

# CSSE 220 Day 25

Sorting Algorithms  
Algorithm Analysis and Big-O  
Function Objects and the Comparator Interface

Checkout *SortingAndSearching* project from SVN

# Tips from Test 2:

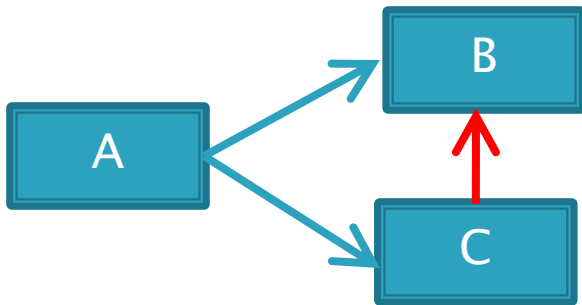
## *polymorphism*

- ▶ `One s = new Two();`  
`s.delta();`
- ▶ `s` is *actually* a `Two`,  
but *declared* to be a `One`
- ▶ Compiles?
  - Yes, `One` has a `delta`
- ▶ When executed,  
`s` morphs to a `Two`:
  - Looks in `Two` for a `delta`,  
doesn't find one
  - Then looks in `One`, finds one and  
runs it (inheritance). Prints "D".
  - Then looks for a `beta` applied to  
*this* – *this* is a `Two`, so runs  
`Two`'s `delta`, printing an "E".

```
class One implements Top {  
  
    public void beta() {  
        System.out.println("B");  
    }  
  
    public void delta() {  
        System.out.println("D");  
        this.beta();  
    }  
}
```

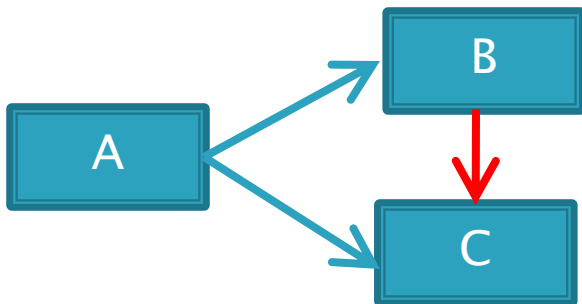
```
class Two extends One  
    implements Top {  
  
    public void beta() {  
        System.out.println("E");  
    }  
  
    // no delta  
}
```

# Tips from Test 2: *implementing has-a*



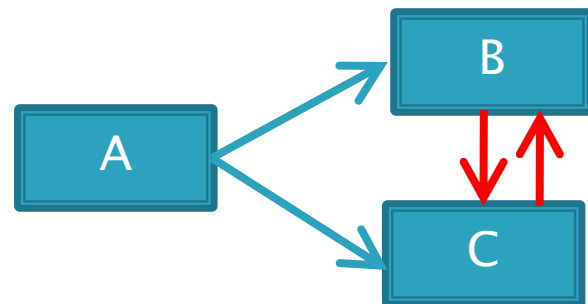
In A:

```
B b = new B (...);  
C c = new C (b, ...);
```



In A:

```
C c = new C (...);  
B b = new B (c, ...);
```



In A:

```
B b = new B (...);  
C c = new C (...);  
b.setC(c);  
c.setB(b);
```

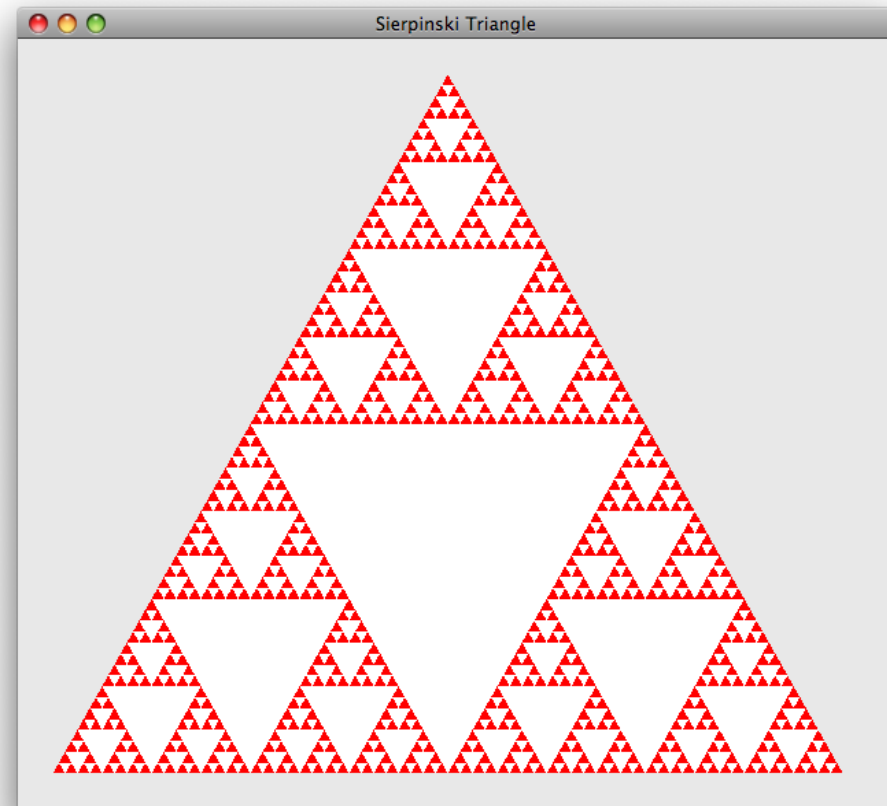
In B (and likewise C):

```
public void setC(C c) {  
    this.c = c;  
}
```

# Tips from Test 2:

## *Recursion*

- ▶ Hint: determine the recursive step first
  - Top-down thinking instead of bottom-up
- ▶ The *shaded area in the whole triangle* is \_\_\_\_  
*the shaded area in what triangle(s)*?
- ▶ Answer: 3 times the shaded area in the lower-left triangle. So the code for the recursive case is:
  - `return 3 * shadedArea(x, y, base/2);`
    - Note that I used a helper method (alternative: construct a triangle with half the base), and that x and y are NOT needed



# Questions

» Exam results

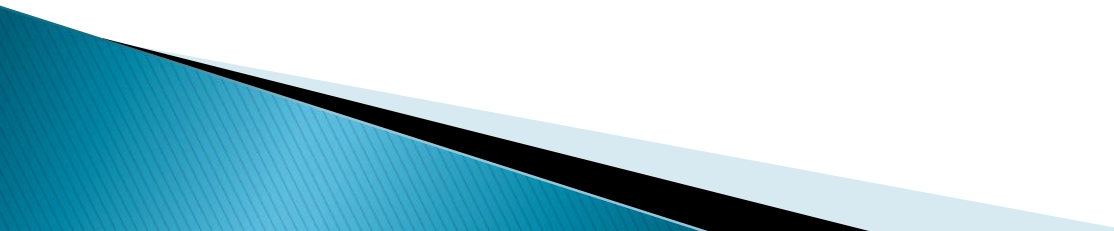
# Remember Selection Sort?

»» Let's see...

# Why study sorting?

» Remember  
Shlemiel the Painter

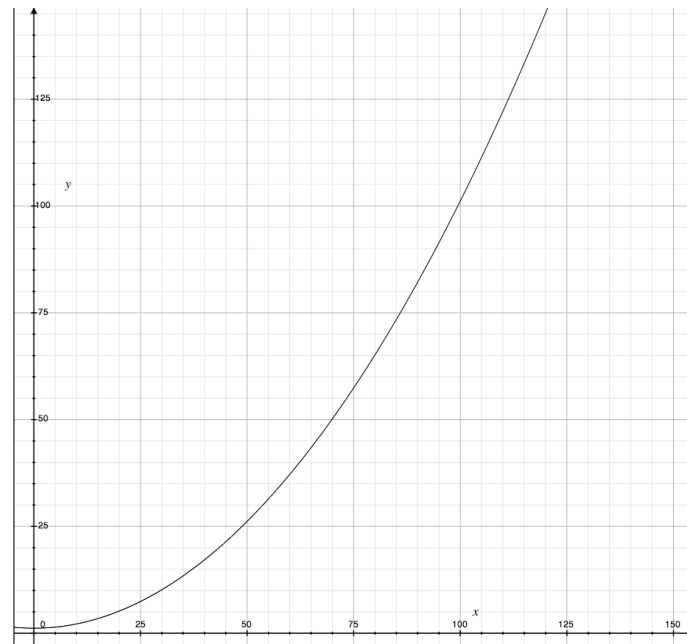
# Course Goals for Sorting: You should...

- ▶ Be able to **describe** basic sorting algorithms:
    - Selection sort
    - Insertion sort
    - Merge sort
    - Quicksort
  - ▶ Know the **run-time efficiency** of each
  - ▶ Know the **best and worst case** inputs for each
- 

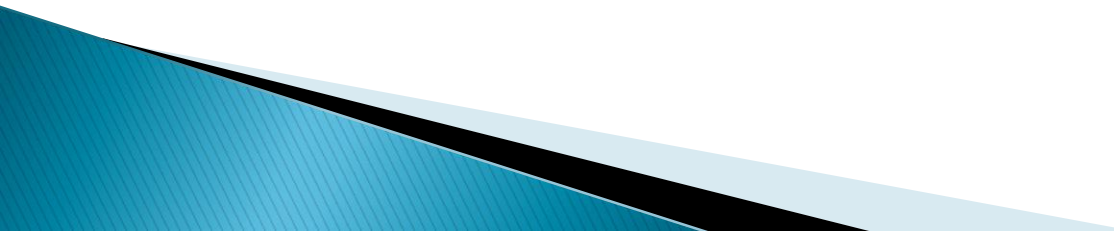


# Profiling Selection Sort

- ▶ **Profiling**: collecting data on the run-time behavior of an algorithm
- ▶ How long does selection sort take on:
  - 10,000 elements?
  - 20,000 elements?
  - ...
  - 80,000 elements?
- ▶  $O(n^2)$



# Big-Oh Notation

- ▶ In analysis of algorithms we care about differences between algorithms on very large inputs
  - ▶ We say, “selection sort takes on the order of  $n^2$  steps”
  - ▶ Big-Oh gives a formal definition for “on the order of”
- 

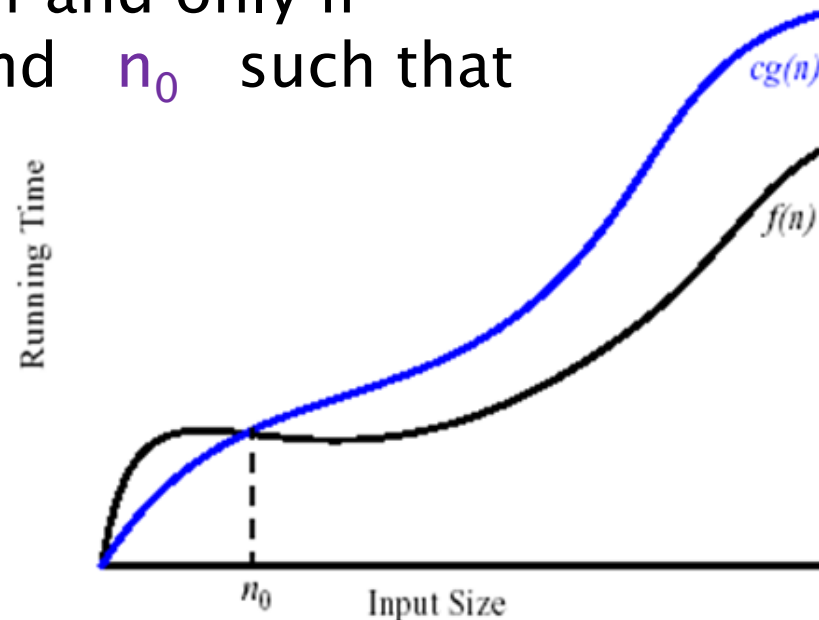
# Definition of big-Oh

## ► Formal:

- We say that  $f(n)$  is  $O(g(n))$  if and only if
- there exist constants  $c$  and  $n_0$  such that
- for every  $n \geq n_0$  we have
- $f(n) \leq c \times g(n)$

## ► Informal:

- $f(n)$  is *roughly proportional* to  $g(n)$ , for large  $n$



- ## ► Example: $7n^3 + 24n^2 + 3000n + 45$ is $O(n^3)$
- Because it is  $\leq 3,077 \times n^3$  for all  $n \geq 1$

# Big-Oh rules

## ► Formal:

- We say that  $f(n)$  is  $O(g(n))$  if and only if
- there exist constants  $c$  and  $n_0$  such that
- for every  $n \geq n_0$  we have
- $f(n) \leq c \times g(n)$

- Polynomials: keep the highest power, discard its coefficient

$$34n^5 + 20n^2 + 10000$$

is  $O(n^5)$

- More generally:

1. Discard all multiplicative constants
2. Pick the “dominating” additive expression per chart to the right, discard other additive terms

$$30n^2 + 4n^3 \log n + 45n + 70n^3 + 85$$


is  $O(n^3 \log n)$

FUNCTION	NAME
$c$	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
$N$	Linear
$N \log N$	$N \log N$ ←
$N^2$	Quadratic
$N^3$	Cubic
$2^N$	Exponential

# Insertion Sort

## ► Basic idea:

- Think of the list as having a sorted part (at the beginning) and an unsorted part (the rest)
- Get the first number in the unsorted part
- Insert it into the correct location in the sorted part, moving larger values up to make room



Repeat until  
unsorted part is  
empty

# Insertion Sort Exercise, Q4–11b

- ▶ **Profile** insertion sort
- ▶ **Analyze** the *worst case*
  - Assume that the inner loop runs as *many* times as it can
  - Count the number of times `compareTo` is executed
  - What input causes this worst-case behavior
- ▶ **Analyze** the *best case*
  - Assume that the inner loop runs as *few* times as it can
  - Count the number of times `compareTo` is executed
  - What input causes this best-case behavior
- ▶ Does the input affect selection sort?

Ask for help if  
you're stuck!

Handy Fact

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

Q6–13b

# Searching

- ▶ For searching *unsorted* data, what's the worst case number of comparisons we would have to make?

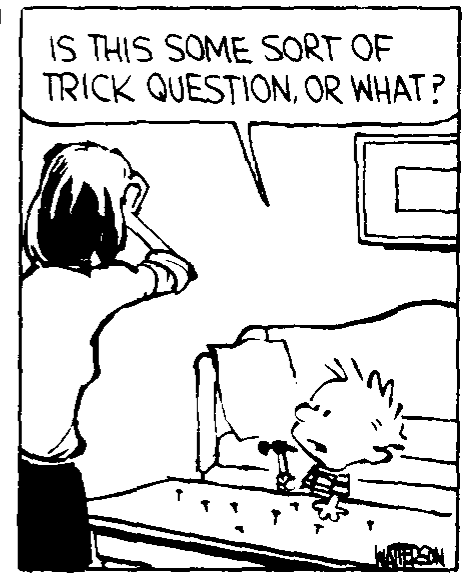
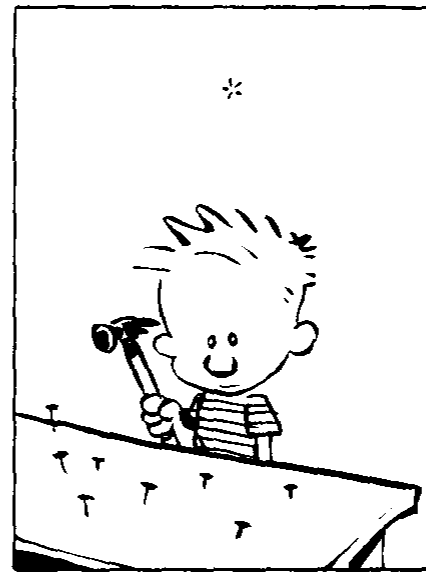
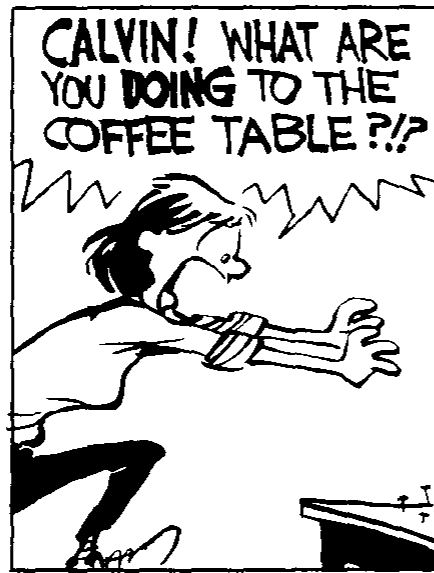
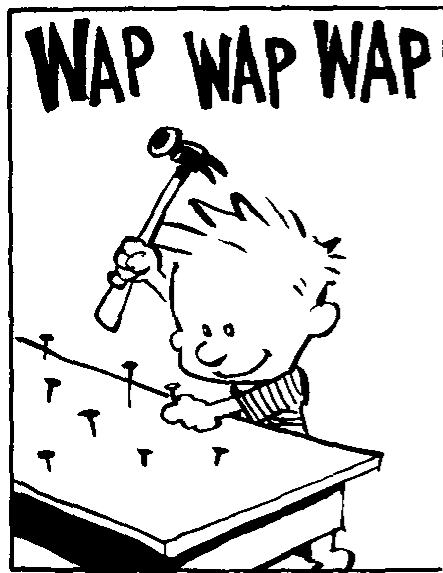
# Binary Search of Sorted Data

- ▶ A **divide and conquer** strategy
- ▶ Basic idea:
  - Divide the list in half
  - Should result be in first or second half?
  - Recursively search that half



# Analyzing Binary Search

- ▶ What's the best case?
- ▶ What's the worst case?



Perhaps it's time for a break.

# Merge Sort

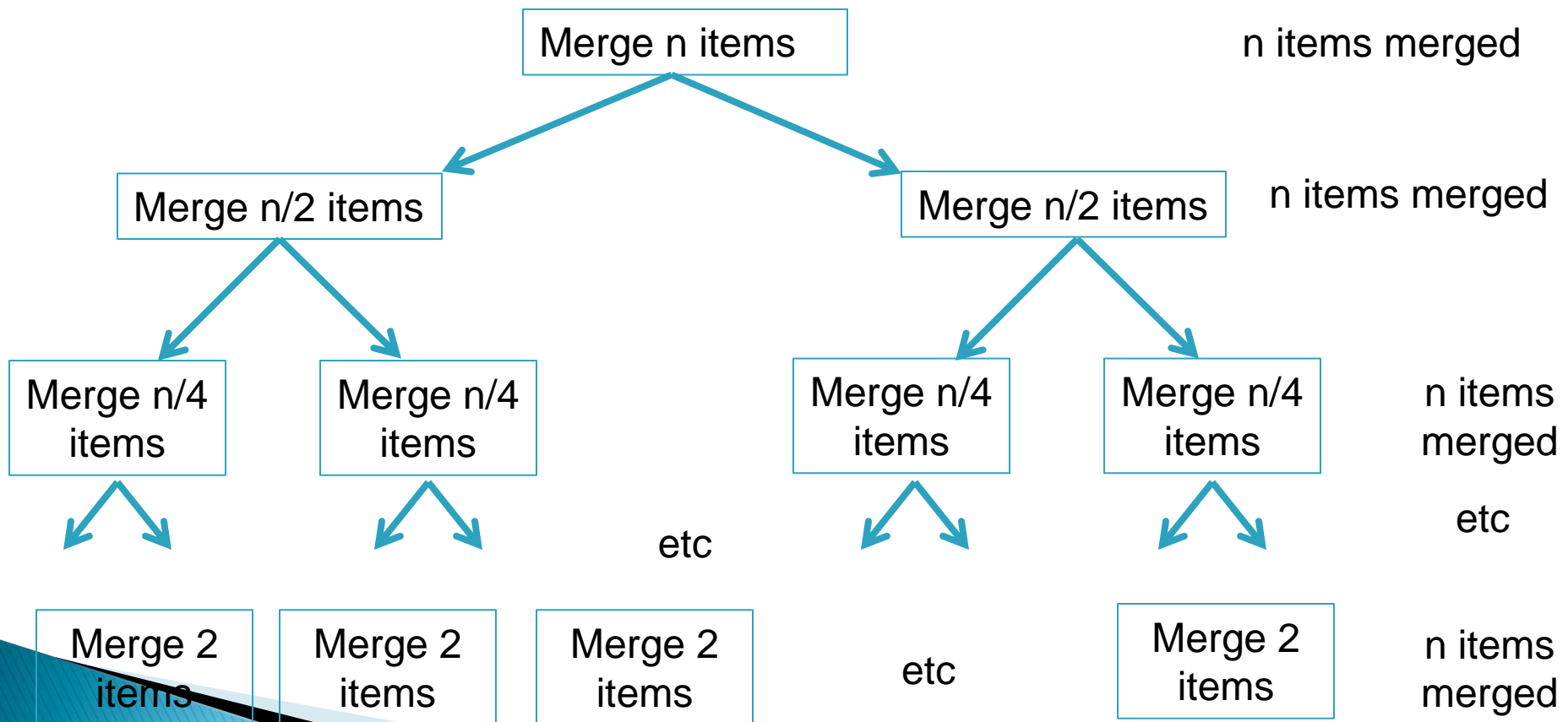
- ▶ Basic recursive idea:
  - If list is length 0 or 1, then it's already sorted
  - Otherwise:
    - Divide list into two halves
    - Recursively sort the two halves
    - **Merge** the sorted halves back together
- ▶ Let's profile it...

# Analyzing Merge Sort

If list is length 0 or 1,  
then it's already sorted

► Otherwise:

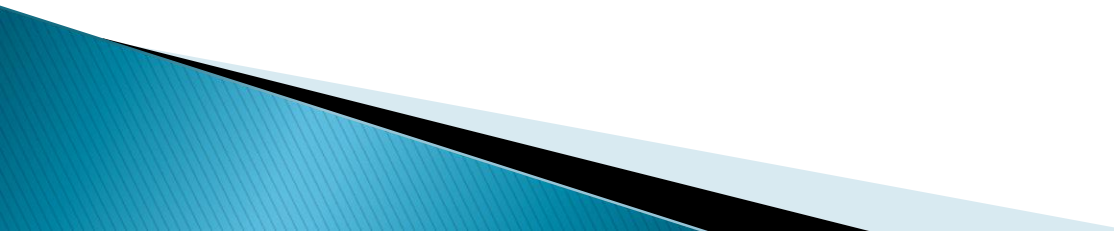
- Divide list into two halves
- Recursively sort the two halves
- **Merge** the sorted halves back together



# Function Objects

- » Another way of creating reusable code

# A Sort of a Different Order

- ▶ Java libraries provide efficient sorting algorithms
    - `Arrays.sort(...)` and `Collections.sort(...)`
  - ▶ But suppose we want to sort by something other than the “natural order” given by `compareTo()`
  - ▶ Function Objects to the rescue!
- 

# Function Objects

- ▶ Objects defined to just “wrap up” functions so we can pass them to other (library) code
- ▶ We’ve been using these for awhile now
  - Can you think where?
- ▶ For sorting we can create a function object that implements Comparator

# Data Structures

- » Understanding the engineering trade-offs when storing data



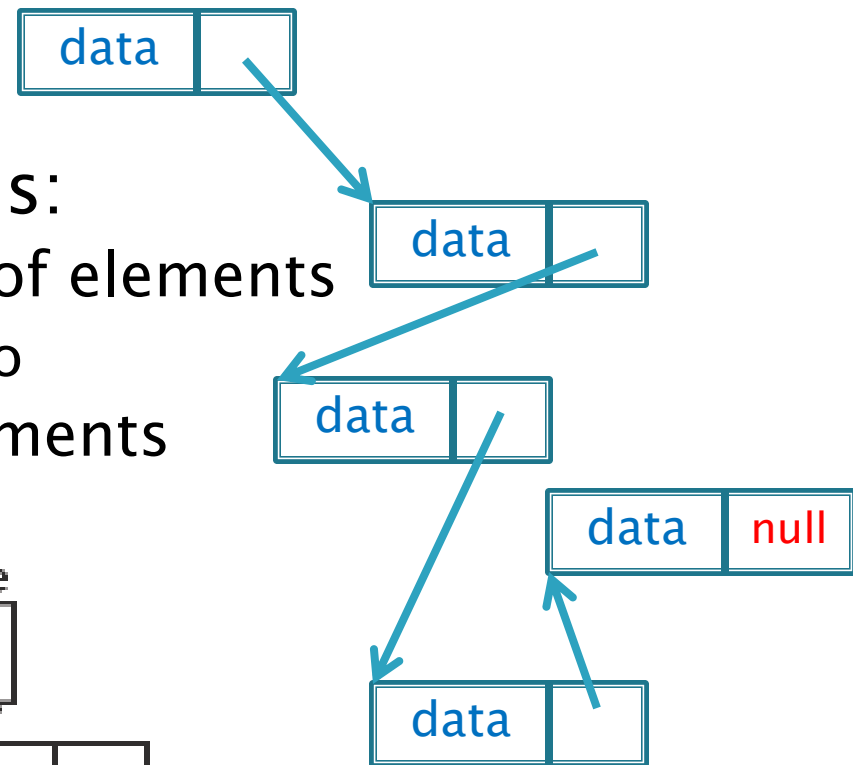
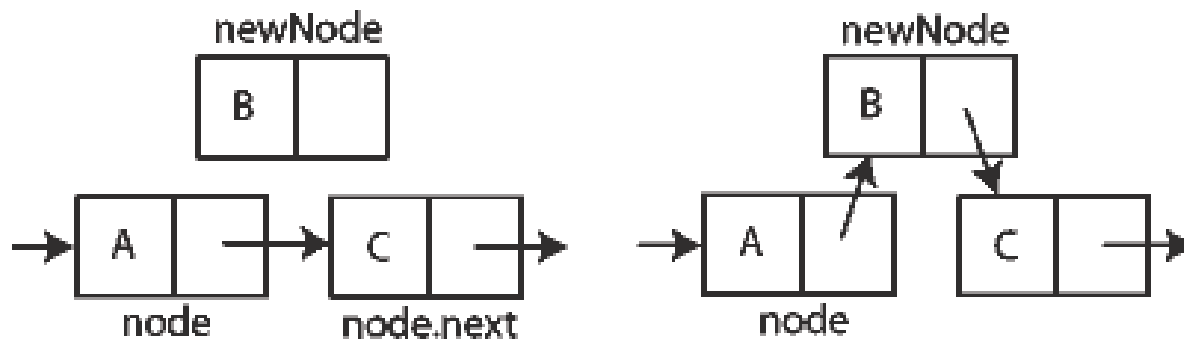
# Data Structures

- ▶ Efficient ways to store data based on how we'll use it
- ▶ So far we've seen ArrayLists
  - Fast addition to end of list
  - Fast access to any existing position
  - Slow inserts to and deletes from middle of list

# Another List Data Structure

- ▶ What if we have to add/remove data from a list frequently?

- ▶ LinkedLists support this:
  - Fast insertion and removal of elements
    - Once we know where they go
  - Slow access to arbitrary elements



Insertion, per Wikipedia

Q17,18

# Next Time

- ▶ Implementing ArrayList and LinkedList
- ▶ A tour of some data structures
- ▶ Some VectorGraphics work time