

Implementing Abstractions

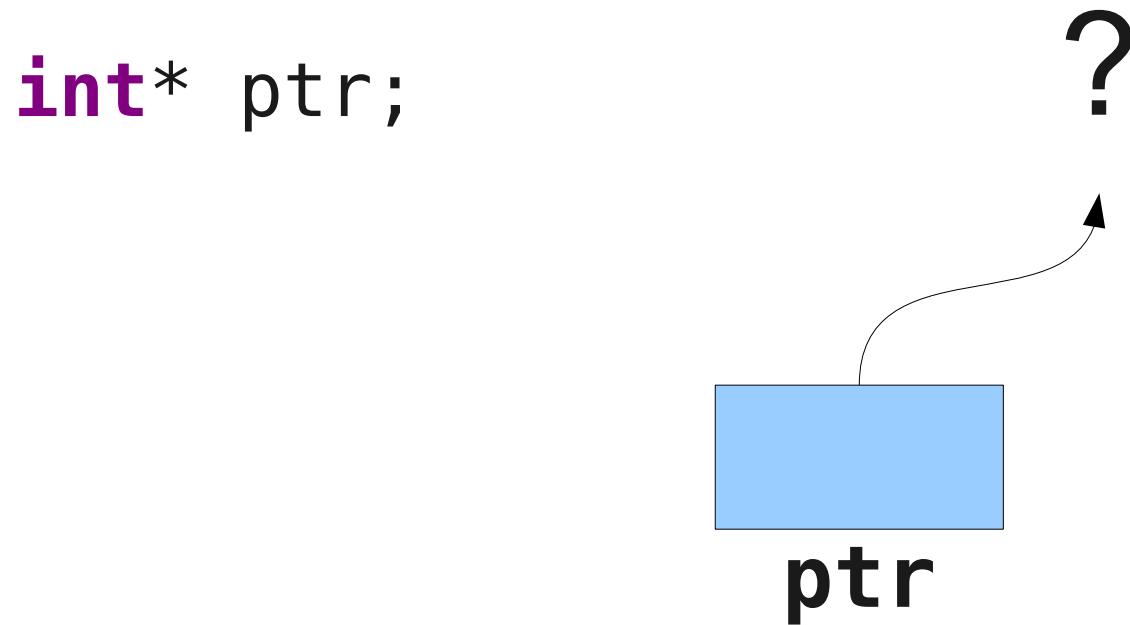
Part Two

Friday Four Square!
4:15PM, Outside Gates

Dynamic Memory Allocation

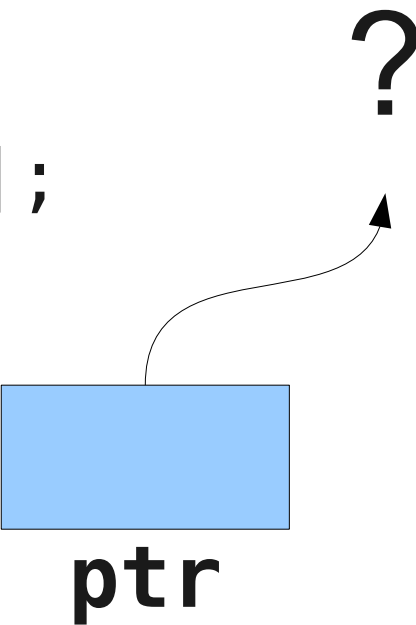
```
int* ptr;
```

Dynamic Memory Allocation



Dynamic Memory Allocation

```
int* ptr;  
ptr = new int[5];
```

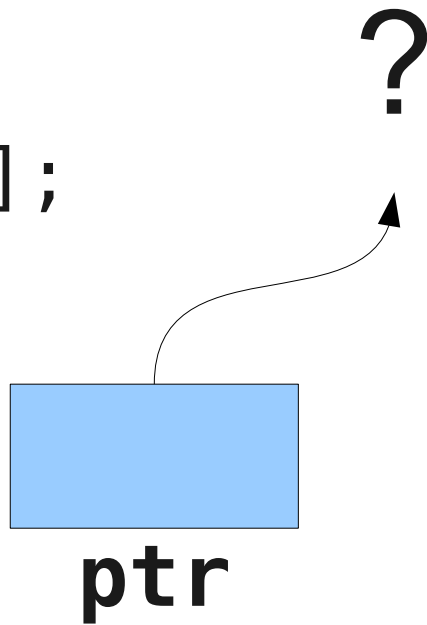


The diagram illustrates the state of memory after the provided C++ code. A light blue rectangular box represents the memory allocated for the array. A curved arrow originates from the top center of this box and points to a large question mark, indicating that the specific memory address is unknown or being queried.

ptr

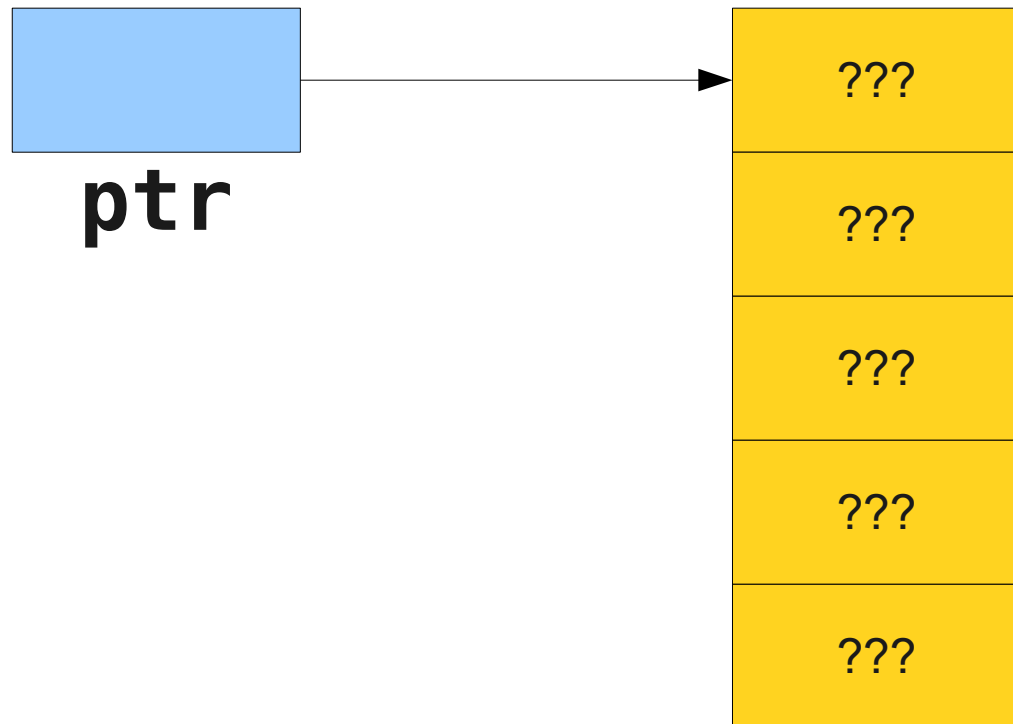
Dynamic Memory Allocation

```
int* ptr;  
ptr = new int[5];
```



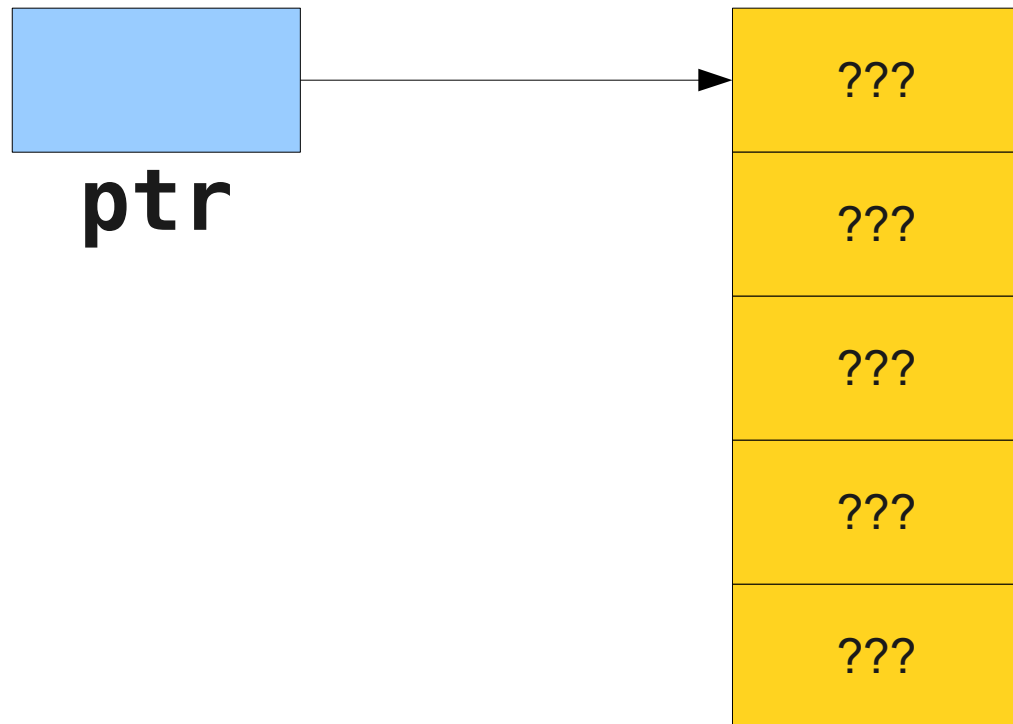
Dynamic Memory Allocation

```
int* ptr;  
ptr = new int[5];
```



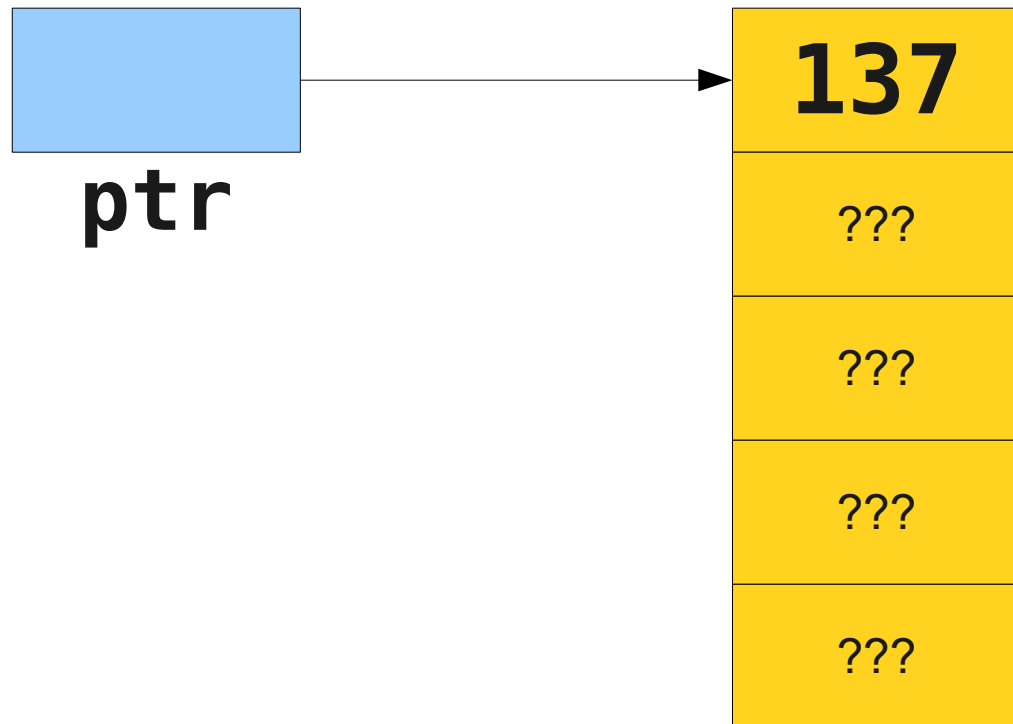
Dynamic Memory Allocation

```
int* ptr;  
ptr = new int[5];  
ptr[0] = 137;
```



Dynamic Memory Allocation

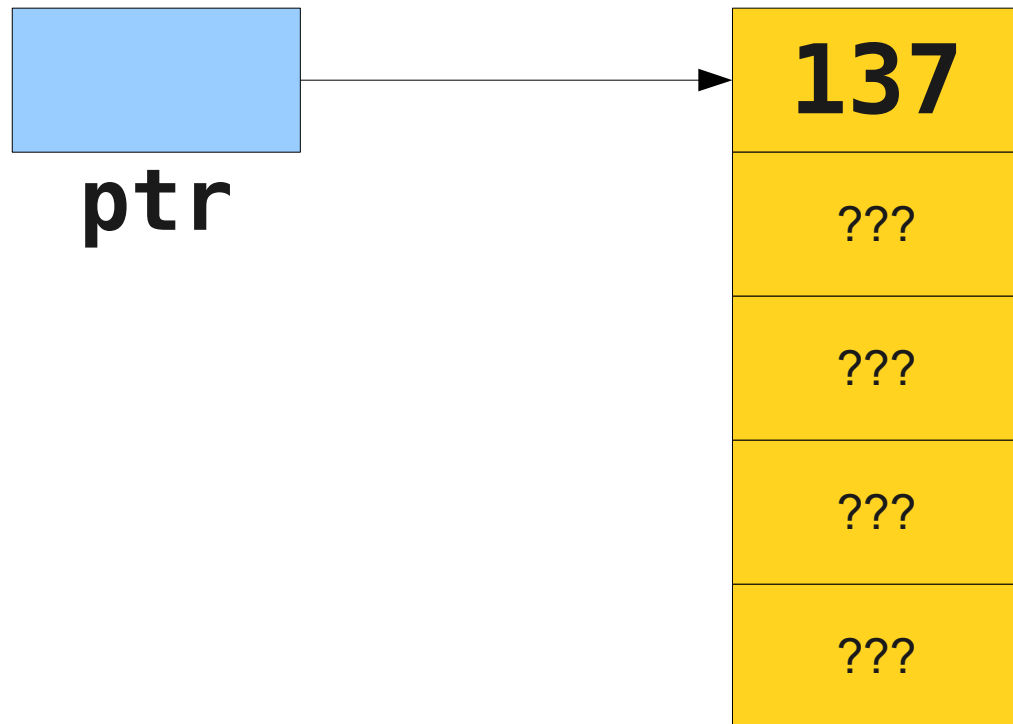
```
int* ptr;  
ptr = new int[5];  
ptr[0] = 137;
```



Dynamic Memory Allocation

```
int* ptr;  
ptr = new int[5];
```

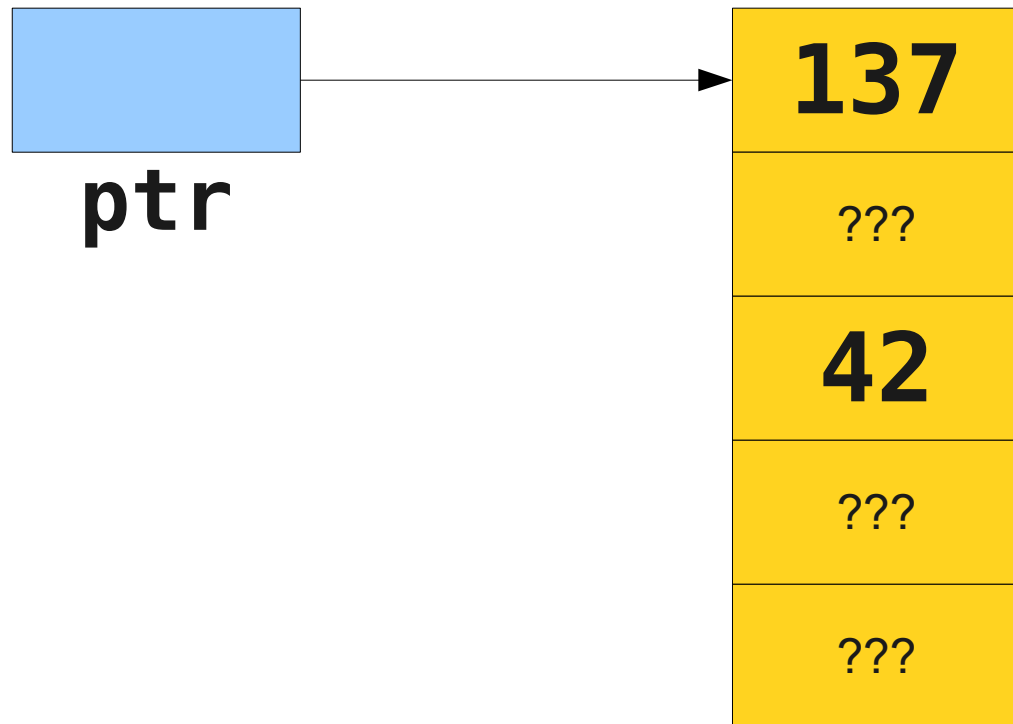
```
ptr[0] = 137;  
ptr[2] = 42;
```



Dynamic Memory Allocation

```
int* ptr;  
ptr = new int[5];
```

```
ptr[0] = 137;  
ptr[2] = 42;
```



Cleaning Up

- Unlike other languages like Java, in C++, you are responsible for deallocating any memory allocated with **new[]**.
- You can deallocate memory with the **delete[]** operator:

delete[] *ptr*;

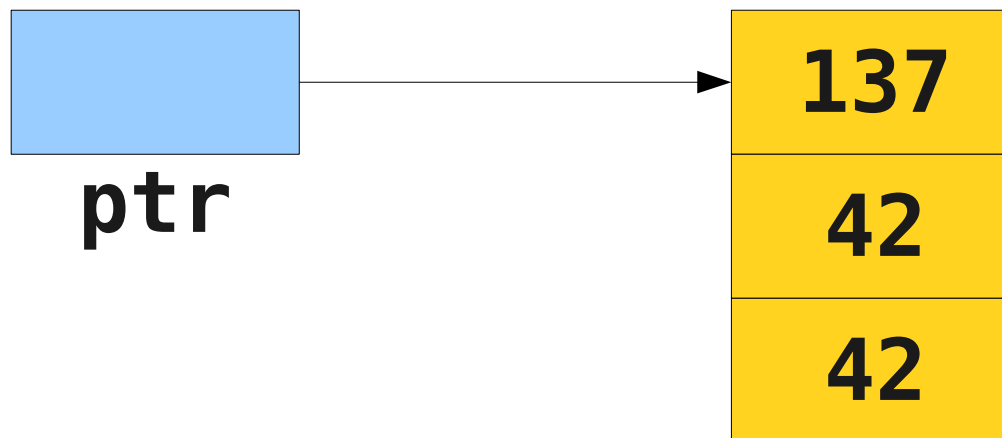
- This destroys the array pointed at by the given pointer, not the pointer itself.

Cleaning Up

- Unlike other languages like Java, in C++, you are responsible for deallocating any memory allocated with **new[]**.
- You can deallocate memory with the **delete[]** operator:

delete[] *ptr*;

- This destroys the array pointed at by the given pointer, not the pointer itself.

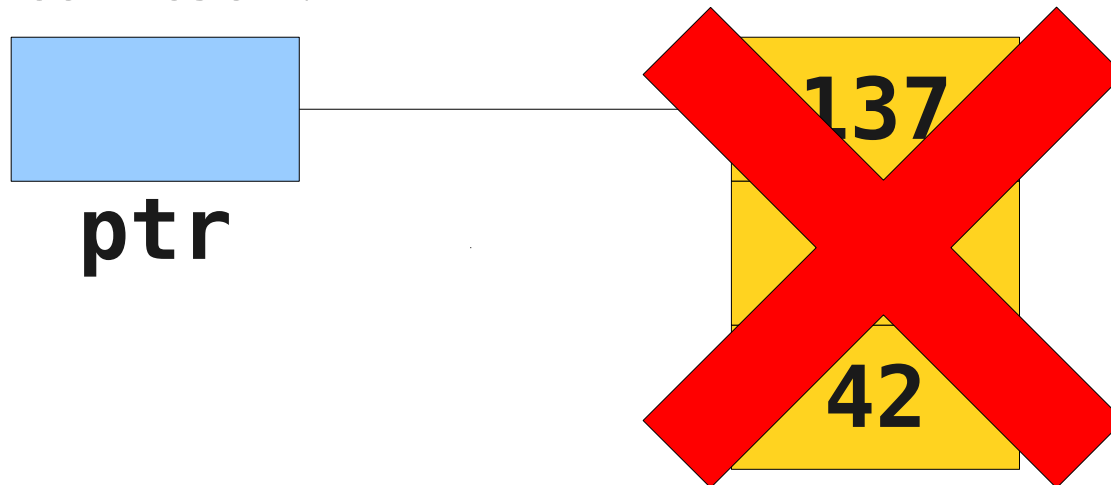


Cleaning Up

- Unlike other languages like Java, in C++, you are responsible for deallocating any memory allocated with **new[]**.
- You can deallocate memory with the **delete[]** operator:

delete[] *ptr*;

- This destroys the array pointed at by the given pointer, not the pointer itself.



Cleaning Up

- Unlike other languages like Java, in C++, you are responsible for deallocating any memory allocated with **new[]**.
- You can deallocate memory with the **delete[]** operator:

delete[] *ptr*;

- This destroys the array pointed at by the given pointer, not the pointer itself.

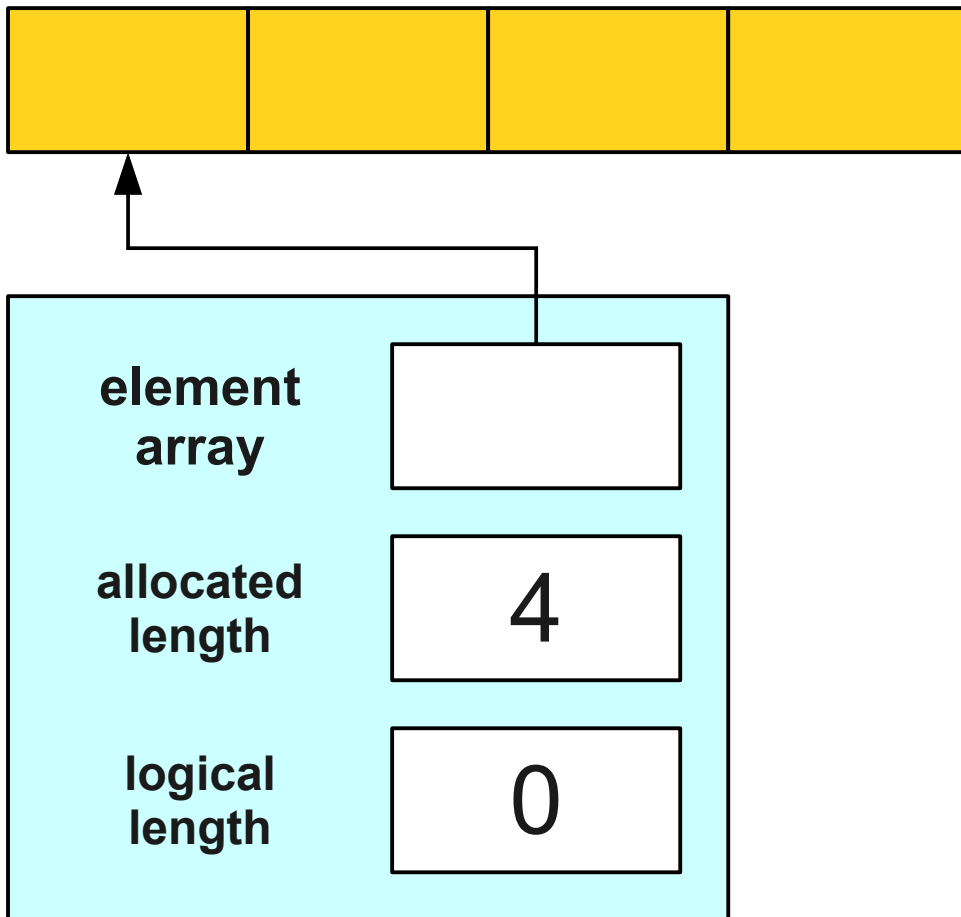


Implementing **Stack**

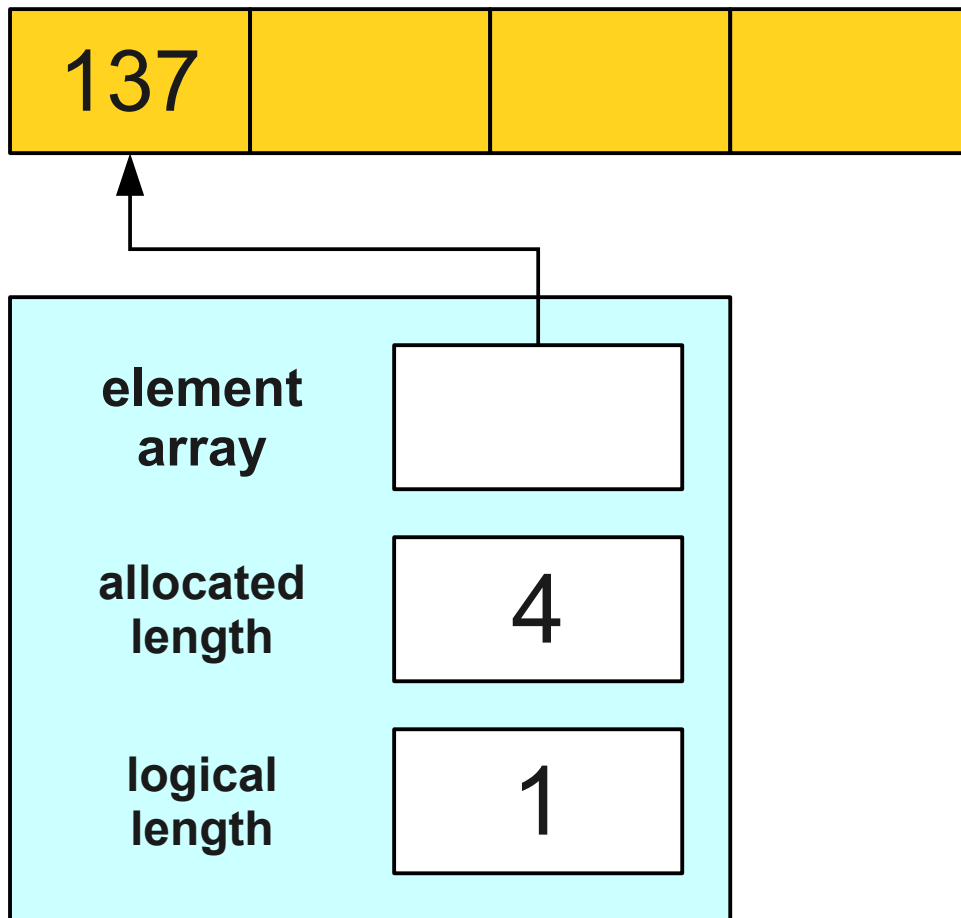
An Initial Idea

- **A bounded stack.**
- Allocate a fixed-size array for elements.
- Add elements to the array when they're pushed.
- Remove elements from the array when they're popped.
- Report an error if we exceed the size of the array.

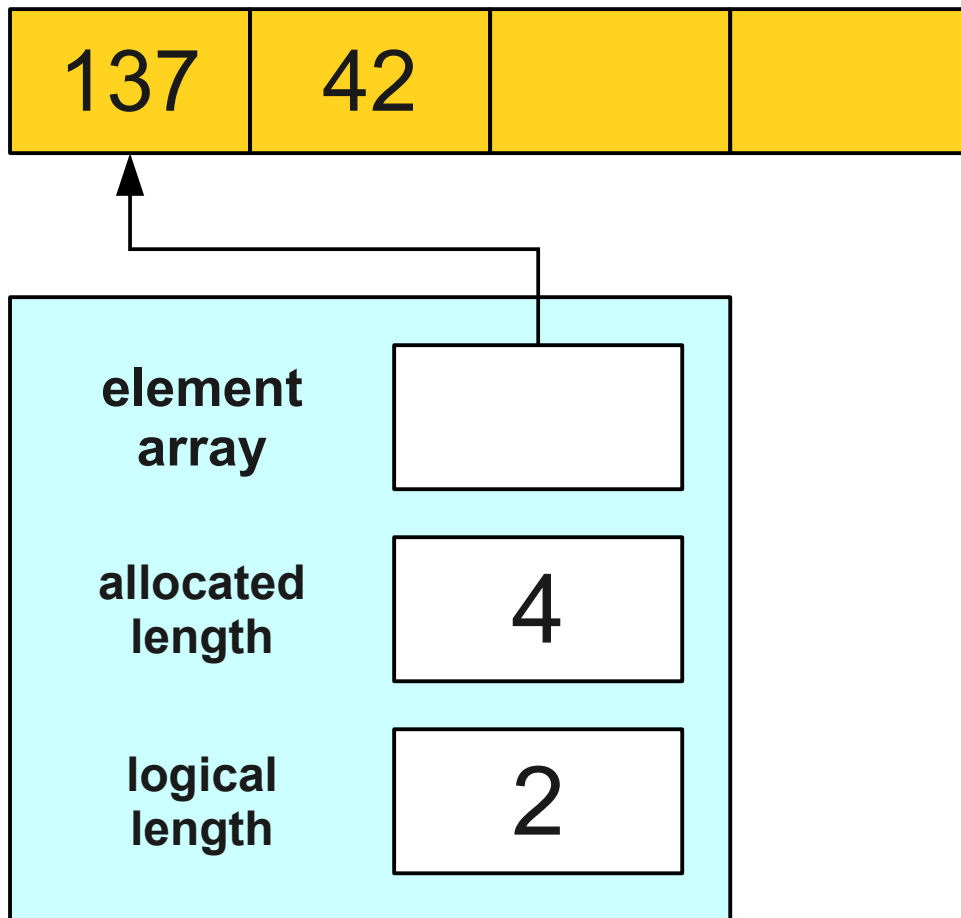
An Initial Idea



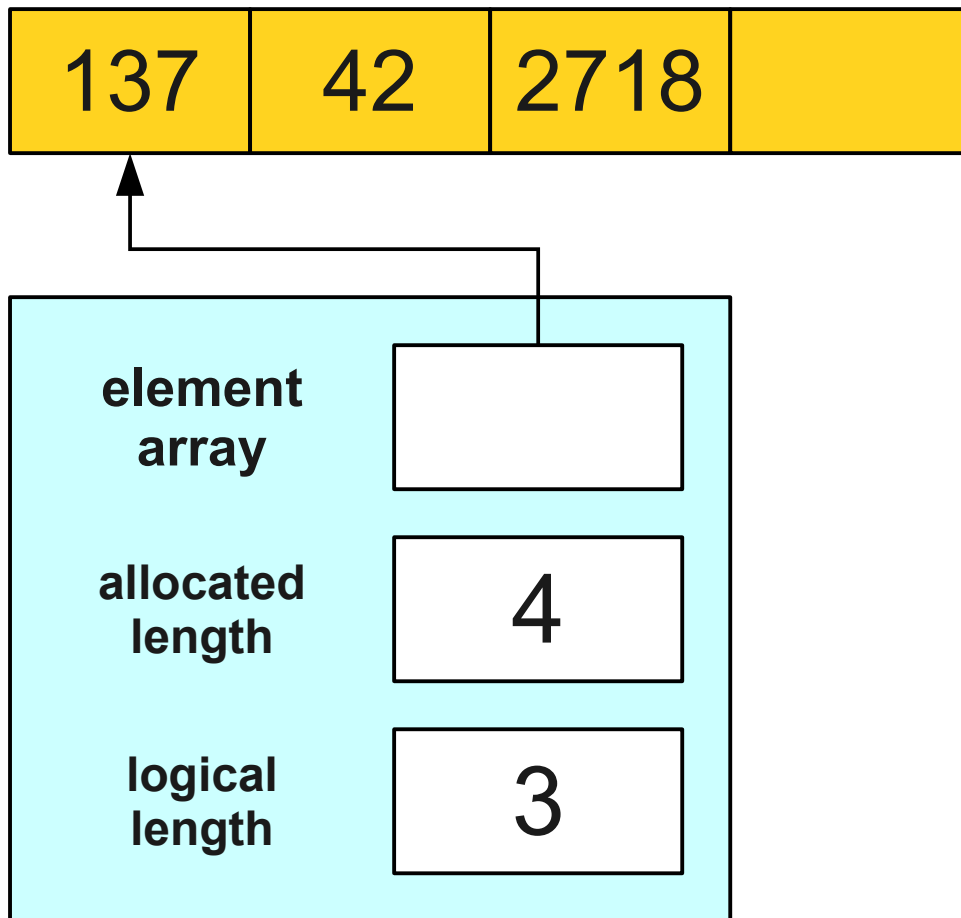
An Initial Idea



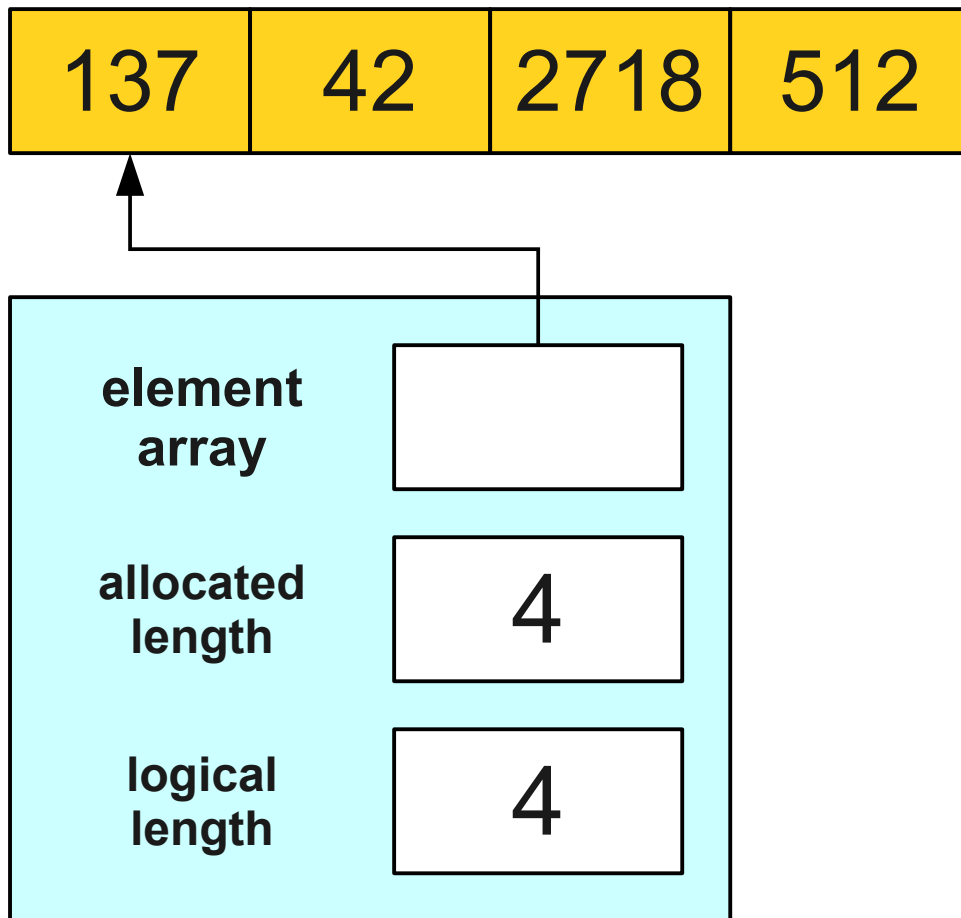
An Initial Idea



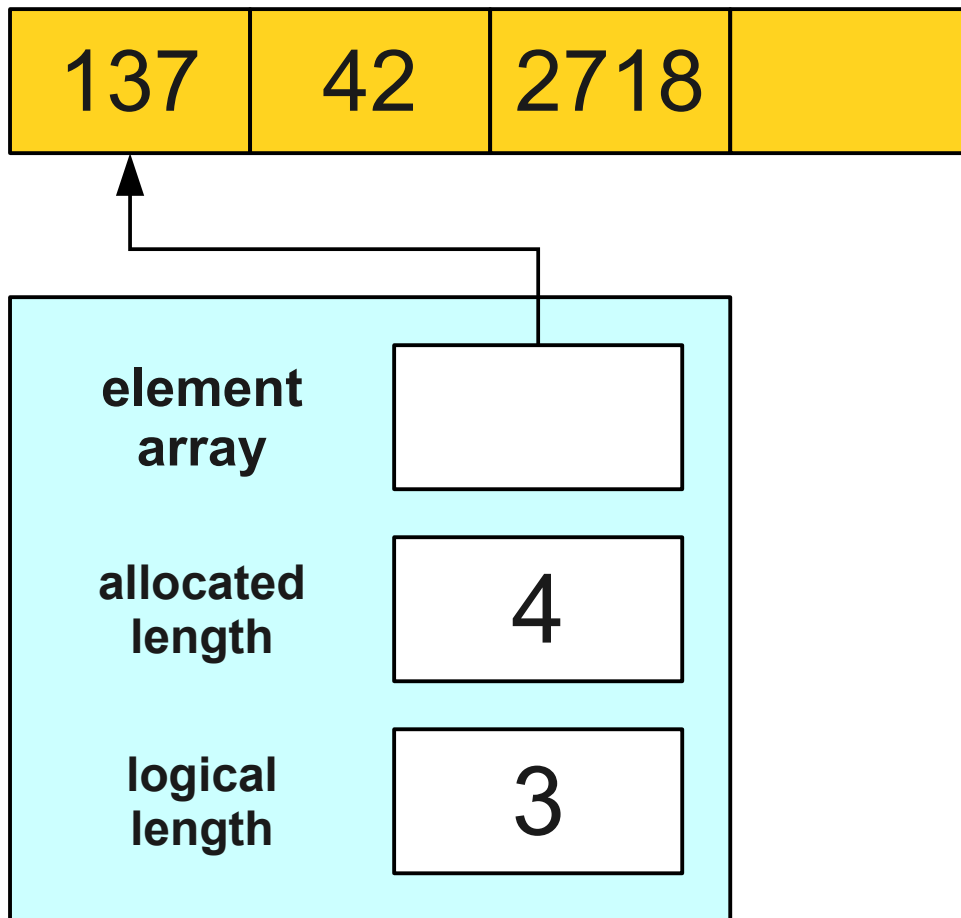
An Initial Idea



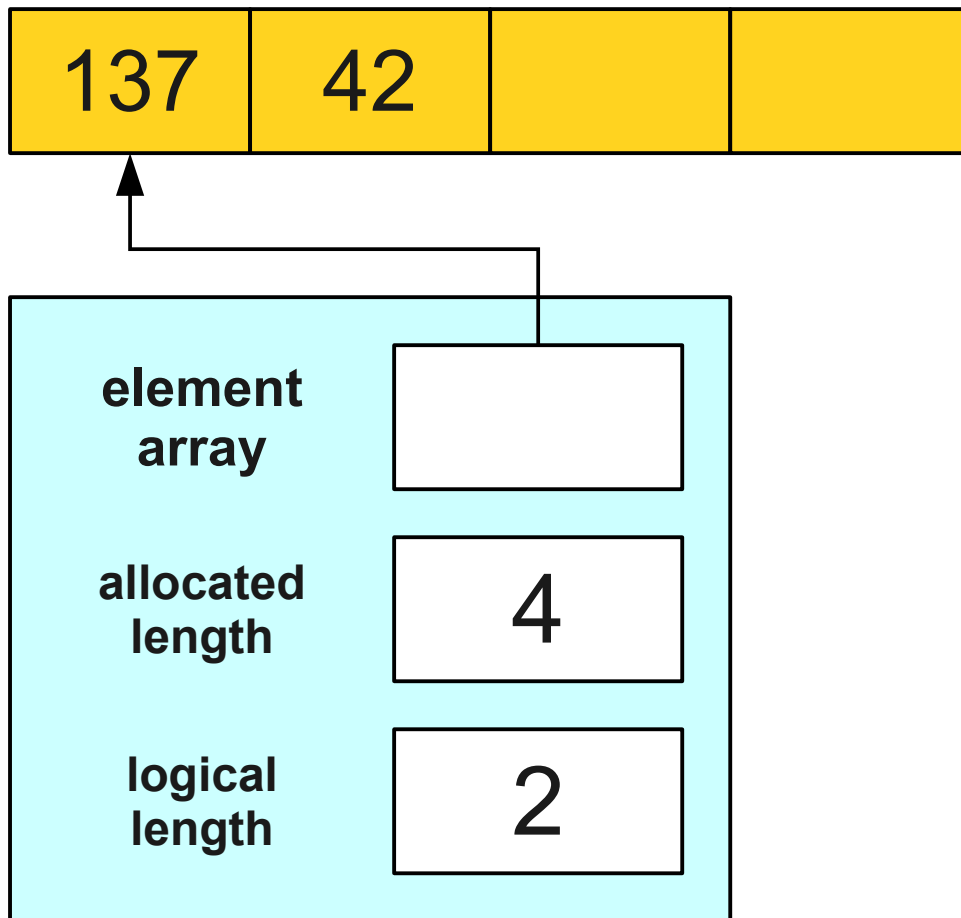
An Initial Idea



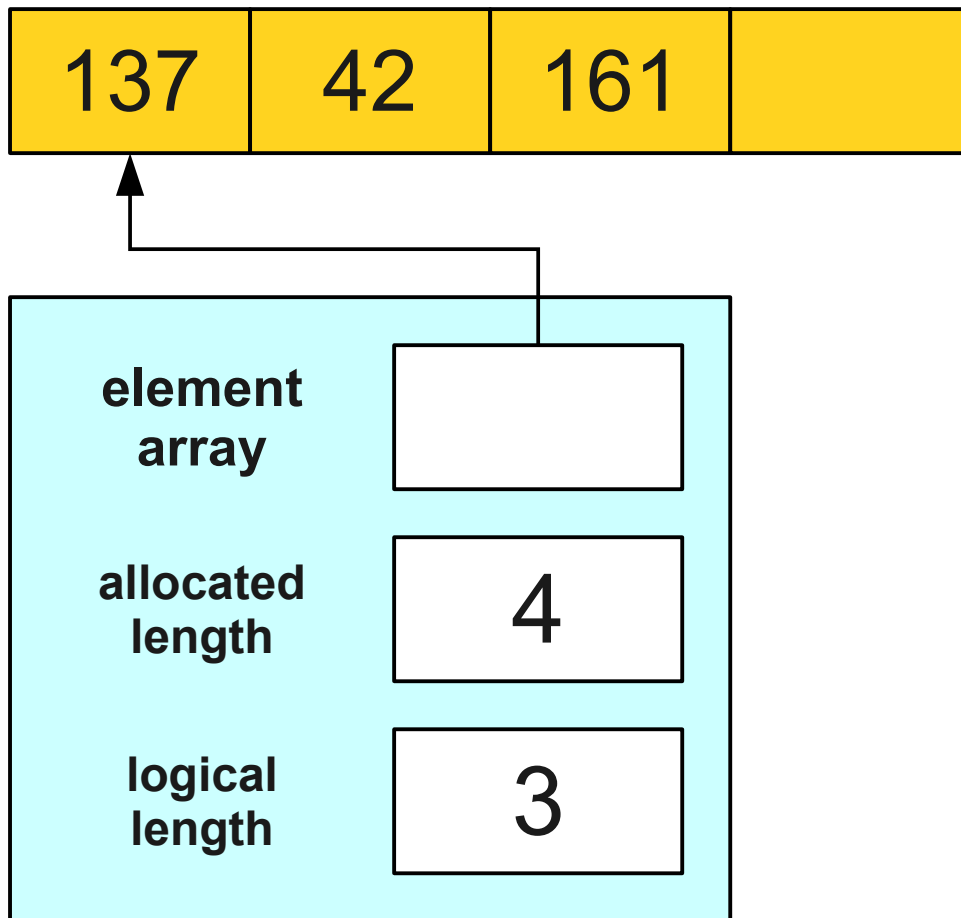
An Initial Idea



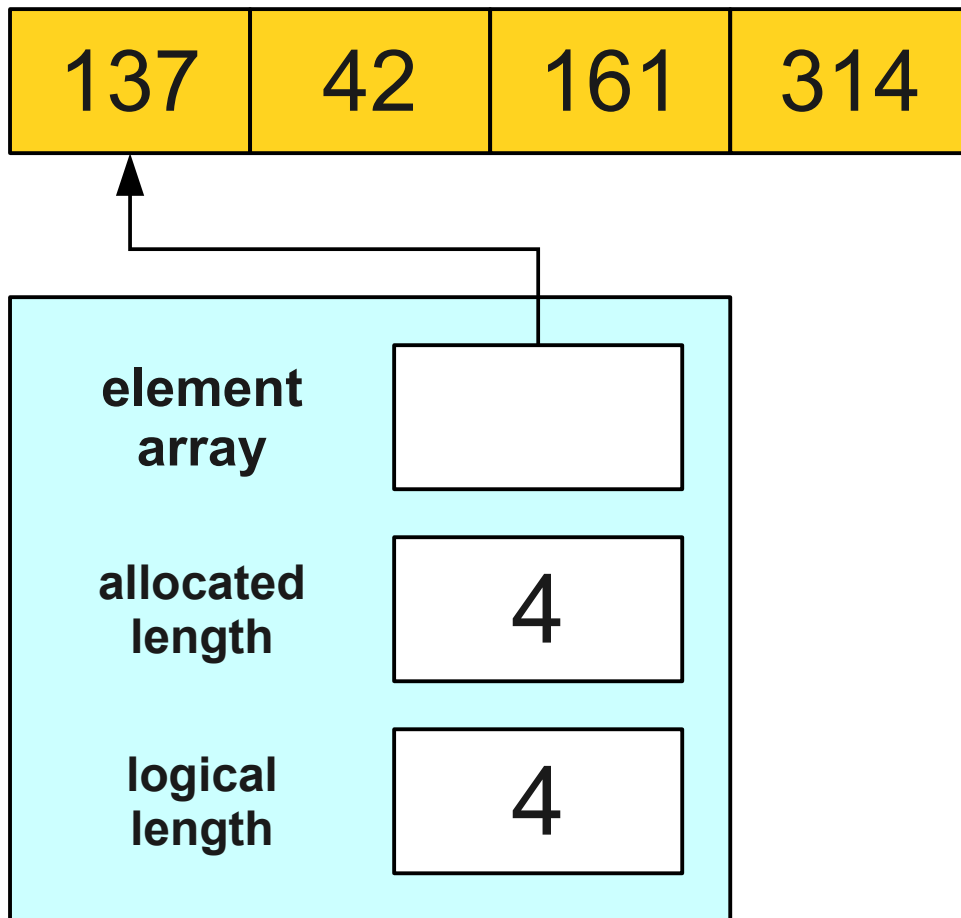
An Initial Idea



An Initial Idea



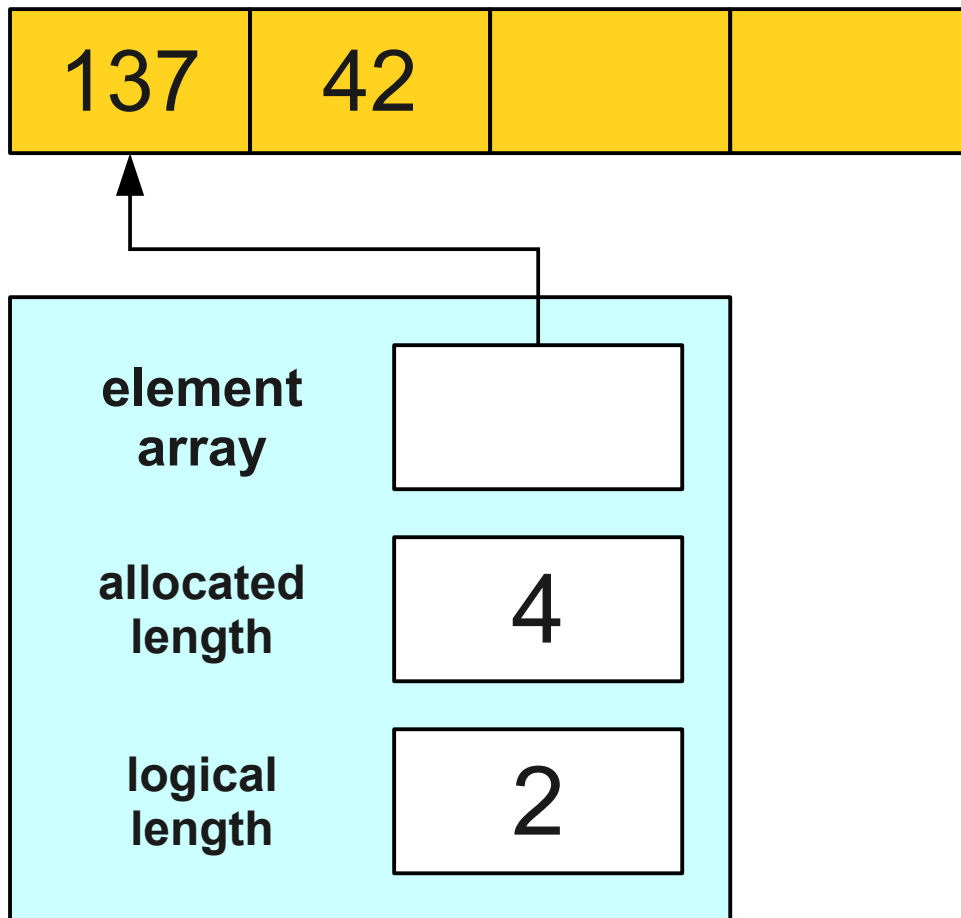
An Initial Idea



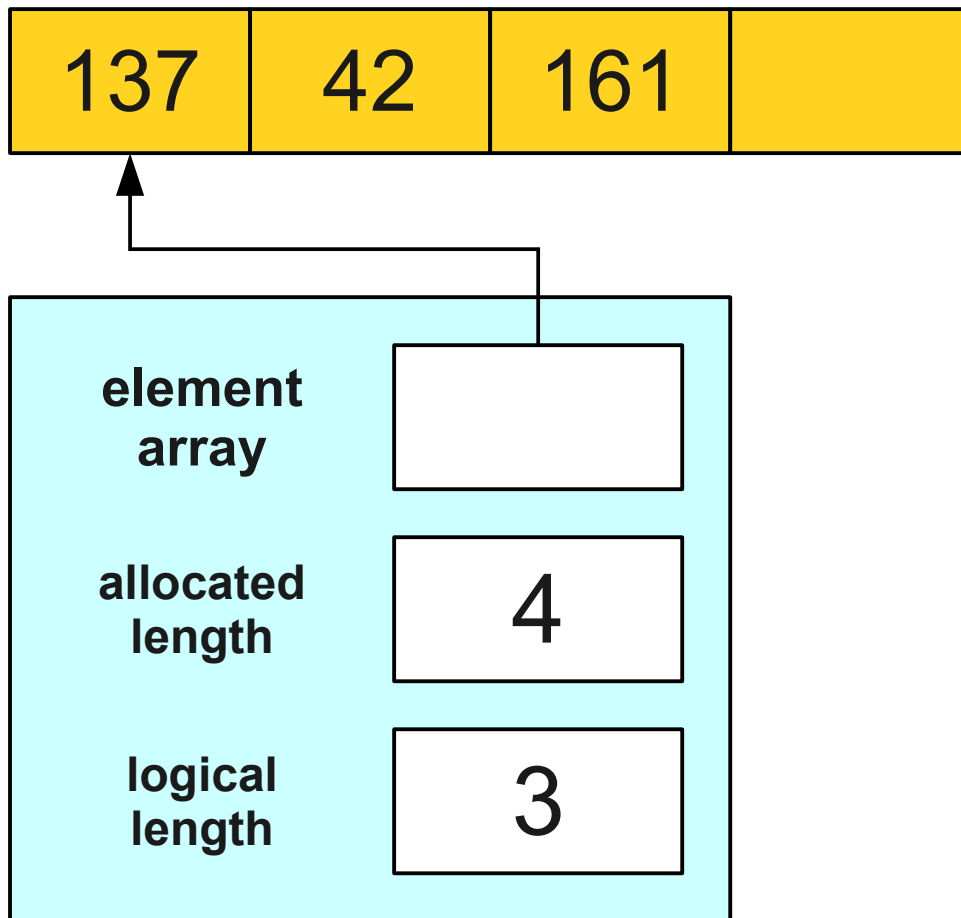
Running out of Space

- Our current implementation very quickly runs out of space to store elements.
- What should we do when this happens?

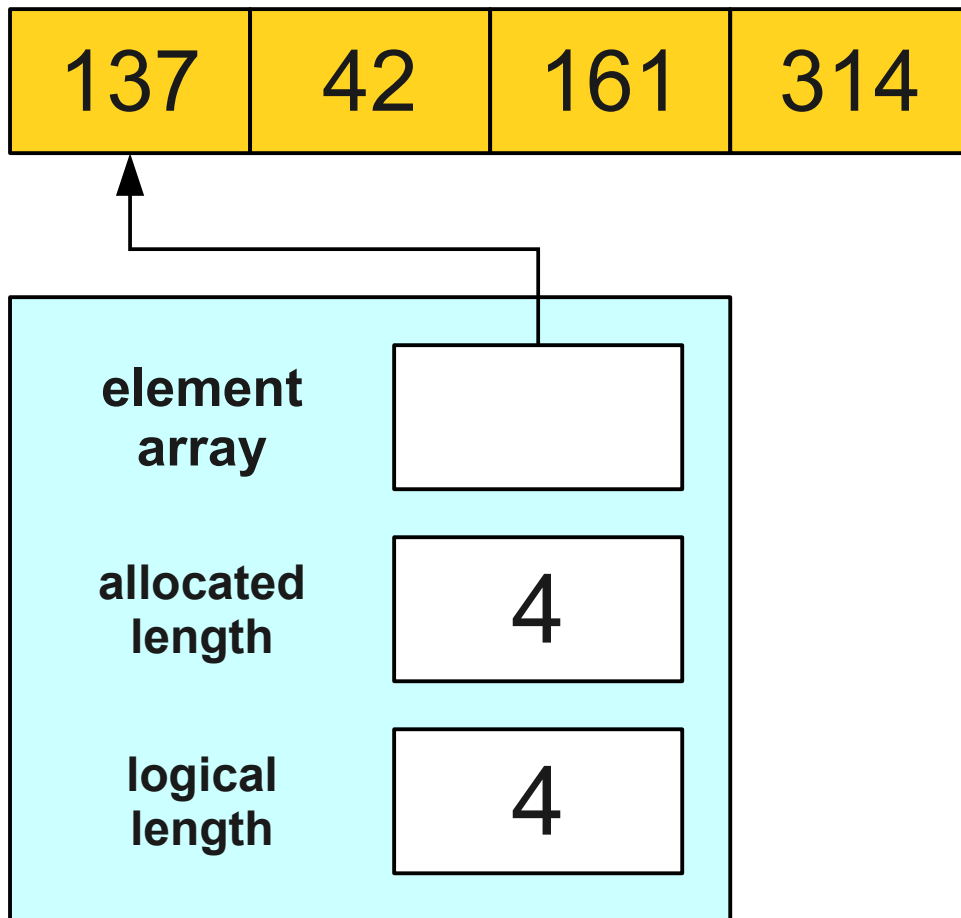
An Initial Idea



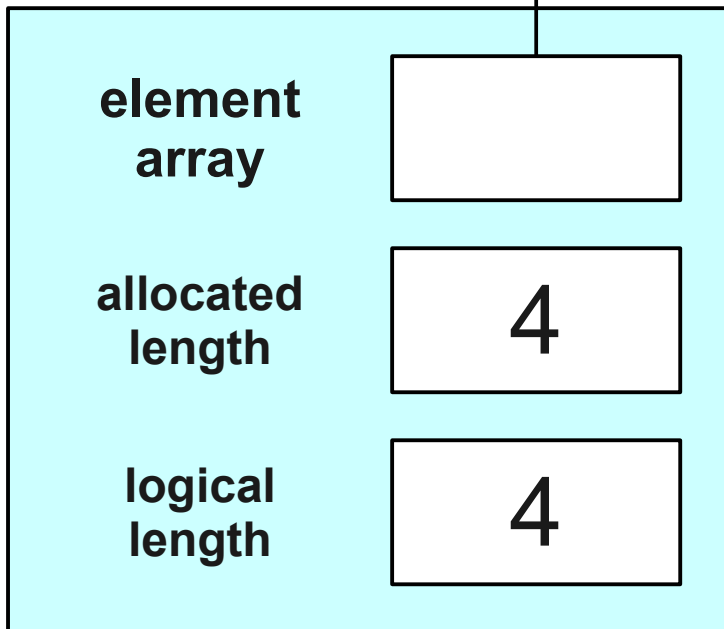
An Initial Idea



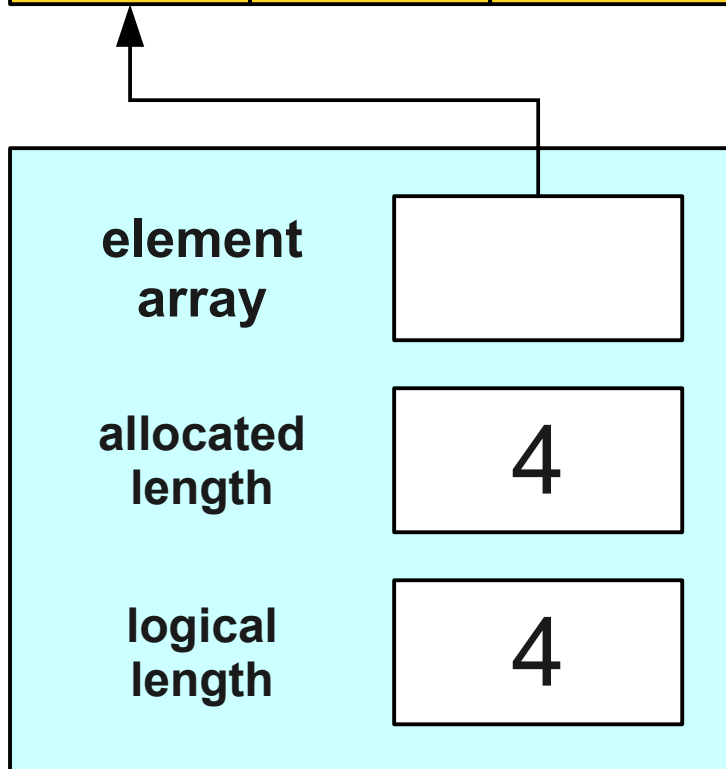
An Initial Idea



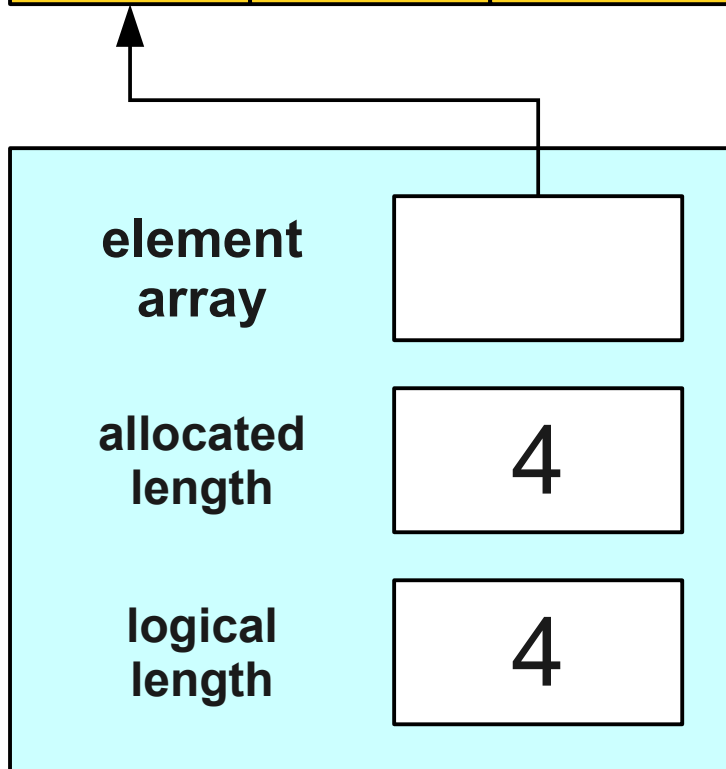
An Initial Idea



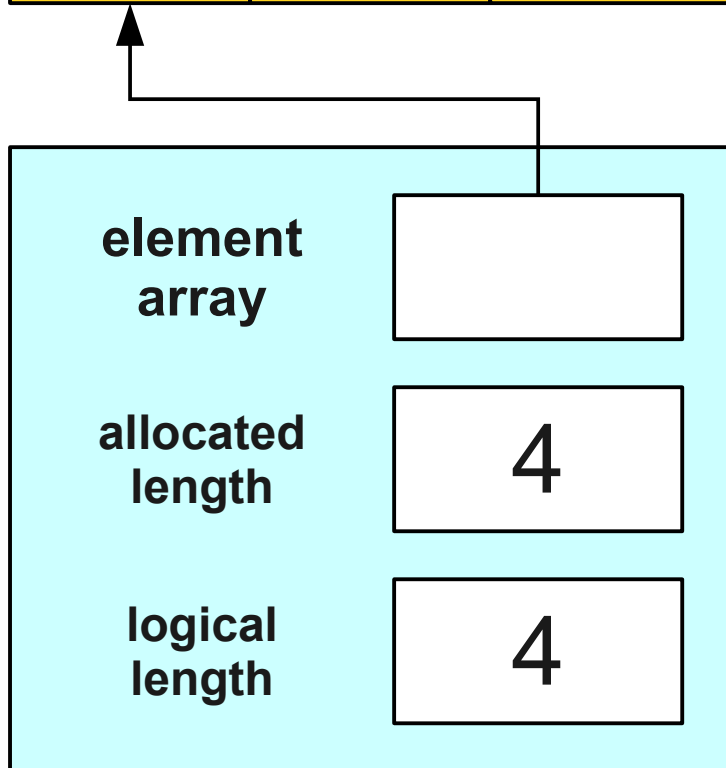
An Initial Idea



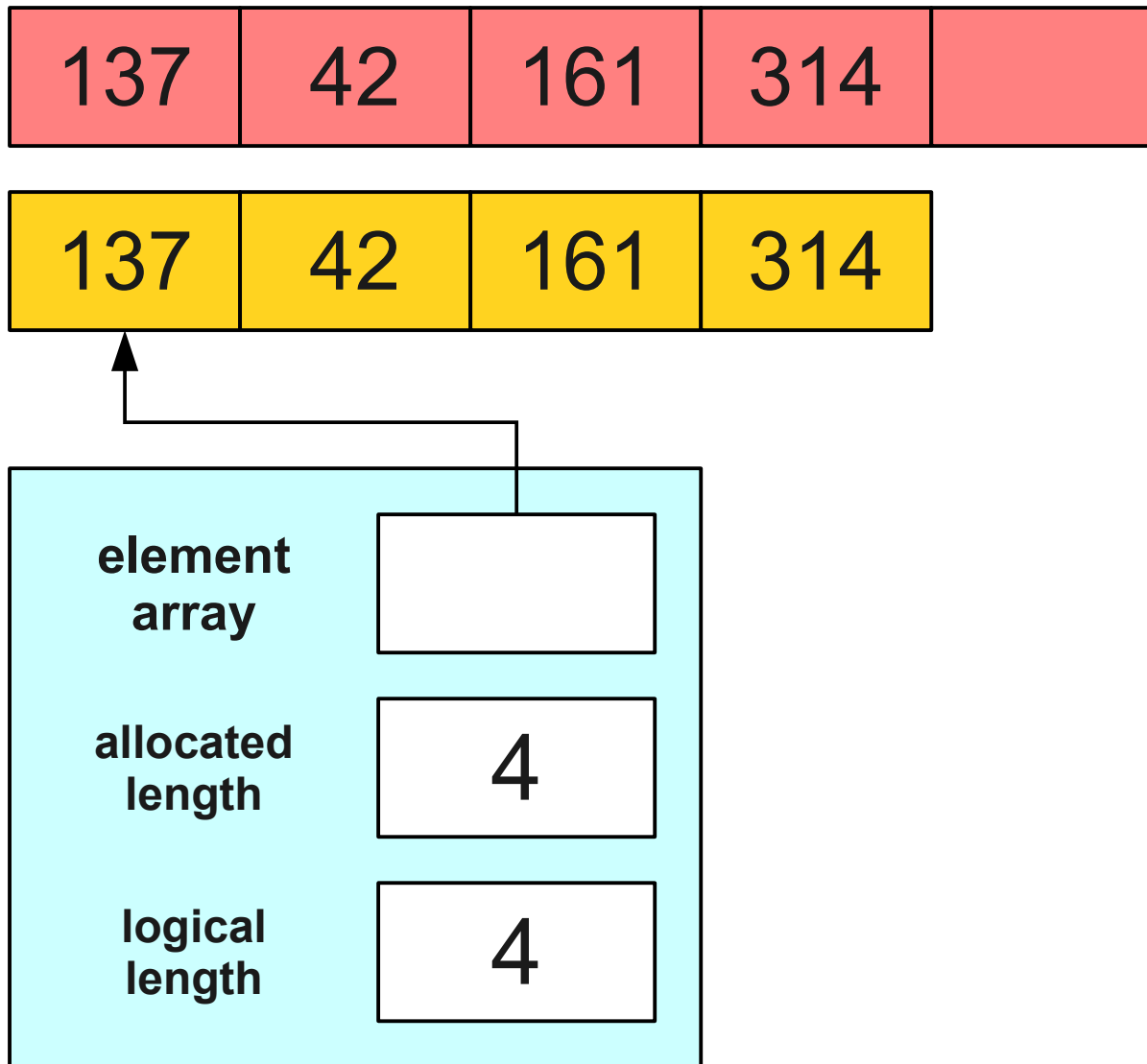
An Initial Idea



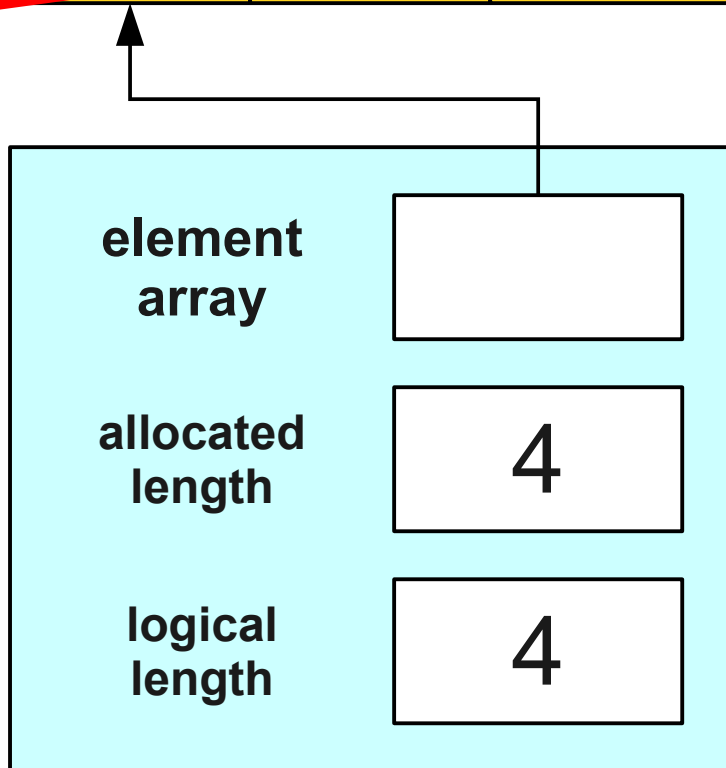
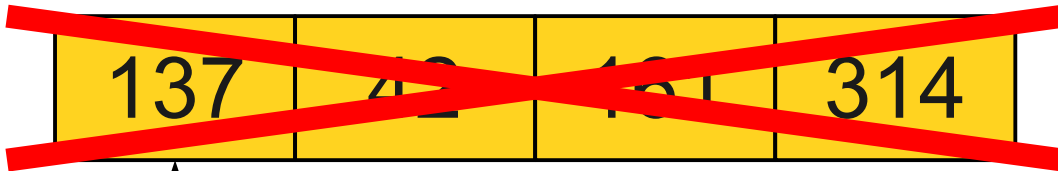
An Initial Idea



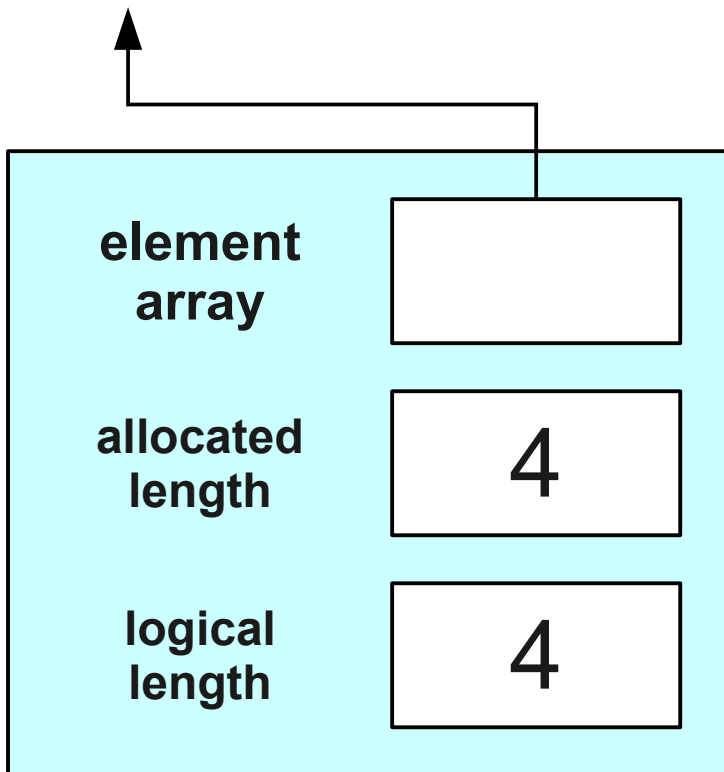
An Initial Idea



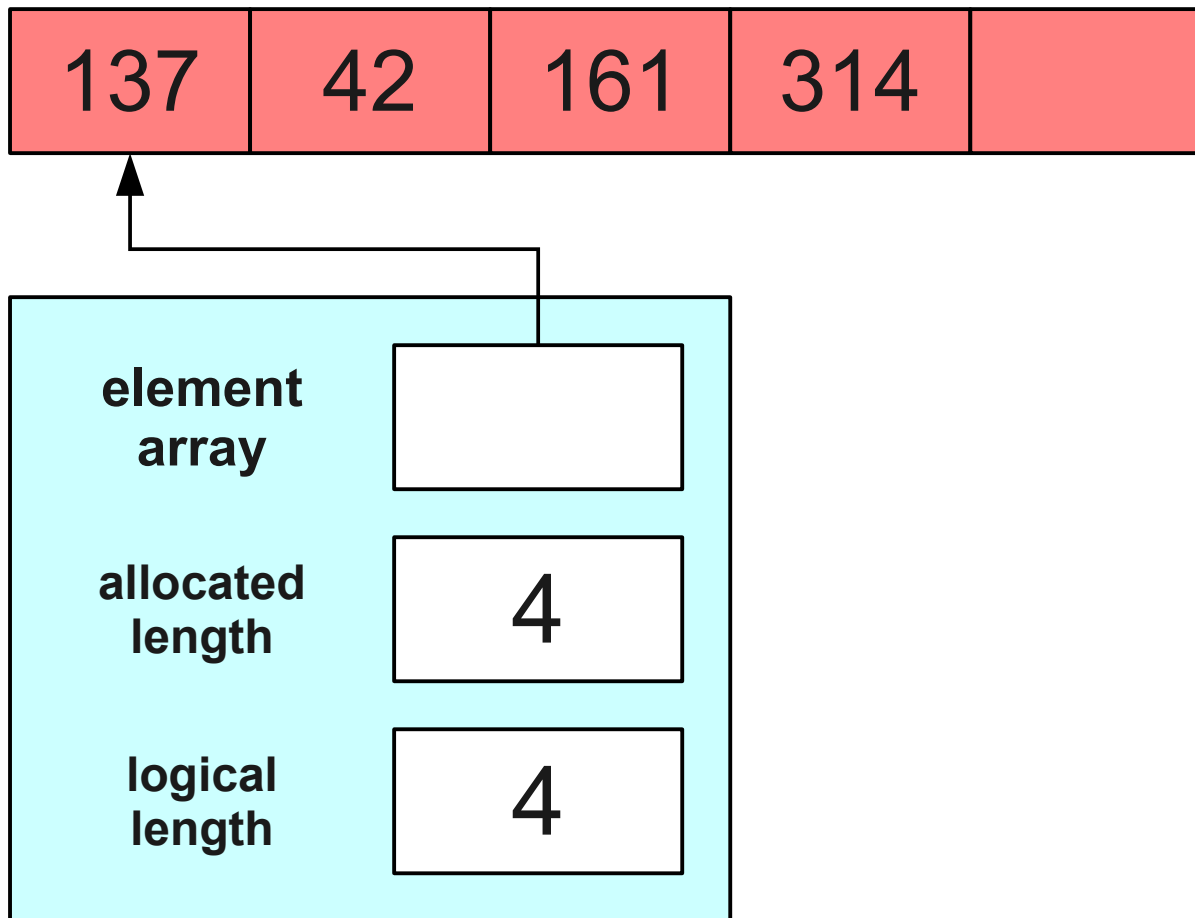
An Initial Idea



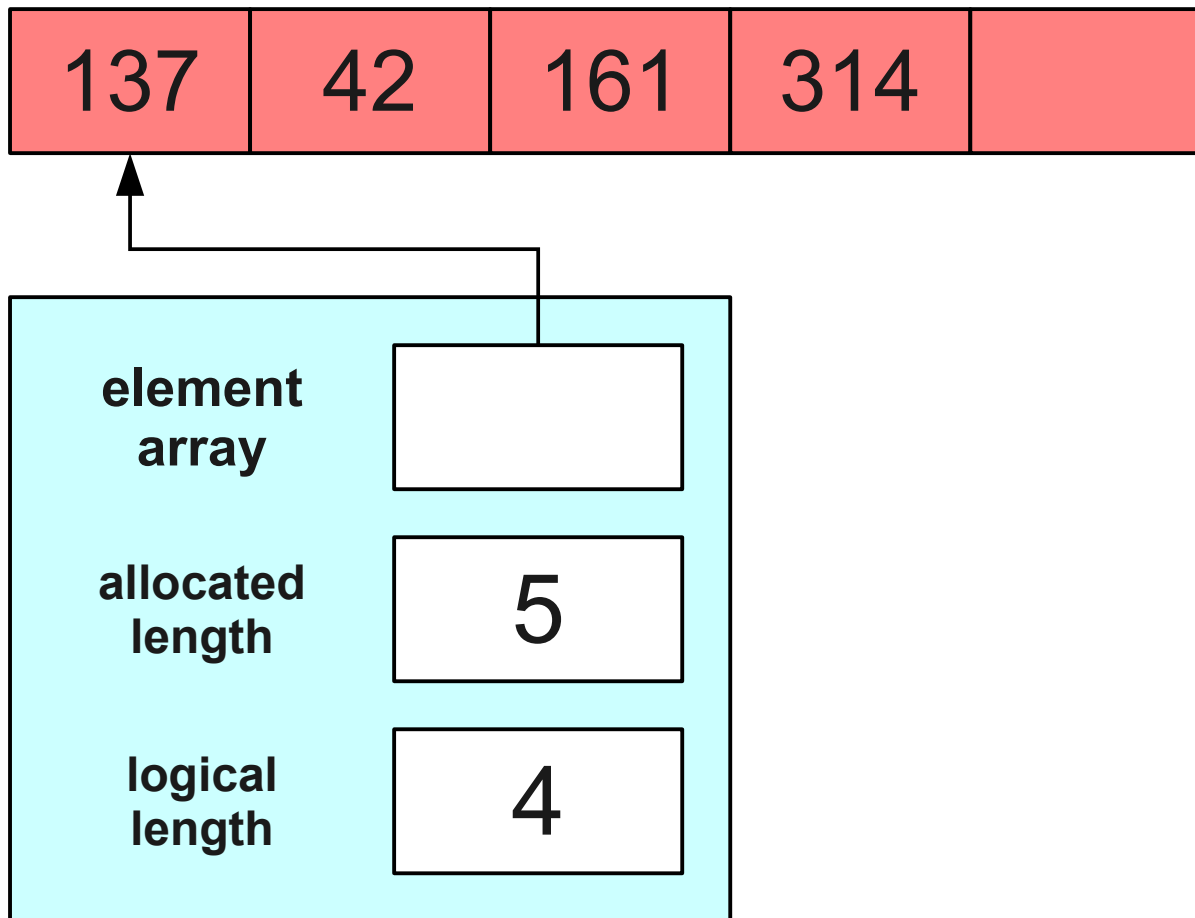
An Initial Idea



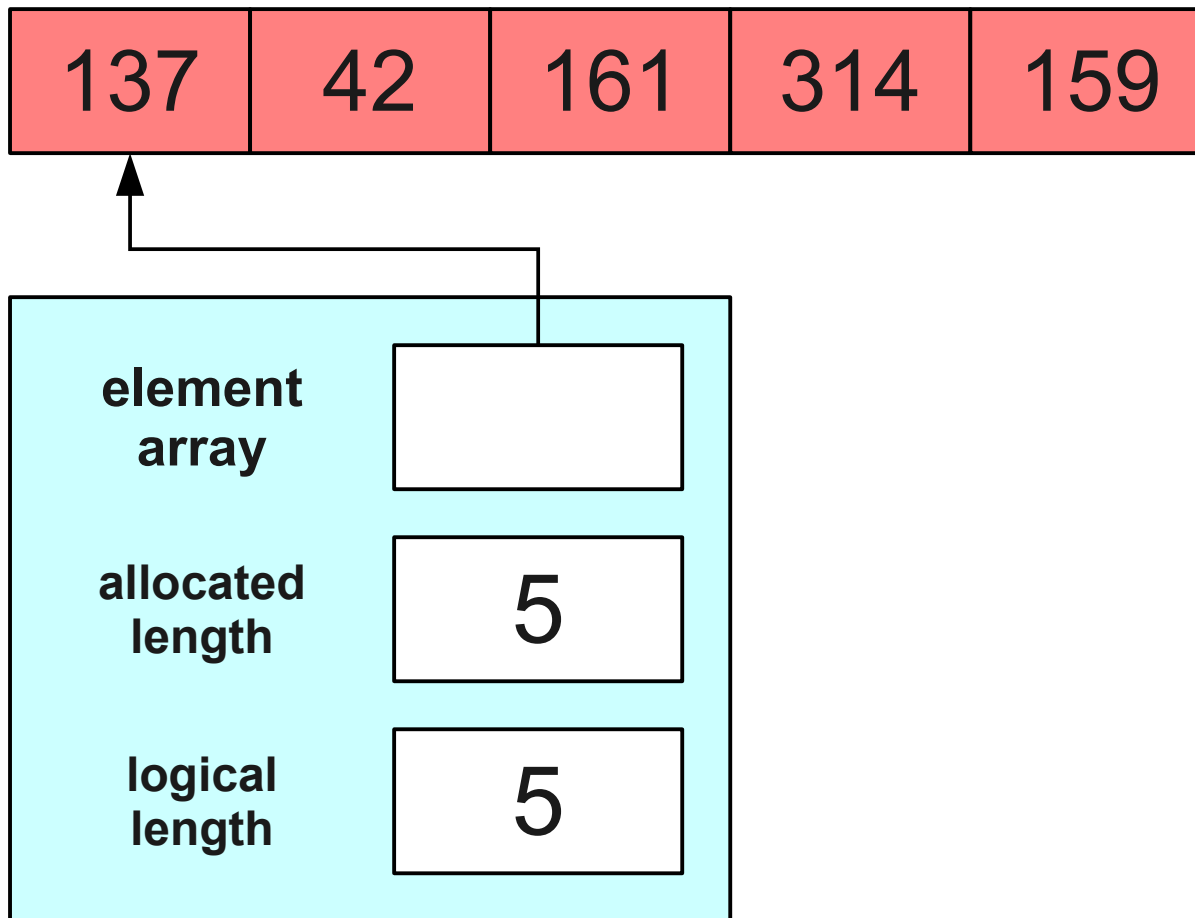
An Initial Idea



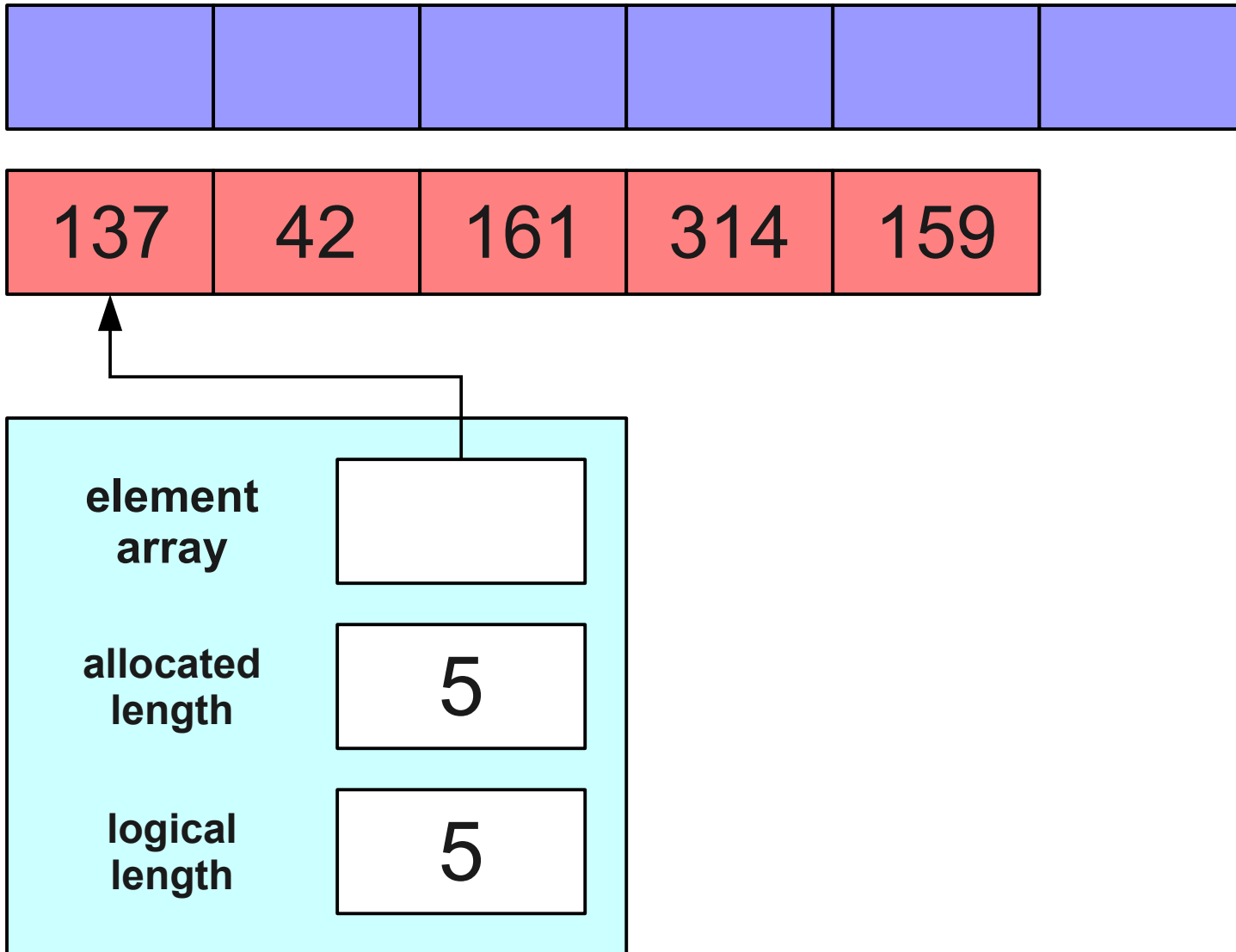
An Initial Idea



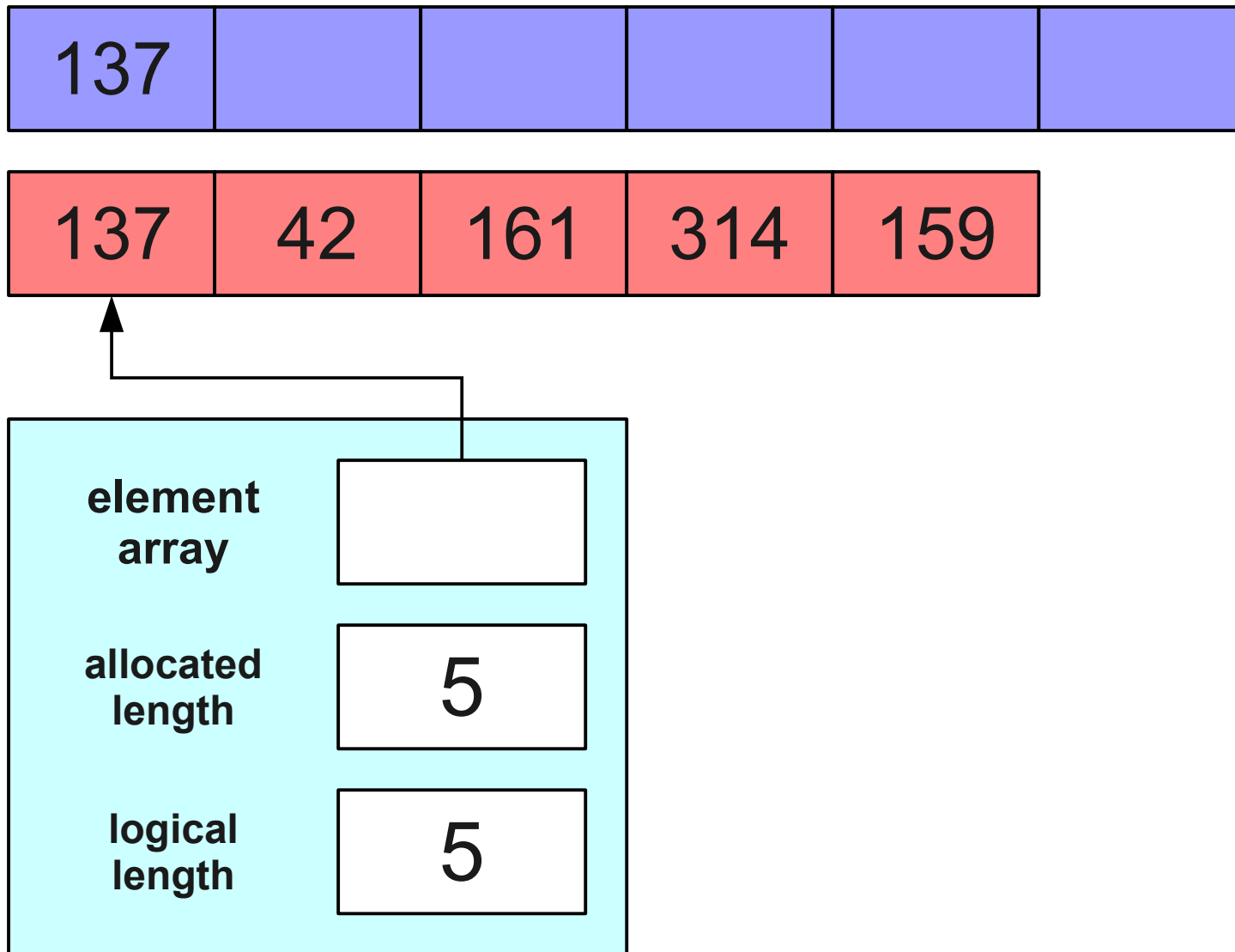
An Initial Idea



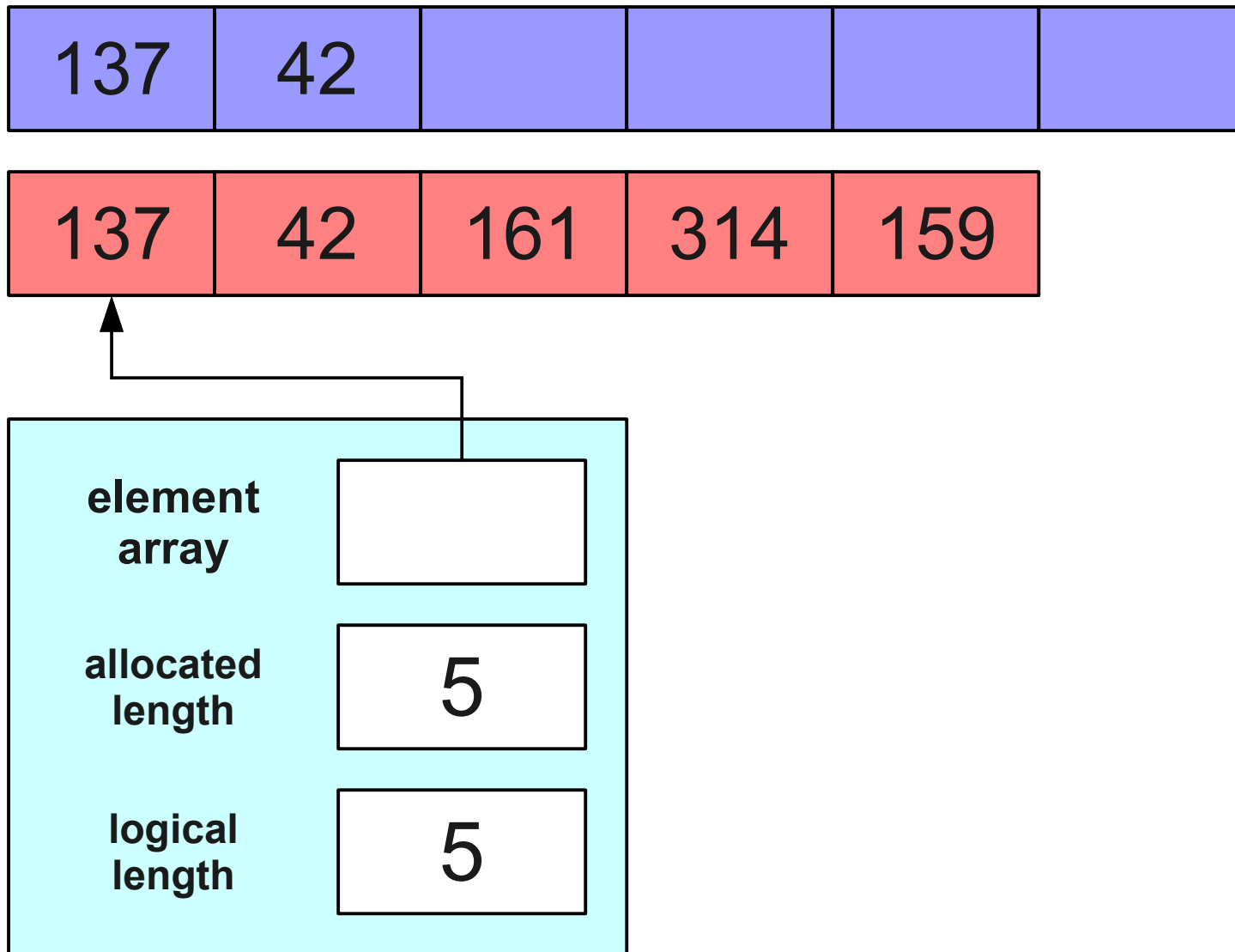
An Initial Idea



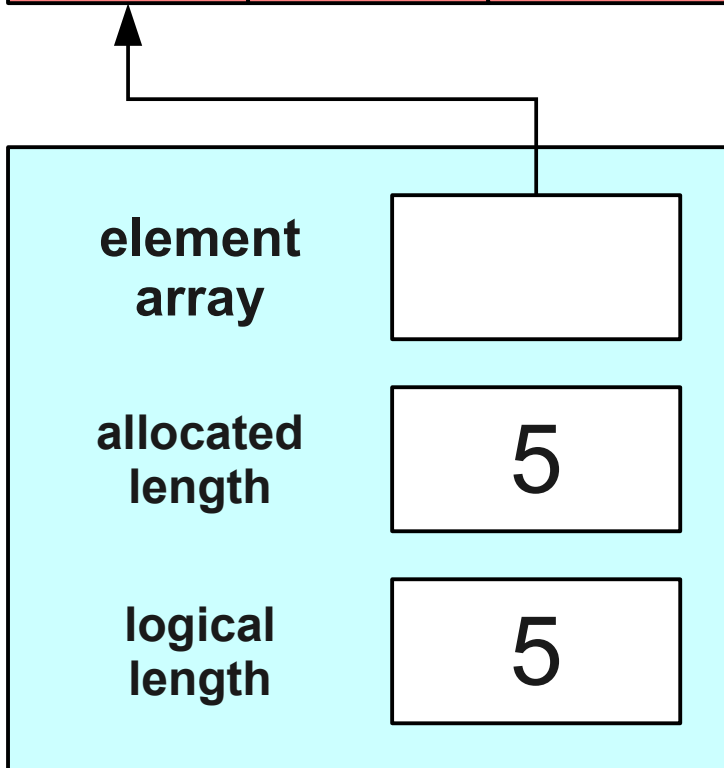
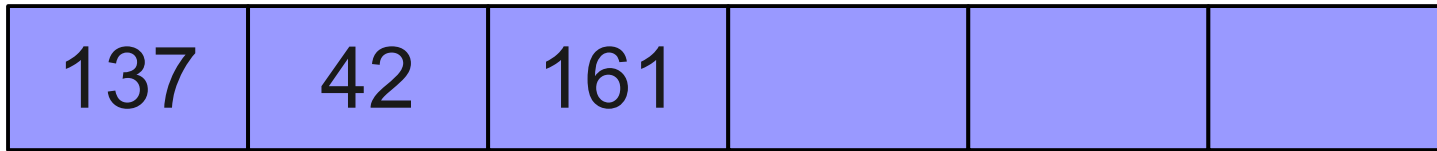
An Initial Idea



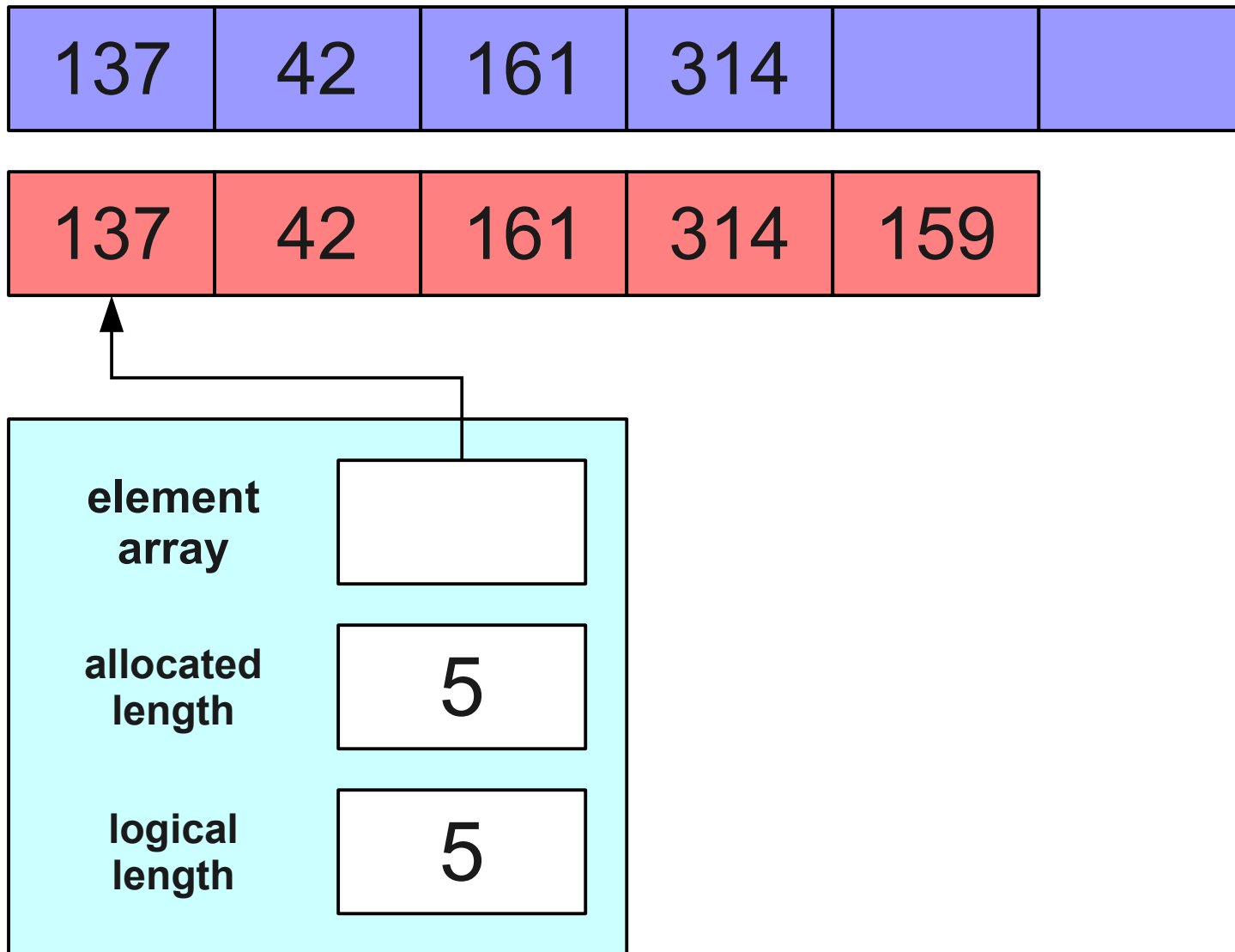
An Initial Idea



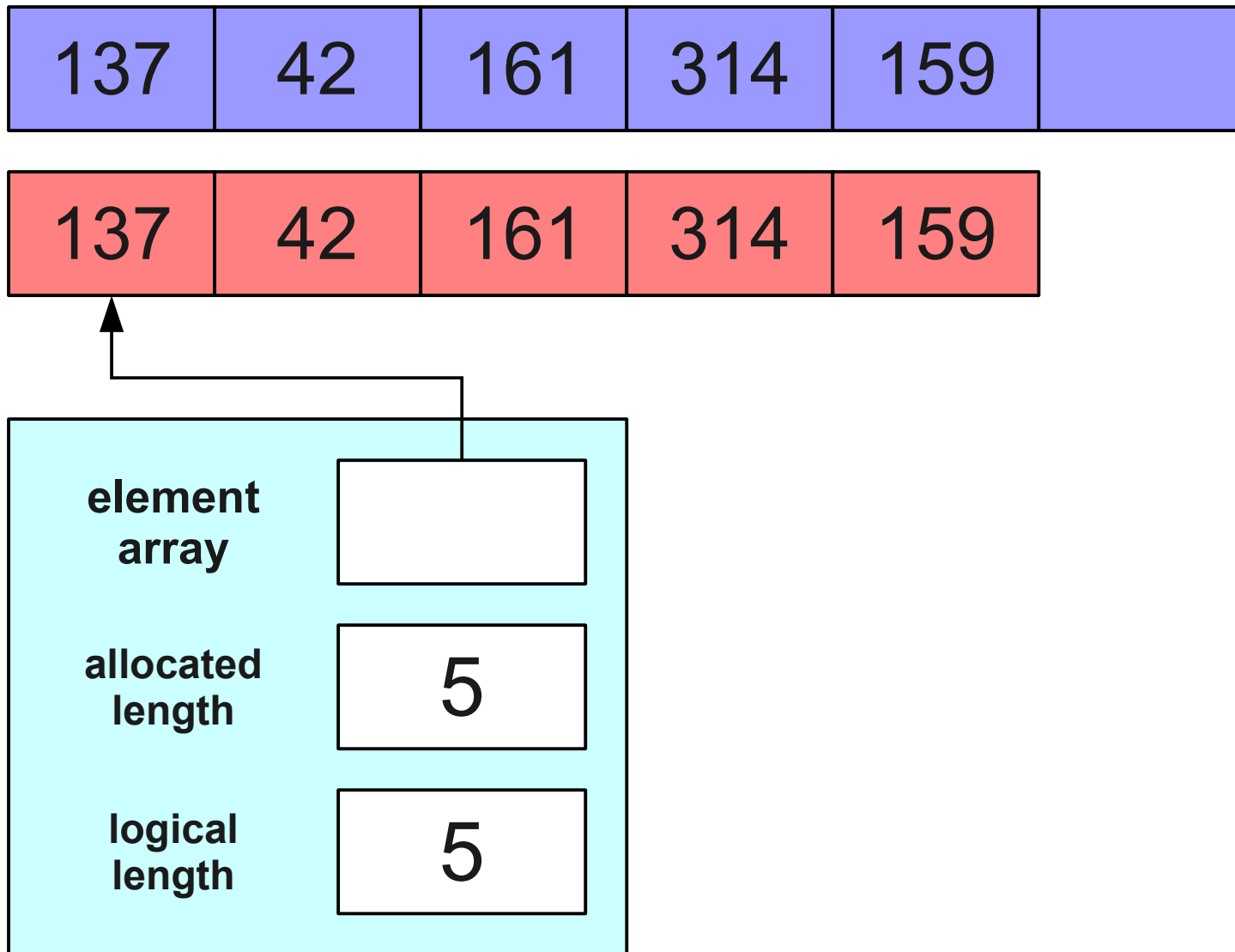
An Initial Idea



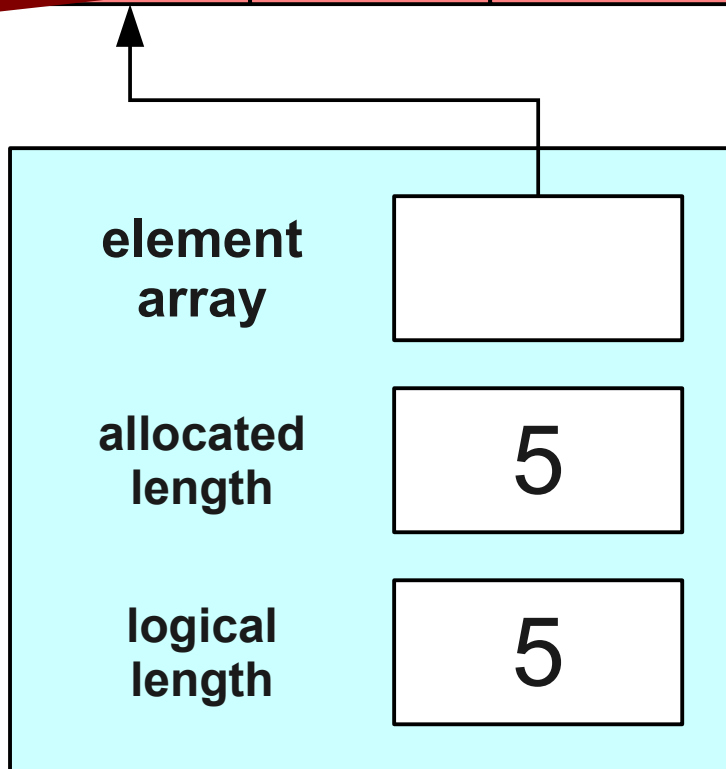
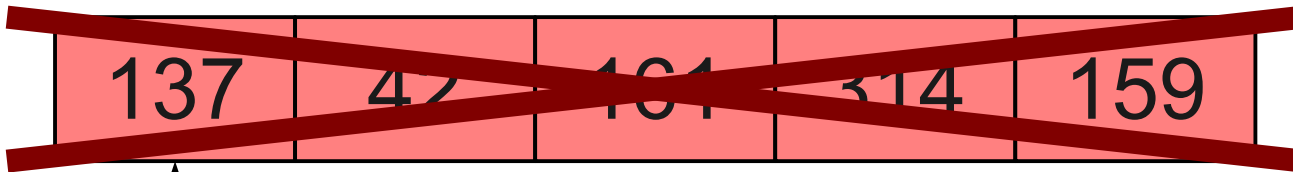
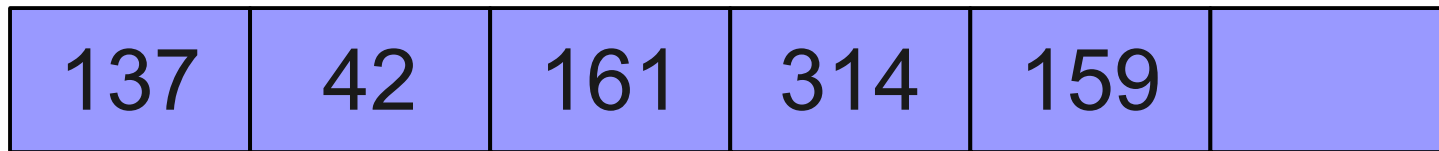
An Initial Idea



An Initial Idea

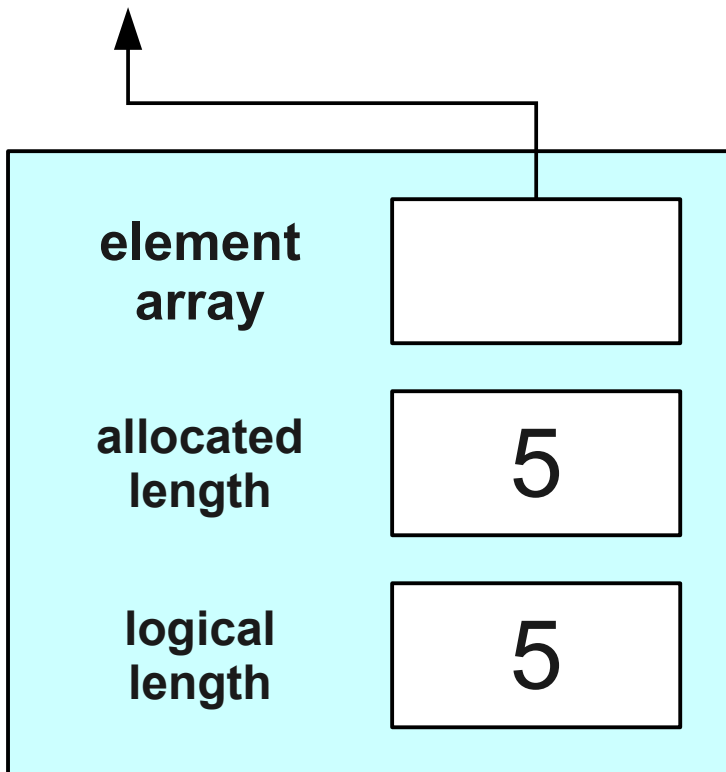


An Initial Idea

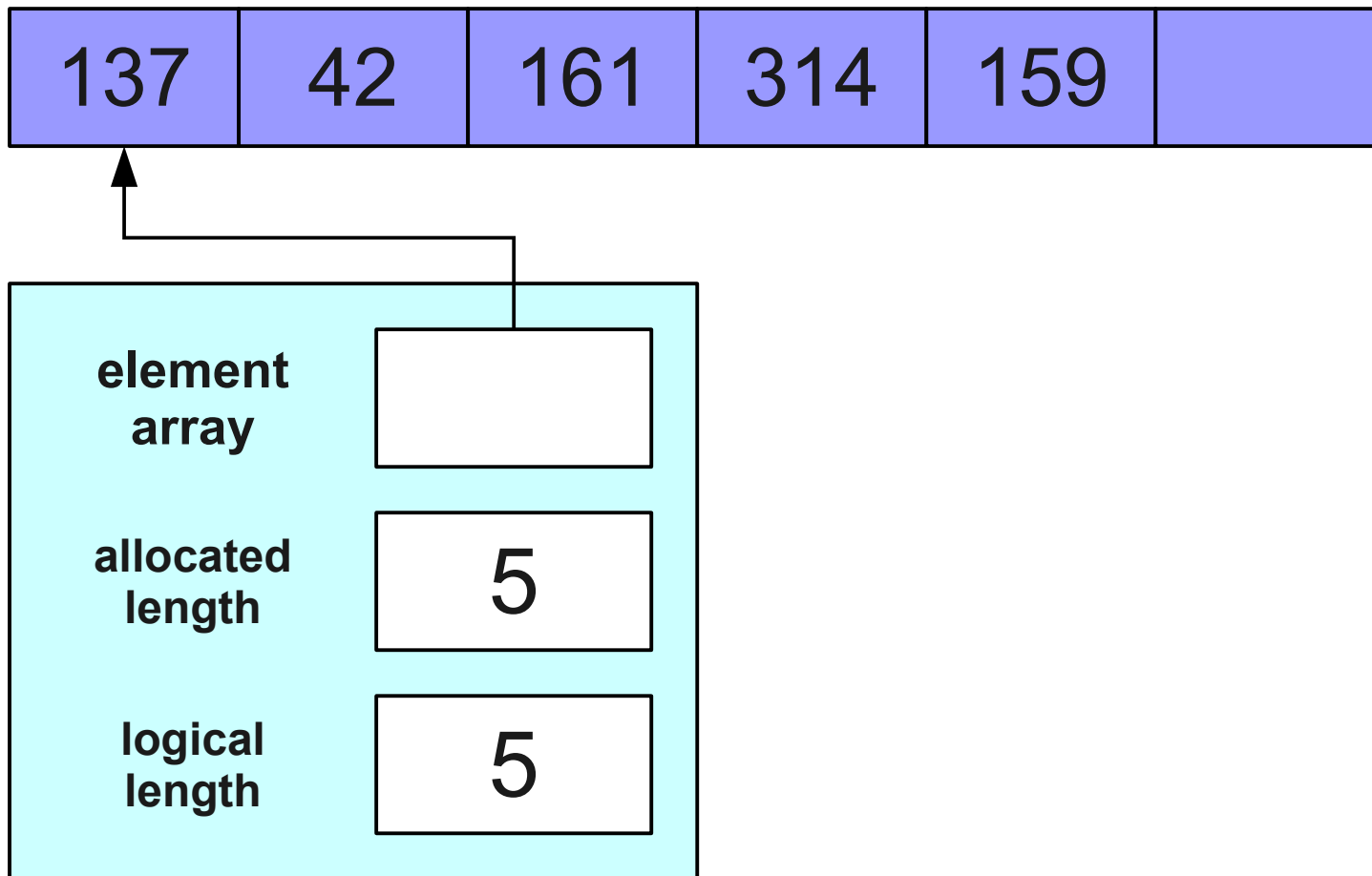


An Initial Idea

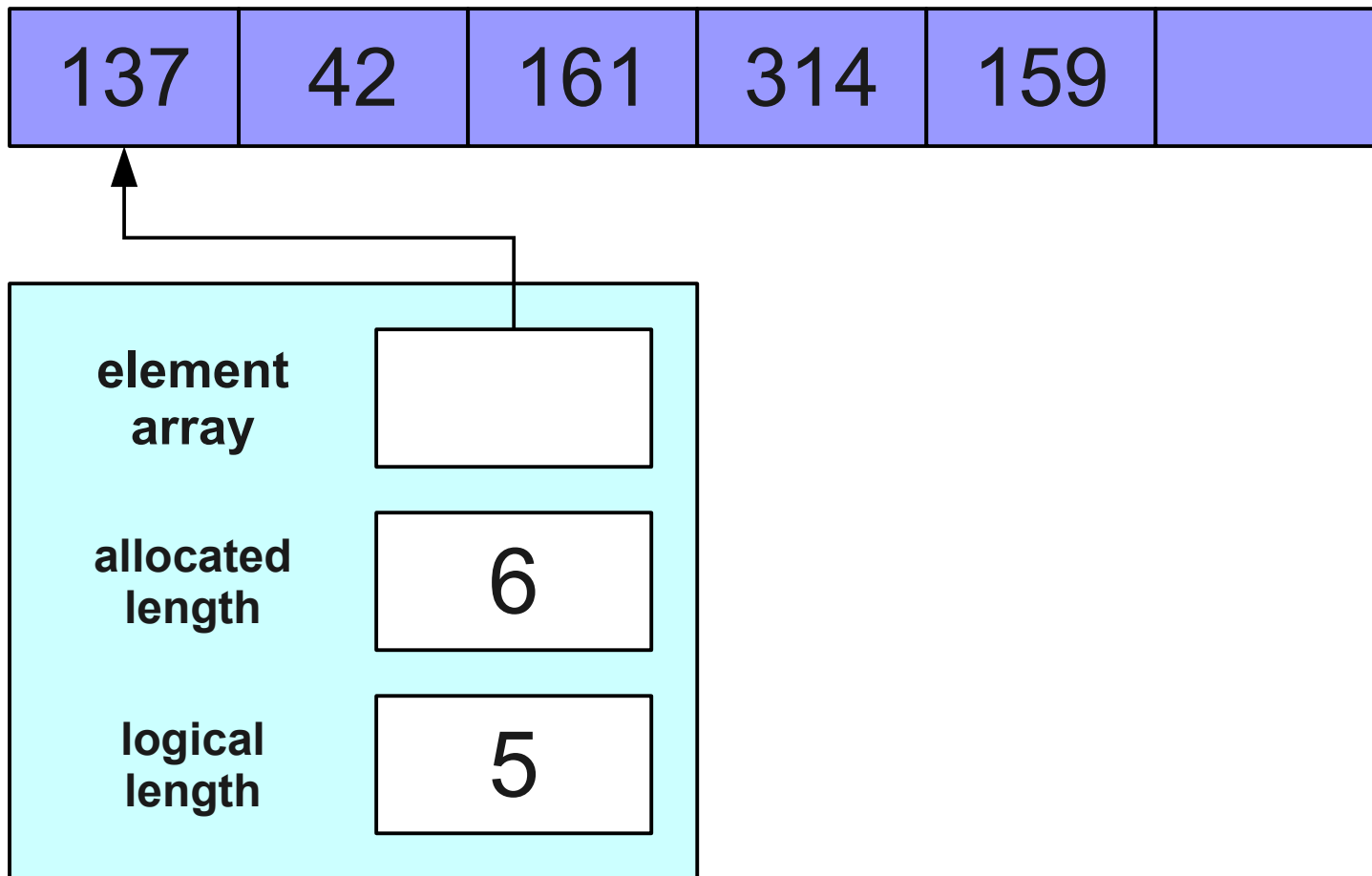
137	42	161	314	159	
-----	----	-----	-----	-----	--



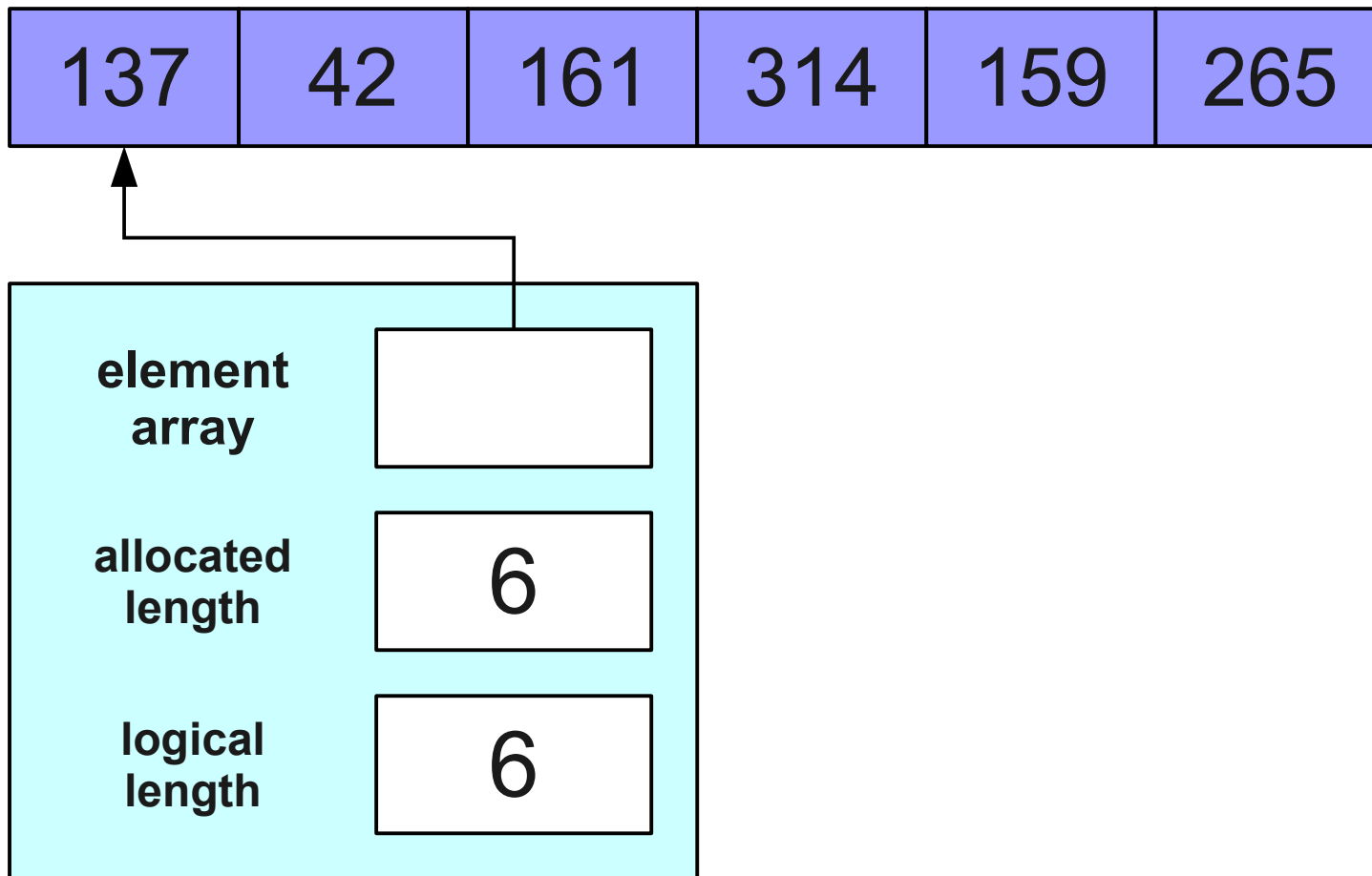
An Initial Idea



An Initial Idea



An Initial Idea



Ready... set... grow!

Analyzing Our Approach

- We now have a working solution, but is it an *efficient* solution?
- Let's analyze the big-O complexity of the five operations.

Analyzing Our Approach

- We now have a working solution, but is it an *efficient* solution?
- Let's analyze the big-O complexity of the five operations.
 - **size: $O(1)$**
 - **isEmpty: $O(1)$**
 - **push: $O(n)$**
 - **pop: $O(1)$**
 - **top: $O(1)$**

What This Means

- What is the complexity of pushing n elements and then popping them?

What This Means

- What is the complexity of pushing n elements and then popping them?
- Cost of the pushes:
 - $1 + 2 + 3 + 4 + \dots + n$

What This Means

- What is the complexity of pushing n elements and then popping them?
- Cost of the pushes:
 - $1 + 2 + 3 + 4 + \dots + n = \mathbf{O(n^2)}$

What This Means

- What is the complexity of pushing n elements and then popping them?
- Cost of the pushes:
 - $1 + 2 + 3 + 4 + \dots + n = \mathbf{O(n^2)}$
- Cost of the pops:
 - $1 + 1 + 1 + 1 + \dots + 1$

What This Means

- What is the complexity of pushing n elements and then popping them?
- Cost of the pushes:
 - $1 + 2 + 3 + 4 + \dots + n = \mathbf{O(n^2)}$
- Cost of the pops:
 - $1 + 1 + 1 + 1 + \dots + 1 = \mathbf{O(n)}$

What This Means

- What is the complexity of pushing n elements and then popping them?
- Cost of the pushes:
 - $1 + 2 + 3 + 4 + \dots + n = \mathbf{O(n^2)}$
- Cost of the pops:
 - $1 + 1 + 1 + 1 + \dots + 1 = \mathbf{O(n)}$
- Total cost:

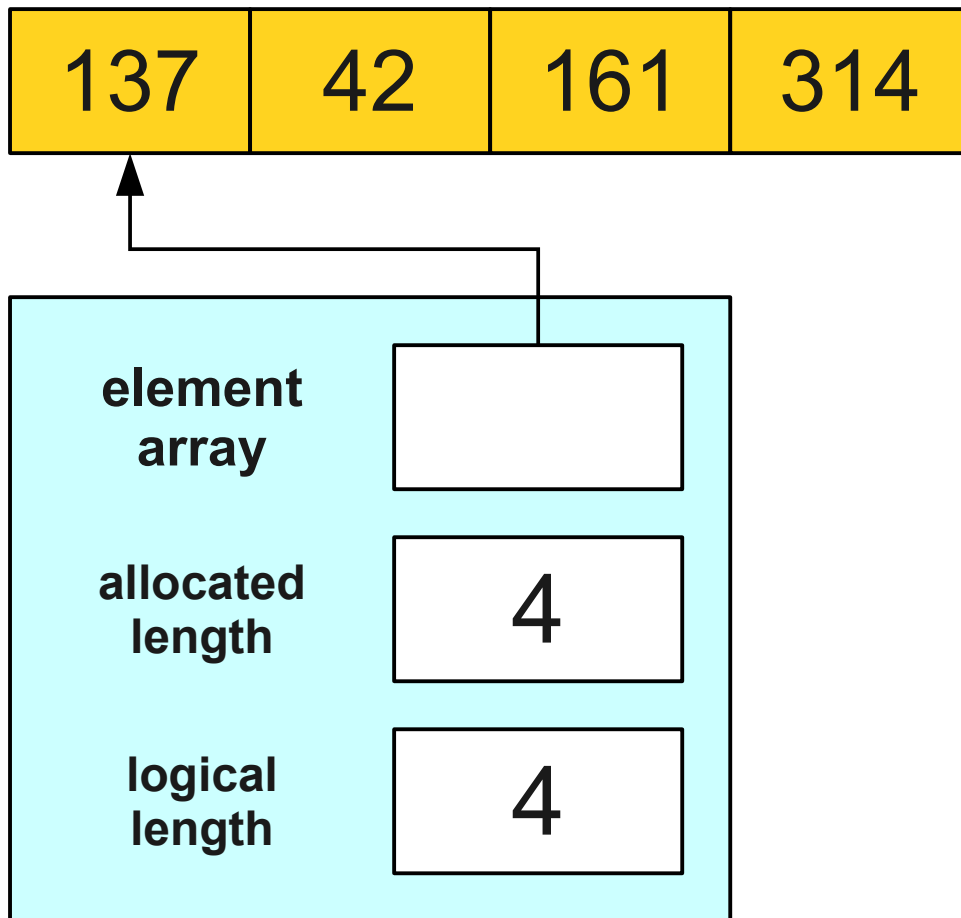
What This Means

- What is the complexity of pushing n elements and then popping them?
- Cost of the pushes:
 - $1 + 2 + 3 + 4 + \dots + n = \mathbf{O(n^2)}$
- Cost of the pops:
 - $1 + 1 + 1 + 1 + \dots + 1 = \mathbf{O(n)}$
- Total cost: $\mathbf{O(n^2)}$

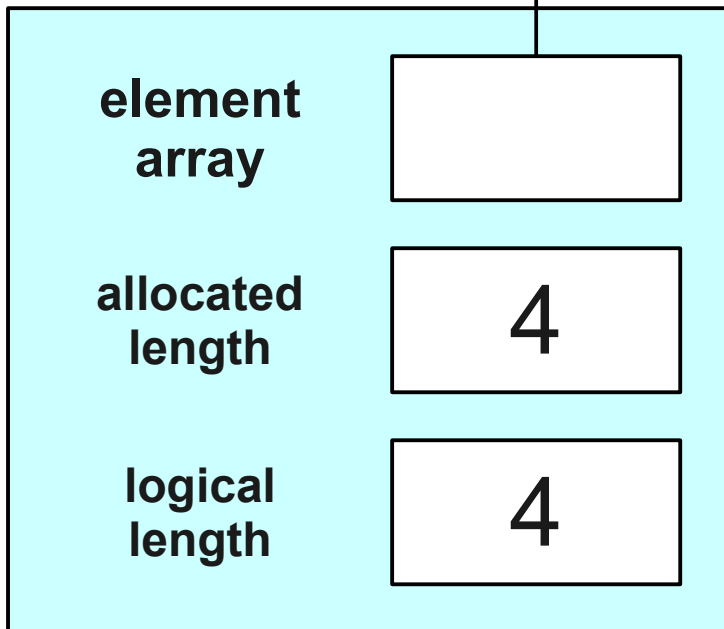
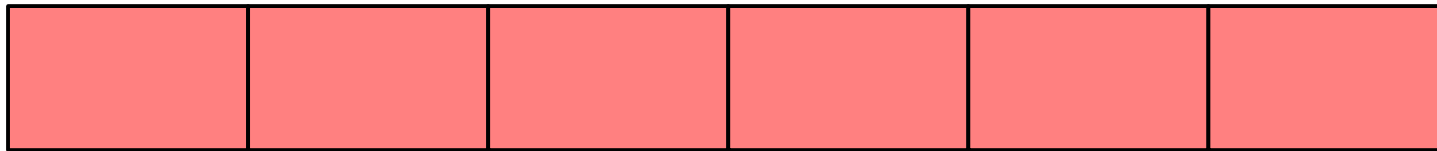
Validating Our Model

Speeding up the Stack

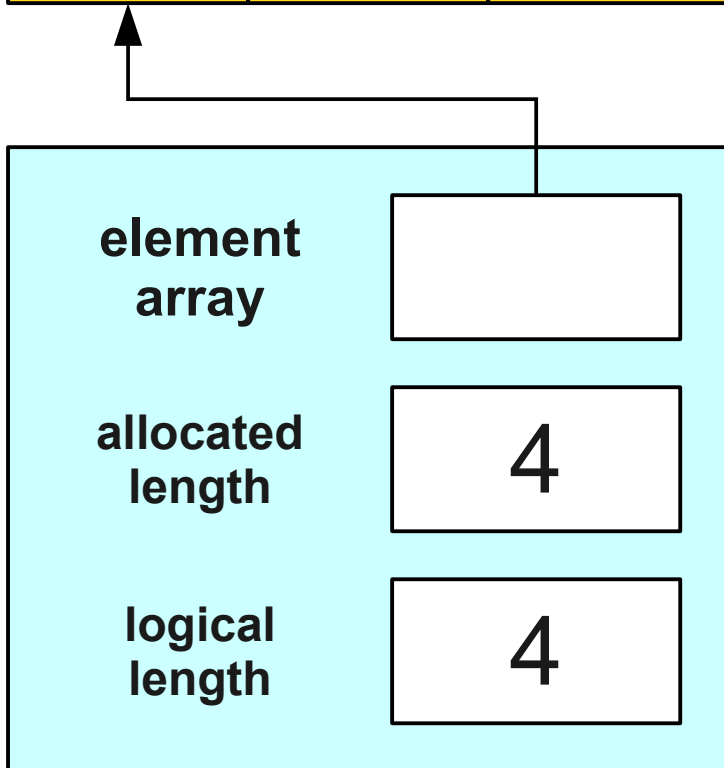
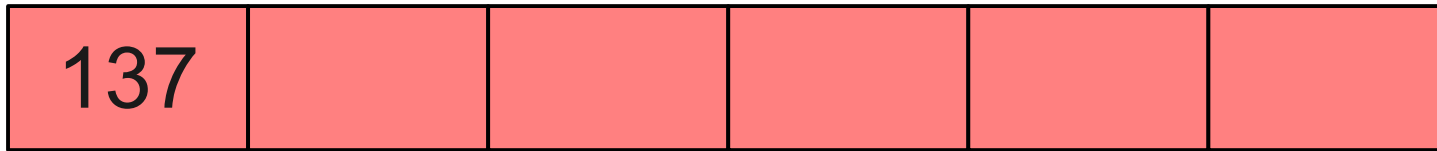
A Better Idea



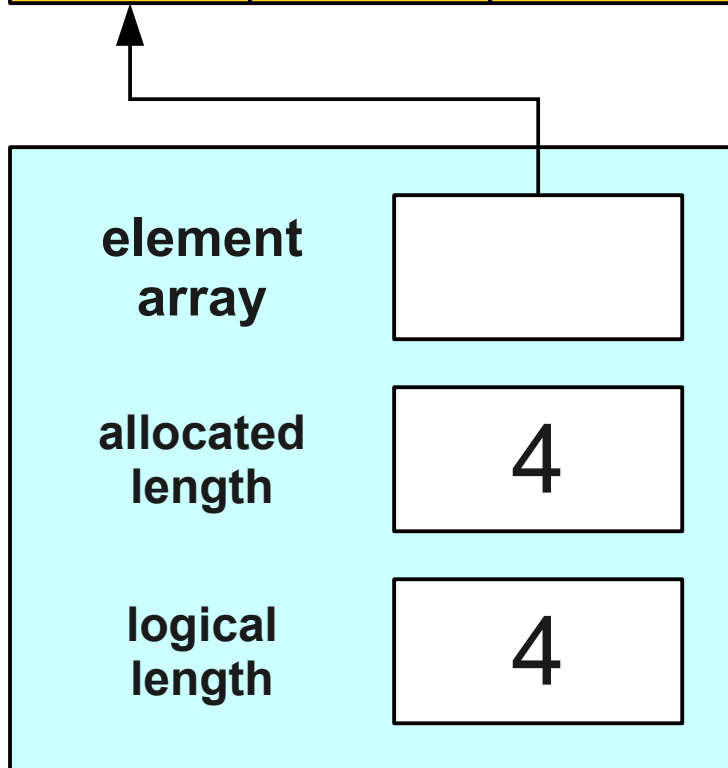
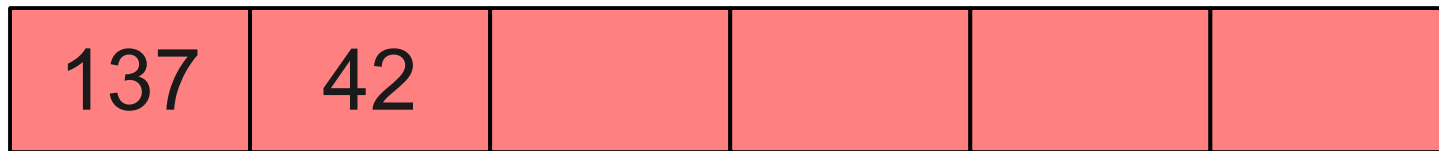
A Better Idea



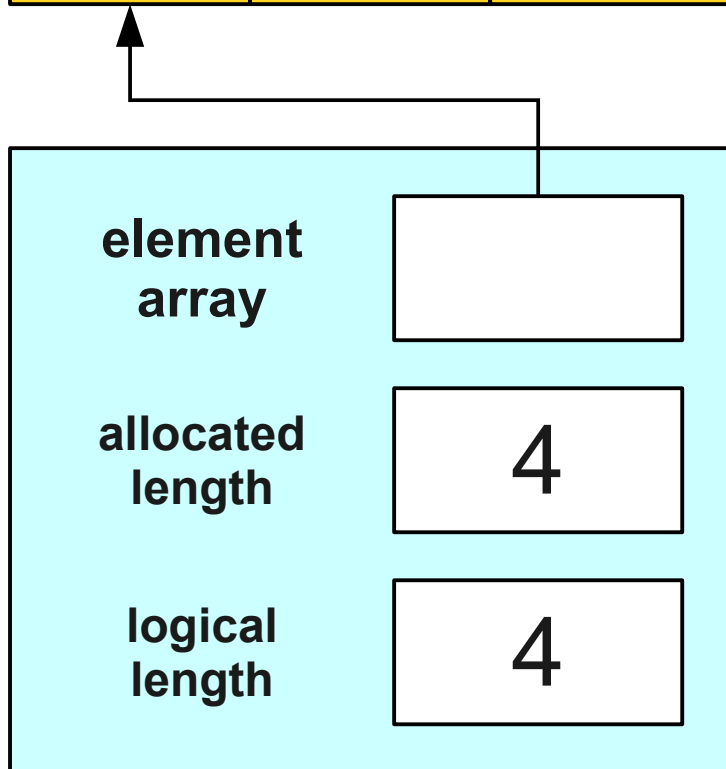
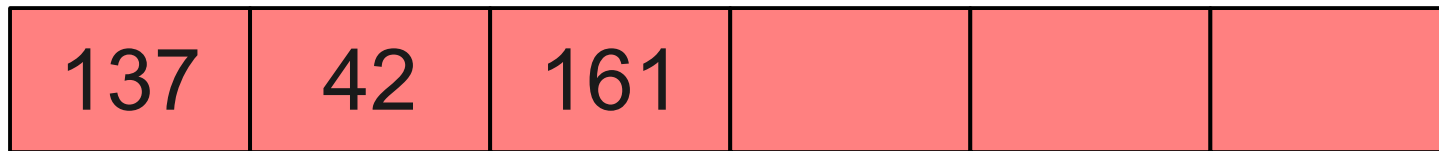
A Better Idea



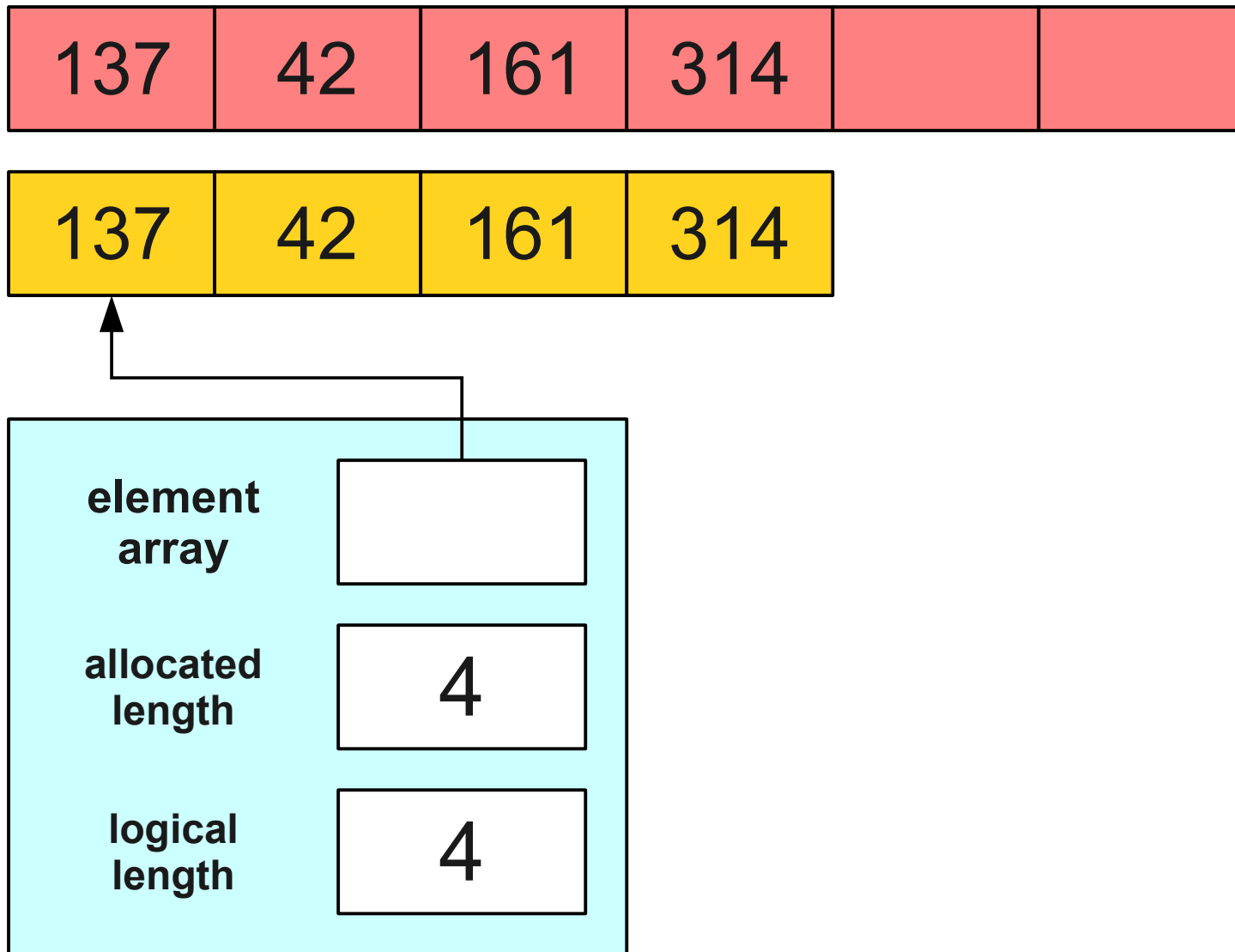
A Better Idea



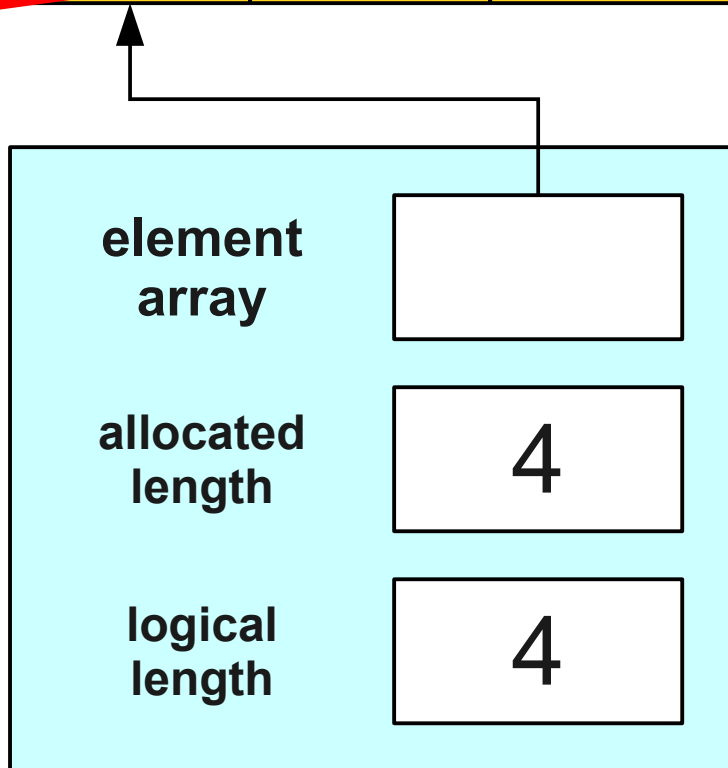
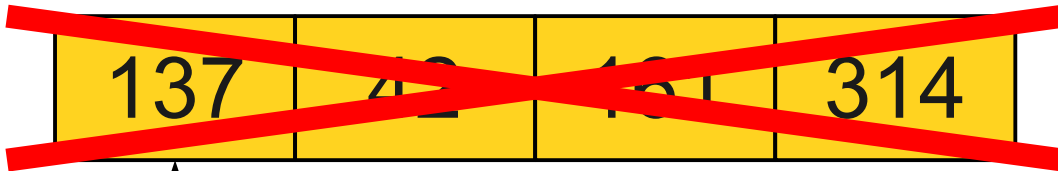
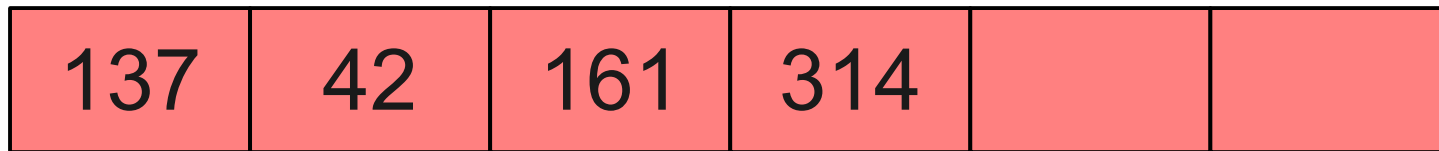
A Better Idea



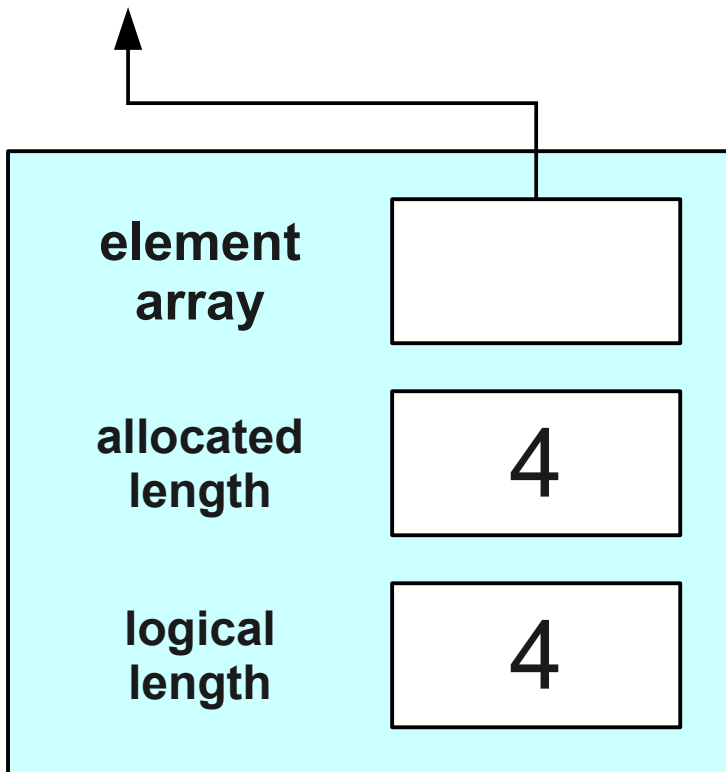
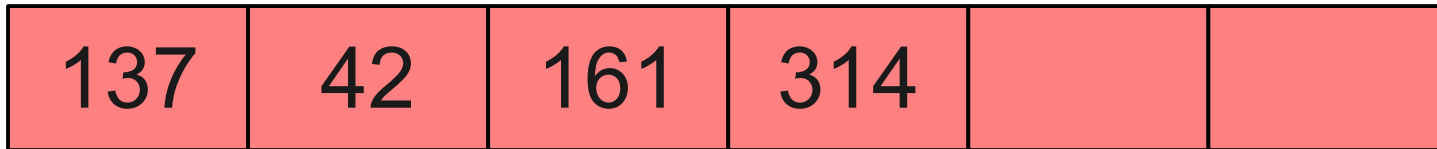
A Better Idea



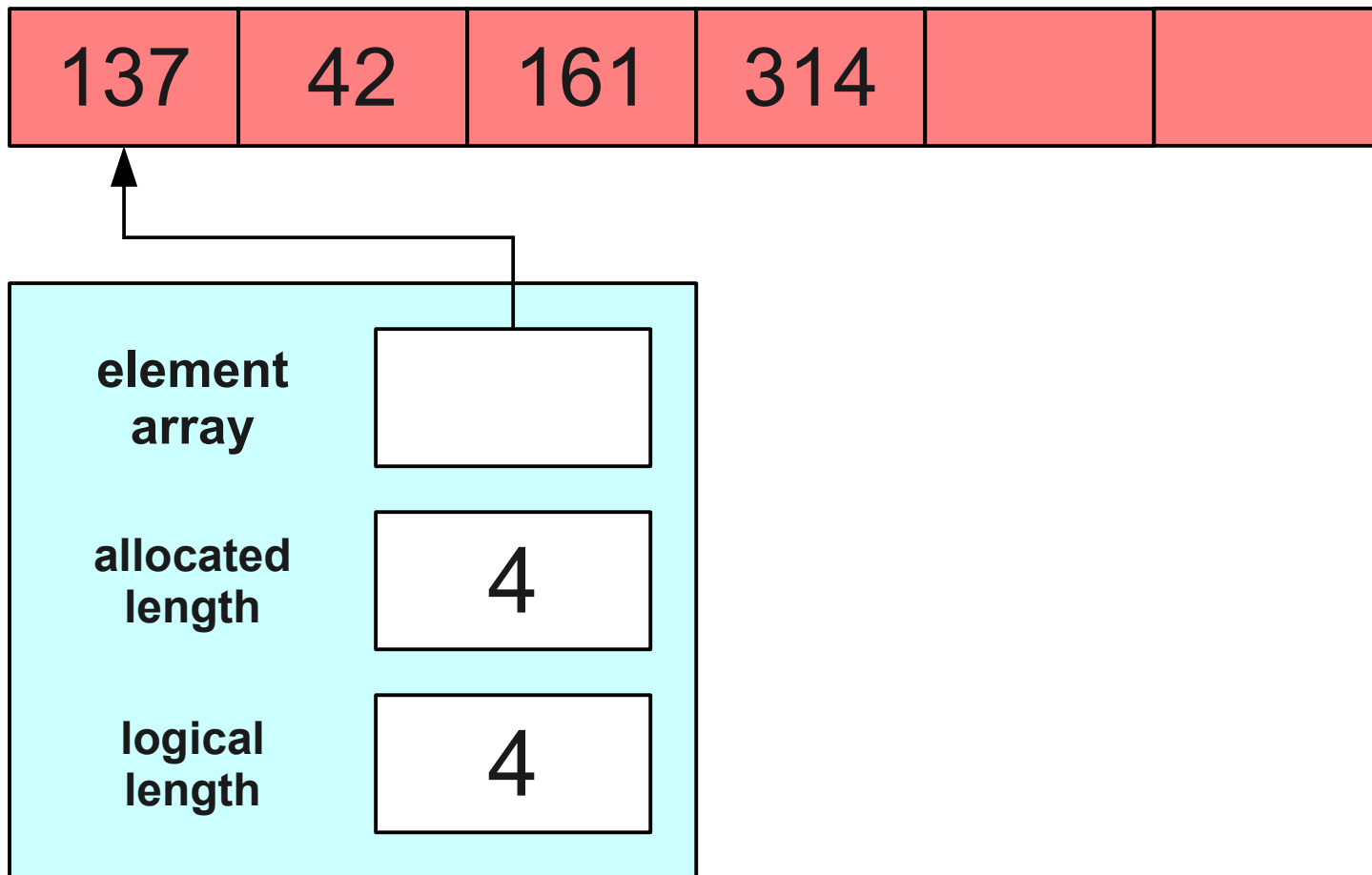
A Better Idea



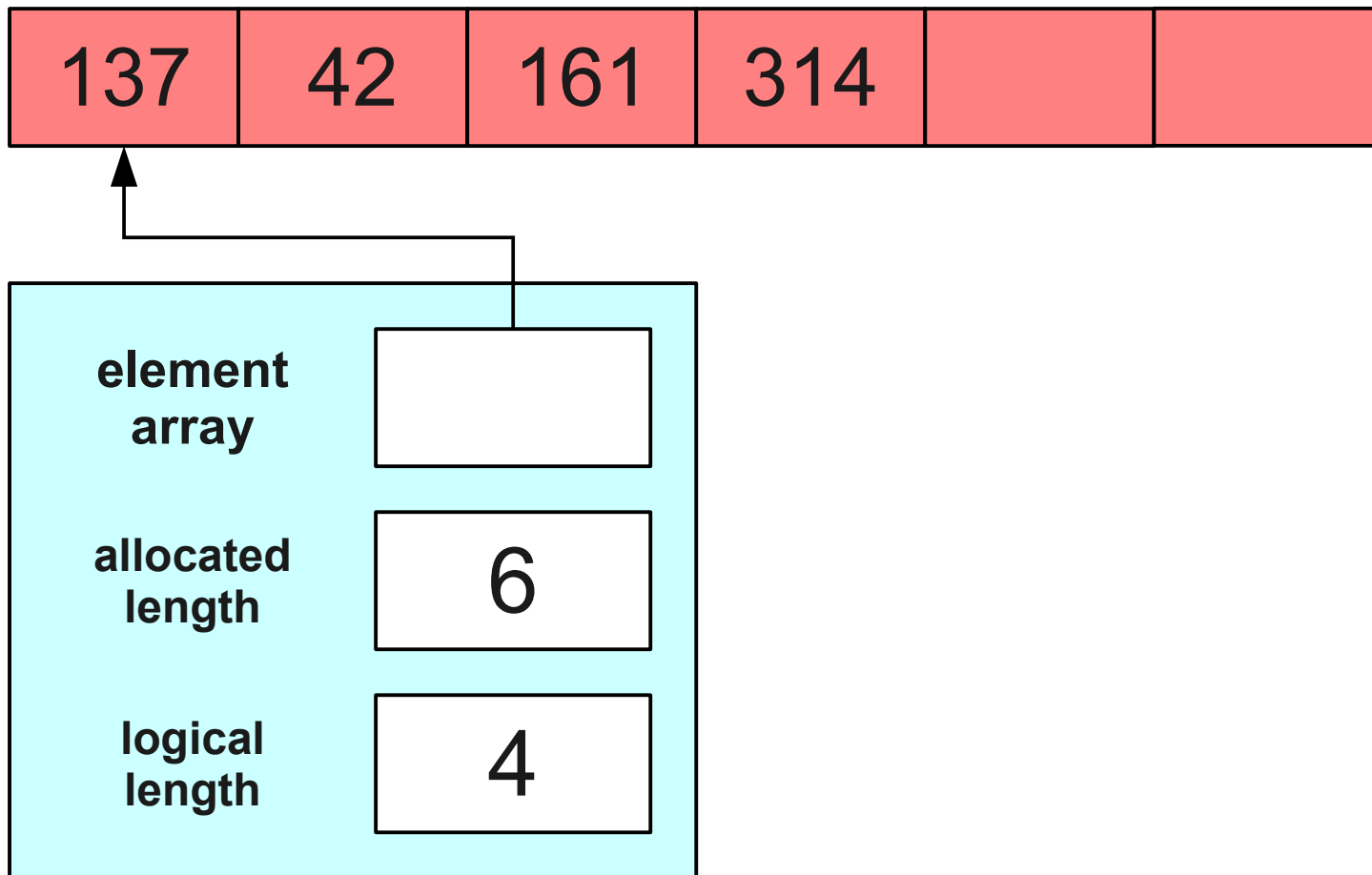
A Better Idea



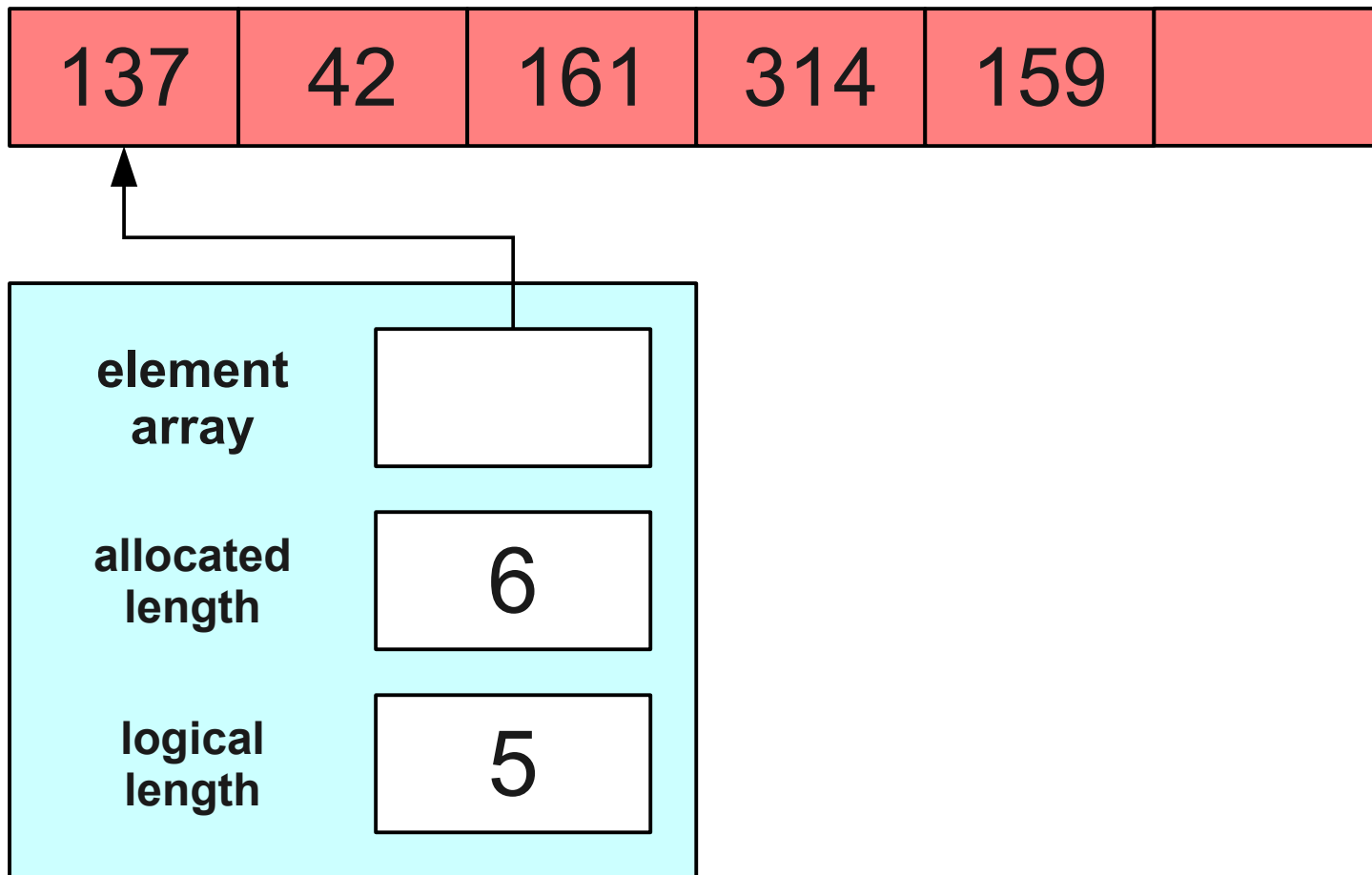
A Better Idea



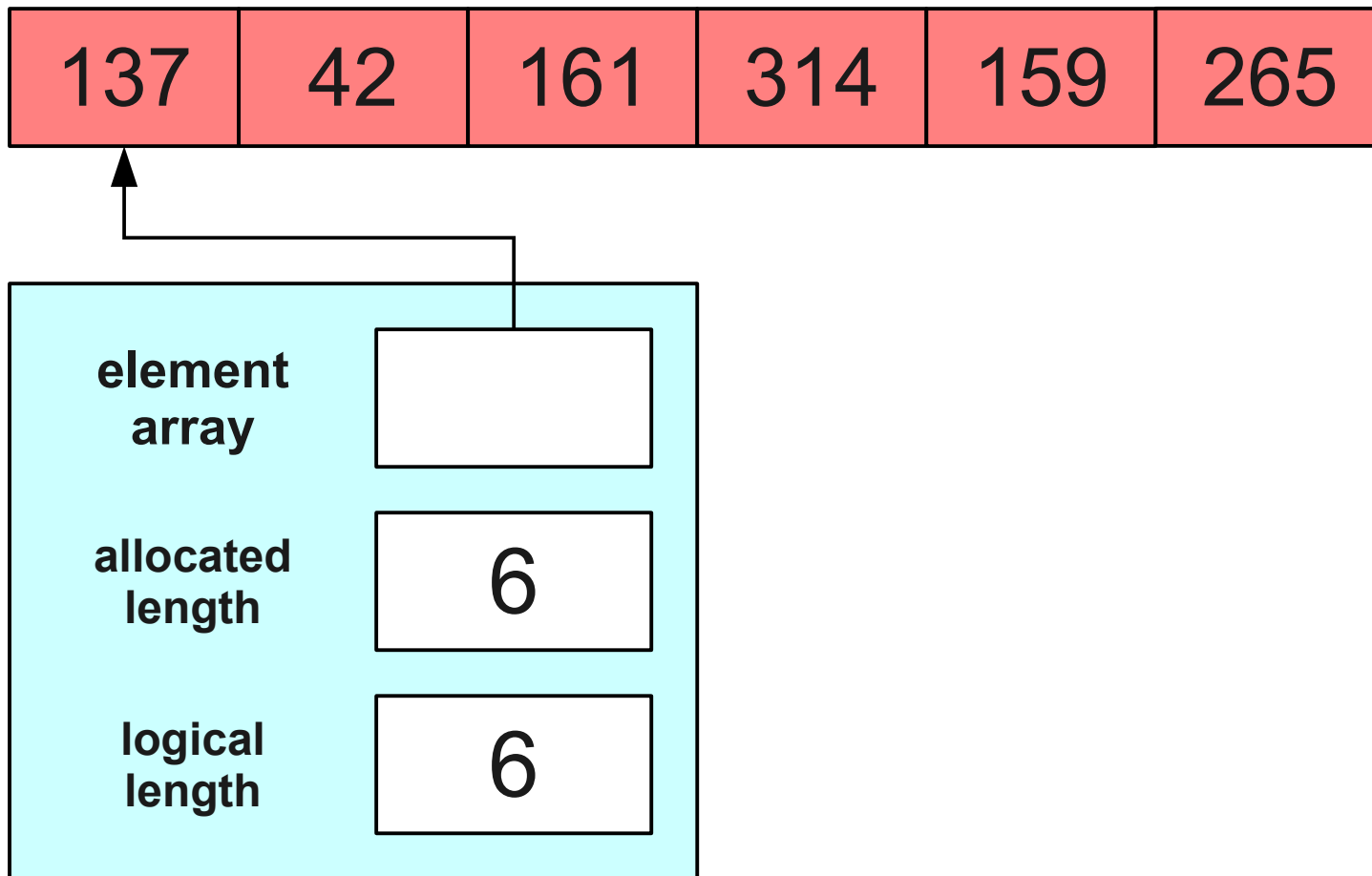
A Better Idea



A Better Idea



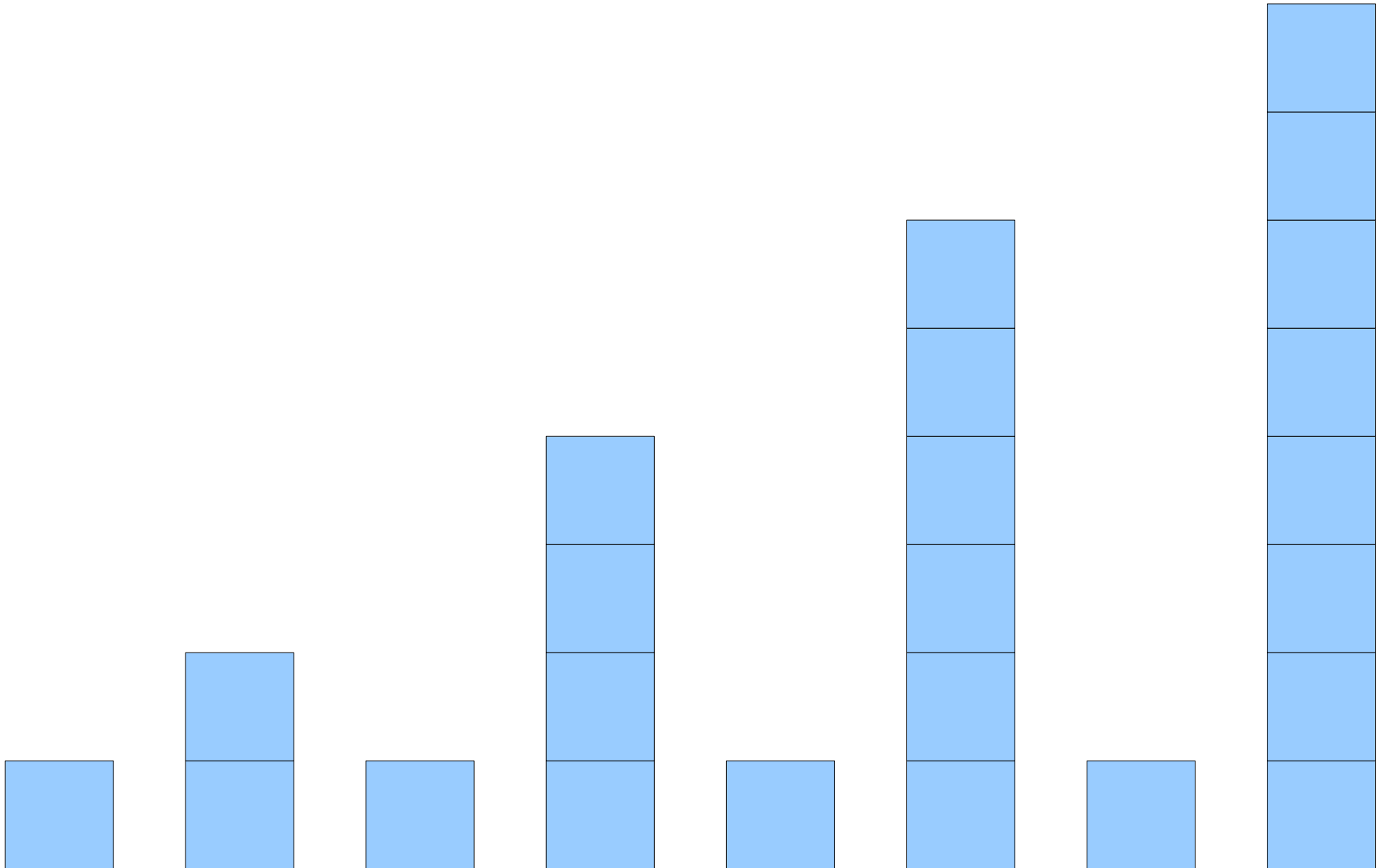
A Better Idea



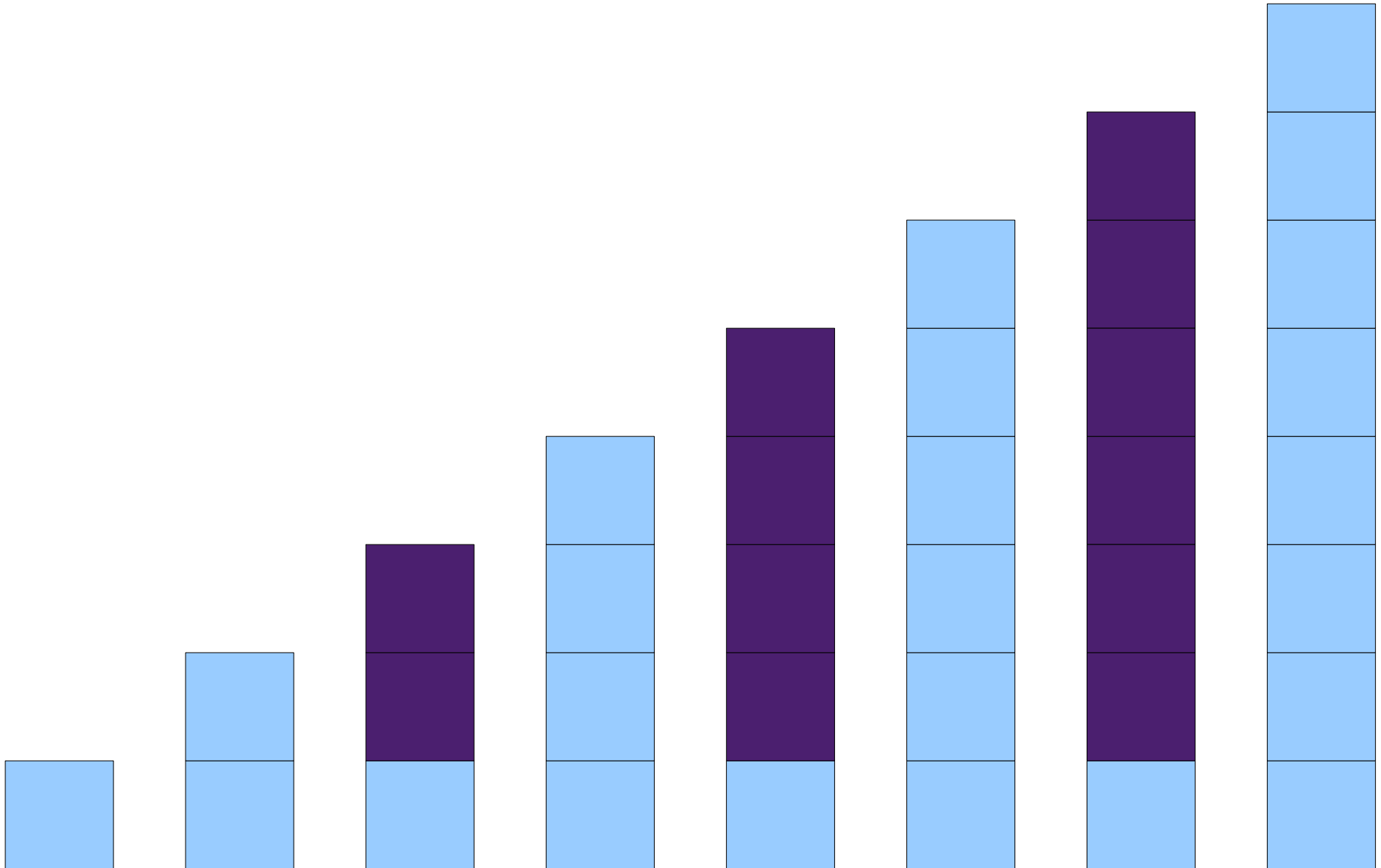
What Just Happened?

- Half of our pushes are now “easy” pushes, and half of our pushes are now “hard” pushes.
- Hard pushes still take time $O(n)$.
- Easy pushes only take time $O(1)$.
- Worst-case is still $O(n)$.
- What about the average case?

Analyzing the Work

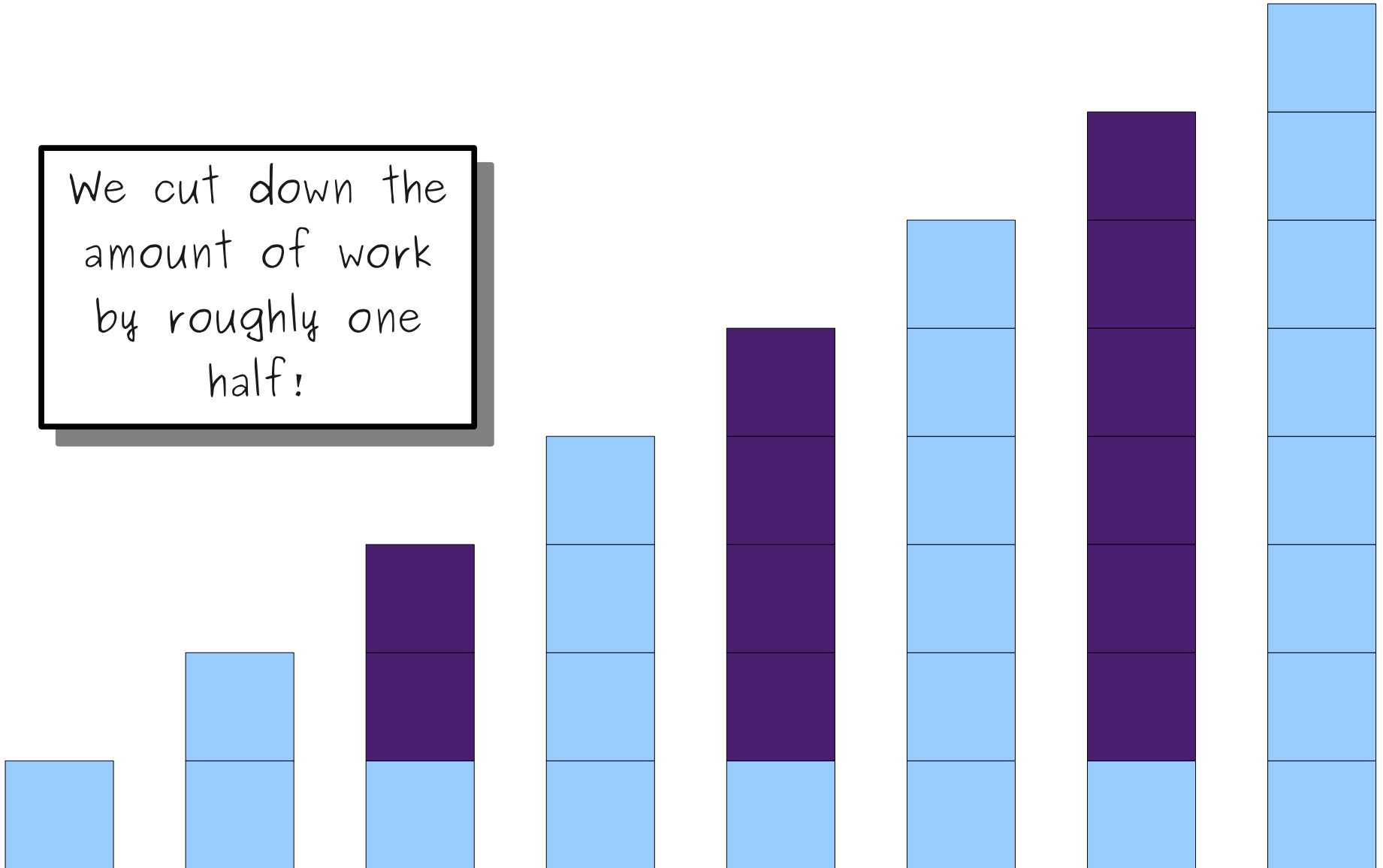


Analyzing the Work



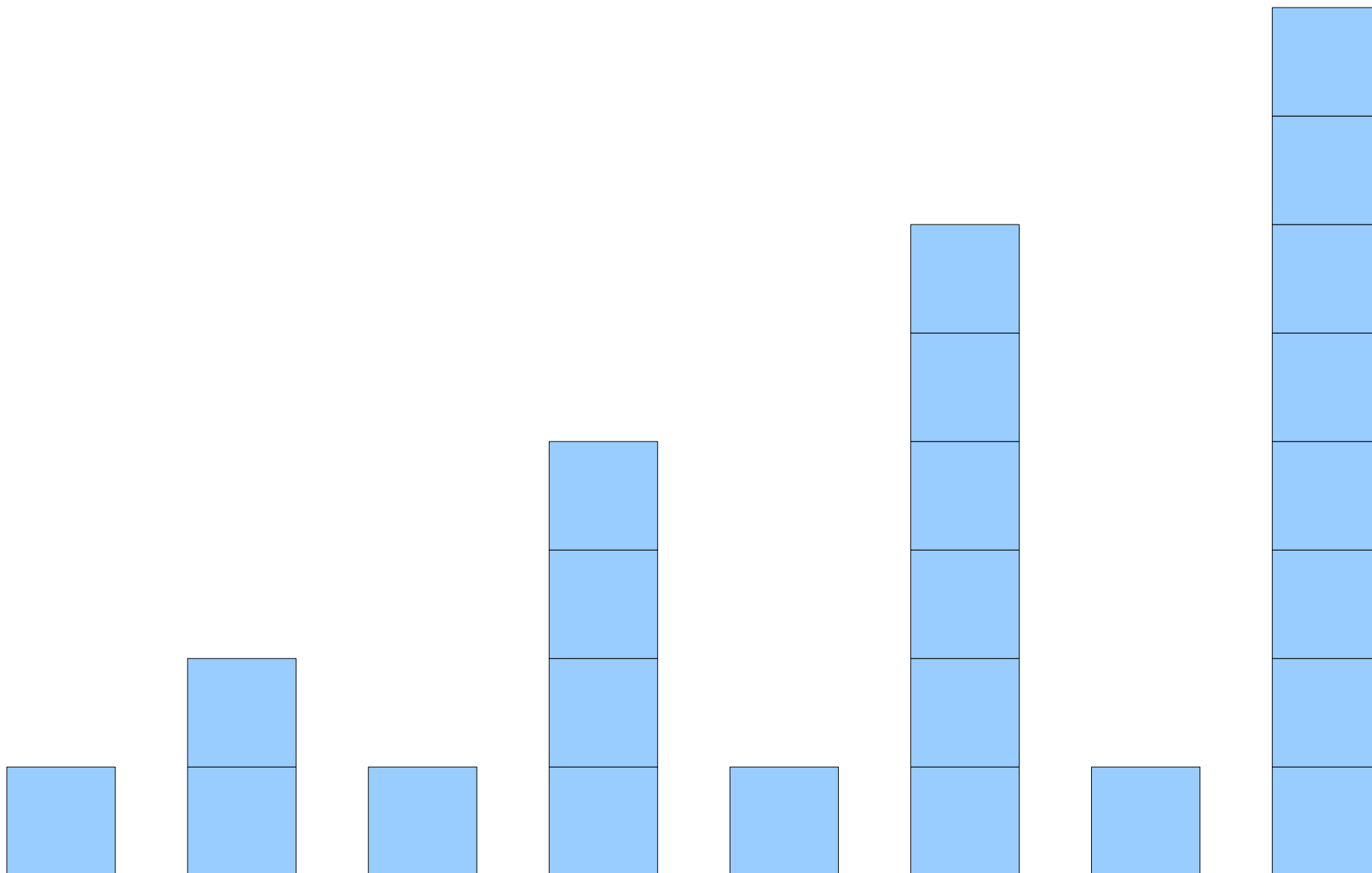
Analyzing the Work

We cut down the
amount of work
by roughly one
half!

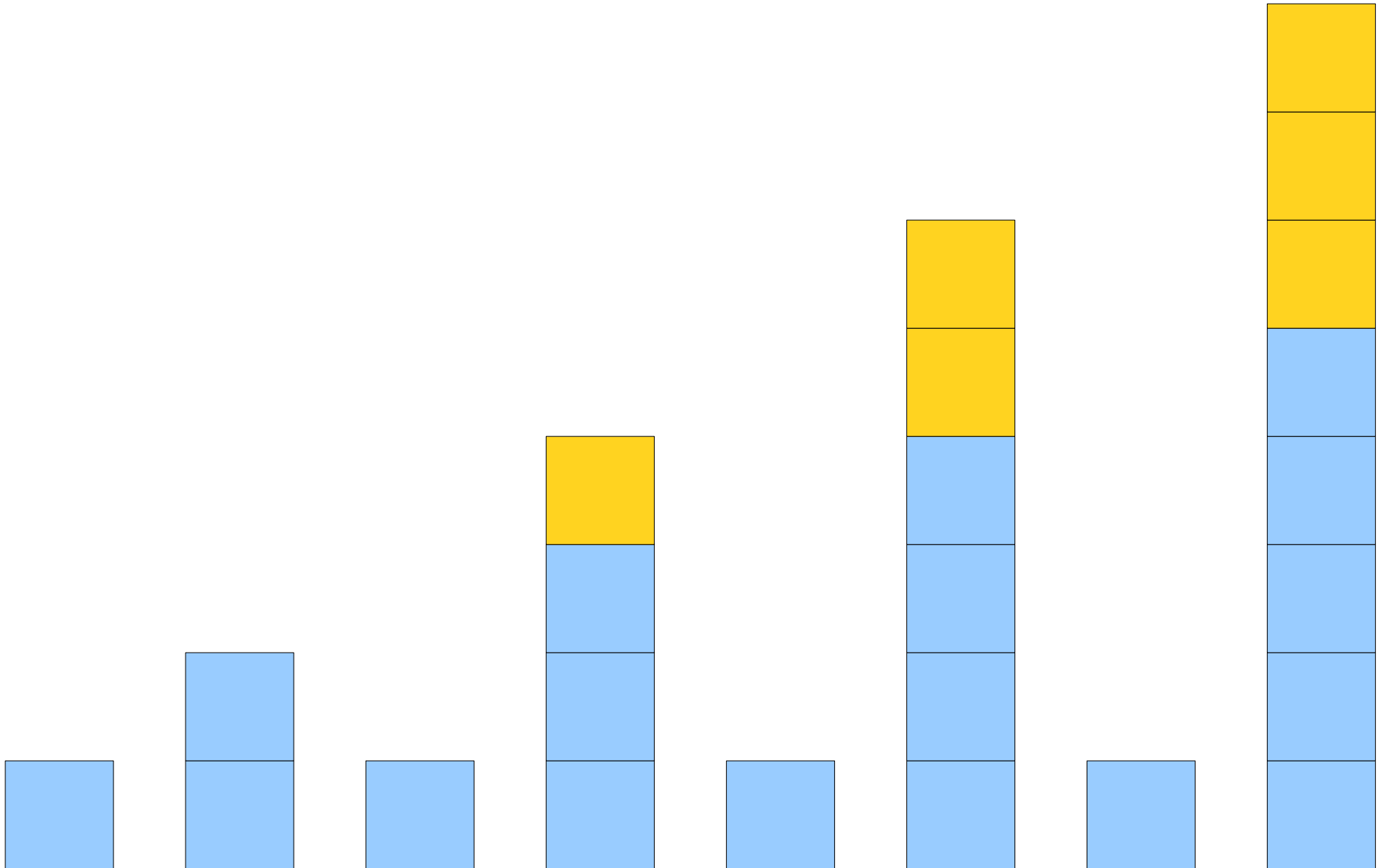


A Different Analysis

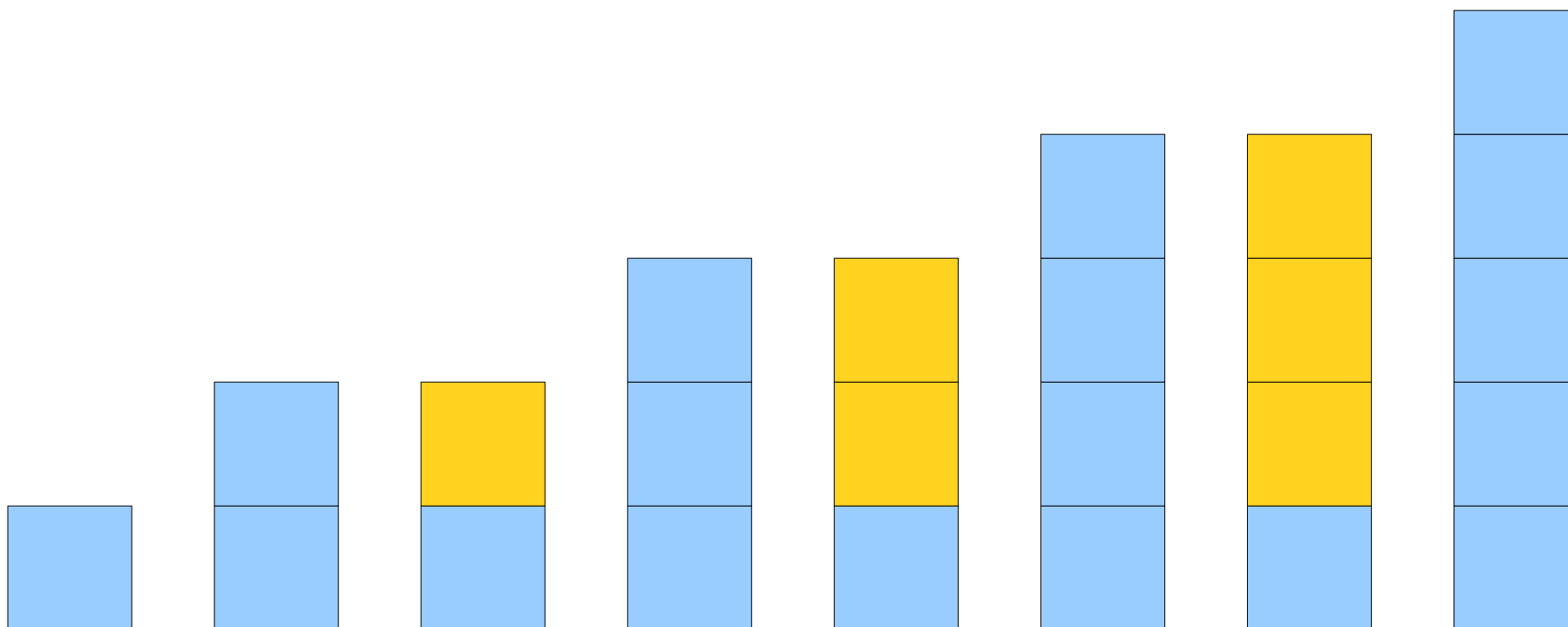
A Different Analysis



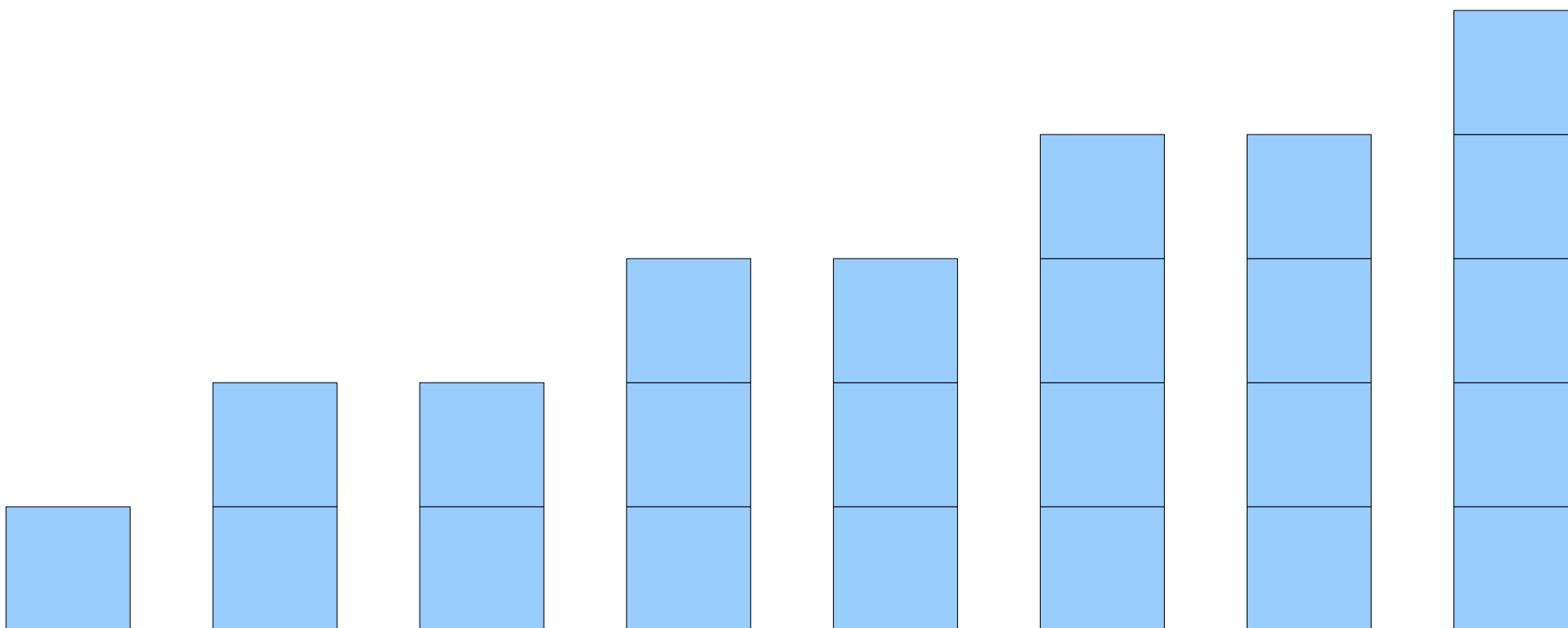
A Different Analysis



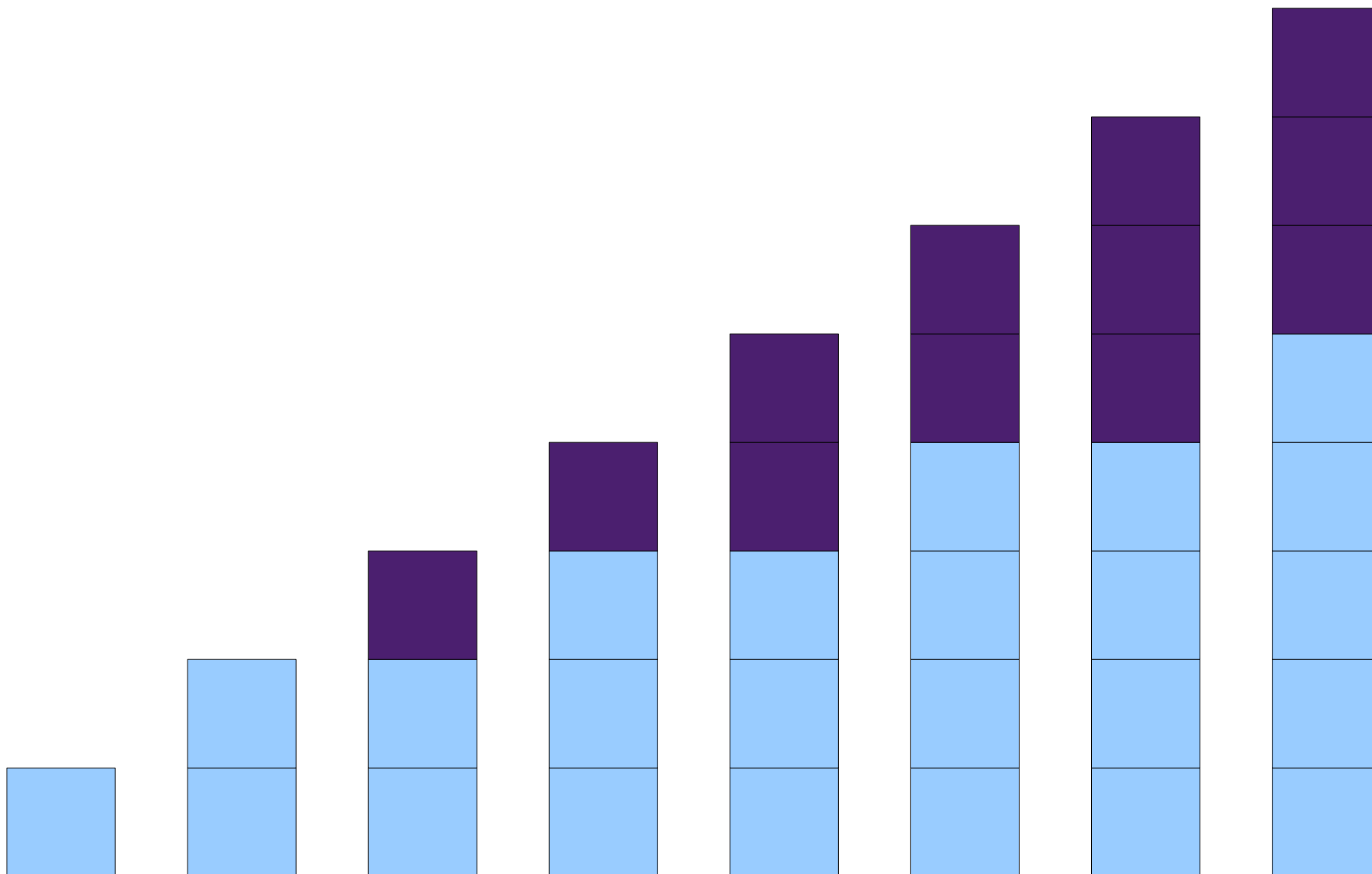
A Different Analysis



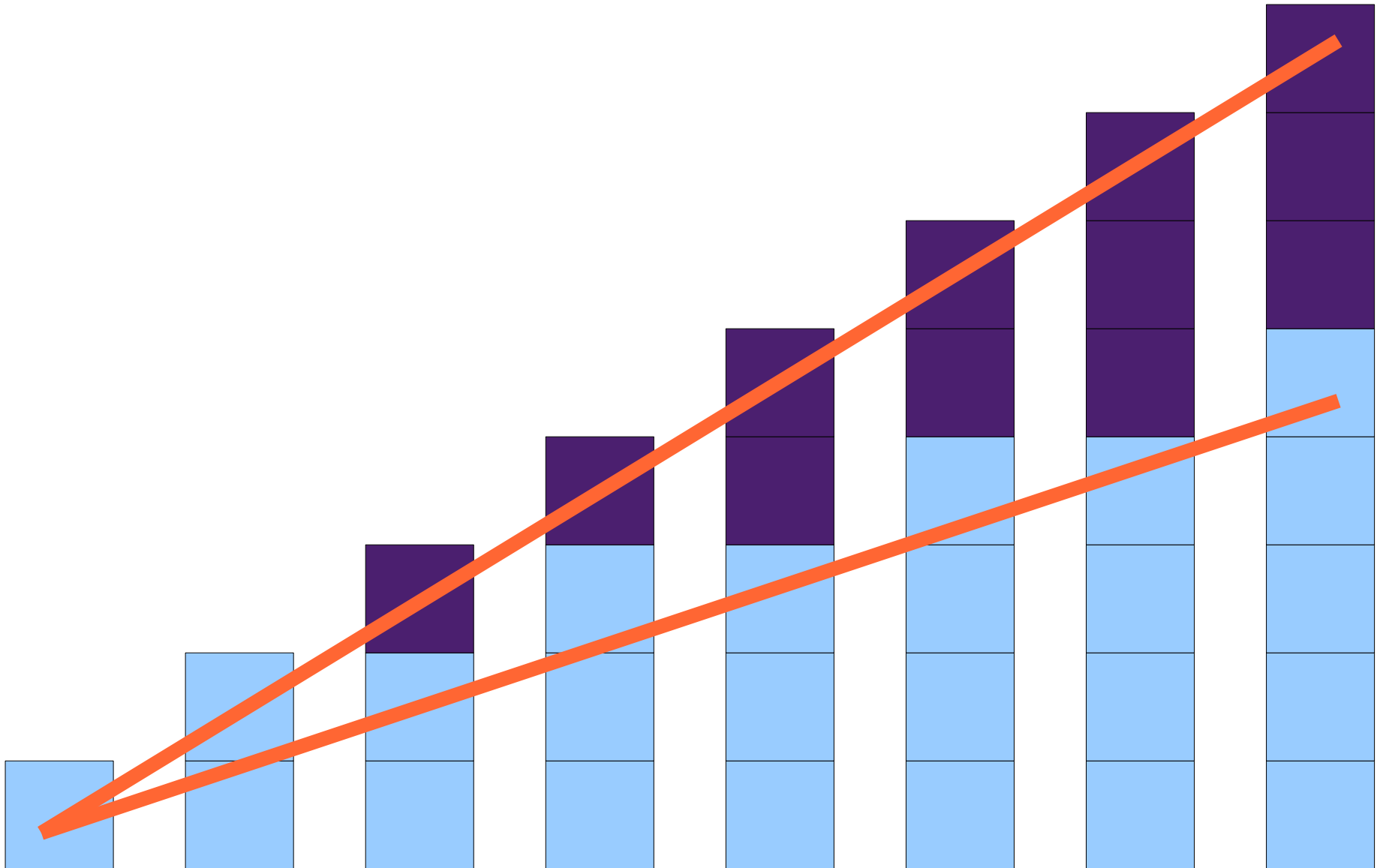
A Different Analysis



A Different Analysis

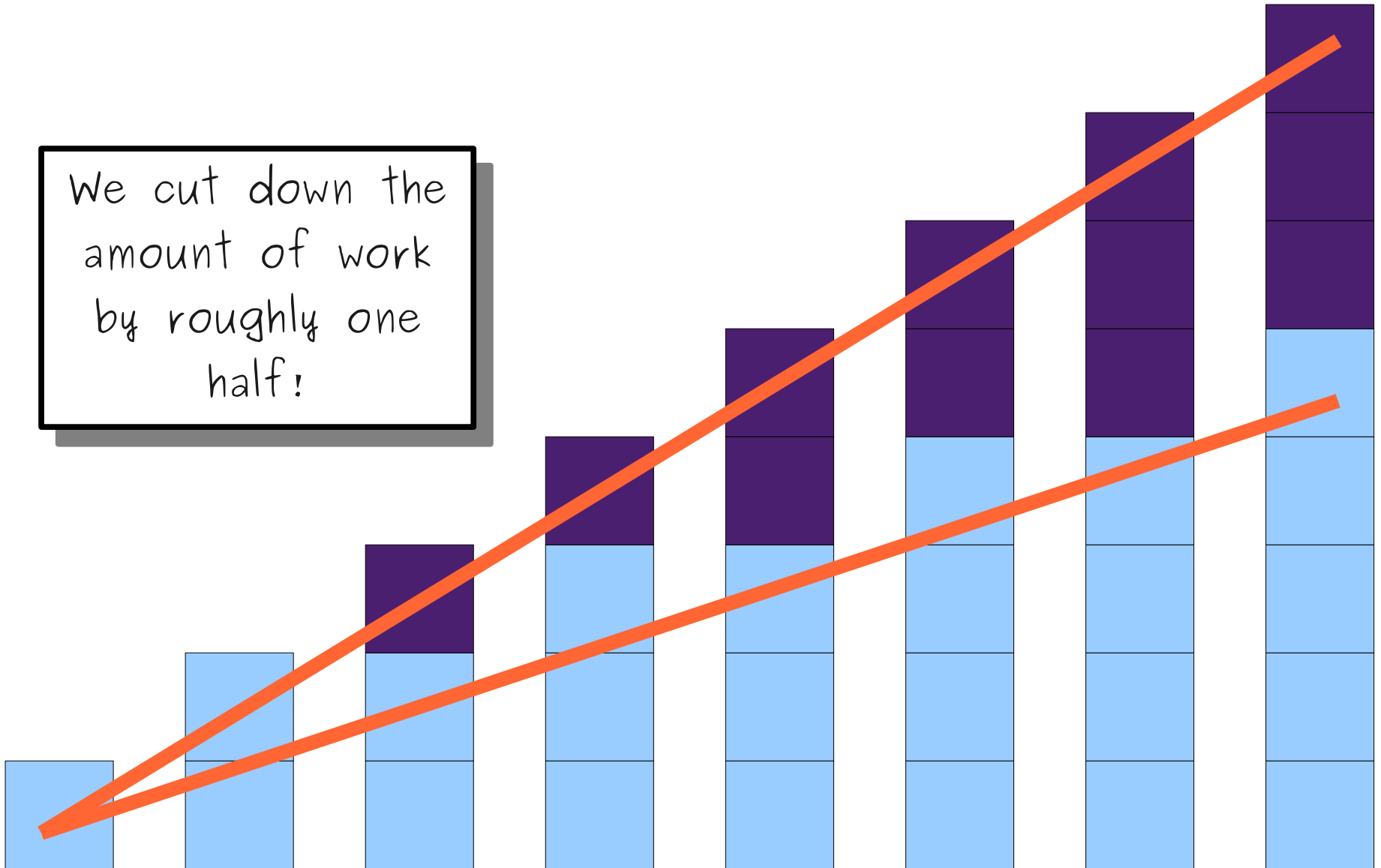


A Different Analysis



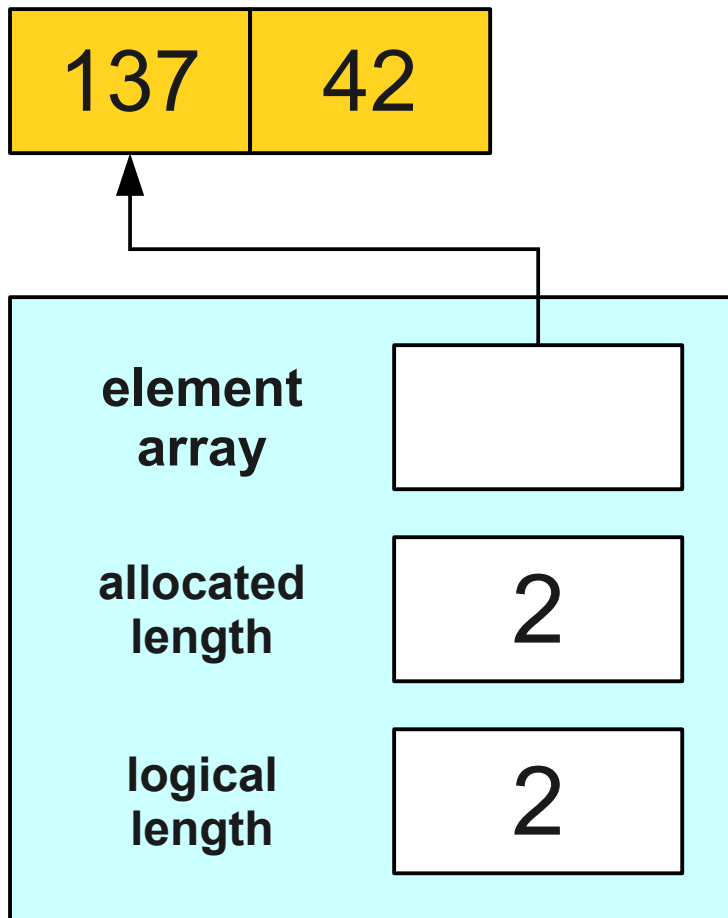
A Different Analysis

We cut down the
amount of work
by roughly one
half!

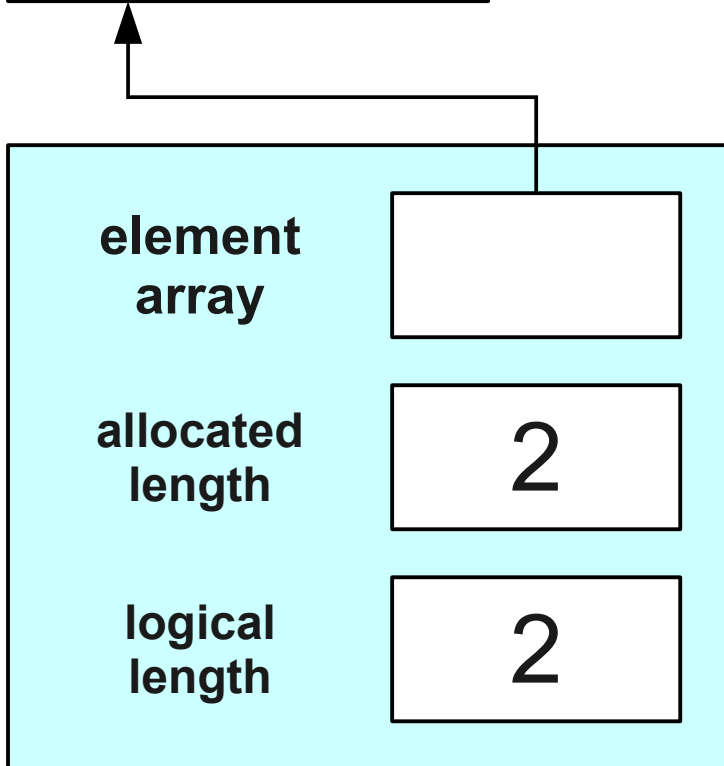
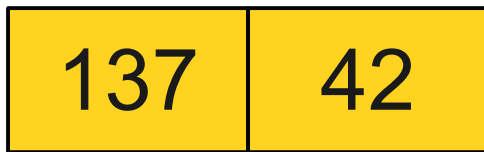
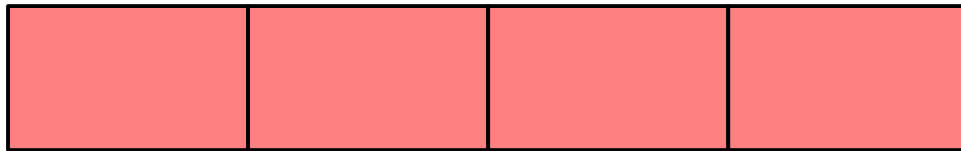


Let's Check it Out!

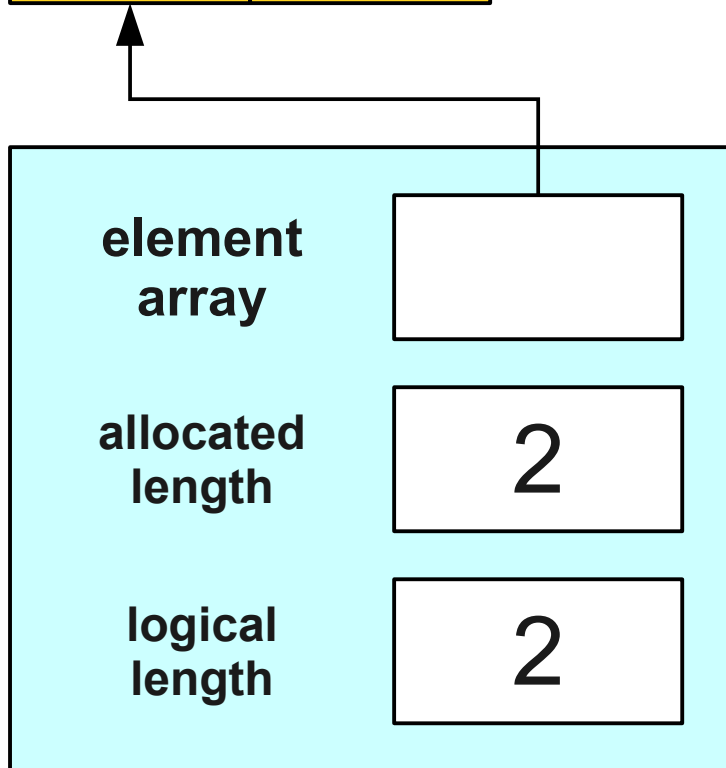
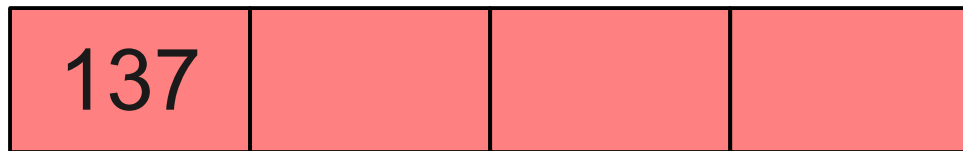
A Much Better Idea



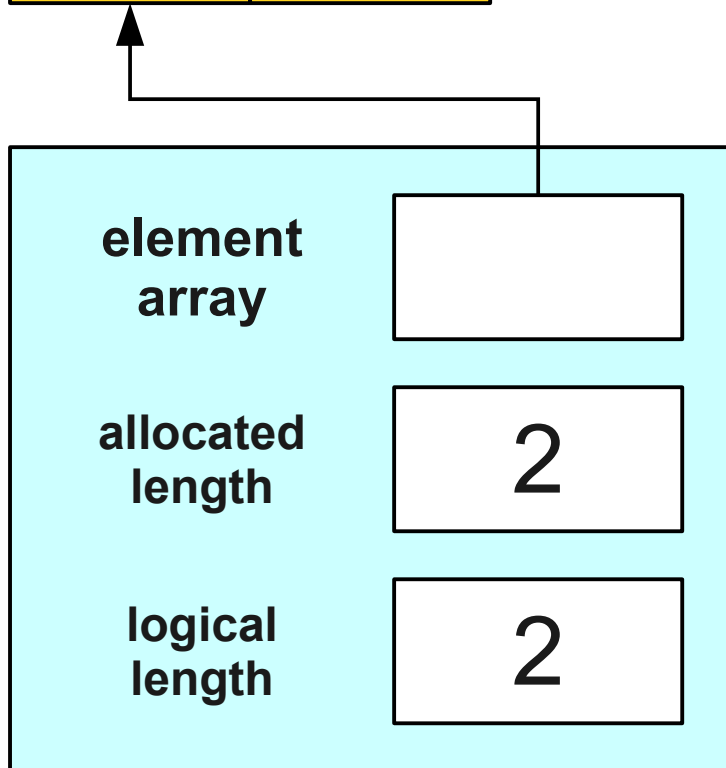
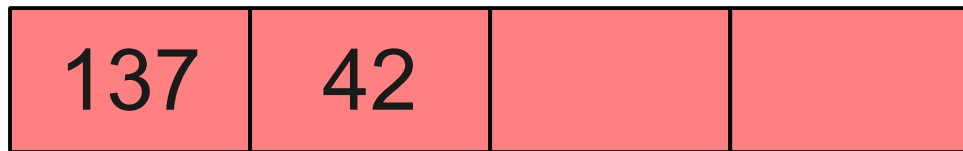
A Much Better Idea



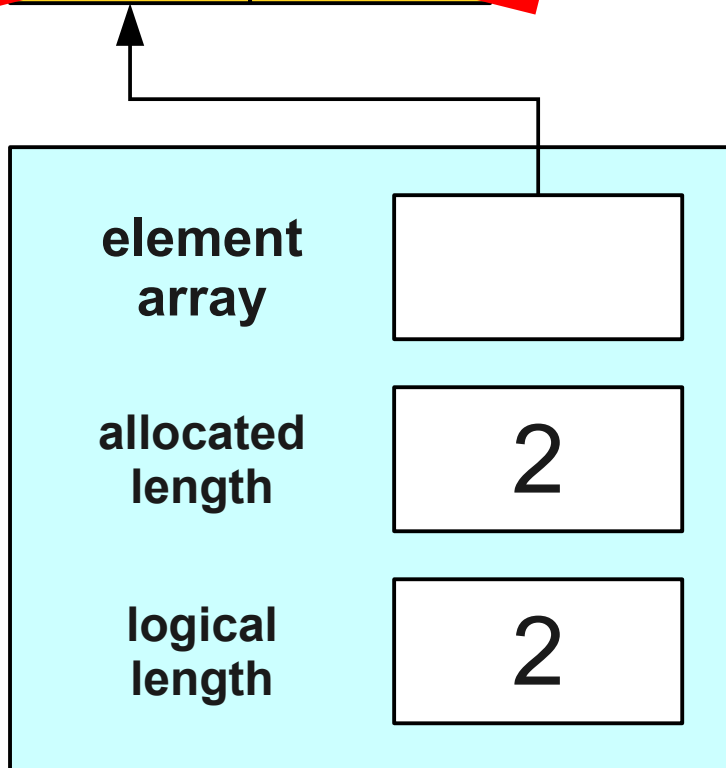
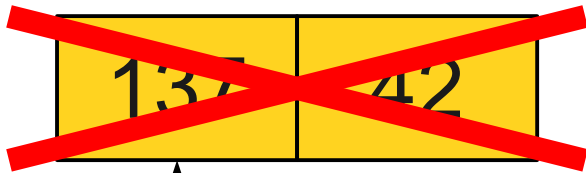
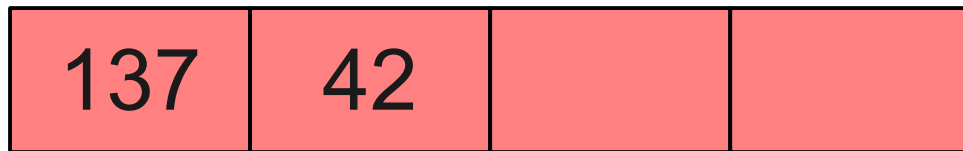
A Much Better Idea



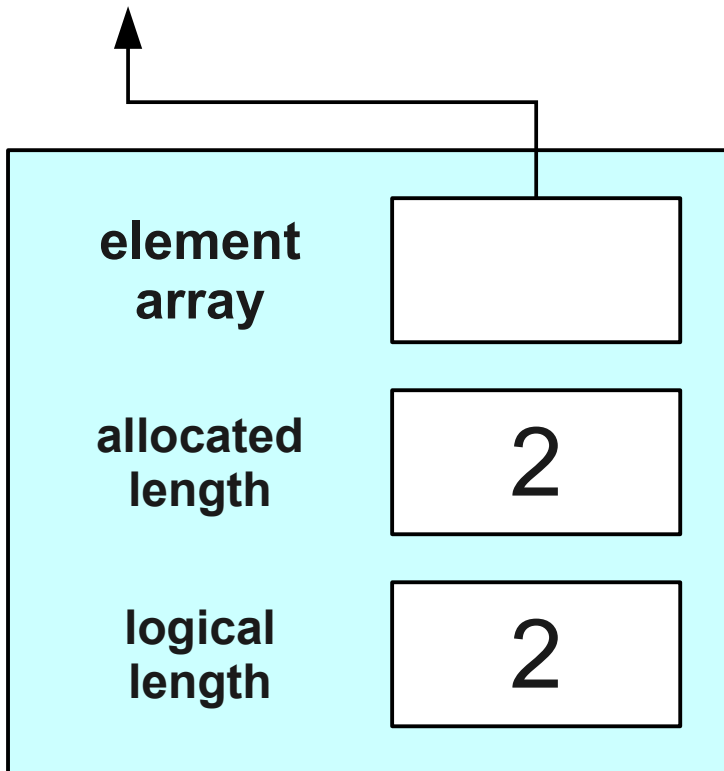
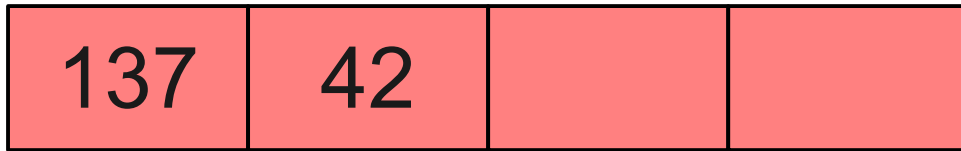
A Much Better Idea



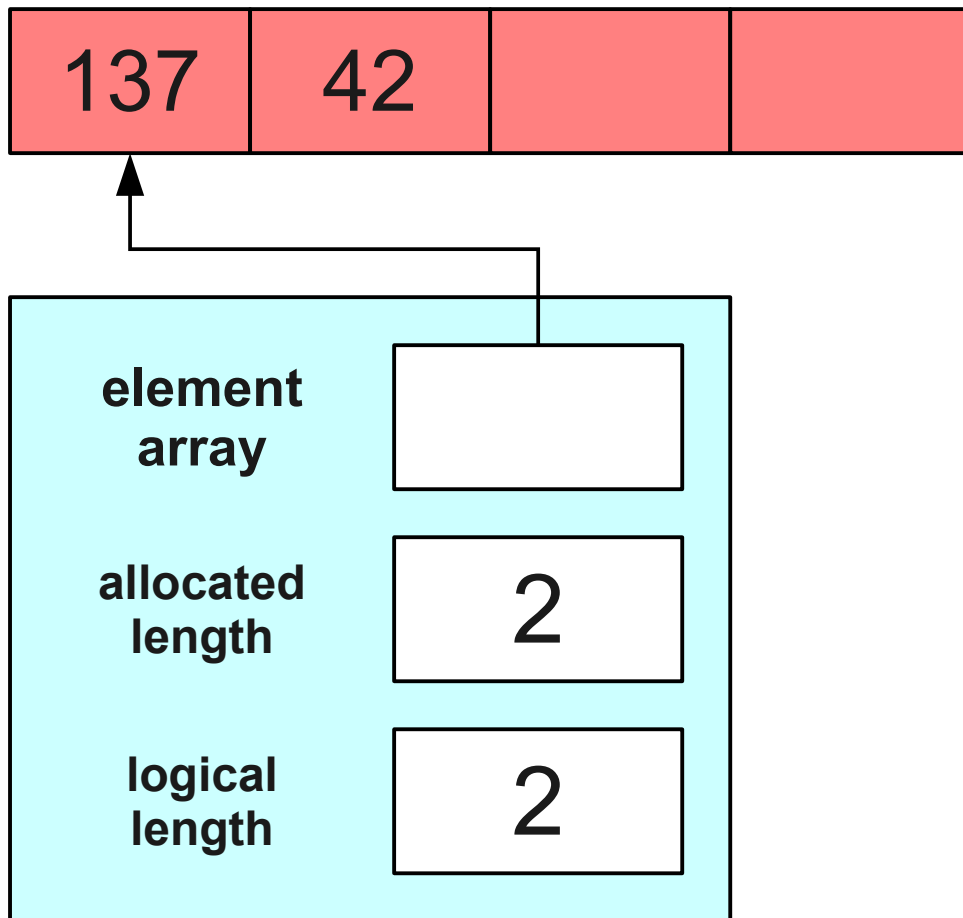
A Much Better Idea



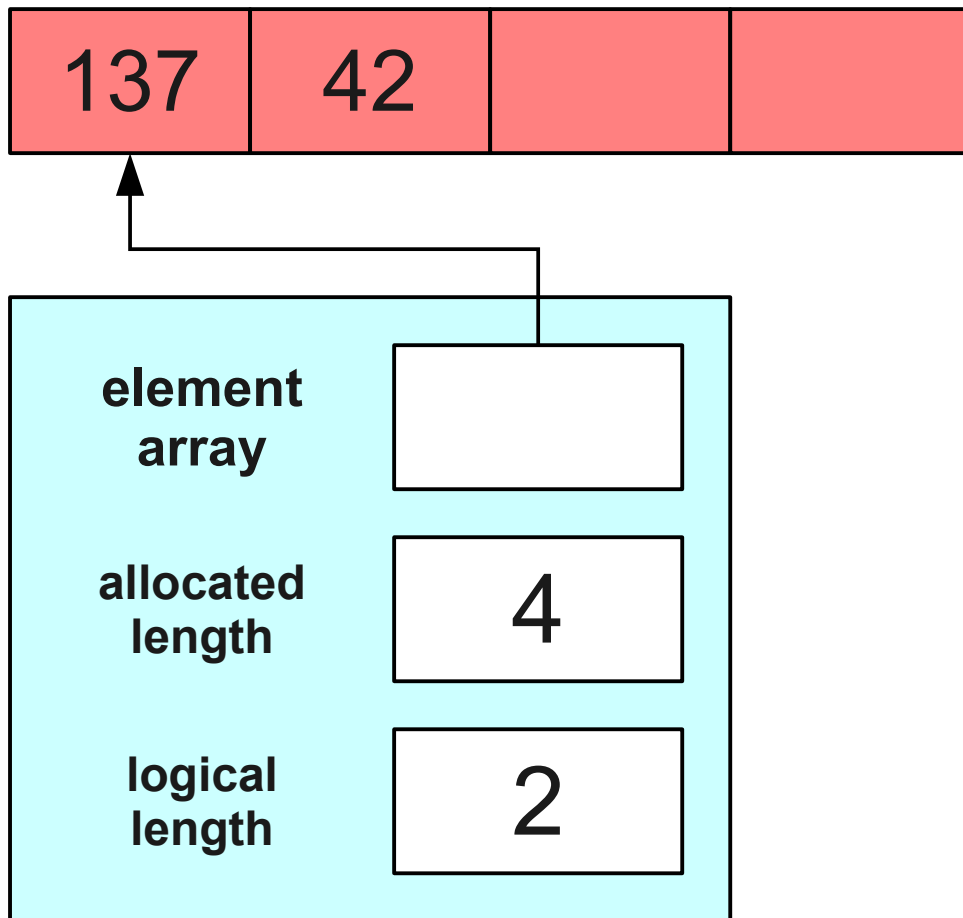
A Much Better Idea



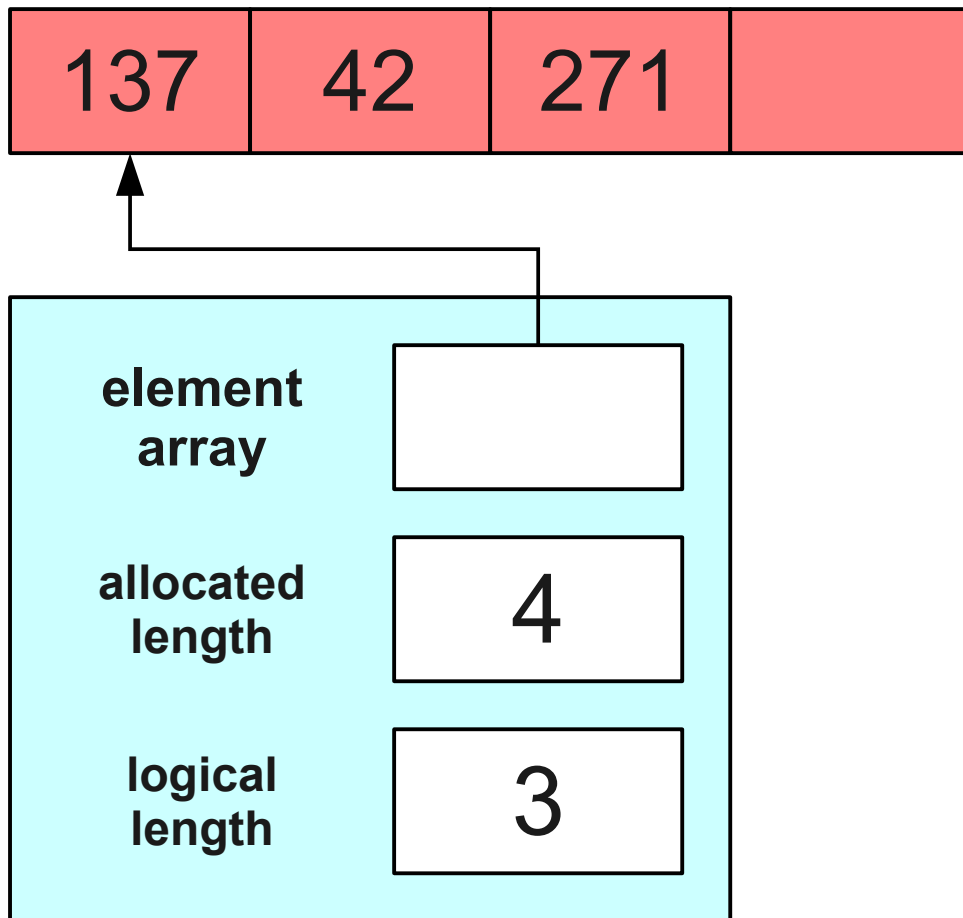
A Much Better Idea



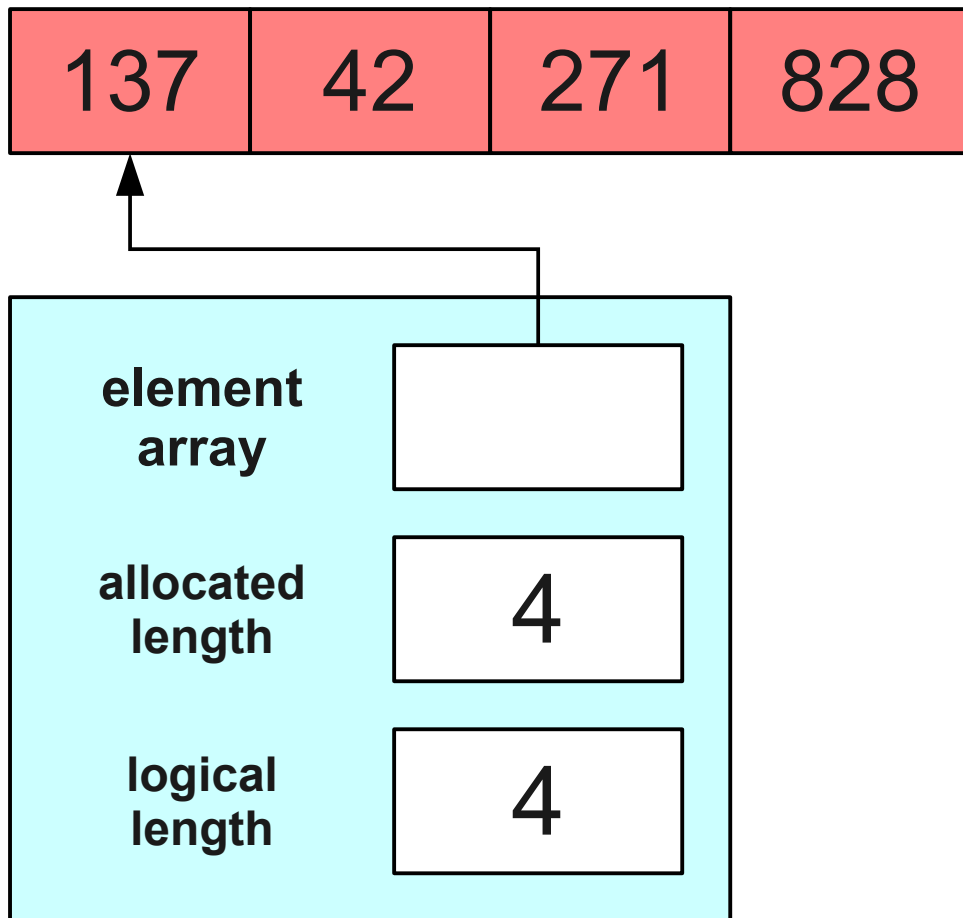
A Much Better Idea



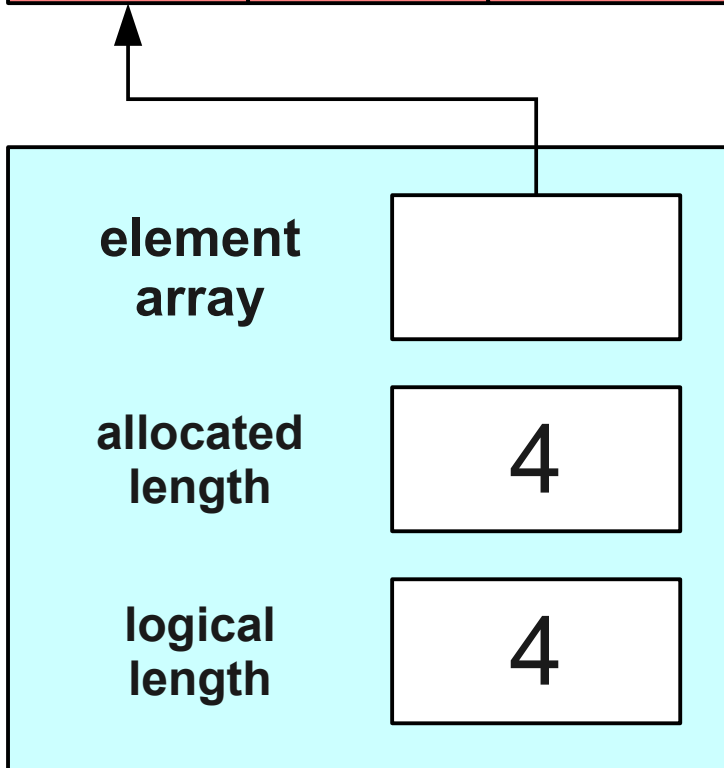
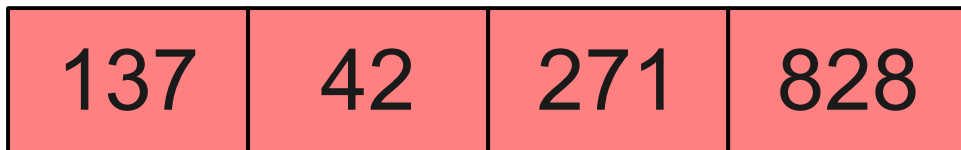
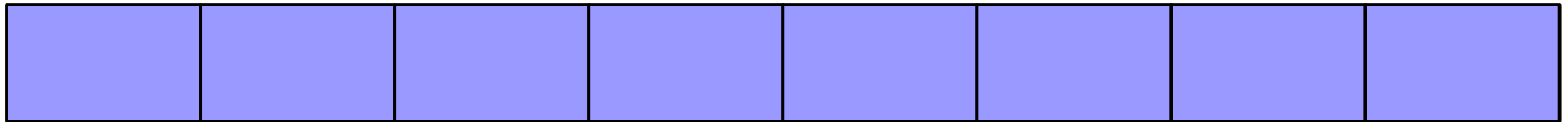
A Much Better Idea



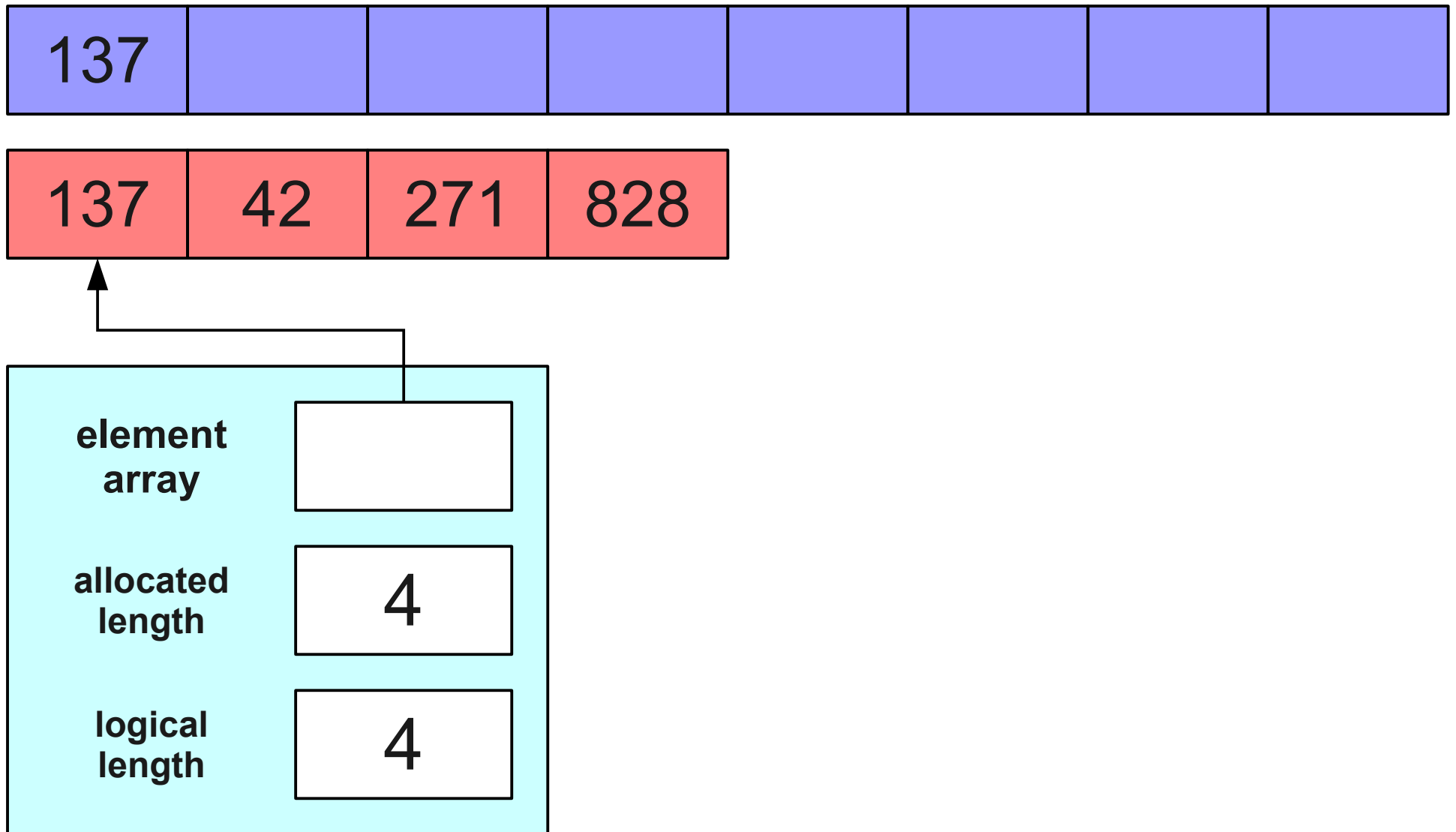
A Much Better Idea



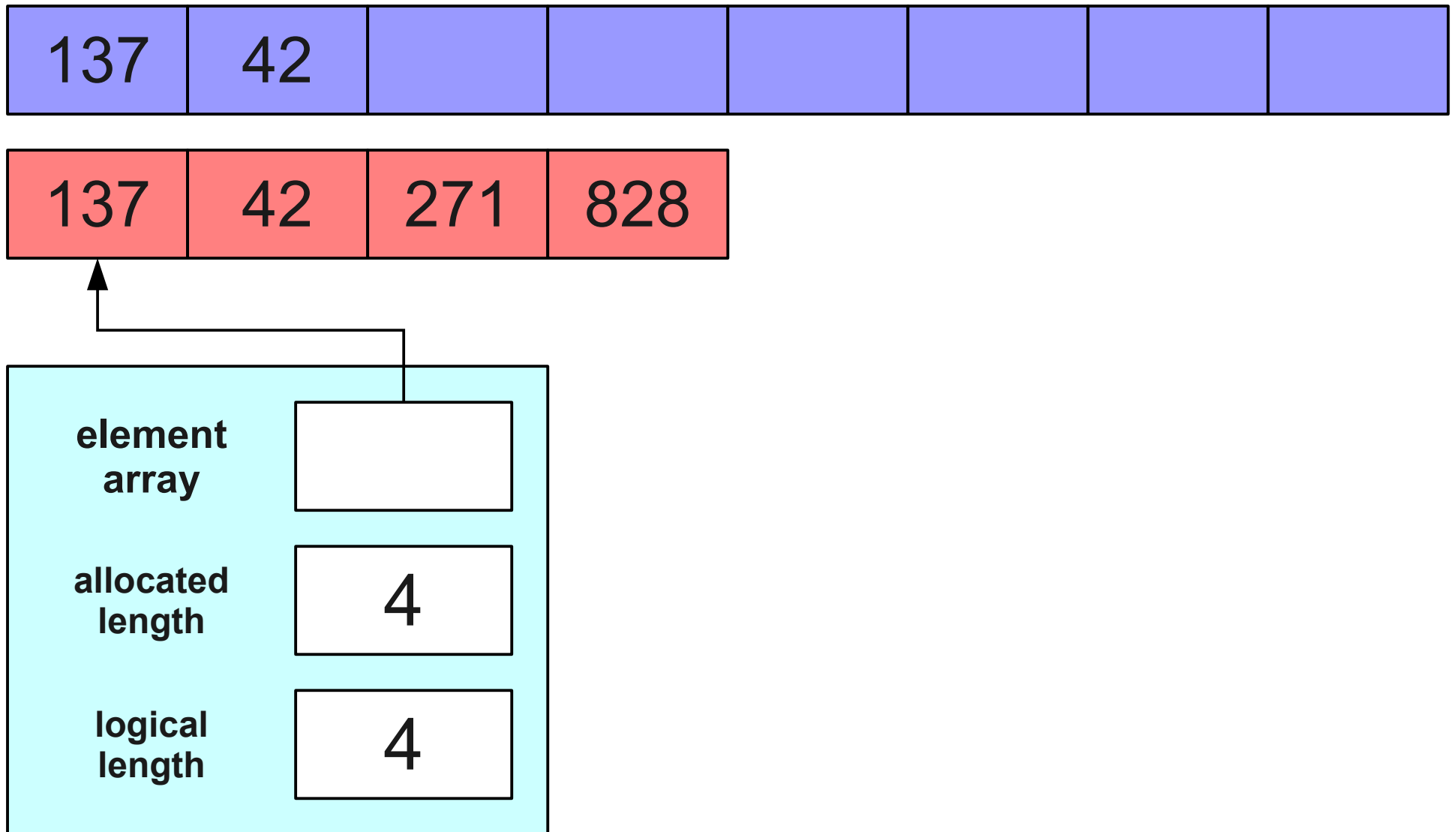
A Much Better Idea



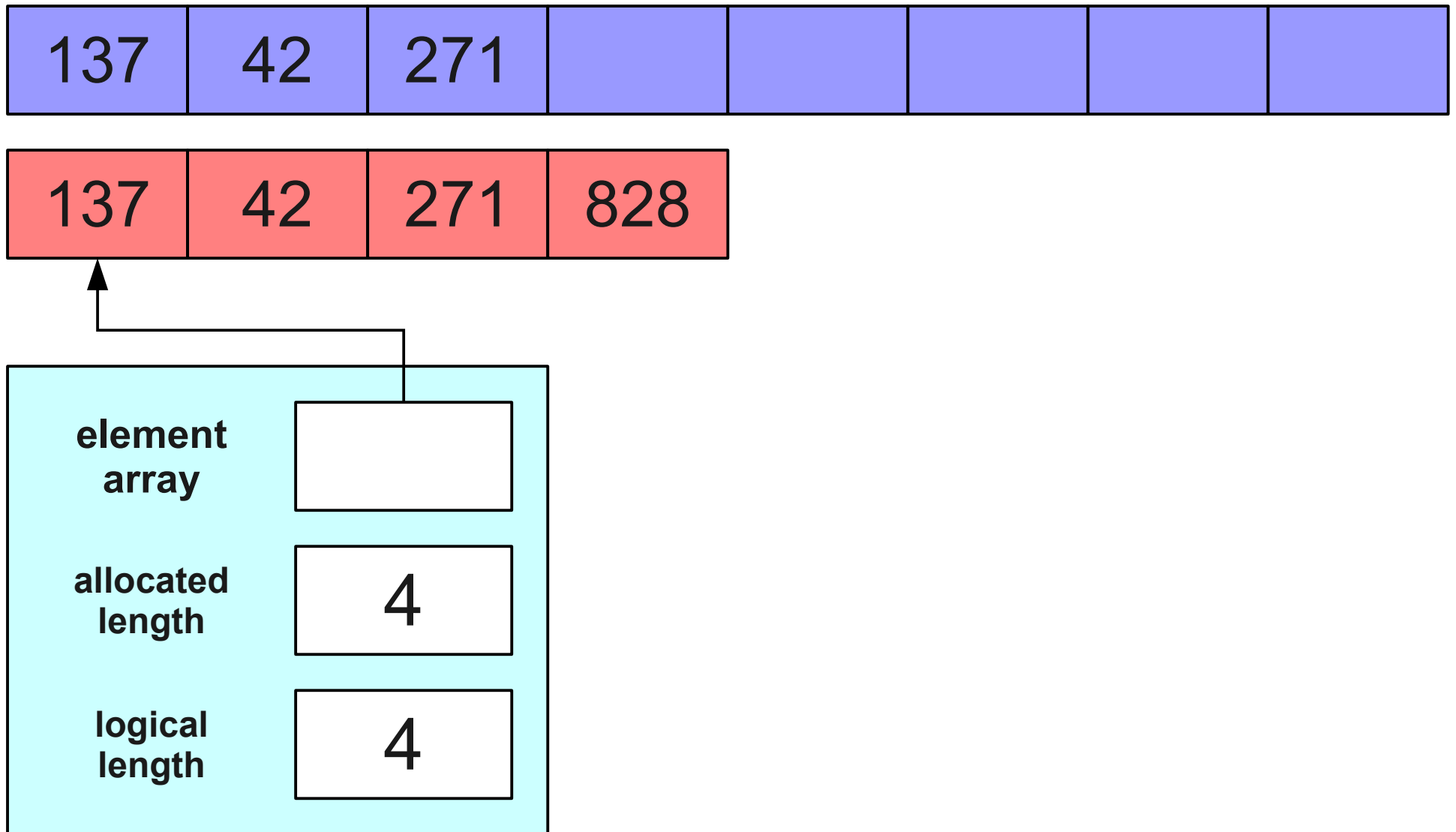
A Much Better Idea



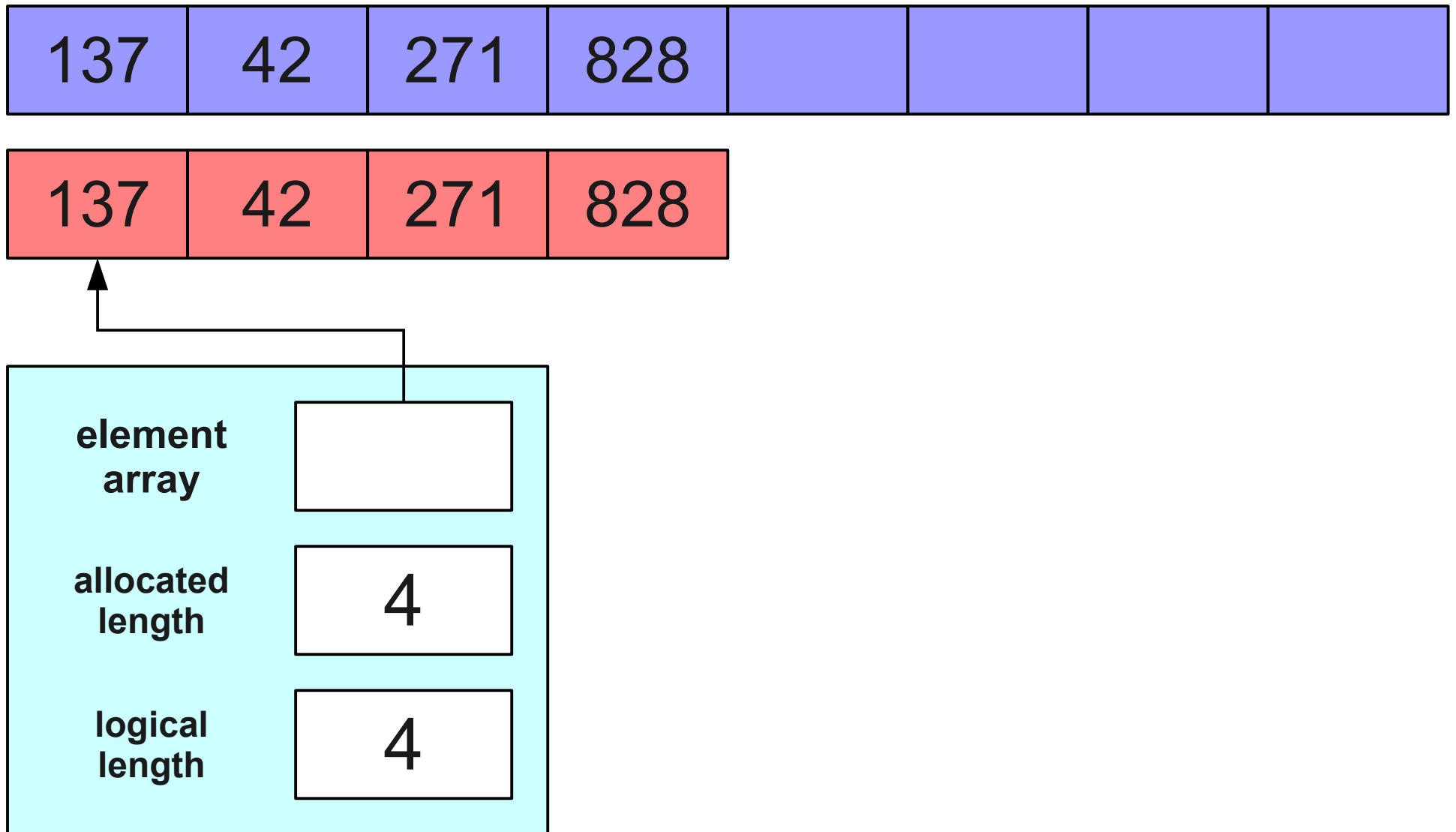
A Much Better Idea



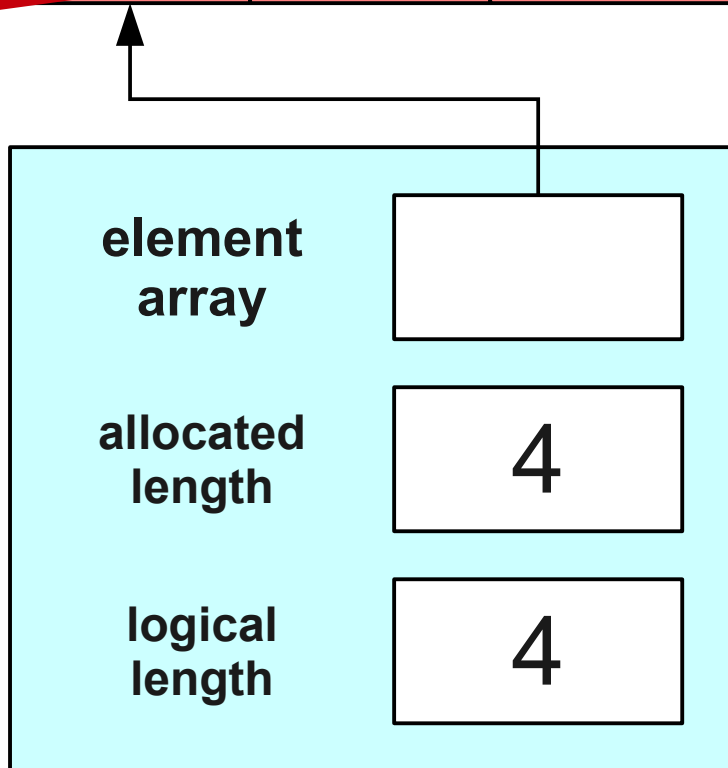
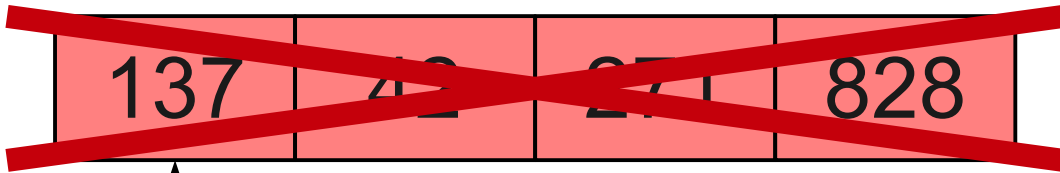
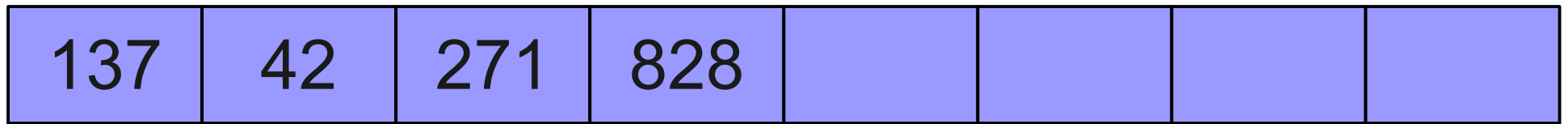
A Much Better Idea



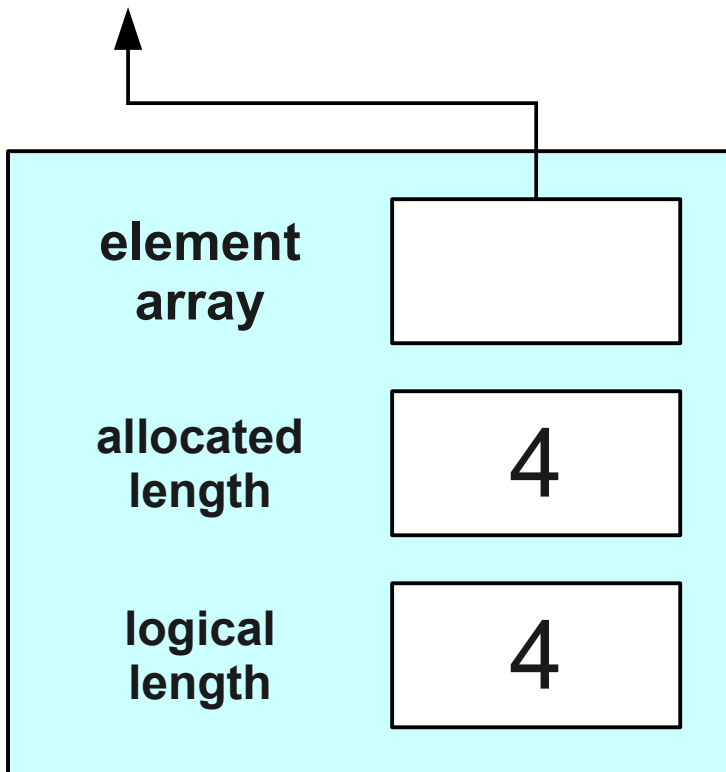
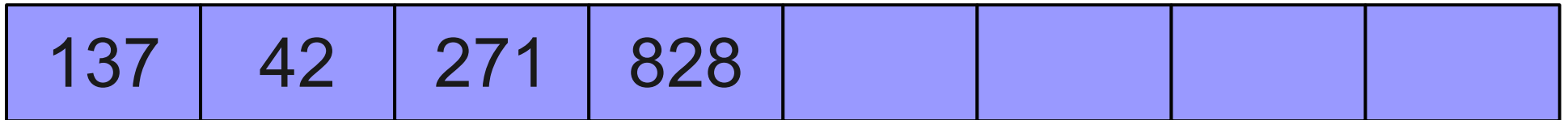
A Much Better Idea



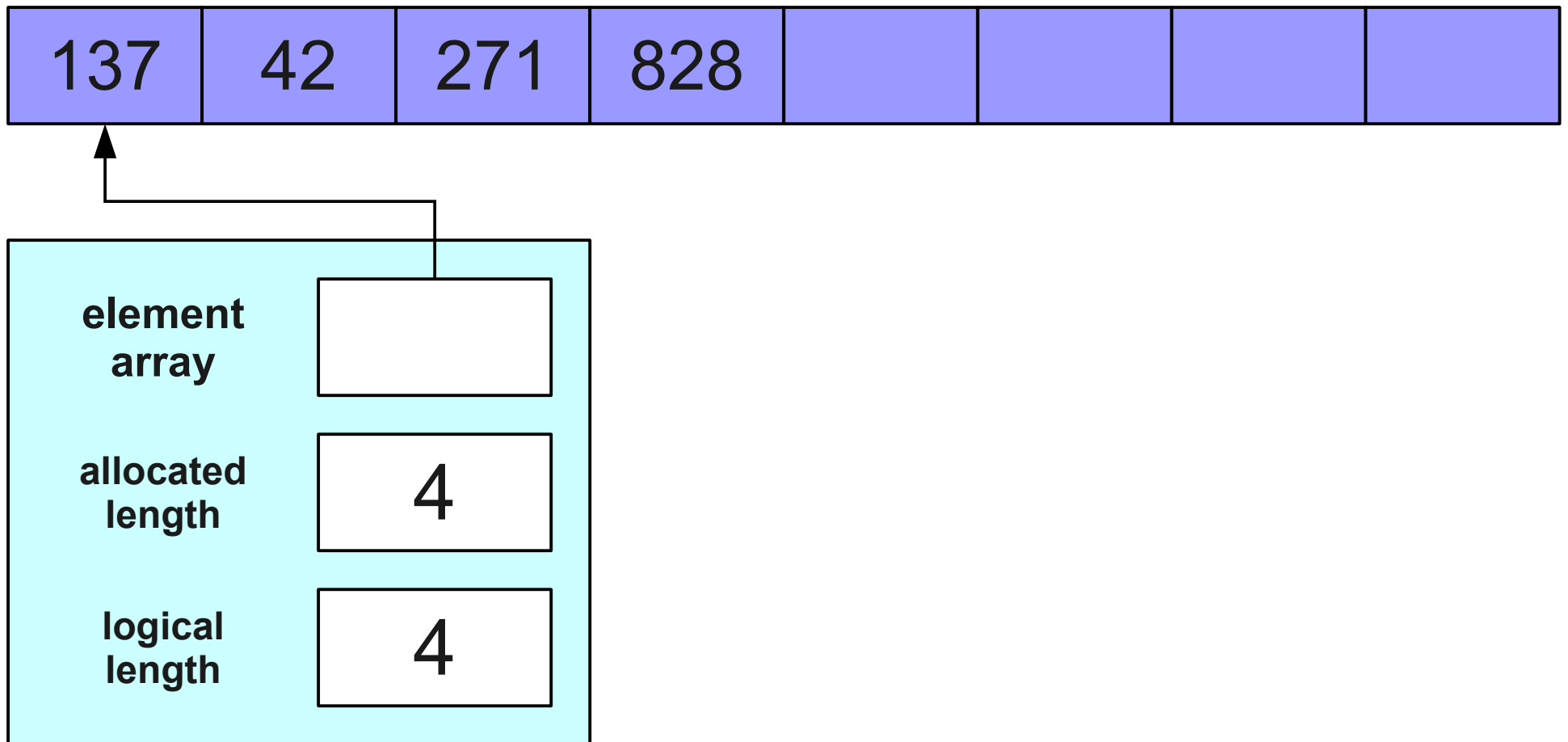
A Much Better Idea



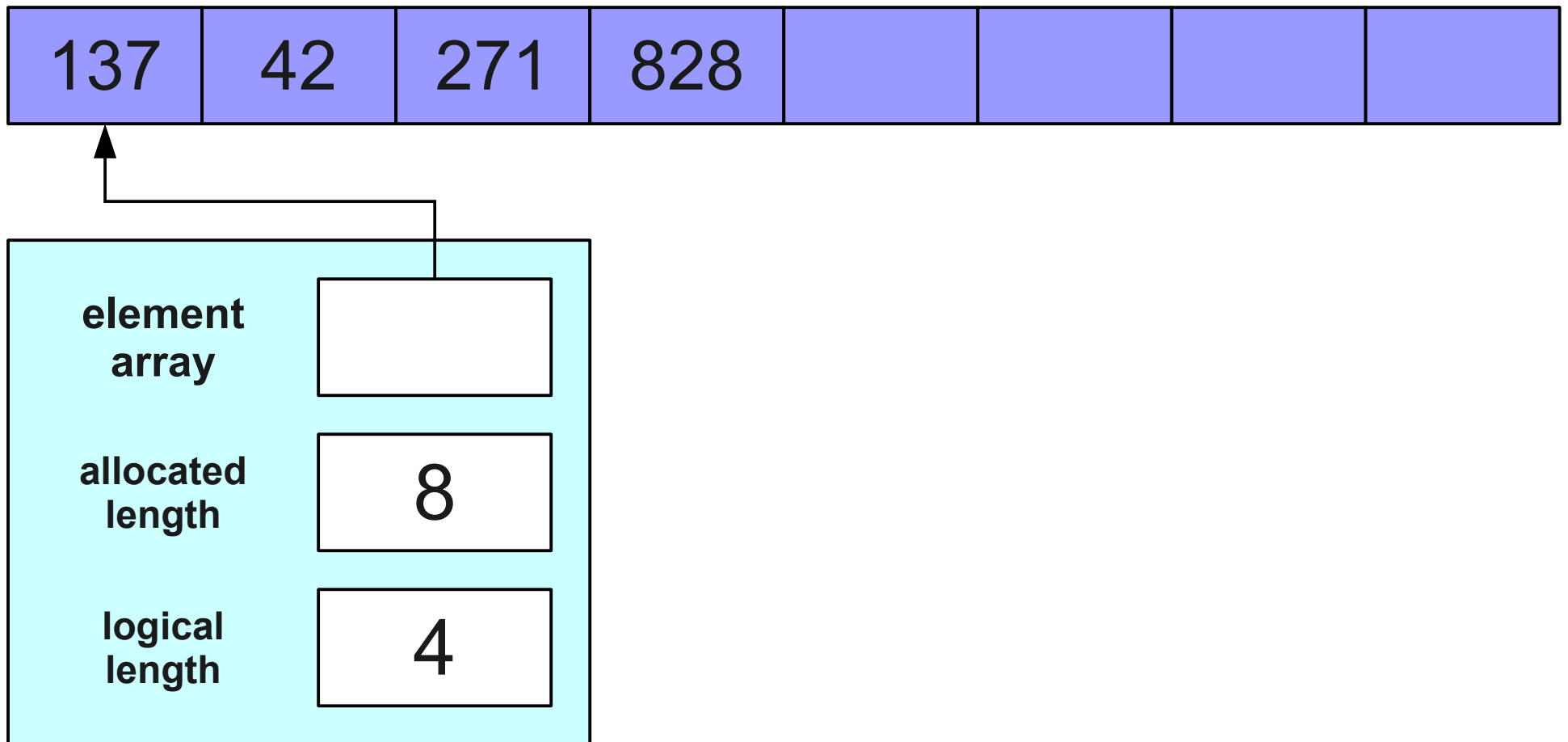
A Much Better Idea



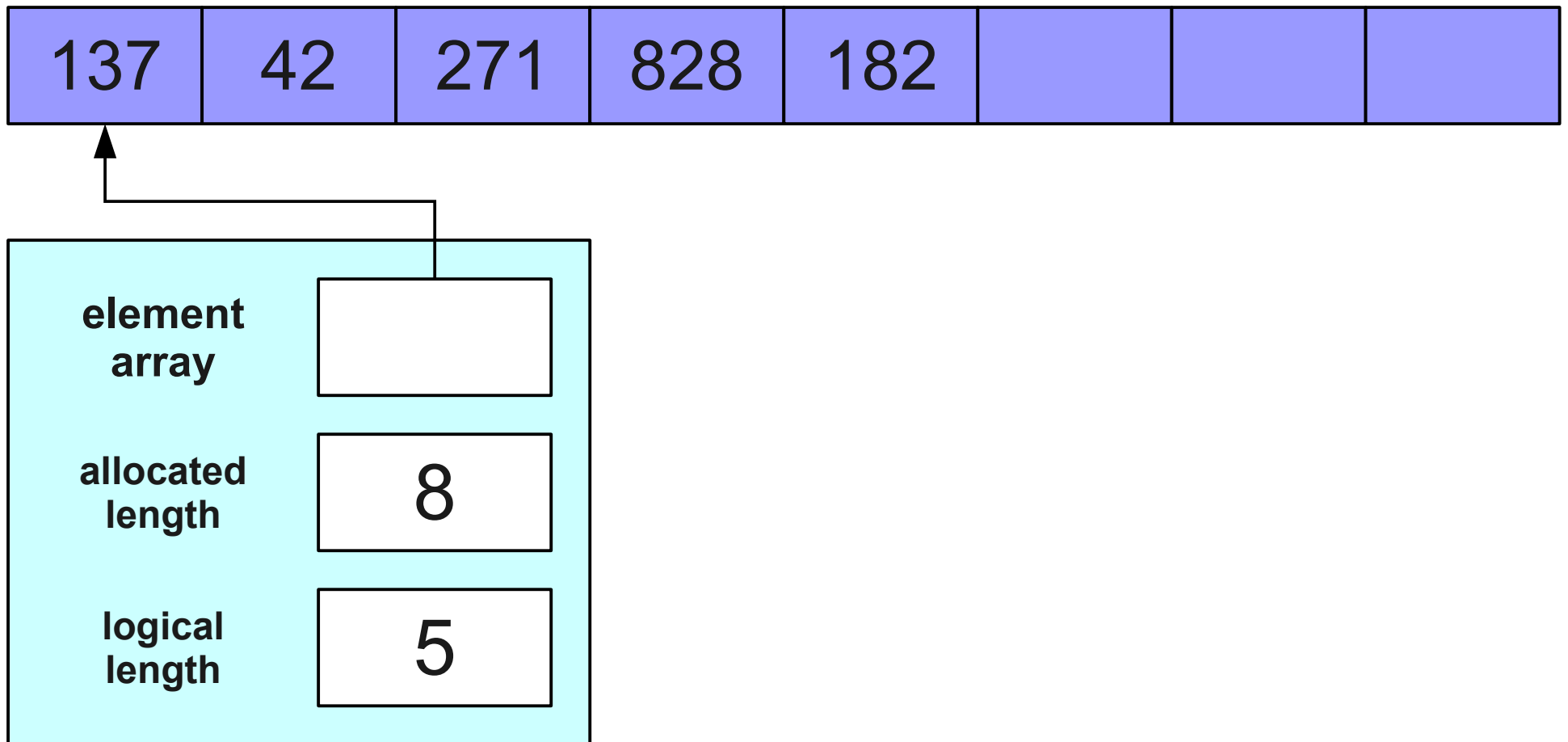
A Much Better Idea



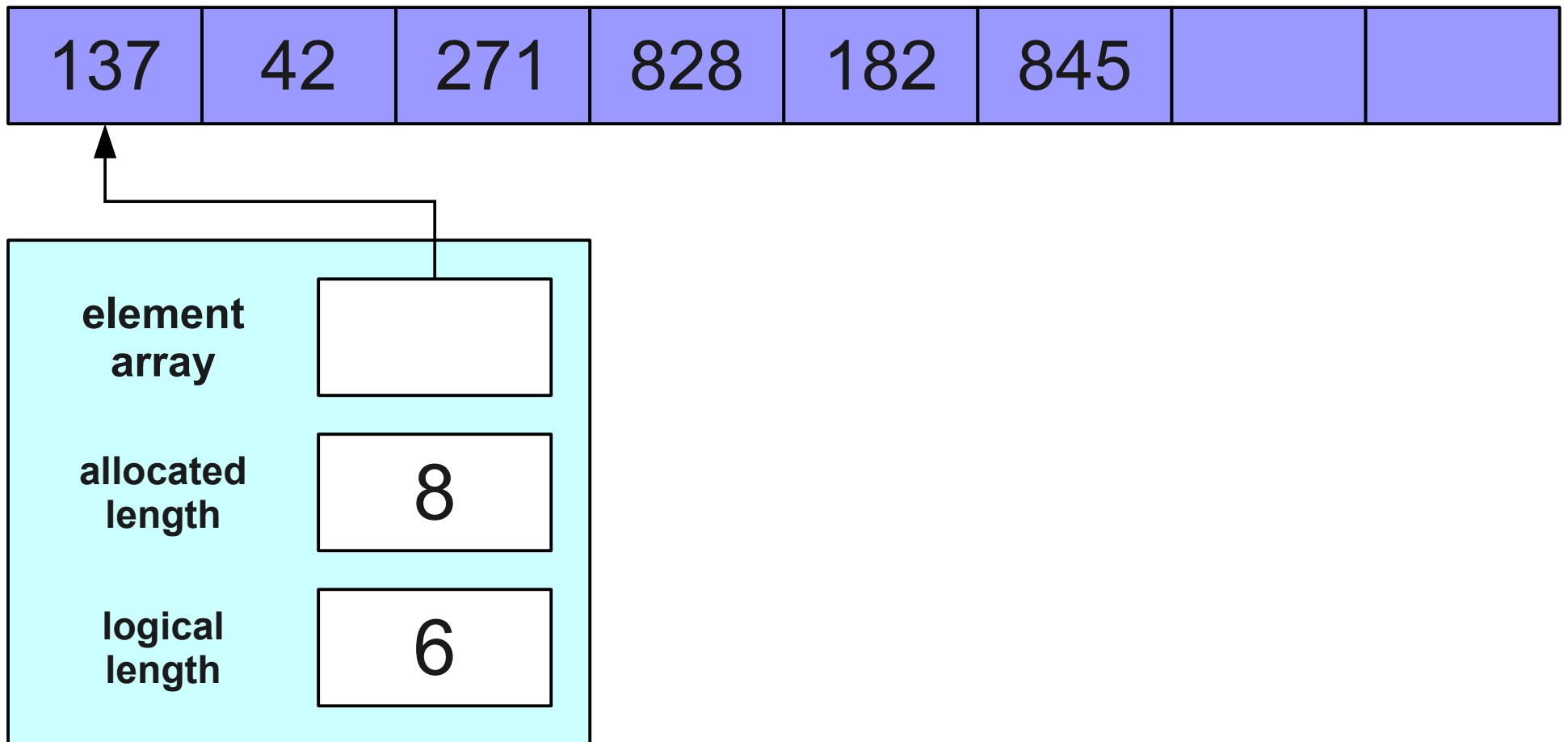
A Much Better Idea



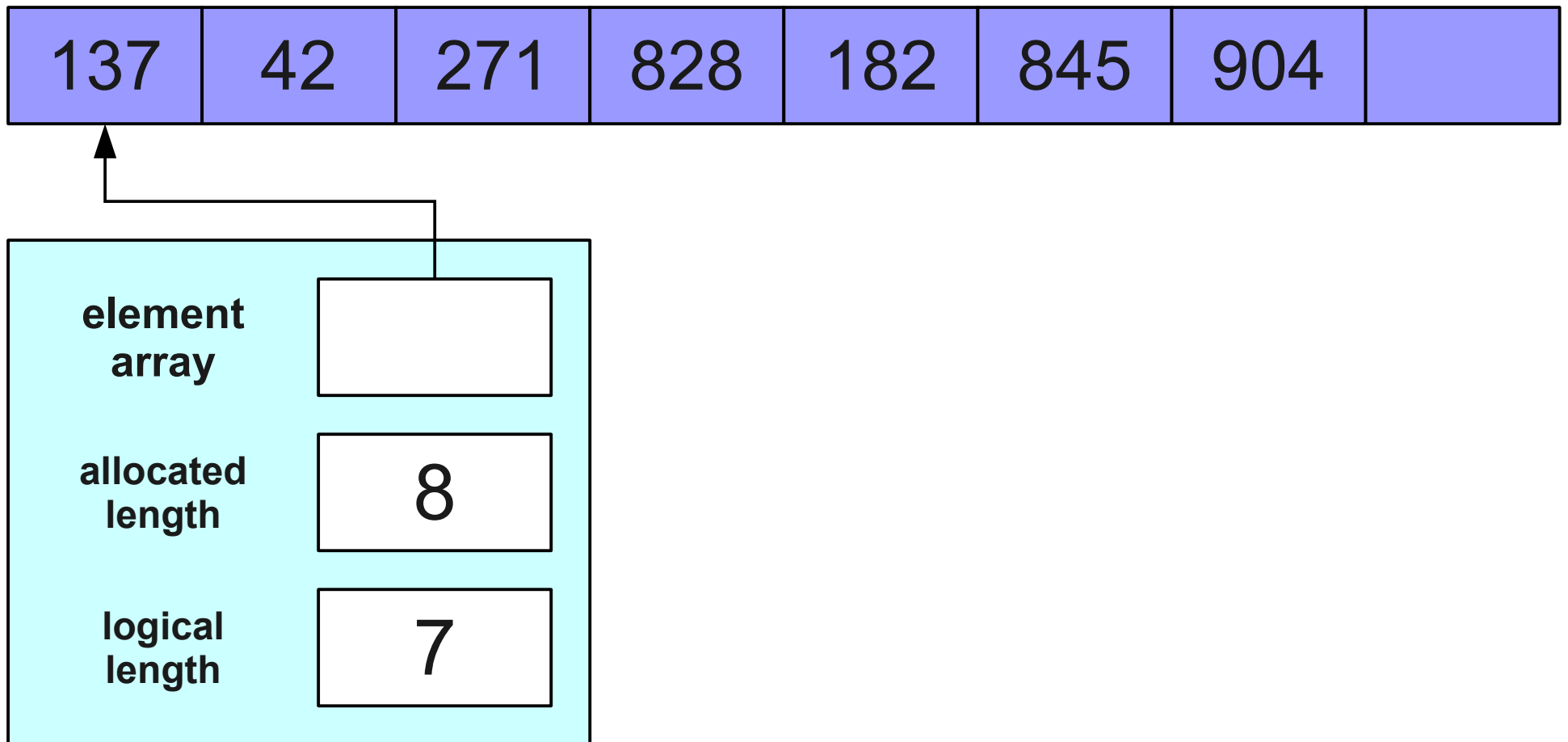
A Much Better Idea



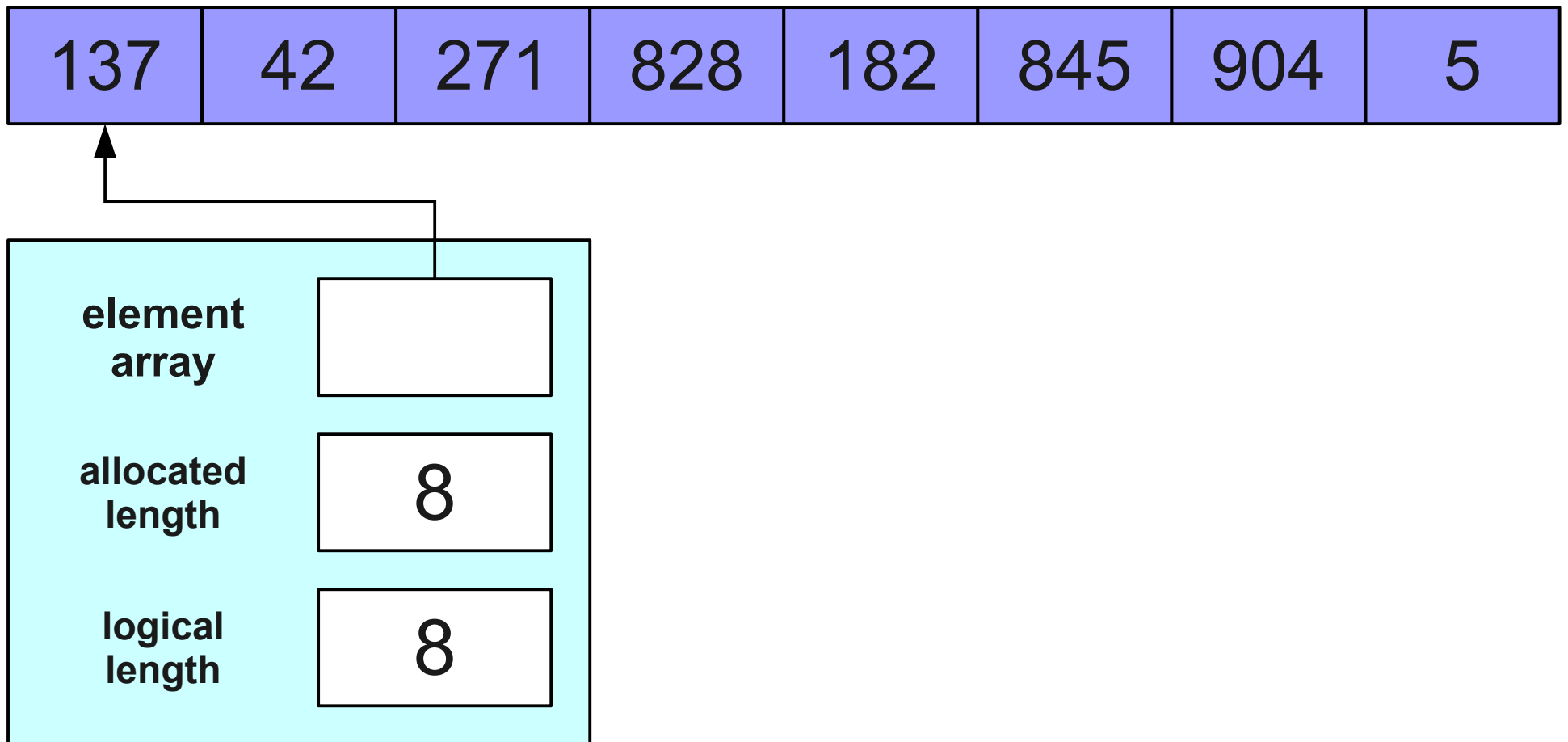
A Much Better Idea



A Much Better Idea



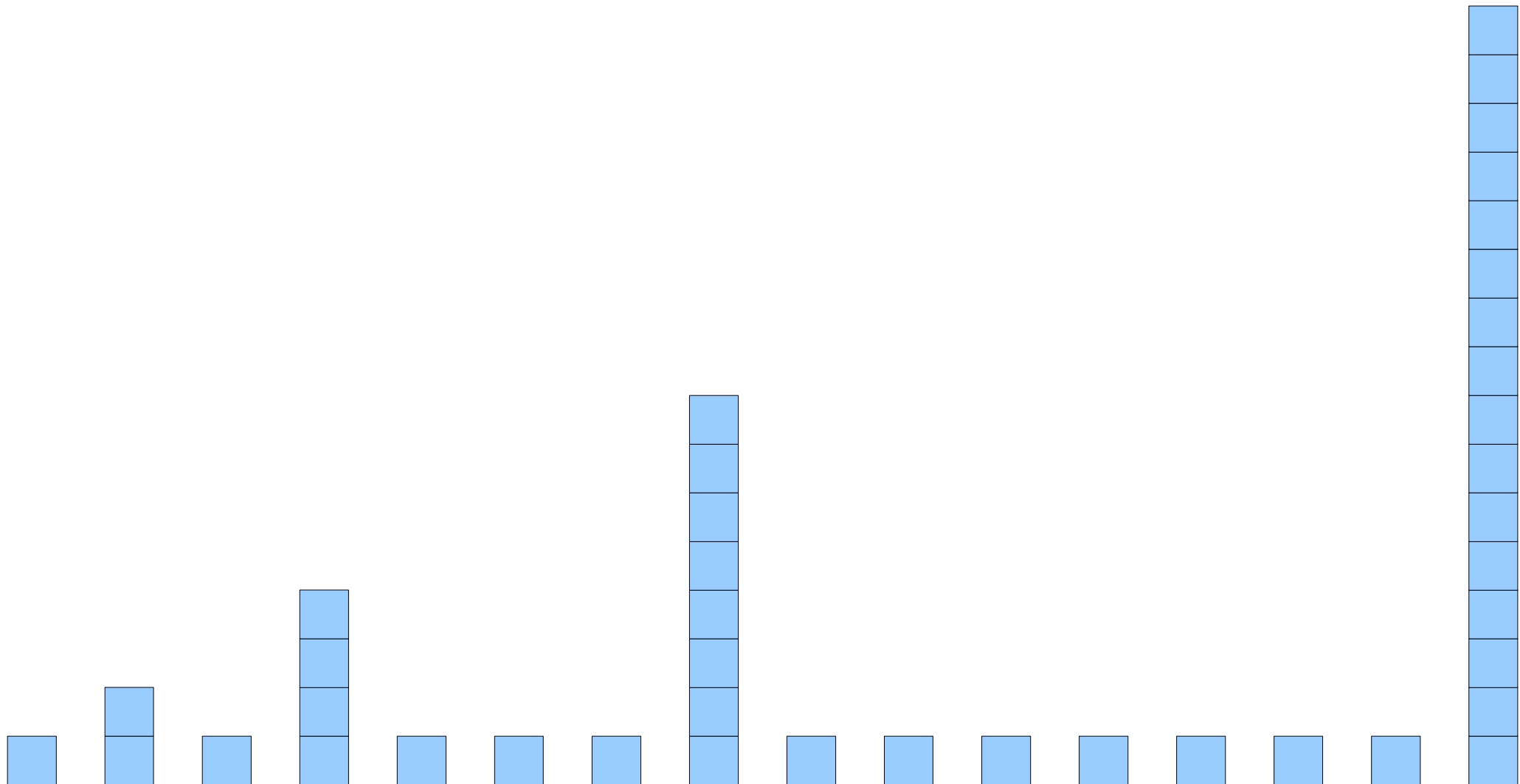
A Much Better Idea



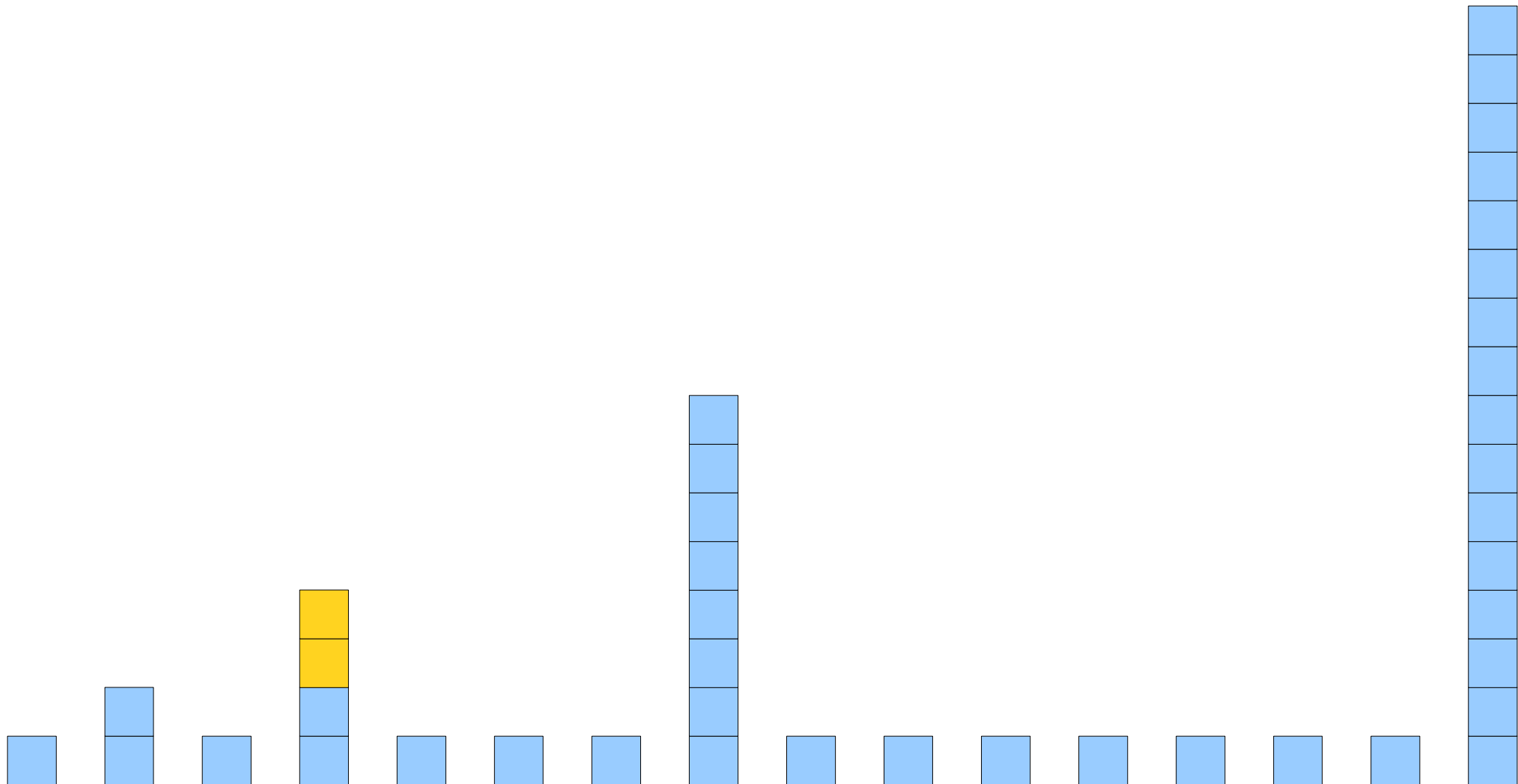
Let's Give it a Try!

How do we analyze this?

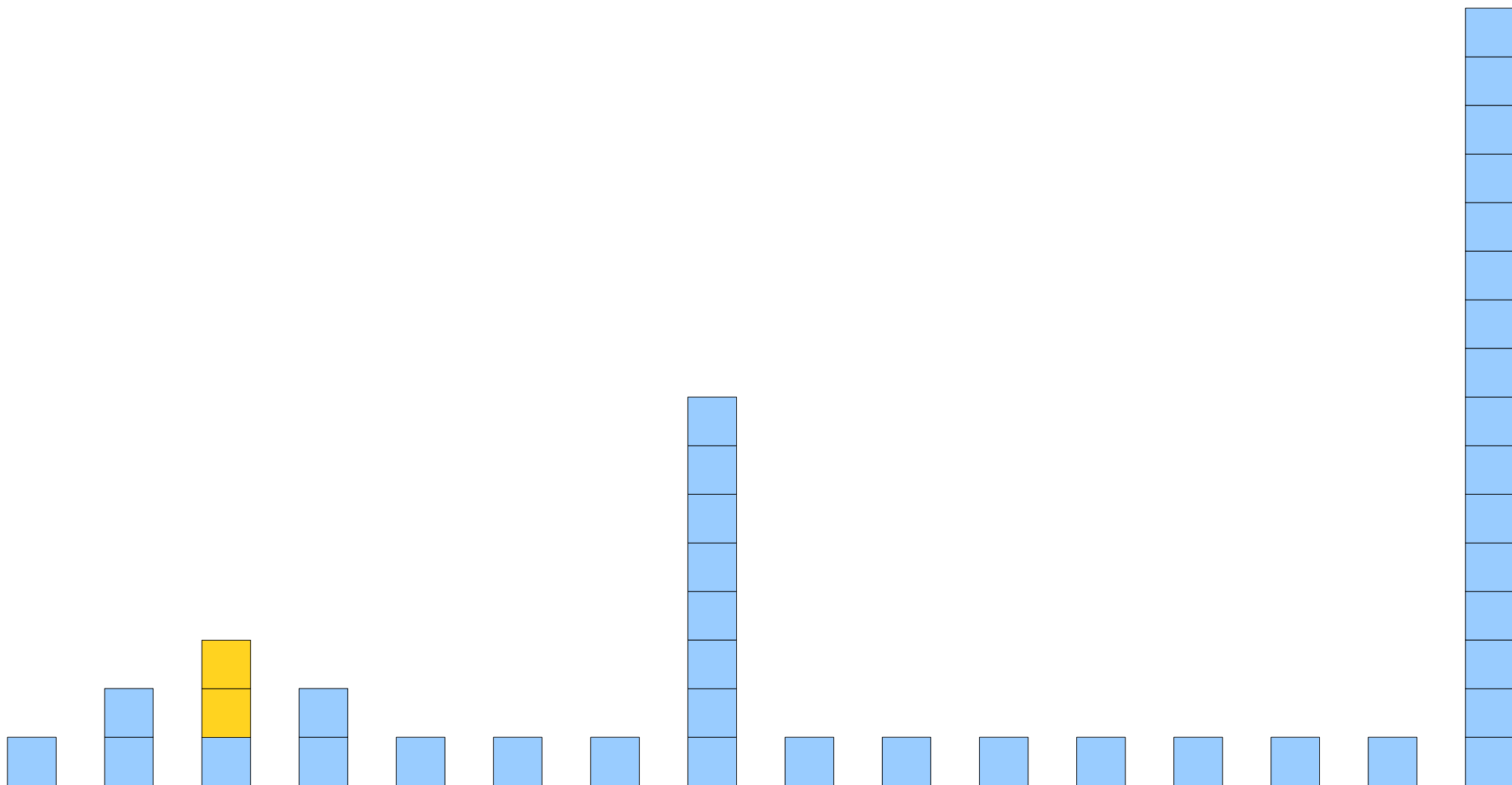
Spreading the Work



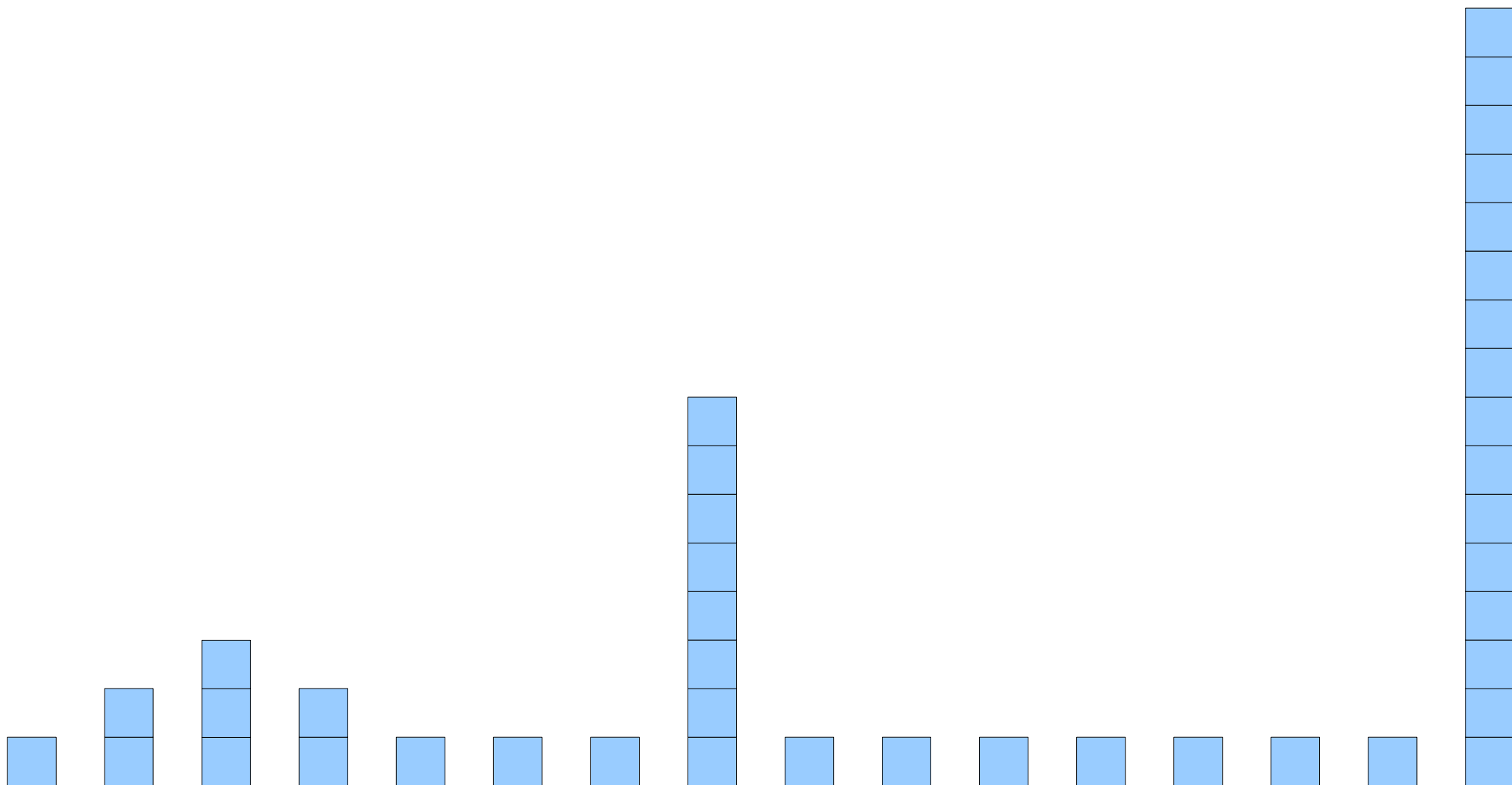
Spreading the Work



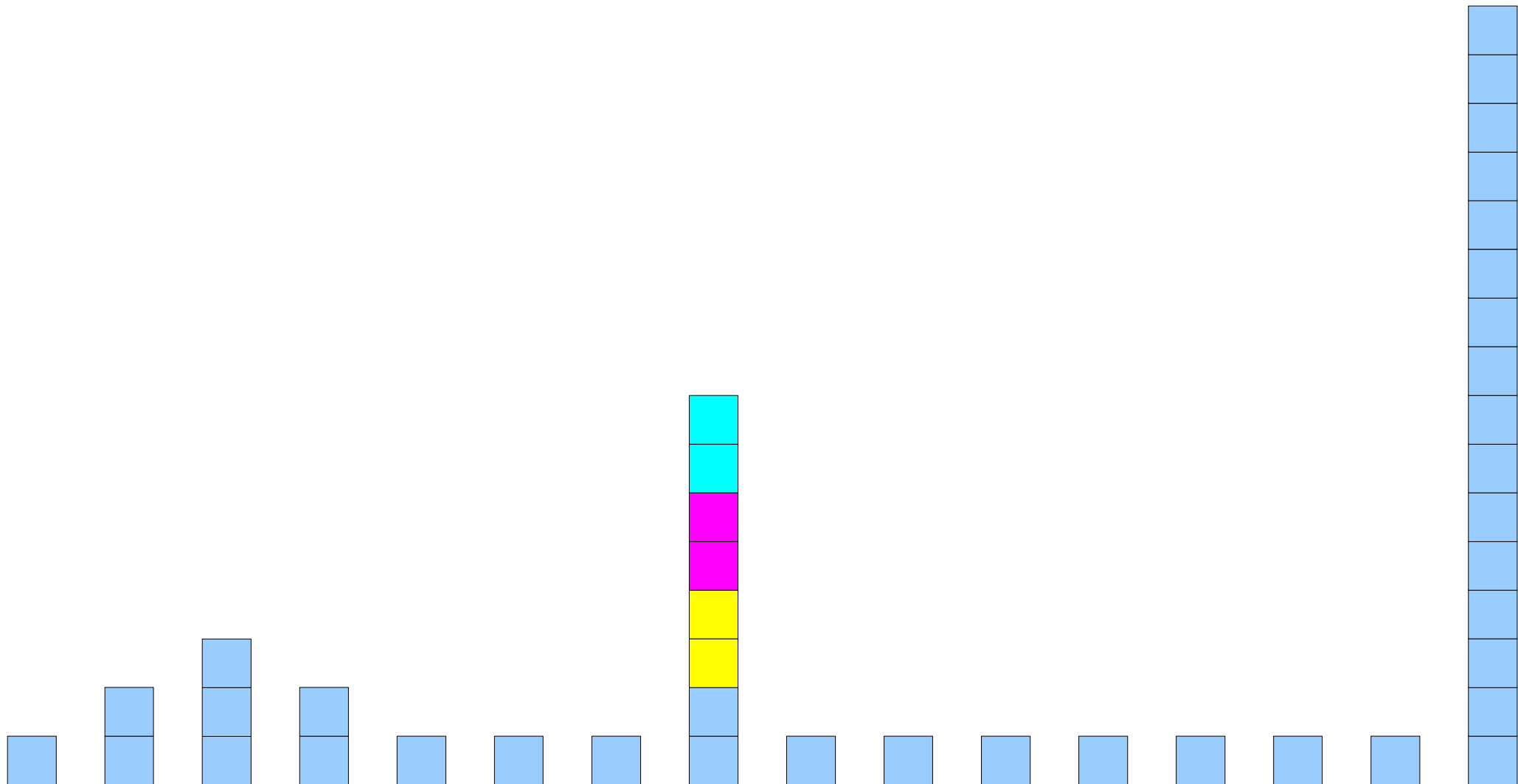
Spreading the Work



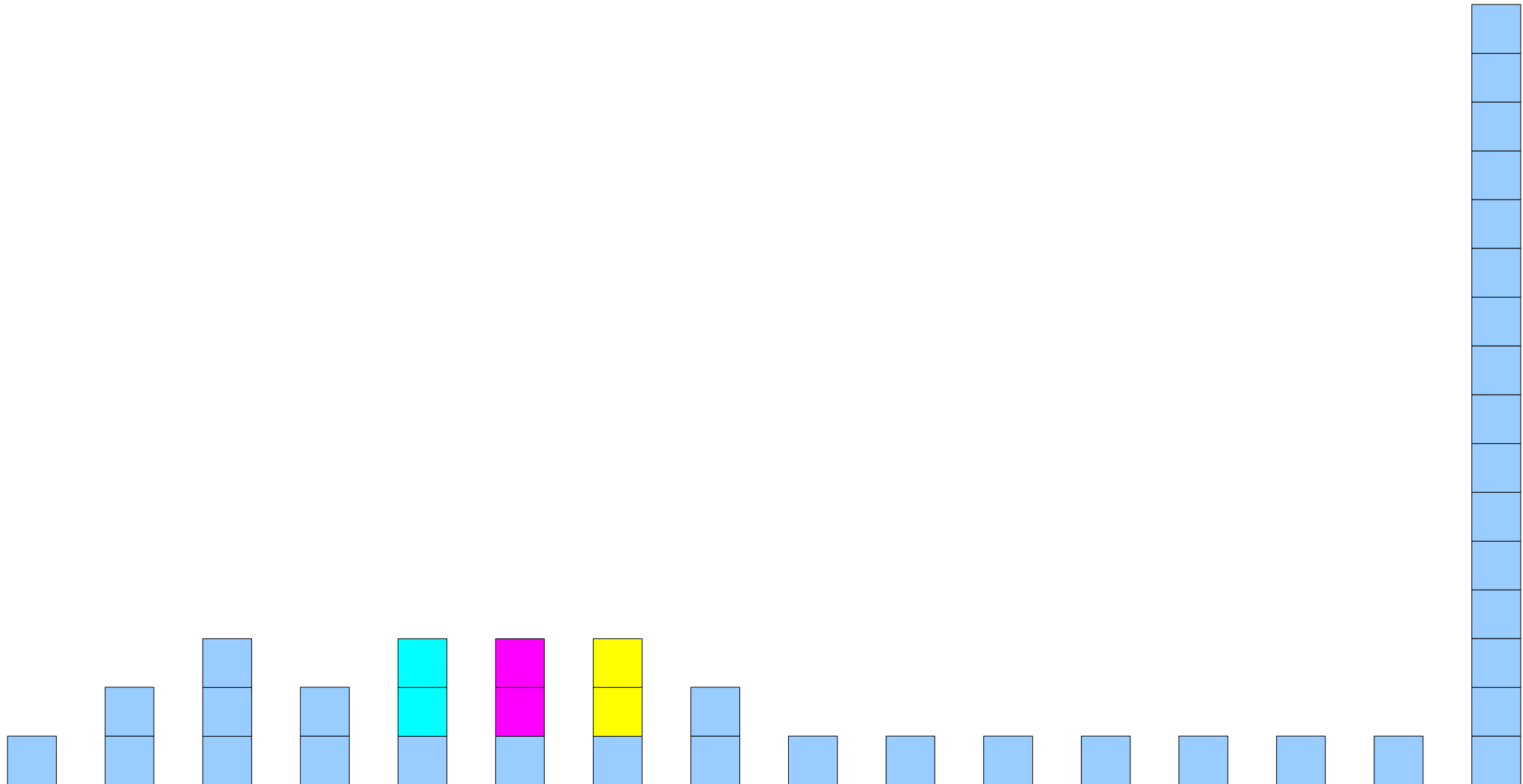
Spreading the Work



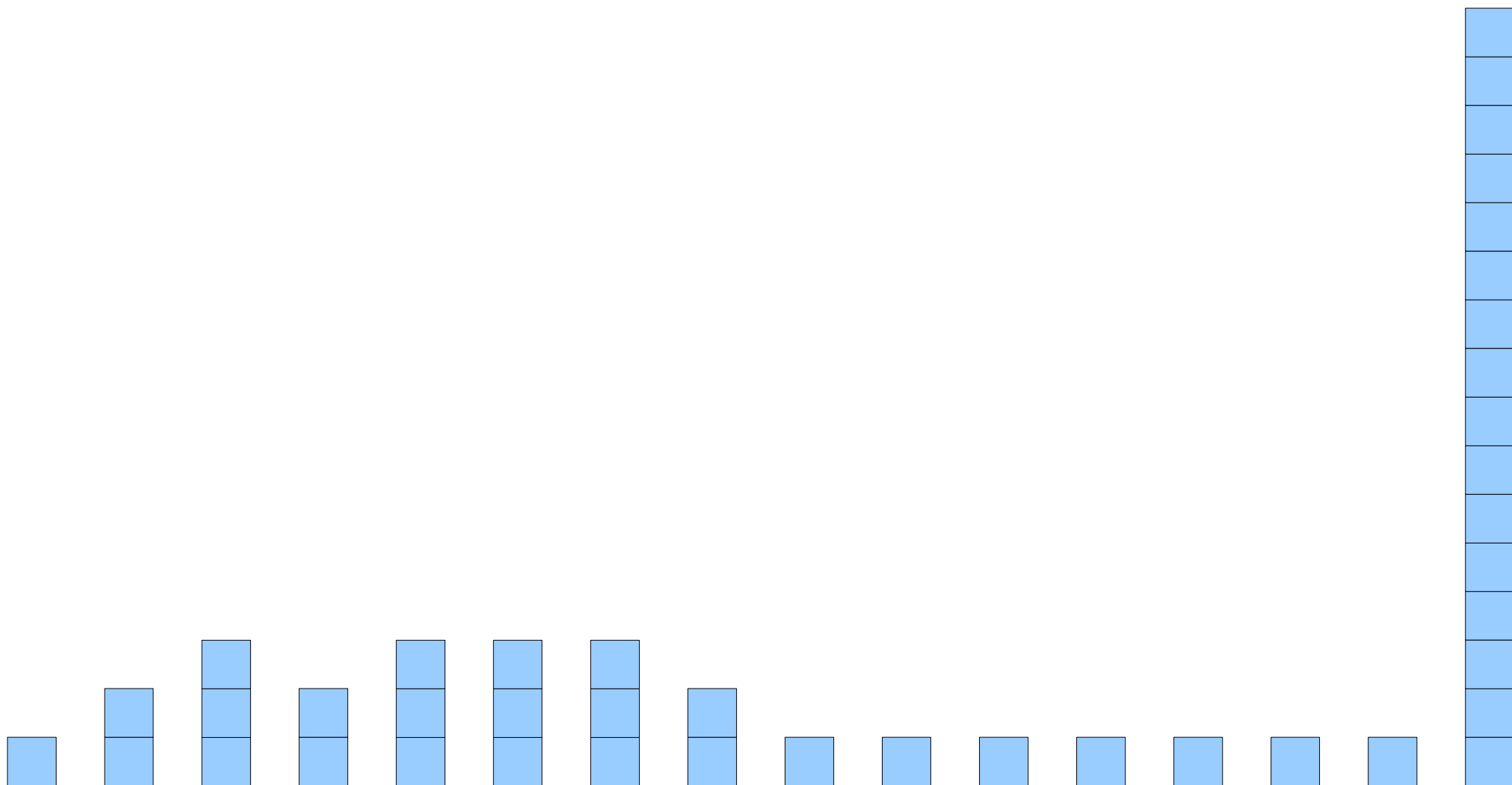
Spreading the Work



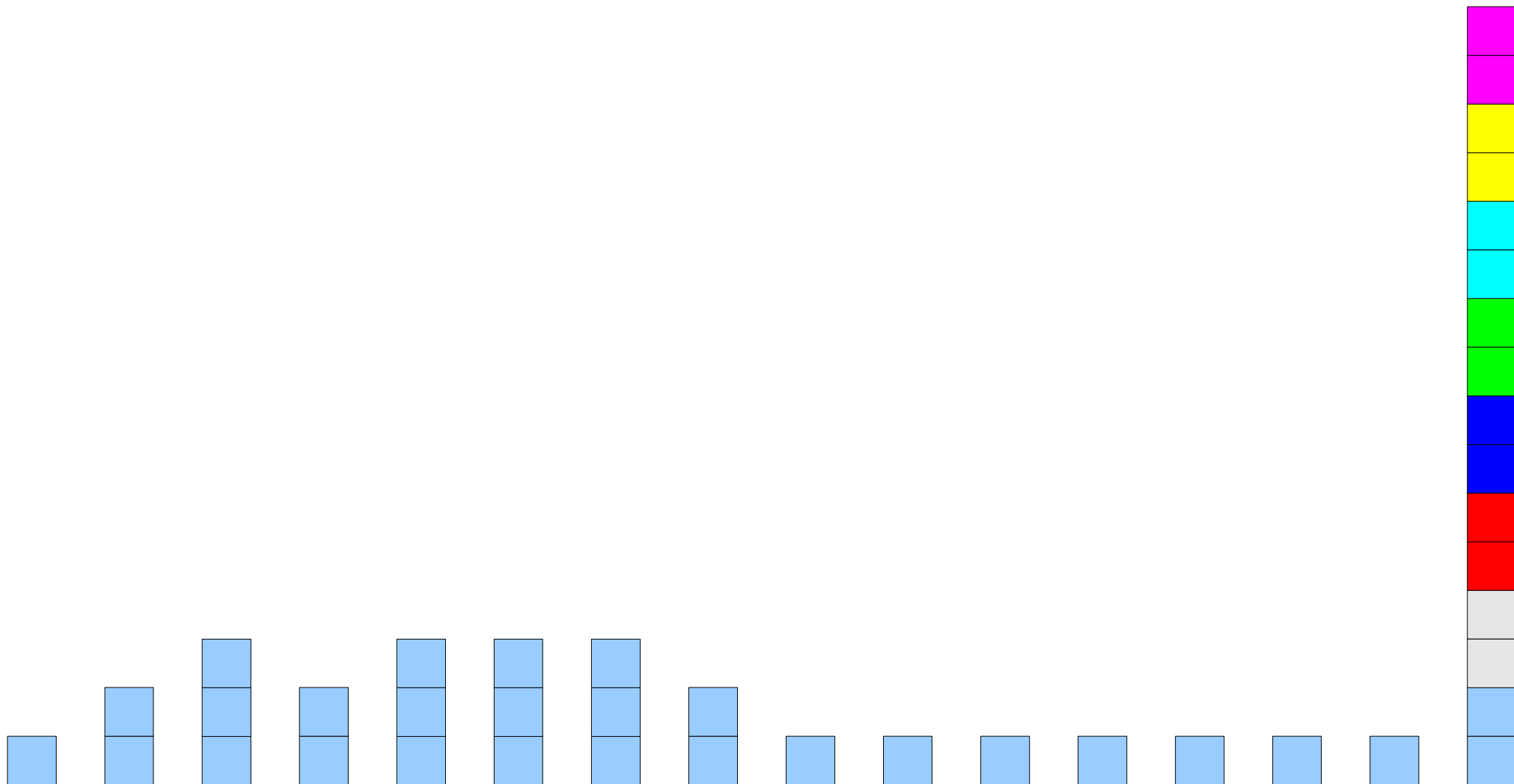
Spreading the Work



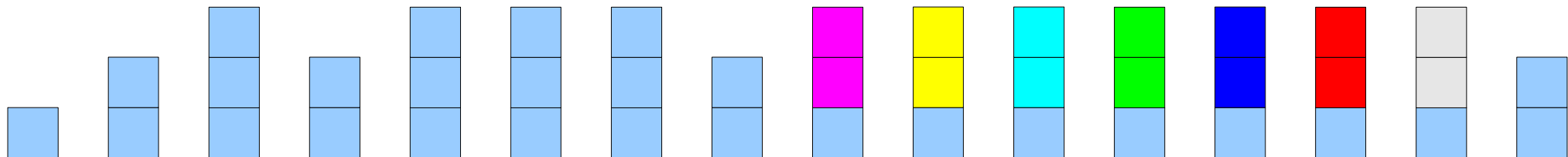
Spreading the Work



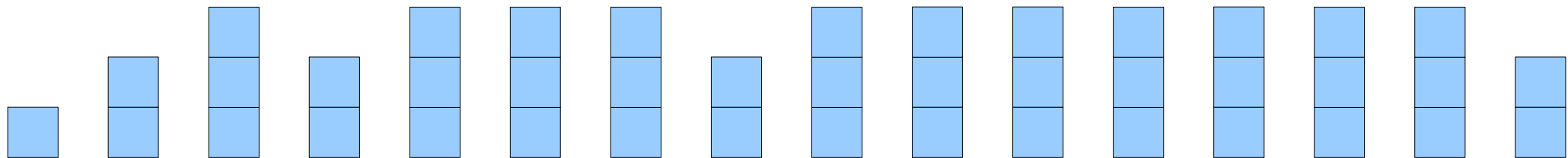
Spreading the Work



Spreading the Work



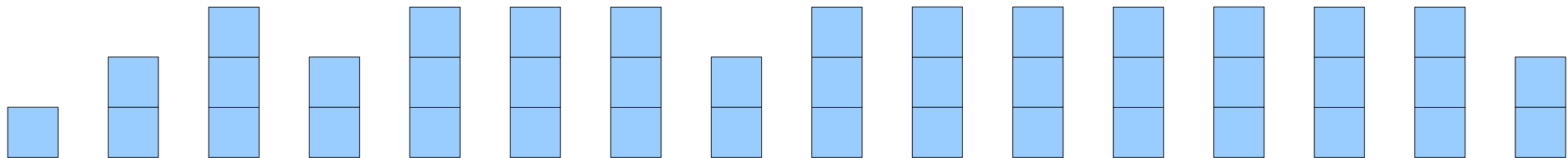
Spreading the Work



Spreading the Work

On average, we do
just 3 units of work!

This is $O(1)$ work on
average!



Sharing the Burden

- We still have “heavy” pushes taking time $O(n)$ and “light” pushes taking time $O(1)$.
- Worst-case time for a push is $O(n)$.
- Heavy pushes become so rare that the **average** time for a push is $O(1)$.
- Can we confirm this?

Amortized Analysis

- The analysis we have just done is called an **amortized analysis**.
- Reason about the total amount of work done, not the work done per operation.
- In an amortized sense, our implementation of the stack is extremely fast!
- This is one of the most common approaches to implementing **Stack**.