

CS61B Lecture #4: Simple Pointer Manipulation

Announcements:

- Discussion 114 (3-4 Thurs.) is now in 289 Cory (used to be 3111 Etch.)
- Next week and (maybe) from then on, discussion 114 will be in 3102 Etcheverry.

Public Service Announcement:

- Residential Computing, which provides tech support in residence halls, is currently hiring in programming, marketing, system administration, and more. Flexible hours and work study, \$12.74 to \$19.32 an hour. Applications due Wednesday, February 1st, 2006; see the notice on www.rescomp.berkeley.edu.

Today: More pointer hacking.

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);

The diagram illustrates the state of pointers and the list after the execution of the code. It shows four pointers: X, Q, L, and P. Pointer X points to a node containing 3. Pointer Q is empty. Pointer L points to the head of a list: 3 → 43 → 56. Pointer P points to the tail of the list, which is the node containing 56. Arrows show the flow of pointers: X points to the first node of the list; Q receives the value of X; L points to the head of the list; and P points to the last node of the list.

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);

The diagram illustrates the state of pointers after the destructive incrementation. It shows four variables: X, Q, L, and P. X and Q are both pointing to the same list structure: a sequence of three boxes containing the numbers 3, 43, and 56 respectively. An arrow points from X to the first box. Another arrow points from Q to the first box. A third arrow points from L to the first box. A fourth arrow points from P to L. After the incrementation, the list structure changes. The first box now contains the number 5. The second box contains 43. The third box contains 56. The fourth box, which previously contained a diagonal line, now also contains a diagonal line. The arrows from X, Q, and P remain pointing to the first box of the modified list. The arrow from L points to the second box of the modified list.

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);

The diagram illustrates the state of pointers X, Q, L, and P relative to a linked list. The list consists of three nodes with values 5, 43, and 56. The pointer L points to the first node (5). The pointer P points to the second node (43). The pointer X points to the third node (56). The pointer Q is shown as a blank box. Arrows indicate the flow of pointers: one arrow from X to the third node, another from Q to the same node, and a third from P to the second node. This visualizes how the dincrList function's modification of the list through P and L also affects the original list's tail pointer, effectively incrementing all elements.

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);

The diagram illustrates the state of pointers X, Q, L, and P relative to a linked list. The list consists of three nodes with values 5, 45, and 56. The pointer L points to the first node (5). The pointer P points to the second node (45). The pointer X points to the third node (56). The pointer Q is shown as a blank box. Arrows indicate the flow of pointers: one arrow points from X to the third node, another from Q to the same node, and a third from P to the second node. This visualizes how the function dincrList (Q) is modifying the list starting from the node pointed to by P.

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);

The diagram illustrates the state of pointers X, Q, L, and P relative to a list of integers. Pointer X points to a node containing 3. Pointer Q points to a node containing 43. Pointer L points to a node containing 5, which is the head of the list. Pointer P points to a node containing 56, which is the tail of the list. An arrow from X to L indicates that X now points to the head of the list. Another arrow from P to the tail of the list indicates that P no longer points to the tail of the list.

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);

The diagram illustrates the state of pointers X, Q, L, and P relative to a list of integers. Pointer X points to a node containing 3. Pointer Q points to a node containing 43. Pointer L points to a node containing 5, which is the head of the list. Pointer P points to a node containing 58, which is the tail of the list. An arrow from X to L indicates that X now points to the head of the list L. Another arrow from P to L indicates that P now points to the tail of the list L.

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;
    }
}

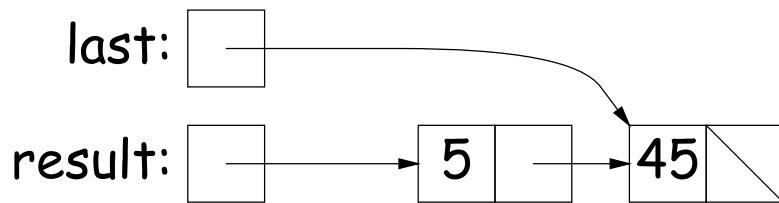
/** List L destructively incremented
 * by n. */
static IntList dincrList (IntList L, int n) {
    // 'for' can do more than count!
    for (IntList p = L; p != null; p = p.tail)
        p.head += n;
    return L;
}
```

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

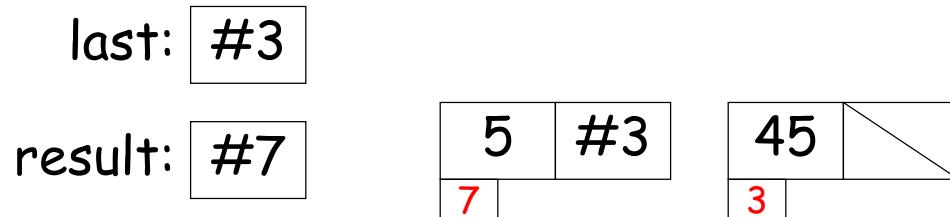
The diagram illustrates the state of pointers X, Q, L, and P after the execution of the code. Pointer X points to a node containing 3. Pointer Q points to a node containing 43. Pointer L points to a node containing 5, which is the head of a list. Pointer P points to a node containing 58, which is the tail of the list. Arrows show that X and Q now point to the same node as L, indicating they have been modified to point to the new head of the list. The original list nodes (3, 43, 58) remain, but their original pointers (X, Q, and the tail of L) now point to the same node as the new head (5).

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;
```

```
}
```

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

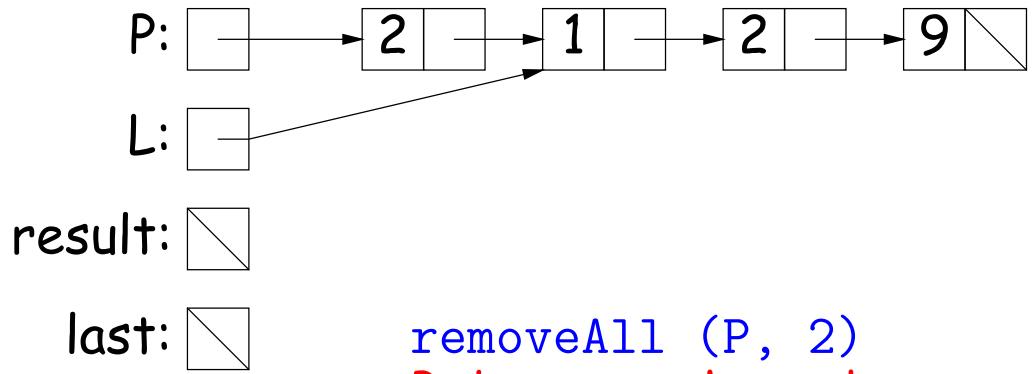
Result is all elements of L_0 not equal to x up to and not including L , and $last$ points to the last element of result, if any. We use L_0 here to mean "the original value of L ."

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;
}
```



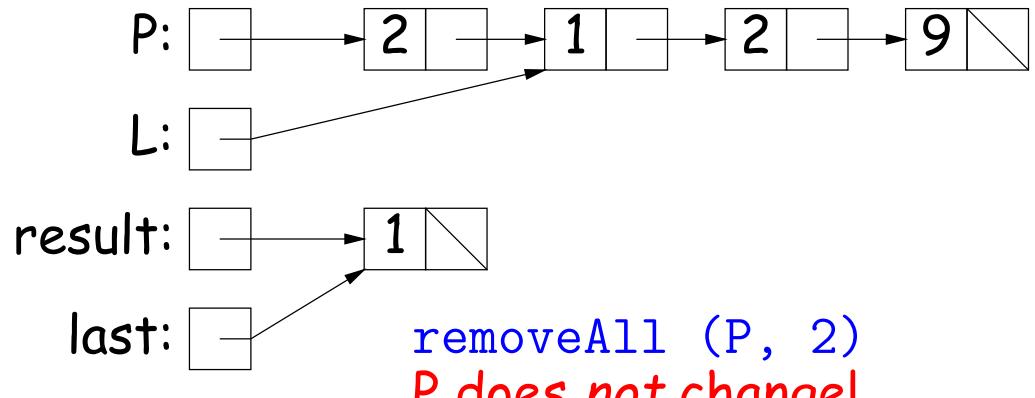
Here, \mathcal{I} is the *loop invariant*:

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;
}
```



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

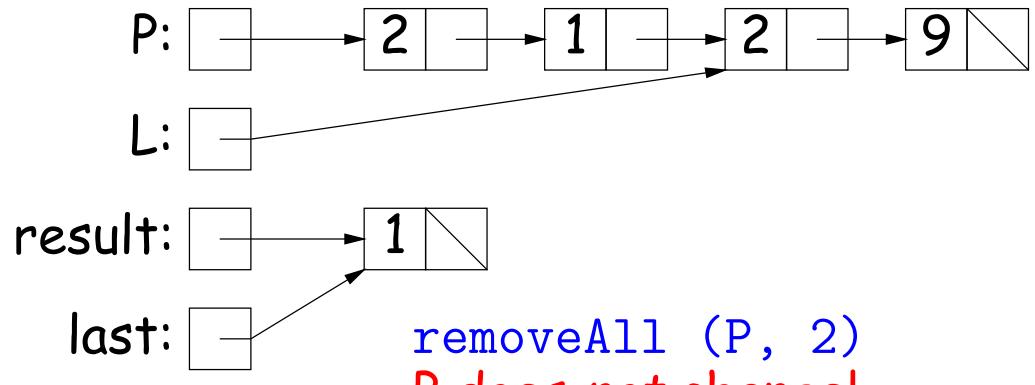
Result is all elements of L_0 not equal to x up to and not including L , and $last$ points to the last element of result, if any. We use L_0 here to mean "the original value of L ."

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!

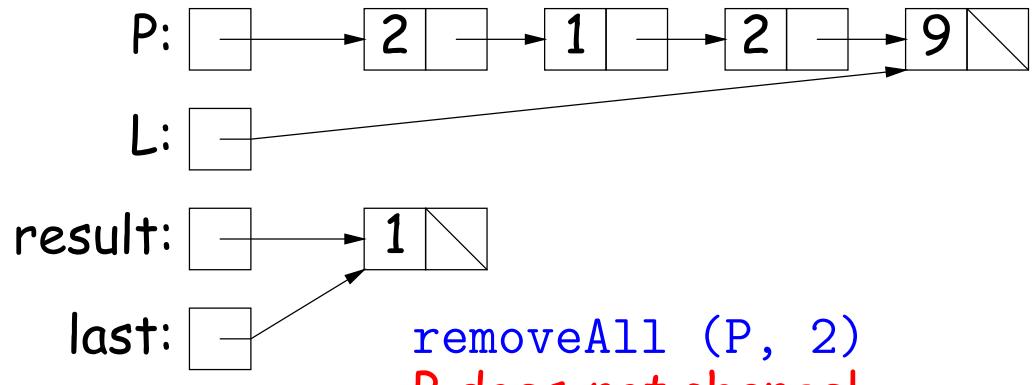
Here, \mathcal{I} is the *loop invariant*:

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!

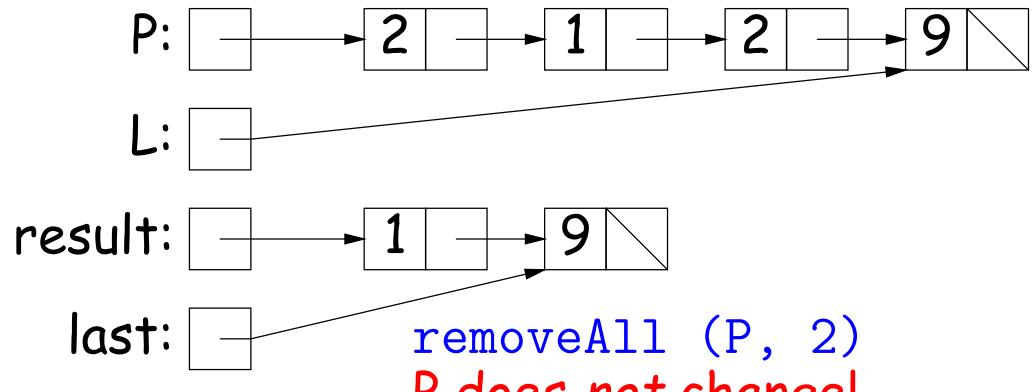
Here, \mathcal{I} is the *loop invariant*:

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*:

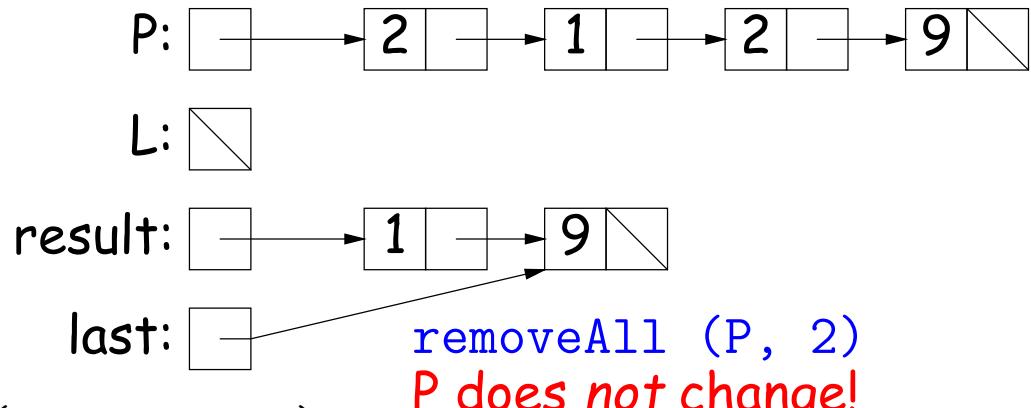
Result is all elements of L_0 not equal to x up to and not including L , and $last$ points to the last element of result, if any. We use L_0 here to mean "the original value of L ."

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;
}
```



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 ."

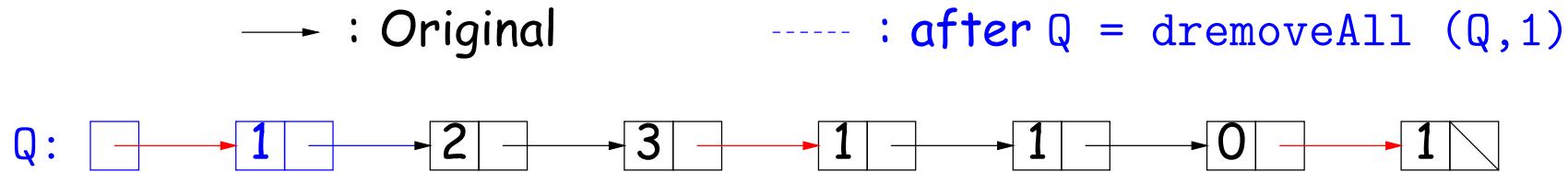
Aside: How to Write a Loop (in Theory)

- 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.
```

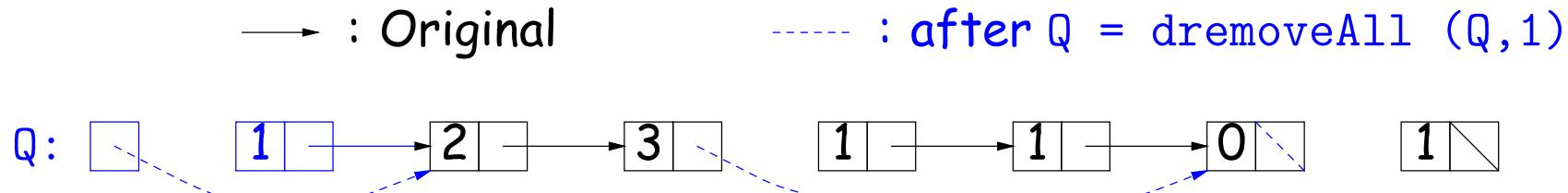
- So if (*invariant* and *not condition*) is enough to insure we've got the answer, we're done!

Destructive Deletion



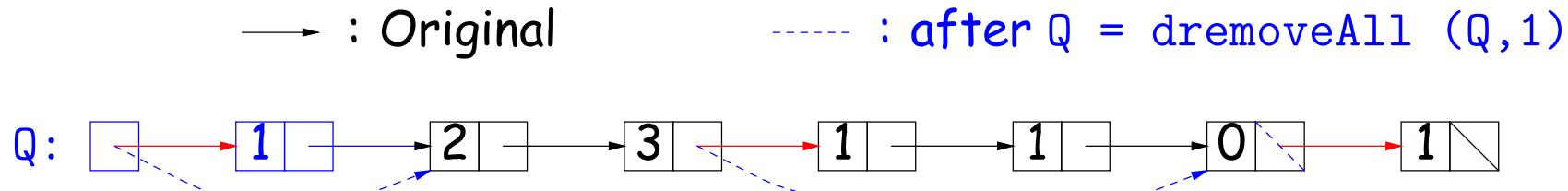
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll (IntList L, int x) {  
    if (L == null)  
        return /*( null with all x's removed );*/;  
    else if (L.head == x)  
        return /*( L with all x's removed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

Destructive Deletion



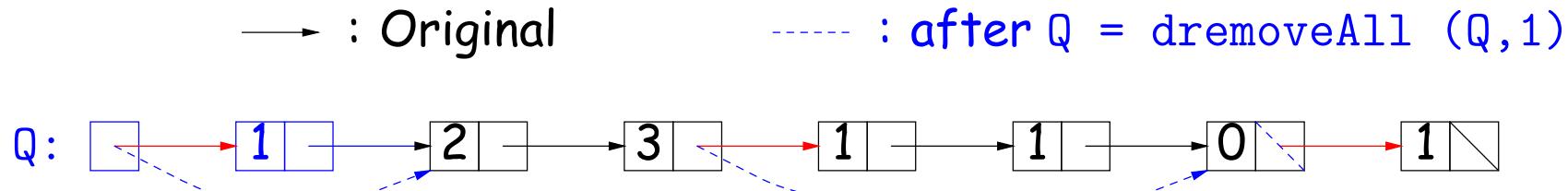
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll (IntList L, int x) {  
    if (L == null)  
        return /*( null with all x's removed );*/;  
    else if (L.head == x)  
        return /*( L with all x's removed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

Destructive Deletion



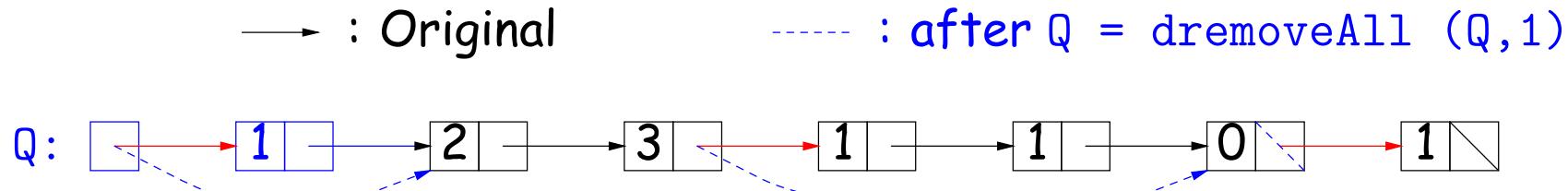
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll (IntList L, int x) {  
    if (L == null)  
        return /*( null with all x's removed );*/;  
    else if (L.head == x)  
        return /*( L with all x's removed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

Destructive Deletion



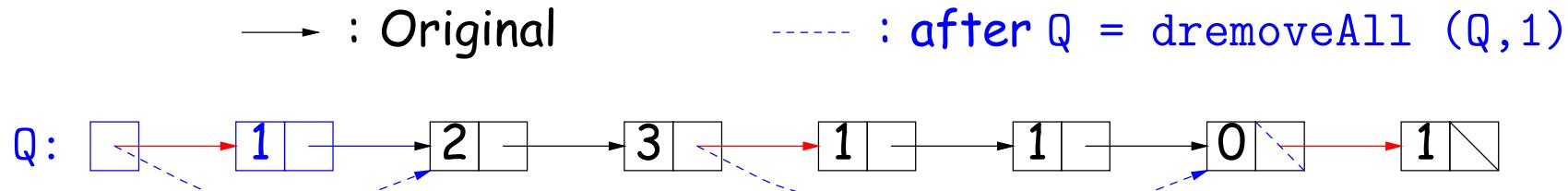
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll (IntList L, int x) {  
    if (L == null)  
        return /*( null with all x's removed );*/;  
    else if (L.head == x)  
        return /*( L with all x's removed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

Destructive Deletion



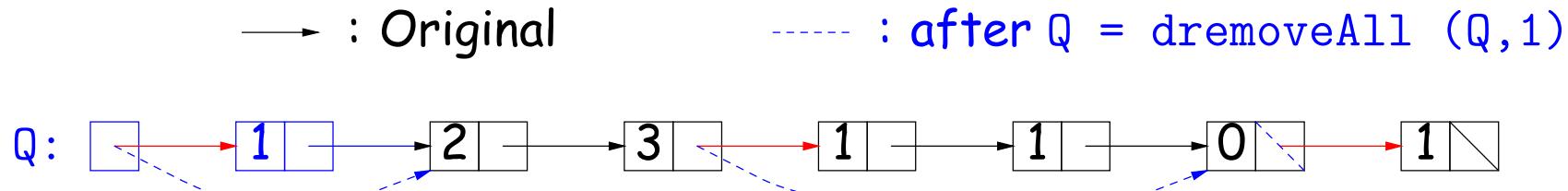
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll (IntList L, int x) {  
    if (L == null)  
        return null;  
    else if (L.head == x)  
        return /*( L with all x's removed (L != null) );*/;  
   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

Destructive Deletion



```
/** 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



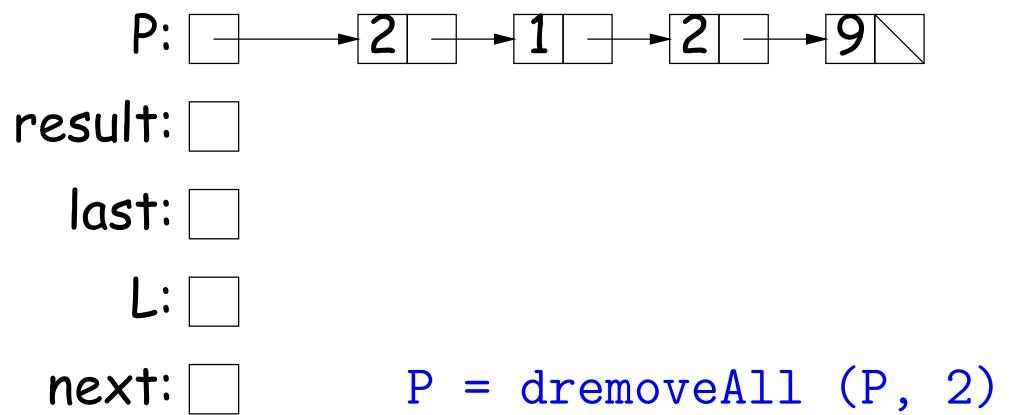
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll (IntList L, int x) {  
    if (L == null)  
        return null;  
    else if (L.head == x)  
        return dremoveAll (L.tail, x);  
    else {  
        L.tail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

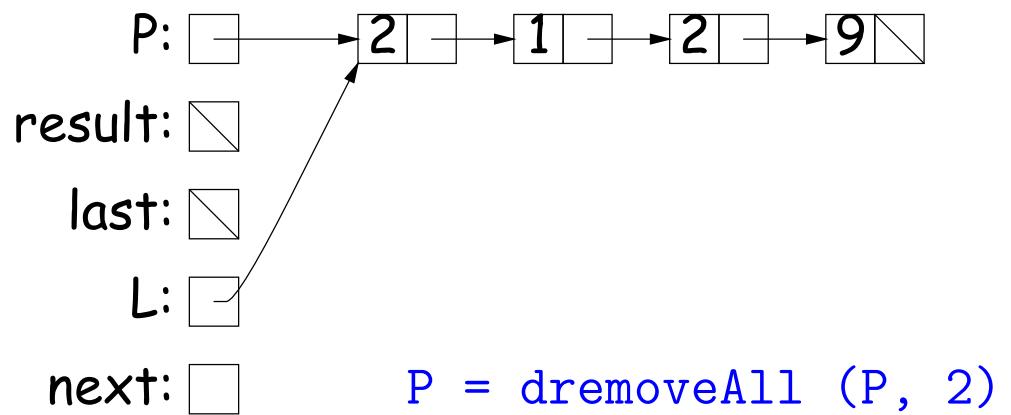
Iterative Destructive Deletion

```
/** 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;  
}
```



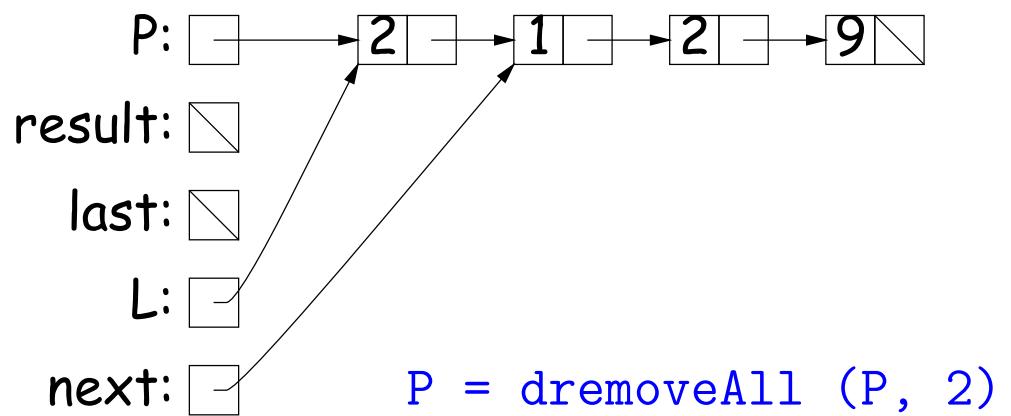
Iterative Destructive Deletion

```
/** 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;  
}
```



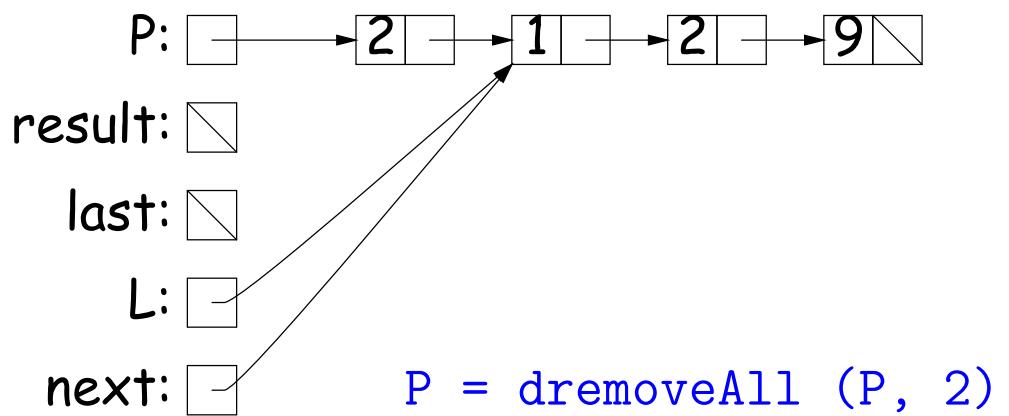
Iterative Destructive Deletion

```
/** 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;  
}
```



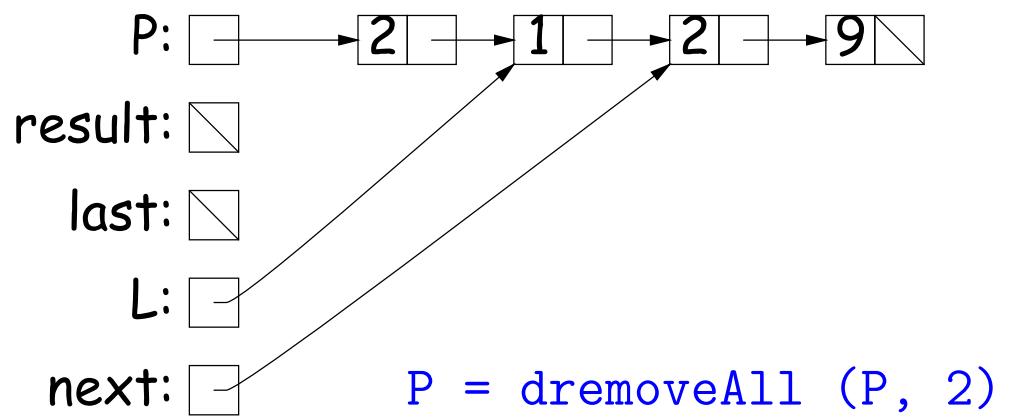
Iterative Destructive Deletion

```
/** 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;  
}
```



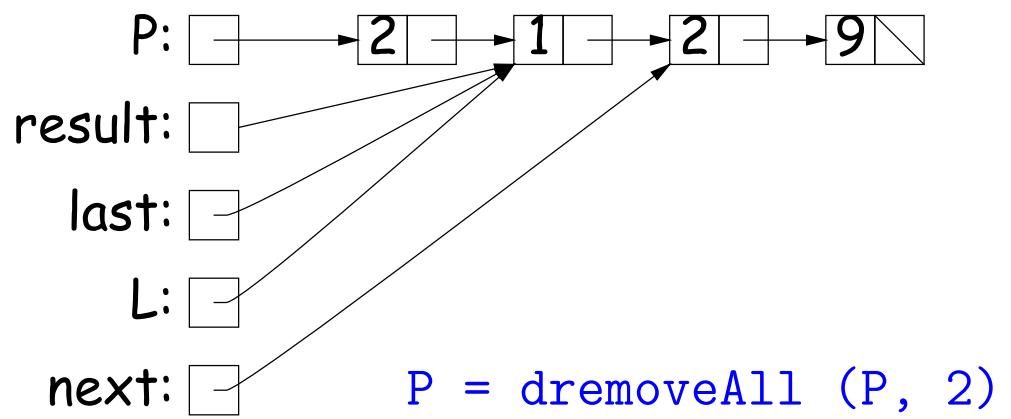
Iterative Destructive Deletion

```
/** 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;  
}
```



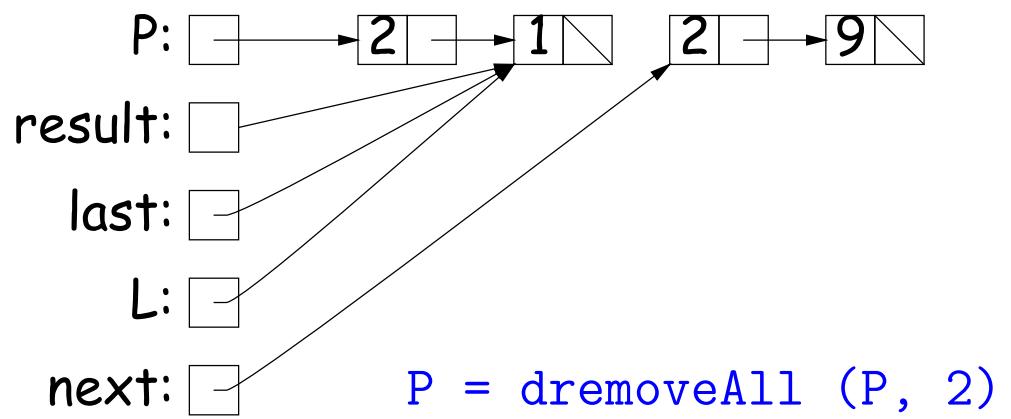
Iterative Destructive Deletion

```
/** 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;  
}
```



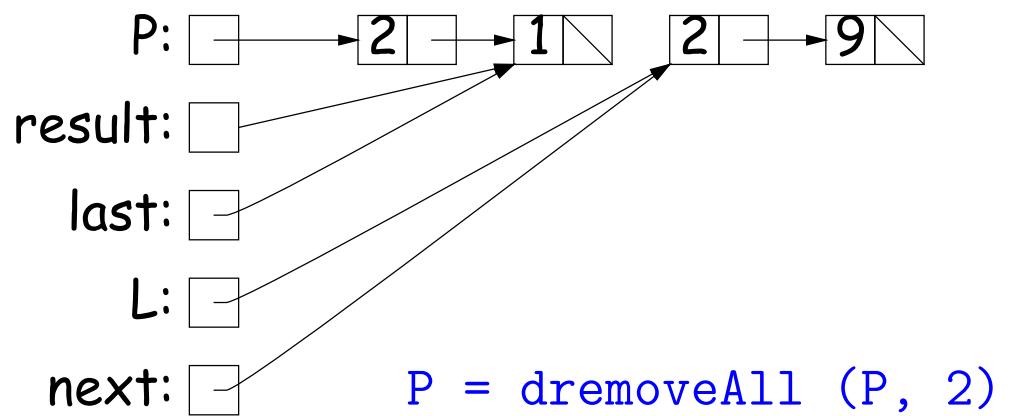
Iterative Destructive Deletion

```
/** 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;  
}
```



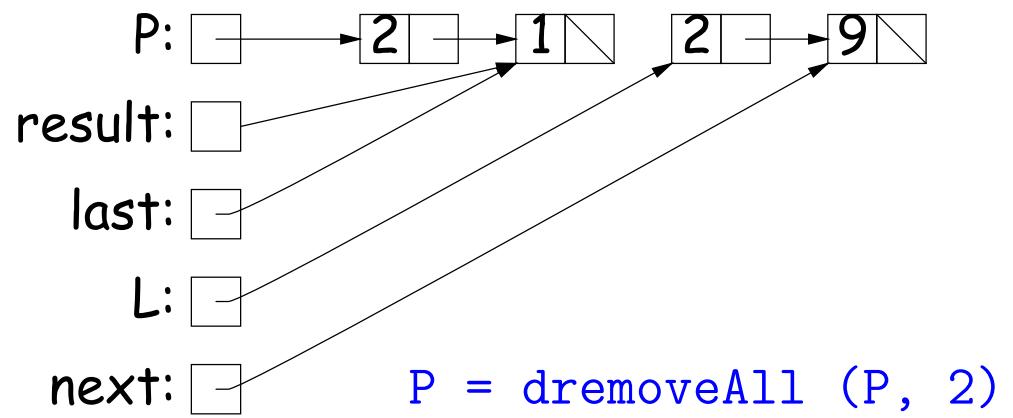
Iterative Destructive Deletion

```
/** 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;  
}
```



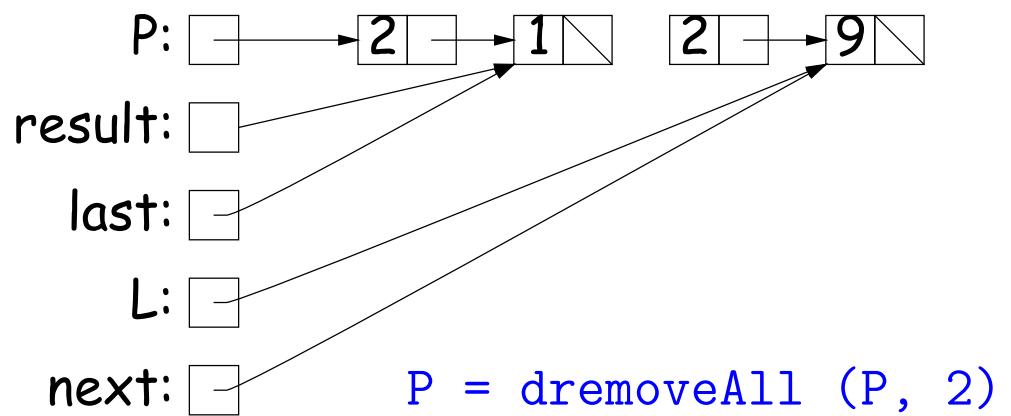
Iterative Destructive Deletion

```
/** 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;  
}
```



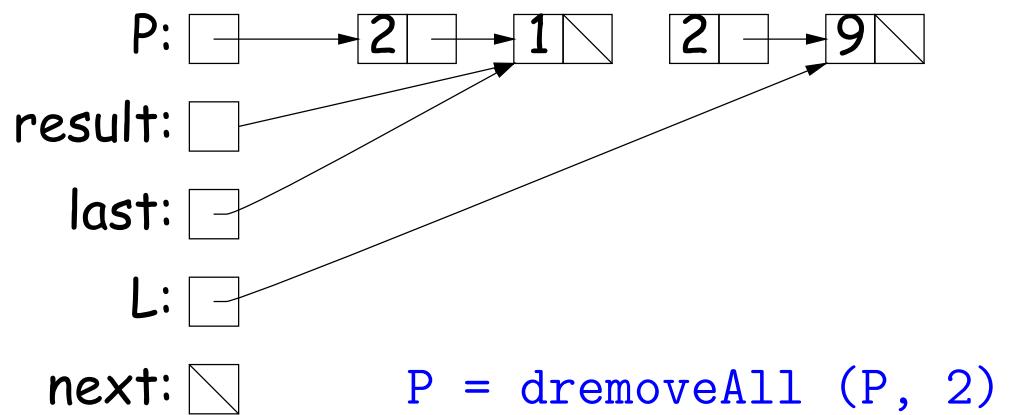
Iterative Destructive Deletion

```
/** 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;  
}
```



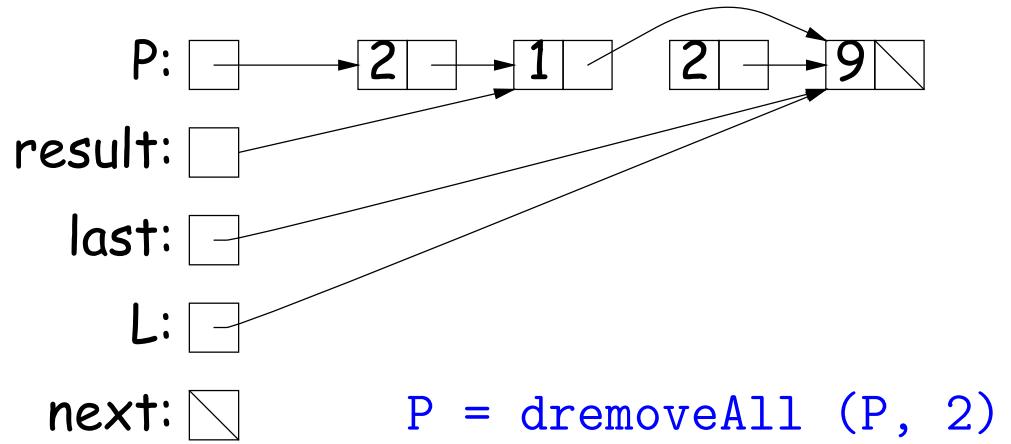
Iterative Destructive Deletion

```
/** 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;  
}
```



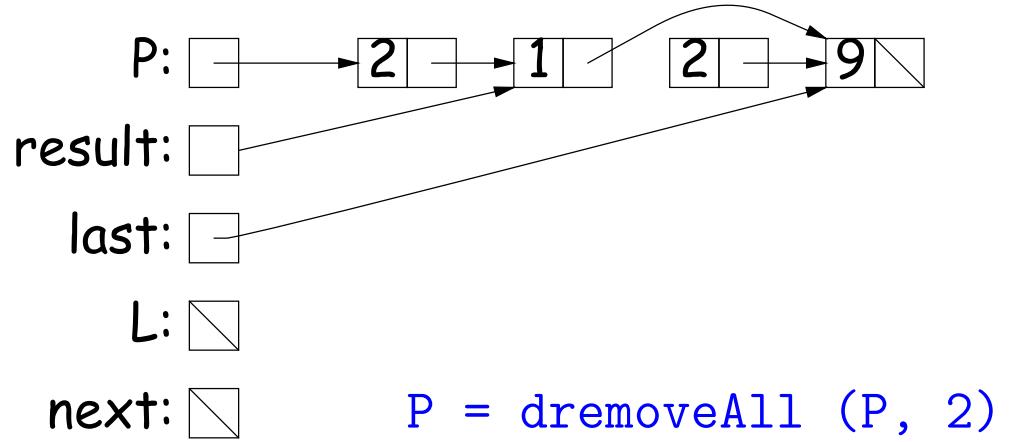
Iterative Destructive Deletion

```
/** 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;  
}
```



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;  
}
```

