CSJ61B Lecture 5

Wednesday, February 12, 2020

Two lectures ago, we built the `IntList` data structure, then improved upon it with the `SLList` middleman in the last lecture.

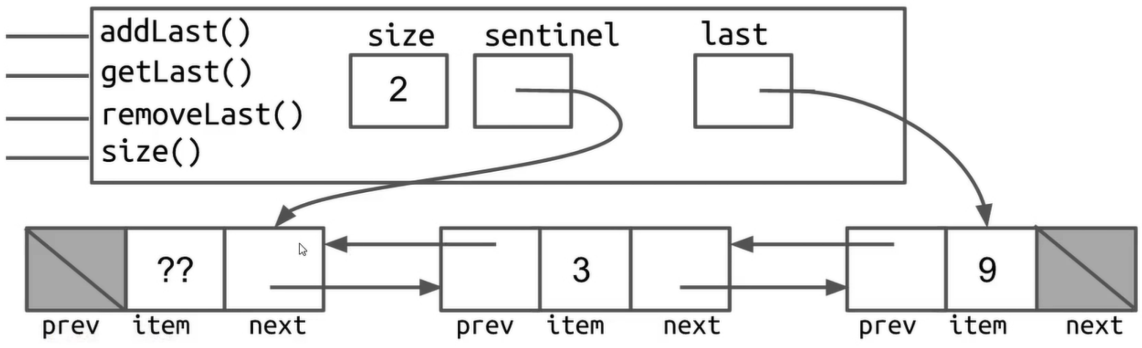
However, our `SLList` data structure still has a few problems: we can't add to the middle, and more fundamentally, adding to the back of the list is very slow, because we have to iterate through the list.

How can we improve our `addLast`? Well, an idea is to cache the `last` box, so we can directly access it.

Welll yes, our `add` operation will be fast. But suppose we wanted to support `getLast` and `removeLast`. `get` will be very fast too, but `remove` will be slow: we need to iterate through the list again to find what the new `last` pointer will be.

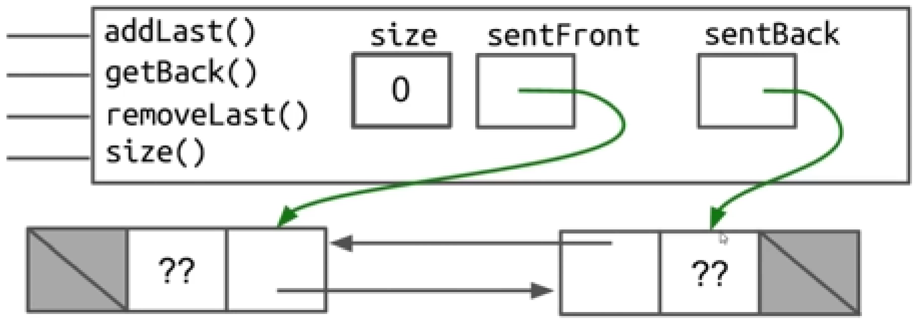
The `DLList`

What could we do to the overall structure to make `removeLast` fast? Well, one way to do this is to give each node a pointer to its predecessor, which yields a doubly linked list called `DLList`, as opposed to our previous singly linked list, hence `SLList`.



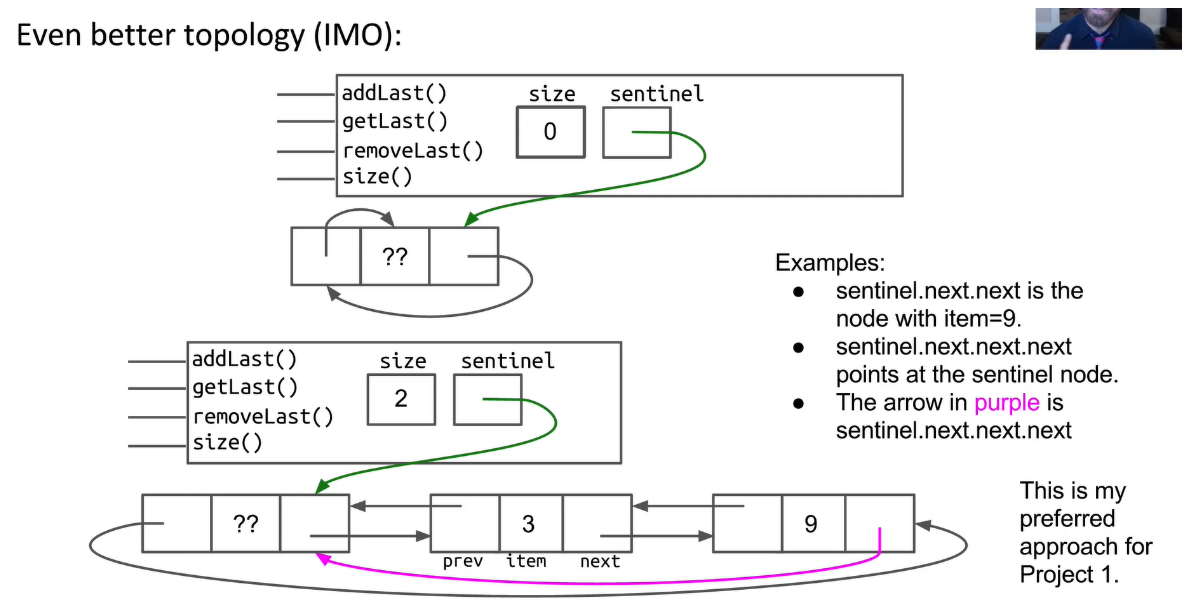
Well, there is a non-obvious problem which you'll only understand if you actually try and code this: sometimes `last` points at a sentinel, and sometimes it points at a "real" node. This will trigger again, a bunch of annoying if statements to your code, which brings us right back to the previous problem.

How can we make this code a little better? We could add a second sentinel! Now, we have `sentFront` and `sentBack`:



But there is possibly an even better approach:

We could have one sentinal that is both the front, and the back.


Even better topology (IMO):
addLast()
getLast()
removeLast()
size()
size
sentinel
0
??
addLast()
getLast()
removeLast()
size()
size
sentinel
2
??
3
9
prev
item
next
Examples:
● sentinel.next.next is the node with item=9.
● sentinel.next.next.next points at the sentinel node.
● The arrow in purple is sentinel.next.next.next
This is my preferred approach for Project 1.

Generic Lists

There's a fairly obvious limitation that makes this `DLList` we've built quite limited in usefulness: it only supports integers. We can't add strings, or other kinds of values.

How can we fix this? One lame way to do so is to copy-paste your entire block of code for `DLList` and `SLList` and replac every instance of `int` with `String`. But an even better, and more common approach way, is to parameterize the type that `SLList` and `DLList` takes in.

We first add some angled braces after the class declaration, and add a temporary name that we will use throughout the class declaration:

```java
public class SLList<LochNess> {
    private class StuffNode {
        public LochNess item;
        public StuffNode next;

        ...
    }
    ...
}
```

We will go through our code and replace every instance of `int` with `LochNess`, which will compile. How do we actually use this? Now, when we use `SLList`, we should specify a type:

```java
public class SLListLauncher {
    public static void main(String[] args) {
        SLList<String> s1 = new SLList<String>("bone");
        // In modern Java, you do not need to declare type
        // on the new SLList side.
        s1.addFirst("thugs");
    }
}
```

The `String` name will replace every instance of `LochNess` in our original `SLList` code. This means we've essentially deferred type selection until declaration.

To use generics, here's a recap:

- In the `.java` file implementing your data structure, specify your "generic type" only once at the very top of the file.
- In `.java` files that use your data structure, specify the desired type once:
    - Write out the desired type during declaration.
    - Use the empty diamond `<>` operator during instantiation.
- When declaring or instantiating your data structure, use the reference types `Integer`, `Double` etc. instead of `int` and `double`.

Arrays

So now, we've built a doubly linked list data structure that supports any kind of data and is infinitely extensible. Now, we are going to pivot and try to build the `AList` data structure that uses arrays to store information, completely orthogonal to our previous approach.

To store information in Java, we use memory boxes, which we can get by declaring variables or instantiating objects. Arrays are a special kind of object which consists of a numbered sequence of memory boxes: to get the `i`-th item of array `A`, use `A[i]`, unlike class instances which have named memory boxes.

Arrays consist of a **fixed** integer length (cannot change!), and a sequence of `N` memory boxes where `N = length`, such that all of the boxes hold the same type of value, and have the sme number of bits, and the boxes are indexed from 0 through `length - 1`.

Like instances of classes, you get one reference when it is created, and if you reassign the variables containing that reference, you may lose access to the array. Unlike classes, arrays do not have methods.

Array syntax

Like classes, arrays are almost always instantiated with the `new` keyword. Here are three perfectly valid notations for instantiating an array:

```
x = new int[3]; // Creates an array of length 3
y = new int[]{1,2,3,4,5};
int[] w = {6,7,8,9,10}; // Can omit new if declaring new variable
```

All three notations create an array with a length field and a sequence of `N` boxes.

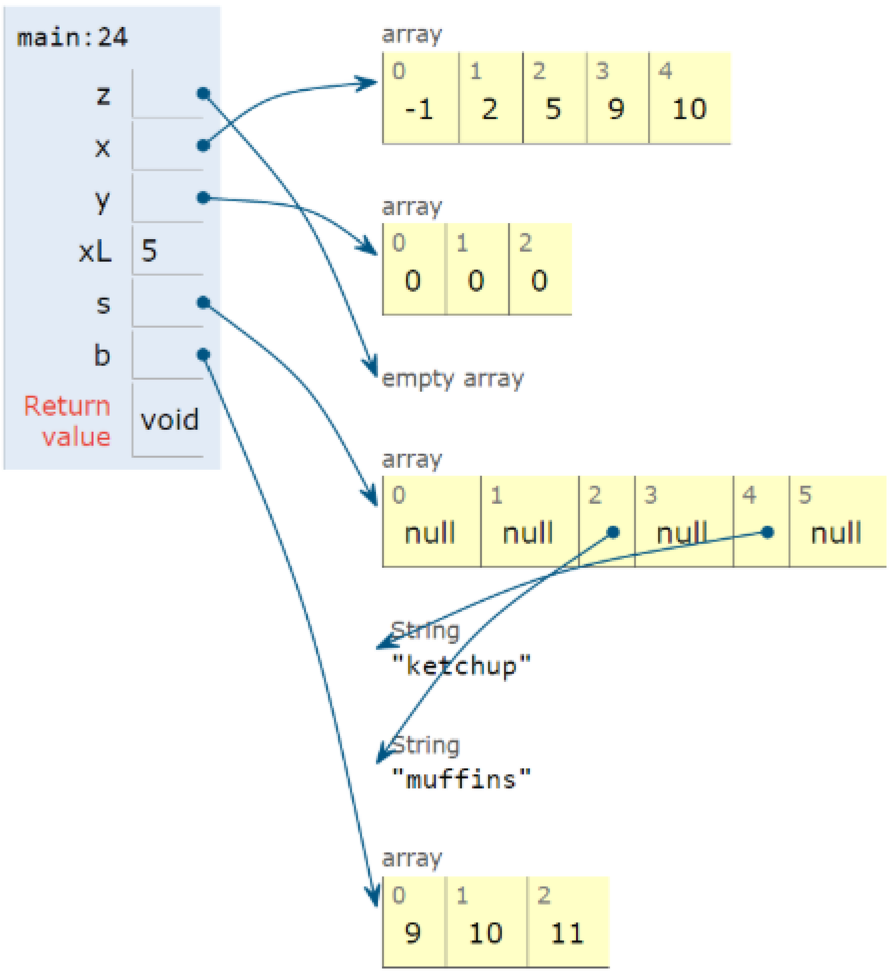
> **Optional Box and Pointer Practice**
>
> Draw the box-and-pointer diagram for the following code:
>
> ```
> int[] z = null;
> int[] x, y;
>
> x = new int[]{1, 2, 3, 4, 5};
> y = x;
> x = new int[]{-1, 2, 5, 4, 99};
> y = new int[3];
> z = new int[0];
> ```

```
int xL = x.length;

String[] s = new String[6];
s[4] = "ketchup";
s[x[3] - x[1]] = "muffins";

int[] b = {9, 10, 11};
System.arraycopy(b, 0, x, 3, 2);
```

Solution



Arraycopy

There are two ways to copy arrays: you can either copy item by item using a for loop, or the `Arraycopy` functionality in Java.

`System.Arraycopy` takes in 5 parameters: a source array, a start position in the source, the target array, the start position in the target, and the number of elements to copy.

The code `System.arraycopy(b, 0, x, 3, 2)` is analogous to the Python code `x[3:5] = b[0:2]`.

`Arraycopy` is likely to be faster, particularly for large arrays, and it is also more compact.

2D Arrays

What if our array's elements were arrays? We could use this to represent matrices or other two-dimensional objects. We can declare it using:

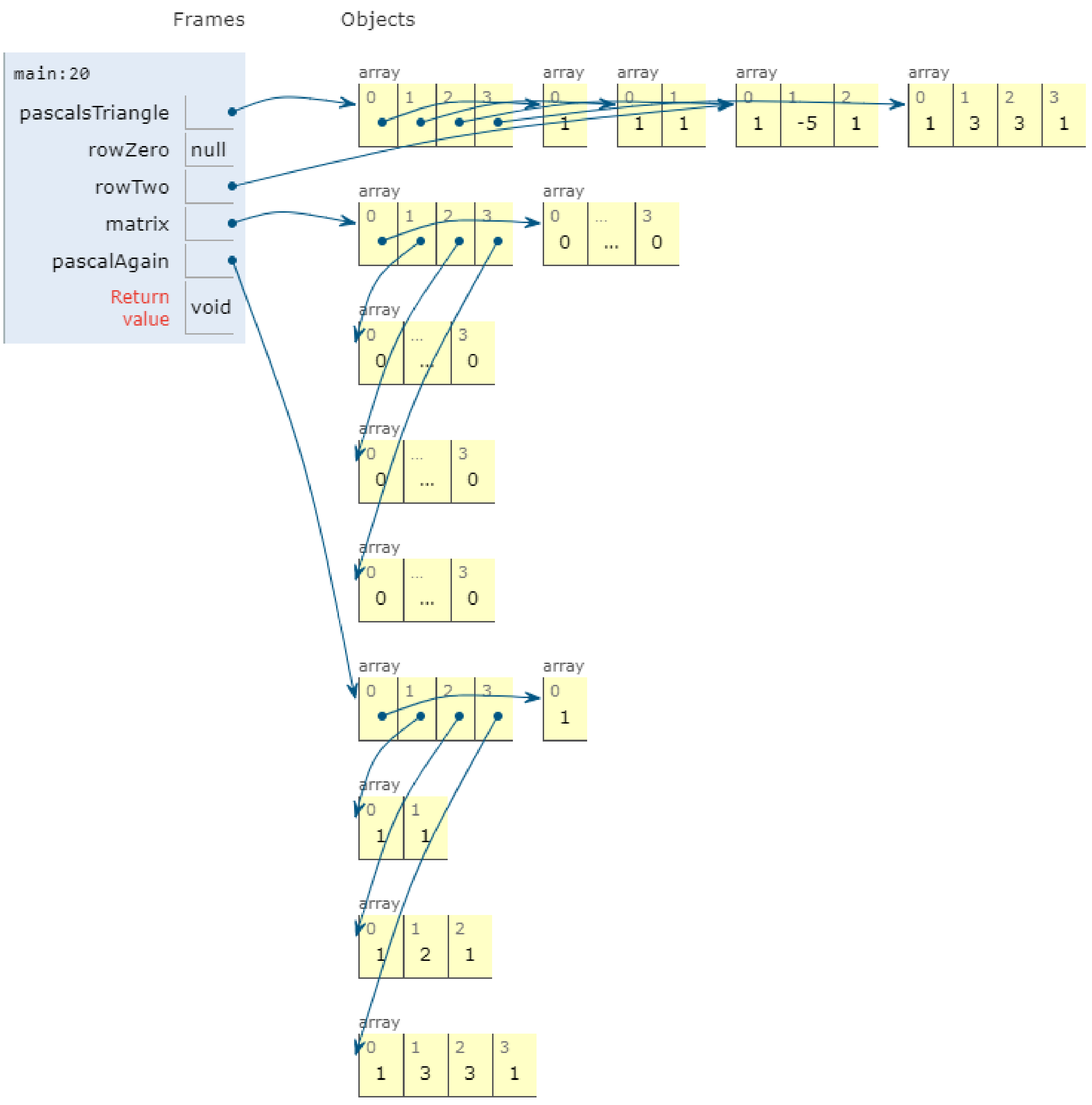
```
int[][] pascalsTriangle;
pascalsTriangle = new int[4][];
int[] rowZero = pascalsTriangle[0];

pascalsTriangle[0] = new int[]{1};
pascalsTriangle[1] = new int[]{1, 1};
pascalsTriangle[2] = new int[]{1, 2, 1};
pascalsTriangle[3] = new int[]{1, 3, 3, 1};
int[] rowTwo = pascalsTriangle[2];
rowTwo[1] = -5;

int[][] matrix;
matrix = new int[4][];
matrix = new int[4][4];

int[][] pascalAgain = new int[][]{{1}, {1, 1},
    {1, 2, 1}, {1, 3, 3, 1}};
```

This would create this array:



In other words, your arrays of arrays are references of references.

Arrays vs. Classes

Let's compare and contrast our arrays and classes. They are both used to organize a bunch of memory boxes. Here's a very quick overview of the basic differences:

Arrays	**Classes**
Accessed using box [] notation	Accessed using dot notation
Must all be of the same type.	Can be of different types.
Have a fixed number of boxes.	Have a fixed number of boxes.

Array indices can be computed at runtime. For example:

```
int[] x = {1,2,3}
int y = x[x[1]] + 3;
```

Meanwhile, you cannot do the same with classes:

```
class X {
    String y = "hello";
    String hello = "bye";

    public static void main(String[] args){
        X obj = new X;
        obj.y = "no"
    }
}
```

This will not reassign `X.hello`, because Java does not treat whatever is after the dot as an expression. It just takes it at face value.