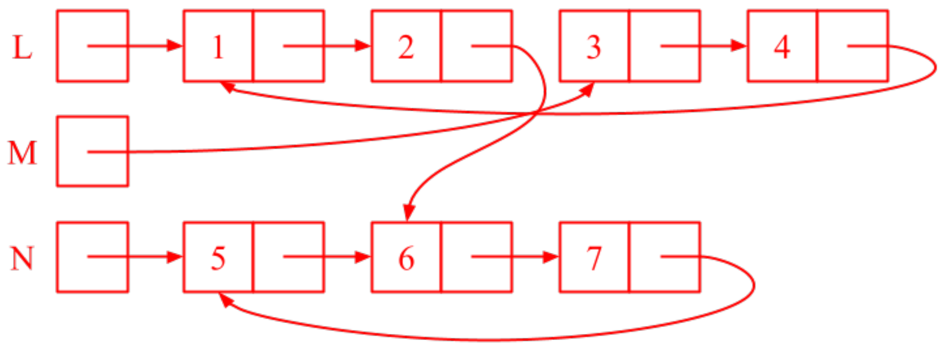
CS 61B Discussion 3: Pointers Fall 2021

1 Boxes and Pointers

Draw a box and pointer diagram to represent the IntLists `L`, `M`, and `N` after each statement.

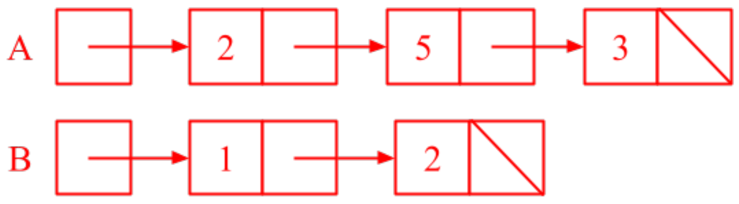
```
IntList L = IntList.list(1, 2, 3, 4);
IntList M = L.tail.tail;
IntList N = IntList.list(5, 6, 7);
N.tail.tail.tail = N;
L.tail.tail = N.tail.tail.tail.tail;
M.tail.tail = L;
```



In lecture, we discussed Consumers. If you are familiar with Python, Consumers are analogous to lambda functions. Draw a box and pointer diagram to represent the IntLists `A` and `B` after each statement.

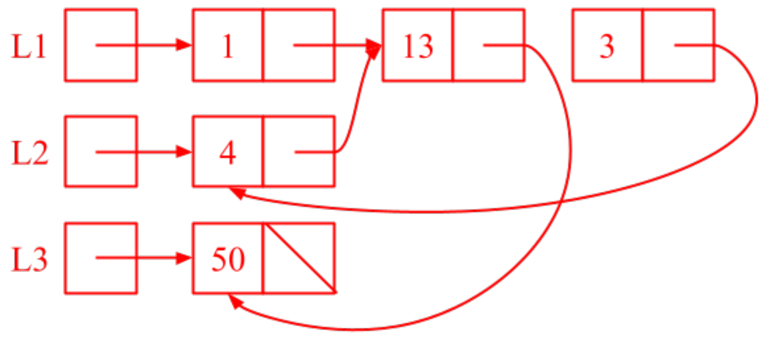
```
Consumer<IntList> trunc = (list) -> {
    int x = list.head;
    for (int i = 0; i < x && list.tail != null; i++){
        list = list.tail;
    }
    list.tail = null;
};

IntList A = IntList.list(2, 5, 3, 1, 2, 4, 8);
IntList B = A.tail.tail.tail;
trunc.accept(B);
trunc.accept(A);
```



Extra: 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.tail);
L2.tail.head = 13;
L1.tail.tail.tail = L2;
IntList L3 = IntList.list(50);
L2.tail.tail = L3;
```



2 Destructive or Nondestructive?

Below is a method that takes in an IntList and returns the value of the head of the IntList. Assume that L is never null.

```
/** Returns the head of IntList L. Assumes that L is not null. */
public static int getHead(IntList L) {
        int listHead = L.head;
        L = new IntList(5, null);
        return listHead;
}
```

Is the above method destructive or nondestructive? Explain.

Nondestructive. The input list itself is never modified (we never see anything assigned to L.head or L.tail).

3 Reversing a Linked List

Implement the following method, which reverses an IntList nondestructively. The original IntList should not be modified. Instead, the method should return a new IntList that contains the elements of L in reverse order.

```java
/** Nondestructively reverses IntList L. */
public static IntList reverseNondestructive(IntList L) {
    IntList returnList = null;
    while (L != null) {
        returnList = new IntList(L.head, returnList);
        L = L.tail;
    }
    return returnList;
}
```

Extra: Implement the following method which destructively reverses an IntList.

```java
/** Destructively reverses IntList L using recursion. */
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;
    }
}
```

This can also be implemented using iteration, as shown below.

```java
/** Destructively reverses IntList L using iteration. */
public static IntList reverseDestructive(IntList L) {
    if (L == null || L.tail == null) {
        return L;
    }
    IntList reversed = L;
    IntList current = L.tail;
    reversed.tail = null;
    IntList next = null;
    while (current != null) {
        next = current.tail;
        current.tail = reversed;
        reversed = current;
        current = next;
    }
    return reversed;
}
```

4 Inserting into a Linked List

Implement the following method to insert an element `item` at a given position `position` of an IntList L. For example, if L is (1 -> 2 -> 4) then the result of calling `insert(L, 3, 2)` yields the list (1 -> 2 -> 3 -> 4). This method should modify the original list (do not create an entirely new list from scratch). Use recursion.

```
/** Inserts item at the given position in IntList L and returns the resulting
 * IntList. If the value of position is past the end of the list, inserts the
 * item at the end of the list. Uses recursion. */
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;
}
```

Extra: Implement the method described above using iteration. `insertIterative` is a destructive method and should therefore modify the original list (just like the previous problem, do not create an entirely new list from scratch).

```
/** Inserts item at the given position in IntList L and returns the resulting
 * IntList. If the value of position is past the end of the list, inserts the
 * item at the end of the list. Uses iteration. */
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 *Extra*: Shifting a Linked List

Implement the following method to circularly shift an IntList to the left by one position *destructively*. For example, if the original list is (5 -> 4 -> 9 -> 1 -> 2 -> 3) then this method should return the list (4 -> 9 -> 1 -> 2 -> 3 -> 5). Because it is a destructive method, the original IntList should be modified. Do not use the word `new`.

```
/** Destructively shifts the elements of the given IntList L to the
 * left by one position. Returns the first node in the shifted list. */
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;
}
```