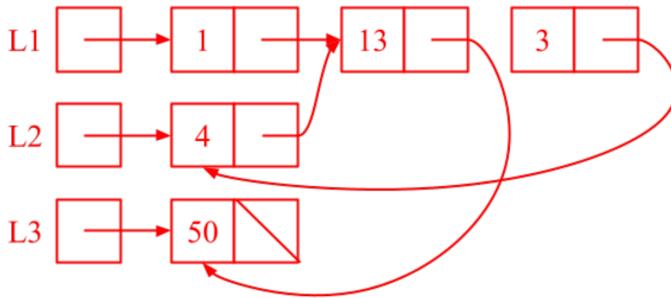


1 Boxes and Pointers

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

- (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 Deduplication

Fill in the blanks to implement the `removeDuplicates` method correctly. Given a sorted linked list of items, remove the duplicates. For example, given `1 -> 2 -> 2 -> 2 -> 3`, mutate it to become `1 -> 2 -> 3` destructively. Your implementation should be destructive and non-recursive.

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

    public static void removeDuplicates(IntList p) {
        if (p == null) {
            return;
        }

        IntList curr = p.rest;

        IntList prev = p;

        while (curr != null) {
            if (curr.first == prev.first) {
                prev.rest = curr.rest;
            } else {
                prev = curr;
            }
            curr = curr.rest;
        }
    }
}
```

Reminder: Only write one statement per line.

4 SLList

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) {
```

```
/* simple solution */
size += 1;
IntNode p = sentinel;

while (position > 0) {
    position -= 1;
    p = p.next;
}

p.next = new IntNode(x, p.next);

/* more complicated solution that handles bad inputs */
size += 1;
IntNode p = sentinel;

// Advance p to the node immediately BEFORE the insertion point.
// 1. position > 0 ensures we don't skip the head (or handles negative inputs).
// 2. p.next != null ensures we stop at the end if position > size.
while (position > 0 && p.next != null) {
    position -= 1;
    p = p.next;
}

// Create the new node and stitch it in.
p.next = new IntNode(x, p.next);
}
}
```