Lecture 7

Arrays and Lists

CS61B, Spring 2024 @ UC Berkeley

Slides credit: Josh Hug

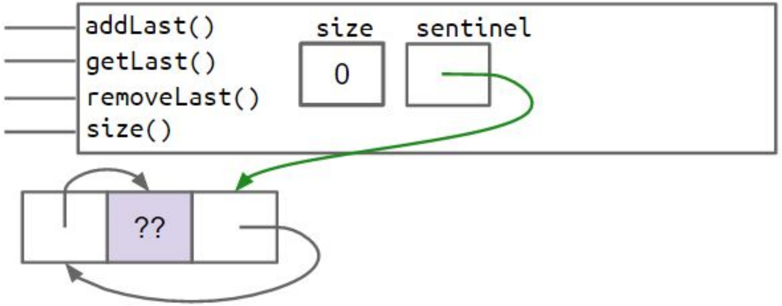
A Last Look at Linked Lists

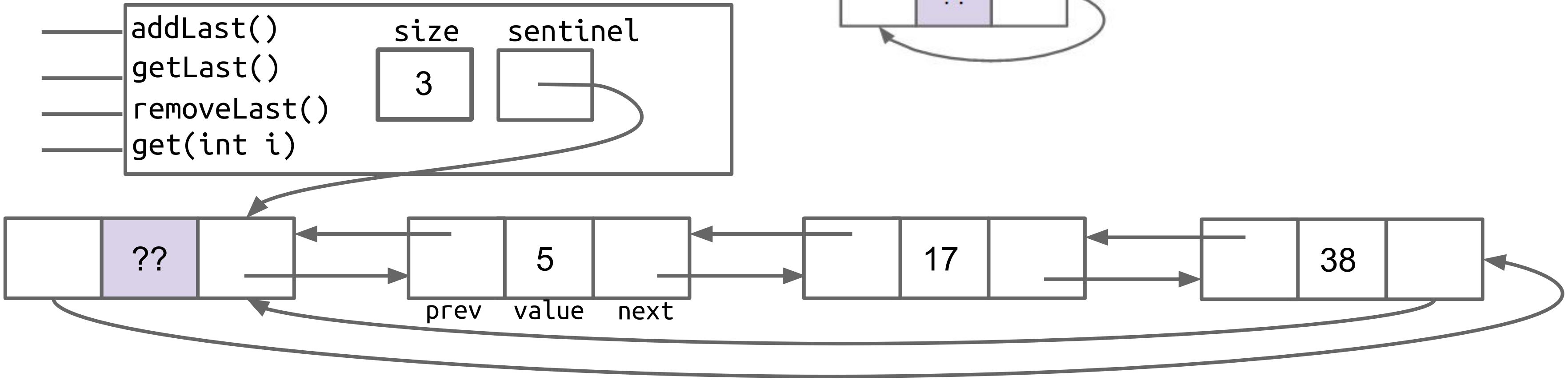
Lecture 7, CS61B, Spring 2024

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.

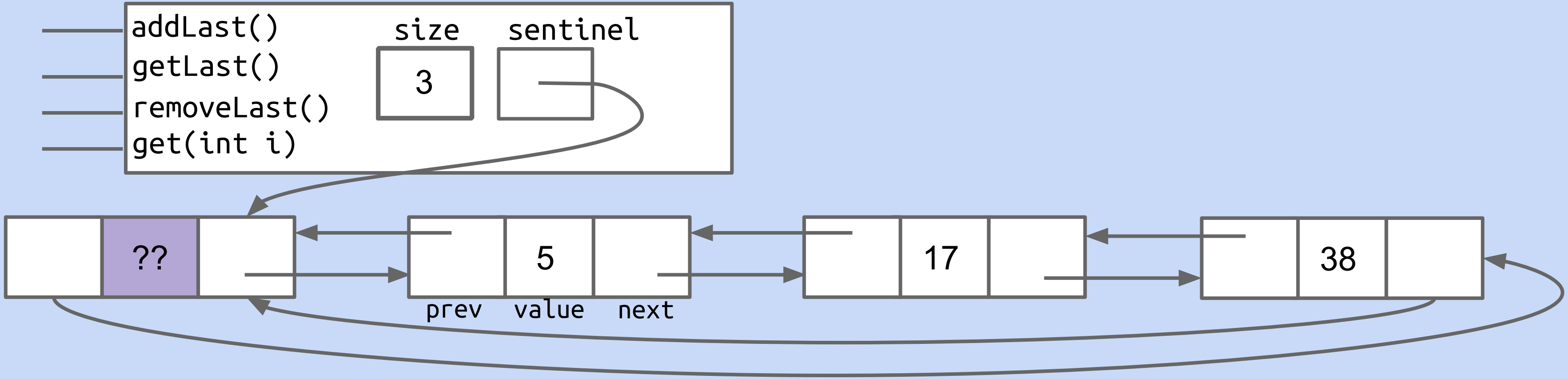




Arbitrary Retrieval

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?

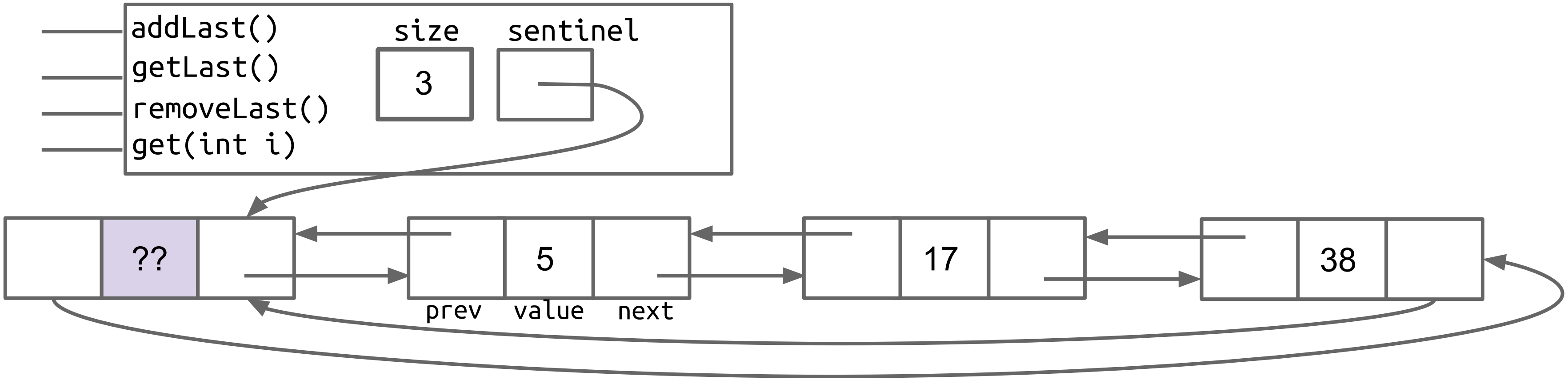


Arbitrary Retrieval

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?

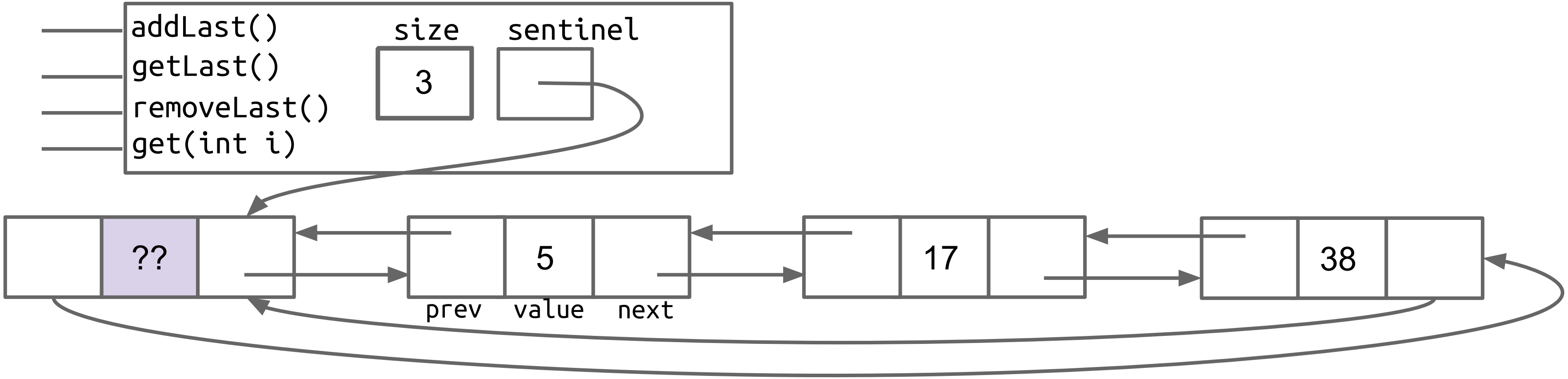


Arbitrary Retrieval

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 things up.
- For today: We'll take a different tack: Using an array instead (no links!).



Basic ArrayList Implementation

Lecture 7, CS61B, Spring 2024

A Last Look at Linked Lists

Naive ArrayLists

- **Basic ArrayList Implementation**
- The Allegory of the Cave
- removeLast Implementation

Resizing ArrayLists

- Resizing Array Theory
- Resizing Array Implementation
- Runtime Analysis (Warmup)
- Runtime Analysis
- Better Resizing Strategy

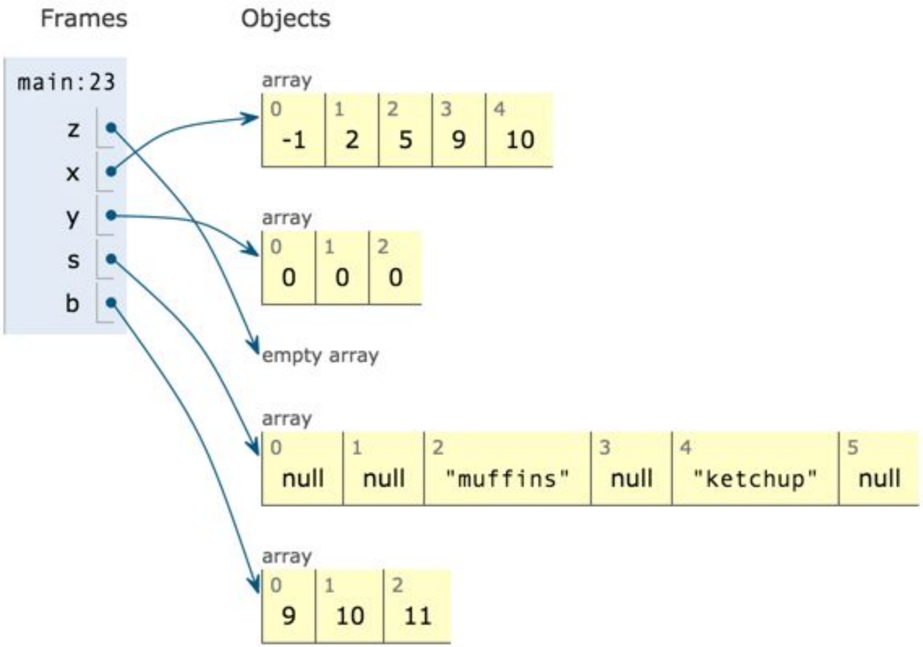
Generic ArrayLists

Obscurantism in Java

Random Access in Arrays

Retrieval from any position of an array is very fast.

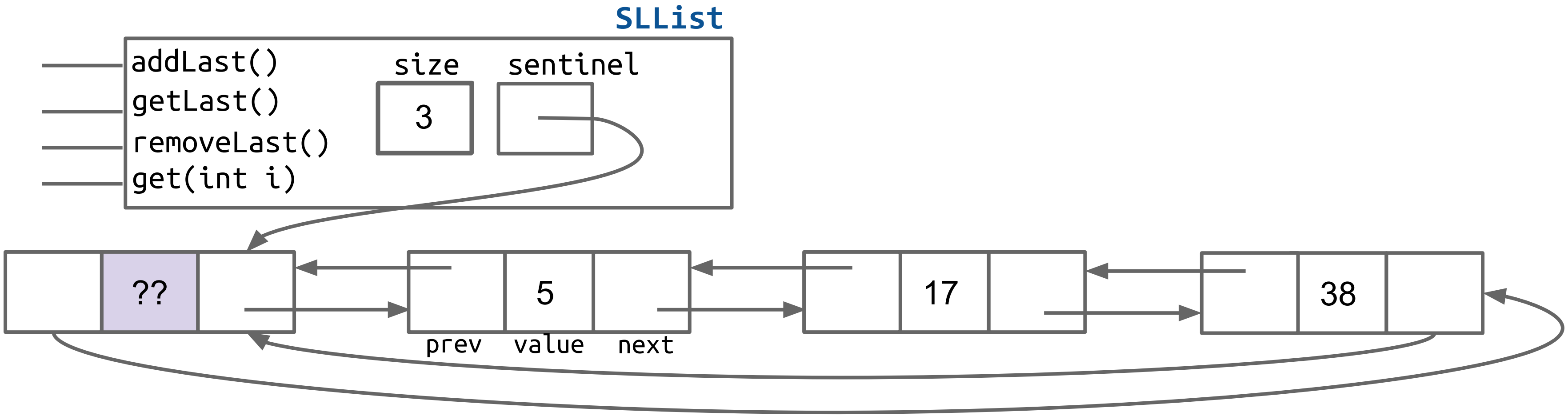
- Independent* of array size.
- 61C Preview: Ultra fast random access results from the fact that memory boxes are the same size (in bits).

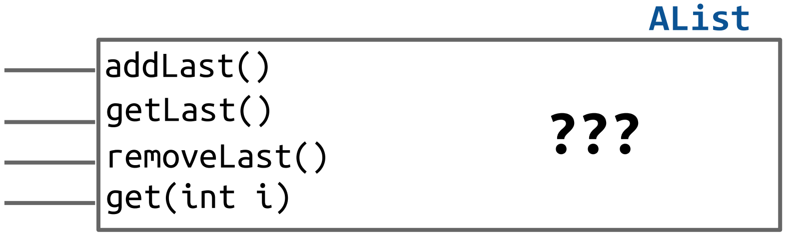


Our Goal: AList.java

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

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





Let's try it out...

Coding Demo: Basic ArrayList Constructor

AList.java

```
public class AList {

    /** Creates an empty list. */
    public AList() {

    }

}
```

Coding Demo: Basic ArrayList Constructor

AList.java

```java
public class AList {

    private int size;

    /** Creates an empty list. */
    public AList() {


    }


}
```

Coding Demo: Basic ArrayList Constructor

AList.java

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

    /** Creates an empty list. */
    public AList() {


    }


}
```

Coding Demo: Basic ArrayList Constructor

AList.java

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

    /** Creates an empty list. */
    public AList() {
        items = new int[100];

    }


}
```

The choice of array size (100) was arbitrary. We'll fix this limitation later.

Coding Demo: Basic ArrayList Constructor

AList.java

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

    /** Creates an empty list. */
    public AList() {
        items = new int[100];
        size = 0;
    }


}
```

Coding Demo: Basic ArrayList addLast

AList.java

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

    /** Inserts x into the back of the list. */
    public void addLast(int x) {


    }


}
```

Let's write a small example to help us think about addLast.

Coding Demo: Basic ArrayList addLast

Call constructor to get empty array.

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	...

size=0

addLast(6)

6	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	...

size=1

addLast(9)

6	9	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	...

size=2

addLast(-1)

6	9	-1	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	...

size=3

What patterns do we spot?

The next item we want to add will go into position size.

Coding Demo: Basic ArrayList addLast

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size



*/
public class AList {
   private int[] items;
   private int size;

   /** Inserts x into the back of the list. */
   public void addLast(int x) {


   }


}
```

Coding Demo: Basic ArrayList addLast

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size



*/
public class AList {
   private int[] items;
   private int size;

   /** Inserts x into the back of the list. */
   public void addLast(int x) {
       items[size] = x;

   }


}
```

Coding Demo: Basic ArrayList addLast

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size



*/
public class AList {
   private int[] items;
   private int size;

   /** Inserts x into the back of the list. */
   public void addLast(int x) {
       items[size] = x;
       size += 1;
   }


}
```

Coding Demo: Basic ArrayList getLast

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size



*/
public class AList {
   private int[] items;
   private int size;

   /** Returns the item from the back of the list. */
   public int getLast() {

   }



}
```

Coding Demo: Basic ArrayList getLast

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size
    getLast: The item we want to return is in position size - 1.

*/
public class AList {
   private int[] items;
   private int size;

   /** Returns the item from the back of the list. */
   public int getLast() {

   }



}
```

Coding Demo: Basic ArrayList getLast

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size
    getLast: The item we want to return is in position size - 1.

*/
public class AList {
   private int[] items;
   private int size;

   /** Returns the item from the back of the list. */
   public int getLast() {
       return items[size - 1];
   }



}
```

Coding Demo: Basic ArrayList get

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size
    getLast: The item we want to return is in position size - 1.

*/
public class AList {
   private int[] items;
   private int size;

   /** Gets the ith item in the list (0 is the front). */
   public int get(int i) {

   }



}
```

Coding Demo: Basic ArrayList get

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size
    getLast: The item we want to return is in position size - 1.

*/
public class AList {
   private int[] items;
   private int size;

   /** Gets the ith item in the list (0 is the front). */
   public int get(int i) {
       return items[i];
   }



}
```

Coding Demo: Basic ArrayList size

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size
    getLast: The item we want to return is in position size - 1.

*/
public class AList {
   private int[] items;
   private int size;

   /** Returns the number of items in the list. */
   public int size() {

   }



}
```

Coding Demo: Basic ArrayList size

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size
    getLast: The item we want to return is in position size - 1.
    size: The number of items in the list should be size.
*/
public class AList {
   private int[] items;
   private int size;

   /** Returns the number of items in the list. */
   public int size() {

   }



}
```

Coding Demo: Basic ArrayList size

AList.java

```java
/** Invariants:
    addLast: The next item we want to add, will go into position size
    getLast: The item we want to return is in position size - 1.
    size: The number of items in the list should be size.
*/
public class AList {
   private int[] items;
   private int size;

   /** Returns the number of items in the list. */
   public int size() {
       return size;
   }




}
```

Naive AList Code

From SLList 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`.

We could also add error checking code, e.g.

```
public int get(int i) {
   if (i >= items.length) {
       throw new IllegalArgumentException();
   }
   return items[i];
}
```

Naive AList Code

From SLList 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 Allegory of the Cave

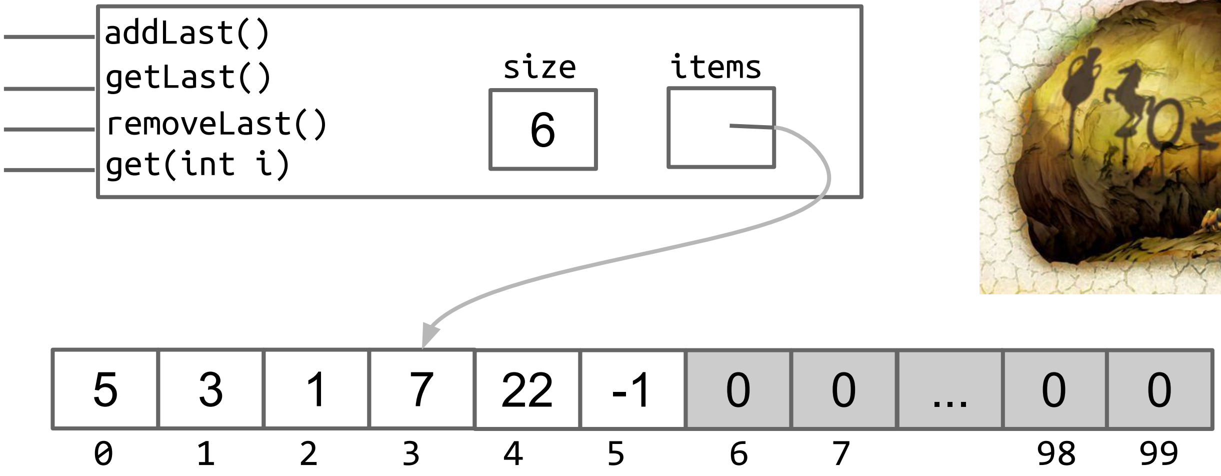
Lecture 7, CS61B, Spring 2024

The Abstract vs. the Concrete

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

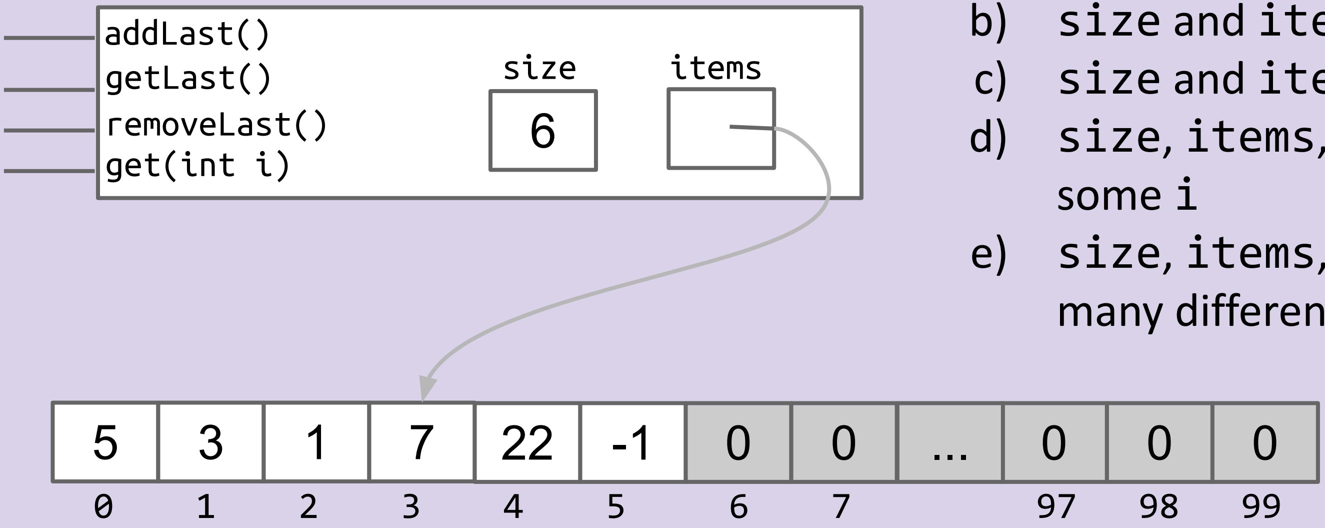
User's mental model: `{5, 3, 1, 7, 22, -1}` → `{5, 3, 1, 7, 22}`

Actual truth:



Deletion: yellkey.com/look

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`.

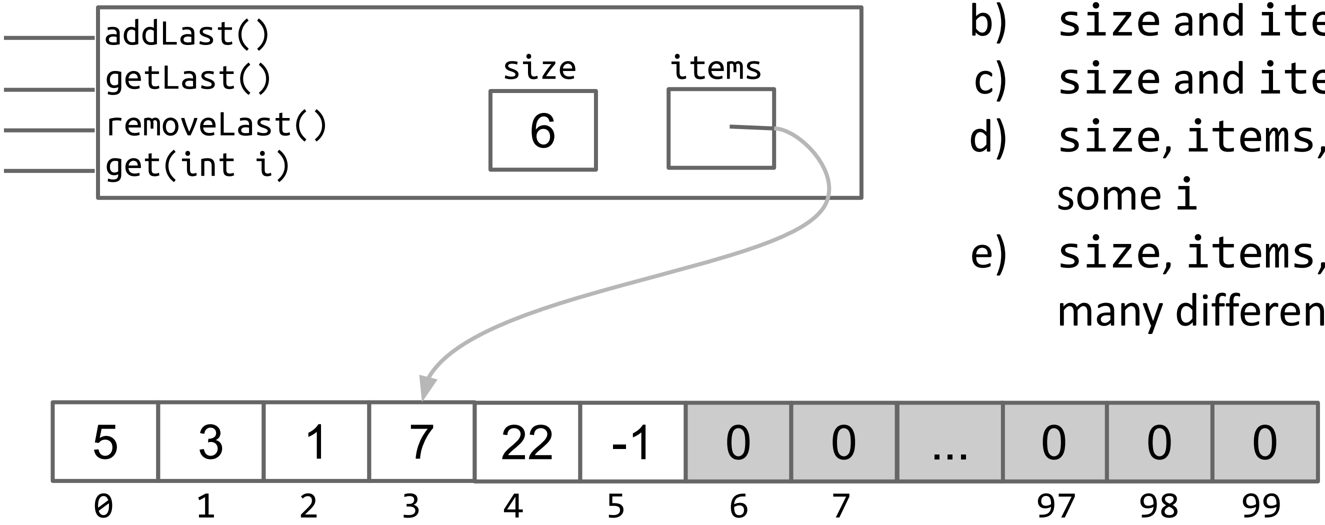
AList invariants.

removeLast Implementation

Lecture 7, CS61B, Spring 2024

Deletion Debrief

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.

Coding Demo: Basic ArrayList removeLast

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size
    getLast: The item we want to return is in position size - 1.
    size: The number of items in the list should be size.
*/
public class AList {
   private int[] items;
   private int size;

   /** Deletes item from back of list and returns deleted item. */
   public int removeLast() {



   }

}
```

Coding Demo: Basic ArrayList removeLast

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size
    getLast: The item we want to return is in position size - 1.
    size: The number of items in the list should be size.
*/
public class AList {
   private int[] items;
   private int size;

   /** Deletes item from back of list and returns deleted item. */
   public int removeLast() {
       int x = getLast();


   }

}
```

Coding Demo: Basic ArrayList removeLast

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size
    getLast: The item we want to return is in position size - 1.
    size: The number of items in the list should be size.
*/
public class AList {
   private int[] items;
   private int size;

   /** Deletes item from back of list and returns deleted item. */
   public int removeLast() {
       int x = getLast();

       return x;
   }

}
```

Coding Demo: Basic ArrayList removeLast

AList.java

```
/** Invariants:
    addLast: The next item we want to add, will go into position size
    getLast: The item we want to return is in position size - 1.
    size: The number of items in the list should be size.
*/
public class AList {
   private int[] items;
   private int size;

   /** Deletes item from back of list and returns deleted item. */
   public int removeLast() {
       int x = getLast();
       size = size - 1;
       return x;
   }

}
```

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`.

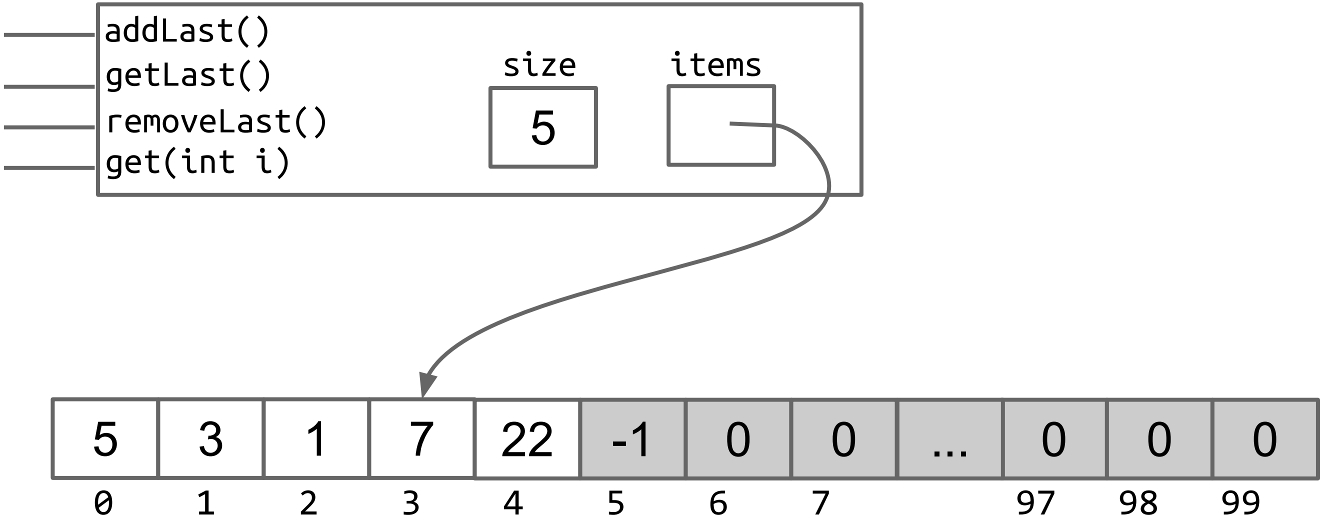
```
public int removeLast() {
   int x = items[size - 1];
   items[size - 1] = 0;
   size -= 1;
   return x;
}
```

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

Note

What about get?

- Some students suggest we should set the value to zero so that we can't get(5).
- There's no specified behavior for what to do when get is out of bounds.
  - IMO an exception is best.

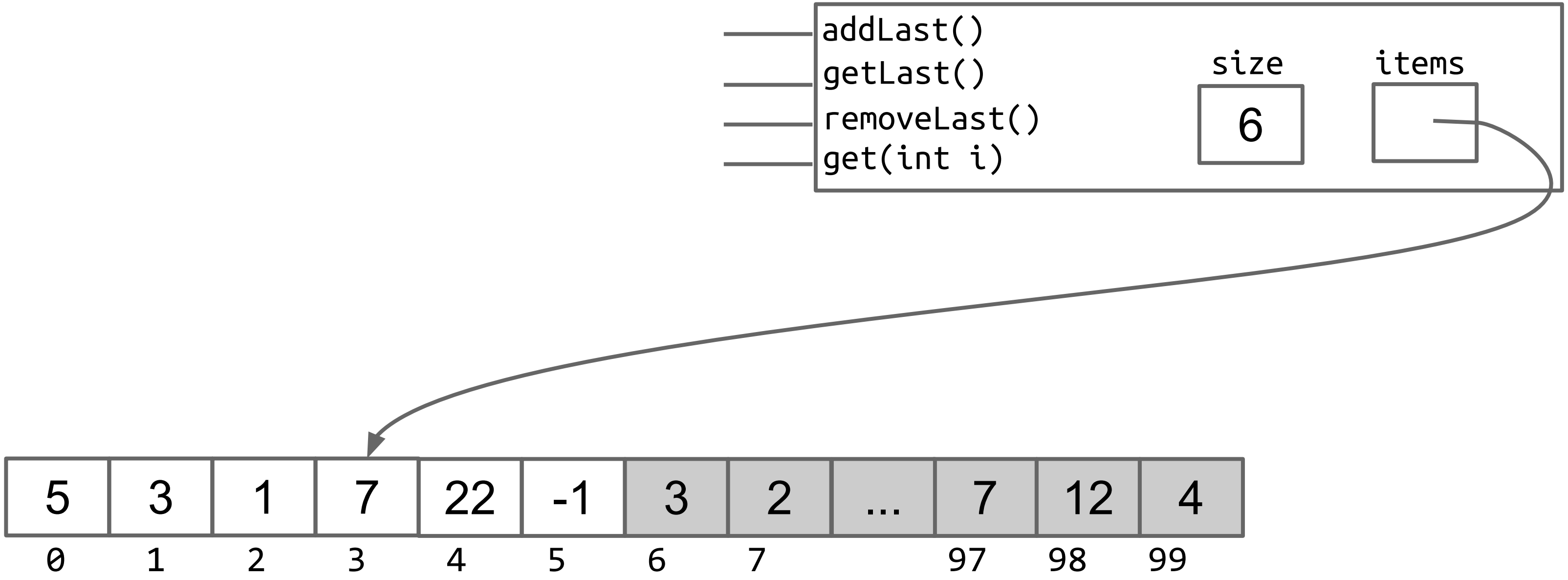


Resizing Array Theory

Lecture 7, CS61B, Spring 2024

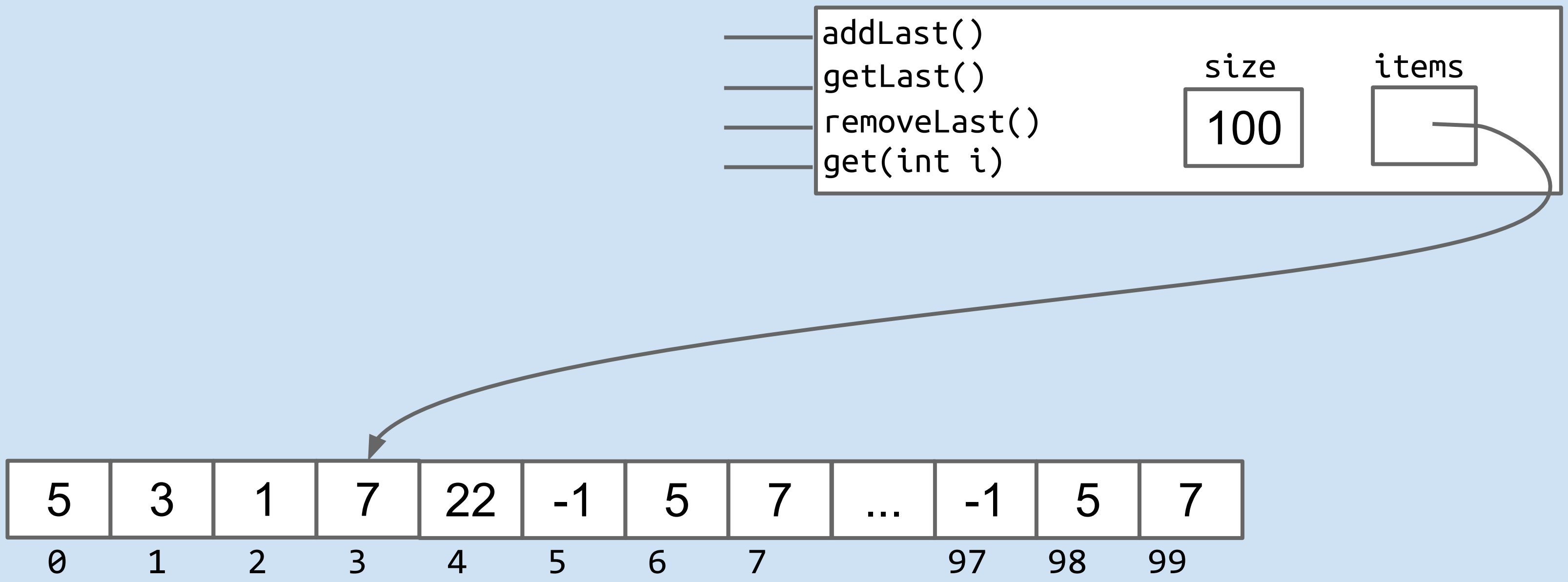
The Mighty AList

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.

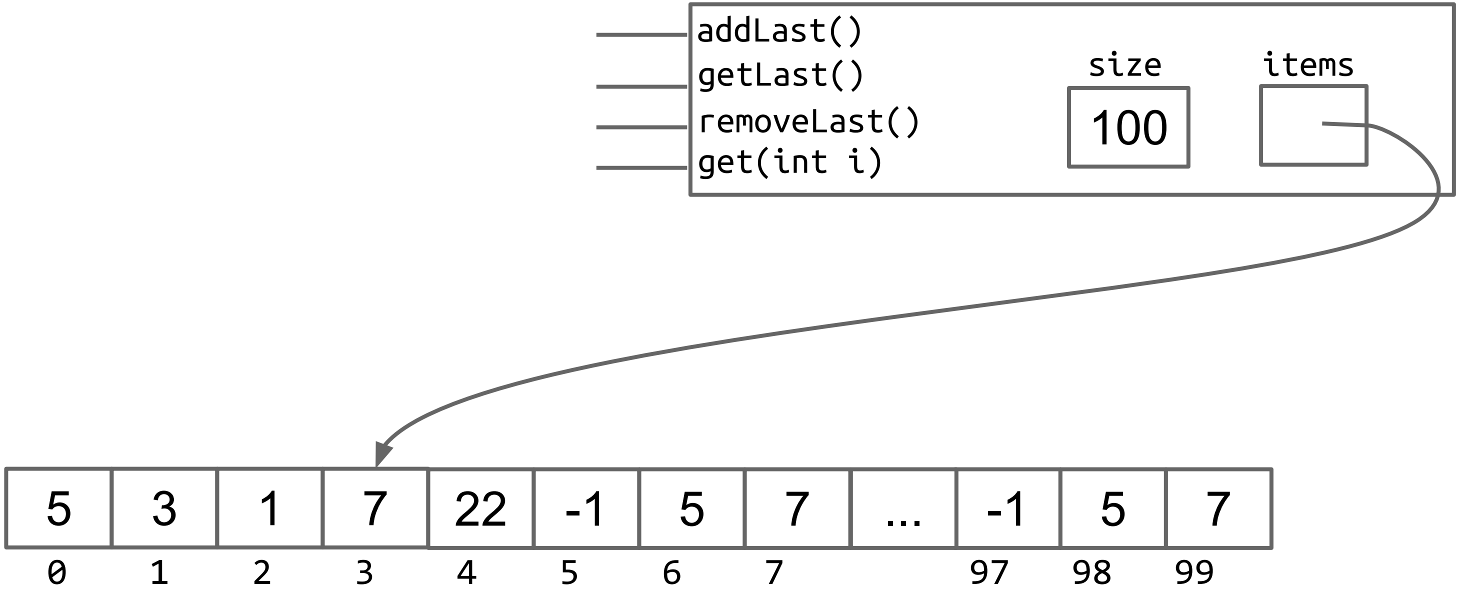


What happens if we insert into the AList above? What should we do about it?

Array Resizing

size==items.length

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

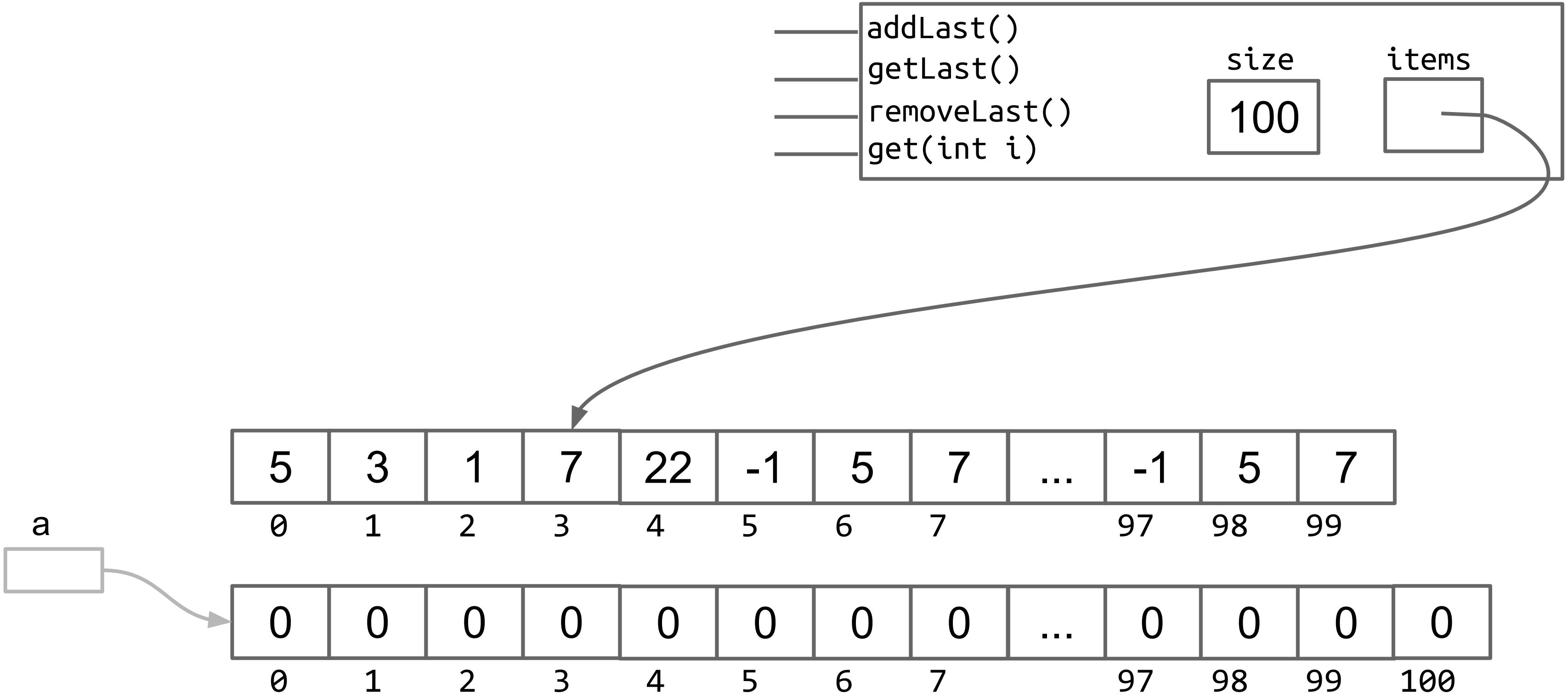


Array Resizing

size==items.length

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

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

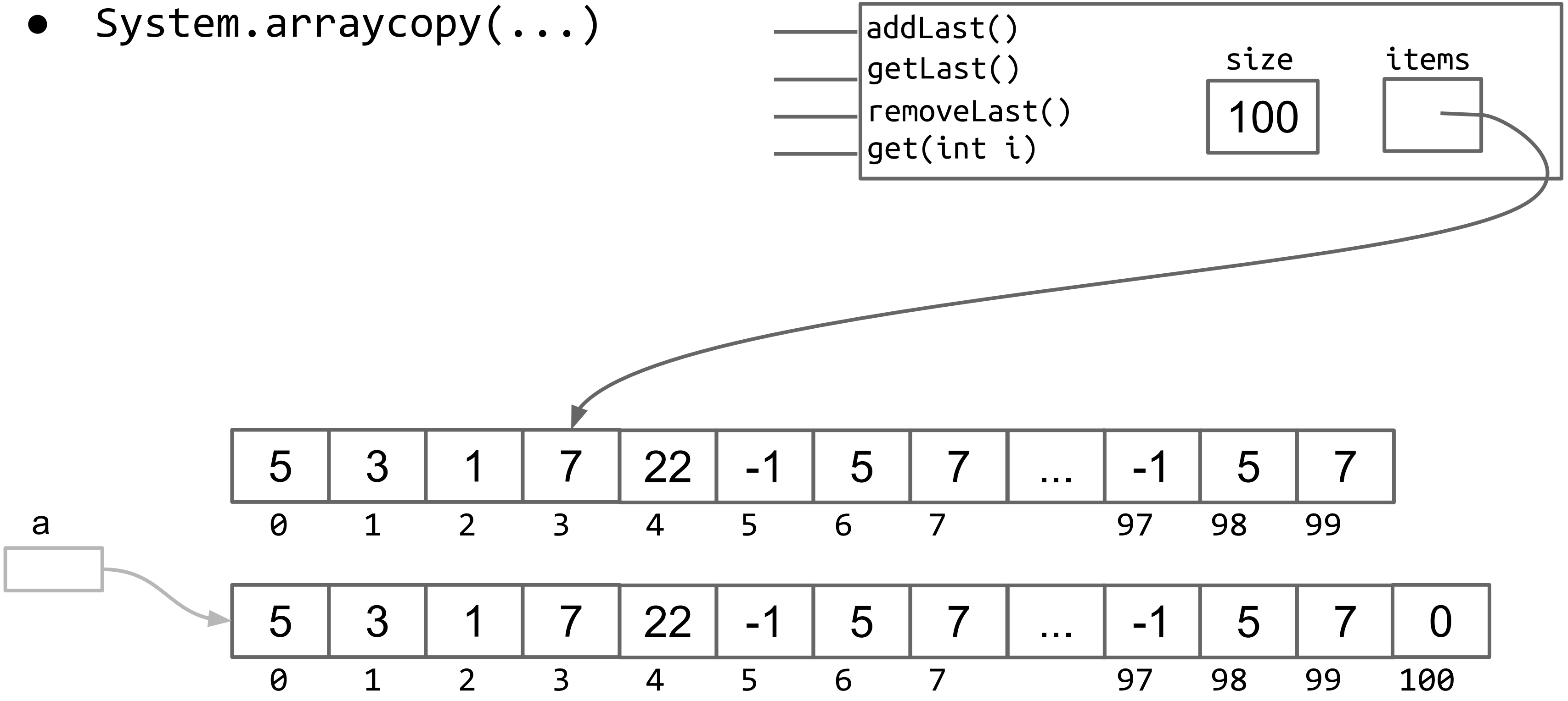


Array Resizing

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(...)`

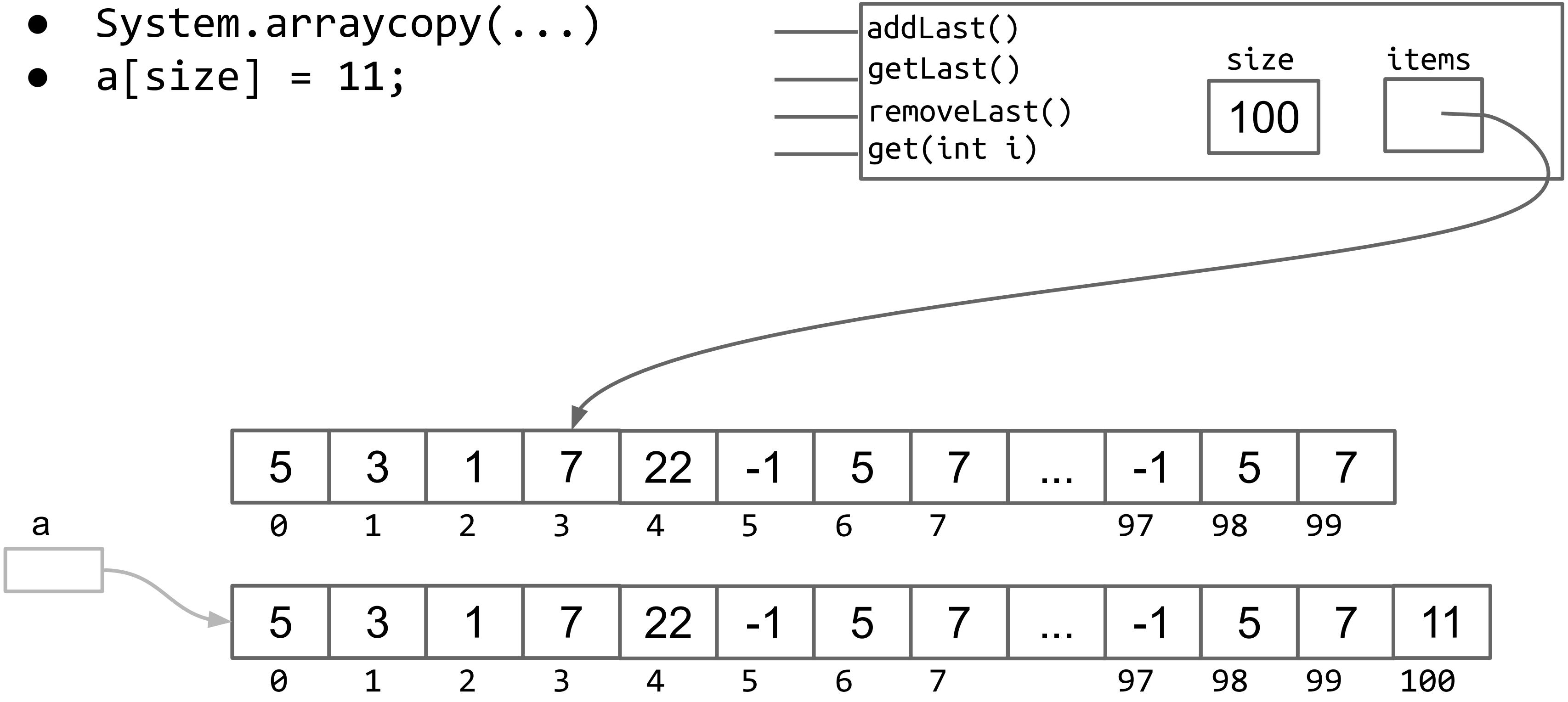


Array Resizing

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;`

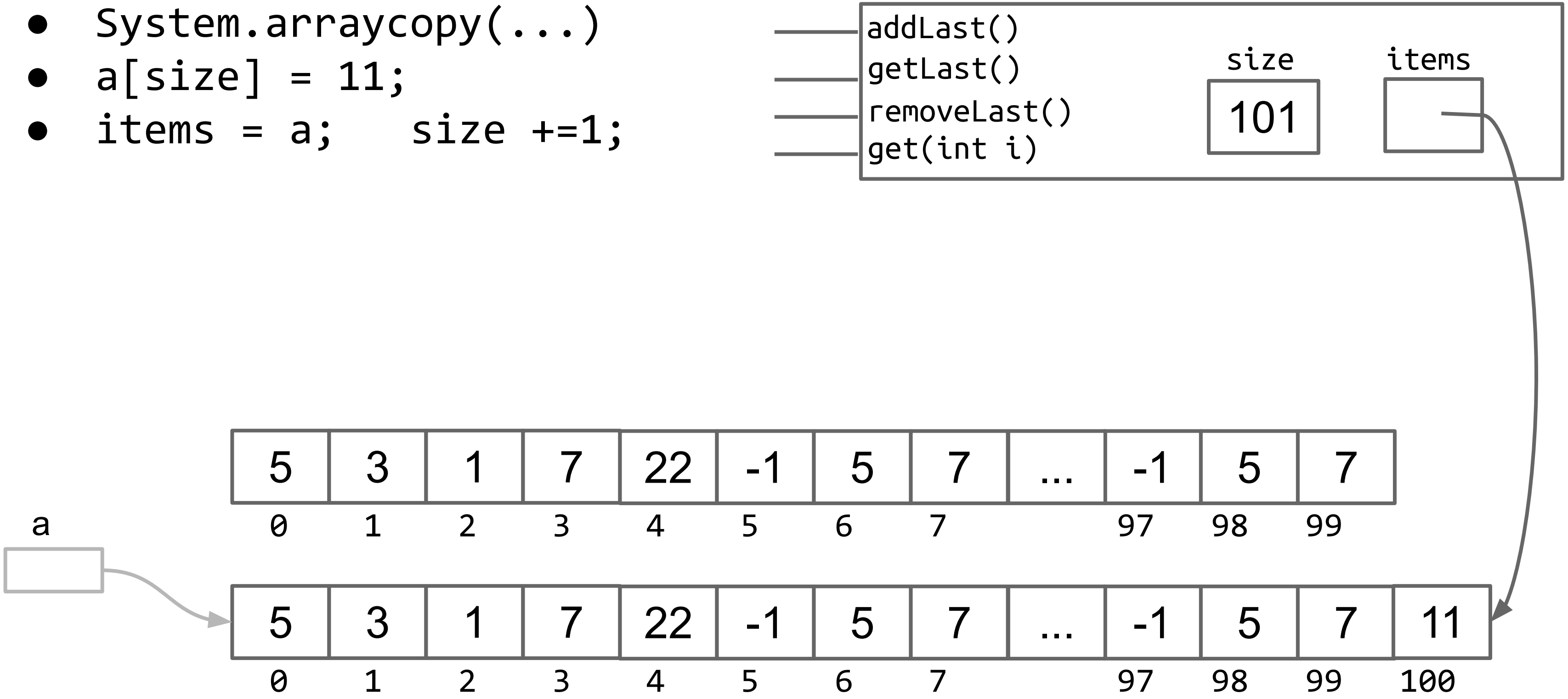


Array Resizing

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;`



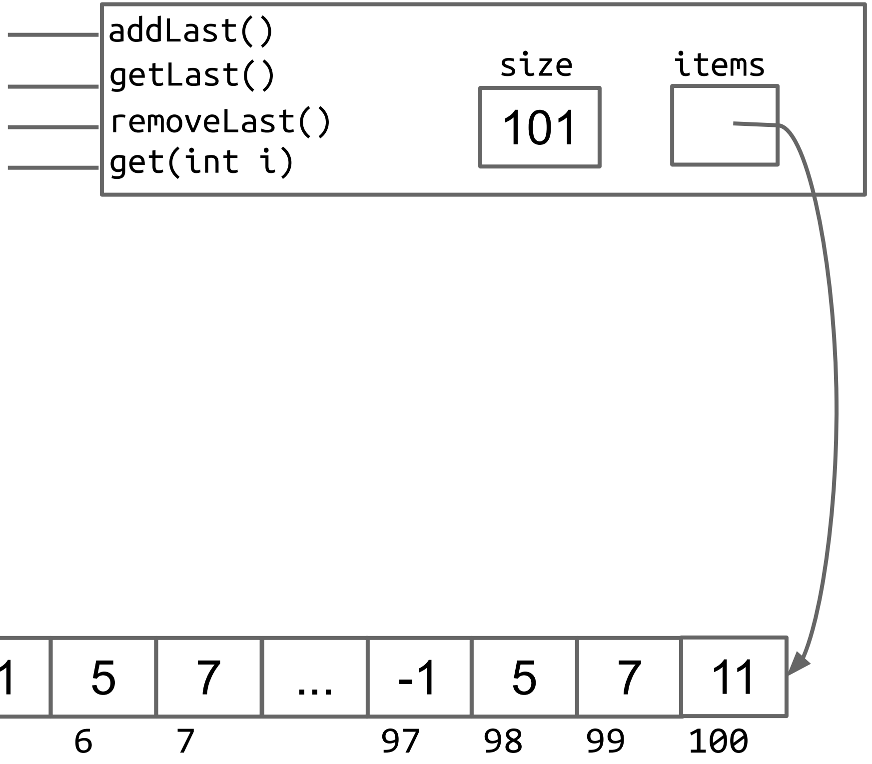
Array Resizing

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”



Resizing Array Implementation

Lecture 7, CS61B, Spring 2024

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.

Coding Demo: Resizing Array

AList.java

```
public class AList {







    /** Inserts x into the back of the list. */
    public void addLast(int x) {





        items[size] = x;
        size += 1;
    }
}
```

Coding Demo: Resizing Array

AList.java

```
public class AList {







    /** Inserts x into the back of the list. */
    public void addLast(int x) {
        if (size == items.length) {



        }
        items[size] = x;
        size += 1;
    }
}
```

Coding Demo: Resizing Array

AList.java

```
public class AList {








    /** Inserts x into the back of the list. */
    public void addLast(int x) {
        if (size == items.length) {
            int[] a = new int[size + 1];


        }
        items[size] = x;
        size += 1;
    }
}
```

Coding Demo: Resizing Array

AList.java

```
public class AList {









    /** Inserts x into the back of the list. */
    public void addLast(int x) {
        if (size == items.length) {
            int[] a = new int[size + 1];
            System.arraycopy(items, 0, a, 0, size);

        }
        items[size] = x;
        size += 1;
    }
}
```

Coding Demo: Resizing Array

AList.java

```
public class AList {







    /** Inserts x into the back of the list. */
    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;
    }
}
```

Coding Demo: Resizing Array

AList.java

```
public class AList {








    /** Inserts x into the back of the list. */
    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;
    }
}
```

The resizing functionality is really its own independent operation, separate from addLast.

We could organize our code better by moving this code into its own function.

Coding Demo: Resizing Array

AList.java

```java
public class AList {
    /** Resizes the underlying array to the target capacity. */
    private void resize(int capacity) {




    }

    /** Inserts x into the back of the list. */
    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;
    }
}
```

Coding Demo: Resizing Array

AList.java

```
public class AList {
    /** Resizes the underlying array to the target capacity. */
    private void resize(int capacity) {
        int[] a = new int[size + 1];
        System.arraycopy(items, 0, a, 0, size);
        items = a;
    }

    /** Inserts x into the back of the list. */
    public void addLast(int x) {
        if (size == items.length) {



        }
        items[size] = x;
        size += 1;
    }
}
```

Coding Demo: Resizing Array

AList.java

```java
public class AList {
    /** Resizes the underlying array to the target capacity. */
    private void resize(int capacity) {
        int[] a = new int[capacity];
        System.arraycopy(items, 0, a, 0, size);
        items = a;
    }

    /** Inserts x into the back of the list. */
    public void addLast(int x) {
        if (size == items.length) {



        }
        items[size] = x;
        size += 1;
    }
}
```

Coding Demo: Resizing Array

AList.java

```
public class AList {
    /** Resizes the underlying array to the target capacity. */
    private void resize(int capacity) {
        int[] a = new int[capacity];
        System.arraycopy(items, 0, a, 0, size);
        items = a;
    }

    /** Inserts x into the back of the list. */
    public void addLast(int x) {
        if (size == items.length) {
            resize(size + 1);


        }
        items[size] = x;
        size += 1;
    }
}
```

Resizing Array Code

```
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

```
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

Easier to read and understand.
Easier to test the correctness of each function.

Runtime Analysis (Warmup)

Lecture 7, CS61B, Spring 2024

Runtime and Space Usage Analysis: yellkey.com/think

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

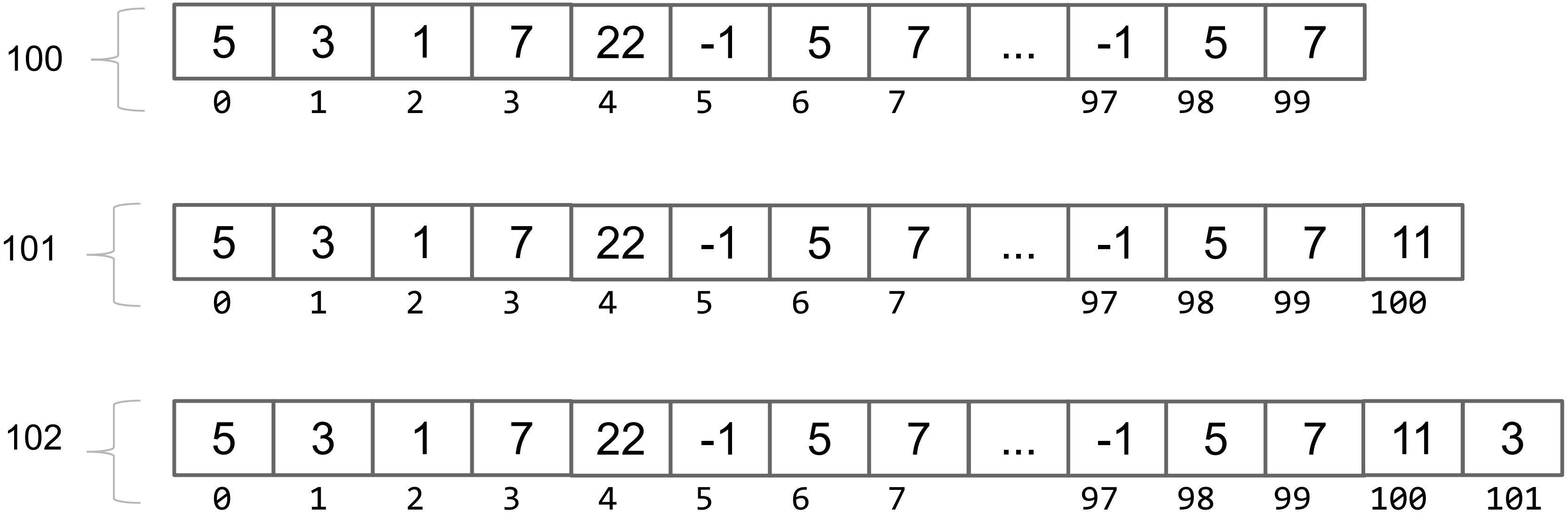
```java
private void resize(int capacity) {
  int[] a = new int[capacity];
  System.arraycopy(items, 0, a, 0, size);
  items = a;
}

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

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.

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 Analysis

Lecture 7, CS61B, Spring 2024

Demo: Speed Testing the ArrayList

```
jug ~/.../lists4/speedtest
$ time java SpeedTestSLList

real    0m0.058s
user    0m0.060s
sys     0m0.004s
```

SpeedTestSLList.java

```java
public class SpeedTestSLList {
   public static void main(String[] args) {
      SLList<Integer> L = new SLList<>();
      int i = 0;
      while (i < 100000) {
         L.addFirst(i);
         i = i + 1;
      }
   }
}
```

Adding 100,000 items to a new SLList is very fast (~0.05 seconds).

Demo: Speed Testing the ArrayList

```
jug ~/.../lists4/speedtest
$ time java SpeedTestAList

real    0m2.945s
user    0m2.872s
sys     0m0.068s
```

SpeedTestAList.java

```java
public class SpeedTestAList {
   public static void main(String[] args) {
      AList L = new AList();
      int i = 0;
      while (i < 100000) {
         L.addLast(i);
         i = i + 1;
      }
   }
}
```

Adding 100,000 items to a new AList is very slow (~3 seconds).

Runtime and Space Usage Analysis: yellkey.com/TODO

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

Runtime and Space Usage Analysis

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

Rough intuition: Form pairs that all sum to N+1:

N + 1

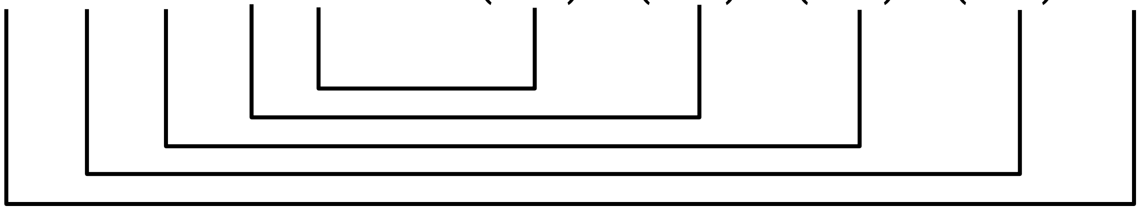
(N-1) + 2

(N-2) + 3

...

There are N/2 such pairs.

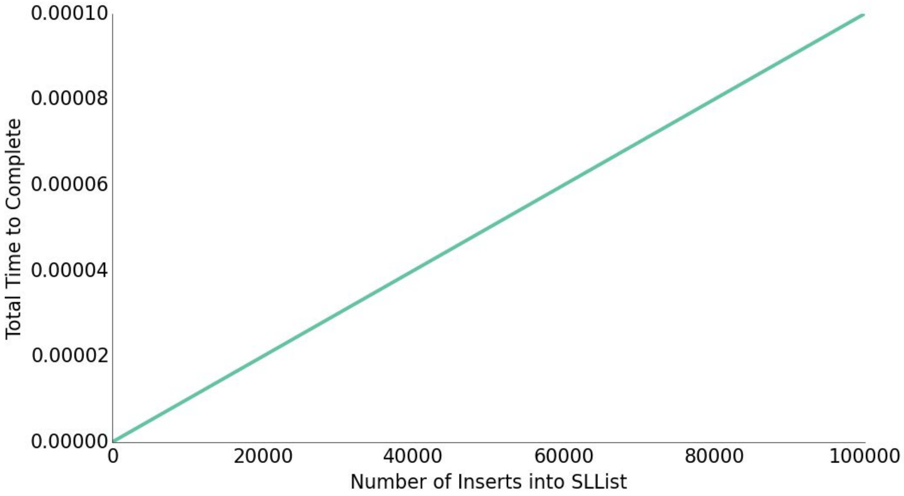
1 + 2 + 3 + 4 + 5 + ... + (N-4) + (N-3) + (N-2) + (N-1) + N



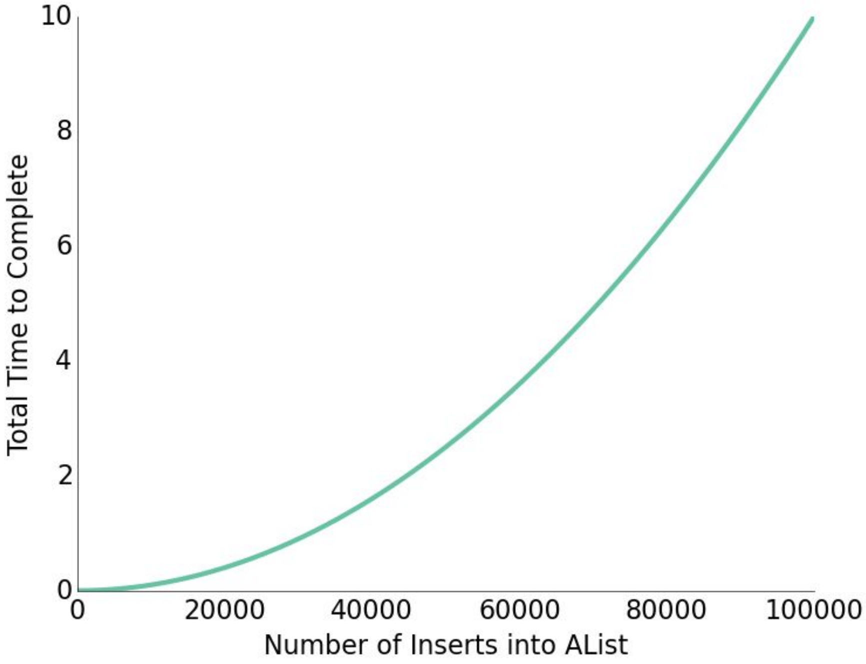
Resizing Slowness

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.



Note: Insert here is addFirst



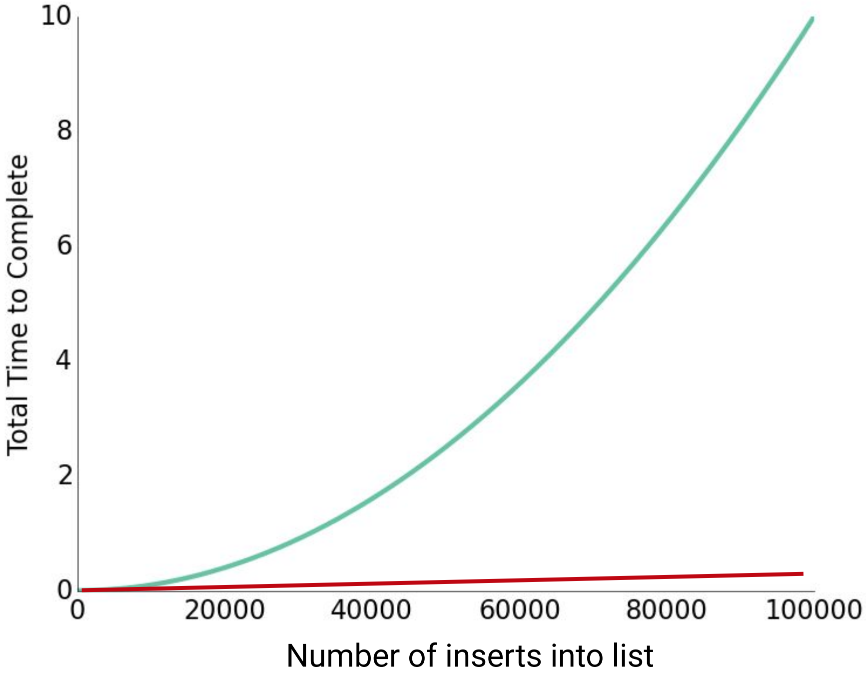
Resizing Slowness

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.

The same graphs from the previous slide, placed on top of each other.

Red line = SLList
Teal line = AList



Better Resizing Strategy

Lecture 7, CS61B, Spring 2024

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;
}
```

Demo: Speed Testing the ArrayList

```
public void addLast(int x) {
   if (size == items.length) {
      resize(size + 10);
   }
   items[size] = x;
   size += 1;
}
```

```
jug ~/.../lists4/speedtest
$ time java SpeedTestAList

real    0m0.373s
user    0m0.328s
sys     0m0.044s
```

Resizing by 10 elements instead of 1 seems to speed up adding 100,000 items…

```
jug ~/.../lists4/speedtest
$ time java SpeedTestAList

real    0m16.572s
user    0m15.968s
sys     0m0.284s
```

…but the problem re-emerges if we try to add 1,000,000 items.

Demo: Speed Testing the ArrayList

```
public void addLast(int x) {
   if (size == items.length) {
       resize(size * 2);
   }
   items[size] = x;
   size += 1;
}
```

```
jug ~/.../lists4/speedtest
$ time java SpeedTestAList

real    0m0.069s
user    0m0.068s
sys     0m0.008s
```

If we double the array capacity every time it's full, then adding 100,000 items is fast…

```
jug ~/.../lists4/speedtest
$ time java SpeedTestAList

real    0m0.112s
user    0m0.088s
sys     0m0.028s
```

…and adding 1,000,000 items is also fast.

(Probably) Surprising Fact

Geometric resizing is much faster.

We can't prove this until later. [See this video for a more detailed analysis.]

Rough intuition: As the array grows larger, we resize less often.

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

Unusably bad.

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

Great performance.

This is how the Python list is implemented.

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.

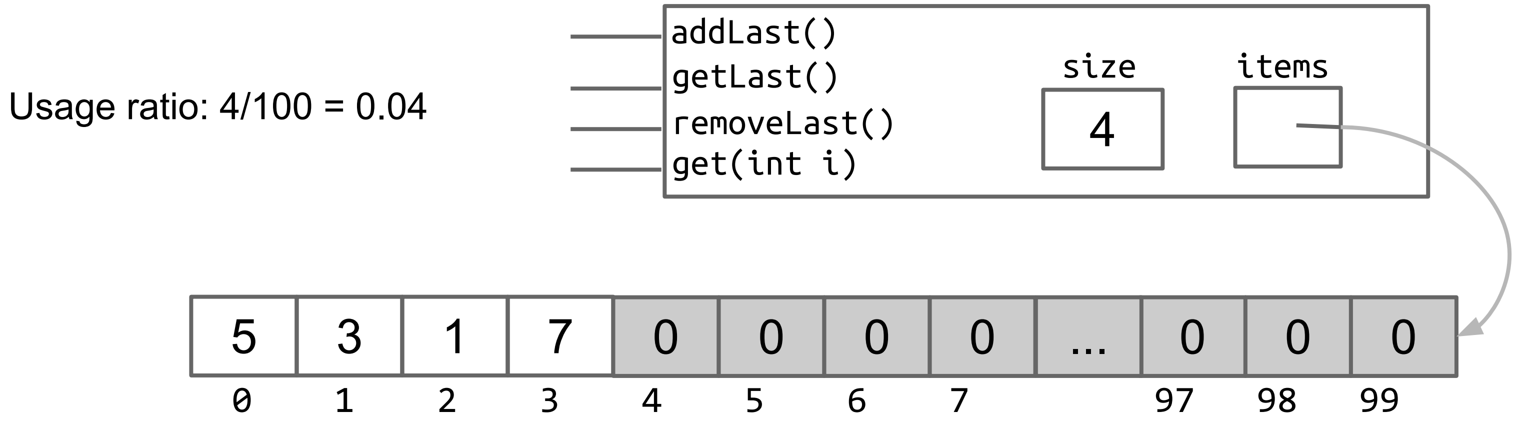
Our data structure will execute these operations acceptably fast, but afterwards there is a problem.

- 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 ArrayLists

Lecture 7, CS61B, Spring 2024

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)

```
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[8];`
- Causes a compiler warning, which you should ignore.

Why not just `new Glorp[cap];`

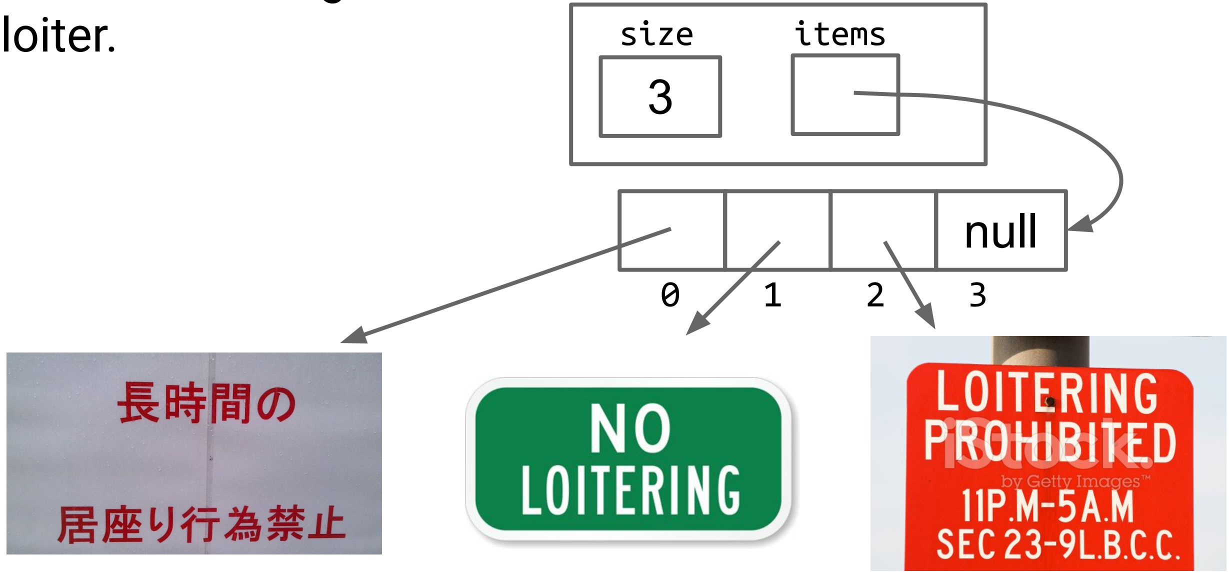
- Will cause a “generic array creation” error.

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.

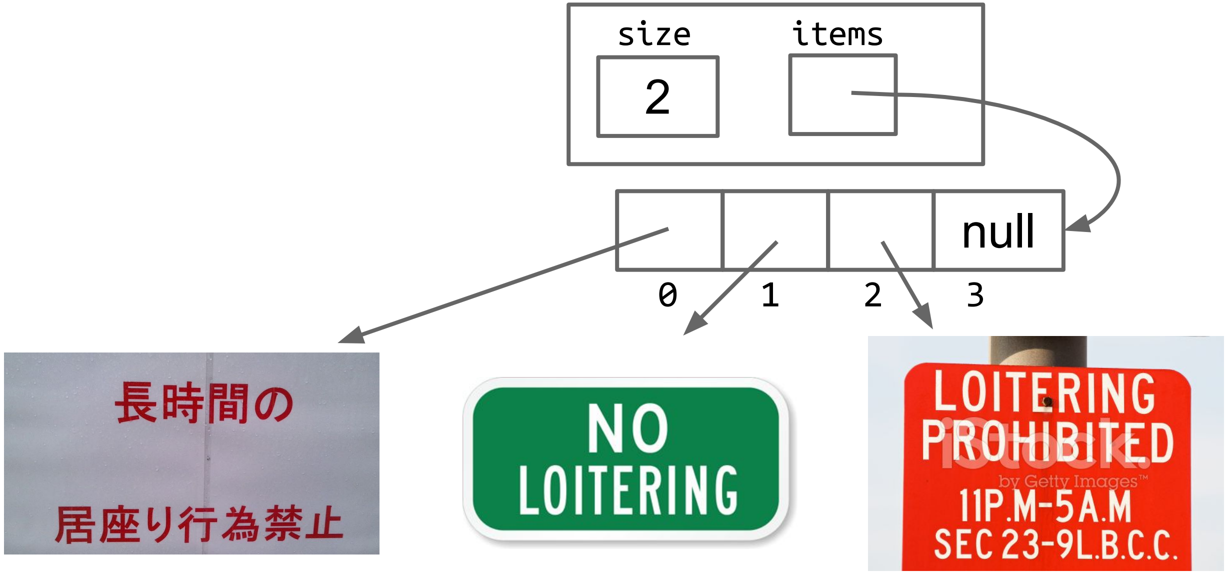
```java
public Glorp removeLast() {
    Glorp returnItem = getLast();
    items[size - 1] = null;
    size -= 1;
    return returnItem;
}
```



Loitering Example

Changing size to 2 yields a correct AList.

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

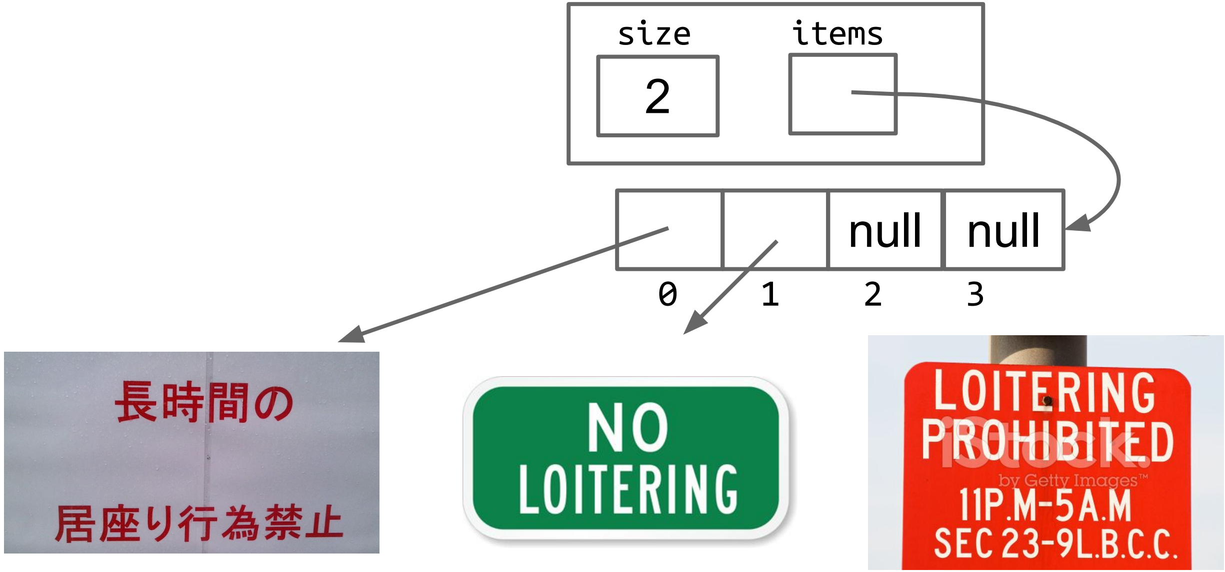


Loitering Example

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.

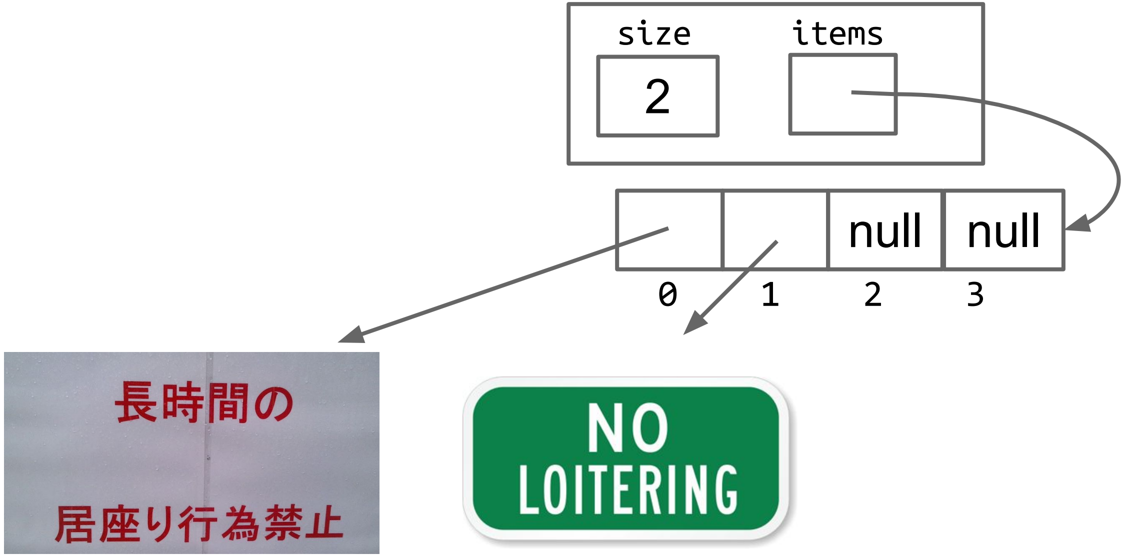


Loitering Example

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

Lecture 7, CS61B, Spring 2023

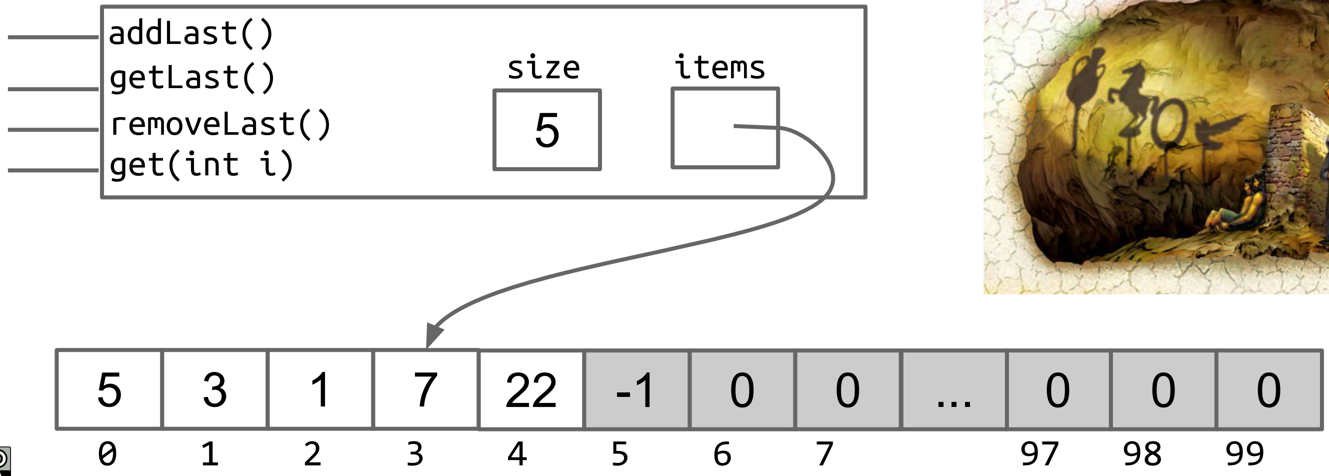
One last thought: Obscurantism in Java

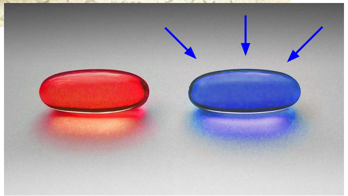
We talk of "layers of abstraction" often in computer science.

- Related concept: 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.

- Related concept: 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 within a class.
  - 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 saw in lecture 6.