

# Recreation

Prove that for every acute angle  $\alpha > 0$ ,

$$\tan \alpha + \cot \alpha \geq 2$$

# 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.

# Destructive Incrementing

Destructive solutions may modify objects in 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 L. Pointer X points to the head of list L. Pointer Q points to the tail of list L. Pointer L points to the first node of list L, which contains the value 3. Pointer P points to the tail of list L, which contains the value 56. Arrows show the flow of pointers: X points to L, Q points to L, and P points to the tail of L. The list L consists of three nodes with values 3, 43, and 56, with arrows indicating their connections.

# Destructive Incrementing

Destructive solutions may modify objects in 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 L. Pointer X points to the head of list L. Pointer Q points to the tail of list L. Pointer L points to the first node of list L, which contains the value 5. Pointer P points to the head of list L. Arrows show the flow of control from X to Q, and from L to P. The list L consists of three nodes with values 5, 43, and 56, with each node pointing to the next. The final state shows the list L with its head modified to 5, 45, and 58, while the original list's head remains at 5.

# Destructive Incrementing

Destructive solutions may modify objects in 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 L. Pointer X points to the head of list L. Pointer Q points to the tail of list L. Pointer L points to the first node of list L, which contains the value 5. Pointer P points to the head of list L. The list L consists of three nodes with values 5, 43, and 56, each pointing to the next node. The final state of list L is shown with the last node's next pointer pointing to a diagonal slash, indicating it is null.

# Destructive Incrementing

Destructive solutions may modify objects in 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 L. Pointer X points to the head of list L. Pointer Q points to the tail of list L. Pointer L points to the first node of list L, which contains the value 5. Pointer P points to the head of list L. The list L consists of three nodes with values 5, 45, and 56, with arrows indicating the sequence. The final state of list L is shown with the last node having a diagonal slash through its tail pointer, indicating it is no longer part of the list.

# Destructive Incrementing

Destructive solutions may modify objects in 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 L. Pointer X points to the head of list L. Pointer Q points to the tail of list L. Pointer L points to the first node of list L, which contains the value 5. Pointer P points to the head of list L. The list L consists of three nodes with values 5, 45, and 56, each pointing to the next node. The final state of list L is shown with the last node having a diagonal slash through its tail pointer, indicating it is no longer part of the list.

# Destructive Incrementing

Destructive solutions may modify objects in 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 L. Pointer X points to the head of list L. Pointer Q points to the tail of list L. Pointer L points to the first node of list L, which contains the value 5. Pointer P points to the head of list L. The list L consists of three nodes with values 5, 45, and 58. The final state of list L is shown with the last node having a diagonal slash through its tail pointer, indicating it is no longer part of the list.

# Destructive Incrementing

Destructive solutions may modify objects in 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 L. Pointer X points to the head of list L. Pointer Q also points to the head of list L. Pointer P points to the tail of list L. The list L itself is shown as a sequence of three nodes: 5 → 45 → 58. The node 58 has a diagonal slash through it, indicating it is no longer valid. Arrows show the flow of pointers: one from X to the first node of L, another from Q to the same node, and a third from P to the node containing 58.

## 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);
```

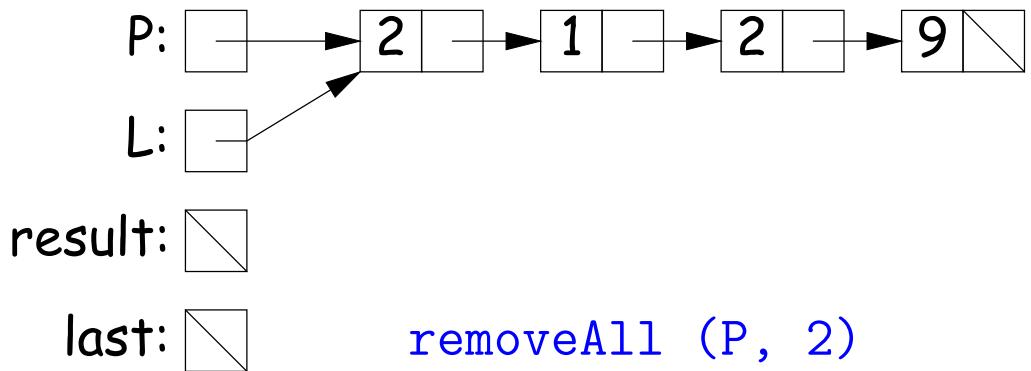
```
}
```

```
    return result;
```

```
}
```

Here,  $I$  is the *loop invariant*:

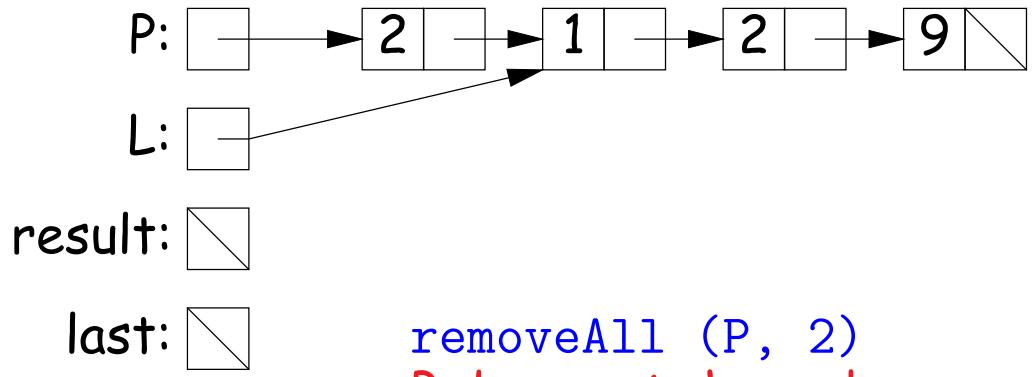
Result is all elements of  $L_0$  not equal to  $x$  up to and not including  $L$ , and  $last$  points to the last element of  $result$ , if any. We use  $L_0$  here to mean "the original sequence of int values in  $L$ ."



# 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  $I$  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:

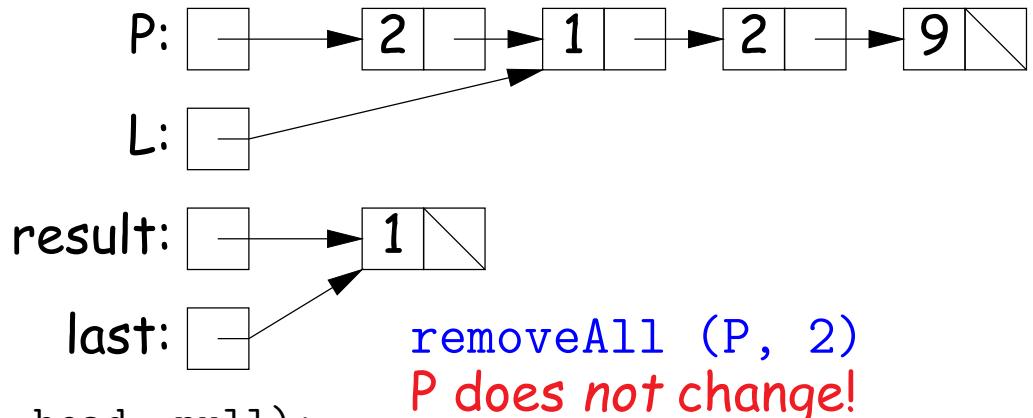
Result is all elements of  $L_0$  not equal to  $x$  up to and not including  $L$ , and  $last$  points to the last element of result, if any. We use  $L_0$  here to mean "the original sequence of int values in  $L$ ."

# 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;
}
```



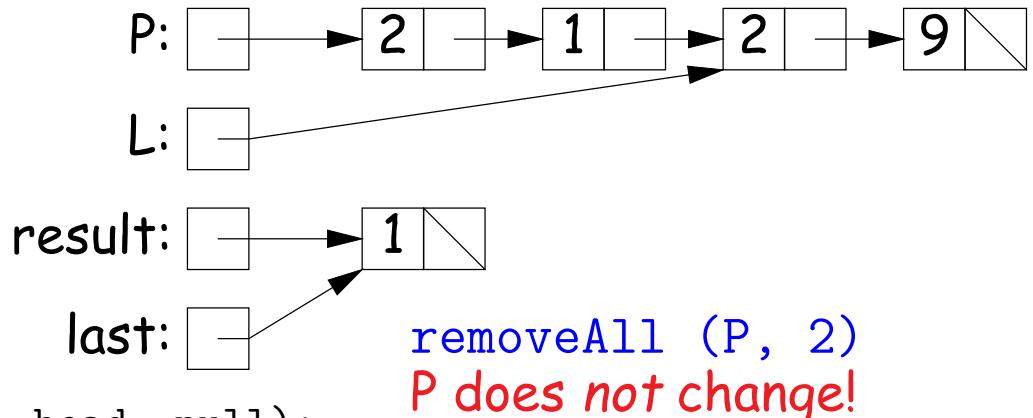
Here,  $\mathcal{I}$  is the loop invariant:

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;
}
```



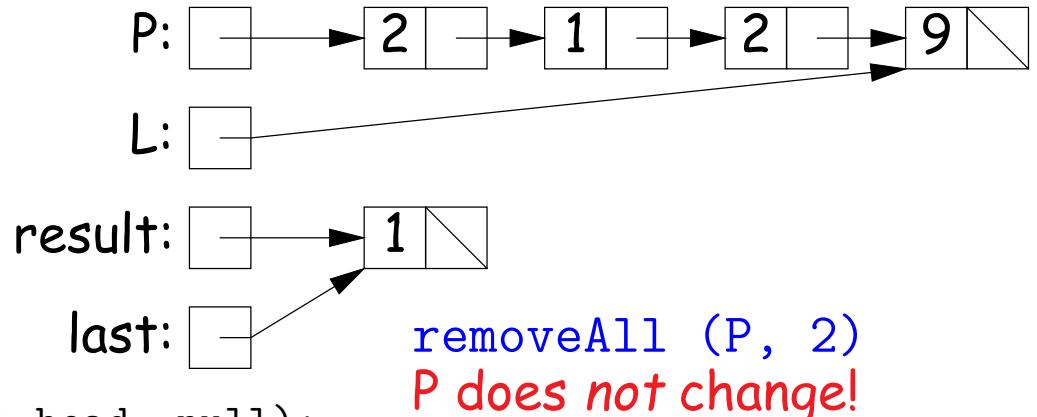
Here,  $\mathcal{I}$  is the loop invariant:

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;
}
```



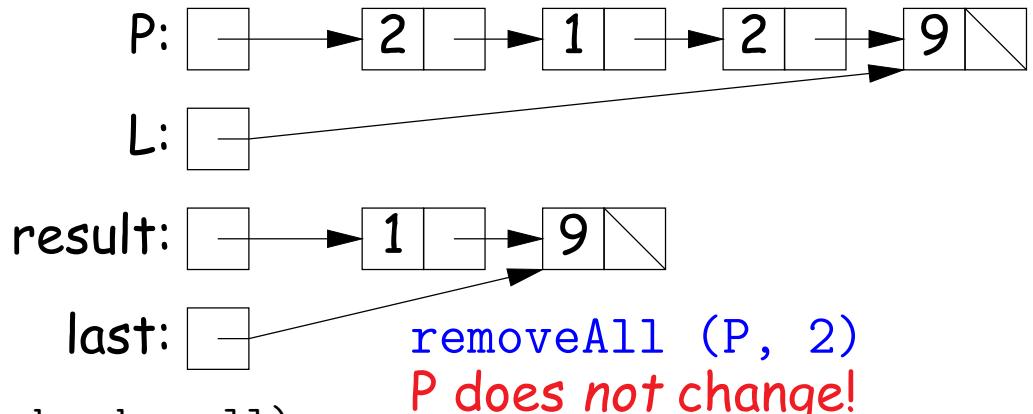
Here,  $\mathcal{I}$  is the *loop invariant*:

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  $\mathcal{I}$  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*:

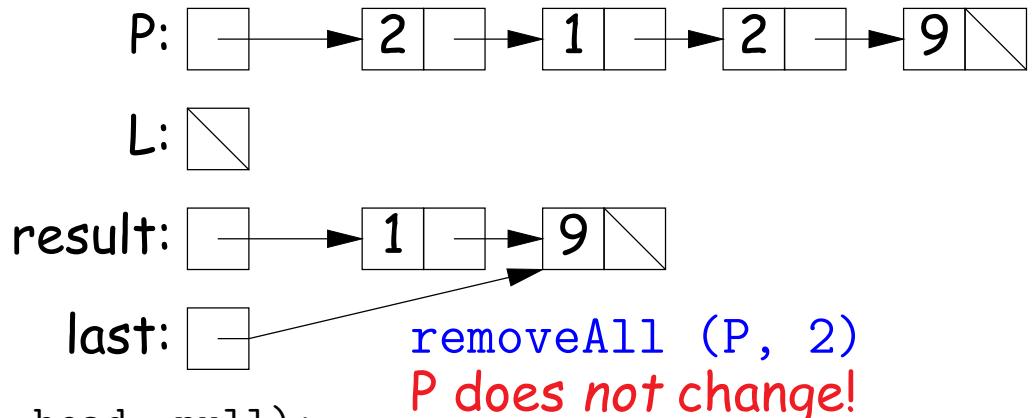
Result is all elements of  $L_0$  not equal to  $x$  up to and not including  $L$ , and  $last$  points to the last element of result, if any. We use  $L_0$  here to mean "the original sequence of int values in  $L$ ."

# 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. */

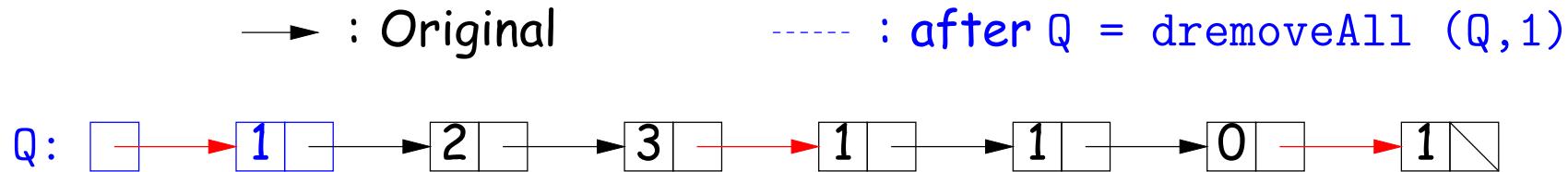
static IntList removeAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    for ( ; L != null; L = L.tail) {
        /* L != null and  $\mathcal{I}$  is true. */
        if (x == L.head)
            continue;
        else if (last == null)
            result = last = new IntList(L.head,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



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

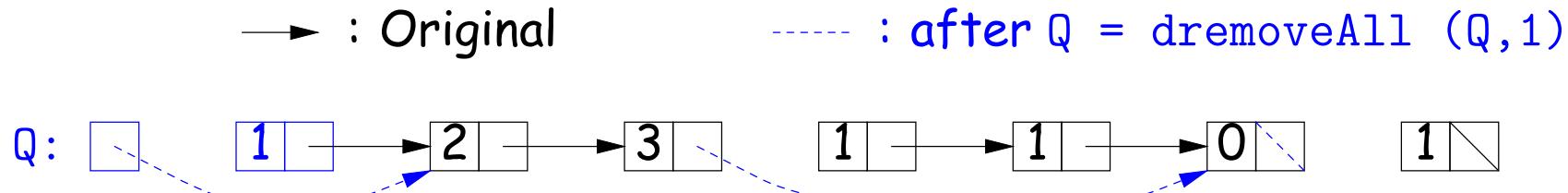
Result is all elements of  $L_0$  not equal to  $x$  up to and not including  $L$ , and  $last$  points to the last element of  $result$ , if any. We use  $L_0$  here to mean "the original sequence of int values in  $L$ ."

# Destructive Deletion



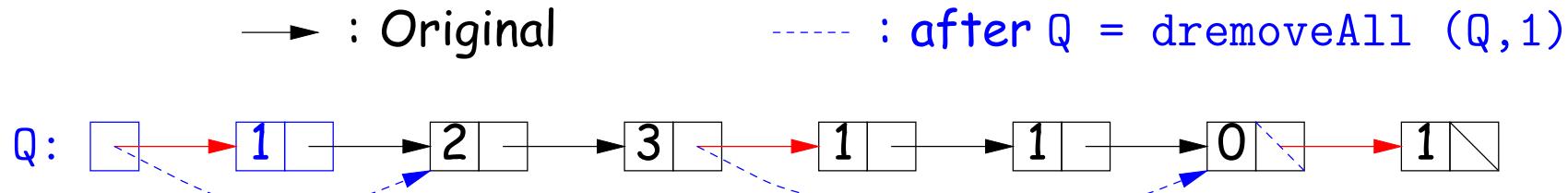
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll(IntList L, int x) {  
    if (L == null)  
        return /*( null with all x's removed );*/;  
    else if (L.head == x)  
        return /*( L with all x's removed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

# Destructive Deletion



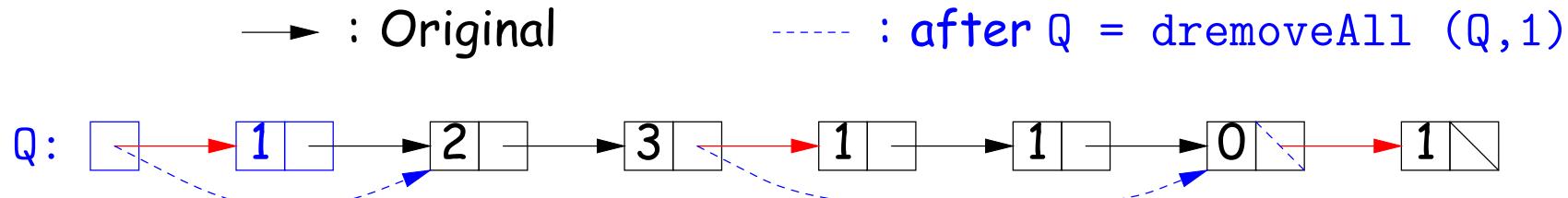
```
/** The list resulting from removing all instances of X from L.
 * The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
    if (L == null)
        return /*( null with all x's removed );*/
    else if (L.head == x)
        return /*( L with all x's removed (L != null)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

# Destructive Deletion



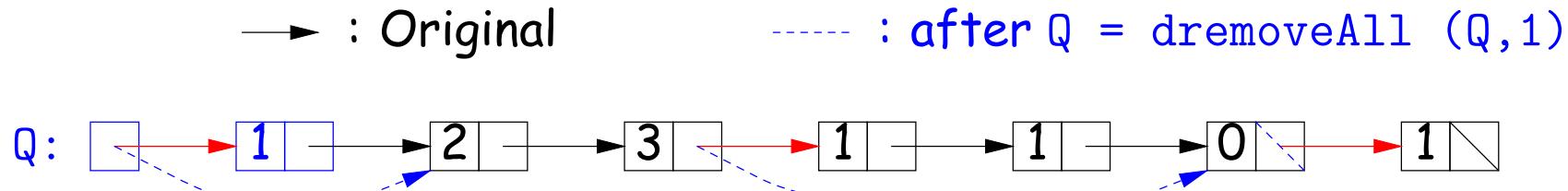
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll(IntList L, int x) {  
    if (L == null)  
        return /*( null with all x's removed );*/;  
    else if (L.head == x)  
        return /*( L with all x's removed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

# Destructive Deletion



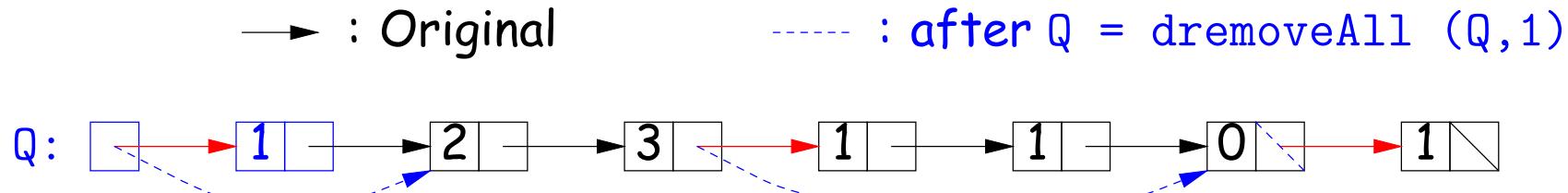
```
/** The list resulting from removing all instances of X from L.
 * The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
    if (L == null)
        return /*( null with all x's removed );*/
    else if (L.head == x)
        return /*( L with all x's removed (L != null)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

# Destructive Deletion



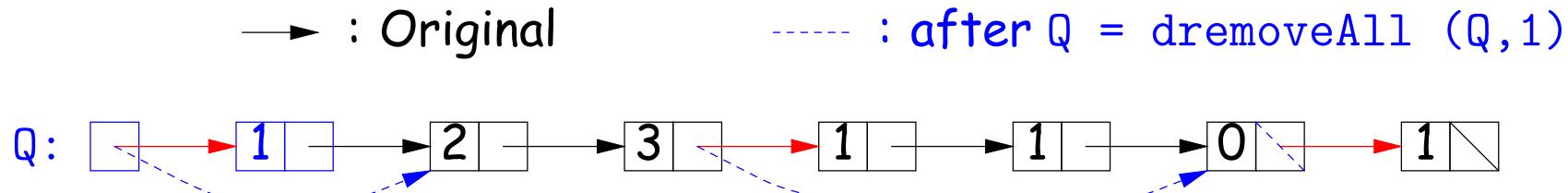
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll(IntList L, int x) {  
    if (L == null)  
        return null;  
    else if (L.head == x)  
        return /*( L with all x's removed (L != null) );*/;  
    else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



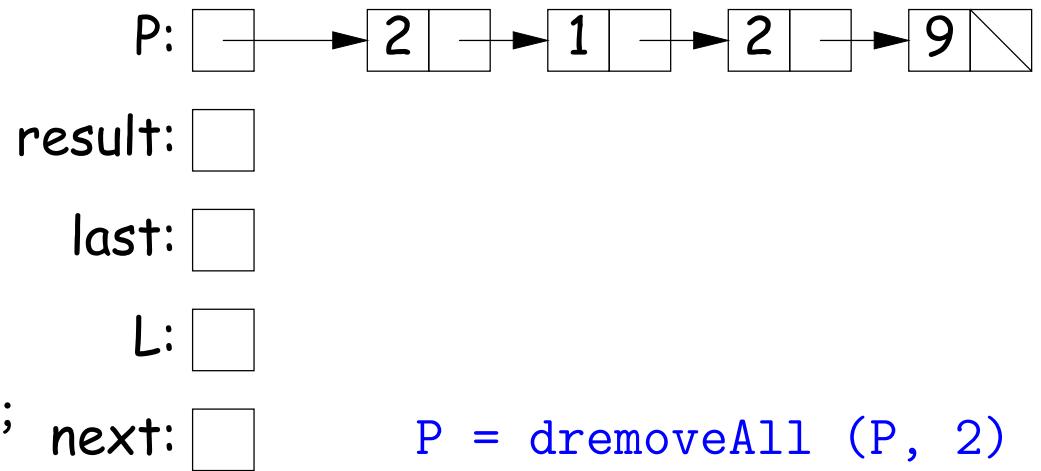
```
/** The list resulting from removing all instances of X from L.  
 * The original list may be destroyed. */  
static IntList dremoveAll(IntList L, int x) {  
    if (L == null)  
        return null;  
    else if (L.head == x)  
        return dremoveAll(L.tail, x);  
    else {  
        L.tail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

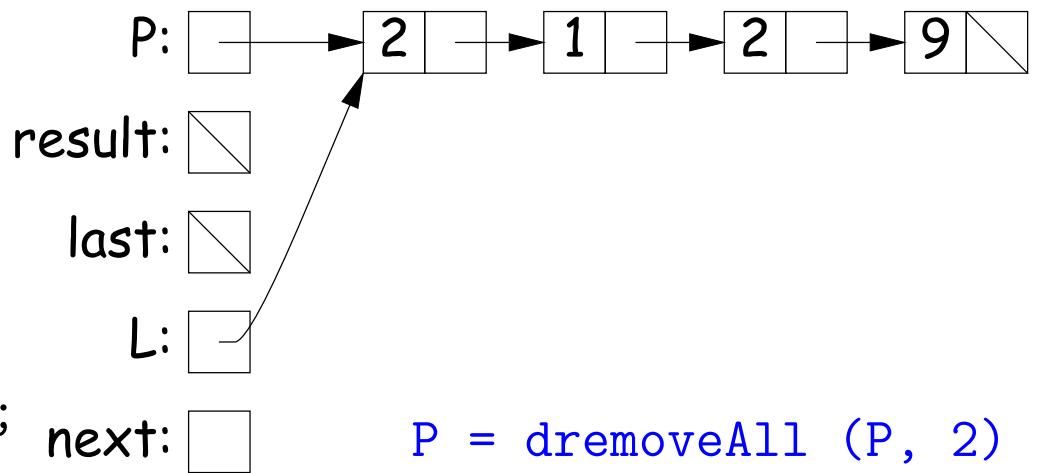
# Iterative Destructive Deletion

```
/** 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;  next:   
            L.tail = null;  
        }  
        L = next;  
    }  
    return result;  
}
```



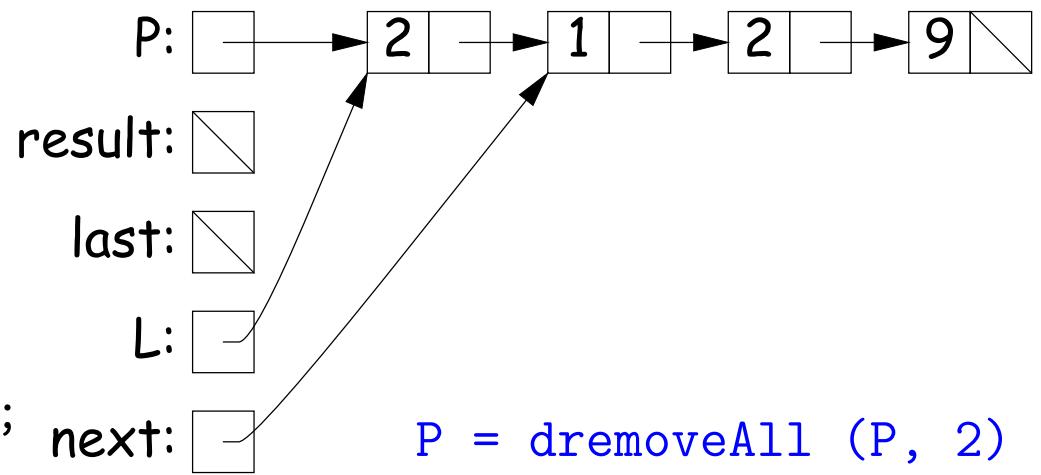
# Iterative Destructive Deletion

```
/** 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;  next:   
            L.tail = null;  
        }  
        L = next;  
    }  
    return result;  
}
```



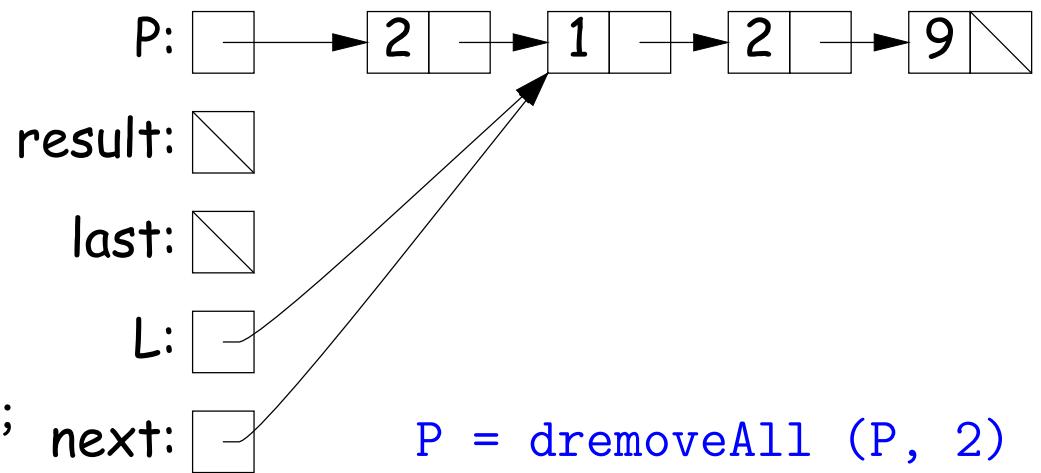
# Iterative Destructive Deletion

```
/** 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;  
}
```



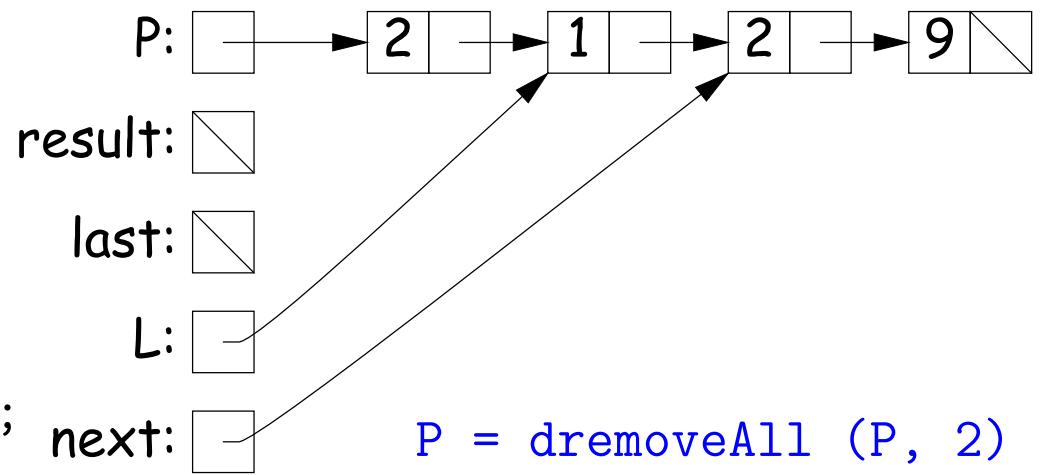
# Iterative Destructive Deletion

```
/** 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;  
}
```



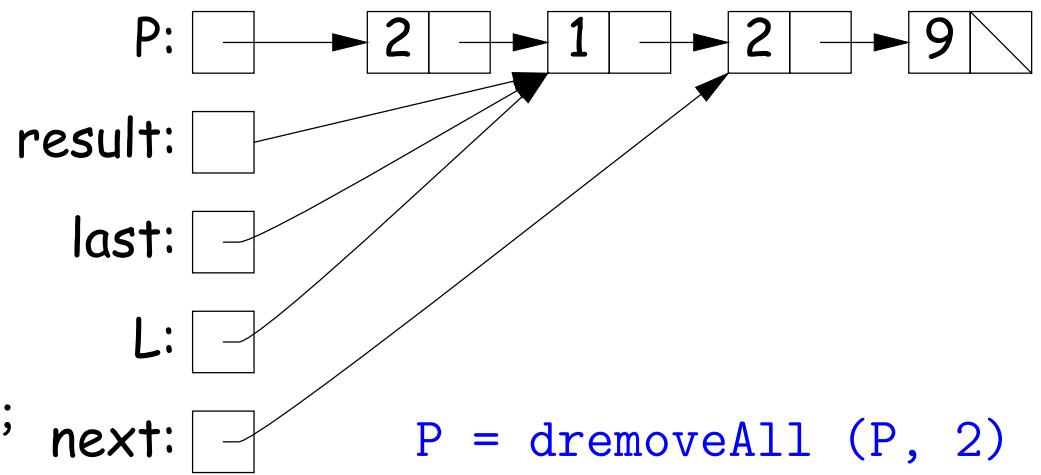
# Iterative Destructive Deletion

```
/** 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;  
}
```



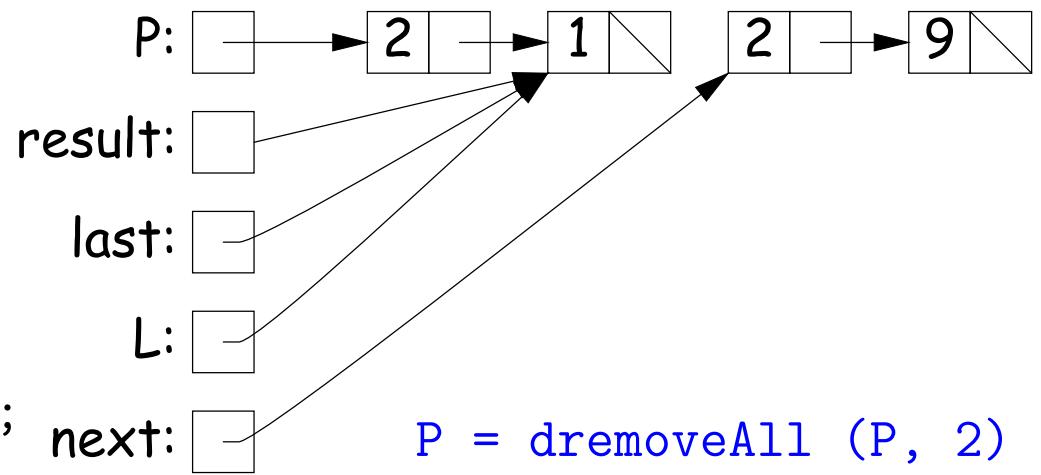
# Iterative Destructive Deletion

```
/** 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;  
}
```



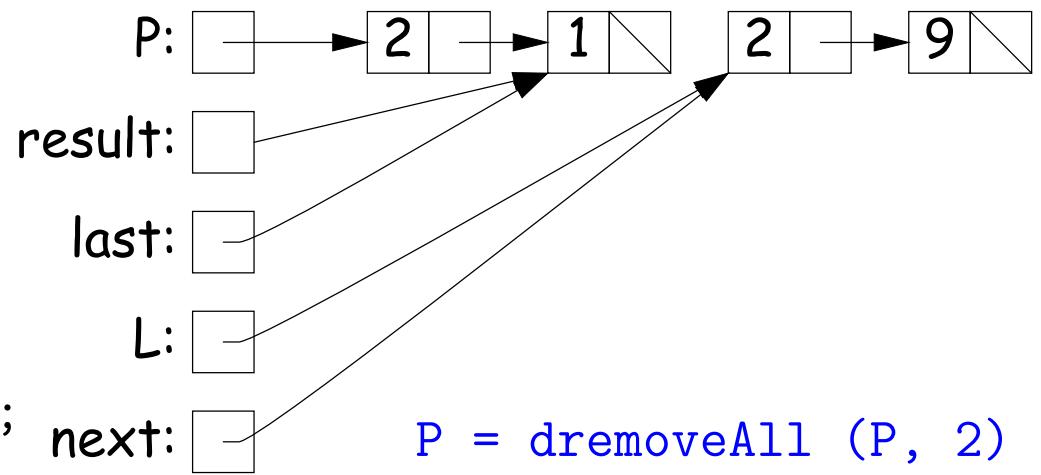
# Iterative Destructive Deletion

```
/** 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;  
}
```



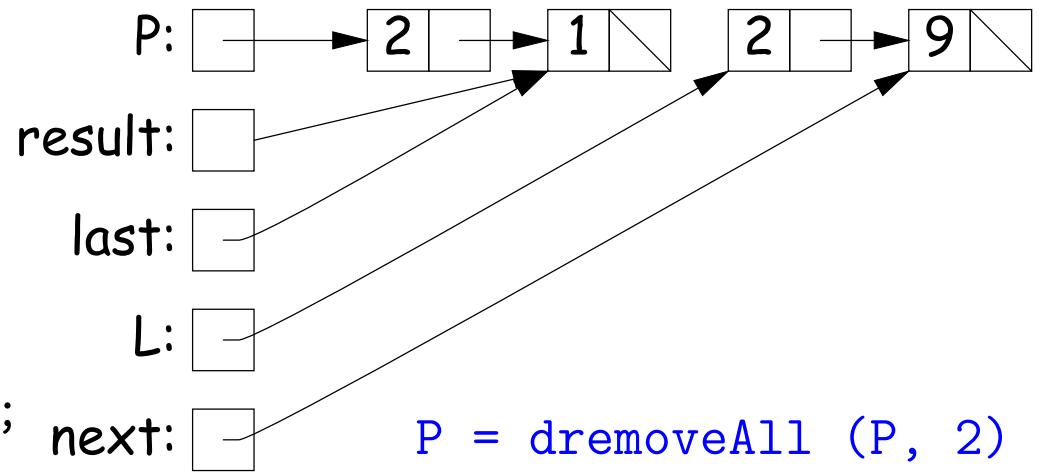
# Iterative Destructive Deletion

```
/** 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;  
}
```



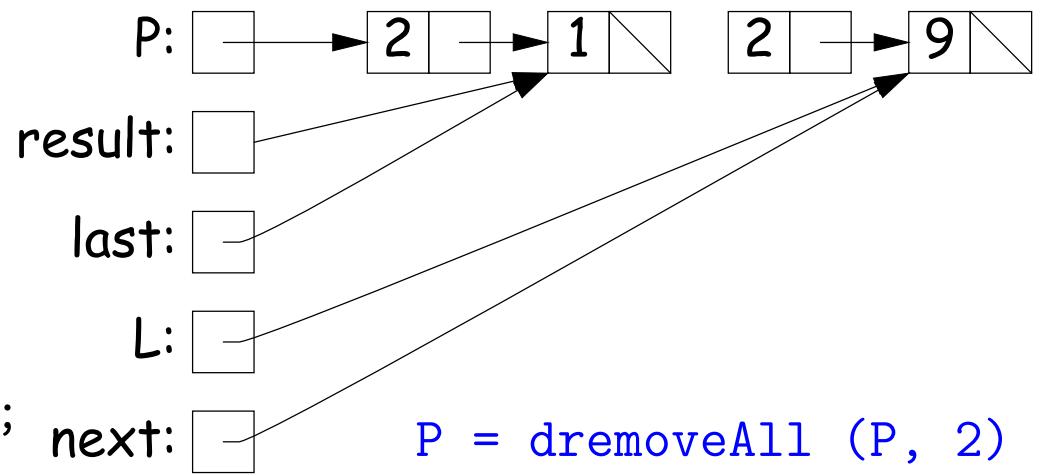
# Iterative Destructive Deletion

```
/** 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;  
}
```



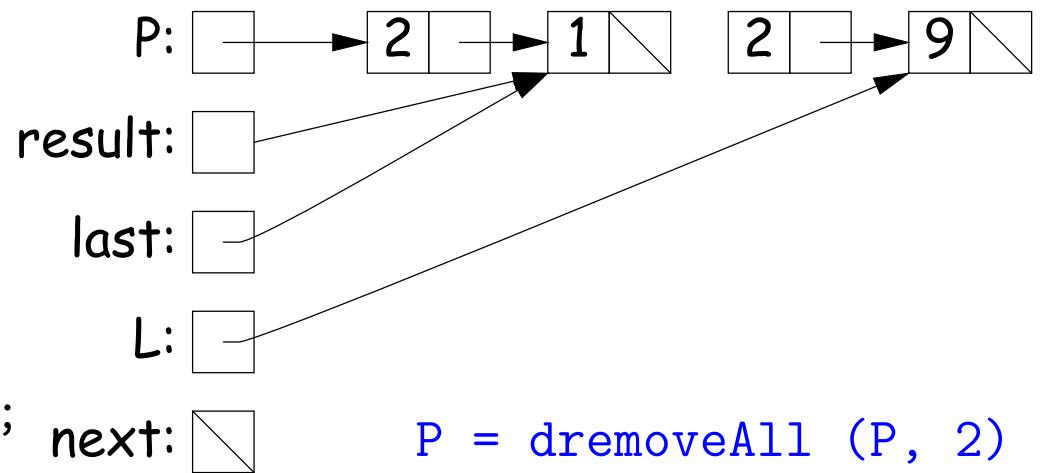
# Iterative Destructive Deletion

```
/** 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;  
}
```



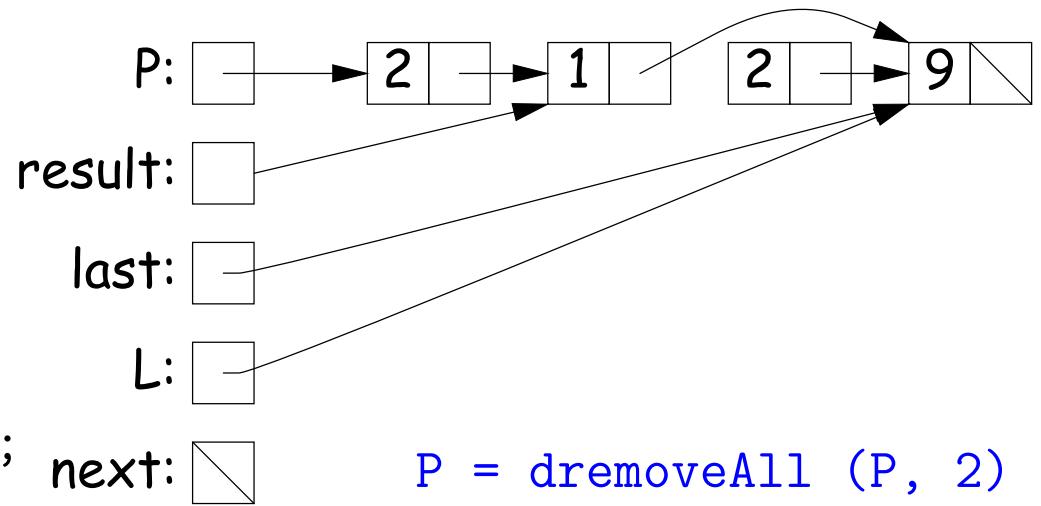
# Iterative Destructive Deletion

```
/** 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;  next:   
            L.tail = null;  
        }  
        L = next;  
    }  
    return result;  
}
```



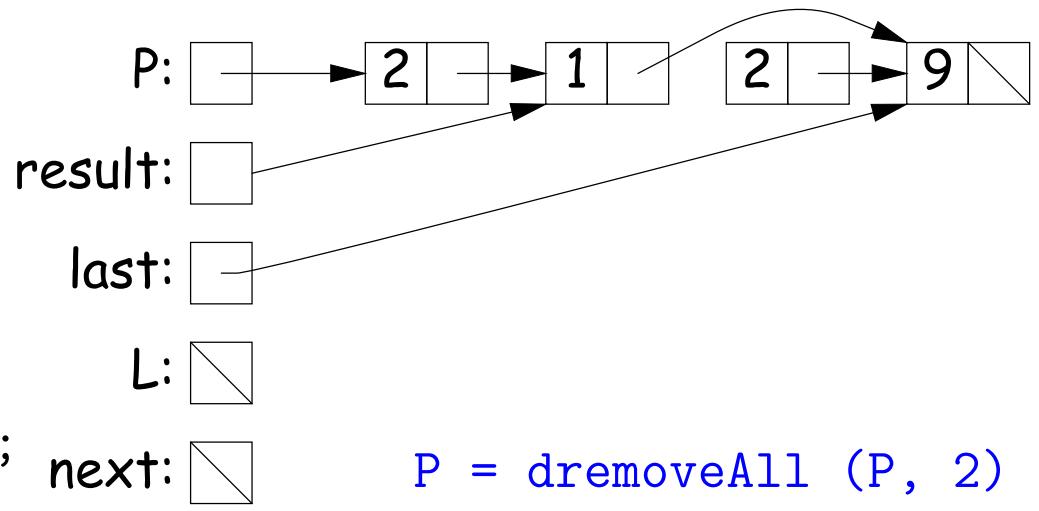
# Iterative Destructive Deletion

```
/** 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;  next: X  
            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;  next:   
            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;  next:   
            L.tail = null;  
        }  
        L = next;  
    }  
    return result;  
}
```

