Pre-Announcement

Women in Data Science (WiDS) 2019 Datathon Collaboration Day this Saturday (tomorrow) in the Wozniak Lounge (430 Soda).

- Open to newcomers.
- The WiDS Datathon 2019 challenge is to create a model that can detect oil palm plantations in high-resolution satellite imagery, building awareness about deforestation and oil palm plantations.
- The Datathon is a great opportunity for both new and experienced data enthusiasts to apply and hone their data science skills. Anyone from those new to data science to veterans of the field are invited to participate!
- The challenge is open to individuals or teams of up to 4, at least half of whose members must be women (female identifying individuals).

https://www.eventbrite.com/e/women-in-data-science-wids-2019-datathon-collaboration-day-tickets-54779020525

One Last Note on Plagiarism

I am teaching CS61B in the Fall. Past and future students of 61B, I'd like your opinions. (self.berkeley)
submitted 1 year ago by joshhug

[-] **comicsncats** 17 points 1 year ago
Keep hilfy's projects :)

[-] **gilabarak** EECS 8 points 1 year ago
I'd like to make a suggestion to preempt cheating. I was in this spring's class, in which around 100 students were flagged for copying code. We were warned in the first lecture that cheaters would be caught and punished, but I think that (apparently) wasn't effective as a deterrent because students don't plan to cheat at the start of the semester, and the warning gets ignored because cheating seems like some abstract, rare thing. I think just sharing the huge number of people who were caught cheating, with ruined grades and other repercussions, could make the problem more real for people. Obviously this is just one step.

Collaboration Policy

We have enumerated very specific rules whose violation will result in a score of -200 for that assignment (see about.html):

- **By You Alone**: All project code that you submit (other than skeleton code) should be written by you (or your partner) alone, except for small snippets that solve tiny subproblems (examples in collaboration policy online).
- **Do Not Possess or Share Code:** Before a project deadline, you should never be in possession of solution code that you did not write (on paper, electronically, etc.). You are equally culpable if you share. **DO NOT GIVE YOUR CODE TO ANYONE, EVEN IF THEY ARE DESPERATE. Also, don't post on GitHub publicly**!
- **Cite Your Sources:** When you receive significant assistance from someone else (with ideas, debugging, code-snippets from stack overflow etc.), you should cite that help somewhere in your source code as a comment that includes "@source". You will not be penalized for receiving this help.

Permissible But with Extreme Caution

- Helping someone debug (don't touch their keyboard/mouse/other).
- Looking at someone else's code to help them.
- **Extra Dangerous:** Looking at someone else's code to understand something. If you do this, don't write code anytime soon after looking at that code, your solution is going to gravitate straight to theirs.
- **Ultra Danger:** Working on a project alongside another person or group of people. Your code should not substantially resemble anyone else's!

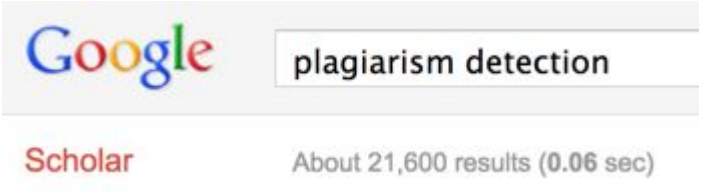
Were it enforceable, I'd say no looking at other students' code at all, but I want you to take these rules seriously (unlike, say, speed limits).

- The effect should be as if you'd never seen anyone's else code at all.

Plagiarism will (Probably) be Detected, and Dealt with Harshly

Plagiarism detection software is very sophisticated.

- Also easy to use!



Last time I taught 61B: **~65 cases** sent to the Office of Student Conduct.

- For some reason people don't believe me. From 2017 incident reports: "To be honest, when Professor Hug said there is a way to detect plagiarism, I did not believe it. I believed there is no way to detect code similarity. I mean, how is that even possible."

Please contact me if 61B is causing massive disruptions to your life.

Announcement

TA Kartik is organizing a study guide discussion on Piazza, see:

- https://piazza.com/class/jqr7hfmf4v74e?cid=373

There is also a live questions thread at:

- It exists it is on Piazza.

CS61B, 2019

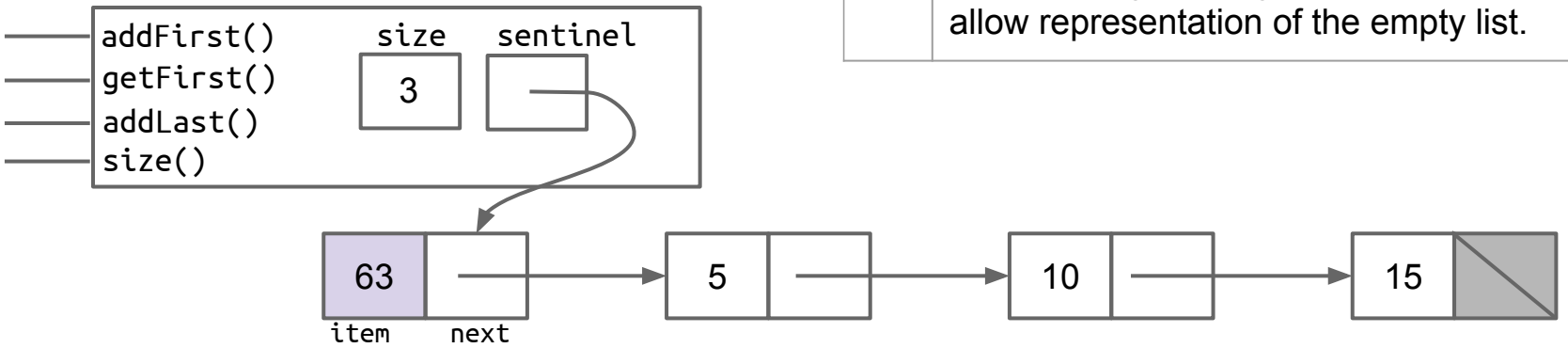
Lecture 5: DLLs and Arrays

- Doubly Linked Lists
- Generic SLLists
- Arrays
- Arrays vs. Classes

Doubly Linked Lists (In Brief)

Summary of Last Time (From IntList to SLList)

Methods	Non-Obvious Improvements	
`addFirst(int x)`	#1	Rebranding: `IntList` → `IntNode`
`getFirst()`	#2	Bureaucracy: `SLList`
`addLast(int x)`	#3	Access Control: `public` → `private`
`size()`	#4	Nested Class: Bringing `IntNode` into `SLList`
	#5	Caching: Saving `size` as an `int`.
	#6	Generalizing: Adding a `sentinel` node to allow representation of the empty list.



One Downside of SLLists

Inserting at the back of an `SLList` is much slower than the front.

```
public void addFirst(int x) {
  sentinel.next =
    new IntNode(x, sentinel.next);
}
```

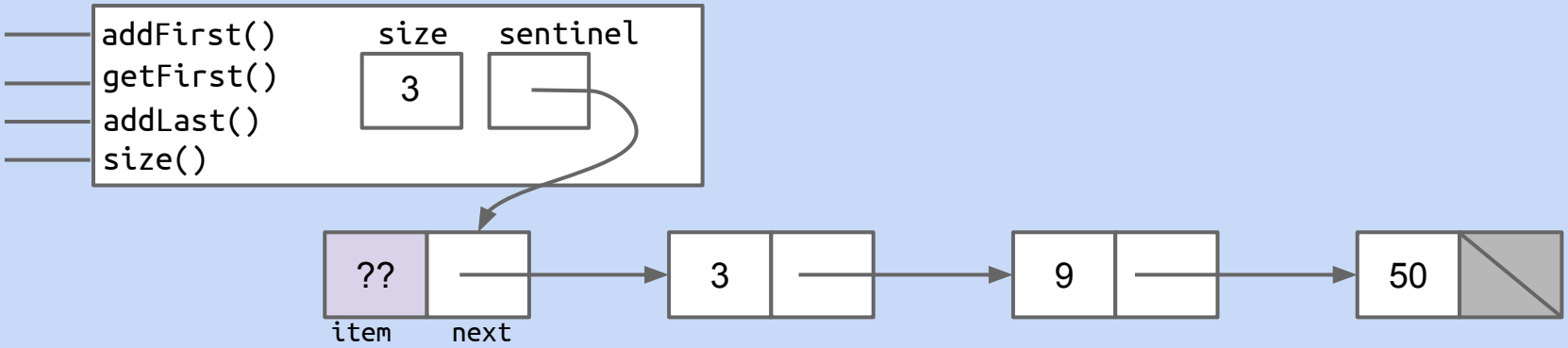
```
public void addLast(int x) {
    size += 1;

    IntNode p = sentinel;
    while (p.next != null) {
        p = p.next;
    }

    p.next = new IntNode(x, null);
}
```

Improvement #7: (???) Goal: Fast addLast

How could we modify our list data structure so that addLast is also fast?

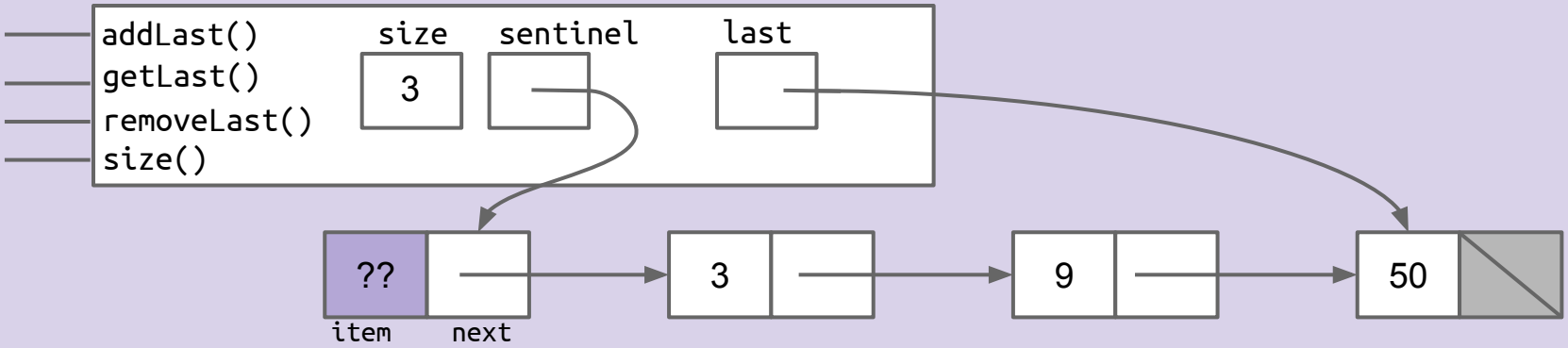


Is .last enough? http://yellkey.com/join

Suppose we want to support **add**, **get**, and **remove** operations for both ends, will having a `last` pointer result for fast operations on long lists?

A. Yes
B. No

If not, which operations would be slow?



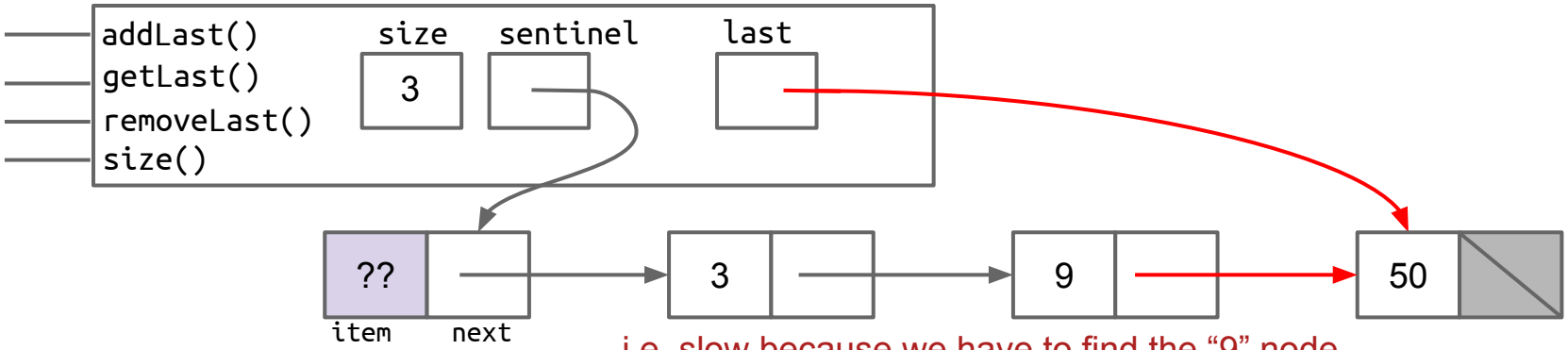
`.last` Is Not Enough

Suppose we want to support add, get, and remove operations, will having a `last` pointer result for fast operations on long lists?

A. Yes
B. No

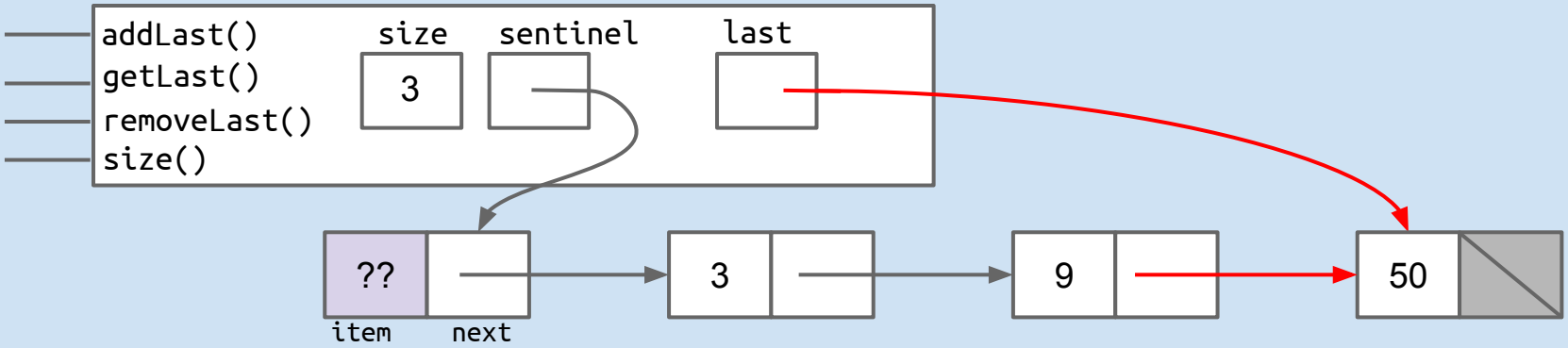
If not, which operations would be slow? **Remove!**

- Requires setting 9's next pointer to null, and point last at the 9 node.



Improvement #7: .last and ??? Goal: Fast operations on last.

We added .last. What other changes might we make so that remove is also fast?



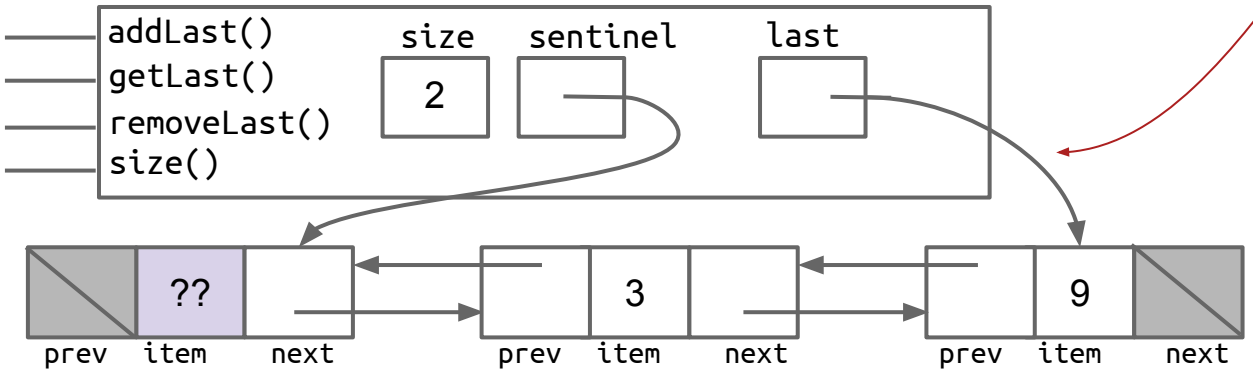
Improvement #7: .last and .prev

We added .last. What other changes might we make so that remove is also fast?

- Add backwards links from every node.
- This yields a "**doubly linked list**" or `DLList`, as opposed to our earlier "**singly linked list**" or `SLList`.

Note: Arrows point at entire nodes, not fields!

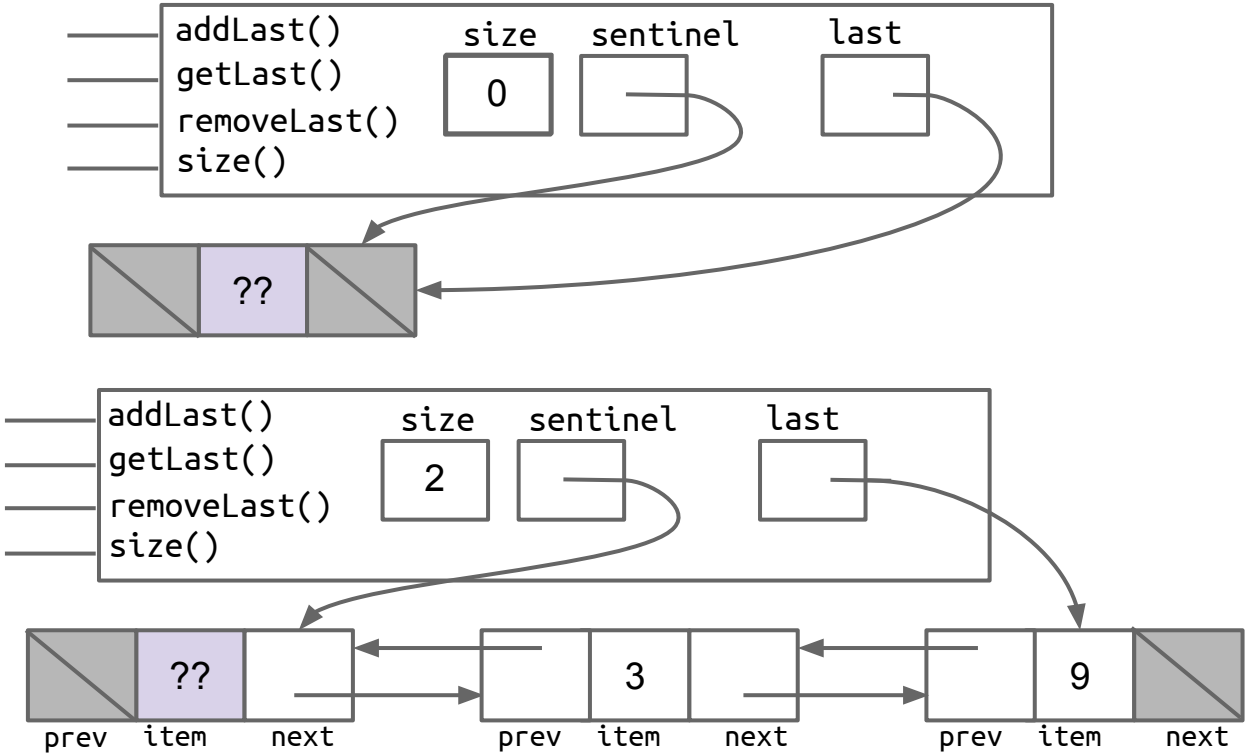
Example: `last` holds the address of the last node, not the `item` field of the sentinel node.



Doubly Linked Lists (Naive)

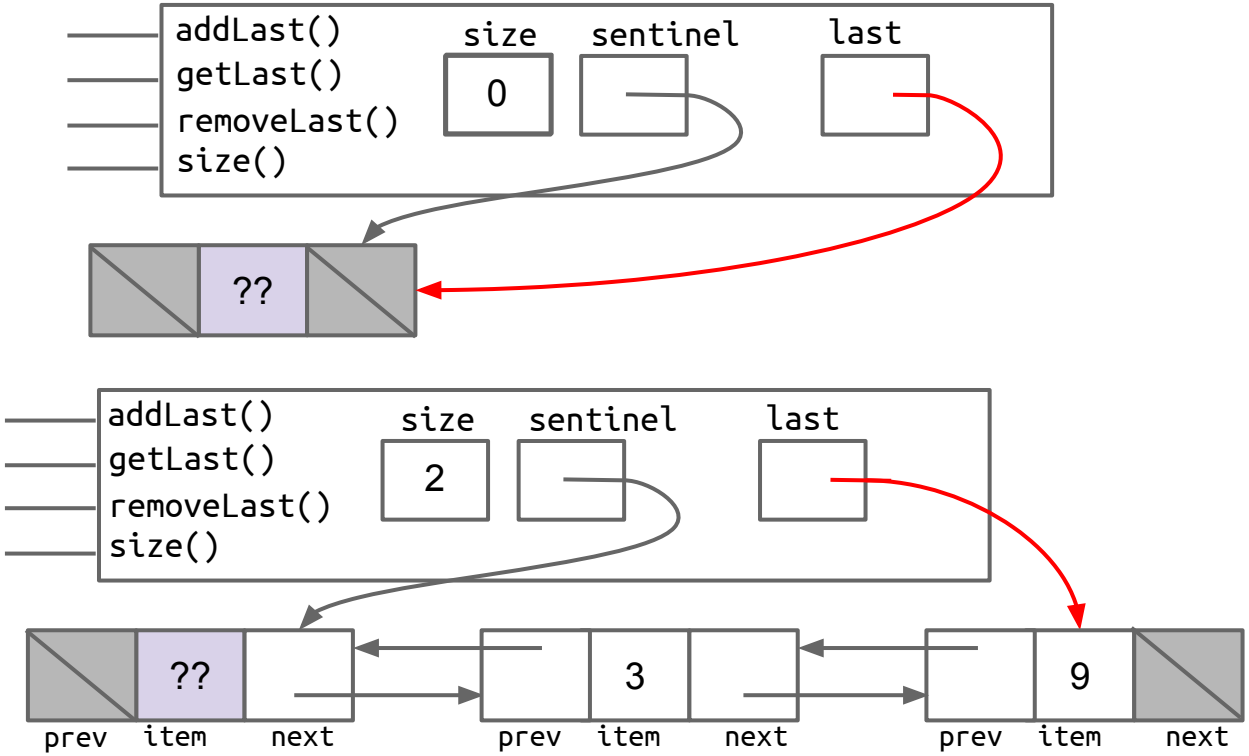
Reverse pointers allow all operations (add, get, remove) to be fast.

- We call such a list a “doubly linked list” or `DLList`.



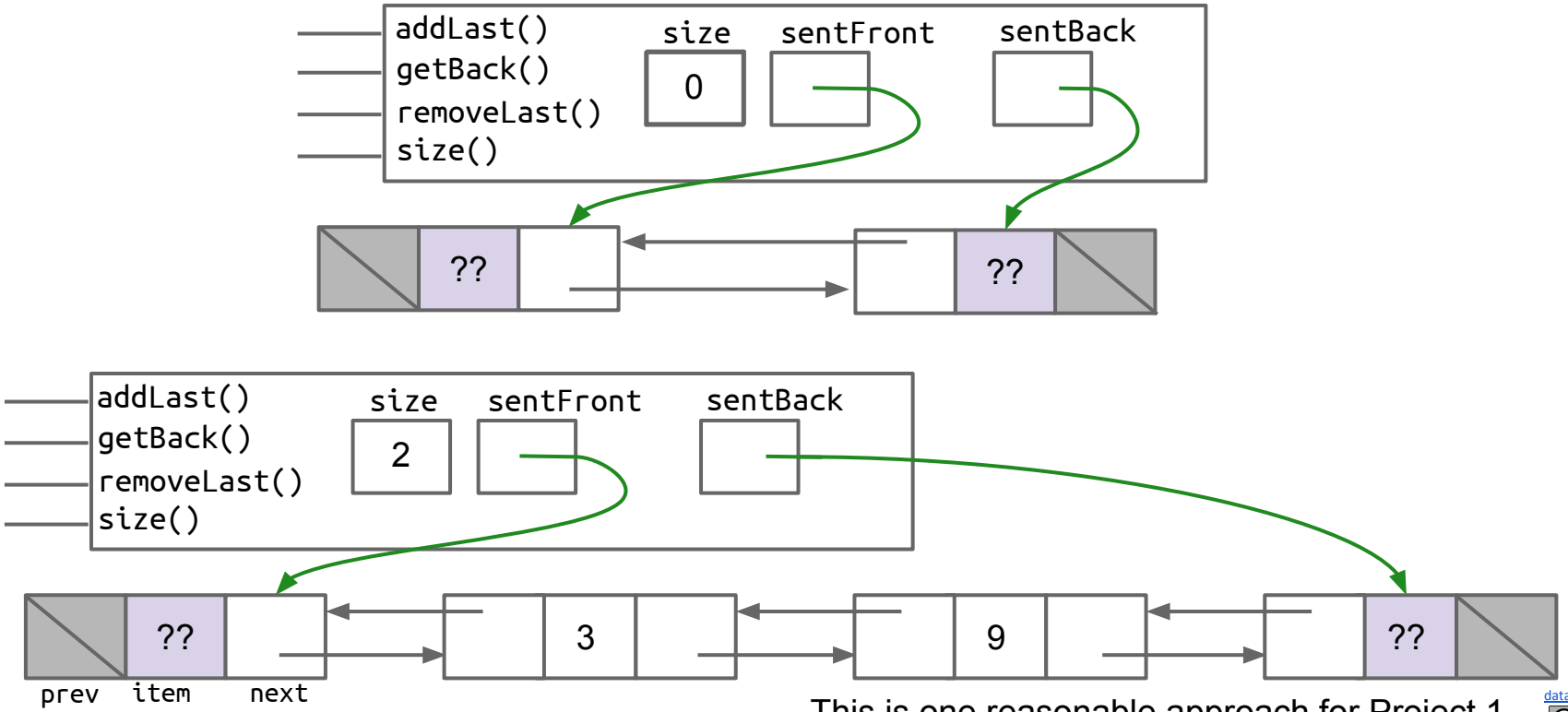
Doubly Linked Lists (Naive)

Non-obvious fact: This approach has an annoying special case: `last` sometimes points at the sentinel, and sometimes points at a 'real' node.



Doubly Linked Lists (Double Sentinel)

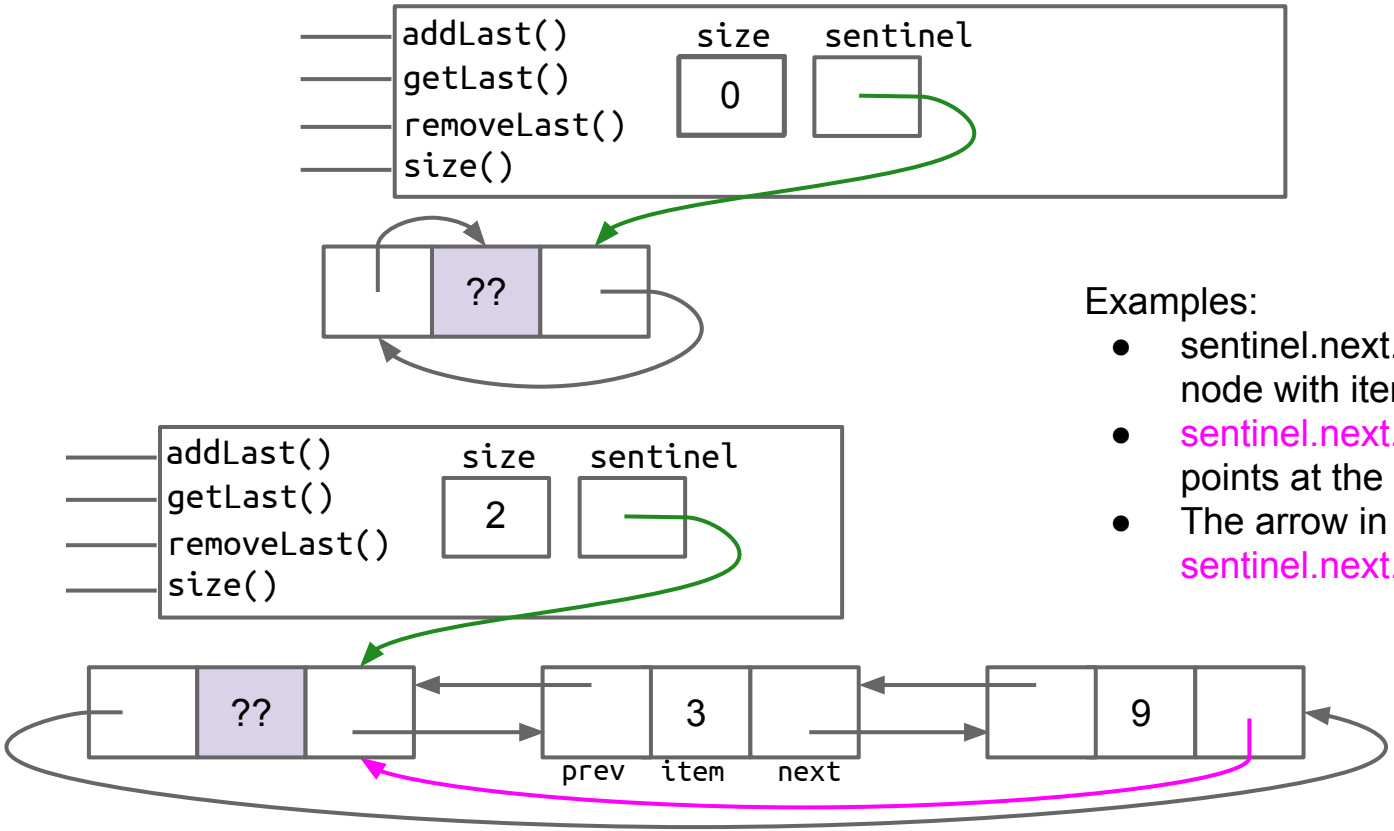
One solution: Have two sentinels.



This is one reasonable approach for Project 1.

Doubly Linked Lists (Circular Sentinel)

Even better topology (IMO):



Examples:

- sentinel.next.next is the node with item=9.
- sentinel.next.next.next points at the sentinel node.
- The arrow in magenta is sentinel.next.next.next

This is my preferred approach for Project 1.

Improvement #8: Fancier Sentinel Node(s)

While fast, adding `.last` and `.prev` introduces lots of special cases.

To avoid these, either:

- Add an additional `sentBack` sentinel at the end of the list.
- Make your linked list circular (highly recommened for project 1), with a single sentinel in the middle.

DLList Summary

Methods	Non-Obvious Improvements	
`addFirst(int x)`	#1	Rebranding: `IntList` → `IntNode`
`getFirst()`	#2	Bureaucracy: `SLList`
`size()`	#3	Access Control: `public` → `private`
`addLast(int x)`	#4	Nested Class: Bringing `IntNode` into `SLList`
`removeLast()`	#5	Caching: Saving `size` as an `int`.
	#6	Generalizing: Adding a `sentinel` node to allow representation of the empty list.
	#7	Looking back: `.last` and `.prev` allow fast `removeLast`
	#8	Sentinel upgrade: Avoiding special cases with `sentBack` or circular list.

Still many steps before we have an industrial strength data structure. Will discuss over coming weeks.

Generic Lists

Integer Only Lists

One issue with our list classes: They only supports integers.

```
public class SLList {
   private IntNode sentinel;
   private int size;

   public class IntNode {
      public int item;
      public IntNode next;
      ...
   }
   ...
}
```

```
SLList s1 = new SLList(5);
s1.addFirst(10);
```

Works fine!

```
SLList s2 = new SLList("hi");
s2.addFirst("apple");
```

```
SLListLauncher.java:6: error:
incompatible types: String cannot be
converted to int

SLList d2 = new SLList("hi");
```

SLists

Java allows us to defer type selection until declaration.

```
public class SLList<BleepBlorp> {
   private IntNode sentinel;
   private int size;

   public class IntNode {
      public BleepBlorp item;
      public IntNode next;
      ...
   }
   ...
}
```

```
SLList<Integer> s1 = new SLList<>(5);
s1.insertFront(10);

SLList<String> s2 = new SLList<>("hi");
s2.insertFront("apple");
```

Generics

We'll spend a lot more time with generics later, but here are the rules of thumb you'll need for project 1:

- 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 desired type **once**:
  - Write out desired type during **declaration**.
  - Use the empty diamond operator <> during **instantiation**.
- When declaring or instantiating your data structure, use the reference type.
  - int: Integer
  - double: Double
  - char: Character
  - boolean: Boolean
  - long: Long
  - etc.

```
DLList<Double> s1 = new DLList<>(5.3);

double x = 9.3 + 15.2;
s1.insertFront(x);
```

Arrays

Our Long Term Goal (next two lectures): The AList

In the last few lectures, we've seen how we can harness a recursive class definition to build an expandable list, ie. the `IntList`, the `SLList`, and the `DLList`.

In the next two, we'll see how we can harness arrays to build such a list.

Getting Memory Boxes

To store information, we need memory boxes, which we can get in Java by declaring variables or instantiating objects. Examples:

- `int x;` ← Gives us a memory box of 32 bits that stores ints.
- `Walrus w1;` ← Gives us a memory box of 64 bits that stores Walrus references.
- `Walrus w2 = new Walrus(30, 5.6);` ← Gives us a memory box of 64 bits that stores Walrus references, and also gives us 96 bits for storing the int size (32 bits) and double tuskSize (64 bits) of our Walrus.

Arrays are a special kind of object which consists of a **numbered** sequence of memory boxes.

- To get ith item of array A, use A[i].
- Unlike **class** instances which have have **named** memory boxes.

Arrays

Arrays consist of:

- A fixed integer **length** (cannot change!)
- A sequence of N memory boxes where **N=length**, such that:
  - All of the boxes hold the same type of value (and have same # of bits).
  - The boxes are numbered 0 through length-1.

Like instances of classes:

- You get one reference when its created.
- If you reassign all variables containing that reference, you can never get the array back.

Unlike classes, arrays do not have methods.

Arrays

Like classes, arrays are (almost always) instantiated with new.

Three valid notations:

- `x = new int[3];`
- `y = new int[]{1, 2, 3, 4, 5};`
- `int[] z = {9, 10, 11, 12, 13};`

Creates array containing 3 int boxes (32 x 3 = 96 bits total). Each container gets a default value.

Can omit the `new` if you are also declaring a variable.

All three notations create an array, which we saw on the last slide comprises:

- A **length** field.
- A sequence of **N boxes**, where **N** = **length**.

As an aside: In Oracle's implementation of Java, all Java objects also have some overhead. Total size of an array=192 + KN bits, where K is the number of bits per item (Sedgewick/Wayne pg. 201 for more)

Array Basics: http://goo.gl/tFyMEJ

```
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);
```

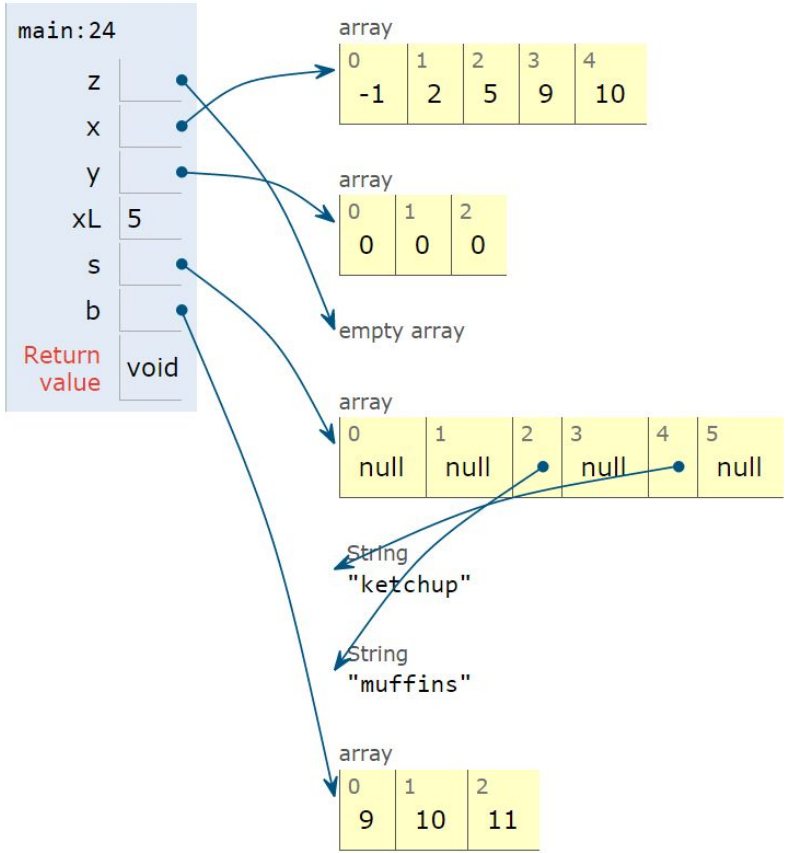
Array Basics: https://goo.gl/gzAuBa

```
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);
```



Arraycopy

Two ways to copy arrays:

- Item by item using a loop.
- Using arraycopy. Takes 5 parameters:
  - Source array
  - Start position in source
  - Target array
  - Start position in target
  - Number to copy

```
System.arraycopy(b, 0, x, 3, 2);
```

```
(In Python): x[3:5] = b[0:2]
```

arraycopy is (likely to be) faster, particularly for large arrays. More compact code.

- Code is (arguably) harder to read.

2D Arrays

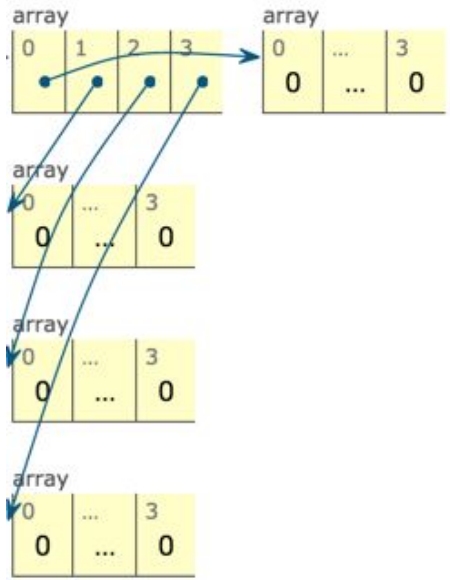
Arrays of Array Addresses (http://goo.gl/VS4cOK)

```
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}};
```



- Syntax for arrays of arrays can be a bit confounding. You'll learn through practice (much later).

Array Boxes Can Contain References to Arrays!

```
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}};
```

Array of int array references.

Create four boxes, each can store an int array reference

Create a new array with three boxes, storing integers 1, 2, 1, respectively. Store a reference to this array in pascalsTriangle box #2.

Creates 1 total array.

Creates 5 total arrays.

- Syntax for arrays of arrays can be a bit confounding. You'll learn through practice (much later).

What Does This Code Do? http://yellkey.com/?

What will be the value of x[0][0] and w[0][0] when the code shown completes?

A. x: 1, w: 1
B. x: 1, w: -1
C. x: -1, w: 1
D. x: -1, w: -1
E. Other

```
int[][] x = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

int[][] z = new int[3][];
z[0] = x[0];
z[0][0] = -z[0][0];

int[][] w = new int[3][3];
System.arraycopy(x[0], 0, w[0], 0, 3);
w[0][0] = -w[0][0];
```

arraycopy parameters are:

1. Source array
2. Start position in source
3. Target array
4. Start position in target
5. Number to copy

Answer: https://goo.gl/CqrZ7Y

Arrays vs. Classes

Arrays vs. Classes

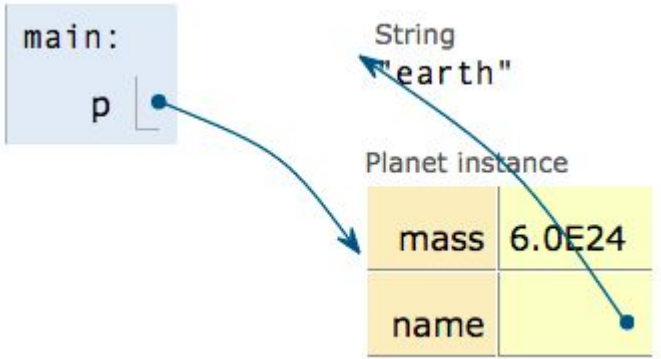
Arrays and Classes can both be used to organize a bunch of memory boxes.

- Array boxes are accessed using [] notation.
- Class boxes are accessed using dot notation.
- Array boxes must all be of the same type.
- Class boxes may be of different types.
- Both have a fixed number of boxes.

```
public class Planet {
   public double mass;
   public String name;

   ...
}
```

```
int[] x = new int[]{100, 101, 102, 103};
Planet p = new Planet(6e24, "earth");
```



Arrays vs. Classes

Array indices can be computed at runtime.

```
int[] x = new int[]{100, 101, 102, 103};
int indexOfInterest = askUser();
int k = x[indexOfInterest];
System.out.println(k);
```

```
jug ~/Dropbox/61b/lec/lists3
$ javac ArrayDemo.java
$ java ArrayDemo
What index do you want? 2
102
```

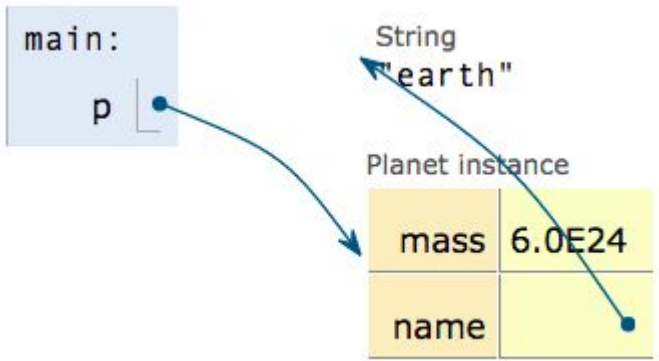
Arrays vs. Classes

Class member variable names CANNOT be computed and used at runtime.

```
String fieldOfInterest = "mass";
Planet earth = new Planet(6e24, "earth");
double mass = earth[fieldOfInterest];
System.out.println(mass);
```

```
jug ~/Dropbox/61b/lec/lists3
$ javac ClassDemo.java
ClassDemo.java:5: error: array required,
   but Planet found.

   double mass = earth[fieldOfInterest];
                       ^
```



… if you reallllly want to do this, you can: https://goo.gl/JxpyLq

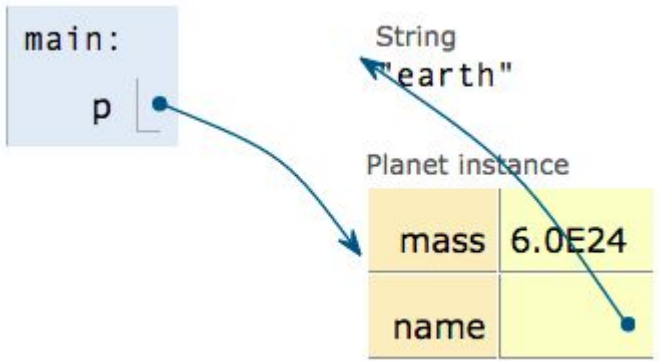
Arrays vs. Classes

Class member variable names CANNOT be computed and used at runtime.

- Dot notation doesn't work either.

```
String fieldOfInterest = "mass";
Planet earth = new Planet(6e24, "earth");
double mass = earth.fieldOfInterest;
System.out.println(mass);
```

```
jug ~/Dropbox/61b/lec/lists3
$ javac ClassDemo.java
ClassDemo.java:5: error: cannot find Symbol
   double mass = earth.fieldOfInterest;
                       ^
   symbol:   variable fieldOfInterest
   location: variable earth of type Planet
```



… if you reallllly want to do this, you can: https://goo.gl/JxpyLq

Another view

The only (easy) way to access a member of a class is with hard-coded dot notation.

```
int k = x[indexOfInterest];     /* no problem */

double m = p.fieldOfInterest;  /* won't work */
double z = p[fieldOfInterest]; /* won't work */
/* No (sane) way to use field of interest */

double w = p.mass;             /* works fine */
```

The Java compiler does not treat text on either side of a dot as an expression, and thus it is not evaluated.

- See a compilers or programming languages class for more!