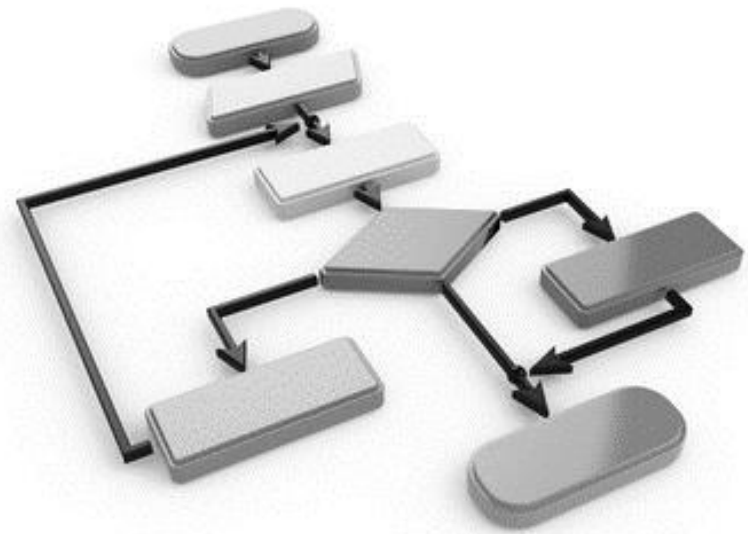


Lecture 04

Divide and Conquer

CSE373: Design and Analysis of Algorithms

Sorting Revisited

So far we've talked about three algorithms to sort an array of numbers

- What is the advantage of merge sort?
  - Answer: O(n lg n) worst-case running time
- What is the advantage of insertion sort?
  - Answer: sorts in place
  - Also: When array "nearly sorted", runs fast in practice

Next on the agenda: *Heapsort*

- Combines advantages of both previous algorithms

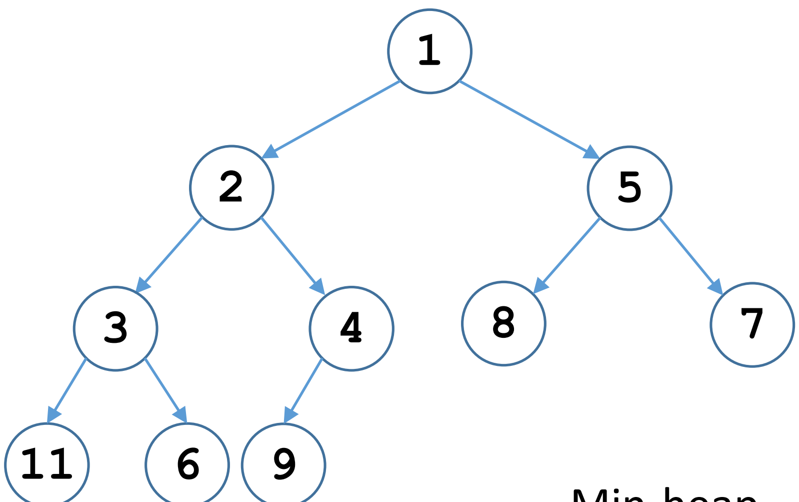
Heaps

- A complete binary tree
  - Each of the elements contains a value that is less than or equal to the value of each of its children (Min-heap)
  - Each of the elements contains a value that is greater than or equal to the value of each of its children (Max-heap)

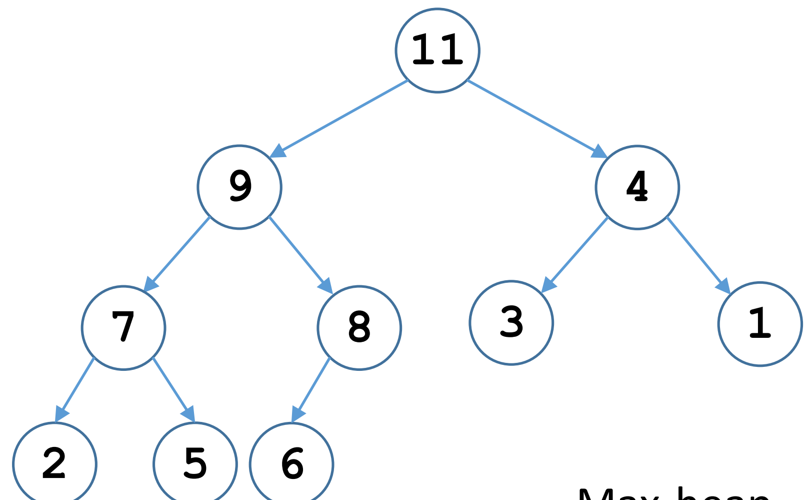
Heaps

A complete binary tree

- Each of the elements contains a value that is less than or equal to the value of each of its children (Min-heap)
- Each of the elements contains a value that is greater than or equal to the value of each of its children (Max-heap)



Min-heap



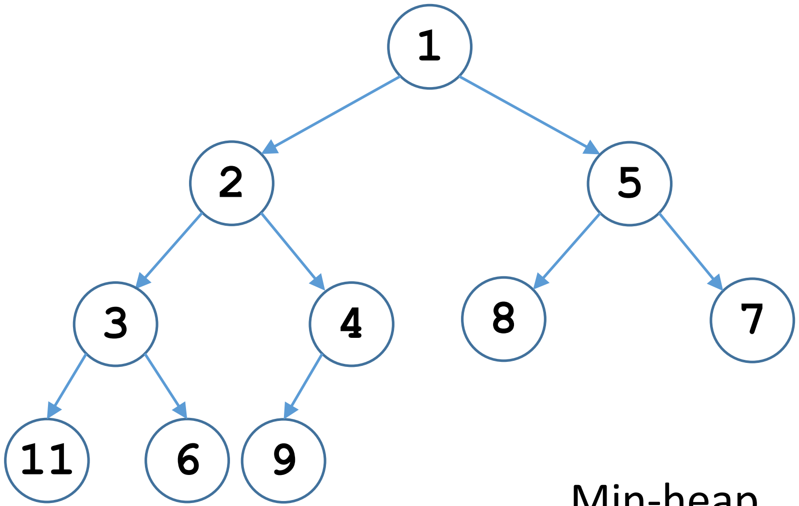
Max-heap

Heaps

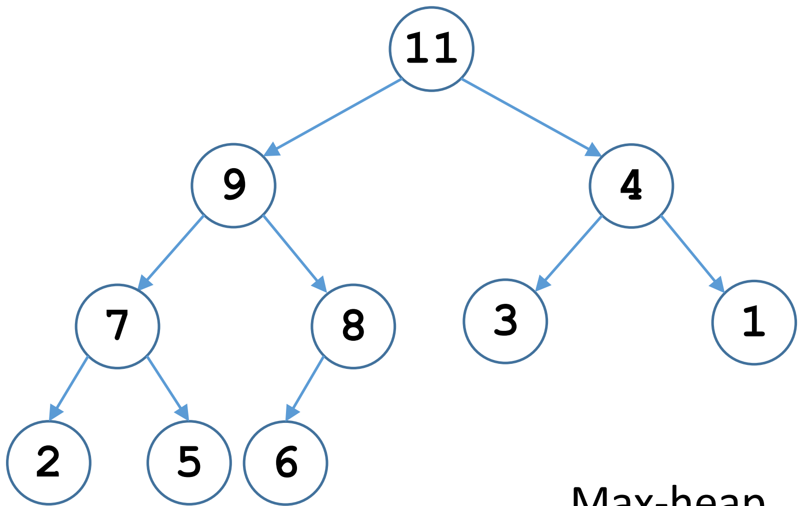
A complete binary tree — Shape property

Each of the elements contains a value that is less than or equal to the value of each of its children (Min-heap)

Each of the elements contains a value that is greater than or equal to the value of each of its children (Max-heap)



Min-heap



Max-heap

- The shape of all heaps with a given number of elements is the same.

Heaps

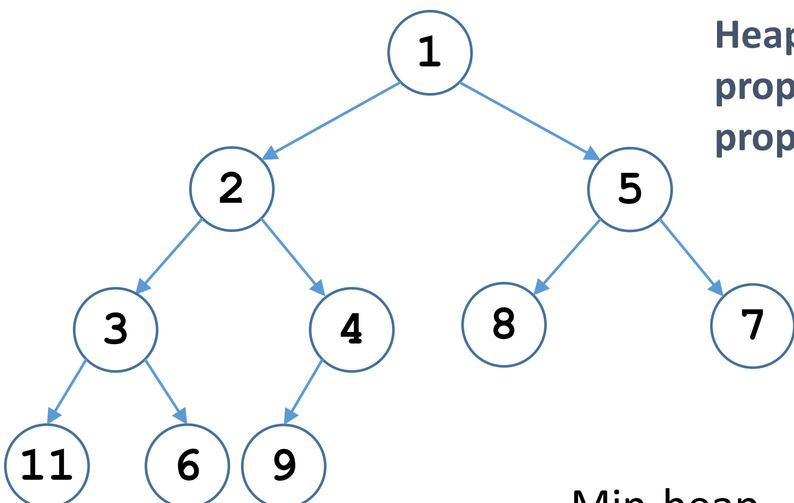
A complete binary tree

Shape property

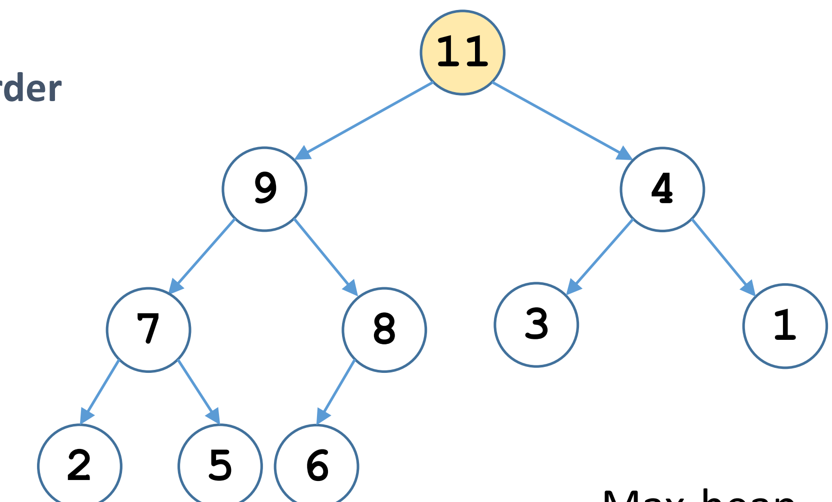
Each of the elements contains a value that is less than or equal to the value of each of its children (Min-heap)

Each of the elements contains a value that is greater than or equal to the value of each of its children (Max-heap)

Heap property/Order property



Min-heap

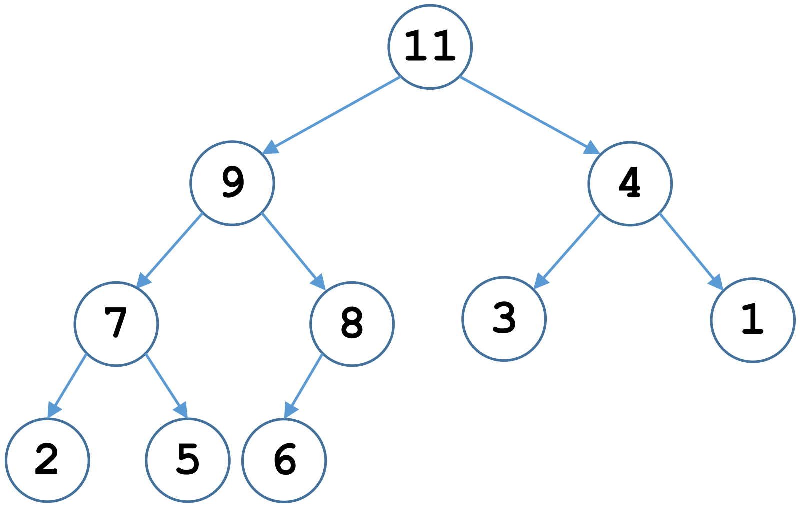


Max-heap

- The shape of all heaps with a given number of elements is the same.
- The root node always contains the largest value in the max-heap (in addition, the subtrees are heaps as well).

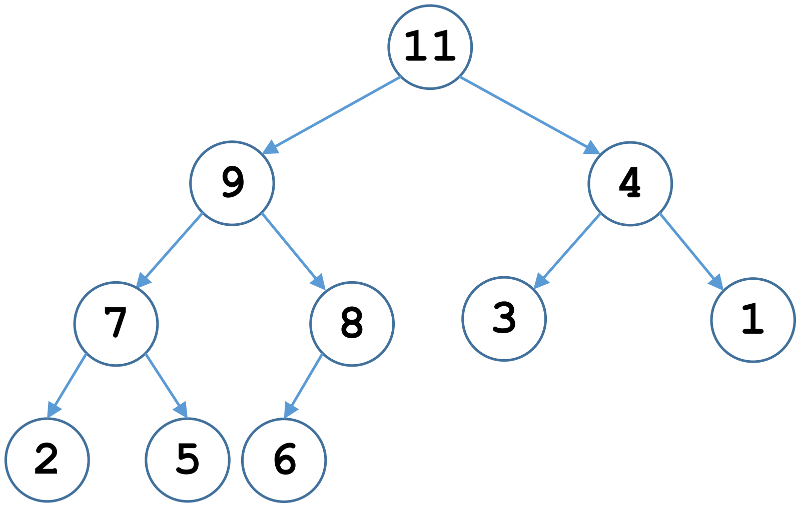
Heaps (Implementation Issue)

Heap elements can be stored as array elements (since the tree is complete, there are not any “holes” in the tree)



Heaps (Implementation Issue)

Heap elements can be stored as array elements (since the tree is complete, there are not any “holes” in the tree)



Index	Value
1	11
2	9
3	4
4	7
5	8
6	3
7	1
8	2
9	5
10	6

Map from array elements to tree nodes and vice versa

Root – $A[1]$

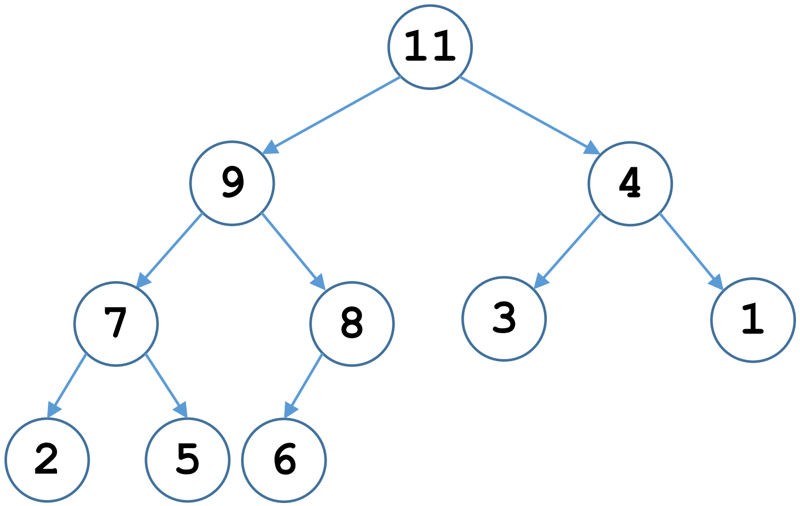
Left[i] – $A[2i]$

Right[i] – $A[2i+1]$

Parent[i] – $A[\lfloor i/2 \rfloor]$

Heaps (Implementation Issue)

Heap elements can be stored as array elements (since the tree is complete, there are not any "holes" in the tree)



Index	Value
1	11
2	9
3	4
4	7
5	8
6	3
7	1
8	2
9	5
10	6

length[*A*] – number of elements in array *A.*
heap-size[*A*] – number of elements in heap stored in *A.*

heap-size[*A*] $\leq$ length[*A*]

No. of leaves = $\lceil n/2 \rceil$

Height of a heap = $\lfloor lg\ n \rfloor$

Heap Operations: Heapify()

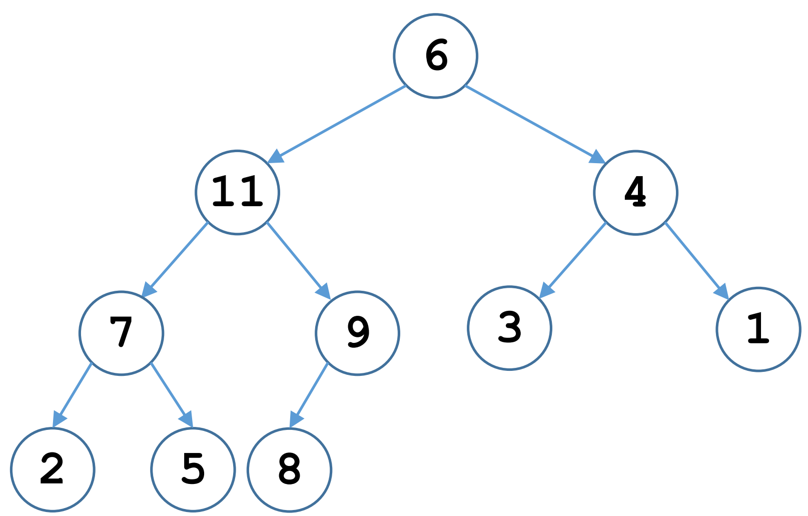
`Heapify()`: maintain the heap property

- **Given:** a node *i* in the heap with children *l* and *r*
- **Given:** two subtrees rooted at *l* and *r*, assumed to be heaps
- **Problem:** The subtree rooted at *i* may violate the heap property **Action:** let the value of the parent node "float down" so subtree at *i* satisfies the heap property
  - May lead to the subtree at the child not being a heap.
  - Recursively fix the children until all of them satisfy the max-heap property.

The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

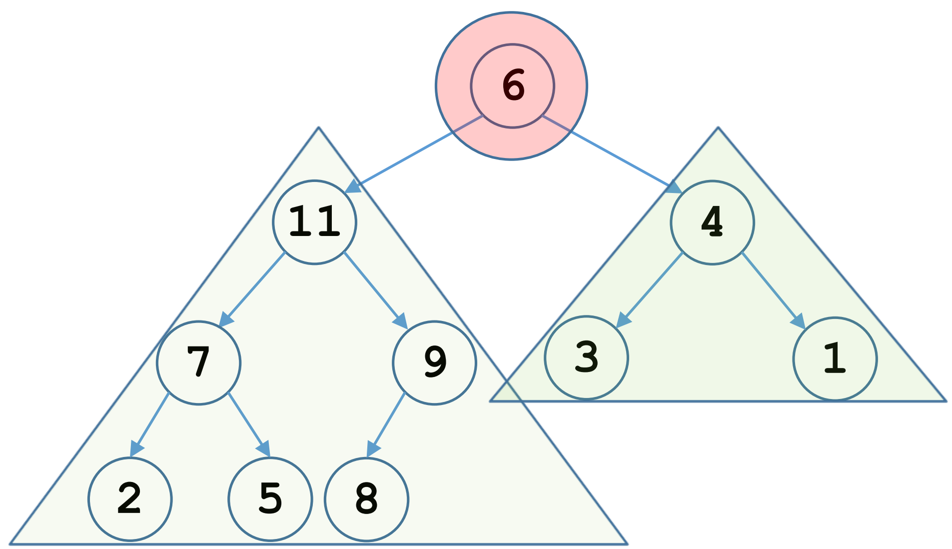
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

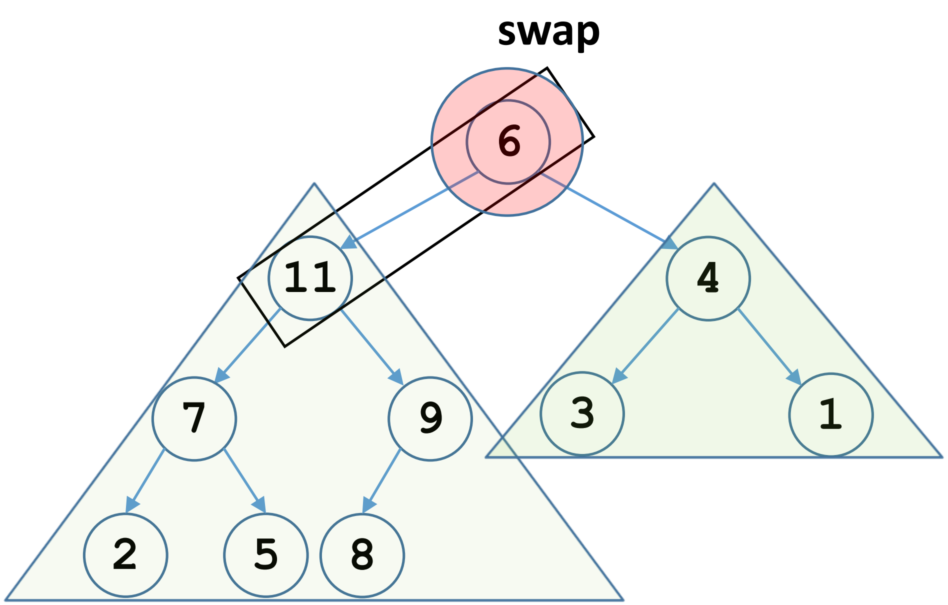
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

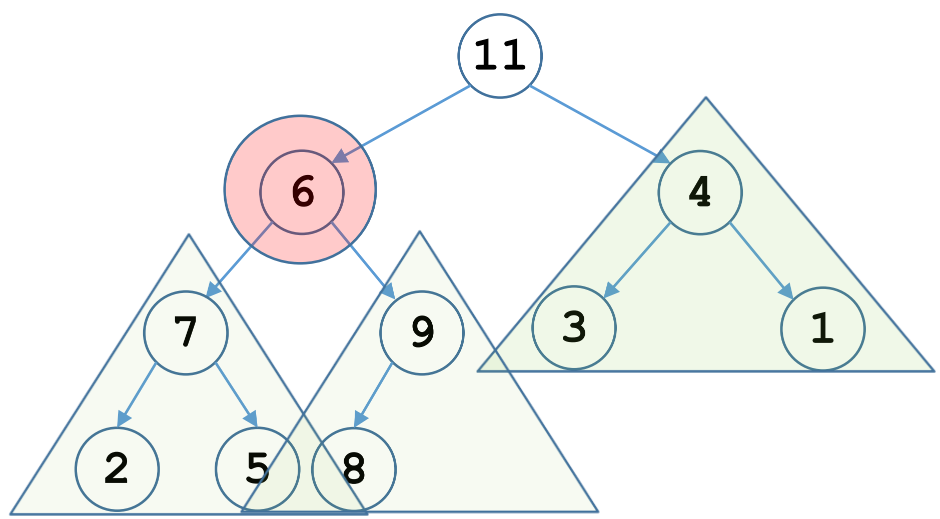
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

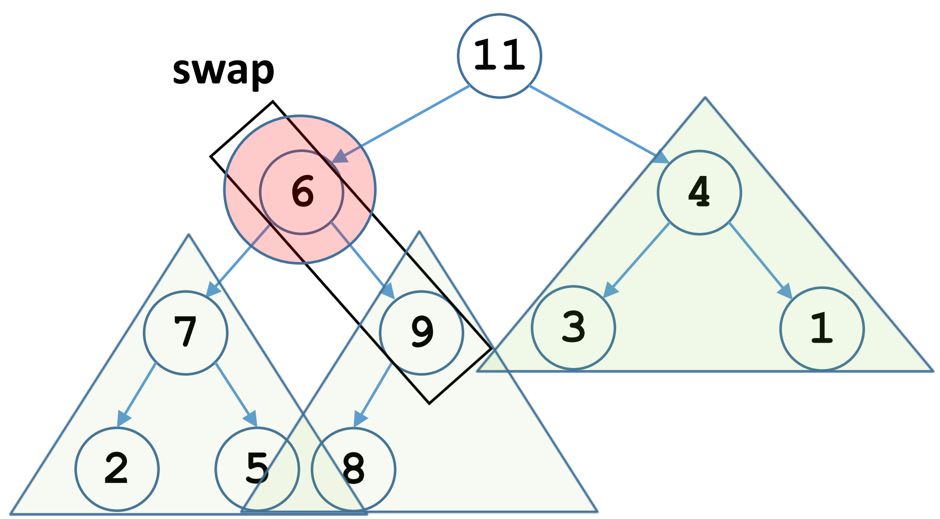
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

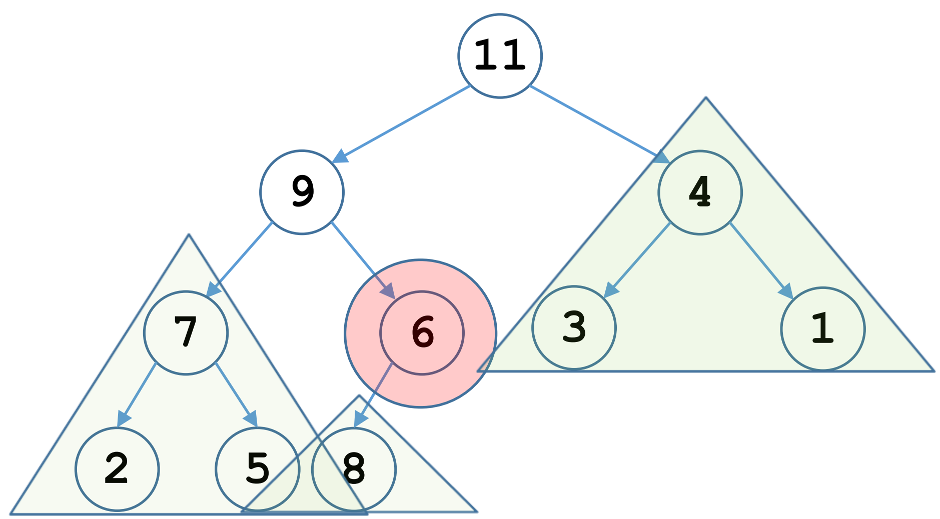
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

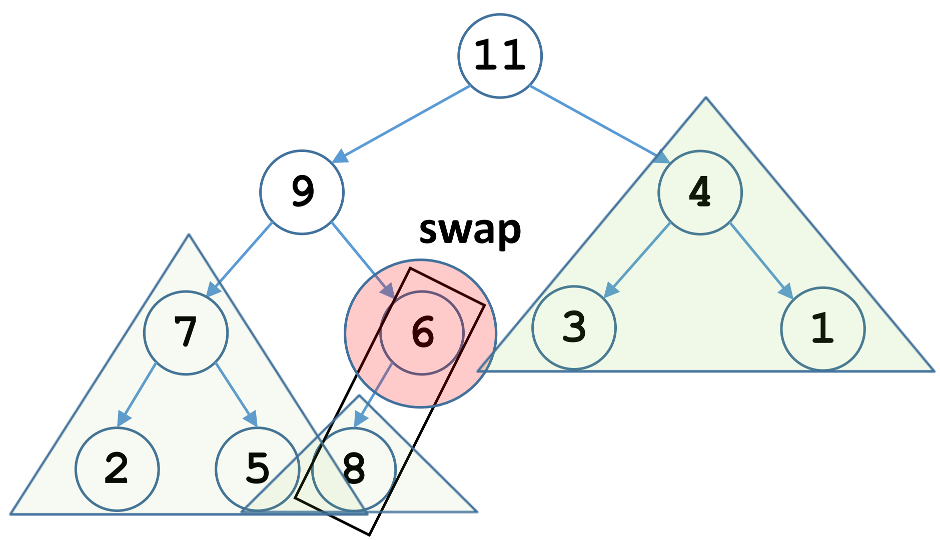
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

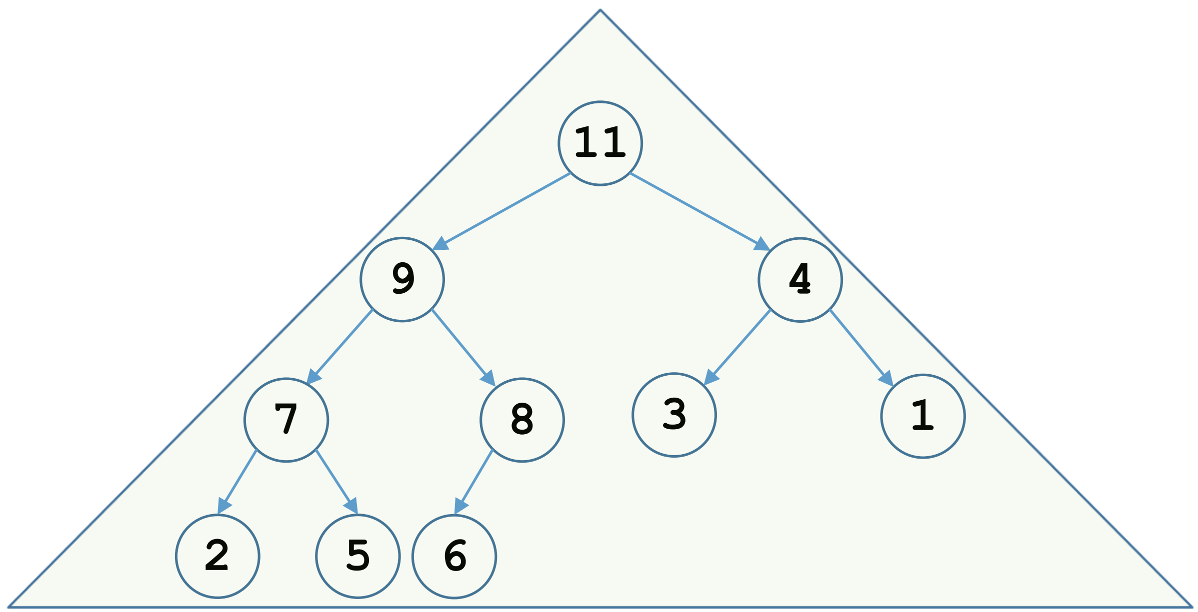
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



Procedure MaxHeapify

<u>MaxHeapify(A, i)</u>

```
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]
            MaxHeapify(A, largest)
```

<u>Assumption:</u>
Left(i) and Right(i) are max-heaps.

Procedure MaxHeapify

<u>MaxHeapify(A, i)</u>

1. l ← left(i)
2. r ← right(i)
3. **if** l ≤ heap-size[A] and A[l] > A[i]
4. **then** largest ← l
5. **else** largest ← i
6. **if** r ≤ heap-size[A] **and** A[r] > A[largest]
7. **then** largest ← r
8. **if** largest ≠ i
9. **then** exchange A[i] ↔ A[largest]
10. MaxHeapify(A, largest)

Time to fix node i and its children = **Θ(1)**

PLUS

Time to fix the subtree rooted at one of i's children = T(size of subree at largest)

Running Time for MaxHeapify(A, n)

$T(n) = T(largest) + \Theta(1)$

$largest \leq 2n/3$ (worst case occurs when the last row of tree is exactly half full)

$T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$

Alternately, MaxHeapify takes $O(h)$ where h is the height of the node where MaxHeapify is applied

Building a heap

Use *MaxHeapify* to convert an array *A* into a max-heap.

How?

Call MaxHeapify on each element in a bottom-up manner.

BuildMaxHeap(A)

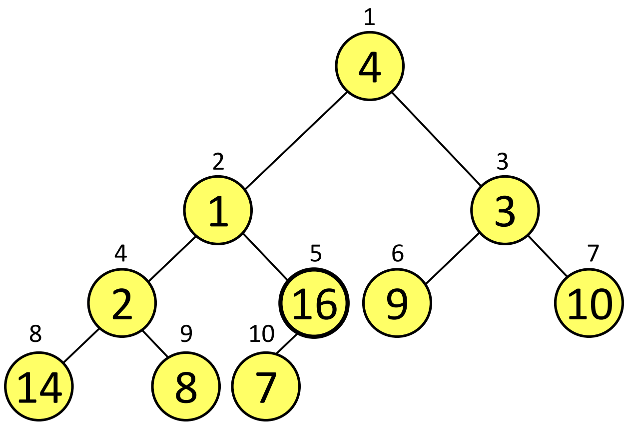
1. heap-size[A] $\leftarrow$ length[A]
2. **for** i $\leftarrow \lfloor$length[A]/2$\rfloor$ **downto** 1
3. **do** MaxHeapify(A, i)

Heap Sort

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

Heap Sort

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

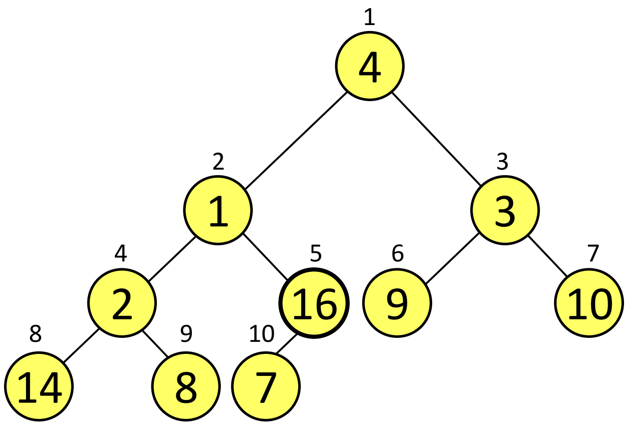


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] ← length[A]
2. **for** i ← ⌊length[A]/2⌋ **downto** 1
3.   **do** MaxHeapify(A, i)

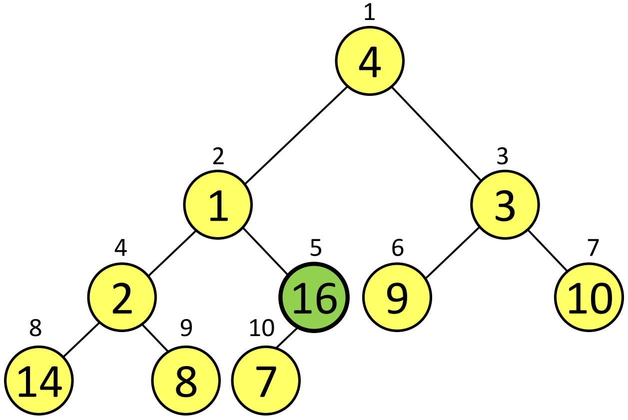


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] ← length[A]
2. **for** i ← $\lfloor$length[A]/2$\rfloor$ **downto** 1
3. **do** MaxHeapify(A, i)

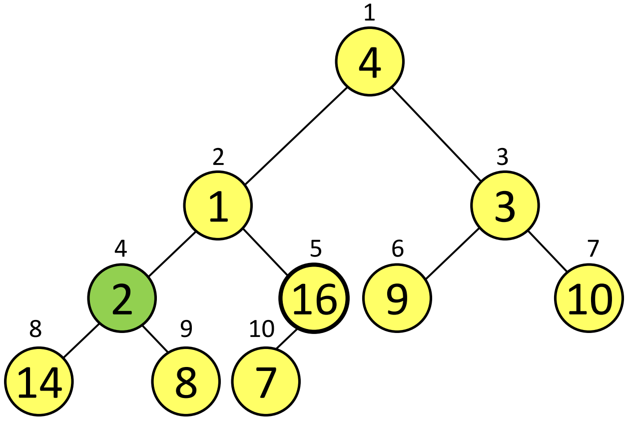


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] ← length[A]
2. **for** i ← $\lfloor$length[A]/2$\rfloor$ **downto** 1
3. **do** MaxHeapify(A, i)

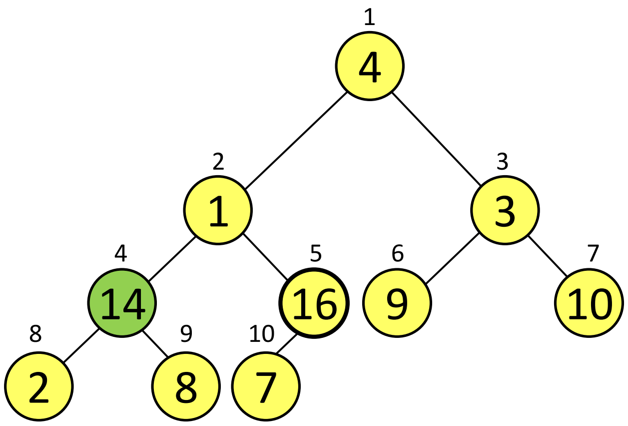


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] $\leftarrow$ length[A]
2. **for** i $\leftarrow \lfloor$length[A]/2$\rfloor$ **downto** 1
3. **do** MaxHeapify(A, i)

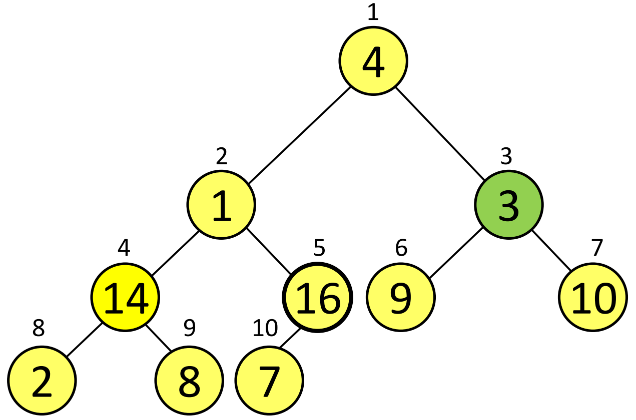


Heap Sort

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

<ins>BuildMaxHeap(A)</ins>

1. heap-size[A] ← length[A]
2. **for** i ← ⌊length[A]/2⌋ **downto** 1
3.   **do** MaxHeapify(A, i)

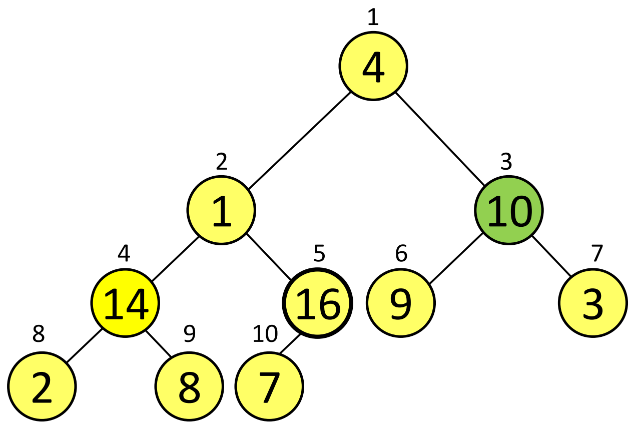


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] $\leftarrow$ length[A]
2. **for** i $\leftarrow \lfloor$length[A]/2$\rfloor$ **downto** 1
3.   **do** MaxHeapify(A, i)

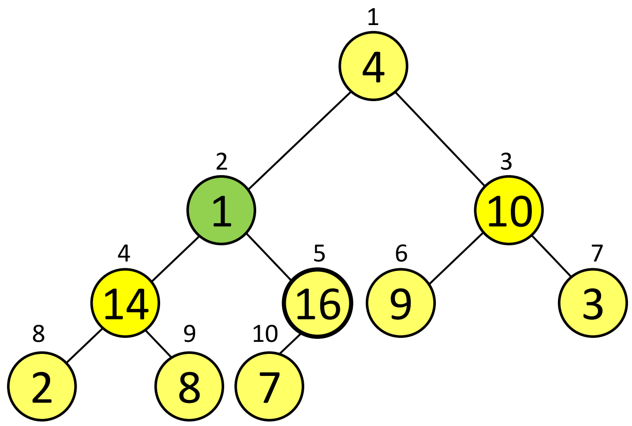


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] ← length[A]
2. **for** i ← $\lfloor$length[A]/2$\rfloor$ **downto** 1
3.   **do** MaxHeapify(A, i)

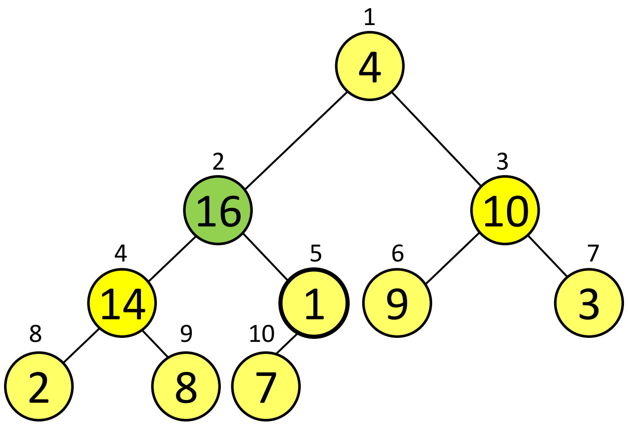


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] ← length[A]
2. **for** i ← ⌊length[A]/2⌋ **downto** 1
3.   **do** MaxHeapify(A, i)

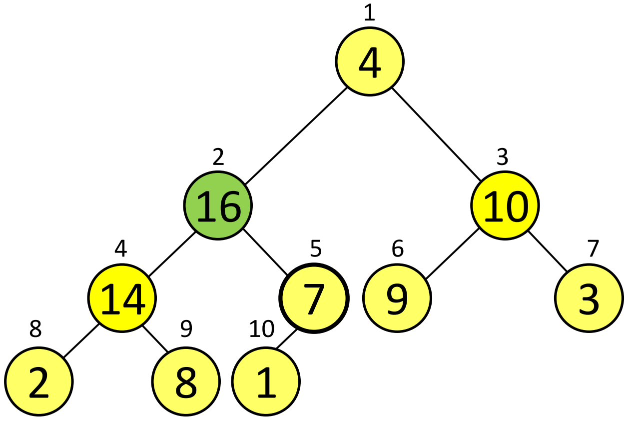


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] ← length[A]
2. **for** i ← ⌊length[A]/2⌋ **downto** 1
3. **do** MaxHeapify(A, i)

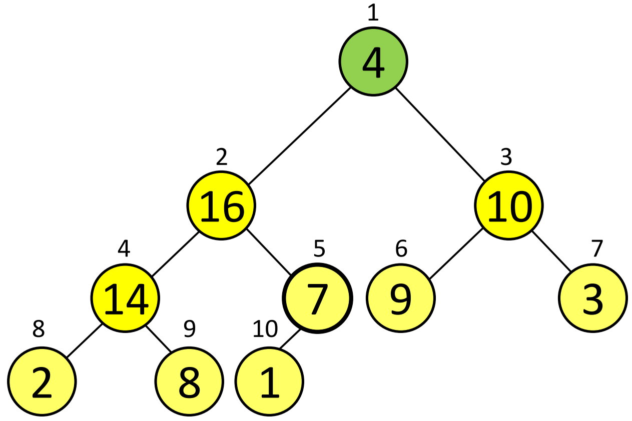


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] ← length[A]
2. **for** i ← $\lfloor$length[A]/2$\rfloor$ **downto** 1
3. **do** MaxHeapify(A, i)

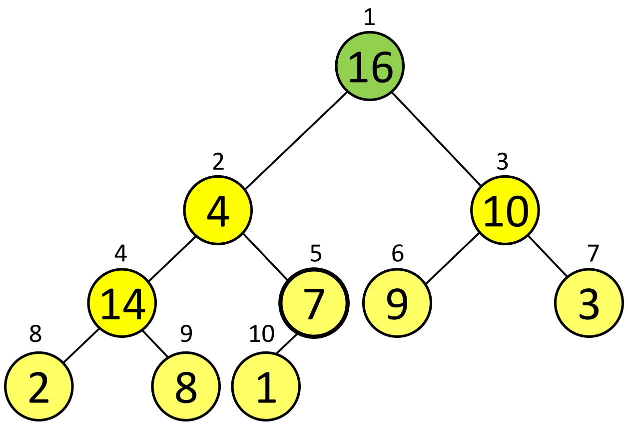


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] $\leftarrow$ length[A]
2. **for** i $\leftarrow \lfloor$length[A]/2$\rfloor$ **downto** 1
3. **do** MaxHeapify(A, i)

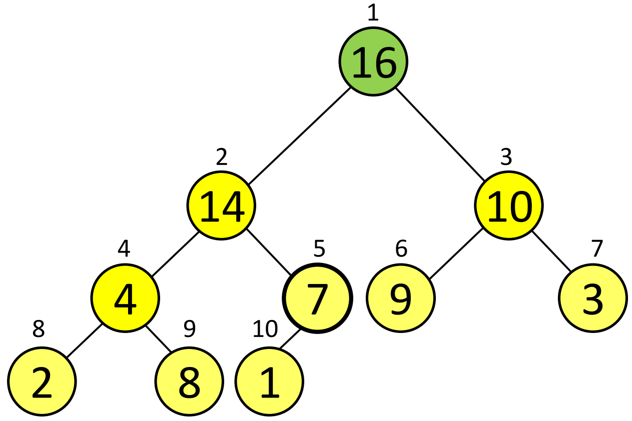


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] $\leftarrow$ length[A]
2. **for** i $\leftarrow \lfloor$length[A]/2$\rfloor$ **downto** 1
3. **do** MaxHeapify(A, i)

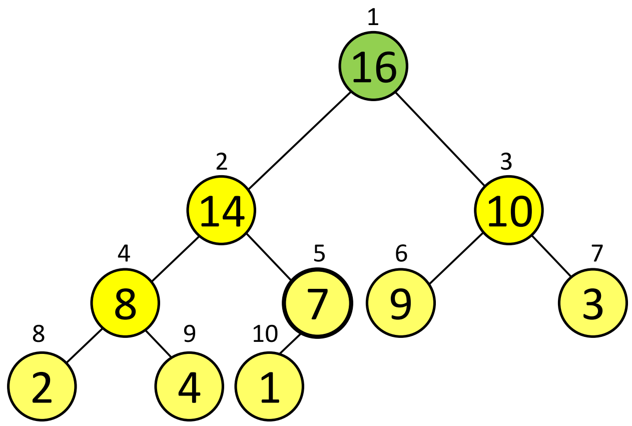


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] $\leftarrow$ length[A]
2. **for** i $\leftarrow \lfloor$length[A]/2$\rfloor$ **downto** 1
3.   **do** MaxHeapify(A, i)

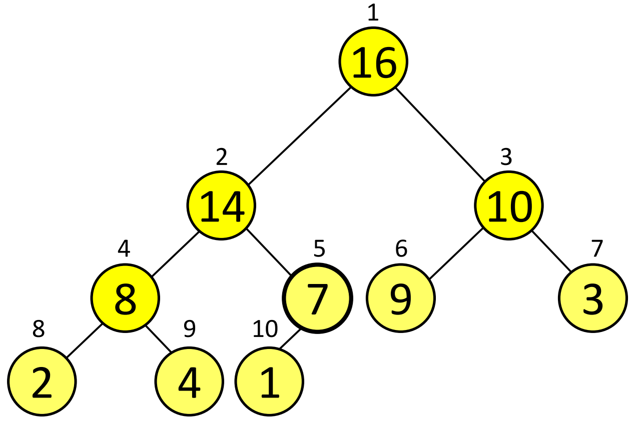


Heap Sort

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

<u>BuildMaxHeap(A)</u>

1. heap-size[A] ← length[A]
2. **for** i ← $\lfloor$length[A]/2$\rfloor$ **downto** 1
3.   **do** MaxHeapify(A, i)



Heap Sort

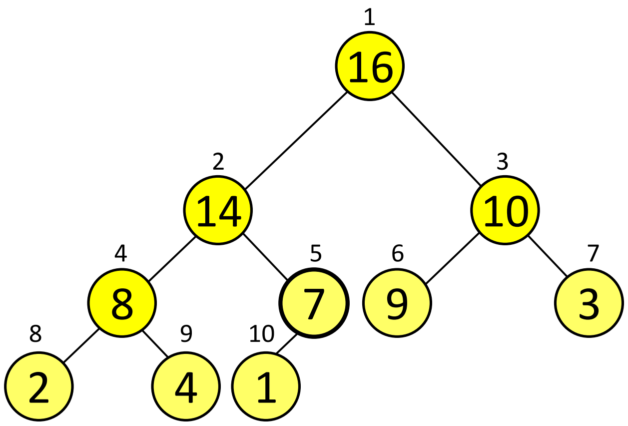
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

<u>BuildMaxHeap(A)</u>

1. heap-size[A] ← length[A]
2. **for** i ← ⌊length[A]/2⌋ **downto** 1
3.   **do** MaxHeapify(A, i)

O(N)

O(logN)



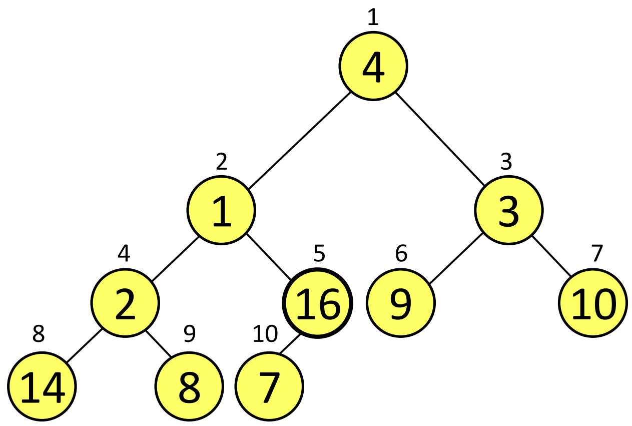
Time complexity: O(N logN) [upper bound]
O(N) [tighter bound]

Heap Sort

```
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
        MaxHeapify(A, 1)
```

Heap Sort

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

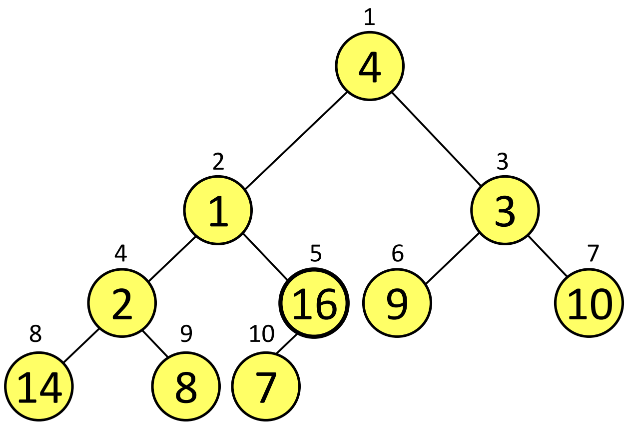


Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3.   **do** exchange A[1] ↔ A[i]
4.   heap-size[A] ← heap-size[A] – 1
5.   MaxHeapify(A, 1)

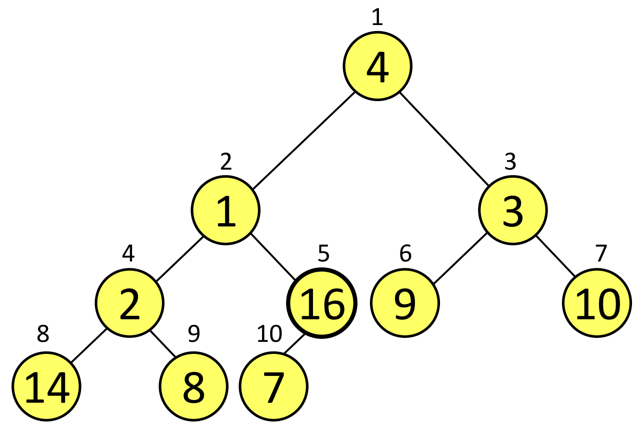


Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)

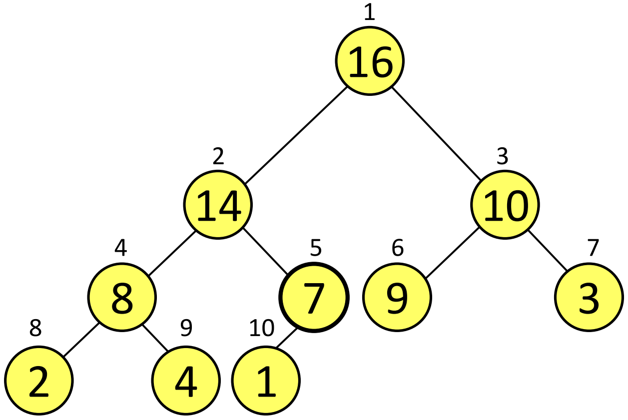


Heap Sort

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

<u>HeapSort(A)</u>

1. $\boxed{\text{Build-Max-Heap(A)}}$
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)

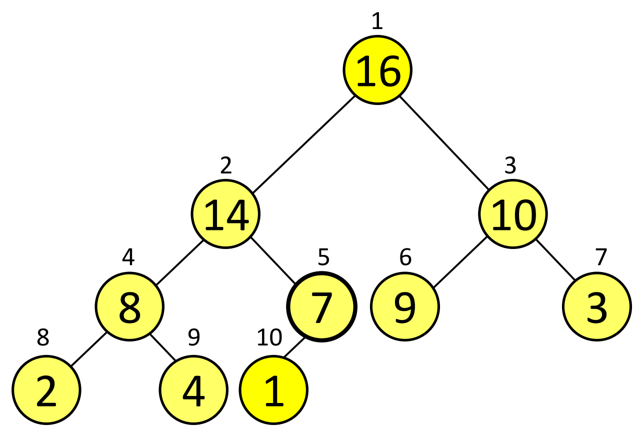


Heap Sort

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

<u>HeapSort(A)</u>

```
Build-Max-Heap(A)
for i ← length[A] downto 2
    do exchange A[1] ↔ A[i]
        heap-size[A] ← heap-size[A] – 1
        MaxHeapify(A, 1)
```

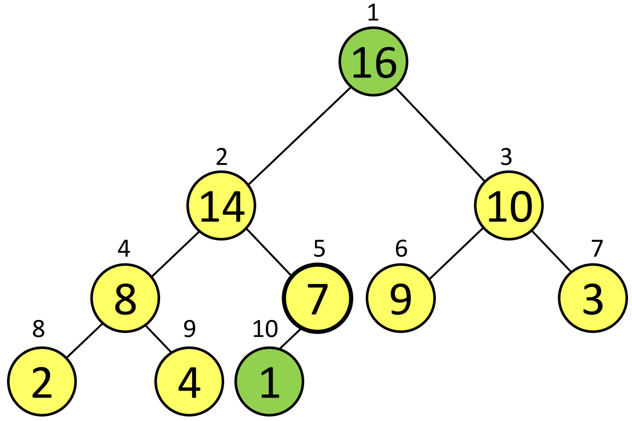


Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3.   **do** exchange A[1] ↔ A[i]
4.   heap-size[A] ← heap-size[A] – 1
5.   MaxHeapify(A, 1)



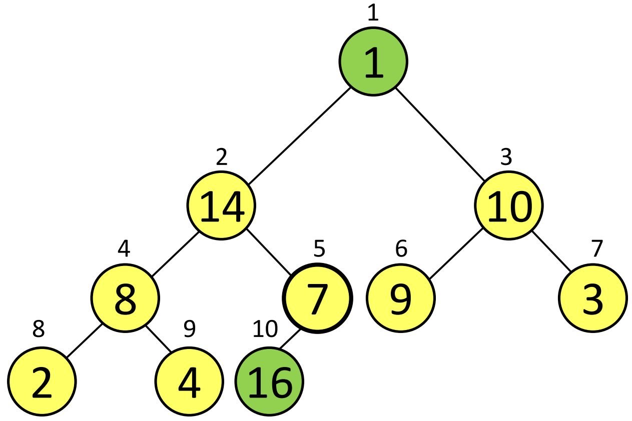
i=10

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



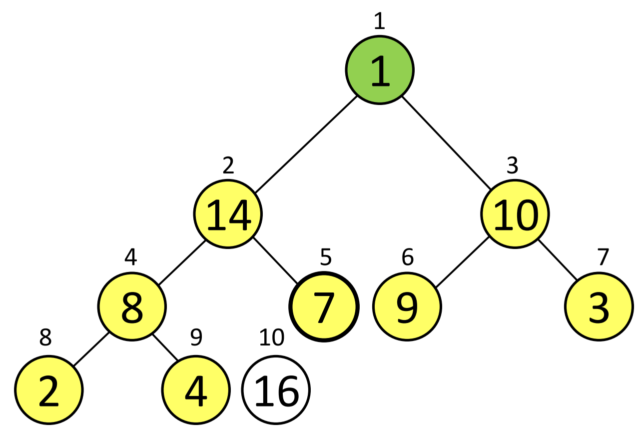
i=10

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



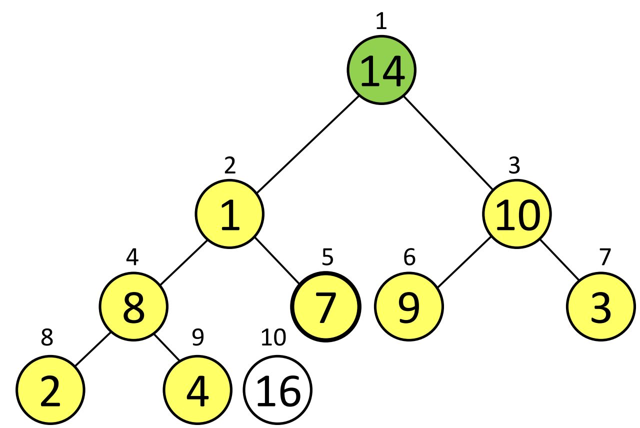
i=10

Heap Sort

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

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
        MaxHeapify(A, 1)
```



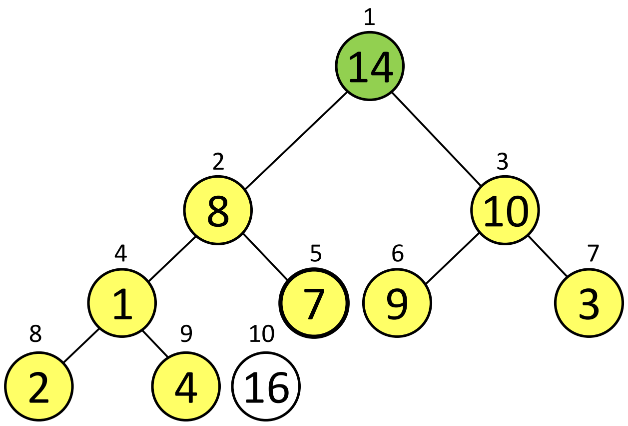
i=10

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



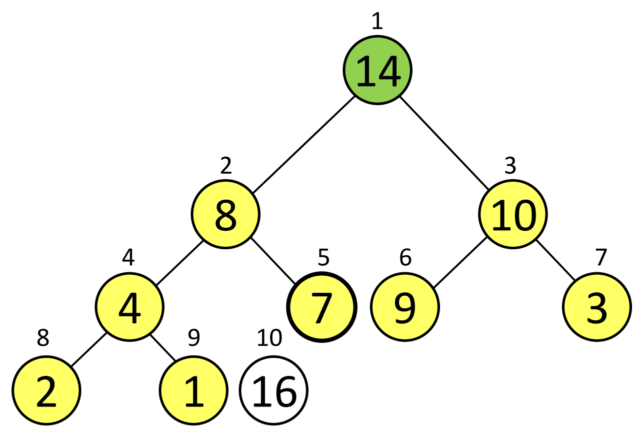
i=10

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



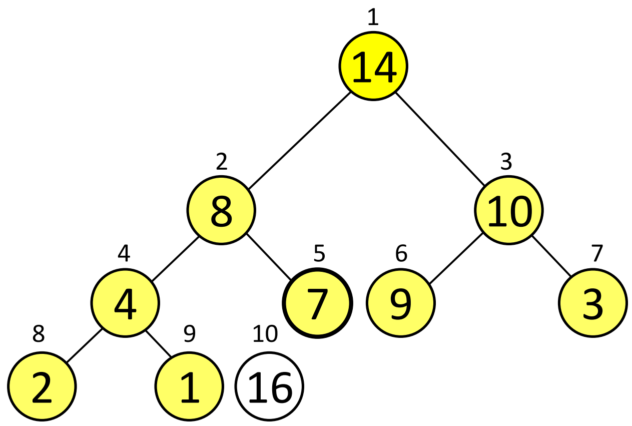
i=10

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



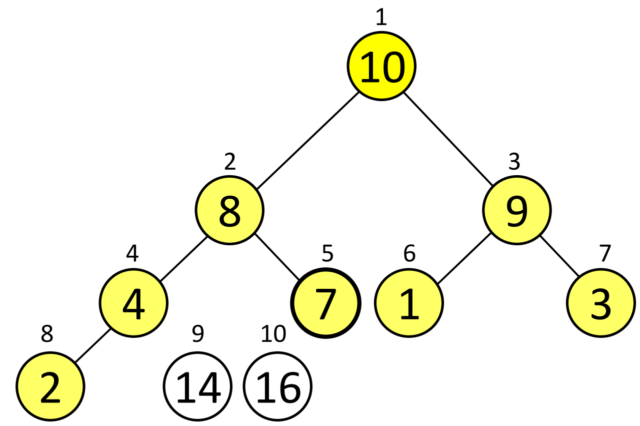
i=10

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



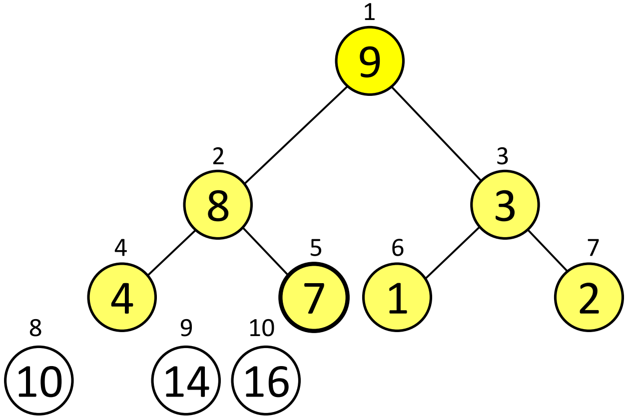
i=9

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



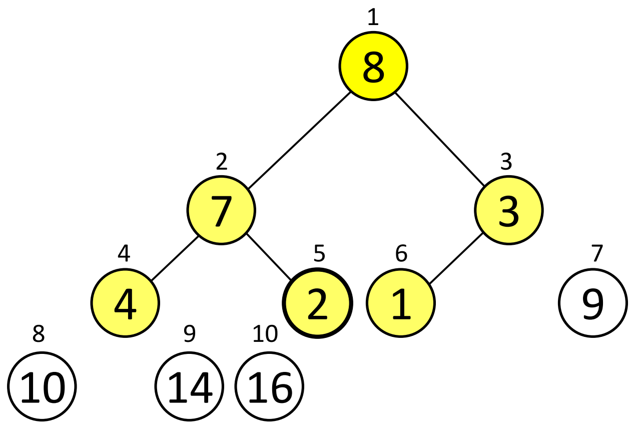
i=8

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



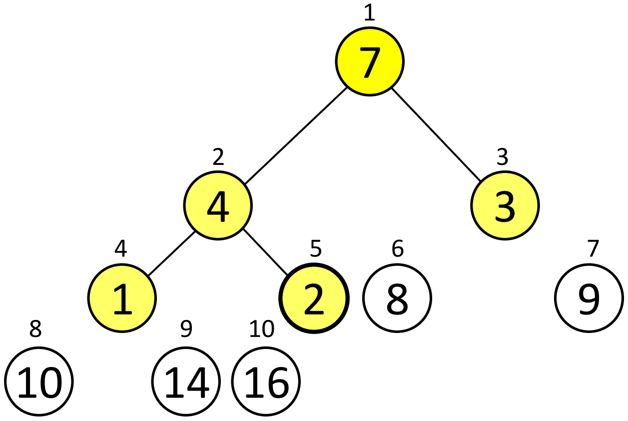
i=7

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



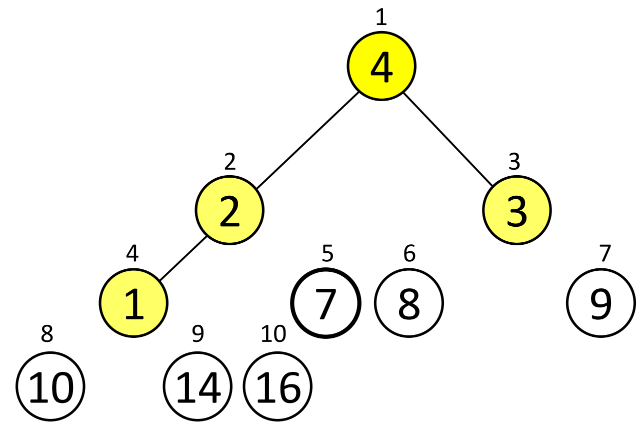
i=6

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



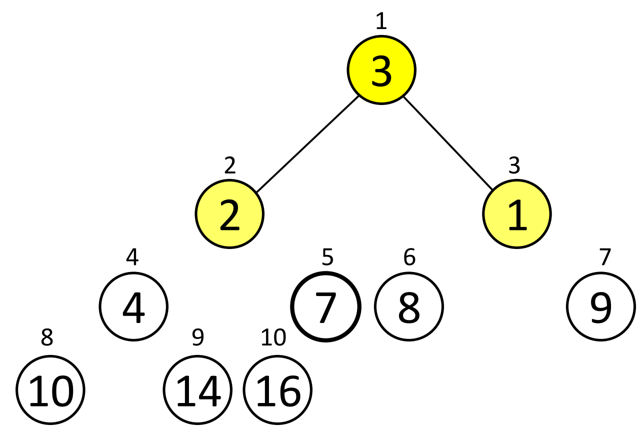
i=5

Heap Sort

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

<u>HeapSort(A)</u>

```
Build-Max-Heap(A)
for i ← length[A] downto 2
    do exchange A[1] ↔ A[i]
        heap-size[A] ← heap-size[A] – 1
        MaxHeapify(A, 1)
```



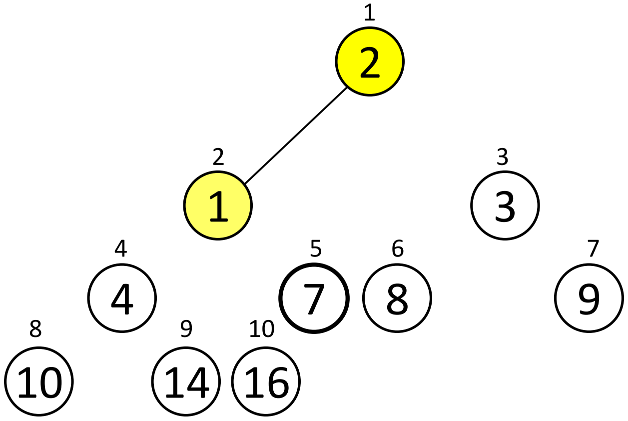
i=4

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



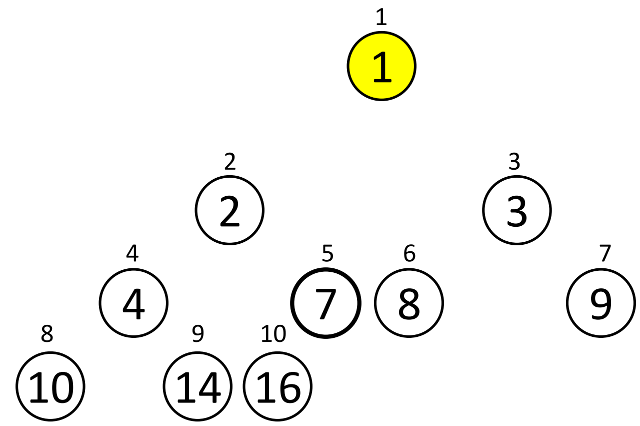
i=3

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



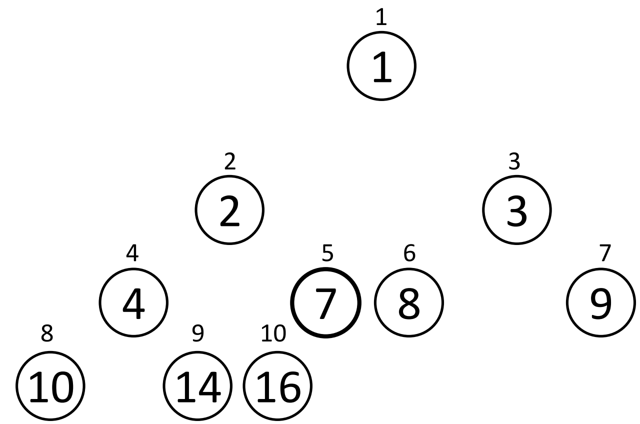
i=2

Heap Sort

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

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3.   **do** exchange A[1] ↔ A[i]
4.   heap-size[A] ← heap-size[A] – 1
5.   MaxHeapify(A, 1)



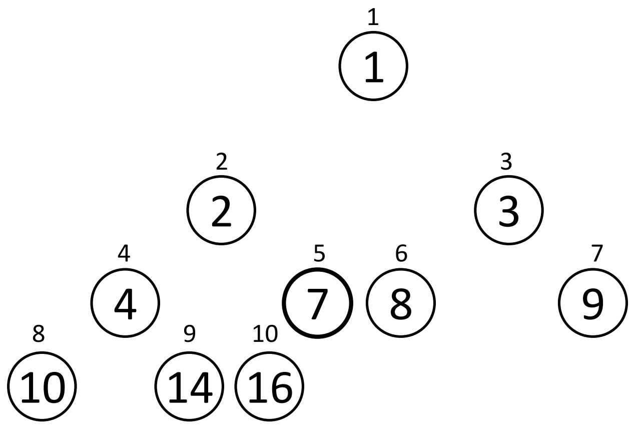
Heap Sort

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

O(N)

<u>HeapSort(A)</u>

1. Build-Max-Heap(A)
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MaxHeapify(A, 1)



O(NlogN)

Time complexity: **O(NlogN)**

Priority Queues

- Data structure
  - MAX-HEAP / MIN-HEAP
- Operations
  - HEAP-EXTRACT-MAX / HEAP-EXTRACT-MIN
  - HEAP-DECREASE-KEY / HEAP-INCREASE-KEY
  - MAX-HEAP-INSERT / MIN-HEAP-INSERT
- One application: Schedule jobs on a shared resource
  - PQ keeps track of jobs and their relative priorities
  - When a job is finished or interrupted, highest priority job is selected from those pending using HEAP-EXTRACT-MAX
  - A new job can be added at any time using
    - MAX-HEAP-INSERT

Implementation of Priority Queue

Sorted linked list: Simplest implementation

- INSERT
  - O(n) time
  - Scan the list to find place and splice in the new item
- EXTRACT-MAX
  - O(1) time
  - Take the first element

►Fast extraction but slow insertion

Implementation of Priority Queue

Unsorted linked list: Simplest implementation

- INSERT
  - O(1) time
  - Put the new item at front
- EXTRACT-MAX
  - O(n) time
  - Scan the whole list

►Fast insertion but slow extraction

Heap Implementation of PQ

HEAP-EXTRACT-MAX implements EXTRACT-MAX

```
HEAP-EXTRACT-MAX(A)
1.  if A.heap-size < 1
2.      then error "heap underflow"
3.  max ← A[1]
4.  A[1] ← A[A.heap-size]
5.  A.heap-size ← A.heap-size - 1
6.  MaxHeapify(A, 1)
7.  return max
```

Running time : Dominated by the running time of MaxHeapify which is $O(\lg n)$

Heap Implementation of PQ

INSERT: Insertion is like that of Insertion-Sort.

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

Running time of MAX-HEAP-INSERT on an *n*-element heap is O(*lg n*)

Heap Implementation of PQ

HEAP-INCREASE-KEY increases *key* at position *i* (max-heap)

```
HEAP-INCREASE-KEY(A, i, key)
if  key < A[i]
    error "new key is smaller than the current key"
A[i] = key
while i > 1 and A[Parent(i)] < A[i]
      exchange A[i] and A[Parent(i)]
      i = Parent(i)
```

Running time is O(*lg n*)

- The path traced from the new increased node to the root has at most length O(*lg n*)