COMP319 Algorithms 1
Lecture 4
Advanced Sorting Methods

Instructor: Gil-Jin Jang

Quick sort

MaxHeapify and MinHeapify

Building a Heap

Heap Sort

QUICK SORT

Sorting Algorithm Comparison

- Insertion/Selection/Bubble sort
  - Advantages: using less extra memory
  - Disadvantages: $T(n) = T(n\text{-}1)+cn \rightarrow O(n\text{^}2)$
- Merge sort
  - Advantages: $T(n) = 2T(n/2)+cn \rightarrow O(n \lg n)$
  - Disadvantages: extra memory of $O(n)$
- *Quicksort*
  - $O(n \lg n)$ without extra memory
- *Heapsort*

Review: Insertion Sort

```
/* Pseudo code: not an actual code,
   index starts from 1 */
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Review: Merge Sort

```
MergeSort(A, left, right) {
   if (left < right) {
        mid = floor((left + right) / 2);
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);
        Merge(A, left, mid, right);
   }
}
// Merge() takes two SORTED subarrays of A and
// merges them into a single sorted subarray of A
//      (how long should this take?)
// It requires O(n) time, and *does* require extra O(n)
   space
```

Quicksort Pseudo Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Partition

- Clearly, all the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the index of the "pivot" element separating the two subarrays
- *How do you suppose we implement this function?*

Partition In Words

- Partition(A, p, r):
  - Select an element to act as the “pivot” (*which?*)
  - Grow two regions, A[p..i] and A[j..r]
    - All elements in A[p..i] <= pivot
    - All elements in A[j..r] >= pivot
  - Increment i until A[i] >= pivot
  - Decrement j until A[j] <= pivot
  - Swap A[i] and A[j]
  - Repeat until i >= j
  - Return j

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Partition Code

```
Partition(A, p, r)
    x = A[p];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

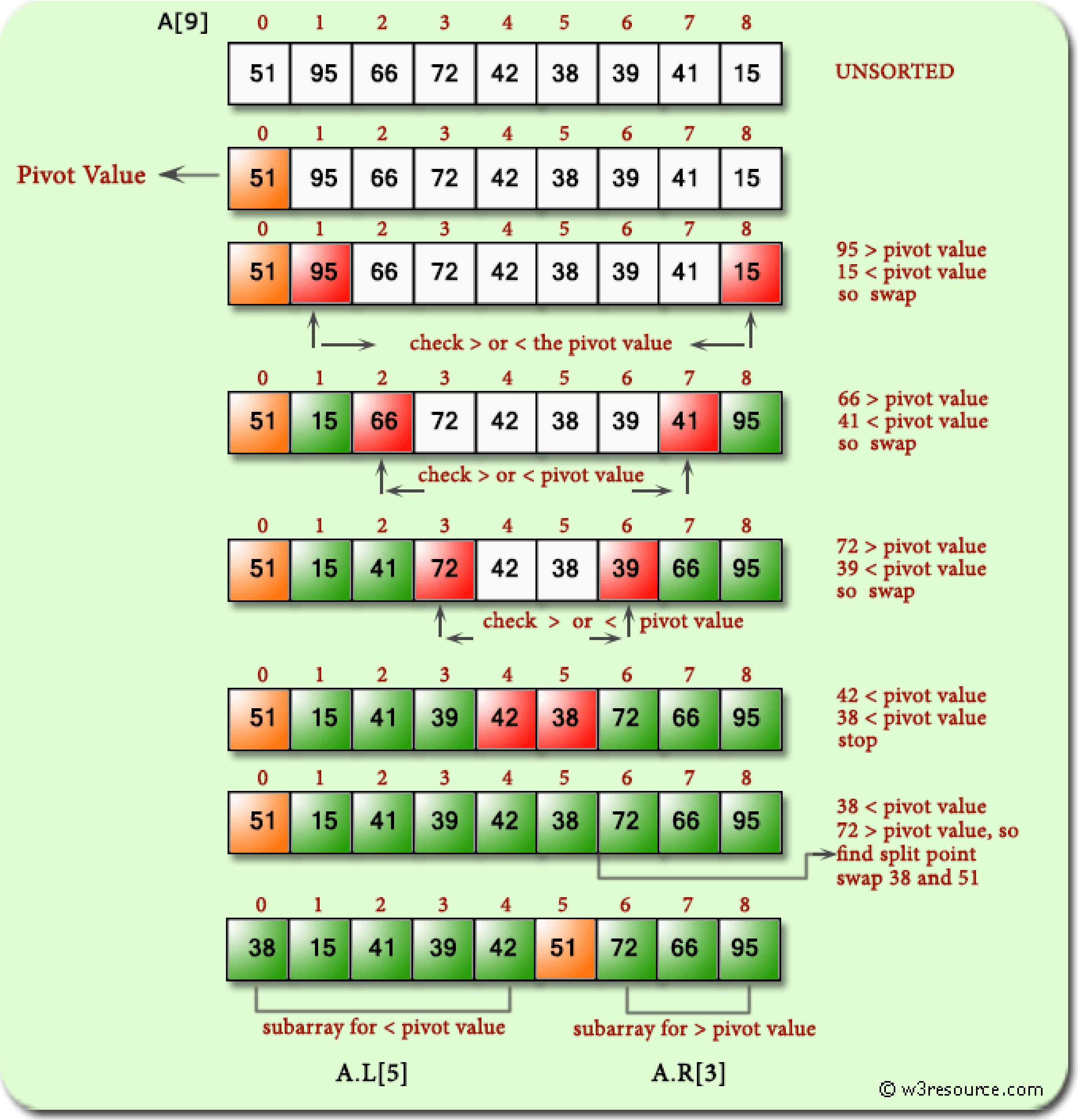
Illustrate on
A = {5, 3, 2, 6, 4, 1, 3, 7};

What is the running time of ***partition()****?*

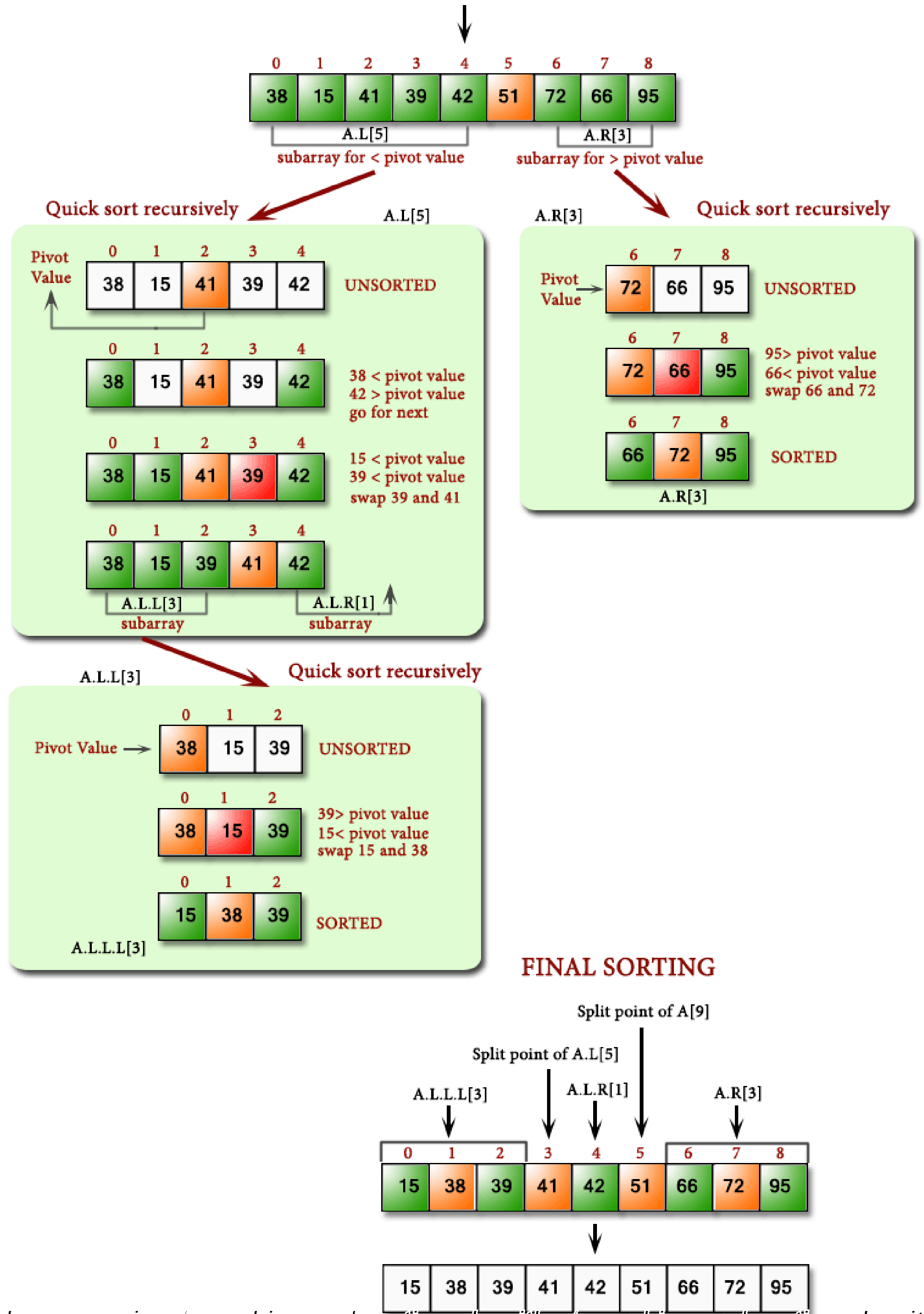
partition() *runs in O(n) time*

Quick Sort

Partition



Recurrence



Slide credit: J. Lillis, UIC's CS

https://www.w3resource.com/csharp-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-9.php

Quicksort properties

- Sorts in place (i.e. requiring constant extra memory)
- Sorts $O(n \log_2 n)$ in the average case
- Sorts $O(n^2)$ in the worst case
- Another *divide-and-conquer* algorithm
  - The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]
    - Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]
  - The subarrays are recursively sorted by calls to quicksort

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort

- *What will be the worst case for the algorithm?*
  - Partition is always unbalanced
- *What will be the best case for the algorithm?*
  - Partition is perfectly balanced
- *Which is more likely?*
  - The latter, by far, except...
- *Will any particular input elicit the worst case?*
  - Yes: Already-sorted input

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort

- In the worst case:

  $T(1) = \Theta(1)$

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

- Works out to

  $T(n) = \Theta(n^2)$

- In the best case:

  $T(n) = 2T(n/2) + \Theta(n)$

- What does this work out to?

  $T(n) = \Theta(n \log_2 n)$

Improving Quicksort

- The real liability of quicksort is that it runs in $O(n^2)$ on already-sorted input
- Book discusses two solutions:
  - Randomize the input array, OR
  - *Pick a random pivot element*
- *How will these solve the problem?*
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time

Quicksort: Radom Pick of Pivots

```
Quicksort(A, left, right) {
    if (left < right) {
                    // choose a random integer in [p, r]
        pivot = random(left, right);
         // swap the leftmost and chosen pivot in array A
        swap(A, left, pivot);

        q = Partition(A, left, right);
        Quicksort(A, left, q);
        Quicksort(A, q+1, right);
    }
}
```

Analyzing Quicksort: Average Case

- Assuming random input, average-case running time is much closer to O(n lg n) than O(n^2)

- First, a more intuitive explanation/example:
  - Suppose that partition() always produces a 9-to-1 split. This looks quite unbalanced!
  - The recurrence is thus:

    $T(n) = T(9n/10) + T(n/10) + n$
  - *How deep will the recursion go?*
    - 9n/10 → 9n/10 * 9/10 → … → 1

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort: Average Case

- Intuitively, a real-life run of quicksort will produce a mix of BAD and GOOD splits
  - Randomly distributed among the recursion tree
  - Pretend for intuition that they alternate between best-case ($n/2 : n/2$) and worst-case (n-1 : 1)
  - *What happens if we bad-split root node, then good-split the resulting size (n-1) node?*
    - We end up with three subarrays, size 1, (n-1)/2, (n-1)/2
    - Combined cost of splits = $n + n - 1 = 2n - 1 = O(n)$
    - No worse than if we had good-split the root node!

Analyzing Quicksort: Average Case

- Intuitively, the O(n) cost of a bad split (or 2 or 3 bad splits) can be absorbed into the O(n) cost of each good split
- Thus running time of alternating bad and good splits is still O(n lg n), with slightly higher constants
- How can we be more rigorous?

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort: Average Case

- For simplicity, assume:
  - All inputs distinct (no repeats)
  - Slightly different **`partition()`** procedure
    - partition around a random element, which is not included in subarrays
    - all splits (0:n-1, 1:n-2, 2:n-3, … , n-1:0) equally likely
- *What is the probability of a particular split happening?*
  - Answer: 1/*n*

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort: Average Case

- So partition generates splits

  (0:n-1, 1:n-2, 2:n-3, … , n-2:1, n-1:0)

  each with probability 1/n
- If T(n) is the expected running time,

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}\left[T(k) + T(n-1-k)\right] + \Theta(n)$$

$$= \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n)$$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - *What's the answer?*
  - Assume that the inductive hypothesis holds
    - *What's the inductive hypothesis?*
  - Substitute it in for some value < n
    - *What value?*
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
    - $T(n) \leq an \lg n + b$ for some constants a and b
  - Substitute it in for some value < n
    - The value k in the recurrence
  - Prove that it follows for n
    - Grind through it…

Analyzing Quicksort: Average Case

$$T(n) = \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n)$$

The recurrence to be solved

$$\le \frac{2}{n}\sum_{k=0}^{n-1} (ak\lg k + b) + \Theta(n)$$

Plug in inductive hypothesis

$$\le \frac{2}{n}\left[b + \sum_{k=1}^{n-1} (ak\lg k + b)\right] + \Theta(n)$$

Expand out the k=0 case

$$= \frac{2}{n}\sum_{k=1}^{n-1} (ak\lg k + b) + \frac{2b}{n} + \Theta(n)$$

2b/n is just a constant, so fold it into Θ(n)

$$= \frac{2}{n}\sum_{k=1}^{n-1} (ak\lg k + b) + \Theta(n)$$

Note: leaving the same recurrence as the book

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort: Average Case

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1}(ak\lg k + b) + \Theta(n)$$

The recurrence to be solved

$$= \frac{2}{n}\sum_{k=1}^{n-1} ak\lg k + \frac{2}{n}\sum_{k=1}^{n-1} b + \Theta(n)$$

Distribute the summation

$$= \frac{2a}{n}\sum_{k=1}^{n-1} k\lg k + \frac{2b}{n}(n-1) + \Theta(n)$$

Evaluate the summation:
b+b+…+b = b (n-1)

$$\leq \frac{2a}{n}\boxed{\sum_{k=1}^{n-1} k\lg k} + 2b + \Theta(n)$$

Since n-1<n, 2b(n-1)/n < 2b

This summation gets its own set of slides later

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort: Average Case

$$T(n) \le \frac{2a}{n} \sum_{k=1}^{n-1} k \lg k + 2b + \Theta(n)$$

The recurrence to be solved

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + 2b + \Theta(n)$$

We'll prove this later

$$= an \lg n - \frac{a}{4}n + 2b + \Theta(n)$$

Distribute the (2a/n) term

$$= an \lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right)$$

Remember, our goal is to get $T(n) \le an \lg n + b$

$$\le an \lg n + b$$

Pick a large enough that $an/4$ dominates $\Theta(n)+b$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort: Average Case

- So $T(n) \le an \lg n + b$ for certain a and b
  - Thus the induction holds
  - Thus $T(n) = O(n \lg n)$
  - Thus quicksort runs in $O(n \lg n)$ time on average (phew!)

- Oh yeah, the summation…

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil -1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k$$

Split the summation for a tighter bound

$$\leq \sum_{k=1}^{\lceil n/2 \rceil -1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg n$$

The $\lg k$ in the second term is bounded by $\lg n$

$$= \sum_{k=1}^{\lceil n/2 \rceil -1} k \lg k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Move the $\lg n$ outside the summation

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The summation bound so far

$$\le \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg(n/2) + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The lg k in the first term is bounded by lg n/2

$$= \sum_{k=1}^{\lceil n/2 \rceil - 1} k(\lg n - 1) + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

lg *n/2* = lg *n* - 1

$$= (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Move (lg n - 1) outside the summation

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The summation bound so far

$$= \lg n \sum_{k=1}^{\lceil n/2 \rceil - 1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Distribute the (lg n - 1)

$$= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The summations overlap in range; combine them

$$= \lg n \left(\frac{(n-1)(n)}{2} \right) - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The Guassian series

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \leq \left(\frac{(n-1)(n)}{2} \right) \lg n - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The summation bound so far

$$\leq \frac{1}{2}[n(n-1)] \lg n - \sum_{k=1}^{n/2-1} k$$

Rearrange first term, place upper bound on second

$$\leq \frac{1}{2}[n(n-1)] \lg n - \frac{1}{2}\left(\frac{n}{2}\right)\left(\frac{n}{2}-1\right)$$

X Guassian series

$$\leq \frac{1}{2}\left(n^2 \lg n - n \lg n\right) - \frac{1}{8}n^2 + \frac{n}{4}$$

Multiply it all out

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le \frac{1}{2}\left(n^2 \lg n - n \lg n\right) - \frac{1}{8}n^2 + \frac{n}{4}$$

$$\le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 \text{ when } n \ge 2$$

Done!!!

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Efficiency Comparison

	Worst Case	**Average Case**
Selection Sort	n^2	n^2
Bubble Sort	n^2	n^2
Insertion Sort	n^2	n^2
Mergesort	$n \log n$	$n \log n$
Quicksort	$\boldsymbol{n^2}$	$\boldsymbol{n \log n}$
Heapsort	$n \log n$	$n \log n$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Max heap and min heap

Heapify operations

Build heaps

HEAP

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Review: Comparing Sorting Methods

- Insertion/selection/bubble sort
  - Advantages: using less extra memory
  - Disadvantages: $T(n) = T(n\text{-}1)+cn \rightarrow O(n\char`^2)$
- Merge sort
  - Advantages: $T(n) = 2T(n/2)+cn \rightarrow O(n \lg n)$
  - Disadvantages: extra memory of $O(n)$
- Quicksort
  - $O(n \lg n)$ without extra memory
  - Disadvantages: in worst case, O(n^2)
- *Heapsort*
  - Combines advantages of the previous algorithms

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Binary Trees

- (1) Full binary tree
- (2) Complete binary tree
- (3) General binary

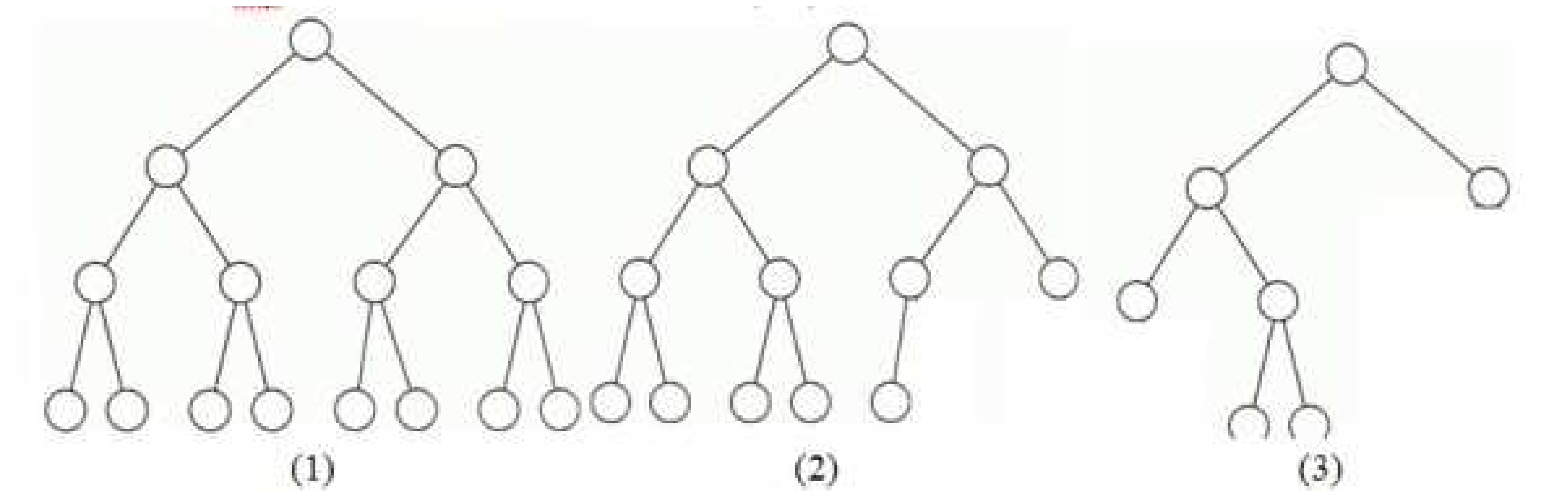
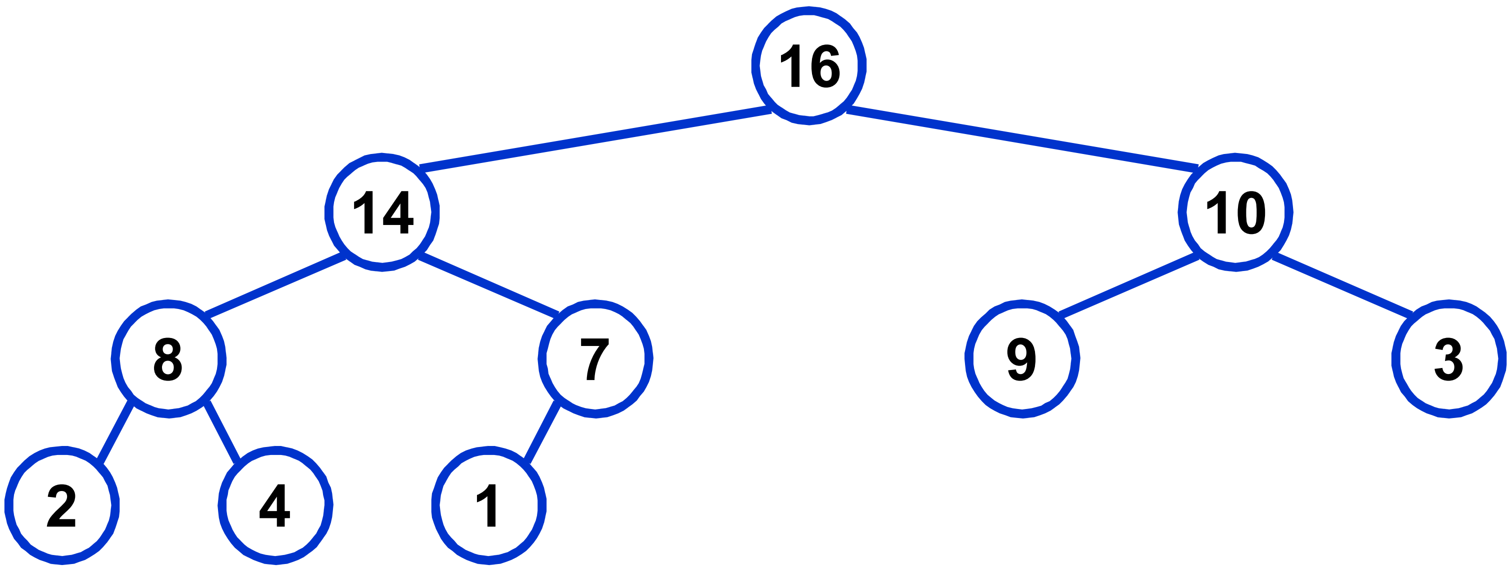


Figure taken from https://gateoverflow.in/122126/full-binary-tree-complete-almost-complete-binary-difference

Slide credit: 홍석원, 명지대학교, Discrete Mathematics, Spring 2013; 김한준, 서울시립대학교, 자료구조 및 실습, Fall 2016

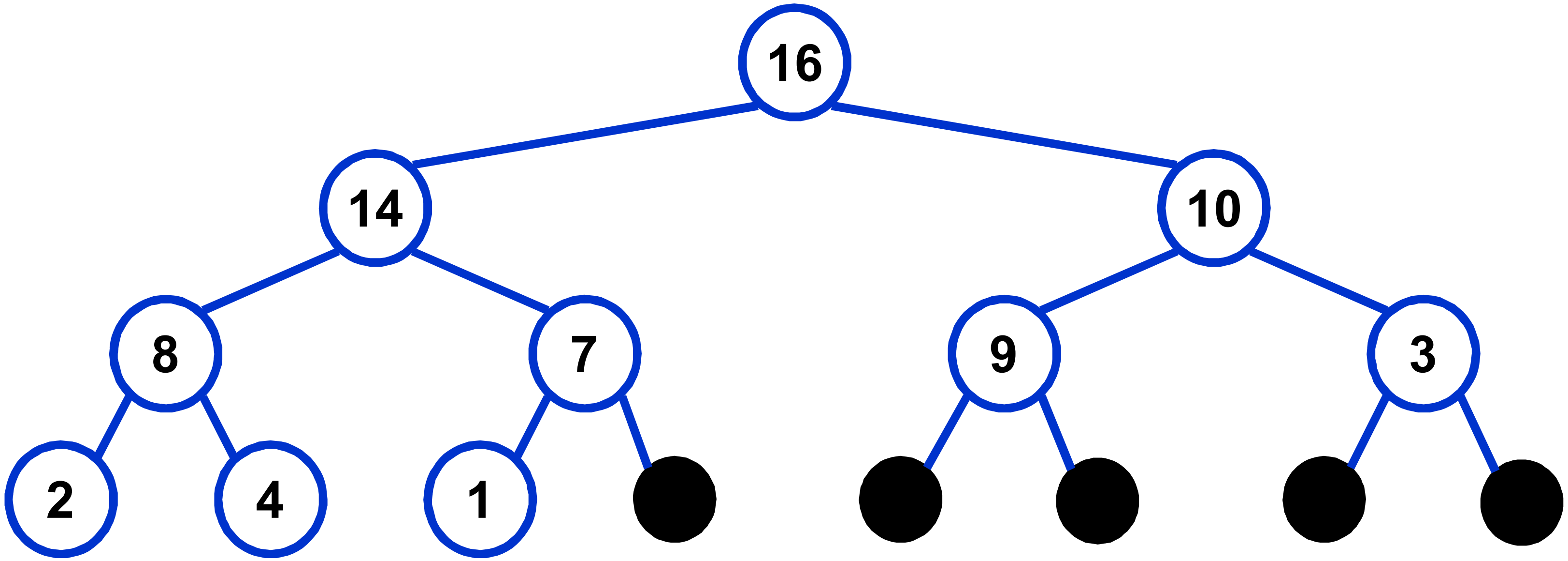
Heaps as Binary Trees

- A *heap* can be seen as a complete binary tree:
  - *What makes a binary tree complete?*
  - *Is the example below complete?*
    - *A complete binary tree is that all nodes are filled from top to bottom, left to right, without any vacancies*



Heaps as Complete Binary Trees

- A heap can be seen as a complete binary tree:
  - Or as **NEARLY FULL** binary trees
    - Unfilled slots are represented as **NULL** pointers (filling dummy values in)



Heap Implementation as Arrays

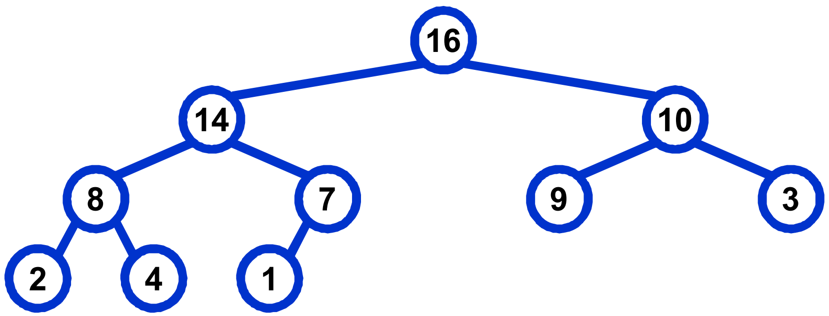
- In practice, heaps (*complete binary trees*) are usually implemented as arrays:
  - The root node is $\mathrm{A}[1]$ (*note: not* $\mathrm{A}[0]$)
  - Node i is $\mathrm{A}[i]$
  - The parent of node i is $\mathrm{A}[i/2]$
    - note: integer division, quotient only
  - The left child of node i is $\mathrm{A}[2i]$
  - The right child of node i is $\mathrm{A}[2i+1]$

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

A =

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

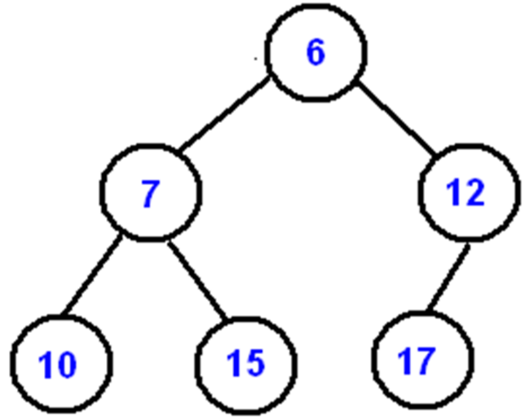
=



The Heap Ordering Properties

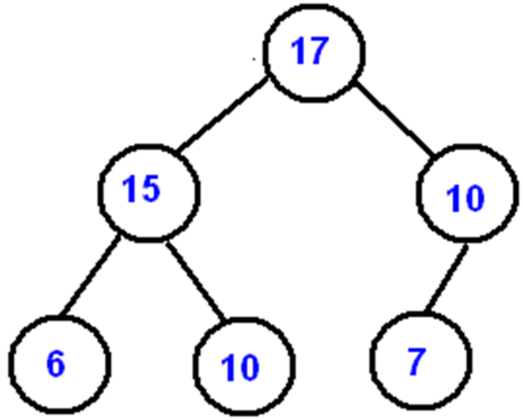
- *min-heap*:
  - the value of each node is **greater than or equal to** the value of its **parent**, resulting in **minimum-value at the root**.
    - 각 노드의 값은 자신의 children의 값보다 크지 않다

  A[*Parent*(*i*)] ≤ A[*i*] for all nodes *i* > 1



- *max-heap*:
  - the value of each node is **less than or equal to** the value of its **parent**, resulting in **maximum-value at the root**.
    - 각 노드의 값은 자신의 children의 값보다 작지 않다

  A[*Parent*(*i*)] ≥ A[*i*] for all nodes *i* > 1



Heap Height

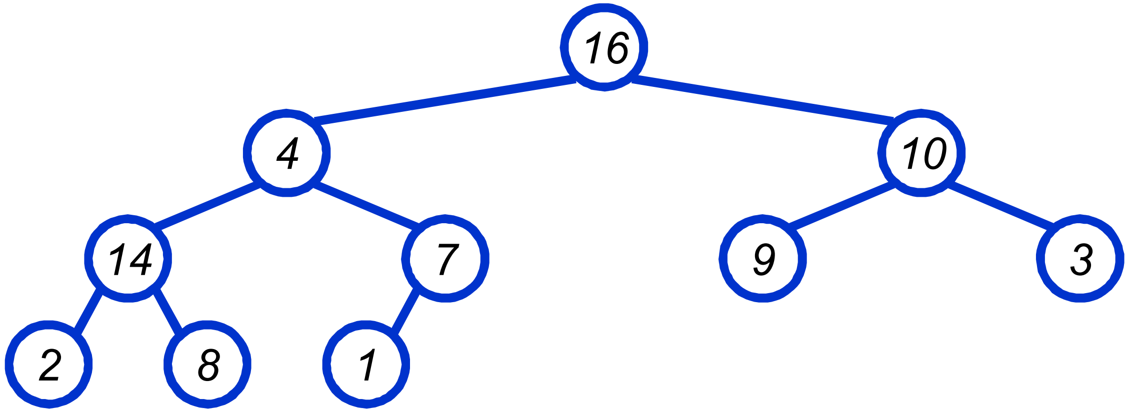
- Definition of HEAP HEIGHT:
  - The *height* of a node in the tree = the number of edges on the longest downward path to a leaf
  - The height of a tree = the height of its root
- *What is the height of an n-element heap?*
  - **`Ceiling(log2(n)): a smallest integer greater than log2(n)`**
- Basic heap operations take at most time proportional to the height of the heap

Heap Operations: MaxHeapify()

- `MaxHeapify()`: to keep the max-heap property
  - Inputs: Array $\mathrm{A}[\,]$ (or binary tree), index i in array
  - Precondition (input): the binary trees rooted at $\mathrm{LEFT}(i)$ and $\mathrm{RIGHT}(i)$ are max-heaps
    - Note: $\mathrm{A}[i]$ may be smaller than its children, but its left and right subtrees satisfy max-heap properties, i.e., $\mathrm{A}[2i]$ and $\mathrm{A}[2i+1]$ are larger than or equal to ALL OF THEIR CHILDREN
  - Postcondition (output): The subtree rooted at index i is a max-heap
    - $\mathrm{A}[i]$ is larger than or equal to ALL OF ITS CHILDREN
  - Action: let the value of the parent node FLOAT-DOWN so subtree at i satisfies the max-heap property

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

MaxHeapify() Example

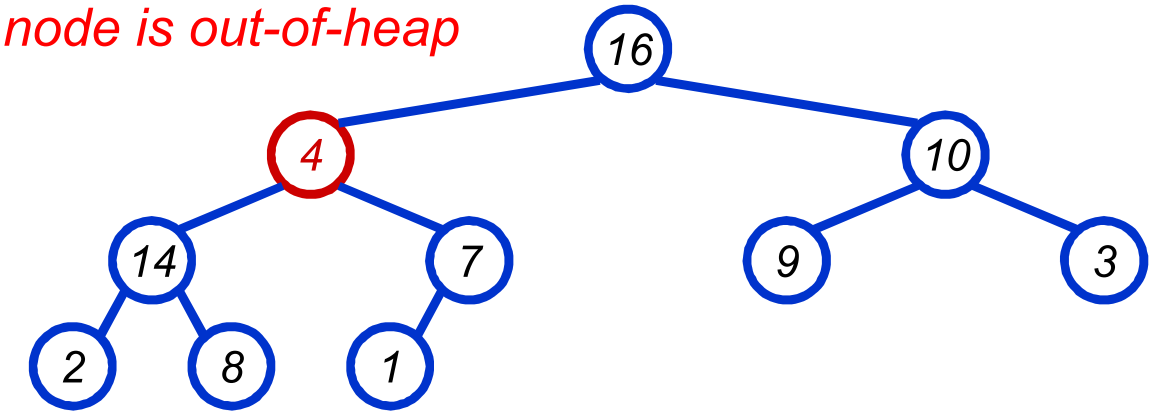


A =

16	4	10	14	7	9	3	2	8	1

MaxHeapify() Example

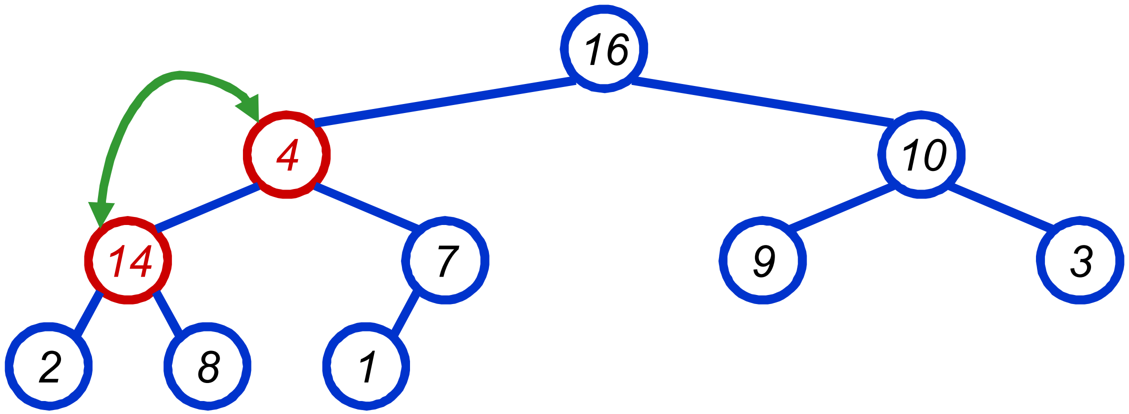
Only a single node is out-of-heap



A =

16	4	10	14	7	9	3	2	8	1

MaxHeapify() Example

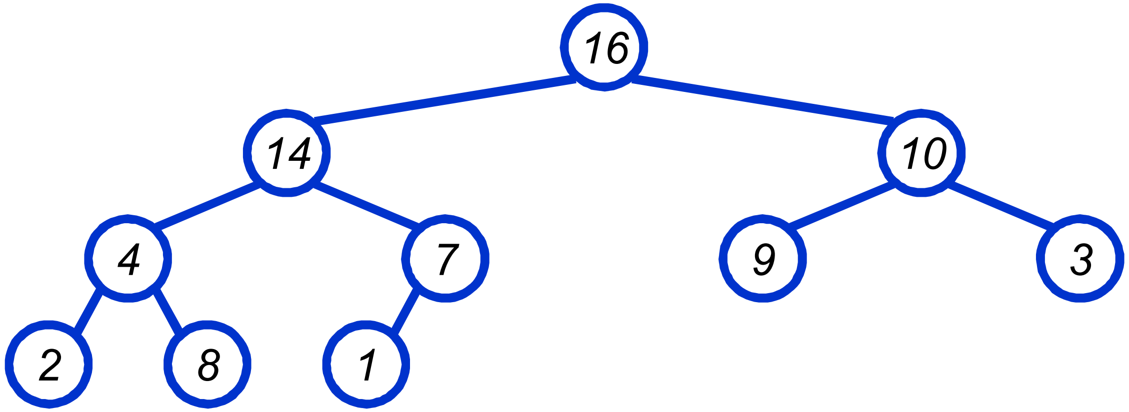


A =

16	4	10	14	7	9	3	2	8	1

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

MaxHeapify() Example

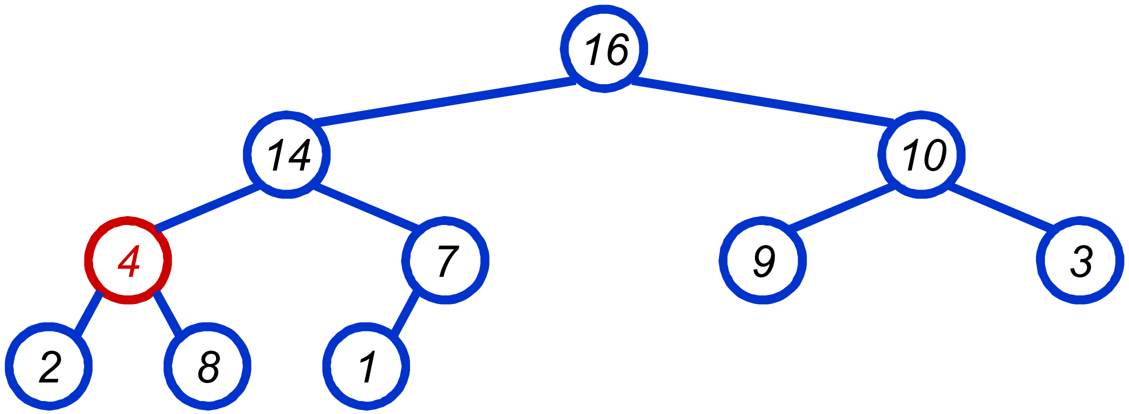


A =

16	14	10	4	7	9	3	2	8	1

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

MaxHeapify() Example

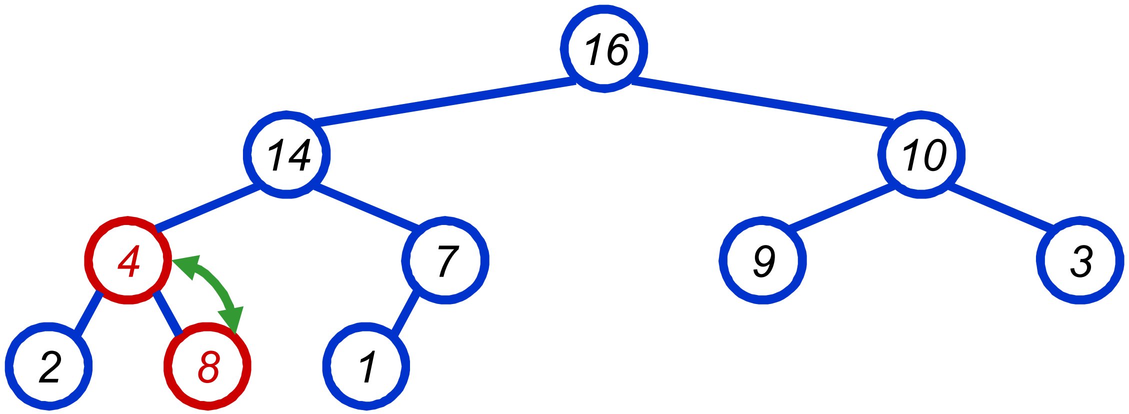


A =

16	14	10	4	7	9	3	2	8	1

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

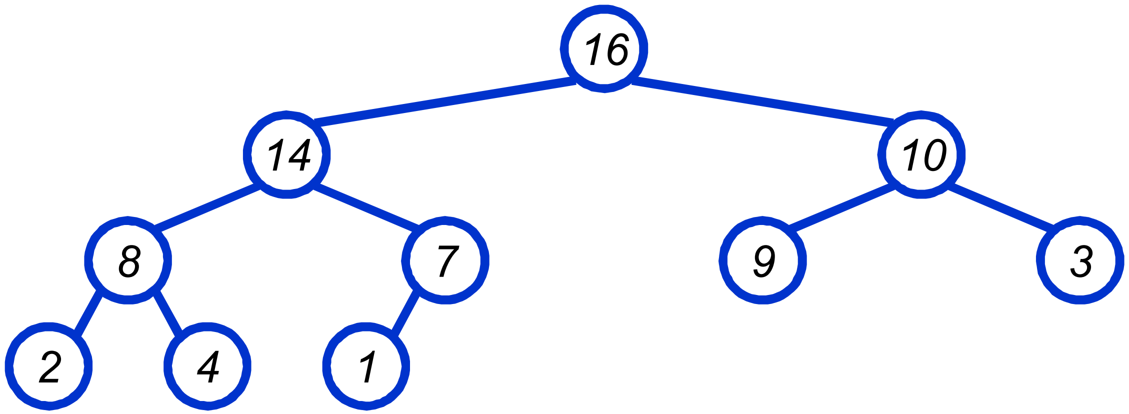
MaxHeapify() Example



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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

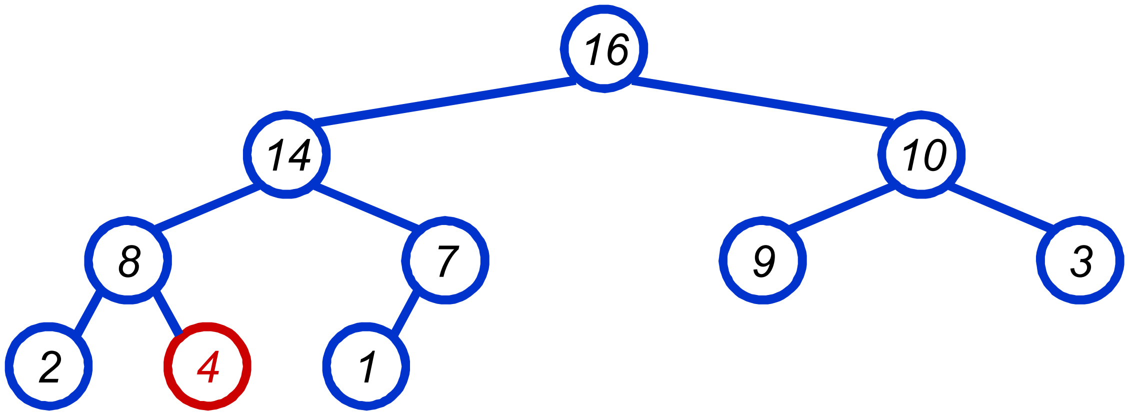
MaxHeapify() Example



A =

16	14	10	8	7	9	3	2	4	1

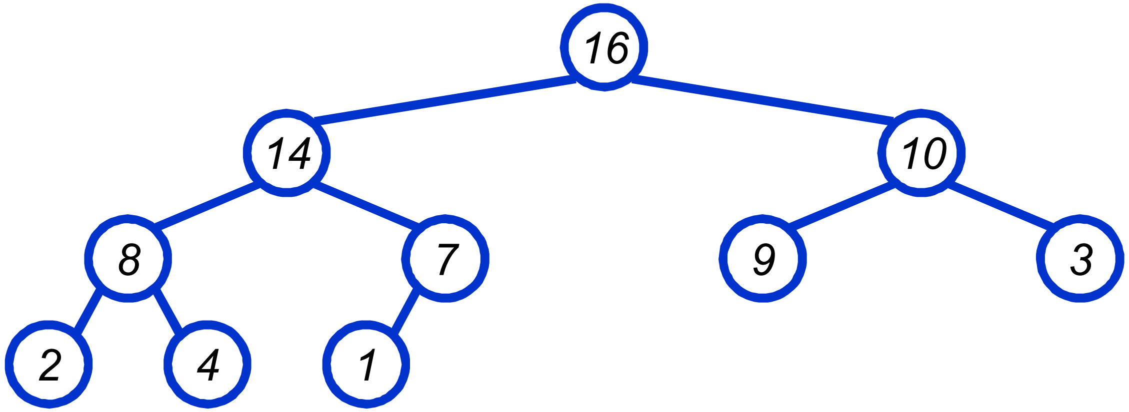
MaxHeapify() Example



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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

MaxHeapify() Example



A =

16	14	10	8	7	9	3	2	4	1

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Heap Operations: MaxHeapify()

i: index of a node that is out-of-heap
→ May occur when the value of node i changes

```
MaxHeapify(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 swap(A[i],A[largest])
         MaxHeapify(A,largest)    // sub-root is changed
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

MaxHeapify() time complexity

- *Aside from the recursive call, what is the running time of* ***`MaxHeapify()`****?*
  - Each call to MaxHeapify takes some constant c steps, because root is compared with direct children of left and right subtrees
- *How many times can* ***`MaxHeapify()`*** *recursively call itself?*
  - MaxHeapify is recursively called "**AT MOST**" h times, where h is the height of the subtree starting at i
    - Why is it not 2**h* times? – only one of left and right subtree is changed
- *Worst-case running time of* ***`MaxHeapify()`*** *on a heap of size n?*
  - for all inputs, **AT MOST** ch steps are needed
  - the worst case time complexity is $O(h) = O(\log n)$ where h is the subtree height with root i

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing MaxHeapify(): 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?*
- *In other words, what is THE LARGEST (WORST CASE) number of elements in the subtree selected at the next recursion step?*
  - Draw it
  - Note: for a full binary tree, #(leaf nodes) = #(non-leaf nodes)+1

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing MaxHeapify(): Formal

- Full: bottom row is full, #(leaf nodes) = #(non-leaf nodes)+1
- Complete, but unbalanced most: leaf nodes in the bottom is ½ full
  - $n_{leaf,left} = \frac{(n_{non-leaf}+1)}{2}$, #(leaf nodes in l-tree) = (#(non-leaf nodes)+1)/2
  - $n_{leaf,right} = 0$, #(leaf nodes in r-tree) = 0
  - $n_{left} = n_{leaf,left} + \frac{n_{non-leaf}}{2} = \frac{(2n_{non-leaf}+1)}{2} \cong n_{non-leaf}$
  - $n_{right} = n_{leaf,right} + \frac{n_{non-leaf}}{2} = \frac{n_{non-leaf}}{2}$
  - $\therefore n_{left}{:}\, n_{right} = 2{:}\,1 = \frac{2}{3}n_{tree}{:}\,\frac{1}{3}n_{tree}$
- If the left tree is selected, the next recursion of MaxHeapify() should be performed on $\frac{2}{3}n_{tree}$ nodes
- So time taken by **`MaxHeapify()`** is given by $T(n) \le T(2n/3) + \Theta(1)$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing MaxHeapify(): 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, **`MaxHeapify()`** takes logarithmic time

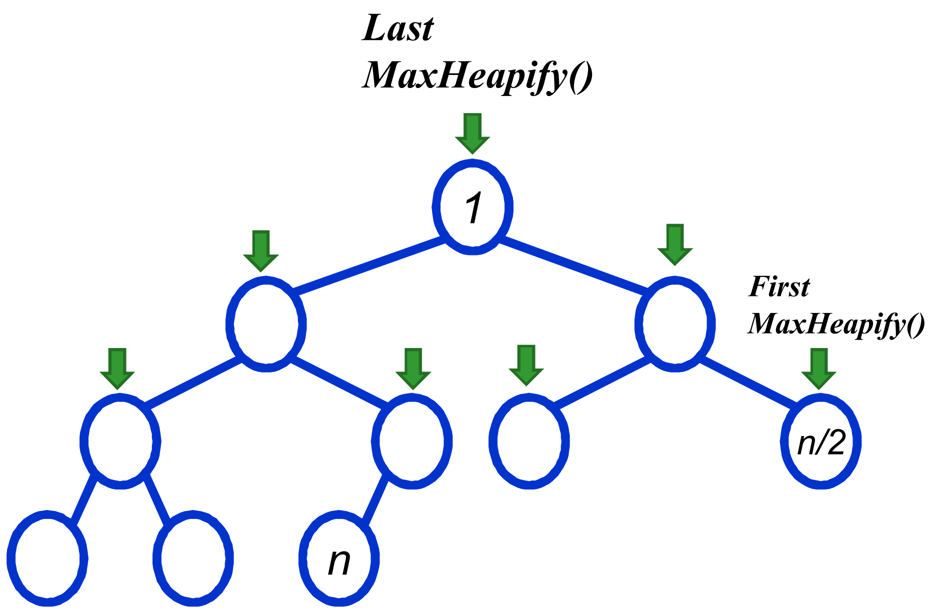
Heap Operations: BuildMaxHeap()

- Question: How efficiently can we build a heap?
- Idea:
  - FIRST create a binary tree (stick each element into a node of the tree) OR put all the elements in an array
  - THEN use MaxHeapify on non-leaf nodes
    - Bottom to top to satisfy the precondition of Heapify
- We can build a heap in a bottom-up manner by running **`MaxHeapify()`** on successive subarrays

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

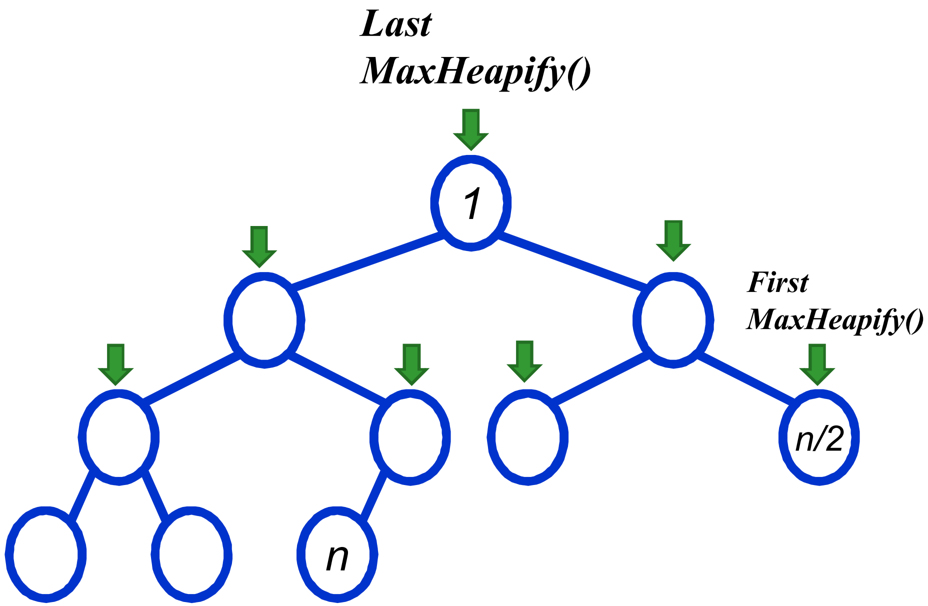
Heap Operations: BuildMaxHeap()

- Leaf nodes
  - Fact: for array of length *n*, all elements in range A[$\lfloor$n/2$\rfloor$ + 1 .. n] are heaps
    - *Why? – a leaf node has no child*
- BuildMaxHeap() in a bottom-up manner:
  - Walk **BACKWARDS** through the array from n/2 to 1, calling **`MaxHeapify()`** on each node.
  - Order of processing guarantees that the children of node *i* are heaps when *i* is processed



BuildMaxHeap()

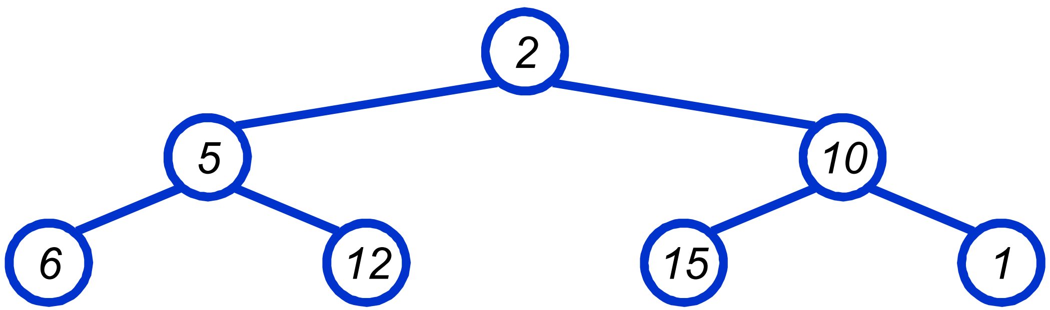
```
// given an unsorted array A
// make A a heap
BuildMaxHeap(A)
{
   heap_size(A) = length(A);
   for (i = ⌊length[A]/2⌋
        downto 1)
        MaxHeapify(A, i);
}
```



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

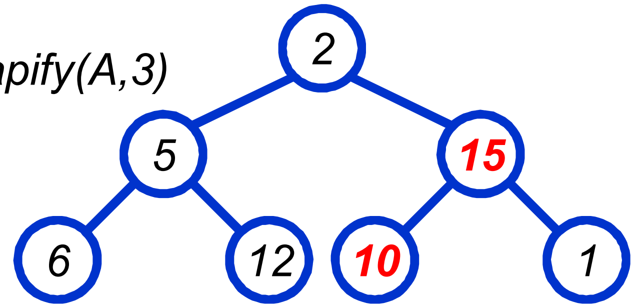
BuildMaxHeap() Example

- Appy BuildMaxHeap() to the binary tree below
  A = {2, 5, 10, 6, 12, 15, 1}
  - `Note: Since length(A) =7,`
    $\lfloor \texttt{length[A]/2} \rfloor = \lfloor 3.5 \rfloor = 3$

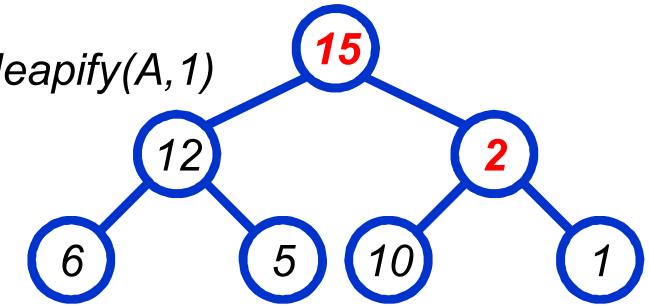


BuildMaxHeap() Example

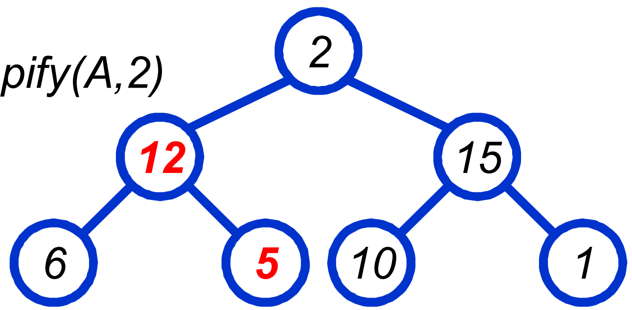
MaxHeapify(A,3)



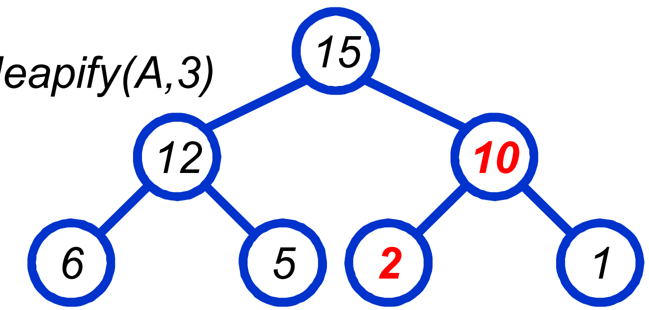
MaxHeapify(A,1)



MaxHeapify(A,2)



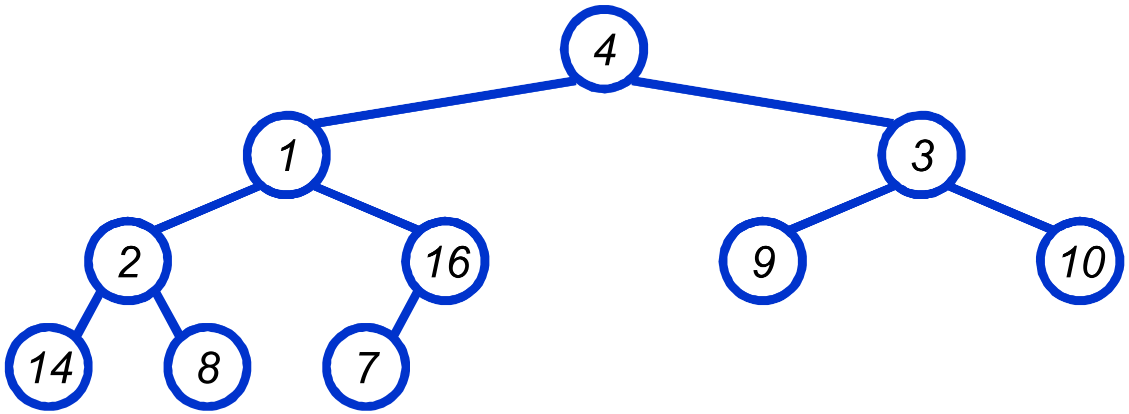
MaxHeapify(A,3)



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BuildMaxHeap() Example

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



Analyzing BuildHeap()

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

BuildMaxHeap: Better Analysis

- The running time needed at each level of the tree
  - For a node at height h, the worst running time is $(c*h)$
  - There are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in the tree

		15				height = 2
	12			10		height = 1
6	5		2		1	height = 0

  - So, worst-case running time of all nodes at height h is **$c * h * \lceil n/2^{h+1} \rceil$**

- The height varies from **0 to $\lfloor \log2(n) \rfloor$**

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BuildMaxHeap: Better Analysis

- Sum this over all the nodes in the tree:

$$T(n) = \sum_{h=0}^{tree\ height} ch\lceil n/2^{h+1}\rceil \le \sum_{h=0}^{\lfloor \log_2 n\rfloor} ch\lceil n/(2\cdot 2^h)\rceil$$

$$\le \frac{cn}{2}\sum_{h=0}^{\infty}\frac{h}{2^h}$$

$$= \frac{cn}{2}\left(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\frac{5}{32}+\cdots\right) \le \frac{cn}{2}c_2 = \frac{cc_2}{2}n \in O(n)$$

  - Proof: http://courses.washington.edu/css343/zander/NotesProbs/heapcomplexity

    https://math.stackexchange.com/questions/1755708/summation-of-an-expression-sum-h-0-ln-n-frach2h

- So, the worst case time complexity in O(n)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

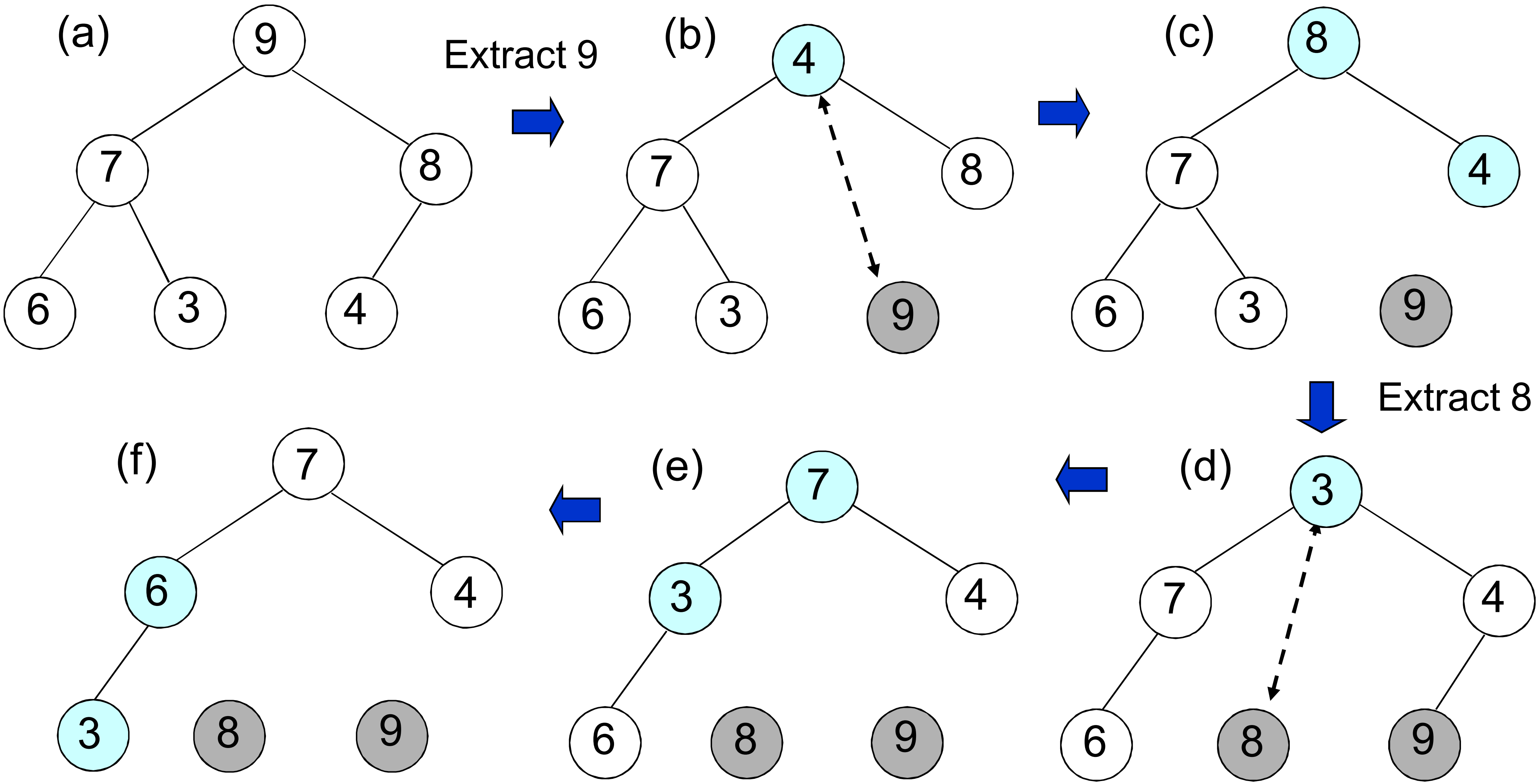
Formulation of an **ascending** sort algorithm using **MaxHeapify**

ASCENDING HEAP SORT

Heapsort

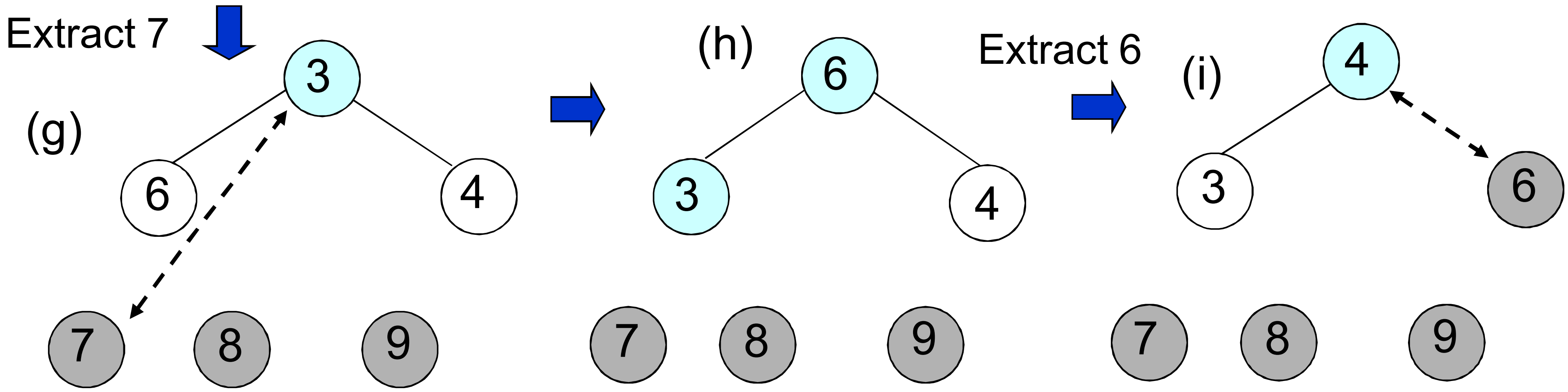
- Basic idea: convert the input array into a heap, and extract one item at a time to build a sorted list
  - Can we make it in-place (without extra array)?
- Given **`BuildHeap()`**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard it by swapping with element at A[*n*]
    - A[*n*] now contains correct value (in order)
    - Decrement heap_size[A]
  - Restore heap property at A[1] by calling **`Heapify()`**
  - Repeat, swapping A[1] for A[heap_size(A)]

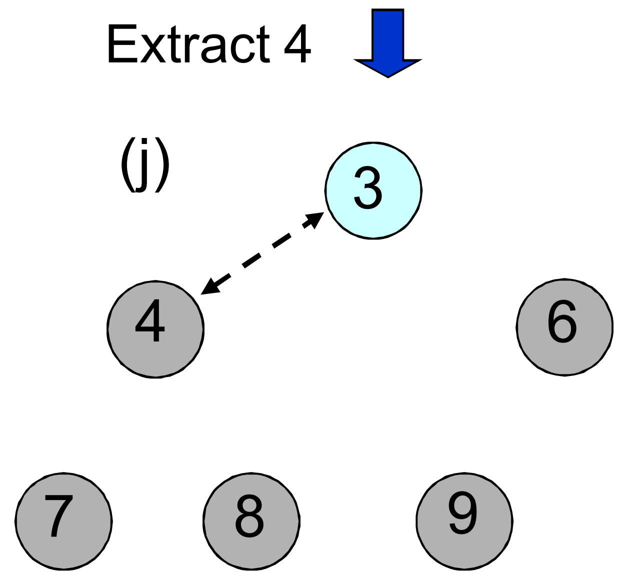
Heapsort, Ascending



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Heapsort, Ascending





Single element at 3,
no need to MaxHeapify,
done sorting

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Heapsort, Ascending

```
HeapsortAscending(A)
/* Ascending sort */
{
    BuildMaxHeap(A);       /* build max heap */
    for (i = length(A) downto 2)
    {
        Swap(A[1], A[i]);   /* send max to last */
        heap_size(A) -= 1;
        MaxHeapify(A, 1);
    }
}
```

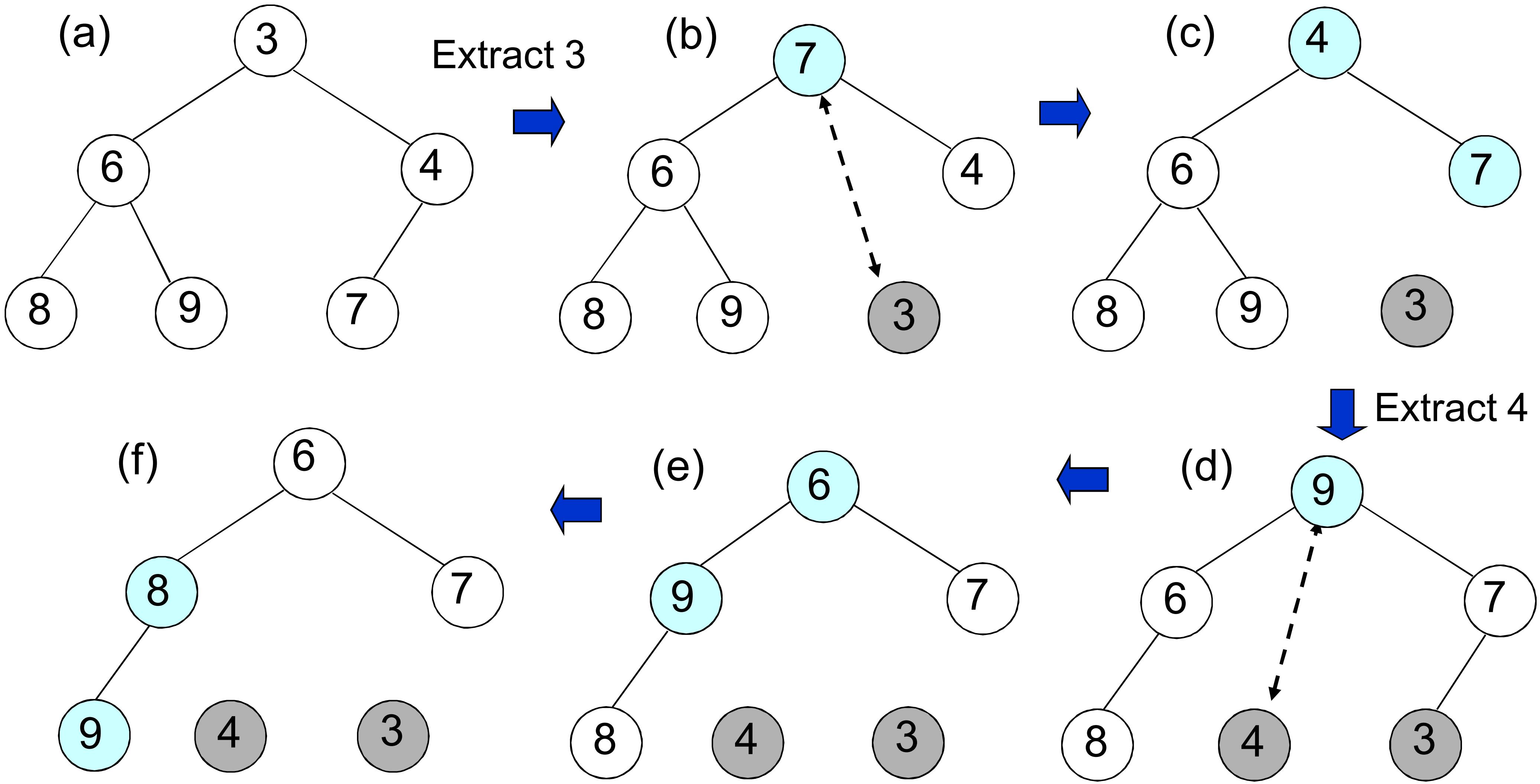
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Formulation of an **descending** sort algorithm using **MinHeapify**

DESCENDING HEAP SORT

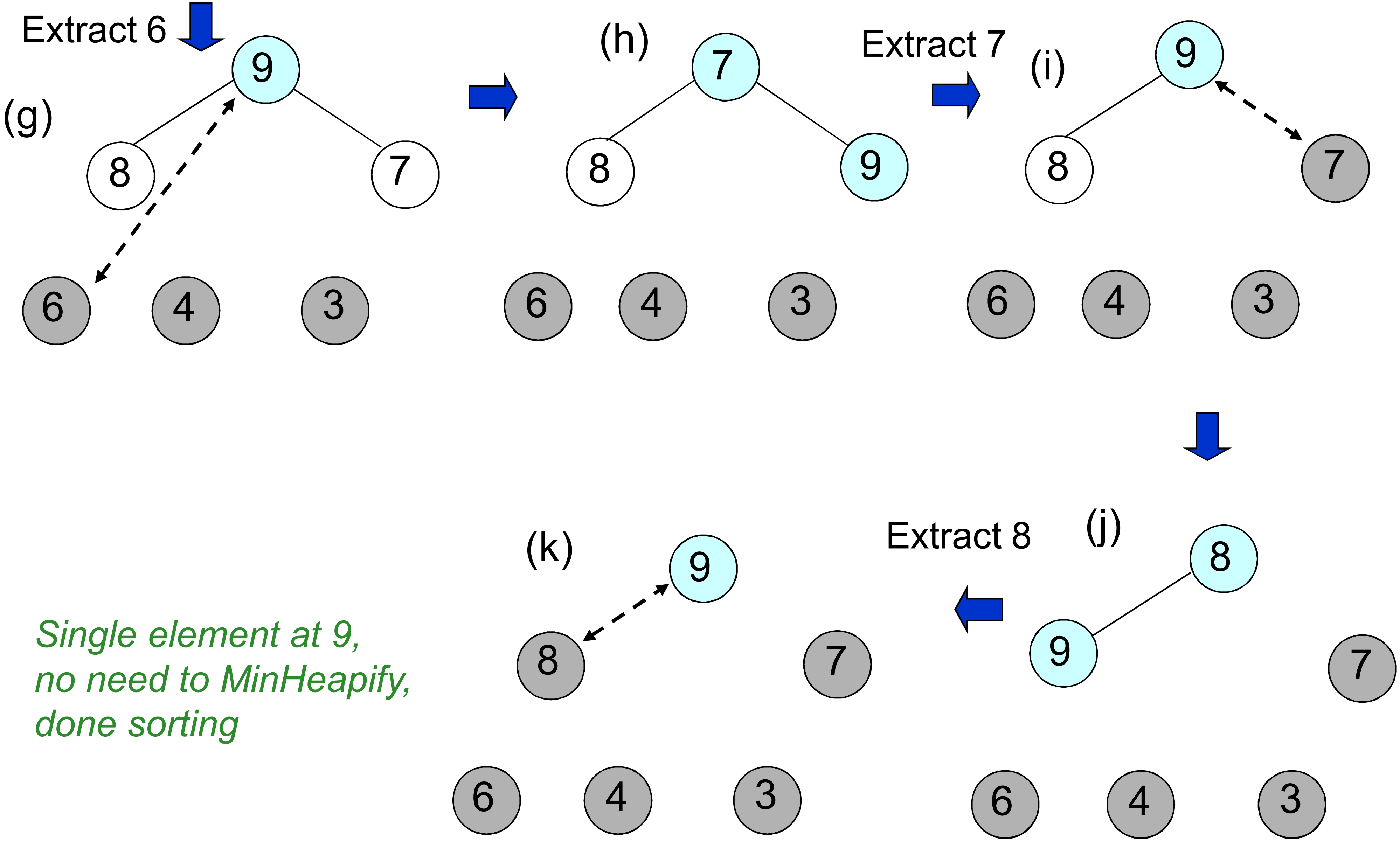
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Heapsort, Descending



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Heapsort, Descending



Single element at 9,
no need to MinHeapify,
done sorting

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Heapsort, Descending

```
HeapsortDescending(A)
/* Descending sort */
{
     BuildMinHeap(A);        /* build min heap */
     for (i = length(A) downto 2)
     {
          Swap(A[1], A[i]); /* send min to last */
          heap_size(A) -= 1;
          MinHeapify(A, 1);
     }
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Heapsort

- The initial call to **`BuildHeap()`** takes O(n) time
- Each of the n - 1 calls to **`Heapify()`** takes O(lg n) time
- Thus the total time taken by **`HeapSort()`**
  = O(n) + (n - 1) O(lg n)
  = O(n) + O(n lg n)
  = O(n lg n)
  - Even in the worst case, O(n lg n) complexity!

Comparing Sorting Algorithms

	Worst Case	Average Case
Selection Sort	n^2	n^2
Bubble Sort	n^2	n^2
Insertion Sort	n^2	n^2
Mergesort (*O(n) extra space)	$n\log n$	$n\log n$
Quicksort	n^2	$n\log n$
Heapsort	$\boldsymbol{n\log n}$	$\boldsymbol{n\log n}$

Heapsort is an efficient algorithm, but in practice Quicksort usually wins

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

END OF LECTURE 4

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I