CS 61B Summer 2020 | Small Group Tutoring Section 3: Linked Lists | Worksheet 3

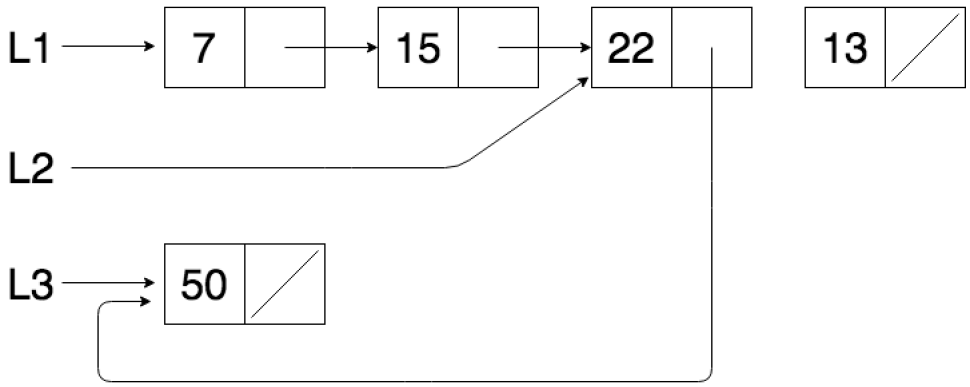
1 Pointer Practice

Draw the resulting box and pointer diagram for the IntLists after the following code is executed:

IntLists

```
IntList L1 = IntList.list(7,15,22,31);
IntList L2 = L1.next.next;
L2.next.item = 13;
L1.next.next.next = L2;
IntList L3 = new IntList(50);
L2.next.next = L3;
```

Solution:



2 Skip Me

Write a function that takes in an IntList L, which must contain at least one element, and returns an IntList with every odd indexed element removed, starting at index 0. For example, if $L = \{1,2,3,4\}$, the function should return an IntList with elements $\{1,3\}$.

1. **Destructive**: input IntList, L, should be modified

```
public static void skipDestructive (IntList L) {
    if (_____) {
        _____;
    }
    L.next = _____;
    skipDestructive(_____);
}
```

Solution:

```
public static void skipDestructive (IntList L) {
    if (L == null || L.next == null) {
        return;
    }
    L.next = L.next.next;
    skipDestructive(L.next);
}
```

In the destructive case, we must modify L so no new IntList objects or pointers should be created. The key idea is to modify the next pointers of every other element in the list to point to the element after its original next (this is achieved in line 5). We can then recursively continue this process for the remainder of L.

2. **Nondestructive**: input IntList, L, should not be modified

```
public static IntList skipNondestructive (IntList L) {
    IntList pointer = _____;
    IntList retPtr = _____;
    IntList retHead = _____;
    while (_____&& _____) {
        retPtr.next = _____;
        pointer = _____;
        retPtr = _____;
```

```
    }
    return ____________________________________________;
}
```

Solution:

```
public static IntList skipNondestructive (IntList L) {
    IntList pointer = L;
    IntList retPtr = new IntList(pointer.item);
    IntList retHead = retPtr;
    while (pointer.next != null &&  pointer.next.next != null) {
        retPtr.next = new IntList(pointer.next.next.item);
        pointer = pointer.next.next;
        retPtr = retPtr.next;
    }
    return retHead;
 }
```

The first three lines initiate IntList pointers: pointer is used to walk through the given list L and retPtr (which stands for return pointer) points to a new list that we are returning. Since we will be appending to the end of retPtr, retPtr will always be pointing to the end of the returned list so we use retHead to maintain a pointer to the front.

Within the while loop, when appending to the returned list, we must initiate a new IntList object at the next each time to ensure that the solution is nondestructive (try what happens if we don't!). We then update the pointers by moving pointer forward twice and retPtr forward once (this does the skipping action). Notice that we make a call for pointer.next.next.item within the while loop so we must ensure that pointer.next is not null (so that it has a next attribute) and pointer.next.next is not null (so that it has a item attribute).

3 Benefits of Enhancements

1. List one advantage of having a sentinel node.
   When iterating through the list, we do not need to worry about reaching a null pointer at any point; when our moving pointer reaches the sentinel, it's the end of the list. The code can be written with disregard for null-checking edge cases and it will still work properly because of the sentinel!

2. Suppose we implement a doubly linked list with a sentinel. In order to write the addFirst method, which pointers will we change?

   (a) sentinel.next The first element in the linked list is after the sentinel so we need to update sentinel.next to refer to our new element.

   (b) sentinel.prev

   (c) sentinel.next.prev The old first element in the linked list is now the second element. We need to update its previous pointer to be the new element.

   (d) sentinel.next.next