

CS61B Lecture #3: Containers

Announcements

- **Last day to log in.** Be sure to do the first lab. Get an account form from me, if needed.
- **Trying to get in?** You must be concurrently enrolled or wait-listed to get in. Be sure you are one of those two!
- **Reminder about readers.** Three are available at Vick Copy. Be sure you have them.
- **Today.** Simple classes. Scheme-like lists. Destructive vs. non-destructive operations. Models of memory.
- **Today's Reading:** *PIJ* 1.1-1.10, Chapter 4.
- **Next Friday's Reading:** *PIJ* 1.11-1.12.

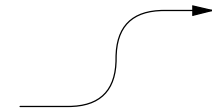
Values and Containers

- *Values* are numbers, booleans, and pointers. Values never change.

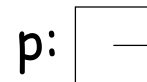
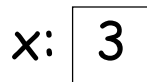
3

'a'

true



- *Simple containers* contain values:



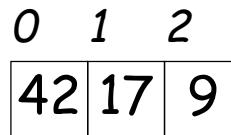
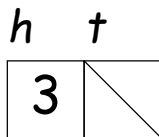
Examples: variables, fields, individual array elements, parameters.

- *Structured containers* contain (0 or more) other containers:

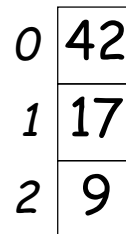
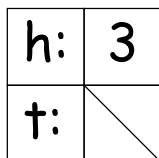
Class Object

Array Object

Empty Object

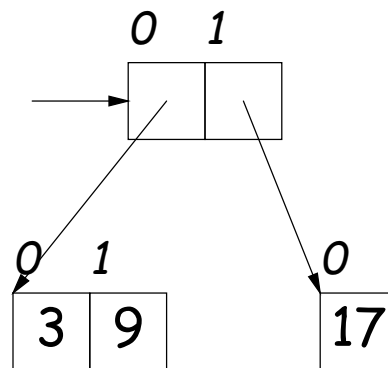


Alternative Notation



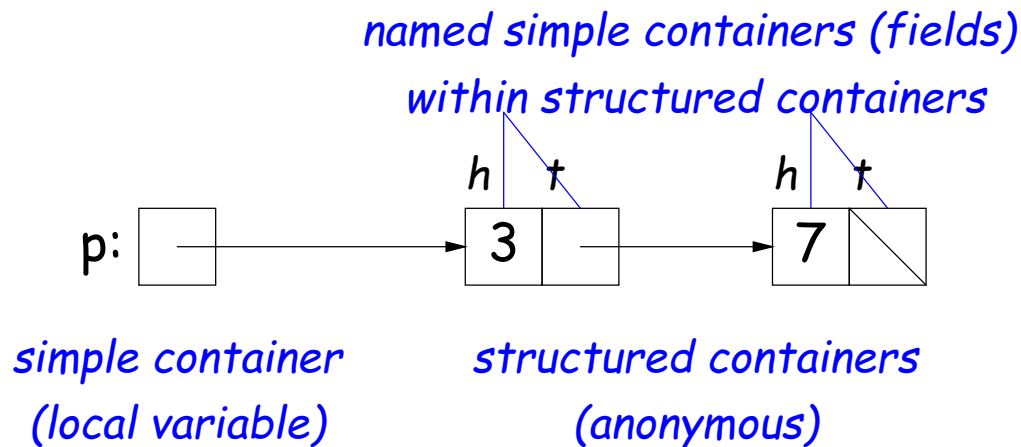
Pointers

- *Pointers* (or *references*) are values that *reference* (point to) containers.
- One particular pointer, called **null**, points to nothing.
- In Java, structured containers contain only simple containers, but pointers allow us to build arbitrarily big or complex structures anyway.



Containers in Java

- Containers may be *named* or *anonymous*.
- In Java, *all* simple containers are named, *all* structured containers are anonymous, and pointers point only to structured containers. (Therefore, structured containers contain only simple containers).



- In Java, assignment copies values into simple containers.
- *Exactly* like Scheme!

Defining New Types of Object

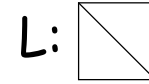
- Class declarations introduce new types of objects.
- Example: list of integers:

```
public class IntList {
    // Constructor function
    // (used to initialize new object)
    /** List cell containing (HEAD, TAIL). */
    public IntList (int head, IntList tail) {
        this.head = head; this.tail = tail;
    }

    // Names of simple containers (fields)
    public int head;
    public IntList tail;
}
```

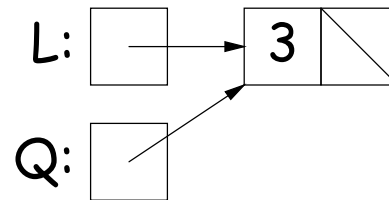
Primitive Operations

```
IntList Q, L;
```



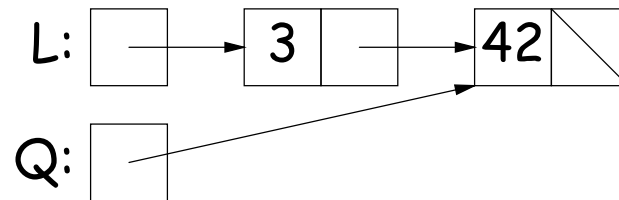
```
L = new IntList(3, null);
```

```
Q = L;
```



```
Q = new IntList(42, null);
```

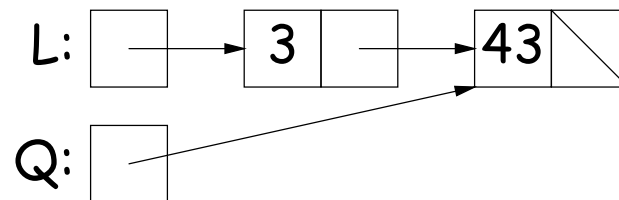
```
L.tail = Q;
```



```
L.tail.head += 1;
```

```
// Now Q.head == 43
```

```
// and L.tail.head == 43
```



Destructive vs. Non-destructive

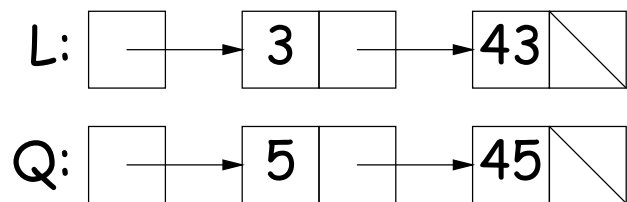
Problem: Given a (pointer to a) list of integers, L , and an integer increment n , return a list created by incrementing all elements of the list by n .

```
/** List of all items in P incremented by n. */
static IntList incrList (IntList P, int n) {
    if (P == null)
        return null;
    else return new IntList (P.head+n, incrList(P.tail, n));
}
```

We say this method is *non-destructive*: that is, computing

```
Q = incrList(L, 2);
```

keeps the original list intact, giving



An Iterative Version

- `incrList` is trickier than previously, because it is *not* tail recursive (is a **new** after the recursive call).
- Easier to build things in reverse order from the recursive version.

```
static IntList incrList (IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last = new IntList (P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail = new IntList (P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

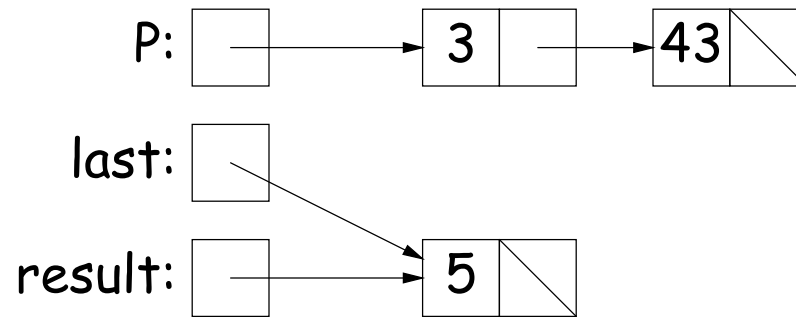
Steps in the Iterative Copy

Tracing:

```
Q = incrList(L, 2);
```

After

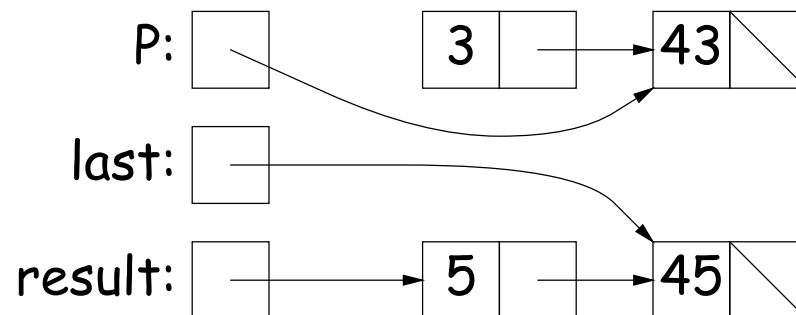
```
result = last = new IntList (P.head+n, null);
```



```
P = P.tail;
```

```
last.tail = new IntList (P.head+n, null);
```

```
last = last.tail;
```



Destructive Operation

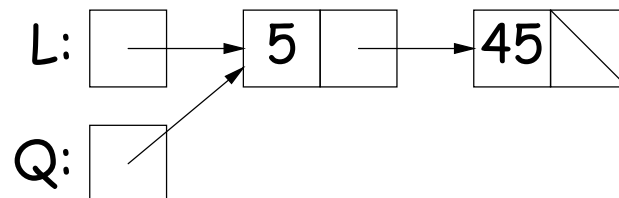
Destructive solutions may modify the original list to save time or space:

```
/** List of all items in P incremented by n.  
 * May destroy original. */  
static IntList dincrList (IntList P, int n) {  
    if (P == null)  
        return null;  
    else {  
        P.head += n;  
        dincrList (P.tail, n);  
        return P;  
    }  
}
```

Initially:



After $Q = \text{dincrList}(L, 2)$:



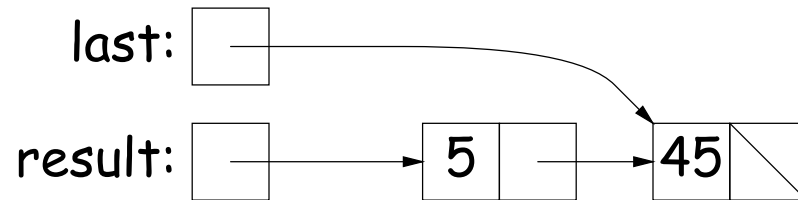
Iterative Destructive Increment

```
/** List L destructively incremented by n. */  
static IntList dincrList (IntList L, int n) {  
    for (IntList p = L; p != null; p = p.tail)  
        p.head += n;  
    return L;  
}
```

- As you can see, **for** doesn't just add.

Another Way to View Pointers

- Some folks find the idea of "copying an arrow" somewhat odd.
- Alternative view: think of a pointer as a *label*, like a street address.
- Each object has a permanent label on it, like the address plaque on a house.
- Then a variable containing a pointer is like a scrap of paper with a street address written on it.
- One view:



- Alternative view:

