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 lab, however, is due this coming Friday at midnight.

- **Today.** Simple classes. Scheme-like lists. Destructive vs. non-destructive operations. Models of memory.
Values and Containers

- **Values** are numbers, booleans, and pointers. **Values never change.** (So, for example, the assignment 3 = 2 would be invalid.)

  \[ 3 \quad 'a' \quad \text{true} \]

- **Simple containers** contain values:

  \[ x: 3 \quad L: \quad p: \]

Examples: variables, fields, individual array elements, parameters. The *contents* of containers can change.
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>( h ) ( t ) ( 3 )</td>
<td>0 1 2 42 17 9</td>
<td>[ ]</td>
</tr>
<tr>
<td><strong>Alternative Notation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h : 3 ) ( t : )</td>
<td>0 42 1</td>
<td>1 17 2 9</td>
</tr>
</tbody>
</table>
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).

```
    named simple containers (fields)
within structured containers

  p: h t

    simple container (local variable)    structured containers (anonymous)
```

- 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.)
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

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
```
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:

result: 5 -> 45
```

- Alternative view:

```
last: #3

result: #7 -> 5 -> #3 -> 45
```

- 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** \( Q = \text{incrList}(L, 2) \):
  - \( L: \) 3 → 43
  - \( Q: \) 5 → 45
- **After** \( Q = \text{dincrList}(L, 2) \) (destructive):
  - \( L: \) 5 → 45
  - \( Q: \)
Nondestructive IncrList: Recursive

```java
/** 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));
    }
}
```

- In the call `incrList(P, 2)`, where P contains 3 and 43, which IntList object gets created first?
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));
    }
}

• In the call incrList(P, 2), where P contains 3 and 43, which IntList object gets created first?

• Answer: The last one.
Nondestructive IncrList: Why Return the Value?

• If I want to update \( Q \) to an incremented list, why must I write

\[
Q = \text{incrList}(Q, 4); \\
\]

• Couldn't I instead just write

\[
\text{incrList2}(Q, 4);
\]

and define

```java
/** List of all items in P incremented by n. */
static IntList incrList2(IntList P, int n) {
    if (P == null) {
        P = null;
    } else {
        P = new IntList(P.head+n, incrList2(P.tail, n));
    }
    return P;
}
```
Nondestructive IncrList: Why Return the Value?

• If I want to update Q to an incremented list, why must I write
  
  \[ Q = \text{incrList}(Q, 4); \]

• Couldn’t I instead just write
  
  \[ \text{incrList2}(Q, 4); \]

  and define

  ```java
  /** List of all items in P incremented by n. */
  static IntList incrList2(IntList P, int n) {
      if (P == null) {
          P = null;
      } else {
          P = new IntList(P.head+n, incrList2(P.tail, n));
      }
      return P;
  }
  ```

• No. Assigning to the formal parameter does not affect the actual. Java uses call by value, just like Python.
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;

    return result;
}
```

```
P:   3   43   56
```
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;

    return result;
}
```

![Diagram of list operations](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:

```java
static IntList incrList(IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;

    return result;
}
```

Last modified: Mon Jan 24 00:22:33 2022
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;

    return result;
}
```

![Diagram showing the process of adding a new value to the list iteratively, with nodes labeled P, last, and 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
class IntList {
    // ... constructors, methods...
}

static IntList incrList(IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;

    return result;
}
```

```
P: 3 → 43 → 56
last: 5 → 45
result: 5 → 45
```
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;

    return result;
}
```

![Diagram](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:

```java
static IntList incrList(IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;

    return result;
}
```

![Diagram showing the iterative process]
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;

    return result;
}
```

![Diagram showing iterative incrList](image)
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 modified: Mon Jan 24 00:22:33 2022
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;
}
```
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: []
result: [5]
```
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: Mon Jan 24 00:22:33 2022
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: Mon Jan 24 00:22:33 2022

CS61B: Lecture #3
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;
}
```

![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;
}
```