

CMP448: Algorithms



Lecture 03: Heapsort and Quicksort

Mohamed Alaa El-Dien Aly
Computer Engineering Department
Cairo University
Fall 2013

Agenda

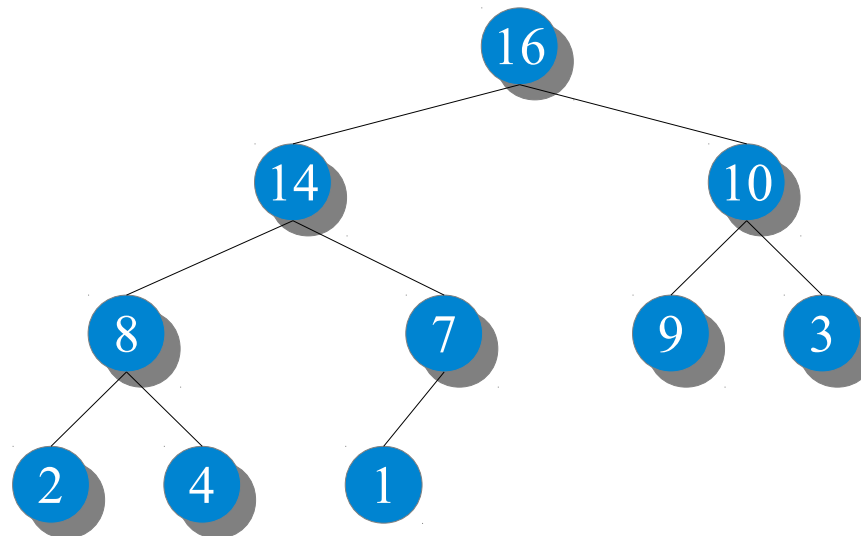
- Heapsort
- Quicksort

Acknowledgment

A lot of slides adapted from the slides of Erik Demaine and Charles Leiserson

Heaps

- “**Nearly-complete**” binary trees i.e. filled except possibly at the leaves
- Each node satisfies the “**heap property**”:
 - **Max-Heap**: the value of the node \geq values of the children
 - **Min-Heap**: the value of the node \leq values of the children

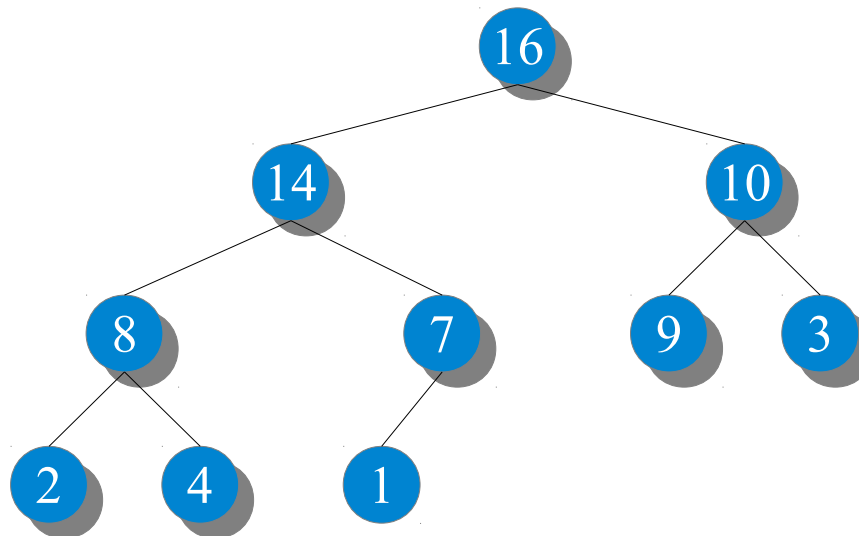


Max-Heap

Heaps

- **Max-Heap**: the value of the node \geq values of the children
- **Fact**: The maximum element is at the **root**.

Max-Heap



Array Representation

- Heaps are better represented as an array filled from root to leaves
- Three functions required:

Parent of a node i

PARENT(i)

1 return $\lfloor i/2 \rfloor$

Left child of a node i

LEFT(i)

1 return $2i$

Right child of a node i

RIGHT(i)

1 return $2i + 1$

Properties of array A

A.length: size of array A

A.heap-size: size of heap inside array $A \leq A.length$

Array Representation

Parent of a node i

$\text{PARENT}(i)$

1 return $\lfloor i/2 \rfloor$

Left child of a node i

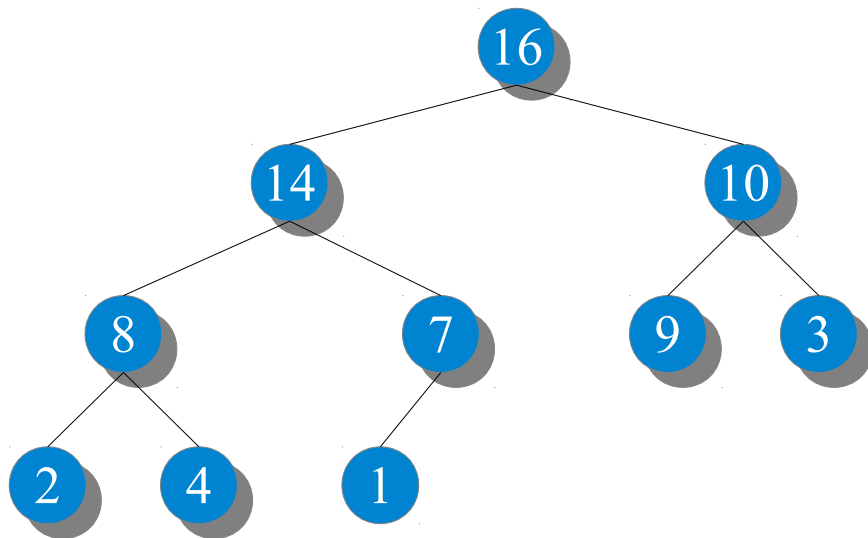
$\text{LEFT}(i)$

1 return $2i$

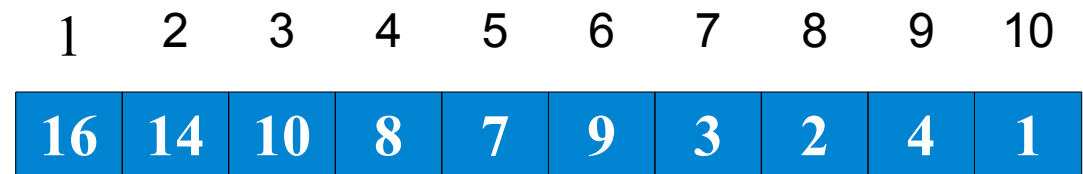
Right child of a node i

$\text{RIGHT}(i)$

1 return $2i + 1$



Conceptual Heap Representation



Array Representation

Note that indices start at 1 *not* 0.

Array Representation

Parent of a node i

$\text{PARENT}(i)$

1 return $\lfloor i/2 \rfloor$

Left child of a node i

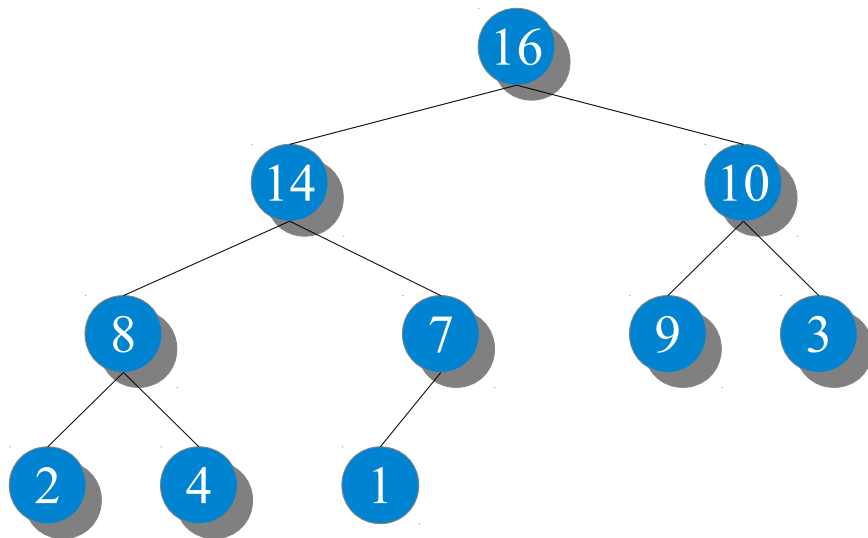
$\text{LEFT}(i)$

1 return $2i$

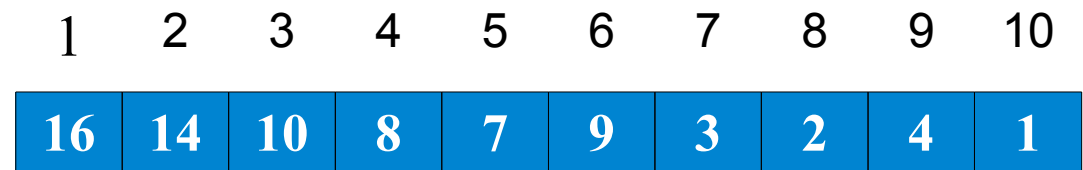
Right child of a node i

$\text{RIGHT}(i)$

1 return $2i + 1$



Conceptual Heap Representation



Array Representation

Note that indices start at 1 *not* 0.

Array Representation

Parent of a node i

$\text{PARENT}(i)$

1 return $\lfloor i/2 \rfloor$

Left child of a node i

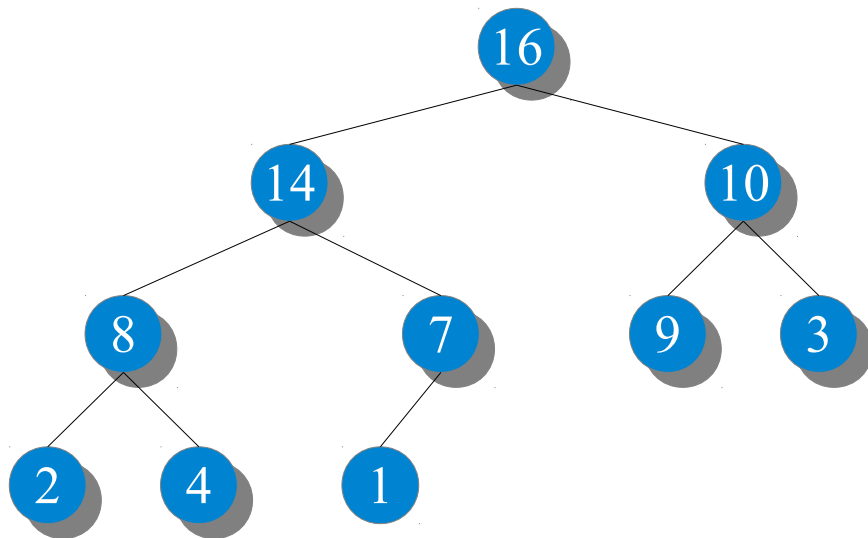
$\text{LEFT}(i)$

1 return $2i$

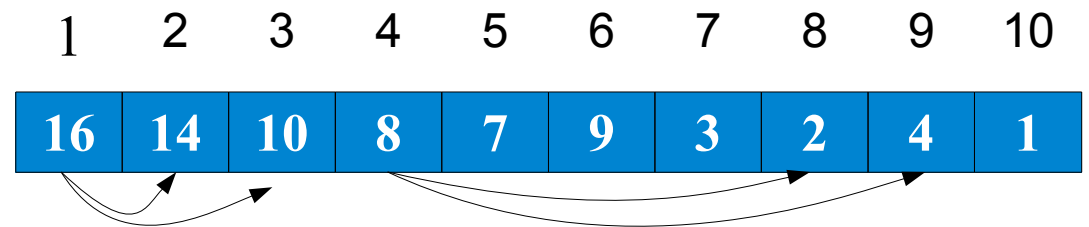
Right child of a node i

$\text{RIGHT}(i)$

1 return $2i + 1$



Conceptual Heap Representation



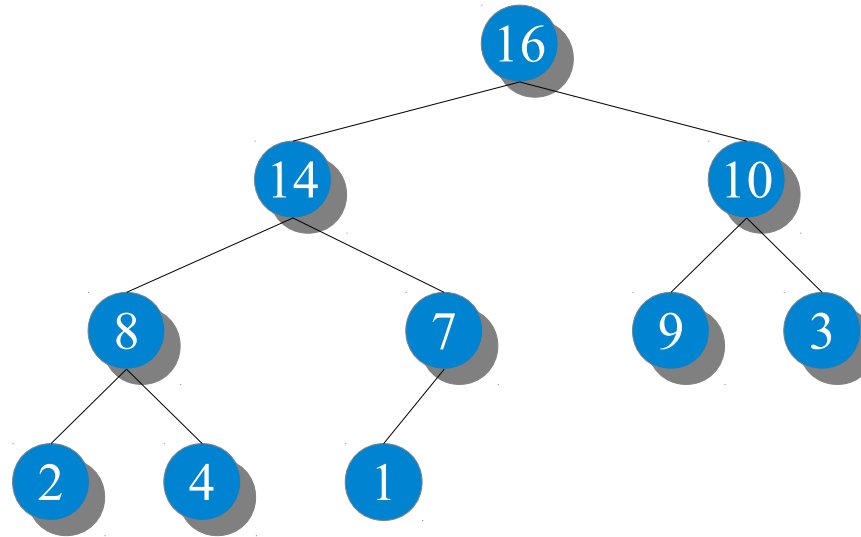
Array Representation

Note that indices start at 1 *not* 0.

Max-Heaps

Max-Heap Property

$$A[\text{Parent}(i)] \geq A[i]$$



Properties of array A

A.length: size of array A

A.heap-size: size of heap inside array $A \leq A.length$

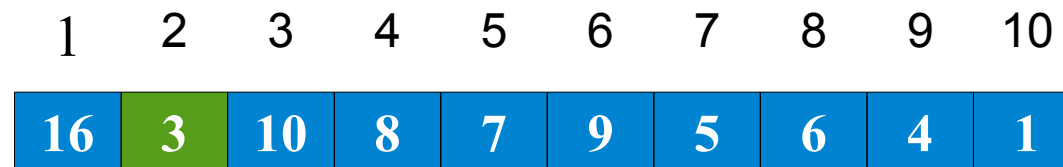
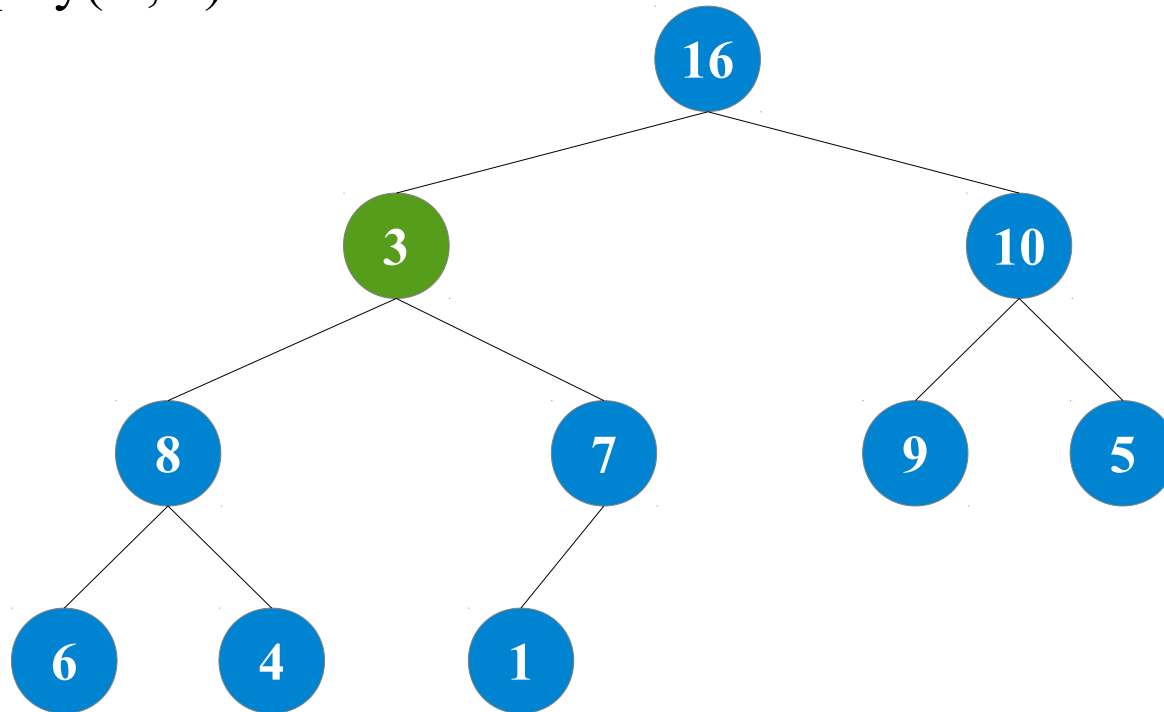
Max-Heapify

- Maintains the Max-Heap property of the heap
- Starting from a node in the heap that might violate the heap property i.e. smaller than its children
- Assumes its children are roots of Max-Heaps

```
MAX-HEAPIFY ( $A, i$ )  
1  $l = \text{LEFT}(i)$   
2  $r = \text{RIGHT}(i)$   
3 if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$   
4    $\text{largest} = l$   
5 else  $\text{largest} = i$   
6 if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$   
7    $\text{largest} = r$   
8 if  $\text{largest} \neq i$   
9   exchange  $A[i]$  with  $A[\text{largest}]$   
10  MAX-HEAPIFY( $A, \text{largest}$ )
```

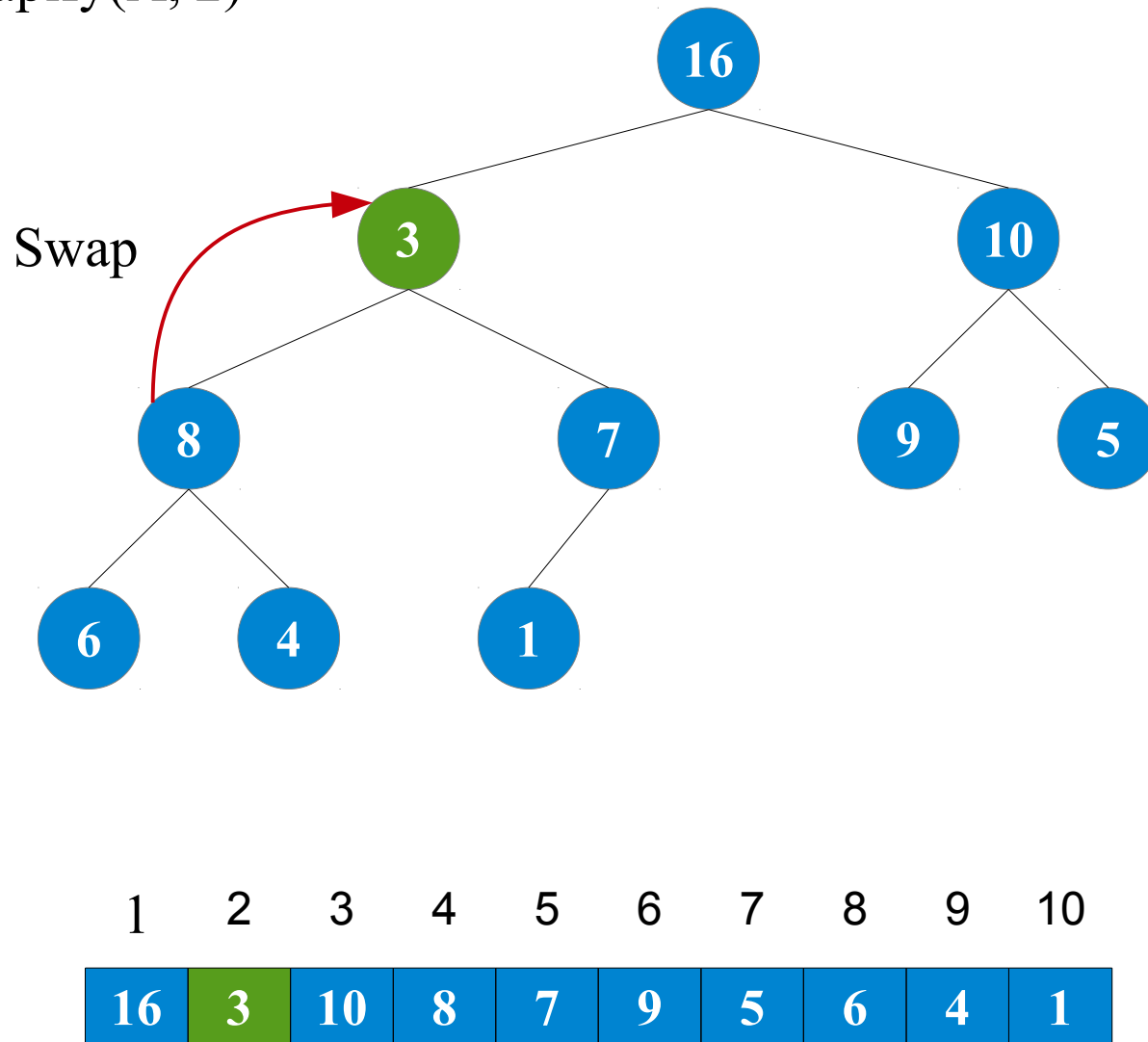
Max-Heapify

Max-Heapify(A, 2)



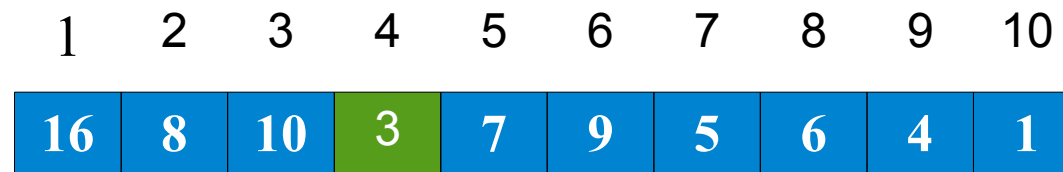
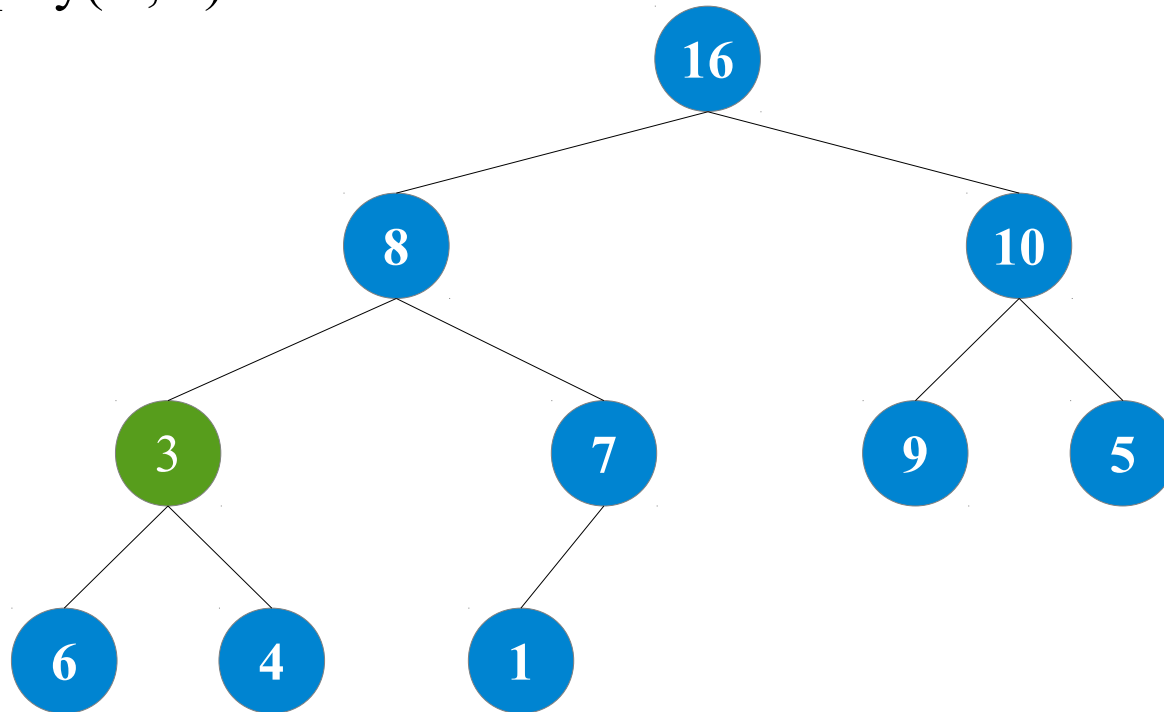
Max-Heapify

Max-Heapify(A, 2)



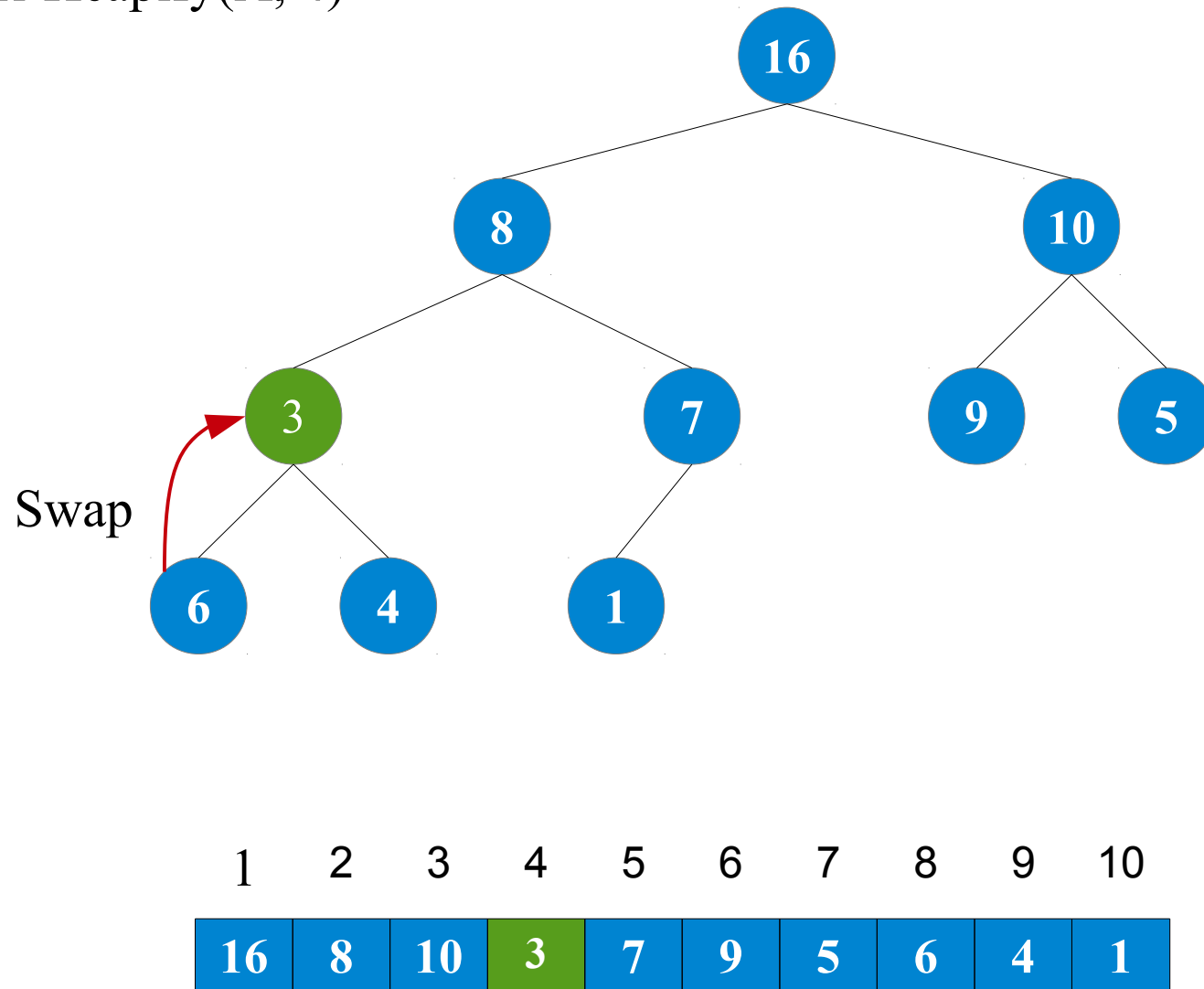
Max-Heapify

Max-Heapify(A, 4)



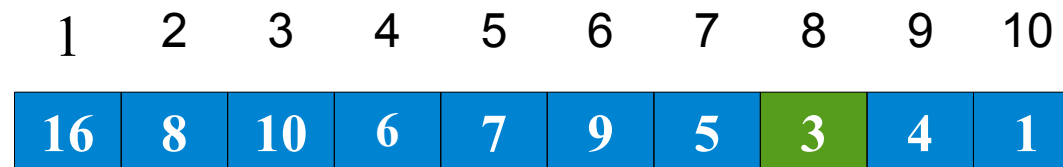
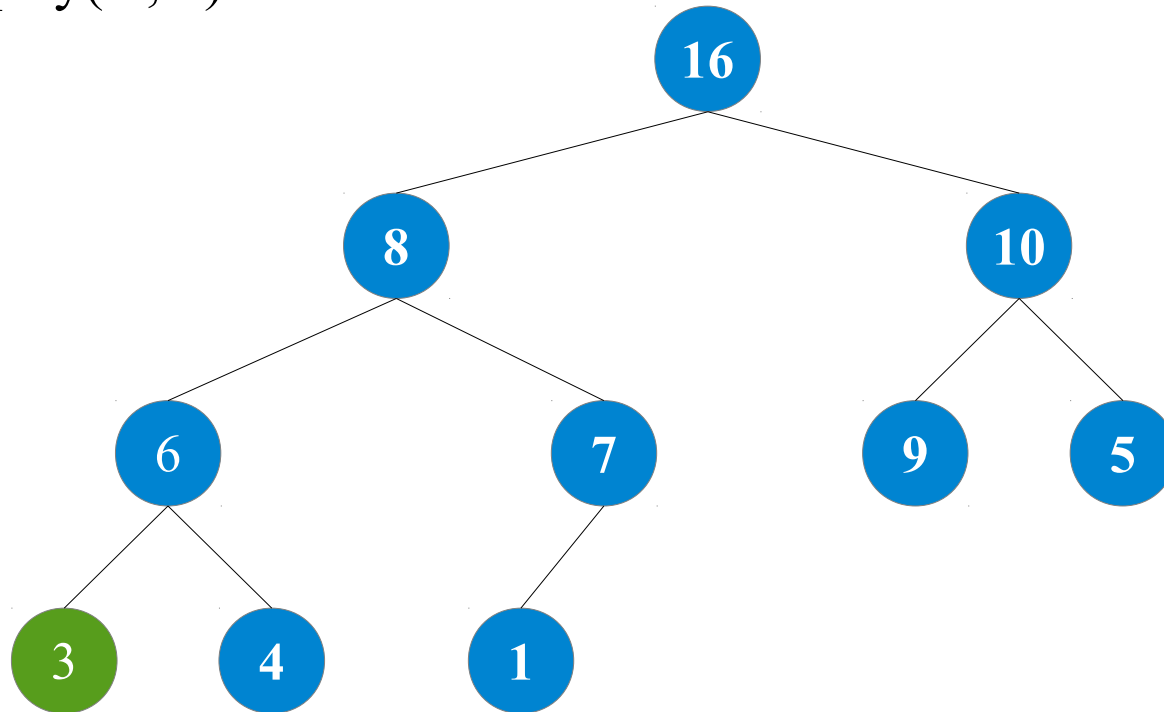
Max-Heapify

Max-Heapify(A, 4)



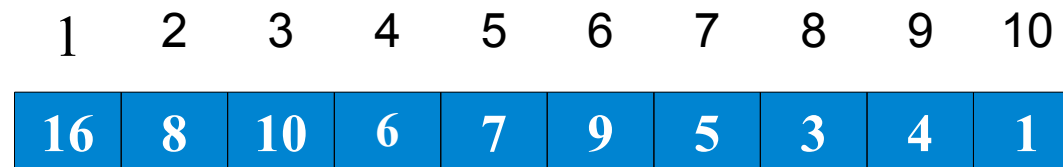
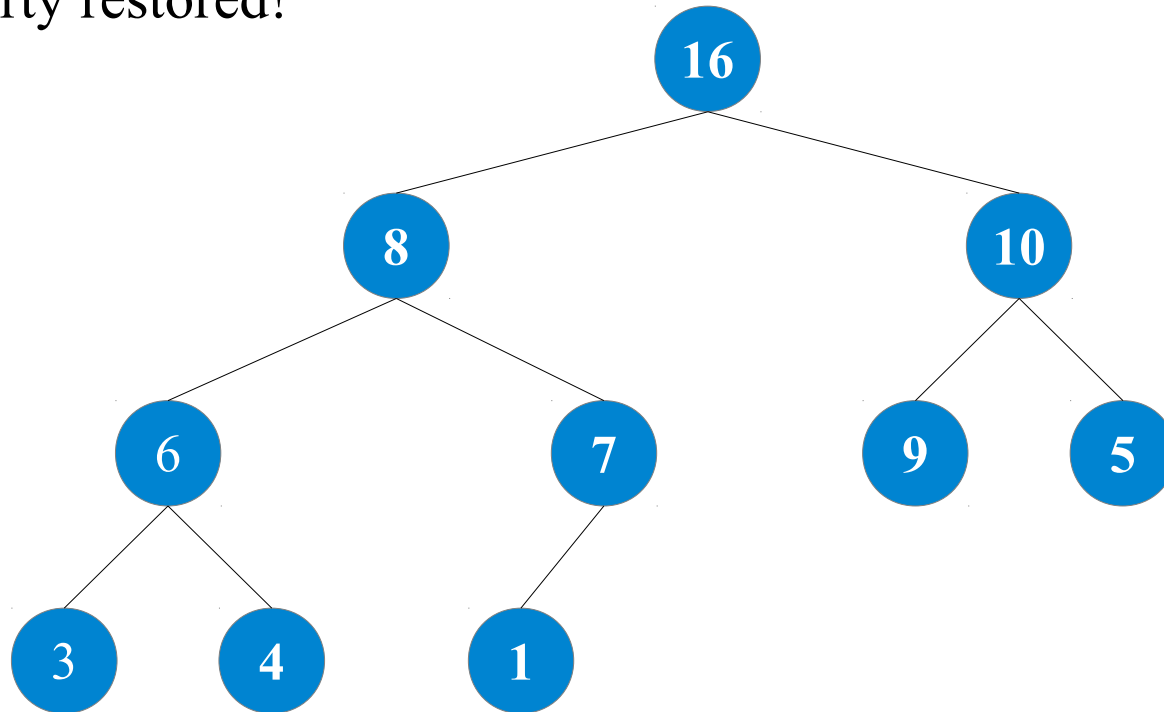
Max-Heapify

Max-Heapify(A, 8)



Max-Heapify

Heap property restored!



Max-Heapify Running Time

- Lines 1-9: $T(n) = \Theta(1)$
- Line 10: call on a subtree of node i . But what's the size of that subtree?

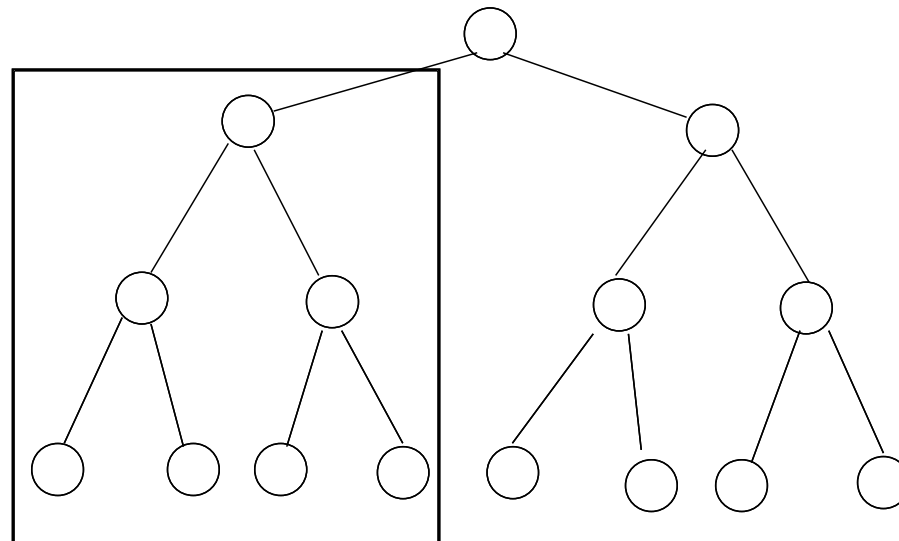
```
MAX-HEAPIFY ( $A, i$ )  
1  $l = \text{LEFT}(i)$   
2  $r = \text{RIGHT}(i)$   
3 if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$   
4    $largest = l$   
5 else  $largest = i$   
6 if  $r \leq A.\text{heap-size}$  and  $A[r] > A[largest]$   
7    $largest = r$   
8 if  $largest \neq i$   
9   exchange  $A[i]$  with  $A[largest]$   
10  MAX-HEAPIFY( $A, largest$ )
```

Max-Heapify Running Time

- Lines 1-9: $T(n) = \Theta(1)$
- Line 10: call on a subtree of node i . But what's the size of that subtree?

Best Case

In a complete binary tree, about **half** the nodes are included in the left subtree



15 nodes in tree

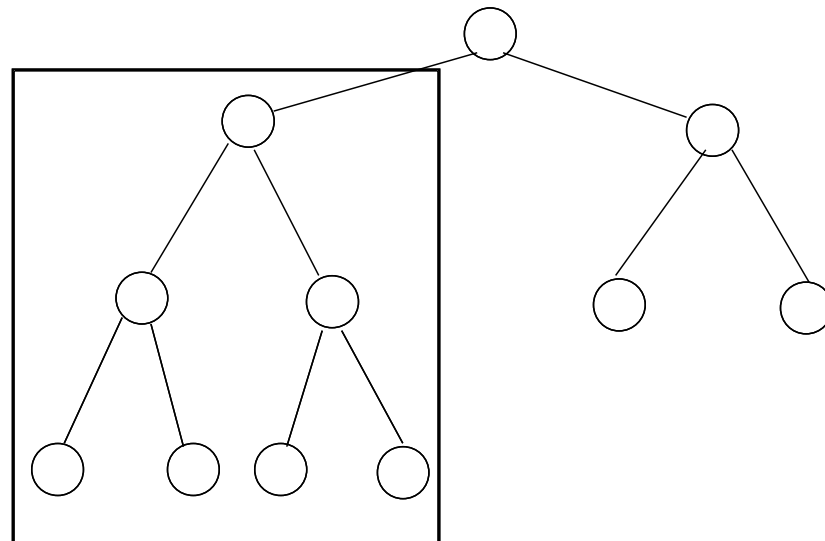
7 nodes in left subtree

Max-Heapify Running Time

- Lines 1-9: $T(n) = \Theta(1)$
- Line 10: call on a subtree of node i . But what's the size of that subtree?

Worst Case

In a binary tree where bottom is half-full, about **two-thirds** of the nodes are included in the left subtree



11 nodes in tree

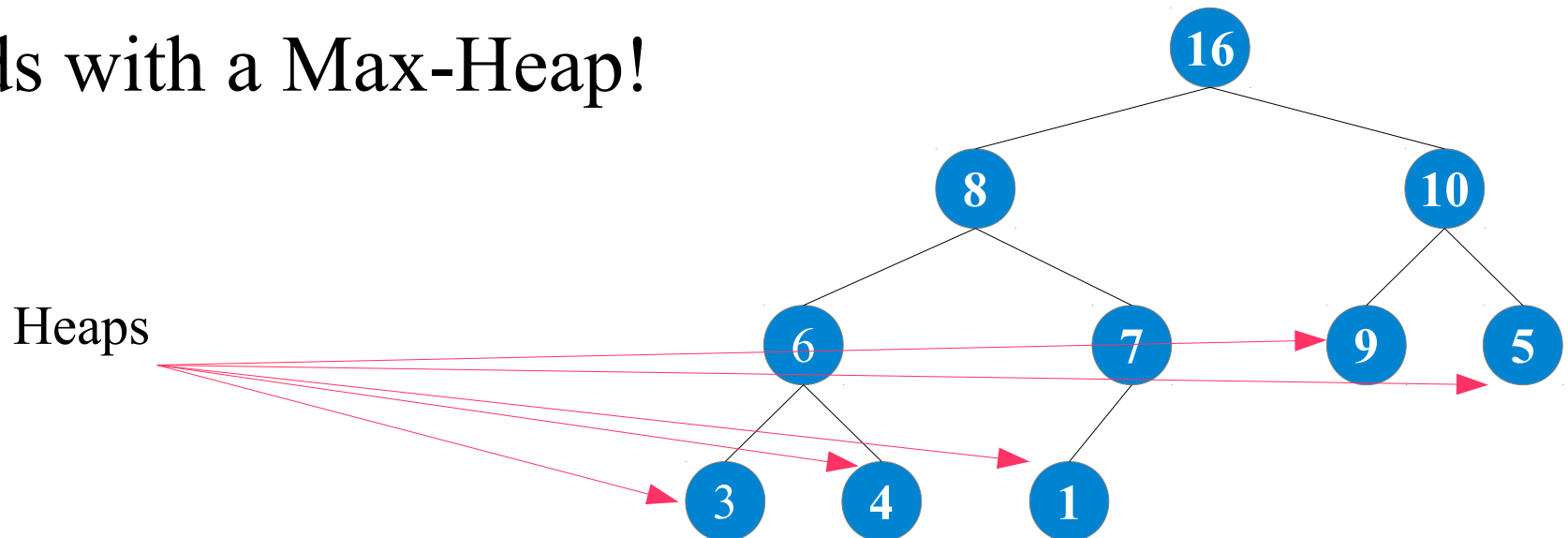
7 nodes in left subtree

Max-Heapify Running Time

- $T(n) \leq T(2n/3) + \Theta(1)$
- **Case 2** of the Master Theorem ($a = 1, b = 3/2$):
 - $T(n) = O(\lg n)$
- Can also represent it as a function of h , the height of the node
 - The height is the number of links from the node to the leaves
 - $T(n) = O(h)$

Heap Building

- Now we can restore the heap property if it's lost.
- But how do we actually build a heap from a bunch of numbers?
- Each leaf is already a heap, albeit a useless one!
- Start from the second to last level, and heapify i.e. move values violating the Heap property downwards
- Ends with a Max-Heap!



Heap Building

```
Build-MAX-HEAP(A)
```

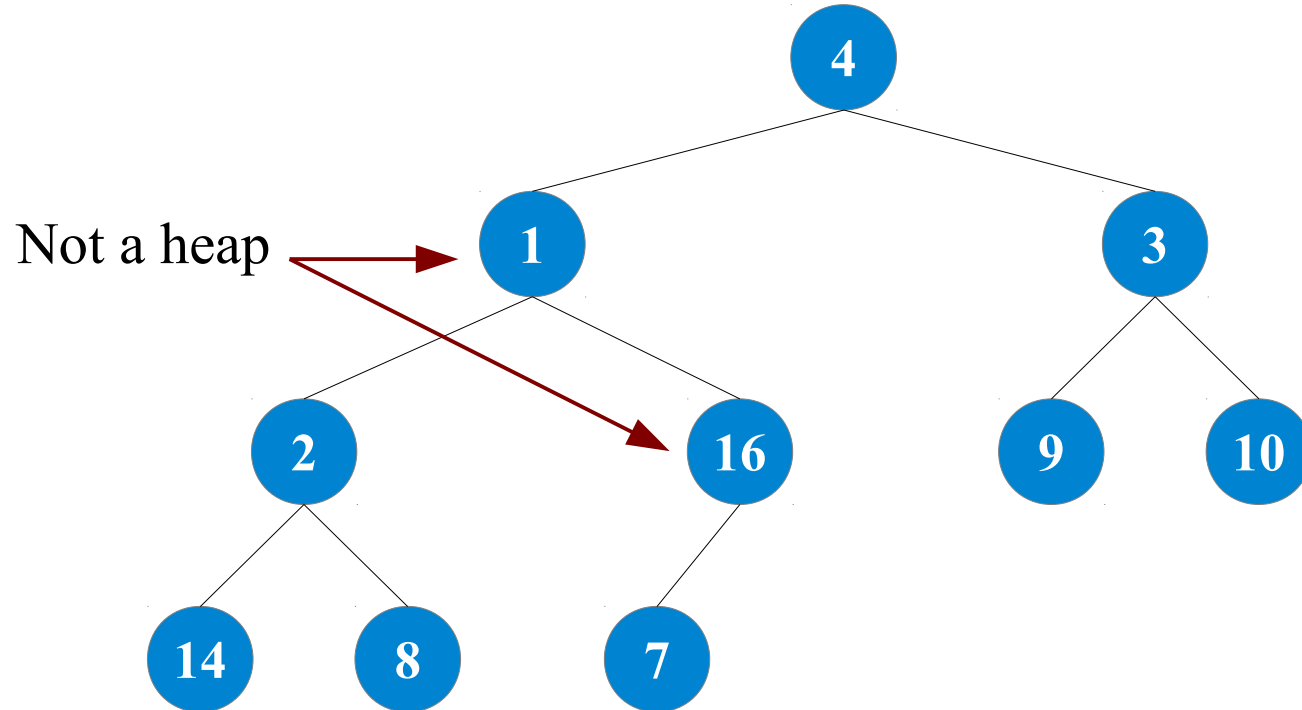
```
1 A.heap-size = A.length
```

```
2 for i =  $\lfloor A.length / 2 \rfloor$  downto 1
```

```
3   MAX-HEAPIFY(A, i)
```

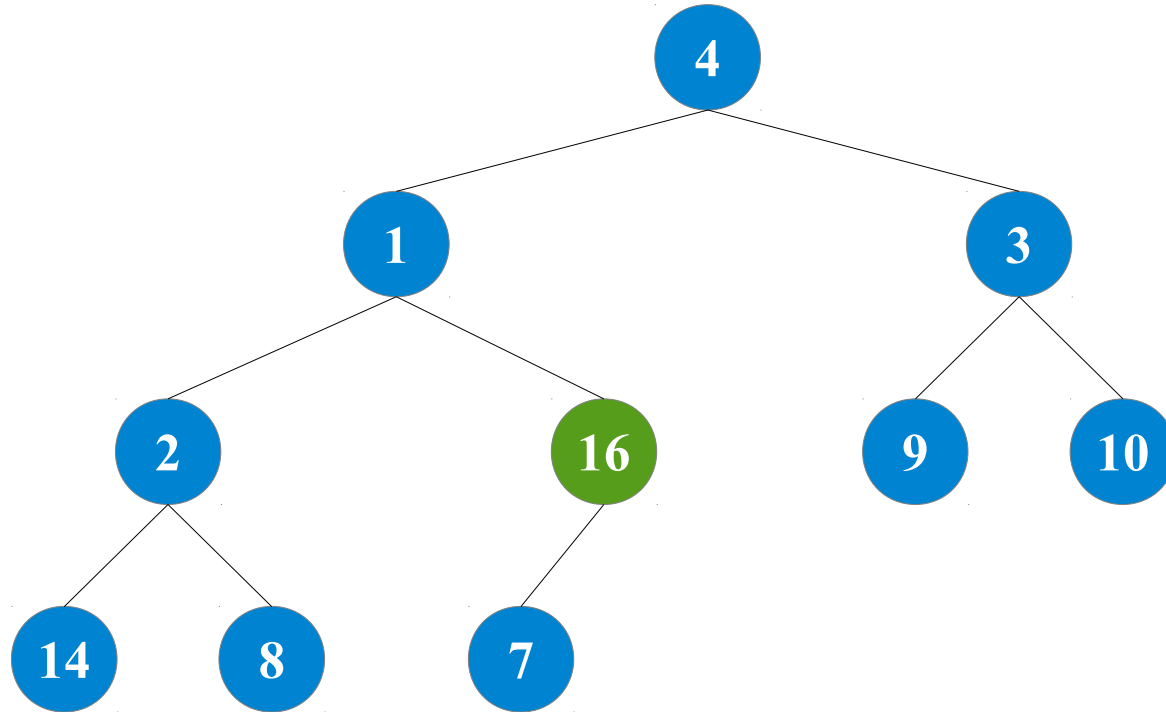
- Each leaf is already a heap, albeit a useless one!
- Start from the second to last level, and heapify i.e. move values violating the Heap property downwards
- Ends with a Max-Heap!

Heap Building



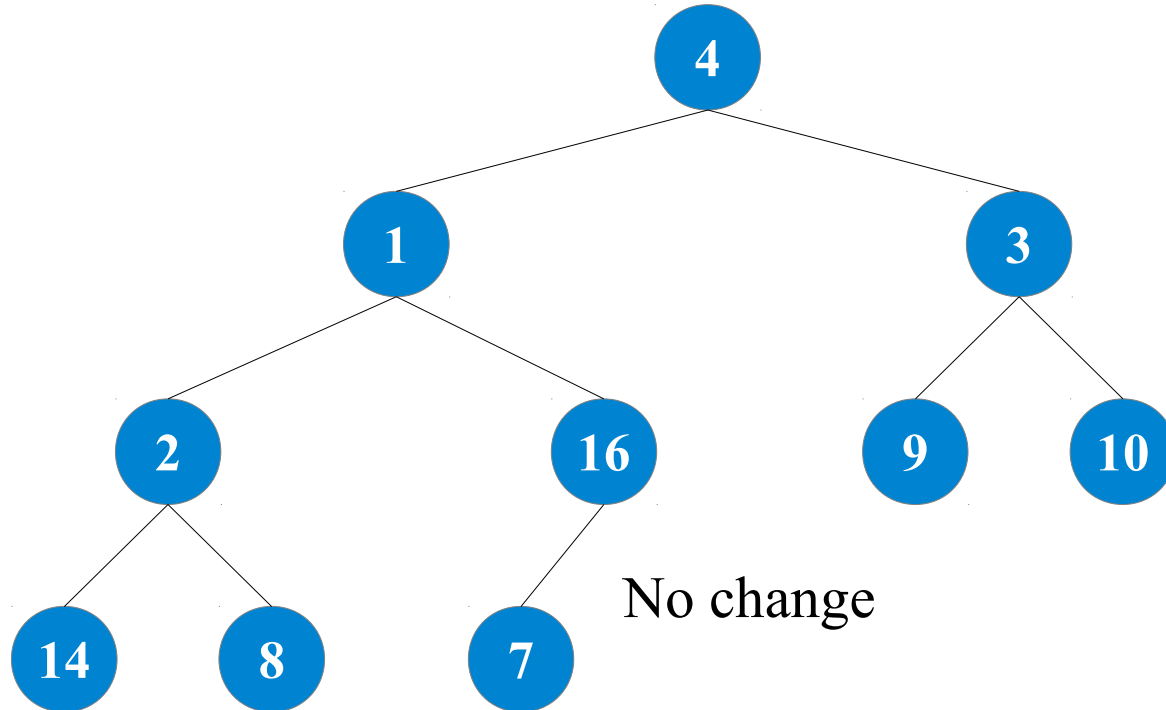
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Heap Building



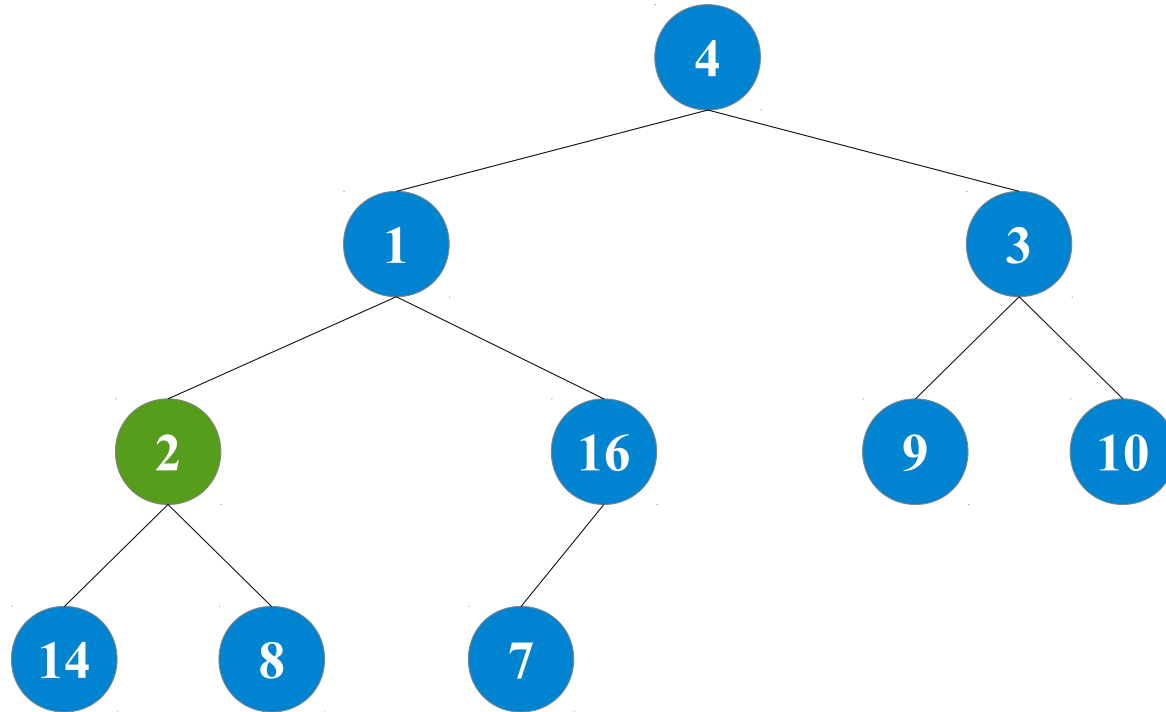
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Heap Building



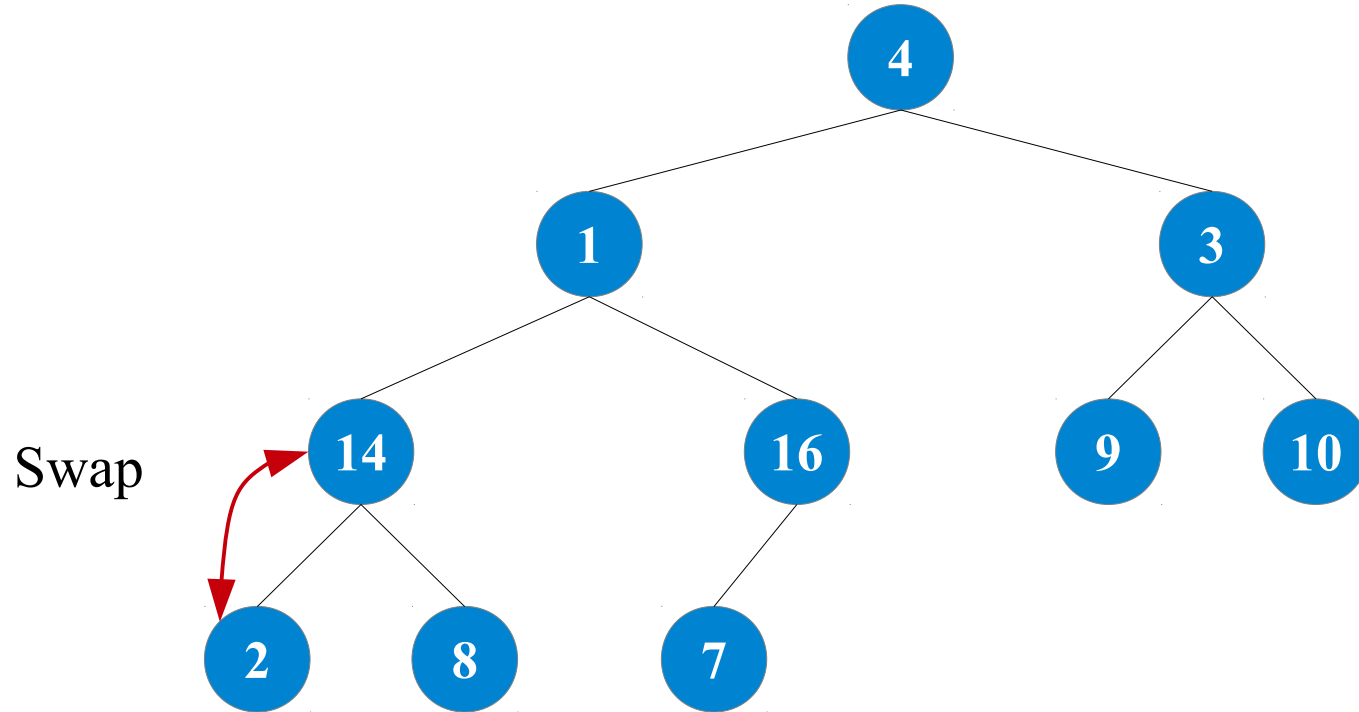
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Heap Building



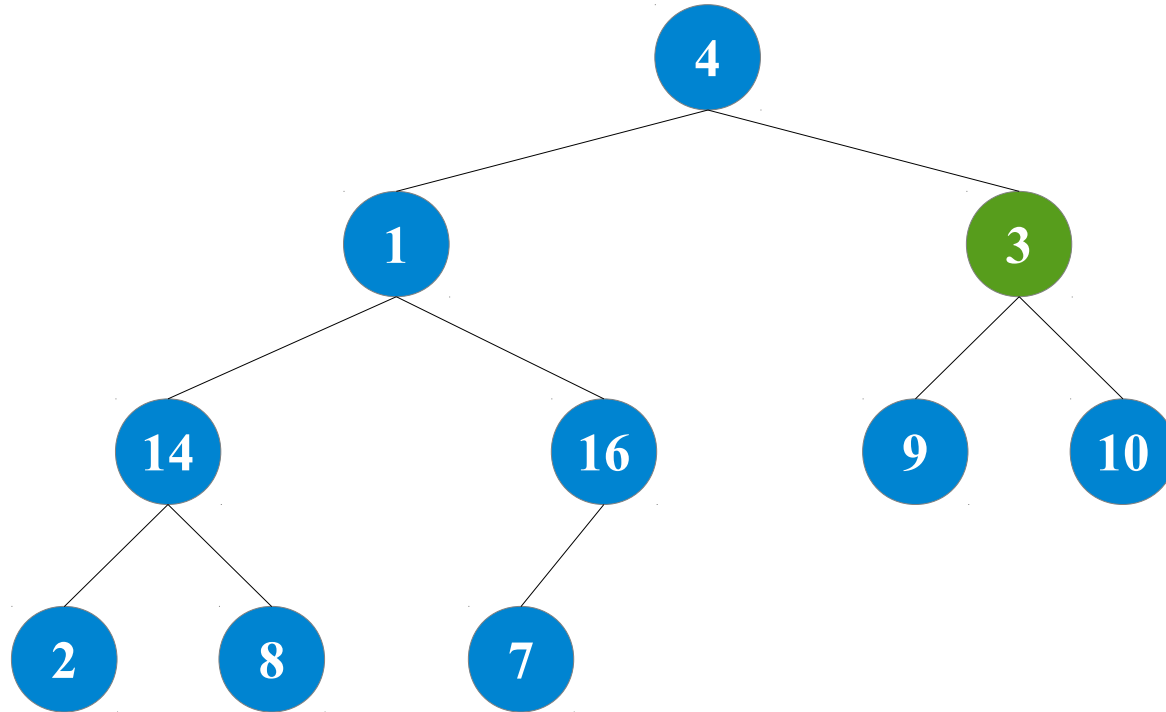
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Heap Building



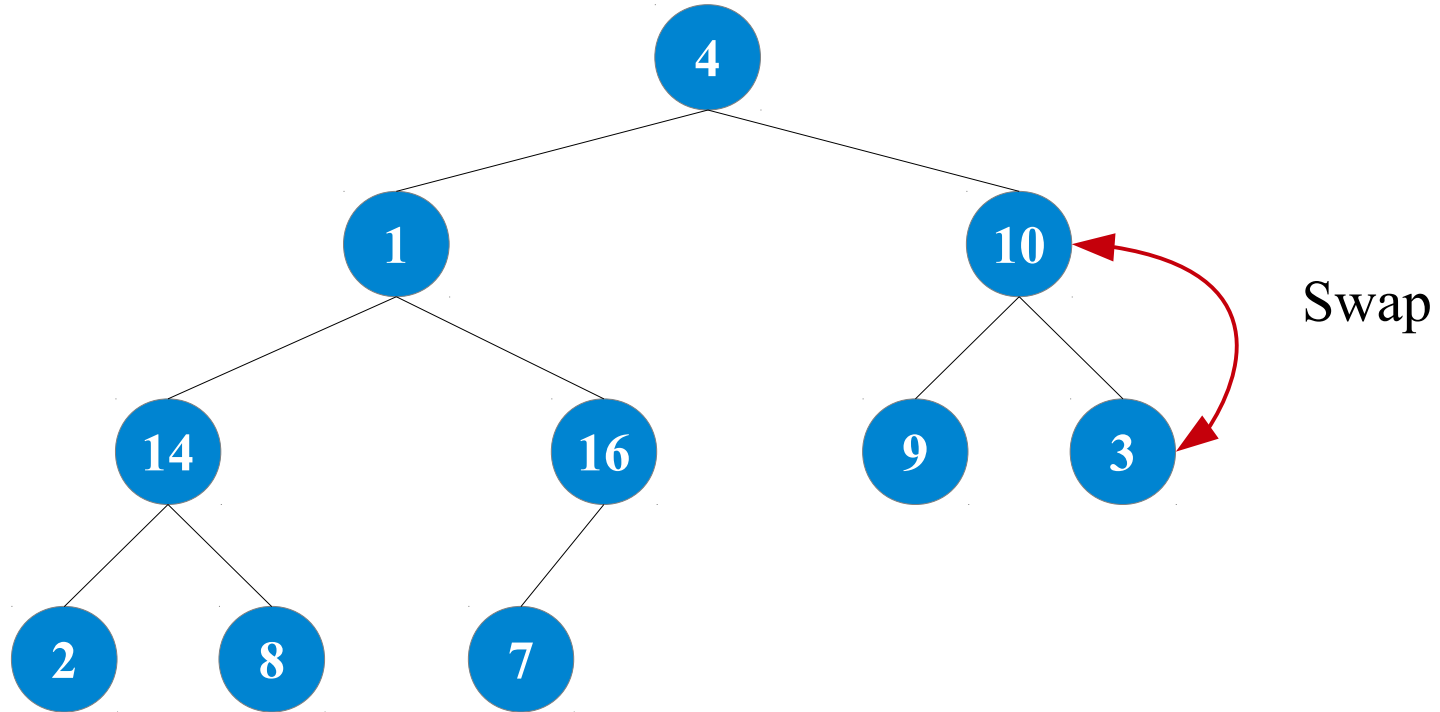
1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7

Heap Building



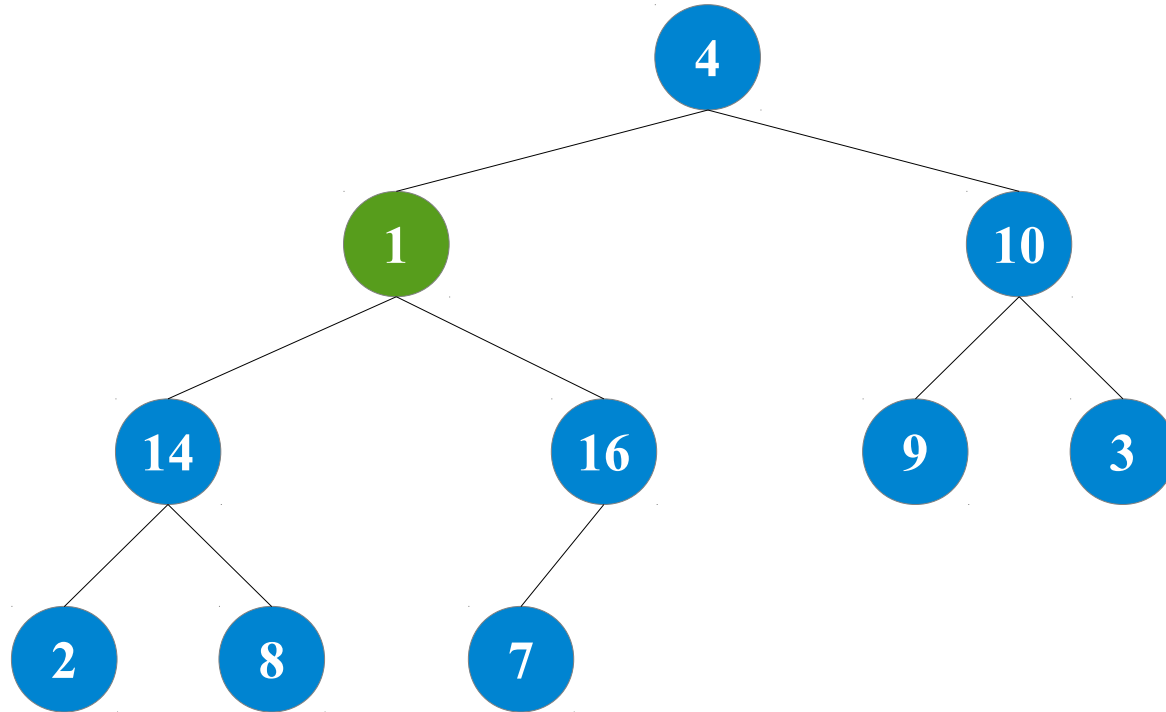
1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7

Heap Building



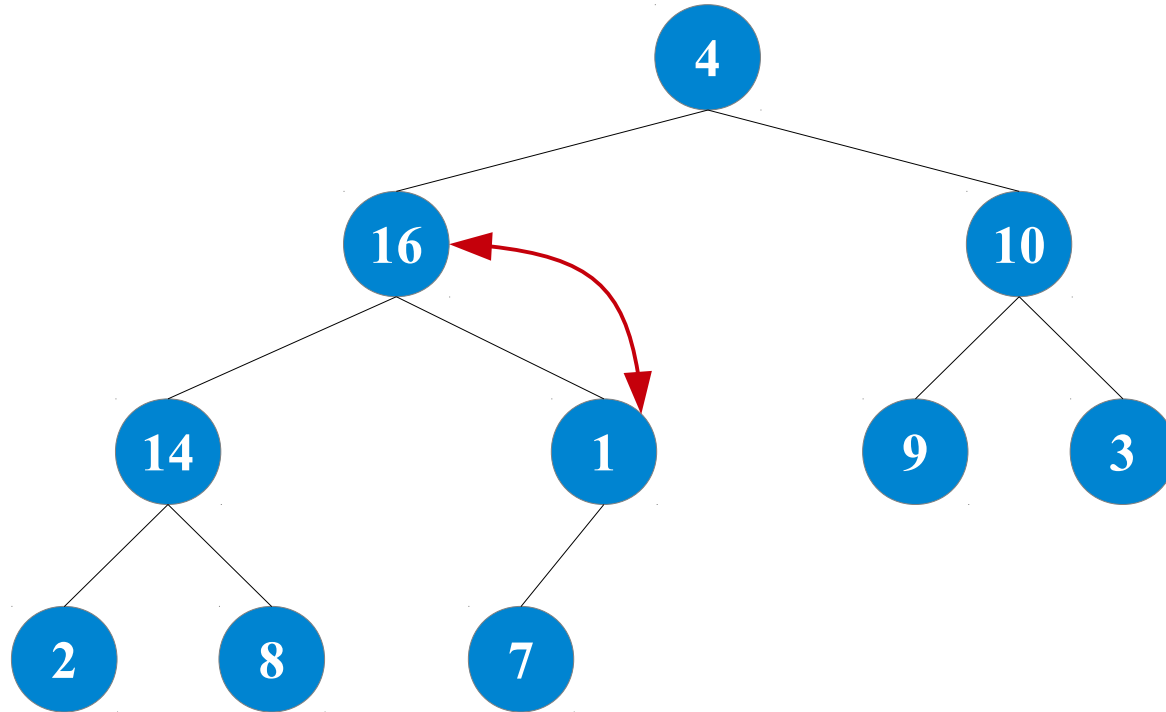
1	2	3	4	5	6	7	8	9	10
4	1	10	14	16	9	3	2	8	7

Heap Building



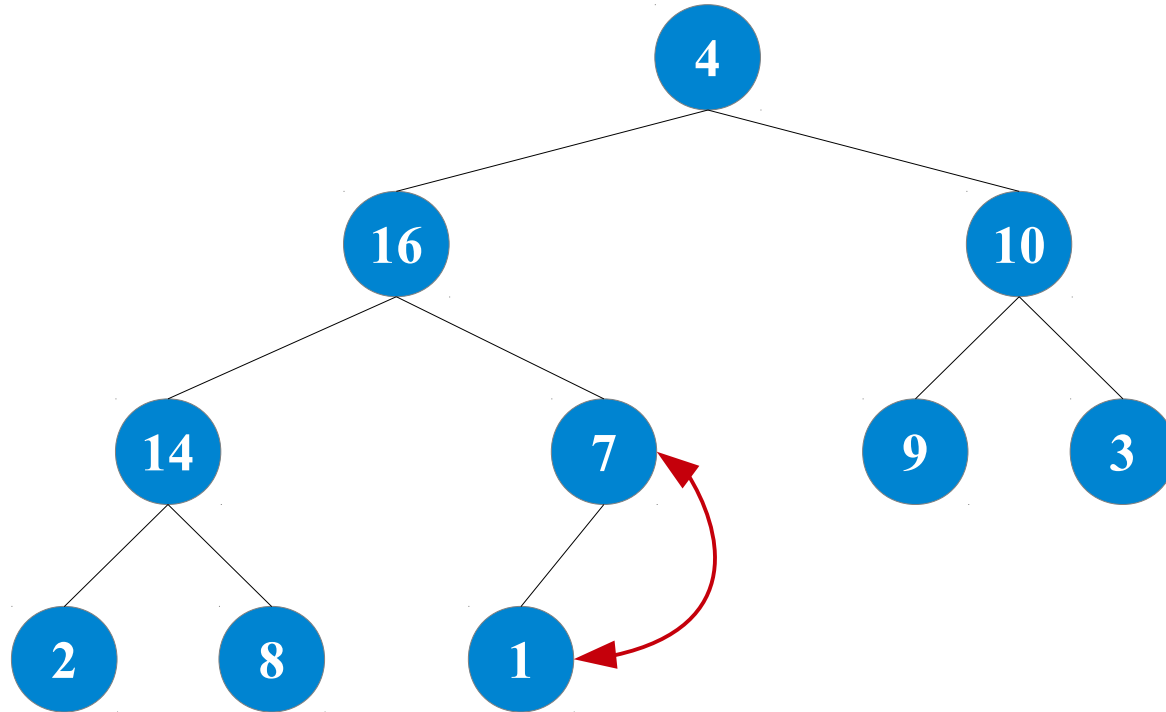
1	2	3	4	5	6	7	8	9	10
4	1	10	14	16	9	3	2	8	7

Heap Building



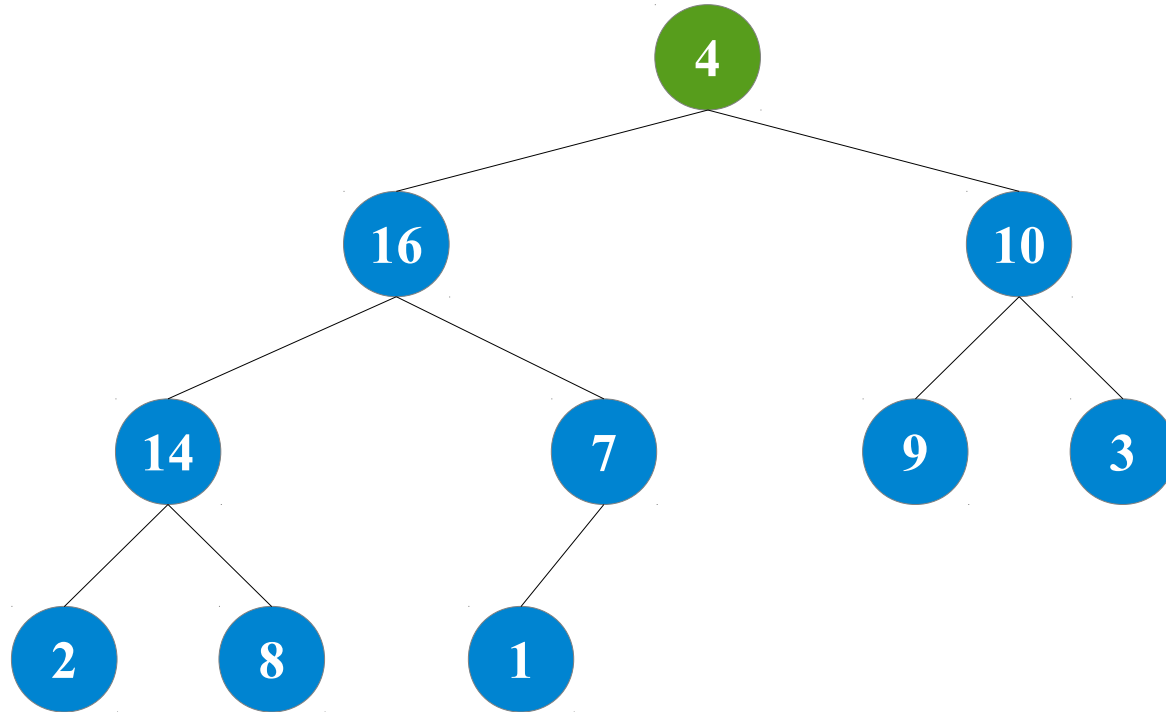
1	2	3	4	5	6	7	8	9	10
4	16	10	14	1	9	3	2	8	7

Heap Building



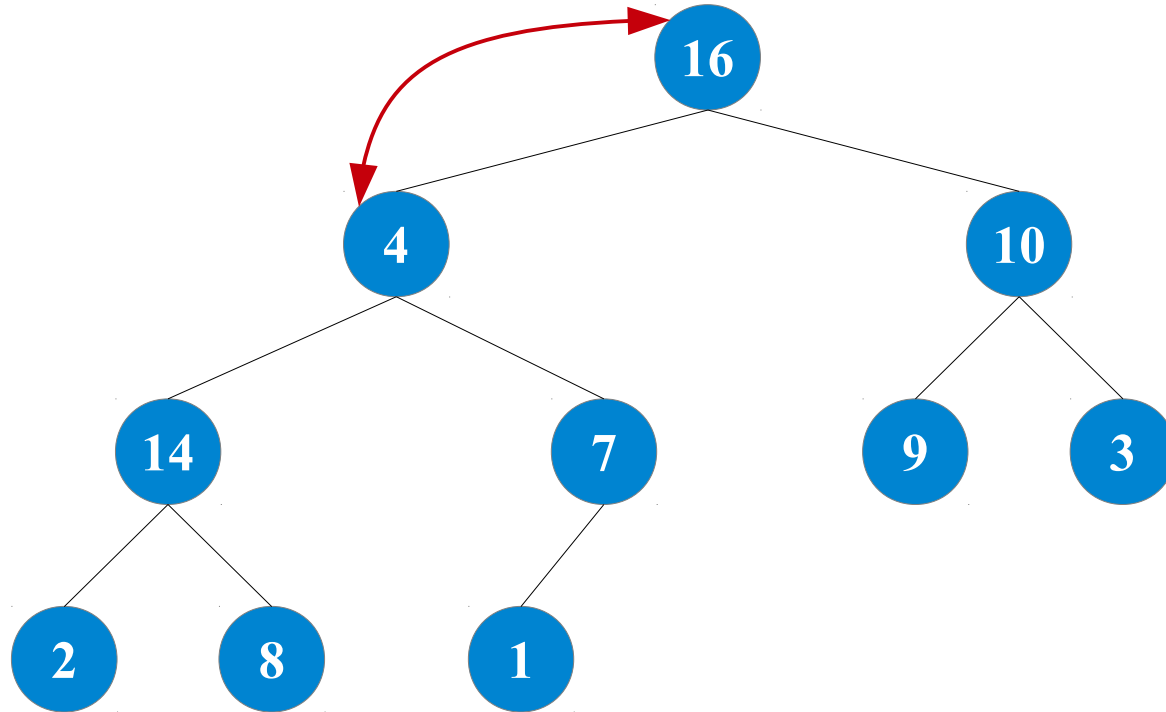
1	2	3	4	5	6	7	8	9	10
4	16	10	14	7	9	3	2	8	1

Heap Building



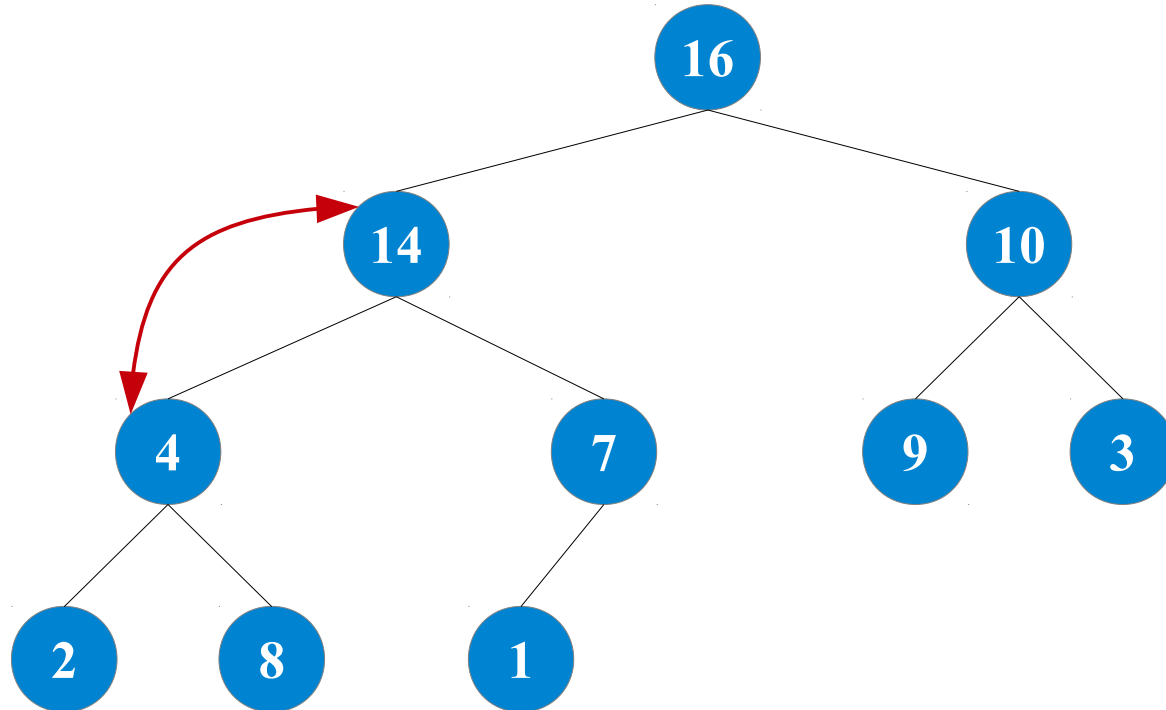
1	2	3	4	5	6	7	8	9	10
4	16	10	14	7	9	3	2	8	1

Heap Building



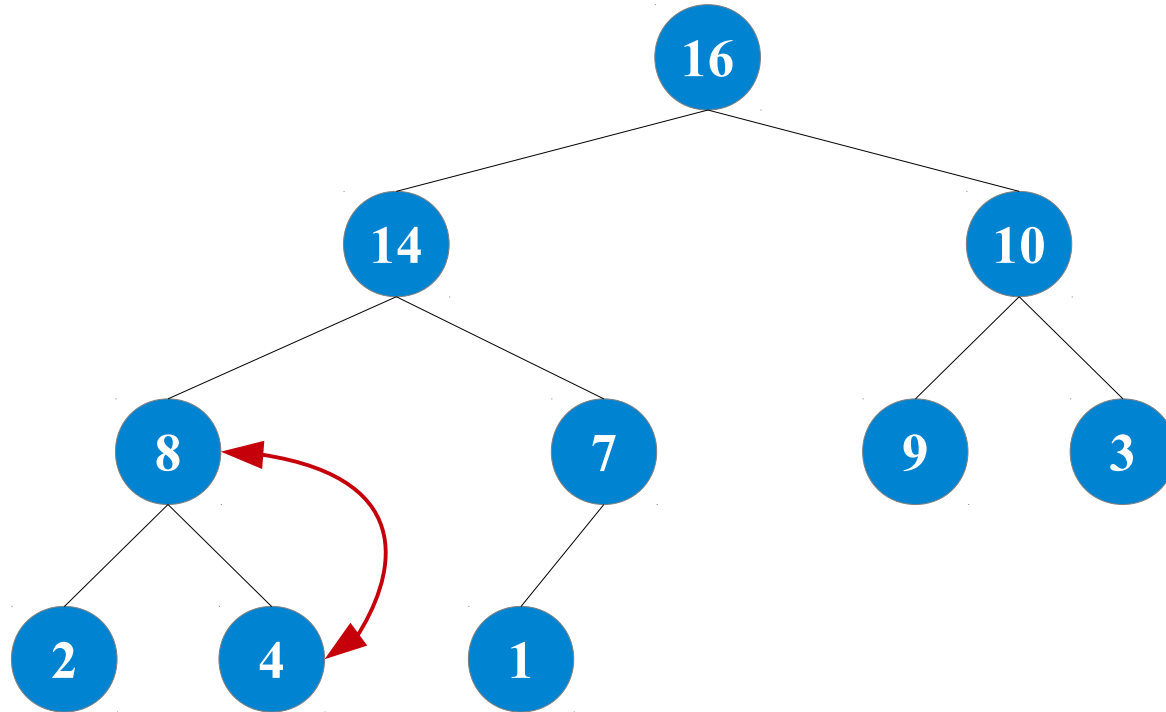
1	2	3	4	5	6	7	8	9	10
16	4	10	14	7	9	3	2	8	1

Heap Building



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

Heap Building



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

Build-Max-Heap Running Time

- Loose upper bound:
 - $n/2$ calls to Max-Heapify, each with time $\lg n$
 - $T(n) = O(n \lg n)$

- Tighter upper bound:

Number of nodes with height h

- $T(n) = O(n)$

$$T(n) = \sum_{h=0}^{\text{floor}(\lg n)} O(h) \left[\frac{n}{2^{h+1}} \right]$$

$$T(n) = O\left(n \sum_{h=0}^{\text{floor}(\lg n)} \frac{h}{2^h} \right)$$

$$\sum_{h=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \quad \text{Why?} \quad \frac{d}{dx} \left(\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \right) \rightarrow \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

$$T(n) = O(n)$$

Correctness

Loop Invariant

- A technique used to prove correctness of algorithms
- A property that is satisfied at each iteration of the loop, hence invariant.
- Show that it is satisfied at the beginning, during, and at the end of the loop

```
Build-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i =  $\lfloor A.length / 2 \rfloor$  downto 1
3    MAX-HEAPIFY(A, i)
```

Correctness

Invariant

At the start of each iteration of the loop, each node $i + 1, i + 2, \dots, n$ is the root of a Max-Heap.

Initialization

At the start, $i = \lfloor n / 2 \rfloor$. Each node $i + 1 \dots n$ is a leaf i.e. a root of a max-heap

```
Build-MAX-HEAP(A)
1  A.heap-size = A.length
2  for  $i = \lfloor A.length / 2 \rfloor$  downto 1
3    MAX-HEAPIFY(A, i)
```

Correctness

Invariant

At the start of each iteration of the loop, each node $i + 1, i + 1, \dots, n$ is the root of a Max-Heap.

Maintenance

Since the children of node i have higher indices (i.e. $2i$ and $2i + 1$), by the loop invariant they are already the roots of max-heaps. Since Max-Heapify maintains the max-heap property at node i , node i after the iteration is the root of a max-heap, maintaining the property for the next iteration.

```
Build-MAX-HEAP( $A$ )
1   $A.heap\text{-}size = A.length$ 
2  for  $i = \lfloor A.length / 2 \rfloor$  downto 1
3    MAX-HEAPIFY( $A, i$ )
```


Correctness

Invariant

At the start of each iteration of the loop, each node $i + 1, i + 1, \dots, n$ is the root of a Max-Heap.

Termination

At the end of the loop $i = 0$, and by Max-Heapify in the last iteration, node 1 is the root of a max-heap. Therefore, the invariant is satisfied and A now contains a max-heap.

```
Build-MAX-HEAP( $A$ )
1   $A.heap\text{-}size = A.length$ 
2  for  $i = \lfloor A.length / 2 \rfloor$  downto 1
3    MAX-HEAPIFY( $A, i$ )
```

Heapsort

- Given a bunch of unsorted numbers, if we build a Max-Heap out of them:
 - where is the maximum number now?
 - where should it be?

Heapsort

HEAPSORT(A)

1 **BUILD-MAX-HEAP**(A)

2 **for** $i = A.length$ **downto** 1

3 exchange $A[i]$ with $A[1]$

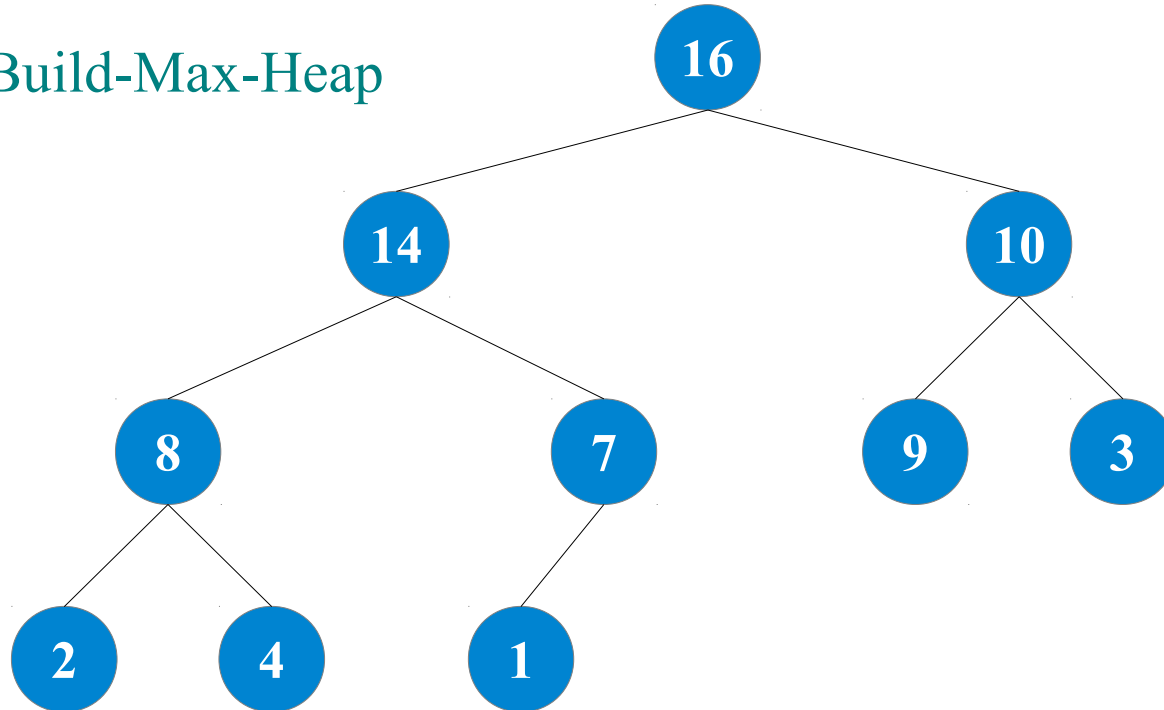
4 $A.heap-size$ --

5 **MAX-HEAPIFY**($A, 1$)

- Build a Max-Heap from the array \rightarrow put the maximum value at the front of A
- Put the max value in its right sorted place at the end and the value at the end in the root
- Heapify and repeat ...

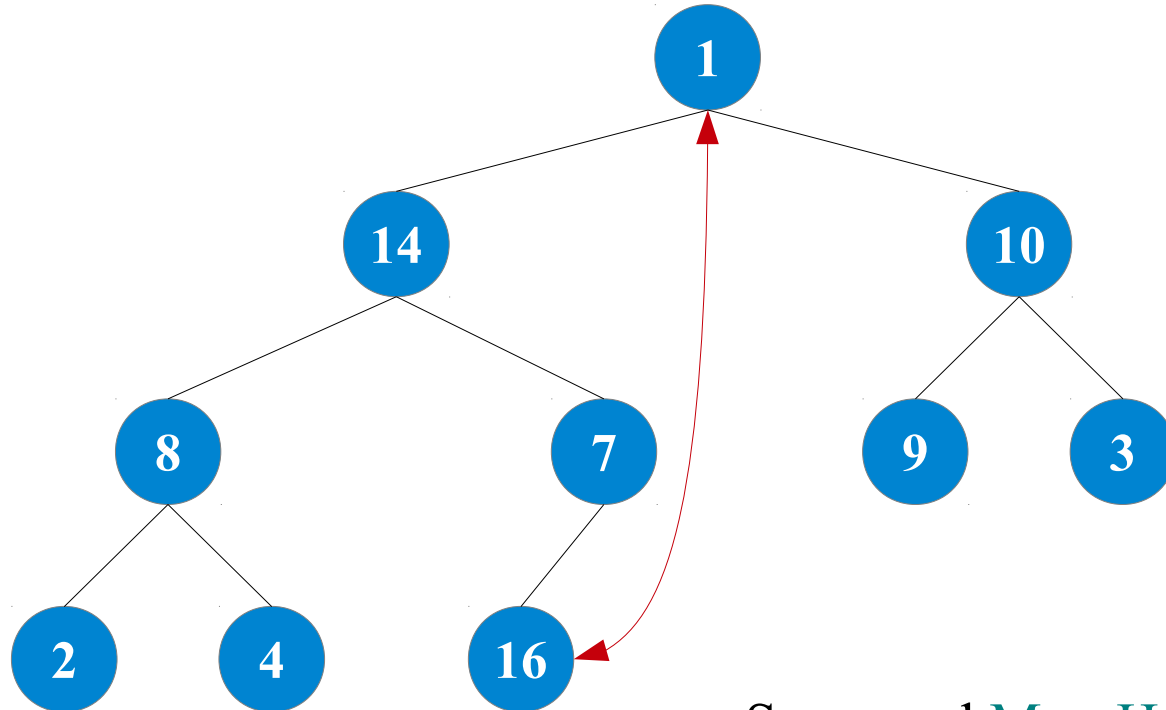
Heapsort

After calling Build-Max-Heap



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

Heapsort

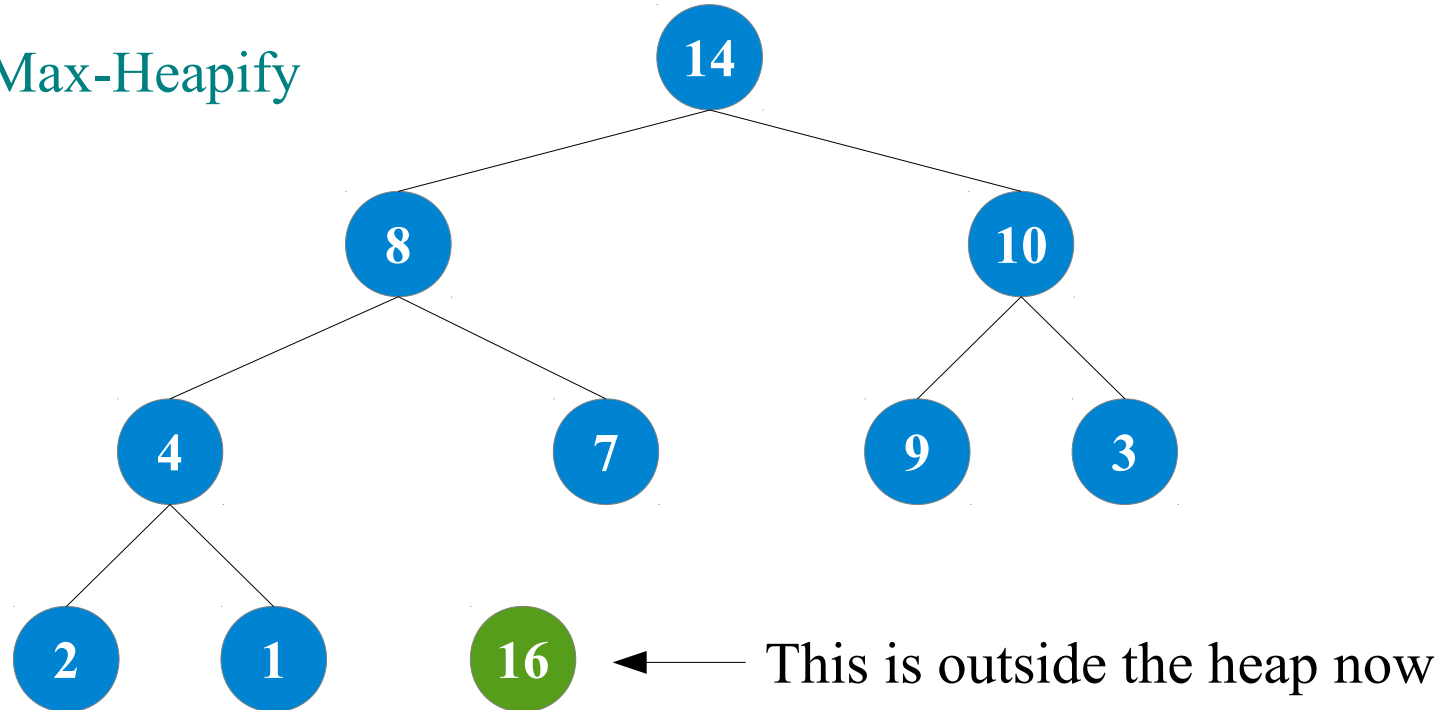


Swap and Max-Heapify

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

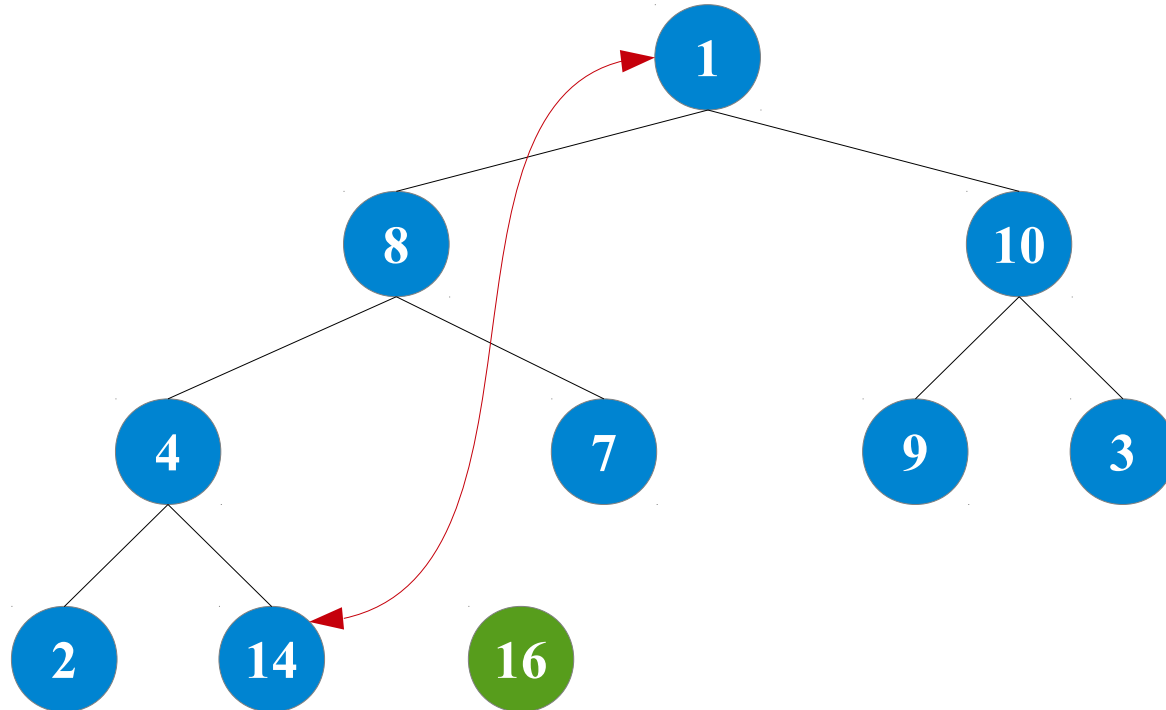
Heapsort

After calling Max-Heapify



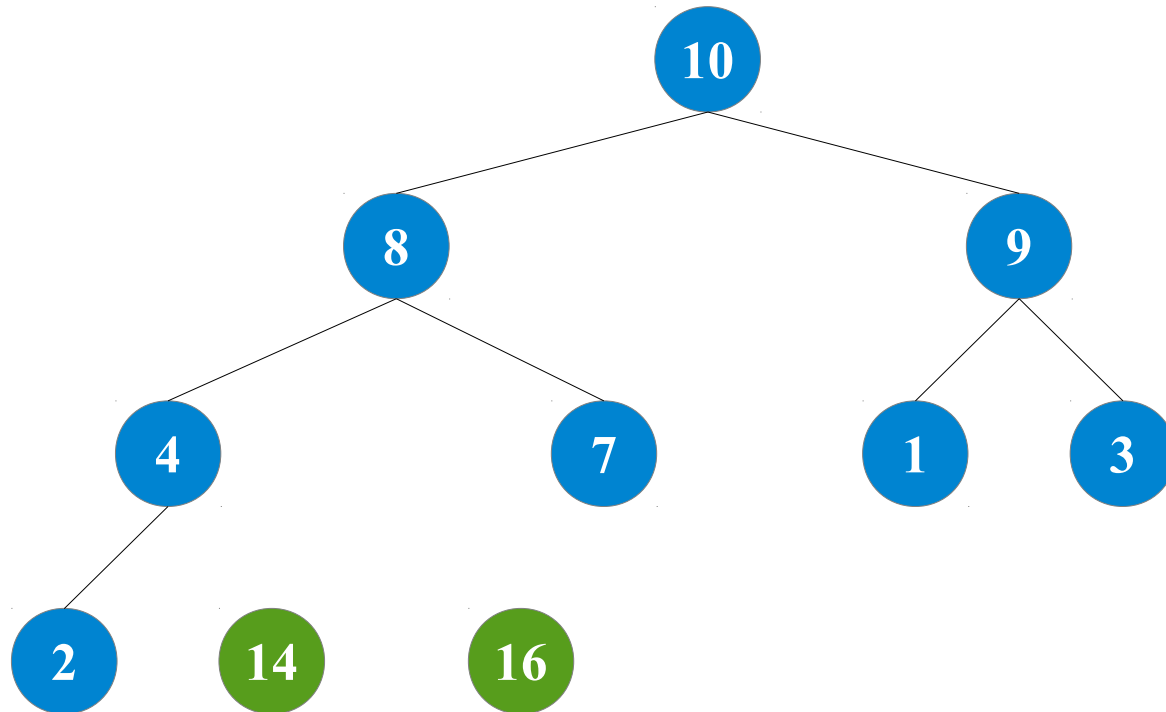
1	2	3	4	5	6	7	8	9	10
14	8	10	4	7	9	3	2	1	16

Heapsort



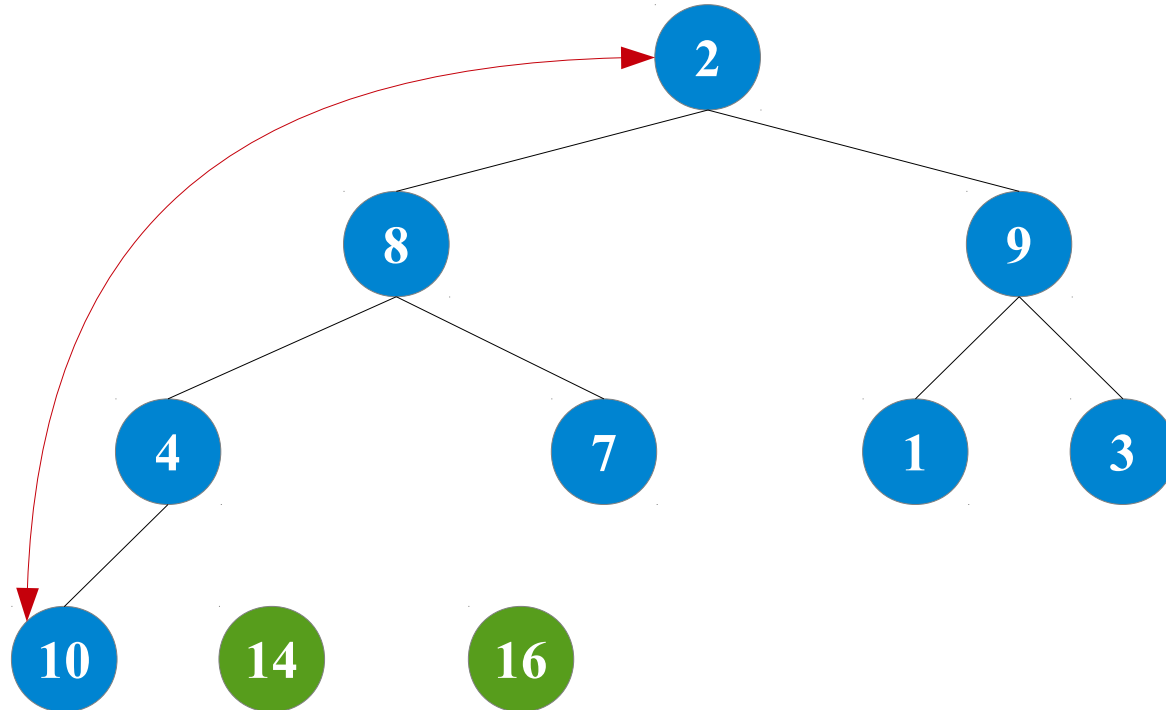
1	2	3	4	5	6	7	8	9	10
1	8	10	4	7	9	3	2	14	16

Heapsort



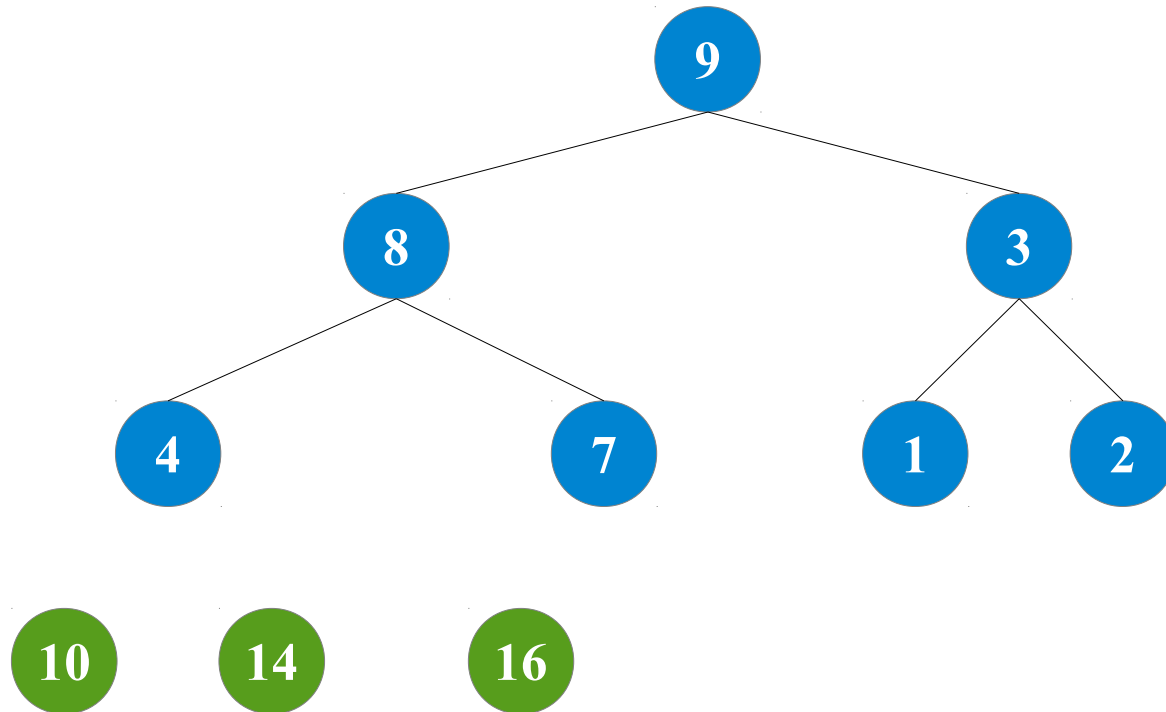
1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	3	2	14	16

Heapsort



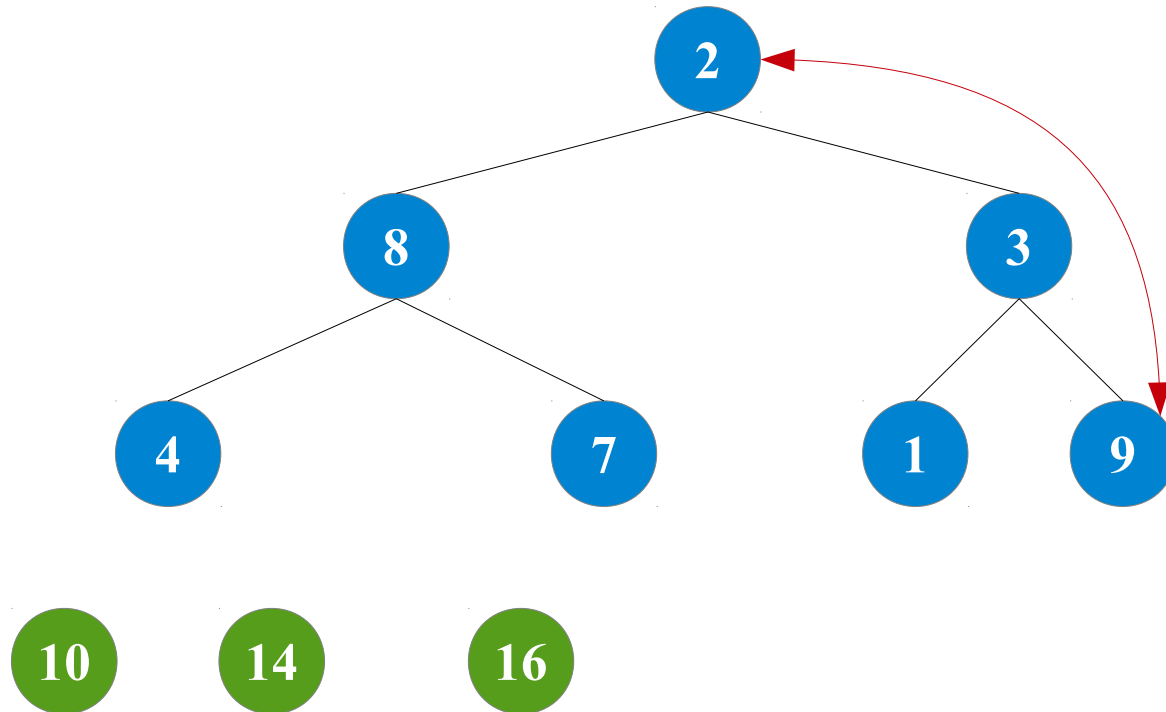
1	2	3	4	5	6	7	8	9	10
2	8	9	4	7	1	3	10	14	16

Heapsort



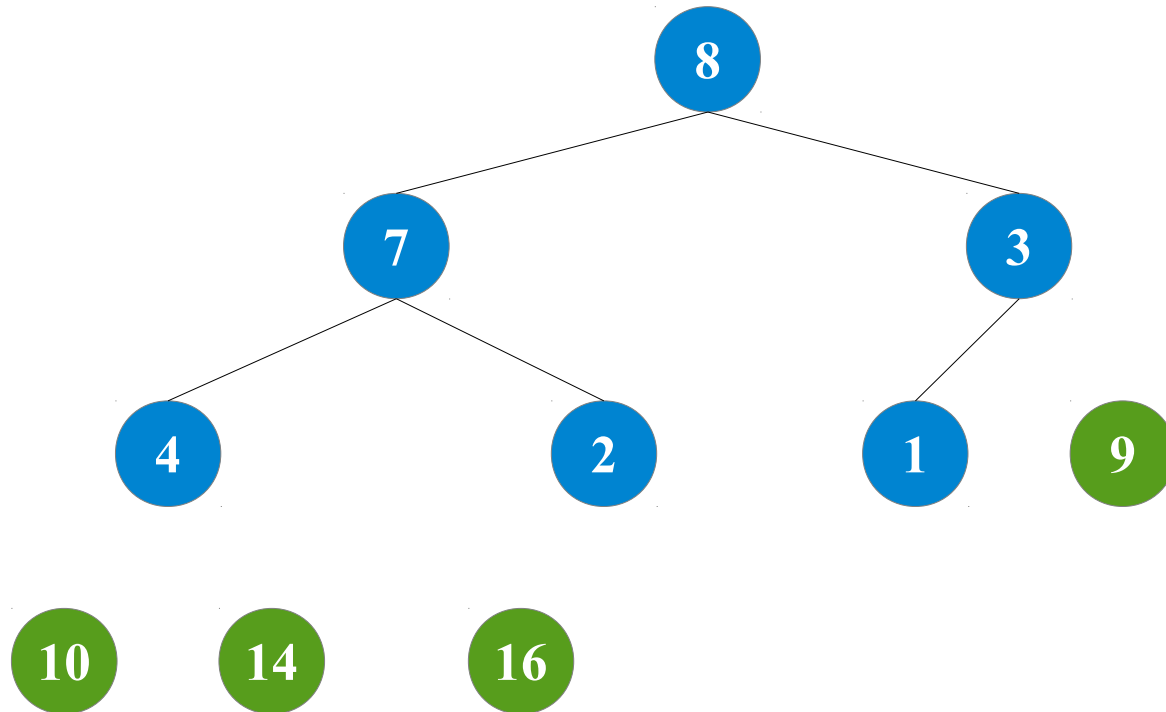
1	2	3	4	5	6	7	8	9	10
9	8	3	4	7	1	2	10	14	16

Heapsort



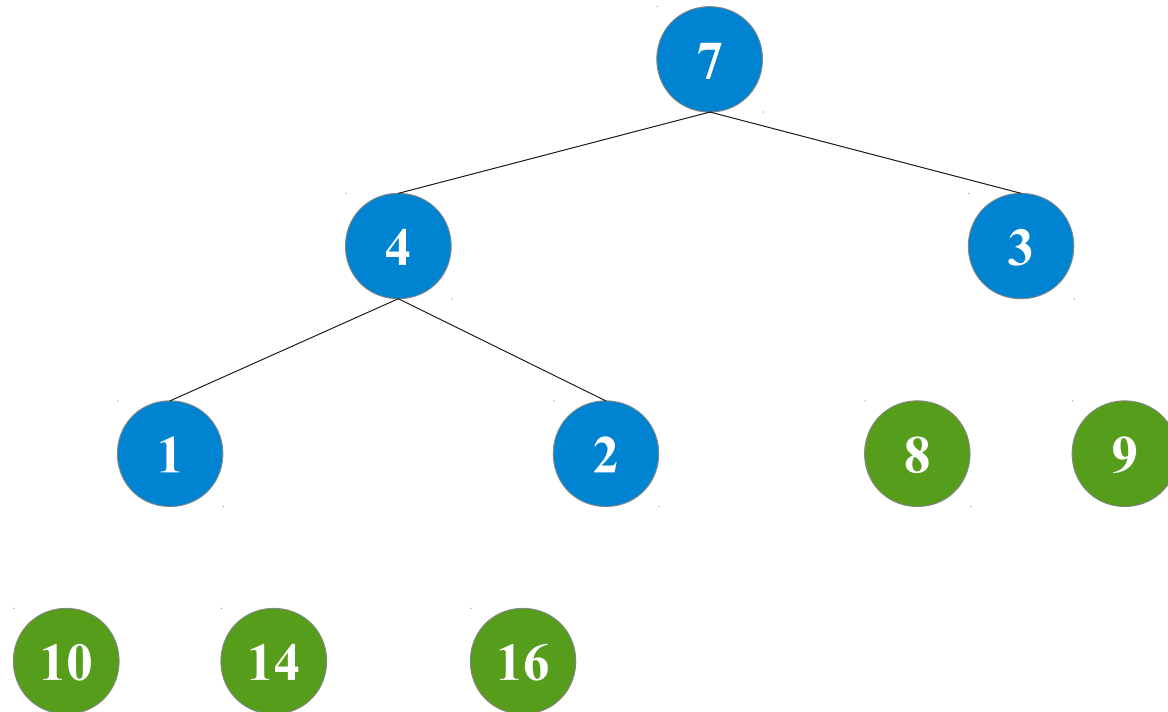
1	2	3	4	5	6	7	8	9	10
2	8	3	4	7	1	9	10	14	16

Heapsort



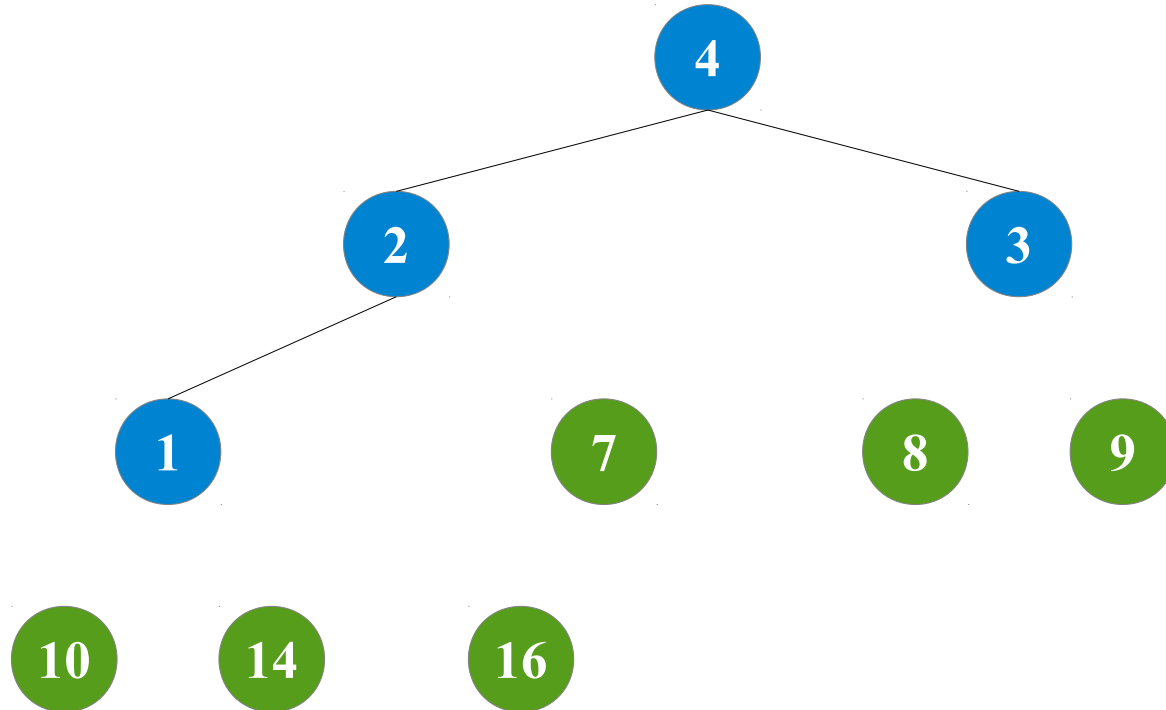
1	2	3	4	5	6	7	8	9	10
8	7	3	4	2	1	9	10	14	16

Heapsort



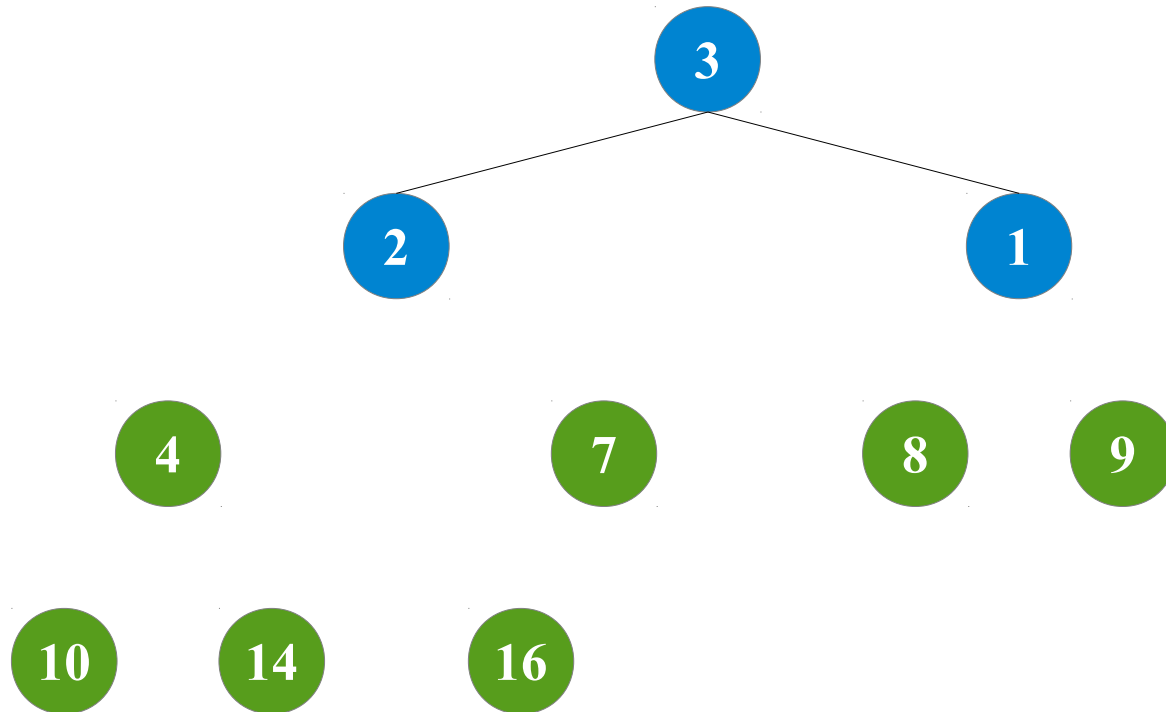
1	2	3	4	5	6	7	8	9	10
7	4	3	1	2	8	9	10	14	16

Heapsort



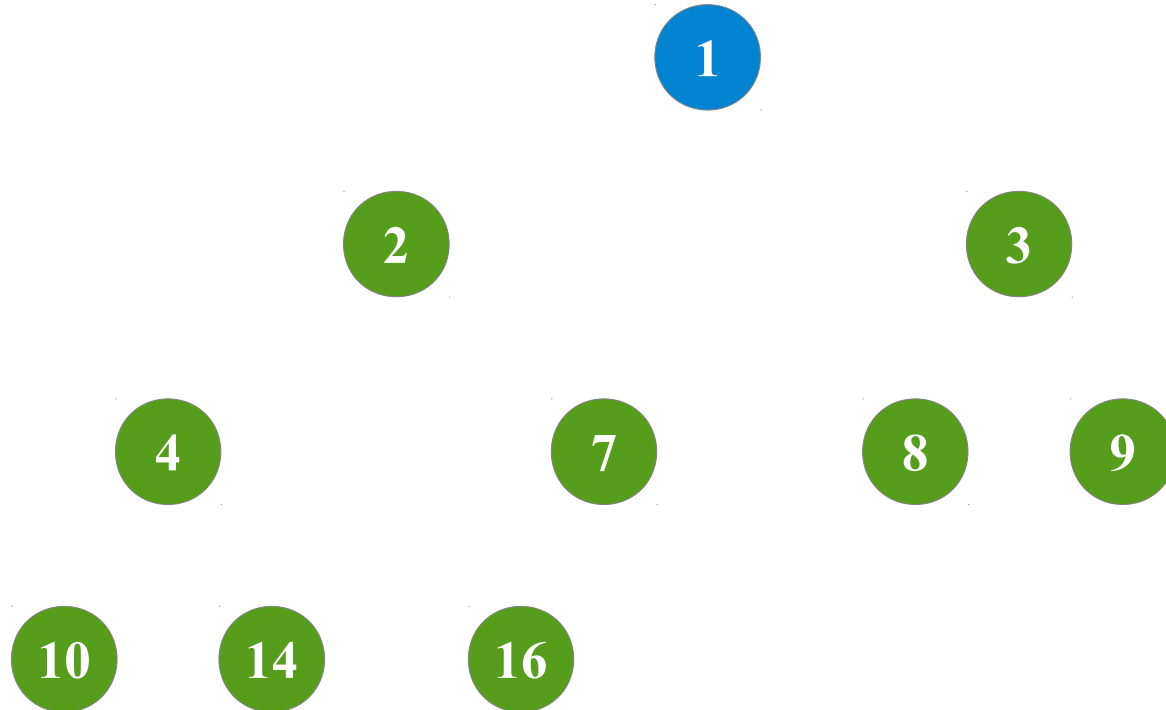
1	2	3	4	5	6	7	8	9	10
4	2	3	1	7	8	9	10	14	16

Heapsort



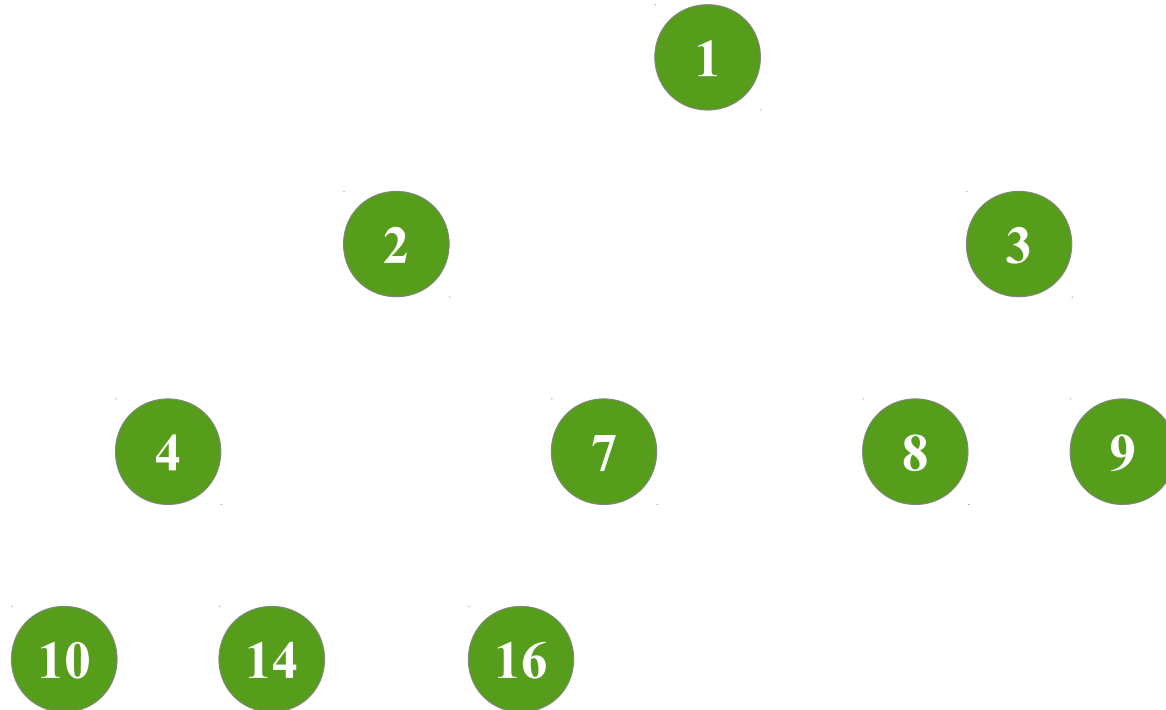
1	2	3	4	5	6	7	8	9	10
3	2	1	4	7	8	9	10	14	16

Heapsort



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

Heapsort



1 2 3 4 5 6 7 8 9 10

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

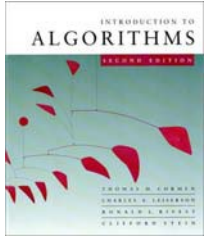
Heapsort Running time

- $T(n) = O(n)$ for Build-Max-Heap +
 n times $O(\lg n)$ for Max-Heapify
 $= O(n \lg n)$
- Space?
 - “in place” sorting $\Theta(1)$
- What was Mergesort and Insertion sort?

```
HEAPSORT(A)  
1  BUILD-MAX-HEAP(A)  
2  for i = A.length downto 1  
3    exchange A[i] with A[1]  
4    A.heap-size --  
5    MAX-HEAPIFY(A, i)
```

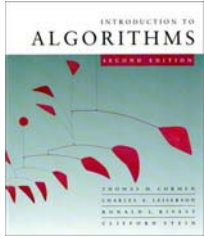
Priority Queues

- They are like **FIFO** (First-in First-out) Queues BUT depend on some **priority** value
- Element with **highest** priority goes out first, even if the last one inserted!
- Applications include:
 - Event-driven simulation
 - Job scheduler etc.
- How is it implemented?
- Read the book



Quicksort

- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts “in place” (like insertion sort, but not like merge sort).
- Very practical (with tuning).



Divide and conquer

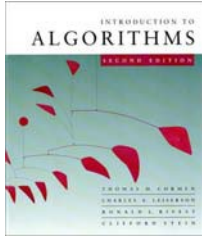
Quicksort an n -element array:

- 1. *Divide:*** Partition the array into two subarrays around a **pivot** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.



- 2. *Conquer:*** Recursively sort the two subarrays.
- 3. *Combine:*** Trivial.

Key: *Linear-time partitioning subroutine.*



Pseudocode for quicksort

QUICKSORT(A, p, r)

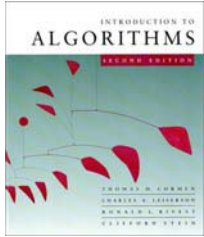
if $p < r$

then $q \leftarrow$ PARTITION(A, p, r)

QUICKSORT($A, p, q-1$)

QUICKSORT($A, q+1, r$)

Initial call: QUICKSORT($A, 1, n$)

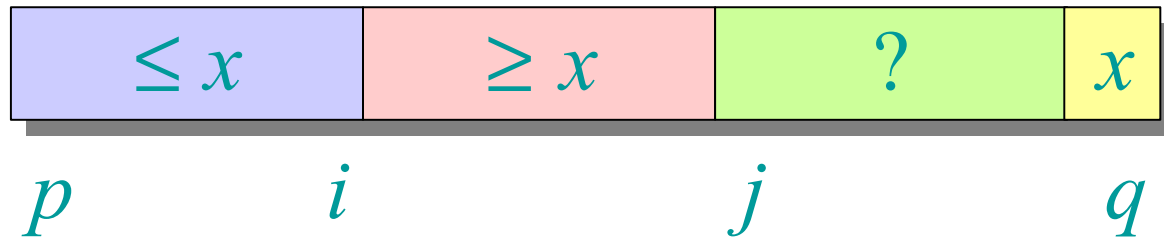


Partitioning subroutine

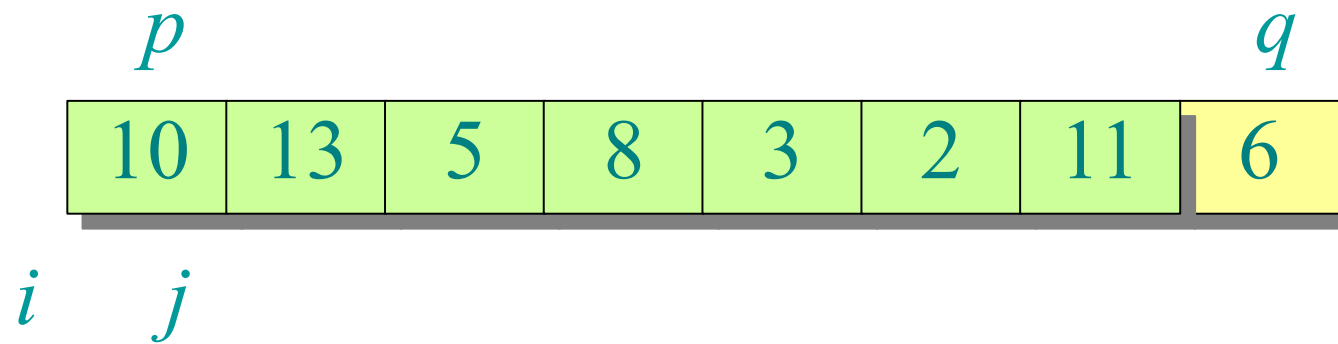
```
PARTITION( $A, p, q$ )  $\triangleright A[p \dots q]$   
   $x \leftarrow A[q]$   $\triangleright$  pivot =  $A[q]$   
   $i \leftarrow p - 1$   
  for  $j \leftarrow p$  to  $q - 1$   
    do if  $A[j] \leq x$   
      then  $i \leftarrow i + 1$   
           exchange  $A[i] \leftrightarrow A[j]$   
  exchange  $A[q] \leftrightarrow A[i + 1]$   
  return  $i$ 
```

Running time
= $O(n)$ for n
elements.

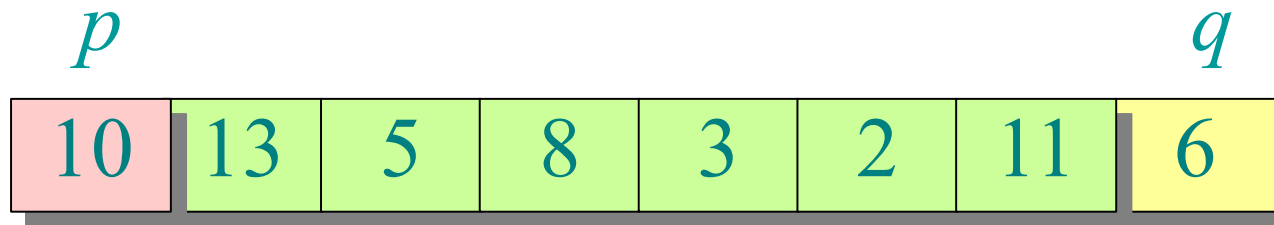
Invariant:



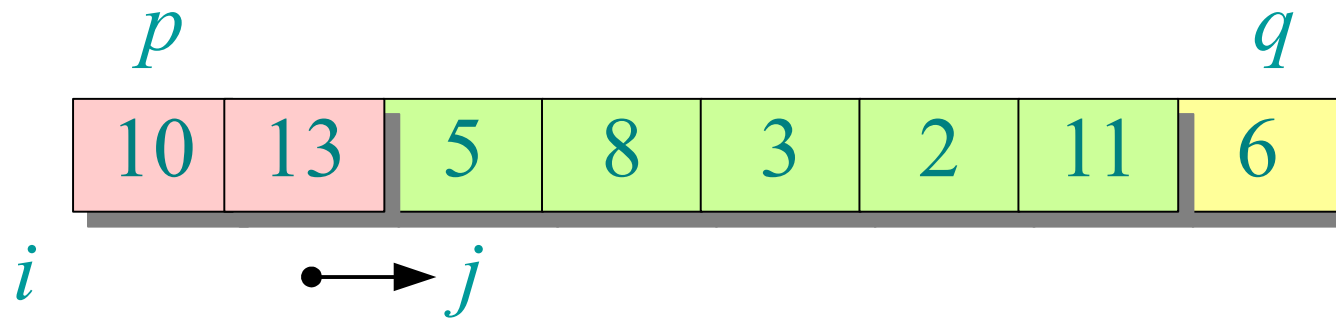
Example of Partitioning



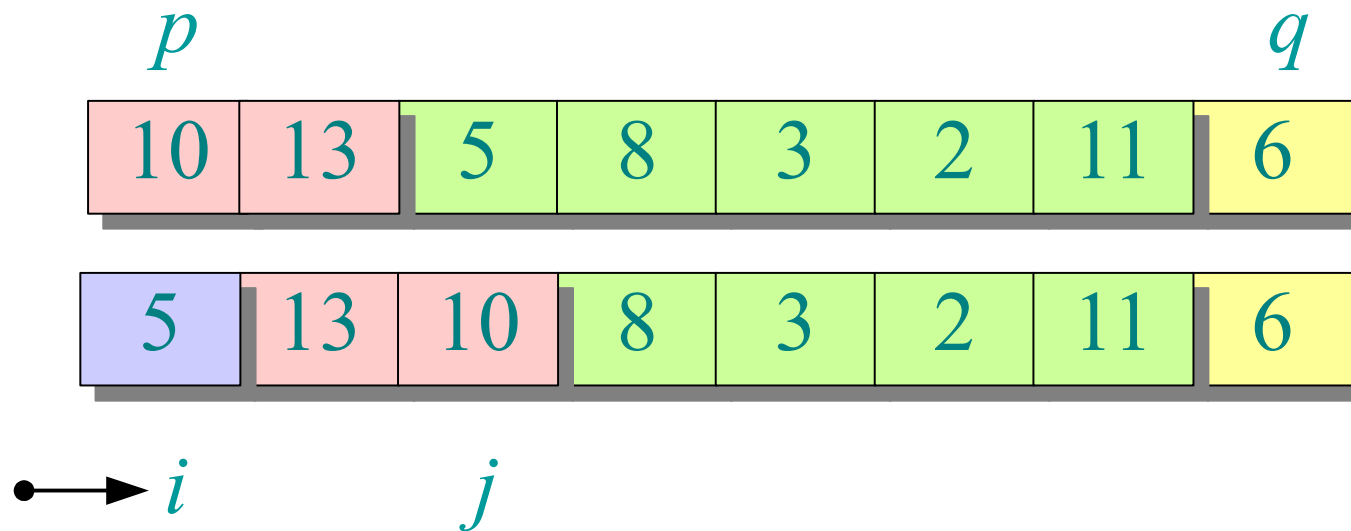
Example of Partitioning



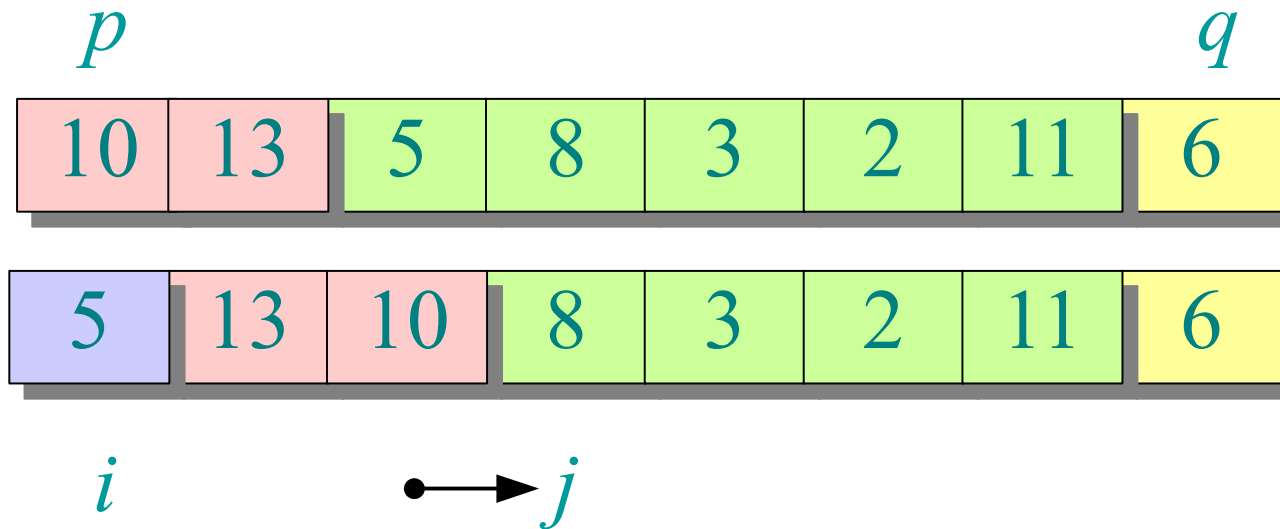
Example of Partitioning



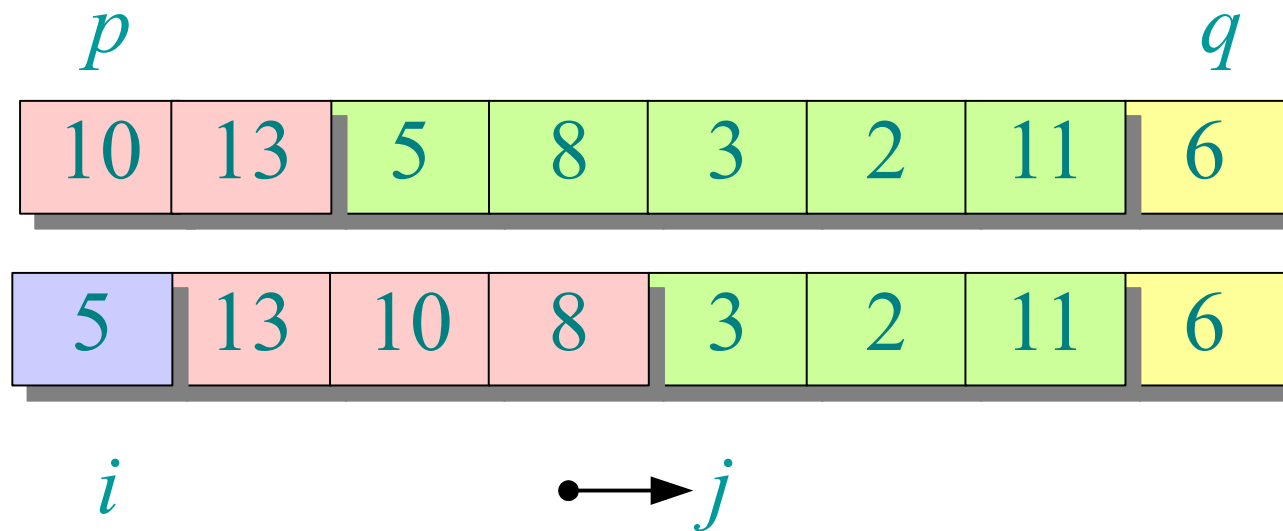
Example of Partitioning



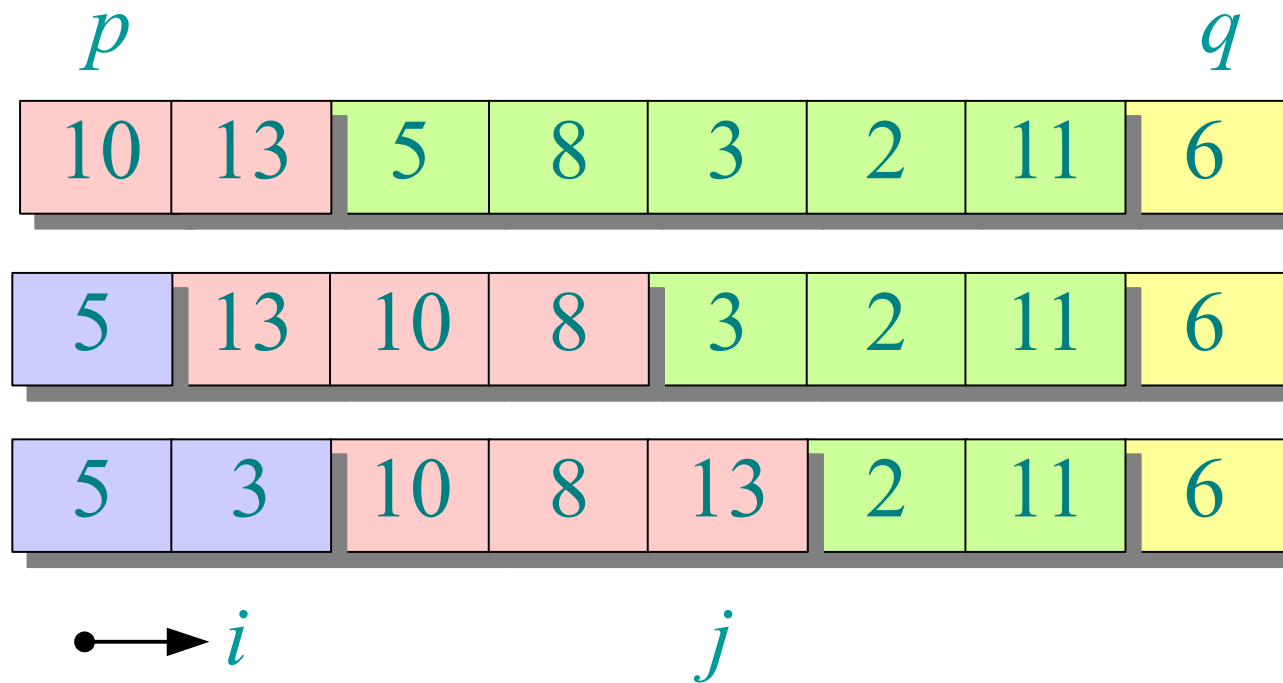
Example of Partitioning



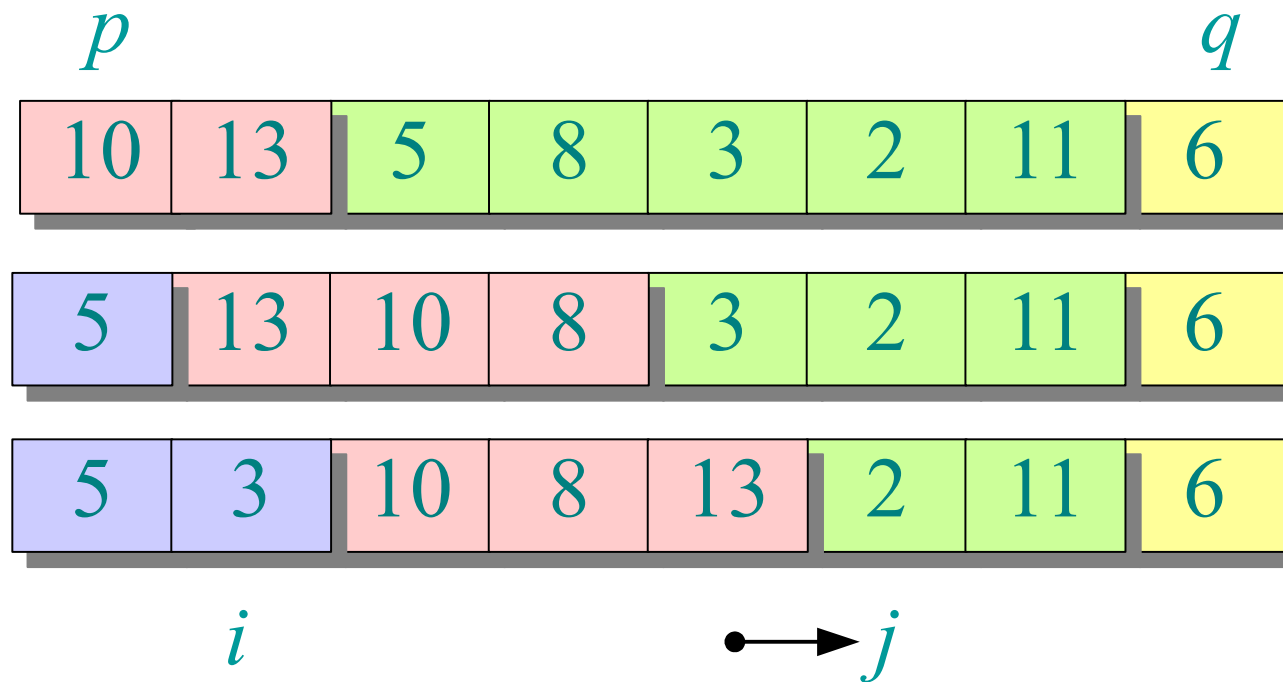
Example of Partitioning



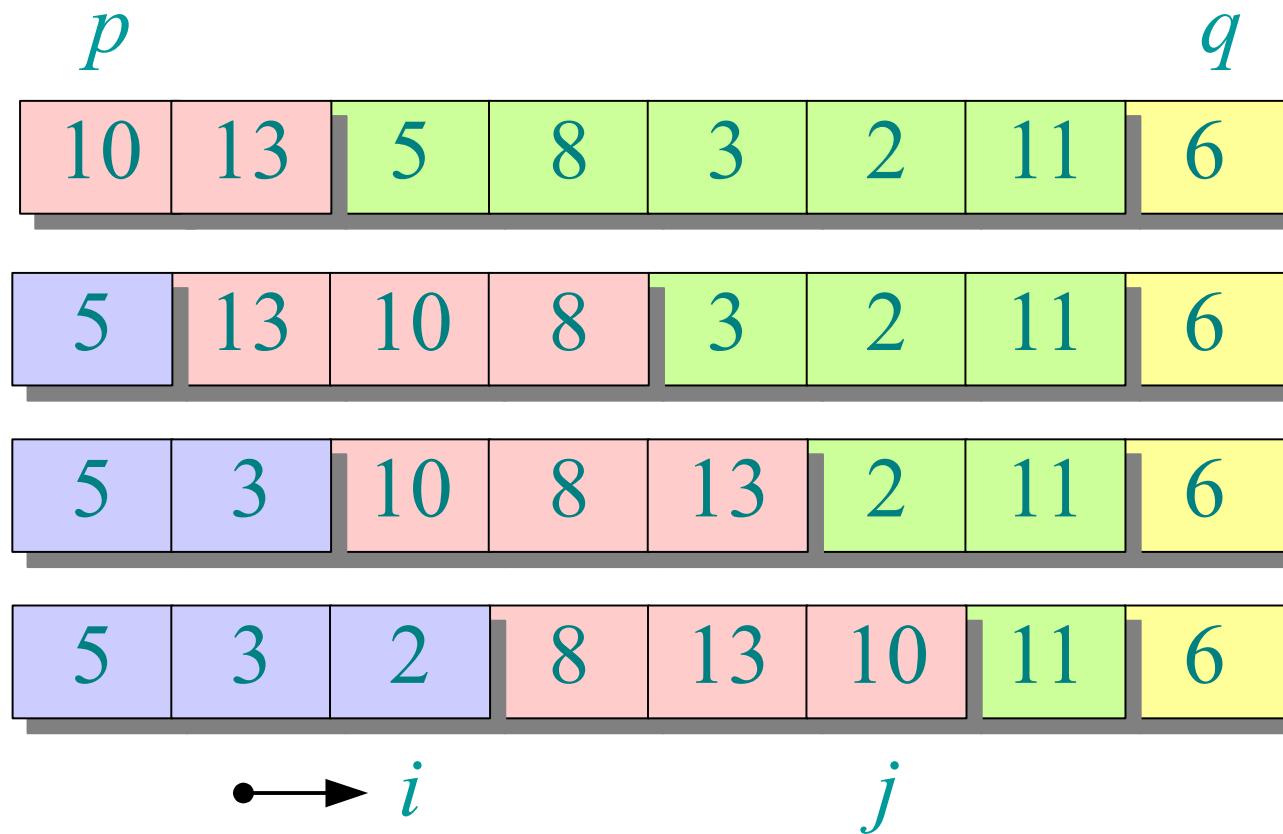
Example of Partitioning



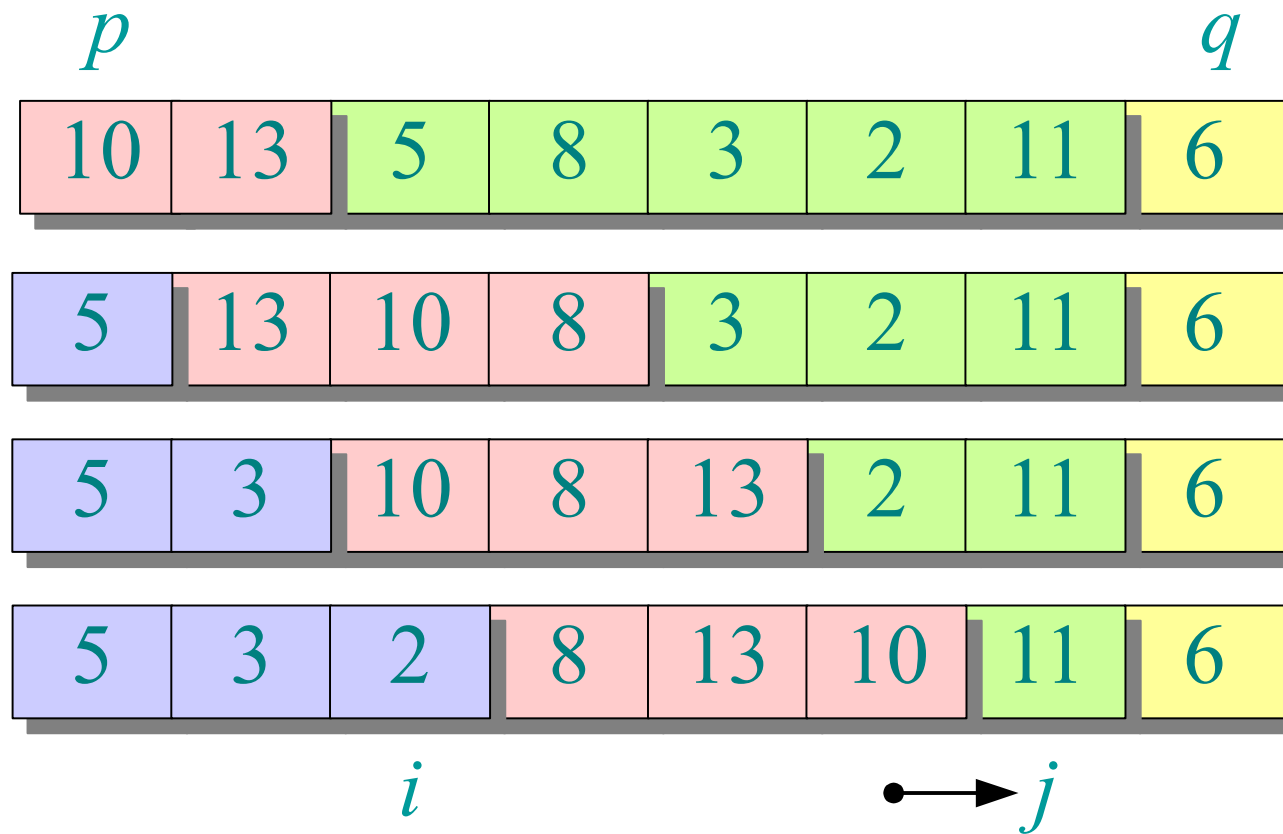
Example of Partitioning



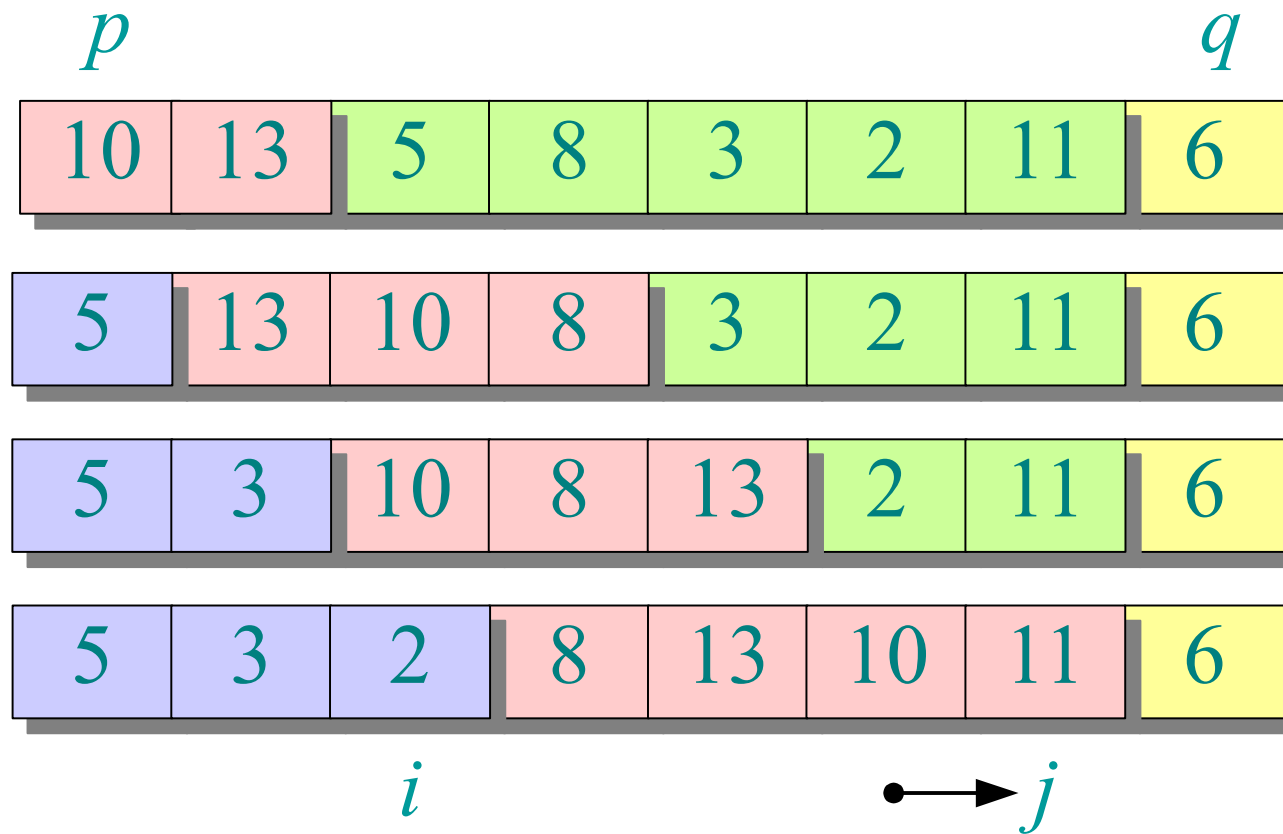
Example of Partitioning



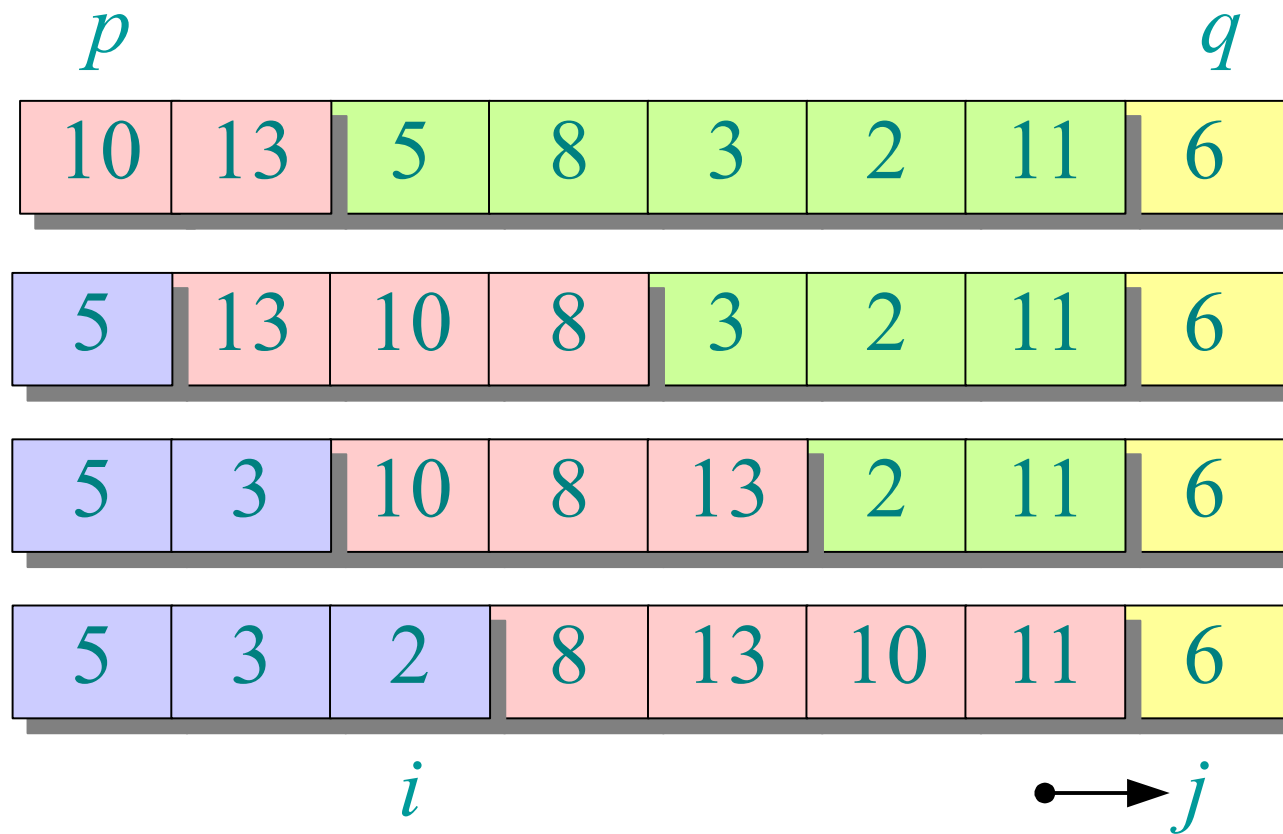
Example of Partitioning



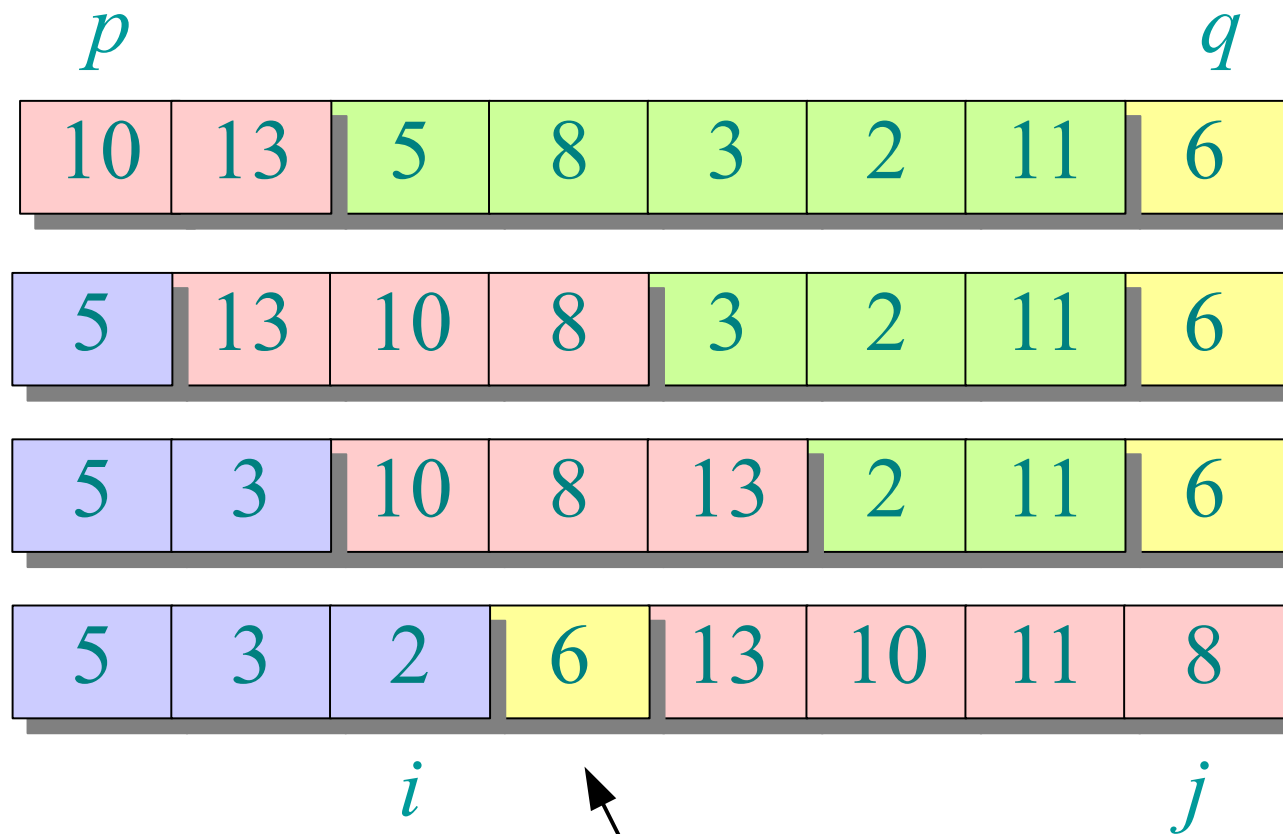
Example of Partitioning



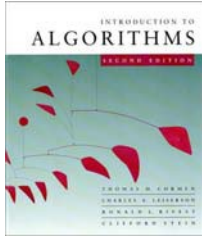
Example of Partitioning



Example of Partitioning



Pivot in place



Pseudocode for quicksort

QUICKSORT(A, p, r)

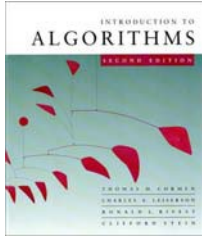
if $p < r$

then $q \leftarrow$ PARTITION(A, p, r)

QUICKSORT($A, p, q-1$)

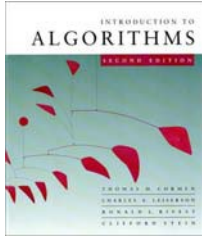
QUICKSORT($A, q+1, r$)

Initial call: QUICKSORT($A, 1, n$)



Analysis of quicksort

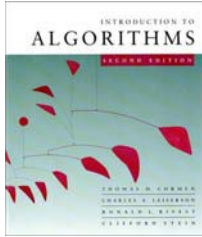
- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n)$ = worst-case running time on an array of n elements.



Worst-case of quicksort

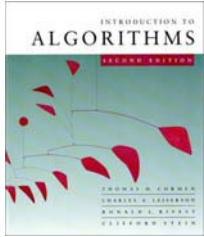
- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$\begin{aligned}T(n) &= T(0) + T(n-1) + \Theta(n) \\&= \Theta(1) + T(n-1) + \Theta(n) \\&= T(n-1) + \Theta(n) \\&= \Theta(n^2) \quad (\textit{arithmetic series})\end{aligned}$$



Worst-case recursion tree

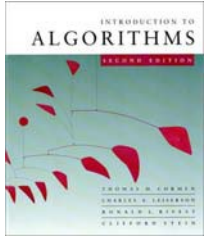
$$T(n) = T(0) + T(n-1) + cn$$



Worst-case recursion tree

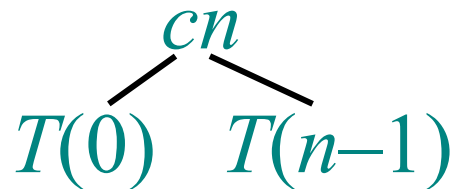
$$T(n) = T(0) + T(n-1) + cn$$

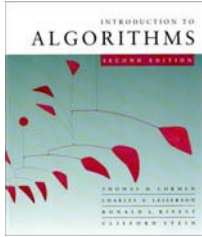
$T(n)$



Worst-case recursion tree

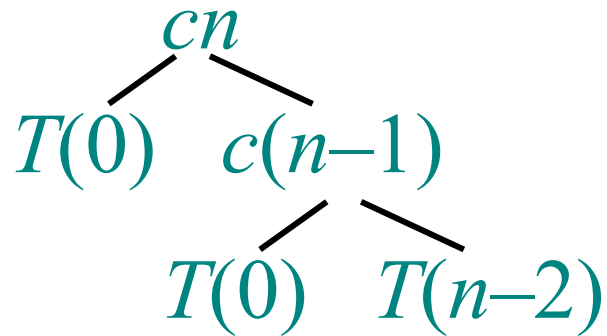
$$T(n) = T(0) + T(n-1) + cn$$

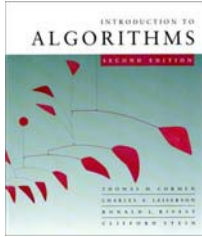




Worst-case recursion tree

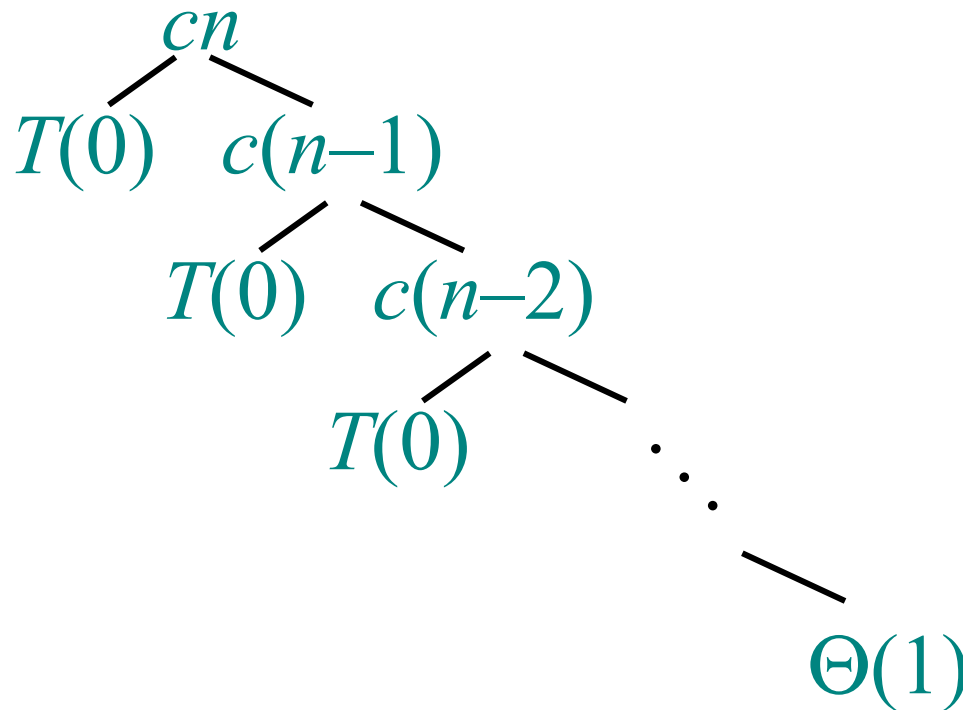
$$T(n) = T(0) + T(n-1) + cn$$

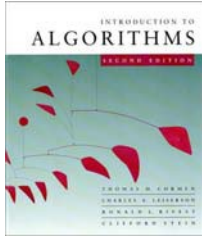




Worst-case recursion tree

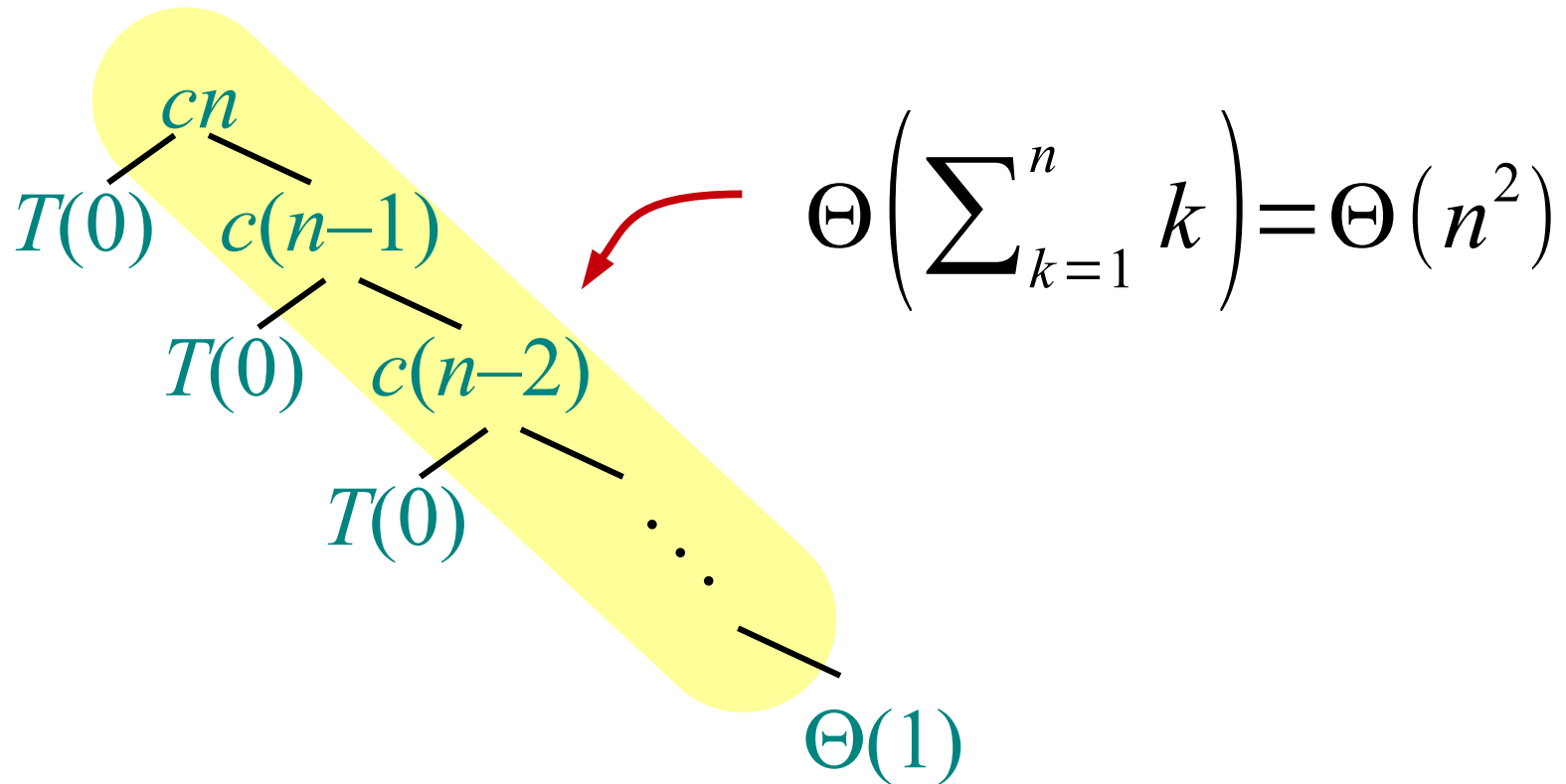
$$T(n) = T(0) + T(n-1) + cn$$

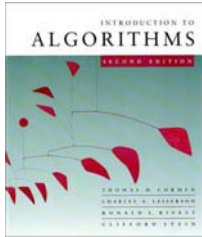




Worst-case recursion tree

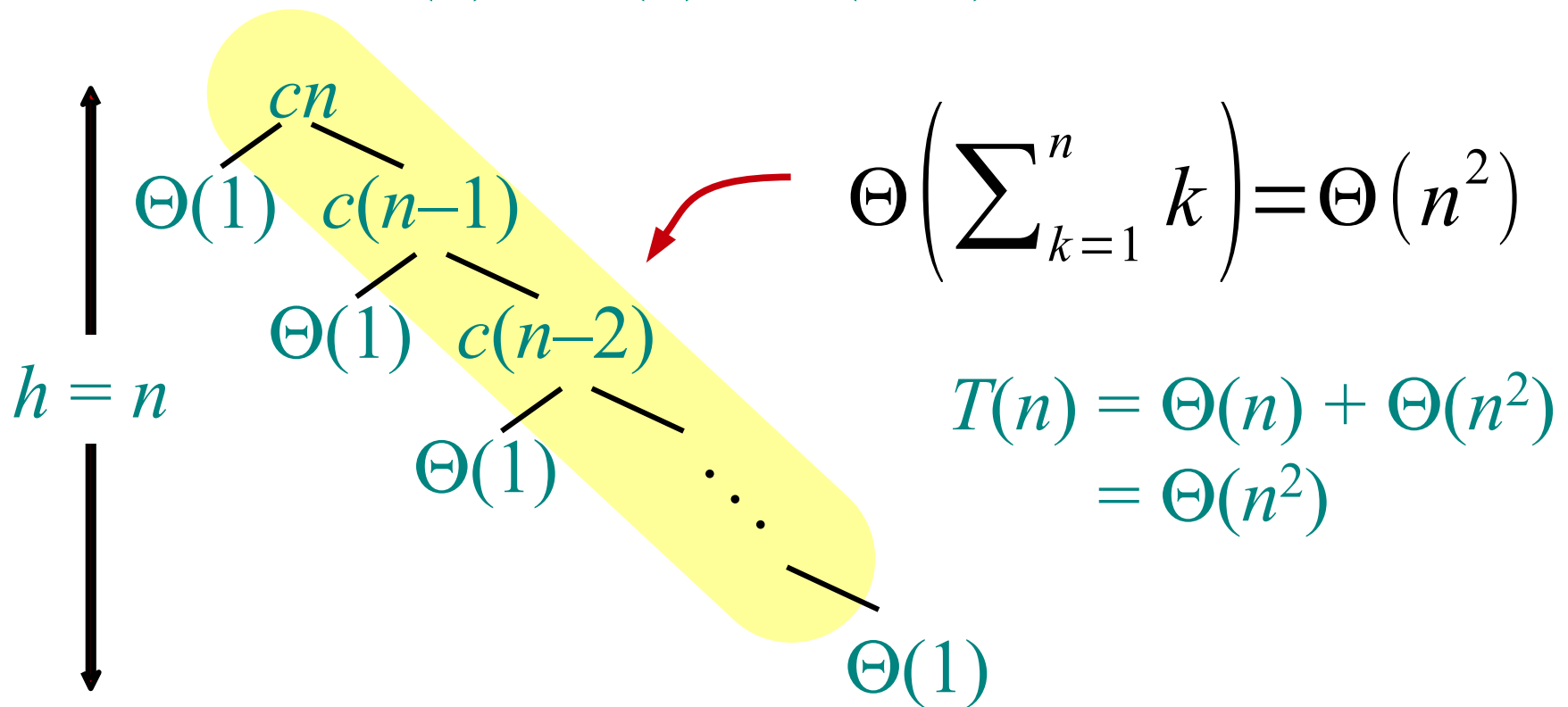
$$T(n) = T(0) + T(n-1) + cn$$

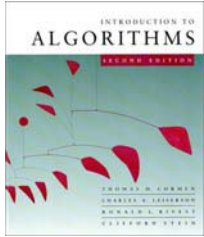




Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$





Best-case analysis

(For intuition only!)

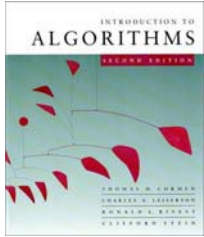
If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \quad (\text{same as merge sort}) \end{aligned}$$

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

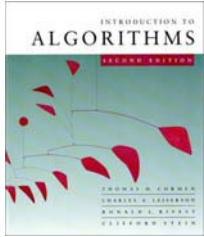
$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$

What is the solution to this recurrence?

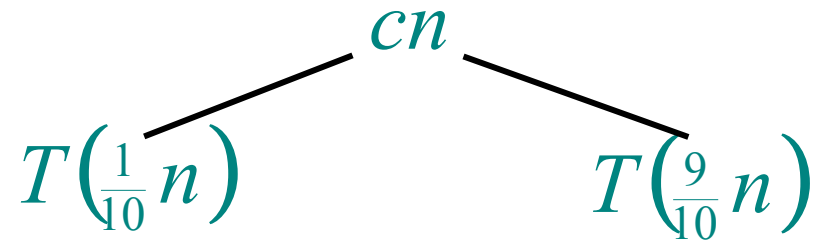


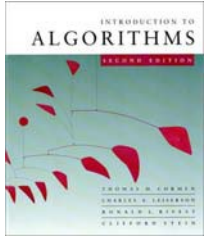
Analysis of “almost-best” case

$$T(n)$$

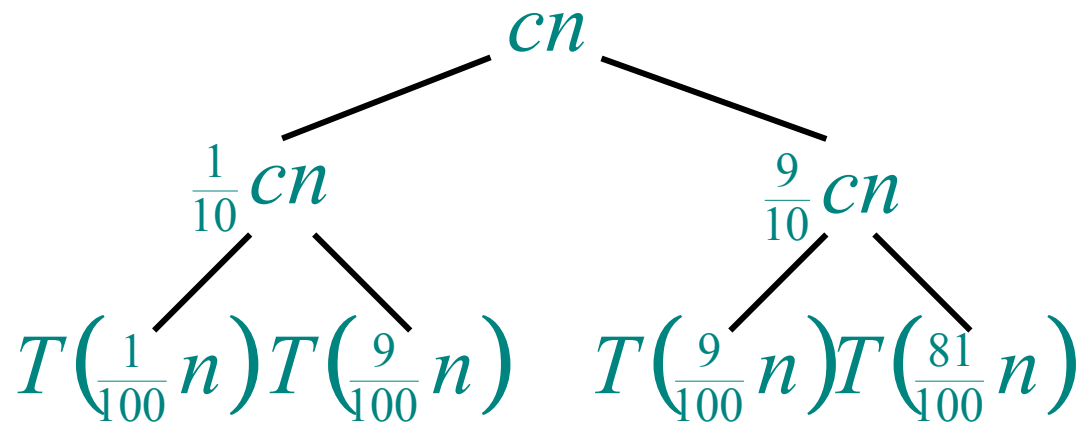


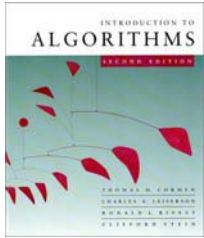
Analysis of “almost-best” case



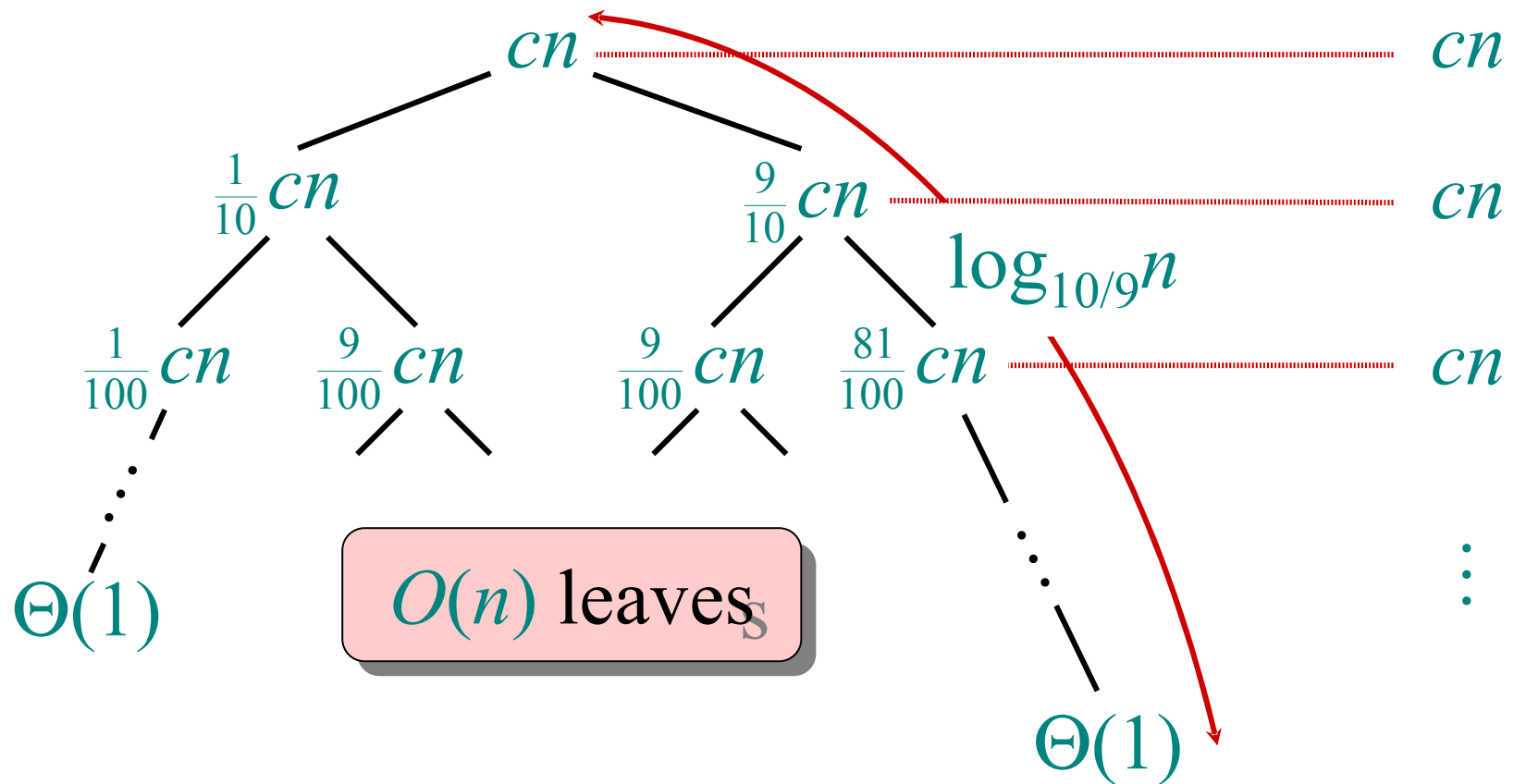


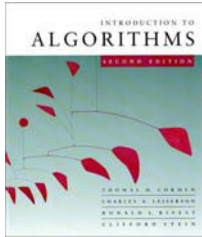
Analysis of “almost-best” case



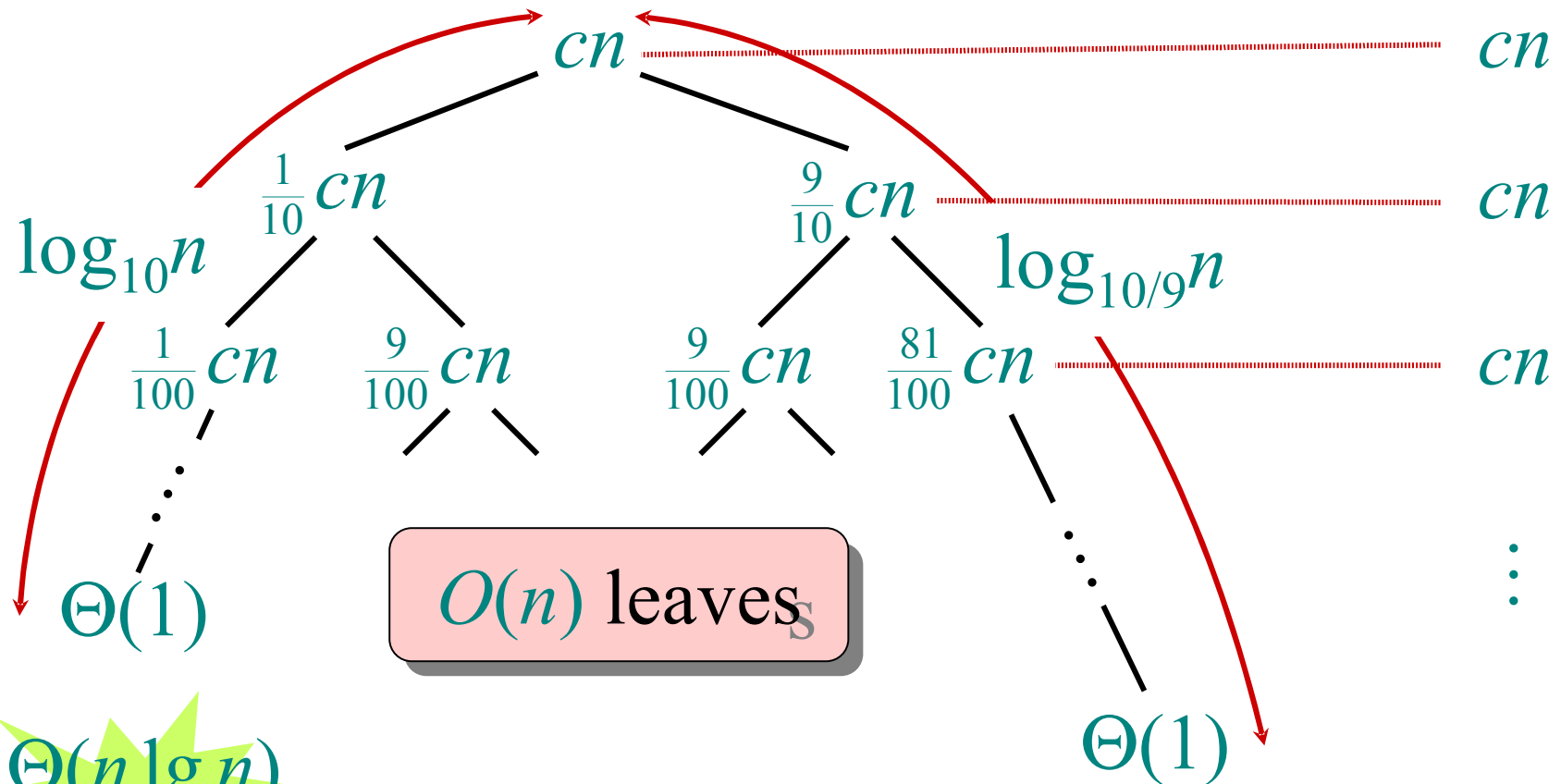


Analysis of “almost-best” case



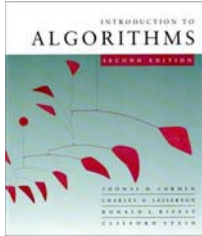


Analysis of “almost-best” case



$\Theta(n \lg n)$
Lucky!

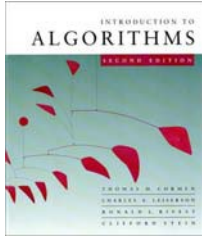
$$cn \log_{10} n \leq T(n) \leq cn \log_{10/9} n + O(n)$$



More intuition

Suppose we alternate lucky, unlucky,
lucky, unlucky, lucky,

$$\begin{aligned}L(n) &= 2U(n/2) + \Theta(n) && \textit{lucky} \\U(n) &= L(n - 1) + \Theta(n) && \textit{unlucky}\end{aligned}$$



More intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky,

$$L(n) = 2U(n/2) + \Theta(n) \quad \textit{lucky}$$

$$U(n) = L(n - 1) + \Theta(n) \quad \textit{unlucky}$$

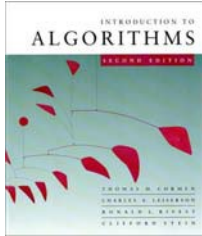
Solving:

$$L(n) = 2(L(n/2 - 1) + \Theta(n/2)) + \Theta(n)$$

$$= 2L(n/2 - 1) + \Theta(n)$$

$$= \Theta(n \lg n) \quad \textit{Lucky!}$$

How can we make sure we are usually lucky?



Randomized quicksort

IDEA: Partition around a *random* element.

- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.

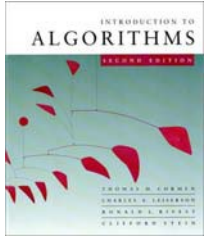
Randomized Partition

RANDOMIZED-PARTITION(A, p, r)

1 $i = \text{RANDOM}(p, r)$

2 exchange $A[r]$ with $A[i]$

3 **return** PARTITION(A, p, r)



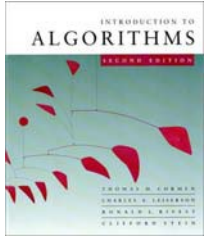
Randomized quicksort analysis

Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size n , assuming random numbers are independent.

For $k = 0, 1, \dots, n-1$, define the *indicator random variable*

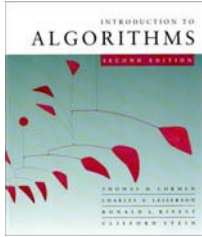
$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_k] = \Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.



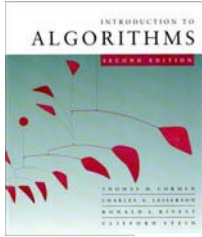
Analysis (continued)

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$



Analysis (continued)

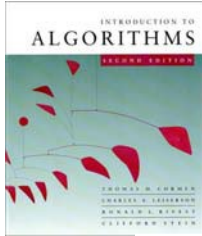
$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$
$$= \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))$$



Calculating expectation

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

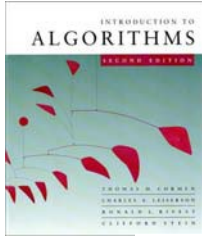
Take expectations of both sides.



Calculating expectation

$$\begin{aligned} E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))\right] \\ &= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \end{aligned}$$

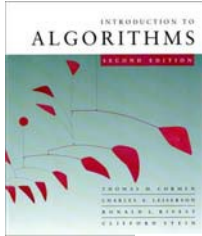
Linearity of expectation.



Calculating expectation

$$\begin{aligned} E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))\right] \\ &= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \\ &= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \end{aligned}$$

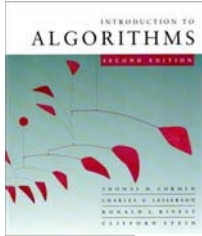
Independence of X_k from other random choices.



Calculating expectation

$$\begin{aligned} E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))\right] \\ &= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \\ &= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \\ &= \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) \end{aligned}$$

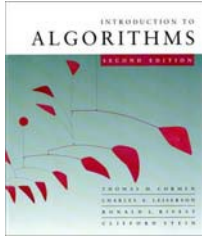
Linearity of expectation; $E[X_k] = 1/n$.



Calculating expectation

$$\begin{aligned} E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))\right] \\ &= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \\ &= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \\ &= \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) \\ &= \frac{2}{n} \sum_{k=1}^{n-1} E[T(k)] + \Theta(n) \end{aligned}$$

Summations have identical terms.



Hairy recurrence

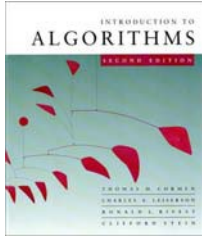
$$E[T(n)] = \frac{2}{n} \sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

(The $k = 0, 1$ terms can be absorbed in the $\Theta(n)$.)

Prove: $E[T(n)] \leq an \lg n$ for constant $a > 0$.

- Choose a large enough so that $an \lg n$ dominates $E[T(n)]$ for sufficiently small $n \geq 2$.

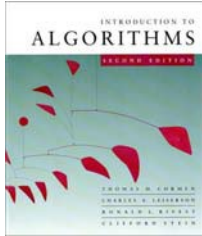
Use fact: $\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$ (exercise).



Substitution method

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

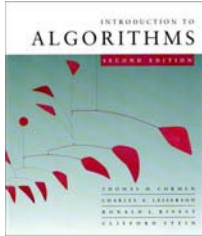
Substitute inductive hypothesis.



Substitution method

$$\begin{aligned} E[T(n)] &\leq \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n) \\ &\leq \frac{2a}{n} \left(\frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + \Theta(n) \end{aligned}$$

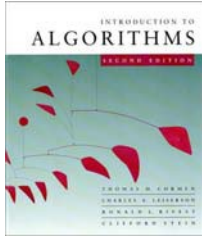
Use fact.



Substitution method

$$\begin{aligned} E[T(n)] &\leq \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n) \\ &\leq \frac{2a}{n} \left(\frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + \Theta(n) \\ &= an \lg n - \left(\frac{an}{4} - \Theta(n) \right) \end{aligned}$$

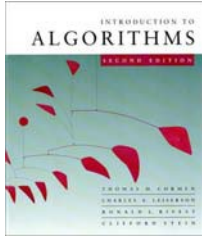
Express as *desired – residual*.



Substitution method

$$\begin{aligned} E[T(n)] &\leq \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n) \\ &\leq \frac{2a}{n} \left(\frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + \Theta(n) \\ &= an \lg n - \left(\frac{an}{4} - \Theta(n) \right) \\ &\leq an \lg n, \end{aligned}$$

if a is chosen large enough so that $an/4$ dominates the $\Theta(n)$.



Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from *code tuning*.
- Quicksort behaves well even with caching and virtual memory.

Summary

- Heaps
 - Useful for building priority queues
- Heapsort
 - Sorts “in place” $\rightarrow \Theta(1)$ space
 - Worst case time $\Theta(n \lg n)$
- Quicksort
 - Sorts “in place” $\rightarrow \Theta(1)$ space
 - Best case time $\Theta(n \lg n)$
 - Average case time $\Theta(n \lg n)$
 - Worst case time $\Theta(n^2)$

Recap

- Heapsort
- Quicksort
- Next:
 - Linear time sorting
 - Order statistics