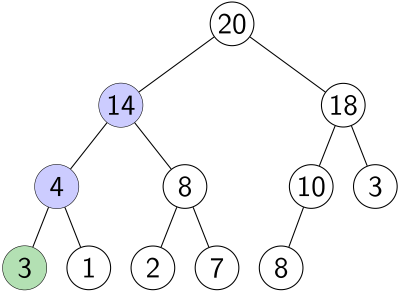
Heap operations

Suppose $H[1..n]$ is a binary max-heap.

Deletion at index i

- Replace the element at index i by the last element.
- Idea Fix the violation by swapping until it finds the right position.

Example Suppose we want to remove the element at index 2:



This process is called HeapifyDown.

Heap operations

Delete($H[1..n]$, i)

$H[i] = H[n]$
$n = n - 1$
`// The new` $H[i]$ `can be either too large or too small`
HeapifyUp(H, i)
HeapifyDown(H, i)

HeapifyDown(H, i)

Exercise

Correctness is analogous to the proof of HeapifyUp and Insertion.

Heap operations

Delete($H[1..n]$, i)

```
H[i] = H[n]
n = n − 1
// The new H[i] can be either too large or too small
HeapifyUp(H, i)
HeapifyDown(H, i)
```

HeapifyDown(H, i)

```
Exercise
```

Correctness is analogous to the proof of HeapifyUp and Insertion.

Time complexity?

HeapifyDown takes $O(h) = O(\log n)$ time, so does Delete.

Heap operations

IncreaseKey(H, i, key)

if $key \leq H[i]$ **then** error('new key is smaller')
$H[i] = key$
HeapifyUp(H, i)

DecreaseKey(H, i, key)

Exercise

Heap operations

IncreaseKey(H, i, key)

- **if** $key \leq H[i]$ **then** error('new key is smaller')
- $H[i] = key$
- HeapifyUp(H, i)

DecreaseKey(H, i, key)

- Exercise

Correctness follows from the correctness of HeapifyUp/Down.

Heap operations

IncreaseKey(H, i, key)

if $key \leq H[i]$ **then** error('new key is smaller')
$H[i] = key$
HeapifyUp(H, i)

DecreaseKey(H, i, key)

Exercise

Correctness follows from the correctness of HeapifyUp/Down.

Time complexity is also $O(\log n)$.

HeapSort

We can use a max-heap to sort an input array as follows:

1. Insert each element of the input into a max-heap H.
2. Write the root $H[1]$ to the end of the input array, call Delete, and repeat.

HeapSort

We can use a max-heap to sort an input array as follows:

1. Insert each element of the input into a max-heap H.
2. Write the root $H[1]$ to the end of the input array, call Delete, and repeat.

Time complexity: $O(n \log n)$, asymptotically optimal!

HeapSort

We can use a max-heap to sort an input array as follows:

1. Insert each element of the input into a max-heap H.
2. Write the root $H[1]$ to the end of the input array, call Delete, and repeat.

Time complexity: $O(n \log n)$, asymptotically optimal!

Space complexity?

HeapSort

We can use a max-heap to sort an input array as follows:

1. Insert each element of the input into a max-heap H.
2. Write the root $H[1]$ to the end of the input array, call Delete, and repeat.

Time complexity: $O(n \log n)$, asymptotically optimal!

Space complexity? $O(n)$ since we used a new array as the heap.

HeapSort

We can use a max-heap to sort an input array as follows:

1. Insert each element of the input into a max-heap H.
2. Write the root $H[1]$ to the end of the input array, call Delete, and repeat.

Time complexity: $O(n \log n)$, asymptotically optimal!

Space complexity? $O(n)$ since we used a new array as the heap.

Can we do better?

HeapSort

We can use a max-heap to sort an input array as follows:

1. Insert each element of the input into a max-heap H.
2. Write the root $H[1]$ to the end of the input array, call Delete, and repeat.

Time complexity: $O(n \log n)$, asymptotically optimal!

Space complexity? $O(n)$ since we used a new array as the heap.

Can we do better?

Note that the second step can be done in place:

`//` $H[1..n]$ `is a max-heap`
$x = H[1]$
Delete(H, 1) `//` $H[n]$ `is now free`
$H[n] = x$

Build a heap in place

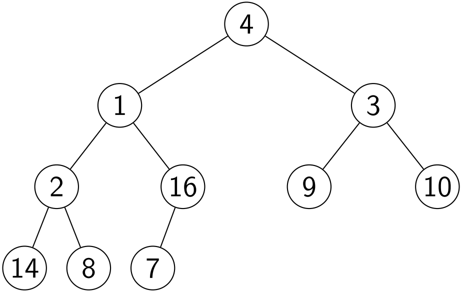
We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**

  HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



Build a heap in place

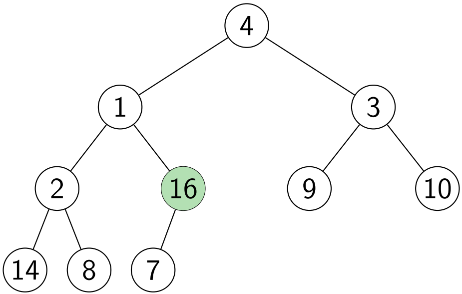
We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**

  HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



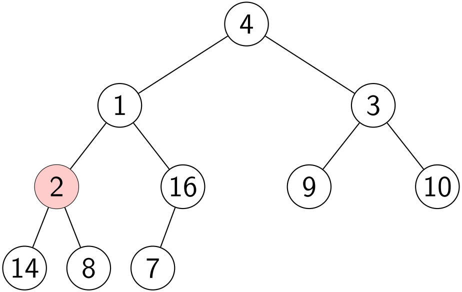
Build a heap in place

We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**
  HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



Build a heap in place

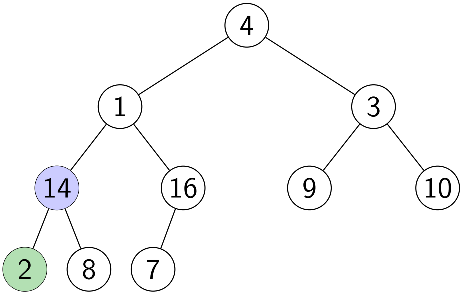
We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**

HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



Build a heap in place

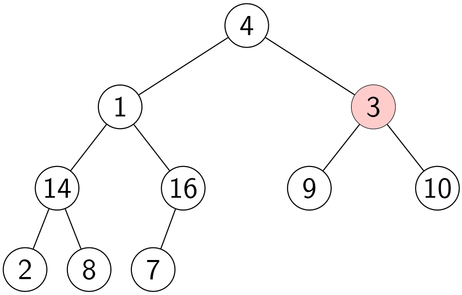
We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**

  HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



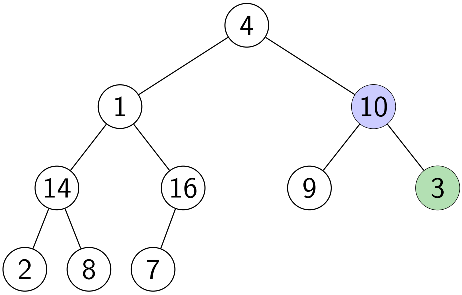
Build a heap in place

We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**
  HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



Build a heap in place

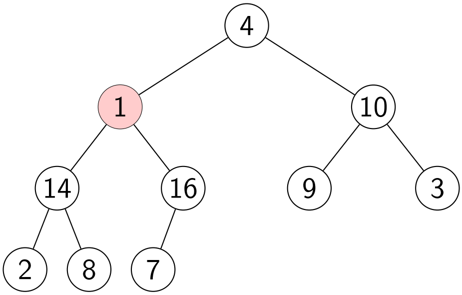
We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**

  HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



Build a heap in place

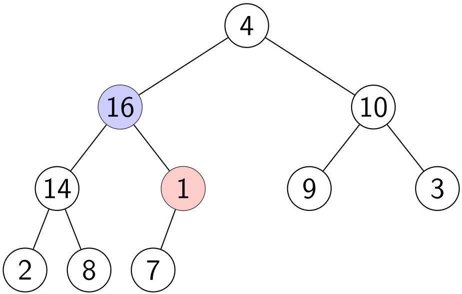
We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**

  HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



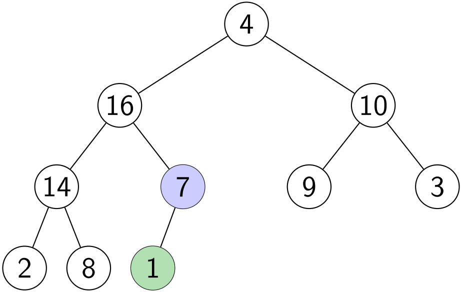
Build a heap in place

We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**
  HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



Build a heap in place

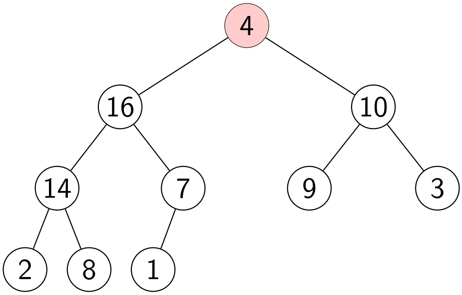
We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**

  HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



Build a heap in place

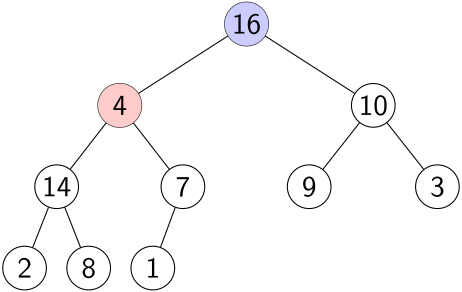
We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**

  HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



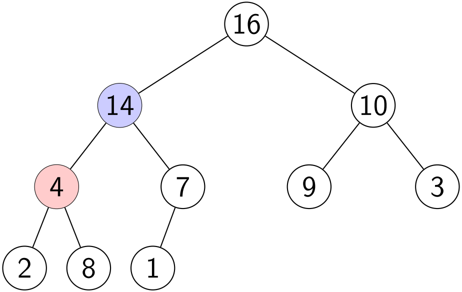
Build a heap in place

We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**
    HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



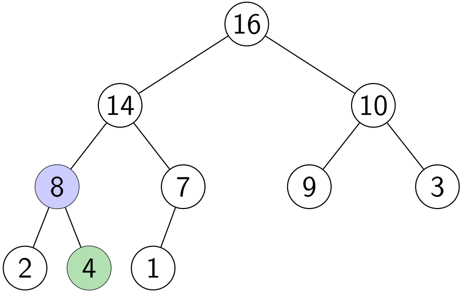
Build a heap in place

We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**
  HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



Build a heap in place

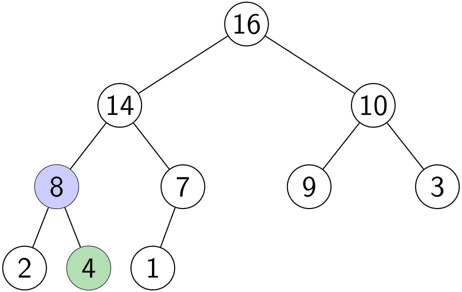
We can convert an array into a max-heap from the bottom up:

BuildHeap($A[1..n]$)

for $i = \lceil n/2 \rceil$ **down to** 1 **do**

HeapifyDown(A, i)

Example: $A[1..10] = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.$



Result: $A[1..10] = 16, 14, 10, 8, 7, 9, 3, 2, 4, 1.$

Build a heap in place

Correctness? Loop invariant: At the start of each for loop, each node at $i+1,\ldots,n$ is the root of a max-heap.

Build a heap in place

Correctness? Loop invariant: At the start of each for loop, each node at $i+1, \ldots, n$ is the root of a max-heap.

Space complexity? $O(1)$,

Build a heap in place

Correctness? Loop invariant: At the start of each for loop, each node at $i+1, \ldots, n$ is the root of a max-heap.

Space complexity? $O(1)$, so HeapSort can be done in place!

Build a heap in place

Correctness? Loop invariant: At the start of each for loop, each node at $i+1, \ldots, n$ is the root of a max-heap.

Space complexity? $O(1)$, so HeapSort can be done in place!

Time complexity?

- $\lceil n/2 \rceil = O(n)$ rounds, $O(\log n)$ each, so $O(n \log n)$.

Build a heap in place

Correctness? Loop invariant: At the start of each for loop, each node at $i+1,\ldots,n$ is the root of a max-heap.

Space complexity? $O(1)$, so HeapSort can be done in place!

Time complexity?

- $\lceil n/2 \rceil = O(n)$ rounds, $O(\log n)$ each, so $O(n \log n)$.
- Is it tight?

Build a heap in place

A more careful analysis:

- At level i, there are at most 2^i nodes, each needs $O(h-i)$ time.

$$
\begin{aligned}
& c\left(2^0 h + 2^1(h-1) + \cdots + 2^{h-2}\cdot 2 + 2^{h-1}\cdot 1\right) \\
=& c2^h \left(\frac{h}{2^h} + \frac{h-1}{2^{h-1}} + \cdots + \frac{2}{2^2} + \frac{1}{2}\right) \\
=& c2^h \sum_{i=1}^{h} \frac{i}{2^i} \le c2^h \sum_{i=1}^{\infty} \frac{i}{2^i} \\
=& c2^h \cdot 2 \le 2cn = \boxed{\text{O(n)}}.
\end{aligned}
$$

The last line used the facts $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$ and $n \ge 2^h$.

Build a heap in place

A more careful analysis:

- At level i, there are at most 2^i nodes, each needs $O(h-i)$ time.
- Total time:

$$
\begin{aligned}
& c\left(2^0 h + 2^1(h-1) + \cdots + 2^{h-2}\cdot 2 + 2^{h-1}\cdot 1\right) \\
=& c2^h\left(\frac{h}{2^h} + \frac{h-1}{2^{h-1}} + \cdots + \frac{2}{2^2} + \frac{1}{2}\right) \\
=& c2^h \sum_{i=1}^{h} \frac{i}{2^i} \le c2^h \sum_{i=1}^{\infty} \frac{i}{2^i} \\
=& c2^h \cdot 2 \le 2cn = \boxed{\mathrm{O(n)}}.
\end{aligned}
$$

The last line used the facts $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$ and $n \ge 2^h$.