CS61B Lecture #4: Simple Pointer Manipulati

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 ac late homework and labs for lab1, lab2, and hw0. Just get part of the point is getting to understand the tools involved. *not* accept submissions by email.
- We will feel free to interpret the absence of a central rep for you or a lack of a lab1 submission from you as indicating t intend to drop the course.
- Project 0 to be released tonight.
- HW1 is released.

Small Test of Understanding

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

```
public class Issue {

    private final IntList aList = new IntList(0, nu

    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 t variable's value may not be changed after the variable is init
- Is the following class valid?

```
public class Issue {

    private final IntList aList = new IntList(0, nu

    public void modify(int k) {
        this.aList.head = k;
    }
}
```

Why or why not?

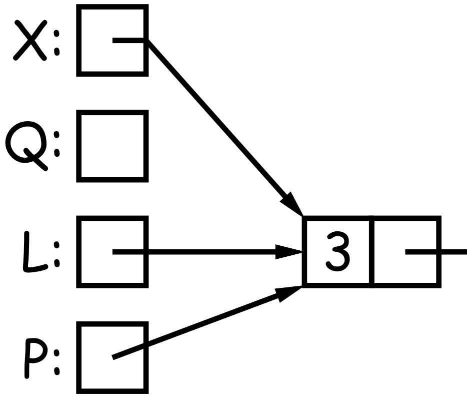
Answer: This is *valid*. Although `modify` changes the `head` of the object pointed to by `aList`, it does *not* modify the c of `aList` itself (which is a pointer).

Destructive Incrementing

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

```
/** Destructively add N to L'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.li
/* IntList.lis
Q = dincrList(
```

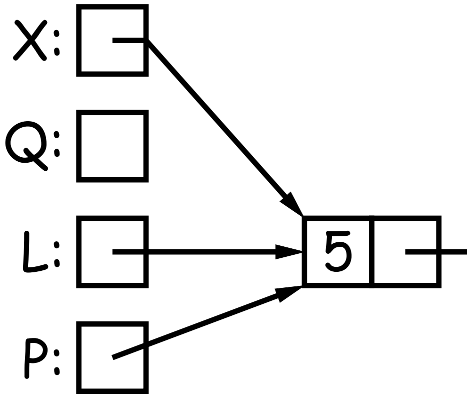


Destructive Incrementing

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

```
/** Destructively add N to L’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.li
/* IntList.lis
Q = dincrList(
```

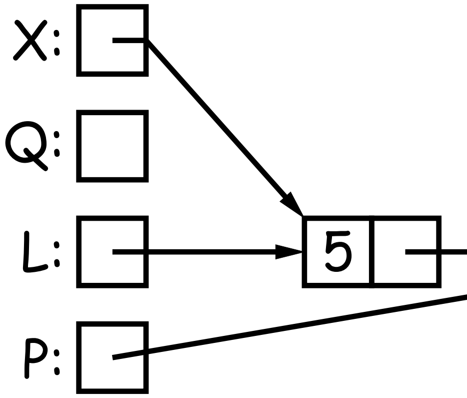


Destructive Incrementing

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

```
/** Destructively add N to L'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.li
/* IntList.lis
Q = dincrList(
```

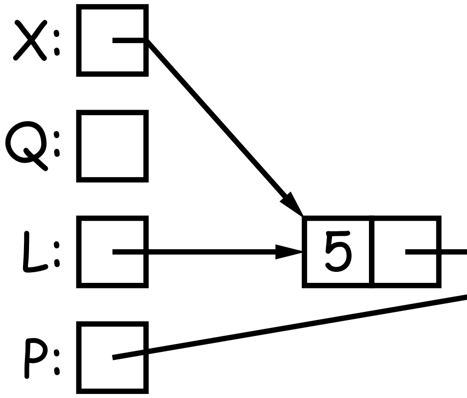


Destructive Incrementing

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

```
/** Destructively add N to L'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.li
/* IntList.lis
Q = dincrList(
```

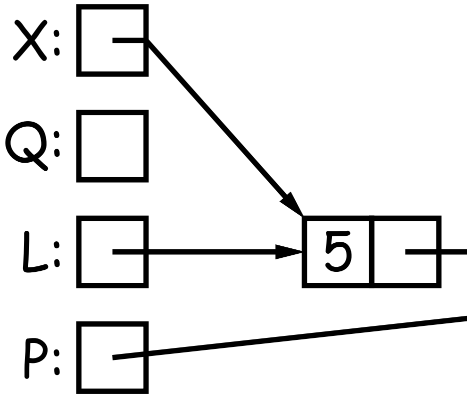


Destructive Incrementing

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

```
/** Destructively add N to L'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.li
/* IntList.lis
Q = dincrList(
```

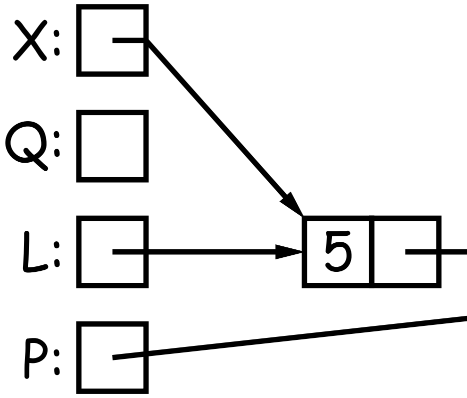


Destructive Incrementing

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

```
/** Destructively add N to L'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.li
/* IntList.lis
Q = dincrList(
```

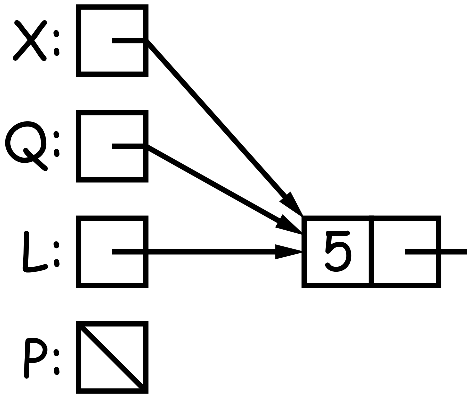


Destructive Incrementing

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

```
/** Destructively add N to L'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.li
/* IntList.lis
Q = dincrList(
```



Another Example: Non-destructive List Deleti

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

```
/** The list resulting from removing all instances of
 *  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.hea
  else
     return /*( L with all x's removed (L!=null, L.hea
}
```

Another Example: Non-destructive List Deleti

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

```
/** The list resulting from removing all instances of
 *  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.hea
  else
     return /*( L with all x's removed (L!=null, L.hea
}
```

Another Example: Non-destructive List Deleti

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

```
/** The list resulting from removing all instances of
 *  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.hea
}
```

Another Example: Non-destructive List Deleti

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

```
/** The list resulting from removing all instances of
 *  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

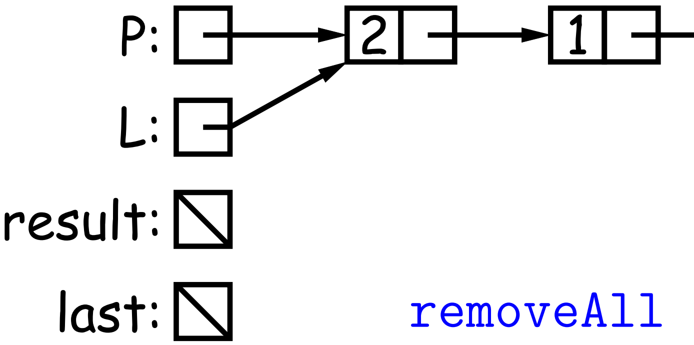
Same as before, but use front-to-back iteration rather than re

```
/** 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 re

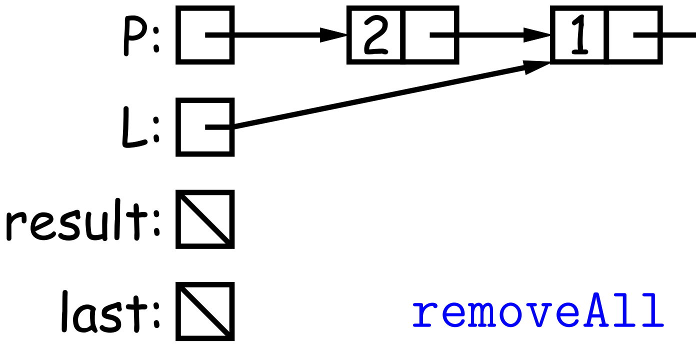
```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last = 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 re

```
/** 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;
}
```



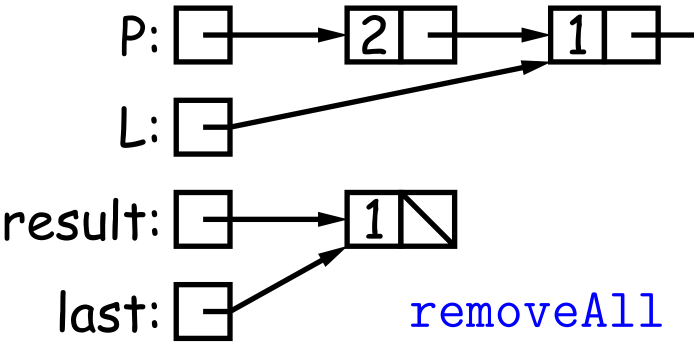
removeAll

P does not c

Iterative Non-destructive List Deletion

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

```
/** 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;
}
```



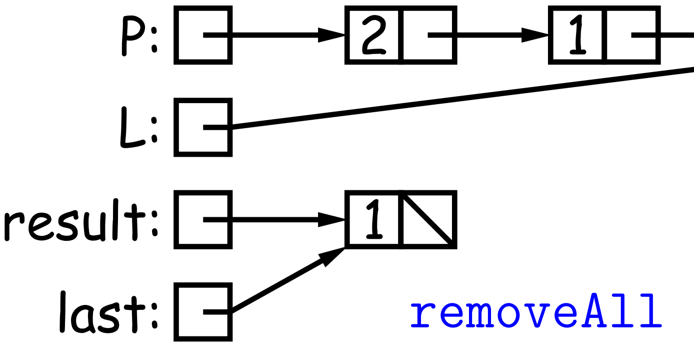
removeAll

P does not c

Iterative Non-destructive List Deletion

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

```
/** 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;
}
```



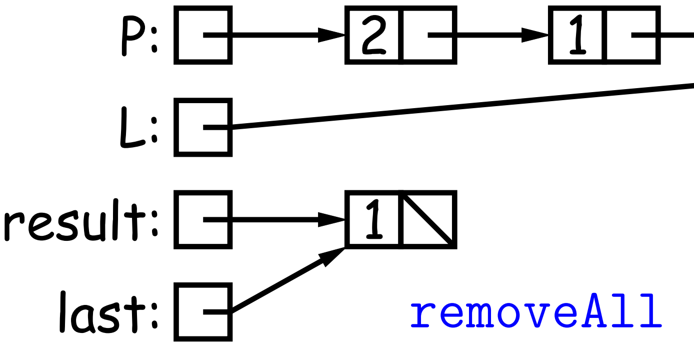
removeAll

P does not c

Iterative Non-destructive List Deletion

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

```
/** 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;
}
```



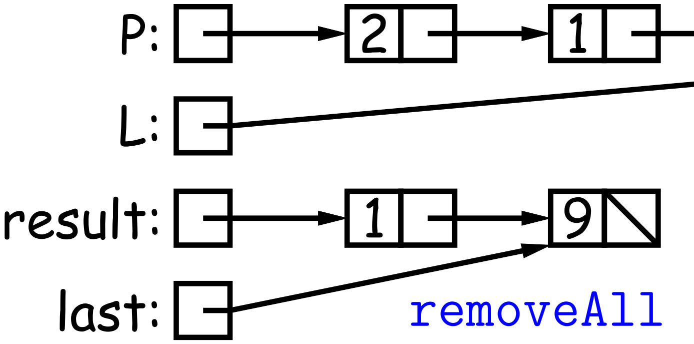
removeAll

P does not c

Iterative Non-destructive List Deletion

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

```
/** 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;
}
```



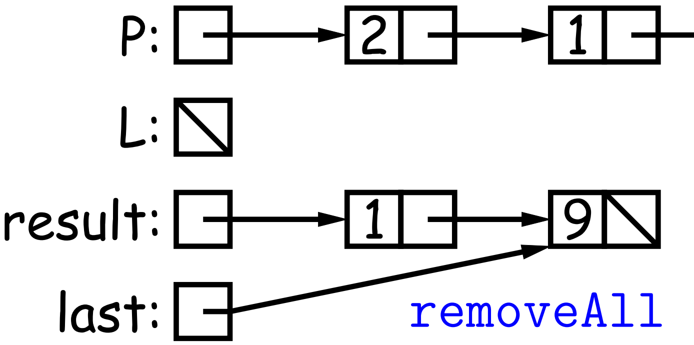
removeAll

P does not c

Iterative Non-destructive List Deletion

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

```
/** 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;
}
```



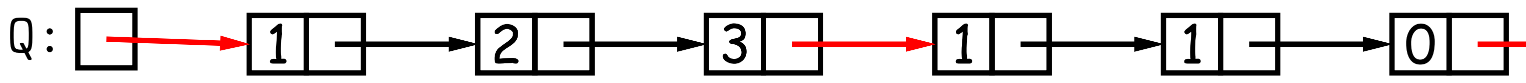
removeAll

P does not c

Destructive Deletion

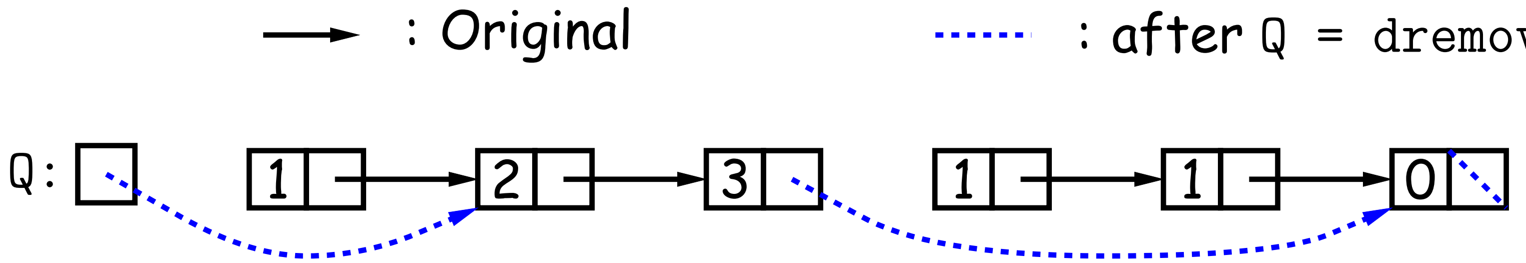
: Original

: after Q = dremov



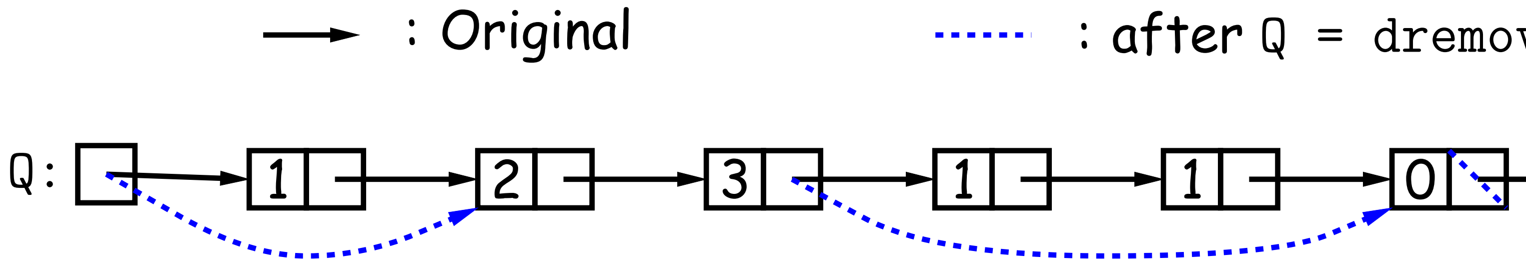
```
/** The list resulting from removing all instances of 
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x's removed )*/;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/
  else {
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

Destructive Deletion



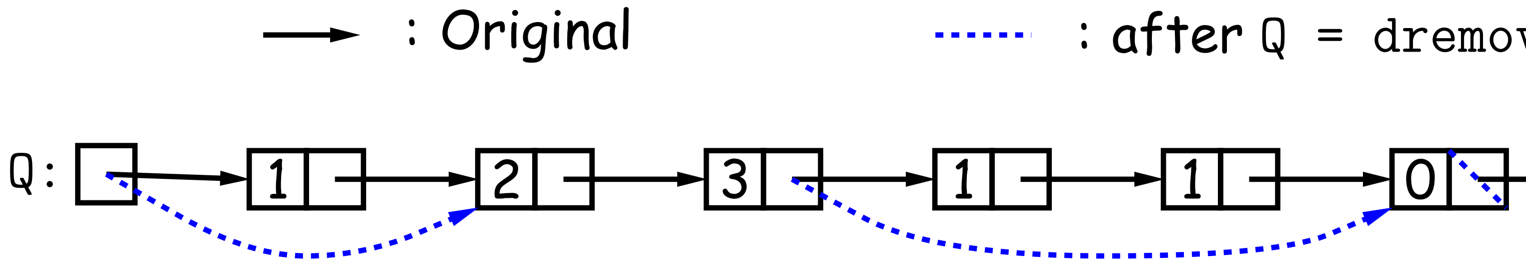
```
/** The list resulting from removing all instances of 
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x's removed )*/;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/
  else {
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

Destructive Deletion



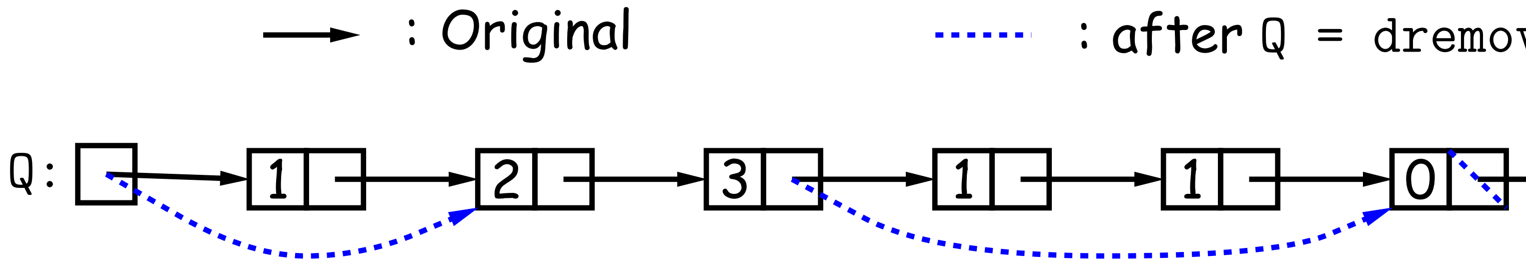
```
/** The list resulting from removing all instances of 
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x's removed )*/;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/
  else {
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

Destructive Deletion



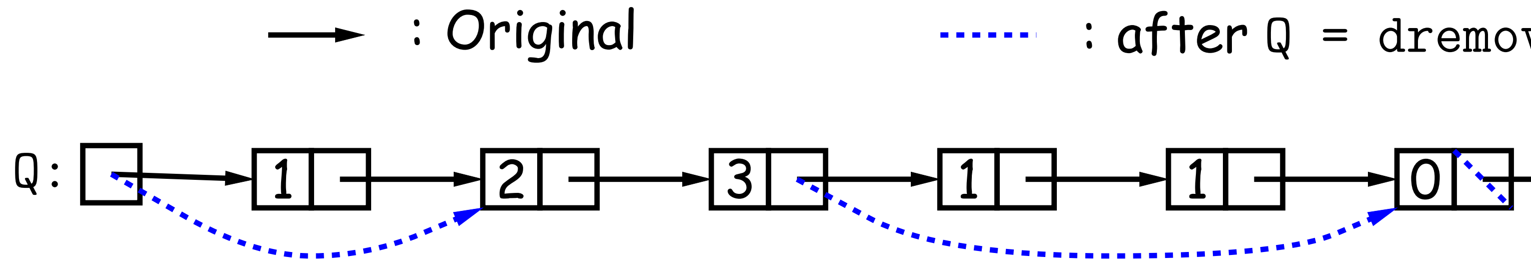
```
/** The list resulting from removing all instances of
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x’s removed )*/;
  else if (L.head == x)
     return /*( L with all x’s removed (L != null) )*/
  else {
 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

Destructive Deletion



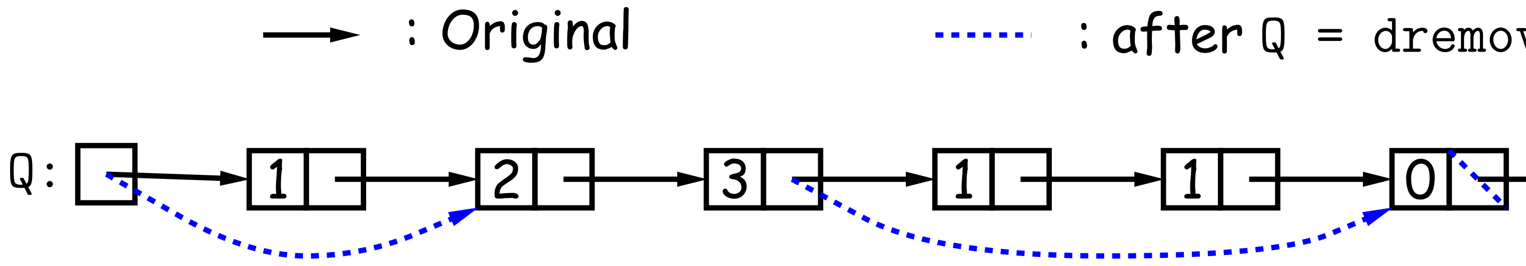
```
/** The list resulting from removing all instances of
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return null;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/
  else {
     /*{ Remove all x's from L's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

Destructive Deletion



```
/** The list resulting from removing all instances of
 *  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

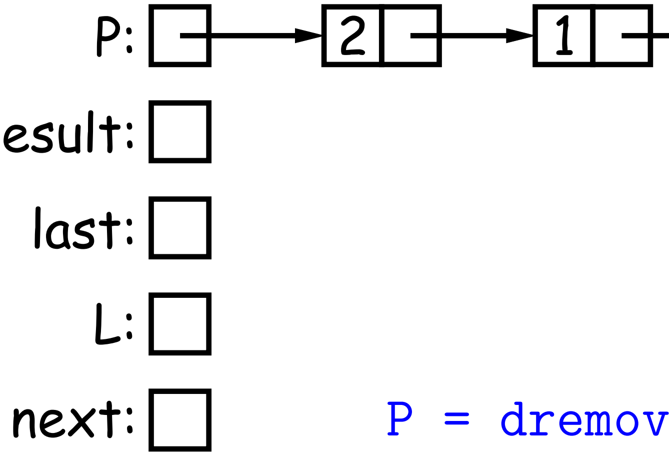


```
/** The list resulting from removing all instances of
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return
  else if (L.head == x)
     return dremoveAll(L.tail, x);
  else {
     L.tail = dremoveAll(L.tail, x);
     return L;
  }
}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

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;
}
```

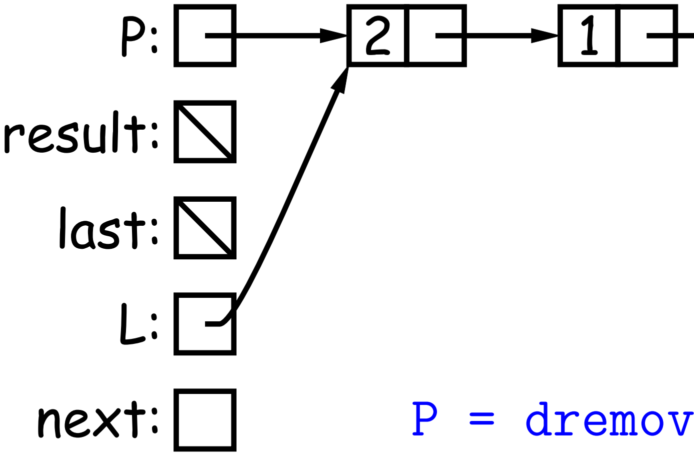
Iterative Destructive Deletion



P = dremov

```
/** 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;
```

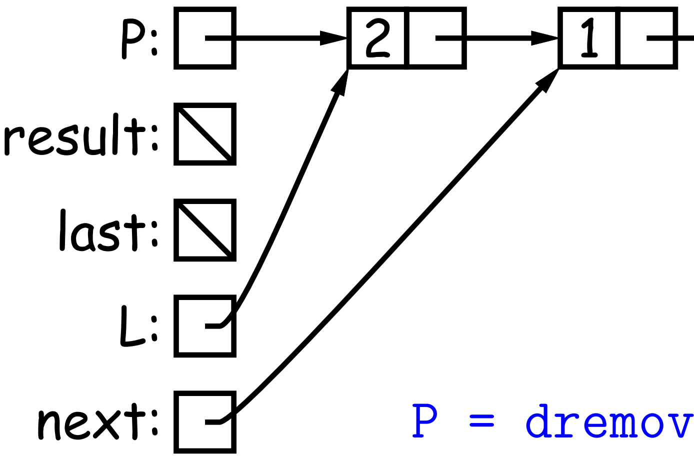
Iterative Destructive Deletion



```
P = dremov
```

```
/** 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;
```

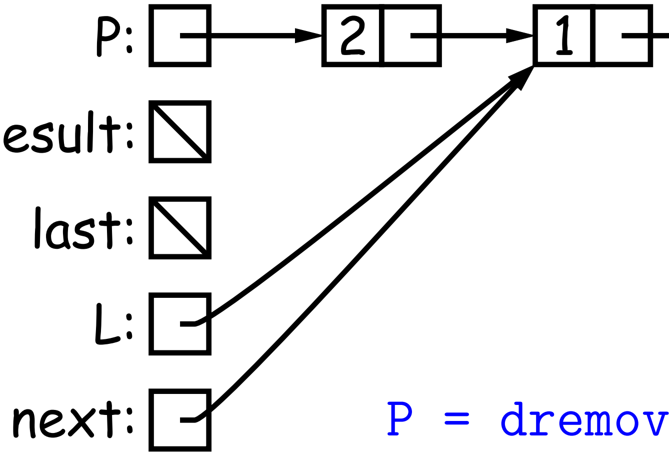
Iterative Destructive Deletion



P = dremov

```
/** 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;
```

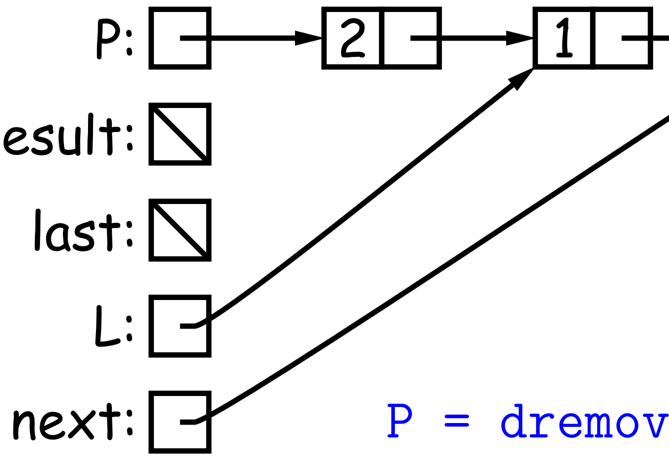
Iterative Destructive Deletion



P = dremov

```
/** 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;
```

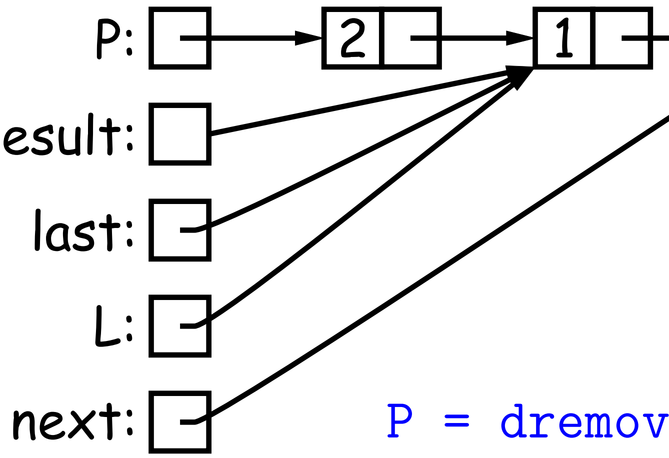
Iterative Destructive Deletion



P = dremov

```
/** 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;
```

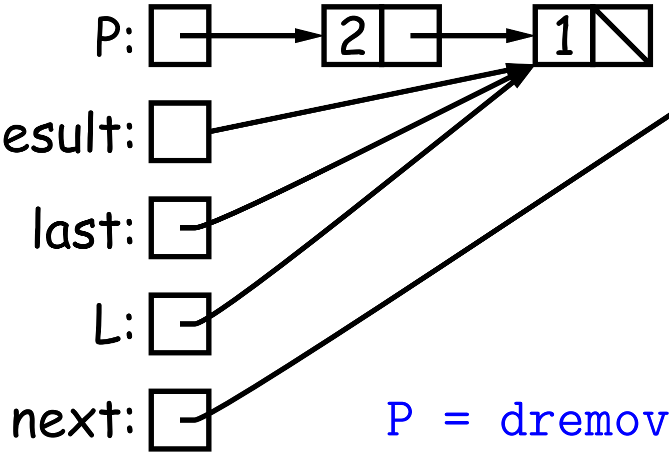
Iterative Destructive Deletion



P = dremov

```
/** 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;
```

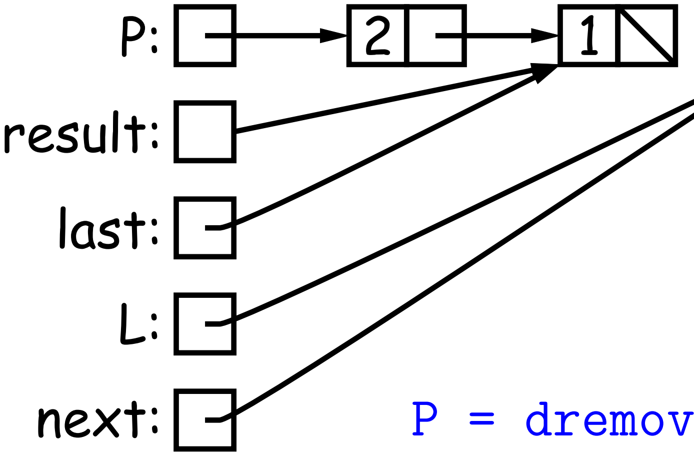
Iterative Destructive Deletion



P = dremov

```
/** 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;
```

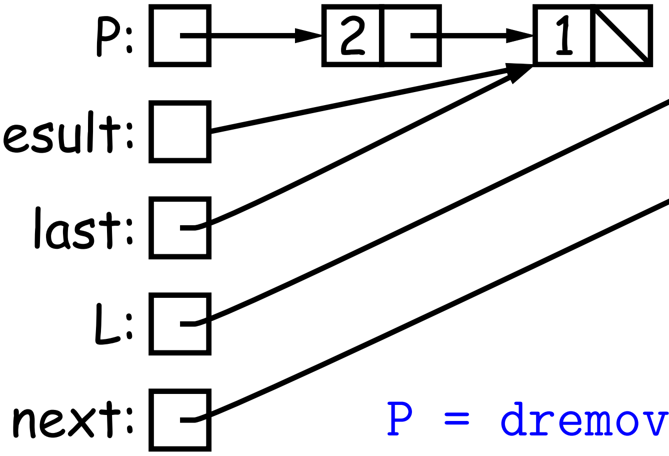
Iterative Destructive Deletion



P = dremov

```
/** 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;
```

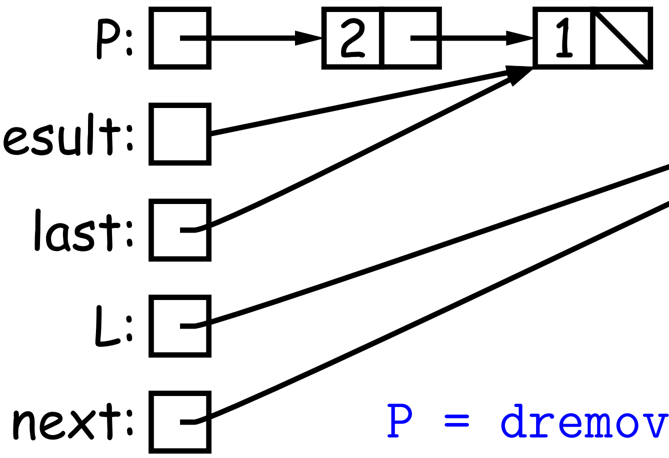
Iterative Destructive Deletion



P = dremov

```
/** 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;
```

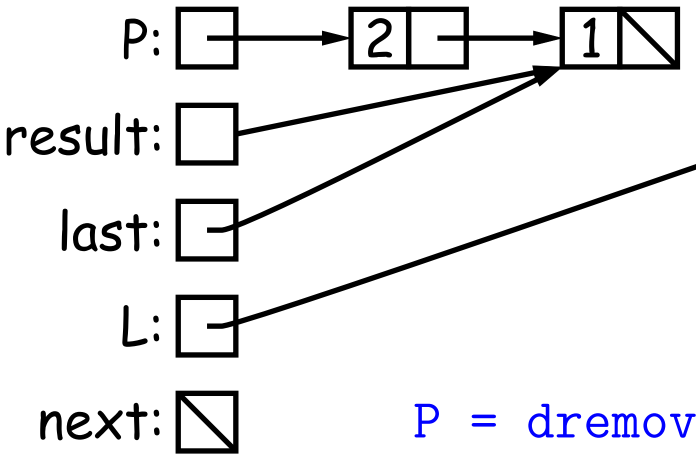
Iterative Destructive Deletion



P = dremov

```
/** 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;
```

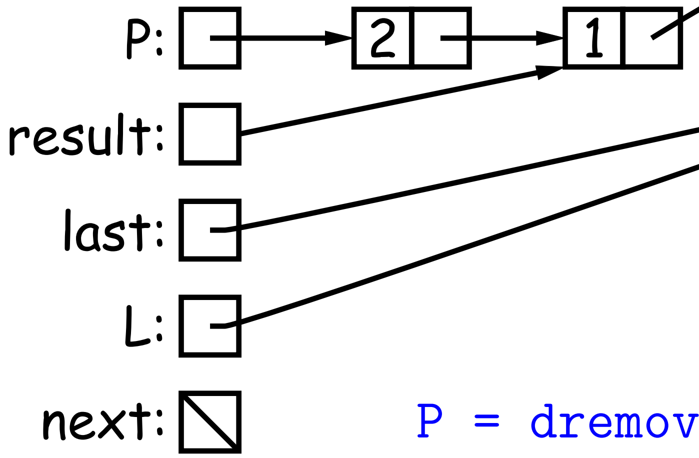
Iterative Destructive Deletion



P = dremov

```
/** 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;
```

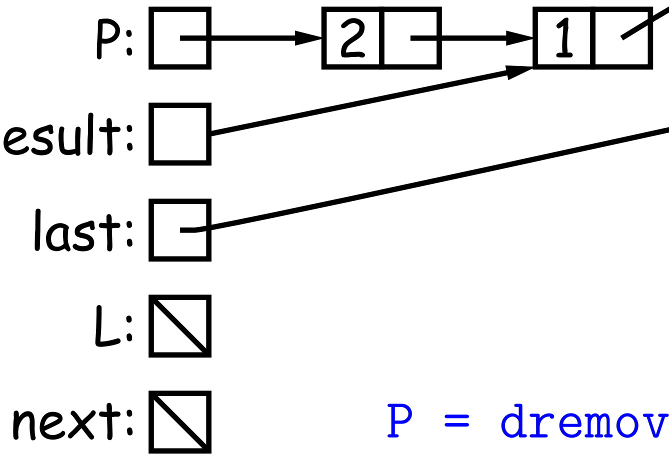
Iterative Destructive Deletion



P = dremov

```
/** 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;
```

Iterative Destructive Deletion



P:

2

1

result:

last:

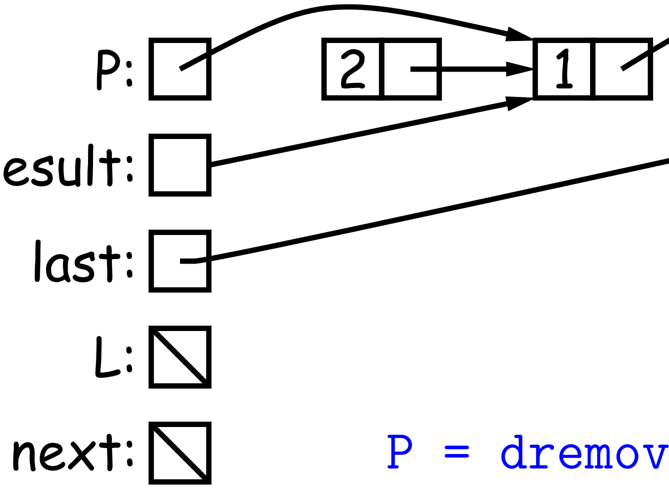
L:

next:

```
P = dremov
```

```
/** 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;
```

Iterative Destructive Deletion



P = dremov

```
/** 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;
```

Aside: How to Write a Loop (in Theory)

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

```
// Invariant must be true here
while (condition) { // condition must not have sid
  // (Invariant will necessarily be 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* and *condition* false, our job is done!

Relationship to Recursion

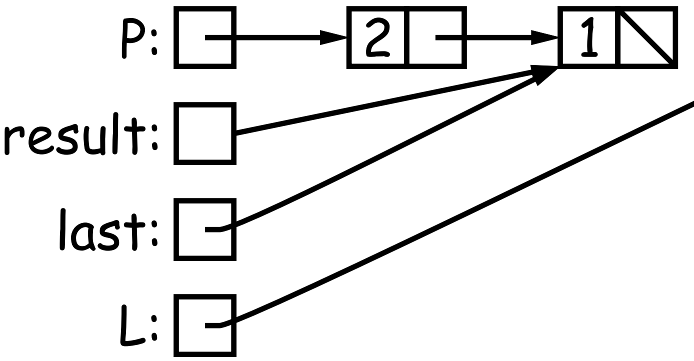
- Another way to see this is to consider an equivalent recurs
  cedure:

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

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

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 = dremov

** Invariant:

- result points to the list o final result except for tho ward.
- L points to an unchange original list of items in L.
- last points to the last it or is null if result is null.