

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:

What did you have breakfast today?

[Respond on sli.do!](#)

Instructor: Miya Natsuhara

TAs:

Arohan	Shiven	Yuntong	Anya
Sreshta	Vrinda	Amiya	Anisha
Rushil	Gavin	Sahana	Trien
Sean B	Shreya	Anirudh	Neal
Chloe	Jonah	Rohan	Evan
Jenny	Renee	Crystal	Rena
Nate	Chris	Eeshani	
Saachi	Ishita	Prakshi	
Hawa	Kuhu	Aidan	
Maggie	Kavya	Cora	
Sean E	Misha	Nhan	

Music: [CSE 123 26sp Lecture Tunes](#)

Announcements

- Programming Assignment 0 is out now, and is due Wed April 15 at 11:59pm!
- Looking ahead:
 - First resubmission opens Wed, April 15 (once C0 feedback is out)
 - 10 days out from our first quiz on Tues, April 21 (more details next week)

Lecture Outline: Revisiting Reflections

- **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)
- When discussing reflections with others...
 - 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: Interface vs. Implementation

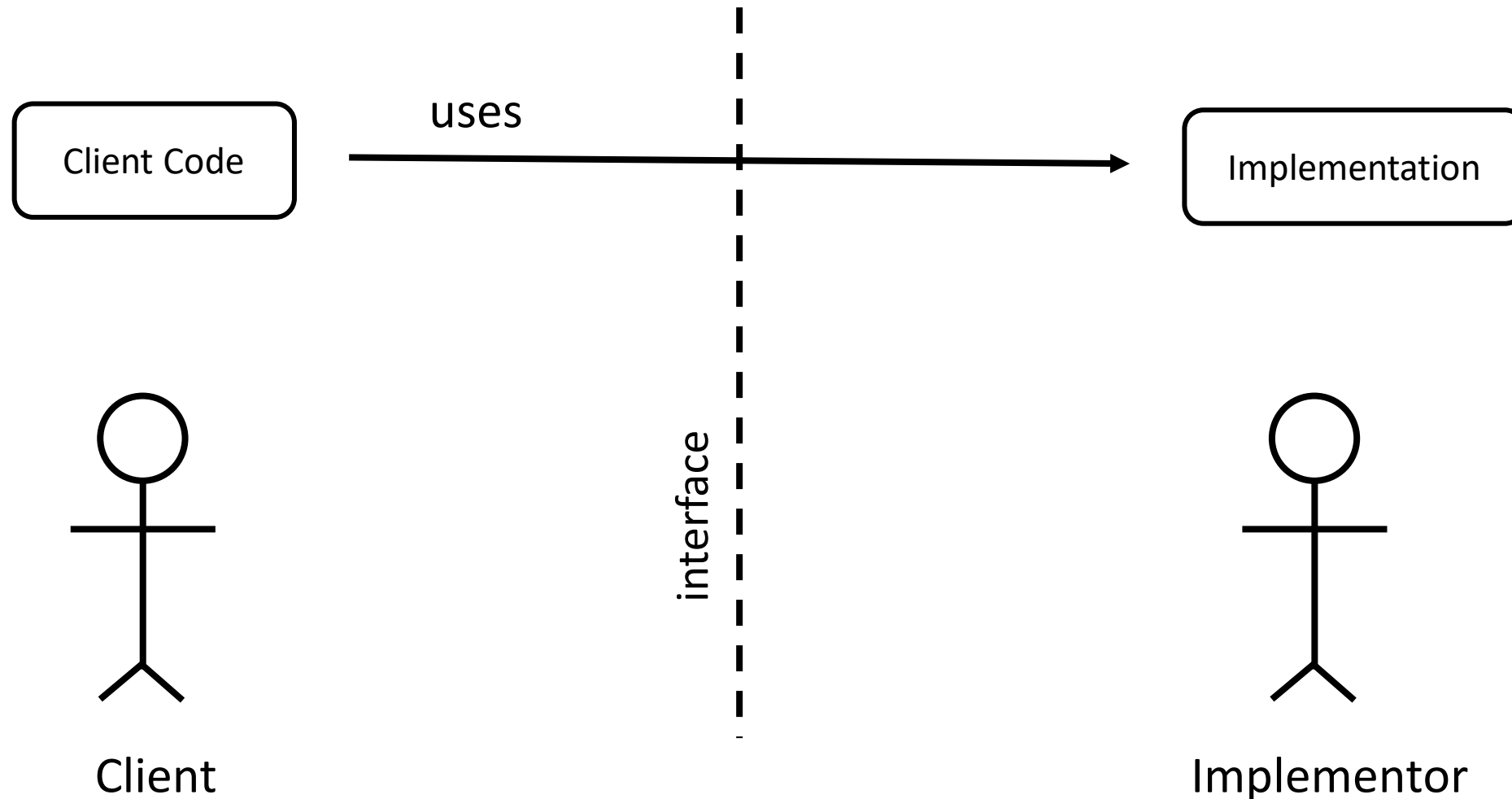
- Revisiting Reflections
- **Interface vs. Implementation** ◀
- Implementing ArrayList

Interface vs. 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 vs. Implementor



Arrays vs. ArrayLists (Review)

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

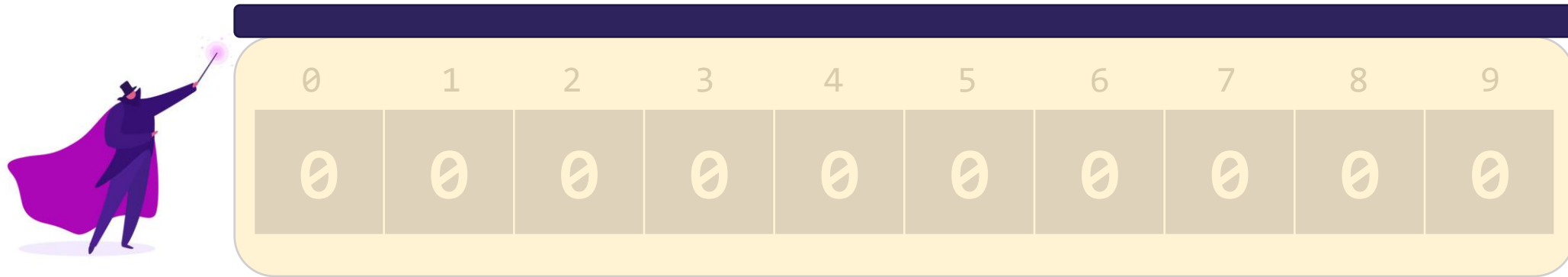
- Choose appropriate fields based on behavior
- Just requires some thinking outside the box

Lecture Outline: Implementing ArrayList

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

ArrayIntLists (step 1)

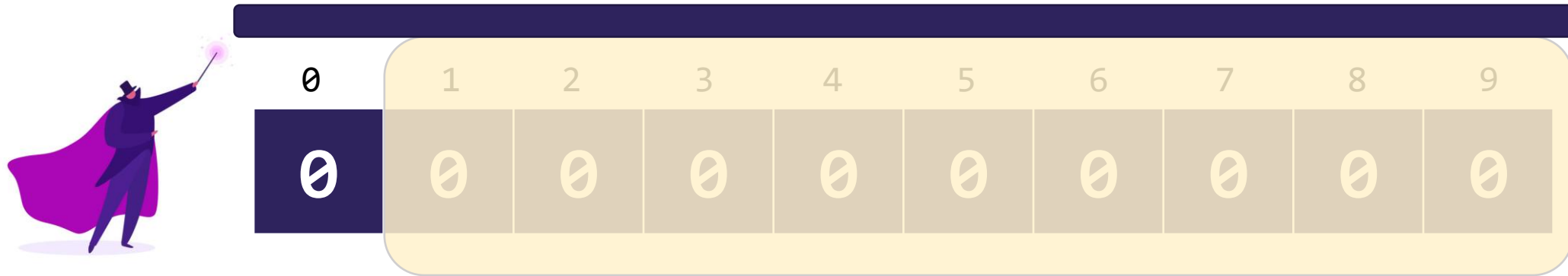
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
 - `int[] elementData;` // Where we store elements
 - `int size;` // Storage boundary



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

ArrayIntLists (step 2)

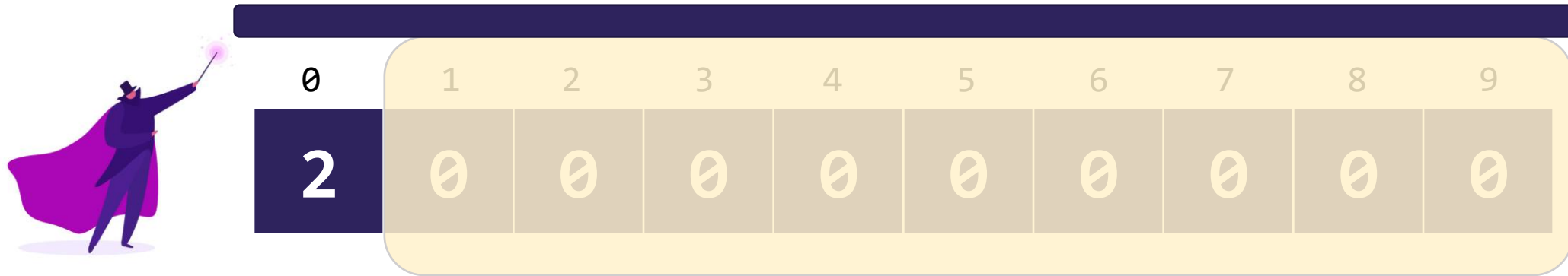
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
 - `int[] elementData;` // Where we store elements
 - `int size;` // Storage boundary



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

ArrayIntLists (step 3)

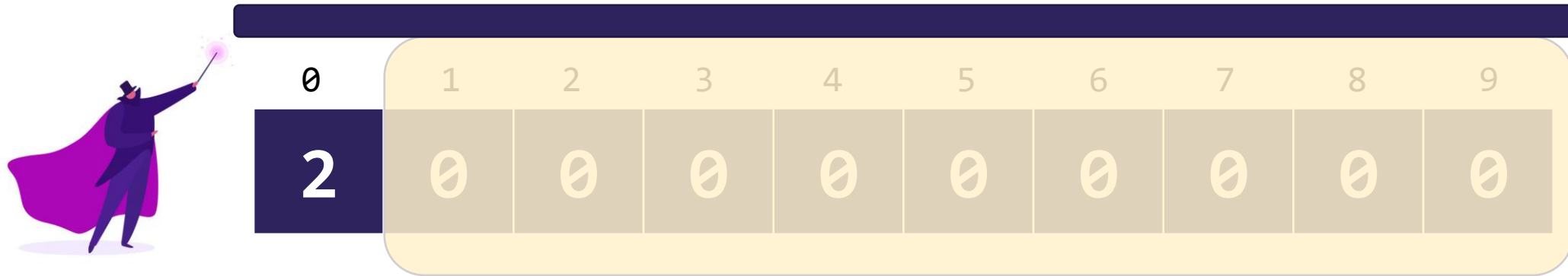
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
 - `int[] elementData;` // Where we store elements
 - `int size;` // Storage boundary



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

ArrayIntLists (step 4)

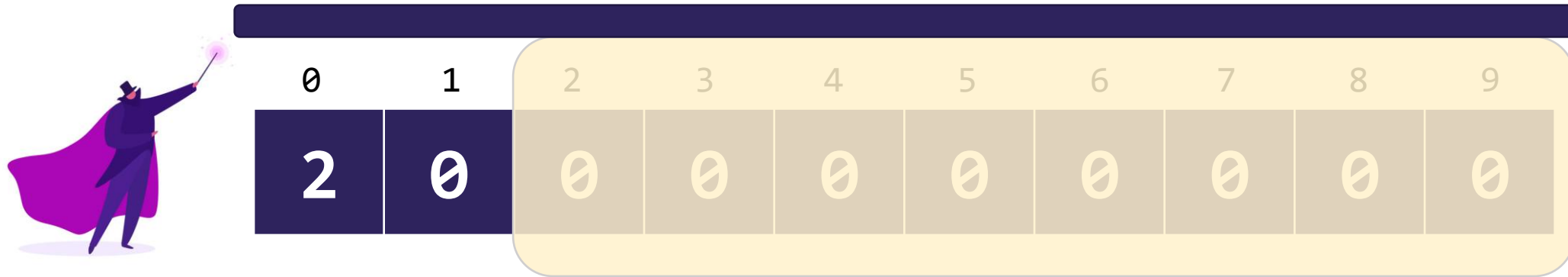
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
 - `int[] elementData;` // Where we store elements
 - `int size;` // Storage boundary



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

ArrayIntLists (step 5)

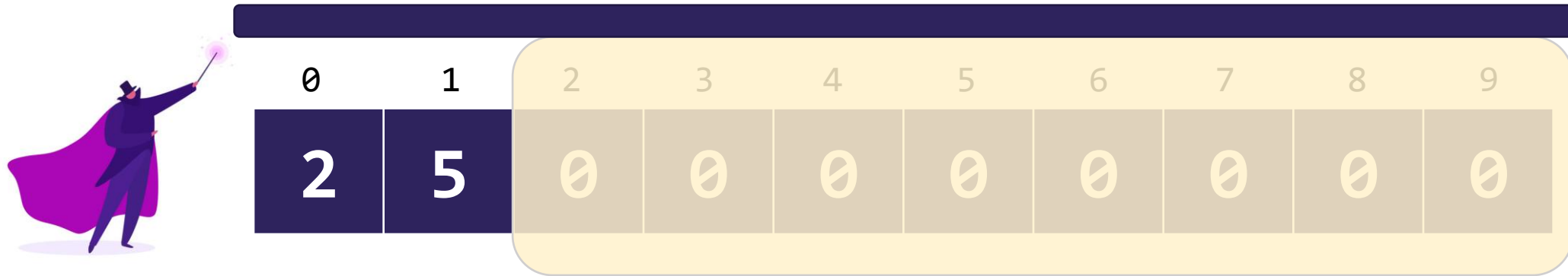
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
 - `int[] elementData;` // Where we store elements
 - `int size;` // Storage boundary



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

ArrayIntLists (step 6)

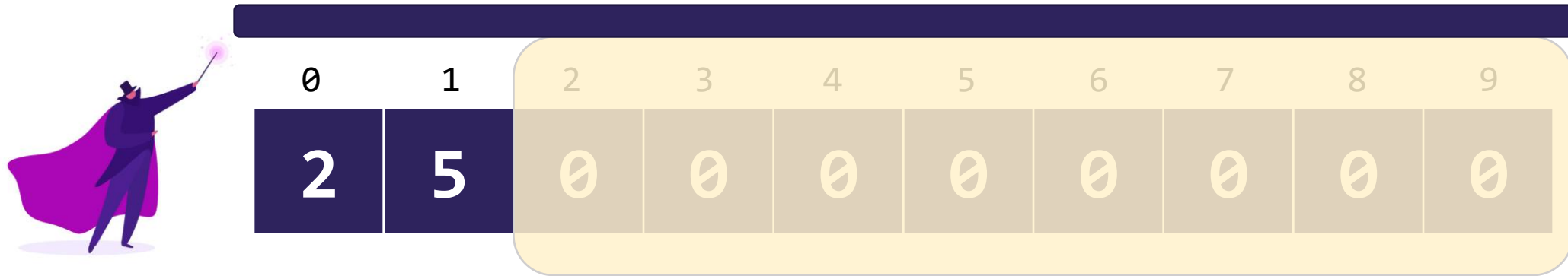
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
 - `int[] elementData;` // Where we store elements
 - `int size;` // Storage boundary



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

ArrayIntLists (step 7)

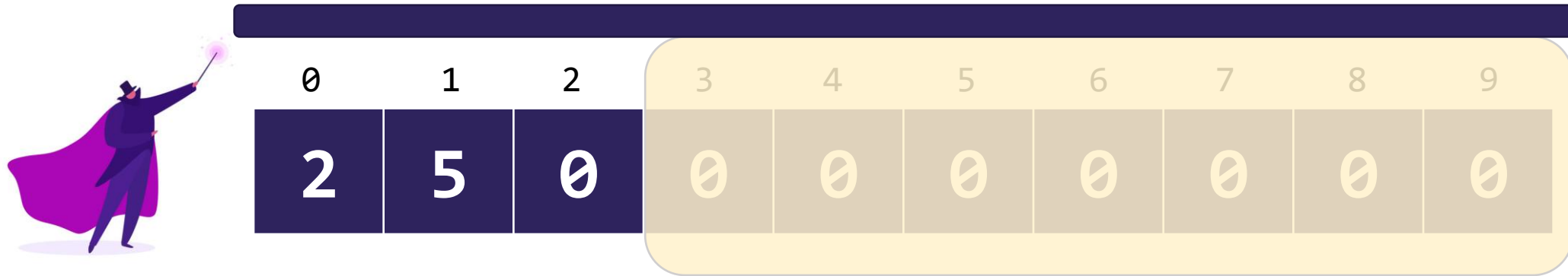
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
 - `int[] elementData;` // Where we store elements
 - `int size;` // Storage boundary



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

ArrayIntLists (step 8)

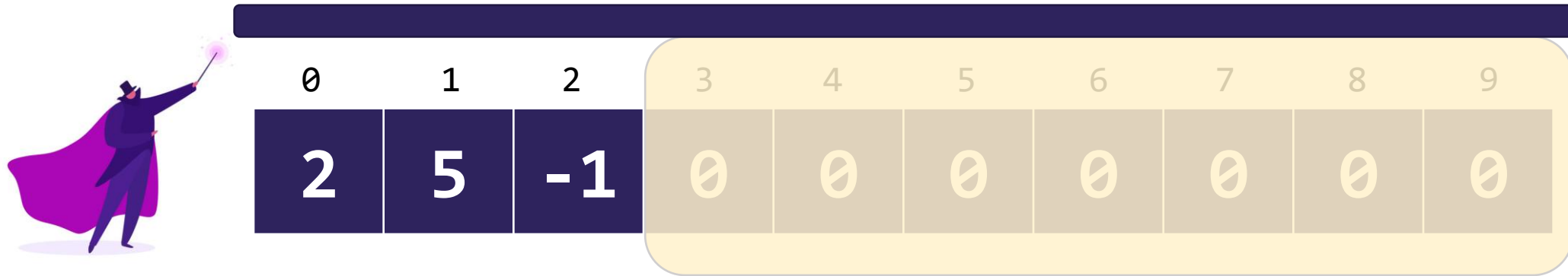
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
 - `int[] elementData;` // Where we store elements
 - `int size;` // Storage boundary



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

ArrayIntLists (step 9)

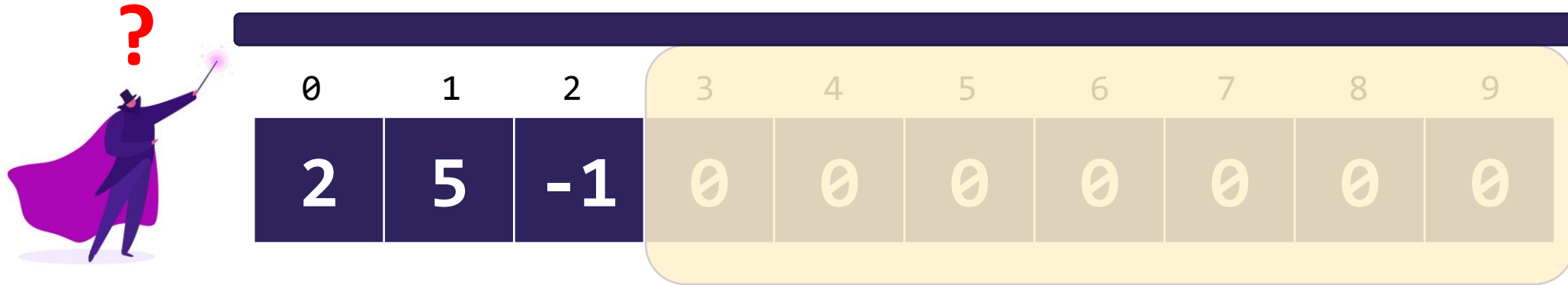
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
 - `int[] elementData;` // Where we store elements
 - `int size;` // Storage boundary



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

ArrayIntLists (step 10)

- 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

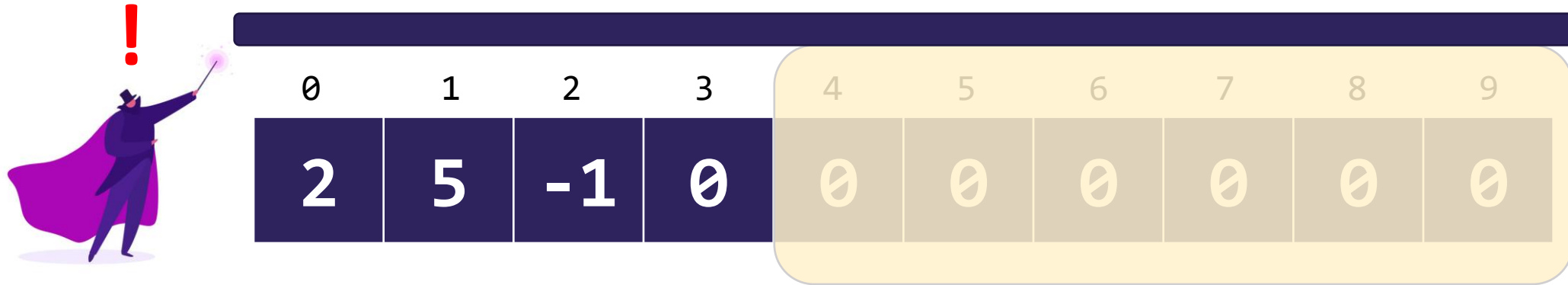


```
a1.add(0, 0);
```

brainstorming

ArrayIntLists (step 11)

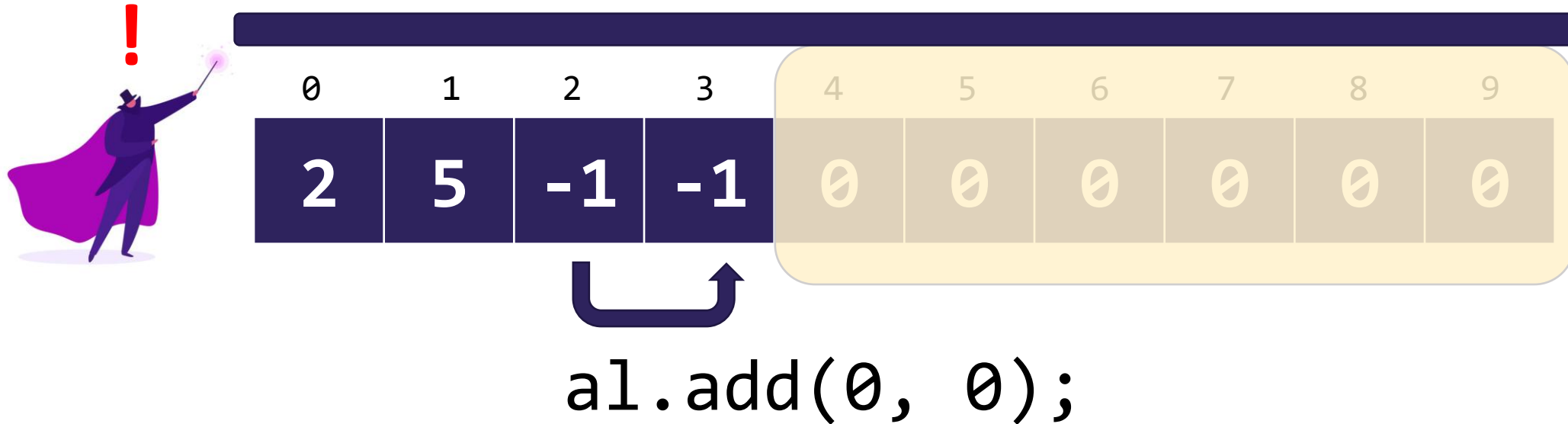
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
 - `int[] elementData;` // Where we store elements
 - `int size;` // Storage boundary



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

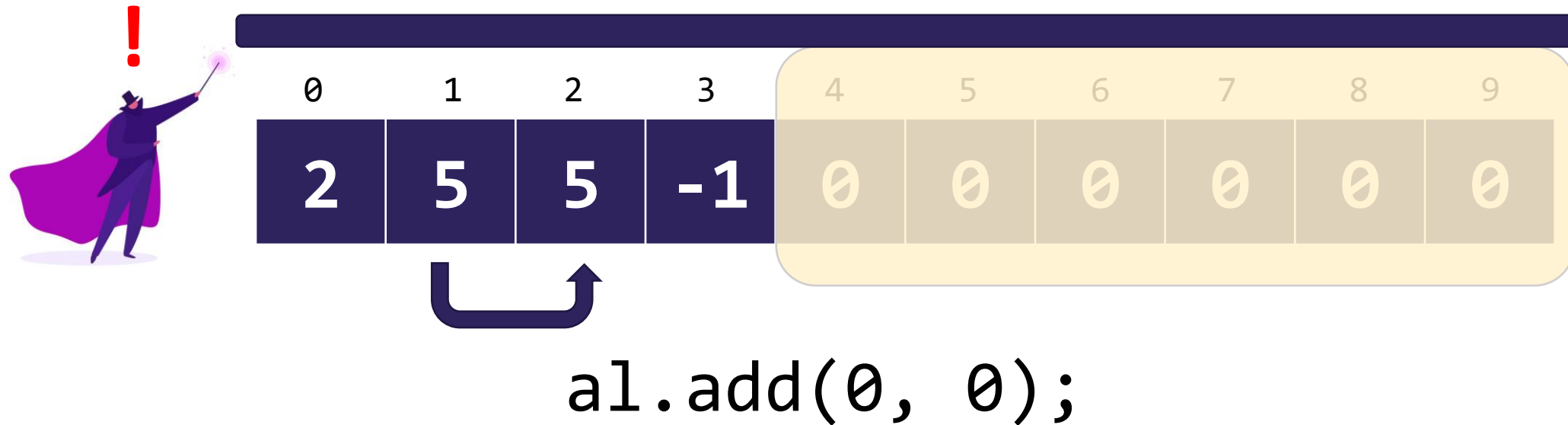
ArrayIntLists (step 12)

- 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



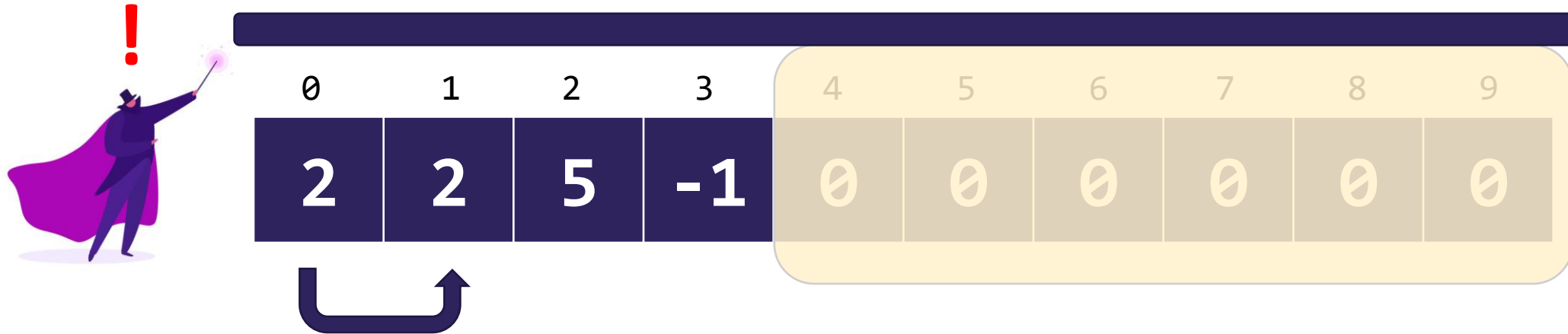
ArrayIntLists (step 13)

- 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 (step 14)

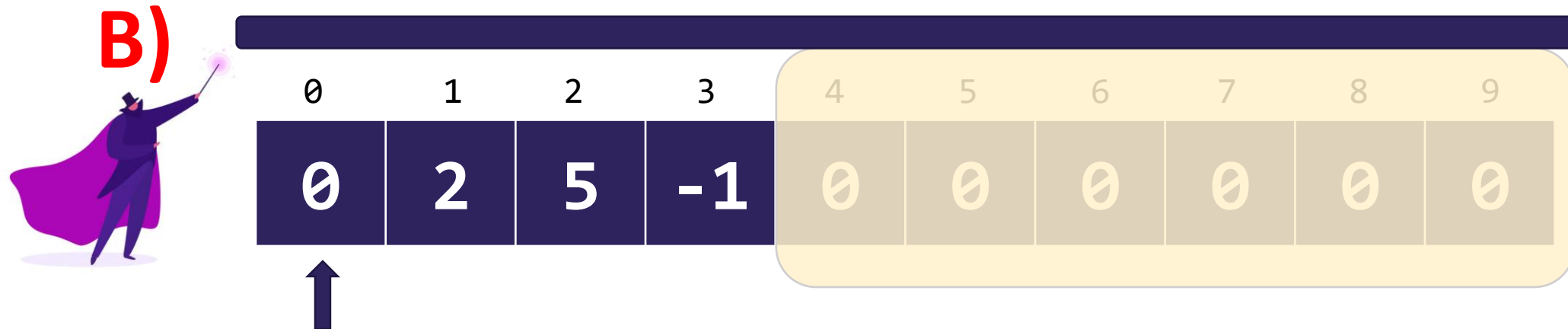
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
 - `int[] elementData;` // Where we store elements
 - `int size;` // Storage boundary



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

ArrayIntLists (step 15)

- 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



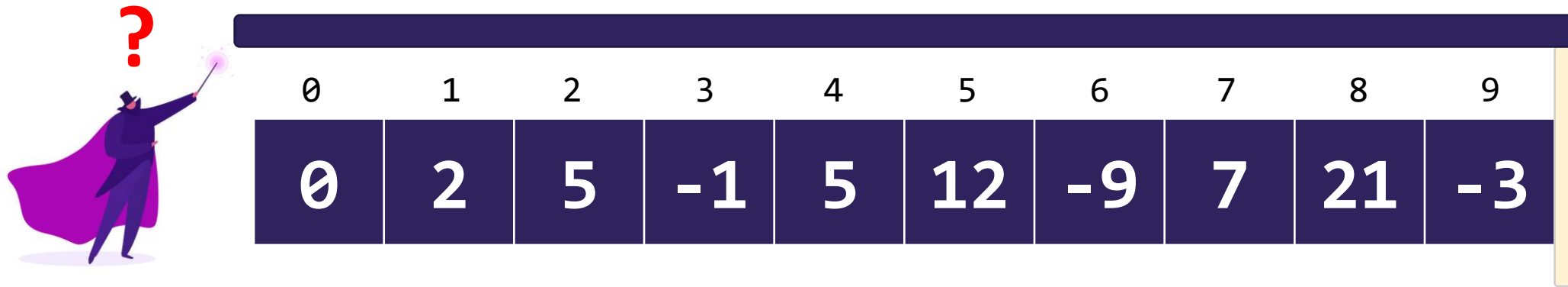
```
a1.add(0, 0);
```

ArrayIntLists Summary

- 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

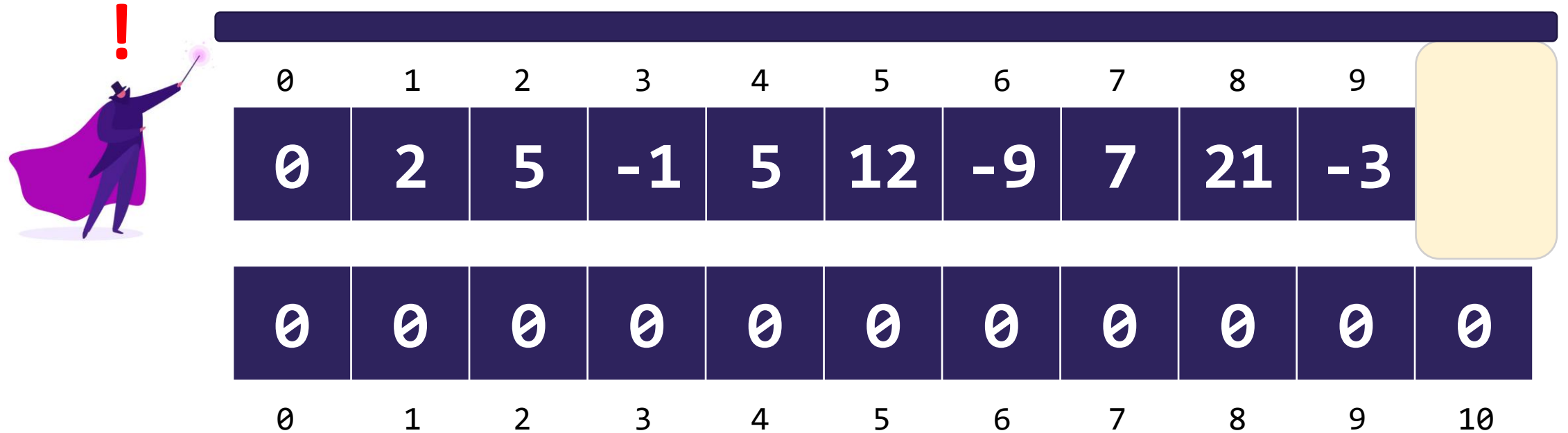
- Capacity = length of underlying array
- Size = number of user-added elements
- What happens if we run out of space? (`size == capacity`)



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

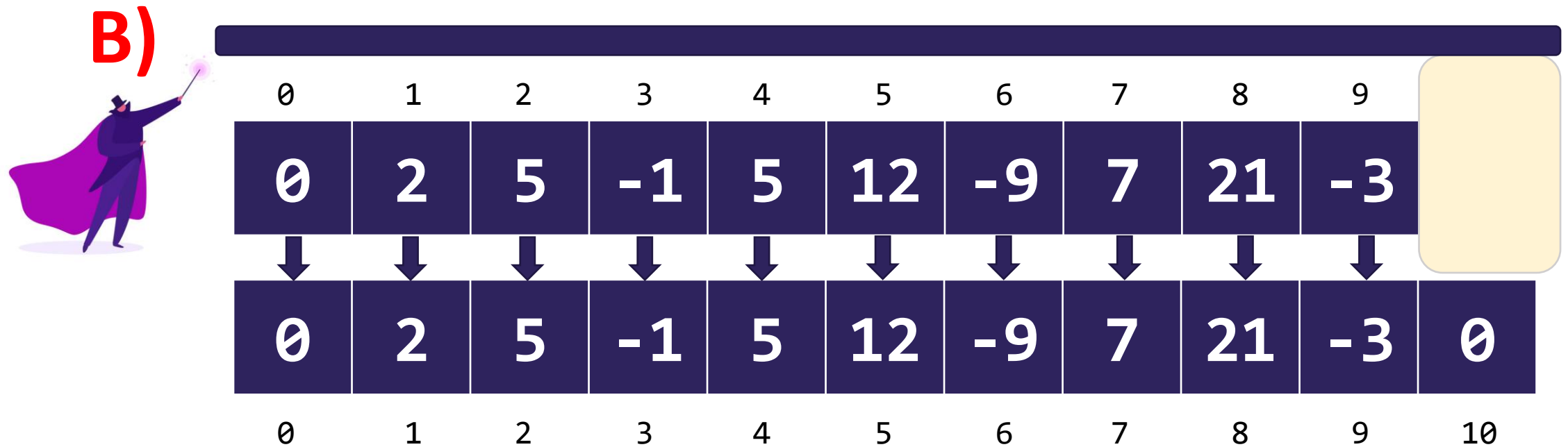
Capacity and Resizing (step 1)

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



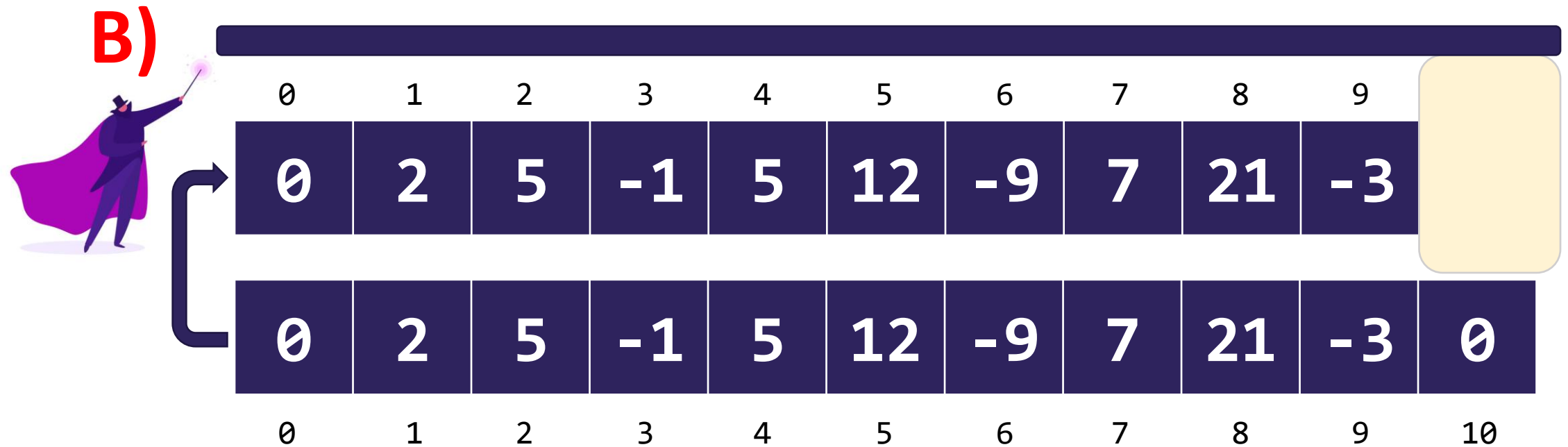
Capacity and Resizing (step 2)

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



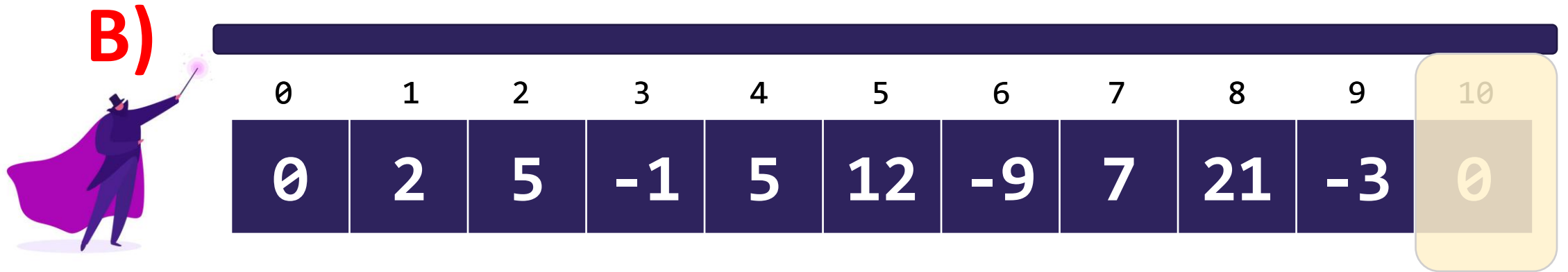
Capacity and Resizing (step 3)

- 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 (step 4)

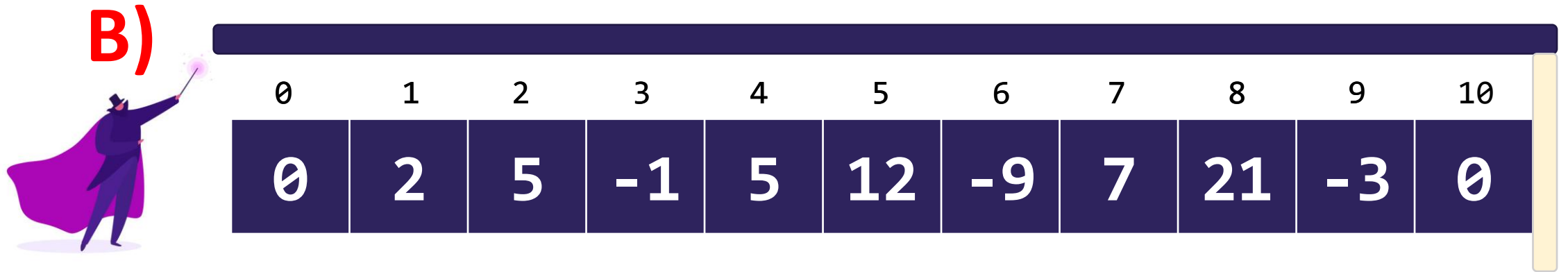
- Capacity = length of underlying array
- Size = number of user-added elements
- What happens if we run out of space? (`size == capacity`)



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

Capacity and Resizing (step 5)

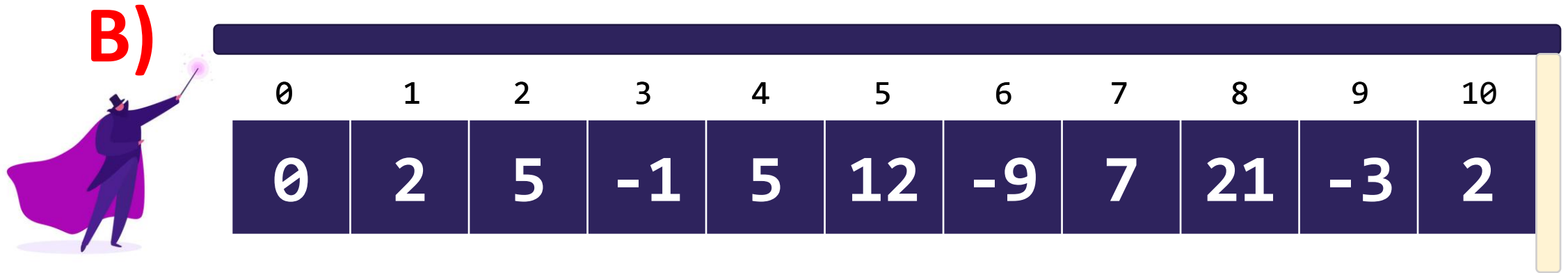
- Capacity = length of underlying array
- Size = number of user-added elements
- What happens if we run out of space? (`size == capacity`)



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

Capacity and Resizing (step 6)

- 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 Summary

- 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?