ecture #4: Simple Pointer Manipulation

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

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

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.

free to interpret the absence of a central repository ack of a lab1 submission from you as indicating that you p the course.

Small Test of Understanding

keyword **final** in a variable declaration means that the ue may not be changed after the variable is initialized.

ing class valid?

```
s Issue {

e final IntList aList = new IntList(0, null);

 void modify(int k) {
his.aList.head = k;
```

not?

Small Test of Understanding

keyword **final** in a variable declaration means that the ue may not be changed after the variable is initialized.

ing class valid?

```
s Issue {

e final IntList aList = new IntList(0, null);

 void modify(int k) {
his.aList.head = k;
```

not?

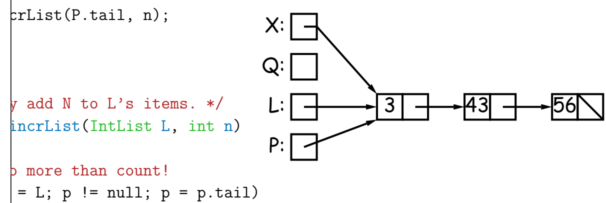
s is *valid*. Although `modify` changes the `head` variable t pointed to by `aList`, it does *not* modify the contents lf (which is a pointer).

Destructive Incrementing

utions may modify objects in the original list to save

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

crList(P.tail, n);

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

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

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

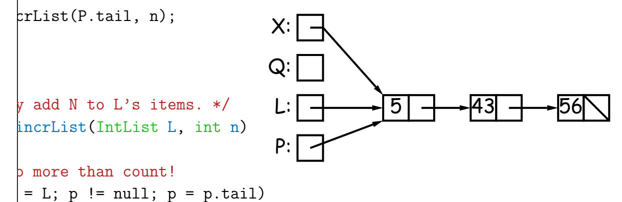


Destructive Incrementing

utions may modify objects in the original list to save

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

crList(P.tail, n);

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

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

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

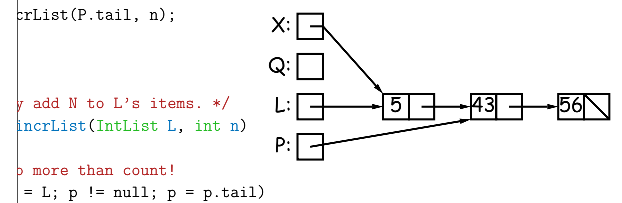


Destructive Incrementing

utions may modify objects in the original list to save

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

crList(P.tail, n);

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

 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 P'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);
```

X: Q: L: P: 5 45 56

Destructive Incrementing

utions may modify objects in the original list to save

```
y add N to P'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);
```

X: Q: L: P: 5 45 56

Destructive Incrementing

utions may modify objects in the original list to save

```
y add N to P'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);
```

X: Q: L: P: 5 45 58

Destructive Incrementing

utions may modify objects in the original list to save

```
y add N to P'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);
```

X: Q: L: P: 5 45 58

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)
ll;
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;
iead == x)
emoveAll(L.tail, 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;
iead == x)
emoveAll(L.tail, x);

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


ative Non-destructive List Deletion

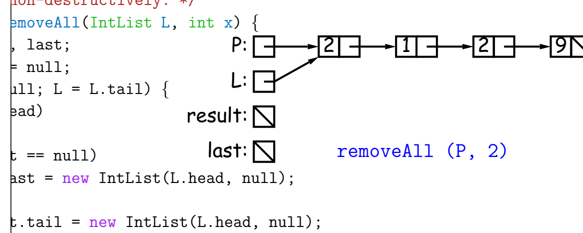
, but use front-to-back iteration rather than recursion.

```
ilting from removing all instances
ion-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);
```


ative Non-destructive List Deletion

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

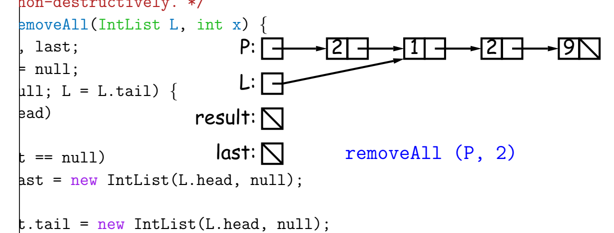
```
ilting from removing all instances
ion-destructively. */
emoveAll(IntList L, int x) {
, last;
= null;
ill; L = L.tail) {
ead)

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

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



ative Non-destructive List Deletion

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

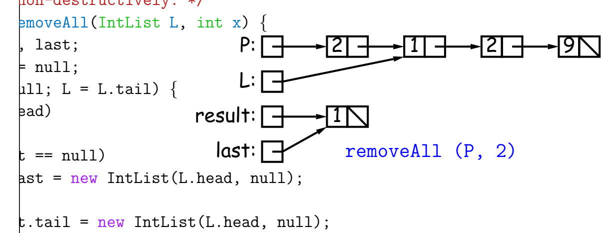
```
ilting from removing all instances
ion-destructively. */
emoveAll(IntList L, int x) {
, last;
= null;
ill; L = L.tail) {
ead)

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

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



ative Non-destructive List Deletion

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

```
ilting from removing all instances
ion-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);
```



ative Non-destructive List Deletion

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

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

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

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

P: 2 1 2 9
L:
result: 1
last:
removeAll (P, 2)

ative Non-destructive List Deletion

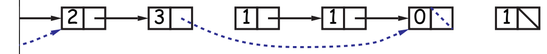
, but use front-to-back iteration rather than recursion.

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

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

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

P: 2 1 2 9
L:
result: 1 9
last:
removeAll (P, 2)

Destructive Deletion

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




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

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

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

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

P: 2 1 2 9
L:
result: 1
last:
removeAll (P, 2)

ative Non-destructive List Deletion

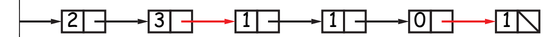
, but use front-to-back iteration rather than recursion.

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

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

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

P: 2 1 2 9
L:
result: 1 9
last:
removeAll (P, 2)

Destructive Deletion

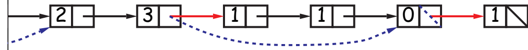
Original ----- : after Q = dremoveAll (Q,1)




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

)riginal ----- : after Q = dremoveAll (Q,1)




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

)riginal ----- : after Q = dremoveAll (Q,1)

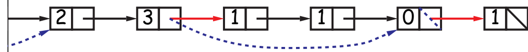
```
esulting from removing all instances of X from L.
al list may be destroyed. */
 dremoveAll(IntList L, int x) {
l)

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

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

)riginal ----- : after Q = dremoveAll (Q,1)




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

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

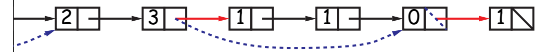

Destructive Deletion

)riginal ----- : after Q = dremoveAll (Q,1)

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

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


Destructive Deletion

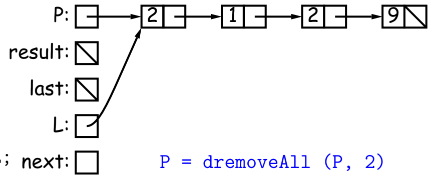
)riginal ----- : after Q = dremoveAll (Q,1)




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

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

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

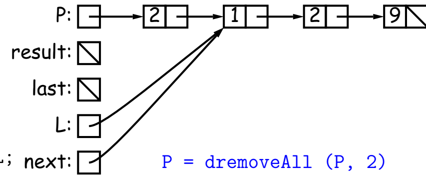
Iterative Destructive Deletion

```
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;
```



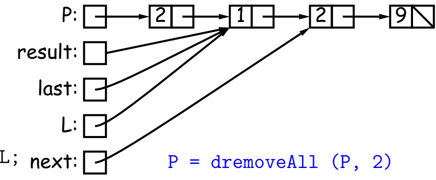
Iterative Destructive Deletion

```
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;
```



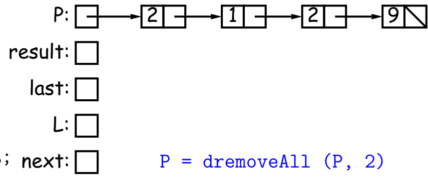
Iterative Destructive Deletion

```
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;
```



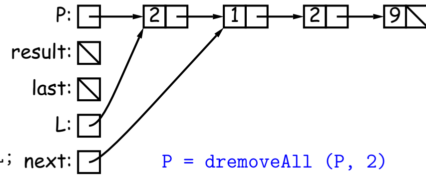
Iterative Destructive Deletion

```
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;
```



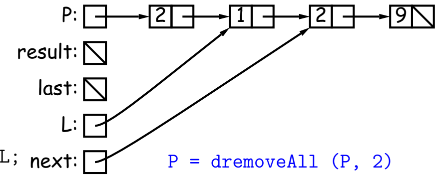
Iterative Destructive Deletion

```
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;
```


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

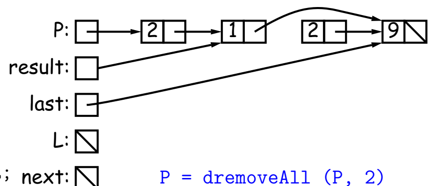
```
esulting from removing all X's from L
ely. */
 dremoveAll(IntList L, int x) {
lt, last;
t = null;
null) {
xt = L.tail;
.head) {
 == null)
 = last = L;

 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;
```

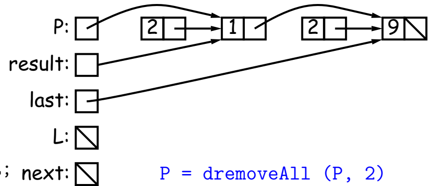
Iterative Destructive Deletion

```
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;
```



k Jump Forward: What, No Functions?

ntains an illustration of an interesting technique in Java with the functions-as-values and higher-order functions prominently in CS61A.

e are no such things. For example, dremoveAll is not a alue". It can only be used in the context of a function All(Q, 7).

the lack of functional values, Java can get the same ing another feature it does share with Python: instance bjects.

ack to this in detail later. For now, let's take a brief

Functional Values

e may write things like this:

```
(L, action):
ly the function F to all items in
uence L in order."""
 in L:
ton(x)

 "b", "c"]
print)                    # Prints a b c on 3 lines.
lambda y: print(y + y))  # Prints aa bb cc
```

by itself denotes a function that can be passed as a lled in doAll as a function.

bda x: ... denotes an anonymous function that prints ation of its argument with itself.

t allow these exactly.

An Alternative

llows another approach:

```
2(L, action):
 in L:
ction.accept(x)
nter1:
ccept(self, y):
rint(y)
nter2:
ccept(self, y):
rint(y + y)
 Printer1())
 Printer2())
```

es have classes and instance methods.

Java Version

Java (as usual) one must specify a good deal more in-

, you need to specify the type of L and action, and the nd returned by accept. For now, we'll just give you the ersion, and explain the details in later lectures.

```
ng {
id doAll(List<String> L, Consumer<String> action) {
String x : L) action.accept(x);


1 implements Consumer<String> {
id accept(String y) { System.out.println(y); }

2 implements Consumer<String> {
id accept(String y) { System.out.println(y + y); }
```

oAll:

```
ll(L, new Printer1());   Something.doAll(L, new Printer2());
```

And Finally, Lambda Expressions

ee, compared to a language such as Python, Java is just
y: we have

```
2 implements Consumer<String> {
id accept(String y) { System.out.println(y + y); }
```

h

```
2()
```

riginal Python version:

```
print(y + y)
```

ficiently annoying that the Java designers decided to
convenient shorthand for the definition of classes like
h

```
> System.out.println(y + y)
```

t of language complexity involved in making it possible
the class definition or most of the `accept` method def-
now, let's just be grateful that someone went to the
rk it out.

Consumer

`sumer` is not actually special; it's simply a generic library
ne `java.util.function.Consumer` if you're curious.

method called `accept`, and `Printer1` and `Printer2` are
at override that method. We'll review what this all

fact, have defined our own class for this purpose, but
advantage of the library?

s type because `doAll` needs a single type for its `action`
but we have at least two different classes (`Printer1`
2) that we want to pass to it.

rves the same purpose as a base type in Python.

gotten all that (or not seen it yet), don't worry; we can
tails later.