CS 61B
Spring 2020

Small Group Tutoring
Section 1: Pointers [Solutions]

Worksheet 3

1 Java Practice

1. Write a function that sums up all the digits in an integer iteratively. For example, sumDigits(31415) should return $3+1+4+1+5=14$.

```
public static int sumDigits (int x) {
}
```

Solution:

```
public static int sumDigits(int x) {
    int total = 0;
    for (int num = x; num > 0; num /= 10) {
        total += num % 10;
    }
    return total;
 }
```

Note: The ”while” loop version may be more familiar for students, but going over the ”/=” notation in the for loop can be good practice!

Alternate Solution:

```
public static int sumDigits(int x) {
    int total = 0;
    while (x > 0) {
        total += x % 10;
        x /= 10;
    }
    return total;
}
```

2. Write a function that sums up all the digits in an integer recursively.

```
public static int sumDigits (int x) {
    if (________________________________) {
        ________________________________;
    }
    return ________________________+ sumDigits(________________________);
}
```

Solution:

```
public static int sumDigits(int x) {
    if (x <= 0) {
        return 0;
    }
    return x % 10 + sumDigits(x/10);
}
```

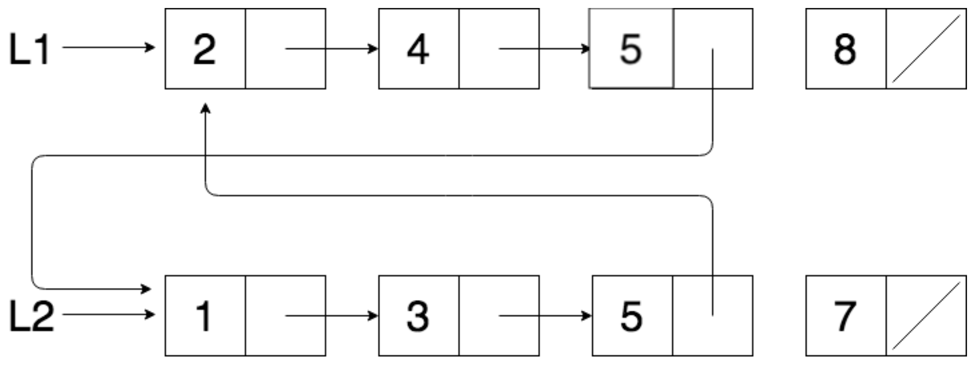
2 Pointer Practice

Draw the resulting box and pointer diagram for the L1 Singly Linked IntList after the following code is executed:

1. **IntLists**

```
IntList L1 = IntList.list(2,4,6,8);
IntList L2 = IntList.list(1,3,5,7);
L1.tail.tail.head = 5;
L2.tail.tail.tail = L1;
L1.tail.tail.tail = L2;
```

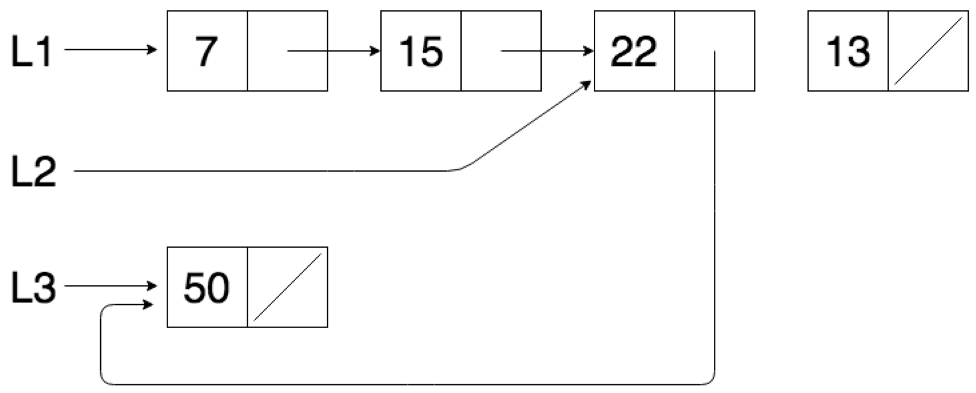
Solution:



2. **IntLists (Optional)**

```
IntList L1 = IntList.list(7,15,22,31);
IntList L2 = L1.tail.tail;
L2.tail.head = 13;
L1.tail.tail.tail = L2;
IntList L3 = IntList.list(50);
L2.tail.tail = L3;
```

Solution:



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

```
public static IntList skipNondestructive (IntList L) {
    IntList pointer = ____________________________;
    IntList retPtr = ____________________________;
    IntList retHead = ____________________________;
    while (____________________________&& ____________________________) {
        retPtr.tail = ________________________________________________________;
        pointer = ______________________________________________;
        retPtr = ______________________________________________;
    }
    return ______________________________________________;
}
```

Solution:

```
public static IntList skipNondestructive (IntList L) {
    IntList pointer = L;
    IntList retPtr = new IntList(pointer.head);
    IntList retHead = retPtr;
    while (pointer.tail != null &&  pointer.tail.tail != null) {
        retPtr.tail = new IntList(pointer.tail.tail.head);
        pointer = pointer.tail.tail;
        retPtr = retPtr.tail;
    }
    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 tail 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.tail.tail.head within the while loop so we must ensure that pointer.tail is not null (so that it has a tail attribute) and pointer.tail.tail is not null (so that it has a head attribute).

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

```
public static void skipDestructive (IntList L) {
    if (______________________________________) {
        ______________________________________;
    }
    L.tail = ______________________________________;
    skipDestructive(____________________);
}
```

Solution:

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

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 tail pointers of every other element in the list to point to the element after its original tail (this is achieved in line 5). We can then recursively continue this process for the remainder of L.