

CS61B Lecture #4: Simple Pointer Manipulation

- *Always* turn homework in and do labs, even if you don't completely get it!
- **Today:** Still working on Chapters 1-3 of the Blue Reader.
- Details on Java-language material to date may be found in the Java Language Specification reader, Chapter 2 (notation), Chapter 3 (just read lightly), and sections 14.2, 14.4-14.9, 14-12-14.18, 15.1-15.3, 15.7, 15.11.1, 15.13-15.15, 15.17, 15.18, 15.20, 15.21, 15.23-15.27
- OASES is a non-profit organization that tutors underprivileged children grades 1-12. It offers up to 2 units for tutors. The OASES Infosession is Thursday, Sept. 9th at 7:30pm in 2040 VLSB.

Destructive Incrementing

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;
```

```
        P.tail = dincrList (P.tail, n);
```

```
        return P;
```

```
    }
```

```
}
```

```
/** List L destructively incremented
```

```
 * by n. */
```

```
static IntList dincrList (IntList L, int n) {
```

```
    // 'for' can do more than count!
```

```
    for (IntList p = L; p != null; p = p.tail)
```

```
        p.head += n;
```

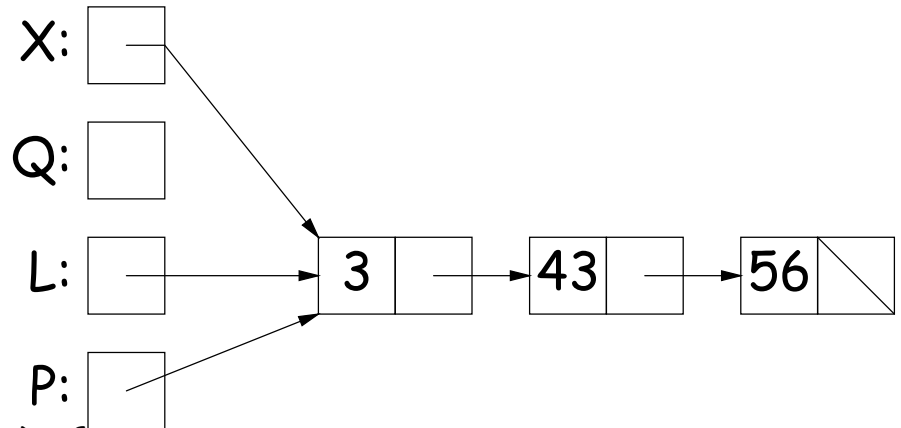
```
    return L;
```

```
}
```

```
X = IntList.list (3, 43, 56);
```

```
/* IntList.list from HW #1 */
```

```
Q = dincrList (X, 2);
```



Destructive Incrementing

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

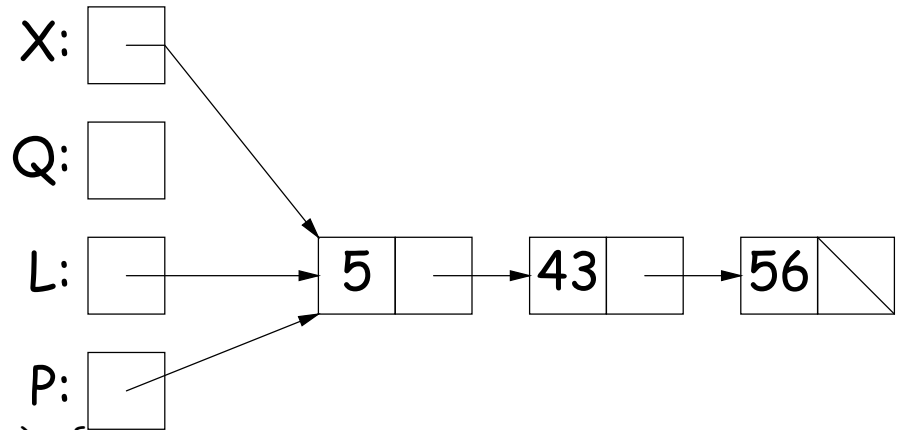
```
/** List of all items in P incremented by n. May destroy original. */
```

```
static IntList dincrList (IntList P, int n) {  
    if (P == null)  
        return null;  
    else {  
        P.head += n;  
        P.tail = dincrList (P.tail, n);  
        return P;  
    }  
}
```

```
/** List L destructively incremented  
 * by n. */
```

```
static IntList dincrList (IntList L, int n) {  
    // 'for' can do more than count!  
    for (IntList p = L; p != null; p = p.tail)  
        p.head += n;  
    return L;  
}
```

```
X = IntList.list (3, 43, 56);  
/* IntList.list from HW #1 */  
Q = dincrList (X, 2);
```



Destructive Incrementing

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

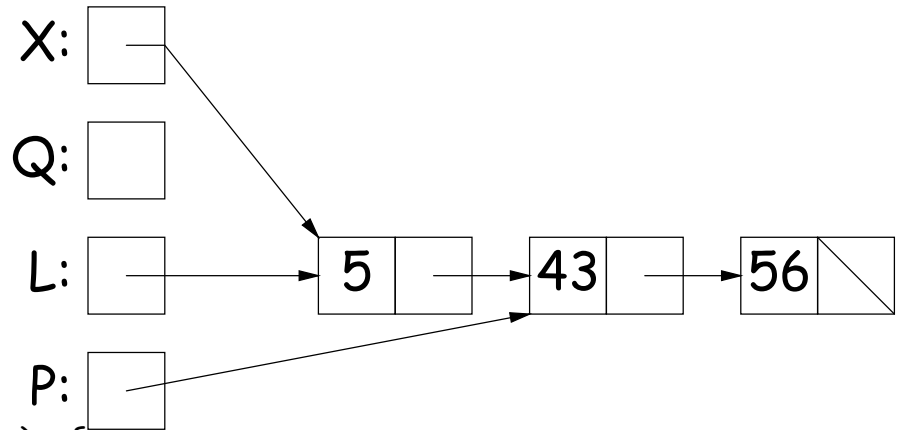
```
/** List of all items in P incremented by n. May destroy original. */
```

```
static IntList dincrList (IntList P, int n) {  
    if (P == null)  
        return null;  
    else {  
        P.head += n;  
        P.tail = dincrList (P.tail, n);  
        return P;  
    }  
}
```

```
/** List L destructively incremented  
 * by n. */
```

```
static IntList dincrList (IntList L, int n) {  
    // 'for' can do more than count!  
    for (IntList p = L; p != null; p = p.tail)  
        p.head += n;  
    return L;  
}
```

```
X = IntList.list (3, 43, 56);  
/* IntList.list from HW #1 */  
Q = dincrList (X, 2);
```



Destructive Incrementing

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

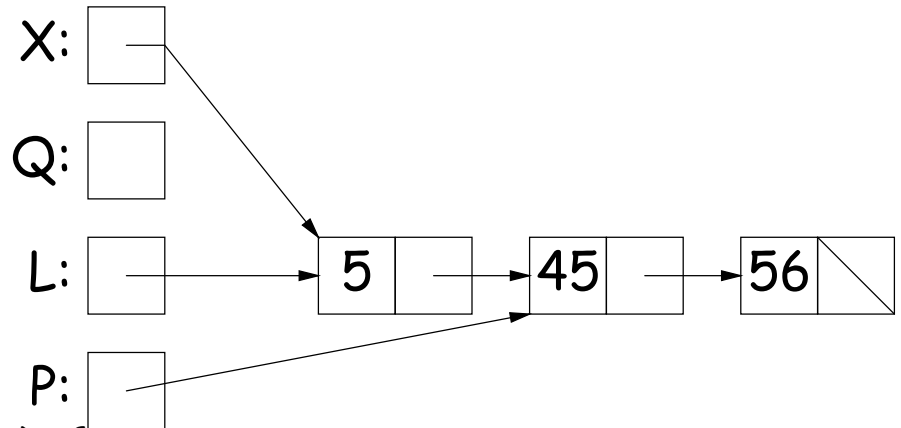
```
/** List of all items in P incremented by n. May destroy original. */
```

```
static IntList dincrList (IntList P, int n) {  
    if (P == null)  
        return null;  
    else {  
        P.head += n;  
        P.tail = dincrList (P.tail, n);  
        return P;  
    }  
}
```

```
/** List L destructively incremented  
 * by n. */
```

```
static IntList dincrList (IntList L, int n) {  
    // 'for' can do more than count!  
    for (IntList p = L; p != null; p = p.tail)  
        p.head += n;  
    return L;  
}
```

```
X = IntList.list (3, 43, 56);  
/* IntList.list from HW #1 */  
Q = dincrList (X, 2);
```



Destructive Incrementing

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

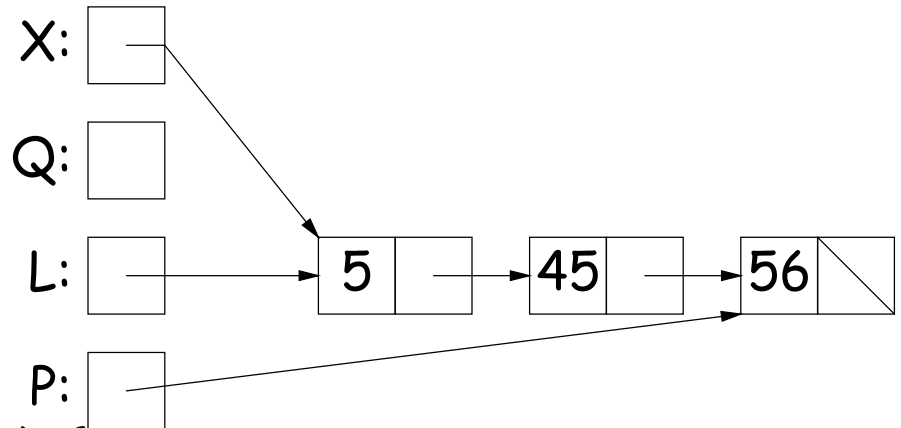
```
/** List of all items in P incremented by n. May destroy original. */
```

```
static IntList dincrList (IntList P, int n) {  
    if (P == null)  
        return null;  
    else {  
        P.head += n;  
        P.tail = dincrList (P.tail, n);  
        return P;  
    }  
}
```

```
/** List L destructively incremented  
 * by n. */
```

```
static IntList dincrList (IntList L, int n) {  
    // 'for' can do more than count!  
    for (IntList p = L; p != null; p = p.tail)  
        p.head += n;  
    return L;  
}
```

```
X = IntList.list (3, 43, 56);  
/* IntList.list from HW #1 */  
Q = dincrList (X, 2);
```



Destructive Incrementing

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

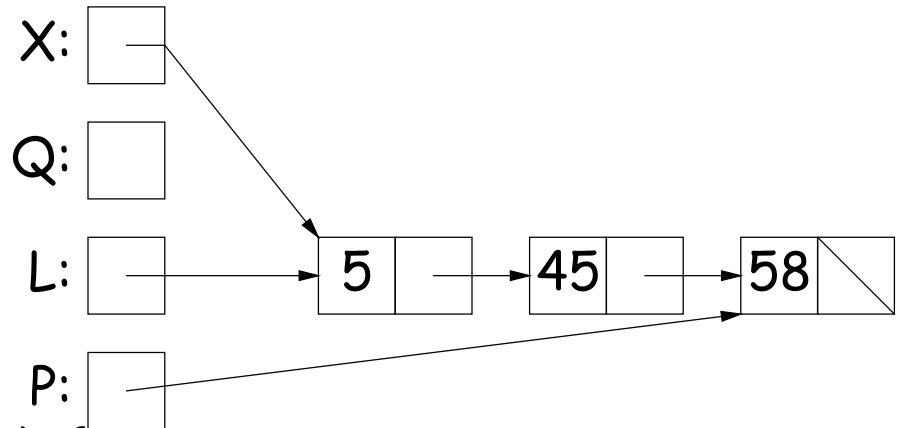
```
/** List of all items in P incremented by n. May destroy original. */
```

```
static IntList dincrList (IntList P, int n) {  
    if (P == null)  
        return null;  
    else {  
        P.head += n;  
        P.tail = dincrList (P.tail, n);  
        return P;  
    }  
}
```

```
/** List L destructively incremented  
 * by n. */
```

```
static IntList dincrList (IntList L, int n) {  
    // 'for' can do more than count!  
    for (IntList p = L; p != null; p = p.tail)  
        p.head += n;  
    return L;  
}
```

```
X = IntList.list (3, 43, 56);  
/* IntList.list from HW #1 */  
Q = dincrList (X, 2);
```



Destructive Incrementing

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

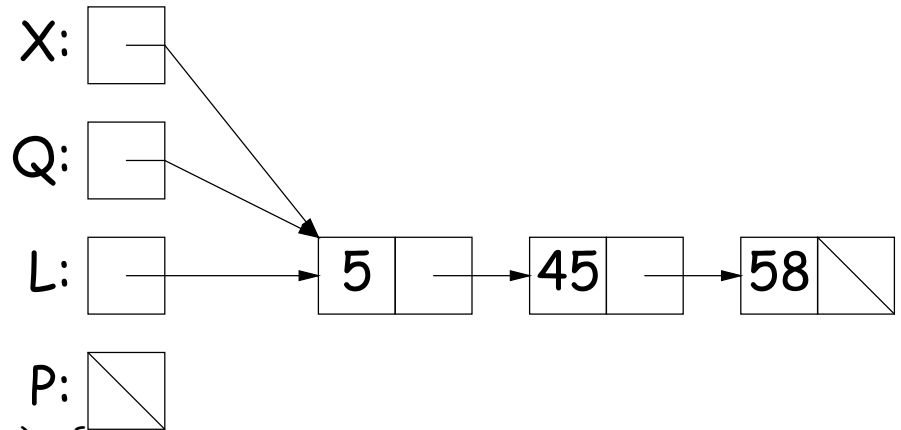
```
/** List of all items in P incremented by n. May destroy original. */
```

```
static IntList dincrList (IntList P, int n) {  
    if (P == null)  
        return null;  
    else {  
        P.head += n;  
        P.tail = dincrList (P.tail, n);  
        return P;  
    }  
}
```

```
/** List L destructively incremented  
 * by n. */
```

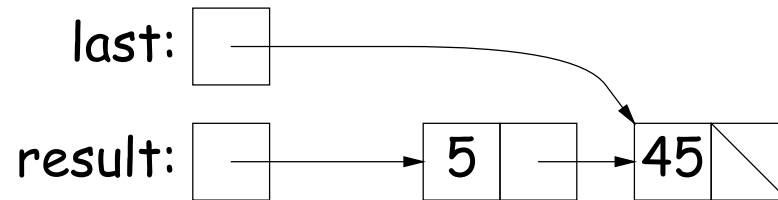
```
static IntList dincrList (IntList L, int n) {  
    // 'for' can do more than count!  
    for (IntList p = L; p != null; p = p.tail)  
        p.head += n;  
    return L;  
}
```

```
X = IntList.list (3, 43, 56);  
/* IntList.list from HW #1 */  
Q = dincrList (X, 2);
```

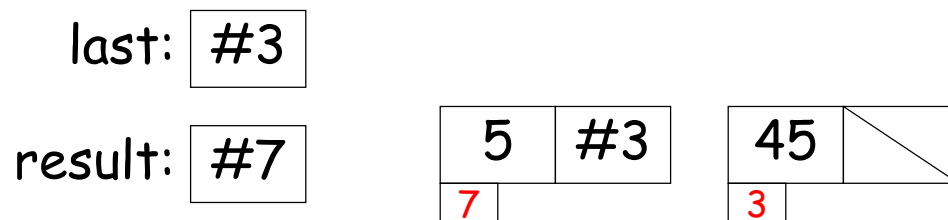


Another Way to View Pointers

- 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:



Another Example: Non-destructive List Deletion

If L is the list $[2, 1, 2, 9, 2]$, we want `removeAll(L,2)` to be the new list $[1, 9]$.

```
/** The list resulting from removing all instances of X from L
 *  non-destructively. */
static IntList removeAll (IntList L, int x) {
    if (L == null)
        return /*( null with all x's removed )*/;
    else if (L.head == x)
        return /*( L with all x's removed (L != null) )*/;
    else
        return /*( L with all x's removed (L != null, L.head!=x) )*/;
}
```

Another Example: Non-destructive List Deletion

If L is the list $[2, 1, 2, 9, 2]$, we want `removeAll(L,2)` to be the new list $[1, 9]$.

```
/** The list resulting from removing all instances of X from L
 *  non-destructively. */
static IntList removeAll (IntList L, int x) {
    if (L == null)
        return null;
    else if (L.head == x)
        return /*( L with all x's removed (L != null) )*/;
    else
        return /*( L with all x's removed (L != null, L.head!=x) )*/;
}
```

Another Example: Non-destructive List Deletion

If *L* is the list [2, 1, 2, 9, 2], we want `removeAll(L,2)` to be the new list [1, 9].

```
/** The list resulting from removing all instances of X from L
 *  non-destructively. */
static IntList removeAll (IntList L, int x) {
    if (L == null)
        return null;
    else if (L.head == x)
        return removeAll (L.tail, x);
    else
        return /*( L with all x's removed (L != null, L.head!=x) )*/;
}
```

Another Example: Non-destructive List Deletion

If L is the list $[2, 1, 2, 9, 2]$, we want `removeAll(L, 2)` to be the new list $[1, 9]$.

```
/** The list resulting from removing all instances of X from L
 *  non-destructively. */
static IntList removeAll (IntList L, int x) {
    if (L == null)
        return null;
    else if (L.head == x)
        return removeAll (L.tail, x);
    else
        return new IntList (L.head, removeAll (L.tail, x));
}
```

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances of X from L
 * non-destructively. */
```

```
static IntList removeAll (IntList L, int x) {
    IntList result, last;
    result = last = null;
    for ( ; L != null; L = L.tail) {
        /* L != null and  $\mathcal{I}$  is true. */
        if (x == L.head)
            continue;
        else if (last == null)
            result = last = new IntList (L.head, null);
        else
            last = last.tail = new IntList (L.head, null);
    }
    return result;
}
```

Here, \mathcal{I} is the *loop invariant*:

Result is all elements of L_0 not equal to x up to and not including L , and $last$ points to the last element of $result$, if any. We use L_0 here to mean "the original value of L ."

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances of X from L
 * non-destructively. */
```

```
static IntList removeAll (IntList L, int x) {
```

```
    IntList result, last;
```

```
    result = last = null;
```

```
    for ( ; L != null; L = L.tail) {
```

```
        /* L != null and  $\mathcal{I}$  is true. */
```

```
        if (x == L.head)
```

```
            continue;
```

```
        else if (last == null)
```

```
            result = last = new IntList (L.head, null);
```

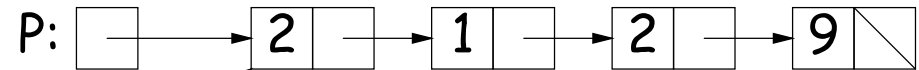
```
        else
```

```
            last = last.tail = new IntList (L.head, null);
```

```
    }
```

```
    return result;
```

```
}
```



removeAll (P, 2)

Here, \mathcal{I} is the *loop invariant*:

Result is all elements of L_0 not equal to x up to and not including L , and $last$ points to the last element of $result$, if any. We use L_0 here to mean "the original value of L ."

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances of X from L
 * non-destructively. */
```

```
static IntList removeAll (IntList L, int x) {
```

```
    IntList result, last;
```

```
    result = last = null;
```

```
    for ( ; L != null; L = L.tail) {
```

```
        /* L != null and  $\mathcal{I}$  is true. */
```

```
        if (x == L.head)
```

```
            continue;
```

```
        else if (last == null)
```

```
            result = last = new IntList (L.head, null);
```

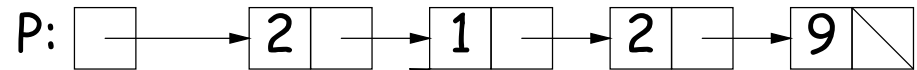
```
        else
```

```
            last = last.tail = new IntList (L.head, null);
```

```
    }
```

```
    return result;
```

```
}
```



removeAll (P, 2)
P does not change!

Here, \mathcal{I} is the *loop invariant*:

Result is all elements of L_0 not equal to x up to and not including L , and $last$ points to the last element of $result$, if any. We use L_0 here to mean "the original value of L ."

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances of X from L
 * non-destructively. */
```

```
static IntList removeAll (IntList L, int x) {
```

```
    IntList result, last;
```

```
    result = last = null;
```

```
    for ( ; L != null; L = L.tail) {
```

```
        /* L != null and  $\mathcal{I}$  is true. */
```

```
        if (x == L.head)
```

```
            continue;
```

```
        else if (last == null)
```

```
            result = last = new IntList (L.head, null);
```

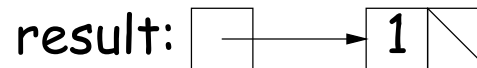
```
        else
```

```
            last = last.tail = new IntList (L.head, null);
```

```
    }
```

```
    return result;
```

```
}
```



removeAll (P, 2)
P does not change!

Here, \mathcal{I} is the *loop invariant*:

Result is all elements of L_0 not equal to x up to and not including L , and $last$ points to the last element of $result$, if any. We use L_0 here to mean "the original value of L ."

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances of X from L
 * non-destructively. */
```

```
static IntList removeAll (IntList L, int x) {
```

```
    IntList result, last;
```

```
    result = last = null;
```

```
    for ( ; L != null; L = L.tail) {
```

```
        /* L != null and  $\mathcal{I}$  is true. */
```

```
        if (x == L.head)
```

```
            continue;
```

```
        else if (last == null)
```

```
            result = last = new IntList (L.head, null);
```

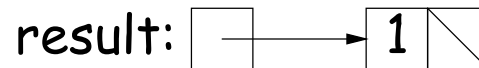
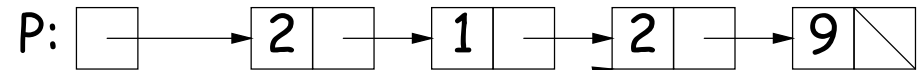
```
        else
```

```
            last = last.tail = new IntList (L.head, null);
```

```
    }
```

```
    return result;
```

```
}
```



removeAll (P, 2)
P does not change!

Here, \mathcal{I} is the *loop invariant*:

Result is all elements of L_0 not equal to x up to and not including L , and $last$ points to the last element of $result$, if any. We use L_0 here to mean "the original value of L ."

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances of X from L
 * non-destructively. */
```

```
static IntList removeAll (IntList L, int x) {
```

```
    IntList result, last;
```

```
    result = last = null;
```

```
    for ( ; L != null; L = L.tail) {
```

```
        /* L != null and  $\mathcal{I}$  is true. */
```

```
        if (x == L.head)
```

```
            continue;
```

```
        else if (last == null)
```

```
            result = last = new IntList (L.head, null);
```

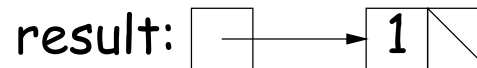
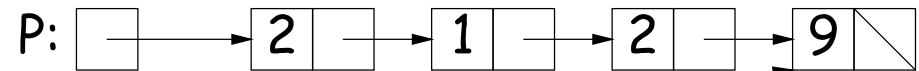
```
        else
```

```
            last = last.tail = new IntList (L.head, null);
```

```
    }
```

```
    return result;
```

```
}
```



removeAll (P, 2)
P does not change!

Here, \mathcal{I} is the loop invariant:

Result is all elements of L_0 not equal to x up to and not including L, and last points to the last element of result, if any. We use L_0 here to mean "the original value of L."

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances of X from L
 * non-destructively. */
```

```
static IntList removeAll (IntList L, int x) {
```

```
    IntList result, last;
```

```
    result = last = null;
```

```
    for ( ; L != null; L = L.tail) {
```

```
        /* L != null and  $\mathcal{I}$  is true. */
```

```
        if (x == L.head)
```

```
            continue;
```

```
        else if (last == null)
```

```
            result = last = new IntList (L.head, null);
```

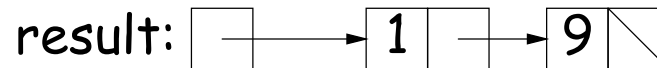
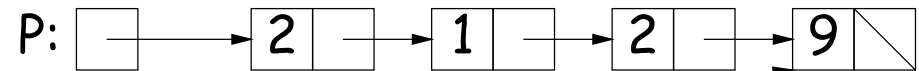
```
        else
```

```
            last = last.tail = new IntList (L.head, null);
```

```
    }
```

```
    return result;
```

```
}
```



removeAll (P, 2)
P does not change!

Here, \mathcal{I} is the *loop invariant*:

Result is all elements of L_0 not equal to x up to and not including L , and $last$ points to the last element of $result$, if any. We use L_0 here to mean "the original value of L ."

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances of X from L
 * non-destructively. */
```

```
static IntList removeAll (IntList L, int x) {
```

```
    IntList result, last;
```

```
    result = last = null;
```

```
    for ( ; L != null; L = L.tail) {
```

```
        /* L != null and  $\mathcal{I}$  is true. */
```

```
        if (x == L.head)
```

```
            continue;
```

```
        else if (last == null)
```

```
            result = last = new IntList (L.head, null);
```

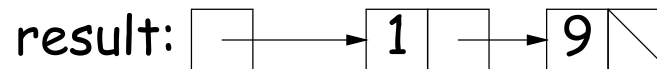
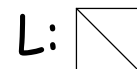
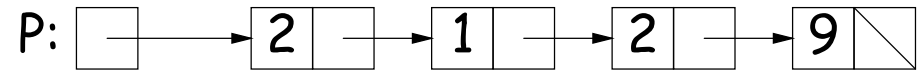
```
        else
```

```
            last = last.tail = new IntList (L.head, null);
```

```
    }
```

```
    return result;
```

```
}
```



removeAll (P, 2)
P does not change!

Here, \mathcal{I} is the loop invariant:

Result is all elements of L_0 not equal to x up to and not including L , and last points to the last element of result, if any. We use L_0 here to mean "the original value of L ."

Aside: How to Write a Loop (in Theory)

- Try to give a description of how things look on *any arbitrary iteration* of the loop.
- This description is known as a *loop invariant*, because it is true from one iteration to the next.
- The loop body then must
 - Start from any situation consistent with the invariant;
 - Make progress in such a way as to make the invariant true again.

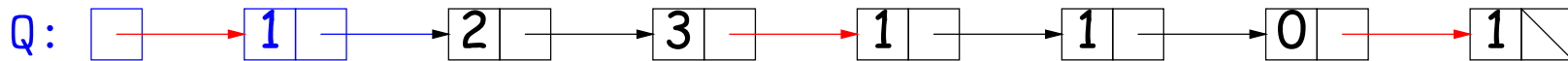
```
while (condition) {  
    // Invariant true here  
    loop body  
    // Invariant again true here  
}  
// Invariant true and condition false.
```

- So if (*invariant* and not *condition*) is enough to insure we've got the answer, we're done!

Destructive Deletion

→ : Original

----- : after Q = dremoveAll (Q,1)



```

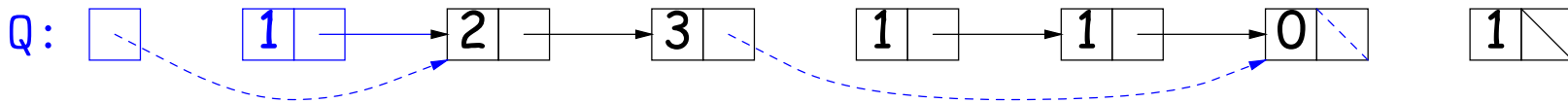
/** The list resulting from removing all instances of X from L.
 * The original list may be destroyed. */
static IntList dremoveAll (IntList L, int x) {
    if (L == null)
        return /*( null with all x's removed )*/;
    else if (L.head == x)
        return /*( L with all x's removed (L != null) )*/;
    else {
        /*{ Remove all x's from L's tail.  }*/;
        return L;
    }
}

```

Destructive Deletion

→ : Original

----- : after Q = dremoveAll (Q,1)



/** The list resulting from removing all instances of X from L.

* The original list may be destroyed. */

```
static IntList dremoveAll (IntList L, int x) {
```

```
    if (L == null)
```

```
        return /*( null with all x's removed )*/;
```

```
    else if (L.head == x)
```

```
        return /*( L with all x's removed (L != null) )*/;
```

```
    else {
```

```
        /*{ Remove all x's from L's tail.  }*/;
```

```
        return L;
```

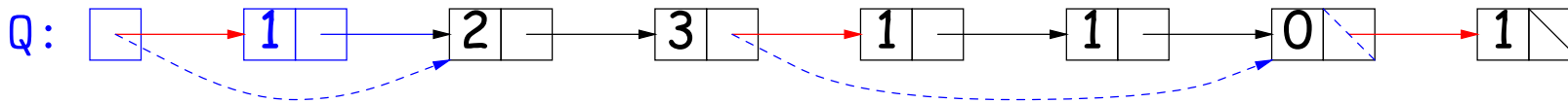
```
    }
```

```
}
```


Destructive Deletion

— : Original

----- : after Q = dremoveAll (Q,1)



```

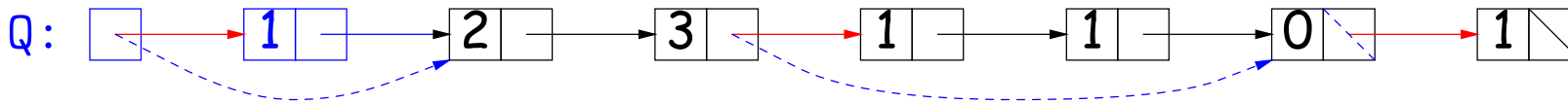
/** The list resulting from removing all instances of X from L.
 * The original list may be destroyed. */
static IntList dremoveAll (IntList L, int x) {
    if (L == null)
        return /*( null with all x's removed )*/;
    else if (L.head == x)
        return /*( L with all x's removed (L != null) )*/;
    else {
        /*{ Remove all x's from L's tail.  }*/;
        return L;
    }
}

```

Destructive Deletion

→ : Original

----- : after Q = dremoveAll (Q,1)



/** The list resulting from removing all instances of X from L.

* The original list may be destroyed. */

static IntList dremoveAll (IntList L, int x) {

if (L == null)

return /*(null with all x's removed)*/;

else if (L.head == x)

return /*(L with all x's removed (L != null))*/;

else {

/*{ Remove all x's from L's tail. }*/;

return L;

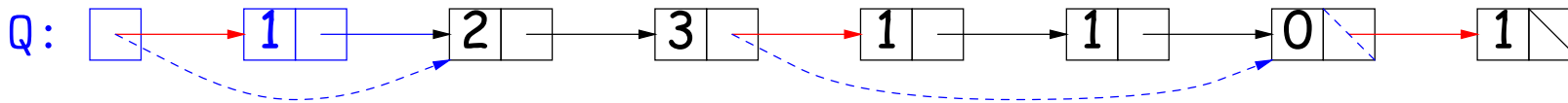
}

}

Destructive Deletion

— : Original

----- : after Q = dremoveAll (Q,1)



```

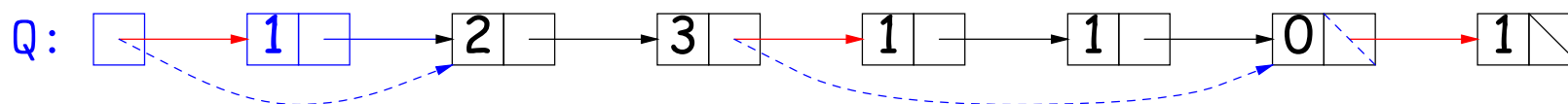
/** The list resulting from removing all instances of X from L.
 * The original list may be destroyed. */
static IntList dremoveAll (IntList L, int x) {
    if (L == null)
        return null;
    else if (L.head == x)
        return /*( L with all x's removed (L != null) )*/;
    else {
        /*{ Remove all x's from L's tail.  }*/;
        return L;
    }
}

```

Destructive Deletion

—→ : Original

----- : after Q = dremoveAll (Q,1)



/** The list resulting from removing all instances of X from L.

* The original list may be destroyed. */

static IntList dremoveAll (IntList L, int x) {

if (L == null)

return null;

else if (L.head == x)

return dremoveAll (L.tail, x);

else {

/*{ Remove all x's from L's tail. }*/;

return L;

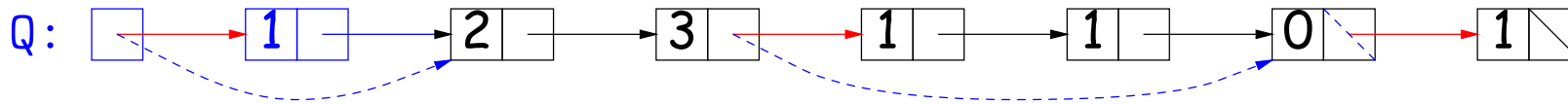
}

}

Destructive Deletion

→ : Original

----- : after Q = dremoveAll (Q,1)



```

/** The list resulting from removing all instances of X from L.
 * The original list may be destroyed. */
static IntList dremoveAll (IntList L, int x) {
    if (L == null)
        return null;
    else if (L.head == x)
        return dremoveAll (L.tail, x);
    else {
        L.tail = dremoveAll (L.tail, x);
        return L;
    }
}

```

Iterative Destructive Deletion

```
/** The list resulting from removing all instances of X from L.
 * Original contents of L may be destroyed. */
static IntList dremoveAll (IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```

Iterative Destructive Deletion

```
/** The list resulting from removing all instances of X from L.
```

```
 * Original contents of L may be destroyed. */
```

```
static IntList dremoveAll (IntList L, int x) {
```

```
    IntList result, last;
```

```
    result = last = null;
```

```
    while (L != null) {
```

```
        IntList next = L.tail;
```

```
        if (x != L.head) {
```

```
            if (last == null)
```

```
                result = last = L;
```

```
            else
```

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

```
            L.tail = null;
```

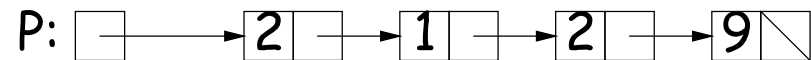
```
        }
```

```
        L = next;
```

```
    }
```

```
    return result;
```

```
}
```



result:

last:

L:

next:

P = dremoveAll (P, 2)

Iterative Destructive Deletion

/** The list resulting from removing all instances of X from L.

* Original contents of L may be destroyed. */

static IntList dremoveAll (IntList L, int x) {

IntList result, last;

result = last = null;

while (L != null) {

IntList next = L.tail;

if (x != L.head) {

if (last == null)

result = last = L;

else

last = last.tail = L;

L.tail = null;

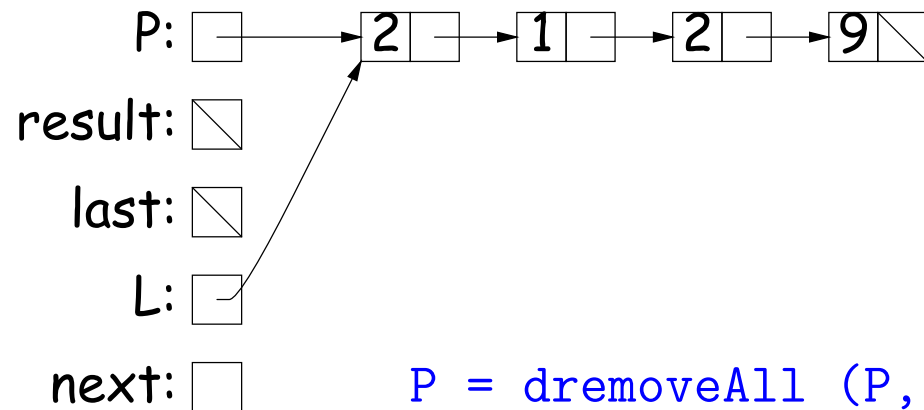
}

L = next;

}

return result;

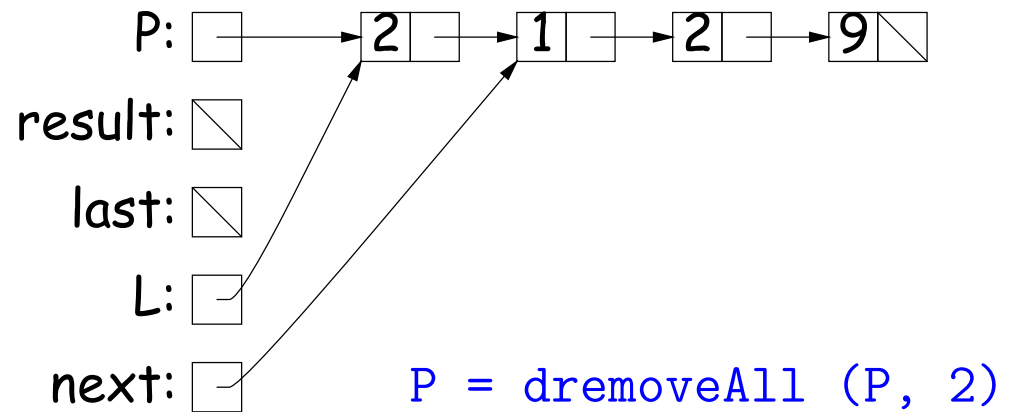
}



P = dremoveAll (P, 2)

Iterative Destructive Deletion

```
/** The list resulting from removing all instances of X from L.
 * Original contents of L may be destroyed. */
static IntList dremoveAll (IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```



Iterative Destructive Deletion

```
/** The list resulting from removing all instances of X from L.
```

```
 * Original contents of L may be destroyed. */
```

```
static IntList dremoveAll (IntList L, int x) {
```

```
    IntList result, last;
```

```
    result = last = null;
```

```
    while (L != null) {
```

```
        IntList next = L.tail;
```

```
        if (x != L.head) {
```

```
            if (last == null)
```

```
                result = last = L;
```

```
            else
```

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

```
            L.tail = null;
```

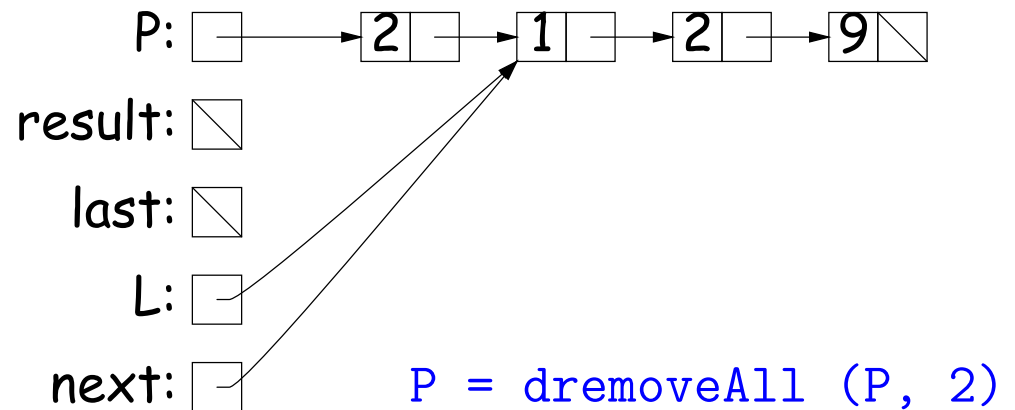
```
        }
```

```
        L = next;
```

```
    }
```

```
    return result;
```

```
}
```



Iterative Destructive Deletion

```
/** The list resulting from removing all instances of X from L.
```

```
 * Original contents of L may be destroyed. */
```

```
static IntList dremoveAll (IntList L, int x) {
```

```
    IntList result, last;
```

```
    result = last = null;
```

```
    while (L != null) {
```

```
        IntList next = L.tail;
```

```
        if (x != L.head) {
```

```
            if (last == null)
```

```
                result = last = L;
```

```
            else
```

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

```
            L.tail = null;
```

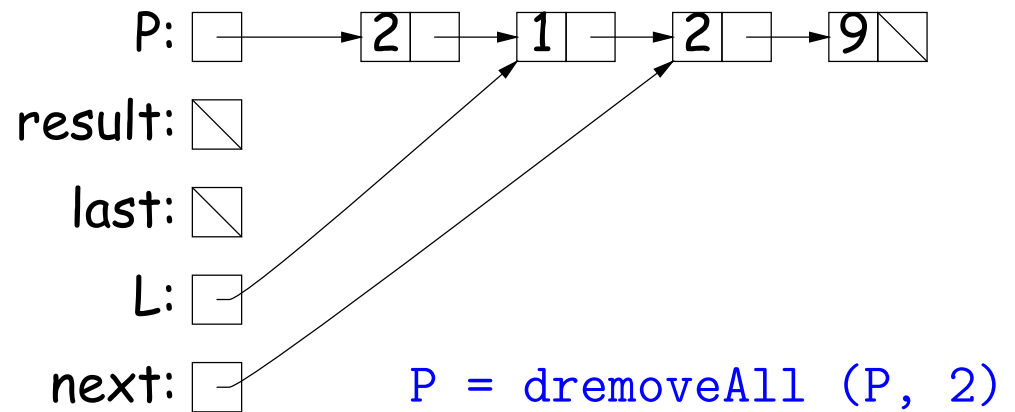
```
        }
```

```
        L = next;
```

```
    }
```

```
    return result;
```

```
}
```



Iterative Destructive Deletion

```
/** The list resulting from removing all instances of X from L.
```

```
 * Original contents of L may be destroyed. */
```

```
static IntList dremoveAll (IntList L, int x) {
```

```
    IntList result, last;
```

```
    result = last = null;
```

```
    while (L != null) {
```

```
        IntList next = L.tail;
```

```
        if (x != L.head) {
```

```
            if (last == null)
```

```
                result = last = L;
```

```
            else
```

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

```
            L.tail = null;
```

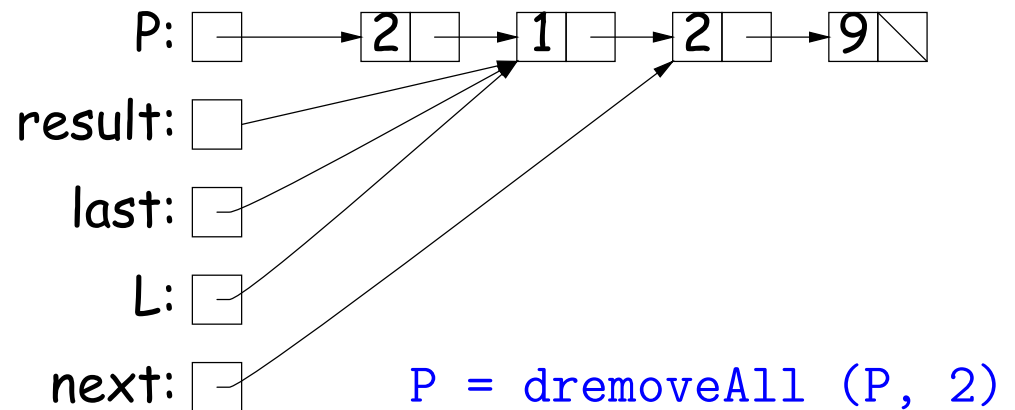
```
        }
```

```
        L = next;
```

```
    }
```

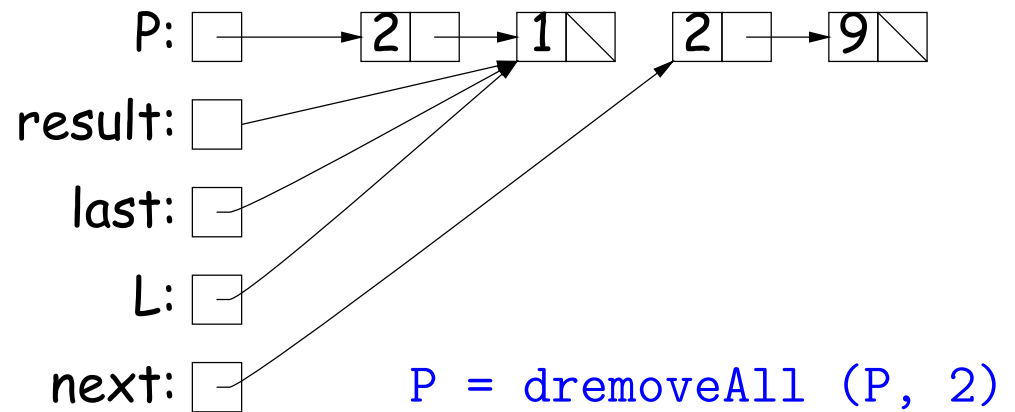
```
    return result;
```

```
}
```



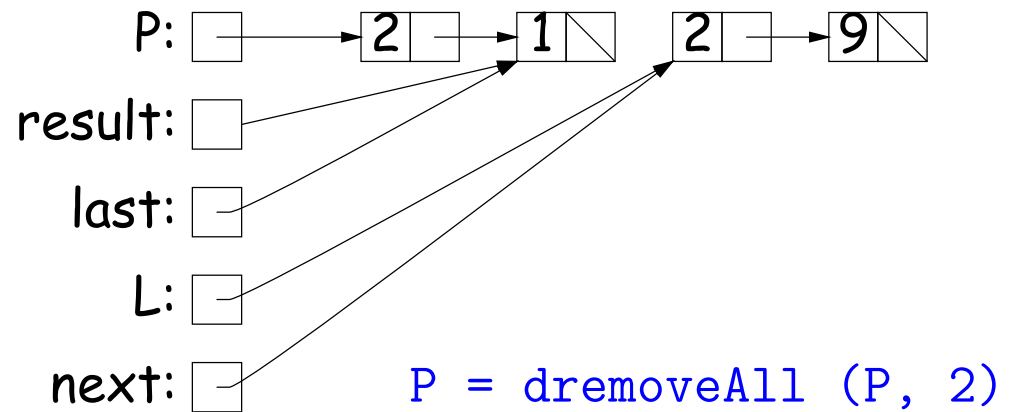
Iterative Destructive Deletion

```
/** The list resulting from removing all instances of X from L.
 * Original contents of L may be destroyed. */
static IntList dremoveAll (IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```



Iterative Destructive Deletion

```
/** The list resulting from removing all instances of X from L.
 * Original contents of L may be destroyed. */
static IntList dremoveAll (IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```



Iterative Destructive Deletion

/** The list resulting from removing all instances of X from L.

* Original contents of L may be destroyed. */

static IntList dremoveAll (IntList L, int x) {

IntList result, last;

result = last = null;

while (L != null) {

IntList next = L.tail;

if (x != L.head) {

if (last == null)

result = last = L;

else

last = last.tail = L;

L.tail = null;

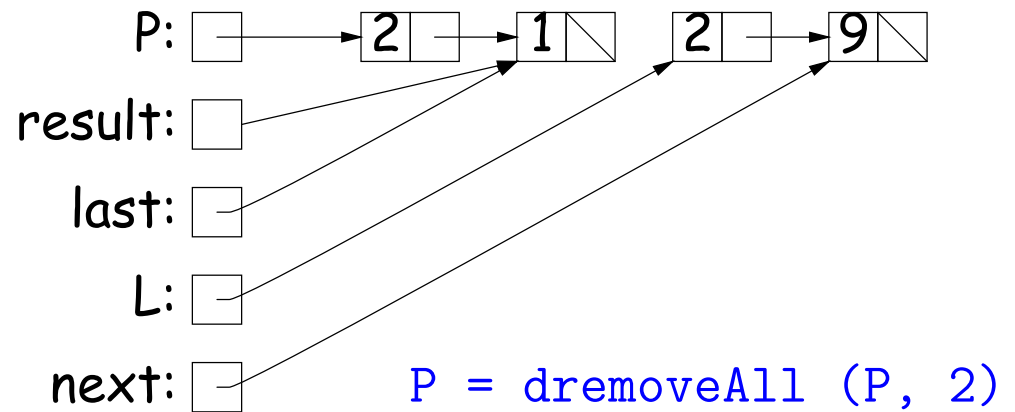
}

L = next;

}

return result;

}

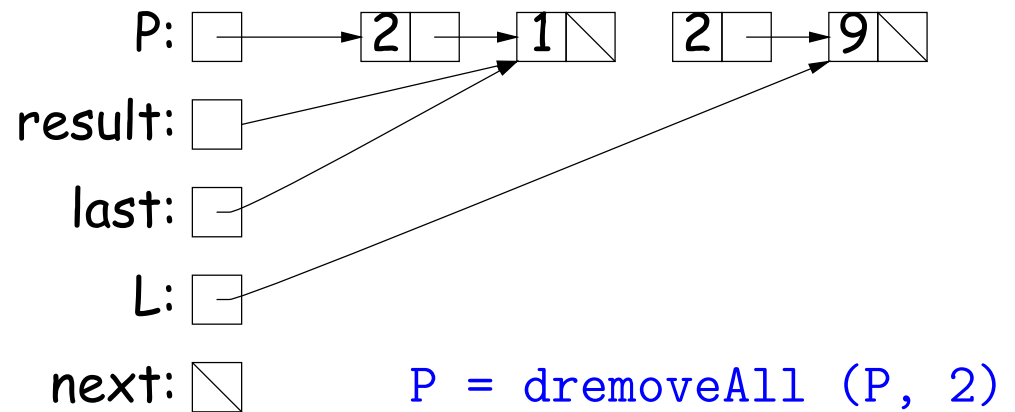


Iterative Destructive Deletion

```

/** The list resulting from removing all instances of X from L.
 * Original contents of L may be destroyed. */
static IntList dremoveAll (IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}

```



Iterative Destructive Deletion

/** The list resulting from removing all instances of X from L.

* Original contents of L may be destroyed. */

static IntList dremoveAll (IntList L, int x) {

IntList result, last;

result = last = null;

while (L != null) {

IntList next = L.tail;

if (x != L.head) {

if (last == null)

result = last = L;

else

last = last.tail = L;

L.tail = null;

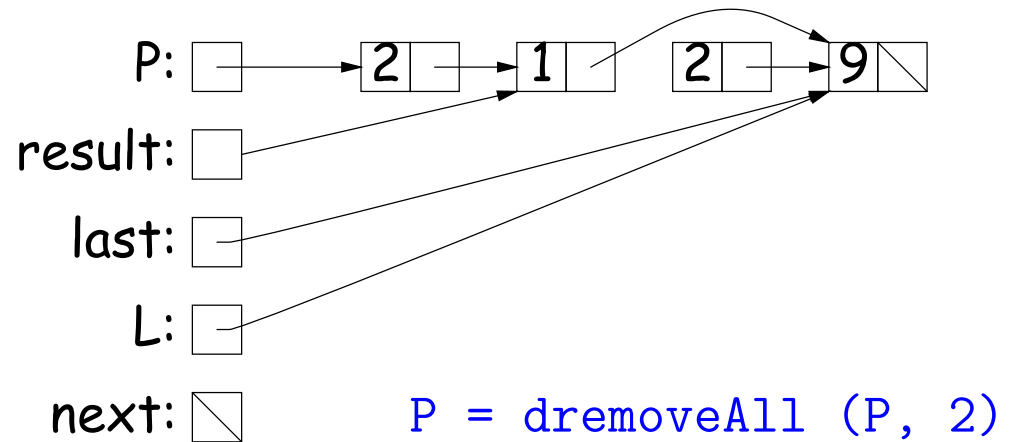
}

L = next;

}

return result;

}



Iterative Destructive Deletion

/** The list resulting from removing all instances of X from L.

* Original contents of L may be destroyed. */

static IntList dremoveAll (IntList L, int x) {

IntList result, last;

result = last = null;

while (L != null) {

IntList next = L.tail;

if (x != L.head) {

if (last == null)

result = last = L;

else

last = last.tail = L;

L.tail = null;

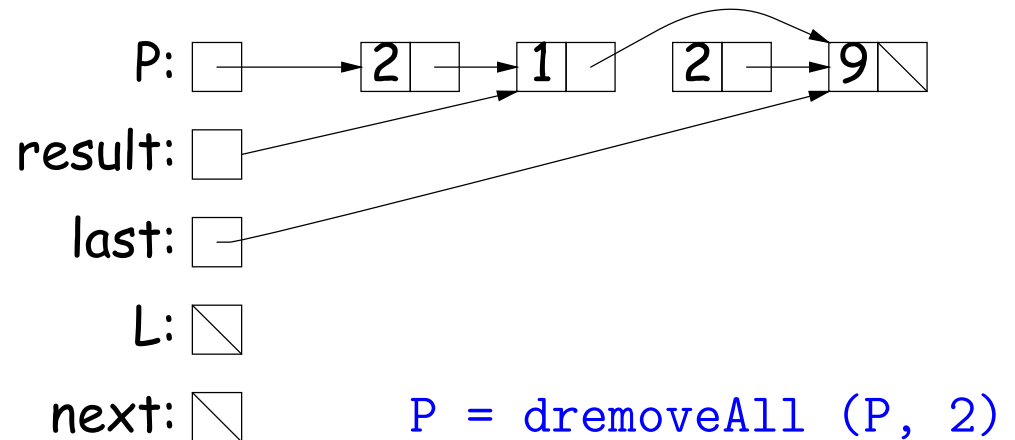
}

L = next;

}

return result;

}



Iterative Destructive Deletion

```
/** The list resulting from removing all instances of X from L.
 * Original contents of L may be destroyed. */
static IntList dremoveAll (IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```

