Linked Lists

CS 121: Data Structures

START RECORDING

Outline

- Attendance quiz
- Linked lists
- Linked lists activities
- Homework 5 check-in

Attendance Quiz

Attendance Quiz: Big Oh, Big Ω, and Big Θ

- Scan the QR code, or find today's attendance quiz under the "Quizzes" tab on Canvas
- Password: to be announced in class

Is n^2...	Upper Bound $O(\ )$	Lower Bound $\Omega(\ )$	Tight Bound $\Theta(\ )$
n^3			
n^2			
n			
$\log(n)$			
1			

Why use Linked Lists?

- Using an array, what is the time complexity of:
  - Updating an element, given the index? $\Theta(1)$
  - Growing the length of the array? $\Theta(n)$
  - Locating an element if the array isn't sorted? $\Theta(n)$
  - Locating an element if the array is sorted? $\Theta(\log(n))$
- Can you think of a program where array length isn't known ahead of time?
  - Linked lists grow in constant-time (i.e., $\Theta(1)$)

12. Stacks and Queues

- APIs
- Clients
- Strawman implementation
- **Linked lists**
- Implementations

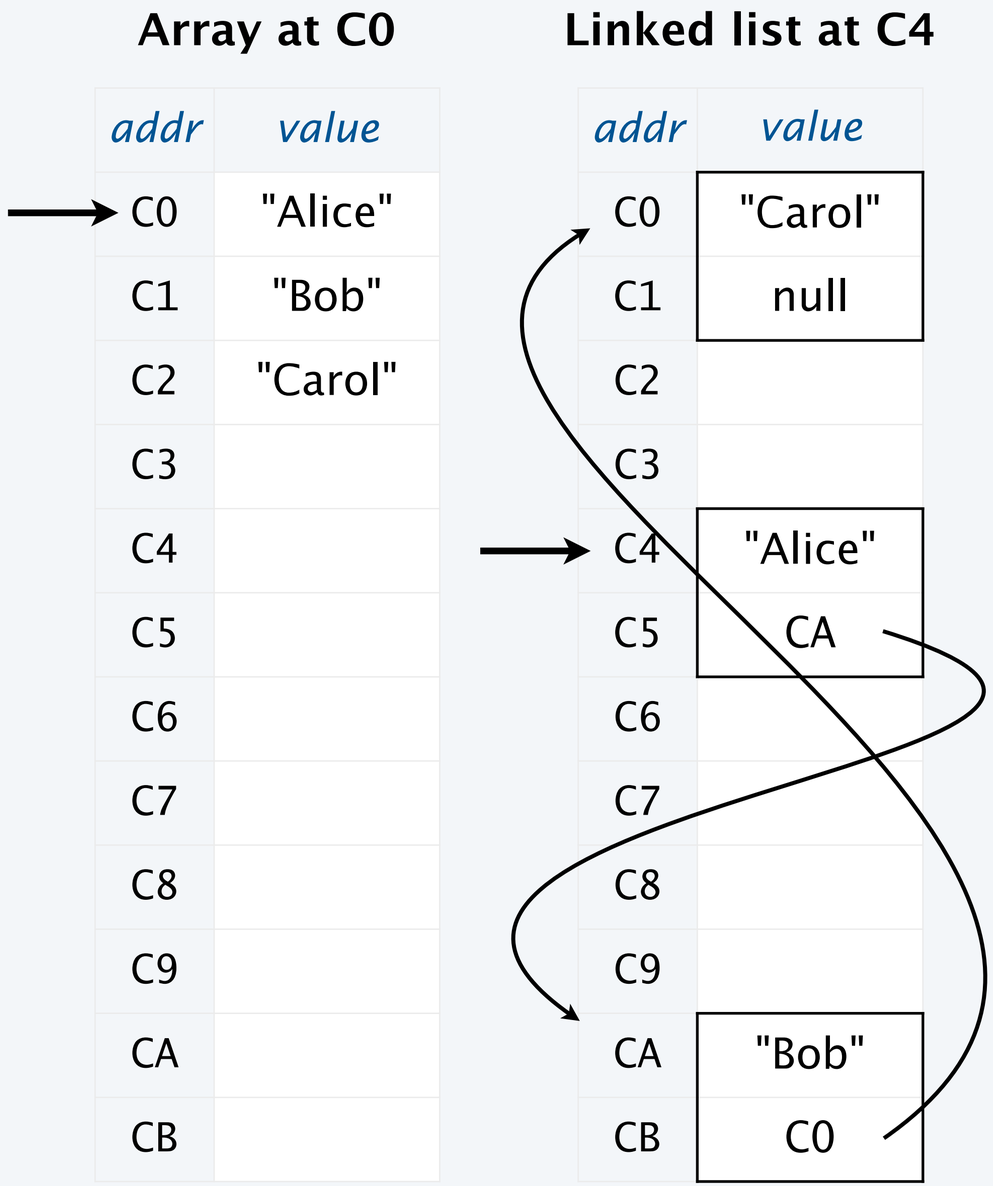
Data structures: sequential vs. linked

Sequential data structure

- Put objects next to one another.
- Machine: consecutive memory cells.
- Java: array of objects.
- Fixed size, arbitrary access. ⟵ *i*th element

Linked data structure

- Associate with each object a link to another one.
- Machine: link is memory address of next object.
- Java: link is reference to next object.
- Variable size, sequential access. ⟵ *next* element
- Overlooked by novice programmers.
- Flexible, widely used method for organizing data.



Simplest singly-linked data structure: linked list

Linked list

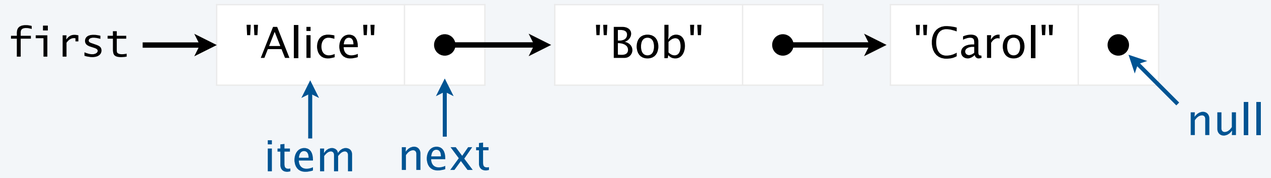
- A recursive data structure.
- Def. A *linked list* is null or a reference to a *node*.
- Def. A *node* is a data type that contains a reference to a node.
- Unwind recursion: A linked list is a sequence of nodes.

```
private class Node
{
  private String item;
  private Node next;
}
```

Representation

- Use a private nested class Node to implement the node abstraction.
- For simplicity, start with nodes having two values: a `String` and a Node.

A linked list

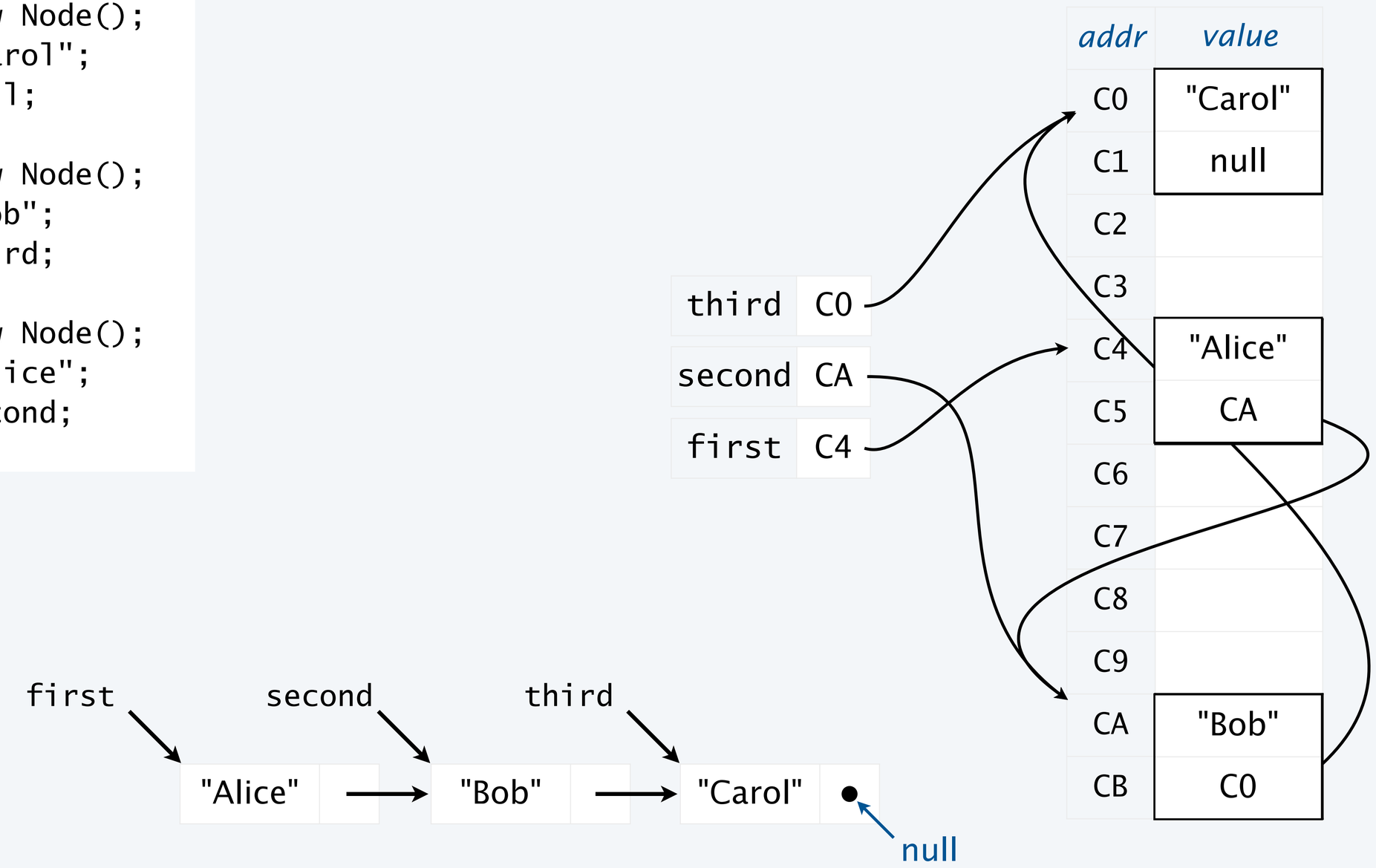


Building a linked list

```
Node third  = new Node();
third.item  = "Carol";
third.next  = null;

Node second = new Node();
second.item = "Bob";
second.next = third;

Node first  = new Node();
first.item  = "Alice";
first.next  = second;
```



List processing code

Standard operations for processing data structured as a singly-linked list

- Add a node at the beginning.
- Remove and return the node at the beginning.
- Add a node at the end (requires a reference to the last node).
- Traverse the list (visit every node, in sequence).

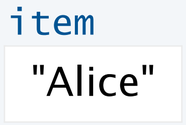
An operation that calls for a *doubly*-linked list (slightly beyond our scope)

- Remove and return the node at the end.

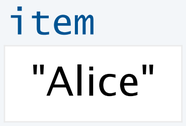
List processing code: Remove and return the first item

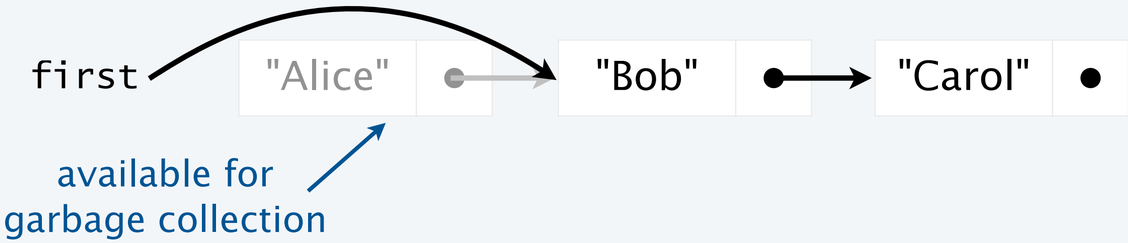
Goal. Remove and return the first item in a linked list `first`.


```
item = first.item;
```

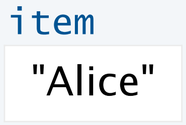



```
first = first.next;
```



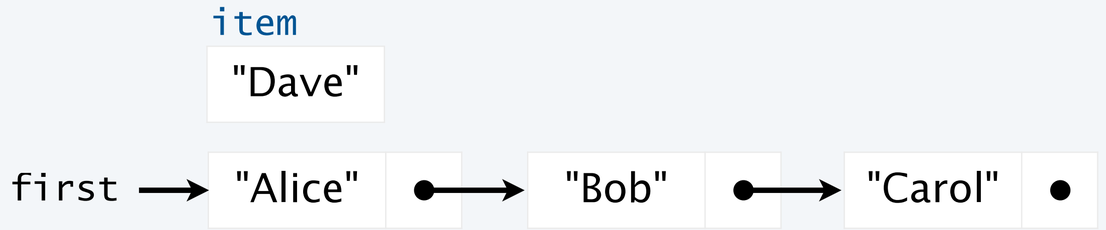



```
return item;
```

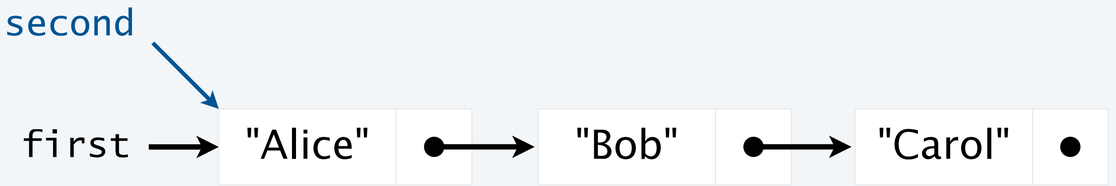


List processing code: Add a new node at the beginning

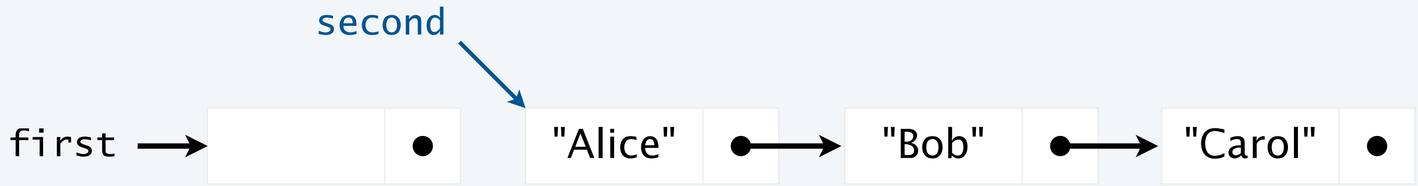
Goal. Add `item` to a linked list `first`.



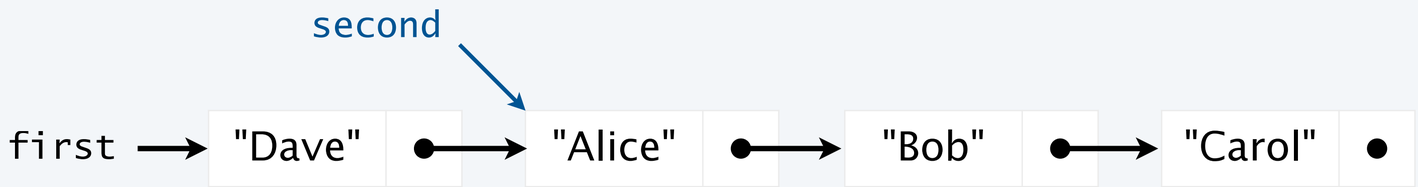

```
Node second = first;
```




```
first = new Node();
```

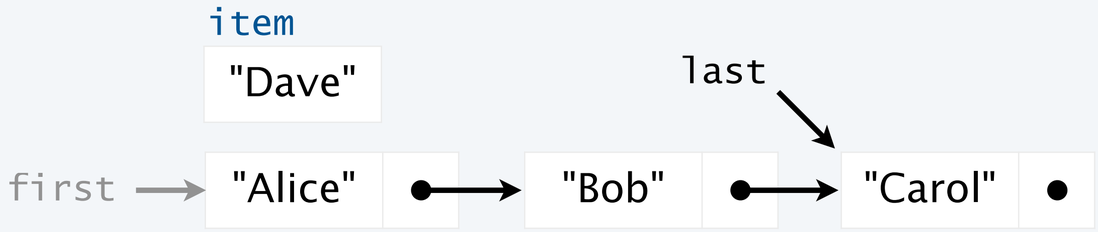



```
first.item = item;
first.next = second;
```

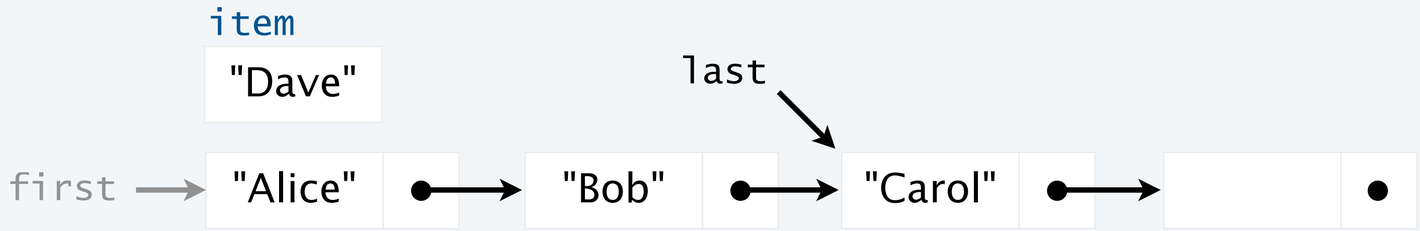


List processing code: Add a new node at the end

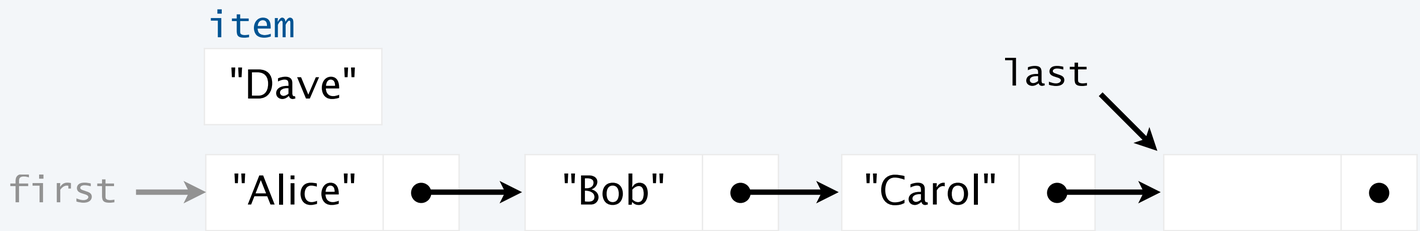
Goal. Add `item` to the end of a linked list. Use and maintain a reference `last` to the last node.



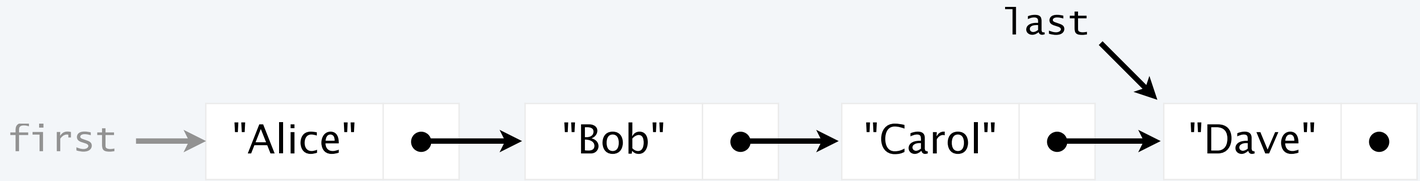

```
last.next = new Node();
```




```
last = last.next;
```



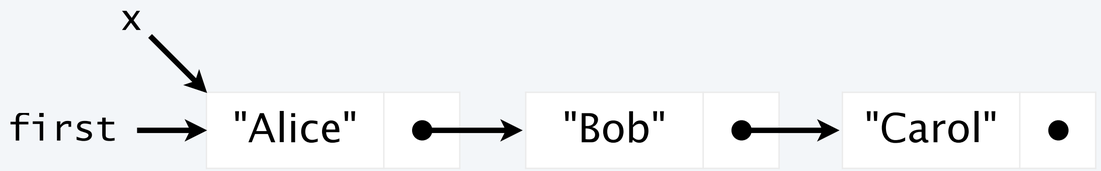

```
last.item = item;
```



List processing code: Traverse a list

Goal. Visit every node on a linked list `first`.

```
Node x = first;
while (x != null)
{
   StdOut.println(x.item);
   x = x.next;
}
```



StdOut

```
Alice
Bob
Carol
```

Pop quiz 1 on linked lists

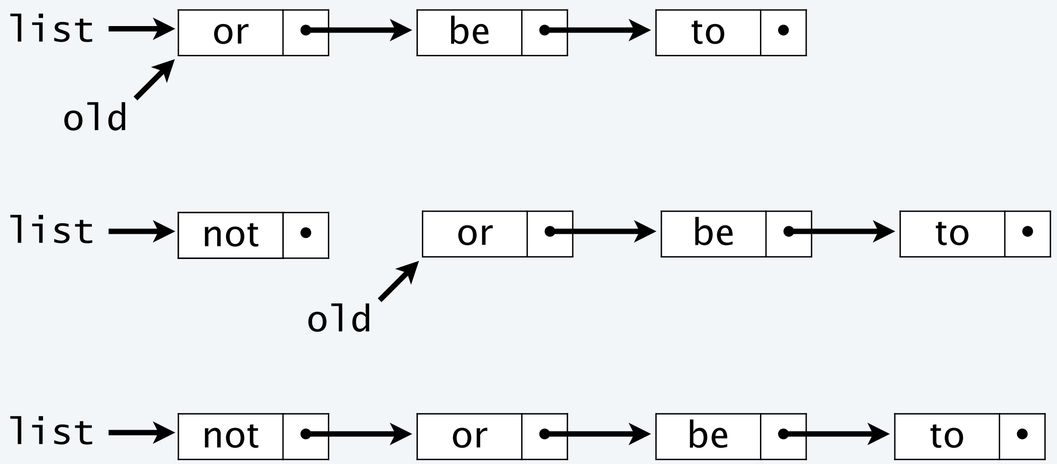
Q. What is the effect of the following code (not-so-easy question)?

```
...
Node list = null;
while (!StdIn.isEmpty())
{
   Node old = list;
   list = new Node();
   list.item = StdIn.readString();
   list.next = old;
}
for (Node t = list; t != null; t = t.next)
   StdOut.println(t.item);
...
```

Pop quiz 1 on linked lists

Q. What is the effect of the following code (not-so-easy question)?

```
...
Node list = null;
while (!StdIn.isEmpty())
{
   Node old = list;
   list = new Node();
   list.item = StdIn.readString();
   list.next = old;
}
for (Node t = list; t != null; t = t.next)
   StdOut.println(t.item);
...
```



A: Prints the strings from StdIn on StdOut, in reverse order

Note: Better to use a *stack* (next lecture!)

Pop quiz 2 on linked lists

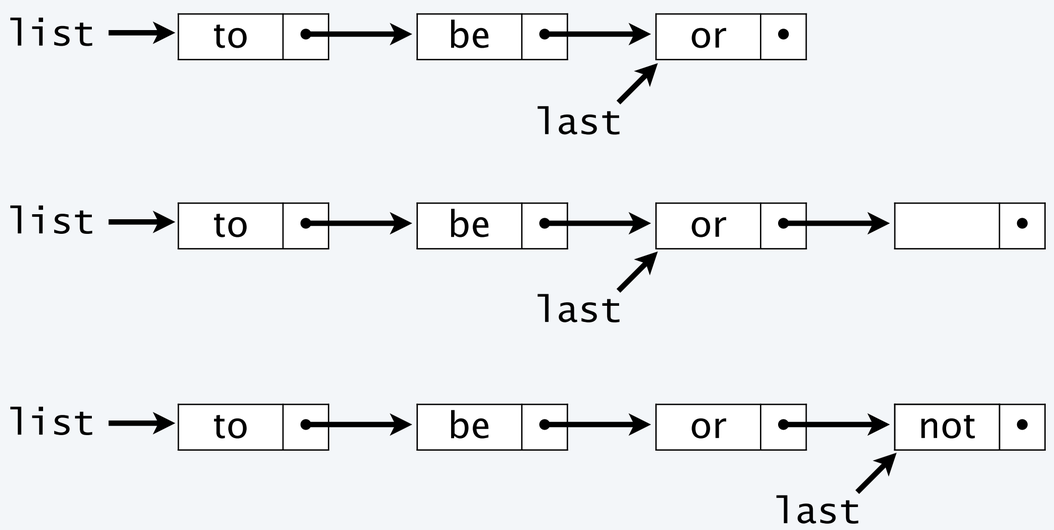
Q. What is the effect of the following code (not-so-easy question)?

```
...
Node list = new Node();
list.item = StdIn.readString();
Node last = list;
while (!StdIn.isEmpty())
{
   last.next = new Node();
   last = last.next;
   last.item = StdIn.readString();
}
...
```

Pop quiz 2 on linked lists

Q. What is the effect of the following code (not-so-easy question)?

```
...
Node list = new Node();
list.item = StdIn.readString();
Node last = list;
while (!StdIn.isEmpty())
{
   last.next = new Node();
   last = last.next;
   last.item = StdIn.readString();
}
...
```



A: Puts the strings from StdIn on a linked list, in the order they are read (assuming at least one string).

Note: Better to use a *queue*, in most applications (next lecture!)

In this course, we restrict use of linked lists to data-type implementations

```
public class IntLinkedList {

    private class Node {
        int val;
        Node next;

        public Node(int v) {
            val = v;
            next = null;
        }
    }

    private Node head; // the first node and access point of the linked list
    private int length; // number of nodes in the list

    // constructor initializes an empty linked list
    public IntLinkedList() {
        head = null;
    }

    // TODO
    // public int length() { }
    // public int get(int i) { }
    // public void addFirst(int val) { }
    // public void addLast(int val) { }
}
```

Linked List Activities

- Implement length()
- Implement addFirst()
- Implement get()
- Implement addLast()

Invariants

- Properties that need to be maintained (e.g., of instance variables)
- For each method:
  - Pre-conditions: what are assumed to be true at the beginning of the method
    - e.g. instance variables, parameters, etc.
  - Post-conditions: what should be true at the end of the method
    - e.g. instance variables, **output**, parameters, etc.

IntLinkedList Invariants

- Instance variables:
  - **head**: should always refer to the first node of the linked list, or null if the list is empty
  - **length**: should always be the number of nodes in the list

Improve addLast()

- addLast() will be slow, if we must traverse the entire LinkedList each time
  - Instead, we can simply keep track of the tail, in addition to the head
- Considering the *invariants* of our class will help us write bug-free code

IntLinkedList Invariants

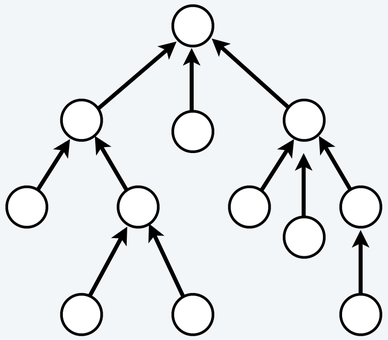
- Instance variables:
  - **head**: should always refer to the first node of the linked list, or null if the list is empty
  - **tail**: should always refer to the last node of the linked list, or null if the list is empty
  - **length**: should always be the number of nodes in the list

Singly-linked data structures

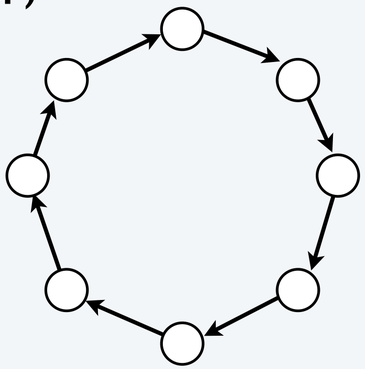
Even with just one link (○⟶) a wide variety of data structures are possible.

Linked list (this lecture)

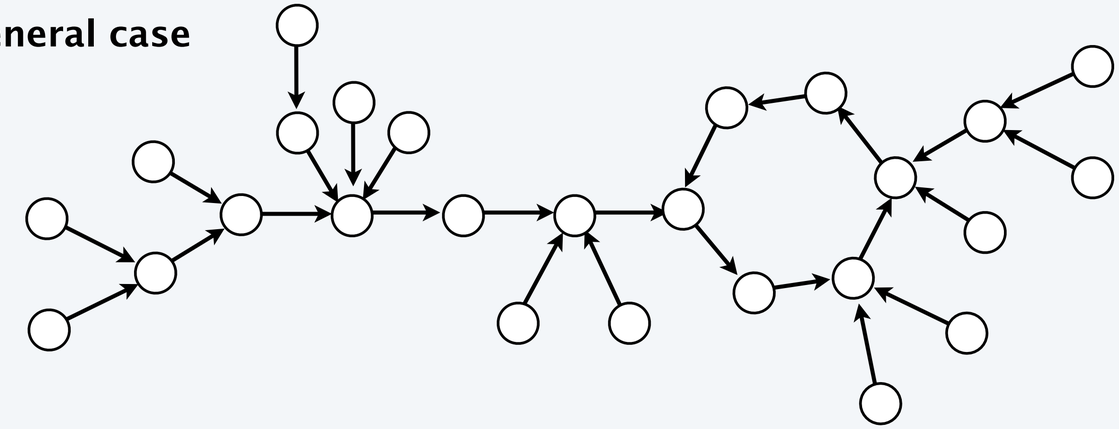
Tree



Circular list (TSP)



General case



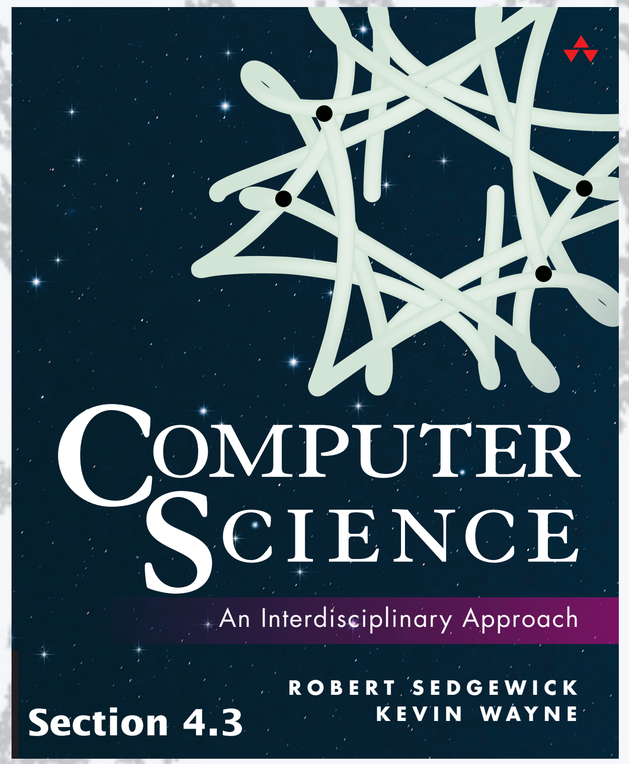
From the point of view of a particular object, all of these structures look the same.

Multiply linked structures: many more possibilities!

COMPUTER SCIENCE

SEDGEWICK / WAYNE

PART II: ALGORITHMS, THEORY, AND MACHINES



http://introcs.cs.princeton.edu

12. Stacks and Queues

Homework Check-in