CMP302: Algorithms

Lecture 04: Heapsort

Mohamed Alaa El-Dien Aly
Computer Engineering Department
Cairo University
Fall 2013

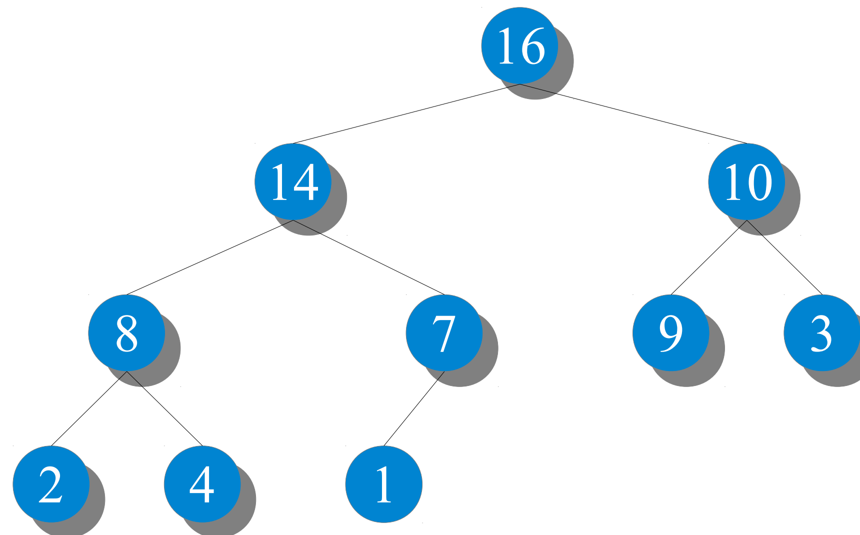
Agenda

- Heaps
- Heapsort

Acknowledgment
A lot of slides adapted from the slides of Erik Demaine and Charles Leiserson

Heaps

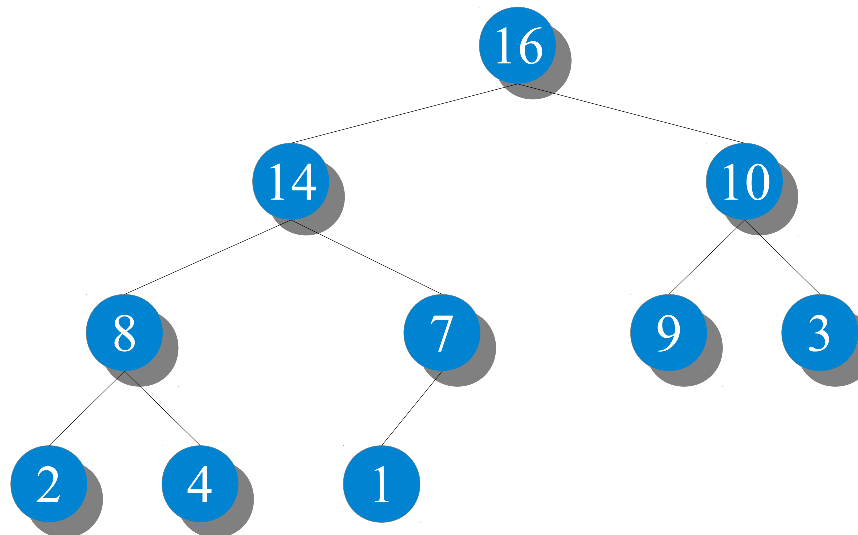
- "Nearly-complete" binary trees i.e. filled except possibly at the leaves
- Each node satisfies the "heap property":
  - Max-Heap: the value of the node ≥ values of the children
  - Min-Heap: the value of the node ≤ values of the children



Max-Heap

Heaps

- Max-Heap: the value of the node ≥ values of the children
- Fact: The maximum element is at the root.



Array Representation

- Heaps are better represented as an array filled from root to leaves
- Three functions required:

Parent of a node *i*

PARENT(i)

1 **return** $\lfloor i/2 \rfloor$

Left child of a node *i*

LEFT(i)

1 **return** $2i$

Right child of a node *i*

RIGHT(i)

1 **return** $2i + 1$

Properties of array *A*

A.length: size of array *A*

A.heap-size: size of heap inside array $A \leq A.length$

Array Representation

Parent of a node i

PARENT(i)

1 **return** $\lfloor i/2 \rfloor$

Left child of a node i

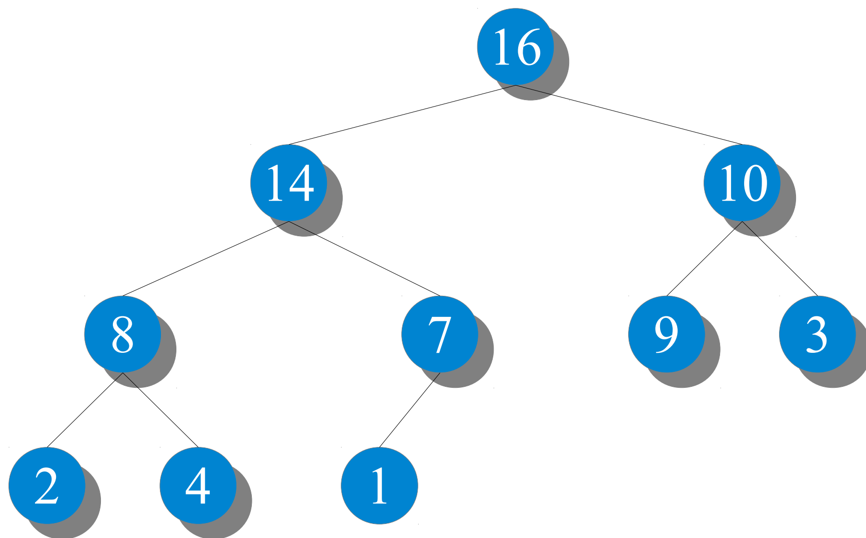
LEFT(i)

1 **return** $2i$

Right child of a node i

RIGHT(i)

1 **return** $2i + 1$



Conceptual Heap Representation

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

Array Representation

Note that indices start at 1 *not* 0.

Array Representation

Parent of a node i

$\text{PARENT}(i)$

1 **return** $\lfloor i/2 \rfloor$

Left child of a node i

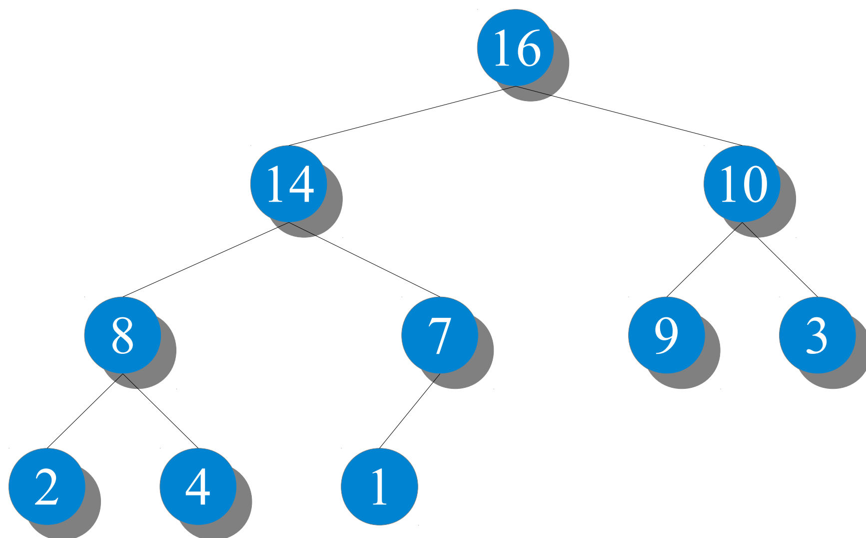
$\text{LEFT}(i)$

1 **return** $2i$

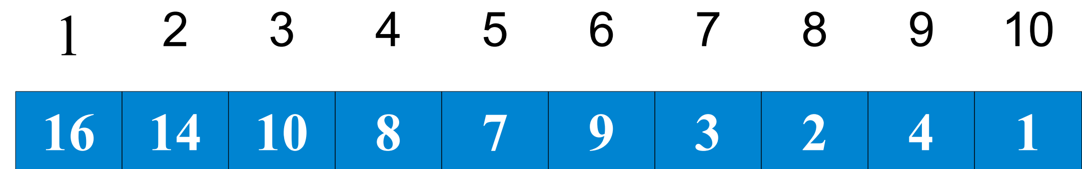
Right child of a node i

$\text{RIGHT}(i)$

1 **return** $2i + 1$



Conceptual Heap Representation



Array Representation

Note that indices start at 1 *not* 0.

Array Representation

Parent of a node *i*

PARENT(i)

1 **return** $\lfloor i/2 \rfloor$

Left child of a node *i*

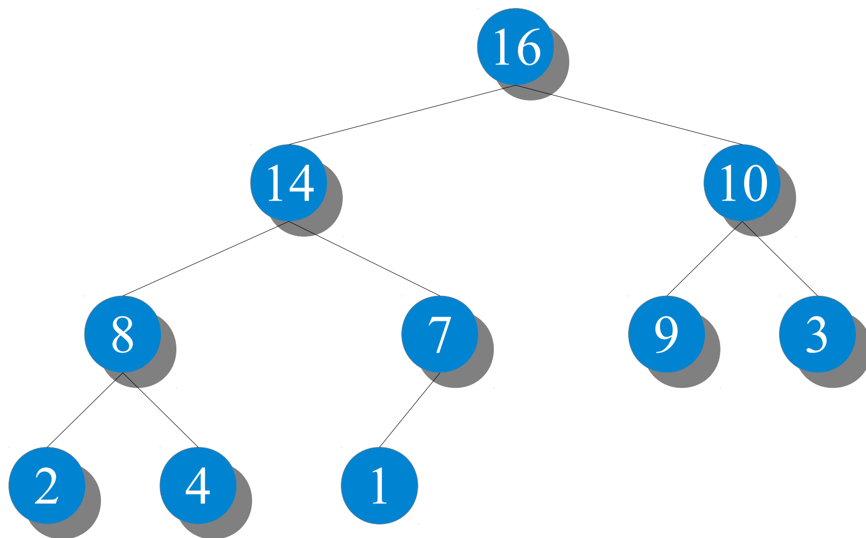
LEFT(i)

1 **return** $2i$

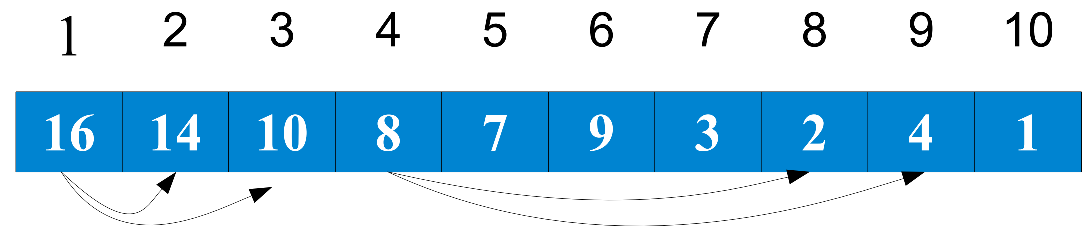
Right child of a node *i*

RIGHT(i)

1 **return** $2i + 1$



Conceptual Heap Representation



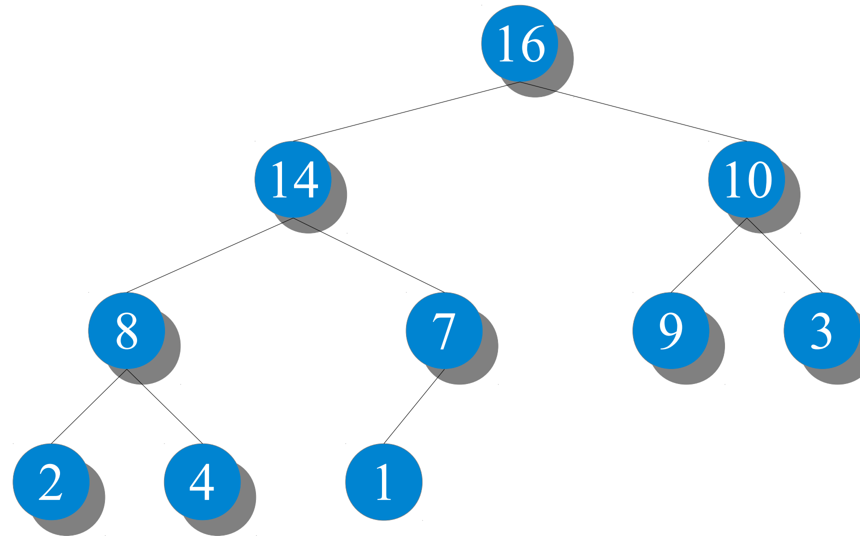
Array Representation

Note that indices start at 1 *not* 0.

Max-Heaps

Max-Heap Property

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



Properties of array *A*

A.length: size of array *A*

A.heap-size: size of heap inside array $A \leq A.length$

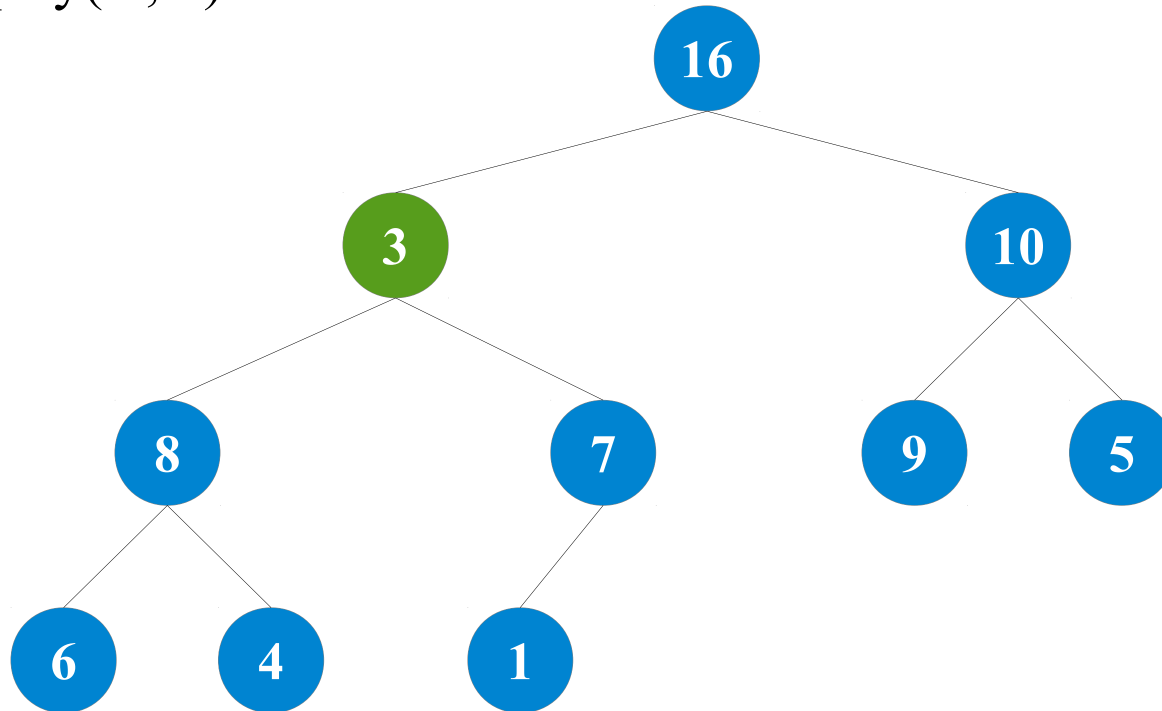
Max-Heapify

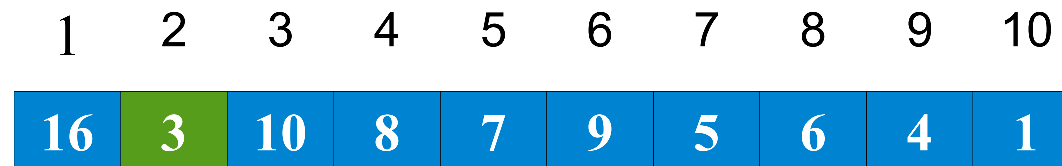
- Maintains the Max-Heap property of the heap
- Starting from a node in the heap that might violate the heap property i.e. smaller than its children
- Assumes its children are roots of Max-Heaps

```
MAX-HEAPIFY (A, i)
1 l = LEFT(i)
2 r = RIGHT(i)
3 if l ≤ A.heap-size and A[l] > A[i]
4     largest = l
5 else largest = i
6 if r ≤ A.heap-size and A[r] > A[largest]
7     largest = r
8 if largest ≠ i
9     exchange A[i] with A[largest]
10    MAX-HEAPIFY(A, largest)
```

Max-Heapify

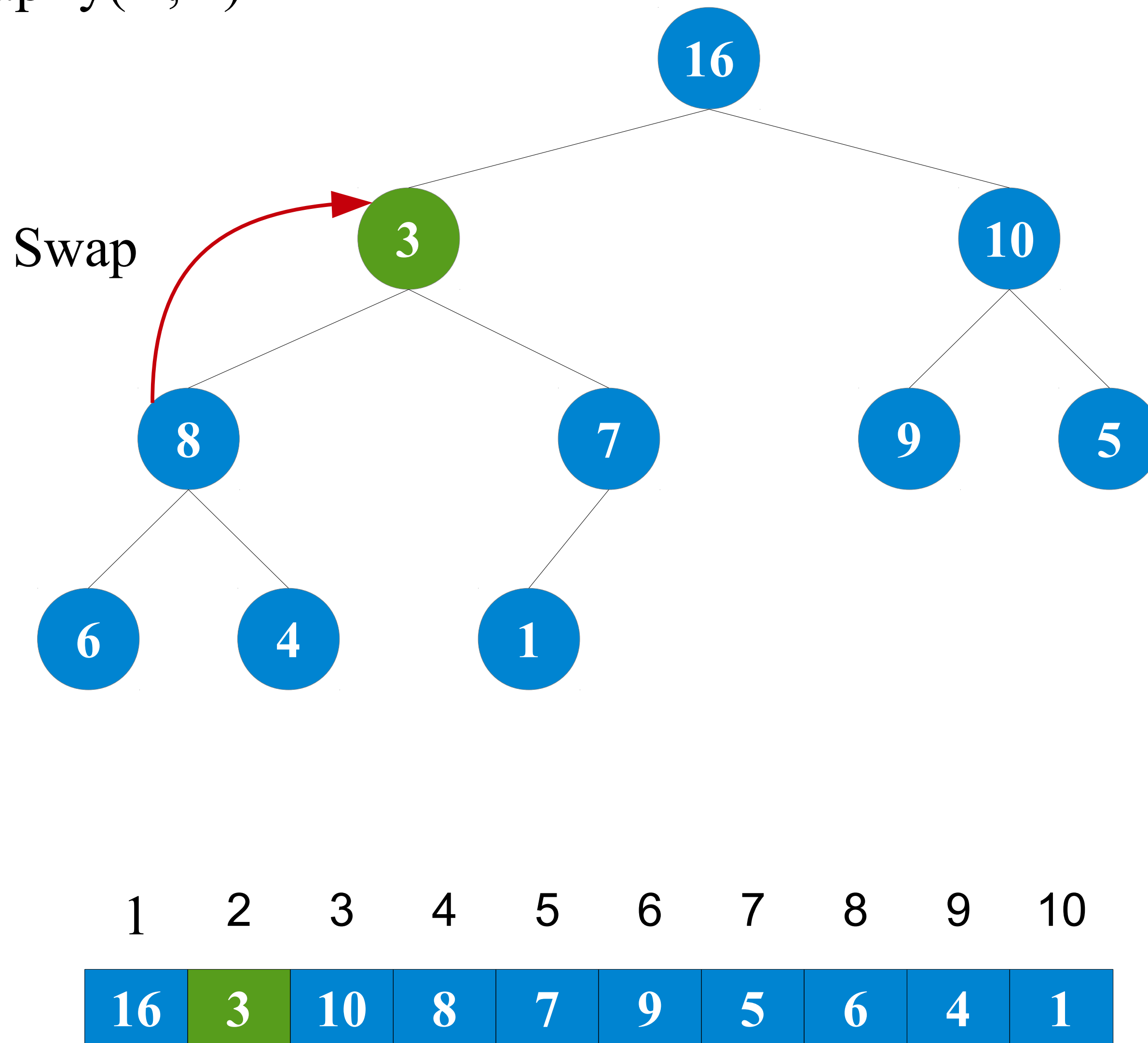
Max-Heapify(A, 2)





Max-Heapify

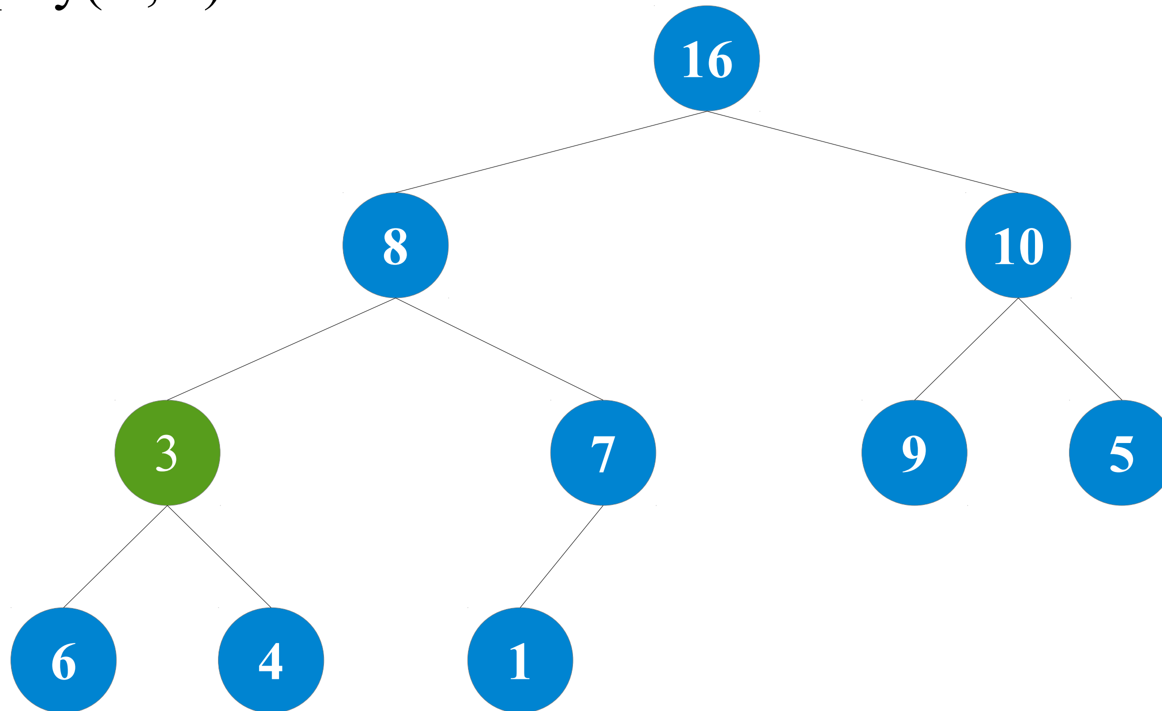
Max-Heapify(A, 2)

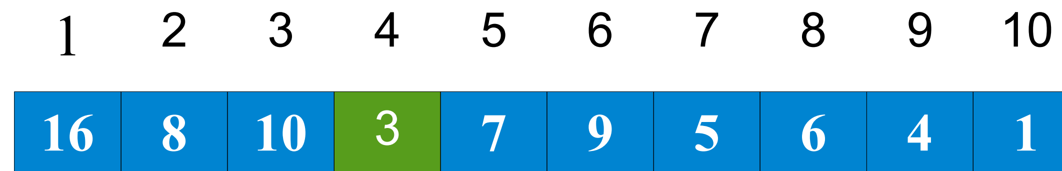


1	2	3	4	5	6	7	8	9	10
16	3	10	8	7	9	5	6	4	1

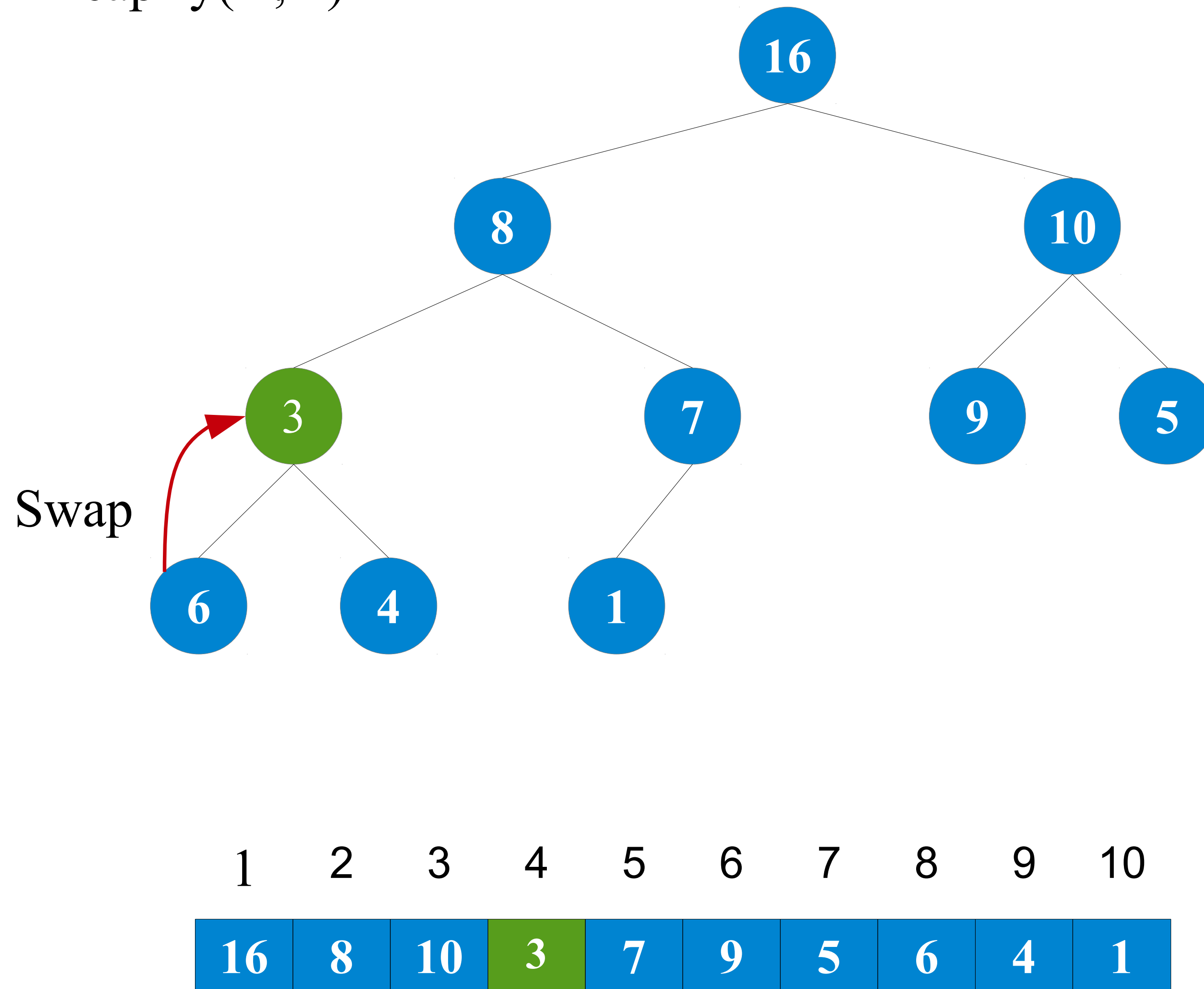
Max-Heapify

Max-Heapify(A, 4)



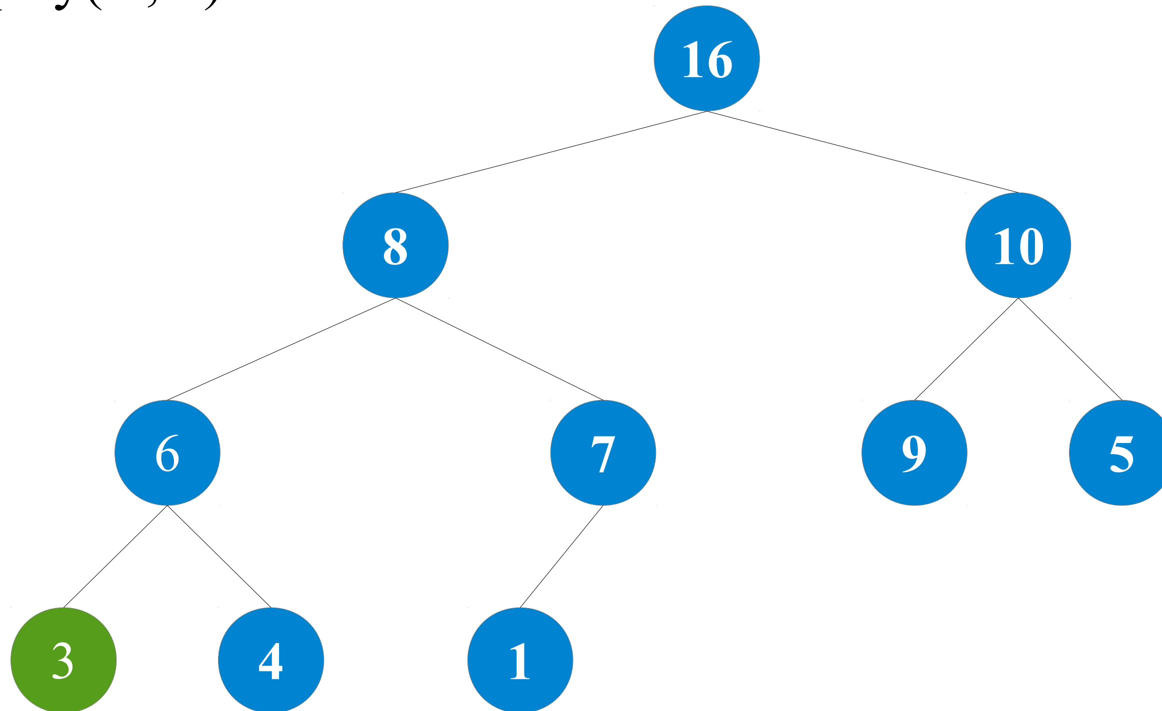


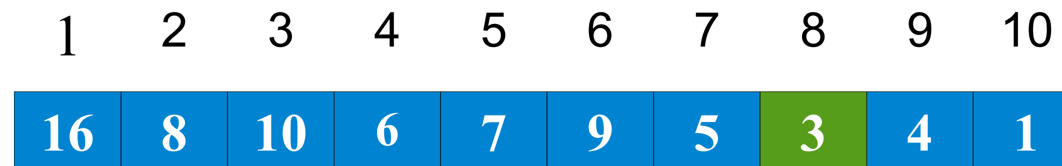
Max-Heapify



Max-Heapify

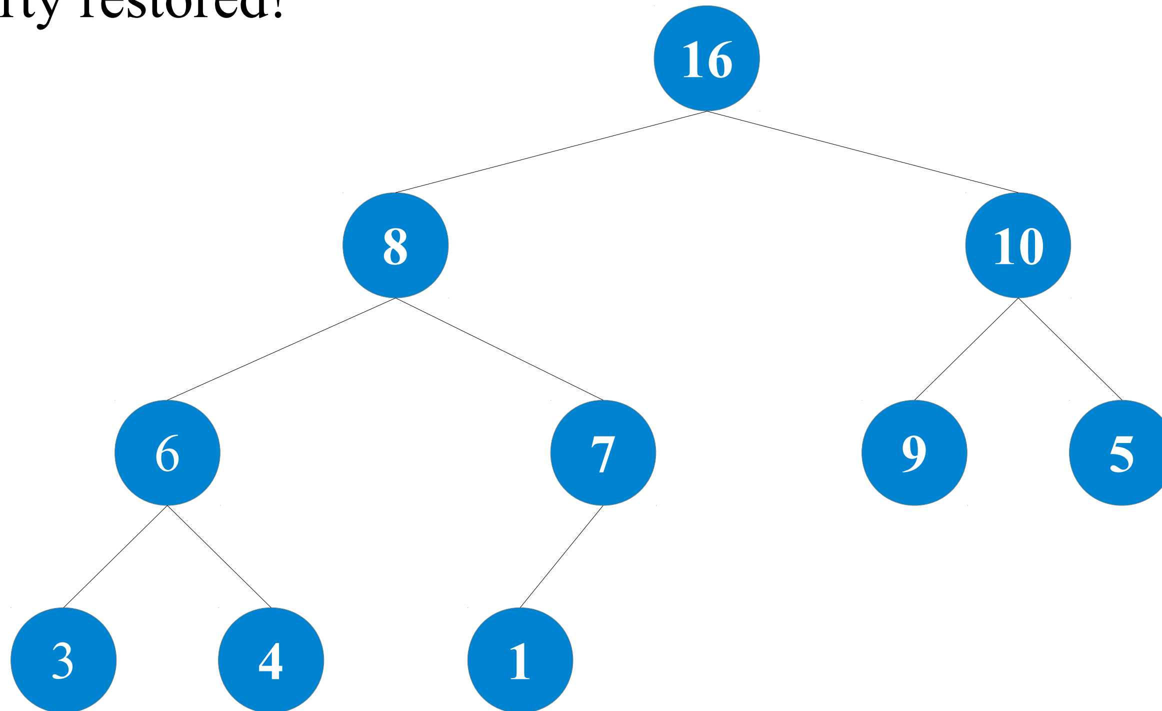
Max-Heapify(A, 8)

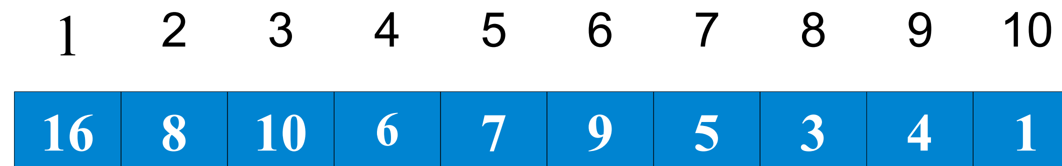




Max-Heapify

Heap property restored!





Max-Heapify Running Time

- Lines 1-9: $T(n) = \Theta(1)$
- Line 10: call on a subtree of node *i*. But what's the size of that subtree?

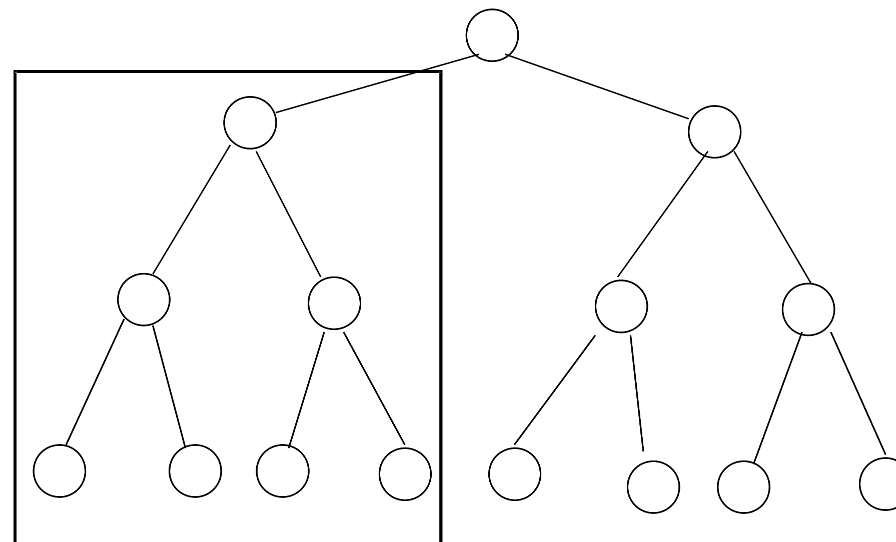
```
MAX-HEAPIFY (A, i)
1 l = LEFT(i)
2 r = RIGHT(i)
3 if l ≤ A.heap-size and A[l] > A[i]
4     largest = l
5 else largest = i
6 if r ≤ A.heap-size and A[r] > A[largest]
7     largest = r
8 if largest ≠ i
9     exchange A[i] with A[largest]
10    MAX-HEAPIFY(A, largest)
```

Max-Heapify Running Time

- Lines 1-9: $T(n) = \Theta(1)$
- Line 10: call on a subtree of node *i*. But what's the size of that subtree?

Best Case

In a complete binary tree, about half the nodes are included in the left subtree



15 nodes in tree

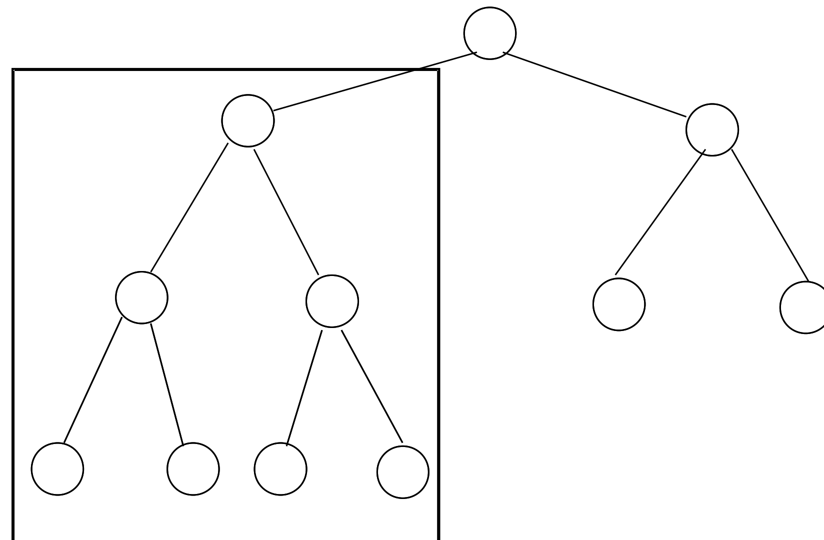
7 nodes in left subtree

Max-Heapify Running Time

- Lines 1-9: $T(n) = \Theta(1)$
- Line 10: call on a subtree of node *i*. But what's the size of that subtree?

Worst Case

In a binary tree where bottom is half-full, about two-thirds of the nodes are included in the left subtree



11 nodes in tree

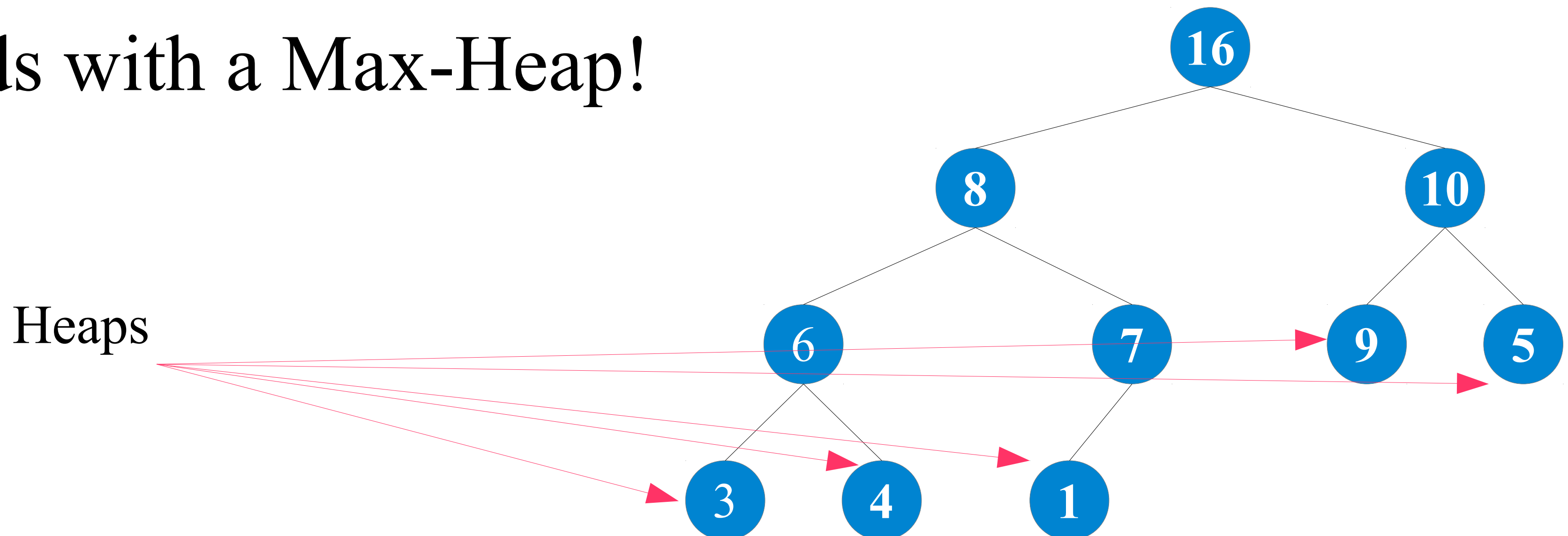
7 nodes in left subtree

Max-Heapify Running Time

- $T(n) \leq T(2n/3) + \Theta(1)$
- Case 2 of the Master Theorem ($a = 1$, $b = 3/2$):
  - $T(n) = O(\lg n)$
- Can also represent it as a function of h, the height of the node
  - The height is the number of links from the node to the leaves
  - $T(n) = O(h)$

Heap Building

- Now we can restore the heap property if it's lost.
- But how do we actually build a heap from a bunch of numbers?
- Each leaf is already a heap, albeit a useless one!
- Start from the second to last level, and heapify i.e. move values violating the Heap property downwards
- Ends with a Max-Heap!

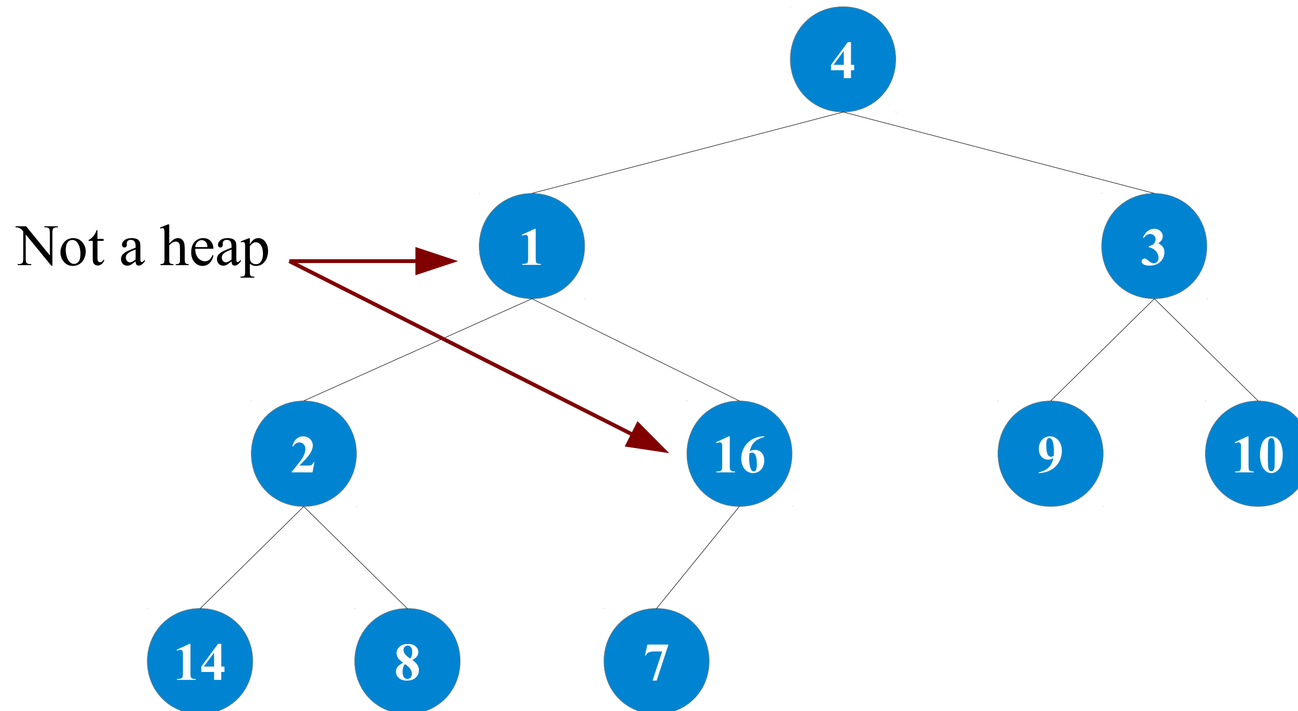


Heap Building

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length / 2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

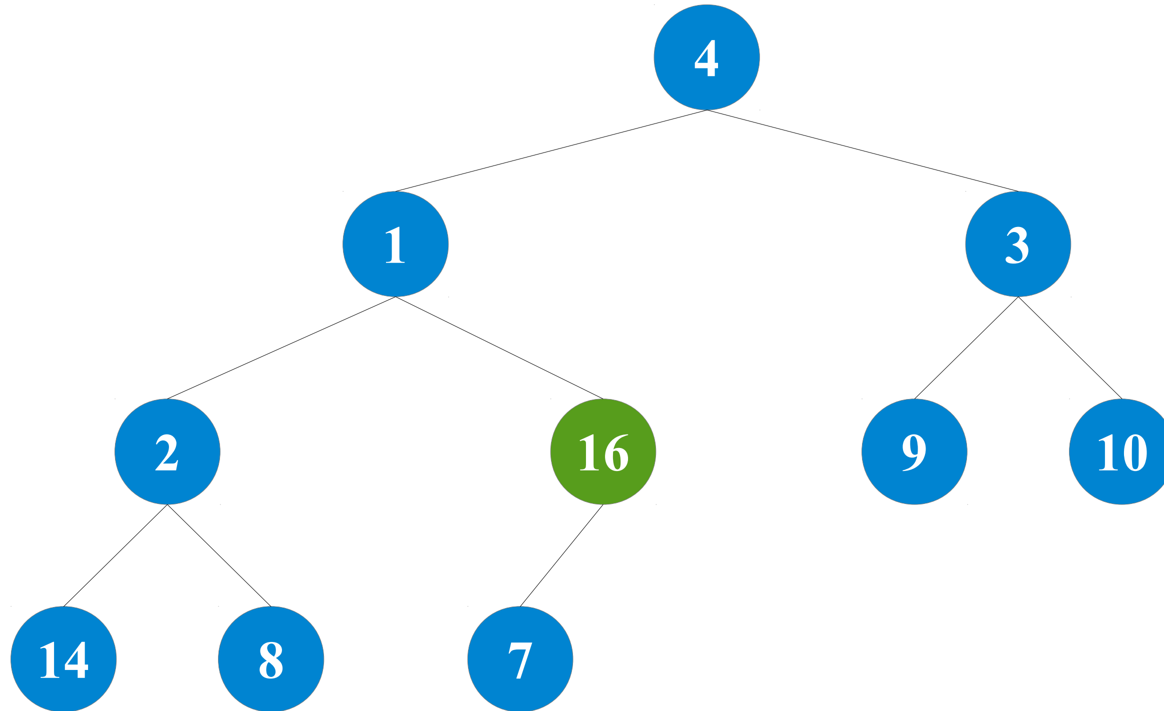
- Each leaf is already a heap, albeit a useless one!
- Start from the second to last level, and heapify i.e. move values violating the Heap property downwards
- Ends with a Max-Heap!

Heap Building



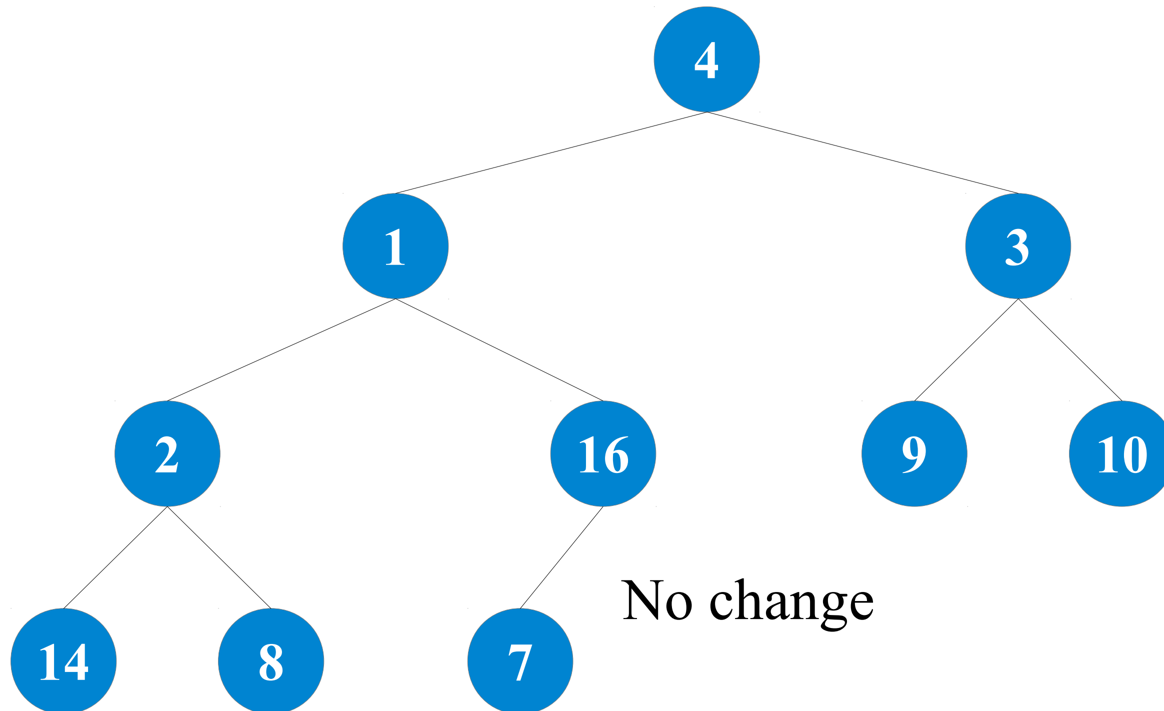
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Heap Building



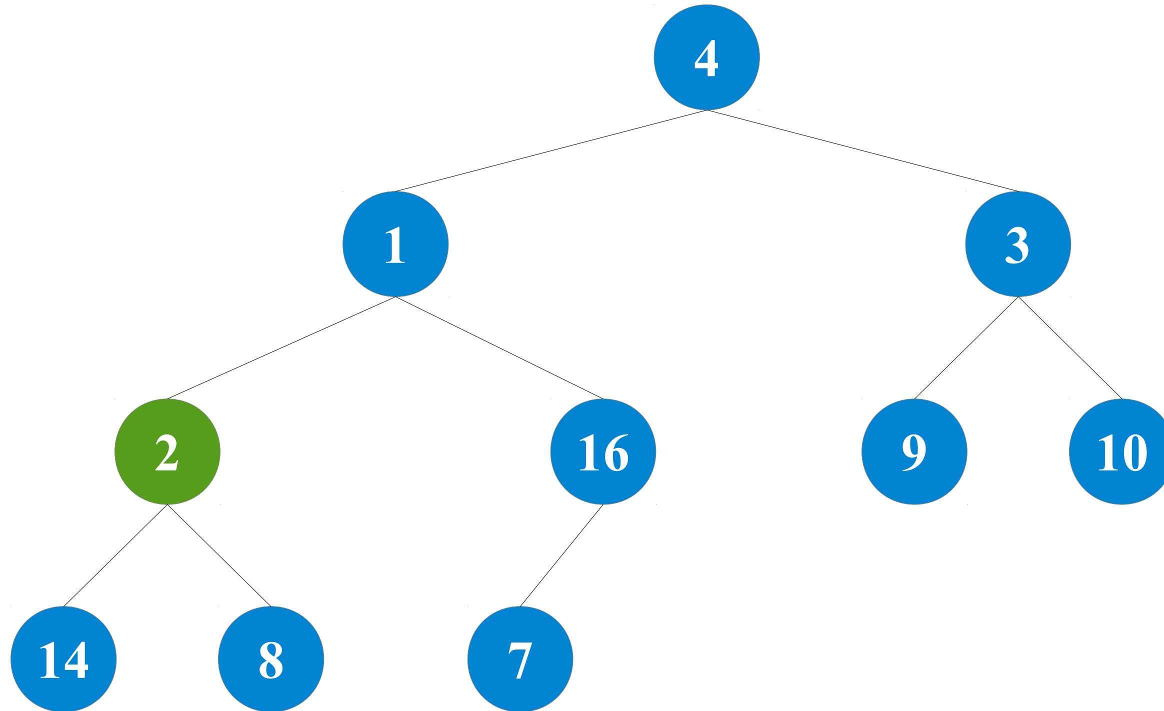
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Heap Building



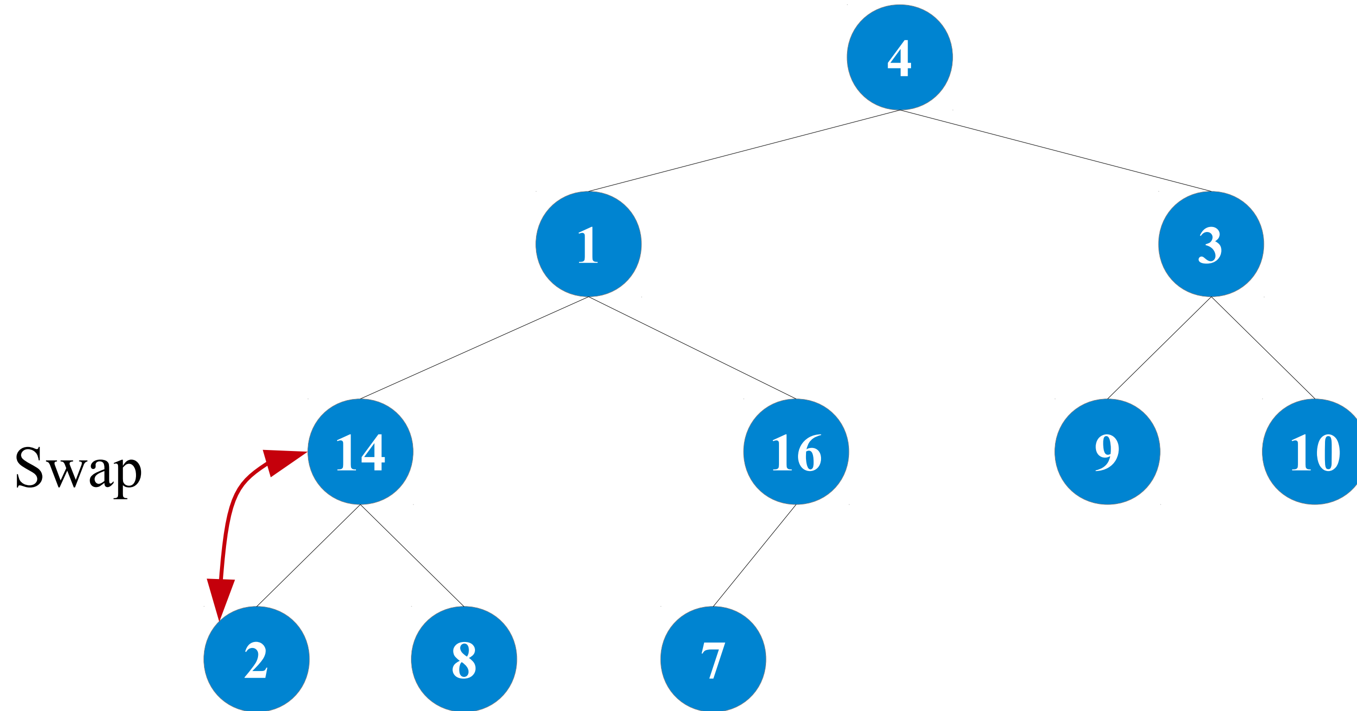
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Heap Building



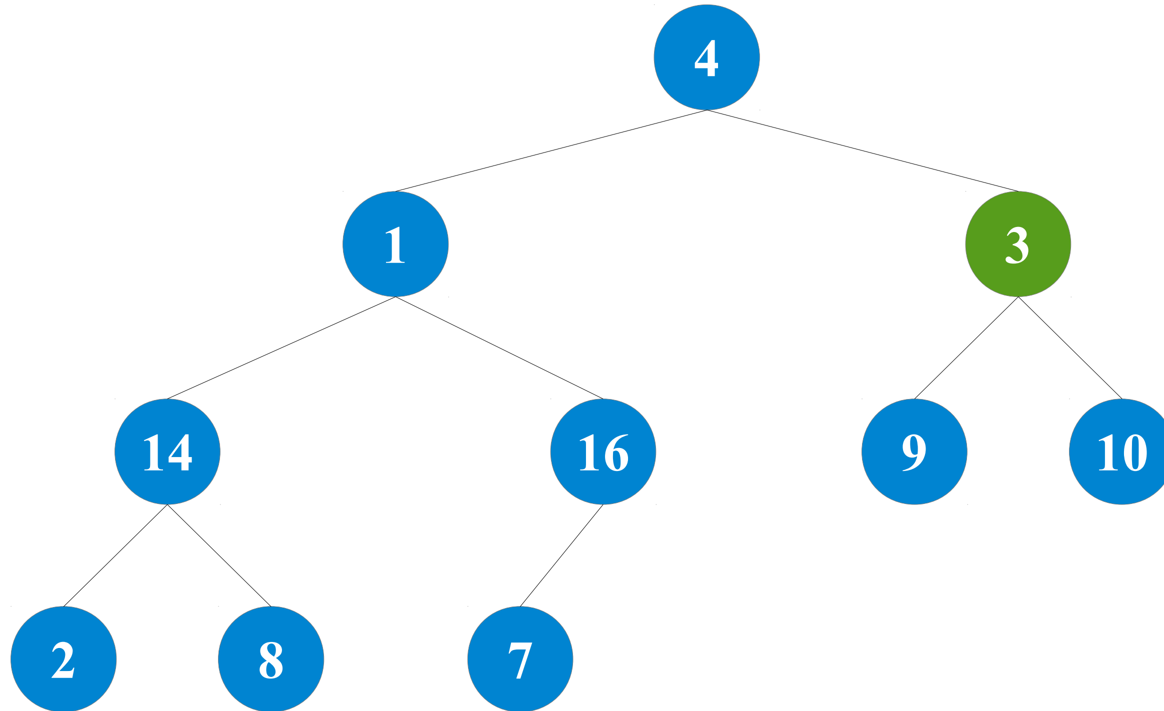
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Heap Building



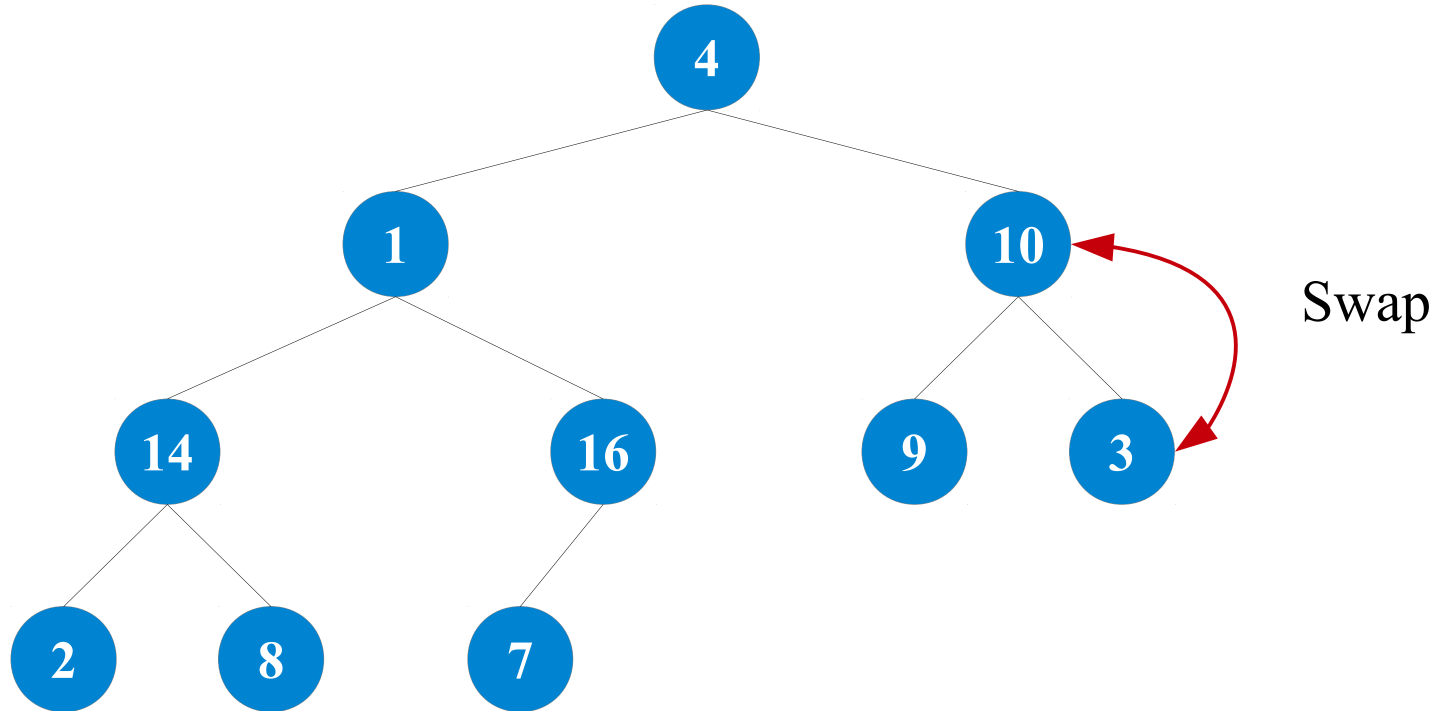
1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7

Heap Building



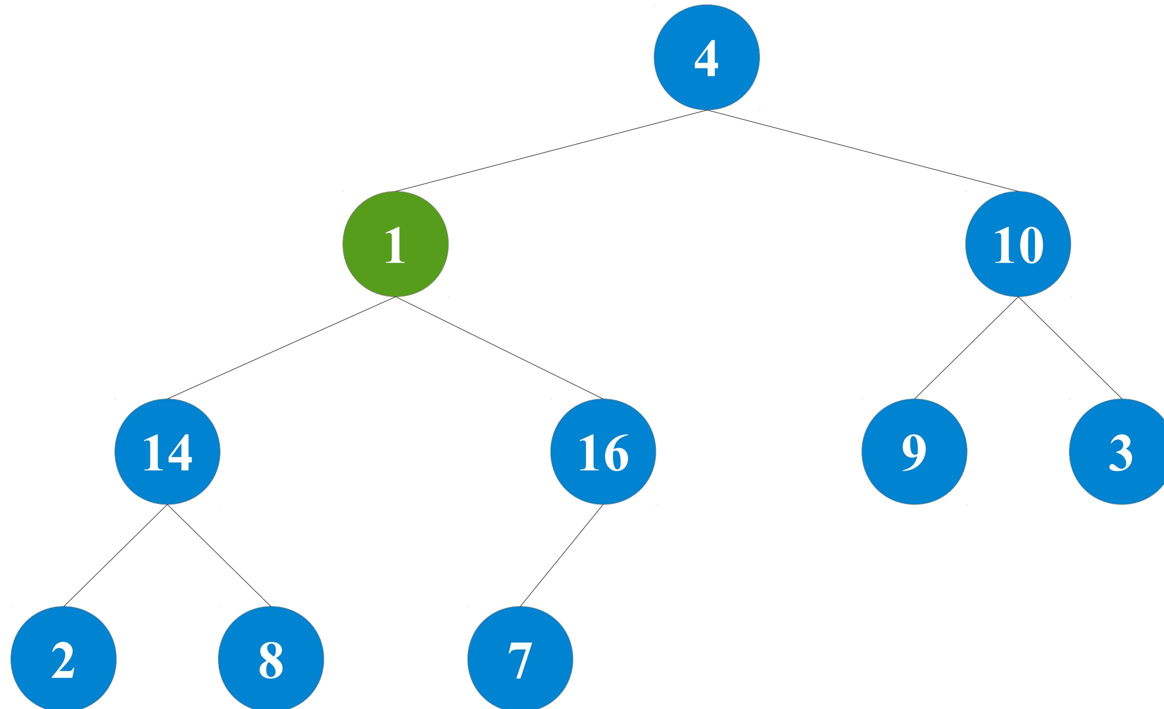
1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7

Heap Building



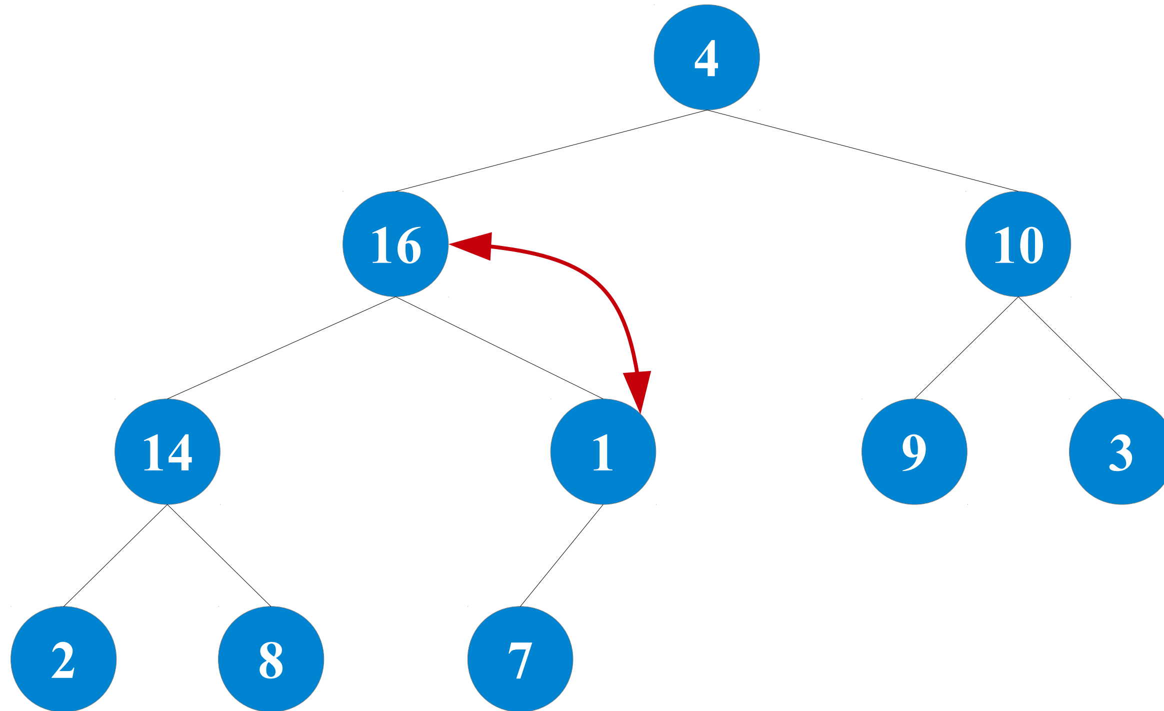
1	2	3	4	5	6	7	8	9	10
4	1	10	14	16	9	3	2	8	7

Heap Building



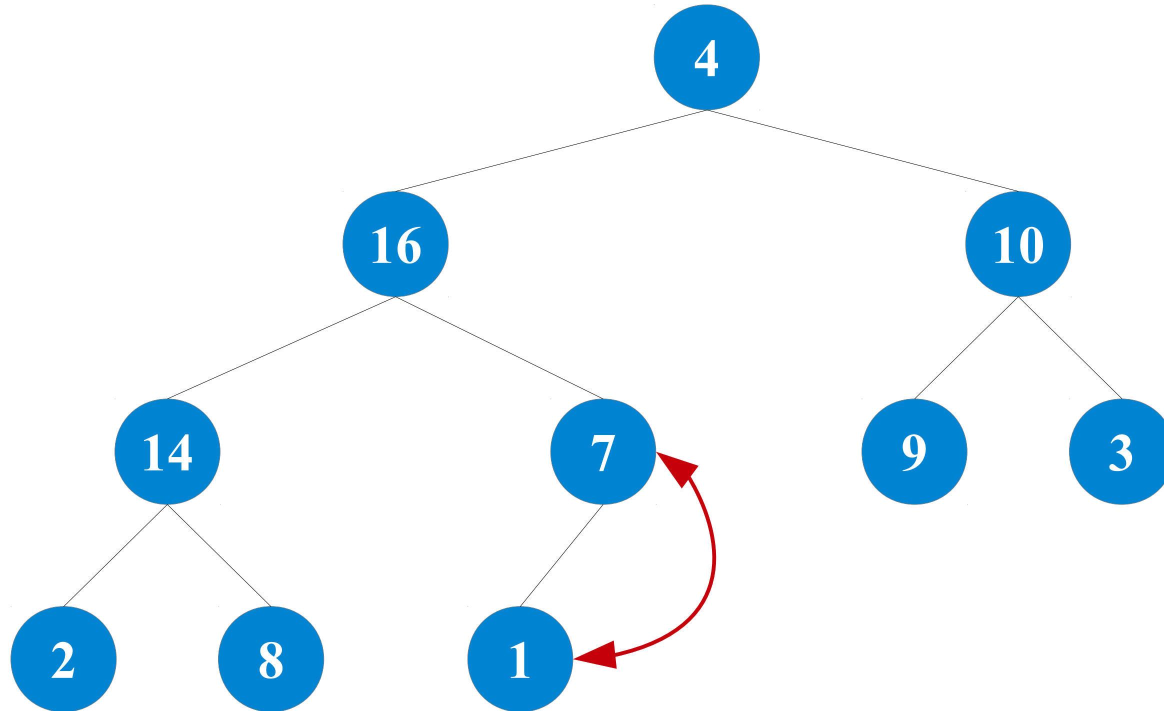
1	2	3	4	5	6	7	8	9	10
4	1	10	14	16	9	3	2	8	7

Heap Building



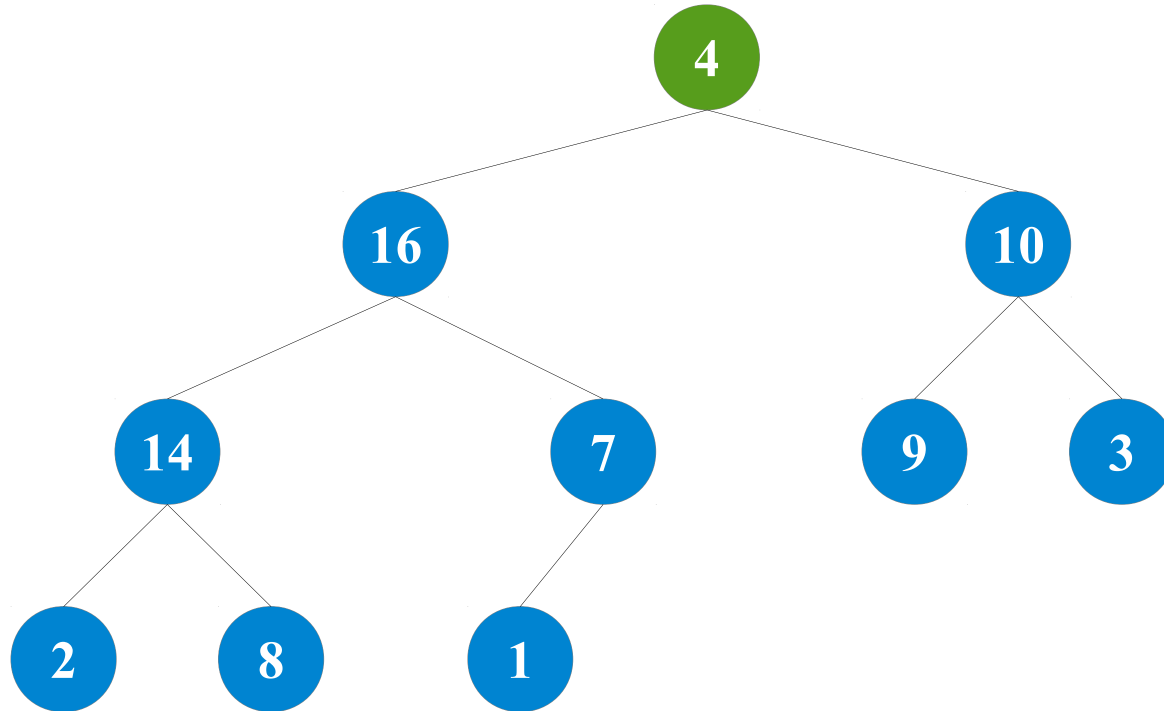
1	2	3	4	5	6	7	8	9	10
4	16	10	14	1	9	3	2	8	7

Heap Building



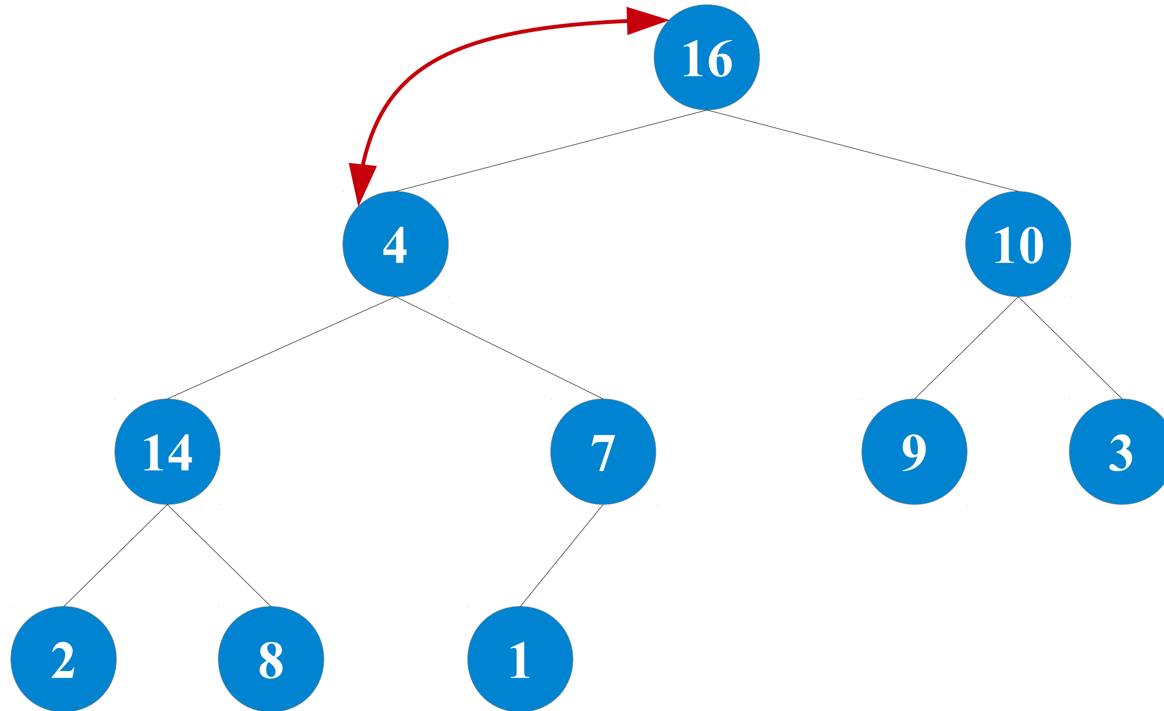
1	2	3	4	5	6	7	8	9	10
4	16	10	14	7	9	3	2	8	1

Heap Building



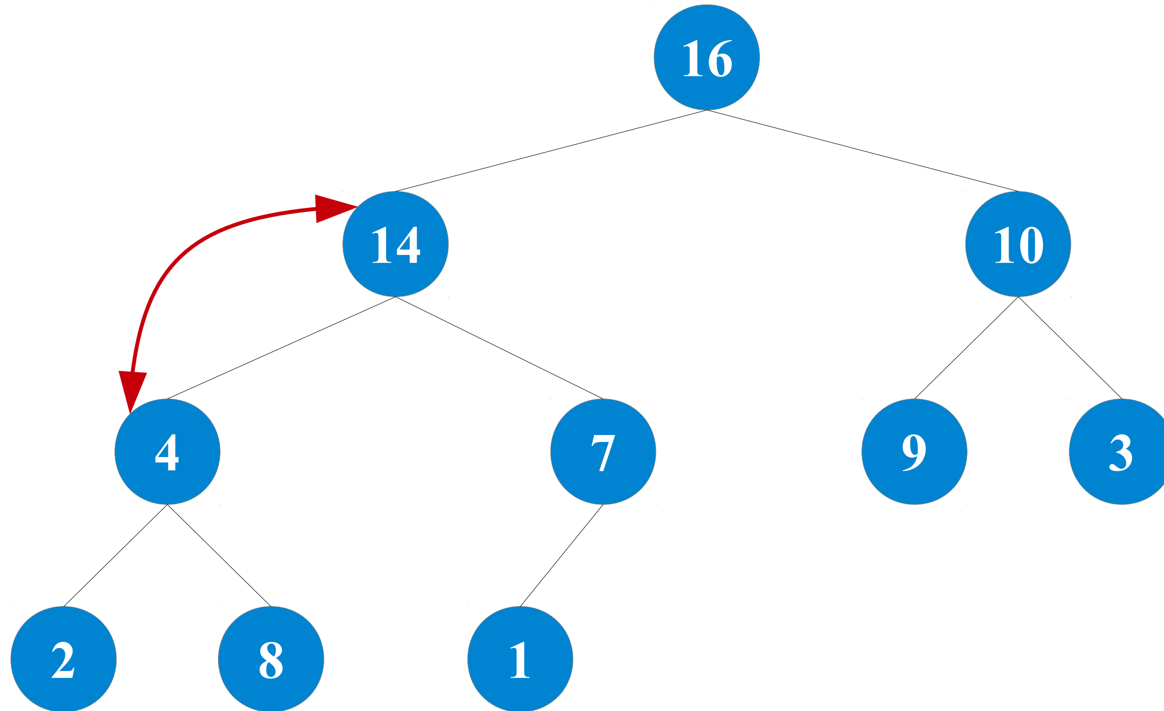
1	2	3	4	5	6	7	8	9	10
4	16	10	14	7	9	3	2	8	1

Heap Building



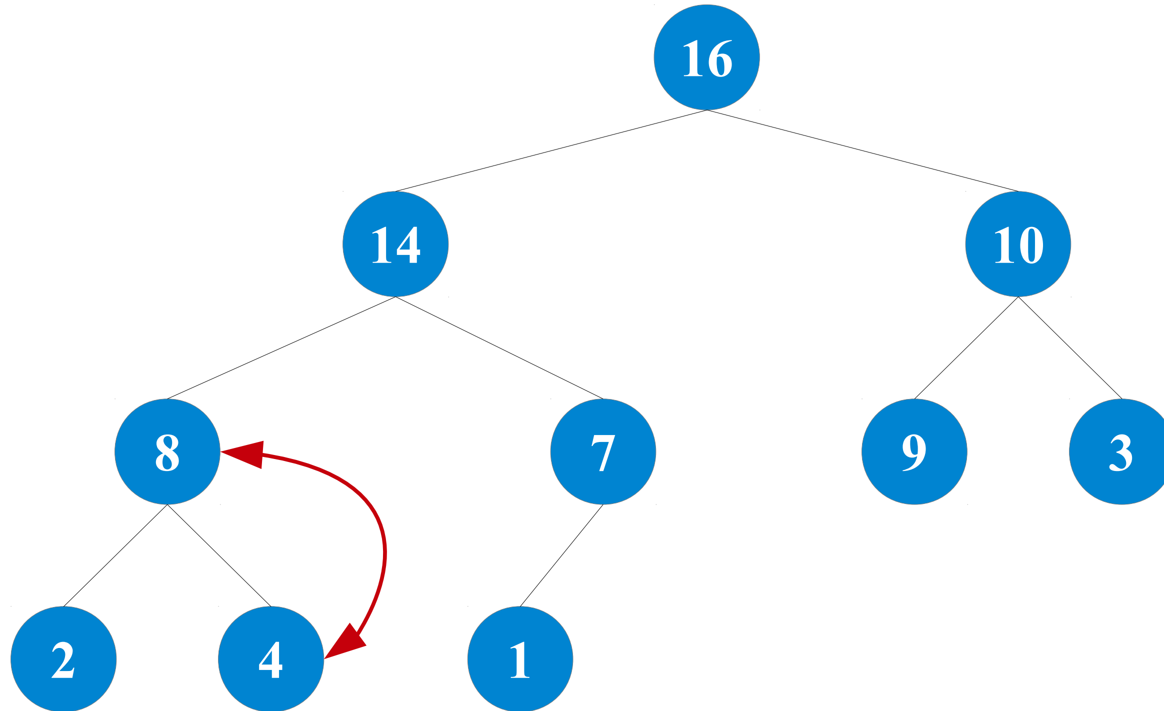
1	2	3	4	5	6	7	8	9	10
16	4	10	14	7	9	3	2	8	1

Heap Building



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

Heap Building



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

Build-Max-Heap Running Time

- Loose upper bound:
  - $n/2$ calls to Max-Heapify, each with time $\lg n$
  - $T(n) = O(n \lg n)$
- Tighter upper bound:
  - $T(n) = O(n)$

Number of nodes with height h

$$T(n) = \sum_{h=0}^{floor(\lg n)} O(h) \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

$$T(n) = O\left(n \sum_{h=0}^{floor(\lg n)} \frac{h}{2^h} \right)$$

$$\sum_{h=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

Why? $\frac{d}{dx}\left(\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \right) \rightarrow \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$

$$T(n) = O(n)$$

Correctness

Loop Invariant

- A technique used to prove correctness of algorithms
- A property that is satisfied at each iteration of the loop, hence invariant.
- Show that it is satisfied at the beginning, during, and at the end of the loop

```
Build-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length / 2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Correctness

Invariant

At the start of each iteration of the loop, each node $i+1, i+2, .., n$ is the root of a Max-Heap.

Initialization

At the start, $i = \lfloor n / 2 \rfloor$. Each node $i + 1 \ldots n$ is a leaf i.e. a root of a max-heap

```
Build-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length / 2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Correctness

Invariant
At the start of each iteration of the loop, each node $i+1, i+1, .., n$ is the root of a Max-Heap.

Maintenance
Since the children of node i have higher indices (i.e. $2i$ and $2i + 1$), by the loop invariant they are already the roots of max-heaps. Since Max-Heapify maintains the max-heap property at node i, node i after the iteration is the root of a max-heap, maintaining the property for the next iteration.

```
Build-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length / 2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Correctness

Invariant

At the start of each iteration of the loop, each node $i+1, i+1, .., n$ is the root of a Max-Heap.

Termination

At the end of the loop $i = 0$, and by Max-Heapify in the last iteration, node 1 is the root of a max-heap. Therefore, the invariant is satisfied and A now contains a max-heap.

```
Build-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length / 2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Heapsort

- Given a bunch of unsorted numbers, if we build a Max-Heap out of them:
  - where is the maximum number now?
  - where should it be?

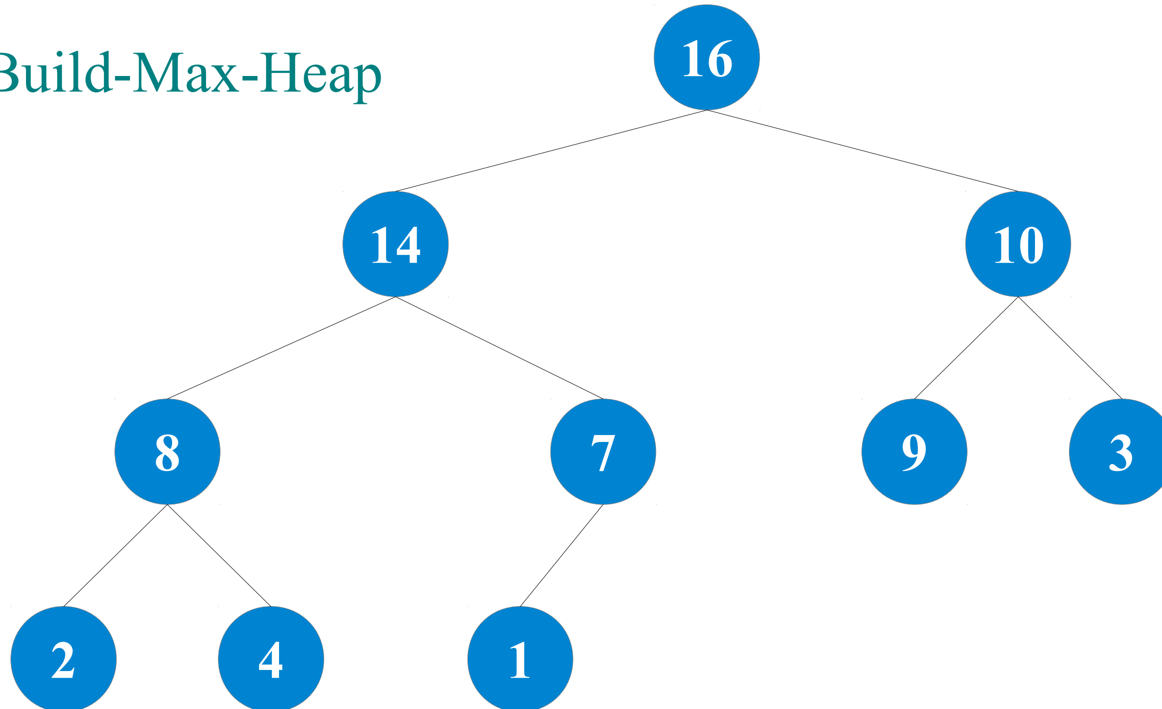
Heapsort

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 1
3      exchange A[i] with A[1]
4      A.heap-size --
5      MAX-HEAPIFY(A, 1)
```

- Build a Max-Heap from the array → put the maximum value at the front of *A*
- Put the max value in its right sorted place at the end and the value at the end in the root
- Heapify and repeat ...

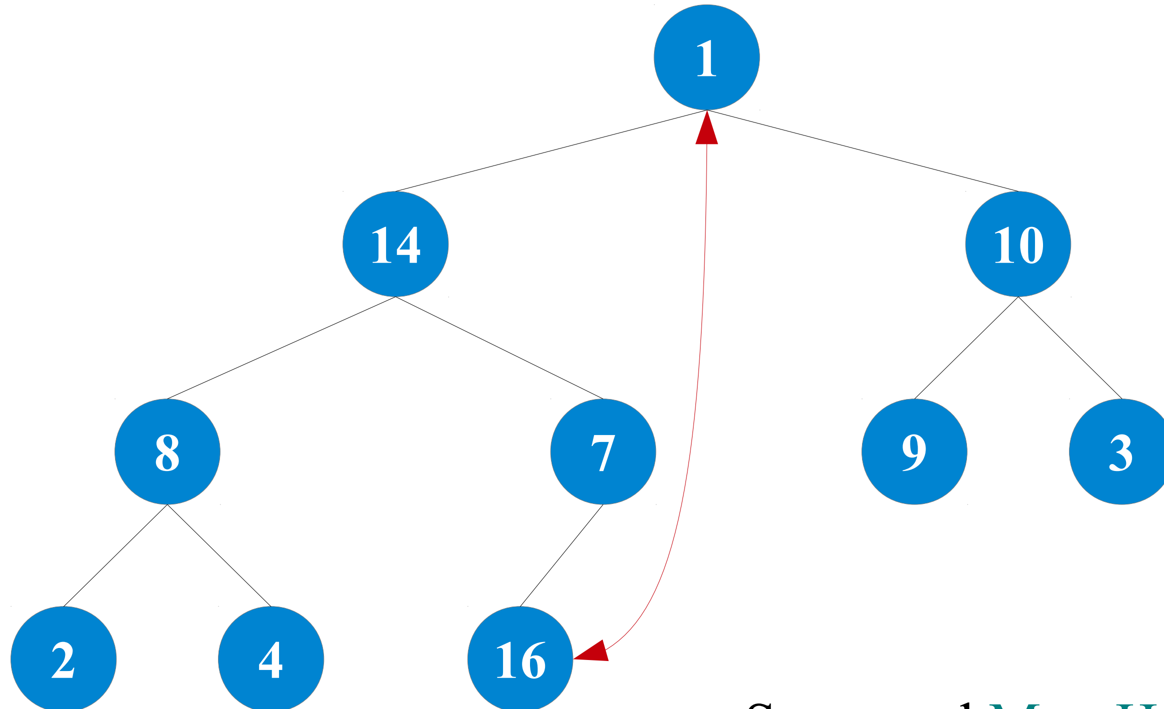
Heapsort

After calling Build-Max-Heap



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

Heapsort

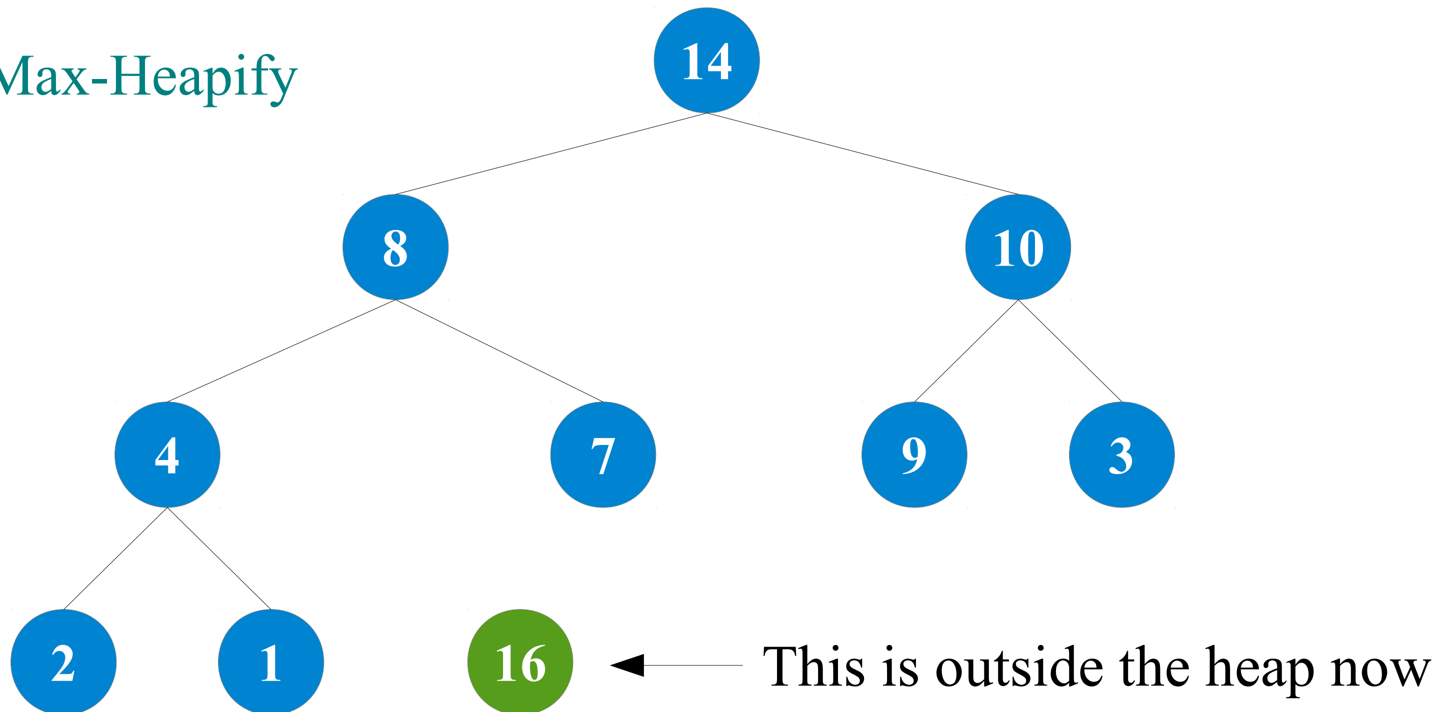


Swap and Max-Heapify

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

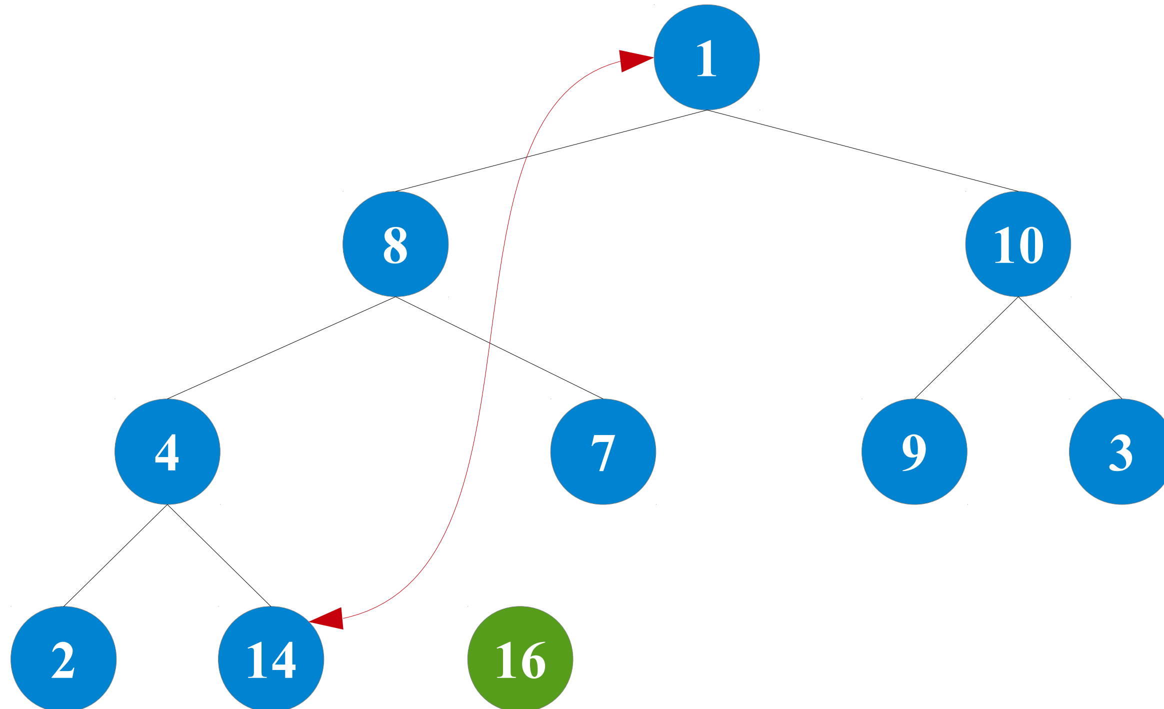
Heapsort

After calling Max-Heapify



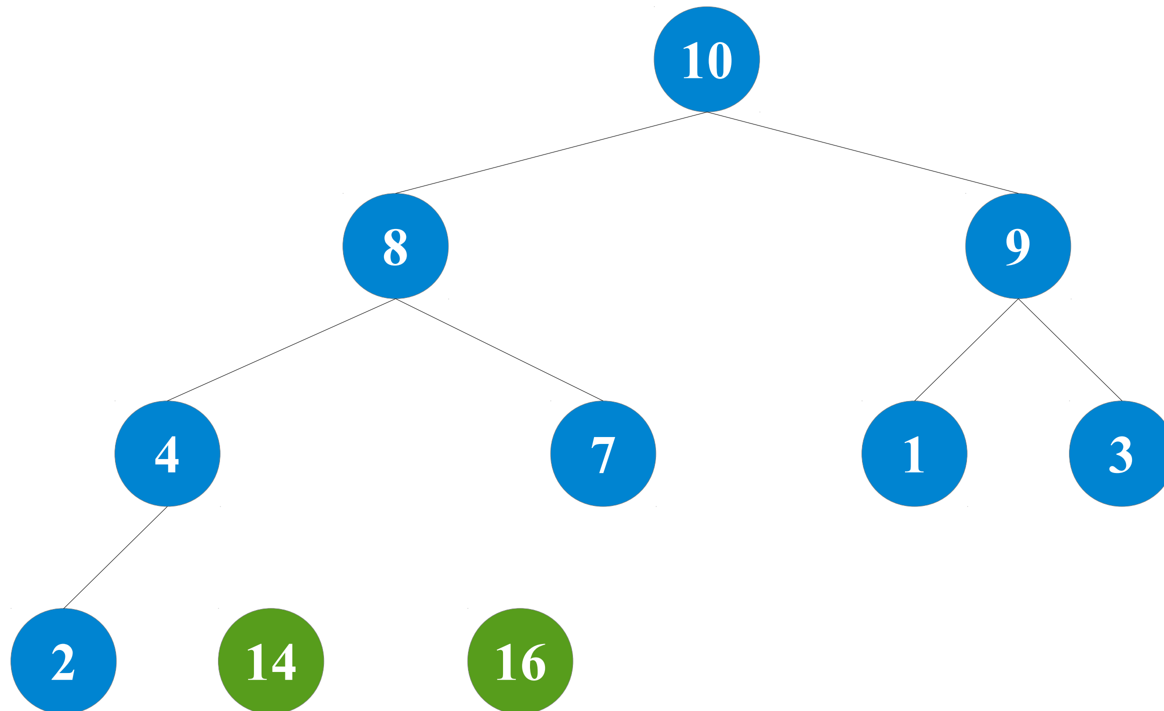
1	2	3	4	5	6	7	8	9	10
14	8	10	4	7	9	3	2	1	16

Heapsort



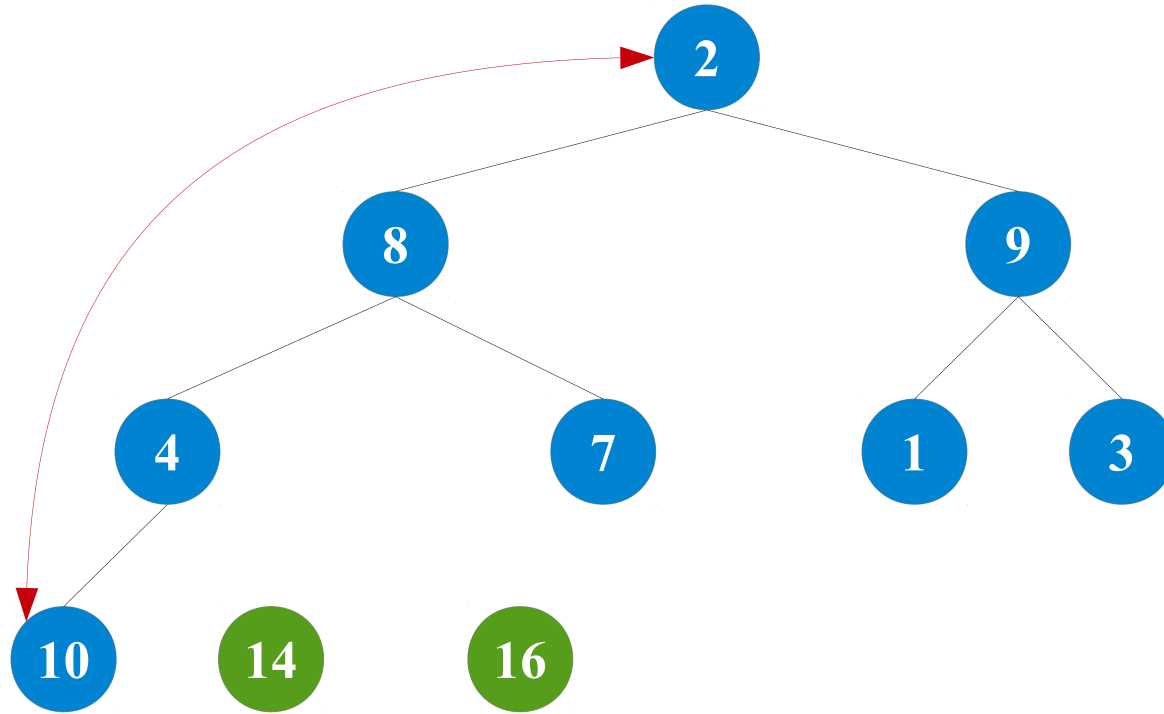
1	2	3	4	5	6	7	8	9	10
1	8	10	4	7	9	3	2	14	16

Heapsort



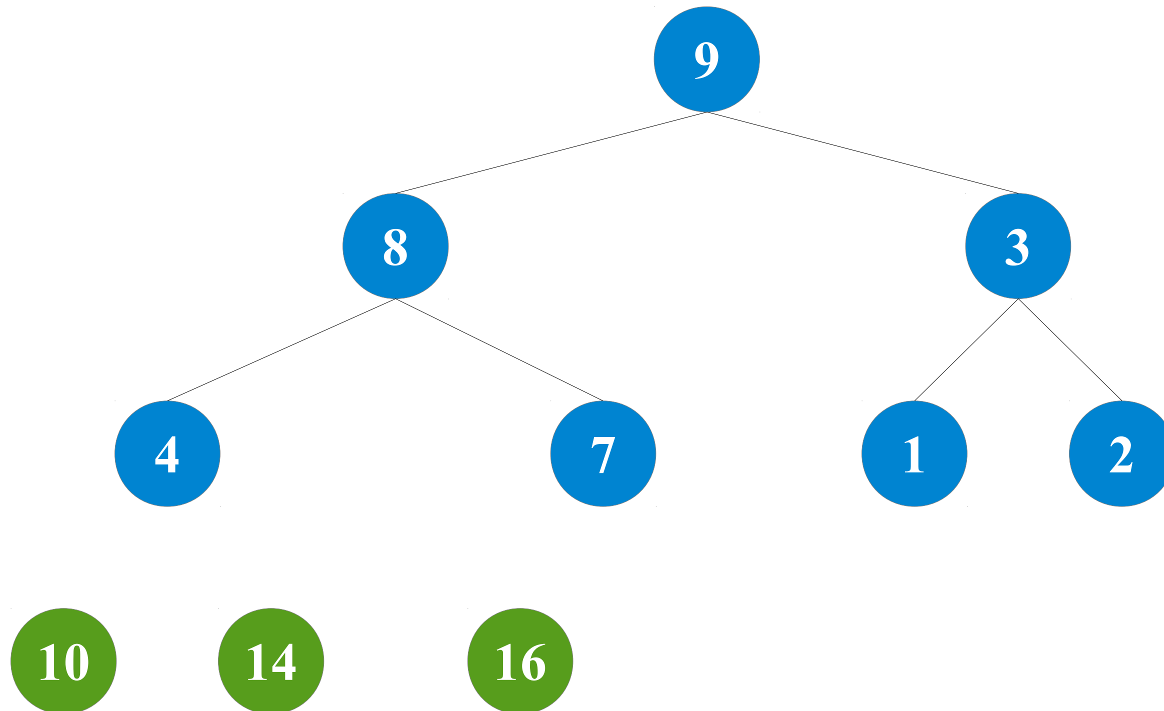
1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	3	2	14	16

Heapsort



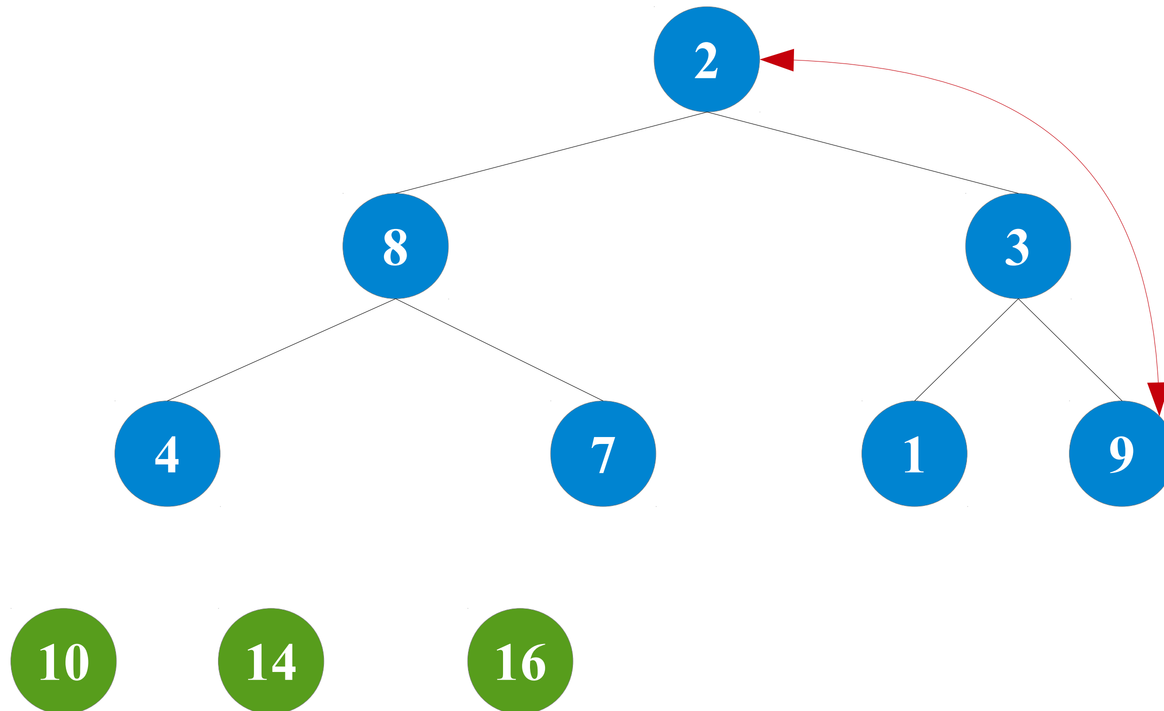
1	2	3	4	5	6	7	8	9	10
2	**8**	**9**	**4**	**7**	**1**	**3**	**10**	**14**	**16**

Heapsort



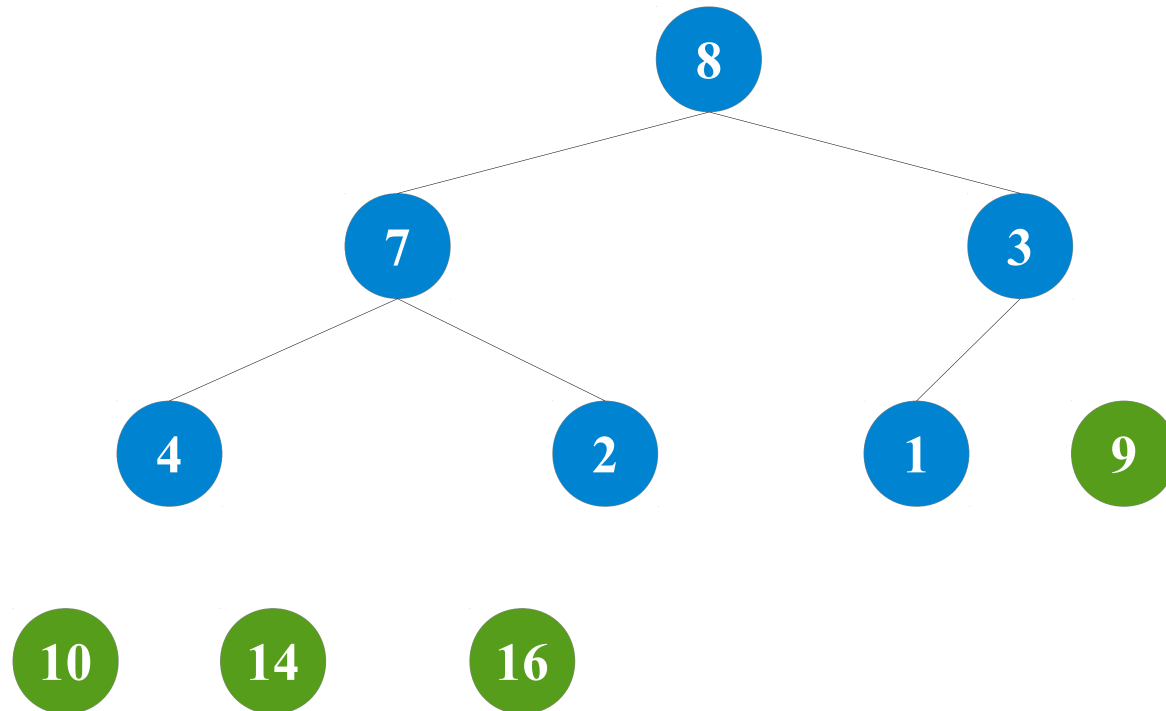
1	2	3	4	5	6	7	8	9	10
9	8	3	4	7	1	2	10	14	16

Heapsort



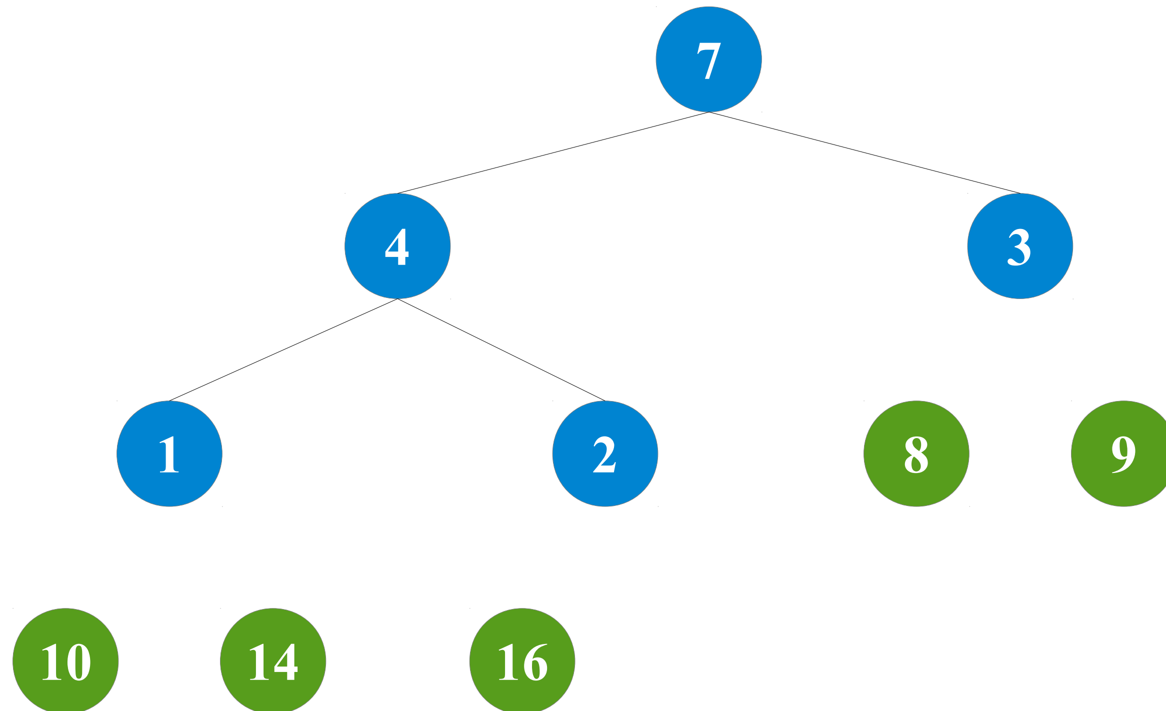
1	2	3	4	5	6	7	8	9	10
2	8	3	4	7	1	9	10	14	16

Heapsort



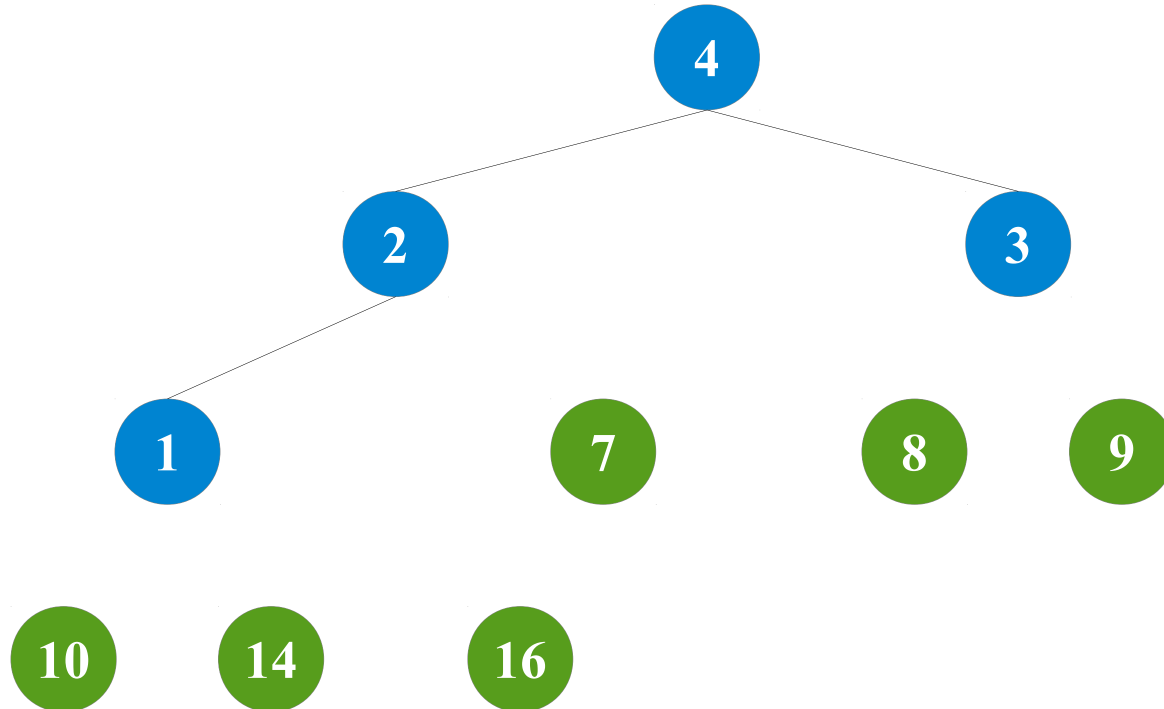
1	2	3	4	5	6	7	8	9	10
8	7	3	4	2	1	9	10	14	16

Heapsort



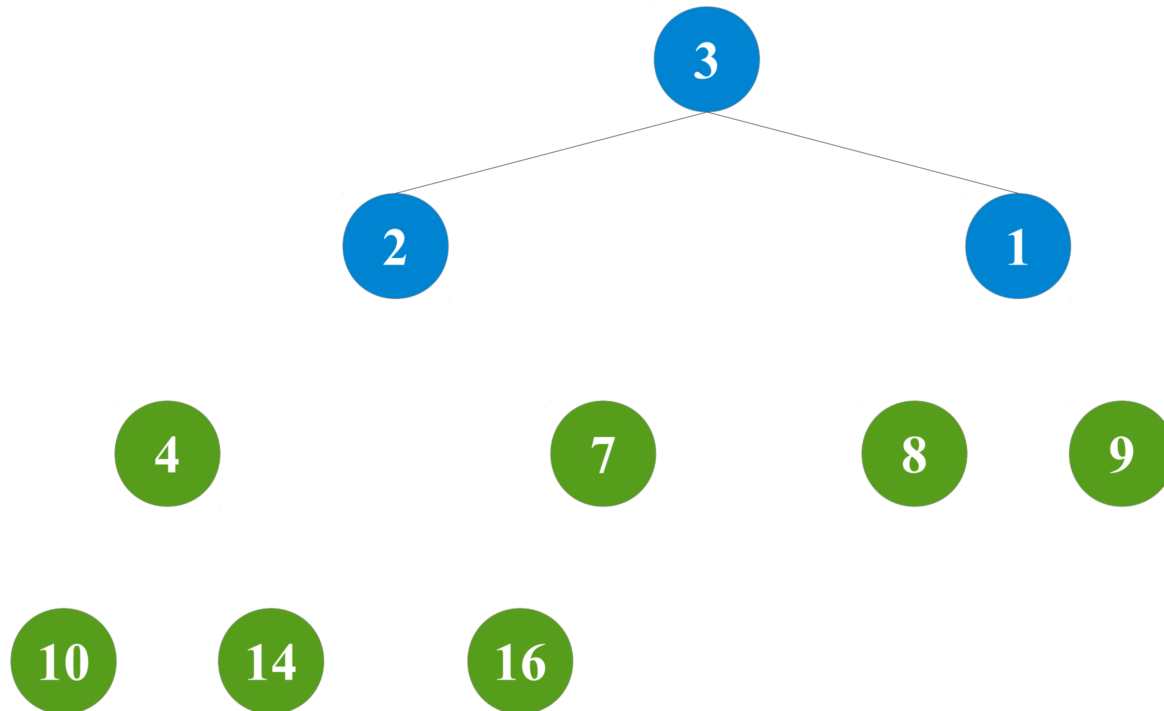
1	2	3	4	5	6	7	8	9	10
7	4	3	1	2	8	9	10	14	16

Heapsort



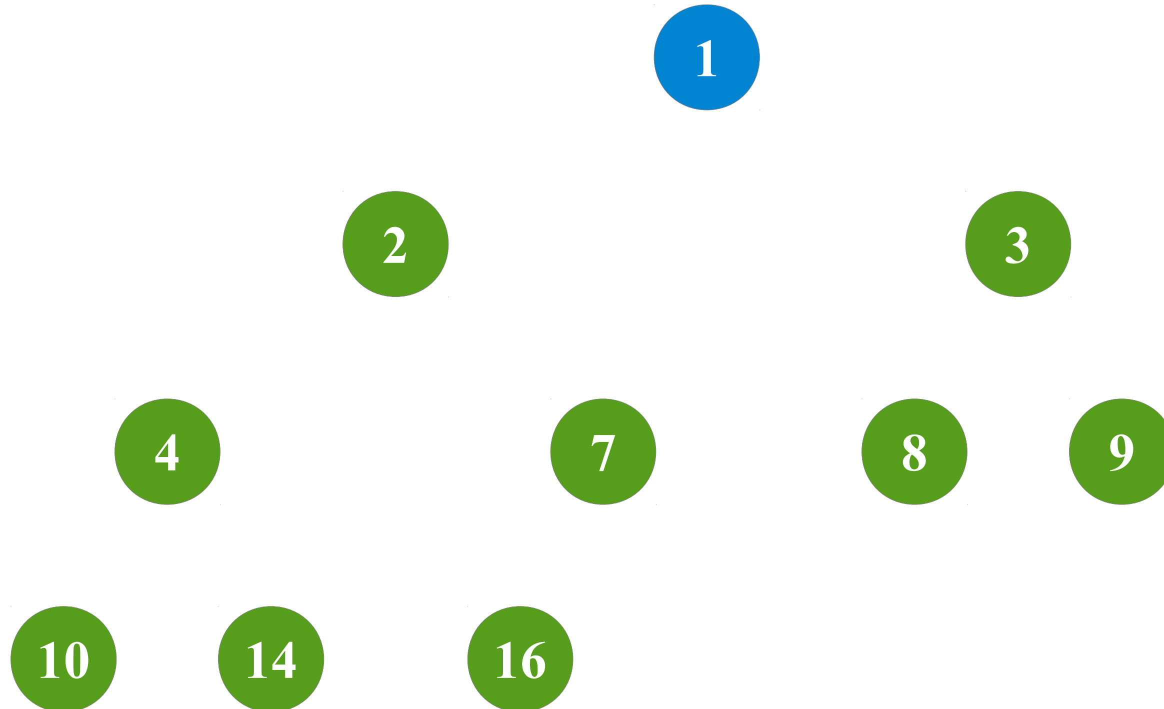
1	2	3	4	5	6	7	8	9	10
4	2	3	1	7	8	9	10	14	16

Heapsort



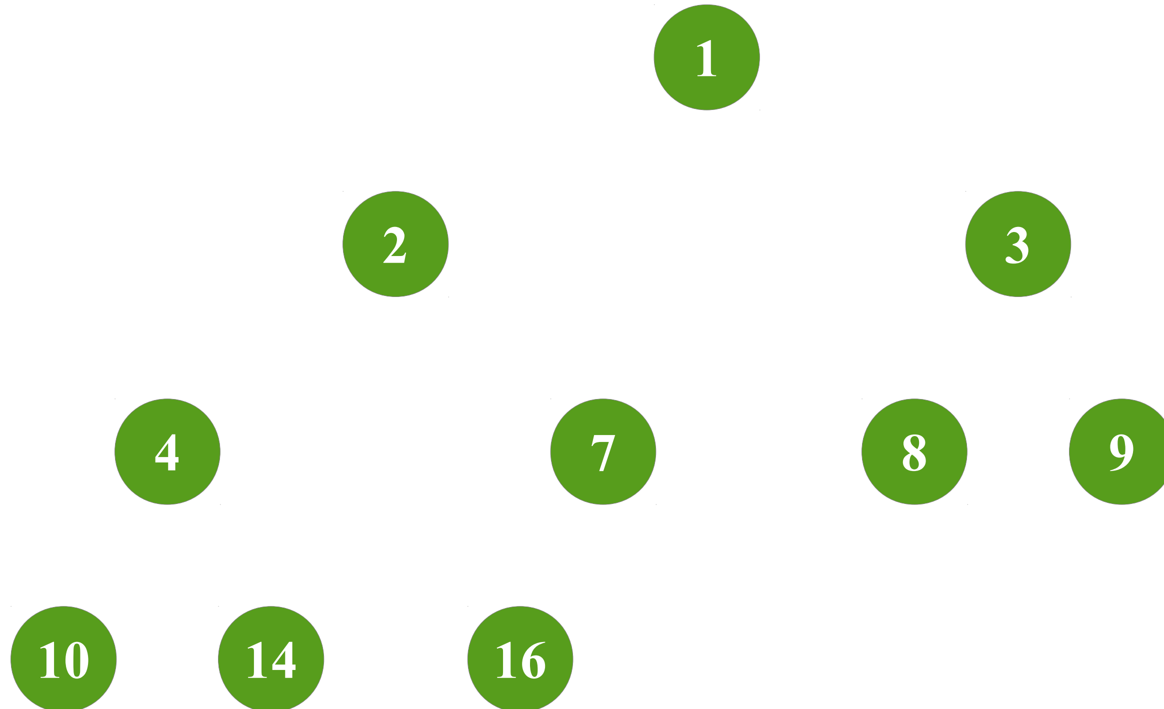
1	2	3	4	5	6	7	8	9	10
3	2	1	4	7	8	9	10	14	16

Heapsort



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

Heapsort



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

Heapsort Running time

- T(n) = $O(n)$ for Build-Max-Heap +
  n times $O(\lg n)$ for Max-Heapify

  = $O(n \lg n)$
- Space?
  - "in place" sorting $\Theta(1)$
- What was Mergesort and Insertion sort?

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 1
3      exchange A[i] with A[1]
4      A.heap-size --
5      MAX-HEAPIFY(A, i)
```

Priority Queues

- They are like FIFO (First-in First-out) Queues BUT depend on some priority value
- Element with highest priority goes out first, even if the last one inserted!
- Applications include:
  - Event-driven simulation
  - Job scheduler etc.
- How is it implemented?
- Read the book

Summary

- Heaps
  - Useful for building priority queues
- Heapsort
  - Sorts "in place" → $\Theta(1)$ space
  - Worst case time $\Theta(n \lg n)$

Recap

- Heaps
- Heapsort
- Next:
  - Quicksort