CS112: Data Structures

Intro to Course

Asymptotic complexity and big-O

Linked Lists

CS112: Data Structures

- Instructor: Prof. Louis Steinberg
  - office: Hill 401
  - email: lou@cs.rutgers.edu
  - Office hours: To be Announced
- TA: Binh Pham
  - Office: Core 336
  - Email: binhpham@cs.rutgers.edu
  - Office hours: 4-5pm Mondays

Class Web Site

- **http://sakai.rutgers.edu**
  - **Log in using Rutgers NetID & password, click on "CS112, Summer 2011" tab**

– **You are assumed to know anything posted.**

Prerequisite

- CS 111 or equivalent
  - Comfortable writing and debugging programs 1 to 2 pages long
  - Basic Java (types, control flow, etc.)
  - Arrays (1D)
  - Sequential search
  - Insertion sort
  - Recursion
  - Using objects (not defining classes)
  - Big-O worst case analysis

Prerequisite

- **Determination to work hard and keep up-to-date on coursework**

Requirements

- Problem sets - not to turn in
- Homework Projects
- Written exams
  - Midterms and Final

Textbook

- ***Data Structures Outside In with Java, 1st Edition.***

  by Sesh Venugopal

  Prentice-Hall, 2006.

  ISBN 978-0131986190.

What is a data structure

- A representation scheme that stores
  - Multiple pieces of data
  - Relationships between pieces of data
- E.g,
  - Object
  - Array
  - Linked List

What to know about a DS

- What operations can we do?
- What do they cost?
  - Time
  - Memory space

How long does it take

- **Problem: actual time depends on**
  - **What computer**
  - **What language**
  - **What compiler**
  - **What programmer**
  - **What input**
- **We want a measure of time that does not depend on these**

Solutions

- Count operations, not time
- Op count = f(input size)
- Among inputs of the same size, use worst or average op count
- Abstract away details of f: O(f)
  - focus on large inputs
  - Ignore constant multiples

Example

Input: double array A, int n, double target

Output: boolean: are any of first n elements of A equal to target

```
for (i = 0; i<n; i++){
   if (A(i) == target){
     break;}}
return i < n;
```

Count Operations, Not Time

- So processor speed doesn't matter
- But which operation to count?

  Count should model time of algorithm
  - Most frequent / inner loop
  - Most time consuming
  - Inherent in algorithm, not language

Size of input

- **Problem: number of operations depends on size of input**
- **Solution: number of ops = f(input size)**
  - **E.g., ops = size, ops = size * (size-1) / 2**
- **How do we define size of input?**
  - **Usually obvious measure**
  - **Sometimes several equally good**
    - **eg, n x n matrix: #rows or # elements**
    - **choose any but be clear which you chose**

Same Size But Not Same Ops

- Sometimes inputs of same size take different numbers of operations
  - In example, ops varies from 1 to size
- Solution: use
  - Worst case
  - Average case
    - Weighted by probability

Array Search Example

- **Worst case:**
  - **Not found or found at end**
  - **Ops = size**

Average Case

- **To compute average cost:**
  - **What determines cost?**
    EG for array search: where in the array will the target be found?
  - **For each such case,figure out the cost of that case (# operations)| and the probablility of that case**
  - **Sum over all cases of the cost times the probablility**

Array Search Example

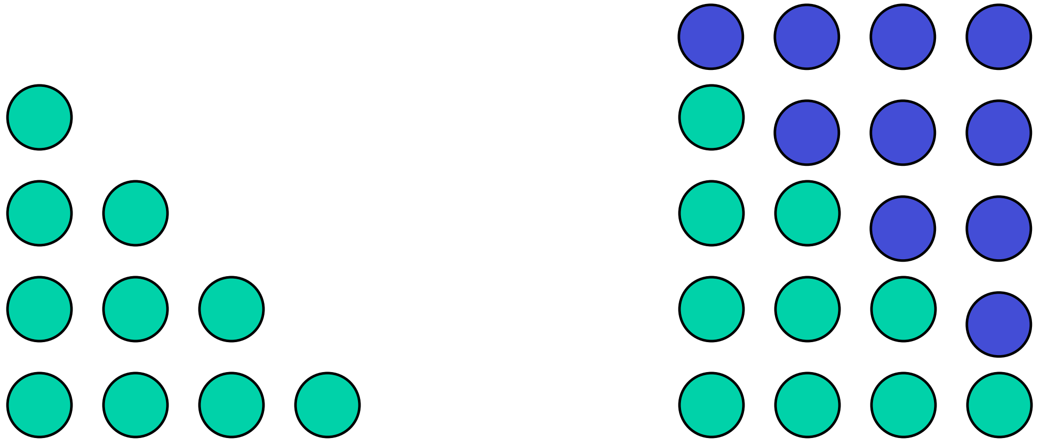
- **Assume:**
  - **Always found**
  - **Target equally likely to be found in each place**
- **Sum over all positions p of**

  ops if found at p * probability of found at p

Let n = size of input.

Average cost = $\sum_{p=1}^{n} (p * 1/n) = \frac{1}{n} \sum_{p=1}^{n} p$

$1 + 2 + \ldots + n$



$$= n * (n + 1) / 2$$

Array Search Example

- **Average cost**

$$\sum_{p=1}^{n}(p*1/n) = \frac{1}{n}\sum_{p=1}^{n}p = \frac{1}{n}\frac{n*(n+1)}{2} = \frac{n+1}{2}$$

Array Search Example

- Assume 50% chance of not found
  - If found, equal chance in each position

Average Cost =

Prob(found) * cost if found

+ Prob(not found) * cost if not found

= 1/2 *(n+1)/2

+1/2 * n

= 3/4 n + 1/4

Worst vs Average Case

Worst case:

- **Often easier to find**
- **No assumptions about probability of inputs**
- **Sometimes what we really care about**
- **But sometimes misleading**
  - **E.g., quicksort**

Average case

- **Often harder to find**
- **Requires assumptions about probability of inputs**
- **Sometimes what we really care about**

Another Example

Input: double array A, int n

Output: largest number in first n elements of A

```
double big = A(0);
for (int i = 1; i<n; i++){
 if (big < A(i)){
     big = A(i)}}
```

More Examples

```
for (int i = 0; i < m; i++){
  for (int j = 0; j < n; j++){
      sum += A(i, j)}}

for (int i = 0; i < m; i++){
  for (int j = 0; j < i; j++){
      sum += A(i, j)}}

for (int i = 0; i < m; i++){
  for (int j = i-5; j < i; j++){
      sum += A(i, j+5)}}
```

Recursive example

```
int foo(int i){
   if (i == 1){
     system.out.println("foo");
    } else {
     foo(i-1);
     foo(i-1);
    }}
```

Asymptotic Complexity

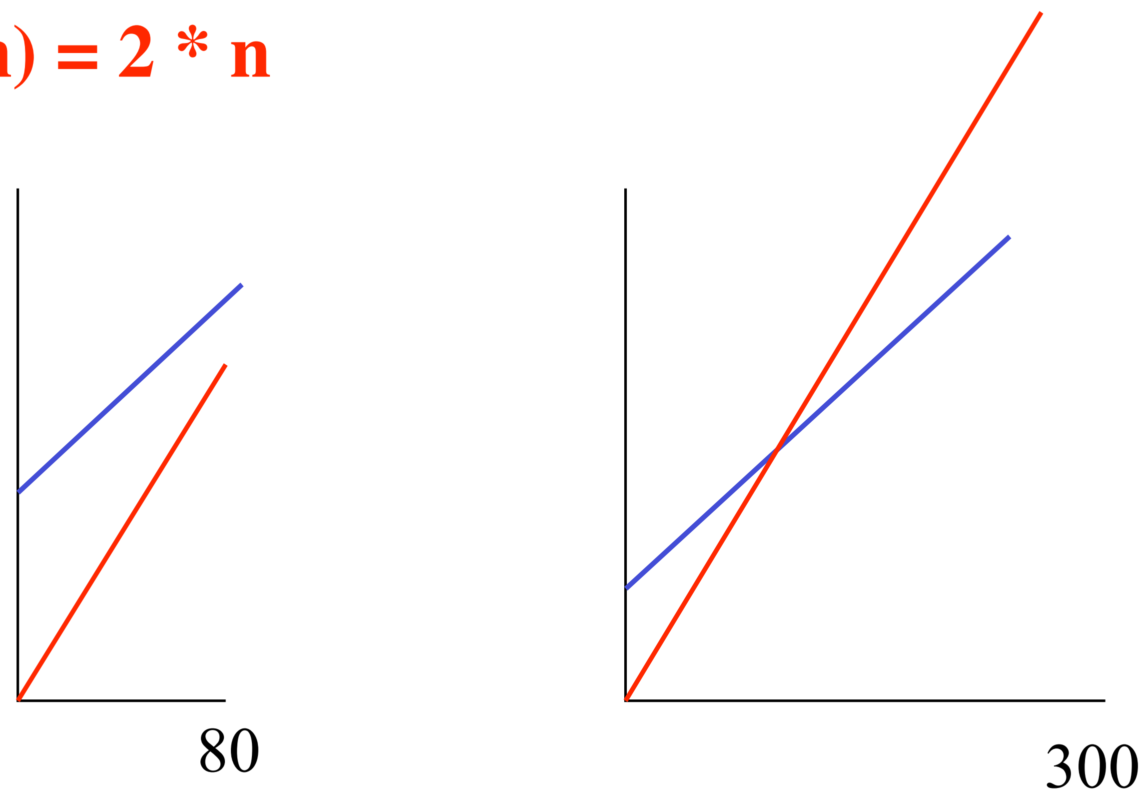
- **Assume a problem, input size n**
  - **Algorithm F takes worst case n+100 ops**
  - **Algorithm G takes worst case 2 * n ops**
- **Which takes longer?**

Big O

- **Which function is bigger?**

  f(n) = n + 100

  g(n) = 2 * n

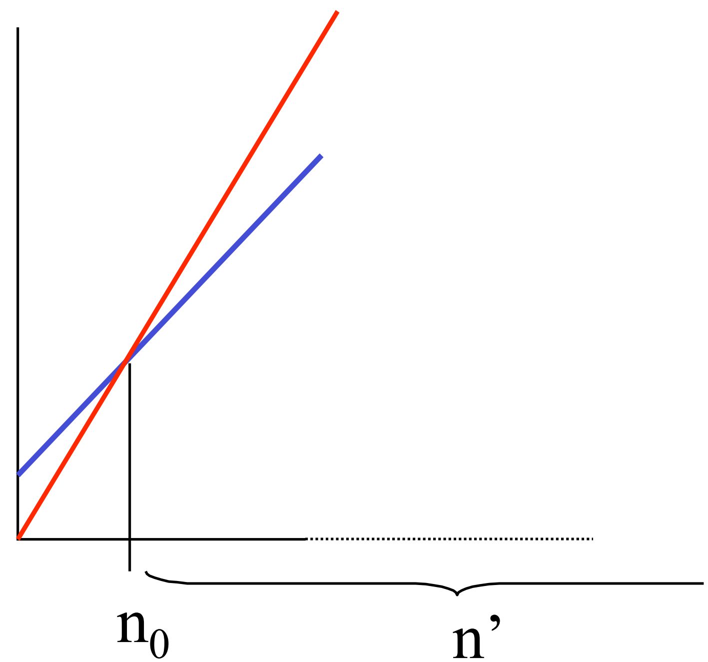


Asymptotically faster

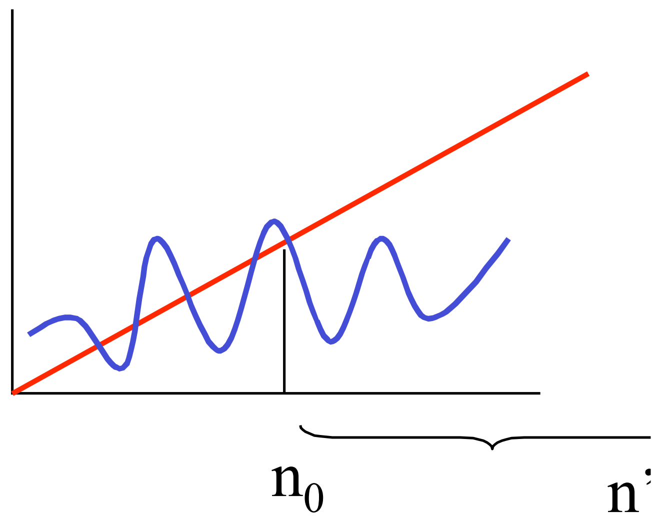
- **Function g(n) grows asymptotically faster than f(n) if**

  there is an n_0 such that for all $n' > n_0$,

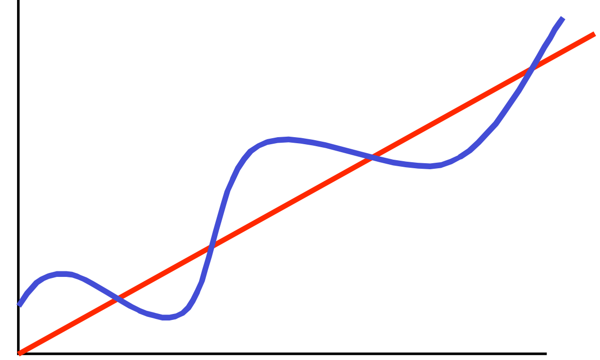
  $g(n') > f(n')$



Asymptotically faster



Yes



No

Big O

- **Which function is bigger?**

  $f(n) = 4 * n$

  $g(n) = 2 * n$

- **What if one algorithm is run on a machine that is twice as fast as the other?**

Big O

- **f(n) is O(g(n)) if there is some constant c such that c*g(n) grows asymptotically faster than f(n)**
- **$3 * n^2 + 7$ is $O(n^2)$ because $4 * n^2$ grows asymptotically faster than $3 * n^2 + 7$**

Big O

- **Informally when we say f(n) is O(g(n)) we mean g(n) is the simplest function for which this is true**
  - **technically, n+4 is $O(3*n^2 - 9*n + 5)$ but we prefer to say n+4 is O(n)**

Rules for Big O

- **k is O(1)**

  341 is O(1)

- **f+g = max (O(f), O(g))**

  n + 1 is max(O(n), O(1)) = O(n)

- **k * f = O(f)**

  $O(4*n^4) = O(n^4)$

- **$O(n^A) < O(n^B)$ if A < B**

  $O(n^3) < O(n^4)$

- **O(polynomial) is O(highest exponent term)**

  $5 n^4 + 44 n^2 + 55 n + 12$ is $O(n^4)$

Names for Big O

- $O(1)$ is constant
- $O(n)$ is linear
- $O(n^2)$ is quadratic
- $O(k^n)$ is exponential

  $O(k^n)$ is bigger than any polynomial

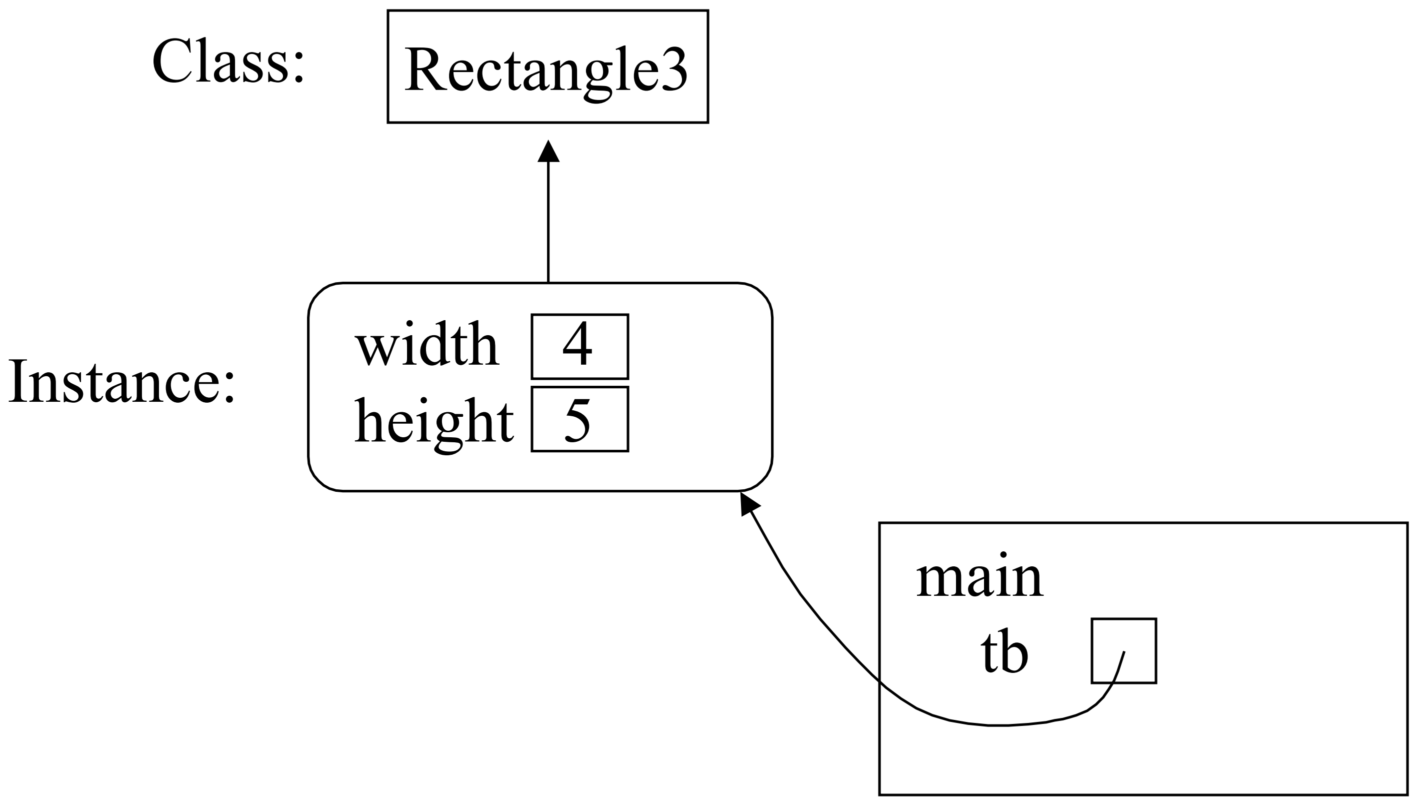
Try These

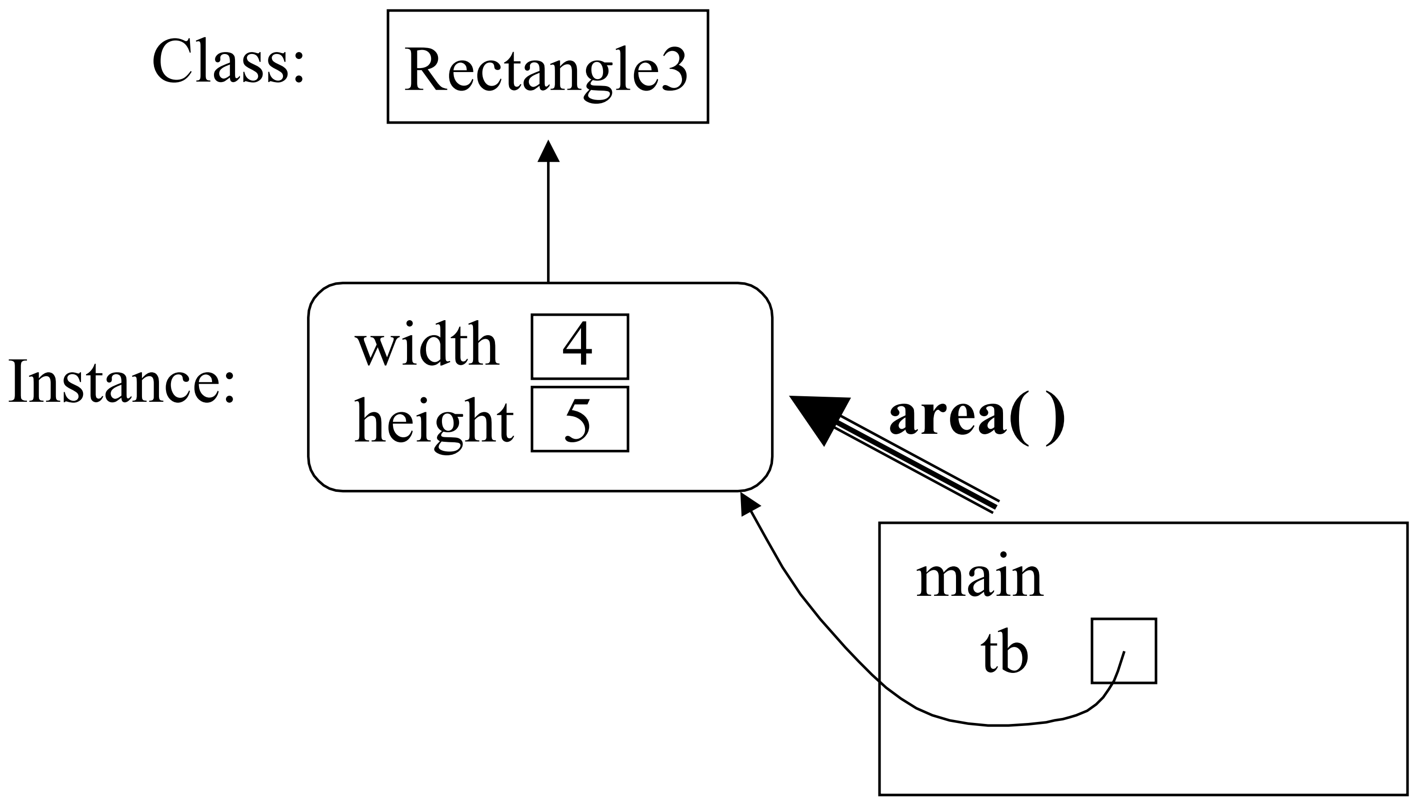
- **What is big-O of these?**

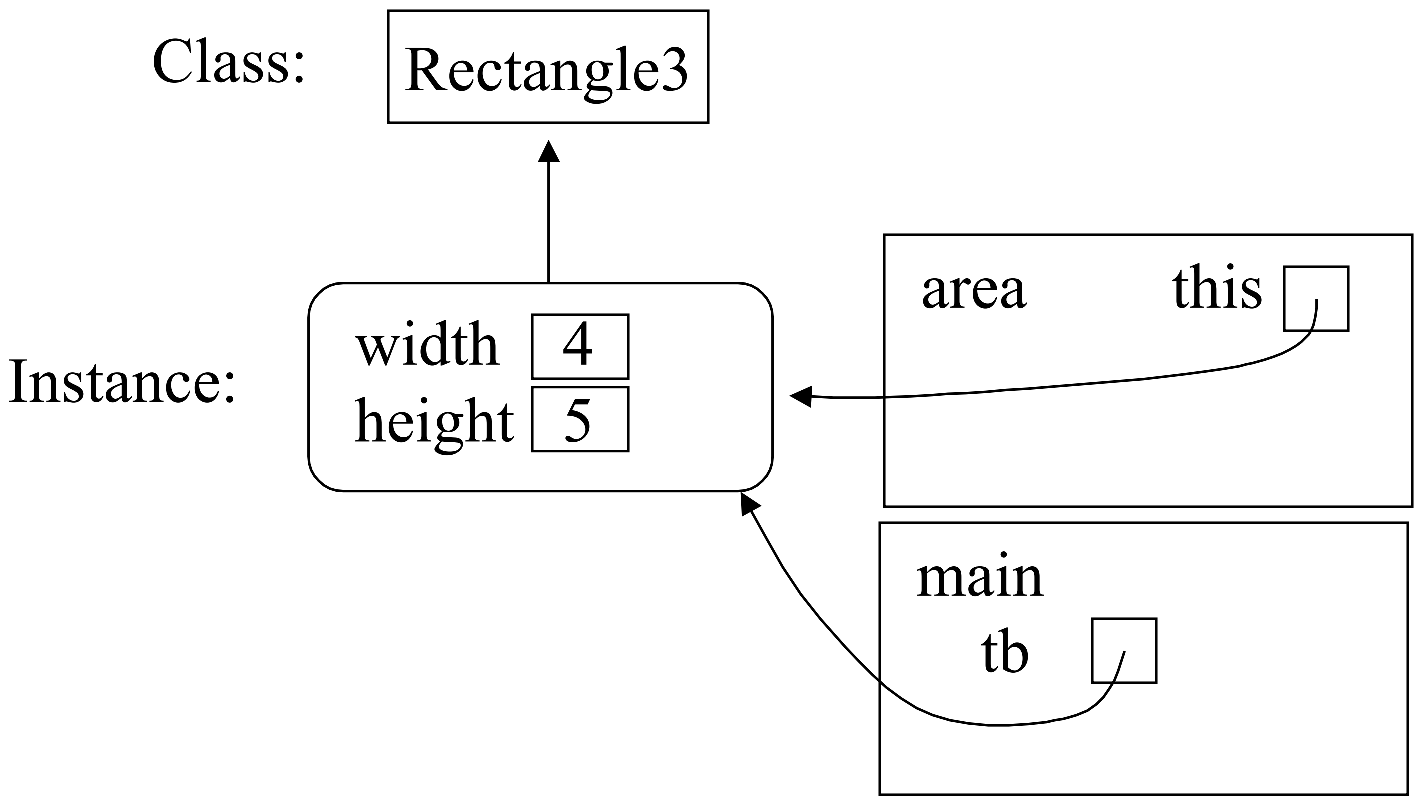
1. $x^2 + 100\,x + 10$
2. $(n-1)*n/2$
3. $10 + \mathrm{sqrt}(n)$
4. $(10 + \mathrm{sqrt}(n))^2$
5. $n + \mathrm{sqrt}(n)$
6. 2^{n-1}
7. 2^{4n}

Objects, methods, and variables

- In Java, data is stored in
  - Parameters and local variables of methods
  - Instance variables of objects
- Variables can hold references to another object
- See Rectangle1.java, Rectangle2.java, Rectangle3.java, and Rectangle4.java


Class:
Rectangle3
Instance:
width
4
height
5
main
tb


Class:
Rectangle3
Instance:
width
4
height
5
area( )
main
tb


Class:
Rectangle3
Instance:
width
4
height
5
area
this
main
tb

Linked Lists

- When you have data in order in an array

```
  0    1    2    3
 Bob  Ed   Sue
```

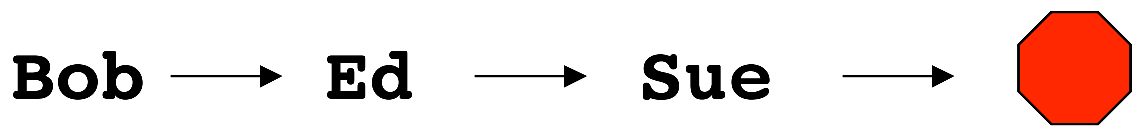
- You are storing who is 1st, 2nd, etc
  - Which largely changes when you insert

```
  0    1    2    3
 Bob  Ben  Ed   Sue
```

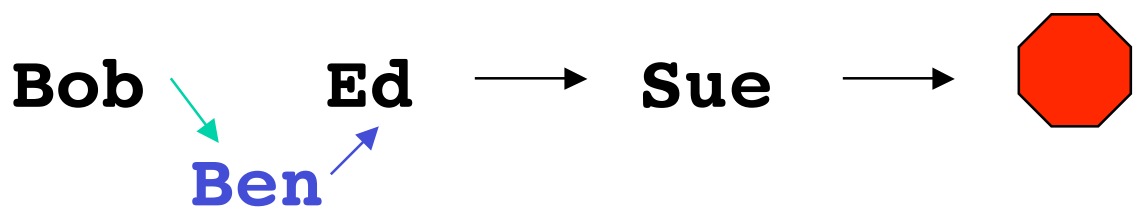
- What if all you care about is who is after who?

Linked Lists

- Suppose you store "who comes next"

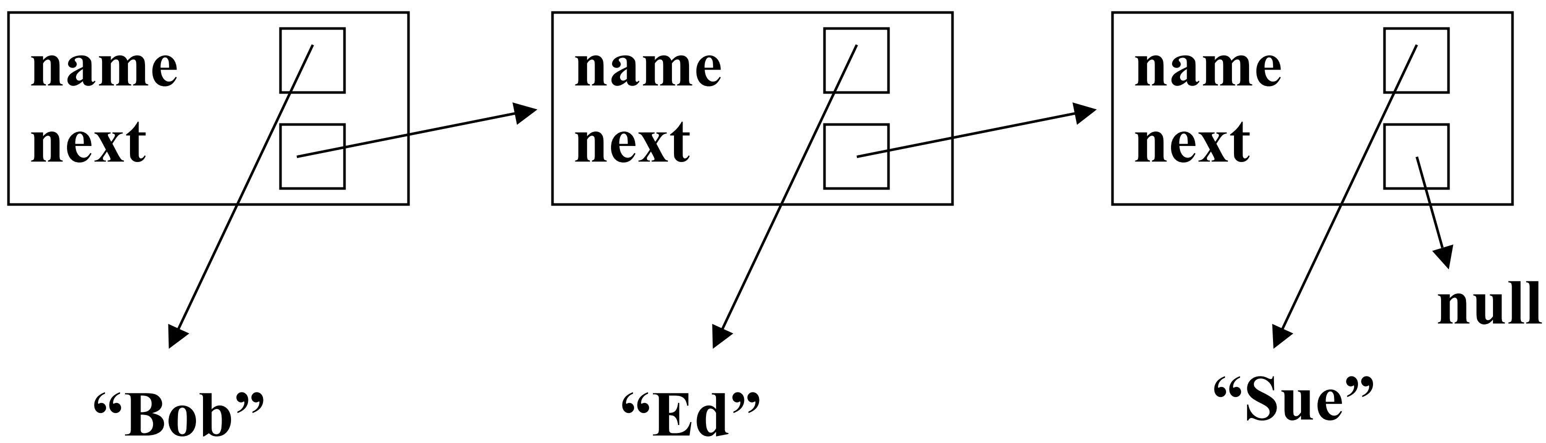


- When you insert, there is less to change



Storing “who is next”

- Class Node: instance variables for
  - A name
  - The next node in order

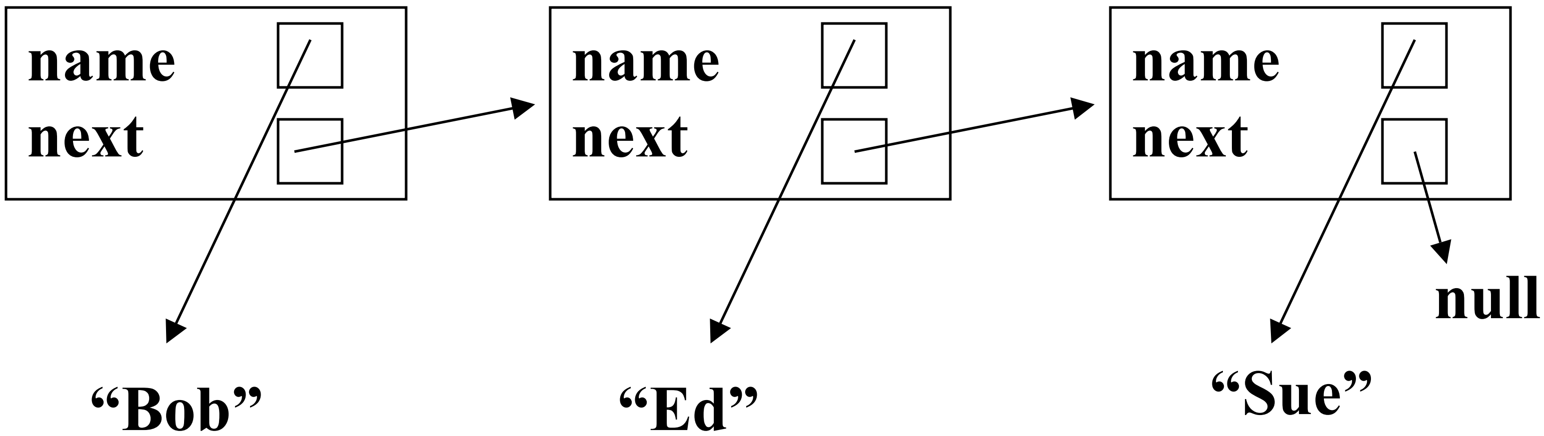


The Node Class

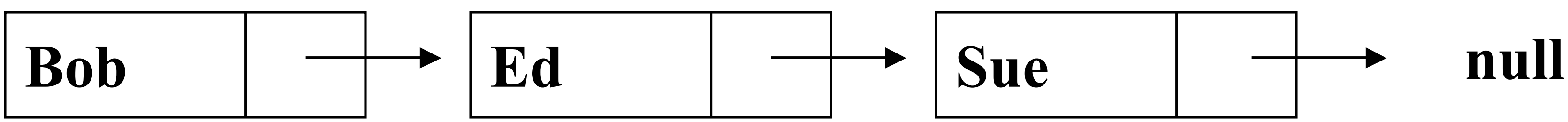
```
public class Node{
   private String name;
   private Node next;

   public Node(String nm, Node nxt){
    name = nm;
    next = nxt;
   }
 …
}
```

Storing “who is next”

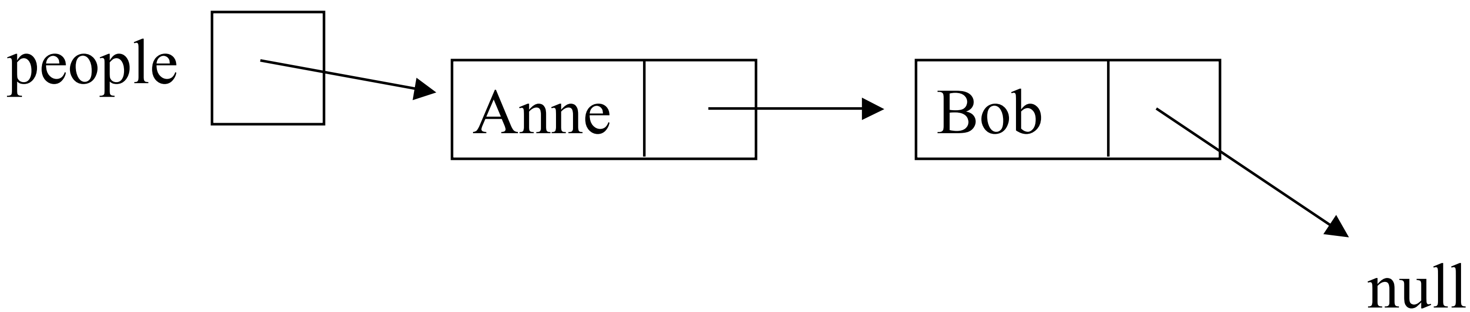


- Can also draw this way



Example Initial State

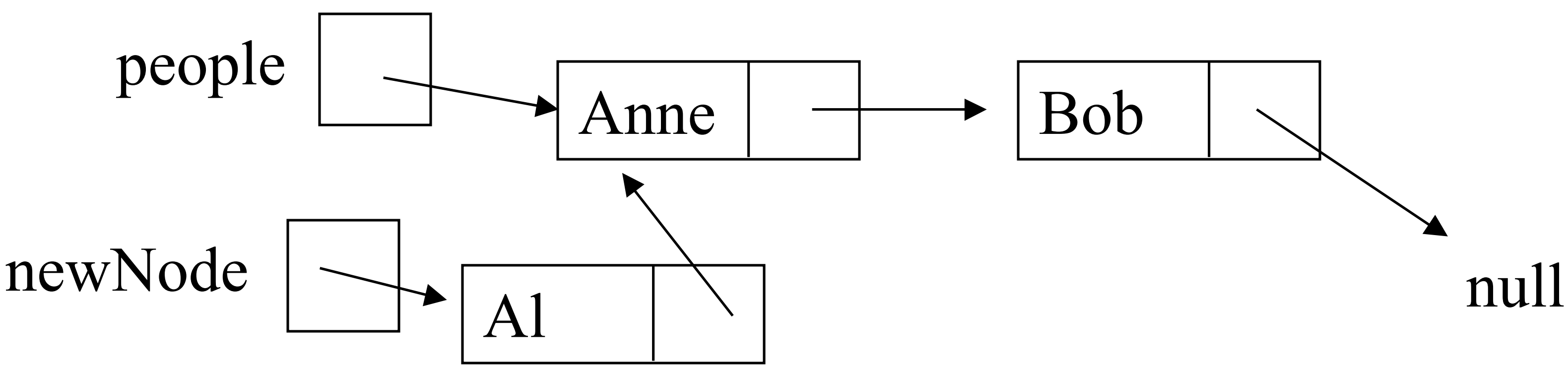
Node people; …



Insert at Head

Insert “Al” at head of list

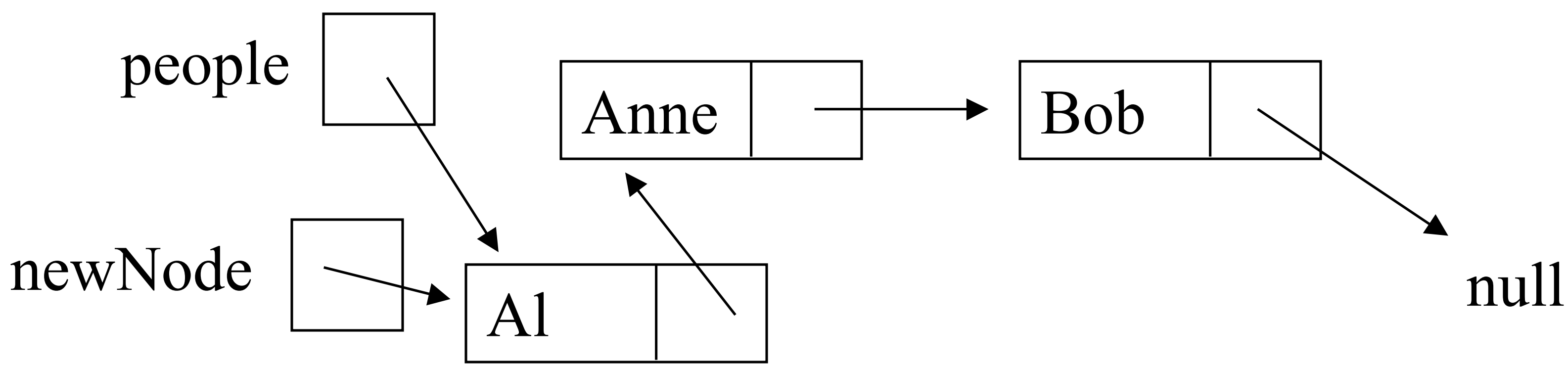
```
Node newNode = new Node(“Al”, people);
```



Insert at Head

Insert “Al” at head of list

```
Node newNode = new Node(“Al”, people);
people = newNode;
```

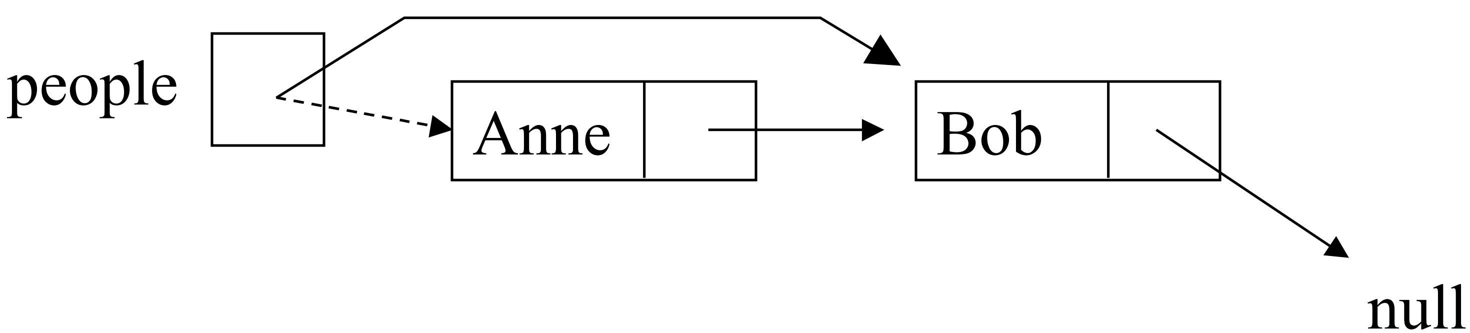


Cost of Insert At Head

- **For linked lists: O(1)**
- **For arrays: O(size)**

Remove at Head

```
people = people.next;
```



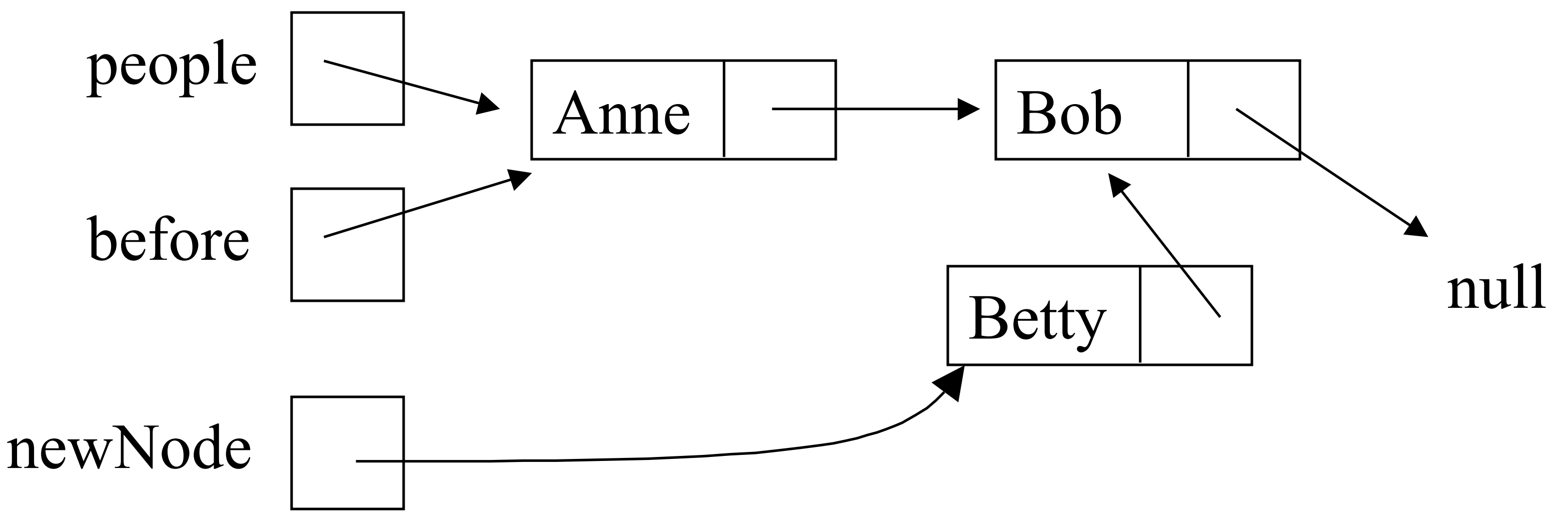
Cost of Remove At Head

- **For linked lists: O(1)**
- **For arrays: O(size)**

Insert after given node

Insert "Betty" after `before`

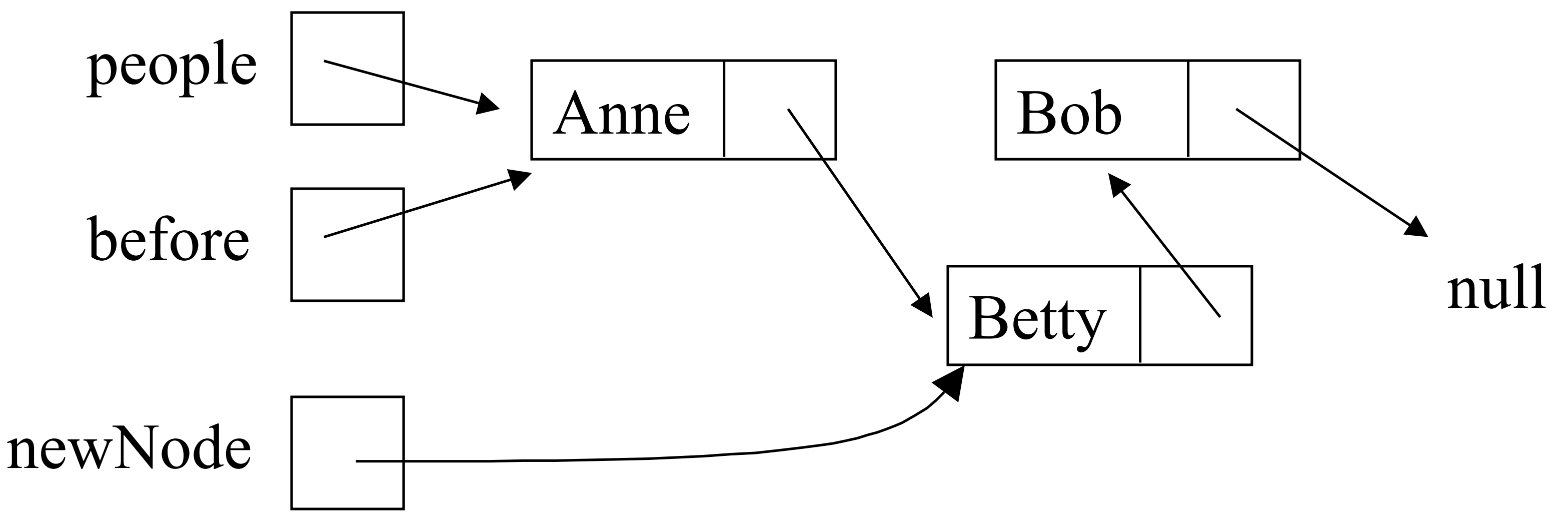
```
Node newNode =
    new Node(newName, before.next);
```



Insert after given node

Insert “Betty” after `before`

```
Node newNode =
    new Node(newName, before.next);
before.next = newNode;
```



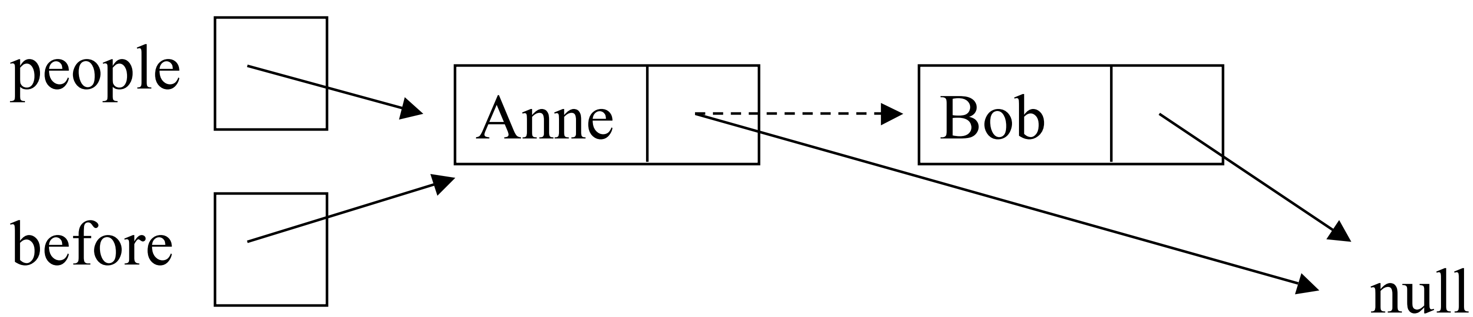
Cost of insert after given node

- For linked lists: O(1)
- For arrays: O(size)

Remove after given node

- Remove the node after `before`

```
before.next = before.next.next
```

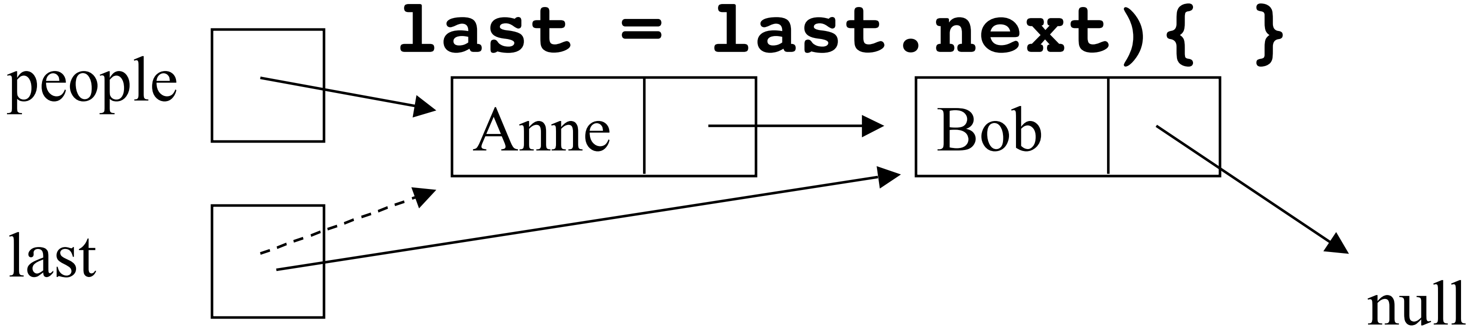


Cost of remove after given node

- **For linked lists: O(1)**
- **For arrays: O(size)**

Find last

```
Node last;
if (people == null){
   last = null;
} else {
    for (last = people;
         last.next != null;
         last = last.next){ }
```



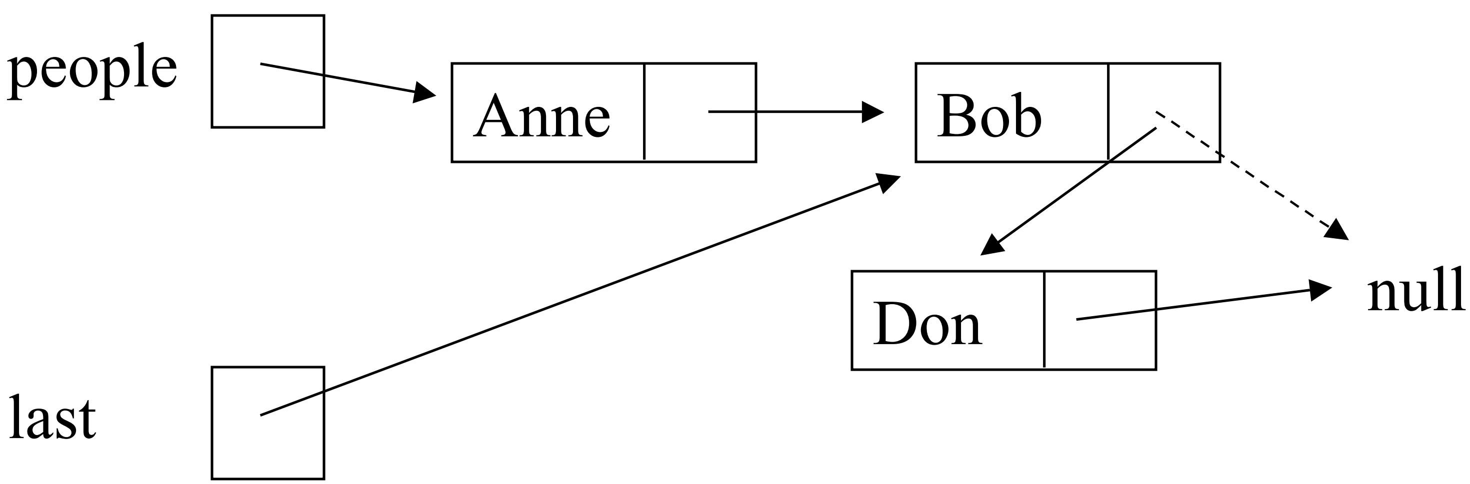
Cost of find last

- **For linked lists: O(size)**
- **For arrays: O(1)**

Insert at End

Insert Don after Bob

```
… find last …
last.next = new Node("Don", null);
```



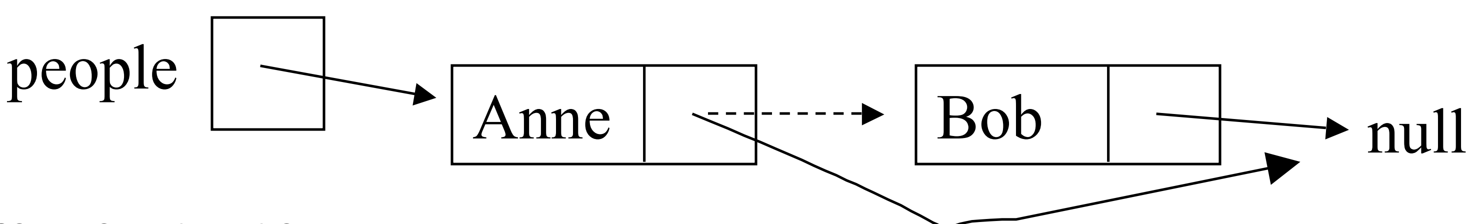
Cost of insert at end

Including find last

- For linked lists: O(size)
- For arrays: O(1)

Remove last

```
if(people == null){
} else {
  if (people.next == null){
     people = null;
  } else  {
     Node place;
       for(place = people;
         place.next.next != null;
         place = place.next;){ }
     place.next = null;
} }
```

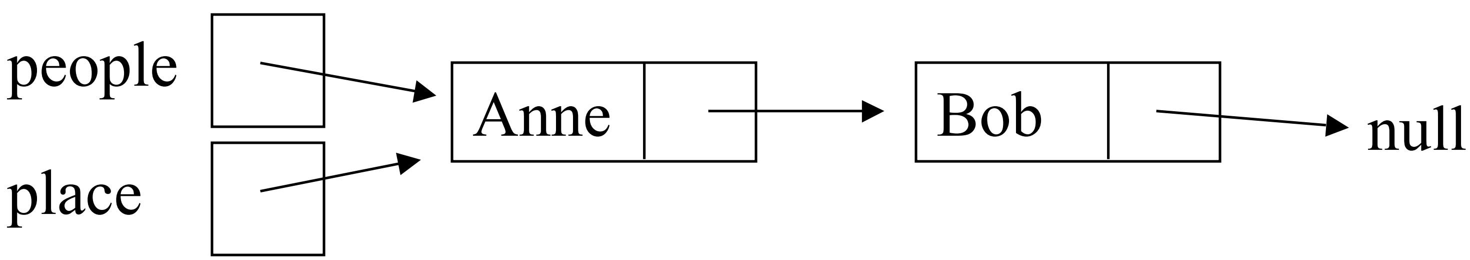


Cost of remove last

- **For linked lists:O(size)**
- **For arrays: O(1)**

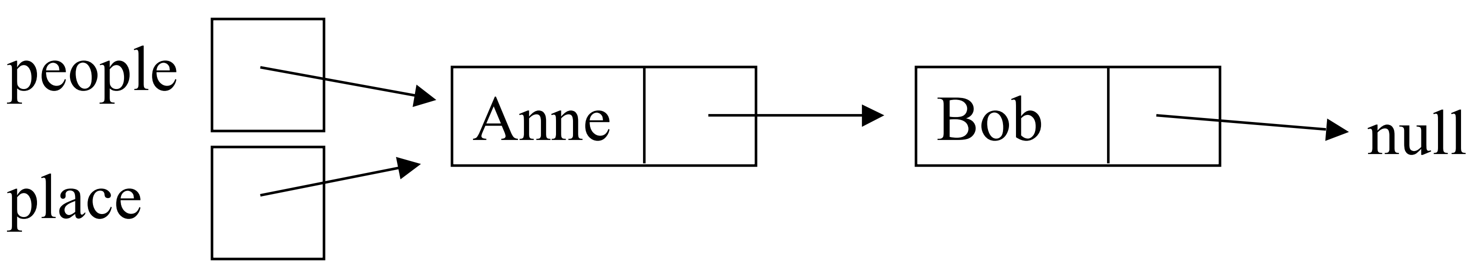
Find element i

```
Node place = people;
for (k = 0;
      place != null && k < i;
      k++){
   place = place.next;
}
```



Find by data

```
Node place;
for (place = people;
      place != null &&
       ! place.name.equals(target);
      place = place.next)
      { }
```



Generic Lists

- The code for, eg, insert at head is very much the same for lists of ints and lists of doubles. Why write it twice if you need both?
- Solution: generic types: see Node.java, LinkedList.java
- See also

http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf