

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.
- Project 0 to be released tonight.

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 1

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 2

Initialization 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?

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 3

Initialization 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).

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 4

original list to save time or space:

Last modified: Fri Sep 6 15:32:48 2019

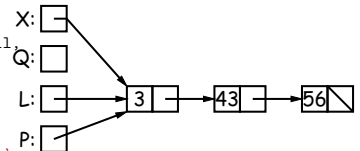
CS61B: Lecture #4 5

```
dincrlist(IntList L, int n)  
{  
    if (P == null) /* IntList.list from  
        return null; HW #1 */  
    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;  
}
```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 6

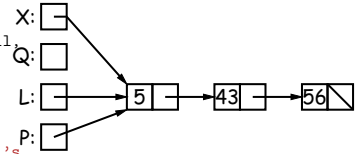


original list to save time or space:

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 7

```
dincrlist(IntList L, 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;
}
```



Last modified: Fri Sep 6 15:32:48 2019

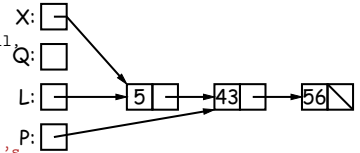
CS61B: Lecture #4 8

original list to save time or space:

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 9

```
dincrlist(IntList L, 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;
}
```



Last modified: Fri Sep 6 15:32:48 2019

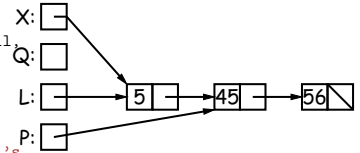
CS61B: Lecture #4 10

original list to save time or space:

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 11

```
dincrlist(IntList L, 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;
}
```



Last modified: Fri Sep 6 15:32:48 2019

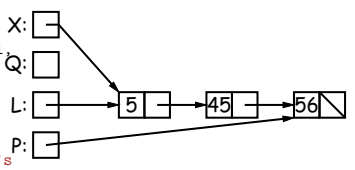
CS61B: Lecture #4 12

original list to save time or space:

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 13

```
dincrlist(IntList l, 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: 

Last modified: Fri Sep 6 15:32:48 2019

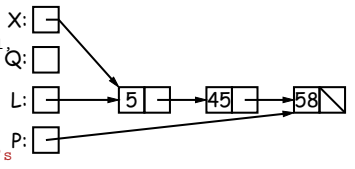
CS61B: Lecture #4 14

original list to save time or space:

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 15

```
dincrlist(IntList l, 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: 

Last modified: Fri Sep 6 15:32:48 2019

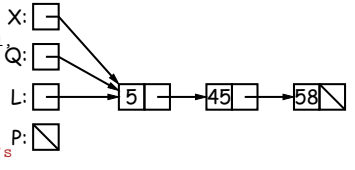
CS61B: Lecture #4 16

original list to save time or space:

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 17

```
dincrlist(IntList l, 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: 

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 18

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) )*/;
}
```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 19

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) )*/;
}
```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 20

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) )*/;
}
```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 21

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));
}
```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 22

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

```
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;
}
```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 23

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 24

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

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 25

```

non-destructively. */
static IntList removeAll(IntList
L, int x) {
    IntList result, last;
    result = last = null;
    for ( ; L != null; L = L.tail)
        removeAll (P, 2)
    {
        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;
}

```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 26

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

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 27

```

non-destructively. */
static IntList removeAll(IntList
L, int x) {
    IntList result, last;
    result = last = null;
    for ( ; L != null; L = L.tail)
        removeAll (P, 2)
    {
        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;
}

```

P does not change!

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 28

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

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 29

```

non-destructively. */
static IntList removeAll(IntList
L, int x) {
    IntList result, last;
    result = last = null;
    for ( ; L != null; L = L.tail)
        removeAll (P, 2)
    {
        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;
}

```

P does not change!

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 30

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

```

non destructively. */
static IntList removeAll(IntList
L, int x) {
    IntList result, last;
    result = last = null;
    for ( ; L != null; last = L.tail)
    {
        if (x == L.head)
            continue;
        else if (last == null)
            result = last = new
IntList(L.head, null);
        else
            last.tail = new
IntList(L.head, null);
    }
    return result;
}

```

Diagram illustrating the state of the list and pointers during the execution of the `removeAll` function. The list contains nodes with values 2, 1, 2, and 9. The `last` pointer is at the first node (2), and the `result` pointer is at the second node (1). The text "P does not change!" is shown in red.

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

```

non destructively. */
static IntList removeAll(IntList
L, int x) {
    IntList result, last;
    result = last = null;
    for ( ; L != null; last = L.tail)
        if (x == L.head)
            continue;
        else if (last == null)
            result = last = new
IntList(L.head, null);
        else
            last.tail = new
IntList(L.head, null);
    }
    return result;
}

```

Diagram illustrating the state of the list and pointers during the execution of the `removeAll` function. The list contains nodes with values 2, 1, 2, 9. The pointer `last` points to the first node (2), and the pointer `result` points to the second node (1). The text "P does not change!" is written in red.

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

```

non destructively. */
static IntList removeAll(IntList
L, int x) {
    IntList result, last;
    result = last = null;
    for ( ; L != null; last = L.tail)
        removeAll(P, 2)
    {
        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;
}

```

Diagram illustrating the state of the list and pointers during the execution of the `removeAll` function. The list contains nodes with values 2, 1, 2, and 9. The pointer `last` is shown pointing to the first node (value 2). The pointer `result` is shown pointing to the second node (value 1). The pointer `L` is shown pointing to the third node (value 2). The pointer `P` is shown pointing to the fourth node (value 9). The text "P does not change!" is written in red.

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

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 37

```

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;
}

```

Diagram illustrating the removal of all instances of 2 from the list [2, 1, 2, 9]. The original list is shown at the top. The result list is shown below, where the nodes containing 2 have been removed, leaving [1, 9]. The pointer 'last' is shown pointing to the node containing 1, and the pointer 'result' is shown pointing to the node containing 1. The pointer 'L' is shown pointing to the node containing 2, which is being skipped.

P does not change!

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 38

Q: [] → [1] → [2] → [3] → [1] → [1] → [0] → [1]

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

```

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

```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 39

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 40

Q: [] → [1] → [2] → [3] → [1] → [1] → [0] → [1]

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

```

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

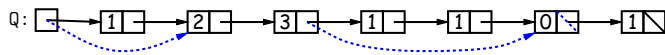
```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 41

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 42



/** 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;
    }
}

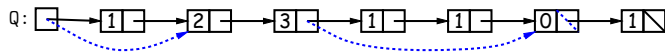
```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 43

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 44



/** 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;
    }
}

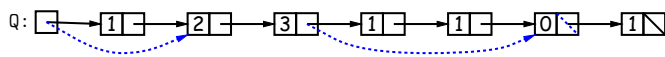
```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 45

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 46



/** 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;
    }
}

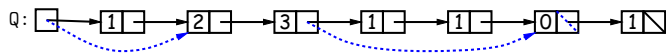
```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 47

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 48



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

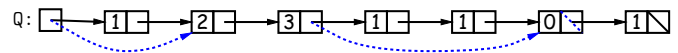
```

 * 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;
    }
}

```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 49



/** 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;
    }
}

```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 50

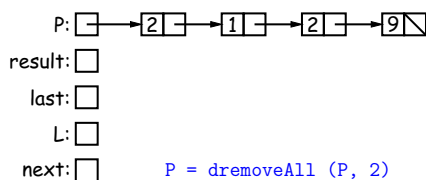
```

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;
}

```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 52



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

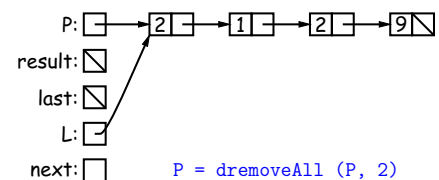
```

 * 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;
}

```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 53



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

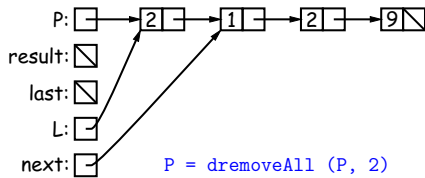
```

 * 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;
}

```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 54

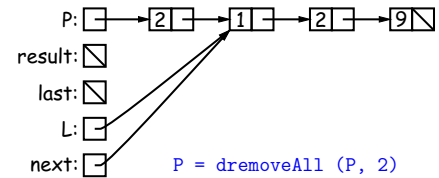


P = dremoveAll (P, 2)

/** 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;
        }
    }
}
```

Last modified: Fri Sep 15 15:32:48 2017 CS61B: Lecture #4 55

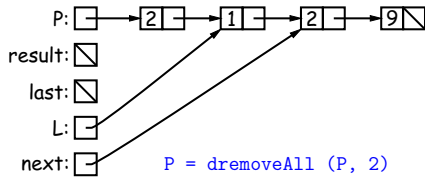


P = dremoveAll (P, 2)

/** 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;
        }
    }
}
```

Last modified: Fri Sep 15 15:32:48 2017 CS61B: Lecture #4 56

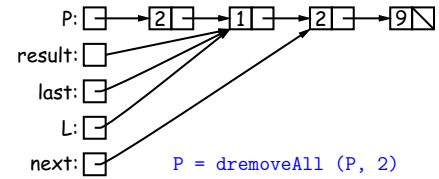


P = dremoveAll (P, 2)

/** 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;
        }
    }
}
```

Last modified: Fri Sep 15 15:32:48 2017 CS61B: Lecture #4 57

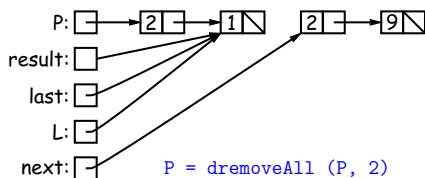


P = dremoveAll (P, 2)

/** 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;
        }
    }
}
```

Last modified: Fri Sep 15 15:32:48 2017 CS61B: Lecture #4 58

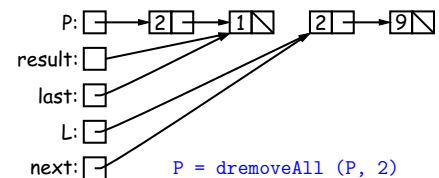


P = dremoveAll (P, 2)

/** 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;
        }
    }
}
```

Last modified: Fri Sep 15 15:32:48 2017 CS61B: Lecture #4 59

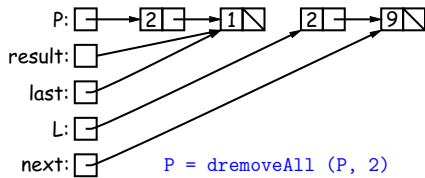


P = dremoveAll (P, 2)

/** 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;
        }
    }
}
```

Last modified: Fri Sep 15 15:32:48 2017 CS61B: Lecture #4 60

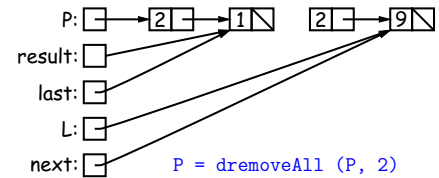


```

P = dremoveAll (P, 2)
/** 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;

```

Last modified: Fri Sep 15 15:32:49 2017 CS61B: Lecture #4 61

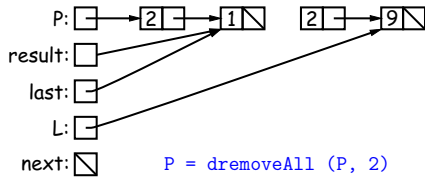


```

P = dremoveAll (P, 2)
/** 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;

```

Last modified: Fri Sep 15 15:32:49 2017 CS61B: Lecture #4 62

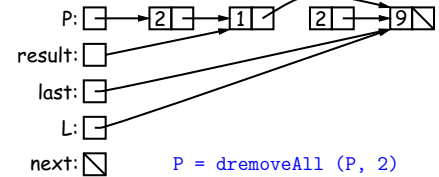


```

P = dremoveAll (P, 2)
/** 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;

```

Last modified: Fri Sep 15 15:32:49 2017 CS61B: Lecture #4 63

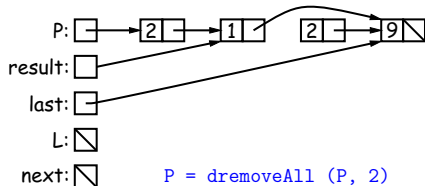


```

P = dremoveAll (P, 2)
/** 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;

```

Last modified: Fri Sep 15 15:32:49 2017 CS61B: Lecture #4 64

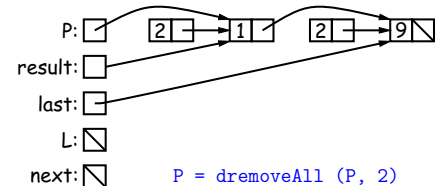


```

P = dremoveAll (P, 2)
/** 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;

```

Last modified: Fri Sep 15 15:32:49 2017 CS61B: Lecture #4 65



```

P = dremoveAll (P, 2)
/** 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;

```

Last modified: Fri Sep 15 15:32:49 2017 CS61B: Lecture #4 66

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

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

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 67

ever *Invariant* is true and *condition* false, our job is done!

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 68

equivalent recursive procedure:

```
/** Assuming Invariant, produce a situation
where Invariant
 * 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 false.

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 69

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 70

```
/** 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->tail = L;
      L = next;
    }
  }
  return result;
}
```

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 71

Last modified: Fri Sep 6 15:32:48 2019

CS61B: Lecture #4 72

- *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.