CS112: Data Structures

Lecture 02

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 Page

- http://sakai.rutgers.edu/

  Login with NETID

  - Policies
  - Syllabus
  - Assignments
  - Lecture notes
  - etc....

- You are assumed to know anything posted.

Review

What is a Data Structure

- A way to store multiple pieces of data
- Stores some relationship among the pieces

What to know about a DS

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

Review
Asymptotic Costs

- Problem: actual cost depends on many details
- We want a measure of cost that does not depend on these

Review: 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)**
  - **If O(f) > O(g), if n gets big enough f(n) will be larger than k * g(n)**

Example

Arrays a1, a2 in increasing order, length=n

Do they have any common element?

```
 int i1 = 0;  int i2 = 0;
while (i1 < n && i2 < n && a1[i1] != a2[i2]){
    if (a1[i1]<a2[i2]){i1++;}
       else {i2++;}}
if (i1 = = n || i2 = = n){
    System.out.println("no");} else {
    System.out.println("yes ");
```

Which Operations to Count?

Count should model time of algorithm

- **Most frequent / inner loop**
- **Most time consuming**
- **Inherent in algorithm, not language**

- **Count a1[i1] != a2[i2]**

Size of input

- Number of ops = f(input size)
- How do we define size of input?
  - For this example: n

Worst / Average Case

- Worst case: $2n-2 = O(n)$
- Average case
  - Assume will find a match
  - Assume each sum $(i1 + i2)$ equally likely to be location of first match
  - Sum over all cases prob(case)*cost(case)

$$\sum_{s=0}^{2n-2} 1/(2n-2) * s$$
$$= (1/(2n-2)) \; * \; (2n-2)(2n-1)/2 = O(n)$$

Review: 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 $O(n^4)$

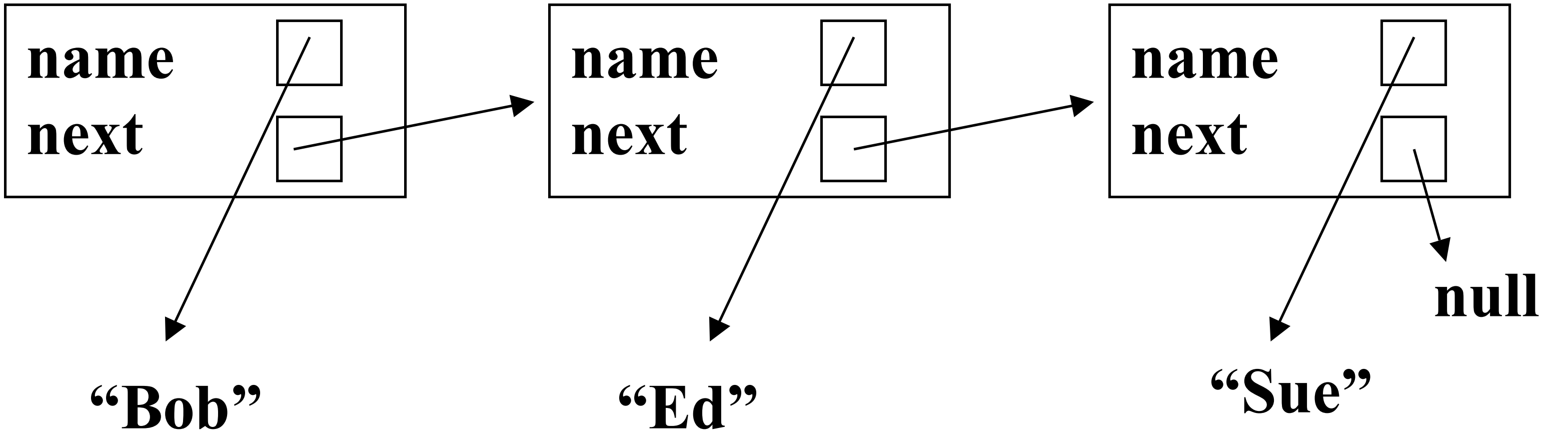
Review: Names for Big O

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

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

Review: Linked Lists

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

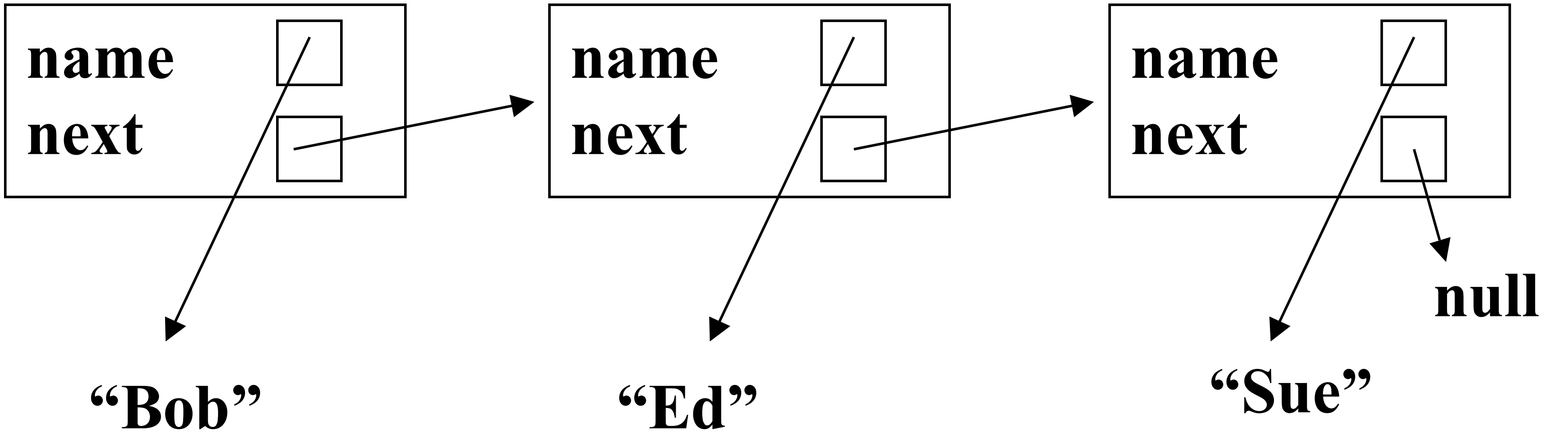


The Node Class

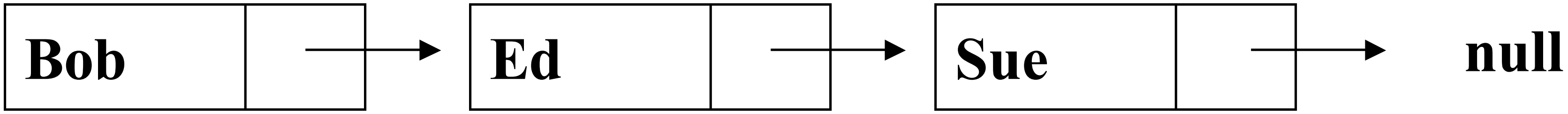
```
public class Node{
    private String name;
    private Node next;

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

Storing “who is next”



- Can also draw this way



Operations on linked lists

- **Insert at head**
- **Remove at head**
- **Insert after given node**
- **Remove after given node**
- **Find last**
- **Insert at end**
- **Remove last**
- **Find element i**
- **Find by data**

Generic Lists

- Problem: suppose you want to have a list of Strings and a list of ints and …
- Class declarations and methods are almost identical
- Solution in older java: list of Object
  - But give up ability for compiler to check
- Solution in java 1.5: “generics”
  - Class & method definitions parameterized by type

Generic List

```
public class Node<E> {
    private E data;
    private Node<E> next;

    public Node<E>(E dat, Node<E> nxt){
     data = dat;
     next = nxt;
    }
   public E getHead(Node<E> head){
       E headData = head.data;
     return headData;} …
```

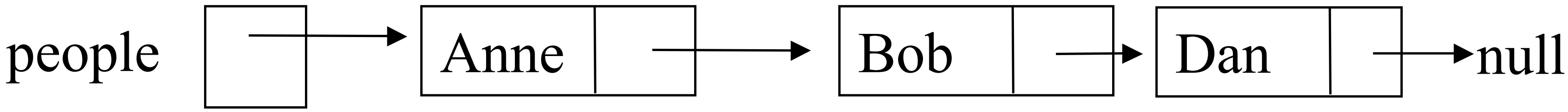
Generic List

In some method:

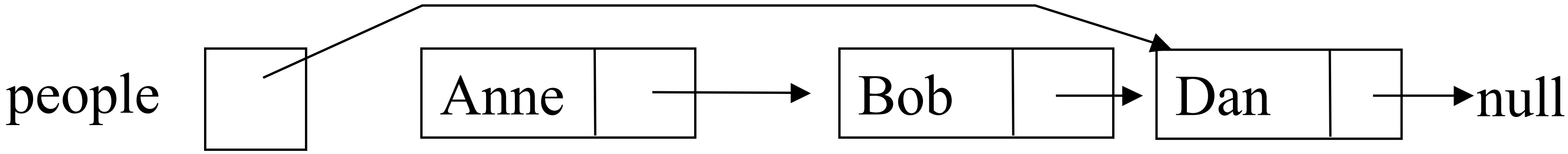
```
 Node<String> n1 = new Node<String>;
   …
   String name = n1.getHead( );
```

Circular Lists

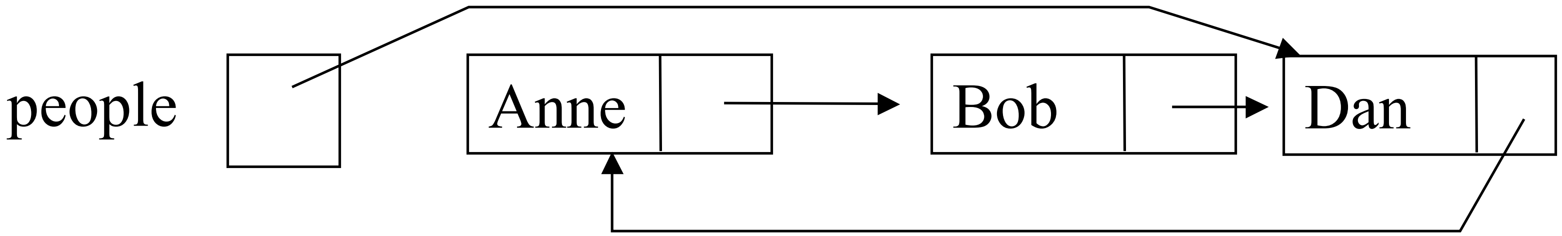
- **Problem: Cost to Access Tail**



- **Solution: point to tail, not head**

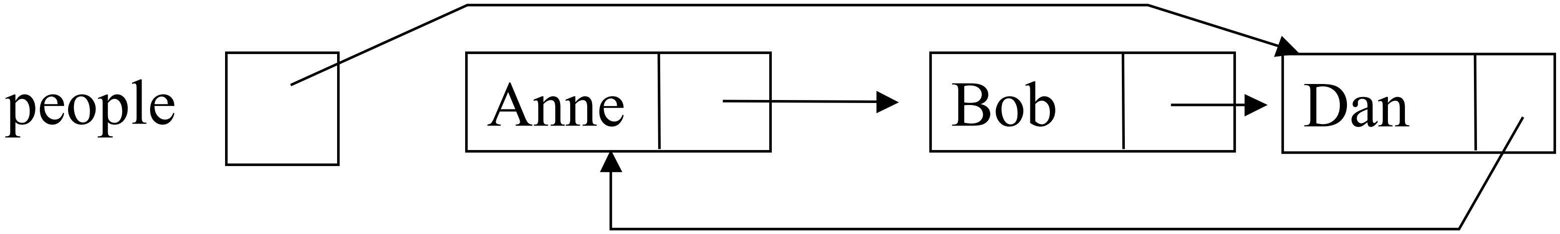


- **Problem: access to head. Solution:**



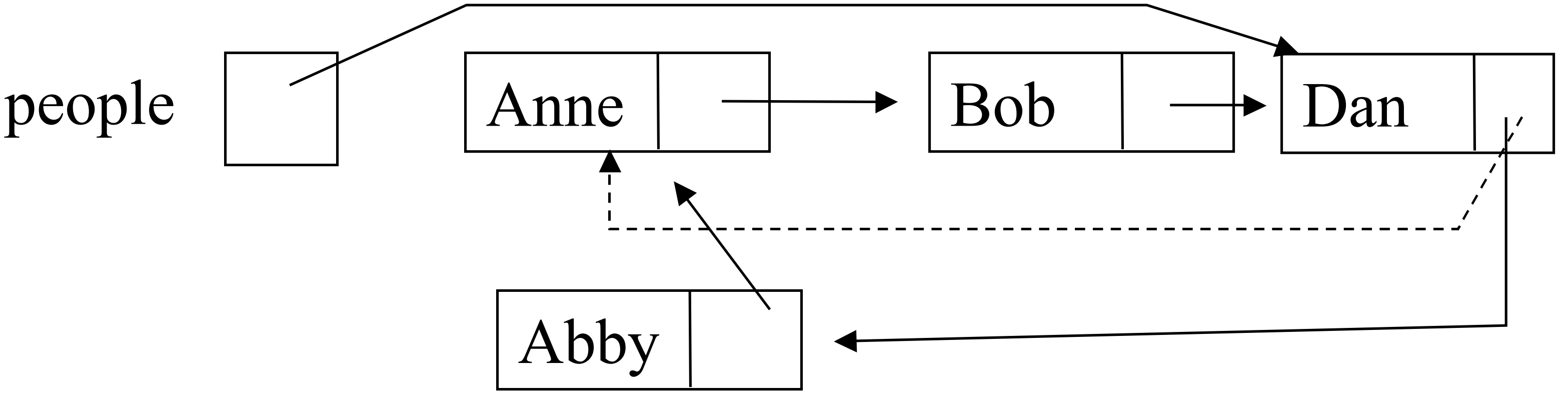
Circular List

- First node is ??
- Variable place points at last node when??

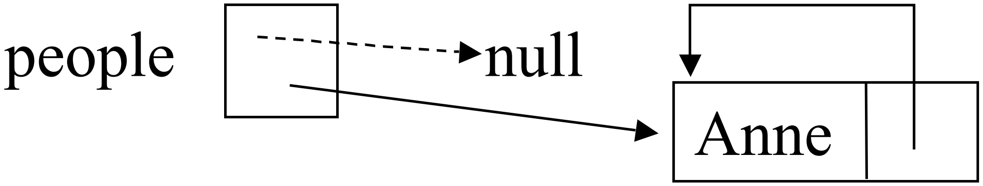


Insert at Head

- **List not empty**



- **List empty**



Insert at Head

```
if(people == null){
    people =  new Node(newName, null);
    people.next = people;
   } else {
    Node newNode= new Node(newName,
                            people.next);
    people.next = newNode;
   }
```

Delete Head

- You write it for circular lists
- Hint: 3 cases:
  - people empty
  - one node
  - more nodes

Other CLL Methods

- See resources => Java examples => fancy lists

DLL Methods

- **See resources => Java examples => fancy lists**

Dummy Headers

- **Problem: delete head is different that delete elsewhere in list**
  - **Change pointer to list as a whole vs change the next field of some node**
- **Solution: Keep an extra "dummy" node at the head of the list**

Iterators

- Abstract data type: a container
  - E.g. array or linked list
  - Can do mostly the same things with them, main difference is cost
  - Problem: one of the things I want to do is go through the data items one by one

Processing Data Items

- Normal way to handle same-process-different-structure problem is with a method that is defined appropriately for each class
  - Same name and abstract behavior
  - Different code
- But what would be the interface?
  - What changes from call to call? Pieces of progam ☹

Solution

- Instead of a method to do whole loop, have methods you can use to build the loop
  - hasNext
  - getNext
- State: an object
  - Represents a particular instance of iteration
  - Initialized by new

Abstract List Traversal

-
  ```
  while (list.hasNext()) {
        print(list.getNext().data);
  }
  ```

- list could be an Array:

  ```
        hasNext()  { return (i != list.length) }
        getNext()  { i++;  return list[i]; }
  ```

Abstract List Traversal

-
  ```
  while (list.hasNext()) {
        print(list.getNext().data);
  }
  ```

- list could be a LinkedList:

  ```
        hasNext()  { return (curr != null) }
        getNext()  { curr = curr.next;
                     return curr; }
  ```

Iterators

- See StringList.java and StringListIterator.java