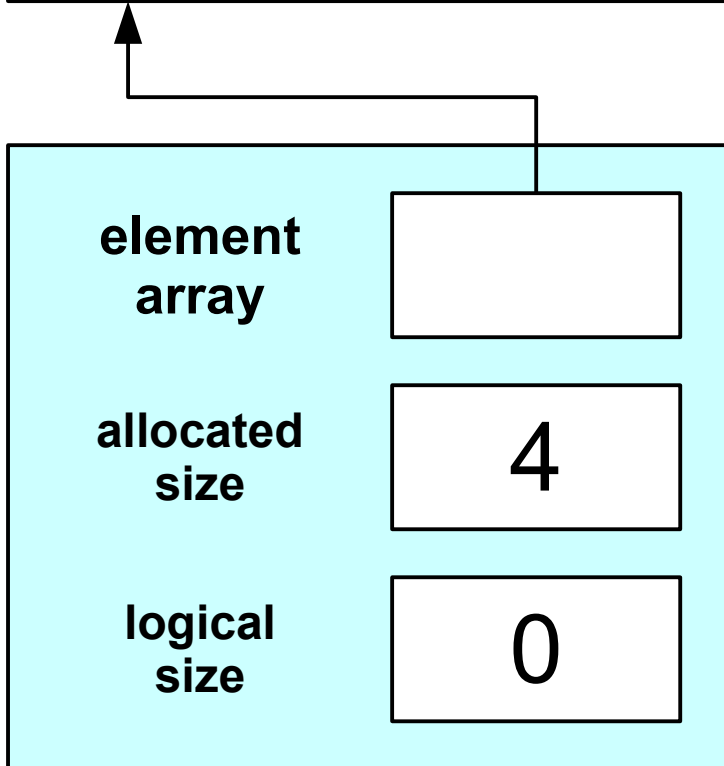
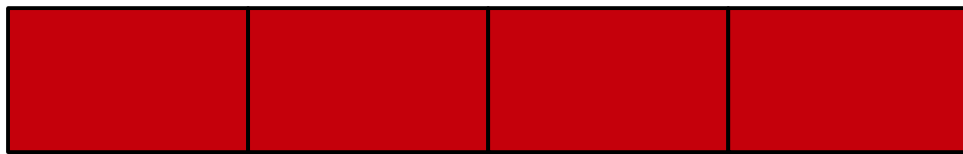


# Implementing Abstractions

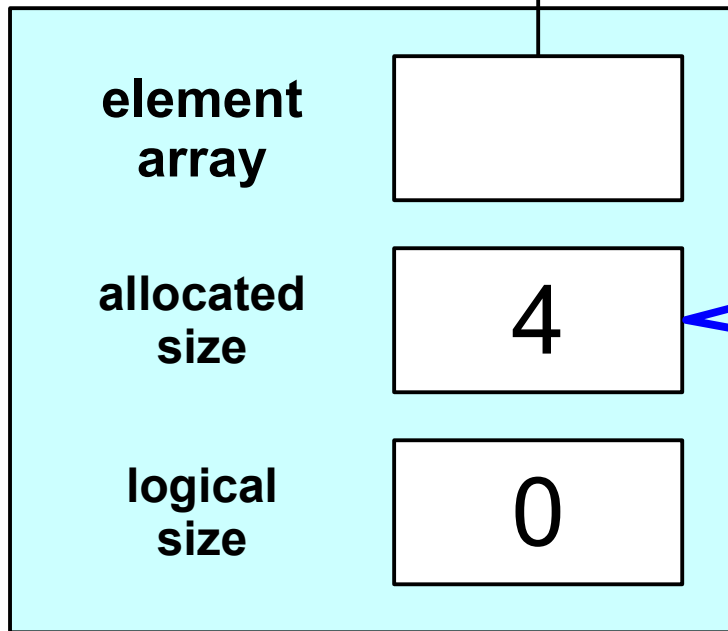
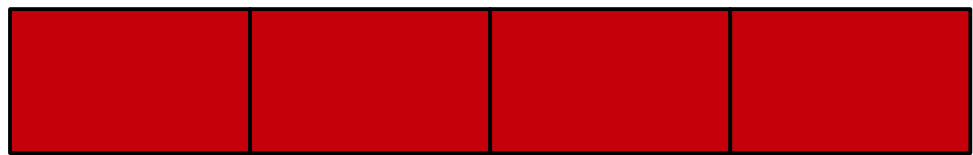
## Part Two

Previously, on CS106B...

# A Bounded Stack

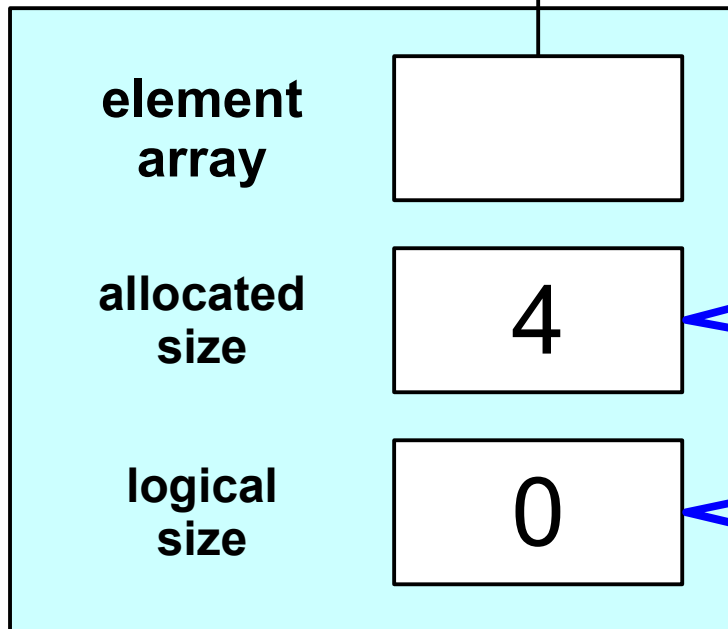
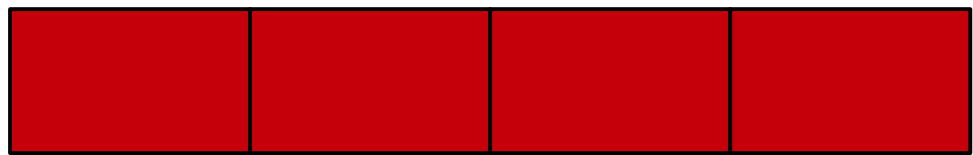


# A Bounded Stack



The stack's *allocated size* is the number of slots in the array. Remember - arrays in C++ cannot grow or shrink.

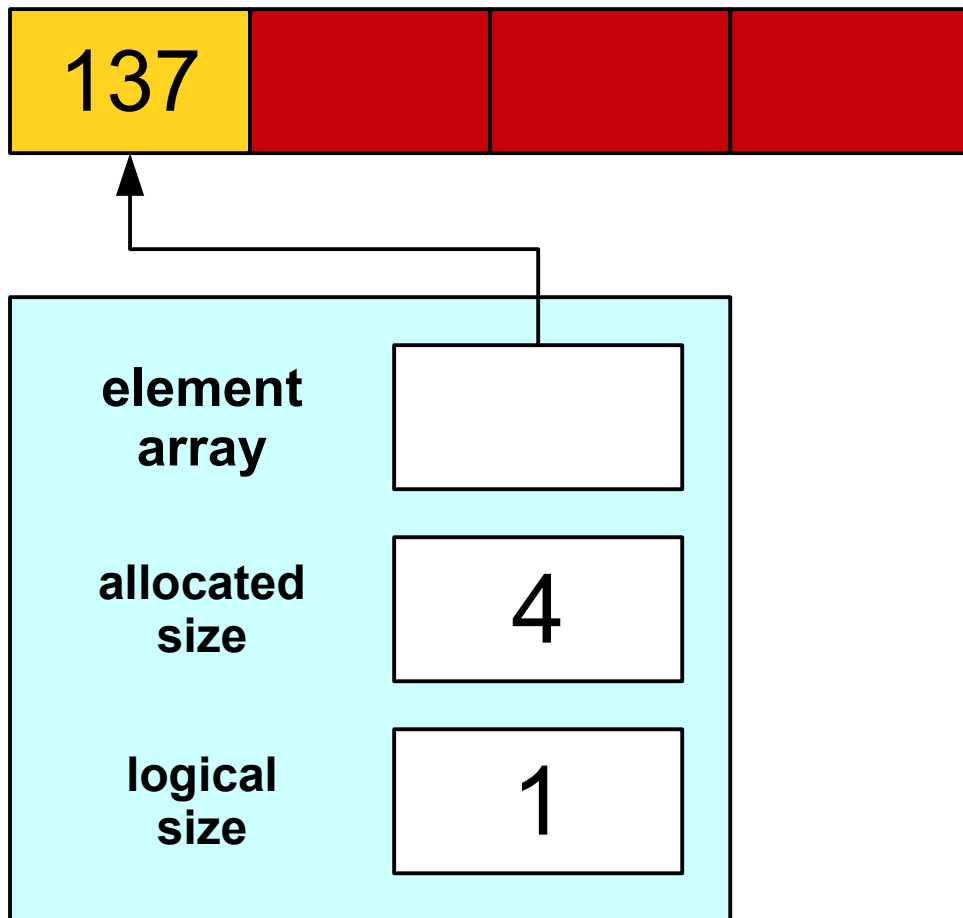
# A Bounded Stack



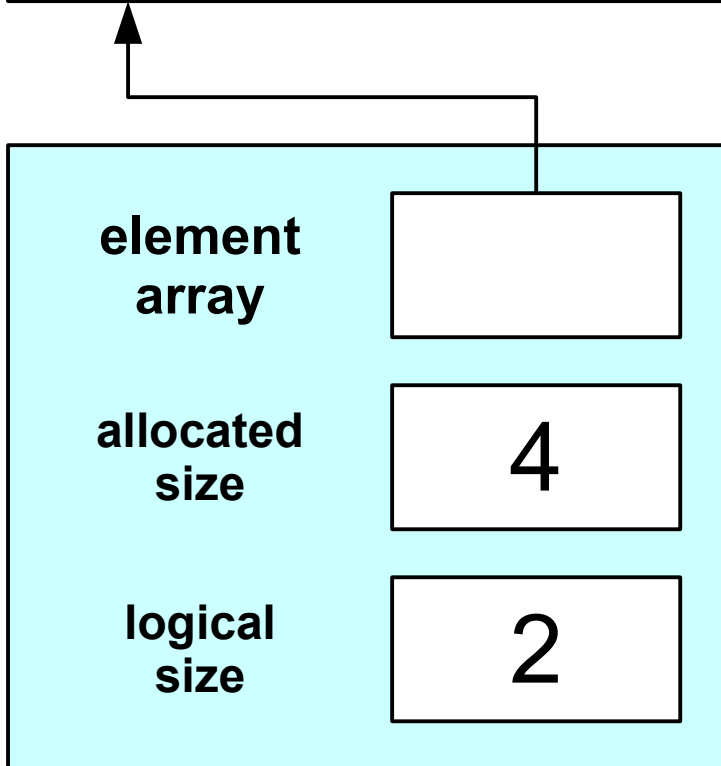
The stack's **allocated size** is the number of slots in the array. Remember - arrays in C++ cannot grow or shrink.

The stack's **logical size** is the number of elements actually stored in the stack. This lets us track how much space we're actually using.

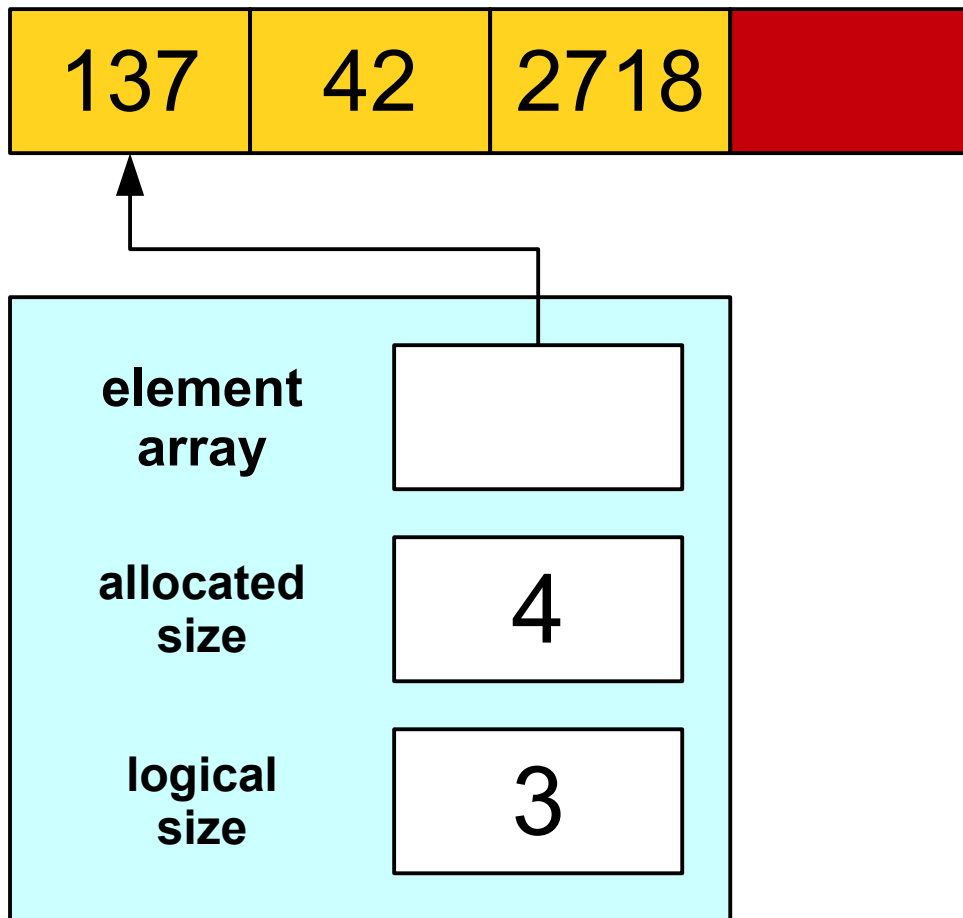
# A Bounded Stack



# A Bounded Stack

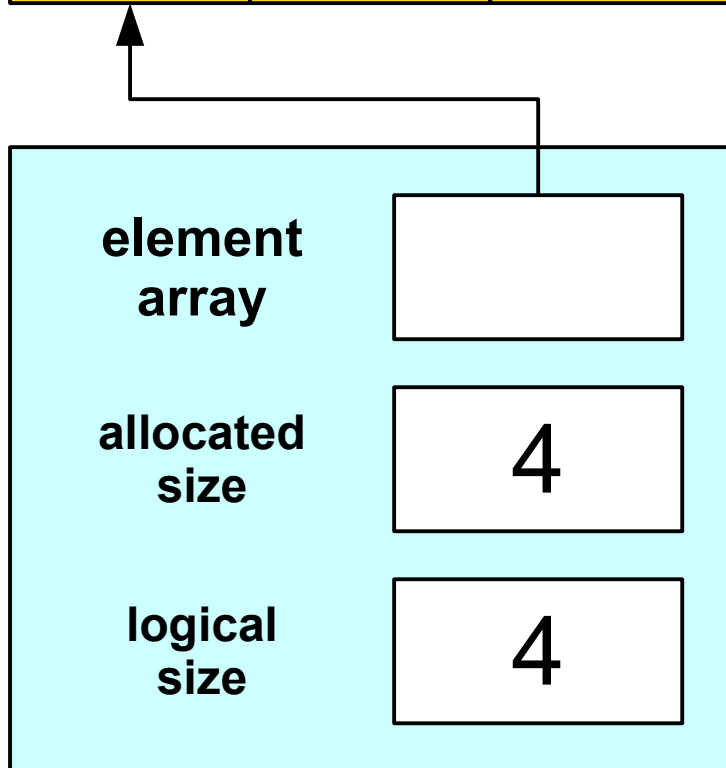


# A Bounded Stack

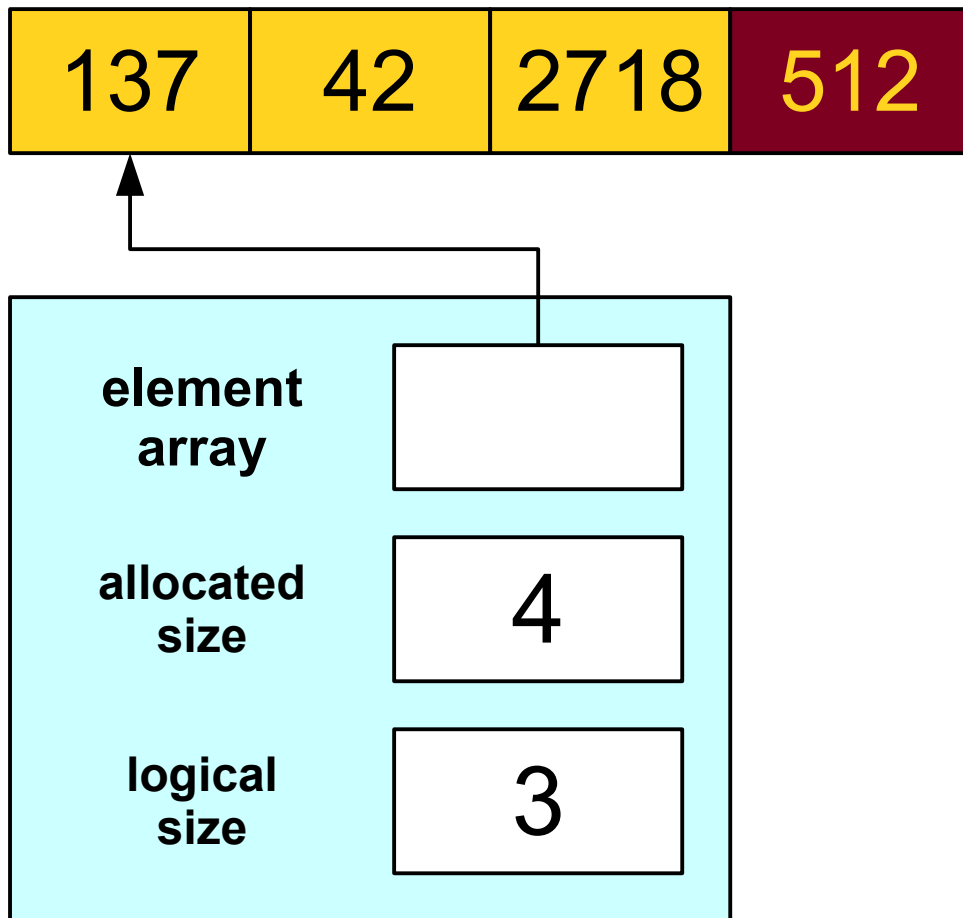




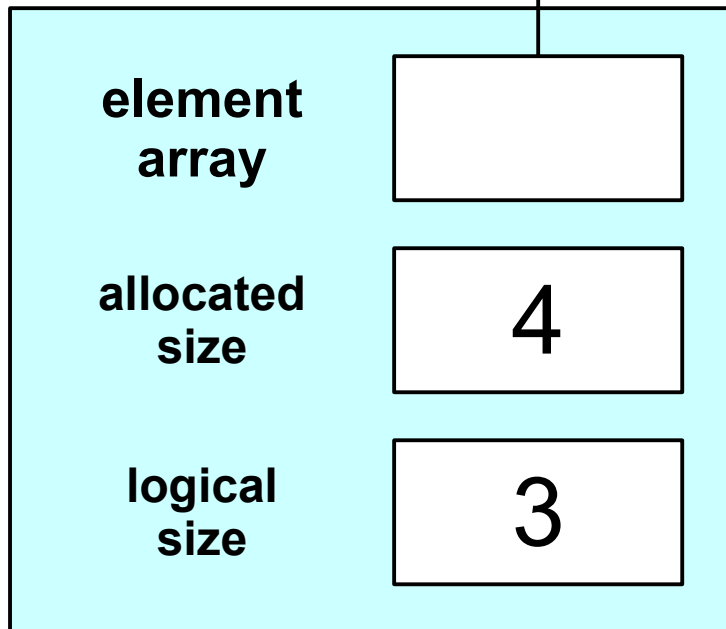
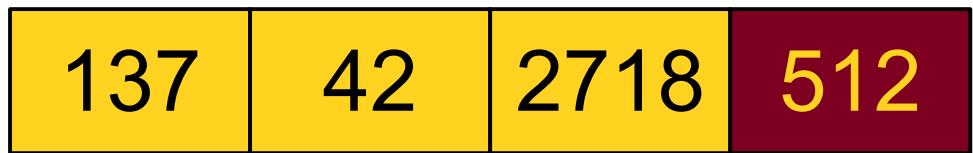
# A Bounded Stack



# A Bounded Stack

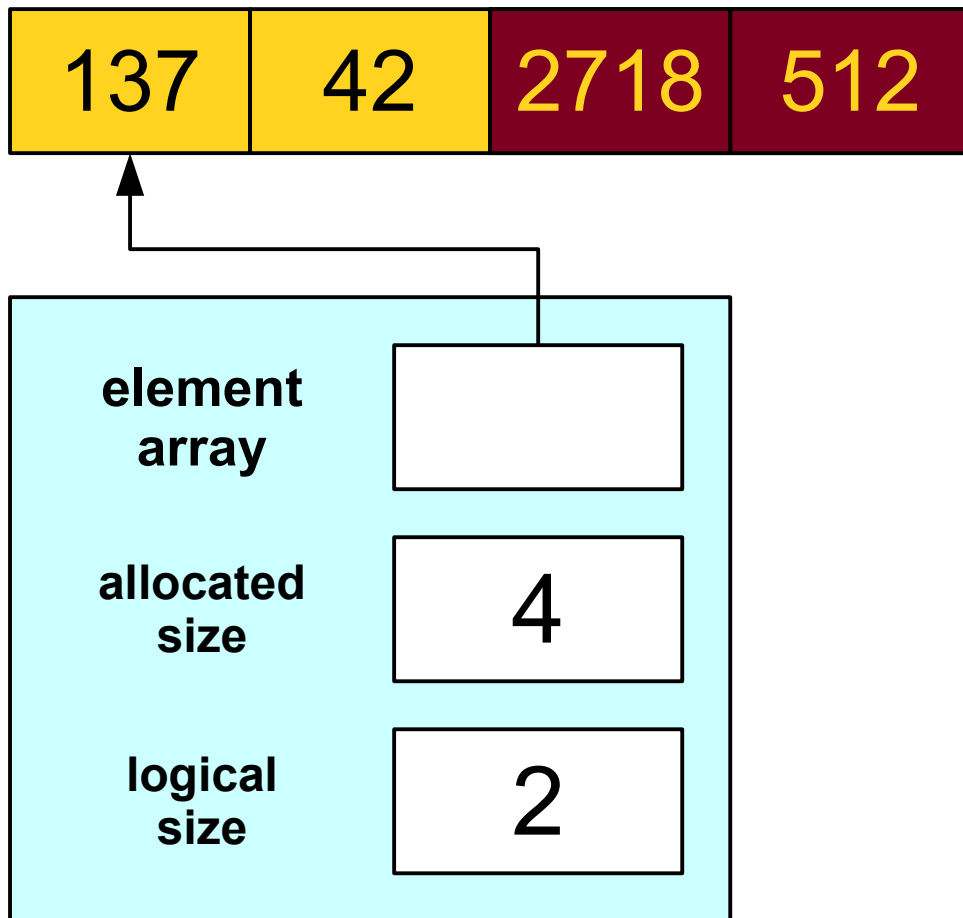


# A Bounded Stack

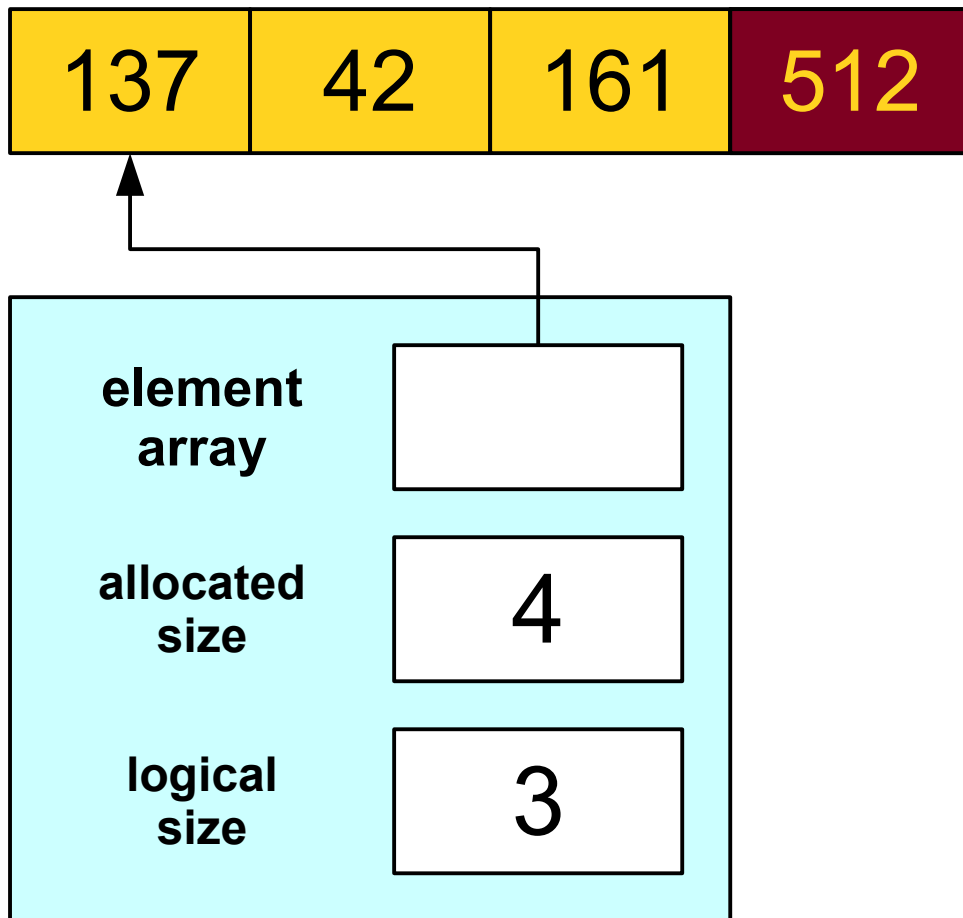


Arrays cannot grow or shrink, so this older value is still technically there in the array. We're just going to pretend it isn't.

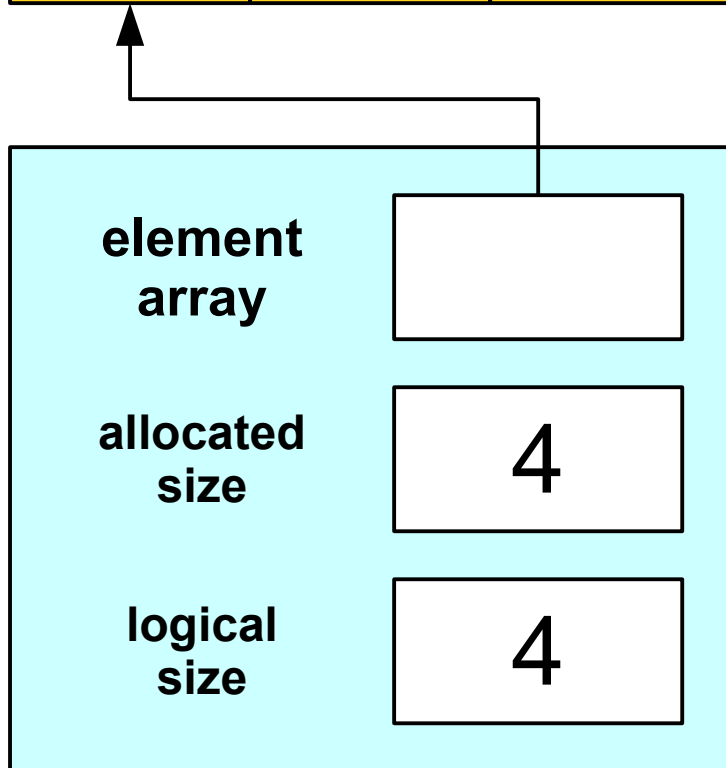
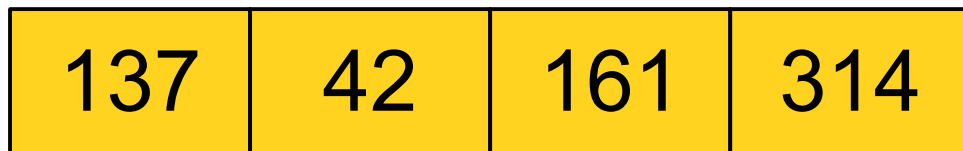
# A Bounded Stack



# A Bounded Stack



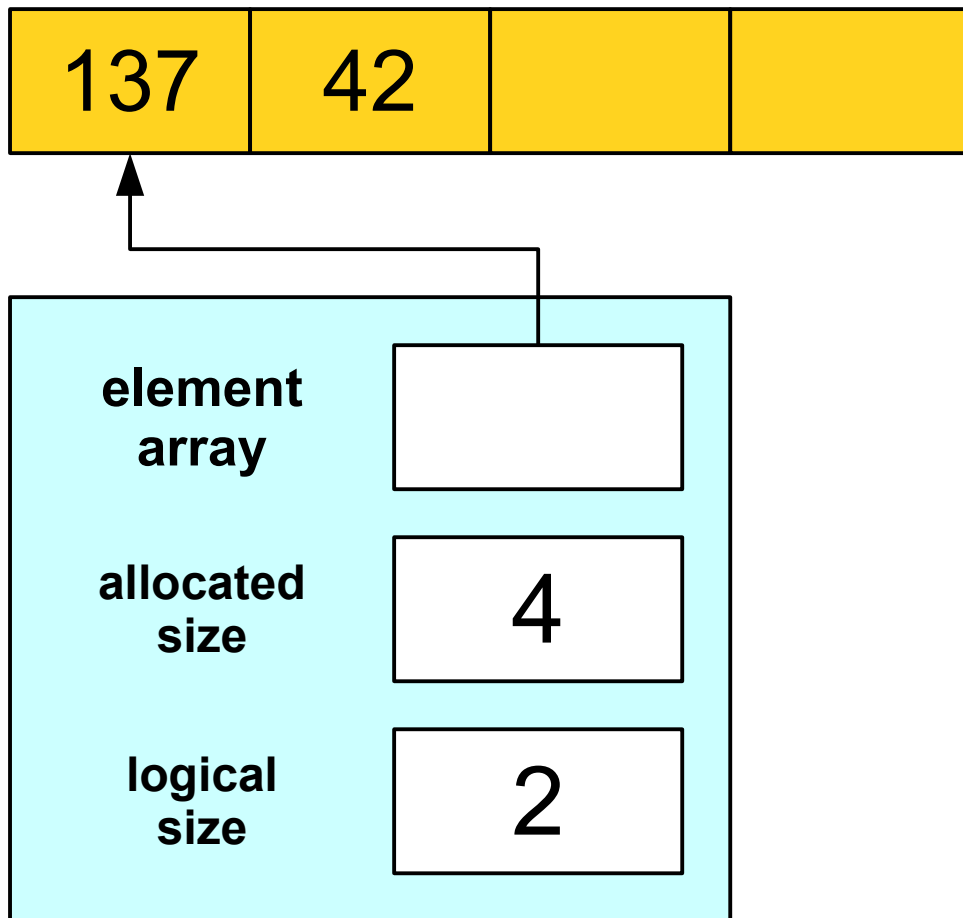
# A Bounded Stack



# Running out of Space

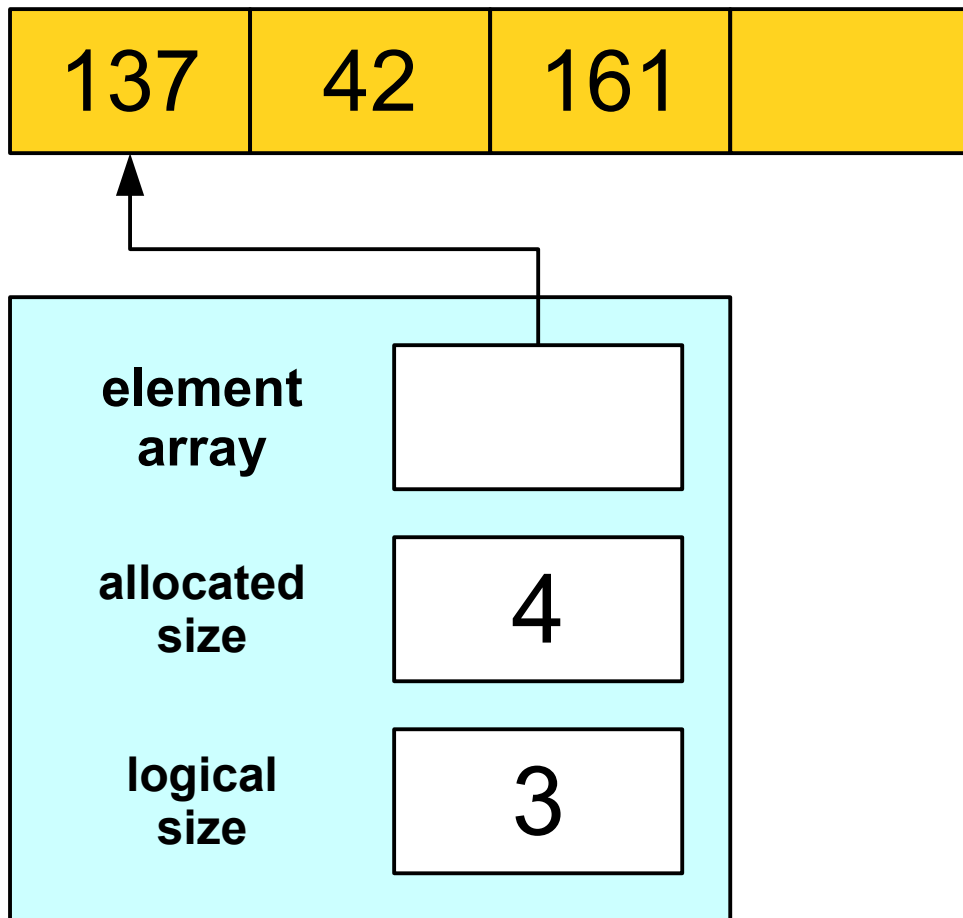
- Our current implementation very quickly runs out of space to store elements.
- What should we do when this happens?

# An Initial Idea

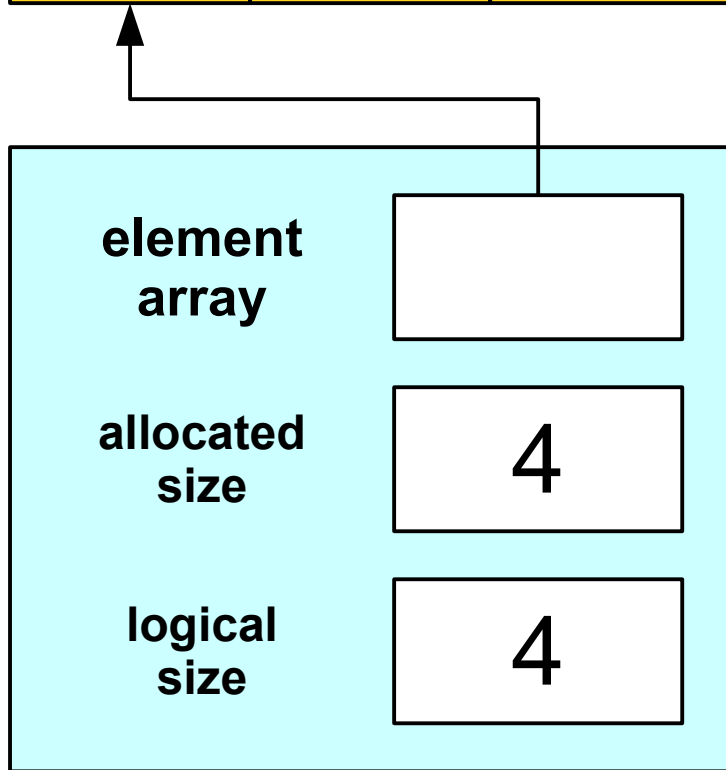
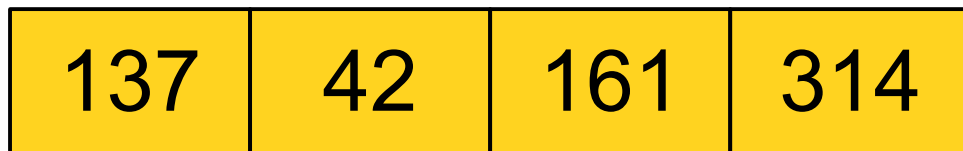




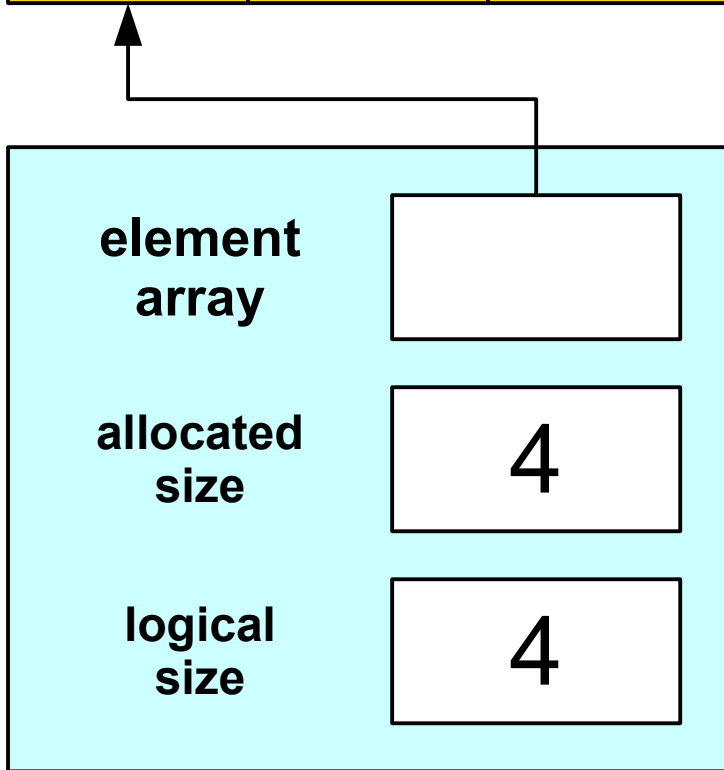
# An Initial Idea



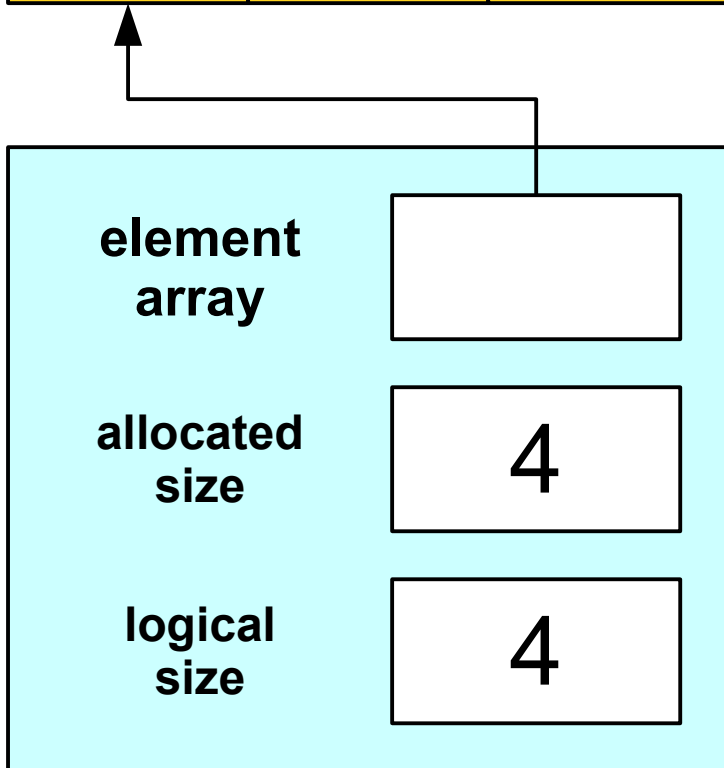
# An Initial Idea



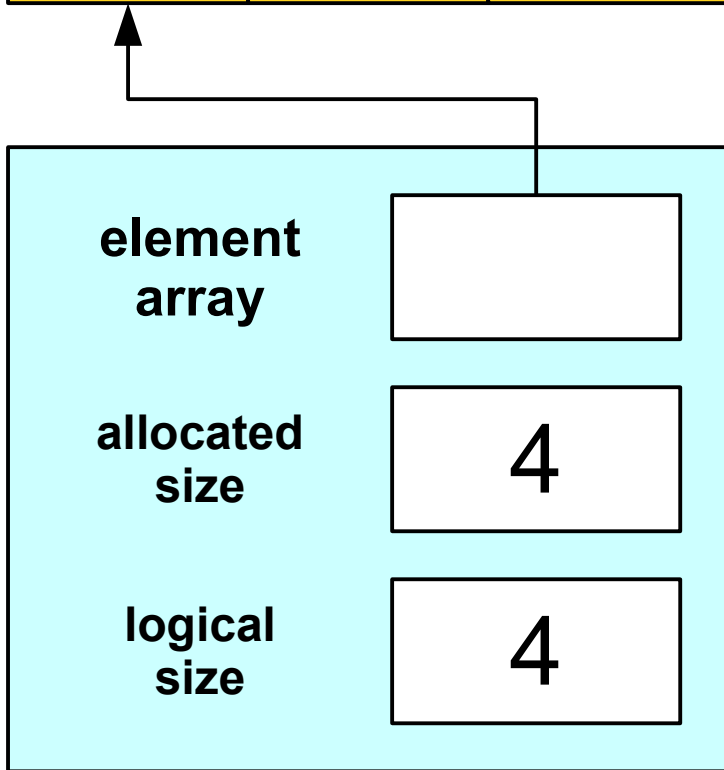
# An Initial Idea



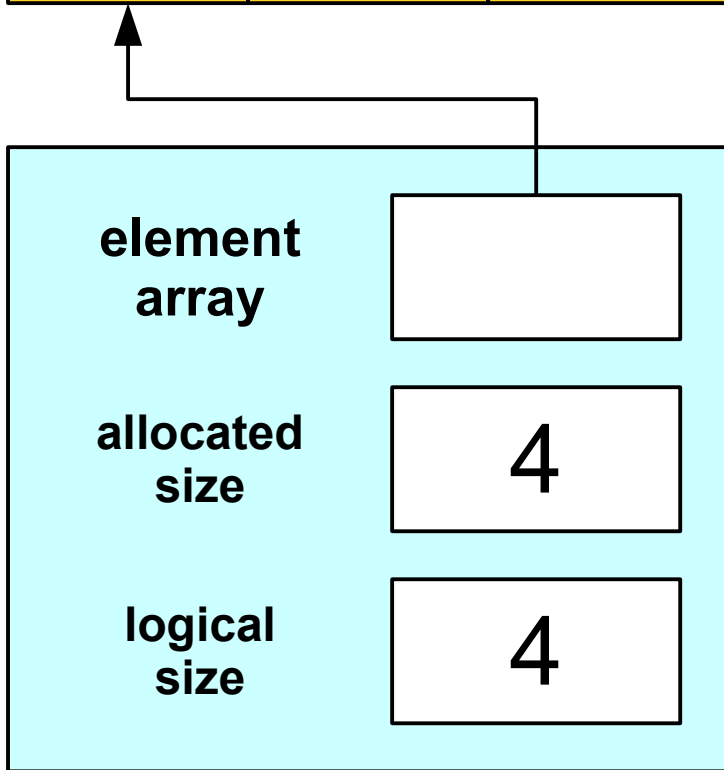
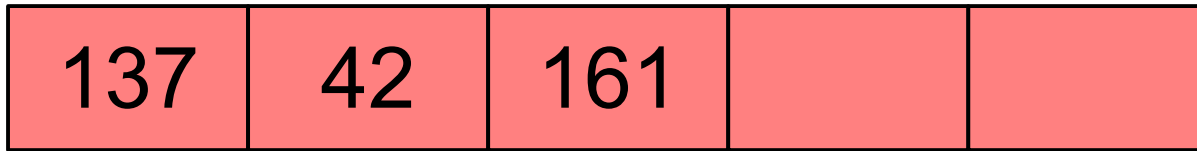
# An Initial Idea



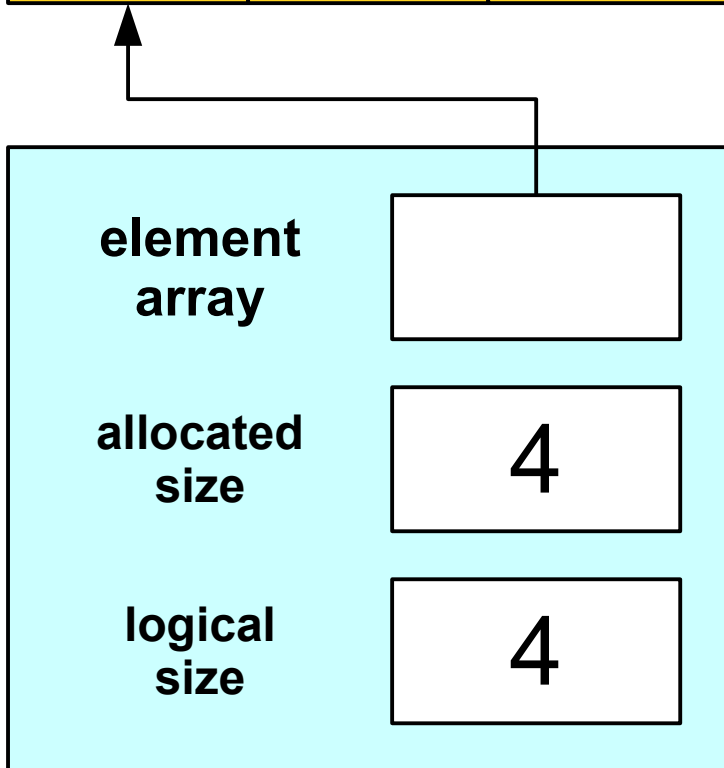
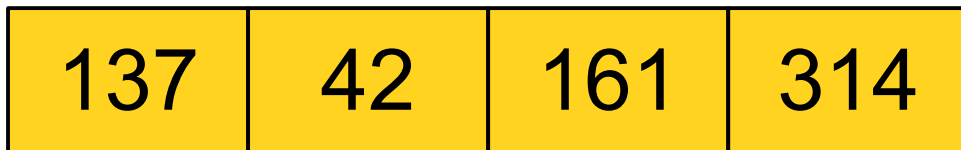
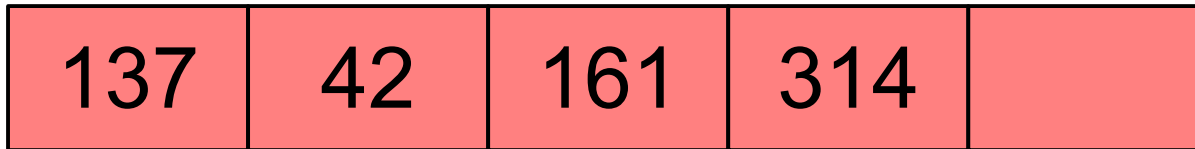
# An Initial Idea



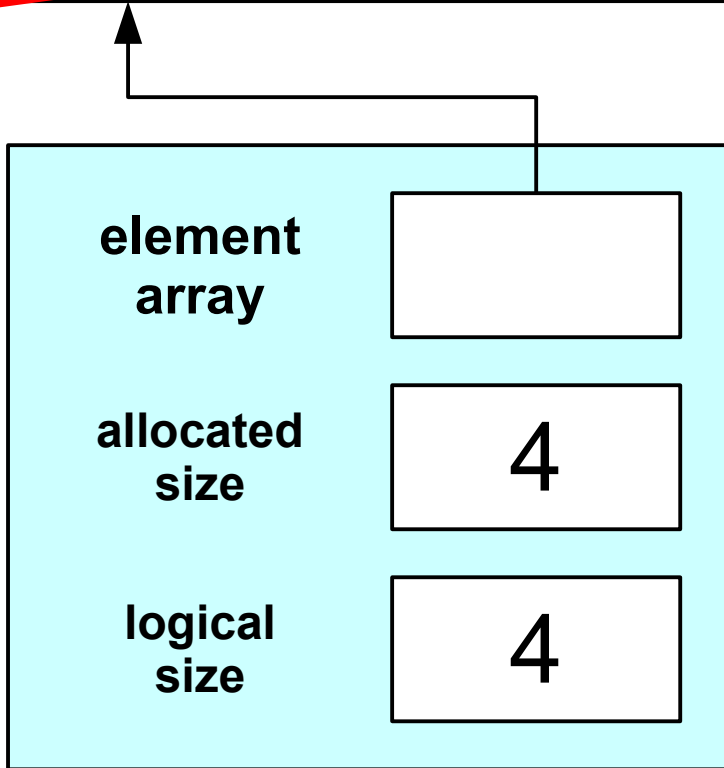
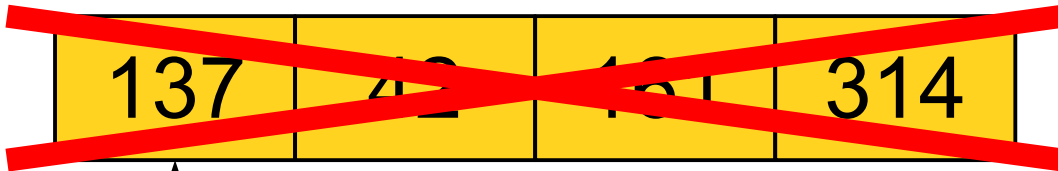
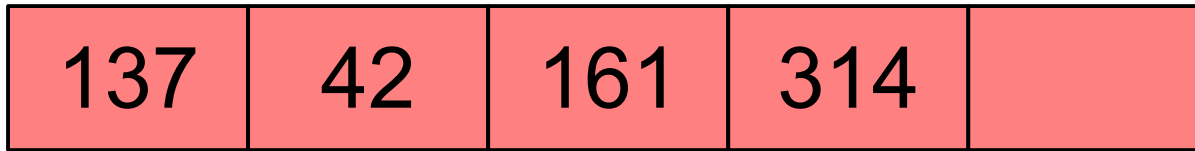
# An Initial Idea



# An Initial Idea

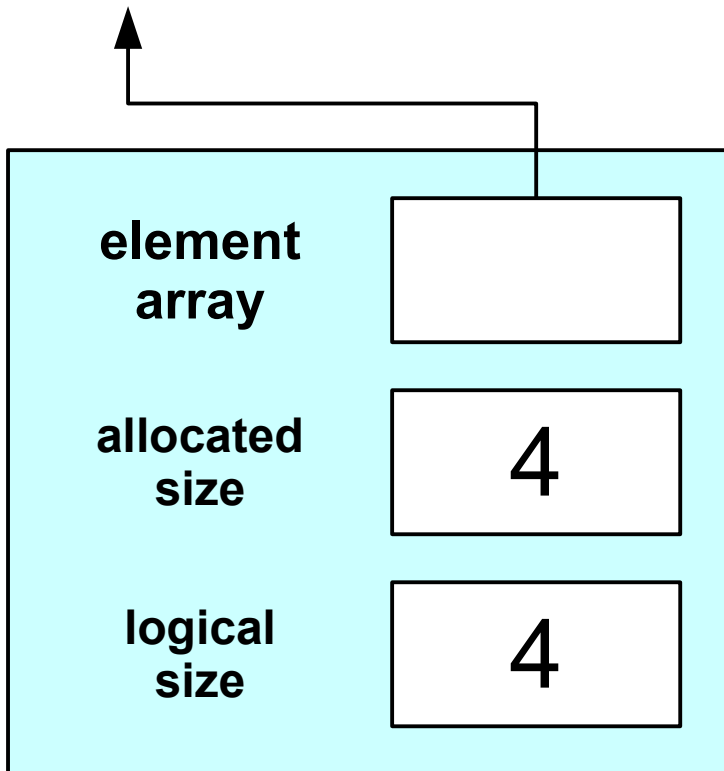
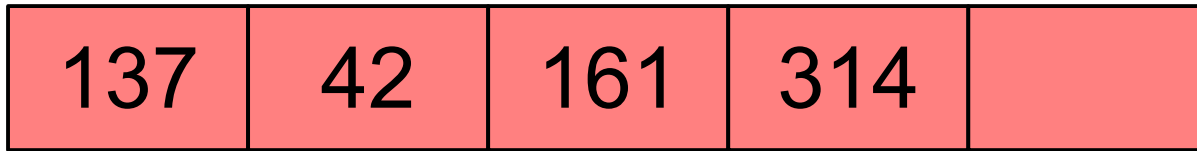


# An Initial Idea

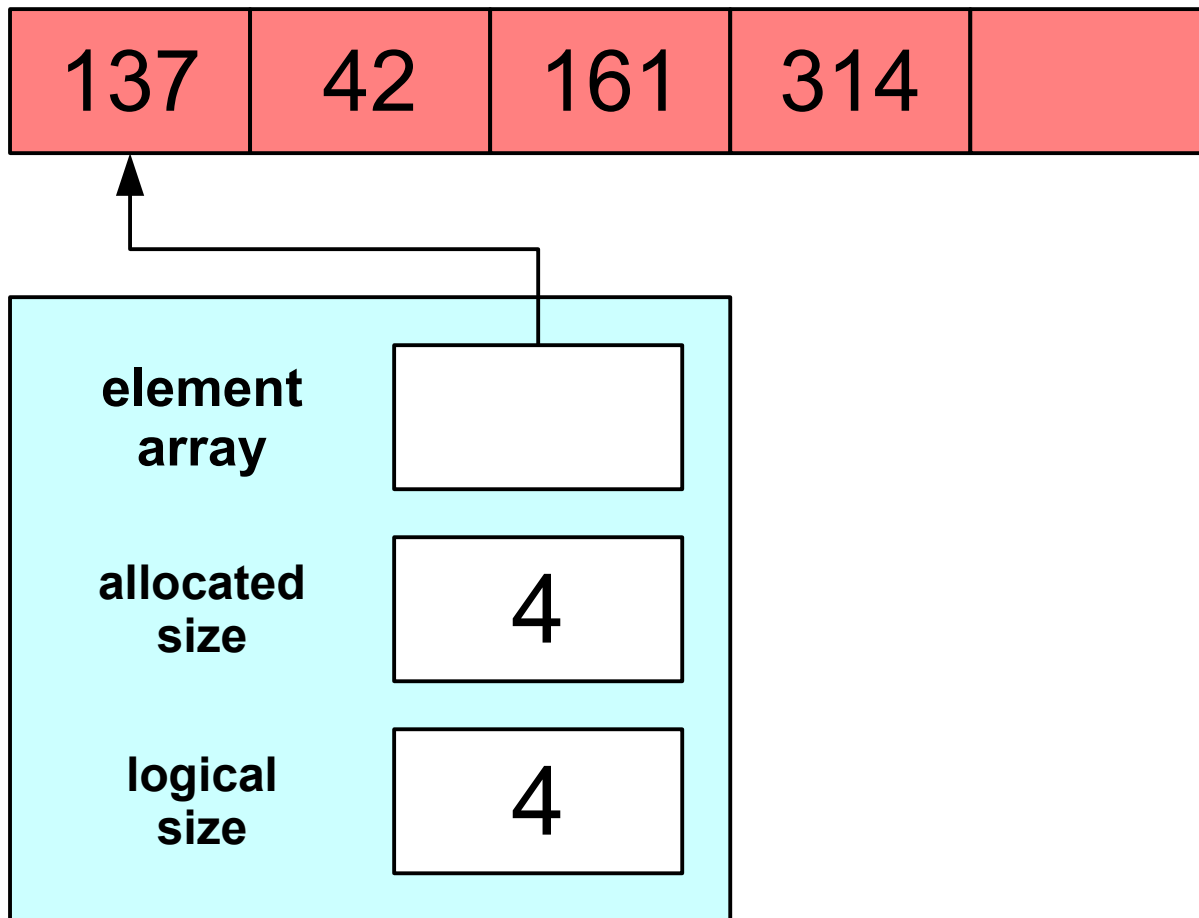




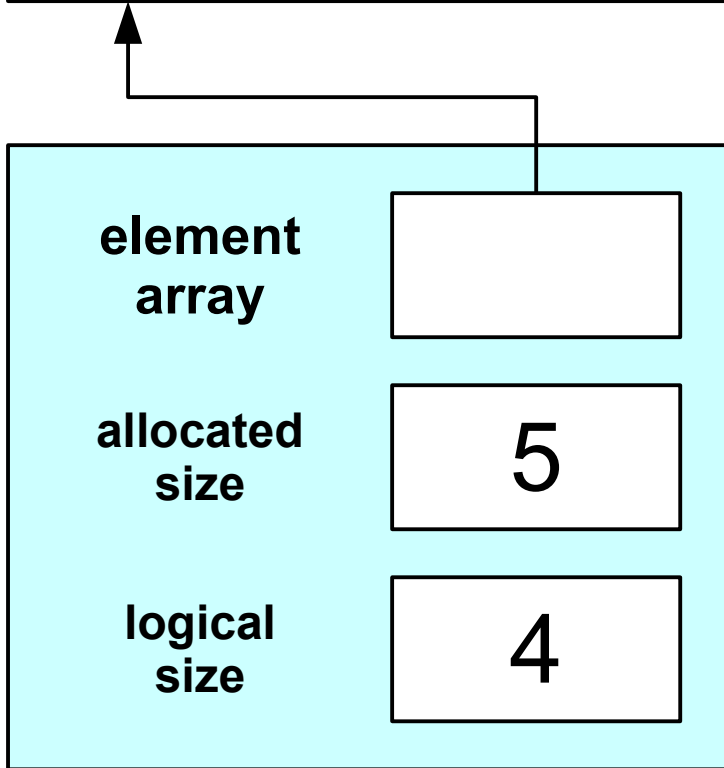
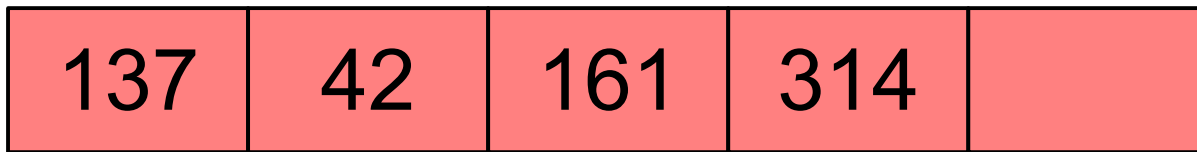
# An Initial Idea



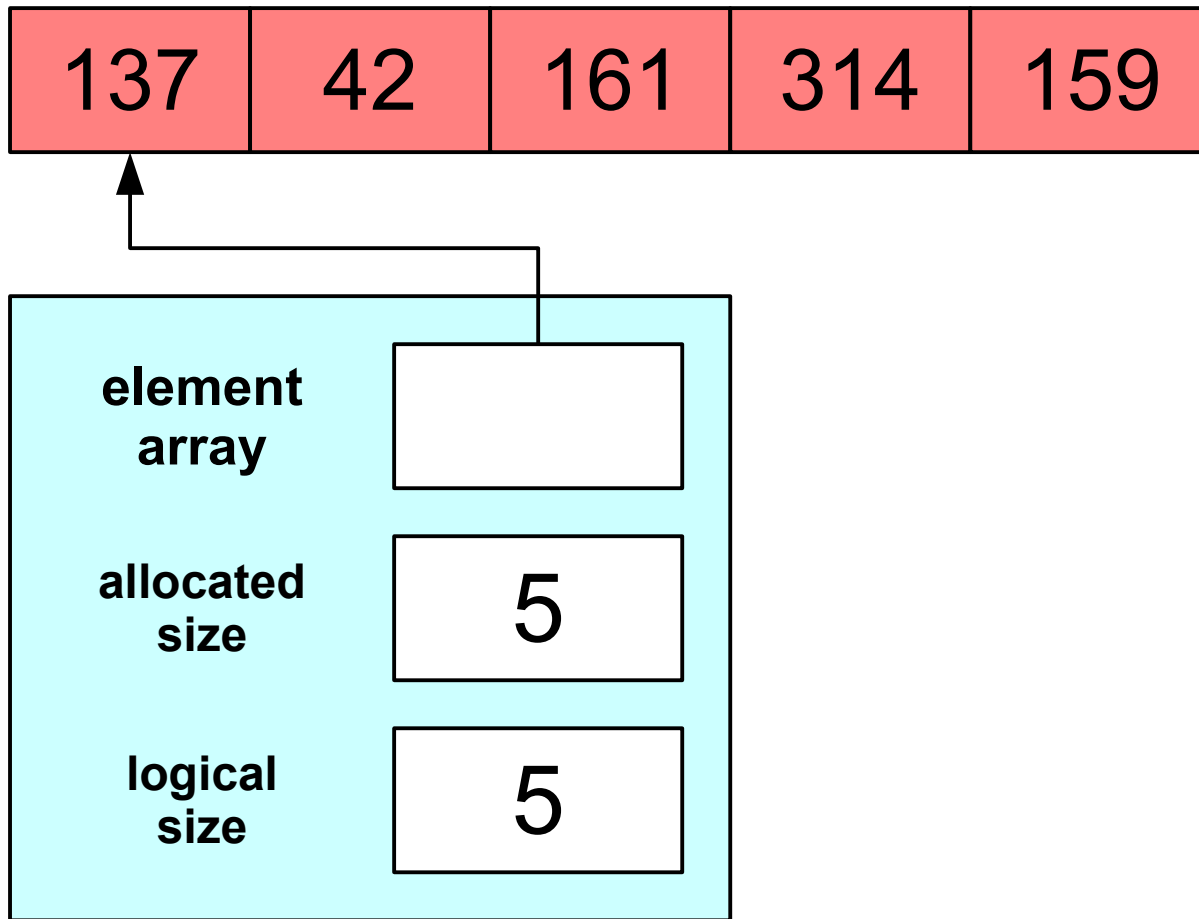
# An Initial Idea



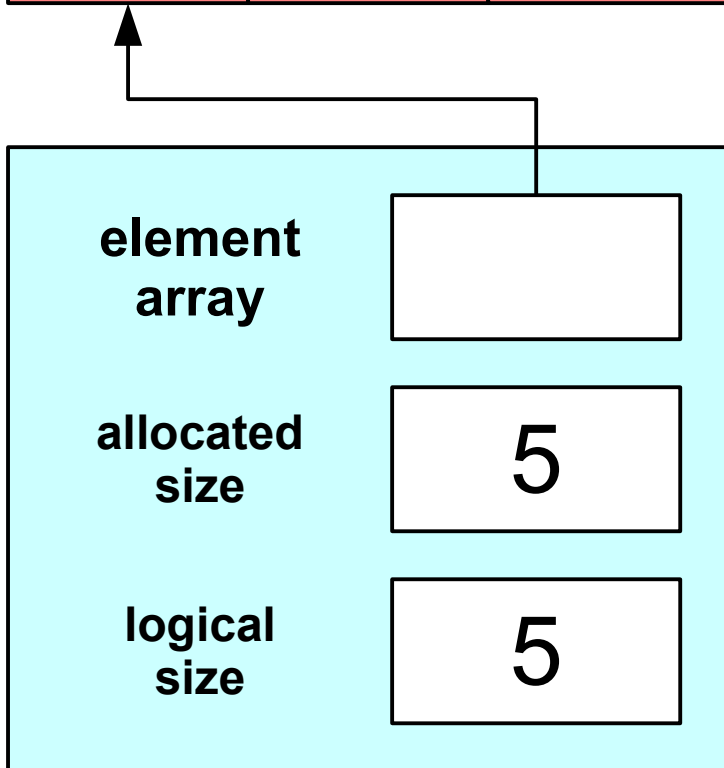
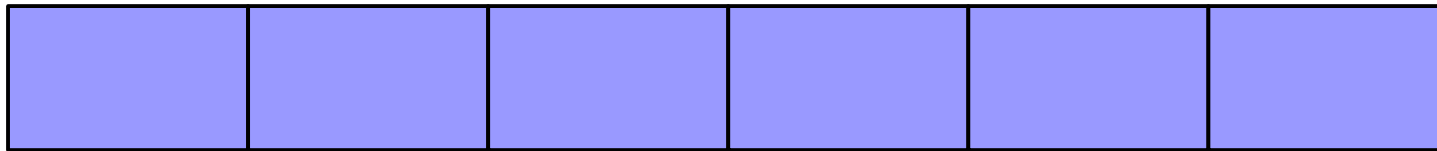
# An Initial Idea



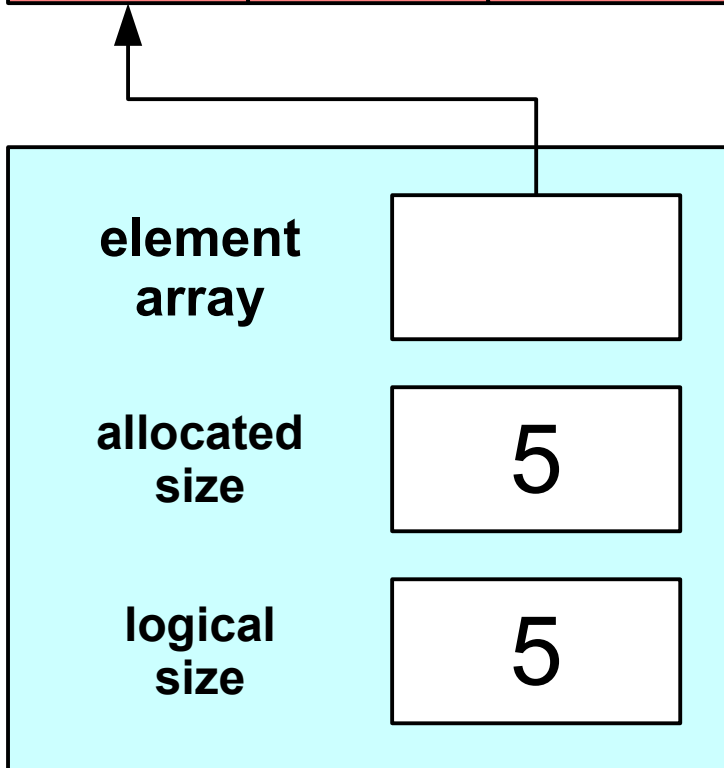
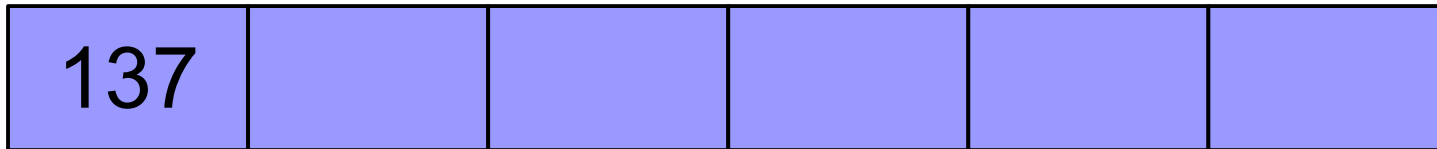
# An Initial Idea



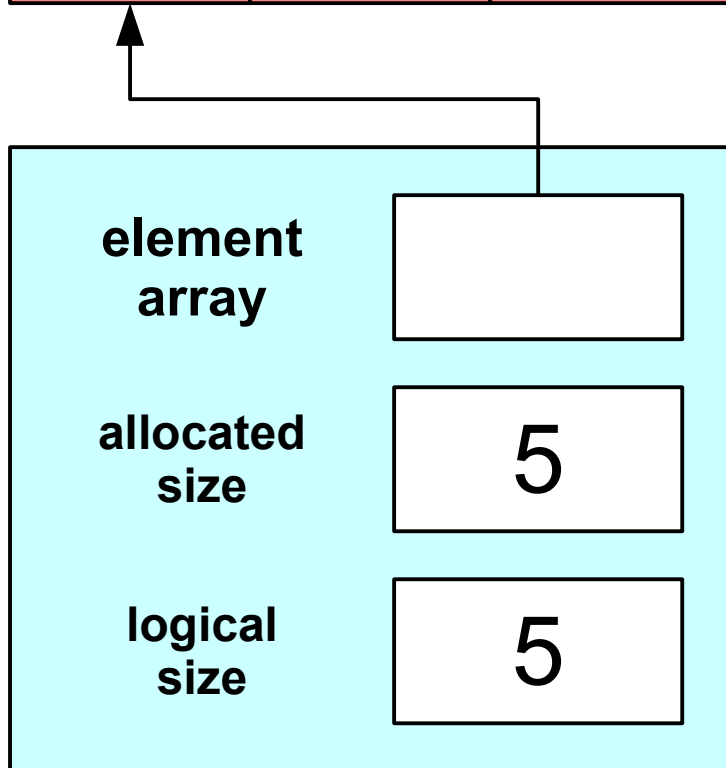
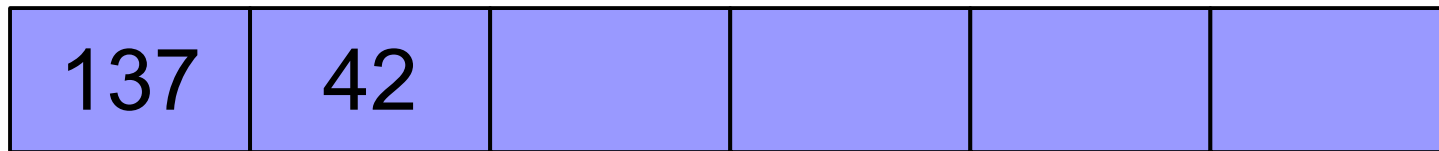
# An Initial Idea



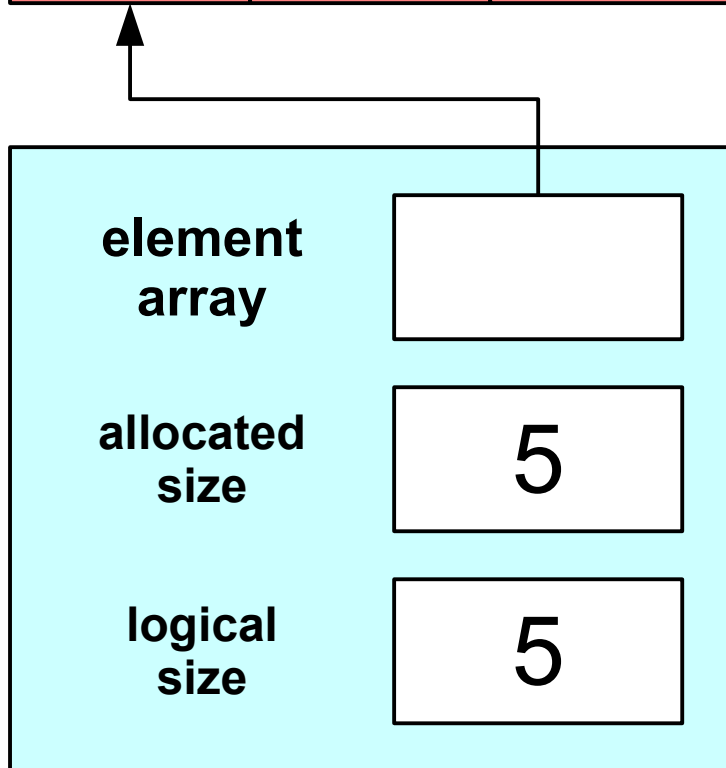
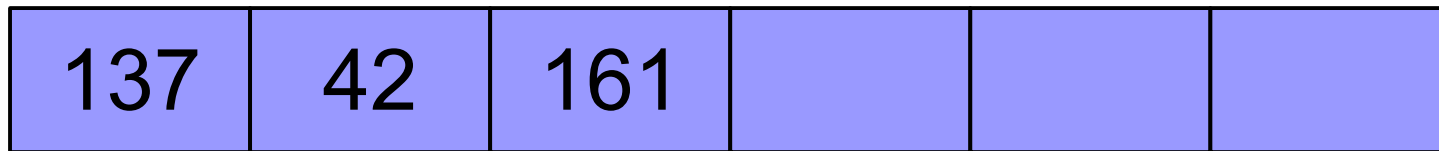
# An Initial Idea



# An Initial Idea

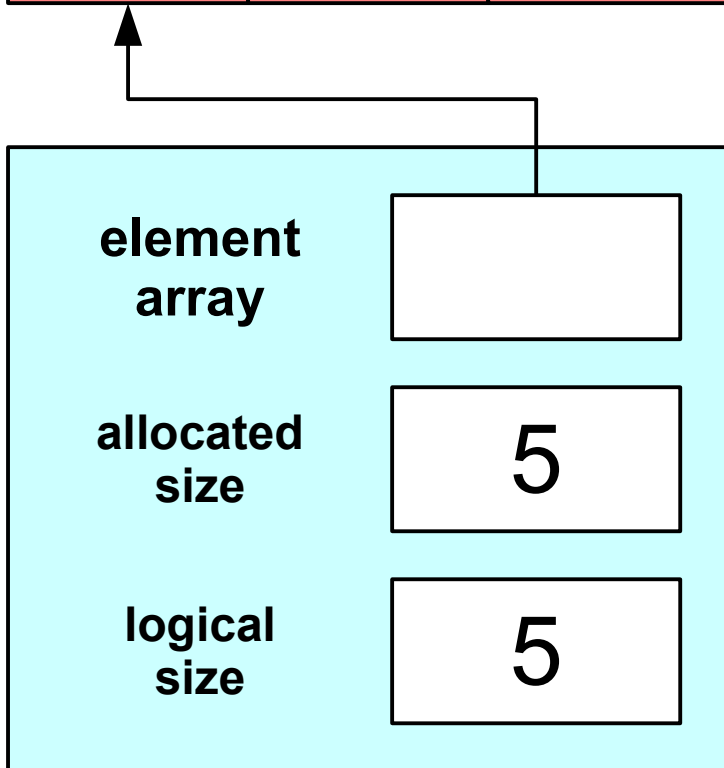
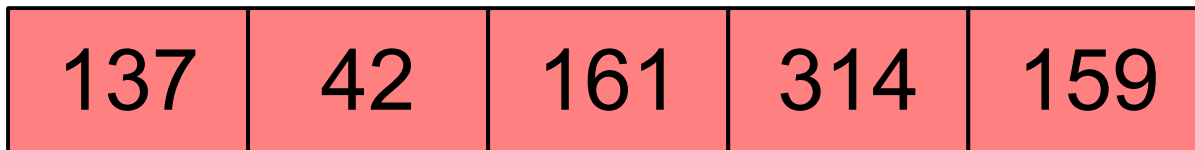
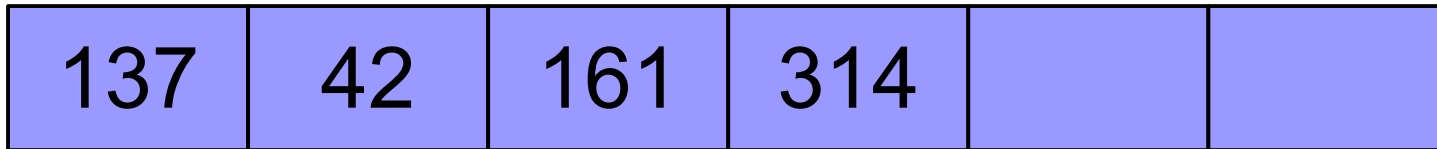


# An Initial Idea

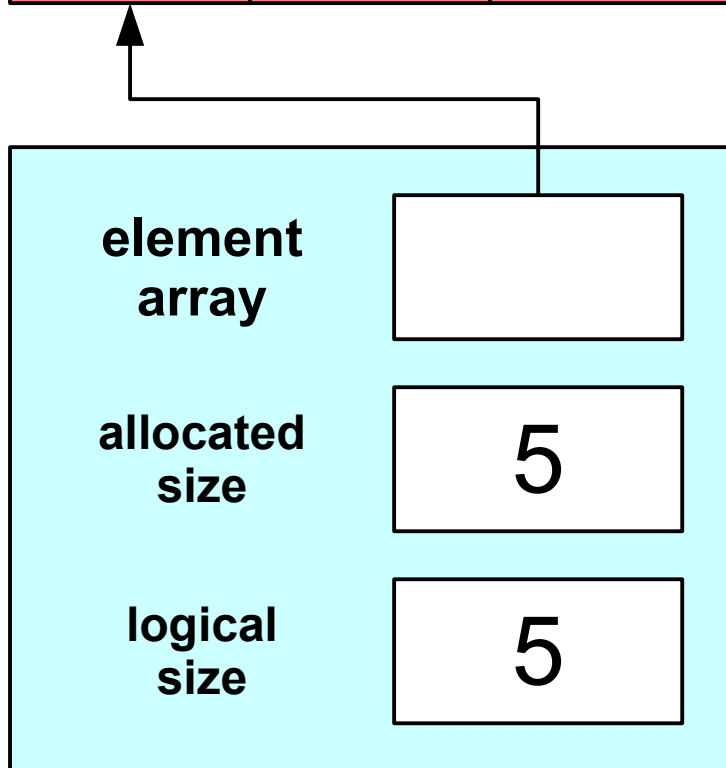
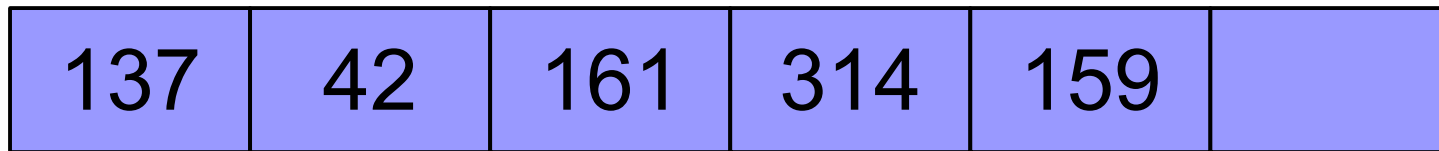




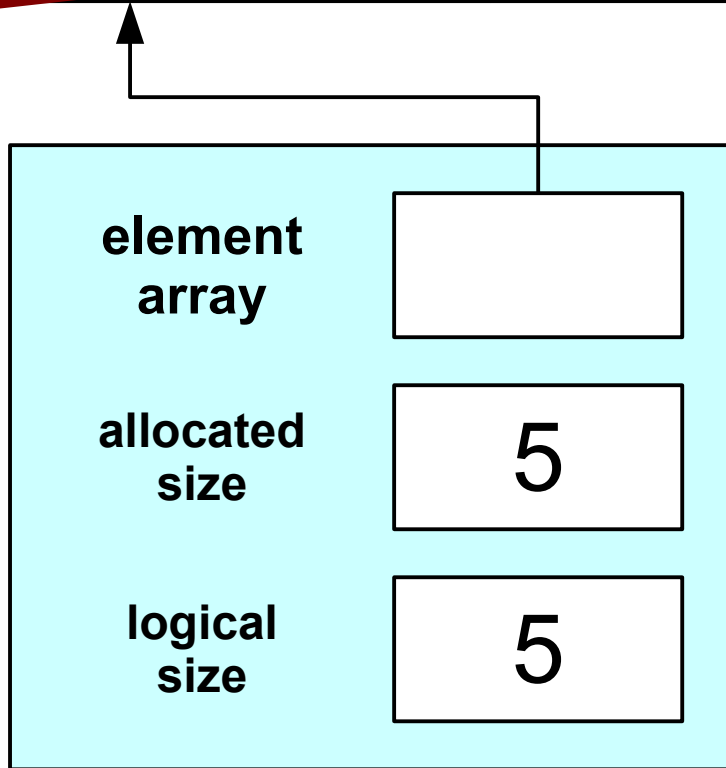
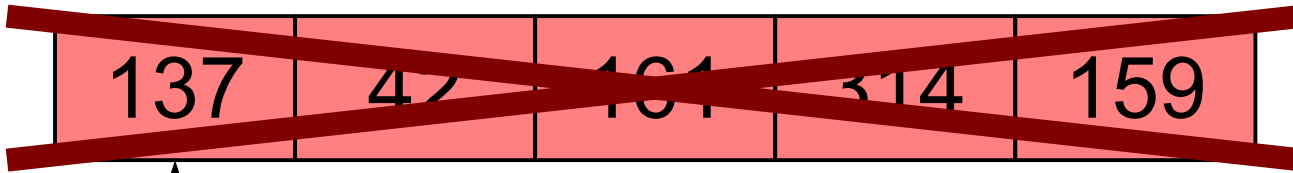
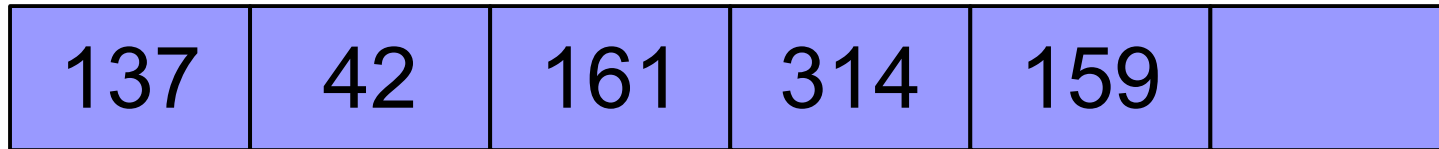
# An Initial Idea



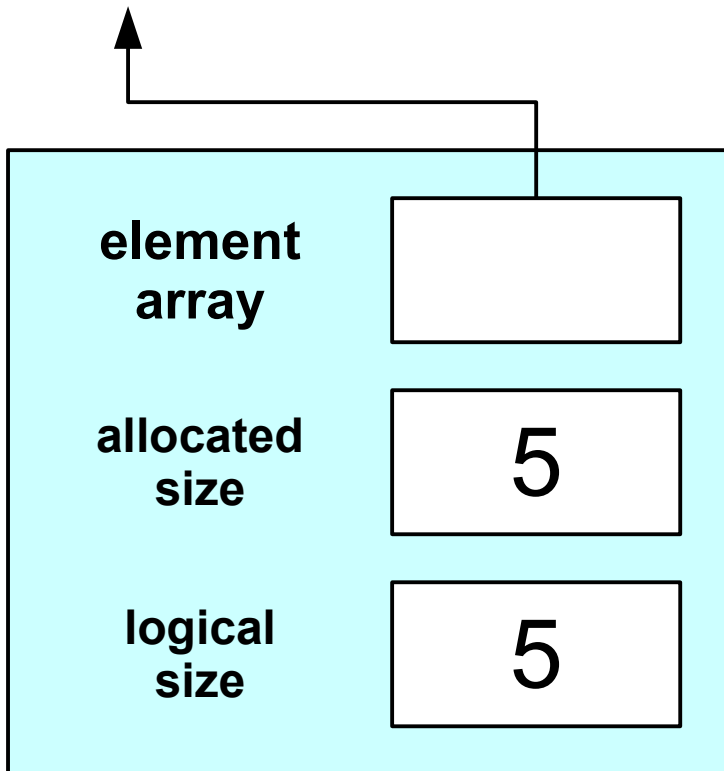
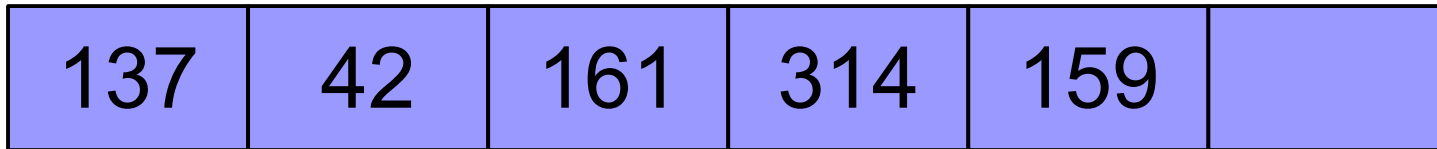
# An Initial Idea



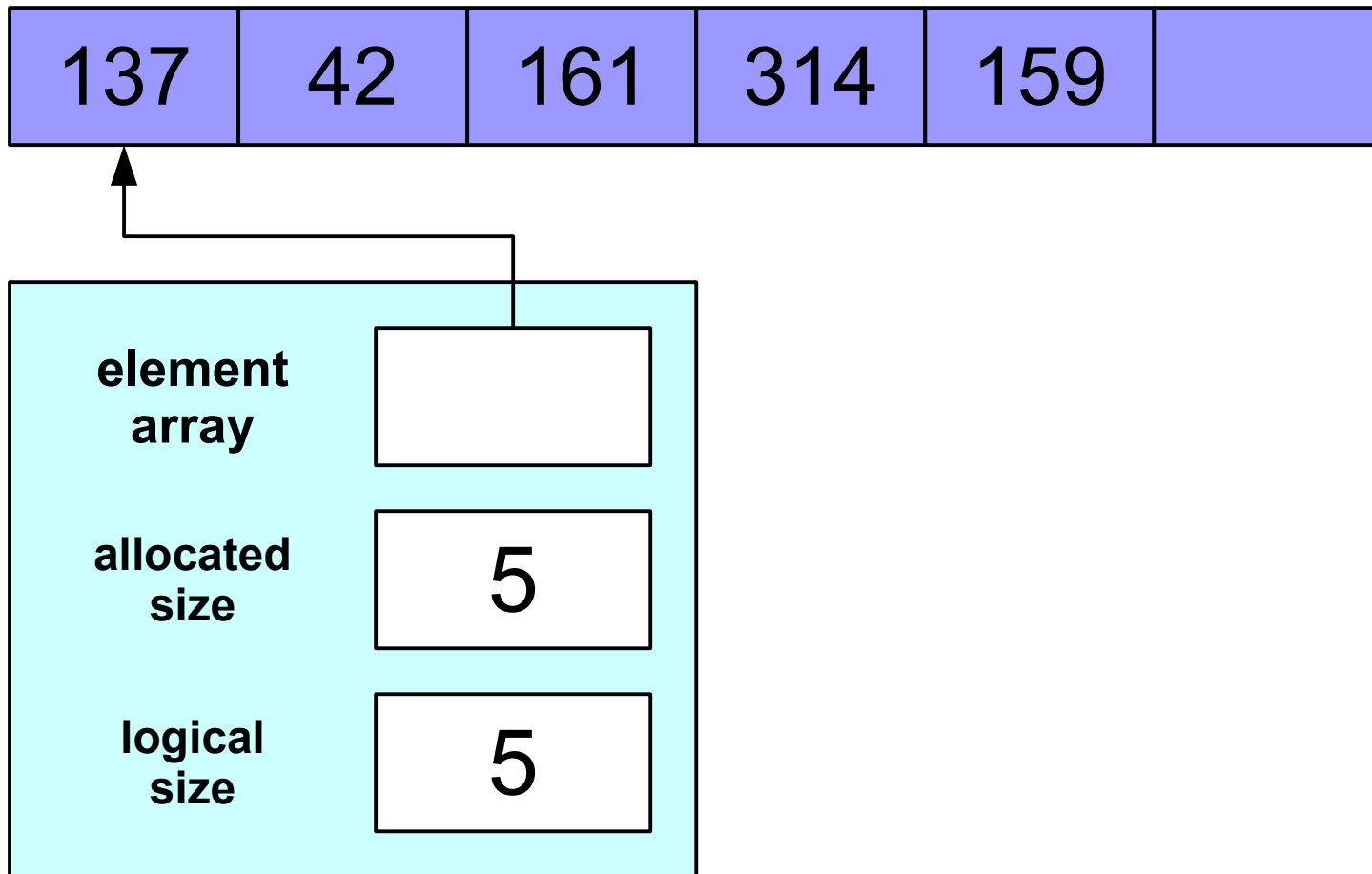
# An Initial Idea



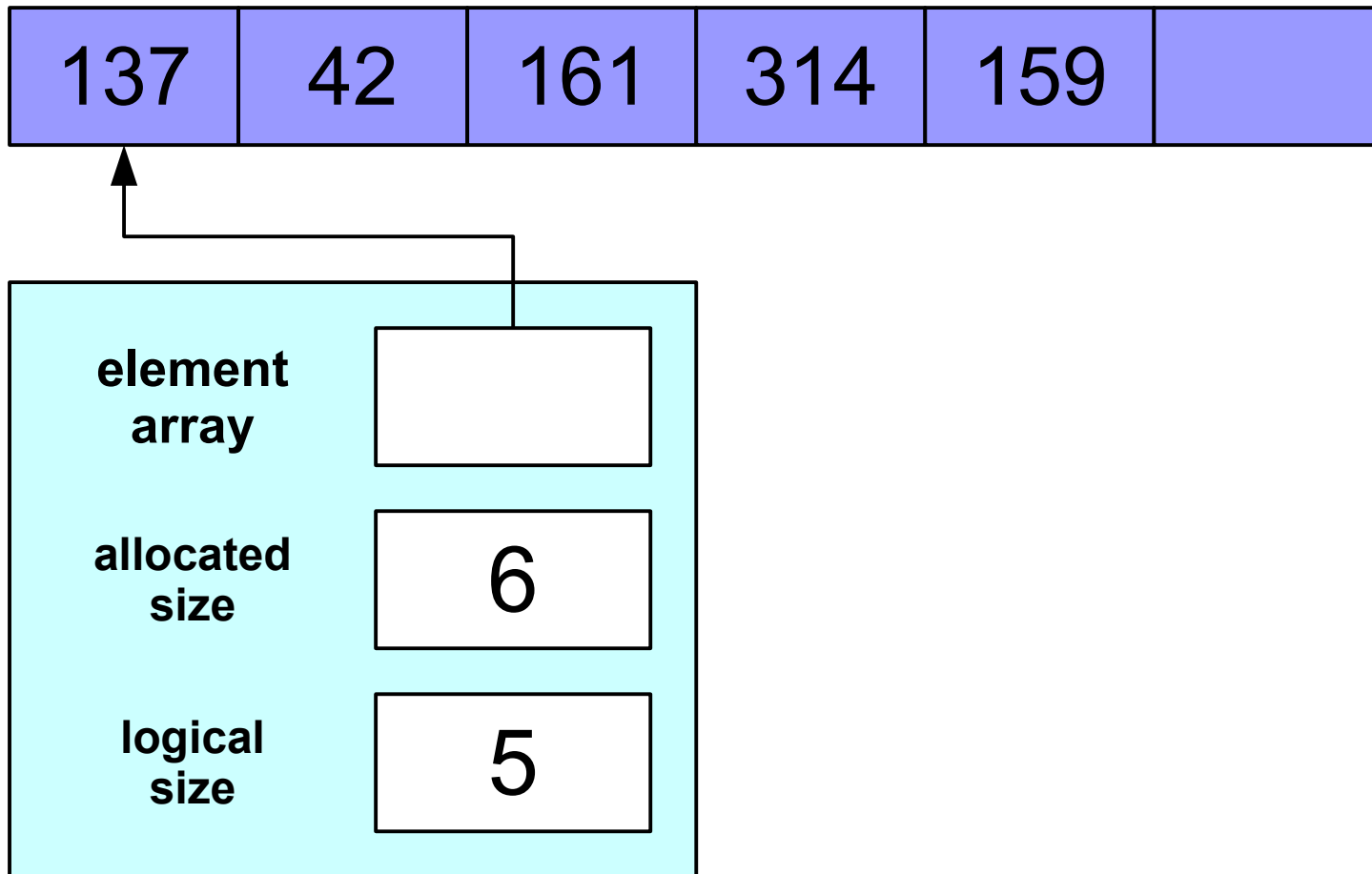
# An Initial Idea



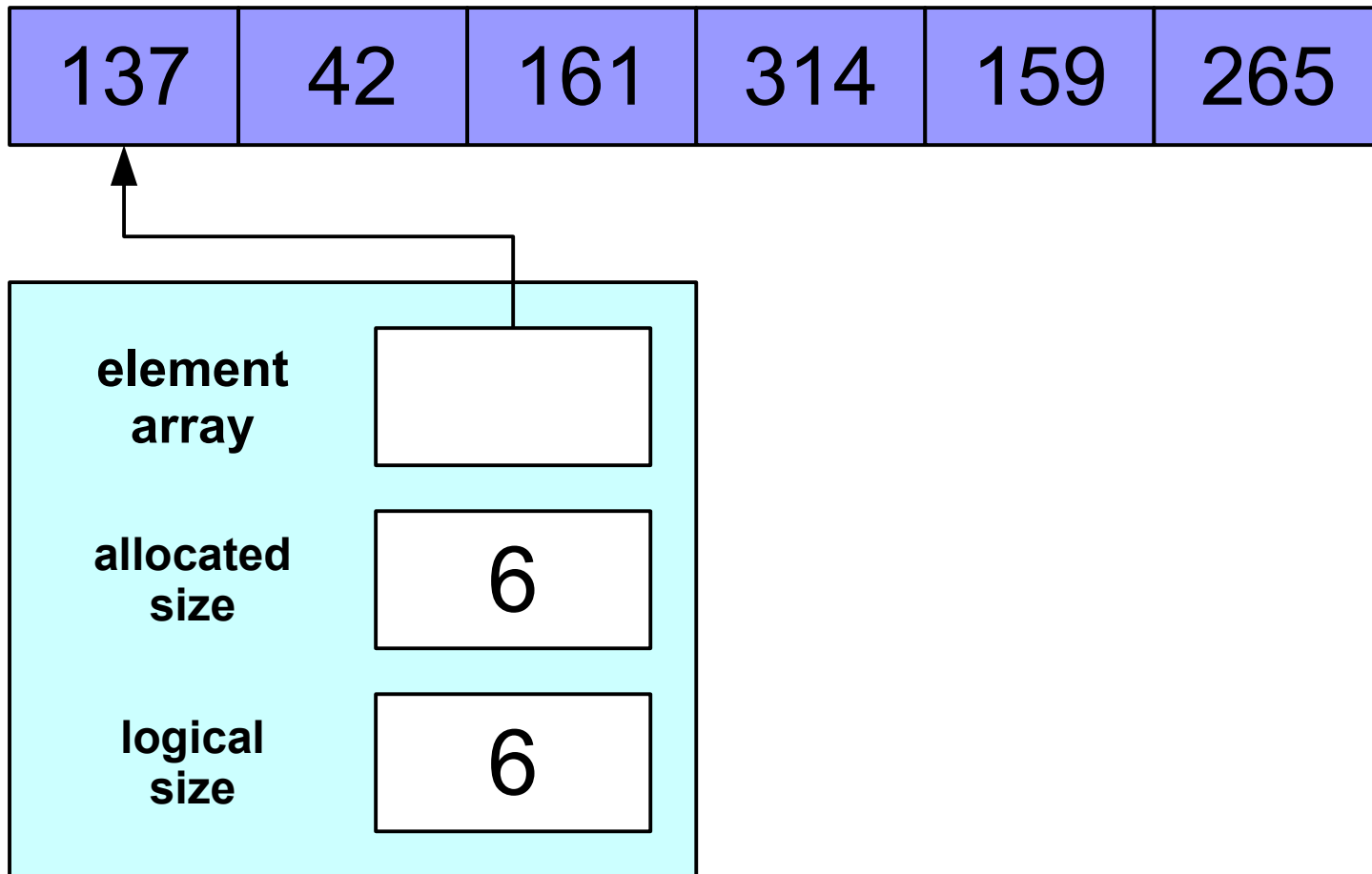
# An Initial Idea



# An Initial Idea



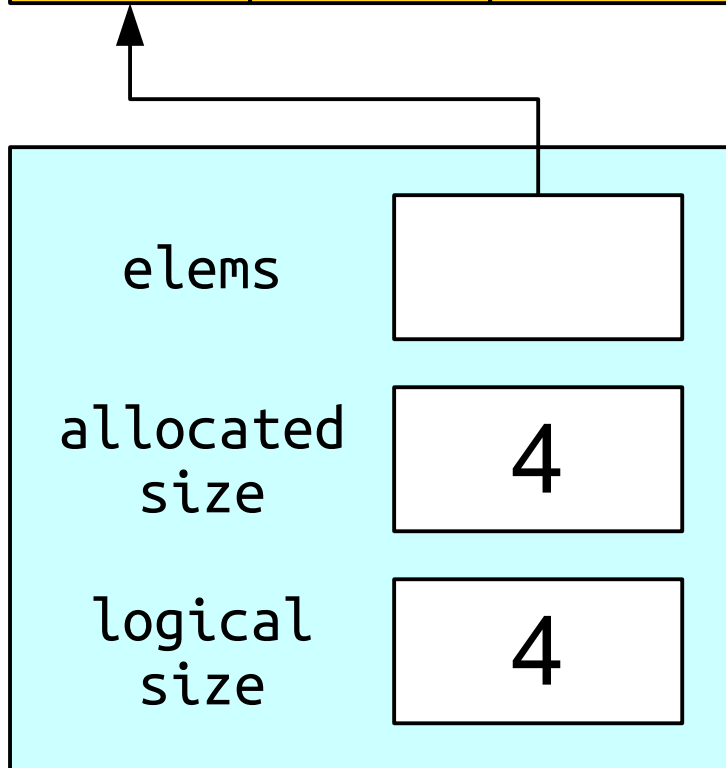
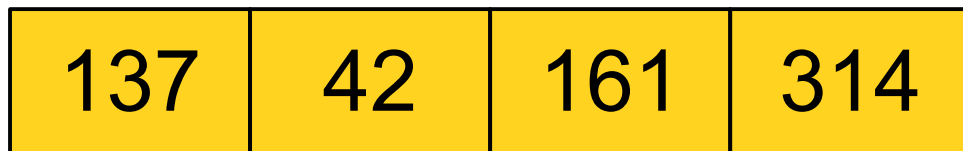
# An Initial Idea



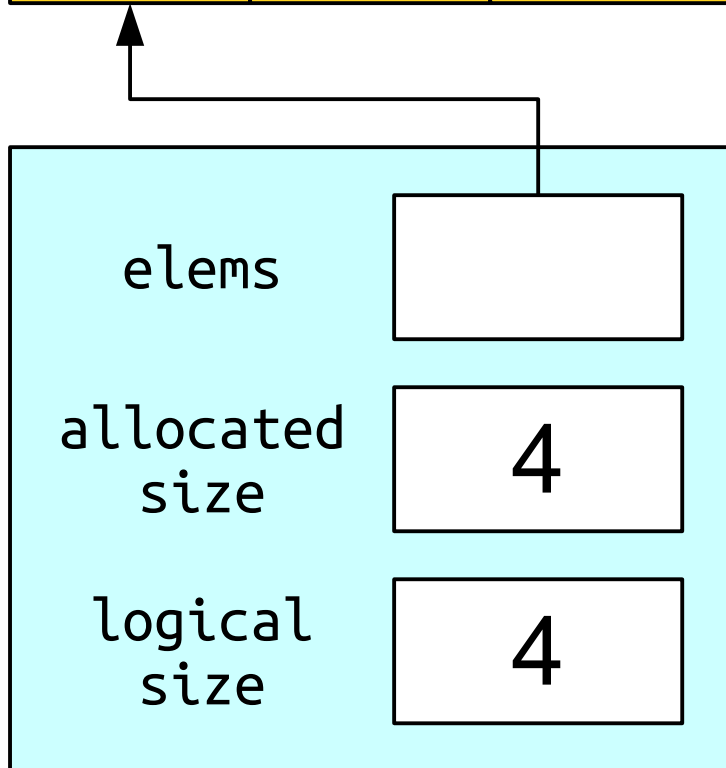
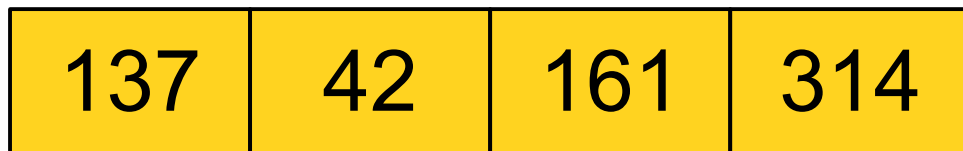
Ready... set... grow!



# An Initial Idea

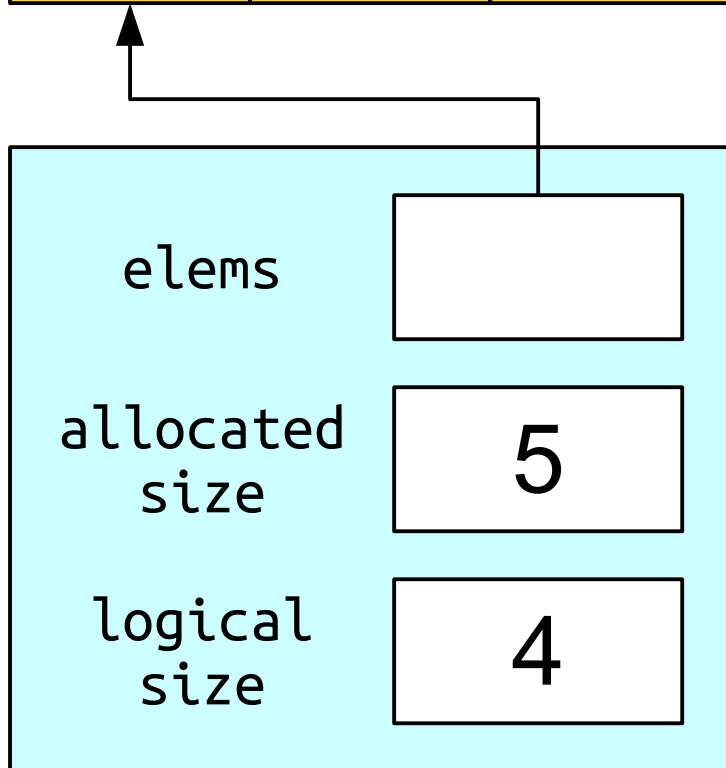
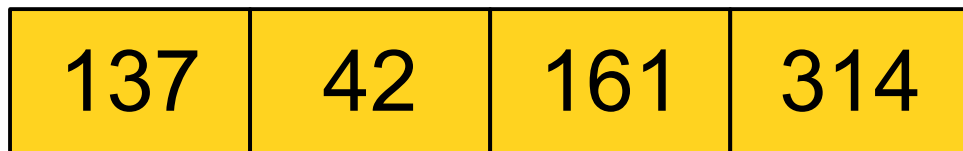


# An Initial Idea



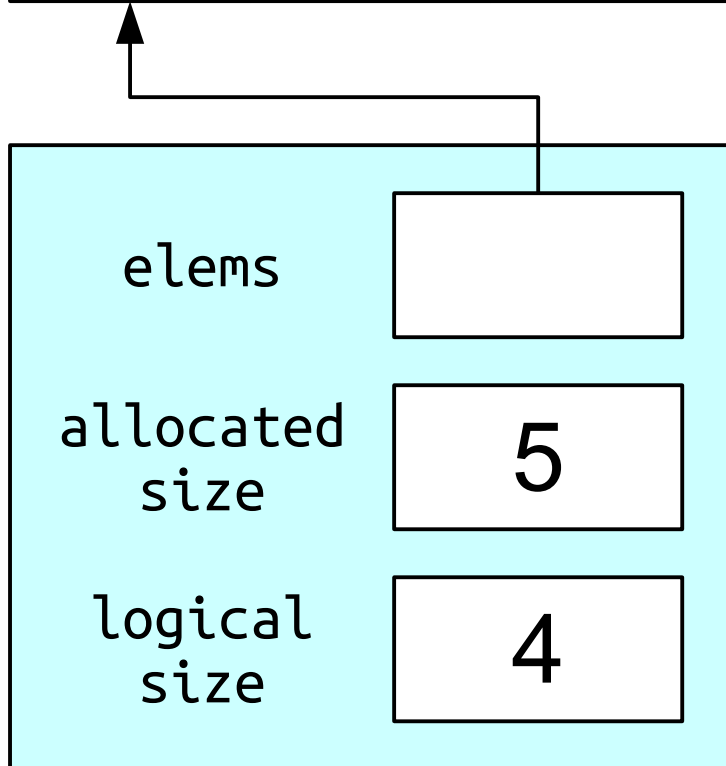
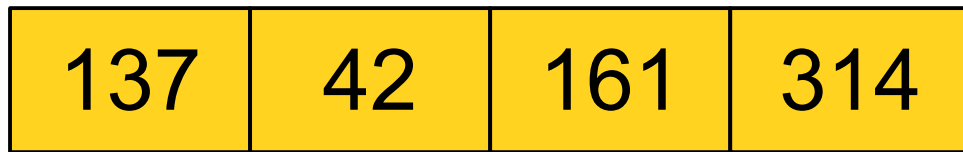
```
void OurStack::grow() {  
    allocatedSize++;  
  
}
```

# An Initial Idea



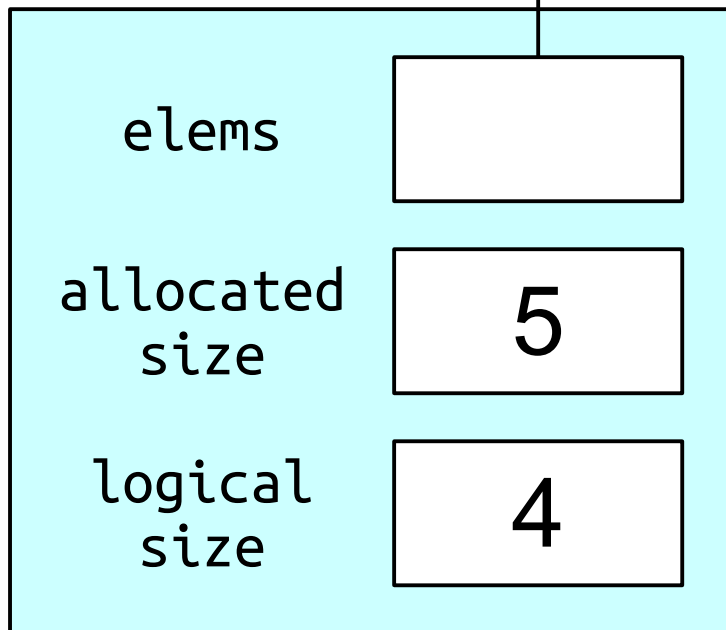
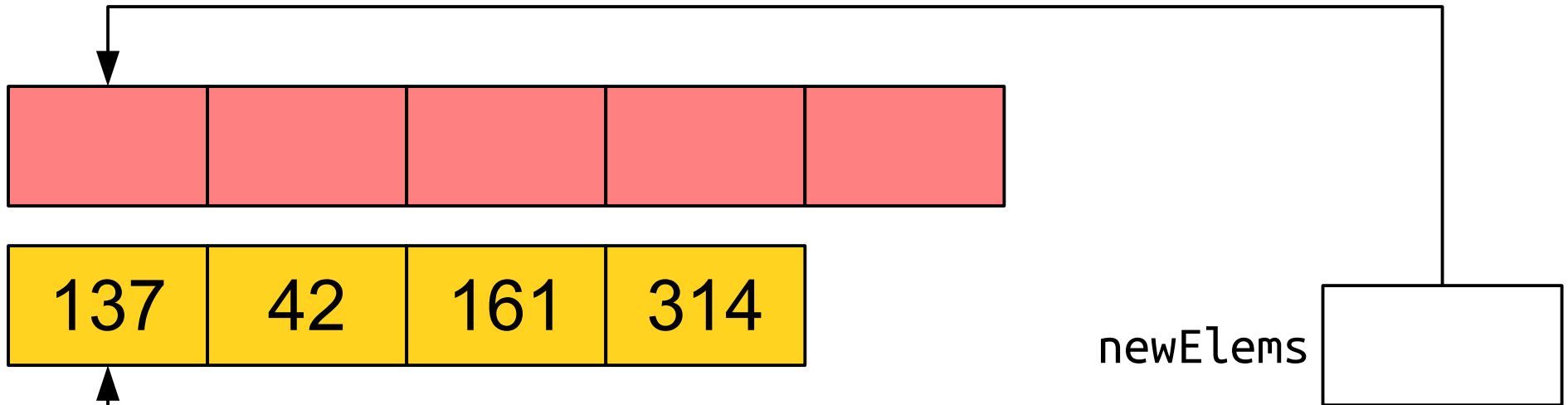
```
void OurStack::grow() {  
    allocatedSize++;  
  
}
```

# An Initial Idea



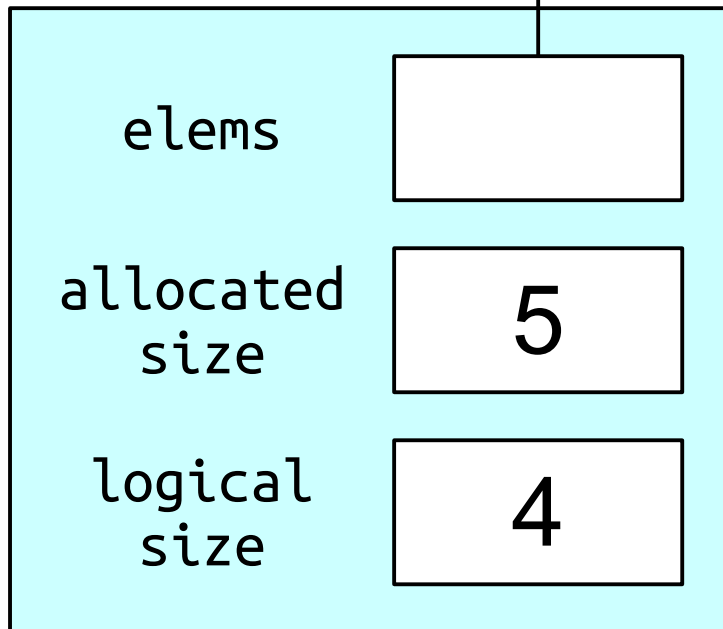
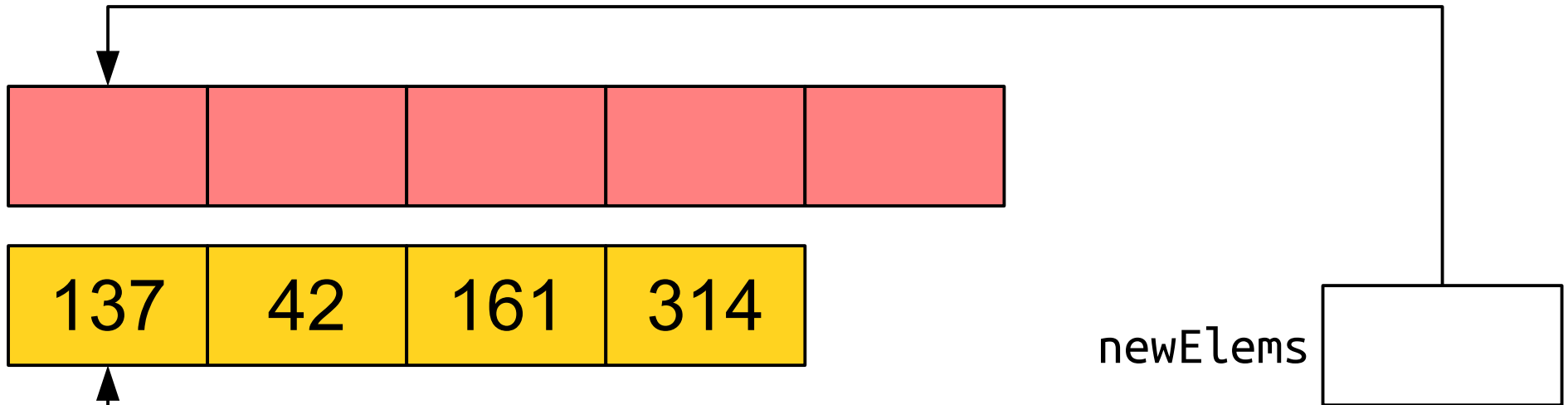
```
void OurStack::grow() {  
    allocatedSize++;  
  
    int* newElems = new int[allocatedSize];  
  
}
```

# An Initial Idea



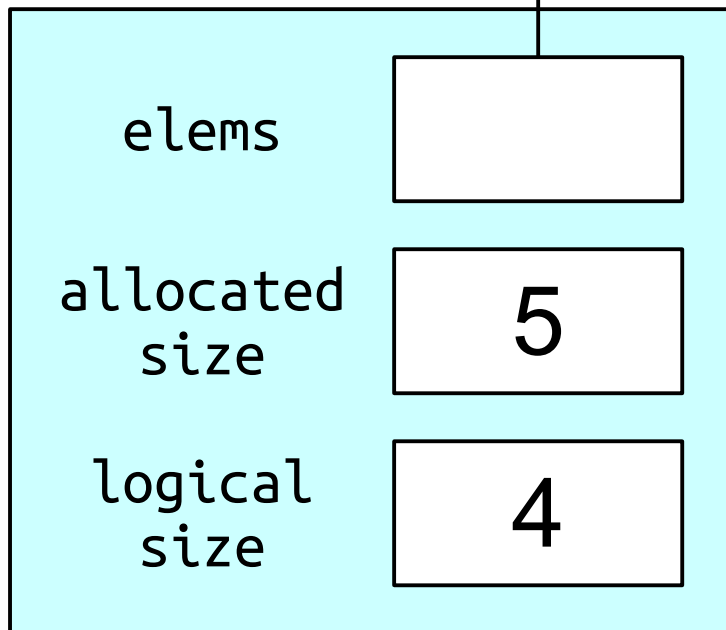
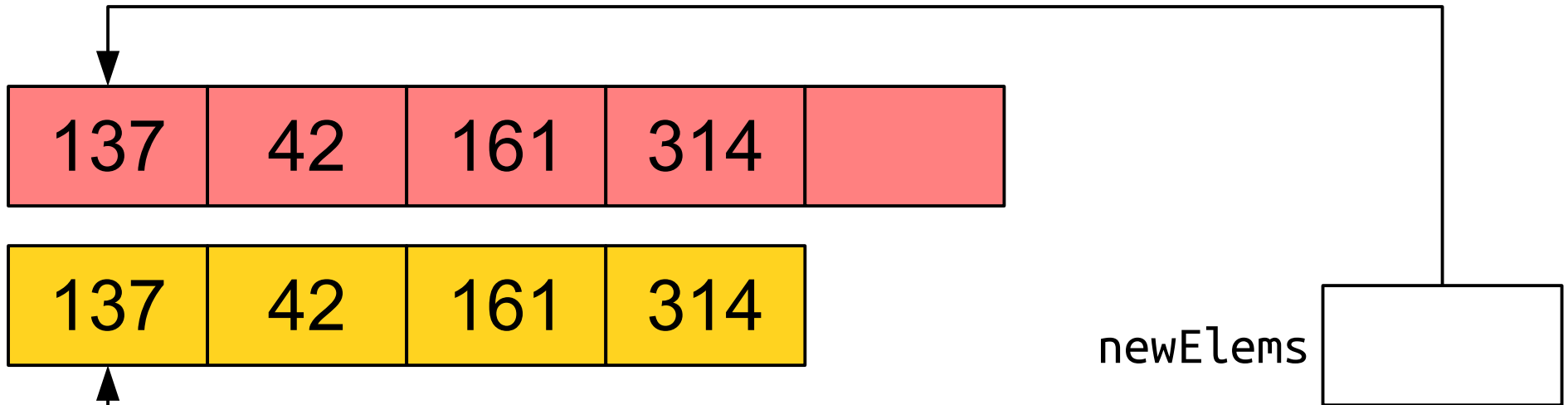
```
void OurStack::grow() {  
    allocatedSize++;  
  
    int* newElems = new int[allocatedSize];  
  
}
```

# An Initial Idea



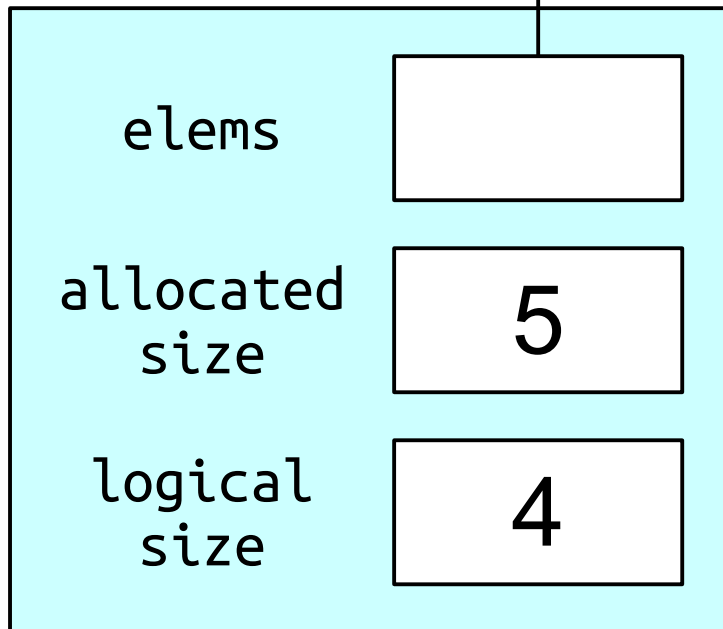
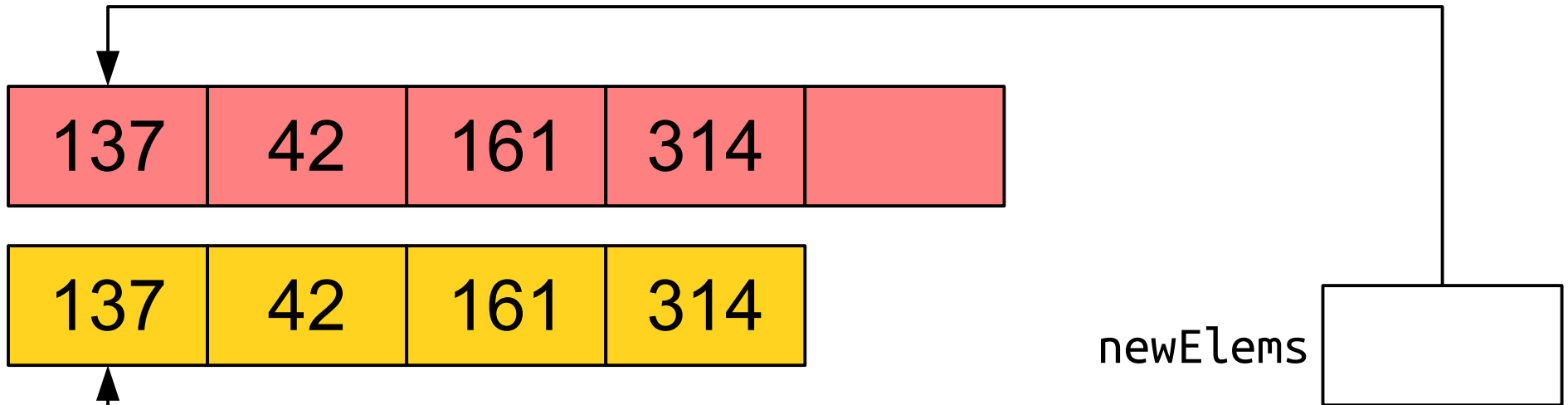
```
void OurStack::grow() {  
    allocatedSize++;  
  
    int* newElems = new int[allocatedSize];  
  
    for (int i = 0; i < size(); i++) {  
        newElems[i] = elems[i];  
    }  
  
}
```

# An Initial Idea



```
void OurStack::grow() {  
    allocatedSize++;  
  
    int* newElems = new int[allocatedSize];  
  
    for (int i = 0; i < size(); i++) {  
        newElems[i] = elems[i];  
    }  
  
}
```

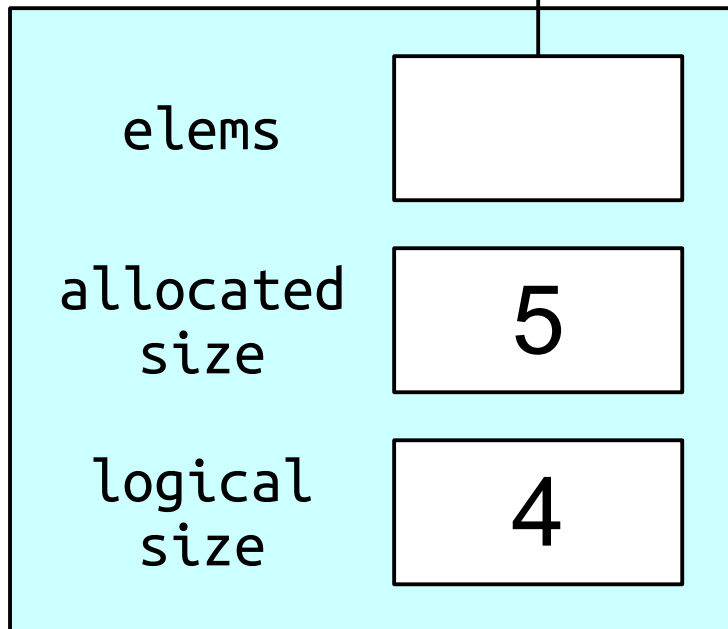
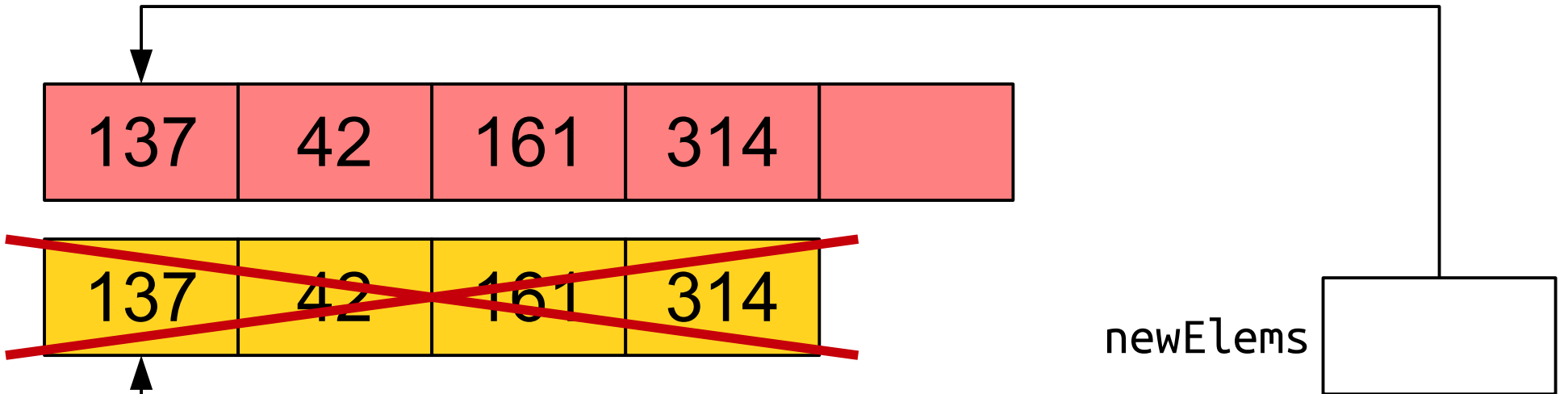
# An Initial Idea



```
void OurStack::grow() {  
    allocatedSize++;  
  
    int* newElems = new int[allocatedSize];  
  
    for (int i = 0; i < size(); i++) {  
        newElems[i] = elems[i];  
    }  
  
    delete[] elems;  
}
```

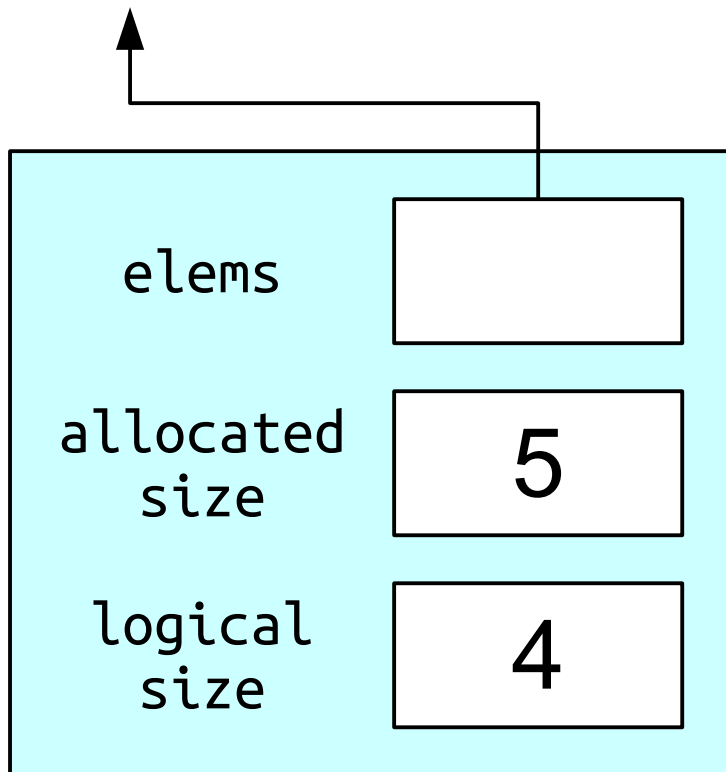
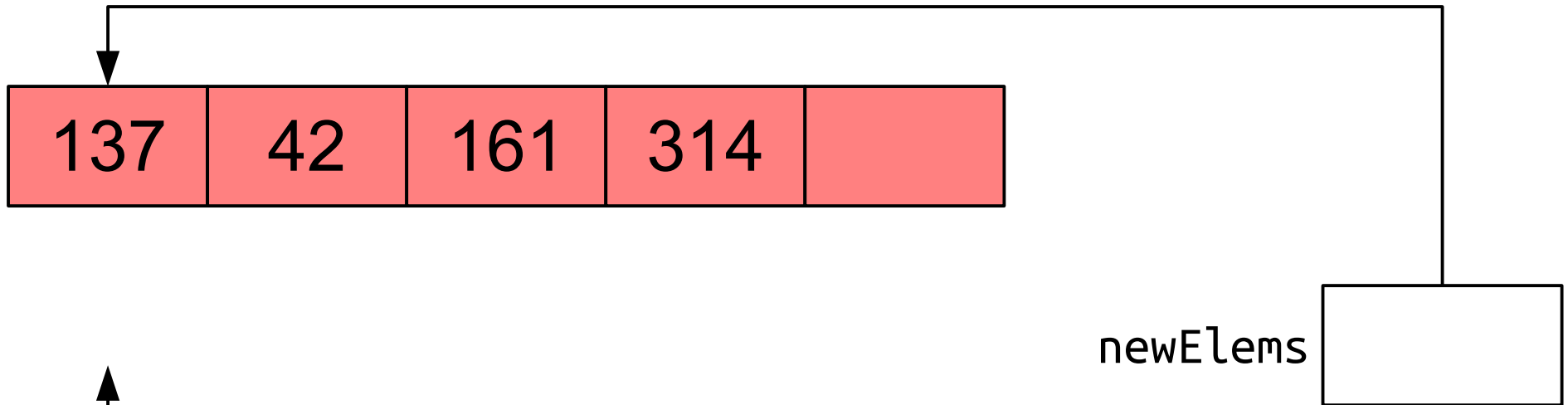


# An Initial Idea



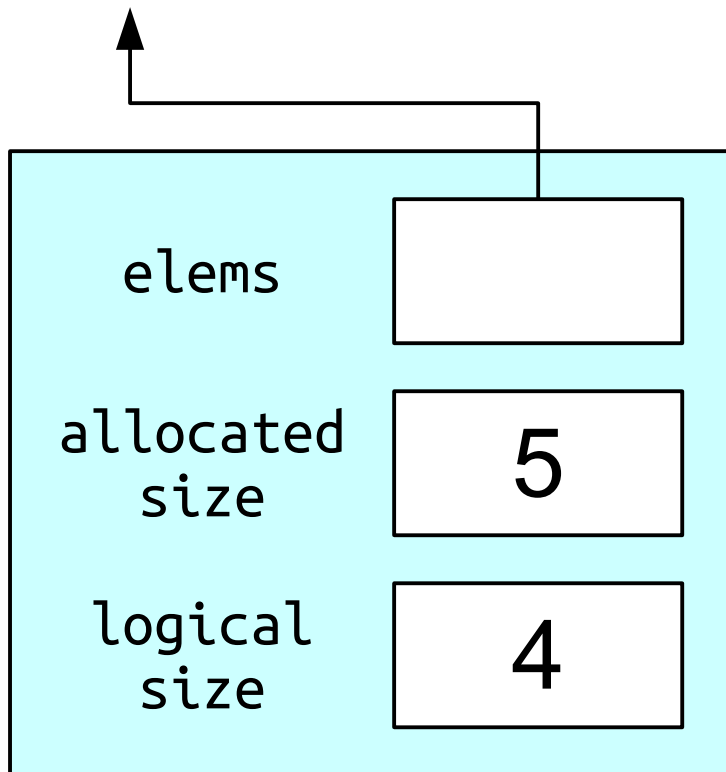
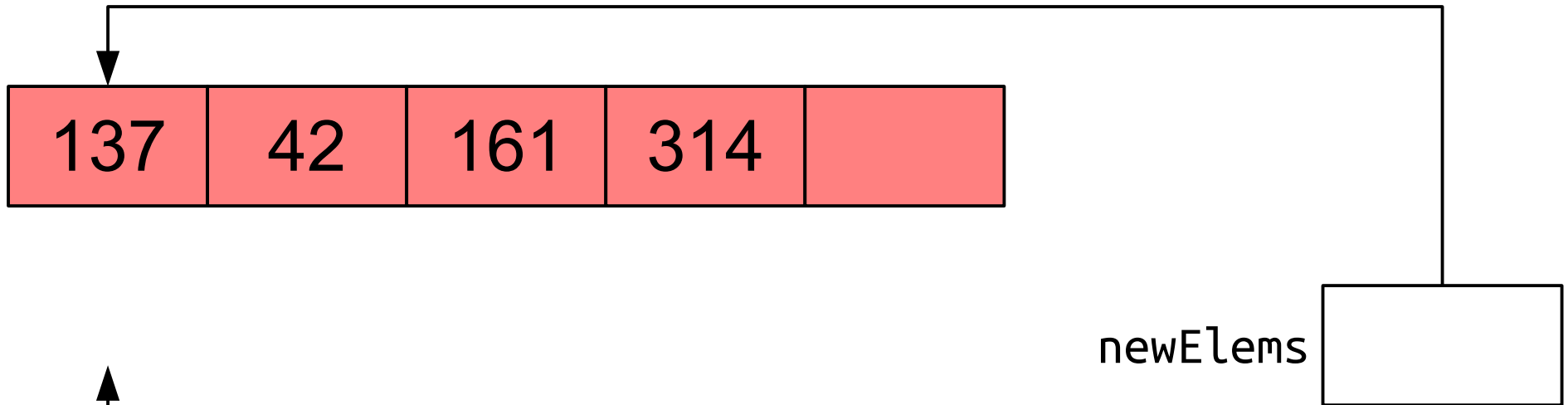
```
void OurStack::grow() {  
    allocatedSize++;  
  
    int* newElems = new int[allocatedSize];  
  
    for (int i = 0; i < size(); i++) {  
        newElems[i] = elems[i];  
    }  
  
    delete[] elems;  
  
}
```

# An Initial Idea



```
void OurStack::grow() {  
    allocatedSize++;  
  
    int* newElems = new int[allocatedSize];  
  
    for (int i = 0; i < size(); i++) {  
        newElems[i] = elems[i];  
    }  
  
    delete[] elems;  
  
}
```

# An Initial Idea



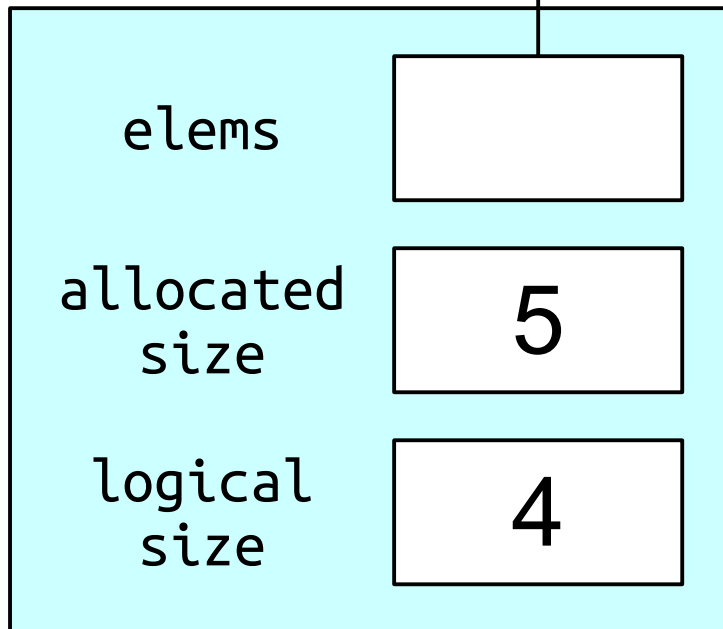
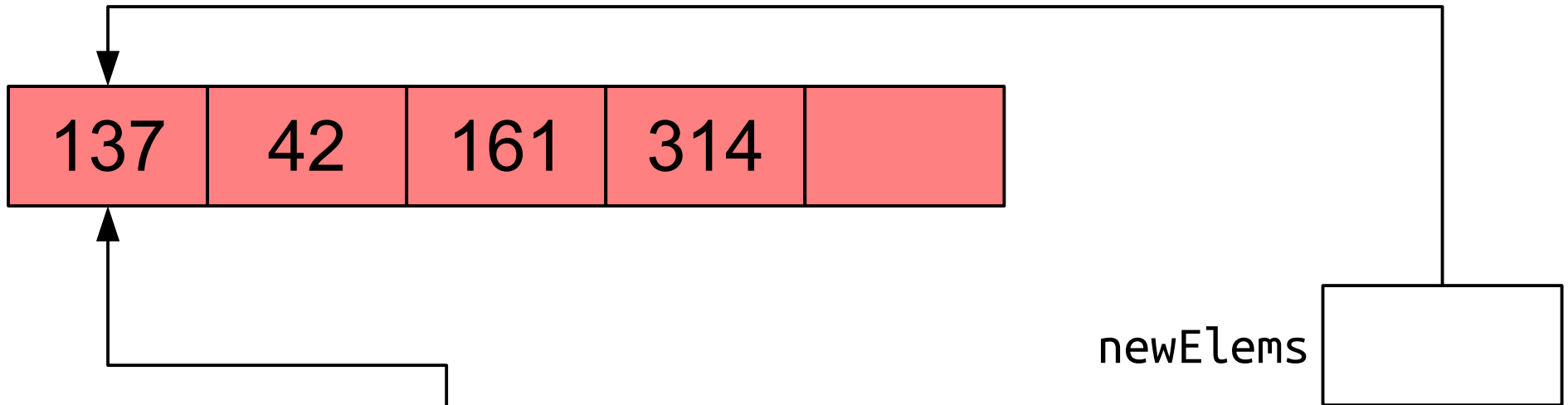
```
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

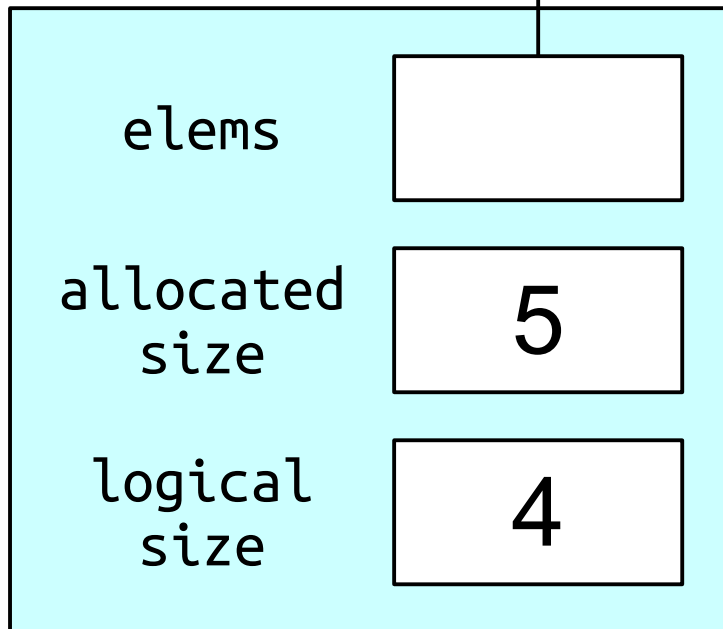
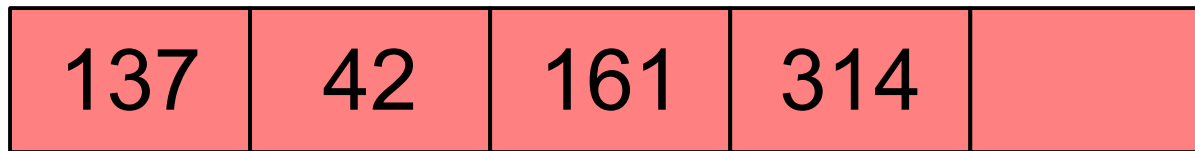
    delete[] elems;
    elems = newElems;
}
```

# An Initial Idea



```
void OurStack::grow() {  
    allocatedSize++;  
  
    int* newElems = new int[allocatedSize];  
  
    for (int i = 0; i < size(); i++) {  
        newElems[i] = elems[i];  
    }  
  
    delete[] elems;  
    elems = newElems;  
}
```

# An Initial Idea



```
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

    delete[] elems;
    elems = newElems;
}
```

# 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:
  - peek:

# 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:  **$O(1)$**
  - peek:  **$O(1)$**

# 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

**Time-Out for Announcements!**



# Assignment 4

- Assignment 4 is due on Friday.
- Recommendation: Aim to complete all the parts of the assignment by the end of this evening.
- We've posted a handy ***Assignment Submission Checklist*** up on the course website. Work through this before submitting - it'll help make sure your code is ready to go!

# Midterm Exam

- The midterm exam is next Tuesday, February 21 from 7:00PM – 10:00PM.
  - Location TBA
- Covers topics up through and including big-O notation, plus Assignments 0 – 4.
- Closed-book, closed-computer, limited-note. You get one double-sided sheet of 8.5" × 11" notes when you take the exam. We also provide a library reference sheet.
- Practice exam posted on the course website.
- Need some practice? Work through the section handouts, the chapter exercises in the textbook, and revisit old assignments. Need more practice? Let us know!



# HACK 101

How to Finesse a Hackathon

**Anyone can hack!!!**

**Tonight!**  
**WCC, 9PM – 10PM**

design · entrepreneurship · technology · teamwork

Want to check out Treehacks?  
A little nervous about it?  
Don't know anyone else who's doing it?

**Come to HACK 101!**

Learn how to be successful at a hackathon!  
Meet teammates for Treehacks!  
Start the ideation process for your project!

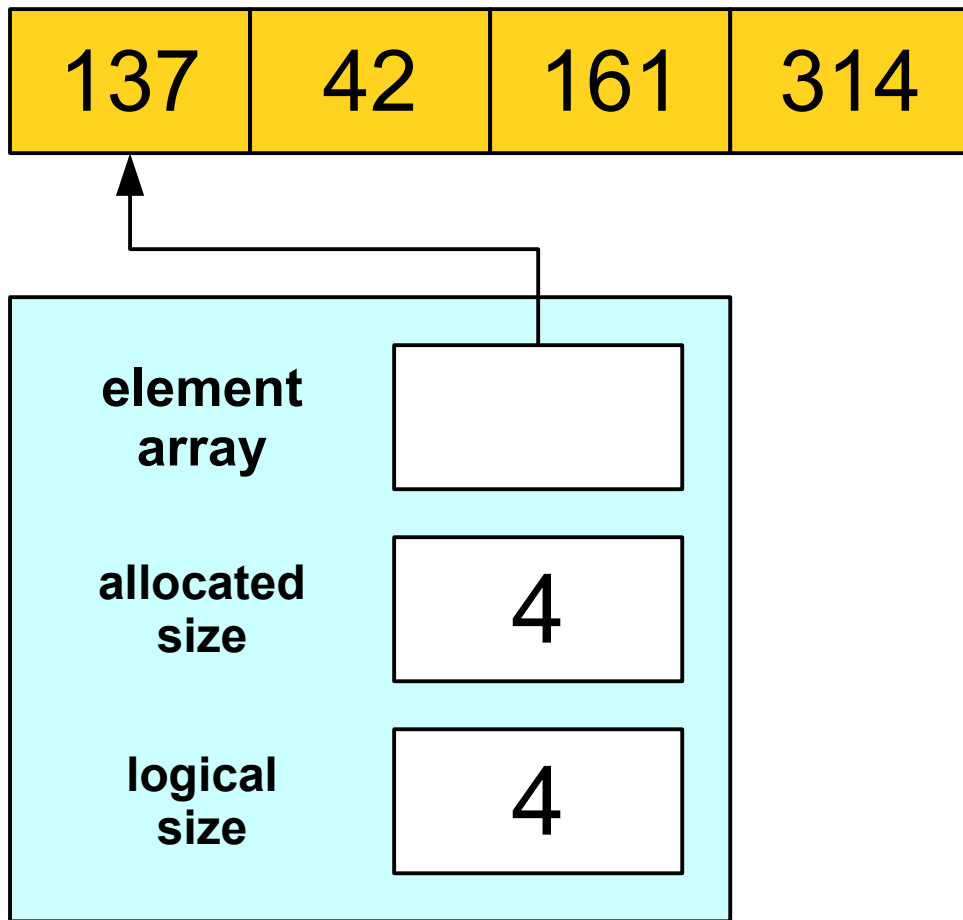
RSVP **HERE**  
*hosted by Black in CS*

Back to the Stack!

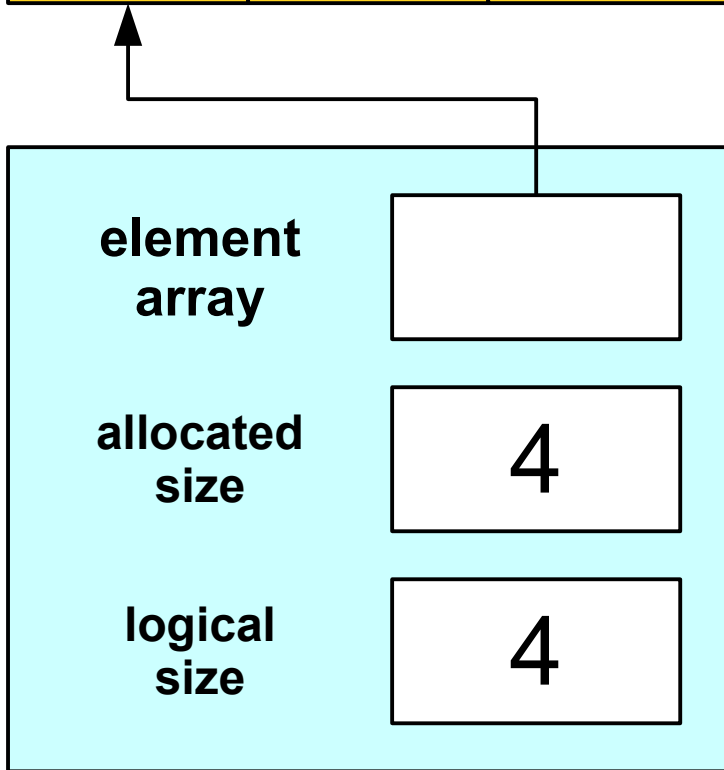
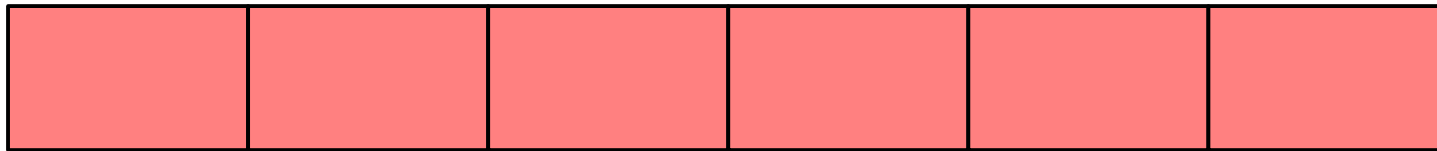
# Speeding up the Stack

Key Idea: ***Plan for the Future***

# A Better Idea

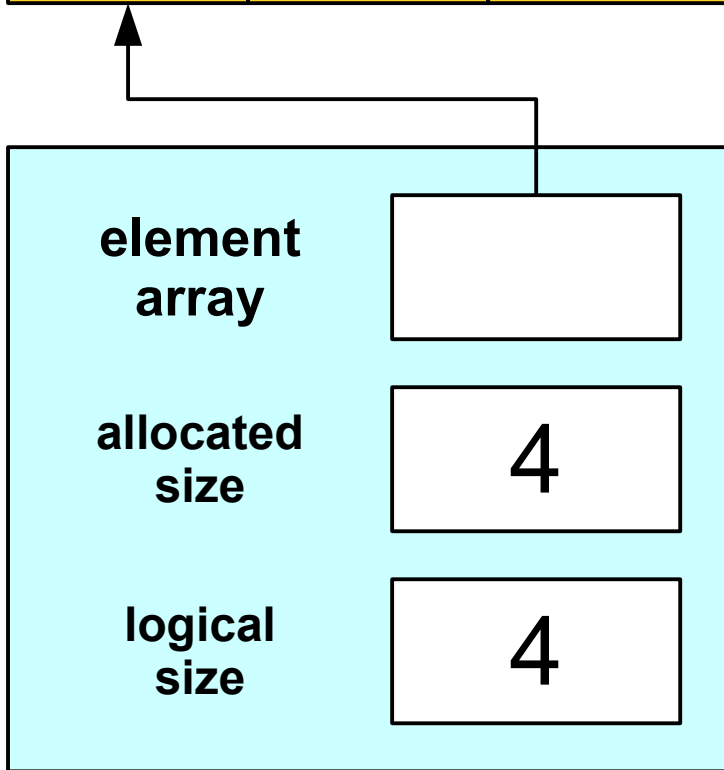
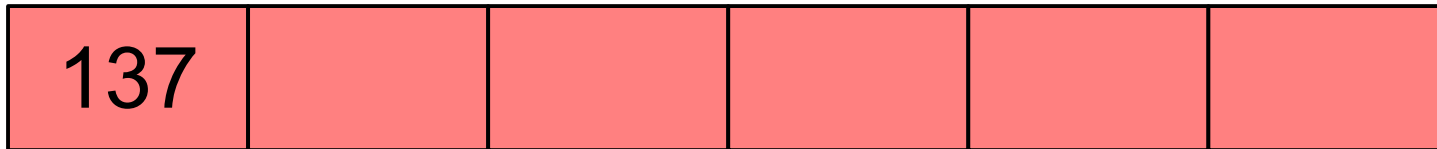


# A Better Idea

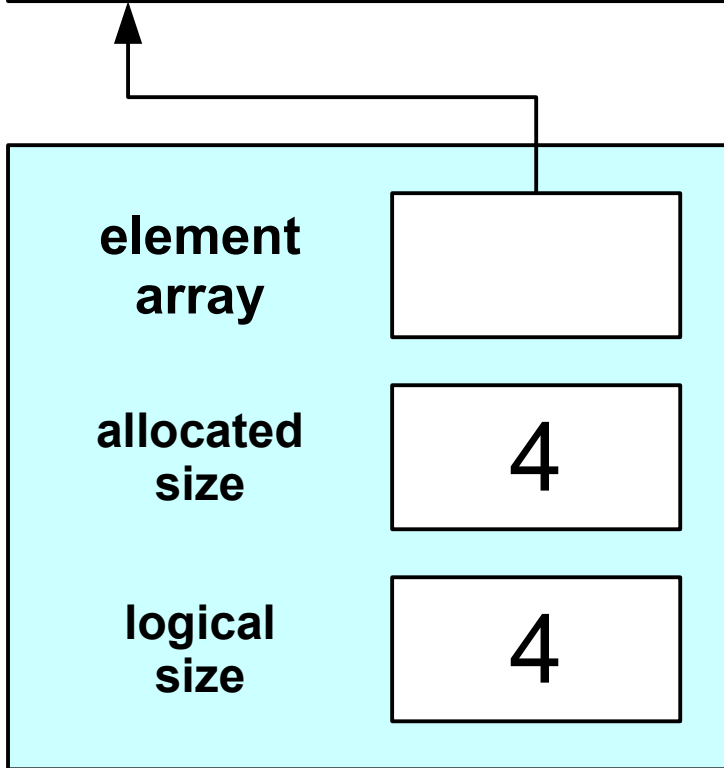
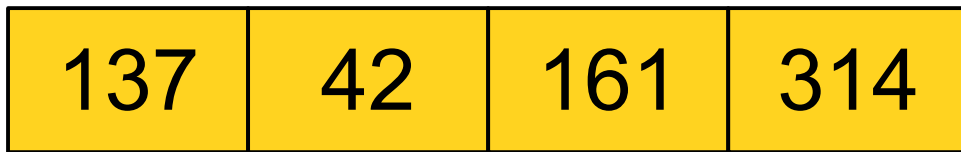
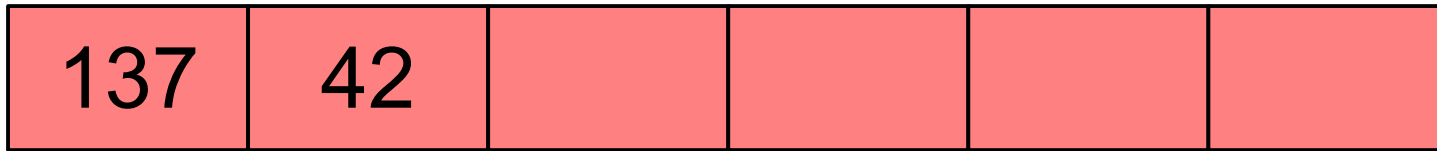




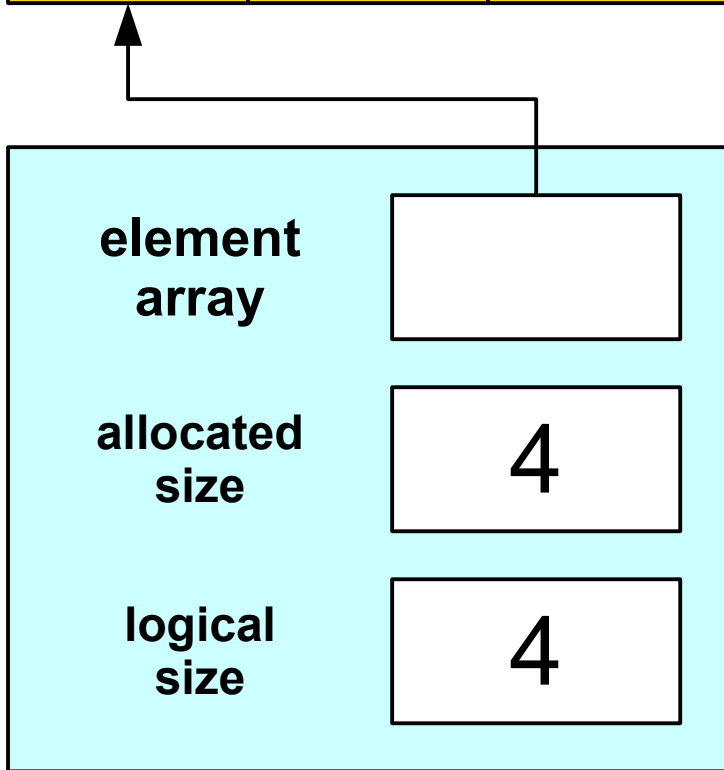
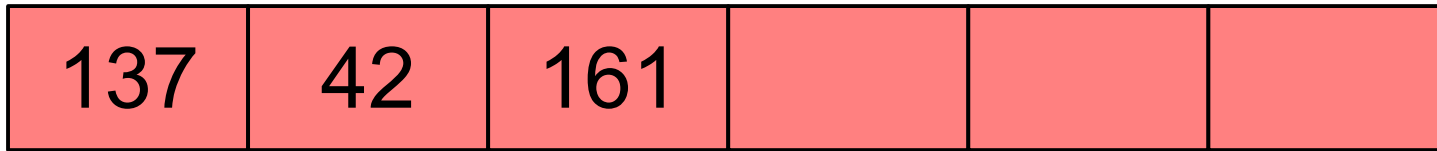
# A Better Idea



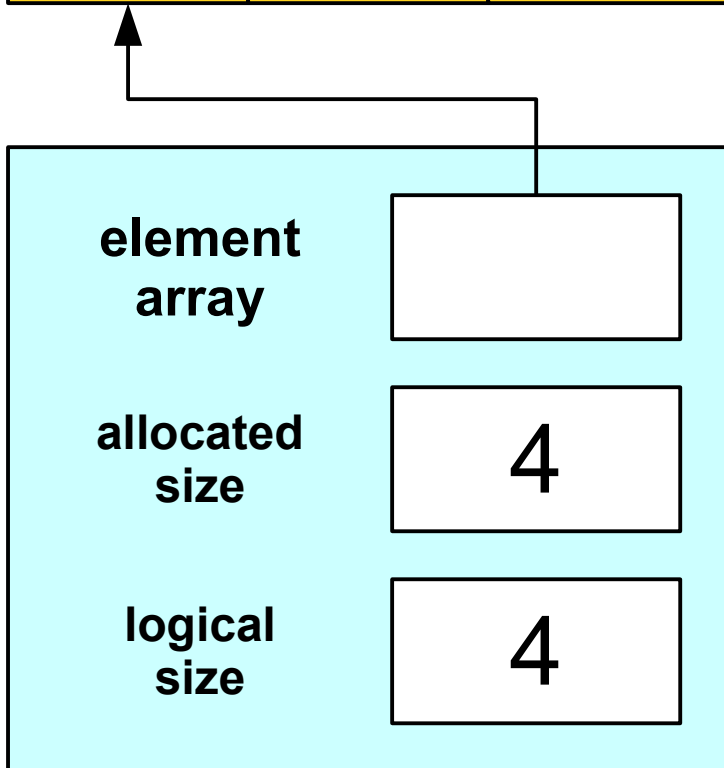
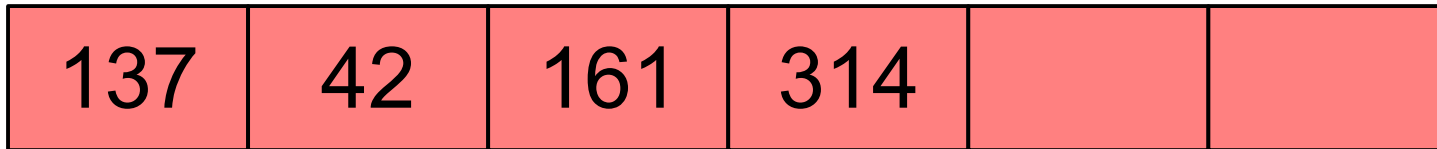
# A Better Idea



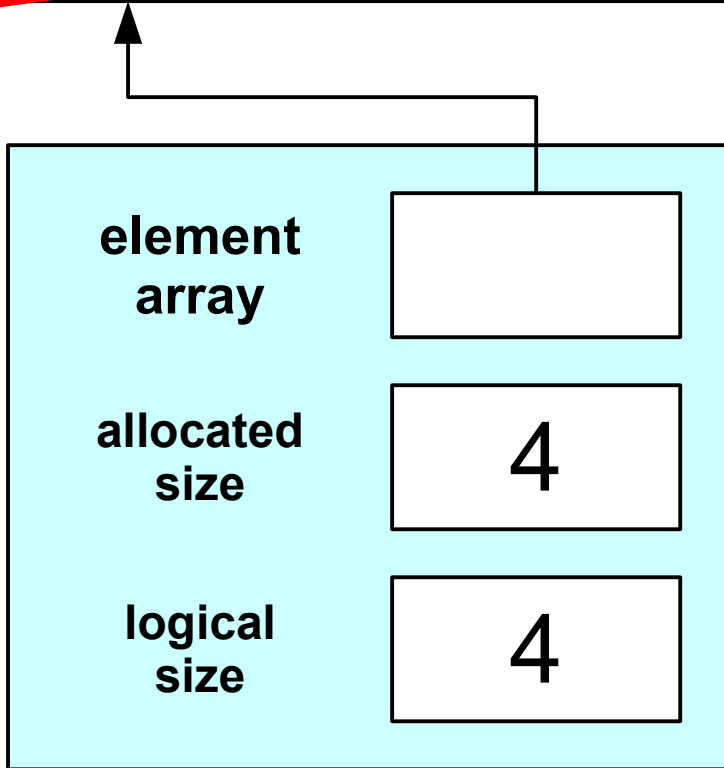
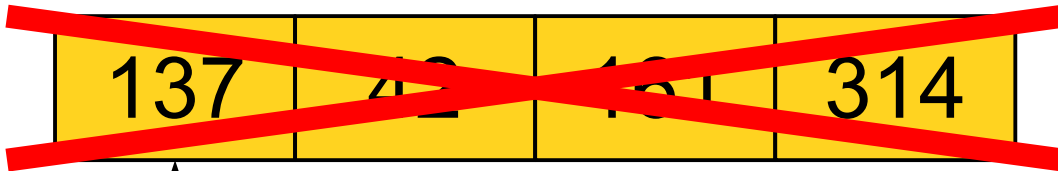
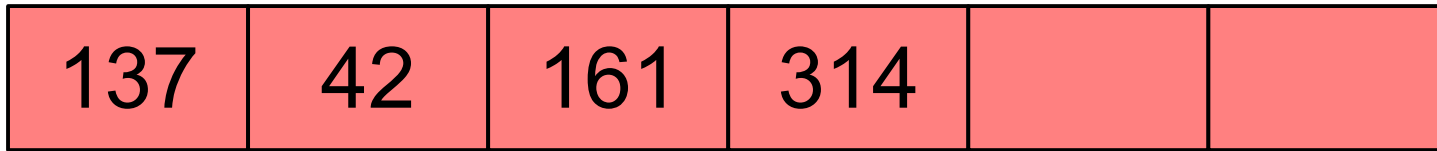
# A Better Idea



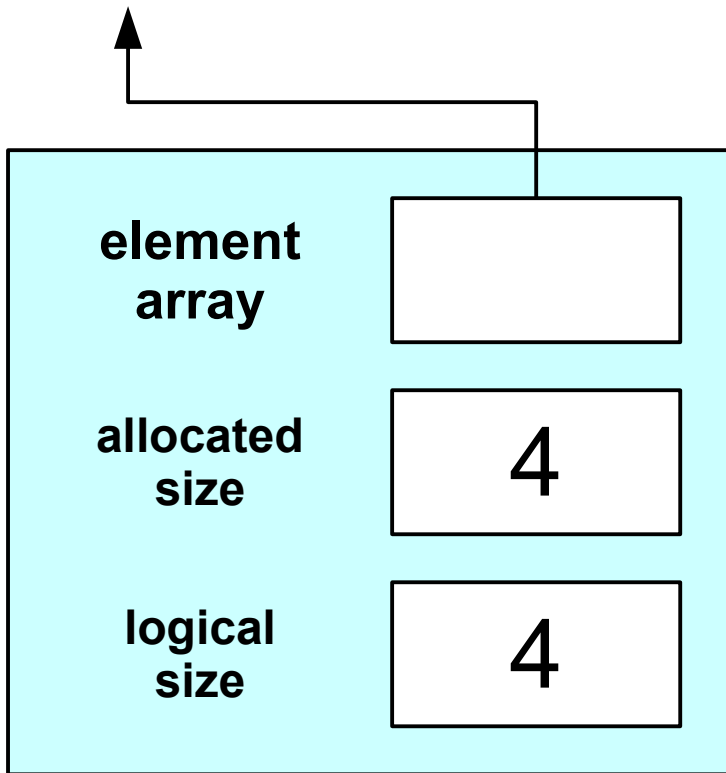
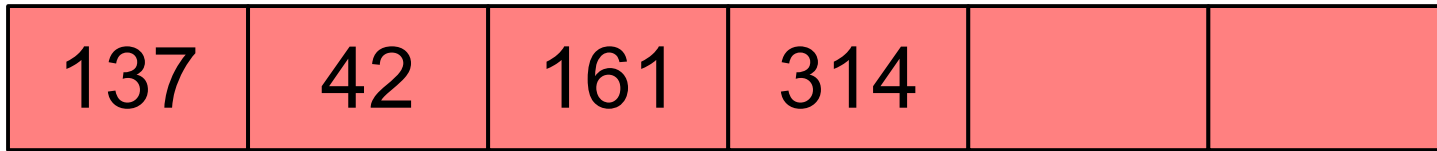
# A Better Idea



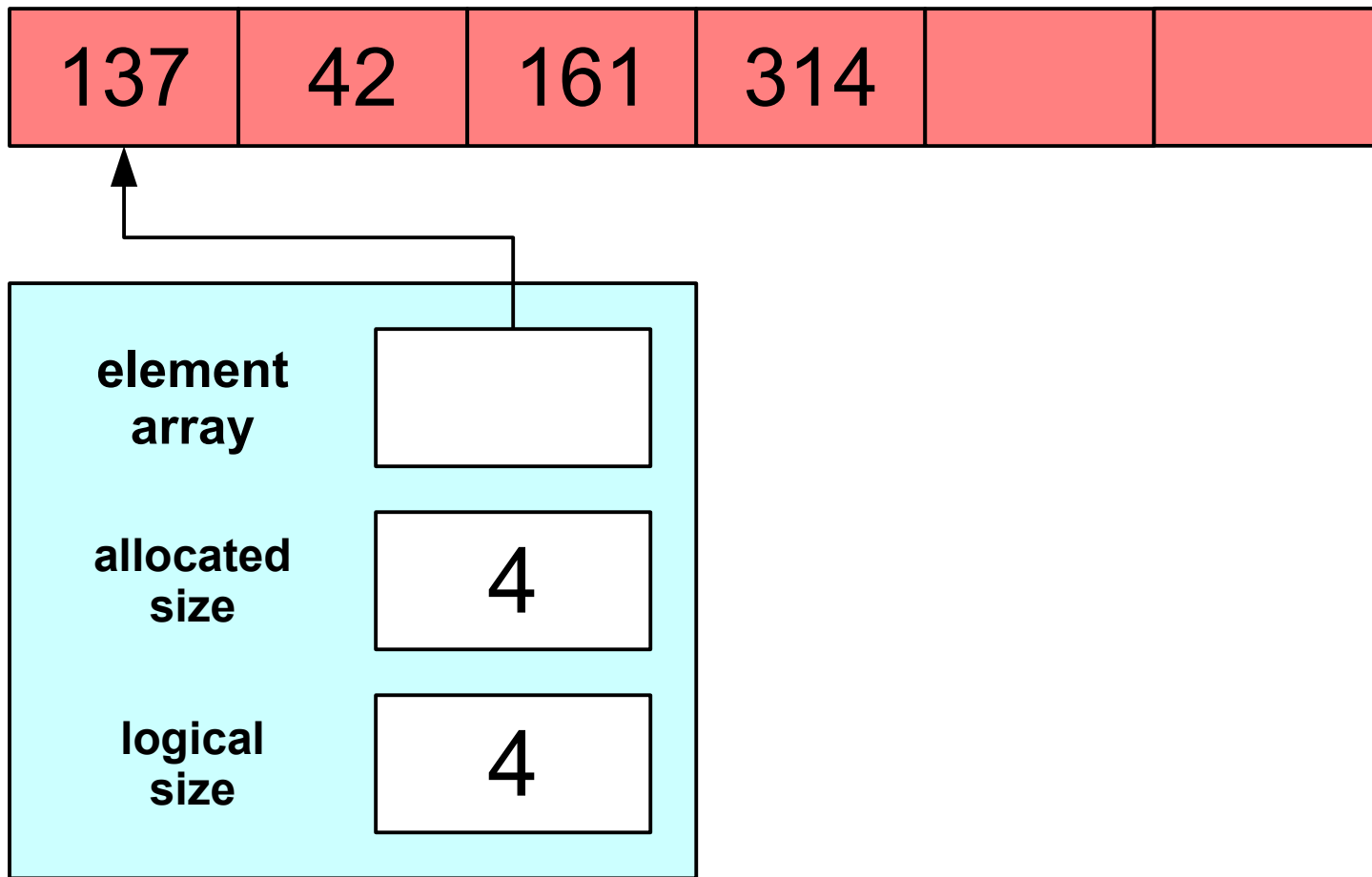
# A Better Idea



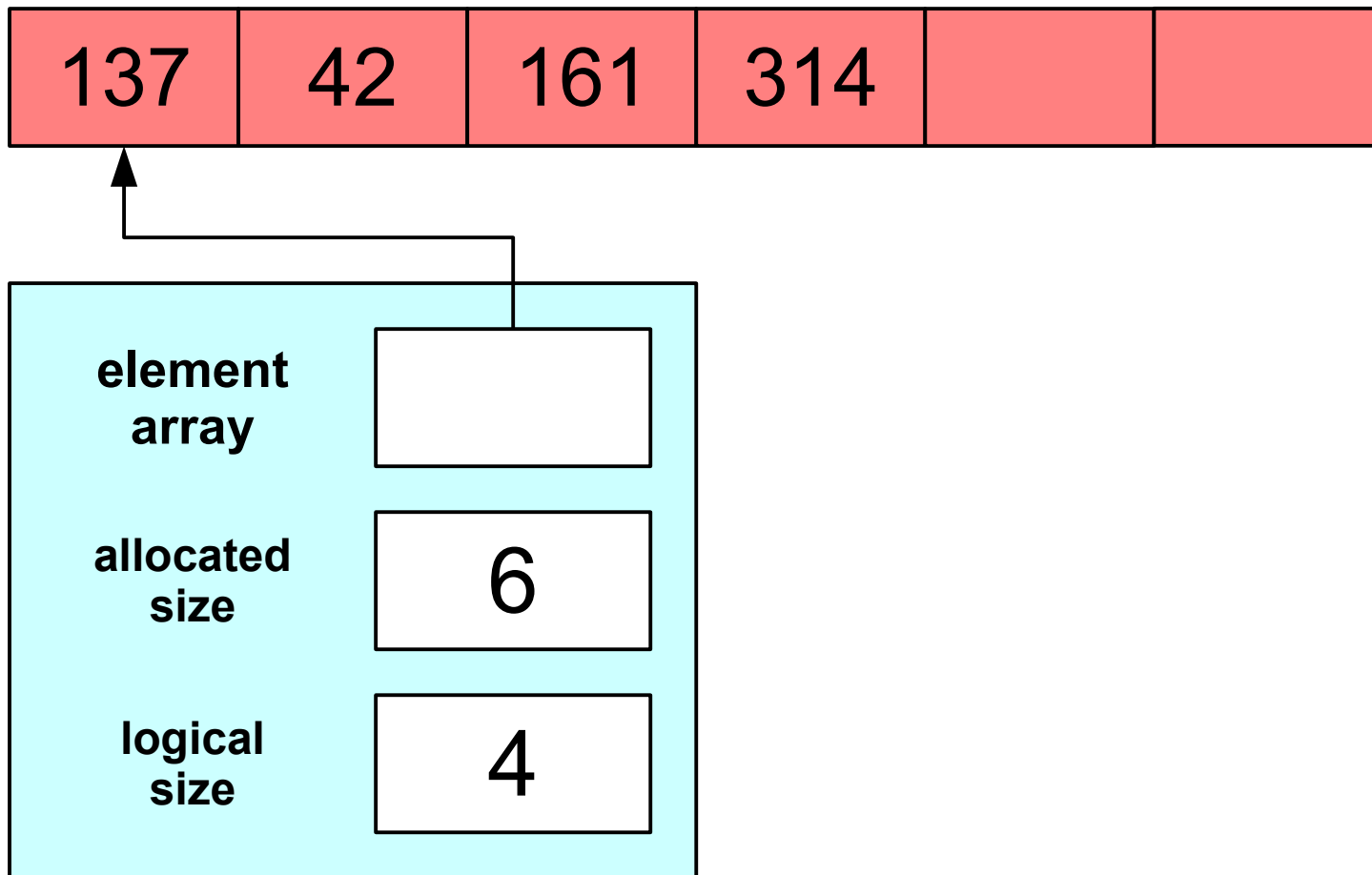
# A Better Idea



# A Better Idea

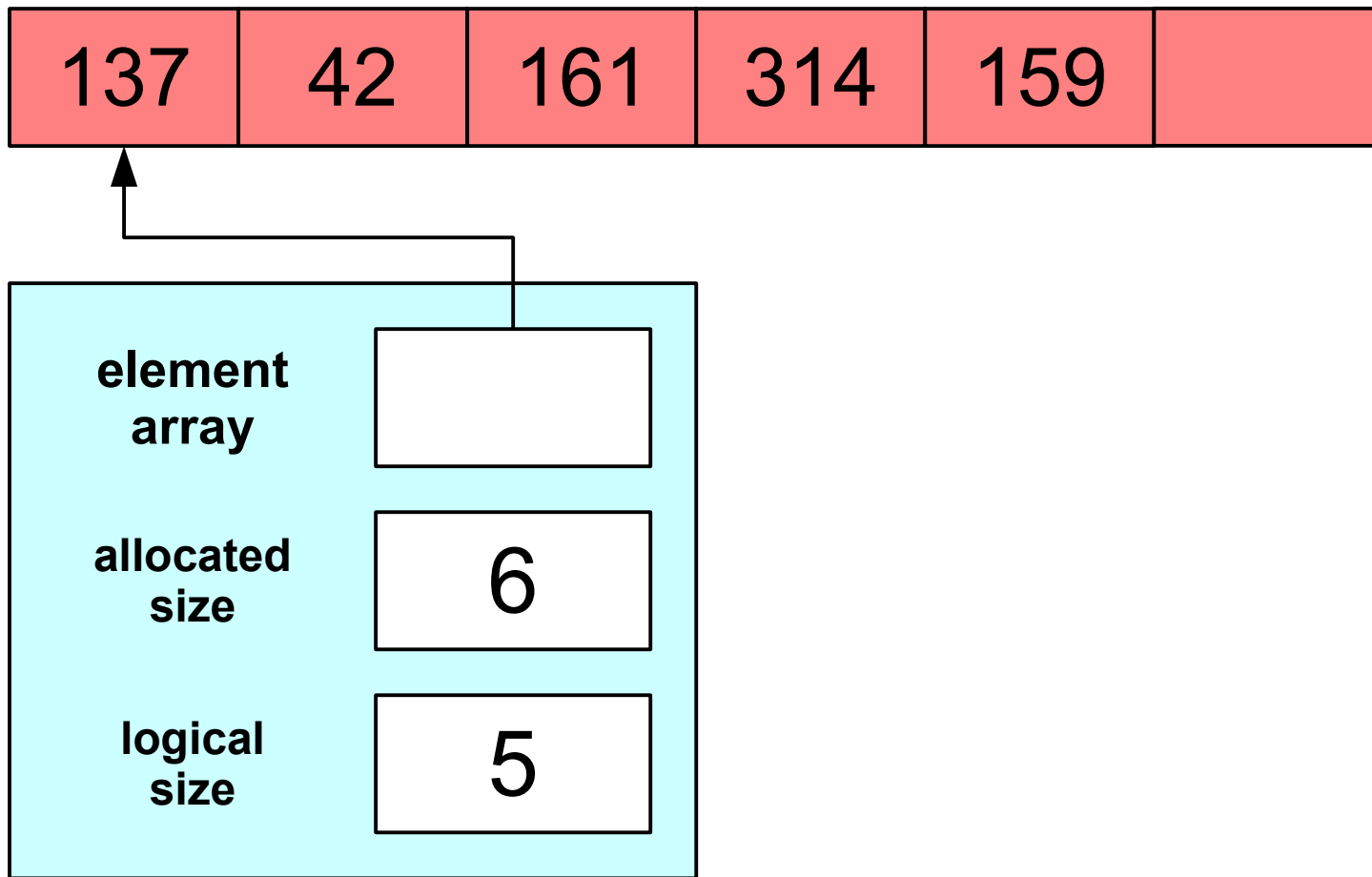


# A Better Idea

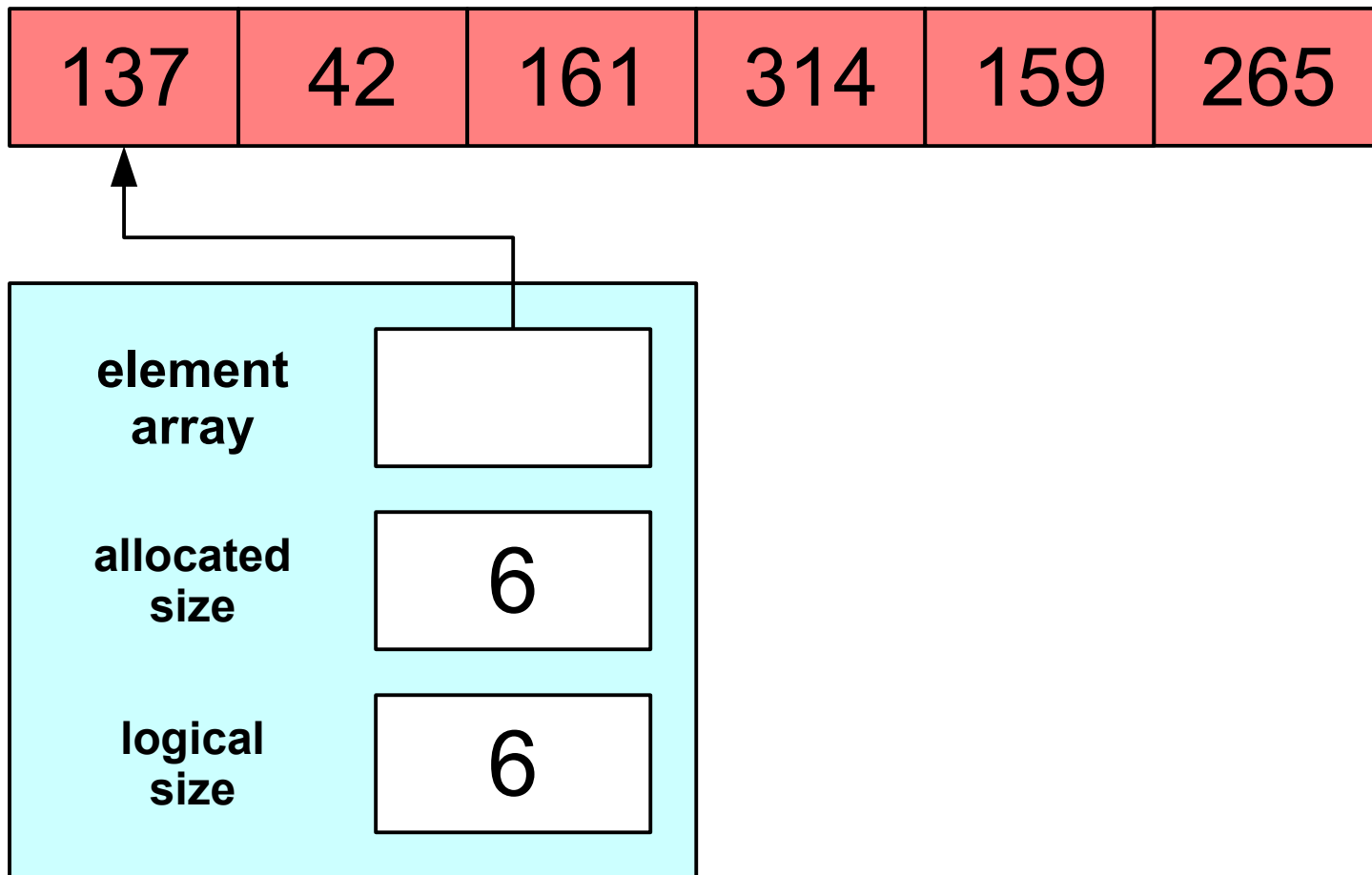




# A Better Idea



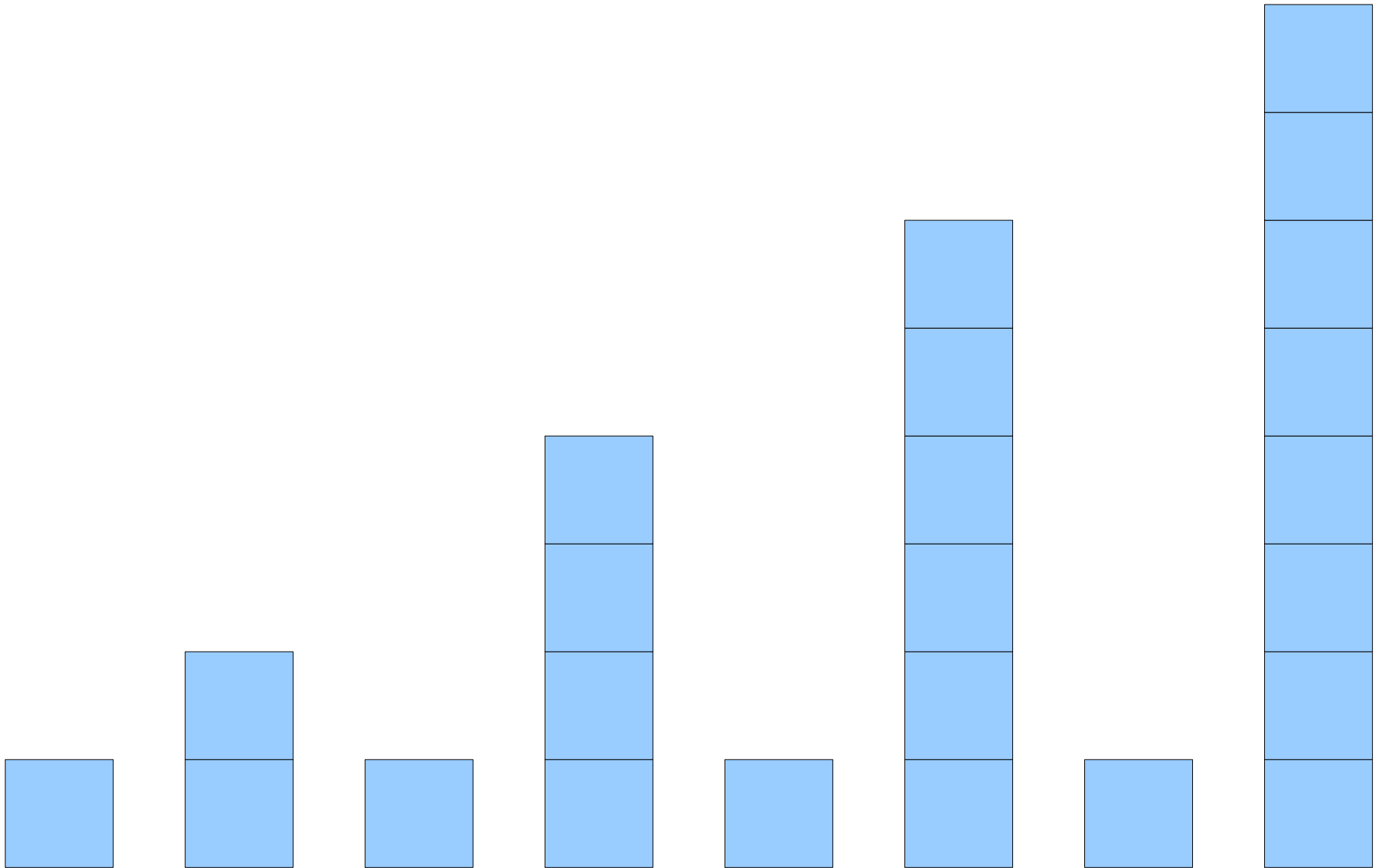
# A Better Idea



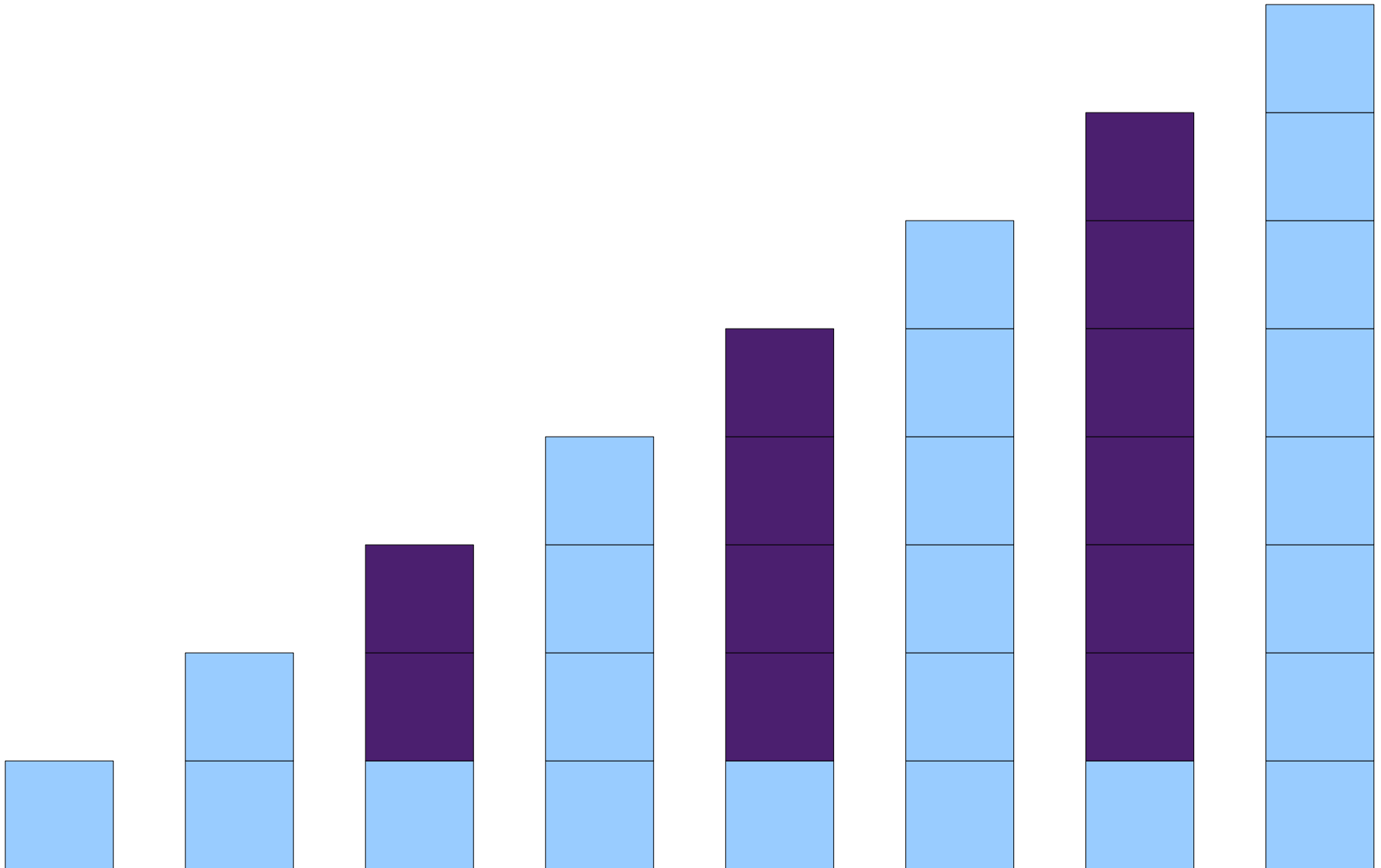
# What Just Happened?

- 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

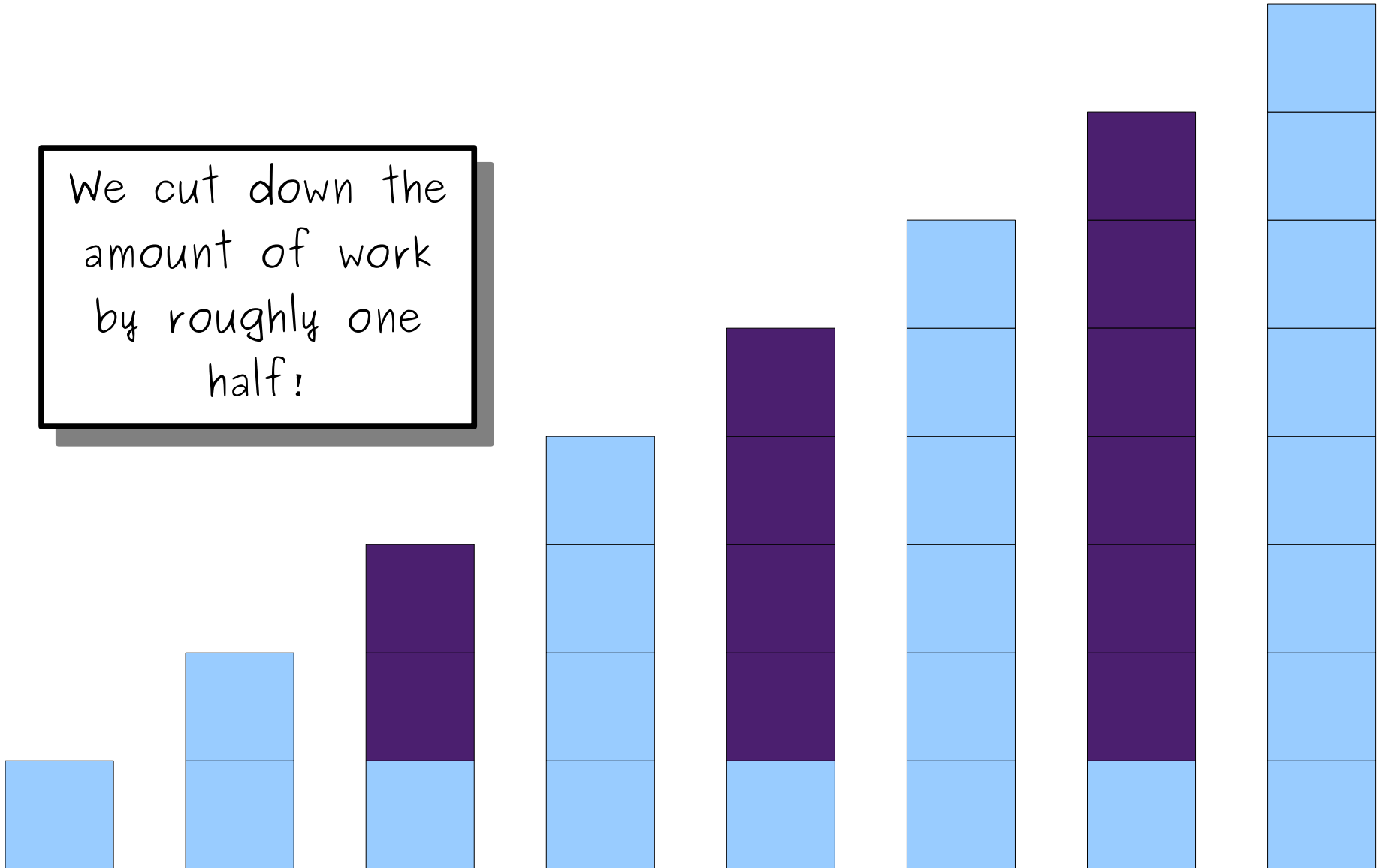


# Analyzing the Work



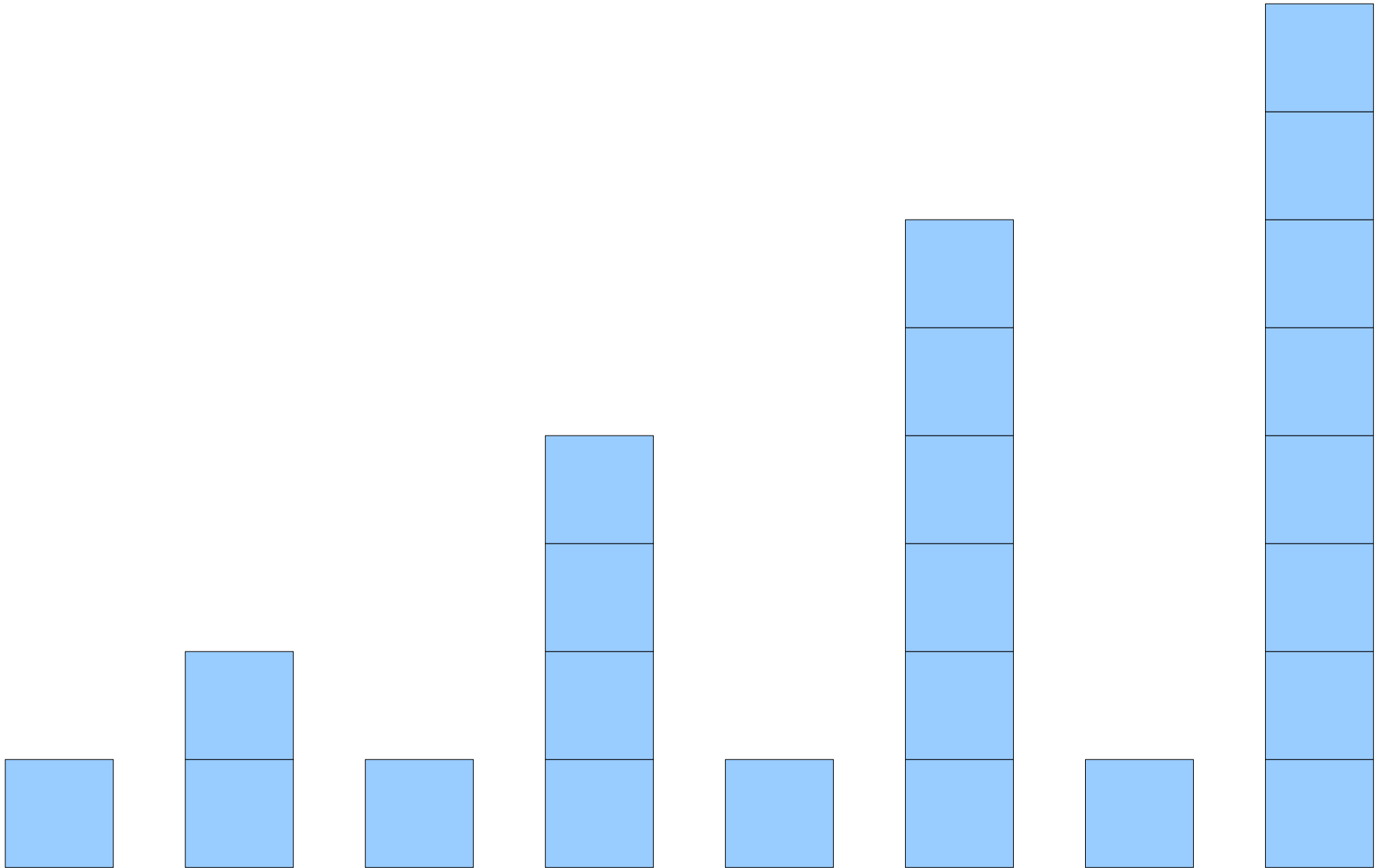
# Analyzing the Work

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



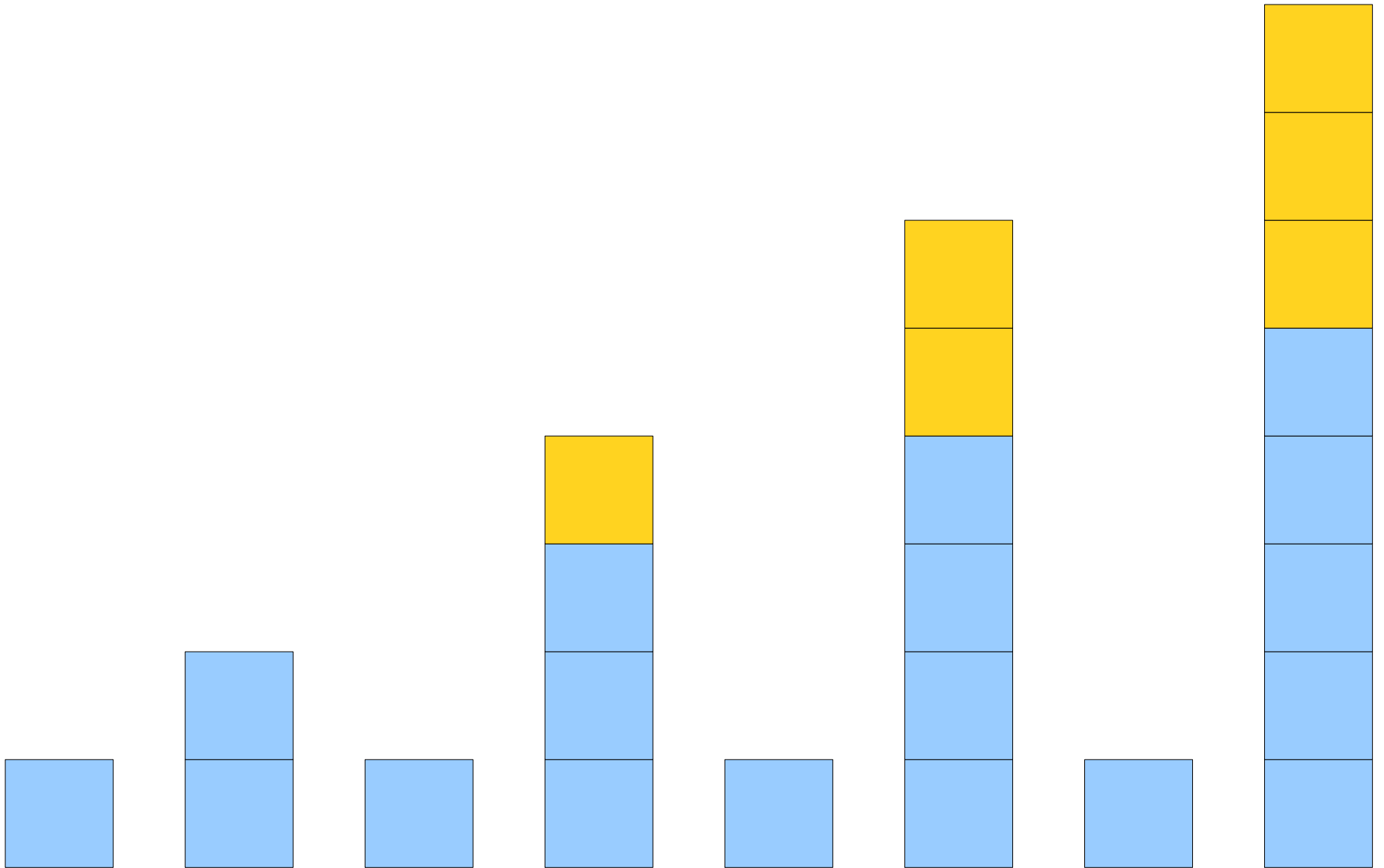
# A Different Analysis

# A Different Analysis

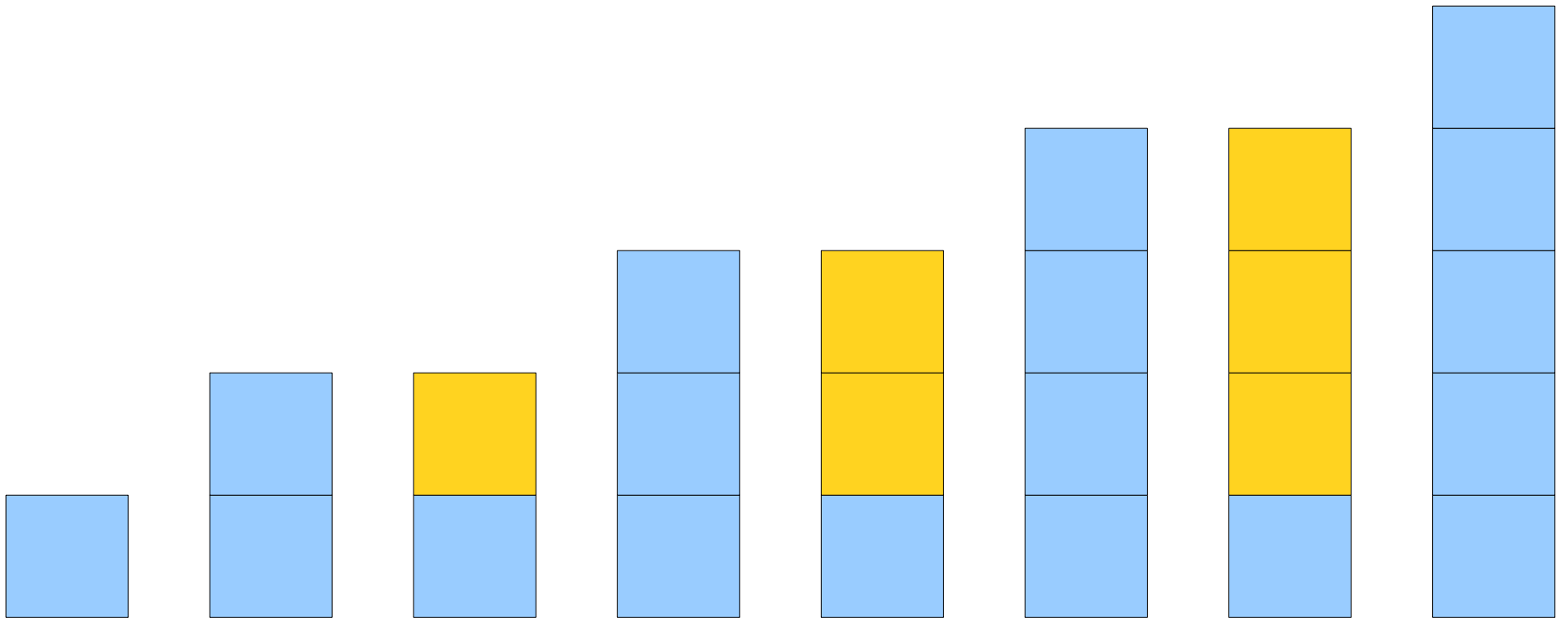




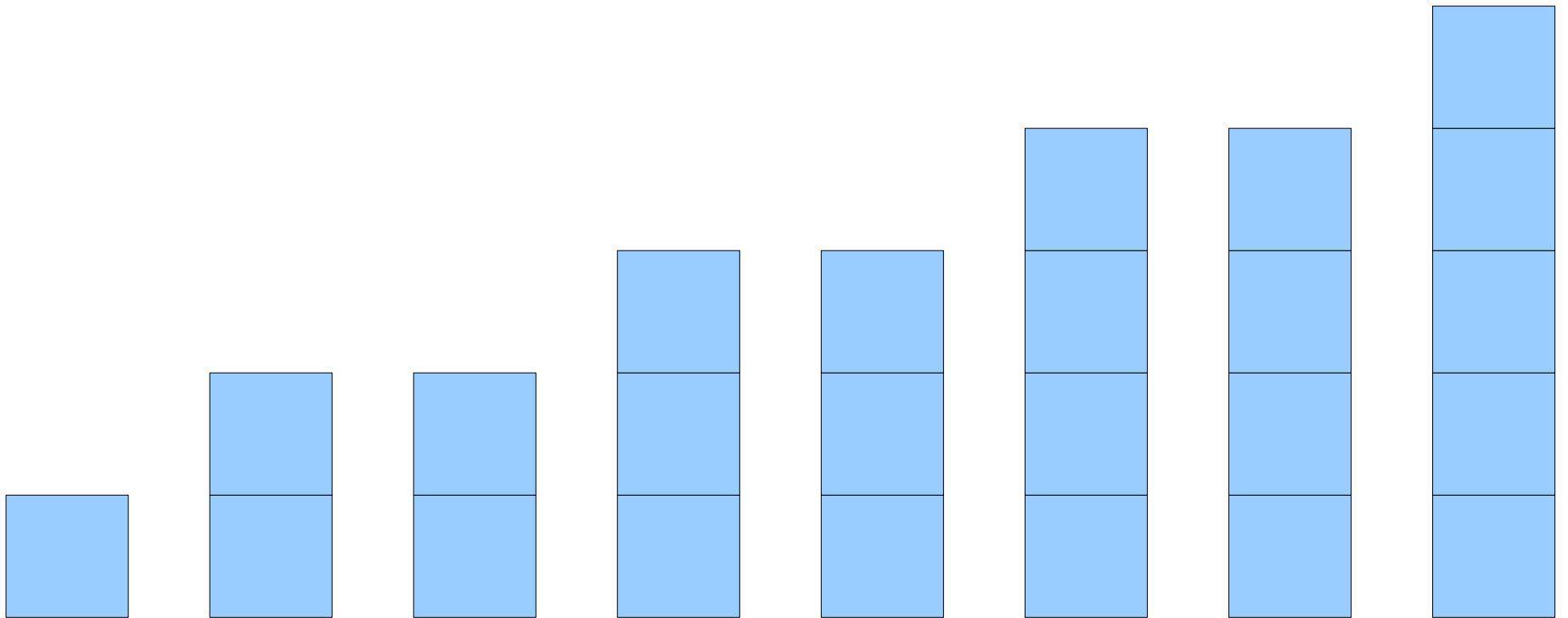
# A Different Analysis



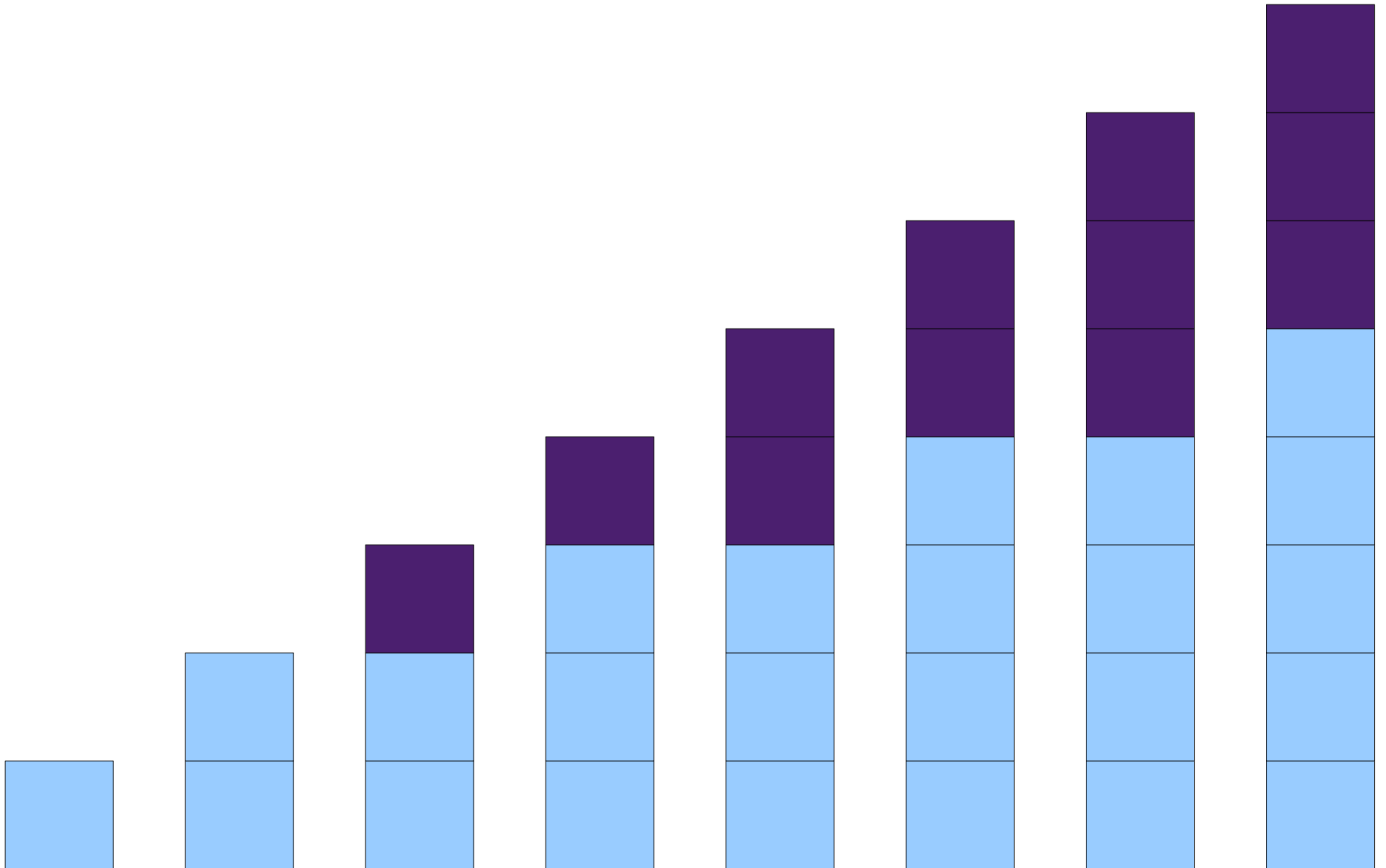
# A Different Analysis



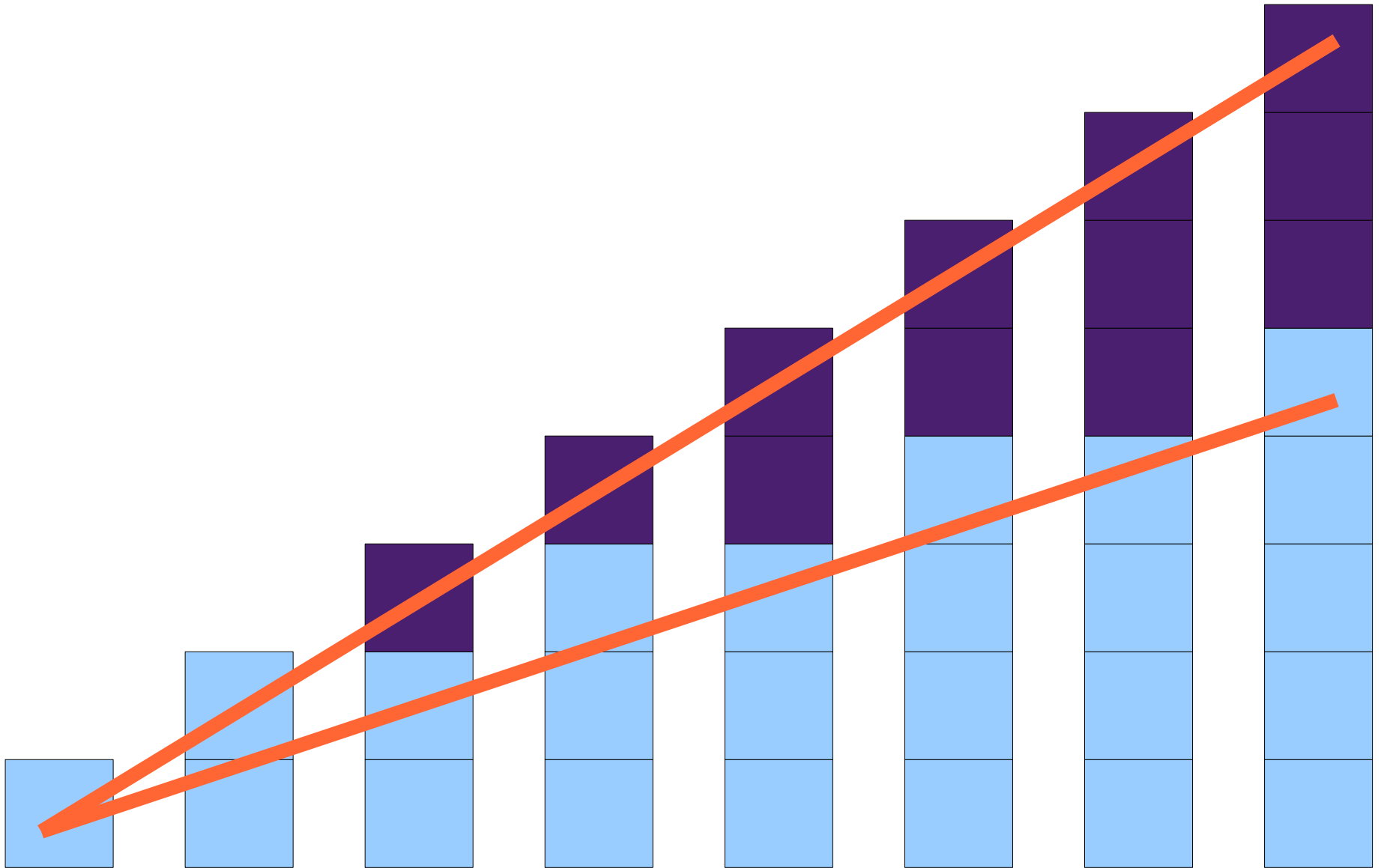
# A Different Analysis



# A Different Analysis

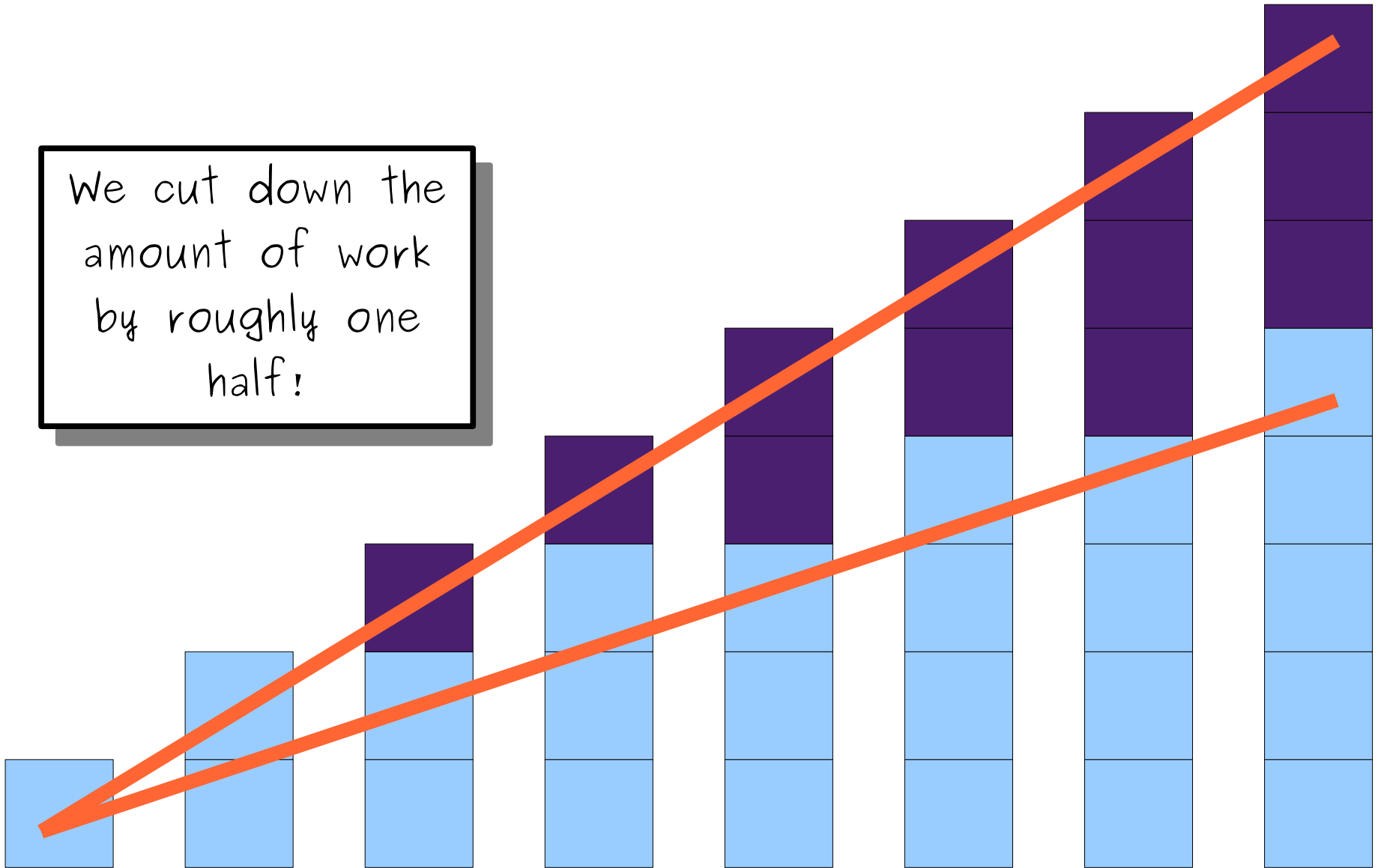


# A Different Analysis



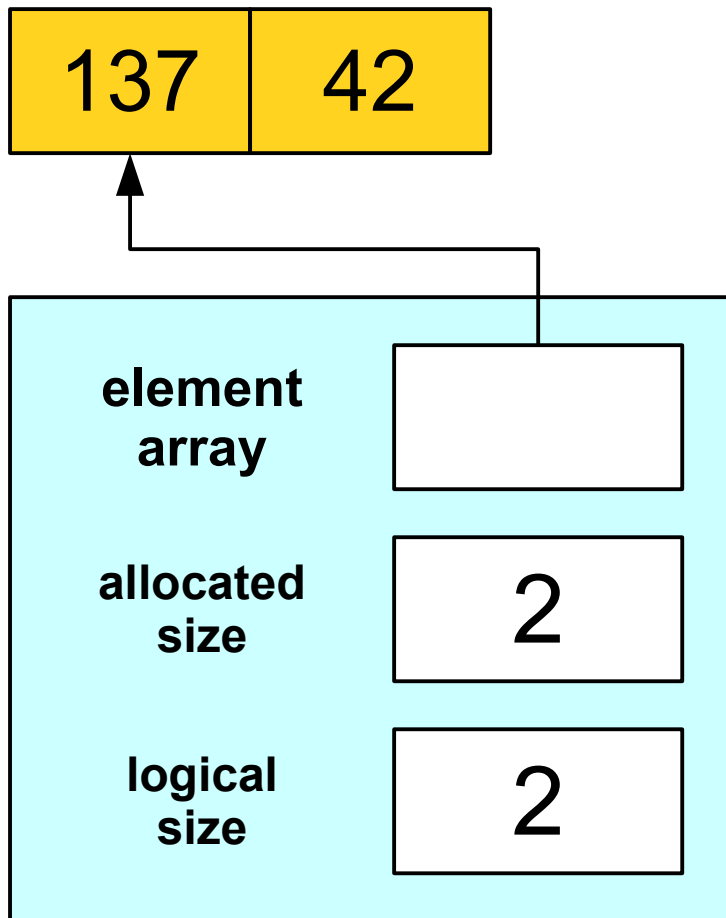
# A Different Analysis

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



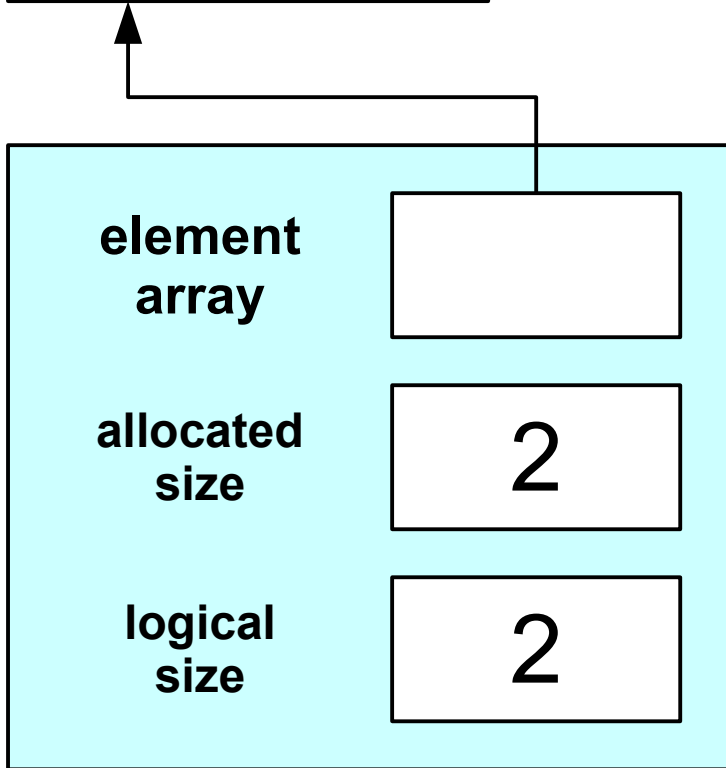
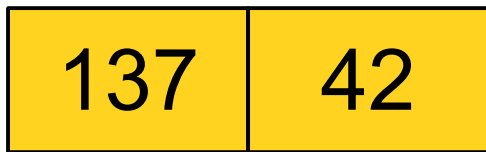
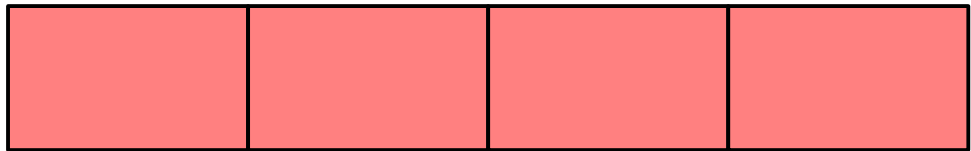
How does it stack up?

# A Much Better Idea

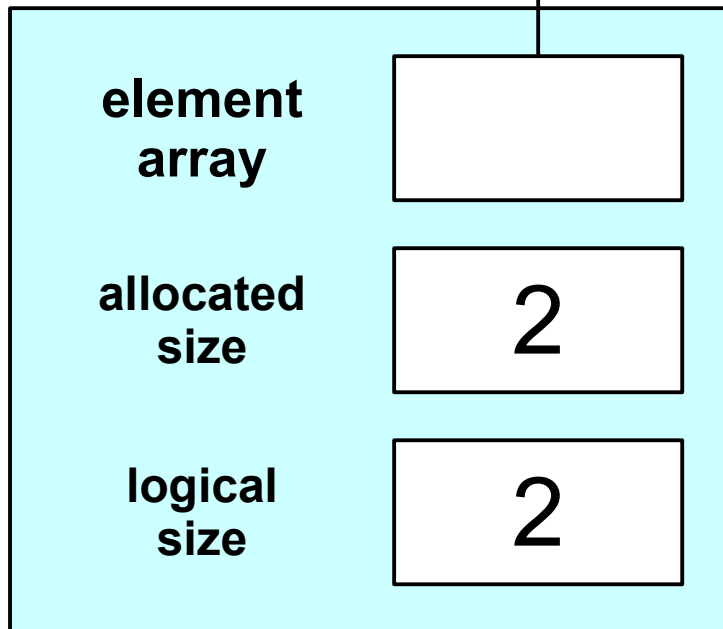
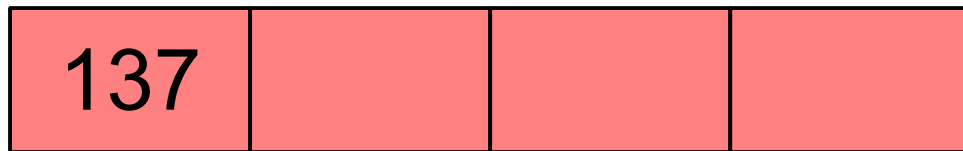




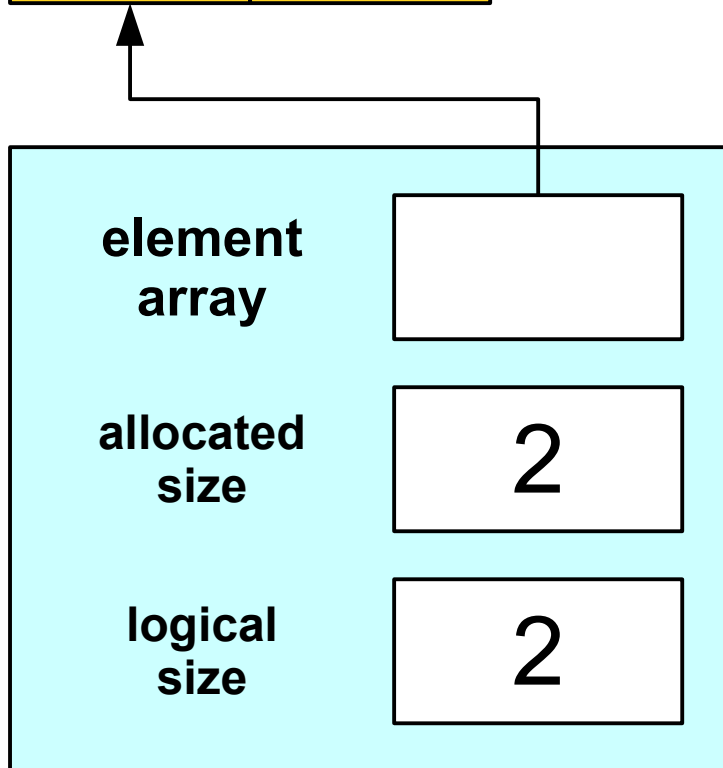
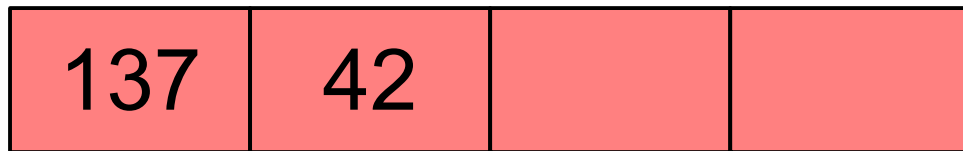
# A Much Better Idea



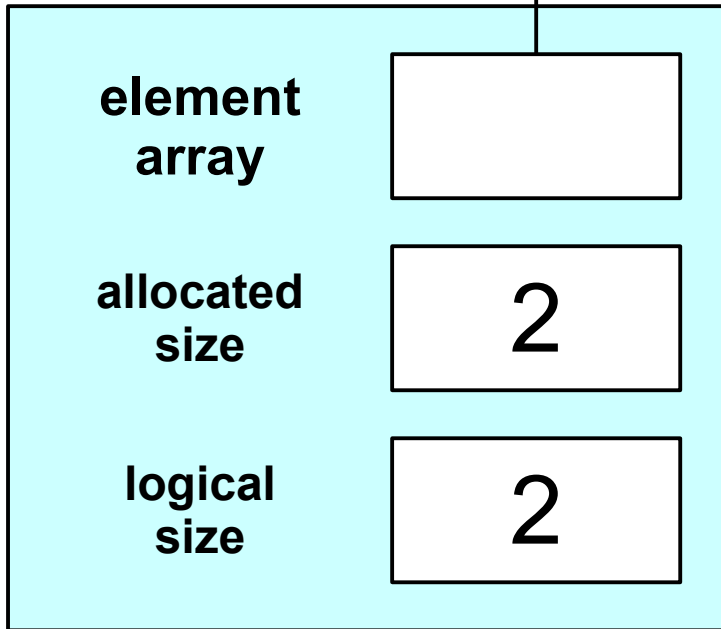
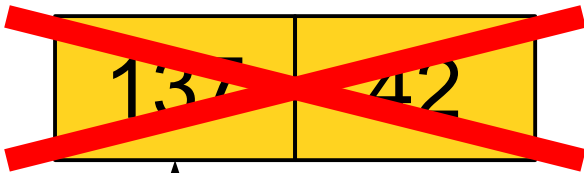
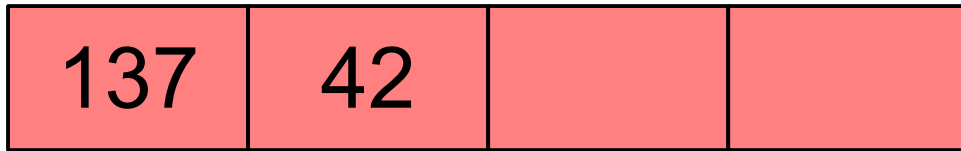
# A Much Better Idea



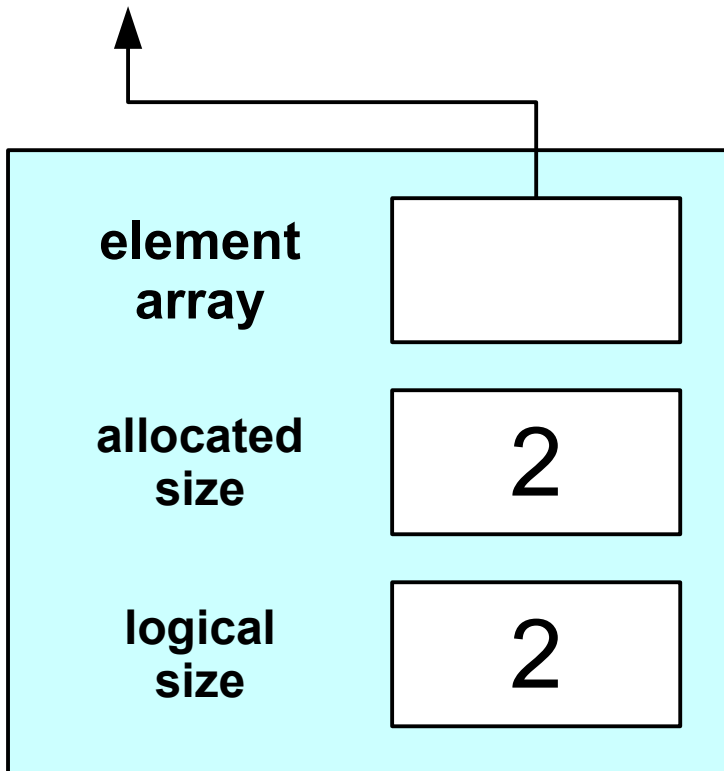
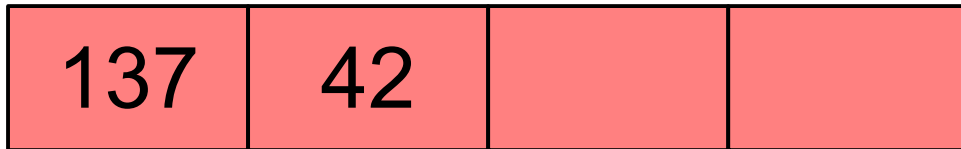
# A Much Better Idea



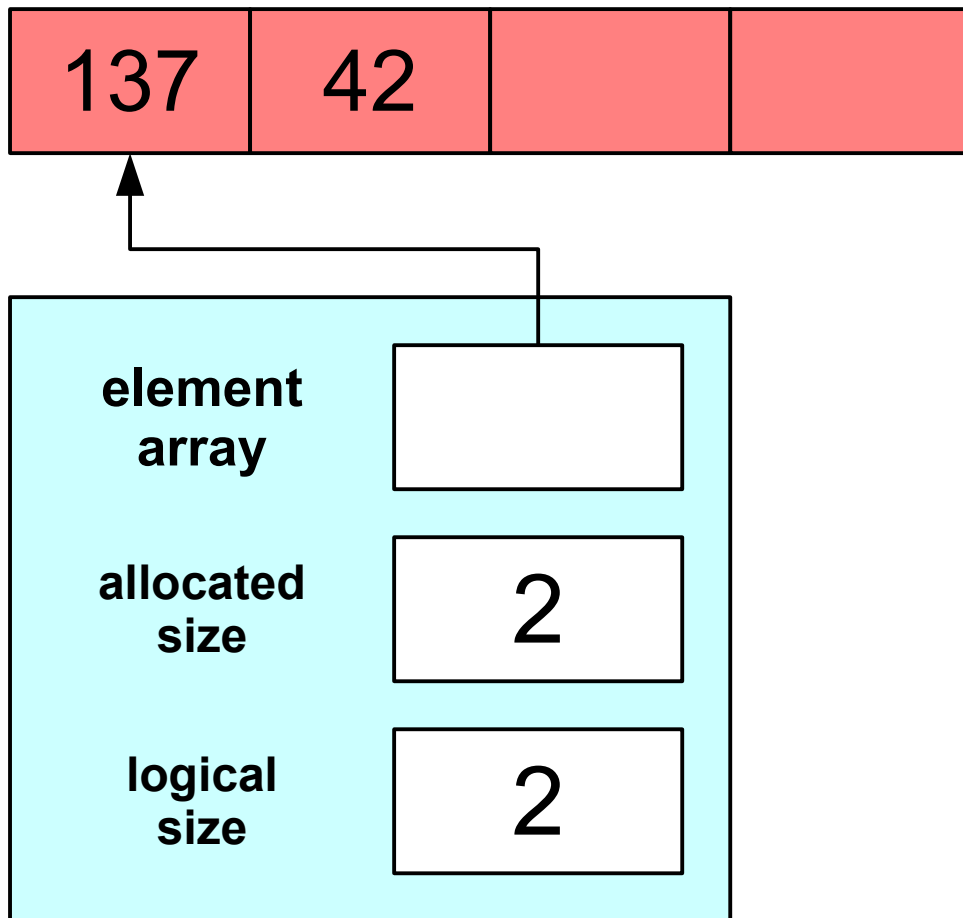
# A Much Better Idea



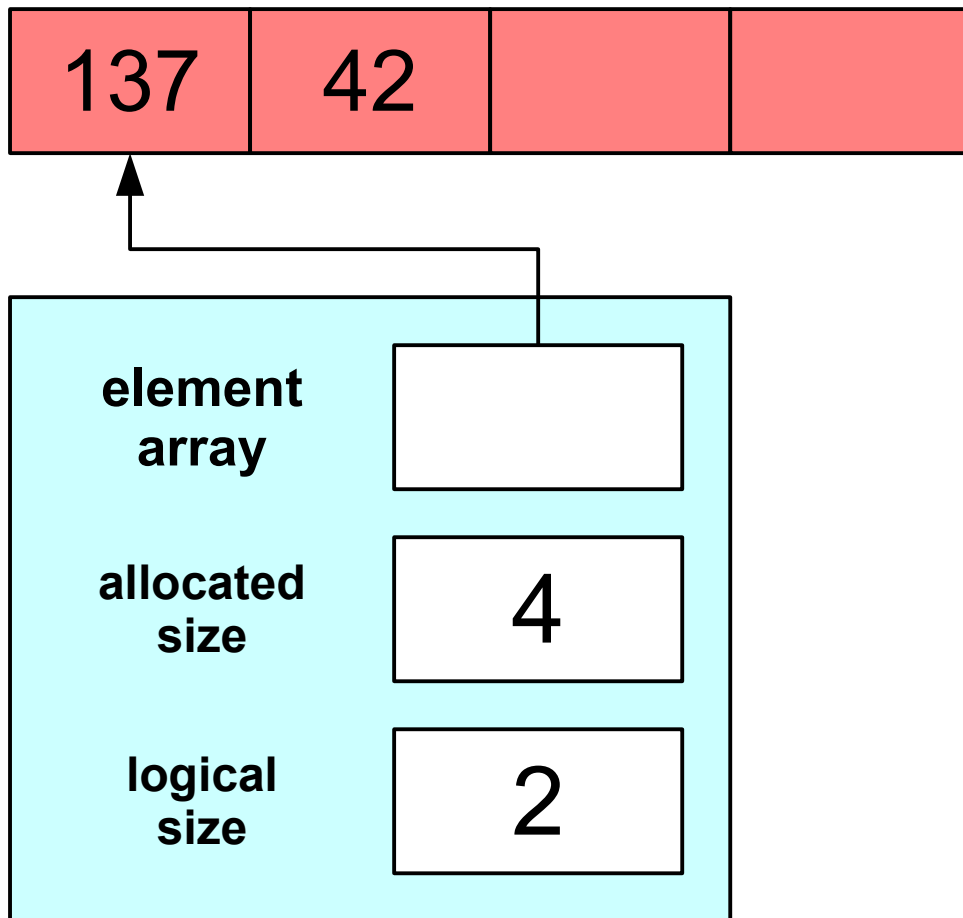
# A Much Better Idea



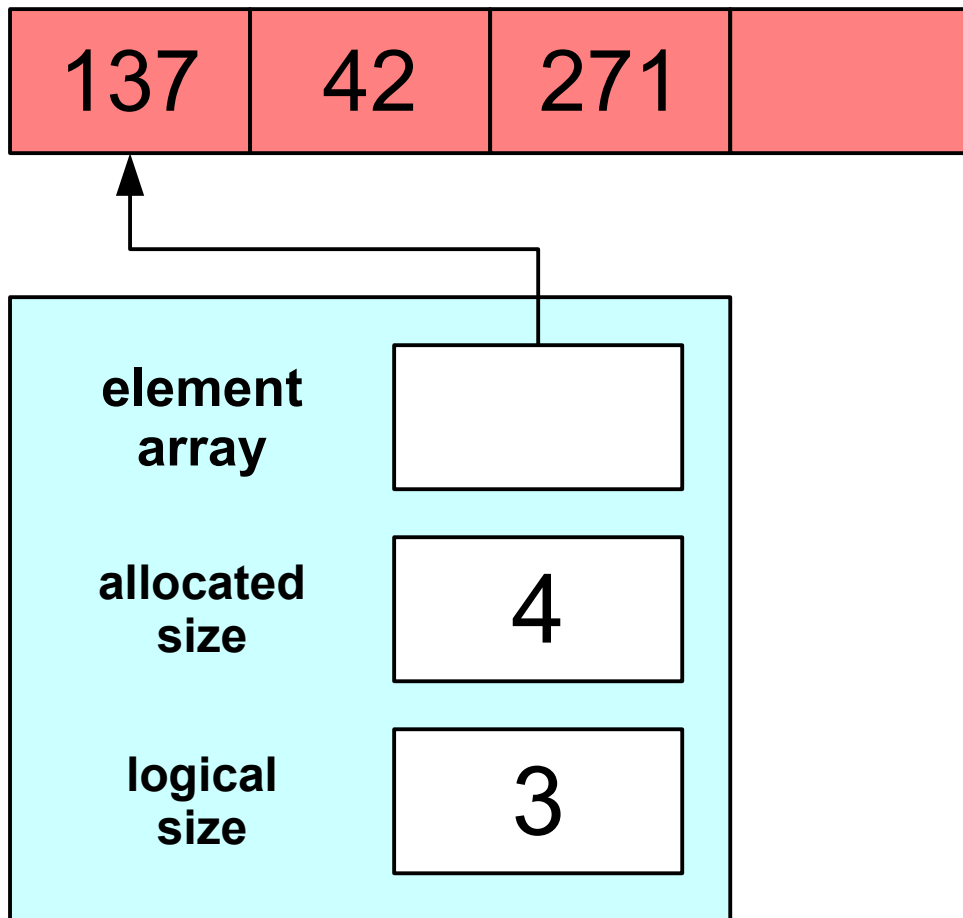
# A Much Better Idea



# A Much Better Idea

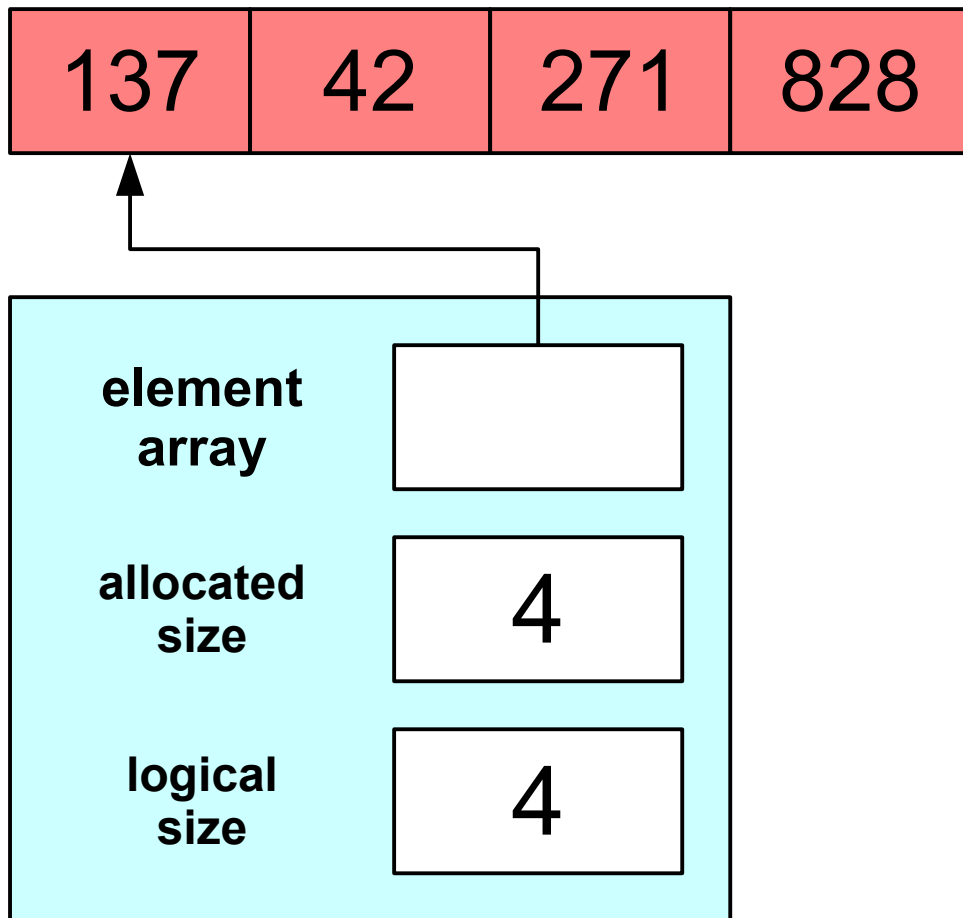


# A Much Better Idea

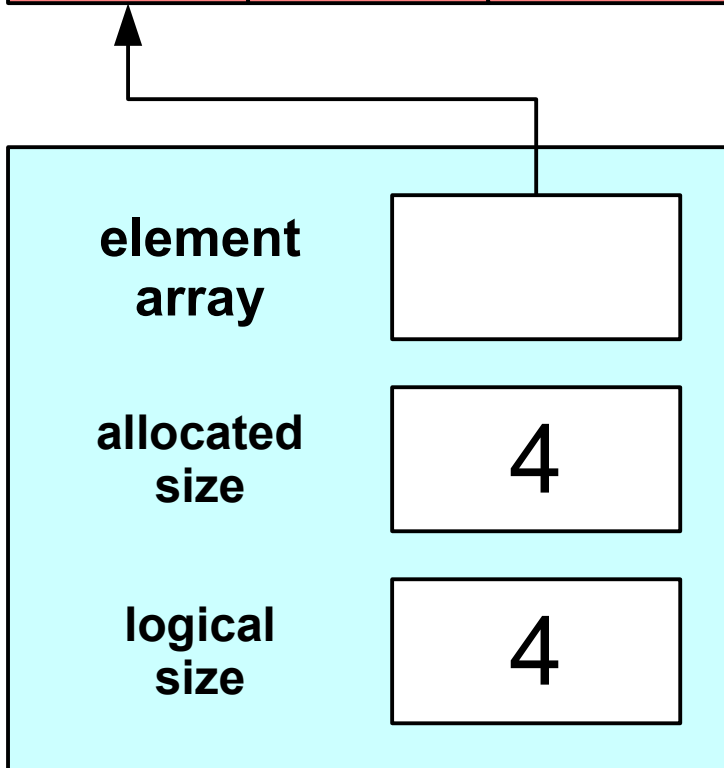
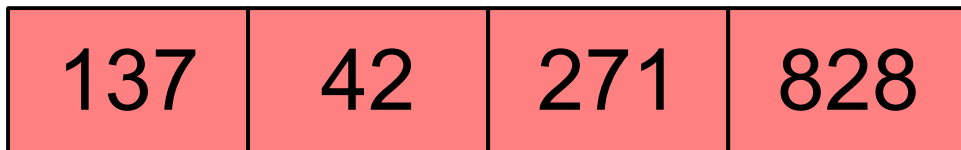
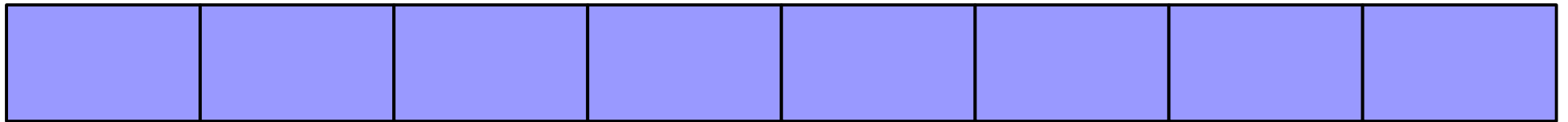




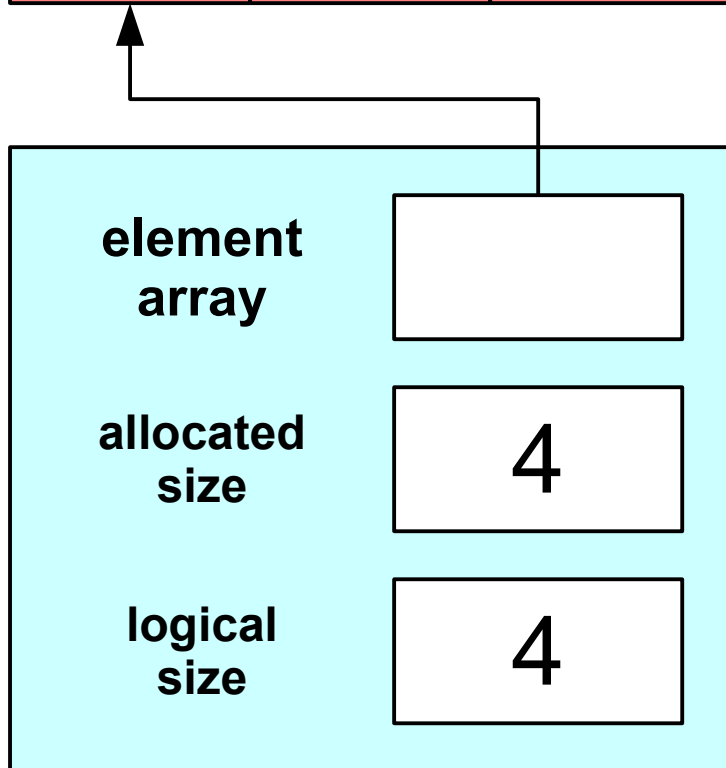
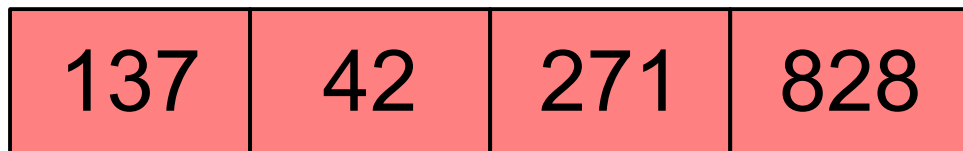
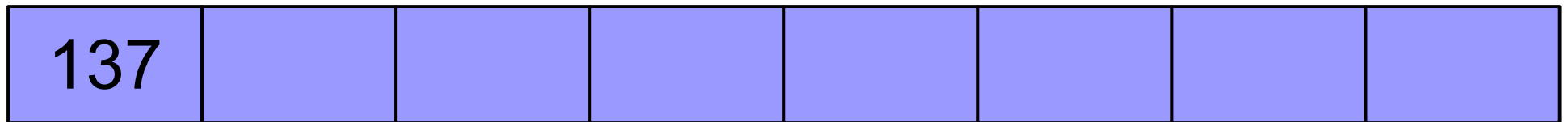
# A Much Better Idea



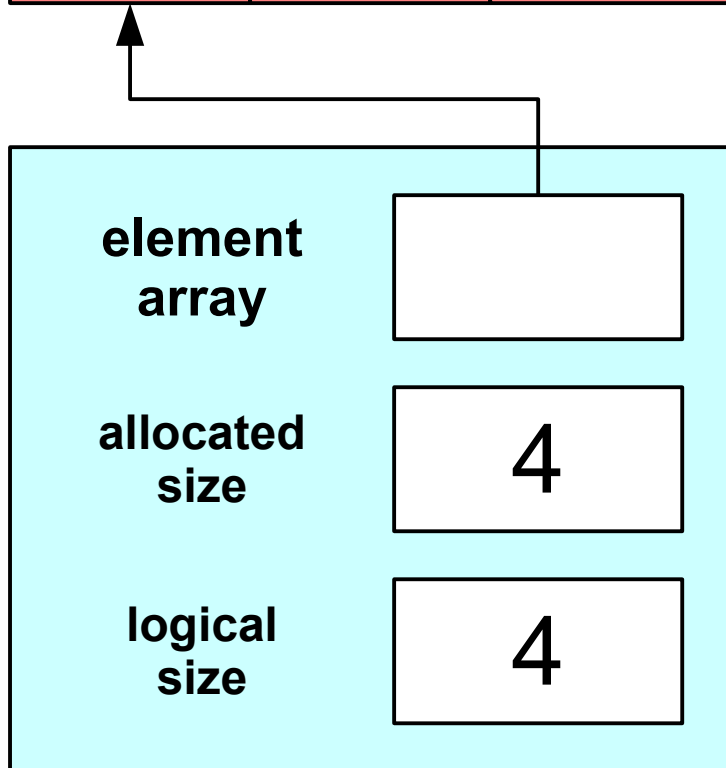
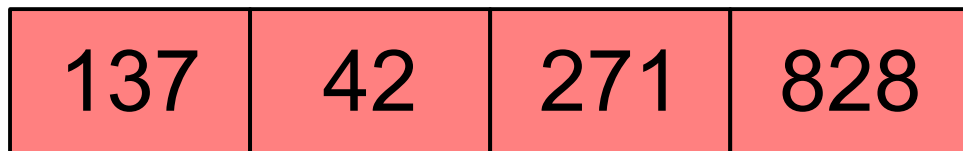
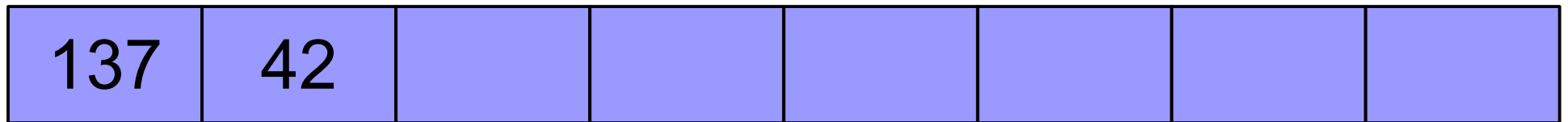
# A Much Better Idea



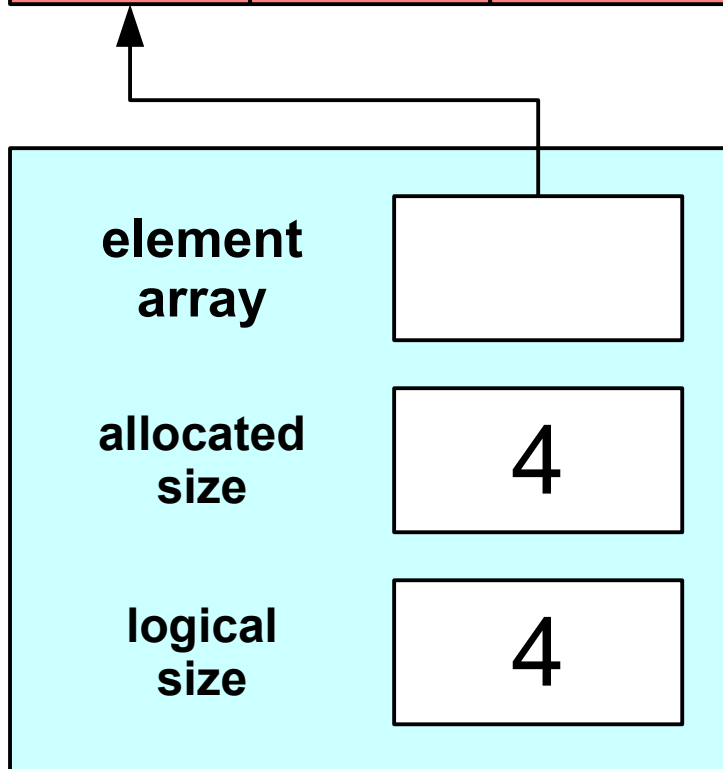
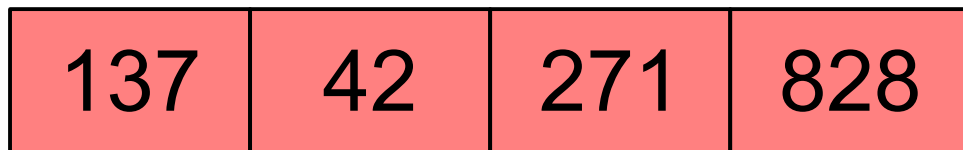
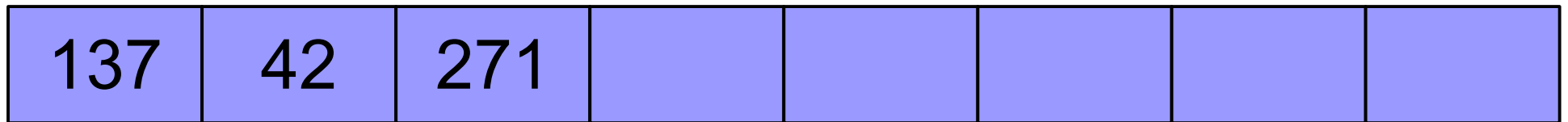
# A Much Better Idea



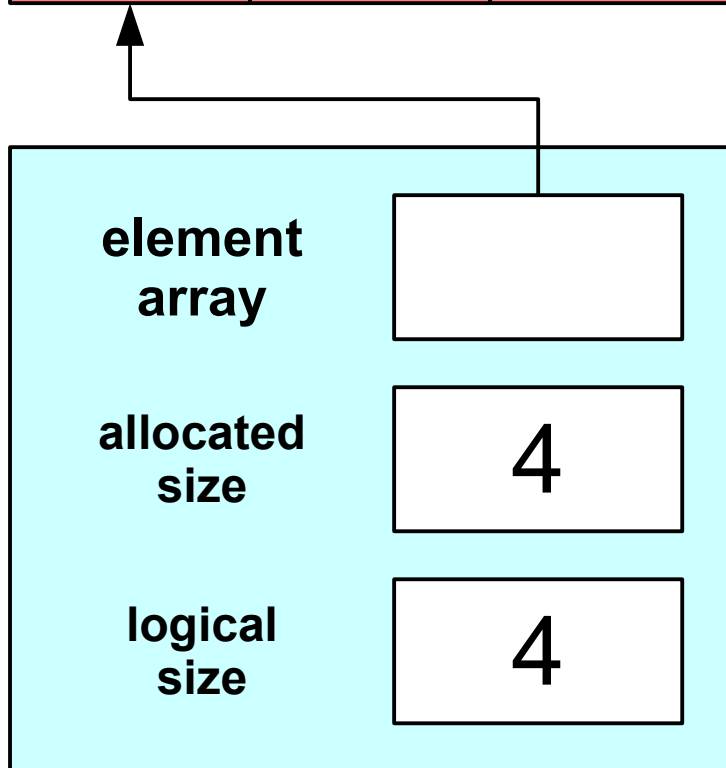
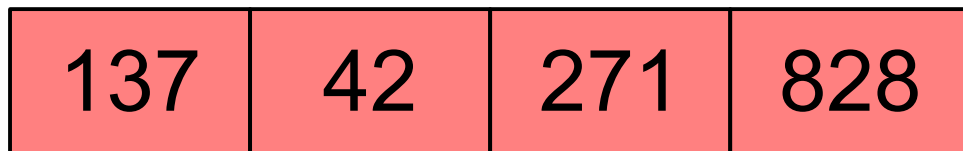
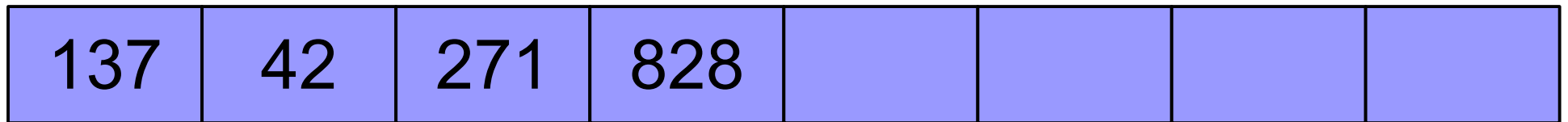
# A Much Better Idea



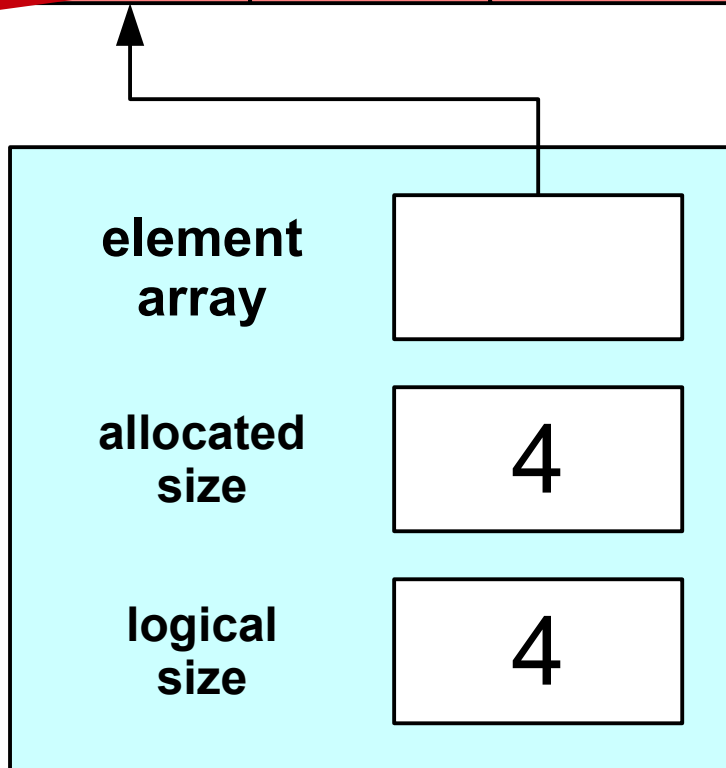
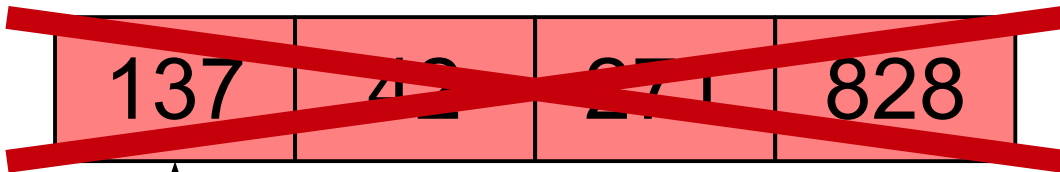
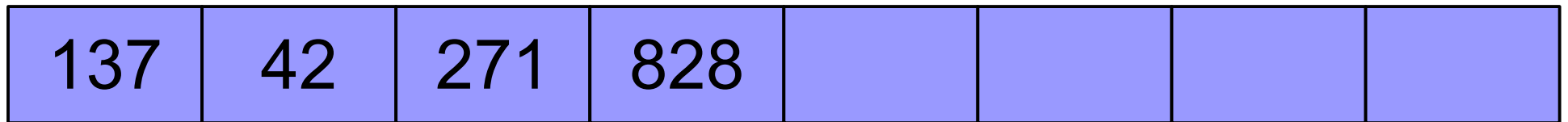
# A Much Better Idea



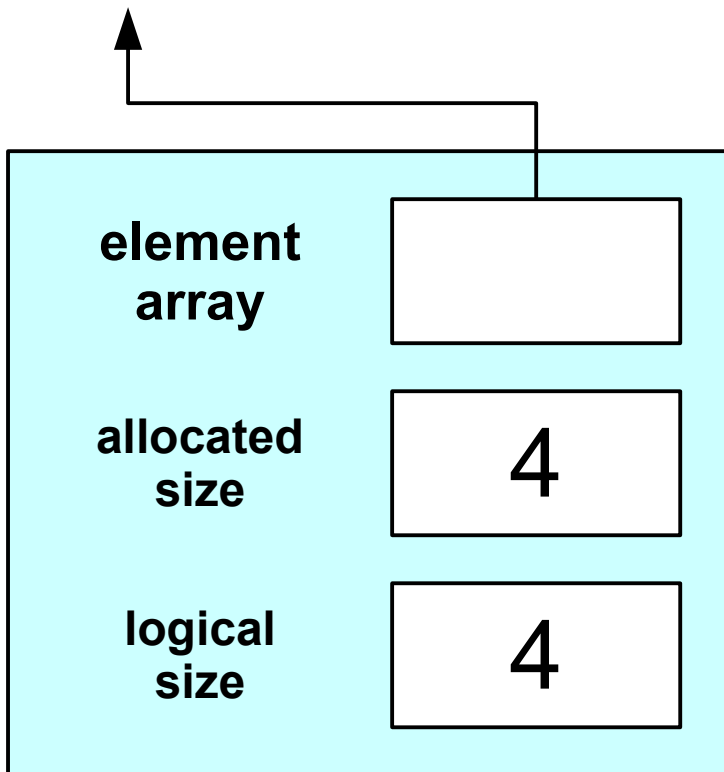
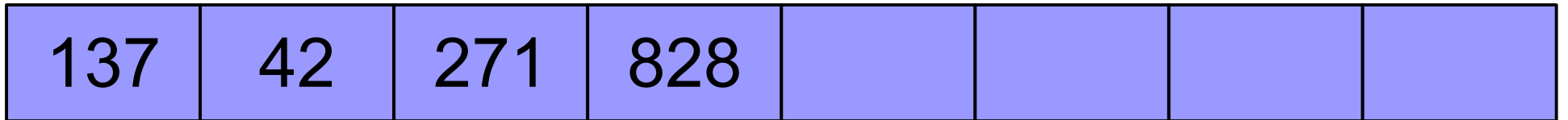
# A Much Better Idea



# A Much Better Idea

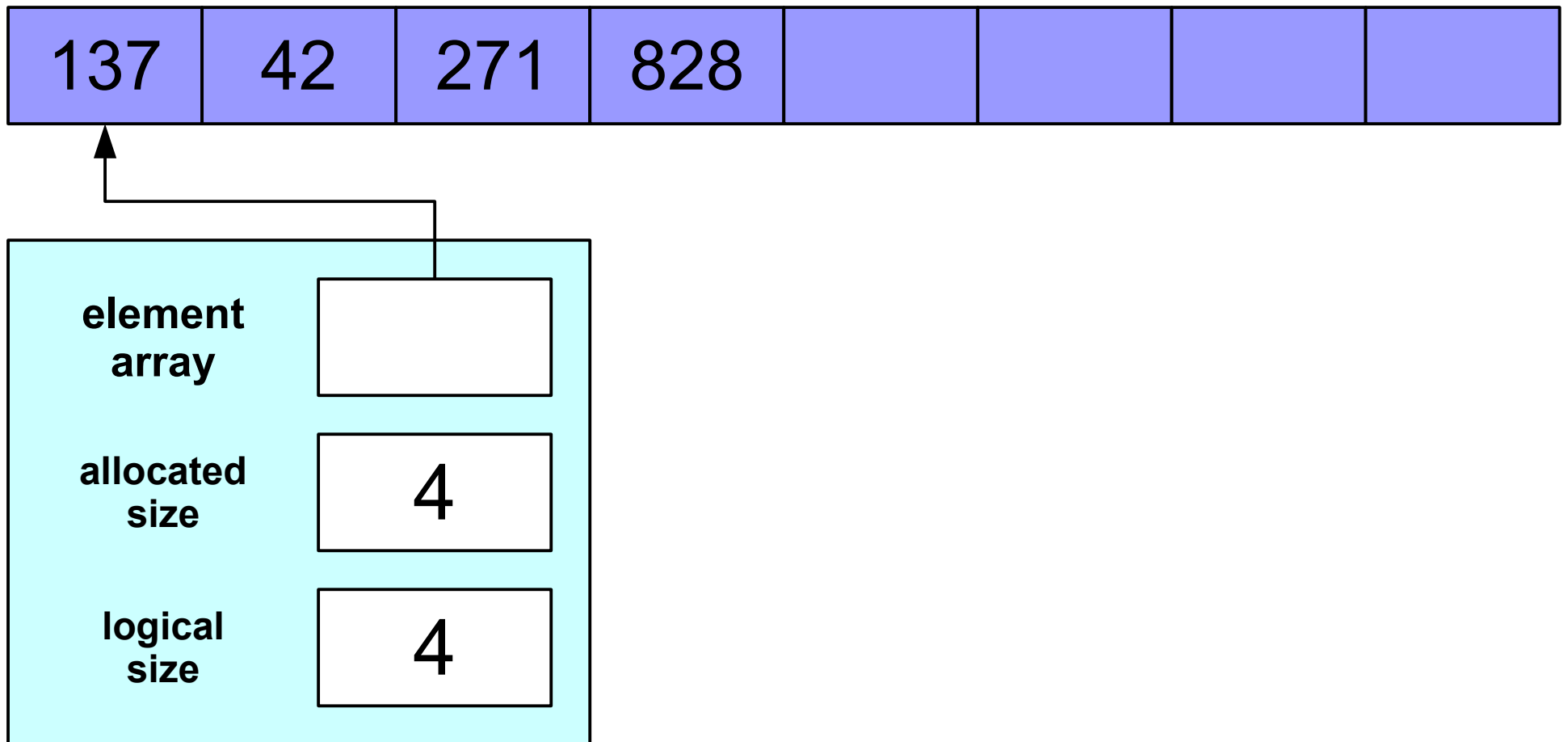


# A Much Better Idea

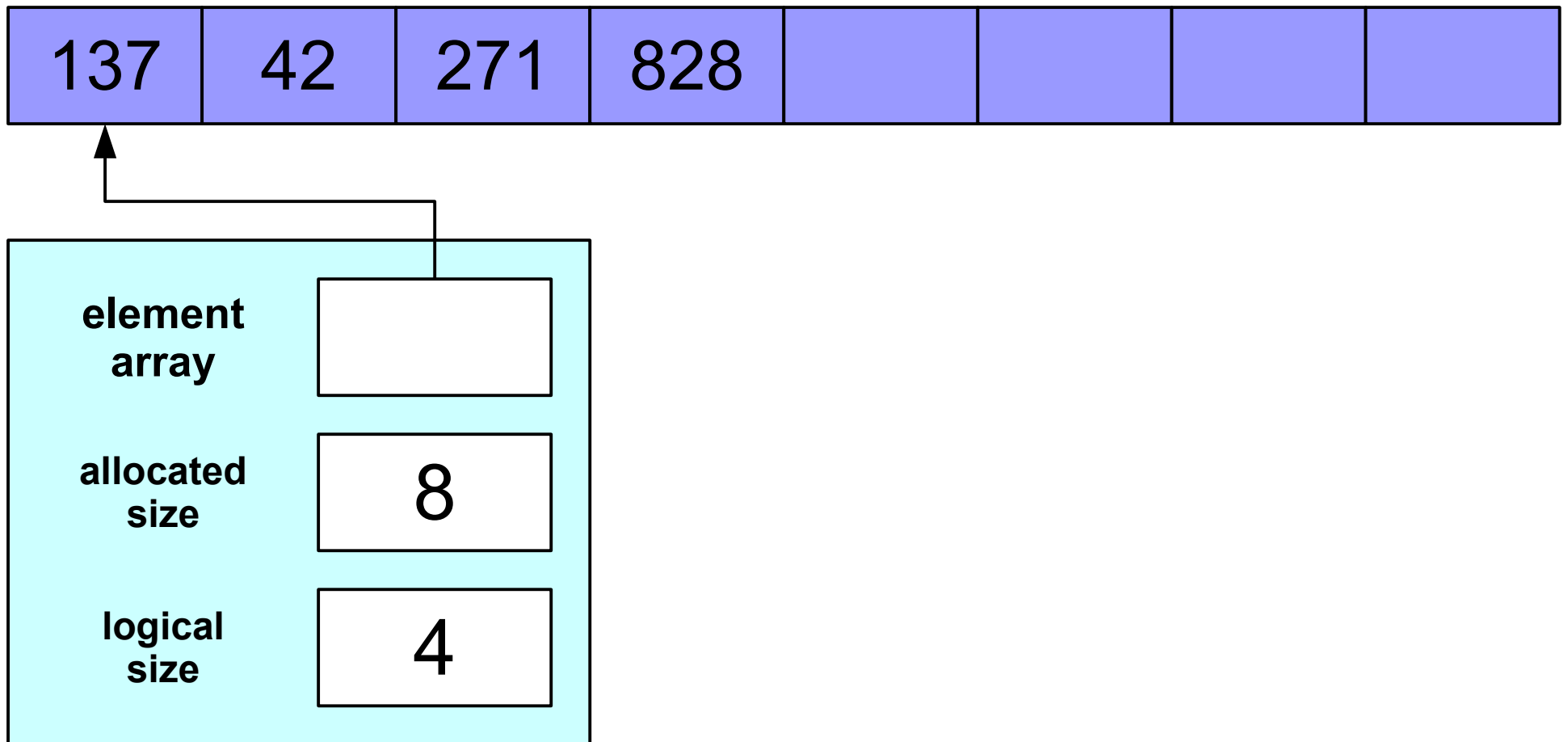




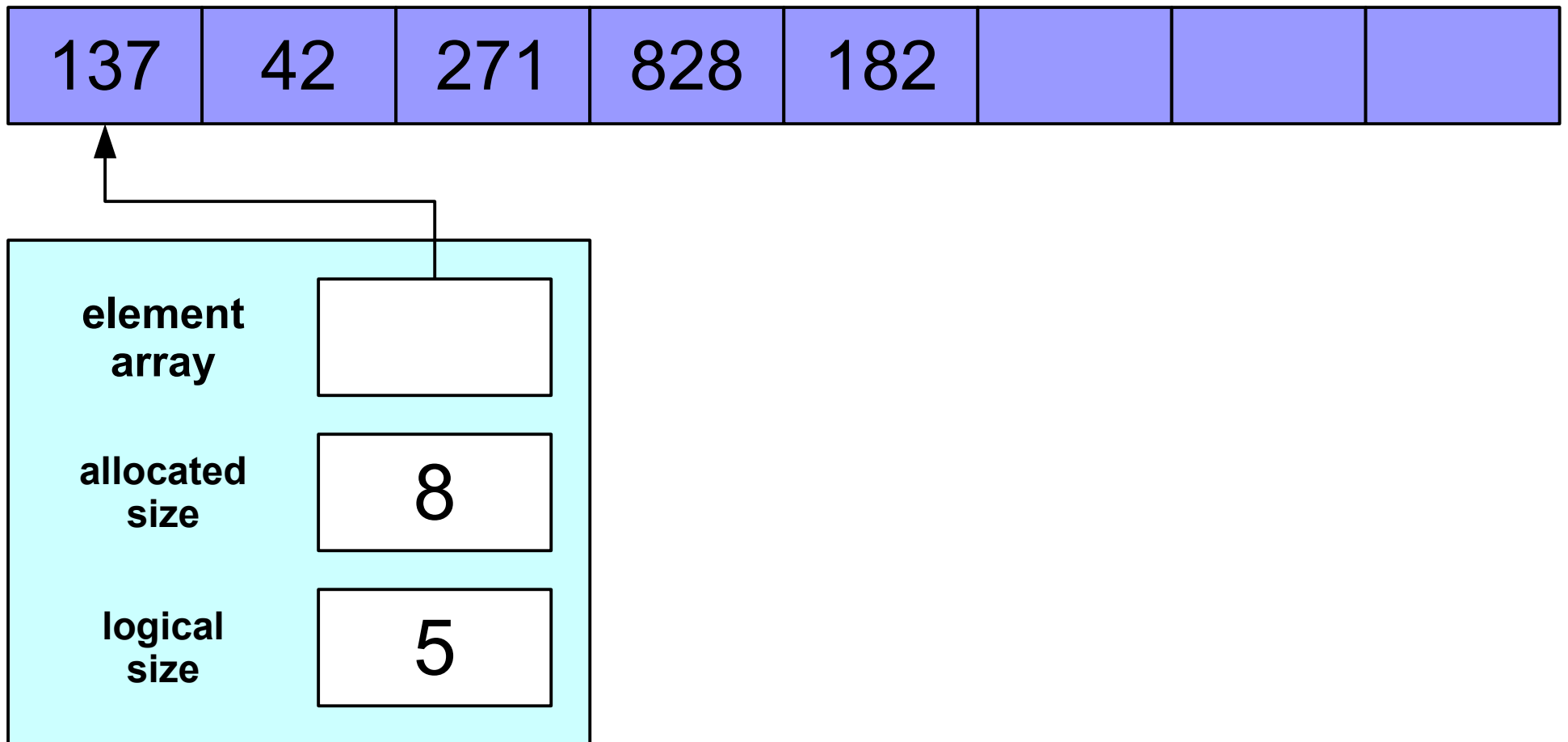
# A Much Better Idea



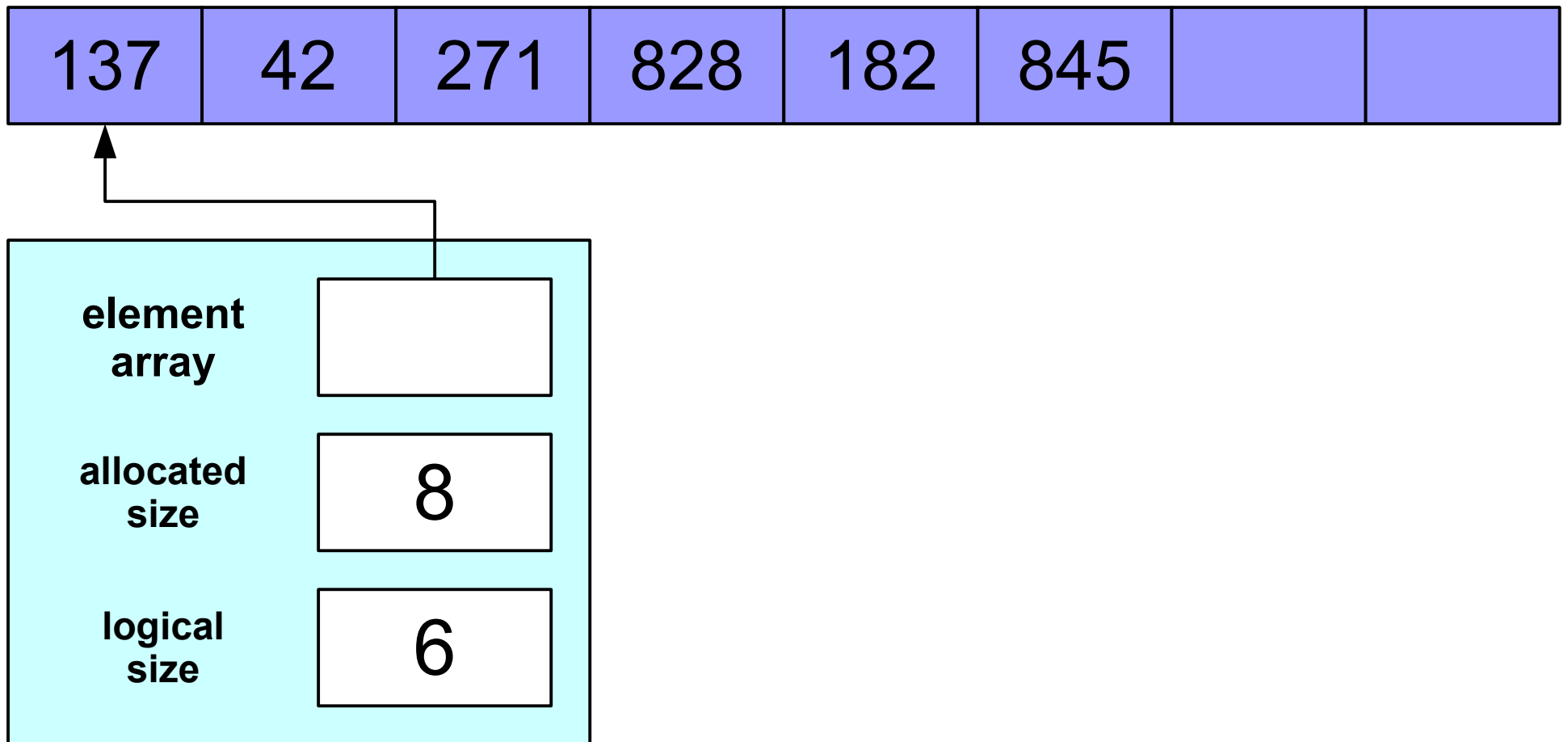
# A Much Better Idea



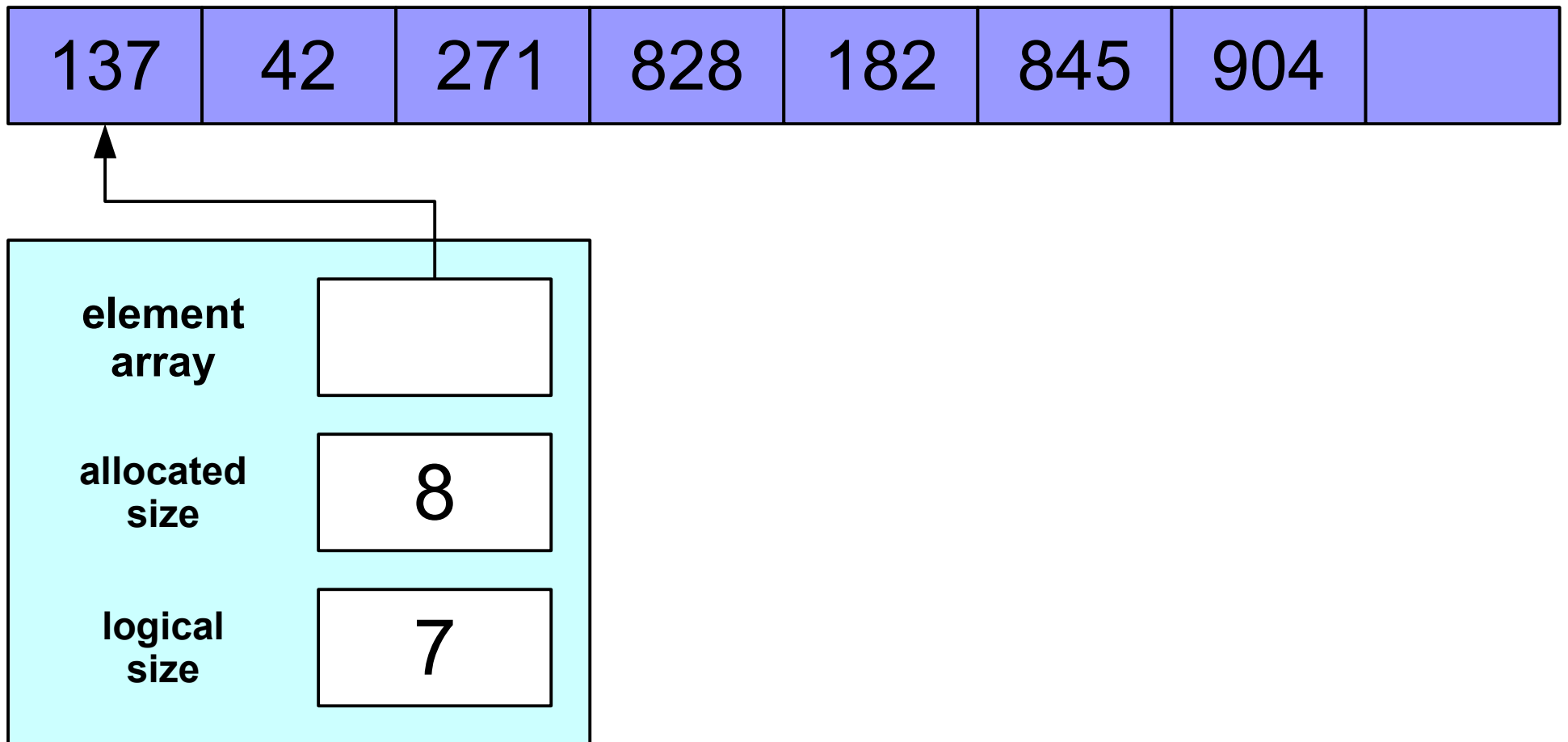
# A Much Better Idea



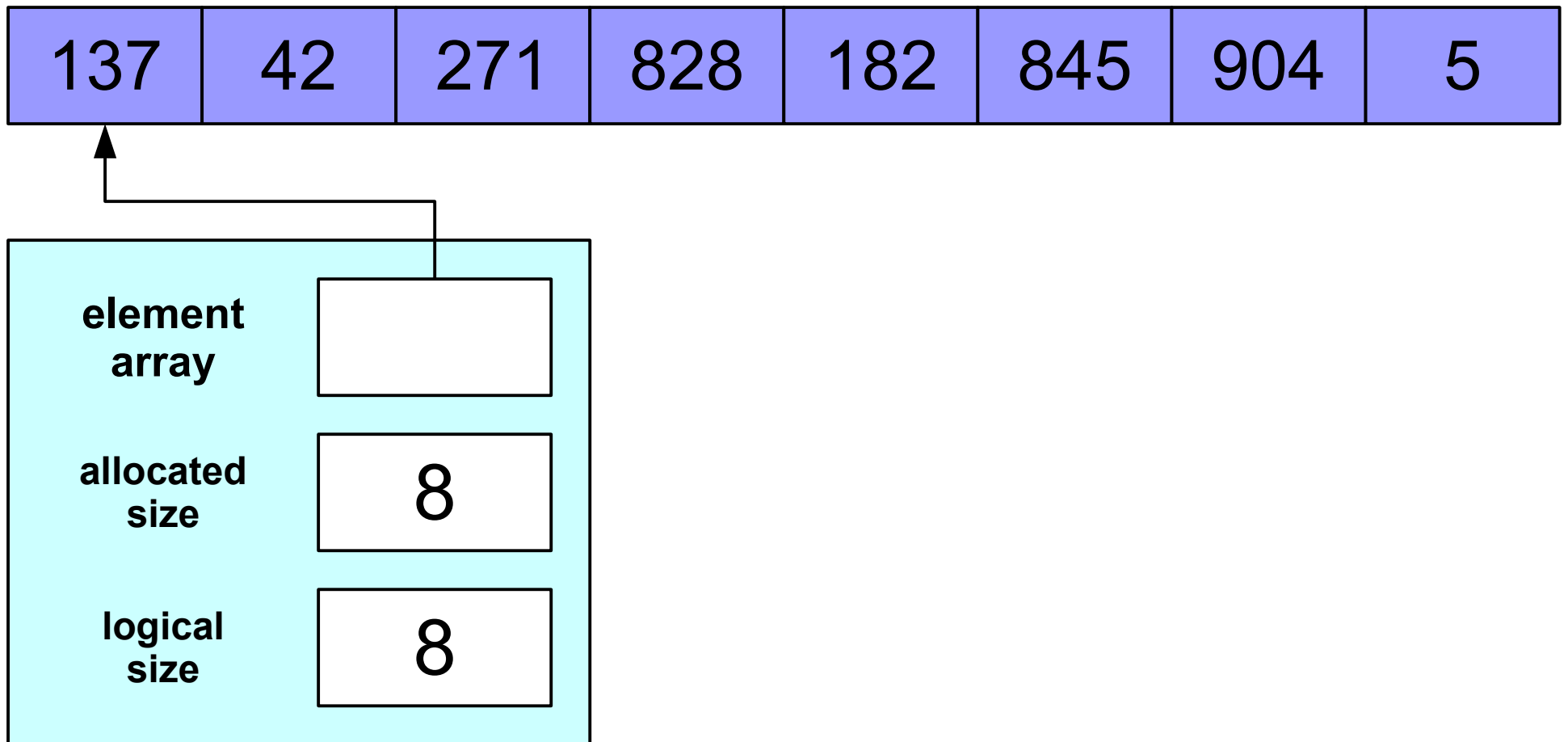
# A Much Better Idea



# A Much Better Idea



# A Much Better Idea

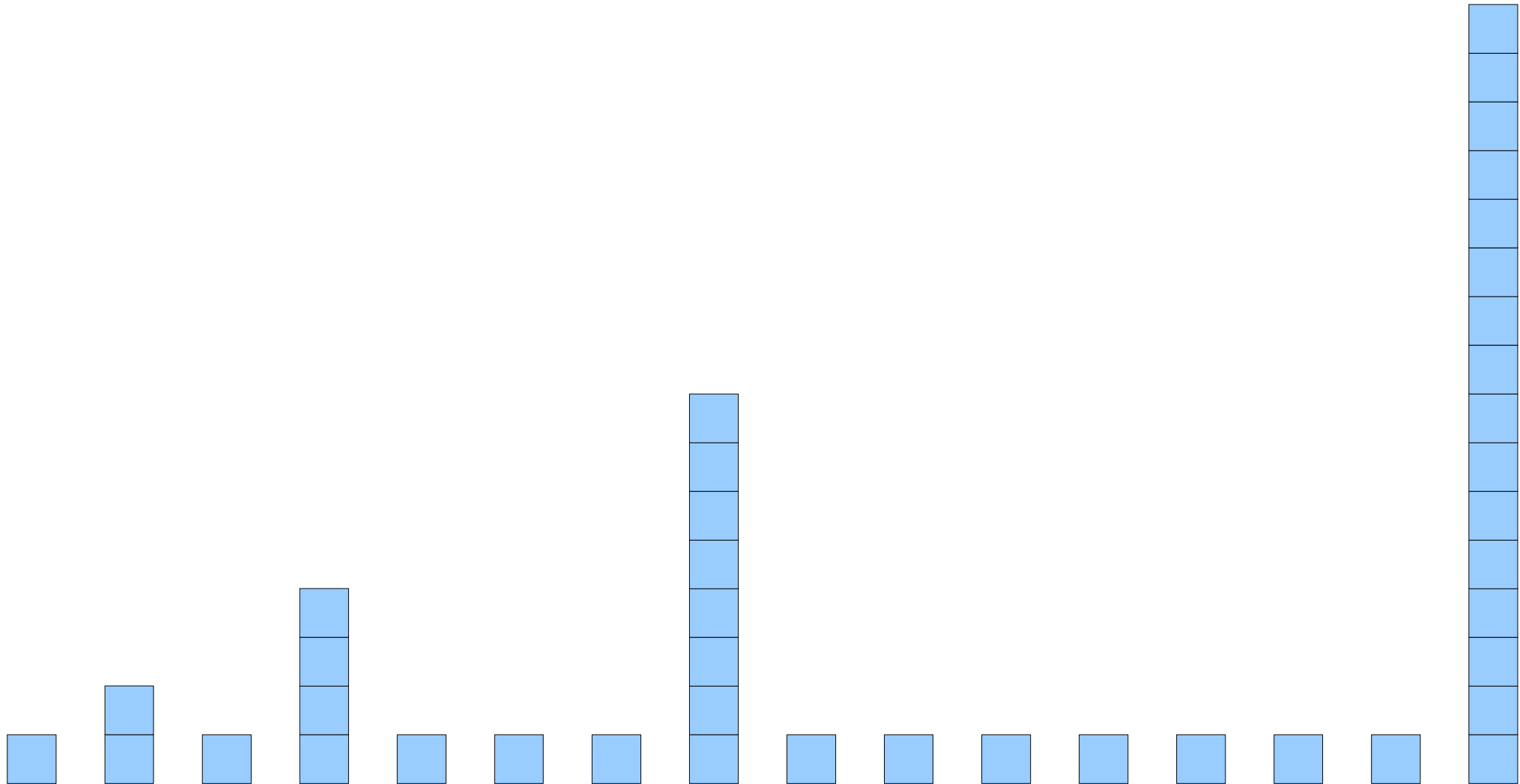


Let's Give it a Try!

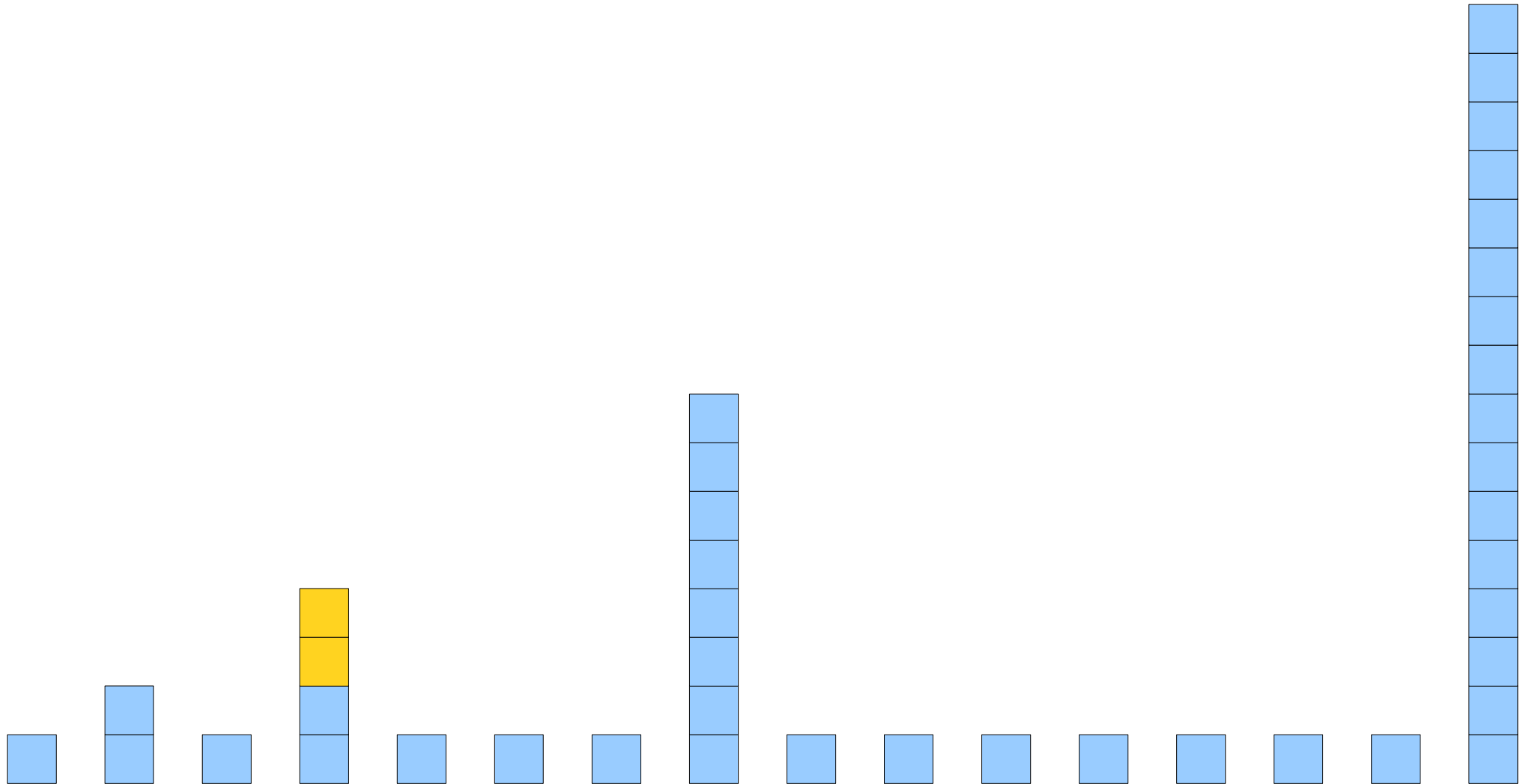
How do we analyze this?



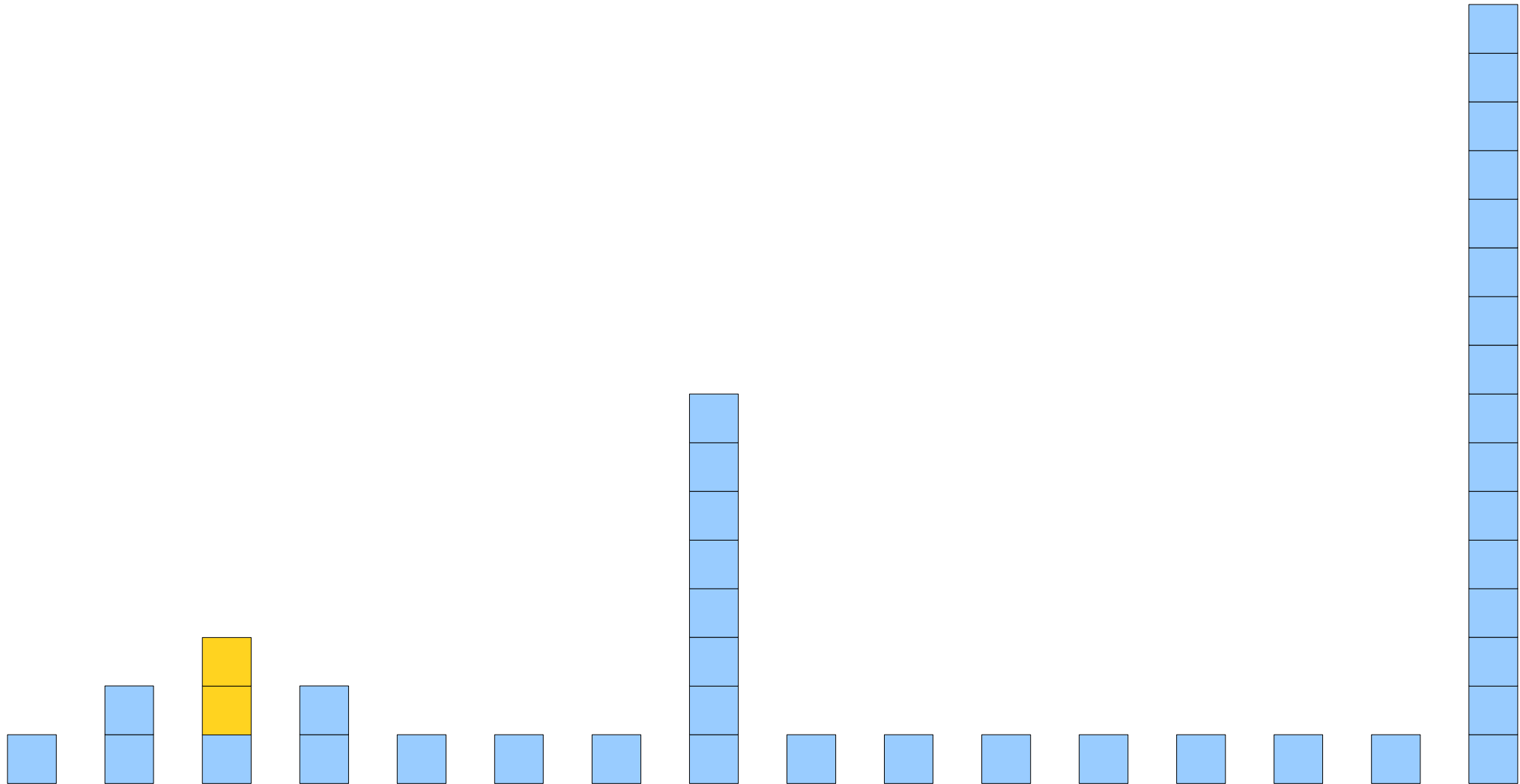
# Spreading the Work



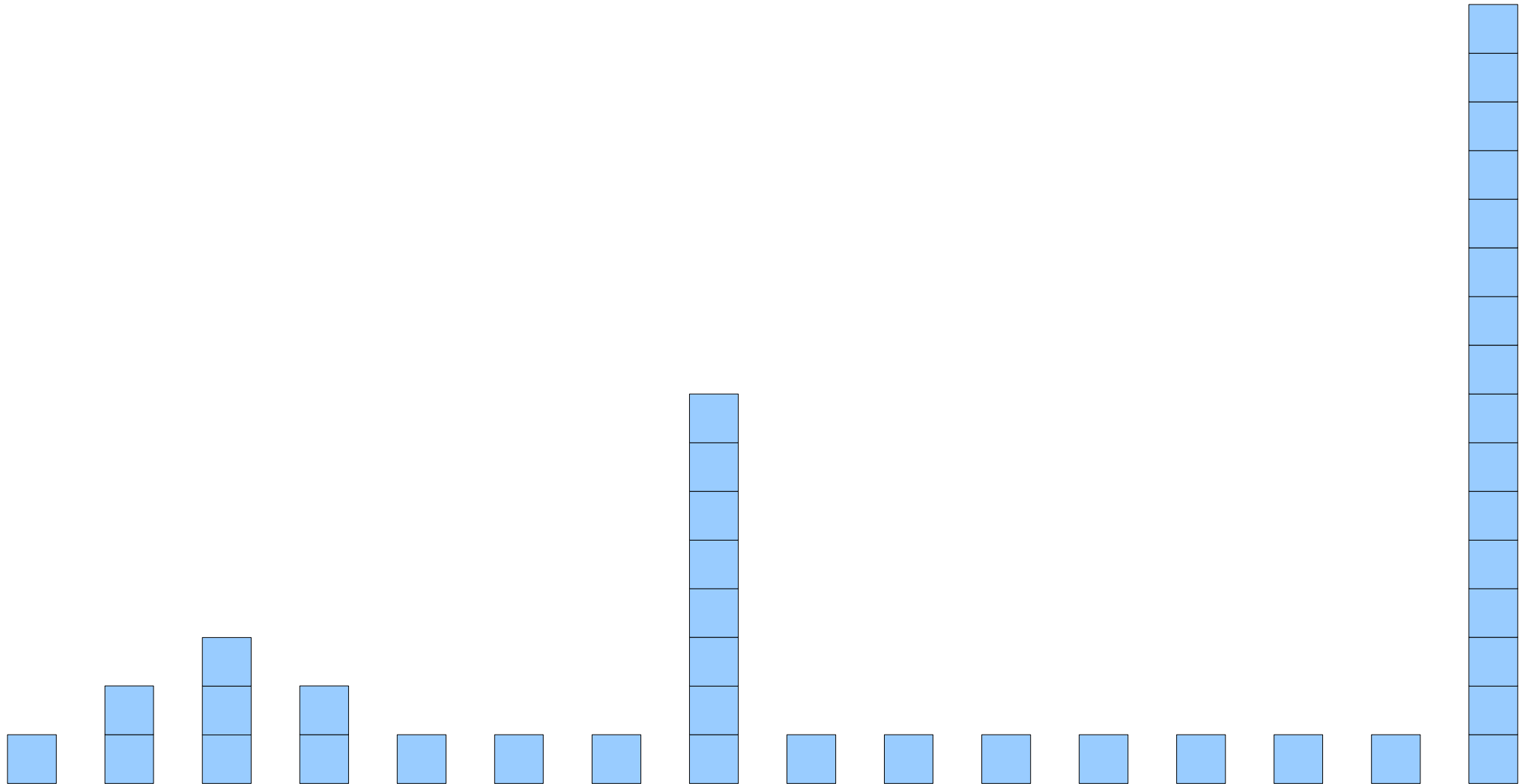
# Spreading the Work



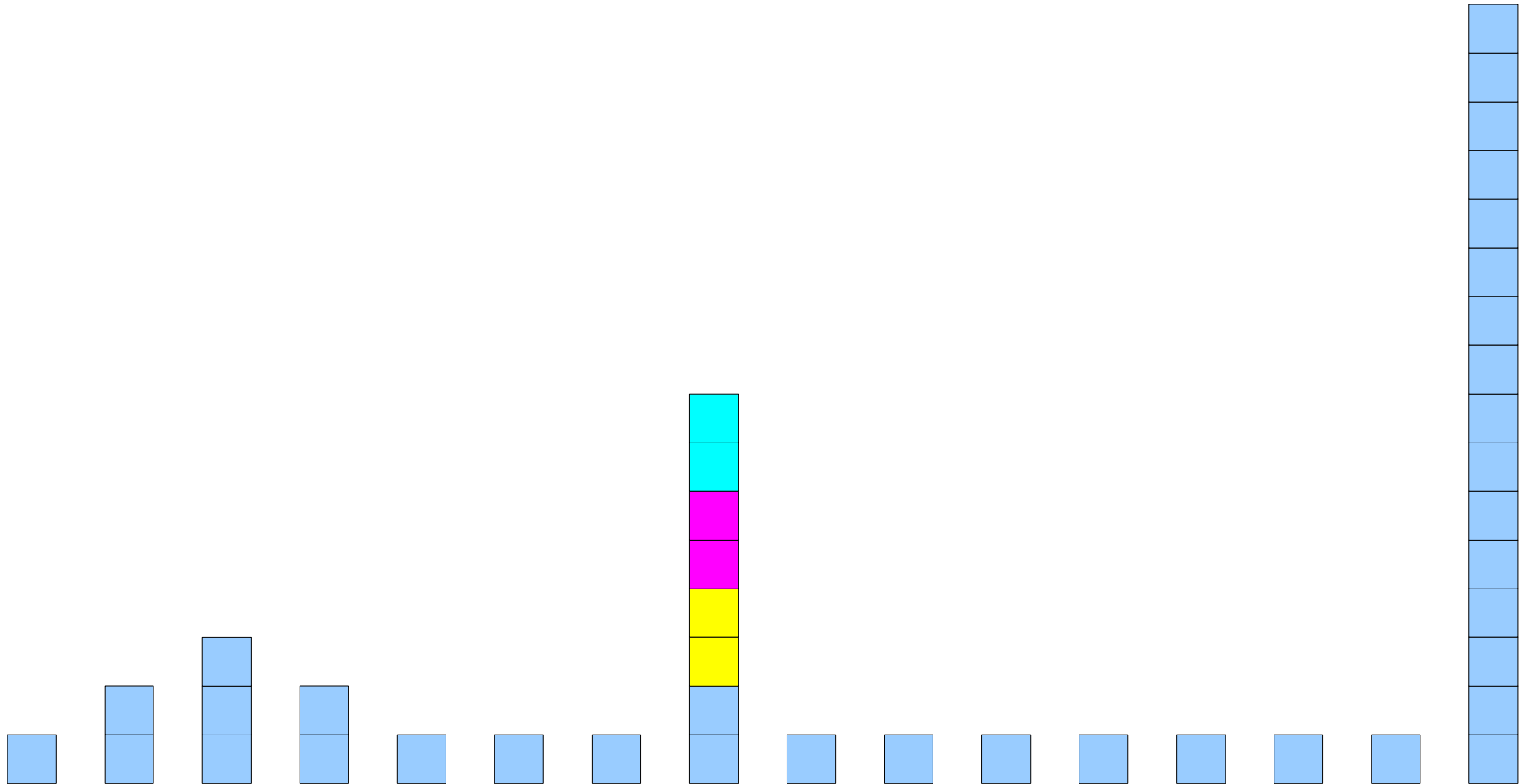
# Spreading the Work



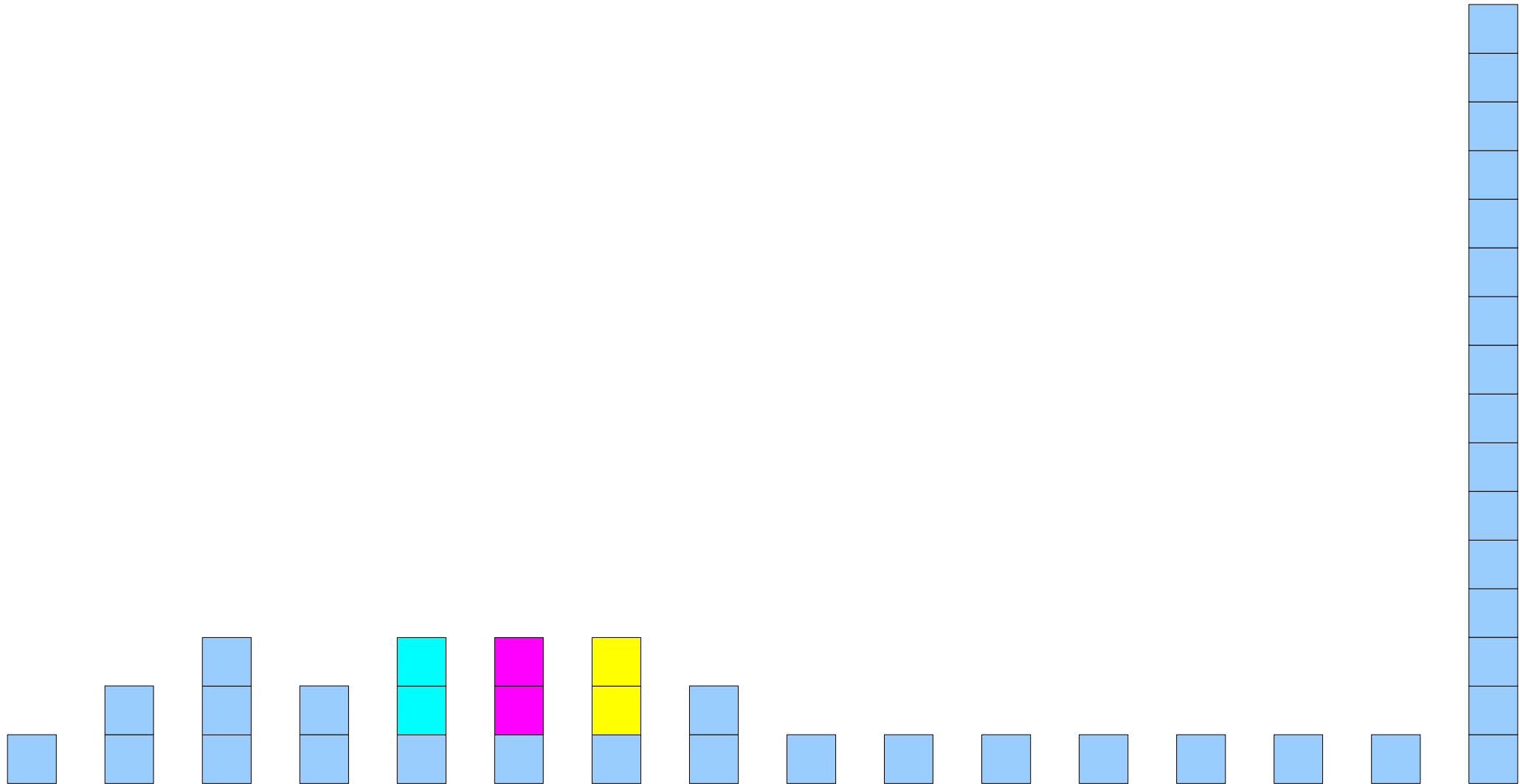
# Spreading the Work



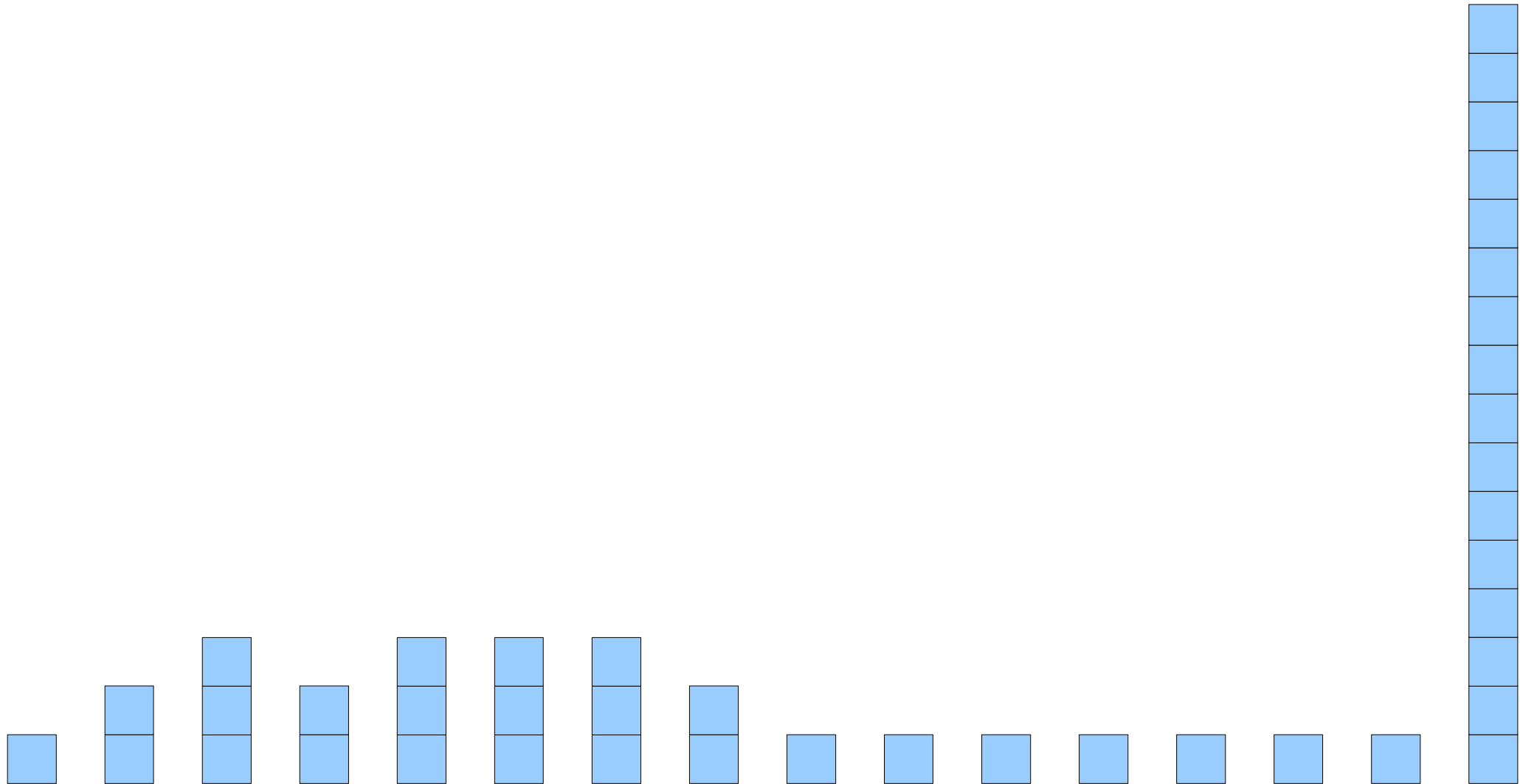
# Spreading the Work



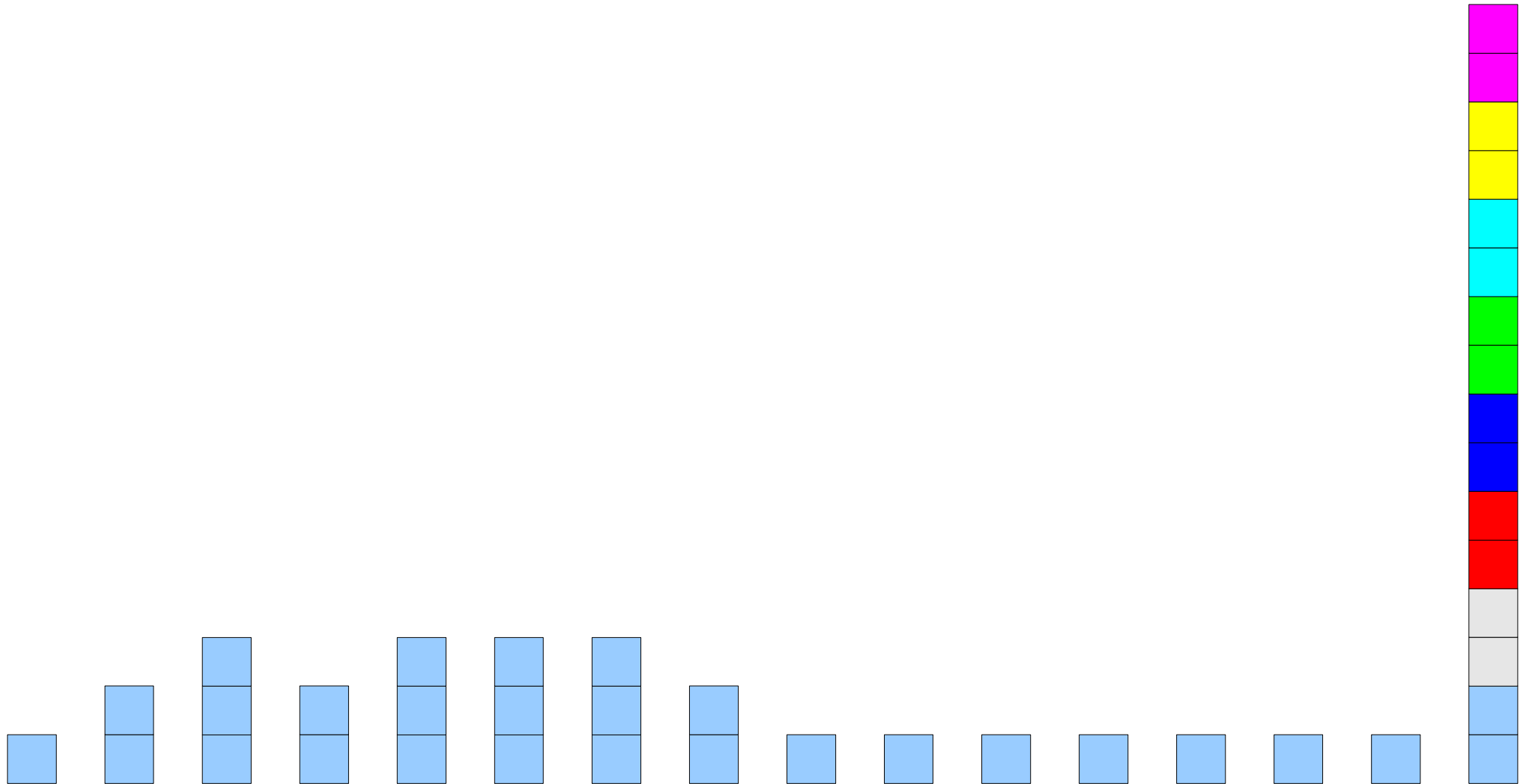
# Spreading the Work



# Spreading the Work

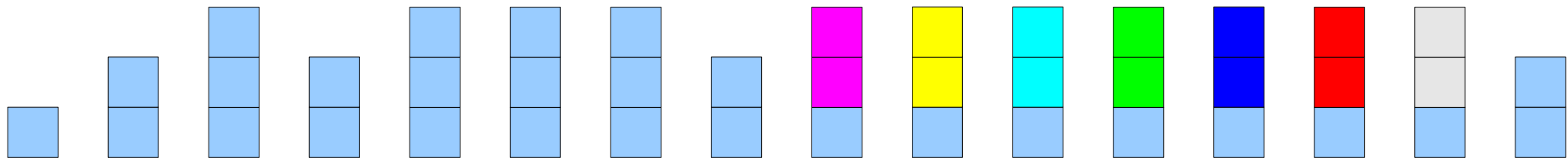


# Spreading the Work

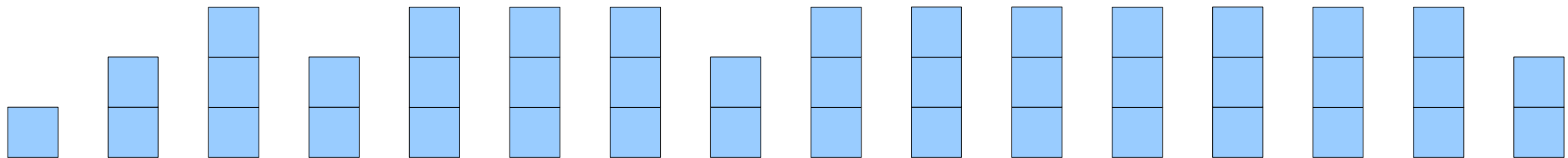




# Spreading the Work



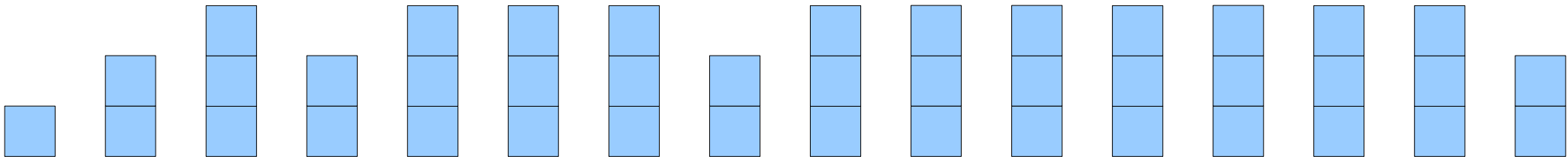
# Spreading the Work



# 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.

# 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



**Out**




**In**



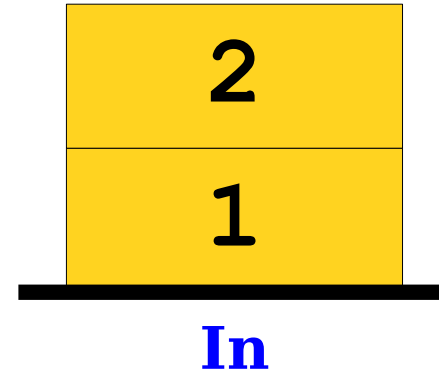
# The Two-Stack Queue

  
**Out**

  
**In**

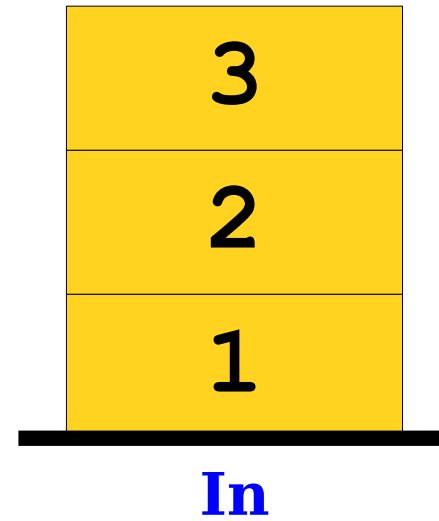
# The Two-Stack Queue

**Out**

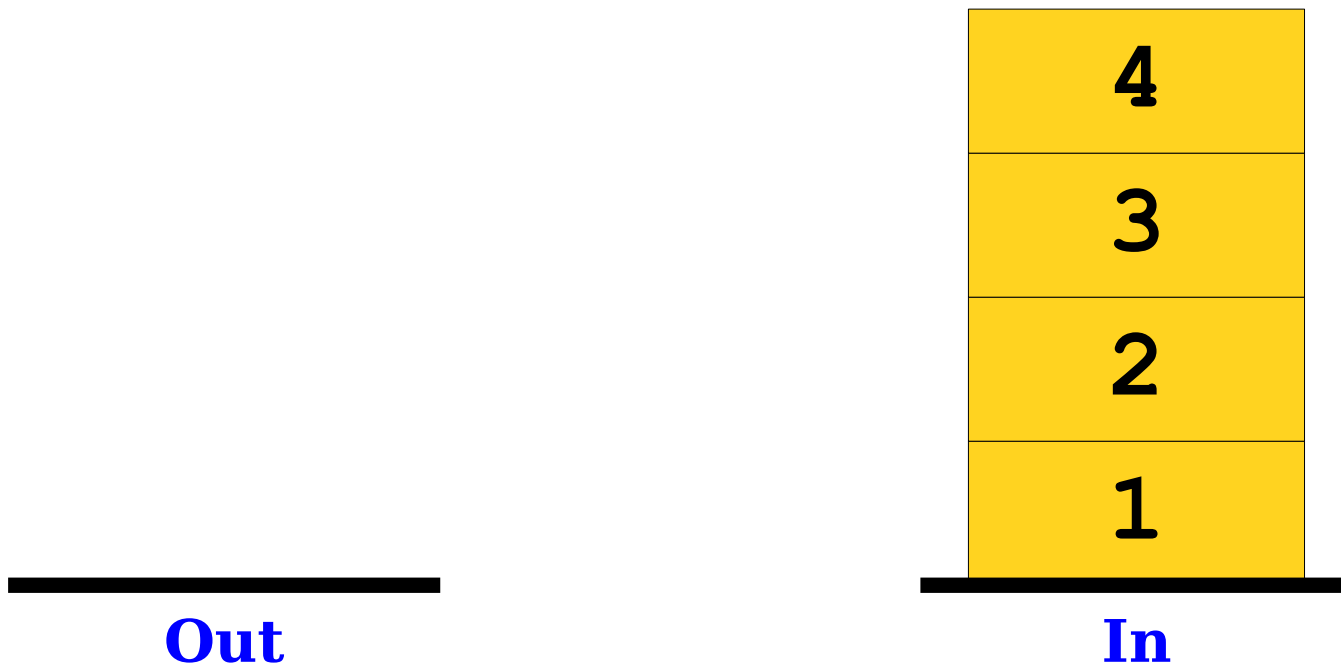


# The Two-Stack Queue

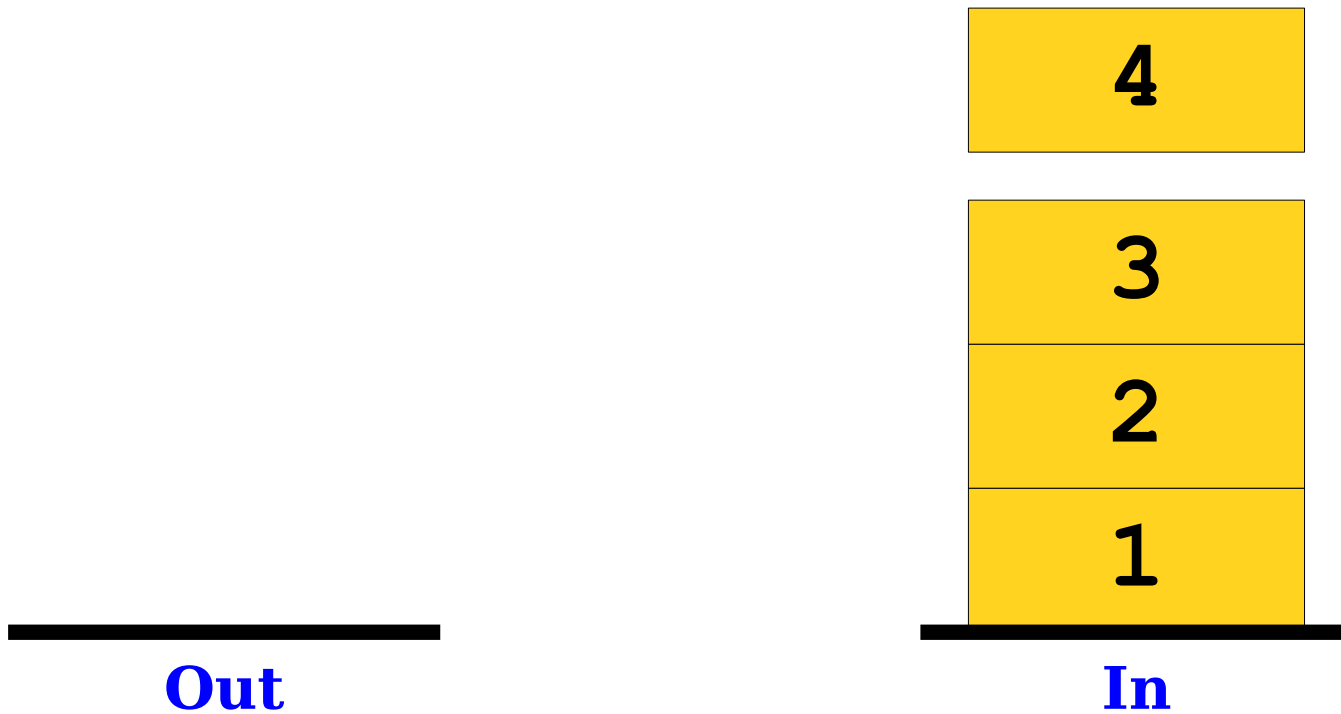
**Out**



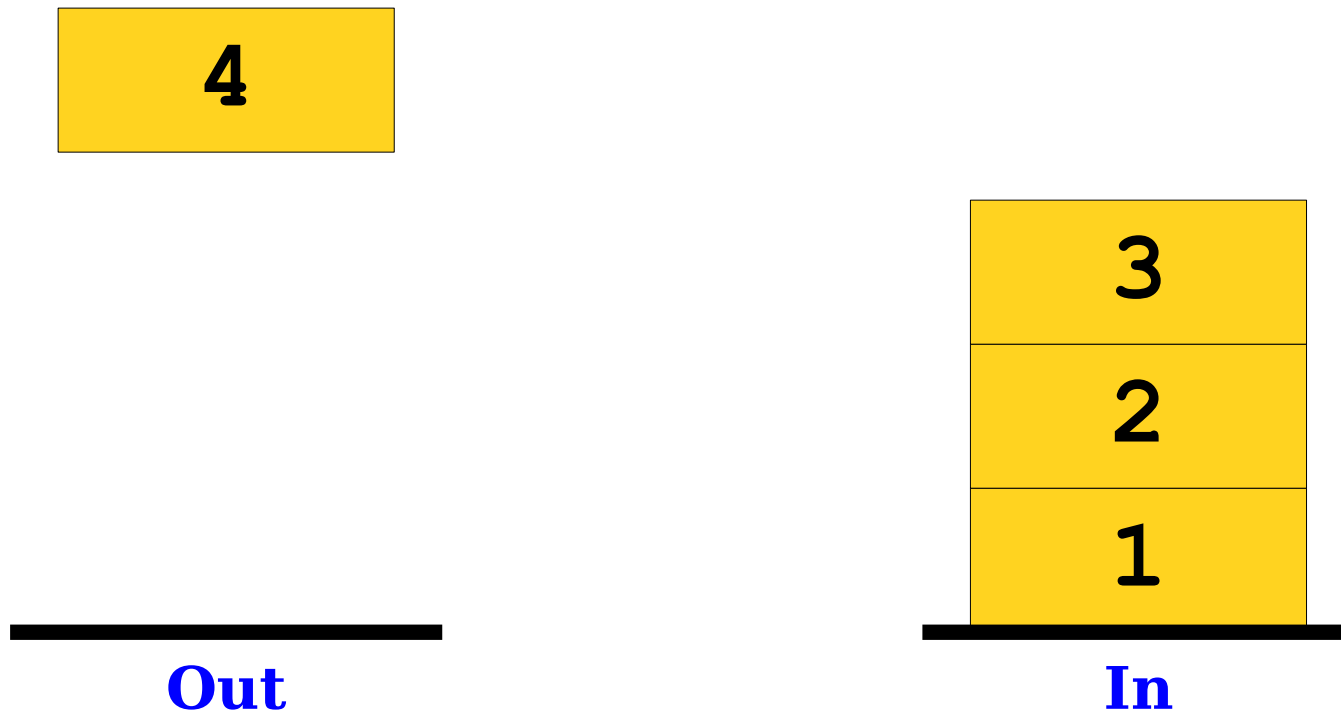
# The Two-Stack Queue



# The Two-Stack Queue



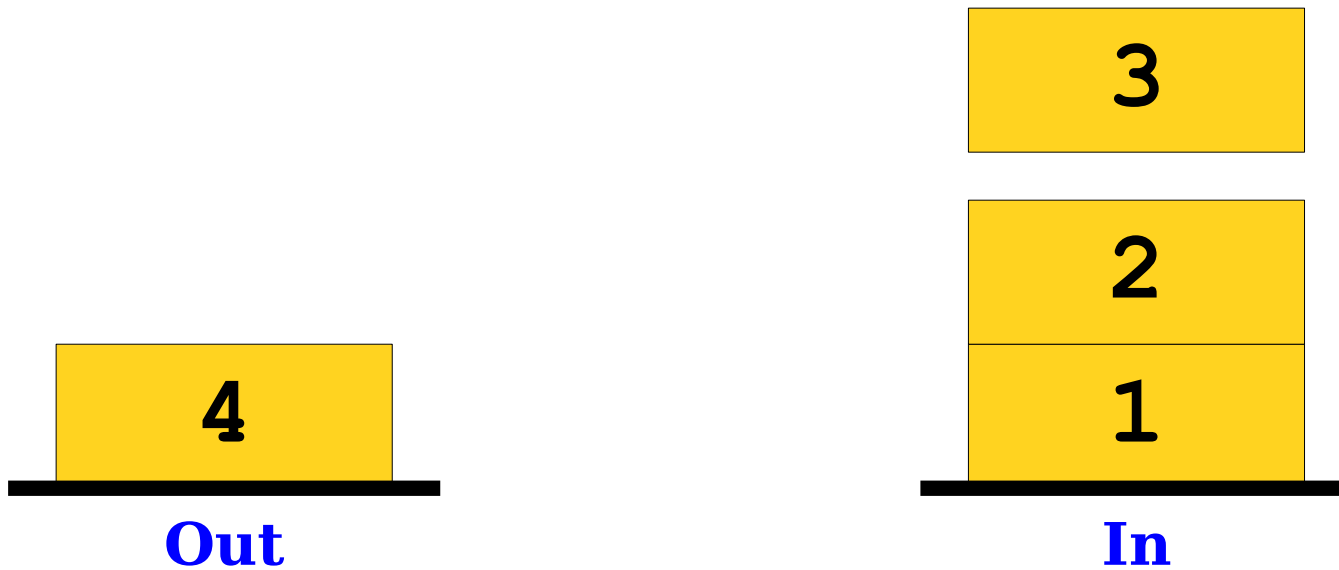
# The Two-Stack Queue



# The Two-Stack Queue

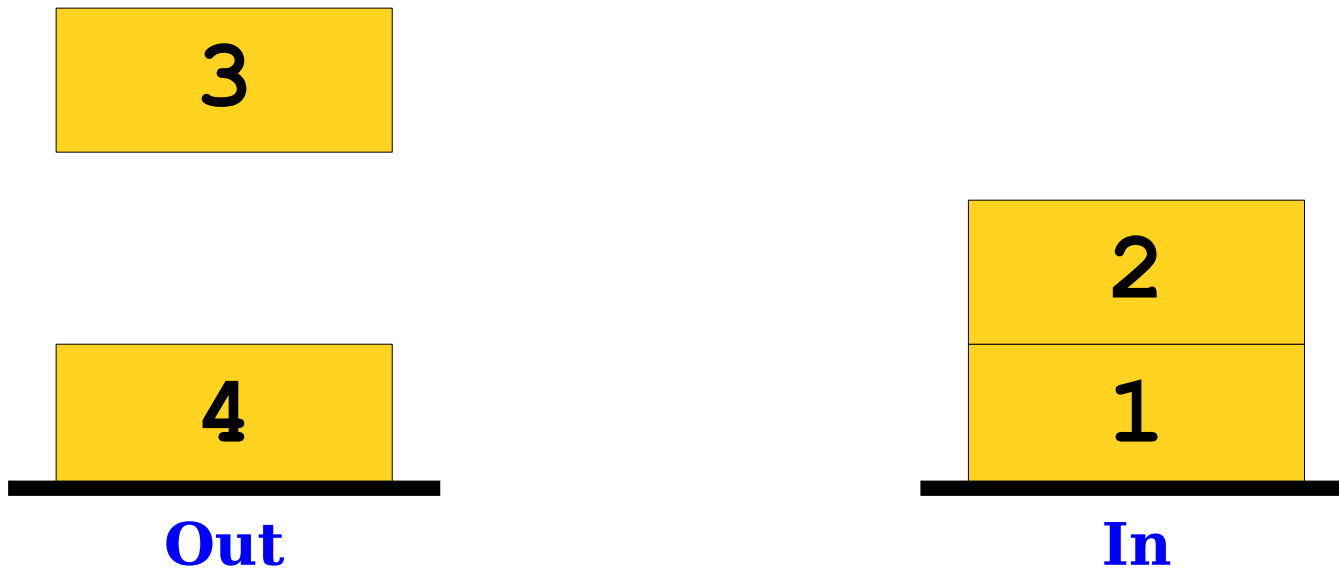


# The Two-Stack Queue





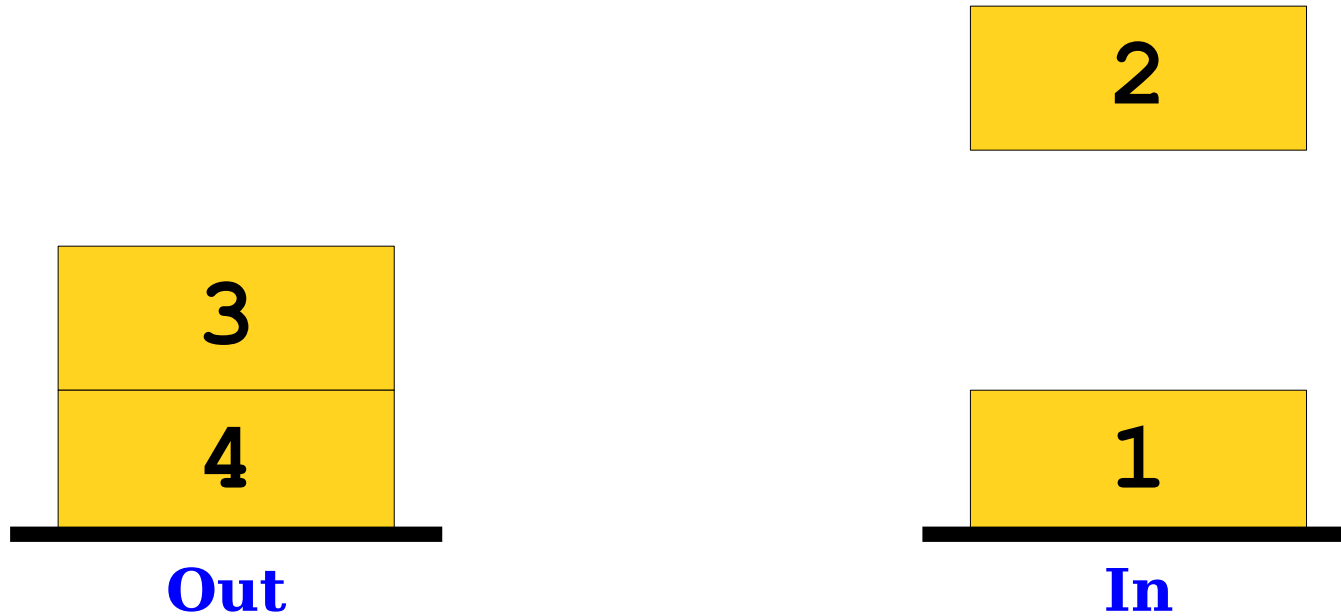
# The Two-Stack Queue



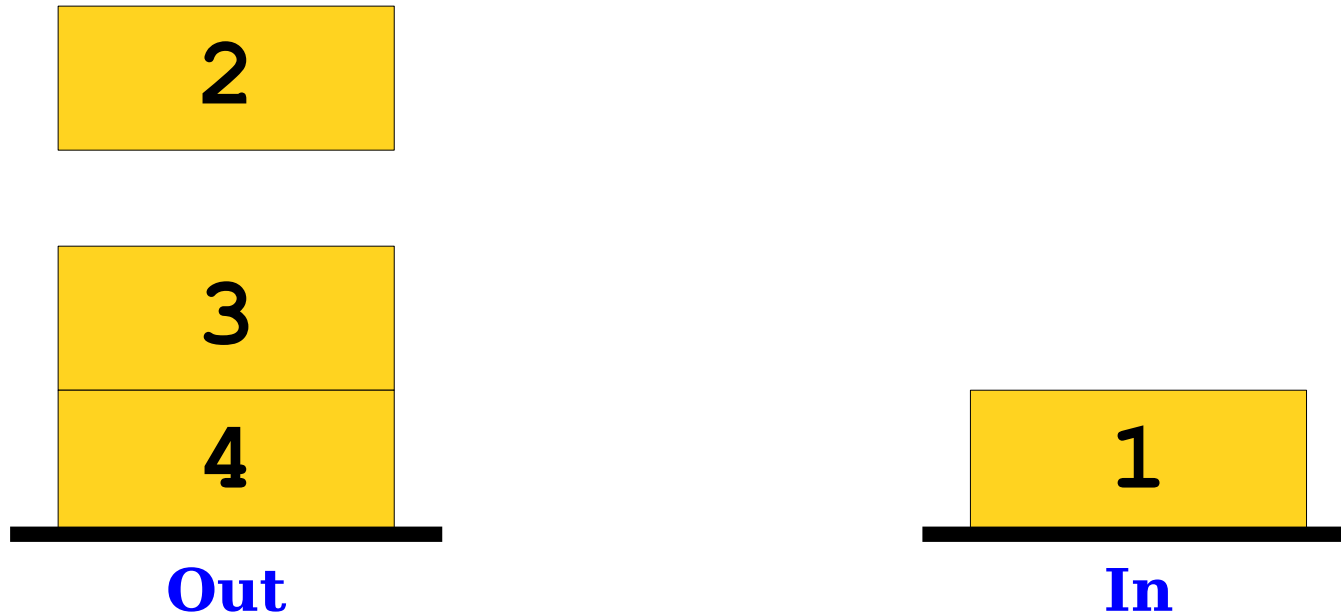
# The Two-Stack Queue



# The Two-Stack Queue



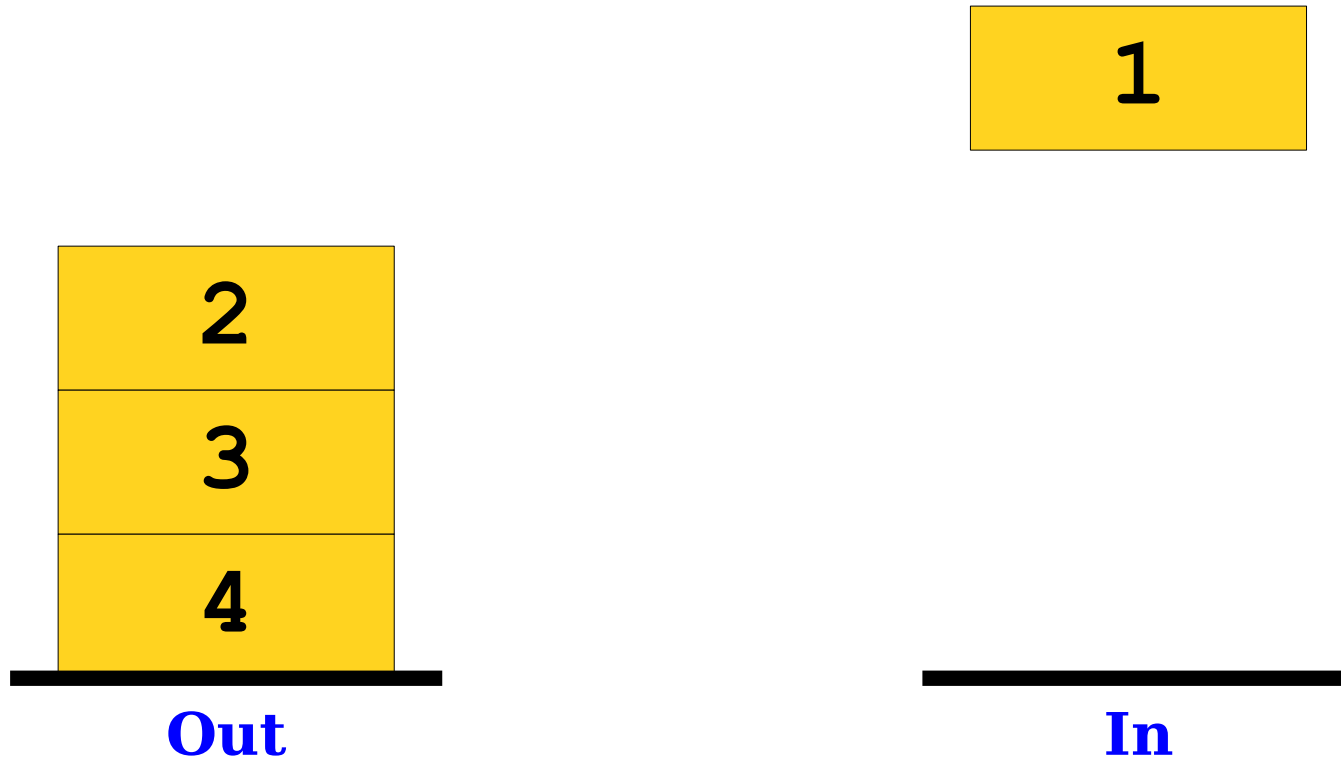
# The Two-Stack Queue



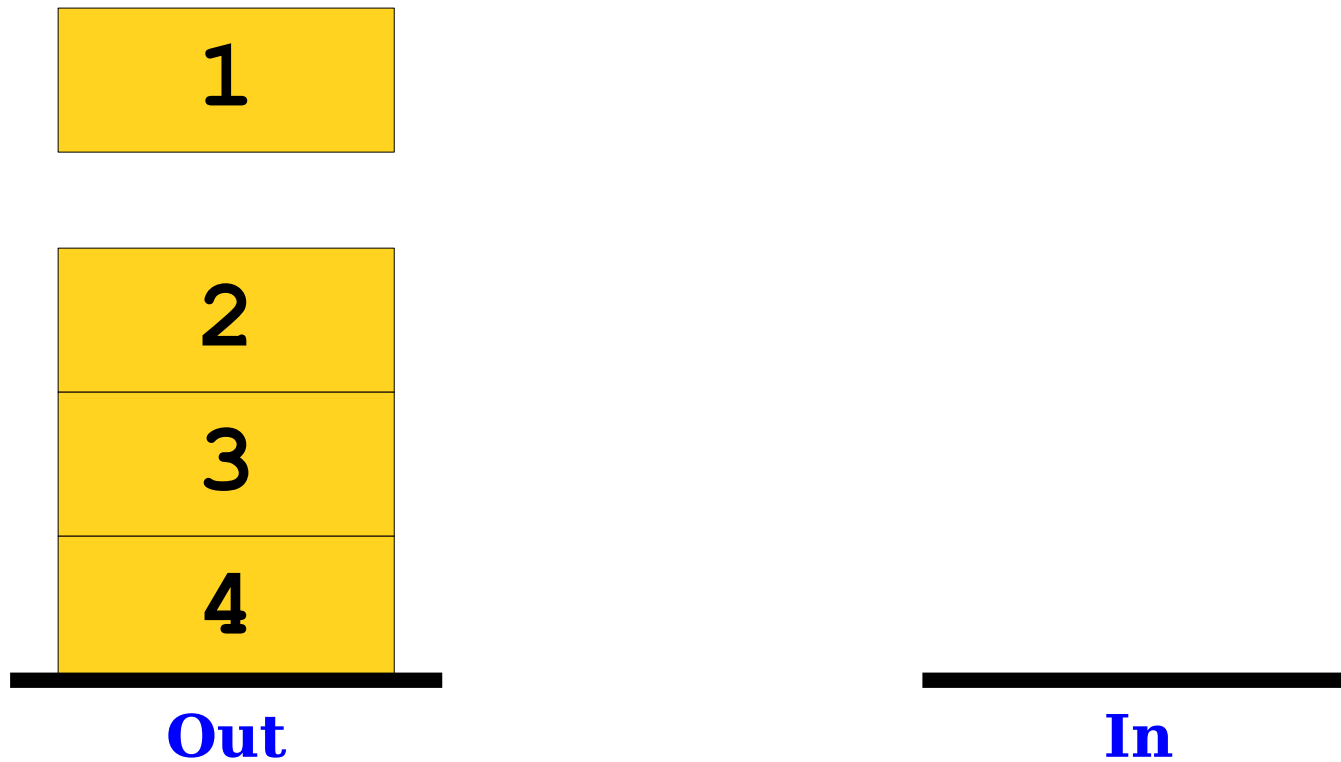
# The Two-Stack Queue



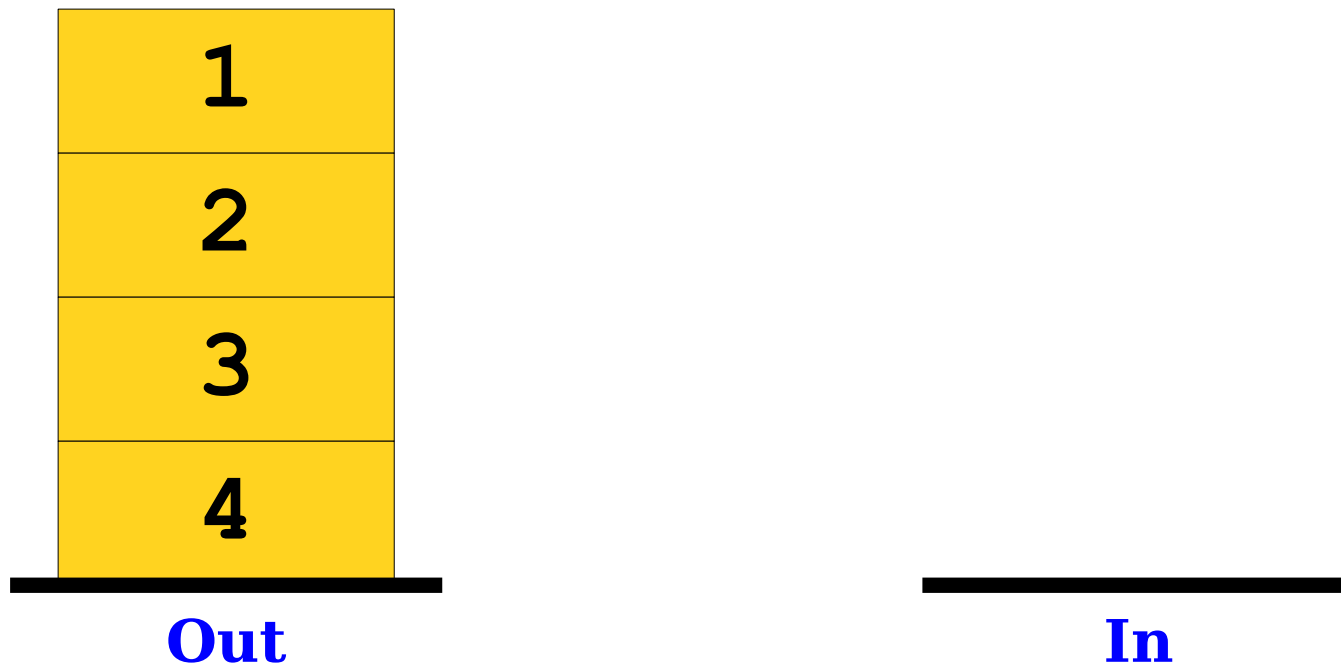
# The Two-Stack Queue



# The Two-Stack Queue

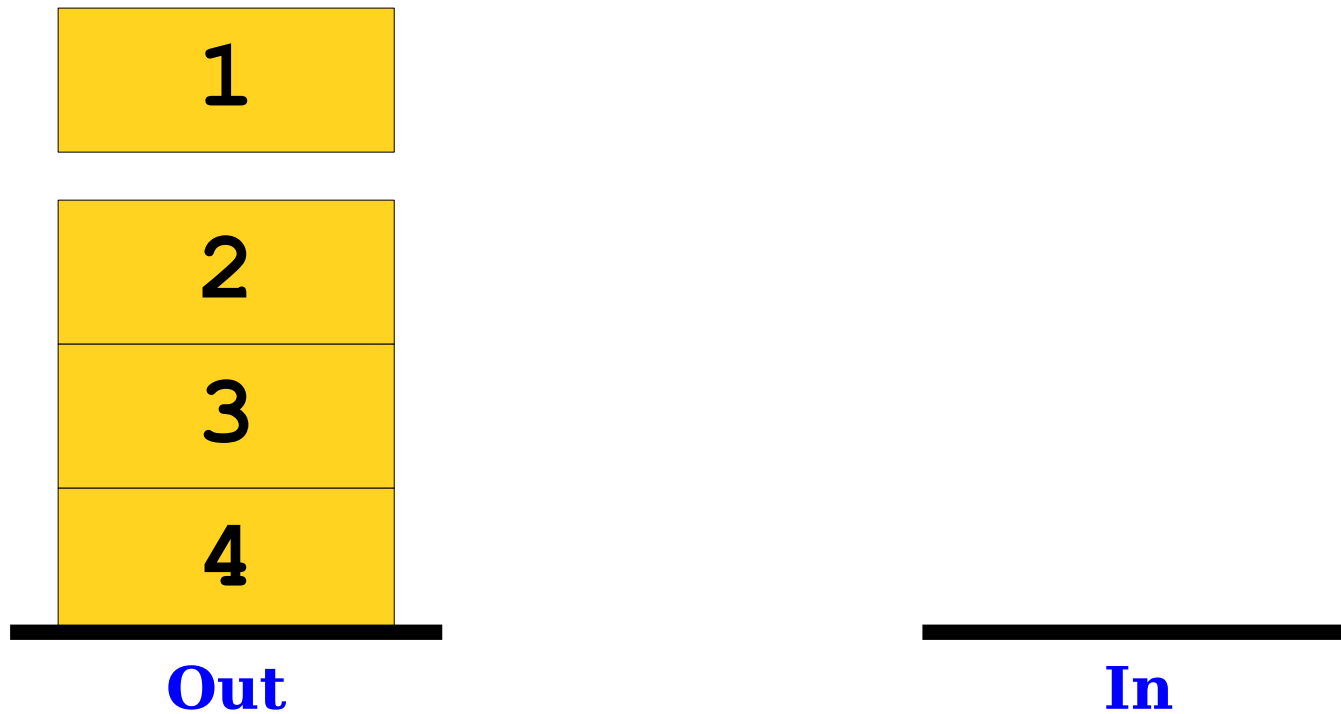


# The Two-Stack Queue





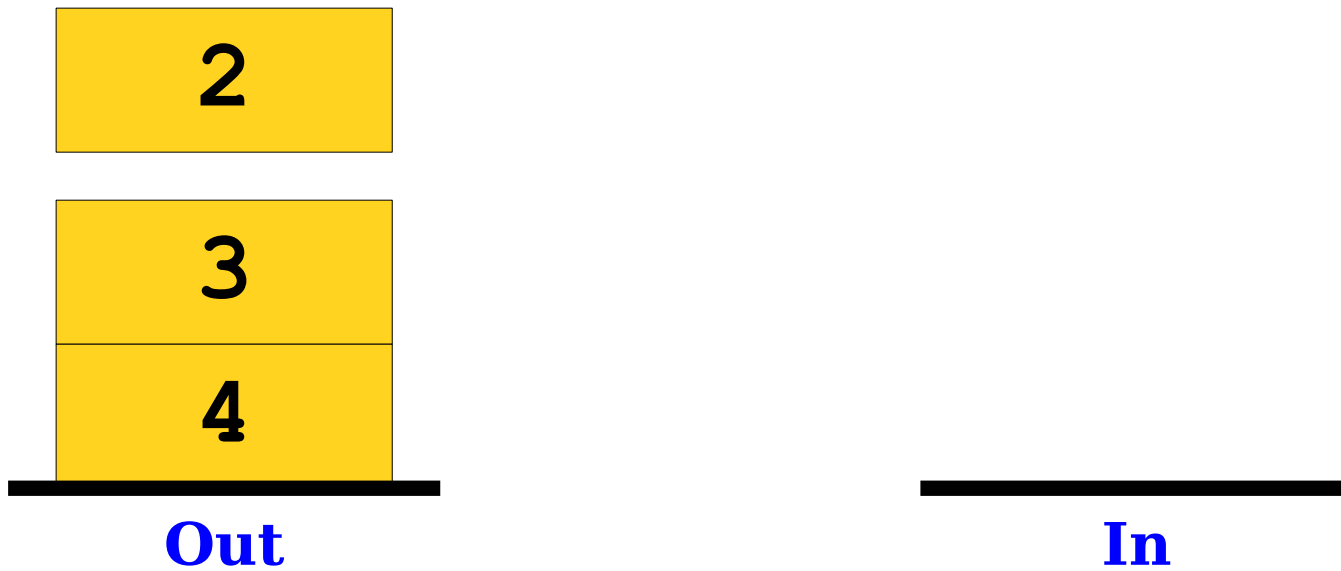
# The Two-Stack Queue



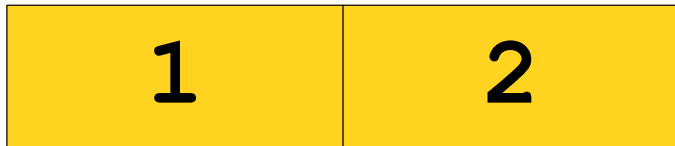
# The Two-Stack Queue



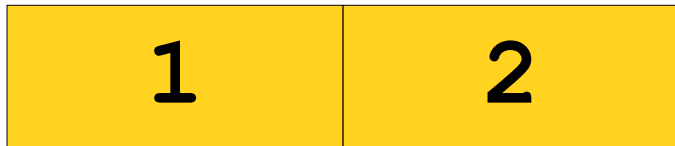
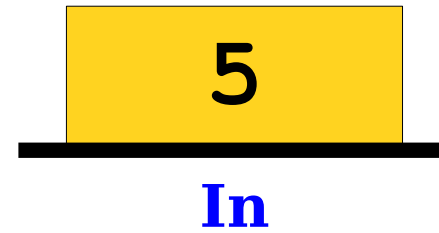
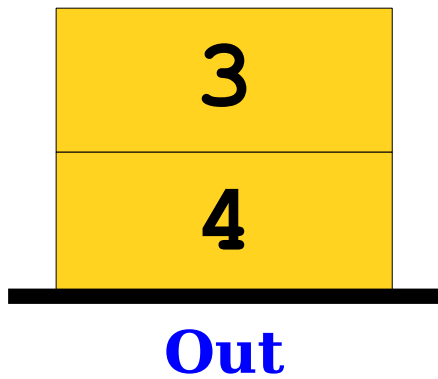
# The Two-Stack Queue



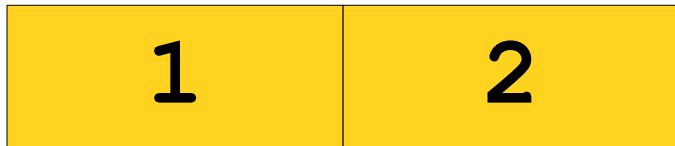
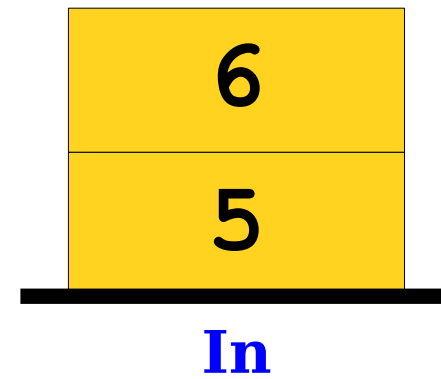
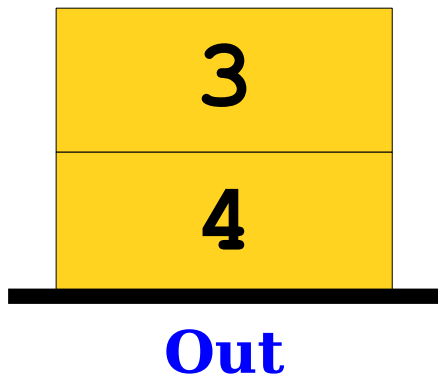
# The Two-Stack Queue



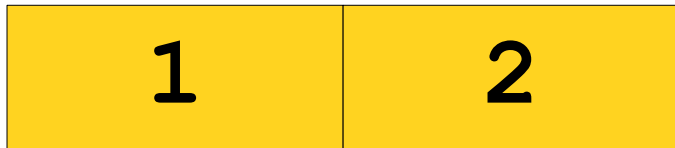
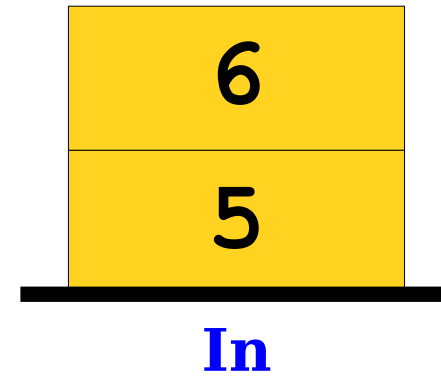
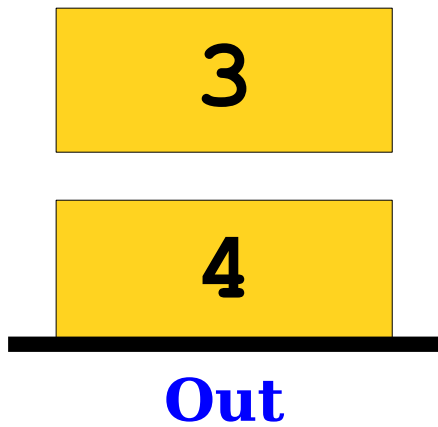
# The Two-Stack Queue



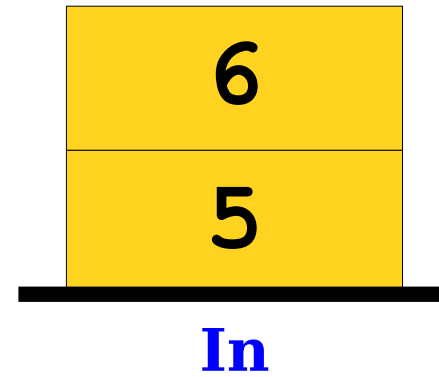
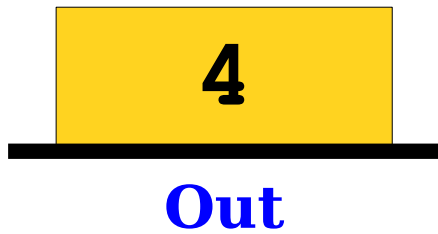
# The Two-Stack Queue



# The Two-Stack Queue

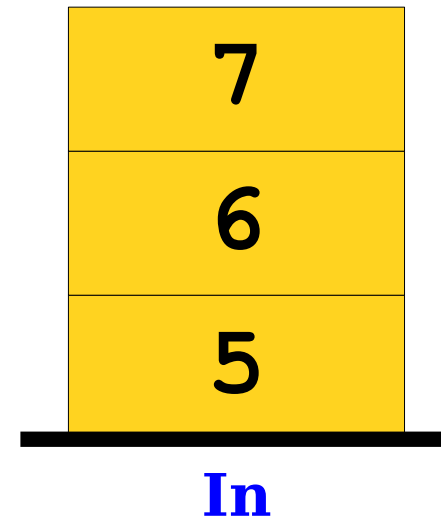


# The Two-Stack Queue

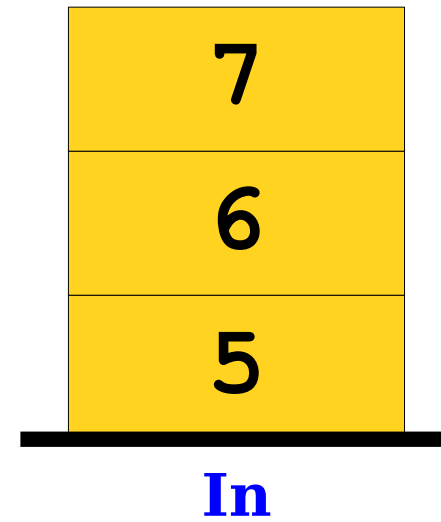
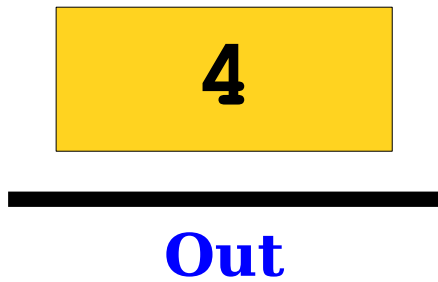




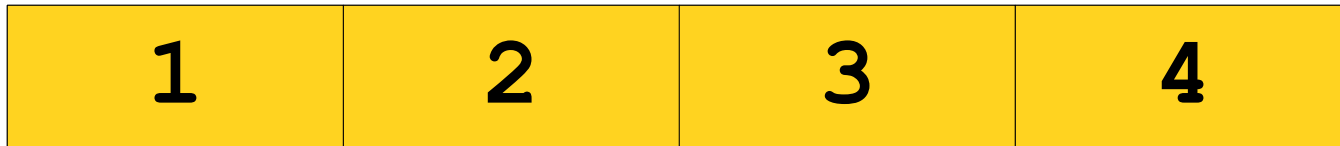
# The Two-Stack Queue



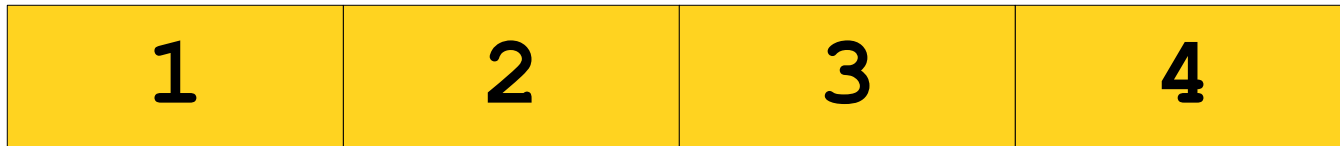
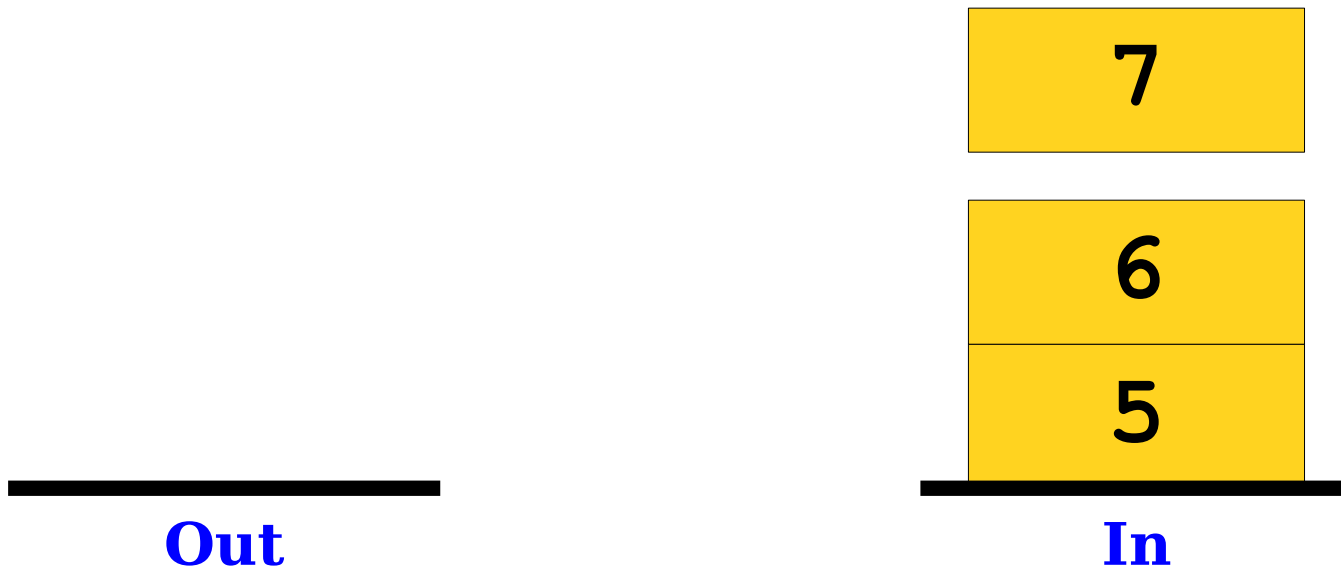
# The Two-Stack Queue



# The Two-Stack Queue



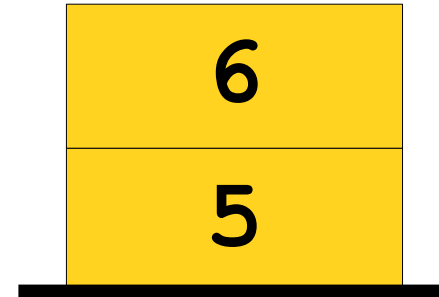
# The Two-Stack Queue



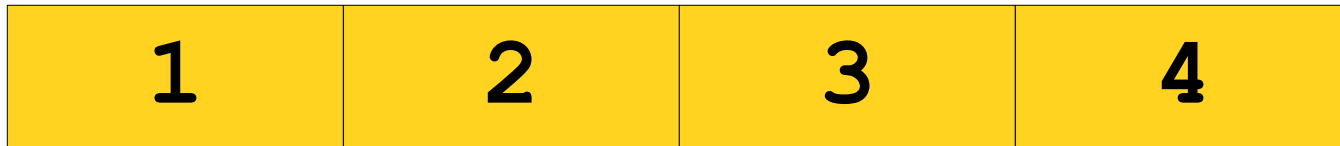
# The Two-Stack Queue



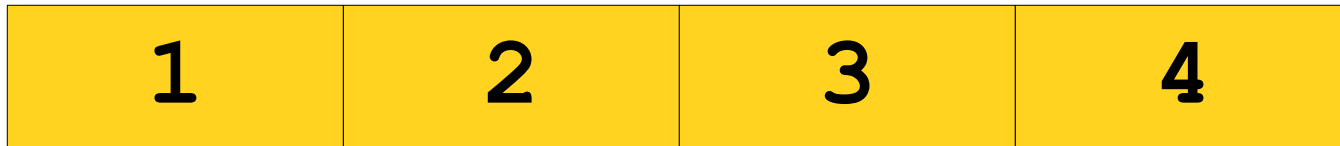
**Out**



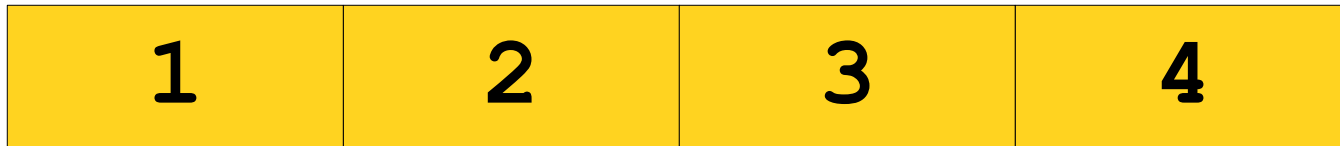
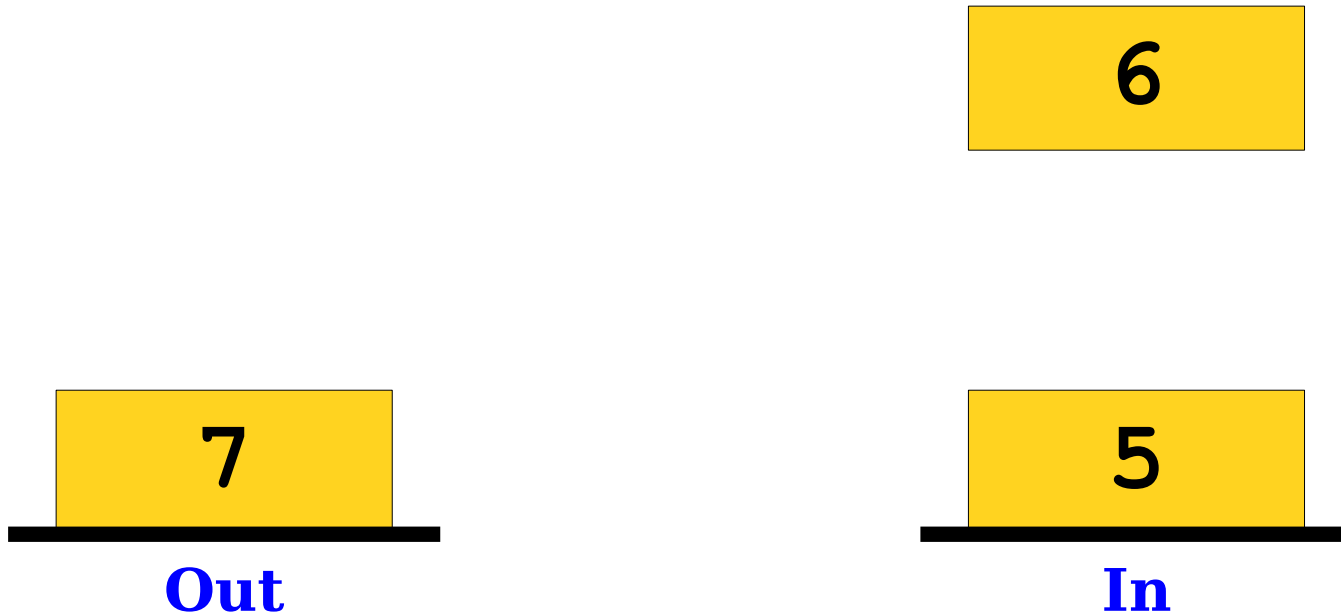
**In**



# The Two-Stack Queue



# The Two-Stack Queue



# The Two-Stack Queue

6

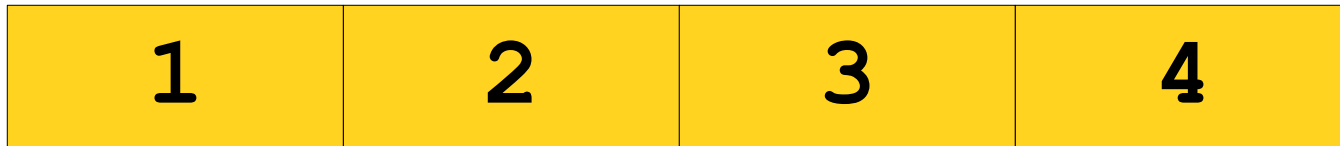
7  
**Out**

5  
**In**

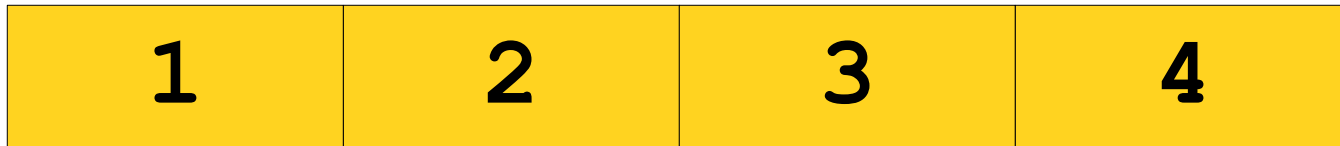
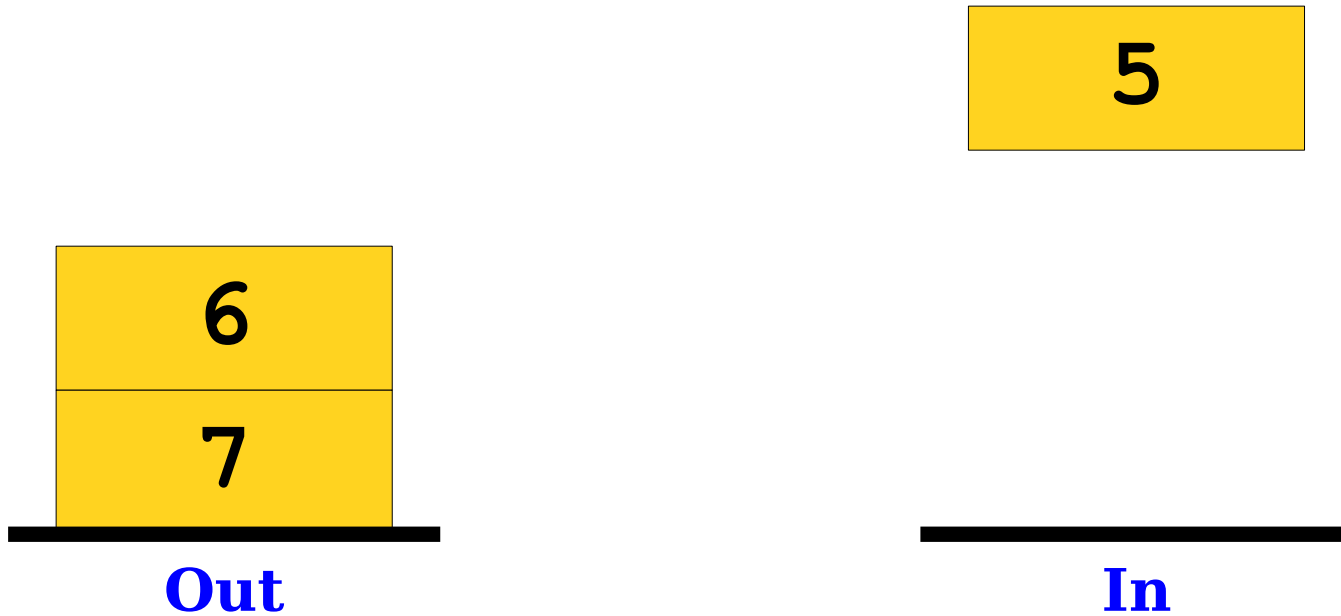
1 2 3 4



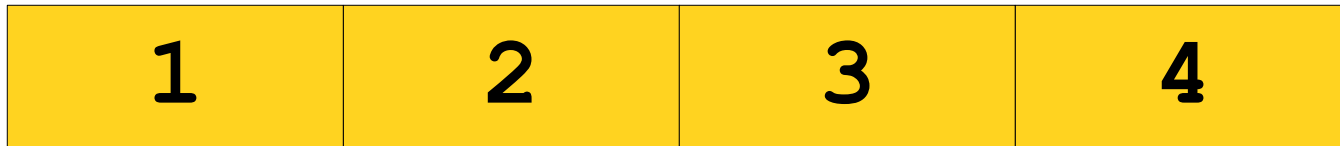
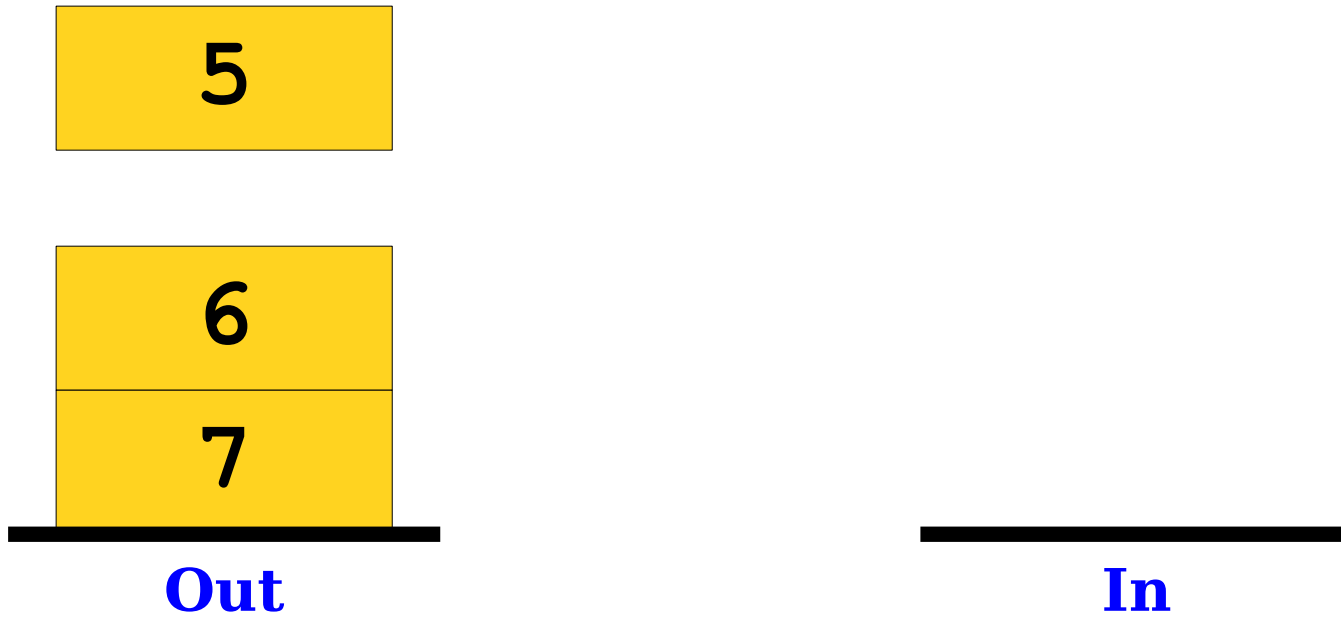
# The Two-Stack Queue



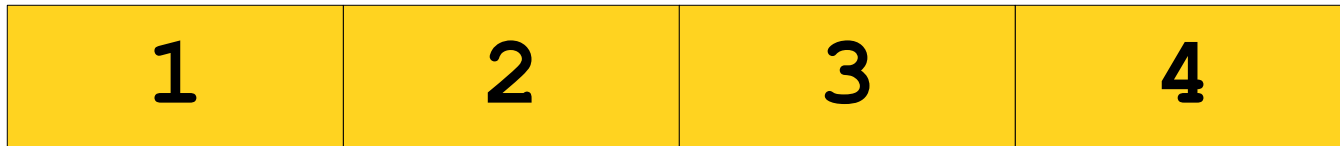
# The Two-Stack Queue



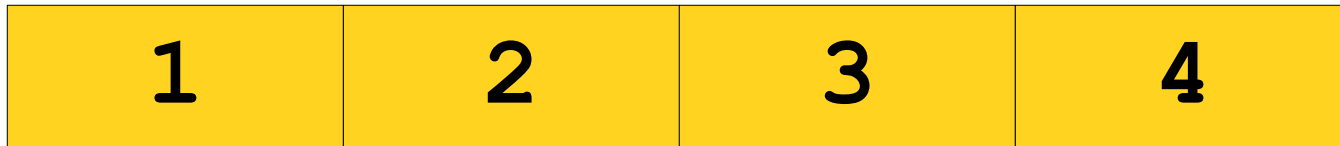
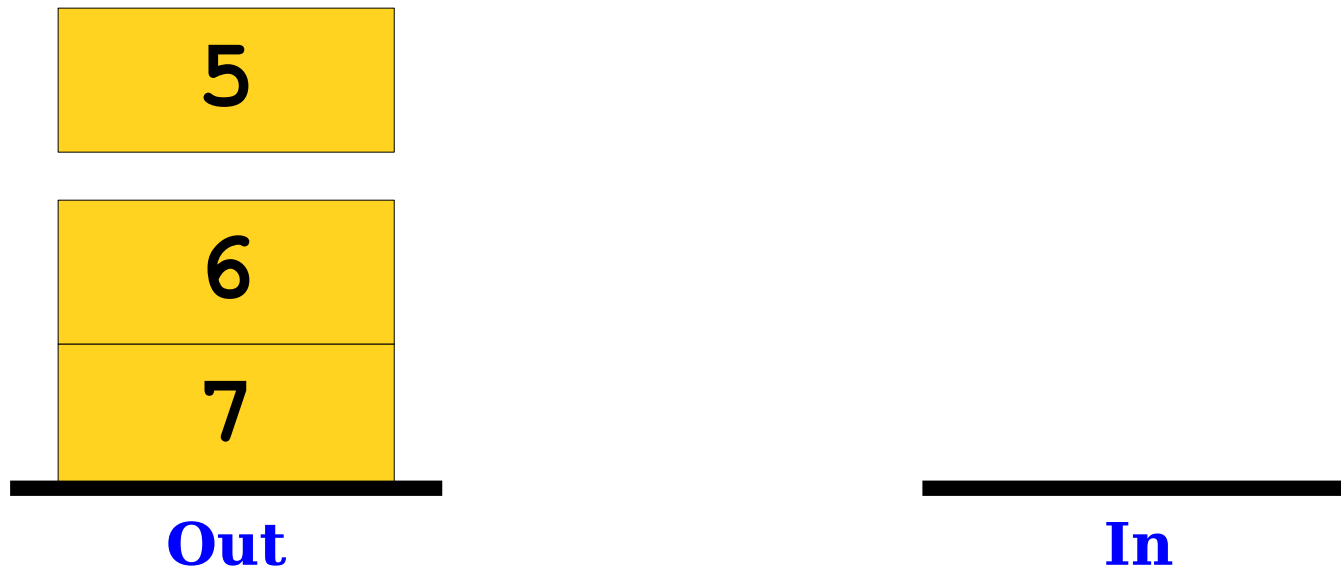
# The Two-Stack Queue



# 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.

# 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.



# Next Time

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