Discussion 3: Pointers

Hi everyone! My name is Adel and I will be your CS 61B TA this semester 🙂

Discussion: Wed 2-3pm 3111 Etcheverry Hall
Lab: Fri 1-3pm 275 Soda Hall
OH: Tues 3-4pm in 109 Morgan Hall

email: asetoodehnia@berkeley.edu
website: asetoodehnia.github.io (or find it through the CS 61B staff webpage)
(I post my discussion materials here if you are interested in referring to them after section)

Announcements!

1. Fill out the midterm conflict form if you have any conflicts (we will not be offering alternate finals).
2. General advising session 2/10, 11am - 2 pm in Cory Courtyard.
3. Tutoring sessions start this week! See Piazza for signup/details.
4. Lab 3 has a mandatory in-person partner checkoff!
5. HW 1 released - due 2/7 .
6. Proj 0 released - due 2/18. **Please start early!**
7. Proj 0 Project parties (hosted in 2nd floor Soda labs) on Sat 2/8, 1-3pm and 2/15, 1-4pm.
8. See Piazza for more announcements.

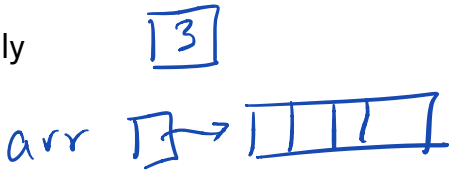
Some review first!

Values

- Values are numbers, booleans, and pointers and never change.
  - *numbers* → Just numbers as we know them, plus characters (letters) that are mapped to their corresponding number
  - *booleans* → True or False
  - *pointers* → Point to a spot in memory where we have an object stored

Containers

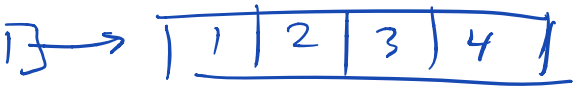
- **Simple Containers** store the values we discussed previously
  - a number
  - can also store a pointer to an object
- **Structured Containers** store other containers or objects
  - an array



Linked Lists and Arrays

- A **linked list** is a data structure that consists of individual blocks that each have two containers, one which holds a value and one which stores a pointer to the next block.

- An **array** is a container that can hold many containers, all of which must contain the *same type of value*.



Destructive vs. Non-Destructive

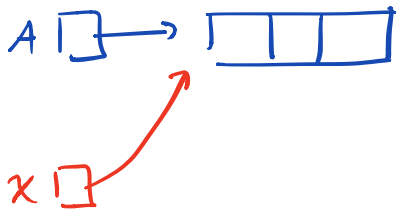
- **Destructive** functions alter the objects we pass in, thereby altering them permanently, even after we leave the function
- **Non-Destructive** functions don't alter the objects we pass in

Pass by Value

- **Pass by value** means that the method parameter values are copied to another variable and then the **copied object is passed**, that's why it's called pass by value.
- This is how Java works, so don't forget!

```
private static void f(int[] x){ ... }

f(A);
```

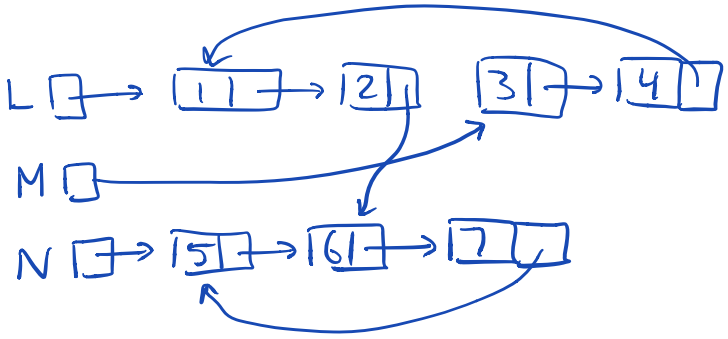


CS 61B Discussion 3: Pointers Spring 2020

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;
```



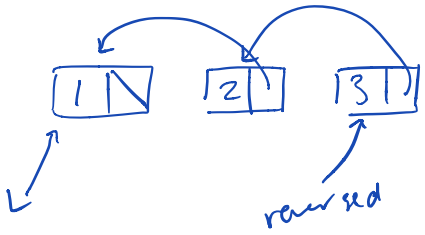
2 Reversing a Linked List

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.

```
/** 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.

```
/** Destructively reverses IntList L. */
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;
    }
}
```



3 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) {
        ...  "same as above two if statements"
        } 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;
}
```

4 *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;
}
```