LEC 03

CSE 123

Implementing Data Structures; `ArrayIntList`

Questions during Class?

Raise hand or send here

sli.do #cse123A

BEFORE WE START

Talk to your neighbors:

Did you eat breakfast today? If so, what?

Instructor: Nathan Brunelle

TAs:

Arohan	Ashar	Neha	Rohini	Rushil
Ido	Zachary	Sebastian	Joshua	Sean
Hayden	Caleb	Justin	Heon	Rashad
Srihari	Benoit	Derek	Chris	Bhaumik
Kuhu	Kavya	Cynthia	Shreya	Ashley
Harshitha	Kieran	Marcus	Crystal	Eeshani
Prakshi	Packard	Cora	Dixon	Nichole
Niyati	Trien	Lawrence	Evan	Cady
Ziao				

Announcements

- Programming Assignment 0 due Wed April 16 at 11:59pm!

- Looking ahead:
  - First resubmission opens Wed, April 16
  - First quiz on Tue, April 22 (more detail next week)
  - C1 will have one (small) part demonstrating a feature of VSCode

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

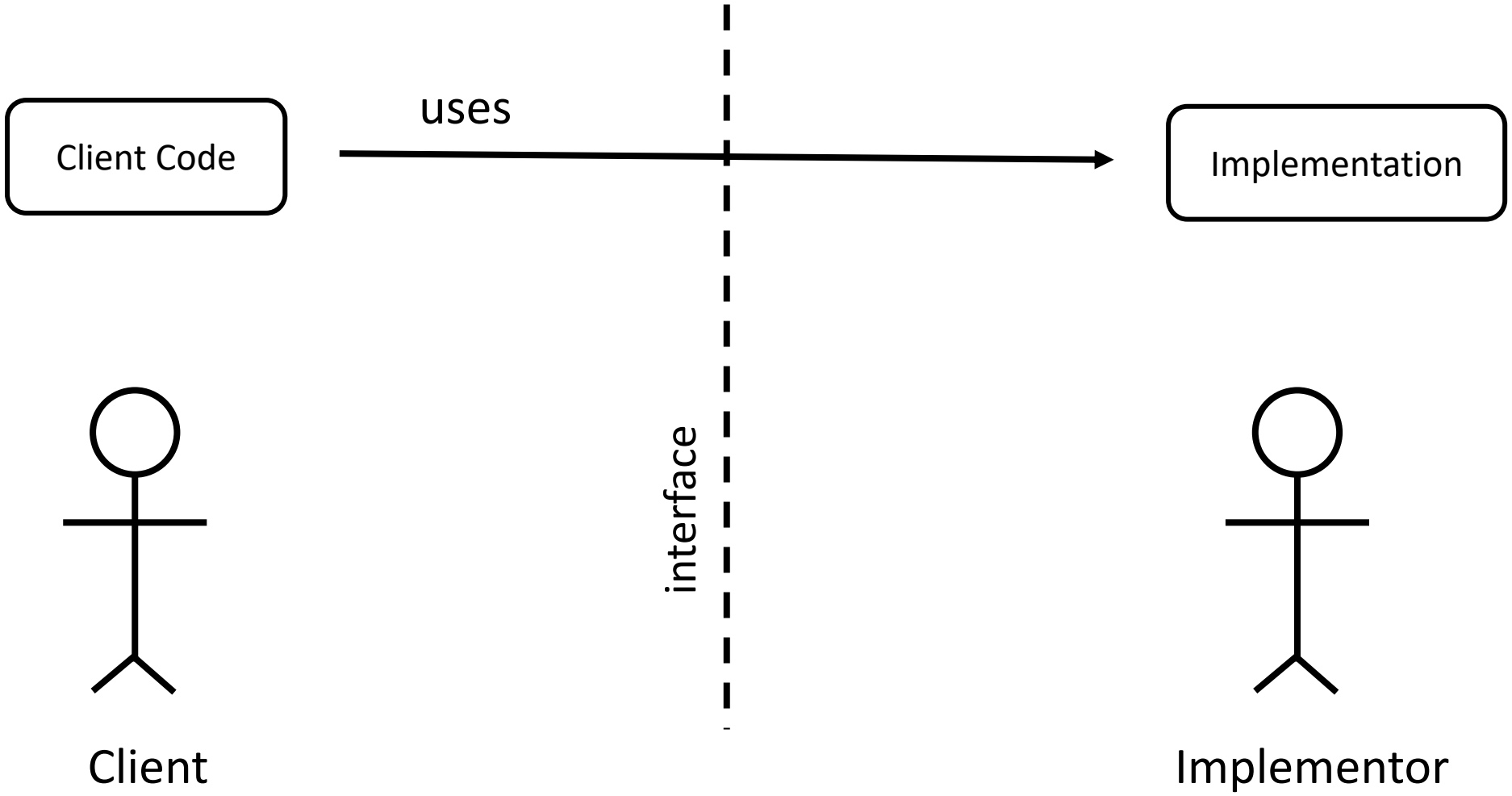
- Revisiting Reflections
- **Interface vs. Implementation**
- Implementing `ArrayList`

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
`int[] arr = new int[x];`	`List<Integer> al = new ArrayList<>();`
`int y = arr[0]`	`int y = al.get(0);`
-	`al.add(2);`
`arr[0] = 5;`	`al.set(0, 5);`
`int length = arr.length;  // Always x`	`int size = al.size();  // Matches # of things added`

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

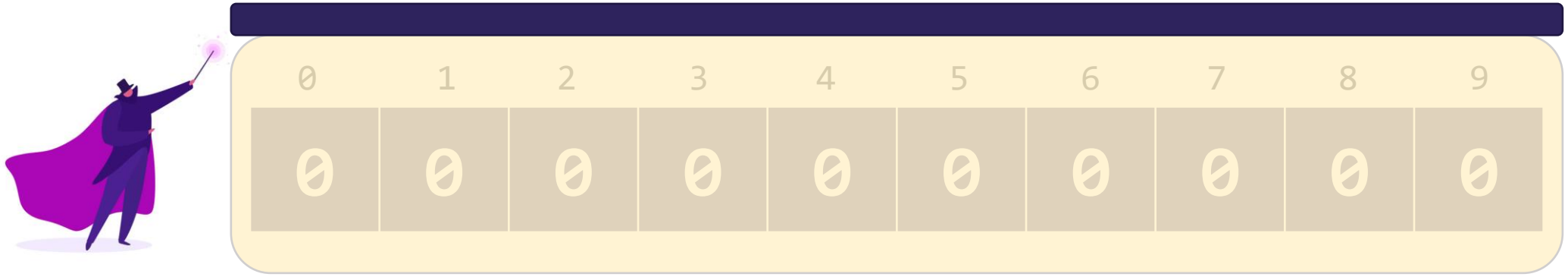
  - Choose appropriate fields based on behavior
- Just requires some thinking outside the box

Lecture Outline

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

ArrayIntLists

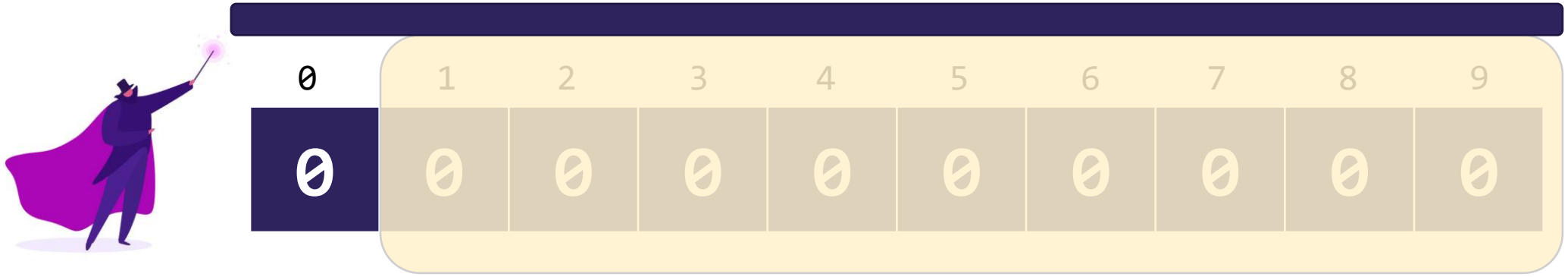
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

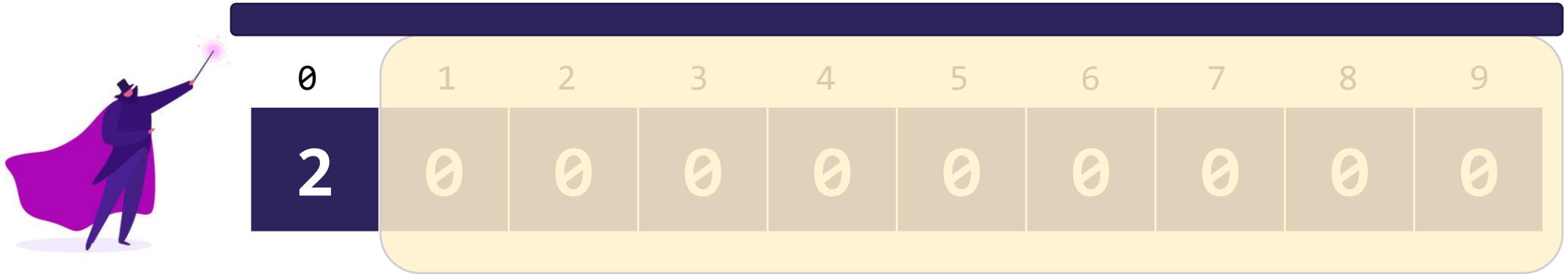
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

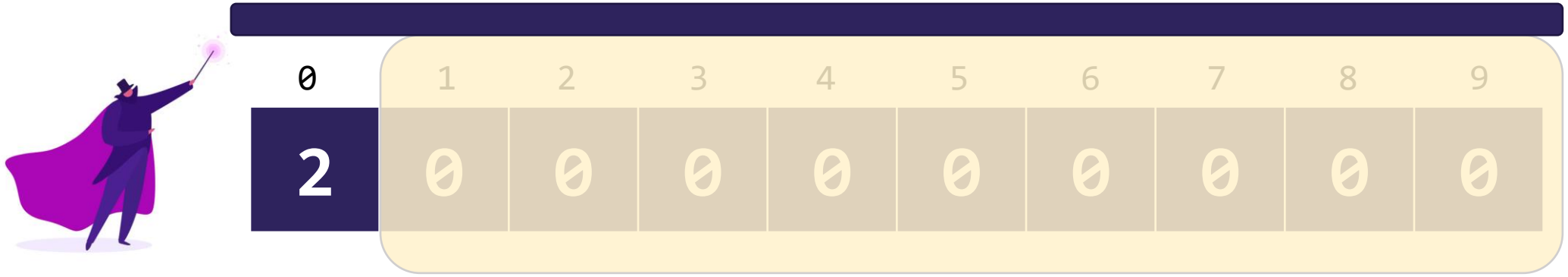
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

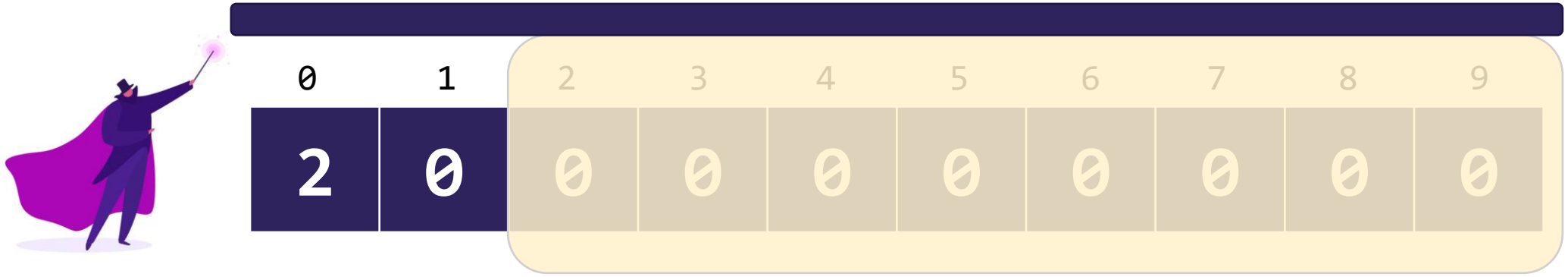
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




```
al.add(5);
```

ArrayIntLists

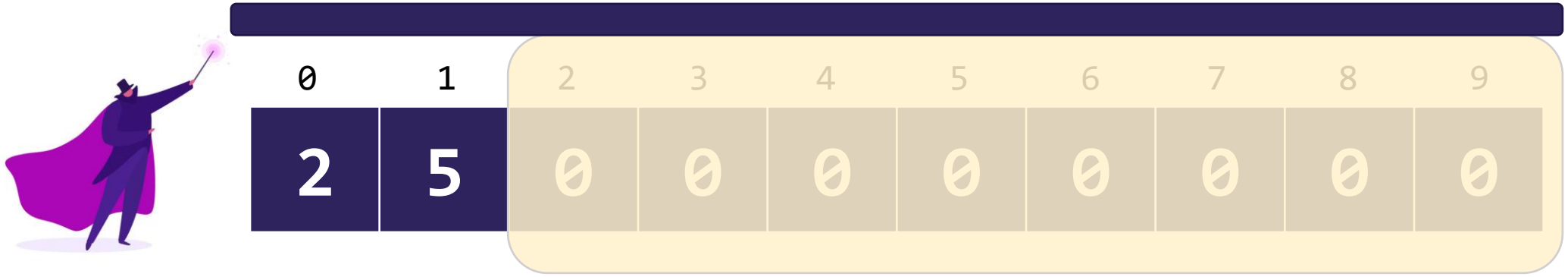
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




```
al.add(5);
```

ArrayIntLists

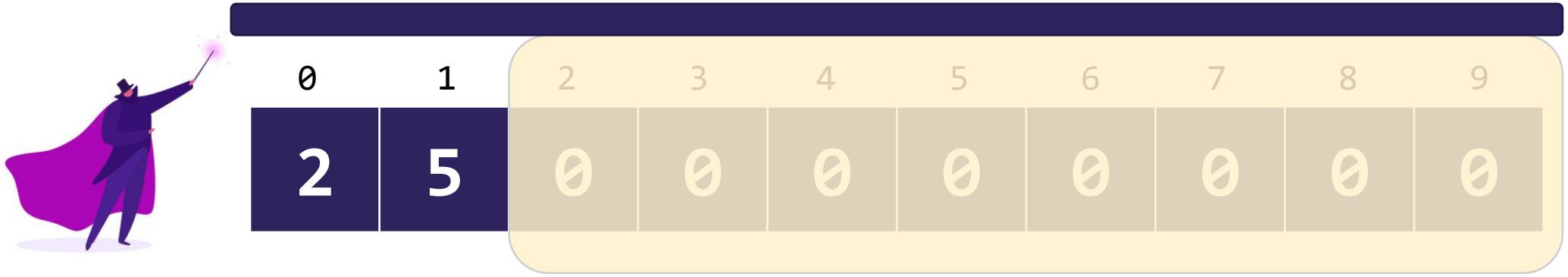
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




```
al.add(5);
```

ArrayIntLists

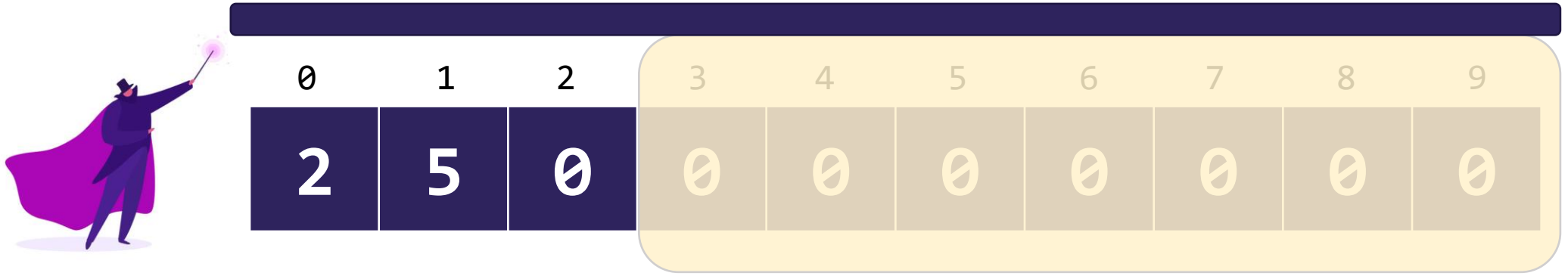
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

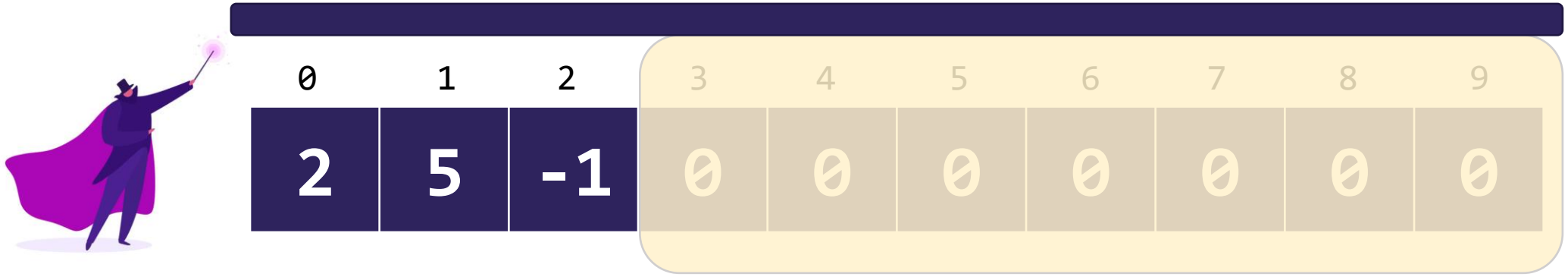
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

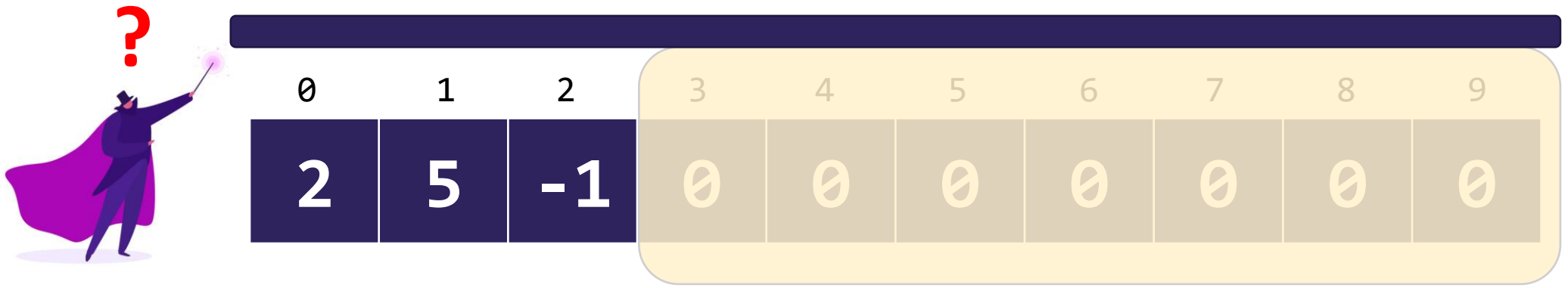
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

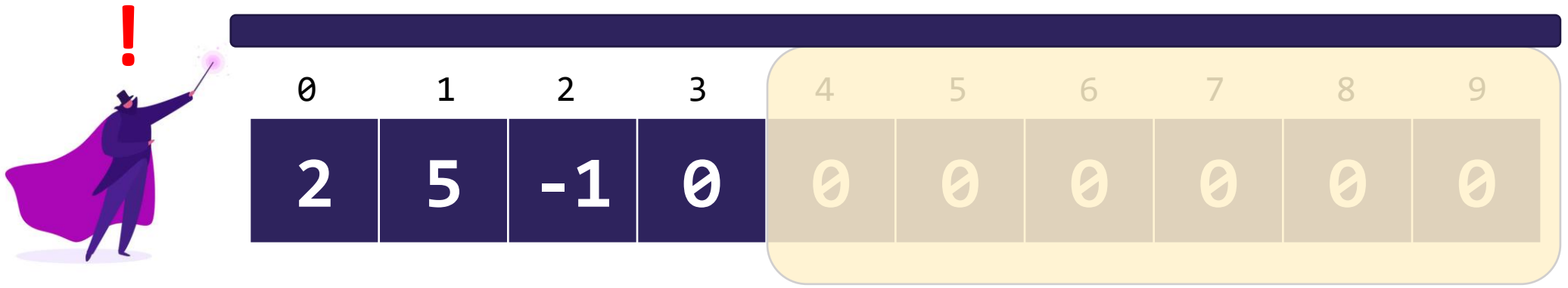
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

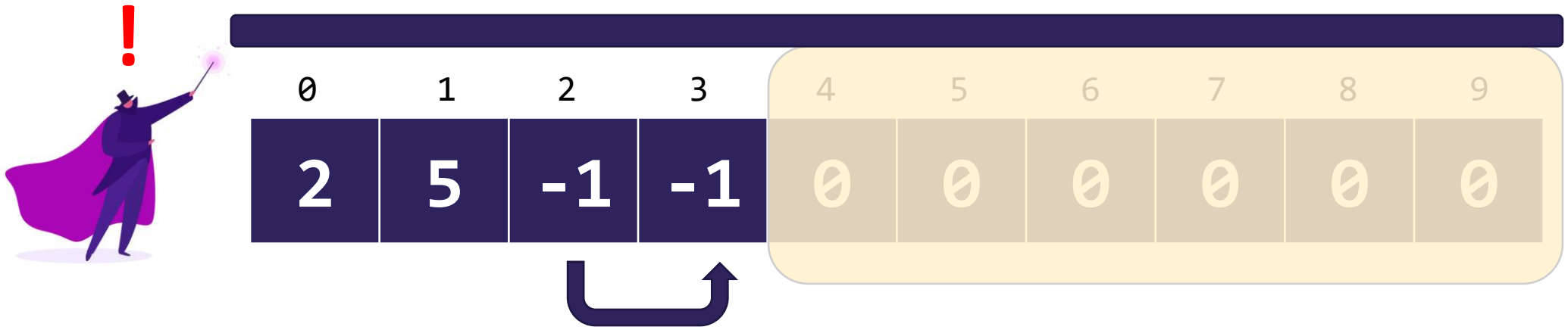
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

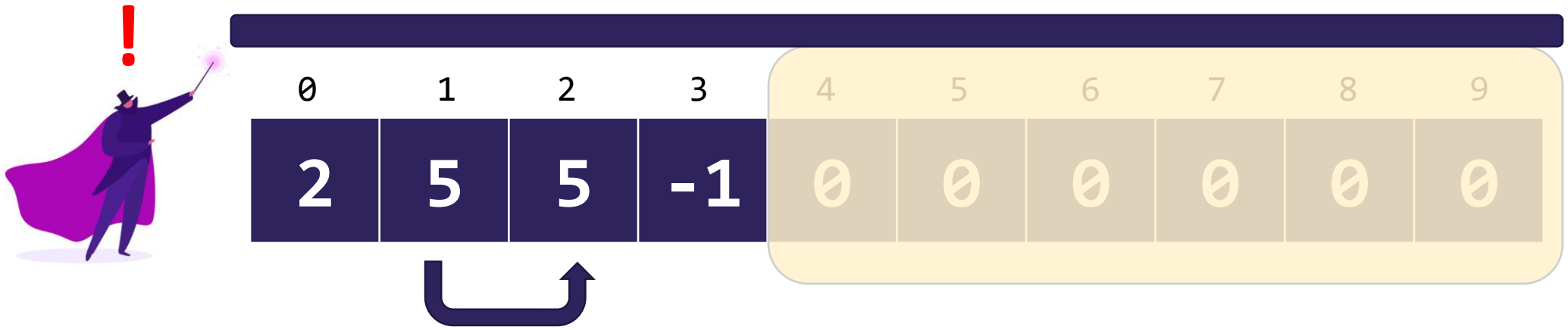
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

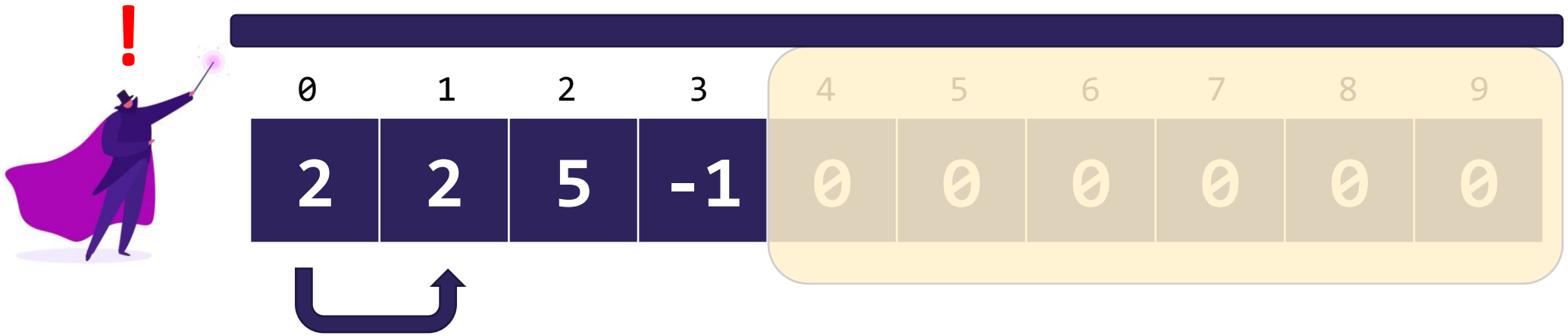
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

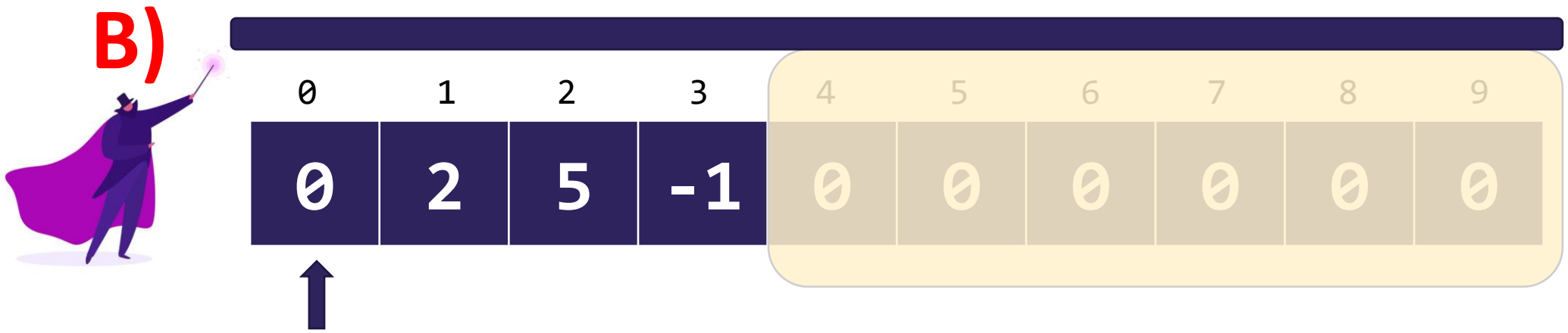
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




```
al.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`




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

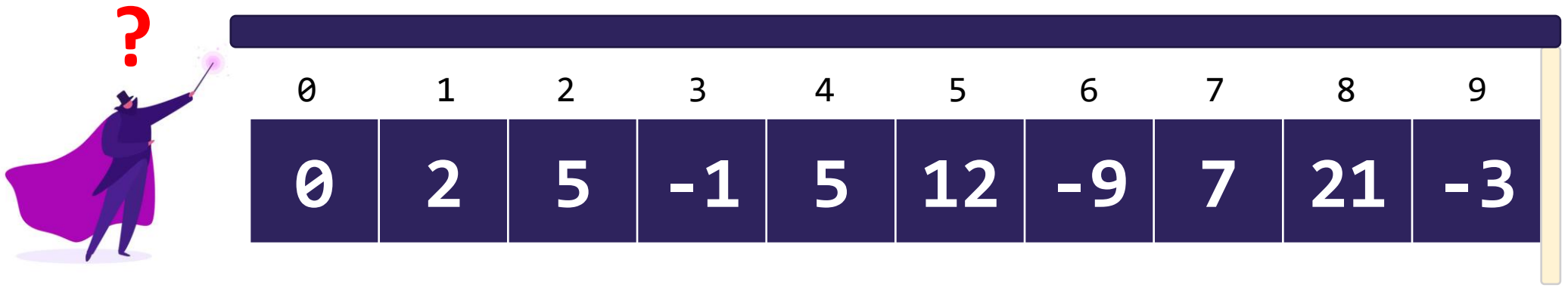
ArrayIntLists

- 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

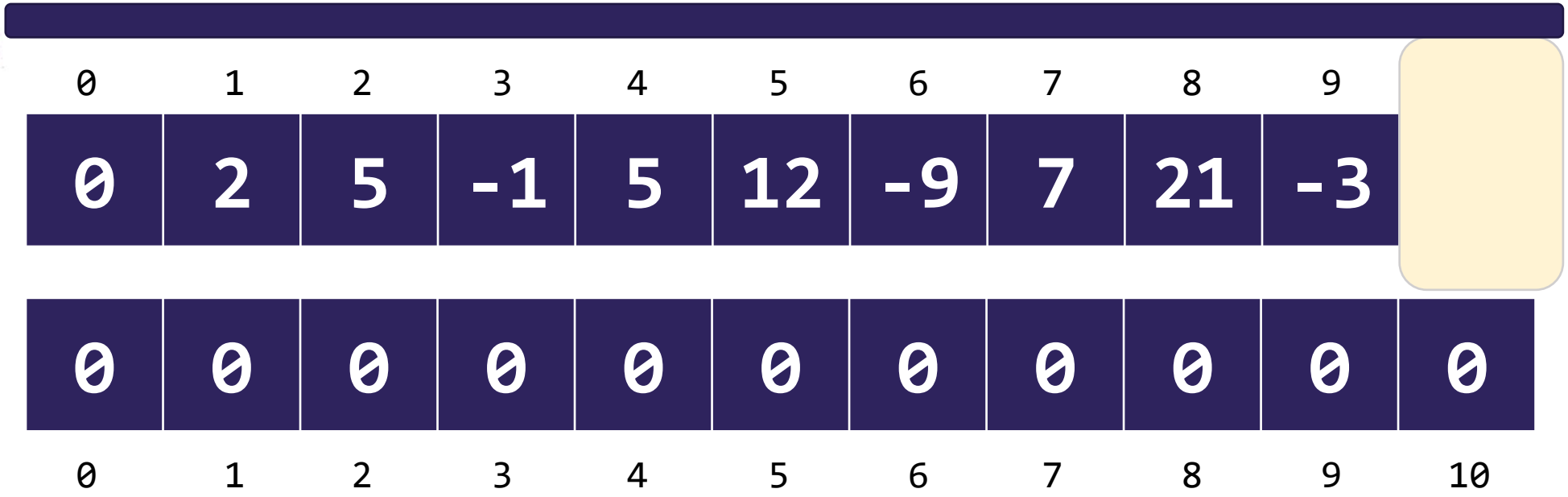
- `Capacity` = length of underlying array
- `Size` = number of user-added elements
- What happens if we run out of space? (`size == capacity`)




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

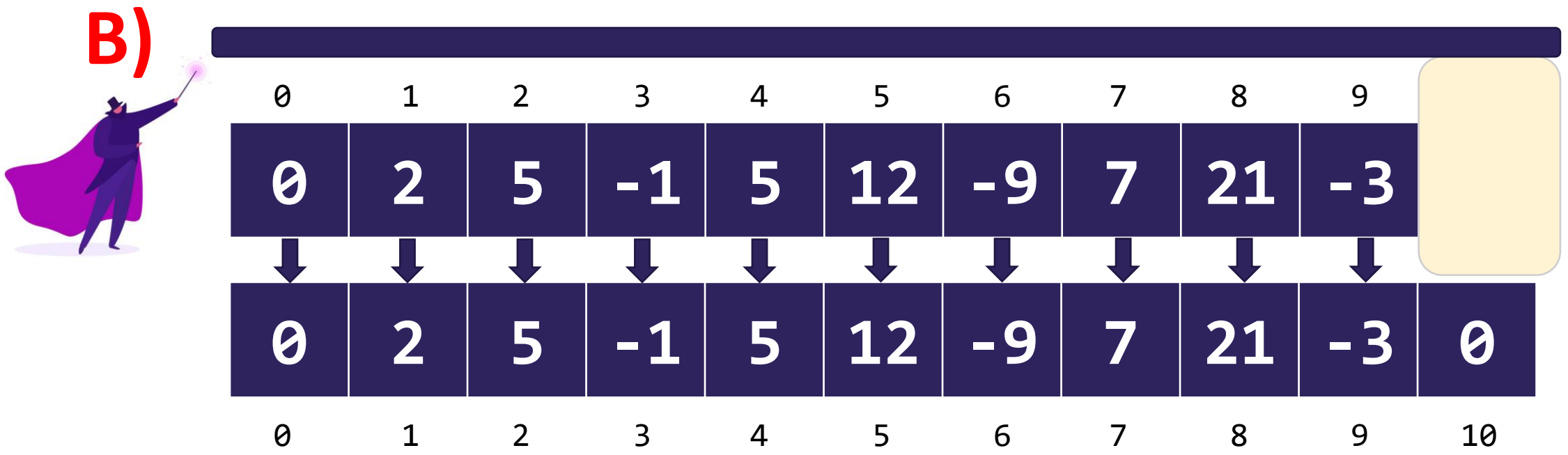
Capacity and Resizing

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



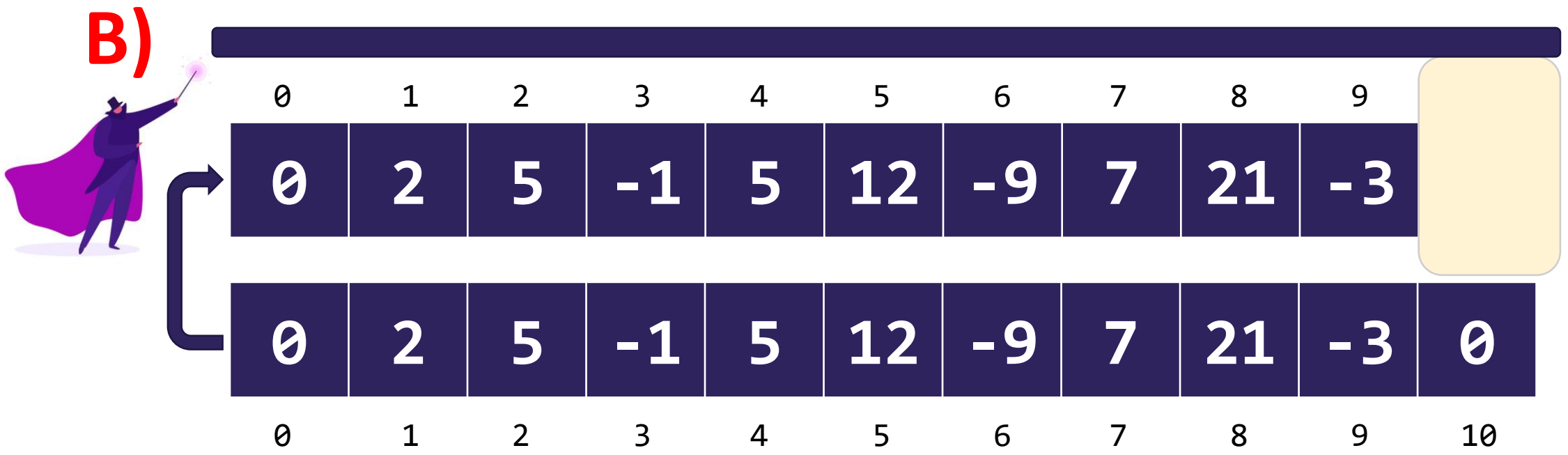
Capacity and Resizing

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



Capacity and Resizing

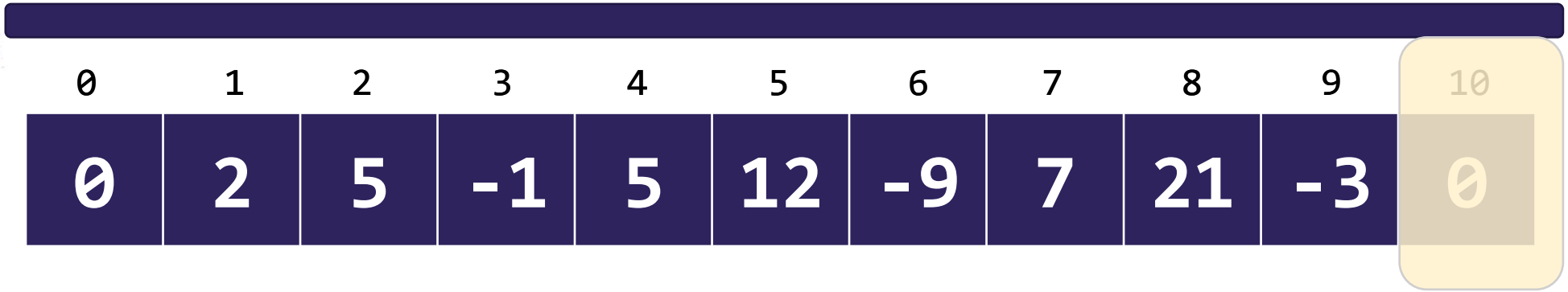
- `Capacity` = length of underlying array
- `Size` = number of user-added elements
- What happens if we run out of space? (`size == capacity`)



Capacity and Resizing

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

B)

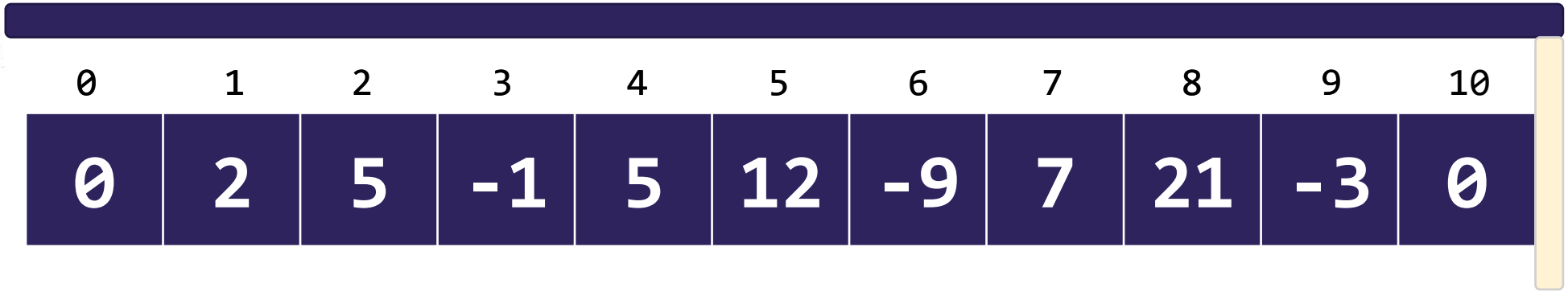



```
al.add(2);
```

Capacity and Resizing

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

B)

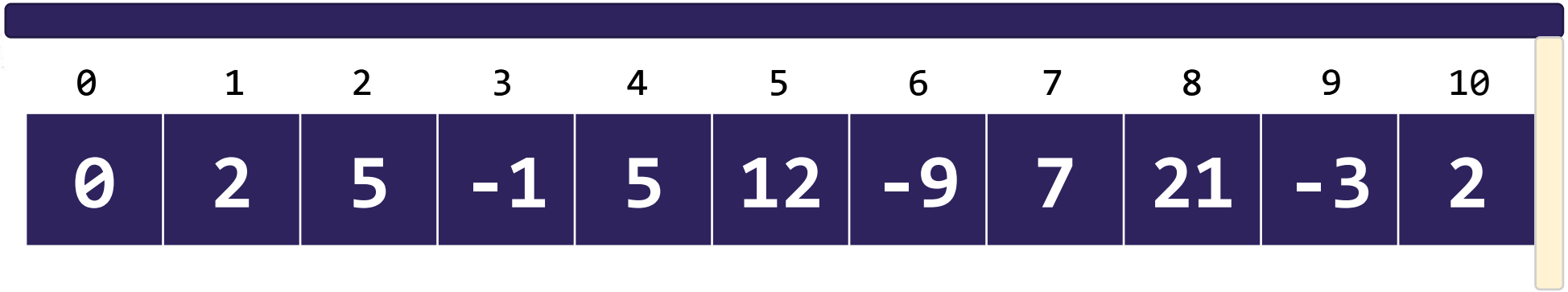



```
al.add(2);
```

Capacity and Resizing

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

B)




```
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`)
  - 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?