Pointers

Discussion 3
Announcements

- HW 0, Lab 1, and Lab 2 due 1/31 - you may use slip days on them if needed
- HW 1 due 2/1
- Weekly Surveys are worth points + due every Monday
- Topical Review Session on Java this Friday 2-3:30 PM
Values & Containers

Simple Containers are named and may contain values or pointers to structured containers. Structured Containers are anonymous and contain simple containers or objects.

Values are numbers, booleans, and pointers and cannot be modified without being replaced.

- **Numbers** → Numbers as we know them (byte, short, int, double, long, float)
- **Letters** → Characters (char)
- **Booleans** → True or False (bool)
- **Pointers** → Memory address to a spot in memory where a structured container is stored
- **Null** → Nothing

![Diagram of Cat with id 2 and age 5]
**Linked Lists & Arrays**

**Linked Lists** are data structures that consist of structured containers, each containing two simple containers.
- `list.head` holds a value
- `list.tail` stores a pointer to the next structured container

![Linked List Diagram]

**Arrays** are data structures which can hold many simple containers of the same type of value.
- `arr[i]` holds a value in the `i`th position of the array

![Array Diagram]
Destructive & Non-Destructive Operations

Java is **pass-by-value**, so you are passing in a copy of the value of the variable.

**Destructive** functions alter the structured container or object passed in, causing changes to remain even after we leave the function (i.e. `x[1] = 5`)

**Non-Destructive** functions don’t alter the structured contained passed in (i.e. `x = new int[] {5, 10}`)

```java
private static void f(int[] x) {
    // ... 
}

f(A)
```
1A Boxes and Pointers

1. IntList L = IntList.list(1, 2, 3, 4);
2. IntList M = L.tail.tail;
3. IntList N = IntList.list(5, 6, 7);
4. N.tail.tail.tail = N;
5. L.tail.tail = N.tail.tail.tail.tail;
6. M.tail.tail = L;

What does the final box and pointer diagram look like?
1A Boxes and Pointers

```java
1 IntList L = IntList.list(1, 2, 3, 4);
2 IntList M = L.tail.tail;
3 IntList N = IntList.list(5, 6, 7);
4 N.tail.tail.tail = N;
5 L.tail.tail = N.tail.tail.tail.tail;
6 M.tail.tail = L;
```

What does the final box and pointer diagram look like?
1A Boxes and Pointers

1. `IntList L = IntList.list(1, 2, 3, 4);`
2. `IntList M = L.tail.tail;`
3. `IntList N = IntList.list(5, 6, 7);`
4. `N.tail.tail.tail = N;`
5. `L.tail.tail = N.tail.tail.tail.tail;`
6. `M.tail.tail = L;`

What does the final box and pointer diagram look like?
1A Boxes and Pointers

```java
IntList L = IntList.list(1, 2, 3, 4);
IntList M = L.tail.tail;
IntList N = IntList.list(5, 6, 7);
N.tail.tail.tail = N;
L.tail.tail = N.tail.tail.tail.tail;
M.tail.tail = L;
```

What does the final box and pointer diagram look like?
1A Boxes and Pointers

1 IntList L = IntList.list(1, 2, 3, 4);
2 IntList M = L.tail.tail;
3 IntList N = IntList.list(5, 6, 7);
4 N.tail.tail.tail = N;
5 L.tail.tail = N.tail.tail.tail.tail;
6 M.tail.tail = L;

What does the final box and pointer diagram look like?

![Diagram of boxes and pointers]
IntList L = IntList.list(1, 2, 3, 4);
IntList M = L.tail.tail;
IntList N = IntList.list(5, 6, 7);
N.tail.tail.tail = N;
L.tail.tail = N.tail.tail.tail.tail;
M.tail.tail = L;

What does the final box and pointer diagram look like?
1A Boxes and Pointers

```java
IntList L = IntList.list(1, 2, 3, 4);
IntList M = L.tail.tail;
IntList N = IntList.list(5, 6, 7);
N.tail.tail.tail = N;
L.tail.tail = N.tail.tail.tail.tail;
M.tail.tail = L;
```

What does the final box and pointer diagram look like?
IntList L1 = IntList.list(1, 2, 3);
IntList L2 = new IntList(4, L1.tail);
L2.tail.head = 13;
L1.tail.tail.tail = L2;
IntList L3 = IntList.list(50);
L2.tail.tail = L3;

What does the final box and pointer diagram look like?
1B Boxes and Pointers *Extra*

1. `IntList L1 = IntList.list(1, 2, 3);`
2. `IntList L2 = new IntList(4, L1.tail);`
3. `L2.tail.head = 13;`
4. `L1.tail.tail.tail = L2;`
5. `IntList L3 = IntList.list(50);`
6. `L2.tail.tail = L3;`

**What does the final box and pointer diagram look like?**

![Diagram](attachment:image.png)
IntList L1 = IntList.list(1, 2, 3);
IntList L2 = new IntList(4, L1.tail);
L2.tail.head = 13;
L1.tail.tail.tail = L2;
IntList L3 = IntList.list(50);
L2.tail.tail = L3;

What does the final box and pointer diagram look like?
1B Boxes and Pointers *Extra*

```java
IntList L1 = IntList.list(1, 2, 3);
IntList L2 = new IntList(4, L1.tail);
L2.tail.head = 13;
L1.tail.tail.tail = L2;
IntList L3 = IntList.list(50);
L2.tail.tail = L3;
```

What does the final box and pointer diagram look like?
1B Boxes and Pointers Extra

1 \( \text{IntList L1 = IntList.list(1, 2, 3);} \)
2 \( \text{IntList L2 = new IntList(4, L1.tail);} \)
3 \( \text{L2.tail.head = 13;} \)
4 \( \text{L1.tail.tail.tail = L2;} \)
5 \( \text{IntList L3 = IntList.list(50);} \)
6 \( \text{L2.tail.tail = L3;} \)

What does the final box and pointer diagram look like?
IntList L1 = IntList.list(1, 2, 3);
IntList L2 = new IntList(4, L1.tail);
L2.tail.head = 13;
L1.tail.tail.tail = L2;
IntList L3 = IntList.list(50);
L2.tail.tail = L3;

What does the final box and pointer diagram look like?
2 Destructive or Non-Destructive?

```java
public static int getHead(IntList L) {
    int listHead = L.head;
    L = new IntList(5, null);
    return listHead;
}
```

**Is the method destructive or non-destructive? Why?**

Non-destructive - the input list itself is never modified.
3A Reversing a List

Implement reverseNondestructive such that it returns a new list with all the elements of L in reverse order.

```java
public static IntList reverseNondestructive (IntList L) {
}
```
3A Reversing a List

Implement reverseNondestructive such that it returns a new list with all the elements of L in reverse order.

```java
public static IntList reverseNondestructive (IntList L) {
    IntList returnList = null; // We need a new list since we aren't modifying the old one
}
```
3A Reversing a List

Implement reverseNondestructive such that it returns a new list with all the elements of L in reverse order.

```java
public static IntList reverseNondestructive (IntList L) {
    IntList returnList = null; // We need a new list since we aren't modifying the old one
    }
```
3A Reversing a List

Implement `reverseNondestructive` such that it returns a new list with all the elements of `L` in reverse order.

```java
public static IntList reverseNondestructive (IntList L) {
    IntList returnList = null;
}
```

}  // We can’t just traverse `L` backwards since it’s a singly linked list...
// So how can we possibly get the elements in reverse?
3A Reversing a List

Implement reverseNondestructive such that it returns a new list with all the elements of L in reverse order.

```java
public static IntList reverseNondestructive (IntList L) {
    IntList returnList = null;
    // What if we build our list backwards?
    return
```

// What if we build our list backwards?

L → □ → □

return □
3A Reversing a List

Implement `reverseNondestructive` such that it returns a new list with all the elements of `L` in reverse order.

```java
public static IntList reverseNondestructive (IntList L) {
    IntList returnList = null;
}

// So we insert the elements into the front instead of the back!
```

L

return
3A Reversing a List

Implement reverseNondestructive such that it returns a new list with all the elements of L in reverse order.

```java
public static IntList reverseNondestructive (IntList L) {
    IntList returnList = null;
    while (L != null) {
        // Check to make sure we haven’t run out of list
        returnList = new IntList(L.head, returnList); // Insert into the front
        L = L.tail; // Move the pointer to the next item in line
    }
}
```
3A Reversing a List

Implement reverseNondestructive such that it returns a new list with all the elements of L in reverse order.

```java
public static IntList reverseNondestructive (IntList L) {
    IntList returnList = null;
    while (L != null) {
        returnList = new IntList(L.head, returnList);
        L = L.tail;
    }
    return returnList; // Finally, return our new, populated list
}
```
Implement `reverseNondestructive` such that it returns a new list with all the elements of \( L \) in reverse order.

```java
class IntList {
    IntList head;
    IntList tail;
}

class IntList {
    IntList head;
    IntList tail;
}

public static IntList reverseNondestructive (IntList L) {
    IntList returnList = null;
    while (L != null) {
        returnList = new IntList(L.head, returnList);
        L = L.tail;
    }
    return returnList;
}
```
3B Reversing a List *Extra*

Implement `reverseDestructive` such that it destructively reverses the elements in `L`.

```java
public static IntList reverseDestructive (IntList L) {
}
```
3B Reversing a List Extra

Implement reverseDestructive such that it destructively reverses the elements in L.

```java
public static IntList reverseDestructive (IntList L) {
}

// Since this one is destructive, let’s try something recursive
```
Implement `reverseDestructive` such that it destructively reverses the elements in L.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null) { // First step: base case
        return L; // If the list is null, there is nothing to reverse
    }
}
```
3B Reversing a List *Extra*

Implement `reverseDestructive` such that it destructively reverses the elements in `L`.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null) {
        return L;
    }
    // How do we approach this?
}
```
3B Reversing a List *Extra*

Implement `reverseDestructive` such that it destructively reverses the elements in `L`.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null) {
        return L;
    }
}
```

// Let’s assume we already have a method already that can reverse the rest of the list
// Where would this go in relation to the first element?
Implement reverseDestructive such that it destructively reverses the elements in L.

public static IntList reverseDestructive (IntList L) {
    if (L == null) {
        return L;
    }
    // Let’s assume we already have a method already that can reverse the rest of the list
    // Where would this go in relation to the first element?
    // Before it!
3B Reversing a List *Extra*

Implement `reverseDestructive` such that it destructively reverses the elements in `L`.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null) {
        return L;
    }
}
// We insert the reversed “rest” of the list ahead of our first element...
// And the full list would be reversed!
```
3B Reversing a List *Extra*

Implement `reverseDestructive` such that it destructively reverses the elements in `L`.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null) {
        return L;
    } else {
        IntList reversed = reverseDestructive(L.tail); // Assume that this works
    }
}
```
Reversing a List \textit{Extra}

Implement \texttt{reverseDestructive} such that it destructively reverses the elements in \texttt{L}.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null) {
        return L;
    } else {
        IntList reversed = reverseDestructive(L.tail);
    }
    IntList reversed = reverseDestructive(L.tail);
}

// We need to be careful that all the pointers are taken care of
```

```java
L \rightarrow \rightarrow \rightarrow
```
Implement reverseDestructive such that it destructively reverses the elements in L.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null) {
        return L;
    } else {
        IntList reversed = reverseDestructive(L.tail);

        // If reverseDestructive was successfully called on L.tail here, then we would get
        // Notice that the first element still points at what used to be after it
```

L
```
```
3B Reversing a List Extra

Implement `reverseDestructive` such that it destructively reverses the elements in `L`.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null) {
        return L;
    } else {
        IntList reversed = reverseDestructive(L.tail);
    }
    // Before we change that pointer, we need to make that element point at our first element
    L
    // Since our element will now come after it!
```

CS 61B // Spring 2022
3B Reversing a List Extra

Implement reverseDestructive such that it destructively reverses the elements in L.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null) {
        return L;
    } else {
        IntList reversed = reverseDestructive(L.tail);
        L.tail.tail = L; // Points “next” element back at “current” element
    }
}
```

// That’s one thing taken care of

L

---

CS 61B // Spring 2022
3B Reversing a List Extra

Implement reverseDestructive such that it destructively reverses the elements in L.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null) {
        return L;
    } else {
        IntList reversed = reverseDestructive(L.tail);
        L.tail.tail = L;
        L.tail = null; // Just in case our node is the last one
    }
}
```

// Now we need to get rid of that old pointer since it doesn’t make sense to keep it
Implement reverseDestructive such that it destructively reverses the elements in L.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null) {
        return L;
    } else {
        IntList reversed = reverseDestructive(L.tail);
        L.tail.tail = L;
        L.tail = null;
        return reversed; // Done! Just have to return it
    }
}
```
3B Reversing a List *Extra*

Implement `reverseDestructive` such that it destructively reverses the elements in `L`.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null) { // Slight problem: if we wait until L is null to end
        return L;
    } else {
        IntList reversed = reverseDestructive(L.tail);
        L.tail.tail = L; // We risk trying to call .tail on null, which would error
        L.tail = null;
        return reversed;
    }
}
```
3B Reversing a List *Extra*

Implement `reverseDestructive` such that it destructively reverses the elements in `L`.

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null || L.tail == null) { // Easy fix
        return L; // If it's the only element, the reverse is the same anyway
    } else {
        IntList reversed = reverseDestructive(L.tail);
        L.tail.tail = L;
        L.tail = null;
        return reversed;
    }
}
```
3B Reversing a List Extra

Implement `reverseDestructive` such that it destructively reverses the elements in \( L \).

```java
public static IntList reverseDestructive (IntList L) {
    if (L == null || L.tail == null) {
        return L;
    } else {
        IntList reversed = reverseDestructive(L.tail);
        L.tail.tail = L;
        L.tail = null;
        return reversed;
    }
}
```
4A Inserting into a Linked List

Implement `insertRecursive` such that it inserts an element `item` at position `position` in the original list.

```java
public static IntList insertRecursive (IntList L, int item, int position) {
    // Implementation goes here...
}
```
4A Inserting into a Linked List

Implement `insertRecursive` such that it inserts an element `item` at position `position` in the original list.

```java
public static IntList insertRecursive (IntList L, int item, int position) {
    // This can be approached recursively since we have a position and list input
```
4A Inserting into a Linked List

Implement `insertRecursive` such that it inserts an element `item` at position `position` in the original list.

```java
public static IntList insertRecursive (IntList L, int item, int position) {
    if (L == null) {
        // Always step one: base case
        return new IntList(item, L); // If the list is empty, item becomes the list
    }
}
```
Implement `insertRecursive` such that it inserts an element `item` at position `position` in the original list.

```java
class IntList {
    int head;
    IntList tail;

    IntList(int item) {
        this.head = this.tail = item;
    }
}

public static IntList insertRecursive(IntList L, int item, int position) {
    if (L == null) {
        return new IntList(item, L);
    }
    // Two options for next step: we are either where we need to insert
    // or we are not
}
```
4A Inserting into a Linked List

Implement `insertRecursive` such that it inserts an element item at position position in the original list.

```java
public static IntList insertRecursive (IntList L, int item, int position) {
    if (L == null) {
        return new IntList(item, L);
    }
    if (position == 0) {
        // If we are where we need to insert, we adjust the pointers
        return new IntList(item, L);
    }
    // But we don’t have access to the previous pointer so we need to get tricky
    // ...}
```
Implement `insertRecursive` such that it inserts an element `item` at position `position` in the original list.

```java
public static IntList insertRecursive (IntList L, int item, int position) {
    if (L == null) {
        return new IntList(item, L);
    }
    if (position == 0) {
        L.tail = new IntList(L.head, L.tail); // Let’s create a copy of the node at that
        // position currently and set that as the next node
    }
}
```
4A Inserting into a Linked List

Implement `insertRecursive` such that it inserts an element `item` at position `position` in the original list.

```java
public static IntList insertRecursive (IntList L, int item, int position) {
    if (L == null) {
        return new IntList(item, L);
    }
    if (position == 0) {
        L.tail = new IntList(L.head, L.tail);
        L.head = item; // Now we can change the old node to have our new value
    }
}
```
Implement `insertRecursive` such that it inserts an element `item` at position `position` in the original list.

```java
public static IntList insertRecursive (IntList L, int item, int position) {
    if (L == null) {
        return new IntList(item, L);
    }
    if (position == 0) {
        L.tail = new IntList(L.head, L.tail);
        L.head = item;
    } else { // In the other case we make our recursive call
        L.tail = insertRecursive(L.tail, item, position - 1); // Moving along the list
    }
}
4A Inserting into a Linked List

Implement `insertRecursive` such that it inserts an element item at position position in the original list.

```java
public static IntList insertRecursive (IntList L, int item, int position) {
    if (L == null) {
        return new IntList(item, L);
    }
    if (position == 0) {
        L.tail = new IntList(L.head, L.tail);
        L.head = item;
    } else {
        L.tail = insertRecursive(L.tail, item, position - 1);
    }
    return L; // Finally, return the list
}
```
Inserting into a Linked List

Implement `insertRecursive` such that it inserts an element `item` at position `position` in the original list.

```java
public static IntList insertRecursive (IntList L, int item, int position) {
    if (L == null) {
        return new IntList(item, L);
    }
    if (position == 0) {
        L.tail = new IntList(L.head, L.tail);
        L.head = item;
    } else {
        L.tail = insertRecursive(L.tail, item, position - 1);
    }
    return L;
}
```
4B Inserting into a Linked List *Extra*

Implement `insertIterative` such that it inserts an element `item` at position `position` in the original list.

```java
public static IntList insertIterative (IntList L, int item, int position) {
```
4B Inserting into a Linked List \textit{Extra}

Implement \texttt{insertIterative} such that it inserts an element \texttt{item} at position \texttt{position} in the original list.

```java
public static IntList insertIterative (IntList L, int item, int position) {
    if (L == null) { // The general framework is the same as the recursive version
        return new IntList(item, L);
    }
    if (position == 0) {
        L.tail = new IntList(L.head, L.tail);
        L.head = item;
    } else { // The big change is in replacing the recursive step with a loop
        ...
    }
    return L;
}
```
Implement `insertIterative` such that it inserts an element `item` at position `position` in the original list.

```java
public static IntList insertIterative (IntList L, int item, int position) {
    if (L == null) {
        return new IntList(item, L);
    } else {
        IntList current = L; // We need a new pointer to iterate through since we need to return L
        if (position == 0) {
            L.tail = new IntList(L.head, L.tail);
            L.head = item;
        }
        return L;
    }
}
```
Implement `insertIterative` such that it inserts an element `item` at position `position` in the original list.

```java
public static IntList insertIterative (IntList L, int item, int position) {
    if (L == null) {
        return new IntList(item, L);
    }
    if (position == 0) {
        L.tail = new IntList(L.head, L.tail);
        L.head = item;
    } else {
        IntList current = L;
        while (position > 1 && current.tail != null) { // Loop until we get to the position we care about
            current = current.tail;
            position -= 1;
        }
    }
    return L;
}
```
Inserting into a Linked List *Extra*

Implement `insertIterative` such that it inserts an element `item` at position `position` in the original list.

```java
public static IntList insertIterative (IntList L, int item, int position) {
    if (L == null) {
        return new IntList(item, L);
    }
    if (position == 0) {
        L.tail = new IntList(L.head, L.tail);
        L.head = item;
    } else {
        IntList current = L;
        while (position > 1 && current.tail != null) {
            current = current.tail;
            position -= 1;
        }
        IntList newNode = new IntList(item, current.tail); // Create the new node
        current.tail = newNode; // Make sure the previous pointer points at it
    }
    return L;
}
```
Implement `insertIterative` such that it inserts an element `item` at position `position` in the original list.

```java
public static IntList insertIterative (IntList L, int item, int position) {
    if (L == null) {
        return new IntList(item, L);
    }
    if (position == 0) {
        L.tail = new IntList(L.head, L.tail);
        L.head = item;
    } else {
        IntList current = L;
        while (position > 1 && current.tail != null) {
            current = current.tail;
            position -= 1;
        }
        IntList newNode = new IntList(item, current.tail);
        current.tail = newNode;
    }
    return L;
}
```
5 Shifting a Linked List *Extra*

Implement `shiftListDestructive` such that it shifts the list circularly by one destructively.

```java
public static IntList shiftListDestructive (IntList L) {
    
    }
```
5 Shifting a Linked List *Extra*

Implement `shiftListDestructive` such that it shifts the list circularly by one destructively.

```java
public static IntList shiftListDestructive (IntList L) {
    if (L == null) { // First things first - base case for if L is null
        return null;
    }
}
```
Implement `shiftListDestructive` such that it shifts the list circularly by one destructively.

```java
public static IntList shiftListDestructive (IntList L) {
    if (L == null) {
        return null;
    }
}
```

// What we want to do is take the list at the beginning and move it to the end.
5 Shifting a Linked List *Extra*

Implement `shiftListDestructive` such that it shifts the list circularly by one destructively.

```java
public static IntList shiftListDestructive (IntList L) {
    if (L == null) {
        return null;
    }
    IntList current = L; // First let's make a pointer and point at the last item
    // Further implementation...
}
```
Implement `shiftListDestructive` such that it shifts the list circularly by one destructively.

```java
public static IntList shiftListDestructive (IntList L) {
    if (L == null) {
        return null;
    }
    IntList current = L;
    while (current.tail != null) { // Iterate until current points at the last item
        current = current.tail;
    }
    // Your code here...
}
```
5 Shifting a Linked List *Extra*

Implement `shiftListDestructive` such that it shifts the list circularly by one destructively.

```java
public static IntList shiftListDestructive (IntList L) {
    if (L == null) {
        return null;
    }
    IntList current = L;
    while (current.tail != null) {
        current = current.tail;
    }
    current.tail = L; // Now, point the tail of the last node at the "first" node
}
```
5 Shifting a Linked List *Extra*

Implement `shiftListDestructive` such that it shifts the list circularly by one destructively.

```java
public static IntList shiftListDestructive (IntList L) {
    if (L == null) {
        return null;
    }
    IntList current = L;
    while (current.tail != null) {
        current = current.tail;
    }
    current.tail = L;
    IntList front = L.tail; // The second node now needs to be at the front
}
```
Implement `shiftListDestructive` such that it shifts the list circularly by one destructively.

```java
public static IntList shiftListDestructive (IntList L) {
    if (L == null) {
        return null;
    }
    IntList current = L;
    while (current.tail != null) {
        current = current.tail;
    }
    current.tail = L;
    IntList front = L.tail;
    L.tail = null; // And the old first node now points at nothing since its at the end
}
```
5 Shifting a Linked List \textit{Extra}

Implement \texttt{shiftListDestructive} such that it shifts the list circularly by one destructively.

```java
public static IntList shiftListDestructive (IntList L) {
    if (L == null) {
        return null;
    }
    IntList current = L;
    while (current.tail != null) {
        current = current.tail;
    }
    current.tail = L;
    IntList front = L.tail;
    L.tail = null;
    return front; // Finally, return our new list!
}
```
5 Shifting a Linked List \textit{Extra}

\textbf{Implement} \texttt{shiftListDestructive} \textit{such that it shifts the list circularly by one destructively.}

\begin{verbatim}
public static IntList shiftListDestructive (IntList L) {
    if (L == null) {
        return null;
    }
    IntList current = L;
    while (current.tail != null) {
        current = current.tail;
    }
    current.tail = L;
    IntList front = L.tail;
    L.tail = null;
    return front;
}
\end{verbatim}