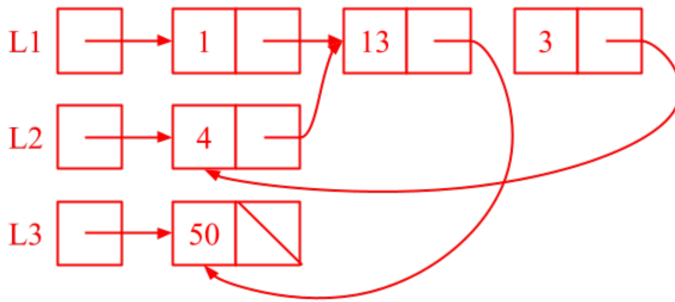


1 Boxes and Pointers (Lecture 4)

Draw a box and pointer diagram to represent the IntLists **L1**, **L2**, and **L3** after each statement.

```
IntList L1 = IntList.list(1, 2, 3);  
IntList L2 = new IntList(4, L1.rest);  
L2.rest.first = 13;  
L1.rest.rest.rest = L2;  
IntList L3 = IntList.list(50);  
L2.rest.rest = L3;
```



2 Replace (Lecture 4)

- (a) Fill in the `IntList` method `static void replace(IntList L, int a, int b)` below. Your implementation should be destructive (modifies the list in place) and iterative (does not use recursion).

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    /** Replaces all instances of a with b in L.  
     *  Modifies the passed list. Non-recursive.  
     */  
    public static void replace(IntList L, int a, int b) {  
        IntList p = L ;  
        while ( p != null ) {  
            if (p.first == a) {  
                p.first = b;  
            }  
            p = p.rest;  
        }  
    }  
}
```

- (b) Fill in the method `static IntList replace(IntList L, int a, int b)` below. Your implementation should be non-destructive (does not modify the invoked list) and recursive.

```

public class IntList {
    public int first;
    public IntList rest;

    /** Returns a copy of the provided list but with all occurrences
     * of a replaced by b. Does not modify the current list. Recursive.
     */
    public static IntList replace(IntList L, int a, int b) {
        if ( L == null ) {
            return null;
        }

        if ( L.first == a ) {
            return new IntList( b, replace(L.rest, a, b) );
        } else {
            return new IntList(L.first, replace(L.rest, a, b)) ;
        }
    }
}

```

(c) Lastly, write a *destructive*, recursive version of `replace` below:

```

/** Replaces all occurrences of a with b. Modifies the current list. Recursive. */
public static void replace(IntList L, int a, int b) {
    if ( L == null ) {
        return ;
    }

    if ( L.first == a ) {
        L.first = b ;
    }

    replace(L.rest, a, b) ;
}

```

3 Reversals (Lecture 5)

In this problem, we are trying to implement a **reverseString** method. This method takes in a string and reverses it.

```
public static String reverseString(String s) {
    // Implementation to be filled in
}
```

- (a) For each of the test cases we have written below, write down whether it is correct, and briefly explain why it is or isn't useful.

Some questions to guide your thinking: Does the test case actually test the method **reverseString**? Does it test the method with a variety of inputs? Does it test edge cases?

```
assertThat("hello world").isEqualTo("dlrow olleh");
```

This wouldn't work because we didn't actually call the method **reverseString**. Key takeaway: always call the method you are trying to test!

```
assertThat(reverseString("Hello")).isEqualTo("olleH");
```

This works well and tests the reversal of a given word.

```
assertThat(reverseString(1234)).isEqualTo(4321);
```

This would give a compiler error: the method **reverseString** is supposed to take in a **String**, while an **int** is given.

```
assertThat(reverseString("@#_!")).isEqualTo("!_#@");
```

This works well and tests whether reversals work for special characters.

```
assertThat(reverseString("")).isEqualTo("");
```

This works well and tests whether reversals work for empty strings - a common edge case.

- (b) Consider the following implementation for
- reverseString**

```
public static String reverseString(String s) {
    return s.substring(1) + s.charAt(0);
}
```

For reference, **substring(i)** returns a substring that begins with the character at the specified index and extends to the end of this string. **charAt(i)** returns the character at index **i**.

If the index is out of bounds, then an **IndexOutOfBoundsException** is thrown, and the program errors.

```
s = "Circle";
s.substring(2); // Returns "rcle"
s.charAt(1); // Returns 'i'
s.substring(2) + s.charAt(1) // Returns "rclei"
s.substring(6) // Errors!
```

Some of the test cases from the previous part are replicated below. Write down what our method would return (could be an error), and determine whether each of these would pass for our implementation.

```
assertThat(reverseString("Hello")).isEqualTo("olleH");
```

This test would fail because our implementation would return “elloH”.

```
assertThat(reverseString("")).isEqualTo("");
```

This test would fail because our implementation would be trying to access index 1 for a string of length 0, resulting in an **IndexOutOfBoundsException**.

```
assertThat(reverseString("baaaa")).isEqualTo("aaaab");
```

This test would pass because our implementation would return “aaaab”, which is the same as the expected answer. The key takeaway from this is that buggy implementations sometimes won’t be caught if your test cases are not comprehensive.

- (c) Write a correct implementation for
- reverseString**
- . You may not need all lines provided.

Hint: use **charAt(i)**, which takes in an integer and returns the character at that index. Remember that strings are 0-indexed, so the last character of a string **s** is **s.charAt(s.length() - 1)**.

```
public static String reverseString(String str) {
    String reversed = "";
    for (int i = str.length() - 1; i >= 0; i--) {
        reversed += str.charAt(i);
    }

    return reversed;
}
```

4 Insert (Lecture 6)

Implement the `SLList.insert` method which takes in an `int x` and an `int position`. It inserts `x` at the given `position`.

For example:

- If the SLList is $5 \rightarrow 6 \rightarrow 2$, then `insert(10, 1)` results in $5 \rightarrow 10 \rightarrow 6 \rightarrow 2$.
- If the SLList is $5 \rightarrow 6 \rightarrow 2$, `insert(10, 7)` results in $5 \rightarrow 6 \rightarrow 2 \rightarrow 10$.

Assume that `position` is a non-negative integer. For an easier challenge, assume `position` is between `0` and `this.size() + 1`. For a bigger challenge, if `position` is less than `0` or greater than `this.size()`, then insert at the leftmost or rightmost end, respectively.

```
public class SLList {
    private class IntNode {
        public int item;
        public IntNode next;
        public IntNode(int item, IntNode next) {
            this.item = item;
            this.next = next;
        }
    }

    private IntNode sentinel;
    public int size;

    public void addFirst(int x) {
        sentinel.next = new IntNode(x, sentinel.next);
        size += 1;
    }

    public void insert(int x, int position) {
        size += 1;
        IntNode p = sentinel;
        for (; position > 0; position--) {
            p = p.next;
        }
        p.next = new IntNode(x, p.next);
    }
}
```

5 Stair Repair (Extra Exam-Level Practice) (Lecture 4)

from Summer 2026 Midterm 1

Dawn is on a hiking trail, and she needs your help fixing some broken stairs.

An integer sequence with one or more elements is called a *stair* if each element (after the first element) is one more, one less, or equal to the previous element.

Examples of Stairs	
[0, 1, 2, 3, 4, 3, 2]	Is a stair
[-4, -3, -2, -2]	Is a stair
[9]	Is a stair
[2, 5, 3]	Is not a stair

Implement **stairify**, a method that adds nodes to an **IntList** to make the resulting sequence a stair.

stairify does not remove or modify any elements.

If the sequence is already a stair, **stairify** does nothing.

Examples of stairify	
Before stairify :	After stairify :
[2, 5, 3]	[2, 3, 4, 5, 4, 3]
[2, 4, 4, 3]	[2, 3, 4, 4, 3]
[2, 3, 4, 5]	[2, 3, 4, 5]

```
public class IntList {

    public int item;
    public IntList next;

    public IntList(int item, IntList next) {
        this.item = item;
        this.next = next;
    }

    public IntList stairify() {
        if (next != null) {
            int difference = next.item - item;
            if (difference > 1) {
                this.next = new IntList(item + 1, next);
            } else if (difference < -1) {
                this.next = new IntList(item - 1, next);
            }
            next.stairify();
        }

        return this;
    }
}
```