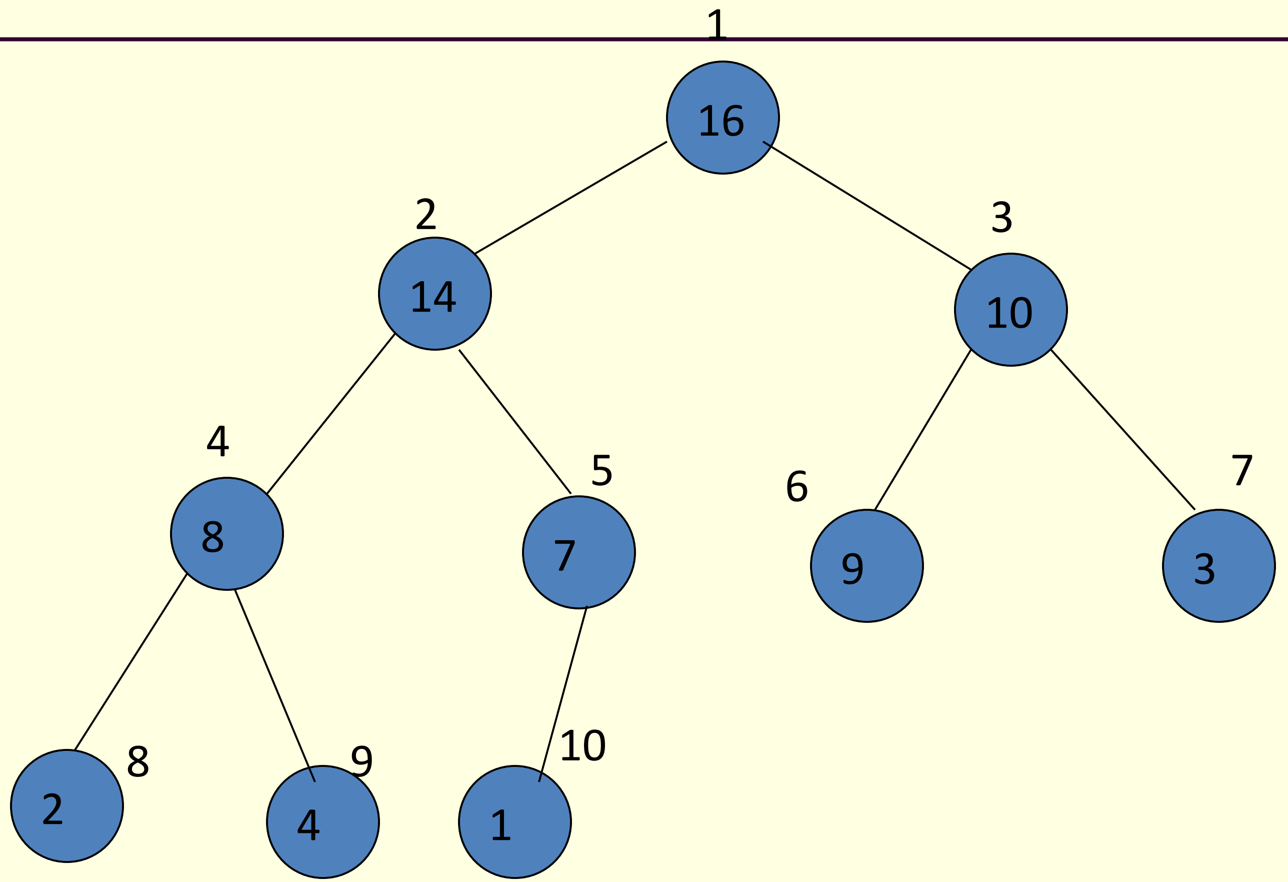
Heap Sort

Kumkum Saxena

Heap Data Structure

The heap data structure is an array object that can be viewed as a nearly complete binary tree.

Heap



(a)

Heap

There are two kinds of binary heaps

- max-heaps
- min heaps

Maintaining the Heap Property

The function of MAX-HEAPIFY is to let the value at A[i] “float down” in the max-heap so that the sub-tree rooted at index i becomes a max-heap

Maintaining the Heap Property

MAX-HEAPIFY(A, i)

l← LEFT(i)

r ← RIGHT(i)

if l <= heap-size[A] and A[l] >A [i]

then largest ← l

else largest ← i

Maintaining the Heap Property

If r<= heap-size[A] and A[r] > A[largest]

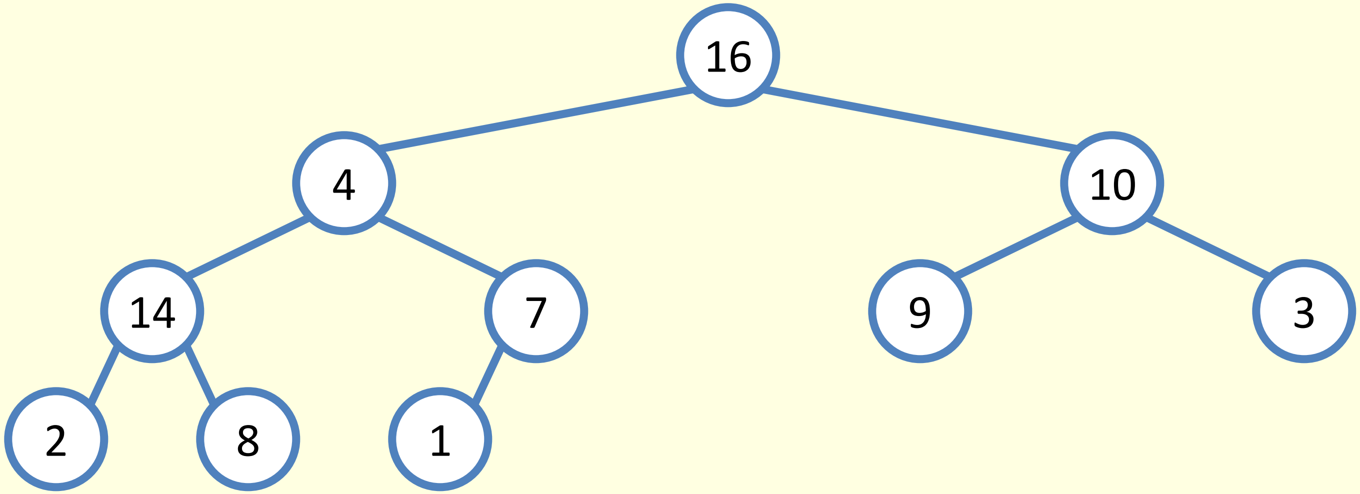
then largest ← r

If largest ≠ i

then exchange A[i] ←→ A[largest]

MAX-HEAPIFY(A, largest)

Heapify() Example

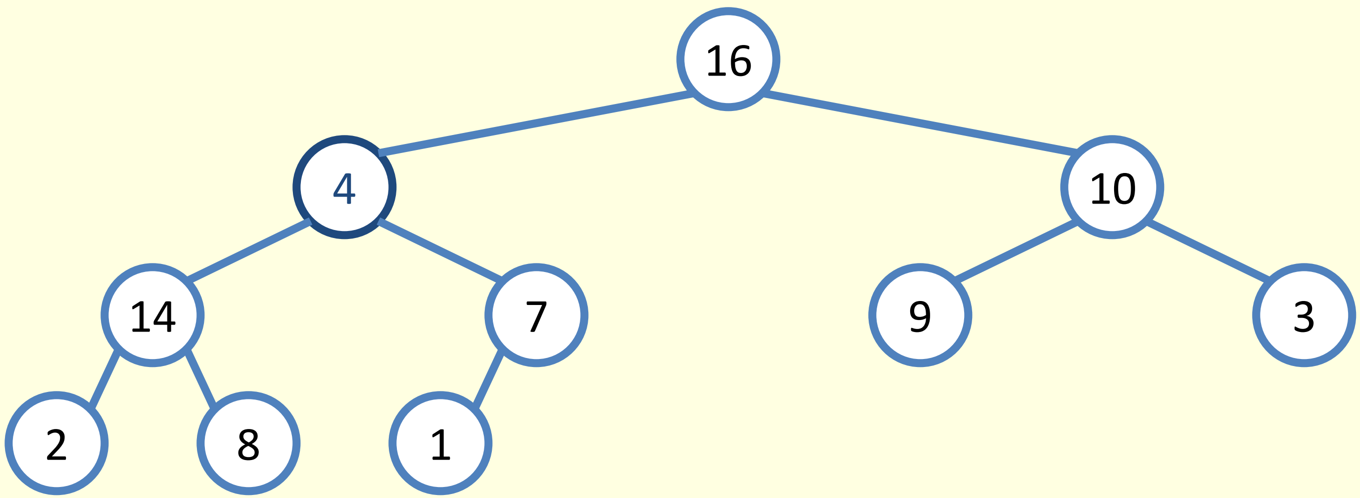


A = | 16 | 4 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |

16	4	10	14	7	9	3	2	8	1

David Luebke

Heapify() Example

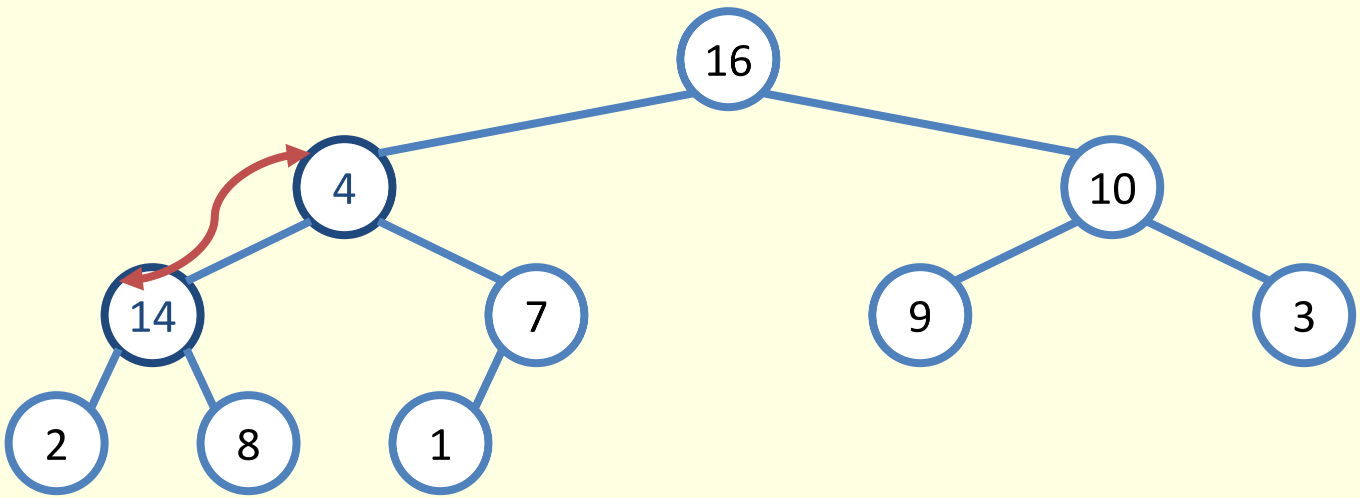


A =

16	4	10	14	7	9	3	2	8	1

David Luebke

Heapify() Example

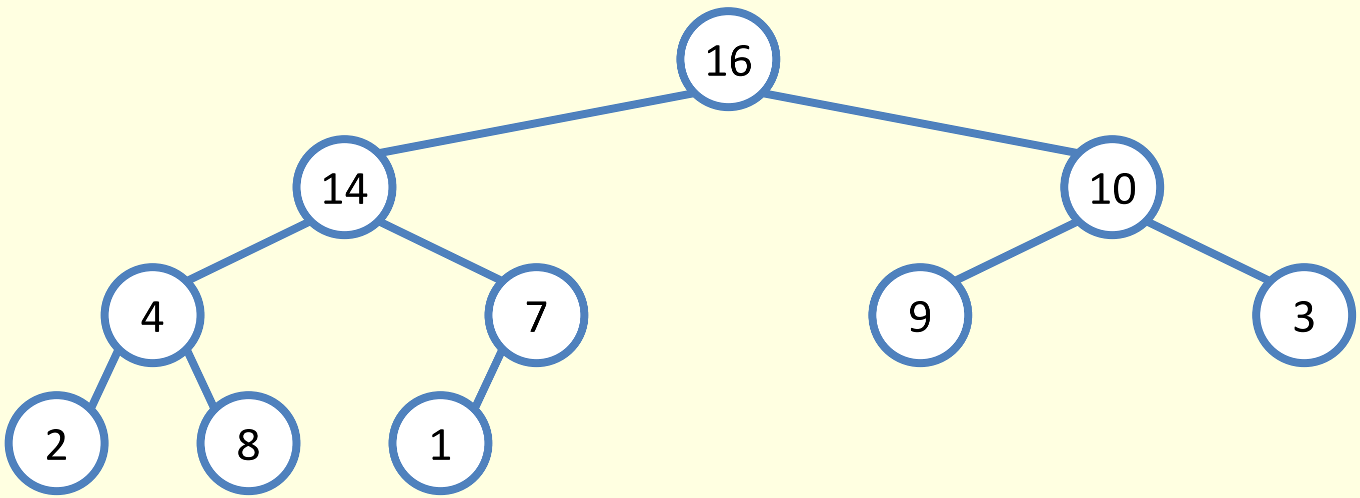


A =

16	4	10	14	7	9	3	2	8	1

David Luebke

Heapify() Example

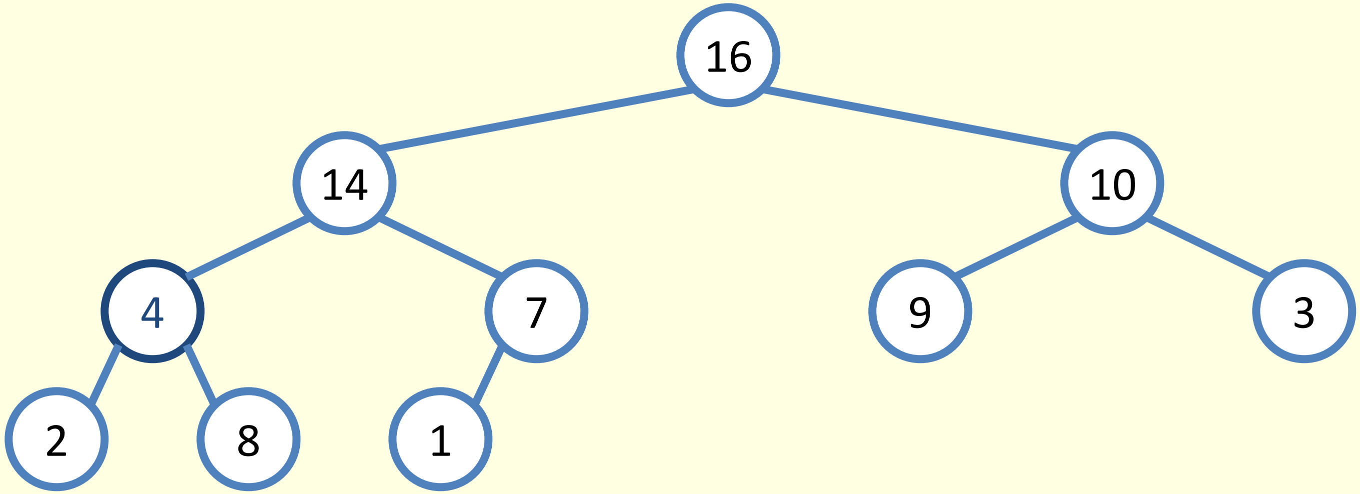


A =

16	14	10	4	7	9	3	2	8	1

David Luebke

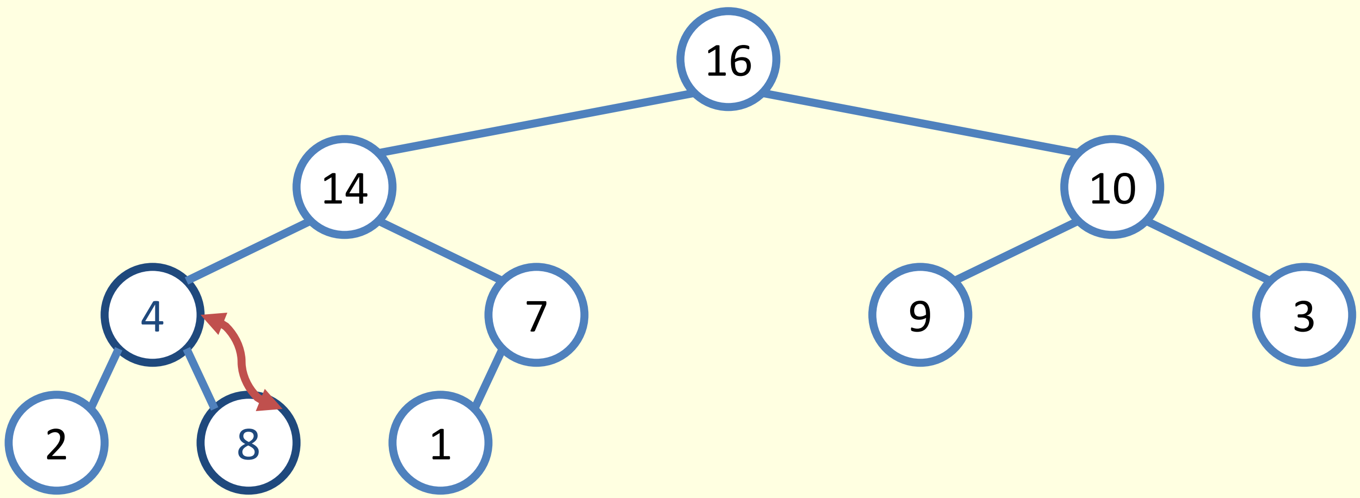
Heapify() Example



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

David Luebke

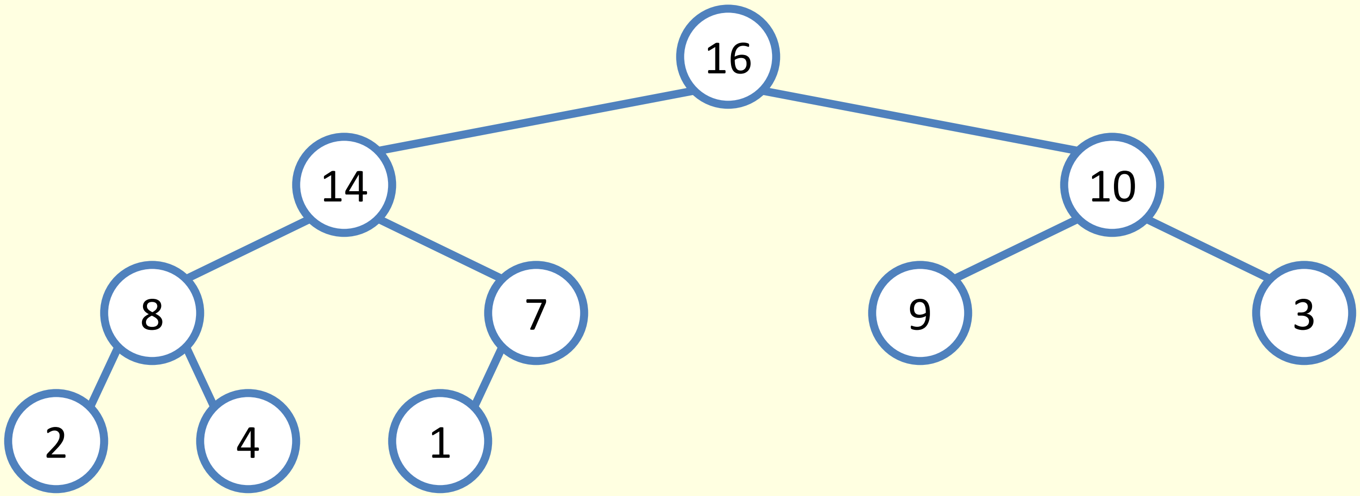
Heapify() Example



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

David Luebke

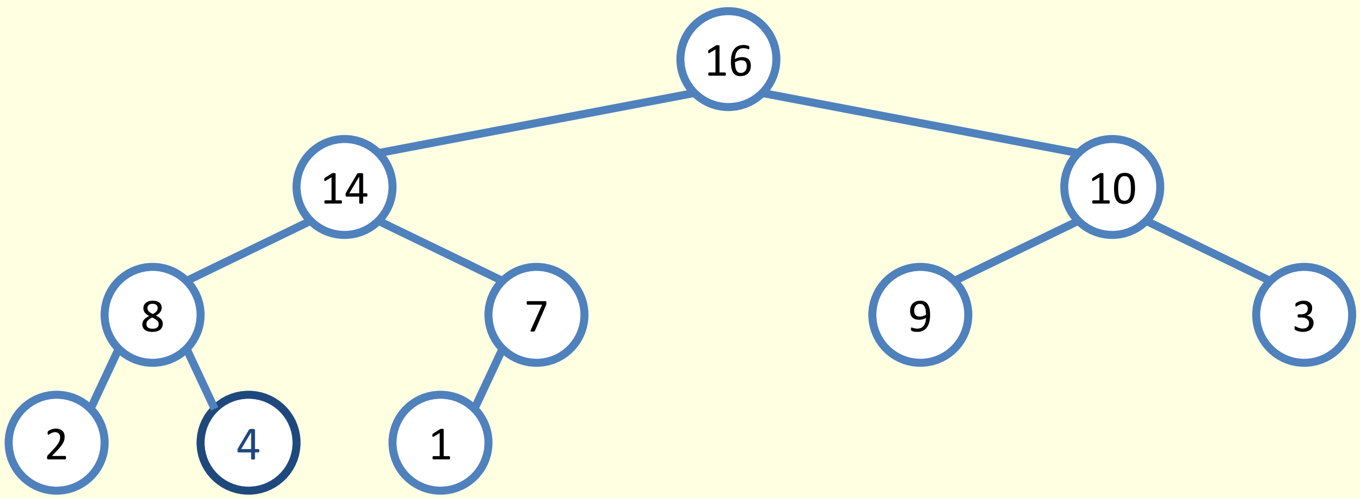
Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

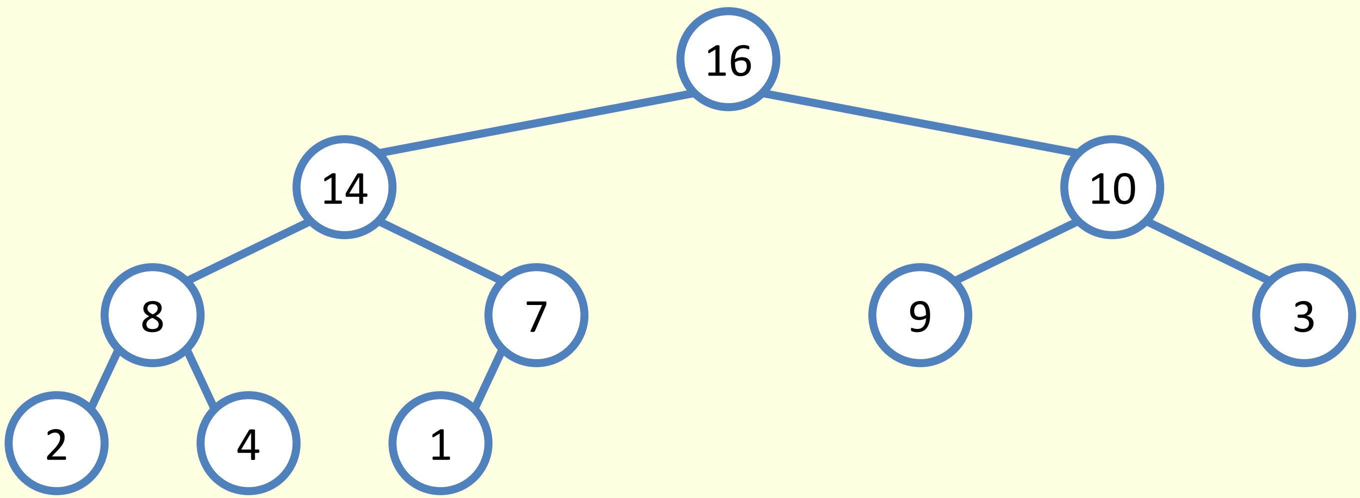
Heapify() Example



A =

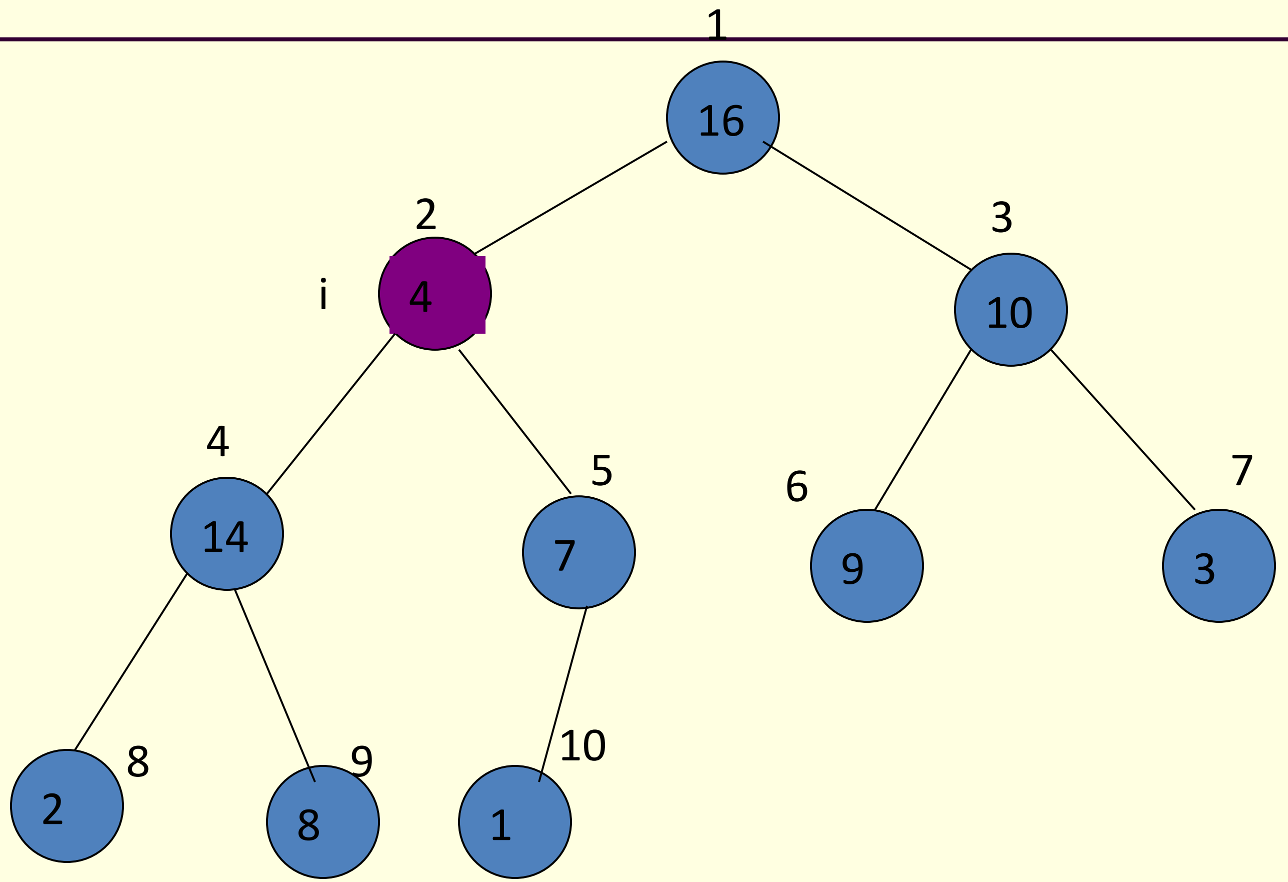
16	14	10	8	7	9	3	2	4	1

Heapify() Example



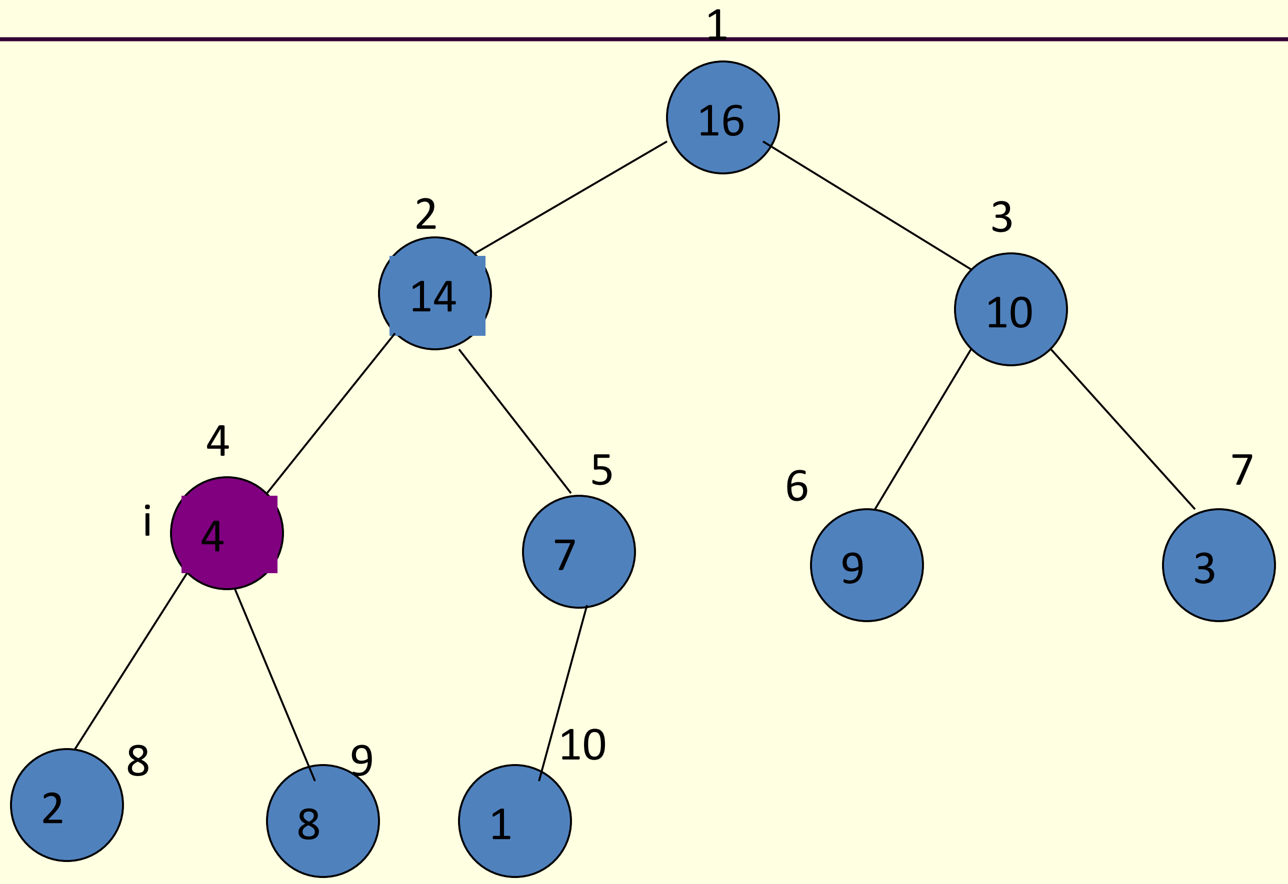
A =										
	16	14	10	8	7	9	3	2	4	1

Maintaining the Heap Property



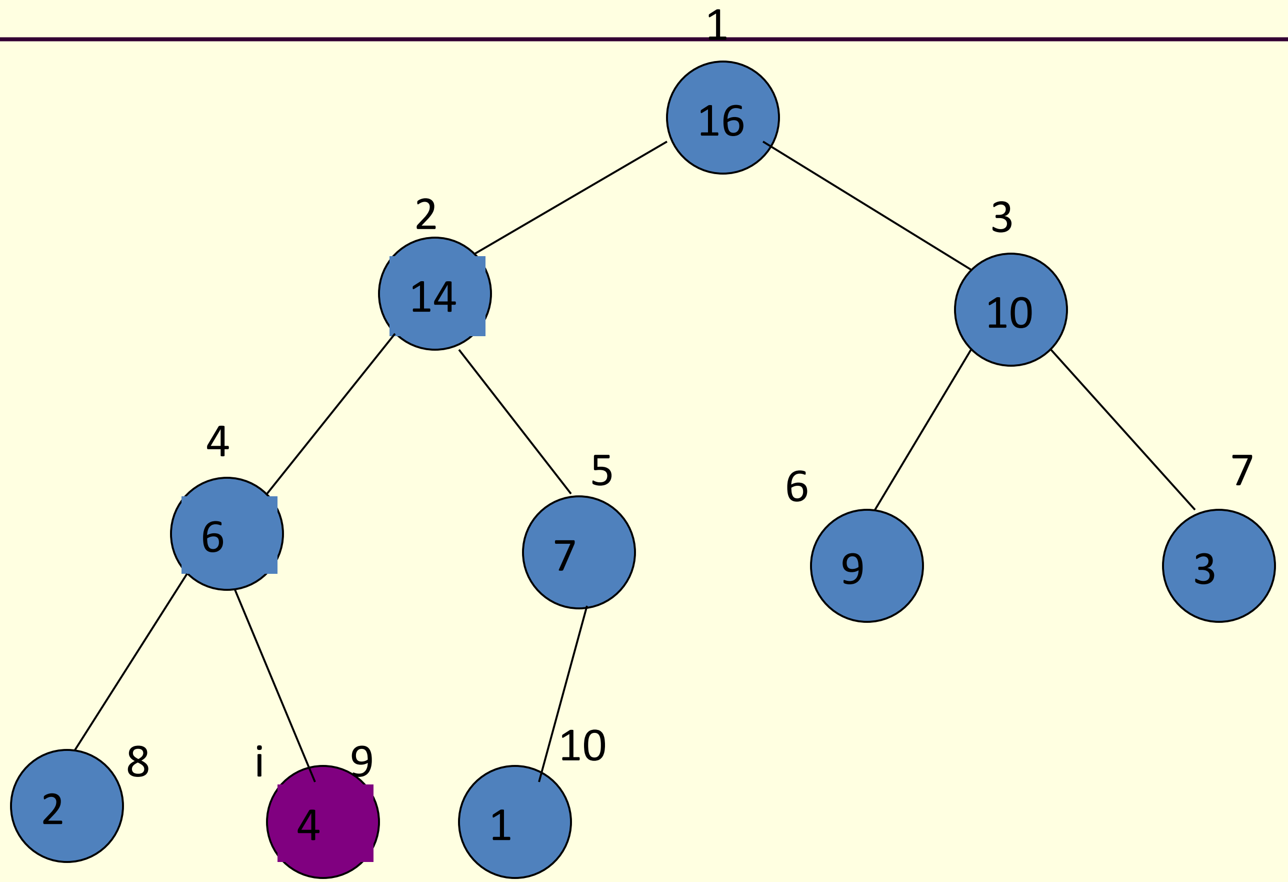
(a)

Maintaining the Heap Property



(b)

Maintaining the Heap Property



(c)

Analyzing Heapify(): Formal

- Fixing up relationships between *i*, *l*, and *r* takes $\Theta(1)$ time
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*
  - Draw it
- Answer: $2n/3$ (worst case: bottom row 1/2 full)
- So time taken by **`Heapify()`** is given by

$$T(n) \le T(2n/3) + \Theta(1)$$

Analyzing Heapify(): Formal

- So we have

$$T(n) \leq T(2n/3) + \Theta(1)$$

- By case 2 of the Master Theorem,

$$T(n) = O(\lg n)$$

- Thus, **`Heapify()`** takes logarithmic time

Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Fact: for array of length *n*, all elements in range
  - $A[\lfloor n/2 \rfloor + 1 .. n]$ are heaps (*Why?*)
  - So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

Building a Heap

BUILD-MAX-HEAP(A)

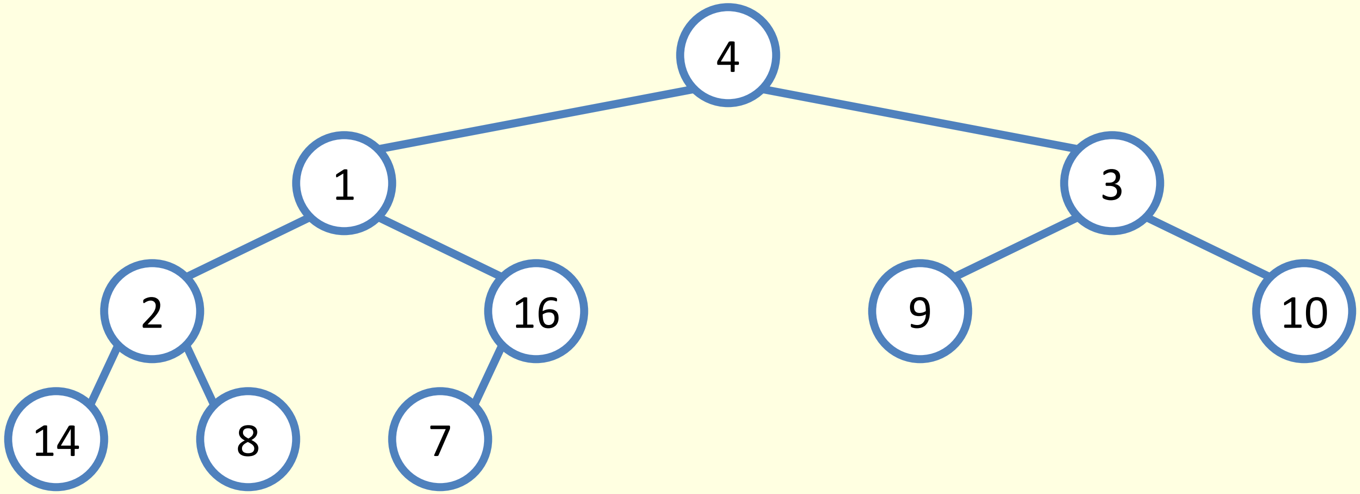
heap-size[A] ← length[A]

for i ← length[A]/2 downto 1

do MAX-HEAPIFY(A, i)

BuildHeap() Example

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

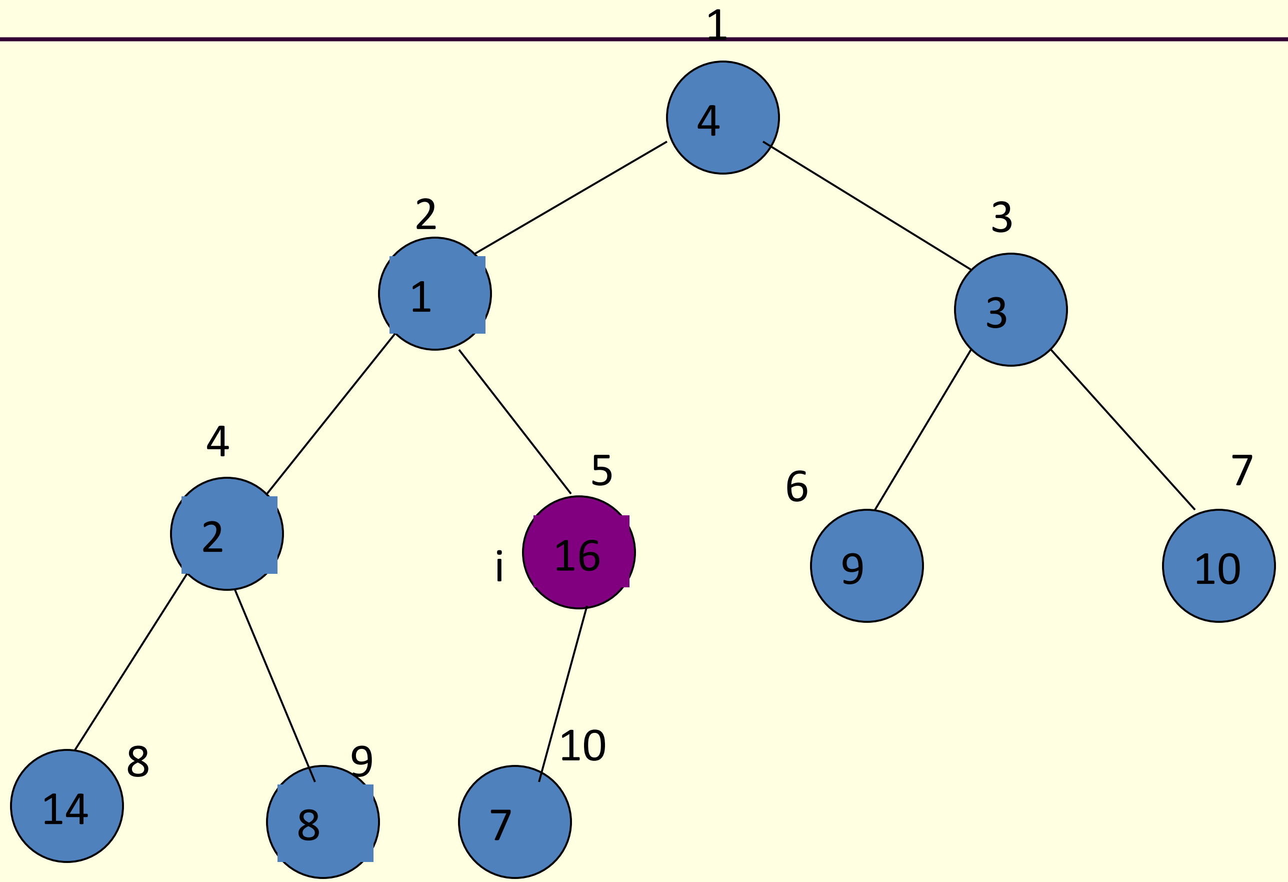


David Luebke

Building a Heap

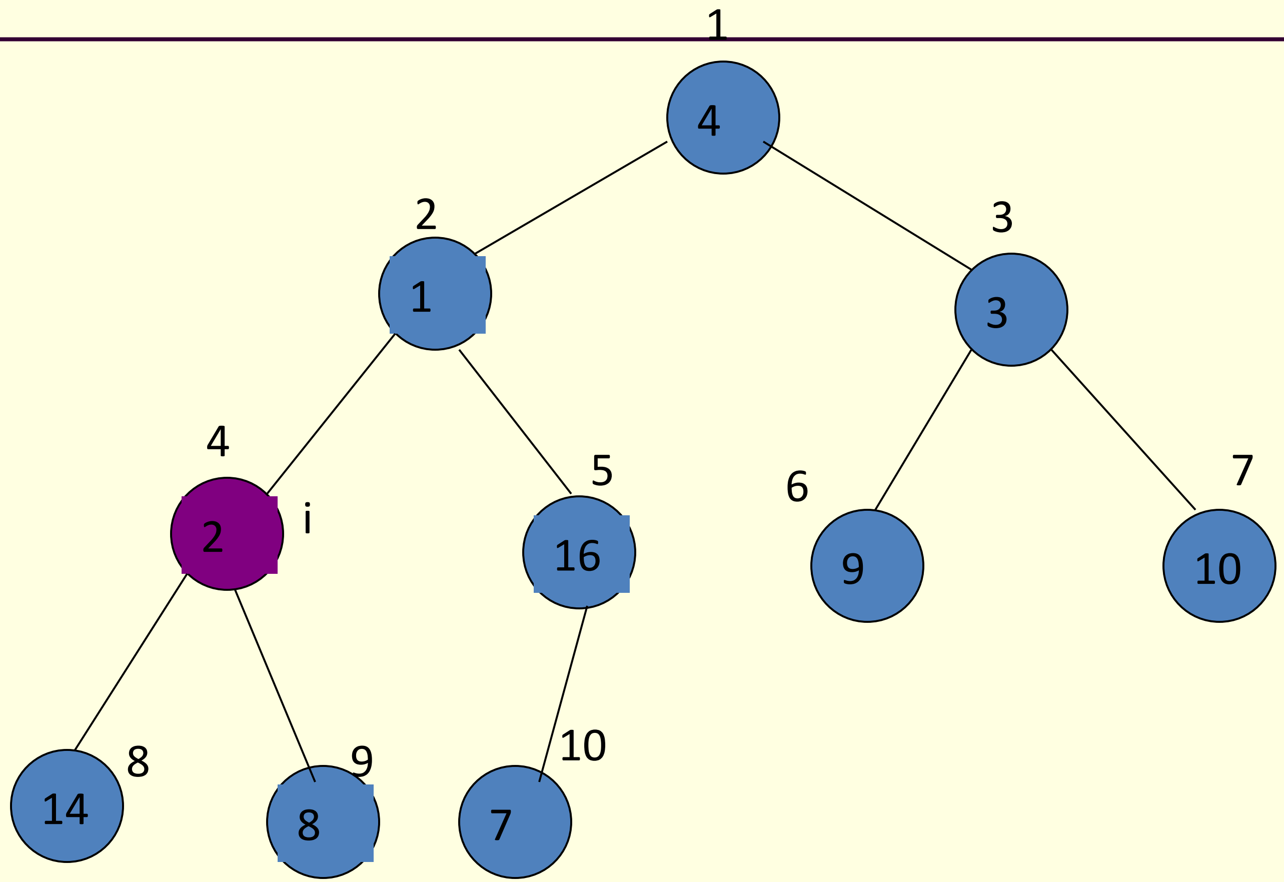
A	4	1	3	2	16	9	10	14	8	7

Building a Heap



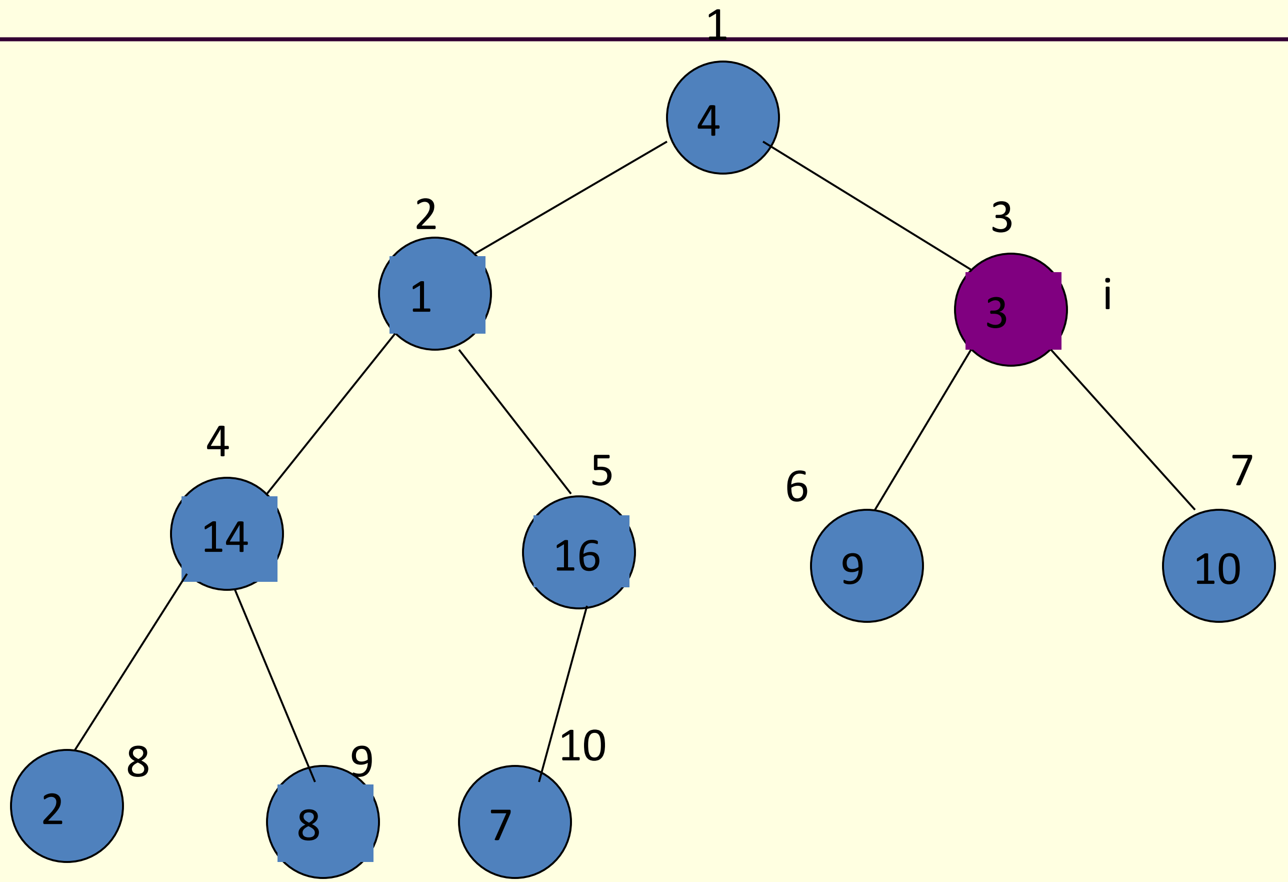
(a)

Building a Heap



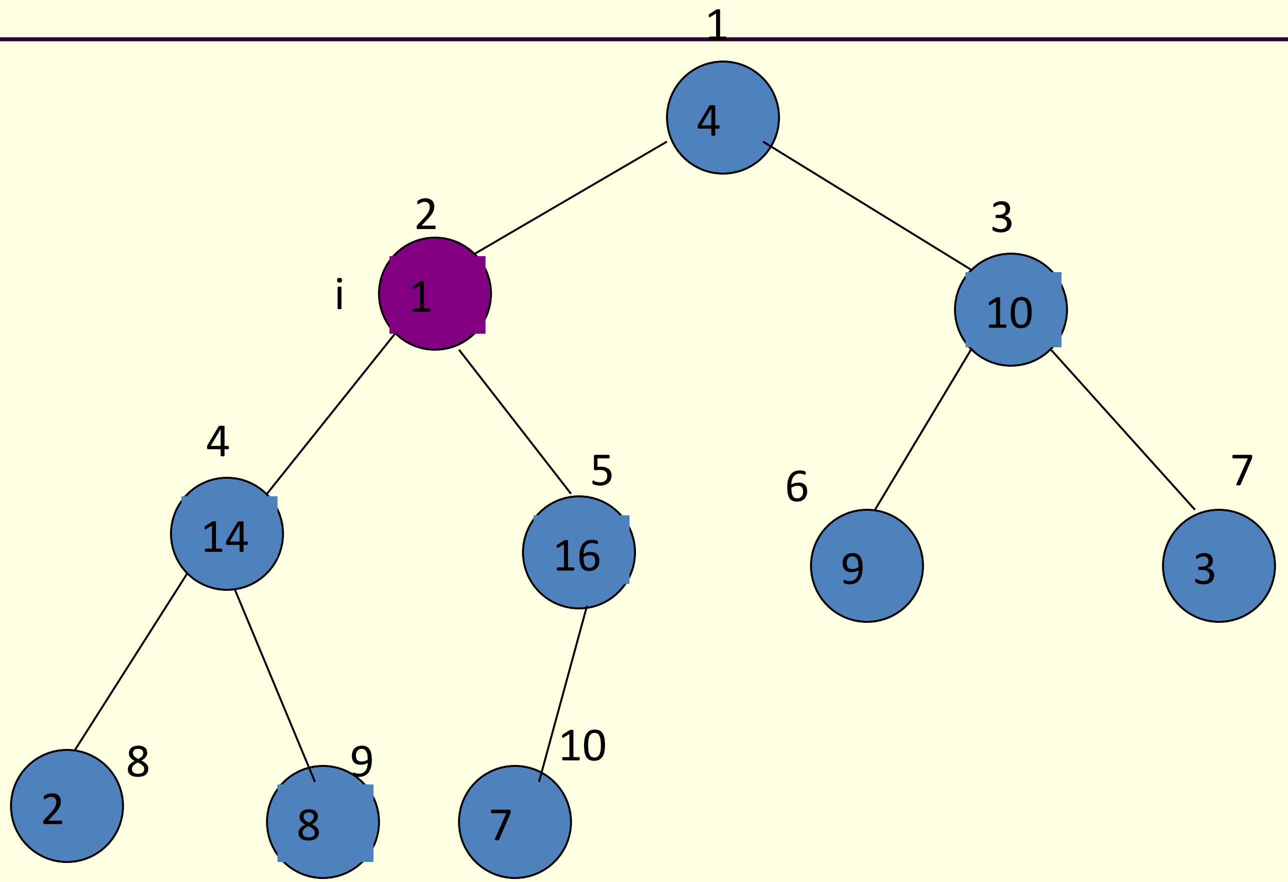
(b)

Building a Heap



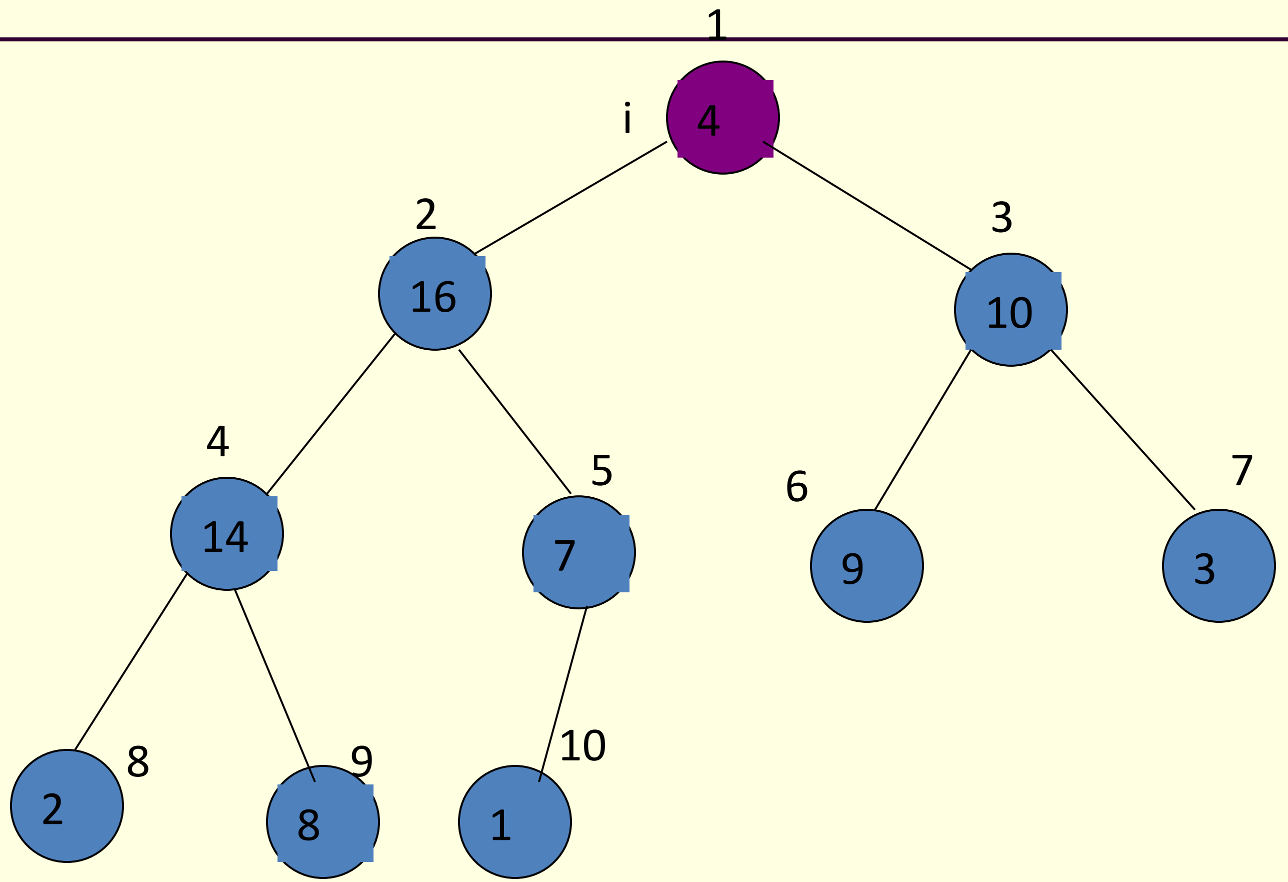
(c)

Building a Heap



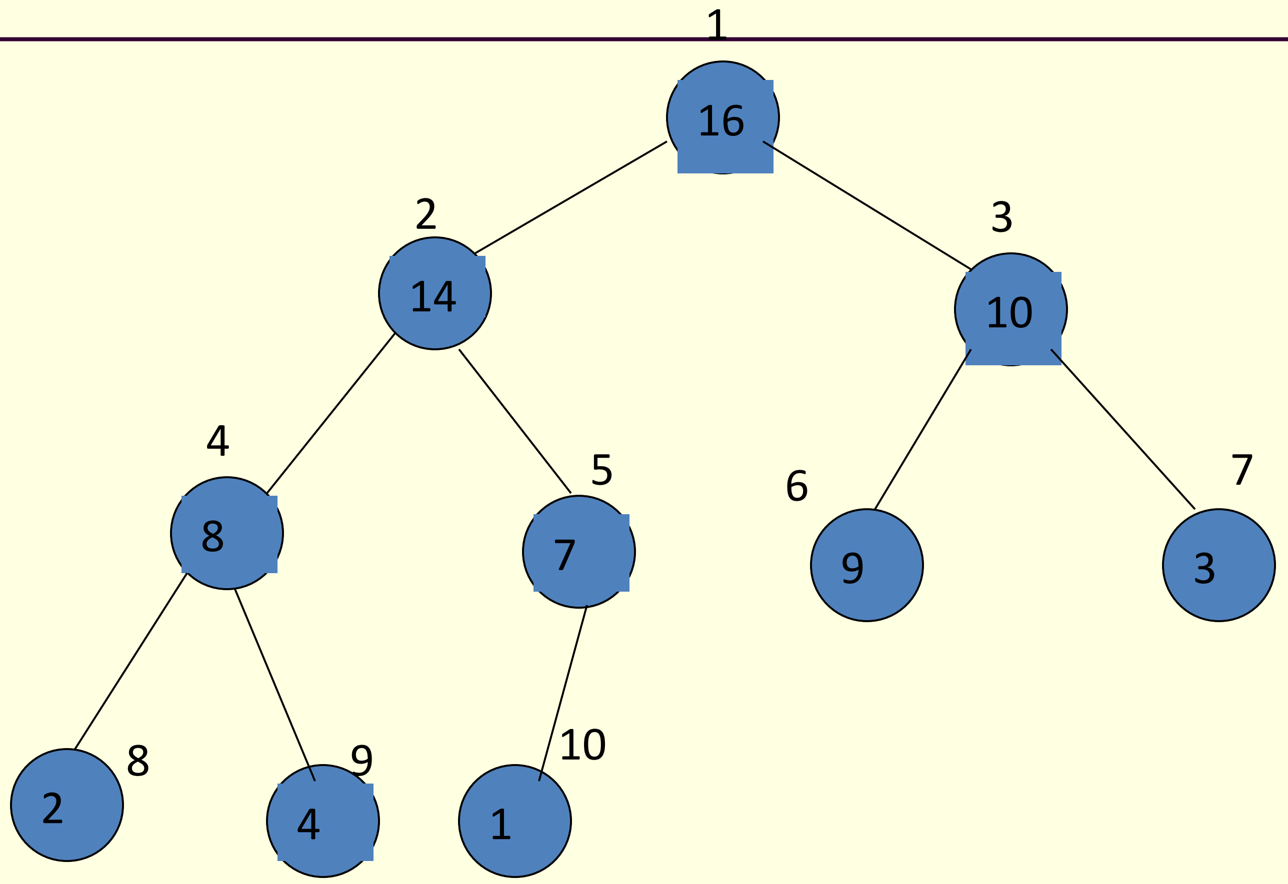
(d)

Building a Heap



(e)

Building a Heap



(f)

Analyzing BuildHeap()

- Each call to **`Heapify()`** takes O(lg *n*) time
- There are O(*n*) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(*n* lg *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?*

David Luebke

Analyzing BuildHeap(): Tight

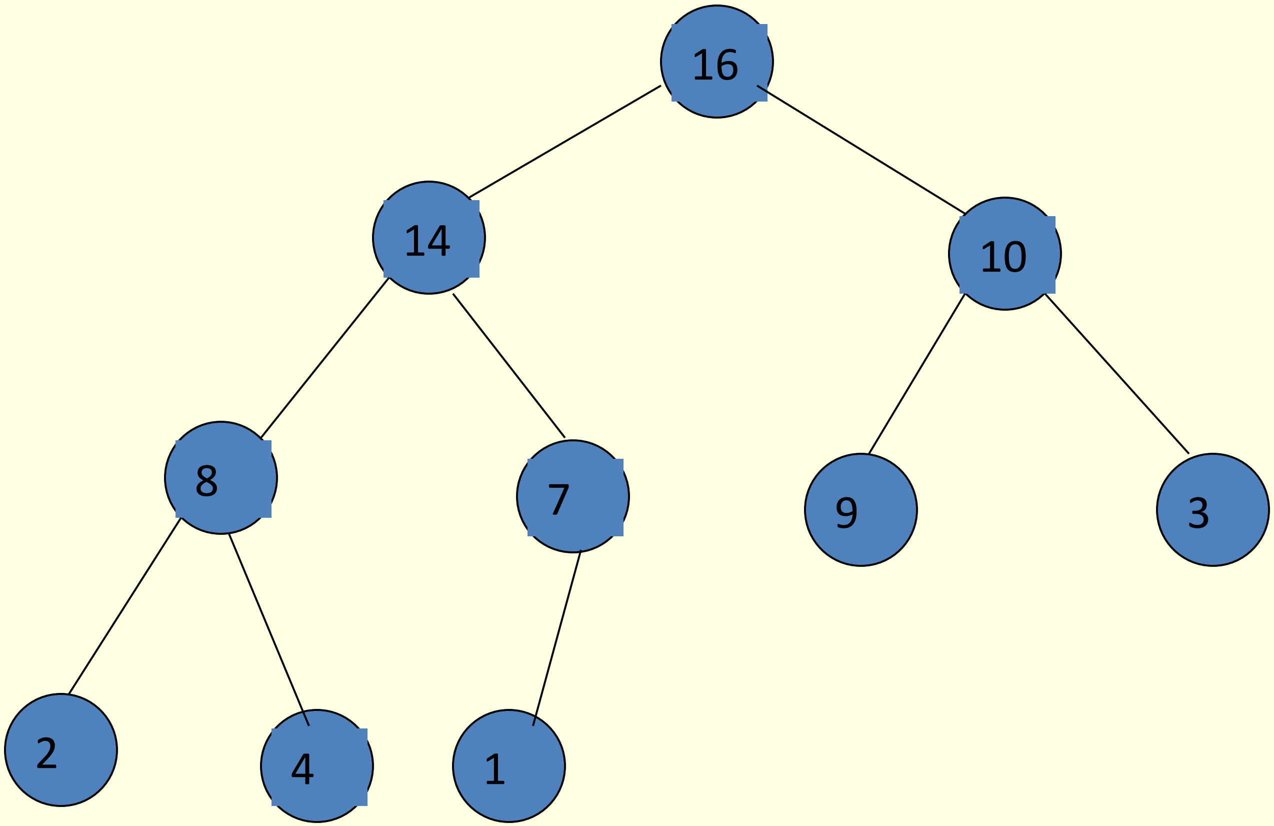
- To **`Heapify()`** a subtree takes O(h) time where h is the height of the subtree
  - h = O(lg m), m = # nodes in subtree
  - The height of most subtrees is small
- Fact: an n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h
- CLR 7.3 uses this fact to prove that **`BuildHeap()`** takes O(n) time

David Luebke

Heapsort Algorithm

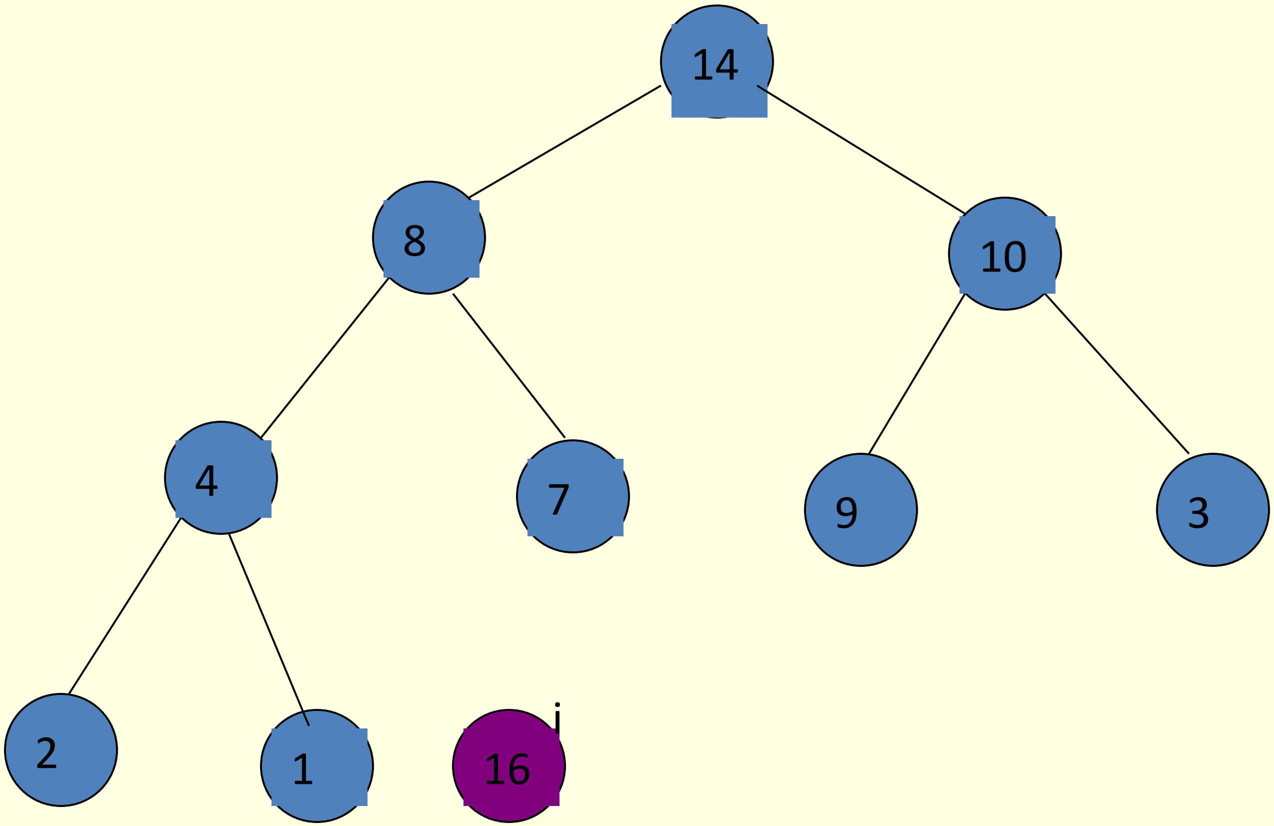
```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  for I ← length[A] downto 2
     do exchange A[1] ← → A[i]
          heap-size[A] ← heap-size[A] -1
          MAX-HEAPIFY(A,1)
```

Heapsort Algorithm



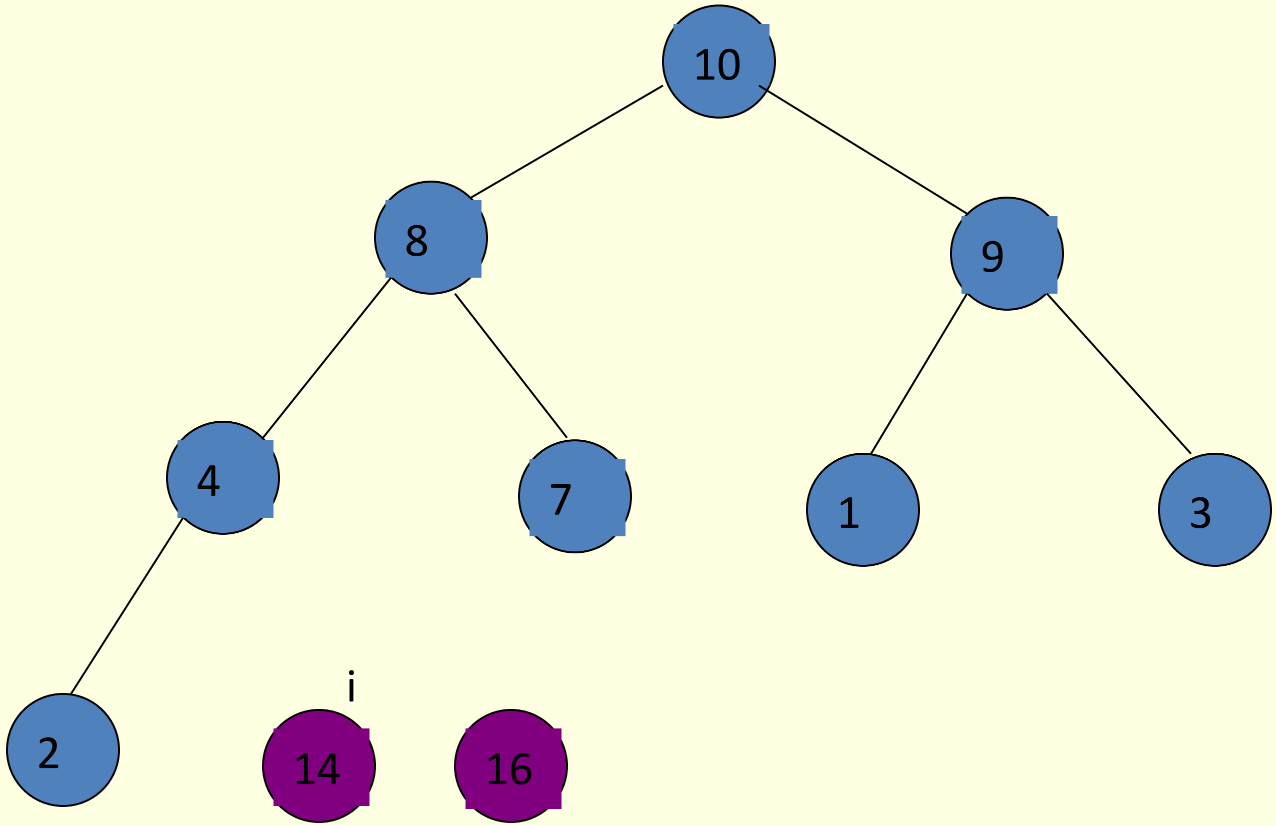
(a)

Heapsort Algorithm



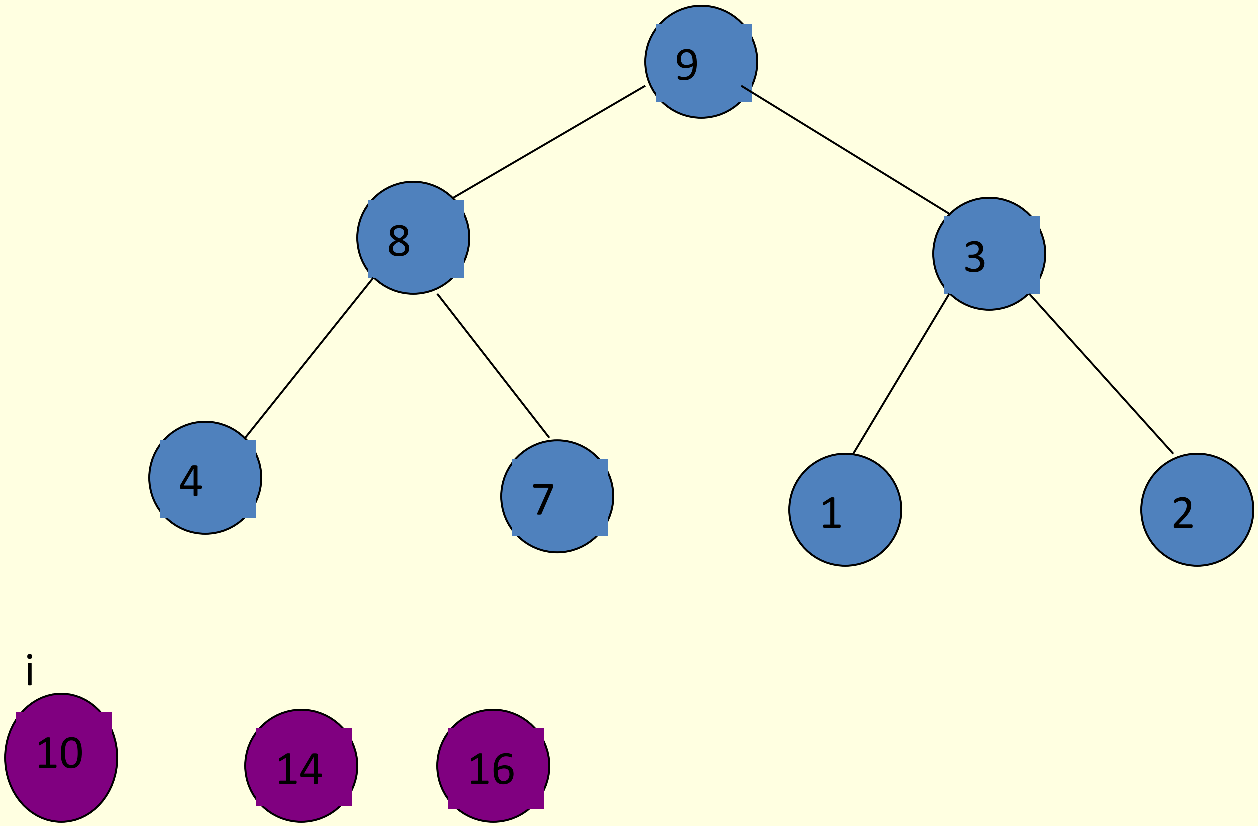
(b)

Heapsort Algorithm



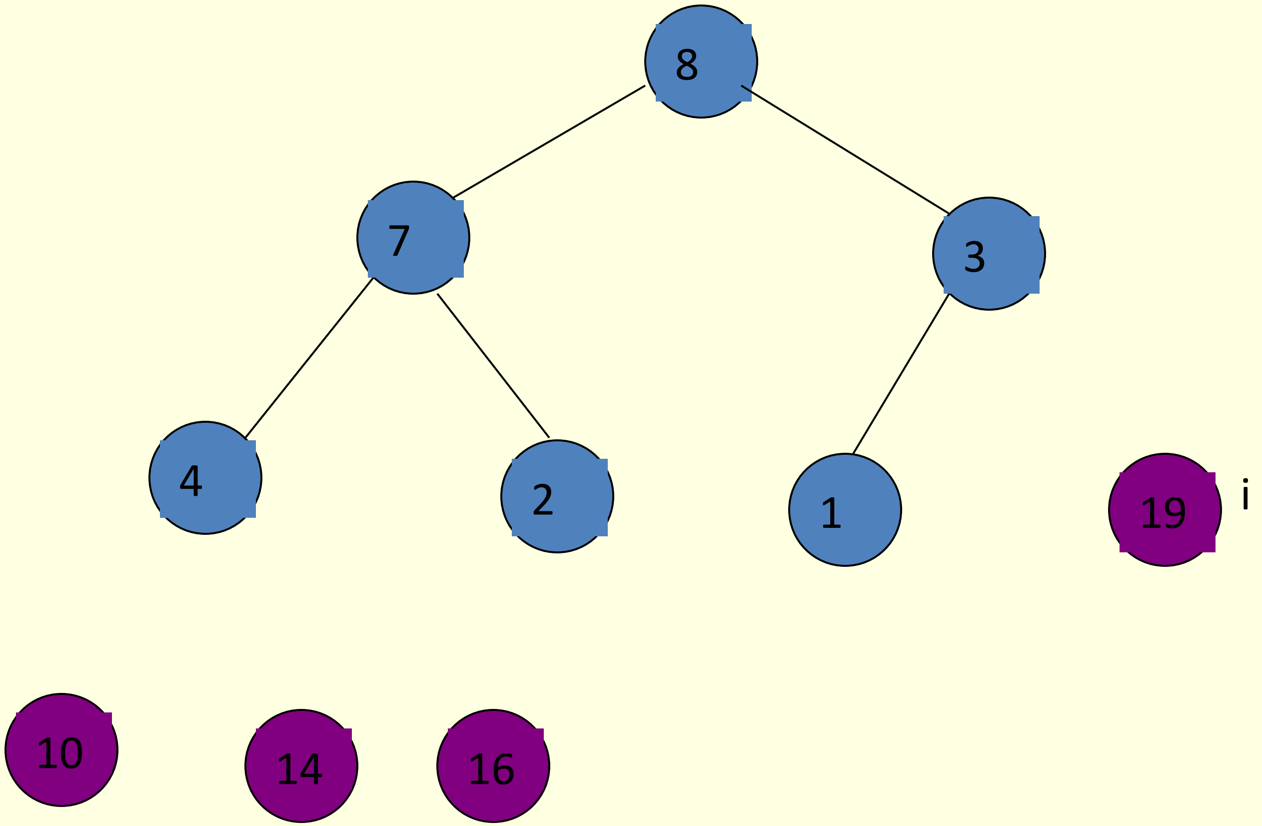
(c)

Heapsort Algorithm



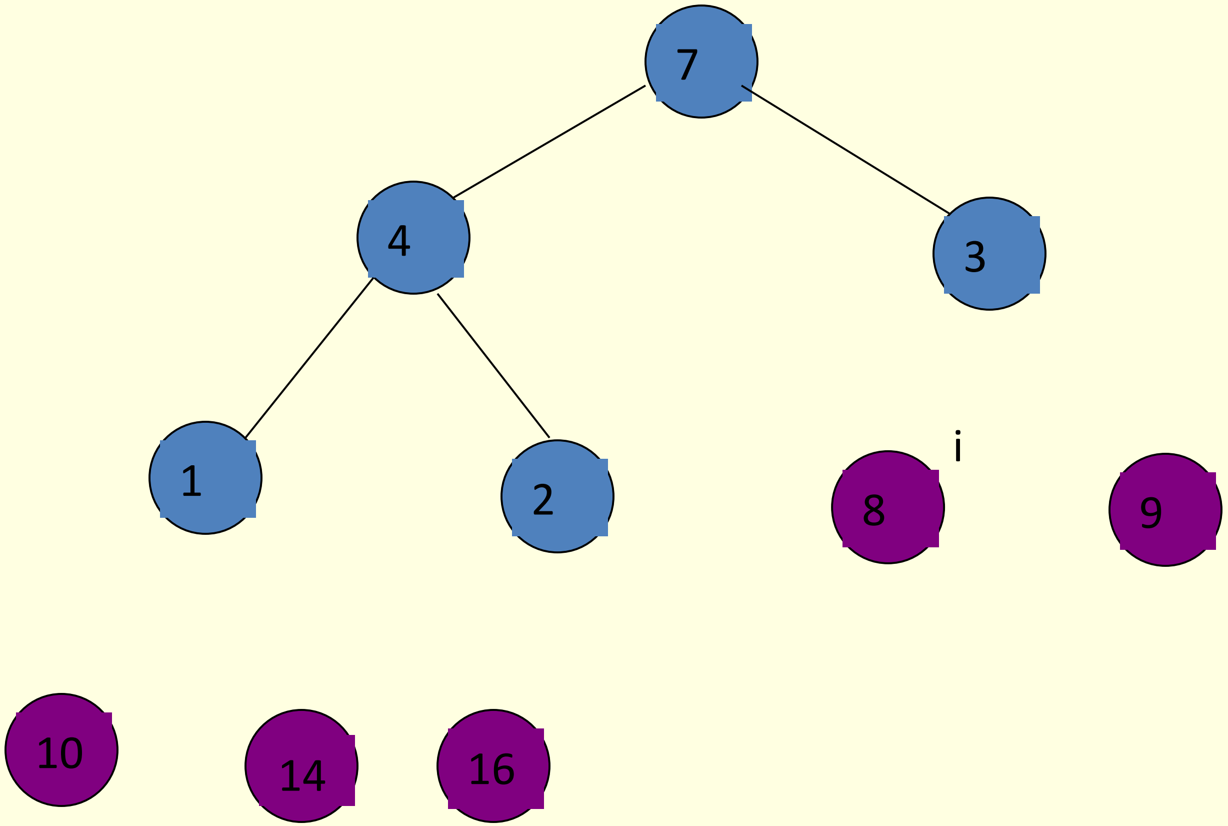
(d)

Heapsort Algorithm



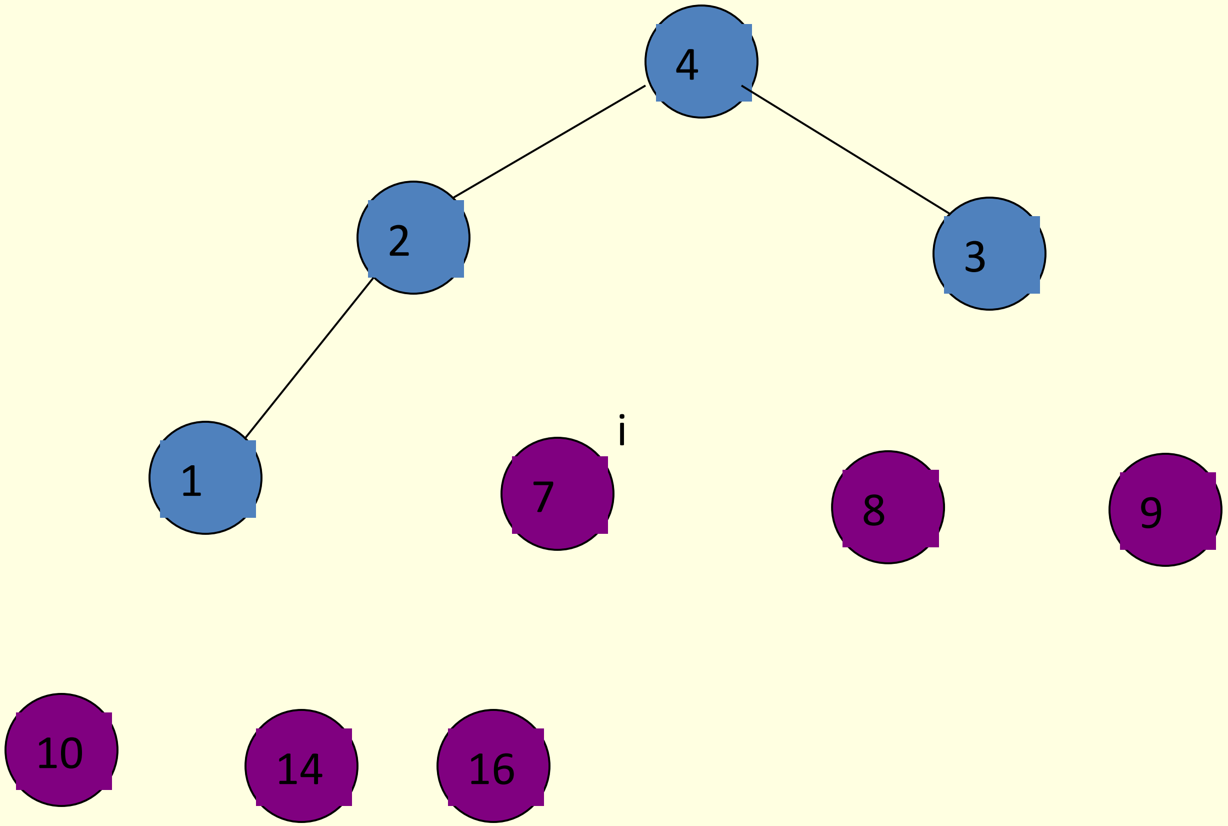
(e)

Heapsort Algorithm



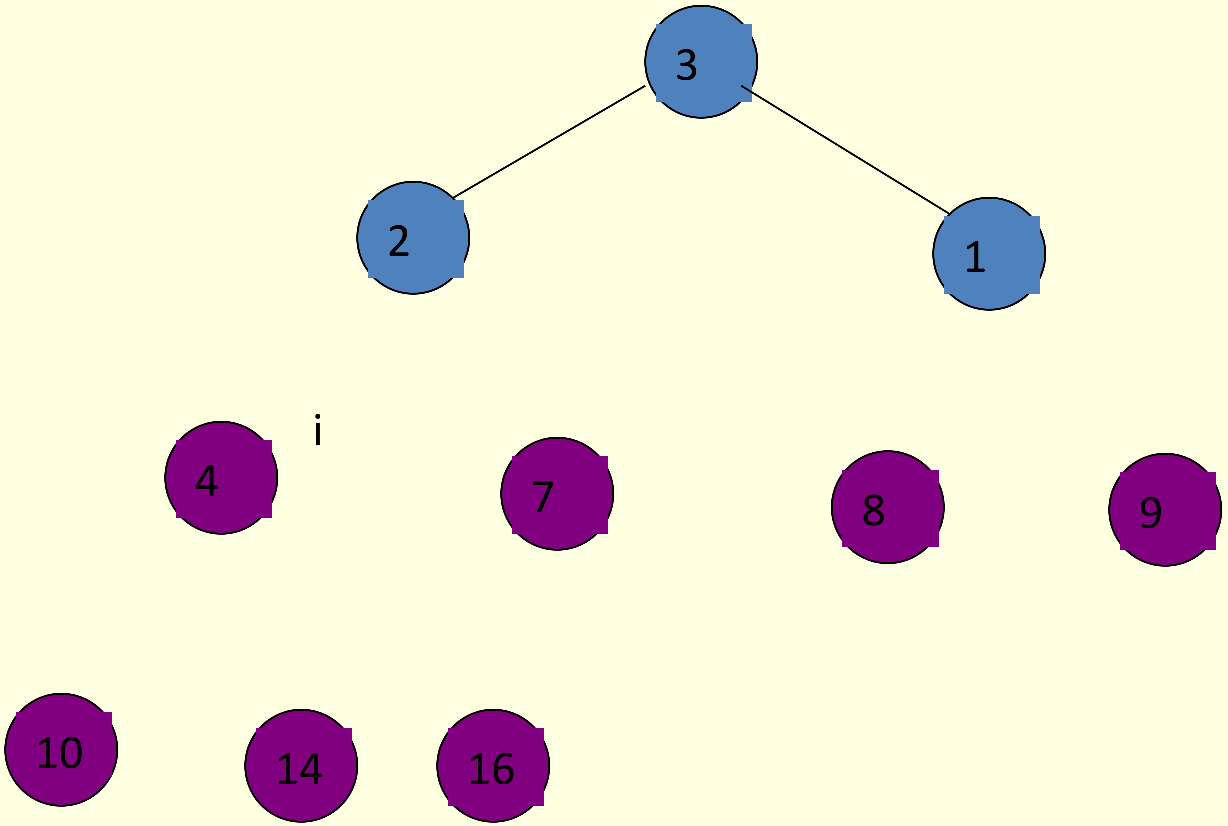
(f)

Heapsort Algorithm



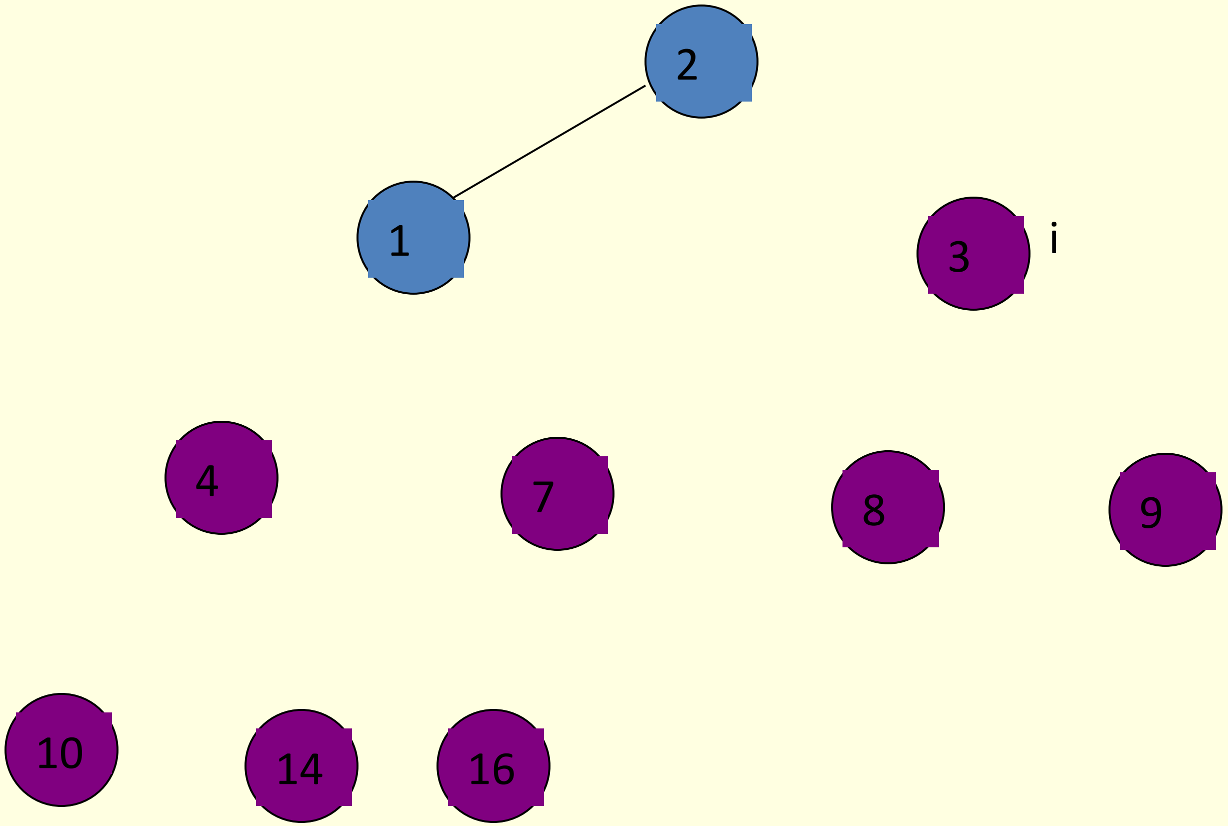
(g)

Heapsort Algorithm



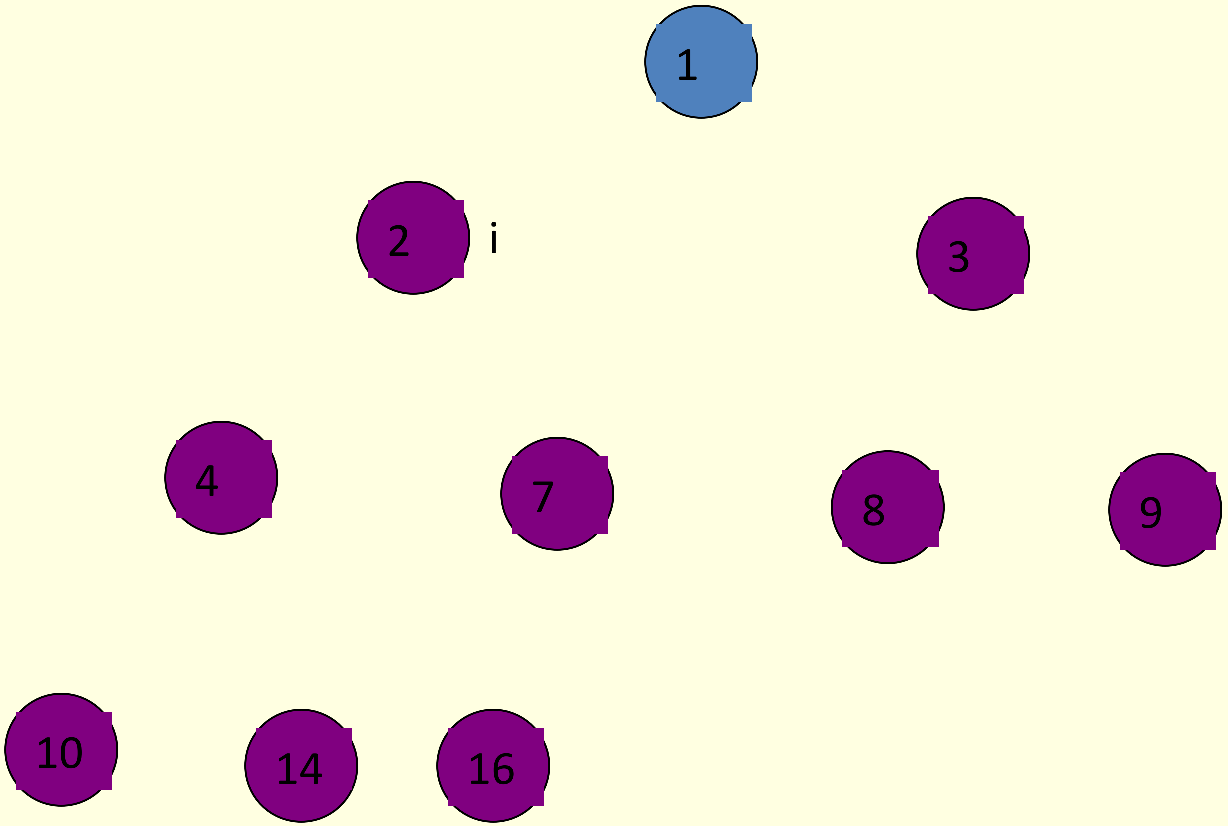
(h)

Heapsort Algorithm



(i)

Heapsort Algorithm



(j)

Heapsort Algorithm

A	1	2	3	4	7	8	9	10	14	16

k

Heap Sort Analysis

- Heap sort is an in-place algorithm.
  Its typical implementation is not stable, but can be made stable .
- **Time Complexity:** Time complexity of max heapify is O(logn). Time complexity of BuildHeap() is O(n) and overall time complexity of Heap Sort is O(nlogn).