

Destructive Incrementing

Functions may modify objects in the original list to save

```
by 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);

    incrList(P.tail, n);

    by add N to L's items. */
    incrList(IntList L, int n)
    P: more than count!
    = L; p != null; p = p.tail!
}
```

11:51 2018

CS61B: Lecture #5 2

Destructive Incrementing

Functions may modify objects in the original list to save

```
by 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);

    incrList(P.tail, n);

    by add N to L's items. */
    incrList(IntList L, int n)
    P: more than count!
    = L; p != null; p = p.tail!
}
```

11:51 2018

CS61B: Lecture #5 4

Destructive Incrementing

Functions may modify objects in the original list to save

```
by 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);

    incrList(P.tail, n);

    by add N to L's items. */
    incrList(IntList L, int n)
    P: more than count!
    = L; p != null; p = p.tail!
}
```

11:51 2018

CS61B: Lecture #5 6

Lecture #5: Simple Pointer Manipulation

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

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

no pointer hacking.

Labs and homework: We'll be lenient about accepting work and labs for lab1, lab2, and hw0. Just get it done: the point is getting to understand the tools involved. We will accept submissions by email.

11:51 2018

CS61B: Lecture #5 1

Destructive Incrementing

Functions may modify objects in the original list to save

```
by 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);

    incrList(P.tail, n);

    by add N to L's items. */
    incrList(IntList L, int n)
    P: more than count!
    = L; p != null; p = p.tail!
}
```

11:51 2018

CS61B: Lecture #5 3

Destructive Incrementing

Functions may modify objects in the original list to save

```
by 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);

    incrList(P.tail, n);

    by add N to L's items. */
    incrList(IntList L, int n)
    P: more than count!
    = L; p != null; p = p.tail!
}
```

11:51 2018

CS61B: Lecture #5 5

Destructive Incrementing

Operations may modify objects in the original list to save

```
by 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);

    incrList(P.tail, n);
}

by add N to L's items. */
incrList(IntList L, int n) {
    // do more than count!
    L = L; p != null; p = p.tail)
}
```

The diagram shows four pointer variables: X, Q, L, and P. X points to a list of three nodes with values 3, 43, and 56. Q points to a list of three nodes with values 5, 45, and 58. L points to a list of three nodes with values 5, 45, and 58. P is null.

11:51 2018

CS61B: Lecture #5 8

Example: Non-destructive List Deletion

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

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

11:51 2018

CS61B: Lecture #5 10

Example: Non-destructive List Deletion

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

```
resulting from removing all instances of X from L
actively. */
removeAll(IntList L, int x) {
    // L
    // head == x
    // removeAll(L.tail, x);
    // new IntList(L.head, removeAll(L.tail, x));
}
```

11:51 2018

CS61B: Lecture #5 12

Destructive Incrementing

Operations may modify objects in the original list to save

```
by 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);

    incrList(P.tail, n);
}

by add N to L's items. */
incrList(IntList L, int n) {
    // do more than count!
    L = L; p != null; p = p.tail)
}
```

The diagram shows four pointer variables: X, Q, L, and P. X points to a list of three nodes with values 3, 43, and 56. Q points to a list of three nodes with values 5, 45, and 58. L points to a list of three nodes with values 5, 45, and 58. P points to a list of three nodes with values 5, 45, and 58.

11:51 2018

CS61B: Lecture #5 7

Example: Non-destructive List Deletion

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

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

11:51 2018

CS61B: Lecture #5 9

Example: Non-destructive List Deletion

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

```
resulting from removing all instances of X from L
actively. */
removeAll(IntList L, int x) {
    // L
    // head == x
    // removeAll(L.tail, x);
    // ( L with all x's removed (L!=null, L.head!=x) )*/;
}
```

11:51 2018

CS61B: Lecture #5 11

Active Non-destructive List Deletion

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

Alternating from removing all instances

non-destructively. */

```
removeAll(IntList L, int x) {
    P: [ ] → [2] → [1] → [2] → [9]
    L: [ ]
    result: [ ]
    last: [ ]
    removeAll(P, 2)
    last = new IntList(L.head, null);
    t.tail = new IntList(L.head, null);
}
```

1:51 2018

CS61B: Lecture #5 14

Active Non-destructive List Deletion

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

Alternating from removing all instances

non-destructively. */

```
removeAll(IntList L, int x) {
    P: [ ] → [2] → [1] → [2] → [9]
    L: [ ]
    result: [ ]
    last: [ ]
    removeAll(P, 2)
    last = new IntList(L.head, null);
    t.tail = new IntList(L.head, null);
}
```

1:51 2018

CS61B: Lecture #5 16

Active Non-destructive List Deletion

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

Alternating from removing all instances

non-destructively. */

```
removeAll(IntList L, int x) {
    P: [ ] → [2] → [1] → [2] → [9]
    L: [ ]
    result: [ ]
    last: [ ]
    removeAll(P, 2)
    last = new IntList(L.head, null);
    t.tail = new IntList(L.head, null);
}
```

1:51 2018

CS61B: Lecture #5 18

Active Non-destructive List Deletion

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

Alternating from removing all instances

non-destructively. */

```
removeAll(IntList L, int x) {
    P: [ ] → [2] → [1] → [2] → [9]
    L: [ ]
    result: [ ]
    last: [ ]
    removeAll(P, 2)
    last = new IntList(L.head, null);
    t.tail = new IntList(L.head, null);
}
```

1:51 2018

CS61B: Lecture #5 13

Active Non-destructive List Deletion

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

Alternating from removing all instances

non-destructively. */

```
removeAll(IntList L, int x) {
    P: [ ] → [2] → [1] → [2] → [9]
    L: [ ]
    result: [ ]
    last: [ ]
    removeAll(P, 2)
    last = new IntList(L.head, null);
    t.tail = new IntList(L.head, null);
}
```

1:51 2018

CS61B: Lecture #5 15

Active Non-destructive List Deletion

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

Alternating from removing all instances

non-destructively. */

```
removeAll(IntList L, int x) {
    P: [ ] → [2] → [1] → [2] → [9]
    L: [ ]
    result: [ ]
    last: [ ]
    removeAll(P, 2)
    last = new IntList(L.head, null);
    t.tail = new IntList(L.head, null);
}
```

1:51 2018

CS61B: Lecture #5 17

Active Non-destructive List Deletion

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

Resulting from removing all instances

non-destructively. */

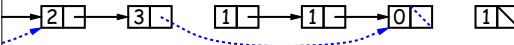
```
removeAll(IntList L, int x) {
    P: last;
    = null;
    all; L = L.tail) {
        L:
        head)
        result:
        last:
        removeAll(P, 2)
        P does not change!
    }
    L.tail = new IntList(L.head, null);
}
```

11:51 2018

CS61B: Lecture #5 20

Destructive Deletion

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



Resulting from removing all instances of X from L.

tail list may be destroyed. */

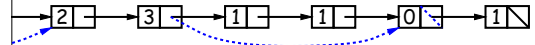
```
dremoveAll(IntList L, int x) {
    L:
    l)
    *( null with all x's removed )*/;
    head == x)
    *( L with all x's removed (L != null) )*/;
    remove all x's from L's tail. }*/;
}
```

11:51 2018

CS61B: Lecture #5 22

Destructive Deletion

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



Resulting from removing all instances of X from L.

tail list may be destroyed. */

```
dremoveAll(IntList L, int x) {
    L:
    l)
    *( null with all x's removed )*/;
    head == x)
    *( L with all x's removed (L != null) )*/;
    remove all x's from L's tail. }*/;
}
```

11:51 2018

CS61B: Lecture #5 24

Active Non-destructive List Deletion

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

Resulting from removing all instances

non-destructively. */

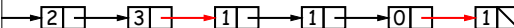
```
removeAll(IntList L, int x) {
    P: last;
    = null;
    all; L = L.tail) {
        L:
        head)
        result:
        last:
        removeAll(P, 2)
        P does not change!
    }
    L.tail = new IntList(L.head, null);
}
```

11:51 2018

CS61B: Lecture #5 19

Destructive Deletion

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



Resulting from removing all instances of X from L.

tail list may be destroyed. */

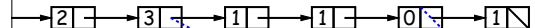
```
dremoveAll(IntList L, int x) {
    L:
    l)
    *( null with all x's removed )*/;
    head == x)
    *( L with all x's removed (L != null) )*/;
    remove all x's from L's tail. }*/;
}
```

11:51 2018

CS61B: Lecture #5 21

Destructive Deletion

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



Resulting from removing all instances of X from L.

tail list may be destroyed. */

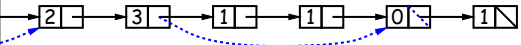
```
dremoveAll(IntList L, int x) {
    L:
    l)
    *( null with all x's removed )*/;
    head == x)
    *( L with all x's removed (L != null) )*/;
    remove all x's from L's tail. }*/;
}
```

11:51 2018

CS61B: Lecture #5 23

Destructive Deletion

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



resulting from removing all instances of X from L.
tail list may be destroyed. */

```

dremoveAll(IntList L, int x) {
  l;
  l == null) {
    return l;
  }
  if (l.head == x)
    dremoveAll(L.tail, x);
  l.tail = dremoveAll(L.tail, x);
}

```

Iterative Destructive Deletion

resulting from removing all X's from L

```

dremoveAll(IntList L, int x) {
  l, last;
  l == null) {
    return l;
  }
  l = l.head;
  if (l == x)
    l = l.next;
  last = l;
  l = l.next;
}

```

t;

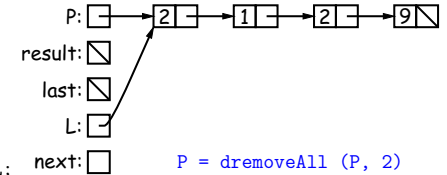
Iterative Destructive Deletion

resulting from removing all X's from L

```

dremoveAll(IntList L, int x) {
  l, last;
  l == null) {
    return l;
  }
  l = l.head;
  if (l == x)
    l = l.next;
  last = l;
  l = l.next;
}

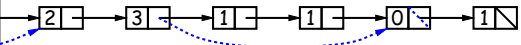
```



t;

Destructive Deletion

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



resulting from removing all instances of X from L.
tail list may be destroyed. */

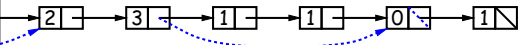
```

dremoveAll(IntList L, int x) {
  l;
  l == null) {
    return l;
  }
  if (l.head == x)
    dremoveAll(L.tail, x);
  l.tail = dremoveAll(L.tail, x);
}

```

Destructive Deletion

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



resulting from removing all instances of X from L.
tail list may be destroyed. */

```

dremoveAll(IntList L, int x) {
  l;
  l == null) {
    return l;
  }
  if (l.head == x)
    dremoveAll(L.tail, x);
  l.tail = dremoveAll(L.tail, x);
}

```

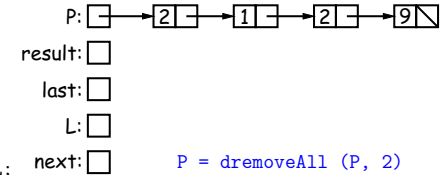
Iterative Destructive Deletion

resulting from removing all X's from L

```

dremoveAll(IntList L, int x) {
  l, last;
  l == null) {
    return l;
  }
  l = l.head;
  if (l == x)
    l = l.next;
  last = l;
  l = l.next;
}

```



t;

Iterative Destructive Deletion

resulting from removing all X's from L

```
rely. */
; dremoveAll(IntList L, int x) {
  lt, last;
  st = null;
  null) {
  xt = L.tail;
  .head) {
  == null)
  = last = L;
  = last.tail = L;
  = null;
  t;
```

P: [] → [2] → [1] → [2] → [9] → []

result: [] → [1]

last: [] → [2]

L: [] → [1]

next: [] → [2]

P = dremoveAll(P, 2)

Iterative Destructive Deletion

resulting from removing all X's from L

```
rely. */
; dremoveAll(IntList L, int x) {
  lt, last;
  st = null;
  null) {
  xt = L.tail;
  .head) {
  == null)
  = last = L;
  = last.tail = L;
  = null;
  t;
```

P: [] → [2] → [1] → [2] → [9] → []

result: [] → [1]

last: [] → [2]

L: [] → [1]

next: [] → [2]

P = dremoveAll(P, 2)

Iterative Destructive Deletion

resulting from removing all X's from L

```
rely. */
; dremoveAll(IntList L, int x) {
  lt, last;
  st = null;
  null) {
  xt = L.tail;
  .head) {
  == null)
  = last = L;
  = last.tail = L;
  = null;
  t;
```

P: [] → [2] → [1] → [2] → [9] → []

result: [] → [1]

last: [] → [2]

L: [] → [1]

next: [] → [2]

P = dremoveAll(P, 2)

Iterative Destructive Deletion

resulting from removing all X's from L

```
rely. */
; dremoveAll(IntList L, int x) {
  lt, last;
  st = null;
  null) {
  xt = L.tail;
  .head) {
  == null)
  = last = L;
  = last.tail = L;
  = null;
  t;
```

P: [] → [2] → [1] → [2] → [9] → []

result: [] → [1]

last: [] → [2]

L: [] → [1]

next: [] → [2]

P = dremoveAll(P, 2)

Iterative Destructive Deletion

resulting from removing all X's from L

```
rely. */
; dremoveAll(IntList L, int x) {
  lt, last;
  st = null;
  null) {
  xt = L.tail;
  .head) {
  == null)
  = last = L;
  = last.tail = L;
  = null;
  t;
```

P: [] → [2] → [1] → [2] → [9] → []

result: [] → [1]

last: [] → [2]

L: [] → [1]

next: [] → [2]

P = dremoveAll(P, 2)

Iterative Destructive Deletion

resulting from removing all X's from L

```
rely. */
; dremoveAll(IntList L, int x) {
  lt, last;
  st = null;
  null) {
  xt = L.tail;
  .head) {
  == null)
  = last = L;
  = last.tail = L;
  = null;
  t;
```

P: [] → [2] → [1] → [2] → [9] → []

result: [] → [1]

last: [] → [2]

L: [] → [1]

next: [] → [2]

P = dremoveAll(P, 2)

Iterative Destructive Deletion

resulting from removing all X's from L

```

1:  rely. */
2:  ; dremoveAll(IntList L, int x) {
3:  lt, last;
4:  st = null;
5:  null) {
6:  xt = L.tail;
7:  .head) {
8:  == null)
9:  = last = L;
10:
11: = last.tail = L;
12: = null;

```

P: [] → [2] → [1] → [] → [2] → [9] → []

result: [] → [1] → []

last: [] → [1] → []

L: [] → [2] → []

next: [] → []

P = dremoveAll (P, 2)

t;

Iterative Destructive Deletion

resulting from removing all X's from L

```

1:  rely. */
2:  ; dremoveAll(IntList L, int x) {
3:  lt, last;
4:  st = null;
5:  null) {
6:  xt = L.tail;
7:  .head) {
8:  == null)
9:  = last = L;
10:
11: = last.tail = L;
12: = null;

```

P: [] → [2] → [1] → [] → [2] → [9] → []

result: [] → [1] → []

last: [] → [1] → []

L: [] → [2] → []

next: [] → []

P = dremoveAll (P, 2)

t;

Iterative Destructive Deletion

resulting from removing all X's from L

```

1:  rely. */
2:  ; dremoveAll(IntList L, int x) {
3:  lt, last;
4:  st = null;
5:  null) {
6:  xt = L.tail;
7:  .head) {
8:  == null)
9:  = last = L;
10:
11: = last.tail = L;
12: = null;

```

P: [] → [2] → [1] → [] → [2] → [9] → []

result: [] → [1] → []

last: [] → [1] → []

L: [] → []

next: [] → []

P = dremoveAll (P, 2)

t;

Iterative Destructive Deletion

resulting from removing all X's from L

```

1:  rely. */
2:  ; dremoveAll(IntList L, int x) {
3:  lt, last;
4:  st = null;
5:  null) {
6:  xt = L.tail;
7:  .head) {
8:  == null)
9:  = last = L;
10:
11: = last.tail = L;
12: = null;

```

P: [] → [2] → [1] → [] → [2] → [9] → []

result: [] → [1] → []

last: [] → [1] → []

L: [] → [2] → []

next: [] → []

P = dremoveAll (P, 2)

t;

Iterative Destructive Deletion

resulting from removing all X's from L

```

1:  rely. */
2:  ; dremoveAll(IntList L, int x) {
3:  lt, last;
4:  st = null;
5:  null) {
6:  xt = L.tail;
7:  .head) {
8:  == null)
9:  = last = L;
10:
11: = last.tail = L;
12: = null;

```

P: [] → [2] → [1] → [] → [2] → [9] → []

result: [] → [1] → []

last: [] → [1] → []

L: [] → [2] → []

next: [] → []

P = dremoveAll (P, 2)

t;

Iterative Destructive Deletion

resulting from removing all X's from L

```

1:  rely. */
2:  ; dremoveAll(IntList L, int x) {
3:  lt, last;
4:  st = null;
5:  null) {
6:  xt = L.tail;
7:  .head) {
8:  == null)
9:  = last = L;
10:
11: = last.tail = L;
12: = null;

```

P: [] → [2] → [1] → [] → [2] → [9] → []

result: [] → [1] → []

last: [] → [1] → []

L: [] → []

next: [] → []

P = dremoveAll (P, 2)

t;

Example: Loop Invariant for dremoveAll

Starting from removing all X's from L

```
/*
removeAll(IntList L, int x) {
    result: [ ]
    last: [ ]
    L: [ ]
    P = dremoveAll (P, 2)
}
*/
```

```
graph LR
    P --> N1[2]
    N1 --> N2[1]
    N2 --> N3[2]
    N3 --> N4[9]
    result --> N1
    last --> N1
    L --> N1
    style N2 stroke-dasharray: 5 5
```

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.

2: How to Write a Loop (in Theory)

A description of how things look on *any arbitrary iteration*.

The invariant is known as a *loop invariant*, because it is always true at the start of each iteration.

When the loop then must

maintain any situation consistent with the invariant;

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

```
(condition) {
```

```
invariant true here
```

```
body
```

```
invariant again true here
```

```
invariant true and condition false.
```

The loop gets the desired answer whenever *invariant* is true

and if false, our job is done!