Pre-Announcements (BEAM)

61B TAs Kevin and Alex and Nicole are here to say in their freetime that they participate in a group called Berkeley Engineering and Mentoring.

- Mentoring for elementary/middle school kids.
- Infosession at 8:00 PM in 145 Dwinelle.
- Also, PIZZA.

Announcements

Project 1A is out.

- Very strongly encouraged to work on this project in IntelliJ.
  - Having the ability to visually debug your code is incredibly useful.
  - Having your IDE yell at you about compilation errors while you are writing code is really nice to avoid issues with, for example, generics.
- Autograder is up, but we still want you to write your own tests.
- Tests not graded.
- On part 1B there will be graded tests, so might be worthwhile to write tests just to save yourself some work next week.

LOST section starts tomorrow.

- Tuesdays 5-6 Dwinelle 187 section. Will cover previous week's material.

CS61B, 2018

Lecture 6: Arrays and Lists

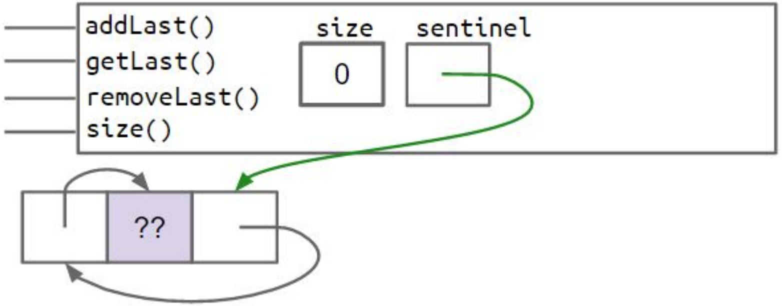
- A Last Look at Linked Lists
- Naive Array Lists
- Resizing Arrays
- Generic ALists
- Obscurantism in Java

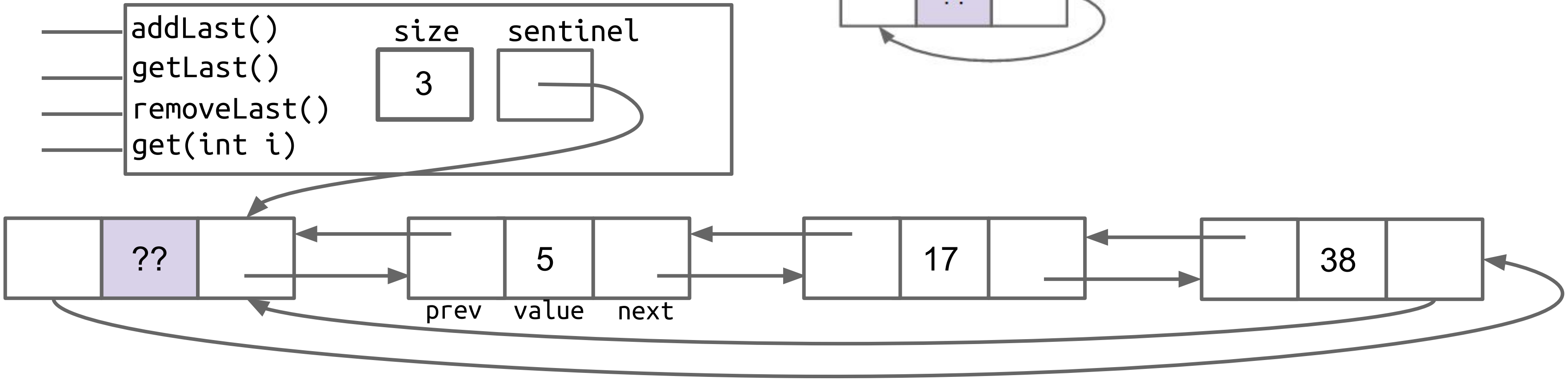
A Last Look at Linked Lists

Doubly Linked Lists

Behold. The state of the art as we arrived at in last week's lecture. Through various improvements, we made all of the following operations fast:

- `addFirst`, `addLast`
- `getFirst`, `getLast`
- `removeFirst`, `removeLast`
- You will build this in project 1A.

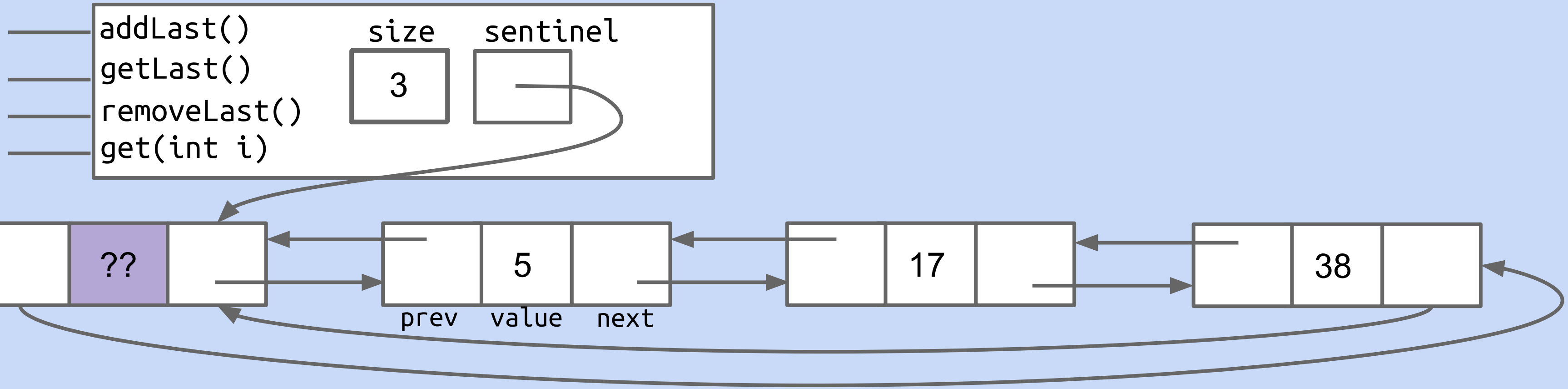




Arbitrary Retrieval

Suppose we added `get(int i)`, which returns the ith item from the list.

Why would `get` be slow for long lists compared to `getLast()`? For what inputs?

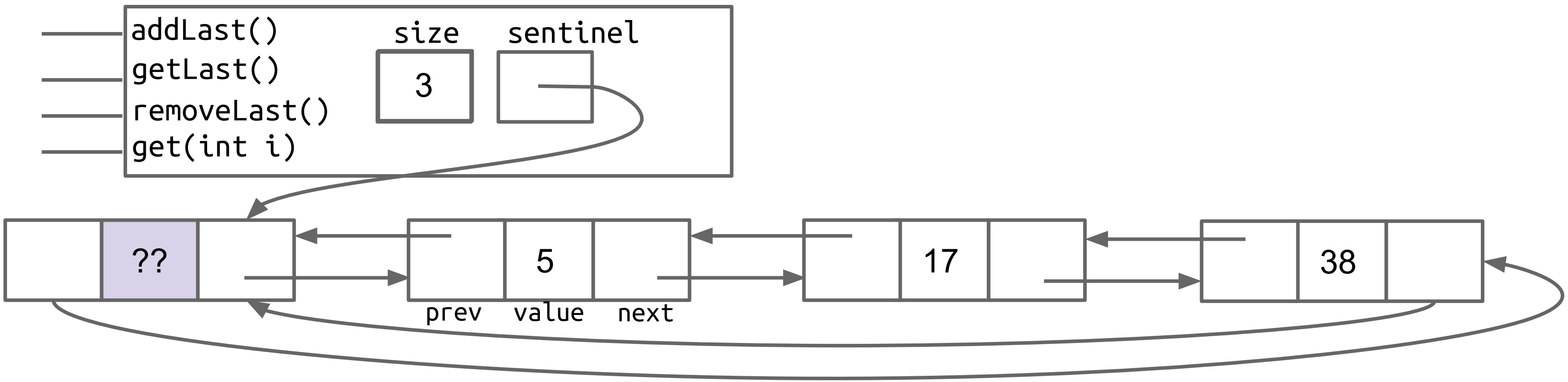


Arbitrary Retrieval

Suppose we added `get(int i)`, which returns the ith item from the list.

Why would `get` be slow for long lists compared to `getLast()`? For what inputs?

- Have to scan to desired position. Slow for any `i` not near the sentinel node.
- How do we fix this?

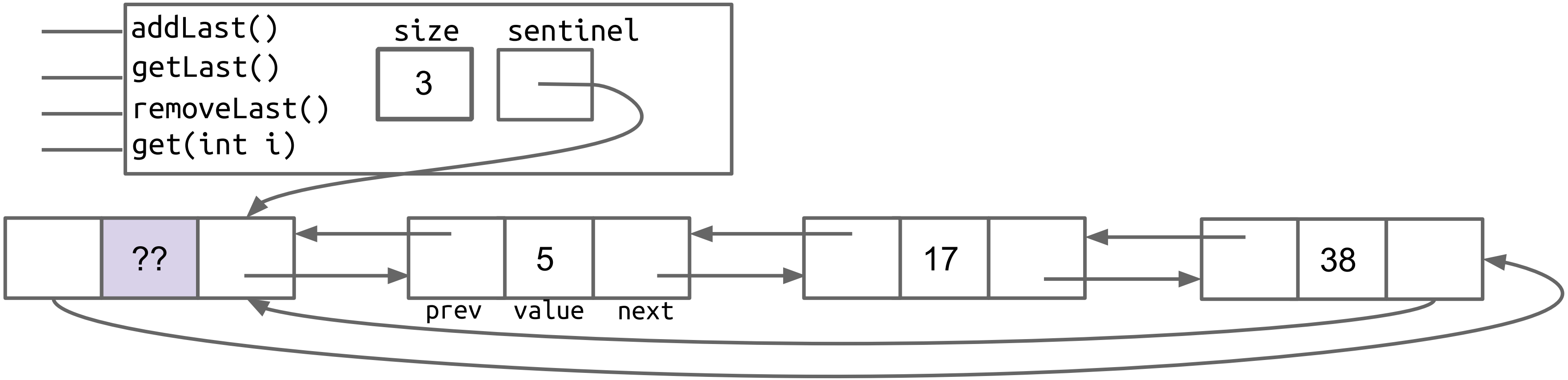


Arbitrary Retrieval

Suppose we added `get(int i)`, which returns the ith item from the list.

Why would `get` be slow for long lists compared to `getLast()`? For what inputs?

- Have to scan to desired position. Slow for any `i` not near the sentinel node.
- Will discuss (much later) sophisticated changes that can speed up lists.
- For now: We'll take a different tack: Using an array instead (no links!).

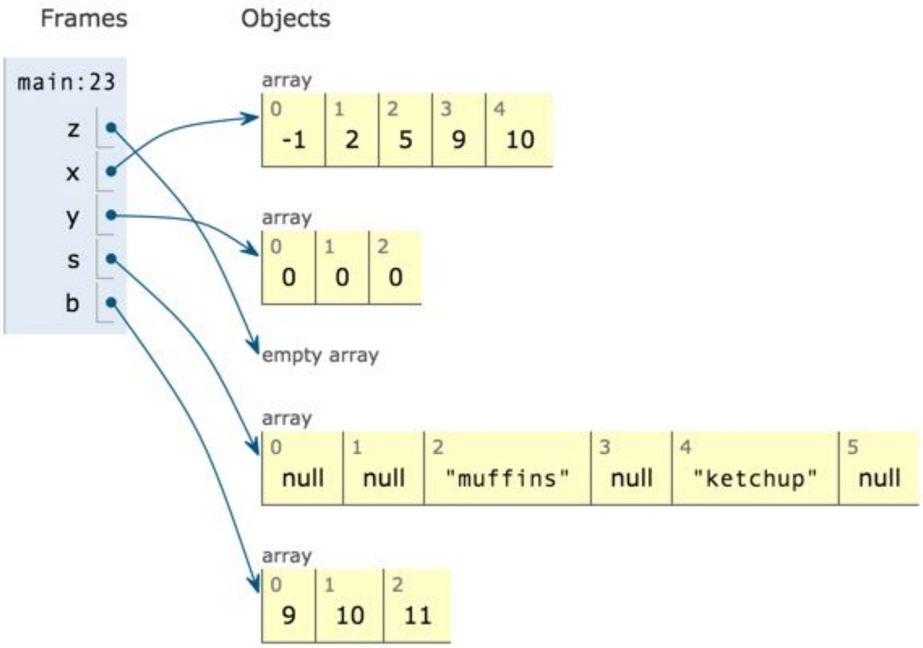


Naive Array Lists

Random Access in Arrays

Retrieval from any position of an array is very fast.

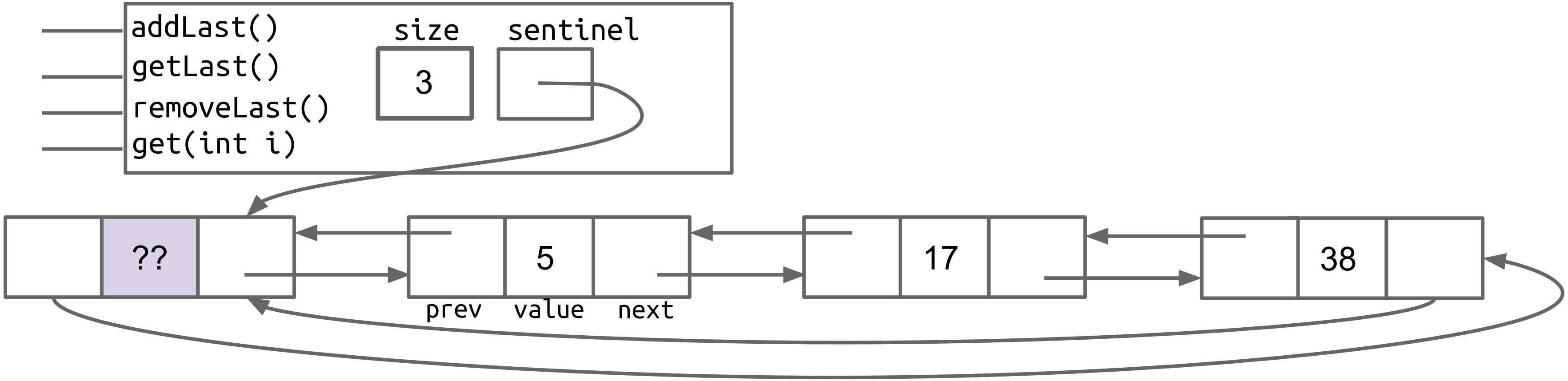
- Independent* of array size.
- 61C Preview: Ultra fast random access results from the fact that memory boxes are the same size (in bits).



Our Goal: AList.java

Want to figure out how to build an array version of a list:

- In lecture we'll only do back operations. Project 1A is the front operations.



Let's try it out...

Naive AList Code

From last lecture, “things that must be true”.

```
public class AList {
  private int[] items;
  private int size;

  public AList() {
    items = new int[100];  size = 0;
  }

  public void addLast(int x) {
    items[size] = x;
    size += 1;
  }

  public int getLast() {
    return items[size - 1];
  }

  public int get(int i) {
    return items[i];
  }

  public int size() {
    return size;
  }
}
```

AList **Invariants**:

- The position of the next item to be inserted is always `size`.
- size is always the number of items in the AList.
- The last item in the list is always in position `size - 1`.

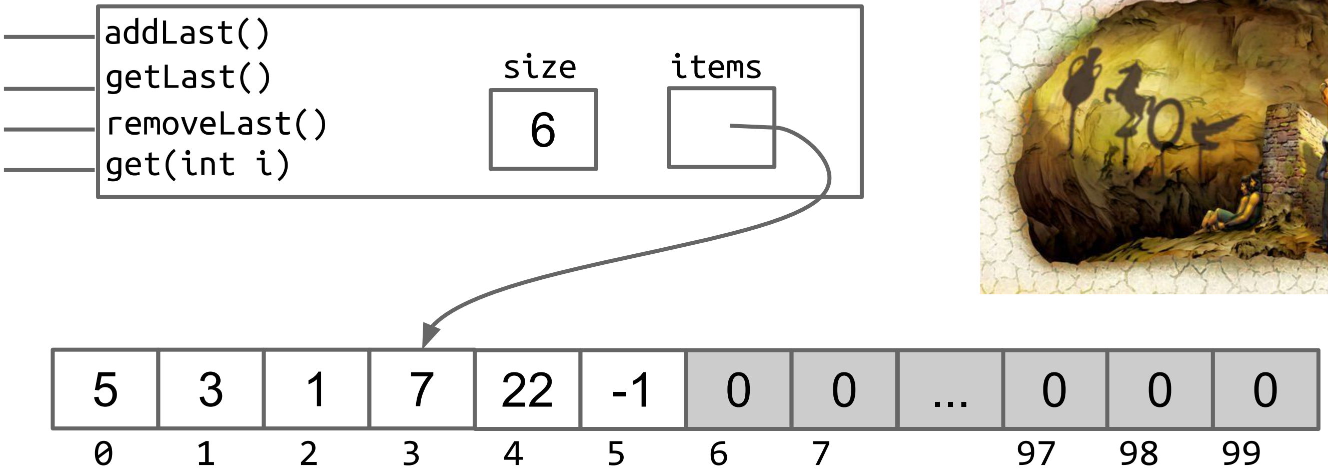
Let’s now discuss delete operations.

The Abstract vs. the Concrete

When we `removeLast()`, which memory boxes need to change? To what?

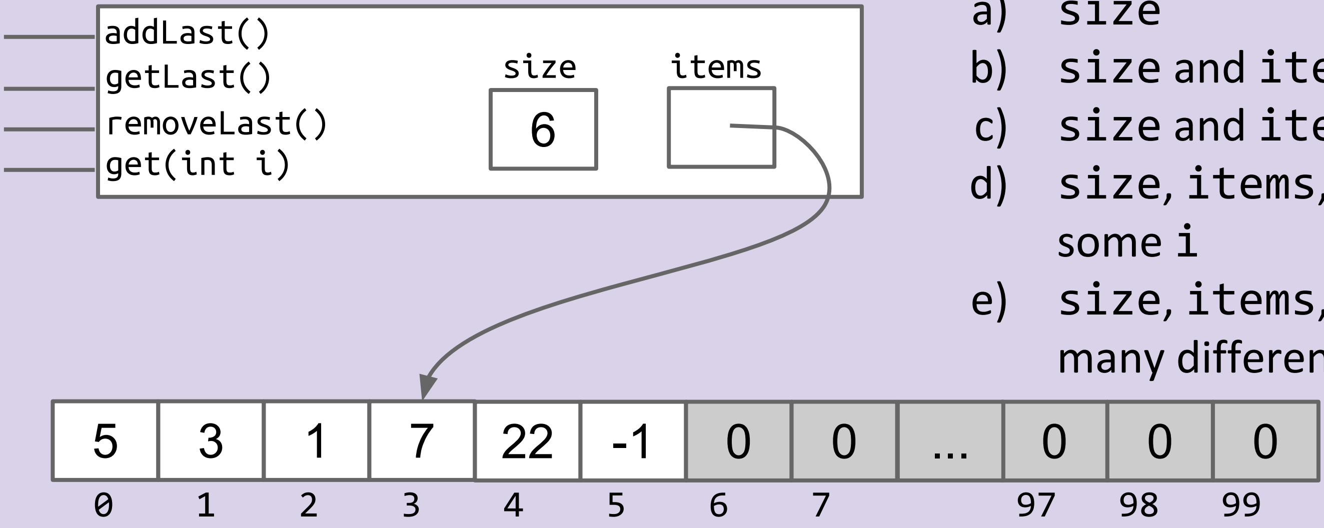
User's mental model: `{5, 3, 1, 7, 22, -1}` → `{5, 3, 1, 7, 22}`

Actual truth:



Deletion: yellkey.com/dark

When we `removeLast()`, which memory boxes need to change? To what?



a) `size`
b) `size` and `items`
c) `size` and `items[i]` for some `i`
d) `size`, `items`, and `items[i]` for some `i`
e) `size`, `items`, and `items[i]` for many different `i`

- The position of the next item to be inserted is always `size`.
- size is always the number of items in the AList.
- The last item in the list is always in position `size - 1`.

AList invariants.

Naive AList Code

```
public class AList {
  private int[] items;
  private int size;

  public AList() {
    items = new int[100];  size = 0;
  }

  public void addLast(int x) {
    items[size] = x;
    size += 1;
  }

  public int getLast() {
    return items[size - 1];
  }

  public int get(int i) {
    return items[i];
  }

  public int size() {
    return size;
  }
}
```

AList Invariants:

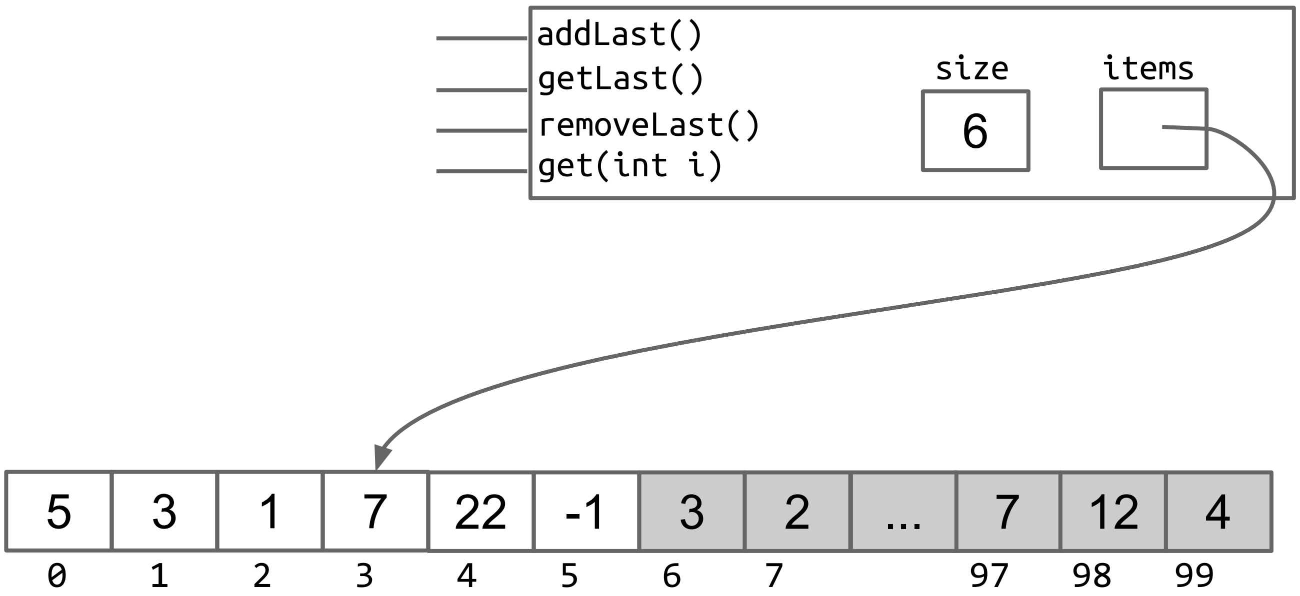
- The position of the next item to be inserted is always `size`.
- size is always the number of items in the AList.
- The last item in the list is always in position `size - 1`.

```
public int removeLast() {
  int returnItem = items[size - 1];
  items[size - 1] = 0;
  size -= 1;
  return returnItem;
}
```

Setting deleted item to zero is not necessary to preserve invariants, and thus not necessary for correctness.

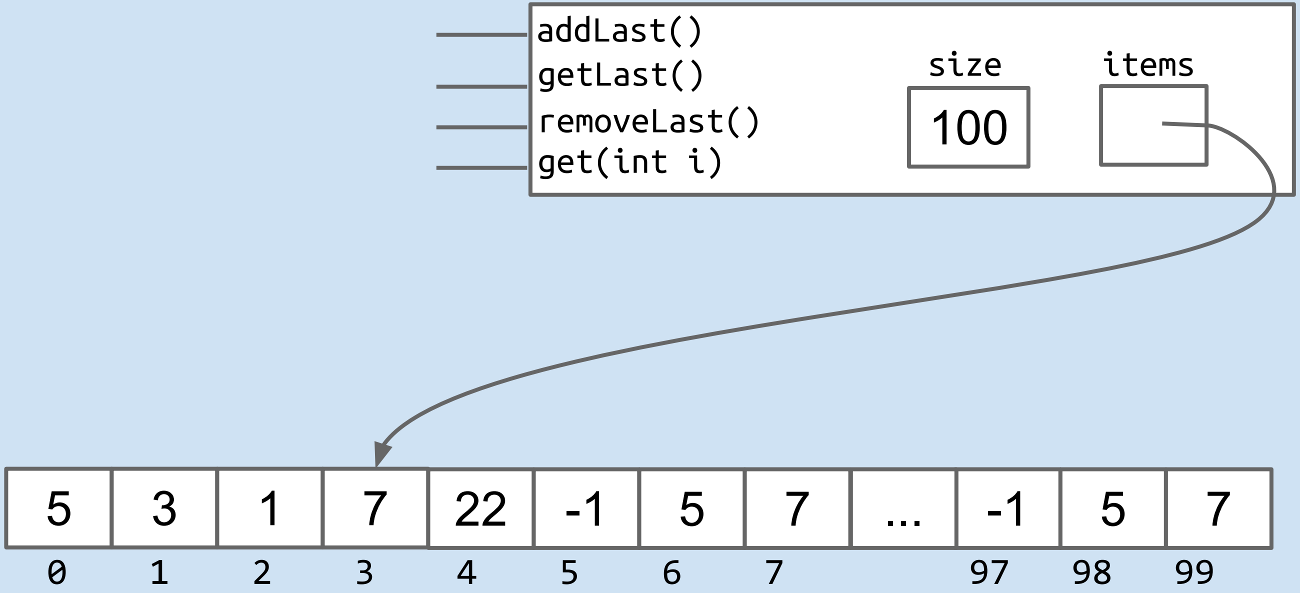
The Mighty AList

Key Idea: Use some subset of the entries of an array.



The Mighty (?) AList

Key Idea: Use some subset of the entries of an array.



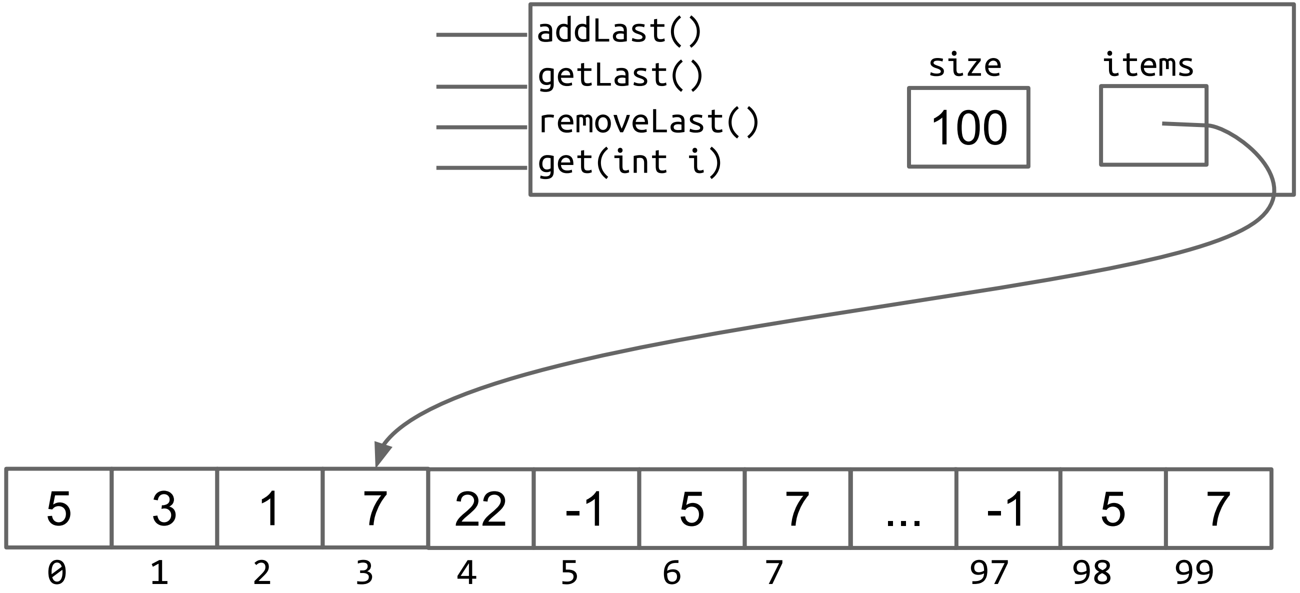
What happens if we insert into the AList above? What should we do about it?

Resizing Arrays

Array Resizing

size==items.length

When the array gets too full, e.g. addLast(11), just make a new array:

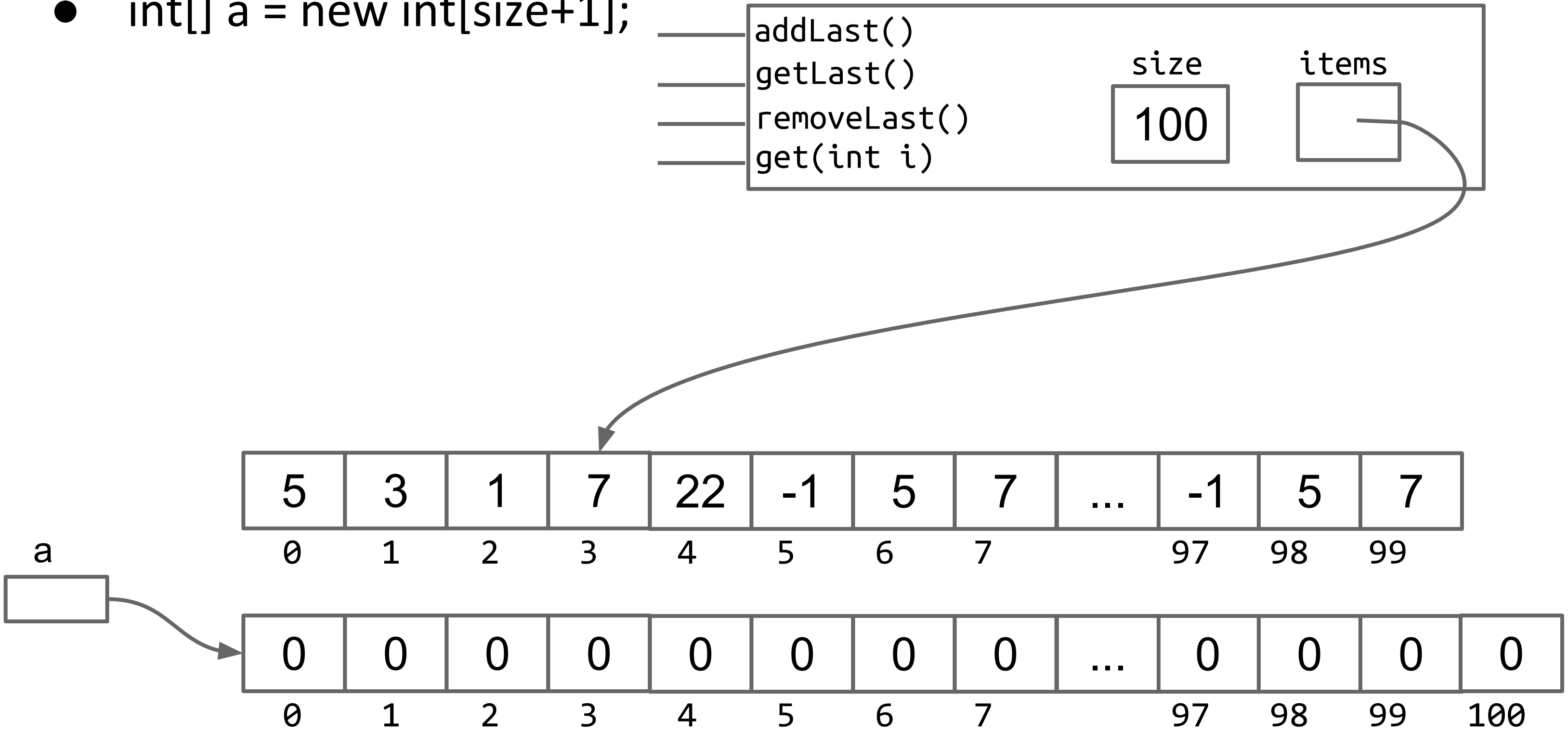


Array Resizing

size==items.length

When the array gets too full, e.g. addLast(11), just make a new array:

- int[] a = new int[size+1];

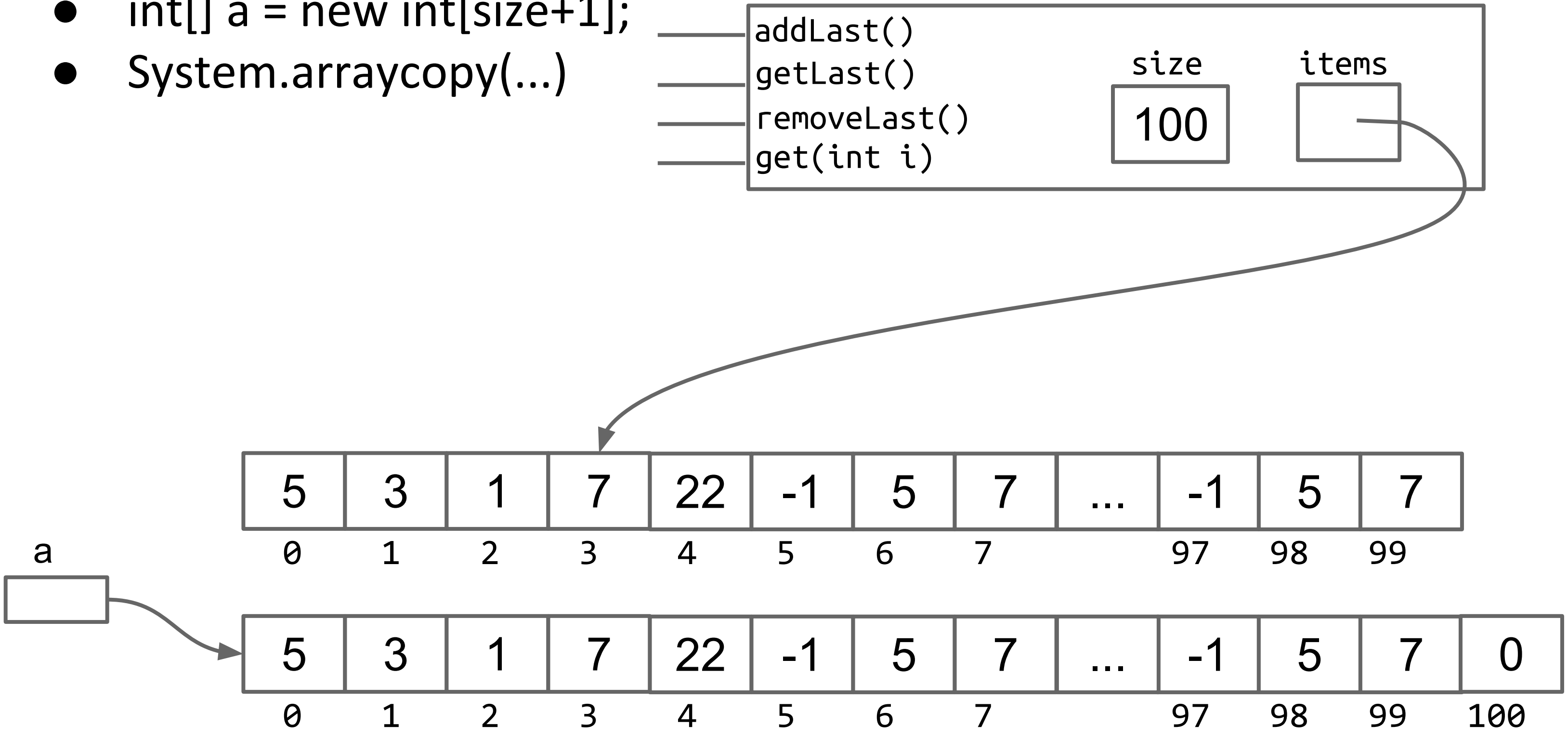


Array Resizing

size==items.length

When the array gets too full, e.g. addLast(11), just make a new array:

- int[] a = new int[size+1];
- System.arraycopy(...)

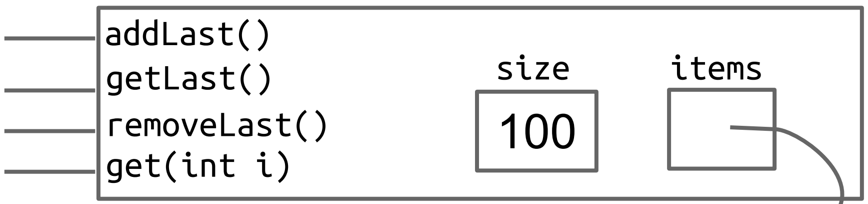


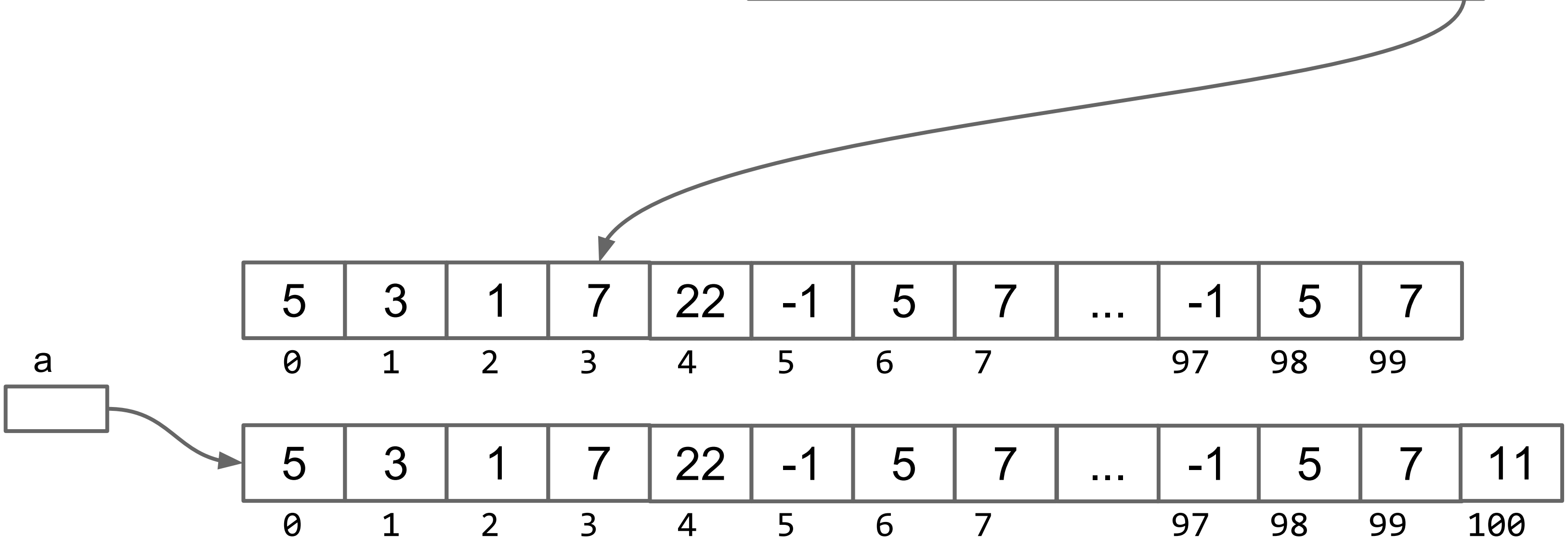
Array Resizing

size==items.length

When the array gets too full, e.g. addLast(11), just make a new array:

- int[] a = new int[size+1];
- System.arraycopy(...)
- a[size] = 11;



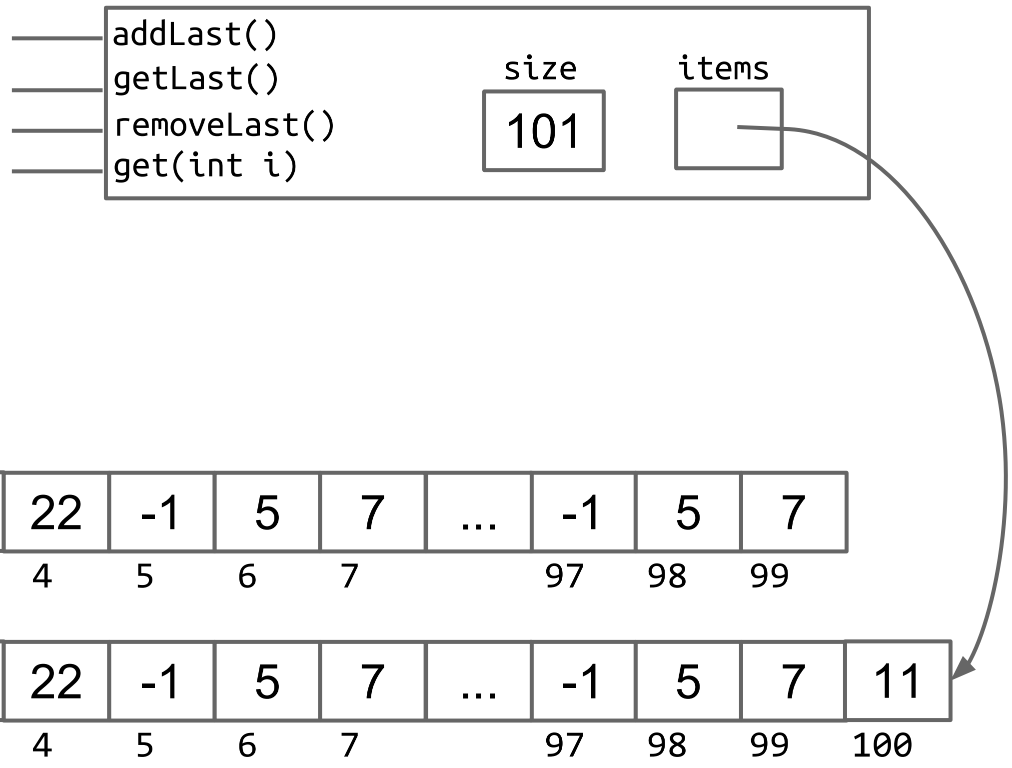


Array Resizing

size==items.length

When the array gets too full, e.g. addLast(11), just make a new array:

- int[] a = new int[size+1];
- System.arraycopy(...)
- a[size] = 11;
- items = a;   size +=1;



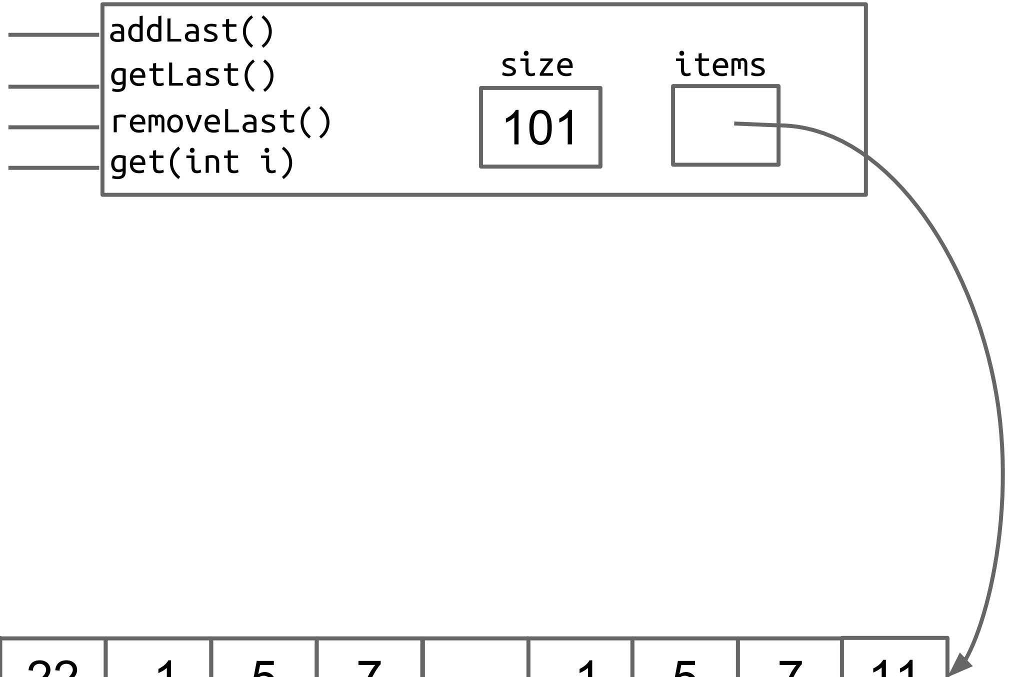
Array Resizing

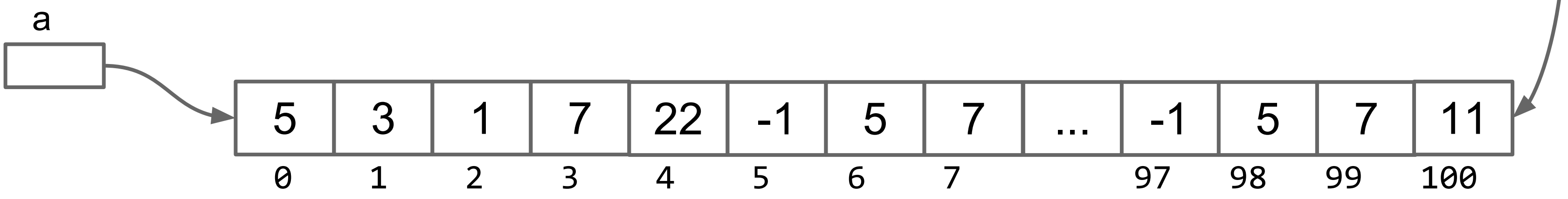
size==items.length

When the array gets too full, e.g. addLast(11), just make a new array:

- int[] a = new int[size+1];
- System.arraycopy(...)
- a[size] = 11;
- items = a;   size +=1;

We call this process "resizing"





Implementation

Let’s implement the resizing capability.

- As usual, for those of you watching online, I recommend trying to implement this on your own before watching me do it.
- Starter code is provided in the lists4 study guide if you want to try it out on a computer.

Resizing Array Code

```java
public void addLast(int x) {
  if (size == items.length) {
    int[] a = new int[size + 1];
    System.arraycopy(items, 0, a, 0, size);
    items = a;
  }
  items[size] = x;
  size += 1;
}
```

Works

```java
private void resize(int capacity) {
  int[] a = new int[capacity];
  System.arraycopy(items, 0, a, 0, size);
  items = a;
}

public void addLast(int x) {
  if (size == items.length) {
     resize(size + 1);
  }
  items[size] = x;
  size += 1;
}
```

Much Better

Runtime and Space Usage Analysis: yellkey.com/protect

Suppose we have a full array of size 100. If we call `addLast` two times, how many **total** array memory boxes will we need to create and fill (for just these 2 calls)?

A. 0
B. 101
C. 203
D. 10,302

Bonus question: What is the maximum number of array boxes that Java will track at any given time? Assume that "garbage collection" happens immediately when all references to an object are lost.

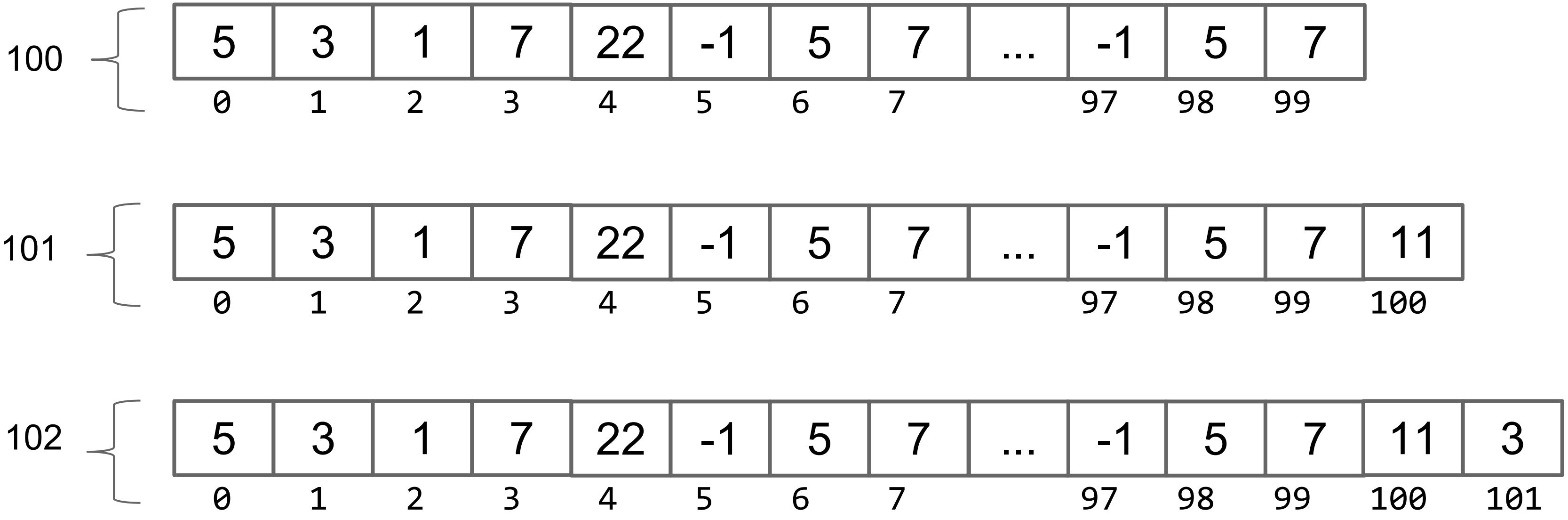
```java
private void resize(int capacity) {
  int[] a = new int[capacity];
  System.arraycopy(items, 0, a, 0, size);
  items = a;
}

public void addLast(int x) {
  if (size == items.length) {
    resize(size + 1);
  }
  items[size] = x;
  size += 1;
}
```

Array Resizing

Resizing twice requires us to create and fill 203 total memory boxes.

- Bonus answer: Most boxes at any one time is 203.
- When the second `addLast` is done, we are left with 102 boxes.



Runtime and Space Usage Analysis: yellkey.com/protect

Suppose we have a full array of size 100. If we call `addLast` until size = 1000, roughly how many total array memory boxes will we need to create and fill?

A. 1,000
B. 500,000
C. 1,000,000
D. 500,000,000,000
E. 1,000,000,000,000

Bonus question: What is the maximum number of array boxes that Java will track at any given time? Assume that "garbage collection" happens immediately when all references to an object are lost.

```
private void resize(int capacity) {
  int[] a = new int[capacity];
  System.arraycopy(items, 0, a, 0, size);
  items = a;
}

public void addLast(int x) {
  if (size == items.length) {
    resize(size + 1);
  }
  items[size] = x;
  size += 1;
}
```

Runtime and Space Usage Analysis

Suppose we have a full array of size 100. If we call `addLast` until size = 1000, roughly how many total array memory boxes will we need to create and fill?

B. 500,000

```
private void resize(int capacity) {
  int[] a = new int[capacity];
  System.arraycopy(items, 0, a, 0, size);
  items = a;
}
```

Going from capacity 100 to 101: 101

From 101 to 102: 102

…

From: 999 to 1000: 1000

We'll be doing a lot of this after the midterm.

Total array boxes created/copied: 101 + 102 + … + 1000

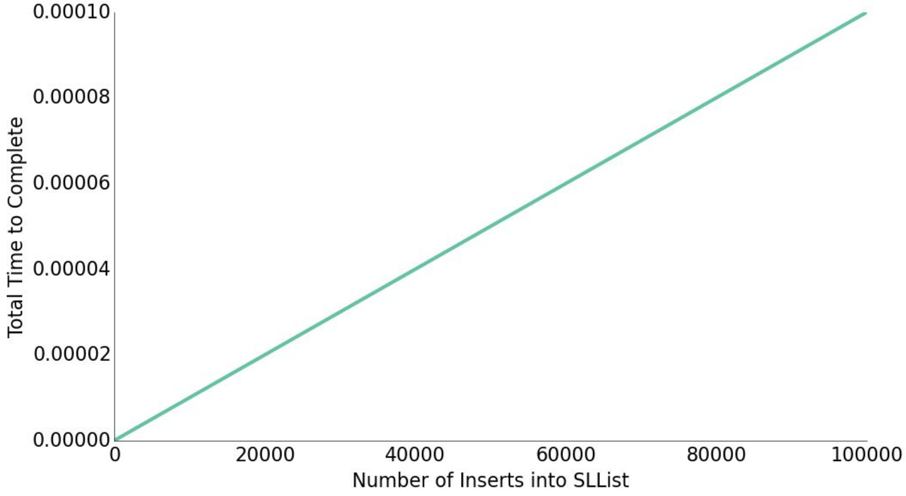
Since sum of 1 + 2 + 3 + … + N = N(N+1)/2, sum(101, …, 1000) is close to 500,000.

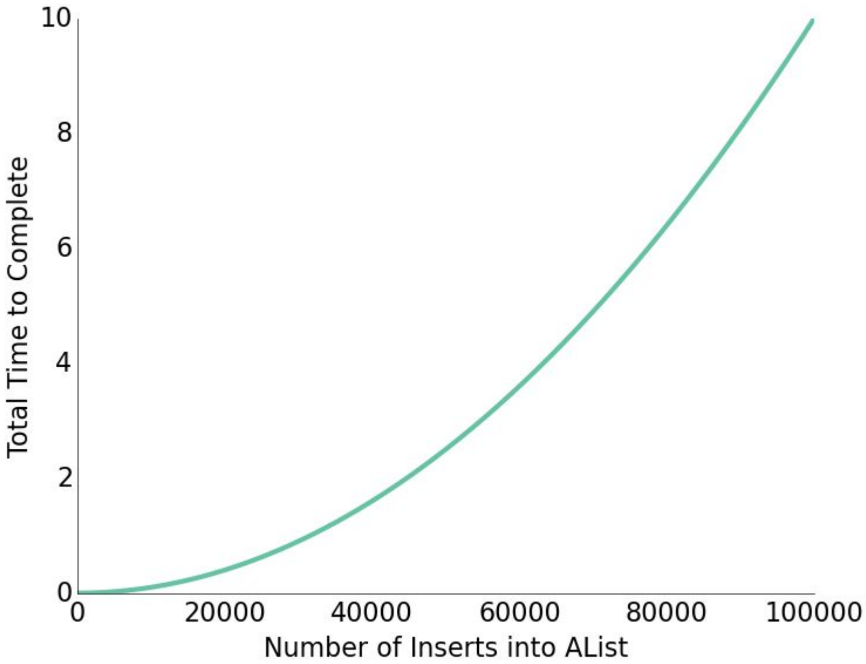
See: http://mathandmultimedia.com/2010/09/15/sum-first-n-positive-integers

Resizing Slowness

Inserting 100,000 items requires roughly 5,000,000,000 new containers.

- Computers operate at the speed of GHz (due billions of things per second).
- No huge surprise that 100,000 items took seconds.





Fixing the Resizing Performance Bug

How do we fix this?

```
private void resize(int capacity) {
  int[] a = new int[capacity];
  System.arraycopy(items, 0, a, 0, size);
  items = a;
}

public void addLast(int x) {
  if (size == items.length) {
     resize(size + 1);
  }
  items[size] = x;
  size += 1;
}
```

(Probably) Surprising Fact

Geometric resizing is much faster: Just how much better will have to wait.

```
public void addLast(int x) {
  if (size == items.length) {
    resize(size + RFACTOR);
  }
  items[size] = x;
  size += 1;
}
```

Unusably bad.

Great performance.

This is how the Python list is implemented.

```
public void addLast(int x) {
  if (size == items.length) {
    resize(size * RFACTOR);
  }
  items[size] = x;
  size += 1;
}
```

Performance Problem #2

Suppose we have a very rare situation occur which causes us to:

- Insert 1,000,000,000 items.
- Then remove 990,000,000 items.

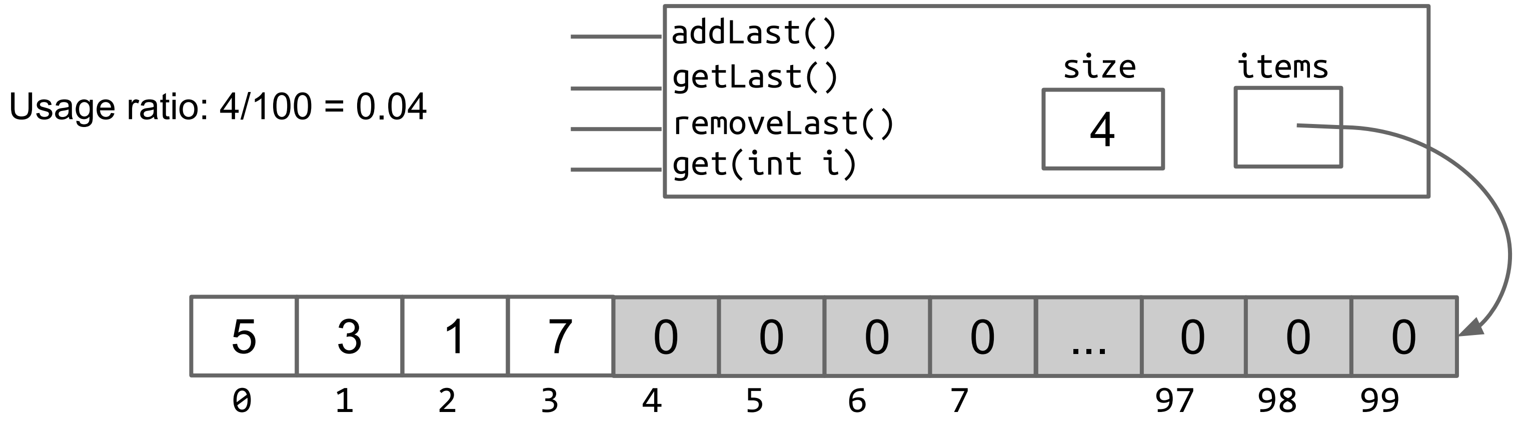
Our data structure will handle this spike of events as well as it could, but afterwards there is a problem.

- What is the problem?

Memory Efficiency

An AList should not only be efficient in time, but also efficient in space.

- Define the "usage ratio" R = size / items.length;
- Typical solution: Half array size when R < 0.25.
- More details in a few weeks.



Later we will consider tradeoffs between time and space efficiency for a variety of algorithms and data structures.

Generic ALists

Generic ALists (similar to generic SLists)

```java
public class AList {
  private int[] items;
  private int size;

  public AList() {
    items = new int[8];
    size = 0;
  }

  private void resize(int capacity) {
    int[] a = new int[capacity];
    System.arraycopy(items, 0,
                     a, 0, size);
    items = a;
  }

  public int get(int i) {
    return items[i];
  }
...
```

```java
public class AList<Glorp> {
  private Glorp[] items;
  private int size;

  public AList() {
    items = (Glorp []) new Object[8];
    size = 0;
  }

  private void resize(int cap) {
    Glorp[] a = (Glorp []) new Object[cap];
    System.arraycopy(items, 0,
                     a, 0, size);
    items = a;
  }

  public Glorp get(int i) {
    return items[i];
  }
...
```

Generic ALists (similar to generic SLists)

```java
public class AList<Glorp> {
  private Glorp[] items;
  private int size;

  public AList() {
    items = (Glorp []) new Object[8];
    size = 0;
  }

  private void resize(int cap) {
    Glorp[] a = (Glorp []) new Object[cap];
    System.arraycopy(items, 0,
                     a, 0, size);
    items = a;
  }

  public Glorp get(int i) {
    return items[i];
  }
...
```

When creating an array of references to Glorps:

- `(Glorp []) new Object[cap];`
- Causes a compiler warning, which you should ignore.

Why not just `new Glorp[cap];`

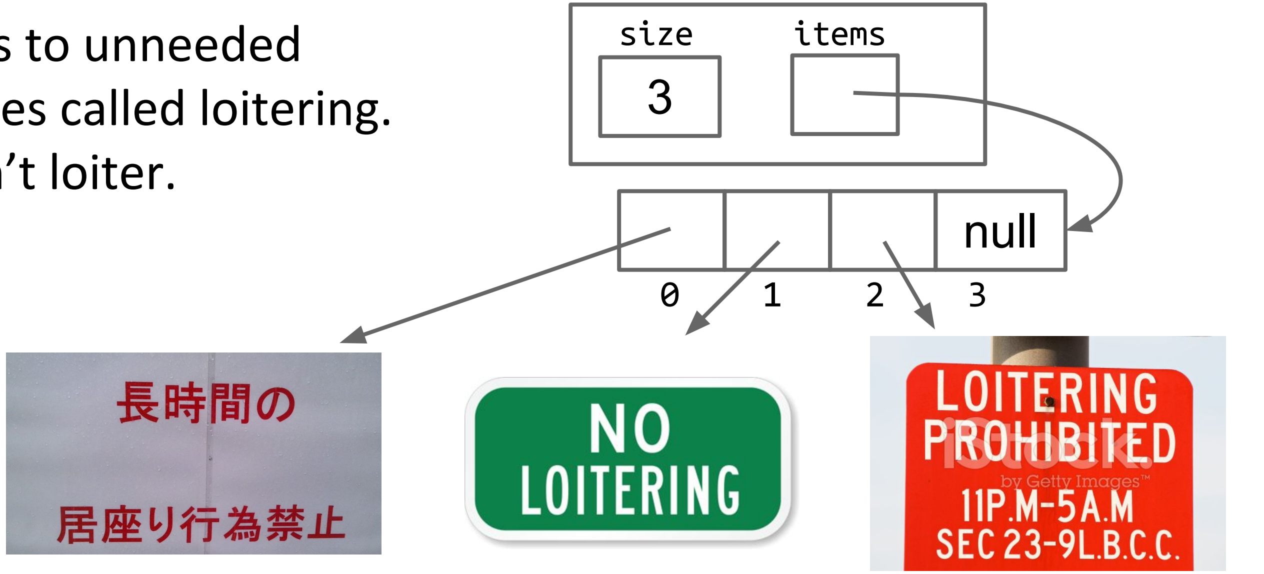
- Will cause a “generic array creation” error.
- Will discuss in a few weeks.

Nulling Out Deleted Items

Unlike integer based ALists, we actually want to null out deleted items.

- Java only destroys unwanted objects when the last reference has been lost.
- Keeping references to unneeded objects is sometimes called loitering.
- Save memory. Don't loiter.

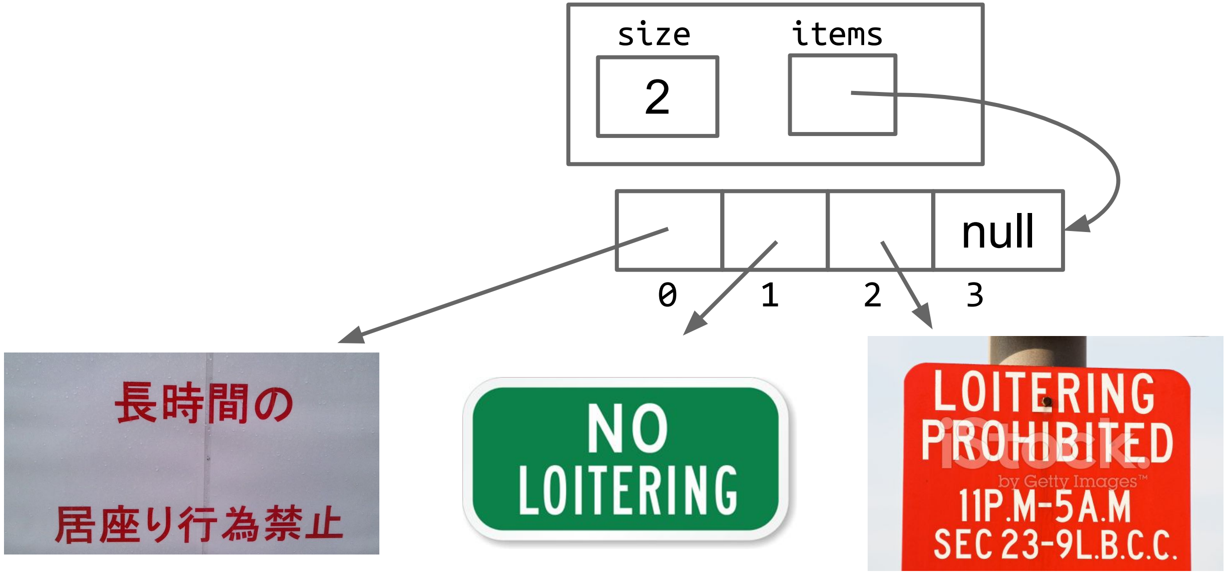
```
public Glorp deleteBack() {
  Glorp returnItem = getBack();
  items[size - 1] = null;
  size -= 1;
  return returnItem;
}
```



Loitering Example

Changing size to 2 yields a correct AList.

- But memory is wasted storing a reference to the red sign image.

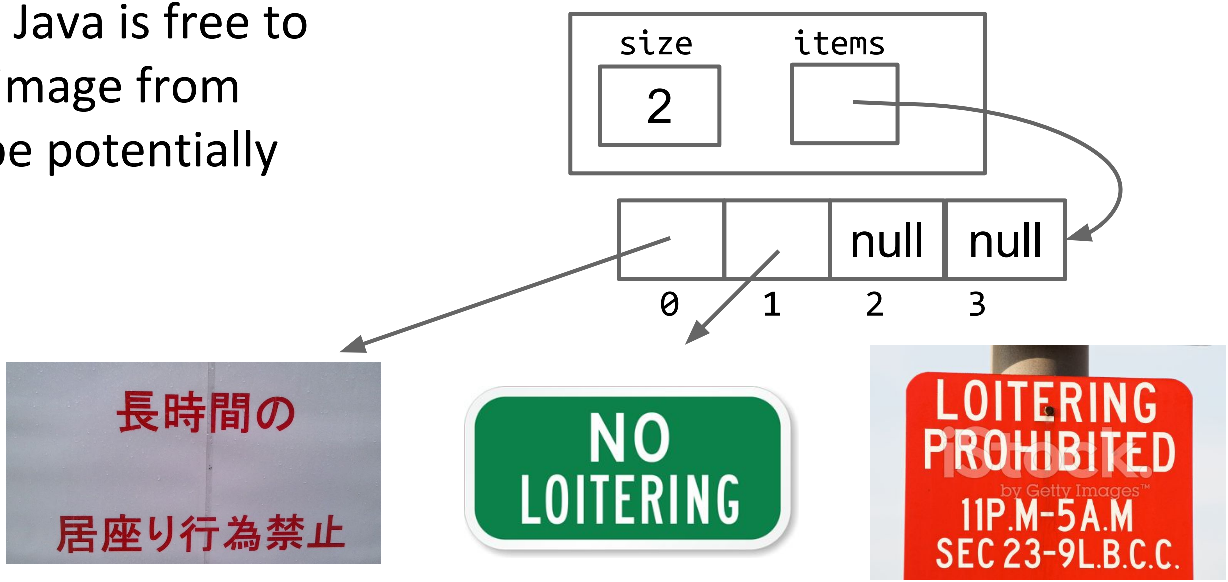


Loitering Example

Changing size to 2 yields a correct AList.

- But memory is wasted storing a reference to the red sign image.

By nulling out items[2], Java is free to destroy the unneeded image from memory, which could be potentially megabytes in size.

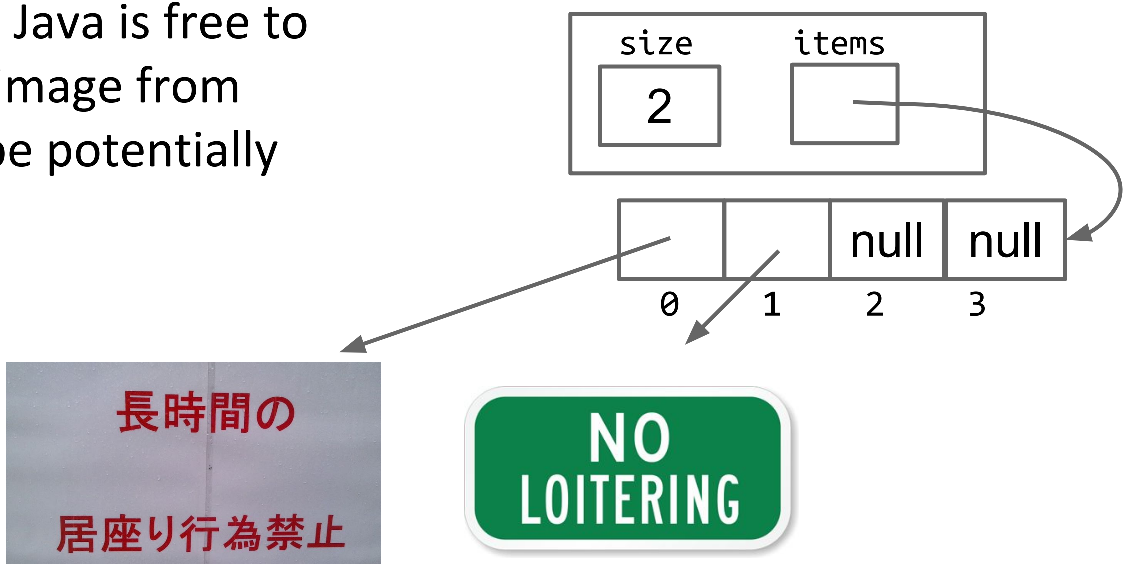


Loitering Example

Changing size to 2 yields a correct AList.

- But memory is wasted storing a reference to the red sign image.

By nulling out items[2], Java is free to destroy the unneeded image from memory, which could be potentially megabytes in size.



Obscurantism in Java

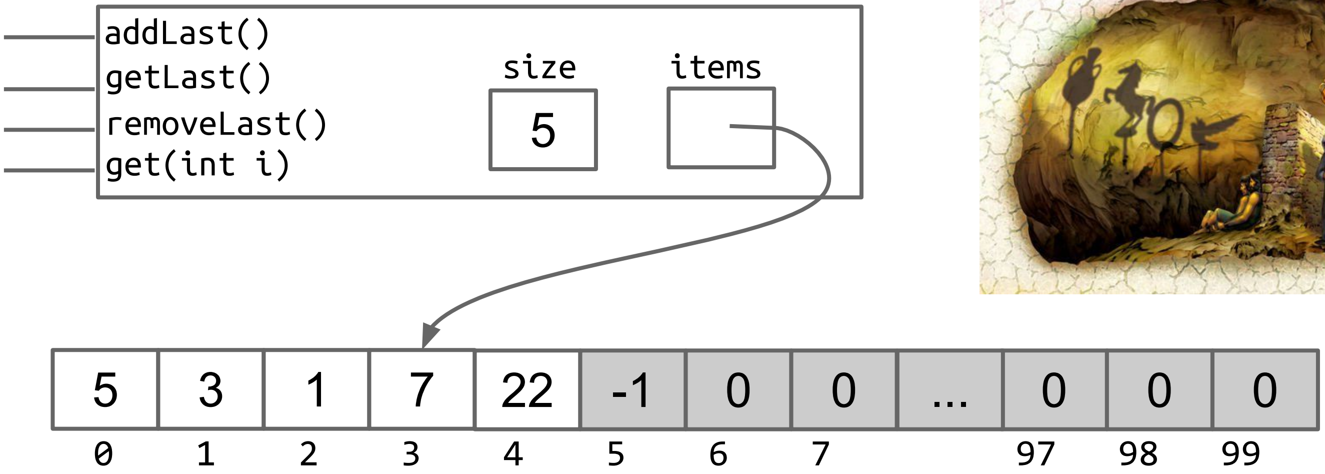
One last thought: Obscurantism in Java

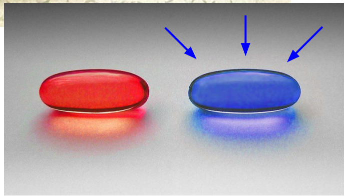
We talk of “layers of abstraction” often in computer science.

- We also rely on obscurantism. The user of a class does not and should not know how it works.

User’s mental model: `{5, 3, 1, 7, 22, -1}` → `{5, 3, 1, 7, 22}`

Actual truth:

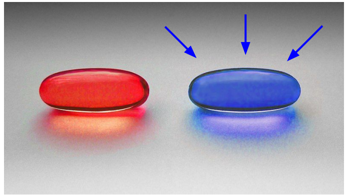




One last thought: Obscurantism in Java

We talk of "layers of abstraction" often in computer science.

- We also rely on obscurantism. The user of a class does not and should not know how it works.
  - The Java language allows you to enforce this with ideas like **`private`**!
- A good programmer obscures details from themselves, even <u>within a class</u>.
  - Example: `addFirst` and `resize` should be written totally independently. You should not be thinking about the details of one method while writing the other. Simply trust that the other works.
  - Breaking programming tasks down into small pieces (especially functions) helps with this greatly!
  - Through judicious use of testing, we can build confidence in these small pieces, as we'll see in the next lecture.



Citations

Hanging Containers:
http://www.portcalls.com/wp-content/uploads/2012/04/hanging_containers1.jpg

Loitering:
http://i.istockimg.com/file_thumbview_approve/19711163/6/stock-photo-19711163-red-loitering-prohibited-sign.jpg

http://images.mysecuritysign.com/img/lg/K/No-Loitering-Sign-K-5418.gif
http://3.bp.blogspot.com/-NV3y2NQDFy0/UAAXB5gINoI/AAAAAAAALi8/F_bM4-dmsm4/s1600/DVC00575.JPG

Red pill/blue pill: https://en.wikipedia.org/wiki/Red_pill_and_blue_pill