

CS61B Lecture #4: Simple Pointer Manipulation

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

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

Announcements

- **Today:** More pointer hacking.
- **Handing in labs and homework:** We'll be lenient about accepting late homework and labs for lab1, lab2, and hw0. Just get it done: part of the point is getting to understand the tools involved. We will **not** accept submissions by email.
- We will feel free to interpret the absence of a central repository for you or a lack of a lab1 submission from you as indicating that you intend to drop the course.
- HW1 to be released tonight (roughly).
- Project 0 to be released Friday.

Small Test of Understanding

- In Java, the keyword **final** in a variable declaration means that the variable's value may not be changed after the variable is initialized.
- Is the following class valid?

```
public class Issue {  
  
    private final IntList aList = new IntList(0, null);  
  
    public void modify(int k) {  
        this.aList.head = k;  
    }  
}
```

Why or why not?

Small Test of Understanding

- In Java, the keyword **final** in a variable declaration means that the variable's value may not be changed after the variable is initialized.
- Is the following class valid?

```
public class Issue {  
  
    private final IntList aList = new IntList(0, null);  
  
    public void modify(int k) {  
        this.aList.head = k;  
    }  
}
```

Why or why not?

Answer: This is *valid*. Although `modify` changes the head variable of the object pointed to by `aList`, it does *not* modify the contents of `aList` itself (which is a pointer).

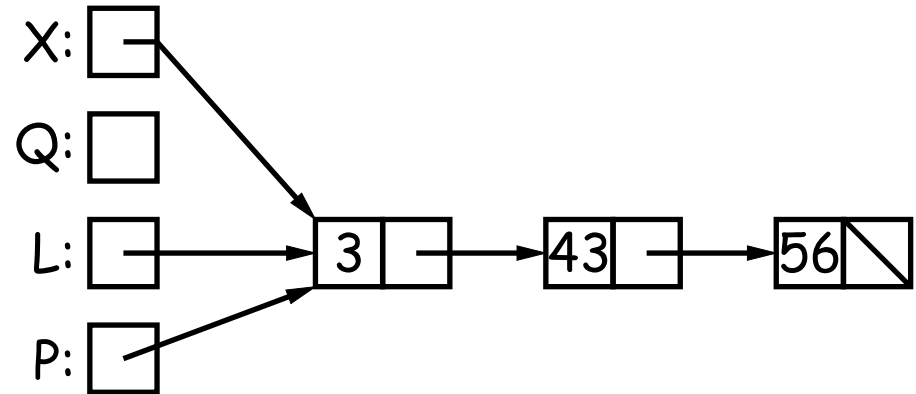
Destructive Incrementing

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

```
/** Destructively add N to P's items. */
static IntList dincrList(IntList P, int n) {
    if (P == null)
        return null;
    else {
        P.head += n;
        P.tail = dincrList(P.tail, n);
        return P;
    }
}
```

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

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



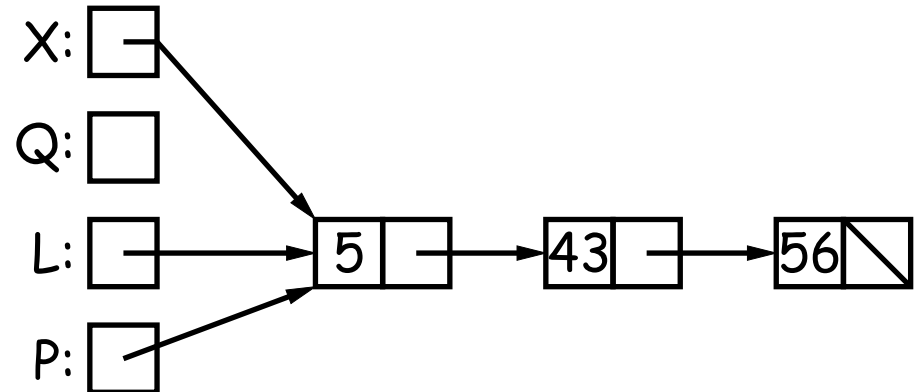
Destructive Incrementing

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

```
/** Destructively add N to P's items. */
static IntList dincrList(IntList P, int n) {
    if (P == null)
        return null;
    else {
        P.head += n;
        P.tail = dincrList(P.tail, n);
        return P;
    }
}
```

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

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



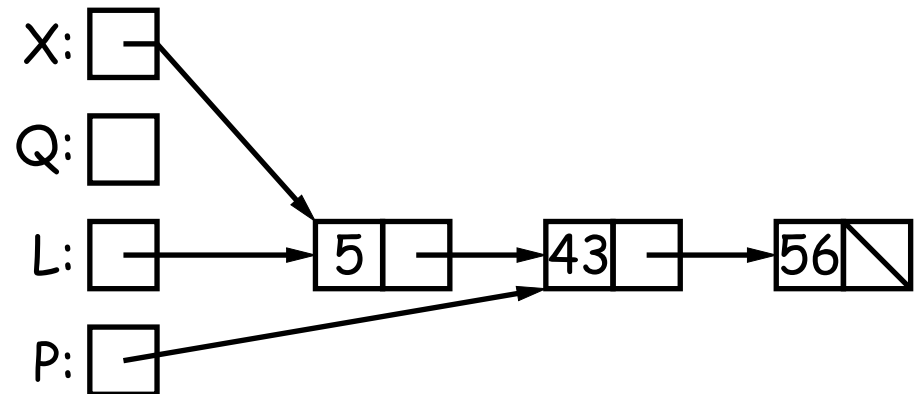
Destructive Incrementing

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

```
/** Destructively add N to P's items. */
static IntList dincrList(IntList P, int n) {
    if (P == null)
        return null;
    else {
        P.head += n;
        P.tail = dincrList(P.tail, n);
        return P;
    }
}
```

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

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



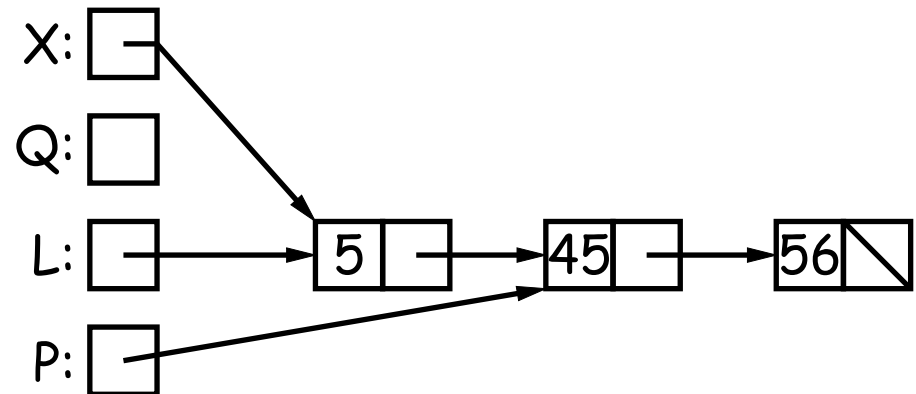
Destructive Incrementing

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

```
/** Destructively add N to P's items. */
static IntList dincrList(IntList P, int n) {
    if (P == null)
        return null;
    else {
        P.head += n;
        P.tail = dincrList(P.tail, n);
        return P;
    }
}
```

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

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



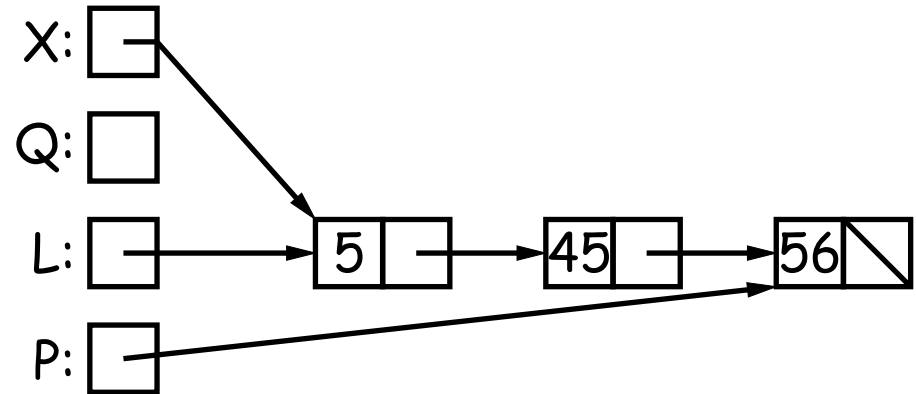
Destructive Incrementing

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

```
/** Destructively add N to P's items. */
static IntList dincrList(IntList P, int n) {
    if (P == null)
        return null;
    else {
        P.head += n;
        P.tail = dincrList(P.tail, n);
        return P;
    }
}

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

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



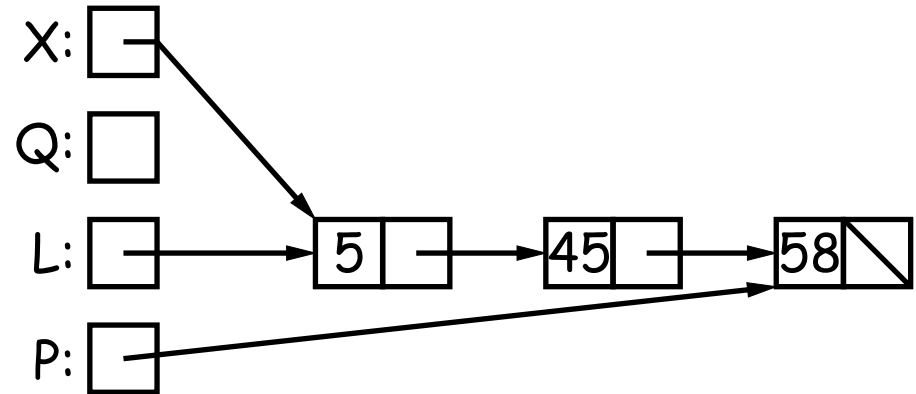
Destructive Incrementing

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

```
/** Destructively add N to P's items. */
static IntList dincrList(IntList P, int n) {
    if (P == null)
        return null;
    else {
        P.head += n;
        P.tail = dincrList(P.tail, n);
        return P;
    }
}
```

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

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



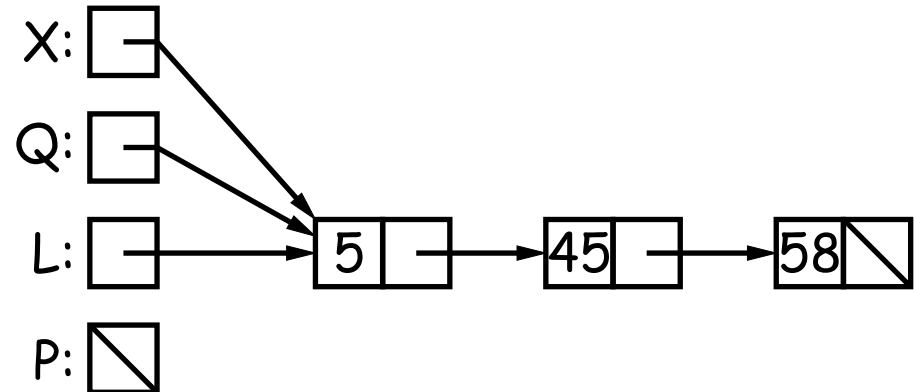
Destructive Incrementing

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

```
/** Destructively add N to P's items. */
static IntList dincrList(IntList P, int n) {
    if (P == null)
        return null;
    else {
        P.head += n;
        P.tail = dincrList(P.tail, n);
        return P;
    }
}

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

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



Another 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, L.head==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 /*( L with all x's removed (L!=null, L.head==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 /*( 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) {
        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;
}
```

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) {
```

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

```
            continue;
```

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

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

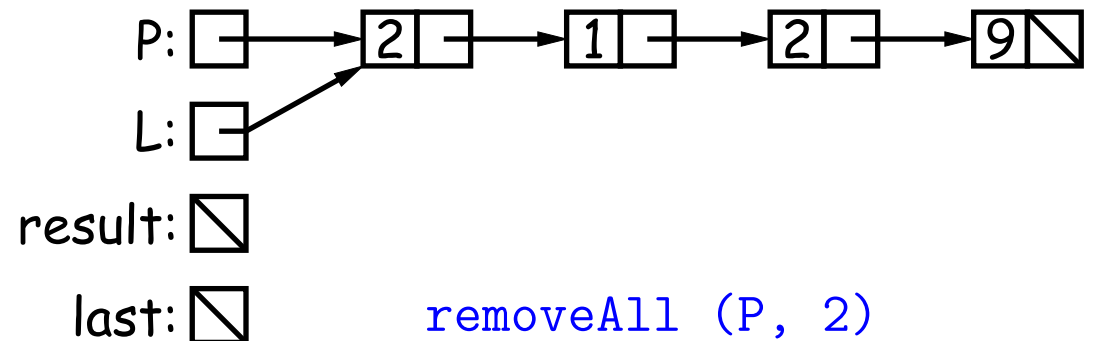
```
        else
```

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

```
    }
```

```
    return result;
```

```
}
```



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) {
```

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

```
            continue;
```

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

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

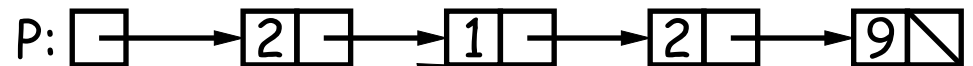
```
        else
```

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

```
    }
```

```
    return result;
```

```
}
```



removeAll (P, 2)

P does not change!

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) {
```

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

```
            continue;
```

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

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

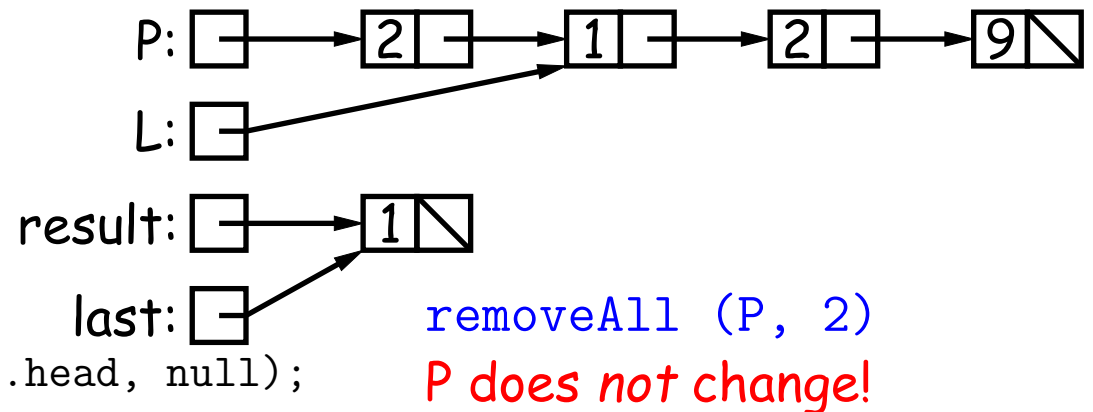
```
        else
```

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

```
    }
```

```
    return result;
```

```
}
```



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) {
```

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

```
            continue;
```

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

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

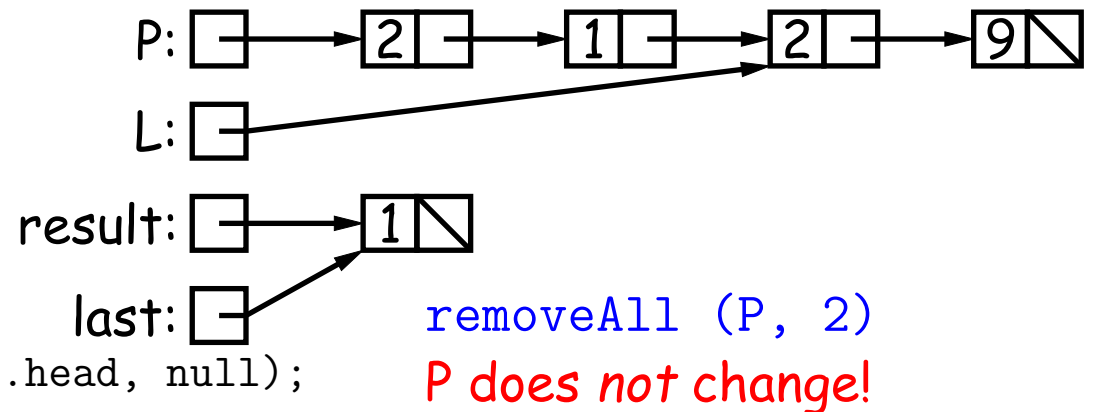
```
        else
```

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

```
    }
```

```
    return result;
```

```
}
```



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) {
```

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

```
            continue;
```

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

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

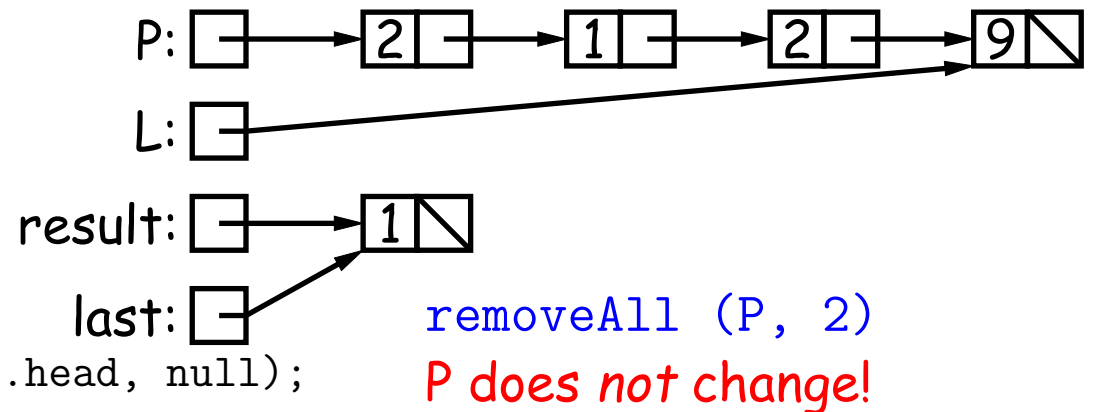
```
        else
```

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

```
    }
```

```
    return result;
```

```
}
```



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) {
```

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

```
            continue;
```

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

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

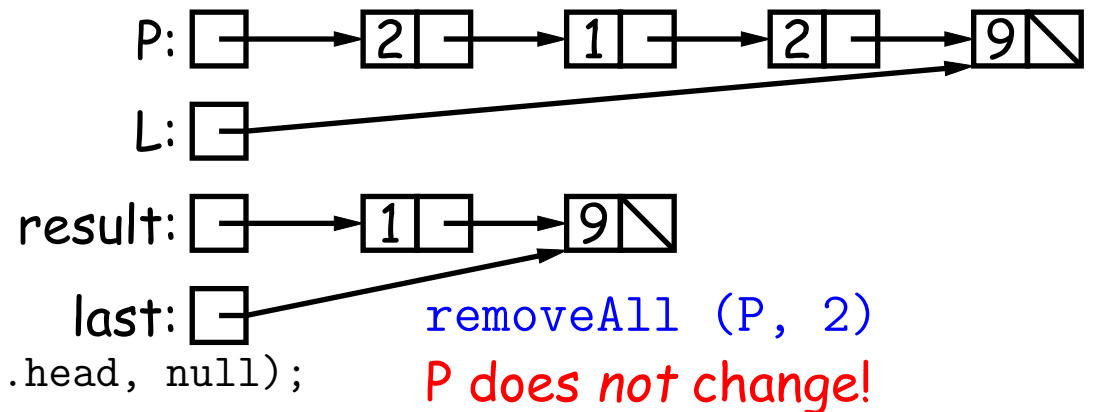
```
        else
```

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

```
    }
```

```
    return result;
```

```
}
```



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) {
```

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

```
            continue;
```

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

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

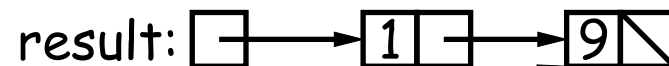
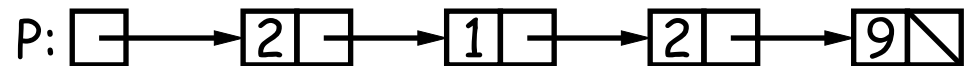
```
        else
```

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

```
    }
```

```
    return result;
```

```
}
```



removeAll (P, 2)

P does not change!

Destructive Deletion

→ : Original

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

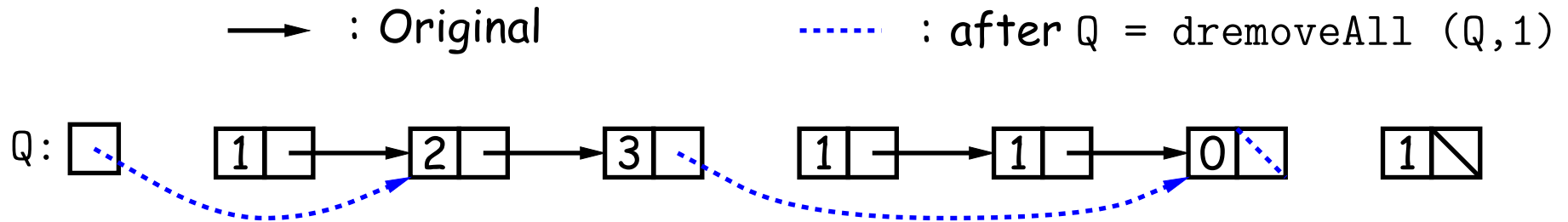


/** The list resulting from removing all instances of X from L.

* The original list may be destroyed. */

```
static IntList dremoveAll(IntList L, int x) {
    if (L == null)
        return /*( null with all x's removed )*/;
    else if (L.head == x)
        return /*( L with all x's removed (L != null) )*/;
    else {
        /*{ Remove all x's from L's tail. }*/;
        return L;
    }
}
```

Destructive Deletion



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

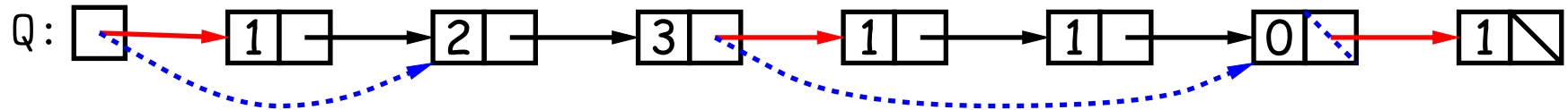
* The original list may be destroyed. */

```
static IntList dremoveAll(IntList L, int x) {
    if (L == null)
        return /*( null with all x's removed )*/;
    else if (L.head == x)
        return /*( L with all x's removed (L != null) )*/;
    else {
        /*{ Remove all x's from L's tail. }*/;
        return L;
    }
}
```


Destructive Deletion

→ : Original

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



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

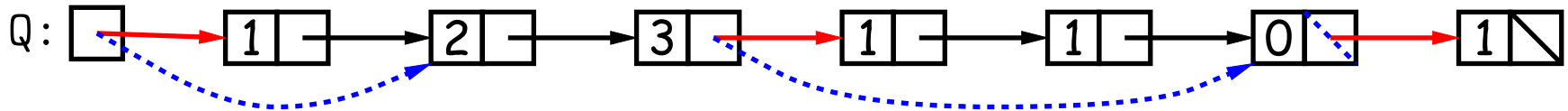
* The original list may be destroyed. */

```
static IntList dremoveAll(IntList L, int x) {
    if (L == null)
        return /*( null with all x's removed )*/;
    else if (L.head == x)
        return /*( L with all x's removed (L != null) )*/;
    else {
        /*{ Remove all x's from L's tail. }*/;
        return L;
    }
}
```

Destructive Deletion

→ : Original

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



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

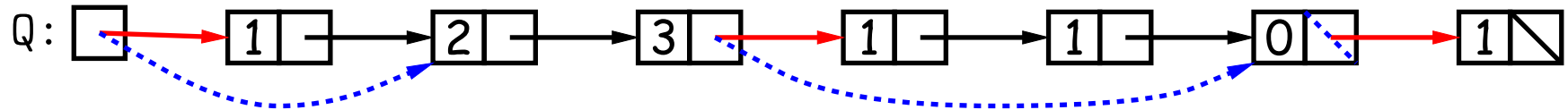
* The original list may be destroyed. */

```
static IntList dremoveAll(IntList L, int x) {
    if (L == null)
        return /*( null with all x's removed )*/;
    else if (L.head == x)
        return /*( L with all x's removed (L != null) )*/;
    else {
        /*{ Remove all x's from L's tail. }*/;
        return L;
    }
}
```

Destructive Deletion

→ : Original

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



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

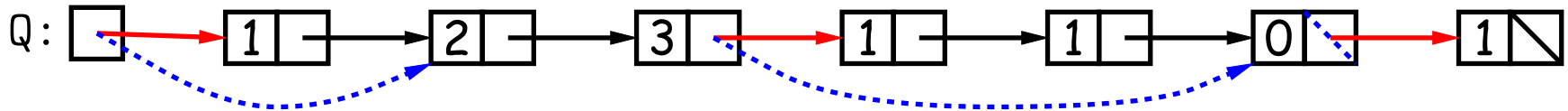
* The original list may be destroyed. */

```
static IntList dremoveAll(IntList L, int x) {
    if (L == null)
        return null;
    else if (L.head == x)
        return /*( L with all x's removed (L != null) )*/;
    else {
        /*{ Remove all x's from L's tail. }*/;
        return L;
    }
}
```

Destructive Deletion

→ : Original

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



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

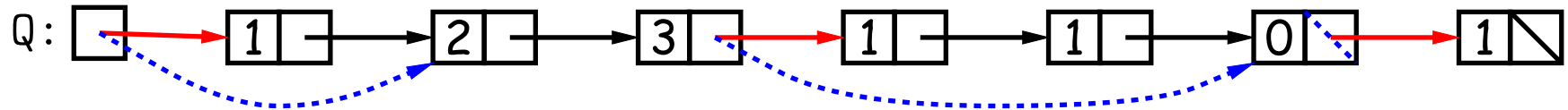
* The original list may be destroyed. */

```
static IntList dremoveAll(IntList L, int x) {
    if (L == null)
        return
    else if (L.head == x)
        return dremoveAll(L.tail, x);
    else {
        /*{ Remove all x's from L's tail. }*/;
        return L;
    }
}
```

Destructive Deletion

→ : Original

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



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

* The original list may be destroyed. */

```
static IntList dremoveAll(IntList L, int x) {
    if (L == null)
        return
    else if (L.head == x)
        return dremoveAll(L.tail, x);
    else {
        L.tail = dremoveAll(L.tail, x);
        return L;
    }
}
```

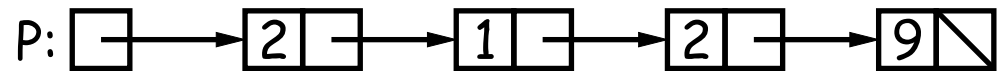
Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
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 X's from L
 * destructively. */
```

```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```



result:

last:

L:

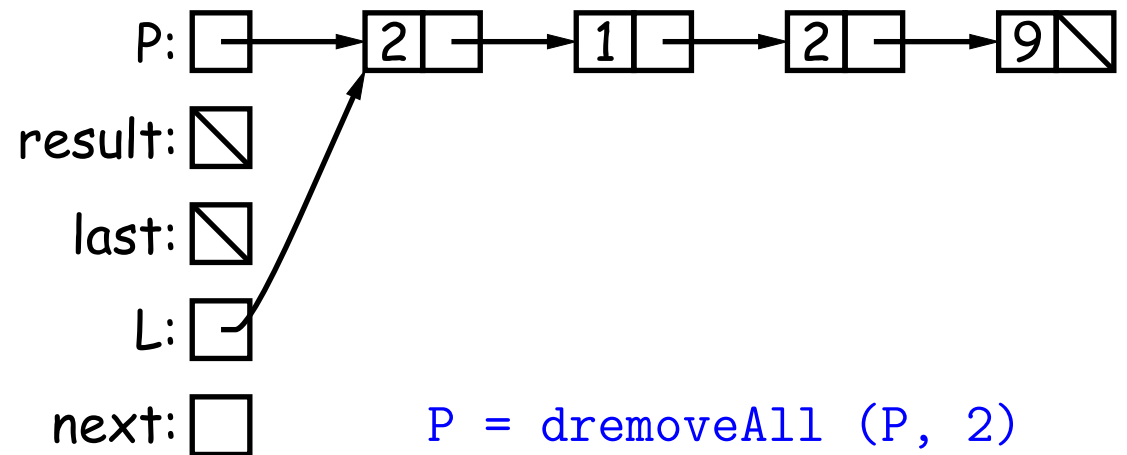
next:

P = dremoveAll (P, 2)

Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```

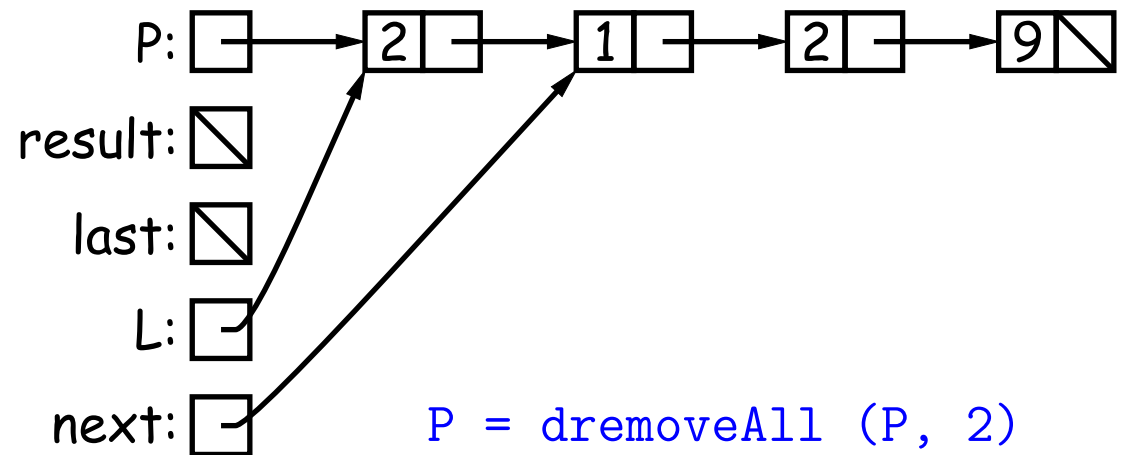


`P = dremoveAll (P, 2)`

Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```

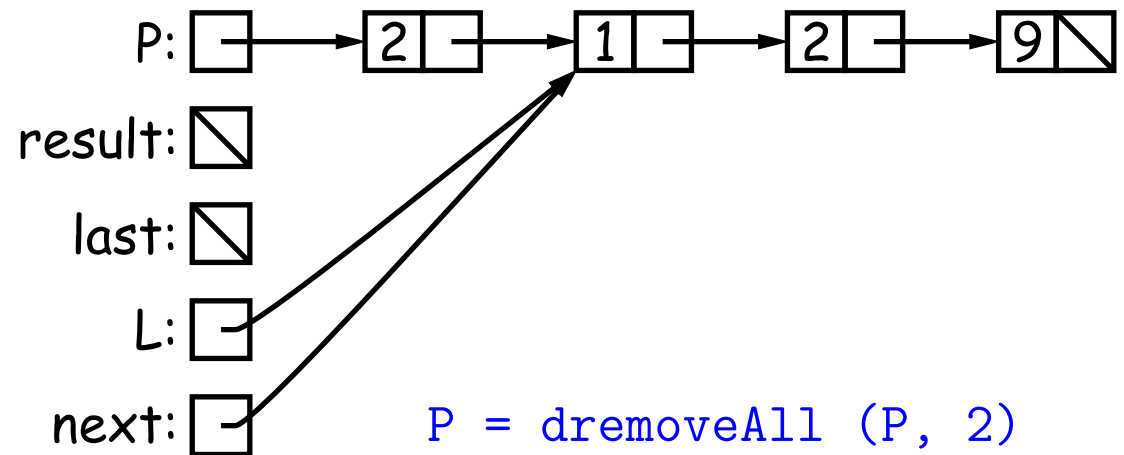


P = dremoveAll (P, 2)

Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```

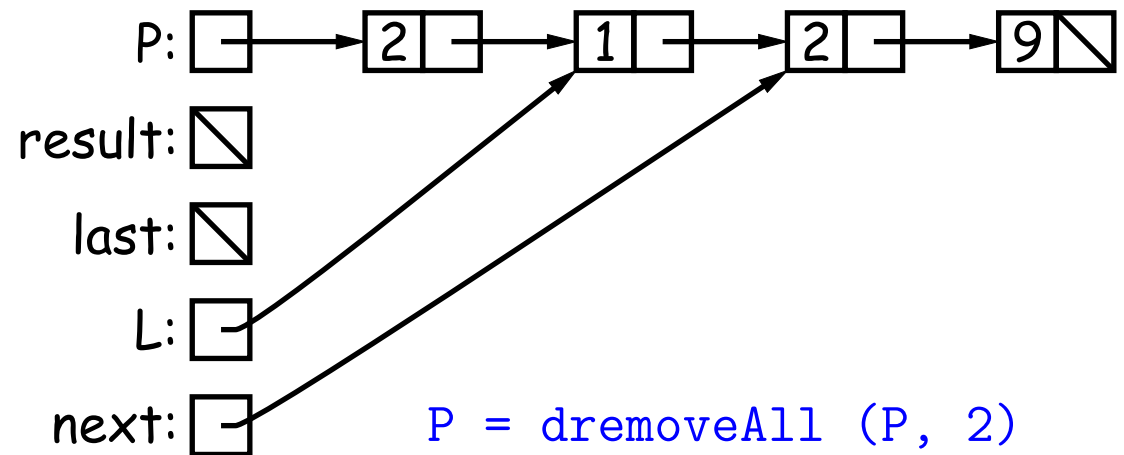


P = dremoveAll (P, 2)

Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```

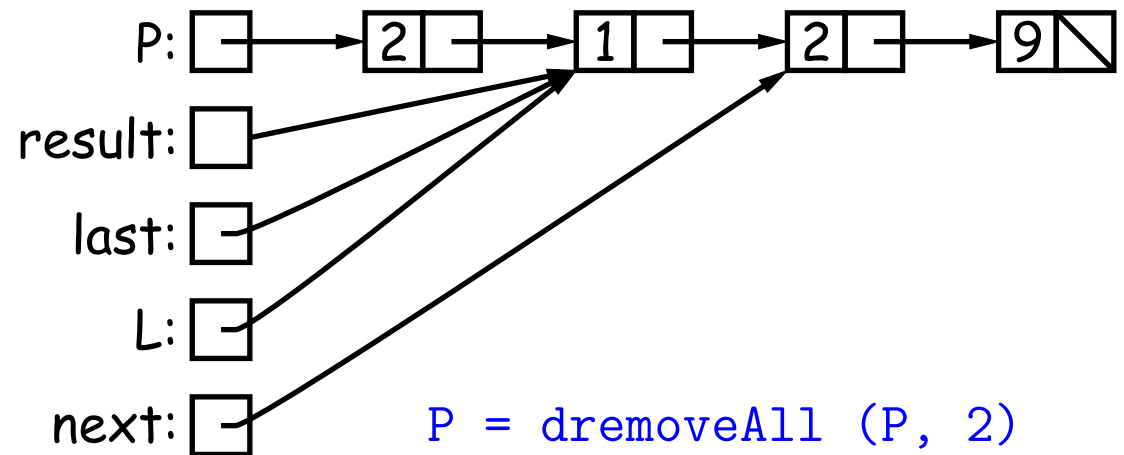


`P = dremoveAll (P, 2)`

Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

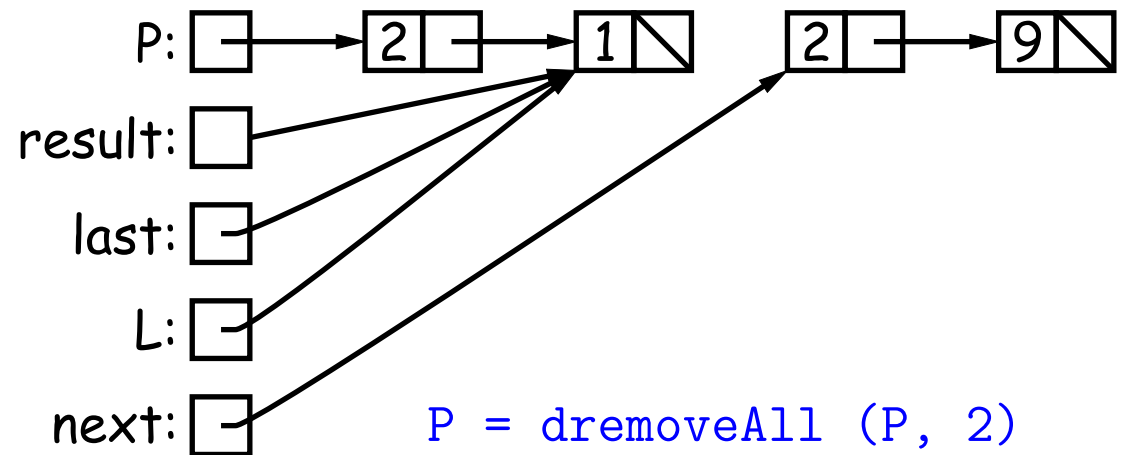
```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```



Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

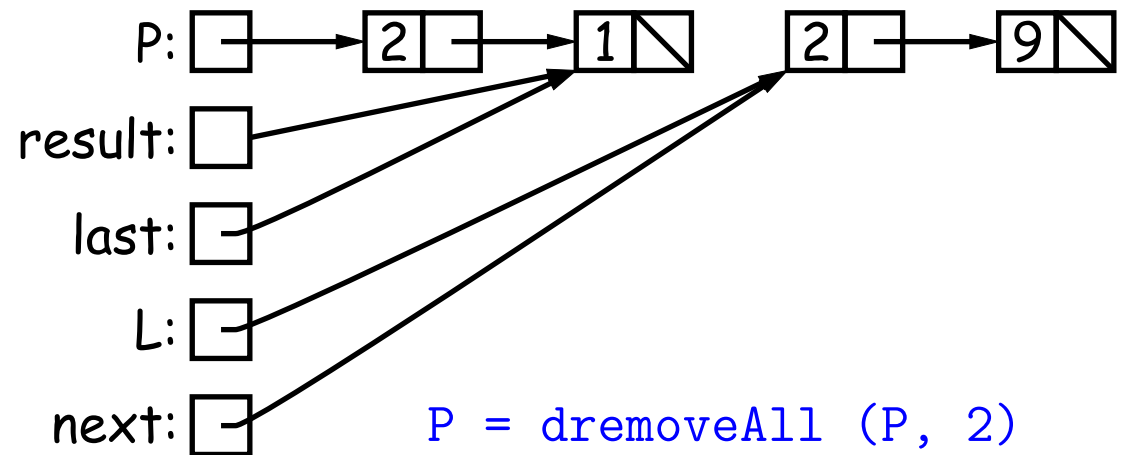
```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```



Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

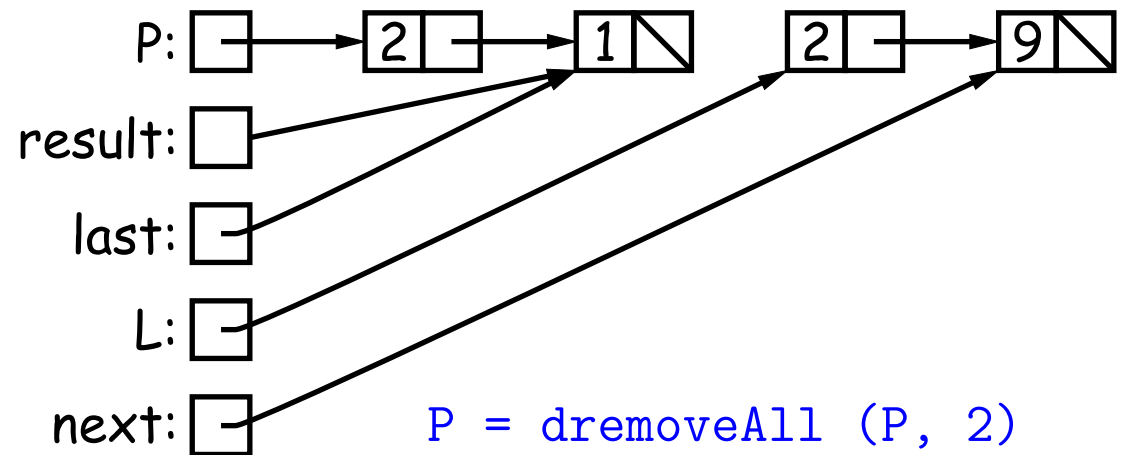
```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```



Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```

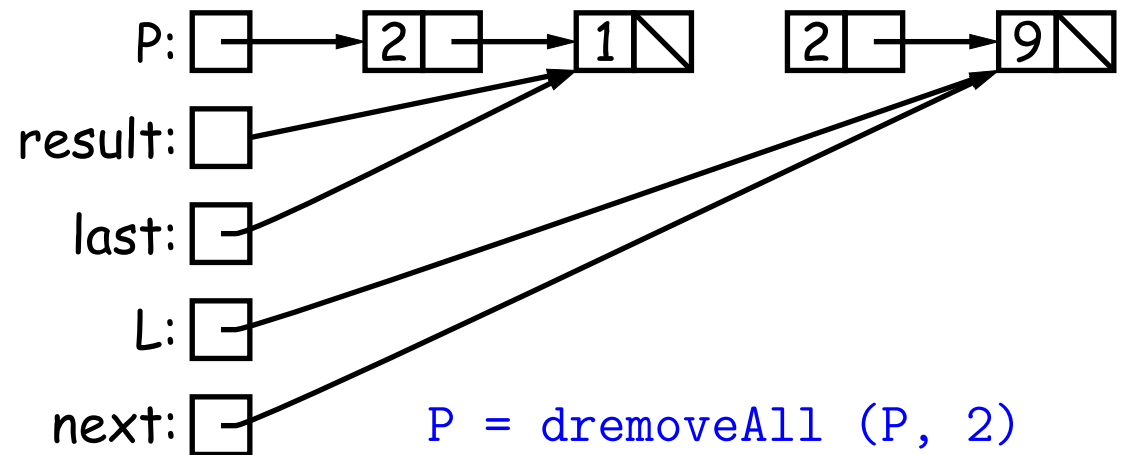


`P = dremoveAll (P, 2)`

Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```

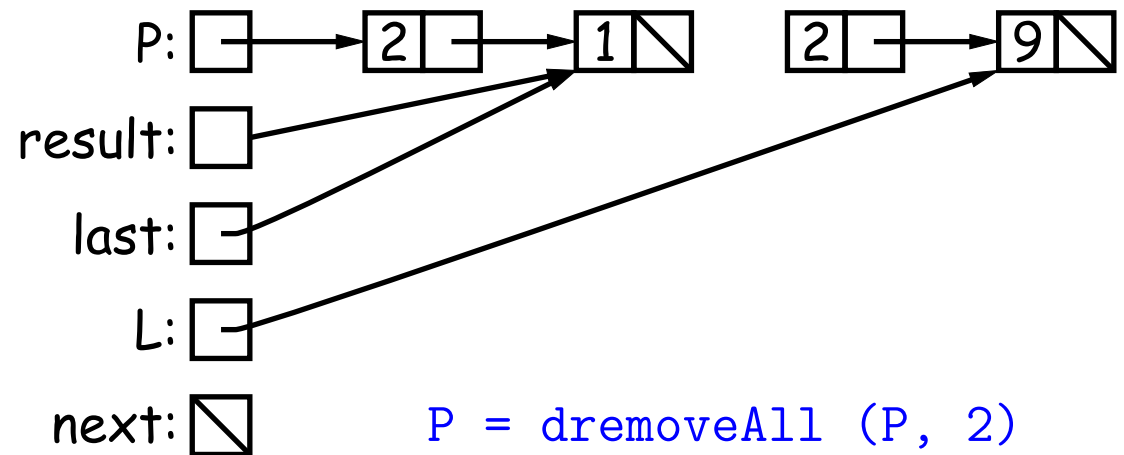


P = dremoveAll (P, 2)

Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```

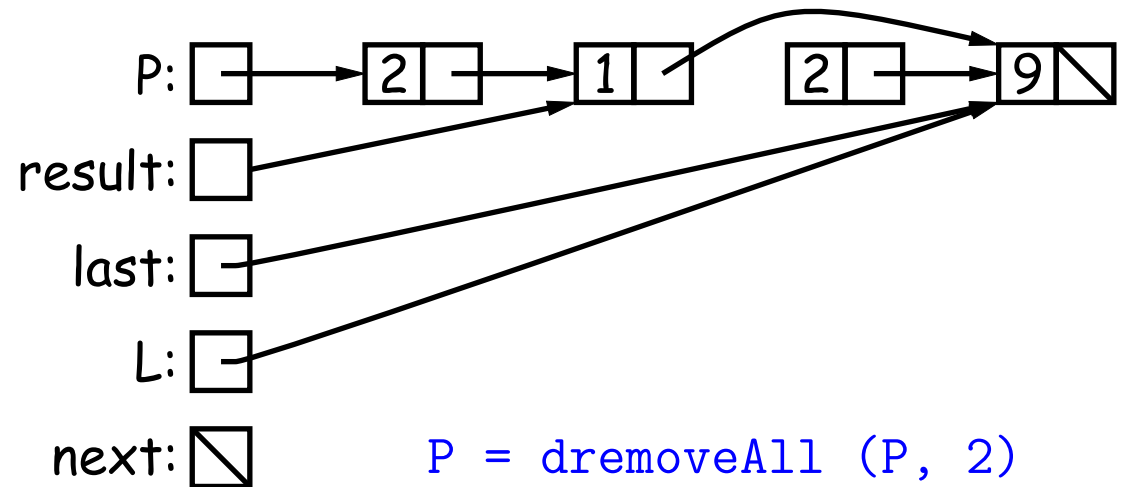


`P = dremoveAll (P, 2)`

Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```

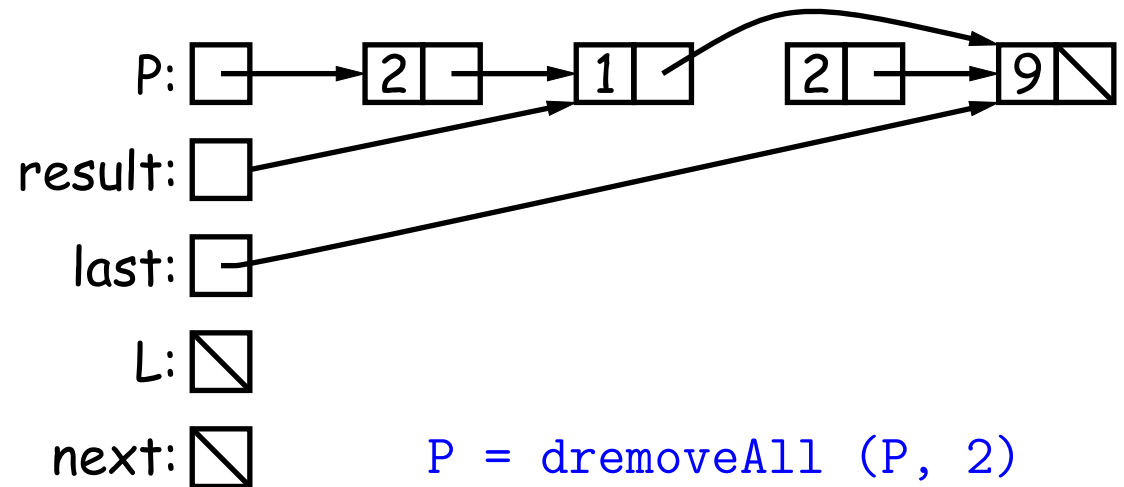


$P = \text{dremoveAll}(P, 2)$

Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```

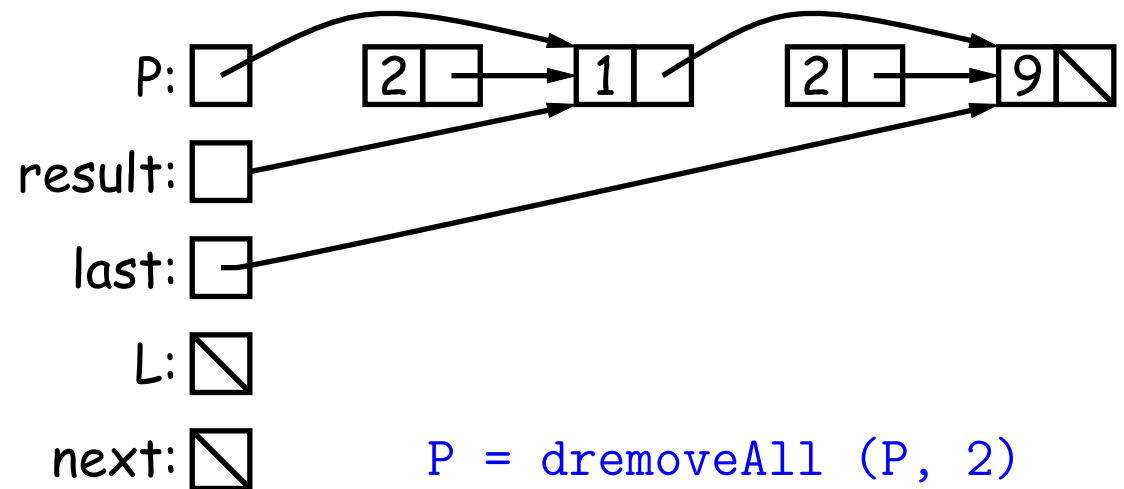


`P = dremoveAll (P, 2)`

Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 * destructively. */
```

```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```



`P = dremoveAll (P, 2)`

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 always true at the start of each iteration.
- The loop body then must
 - Start from any situation consistent with the invariant and condition;
 - Make progress in such a way as to make the invariant true again.

```
// Invariant must be true here
while (condition) { // condition must not have side-effects.
    // (Invariant and condition are necessarily true here.)
    loop body
    // Invariant must again be true here
}
```

// *Invariant* true and *condition* false.

- So if our loop gets the desired answer whenever *Invariant* is true and *condition* false, our job is done!

Relationship to Recursion

- Another way to see this is to consider an equivalent recursive procedure:

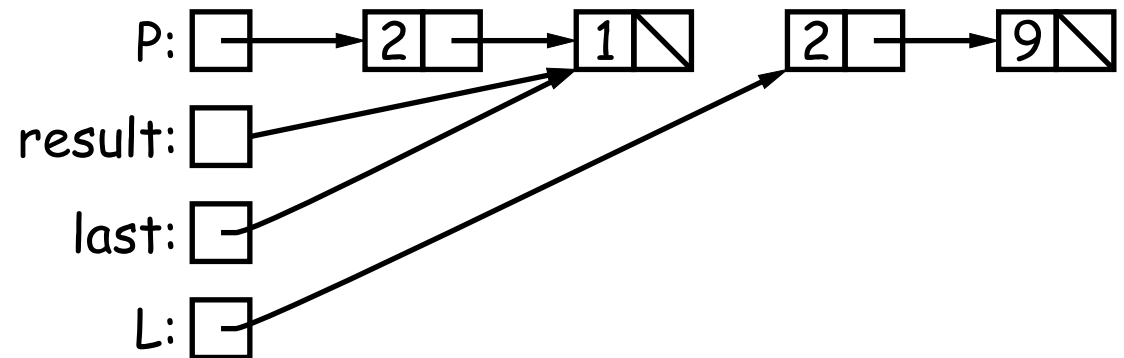
```
/** Assuming Invariant, produce a situation where Invariant
 * is true and condition is false. */
void loop() {
    // Invariant assumed true here.
    if (condition) {
        // Invariant and condition true here.
        loop body
        // Invariant must be true here.
        loop()
        // Invariant true here and condition false.
    } else { /* condition false here. */ }
}
```

- Here, the invariant is the precondition of the function **loop**.
- The loop maintains the invariant while making the condition false.
- Idea is to arrange that our actual goal is implied by this post-condition.

Example: Loop Invariant for dremoveAll

```
/** The list resulting from removing all X's from L
 * destructively. */
```

```
static IntList dremoveAll(IntList L, int x) {
    IntList result, last;
    result = last = null;
    while ** (L != null) {
        IntList next = L.tail;
        if (x != L.head) {
            if (last == null)
                result = last = L;
            else
                last = last.tail = L;
            L.tail = null;
        }
        L = next;
    }
    return result;
}
```



$P = \text{dremoveAll}(P, 2)$

**** Invariant:**

- result points to the list of items in the final result except for those from L onward.
- L points to an unchanged tail of the original list of items in L.
- last points to the last item in result or is null if result is null.