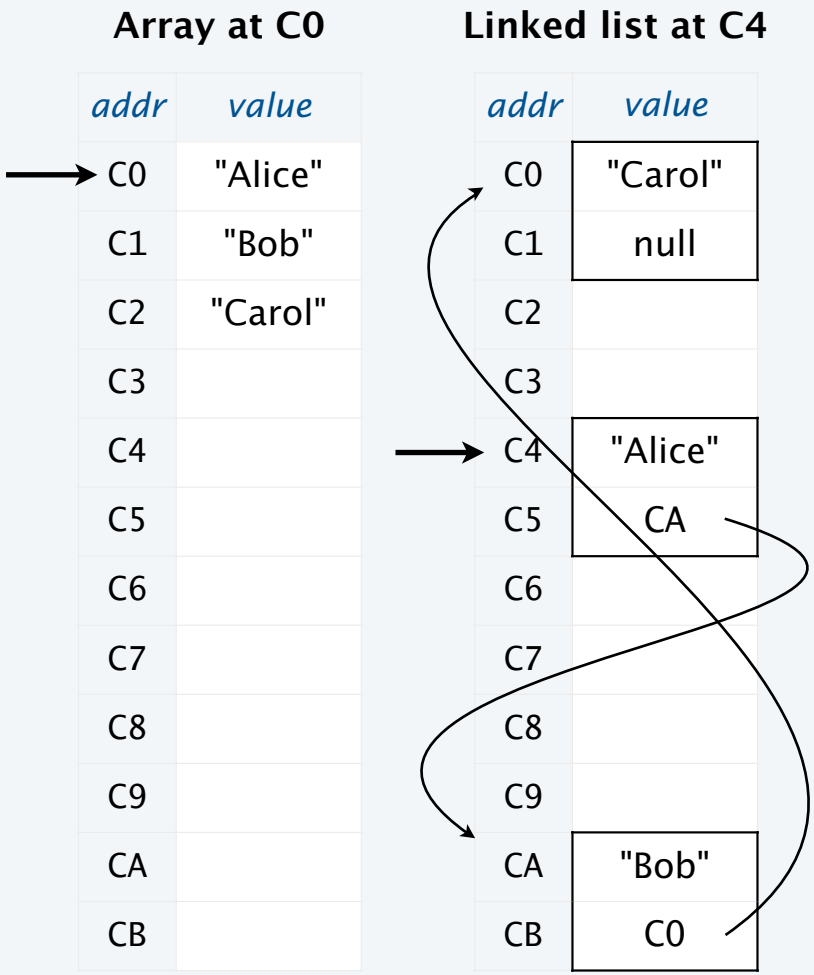
Data structures: sequential vs. linked

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

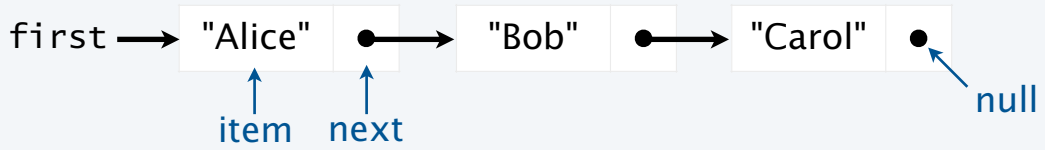
- A recursive data structure.
- Def. A *linked list* is null or a reference to a *node*.
- Def. A *node* is a data type that contains a reference to a node.
- Unwind recursion: A linked list is a sequence of nodes.

```
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

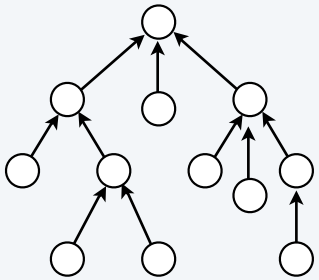


Singly-linked data structures

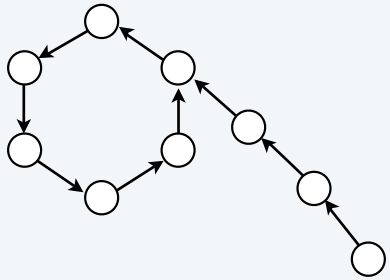
Even with just one link (○⟶) a wide variety of data structures are possible.

Linked list (this lecture)

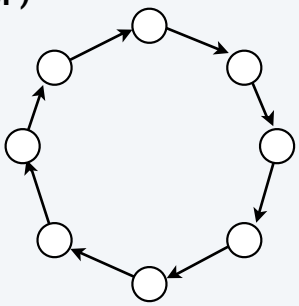
Tree



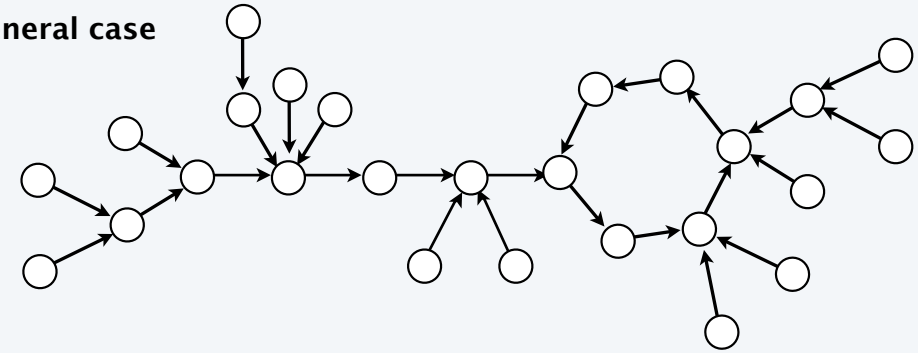
Rho



Circular list (TSP)



General case



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

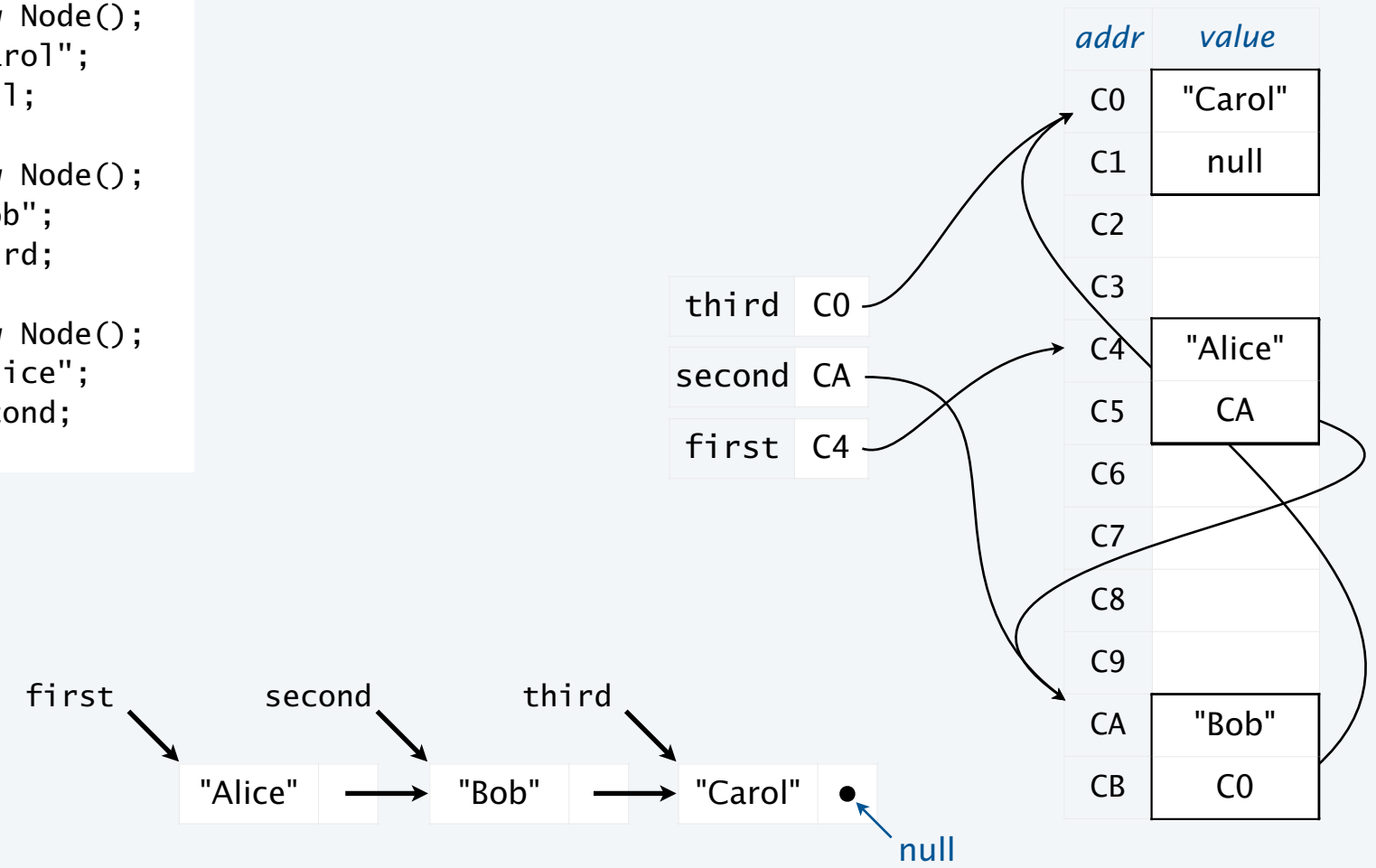
Multiply linked structures: many more possibilities!

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)

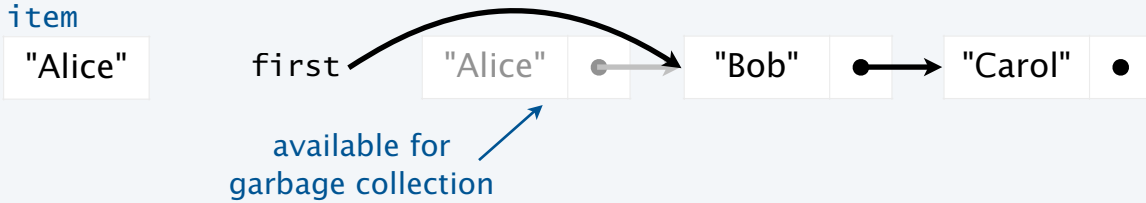
- Remove and return the node at the end.

List processing code: Remove and return the first item

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


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

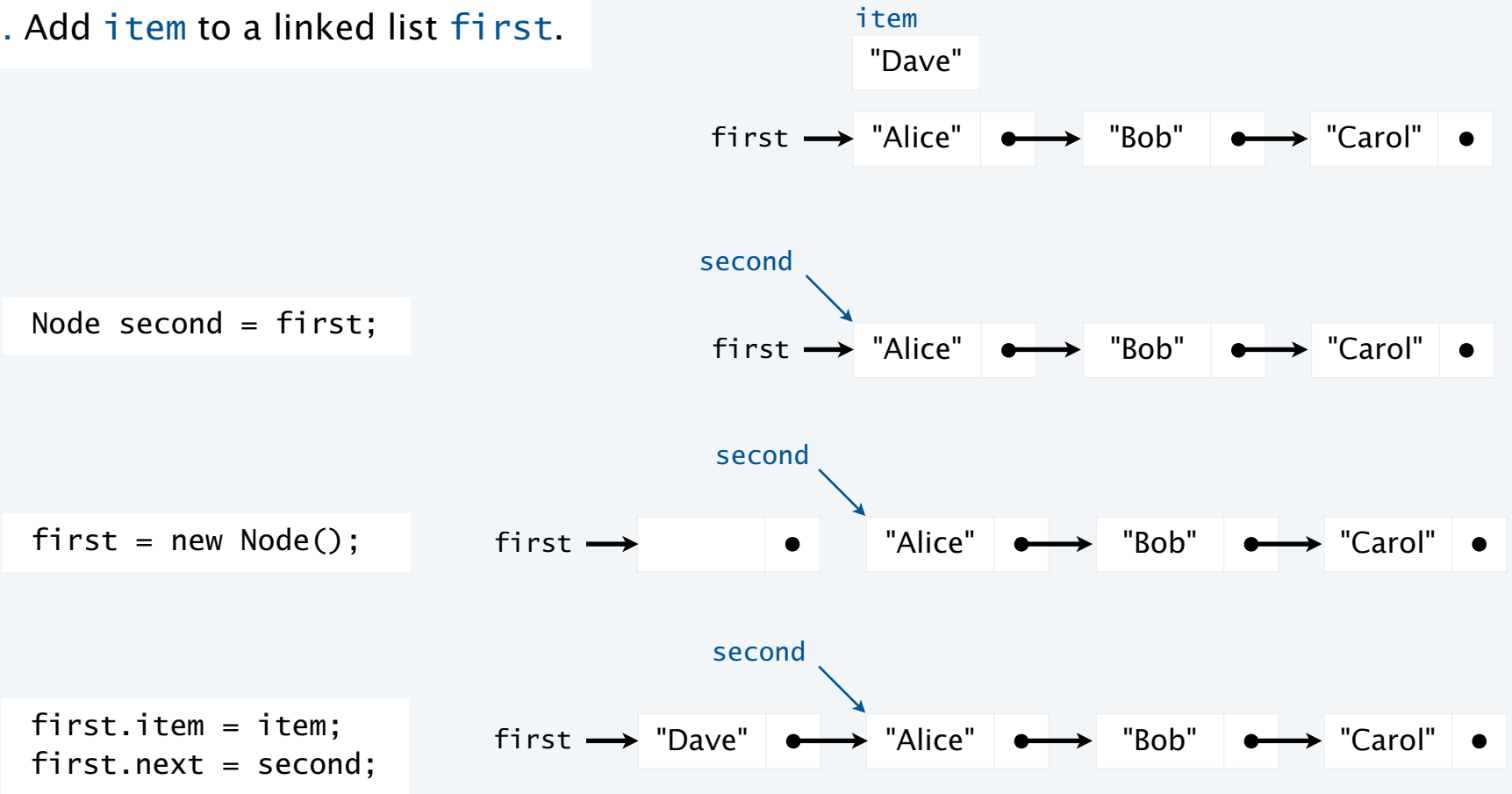

```
first = first.next;
```




```
return item;
```

List processing code: Add a new node at the beginning

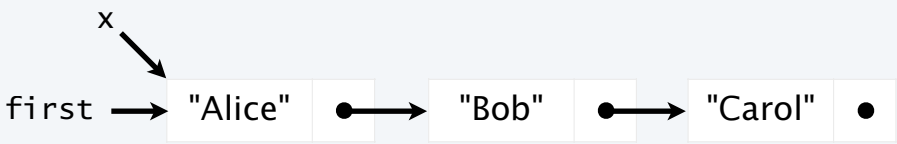
Goal. Add `item` to a linked list `first`.



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 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();
}
...
```