# Chapter 17 Sorting

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Objectives

- To study and analyze time efficiency of various sorting algorithms (§§17.2–17.7).
- To design, implement, and analyze insertion sort (§17.2).
- To design, implement, and analyze bubble sort (§17.3).
- To design, implement, and analyze merge sort (§17.4).
- To design, implement, and analyze quick sort (§17.5).
- To design and implement a heap (§17.6).
- To design, implement, and analyze heap sort (§17.6).
- To design, implement, and analyze bucket sort and radix sort (§17.7).

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# why study sorting?

Sorting is a classic subject in computer science. There are three reasons for studying sorting algorithms.

- First, sorting algorithms illustrate many creative approaches to problem solving and these approaches can be applied to solve other problems.
- Second, sorting algorithms are good for practicing fundamental programming techniques using selection statements, loops, methods, and arrays.
- Third, sorting algorithms are excellent examples to demonstrate algorithm performance.

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# what data to sort?

The data to be sorted might be integers, doubles, characters, or objects. For simplicity, this chapter assumes:

- data to be sorted are integers,
- data are stored in a list, and
- data are sorted in ascending order

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Insertion Sort

myList = [2, 9, 5, 4, 8, 1, 6] # Unsorted

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist. | 2 | 9 | 5 | 4 | 8 | 1 | 6 |
| Step2: The sorted sublist is [2, 9]. Insert 5 into the sublist. | 2 | 9 | 5 | 4 | 8 | 1 | 6 |
| Step 3: The sorted sublist is [2, 5, 9]. Insert 4 into the sublist. | 2 | 5 | 9 | 4 | 8 | 1 | 6 |
| Step 4: The sorted sublist is [2, 4, 5, 9]. Insert 8 into the sublist. | 2 | 4 | 5 | 9 | 8 | 1 | 6 |
| Step 5: The sorted sublist is [2, 4, 5, 8, 9]. Insert 1 into the sublist. | 2 | 4 | 5 | 8 | 9 | 1 | 6 |
| Step 6: The sorted sublist is [1, 2, 4, 5, 8, 9]. Insert 6 into the sublist. | 1 | 2 | 4 | 5 | 8 | 9 | 6 |
| Step 7: The entire list is now sorted | 1 | 2 | 4 | 5 | 6 | 8 | 9 |

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

animation

# Insertion Sort Animation

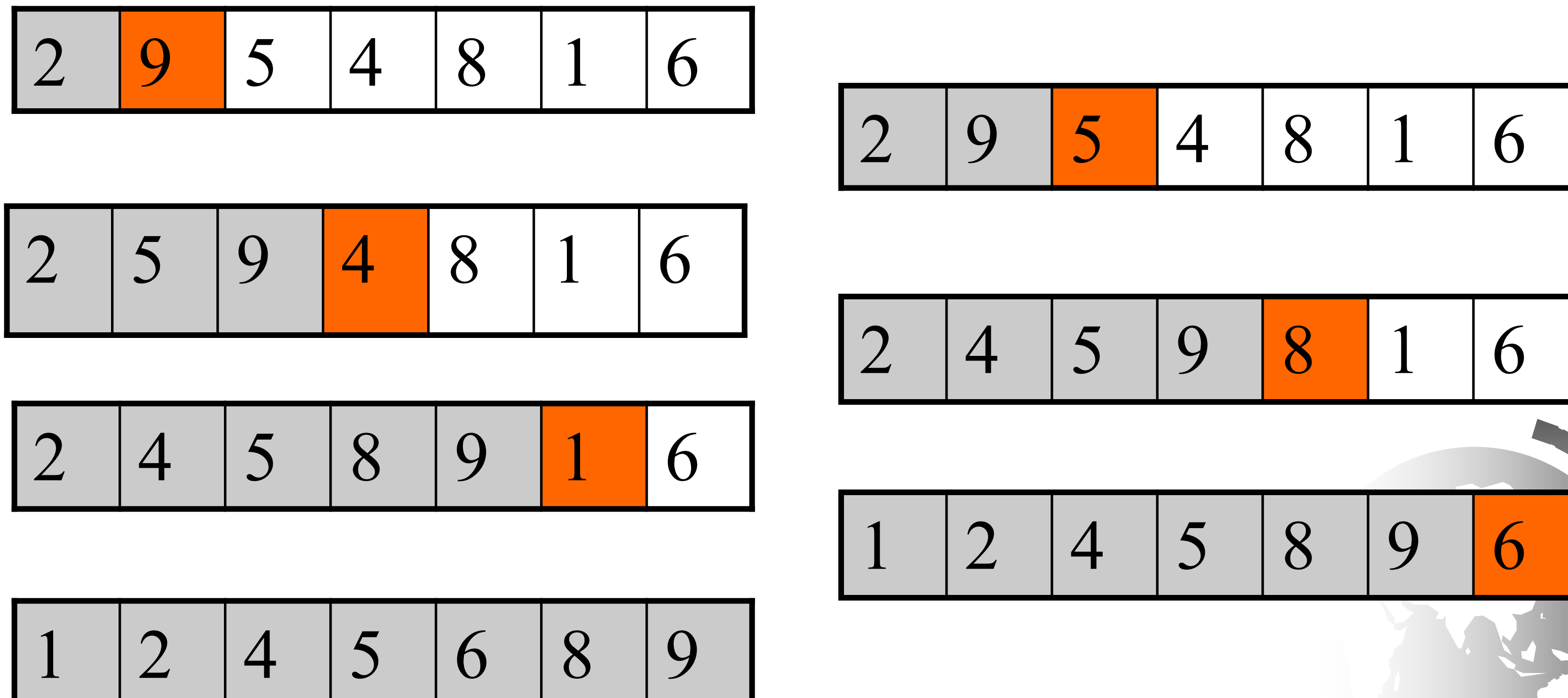
https://liveexample.pearsoncmg.com/dsanimation/InsertionSortNeweBook.html

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

animation

# Insertion Sort

myList = [2, 9, 5, 4, 8, 1, 6] # Unsorted




© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# How to Insert?

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | |
|---|---|---|---|---|---|---|---|---|
| list | 2 | 5 | 9 | 4 | | | | Step 1: Save 4 to a temporary variable currentElement |

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | |
|---|---|---|---|---|---|---|---|---|
| list | 2 | 5 | | 9 | | | | Step 2: Move list[2] to list[3] |

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | |
|---|---|---|---|---|---|---|---|---|
| list | 2 | | 5 | 9 | | | | Step 3: Move list[1] to list[2] |

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | |
|---|---|---|---|---|---|---|---|---|
| list | 2 | 4 | 5 | 9 | | | | Step 4: Assign currentElement to list[1] |

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# From Idea to Solution

```
for i in range(1, len(lst)):
    insert lst[i] into a sorted sublist lst[0..i-1] so that
    lst[0..i] is sorted.
```

```
lst[0]

lst[0] lst[1]

lst[0] lst[1] lst[2]

lst[0] lst[1] lst[2] lst[3]

lst[0] lst[1] lst[2] lst[3] ...
```

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# From Idea to Solution

```
for i in range(1, len(lst)):
    insert lst[i] into a sorted sublist lst[0..i-1] so that
    lst[0..i] is sorted.
```

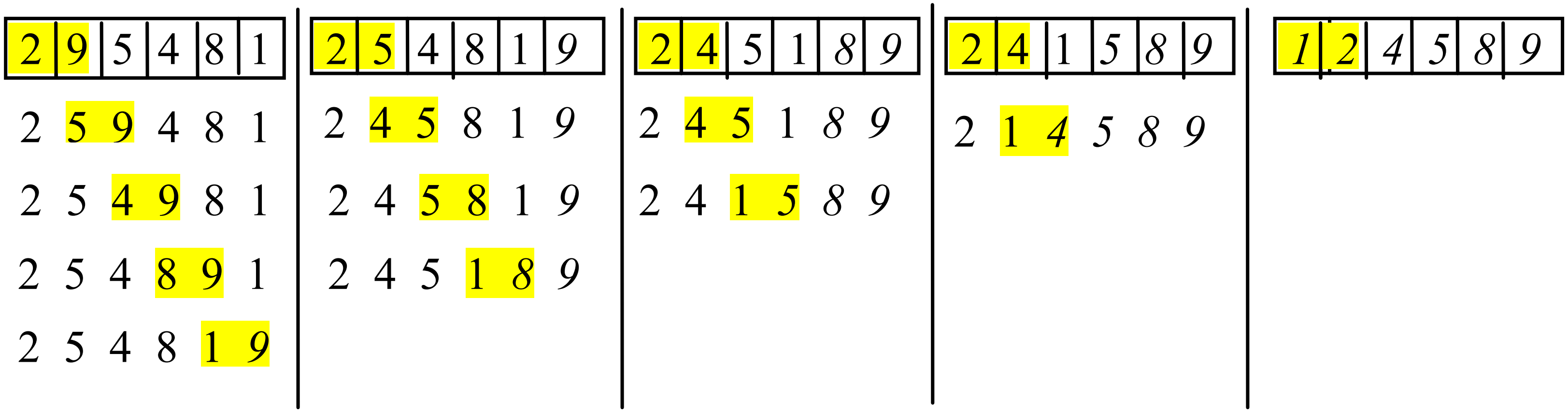
## Expand

```
k = i - 1
while k >= 0 and lst[k] > currentElement:
           lst[k + 1] = lst[k]
           k -= 1
# Insert the current element into lst[k + 1]
lst[k + 1] = currentElement
```

InsertionSort

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Bubble Sort




(a) 1st pass (b) 2nd pass (c) 3rd pass (d) 4th pass (e) 5th pass

Bubble sort time: $O(n^2)$

$$(n-1) + (n-2) + ... + 2 + 1 = \frac{n^2}{2} - \frac{n}{2}$$

BubbleSort

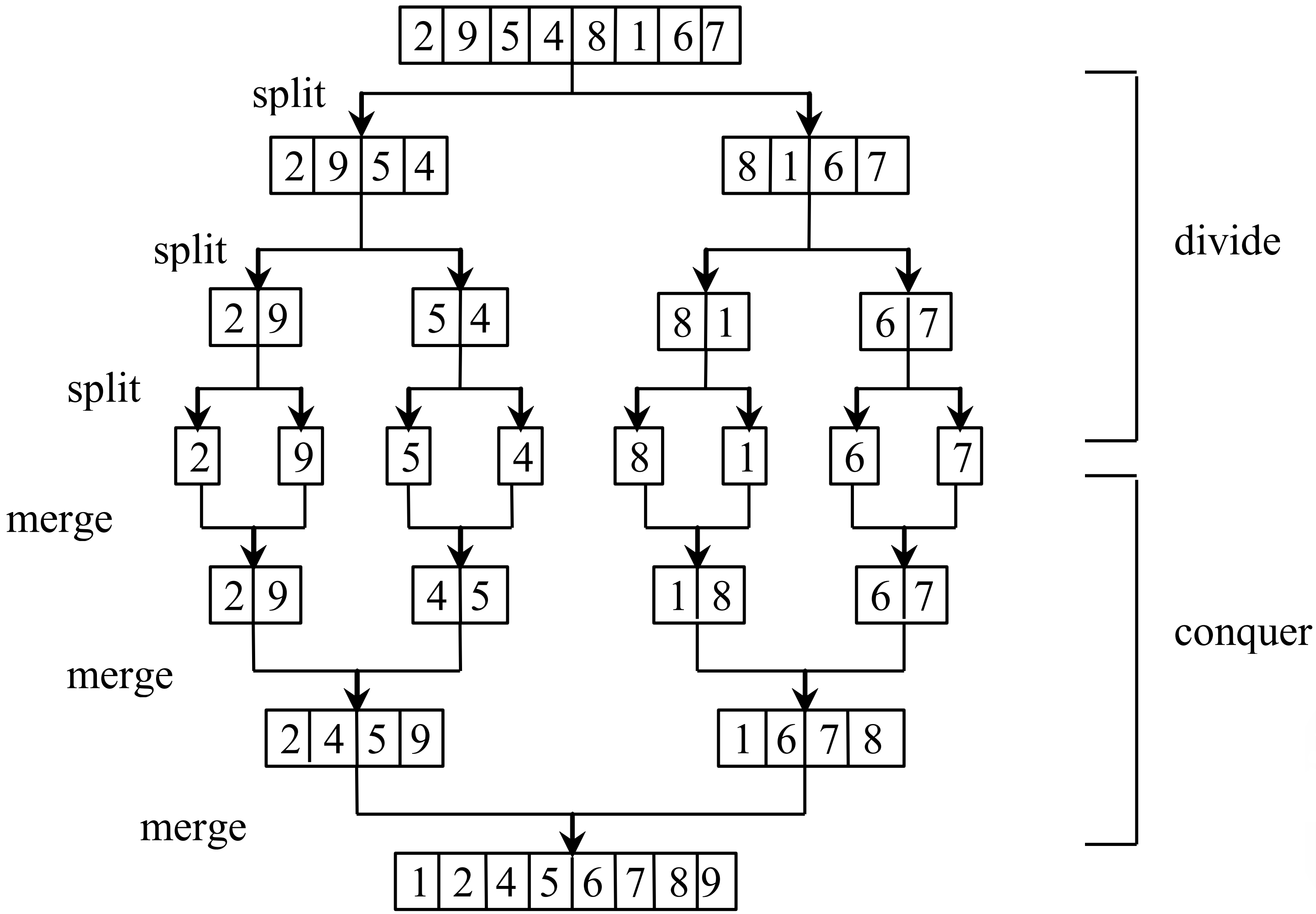
© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Bubble Sort Animation

https://liveexample.pearsoncmg.com/dsanimation/BubbleSortNeweBook.html

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Merge Sort




MergeSort

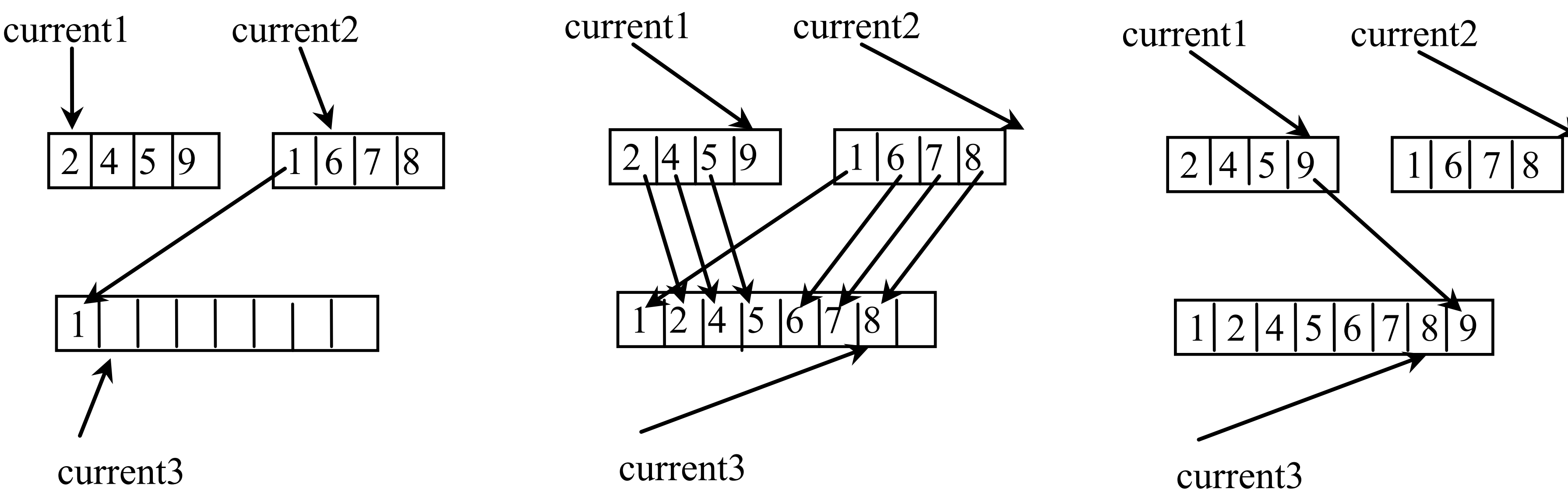
© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Merge Sort

```
mergeSort(list):
    firstHalf = mergeSort(firstHalf);
    secondHalf = mergeSort(secondHalf);
    list = merge(firstHalf, secondHalf);
```

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Merge Two Sorted Lists




(a) After moving 1 to temp

(b) After moving all the elements in list2 to temp

(c) After moving 9 to temp

Animation for Merging Two Sorted Lists

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Merge Sort Time

Let *T(n)* denote the time required for sorting an array of *n* elements using merge sort. Without loss of generality, assume *n* is a power of 2. The merge sort algorithm splits the array into two subarrays, sorts the subarrays using the same algorithm recursively, and then merges the subarrays. So,

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + mergetime$$

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Merge Sort Time

The first *T(n/2)* is the time for sorting the first half of the array and the second *T(n/2)* is the time for sorting the second half. To merge two subarrays, it takes at most *n-1* comparisons to compare the elements from the two subarrays and *n* moves to move elements to the temporary array. So, the total time is *2n-1*. Therefore,

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + 2n - 1 = 2(2T(\frac{n}{4}) + 2\frac{n}{2} - 1) + 2n - 1 = 2^2 T(\frac{n}{2^2}) + 2n - 2 + 2n - 1 \\
&= 2^k T(\frac{n}{2^k}) + 2n - 2^{k-1} + \ldots + 2n - 2 + 2n - 1 \\
&= 2^{\log n} T(\frac{n}{2^{\log n}}) + 2n - 2^{\log n - 1} + \ldots + 2n - 2 + 2n - 1 \\
&= n + 2n\log n - 2^{\log n} + 1 = 2n\log n + 1 = O(n\log n)
\end{aligned}
$$

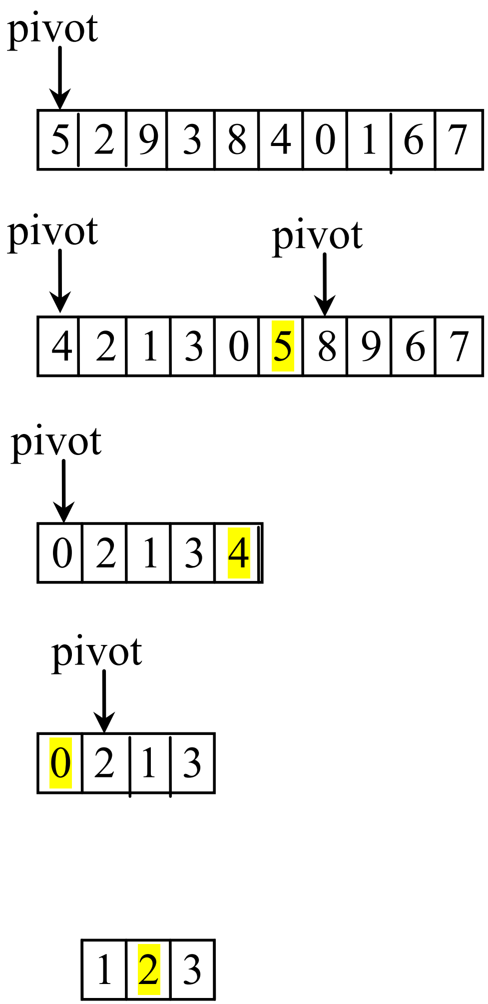
© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Quick Sort

Quick sort, developed by C. A. R. Hoare (1962), works as follows: The algorithm selects an element, called the *pivot*, in the array. Divide the array into two parts such that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than the pivot. Recursively apply the quick sort algorithm to the first part and then the second part.

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Quick Sort




(a) The original array

(b)The original array is partitioned

(c) The partial array (4 2 1 3 0) is partitioned

(d) The partial array (0 2 1 3) is partitioned

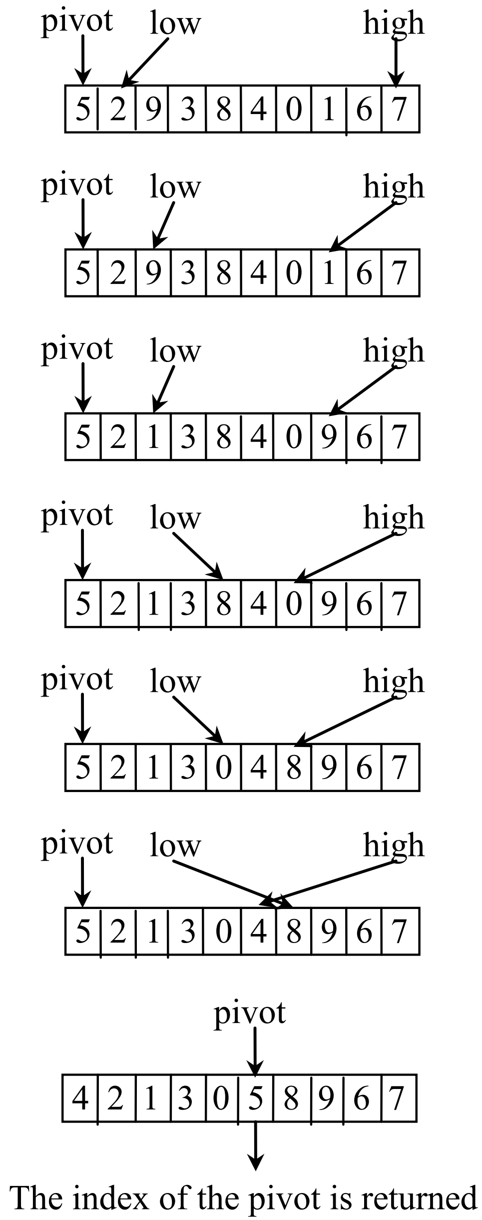
(e) The partial array (2 1 3) is partitioned

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Partition

Animation for partition

QuickSort




(a) Initialize pivot, low, and high

(b) Search forward and backward

(c) 9 is swapped with 1

(d) Continue search

(e) 8 is swapped with 0

(f) when high < low, search is over

(g) pivot is in the right place

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Quick Sort Time

To partition an array of $n$ elements, it takes $n\text{-}1$ comparisons and $n$ moves in the worst case. So, the time required for partition is $O(n)$.

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Worst-Case Time

In the worst case, each time the pivot divides the array into one big subarray with the other empty. The size of the big subarray is one less than the one before divided. The algorithm requires $O(n^2)$ time:

$$(n-1)+(n-2)+...+2+1=O(n^2)$$

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Best-Case Time

In the best case, each time the pivot divides the array into two parts of about the same size. Let *T(n)* denote the time required for sorting an array of elements using quick sort. So,

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + n = O(n \log n)$$

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Average-Case Time

On the average, each time the pivot will not divide the array into two parts of the same size nor one empty part. Statistically, the sizes of the two parts are very close. So the average time is *O(nlogn)*. The exact average-case analysis is beyond the scope of this book.

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Heap

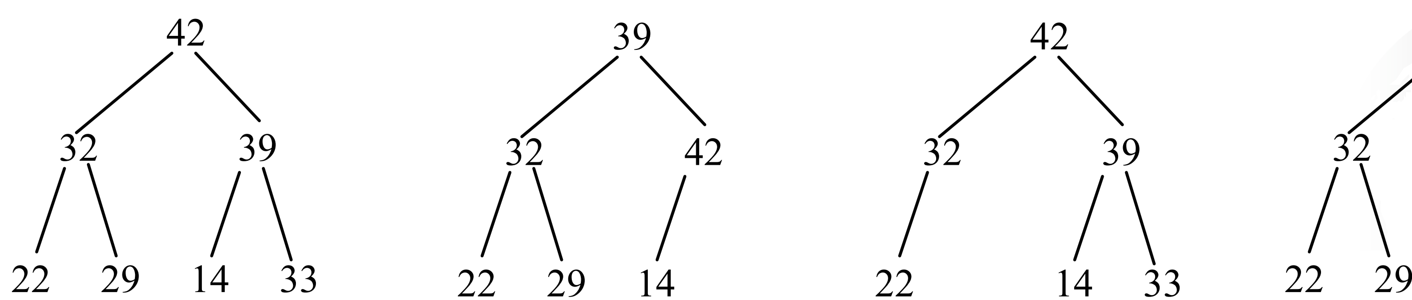
Heap is a useful data structure for designing efficient sorting algorithms and priority queues. A *heap* is a binary tree with the following properties:

- ✦It is a complete binary tree.
- ✦Each node is greater than or equal to any of its children.

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Complete Binary Tree

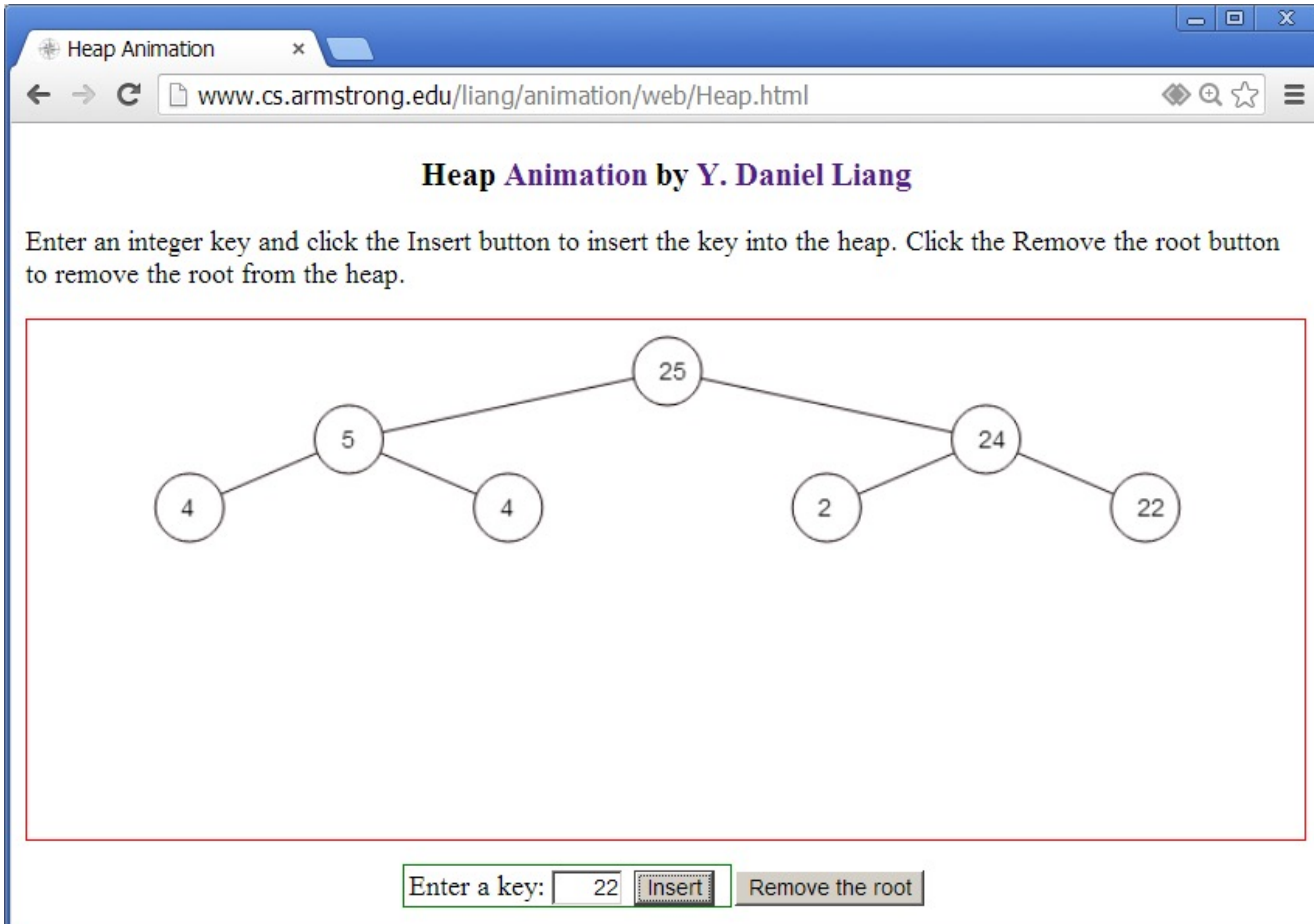
A binary tree is *complete* if every level of the tree is full except that the last level may not be full and all the leaves on the last level are placed left-most. For example, in the following figure, the binary trees in (a) and (b) are complete, but the binary trees in (c) and (d) are not complete. Further, the binary tree in (a) is a heap, but the binary tree in (b) is not a heap, because the root (39) is less than its right child (42).




© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# See How a Heap Works

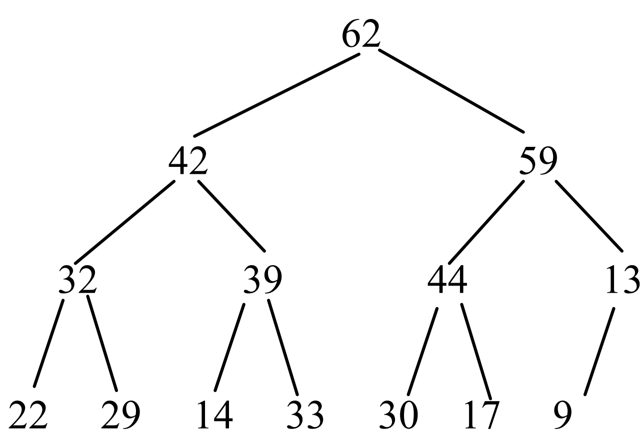
https://liveexample.pearsoncmg.com/dsanimation/HeapeBook.html

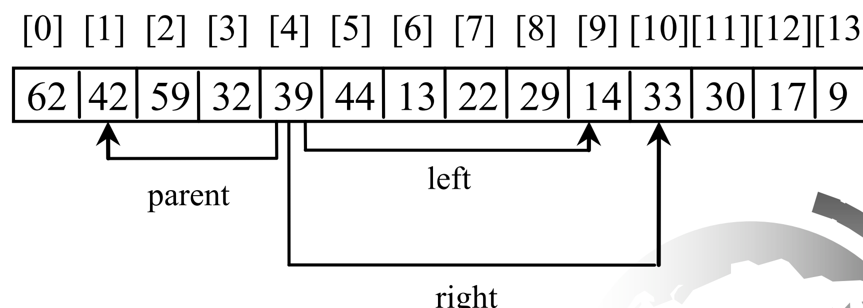



© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Representing a Heap

For a node at position *i*, its left child is at position *2i+1* and its right child is at position *2i+2*, and its parent is *(i-1)/2*. For example, the node for element 39 is at position 4, so its left child (element 14) is at 9 (*2*4+1*), its right child (element 33) is at 10 (*2*4+2*), and its parent (element 42) is at 1 (*(4-1)/2*).

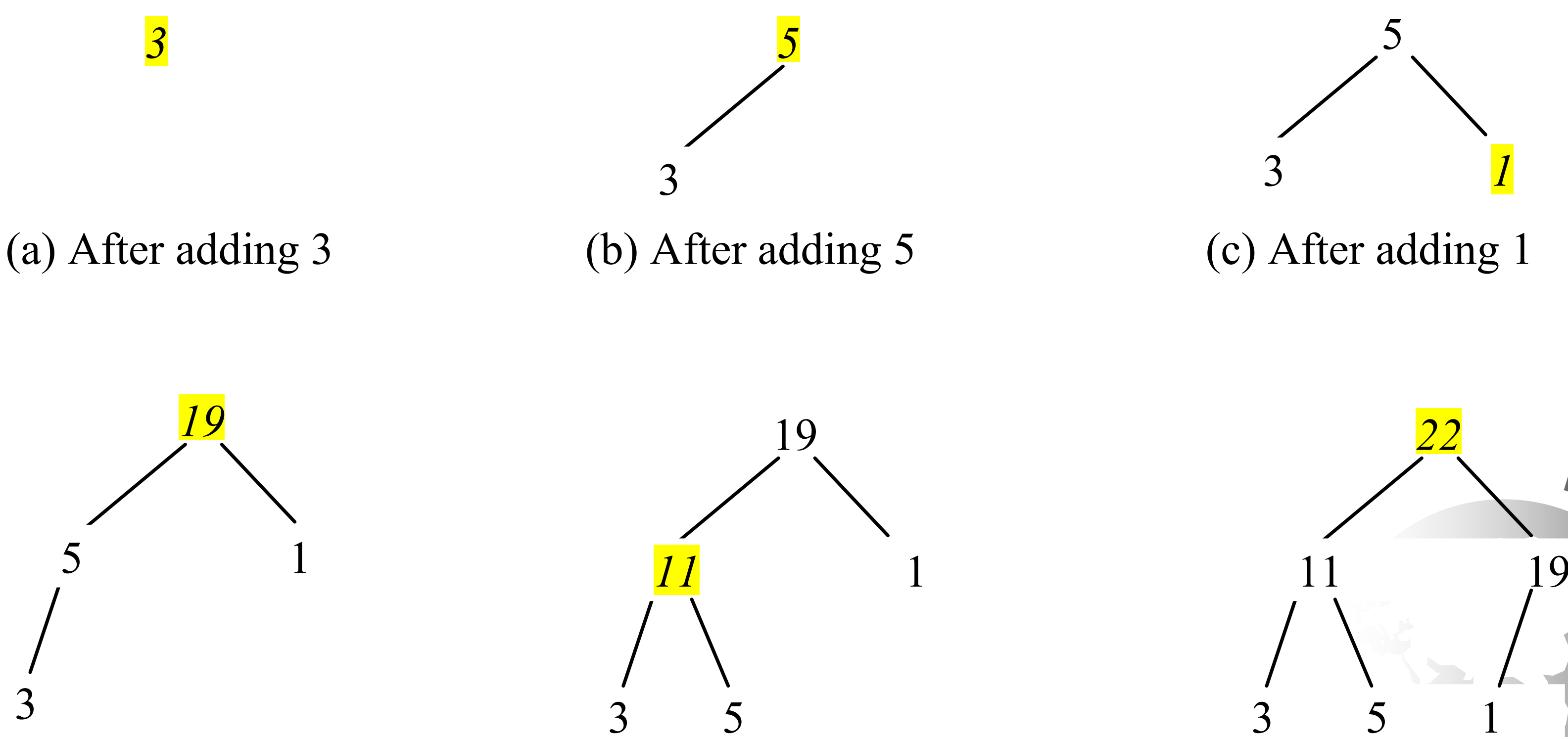



© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Adding Elements to the Heap

Adding 3, 5, 1, 19, 11, and 22 to a heap, initially empty



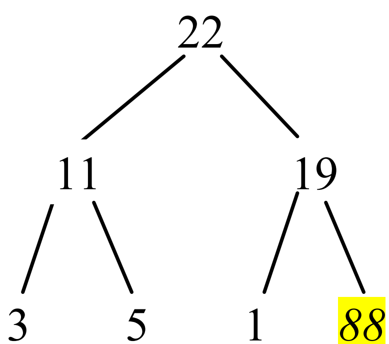

(a) After adding 3 (b) After adding 5 (c) After adding 1

(d) After adding 19 (e) After adding 11 (f) After adding 22

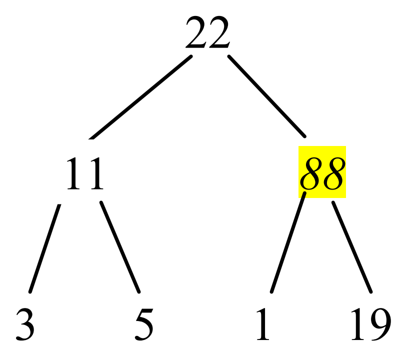
© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Rebuild the heap after adding a new node

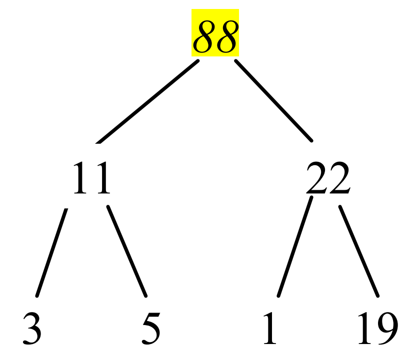
Adding 88 to the heap




(a) Add 88 to a heap




(b) After swapping 88 with 19

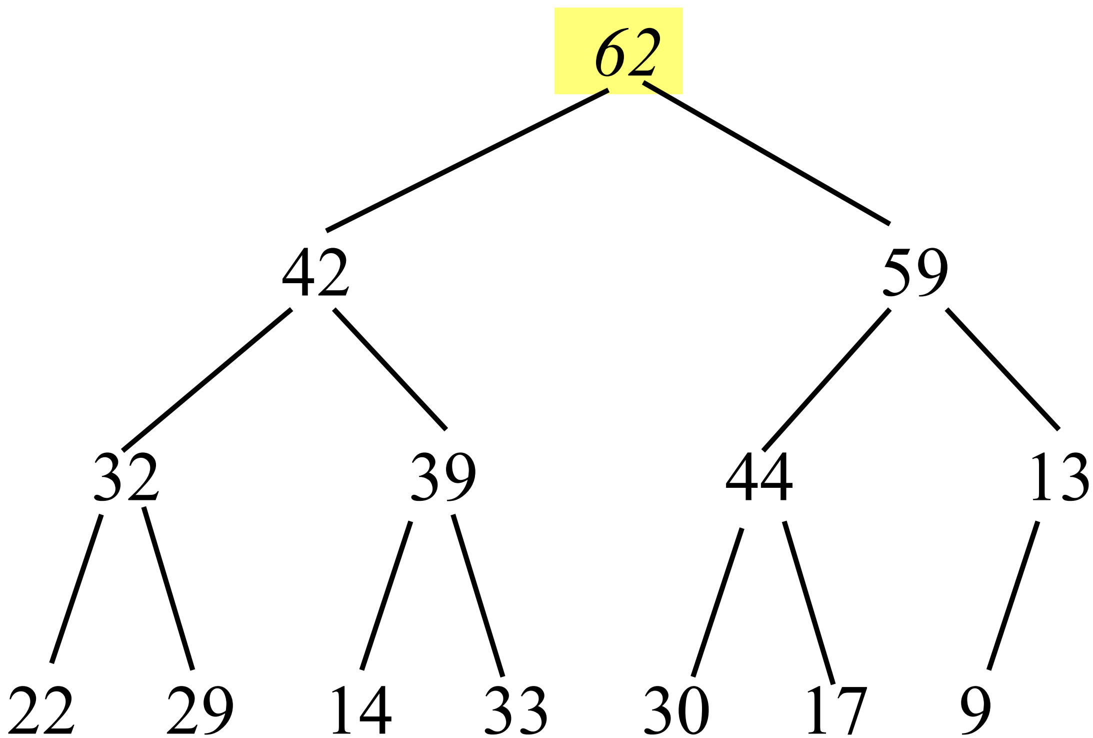



(b) After swapping 88 with 22

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Removing the Root and Rebuild the Tree

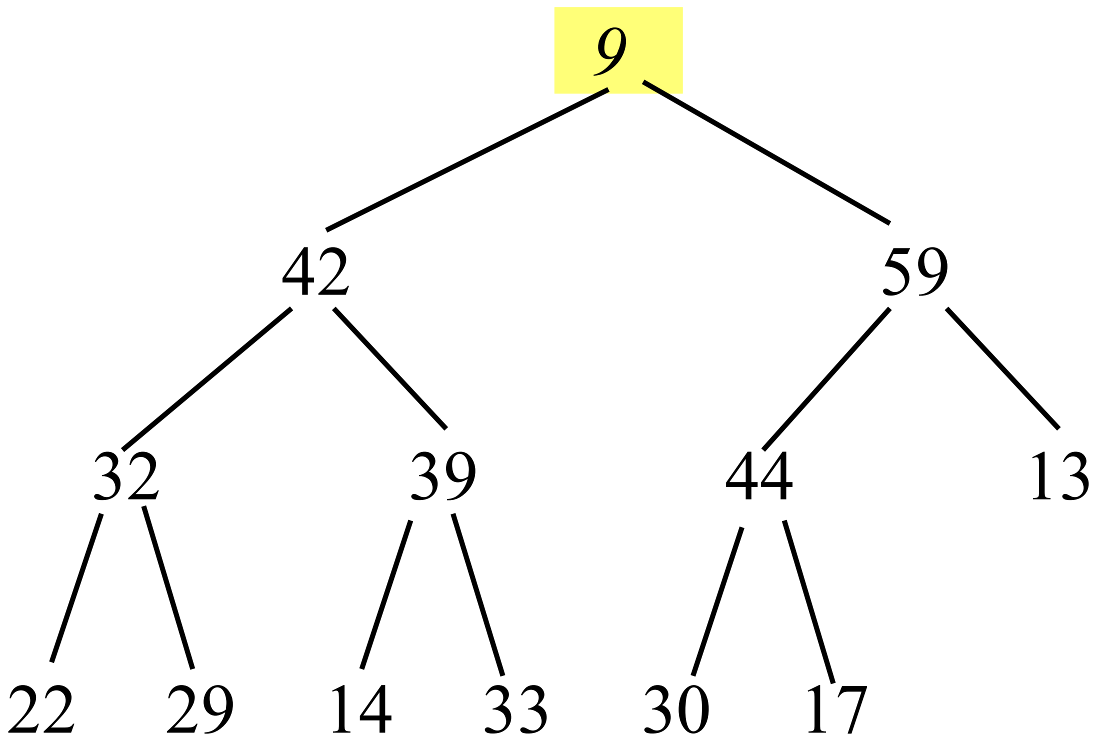
Removing root 62 from the heap




© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Removing the Root and Rebuild the Tree

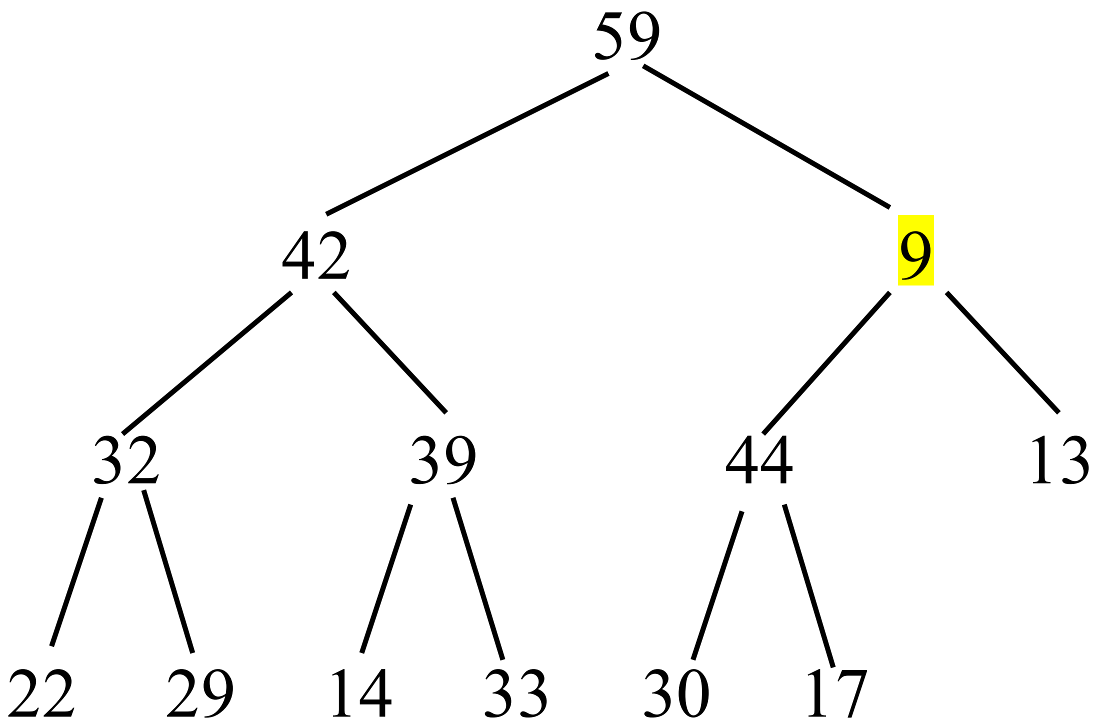
Move 9 to root




© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Removing the Root and Rebuild the Tree

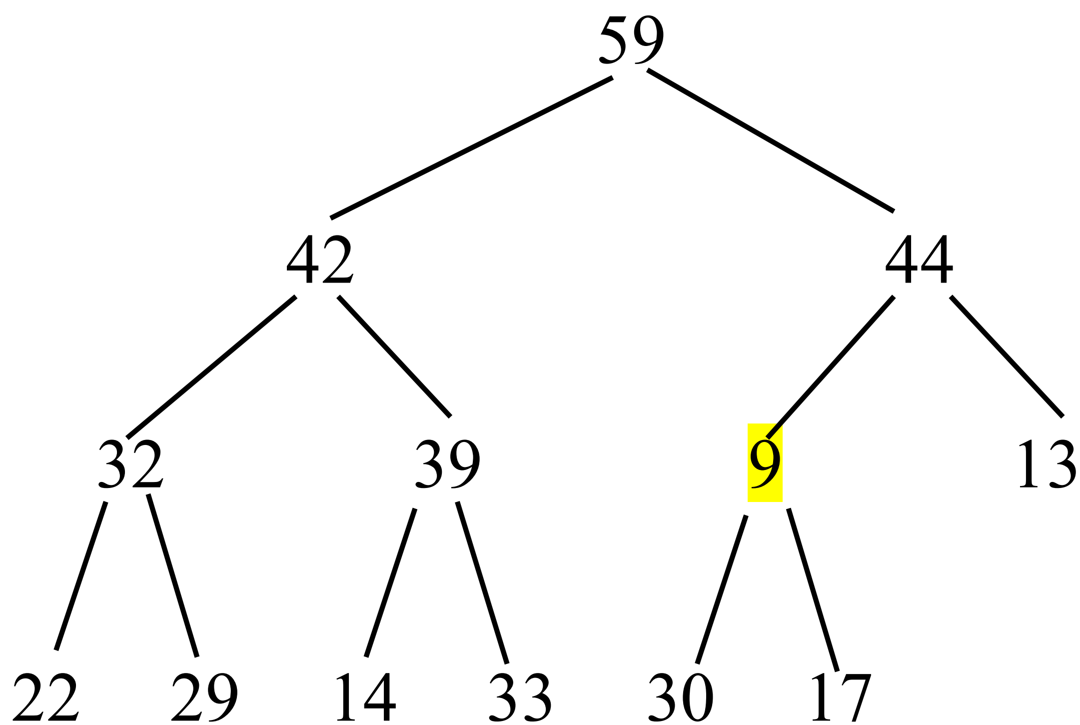
Swap 9 with 59




© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Removing the Root and Rebuild the Tree

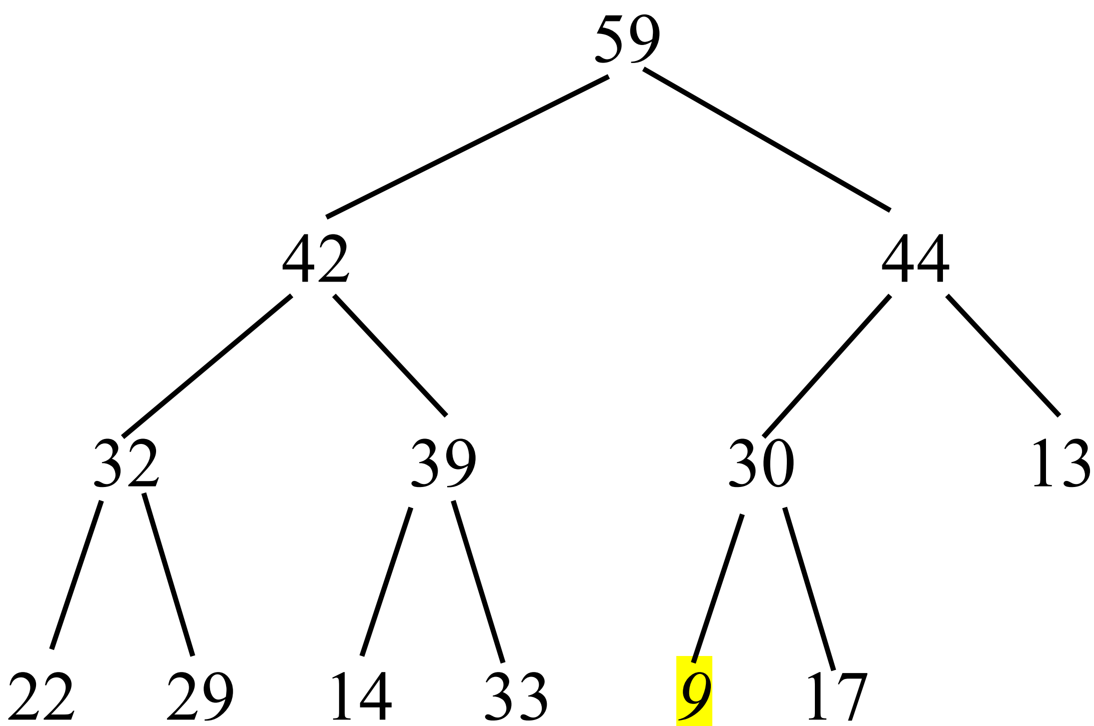
Swap 9 with 44




© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Removing the Root and Rebuild the Tree

Swap 9 with 30




© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# The Heap Class

| Heap | |
|---|---|
| -lst: list | Values are stored in a list internally. |
| Heap() | Creates an empty heap. |
| add(e: object): None | Adds a new element to the heap. |
| remove(): object | Removes the root from the heap and returns it. |
| getSize(): int | Returns the size of the heap. |
| isEmpty(): bool | Returns True if the list is empty. |
| peek(): object | Returns the largest element in the heap without removing it. |
| getLst(): list | Returns the list for the heap. |

Heap

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Heap Sort

HeapSort

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Heap Sort Time

Let $h$ denote the height for a heap of $n$ elements. Since a heap is a complete binary tree, the first level has 1 node, the second level has 2 nodes, the $k$th level has $2^{(k-1)}$ nodes, the *(h-1)*th level has $2^{(h-2)}$ nodes, and the hth level has at least one node and at most $2^{(h-1)}$ nodes. Therefore,

$$1 + 2 + ... + 2^{h-2} < n \leq 1 + 2 + ... + 2^{h-2} + 2^{h-1}$$

$$2^{h-1} - 1 < n \leq 2^h - 1 \quad 2^{h-1} < n + 1 \leq 2^h \quad \log 2^{h-1} < \log(n+1) \leq \log 2^h$$

$$h - 1 < \log(n+1) \leq h \qquad \log(n+1) \leq h < \log(n+1) + 1$$

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

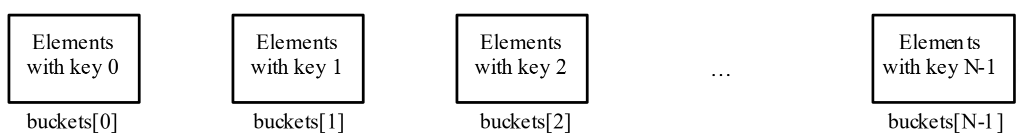
# Bucket Sort and Radix Sort

All sort algorithms discussed so far are general sorting algorithms that work for any types of keys (e.g., integers, strings, and any comparable objects). These algorithms sort the elements by comparing their keys. The lower bound for general sorting algorithms is O(nlogn). So, no sorting algorithms based on comparisons can perform better than O(nlogn). However, if the keys are small integers, you can use bucket sort without having to compare the keys.

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Bucket Sort

The bucket sort algorithm works as follows. Assume the keys are in the range from 0 to N-1. We need N buckets labeled 0, 1, ..., and N-1. If an element's key is i, the element is put into the bucket i. Each bucket holds the elements with the same key value. You can use an ArrayList to implement a bucket.




© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Radix Sort

Sort 331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9

| buckets[0] | buckets[1] | buckets[2] | buckets[3] | buckets[4] | buckets[5] | buckets[6] | buckets[7] | buckets[8] | buckets[9] |
|---|---|---|---|---|---|---|---|---|---|
| 230 | 331<br>231 | | 343<br>453 | 454<br>34 | 45<br>345 | | | | 59<br>9 |

230, 331, 231, 343, 453, 454, 34, 45, 345, 59, 9

| buckets[0] | buckets[1] | buckets[2] | buckets[3] | buckets[4] | buckets[5] | buckets[6] | buckets[7] | buckets[8] | buckets[9] |
|---|---|---|---|---|---|---|---|---|---|
| 9 | | | 230<br>331<br>231<br>34 | 343<br>45<br>345 | 453<br>454<br>59 | | | | |

9, 230, 331, 231, 34, 343, 45, 345, 453, 454, 59

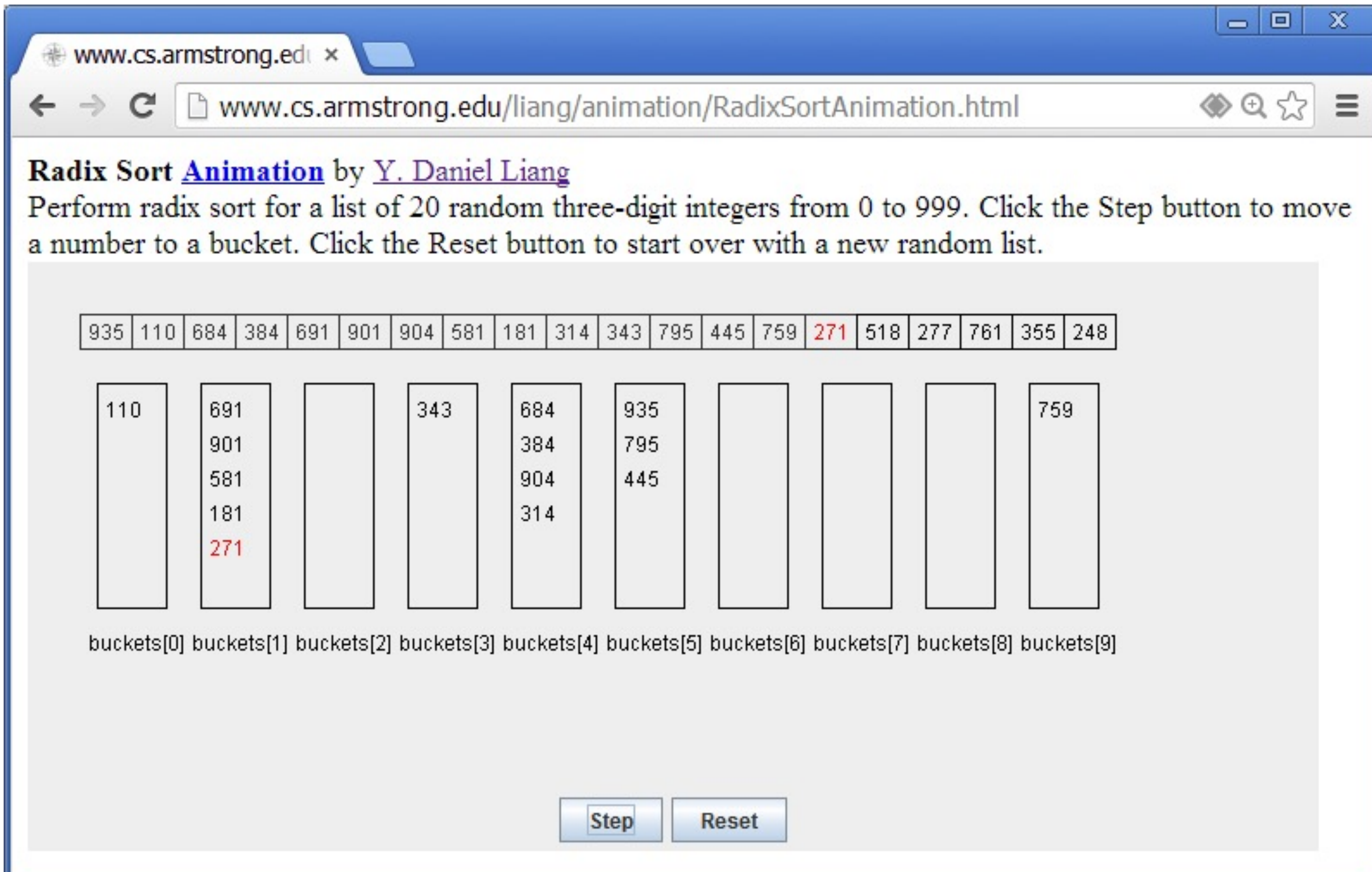
| buckets[0] | buckets[1] | buckets[2] | buckets[3] | buckets[4] | buckets[5] | buckets[6] | buckets[7] | buckets[8] | buckets[9] |
|---|---|---|---|---|---|---|---|---|---|
| 9<br>34<br>45<br>59 | | 230<br>231 | 331<br>343<br>345 | 453 | | | | | |

9, 34, 45, 59, 230, 231, 331, 343, 345, 453

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

# Radix Sort Animation

https://liveexample.pearsoncmg.com/dsanimation/RadixSorteBook.html




© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.