ecture #5: Simple Pointer Manipulation

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

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

e pointer hacking.

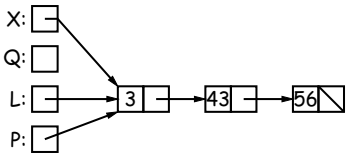
labs and homework: We'll be lenient about accepting rk and labs for lab1, lab2, and hw0. Just get it done: oint is getting to understand the tools involved. We will ubmissions by email.

Destructive Incrementing

utions may modify objects in the original list to save

```
y add N to L's items. */
incrList(IntList P, int n) {

crList(P.tail, n);

y add N to L's items. */
incrList(IntList L, int n)

o more than count!
 = L; p != null; p = p.tail)
```

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

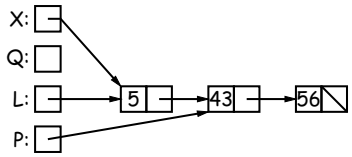


Destructive Incrementing

utions may modify objects in the original list to save

```
y add N to L's items. */
incrList(IntList P, int n) {

crList(P.tail, n);

y add N to L's items. */
incrList(IntList L, int n)

o more than count!
 = L; p != null; p = p.tail)
```

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

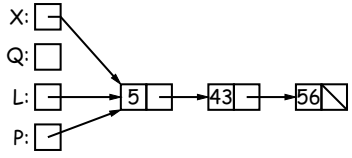


Destructive Incrementing

utions may modify objects in the original list to save

```
y add N to L's items. */
incrList(IntList P, int n) {

crList(P.tail, n);

y add N to L's items. */
incrList(IntList L, int n)

o more than count!
 = L; p != null; p = p.tail)
```

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

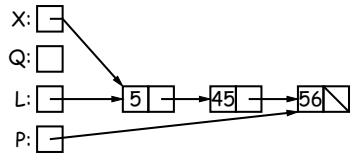


Destructive Incrementing

utions may modify objects in the original list to save

```
y add N to L's items. */
incrList(IntList P, int n) {

crList(P.tail, n);

y add N to L's items. */
incrList(IntList L, int n)

o more than count!
 = L; p != null; p = p.tail)
```

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

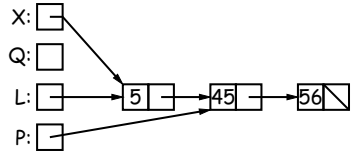


Destructive Incrementing

utions may modify objects in the original list to save

```
y add N to L's items. */
incrList(IntList P, int n) {

crList(P.tail, n);

y add N to L's items. */
incrList(IntList L, int n)

o more than count!
 = L; p != null; p = p.tail)
```

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



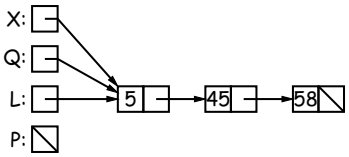
Destructive Incrementing

utions may modify objects in the original list to save

```
y add N to L's items. */
incrList(IntList P, int n) {
```

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

```
crList(P.tail, n);
```




```
y add N to L's items. */
incrList(IntList L, int n)

o more than count!
 = L; p != null; p = p.tail)
```


Example: Non-destructive List Deletion

[2, 1, 2, 9, 2], we want `removeAll(L,2)` to be the new

```
esulting from removing all instances of X from L
ictively. */
 removeAll(IntList L, int x) {
l1)
ill;
ead == x)
( L with all x's removed (L!=null, L.head==x) )*/;

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


Example: Non-destructive List Deletion

[2, 1, 2, 9, 2], we want `removeAll(L,2)` to be the new

```
esulting from removing all instances of X from L
ictively. */
 removeAll(IntList L, int x) {
l1)
ill;
ead == x)
moveAll(L.tail, x);

w IntList(L.head, removeAll(L.tail, x));
```

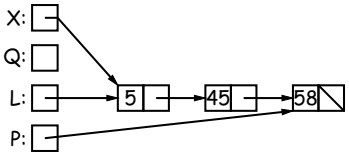

Destructive Incrementing

utions may modify objects in the original list to save

```
y add N to L's items. */
incrList(IntList P, int n) {
```

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

```
crList(P.tail, n);
```




```
y add N to L's items. */
incrList(IntList L, int n)

o more than count!
 = L; p != null; p = p.tail)
```


Example: Non-destructive List Deletion

[2, 1, 2, 9, 2], we want `removeAll(L,2)` to be the new

```
esulting from removing all instances of X from L
ictively. */
 removeAll(IntList L, int x) {
l1)
( null with all x's removed )*/;
ead == x)
( L with all x's removed (L!=null, L.head==x) )*/;

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


Example: Non-destructive List Deletion

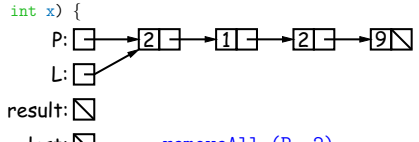
[2, 1, 2, 9, 2], we want `removeAll(L,2)` to be the new

```
esulting from removing all instances of X from L
ictively. */
 removeAll(IntList L, int x) {
l1)
ill;
ead == x)
moveAll(L.tail, x);

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

ative Non-destructive List Deletion

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

```
ilting from removing all instances
non-destructively. */
emoveAll(IntList L, int x) {
, last;
= null;
ill; L = L.tail) {
ead)

t == null)
ast = new IntList(L.head, null);

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

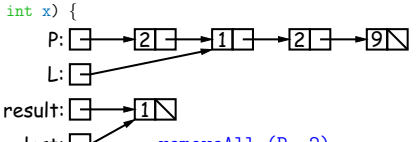


removeAll (P, 2)

ative Non-destructive List Deletion

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

```
ilting from removing all instances
non-destructively. */
emoveAll(IntList L, int x) {
, last;
= null;
ill; L = L.tail) {
ead)

t == null)
ast = new IntList(L.head, null);

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



removeAll (P, 2)

ative Non-destructive List Deletion

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

```
ilting from removing all instances
non-destructively. */
emoveAll(IntList L, int x) {
, last;
= null;
ill; L = L.tail) {
ead)

t == null)
ast = new IntList(L.head, null);

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


removeAll (P, 2)

P does not change!

ative Non-destructive List Deletion

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

```
ilting from removing all instances
non-destructively. */
emoveAll(IntList L, int x) {
, last;
= null;
ill; L = L.tail) {
ead)

t == null)
ast = new IntList(L.head, null);

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


removeAll (P, 2)

P does not change!

ative Non-destructive List Deletion

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

```
ilting from removing all instances
non-destructively. */
emoveAll(IntList L, int x) {
, last;
= null;
ill; L = L.tail) {
ead)

t == null)
ast = new IntList(L.head, null);

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


removeAll (P, 2)

P does not change!

ative Non-destructive List Deletion

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

```
ilting from removing all instances
non-destructively. */
emoveAll(IntList L, int x) {
, last;
= null;
ill; L = L.tail) {
ead)

t == null)
ast = new IntList(L.head, null);

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

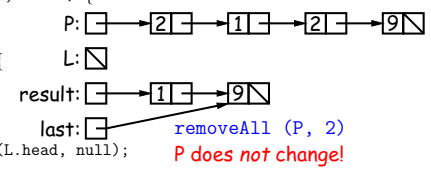

removeAll (P, 2)

P does not change!

ative Non-destructive List Deletion

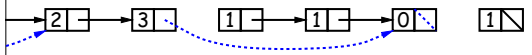
, but use front-to-back iteration rather than recursion.

```
ilting from removing all instances
non-destructively. */
emoveAll(IntList L, int x) {
, last;
= null;
ill; L = L.tail) {
ead)

t == null)
ast = new IntList(L.head, null);

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



Destructive Deletion

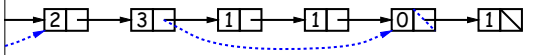
: Original ····· : after Q = dremoveAll (Q,1)




```
esulting from removing all instances of X from L.
al list may be destroyed. */
 dremoveAll(IntList L, int x) {
l1)
( null with all x's removed )*/;
ead == x)
( L with all x's removed (L != null) )*/;

e all x's from L's tail. }*/;
```


Destructive Deletion

: Original ····· : after Q = dremoveAll (Q,1)

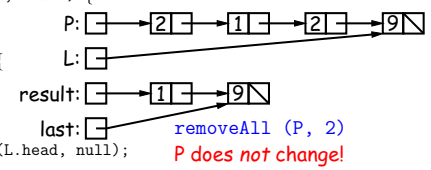



```
esulting from removing all instances of X from L.
al list may be destroyed. */
 dremoveAll(IntList L, int x) {
l1)
( null with all x's removed )*/;
ead == x)
( L with all x's removed (L != null) )*/;

e all x's from L's tail. }*/;
```


ative Non-destructive List Deletion

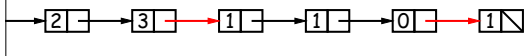
, but use front-to-back iteration rather than recursion.

```
ilting from removing all instances
non-destructively. */
emoveAll(IntList L, int x) {
, last;
= null;
ill; L = L.tail) {
ead)

t == null)
ast = new IntList(L.head, null);

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



Destructive Deletion

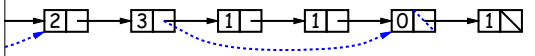
: Original ····· : after Q = dremoveAll (Q,1)




```
esulting from removing all instances of X from L.
al list may be destroyed. */
 dremoveAll(IntList L, int x) {
l1)
( null with all x's removed )*/;
ead == x)
( L with all x's removed (L != null) )*/;

e all x's from L's tail. }*/;
```


Destructive Deletion

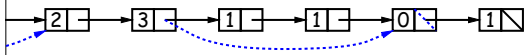
: Original ····· : after Q = dremoveAll (Q,1)




```
esulting from removing all instances of X from L.
al list may be destroyed. */
 dremoveAll(IntList L, int x) {
l1)
( null with all x's removed )*/;
ead == x)
( L with all x's removed (L != null) )*/;

e all x's from L's tail. }*/;
```

Destructive Deletion

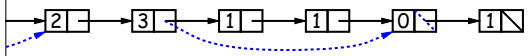
: Original ····· : after Q = dremoveAll (Q,1)




```
esulting from removing all instances of X from L.
al list may be destroyed. */
 dremoveAll(IntList L, int x) {
l1)

ead == x)
emoveAll(L.tail, x);

e all x's from L's tail. }*/;
```


Destructive Deletion

: Original ····· : after Q = dremoveAll (Q,1)



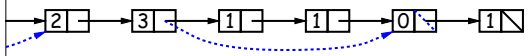

```
esulting from removing all instances of X from L.
al list may be destroyed. */
 dremoveAll(IntList L, int x) {
l1)
ll;
ead == x)
( L with all x's removed (L != null) )*/;

e all x's from L's tail. }*/;
```


Iterative Destructive Deletion

```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```


Destructive Deletion

: Original ····· : after Q = dremoveAll (Q,1)




```
esulting from removing all instances of X from L.
al list may be destroyed. */
 dremoveAll(IntList L, int x) {
l1)

ead == x)
emoveAll(L.tail, x);

dremoveAll(L.tail, x);
```


Iterative Destructive Deletion

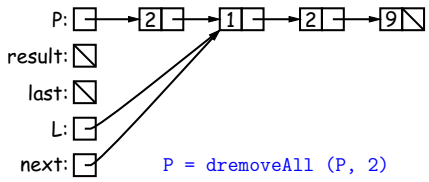
```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```


Iterative Destructive Deletion

```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```

Iterative Destructive Deletion

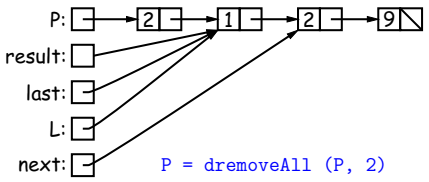
```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```



P = dremoveAll (P, 2)

Iterative Destructive Deletion

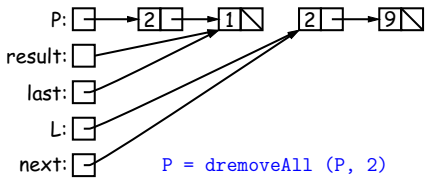
```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```



P = dremoveAll (P, 2)

Iterative Destructive Deletion

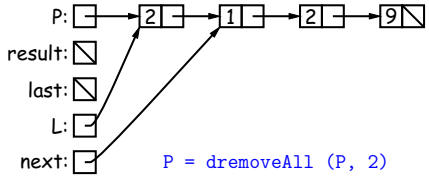
```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```



P = dremoveAll (P, 2)

Iterative Destructive Deletion

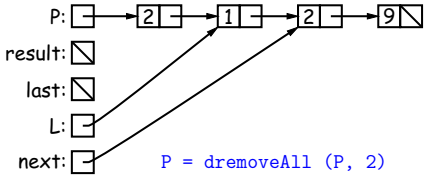
```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```



P = dremoveAll (P, 2)

Iterative Destructive Deletion

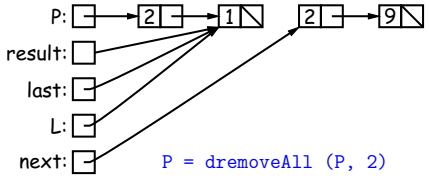
```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```



P = dremoveAll (P, 2)

Iterative Destructive Deletion

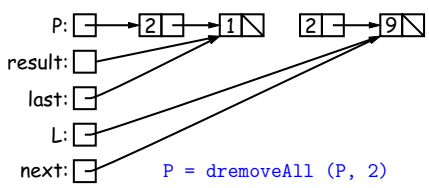
```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```



P = dremoveAll (P, 2)

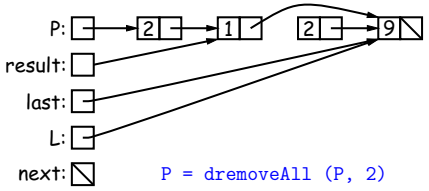
Iterative Destructive Deletion

```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```



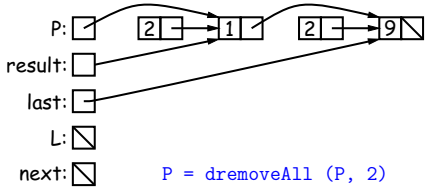
Iterative Destructive Deletion

```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```



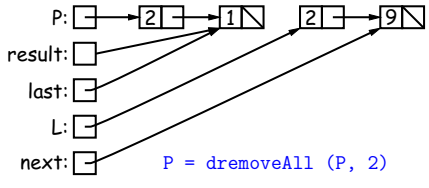
Iterative Destructive Deletion

```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```



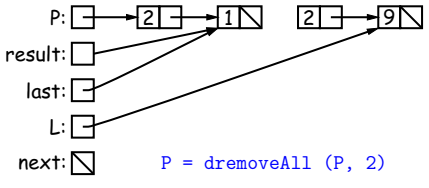
Iterative Destructive Deletion

```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```



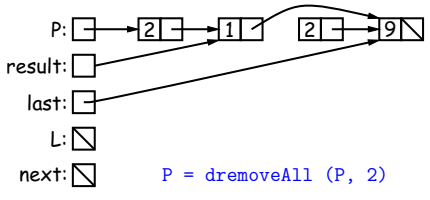
Iterative Destructive Deletion

```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```



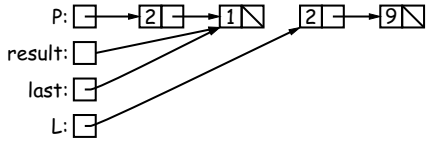
Iterative Destructive Deletion

```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 last.tail = L;
 null;



t;
```



nple: Loop Invariant for dremoveAll

```
ulting from removing all X's from L
y. */
removeAll(IntList L, int x) {
, last;
= null;
 null) {
 = L.tail;
ead) {
= null)
 last = L;

ast.tail = L;
ull;
```



P = 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.

e: How to Write a Loop (in Theory)

description of how things look on *any arbitrary itera-*
op.

tion is known as a *loop invariant,* because it is always
tart of each iteration.

y then must

m any situation consistent with the invariant;

gress in such a way as to make the invariant true again.

condition) {
nvariant true here
body
nvariant again true here

riant true and *condition* false.

p gets the desired answer whenever *invariant* is true
if false, our job is done!