

LEC 03

# CSE 123

## Implementing Data Structures; ArrayIntList

Questions during Class?

Raise hand or send here

sli.do #cse123



### BEFORE WE START

*Talk to your neighbors:  
Did you eat breakfast today? If so,  
what?*

**Instructors:** Brett Wortzman  
Miya Natsuhara

**TAs:**

Arohan	Neha	Rushil	Johnathan	Nicholas
Sean	Hayden	Srihari	Benoit	Isayah
Audrey	Chris	Andras	Jessica	Kavya
Cynthia	Shreya	Kieran	Rohan	Eeshani
Amy	Packard	Cora	Dixon	Nichole
Trien	Lawrence	Liza	Helena	

Music: [CSE 123 25wi Lecture Tunes](#)

# Announcements

- Programming Assignment 0 due Wed Jan 22 at 11:59pm!
- Monday, Jan 20 is a university holiday (MLK Jr. day) so campus will be closed
  - Instructor office hours will be cancelled
  - IPL will be closed
  - Message board will still be available, but response time may vary
- Looking ahead:
  - First resubmission opens Wed, Jan 22
  - First quiz on Tue, Jan 28

# Lecture Outline

- **Revisiting Reflections** ◀
- Interface v. Implementation
- Implementing `ArrayList`

# Revisiting Reflections

- Throughout this course, we'll ask you to form opinions on topics
  - Provide exposure to issues so you can decide for yourself
- Opinions aren't formed in a vacuum
  - Exposure to various viewpoints reinforces/challenges perspectives
  - Shouldn't be making arbitrary decisions
    - Rationalization is often important! (Not always necessary, but helps in communication)
- Integrating reflections to in-class components
  - Discuss opinions, challenge assumptions, potentially change minds
  - Please be respectful of other people's opinions
    - There are no "right" or "wrong" answers to these questions
    - Everyone has different experiences with the world that informs their decisions

# Lecture Outline

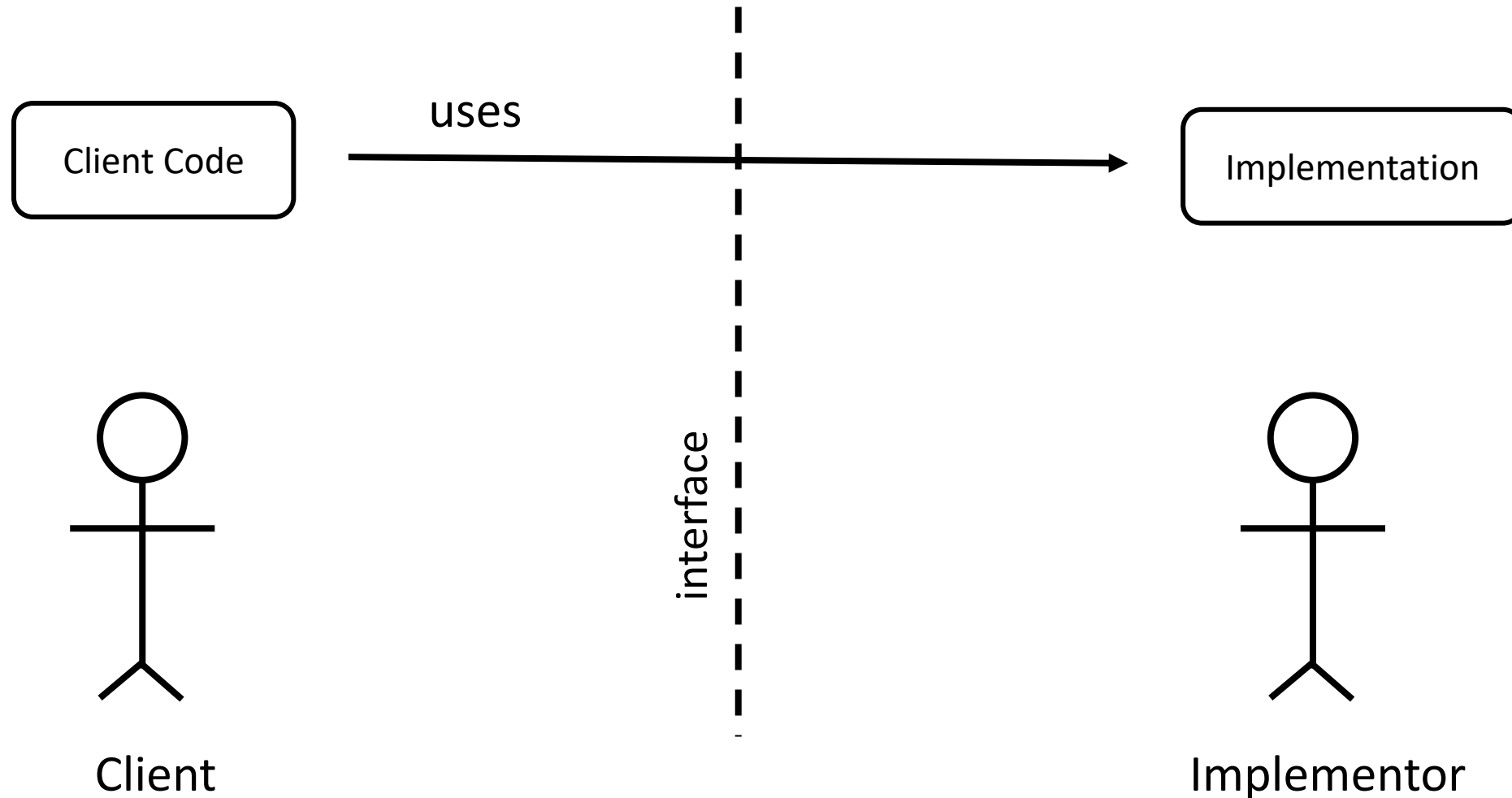
- Revisiting Reflections
- **Interface vs. Implementation** ◀
- Implementing ArrayList

# Interface versus Implementation

- Interface: what something *should* do
- Implementation: *how* something is done
- These are different!
- Big theme of CSE 123:

choose between different implementations of same interface

# Client versus Implementor



# Arrays vs. ArrayLists

Arrays	ArrayLists
<pre>int[] arr = new int[x];</pre>	<pre>List&lt;Integer&gt; al = new ArrayList&lt;&gt;();</pre>
<pre>int y = arr[0]</pre>	<pre>int y = al.get(0);</pre>
-	<pre>al.add(2);</pre>
<pre>arr[0] = 5;</pre>	<pre>al.set(0, 5);</pre>
<pre>int length = arr.length; // <i>Always x</i></pre>	<pre>int size = al.size(); // <i>Matches # of</i> // <i>things added</i></pre>
Fundamental data structure	Class within java.util
Fixed length	Illusion of resizing



# Implementing Data Structures

- No different from designing any other class!
  - Specified behavior (`List` interface):

Method	Description
<code>add(E value)</code>	Adds the given value to the end of the list
<code>add(int index, E value)</code>	Adds the given value at the given index
<code>remove(E value)</code>	Removes the given value if it exists
<code>remove(int index)</code>	Removes the value at the given index
<code>get(int index)</code>	Returns the value at the given index
<code>set(int index, int value)</code>	Updates the value at the given index to the one given
<code>size()</code>	Returns the number of elements in the list

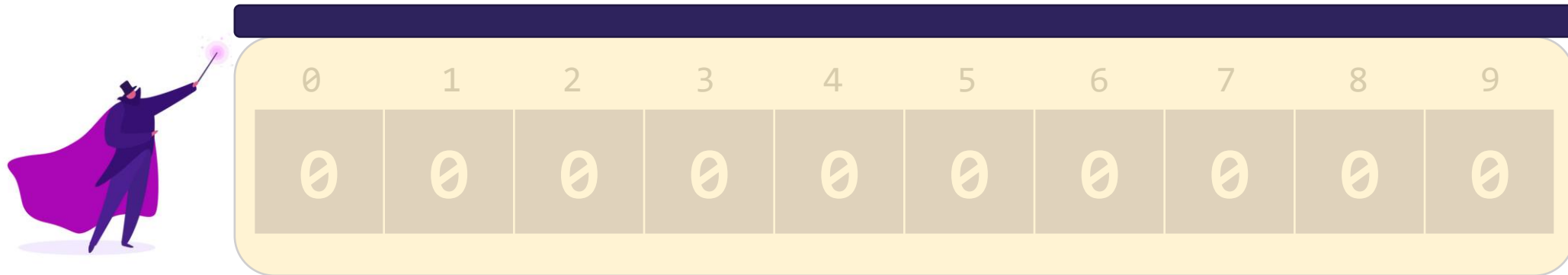
- Choose appropriate fields based on behavior
- Just requires some thinking outside the box

# Lecture Outline

- Revisiting Reflections
- Interface v. Implementation
- **Implementing ArrayList** ◀

# ArrayIntLists

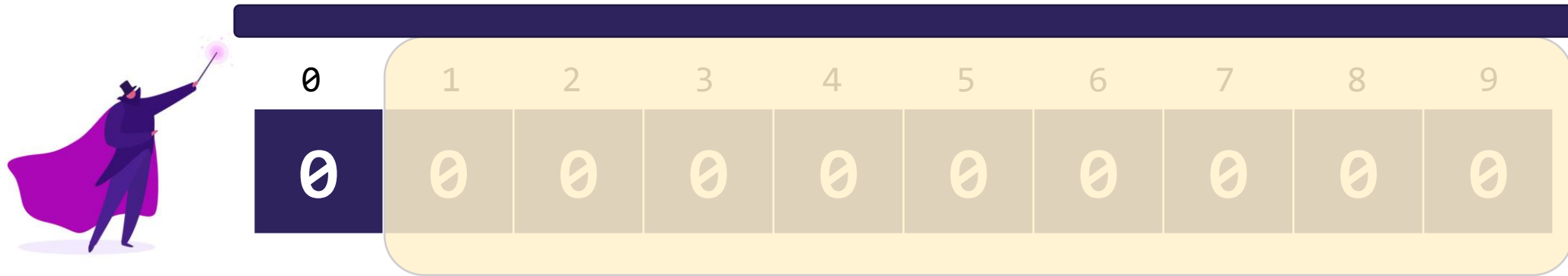
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



```
a1.add(2);
```

# ArrayIntLists

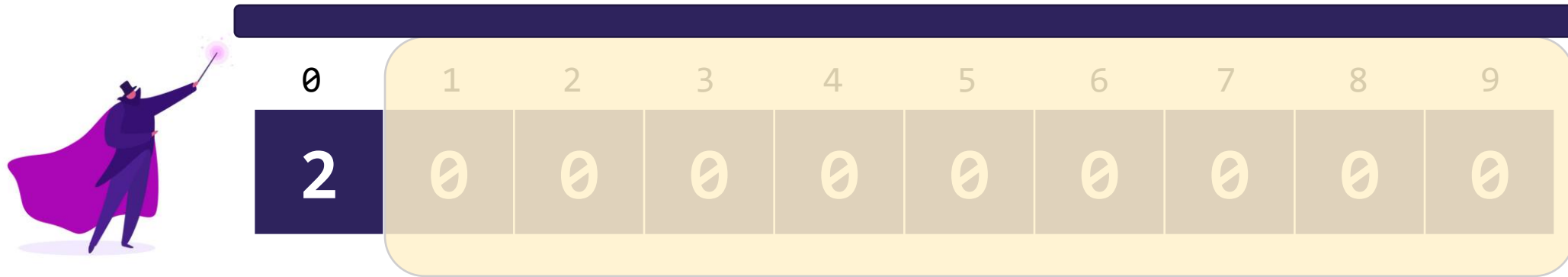
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



```
a1.add(2);
```

# ArrayIntLists

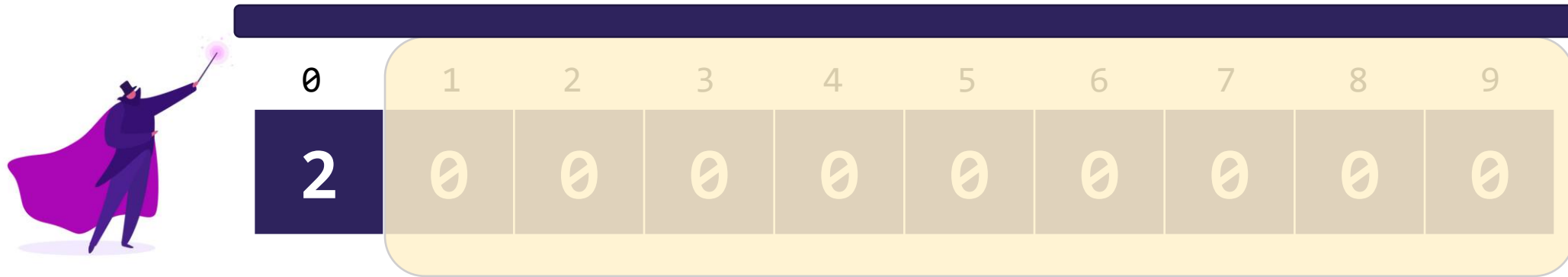
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



```
a1.add(2);
```

# ArrayIntLists

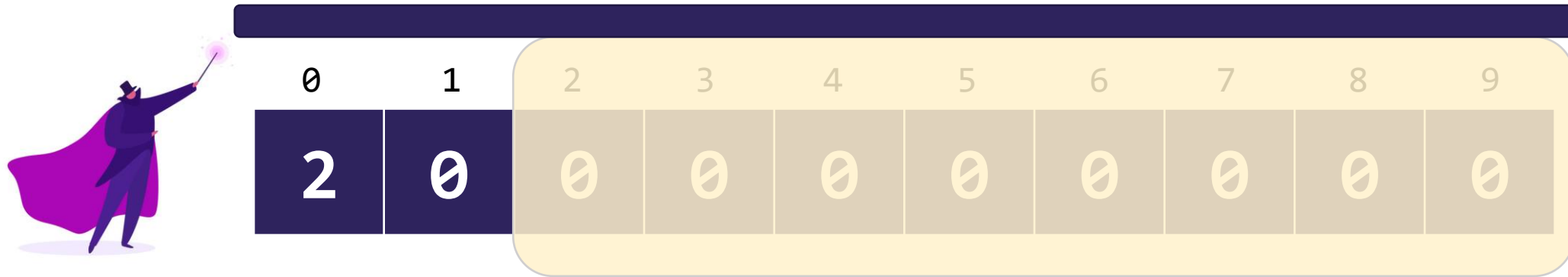
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



```
a1.add(5);
```

# ArrayIntLists

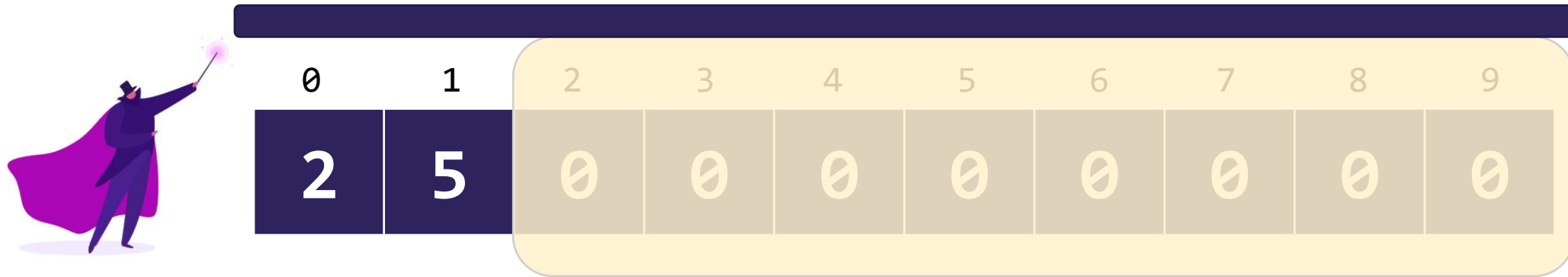
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



```
a1.add(5);
```

# ArrayIntLists

- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary

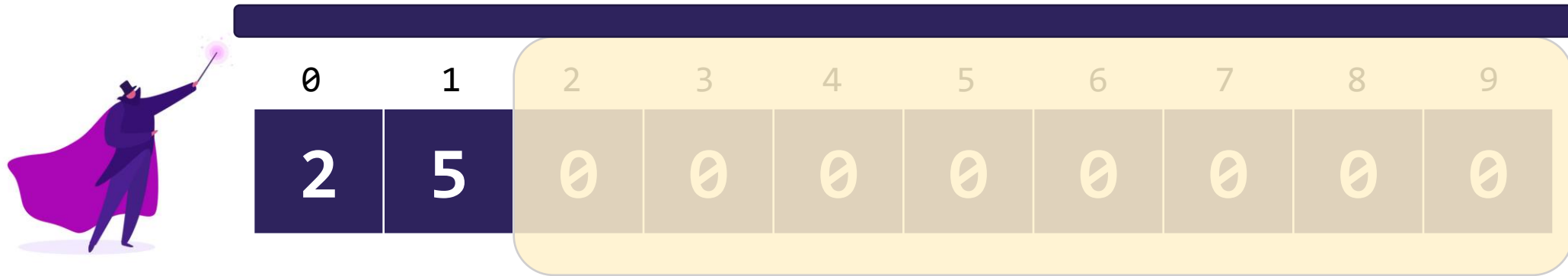


```
a1.add(5);
```



# ArrayIntLists

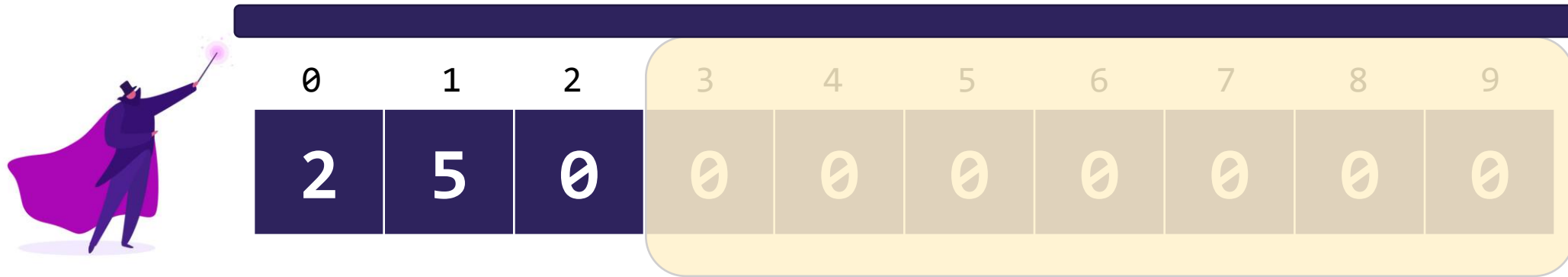
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



```
a1.add(-1);
```

# ArrayIntLists

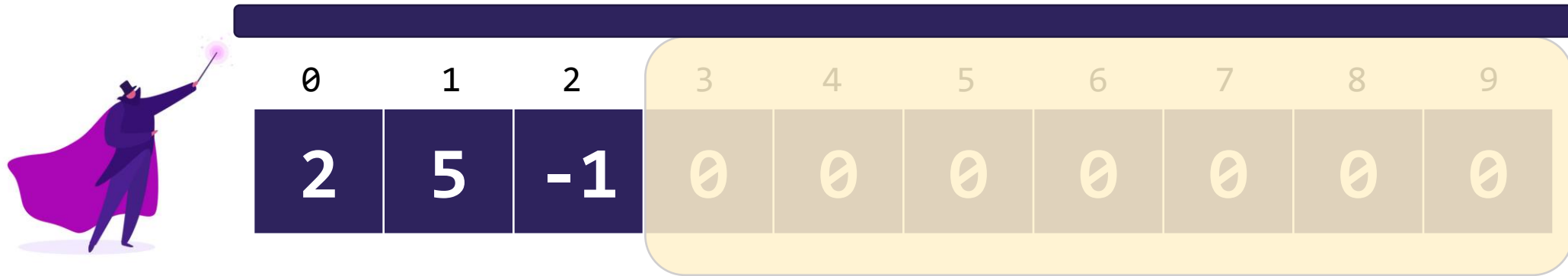
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



```
a1.add(-1);
```

# ArrayIntLists

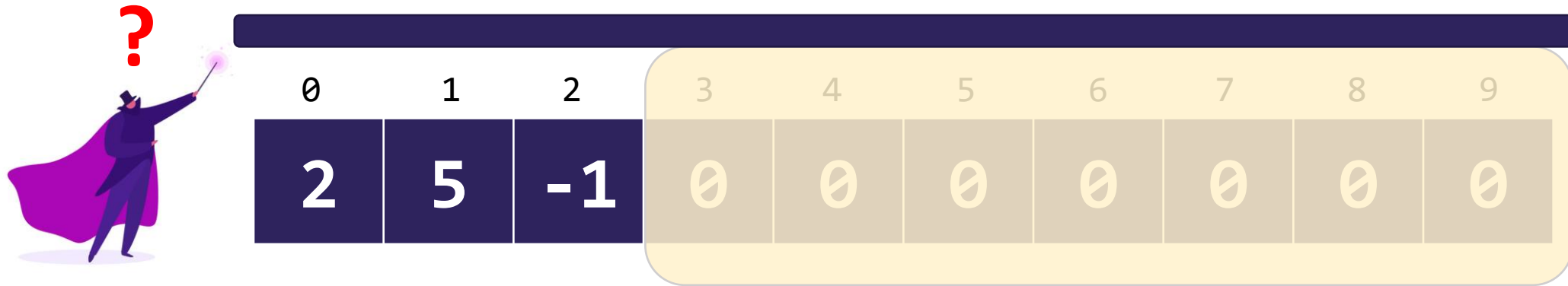
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



```
a1.add(-1);
```

# ArrayIntLists

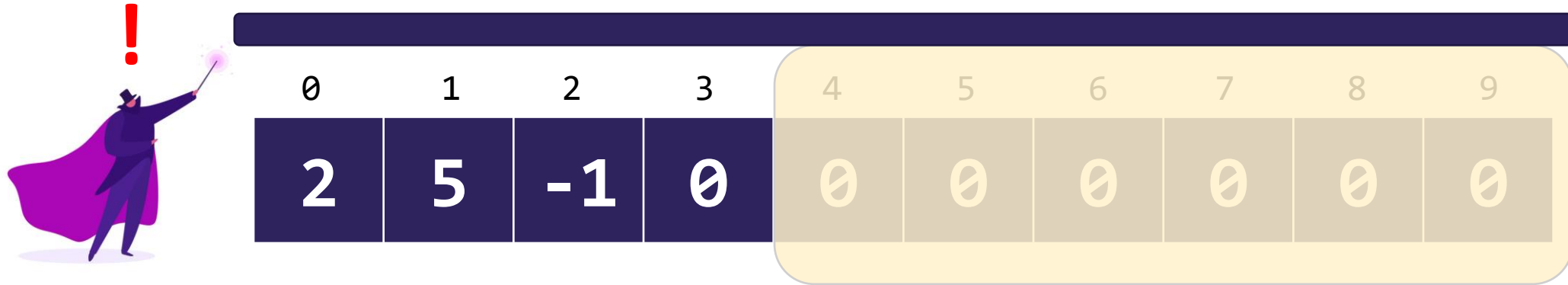
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



```
a1.add(0, 0);
```

# ArrayIntLists

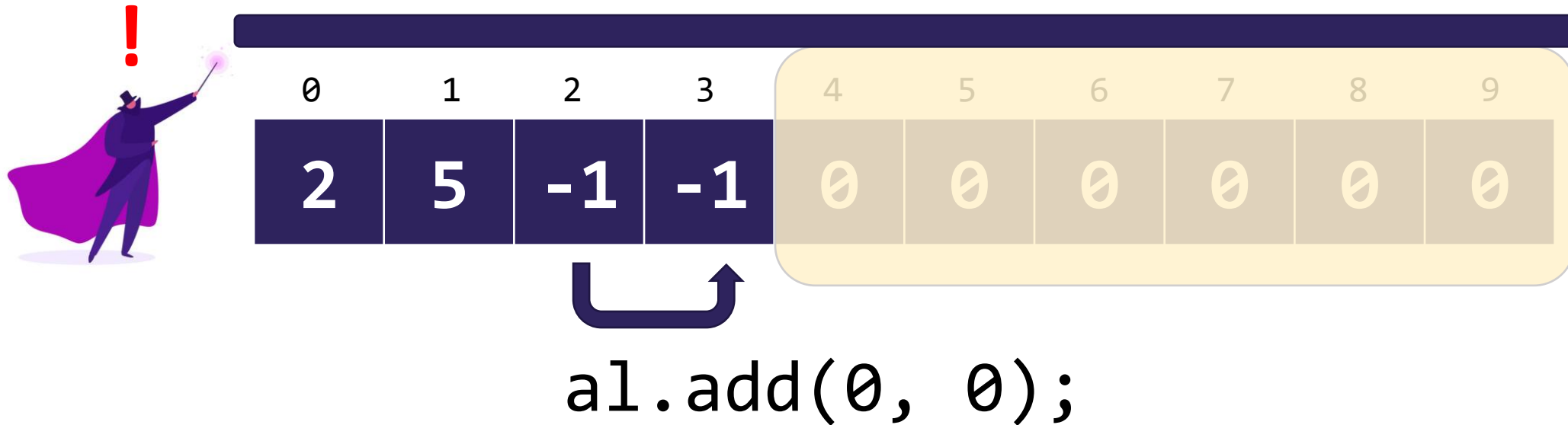
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



```
a1.add(0, 0);
```

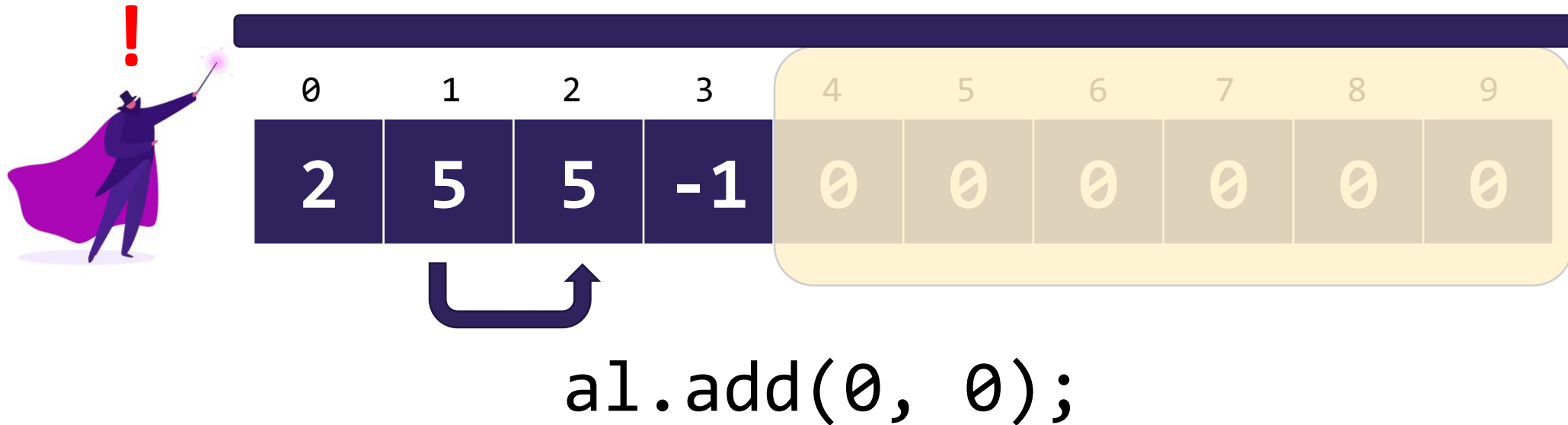
# ArrayIntLists

- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



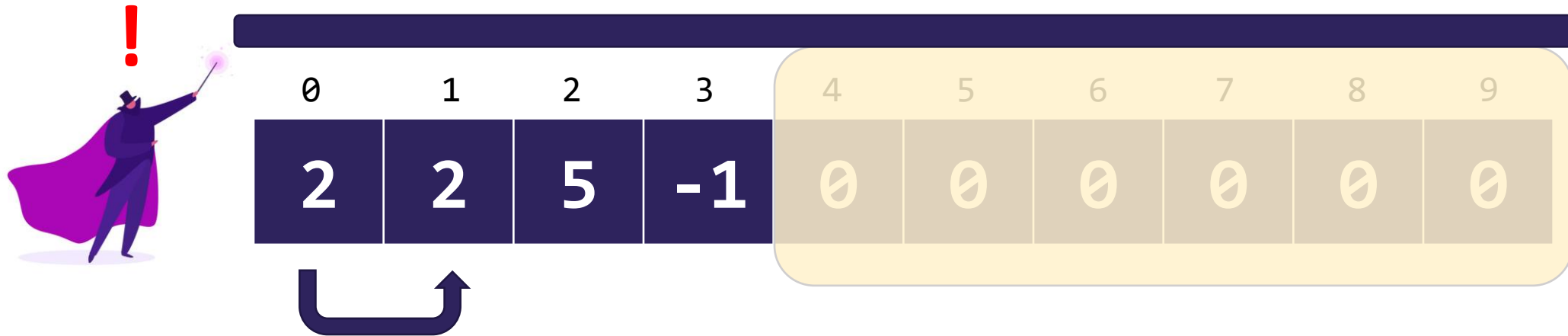
# ArrayIntLists

- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



# ArrayIntLists

- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary

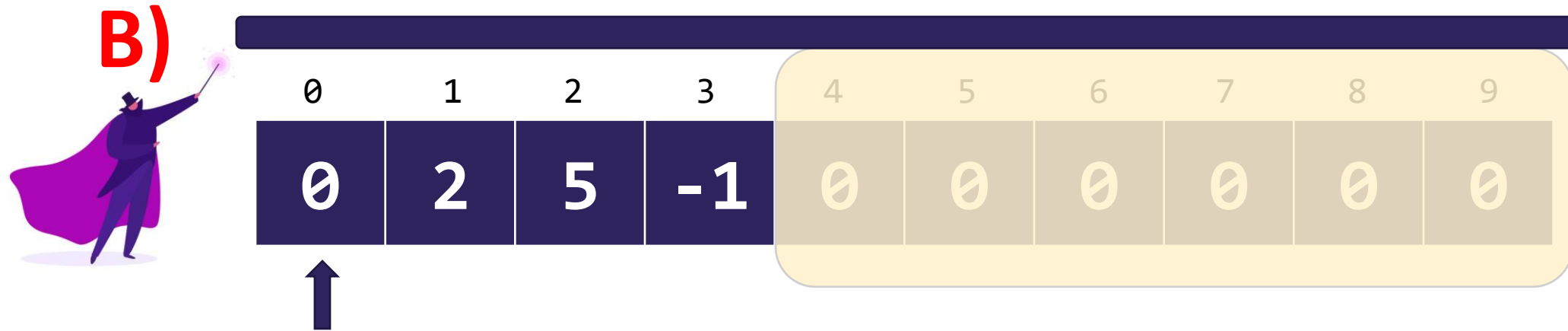


```
a1.add(0, 0);
```



# ArrayIntLists

- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary



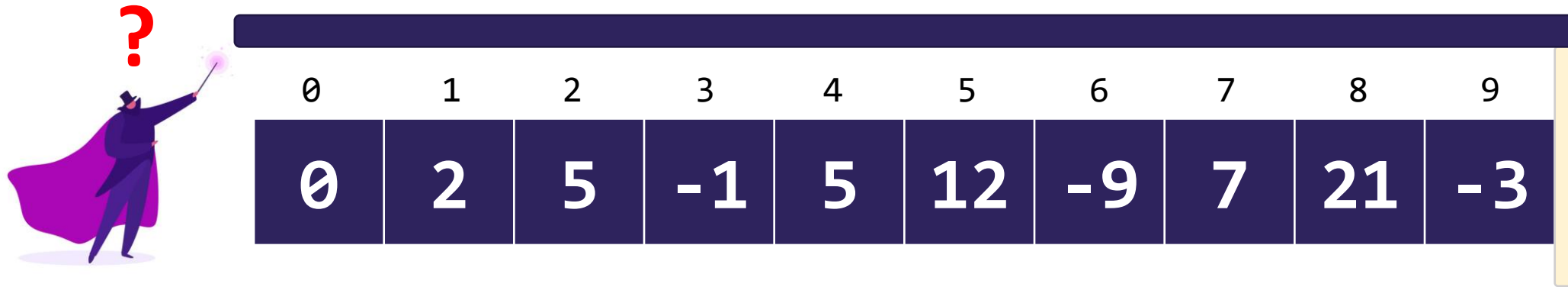
```
a1.add(0, 0);
```

# ArrayIntLists

- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;` // Where we store elements
  - `int size;` // Storage boundary
- Important points:
  - `size` represents how far the curtain is peeled back
    - Can't use a hardcoded value!
  - Starting value is always at index 0
    - Adding to / removing from beginning requires shifting elements

# Capacity and Resizing

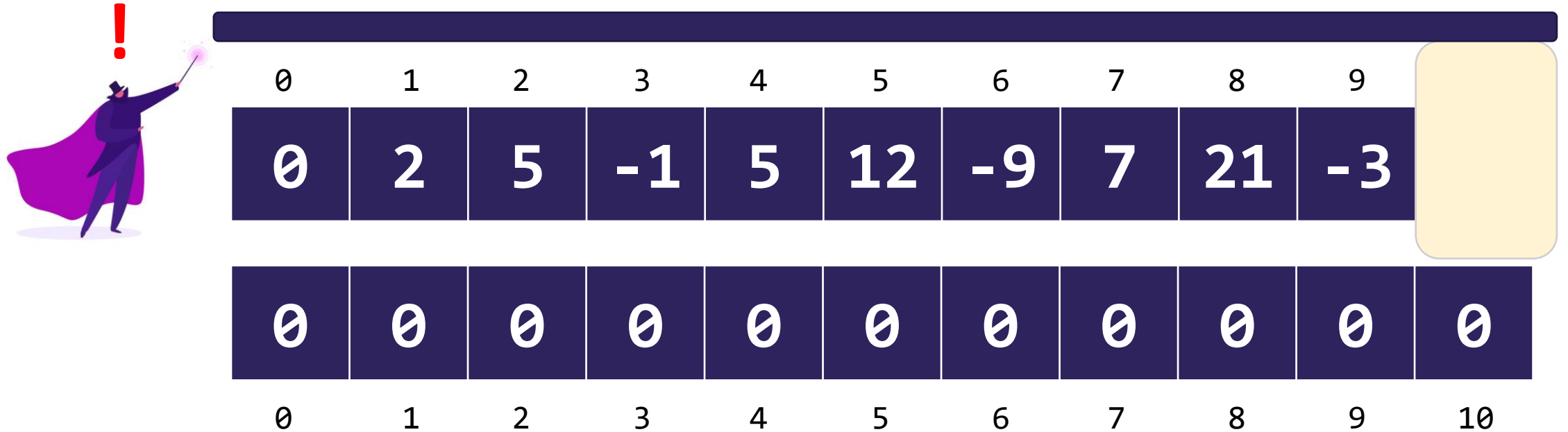
- Capacity = length of underlying array
- Size = number of user-added elements
- What happens if we run out of space? (`size == capacity`)



```
al.add(2);
```

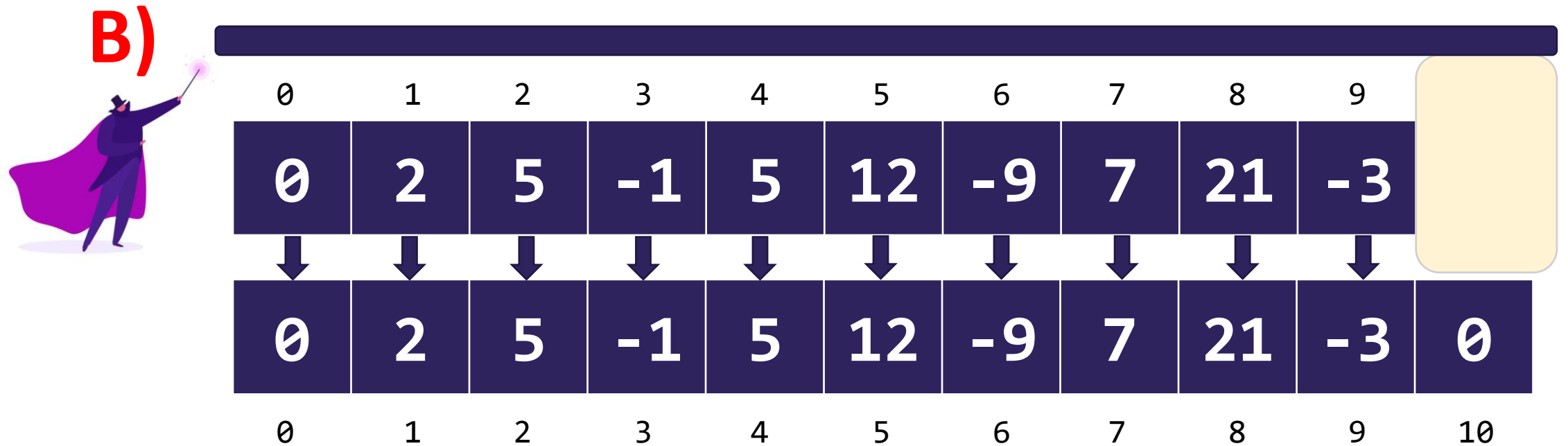
# Capacity and Resizing

- Capacity = length of underlying array
- Size = number of user-added elements
- What happens if we run out of space? (`size == capacity`)



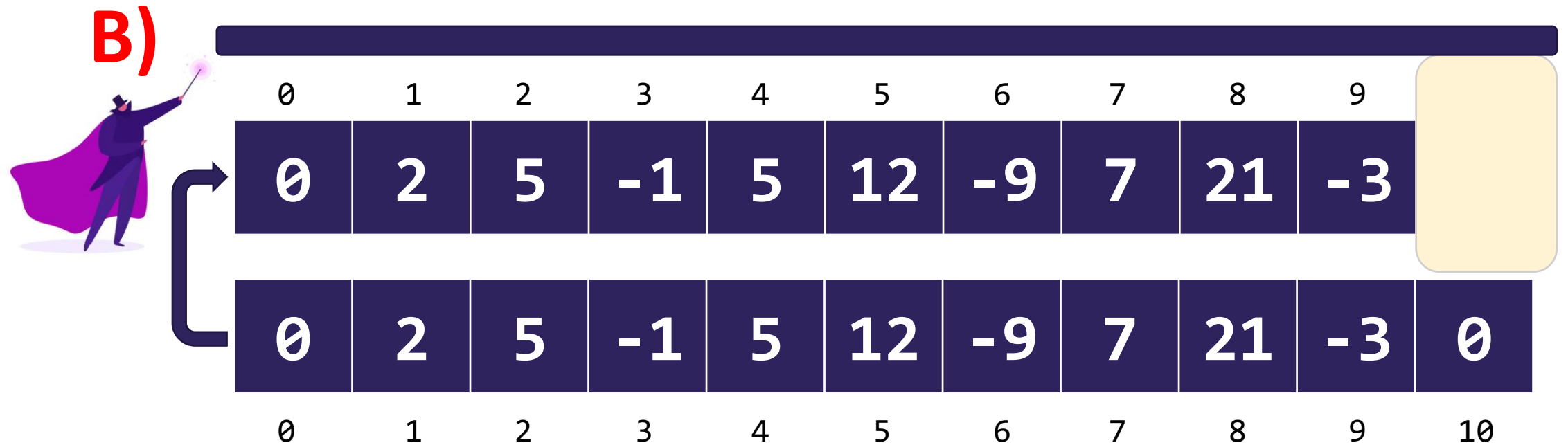
# Capacity and Resizing

- Capacity = length of underlying array
- Size = number of user-added elements
- What happens if we run out of space? ( $\text{size} == \text{capacity}$ )



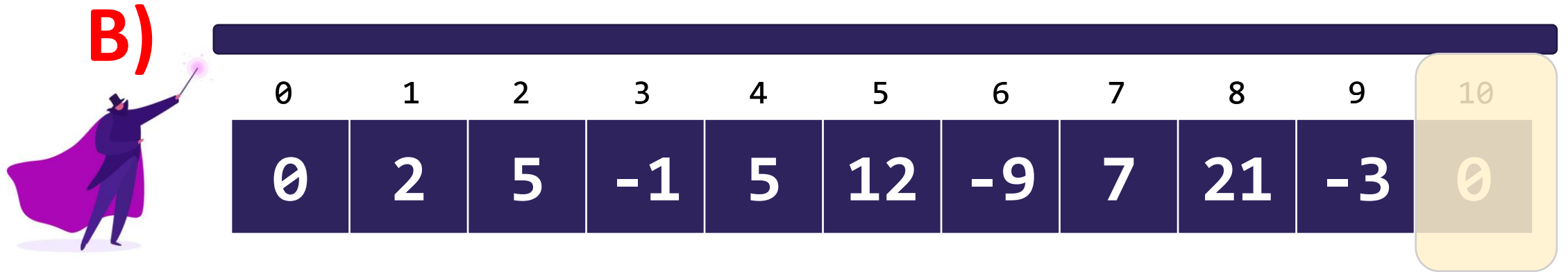
# Capacity and Resizing

- Capacity = length of underlying array
- Size = number of user-added elements
- What happens if we run out of space? (`size == capacity`)



# Capacity and Resizing

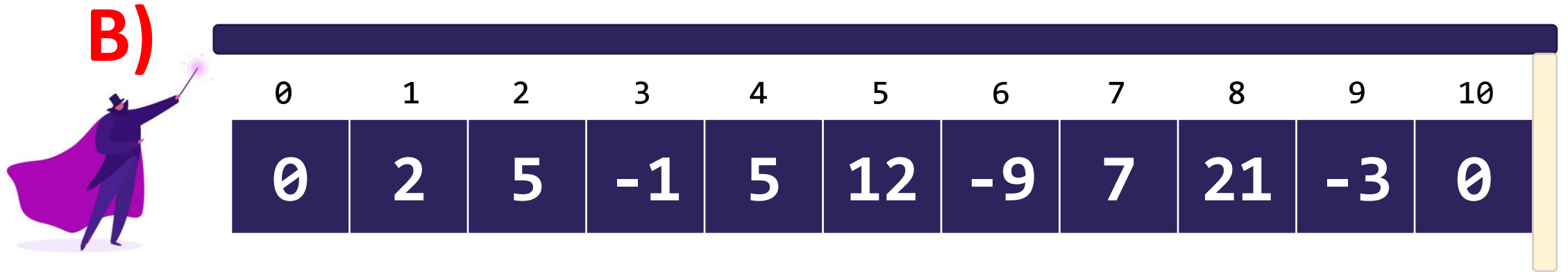
- Capacity = length of underlying array
- Size = number of user-added elements
- What happens if we run out of space? (`size == capacity`)



```
a1.add(2);
```

# Capacity and Resizing

- Capacity = length of underlying array
- Size = number of user-added elements
- What happens if we run out of space? (`size == capacity`)

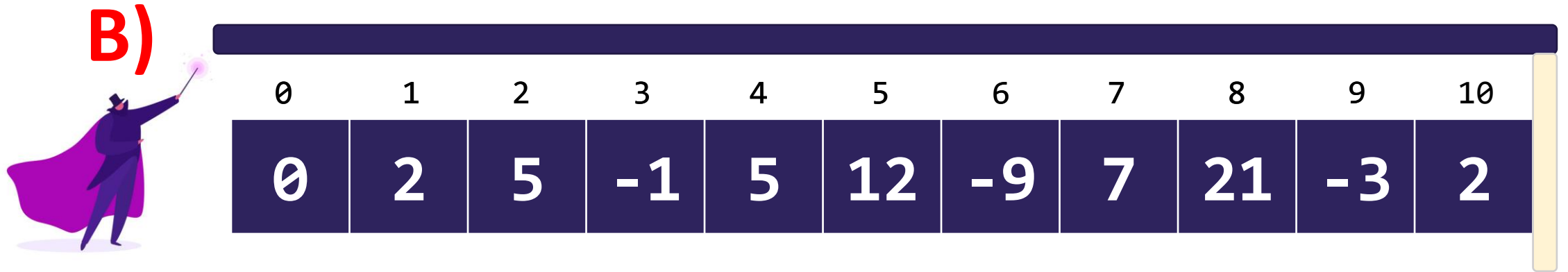


```
a1.add(2);
```



# Capacity and Resizing

- Capacity = length of underlying array
- Size = number of user-added elements
- What happens if we run out of space? (`size == capacity`)



```
a1.add(2);
```

# Capacity and Resizing

- Capacity = length of underlying array
- Size = number of user-added elements
- What happens if we run out of space? (`size == capacity`)
  - We make a new (bigger array) and copy things over
  - Another layer to the resizing illusion!
  
- In reality, we don't typically add a single spot
  - What happens if we add again?