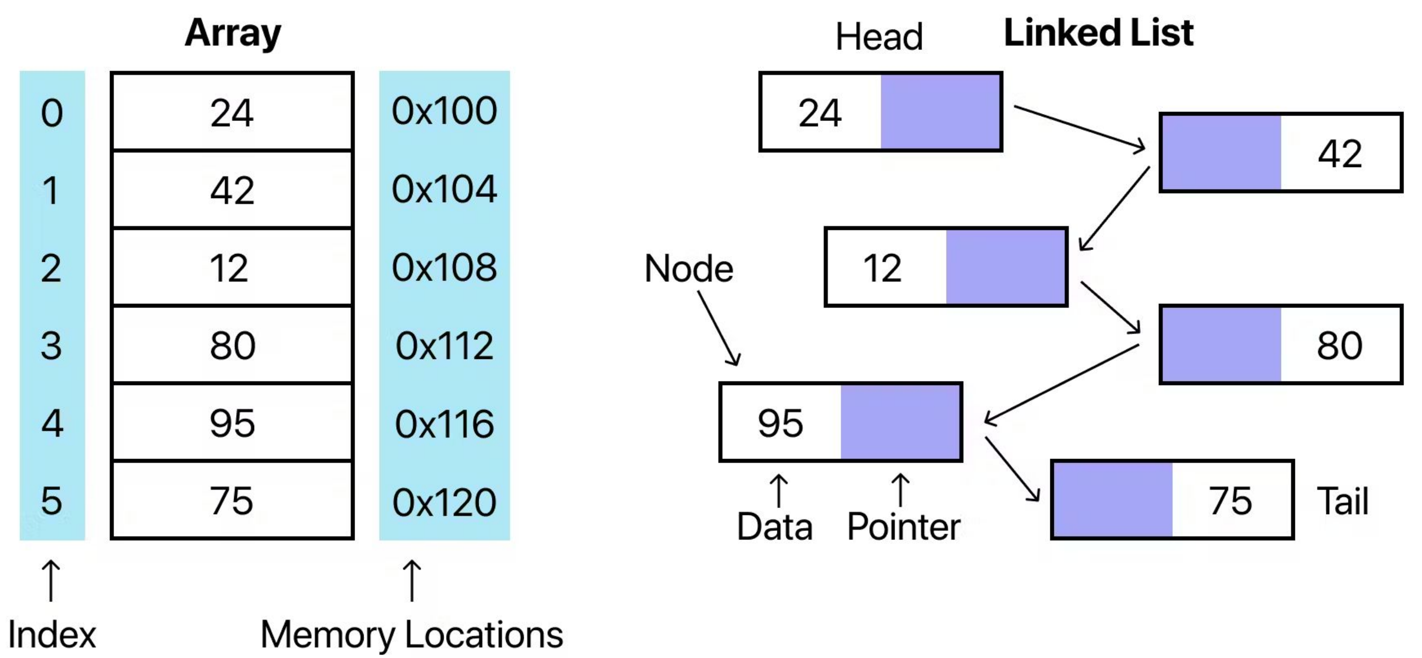
Heap (Priority Queue) & Heap Sort

CSE-215
Data Structure & Algorithm II

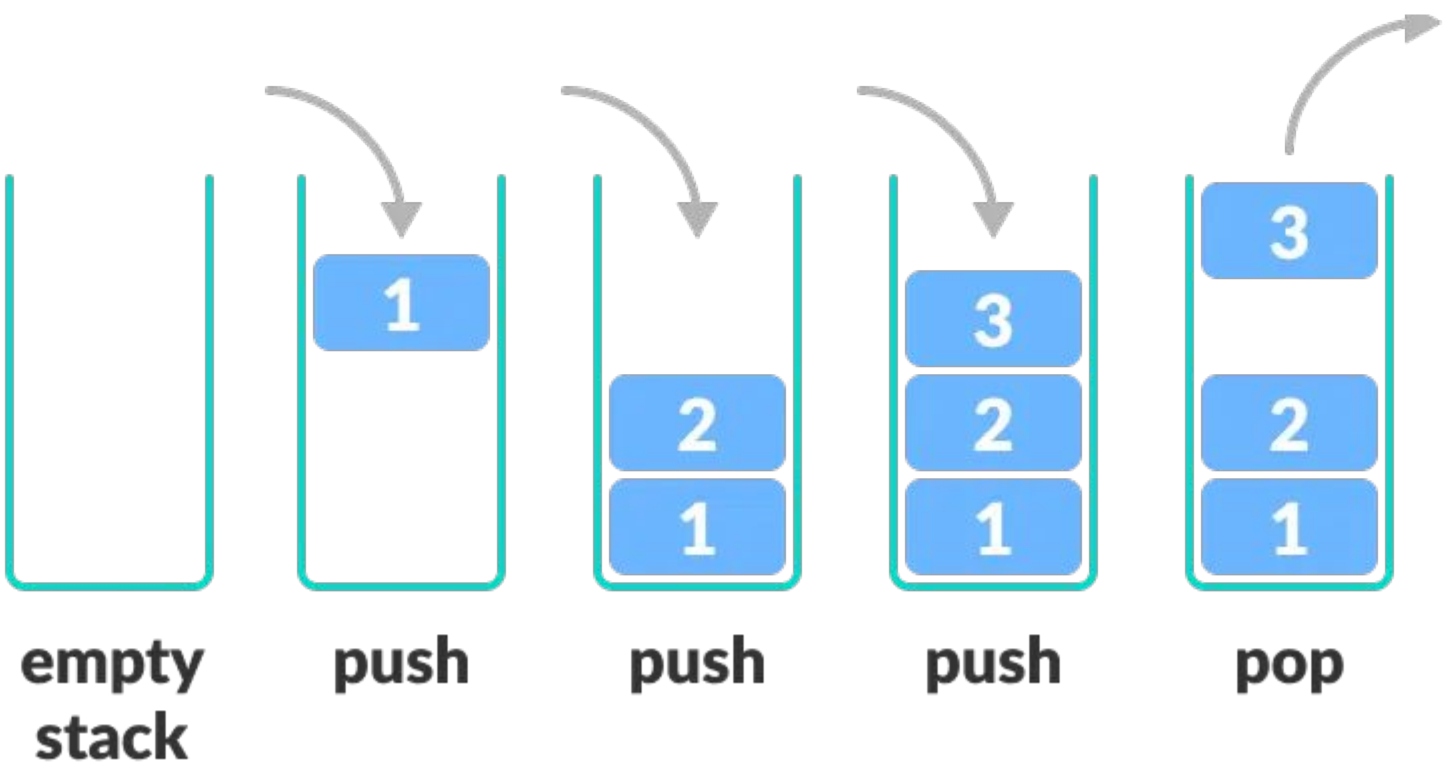
Linked List

- Linked list is a fundamental data structure in computer science. It mainly allows efficient insertion and deletion operations compared to arrays.



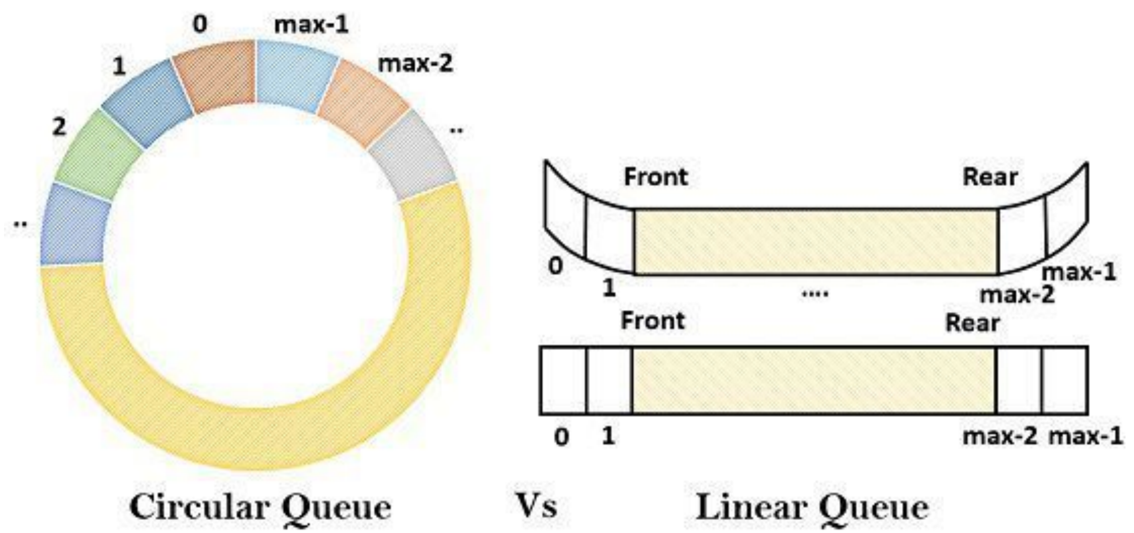
Stack

- Stack is a linear data structure that follows a particular order (LIFO or Last In First Out) in which the operations are performed.



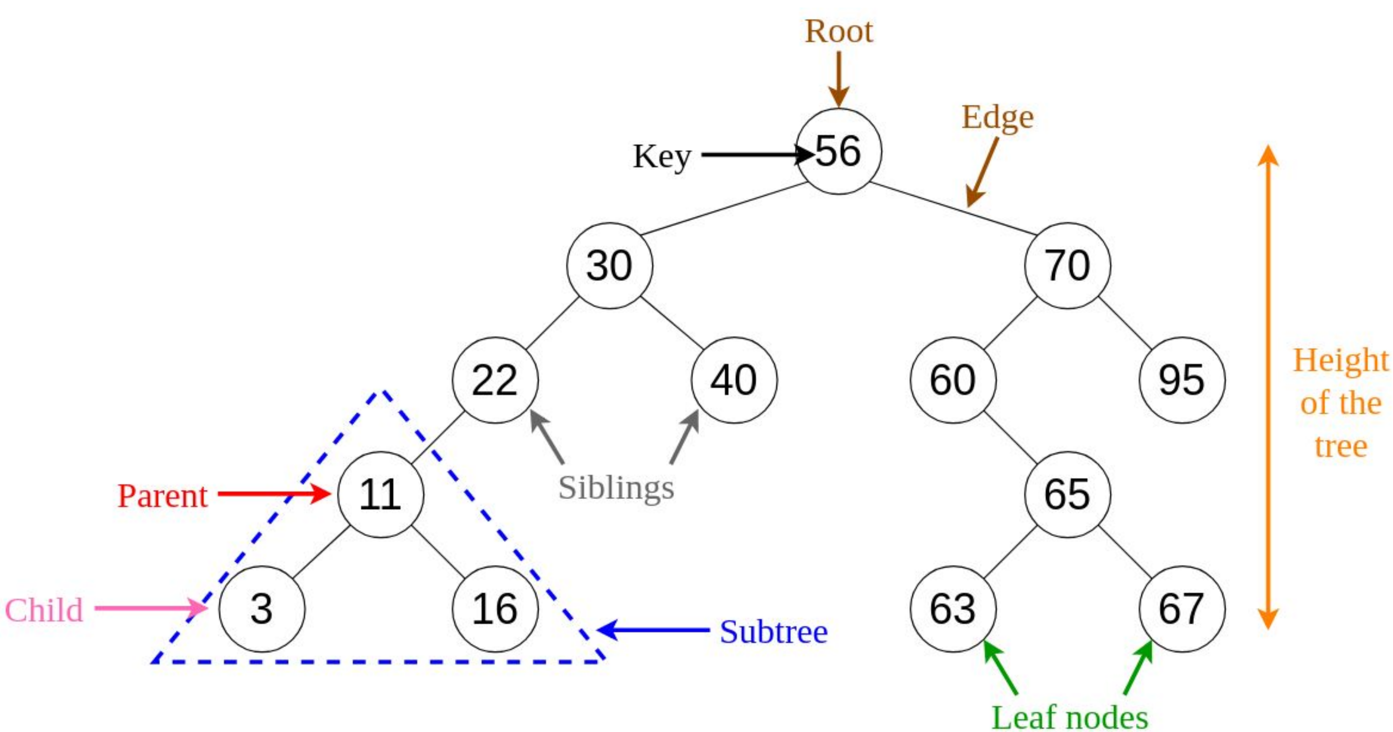
Queue

- **Queue is a linear data structure that follows a particular order (FIFO or Fast In First Out) in which the operations are performed.**



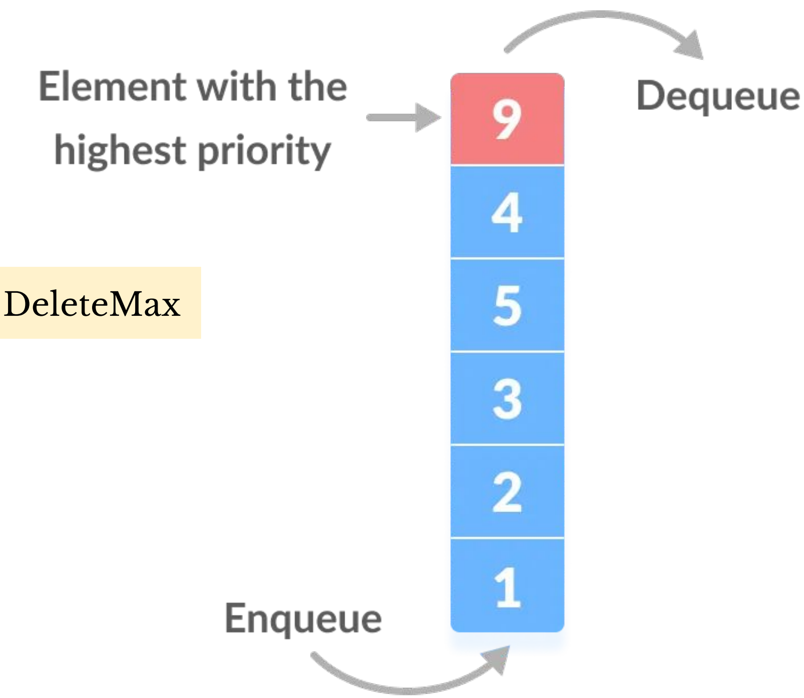
Binary Search Tree

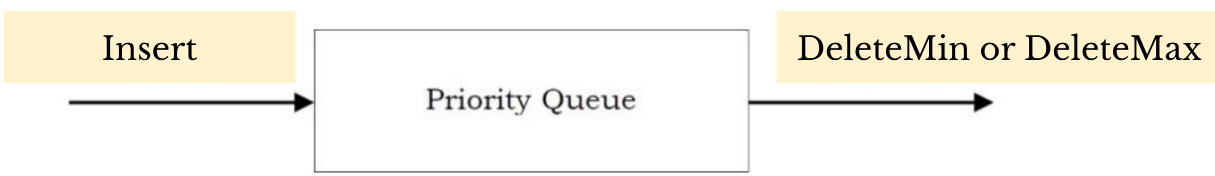
- Binary Search Tree is a binary tree that additionally satisfies the binary search property. The number of elements to compare decreases every time the search progresses.



Priority Queue

- Priority Queue is a data structure that supports the operations Insert and DeleteMin (which returns and removes the minimum element) or DeleteMax (which returns and removes the maximum element).





Priority Queue: Main Operations

- A priority queue is a container of elements, each having an associated key.
  - Insert (key, data): Inserts data with key to the priority queue. Elements are ordered based on key.
  - DeleteMin/DeleteMax: Remove and return the element with the smallest/largest key.
  - GetMinimum/GetMaximum: Return the element with the smallest/largest key without deleting it.
  - Increase-Key (data, newkey): Increases the value of data's key to the newkey.

Comparing Implementations

- **Comparison based on the operations:**

Implementation	Insertion	Deletion (DeleteMin)	Find Min
Unordered array	1	n	n
Unordered list	1	n	n
Ordered array	n	1	1
Ordered list	n	1	1
Binary Search Trees	$logn$ (average)	$logn$ (average)	$logn$ (average)
Balanced Binary Search Trees	$logn$	$logn$	$logn$

Comparing Implementations

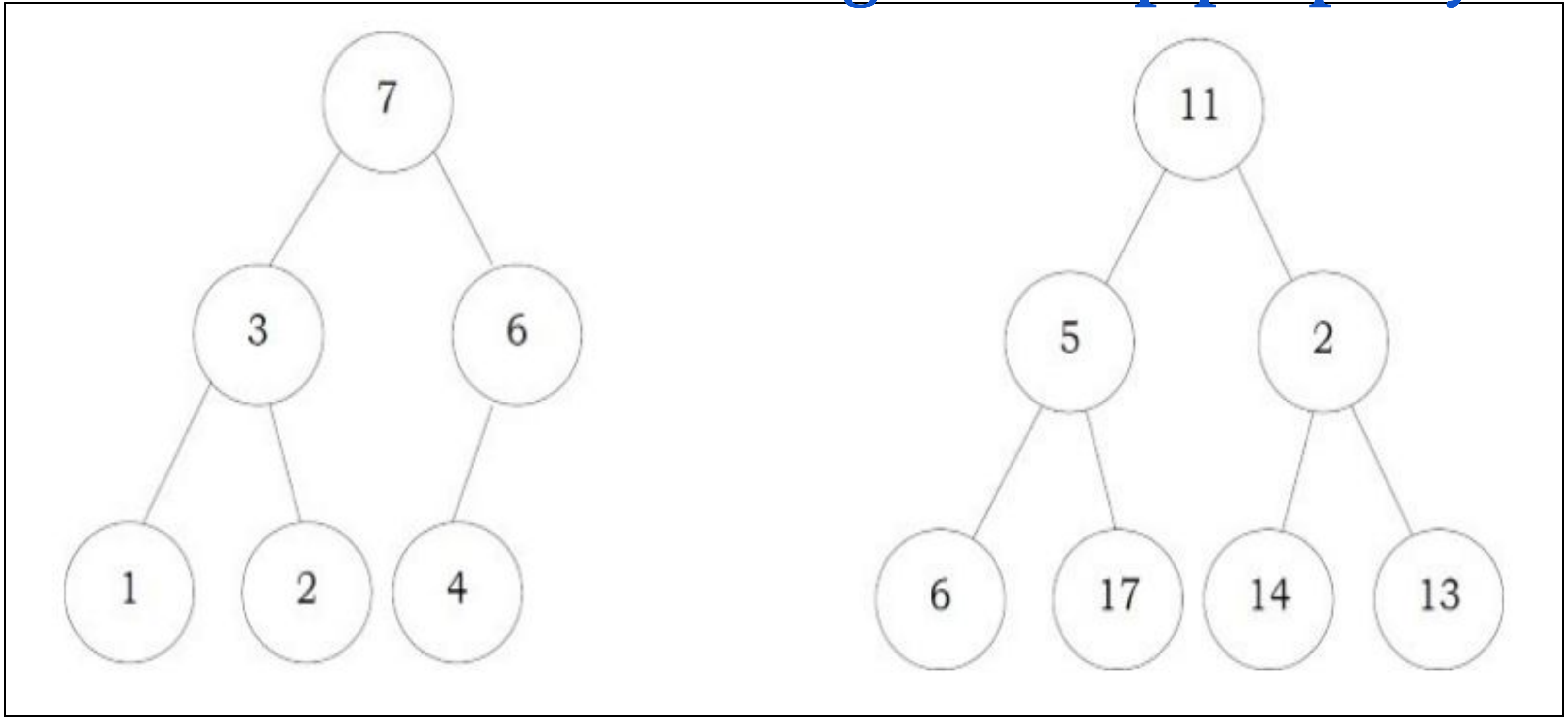
- **Comparison based on the operations:**

Implementation	Insertion	Deletion (DeleteMin)	Find Min
Unordered array	1	n	n
Unordered list	1	n	n
Ordered array	n	1	1
Ordered list	n	1	1
Binary Search Trees	$logn$ (average)	$logn$ (average)	$logn$ (average)
Balanced Binary Search Trees	$logn$	$logn$	$logn$
Binary Heaps	$logn$	$logn$	1

Heaps and Binary Heaps

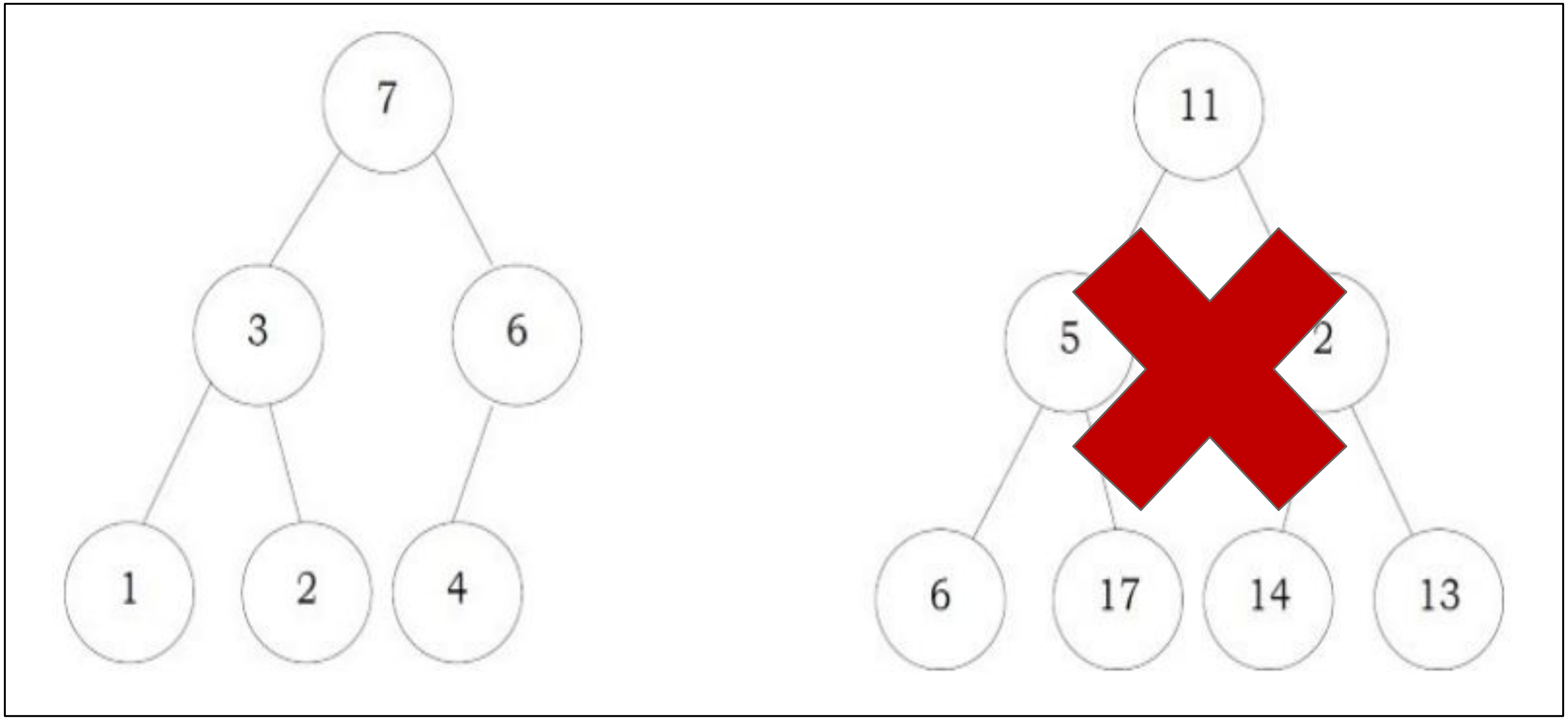
- <u>Heap Property:</u> The basic requirement of a heap is that the value of a node must be ≥ (or ≤) than the values of its children.

Which one is following the heap property?



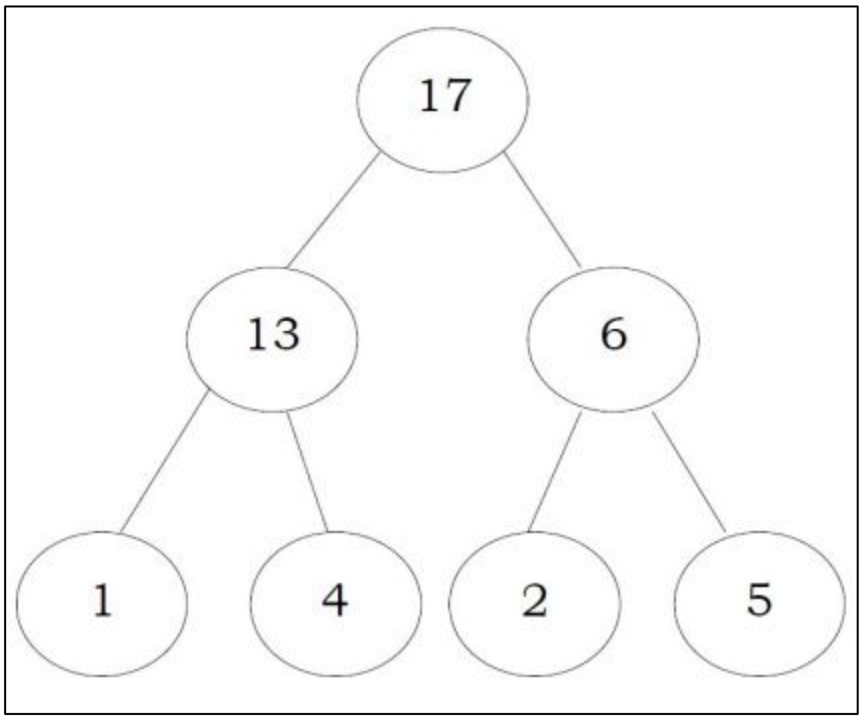
Heaps and Binary Heaps

- **<u>Heap Property:</u> The basic requirement of a heap is that the value of a node must be ≥ (or ≤) than the values of its children.**

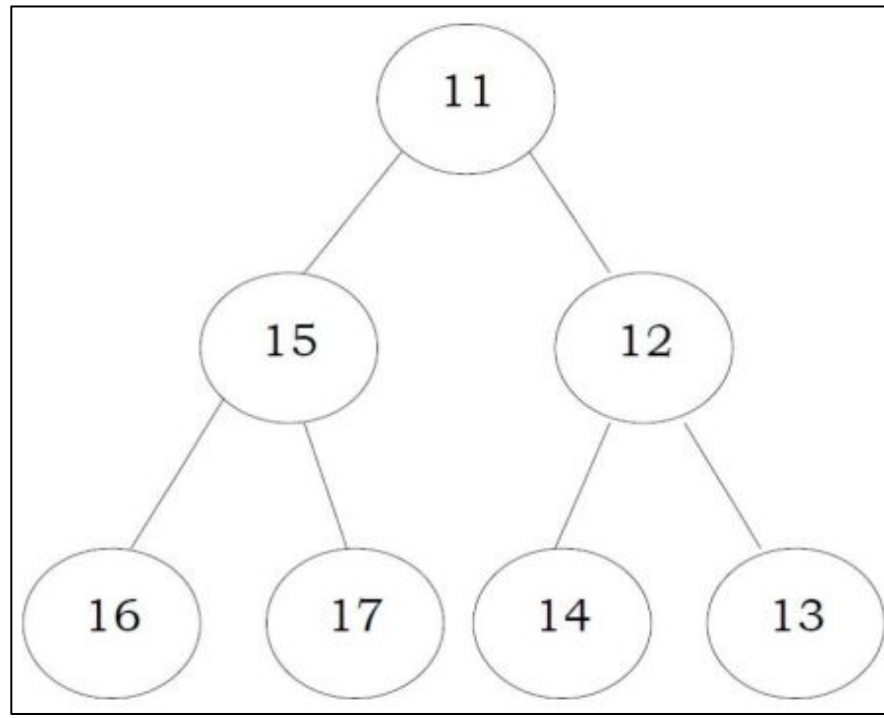


Heaps and Binary Heaps

- **Types of Heaps:**
  - **Min heap:** The value of a node must be less than or equal to the values of its children
  - **Max heap:** The value of a node must be greater than or equal to the values of its children



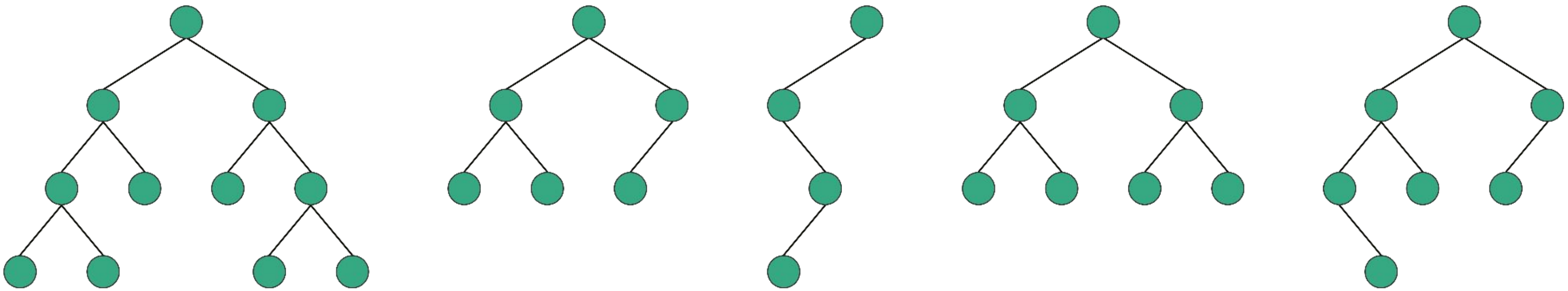
Max Heap



Min Heap

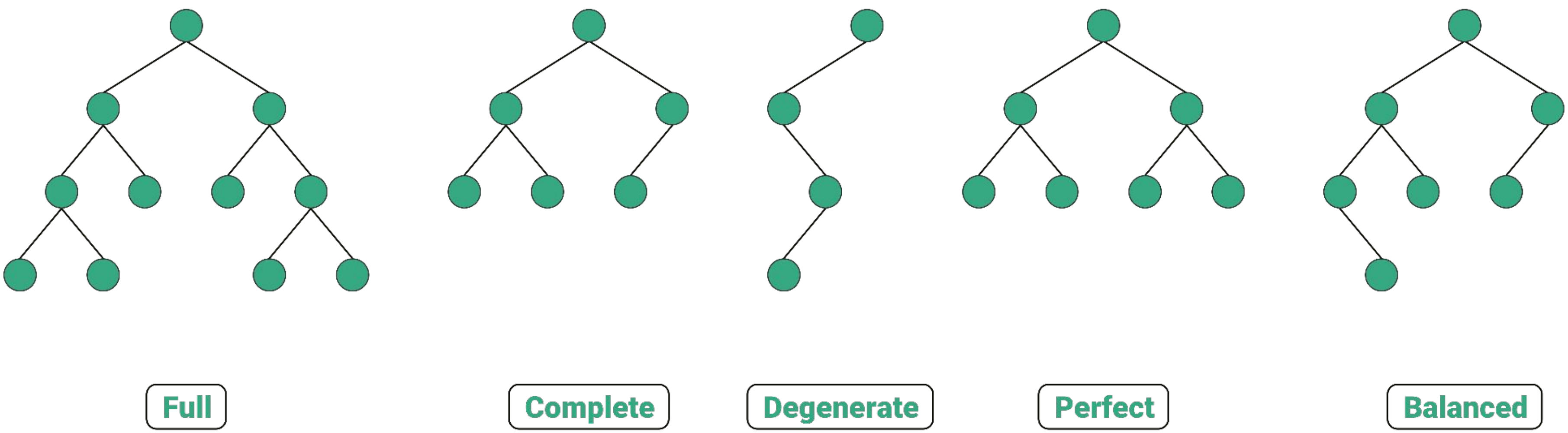
Heaps and Binary Heaps

- <u>**Complete Binary Tree:**</u> **Heap has the additional property that all leaves should be at h or $h - 1$ levels (where h is the height of the tree).**



Heaps and Binary Heaps

- **Complete Binary Tree:** Heap has the additional property that all leaves should be at h or h – 1 levels (where h is the height of the tree).



*A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

Heaps and Binary Heaps

- Representing Heaps: One possibility is using arrays. Since heaps are forming complete binary trees, there will not be any wastage of locations.
- Why Heap can be represent using an Array but BST can not?
- To represent a complete binary tree as an array:
  - The root node is A[1]
  - The root stores the largest/smallest value (key)
  - Node i is A[i]
  - The parent of node i is A[i/2]
  - The left child of node i is A[2i]
  - The right child of node i is A[2i + 1]

Heaps and Binary Heaps

A =

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

Draw the tree (*Complete Binary Tree) where

```
Parent(i)
    return floor(i/2)
```

```
Left(i)
    return 2i
```

```
Right(i)
    return 2i+1
```

Heaps and Binary Heaps

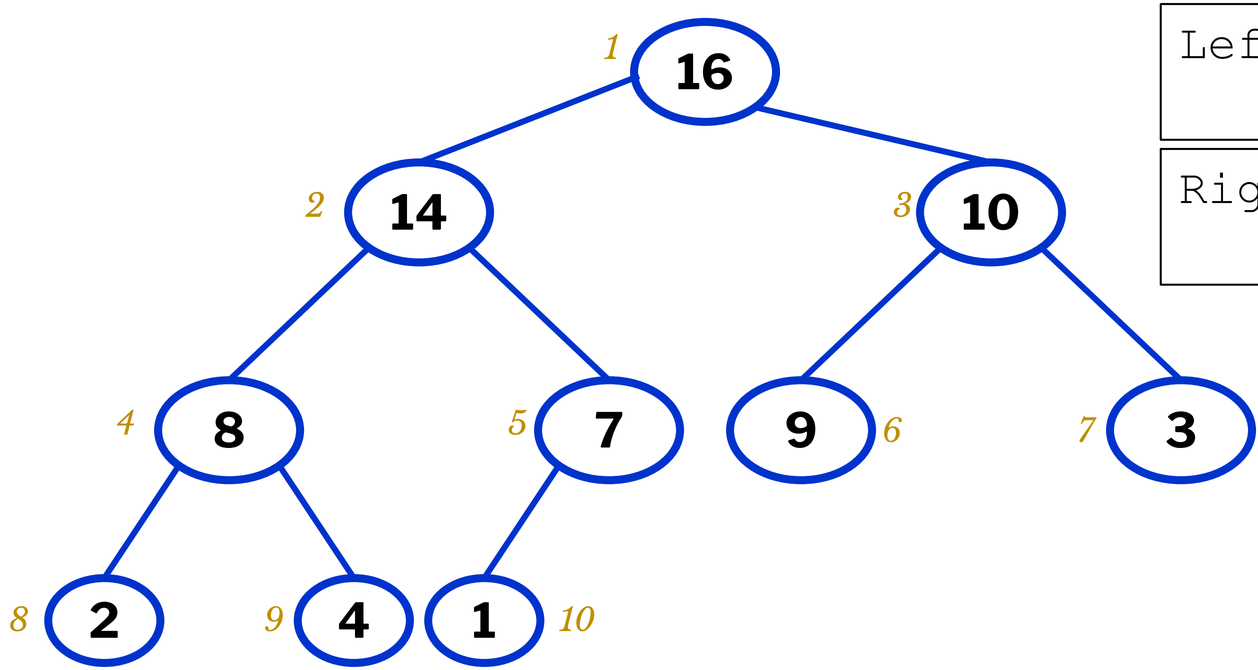
- **Represent the complete binary tree as an array:**

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

```
Parent(i)
    return floor(i/2)
```

```
Left(i)
    return 2i
```

```
Right(i)
    return 2i+1
```



Heaps and Binary Heaps

- Heap Property (Representing using an array)
  - Max-Heaps satisfy the heap property:

    $A[Parent(i)] \geq A[i]$ for all nodes $i > 1$

    - In other words, the value of a node is at most the value of its parent
    - The largest element is stored at the index 1 or root
  - Min-Heaps satisfy the heap property:

    $A[Parent(i)] \leq A[i]$ for all nodes $i > 1$

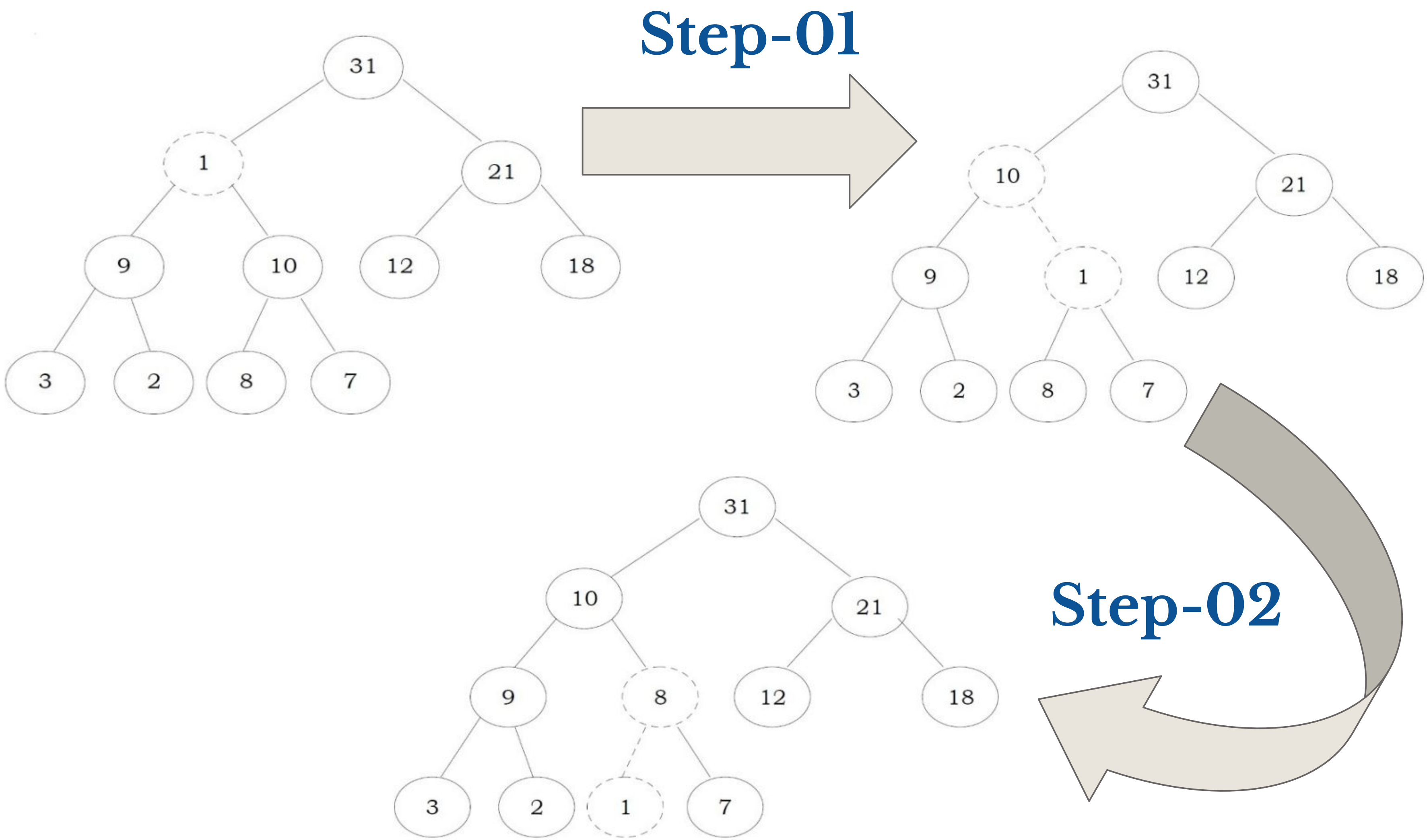
    - In other words, the value of a node is at least the value of its parent
    - The smallest element is stored at the index 1 or root

Binary Heaps: Heapifying

- Heapifying an Element (maintain the heap property)
  - After inserting an element into heap or deleting the root (minimum/ maximum) from heap, it may not satisfy the heap property.
  - In that case we need to adjust the locations of the heap to make it heap again. This process is called heapifying.
    - *PercolateDown*: Compare Parent and Children towards Leaf
    - *PercolateUp*: Compare Parent and Children towards Root
  - Time Complexity:
    - Height of the tree (*Complete Binary Tree) = O(logn)

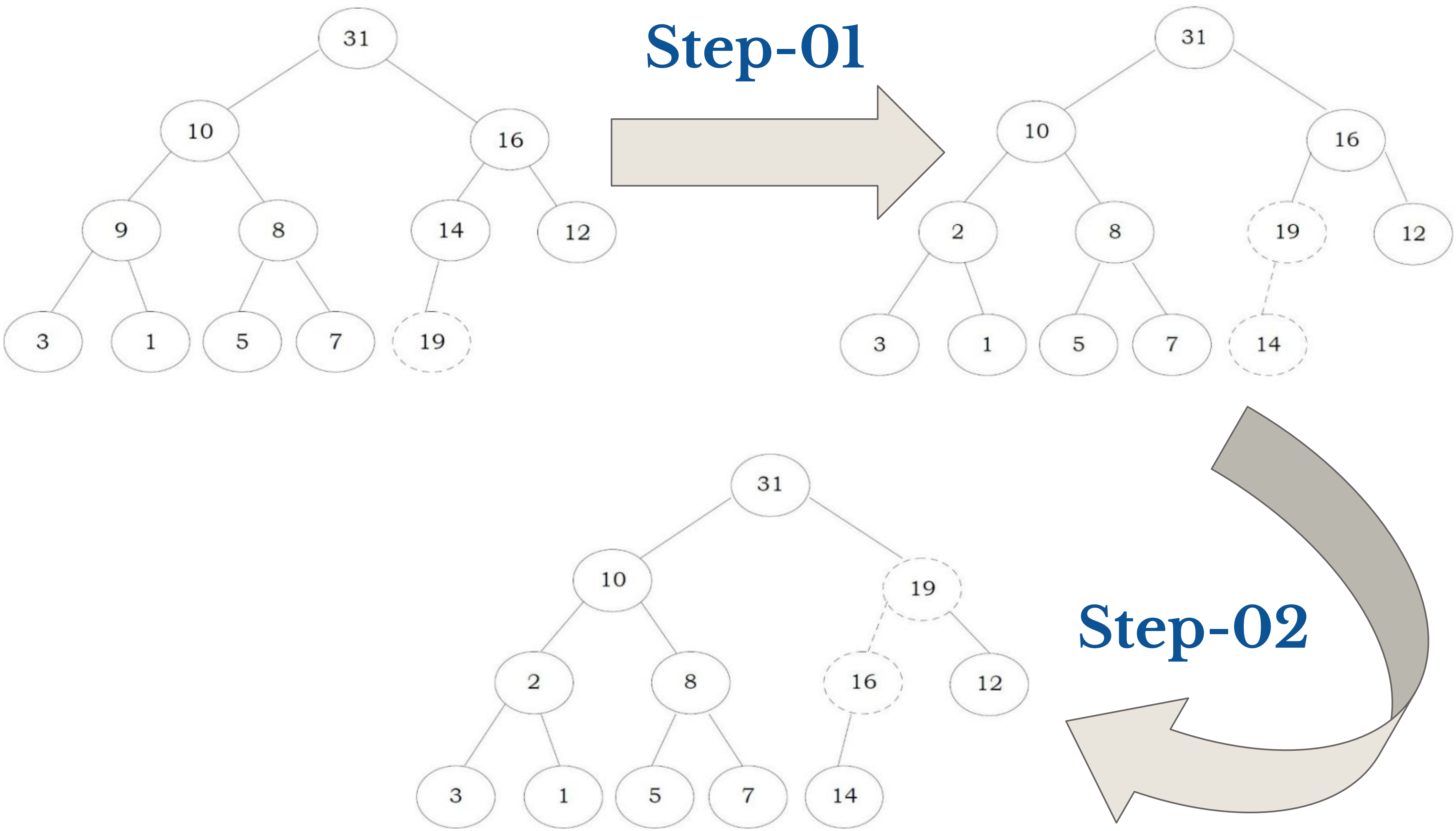
Binary Heaps: Max Heapifying

- *PercolateDown*



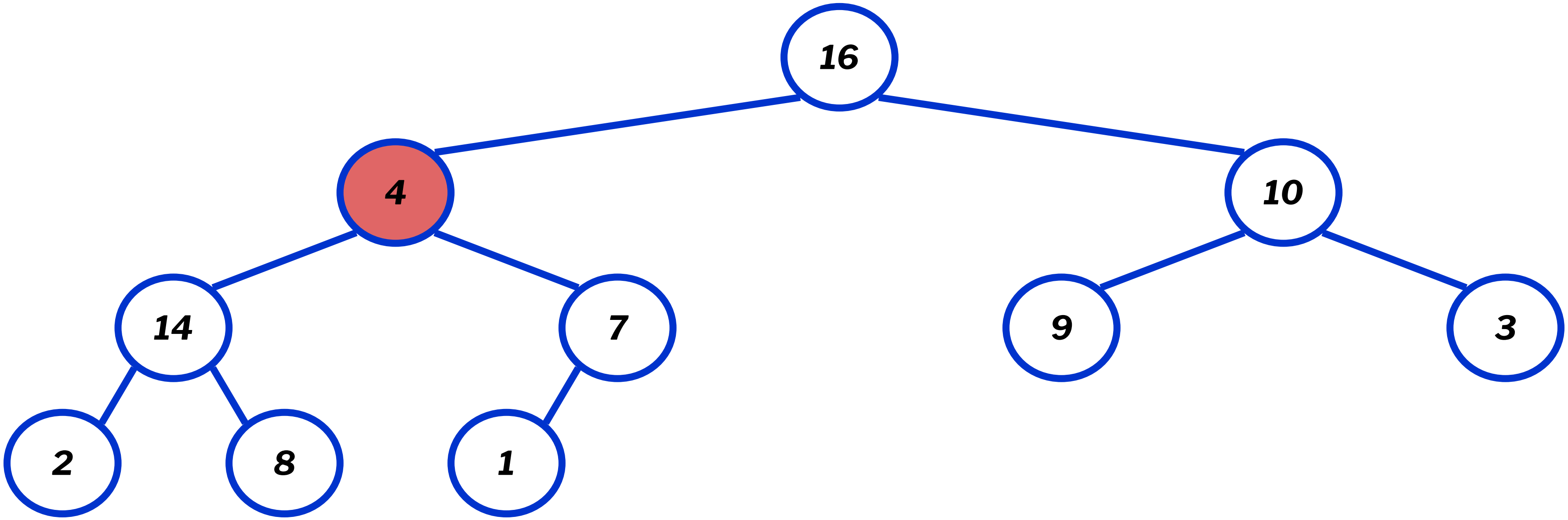
Binary Heaps: Max Heapifying

- *PercolateUp*



Binary Heaps: Max Heapifying

- *PercolateDown the value at index 2 (4)*

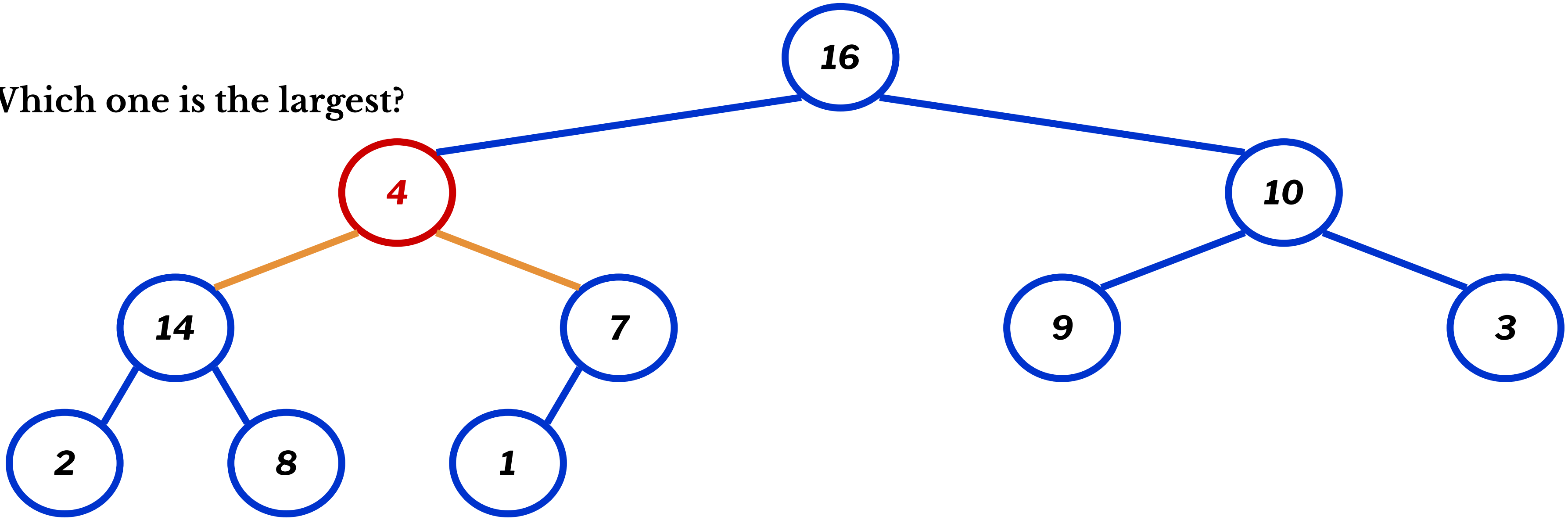


i	1	2	3	4	5	6	7	8	9	10
A =	16	4	10	14	7	9	3	2	8	1

Binary Heaps: Heapifying

- *PercolateDown*

Which one is the largest?



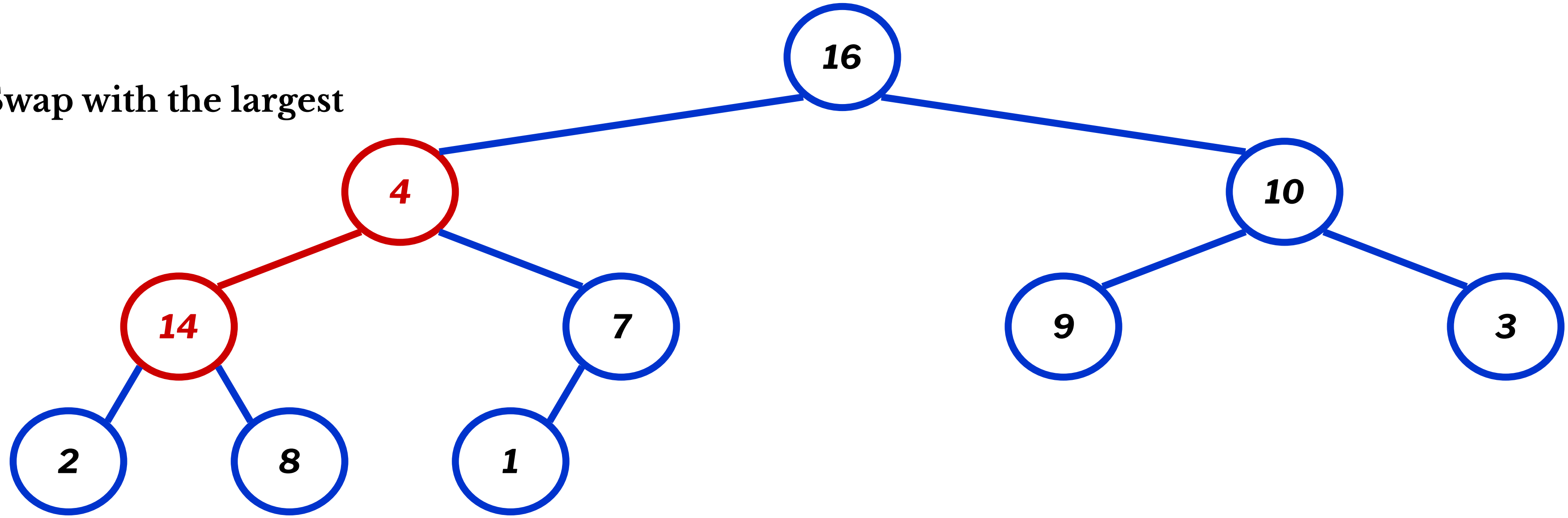
A =

16	4	10	14	7	9	3	2	8	1

Binary Heaps: Heapifying

- *PercolateDown*

Swap with the largest

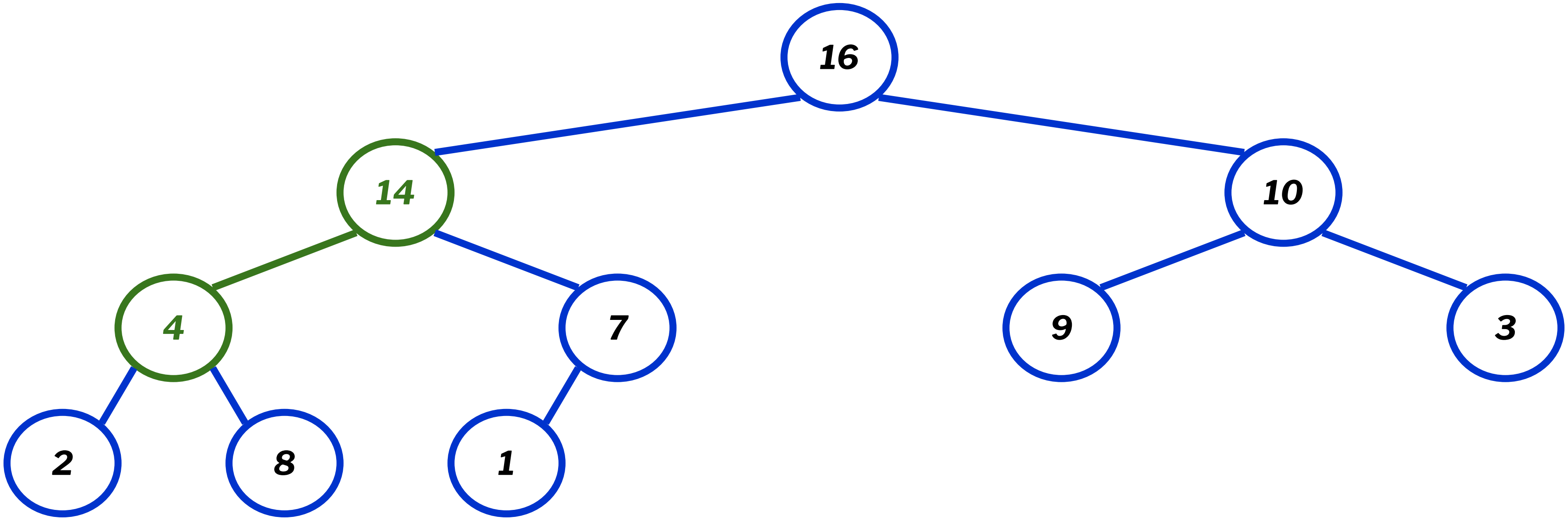


A =

16	4	10	14	7	9	3	2	8	1

Binary Heaps: Heapifying

- *PercolateDown*

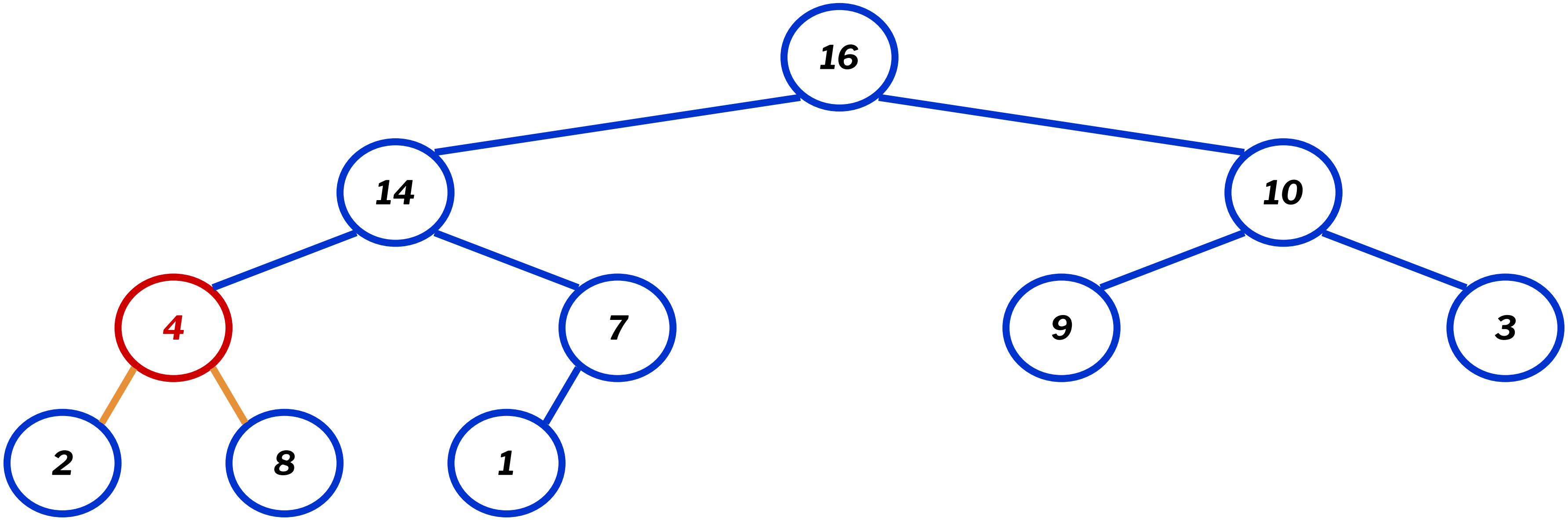


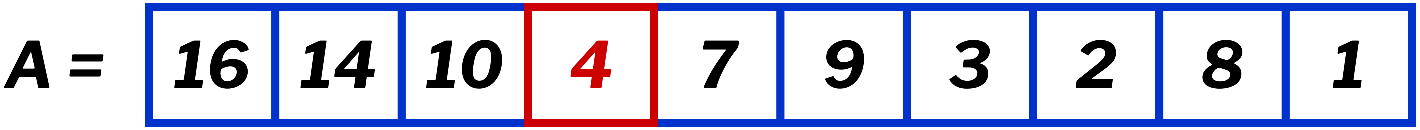
A =

16	14	10	4	7	9	3	2	8	1

Binary Heaps: Heapifying

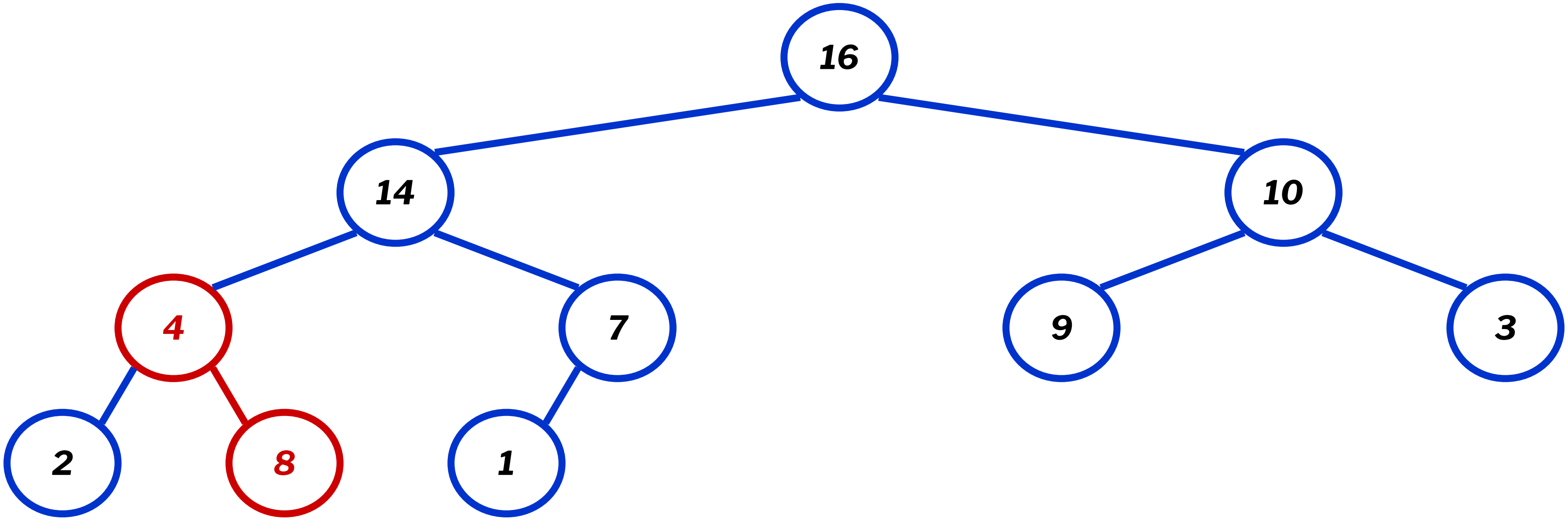
- *PercolateDown*





Binary Heaps: Heapifying

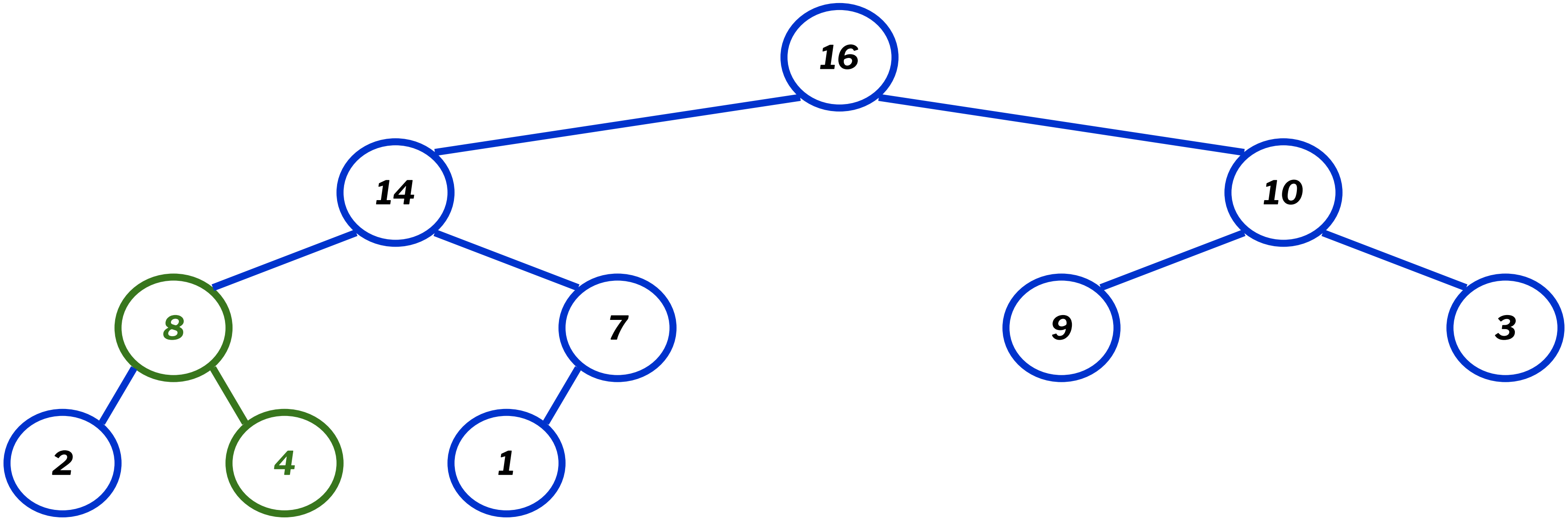
- *PercolateDown*



A =									
16	14	10	4	7	9	3	2	8	1

Binary Heaps: Heapifying

- *PercolateDown*

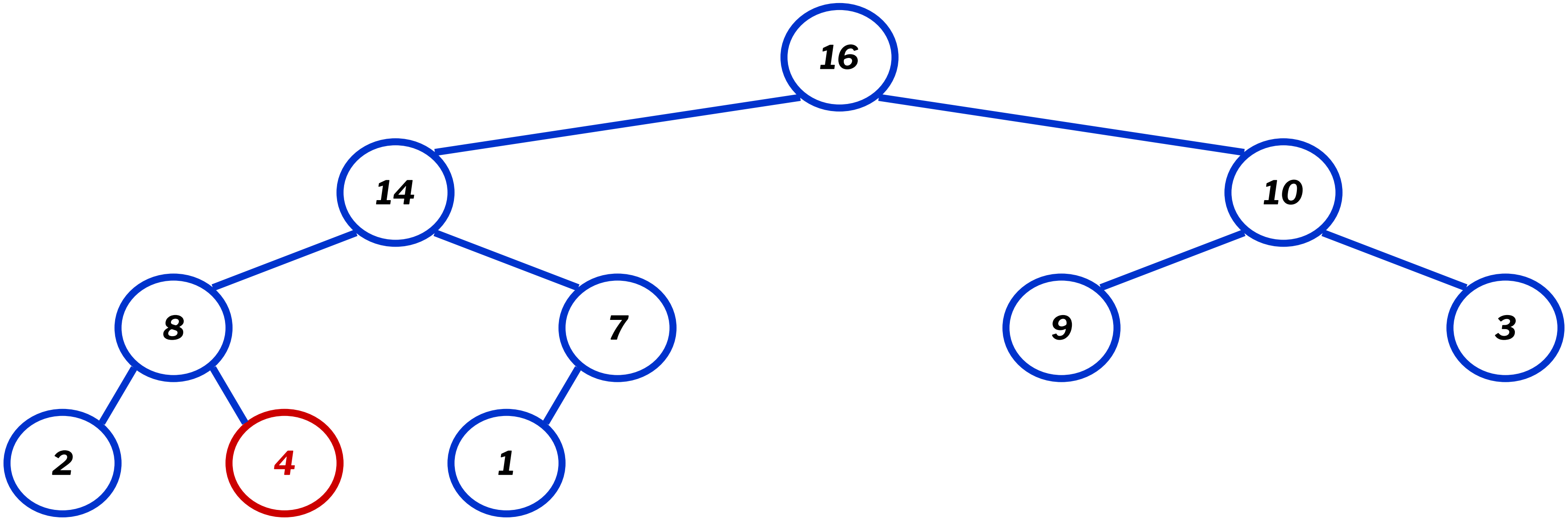


A =

16	14	10	8	7	9	3	2	4	1

Binary Heaps: Heapifying

- *PercolateDown*

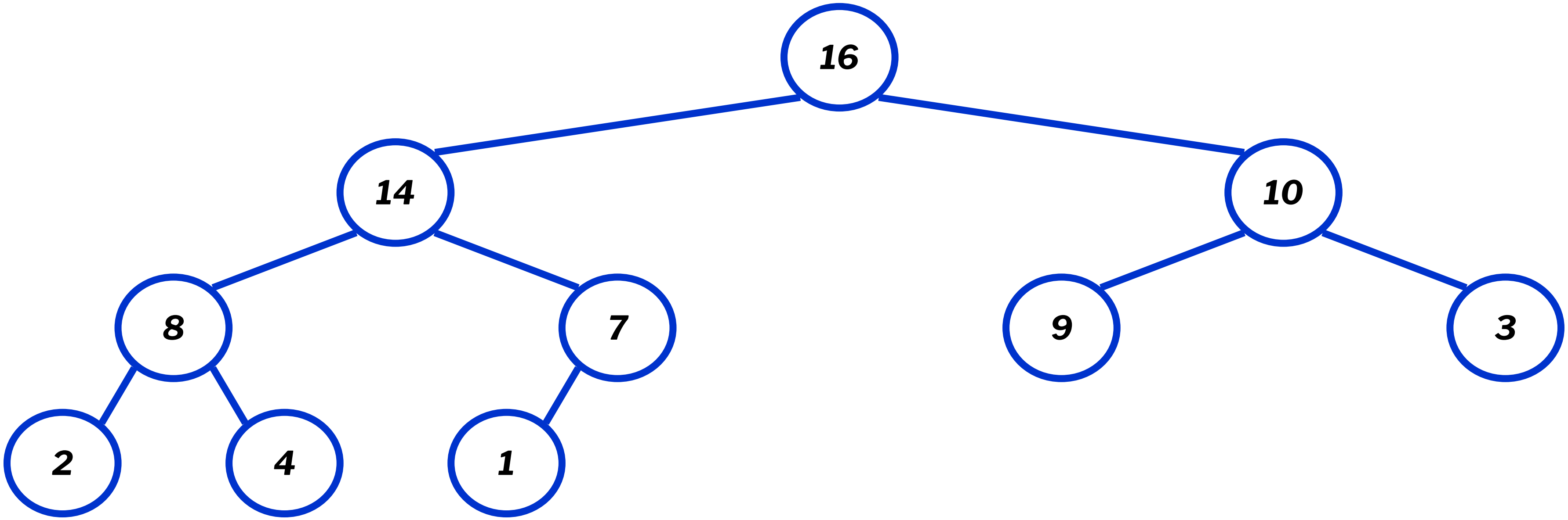


A =

16	14	10	8	7	9	3	2	4	1

Binary Heaps: Heapifying

- *PercolateDown*



A =

16	14	10	8	7	9	3	2	4	1

Binary Heaps: Heapifying

- ***PercolateDown***

Assume that the binary trees rooted at LEFT(i) and RIGHT(i) are already max-heaps.

```
PercolateDown (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]
              PercolateDown (A, largest)
```

Binary Heaps: DeleteMax/ DeleteMin

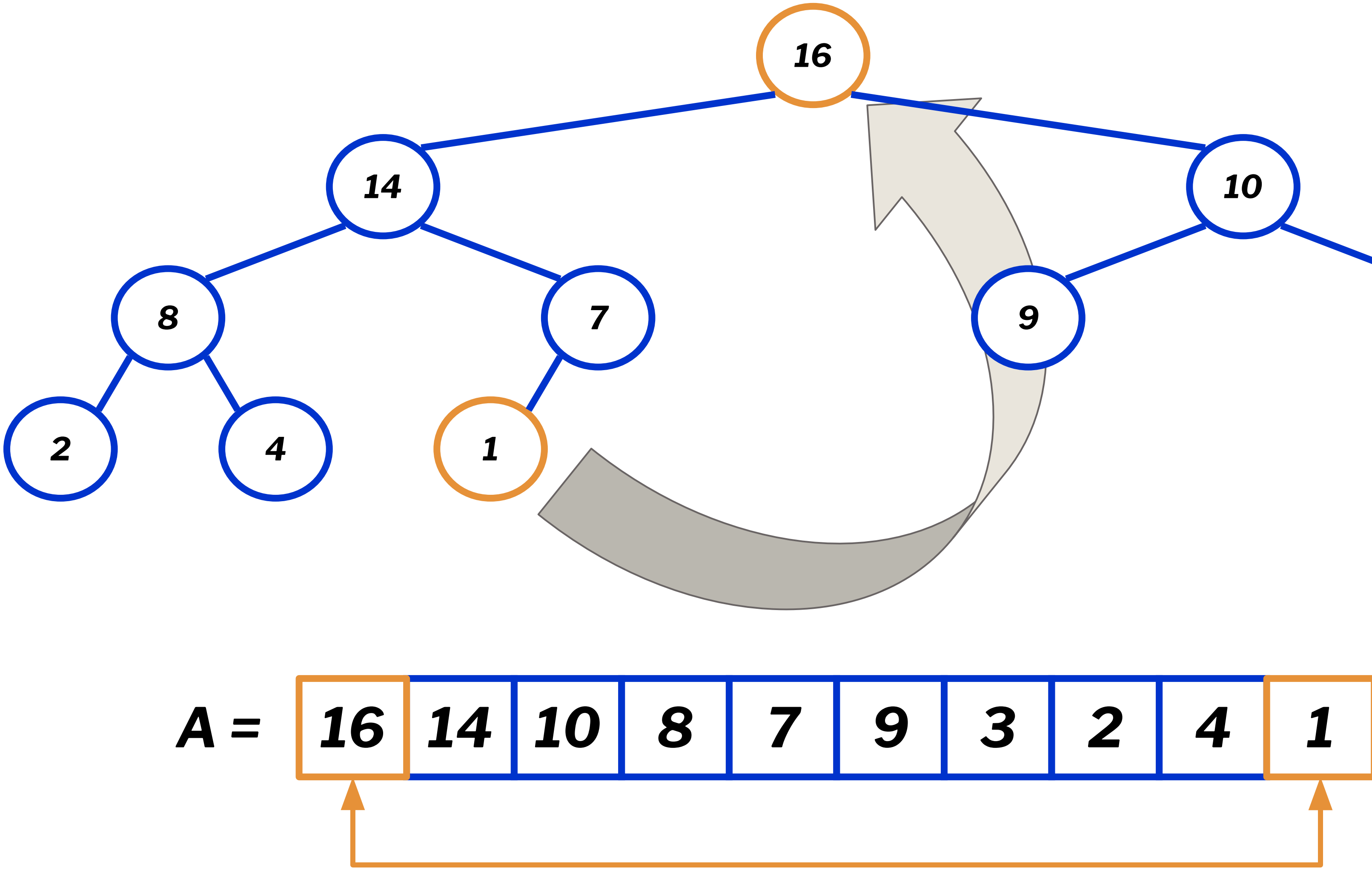
- Steps:
  - Copy the first element into some variable
  - Copy the last element into first element location
  - Reduce the heap size
  - *PercolateDown* the first element

```
DeleteMax (A)

if heap-size[A] < 1
    then error "heap underflow"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] − 1
PercolateDown (A, 1)
return max
```

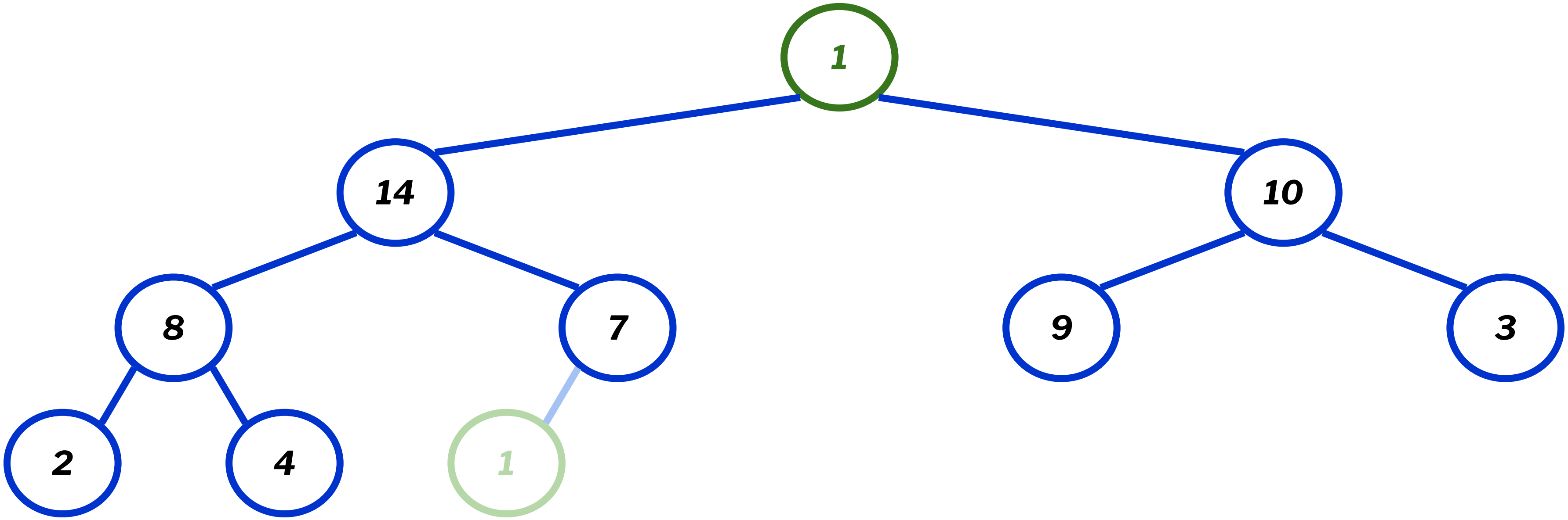
Binary Heaps: DeleteMax

- Copy the last element into first element location



Binary Heaps: DeleteMax

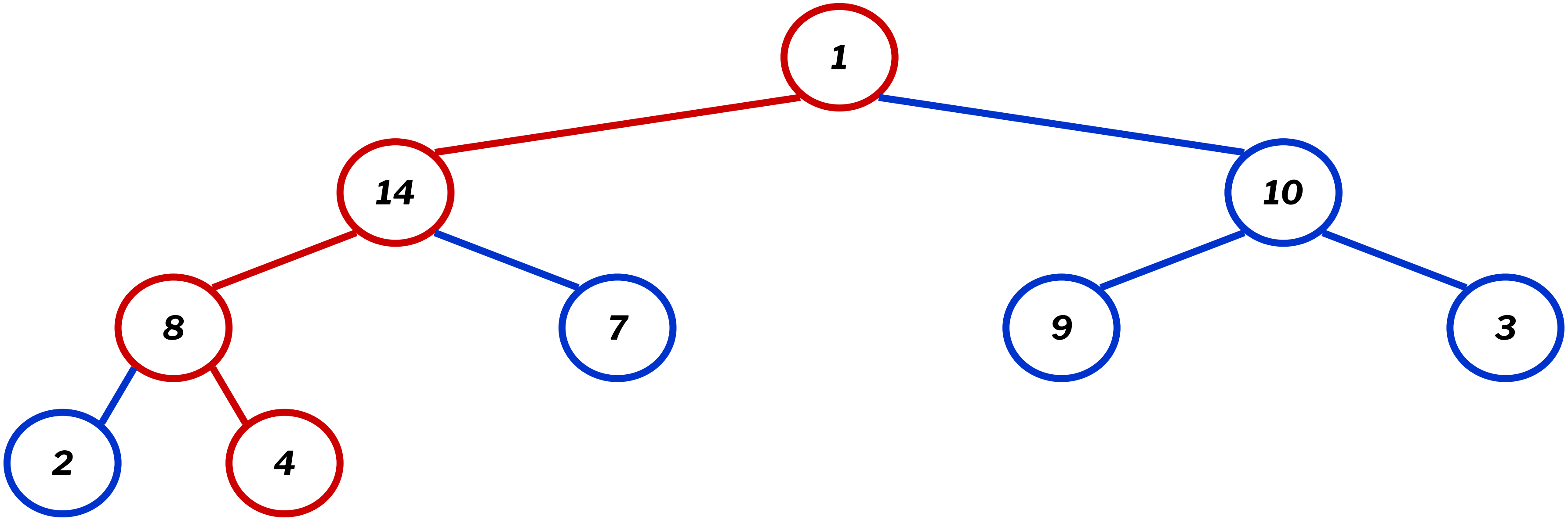
- Reduce the heap size



A =	1	14	10	8	7	9	3	2	4	1

Binary Heaps: DeleteMax

- *PercolateDown* the first element

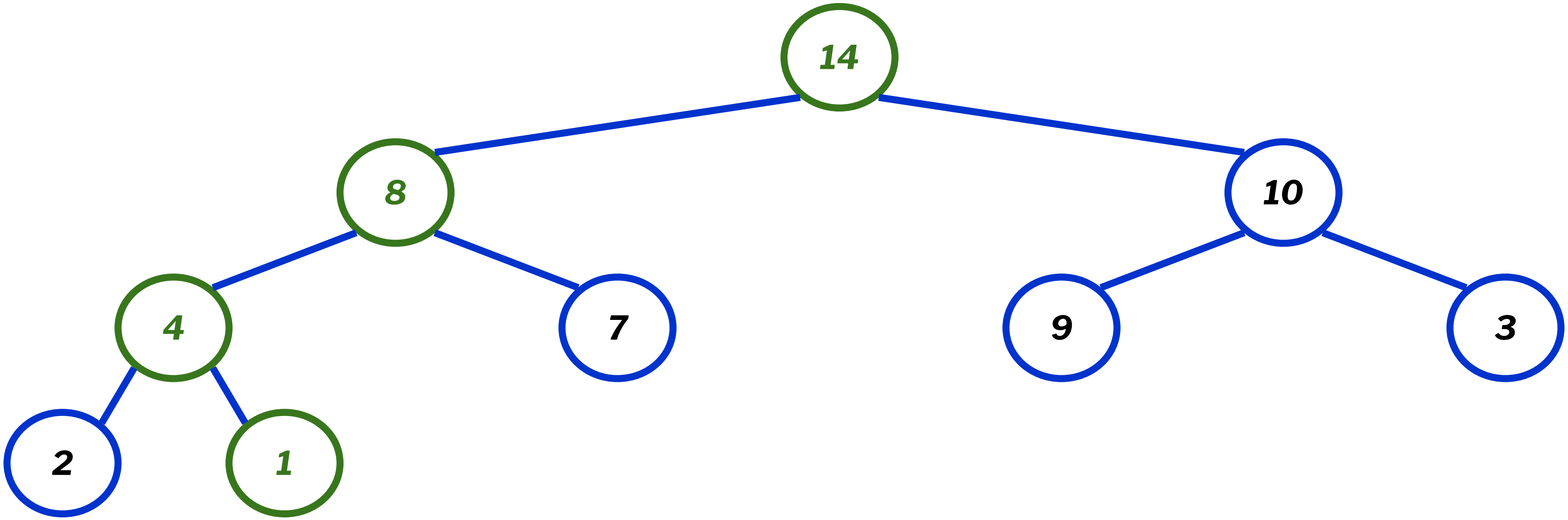


A =

1	14	10	8	7	9	3	2	4

Binary Heaps: DeleteMax

- Satisfy the Heap property



A =

14	8	10	4	7	9	3	2	1

Binary Heaps: Increase Key

- Steps:
  - Update the value/ key
  - *PercolateUp* the first element

```
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)
```

Percolate Up

Binary Heaps: Increase Key

- ***Increase Key***

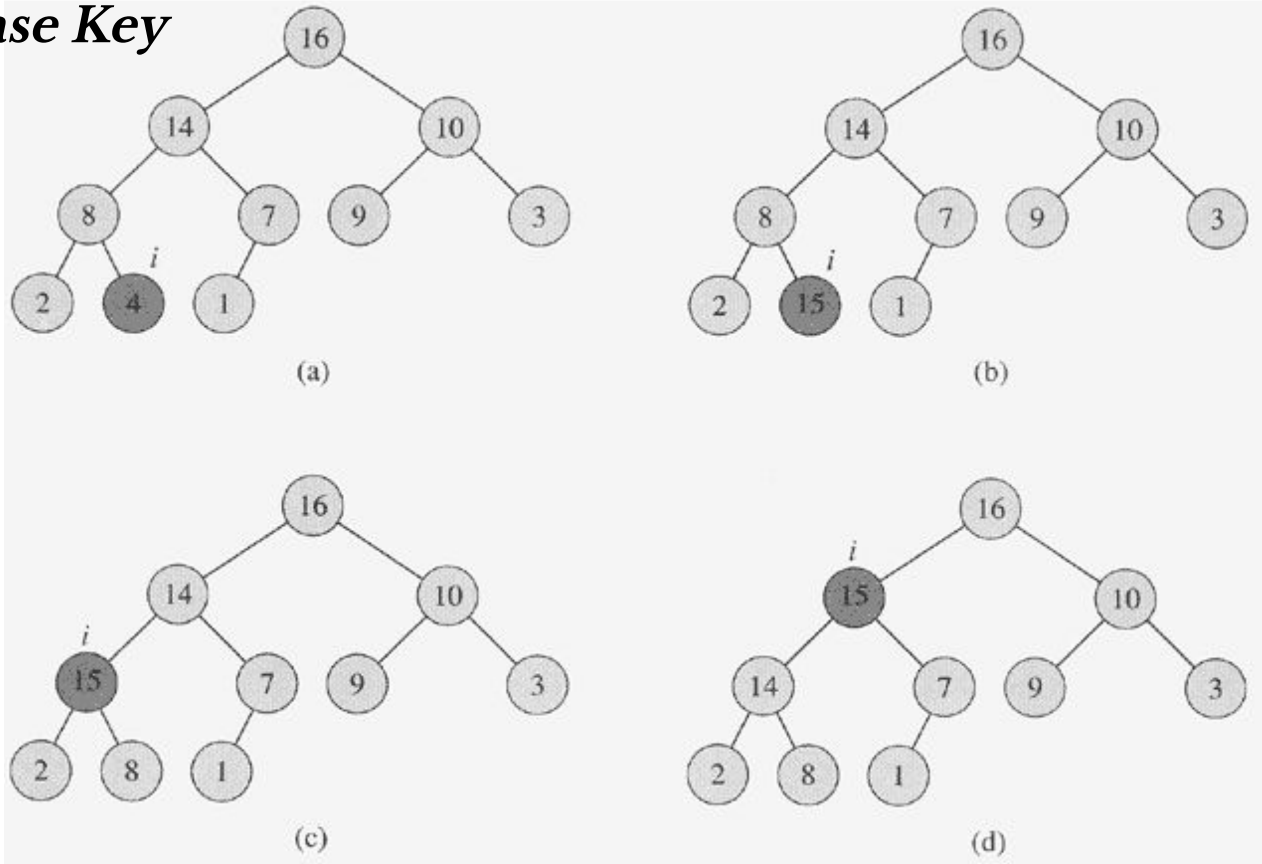


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

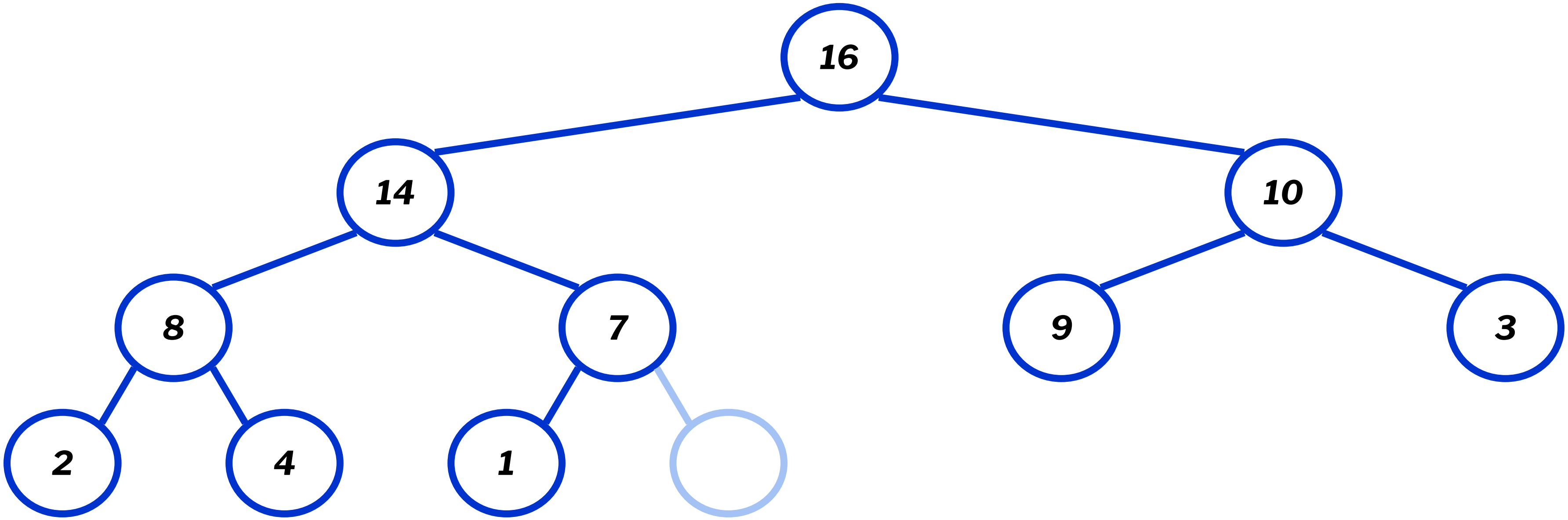
Binary Heaps: Insert

- Steps:
  - Increase the heap size
  - Keep the new element at the end of the heap (tree)
  - *PercolateUp* the new element from bottom to top (root)

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

Binary Heaps: Insert

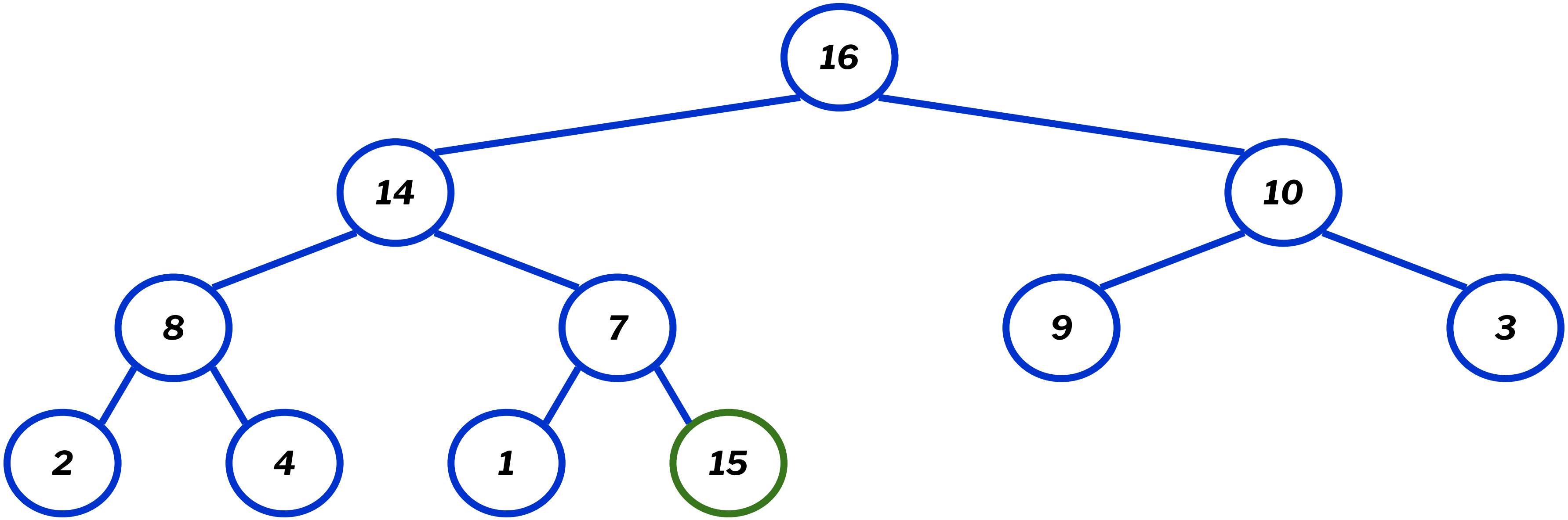
- Increase the heap size



A =										
	16	14	10	8	7	9	3	2	4	1

Binary Heaps: Insert

- Keep the new element at the end of the heap (tree)

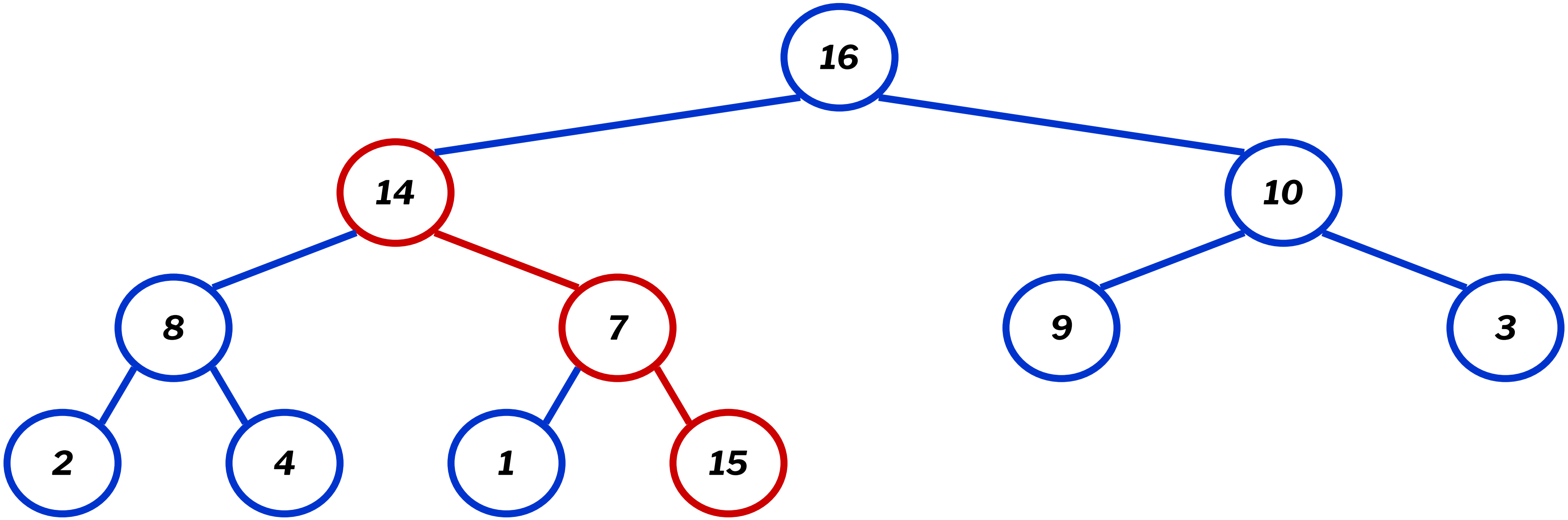


A =

16	14	10	8	7	9	3	2	4	1	15

Binary Heaps: Insert

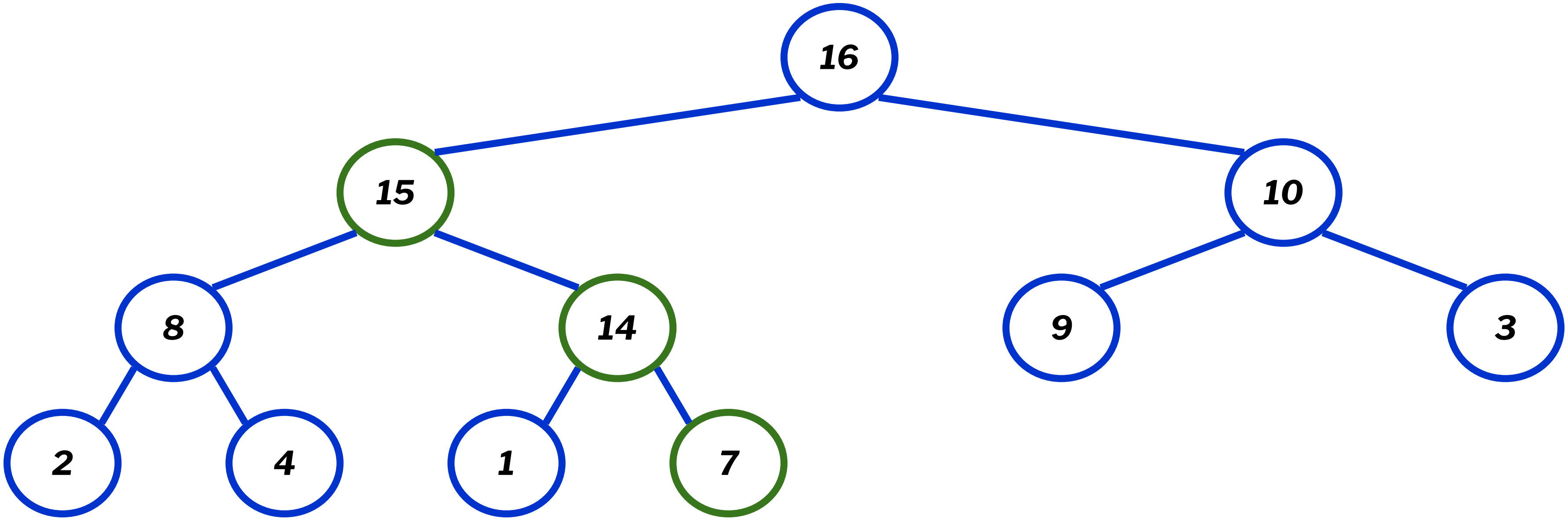
- *PercolateDown*



A =	16	14	10	8	7	9	3	2	4	1	15

Binary Heaps: Insert

- Satisfy the Heap property



A =

16	15	10	8	14	9	3	2	4	1	7

Binary Heaps: Build Heap

- Steps:
  - Walk backwards through the array from n/2 to 1, calling *PercolateDown* on each node.

*Order of processing guarantees that the children of node i are heaps when i is processed.

BUILD-MAX-HEAP(A)

$heap\text{-}size[A] \leftarrow length[A]$

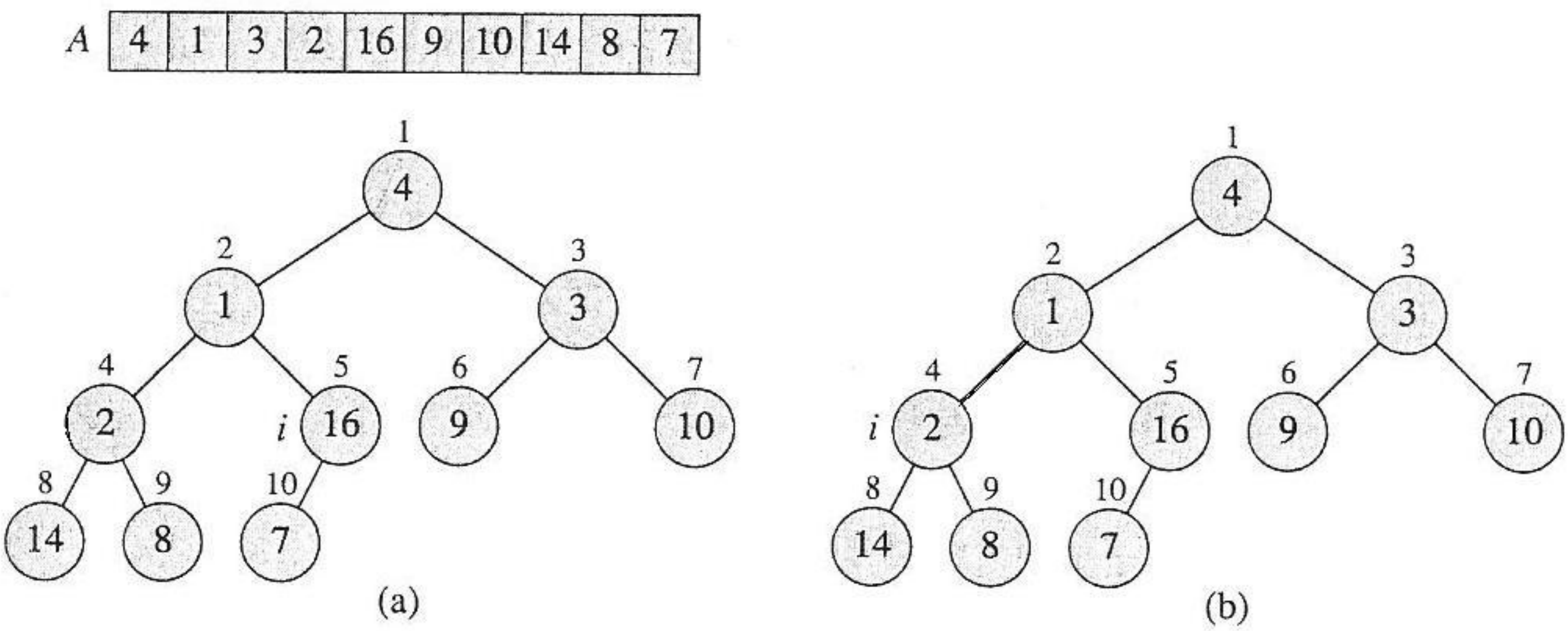
for $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1

  do PercolateDown (A, i)

Converts an unorganized array A into a max-heap.

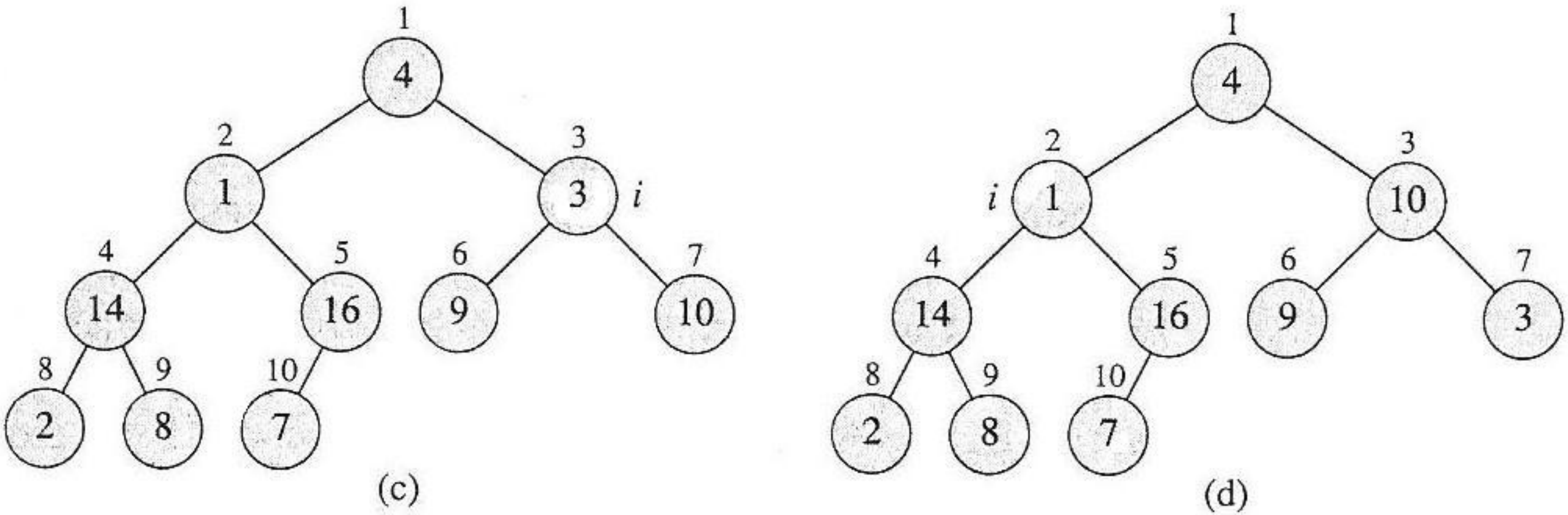
Binary Heaps: Build Heap

- Work through example:
  - A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



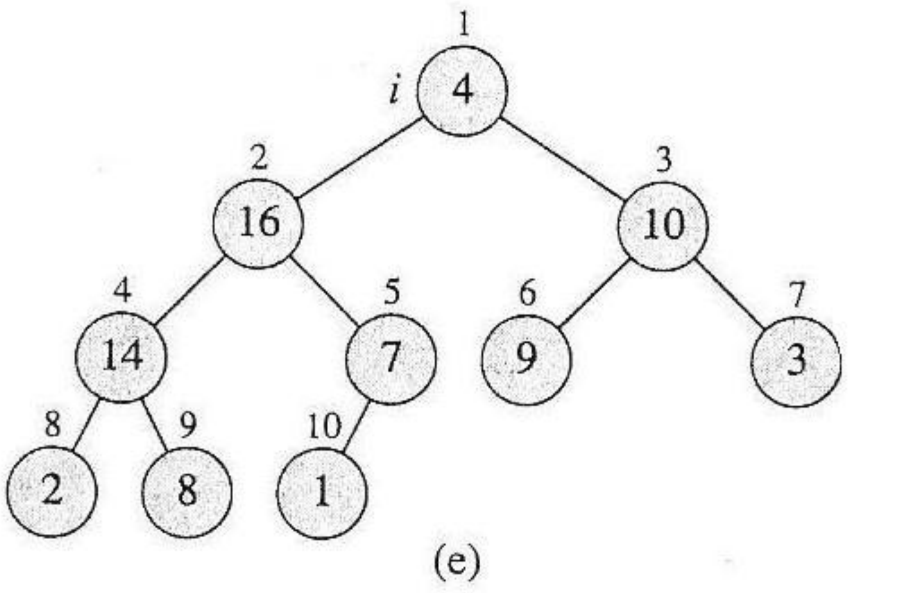
Binary Heaps: Build Heap

- **Work through example:**
  - **A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}**



Binary Heaps: Build Heap

- **Work through example:**
  - **A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}**



Binary Heaps: Build Heap

- Show that the height of a heap with n elements is logn.
  - A heap is a complete binary tree.
  - All the levels, except the lowest, are completely full.
  - A heap has at least 2^h elements ((if the lowest level has just 1 element and all the other levels are complete)
  - A heap has at most elements $2^{h+1} – 1$.
  - Hence, $2^h \leq n \leq 2^{h+1} – 1$
  - This implies, $h \leq \log n \leq h + 1$.
  - Since h is an integer, $h = \log n$.

Binary Heaps: Build Heap

- Analyzing BuildHeap
  - Each call to *PercolateDown* takes O(log n) time
  - There are O(n) such calls (specifically, $\lfloor n/2 \rfloor$)
  - Thus the running time is O(n log n)
    - Is this a correct asymptotic upper bound?
    - Is this an asymptotically tight bound?
  - A tighter bound is O(n)
    - How can this be? Is there a flaw in the above reasoning?

Binary Heaps: Build Heap

- Prove that, for a complete binary tree of height h the sum of the height of all nodes is O(n – h)
  - A complete binary tree has 2i nodes on level i.
  - A node on level i has depth i and height h – i.
  - Let us assume that S denotes the sum of the heights of all these nodes and S can be calculated as:

$$S = \sum_{i=0}^{h} 2^i (h - i)$$

$$S = h + 2(h - 1) + 4(h - 2) + \cdots + 2^{h-1}$$

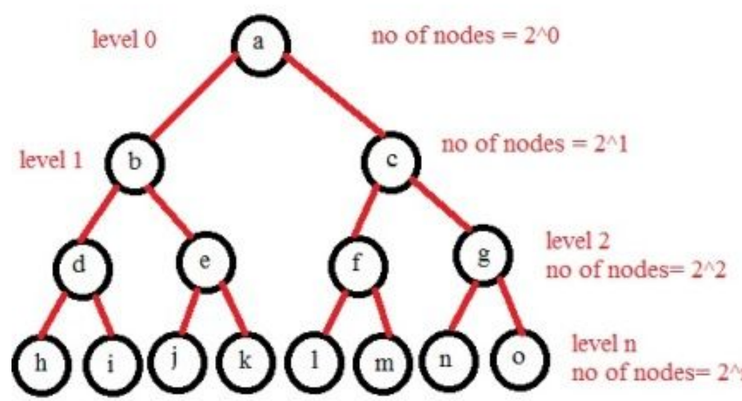
$$2S = 2h + 4(h - 1) + 8(h - 2) + \cdots + 2h$$

$$2S - S = -h + 2 + 4 + \cdots + 2h$$

$$\Rightarrow S = (2^{h+1} - 1) - (h - 1)$$

$$\Rightarrow S = (2^{h+1} - 1) - (h - 1) = n - (h - 1) = n - h + 1$$

$$\Rightarrow O(n-h)$$



Binary Heaps: Build Heap

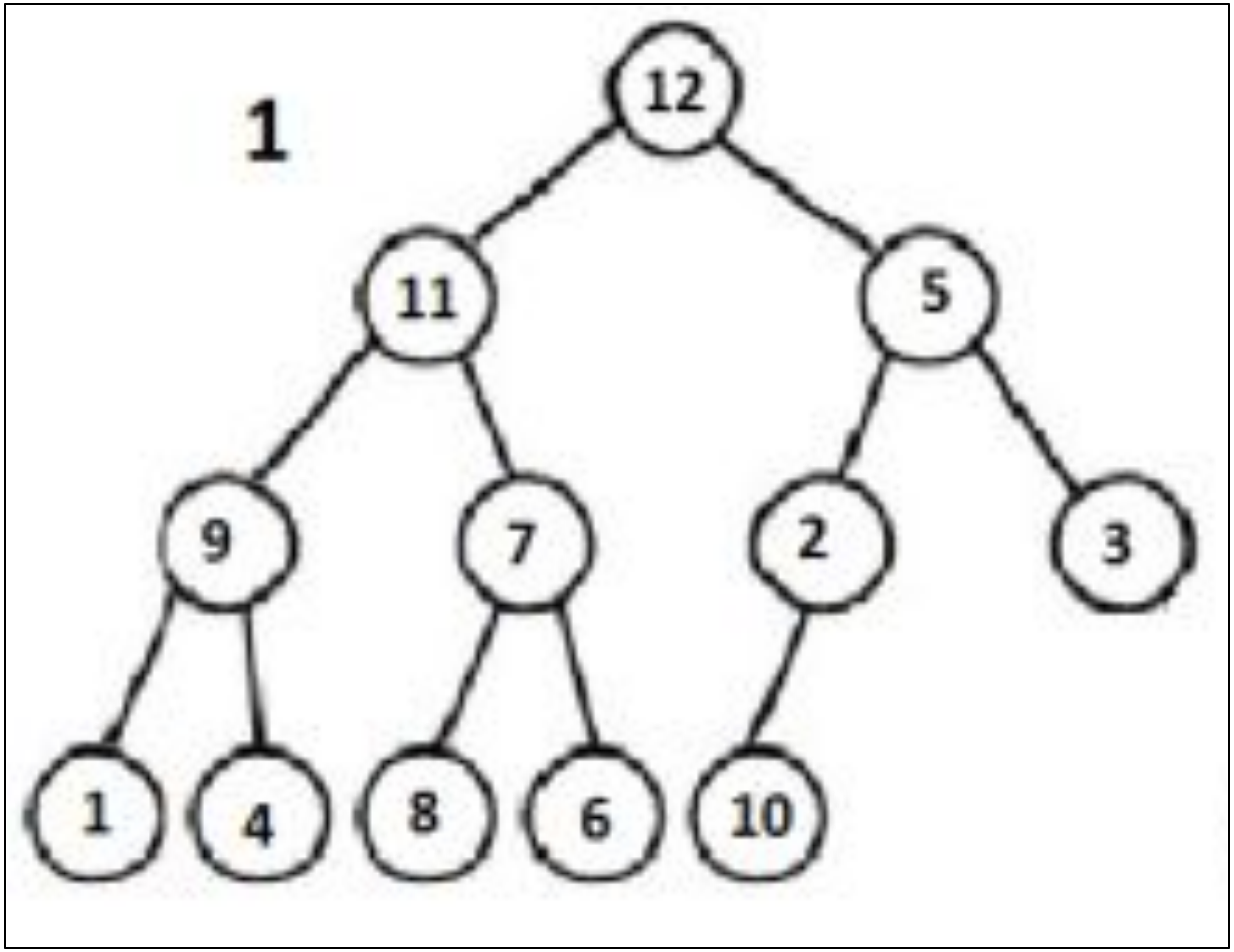
- Time Complexity:
  - The linear time bound of building heap can be shown by computing the sum of the heights of all the nodes.
  - For a complete binary tree of height h containing $n = 2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $n - h - 1 = n - \log n - 1$
  - That means, building the heap operation can be done in linear time (O(n)) by applying a *PercolateDown* function to the nodes in reverse level order.

Heap Sort

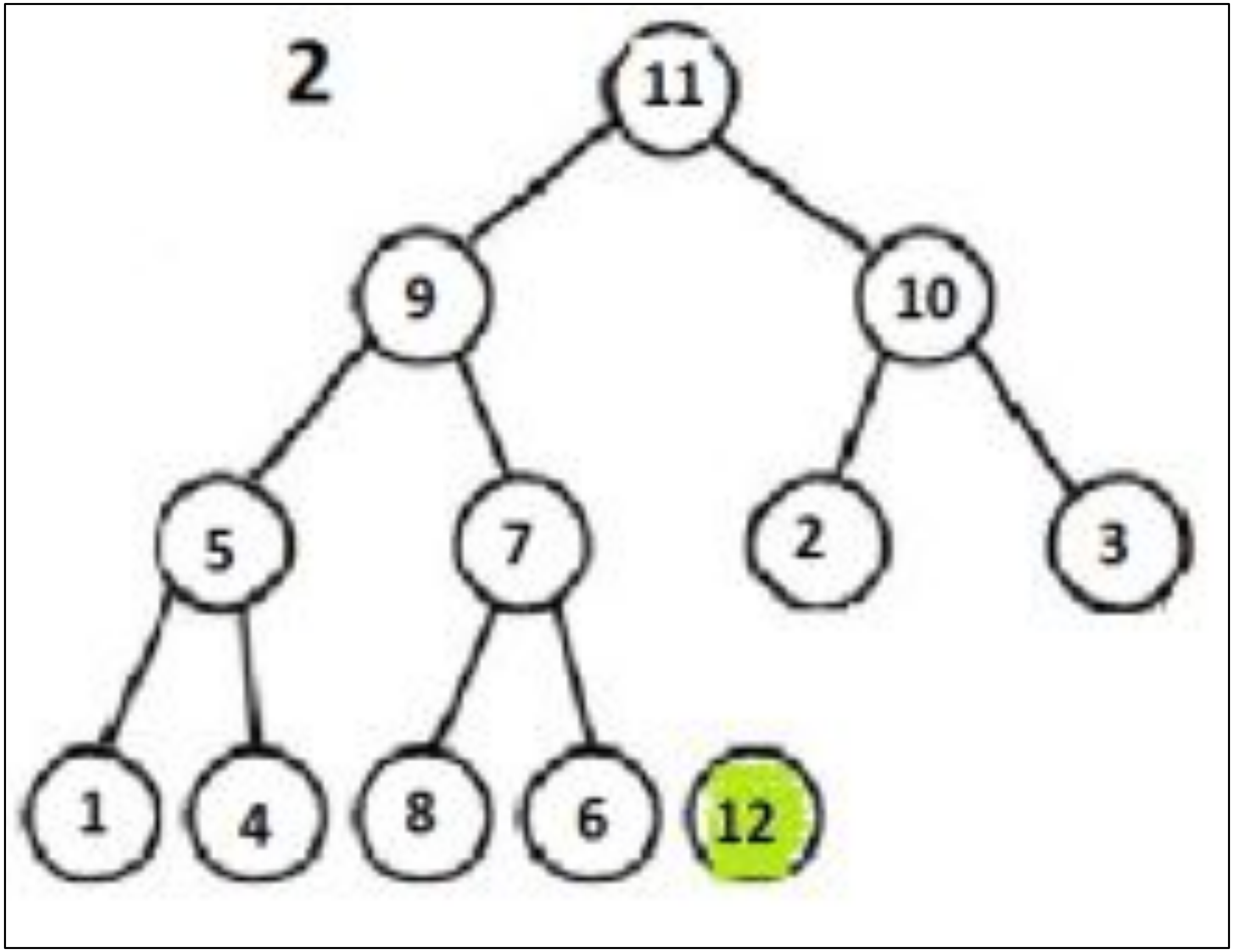
- **Steps:**
  - Maximum element is at A[1]
  - Discard by swapping with element at A[*n*]
    - Decrement heap_size[A]
    - A[*n*] now contains maximum
- Restore heap property at A[1] by calling Heapify [PercolateDown]
- Repeat, always swapping A[1] for A[heap_size(A)]

```
Heapsort(A) {
   BuildHeap(A)
   for i <- length(A) downto 2 {
      exchange A[1] <-> A[i]
      heapsize <- heapsize -1
      Heapify(A, 1)
}
```

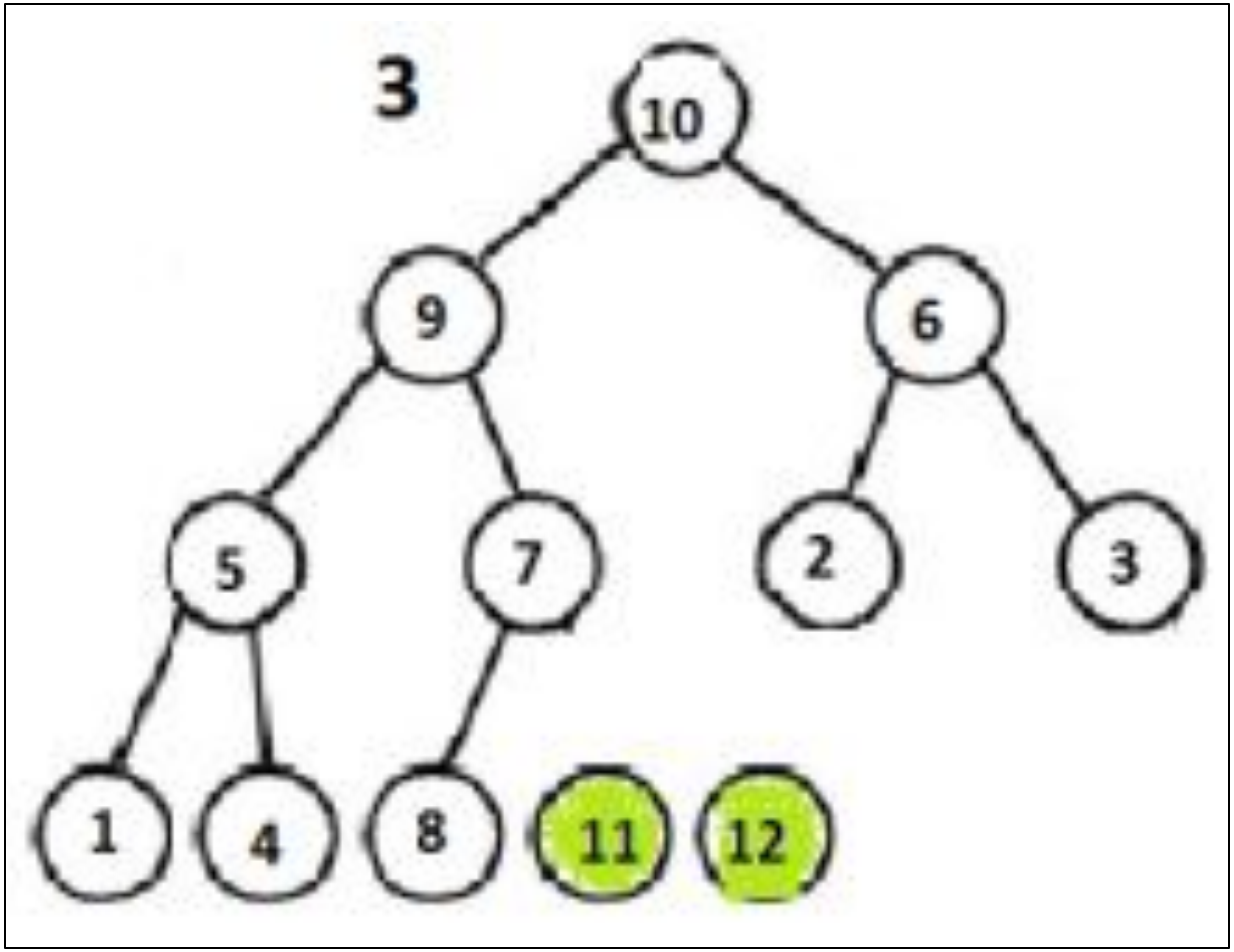
Heap Sort



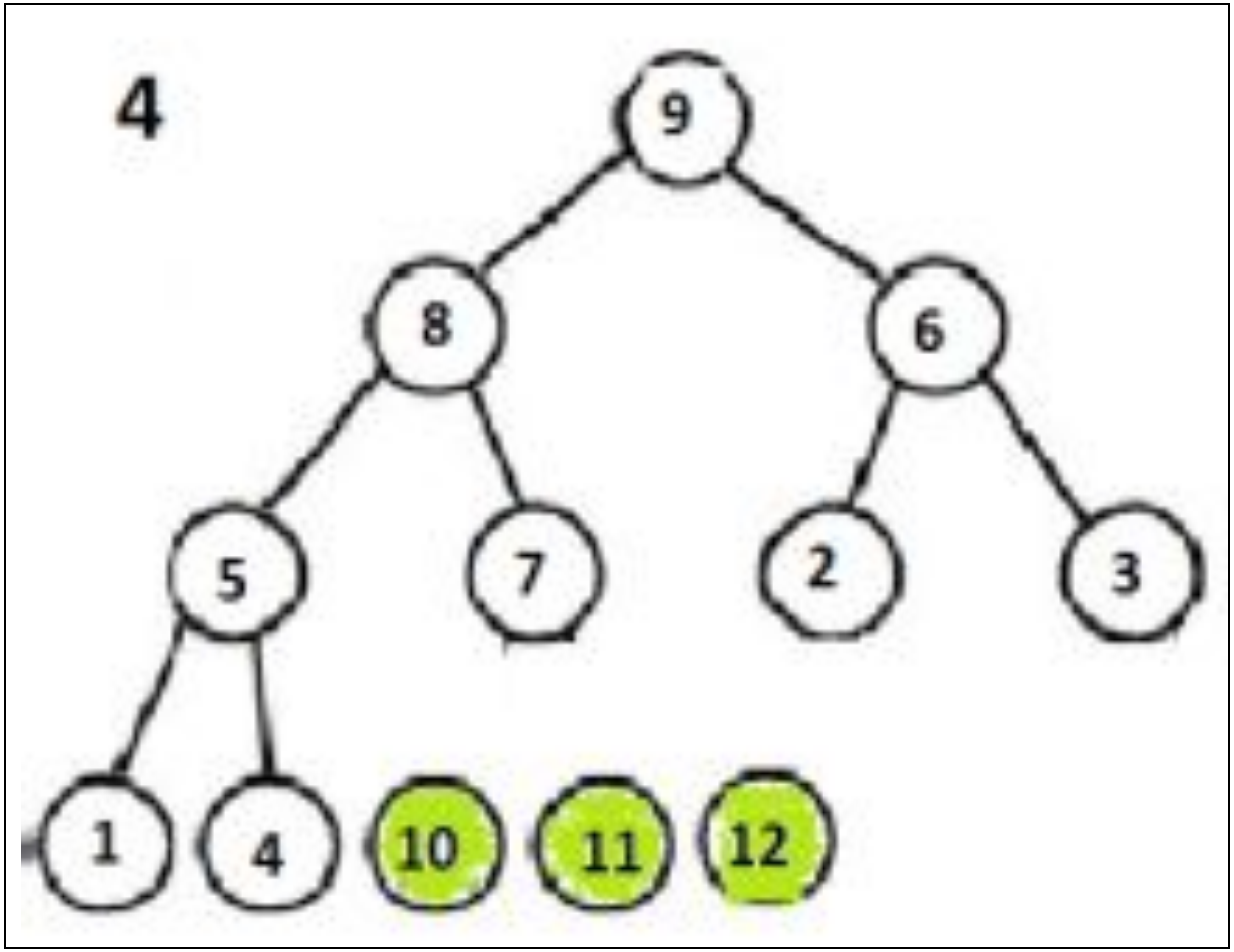
Heap Sort



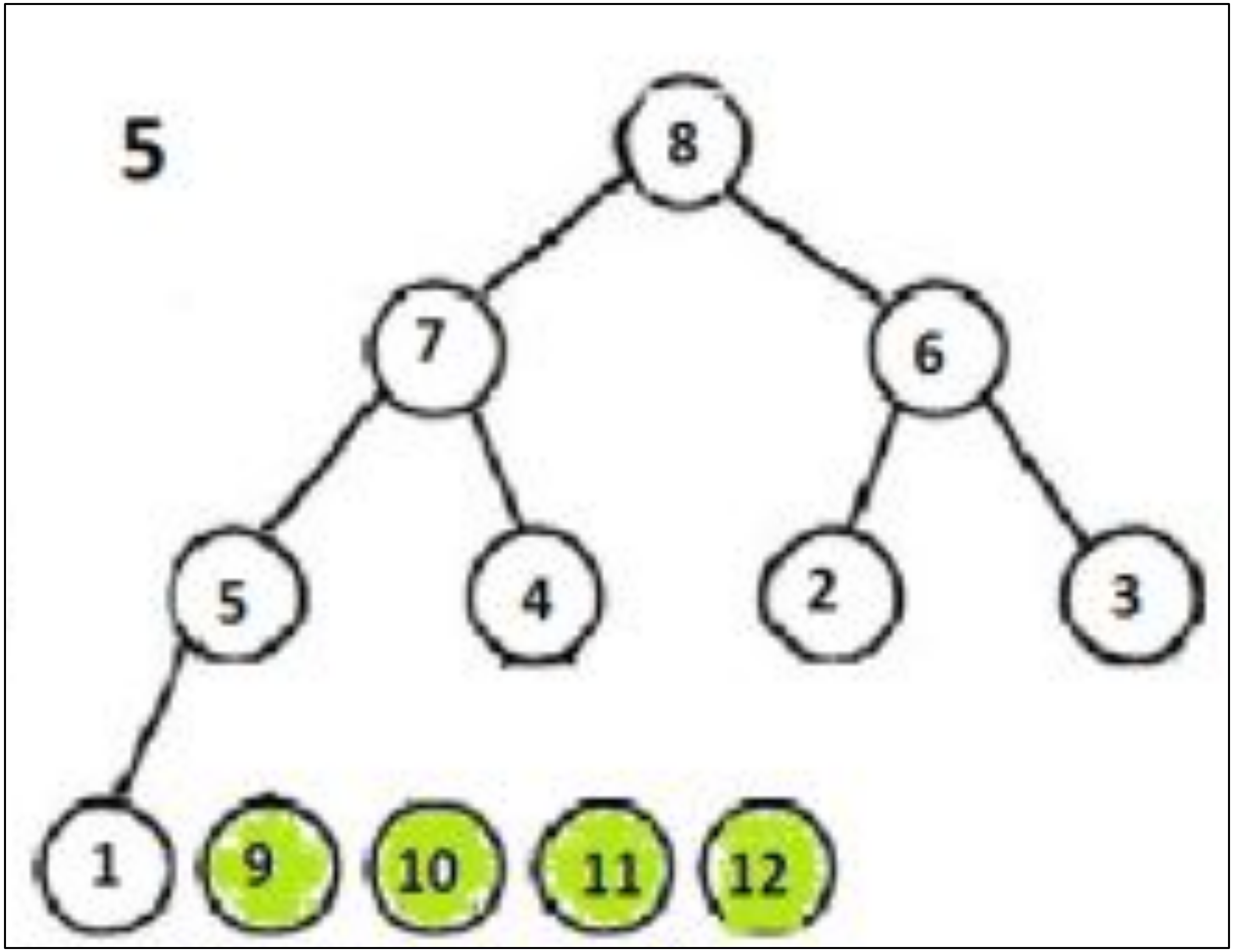
Heap Sort



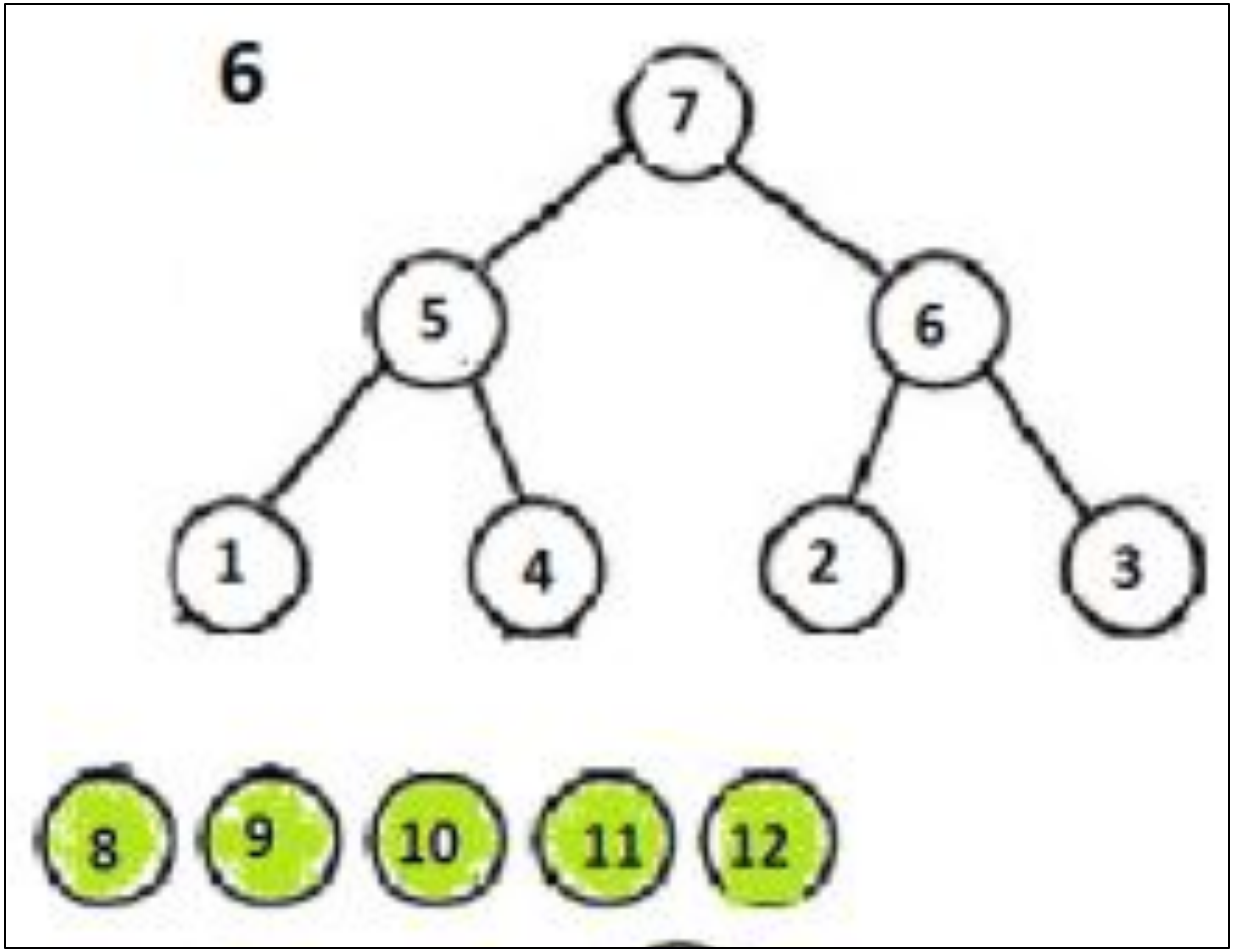
Heap Sort



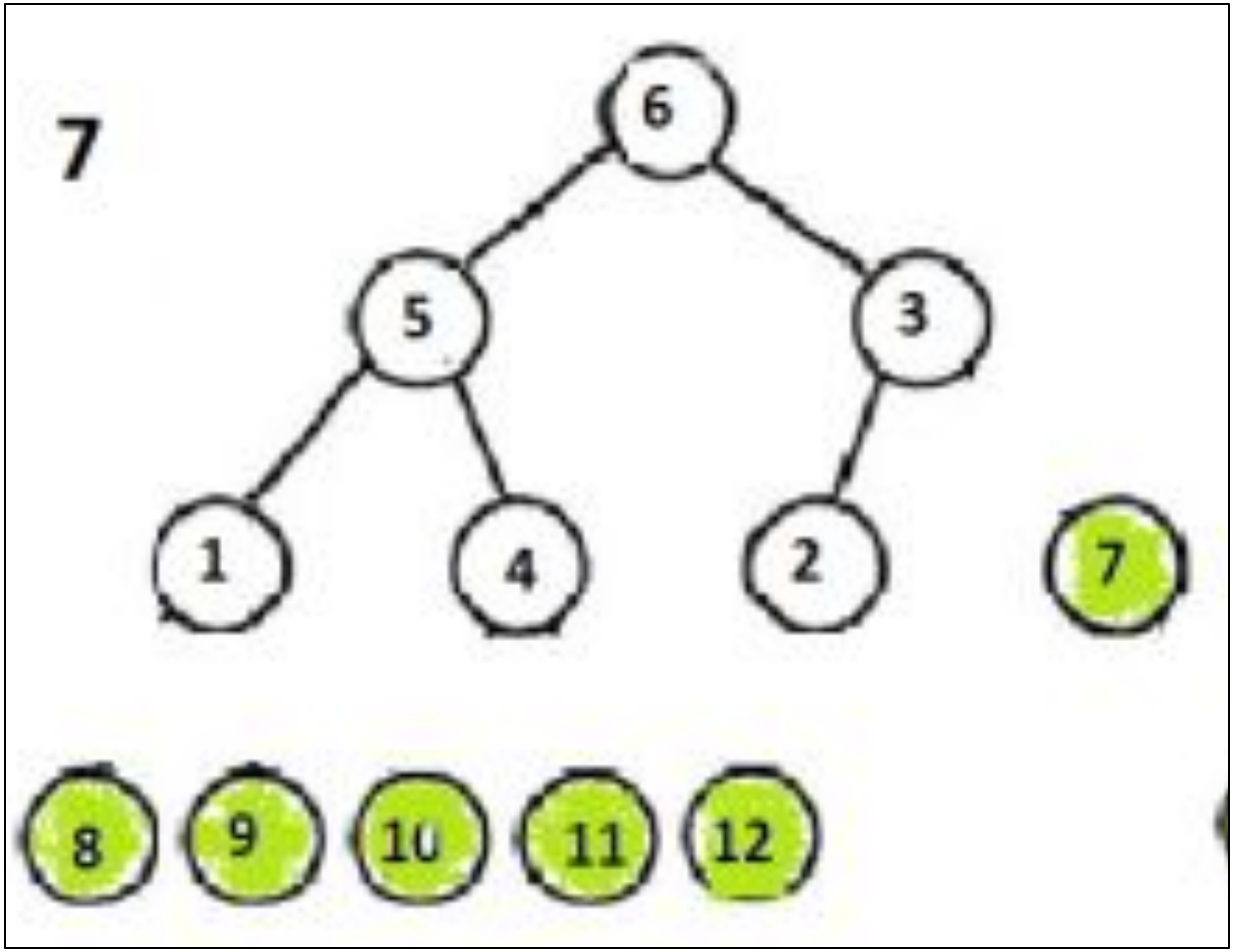
Heap Sort



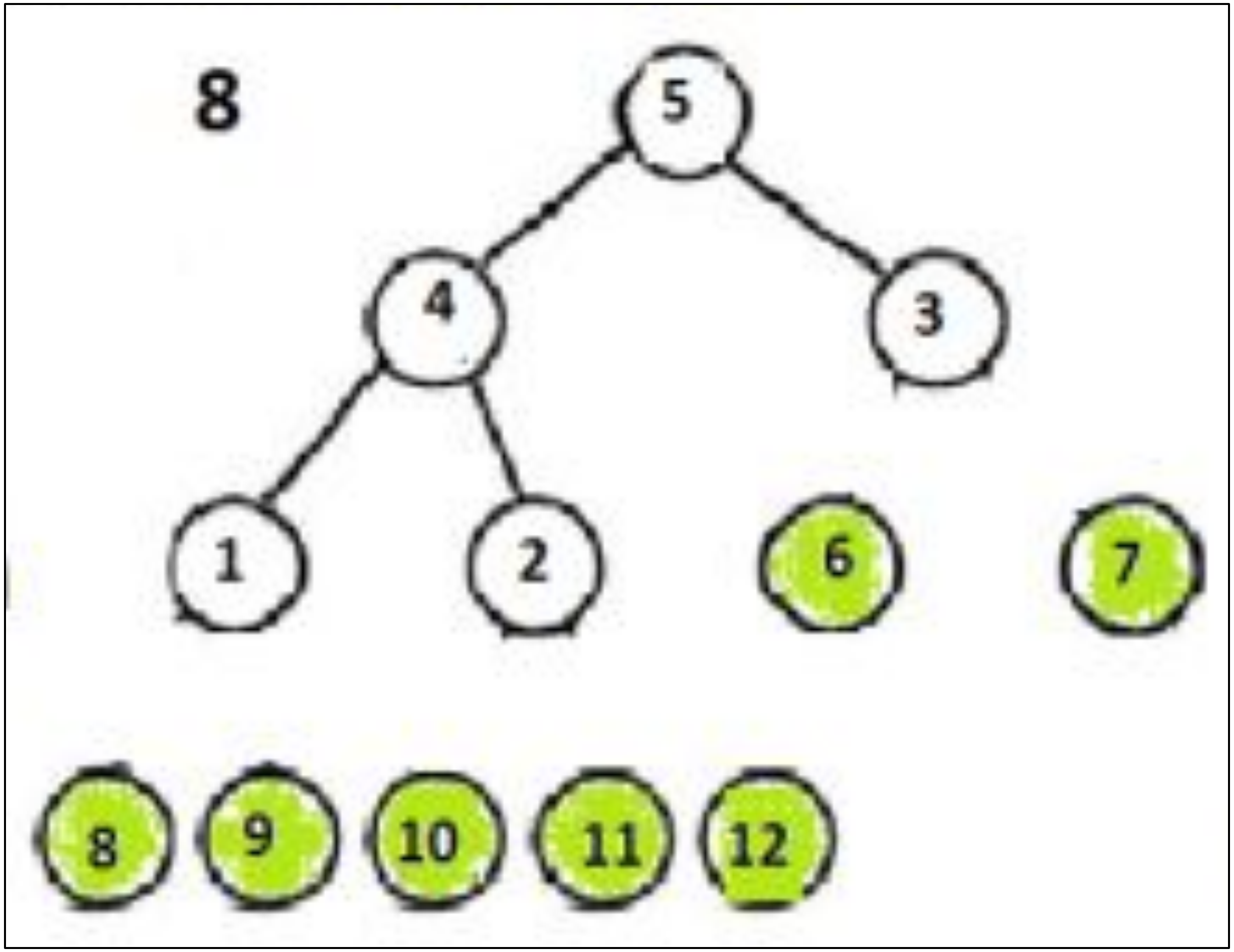
Heap Sort



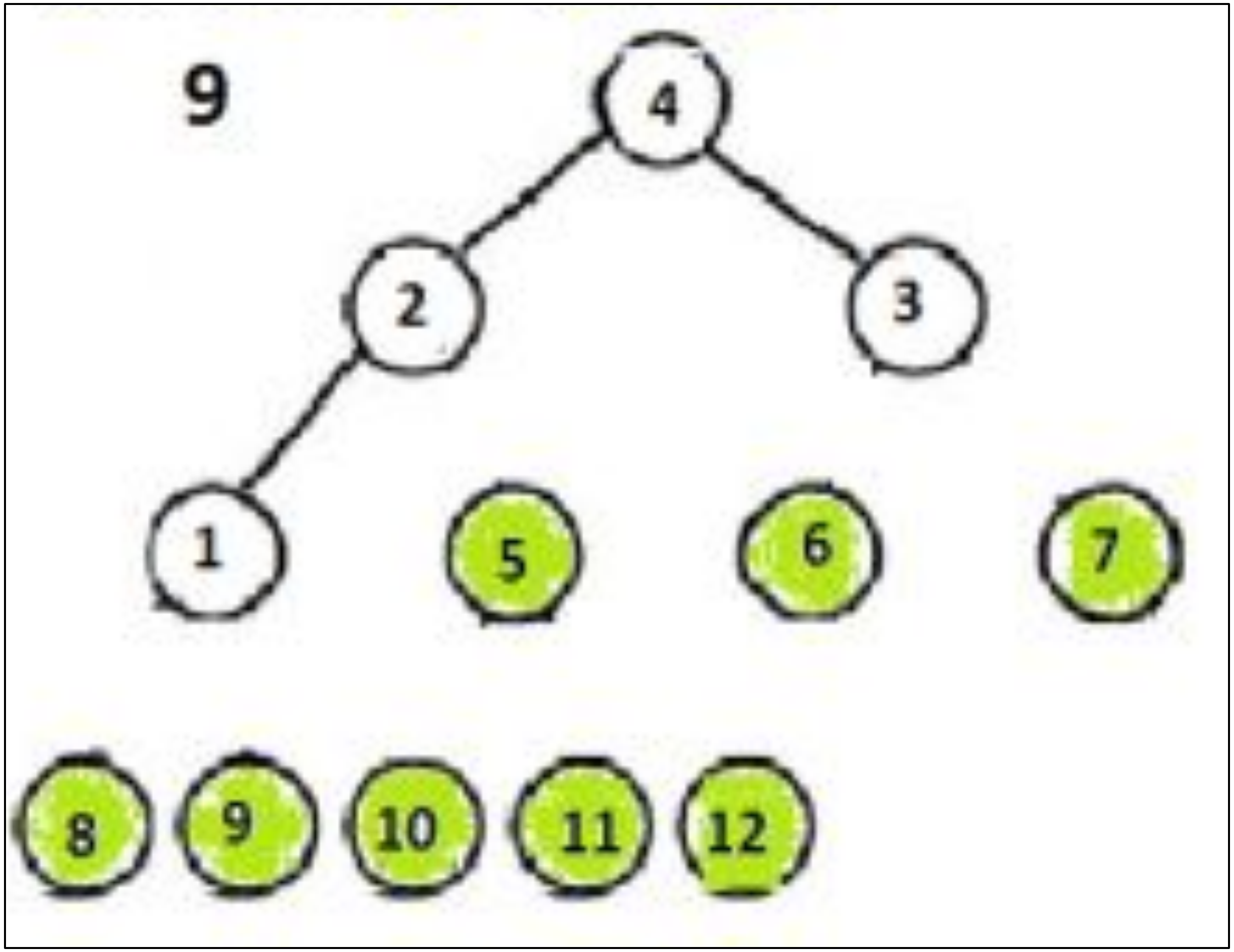
Heap Sort



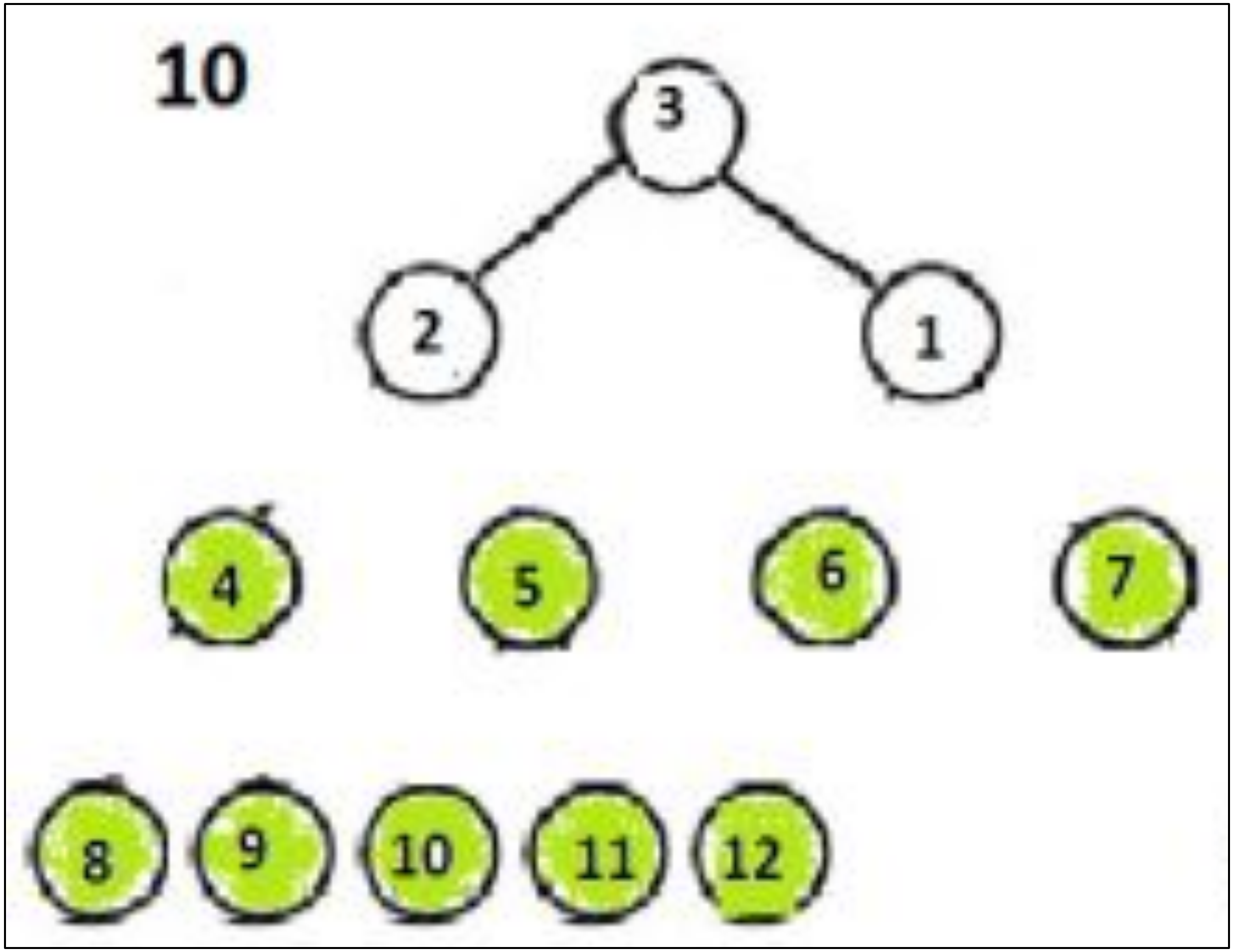
Heap Sort



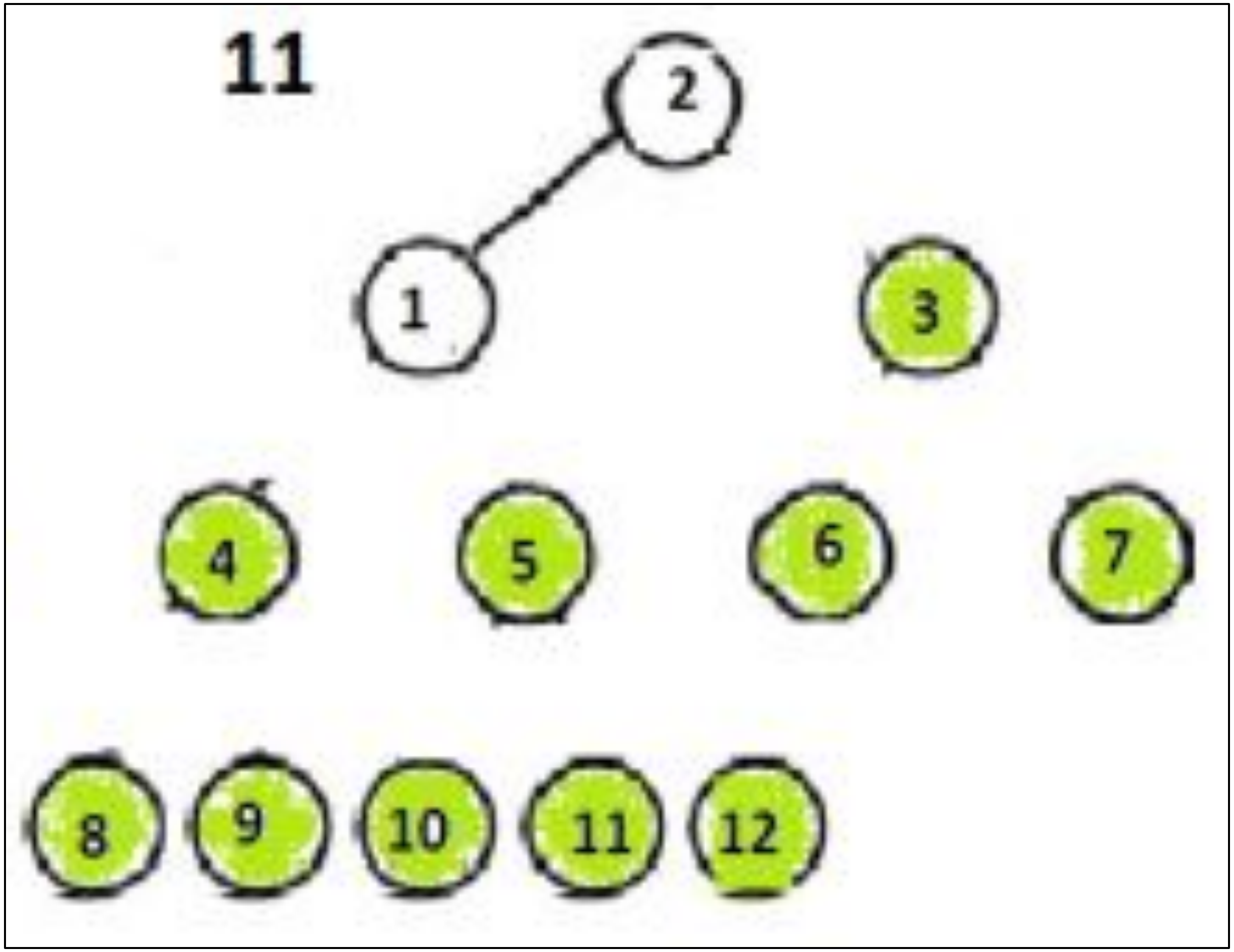
Heap Sort



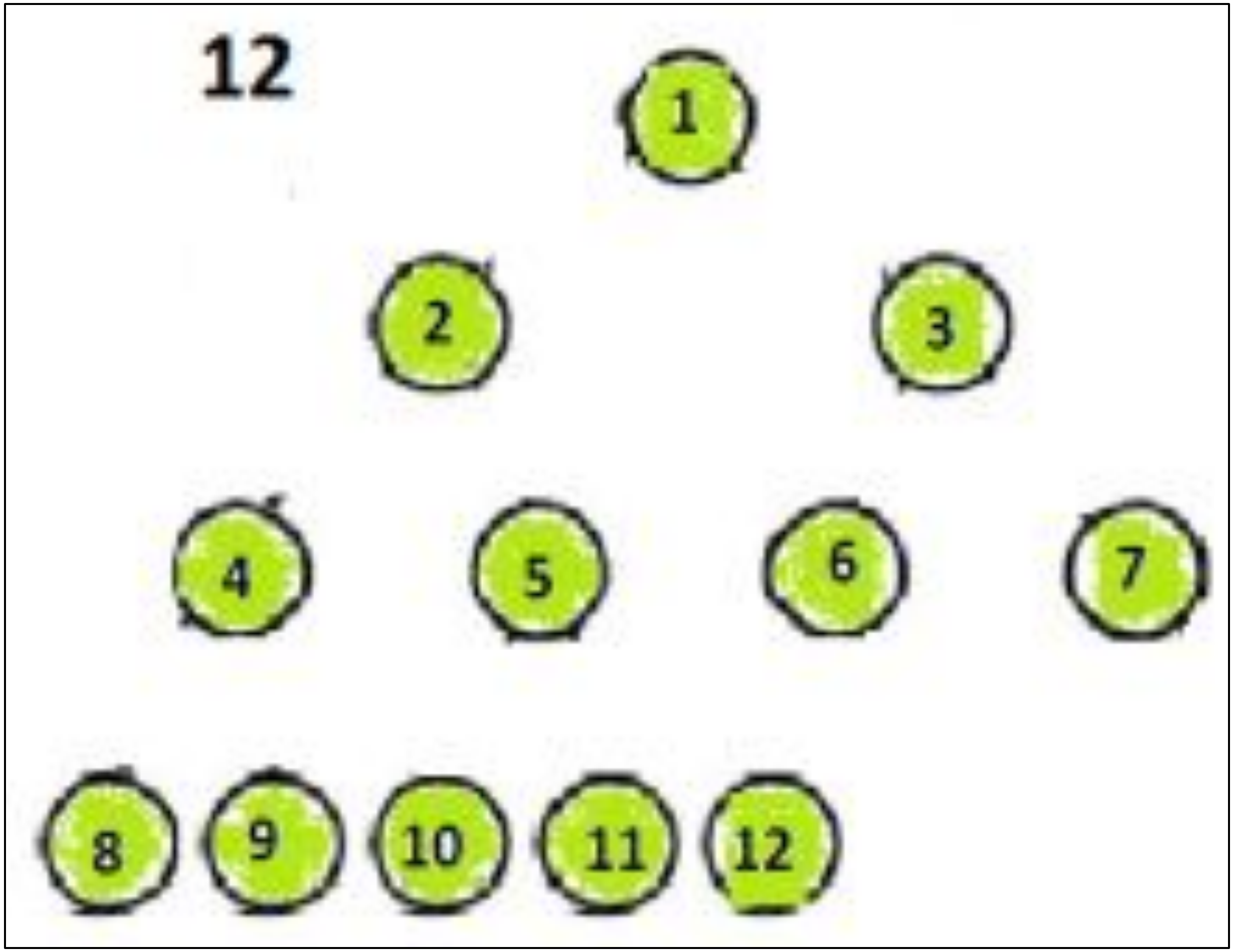
Heap Sort



Heap Sort



Heap Sort



Heap Sort

- **Analyzing Heap Sort:**
  - The call to BuildHeap() takes $O(n)$ time
  - Each of the $n - 1$ calls to Heapify() takes $O(\log n)$ time
  - Thus the total time taken by HeapSort()

$$= O(n) + (n - 1)\, O(\log n)$$

$$= O(n) + O(n \log n)$$

$$= O(n \log n)$$