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.

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?

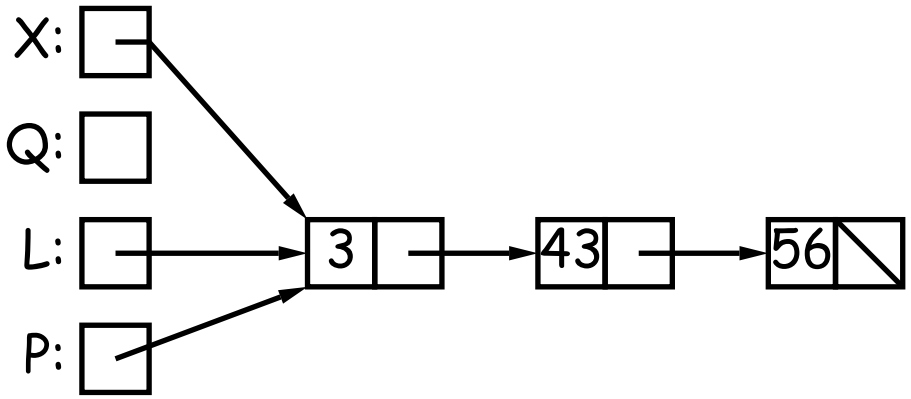
Answer: This is *valid*. Although `modify` changes the `head` variable of the object pointed to by `aList`, it does *not* modify the contents of `aList` itself (which is a pointer).

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);
```

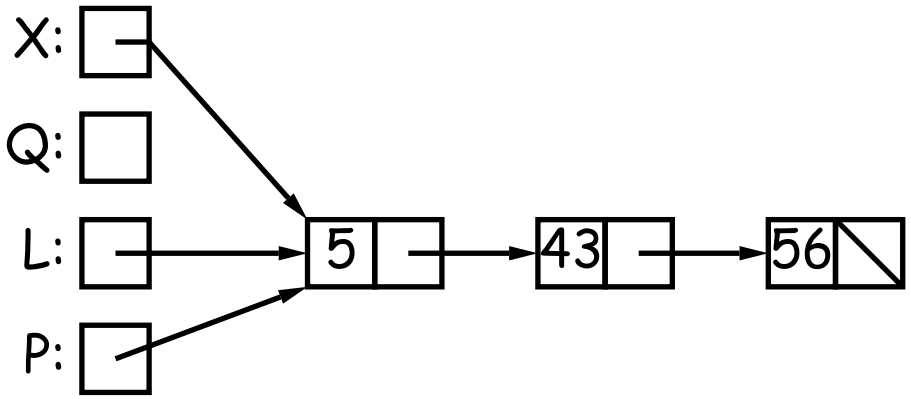


Destructive Incrementing

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);
```

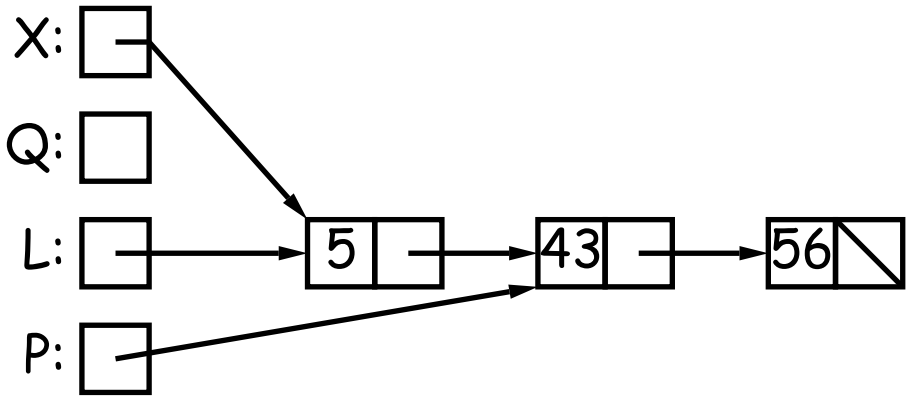


Destructive Incrementing

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);
```

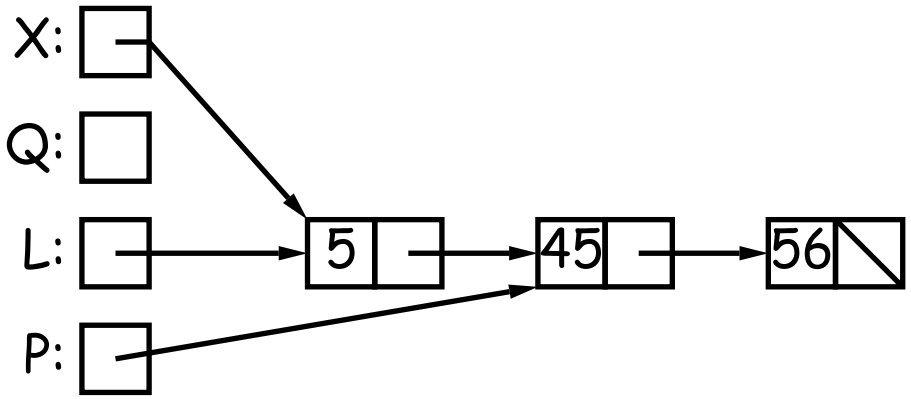


Destructive Incrementing

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);
```

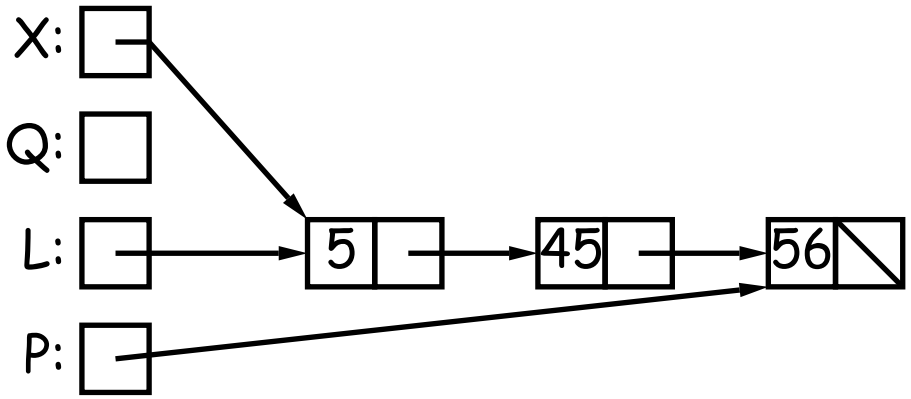


Destructive Incrementing

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);
```

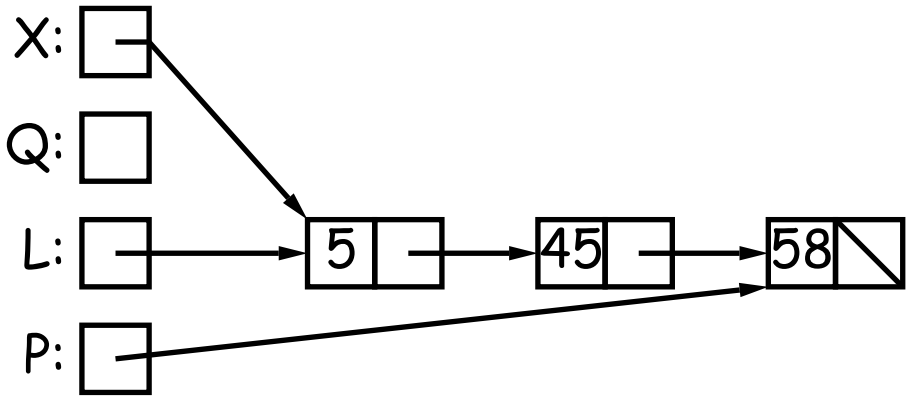


Destructive Incrementing

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);
```

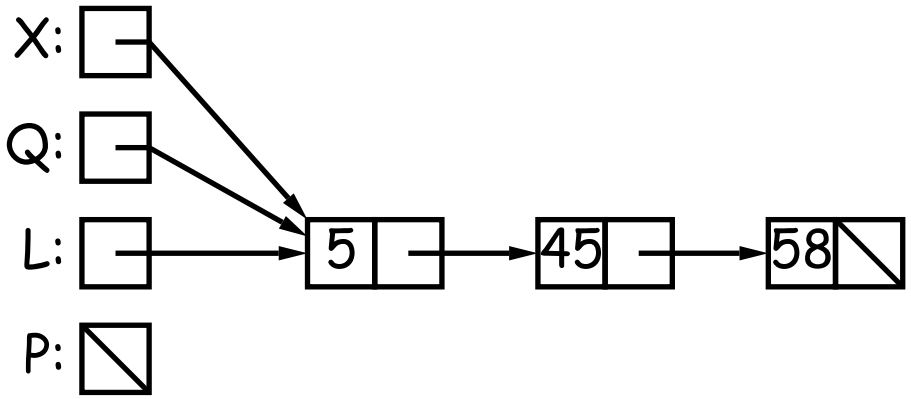


Destructive Incrementing

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

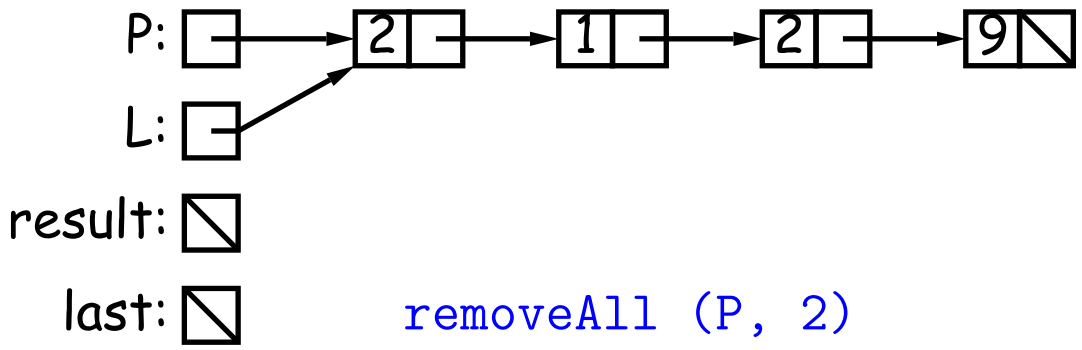
Same as before, but use front-to-back iteration rather than recursion.

```
/** 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.

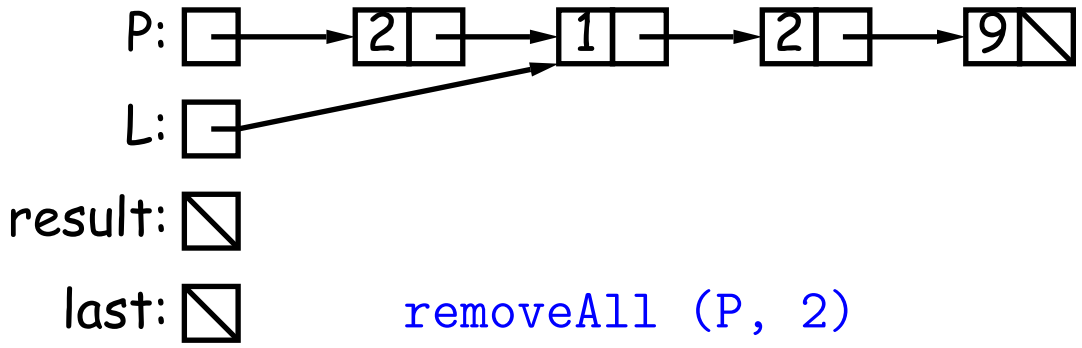
```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last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

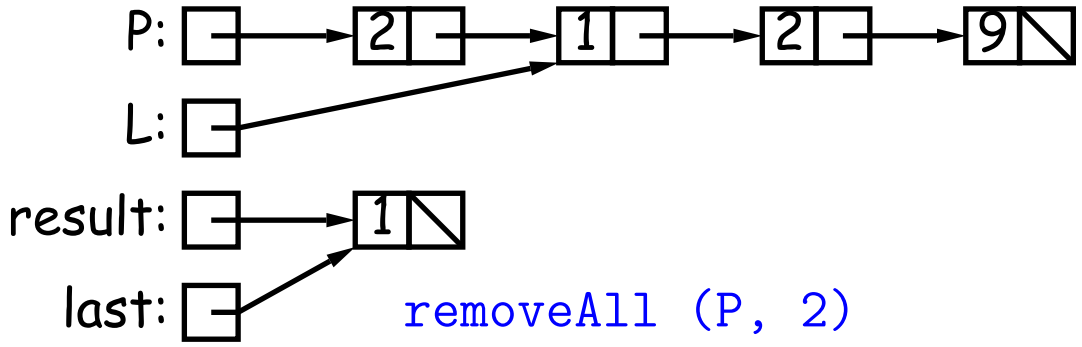
```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last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



Iterative Non-destructive List Deletion

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

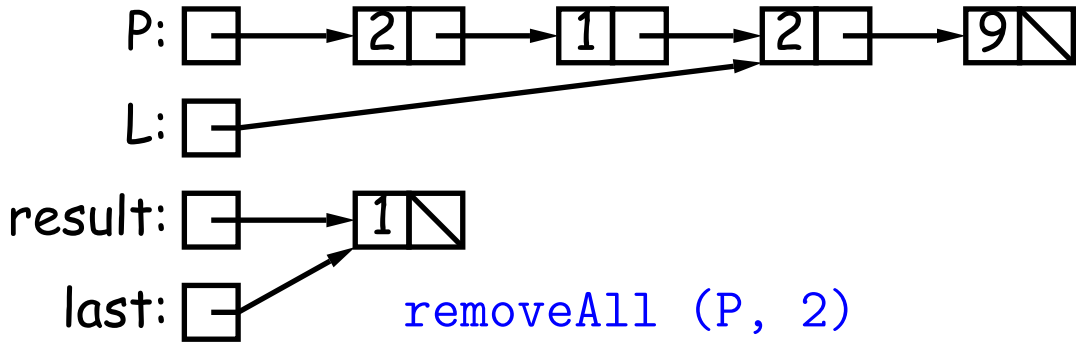
```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last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

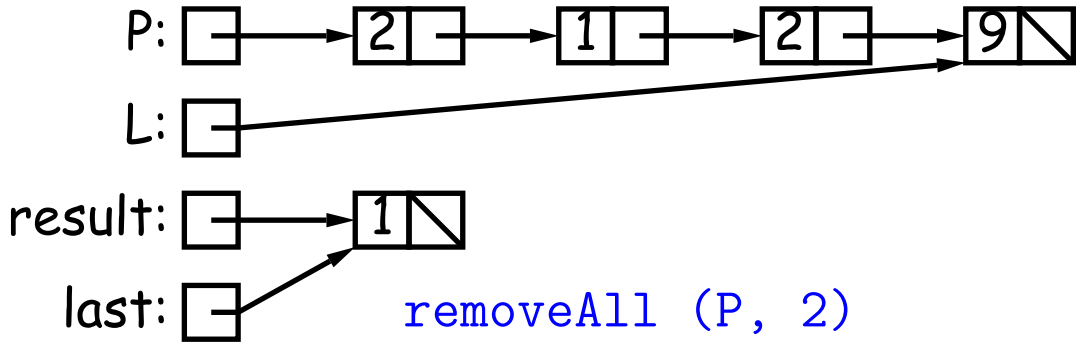
```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last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

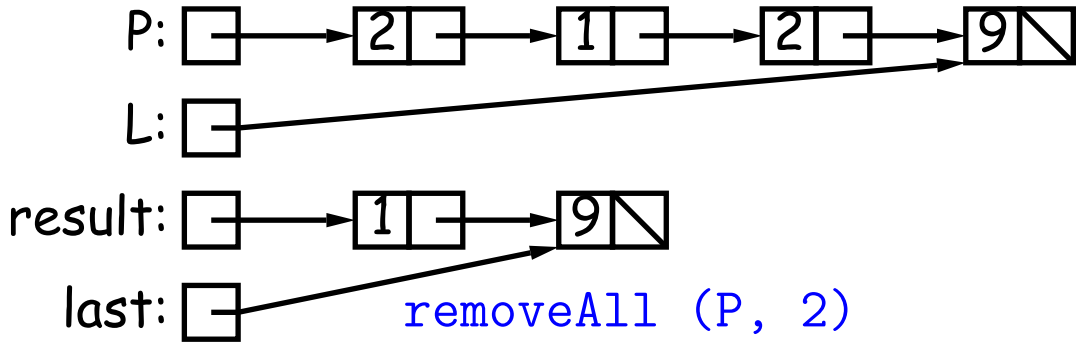
```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last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

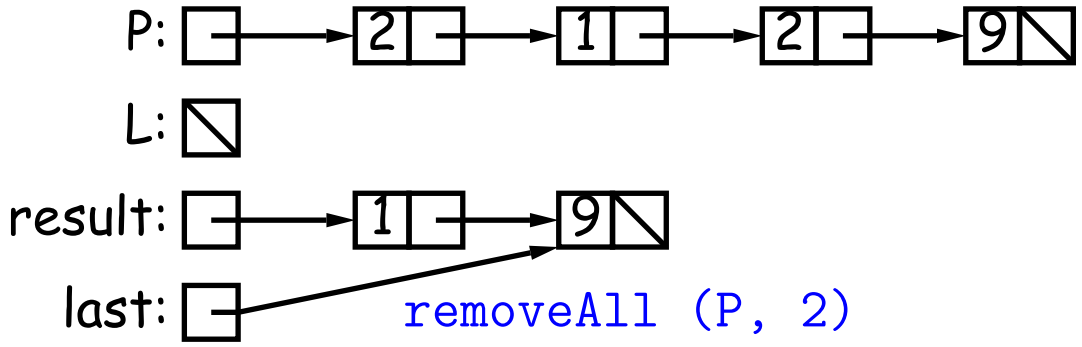
```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last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



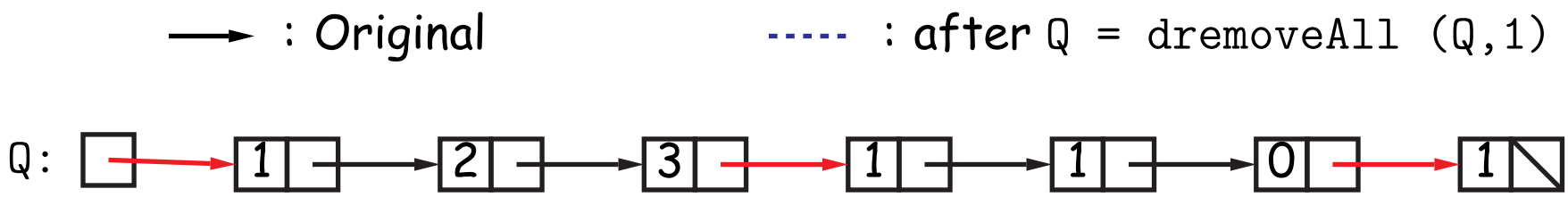
Iterative Non-destructive List Deletion

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;
}
```

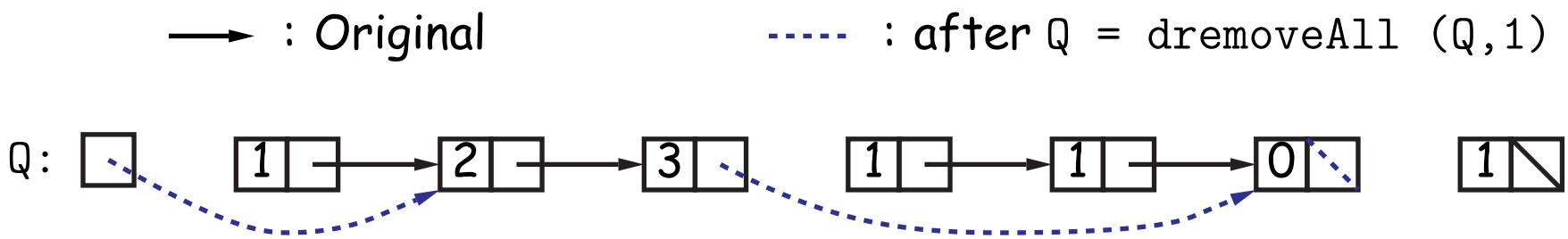


Destructive Deletion



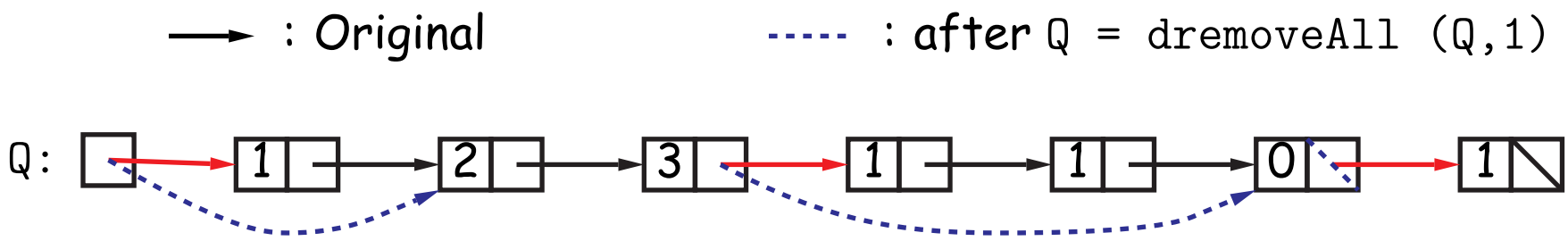

```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x's removed )*/;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/;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

Destructive Deletion



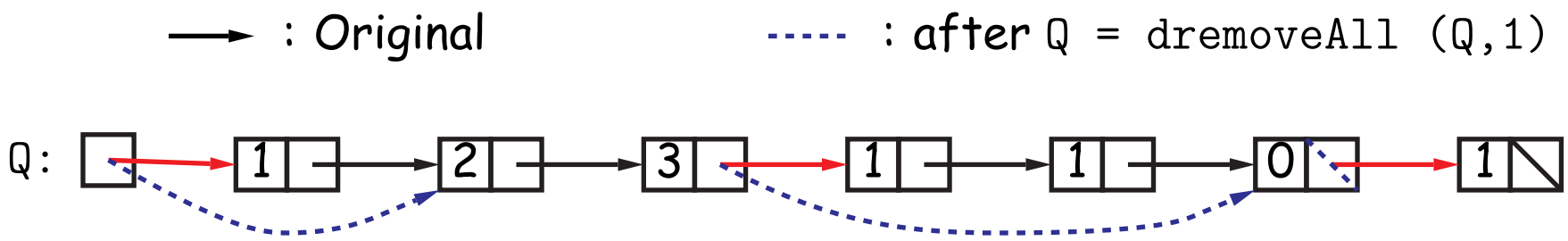

```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x's removed )*/;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/;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

Destructive Deletion



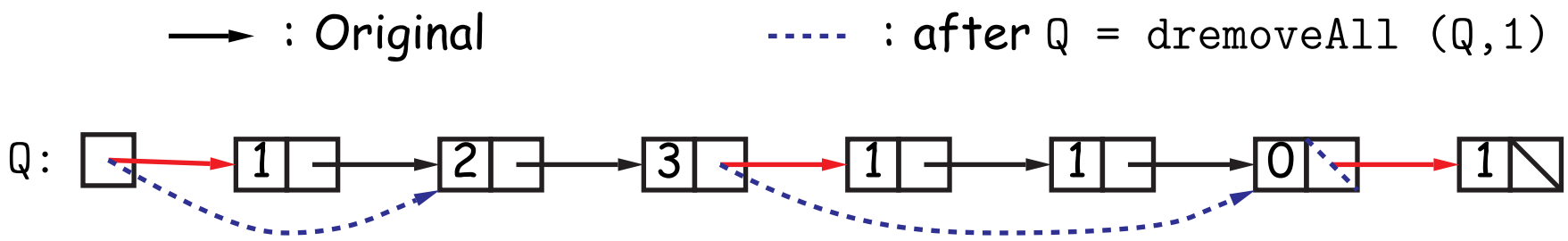

```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x's removed )*/;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/;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

Destructive Deletion



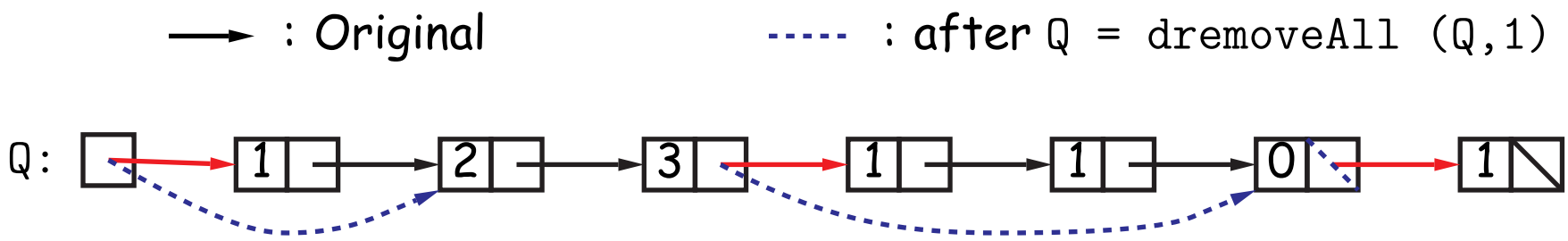

```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x's removed )*/;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/;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

Destructive Deletion



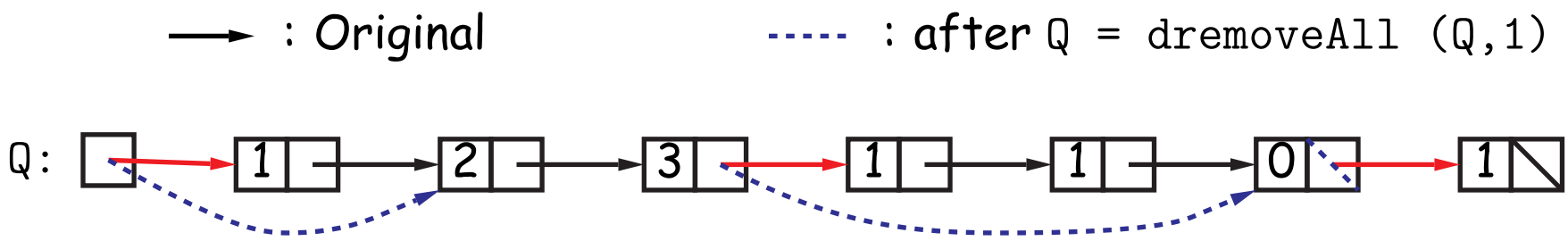

```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return null;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/;
  else {
     /*{ Remove all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

Destructive Deletion




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

Destructive Deletion



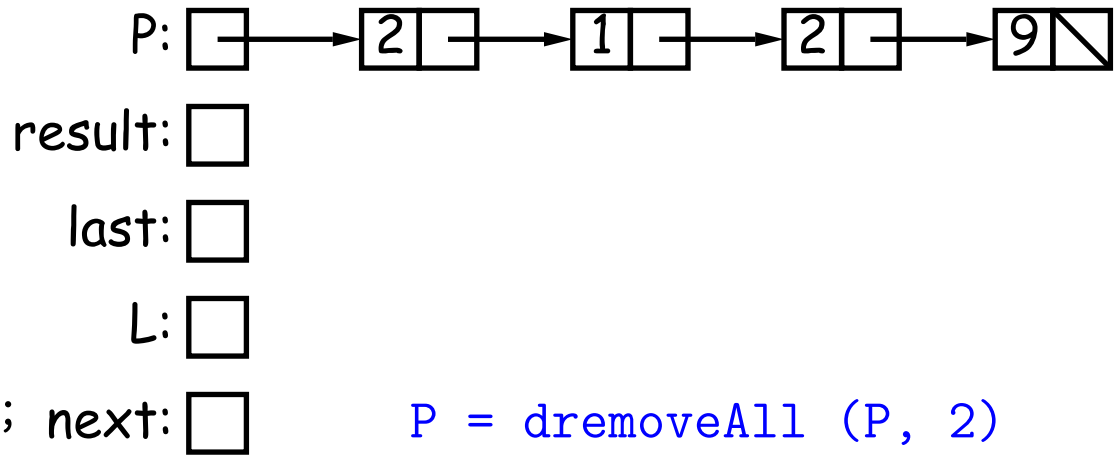

```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return
  else if (L.head == x)
     return dremoveAll(L.tail, x);
  else {
     L.tail = dremoveAll(L.tail, x);
     return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

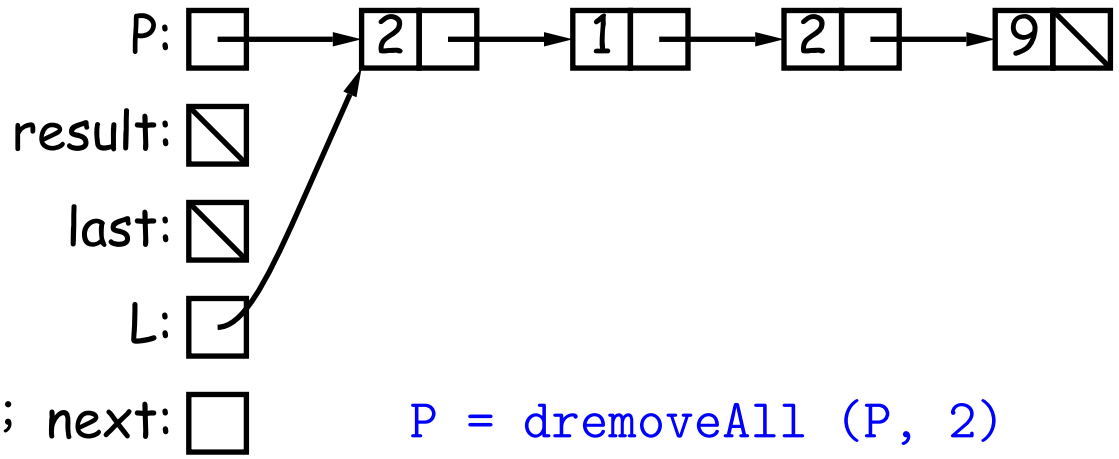
Iterative Destructive Deletion

```
/** 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;
}
```



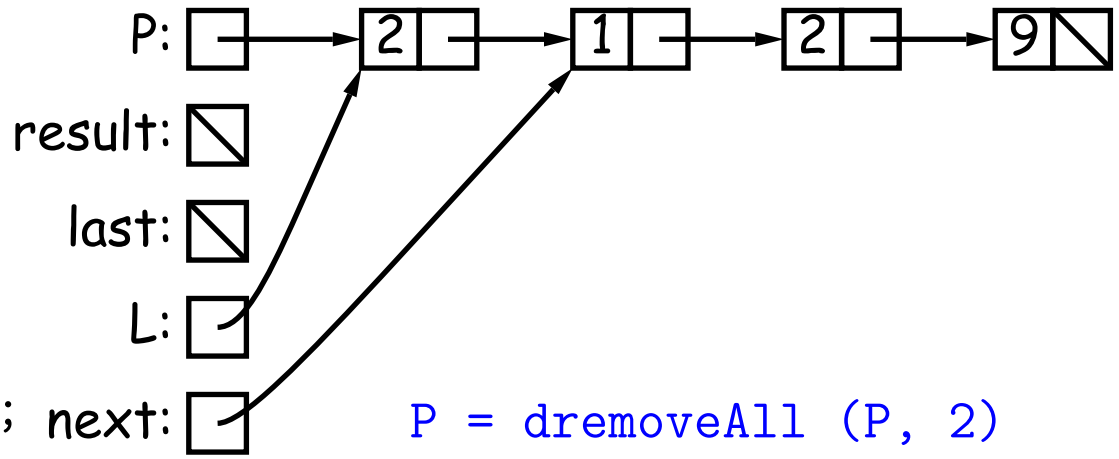
Iterative Destructive Deletion

```
/** 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;
}
```



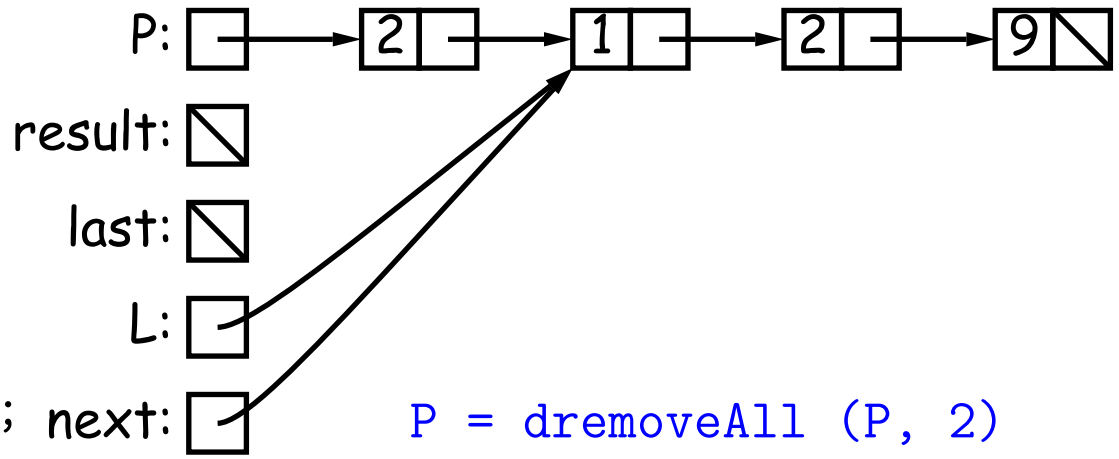
Iterative Destructive Deletion

```
/** 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;
}
```



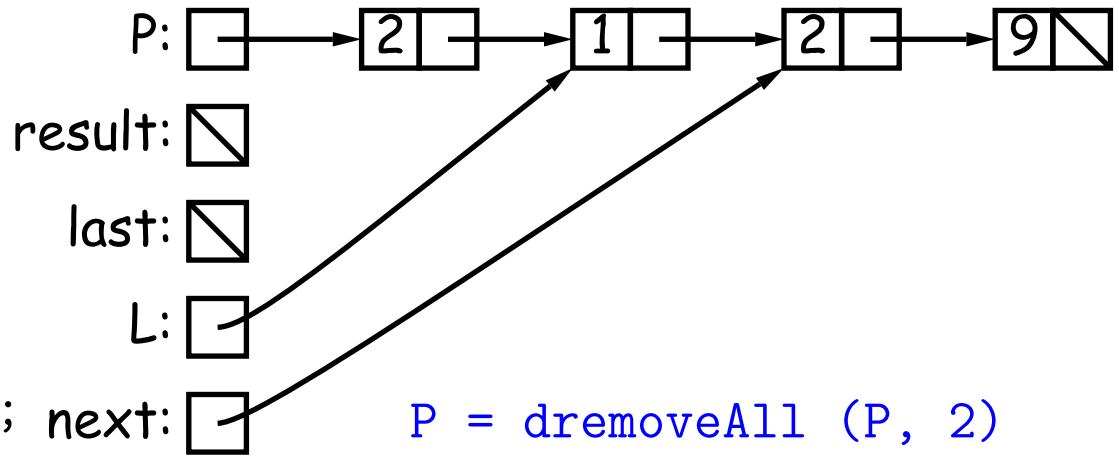
Iterative Destructive Deletion

```
/** 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;
}
```



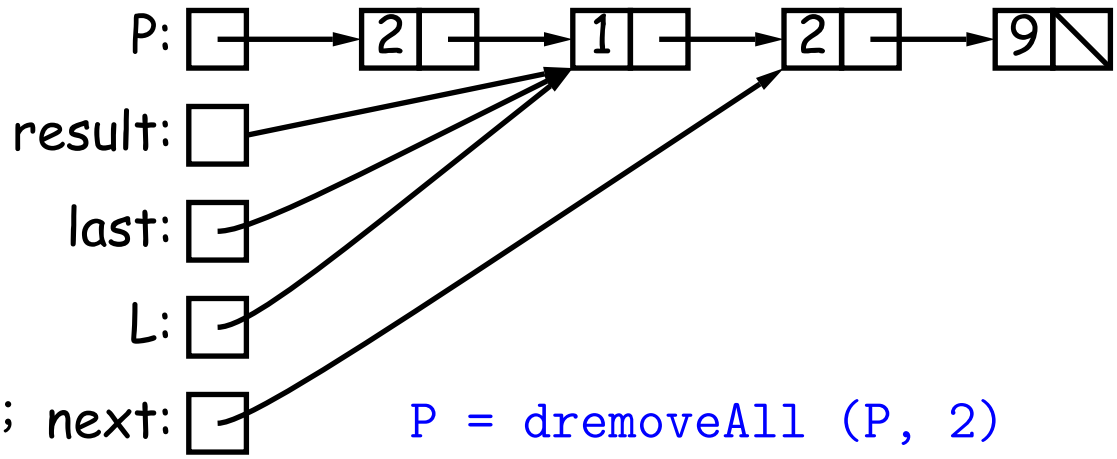
Iterative Destructive Deletion

```
/** 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;
}
```



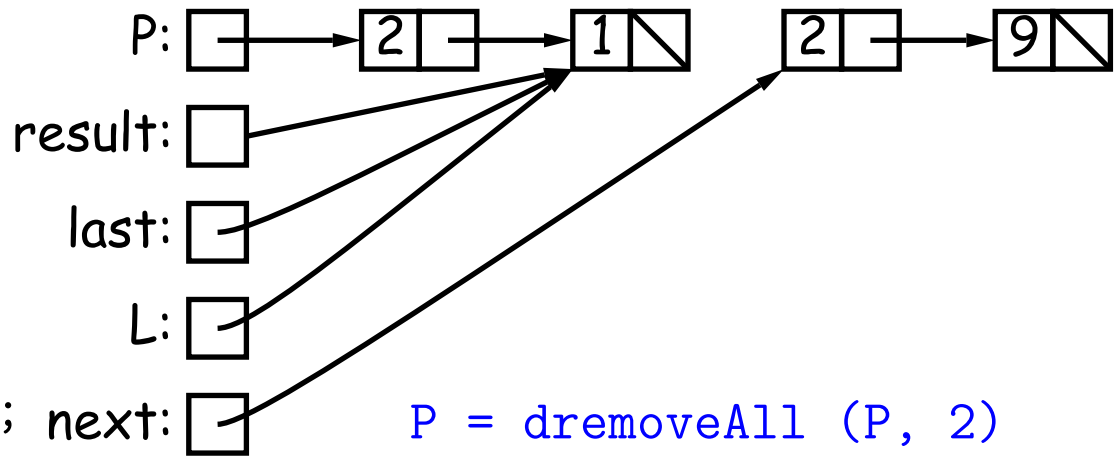
Iterative Destructive Deletion

```
/** 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;
}
```



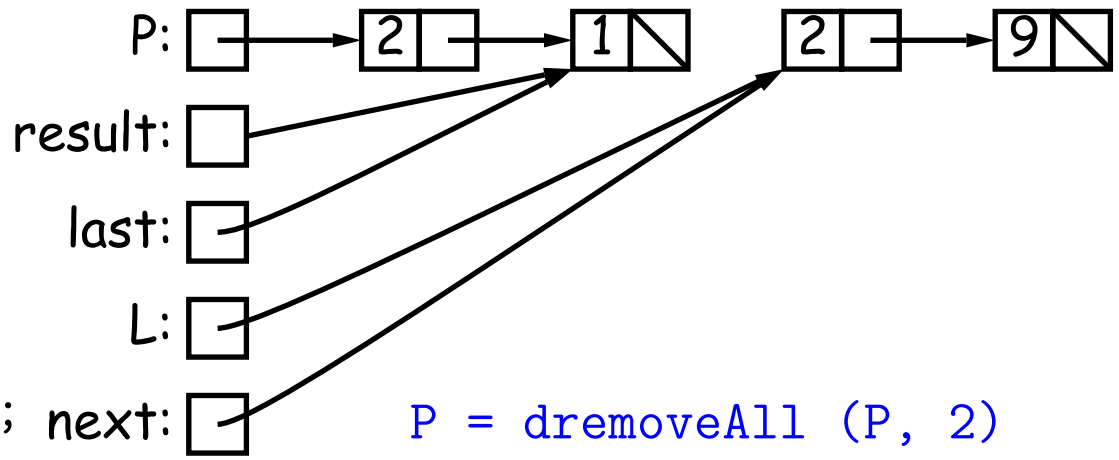
Iterative Destructive Deletion

```
/** 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;
}
```



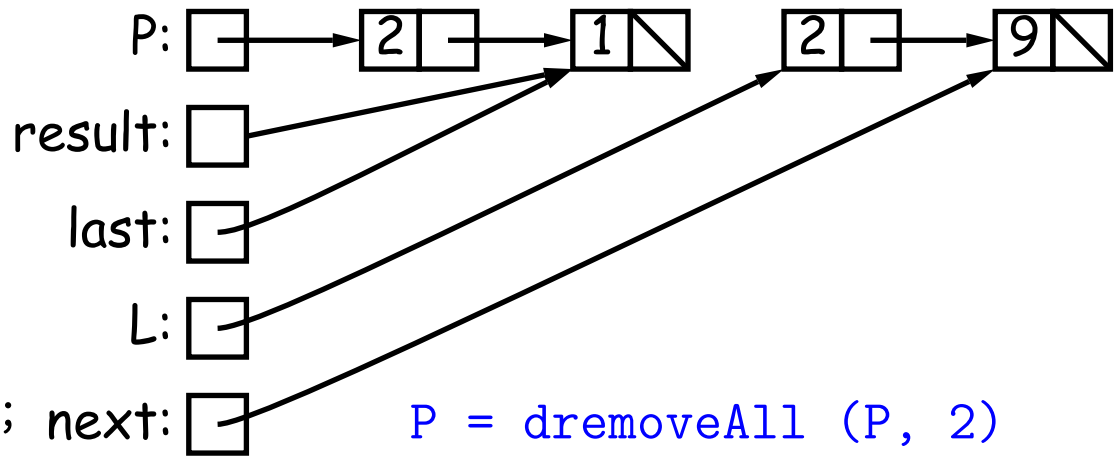
Iterative Destructive Deletion

```
/** 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;
}
```



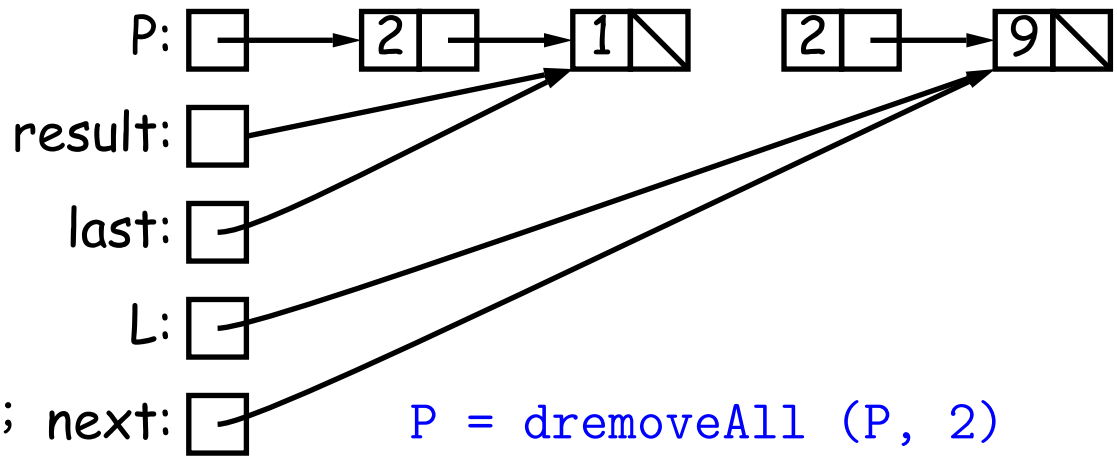
Iterative Destructive Deletion

```
/** 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;
}
```



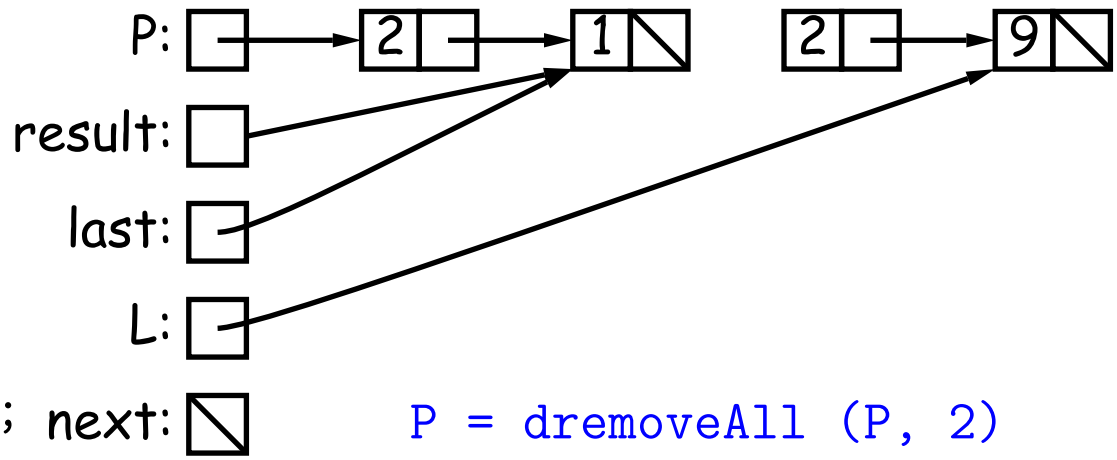
Iterative Destructive Deletion

```
/** 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;
}
```



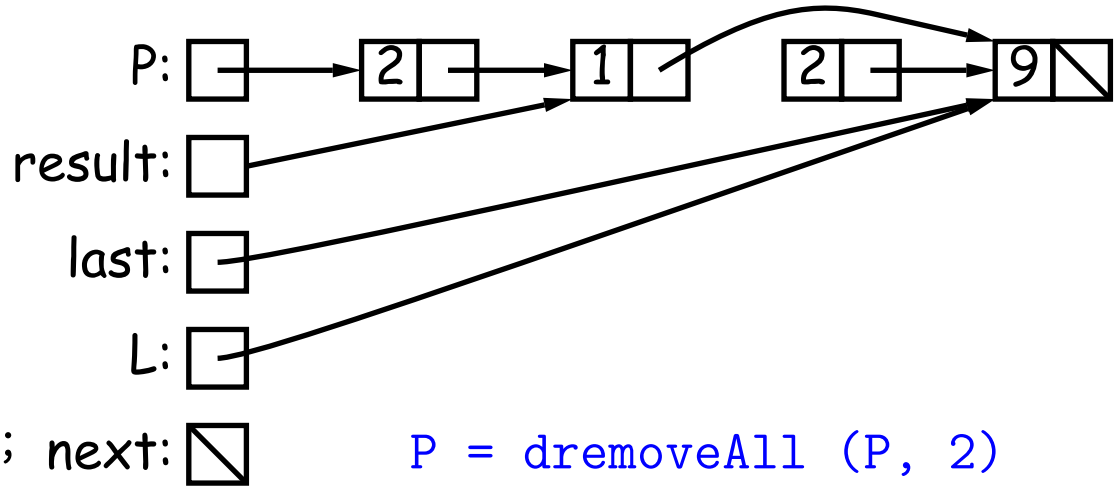
Iterative Destructive Deletion

```
/** 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;
}
```



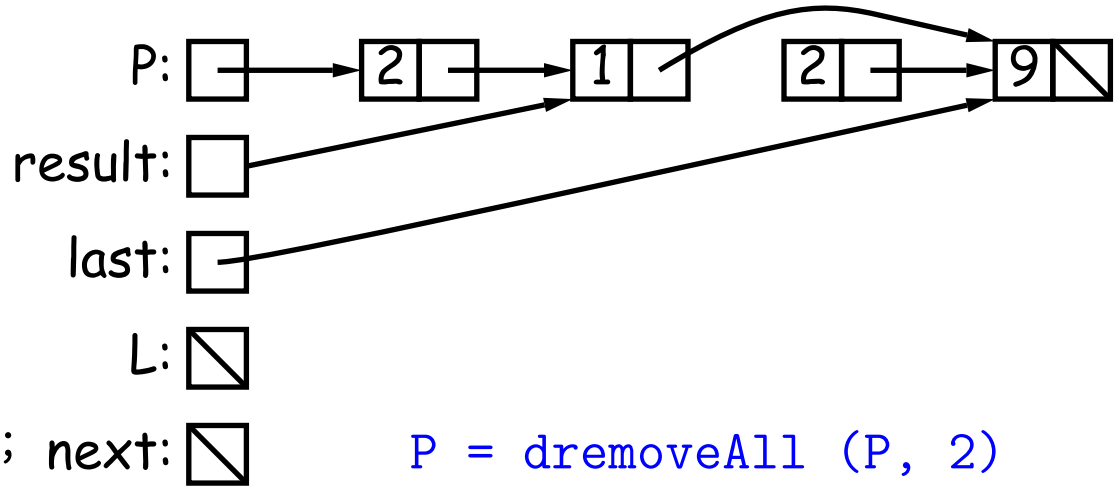
Iterative Destructive Deletion

```
/** 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;
}
```



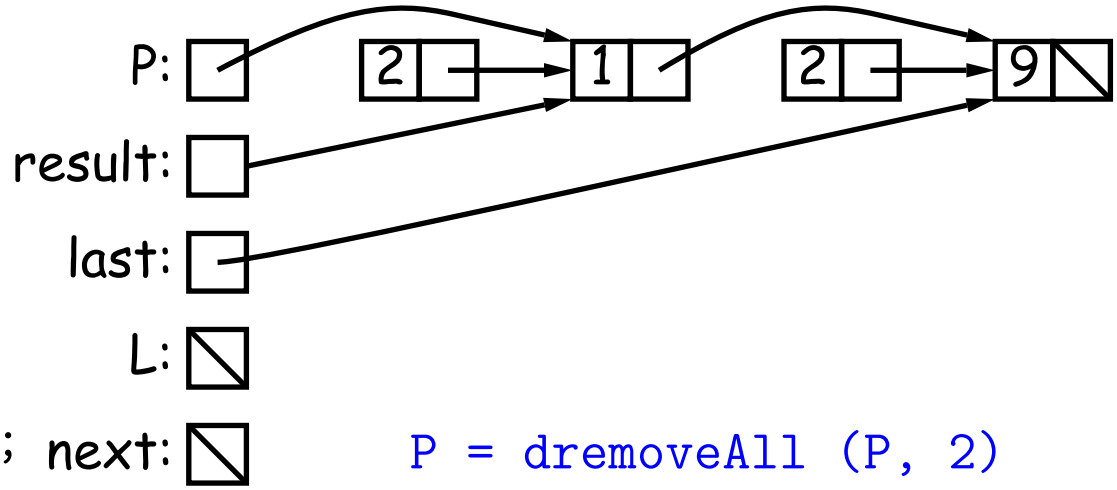
Iterative Destructive Deletion

```
/** 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;
}
```



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;
}
```



A Quick Jump Forward: What, No Functions?

- Project 0 contains an illustration of an interesting technique in Java having to do with the functions-as-values and higher-order functions that figured prominently in CS61A.
- In Java, there are no such things. For example, `dremoveAll` is not a "first-class value". It can only be used in the context of a function call: `dremoveAll(Q, 7)`.
- But despite the lack of functional values, Java can get the same effect by using another feature it does share with Python: instance methods of objects.
- We'll come back to this in detail later. For now, let's take a brief look ahead.

Functional Values

- In Python, we may write things like this:

```
def doall(L, action):
   """Apply the function F to all items in
      sequence L in order."""
   for x in L:
       acton(x)

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

- So `print` all by itself denotes a function that can be passed as a value, and called in `doAll` as a function.
- Likewise, `lambda x: ...` denotes an anonymous function that prints the concatenation of its argument with itself.
- Java does not allow these exactly.

An Alternative

- Python *also* allows another approach:

```
def doAll2(L, action):
    for x in L:
        action.accept(x)
class Printer1:
    def accept(self, y):
        print(y)
class Printer2:
    def accept(self, y):
        print(y + y)
doAll2(L, Printer1())
doAll2(L, Printer2())
```

- And Java *does* have classes and instance methods.

Java Version

- However, In Java (as usual) one must specify a good deal more information.
- In particular, you need to specify the type of `L` and `action`, and the types taken and returned by `accept`. For now, we'll just give you the "cookbook" version, and explain the details in later lectures.

```
class Something {
    static void doAll(List<String> L, Consumer<String> action) {
        for (String x : L) action.accept(x);
    }
}
class Printer1 implements Consumer<String> {
    public void accept(String y) { System.out.println(y); }
}
class Printer2 implements Consumer<String> {
    public void accept(String y) { System.out.println(y + y); }
}
```

- And to call `doAll`:

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

Consumer

- The type `Consumer` is not actually special; it's simply a generic library type (full name `java.util.function.Consumer` if you're curious.
- It defines a method called `accept`, and `Printer1` and `Printer2` are subtypes that override that method. We'll review what this all means later.
- We could, in fact, have defined our own class for this purpose, but why not take advantage of the library?
- We need this type because `doAll` needs a single type for its `action` parameter, but we have at least two different classes (`Printer1` and `Printer2`) that we want to pass to it.
- `Consumer` serves the same purpose as a base type in Python.
- If you've forgotten all that (or not seen it yet), don't worry; we can fill in the details later.

And Finally, Lambda Expressions

- As you can see, compared to a language such as Python, Java is just a bit...wordy: we have

```
class Printer2 implements Consumer<String> {
    public void accept(String y) { System.out.println(y + y); }
}
```

  together with

```
new Printer2()
```

  versus the original Python version:

```
lambda y: print(y + y)
```

- This was sufficiently annoying that the Java designers decided to introduce a convenient shorthand for the definition of classes like `Printer2` with

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

- There is a *lot* of language complexity involved in making it possible not to write the class definition or most of the `accept` method definition! For now, let's just be grateful that someone went to the trouble to work it out.