Implementing Abstractions

Part Two

Announcements

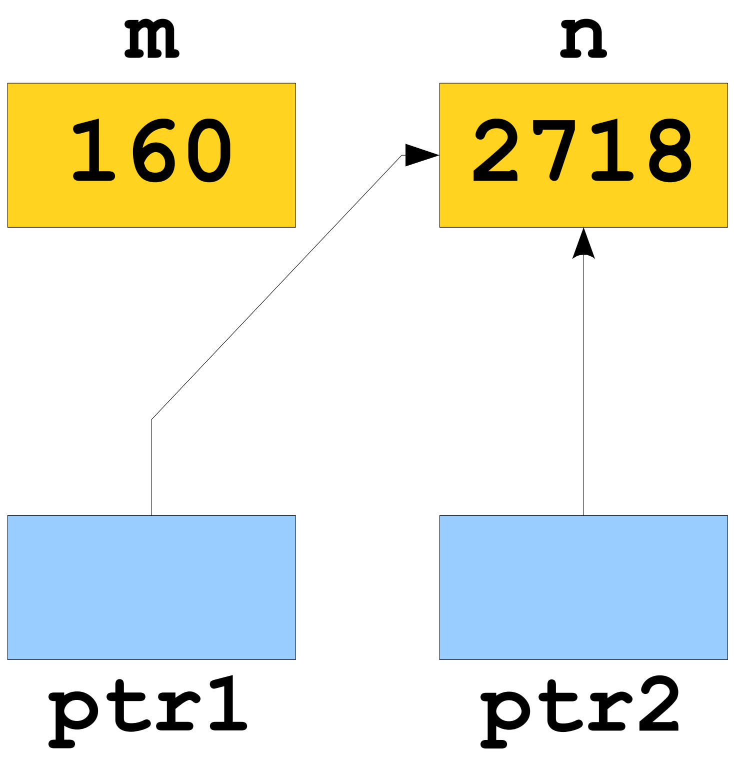
- Aubrey's Office hours cancelled today because he is sleepy

Exams

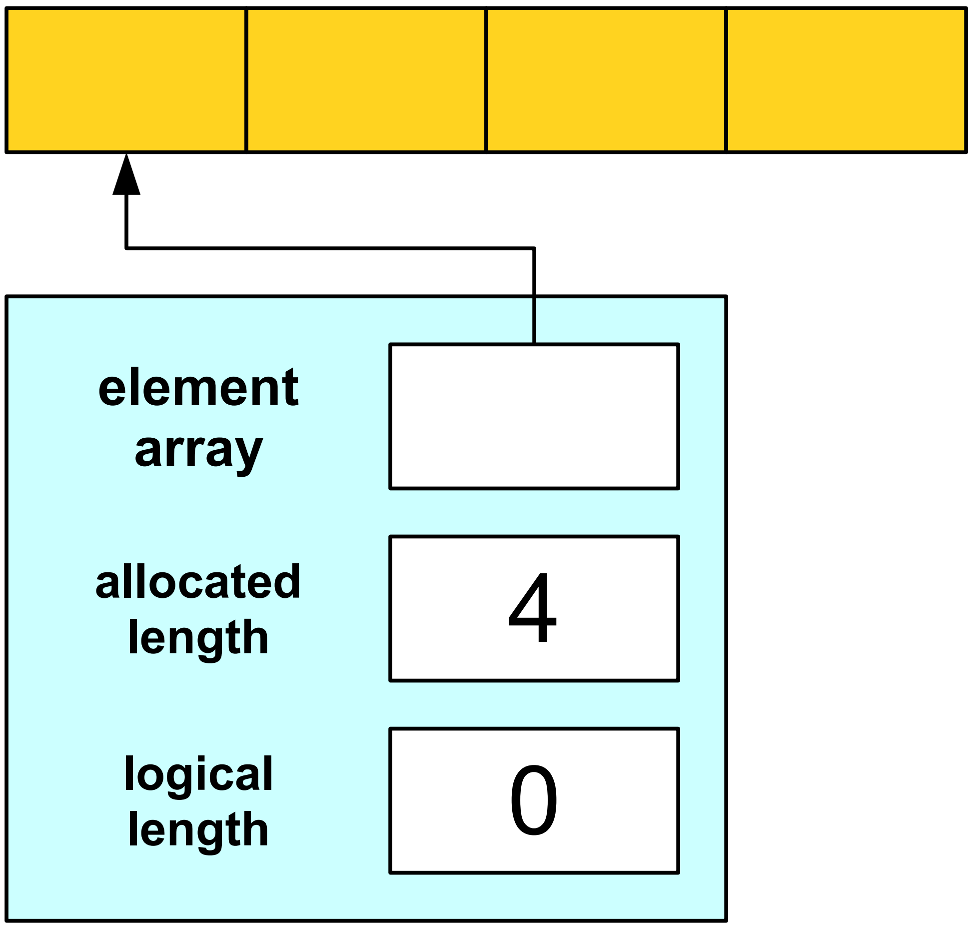
- Can pick the up after class today or when Michael or I are in Gates 160
- Criteria will be posted online later today
  - Please take a look, especially if you think there's a grading error
  - Regrade requests need to happen in the next week
    - We reserve the right to fix any grading mistakes we see (not just ones in your favor) so please look at the entire exam!
- Exam statistics at the end of lecture

Pointers, Visually

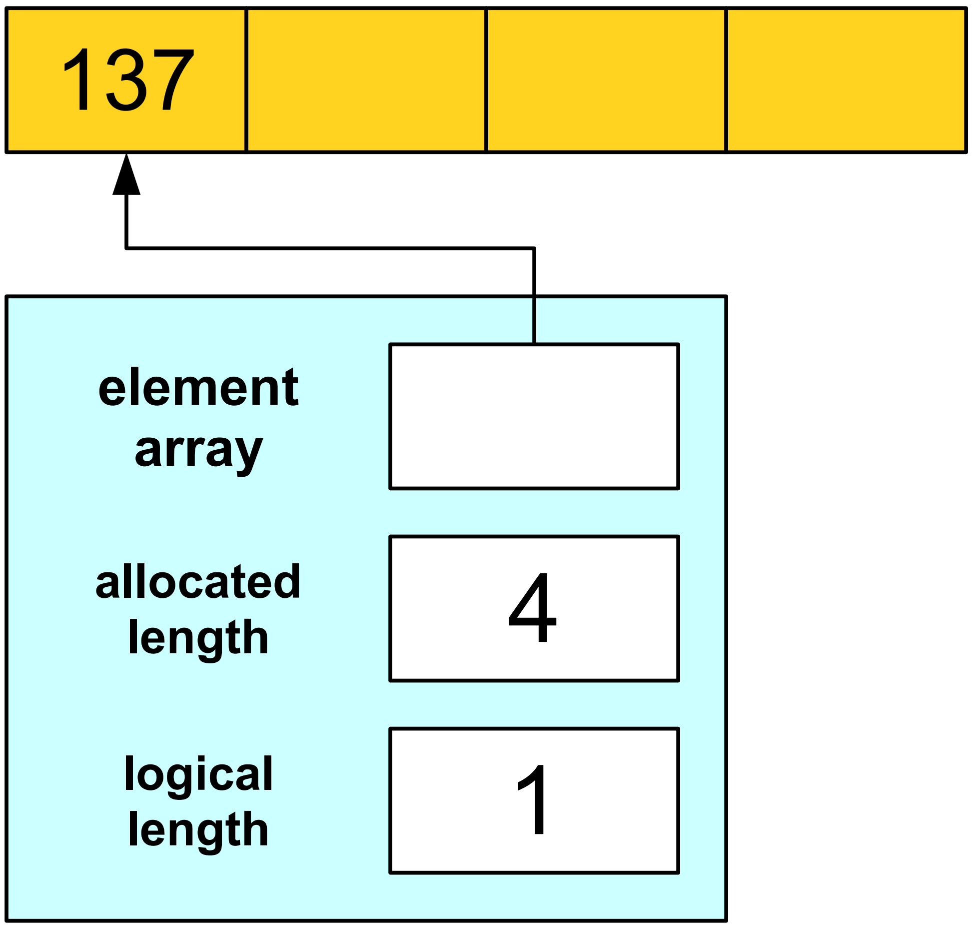
```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
m = 160;

ptr1 = ptr2;
```



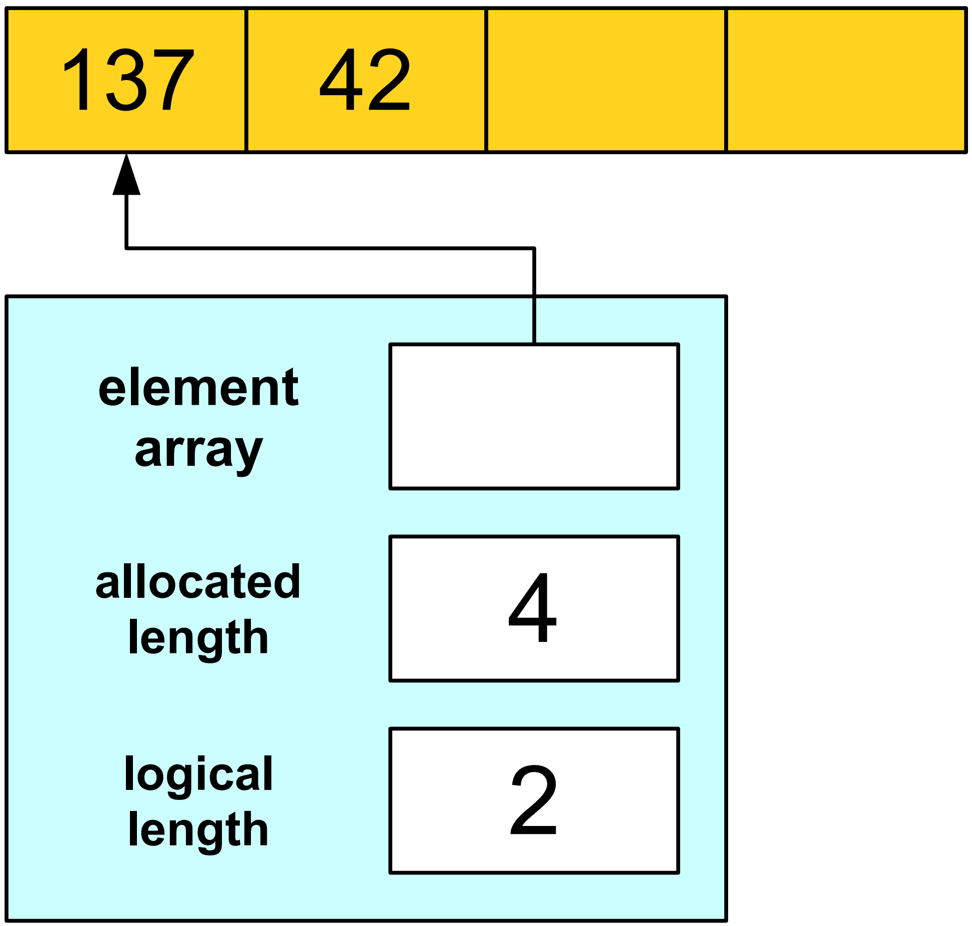
An Initial Idea



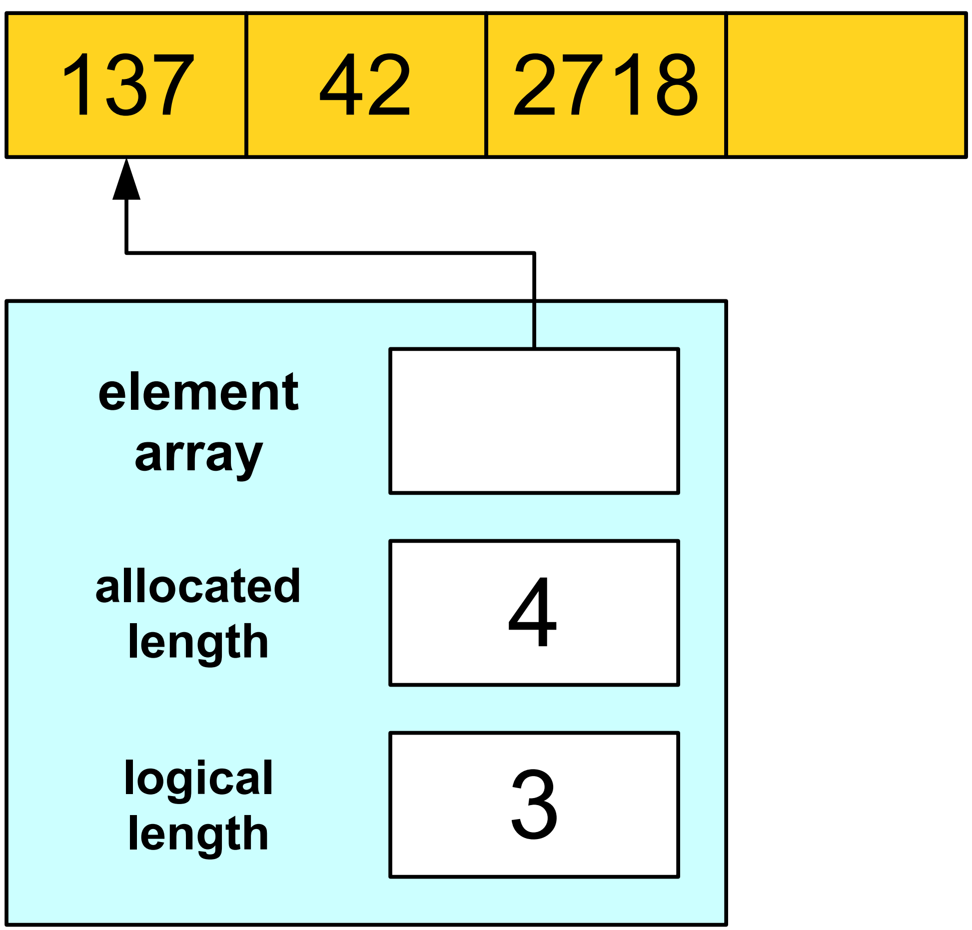
An Initial Idea



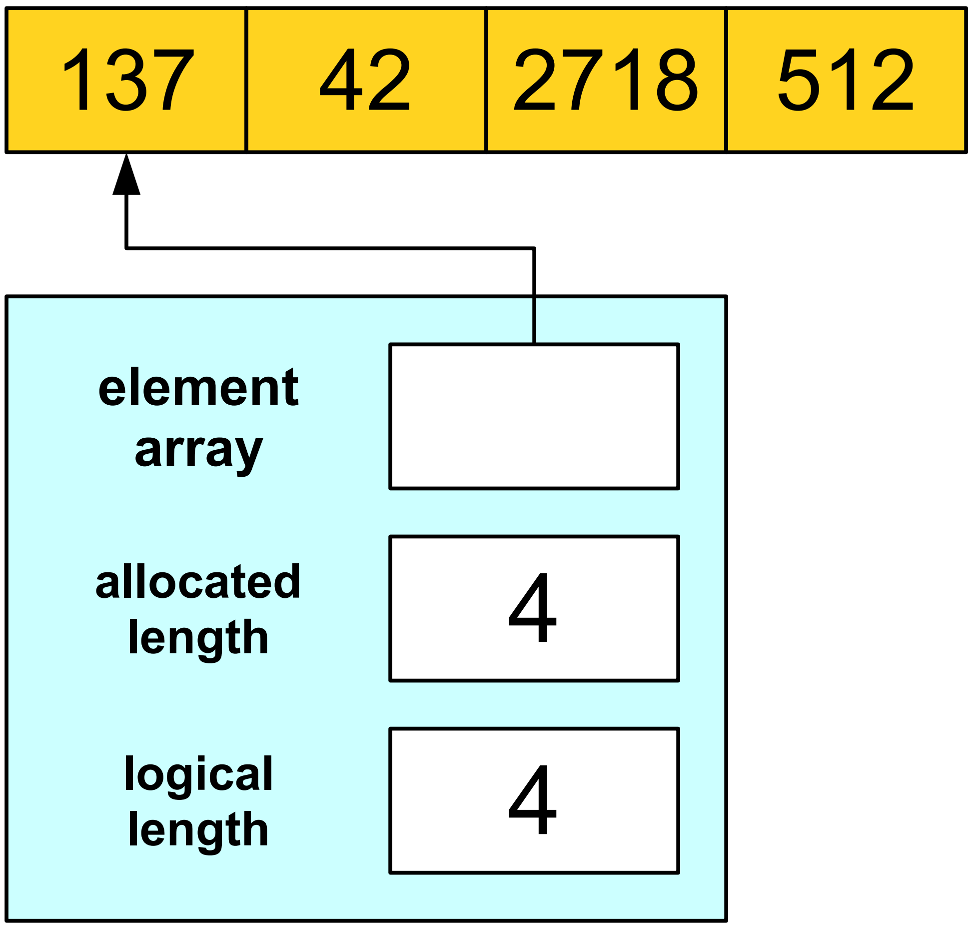
An Initial Idea



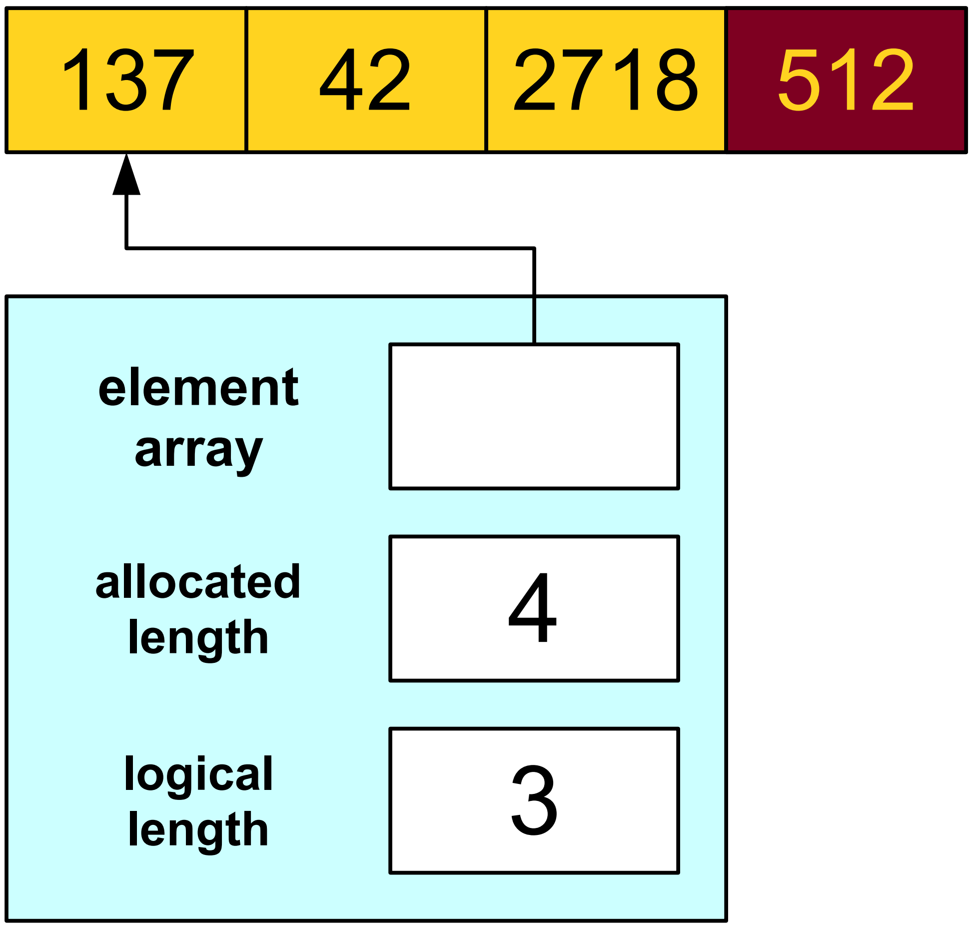
An Initial Idea



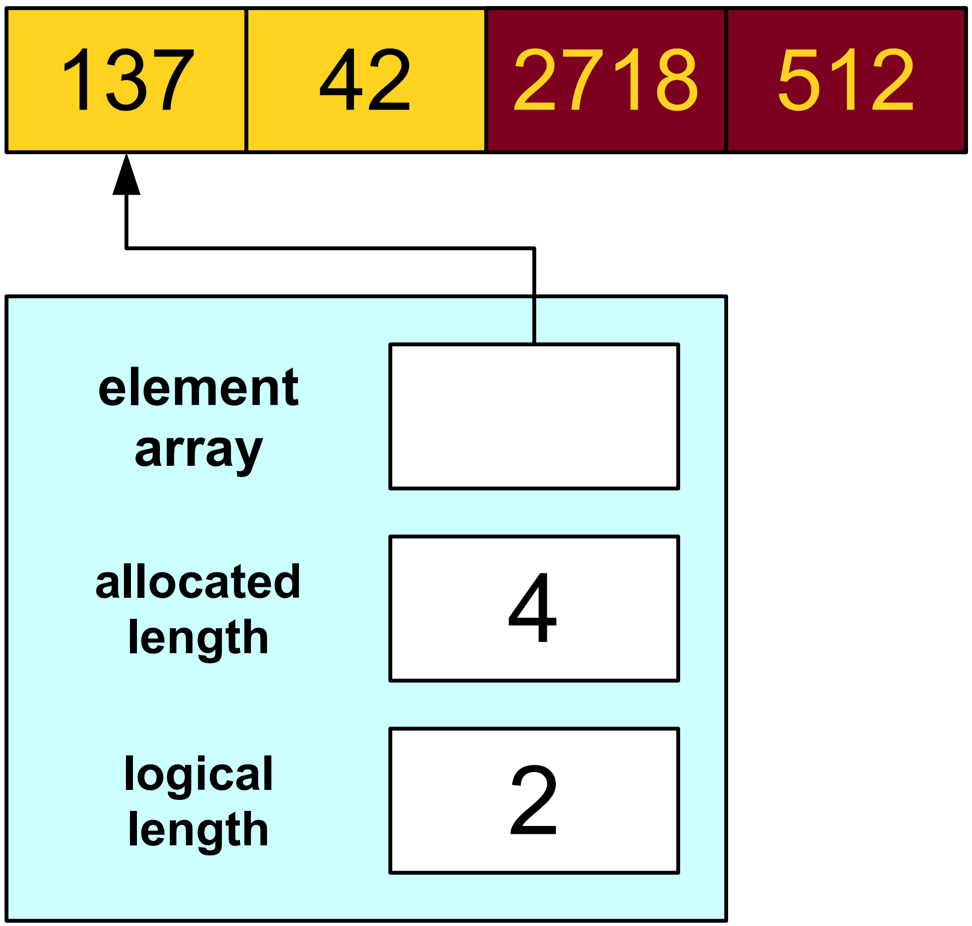
An Initial Idea



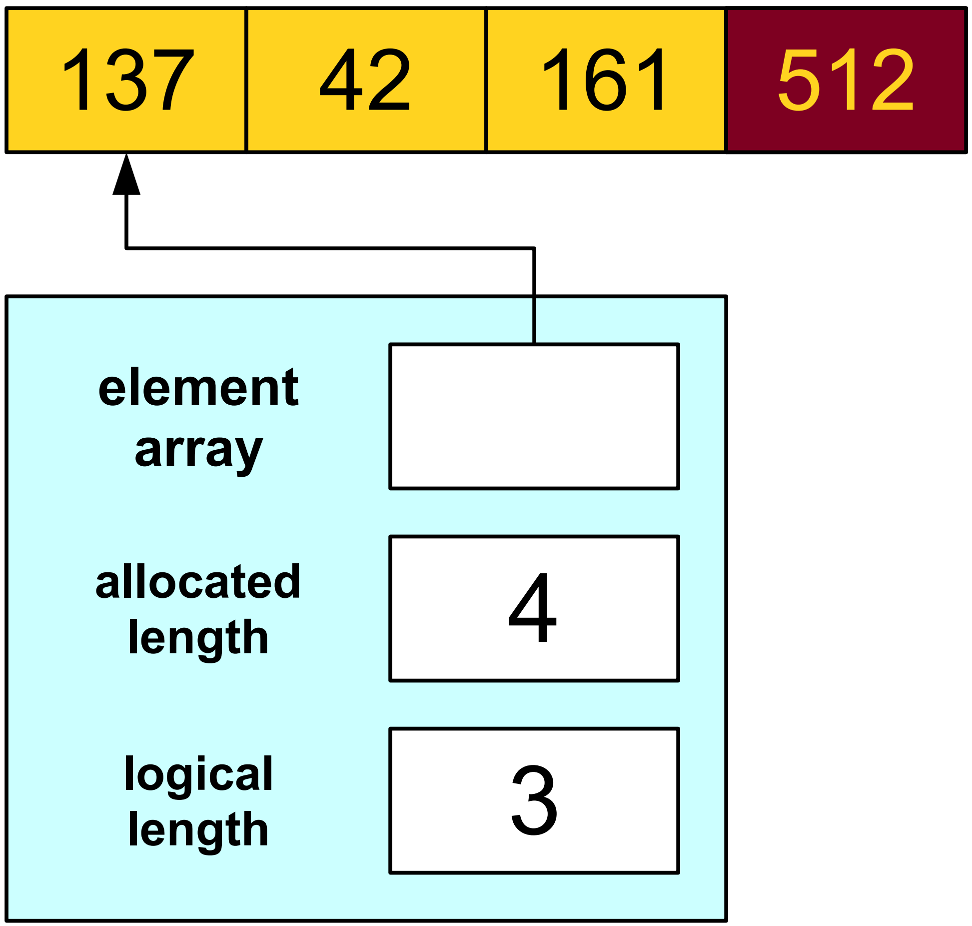
An Initial Idea



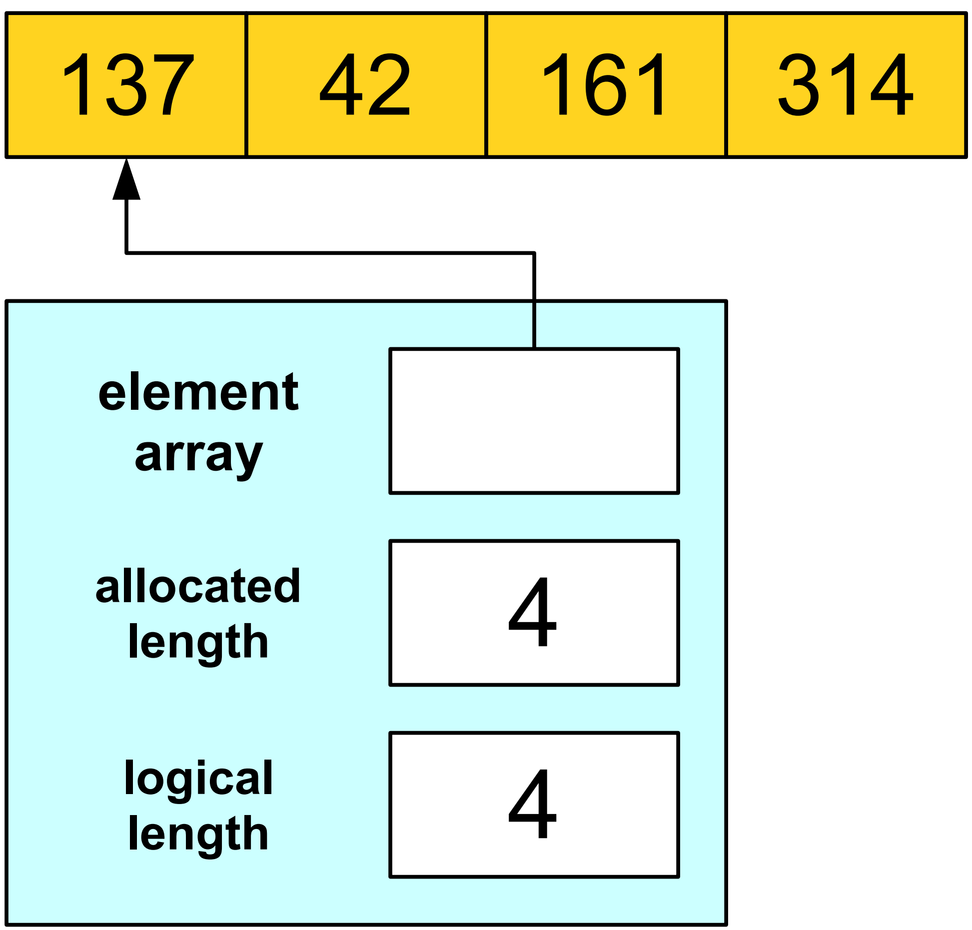
An Initial Idea



An Initial Idea



An Initial Idea

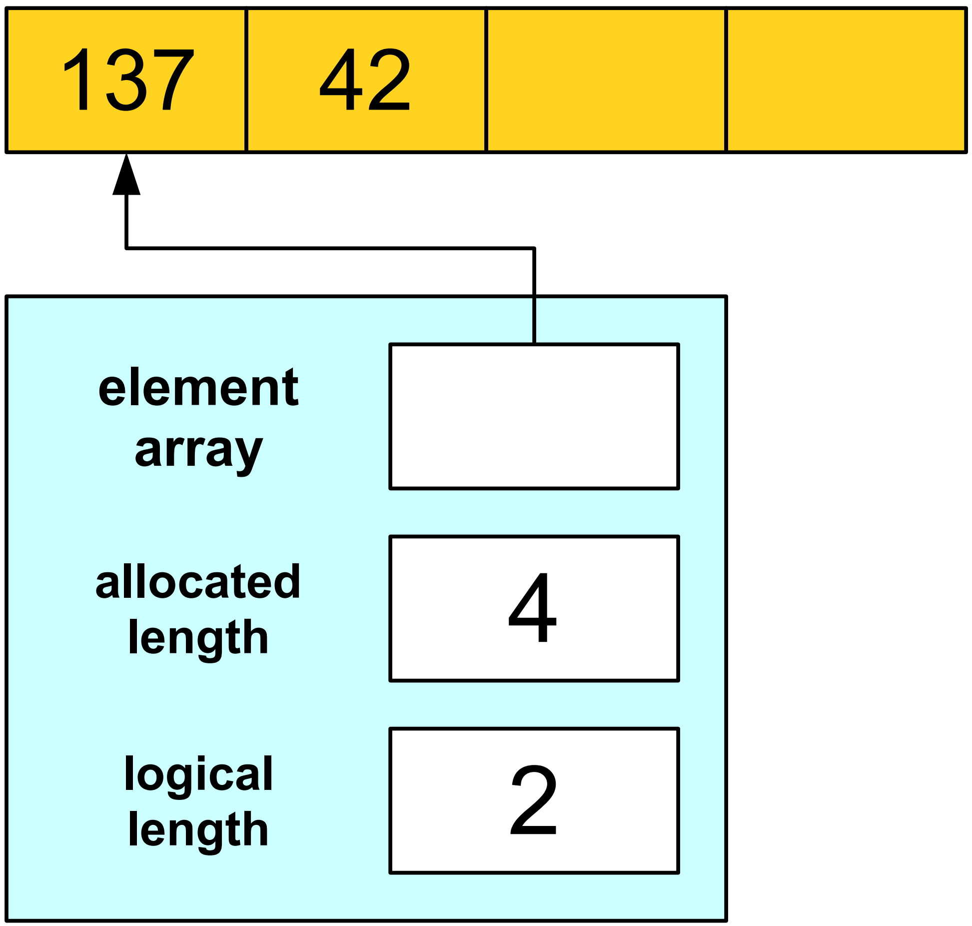


Finish Bounded Stack...

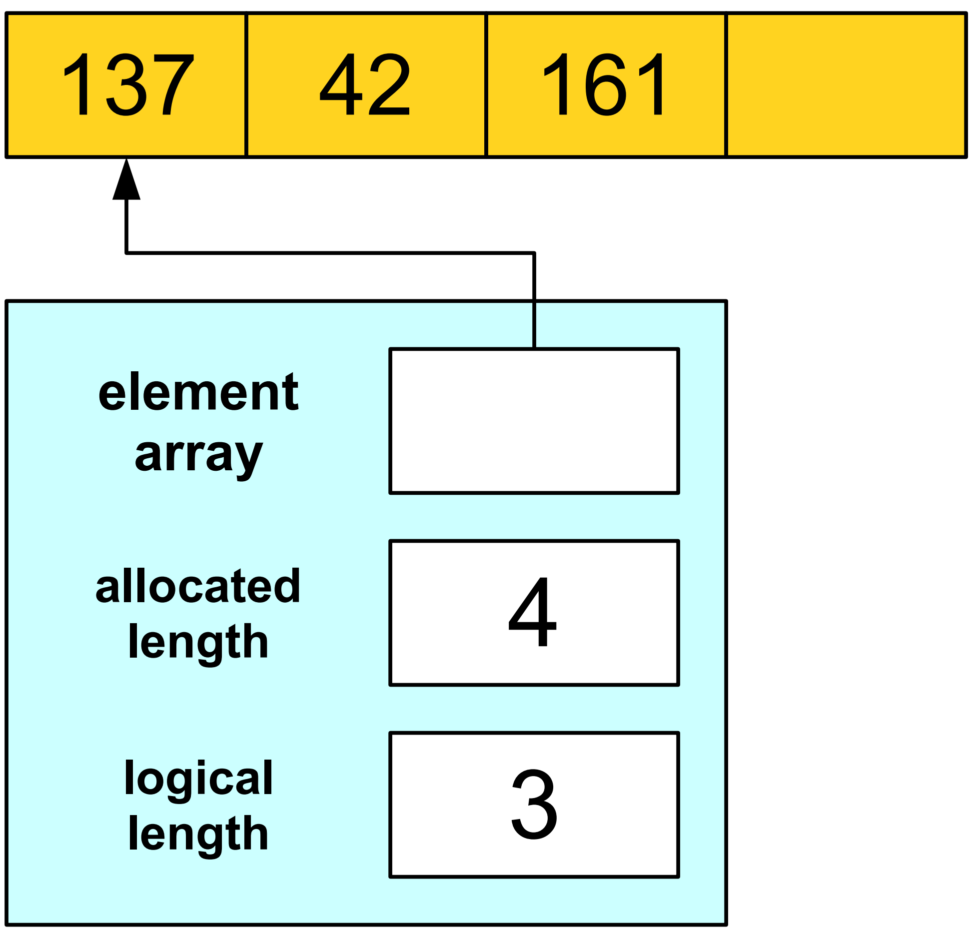
Running out of Space

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

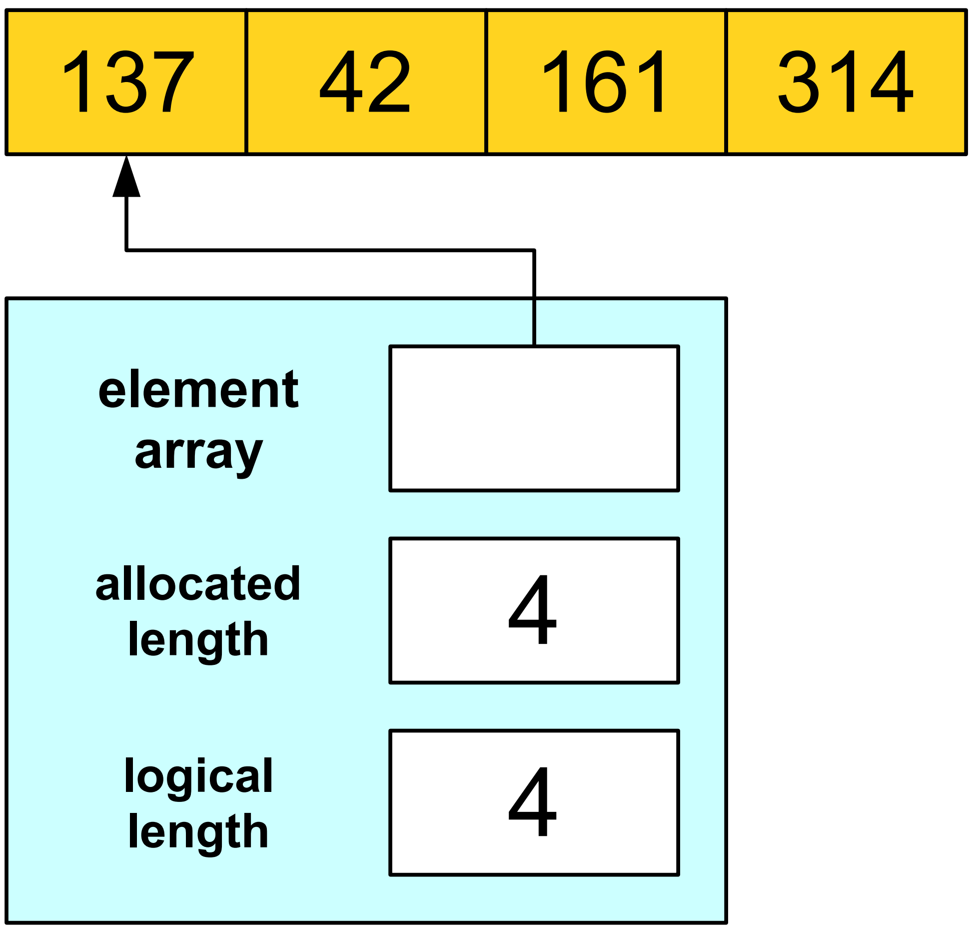
An Initial Idea



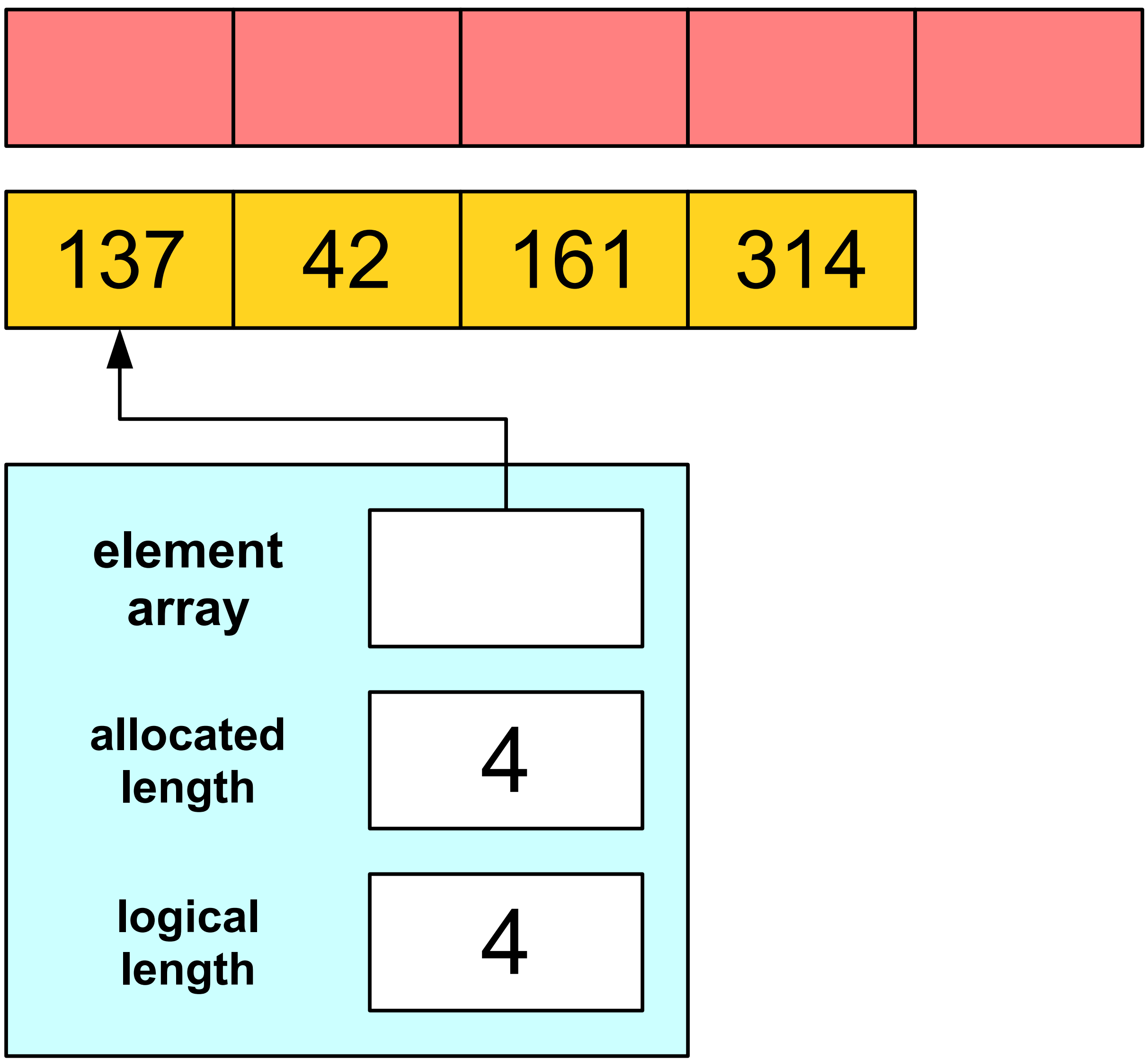
An Initial Idea



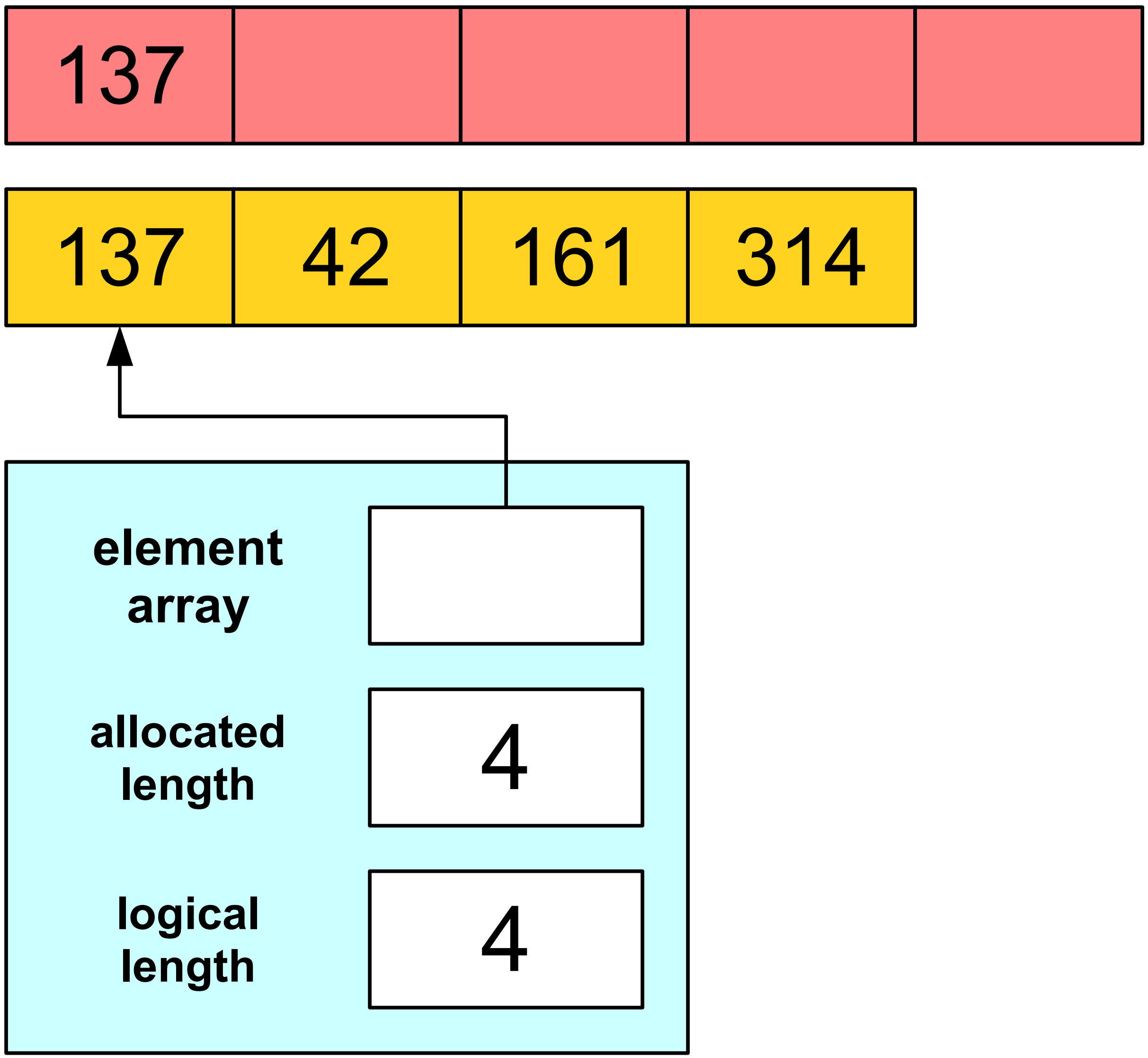
An Initial Idea



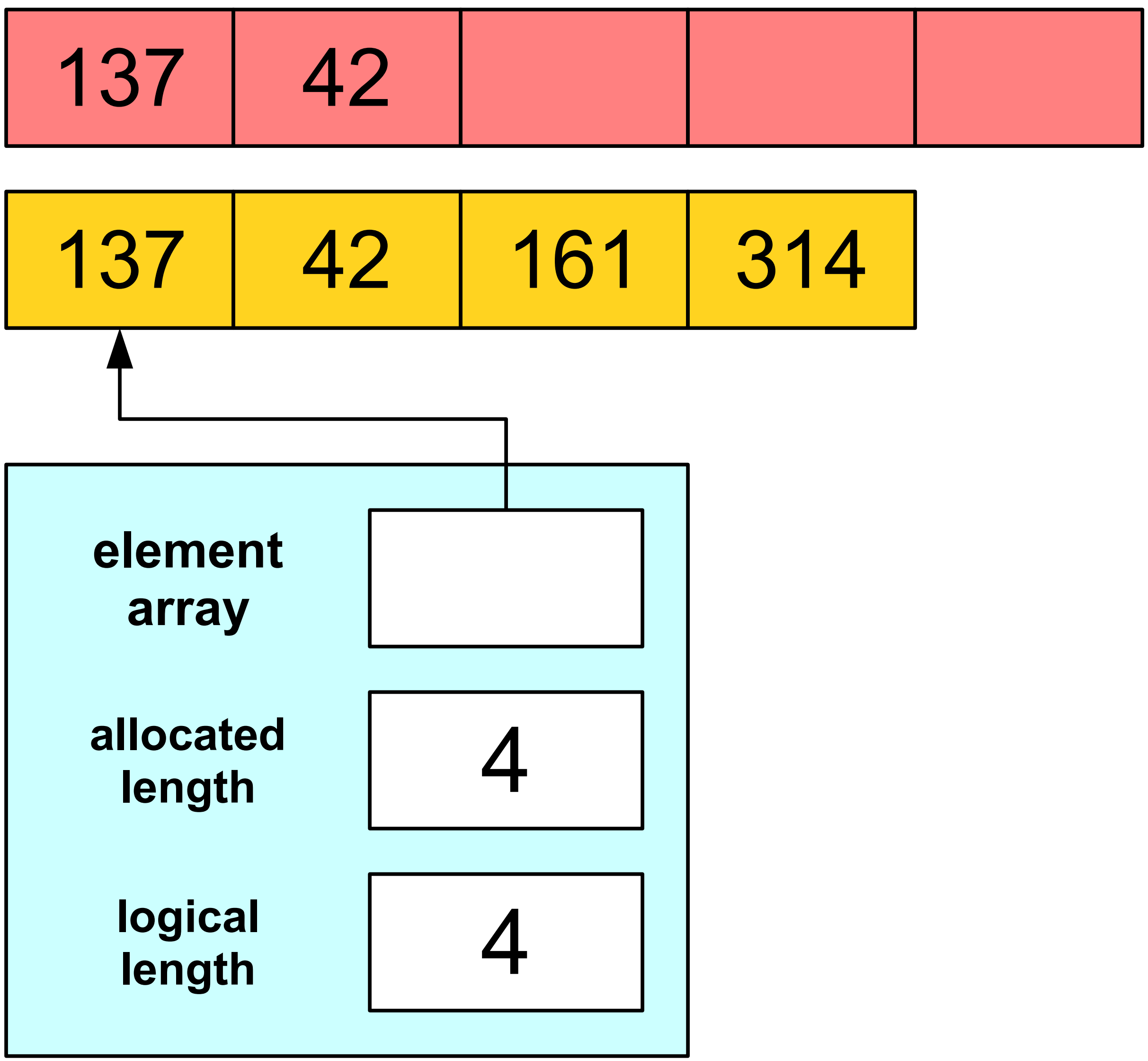
An Initial Idea



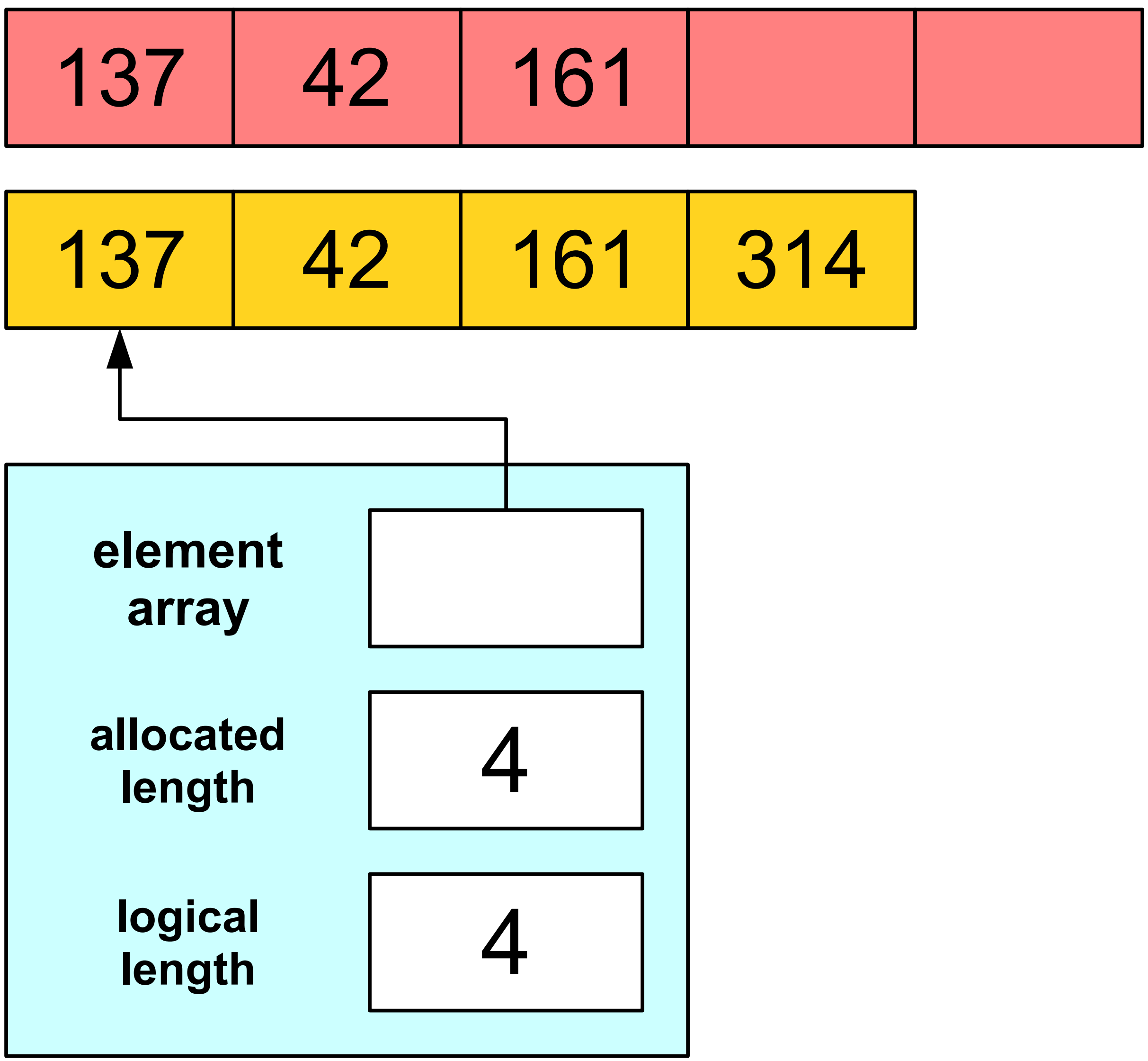
An Initial Idea



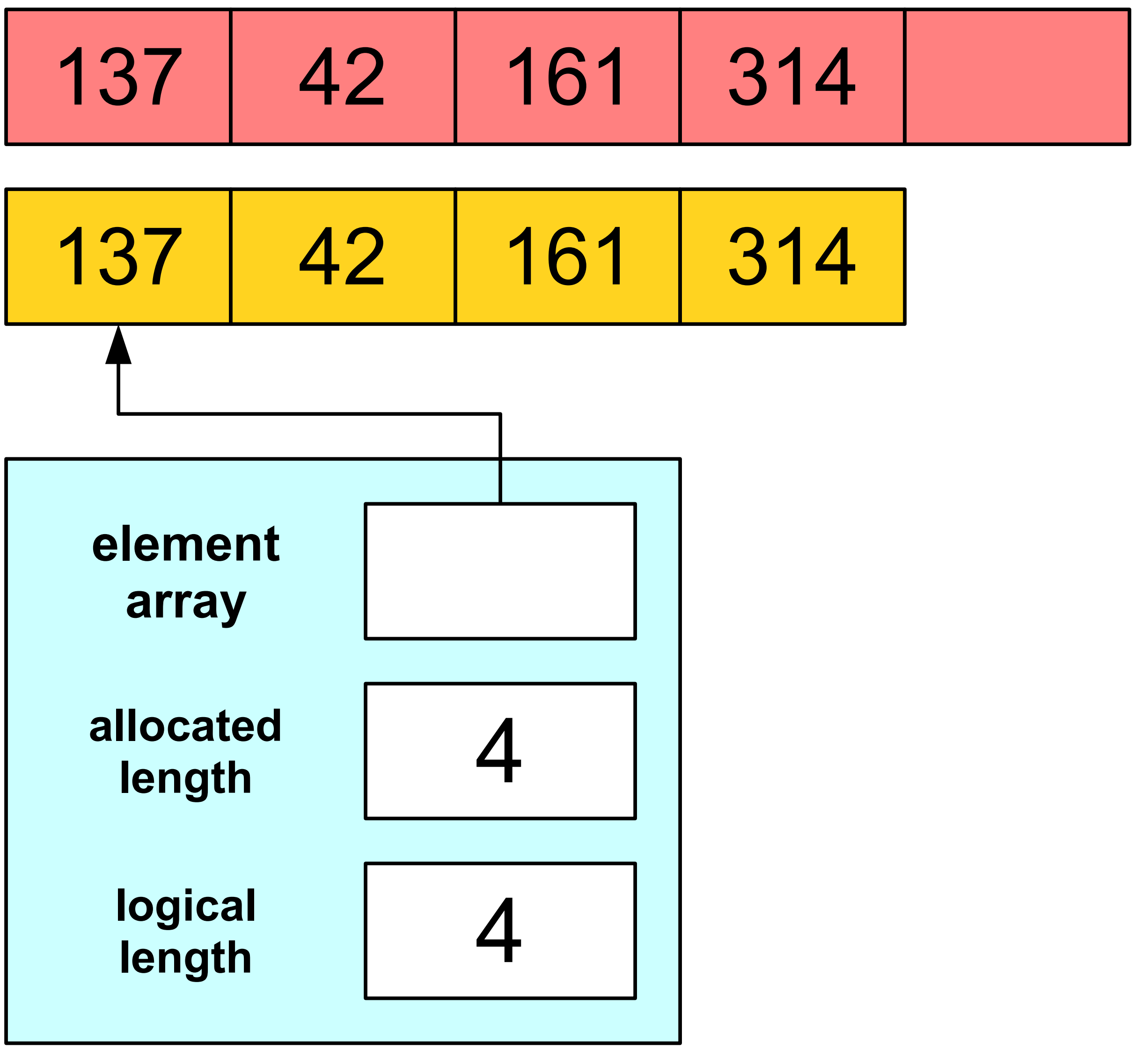
An Initial Idea



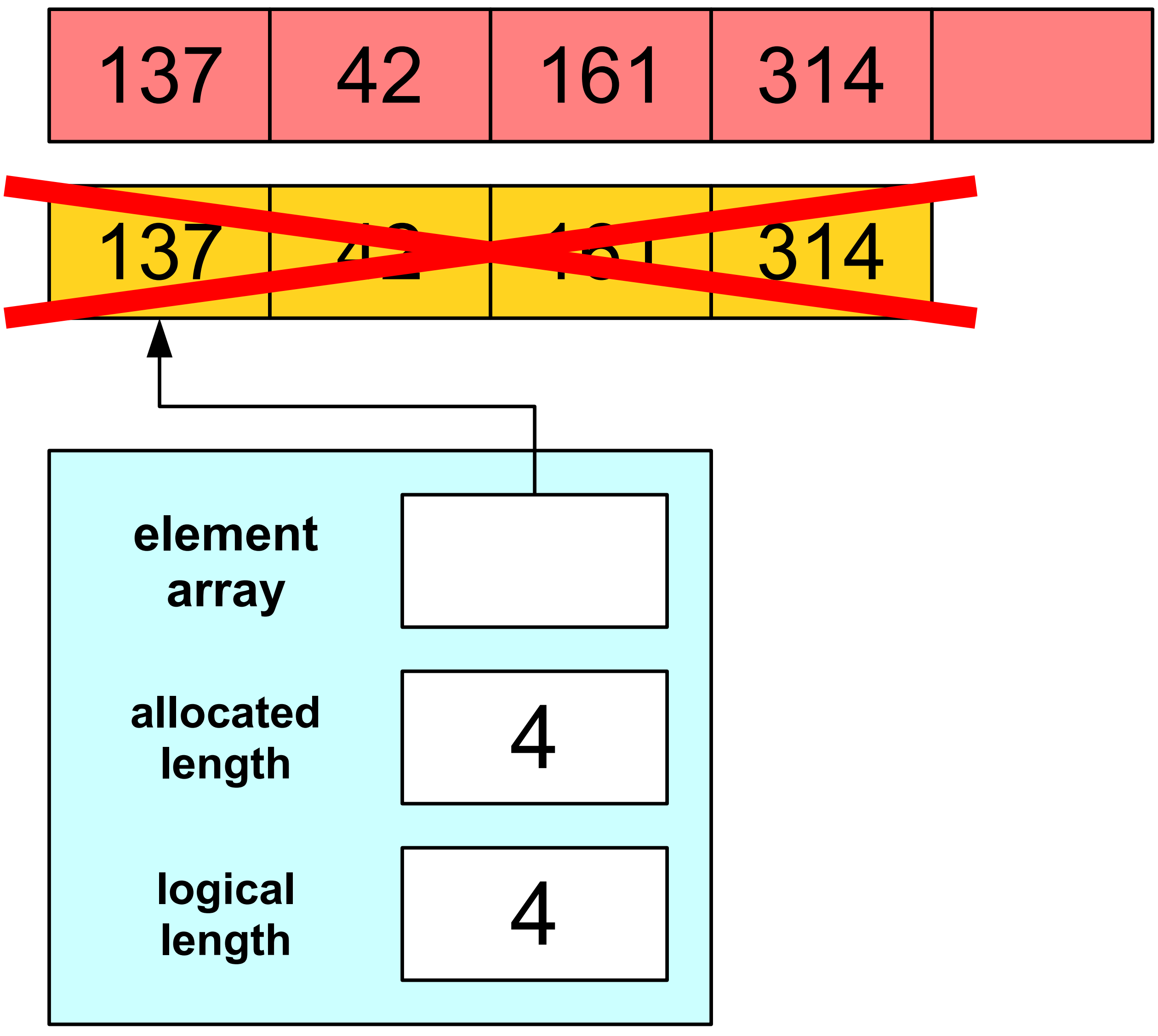
An Initial Idea



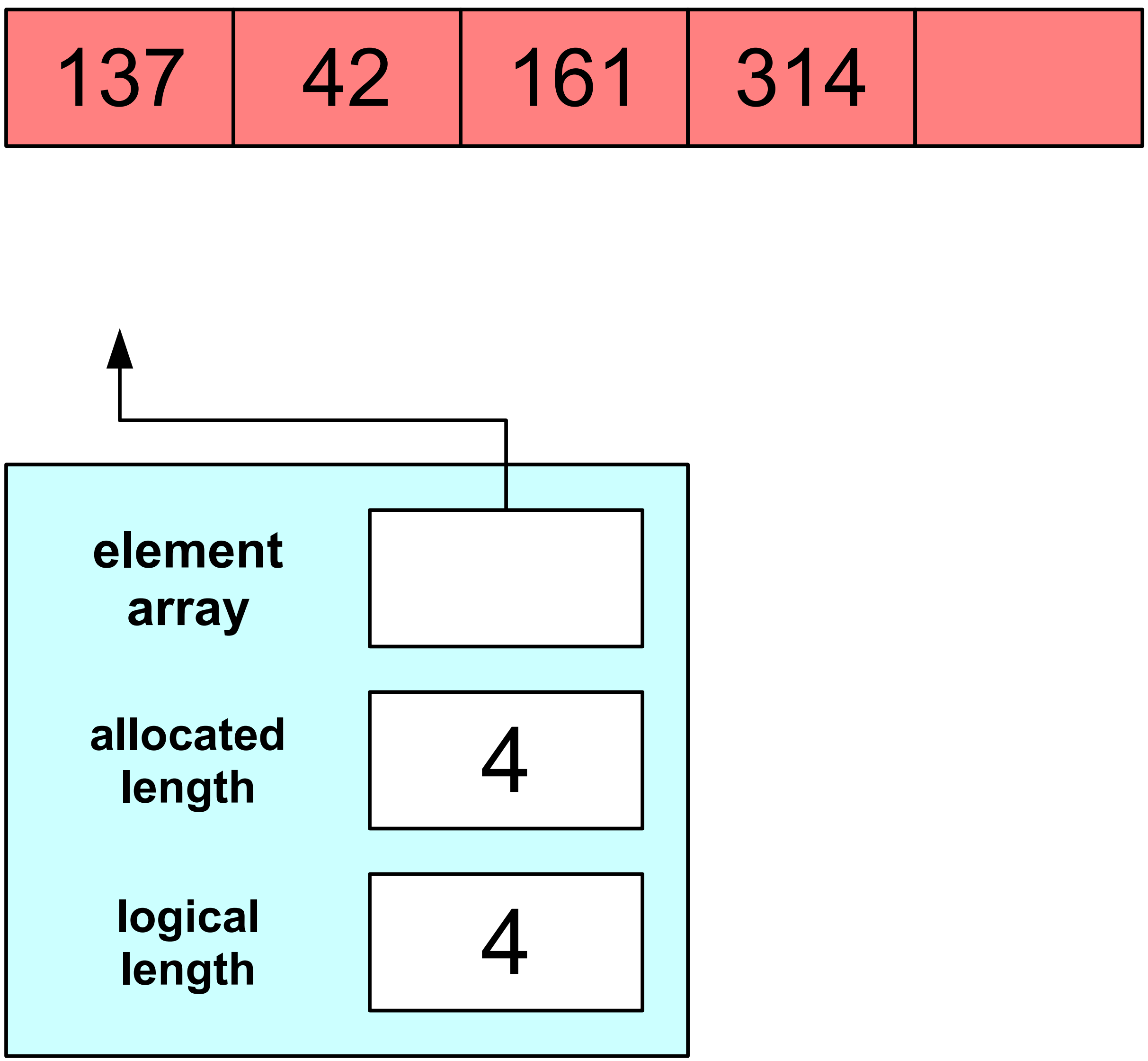
An Initial Idea



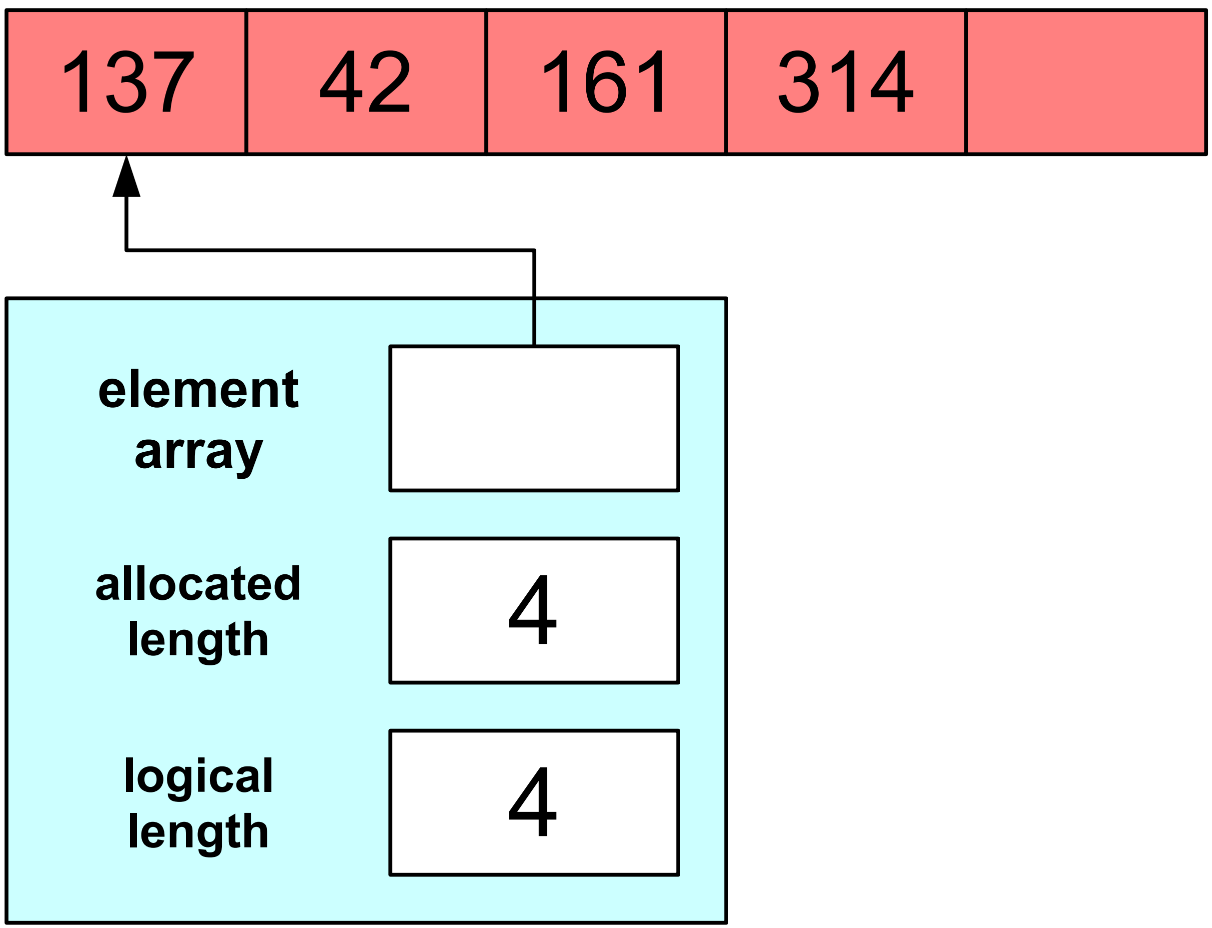
An Initial Idea



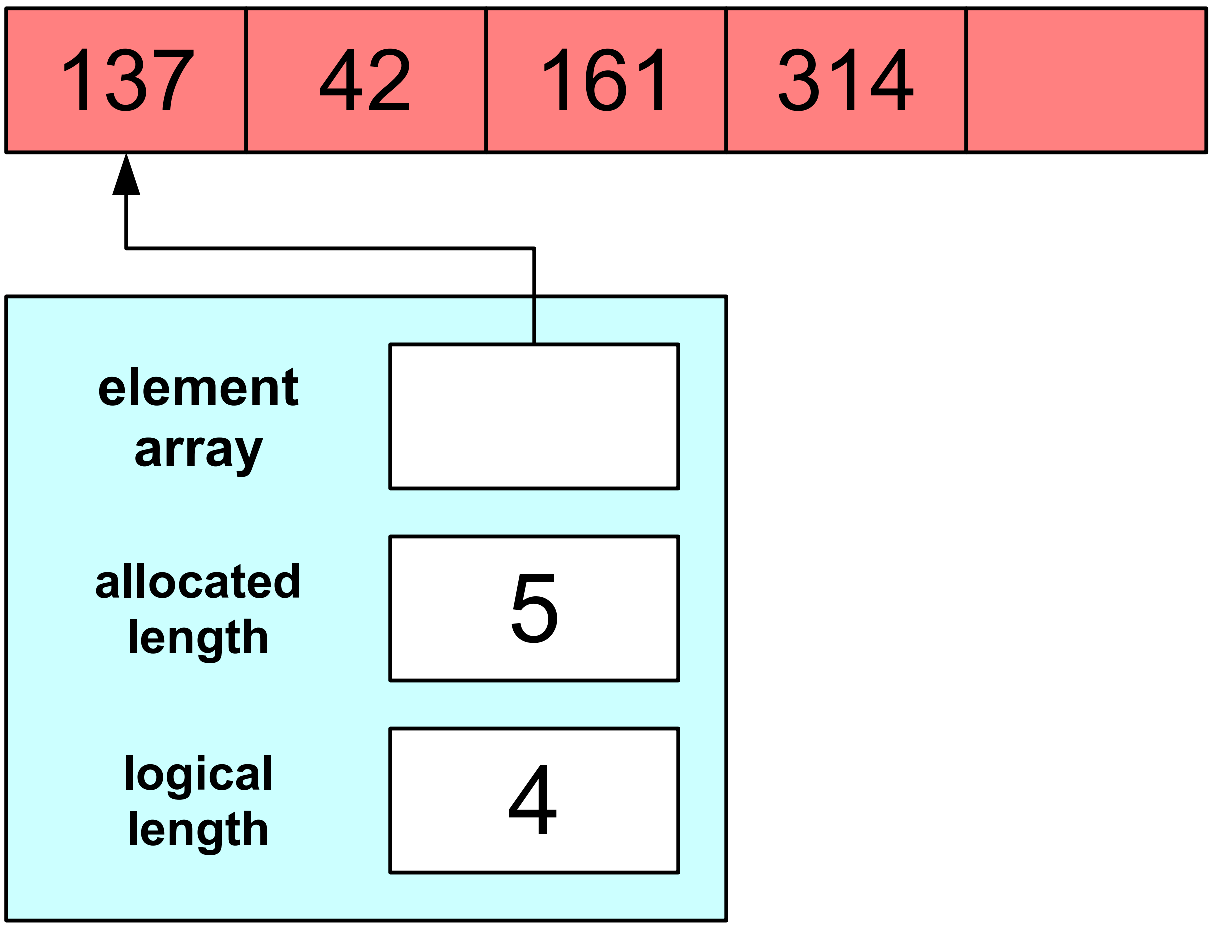
An Initial Idea



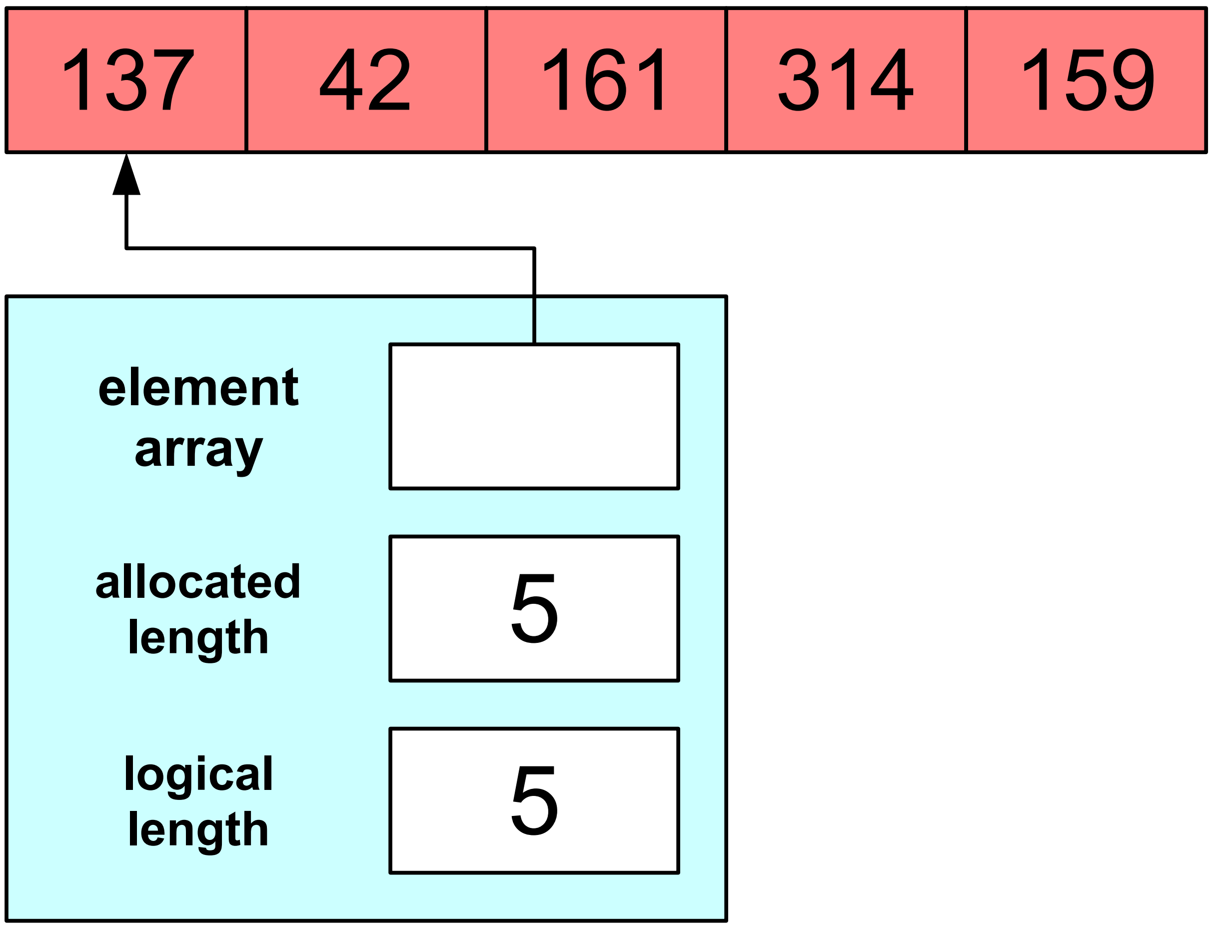
An Initial Idea



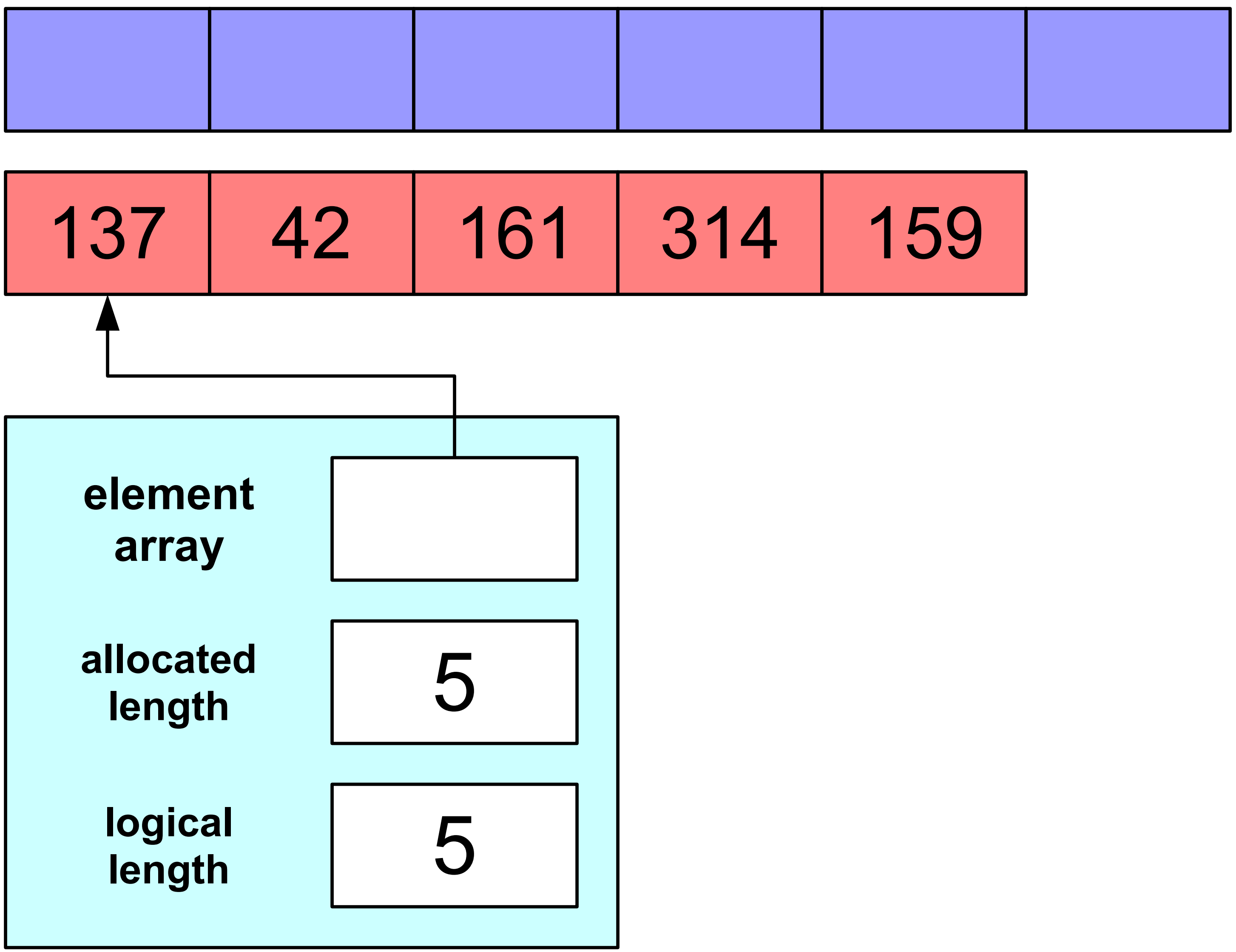
An Initial Idea



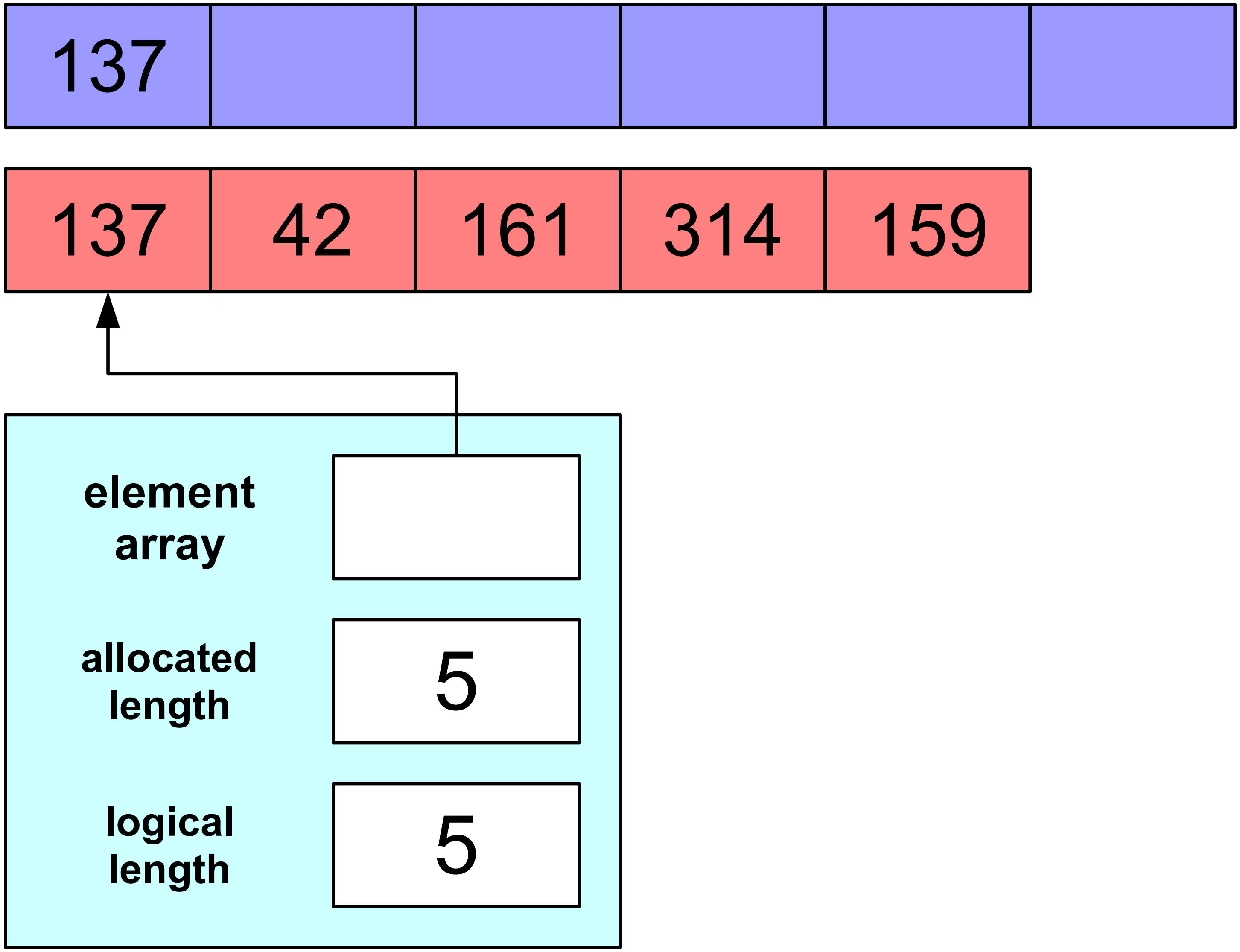
An Initial Idea



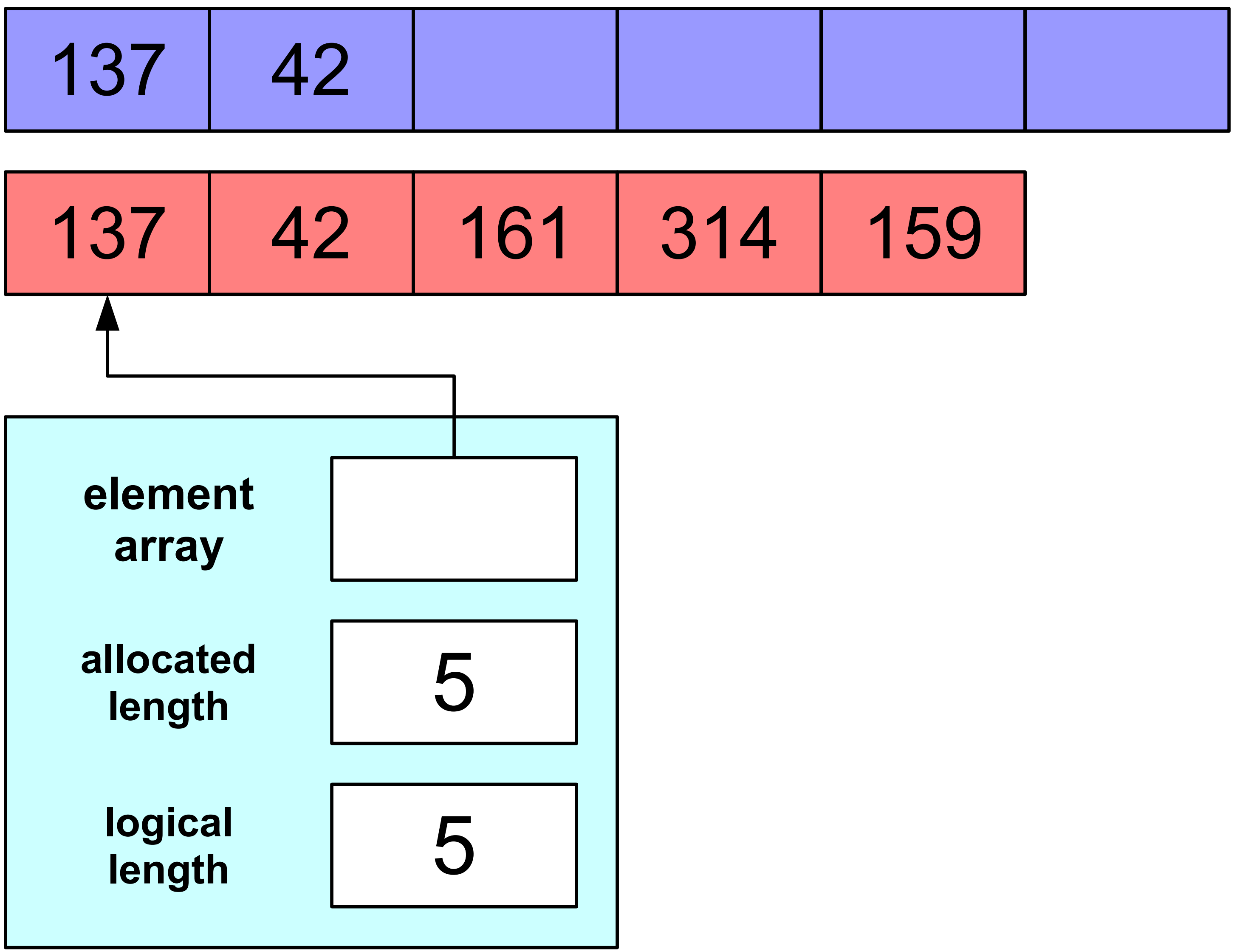
An Initial Idea



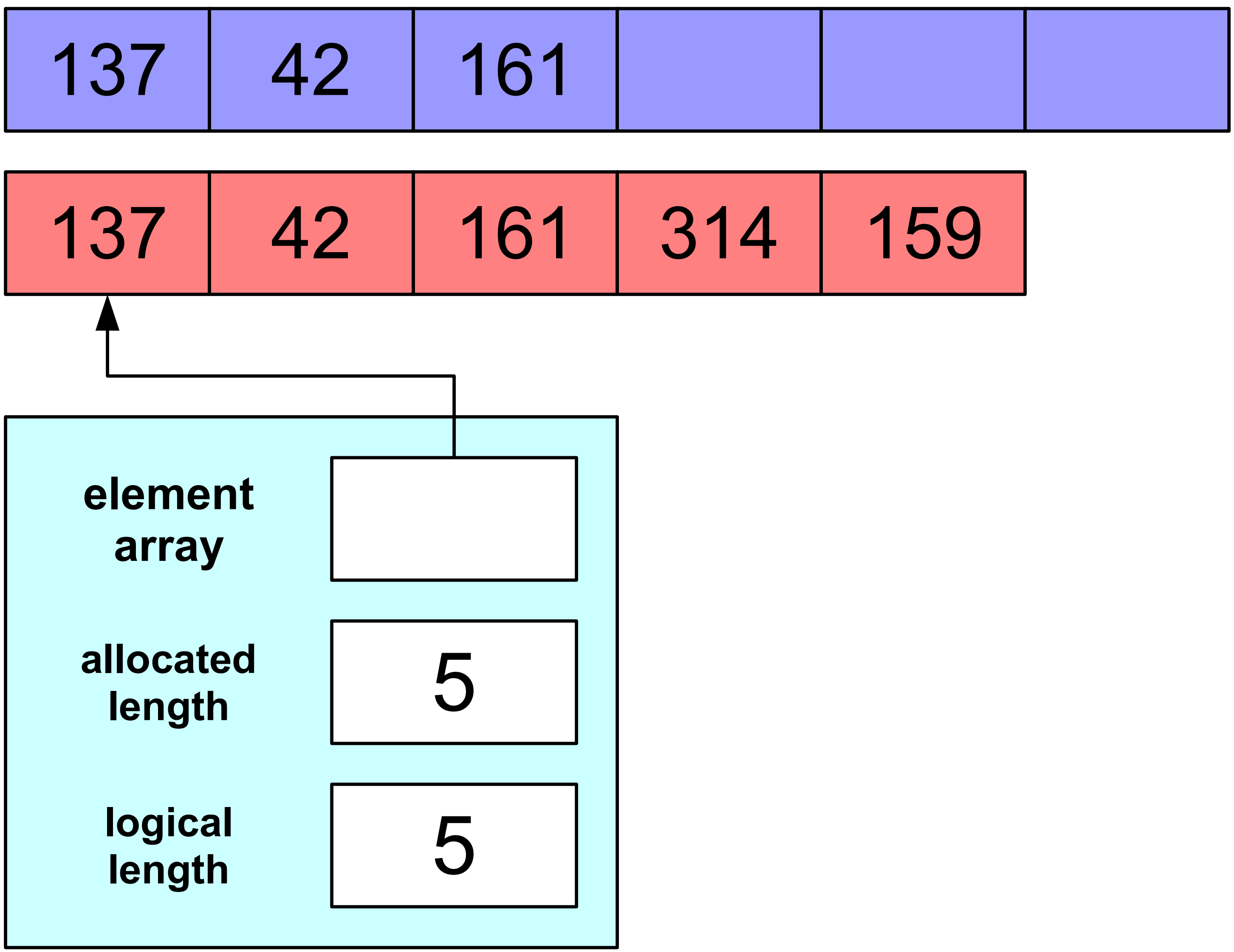
An Initial Idea



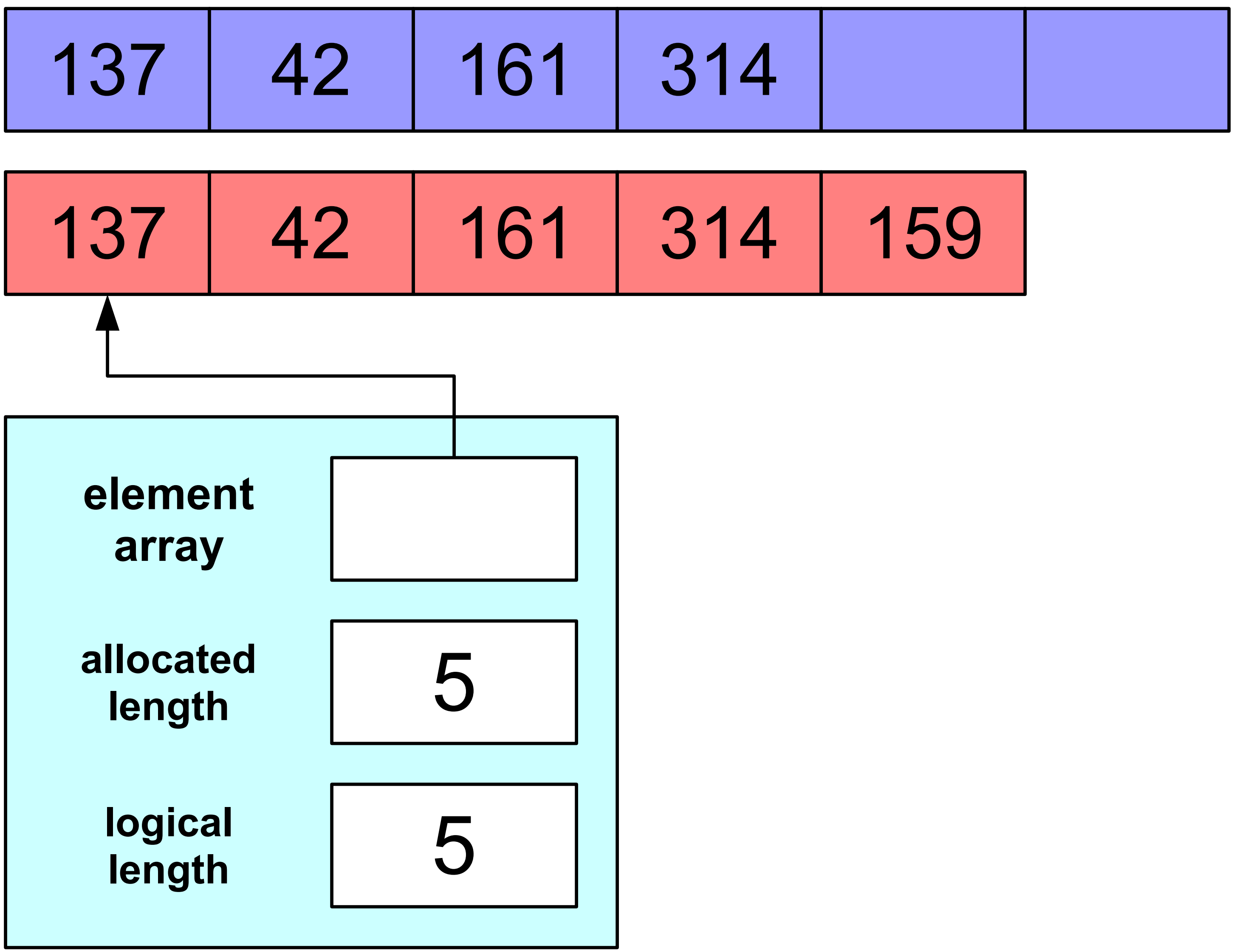
An Initial Idea



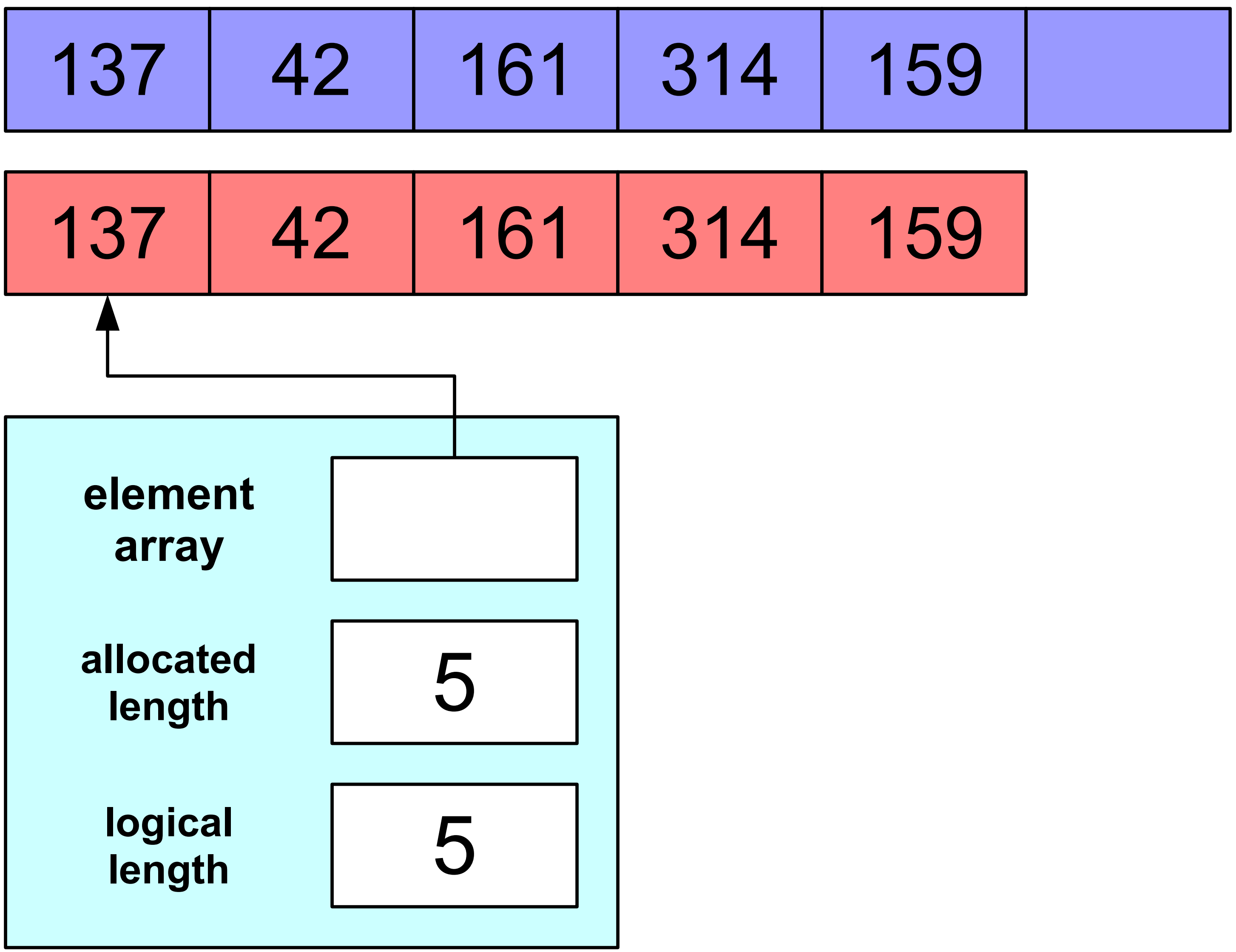
An Initial Idea



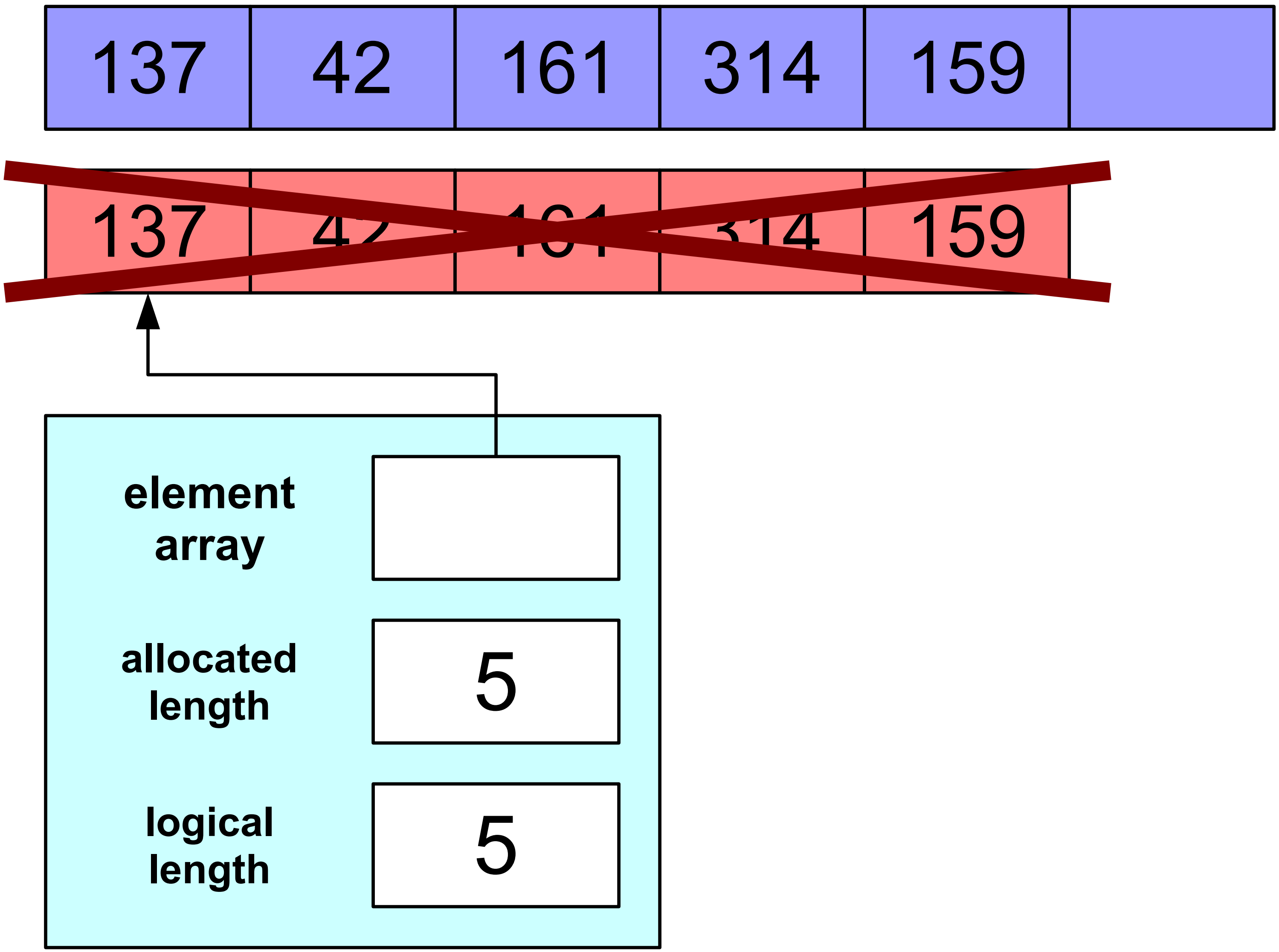
An Initial Idea



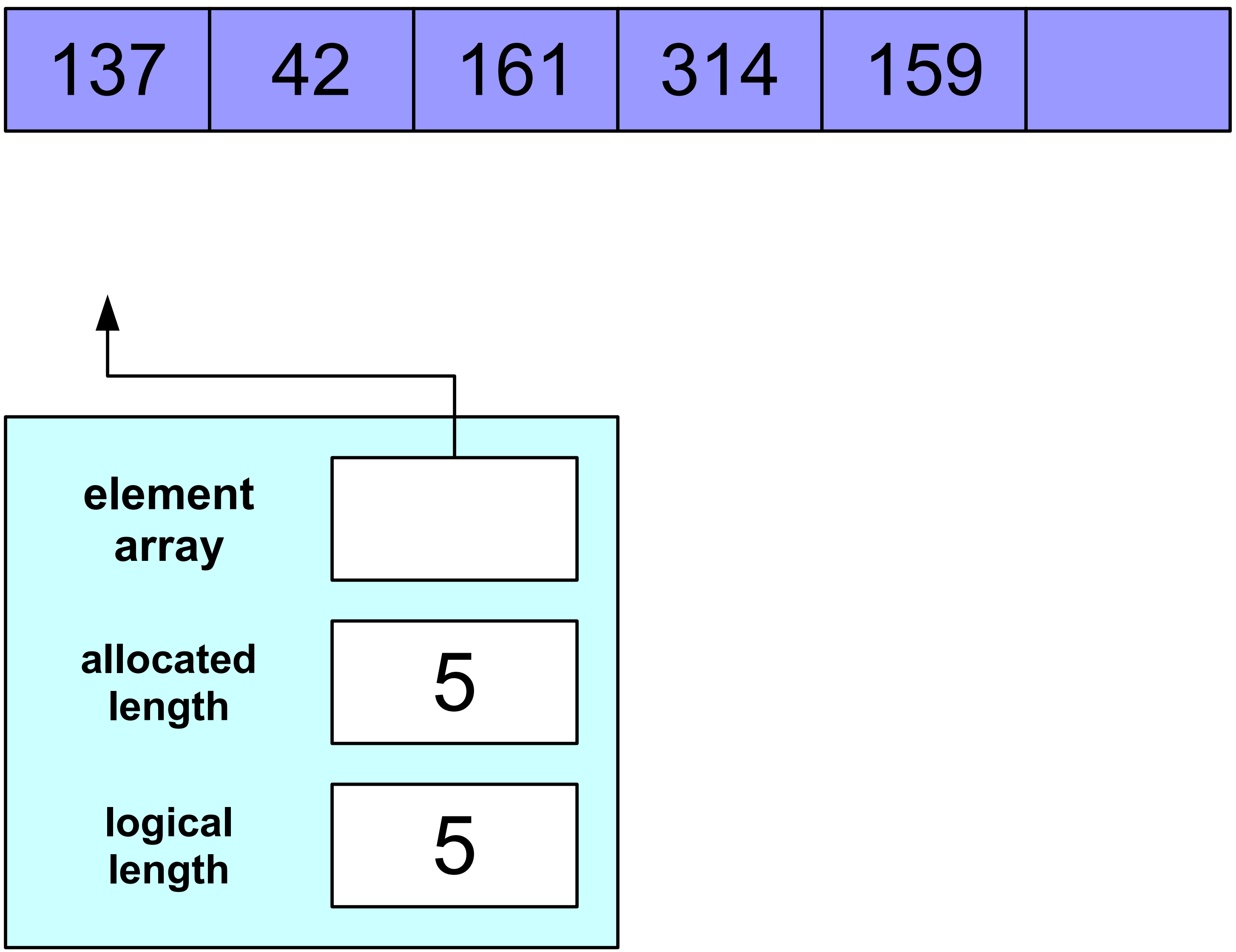
An Initial Idea



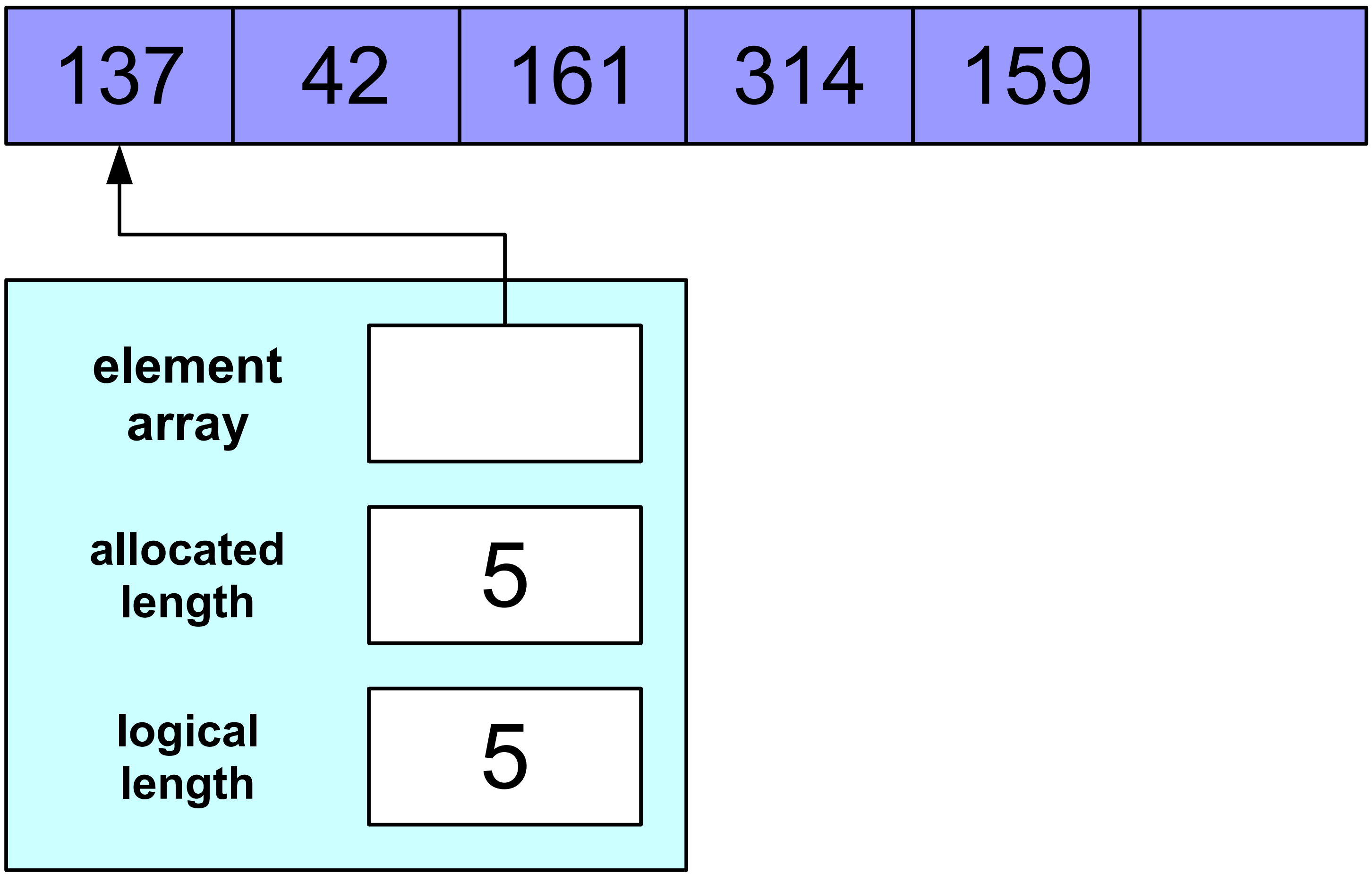
An Initial Idea



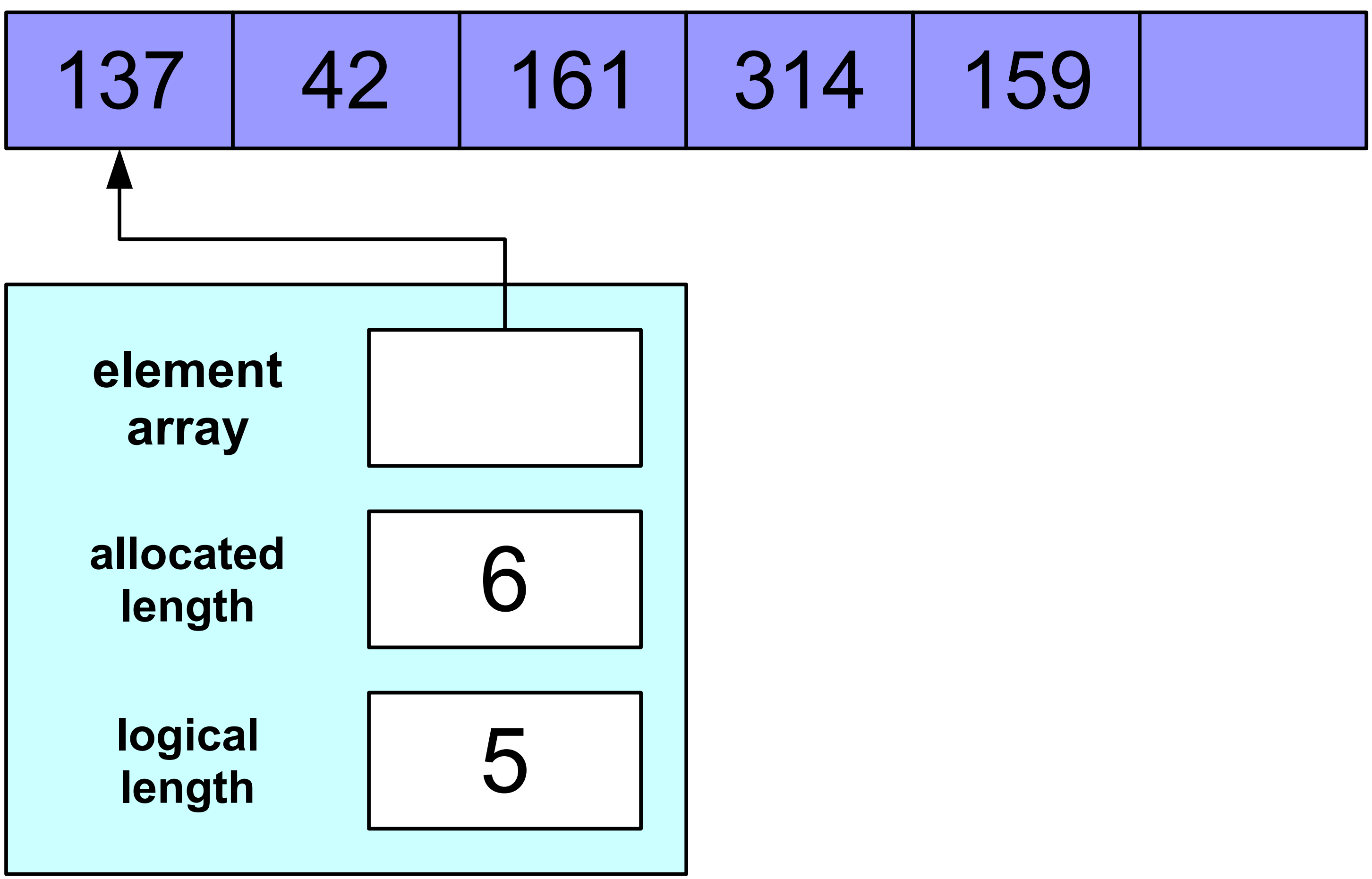
An Initial Idea



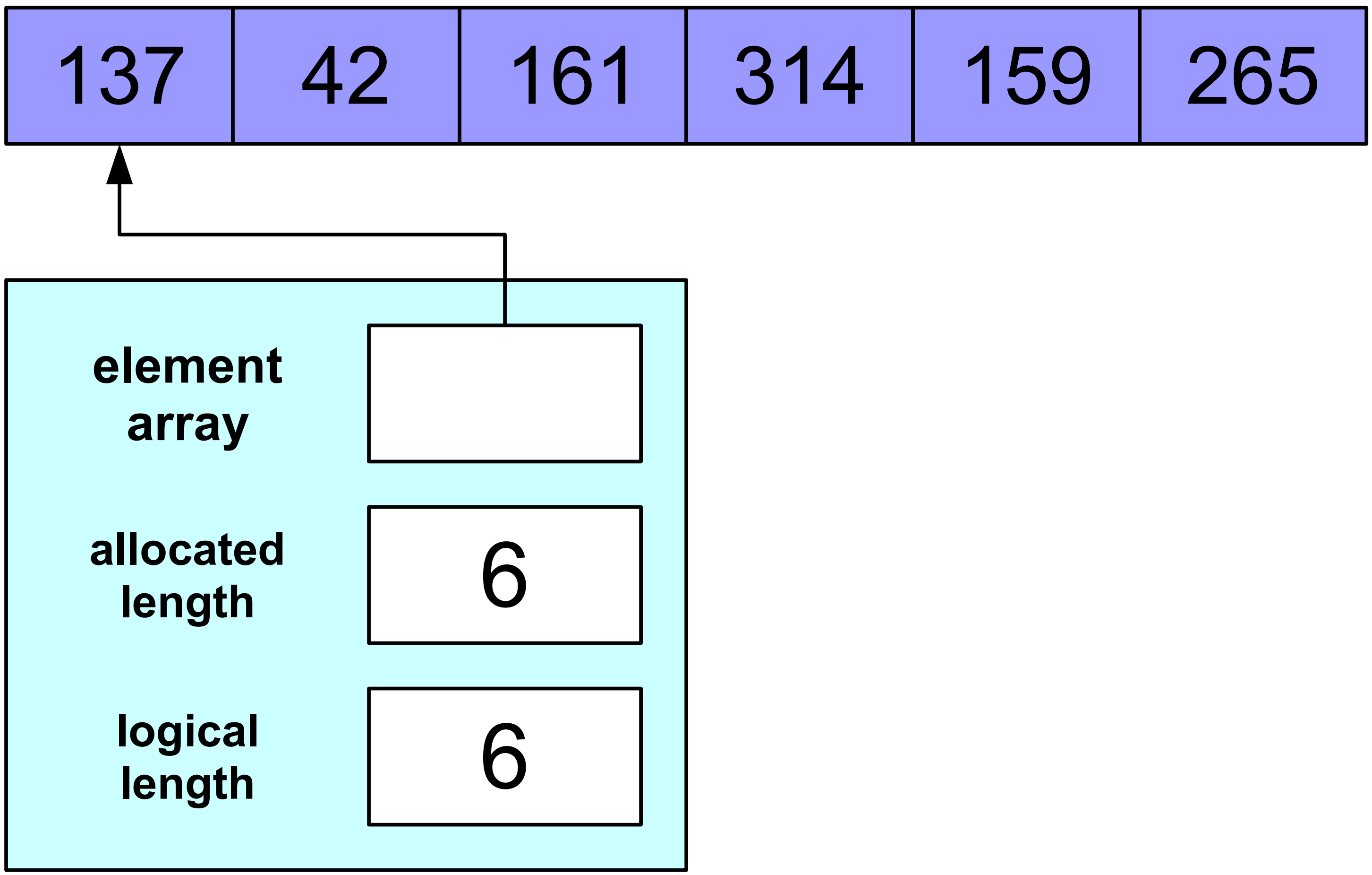
An Initial Idea



An Initial Idea



An Initial Idea



OurStack::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.
  - **`size`**:
  - **`isEmpty`**:
  - **`push`**:
  - **`pop`**:
  - **`top`**:

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:
  - $1 + 2 + 3 + 4 + \ldots + n$

What This Means

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

What This Means

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

What This Means

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

What This Means

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

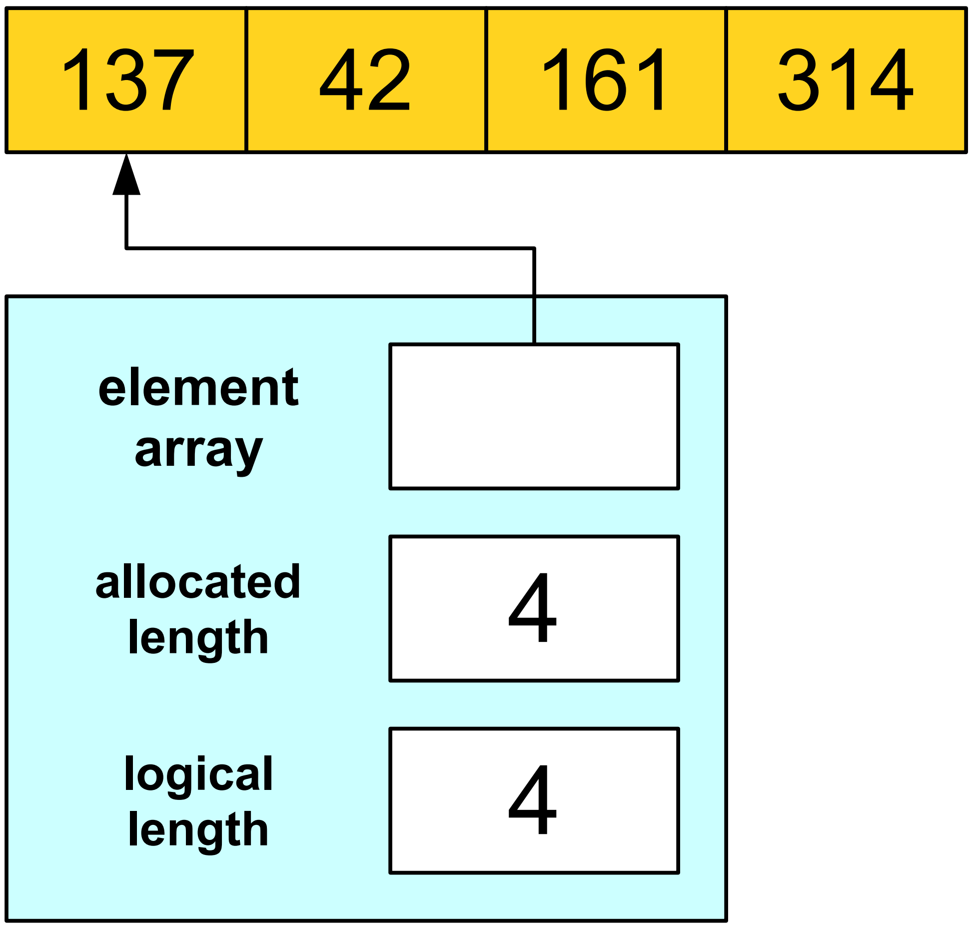
What This Means

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

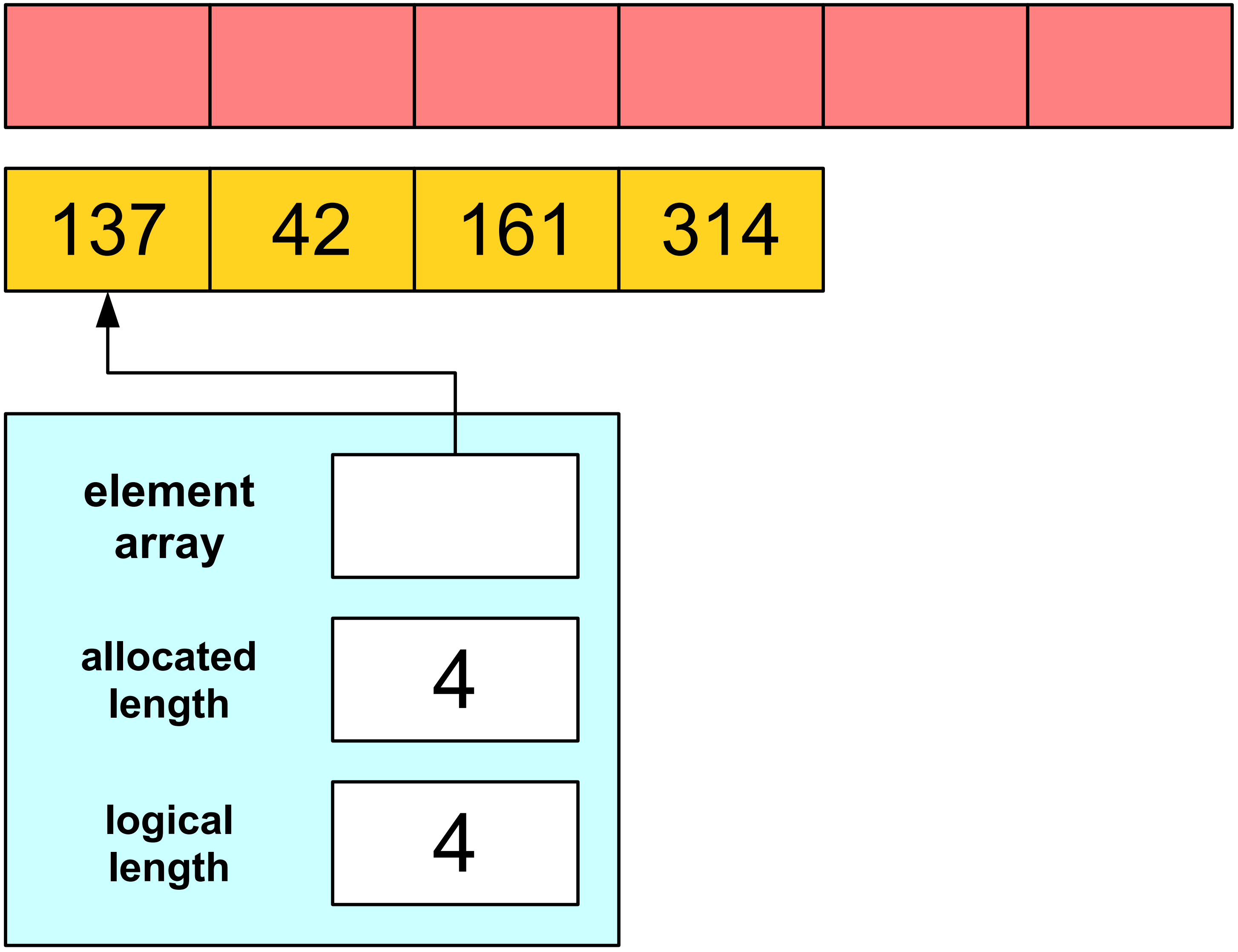
Validating Our Model

Speeding up the Stack

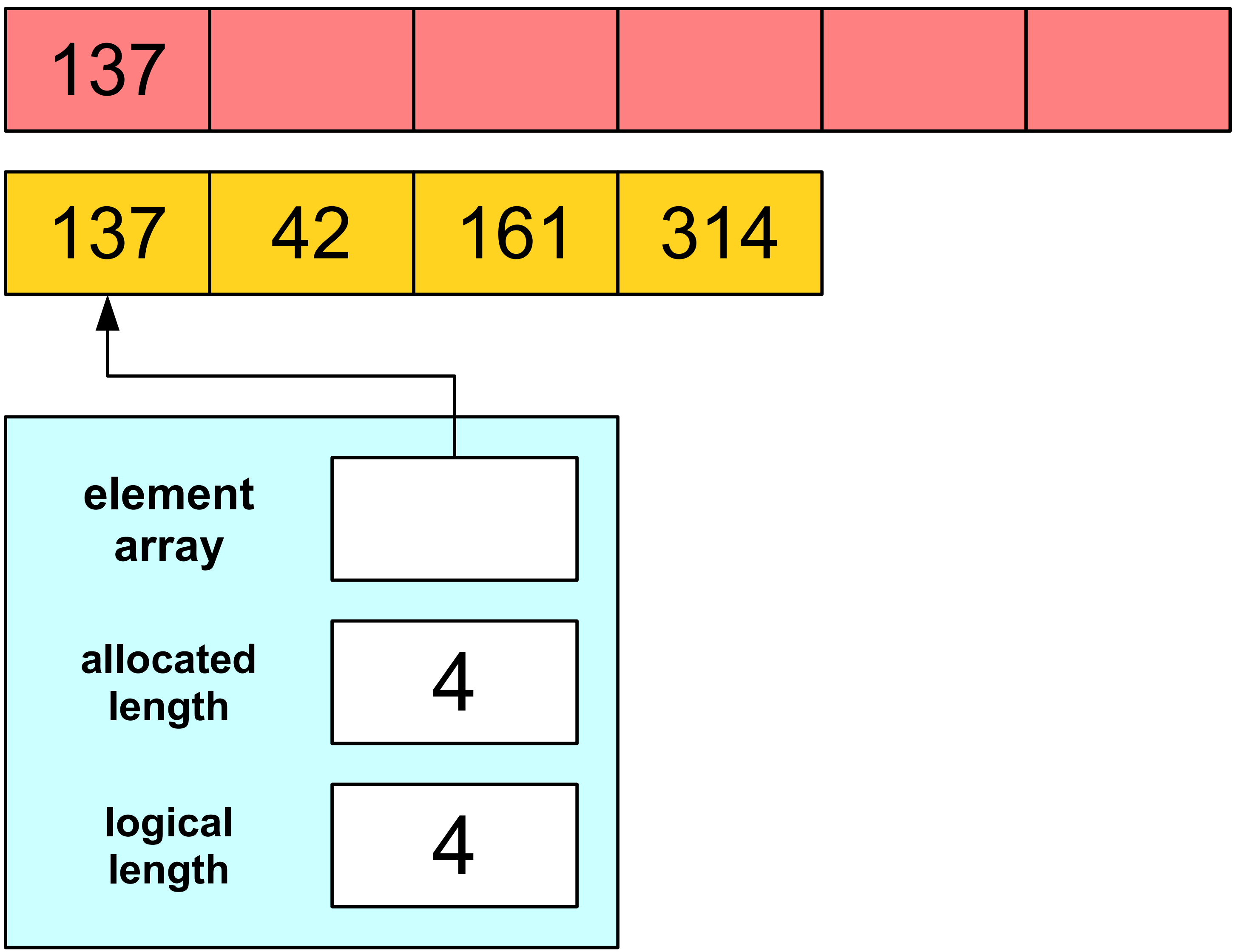
A Better Idea



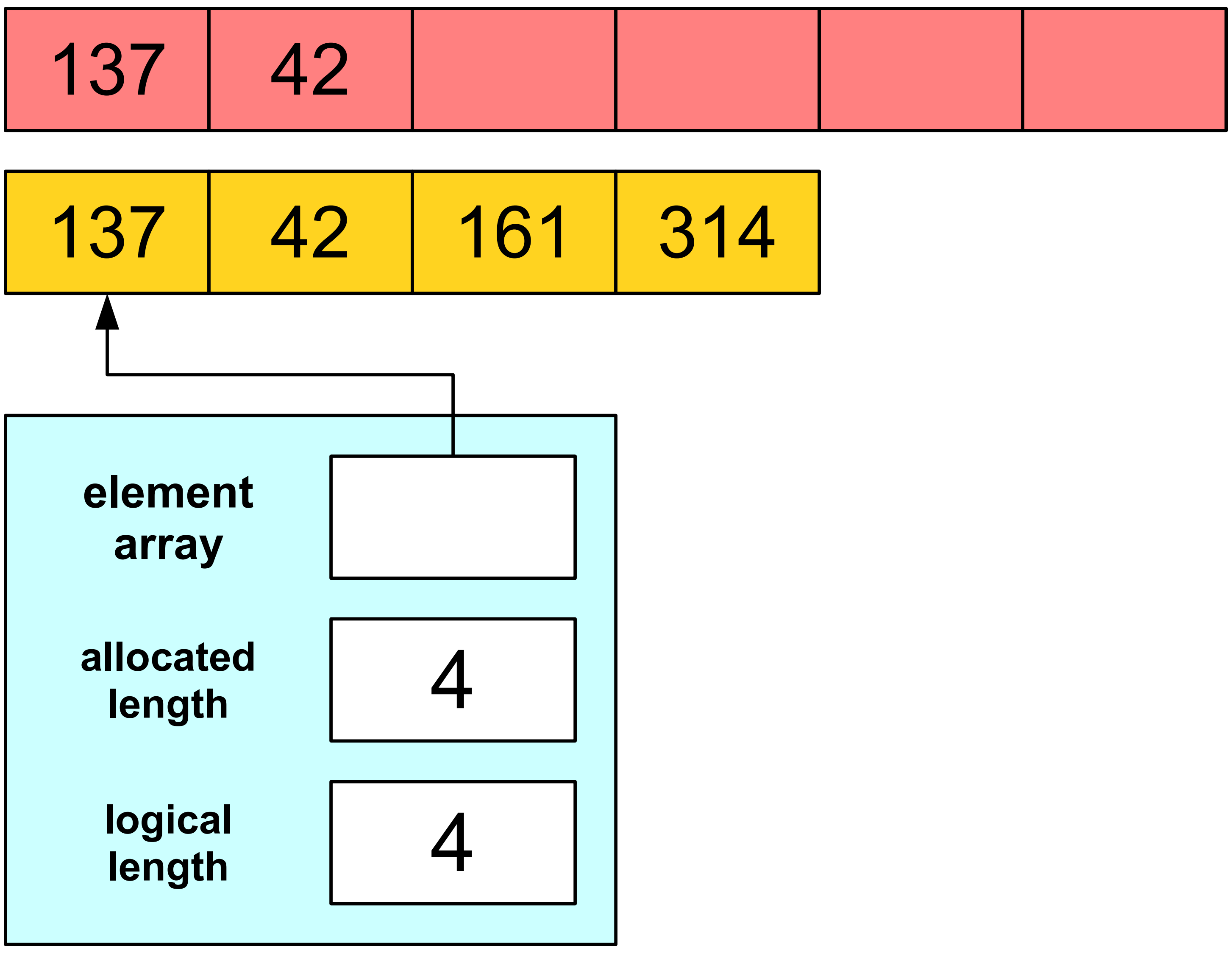
A Better Idea



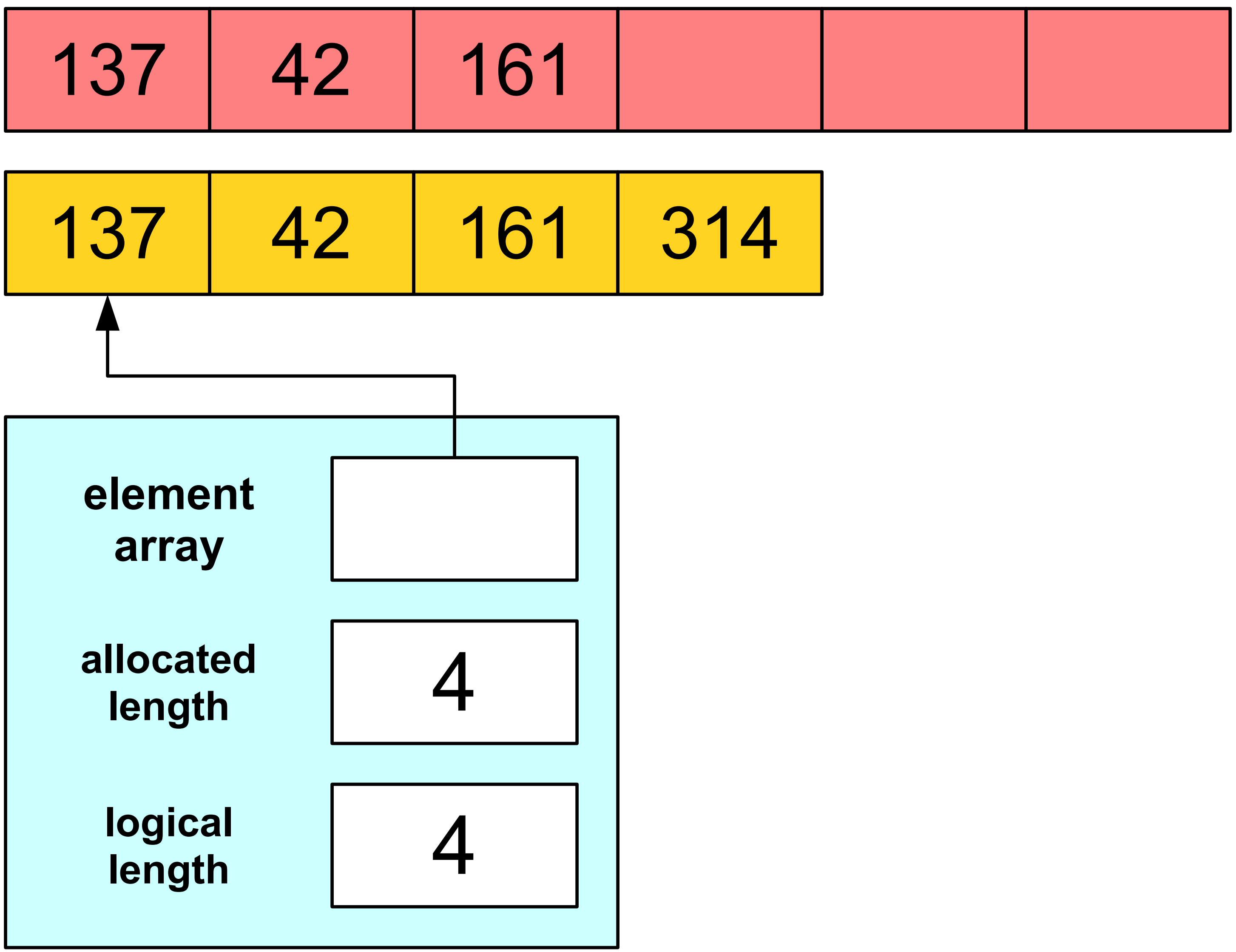
A Better Idea



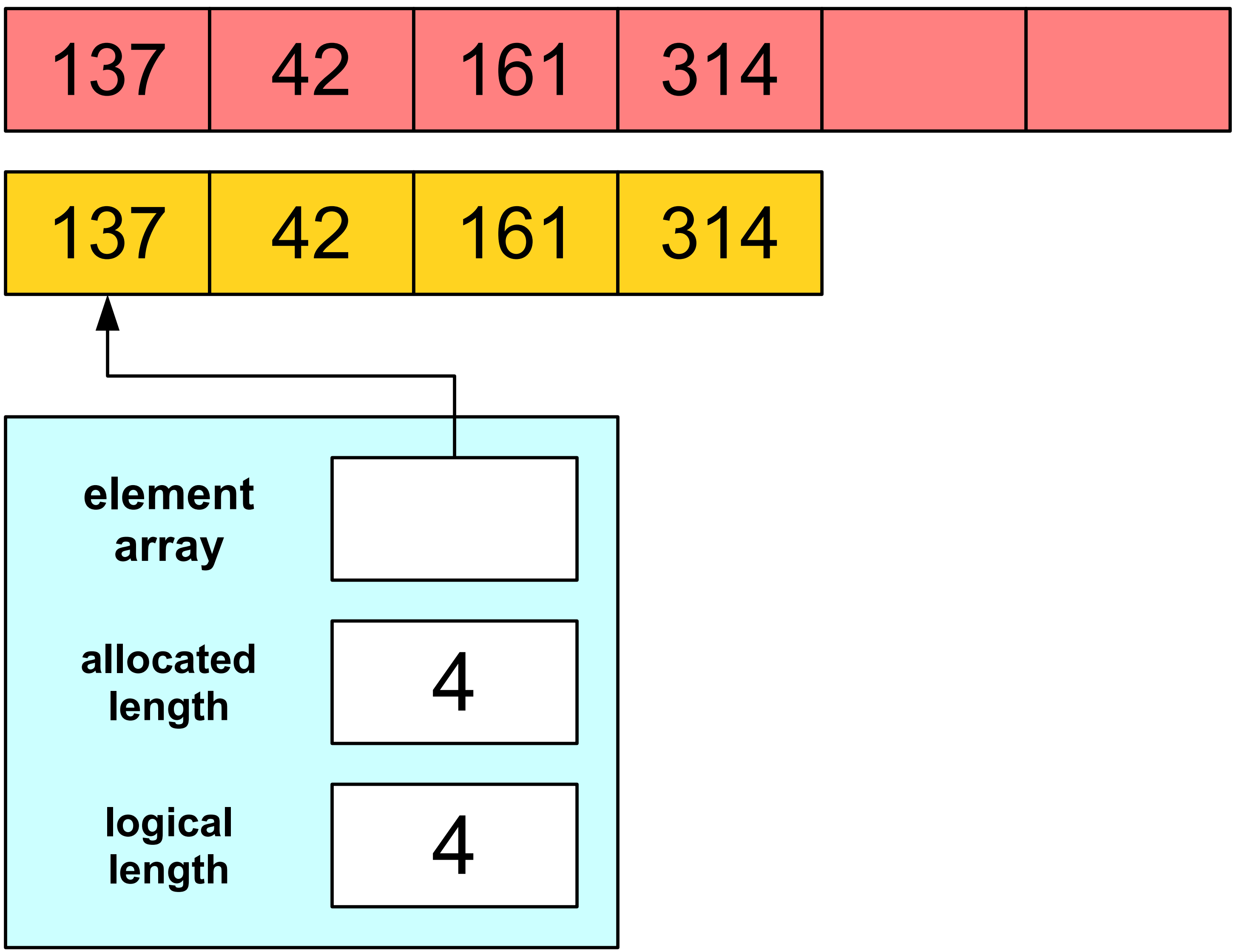
A Better Idea



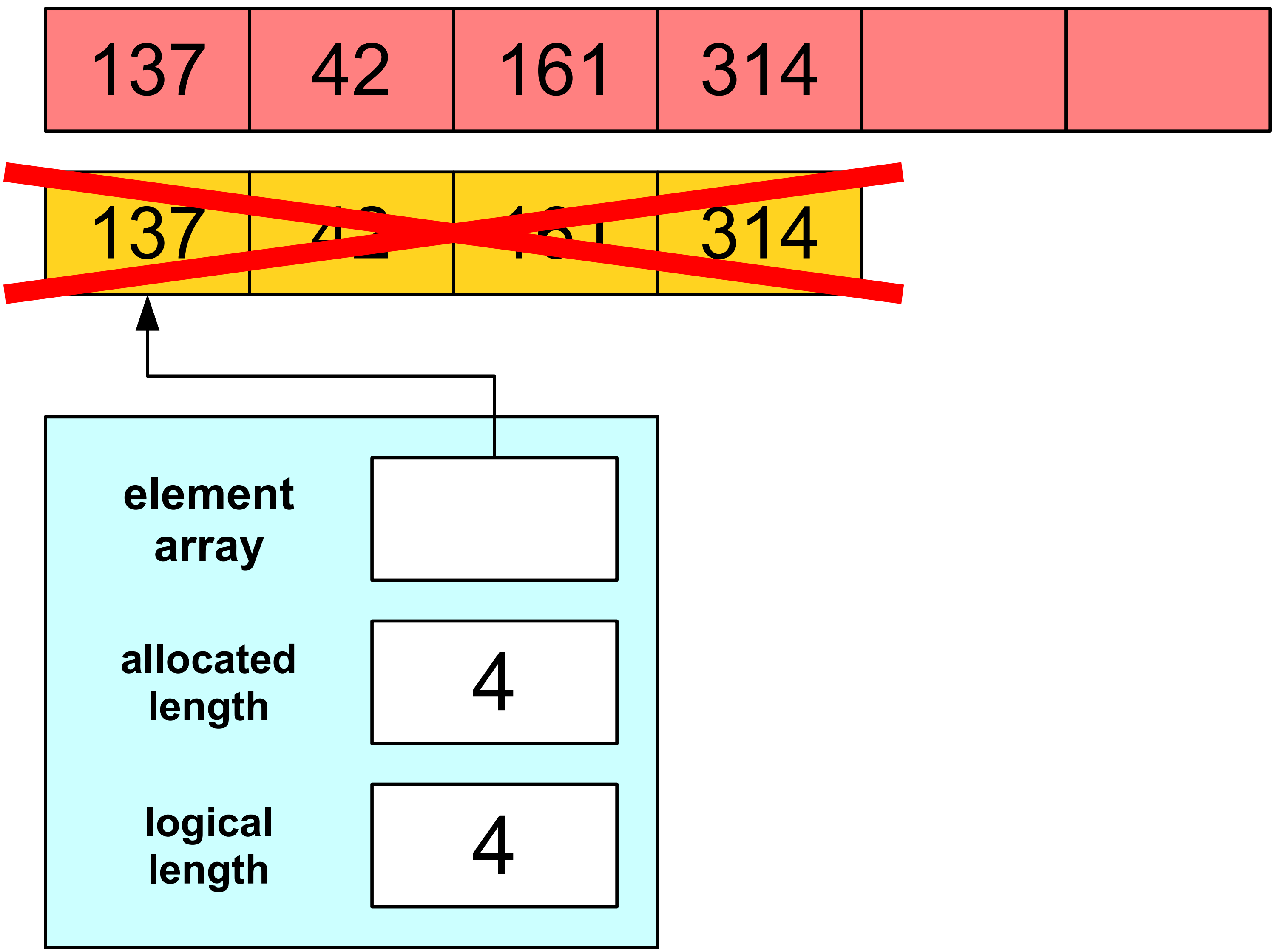
A Better Idea



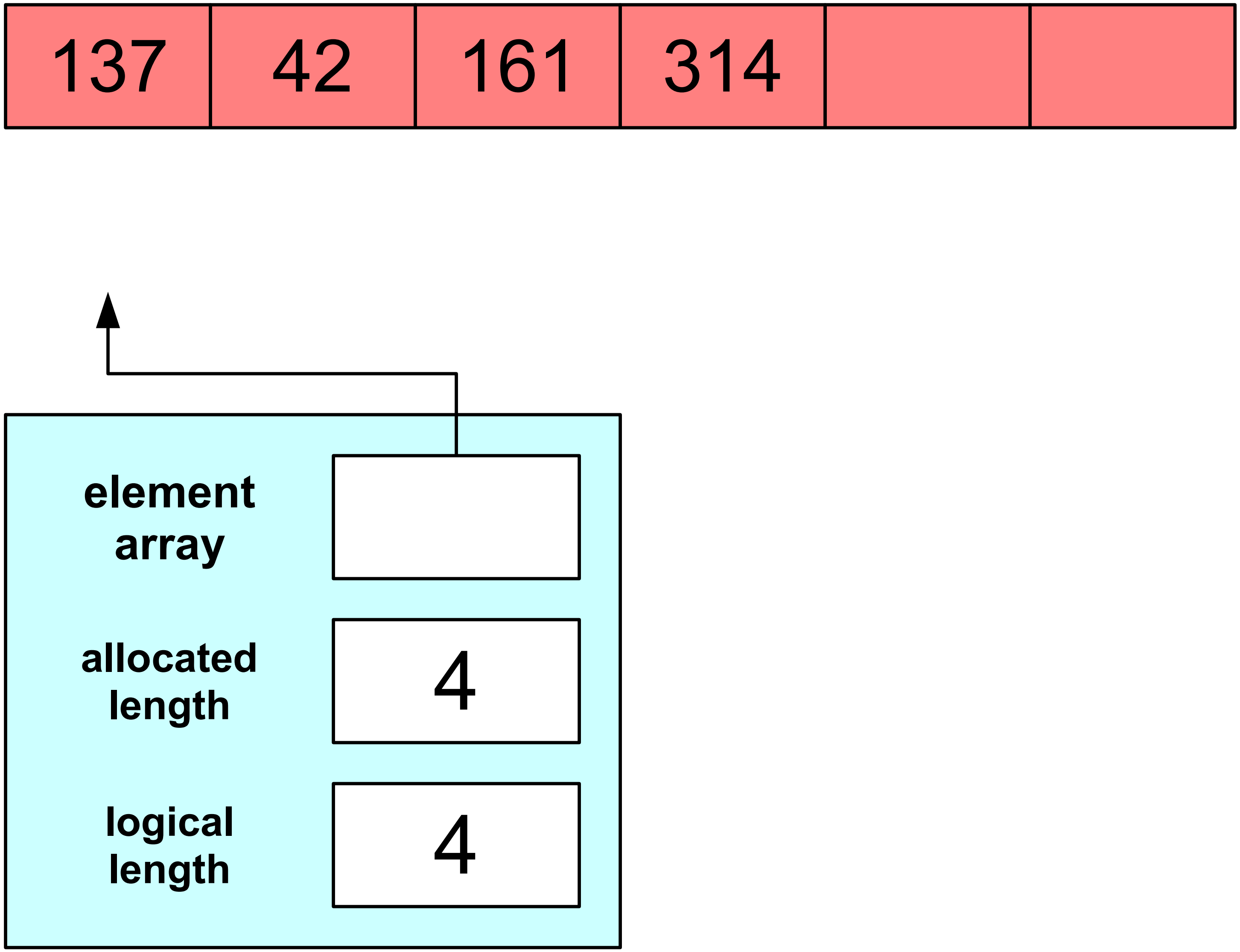
A Better Idea



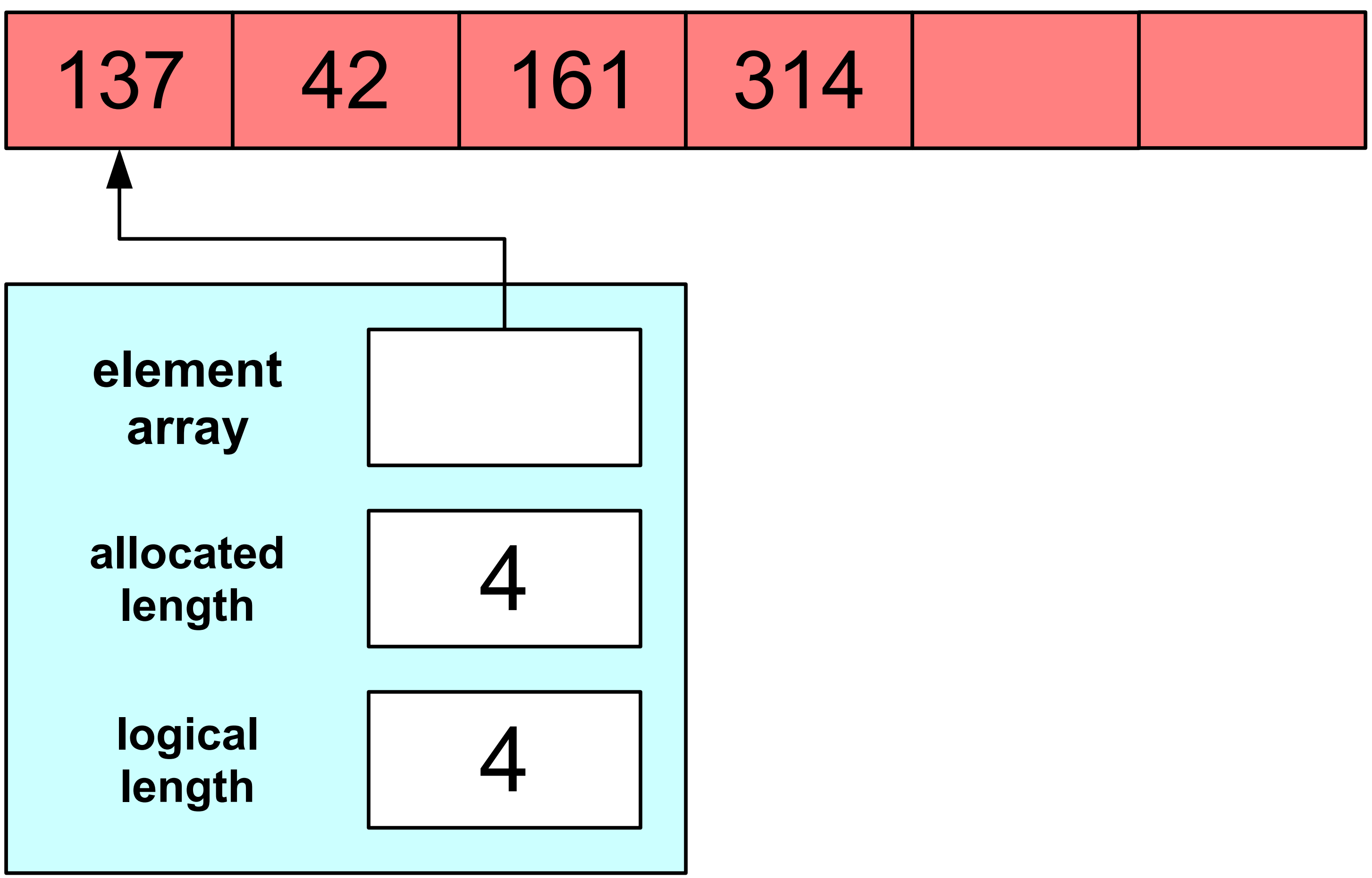
A Better Idea



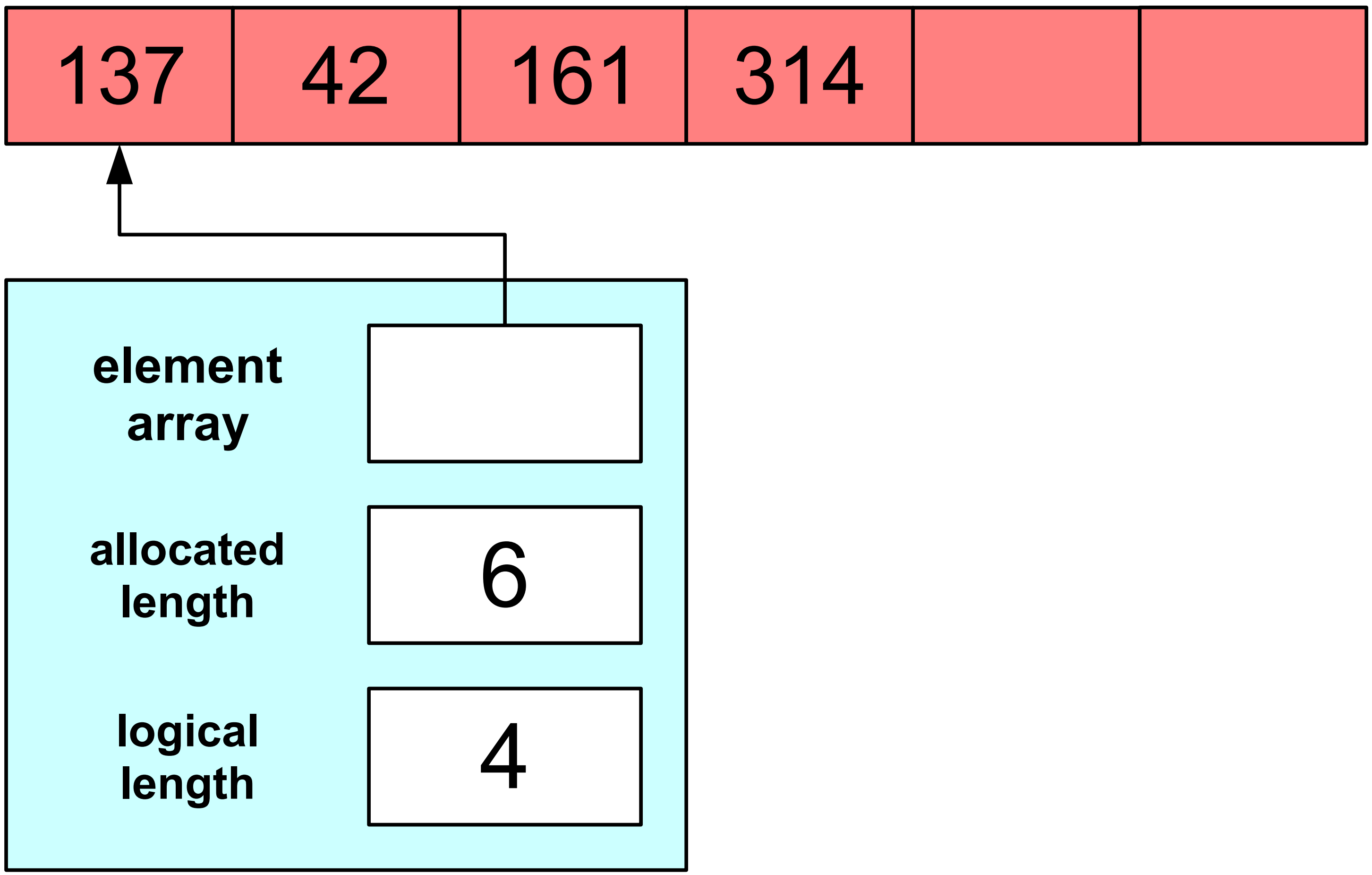
A Better Idea



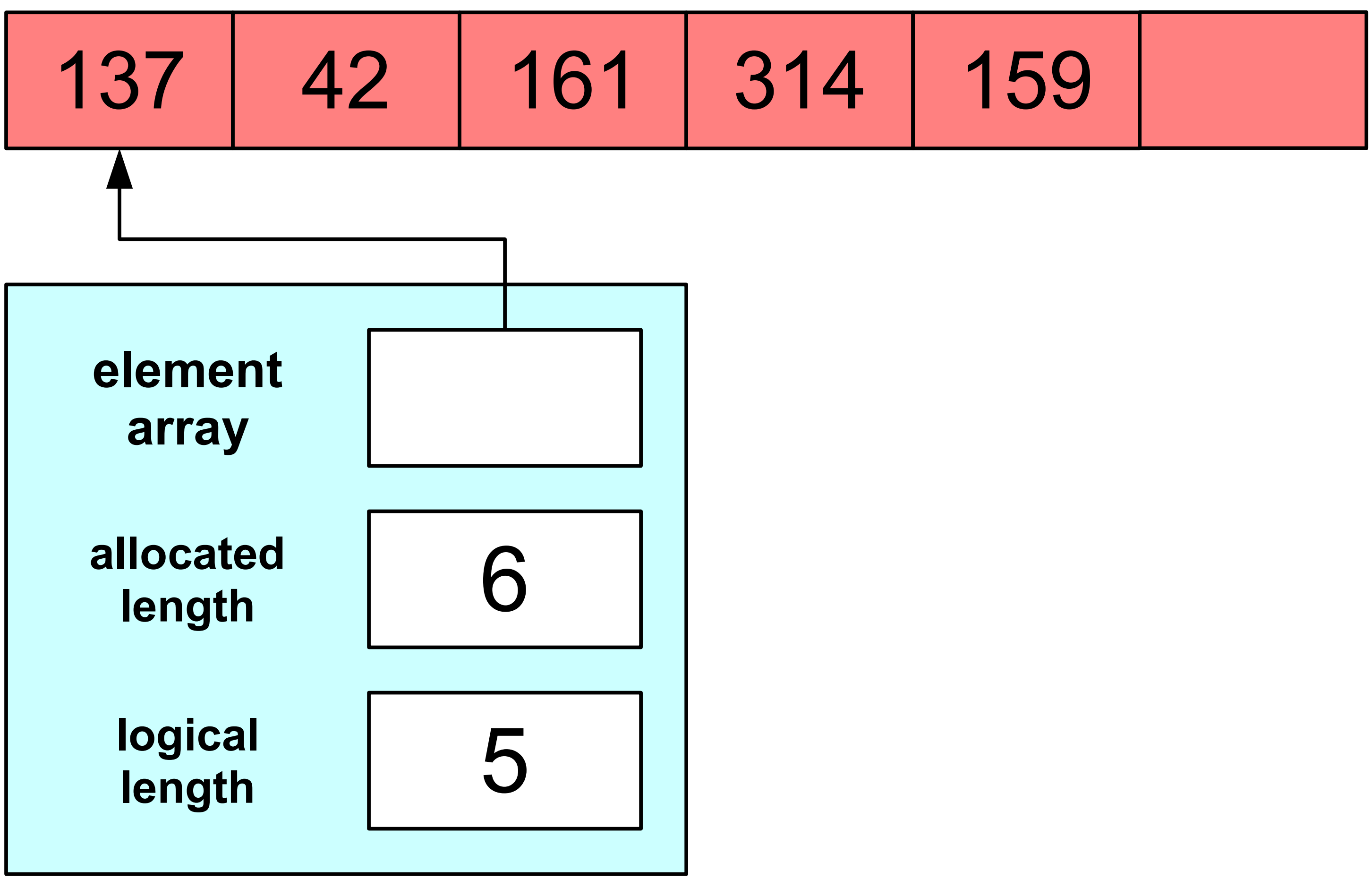
A Better Idea



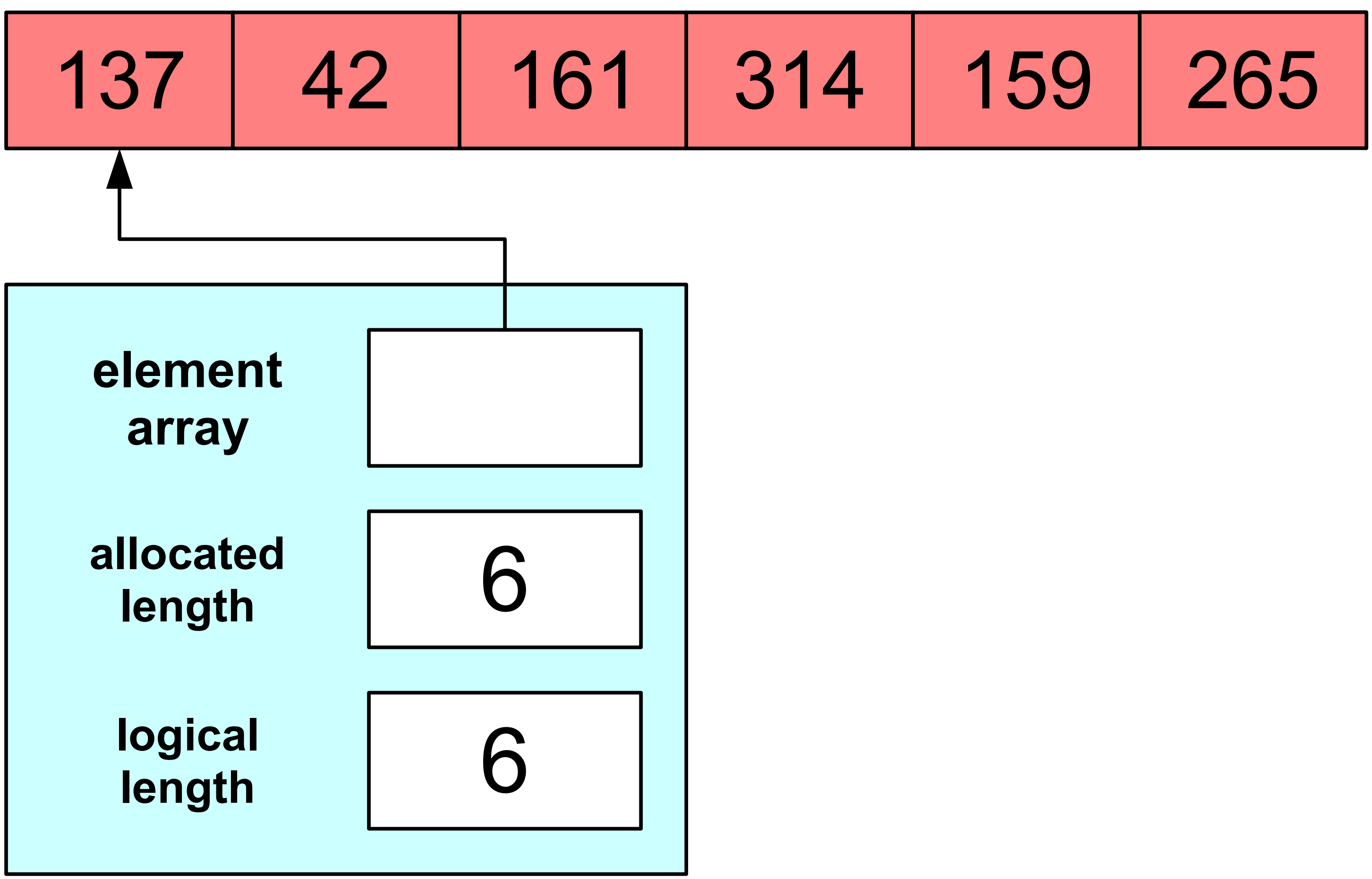
A Better Idea



A Better Idea



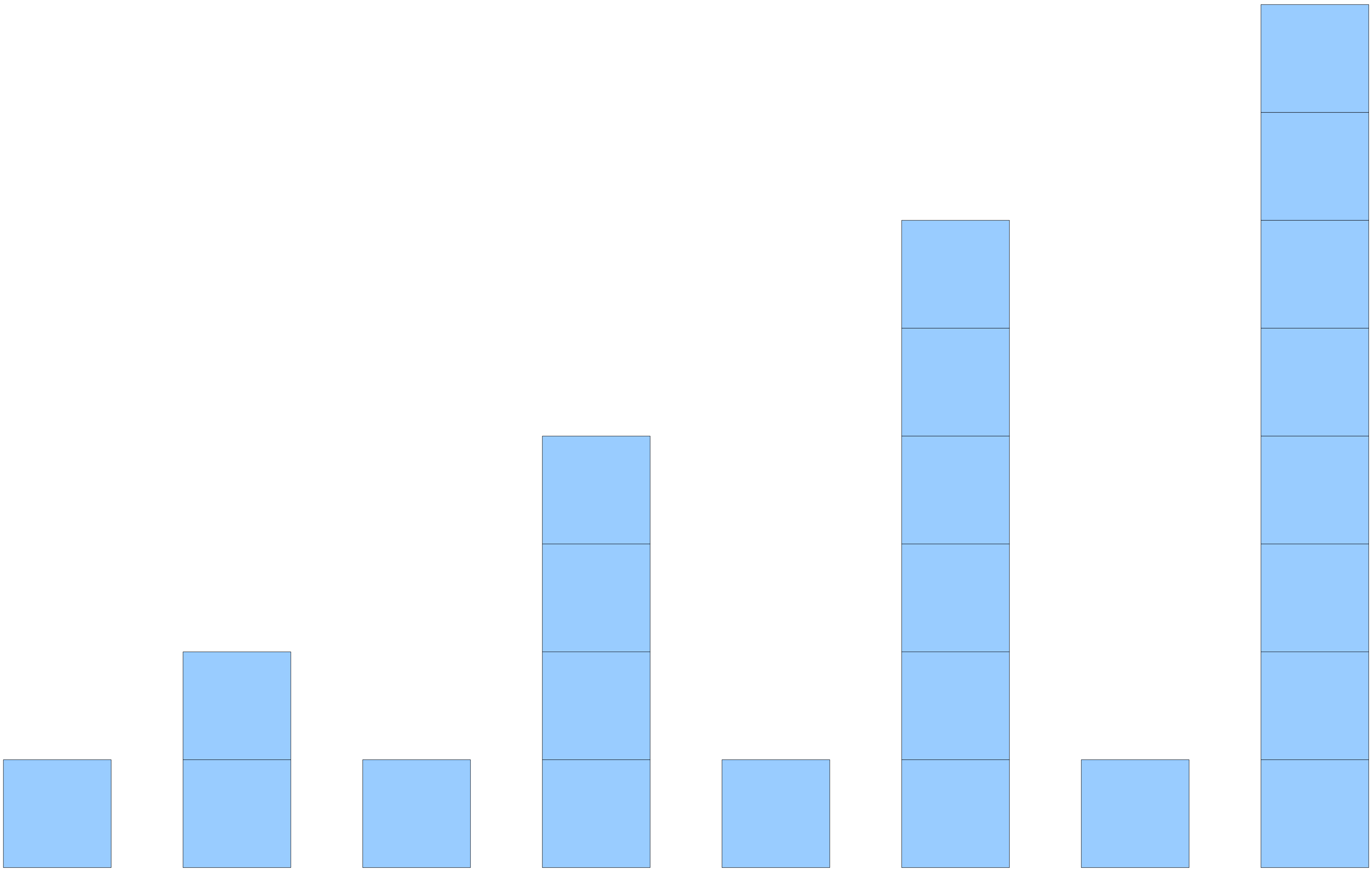
A Better Idea



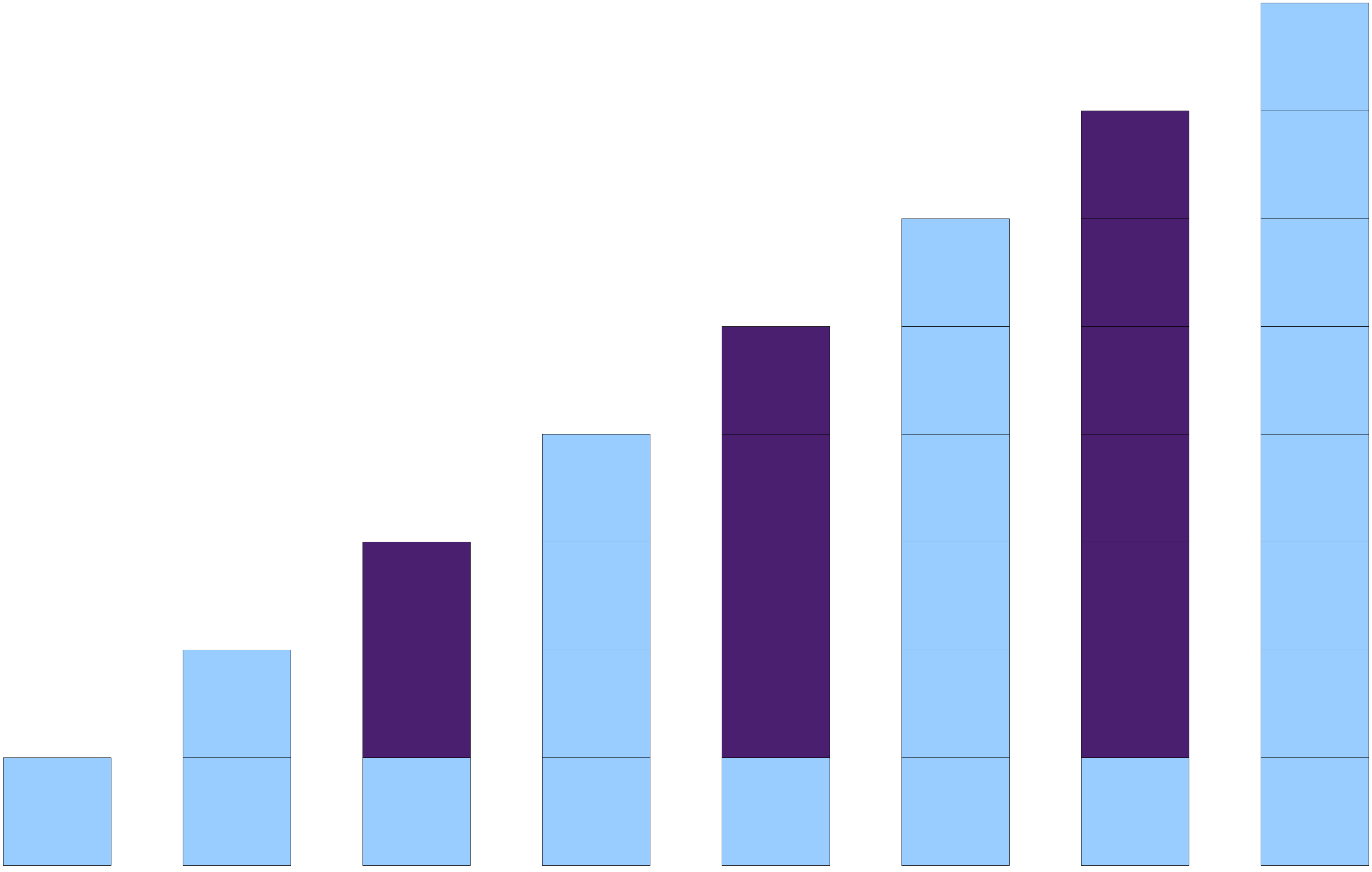
What Just Happened?

- 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?

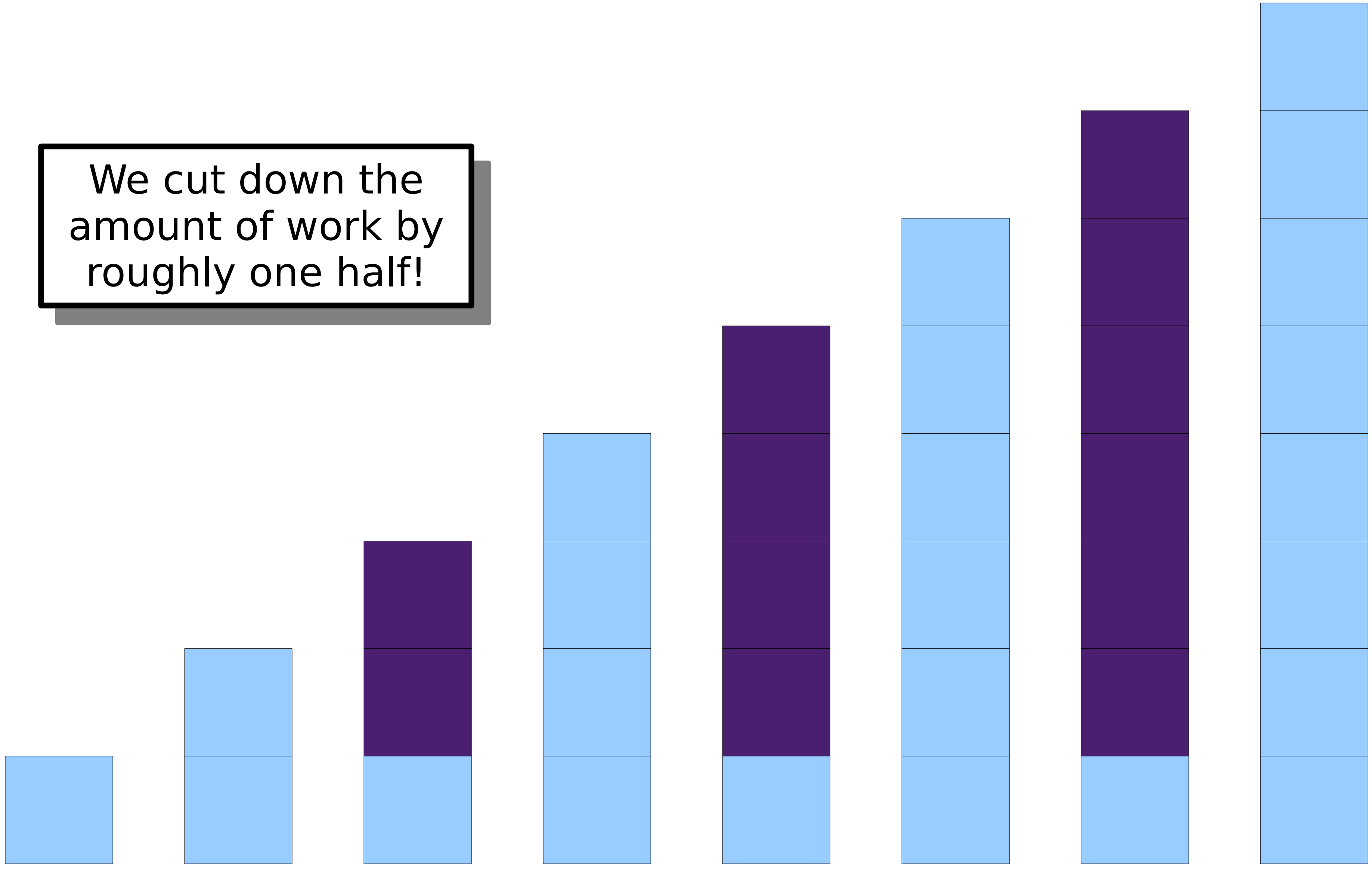
Analyzing the Work



Analyzing the Work

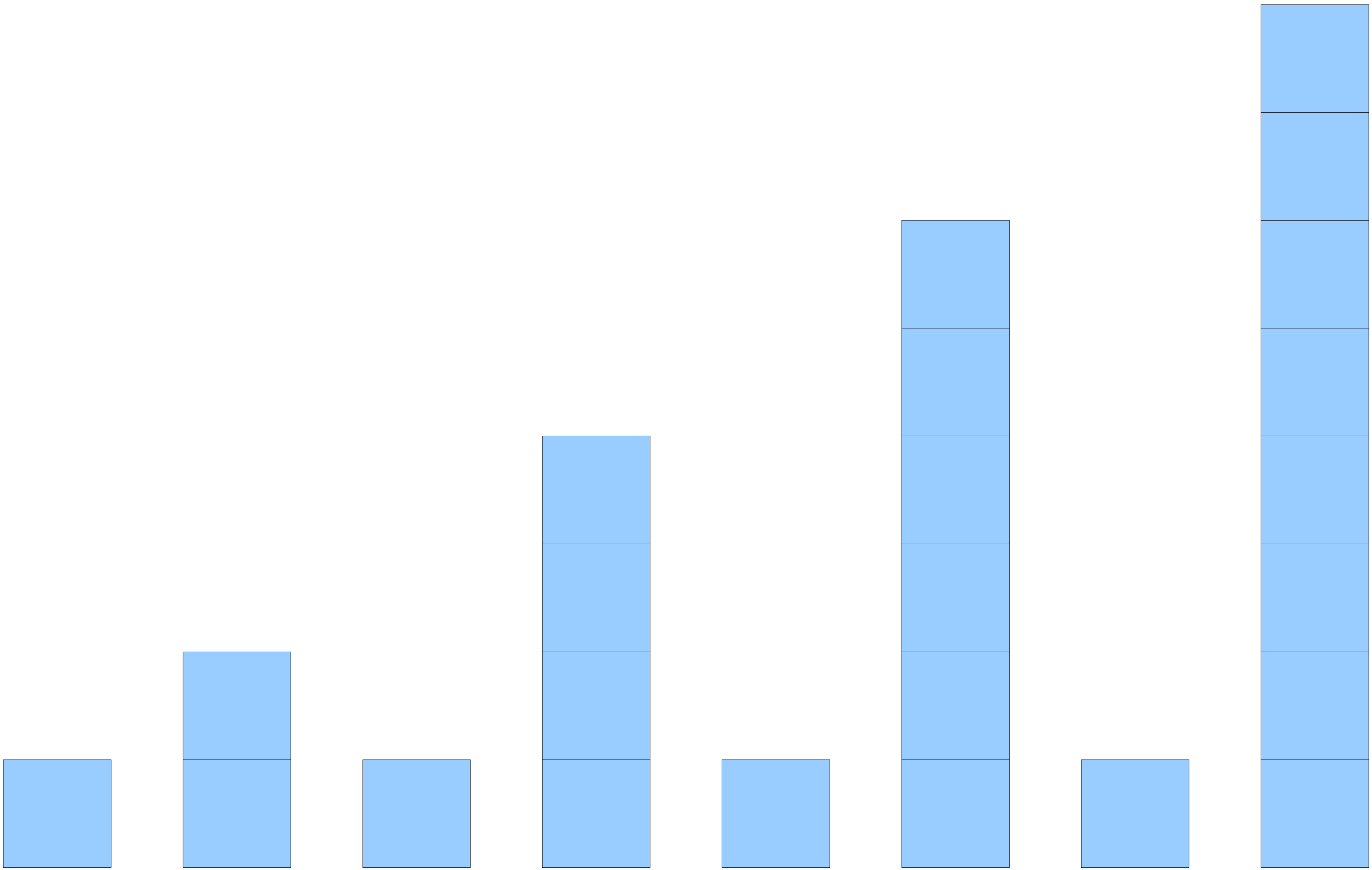


Analyzing the Work

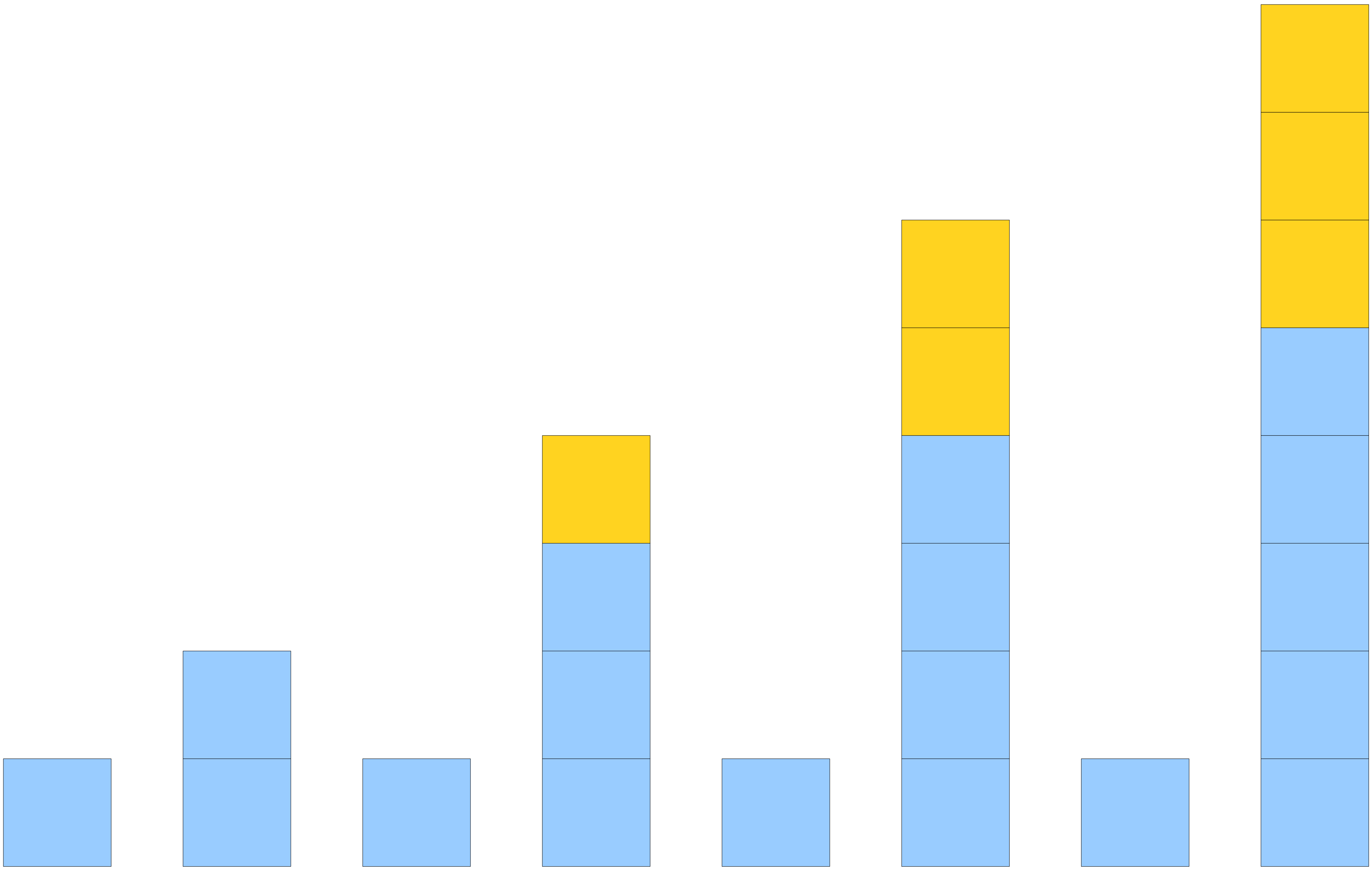


A Different Analysis

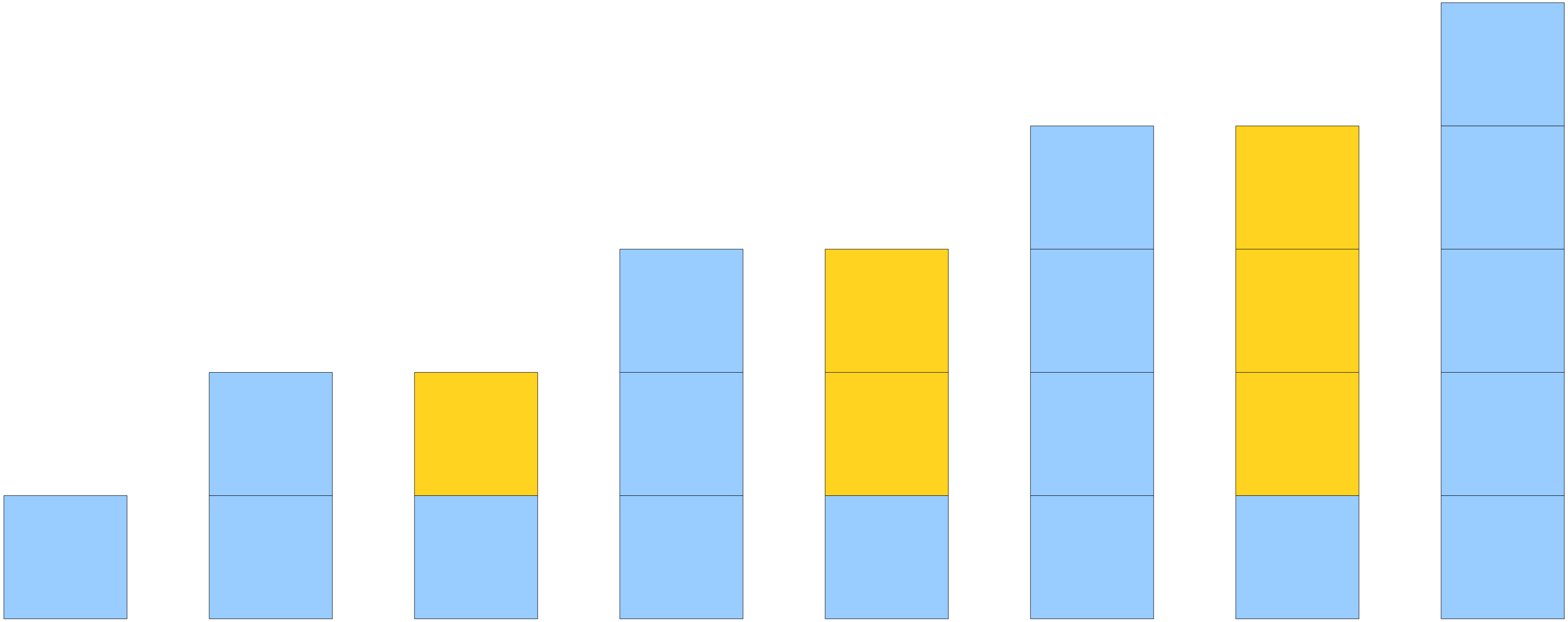
A Different Analysis



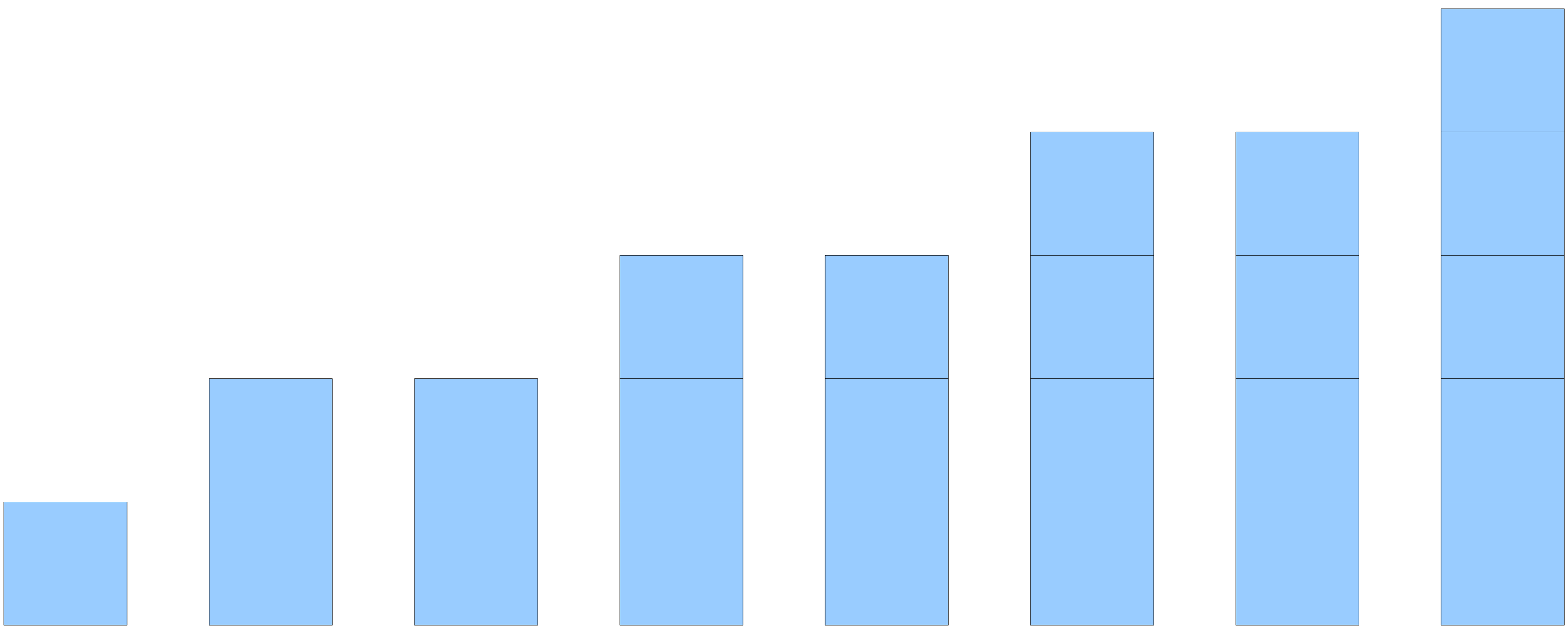
A Different Analysis



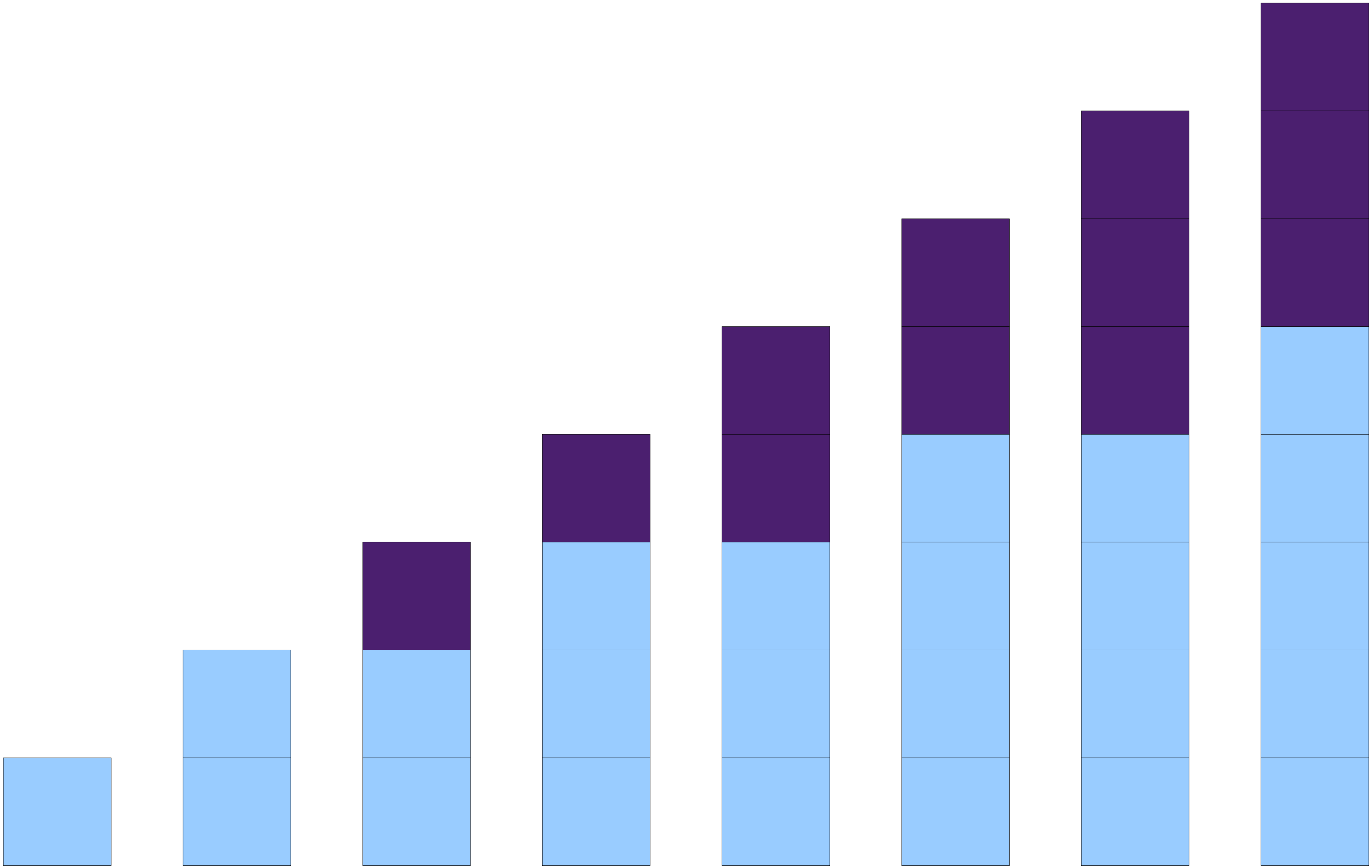
A Different Analysis



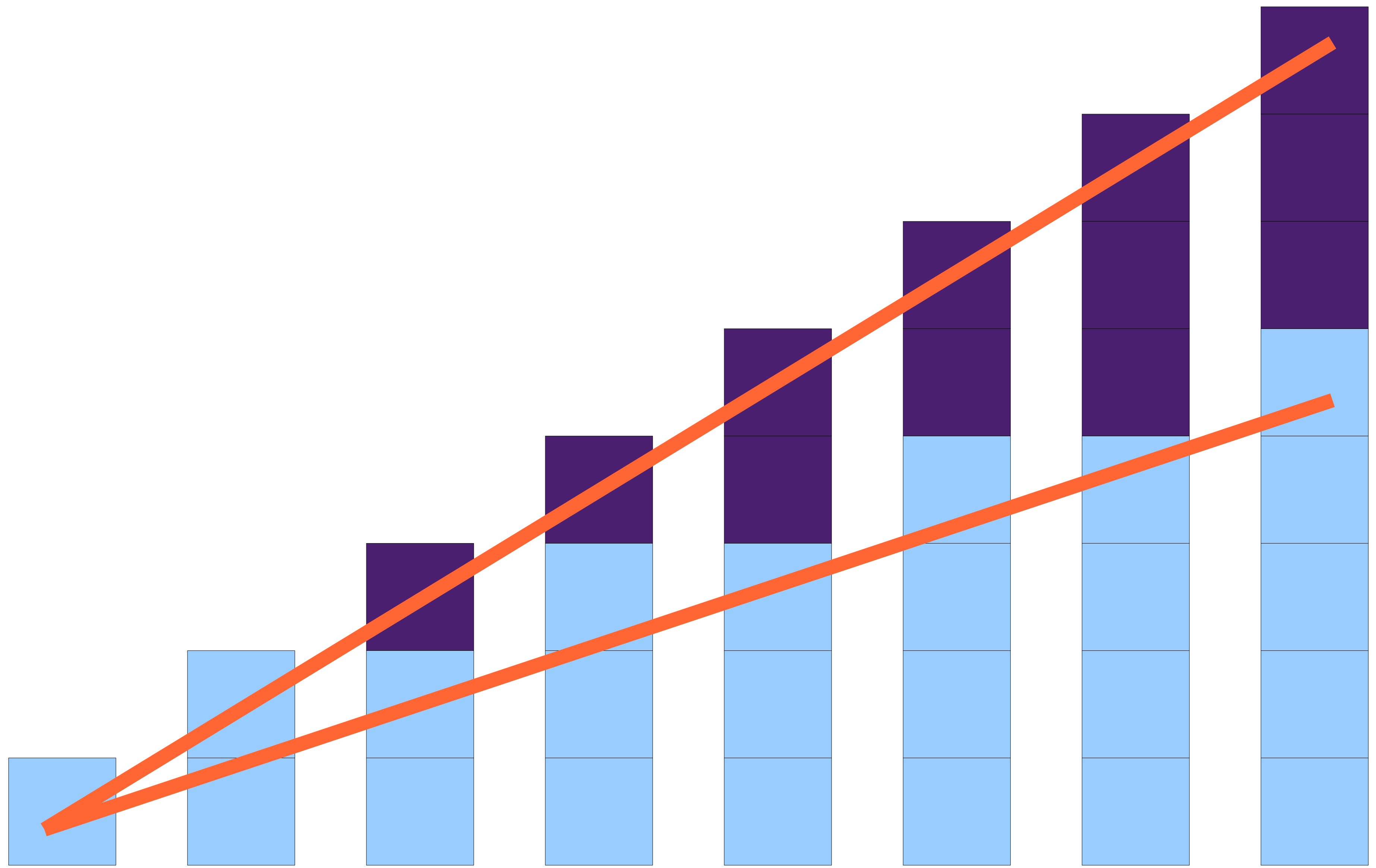
A Different Analysis



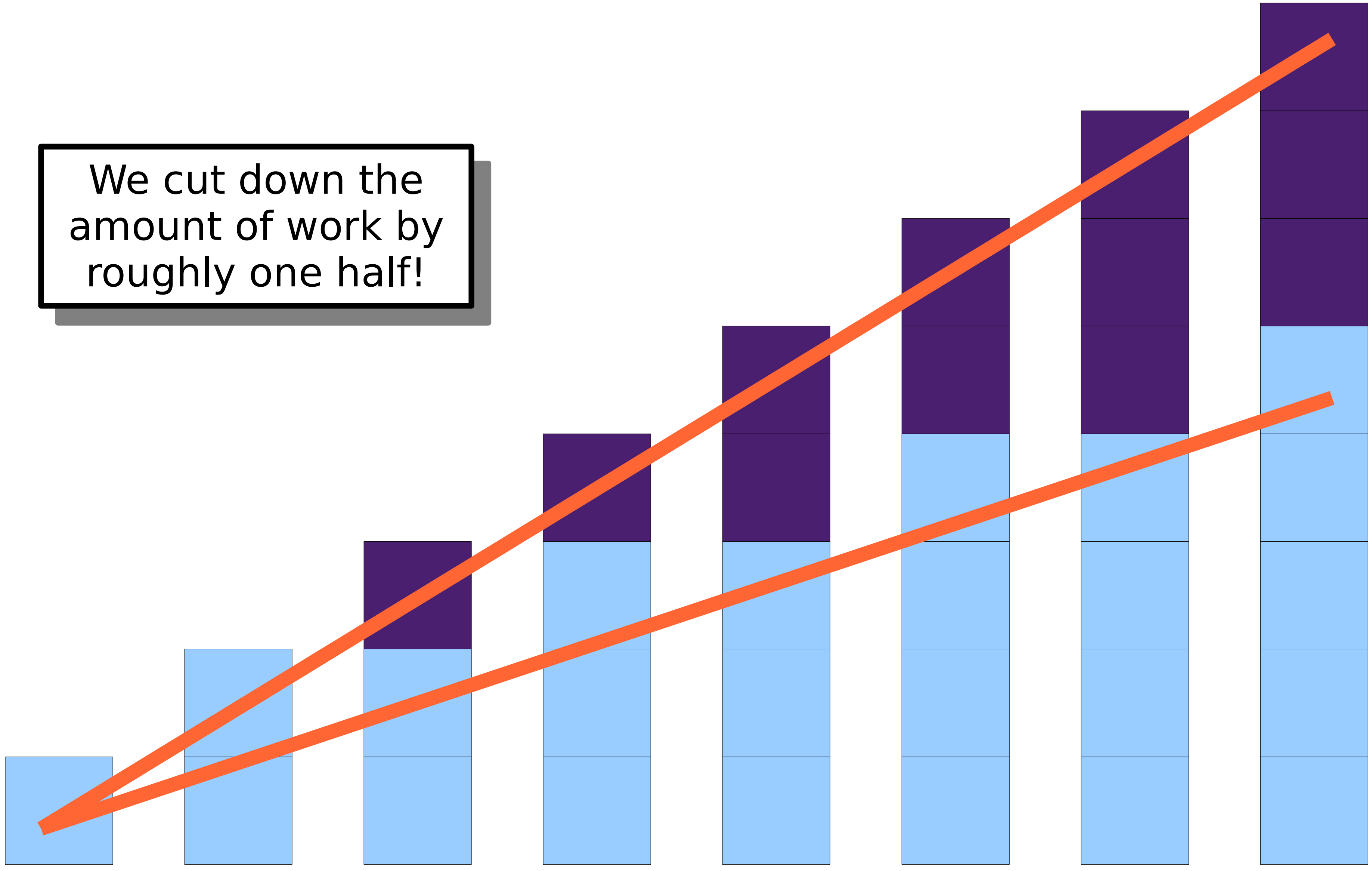
A Different Analysis



A Different Analysis

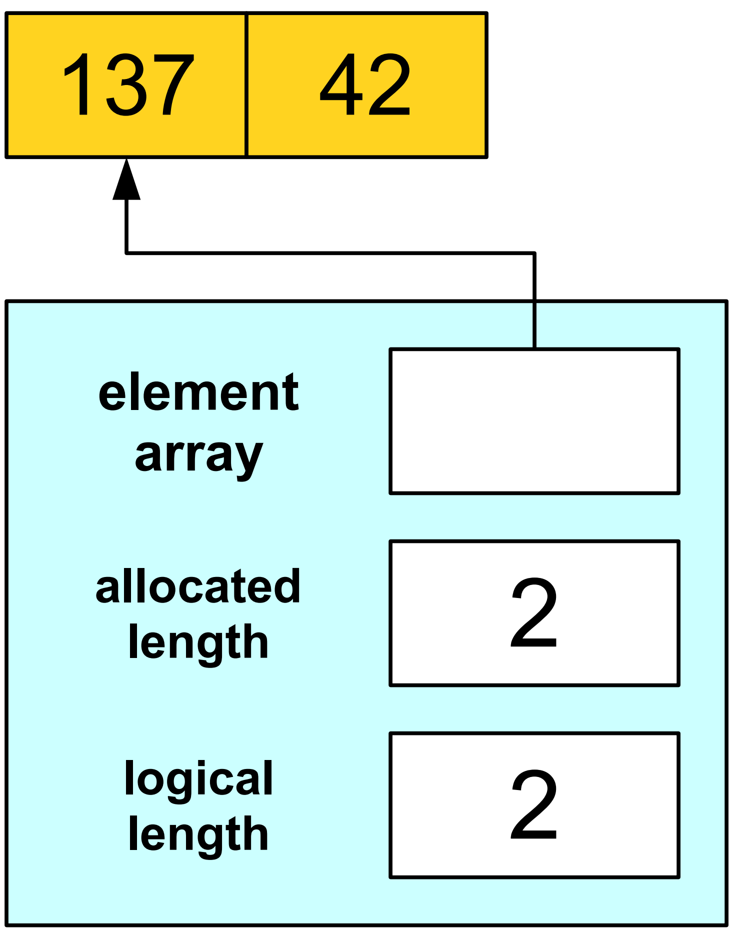


A Different Analysis

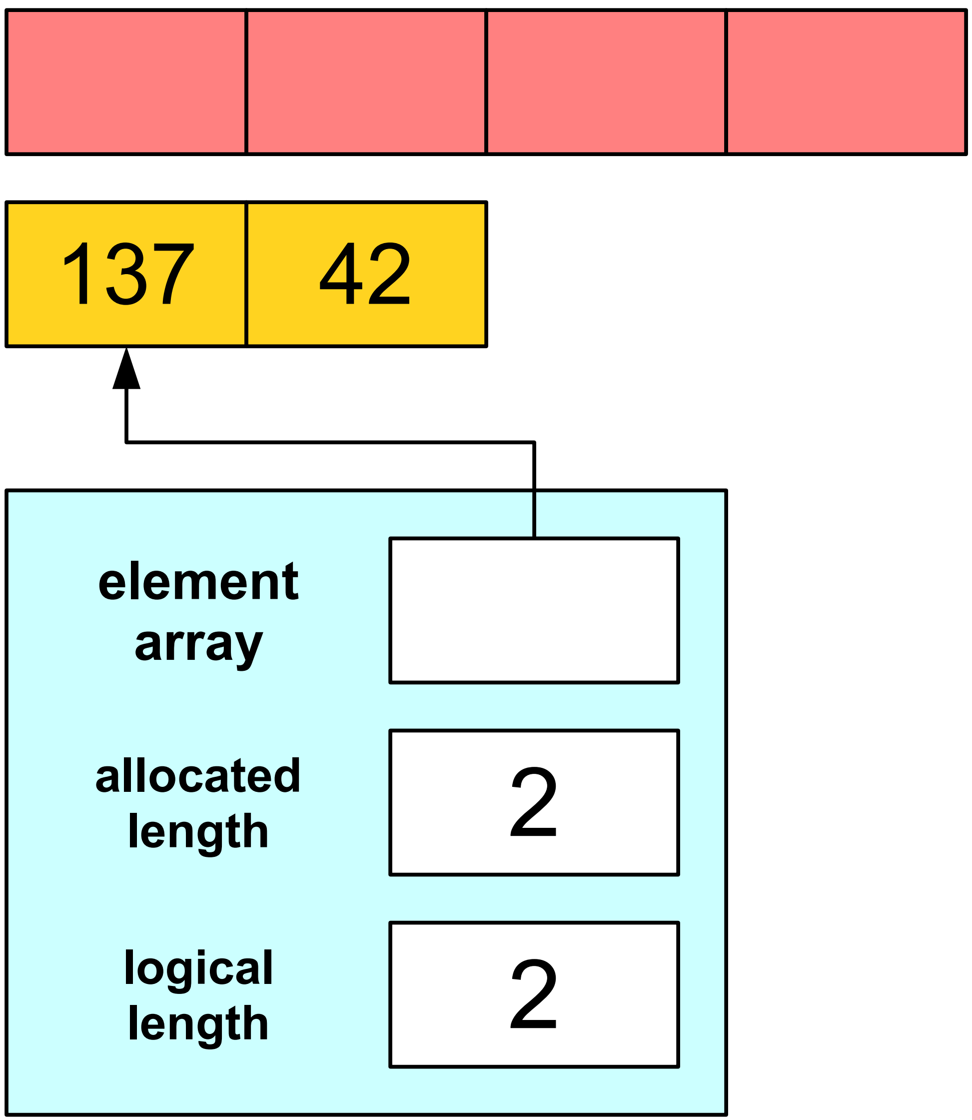


How does it stack up?

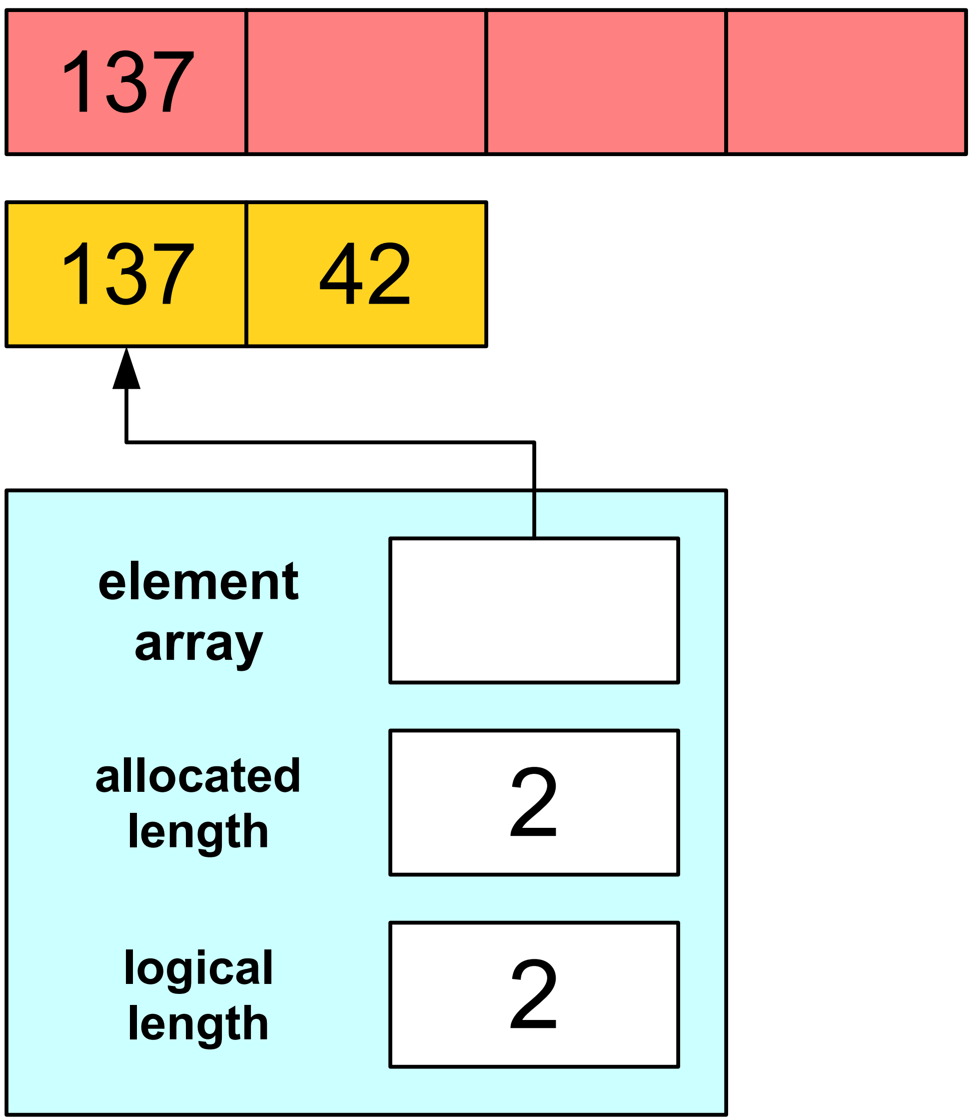
A Much Better Idea



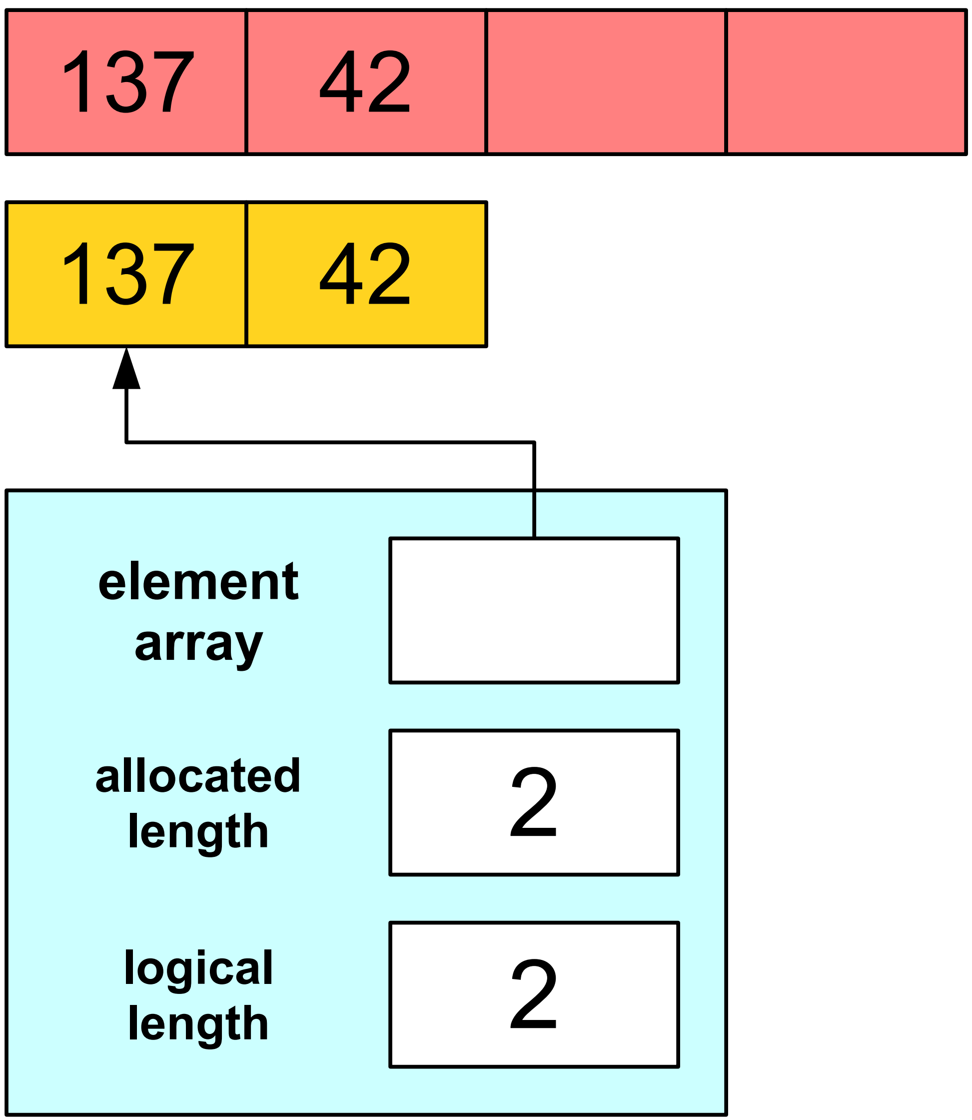
A Much Better Idea



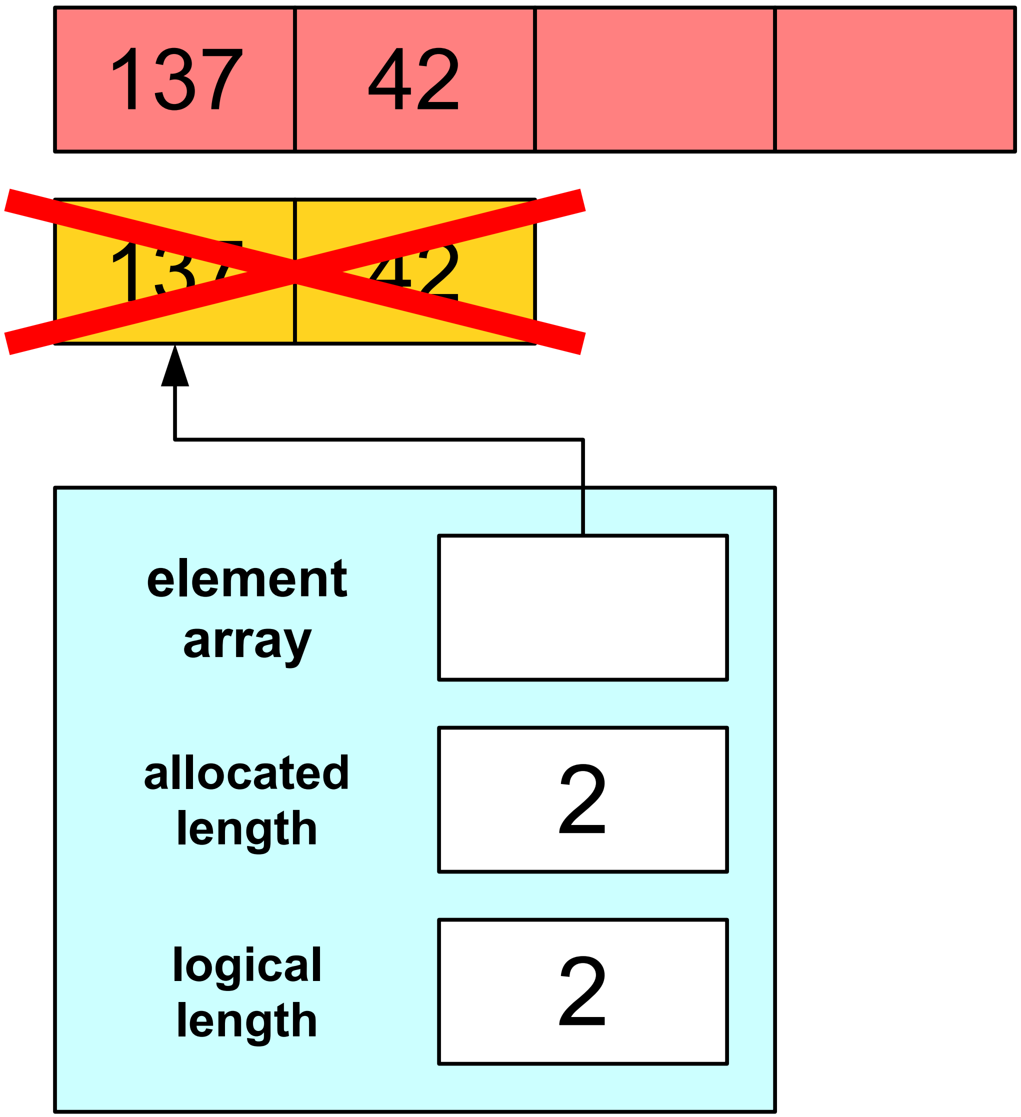
A Much Better Idea



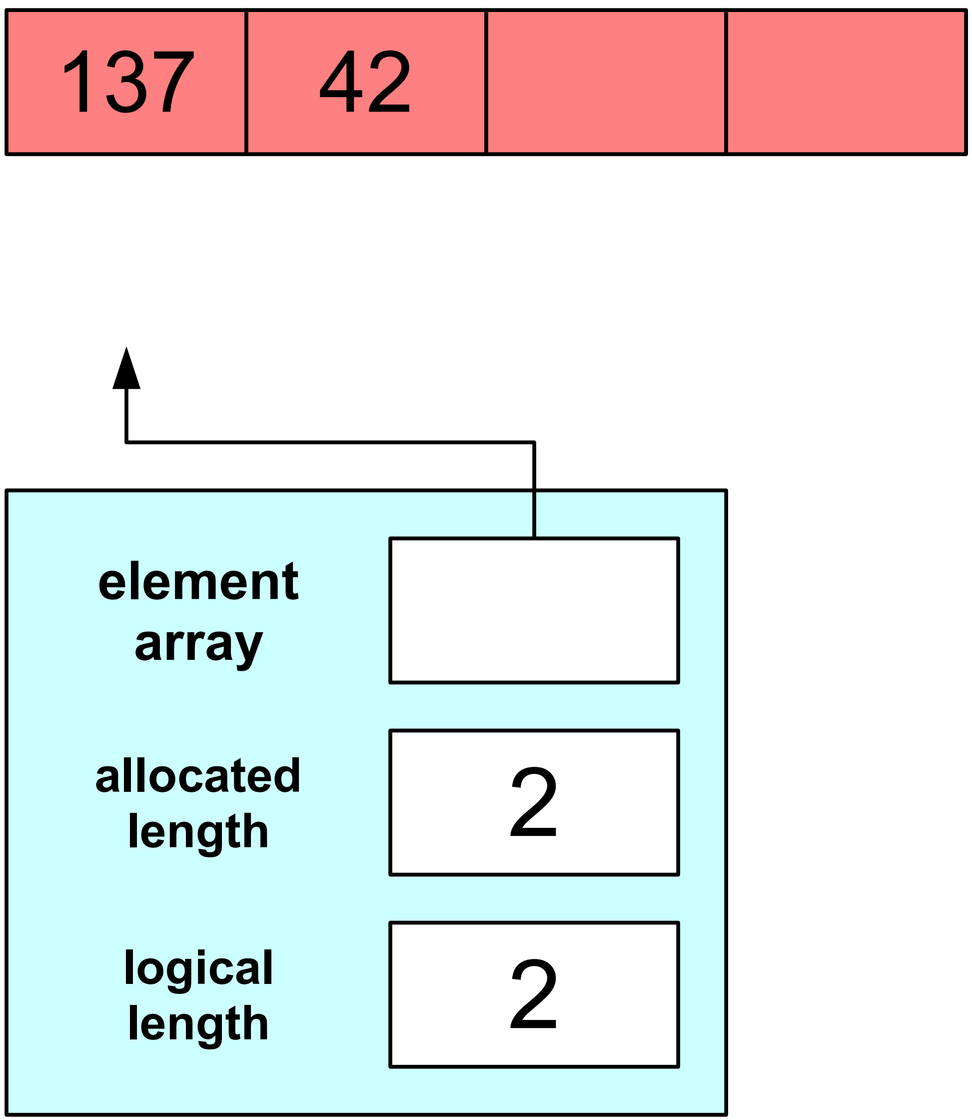
A Much Better Idea



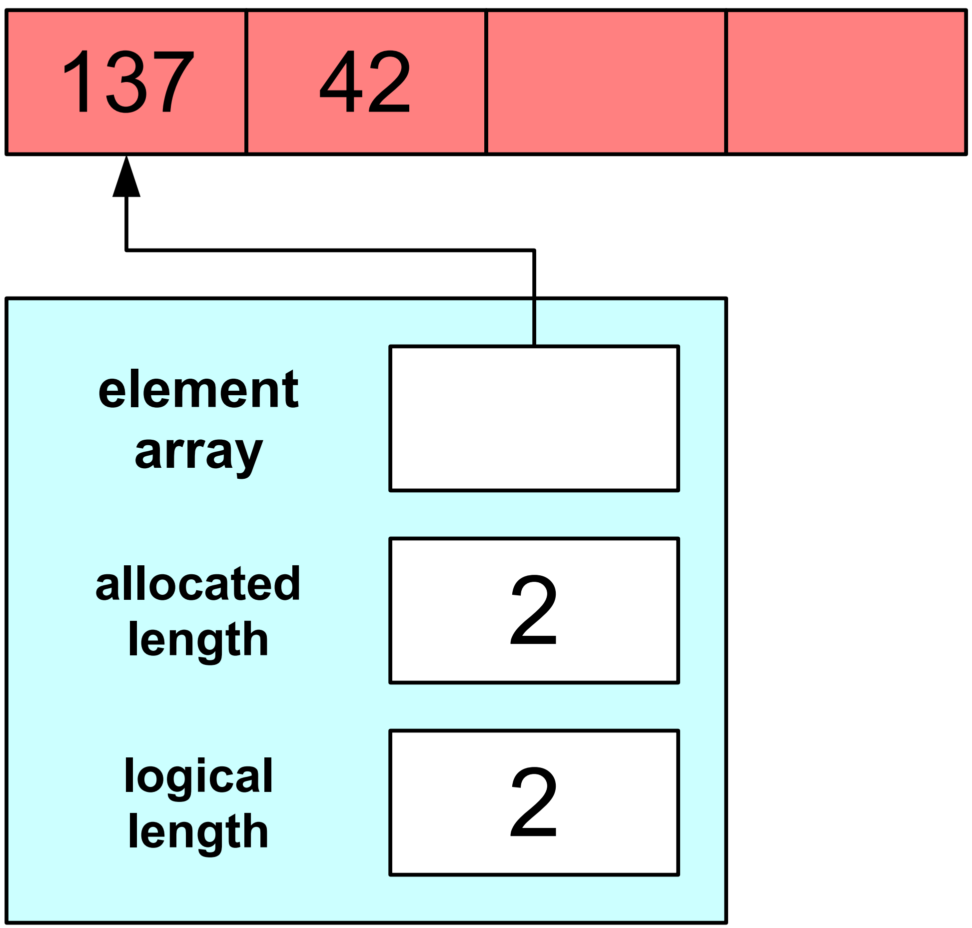
A Much Better Idea



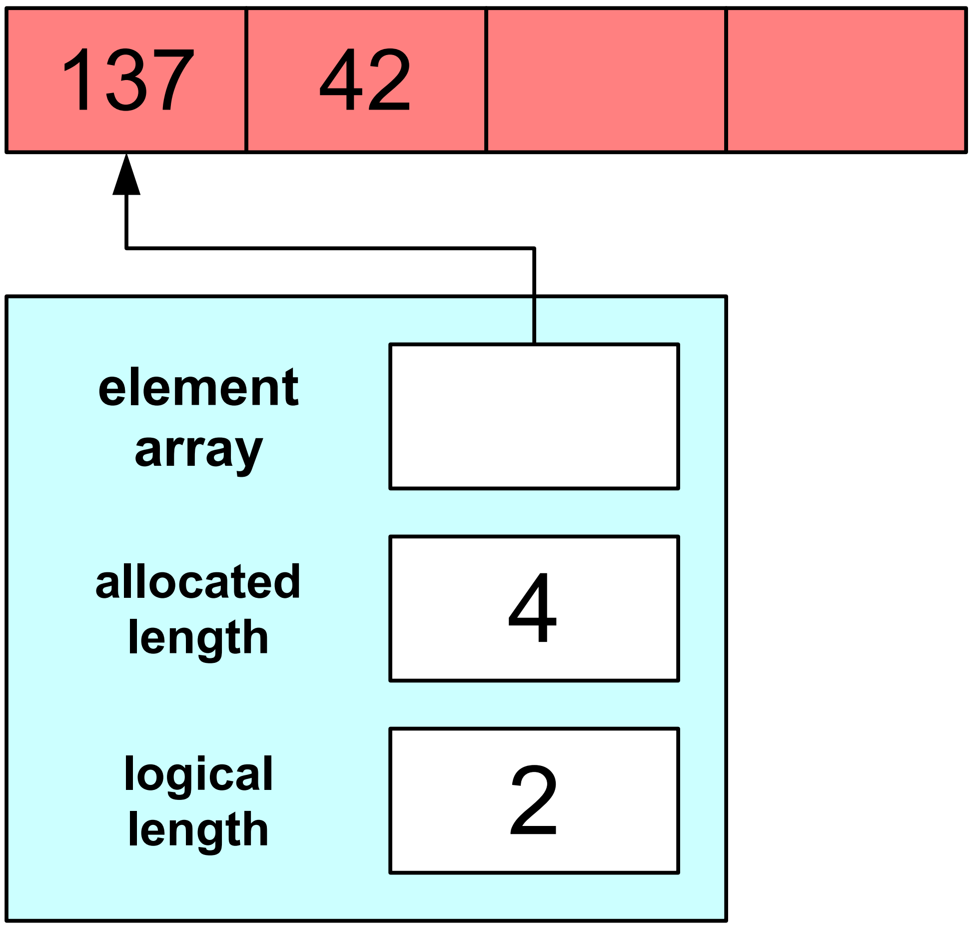
A Much Better Idea



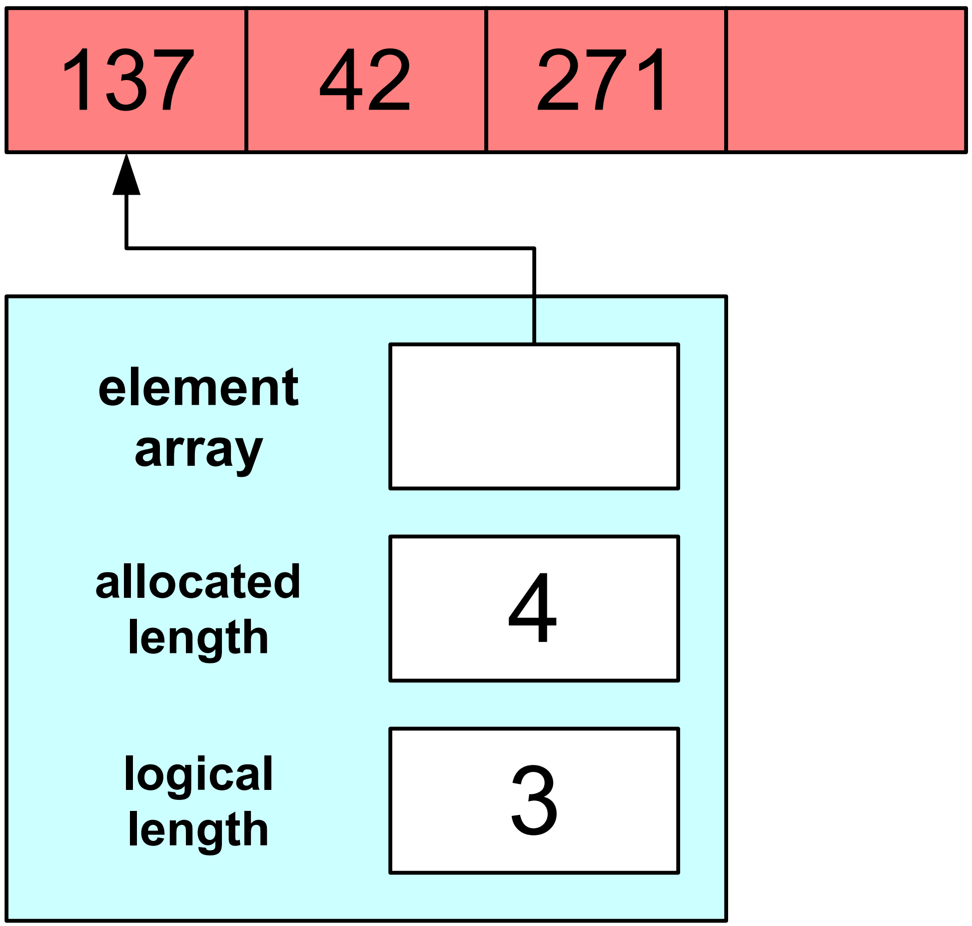
A Much Better Idea



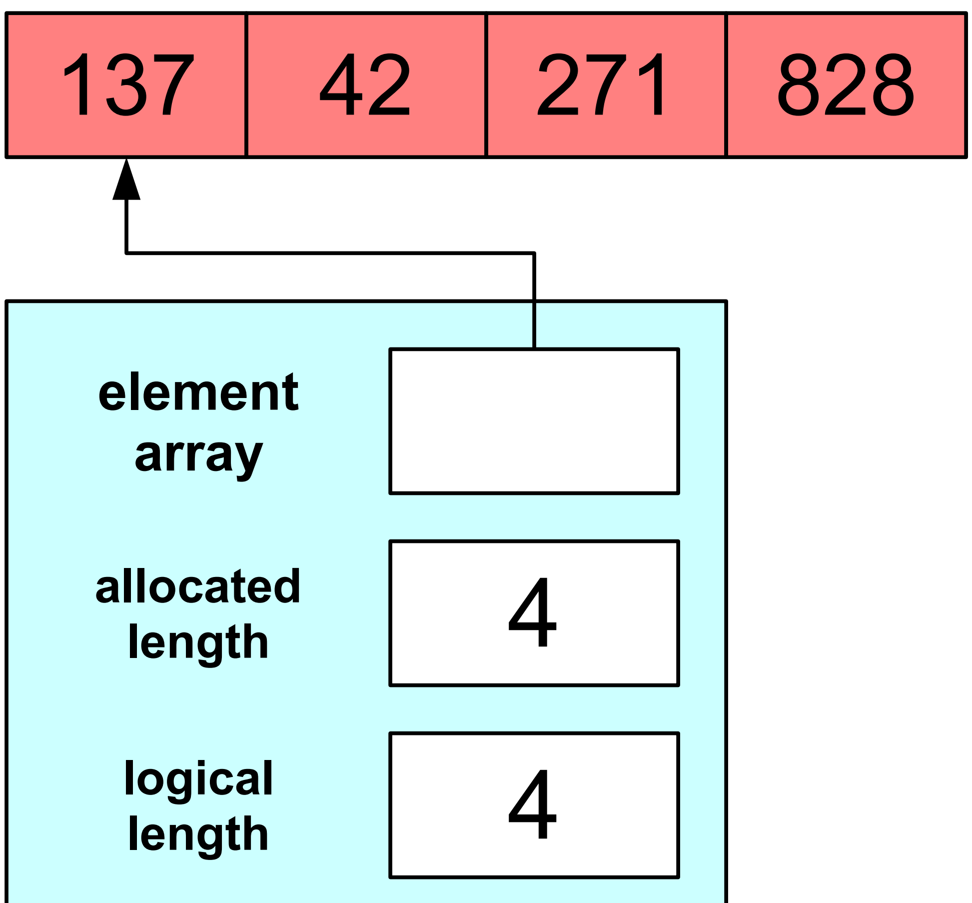
A Much Better Idea



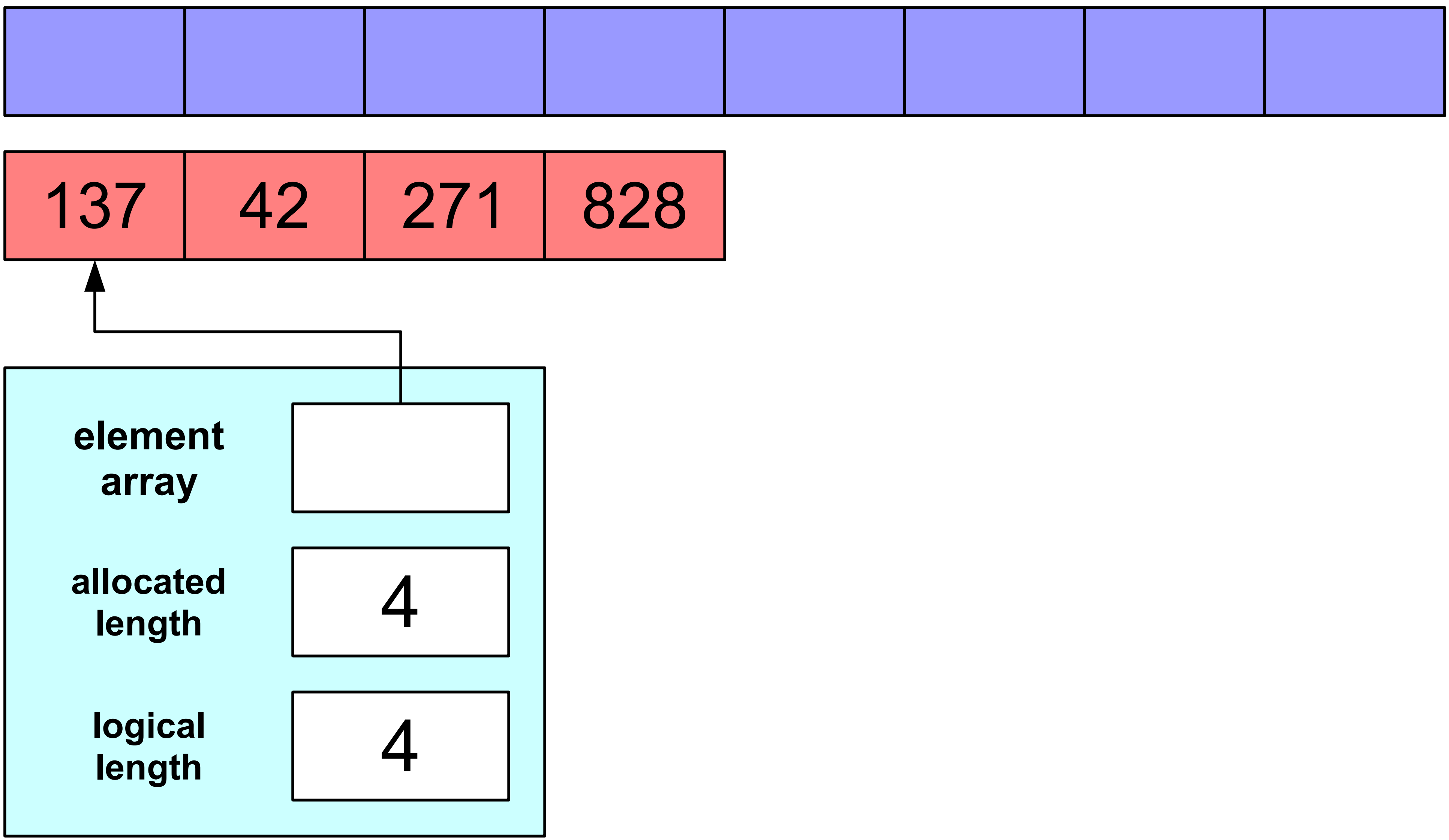
A Much Better Idea



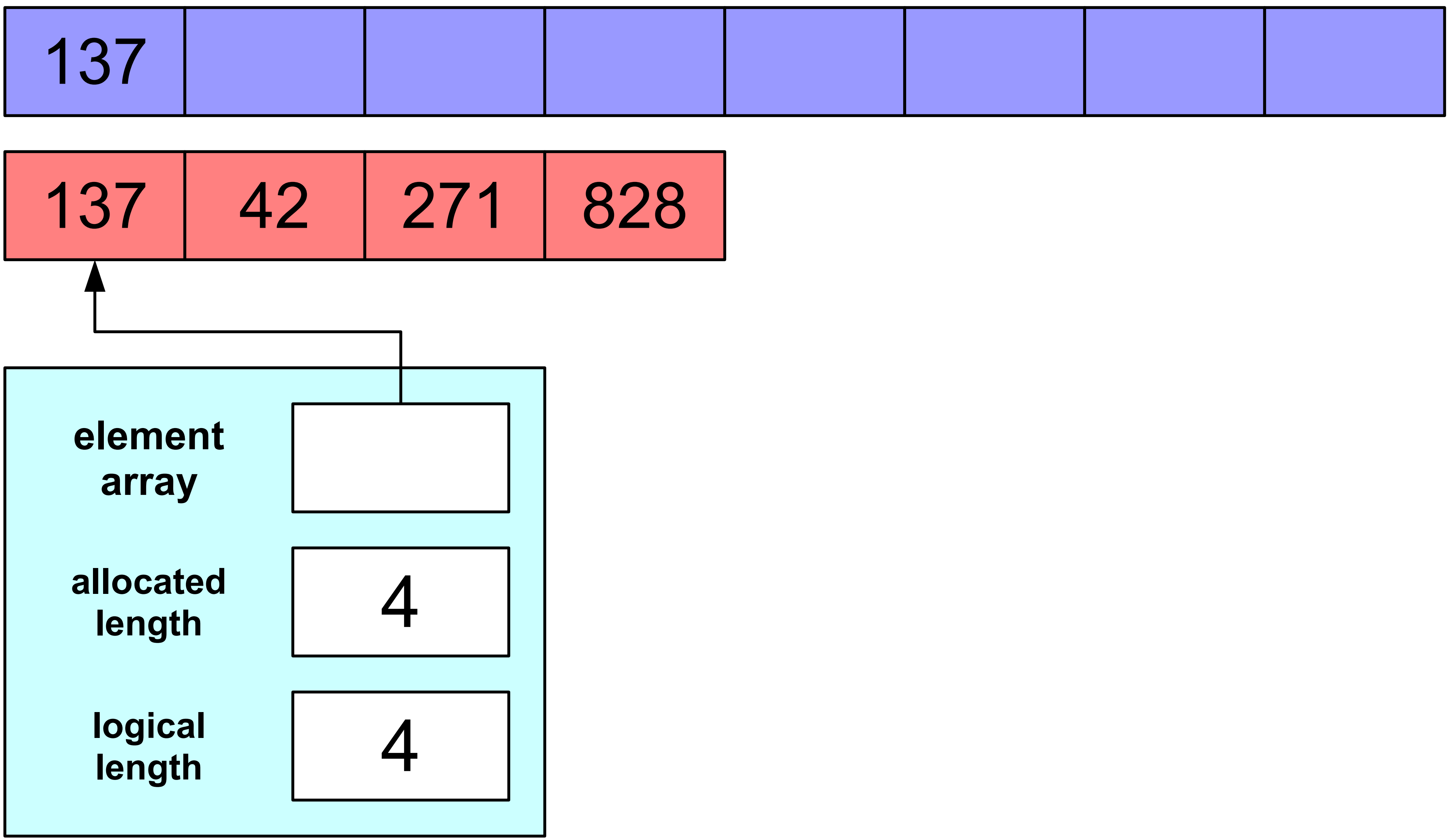
A Much Better Idea



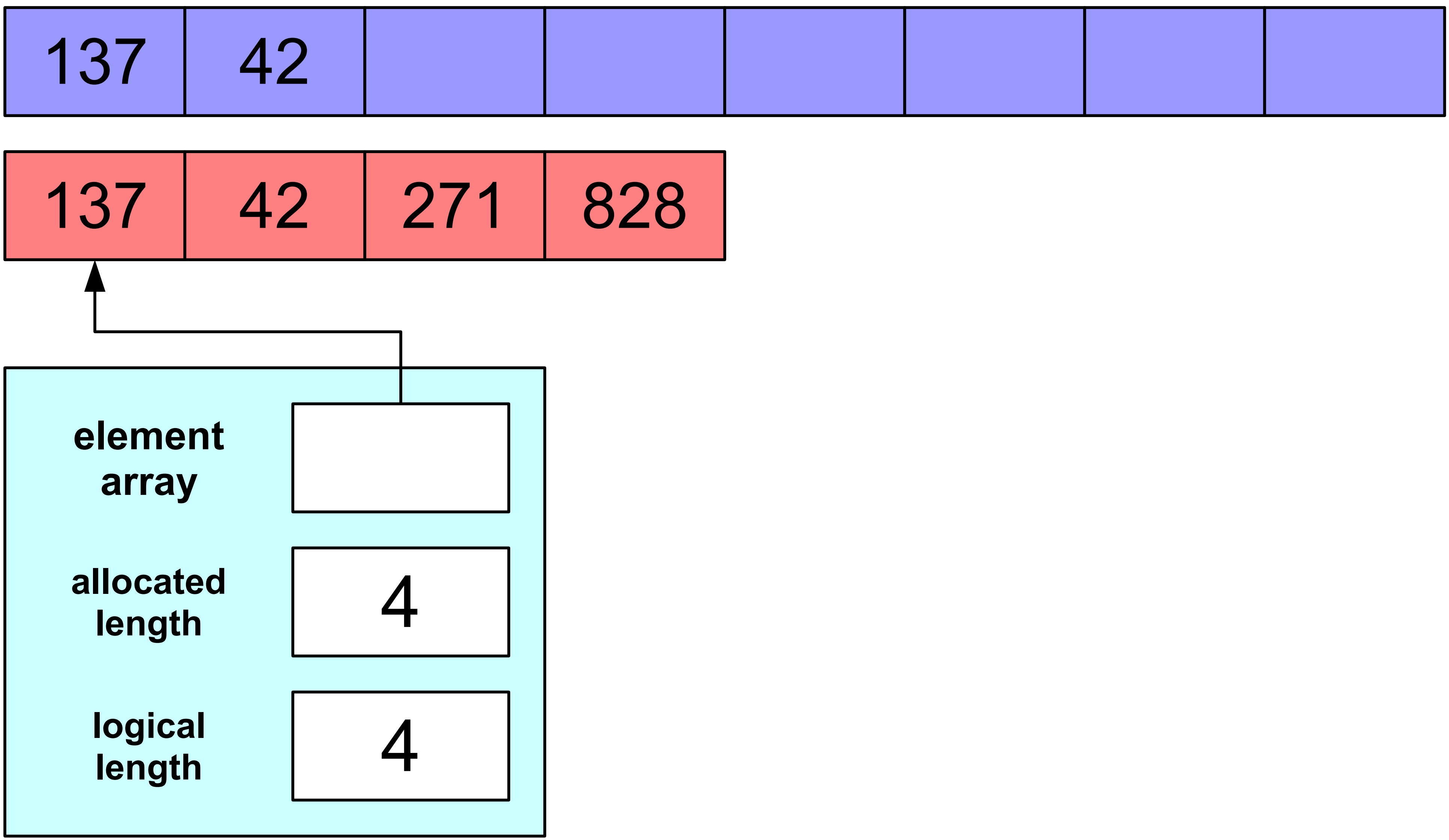
A Much Better Idea



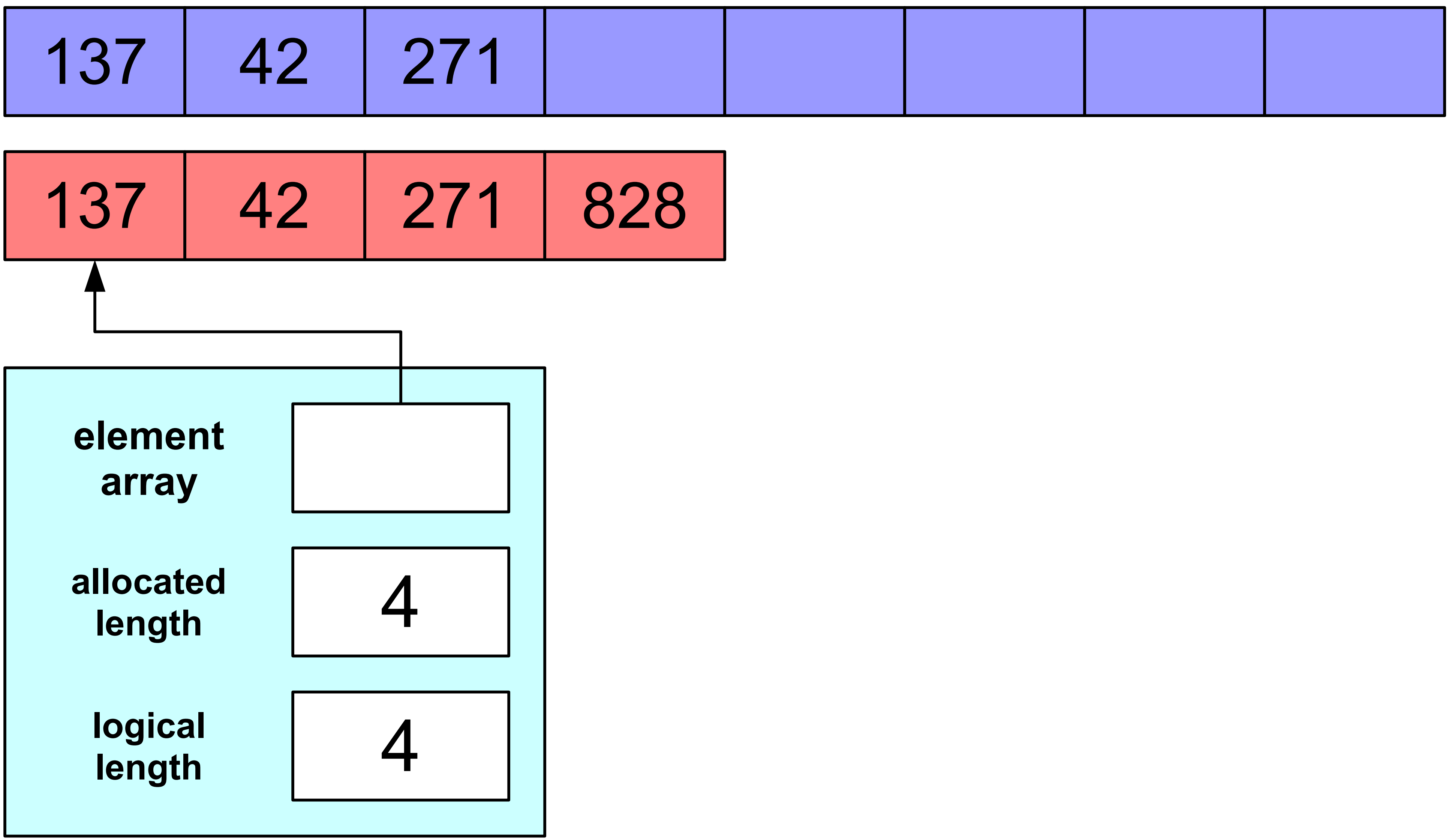
A Much Better Idea



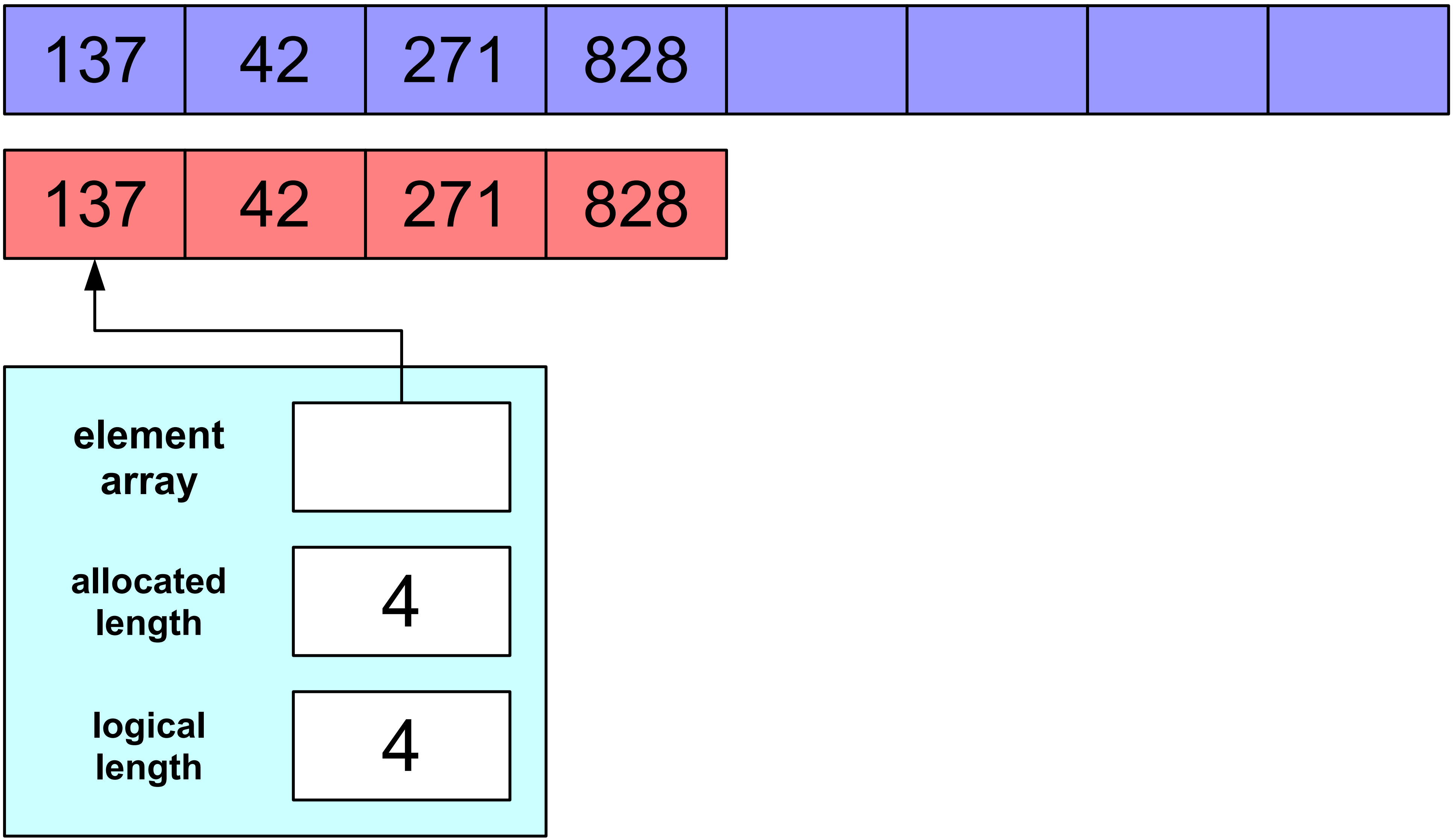
A Much Better Idea



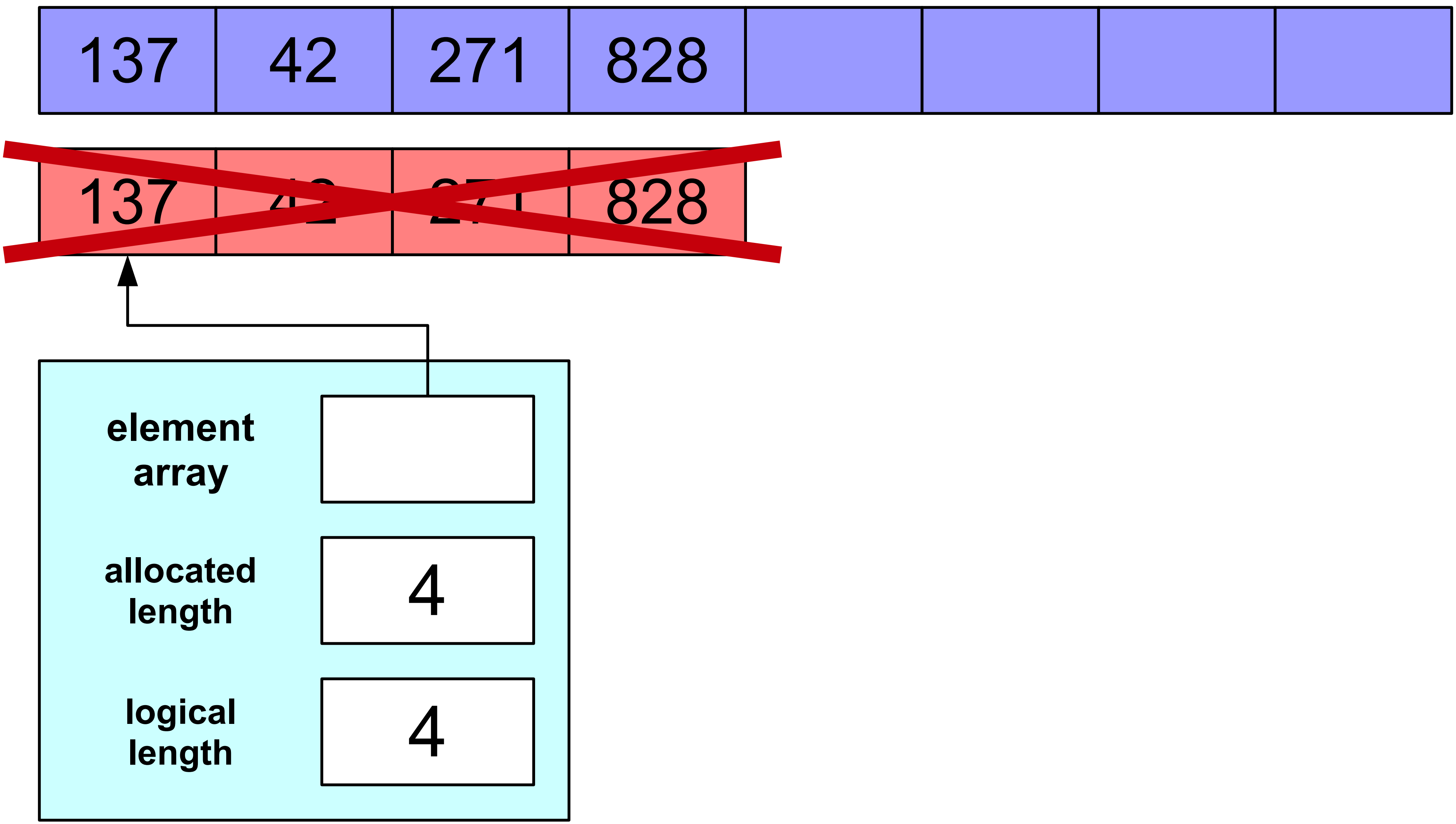
A Much Better Idea



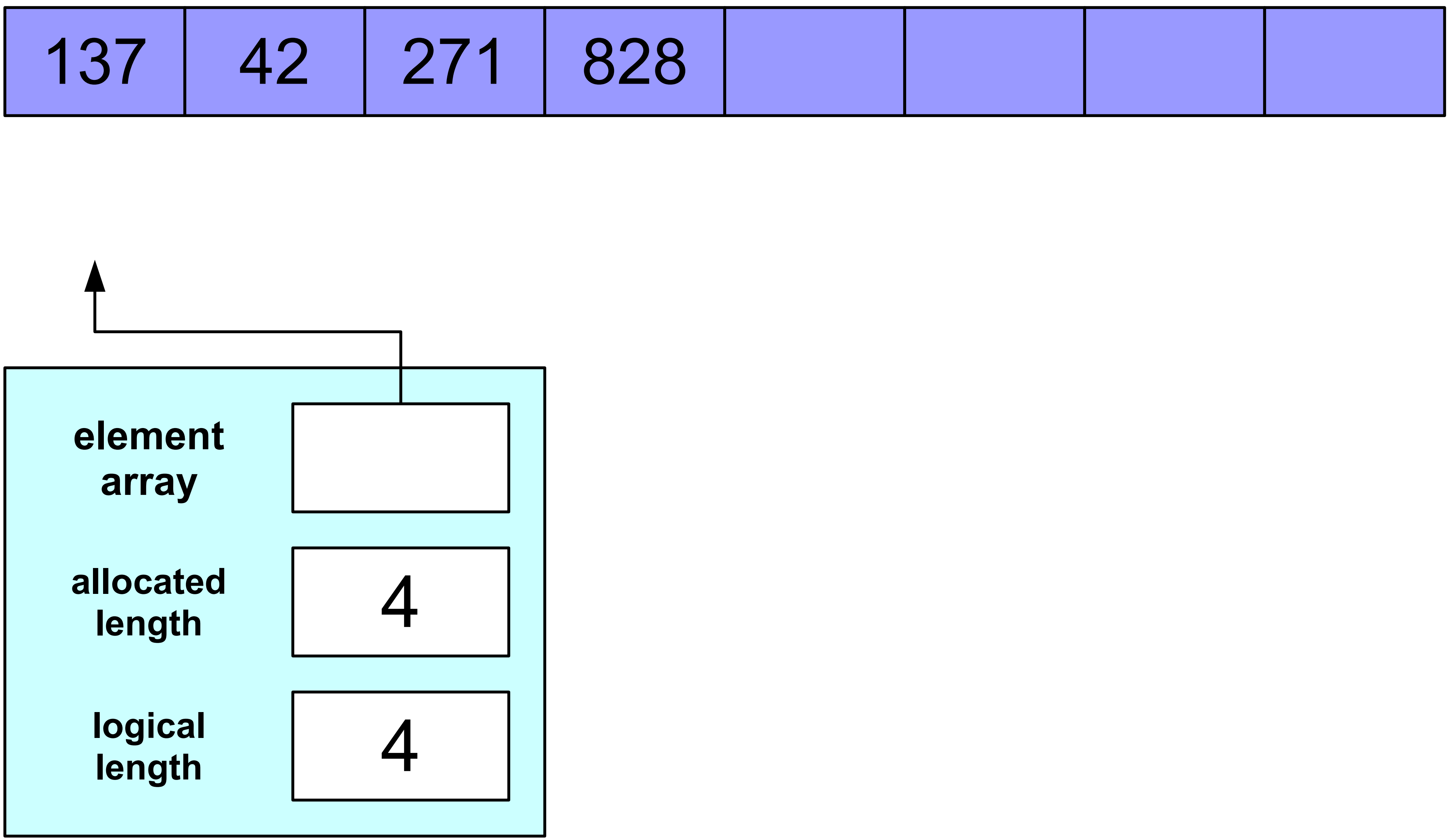
A Much Better Idea



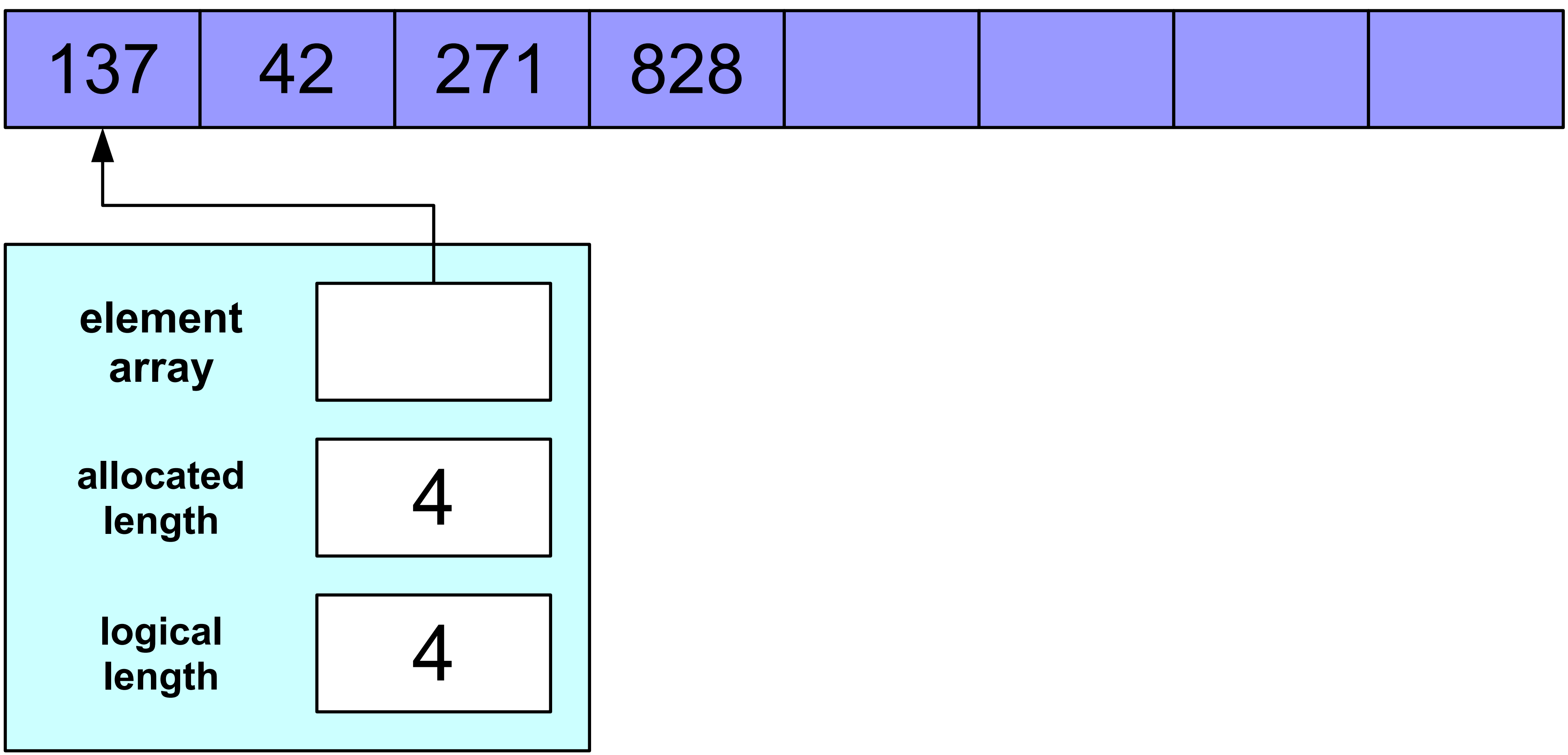
A Much Better Idea



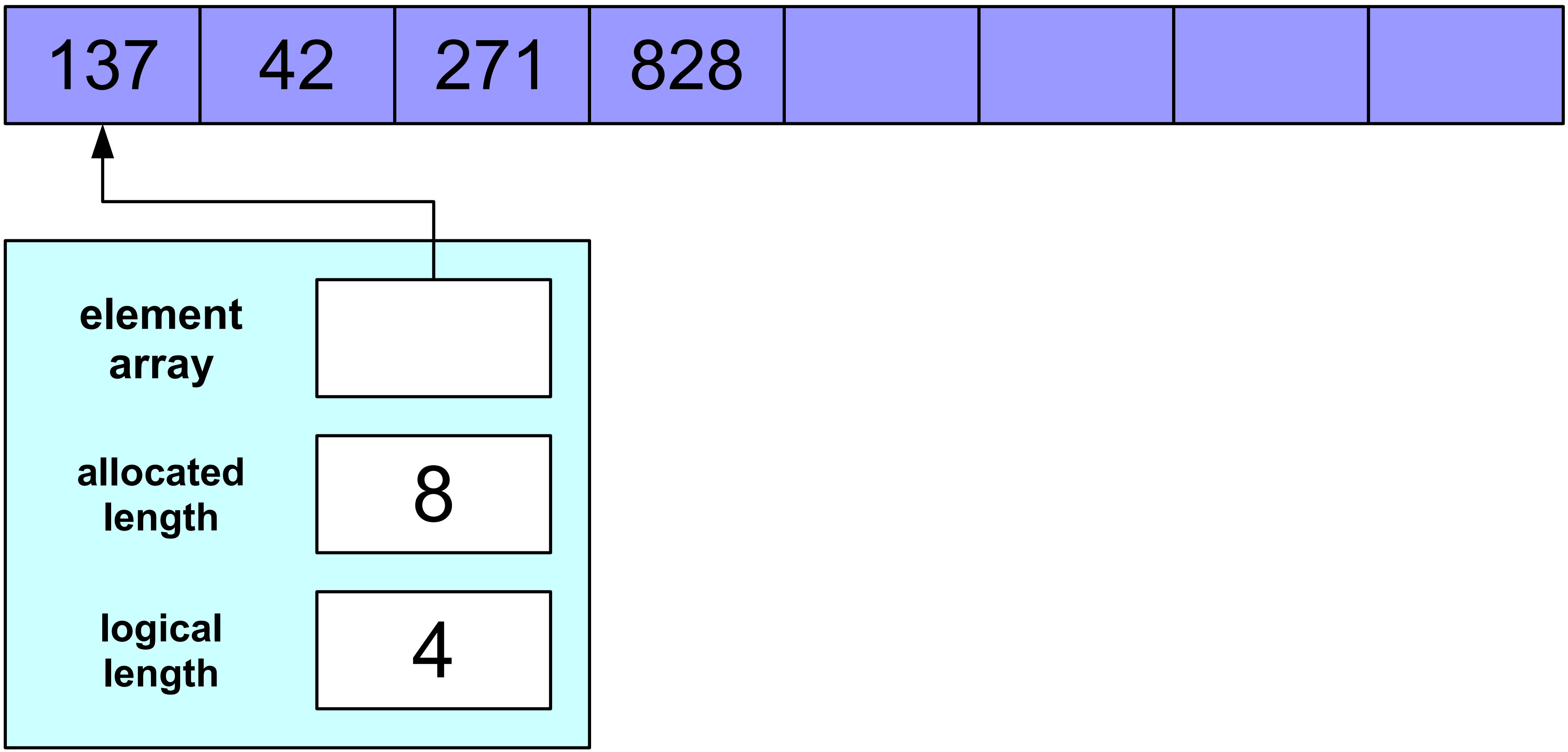
A Much Better Idea



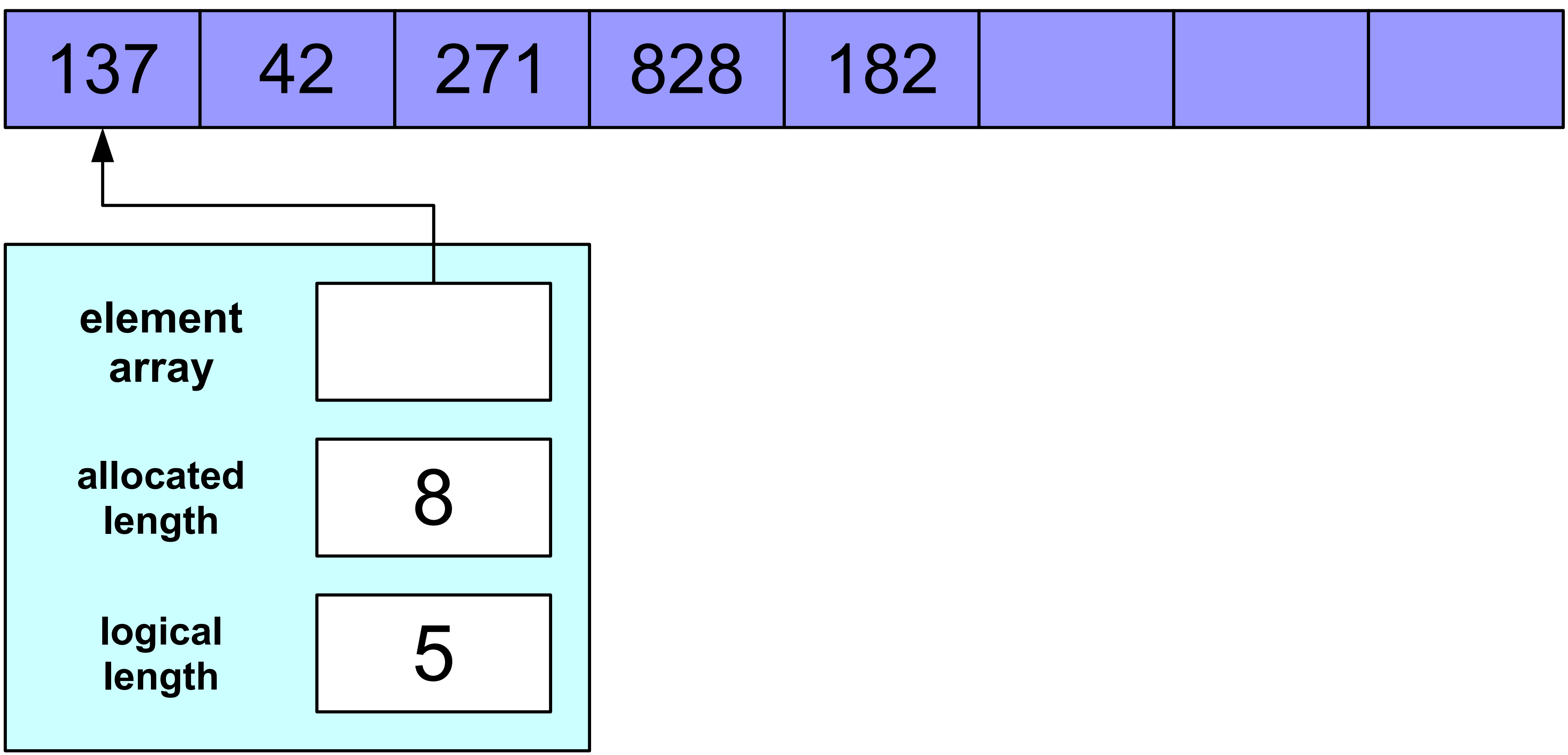
A Much Better Idea



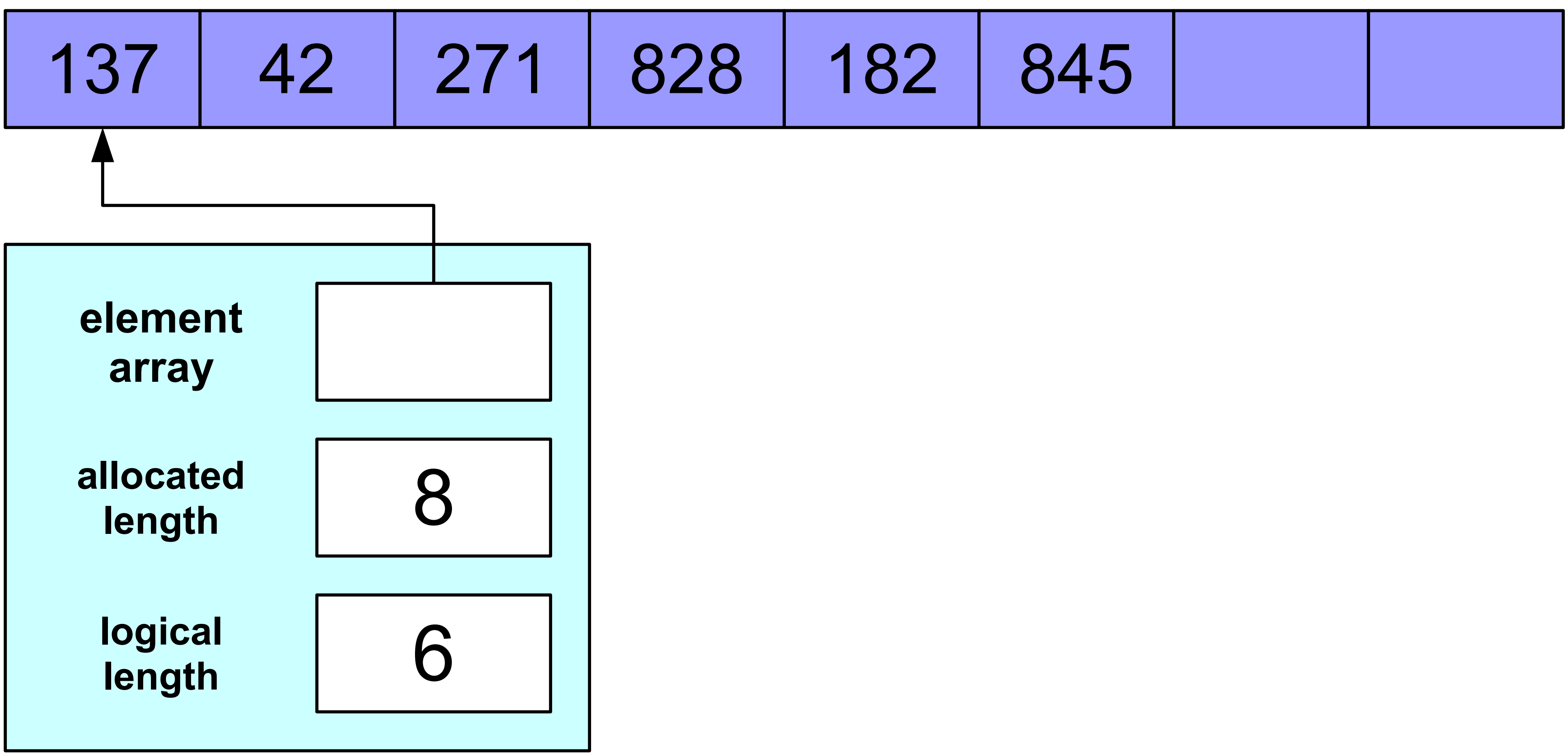
A Much Better Idea



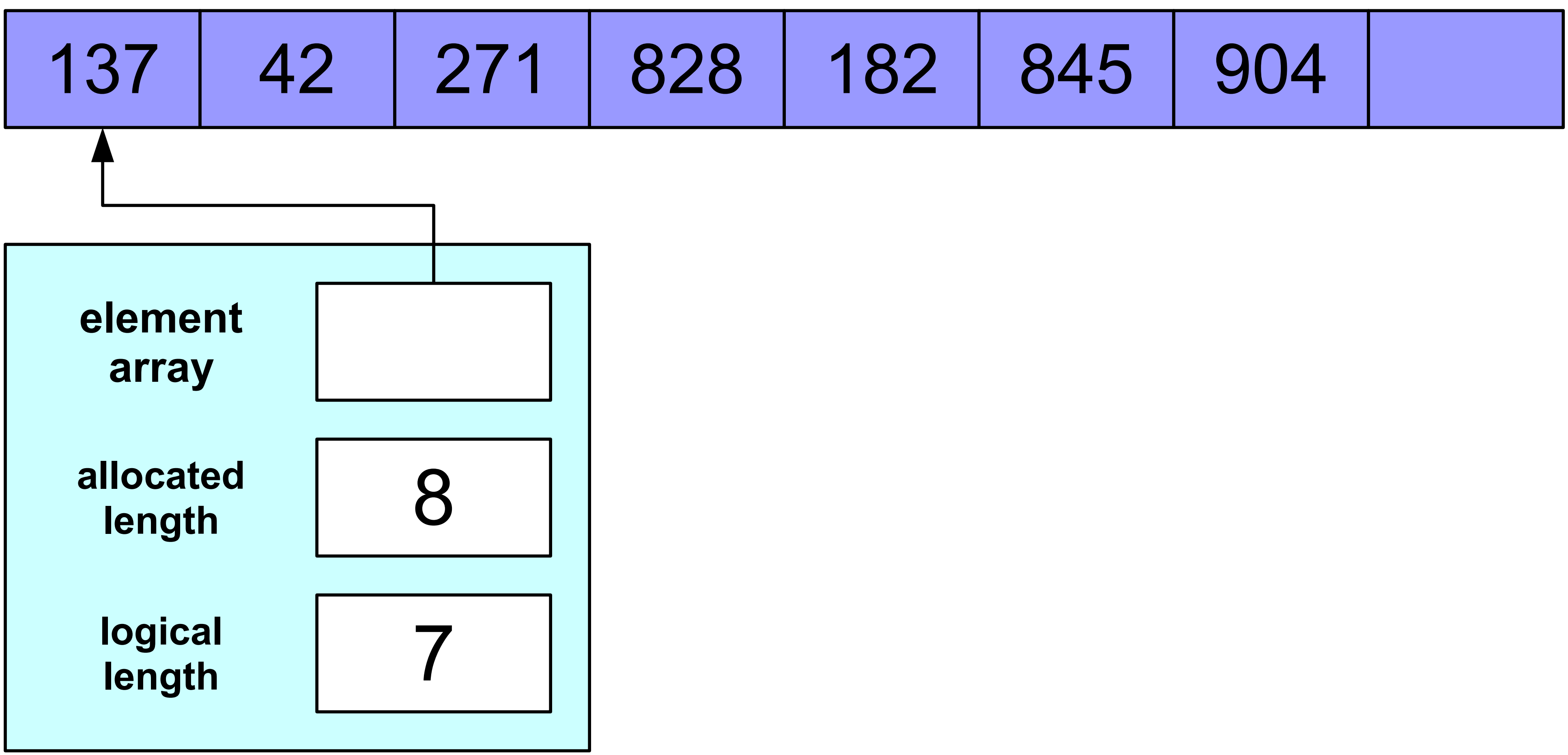
A Much Better Idea



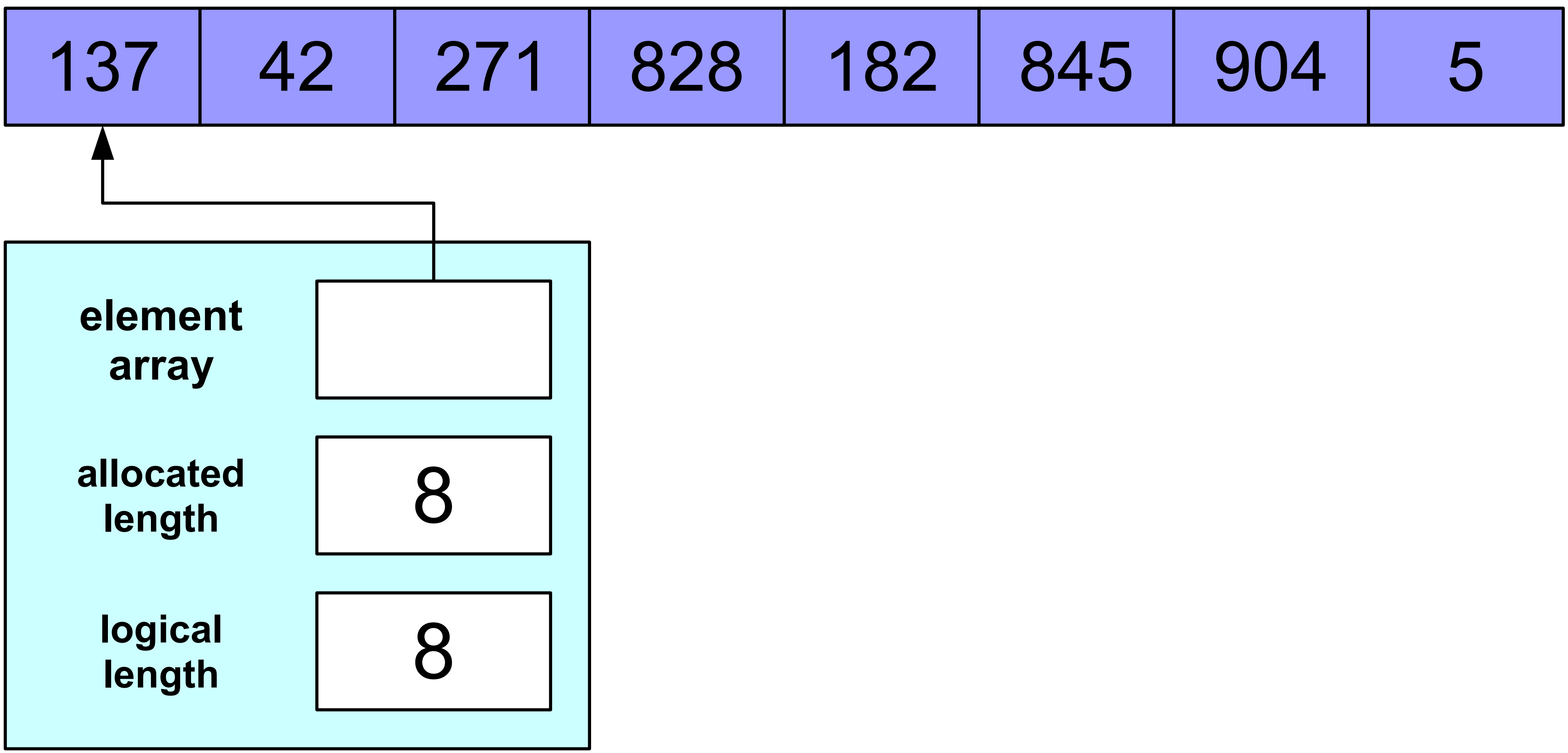
A Much Better Idea



A Much Better Idea



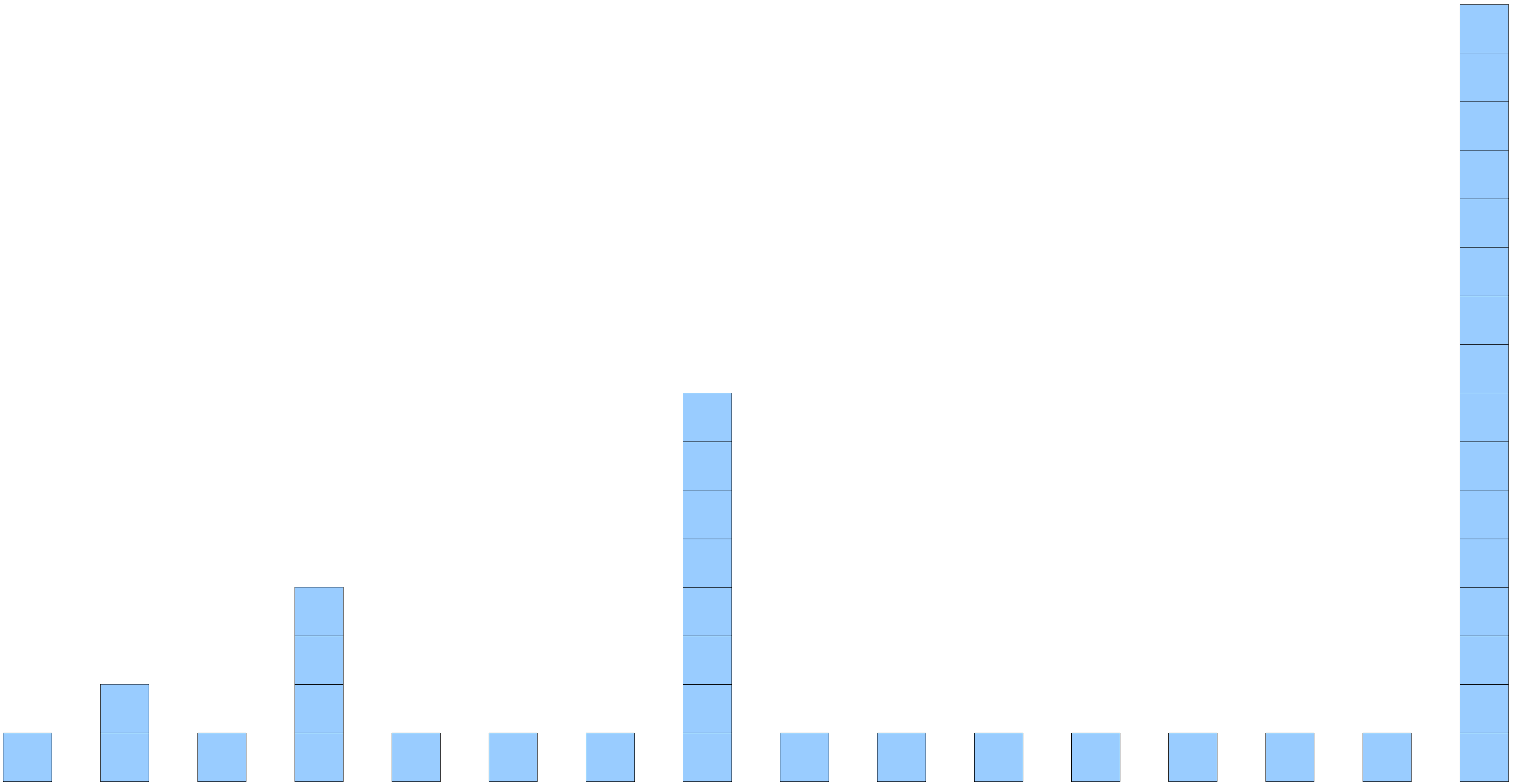
A Much Better Idea



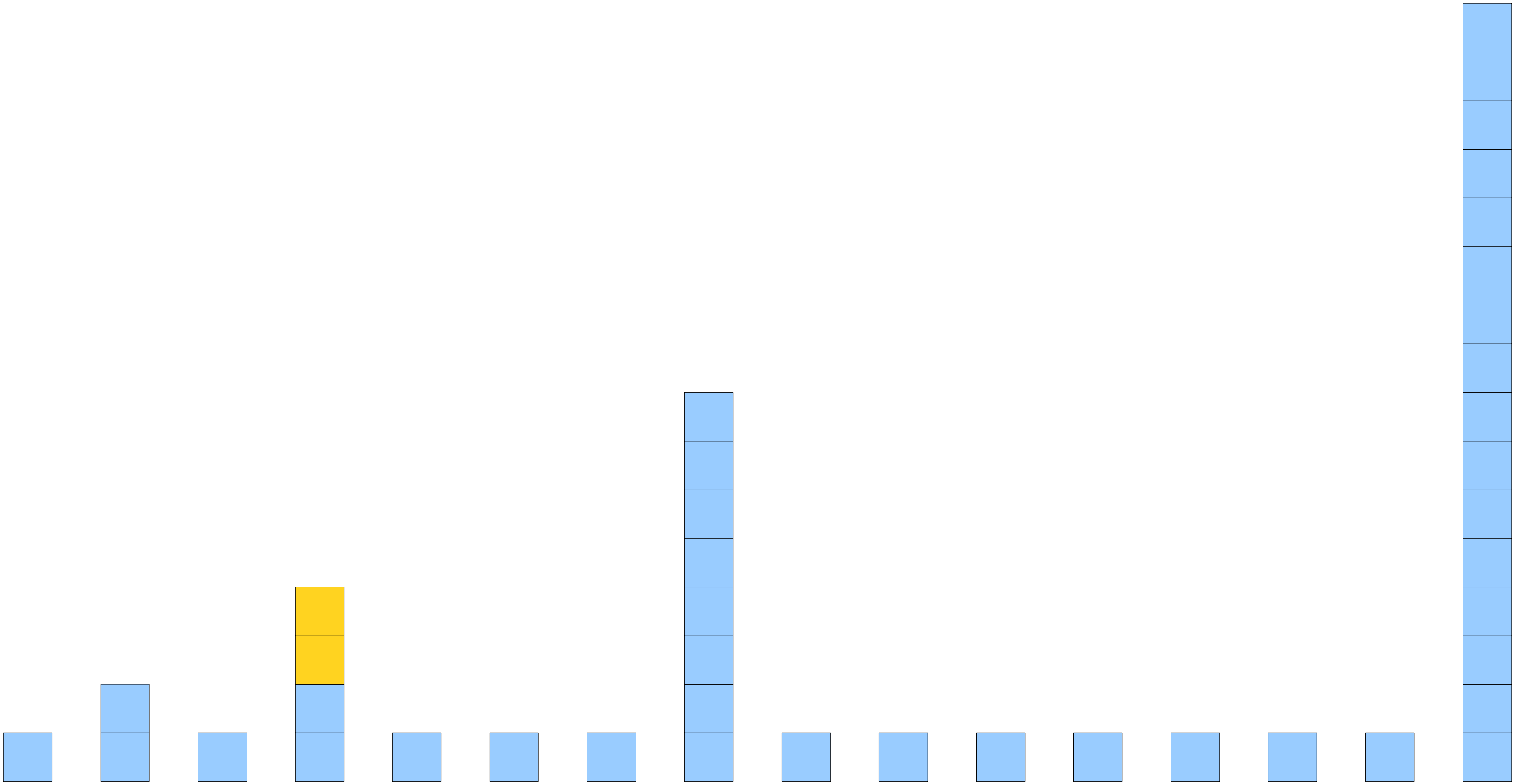
Let's Give it a Try!

How do we analyze this?

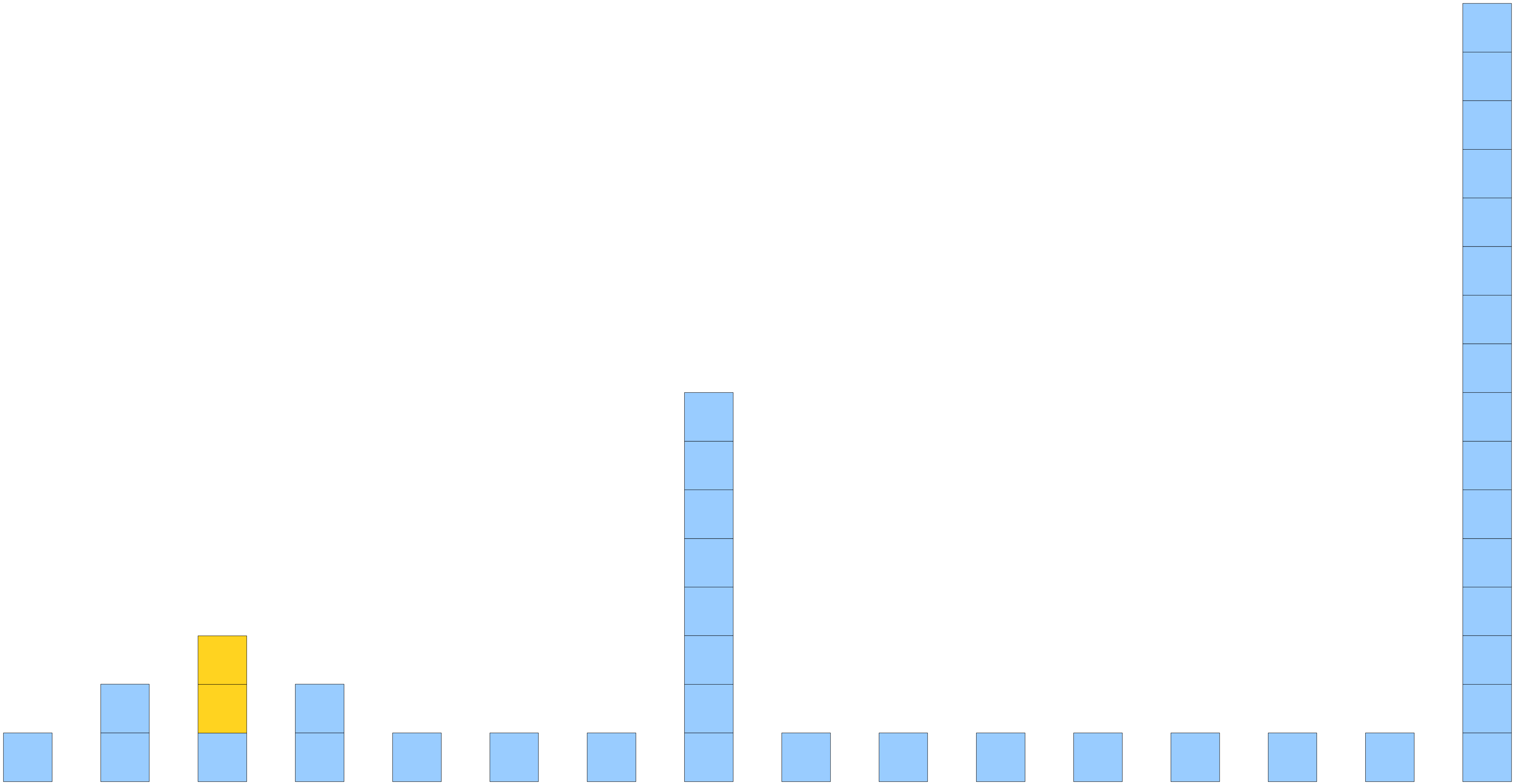
Spreading the Work



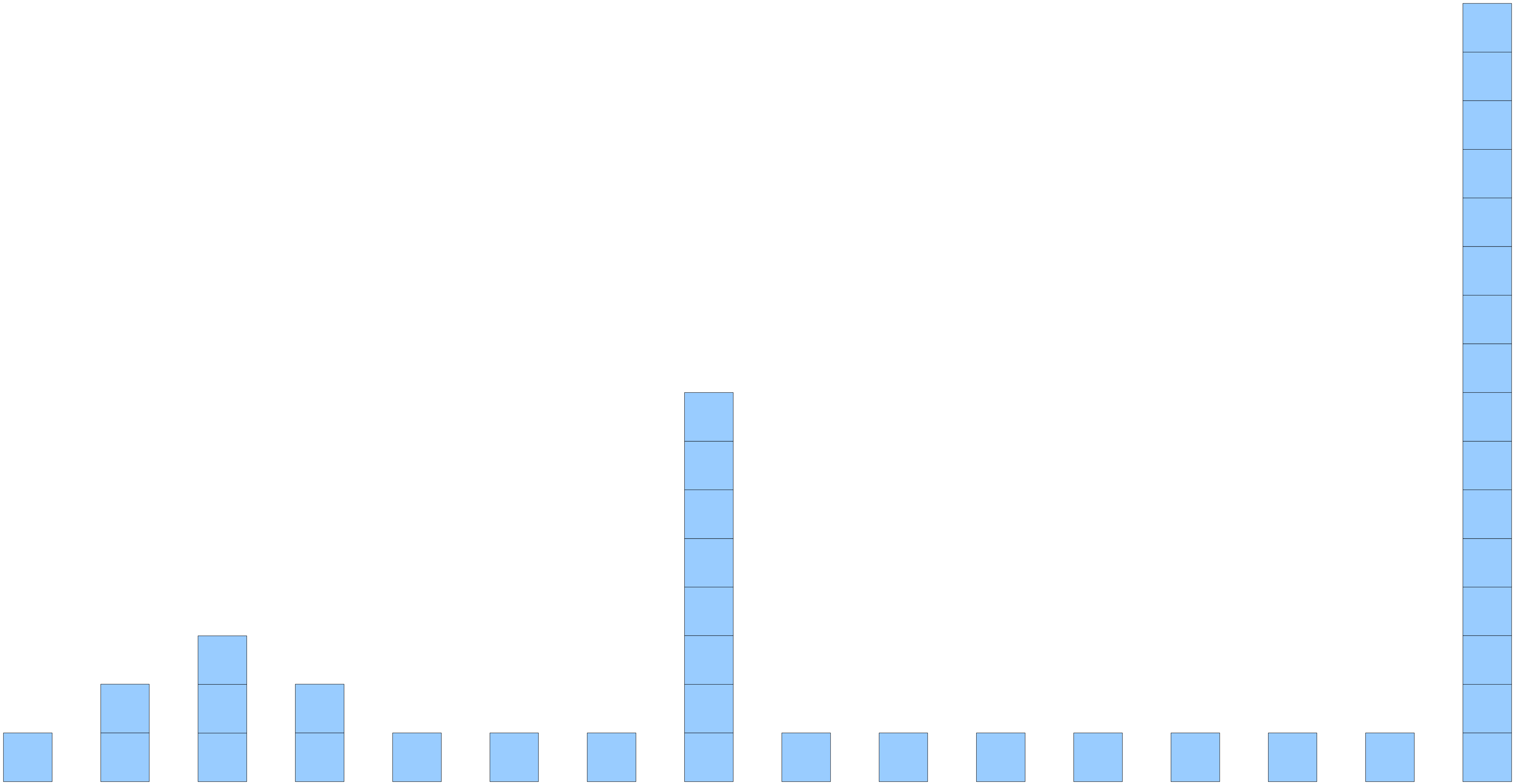
Spreading the Work



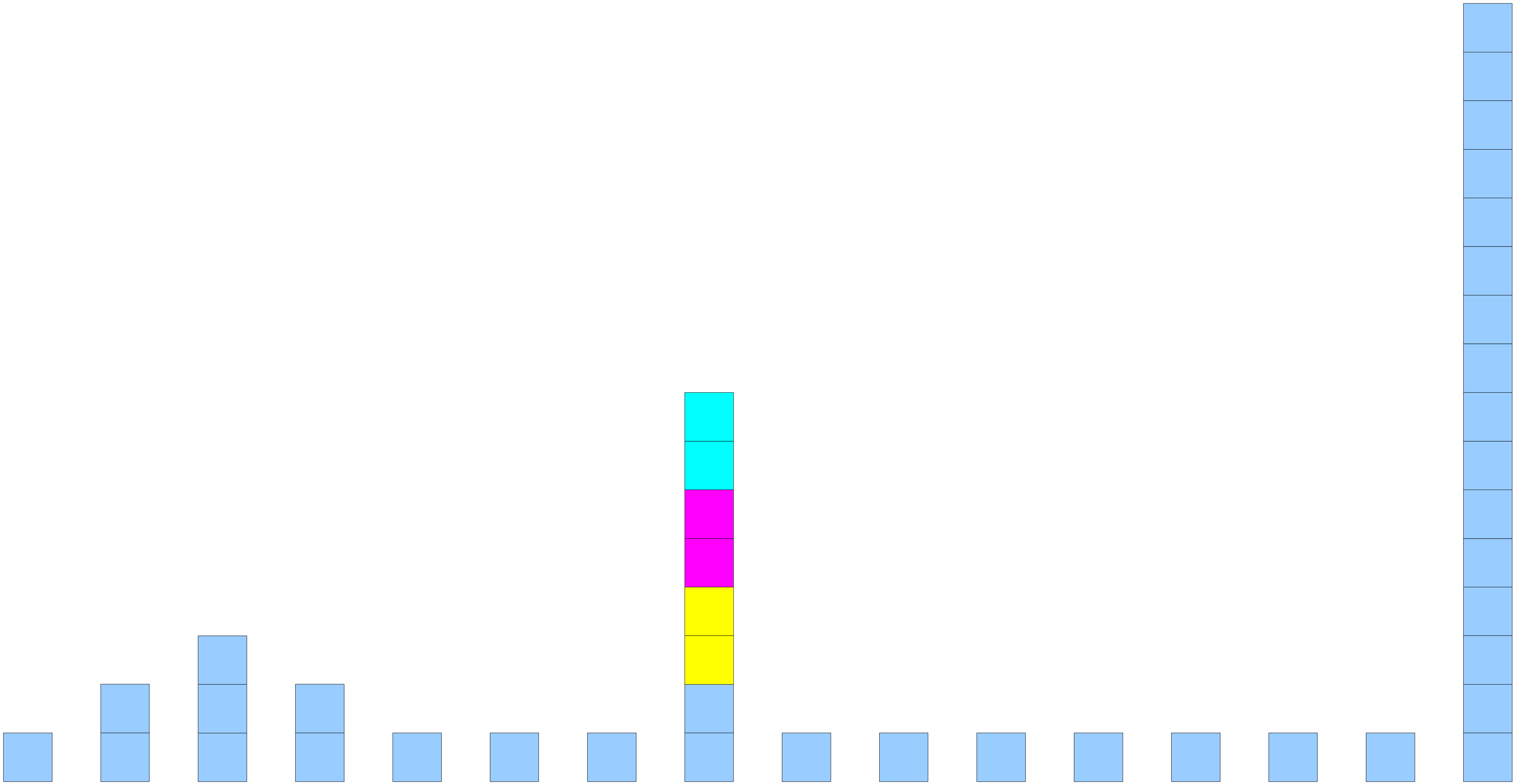
Spreading the Work



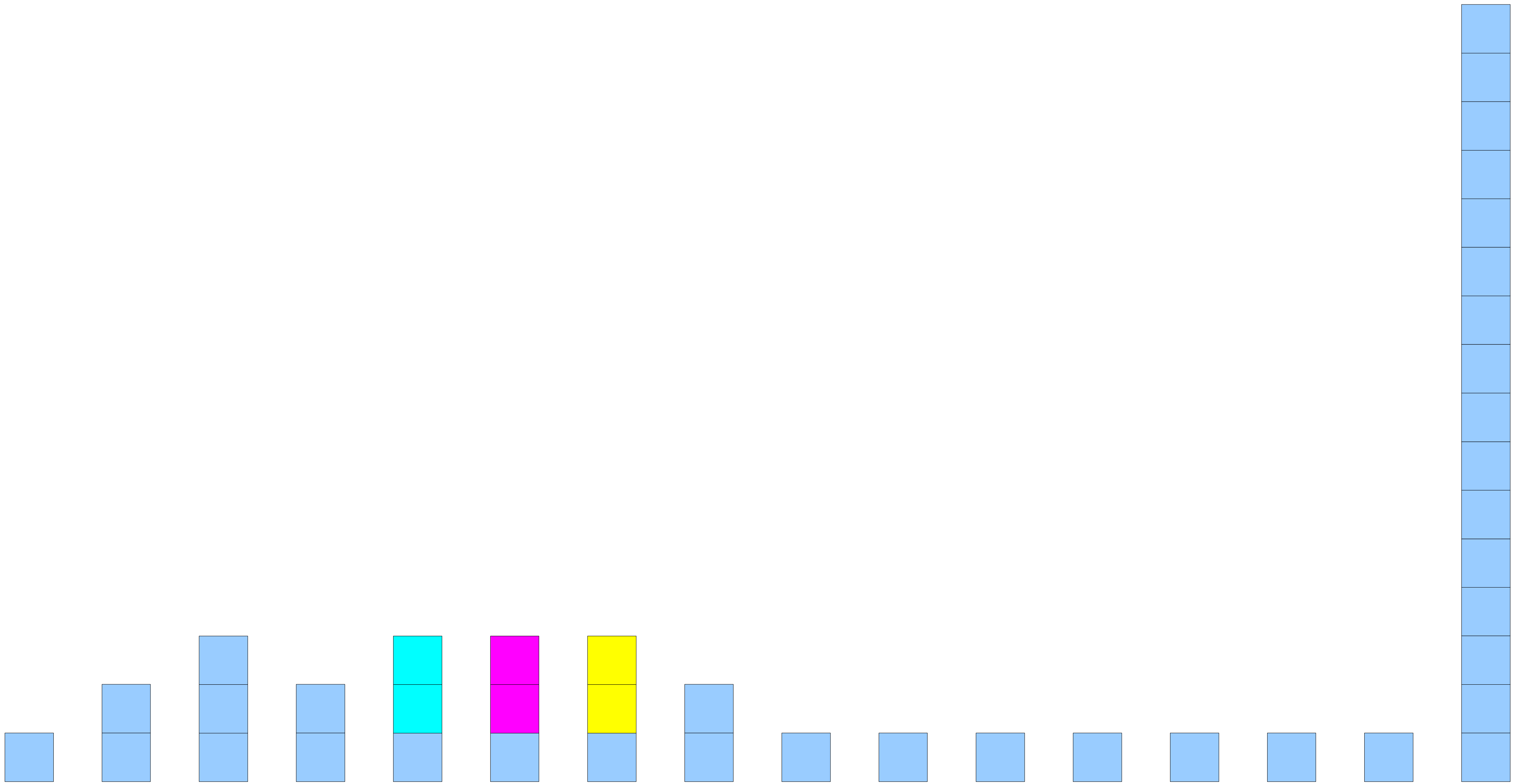
Spreading the Work



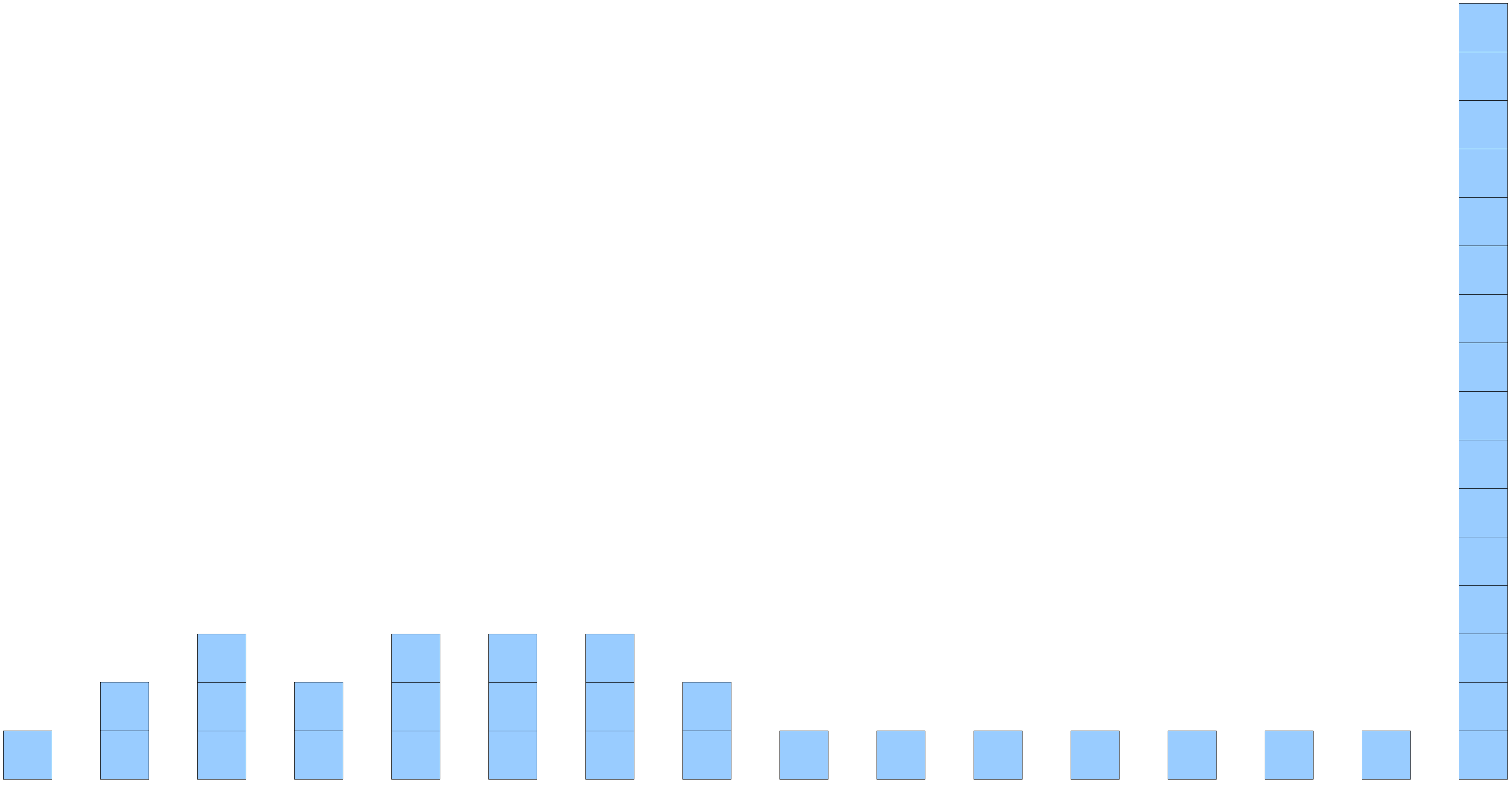
Spreading the Work



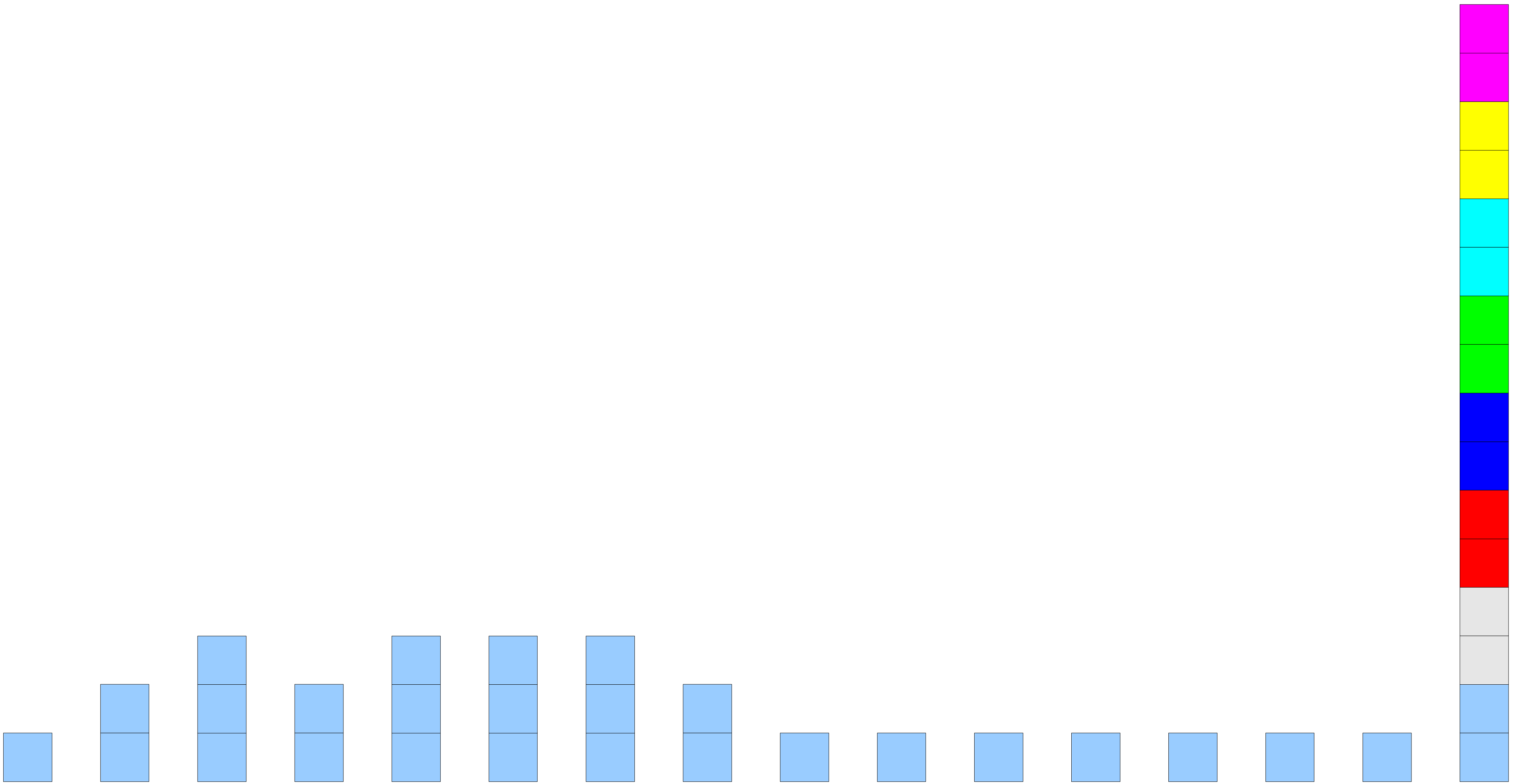
Spreading the Work



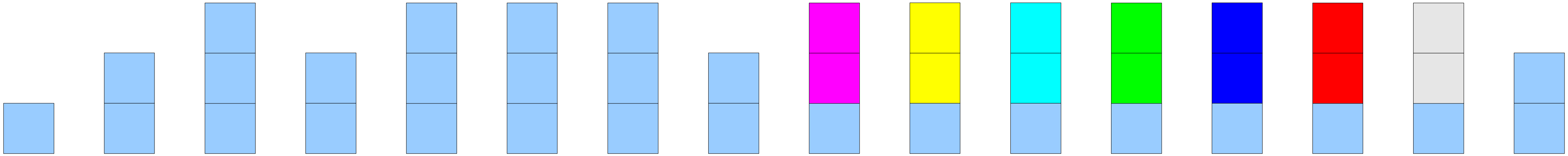
Spreading the Work



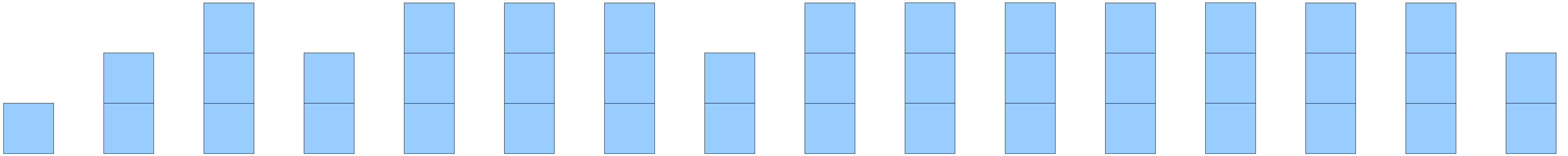
Spreading the Work



Spreading the Work



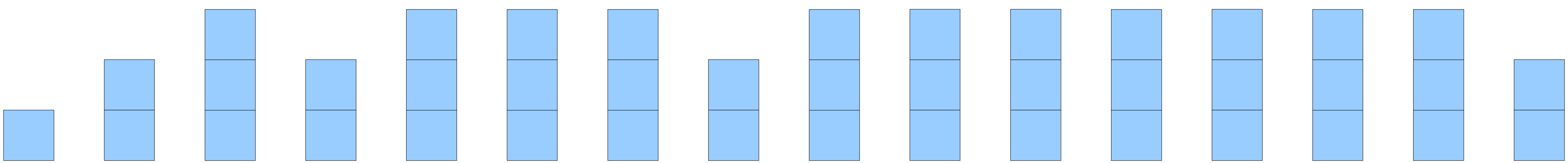
Spreading the Work



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`**.

Amortized Analysis

- People are amortizing ***all the time!***
  - E.g. Buying Groceries
    - Driving to Store takes 30 minutes roundtrip.
    - Actually shopping takes 5 minutes per days worth of food you buy
    - If we only buy 1 days worth of food per visit then we'll spend 35 minutes a day shopping
    - If we buy 1 weeks worth of food per visit, then we'll spend only 65/7 ~ 9 minutes *on average* per day.

Implementing `Queue`

Implementing `Queue`

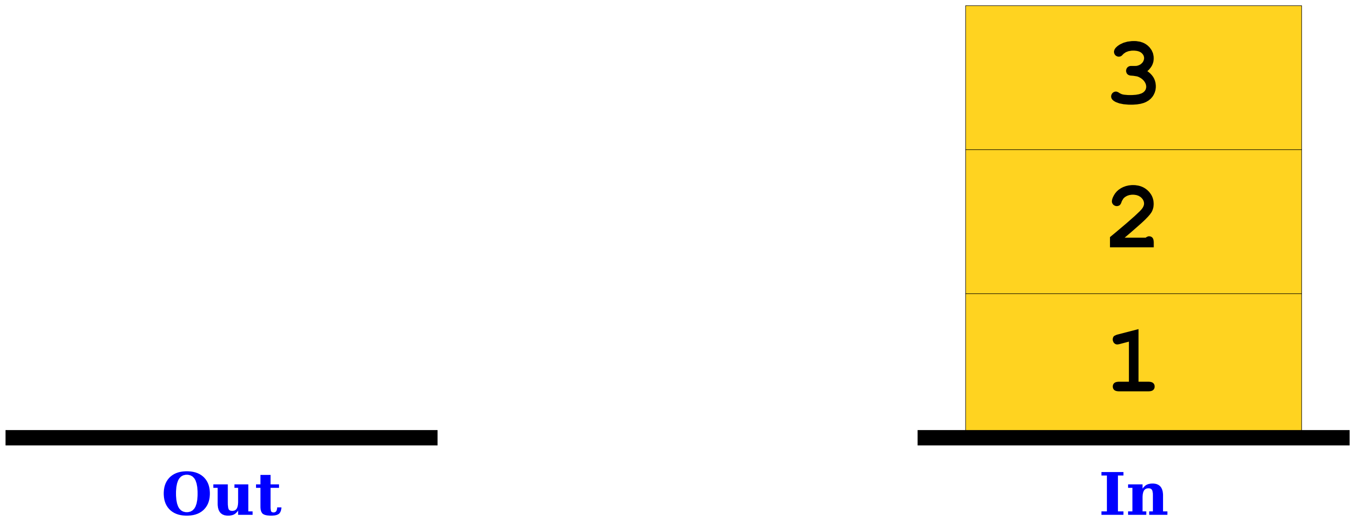
- We've just used dynamic arrays to implement a stack. Could we use them to implement a queue?
- Yes, but here's a better idea: **could we use our stack to implement a queue?**

The Two-Stack Queue

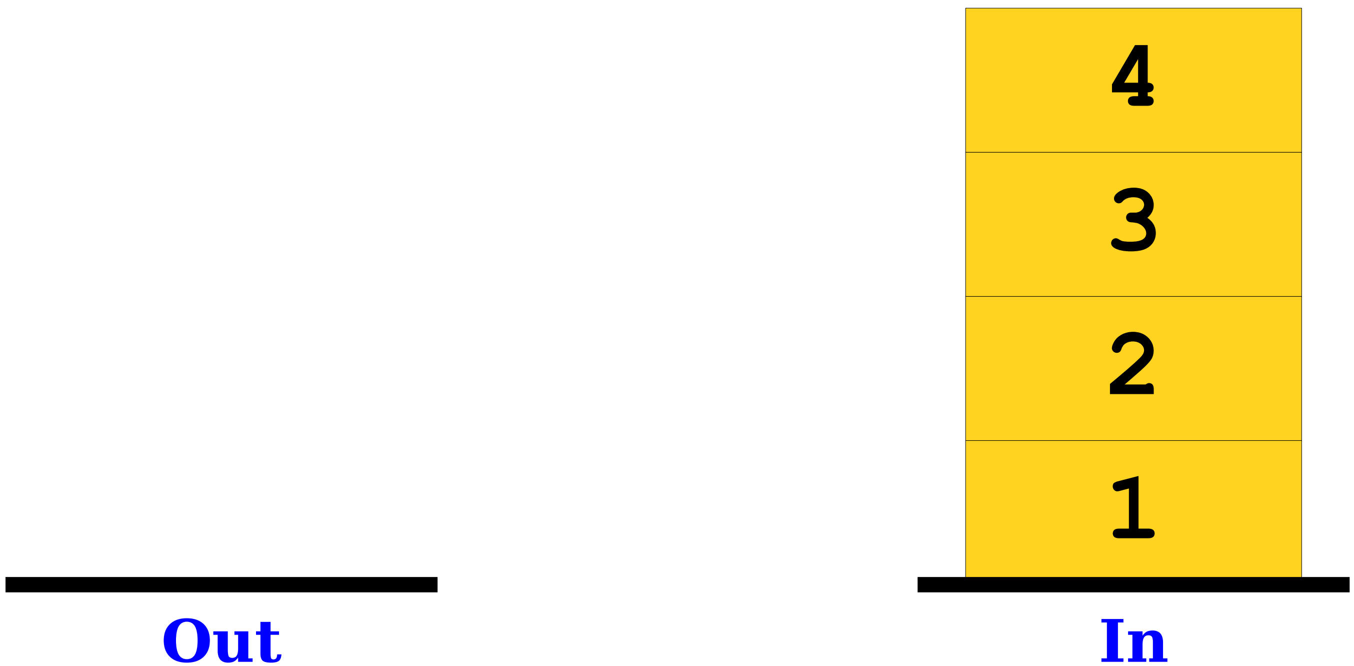
The Two-Stack Queue

The Two-Stack Queue

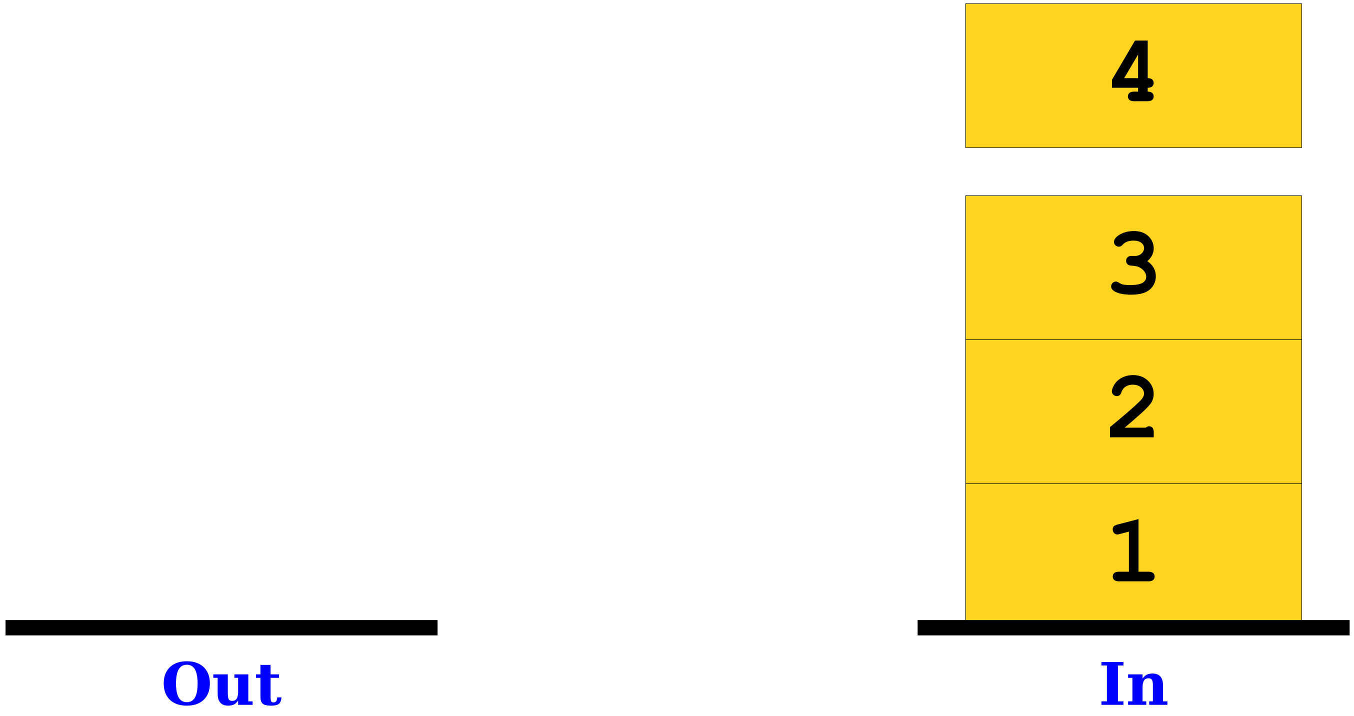
The Two-Stack Queue



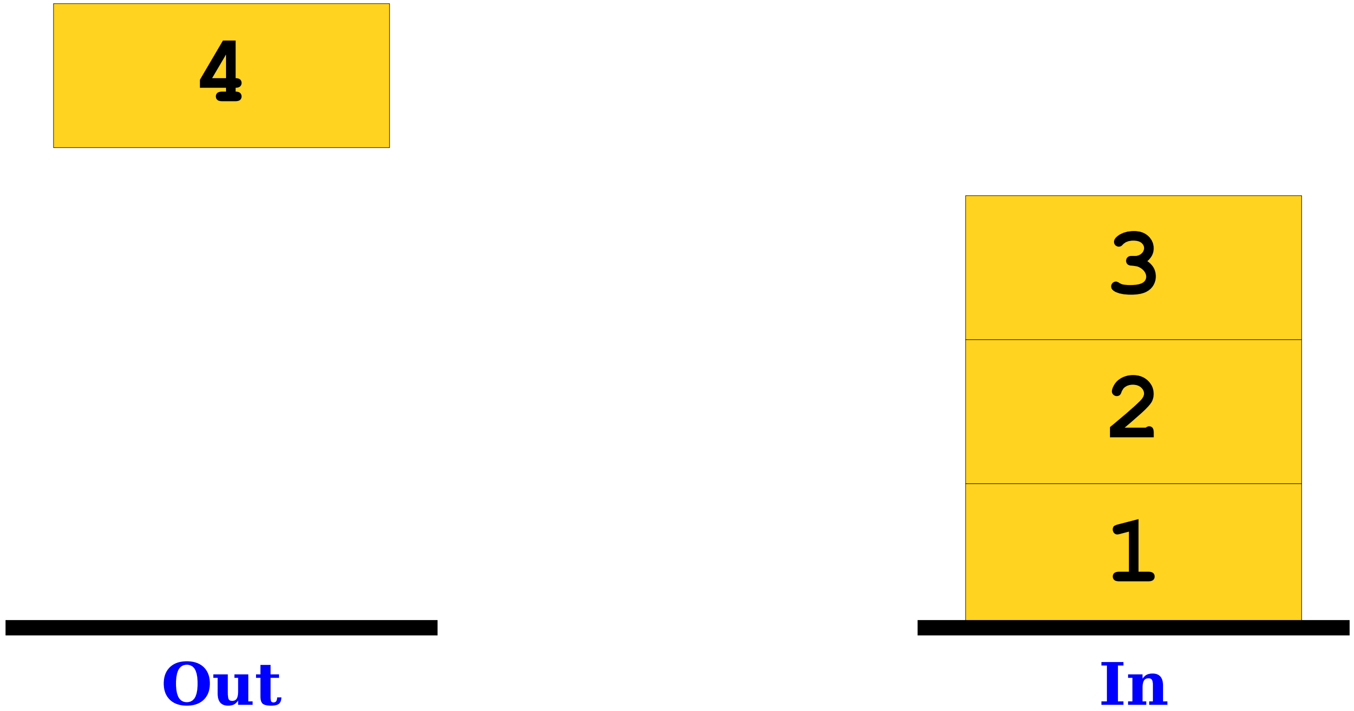
The Two-Stack Queue



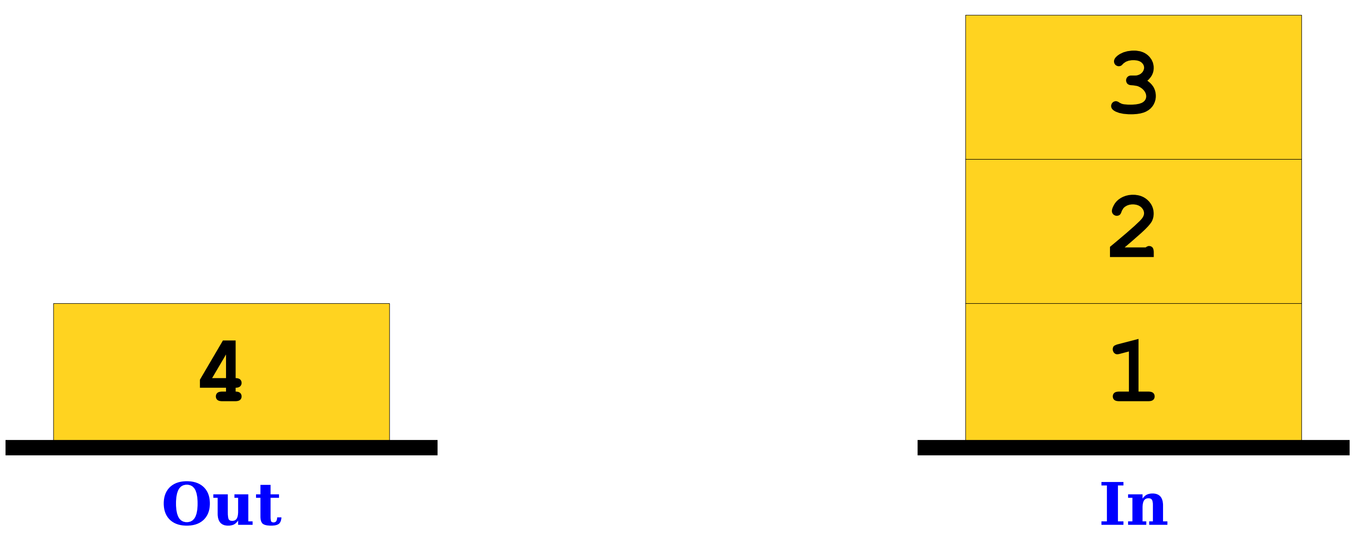
The Two-Stack Queue



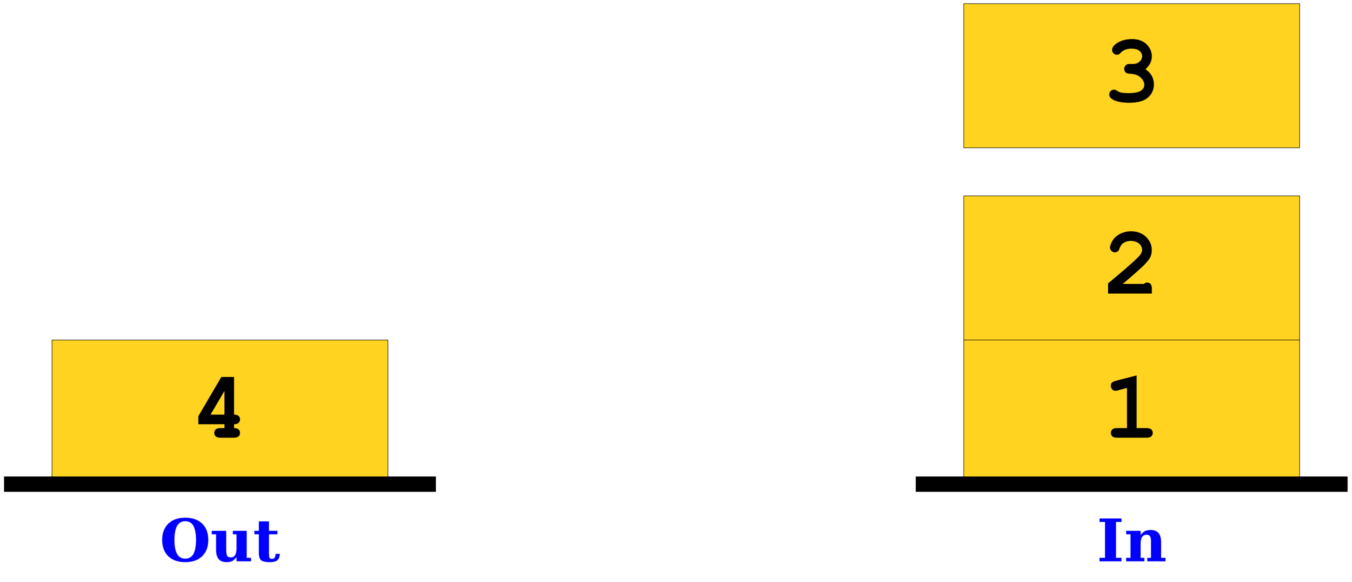
The Two-Stack Queue



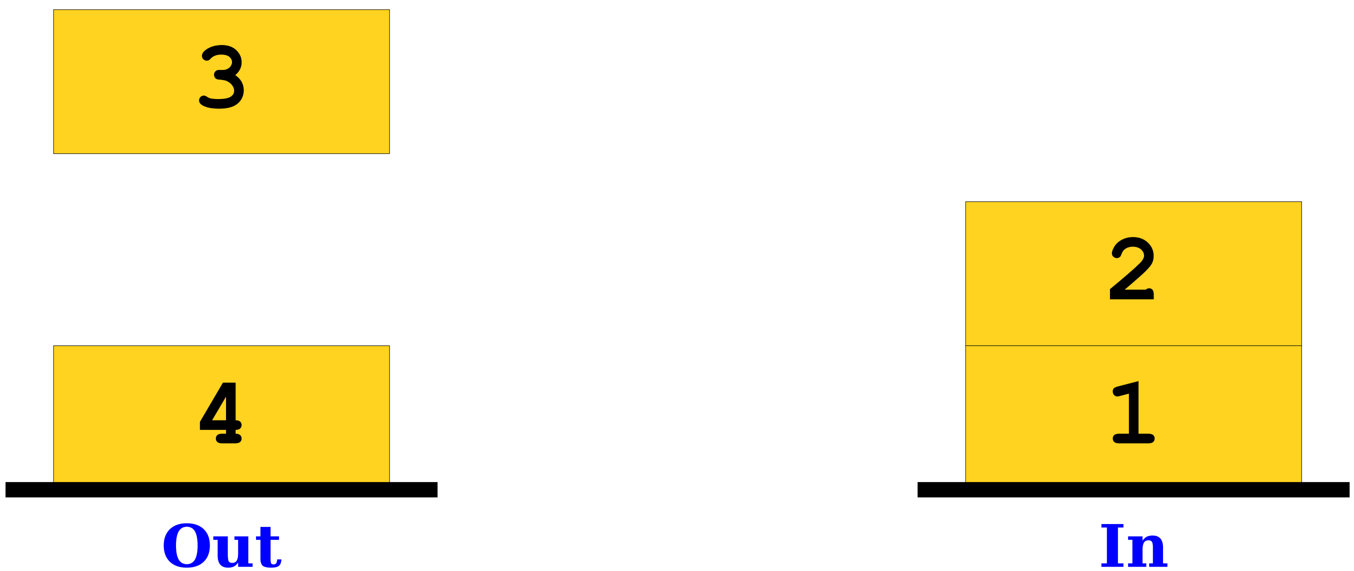
The Two-Stack Queue



The Two-Stack Queue

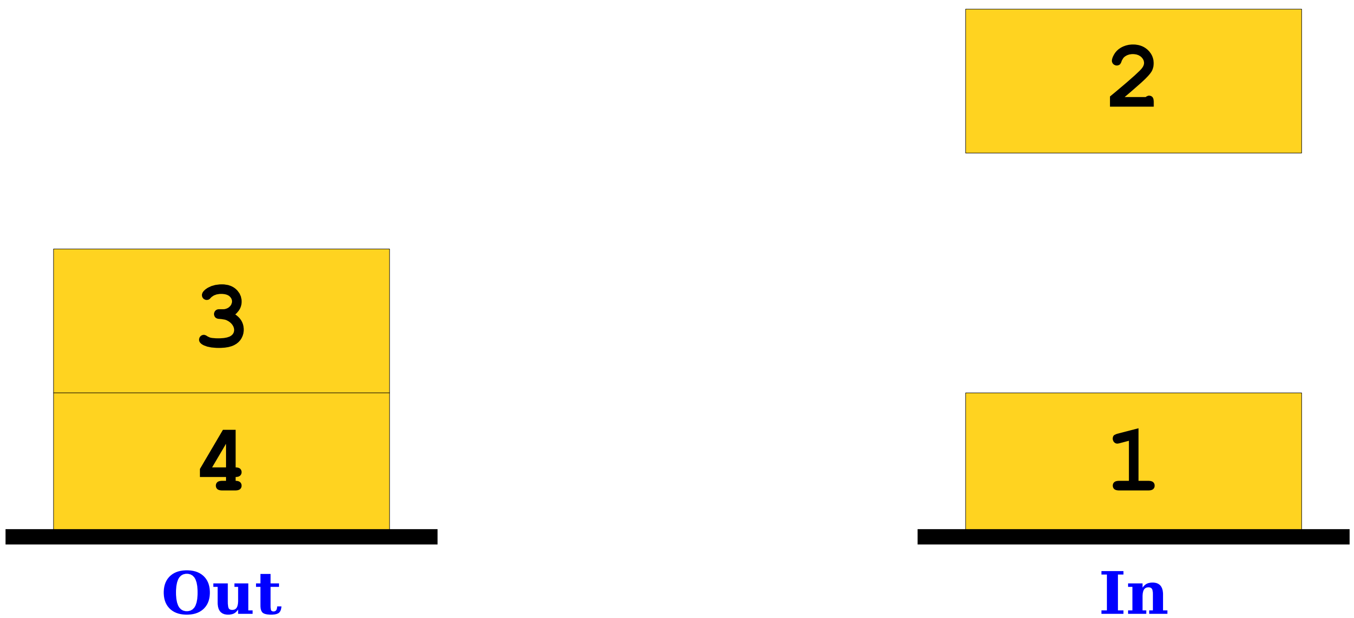


The Two-Stack Queue

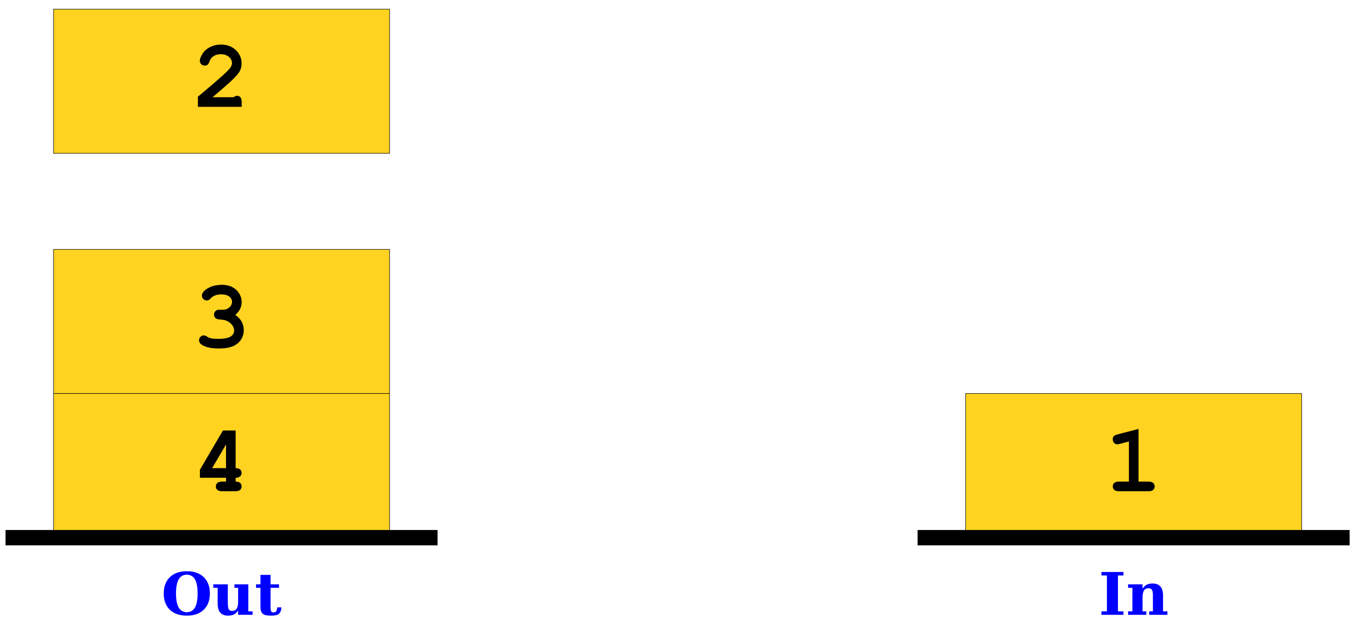


The Two-Stack Queue

The Two-Stack Queue

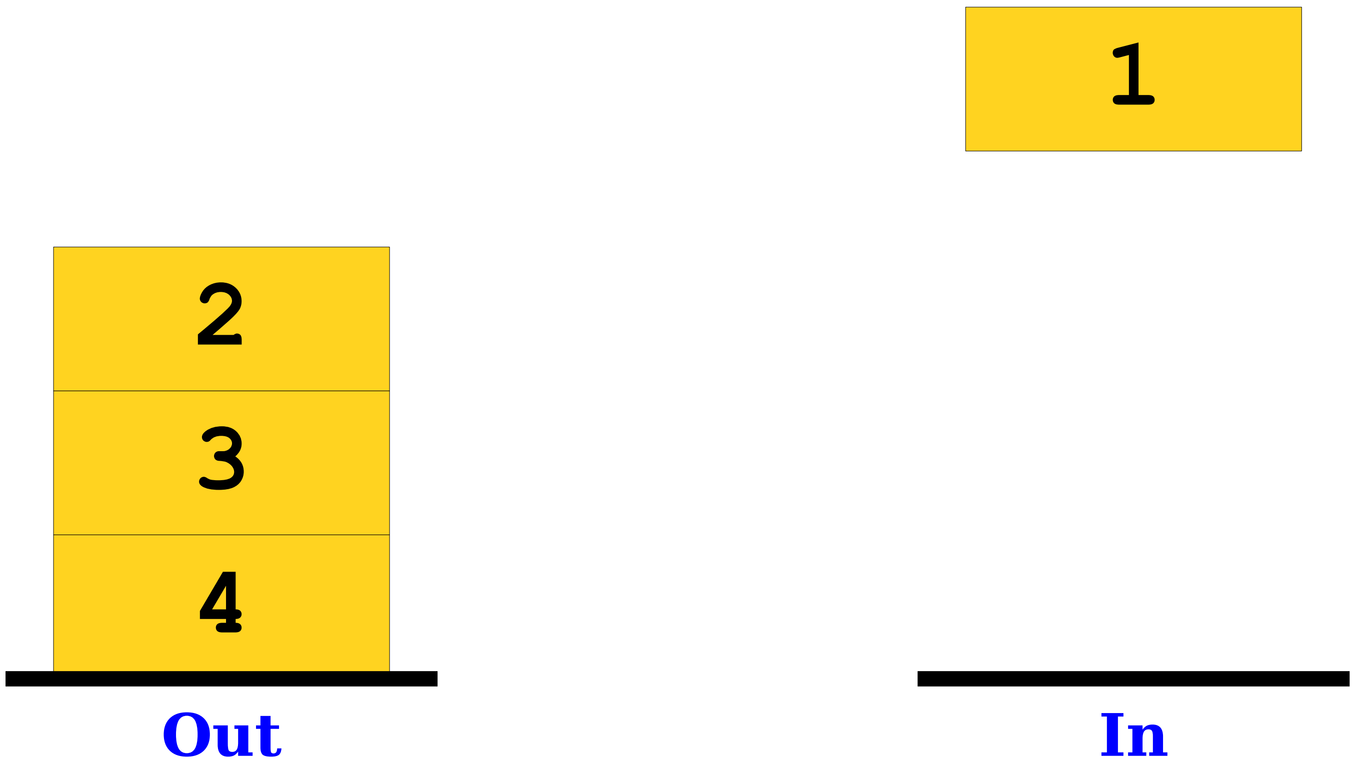


The Two-Stack Queue

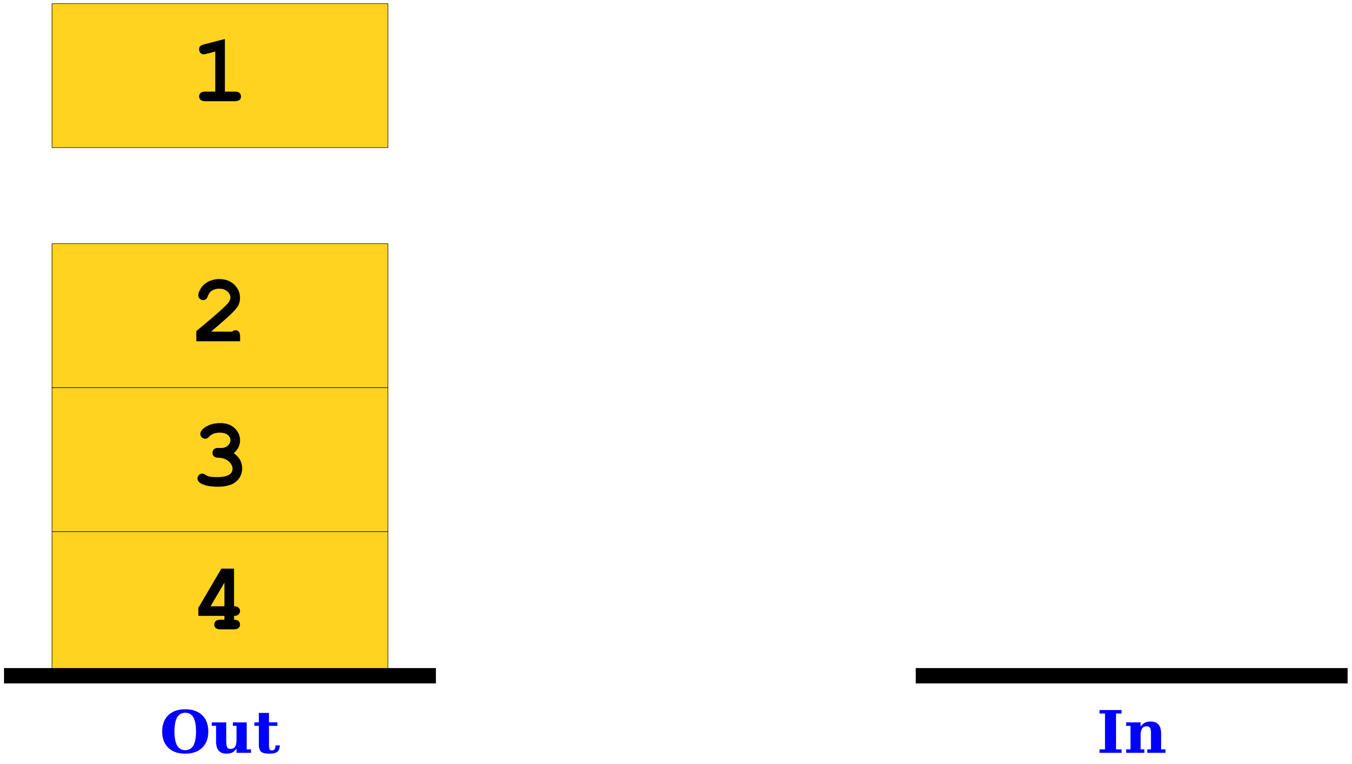


The Two-Stack Queue

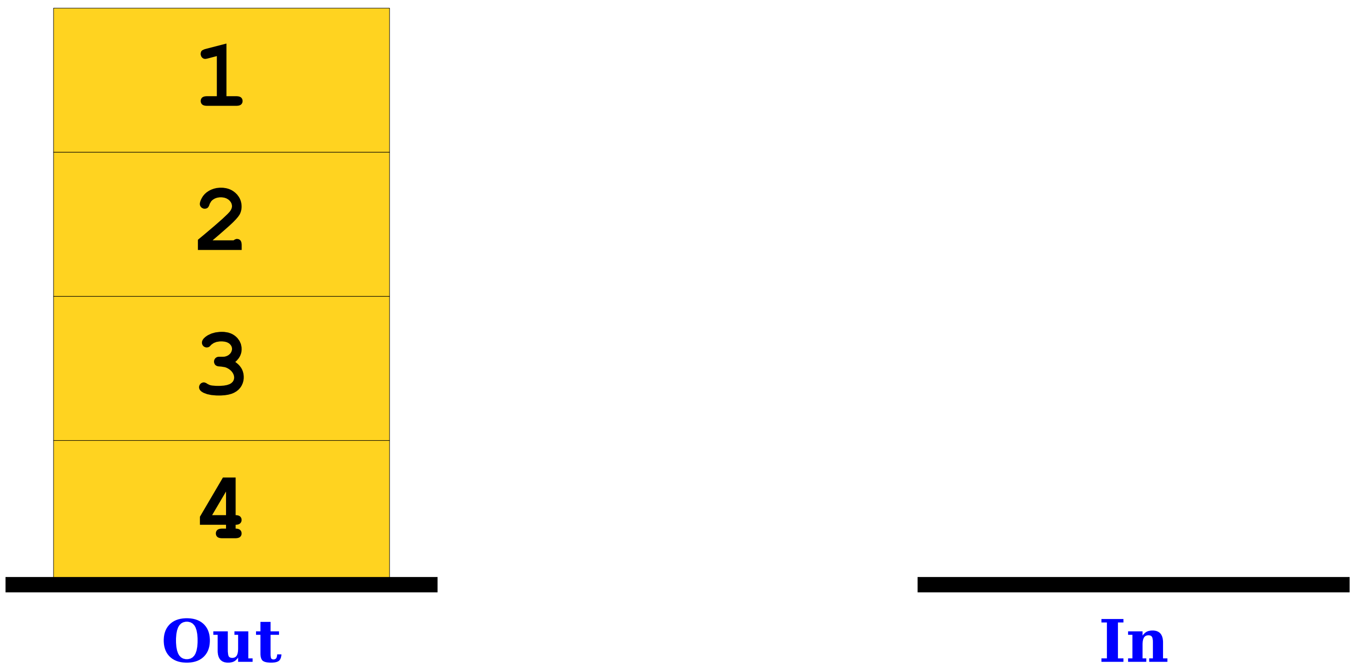
The Two-Stack Queue



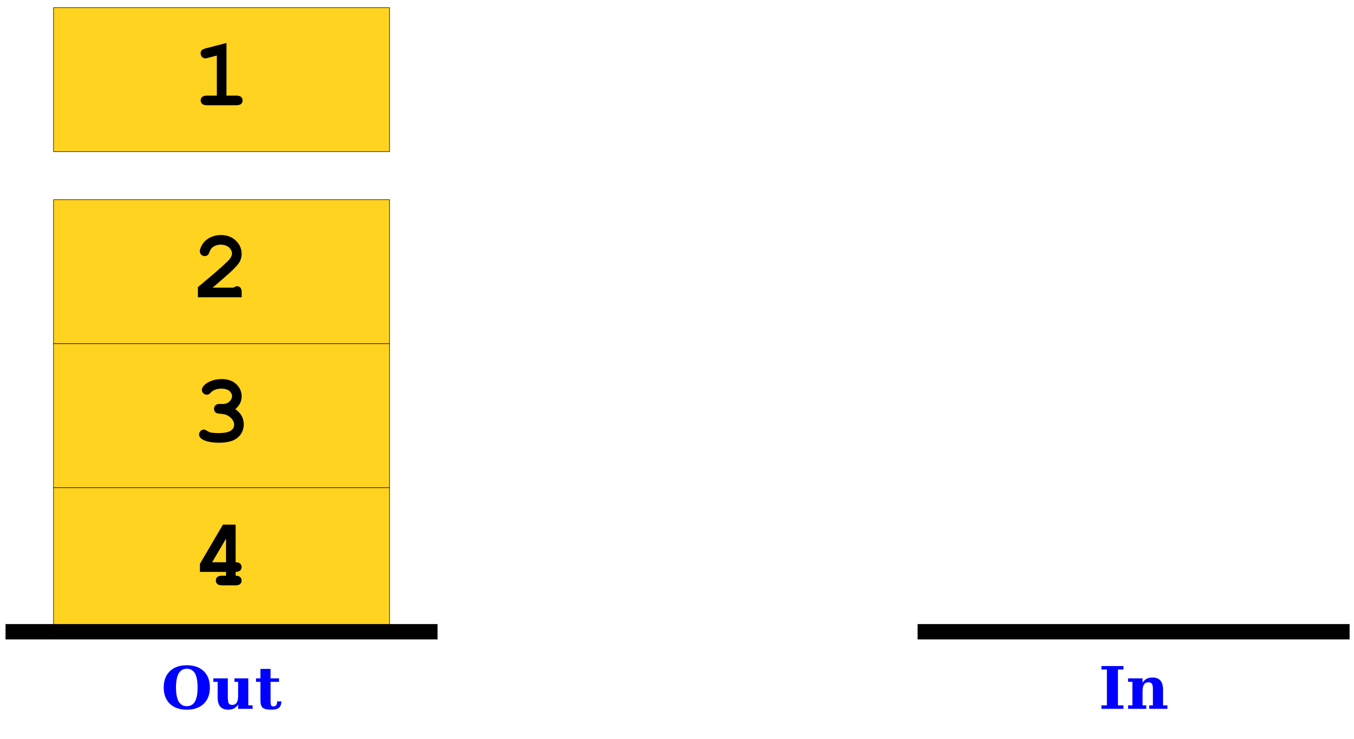
The Two-Stack Queue



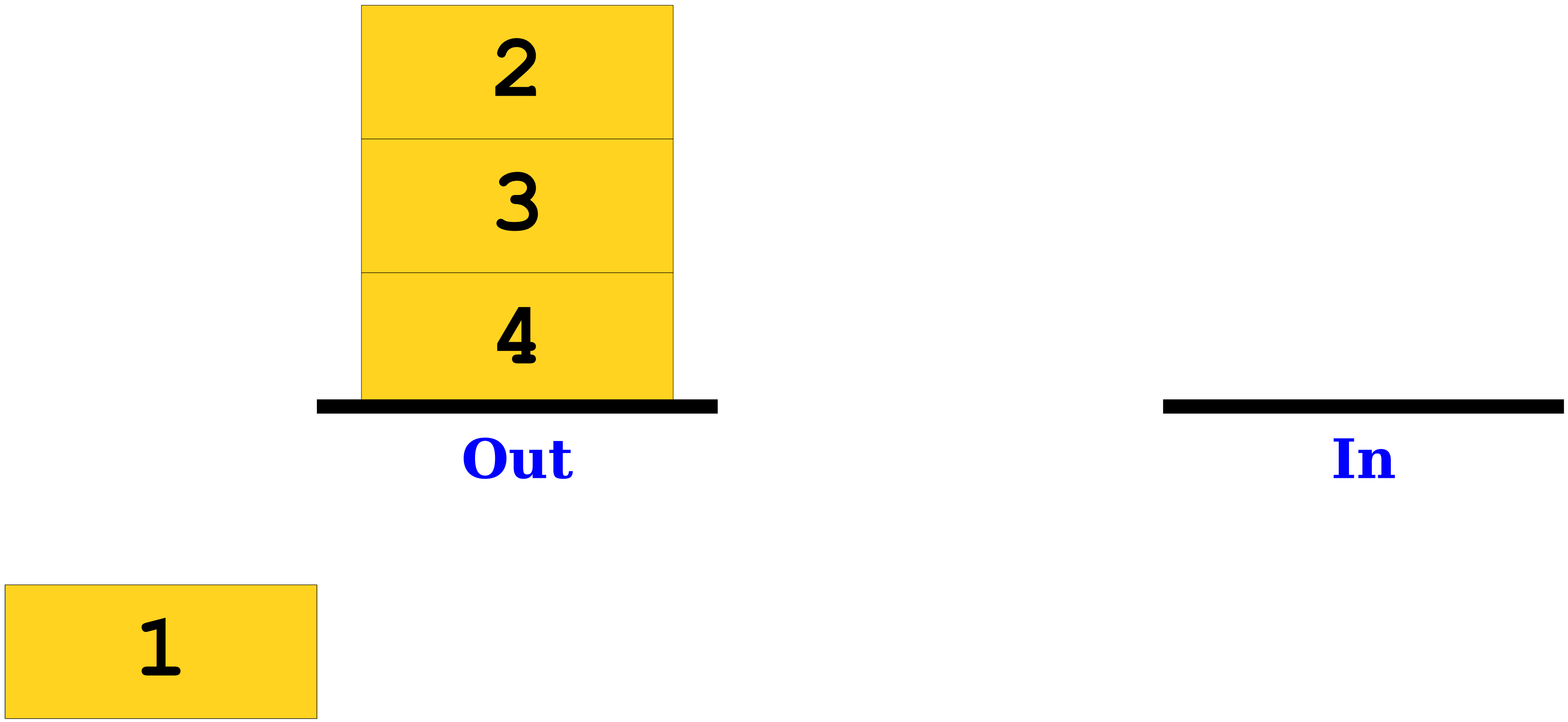
The Two-Stack Queue



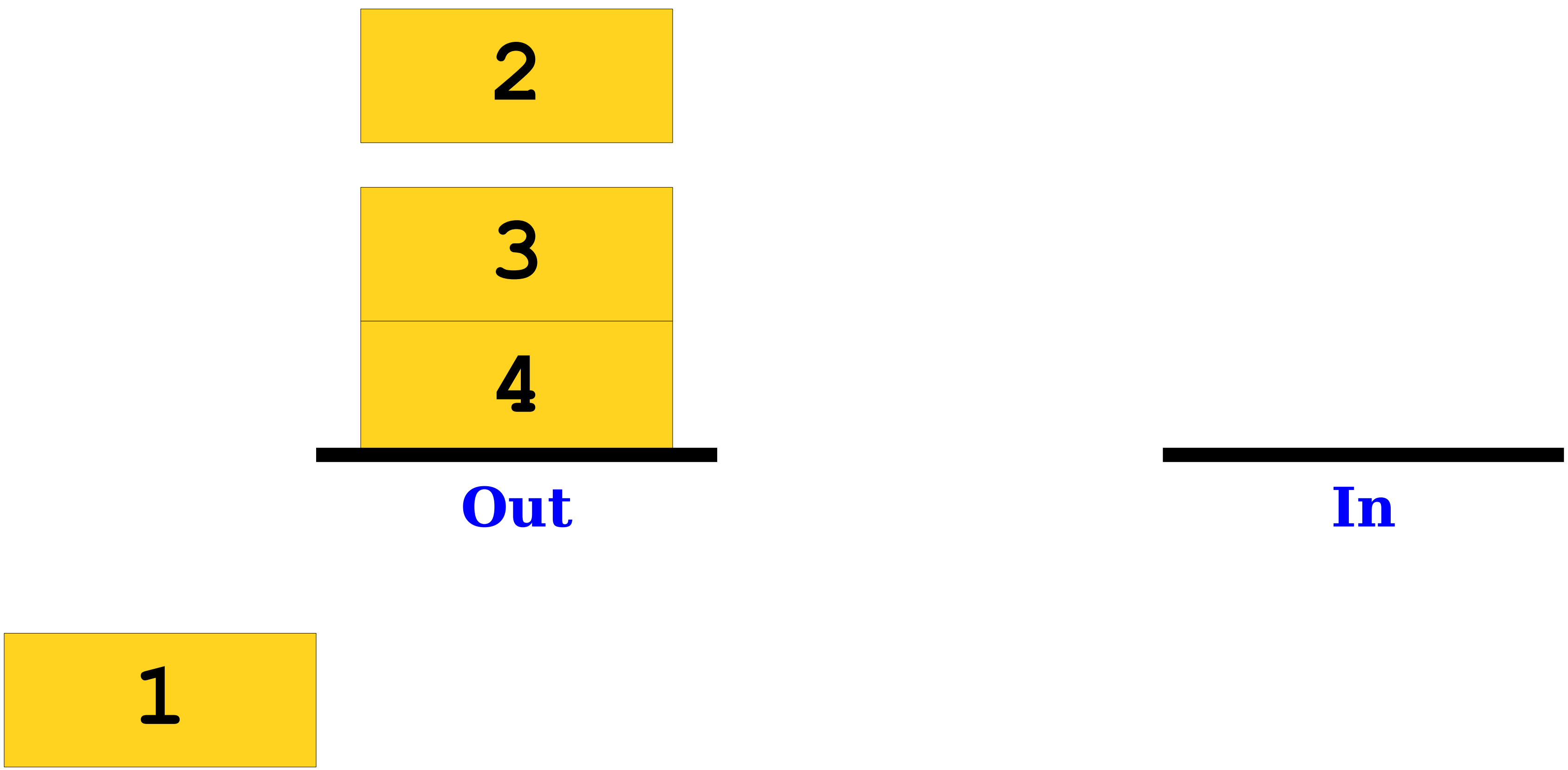
The Two-Stack Queue



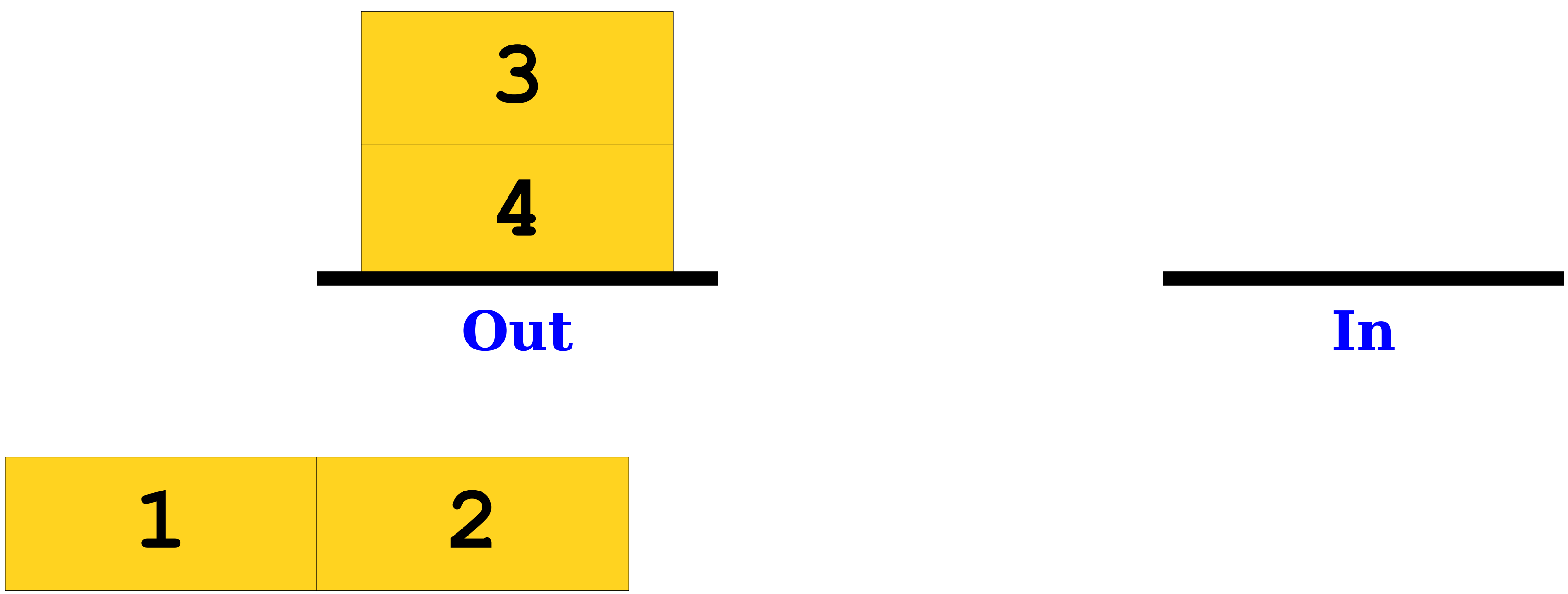
The Two-Stack Queue



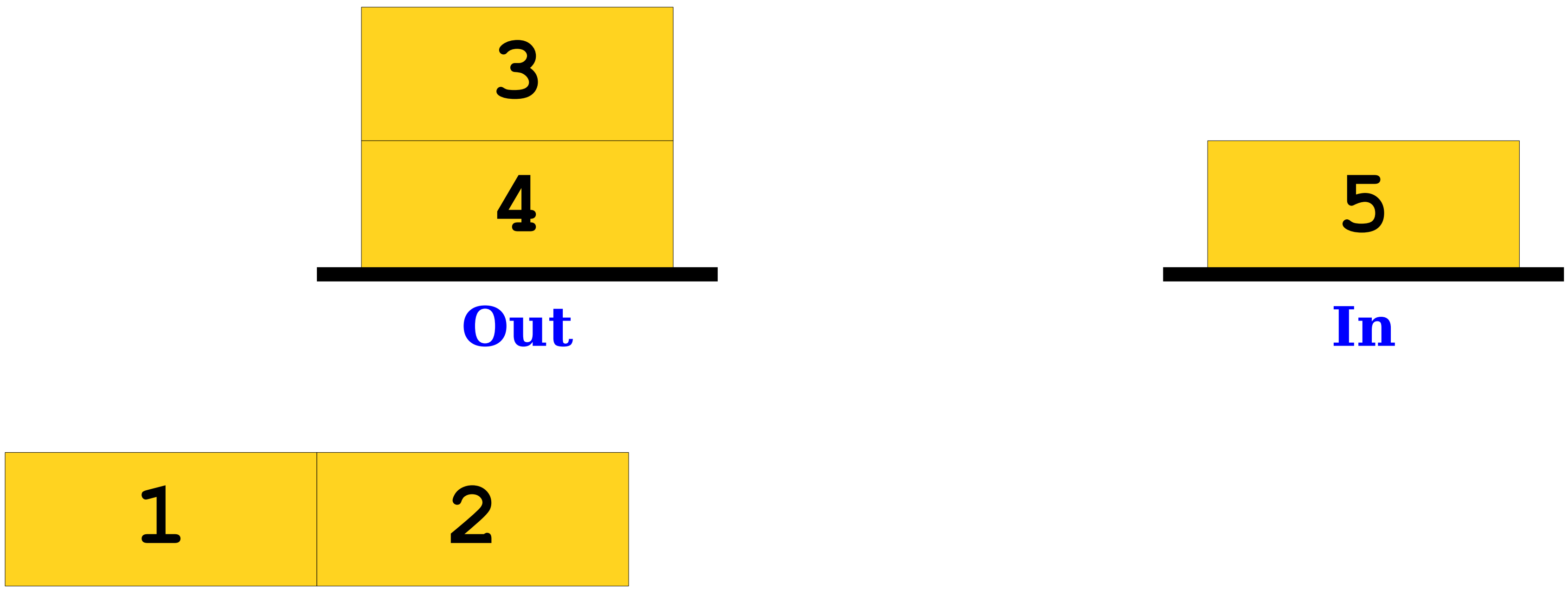
The Two-Stack Queue



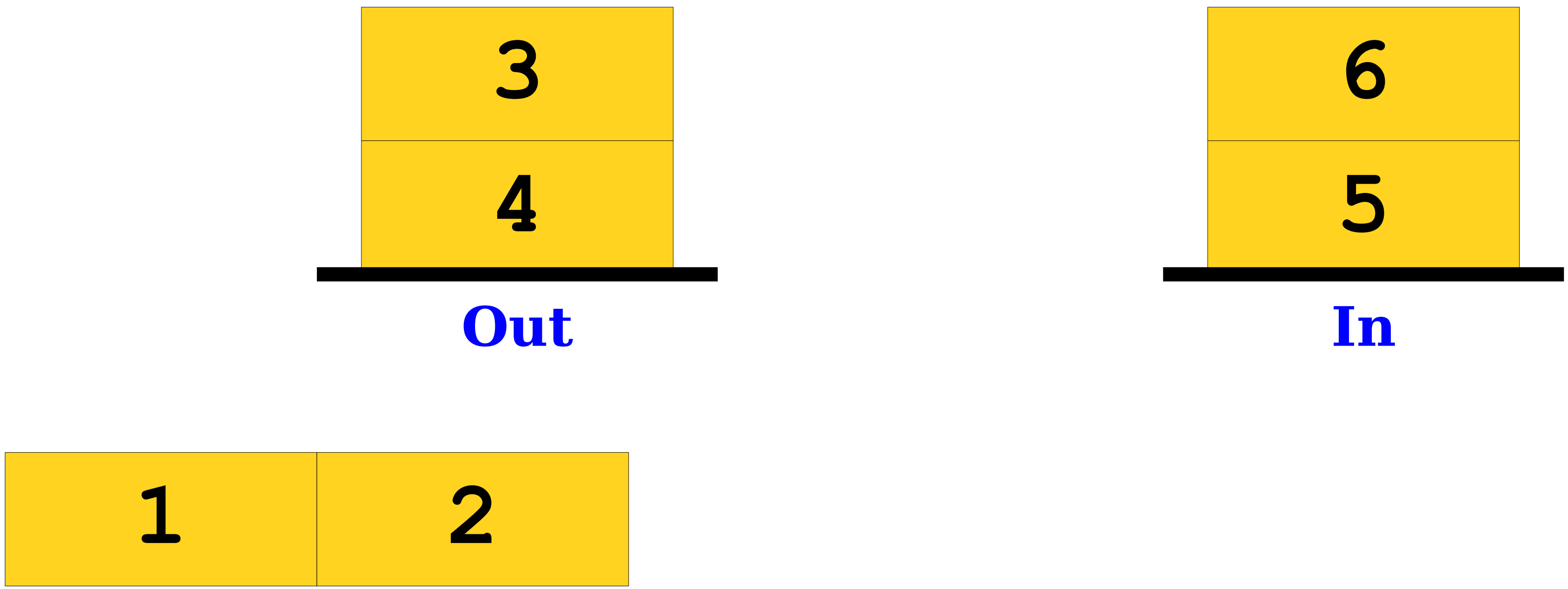
The Two-Stack Queue



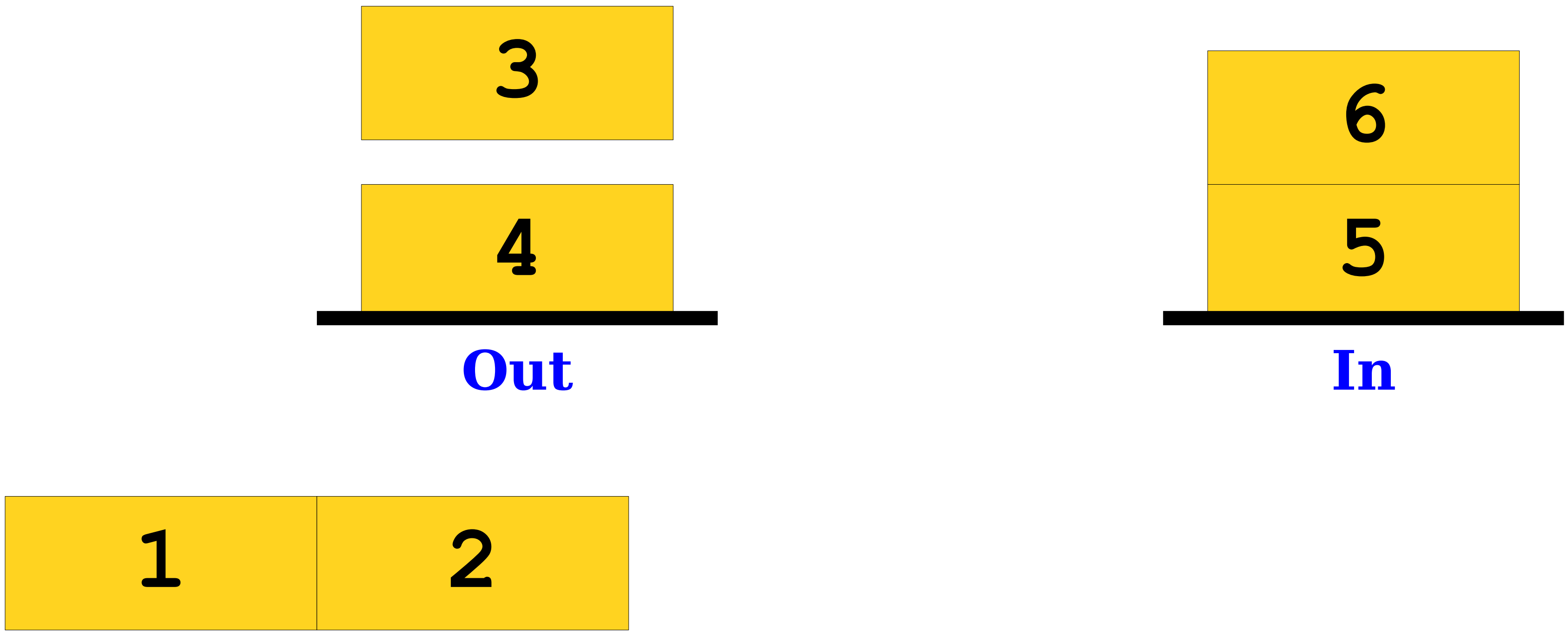
The Two-Stack Queue



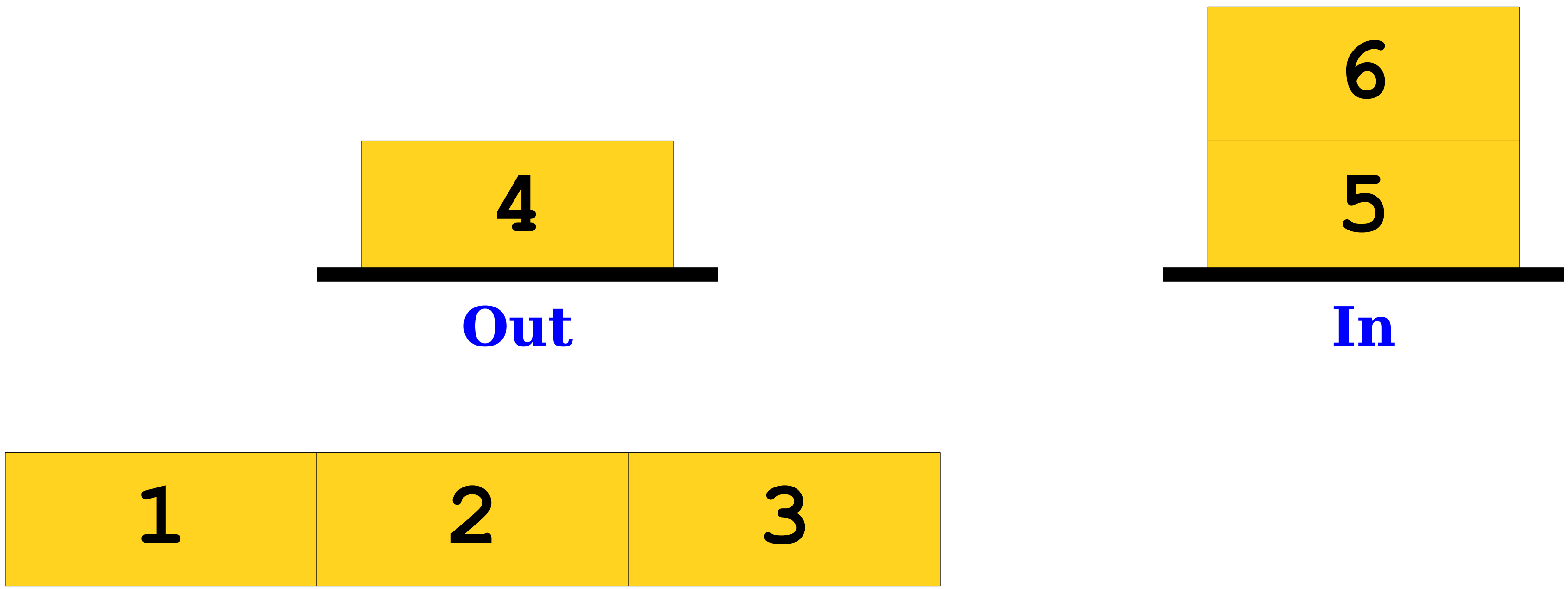
The Two-Stack Queue



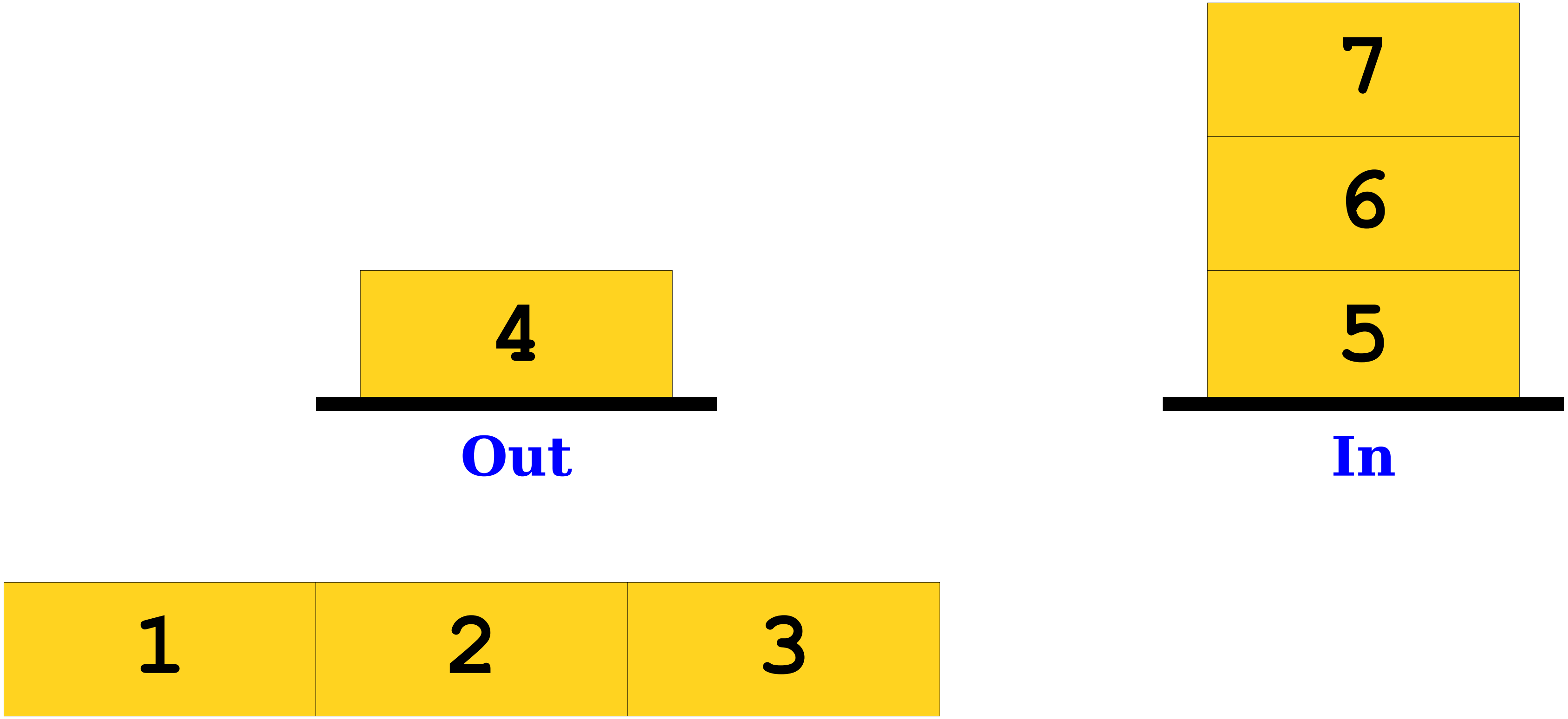
The Two-Stack Queue



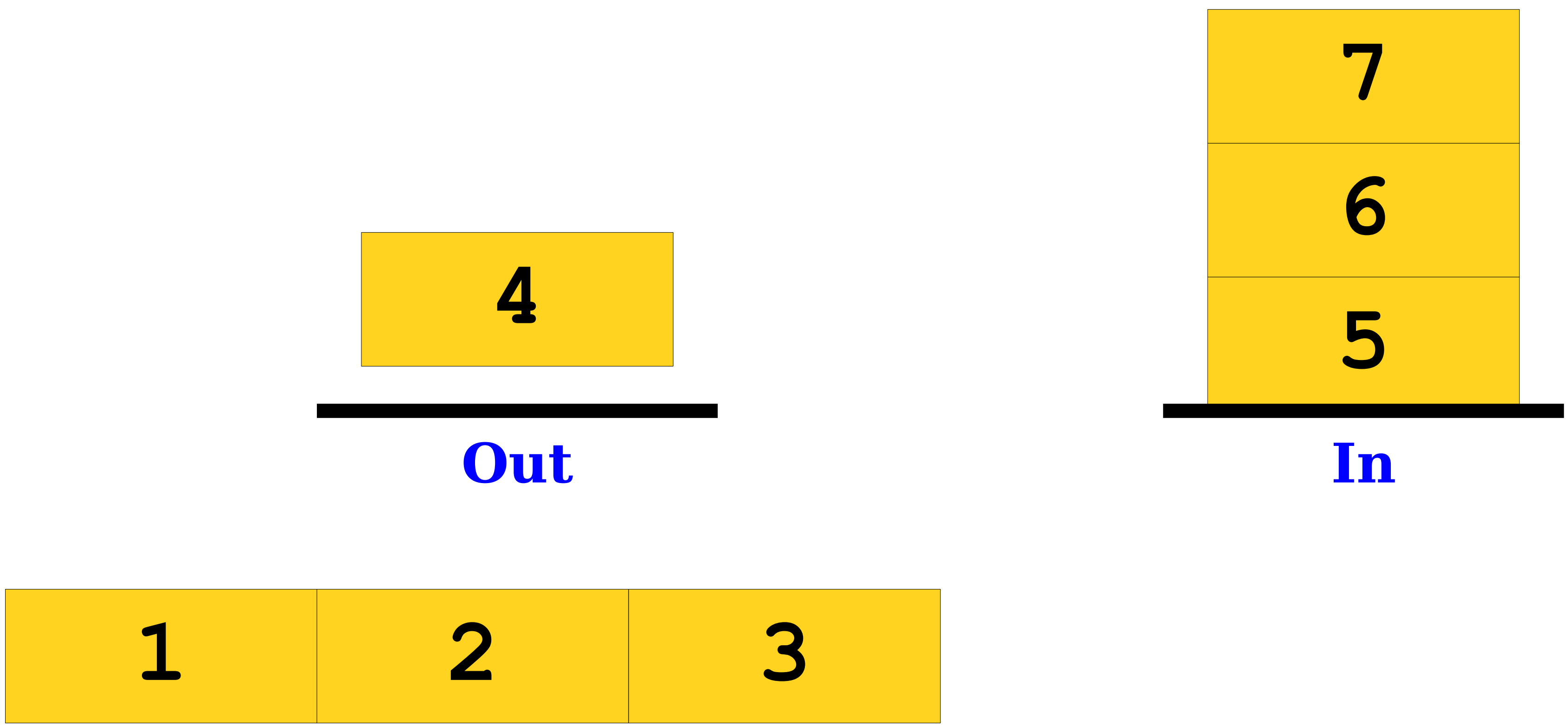
The Two-Stack Queue



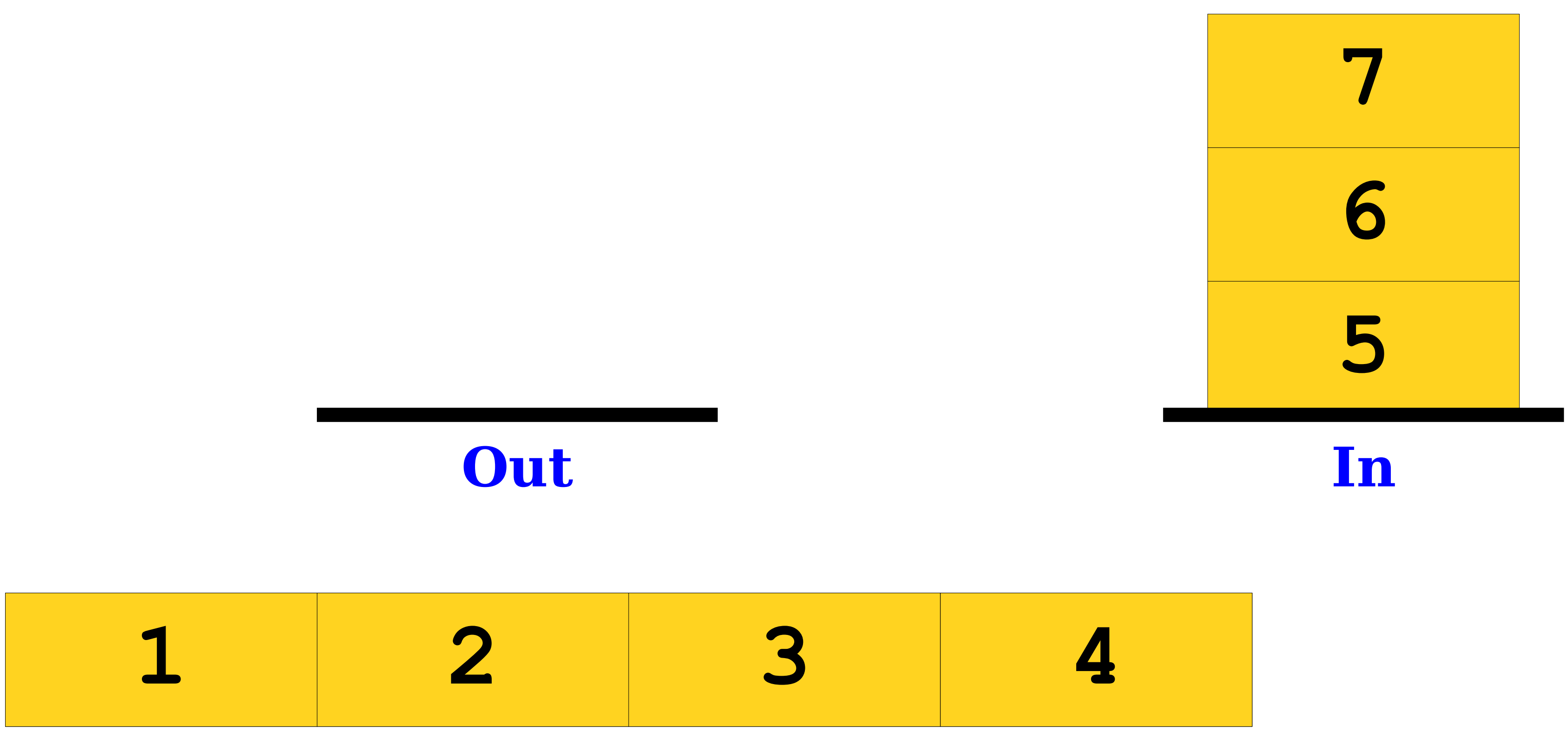
The Two-Stack Queue



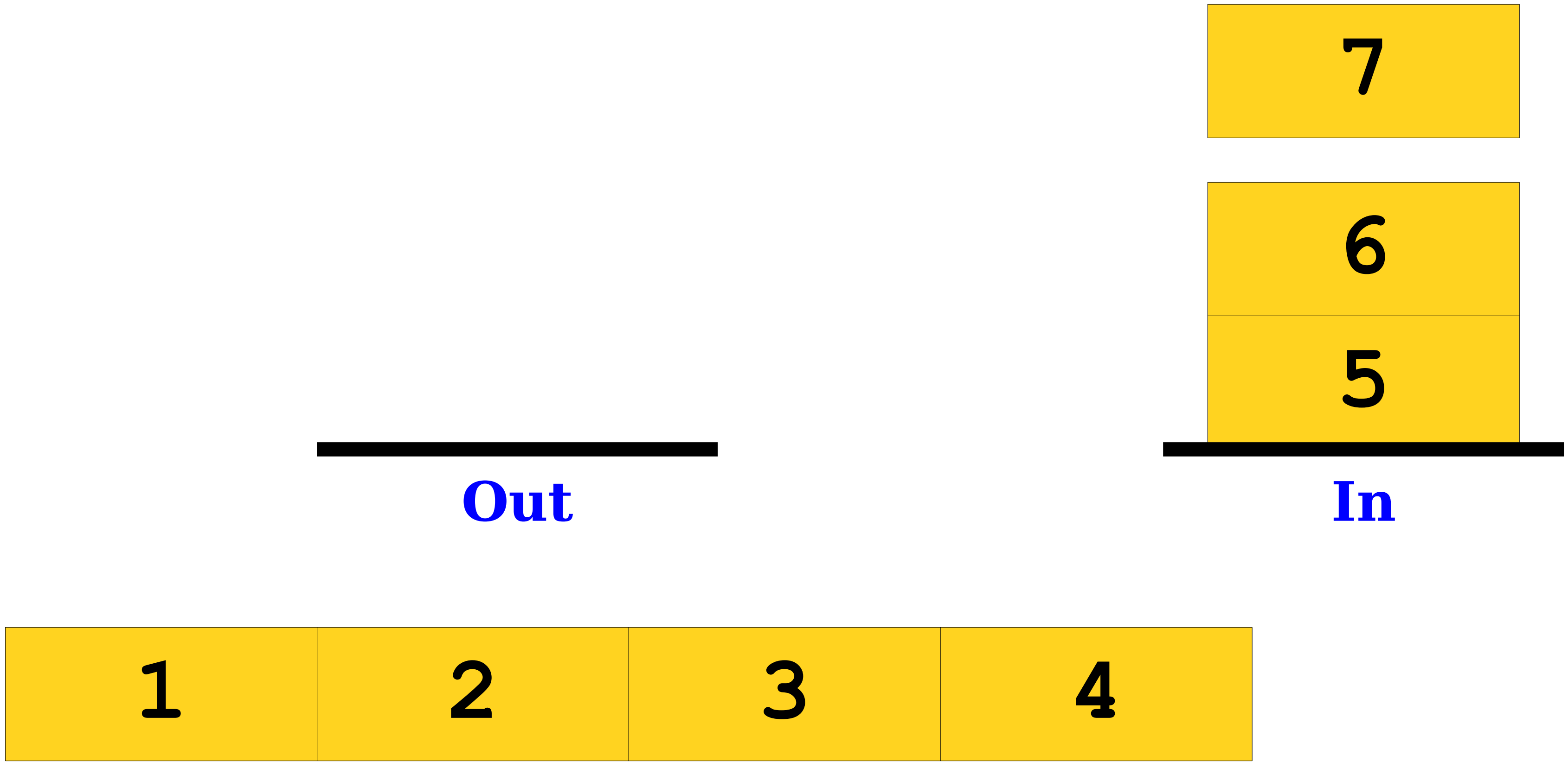
The Two-Stack Queue



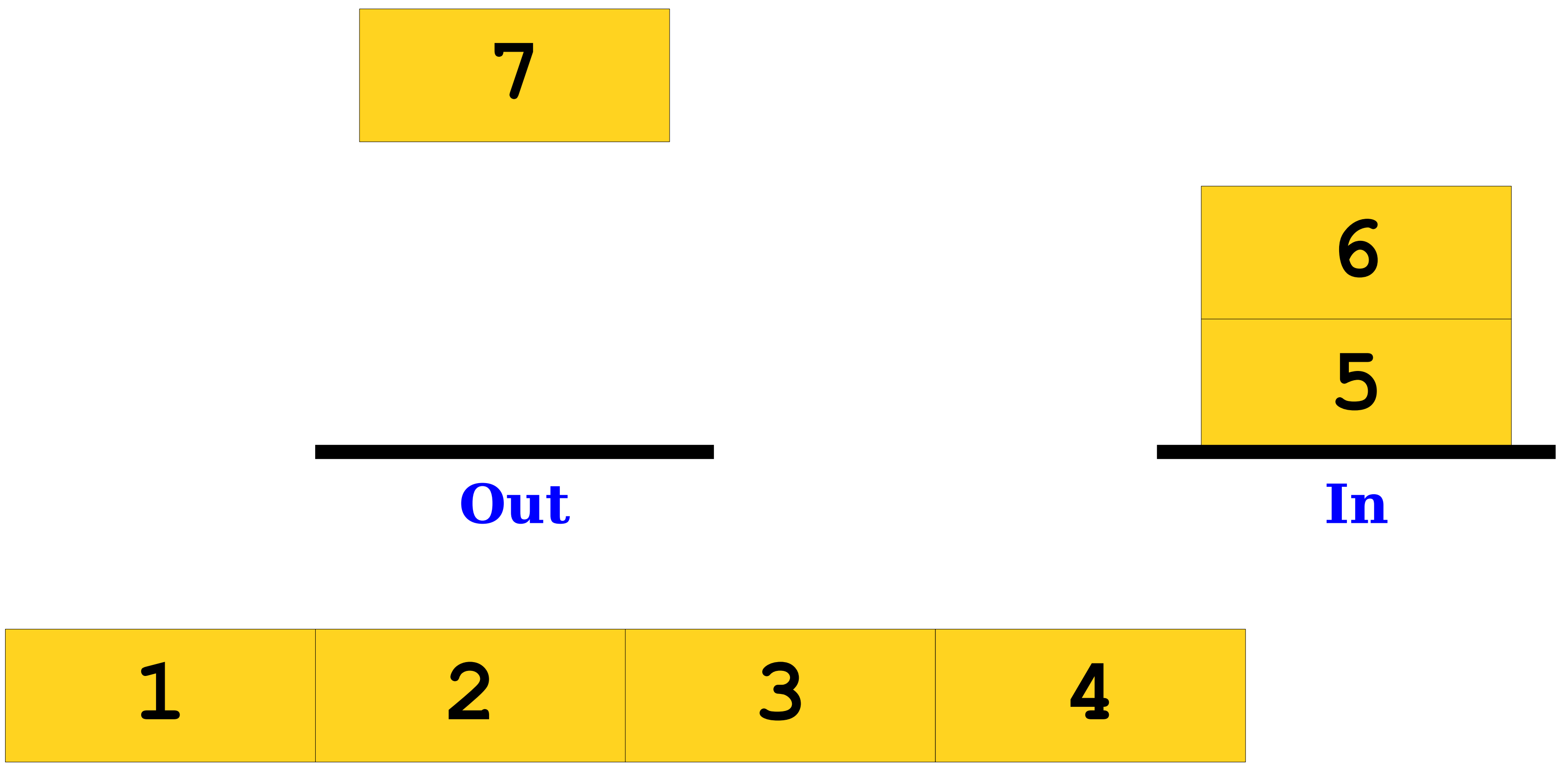
The Two-Stack Queue



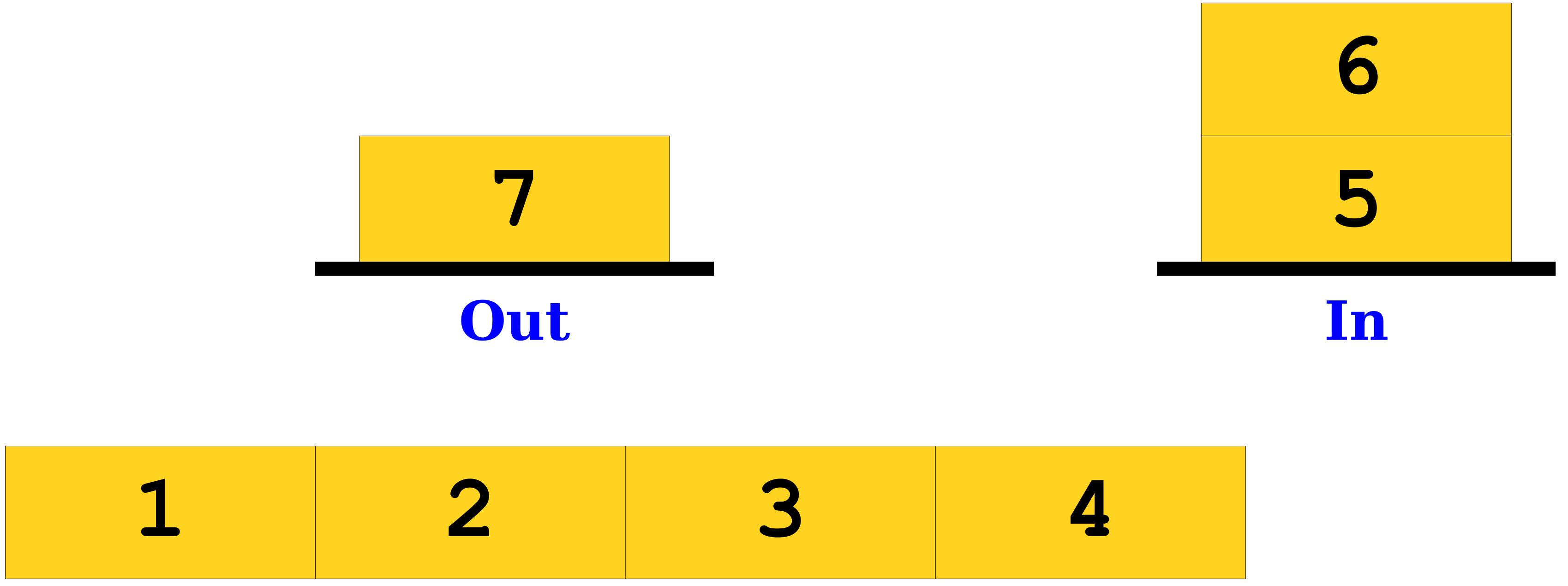
The Two-Stack Queue



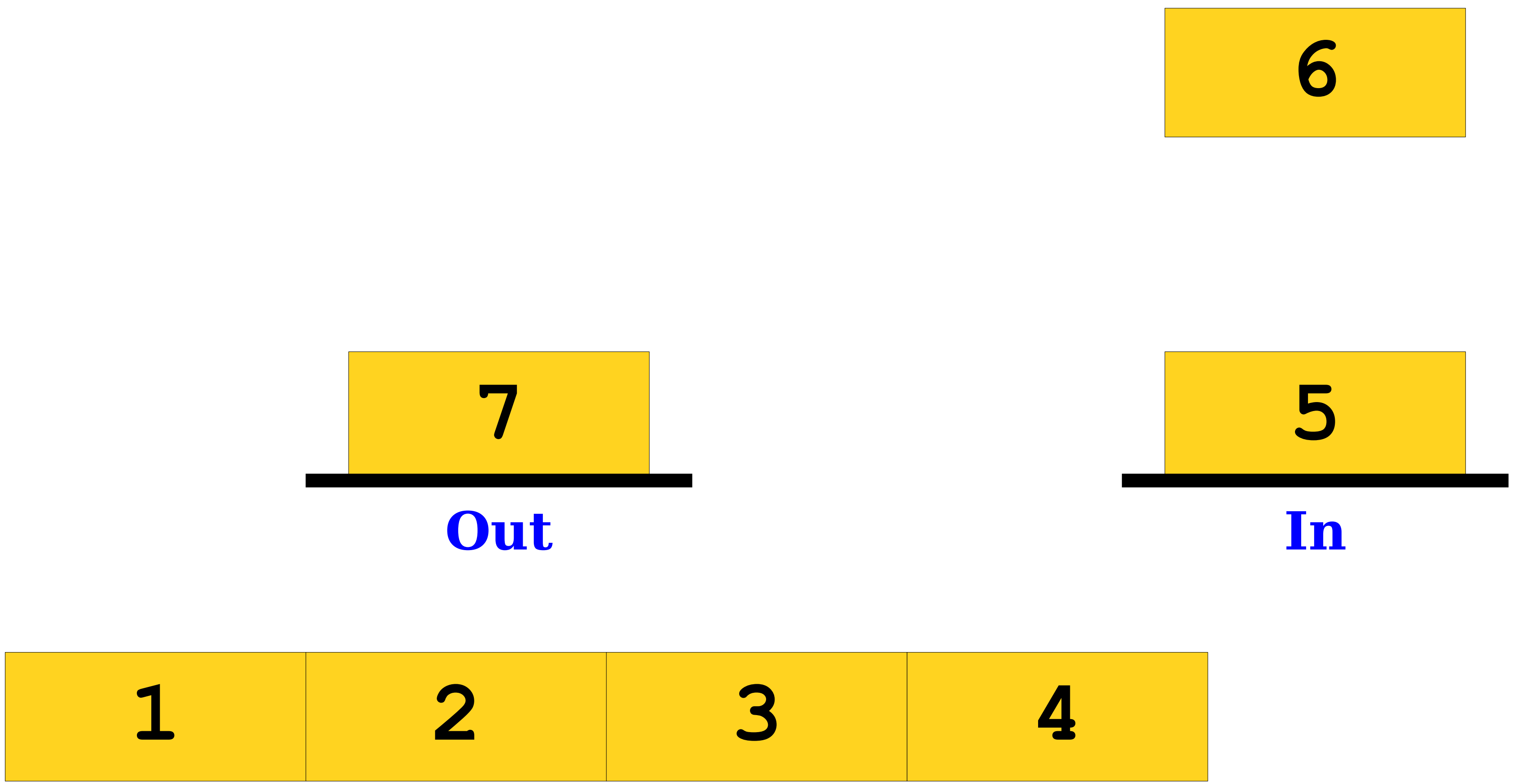
The Two-Stack Queue



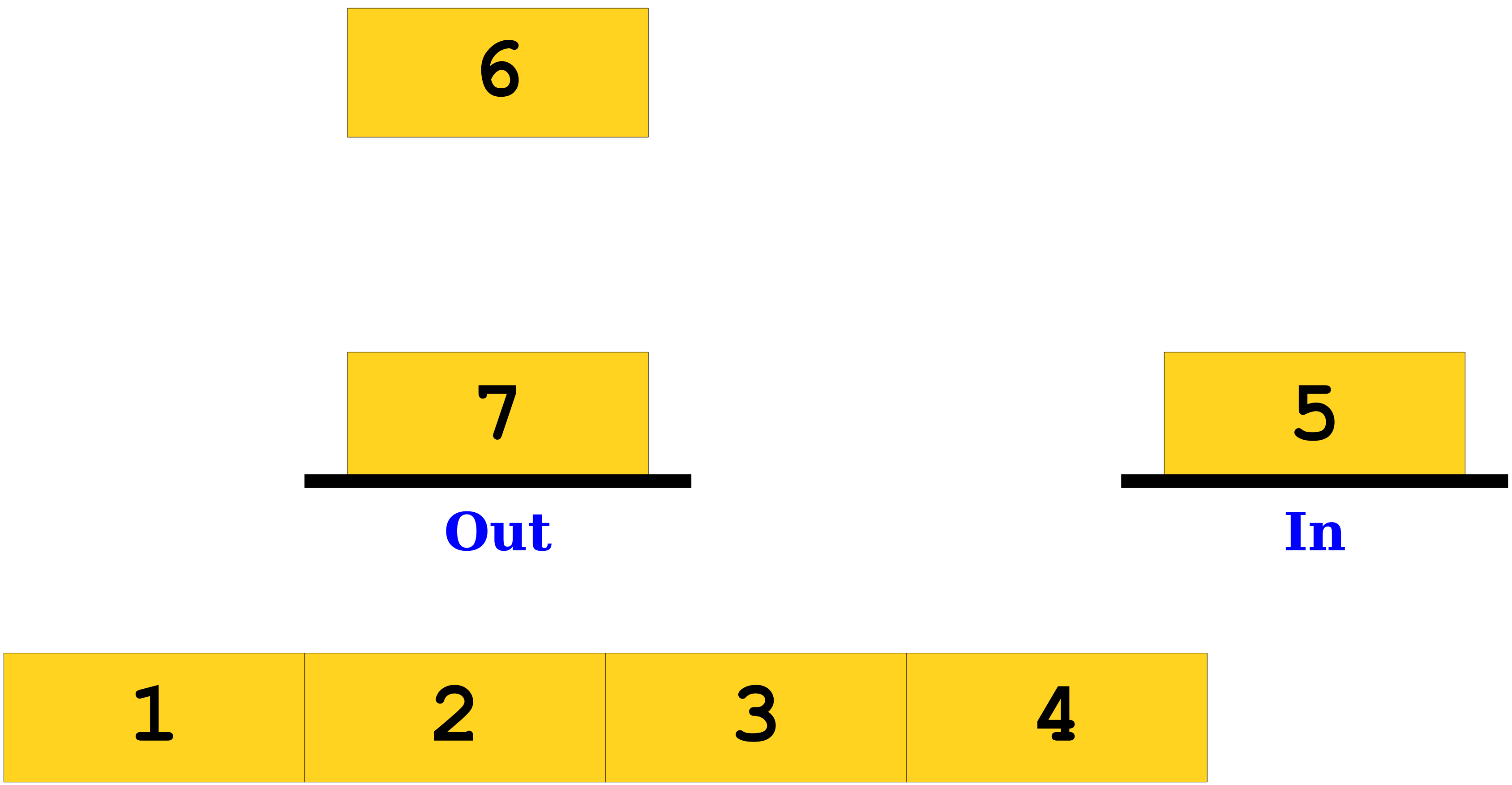
The Two-Stack Queue



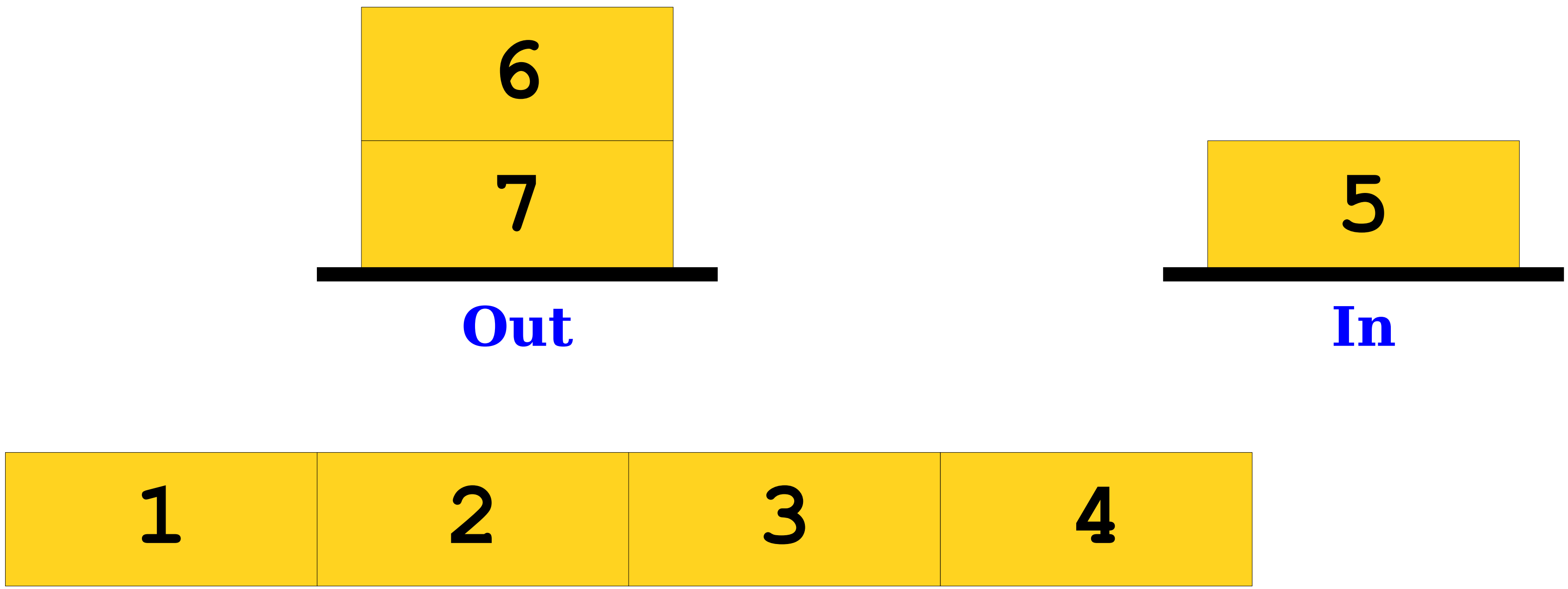
The Two-Stack Queue



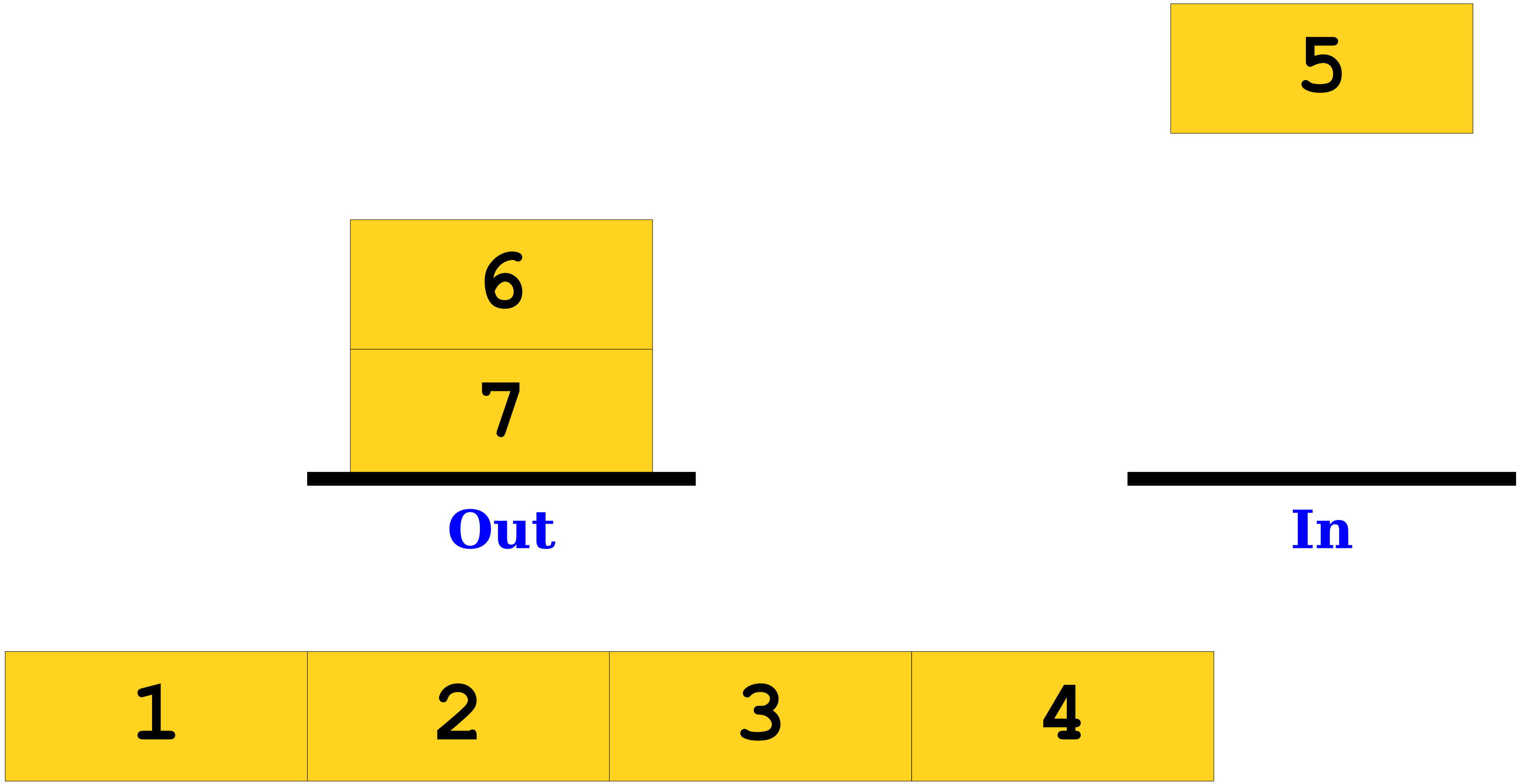
The Two-Stack Queue



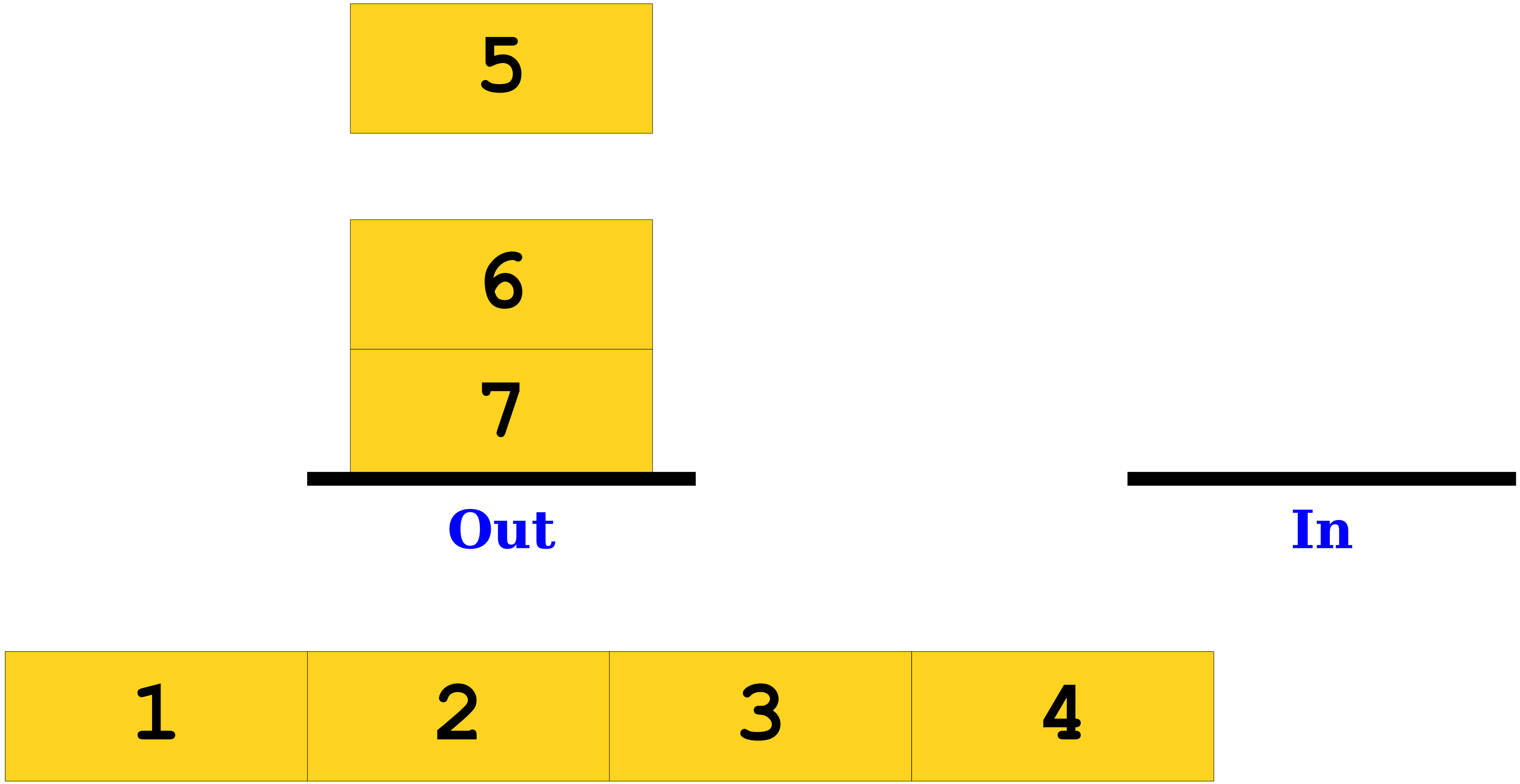
The Two-Stack Queue



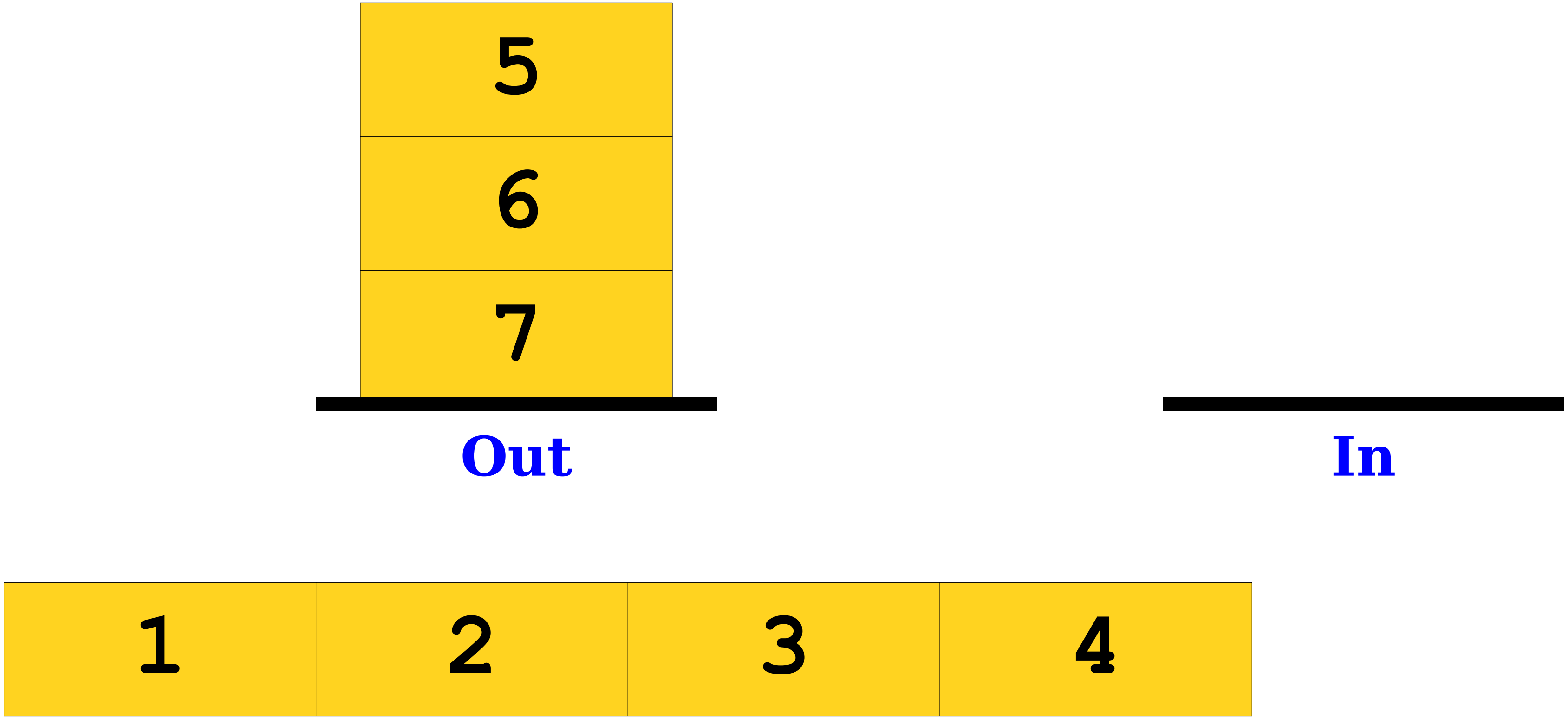
The Two-Stack Queue



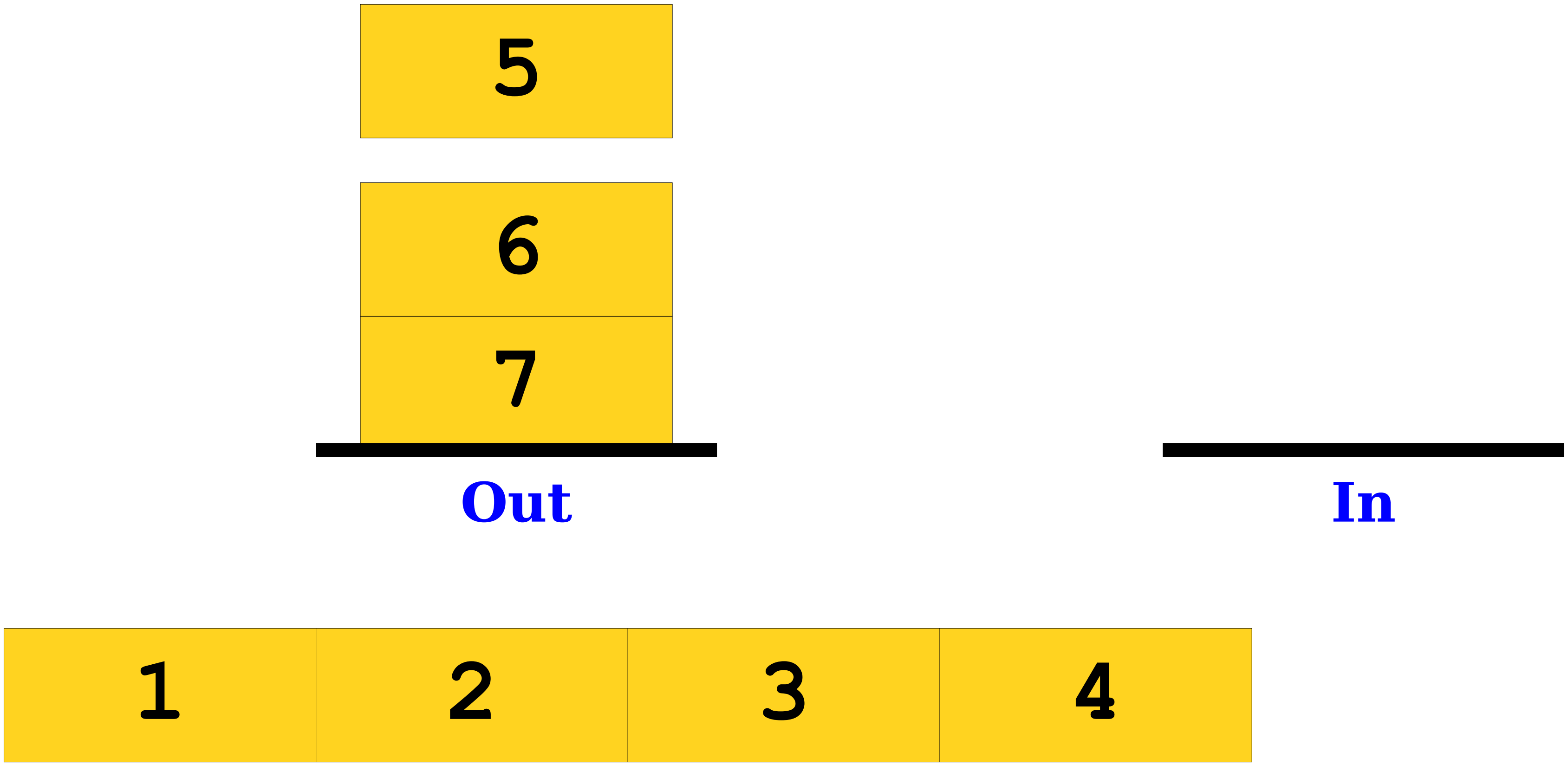
The Two-Stack Queue



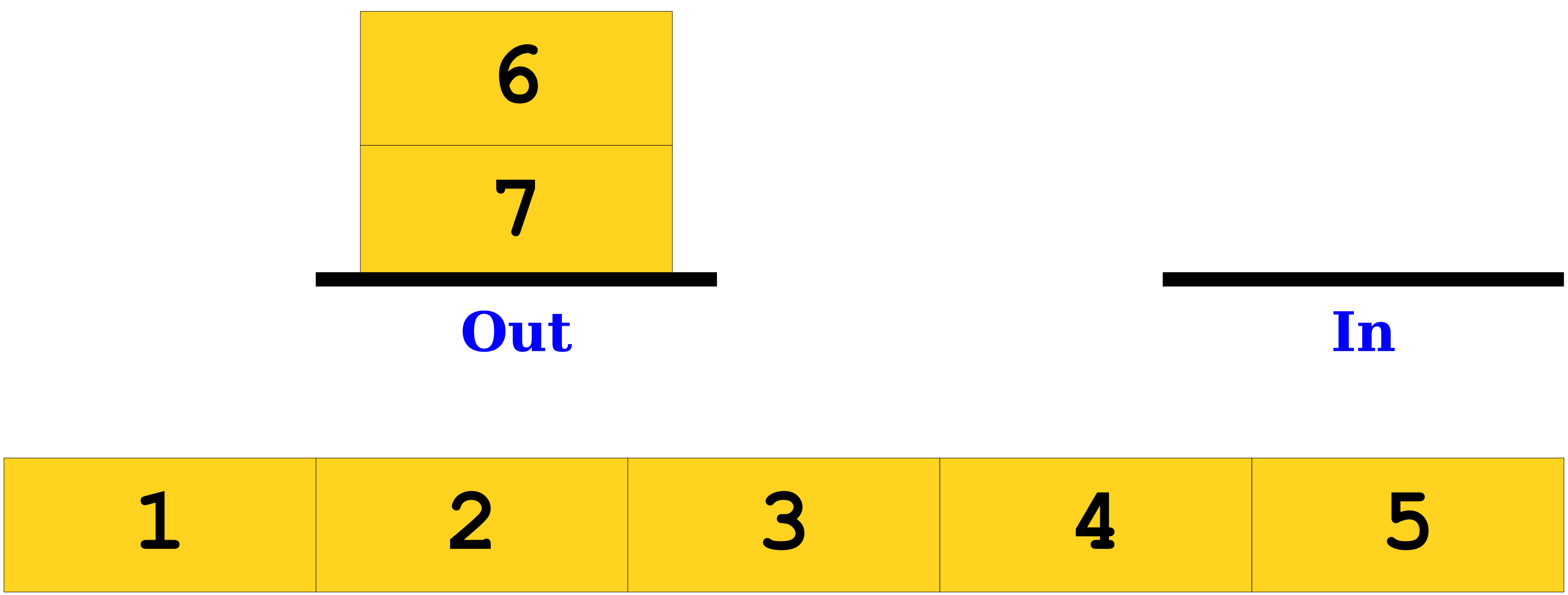
The Two-Stack Queue



The Two-Stack Queue



The Two-Stack Queue



The Two-Stack Queue

- Maintain two stacks, an **In** stack and an **Out** stack.
- To enqueue an element, push it onto the **In** stack.
- To dequeue an element:
  - If the **Out** stack is empty, pop everything off the **In** stack and push it onto the **Out** stack.
  - Pop the **Out** stack and return its value.

Let's code it up...

Analyzing Efficiency

- How efficient is our two-stack queue?
- All enqueues just do one push.
- A dequeue might do a lot of pushes *and* a lot of pops.
- However, let's do an amortized analysis:
  - Each element is pushed at most twice and popped at most twice.
  - n enqueues and n dequeues thus do at most $4n$ pushes and pops.
  - Any $4n$ pushes / pops takes $O(n)$ amortized time.
  - Amortized cost: **O(1)** per operation.

Exam Notes

- Out of 71 Points
  - Mean: 56
  - Standard Deviation: 13
- If you have any concerns with your grade then come chat with me at some point.

Next Time

- **Linked Lists**
  - A different way to represent sequences of elements.
- **Dynamic Allocation Revisited**
  - What else can we allocate?