

CS61B Lecture #5: Simple Pointer Manipulation

Announcement

- Today: More pointer hacking.
- Handing in labs and homework: We'll be lenient about accepting late homework and labs for the first few. Just get it done: part of the point is getting to understand the tools involved. We will **not** accept submissions by email.
- For bugs, use bug-submit. There are instructions on the class home-page Announcements.

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);

X: → Q: → L: → P:

Q: → 5 → 45 → 58 →

L: → 3 → 43 → 56 →

P: → 5 → 45 → 58 →

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 the head of a list [3, 43, 56]. Pointer Q points to the head of a list [5, 43, 56]. Pointer L points to the head of a list [5, 43, 56]. Pointer P points to the head of a list [5, 43, 56]. Arrows show the flow of pointers: X points to the first node of the list; Q points to the second node; L points to the third node; and P points to the fourth node. The list nodes are represented as boxes containing the values 5, 43, and 56, with arrows indicating their sequence.

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 function dincrList (Q) is modifying the list starting from node 56, which is also the tail of the original list 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 a list: 5 -> 45 -> 56 -> null. Pointer P points to a node containing 56, which is the tail of the list. Arrows show the flow of pointers: X points to the first node, Q points to the second node, L points to the head of the list, and P points to the last node. The list nodes are represented as boxes with horizontal lines through them, indicating they are mutable structures.

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 linked list. 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, and Q points to the previous head (now 43). A diagonal slash through the tail node of L indicates that the list has been modified.

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 58. Node 58 has a diagonal slash through its tail pointer. Pointer X points to the first node. Pointer Q points to the second node. Pointer L points to the first node. Pointer P points to the third node. Arrows show the flow of pointers: X and Q point to the first node, which is also pointed to by L. P points to the third node, which is the tail of the list.

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));
}
```

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!

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 sequence of int values in 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  $I$  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;
```

```
}
```

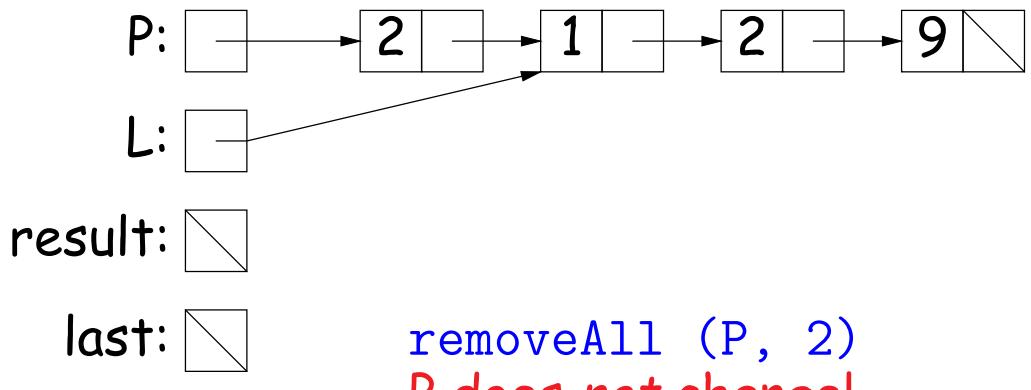
Here, I is the *loop invariant*:

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 sequence of int values in 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;
}
```



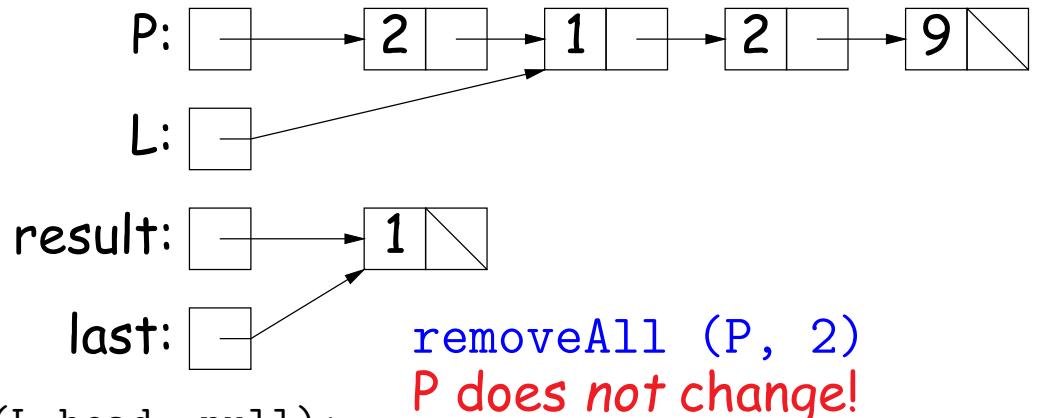
Here, \mathcal{I} is the *loop invariant*:

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 sequence of int values in 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;
}
```



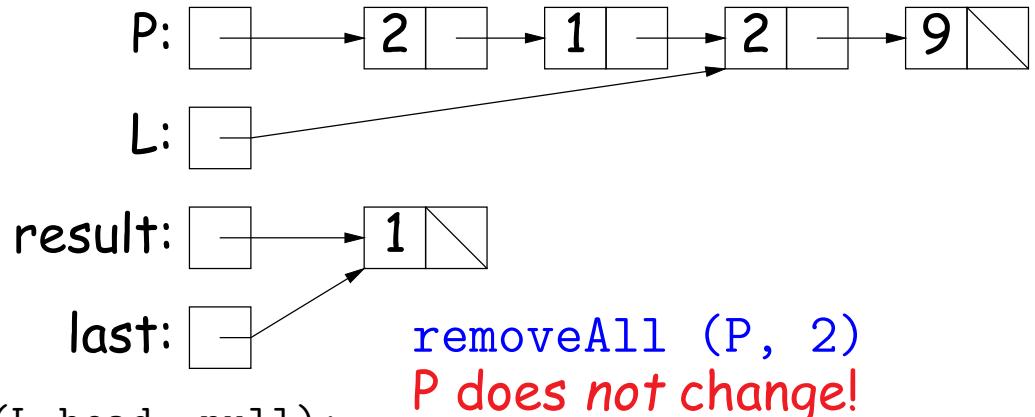
Here, \mathcal{I} is the *loop invariant*:

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 sequence of int values in 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;
}
```



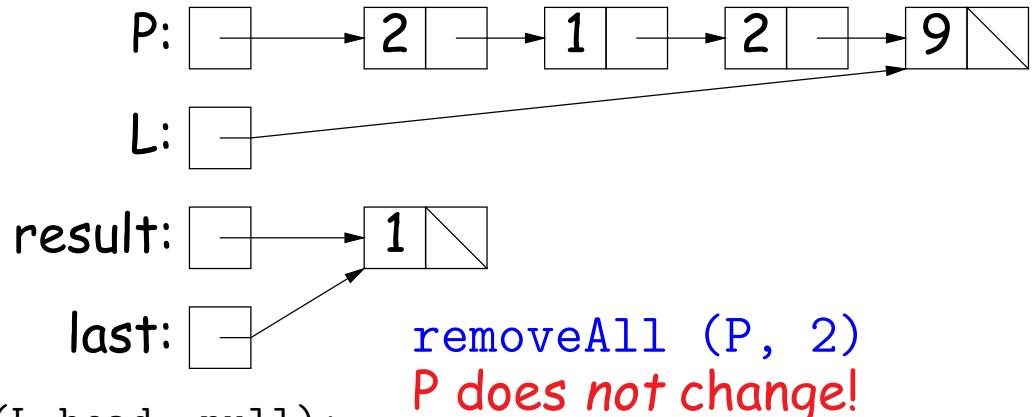
Here, \mathcal{I} is the *loop invariant*:

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 sequence of int values in 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  $I$  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;
}
```



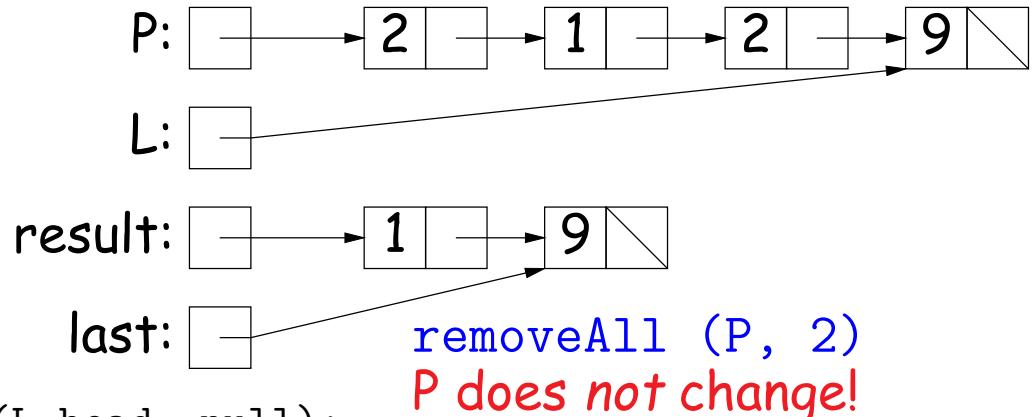
Here, I is the loop invariant:

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 sequence of int values in 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  $I$  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;
}
```



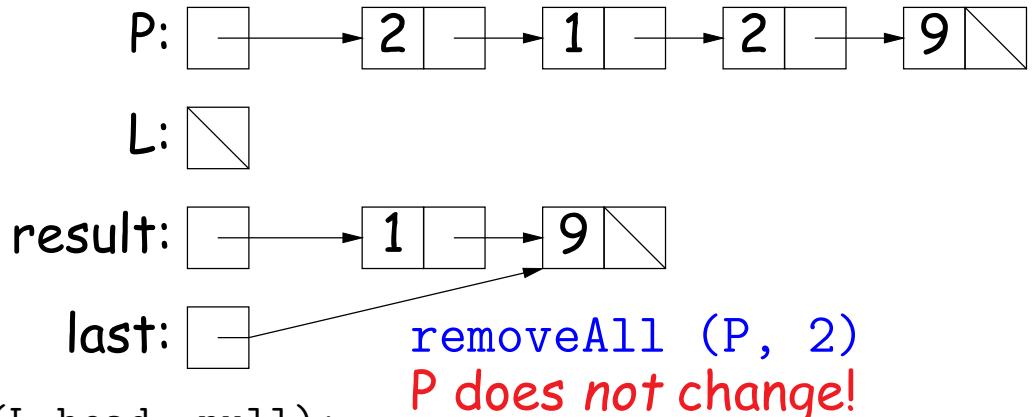
Here, I is the *loop invariant*:

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 sequence of int values in L ."

Iterative Non-destructive List Deletion

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

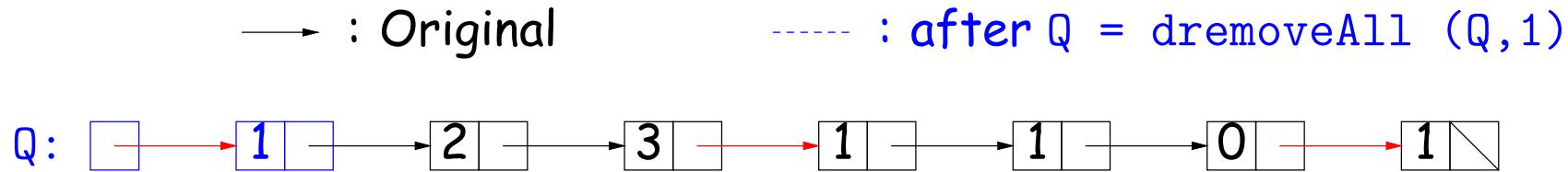
```
/** The list resulting from removing all instances of X from L
 * non-destructively. */
static IntList removeAll (IntList L, int x) {
    IntList result, last;
    result = last = null;
    for ( ; L != null; L = L.tail) {
        /* L != null and  $\mathcal{I}$  is true. */
        if (x == L.head)
            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:

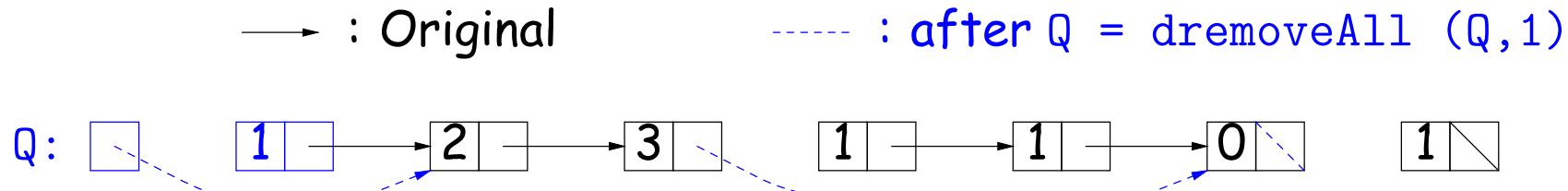
Result is all elements of L_0 not equal to x up to and not including L , and $last$ points to the last element of $result$, if any. We use L_0 here to mean "the original sequence of int values in L ."

Destructive Deletion



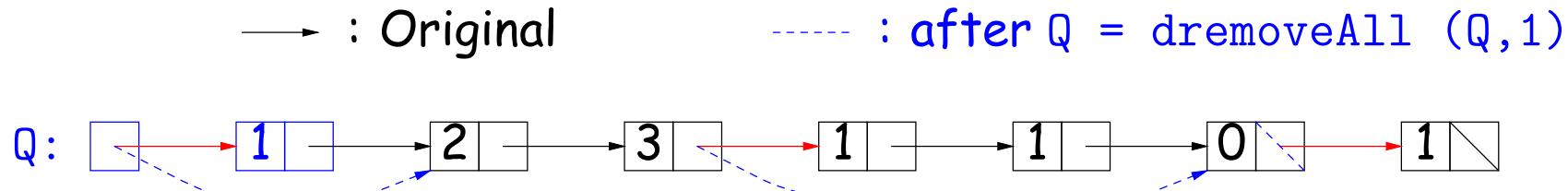
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll (IntList L, int x) {  
    if (L == null)  
        return /*( null with all x's removed );*/;  
    else if (L.head == x)  
        return /*( L with all x's removed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

Destructive Deletion



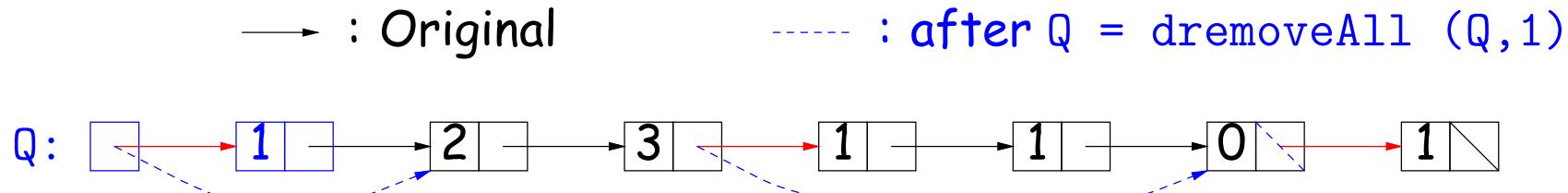
```
/** The list resulting from removing all instances of X from L.
 * The original list may be destroyed. */
static IntList dremoveAll (IntList L, int x) {
    if (L == null)
        return /*( null with all x's removed );*/
    else if (L.head == x)
        return /*( L with all x's removed (L != null)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

Destructive Deletion



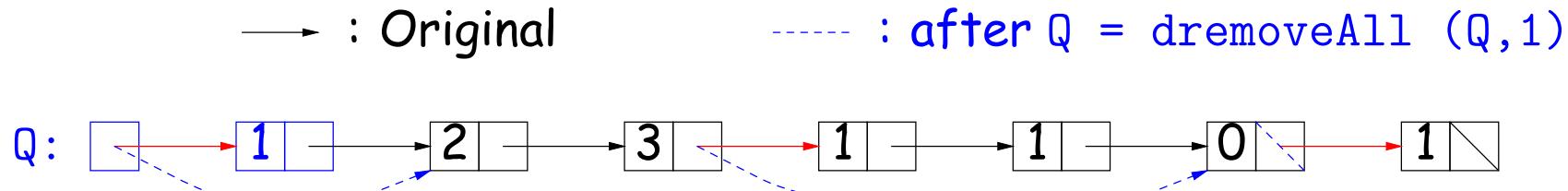
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll (IntList L, int x) {  
    if (L == null)  
        return /*( null with all x's removed );*/;  
    else if (L.head == x)  
        return /*( L with all x's removed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

Destructive Deletion



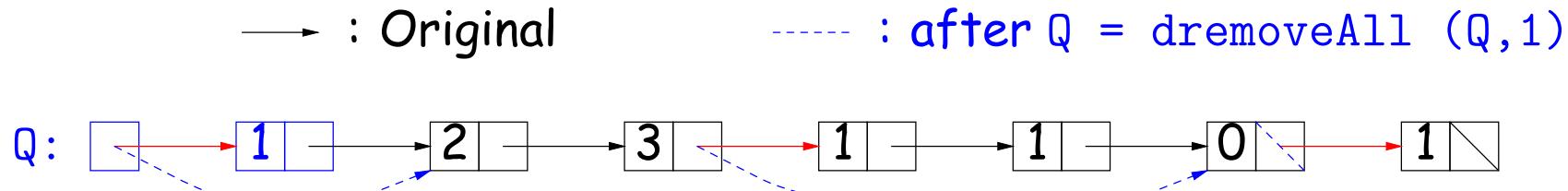
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll (IntList L, int x) {  
    if (L == null)  
        return /*( null with all x's removed );*/;  
    else if (L.head == x)  
        return /*( L with all x's removed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

Destructive Deletion



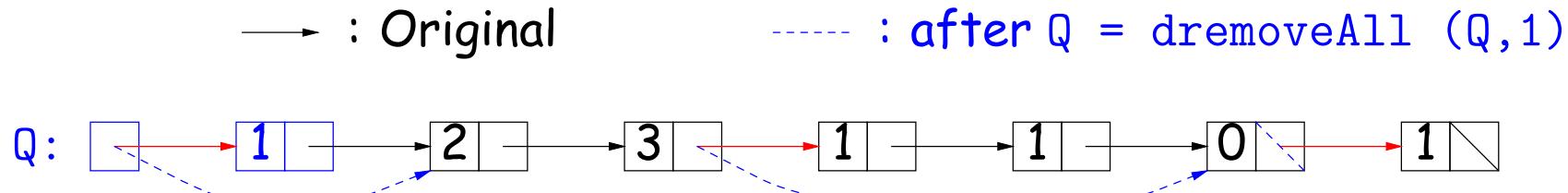
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll (IntList L, int x) {  
    if (L == null)  
        return null;  
    else if (L.head == x)  
        return /*( L with all x's removed (L != null) );*/;  
   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



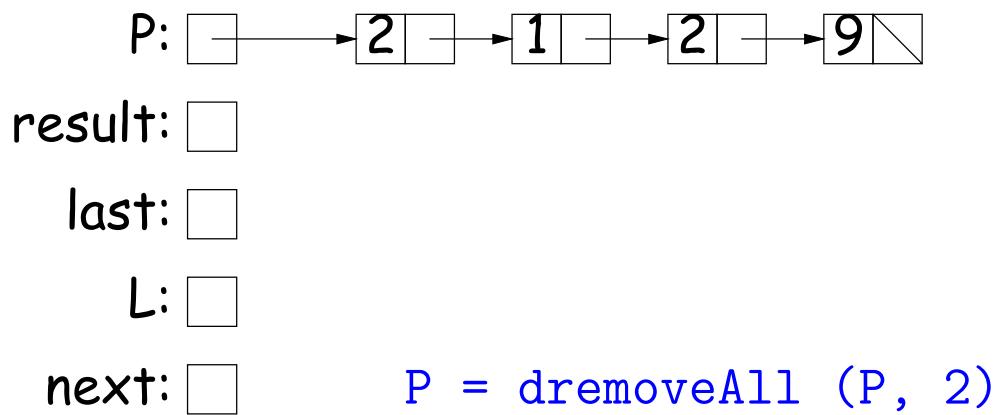
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll (IntList L, int x) {  
    if (L == null)  
        return null;  
    else if (L.head == x)  
        return dremoveAll (L.tail, x);  
    else {  
        L.tail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

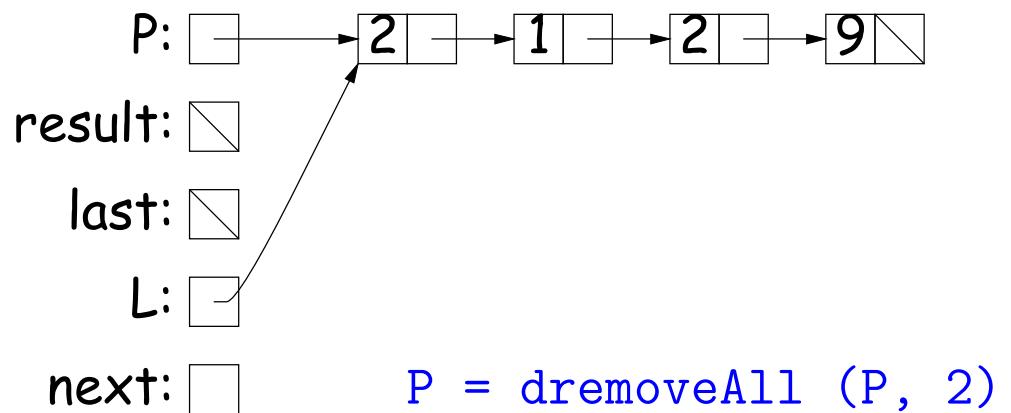
Iterative Destructive Deletion

```
/** 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;  
}
```



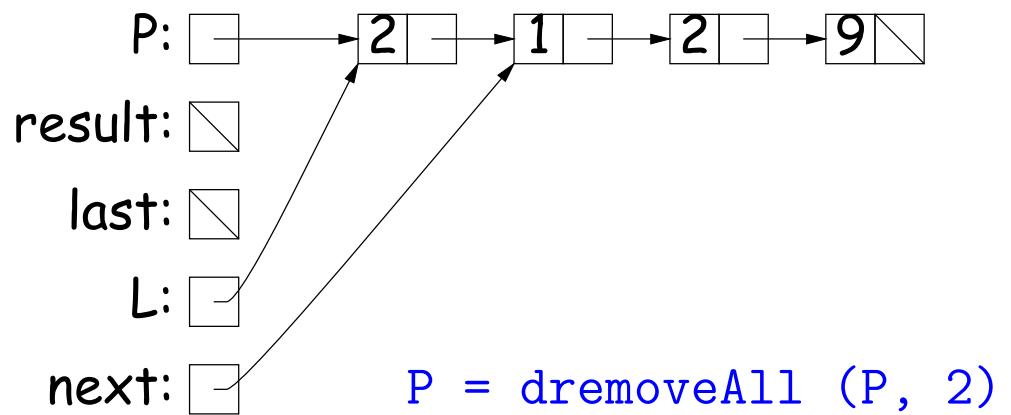
Iterative Destructive Deletion

```
/** 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;  
}
```



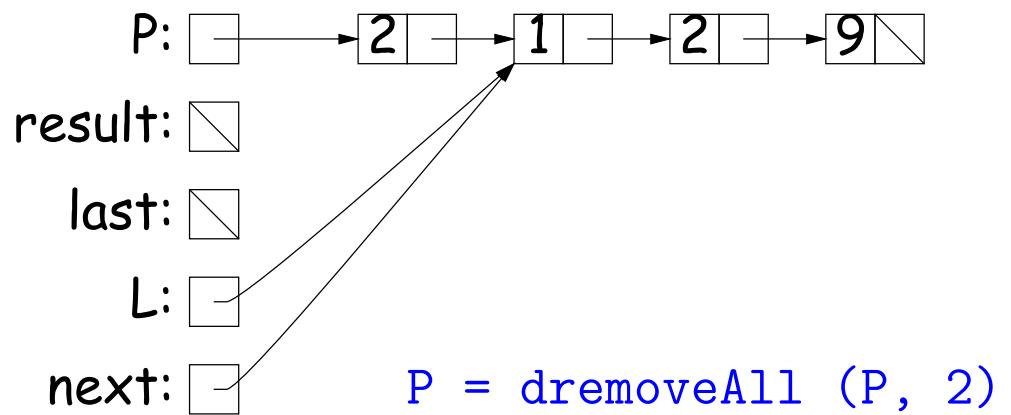
Iterative Destructive Deletion

```
/** 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;  
}
```



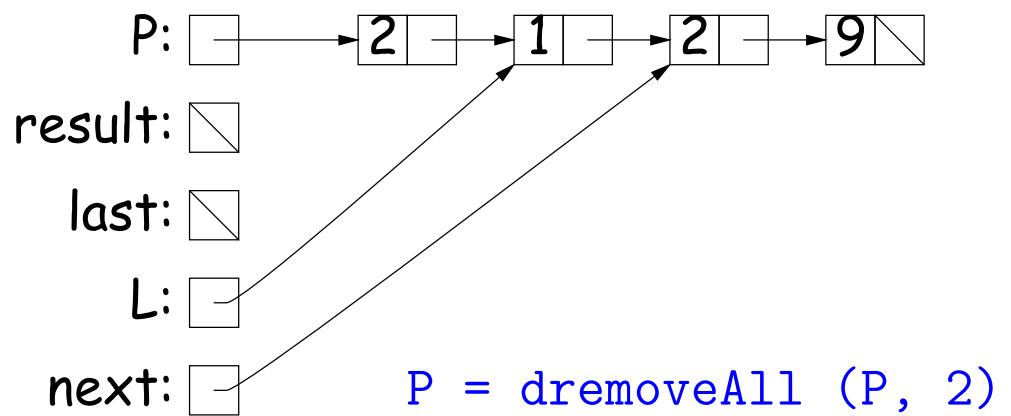
Iterative Destructive Deletion

```
/** 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;  
}
```



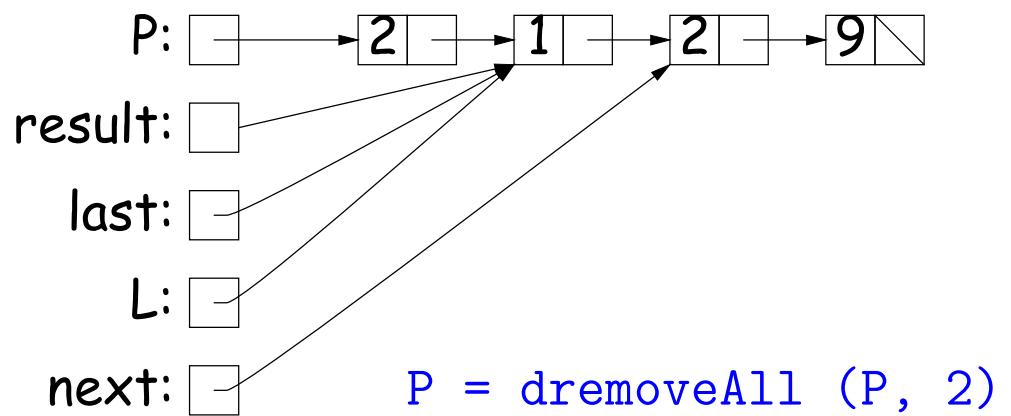
Iterative Destructive Deletion

```
/** 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;  
}
```



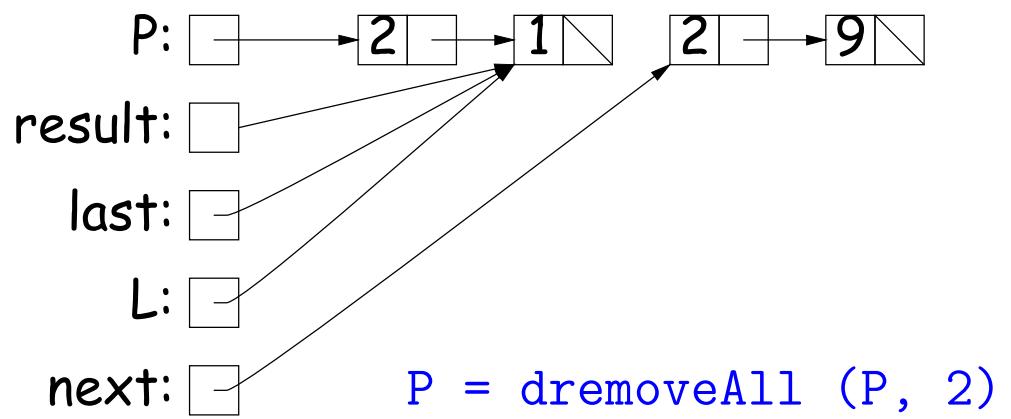
Iterative Destructive Deletion

```
/** 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;  
}
```



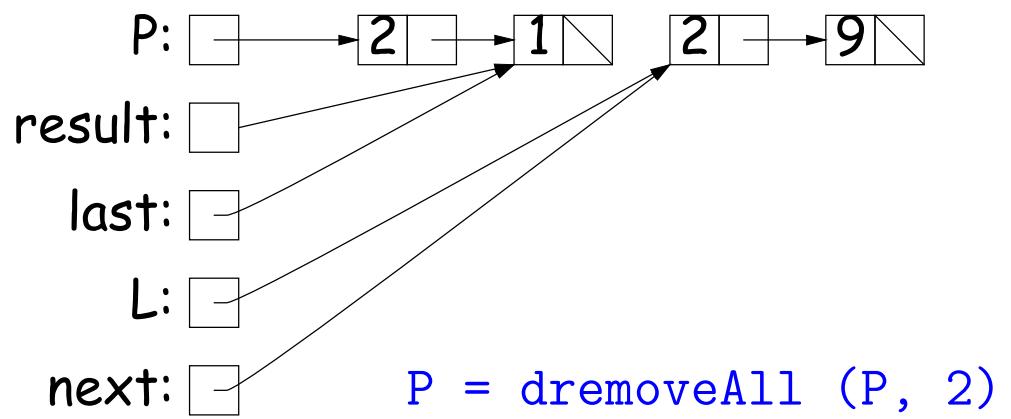
Iterative Destructive Deletion

```
/** 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;  
}
```



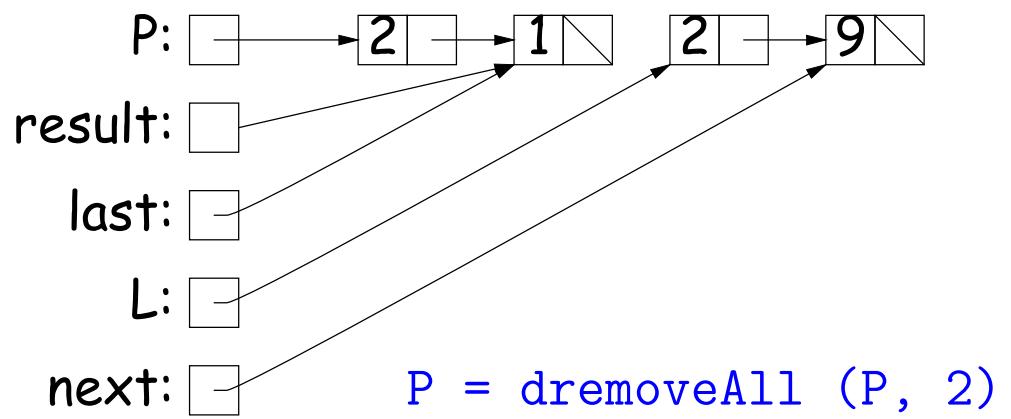
Iterative Destructive Deletion

```
/** 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;  
}
```



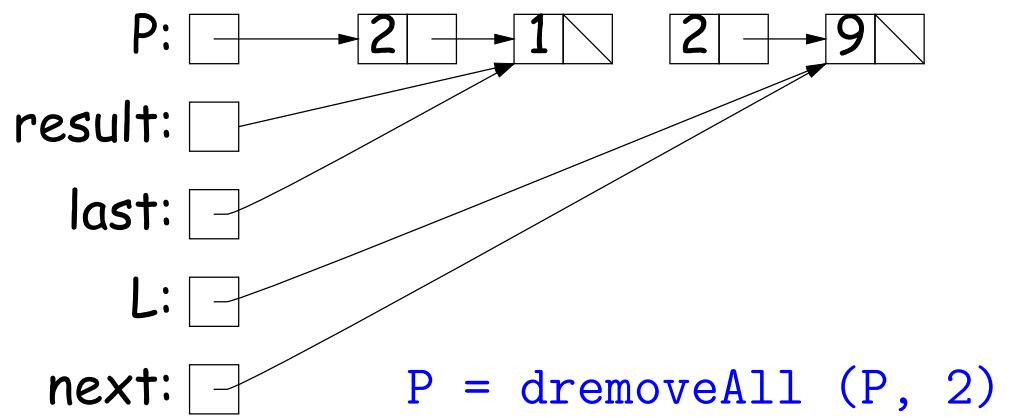
Iterative Destructive Deletion

```
/** 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;  
}
```



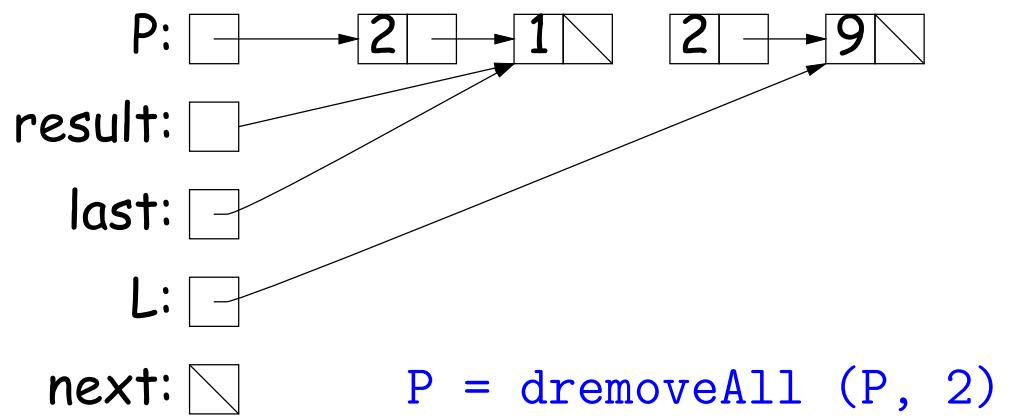
Iterative Destructive Deletion

```
/** 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;  
}
```



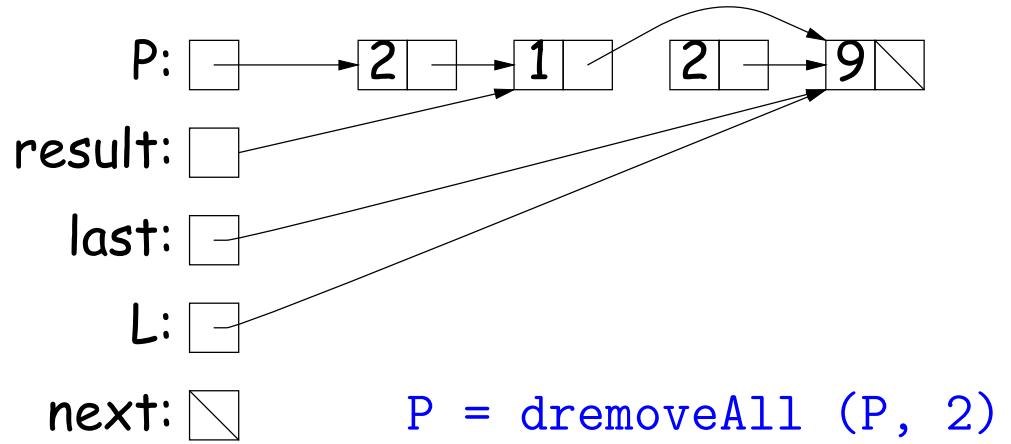
Iterative Destructive Deletion

```
/** 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;  
}
```



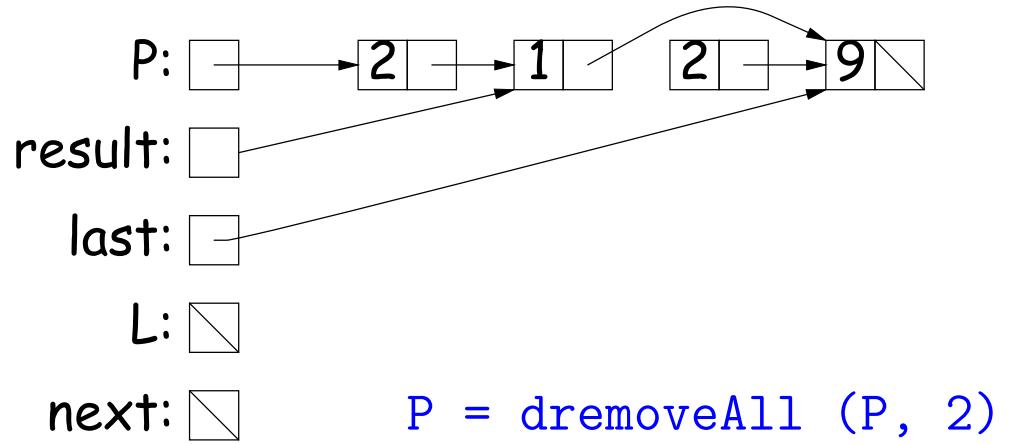
Iterative Destructive Deletion

```
/** 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;  
}
```

