disc. 3
cs61b sp22

pointers

values, containers, etc.

slides
bit.ly/abhi-disc

attendance
bit.ly/abhi-attendance

announcements

announcements

1. HW 0, Lab 1, and Lab 2 due today

announcements

1. HW 0, Lab 1, and Lab 2 due today
2. HW 1 due 2/1 (tomorrow)

announcements

1. HW 0, Lab 1, and Lab 2 due today
2. HW 1 due 2/1 (tomorrow)
3. Weekly Surveys are worth points + due every Monday

announcements

1. HW 0, Lab 1, and Lab 2 due today
2. HW 1 due 2/1 (tomorrow)
3. Weekly Surveys are worth points + due every Monday
4. Topical Review Session on Java this Friday 2-3:30 PM

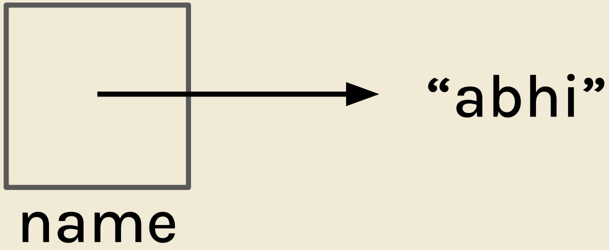
containers

containers

- simple container
  - named, contain values/pointers

containers

- simple container
    - named, contain values/pointers

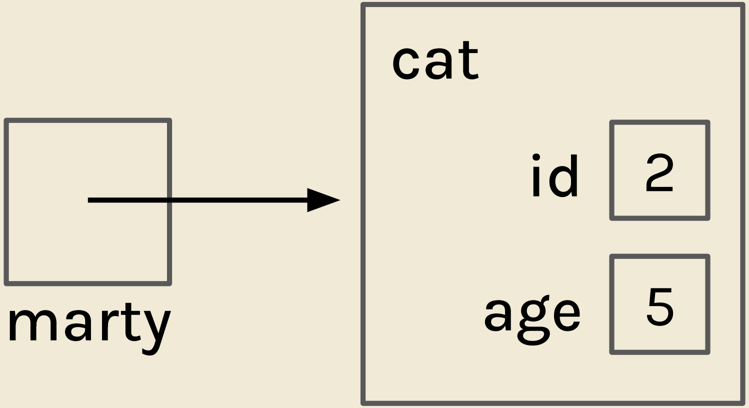


containers

- structured container
  - anonymous, contain simple containers/objects

containers

- structured container
  - anonymous, contain simple containers/objects



containers

- <u>simple container</u>
  - named, contain values/pointers
- <u>structured container</u>
  - anonymous, contain simple containers/objects

values

values

- things that can’t be *modified* without being *replaced*

values

- things that can’t be *modified* without being *replaced*
    - Numbers → Numbers as we know them (byte, short, int, double, long, float)

values

- things that can’t be ***modified*** without being ***replaced***
  - Numbers → Numbers as we know them (byte, short, int, double, long, float)
  - Letters → Characters (char)

values

- things that can't 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)

values

- things that can’t 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

values

- things that can’t 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

linked lists

linked lists

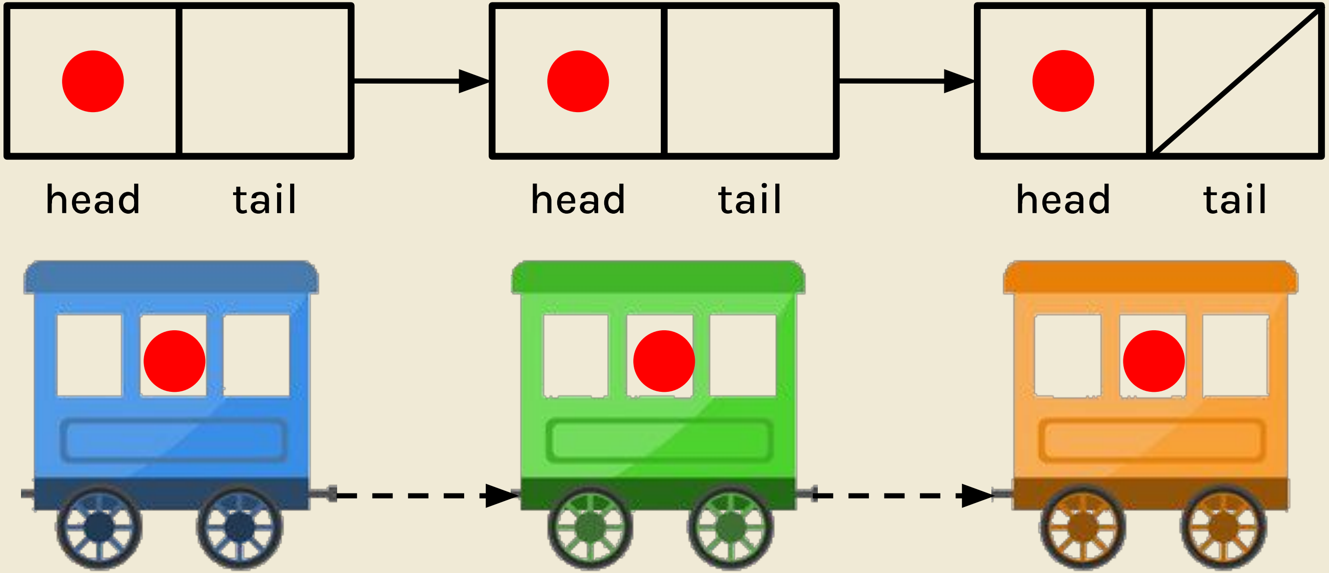
- data structures of structured containers

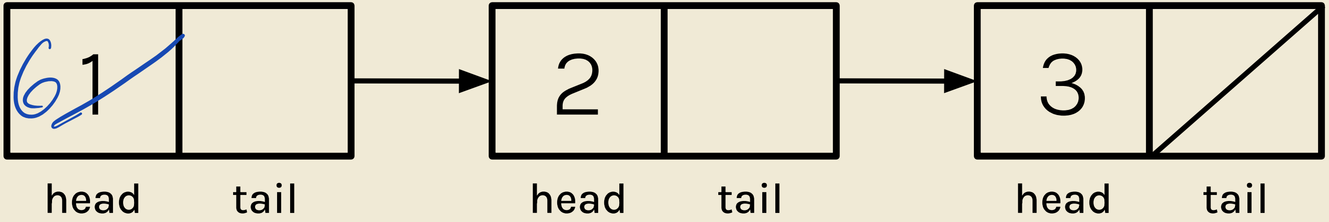
linked lists

- data structures of structured containers
    - each container has two simple containers

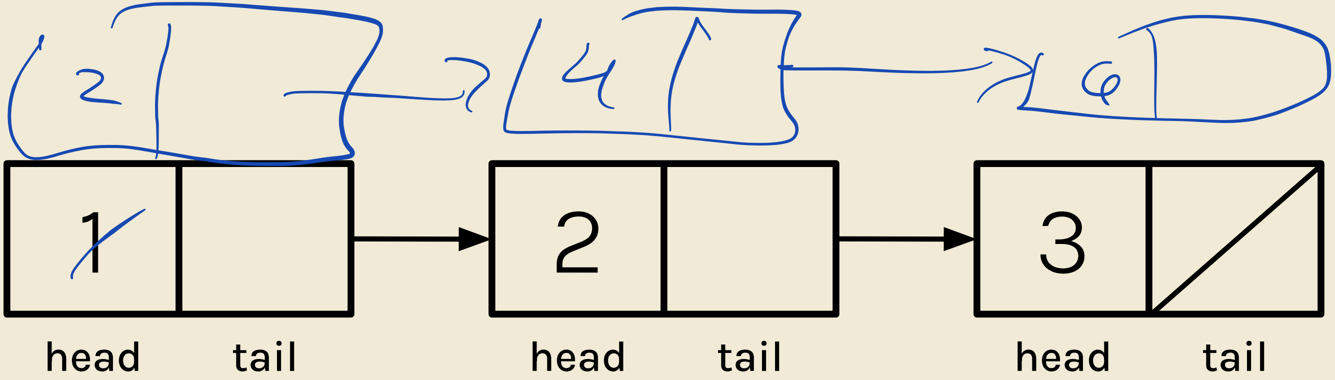
linked lists

- data structures of structured containers
  - each container has two simple containers
    - <u>list.head:</u> a value
    - <u>list.tail:</u> a pointer to the next structured container


head
tail
head
tail
head
tail



<u>destructive:</u> modifying object parameters



destructive: modifying object parameters

nondestructive: no modification to original object

arrays

arrays

- data structures of simple containers of the same type of value (int, String, etc.)
  - arr[i] holds value in ith position of array

arrays

- data structures of simple containers of the same type of value (int, String, etc.)
  - arr[i] holds value in ith position of array

Link <Link<String>>

	1	4	2
i	0	1	2

Strng int [ ][ ]

worksheet

(on 61B website)

1A Boxes and Pointers

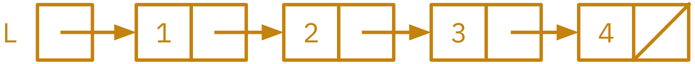
```
IntList L = IntList.list(1, 2, 3, 4);
IntList M = L.tail.tail;
IntList N = IntList.list(5, 6, 7);
N.tail.tail.tail = N;
L.tail.tail = N.tail.tail.tail.tail;
M.tail.tail = L;
```

What does the final box and pointer diagram look like?

1A Boxes and Pointers

```
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;
```

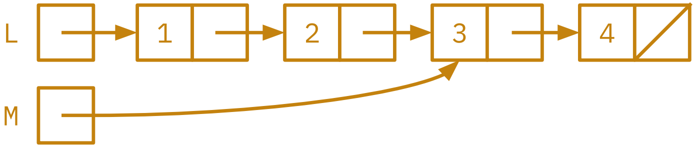
What does the final box and pointer diagram look like?



1A Boxes and Pointers

```
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;
```

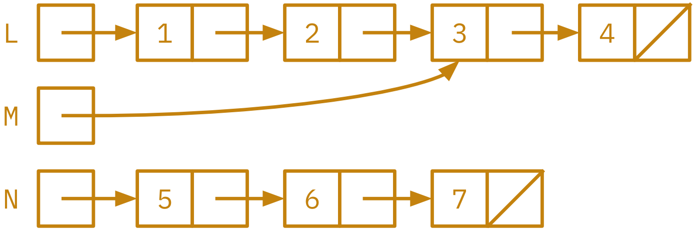
What does the final box and pointer diagram look like?



1A Boxes and Pointers

```
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;
```

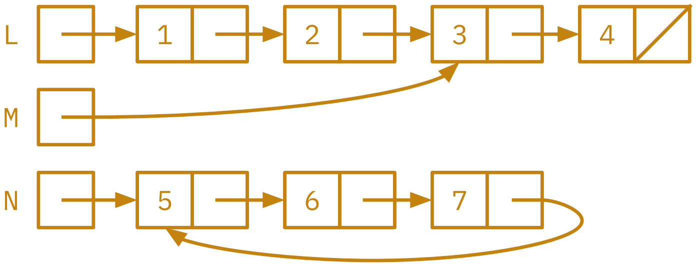
What does the final box and pointer diagram look like?



1A Boxes and Pointers

```
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;
```

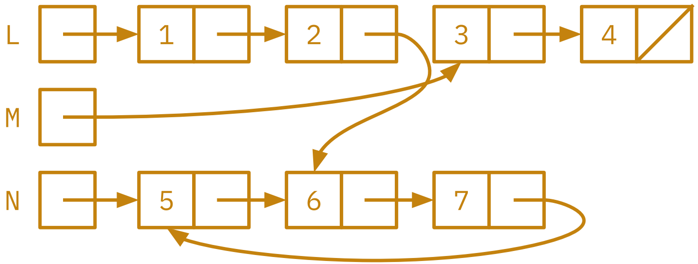
What does the final box and pointer diagram look like?



1A Boxes and Pointers

```
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;
```

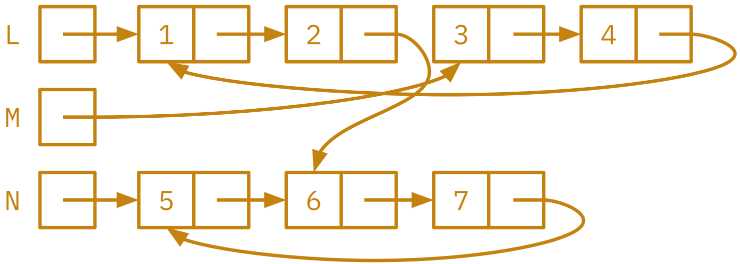
What does the final box and pointer diagram look like?



1A Boxes and Pointers

```
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;
```

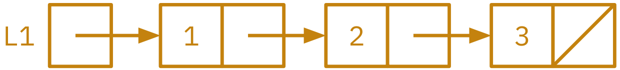
What does the final box and pointer diagram look like?



1B Boxes and Pointers *Extra*

```
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;
```

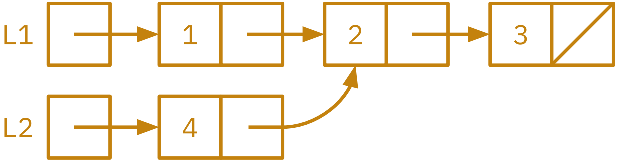
What does the final box and pointer diagram look like?



1B Boxes and Pointers *Extra*

```
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;
```

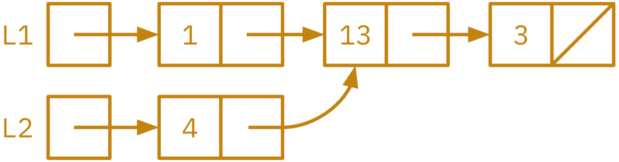
What does the final box and pointer diagram look like?



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;
```

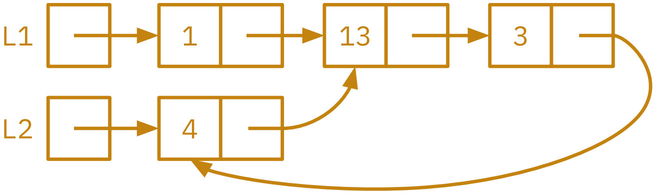
What does the final box and pointer diagram look like?



1B Boxes and Pointers *Extra*

```
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;
```

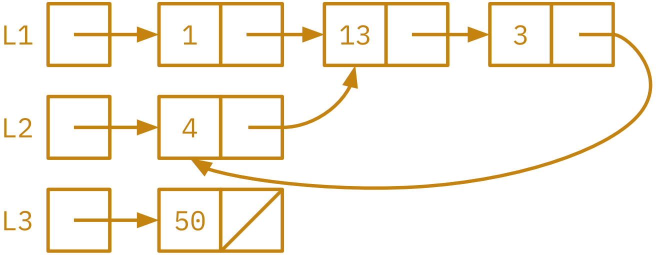
What does the final box and pointer diagram look like?



1B Boxes and Pointers *Extra*

```
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;
```

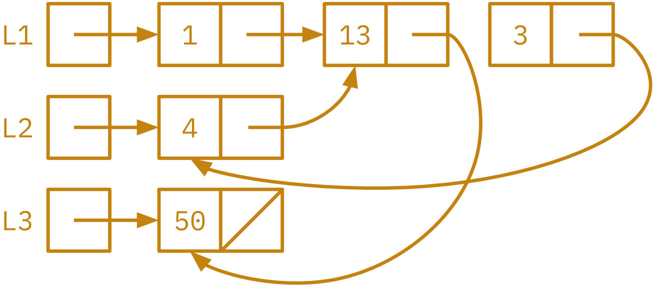
What does the final box and pointer diagram look like?



1B Boxes and Pointers *Extra*

```
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?

2 Destructive or Non-Destructive?

```
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.

```
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.

```
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.

```
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.

```
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

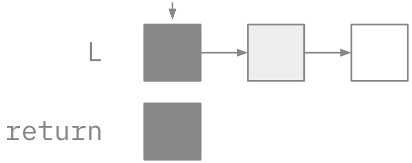
Implement reverseNondestructive such that it returns a new list with all the elements of L in reverse order.

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





}
// What if we build our list backwards?
```



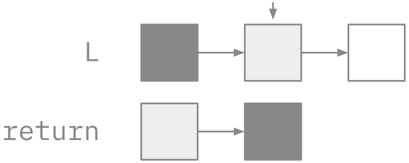
3A Reversing a List

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

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




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



3A Reversing a List

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

```
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
}
```

3A Reversing a List

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

```
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.

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









}
```

3B Reversing a List *Extra*

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

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







}
// Since this one is destructive, let's try something recursive
```

3B Reversing a List *Extra*

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

```
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.

```
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.

```
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?
```

3B Reversing a List *Extra*

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.

```
public static IntList reverseDestructive (IntList L) {
     if (L == null) {
          return L;
     } else {
          IntList reversed = reverseDestructive(L.tail); // Assume that this works



     }
}
```

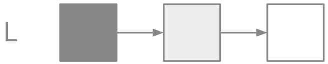
3B Reversing a List *Extra*

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);



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



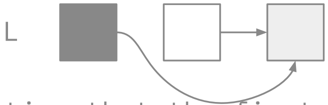
3B Reversing a List *Extra*

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

```
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
```

3B Reversing a List *Extra*

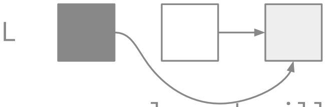
Implement reverseDestructive such that it destructively reverses the elements in L.

```
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



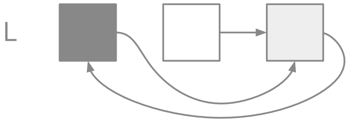
// Since our element will now come after it!

3B Reversing a List *Extra*

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

```
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
```

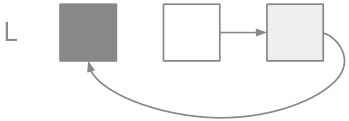


3B Reversing a List *Extra*

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

```
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
```



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;
          return reversed; // Done! Just have to return it
     }
}
```

3B Reversing a List *Extra*

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

```
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.

```
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.

```
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.

```
public static IntList insertRecusrive (IntList L, int item, int position) {




}
```

4A Inserting into a Linked List

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

```
public static IntList insertRecusrive (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.

```
public static IntList insertRecusrive (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
     }









}
```

4A Inserting into a Linked List

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

```
public static IntList insertRecusrive (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.

```
public static IntList insertRecusrive (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


    } // But we don’t have access to the previous pointer so we need to get tricky




}
```

4A Inserting into a Linked List

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

```
public static IntList insertRecusrive (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.

```
public static IntList insertRecusrive (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
     }



}
```

4A Inserting into a Linked List

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

```
public static IntList insertRecusrive (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.

```
public static IntList insertRecusrive (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
}
```

4A Inserting into a Linked List

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

```
public static IntList insertRecusrive (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.

```
public static IntList insertIterative (IntList L, int item, int position) {











}
```

4B Inserting into a Linked List *Extra*

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

```
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;
}
```

4B Inserting into a Linked List *Extra*

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

```
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; // We need a new pointer to iterate through since we need to return L







      }
      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) {
      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;
}
```

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) {
      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;
}
```

4B Inserting into a Linked List *Extra*

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

```
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.

```
public static IntList shiftListDestructive (IntList L) {









}
```

5 Shifting a Linked List *Extra*

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

```
public static IntList shiftListDestructive (IntList L) {
     if (L == null) { // First things first - base case for if L is null
          return null;
     }

}
```

5 Shifting a Linked List *Extra*

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

```
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.

```
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







}
```

5 Shifting a Linked List *Extra*

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

```
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;
     }




}
```

5 Shifting a Linked List *Extra*

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

```
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.

```
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


}
```

5 Shifting a Linked List *Extra*

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

```
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 *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;
     L.tail = null;
     return front; // Finally, return our new list!
}
```

5 Shifting a Linked List *Extra*

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

```
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;
}
```

attendance

bit.ly/abhi-attendance

feedback

bit.ly/abhi-feedback

slides: **bit.ly/abhi-disc**