CSC 225 - Summer 2019

Priority Queues II

Bill Bird

Department of Computer Science
University of Victoria

June 17, 2019

Running Time of Heap Operations

Both INSERT and REMOVEMIN require $O(h)$ time, where h is the height of the tree. This can be justified by observing that bubble-up traces a single path from the bottom to the top of the tree, and bubble-down traces a single path from the top to bottom. In the worst case, the height of a binary tree on n nodes is $O(n)$.

We will prove that for a heap, $h \in O(\log_2 n)$. This gives a $O(\log n)$ running time for INSERT and REMOVEMIN.

The Height of a Heap (1)

Theorem: In a full binary tree of height h, the number of nodes at each level $k \leq h$ is

$$2^k$$

Proof Outline:
At level 0, there is $1 = 2^0$ node. Each successive level will have exactly twice as many nodes as the level before it (since each node in the previous level will have exactly two children in a full binary tree).
Note that heaps are **not** full binary trees. However, the first $h - 1$ levels of a heap are full (the last level may not be full).

The Height of a Heap (2)

Theorem: In a heap with n nodes and height h,

$$n \geq 2^h$$

Proof:
Using the theorem on the previous slide, there must be 2^i nodes at levels $0, 1, \ldots, h-1$, since the first $h-1$ levels of a heap are all full. Additionally, there must be at least one node on level h. The total number of nodes must be at least the sum of all nodes above level h plus the first node on level h, giving

$$\begin{aligned} n &\geq 1 + \sum_{i=0}^{h-1} 2^i \\ &= 1 + \left[2^h - 1\right] \\ &= 2^h \end{aligned}$$

The Height of a Heap (3)

Corollary: In a heap with n nodes and height h,

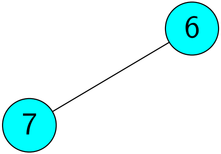
$$h \leq \log_2 n$$

Building Heaps (1)

- Task: Build a heap containing elements of the sequence

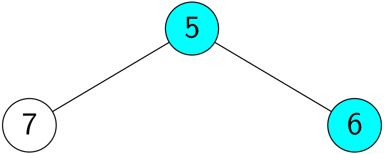
$$S = 7, 6, 5, 4, 3, 2, 1$$

Building Heaps (2)

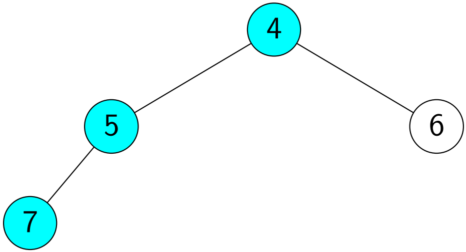


- The simplest algorithm for this task starts with an empty heap and uses n INSERT operations to add the elements.

Building Heaps (3)

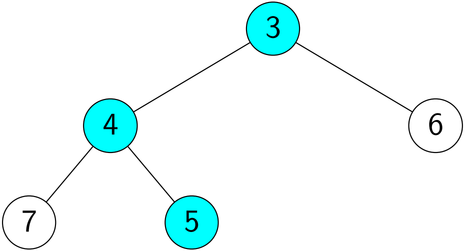


Building Heaps (4)



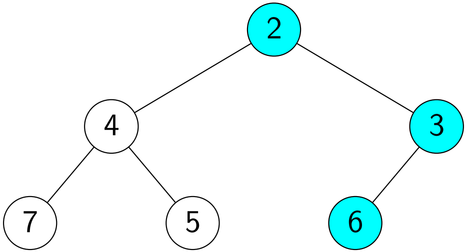
- After $n/2$ insertions, the heap will have height $\log_2 n$.
- Therefore, the last $n/2$ insertions could require $\Theta(\log_2 n)$ time.

Building Heaps (5)

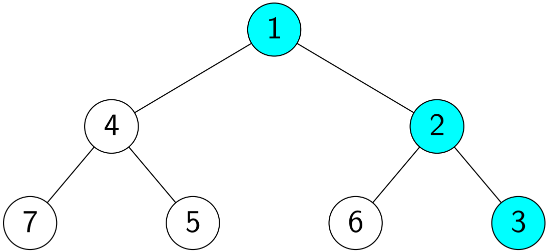


- Building a heap in this fashion therefore requires $\Theta(n \log n)$ time in the worst case.

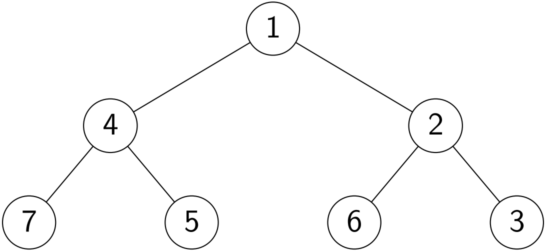
Building Heaps (6)



Building Heaps (7)



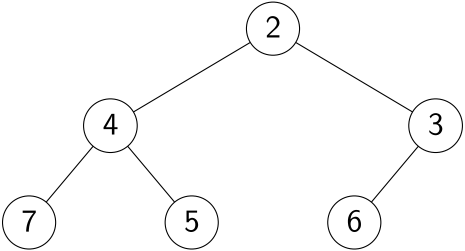
Sorting With Heaps (1)



A:

- A sequence of REMOVEMIN operations can be used to extract the elements of a heap in sorted order.

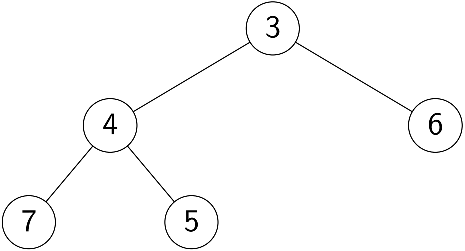
Sorting With Heaps (2)



A: 1

- As each element is removed, it is appended to the sorted array *A*.

Sorting With Heaps (3)

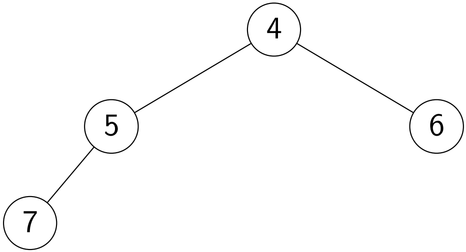


A: 1 2

- The height of the heap during the first $n/2$ RemoveMin calls will be at least

$$\log_2(n/2) = \log_2(n) - 1$$

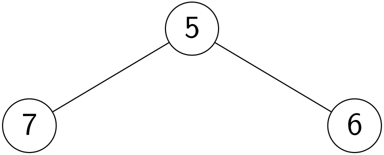
Sorting With Heaps (4)



A: 1 2 3

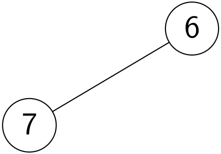
- As a result, the sequence of removals will require $\Theta(n \log n)$ time in the worst case.

Sorting With Heaps (5)



A: 1 2 3 4

Sorting With Heaps (6)



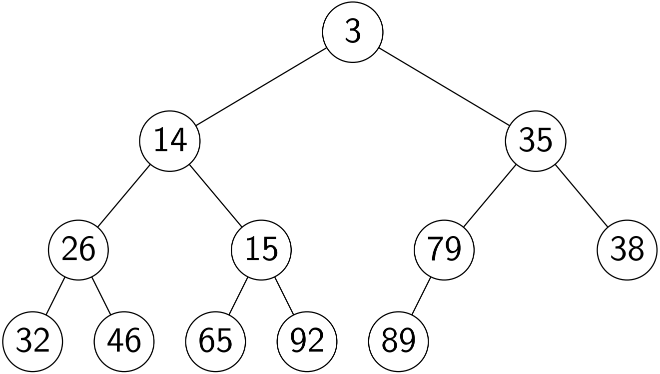
A: 1 2 3 4 5

A: 1 2 3 4 5 6

Sorting With Heaps (8)

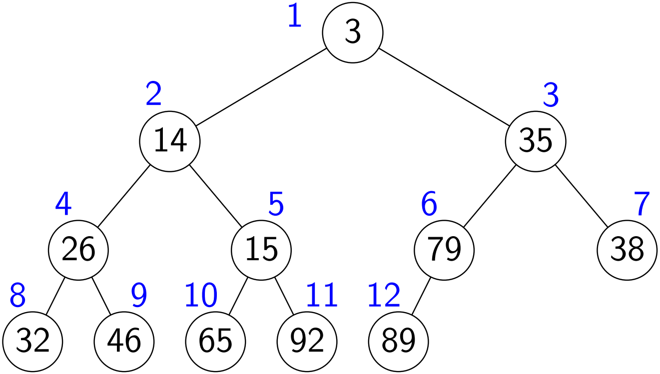
A: 1 2 3 4 5 6 7

Array Based Heaps (1)



- Heaps can be represented with a linked data structure (using objects and pointers).

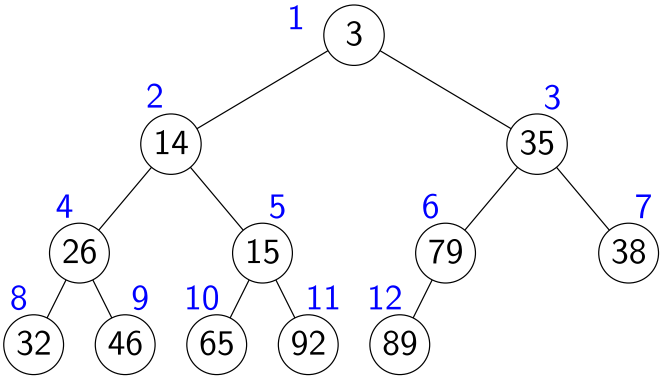
Array Based Heaps (2)



Index	1	2	3	4	5	6	7	8	9	10	11	12
Value	3	14	35	26	15	79	38	32	46	65	92	89

- Heaps can also be represented by an array with a convenient numbering scheme.

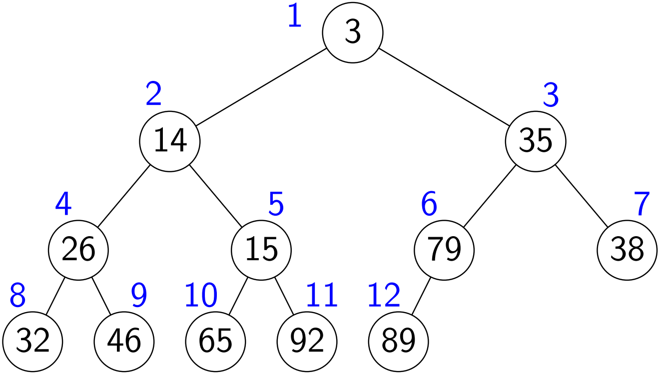
Array Based Heaps (3)



Index	1	2	3	4	5	6	7	8	9	10	11	12
Value	3	14	35	26	15	79	38	32	46	65	92	89

- For this purpose, array indexing is 1-based.

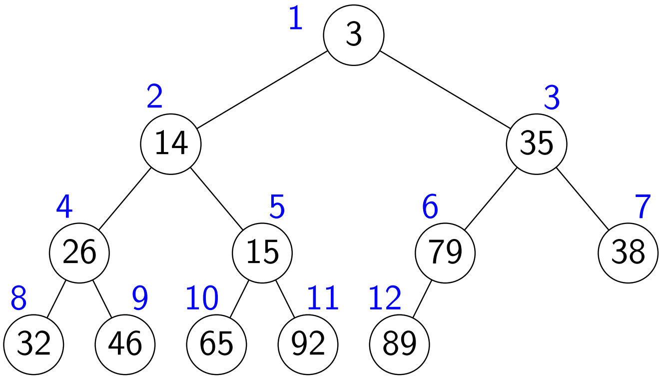
Array Based Heaps (4)



Index	1	2	3	4	5	6	7	8	9	10	11	12
Value	3	14	35	26	15	79	38	32	46	65	92	89

- Nodes are numbered level by level, from left to right, starting at the root.

Array Based Heaps (5)



Index	1	2	3	4	5	6	7	8	9	10	11	12
Value	3	14	35	26	15	79	38	32	46	65	92	89

- This indexing scheme allows convenient traversal of the tree.

$$\textsc{Parent}(i) = i/2 \text{ (integer division)}$$

$$\textsc{LeftChild}(i) = 2i$$

$$\textsc{RightChild}(i) = 2i + 1$$

Heapify (1)

26	92	43	89	14	65	79	35	32	38	46	3	15

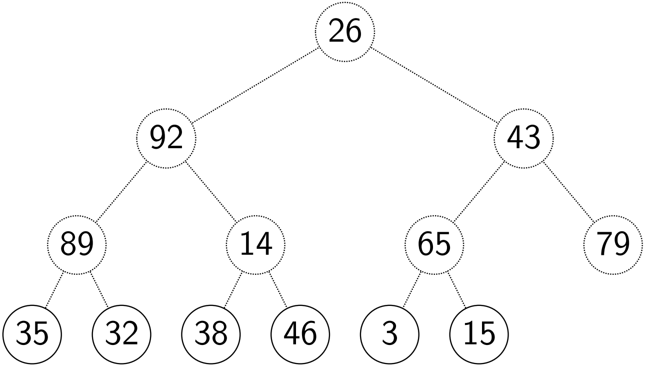
- **Problem**: Create a heap containing the values in the array above.

Heapify (2)

26	92	43	89	14	65	79	35	32	38	46	3	15

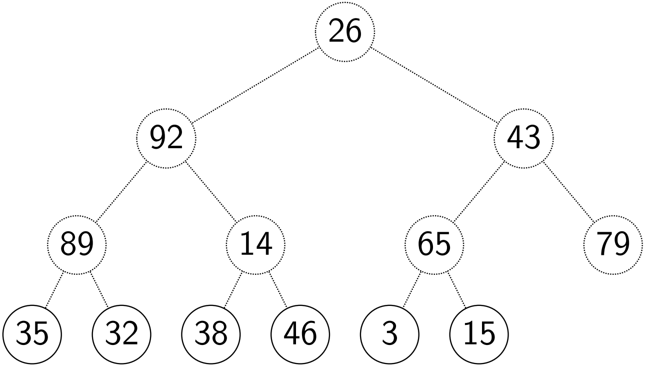
- The most obvious method is to create an empty heap, then add elements one-by-one.
- This requires $\Theta(n \log n)$ operations, because the last $n/2$ insertions will be into a tree of height at least $\log_2 n - 1$.
- Inserting elements one-by-one can be viewed as a 'top-down' approach to building a heap from a collection of elements.

Heapify (3)



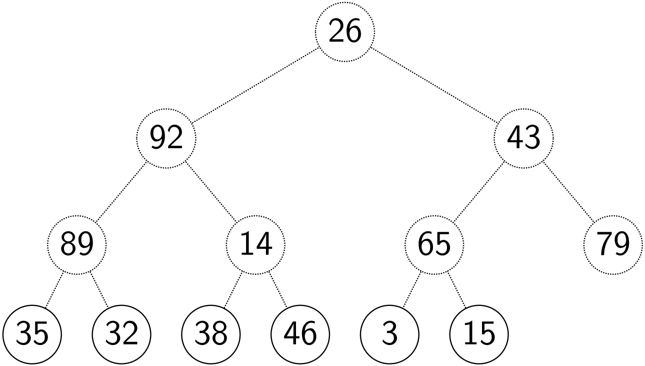
- It is also possible to build a heap with a ‘bottom-up’ approach.
- The bottom-up approach uses an algorithm called HEAPIFY.

Heapify (4)



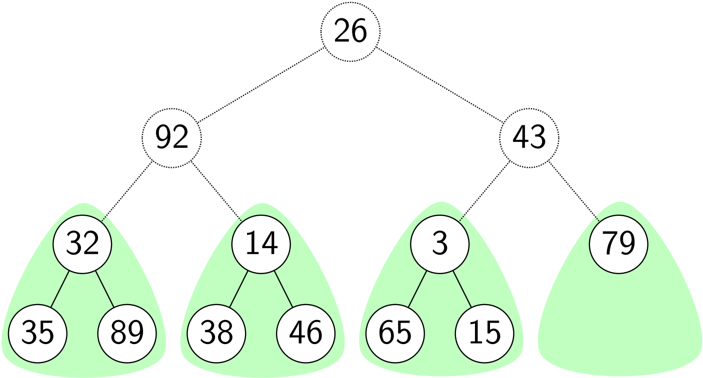
- A complete binary tree containing the input sequence is created and gradually converted into a heap from the bottom to the top.

Heapify (5)



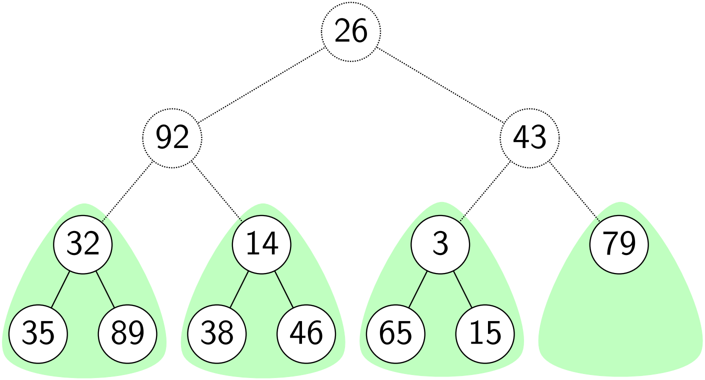
- First, all single nodes are converted to heaps.
- This is trivial since any single node is a heap already.

Heapify (6)



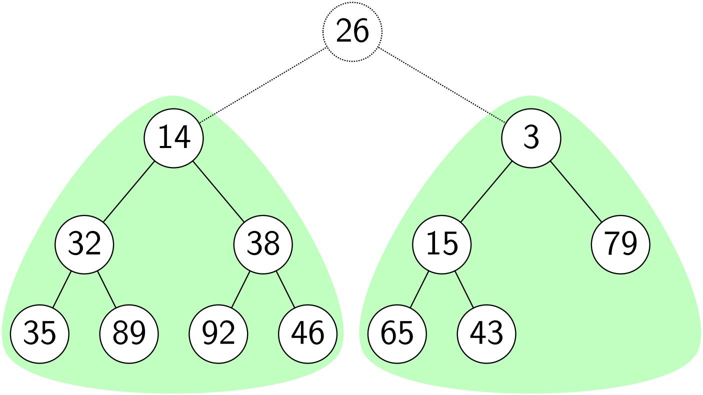
- Next, convert the subtrees of height 1 to heaps.

Heapify (7)



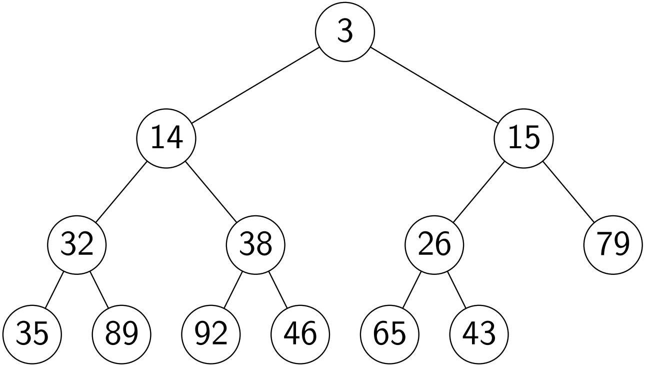
- Observe that the subtrees below the node containing 14 are both heaps.
- Converting the subtree of 14 to a heap requires a single bubble-down operation.

Heapify (8)



- The bottom-up construction merges subtrees into larger heaps repeatedly, working upwards from the bottom of the tree.

Heapify (9)



- Eventually, the entire tree is a heap.

Heapify (10)

```
procedure HeapifyRecursive(v)
    if v = null then
        return
    end if
    HeapifyRecursive(v.left)
    HeapifyRecursive(v.right)
    BubbleDown(v)
end procedure
```

- The HeapifyRecursive algorithm above is suitable for converting any complete binary tree to a heap.
- When an array-based heap is used, an iterative version of Heapify can also be used.

Heapify (11)

```
procedure HeapifyIterative(A, n)
    //Indexing is 1-based, so the last element is A[n]
    for i ← n, n − 1, n − 2, . . . , 2, 1 do
        //Bubble-down the element at index i
        BubbleDown(i)
    end for
end procedure
```

- Working backwards from index n ensures that when index i is processed, all of its subtrees are valid heaps (so the only step remaining is to bubble-down element $A[i]$).

Heapify (12)

```
procedure HeapifyIterative(A, n)
    //Indexing is 1-based, so the last element is A[n]
    for i ← n, n − 1, n − 2, . . . , 2, 1 do
        //Bubble-down the element at index i
        BubbleDown(i)
    end for
end procedure
```

- Both the iterative and recursive versions of Heapify perform n bubble-down operations.
- However, since most of the operations are performed on small subtrees, the total running time of Heapify is better than the top-down construction.

Claim: HEAPIFY constructs a heap of n elements with $\Theta(n)$ operations.

Proof: A cursory analysis seems to suggest that the n bubble-down operations require $\Theta(n \log n)$ operations. To prove that only $\Theta(n)$ operations are necessary, we will consider the operations at each level of the tree separately.

The total height of the tree is $h = \lfloor \log_2 n \rfloor$. A subtree rooted at depth i has height at most $h - i$. Therefore, bubble-down at level i requires $O(h - i)$ operations, which is at most $c(h - i)$ for some constant $c > 0$. The total number of elements at each level i is 2^i.

Heapify (14)

The total height of the tree is $h = \lfloor \log_2 n \rfloor$. A subtree rooted at depth i has height at most $h - i$. Therefore, bubble-down at level i requires $O(h - i)$ operations, which is at most $c(h - i)$ for some constant $c > 0$. The total number of elements at each level i is 2^i.

The total number of elements at each level i is 2^i. The total number of operations for all bubble-down calls is then

$$\sum_{i=0}^{h} c(h-i)2^i = c[2^0 \cdot (h-0) + 2^1 \cdot (h-1) + \ldots + 2^{h-1} \cdot 1 + 2^h \cdot (h-h)]$$

which can be rewritten as

$$c \sum_{i=0}^{h} i 2^{h-i} = c 2^h \sum_{i=0}^{h} \frac{i}{2^i}$$

Heapify (15)

Using the identity

$$\sum_{i=0}^{k} \frac{i}{2^i} = 2 - \frac{k+2}{2^k}$$

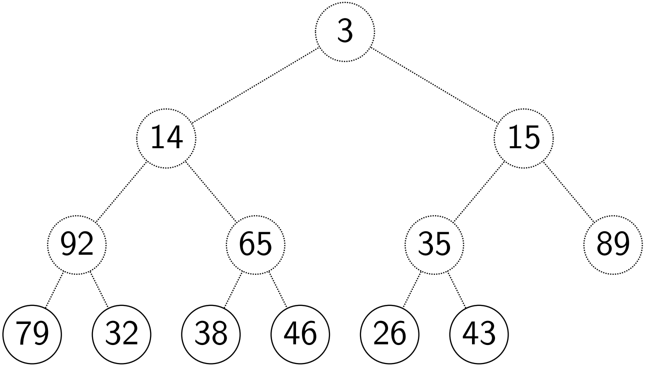
(which we can prove by induction), the running time of HEAPIFY becomes

$$c2^h \sum_{i=0}^{h} \frac{i}{2^i} \leq c2^h \cdot 2$$

and since $h \leq \log_2 n$, the running time is

$$c2^h \cdot 2 \leq c2 \cdot 2^{\log_2 n} = 2cn \in \Theta(n)$$

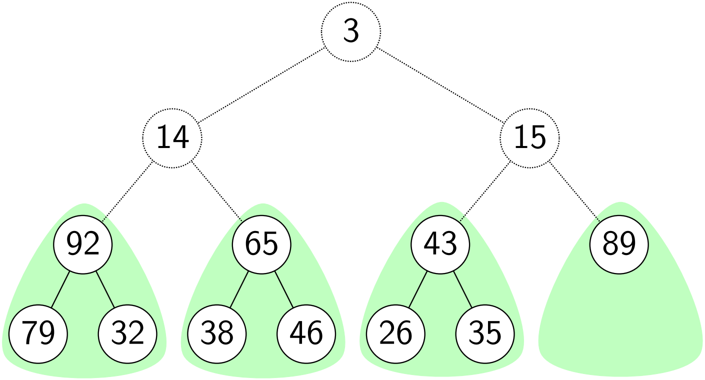
In-place Heap Sort (1)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	3	14	15	92	65	35	89	**79**	**32**	**38**	**46**	**26**	**43**

- Heap Sort can be implemented as an in-place algorithm using array-based heaps, by converting the input array to a max-heap using Heapify,

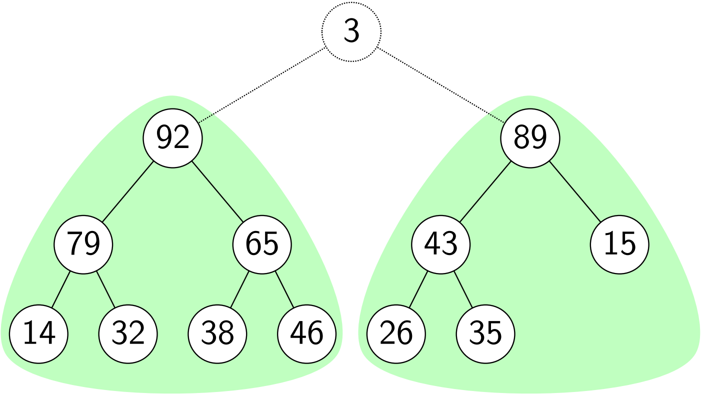
In-place Heap Sort (2)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	3	14	15	**92**	**65**	**43**	**89**	**79**	**32**	**38**	**46**	**26**	**35**

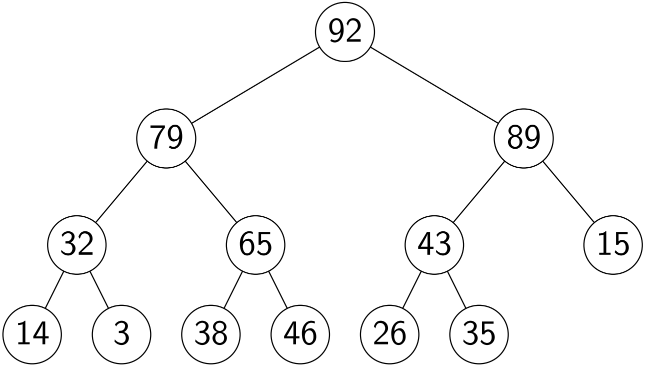
- Bolded values correspond to elements of valid heaps.

In-place Heap Sort (3)



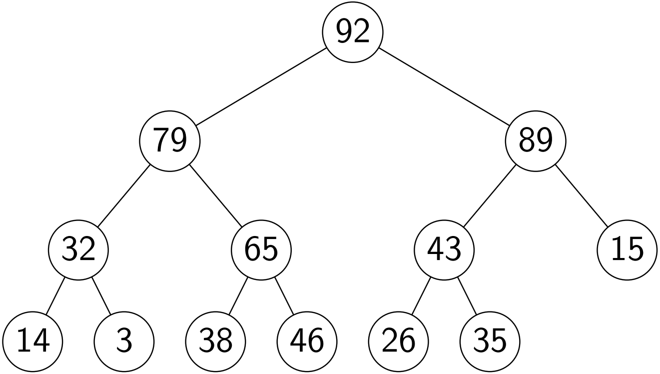
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	3	**92**	**89**	**79**	**65**	**43**	**15**	**14**	**32**	**38**	**46**	**26**	**35**

In-place Heap Sort (4)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**79**	**89**	**32**	**65**	**43**	**15**	**14**	**3**	**38**	**46**	**26**	**35**

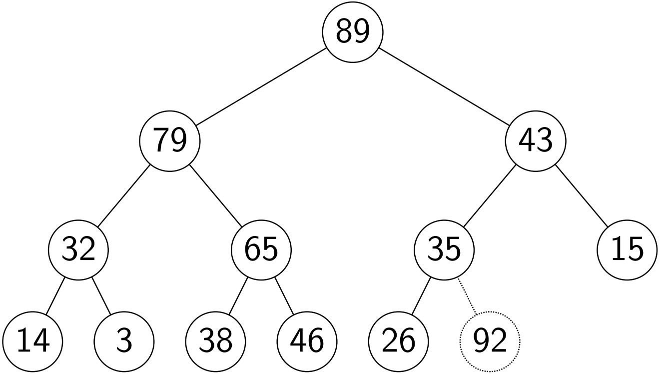
In-place Heap Sort (5)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**79**	**89**	**32**	**65**	**43**	**15**	**14**	**3**	**38**	**46**	**26**	**35**

- After the input array has been converted to a max-heap, the RemoveMax operation is used to repeatedly remove the maximum element.

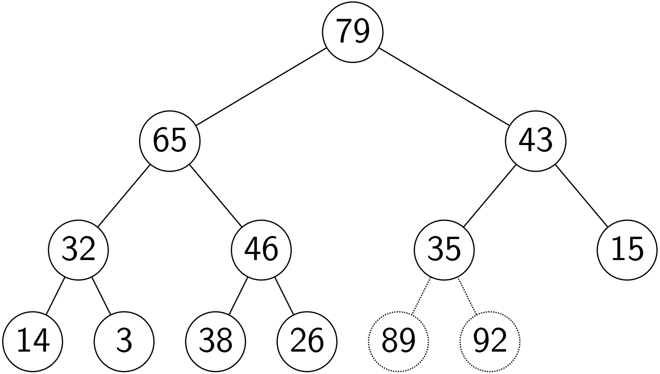
In-place Heap Sort (6)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**89**	**79**	**43**	**32**	**65**	**35**	**15**	**14**	**3**	**38**	**46**	**26**	92

- As each element is removed, it is placed at the end of the array, in the position freed up by the remove operation (which is no longer part of the tree).

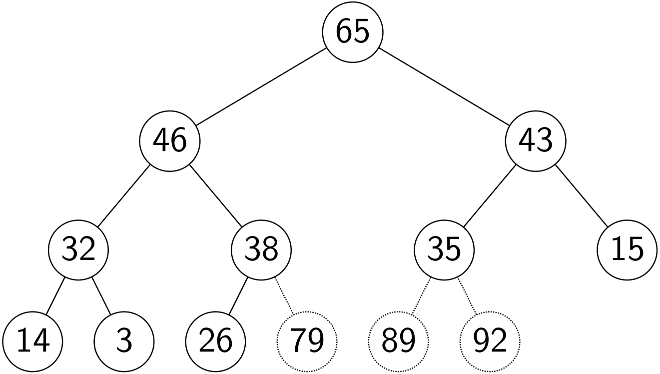
In-place Heap Sort (7)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**79**	**65**	**43**	**32**	**46**	**35**	**15**	**14**	**3**	**38**	**26**	89	92

- As each element is removed, it is placed at the end of the array, in the position freed up by the remove operation (which is no longer part of the tree).

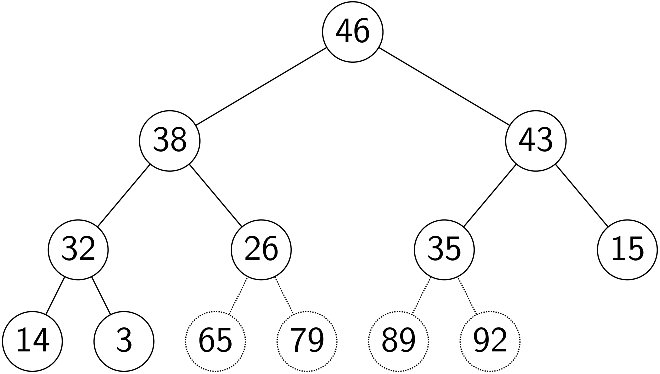
In-place Heap Sort (8)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**65**	**46**	**43**	**32**	**38**	**35**	**15**	**14**	**3**	**26**	79	89	92

- As each element is removed, it is placed at the end of the array, in the position freed up by the remove operation (which is no longer part of the tree).

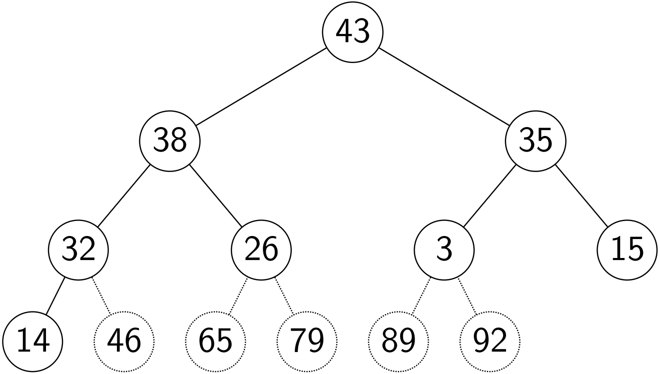
In-place Heap Sort (9)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**46**	**38**	**43**	**32**	**26**	**35**	**15**	**14**	**3**	65	79	89	92

- As each element is removed, it is placed at the end of the array, in the position freed up by the remove operation (which is no longer part of the tree).

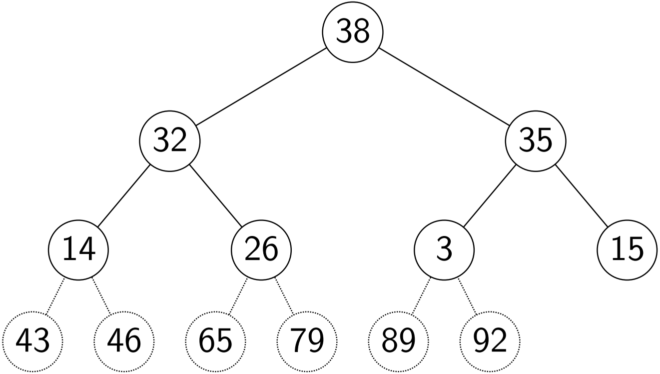
In-place Heap Sort (10)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**43**	**38**	**35**	**32**	**26**	**3**	**15**	**14**	46	65	79	89	92

- As each element is removed, it is placed at the end of the array, in the position freed up by the remove operation (which is no longer part of the tree).

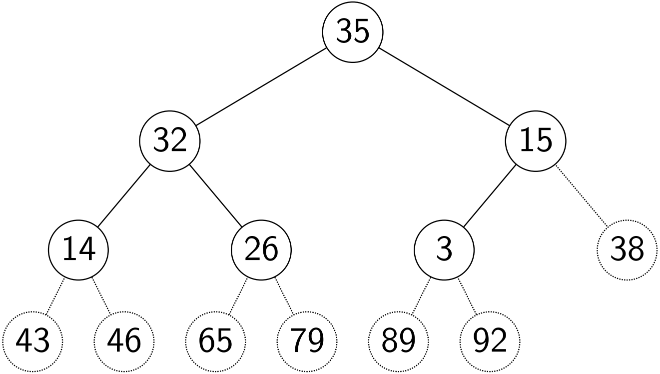
In-place Heap Sort (11)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**38**	**32**	**35**	**14**	**26**	**3**	**15**	43	46	65	79	89	92

- As each element is removed, it is placed at the end of the array, in the position freed up by the remove operation (which is no longer part of the tree).

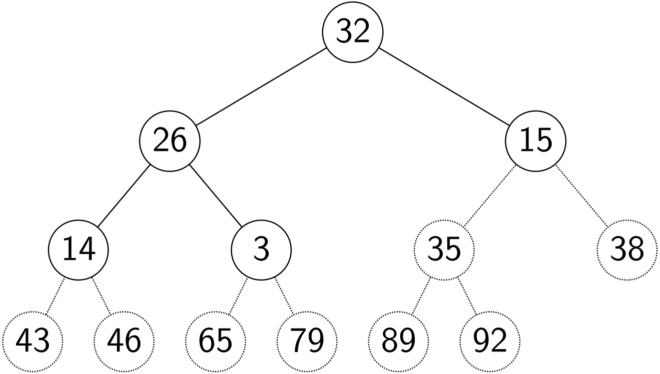
In-place Heap Sort (12)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**35**	**32**	**15**	**14**	**26**	**3**	38	43	46	65	79	89	92

- As each element is removed, it is placed at the end of the array, in the position freed up by the remove operation (which is no longer part of the tree).

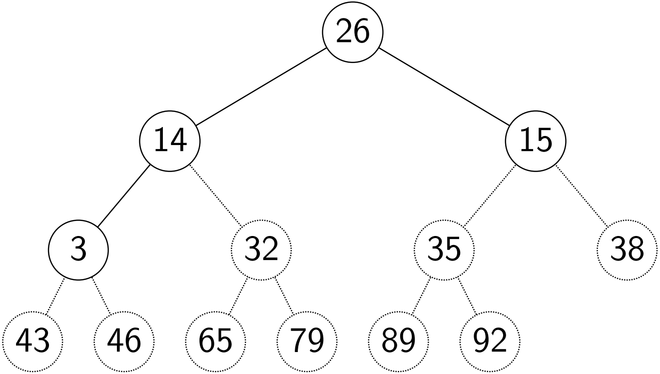
In-place Heap Sort (13)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**32**	**26**	**15**	**14**	**3**	35	38	43	46	65	79	89	92

- As each element is removed, it is placed at the end of the array, in the position freed up by the remove operation (which is no longer part of the tree).

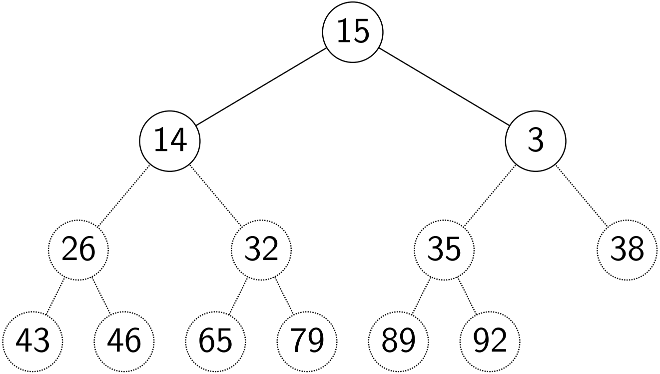
In-place Heap Sort (14)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**26**	**14**	**15**	**3**	32	35	38	43	46	65	79	89	92

- As each element is removed, it is placed at the end of the array, in the position freed up by the remove operation (which is no longer part of the tree).

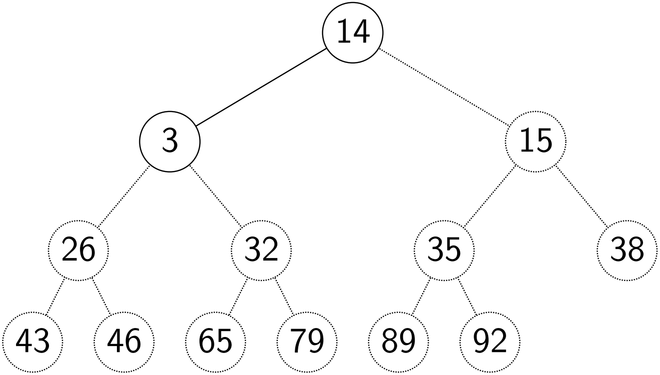
In-place Heap Sort (15)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**15**	**14**	**3**	26	32	35	38	43	46	65	79	89	92

- As each element is removed, it is placed at the end of the array, in the position freed up by the remove operation (which is no longer part of the tree).

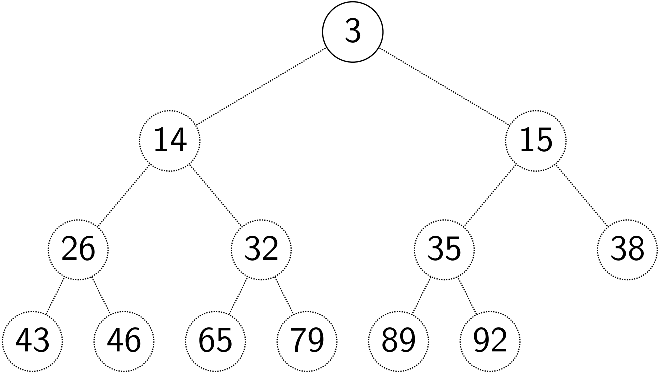
In-place Heap Sort (16)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**14**	**3**	15	26	32	35	38	43	46	65	79	89	92

- As each element is removed, it is placed at the end of the array, in the position freed up by the remove operation (which is no longer part of the tree).

In-place Heap Sort (17)



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**3**	14	15	26	32	35	38	43	46	65	79	89	92

- After the last heap removal, the array is sorted.
- Heap Sort is the only $\Theta(n \log_2 n)$ sorting algorithm which requires $\Theta(1)$ extra space.

Sorting Algorithms

	Running Time: Best Case	Running Time: Expected Case	Running Time: Worst Case	Extra Space	Stable?
Selection Based					
Heap Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(1)$	No
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	Yes
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	No
Divide and Conquer					
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	Yes
Quicksort	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n)$	Yes
Other					
Bubble Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	Yes
Radix Sort[1]	$\Theta(dn + b)$	$\Theta(dn + b)$	$\Theta(dn + b)$	$\Theta(n + b)$	Yes

[1] Integers only: d-digit values in base b

Sorting Algorithms

	Running Time			Extra Space	Stable?
	Best Case	**Expected Case**	**Worst Case**		
Selection Based					
Heap Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(1)$	No
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	Yes
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	No
Divide and Conquer					
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	Yes
Quicksort	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n)$	Yes
Other					
Bubble Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	Yes
Radix Sort[1]	$\Theta(dn + b)$	$\Theta(dn + b)$	$\Theta(dn + b)$	$\Theta(n + b)$	Yes

[1] Integers only: d-digit values in base b

Heap Sort

procedure $\textsc{HeapSort}(A, n)$
  $H \leftarrow$ Empty Heap
  for $i = 0, 1, \ldots, n-1$ **do**
    $H.\textsc{Insert}(A[i])$
  end for
  for $i = 0, 1, \ldots, n-1$ **do**
    $A[i] \leftarrow H.\textsc{RemoveMin}(\ )$
  end for
end procedure

- Both loops require $\Theta(n \log n)$ time, so $\textsc{HeapSort}$ is $\Theta(n \log n)$.
- Using the $\textsc{Heapify}$ algorithm instead of the first loop, it is possible to construct the heap in $\Theta(n)$ time instead of $\Theta(n \log n)$ time. The second loop will still require $\Theta(n \log n)$ time.

Heap Sort is Optimal (1)

Theorem: Any comparison-based sorting algorithm requires

$$\Omega(n \log n)$$

operations in the worst case.

- This theorem implies that no general-purpose sorting algorithm is better than $\Theta(n \log n)$.
- Therefore, asymptotically, heap sort is optimal.
- In practice, heap sort has very high overhead in most cases.

Heap Sort is Optimal (2)

Theorem: Any comparison-based sorting algorithm requires

$$\Omega(n \log n)$$

operations in the worst case.

Good Assignment Question: Prove that it is impossible to develop a comparison-based priority queue such that

$$\textsc{Insert} \in \Theta(\log \log n) \text{ and}$$

$$\textsc{RemoveMin} \in \Theta(\log \log n).$$