CS 361
Data Structures & Algs
Lecture 6

Prof. Tom Hayes
University of New Mexico
09-09-2010

To do:

- Quizzes from Tuesday
- Reading: up to sec 2.4 done.
- Programming Assignment 1 - due Monday
- Written Assignment 2: problems
  1.8, 2.1, 2.2, 2.3, 2.4, 2.5*, 2.6, 2.7, 2.8*
  *:tricky. +:challenging
  Quiz: 2 weeks from today.

Quizzes

Grade distribution: one 20, sixteen 10's, two 9's, one 6, fourteen ≤2. (three absent)

The problem?

(a) almost nobody got anywhere on problem 2.

(b) Fallacy: proof by example.

(c) assignment was not done.

Let's go over the correct answers now.

Re Programming Assn

(1) You must not modify the test.sh file.

(2) Your code must work with the existing test.sh file.

(3) Ditto for GSTest.java

(4) problems with test.sh?

  (a) ssh to brown.cs.unm.edu

  (b) diagnose & fix

Re Next Written Assn

Work together!

(a) in small groups, to come up with solutions

(b) online discussion, to check solutions, and test whether you know the difference between a right and wrong answer.

This assignment is long and hard! Start early!

Implementing the Gale-Shapley algorithm

Goal: loop iterations run in O(1) time.

Not a practical issue for the programming assignment, but perhaps for a large dating service.

2nd goal: learn to choose data structures.

Gale-Shapley, pseudocode

Initially, all men and women are free
While (exists m in {free men})
    Let w = “top” woman in m’s pref list
    If (w is free): (m,w) become engaged
    else if (w engaged to m’ but prefers m) {
      m’ becomes free again.
      m’ removes w from his pref list.
     (m,w) become engaged.
} else { // w engaged to m’, prefers m’
    m stays free, removes w from pref list
}   // (see page 6 in text).

Implementation issues

Broad considerations:

- What should be the Objects? What should be the Methods?
- Having decided this, how should these be implemented?
  - How is data stored?
  - Algorithms for the Methods
  - Efficiency: bottlenecks

Objects 1

free_men: the men who are not engaged.

Operations supported:

take_next(): get one free man, remove from free_men.

add(m): adds m to free_men

hasNext(): true if free_men is not empty

Objects 2

fiance(w): which man is w engaged to? Initially null.

M_pref(m): supports operations:

take_first(): removes top-ranked woman (un-proposed to) from list, and passes her as return-value.

w_pref(w,m1,m2): does w prefer m1 to m2?

```
Init:  free_men = all men.  fiance = all null.
M_pref, w_pref initialized using input.
While (free_men.hasNext())
    m = free_men.take_next()  // removed
    w = M_pref(m).take_first()
    if (fiance(w)==null): make-engaged(w,m)
    else {
     m’ = fiance(w)
     if (w_pref(w,m,m’))
       {winner = m, loser = m’}
    else {winner = m’, loser = m}
    make-engaged(w,winner)
    free_men.add(loser)
    }
```

free_men

Data: stores a bunch of men

Operations:

take_next(): get one free man, remove from free_men.

add(m): adds m to free_men

hasNext(): true if free_men is not empty

Look at Collection<E> interface in Java. But, which Collection should we use?

Only Once!

```
Init:  free_men = all men.  fiance = all null.
M_pref, w_pref initialized using input.
While (free_men.hasNext())
    m = free_men.take_next()  // removed
    w = M_pref(m).take_first()
    if (fiance(w)==null): make-engaged(w,m)
    else {
        m' = fiance(w)
     if (w_pref(w,m,m'))
       {winner = m, loser = m'}
    else {winner = m', loser = m}
    make-engaged(w,winner)
    free_men.add(loser)
    }
```

roughly n^2 times each (equal importance)

Arrays vs Linked Lists

Array: Fixed size. (well...)

Linked List: Dynamically resizeable.

Array: Get i'th element in O(1) time.

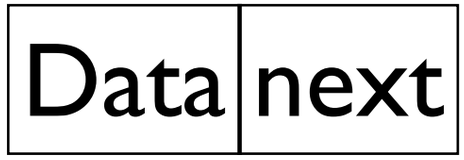
LL: Takes i steps to get i'th element.

Delete/add at middle: perhaps faster for LL

Delete/add at end: O(1) time for both,
UNLESS exceed size of the array.

(Singly) Linked Lists

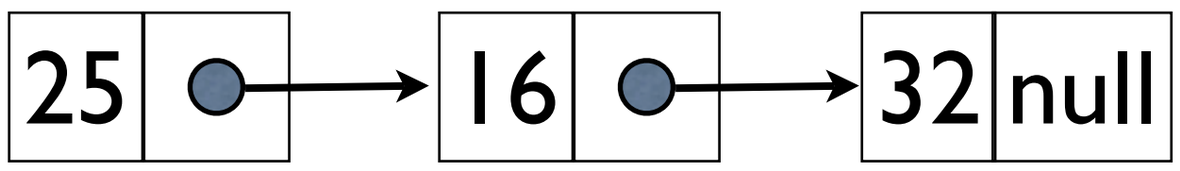
Two fields:



Data: stores list element

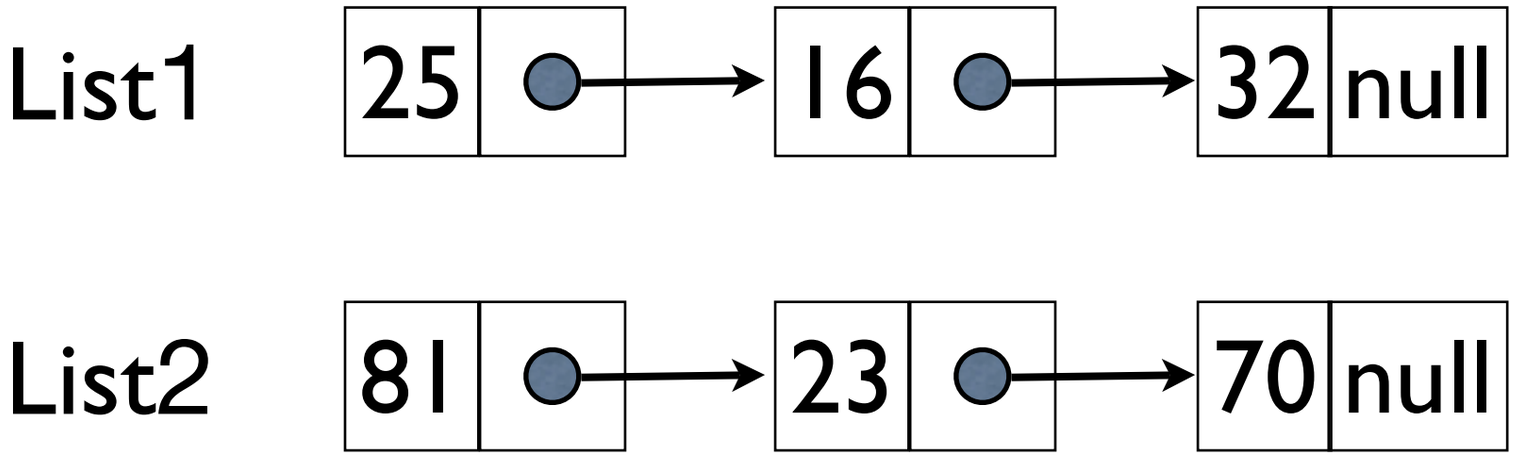
Next: points to next item in list

For example, to store the (ordered) list 25, 16, 32, we would have:



Concatenating Lists

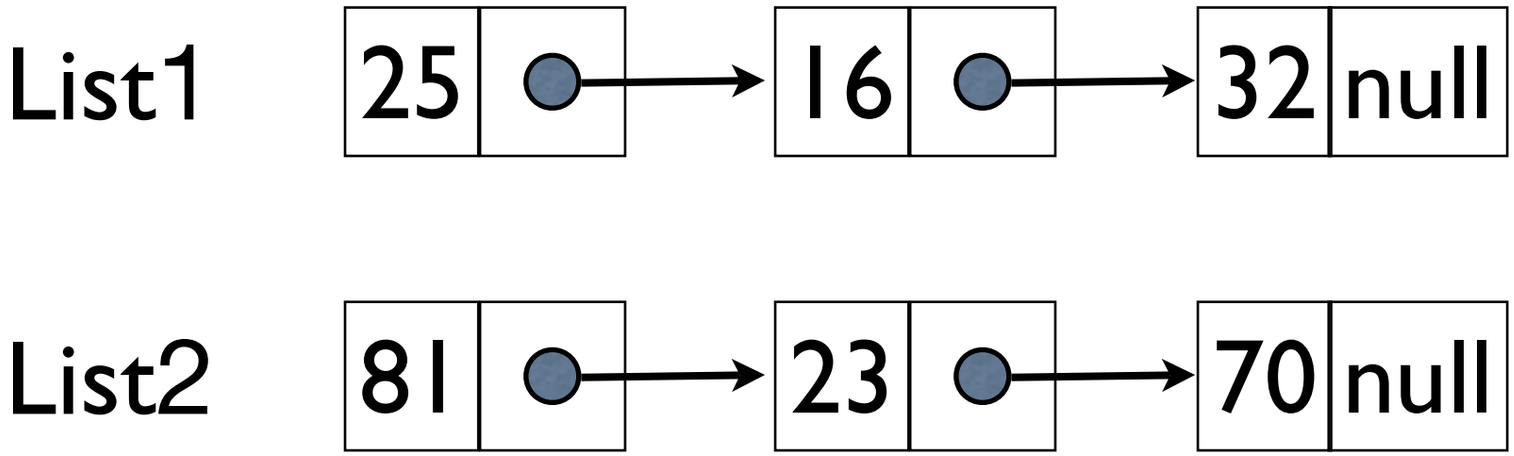
How would we concatenate List2 on the end of List1?



(a) Find end of List1

Concatenating Lists

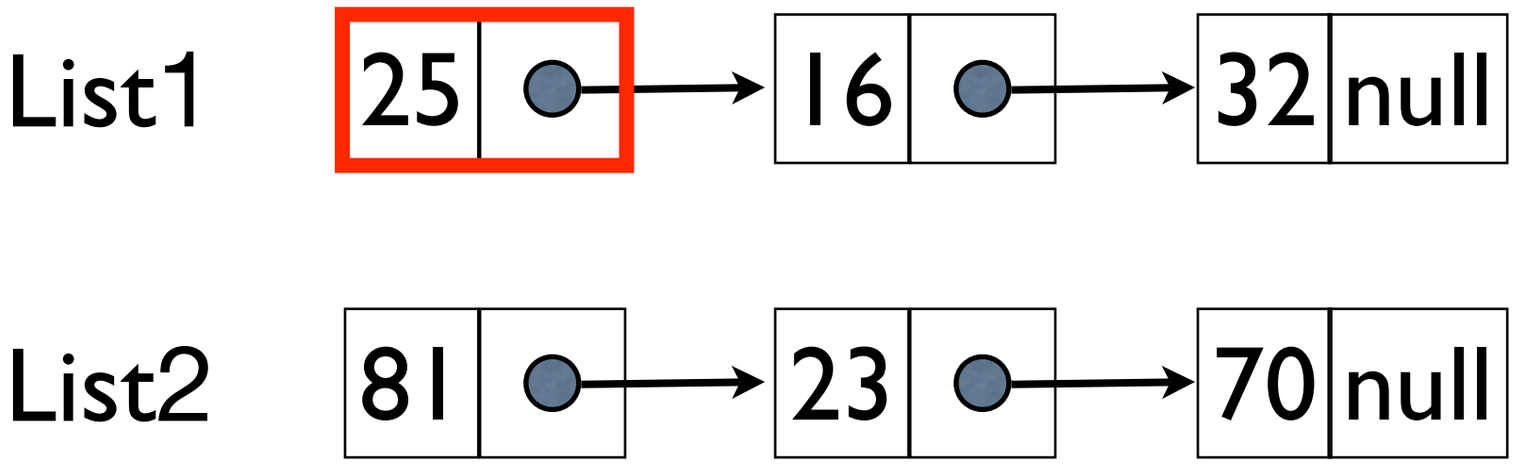
How would we concatenate List2 on the end of List1?



(a) Find end of List1

Concatenating Lists

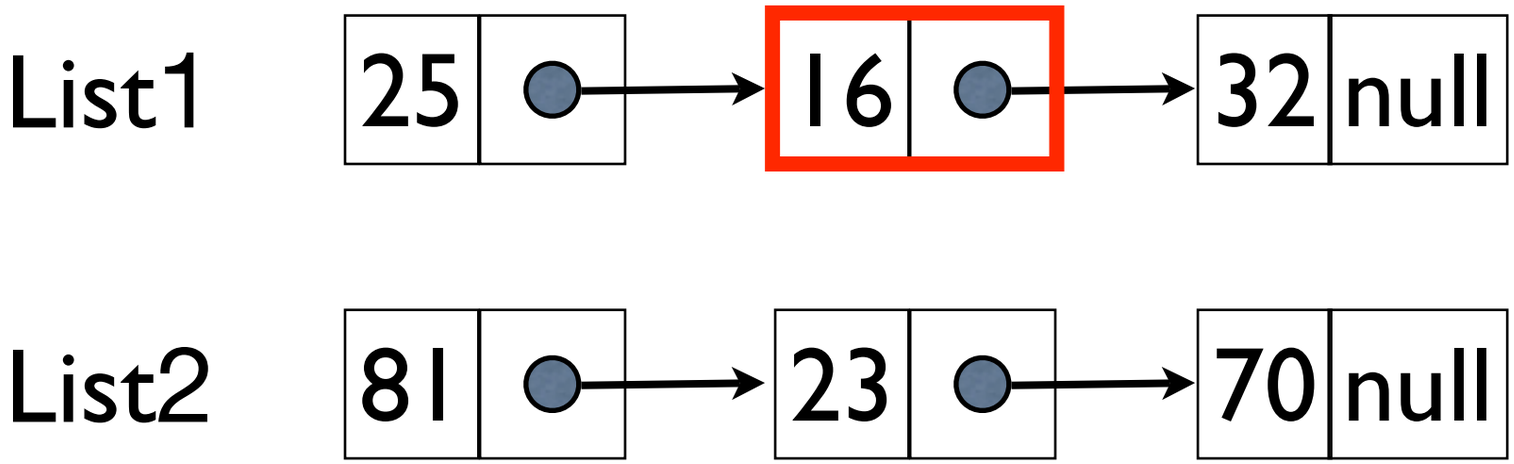
How would we concatenate List2 on the end of List1?



(a) Find end of List1

Concatenating Lists

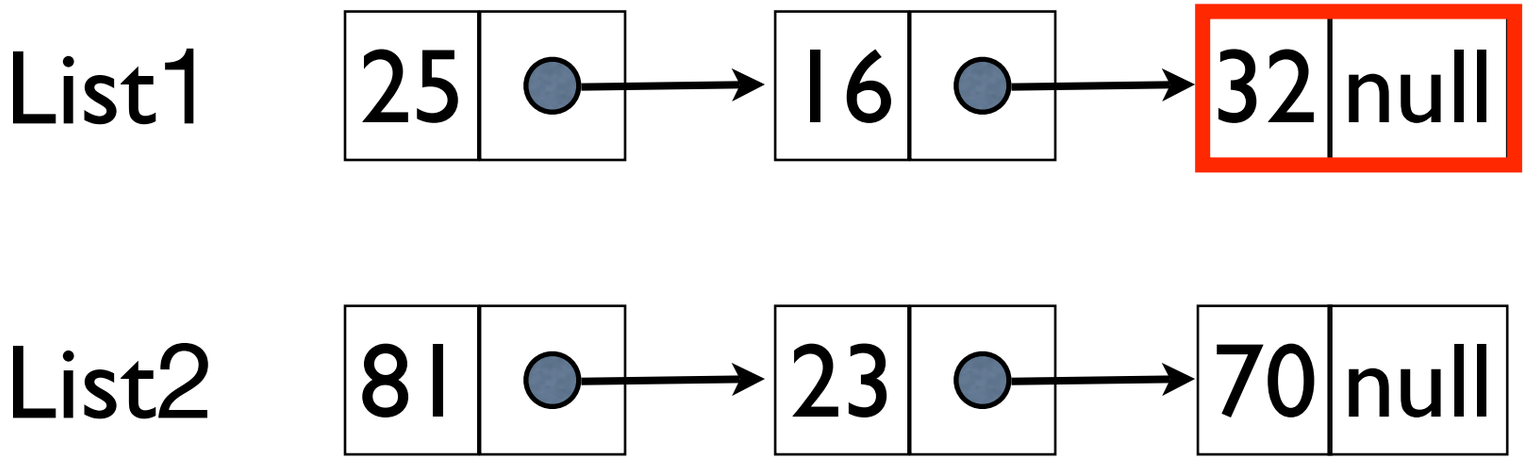
How would we concatenate List2 on the end of List1?



(a) Find end of List1

Concatenating Lists

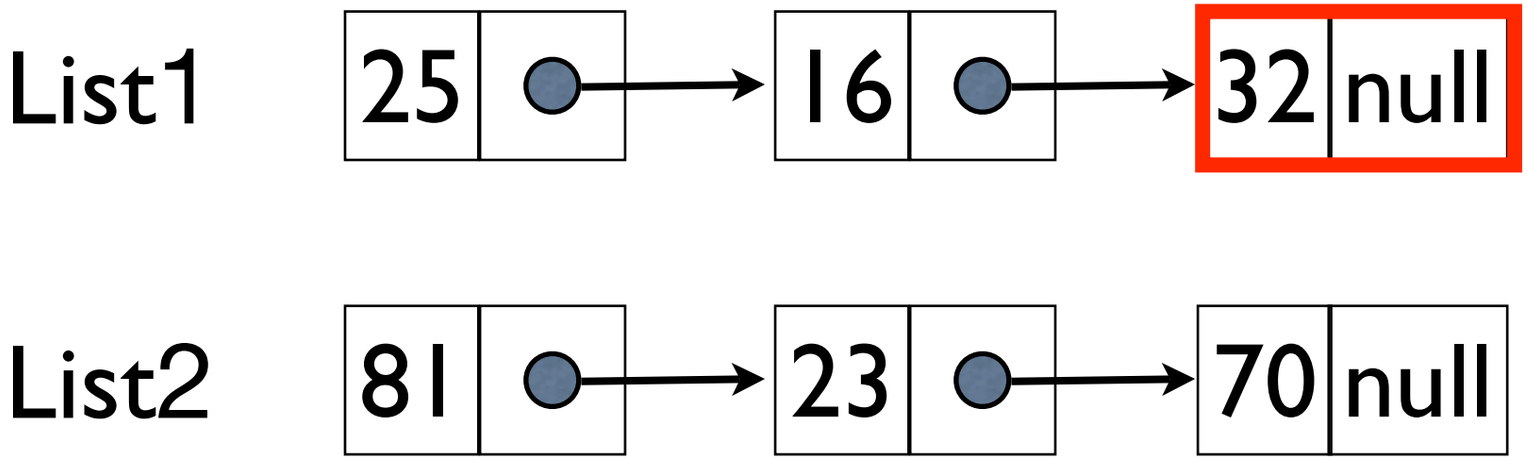
How would we concatenate List2 on the end of List1?



(a) Find end of List1

Concatenating Lists

How would we concatenate List2 on the end of List1?

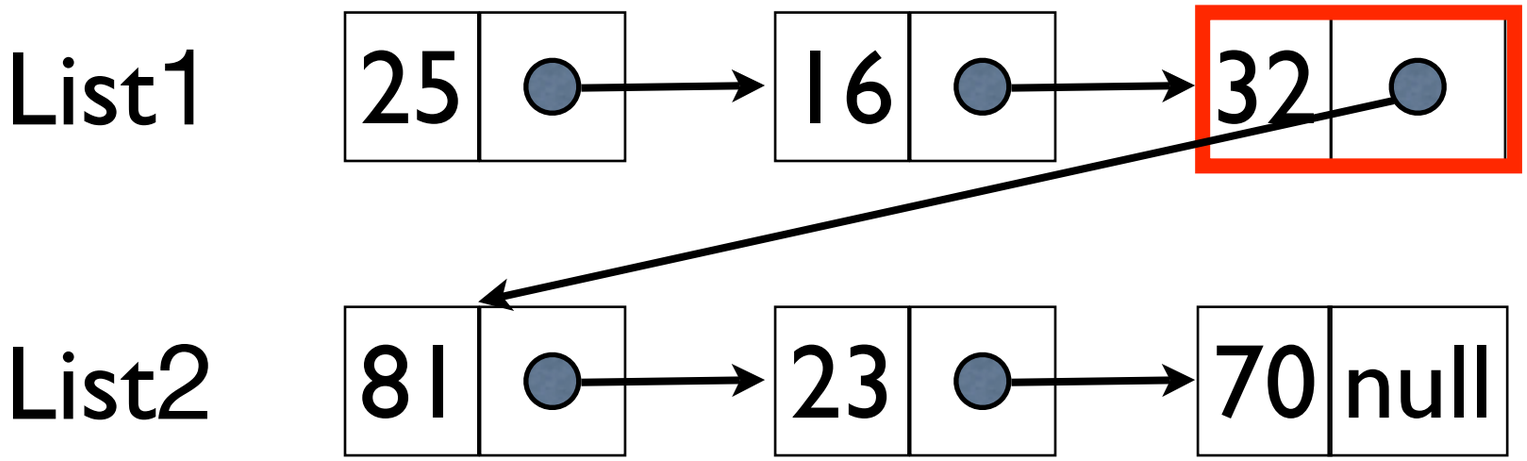


(a) Find end of List1

(b) Make it point to start of List2

Concatenating Lists

How would we concatenate List2 on the end of List1?



(a) Find end of List1

(b) Make it point to start of List2

Implementing G-S

Use a LinkedList for free_men.

(Alternatively, could use a Stack or Queue, etc)

Why? It gives a guaranteed O(1) time for the operations we need: add or delete one element from the front of the list.

No such guarantee for Array class.

Implementing G-S

Use an array for fiance.

Why? We want fast "random access." That is, we want to be able to look up fiance(w) in time O(1).

This is exactly what arrays are good for.

Note: the size of array is not going to change during the algorithm. So we can use a regular [ ] array rather than an Array object.

Implementing G-S

Use a linked list for M_pref(m).

Why? Want to "extract from front" in time O(1).

Can do this with an array too: just keep a pointer to where we are in the array.

(see board)

Question: Why would we prefer one or the other?

Implementing G-S

Use a linked list for M_pref(m).

Why? Want to "extract from front" in time O(1).

Can do this with an array too: just keep a pointer to where we are in the array.

(see board)

Question: Why would we prefer one or the other? Answer: Whichever one is already coded for us! (i.e. LinkedList, or Stack, etc)

Implementing G-S

Use a linked list for M_pref(m).

M_pref should be an array of linked lists, indexed by men.

Why?

Implementing G-S

Use a linked list for M_pref(m).

M_pref should be an array of linked lists, indexed by men.

Why? Want fast “random access” to the list for man m. For “fast,” read O(1).

Implementing G-S

What about w_pref(w,m1,m2)?
(“true” if w prefers m1 to m2.)

We could just build up a triply-indexed array of booleans! Cost? (see code)

Only Once!

```
Init:  free_men = all men.  fiance = all null.
M_pref, w_pref initialized using input.
While (free_men.hasNext())
    m = free_men.take_next()  // removed
    w = M_pref(m).take_first()
    if (fiance(w)==null): make-engaged(w,m)
    else {
     m’ = fiance(w)
     if (w_pref(w,m,m’))
       {winner = m, loser = m’}
    else {winner = m’, loser = m}
    make-engaged(w,winner)
    free_men.add(loser)
    }
```

roughly n^2 times

Implementing G-S

What about w_pref(w,m1,m2)?
("true" if w prefers m1 to m2.)

We could just build up a triply-indexed array of booleans! Cost? (see code)

$\Theta(1)$ cost per lookup. So $O(n^2)$ cost in loop body.

Initialization cost: $\Theta(n^3)$. Too much!

Implementing G-S

What about w_pref(w,m1,m2)?
(“true” if w prefers m1 to m2.)

Idea 2: Could just keep the ordered list of preferences. Cost? (see code)

Only Once!

```
Init:  free_men = all men.  fiance = all null.
M_pref, w_pref initialized using input.
While (free_men.hasNext())
    m = free_men.take_next()  // removed
    w = M_pref(m).take_first()
    if (fiance(w)==null): make-engaged(w,m)
    else {
     m’ = fiance(w)
     if (w_pref(w,m,m’))
       {winner = m, loser = m’}
    else {winner = m’, loser = m}
    make-engaged(w,winner)
    free_men.add(loser)
    }
```

roughly n^2 times

Implementing G-S

What about w_pref(w,m1,m2)?
("true" if w prefers m1 to m2.)

Idea 2: Could just keep the ordered list of preferences. Cost? (see code)

Initialization cost: $O(n^2)$. cool.

O(n) cost per lookup, since have to search through the list. So $O(n^3)$ cost in loop body. Not good enough!

Implementing G-S

What about w_pref(w,m1,m2)?
("true" if w prefers m1 to m2.)

Idea 2: Could just keep the ordered list of preferences. Cost? (see code)

Initialization cost: $O(n^2)$. cool.

O(n) cost per lookup, since have to search through the list. So $O(n^3)$ cost in loop body. Not good enough!

Q: why can't we do binary search?

Implementing G-S

What about w_pref(w,m1,m2)?
("true" if w prefers m1 to m2.)

Idea 3: Store w's "ranking" of each man in an array. Cost?

Implementing G-S

What about w_pref(w,m1,m2)?
("true" if w prefers m1 to m2.)

Idea 3: Store w's "ranking" of each man in an array. Cost?

Per lookup: O(1). Testing:

is (w_ranking[w][m1] < w_ranking[w][m2]) ?

Initialization: O(n) per woman, so $O(n^2)$.
Can do in one pass through the input.

Summary

free_men: LinkedList (or Stack)

fiance: array of men, indexed by women

M_pref: array of LinkedList (or Stack, etc)

w_pref: instead, replace with

w_ranking: array of ints, indexed by [w][m].

Related reading: Section 2.3