Scope, Static, Linked Lists, Arrays

Exam-Level 03

Agenda

- 9:10 - 9:15 ~ hello! & announcements
- 9:15 - 9:25 ~ content review
- 9:25 - 9:40 ~ question 3
- 9:40 - 9:55 ~ question 2
- Question 1 if time (probably not)

Meet Your TA!

Hi! I'm Aniruth.

- He series
- aniruth.n@berkeley.edu
- EECS and Business
- Senior
- Coming off a gap year! Last was in school (and TA'd 61B) in Spring 2022
- Cook a lot (probably too much, food mad expensive)
- Have a 43 inch monitor in my room
- Former 61Baller

Announcements

- Weekly Survey 2 - due this Tuesday 9/5 (yesterday)
- Lab 3 - due next Monday 9/11
- Proj 1A - due next Monday 9/11
- Project Party 9/6
- Carefully read the OH guidelines if you attend

Content Review

GRoE: Golden Rule of Equals

```
“Given variables y and x:
y = x copies all the bits from x into y.”
```

Java is pass-by-value: when you call a function and give it some arguments, the function called receives an exact copy of those arguments, tied to its own local variables.

“Copies all the bits” means different things for primitive vs. reference types.

Primitive vs. Reference Types

- Primitive Types are represented by a certain number of bytes stored at the location of the variable in memory. There are only 8 in Java.

  Examples: `byte, short, int, long, float, double, boolean, char`

- Reference Types are represented by a memory address stored at the location of the variable which points to where the full object is (all objects are stored at addresses in memory). This memory address is often referred to as a *pointer*.

  Examples: Strings, Arrays, Linked Lists, Dogs, etc.

Back to the GRoE

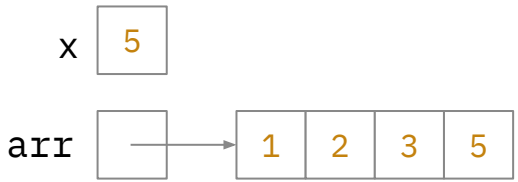
"Given variables y and x:
y = x copies all the bits from x into y."

- The value of a primitive type gets copied directly upon variable assignment
  - Ex. `int x = 5;` means that variable `x` stores the value of 5

- The value of a reference type is a "shallow" copy upon variable assignment: the pointer (memory address) is copied, and the object itself in memory is not
  - Exception: `null` is a special pointer that we compare with ==

A Quick Example

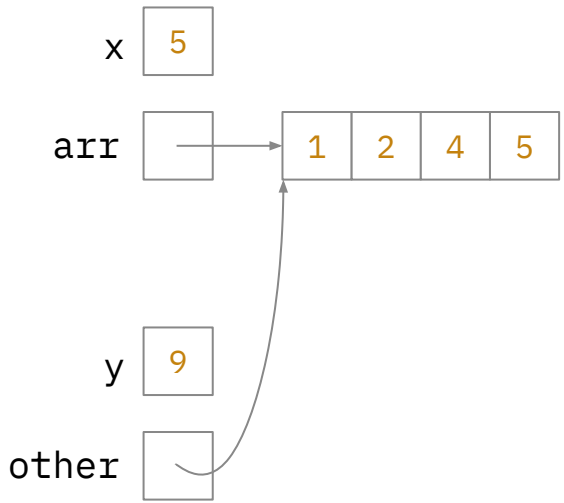
```
int x = 5;
int[] arr = new int[]{1, 2, 3, 5};
```



A Quick Example

```
int x = 5;
int[] arr = new int[]{1, 2, 3, 5};
doSomething(x, arr);
...

public void doSomething(int y, int[] other) {
    y = 9;
    other[2] = 4;
}
```



Static vs. Instance, Revisited

Static variables and functions belong to the whole class.
Example: Every 61B Student shares the same professor, and if the professor were to change it would change for everyone.

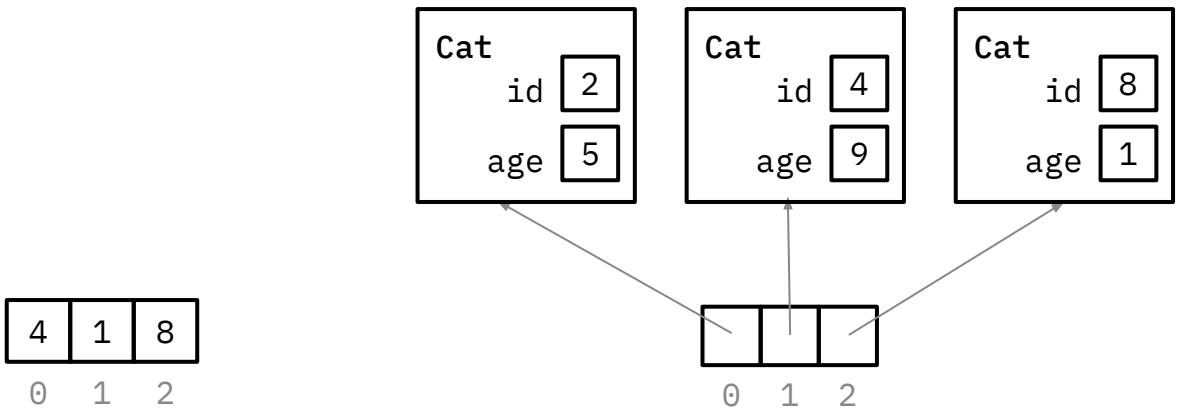
Instance variables and functions belong to each individual instance.
Example: Each 61B Student has their own ID number, and changing a student's ID number doesn't change anything for any other student.

`this` vs. `static`

- `this`
  - Non-static methods can only be called using an instance of that object, so during evaluation of that function, you will always have access to this instance of the object, referred to as `this`
- `static` methods
  - do not require an instance of that object in order to be called, so during evaluation of that function, you cannot rely on access to this instance of the object
- `static` variables
  - shared by all instances of the class; each instance does not get its own copy but can access
- Check for understanding: can you reference `this` in static methods? Can you reference static variables in instance methods? Why or why not?

Arrays

Arrays are data structures that can only hold elements of the same (primitive or reference) type of value. `arr[i]` holds a value in the ith position of the array (zero-indexed). We can also have n-dimensional arrays (ie. `int[][] a = new int[3][2];` you can index into these like `a[2][1]`)



Arrays have a set length when instantiated, so they cannot be extended / shortened with pointers like a Linked List. To resize, we need to copy over all elements to a new array (ie. `System.arraycopy`)

Linked Lists

Linked Lists are modular lists that are made up of nodes that each contain a value and a pointer to the next node. To access values in a Linked List, you must use dot notation.
Example: `intList.get(2)`

- Can be extended or shortened by changing the pointers of its nodes (unlike arrays)
- Can't be indexed directly into like an array: instead, the computer has to iterate through all of the nodes up to that point and follow their next pointers
- A sentinel is a special type of node that is often used as an empty placeholder for ease of adding / deleting nodes, especially from the front or back of the Linked List
    - In a circular doubly-linked implementation, the sentinel's `next` and `prev` pointers are the first and last nodes respectively

Approaching Exam Questions

Two step process:

1. Understand the question
2. Solve
    a. Find the Key Insight

Worksheet

3 Remove Duplicates

Using the simplified `DLList` class, implement the `removeDuplicates` method.

`removeDuplicates` should remove all duplicate items from the `DLList`. For example, if our initial list `[8, 4, 4, 6, 4, 10, 12, 12]`, our final list should be `[8, 4, 6, 10, 12]`. You may *not* assume that duplicate items are grouped together, or that the list is sorted!

```
public class DLList {
    Node sentinel;

    public DLList() {
        // ...
    }

    public class Node {
        int item;
        Node prev;
        Node next;
    }
```

3 Remove Duplicates

```
Node ref = ____________________________________________________________;
Node checker;
while (_______________________________________________________________) {
     checker = ____________________________________________________________;
     while (_____________________________________________________________) {
          if (_____________________________________________________________) {
               Node checkerPrev = checker.prev;
               Node checkerNext = checker.next;
               ____________________________________________________________;
               ____________________________________________________________;
               ____________________________________________________________;
               checker = ___________________________________________________;
          } else {
               checker = ___________________________________________________;
          }
     }
     ref = _____________________________________________________________;
}
```

3 Remove Duplicates

```
Node ref = sentinel.next;
Node checker;
while (ref != sentinel) {
     checker = ref.next;
     while (checker != sentinel) {
          if (ref.item == checker.item) {
               Node checkerPrev = checker.prev;
               Node checkerNext = checker.next;
               checkerPrev.next = checker.next;
               checkerNext.prev = checker.prev;
               checker = checkerNext;
           } else {
               checker = checker.next;
           }
       }
       ref = ref.next;
}
```

2 Partition

Implement `partition`, which takes in an IntList `lst` and an integer k, and destructively partitions lst into k IntLists such that each list has the following properties:

- It is the same length as the other lists. You may assume it is evenly divisible.
- Its ordering is consistent with the ordering of `lst`.

These lists should be put in an array of length k, and this array should be returned.

For instance, if `lst` contains the elements 5, 4, 3, 2, 1, and `k = 2`, then a possible partition is putting elements [5, 3, 2] at index 0, and elements [4, 1] at index 1.

You may assume you have the access to the method reverse, which destructively reverses the ordering of a given IntList and returns a pointer to the reversed IntList. You may not create any IntList instances.

2 Partition

```
public static IntList[] partition(IntList lst, int k) {
    IntList[] array = new IntList[k];
    int index = 0;
    IntList L = ____________________________________
    while (L != null) {
        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
    }
    return array;
}
```

2 Partition

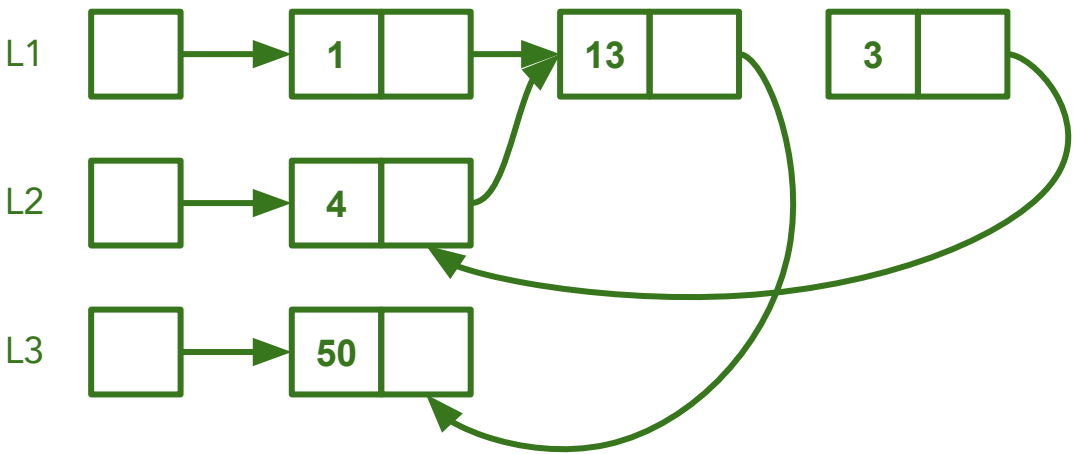
```
public static IntList[] partition(IntList lst, int k) {
    IntList[] array = new IntList[k];
    int index = 0;
    IntList L = reverse(lst);
    while (L != null) {
        IntList prevAtIndex = array[index];
        IntList next = L.rest;
        array[index] = L;
        array[index].rest = prevAtIndex;
        L = next;
        index = (index + 1) % array.length;
    }
    return array;
}
```

1 Boxes and Pointers

```
1  IntList L1 = IntList.list(1, 2, 3);
2  IntList L2 = new IntList(4, L1.rest);
3  L2.rest.first = 13;
4  L1.rest.rest.rest = L2;
5  IntList L3 = IntList.list(50);
6  L2.rest.rest = L3;
```

1 Boxes and Pointers

```
1  IntList L1 = IntList.list(1, 2, 3);
2  IntList L2 = new IntList(4,
L1.rest);
3  L2.rest.first = 13;
4  L1.rest.rest.rest = L2;
5  IntList L3 = IntList.list(50);
6  L2.rest.rest = L3;
```



CS 61B Fall 2023

Arrays, Linked Lists

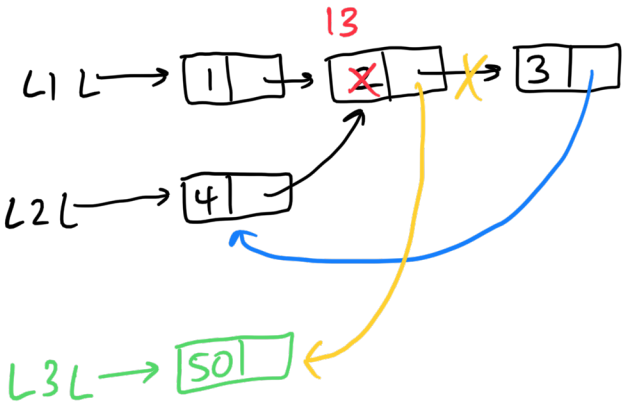
Exam-Level 3: September 4, 2023

1 Boxes and Pointers

Draw a box and pointer diagram to represent the IntLists `L1`, `L2`, and `L3` after each statement.

```
1 IntList L1 = IntList.list(1, 2, 3);
2 IntList L2 = new IntList(4, L1.rest);
3 L2.rest.first = 13;
4 L1.rest.rest.rest = L2;
5 IntList L3 = IntList.list(50);
6 L2.rest.rest = L3;
```

Each color is a different step!



2 Partition

Implement `partition`, which takes in an `IntList lst` and an integer `k`, and *destructively* partitions `lst` into `k IntLists` with the following properties:

- It is the **same** length as the other lists. You may assume the `IntList` is evenly divisible.
- Its ordering is consistent with the ordering of `lst`, i.e. items in earlier in `lst` must **precede** items that are later.

These lists should be put in an array of length `k`, and this array should be returned.
For instance, if `lst` contains the elements 6, 5, 4, 3, 2, 1, and `k = 2`, then a **possible** partition, is putting elements [6 4, 2] at index 0, and elements [5, 3, 1] at index 1.

You may assume you have the access to the method `reverse`, which destructively reverses the ordering of a given `IntList` and returns a pointer to the reversed `IntList`. **Hint:** Think about how to build up the IntList backward at each index, starting with `null`.

↑ Pretty major hint!

You may not create any `IntList` instances.

```
1  public static IntList[] partition(IntList lst, int k) {
2      IntList[] array = new IntList[k];
3      int index = 0;
4      IntList L = reverse(lst)
5      while (L != null) {
6
7          IntList curr = array[index];
8
9          IntList next = L.rest;
10
11         array[index] = L;
12
13         array[index].rest = curr;
14
15         
16
17         L = L.rest   next
18
19         index = index+1 % k;
20     }
21     return array;
22 }
```

Use index to index into array

Order of Completion:
4 19 17
7 9 11 13 17

Example:
L → 1, 2, 3, 4, 5, 6

array [0] → 1 ← curr
[1] → 2
array [0] → 3 → 1

Key Insight:
Shifting over is tricky. Need to create a pointer to remember it, grab the next item, slot it in, then put the old item in.

Initial Solution: Problem is updating L - it needs to be stored so it can proceed (since L.rest is set to curr).

```
IntList curr = array[index];
array[index] = L;
L.rest = curr;
```

↑ changed to array[index] in this initial solution both are the same

3 Remove Duplicates

Using the simplified `DLList` class defined on the next page, implement the `removeDuplicates` method.

`removeDuplicates` should remove all duplicate items from the **DLList**. For example, if our initial list is `[8, 4, 4, 6, 4, 10, 12, 12]`, our final list should be `[8, 4, 6, 10, 12]`. You may **not** assume that duplicate items are grouped together, or that the list is sorted!

```
1  public class DLList {
2      Node sentinel;
3
4      public DLList() {
5          // ...
6      }
7
8      public class Node {
9          int item;
10         Node prev;
11         Node next;
12     }
13
14     public void removeDuplicates() {
15
16         Node ref = sentinel.next;
17         Node checker;
18
19         while (ref != null) {
20
21             checker = ref.next;
22
23             while (checker != null) {
24
25                 if (ref.item = checker.item) {
26
27                     Node checkerPrev = checker.prev;
28                     Node checkerNext = checker.next;
29
30                     checkerPrev.next = checkerNext;
31
32                     checkerNext.prev = checkerPrev;
33
34                     ;
35
36                     checker = checkerNext;
37                 } else {
38
39                     checker = checker.next;
40                 }
41             }
42
43             ref = ref.next;
44         }
45     }
46 }
```

Key Insight:

When first approaching the problem - look at the ref, checker, and while condition. These are the hints on the strategy used here. Also notice ref is updated on every iteration.

Next is the inner while loop. The else condition is one line - so keep going, move checker.

Order of Completion:
16, 19, 43
21, 23, 39
25, 32, 33, 36

could be done in 2 lines