CMP448: Algorithms

Lecture 03: Heapsort and Quicksort

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

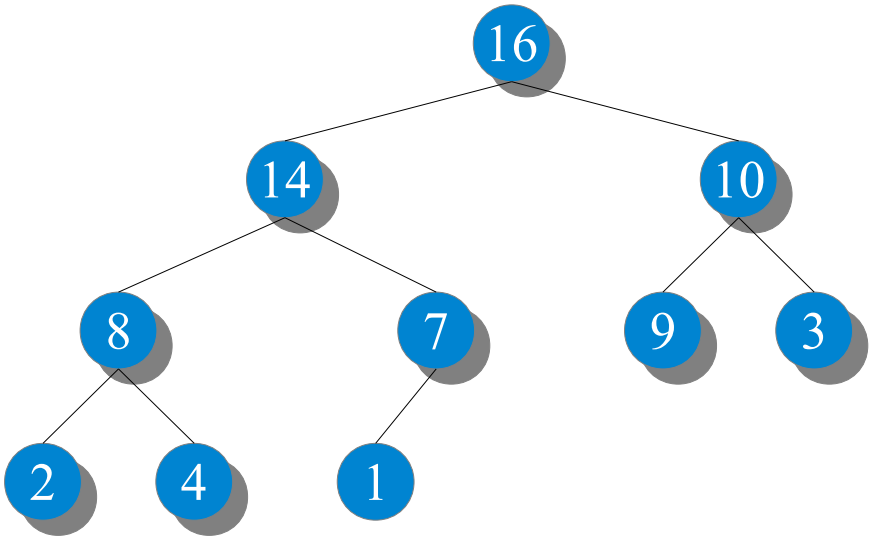
Agenda

- 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 ≥ values of the children
  - Min-Heap: the value of the node ≤ values of the children

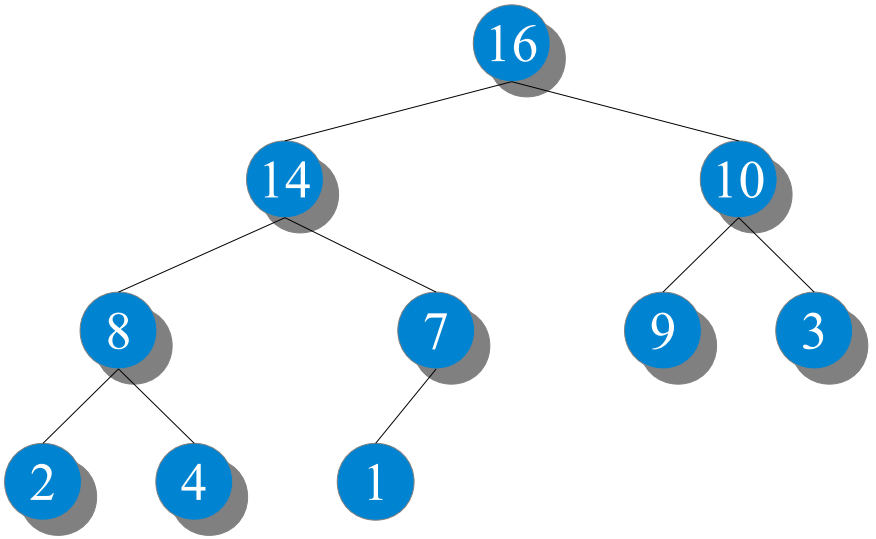


Max-Heap

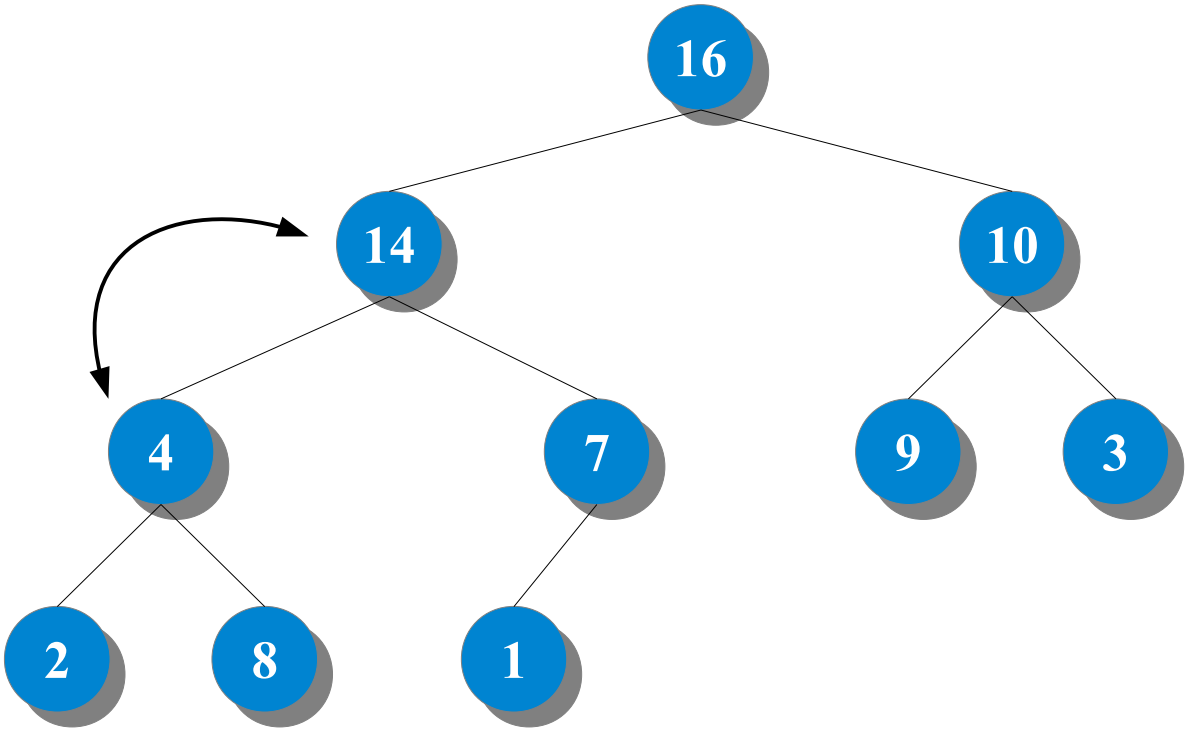
Heaps

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

Max-Heap



Heap Building



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

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 ⌊i/2⌋
```

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

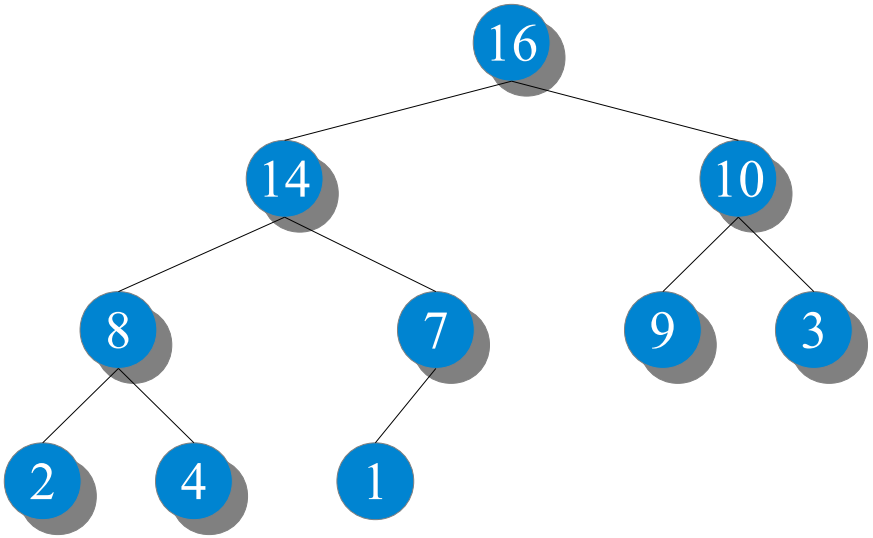
```
PARENT(i)
1  return ⌊i/2⌋
```

Left child of a node i

```
LEFT(i)
1  return 2i
```

Right child of a node i

```
RIGHT(i)
1  return 2i + 1
```



Conceptual Heap Representation

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

Array Representation

Array Representation

Parent of a node i

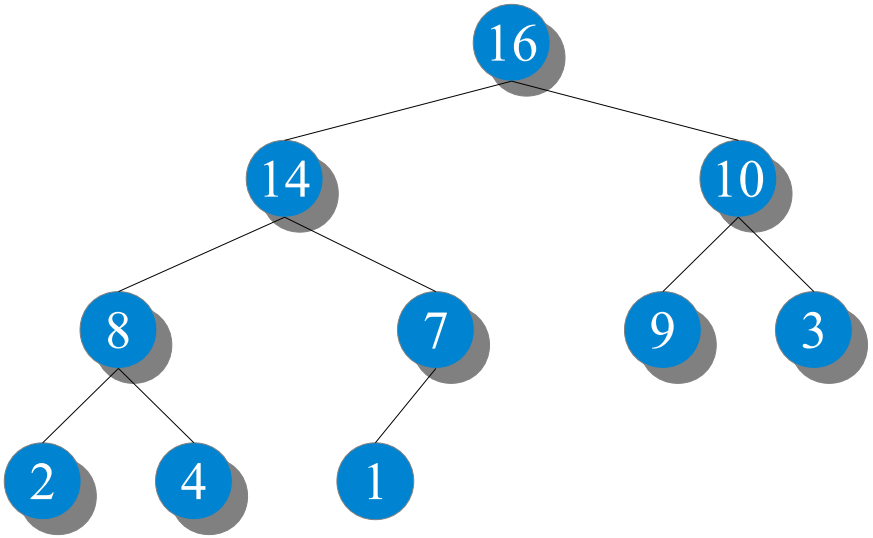
```
PARENT(i)
1  return ⌊i/2⌋
```

Left child of a node i

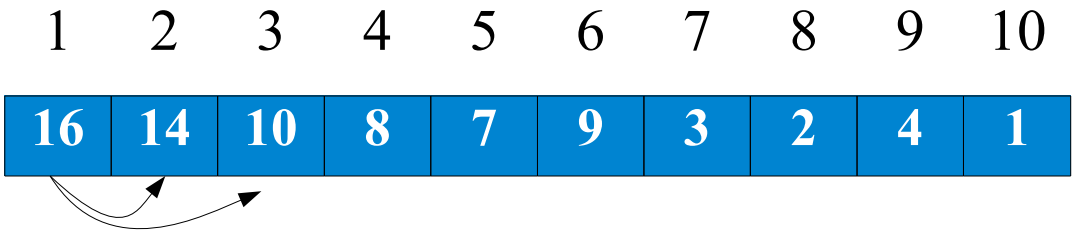
```
LEFT(i)
1  return 2i
```

Right child of a node i

```
RIGHT(i)
1  return 2i + 1
```



Conceptual Heap Representation



Array Representation

Array Representation

Parent of a node i

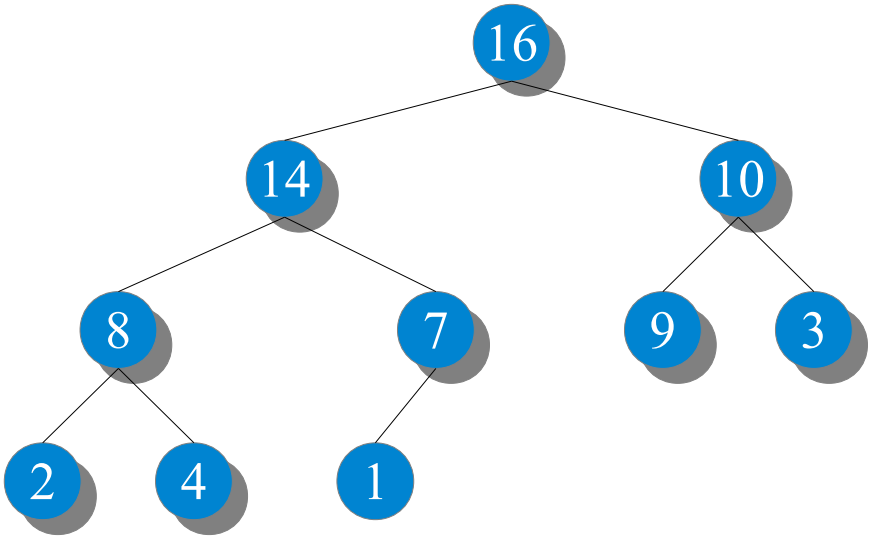
```
PARENT(i)
1  return ⌊i/2⌋
```

Left child of a node i

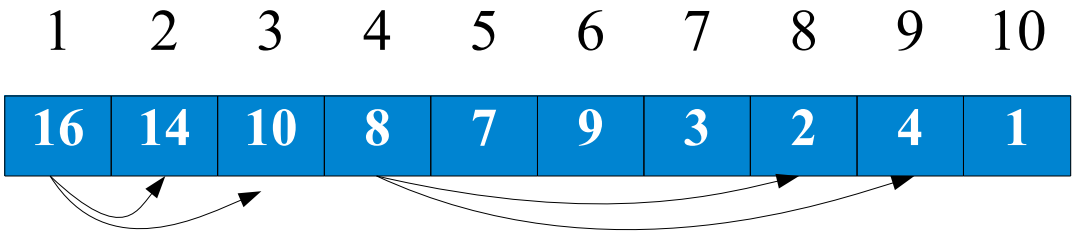
```
LEFT(i)
1  return 2i
```

Right child of a node i

```
RIGHT(i)
1  return 2i + 1
```



Conceptual Heap Representation

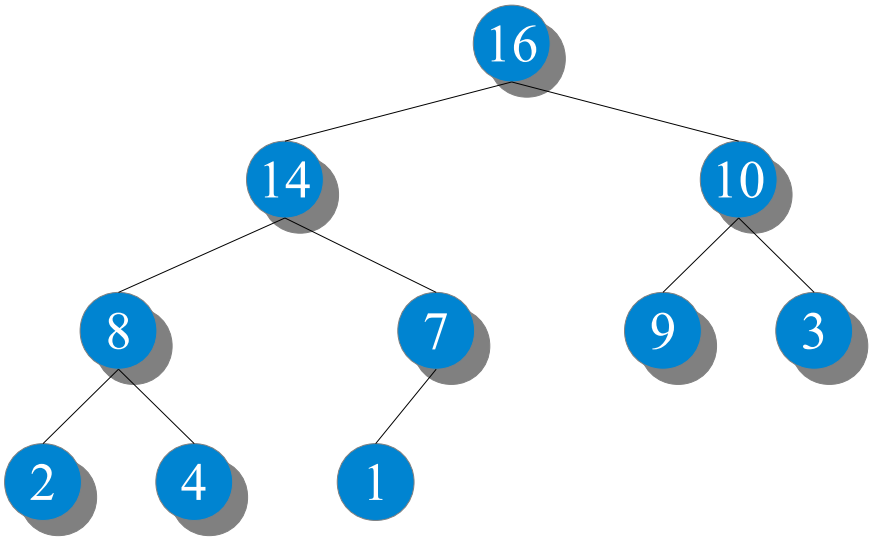


Array Representation

Max-Heaps

Max-Heap Property

$$A[Parent(i)] \geq A[i]$$



Properties of array *A*

A.length: size of array *A*

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

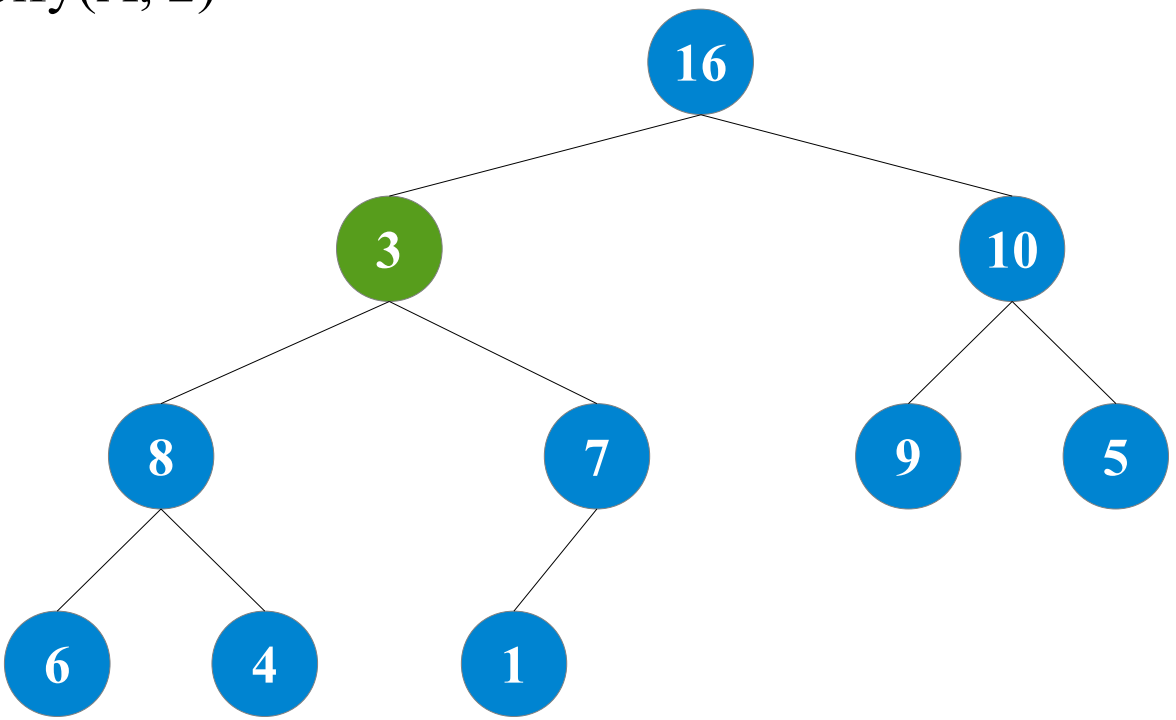
Max-Heapify

- 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

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

Max-Heapify

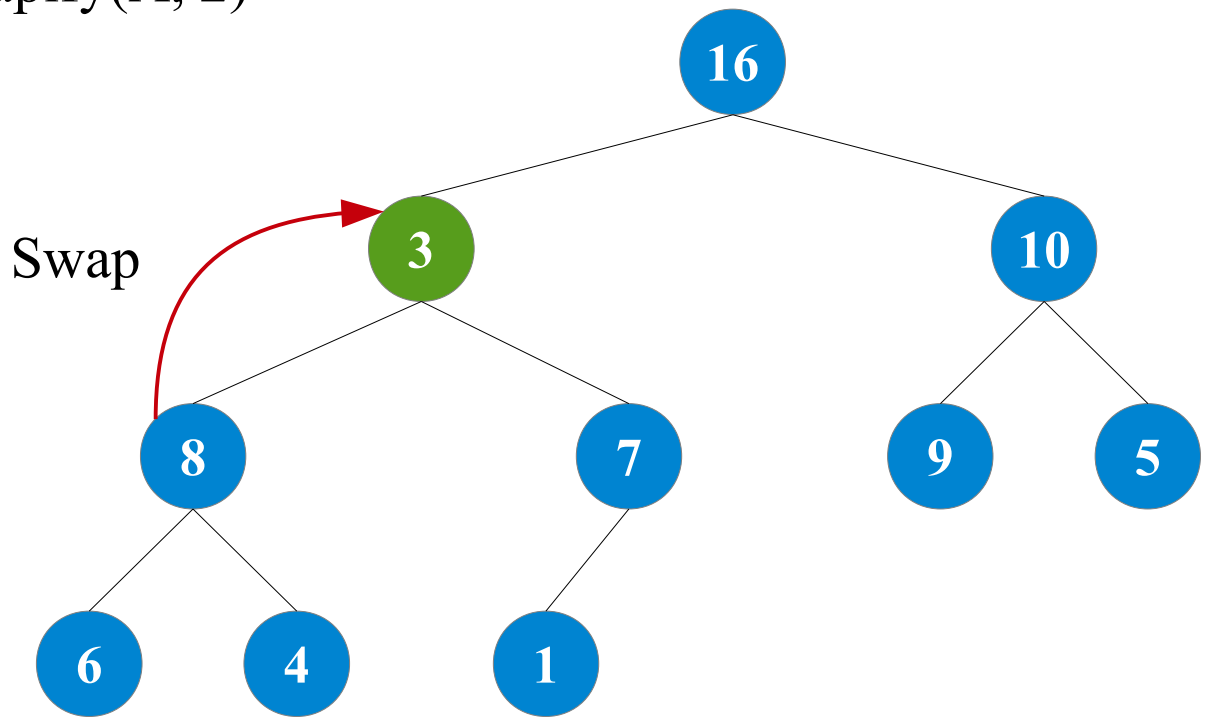
Max-Heapify(A, 2)



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

Max-Heapify

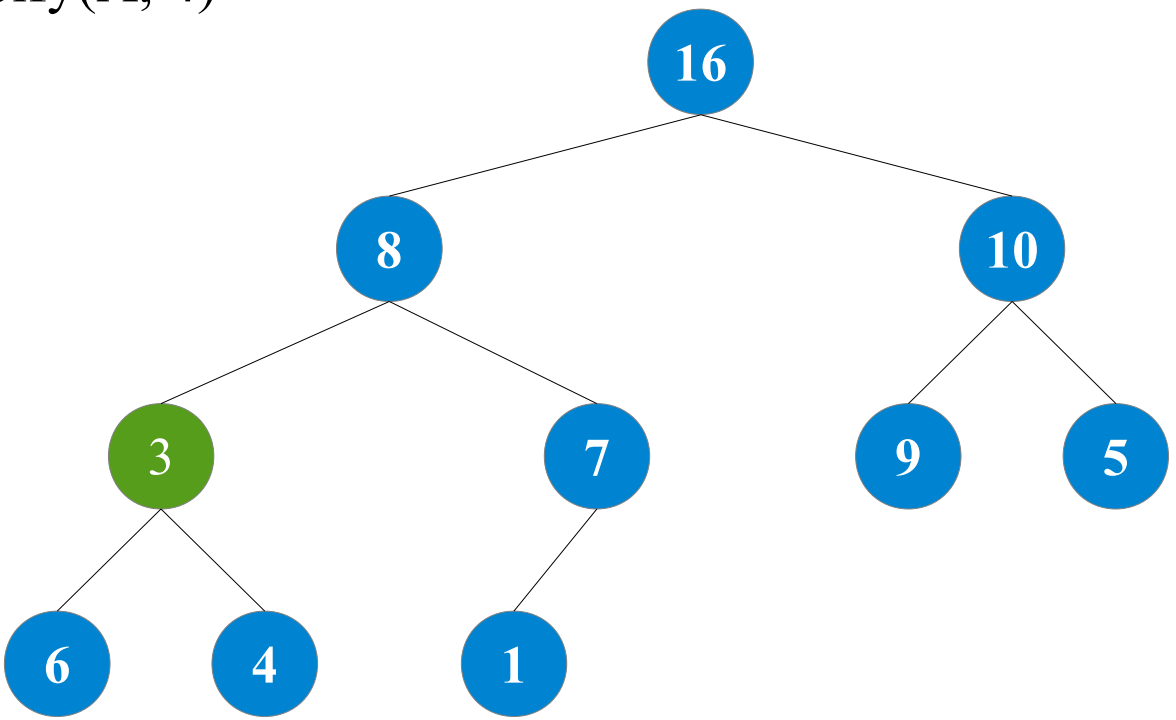
Max-Heapify(A, 2)



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

Max-Heapify

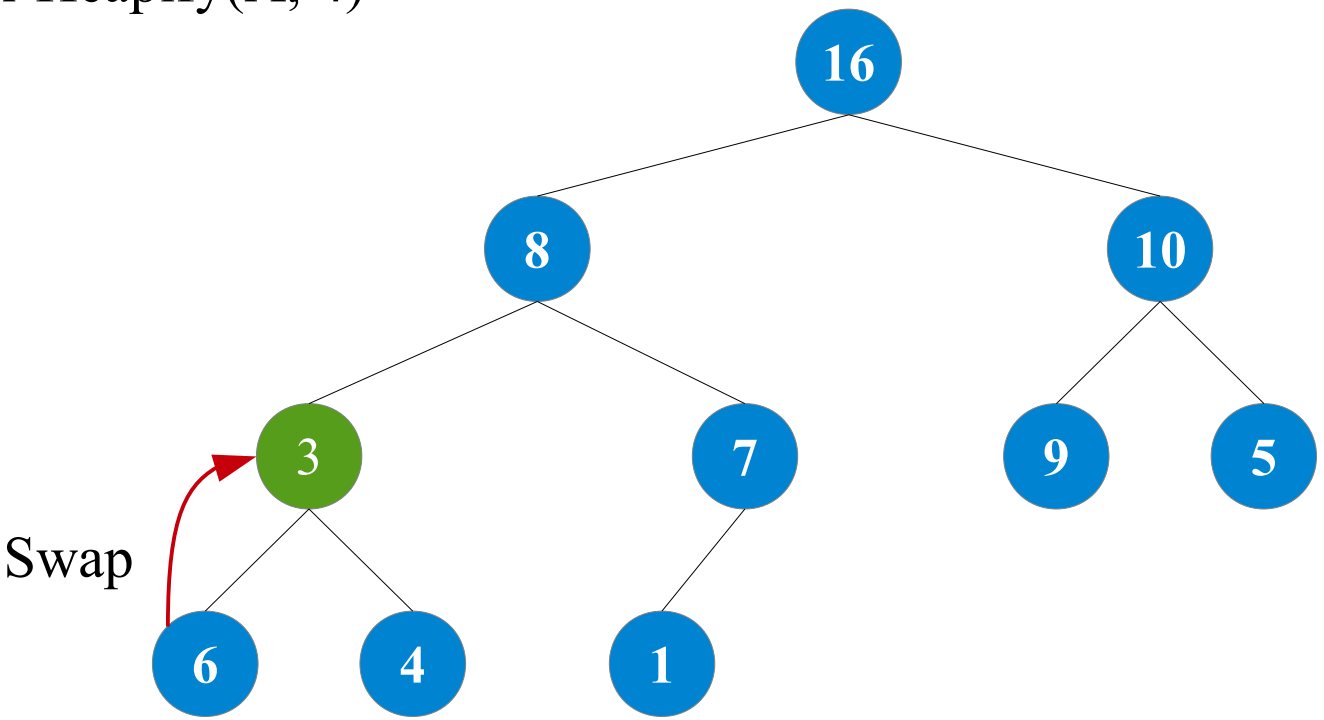
Max-Heapify(A, 4)



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

Max-Heapify

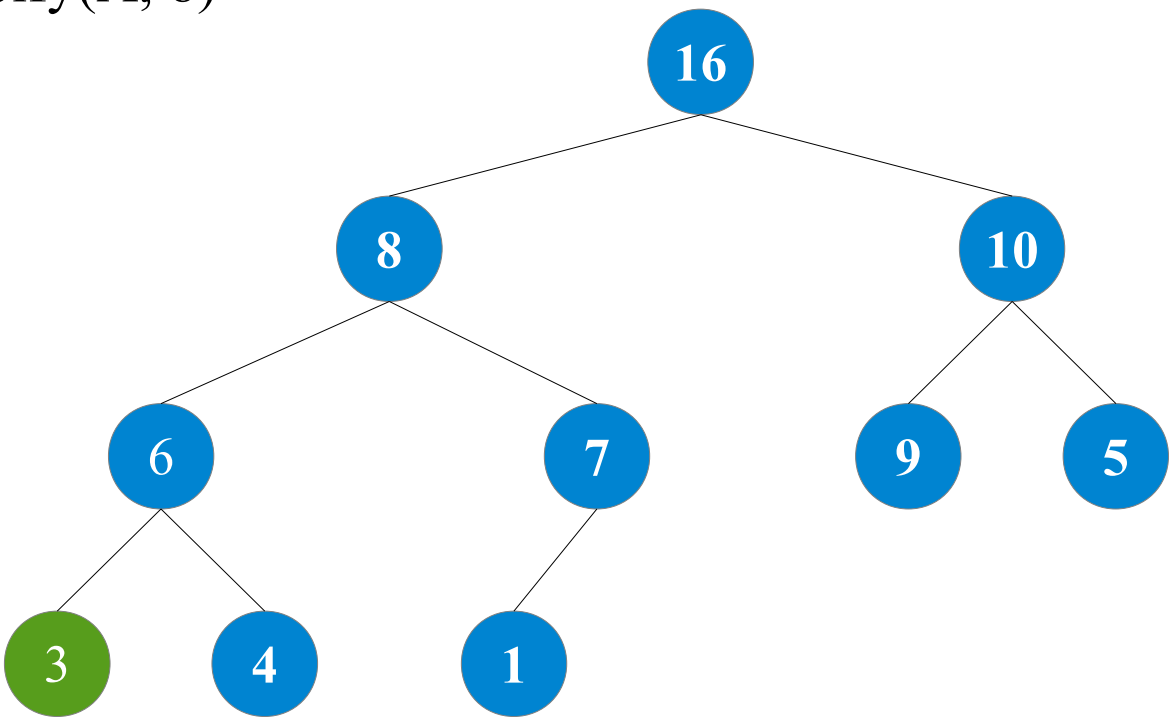
Max-Heapify(A, 4)



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

Max-Heapify

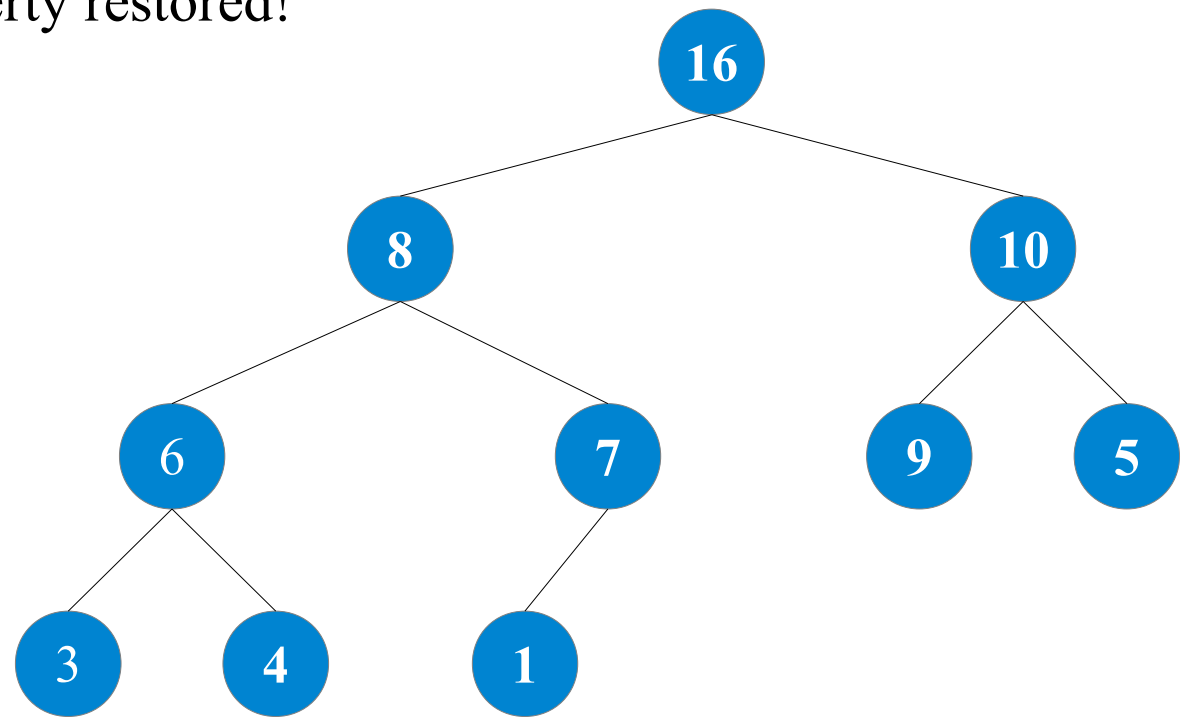
Max-Heapify(A, 8)



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

Max-Heapify

Heap property restored!



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

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?

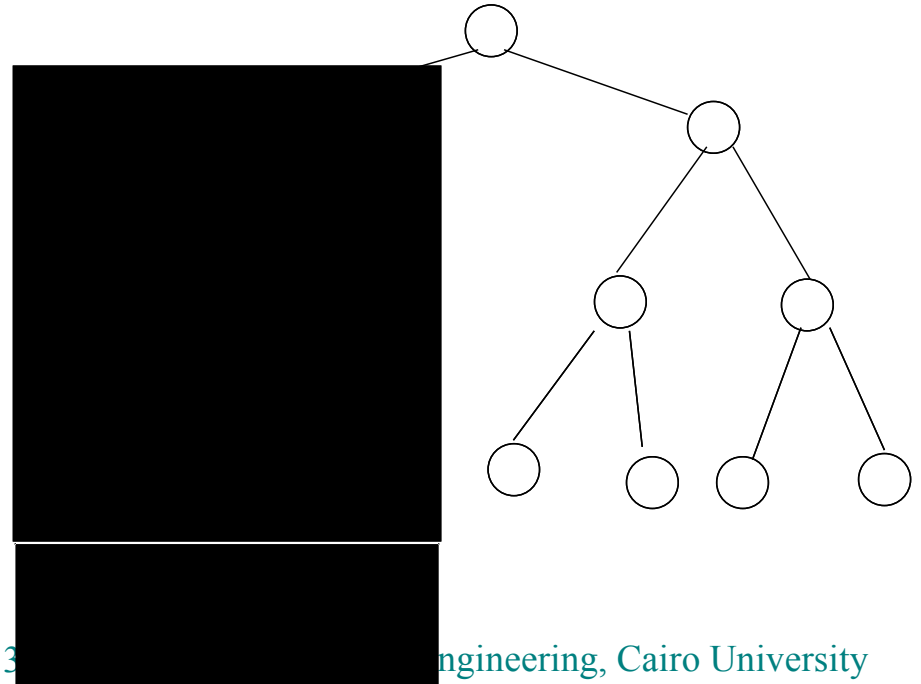
```
MAX-HEAPIFY (A, i)
1 l = LEFT(i)
2 r = RIGHT(i)
3 if l ≤ A.heap-size and A[l] > A[i]
4     largest = l
5 else largest = i
6 if r ≤ A.heap-size and A[r] > A[largest]
7     largest = r
8 if largest ≠ i
9     exchange A[i] with A[largest]
10    MAX-HEAPIFY(A, largest)
```

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

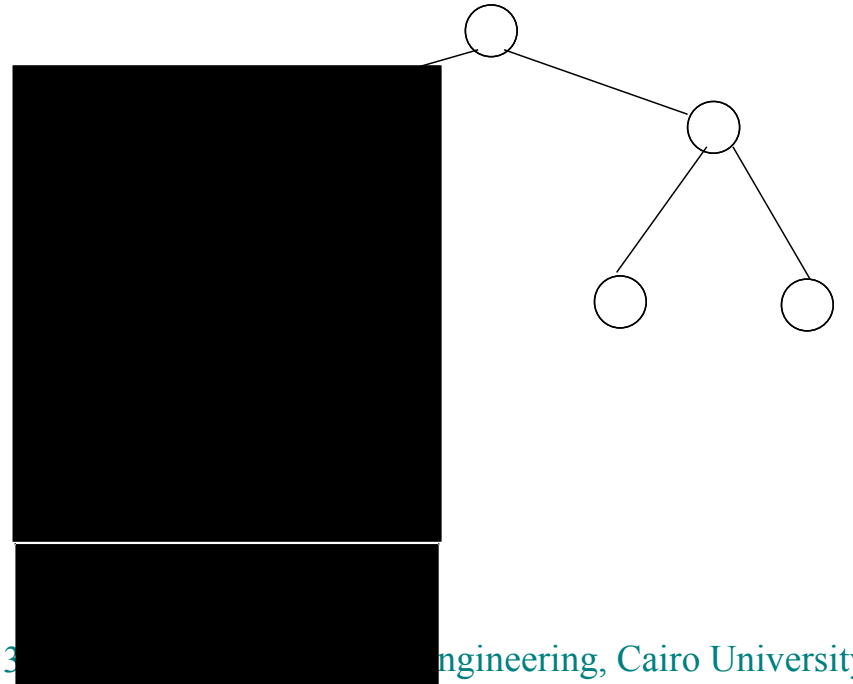
7 nodes in left subtree

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

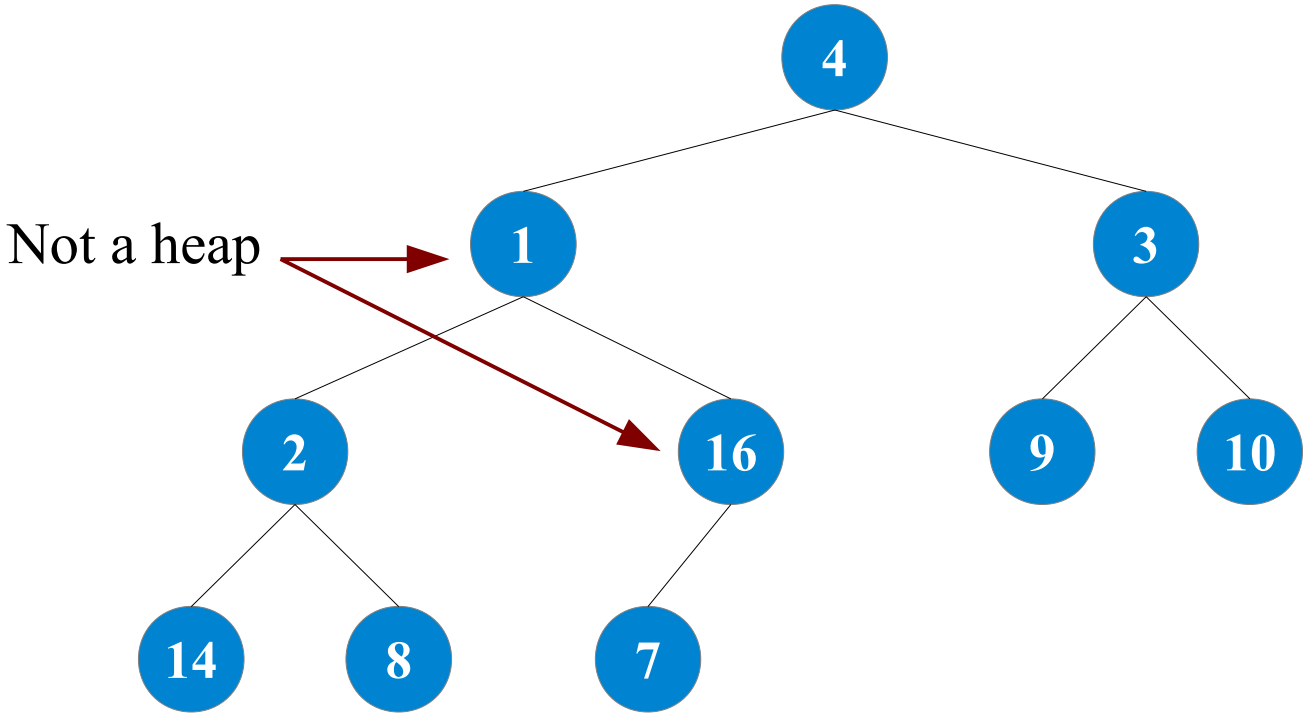
- $T(n) \leq T(2n/3) + \Theta(1)$
- Case 2 of the Master Theorem ($a = 1$, $b = 3/2$):
  - $T(n) = O(\lg n)$
- Can also represent it as a function of h, the height of the node
  - The height is the number of links from the node to the leaves
  - $T(n) = O(h)$

Heap Building

```
Build-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length / 2⌋ downto 1
3     MAX-HEAPIFY(A, i)
```

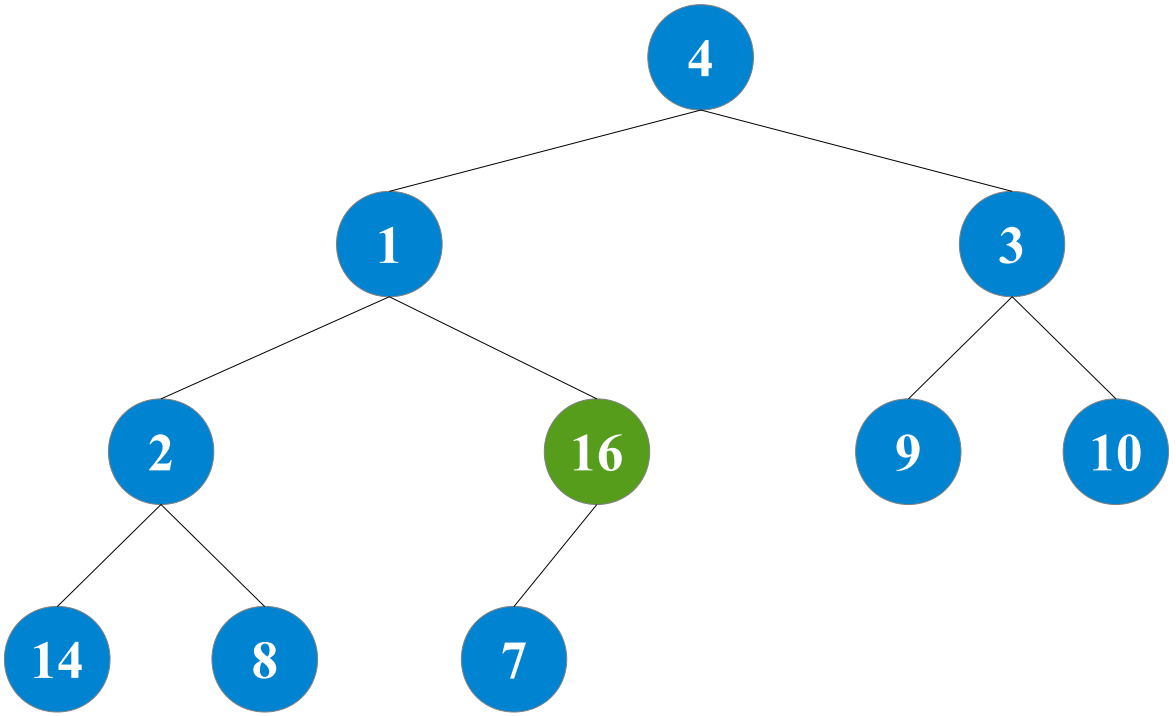
- Each leaf is already a heap, albeit a useless one!
- Start from the second to last level, and heapify i.e. move values violating the Heap property downwards
- Ends with a Max-Heap!

Heap Building



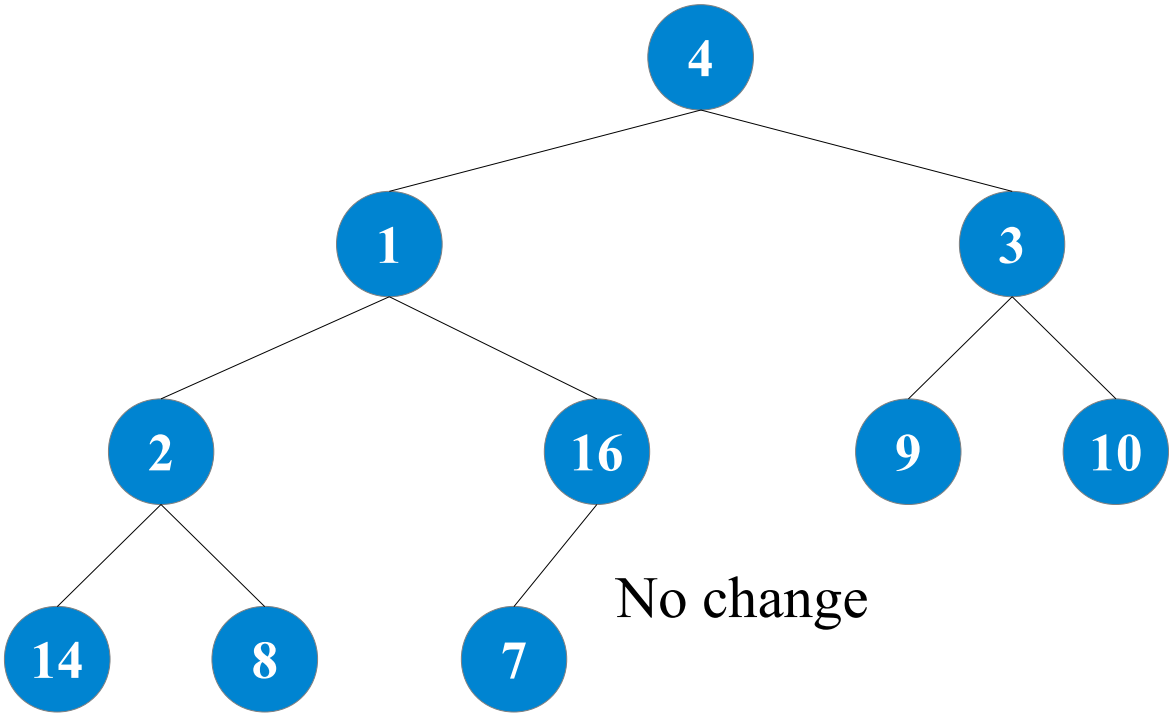
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Heap Building



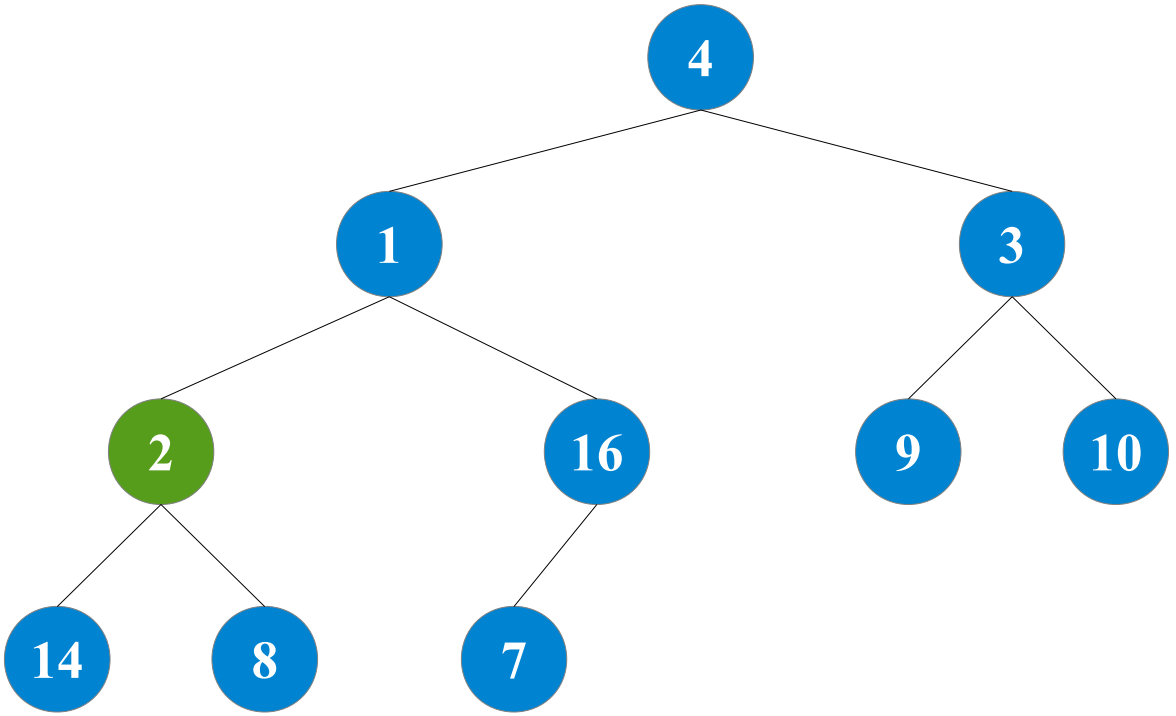
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Heap Building



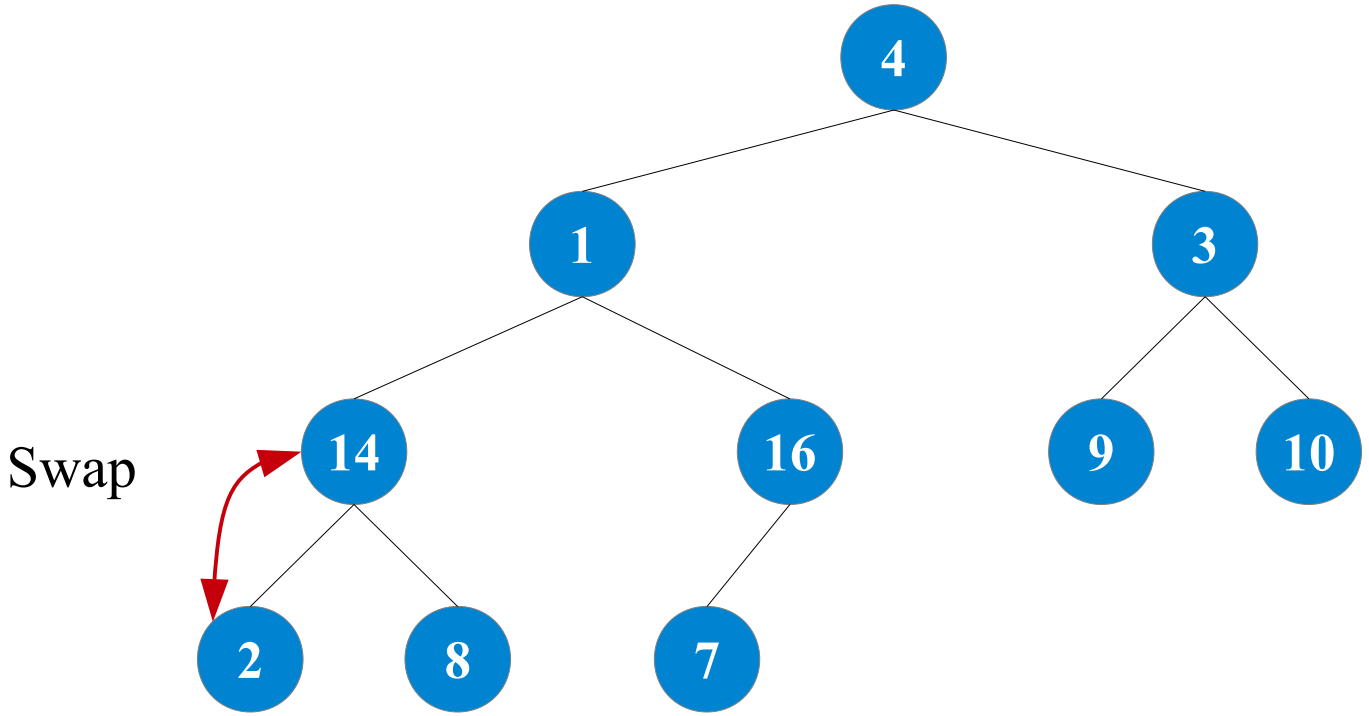
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Heap Building



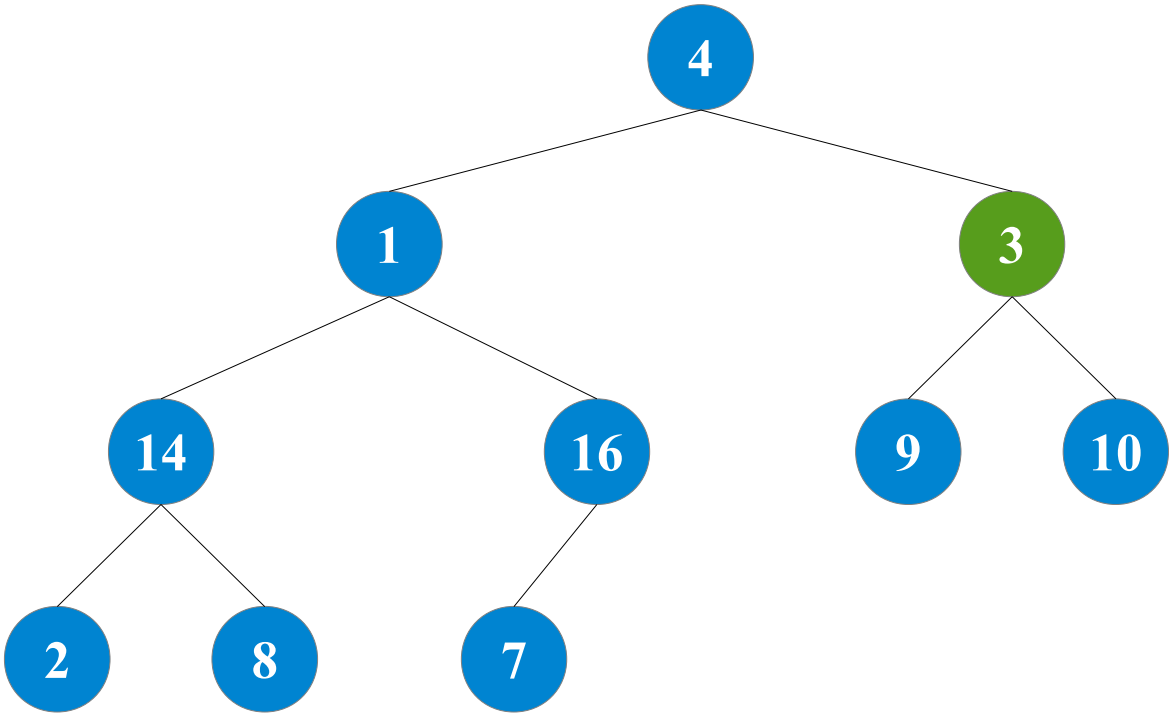
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Heap Building



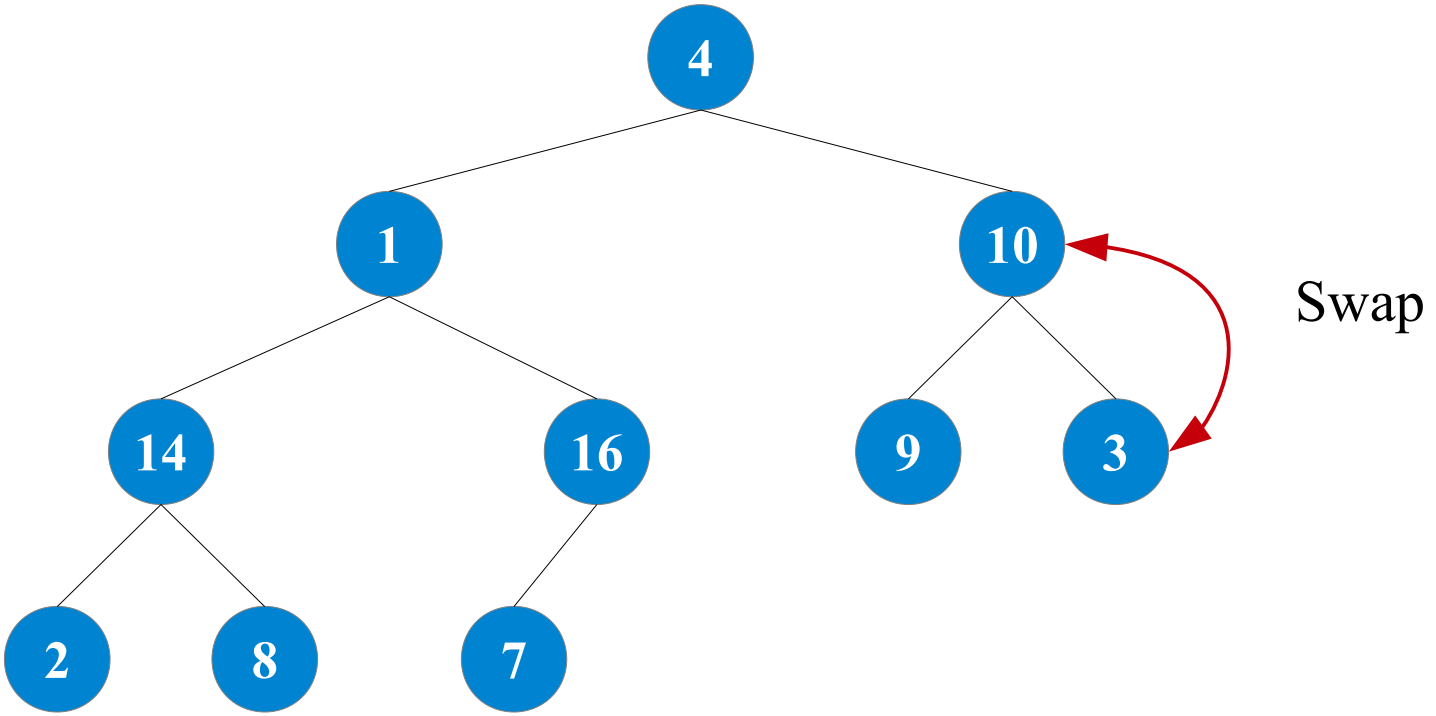
1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7

Heap Building



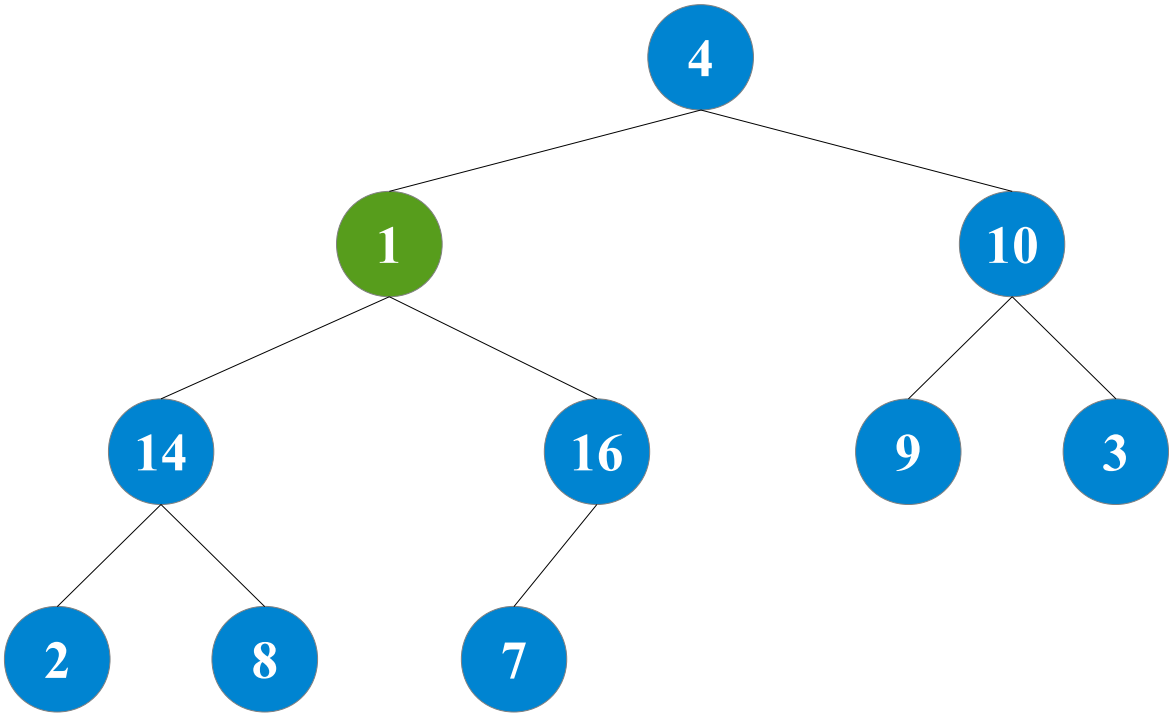
1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7

Heap Building



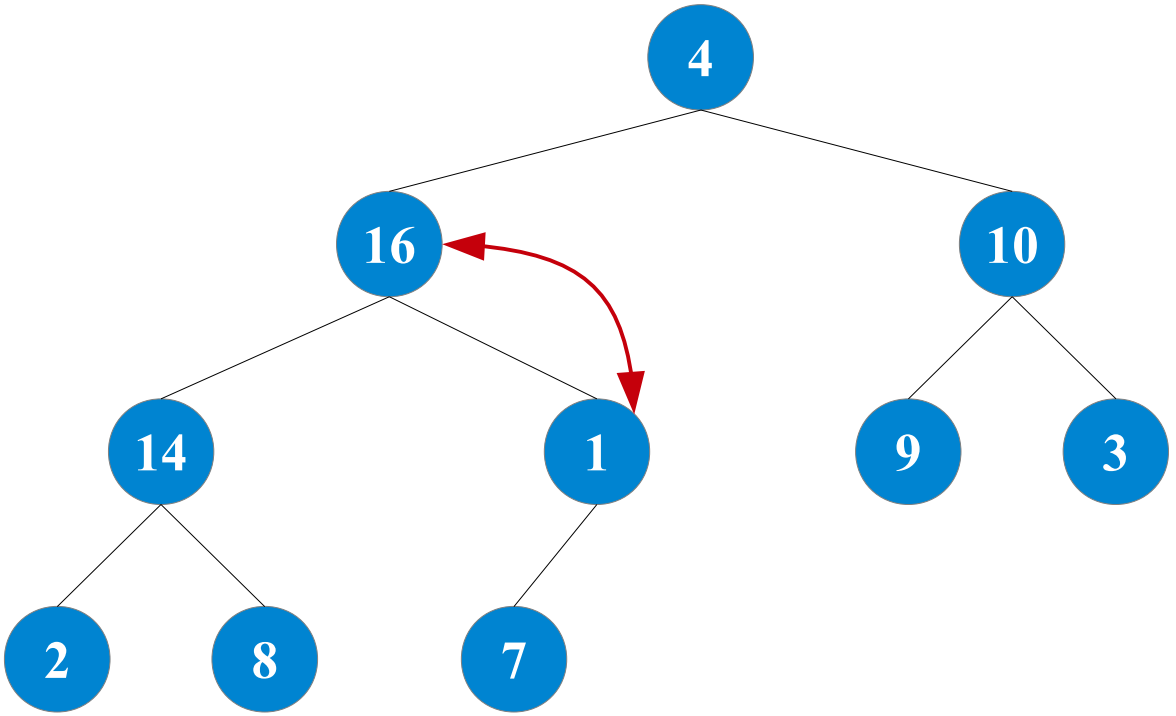
1	2	3	4	5	6	7	8	9	10
4	1	10	14	16	9	3	2	8	7

Heap Building



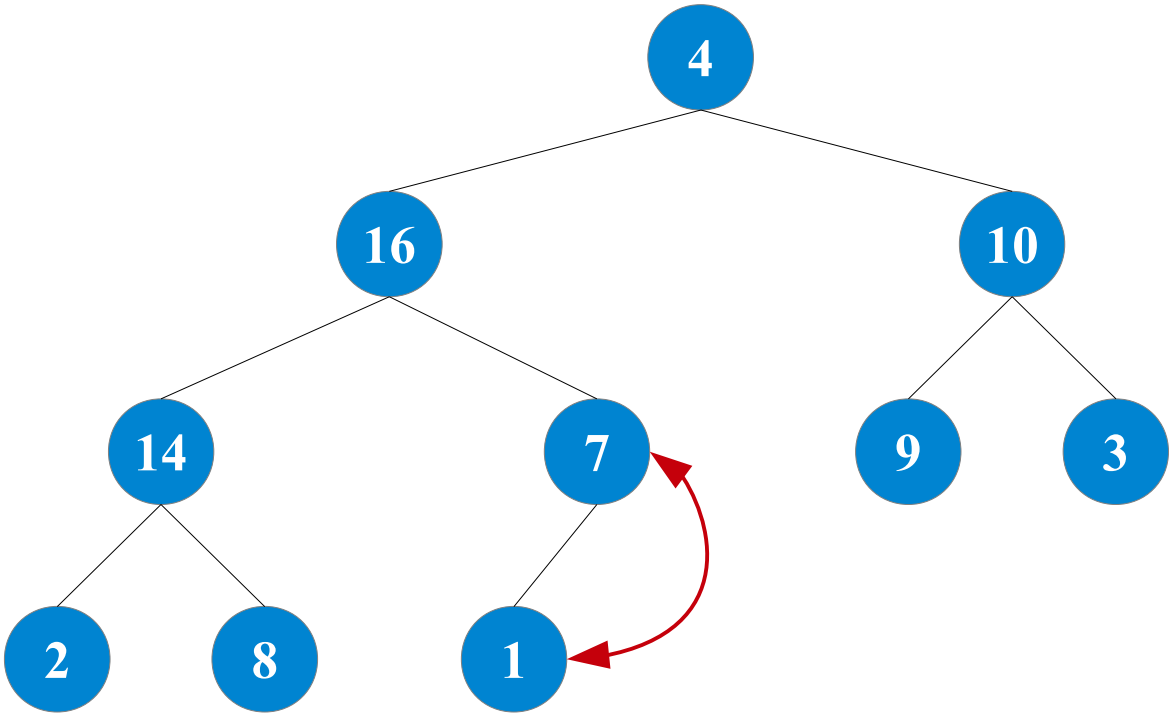
1	2	3	4	5	6	7	8	9	10
4	1	10	14	16	9	3	2	8	7

Heap Building



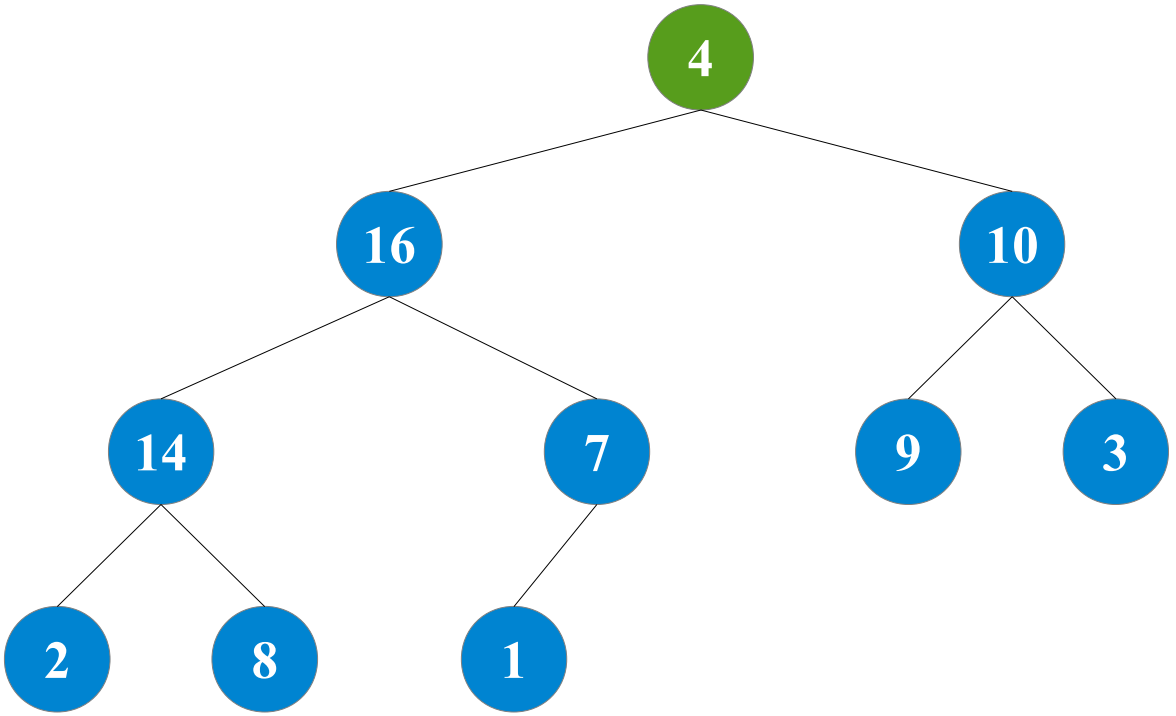
1	2	3	4	5	6	7	8	9	10
4	16	10	14	1	9	3	2	8	7

Heap Building



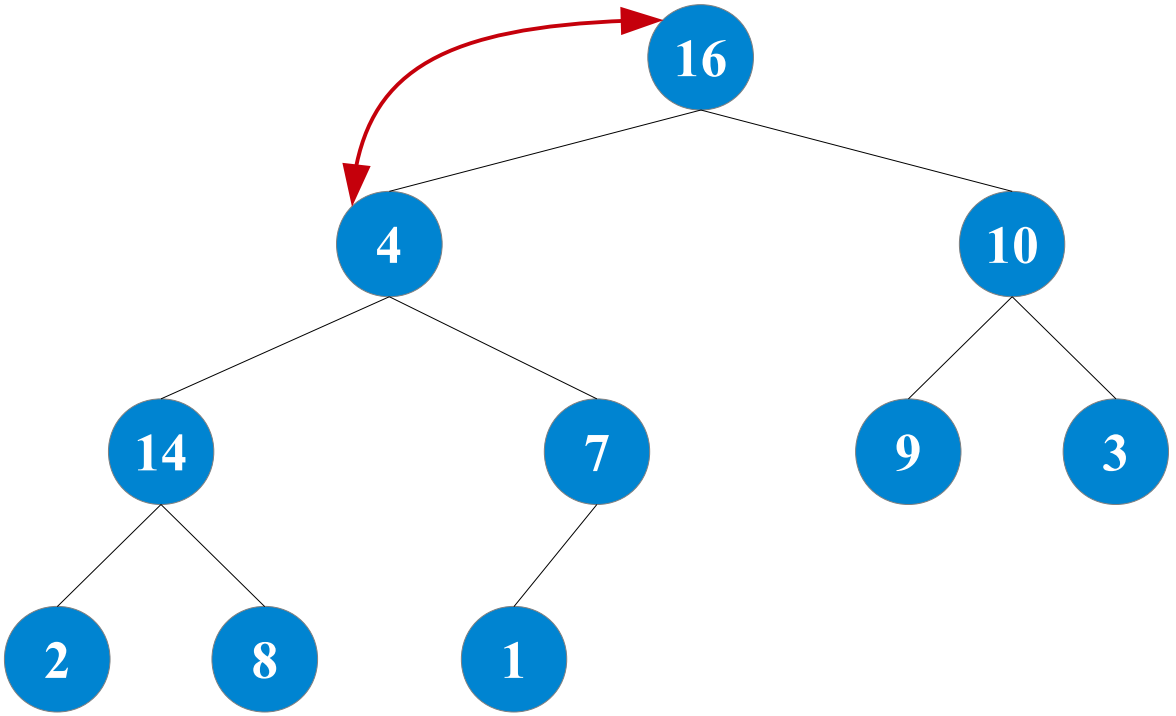
1	2	3	4	5	6	7	8	9	10
4	16	10	14	7	9	3	2	8	1

Heap Building



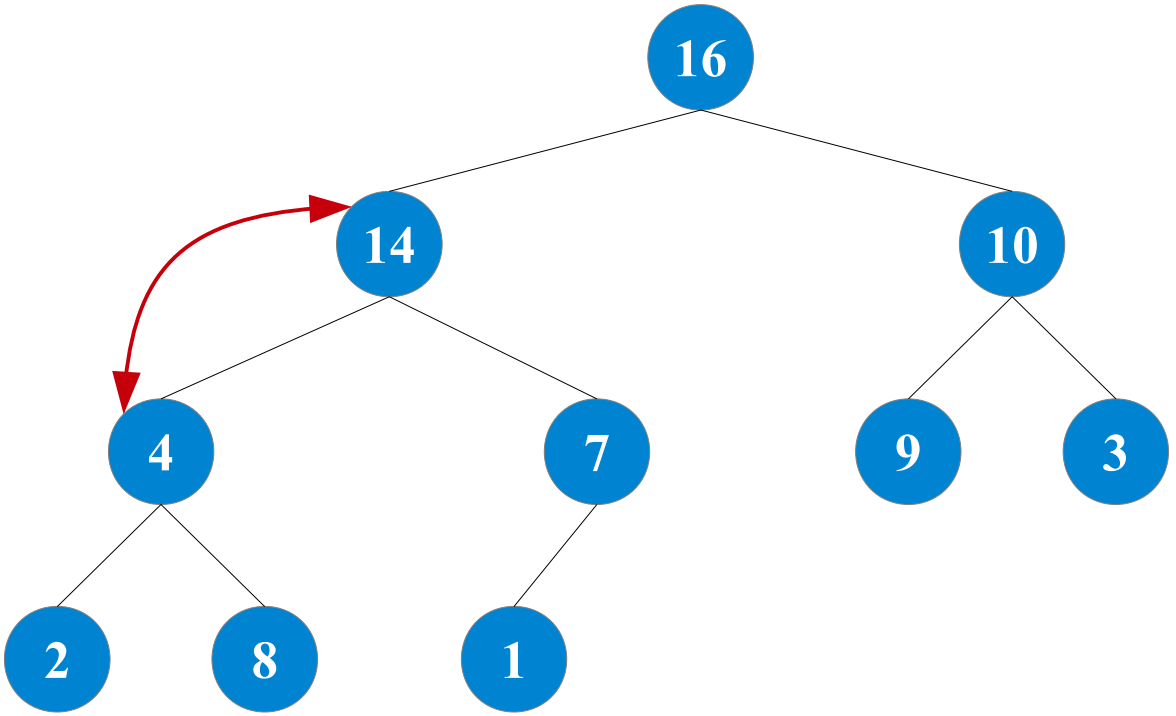
1	2	3	4	5	6	7	8	9	10
4	16	10	14	7	9	3	2	8	1

Heap Building



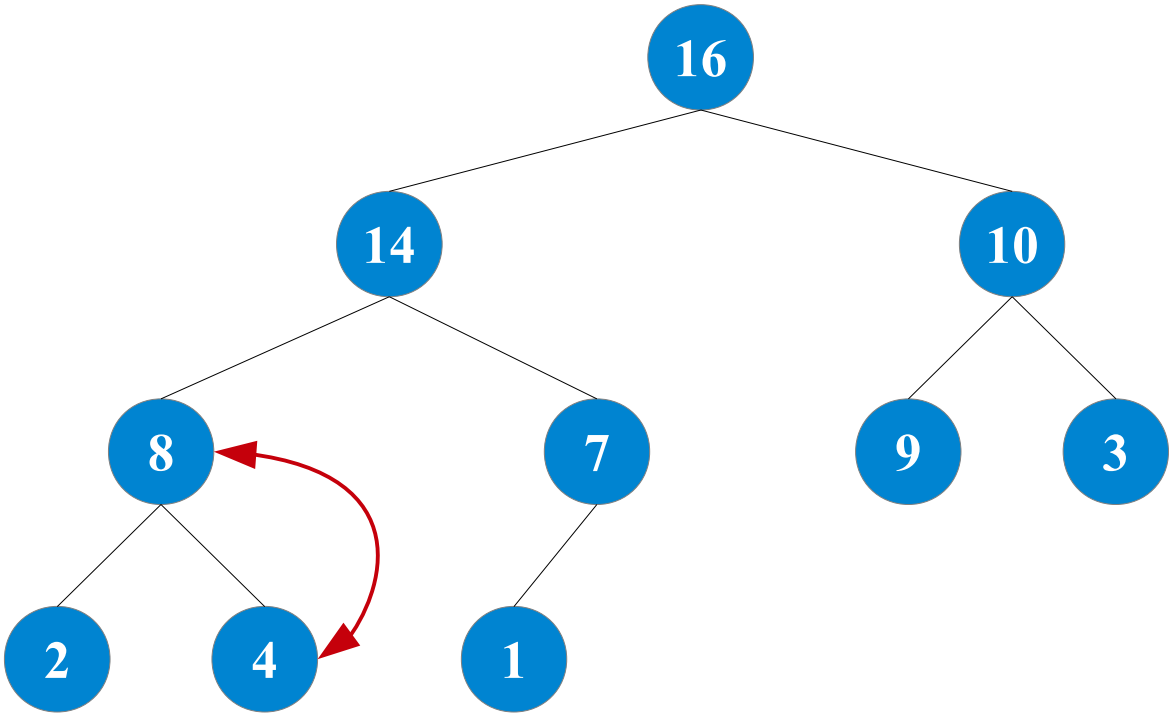
1	2	3	4	5	6	7	8	9	10
16	4	10	14	7	9	3	2	8	1

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

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

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

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

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

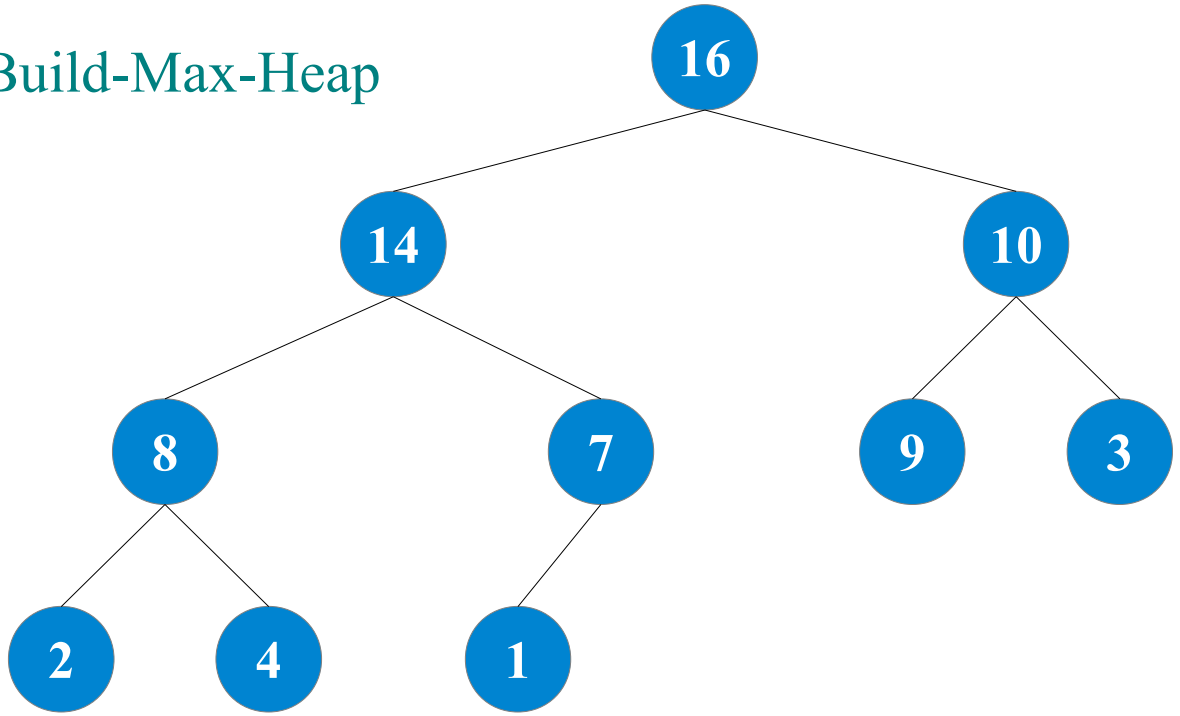
Heapsort

```
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, i)
```

- Build a Max-Heap from the array → 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 ...

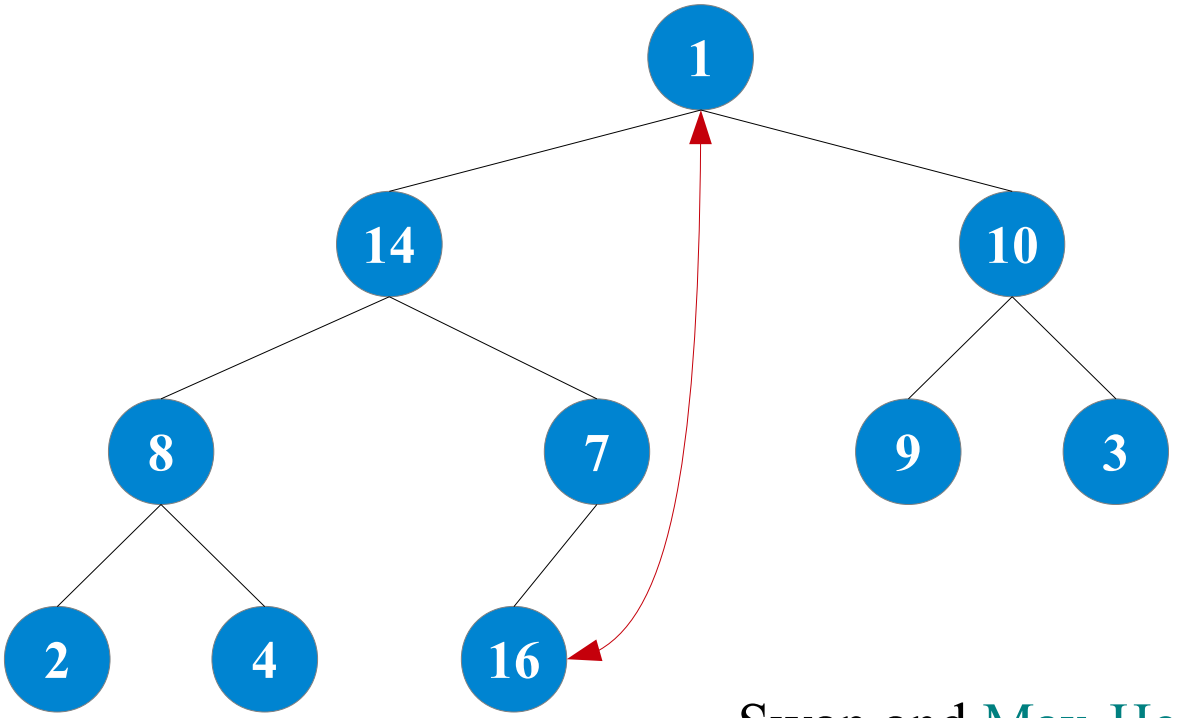
Heapsort

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

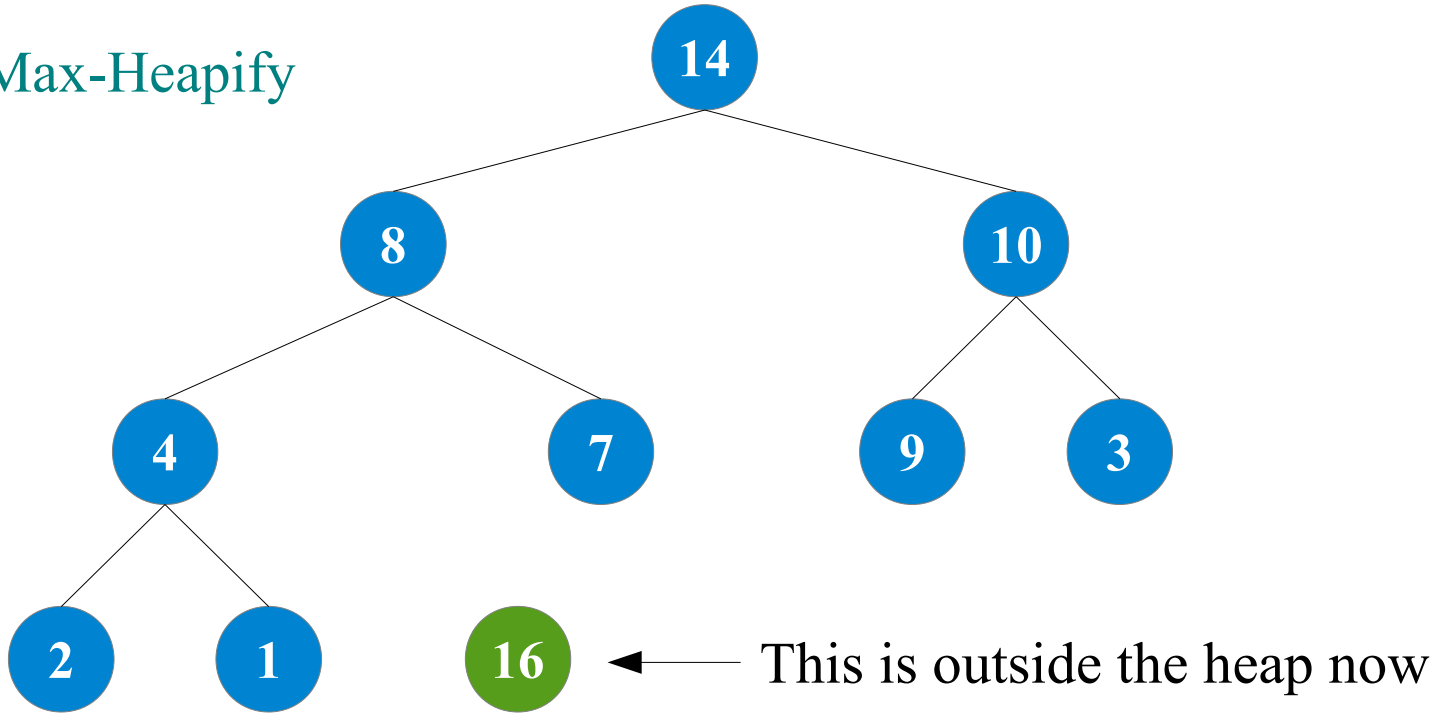


Swap and Max-Heapify

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

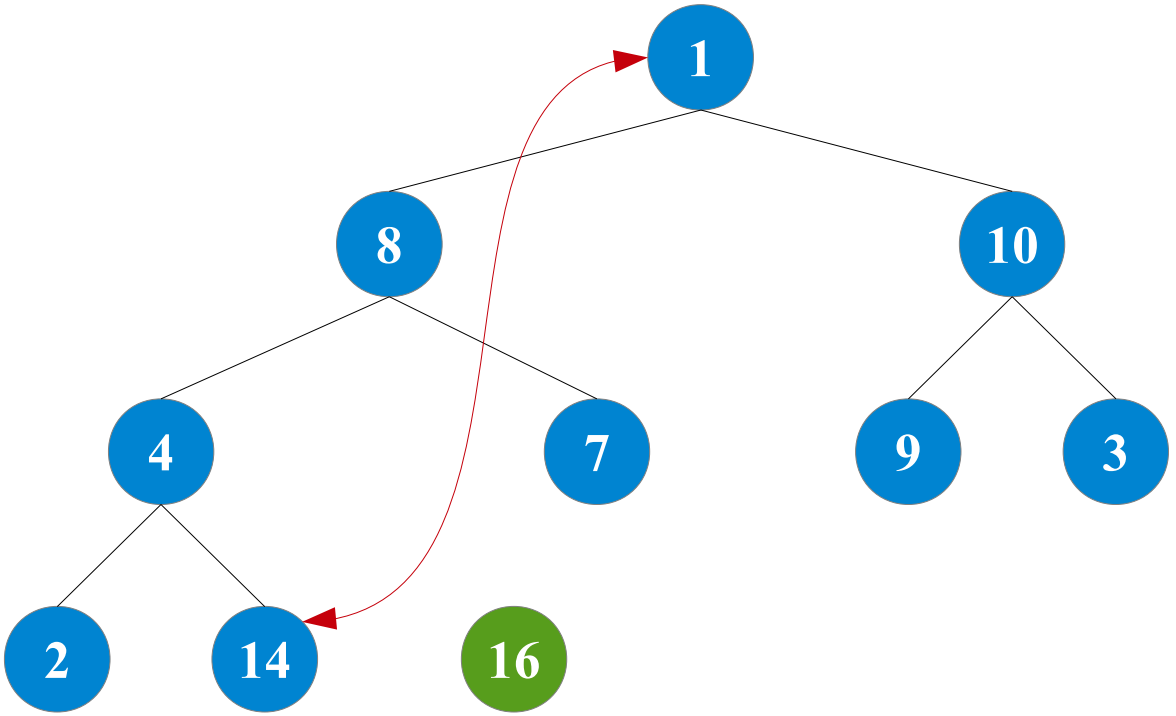
Heapsort

After calling Max-Heapify



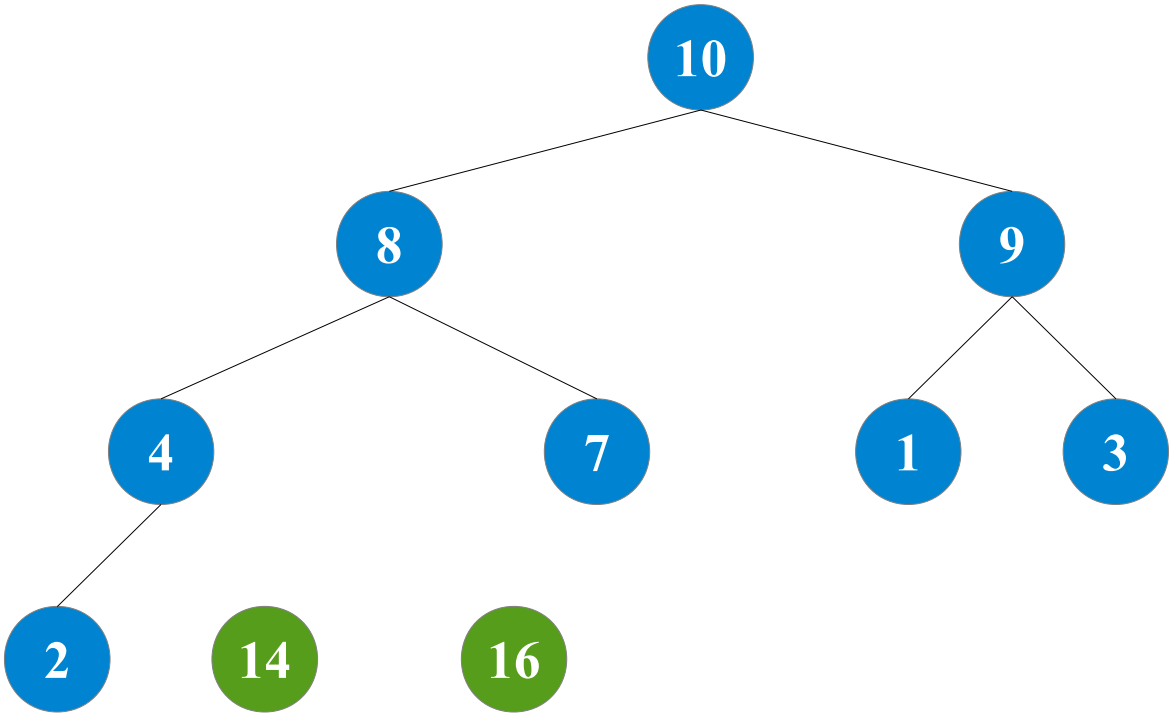
1	2	3	4	5	6	7	8	9	10
14	8	10	4	7	9	3	2	1	16

Heapsort



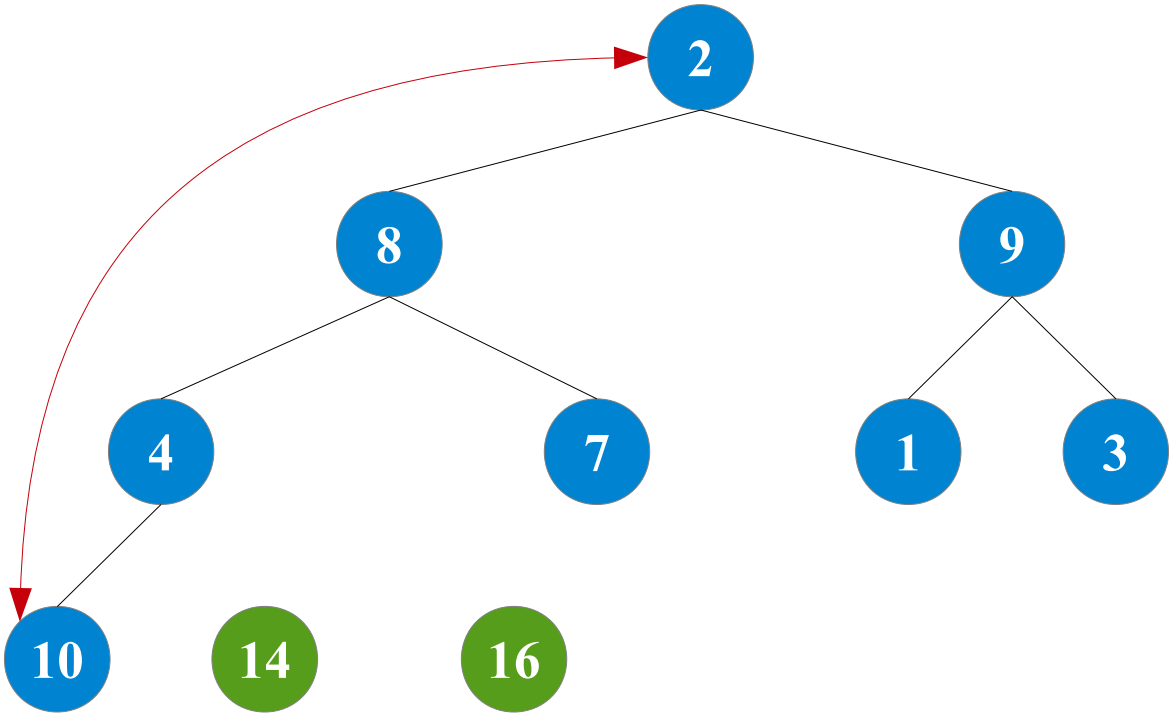
1	2	3	4	5	6	7	8	9	10
1	8	10	4	7	9	3	2	14	16

Heapsort



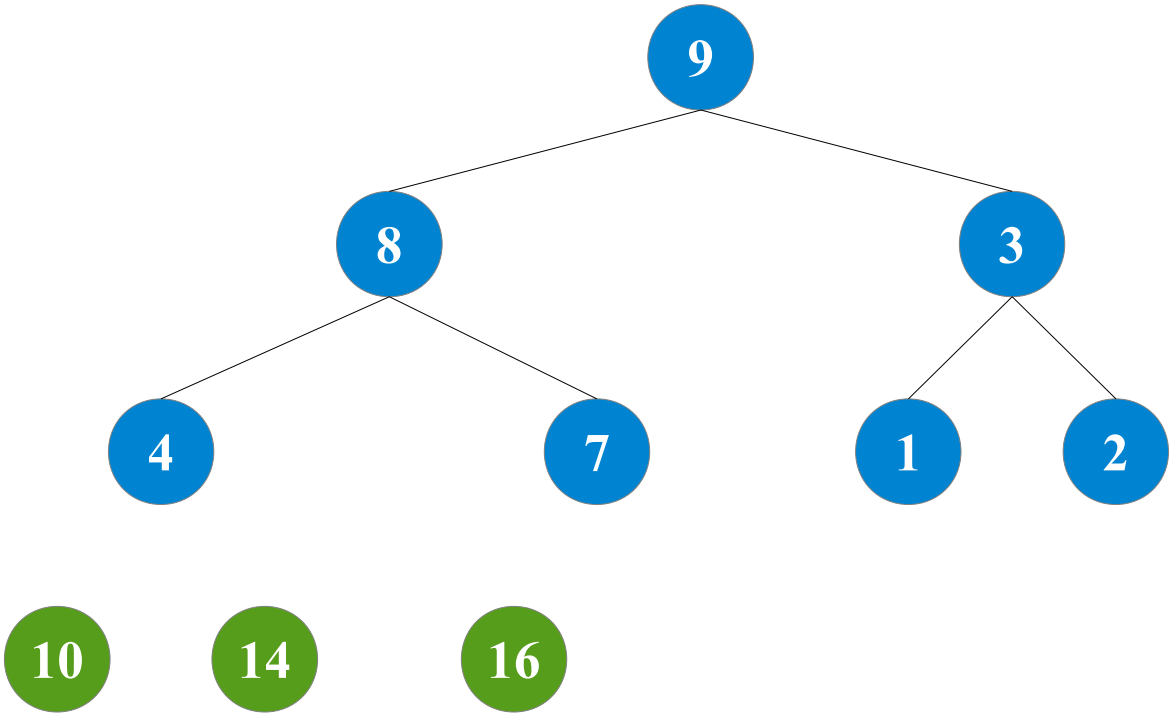
1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	3	2	14	16

Heapsort



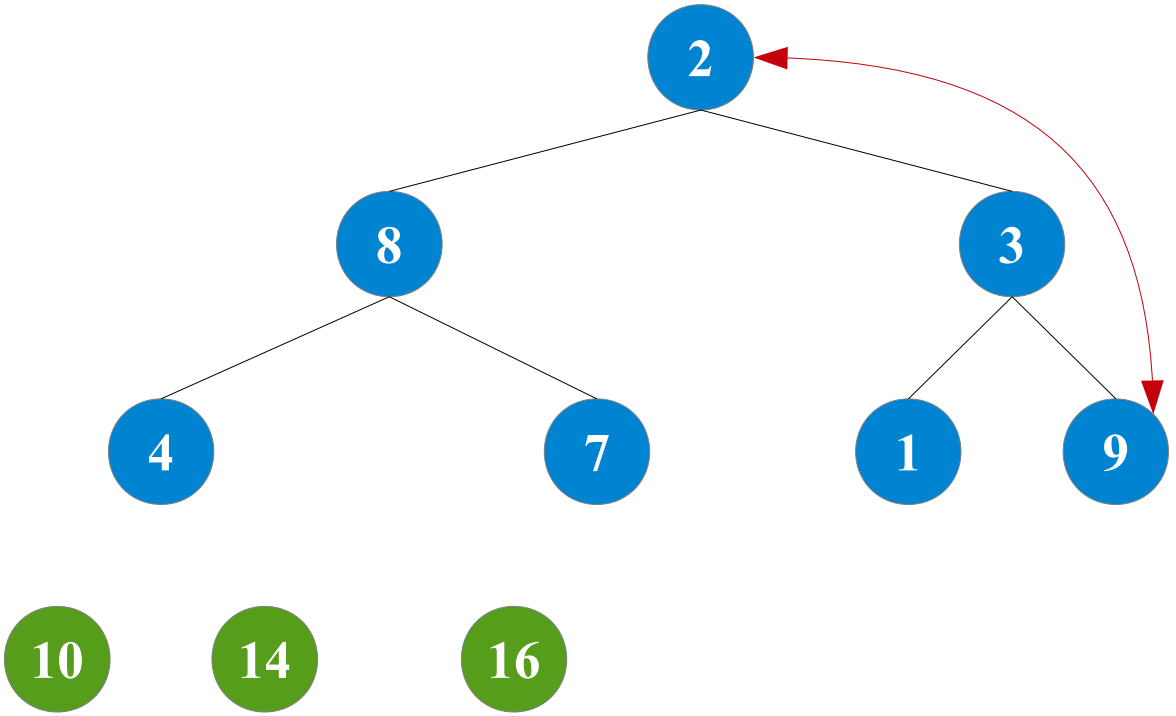
1	2	3	4	5	6	7	8	9	10
2	8	9	4	7	1	3	10	14	16

Heapsort



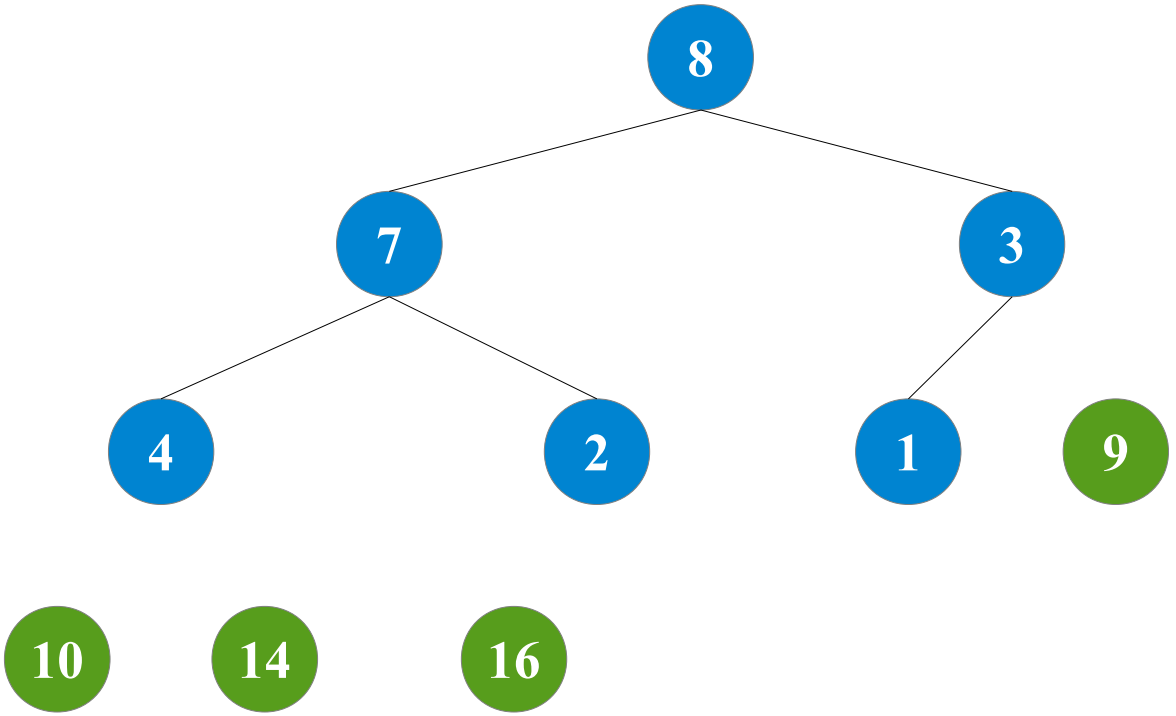
1	2	3	4	5	6	7	8	9	10
9	8	3	4	7	1	2	10	14	16

Heapsort



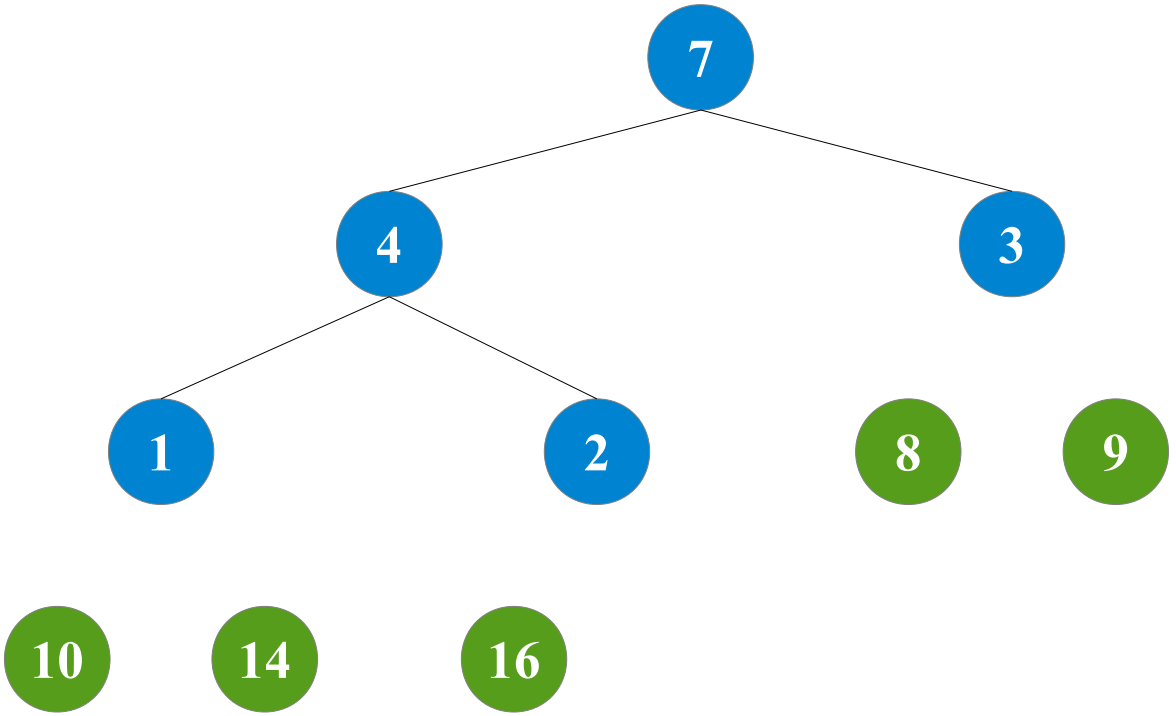
1	2	3	4	5	6	7	8	9	10
2	8	3	4	7	1	9	10	14	16

Heapsort



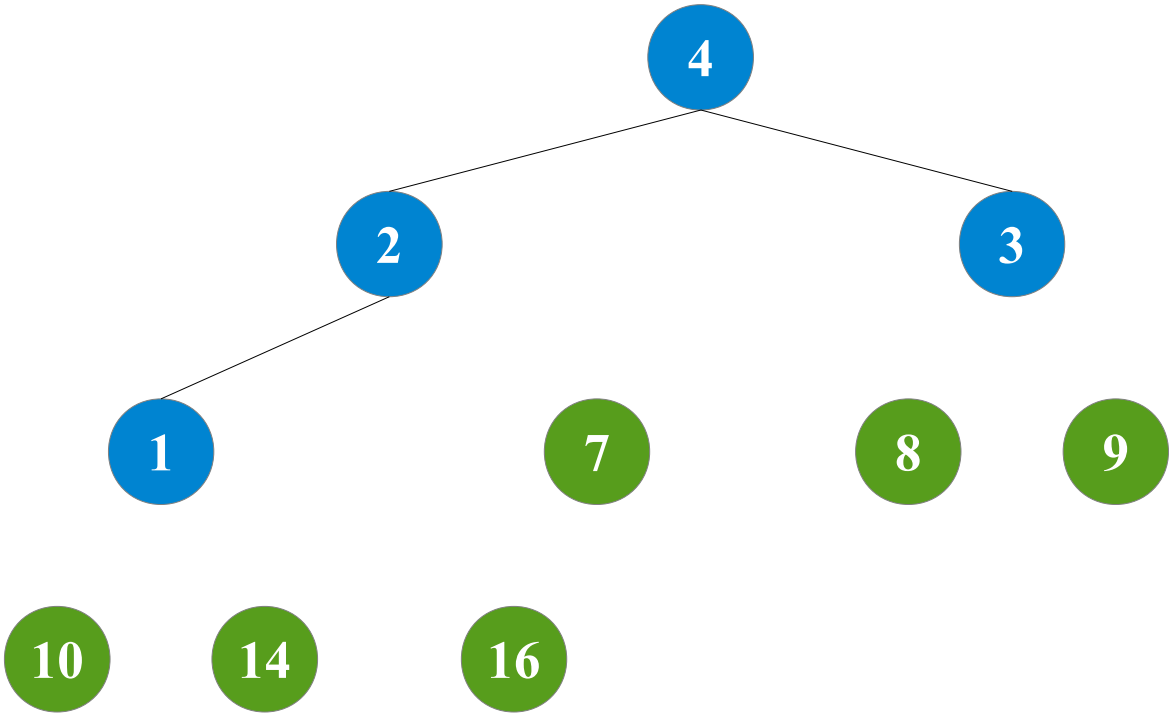
1	2	3	4	5	6	7	8	9	10
8	7	3	4	2	1	9	10	14	16

Heapsort



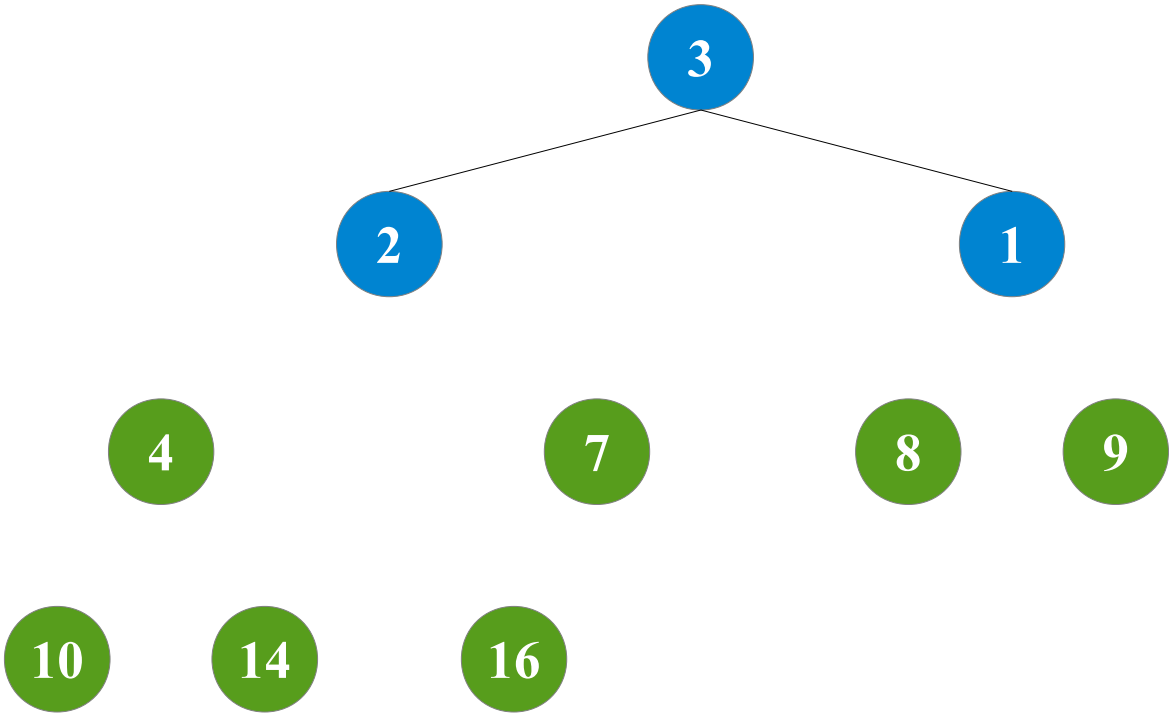
1	2	3	4	5	6	7	8	9	10
7	4	3	1	2	8	9	10	14	16

Heapsort



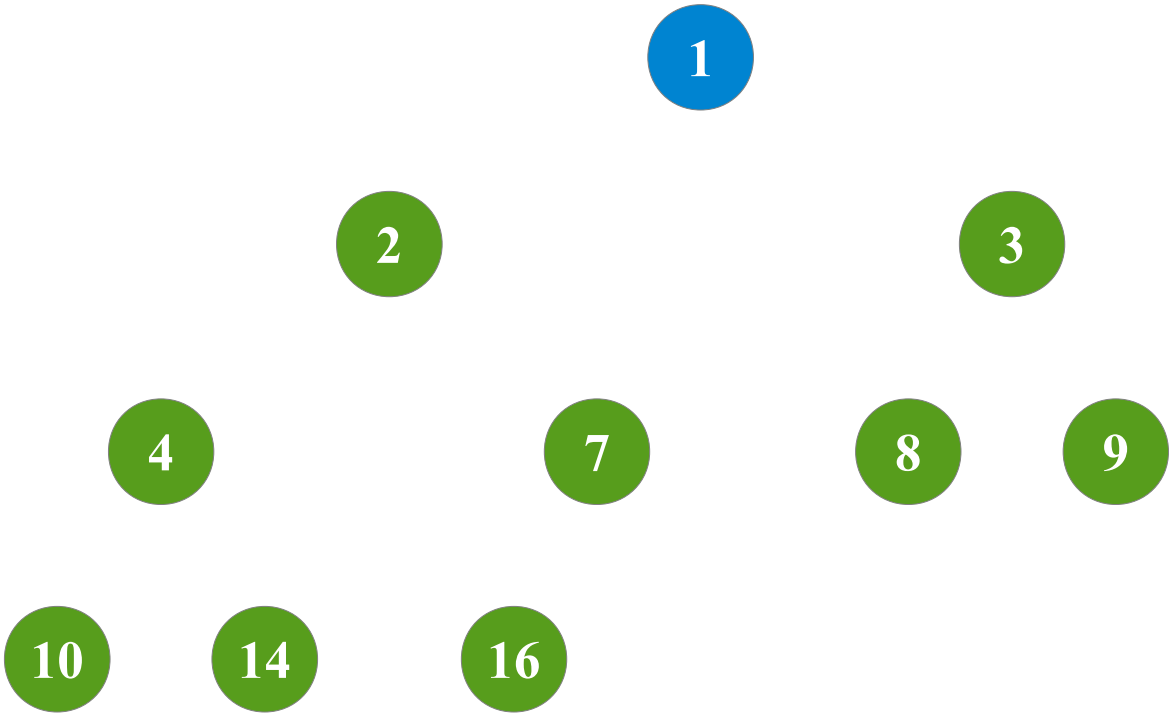
1	2	3	4	5	6	7	8	9	10
4	2	3	1	7	8	9	10	14	16

Heapsort



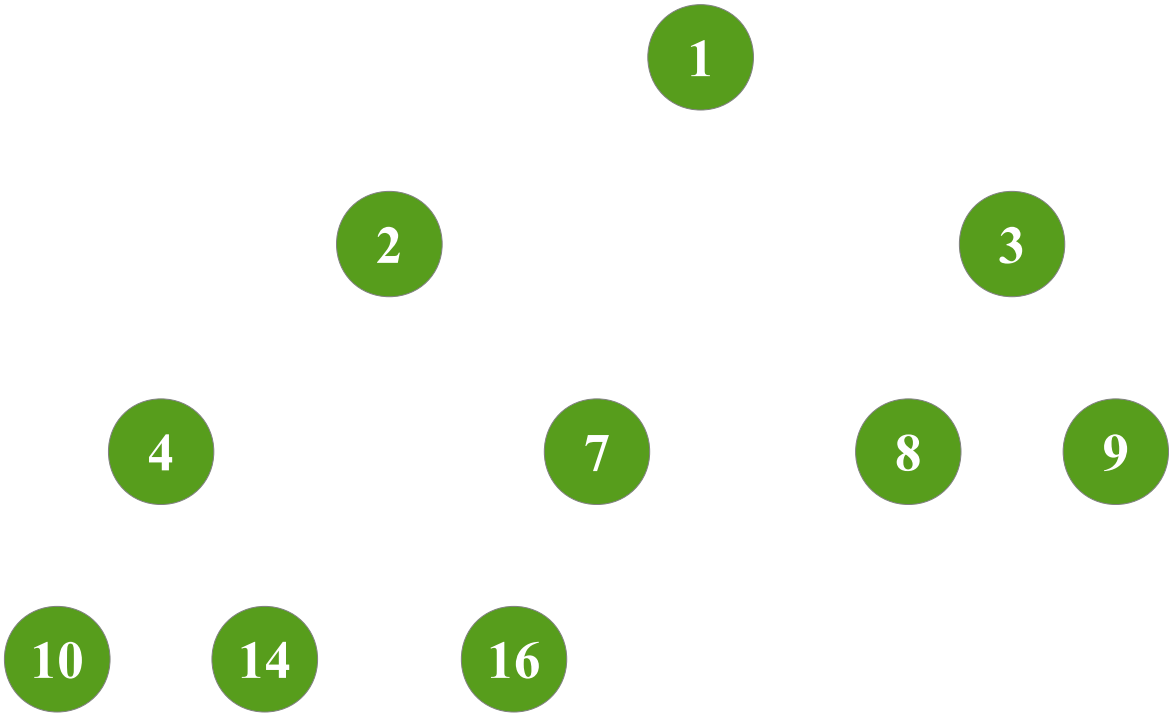
1	2	3	4	5	6	7	8	9	10
3	2	1	4	7	8	9	10	14	16

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

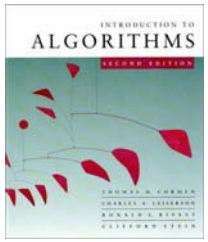
Heapsort Running time

- T(n) = $O(n)$ for Build-Max-Heap +
  n times $O(\lg n)$ for Max-Heapify
  = $O(n \lg n)$
- Constant 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)
```

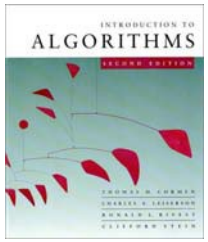
Priority Queues

- They are like FIFO (First-in First-out) Queues BUT depend on some priority value
- Elements 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).



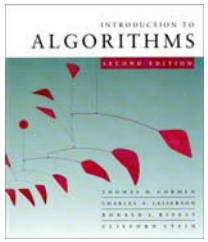
Divide and conquer

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.

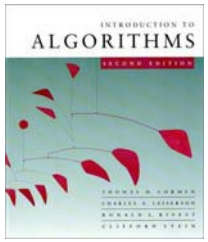
Key: *Linear-time partitioning subroutine.*



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$(A, 1, n)$

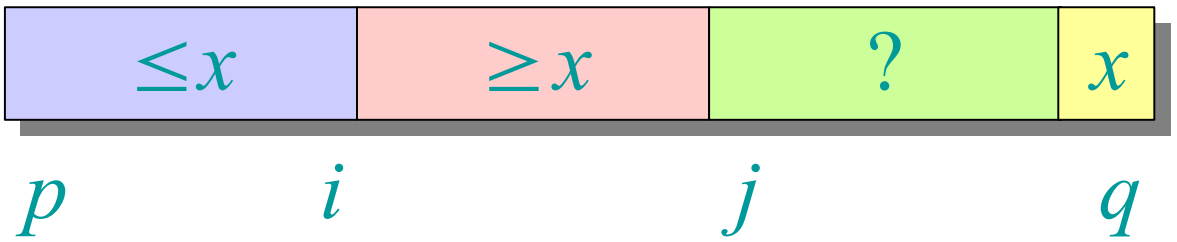


Partitioning subroutine

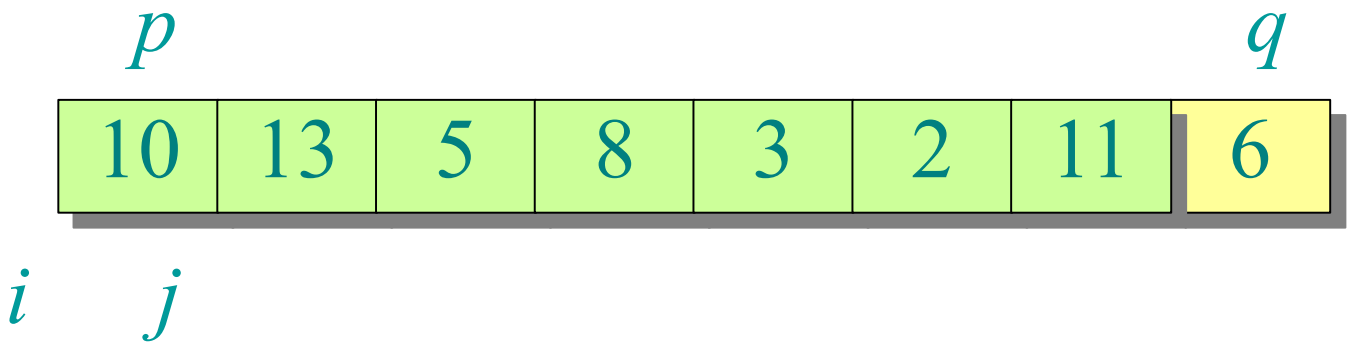
```
PARTITION(A, p, q)   ◁ A[p . . q]
    x ← A[q]         ◁ pivot = A[q]
    i ← p - 1
    for j ← p to q - 1
        do if A[j] ≤ x
                then  i ← i + 1
                      exchange A[i] ↔ A[j]
    exchange A[q] ↔ A[i+1]
    return i
```

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

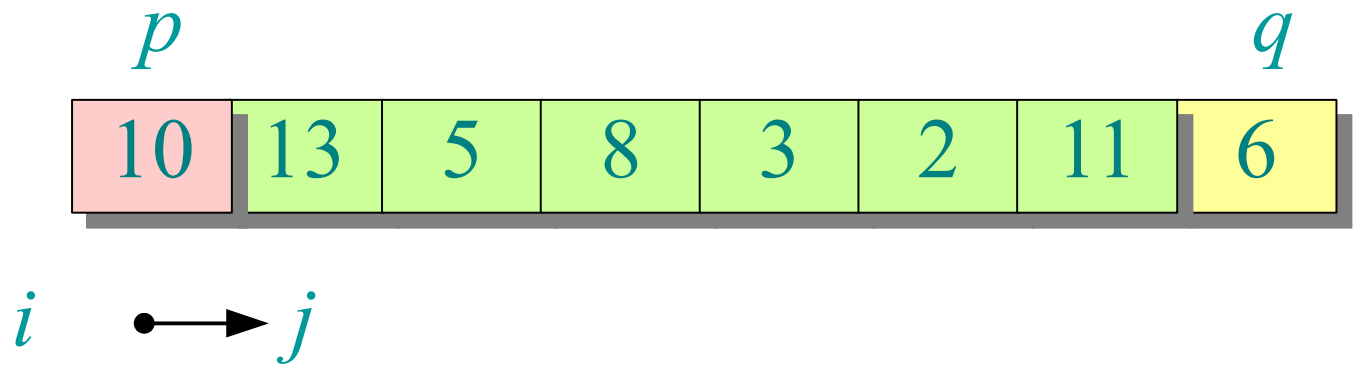
Invariant:



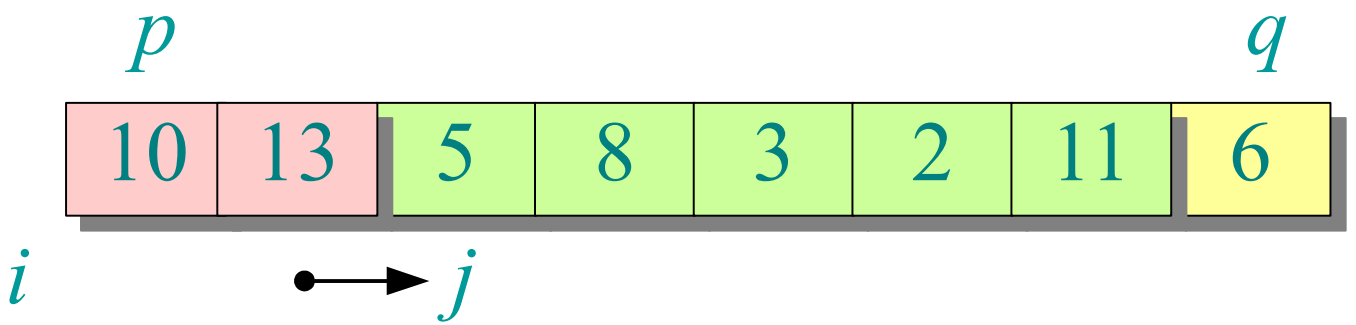
Example of Partitioning



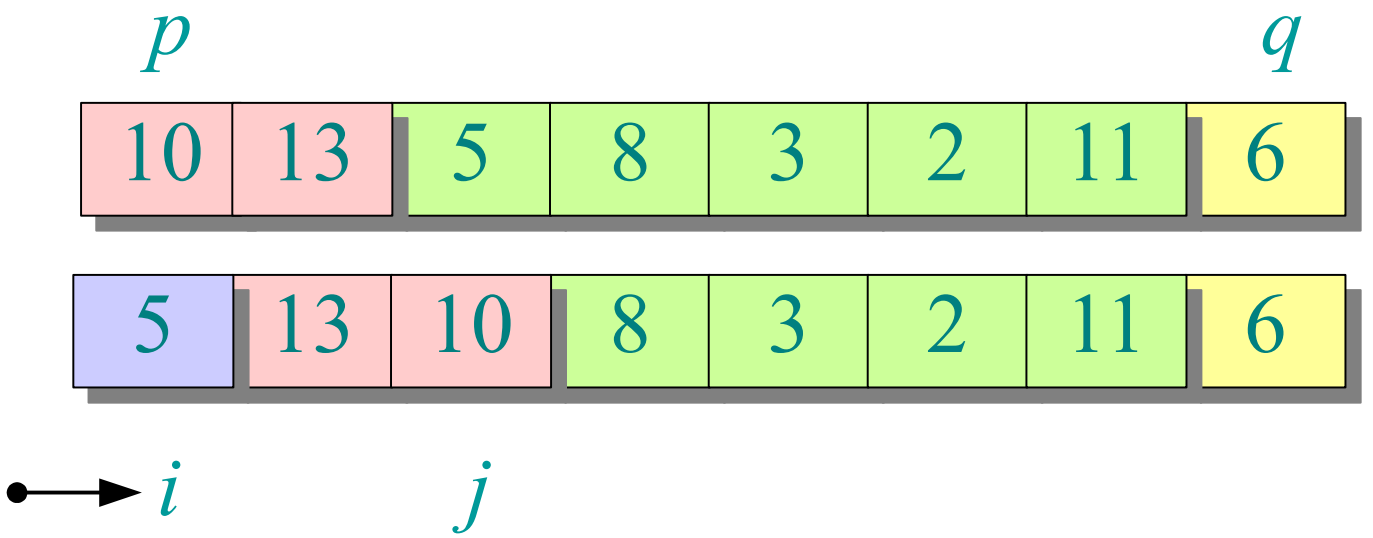
Example of Partitioning



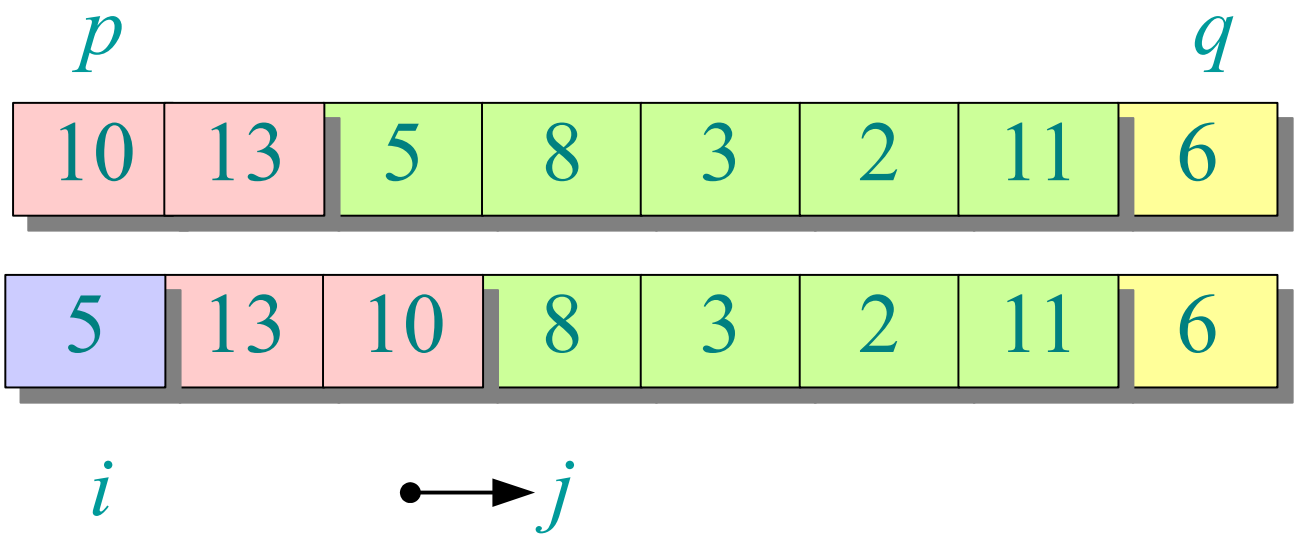
Example of Partitioning



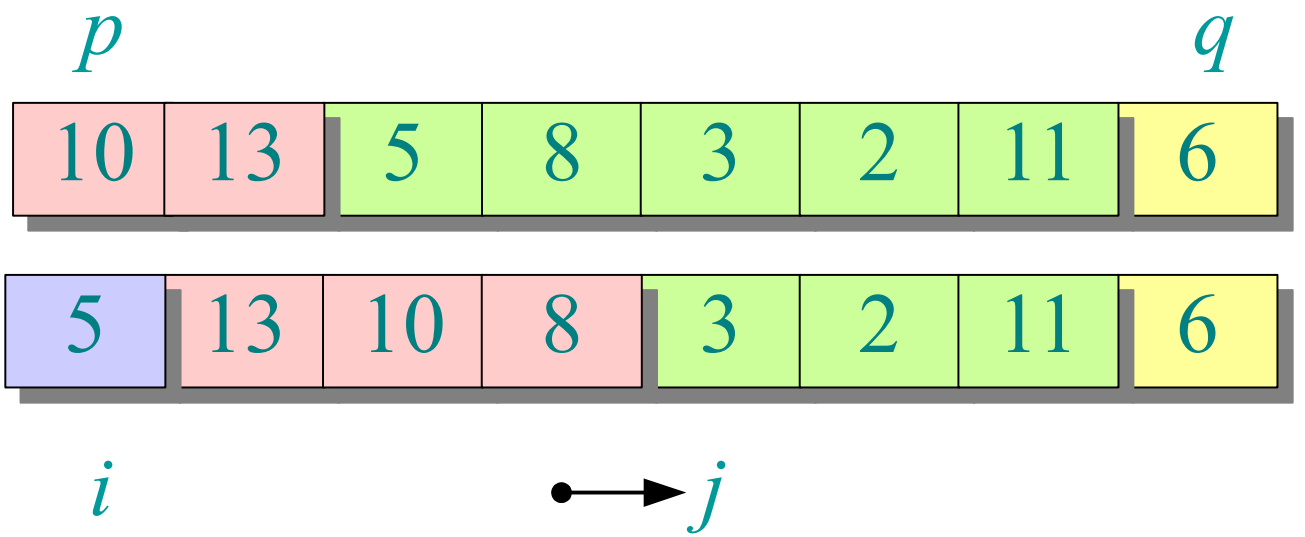
Example of Partitioning



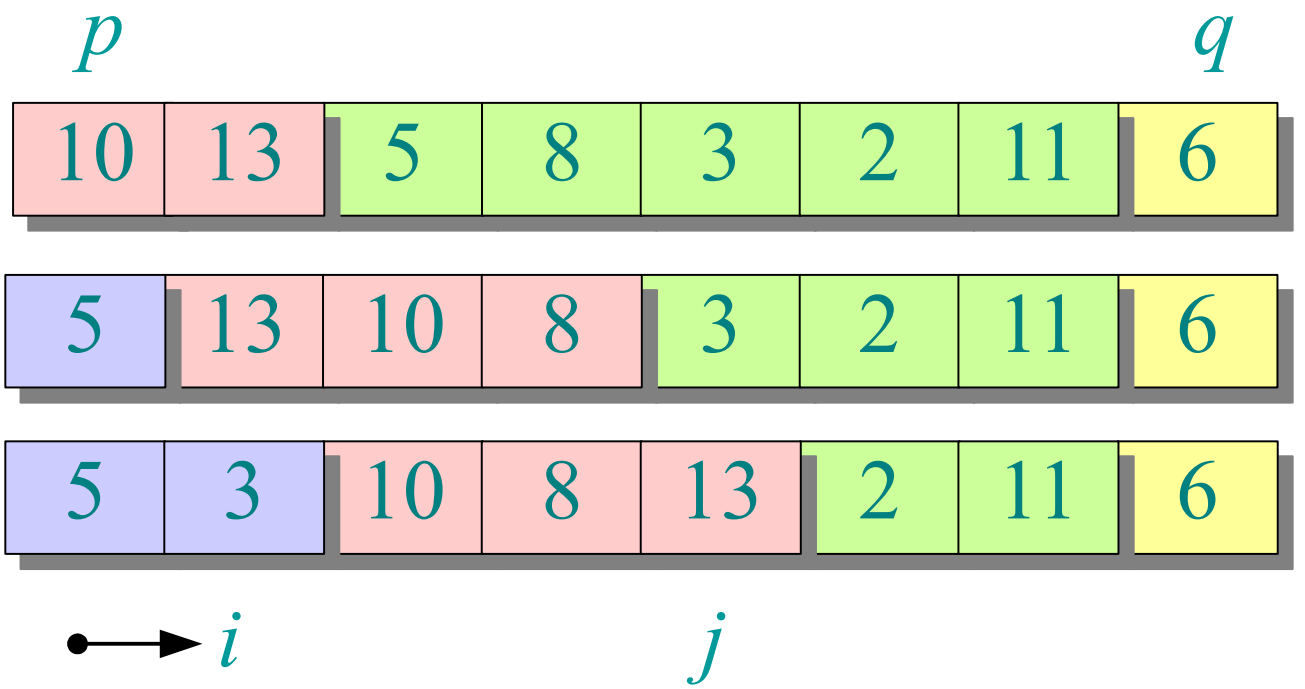
Example of Partitioning



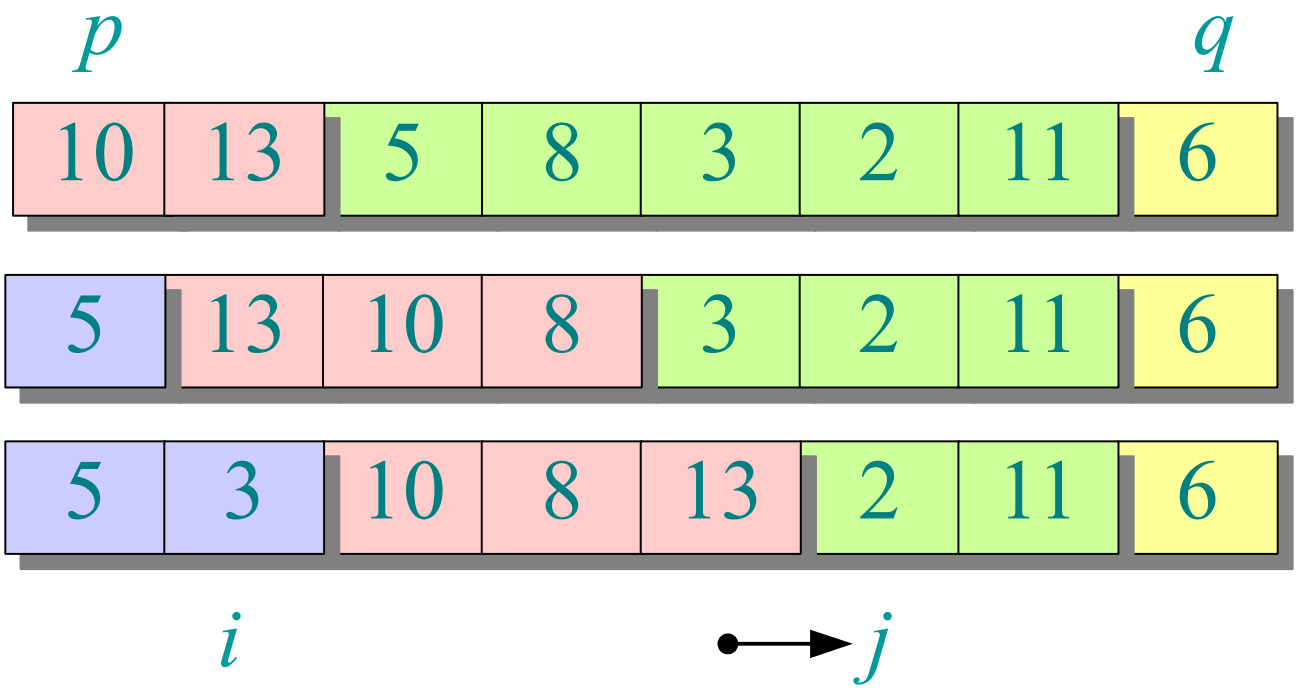
Example of Partitioning



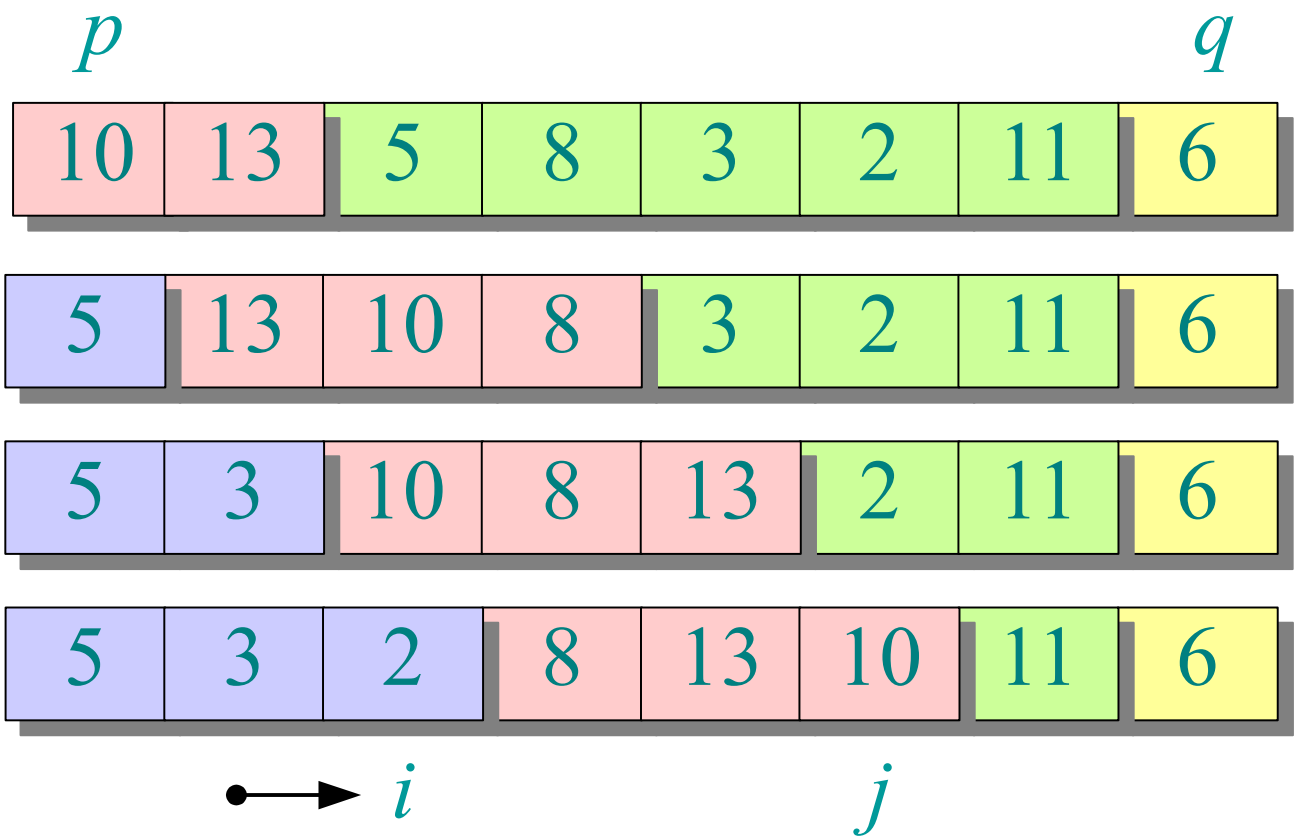
Example of Partitioning



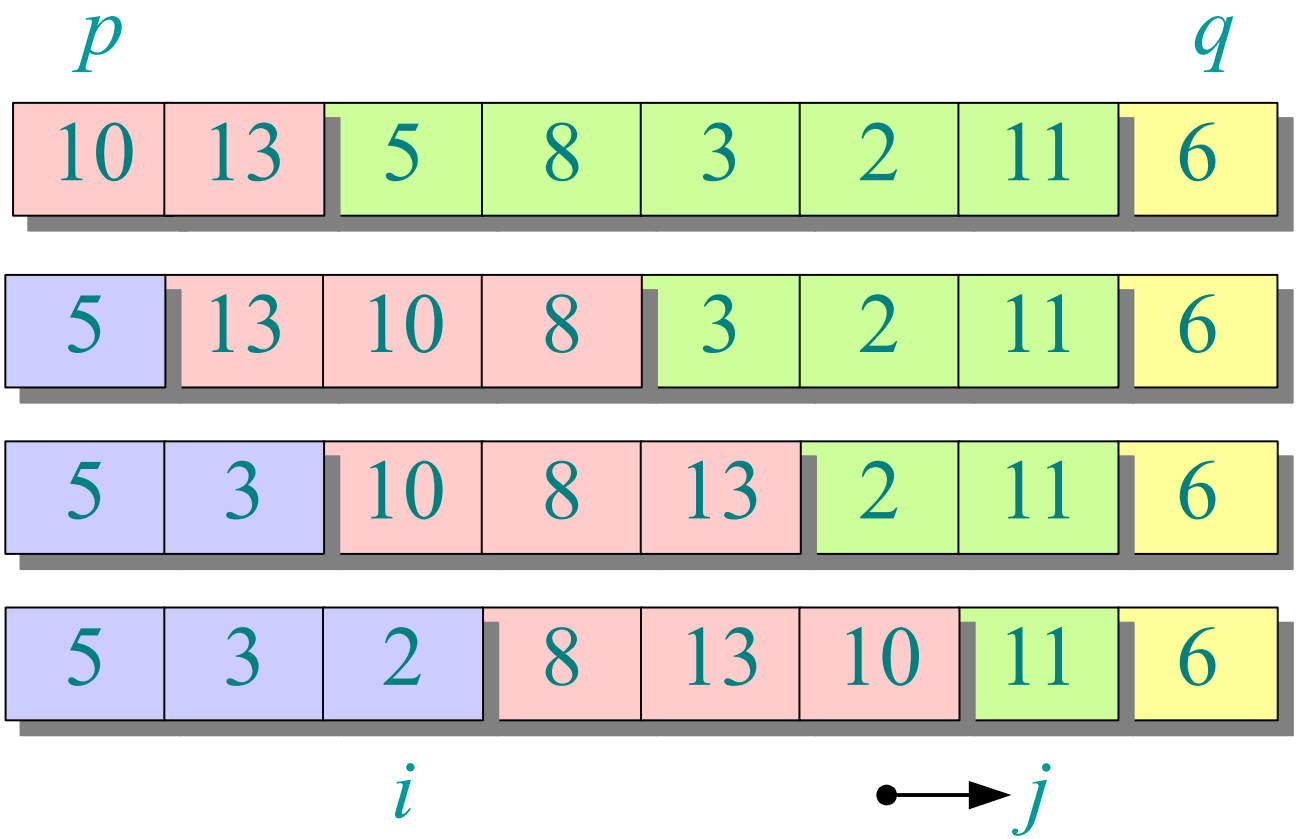
Example of Partitioning



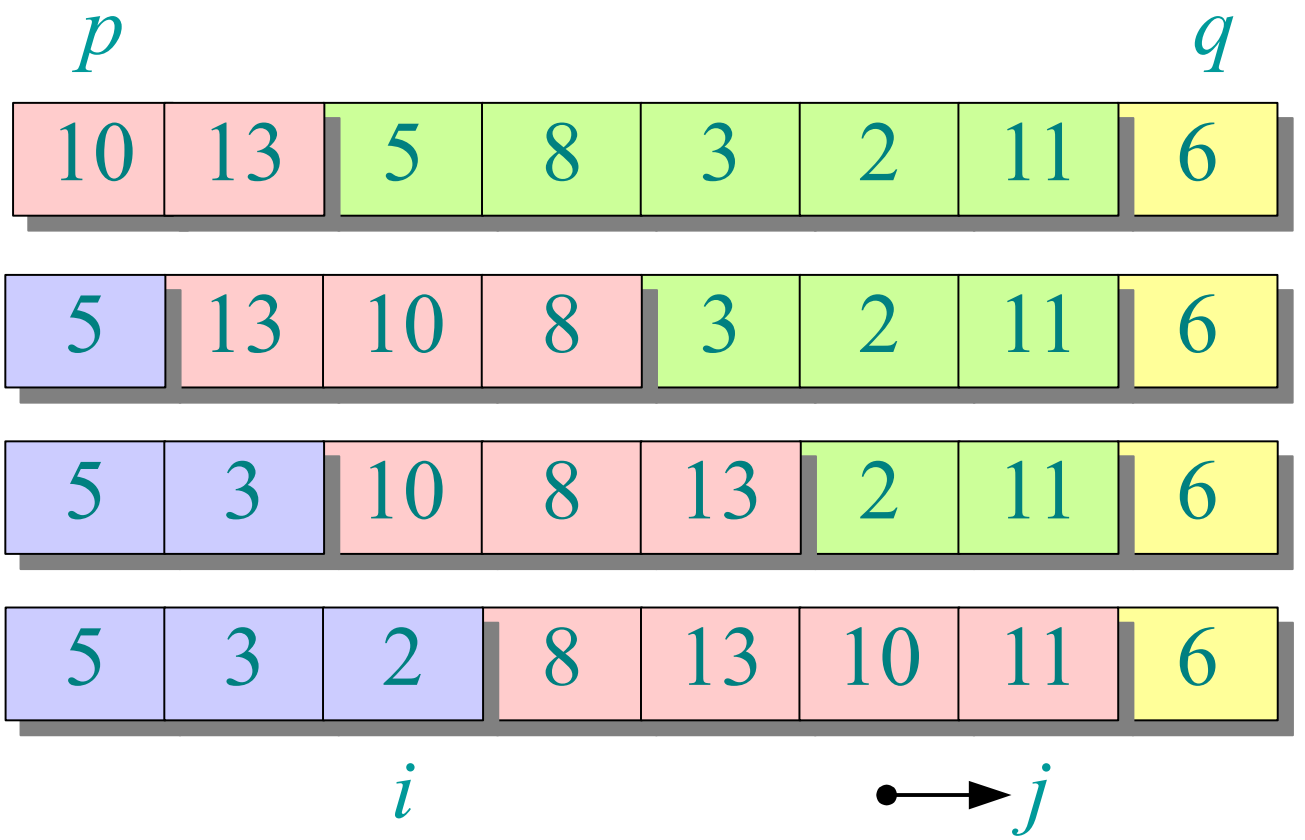
Example of Partitioning



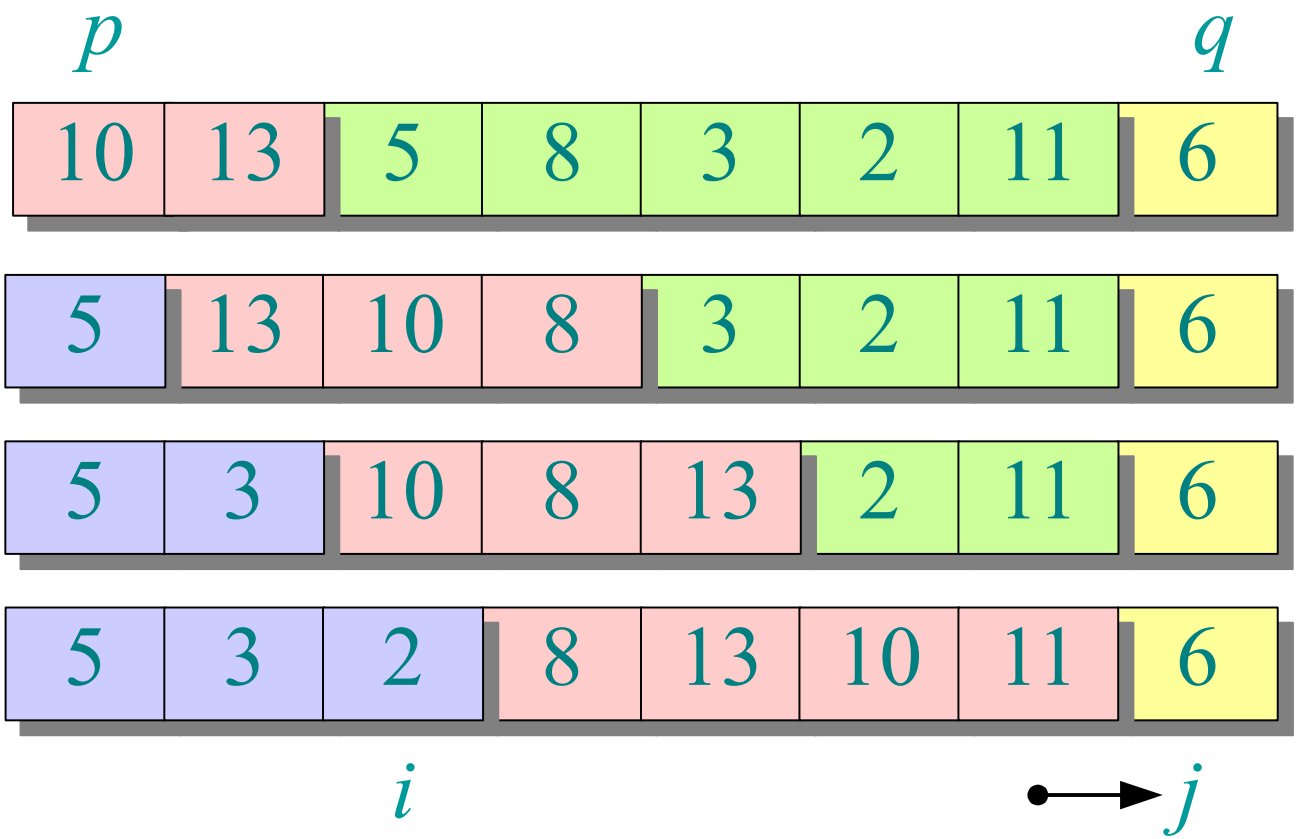
Example of Partitioning



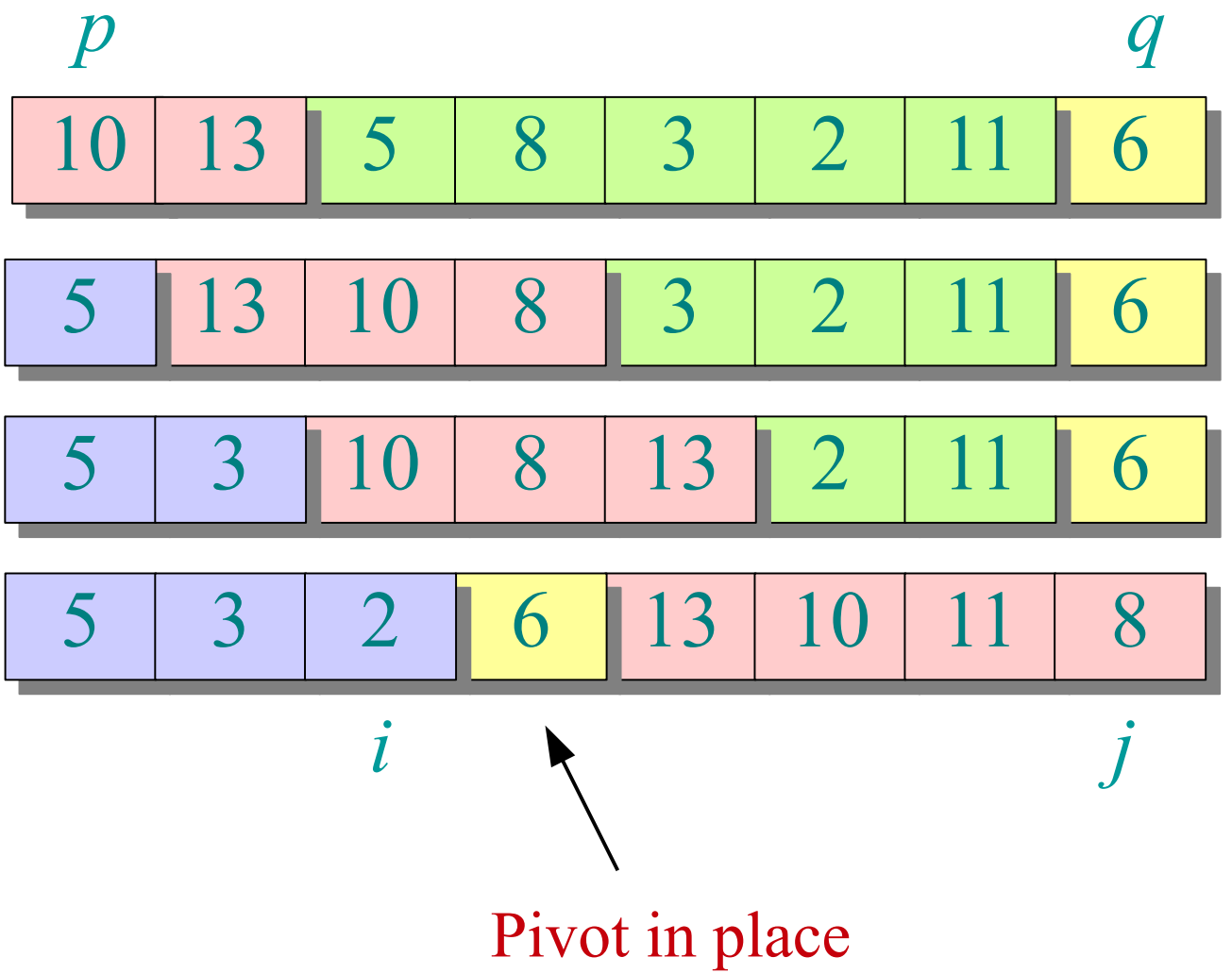
Example of Partitioning

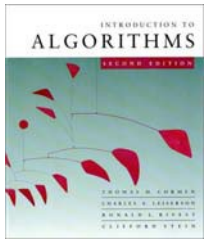


Example of Partitioning



Example of Partitioning

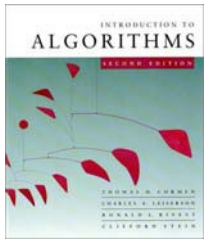




Pseudocode for quicksort

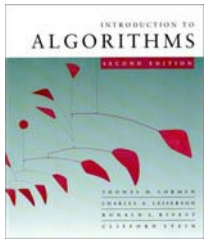
QUICKSORT(A, p, r)
  if $p < r$
    then $q \leftarrow$ PARTITION(A, p, r)
      QUICKSORT$(A, p, q-1)$
      QUICKSORT$(A, q+1, r)$

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



Analysis of quicksort

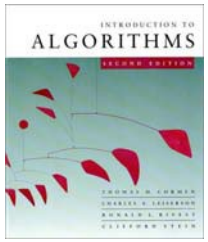
- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n)$ = worst-case running time on an array of n elements.



Worst-case of quicksort

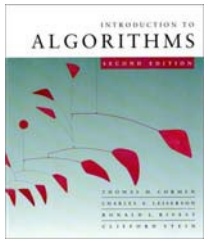
- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

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



Worst-case recursion tree

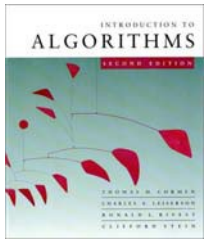
$$T(n) = T(0) + T(n-1) + cn$$



Worst-case recursion tree

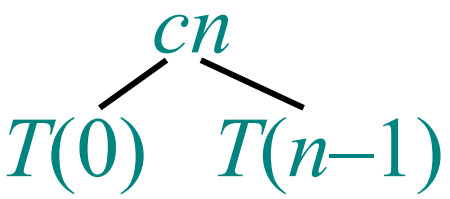
$$T(n) = T(0) + T(n-1) + cn$$

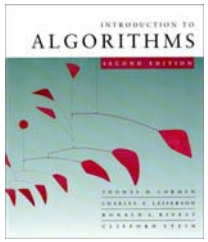
$T(n)$



Worst-case recursion tree

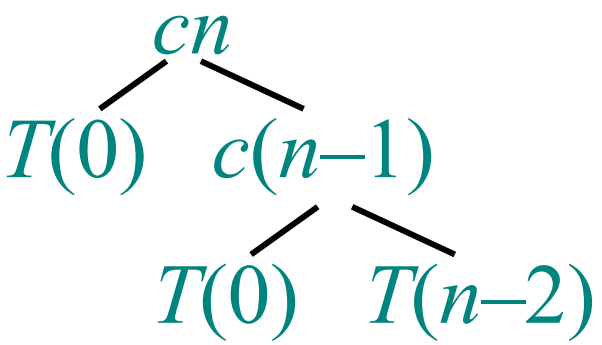
$$T(n) = T(0) + T(n-1) + cn$$

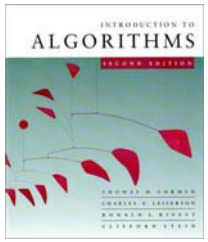




Worst-case recursion tree

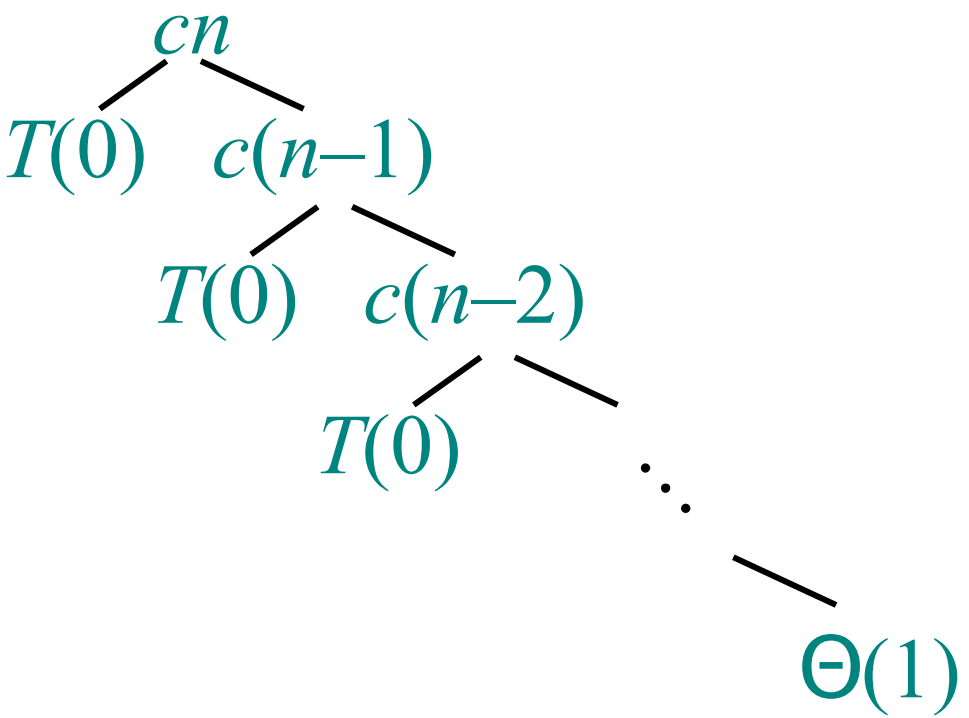
$$T(n) = T(0) + T(n–1) + cn$$

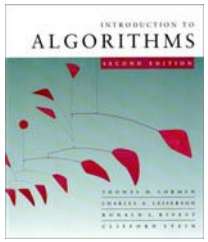




Worst-case recursion tree

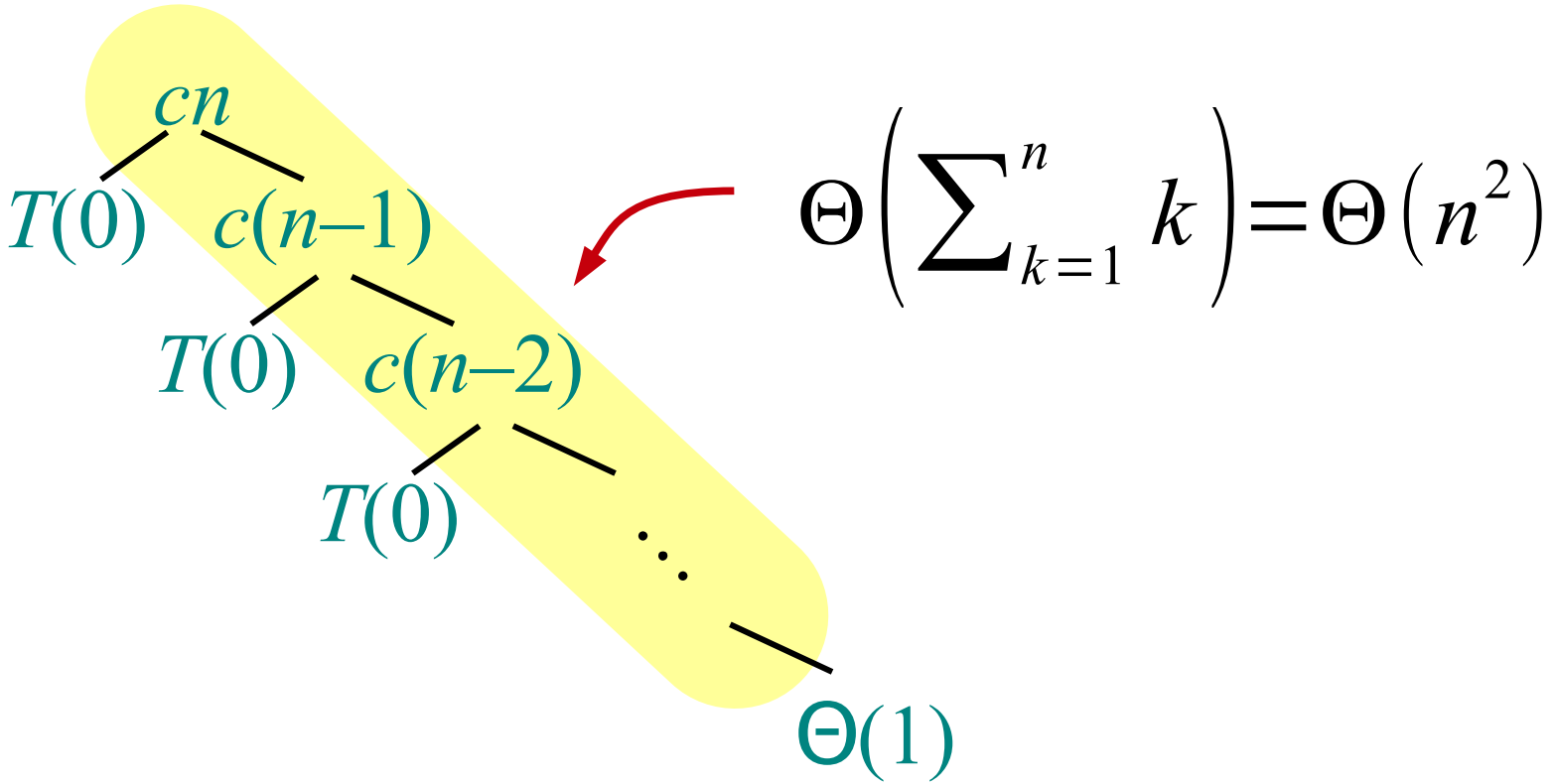
$$T(n) = T(0) + T(n-1) + cn$$



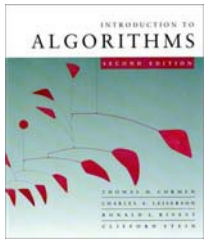


Worst-case recursion tree

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

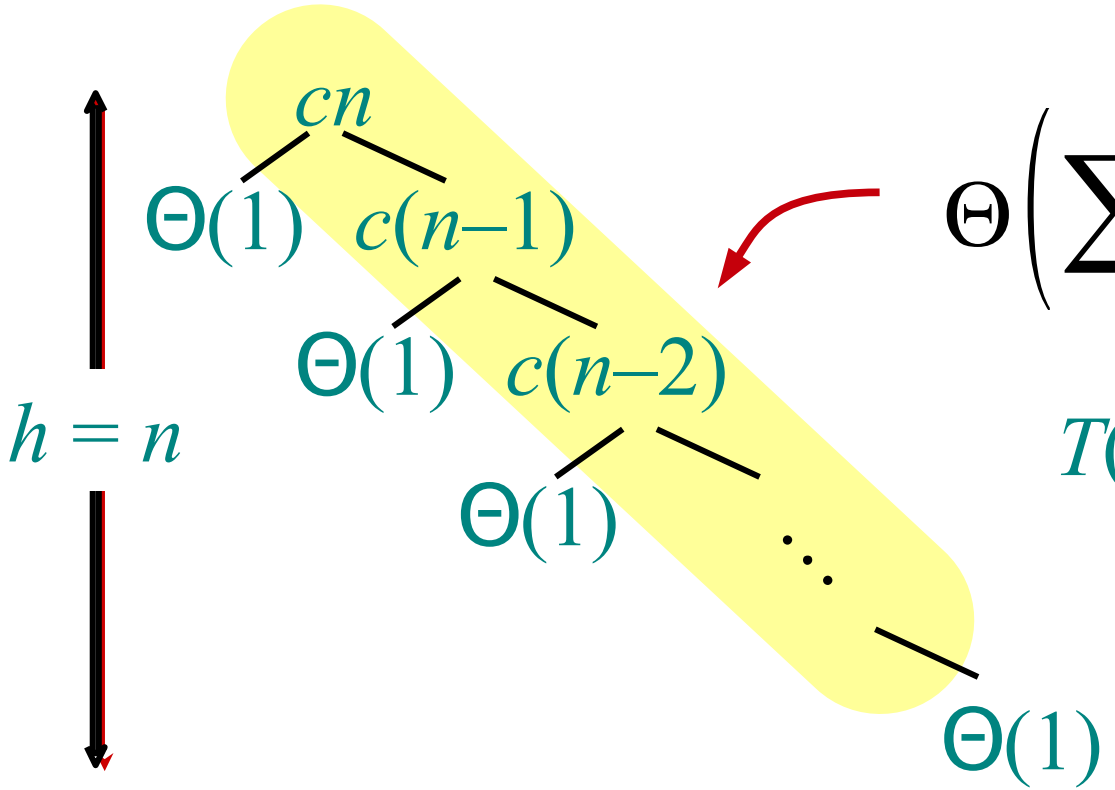


$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$



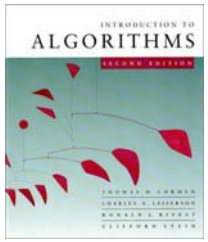
Worst-case recursion tree

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



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

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



Best-case analysis

(For intuition only!)

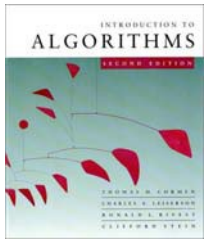
If we're lucky, PARTITION splits the array evenly:

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \lg n) \qquad \text{(same as merge sort)}$$

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

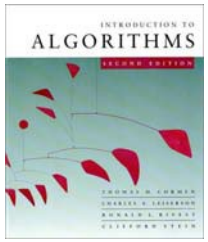
$$T(n) = T\left(\tfrac{1}{10} n\right) + T\left(\tfrac{9}{10} n\right) + \Theta(n)$$

What is the solution to this recurrence?

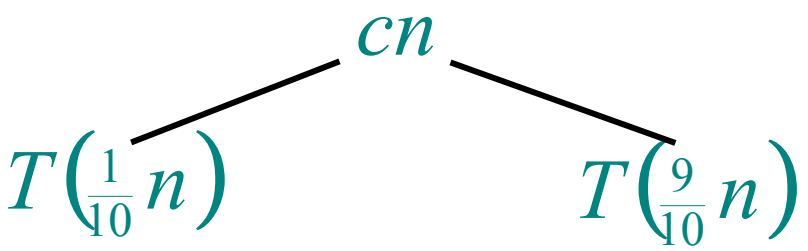


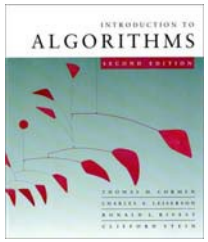
Analysis of "almost-best" case

$T(n)$

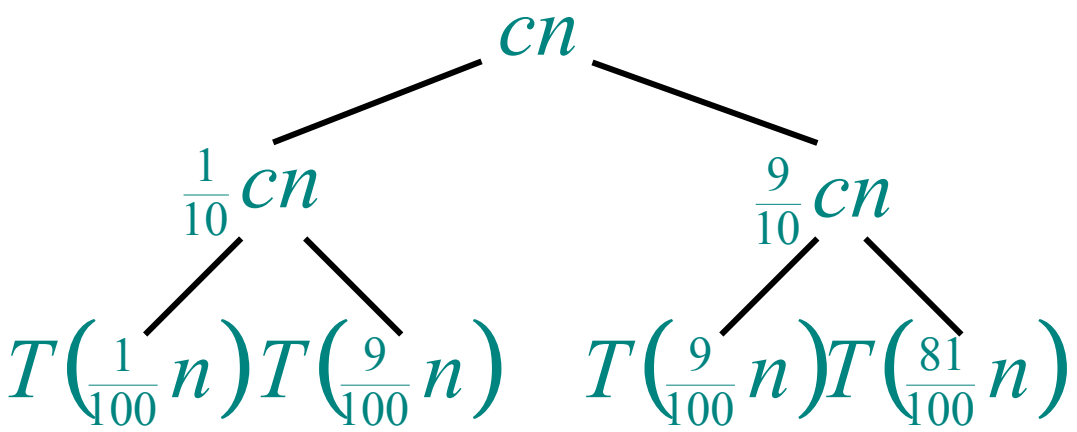


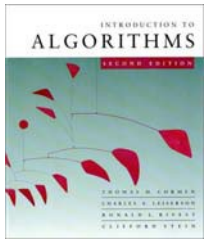
Analysis of "almost-best" case



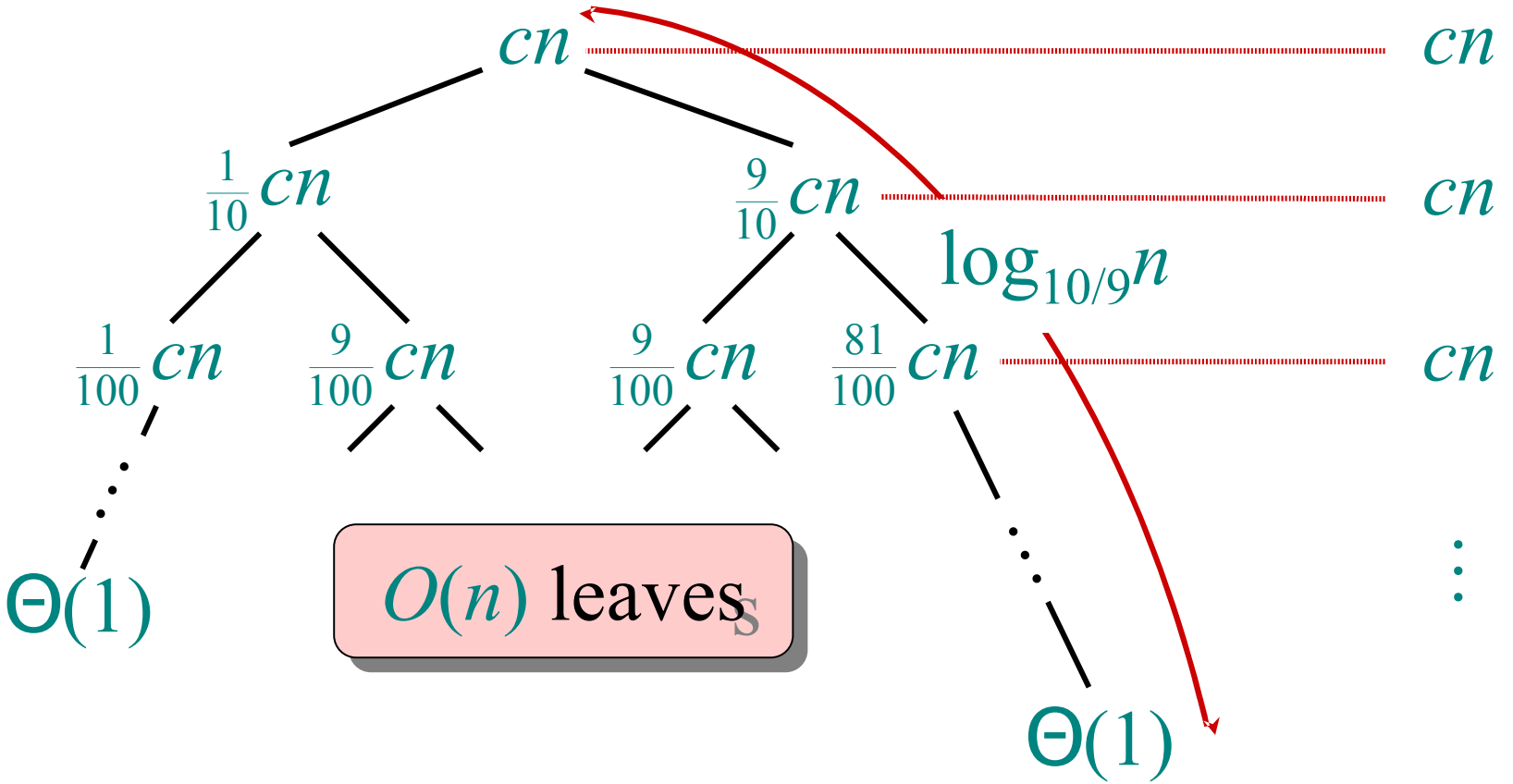


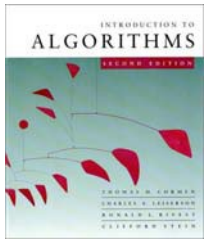
Analysis of "almost-best" case



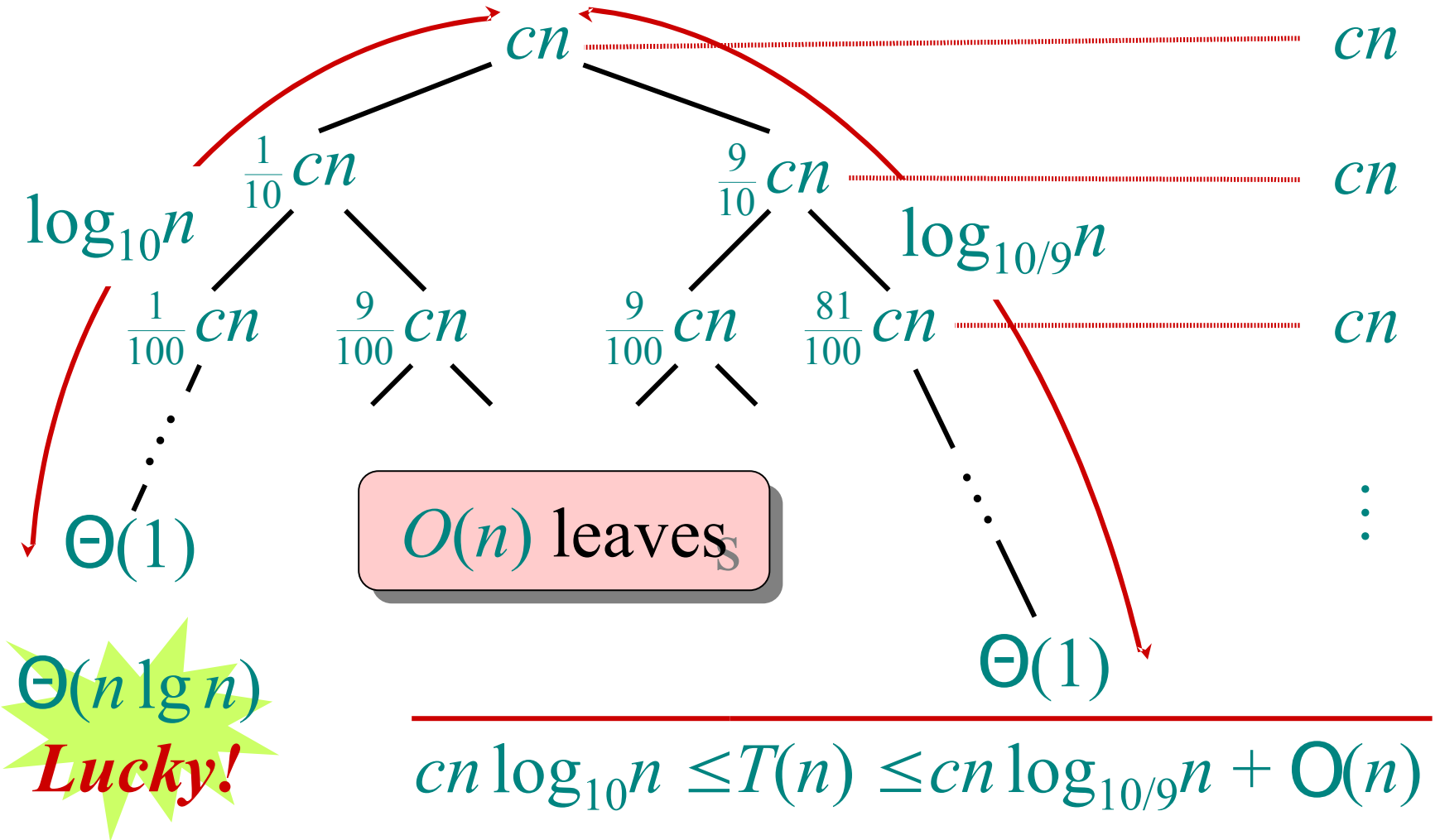


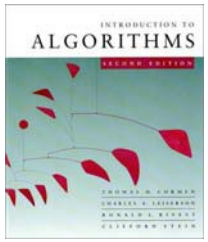
Analysis of "almost-best" case





Analysis of "almost-best" case





More intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ….

$L(n) = 2U(n/2) + \Theta(n)$ ***lucky***

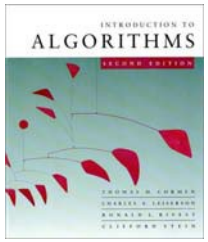
$U(n) = L(n-1) + \Theta(n)$ ***unlucky***

Solving:

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

Lucky!

How can we make sure we are usually lucky?



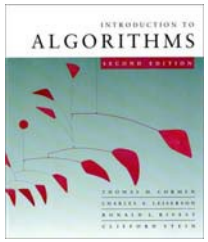
Randomized quicksort

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 = RANDOM(p, r)
2 exchange A[r] with A[i]
3 return PARTITION(A, p, r)
```



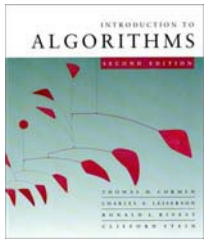
Randomized quicksort analysis

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

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

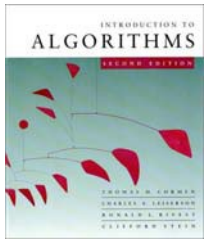
$E[X_k] = \Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.



Analysis (continued)

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

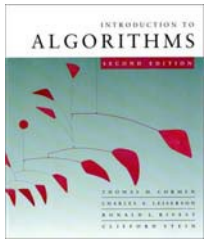
$$= \sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n) \right)$$



Calculating expectation

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

Take expectations of both sides.

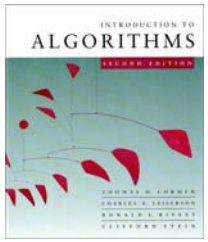


Calculating expectation

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

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

Linearity of expectation.



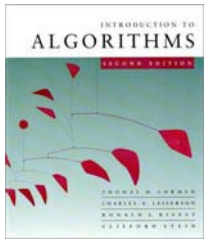
Calculating expectation

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

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

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

Independence of X_k from other random choices.



Calculating expectation

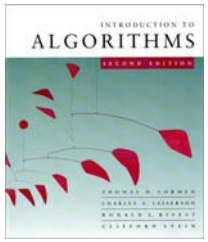
$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

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

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

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

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



Calculating expectation

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

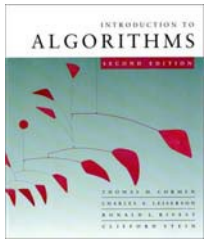
$$= \sum_{k=0}^{n-1} E\left[X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

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

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

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

Summations have identical terms.



Hairy recurrence

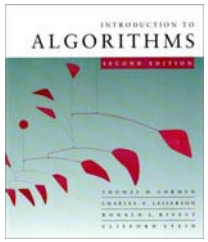
$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

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

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

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

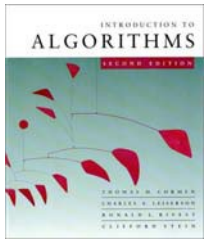
Use fact: $\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$ (exercise).



Substitution method

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

Substitute inductive hypothesis.

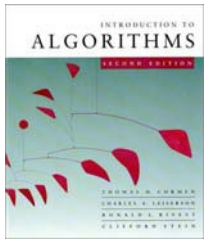


Substitution method

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

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

Use fact.



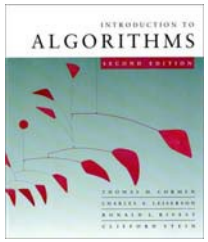
Substitution method

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

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

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

Express as ***desired – residual***.



Substitution method

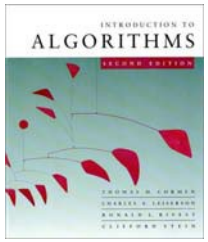
$$E[T(n)] \le \frac{2}{n}\sum_{k=2}^{n-1} ak \lg k + \Theta(n)$$

$$= \frac{2a}{n}\left[\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right)\right] + \Theta(n)$$

$$= an \lg n - \left[\left(\frac{an}{4} - \Theta(n)\right)\right]$$

$$\le an \lg n ,$$

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" → $\Theta(1)$ space
  - Worst case time $\Theta(n \lg n)$
- Quicksort
  - Sorts "in place" → $\Theta(1)$ 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