Recreation

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

CS61B Lecture #3: Values and Containers

- Labs are normally due at midnight Friday. Last week’s is due tonight.
Values and Containers

- **Values** are numbers, booleans, and pointers. **Values never change.**

  3   'a'   true   \_   

- **Simple containers** contain values:

  x: 3   L:   p:   

Examples: variables, fields, individual array elements, parameters.
**Structured Containers**

*Structured containers* contain (0 or more) other containers:

<table>
<thead>
<tr>
<th>Class Object</th>
<th>Array Object</th>
<th>Empty Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Alternative Notation

- Class Object: `h: 3`, `t: 42, 17, 9`
- Array Object: `0: 42`, `1: 17`, `2: 9`
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.
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).

![Diagram of containers]

- In Java, assignment copies values into simple containers.
- Exactly like Scheme and Python!
- (Python also has slice assignment, as in \texttt{x[3:7]}=\ldots, which is shorthand for something else entirely.)
Defining New Types of Object

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

```java
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;
}
```

**Primitive Operations**

```java
IntList Q, L;

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

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

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

```
  last:               result:
                   5   45
```

• Alternative view:

```
  last: #7               result: #7
       5   #3            45
       7       3
```
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

  last: [graph]

  result: [graph]

- Alternative view:

  last: #3

  result: #7

  [graph]

- 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.
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 $n$.

```c
/** 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:

**After** $Q = incrList(L, 2)$:

<table>
<thead>
<tr>
<th>L:</th>
<th>Q:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

**After** $Q = dincrList(L, 2)$ (destructive):

<table>
<thead>
<tr>
<th>L:</th>
<th>Q:</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>
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?
An Iterative Version

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

```java
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;
}
```

![Diagram of list incrementation](image-url)
An Iterative Version

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

```
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;
}
```
An Iterative Version

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

```java
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: Wed Jan 29 13:26:29 2020
An Iterative Version

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

```java
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: Wed Jan 29 13:26:29 2020
An Iterative Version

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

```java
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;
}
```

P: 3 43 56
last:
result: 5 45

Last modified: Wed Jan 29 13:26:29 2020
An Iterative Version

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

```java
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: Wed Jan 29 13:26:29 2020

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:

```java
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: Wed Jan 29 13:26:29 2020
An Iterative Version

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

```java
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;
}
```

![Diagram showing the iterative version of incrList](image)

Last modified: Wed Jan 29 13:26:29 2020