LEC 03

CSE 123

ArrayIntList

BEFORE WE START

Talk to your neighbors:

Did you eat breakfast today? If so, what?

Instructor: James Wilcox

Questions during Class?

Raise hand or send here

sli.do #cse123

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

C0 Reflection

- Video: *The Moral Bias Behind your Search Results*

- Q3: Do you think whoever comes up with moral rules and judgements surrounding search engine ranking results at Google should have that power / responsibility?

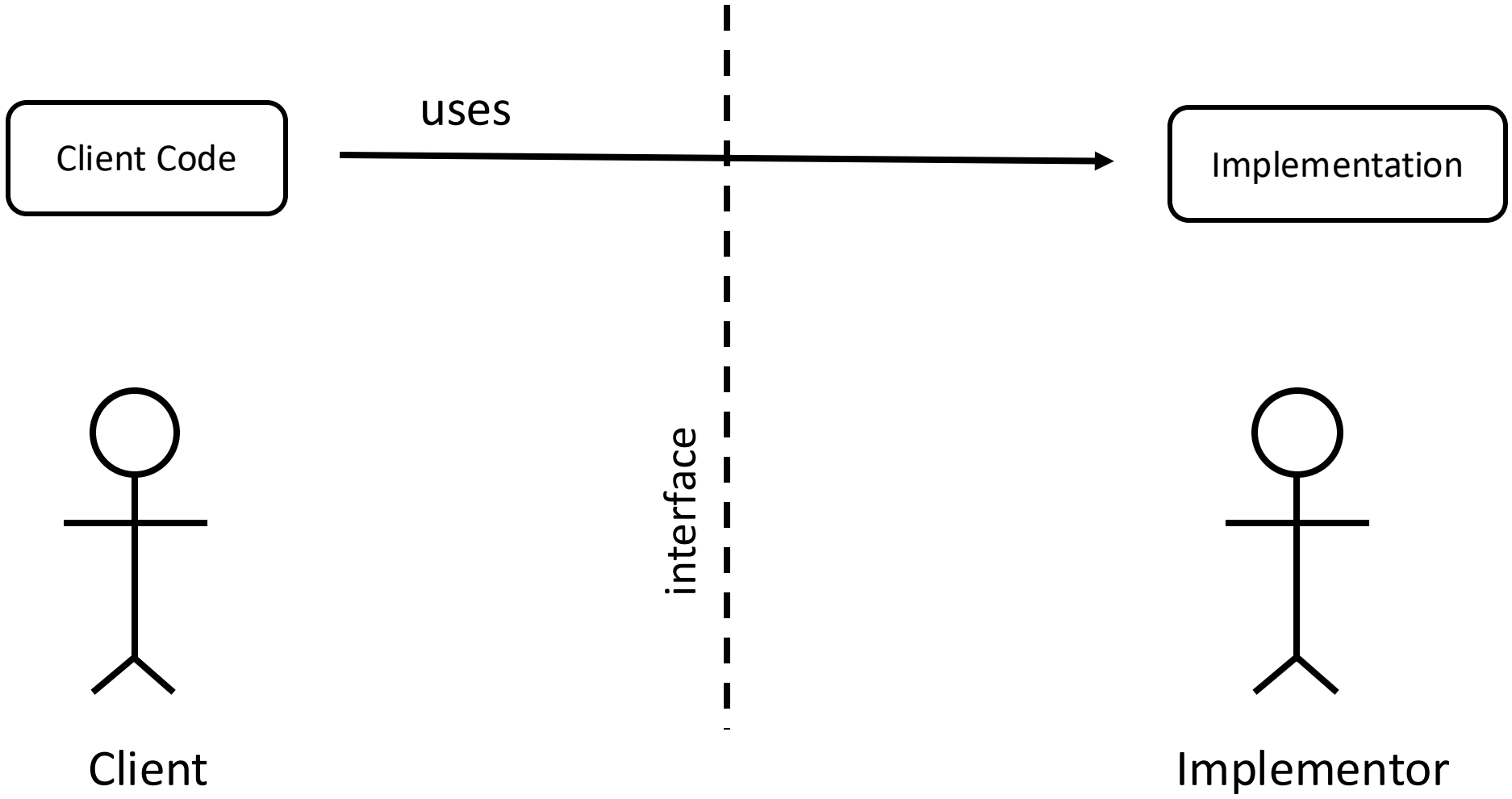
- Q5: Do you think that software reflects the biases of the programmer? Have you ever encountered bias programs / applications?

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

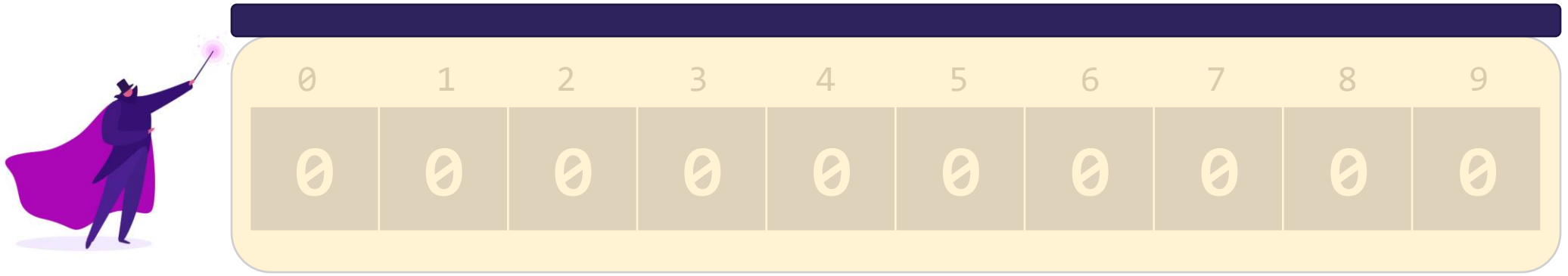
- No different from designing any other class!
  - Specified behavior (`List` interface):

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

ArrayIntLists

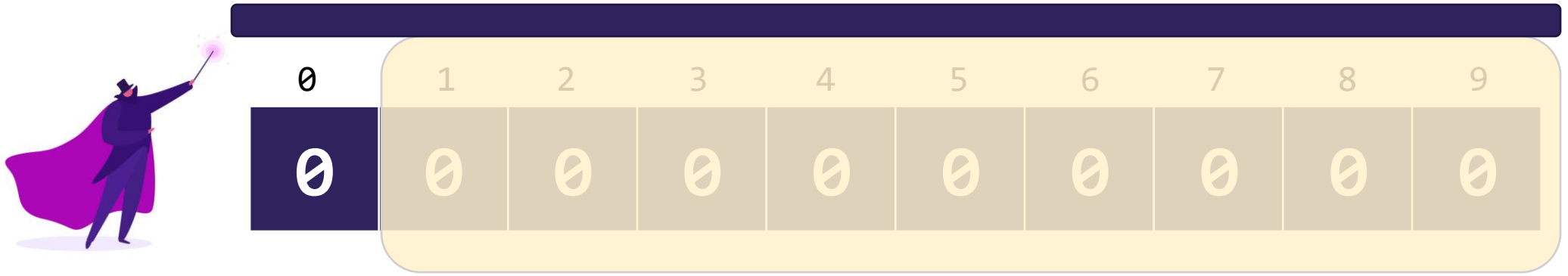
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

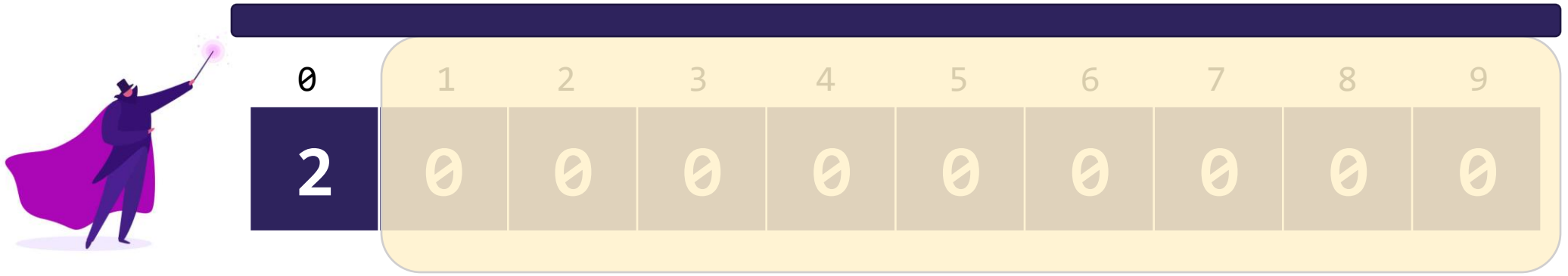
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

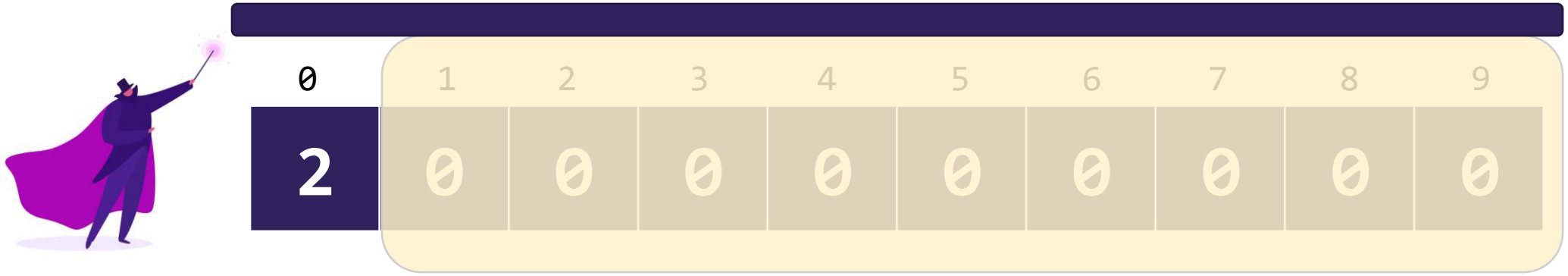
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

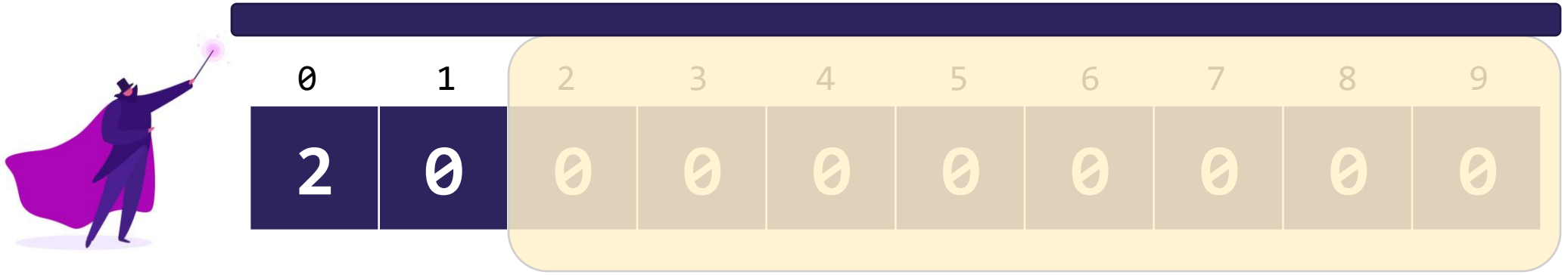
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

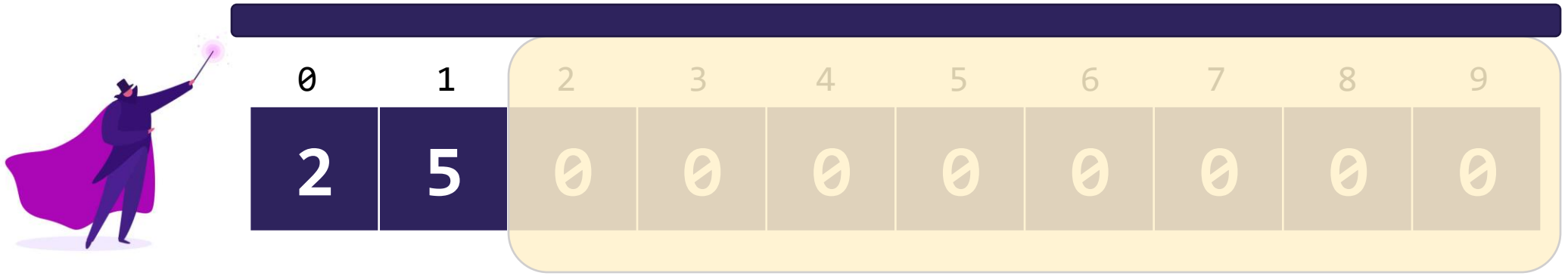
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

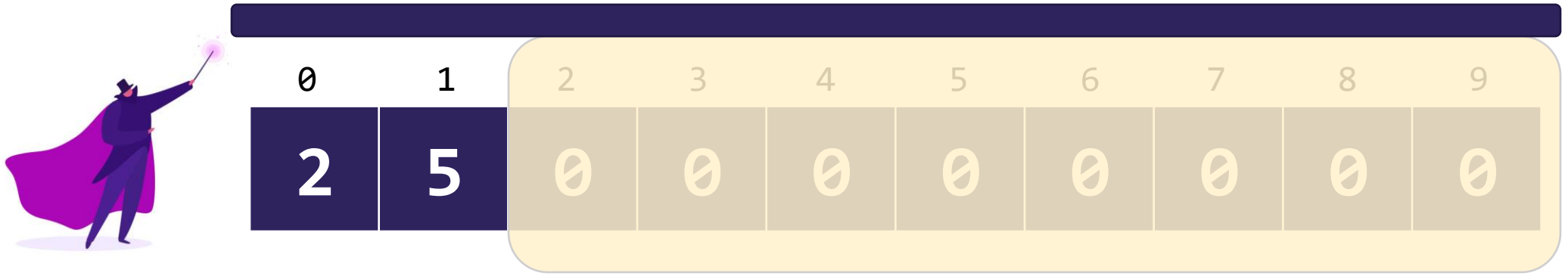
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

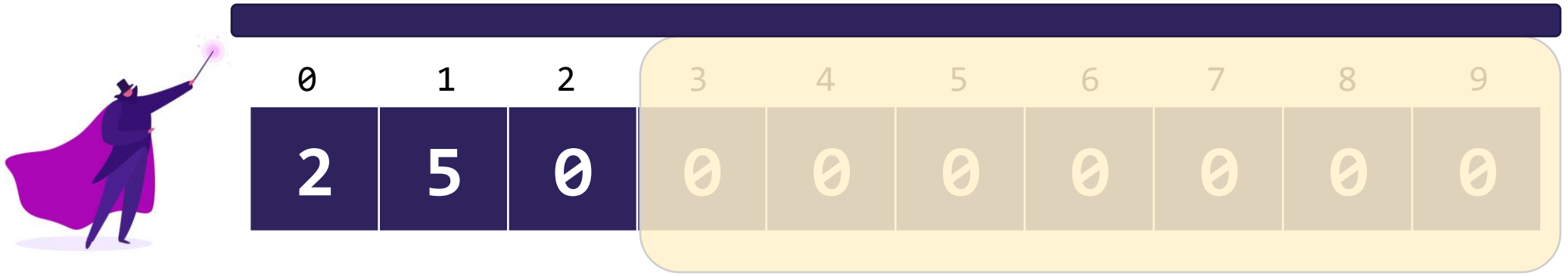
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

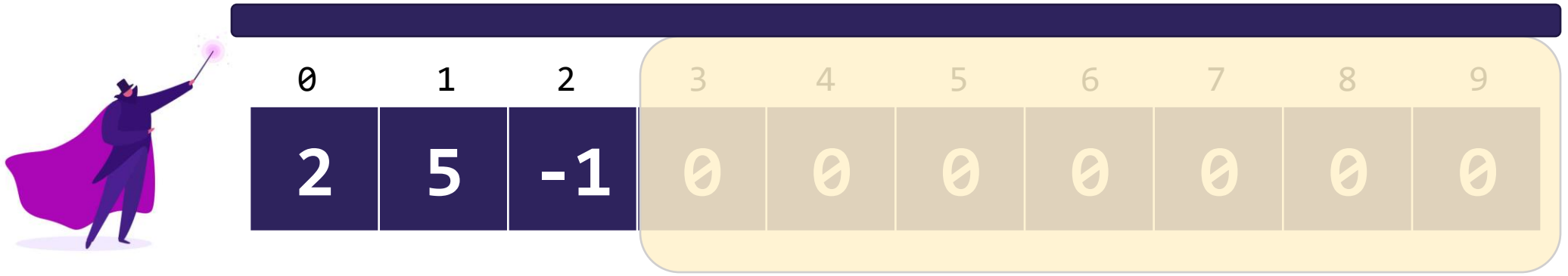
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




```
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`

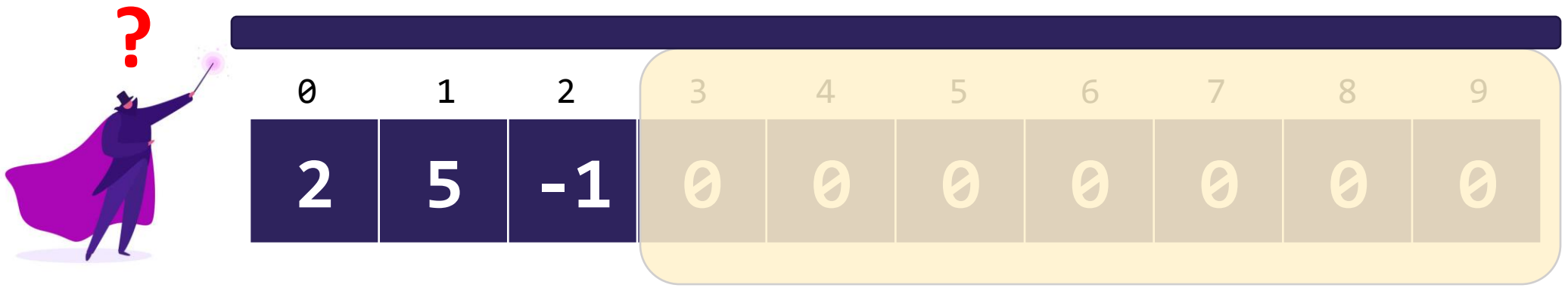



```
al.add(-1);
```

Implementing add()

ArrayIntLists

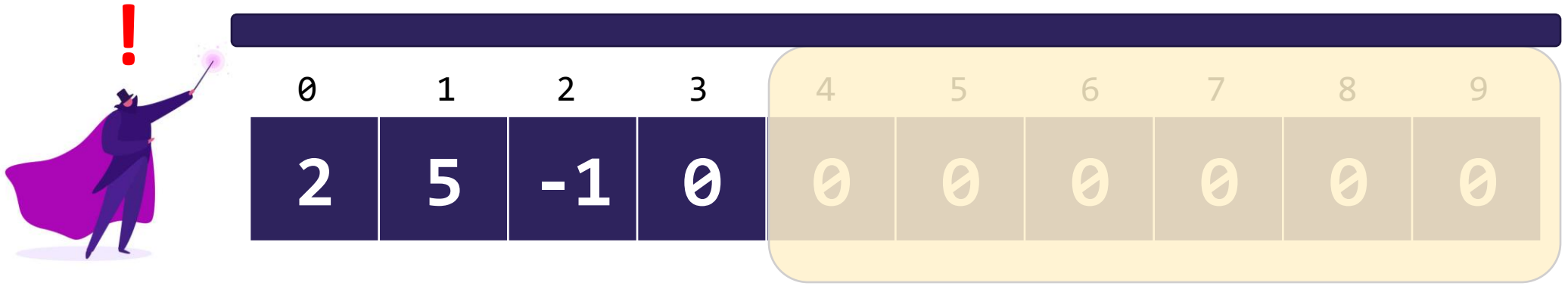
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

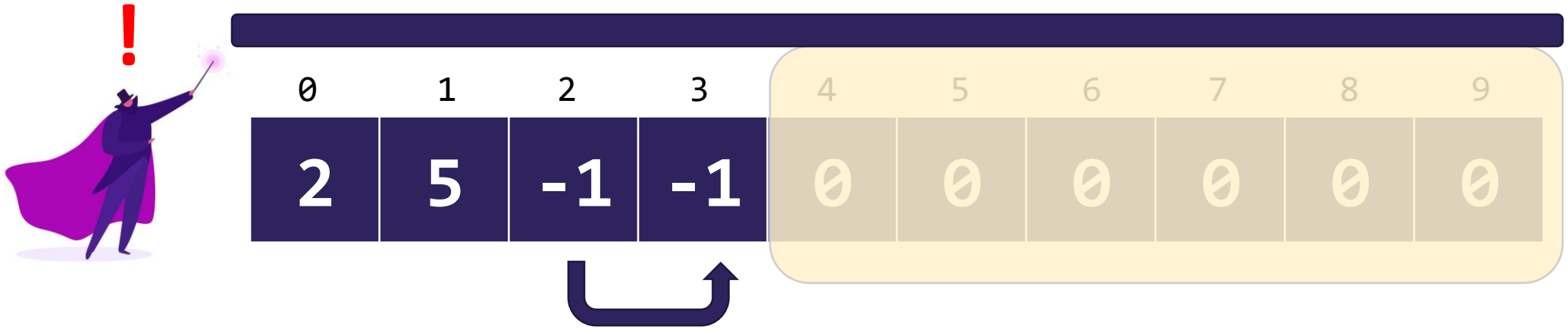
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

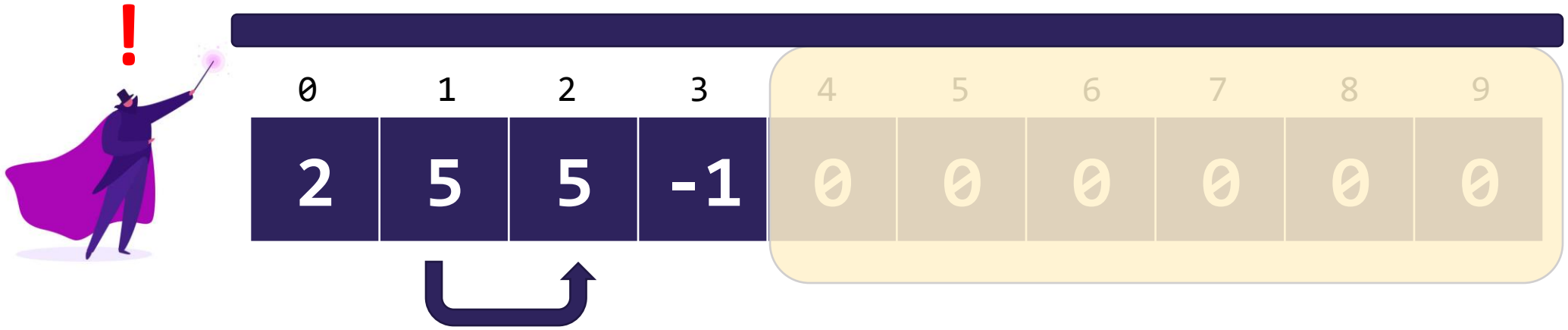
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

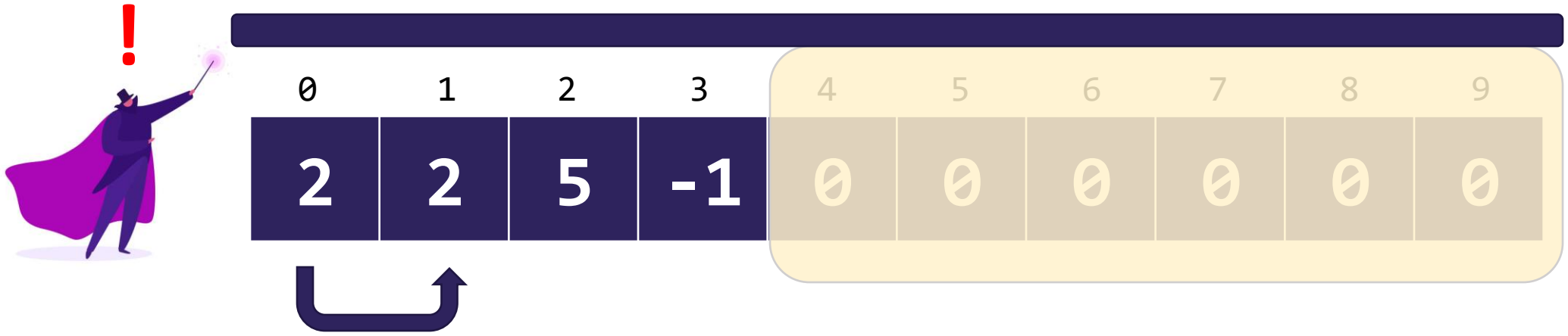
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




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

ArrayIntLists

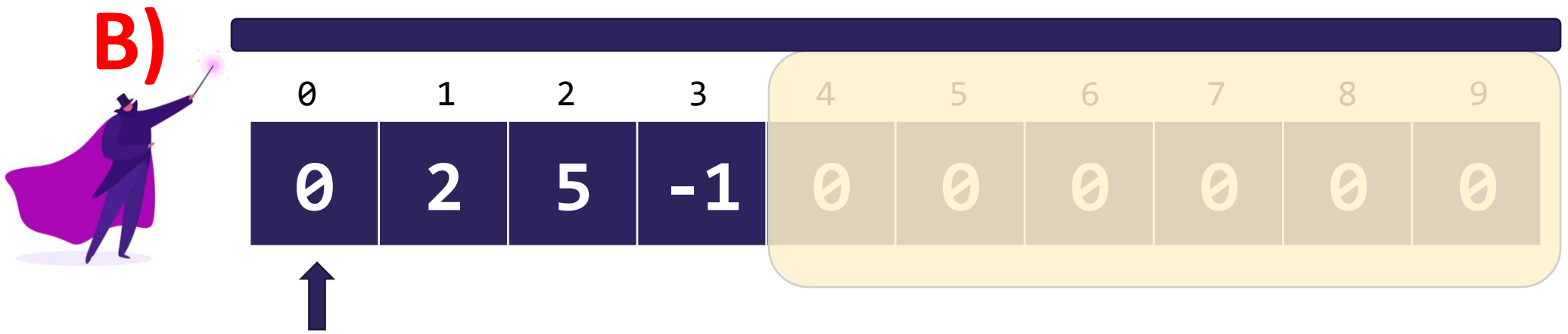
- For simplicity: only about storing ints (no type variables)
- How do we accomplish resizing magic trick? Two fields:
  - `int[] elementData;  // Where we store elements`
  - `int size;  // Storage boundary`




```
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);
```

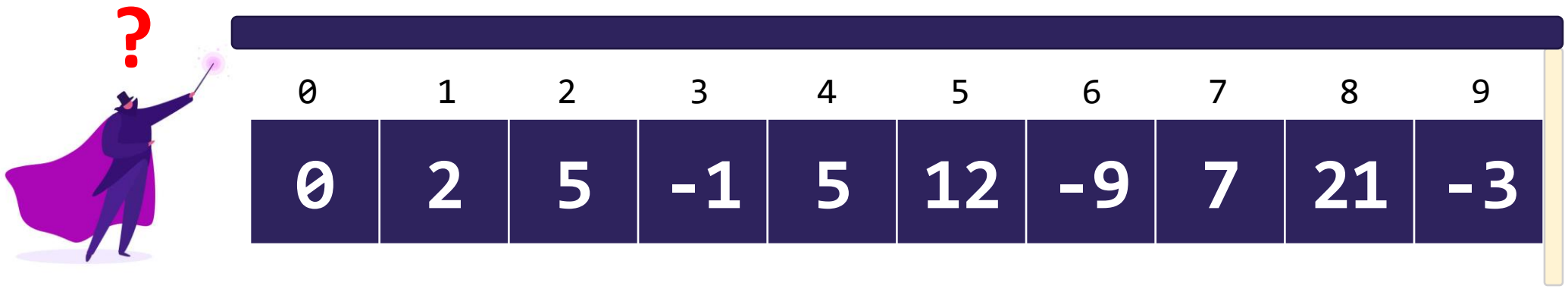
ArrayIntLists

- 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

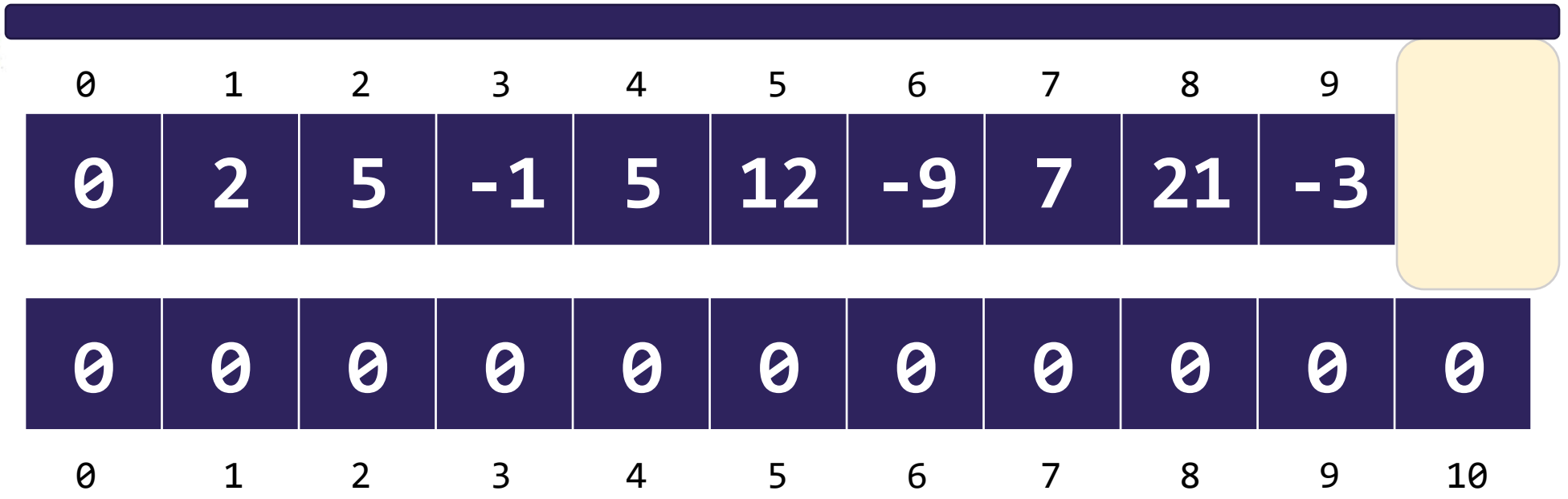
- `Capacity` = length of underlying array
- `Size` = number of user-added elements
- What happens if we run out of space? (`size == capacity`)




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

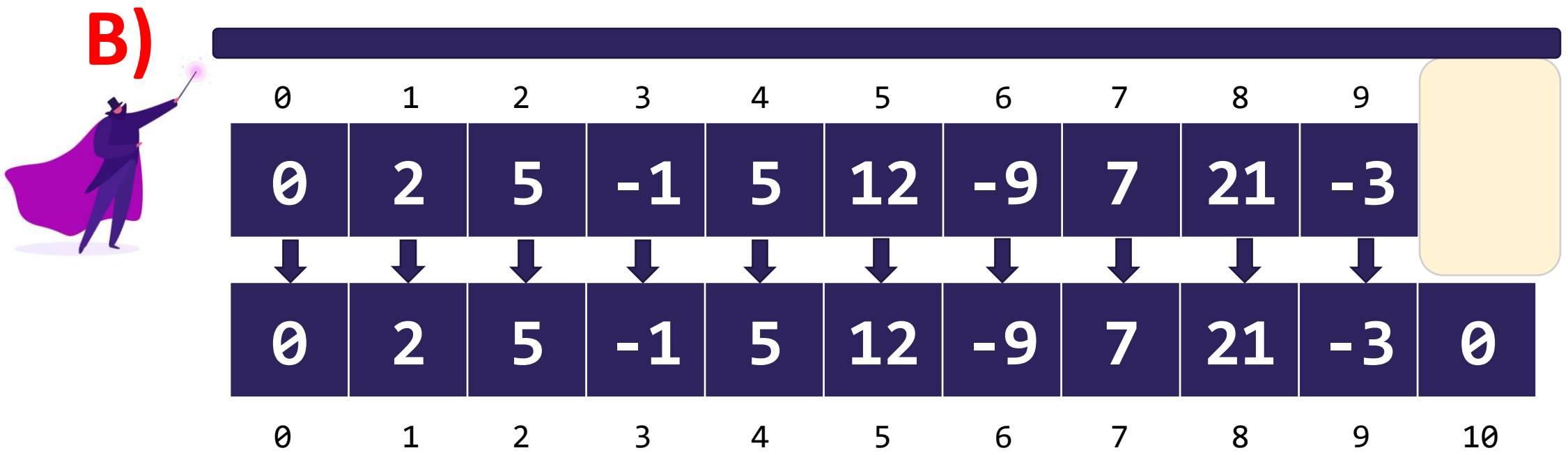
Capacity and Resizing

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



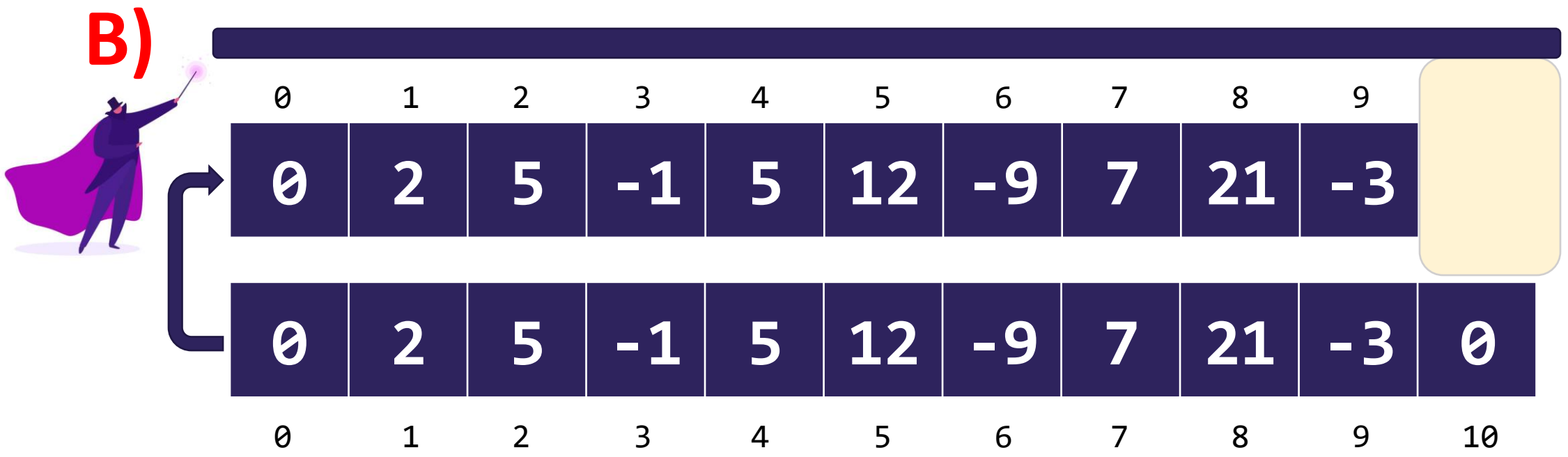
Capacity and Resizing

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



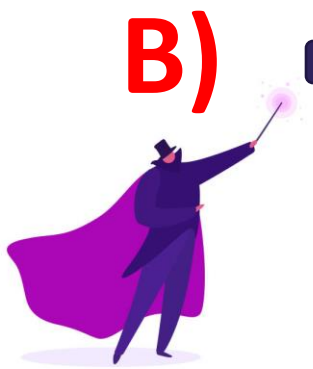
Capacity and Resizing

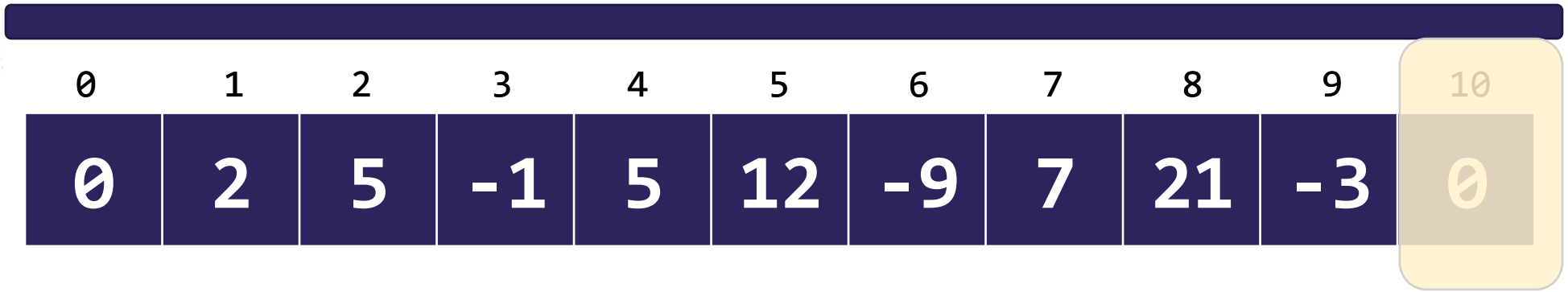
- `Capacity` = length of underlying array
- `Size` = number of user-added elements
- What happens if we run out of space? (`size == capacity`)



Capacity and Resizing

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



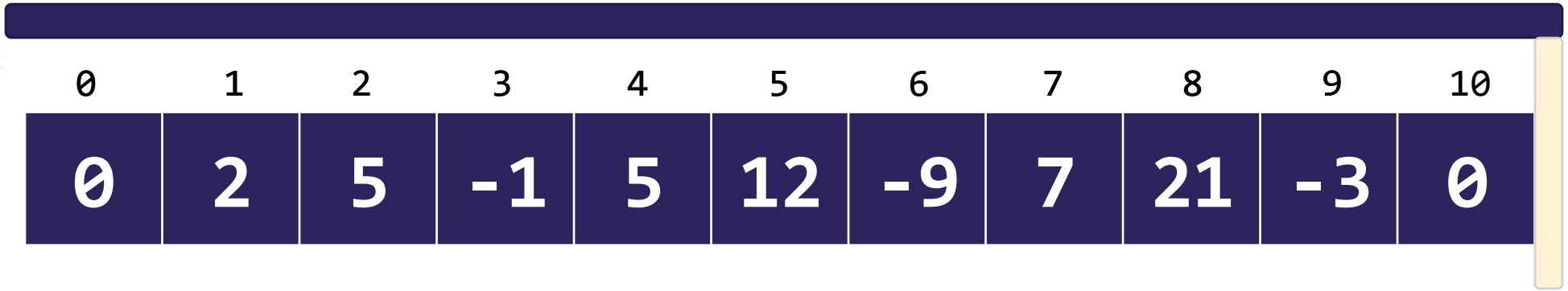



```
al.add(2);
```

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)

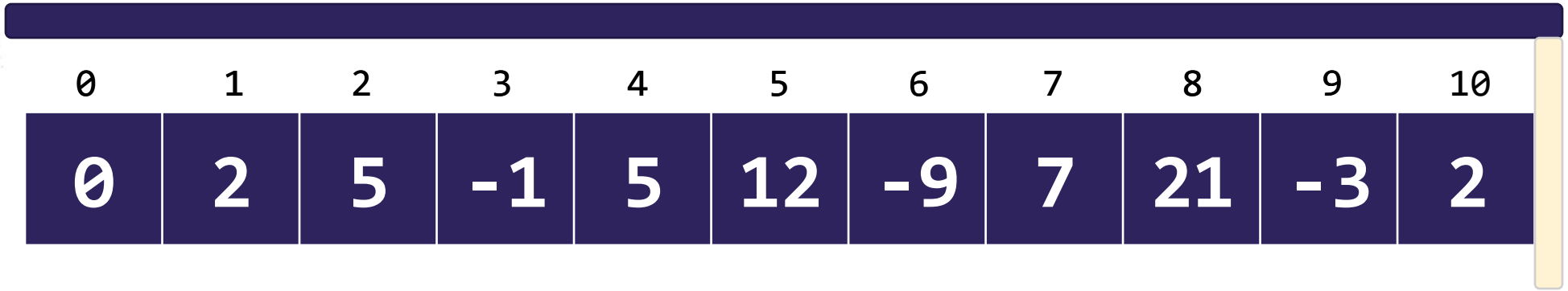



```
al.add(2);
```

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?