

Recreation

Prove that $\lfloor (2 + \sqrt{3})^n \rfloor$ is odd for all integer $n \geq 0$.

[Source: D. O. Shklarsky, N. N. Chentzov, I. M. Yaglom, *The USSR Olympiad Problem Book*, Dover ed. (1993), from the W. H. Freeman edition, 1962.]

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 1

CS61B Lecture #3: Values and Containers

- Labs are normally due at midnight Friday. Last week's is due tonight.
- **Today.** Simple classes. Scheme-like lists. Destructive vs. non-destructive operations. Models of memory.

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 2

Values and Containers

- **Values** are numbers, booleans, and pointers. **Values never change.**
3 'a' true \perp \ ↗

- **Simple containers** contain values:

x:

3

 l:

--

 p:

--

 ↗

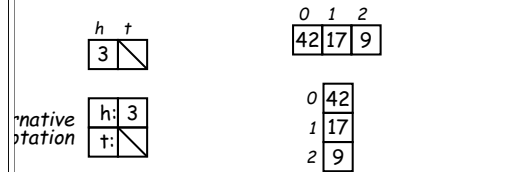
Examples: variables, fields, individual array elements, parameters.

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 3

Structured Containers

Structured containers contain (0 or more) other containers:
Class Object Array Object Empty Object

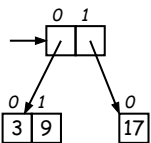


Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 4

Pointers

- **Pointers** (or **references**) are values that **reference** (point to) containers.
- One particular pointer, called **null**, points to nothing.
- In Java, structured containers contain only simple containers, but pointers allow us to build arbitrarily big or complex structures anyway.

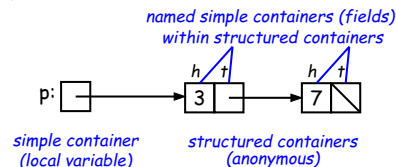


Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 5

Containers in Java

- Containers may be **named** or **anonymous**.
- In Java, **all** simple containers are named, **all** structured containers are anonymous, and pointers point only to structured containers. (Therefore, structured containers contain only simple containers).



- In Java, assignment copies values into simple containers.
- **Exactly** like Scheme and Python!
- (Python also has slice assignment, as in `x[3:7] = ...`, which is shorthand for something else entirely.)

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 6

Defining New Types of Object

- Class declarations introduce new types of objects.
- Example: list of integers:

```
public class IntList {
    // Constructor function (used to initialize new object)
    /** List cell containing (HEAD, TAIL). */
    public IntList(int head, IntList tail) {
        this.head = head; this.tail = tail;
    }

    // Names of simple containers (fields)
    // WARNING: public instance variables usually bad style!
    public int head;
    public IntList tail;
}
```

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 7

Primitive Operations

```
IntList Q, L;
```

L:

Q:

```
L = new IntList(3, null);
Q = L;
```

L:

Q:

```
Q = new IntList(42, null);
L.tail = Q;
```

L:

Q:

```
L.tail.head += 1;
// Now Q.head == 43
// and L.tail.head == 43
```

L:

Q:

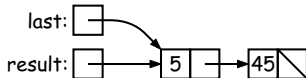
Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 8

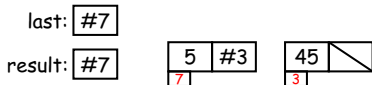
Side Excursion: Another Way to View Pointers

- Some folks find the idea of "copying an arrow" somewhat odd.
- Alternative view: think of a pointer as a *label*, like a street address.
- Each object has a permanent label on it, like the address plaque on a house.
- Then a variable containing a pointer is like a scrap of paper with a street address written on it.

- One view:



- Alternative view:



Last modified: Mon Sep 2 13:38:44 2019

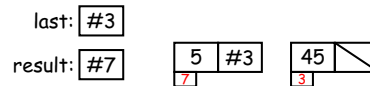
CS61B: Lecture #3 9

Another Way to View Pointers (II)

- Assigning a pointer to a variable looks just like assigning an integer to a variable.
- So, after executing "last = last.tail;" we have



- Alternative view:



- Under alternative view, you might be less inclined to think that assignment would change object #7 itself, rather than just "last".
- BEWARE! Internally, pointers really are just numbers, but Java treats them as more than that: they have *types*, and you can't just change integers into pointers.

Last modified: Mon Sep 2 13:38:44 2019

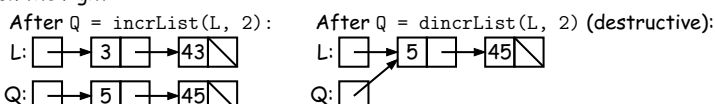
CS61B: Lecture #3 10

Destructive vs. Non-destructive

Problem: Given a (pointer to a) list of integers, *L*, and an integer increment *n*, return a list created by incrementing all elements of the list by *n*.

```
/** List of all items in P incremented by n. Does not modify
 * existing IntLists. */
static IntList incrList(IntList P, int n) {
    return /*( P, with each element incremented by n )*/
}
```

We say *incrList* is *non-destructive*, because it leaves the input objects unchanged, as shown on the left. A *destructive* method may modify the input objects, so that the original data is no longer available, as shown on the right:



Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 11

Nondestructive IncrList: Recursive

```
/** List of all items in P incremented by n. */
static IntList incrList(IntList P, int n) {
    if (P == null)
        return null;
    else return new IntList(P.head+n, incrList(P.tail, n));
}
```

- Why does *incrList* have to return its result, rather than just setting *P*?
- In the call *incrList(P, 2)*, where *P* contains 3 and 43, which *IntList* object gets created first?

Last modified: Mon Sep 2 13:38:44 2019

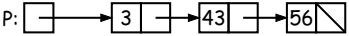
CS61B: Lecture #3 12

An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 13

P: 

```
static IntList incrList(IntList P, int n) {
    if (P == null) <<<
        return null;
    IntList result, last;
    result = last
        = new IntList(P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList(P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

Last modified: Mon Sep 2 13:38:44 2019

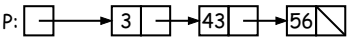
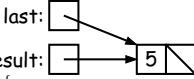
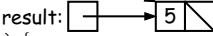
CS61B: Lecture #3 14

An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 15

P: 
last: 
result: 

```
static IntList incrList(IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList(P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList(P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

Last modified: Mon Sep 2 13:38:44 2019

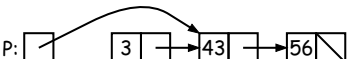
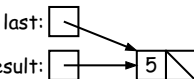
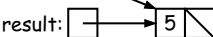
CS61B: Lecture #3 16

An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 17

P: 
last: 
result: 

```
static IntList incrList(IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList(P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList(P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

Last modified: Mon Sep 2 13:38:44 2019

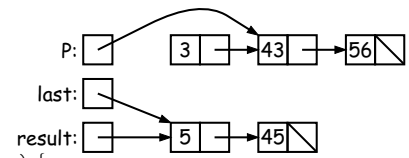
CS61B: Lecture #3 18

An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 19



```
static IntList incrList(IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList(P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList(P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

Last modified: Mon Sep 2 13:38:44 2019

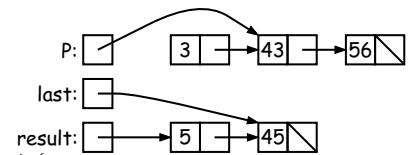
CS61B: Lecture #3 20

An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 21



```
static IntList incrList(IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList(P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList(P.head+n, null);
        last = last.tail; <<<
    }
    return result;
}
```

Last modified: Mon Sep 2 13:38:44 2019

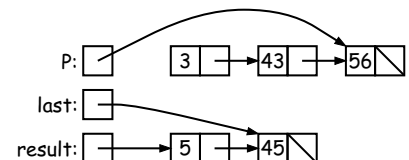
CS61B: Lecture #3 22

An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 23



```
static IntList incrList(IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList(P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList(P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

Last modified: Mon Sep 2 13:38:44 2019

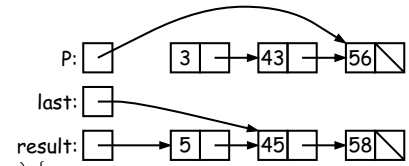
CS61B: Lecture #3 24

An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 25



```
static IntList incrList(IntList P, int n) {  
    if (P == null)  
        return null;  
    IntList result, last;  
    result = last  
        = new IntList(P.head+n, null);  
    while (P.tail != null) {  
        P = P.tail;  
        last.tail  
            = new IntList(P.head+n, null);  
        last = last.tail;  
    }  
    return result;  
}
```

Last modified: Mon Sep 2 13:38:44 2019

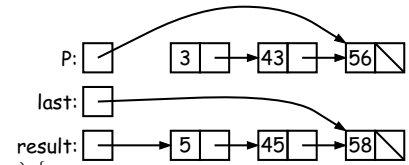
CS61B: Lecture #3 26

An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 27



```
static IntList incrList(IntList P, int n) {  
    if (P == null)  
        return null;  
    IntList result, last;  
    result = last  
        = new IntList(P.head+n, null);  
    while (P.tail != null) {  
        P = P.tail;  
        last.tail  
            = new IntList(P.head+n, null);  
        last = last.tail; <<<  
    }  
    return result;  
}
```

Last modified: Mon Sep 2 13:38:44 2019

CS61B: Lecture #3 28