ICSI 403
Design and Analysis of Algorithms

Chapter 6 - Heaps, Heapsort, and Priority Queues

J Marques de Carvalho

Heapsort - Introduction

- Introduces a new data structure (the heap)
- Combines best of Merge Sort & Insertion Sort
- $O(n \lg n)$ (guaranteed), like Merge Sort
  - Better than Insertion Sort's $O(n^2)$
- In-place sort like Insertion Sort
  - Requires only a constant amount of storage beyond the array to sort, unlike Merge Sort, which requires $O(n)$ additional storage space, or Quicksort (which requires stack space)
- Not as fast (in practice) as Quicksort

Introduction

- “A Heap”
  - Originally derived from Heapsort, but later used to refer to the garbage-collected data storage pools in LISP, C, and Java
    - Some other languages use garbage-collected heaps, too
  - Don’t get the two confused – Heapsort has nothing to do with garbage-collected data storage
- There are “Min Heaps” and “Max Heaps”
  - Details to follow

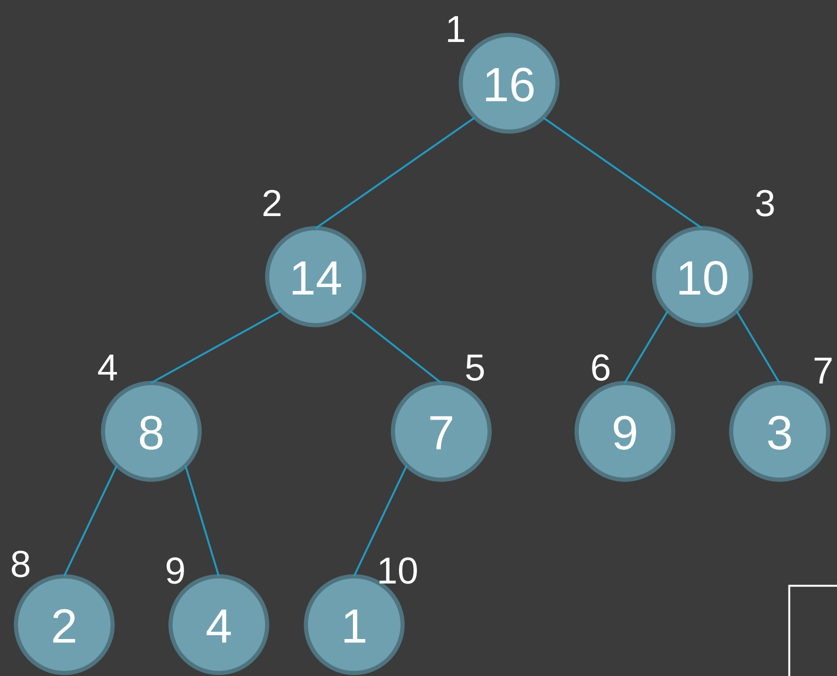
A Heap

- A nearly-full binary tree
- NOT a Binary Search Tree (BST)
  - We don’t go down through the tree comparing keys to find information.
- The tree is completely full, except for the right part of the lowest level (the left part of the lowest level IS completely full, up to a point)
- A simple array can be used to store a heap – we don’t need tree nodes and pointers!

Storing a Heap in an Array

- Suppose we store a heap in array A
- We have two pieces of information to keep track of regarding A
  - A's <u>*length*</u> (how large is A) – $A.length$
  - A's <u>*heapsize*</u> (how many items in A are part of the heap) – $A.heapsize$
    - Not everything in A is required to be part of the heap (part of what's in A can be outside of the heap – we'll come back to this later)
    - There are $A.length - A.heapsize$ elements of A that are not part of the heap

An Example Heap



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

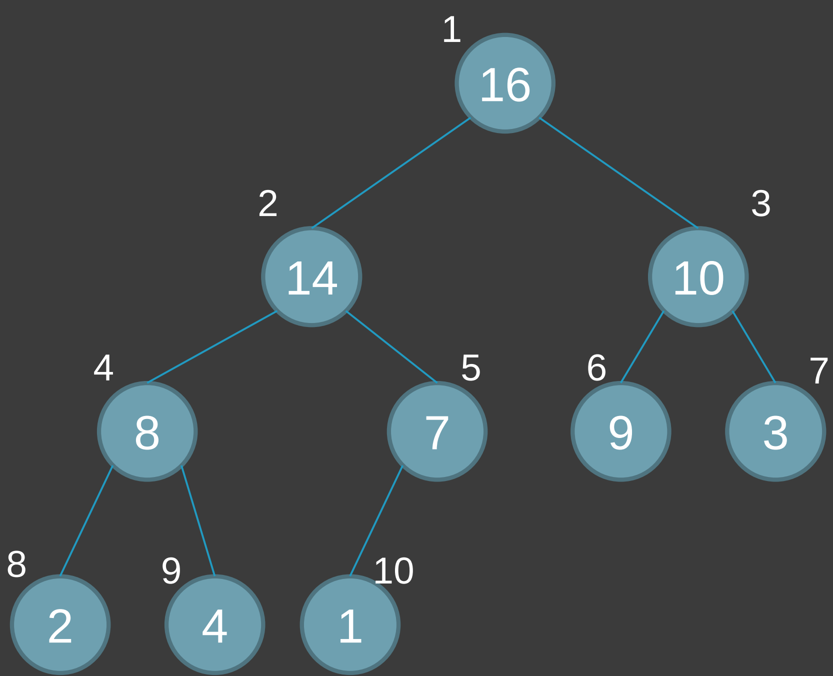
If we number the nodes top-to-bottom, left-to-right, …

We get the subscripts for storing the heap in an array

The root is stored in $A[1]$. Heaps are not typically stored as zero-based arrays

For a given node in the tree (i.e., member of array A), whose subscript (index) is i
The subscript (index) of its left child is $2i$
The subscript (index) of its right child is $2i+1$
The subscript (index) of its parent is $\lfloor i/2 \rfloor$

An Example Heap

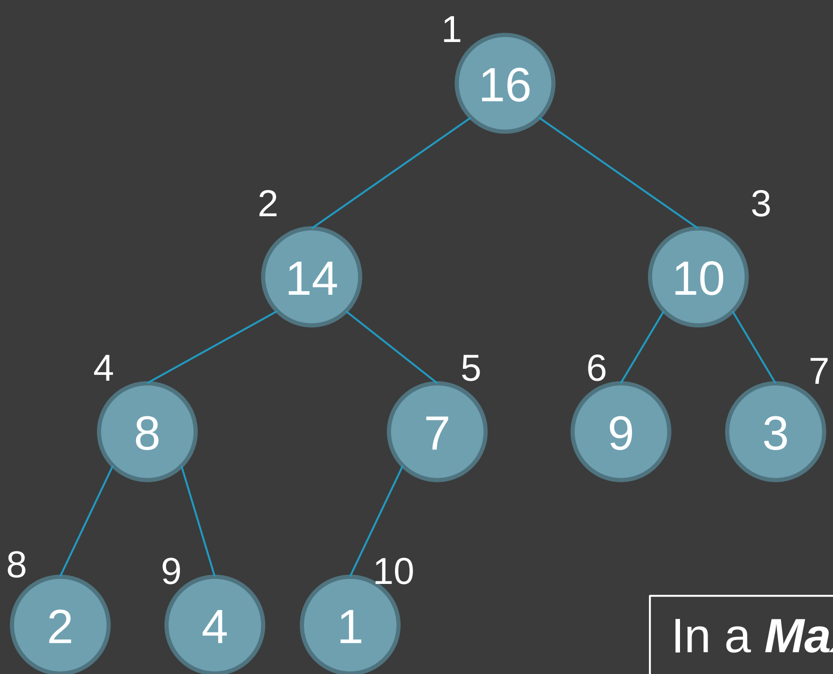


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

$2i$, $2i+1$, and $\lfloor i/2 \rfloor$ are efficiently calculated using left and right shift operations (and an add)

For a given node in the tree (i.e., member of array A), whose subscript (index) is i
The subscript (index) of its left child is $2i \rightarrow A[i].\text{LEFT} = A[2i]$
The subscript (index) of its right child is $2i+1 \rightarrow A[i].\text{RIGHT} = A[2i+1]$
The subscript (index) of its parent is $\lfloor i/2 \rfloor \rightarrow A[i].\text{PARENT} = A[i/2]$

An Example Heap



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

For a ***Min***-Heap, simply change $\geq$ to $\leq$

We said there are two kinds of heaps: Max-heaps and Min-heaps, which satisfy either the <u>*Max-Heap*</u> or <u>*Min-Heap*</u> property

In a ***Max-Heap***, for every node i other than the root:

$$A[i].\text{P}\textsc{arent} \geq A[i]$$

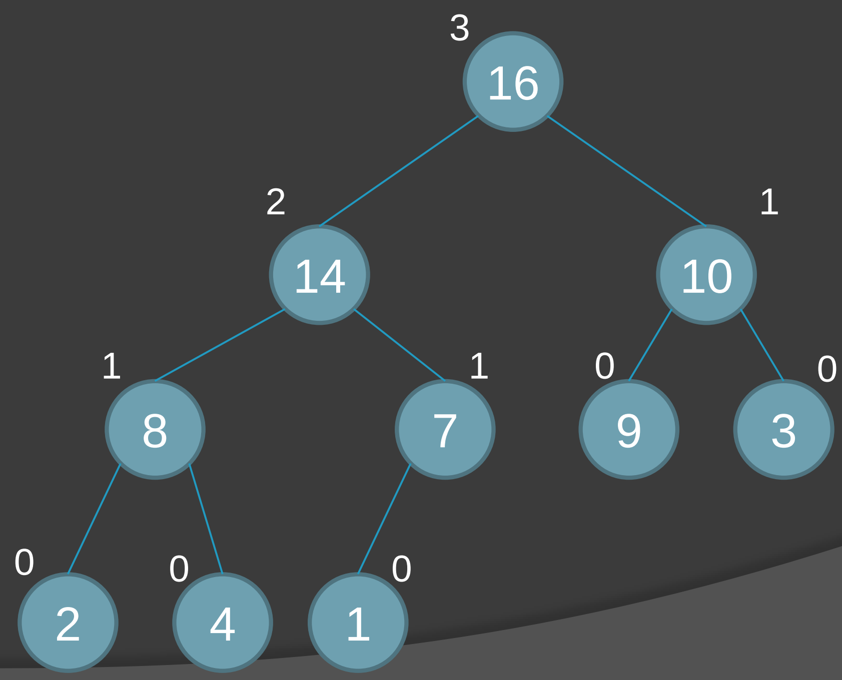
In other words, the value at a node is, at most, the value of its parent.

Alternatively, the value at a node is, at least, as large as its child(ren).

For a ***Max***-Heap, the largest value is always at the root, or $A[1]$

Height of a Node in a Heap

- The *height* of a node in a heap is the number of edges (child links) on the longest downward path from that node to a leaf
- The height of the *entire heap* is the height of the root



Height of a Heap

- Since a heap of n elements is based on a complete binary tree, the height of a heap is $\Theta(\lg n)$
- Basic operations on heaps run in time proportional to (at most) the height of the tree, and therefore take $\Theta(\lg n)$ time.

Heap Operations on a Max-Heap

- MAX-HEAPIFY – runs in $O(\lg n)$ time – used to maintain the Max-Heap property
- Before we run MAX-HEAPIFY, $A[i]$ may be smaller than its children
- Assumes that the left and right subtrees of i are heaps
- MAX-HEAPIFY lets $A[i]$ “float down” the heap while $A[i]$ < its children
- After MAX-HEAPIFY runs, the left and right subtrees rooted at i are Max-Heaps

MAX-HEAPIFY Pseudocode

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heapsize and A[l] > A[i]
        largest = l
else  largest = i
if r ≤ A.heapsize and A[r] > A[largest]
    then largest = r
if largest ≠ i
    exchange A[i] ↔ A[largest]
    MAX-HEAPIFY(A, largest)
```

Find the largest of $A[i]$ and its children

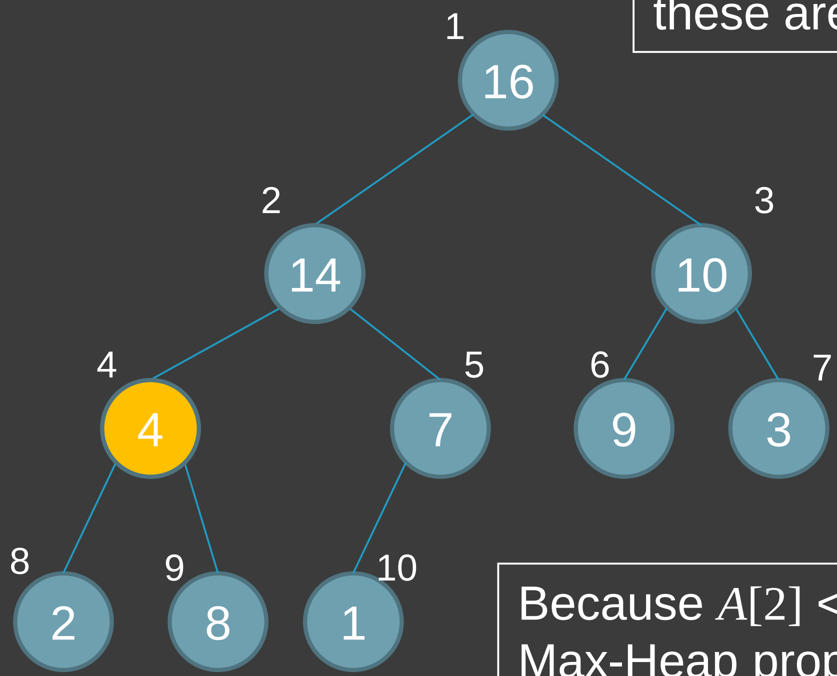
How MAX-HEAPIFY Works

- Compare $A[i]$, $A[A[i].\text{LEFT}]$, and $A[A[i].\text{RIGHT}]$
- If necessary, swap $A[i]$ with the larger of its children to preserve the Max-heap property
- Keep comparing and swapping down the heap, until the subtree rooted at i is a Max-heap.
  - If we hit a leaf, then the subtree rooted at the leaf is trivially a Max-heap.
- Since we only need to go through it once for every level, the running time is $O(\lg n)$

Example

- $\text{MAX-HEAPIFY}(A, 2)$

Remember, we're really just moving items around in an array; these are *not* nodes with pointers!



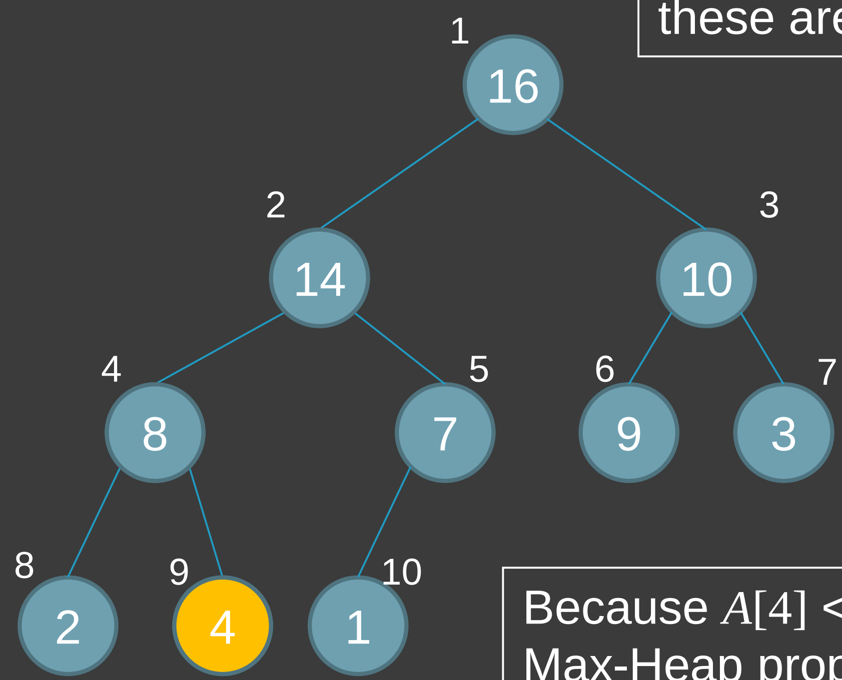
Because $A[2]$ < one of its children, the Max-Heap property is violated at $A[2]$

The larger of $\mathrm{A}[2]$'s children is $\mathrm{A}[4]$ (with a value of 14), so we swap $A[2]$ and $A[4]$, and call $\text{MAX-HEAPIFY}(A, 4)$, because we may have just destroyed the Max-Heap property at $A[4]$.

Example

- MAX-HEAPIFY(A, 4)

Remember, we're really just moving items around in an array; these are *not* nodes with pointers!



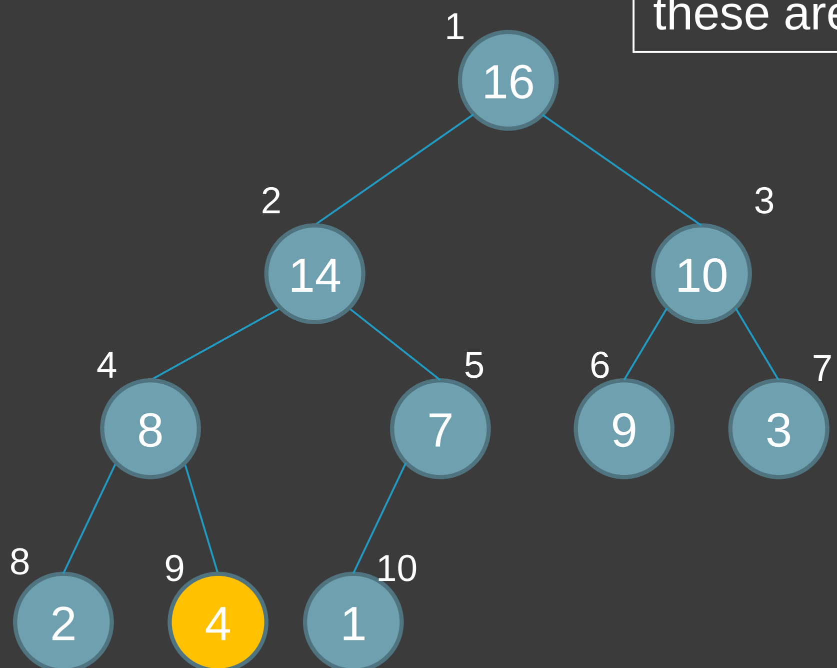
Because $A[4]$ < one of its children, the Max-Heap property is violated at $A[4]$

The larger of A[4]'s children is A[9] (with a value of 8), so we swap $A[4]$ and $A[9]$, and call MAX-HEAPIFY(A, 9), because we may have just destroyed the Max-Heap property at $A[9]$.

Example

- MAX-HEAPIFY(A, 9)

Remember, we're really just moving items around in an array; these are *not* nodes with pointers!



Because $A[9]$ is a leaf, it is a trivial Max-heap, and MAX-HEAPIFY ends Because no statements in MAX-HEAPIFY follow the recursive call to MAX-HEAPIFY, the whole process ends.

Building a Heap

- We can use MAX-HEAPIFY to make a heap out of an array from the bottom up
- Suppose we have an array $A[1..n]$, where
  $n = A.\textsc{Length}$
- Each leaf is already a heap, so there's no sense running MAX-HEAPIFY on the leaves.
- MAX-HEAPIFY can take two subtrees and their parent and make them a Max-Heap
- We run MAX-HEAPIFY from all leaves going upward, and then we have a full heap

Build-Max-Heap Pseudocode

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

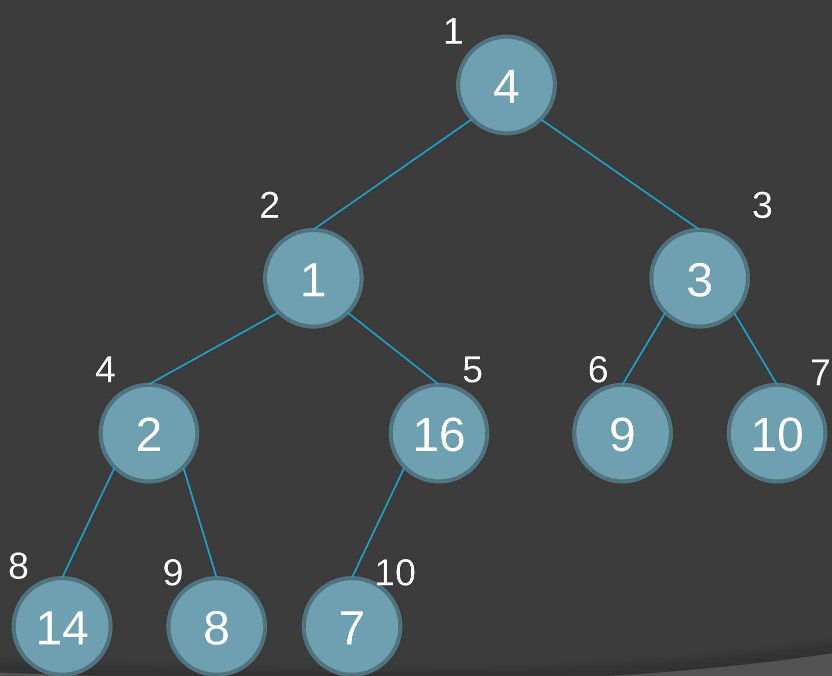
Note: $\lfloor A.length/2 \rfloor$ is just the left half of the array (less one, if the size of the array is odd)

Example

- Consider the following 10-element array A

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

- If we represent this array as a heap, we have



The BUILD-MAX-HEAP algorithm says:
for $i = \lfloor A.length/2 \rfloor$ **downto** 1
do MAX-HEAPIFY(A, i)

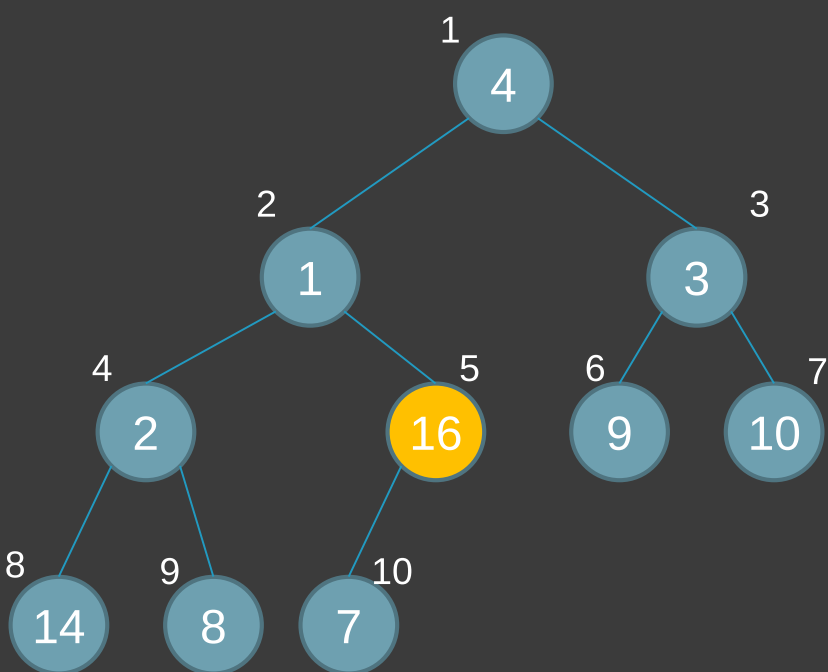
for $i = \lfloor\ 10/2\ \rfloor$ **downto** 1

Why do we only MAX-HEAPIFY from $A[5]$ down?

Example

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

for $i = \lfloor A.length/2 \rfloor$ **downto** 1
  do MAX-HEAPIFY(A, i)



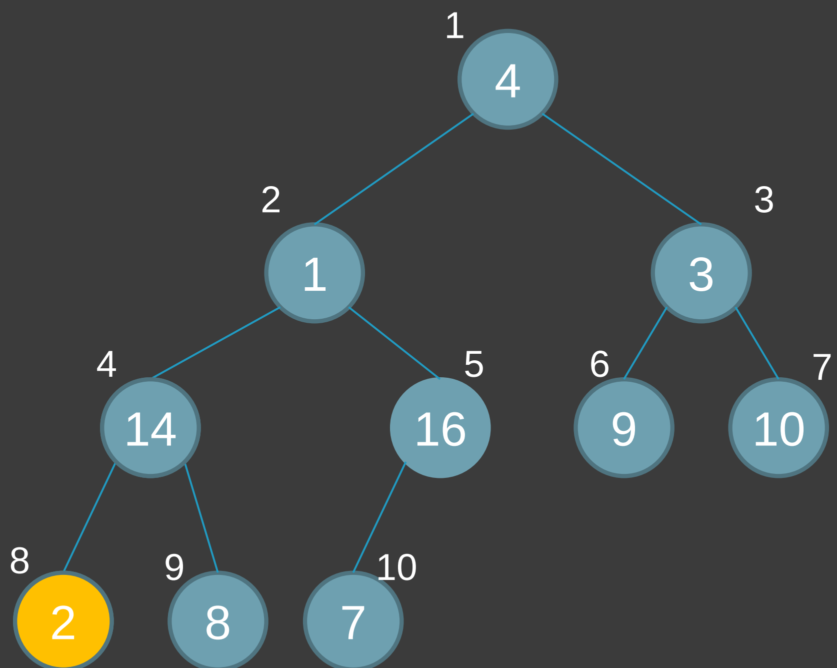
A[5] (16) is already $\geq$ its child(ren), so there's nothing to do here.

Go on to A[4]

Example

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

for $i = \lfloor A.length/2 \rfloor$ **downto** 1
 do MAX-HEAPIFY(A, i)



A[4] (2) is < its largest child (14), so
 swap A[4] and A[8]

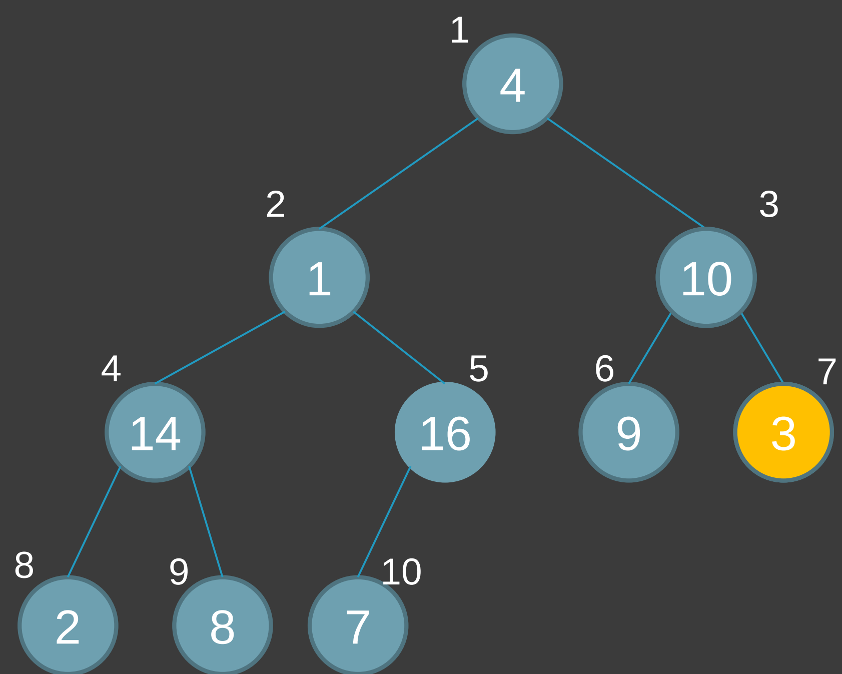
Then we MAX-HEAPIFY(A, 8)
 A[8] is a leaf, so we're done here

Go on to A[3]

Example

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

for $i = \lfloor A.length/2 \rfloor$ **downto** 1
  do MAX-HEAPIFY(A, i)



$A[3]$ (3) is < its largest child (10), so
  swap $A[3]$ and $A[7]$

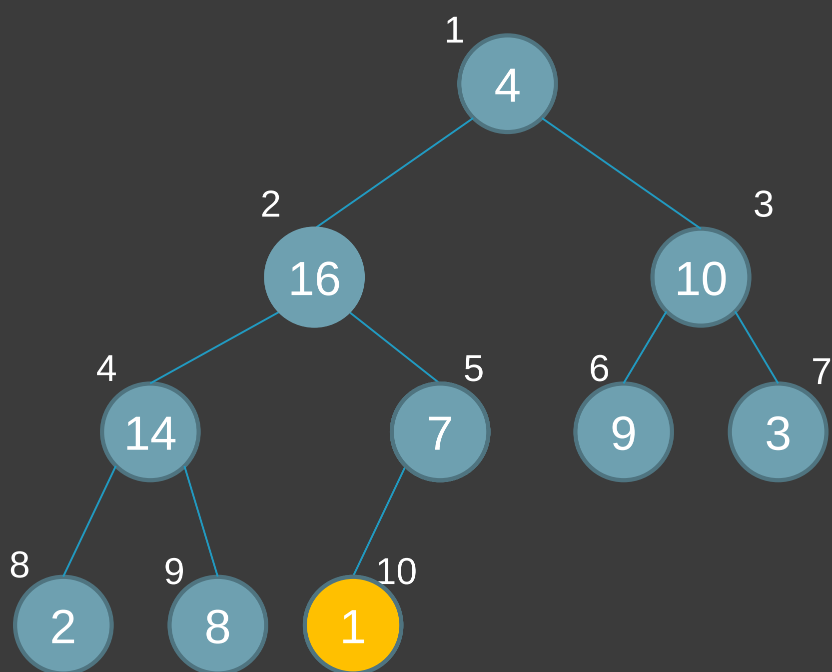
Then we MAX-HEAPIFY(A, 7)
  $A[7]$ is a leaf, so we're done here

Go on to $A[2]$

Example

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

for $i = \lfloor A.length/2 \rfloor$ **downto** 1
  do MAX-HEAPIFY(A, i)



A[2] (1) is < its largest child (16), so
  swap *A*[2] and *A*[5]

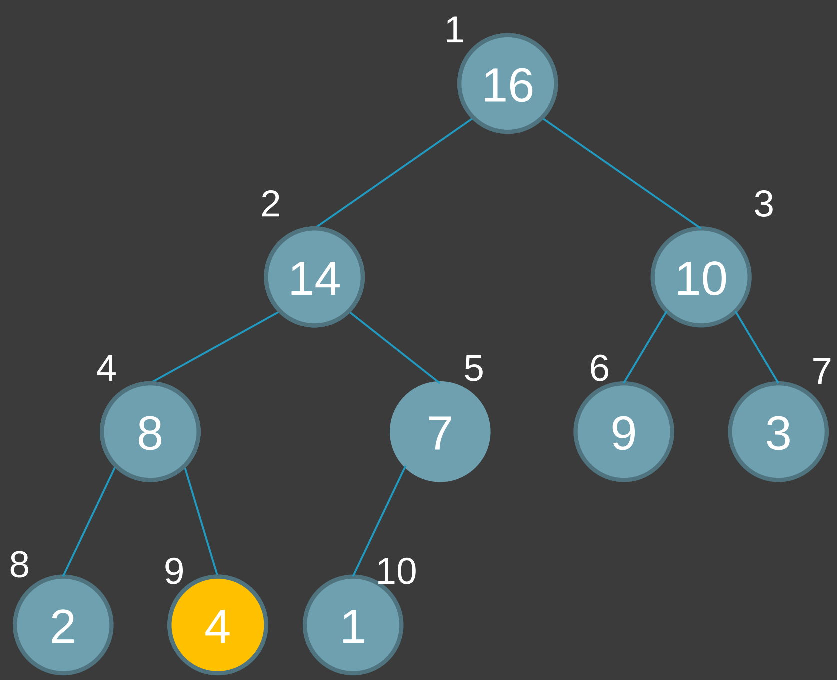
Then we MAX-HEAPIFY(*A*, 5)
  A[5] is < its largest child (7), so
    swap *A*[5] and *A*[10]

Then we MAX-HEAPIFY(*A*, 10)
  A[10] is a leaf, so we're done

Go on to A[1]

Example

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



for $i = \lfloor A.length/2 \rfloor$ **downto** 1
do MAX-HEAPIFY(A, i)

A[1] (4) is < its largest child (16), so swap A[1] and A[2]

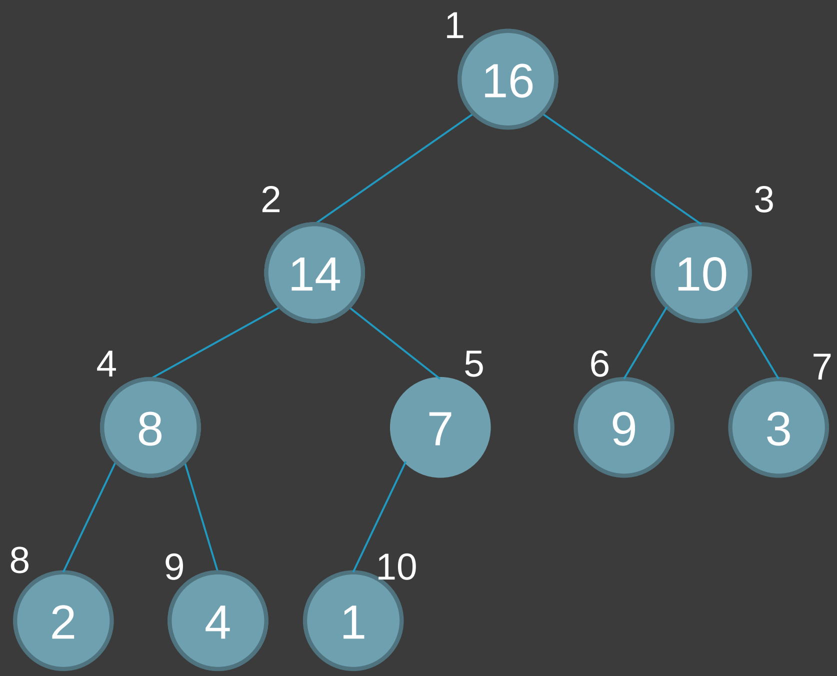
Then we MAX-HEAPIFY(A, 2)
A[2] (4) is < its largest child (14), so swap A[2] and A[4]

Then we MAX-HEAPIFY(A, 4)
A[4] (4) < its largest child (8), so swap A[4] and A[9] and then MAX-HEAPIFY(A, 9) → leaf

The heap is complete

Example

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



Note:
We now have a valid Max-Heap

This does ***not*** mean the array is now sorted; it only means that the Max-Heap property is now valid at every location

But having a valid Max-Heap *is* the prerequisite for running Heapsort

Heapsort (1)

- Now that we can build a heap, and we can "re-heapify" a heap, we are ready to add one more simple step to create Heapsort.
- We start with the array A in no particular order, and use BUILD-MAX-HEAP
- At this point, with a proper heap, $A[1]$, the root of the heap, is the largest value in the array.

Heapsort (2)

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

- Swap $A[1]$ and $A[n]$. This puts the largest value in the n^{th} position, then $A.heapsize$--

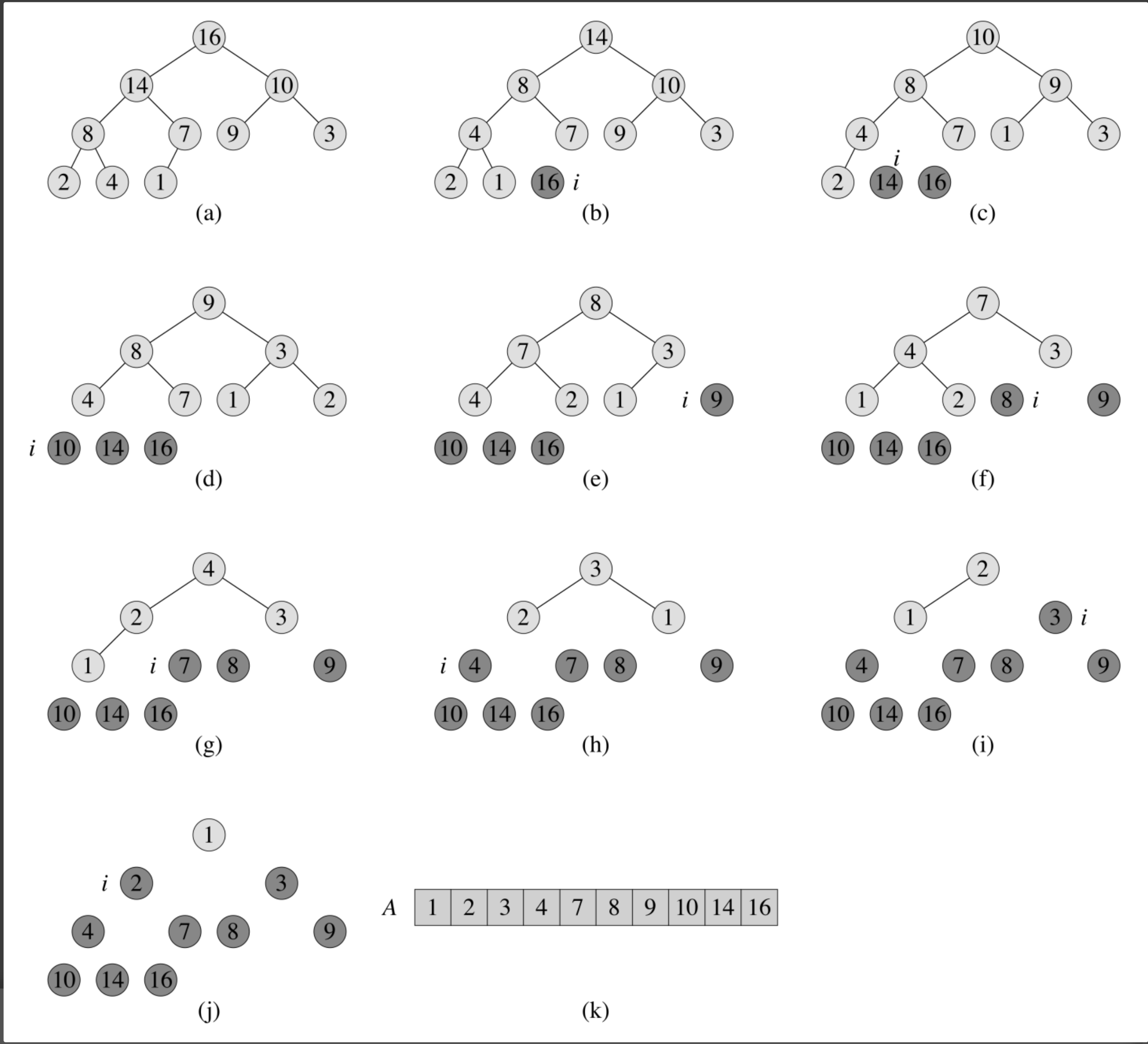
1	2	3	4	5	6	7	8	9	10
1	14	10	8	7	9	3	2	4	16

- That may have violated (probably did) the Max-Heap property at $A[1]$.
- Call $\textsc{Max-Heapify}(1)$ to “re-heap” it
- That will put the next-largest item in $A[1]$.
- Swap $A[1]$ and $A[A.heapsize]$, then $A.heapsize$--
- Repeat…

Heapsort Pseudocode

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] ↔ A[i]
    A.heapsize = A.heapsize – 1
    MAX-HEAPIFY(A, 1)
```

Example



16
14
10
8
7
9
3
2
4
1
(a)
14
8
10
4
7
9
3
2
1
16
i
(b)
10
8
9
4
7
1
3
i
2
14
16
(c)
9
8
3
4
7
1
2
i
10
14
16
(d)
8
7
3
4
2
1
i
9
10
14
16
(e)
7
4
3
1
2
8
i
9
10
14
16
(f)
4
2
3
1
i
7
8
9
10
14
16
(g)
3
2
1
i
4
7
8
9
10
14
16
(h)
2
1
3
i
4
7
8
9
10
14
16
(i)
1
i
2
3
4
7
8
9
10
14
16
(j)
A
1 2 3 4 7 8 9 10 14 16
(k)

Heapsort - Summary

- Combines best of Merge Sort…
  - O(*n* lg *n*) (guaranteed) execution time, and …
- … and Insertion Sort
  - in-place sorting (requires only a constant amount of storage beyond the array to sort, unlike Merge Sort, which requires O(n) additional storage space)

Proving BUILD-MAX-HEAP's Correctness

(See Chapter 2)

Loop Invariant: At the start of every iteration of the **for** loop, each node $i + 1$, $i + 2$, …, n is the root of a max-heap.

Initialization: We know that every node $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, …, n is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the **for** loop, the invariant is initially true.

BUILD-MAX-HEAP's Correctness (2)

Maintenance: Children of node *i* are indexed higher than *i*, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that i+1, i+2, … *n* are all roots of max-heaps, MAX-HEAPIFY makes node *i* a max-heap root. Decrementing *i* reestablishes the loop invariant at each iteration.

Termination: When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

Priority Queues

- A data structure for maintaining a set of elements
- Min-priority and max-priority queues (based on min-heaps and max-heaps, respectively).
- The book focuses on max-priority queues; implementing min-priority queues is left as an exercise (6.5-3).

Max-priority Queue Operations

- A max-priority queue supports the following operations:
  - INSERT(S, x) – inserts the element x into the set S
  - MAXIMUM(S) – returns the element of S with the largest key
  - EXTRACT-MAX(S) removes and returns the element of S with the largest key
  - INCREASE-KEY(S, x, k) – increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Max-priority Queue vs. Min-priority

- Operations of max-priority and min-priority queues:

Max-priority	**Min-priority**
INSERT(S, x)	INSERT(S, x)
MAXIMUM(S)	MINIMUM(S)
EXTRACT-MAX(S)	EXTRACT-MIN(S)
INCREASE-KEY(S, x, k)	DECREASE-KEY(S, x, k)

Max-priority Queue Application

- Computer job scheduling
- Max-priority queue keeps track of the jobs and their priorities
- When one job finishes (or is interrupted), EXTRACT-MAX is used to select the highest-priority remaining job to run next
- A new job can be added to the queue at any time with INSERT
- The priority of a job already in the queue can be changed with INCREASE-KEY

Min-priority Queue Application

- Event-driven simulator
- Items in the queue are events to simulate
- Key values are event occurrence times
- Events must be simulated in order of occurrence times
- EXTRACT-MIN picks the next event to simulate
- New events to be simulated are INSERTed into the queue.

Implementing Priority Queues

- Heaps can be used to implement priority queues
  - Often, we need to store a *handle* (pointer, index, etc.) to the corresponding application object in each heap element
  - How we implement and manage the handles will be application-dependent

Implementing Max-priority Queue Operations

HEAP-MAXIMUM(A)

1 **return** $A[1]$

Runs in $\Theta(1)$ time

Implementing Max-priority Queue Operations

HEAP-EXTRACT-MAX(A)

If $A.heapsize < 1$

    error "heap underflow"

$max = A[1]$

$A[1] = A[A.heapsize]$

$A.heapsize = A.heapsize - 1$

MAX-HEAPIFY(A, 1)

return max

Runs in $\Theta(\lg n)$ time: MAX-HEAPIFY is $\Theta(\lg n)$; all else is $\Theta(1)$

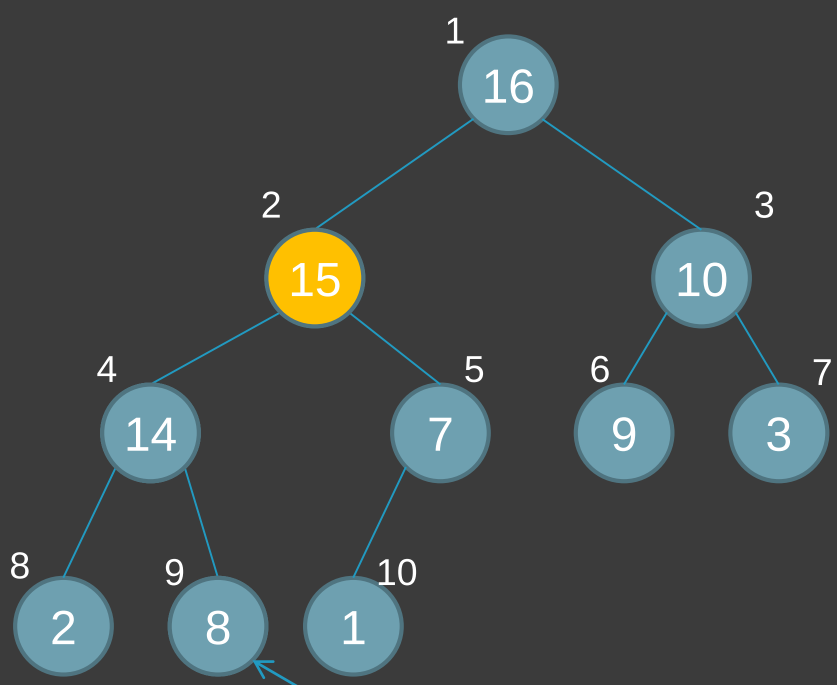
Implementing Max-priority Queue Operations

HEAP-INCREASE-KEY(A, i, key)

1 **if** $key < A[i]$
2 **error** "new key is less than current key"
3 $A[i] = key$
4 **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5 exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
6 $i = \text{PARENT}(i)$

Runs in $O(\lg n)$ time: may move new value up heap to root; heap height is $\Theta(\lg n)$

HEAP-INCREASE-KEY Example



Swap this value with its parent as long as it is greater than its parent (or it has made it up to the root)

Increase *A*[9]'s key from 4 to 15

Implementing Max-priority Queue Operations

MAX-HEAP-INSERT(A, key)

1 $A.heapsize = A.heapsize + 1$ // Enlarge Heap

2 $A[A.heapsize] = -\infty$ // add new leaf

3 HEAP-INCREASE-KEY(A, $A.heapsize$, key)

// Set desired priority (and percolate to proper location)

Runs in $O(\lg n)$ time:

$$\Theta(1) + \Theta(1) + O(\lg n)$$

Line 1 2 3

Summary

- Heaps can be used to implement priority queues with run time of $O(\lg n)$ for any of the priority queue operations

End of Chapter 6

Read your textbook for details.