Sorting

SFWRENG 2CO3: Data Structures and Algorithms

Jelle Hellings

Department of Computing and Software
McMaster University

Winter 2024

A final sort algorithm: HeapSort

MergeSort Worst-case $\Theta(N \log_2(N))$ runtime complexity.
But also: $\Theta(N)$ memory usage, high constants.

QuickSort Expected $\Theta(N \log_2(N))$ runtime complexity.
But also: $\Theta(\log_2(N))$ memory usage, finicky pivot choices, worst-case $\Theta(N^2)$.

A final sort algorithm: HeapSort

MergeSort Worst-case $\Theta(N \log_2(N))$ runtime complexity.
But also: $\Theta(N)$ memory usage, high constants.

QuickSort Expected $\Theta(N \log_2(N))$ runtime complexity.
But also: $\Theta(\log_2(N))$ memory usage, finicky pivot choices, worst-case $\Theta(N^2)$.

Next: HeapSort
Worst-case $\Theta(N \log_2(N))$ runtime complexity and $\Theta(1)$ memory usage!

HEAPSORT: High-level overview

Algorithm SELECTIONLIKESORT(L):
Input: List $L[0 \ldots N)$ of N values.

```
for pos := N to 2 do
    Find the position p of the
        maximum value in L[0 ... pos).
    Exchange L[pos − 1] and L[p].
```

Comparisons: $\sum_{pos=2}^{N} pos = \Theta\left(N^2\right)$.

HeapSort: High-level overview

Algorithm HeapSort(L):

Input: List $L[0 \dots N)$ of N values.

1: Restructure L so that it is easy to find the *maximum*.
2: **for** $pos := N$ **to** 2 **do**
3: Use structure to find the *maximum* and efficiently remove the *maximum*.
4: Place the *maximum* at $L[pos - 1]$.

HeapSort: High-level overview

Algorithm HeapSort(L):

Input: List $L[0\dots N)$ of N values.

```
Restructure L so that it is easy to find the maximum    ← a binary max-heap.
for pos := N to 2 do
    Use structure to find the maximum and
          efficiently remove the maximum.
    Place the maximum at L[pos − 1].
```

Max-heaps

A max-heap H is a collection of values

Add(H, v) add value v to a max-heap H;

DelMax(H) removes the maximum value $w \in H$ and return w.

Size(H) returns the number of values in H.

HeapSort: High-level overview

Algorithm HeapSort(L):

Input: List $L[0 \dots N)$ of N values.

1: Restructure L so that it is easy to find the *maximum* ← a *binary max-heap*.
2: **for** $pos := N$ **to** 2 **do**
3: Use structure to find the *maximum* and efficiently remove the *maximum*.
4: Place the *maximum* at $L[pos - 1]$.

Max-heaps

A max-heap H is a collection of values

Add(H, v) add value v to a max-heap H; ← in $\Theta(\log_2(|H|))$.

DelMax(H) removes the maximum value $w \in H$ and return w. ← in $\Theta(\log_2(|H|))$.

Size(H) returns the number of values in H.

HeapSort: High-level overview

Algorithm HeapSort(L):

Input: List $L[0 \dots N)$ of N values.

Restructure L so that it is easy to find the *maximum* $\leftarrow$ a *binary max-heap*.
for $pos := N$ **to** 2 **do**
  Use structure to find the *maximum* and
    efficiently remove the *maximum*.
  Place the *maximum* at $L[pos - 1]$.

Max-heaps

A max-heap H is a collection of values

Add(H, v)	add value v to a max-heap H;	$\leftarrow$ in $\Theta(\log_2(\lvert H \rvert))$.
DelMax(H)	removes the maximum value $w \in H$ and return w.	$\leftarrow$ in $\Theta(\log_2(\lvert H \rvert))$.
Size(H)	returns the number of values in H.	

We can store a max-heap of $|H|$ values in an array of $|H|$ values.

HeapSort: High-level overview

Algorithm HeapSort(L):

Input: List $L[0 \dots N)$ of N values.

1: Restructure L so that it is easy to find the *maximum*. $\approx N$ Adds $\rightarrow \Theta(N \log_2(N))$.

2: **for** $pos := N$ **to** 2 **do**

3: Use structure to find the *maximum* and efficiently remove the *maximum*. $\approx N$ DelMaxs $\rightarrow \Theta(N \log_2(N))$.

4: Place the *maximum* at $L[pos - 1]$.

Max-heaps

A max-heap H is a collection of values

Add(H, v) add value v to a max-heap H; $\leftarrow$ in $\Theta(\log_2(|H|))$.

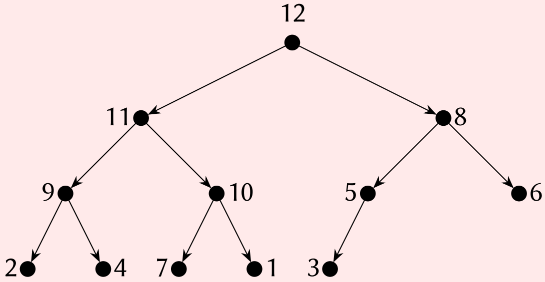
DelMax(H) removes the maximum value $w \in H$ and return w. $\leftarrow$ in $\Theta(\log_2(|H|))$.

Size(H) returns the number of values in H.

We can store a max-heap of $|H|$ values in an array of $|H|$ values.

The binary max-heap data structure

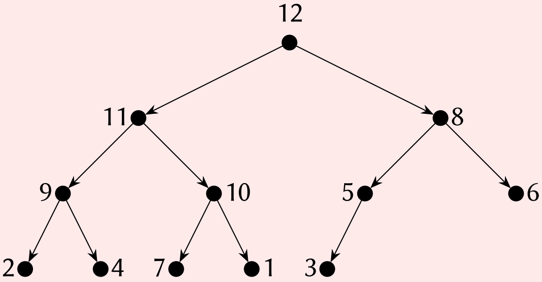
A binary max-heap is a *binary tree* in which each node n has a key $k(n)$ such that:



The binary max-heap data structure

A binary max-heap is a *binary tree* in which each node n has a key $k(n)$ such that:

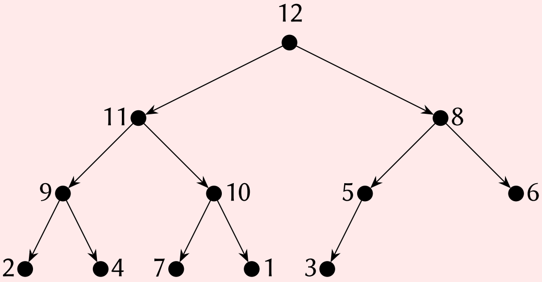
- the tree is *nearly complete*.
  Or: the tree is filled from top-to-bottom, left-to-right.



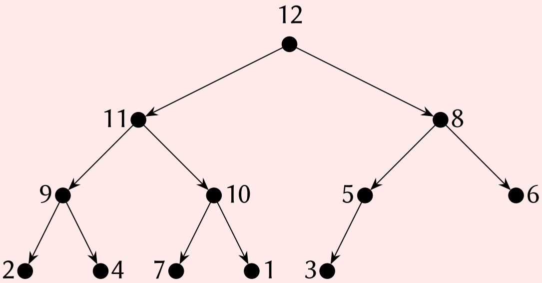
The binary max-heap data structure

A binary max-heap is a *binary tree* in which each node n has a key $k(n)$ such that:

- the tree is *nearly complete*.
  Or: the tree is filled from top-to-bottom, left-to-right.
- the tree satisfies the *heap property*: if node n has child c, then $k(n) \geq k(c)$.
  Or: the key in each node is larger-or-equal to the keys in the children of n.

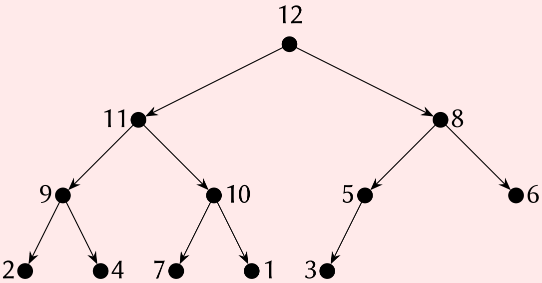


The binary max-heap data structure



The *maximum* is straightforward to find: root of the tree.

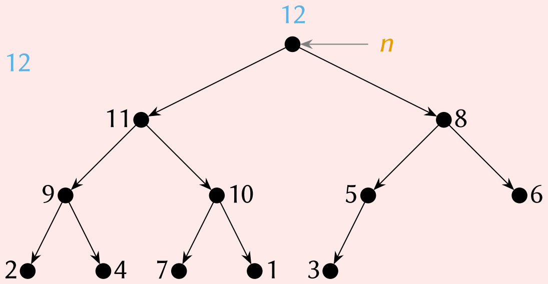
The binary max-heap data structure



Algorithm DelMax(H) (high-level overview):

The binary max-heap data structure

$v = 12$

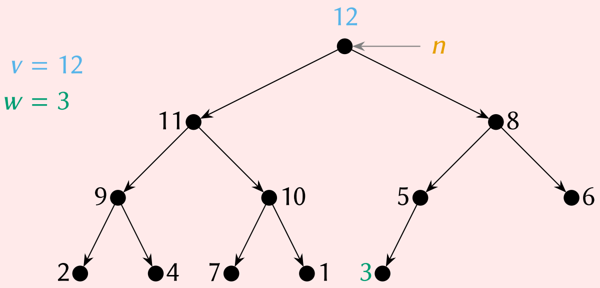


Algorithm DelMax(H) (high-level overview):

Let v be the value at the root n of H.

return v.

The binary max-heap data structure



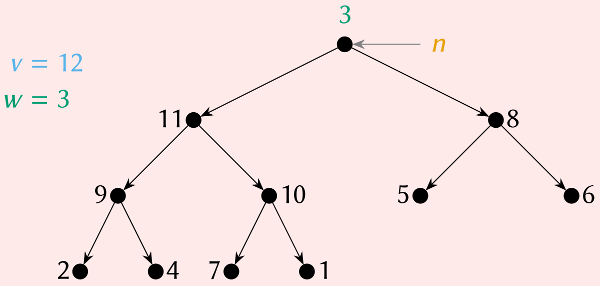
Algorithm DelMax(H) (high-level overview):

1: Let v be the value at the root n of H.

2: Let w be the last value in H.

5: **return** v.

The binary max-heap data structure

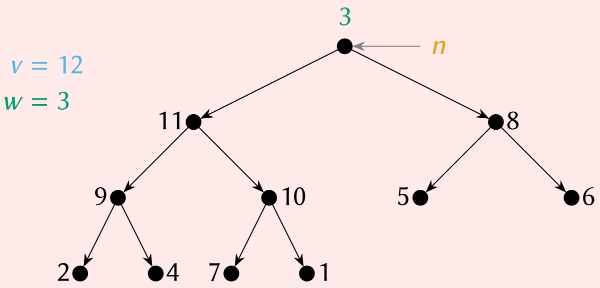


Algorithm DelMax(H) (high-level overview):

Let v be the value at the root n of H.
Let w be the last value in H.
Remove the last node in H and set $k(n) := w$.

return v.

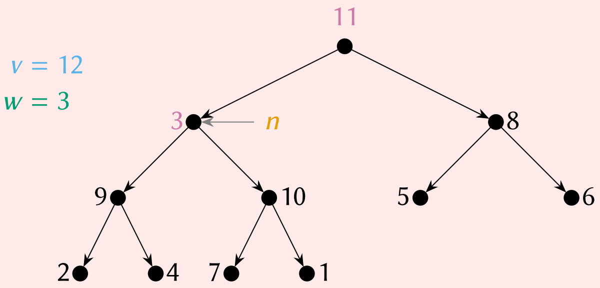
The binary max-heap data structure



Algorithm DelMax(H) (high-level overview):

1: Let v be the value at the root n of H.
2: Let w be the last value in H.
3: Remove the last node in H and set $k(n) := w$.
4: Sink n to a valid position (reestablish the heap property).
5: **return** v.

The binary max-heap data structure



Algorithm DelMax(H) (high-level overview):

1: Let v be the value at the root n of H.

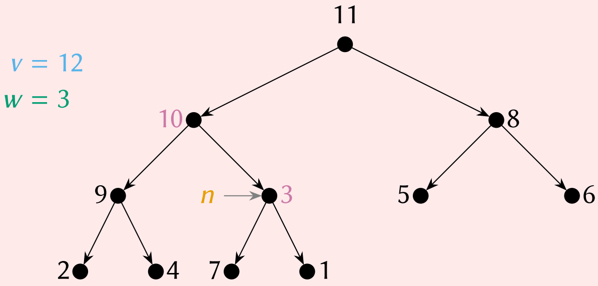
2: Let w be the last value in H.

3: Remove the last node in H and set $k(n) := w$.

4: Sink n to a valid position (reestablish the heap property).

5: **return** v.

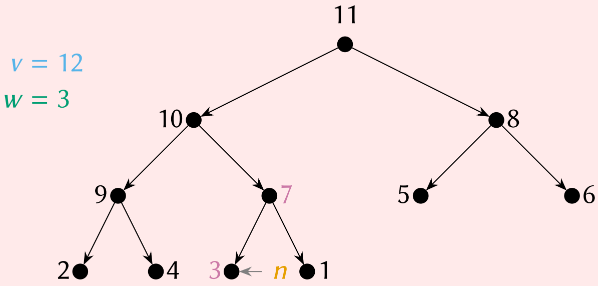
The binary max-heap data structure



Algorithm DelMax(H) (high-level overview):

1: Let v be the value at the root n of H.
2: Let w be the last value in H.
3: Remove the last node in H and set $k(n) := w$.
4: Sink n to a valid position (reestablish the heap property).
5: **return** v.

The binary max-heap data structure



Algorithm DelMax(H) (high-level overview):

1: Let v be the value at the root n of H.

2: Let w be the last value in H.

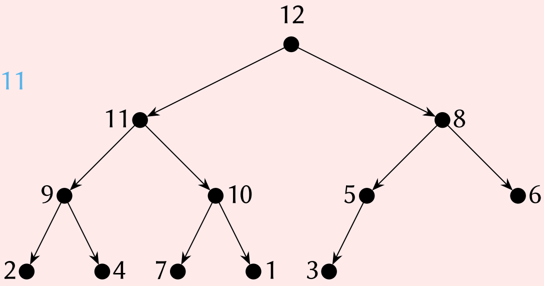
3: Remove the last node in H and set $k(n) := w$.

4: Sink n to a valid position (reestablish the heap property).

5: **return** v.

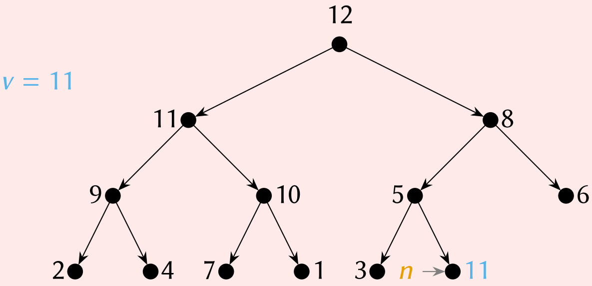
The binary max-heap data structure

$v = 11$



Algorithm $\textsc{Add}(H, v)$ (high-level overview):

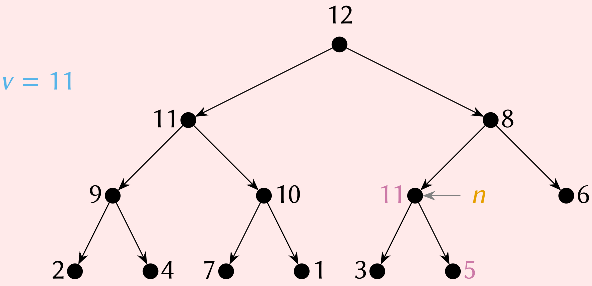
The binary max-heap data structure



Algorithm Add(H, v) (high-level overview):

1: Add a node n to the end of H with $k(n) := v$.

The binary max-heap data structure



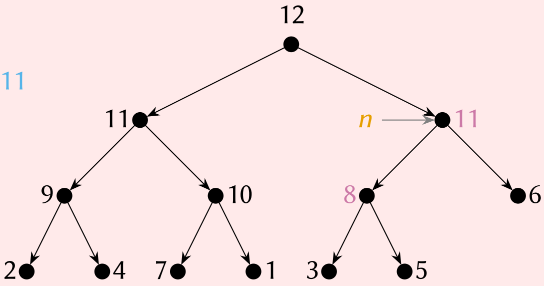
Algorithm Add(H, v) (high-level overview):

1: Add a node n to the end of H with $k(n) := v$.

2: Swim n upward to a valid position (reestablish the heap property).

The binary max-heap data structure

$v = 11$

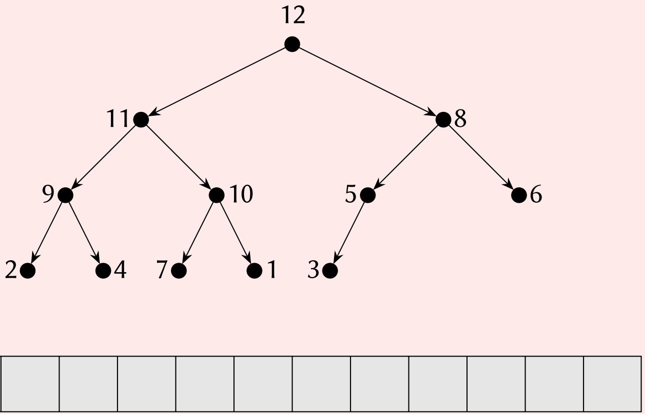


Algorithm ADD(H, v) (high-level overview):

1: Add a node n to the end of H with $k(n) := v$.

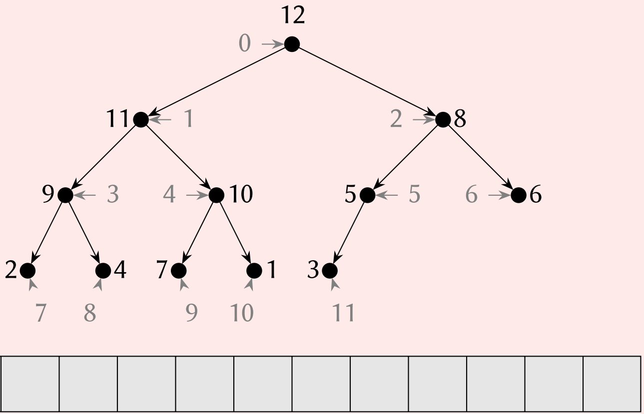
2: Swim n upward to a valid position (reestablish the heap property).

The binary max-heap data structure



Storing a max-heap in an array

The binary max-heap data structure

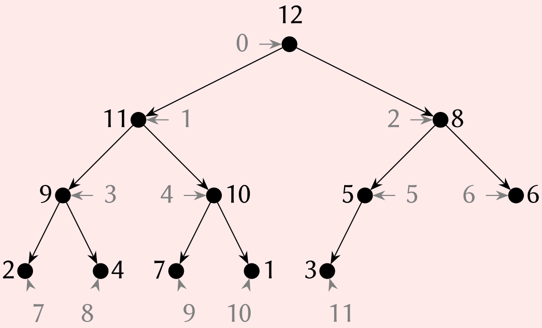


Storing a max-heap in an array

Number the nodes from top-to-bottom, left-to-right.

Warning: The book numbers values in max-heaps starting at 1 instead of 0!

The binary max-heap data structure

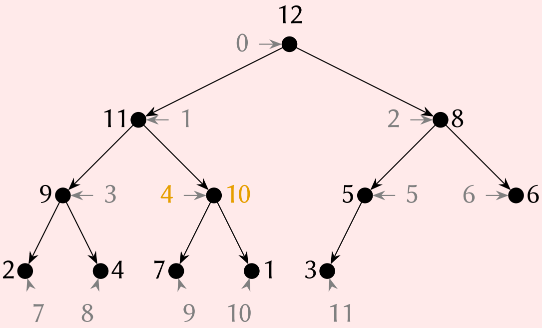


12	11	8	9	10	5	6	2	4	7	1	3

Storing a max-heap in an array

Number the nodes from top-to-bottom, left-to-right → positions in the array.

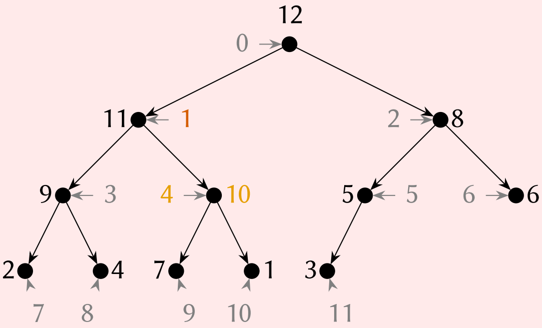
The binary max-heap data structure



12	11	8	9	10	5	6	2	4	7	1	3

If a node is at position p,

The binary max-heap data structure

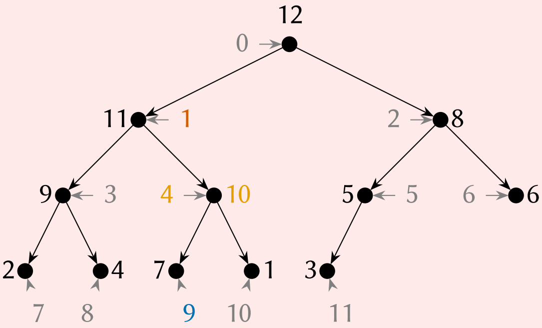


12	11	8	9	10	5	6	2	4	7	1	3

If a node is at position p,

- then the parent is at position $\text{parent}(p) = (p - 1) \operatorname{div} 2$.

The binary max-heap data structure

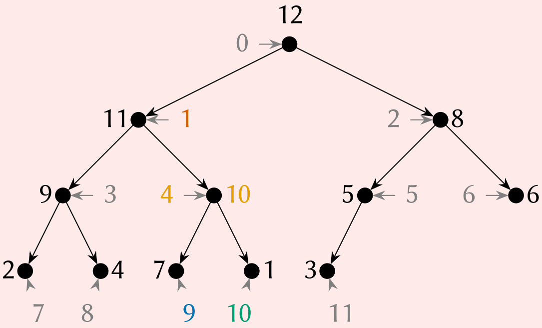


12	11	8	9	10	5	6	2	4	7	1	3

If a node is at position p,

- then the parent is at position $\mathrm{parent}(p) = (p - 1) \operatorname{div} 2$.
- then the left child is at position $\mathrm{lchild}(p) = 2 \cdot p + 1$.

The binary max-heap data structure



12	11	8	9	10	5	6	2	4	7	1	3

If a node is at position p,

- then the parent is at position $\text{parent}(p) = (p - 1) \operatorname{div} 2$.
- then the left child is at position $\text{lchild}(p) = 2 \cdot p + 1$.
- then the right child is at position $\text{rchild}(p) = 2 \cdot p + 2$.

The binary max-heap data structure

If a node is at position p,

- then the parent is at position $\mathrm{parent}(p) = (p - 1) \operatorname{div} 2$.
- then the left child is at position $\mathrm{lchild}(p) = 2 \cdot p + 1$.
- then the right child is at position $\mathrm{rchild}(p) = 2 \cdot p + 2$.

Algorithm $\textsc{Sink}(L[0 \dots N], p)$**:**

Sink $L[p]$ to a valid position.

Algorithm $\textsc{Swim}(L[0 \dots N], p)$**:**

Swim $L[p]$ upward to a valid position.

The binary max-heap data structure

If a node is at position p,

- then the parent is at position $\text{parent}(p) = (p - 1) \text{ div } 2$.
- then the left child is at position $\text{lchild}(p) = 2 \cdot p + 1$.
- then the right child is at position $\text{rchild}(p) = 2 \cdot p + 2$.

Algorithm $\textsc{Sink}(L[0 \dots N), p)$:

while true **do**
  $np := p$.
  if $\text{lchild}(p) < N$ **and** $L[np] < L[\text{lchild}(p)]$ **then**
    $np := \text{lchild}(p)$.
  if $\text{rchild}(p) < N$ **and** $L[np] < L[\text{rchild}(p)]$ **then**
    $np := \text{rchild}(p)$.
  if $np = p$ **then return**.
  Exchange $L[p]$ and $L[np]$.
  $p := np$.

Algorithm $\textsc{Swim}(L[0 \dots N), p)$:

while $p \neq 0$ **and** $L[p] > L[\text{parent}(p)]$ **do**
  Exchange $L[p]$ and $L[\text{parent}(p)]$.
  $p := \text{parent}(p)$.

HeapSort: Filling in the details

Algorithm HeapSort(L):

Input: List $L[0 \dots N)$ of N values.

1: Turn L into a max-heap.

2: **for** $pos := N$ **to** 2 **do**

3: $\quad max := \text{DelMax}(L[0 \dots pos))$

4: $\quad L[pos - 1] := max$.

HeapSort: Filling in the details

Algorithm HeapSort(L):

Input: List $L[0 \ldots N)$ of N values.

Turn L into a max-heap.
for $pos := N$ **to** 2 **do**
    $max := \textsc{DelMax}(L[0 \ldots pos))$
    $L[pos - 1] := max$.

Algorithm MakeHeap($L[0 \ldots N)$):

$len := 1$.
/* inv: $L[0 \ldots len$ is a max-heap, bf: $N - len$ */
while $len \neq N$ **do**
    $\textsc{Add}(L[0 \ldots len), L[len])$.
    $len := len + 1$.

HeapSort: Filling in the details

Algorithm HeapSort(L):

Input: List $L[0 \dots N)$ of N values.

```
Turn L into a max-heap.
for pos := N to 2 do
    max := DelMax(L[0 ... pos))
    L[pos − 1] := max.
```

Algorithm MakeHeap($L[0 \dots N)$):

```
len := 1.
/* inv: L[0 ... len is a max-heap, bf: N − len */
while len ≠ N do
    Add(L[0 ... len), L[len]).
    len := len + 1.
```

$N - 1$ Swim operations.

HeapSort: Filling in the details

A *faster* MakeHeap

L :

?	?	?	?	5	?	?	?	?	10	13	?

Consider a position p in L such that

- the left child at position $\text{lchild}(p)$ already forms a valid max-heap in L; and
- the right child at position $\text{rchild}(p)$ already forms a valid max-heap in L.

HeapSort: Filling in the details

A *faster* MakeHeap

L :	?	?	?	?	5	?	?	?	?	10	13	?
					$\uparrow$					$\uparrow$	$\uparrow$	
					p					lc	rc	

Consider a position p in L such that

- the left child at position $\text{lchild}(p)$ already forms a valid max-heap in L; and
- the right child at position $\text{rchild}(p)$ already forms a valid max-heap in L.

HeapSort: Filling in the details

A *faster* MakeHeap

L :	?	?	?	?	5	?	?	?	?	10	13	?
					p					lc	rc	

Consider a position p in L such that

- the left child at position lchild(p) already forms a valid max-heap in L; and
- the right child at position rchild(p) already forms a valid max-heap in L.

Sink(L[0…N], p) assures that p also forms a valid max-heap in L.

HeapSort: Filling in the details

A *faster* MakeHeap

L :	?	?	?	?	13	?	?	?	?	10	5	?
					↑					↑	↑	
					p					lc	rc	

Consider a position p in L such that

- the left child at position lchild(p) already forms a valid max-heap in L; and
- the right child at position rchild(p) already forms a valid max-heap in L.

Sink(L[0...N], p) assures that p also forms a valid max-heap in L.

HeapSort: Filling in the details

A *faster* MakeHeap

L:	?	?	?	?	13	?	?	?	?	10	5	?
					$\uparrow$ p					$\uparrow$ lc	$\uparrow$ rc	

Consider a position p in L such that

- the left child at position lchild(p) already forms a valid max-heap in L; and
- the right child at position rchild(p) already forms a valid max-heap in L.

Sink(L[0…N], p) assures that p also forms a valid max-heap in L.

Leaves *always* form valid max-heaps.

HeapSort: Filling in the details

A *faster* MakeHeap

L :	?	?	?	?	13	?	?	?	?	10	5	?
					p					lc	rc	

Consider a position p in L such that

- the left child at position $\text{lchild}(p)$ already forms a valid max-heap in L; and
- the right child at position $\text{rchild}(p)$ already forms a valid max-heap in L.

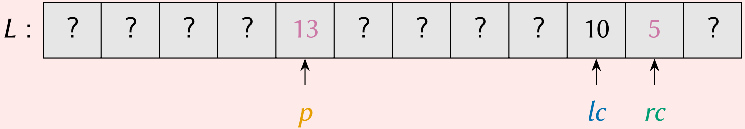
Sink(L[0…N], p) assures that p also forms a valid max-heap in L.

Leaves *always* form valid max-heaps.

Positions $m \leq N$ with $\text{lchild}(m) = m \cdot 2 + 1 \geq N$ represent the leaves in a max-heap of L.

HeapSort: Filling in the details

A *faster* MakeHeap



Consider a position p in L such that

- the left child at position $\mathrm{lchild}(p)$ already forms a valid max-heap in L; and
- the right child at position $\mathrm{rchild}(p)$ already forms a valid max-heap in L.

Sink(L[0…N], p) assures that p also forms a valid max-heap in L.

Leaves *always* form valid max-heaps.

Positions $m \leq N$ with $\mathrm{lchild}(m) = m \cdot 2 + 1 \geq N$ represent the leaves in a max-heap of L

$\longrightarrow$ The last non-child in a max-heap of L values is at position $\left\lfloor \frac{N-1}{2} \right\rfloor$.

HEAPSORT: Filling in the details

A *faster* MAKEHEAP

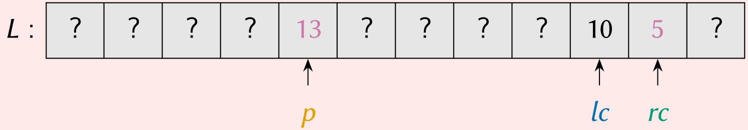
L :												
	?	?	?	?	13	?	?	?	?	10	5	?
					↑ p					↑ lc	↑ rc	

Algorithm FASTMAKEHEAP(L):

$k := N - (N \text{ div } 2)$.
while $k \neq 0$ **do**
  SINK($L[0 \dots N)$, k).
  $k := k - 1$.

HeapSort: Filling in the details

A *faster* MakeHeap



Algorithm FastMakeHeap(L):

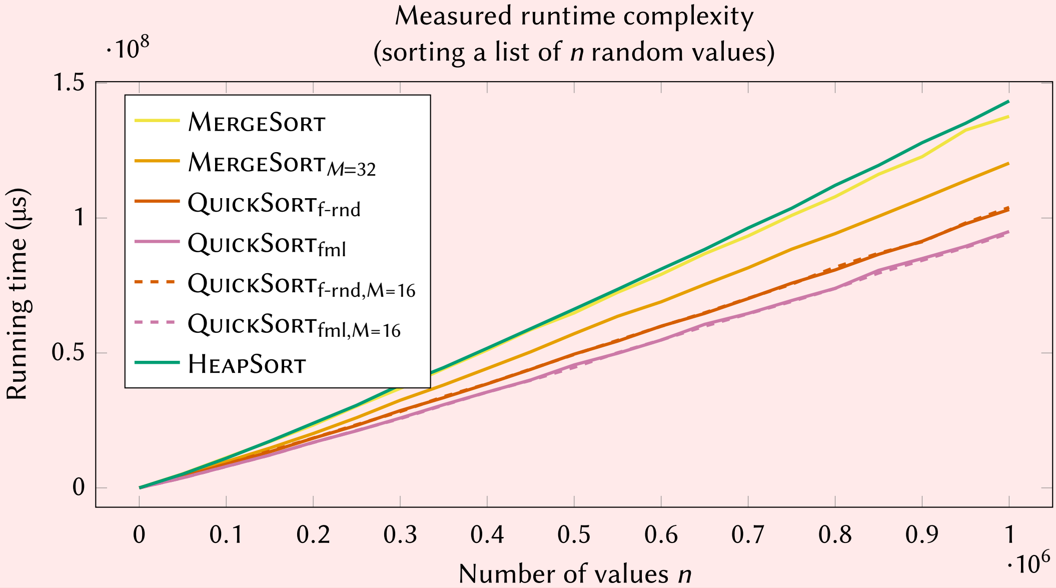
$k := N - (N \operatorname{div} 2)$.
while $k \neq 0$ **do**
  Sink($L[0 \ldots N), k$).
  $k := k - 1$.

$\left\lfloor \frac{N-1}{2} \right\rfloor$ Sink operations.

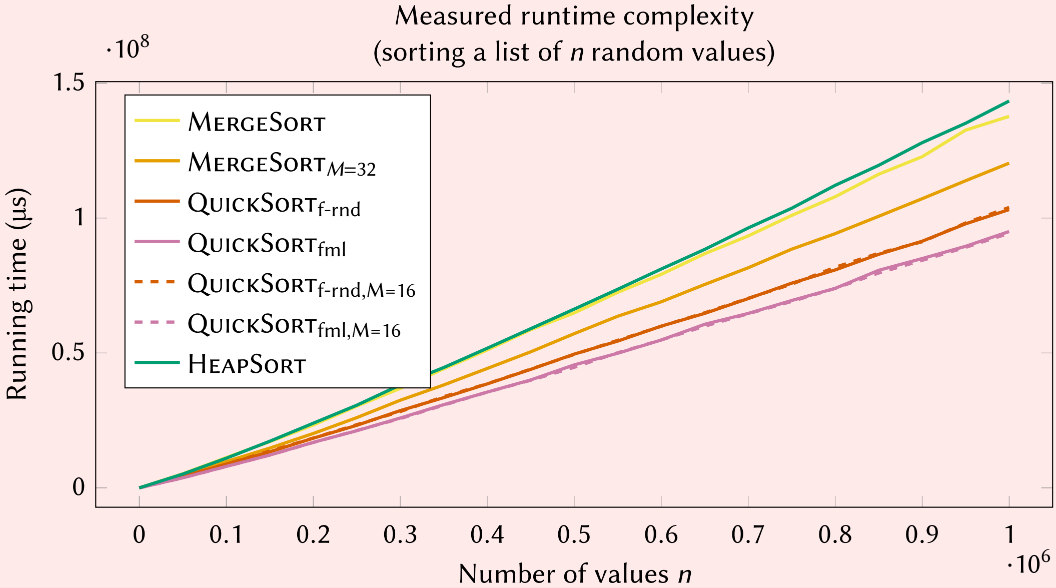
Comparing HeapSort with MergeSort and QuickSort

	Comparisons	Changes	Memory	Stable
MergeSort	$\Theta\left(N \log_2(N)\right)$	$N \log_2(N)$	$\Theta\left(N\right)$	yes
QuickSort	$\Theta\left(N \log_2(N)\right)$ (expected)	$\Theta\left(N \log_2(N)\right)$ (expected)	$\Theta\left(\log_2(N)\right)$ (expected)	no
HeapSort	$\Theta\left(N \log_2(N)\right)$	$N \log_2(N)$	1	no

Comparing HeapSort with MergeSort and QuickSort



Comparing HeapSort with MergeSort and QuickSort



Final notes on HeapSort

A *max-heap* is often referred to as a *Priority Queue*.

There are also *min-heaps* that provide fast access to *minimum values*.

Final notes on HeapSort

A *max-heap* is often referred to as a *Priority Queue*.

There are also *min-heaps* that provide fast access to *minimum values*.

	C++	Java
Priority Queues	`std::priority_queue`	`java.util.PriorityQueue`
Add DelMax	`std::push_heap` `std::pop_heap`	

(related)	`std::make_heap`
	`std::is_heap`
	`std::is_heap_until`
	`std::sort_heap`	

IntroSort: Putting all sorts together

Algorithm $\textsc{IntroSort}(L[start \ldots end), potential)$**:**

potential is the number of values we *could have* sorted with perfect pivot choices.

```
if end − start ≤ M then
    Sort L[start ... end] using InsertionSort.
else if potential < 1.5 · |L| then
    Sort L[start ... end] using HeapSort.
else
    Choose the position p ∈ [start, end) of the pivot value v := L[pos].
    pos := Partition(L, start, end, p).
    IntroSort(L[start ... pos), 2 · potential).
    IntroSort(L[pos + 1 ... end), 2 · potential).
```

Algorithm $\textsc{IntroSort}(L[0 \ldots N))$**:**

```
IntroSort(L[0 ... N), 1).
```

IntroSort: Putting all sorts together