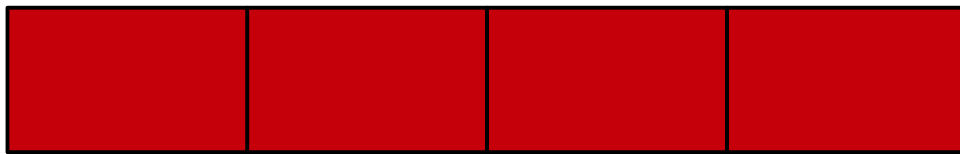
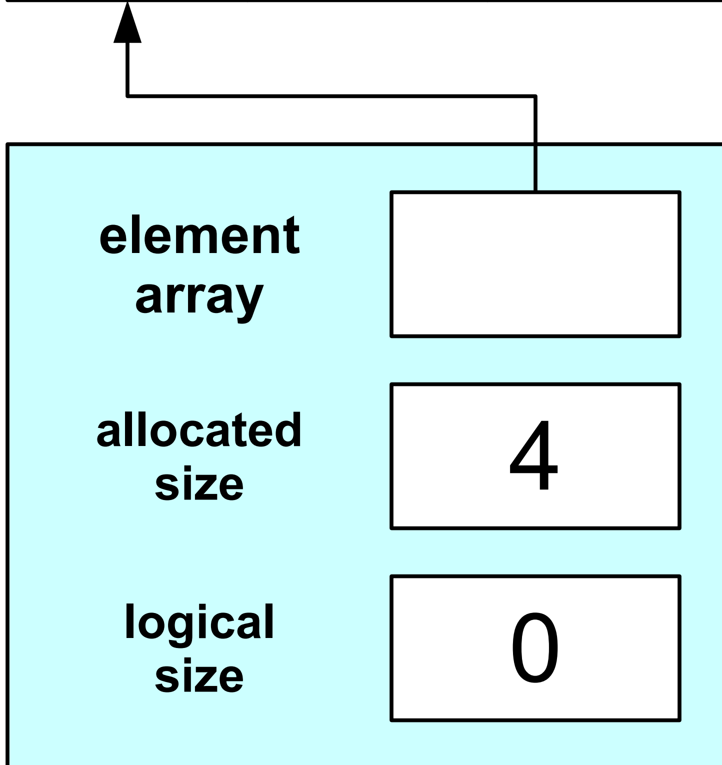
Implementing Abstractions

Part Two

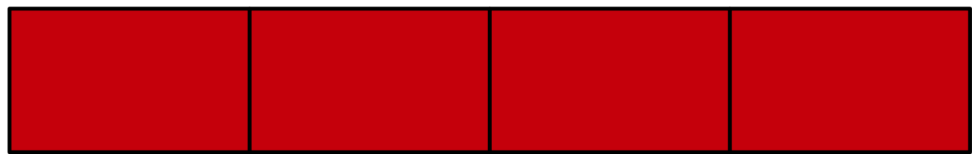
Previously, on CS106B…

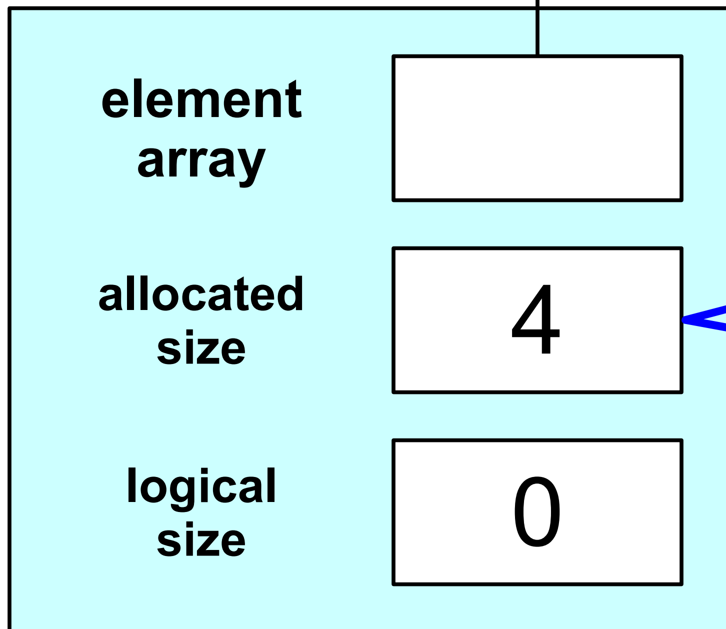
A Bounded Stack





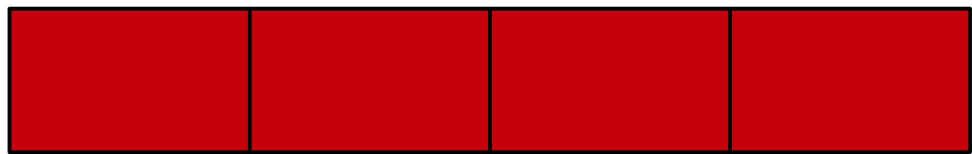
A Bounded Stack

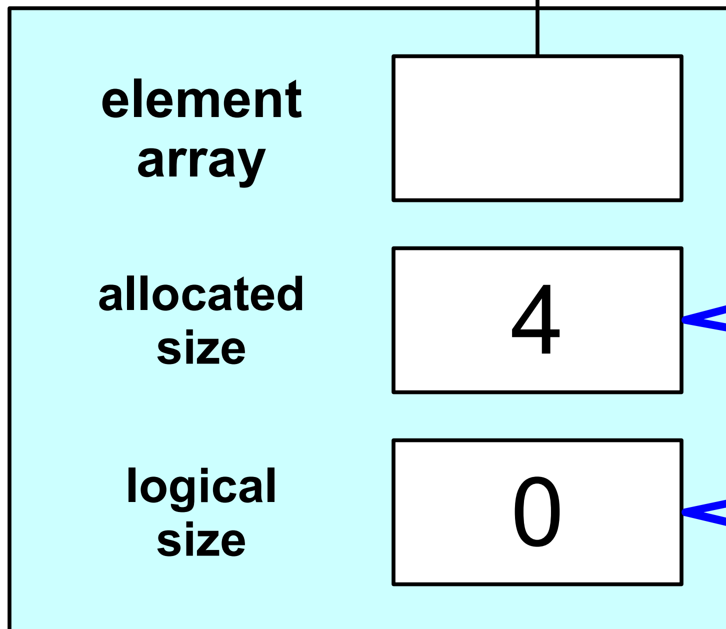




The stack's ***allocated size*** is the number of slots in the array. Remember – arrays in C++ cannot grow or shrink.

A Bounded Stack

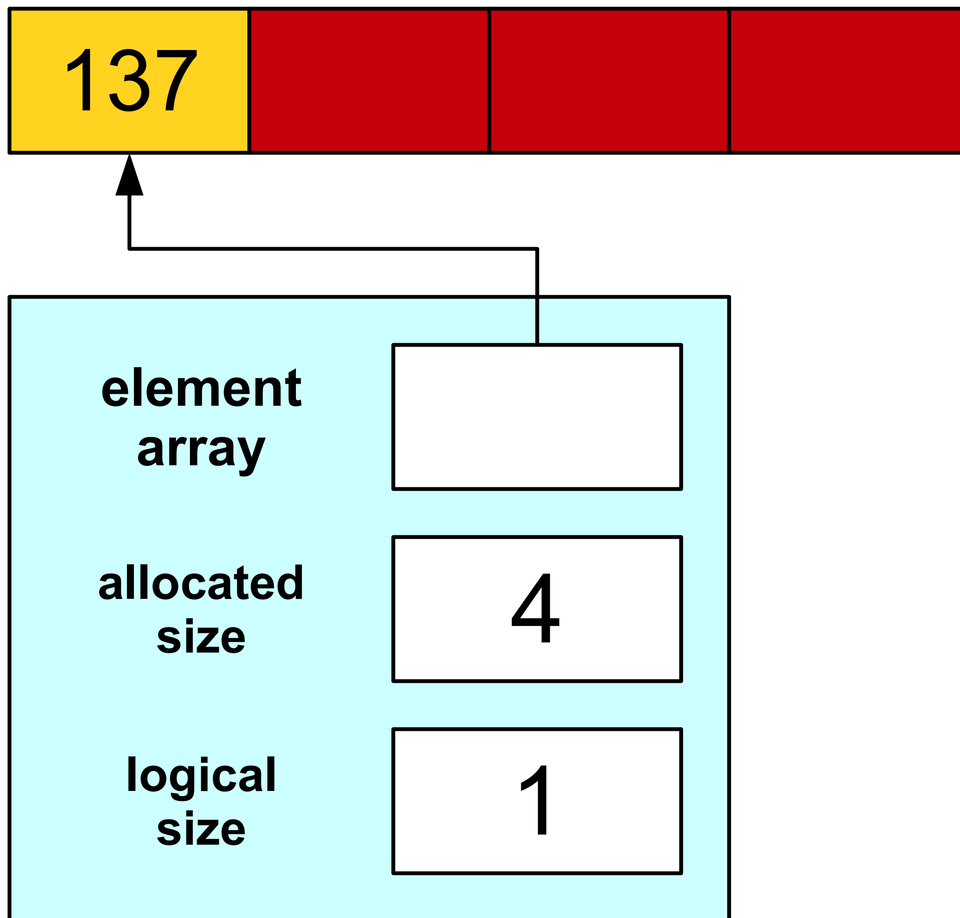




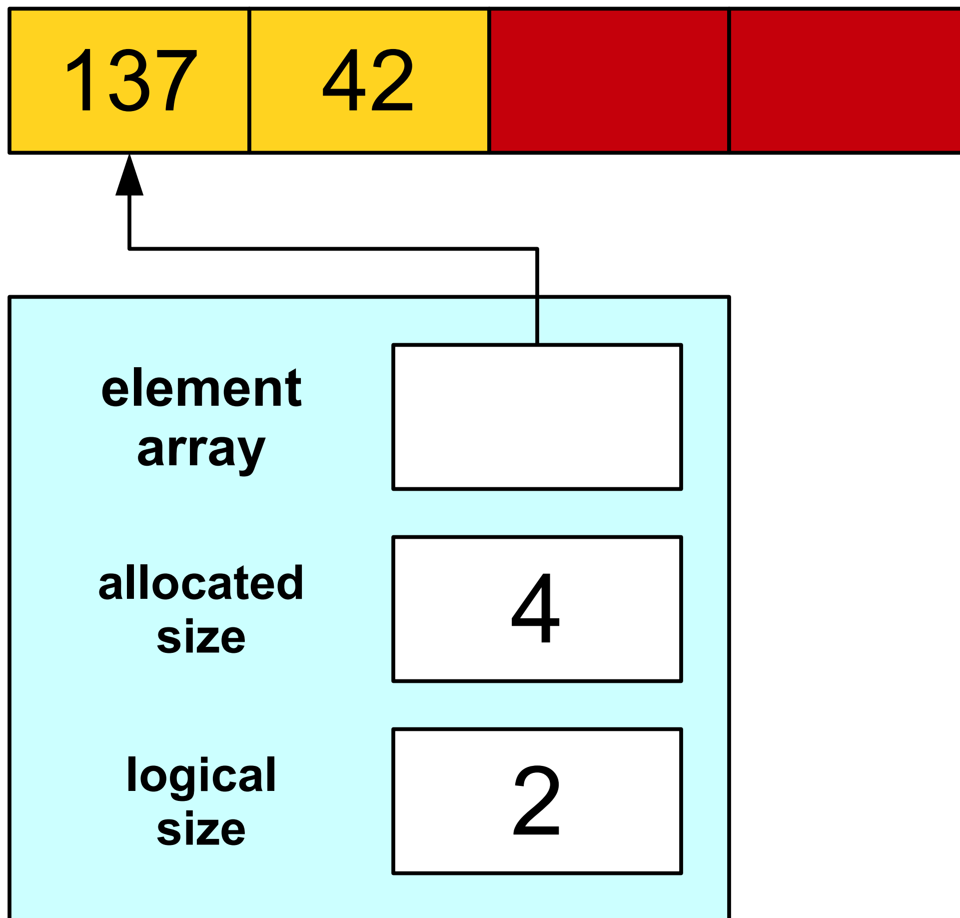
The stack's ***allocated size*** is the number of slots in the array. Remember – arrays in C++ cannot grow or shrink.

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.

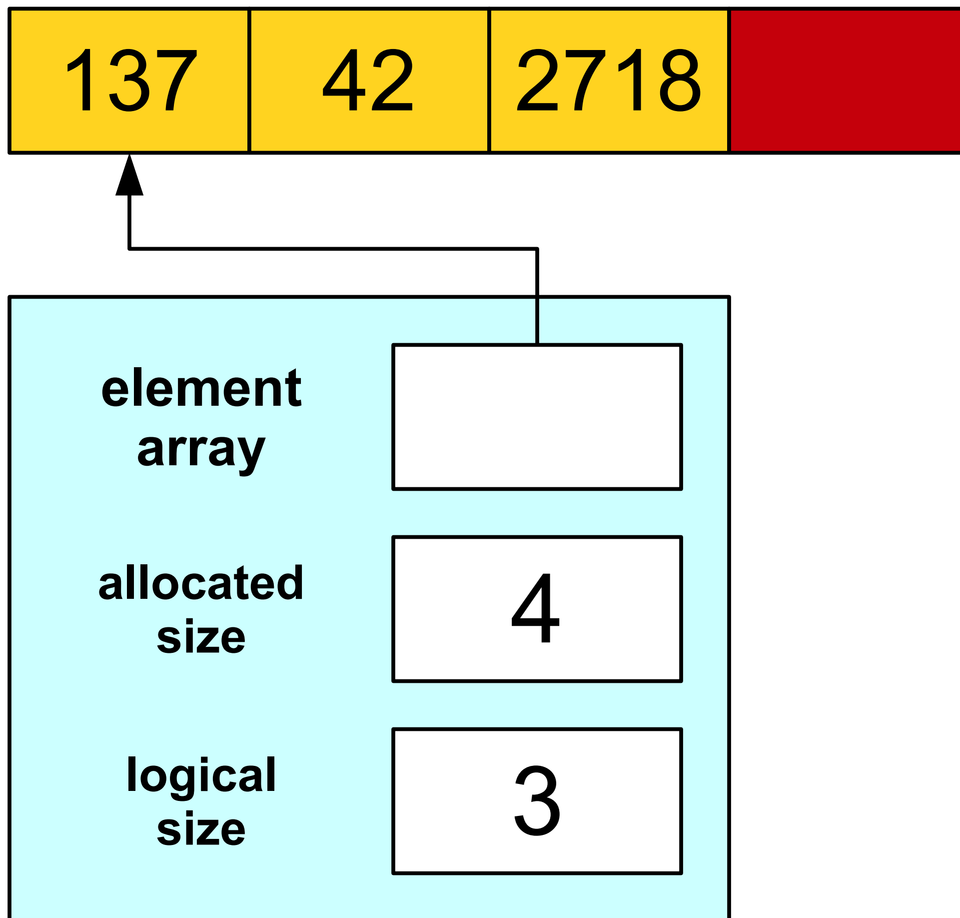
A Bounded Stack



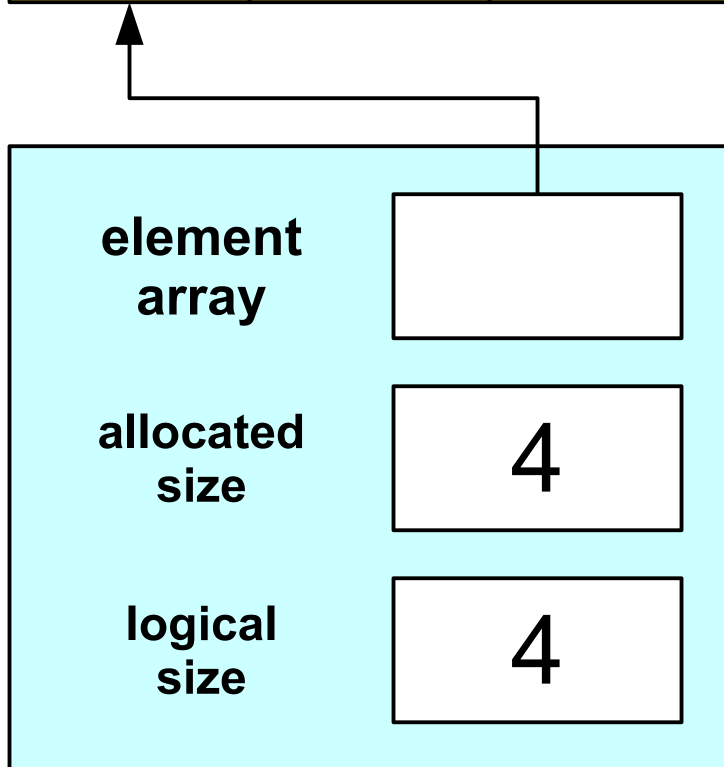
A Bounded Stack



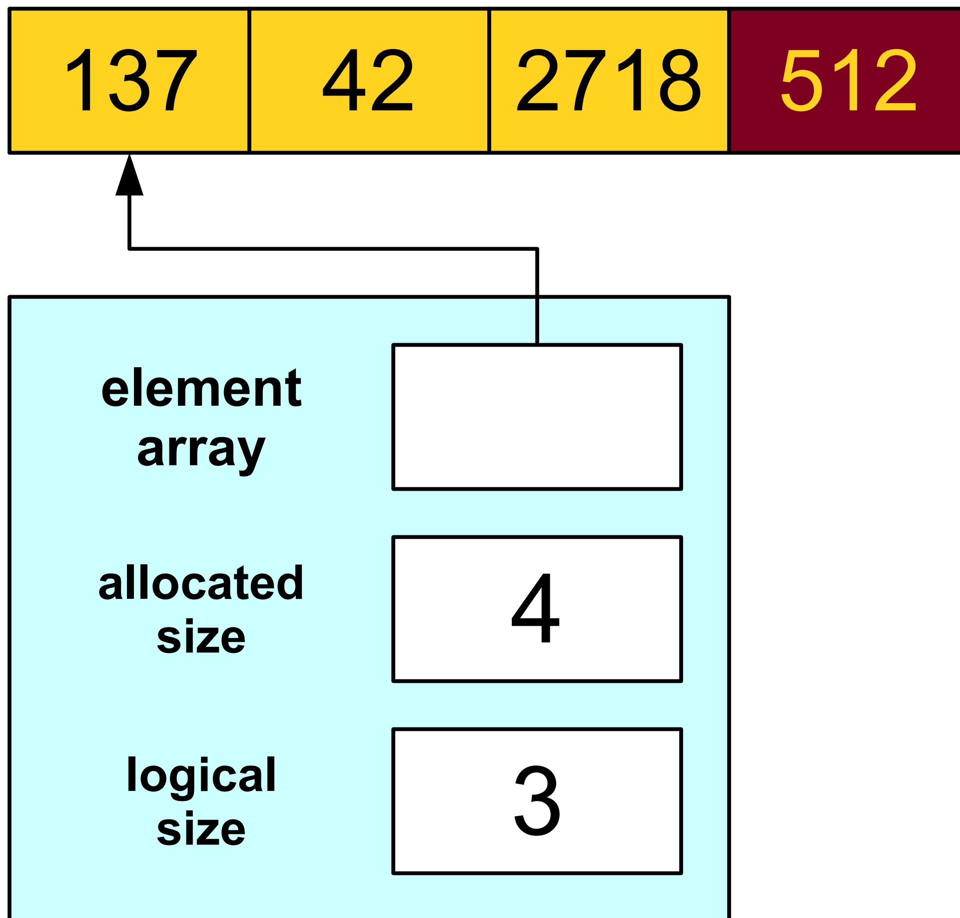
A Bounded Stack



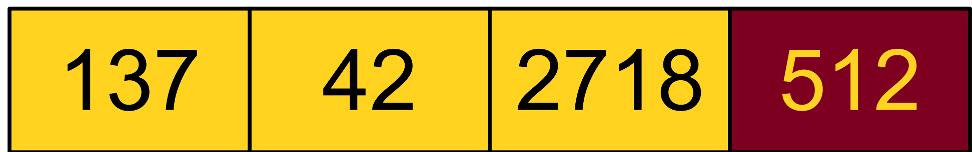
A Bounded Stack

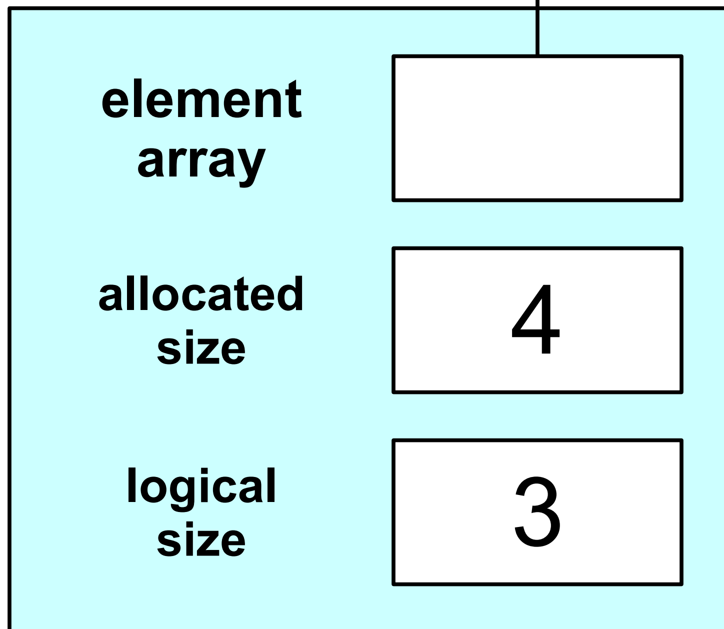


A Bounded Stack



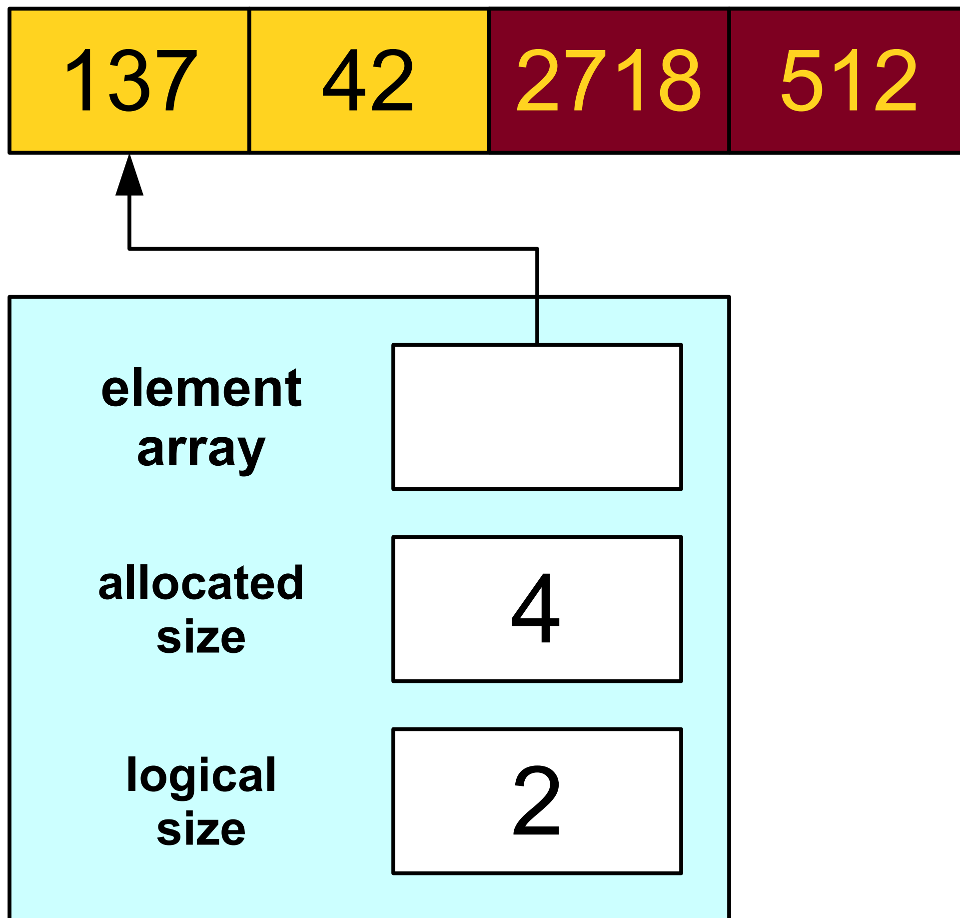
A Bounded Stack



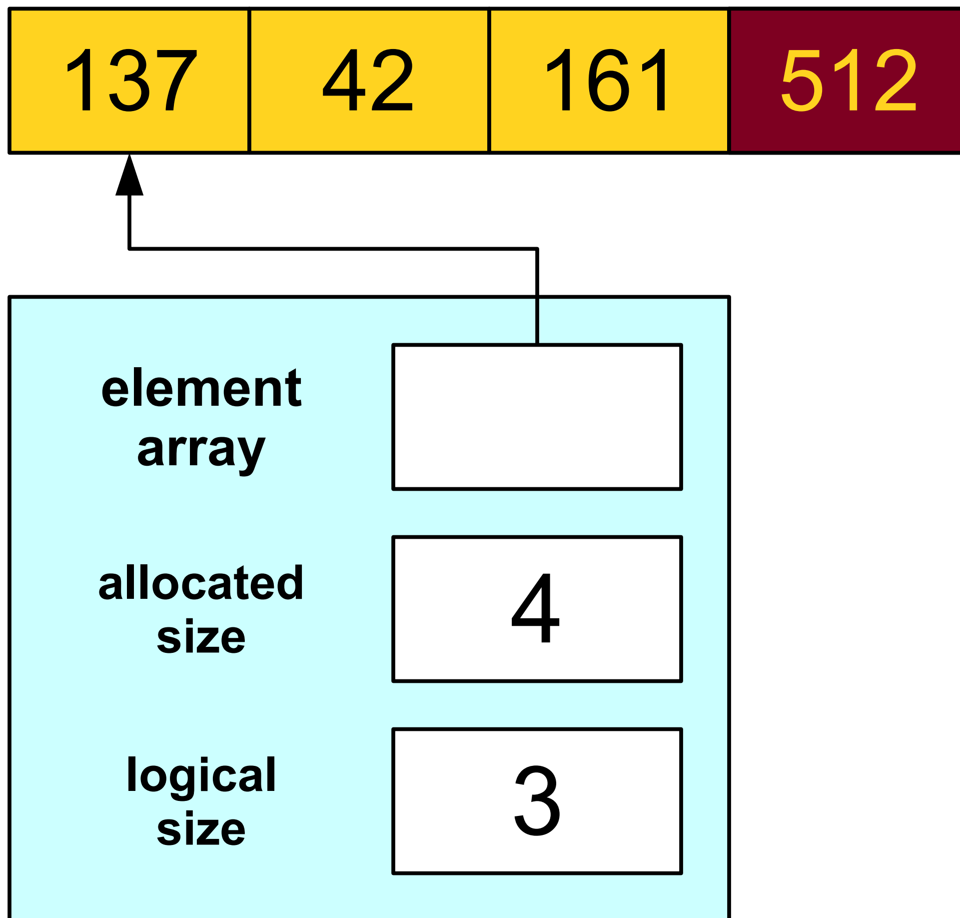


Arrays cannot grow or shrink, so this older value is still technically there in the array. We're just going to pretend it isn't.

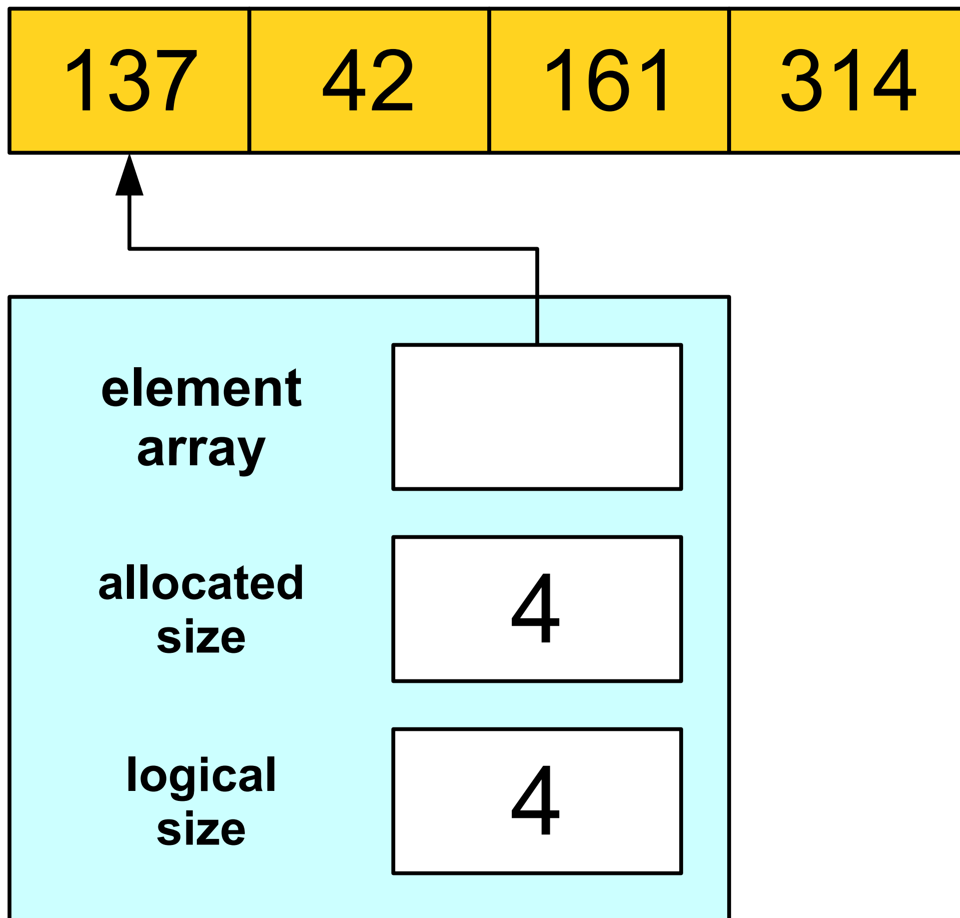
A Bounded Stack



A Bounded Stack



A Bounded Stack

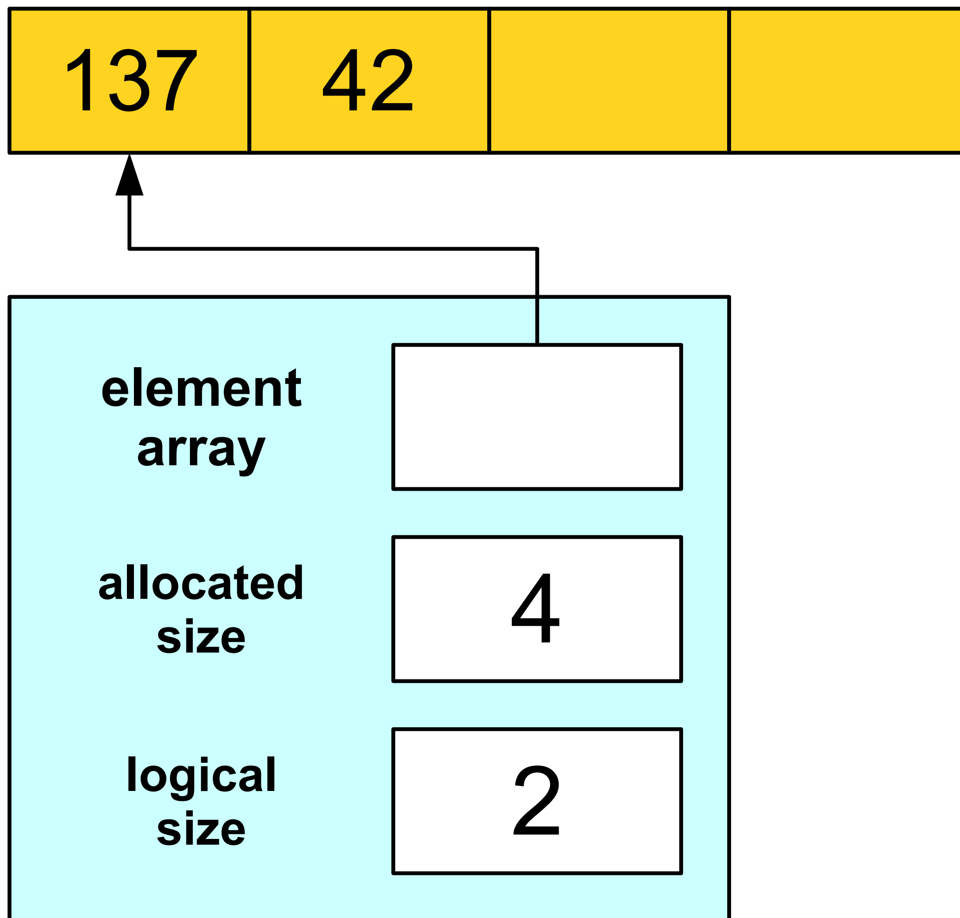


New Stuff!

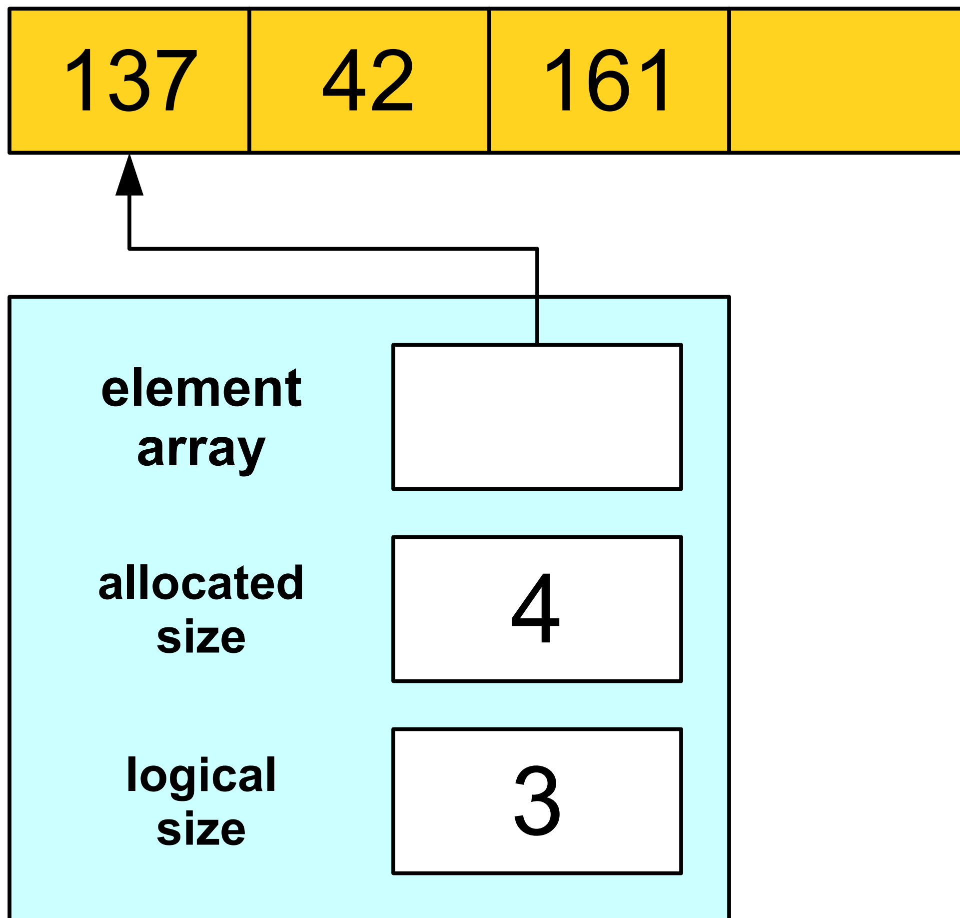
Running out of Space

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

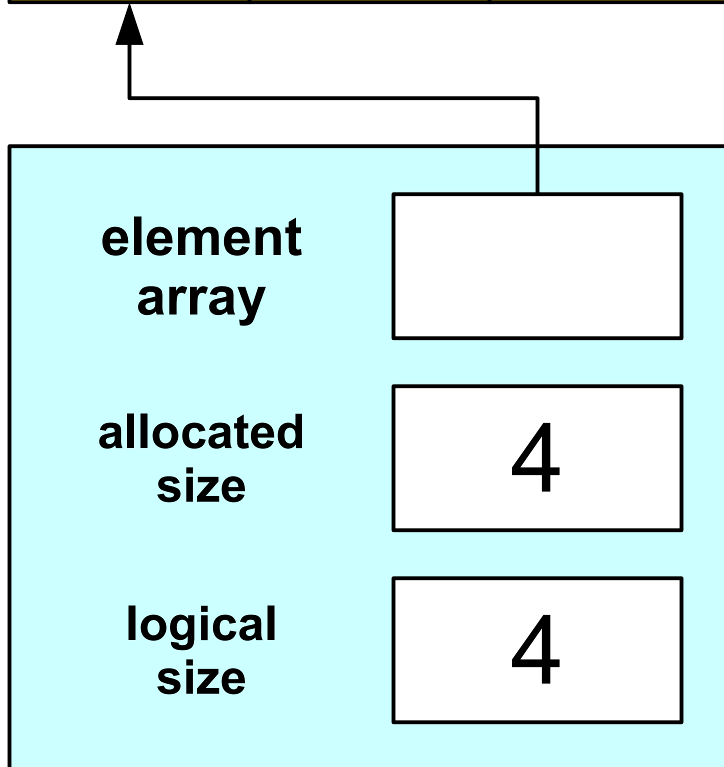
An Initial Idea



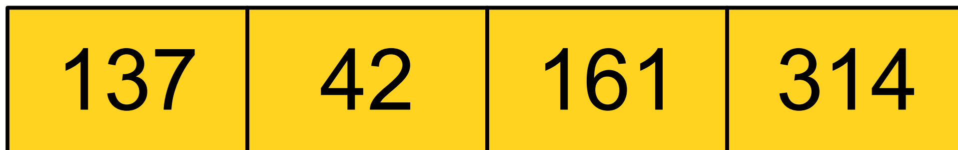
An Initial Idea

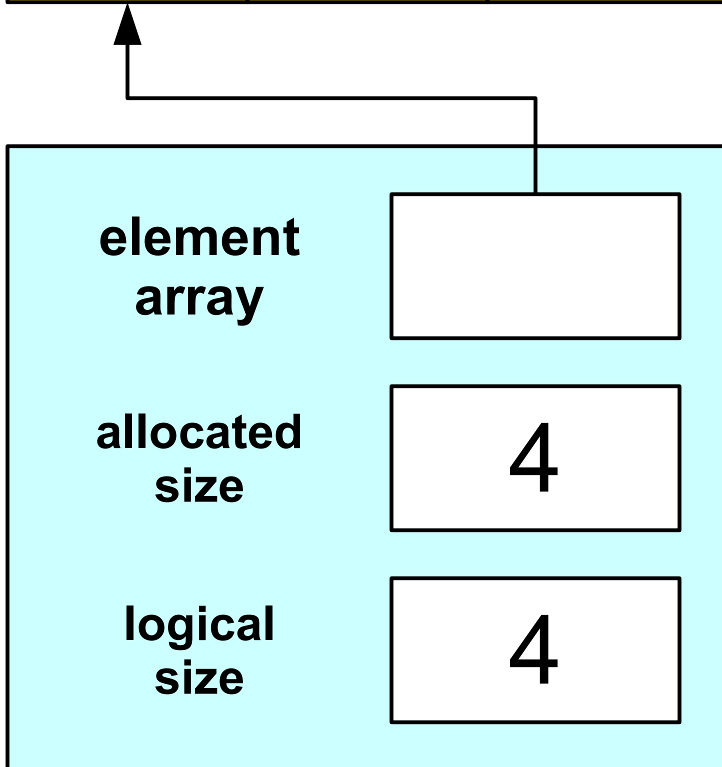


An Initial Idea

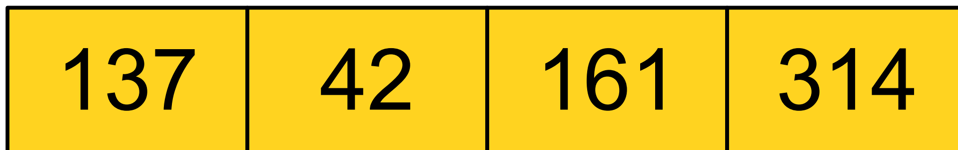


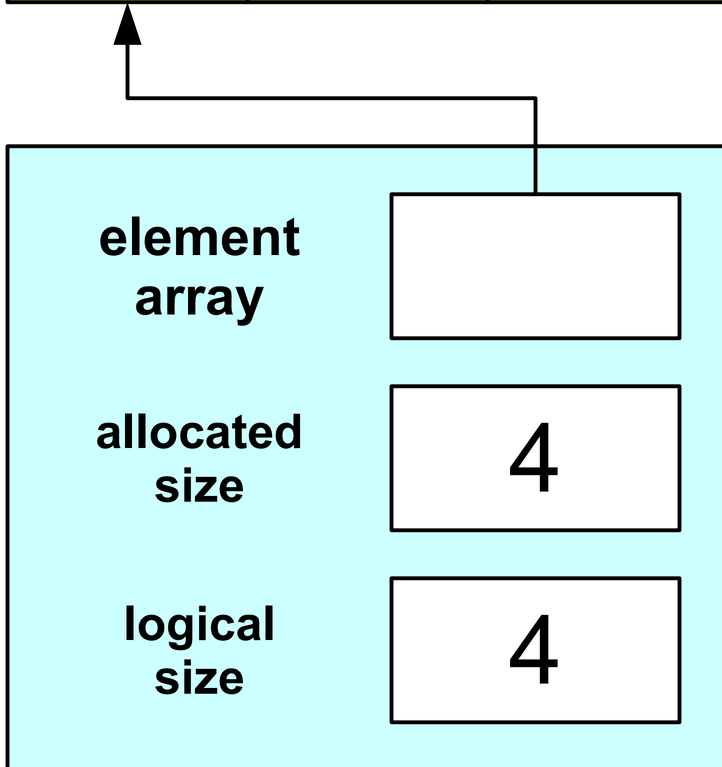
An Initial Idea



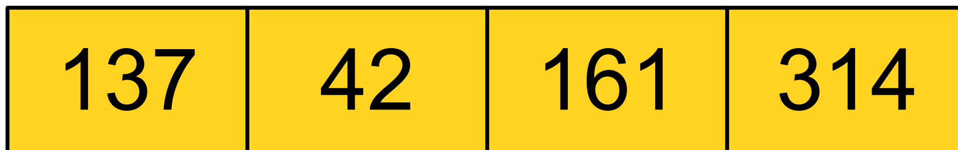


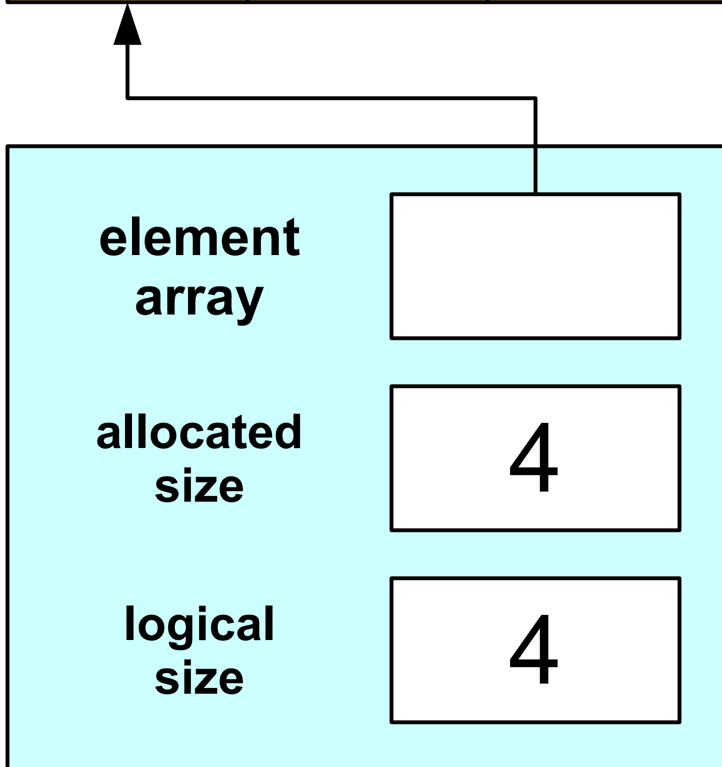
An Initial Idea



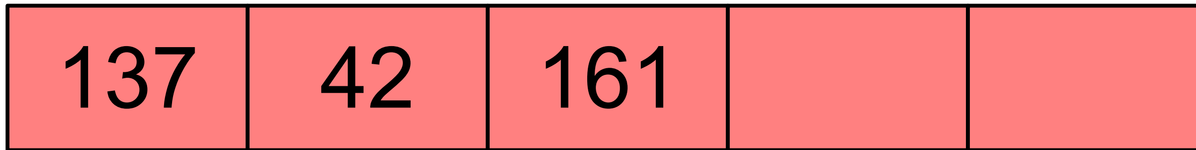


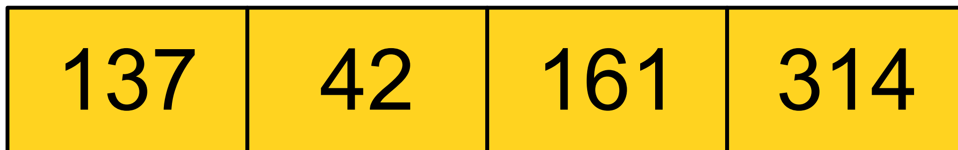
An Initial Idea

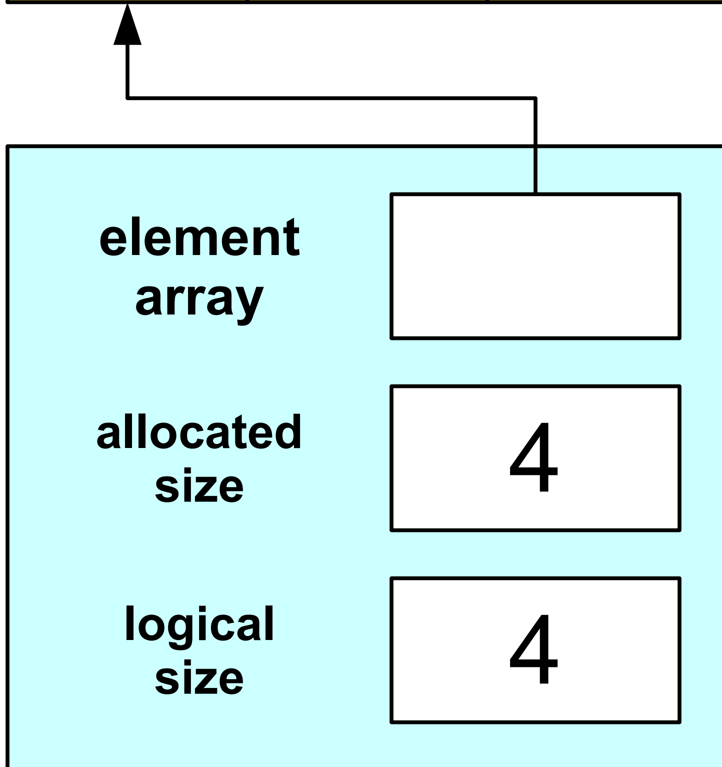




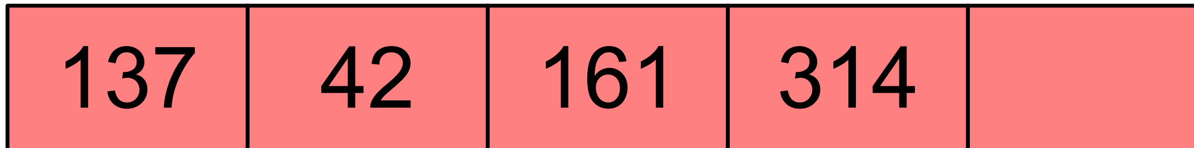
An Initial Idea

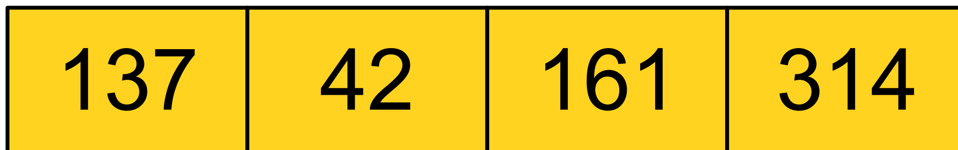


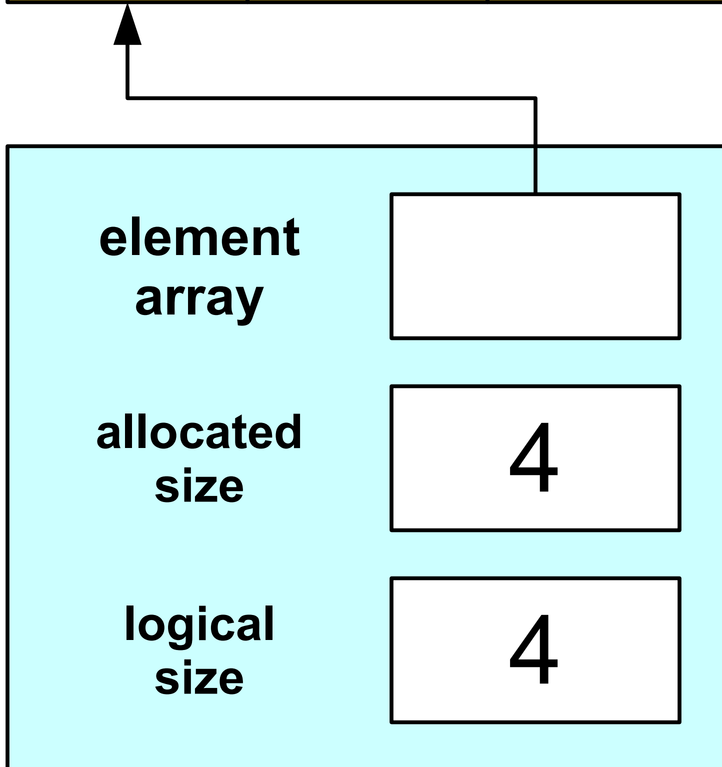




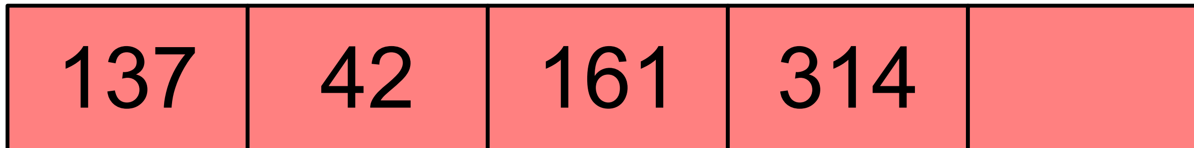
An Initial Idea

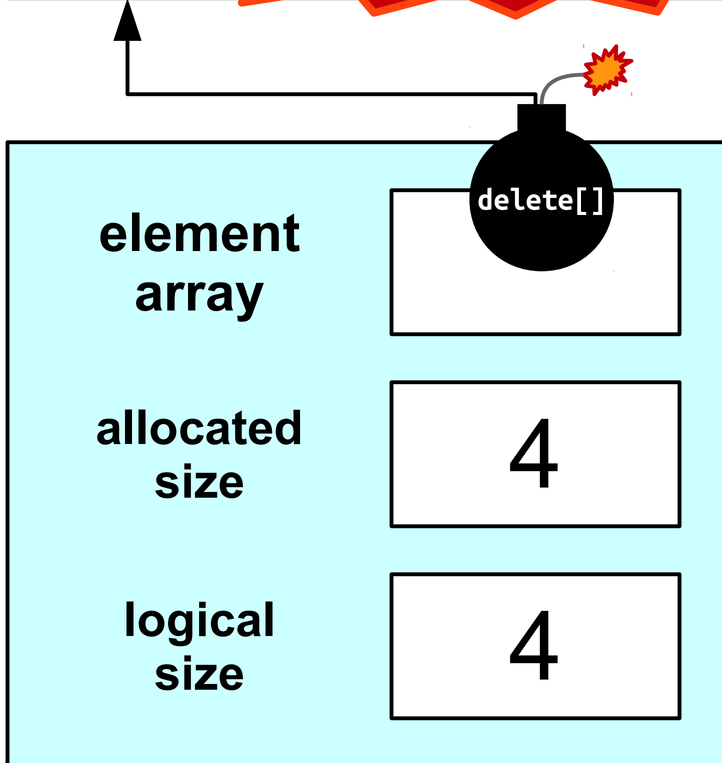




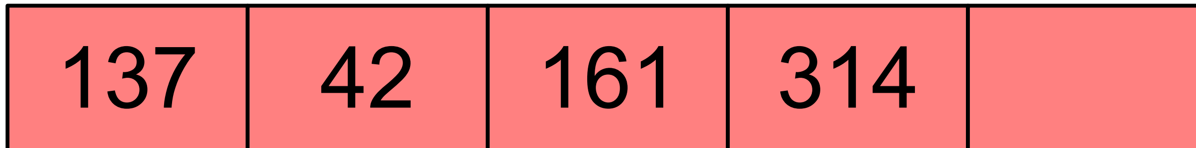


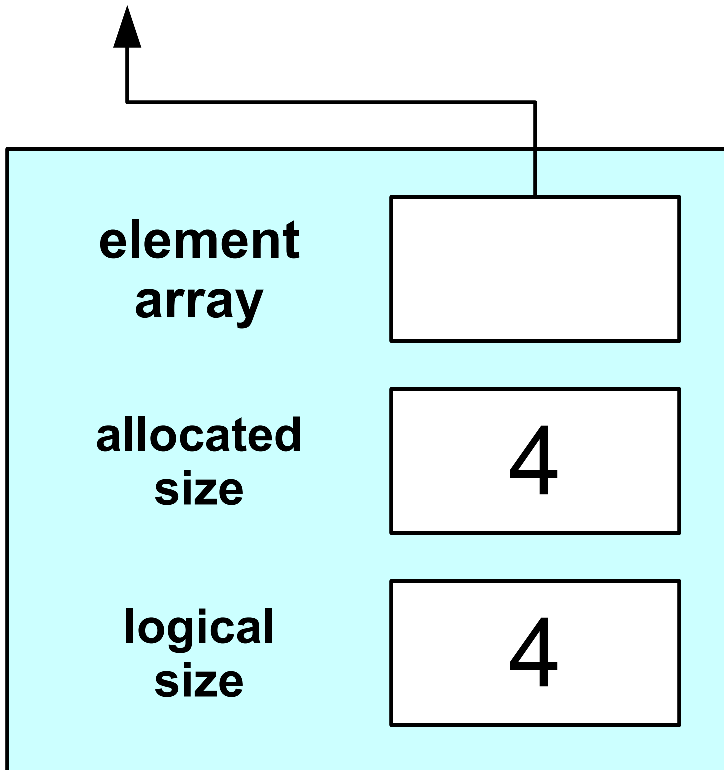
An Initial Idea



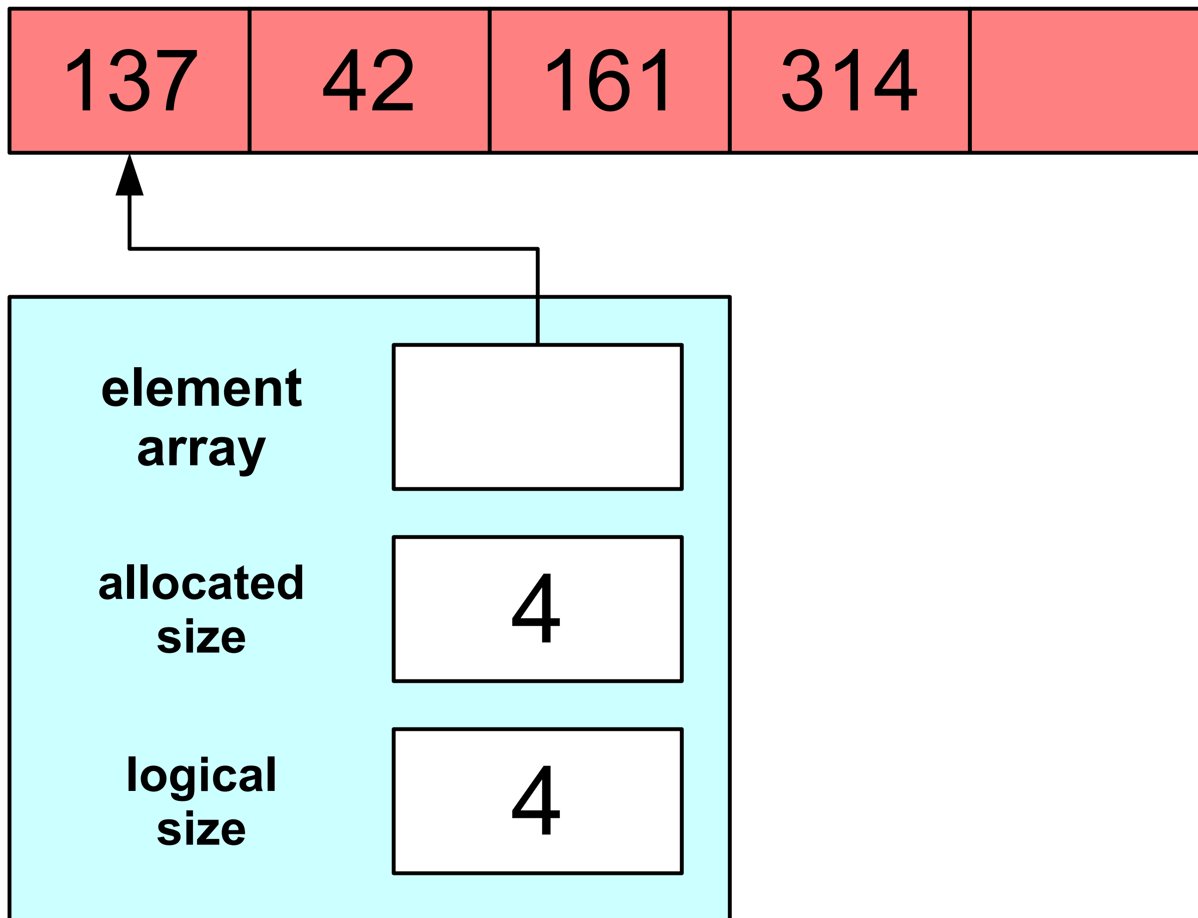


An Initial Idea

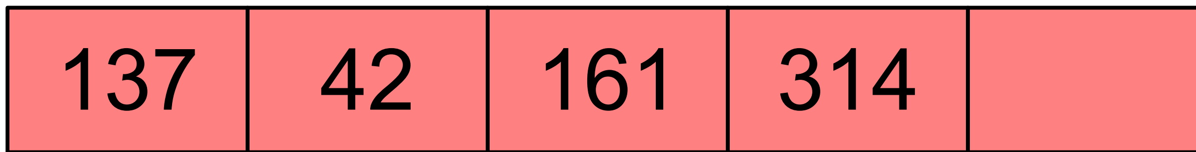


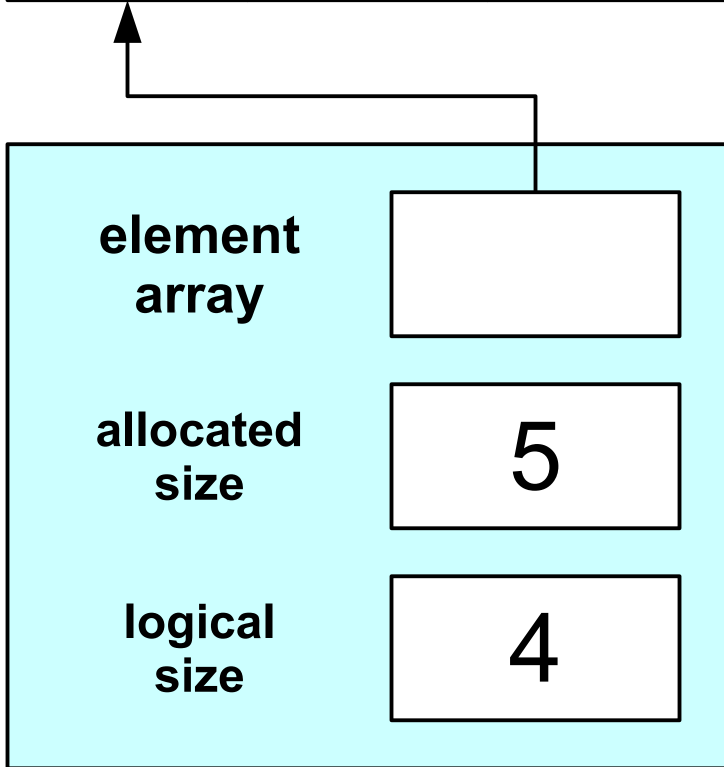


An Initial Idea

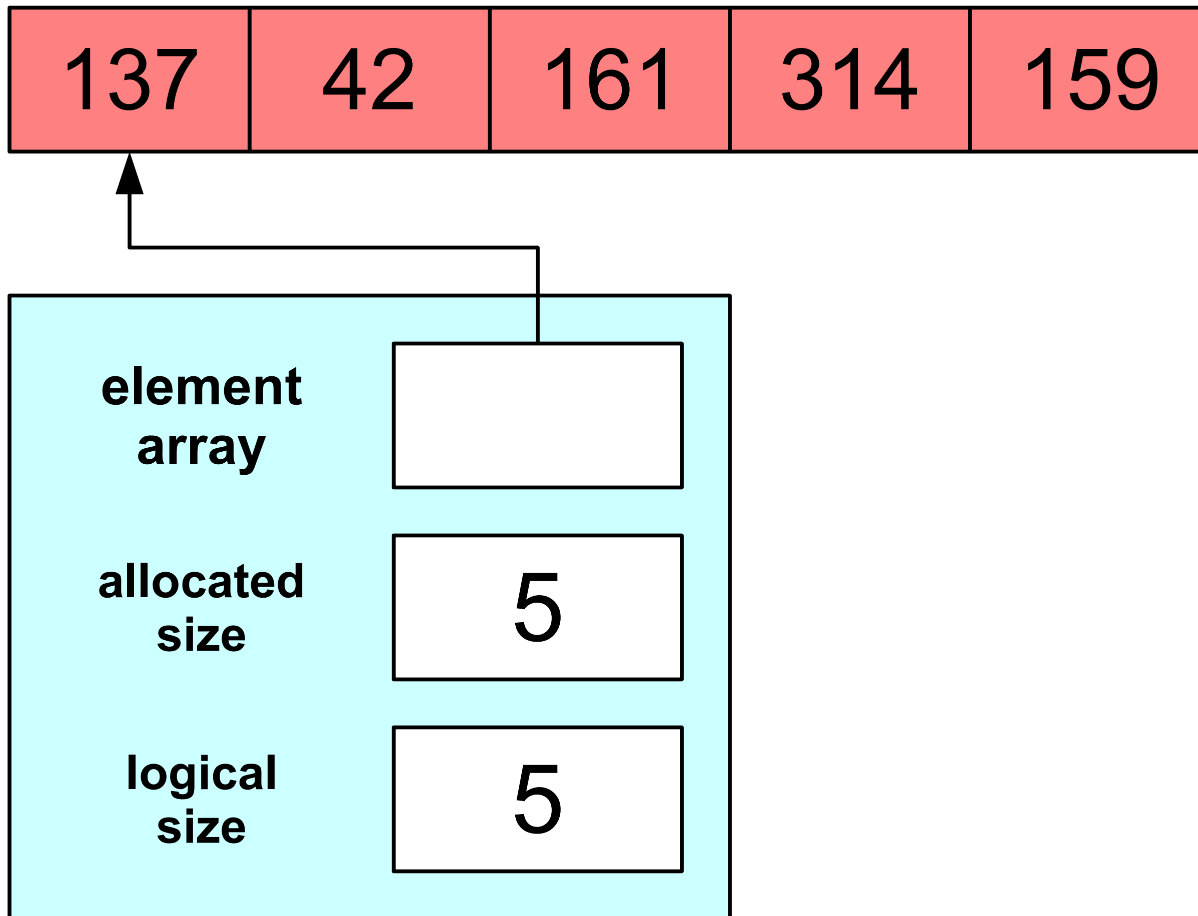


An Initial Idea

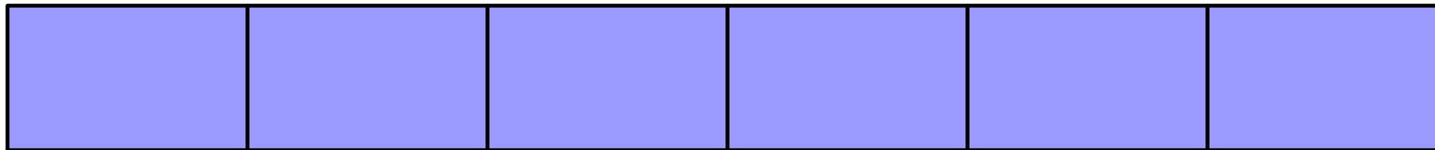


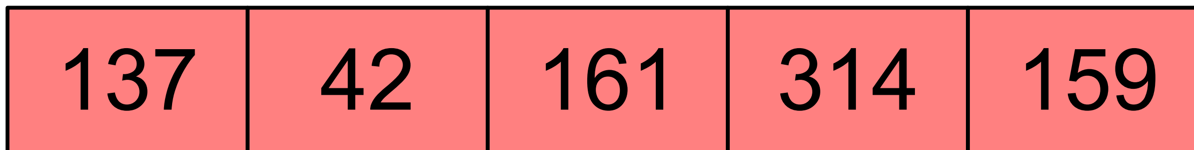


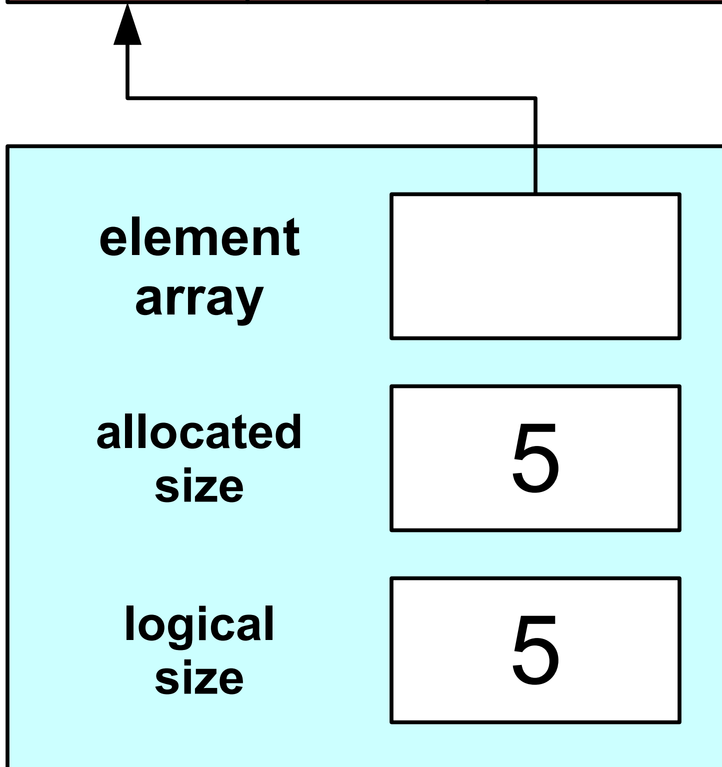
An Initial Idea



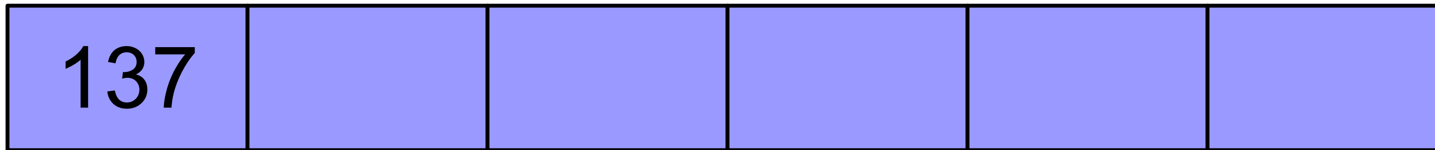
An Initial Idea

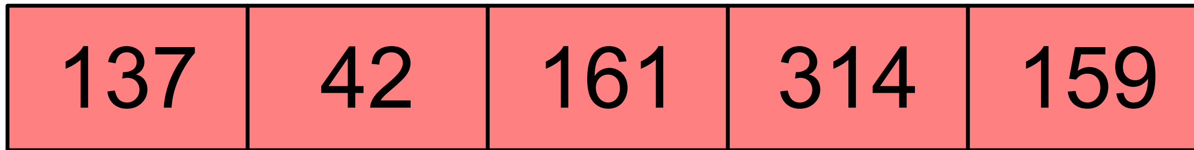


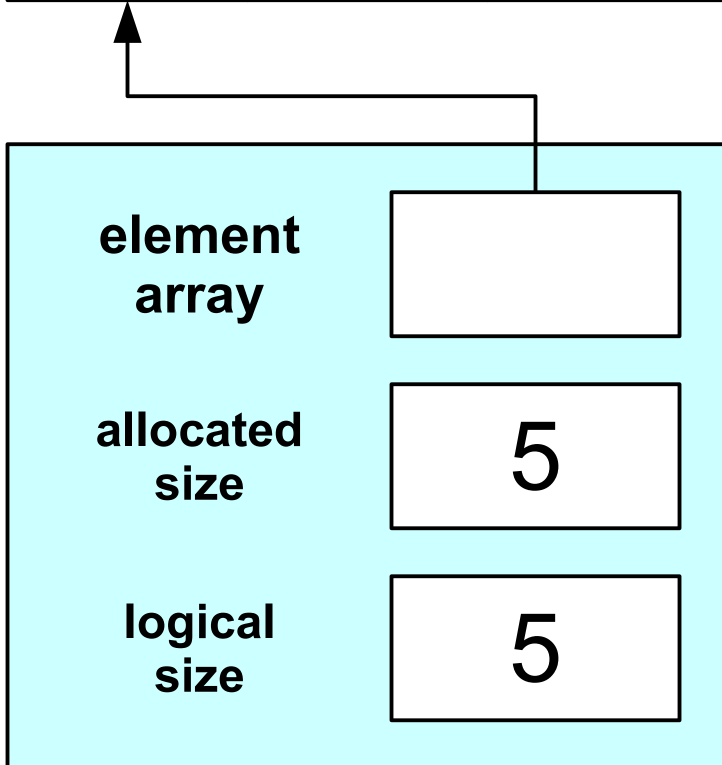




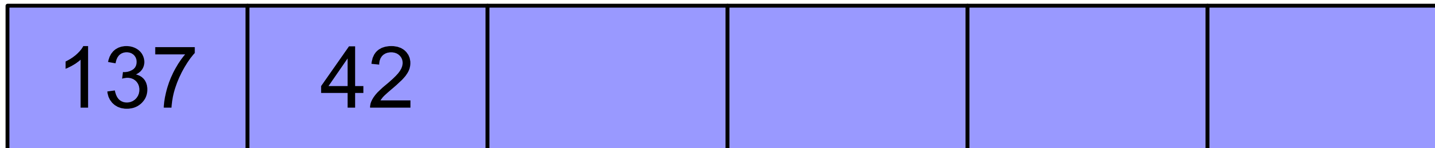
An Initial Idea

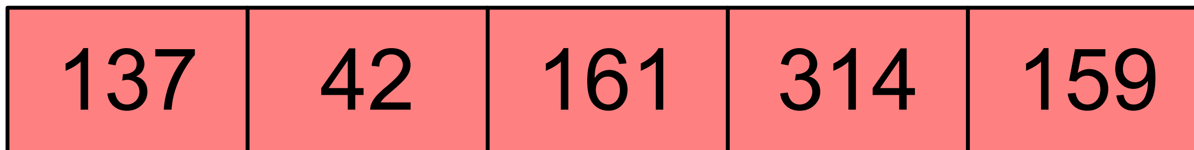


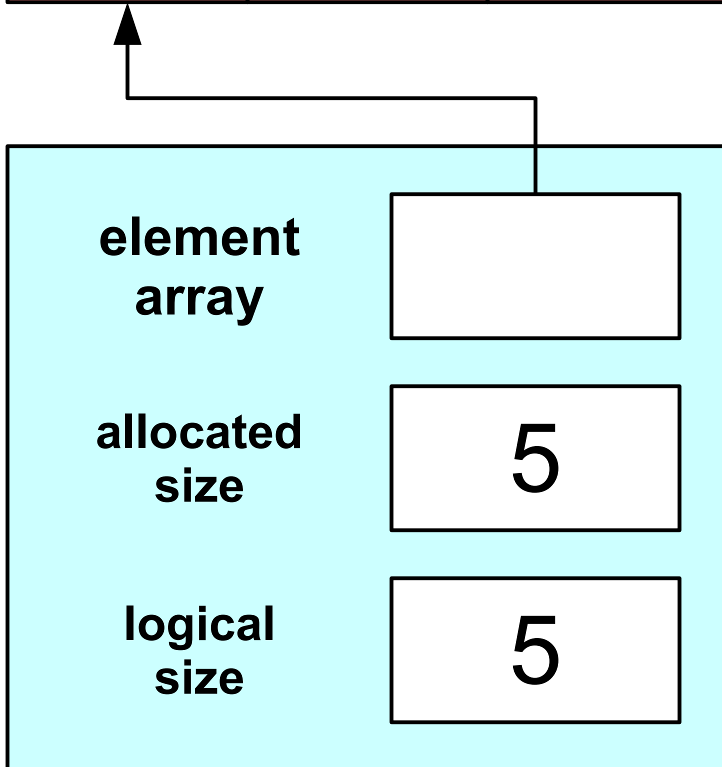




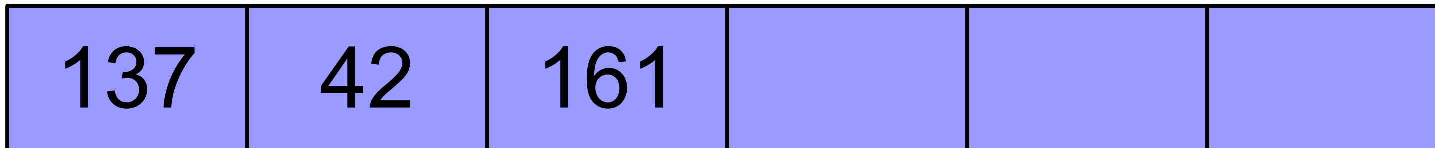
An Initial Idea

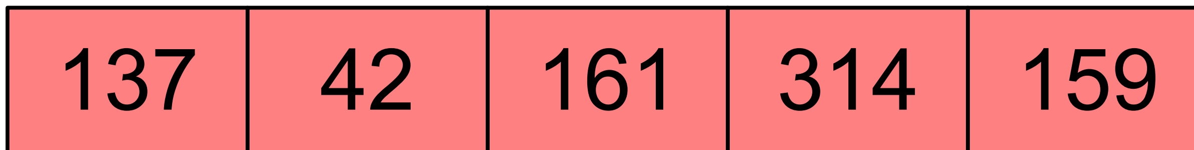


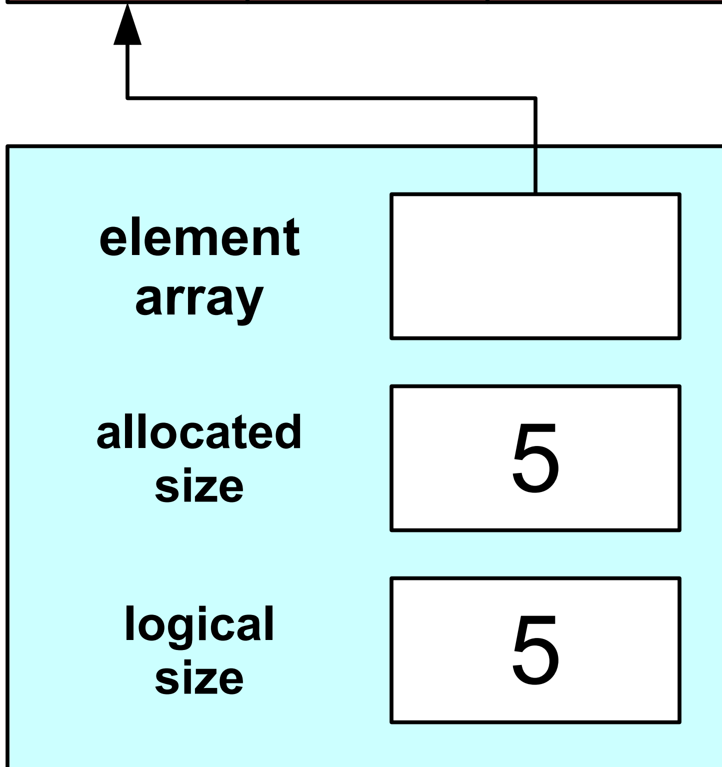




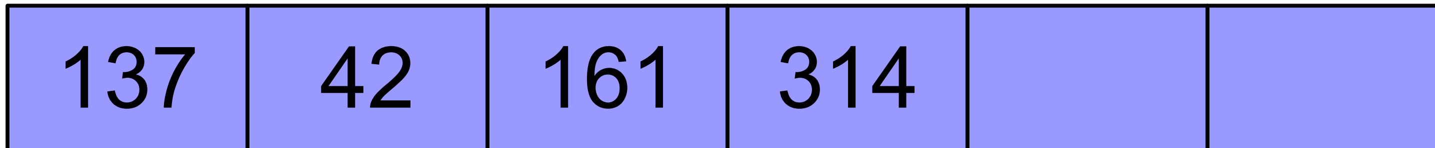
An Initial Idea

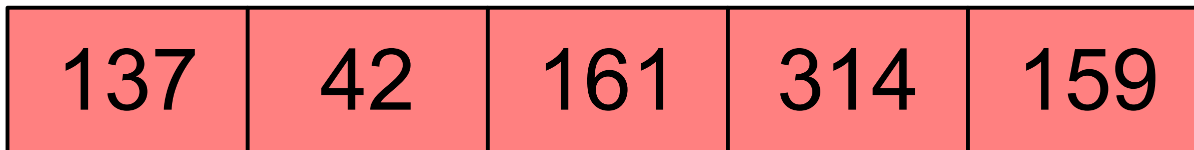


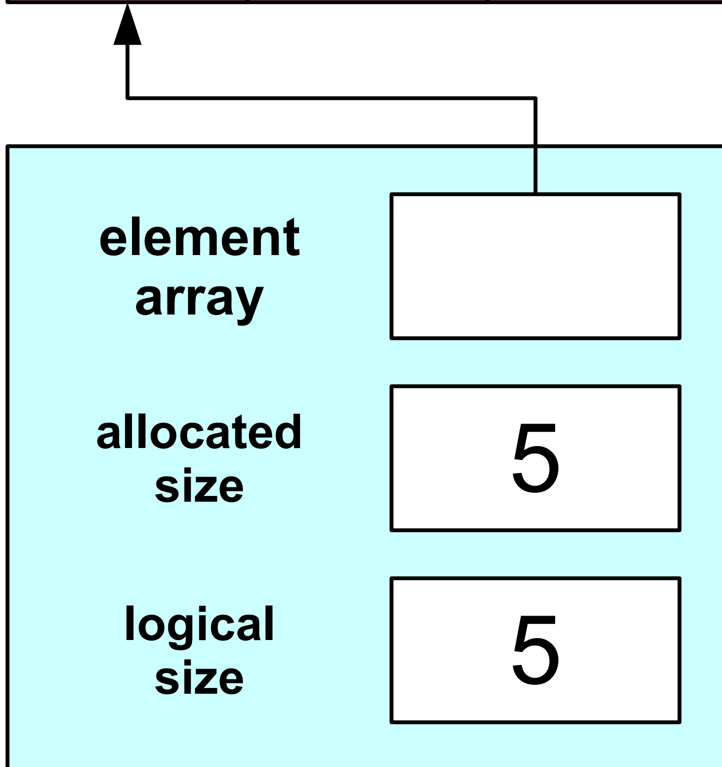




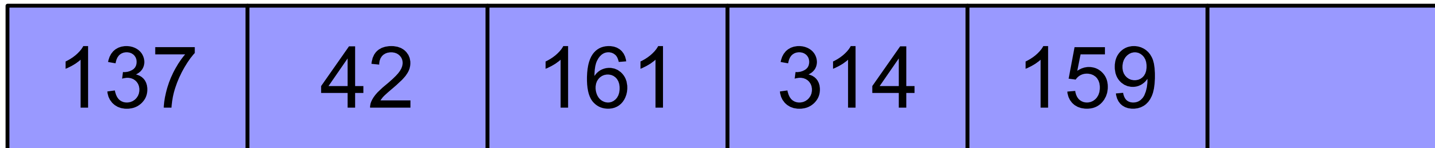
An Initial Idea

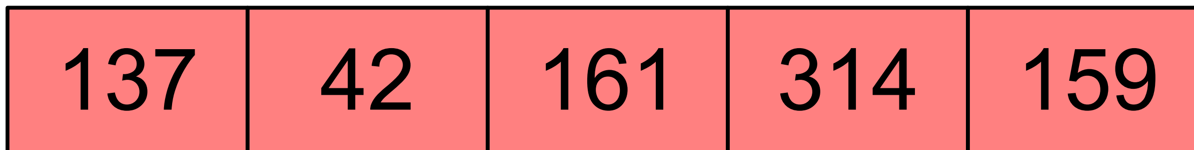


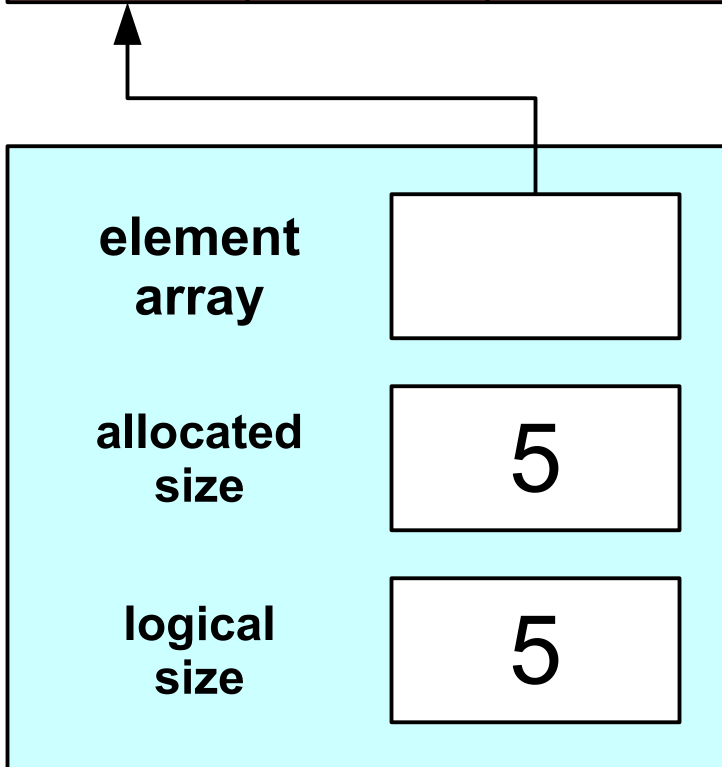




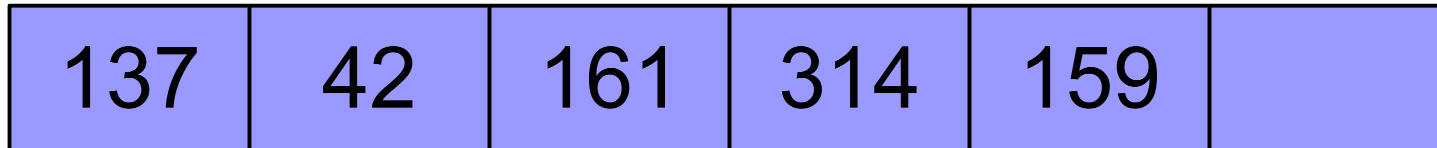
An Initial Idea

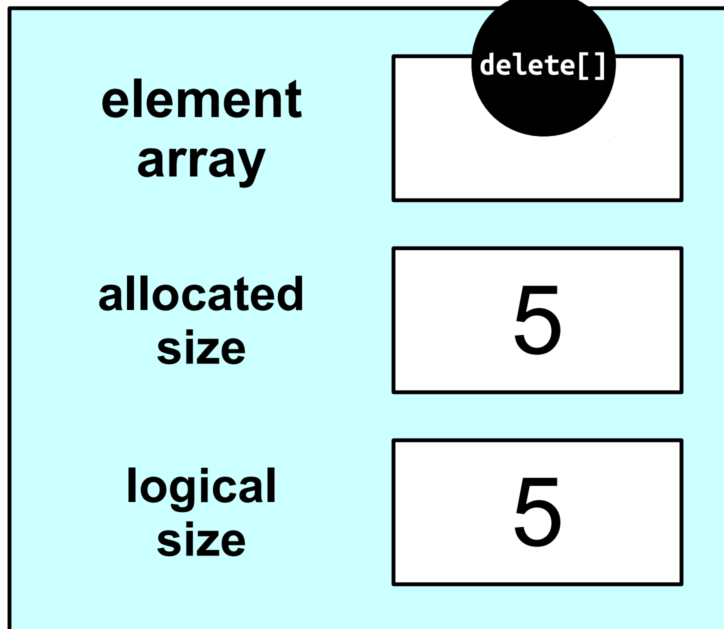




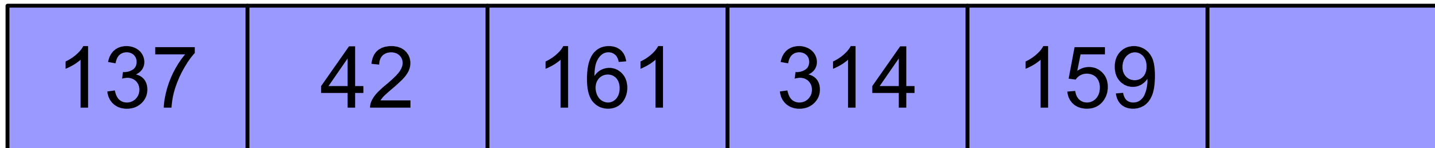


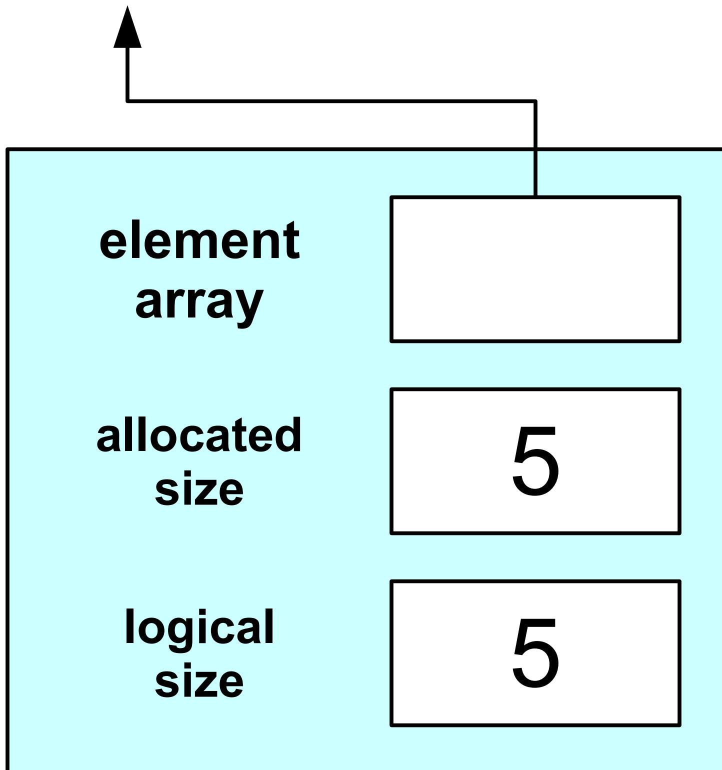
An Initial Idea



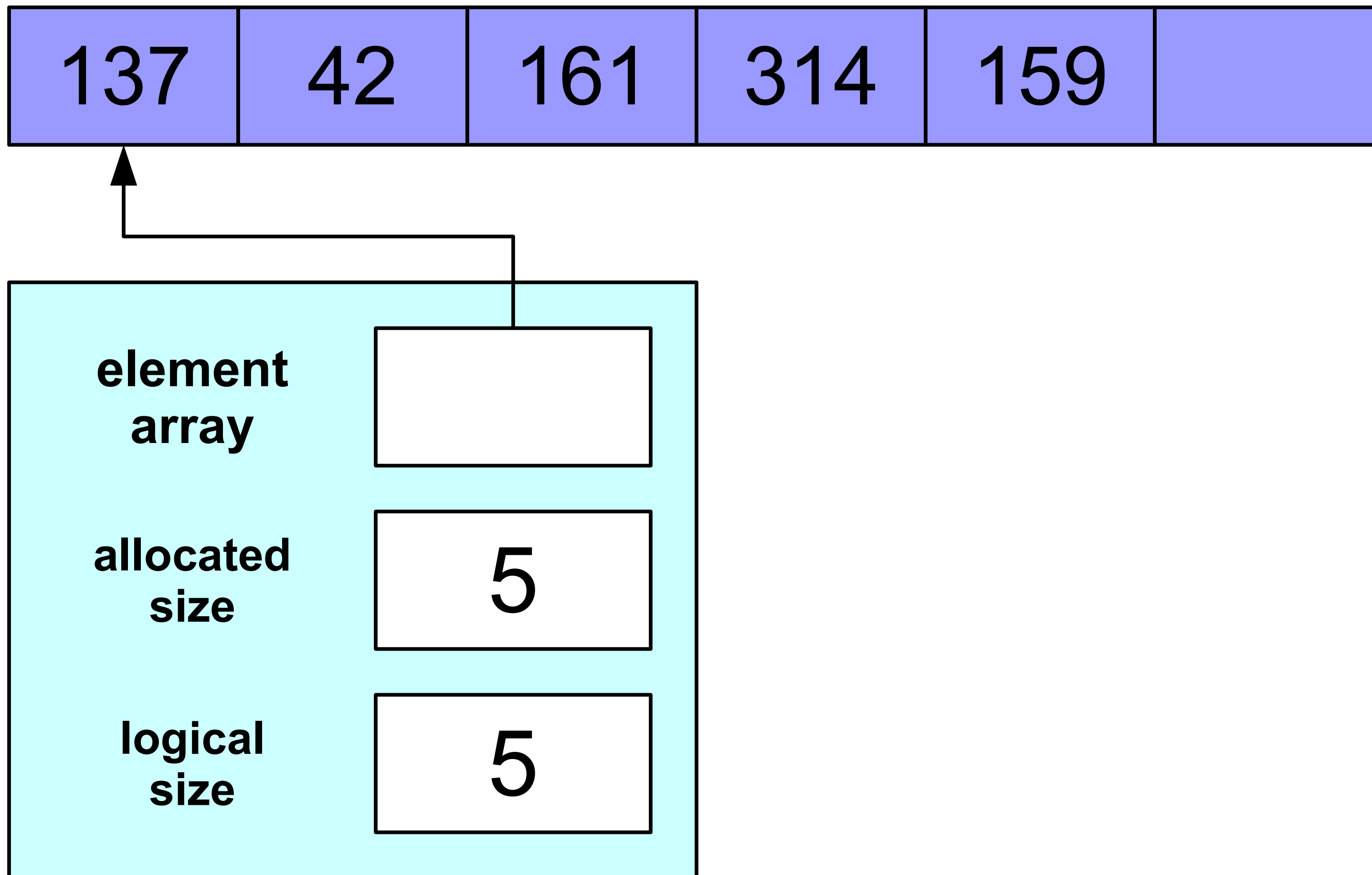


An Initial Idea

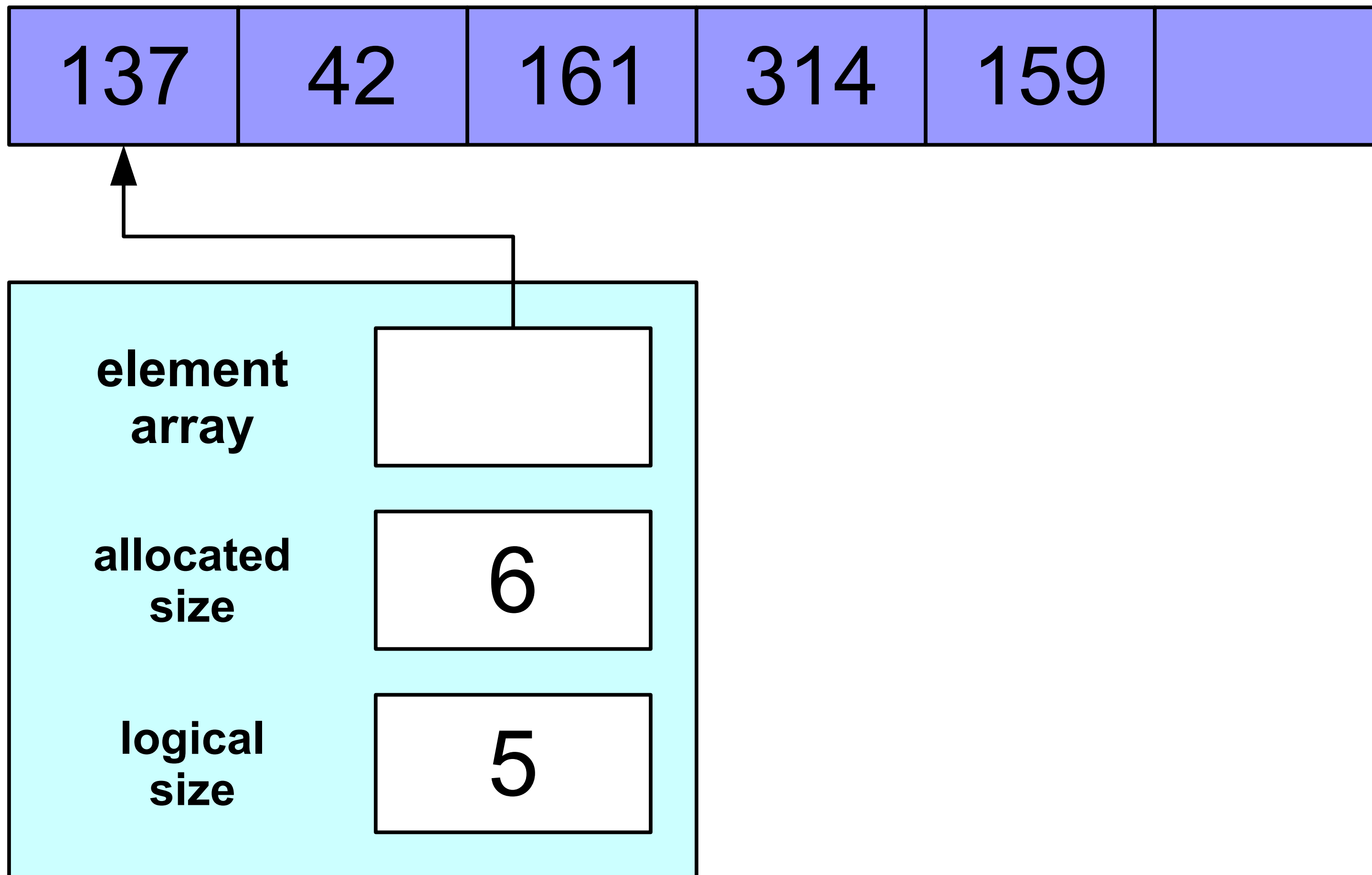




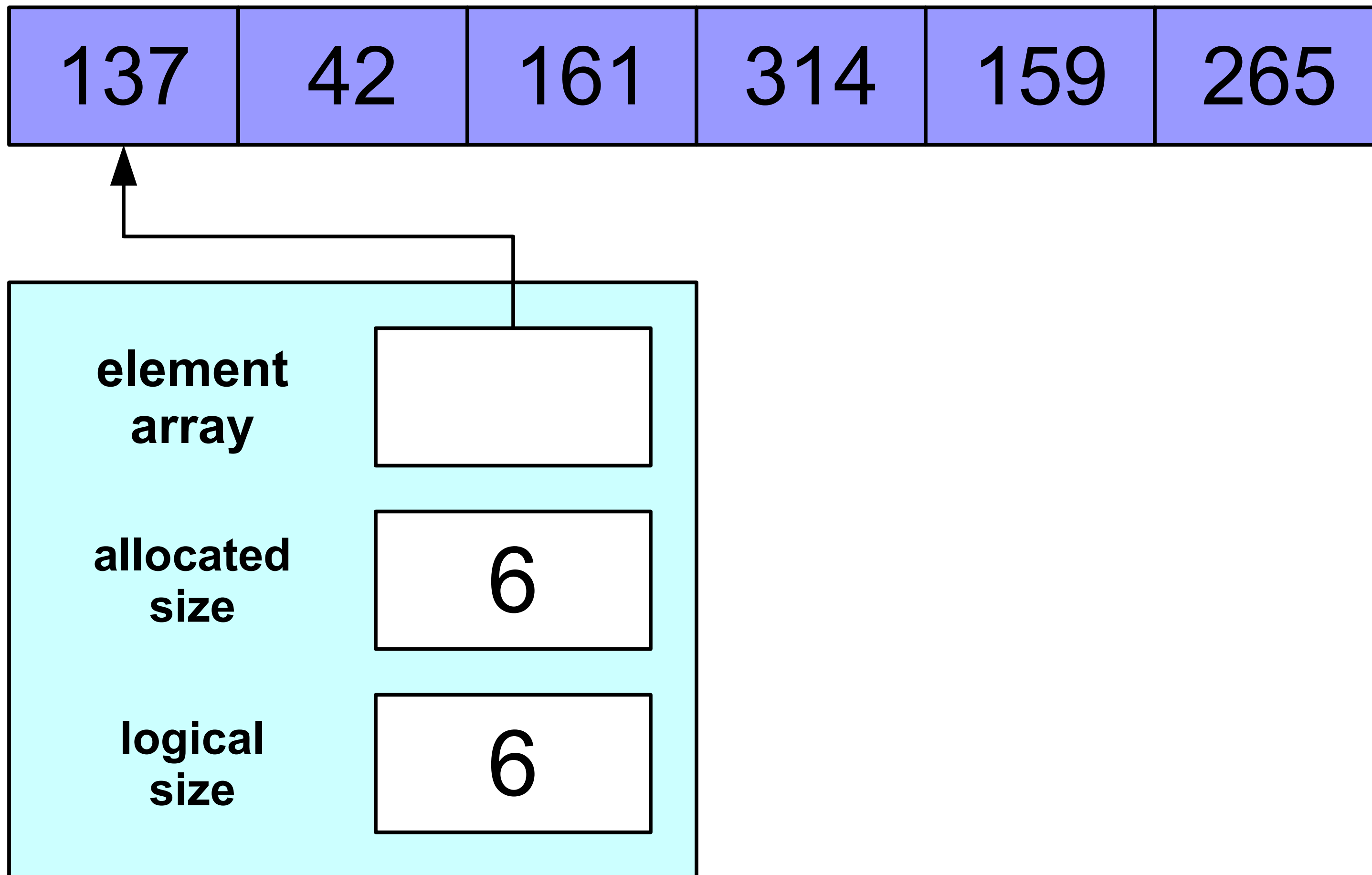
An Initial Idea



An Initial Idea



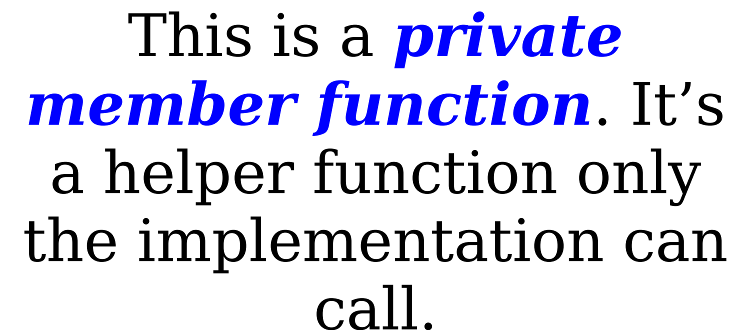
An Initial Idea



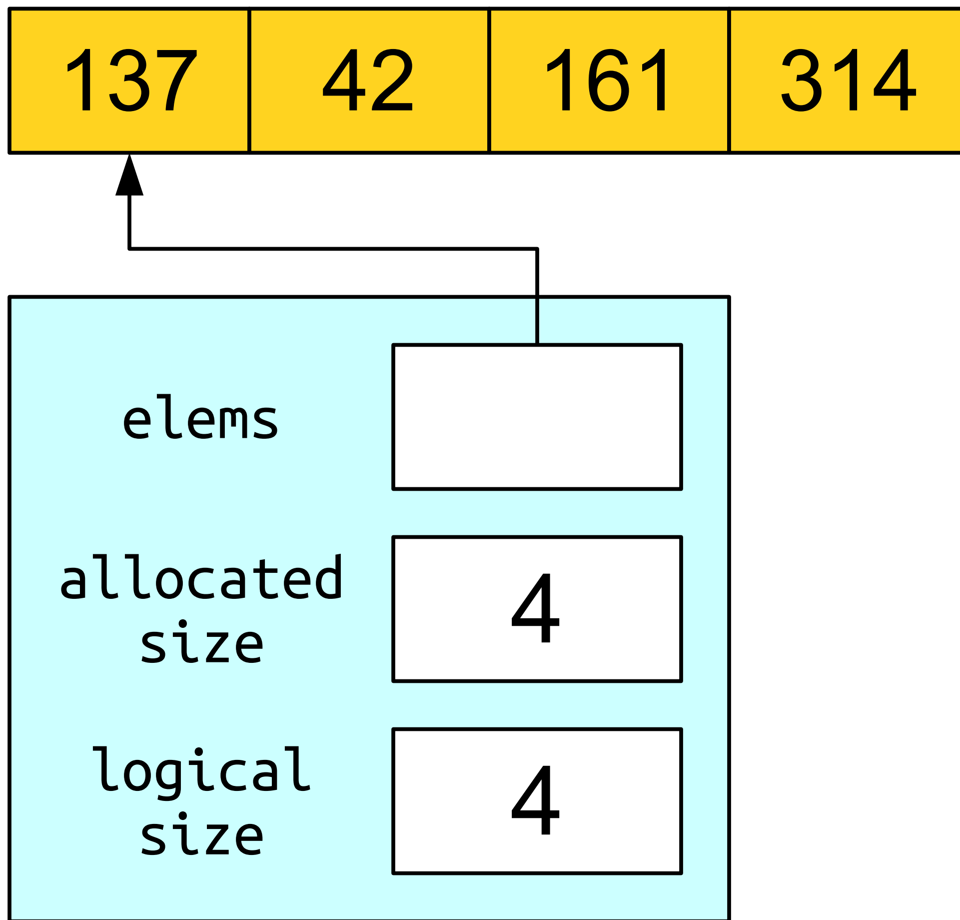
Ready… set… grow!

```cpp
class OurStack {
public:
    OurStack();
    ~OurStack();

    void push(int value);
    int  pop();
    int  peek() const;

    int  size() const;
    bool isEmpty() const;

private:


    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

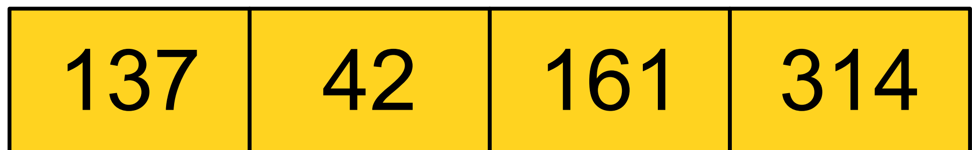
```
class OurStack {
public:
    OurStack();
    ~OurStack();

    void push(int value);
    int  pop();
    int  peek() const;

    int  size() const;
    bool isEmpty() const;

private:
    void grow();

    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

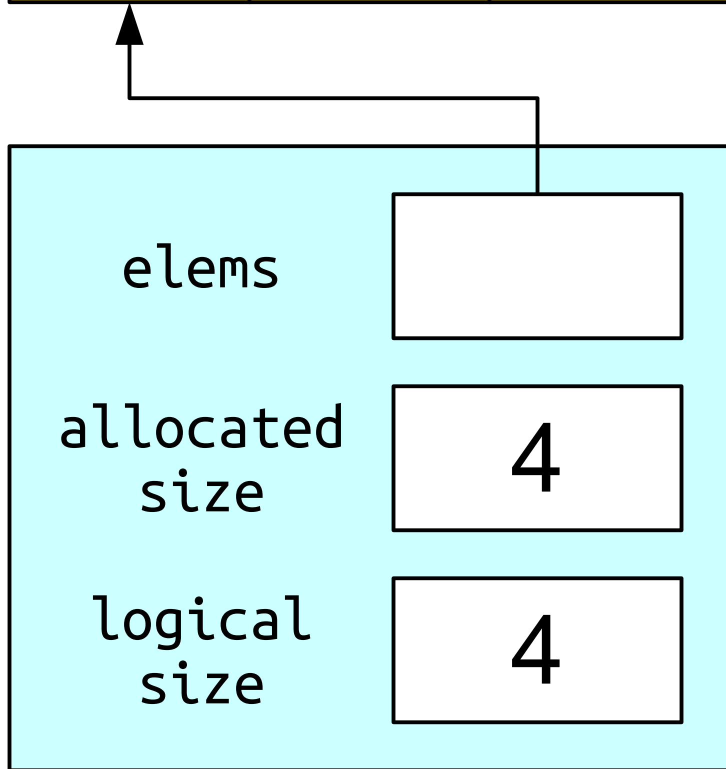


An Initial Idea



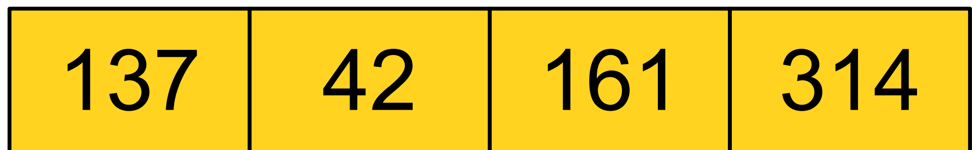
An Initial Idea

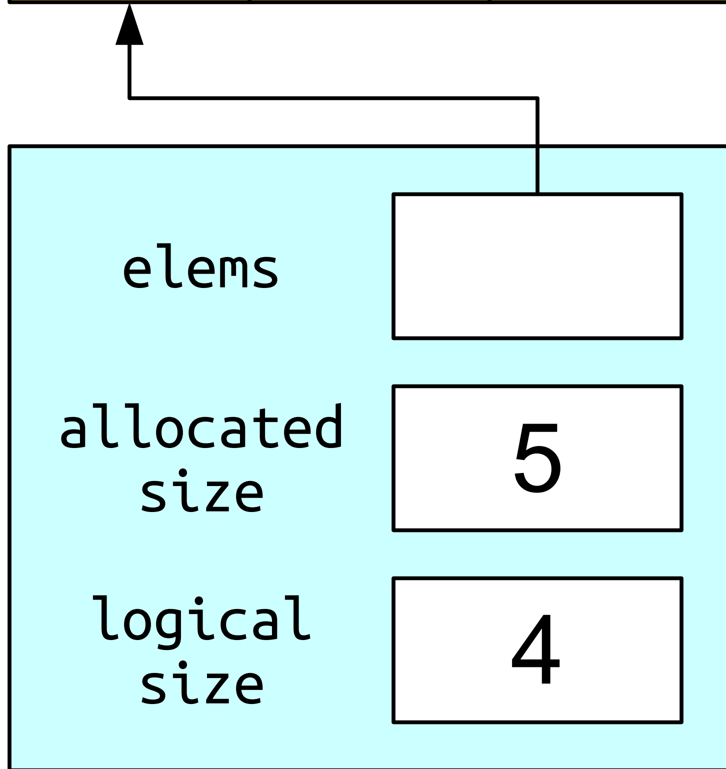





```
void OurStack::grow() {
    allocatedSize++;



}
```

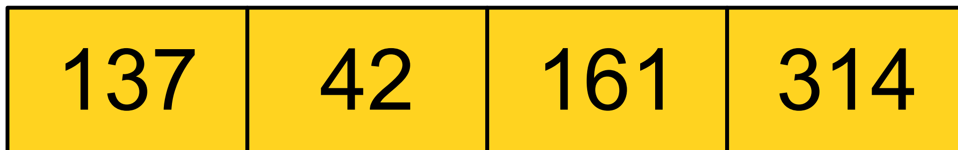
An Initial Idea

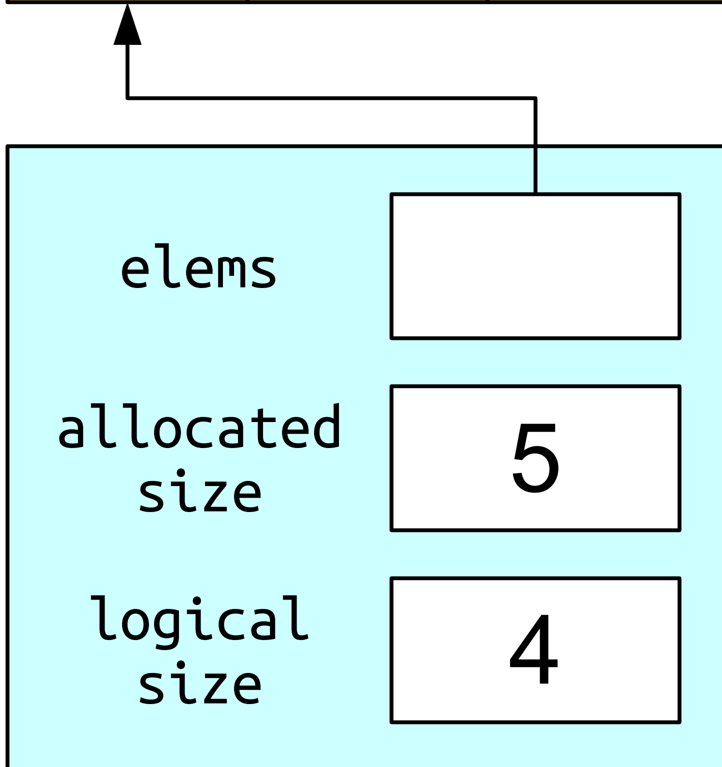





```
void OurStack::grow() {
    allocatedSize++;



}
```

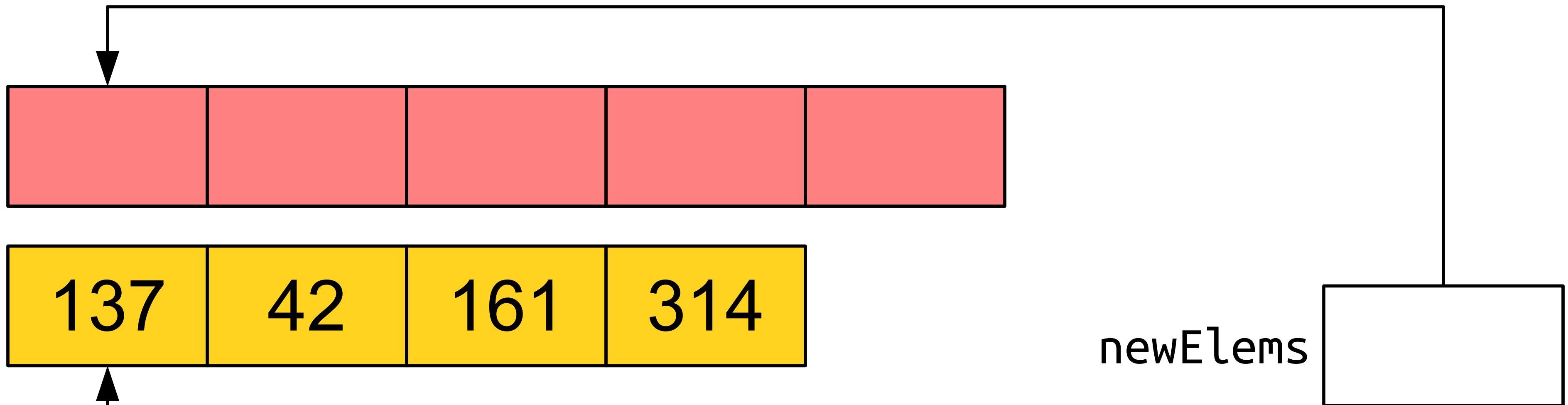
An Initial Idea

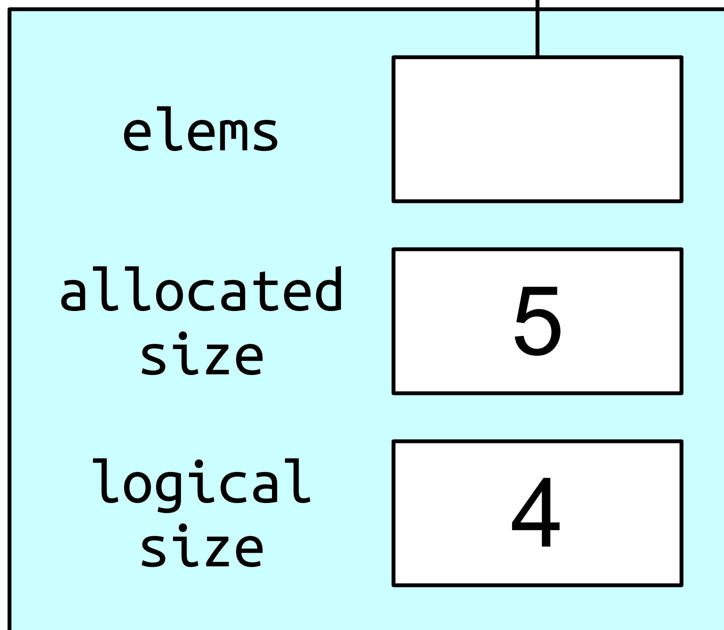





```
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];






}
```

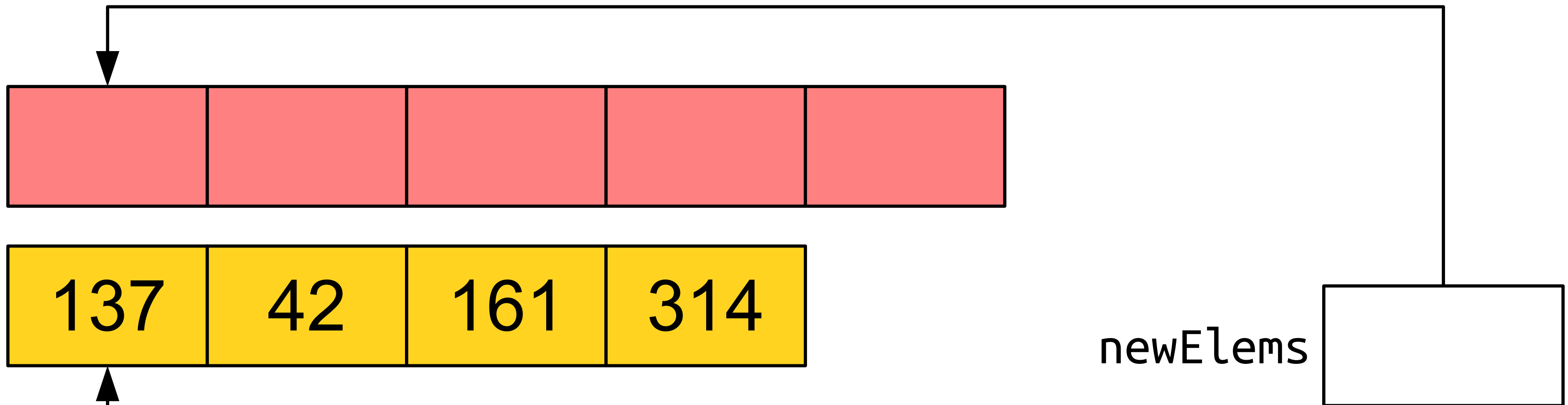
An Initial Idea

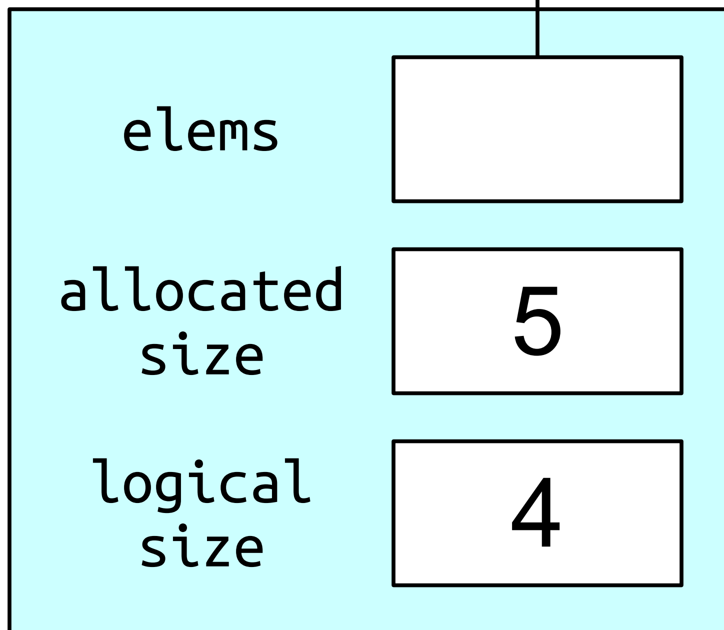





```
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];


}
```

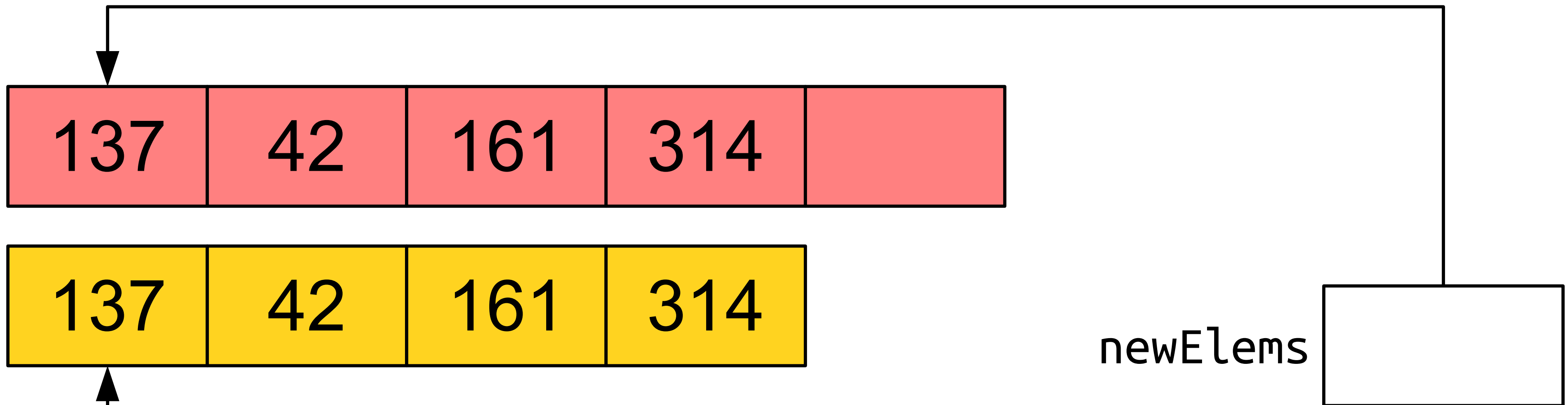
An Initial Idea

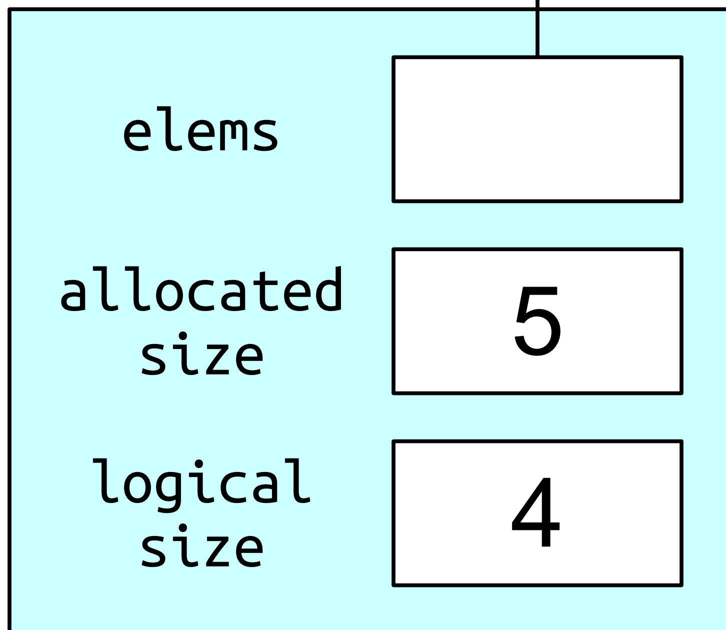





```
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

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



}
```

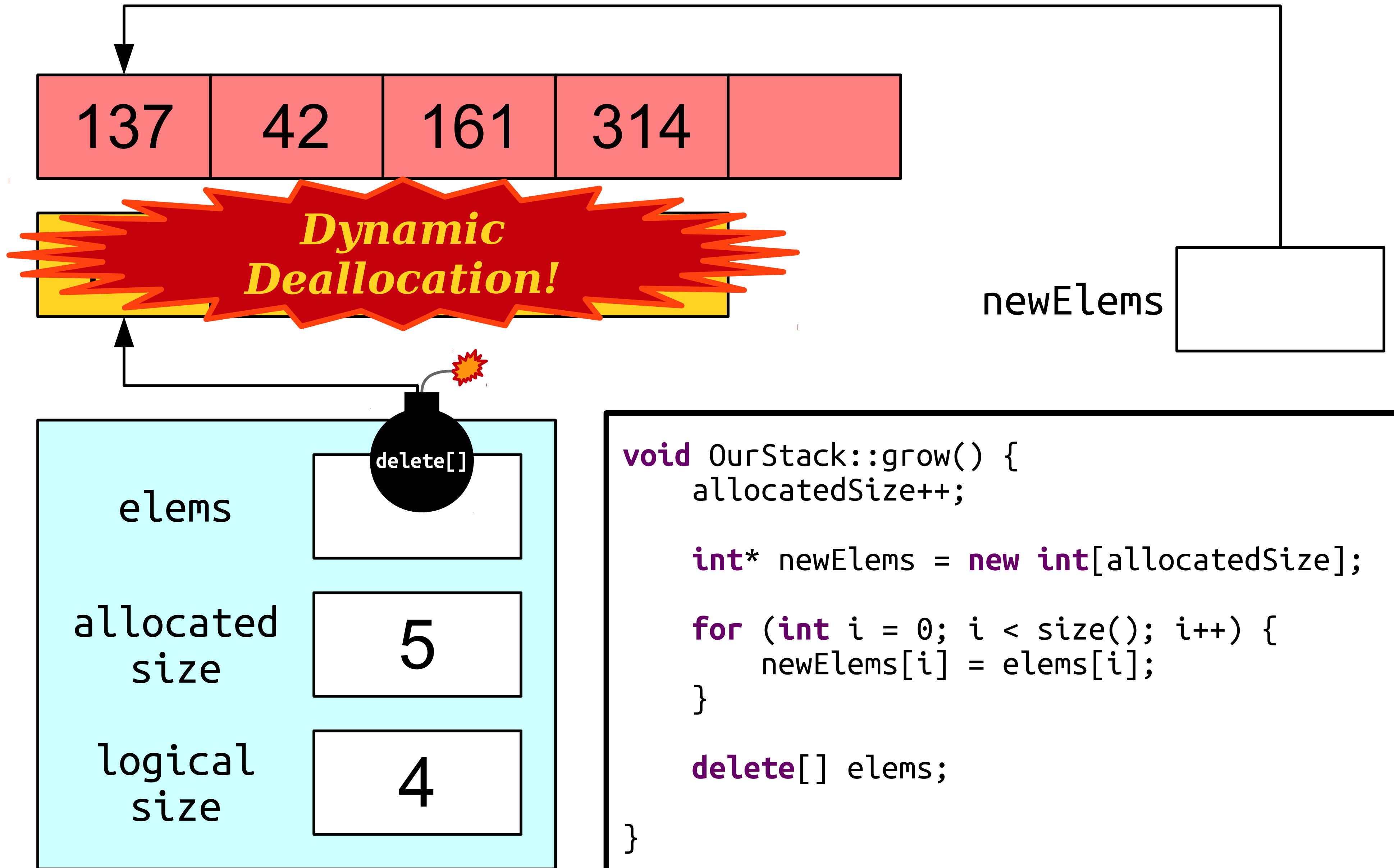
An Initial Idea



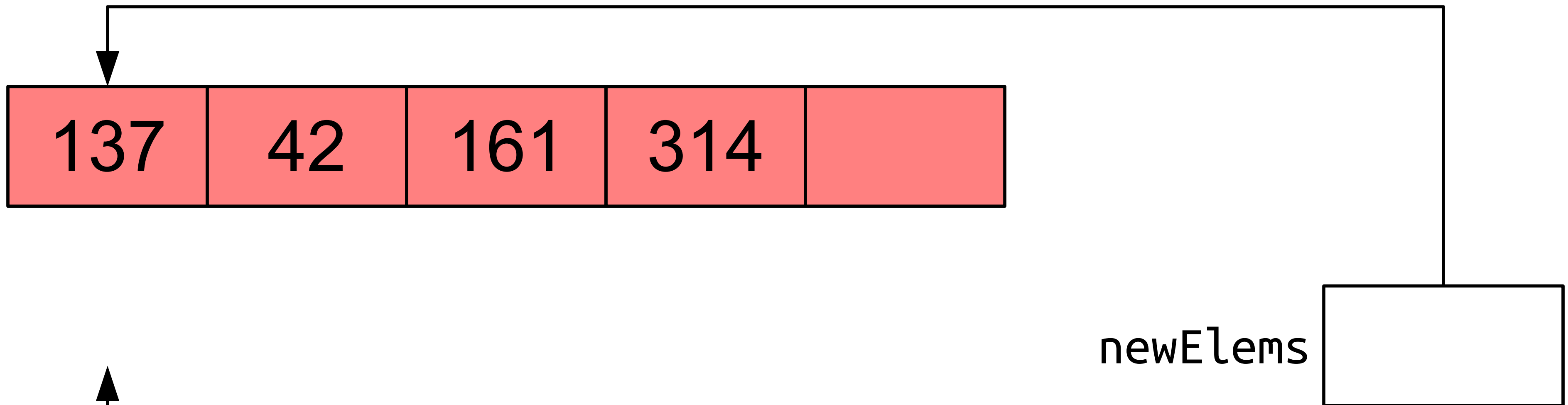


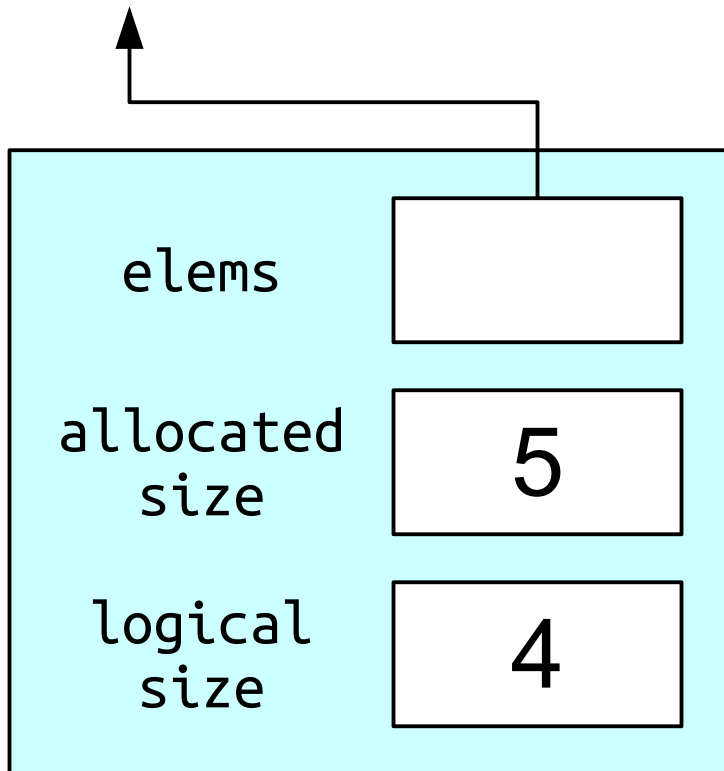

```
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

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



}
```

An Initial Idea



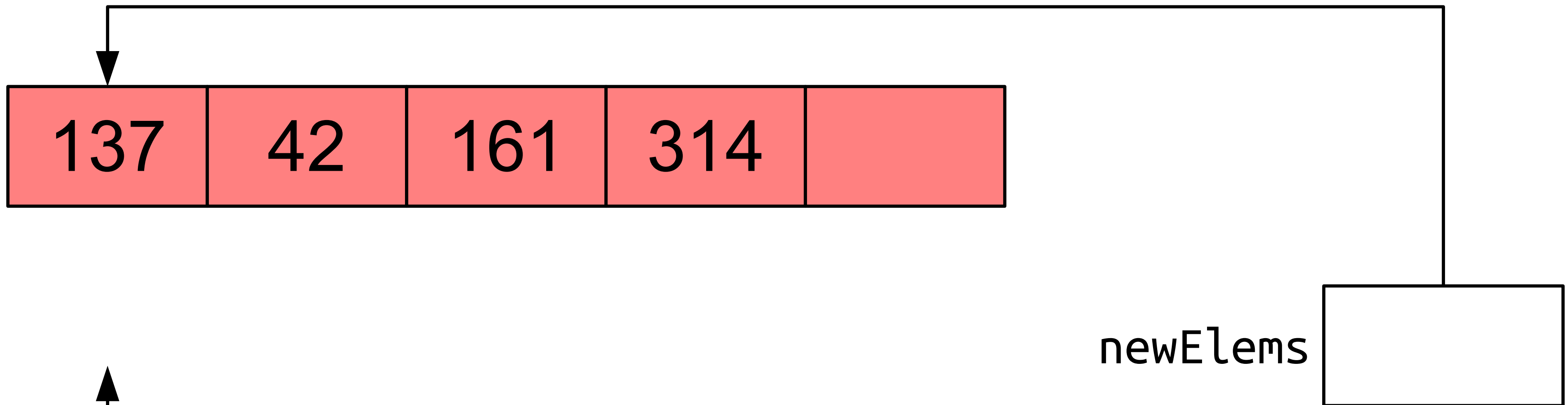
An Initial Idea

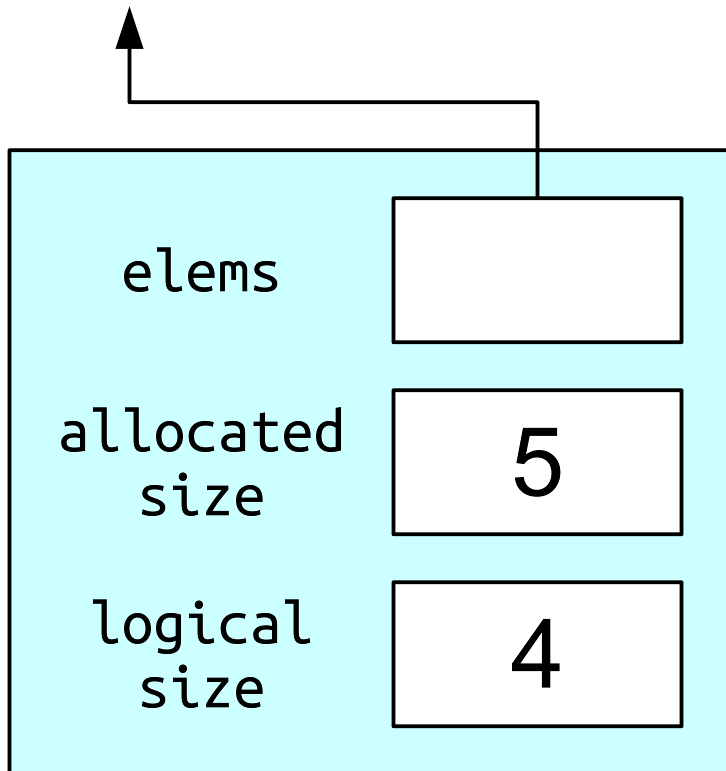





```
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

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

    delete[] elems;

}
```

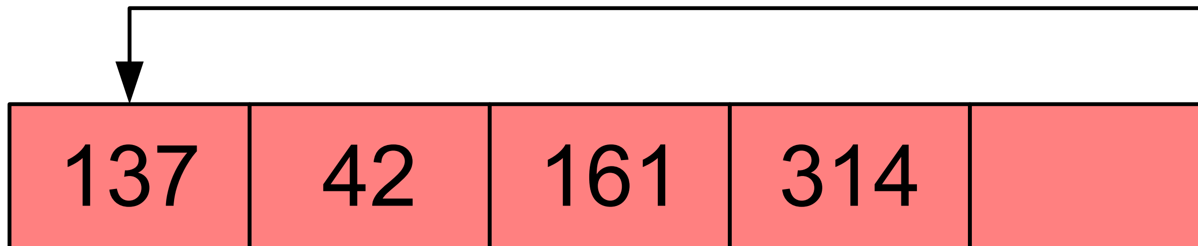
An Initial Idea

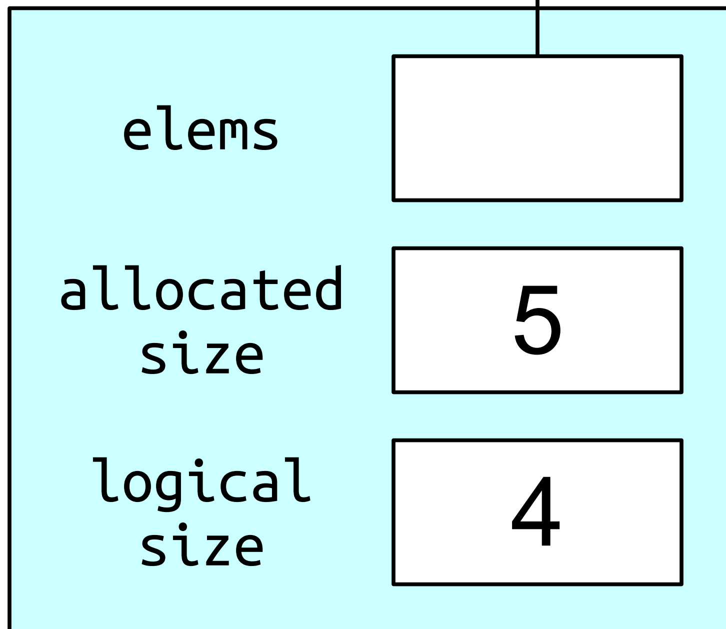





```
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

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

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

An Initial Idea





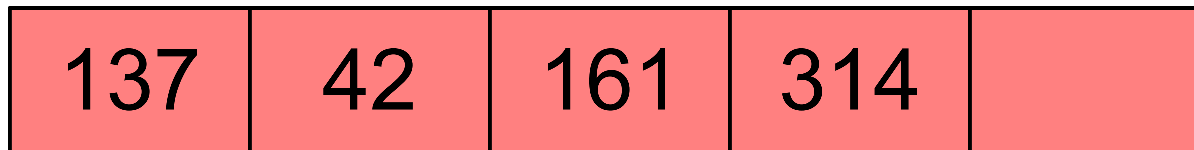

```
vo

    int* newElems = new int[allocatedSize];

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

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

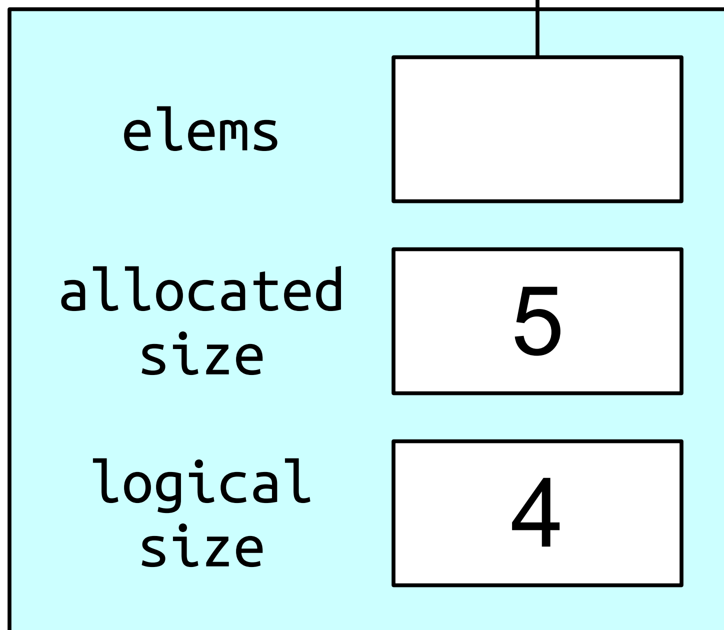
If a and b are pointers, then

```
a = b;
```

makes a point where b is pointing.

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. As usual, let *n* denote the number of items in the stack when the operation is performed.
  - `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. As usual, let *n* denote the number of items in the stack when the operation is performed.
  - `size`: **O(1)**
  - `isEmpty`: **O(1)**
  - `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. As usual, let n denote the number of items in the stack when the operation is performed.
  - `size`: **O(1)**
  - `isEmpty`: **O(1)**
  - `push`: **O(n)**
  - `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. As usual, let n denote the number of items in the stack when the operation is performed.
  - `size`: **O(1)**
  - `isEmpty`: **O(1)**
  - `push`: **O(*n*)**
  - `pop`: **O(1)**
  - `peek`: **O(1)**

What This Means

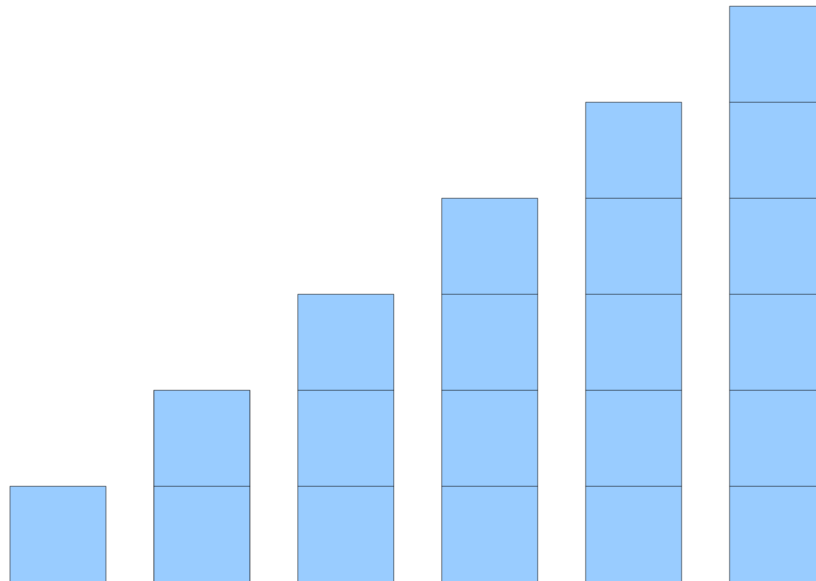
- What is the complexity of pushing n elements and then popping them?

What This Means

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

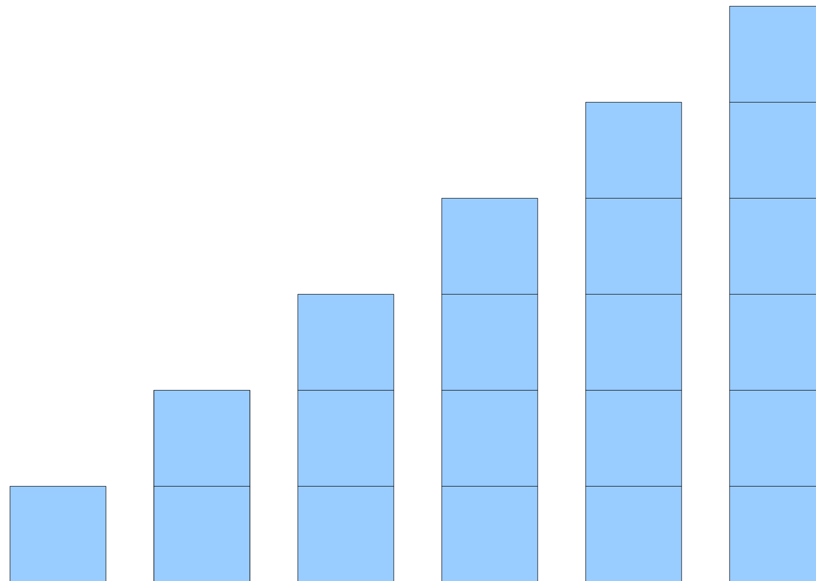
What This Means

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



What This Means

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



What This Means

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

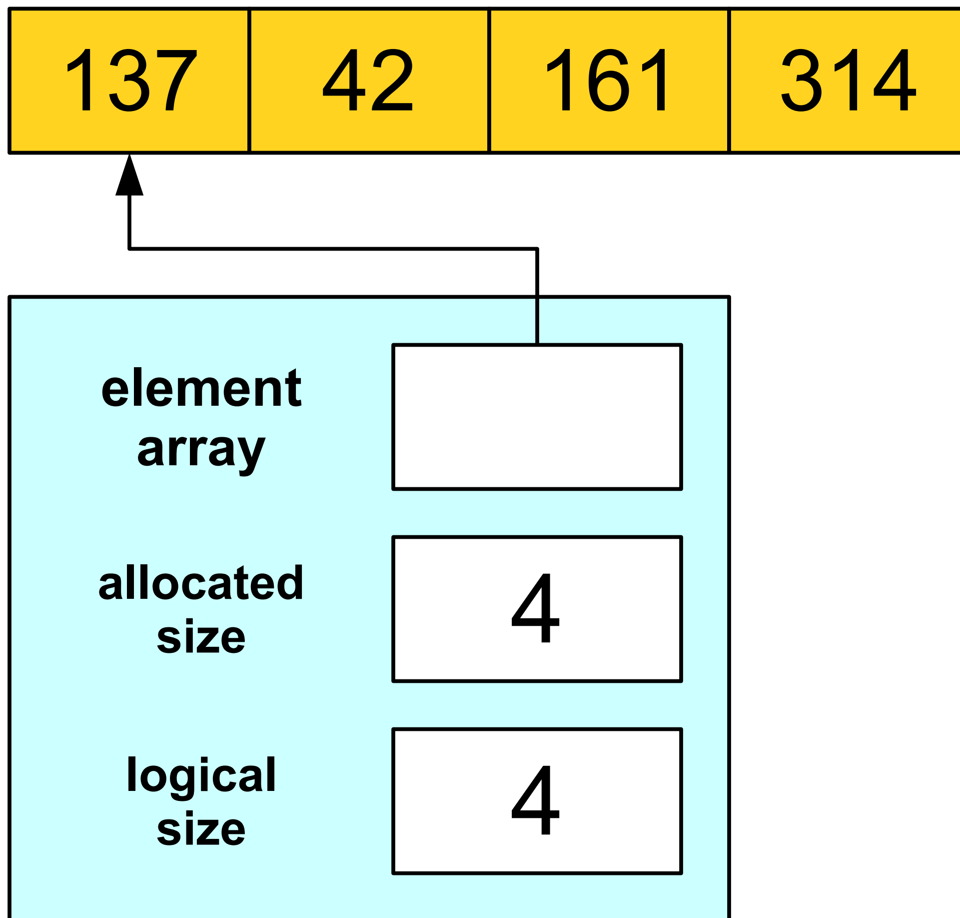
- What is the complexity of pushing n elements and then popping them?
- Cost of the pushes:
  - $1 + 2 + 3 + 4 + \ldots + n =$ **$O(n^2)$**
- Cost of the pops:
  - $1 + 1 + 1 + 1 + \ldots + 1 =$ **$O(n)$**
- Total cost: **$O(n^2)$**

Validating Our Model

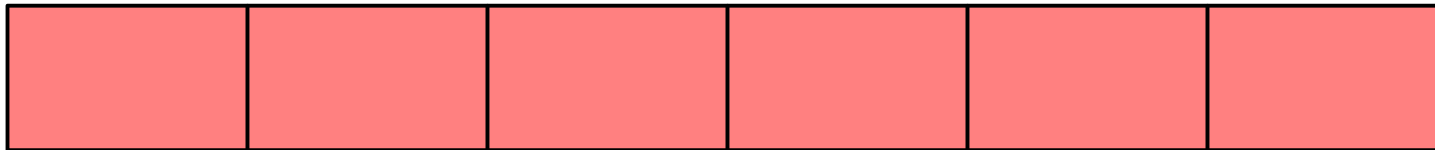
Speeding up the Stack

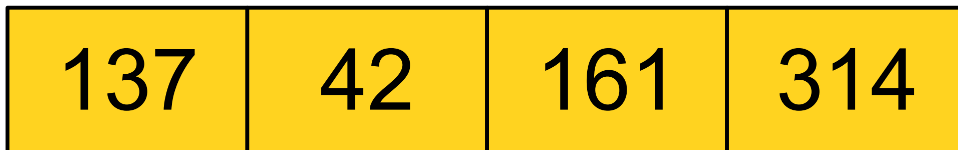
Key Idea: ***Plan for the Future***

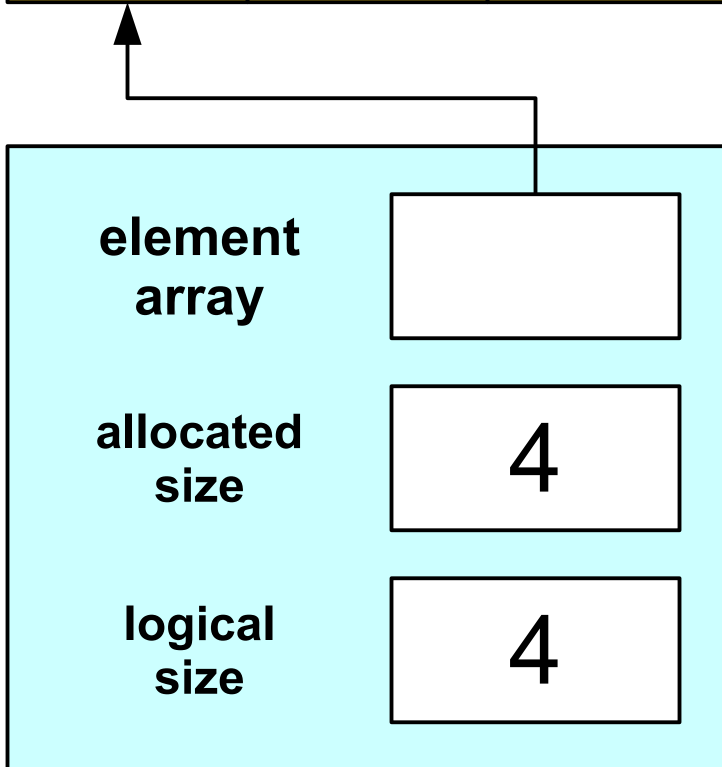
A Better Idea



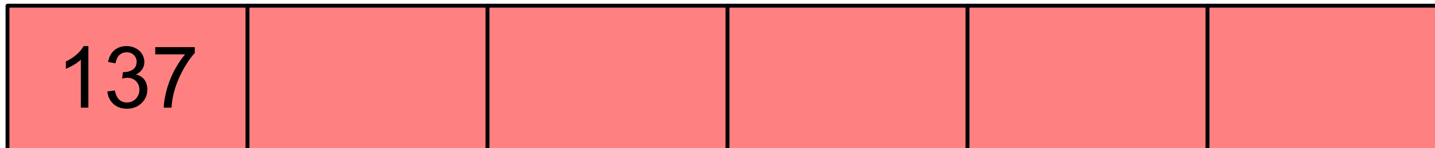
A Better Idea

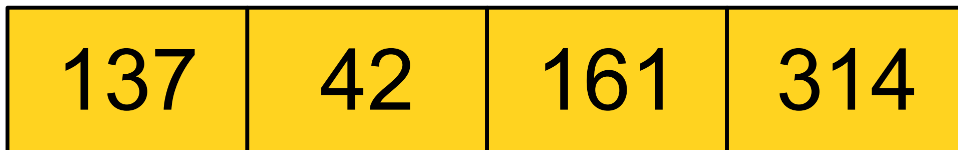


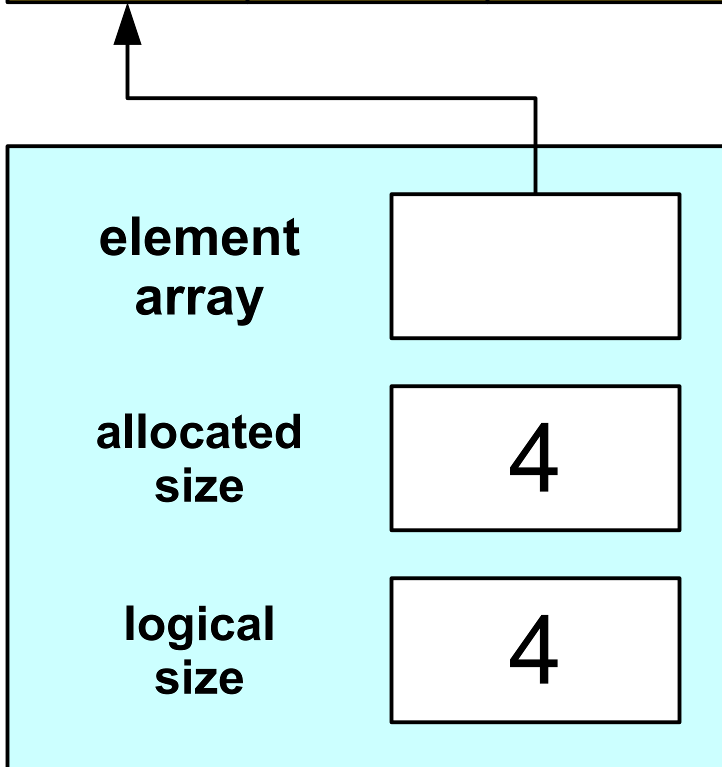




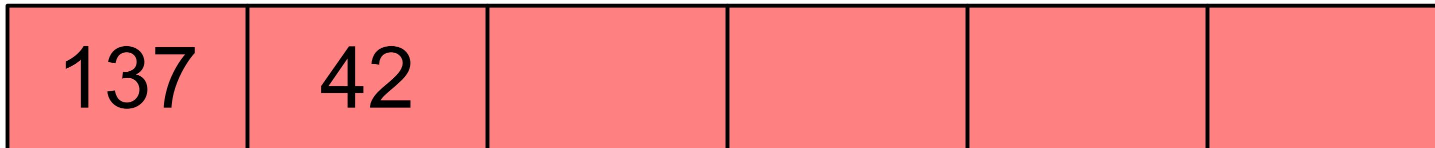
A Better Idea

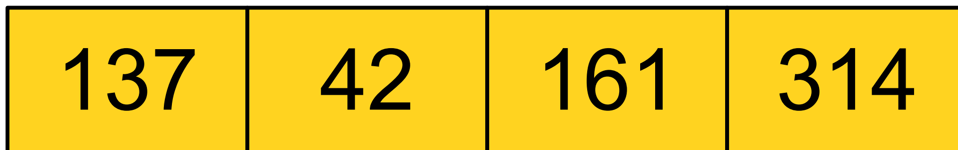


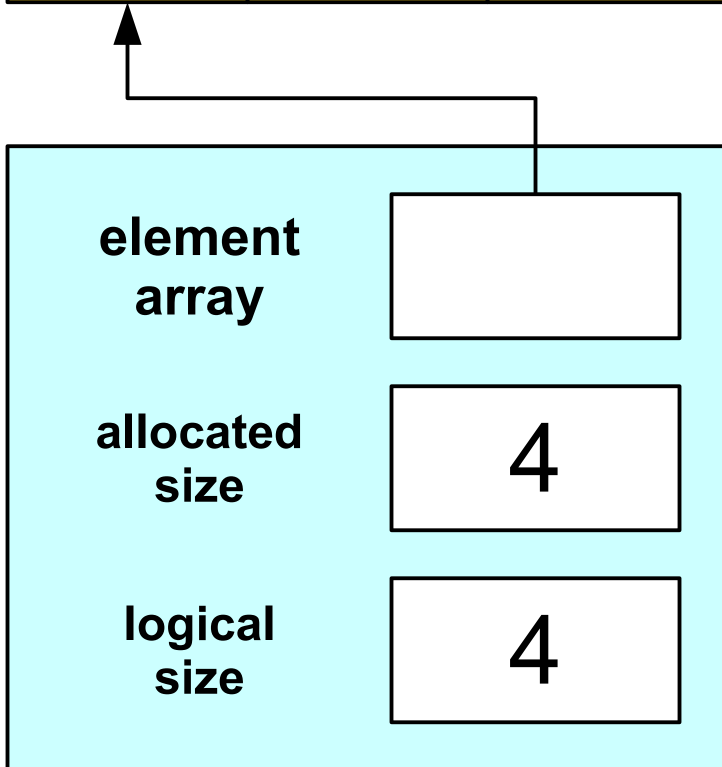




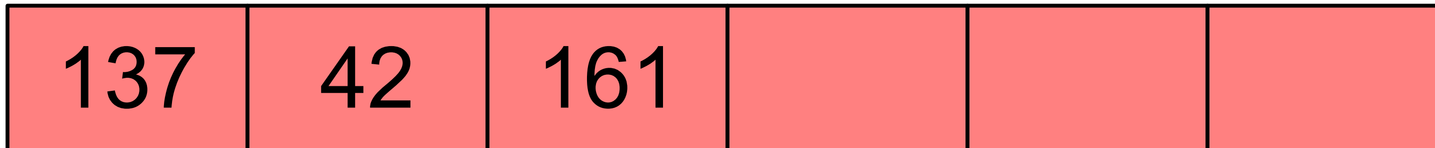
A Better Idea

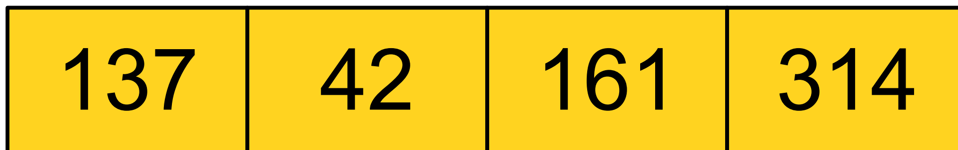


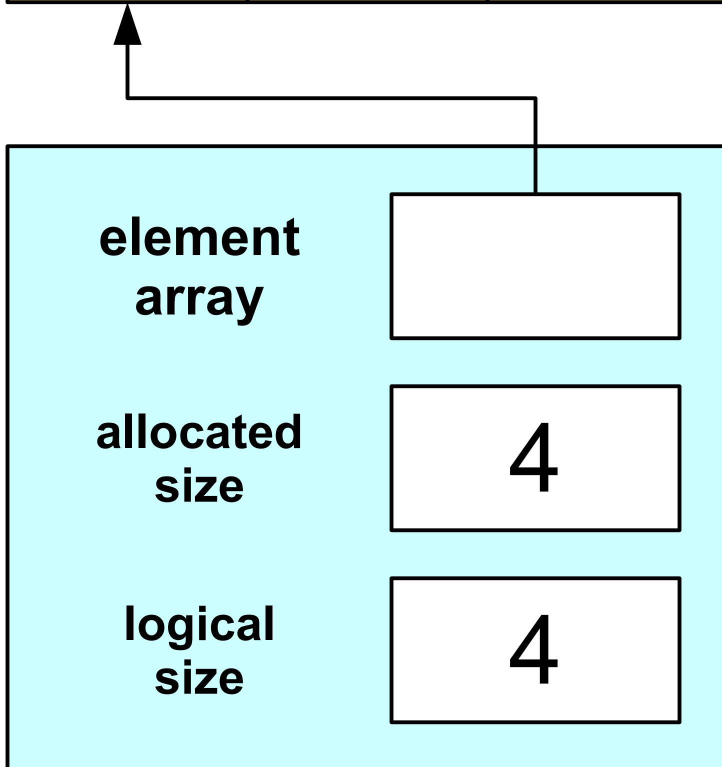




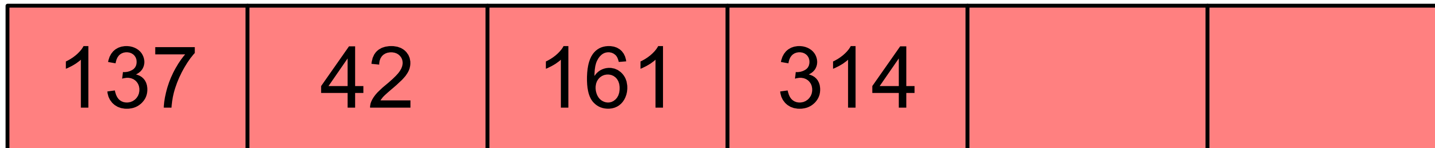
A Better Idea

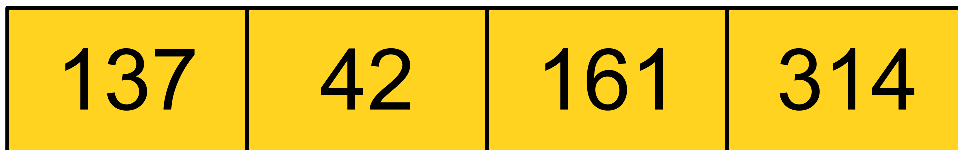


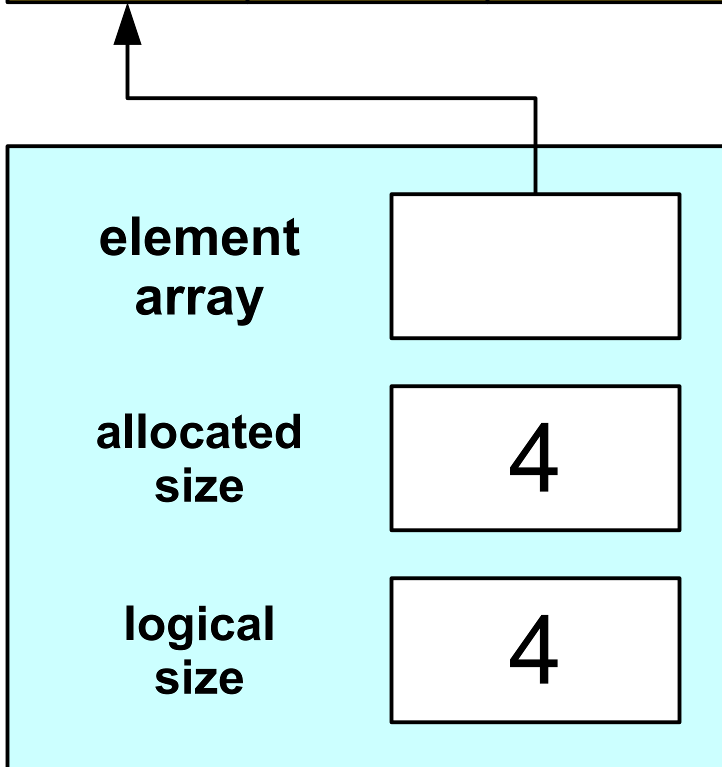




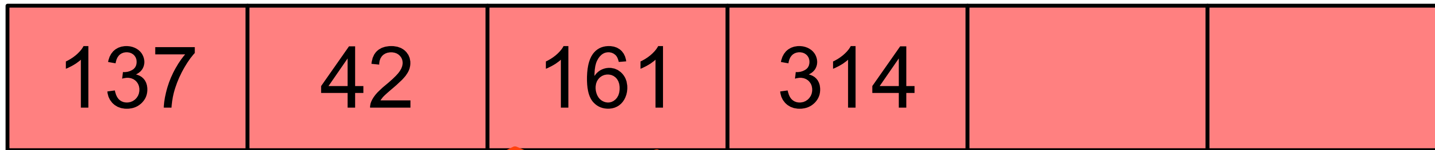
A Better Idea

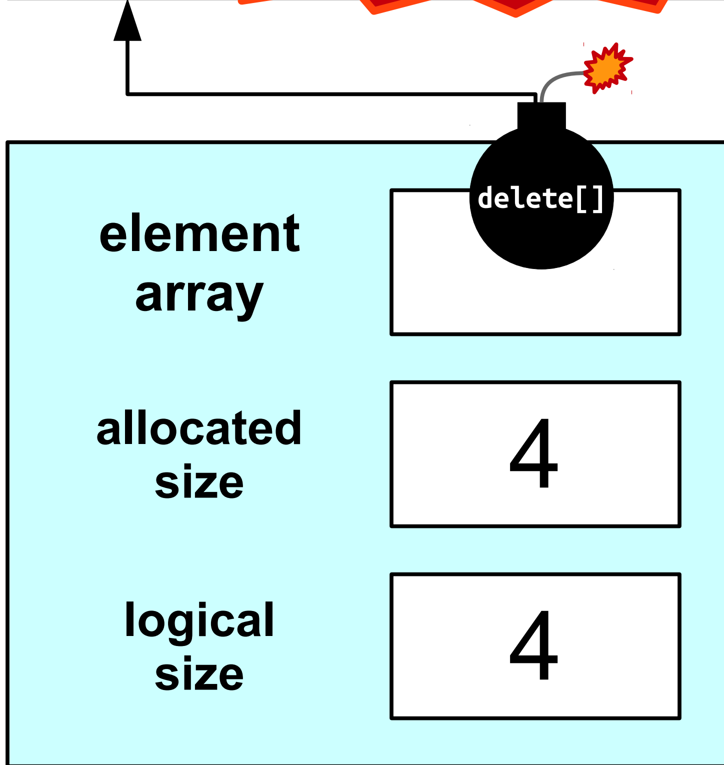




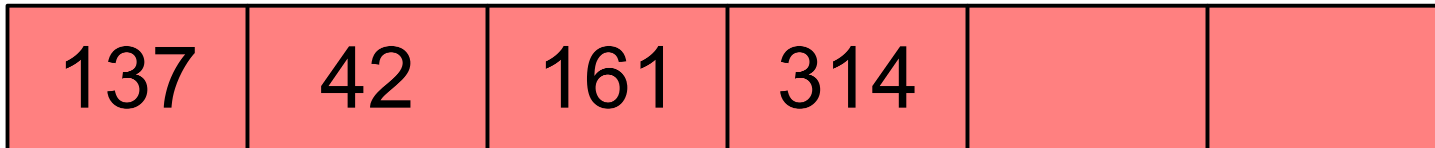


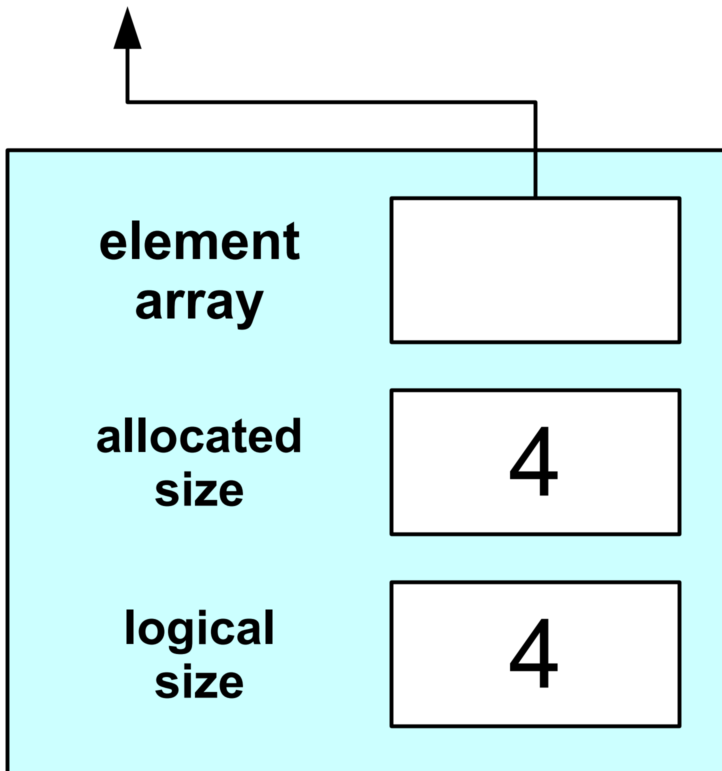
A Better Idea



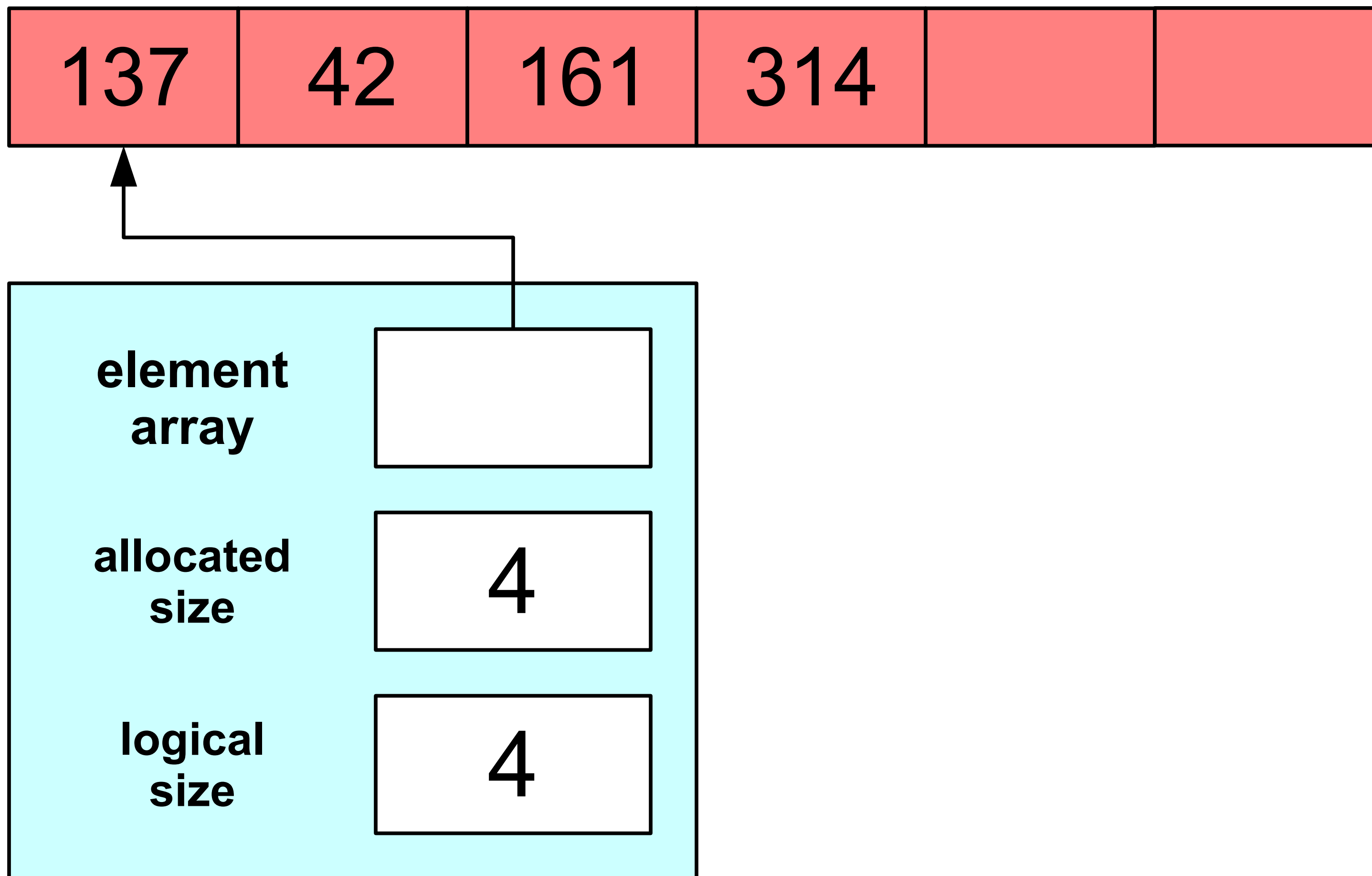


A Better Idea

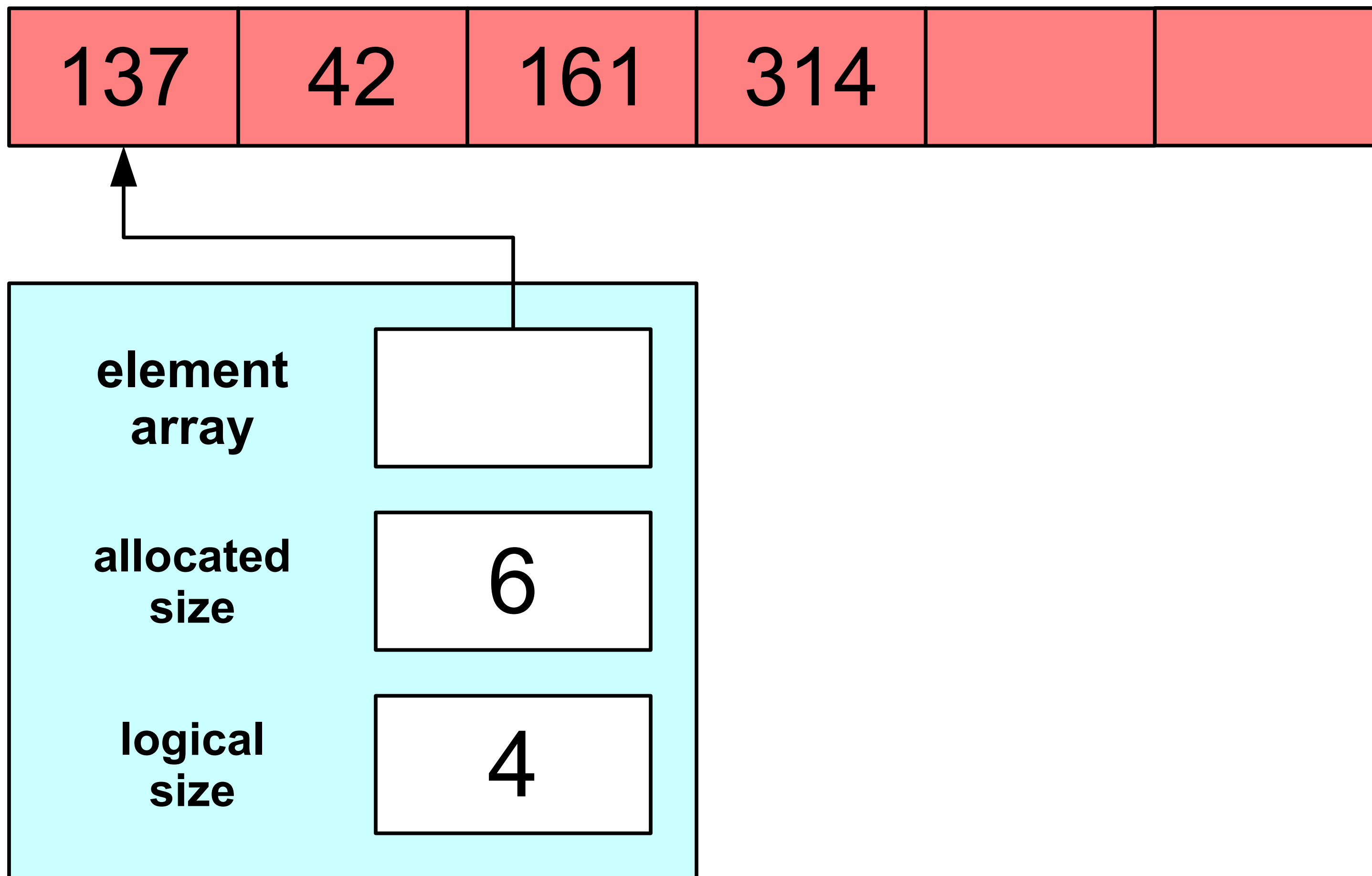




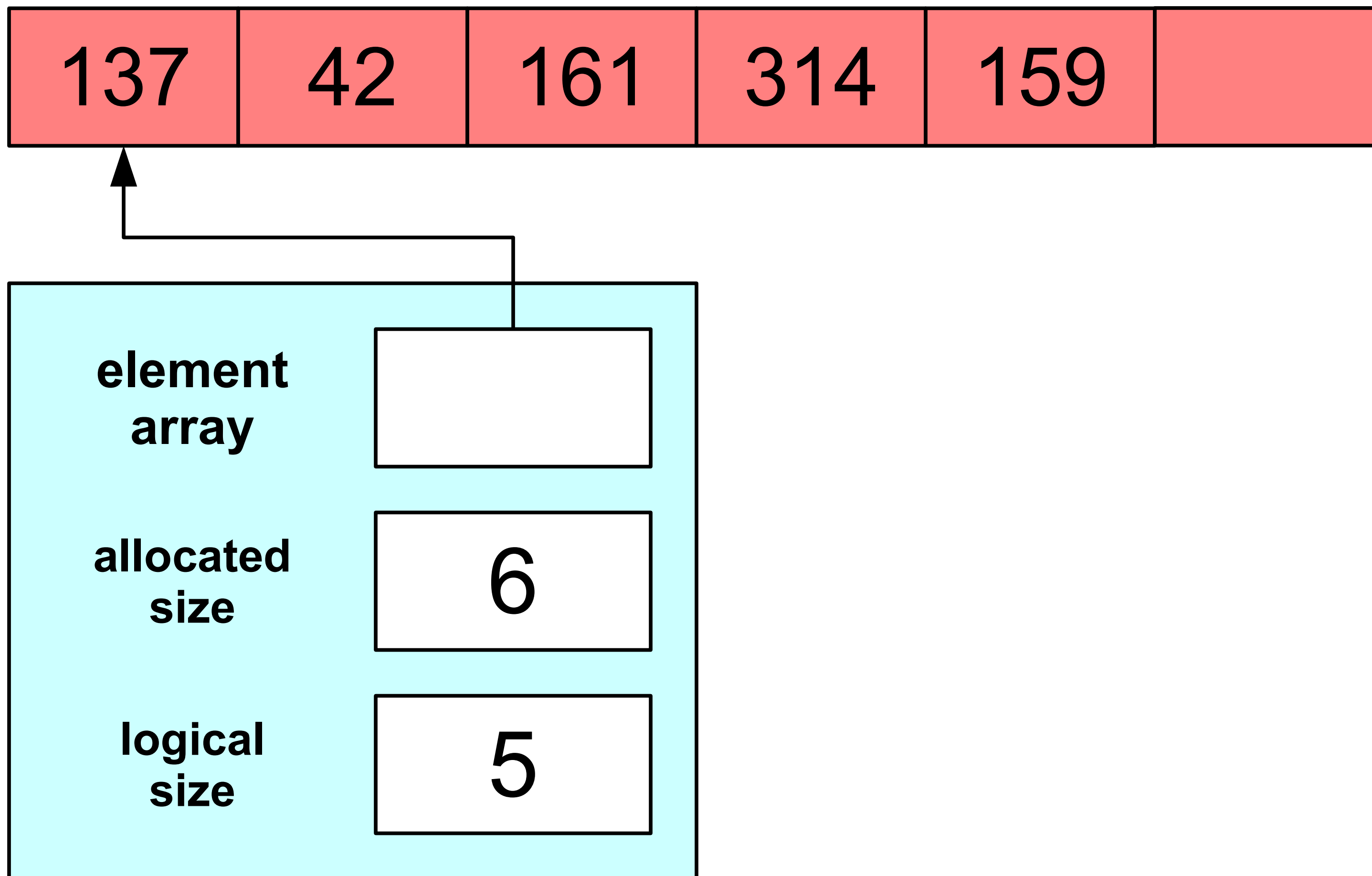
A Better Idea



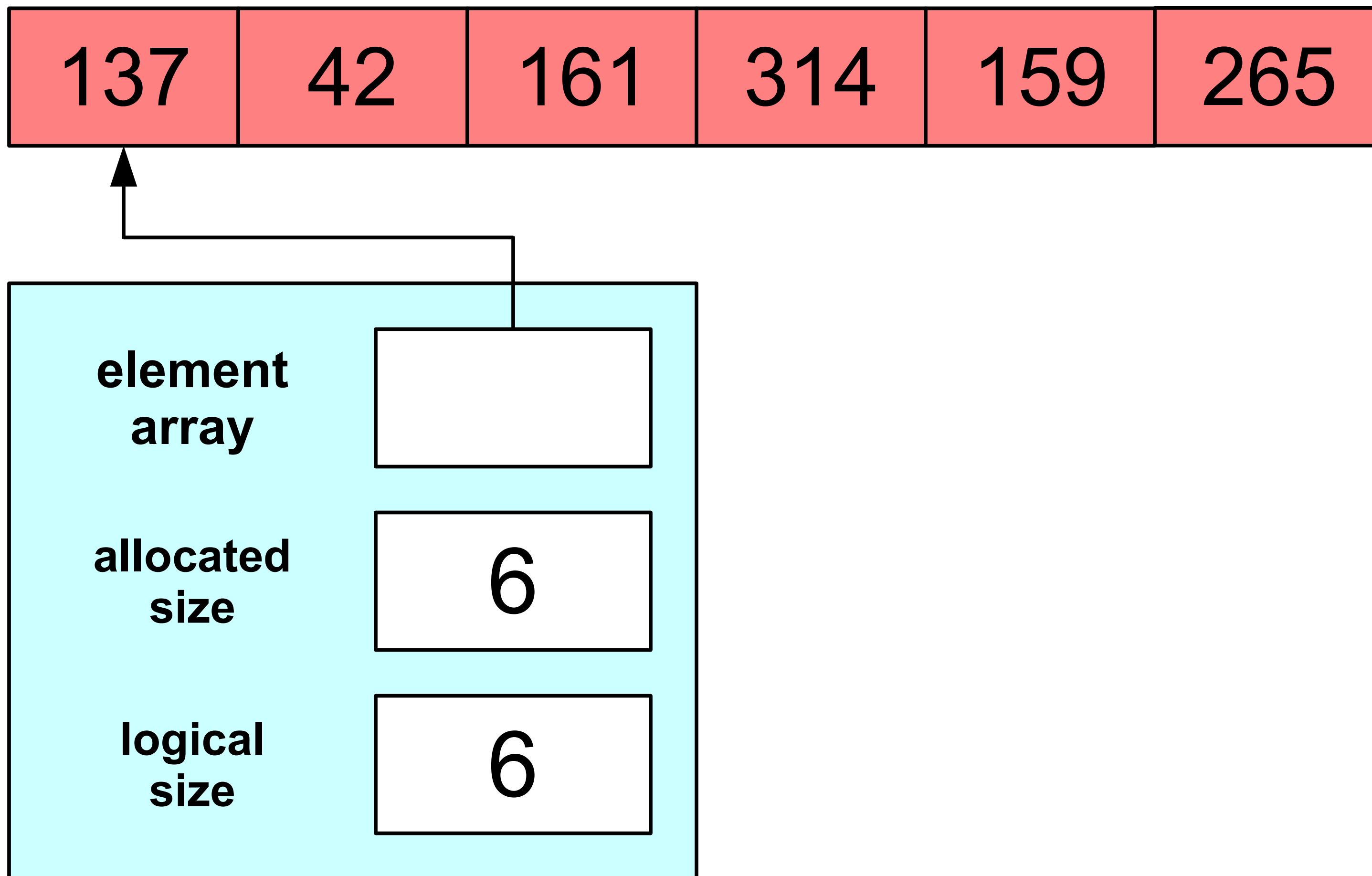
A Better Idea



A Better Idea



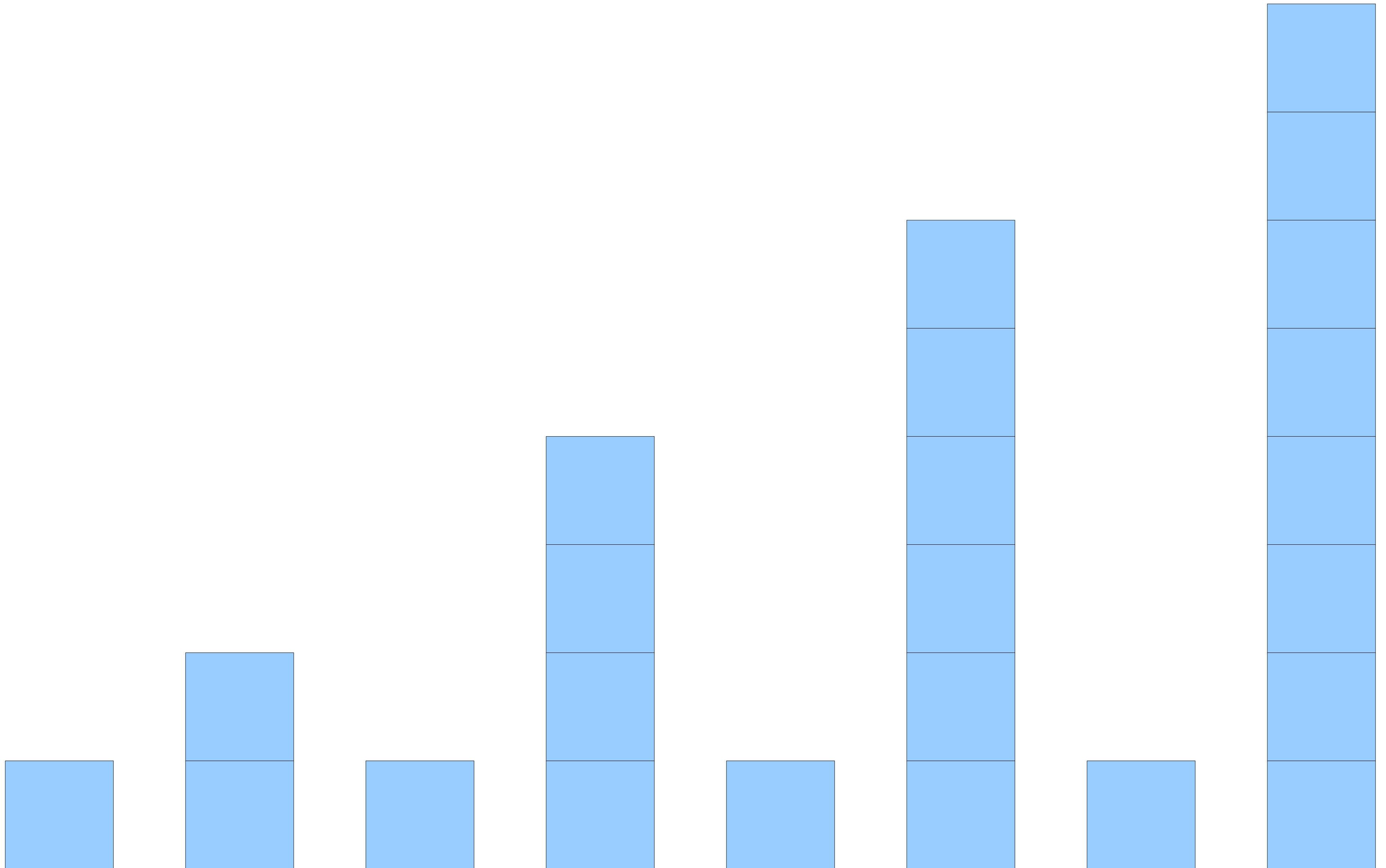
A Better Idea



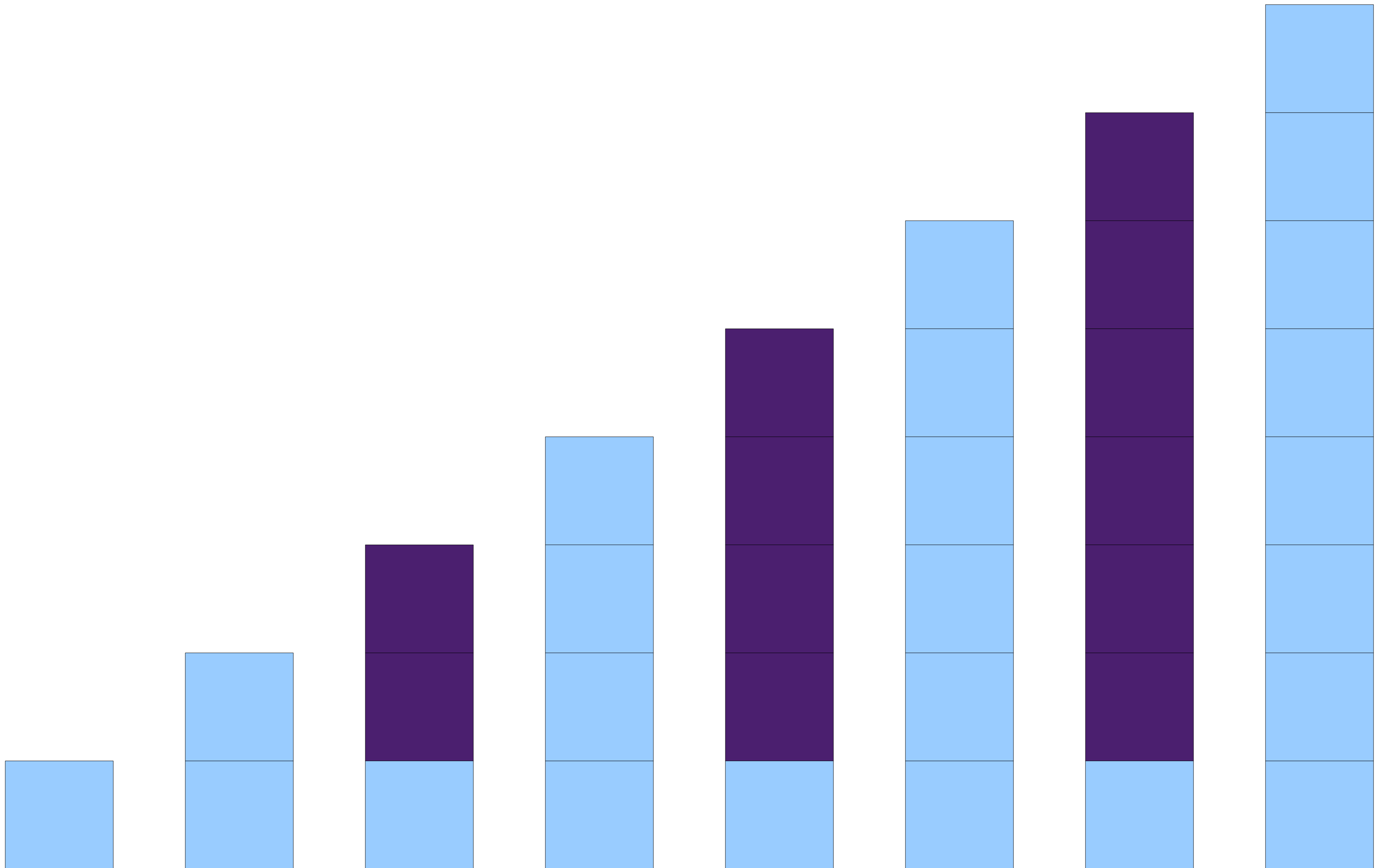
What Just Happened?

- 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?

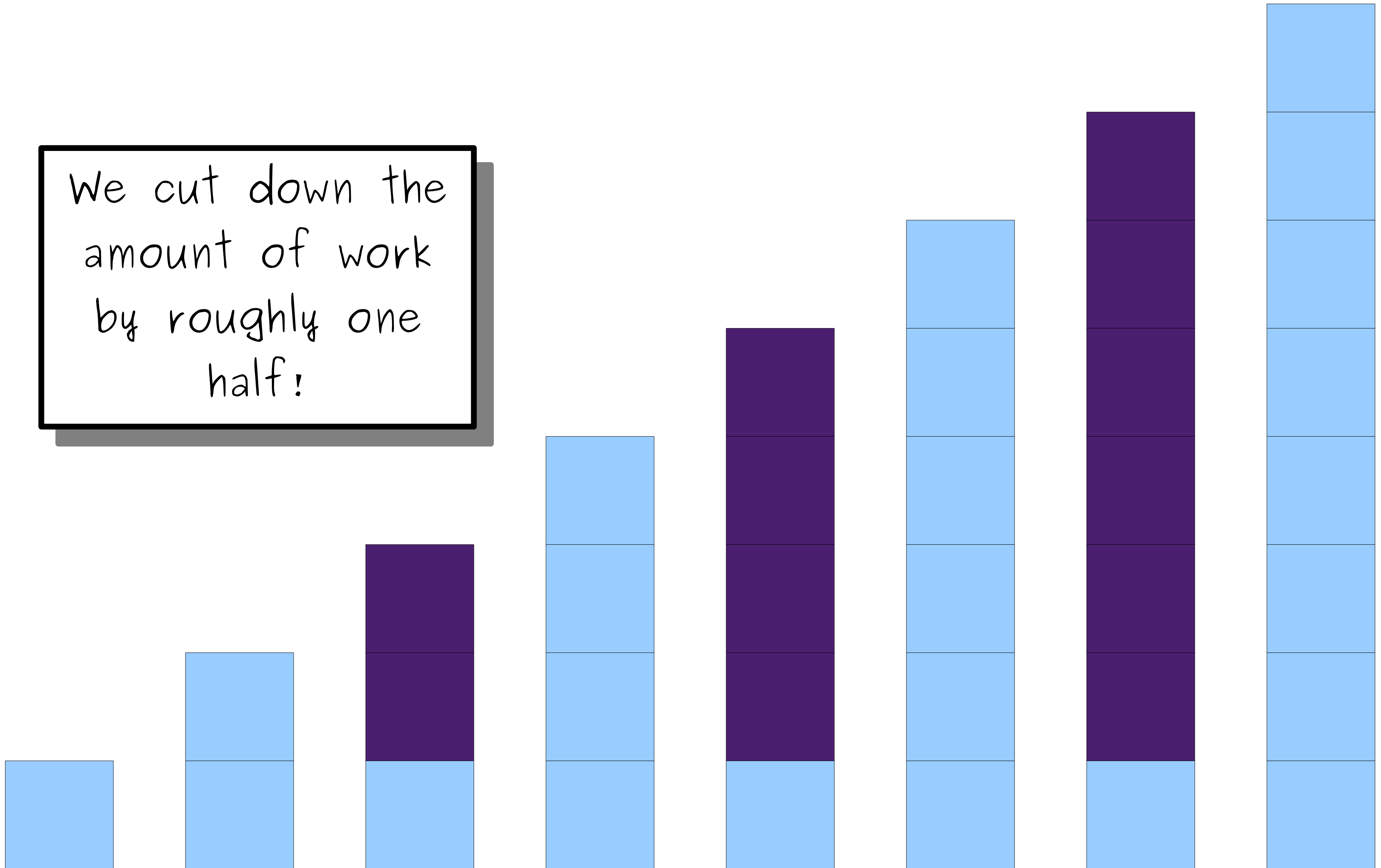
Analyzing the Work



Analyzing the Work

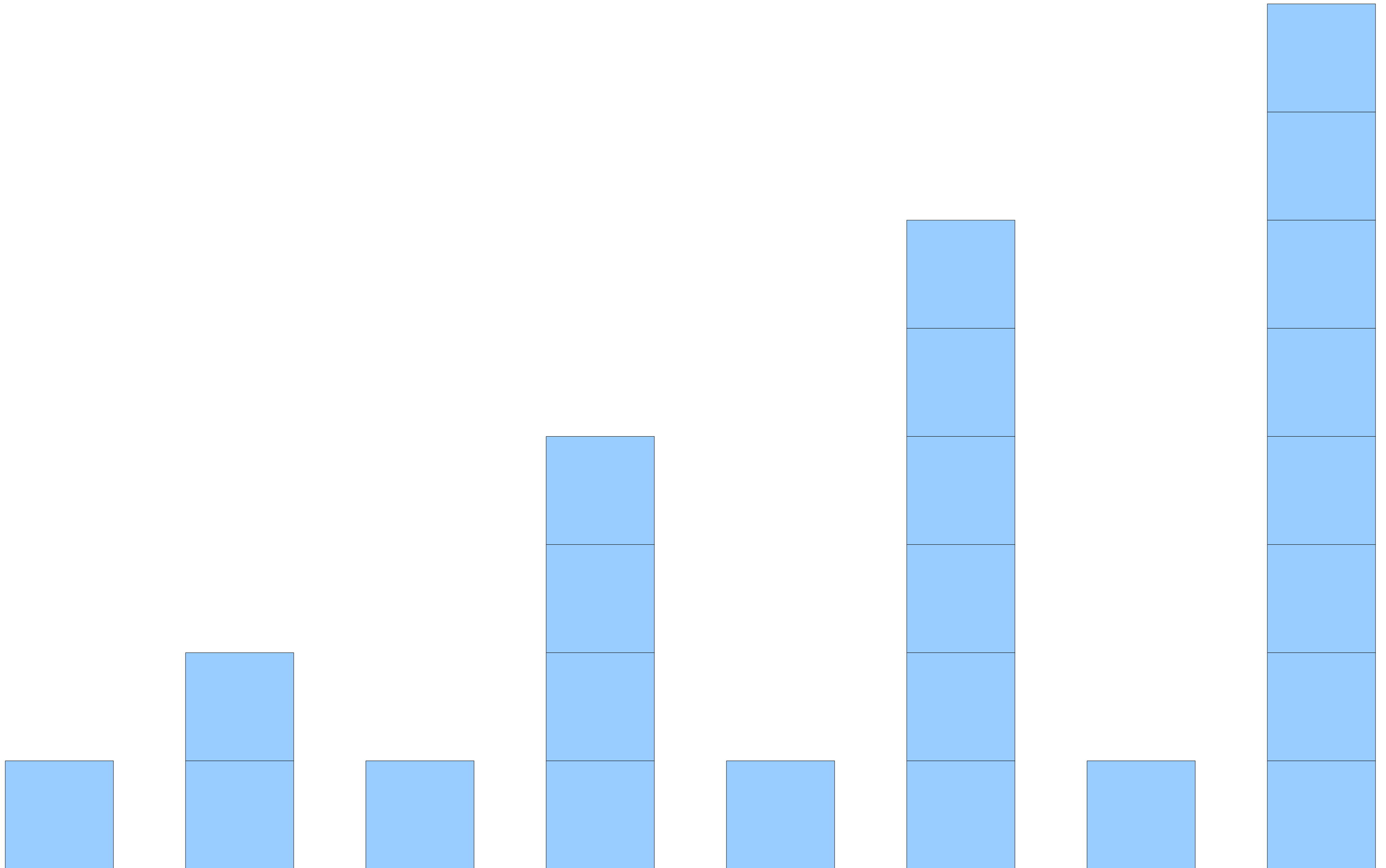


Analyzing the Work

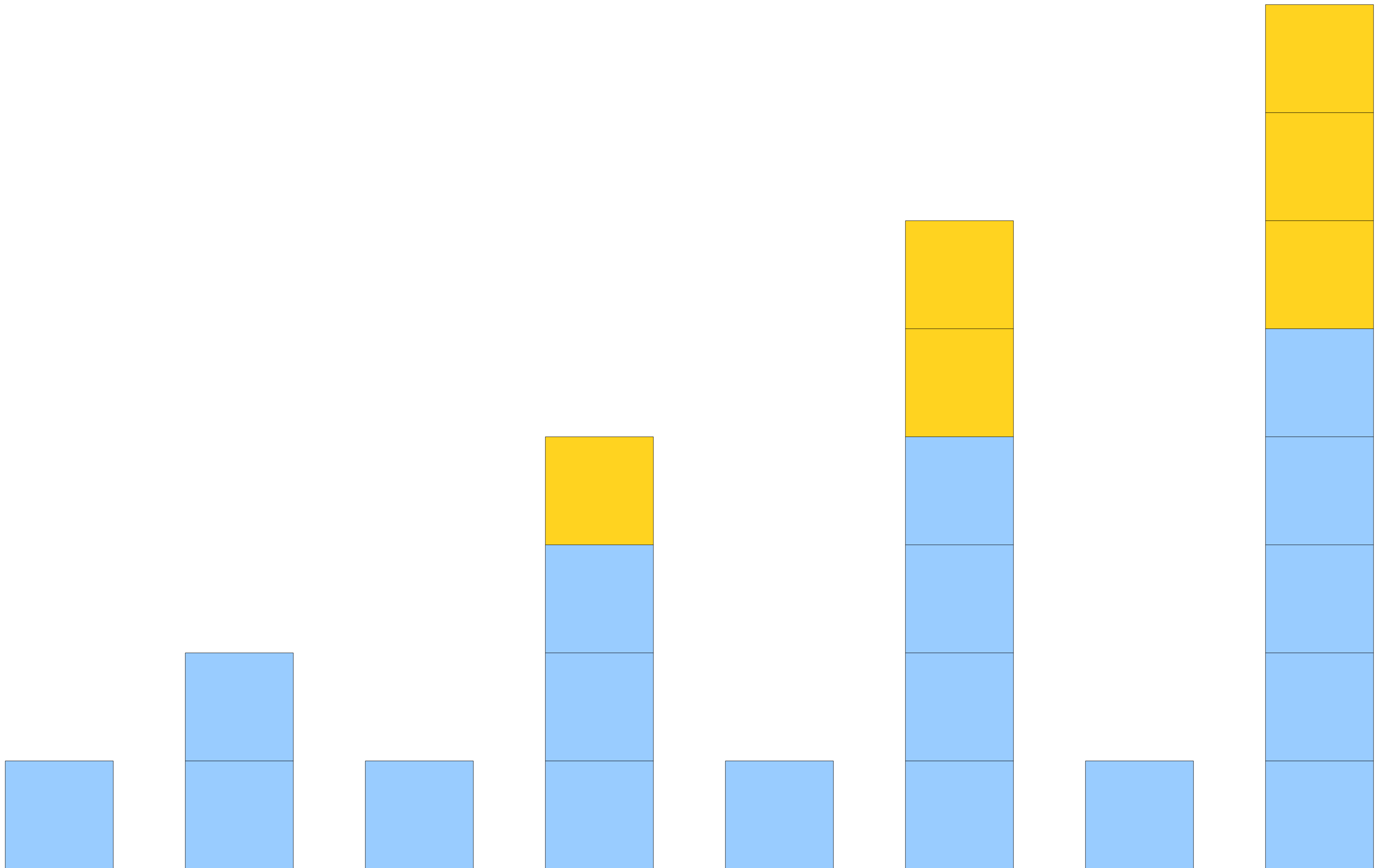


A Different Analysis

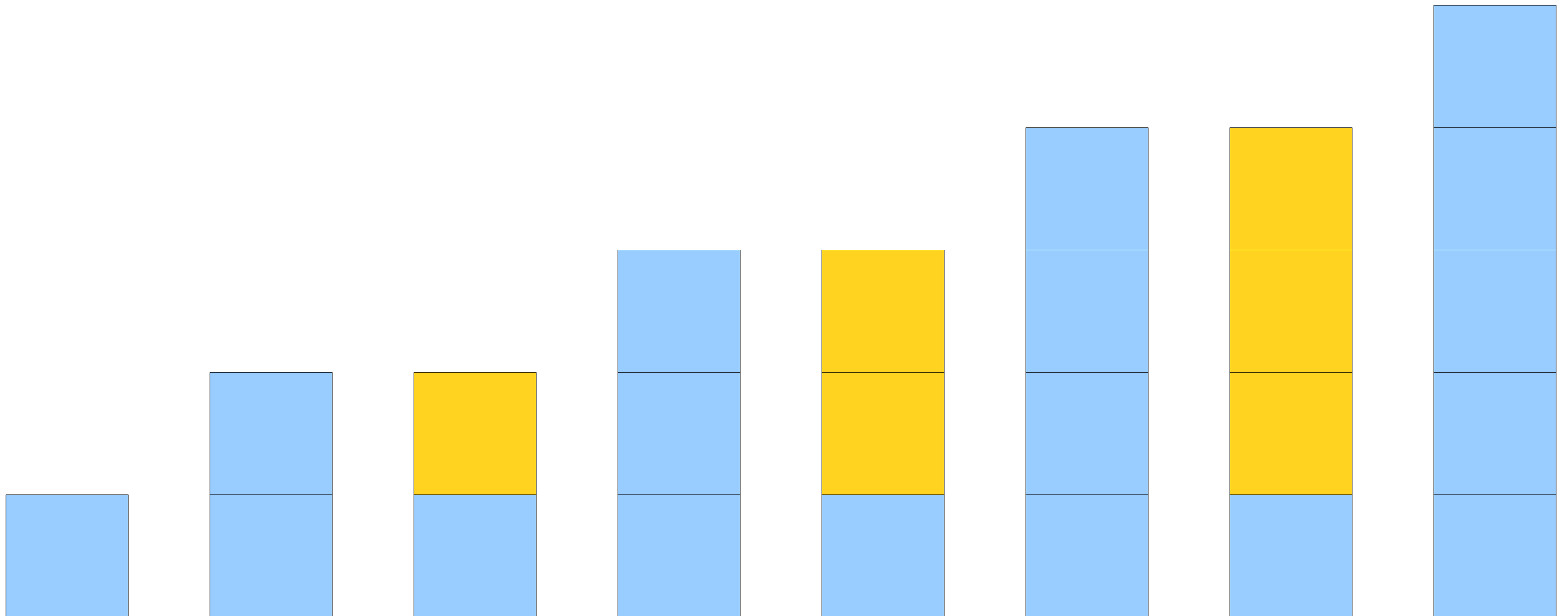
A Different Analysis



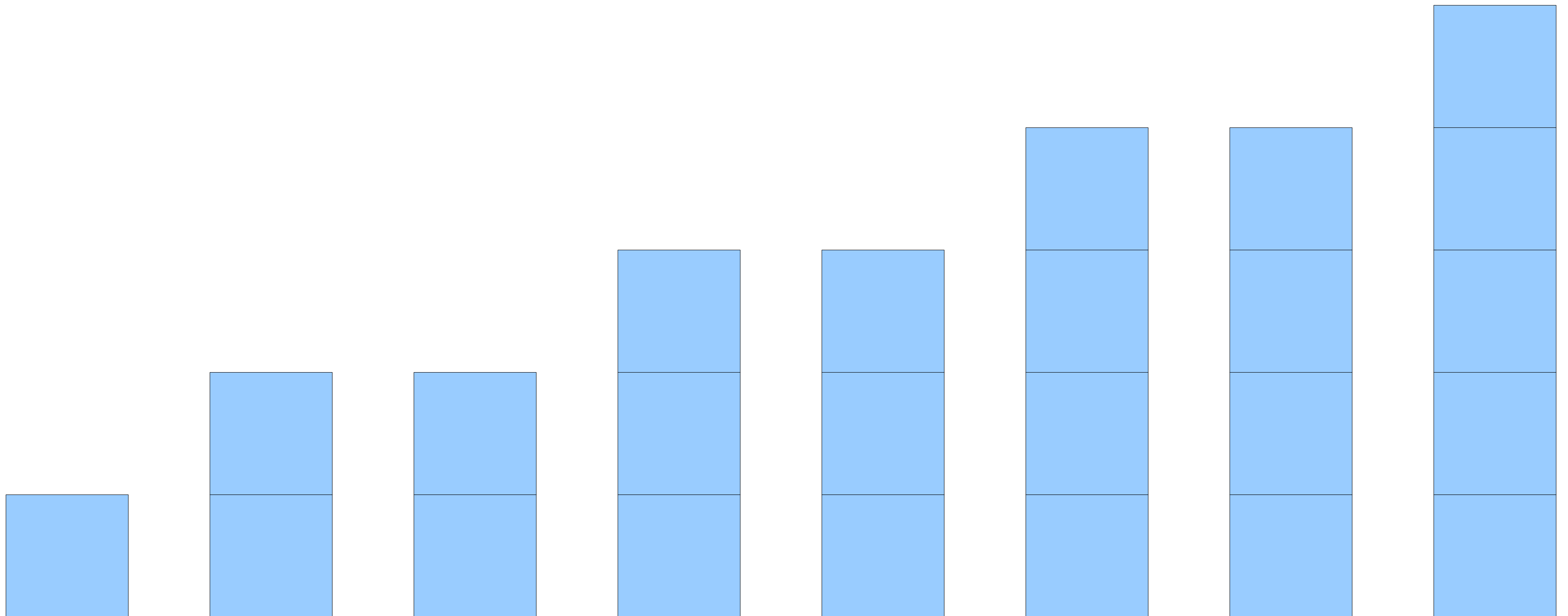
A Different Analysis



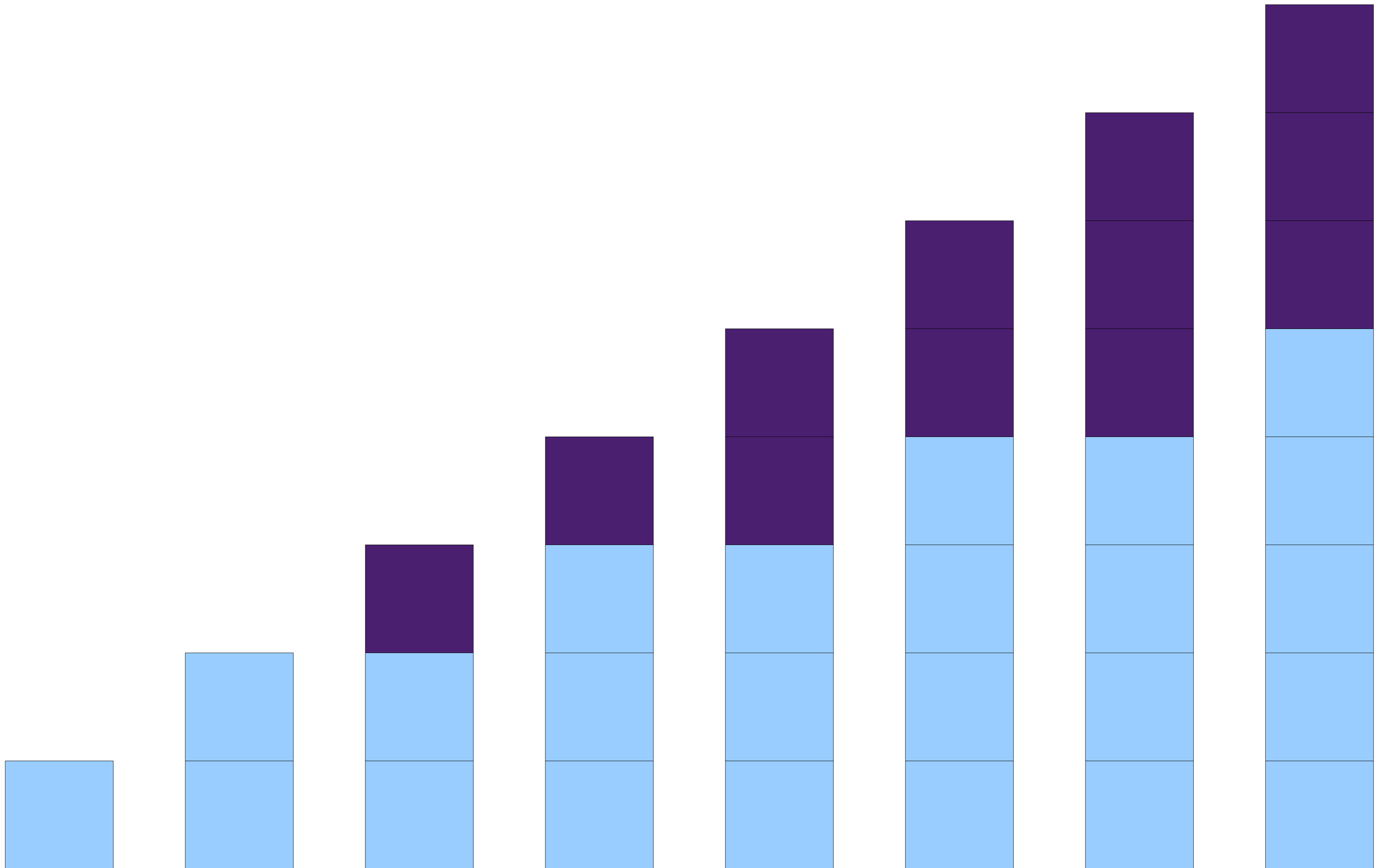
A Different Analysis



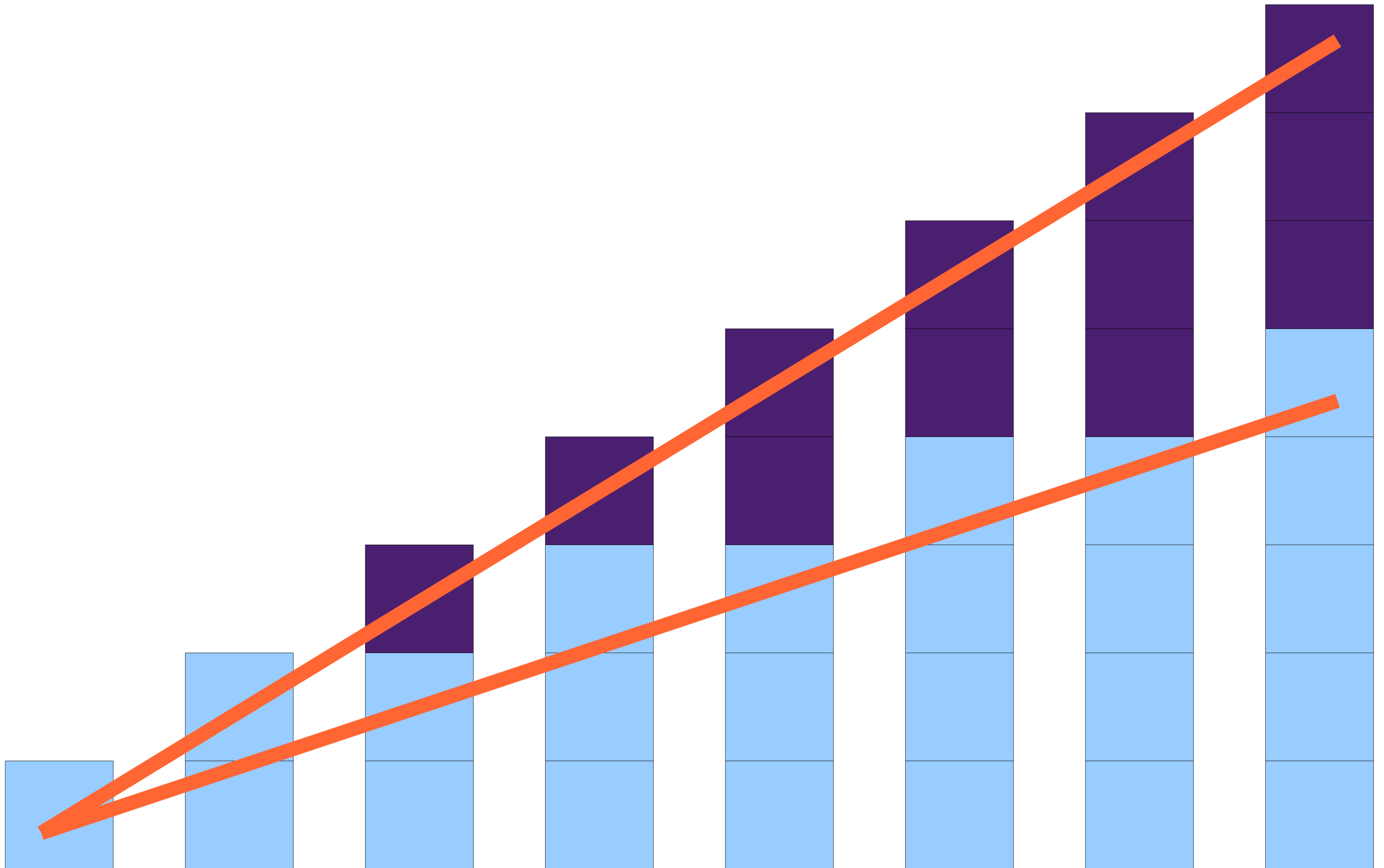
A Different Analysis



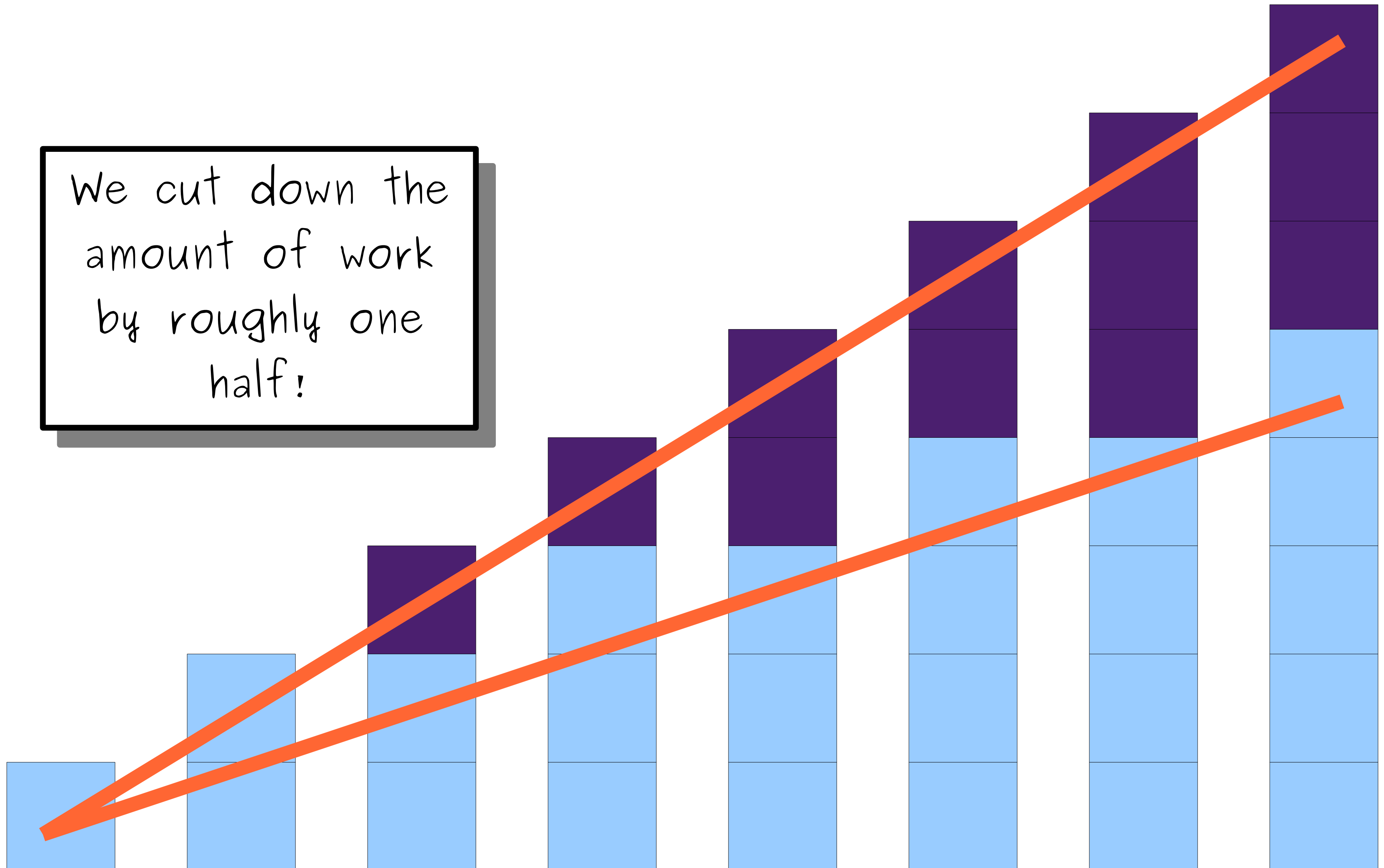
A Different Analysis



A Different Analysis

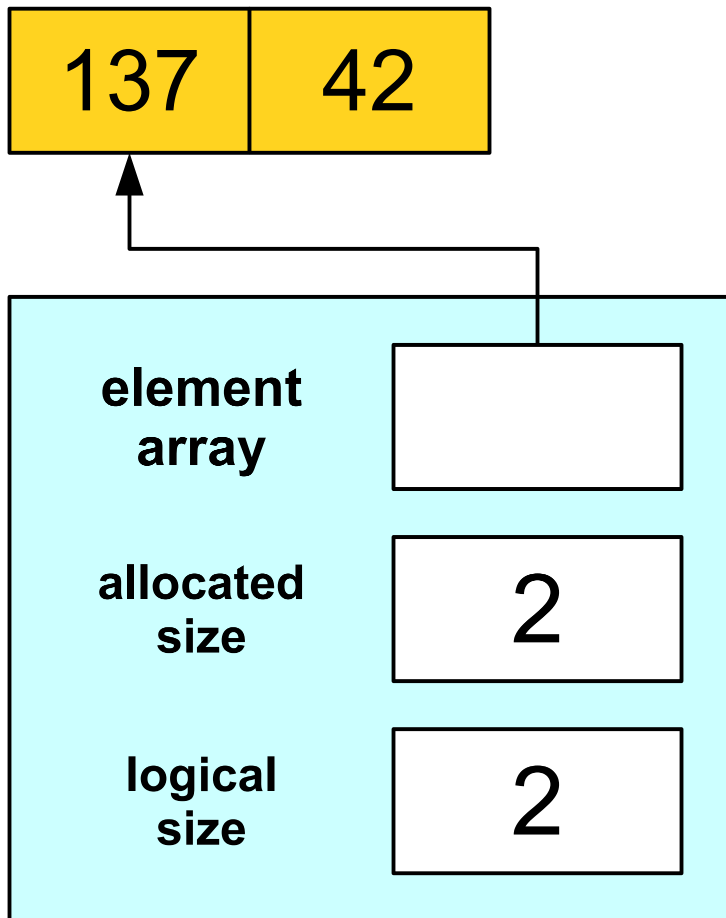


A Different Analysis

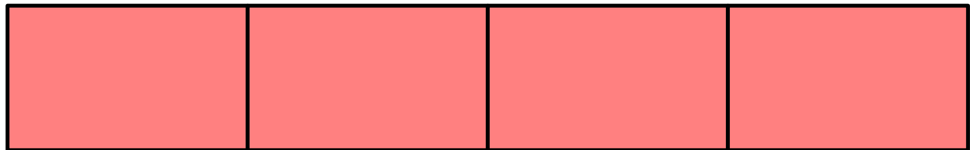


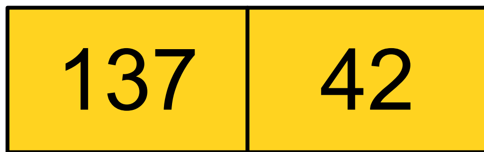
How does it stack up?

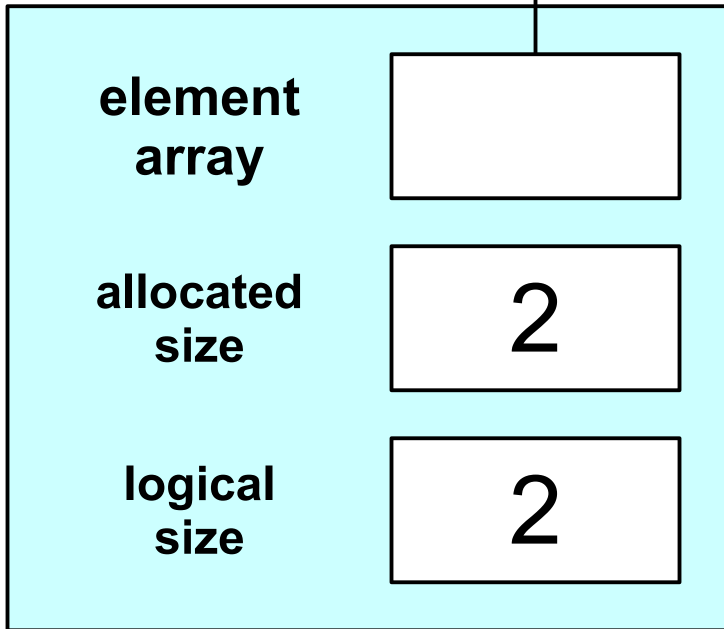
A Much Better Idea



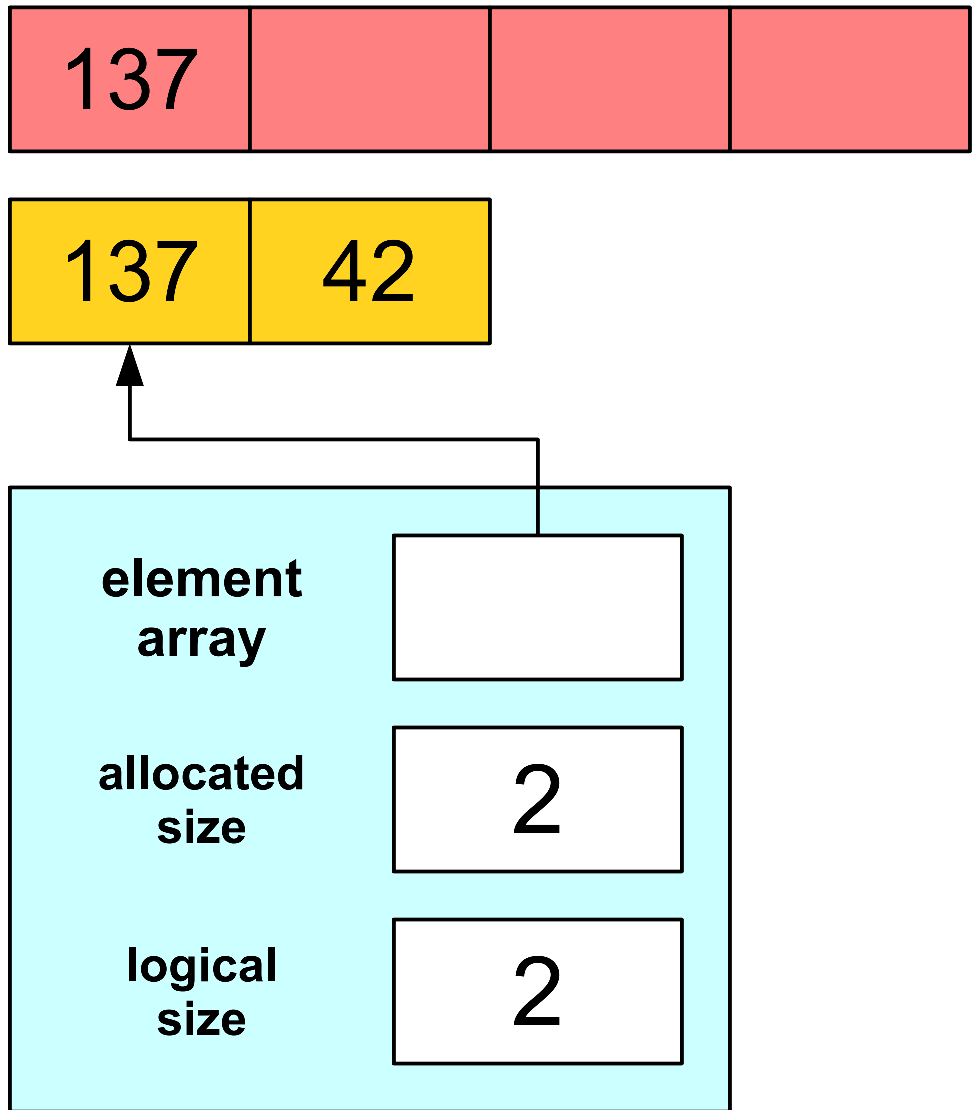
A Much Better Idea



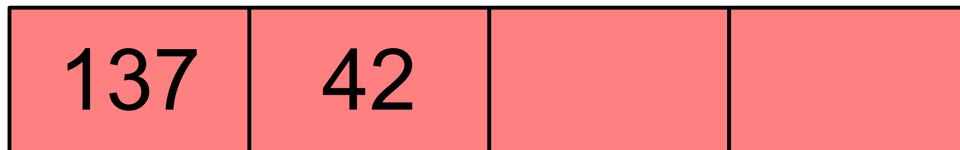


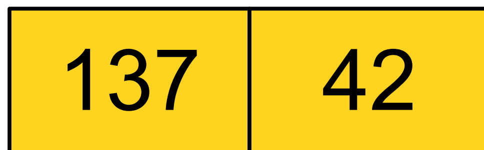


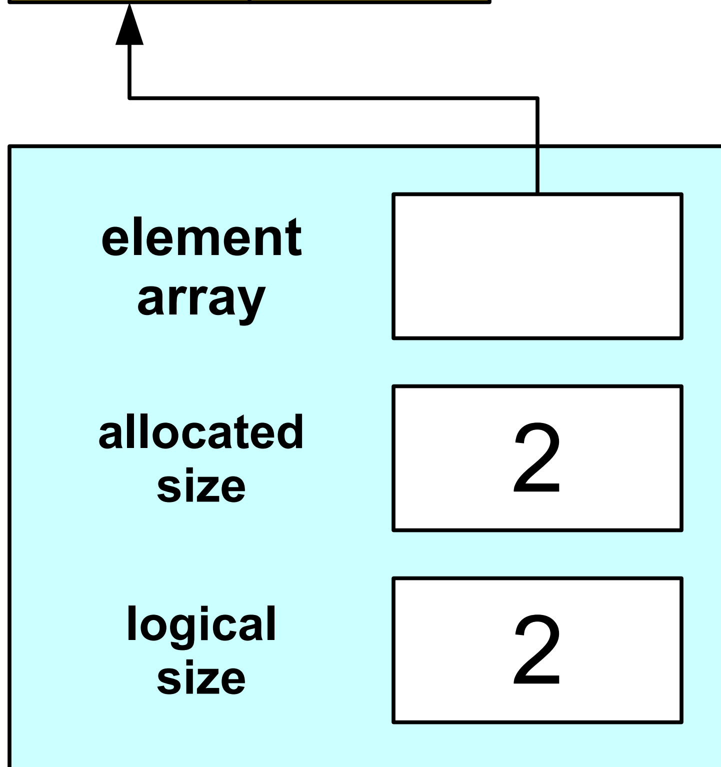
A Much Better Idea



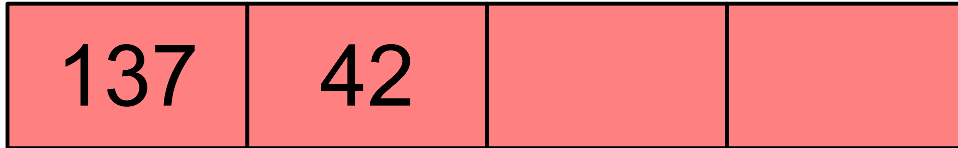
A Much Better Idea

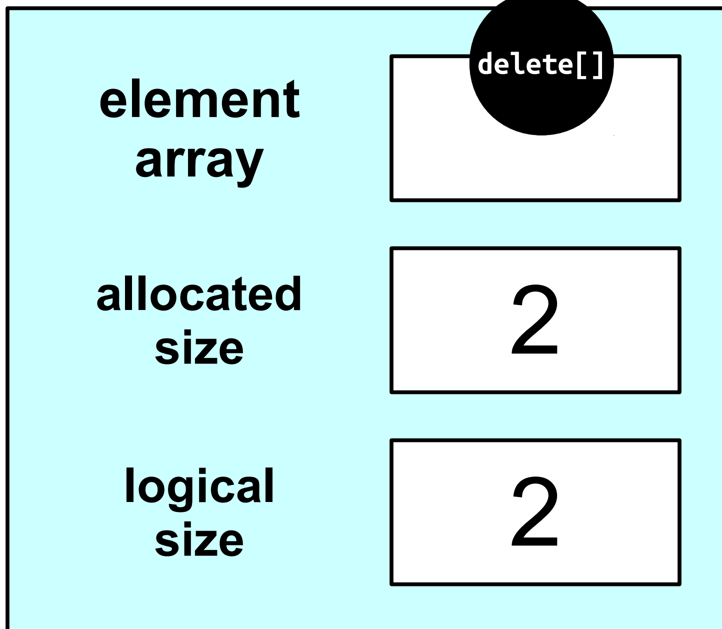




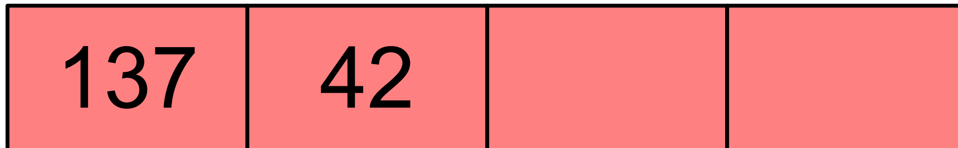


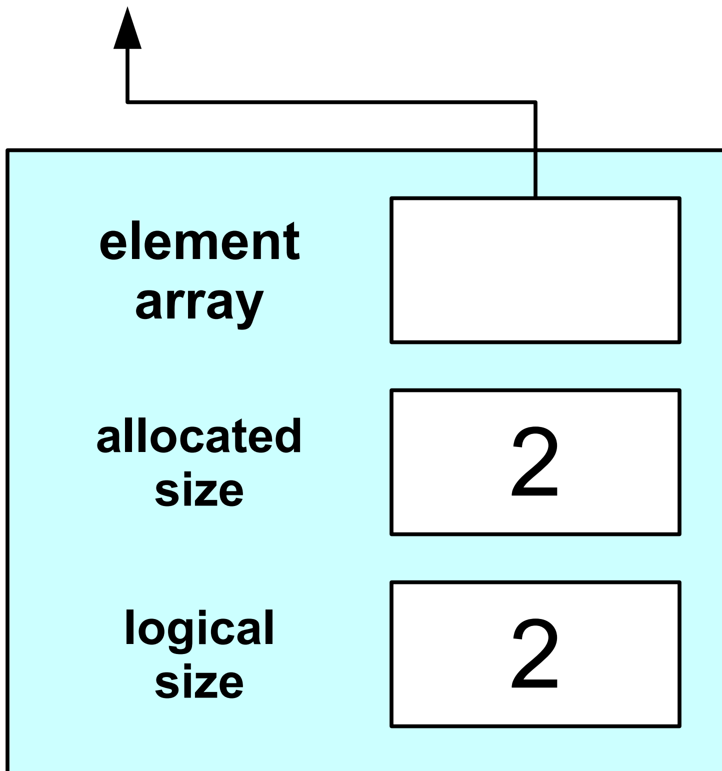
A Much Better Idea



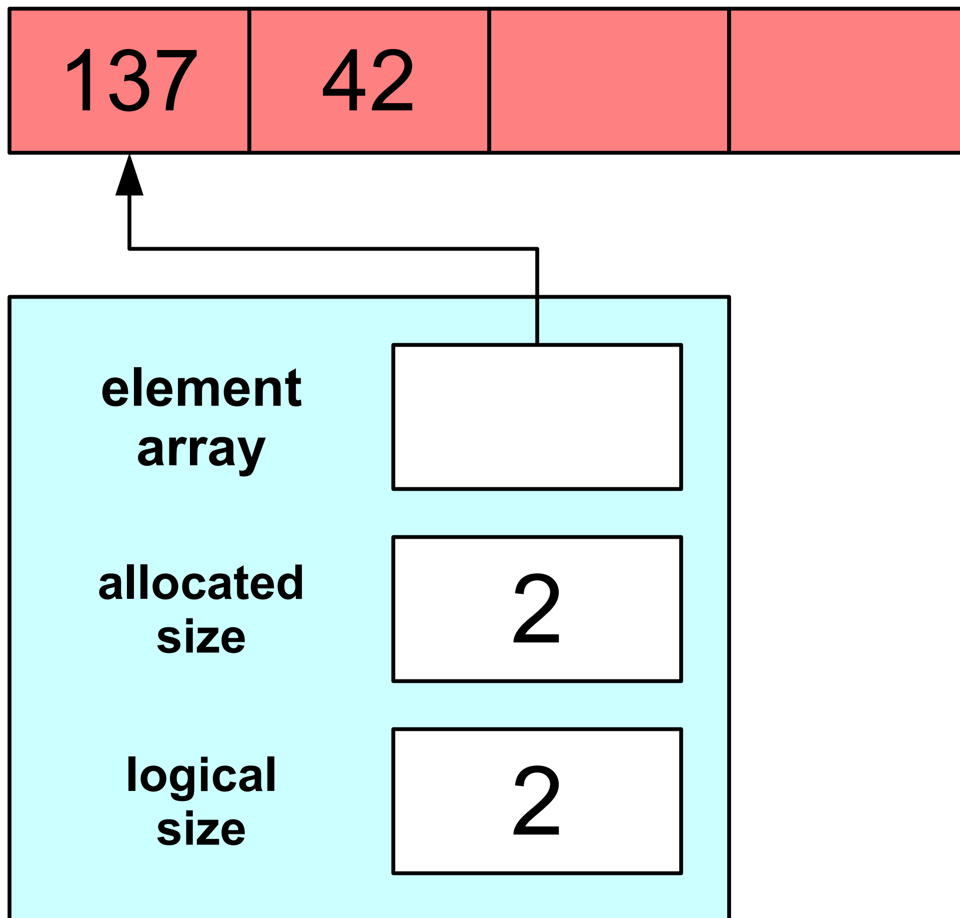


A Much Better Idea

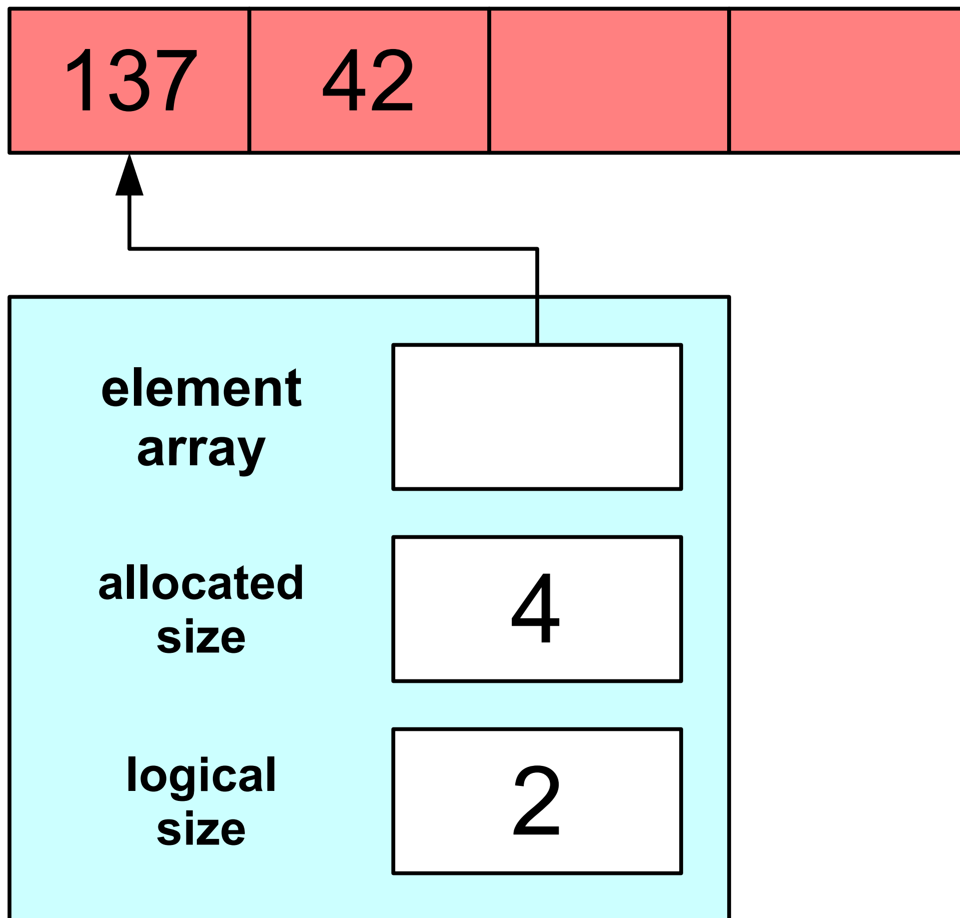




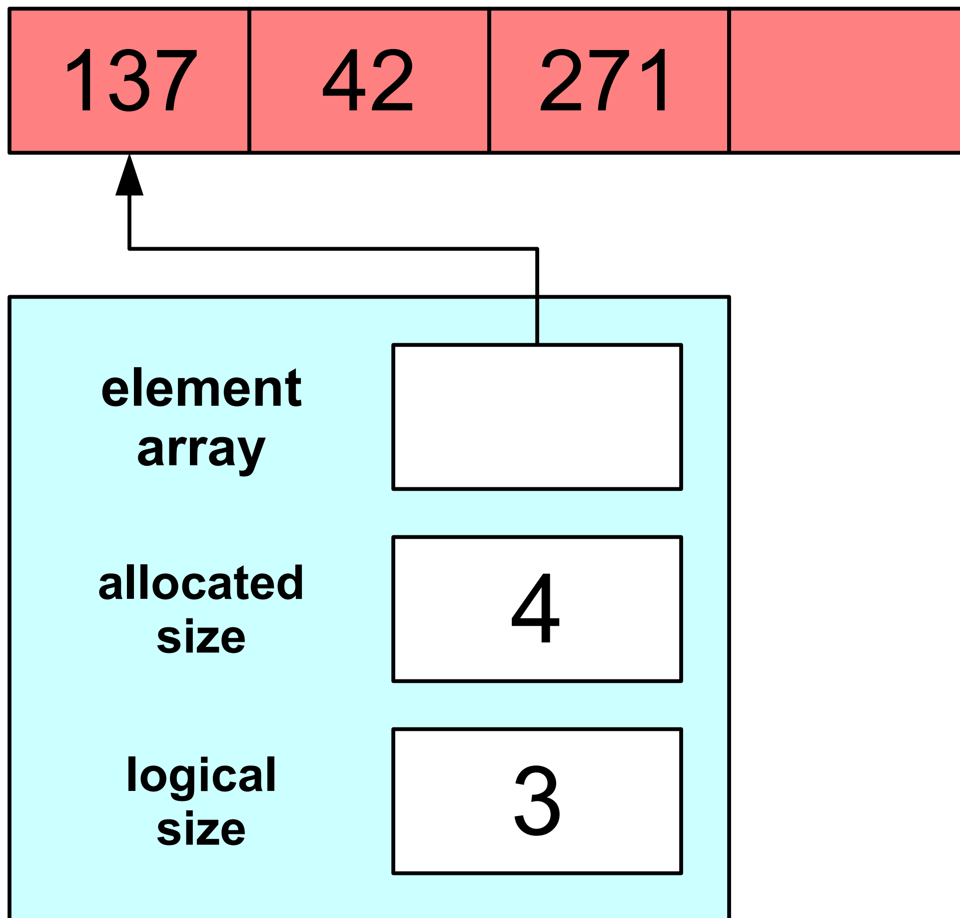
A Much Better Idea



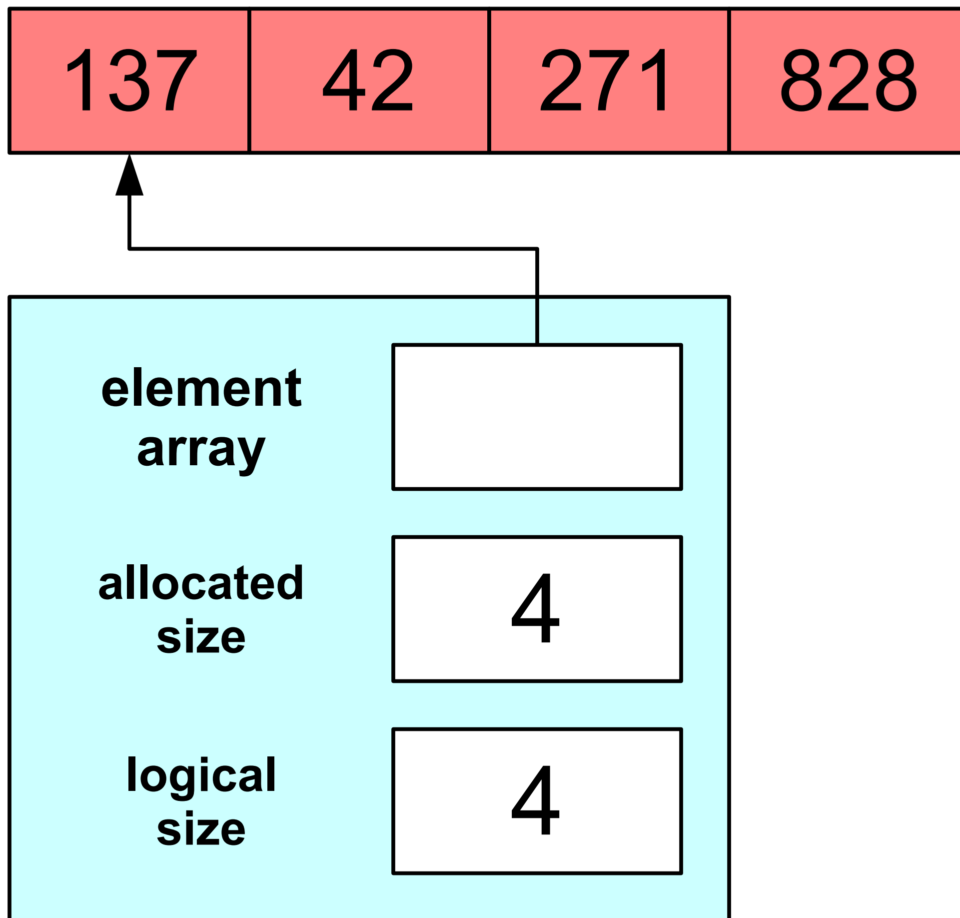
A Much Better Idea



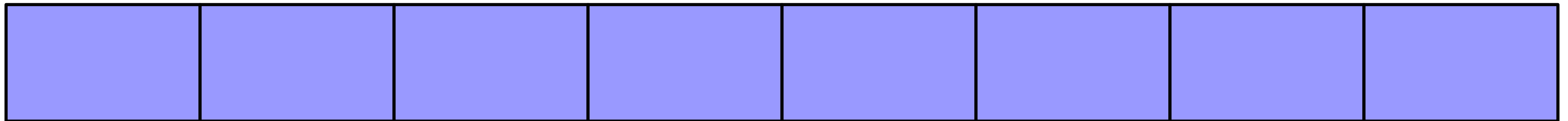
A Much Better Idea

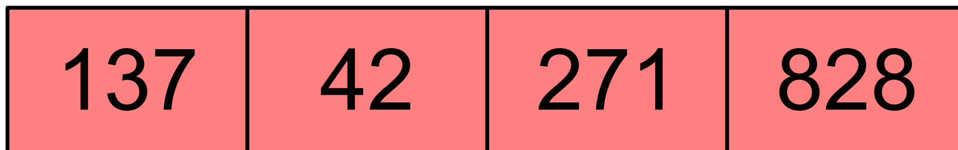


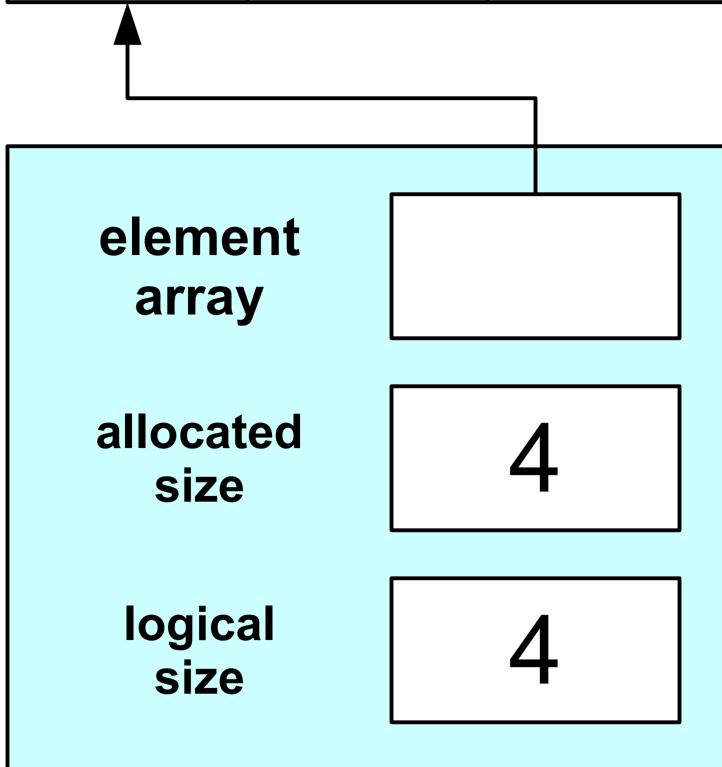
A Much Better Idea



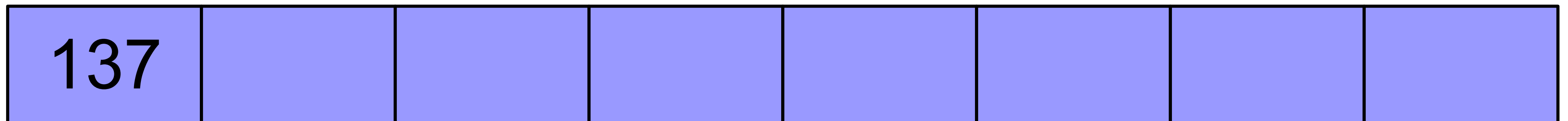
A Much Better Idea

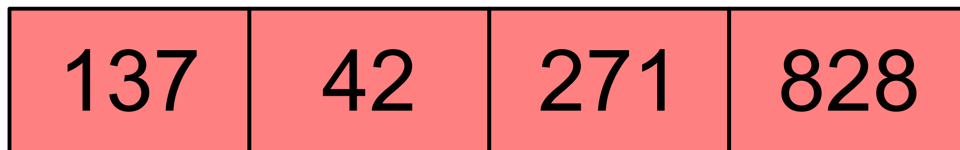


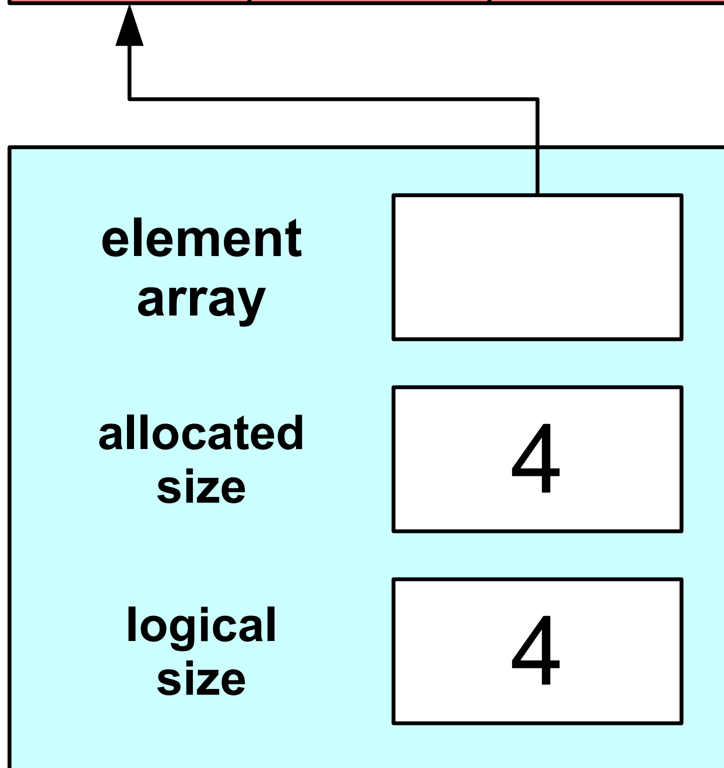




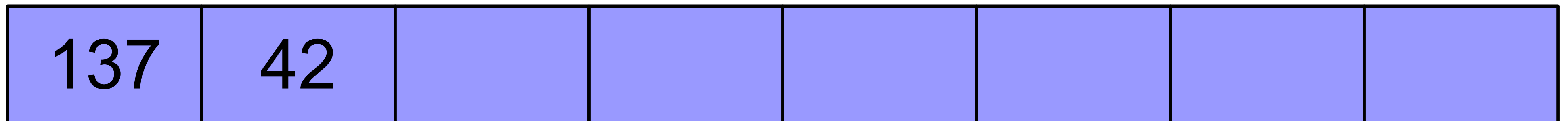
A Much Better Idea

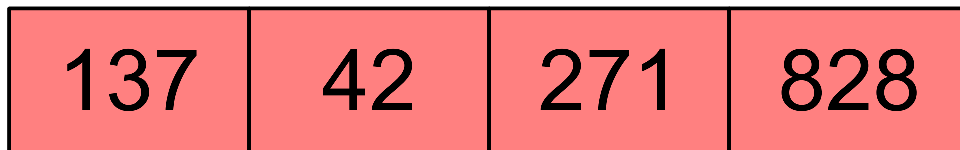


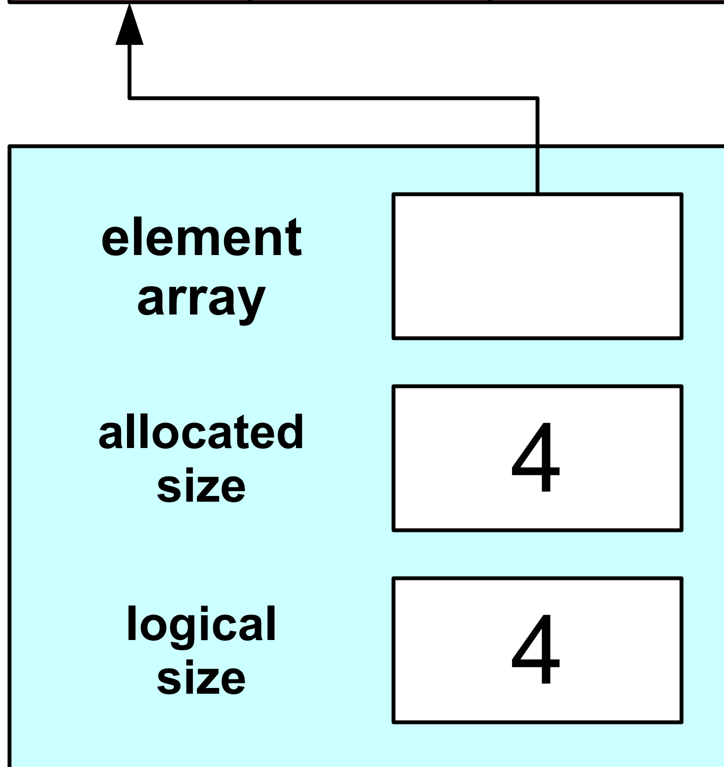




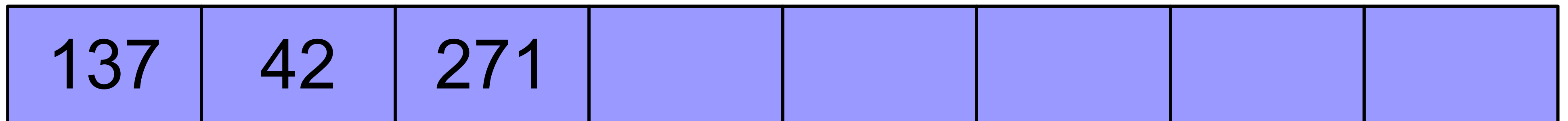
A Much Better Idea

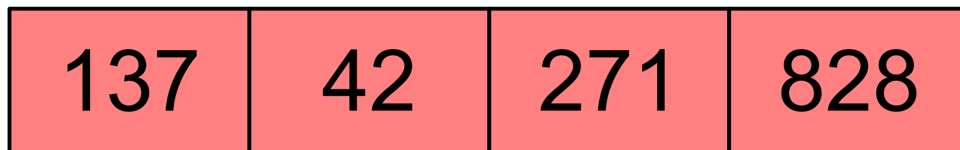


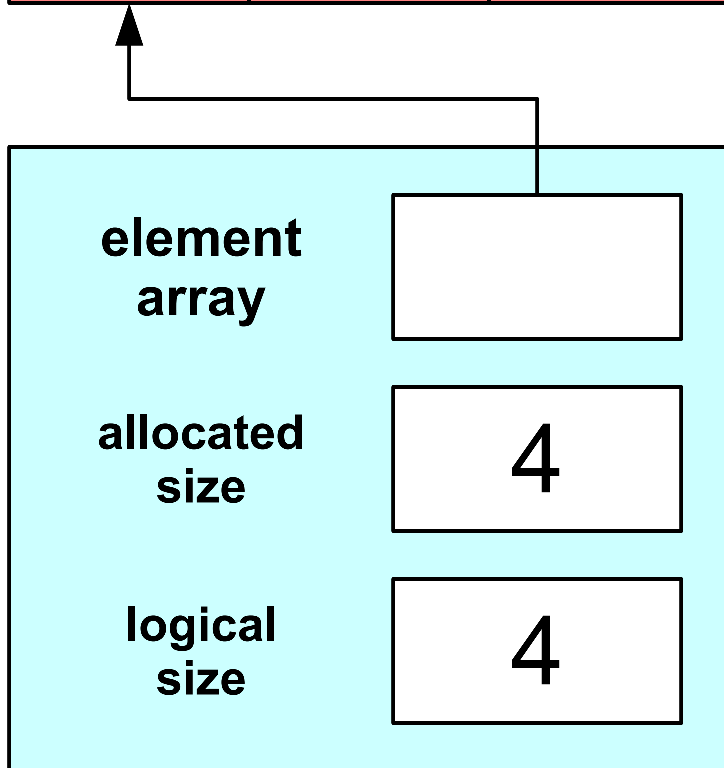




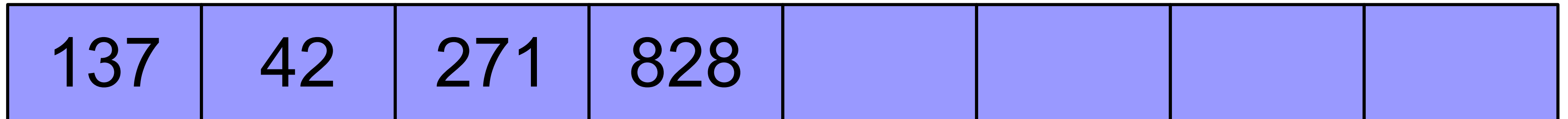
A Much Better Idea

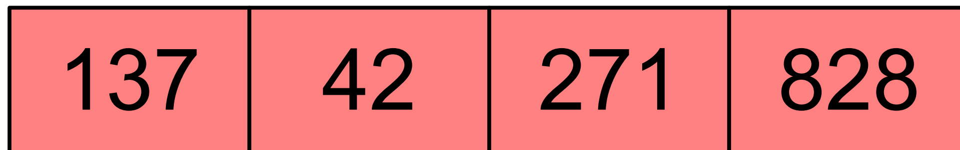


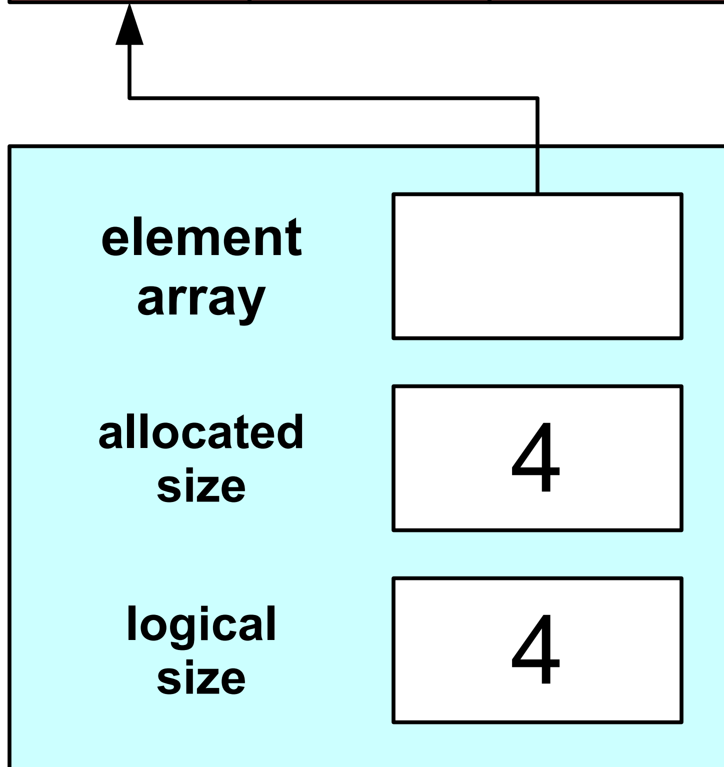




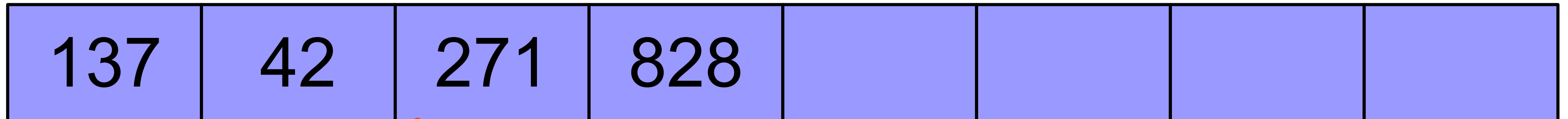
A Much Better Idea

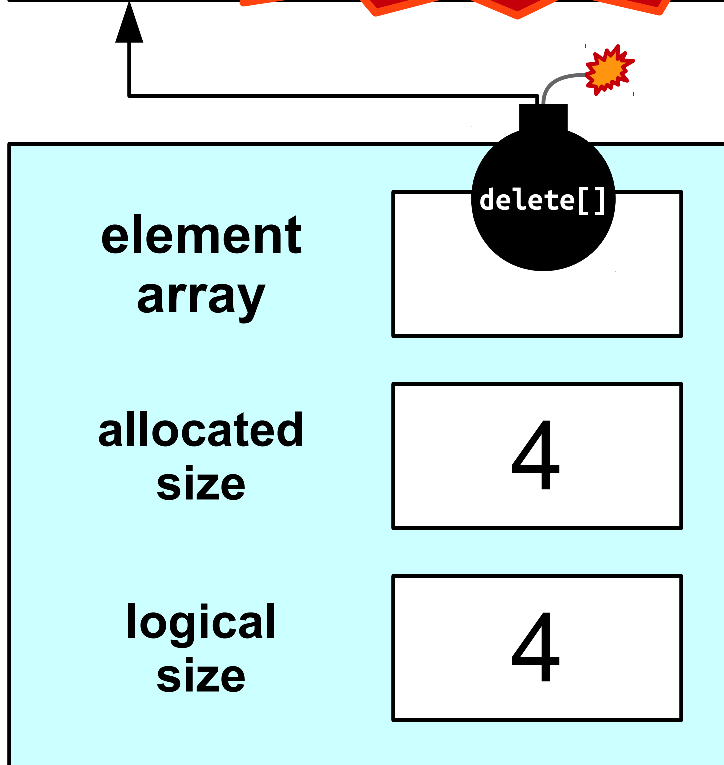




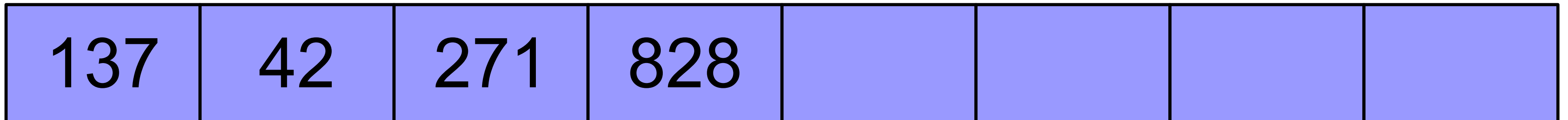


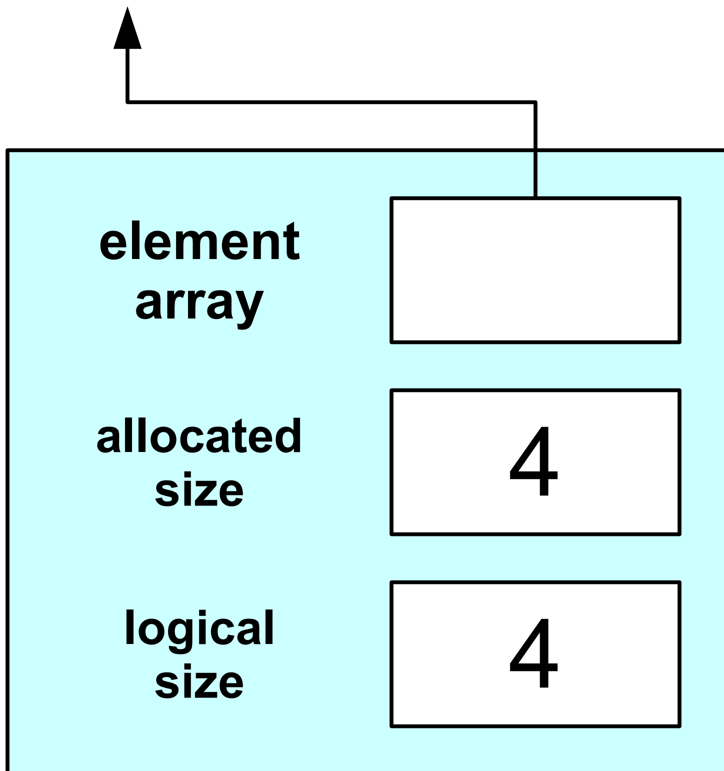
A Much Better Idea



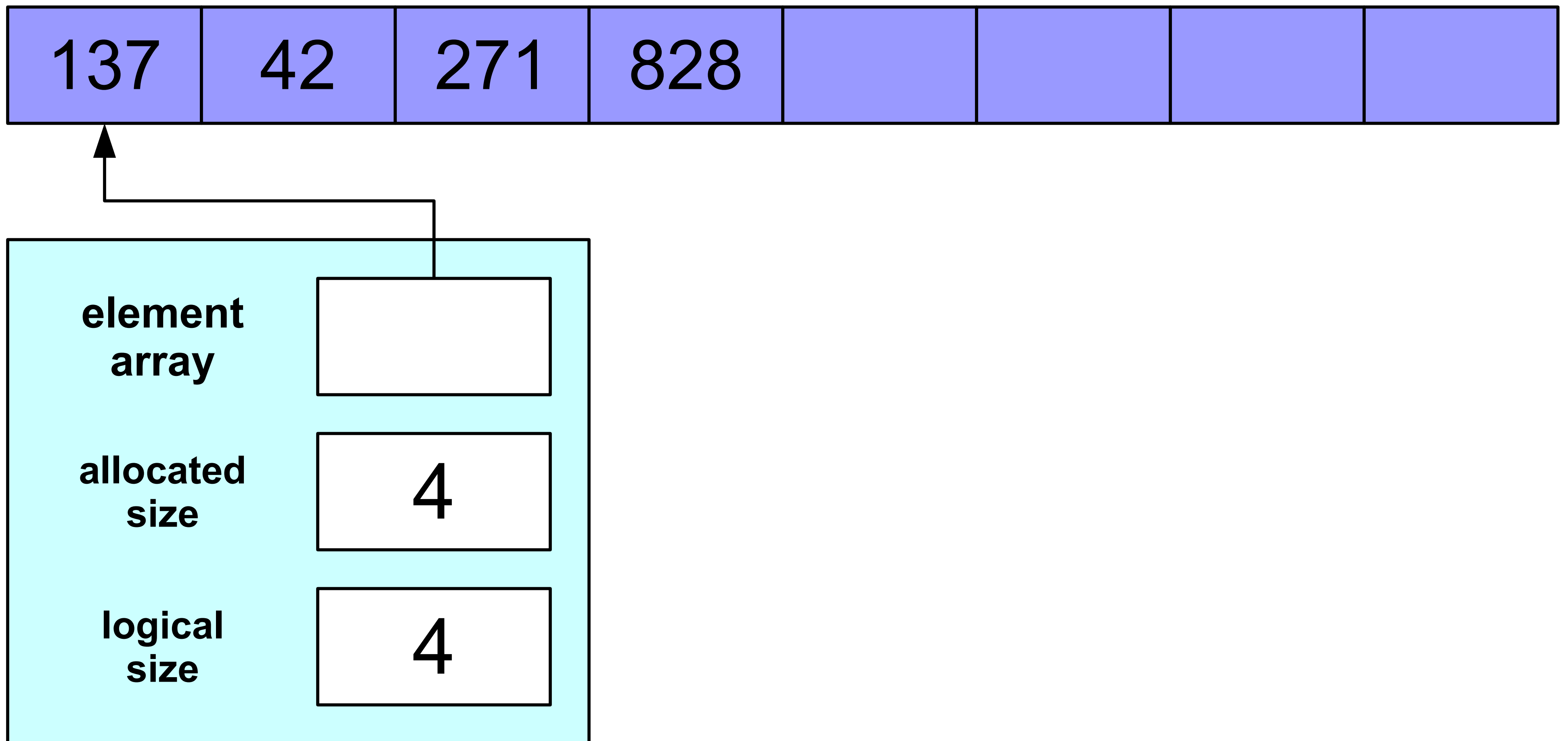


A Much Better Idea

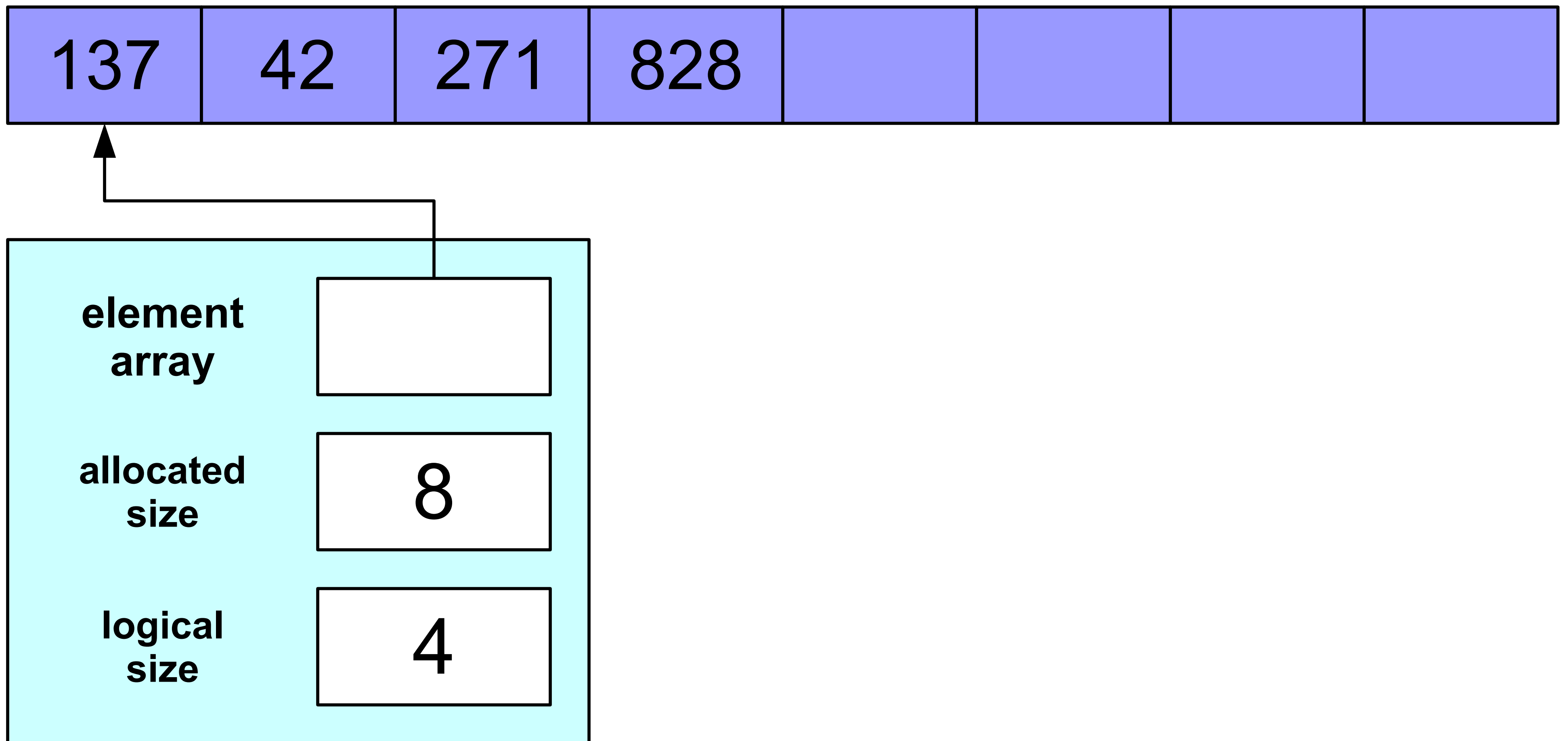




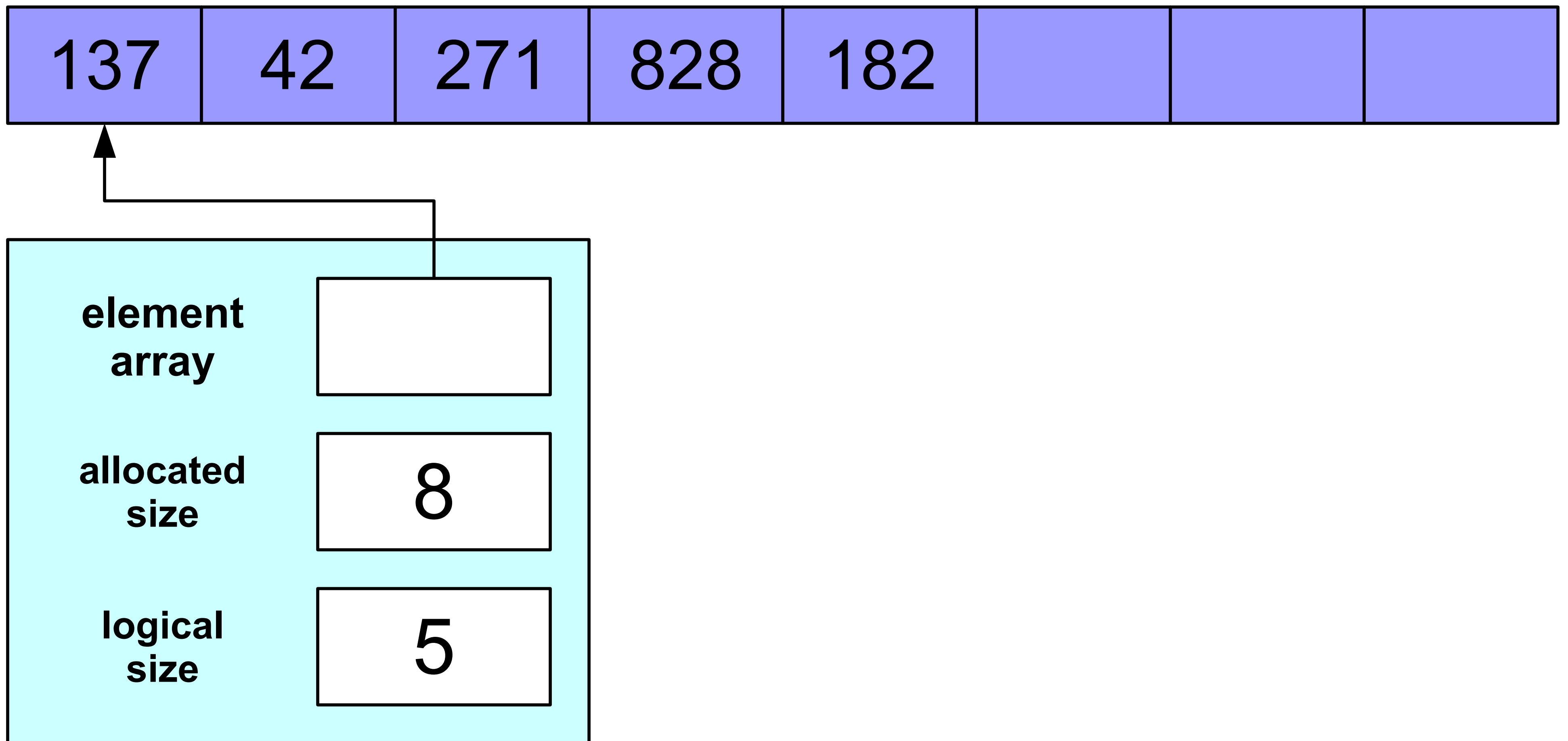
A Much Better Idea



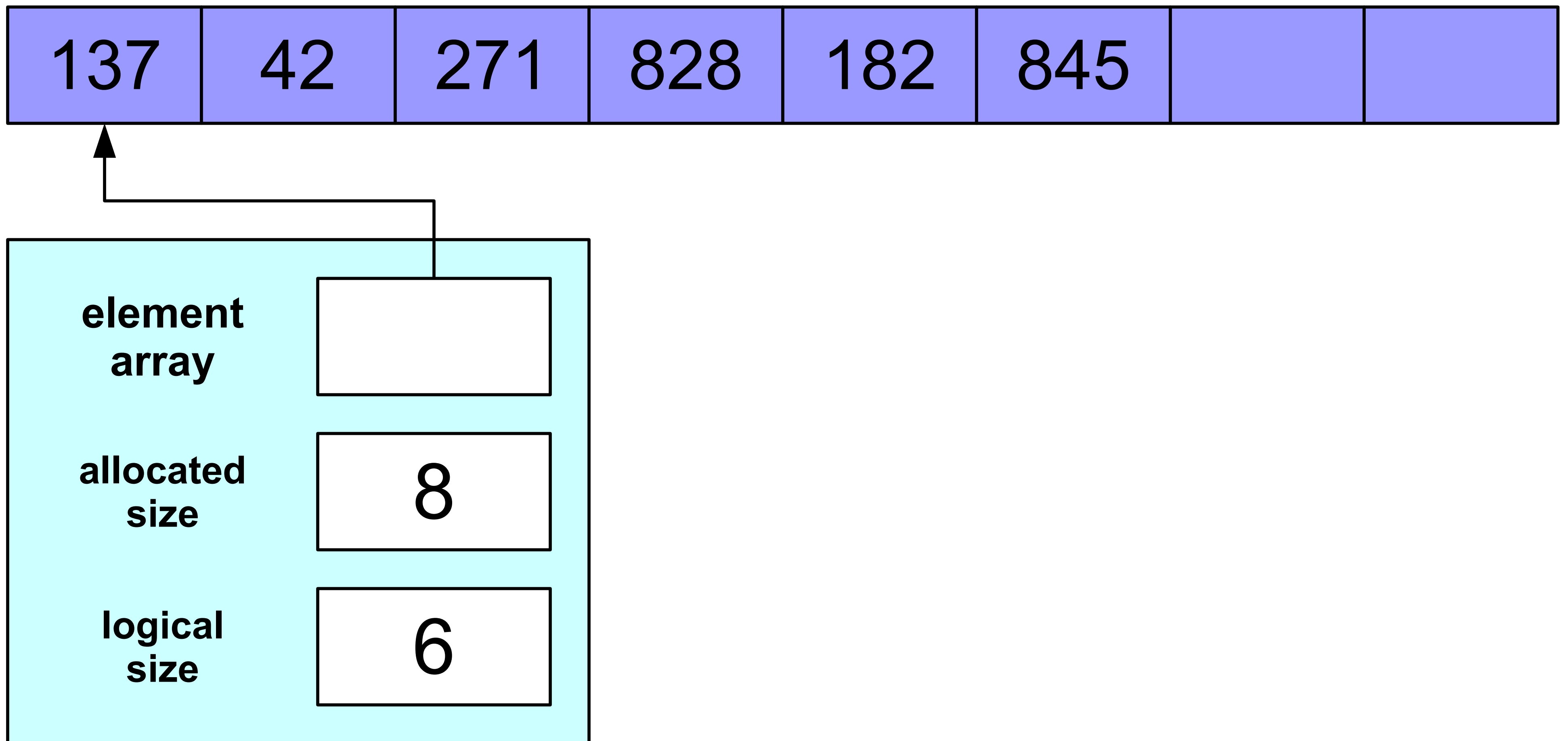
A Much Better Idea



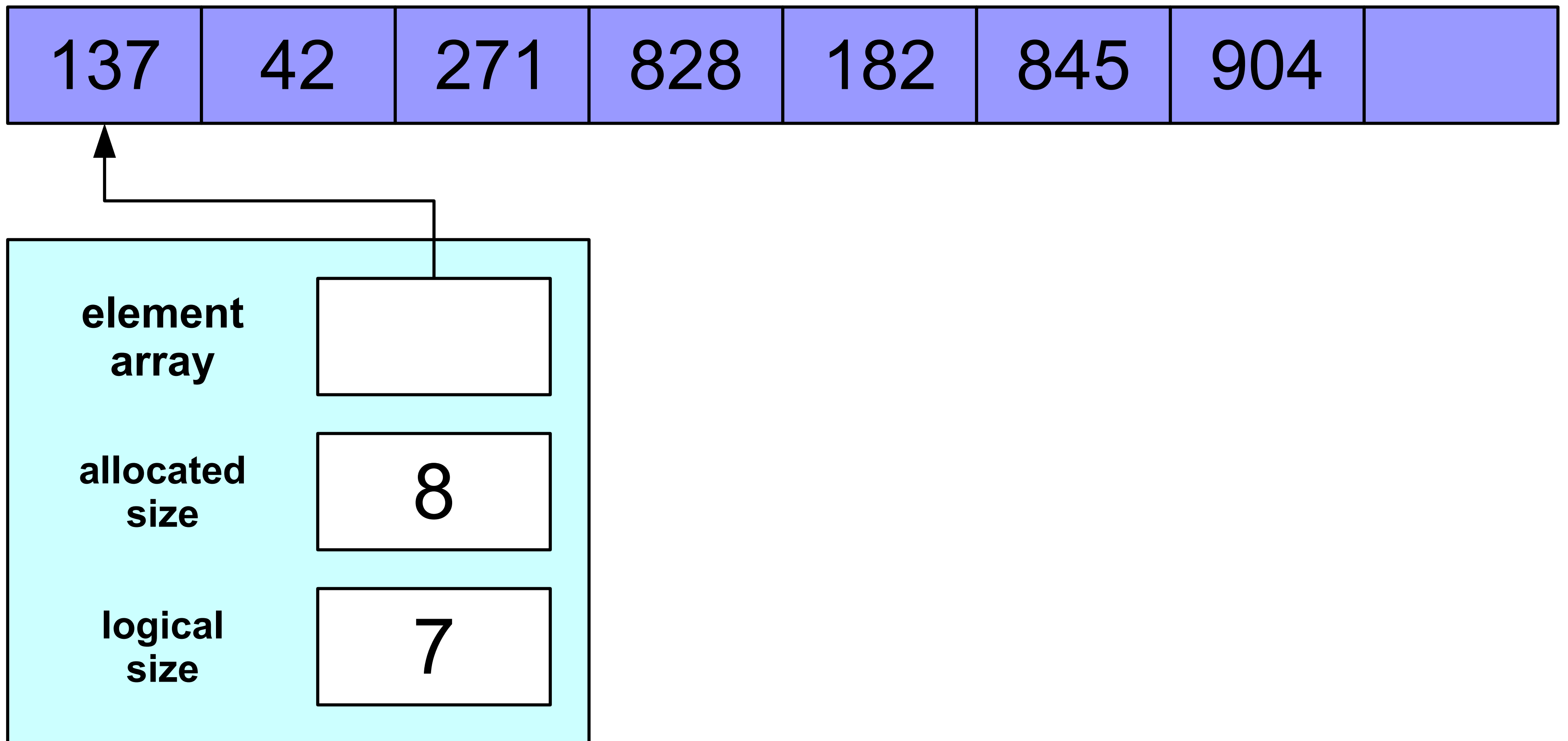
A Much Better Idea



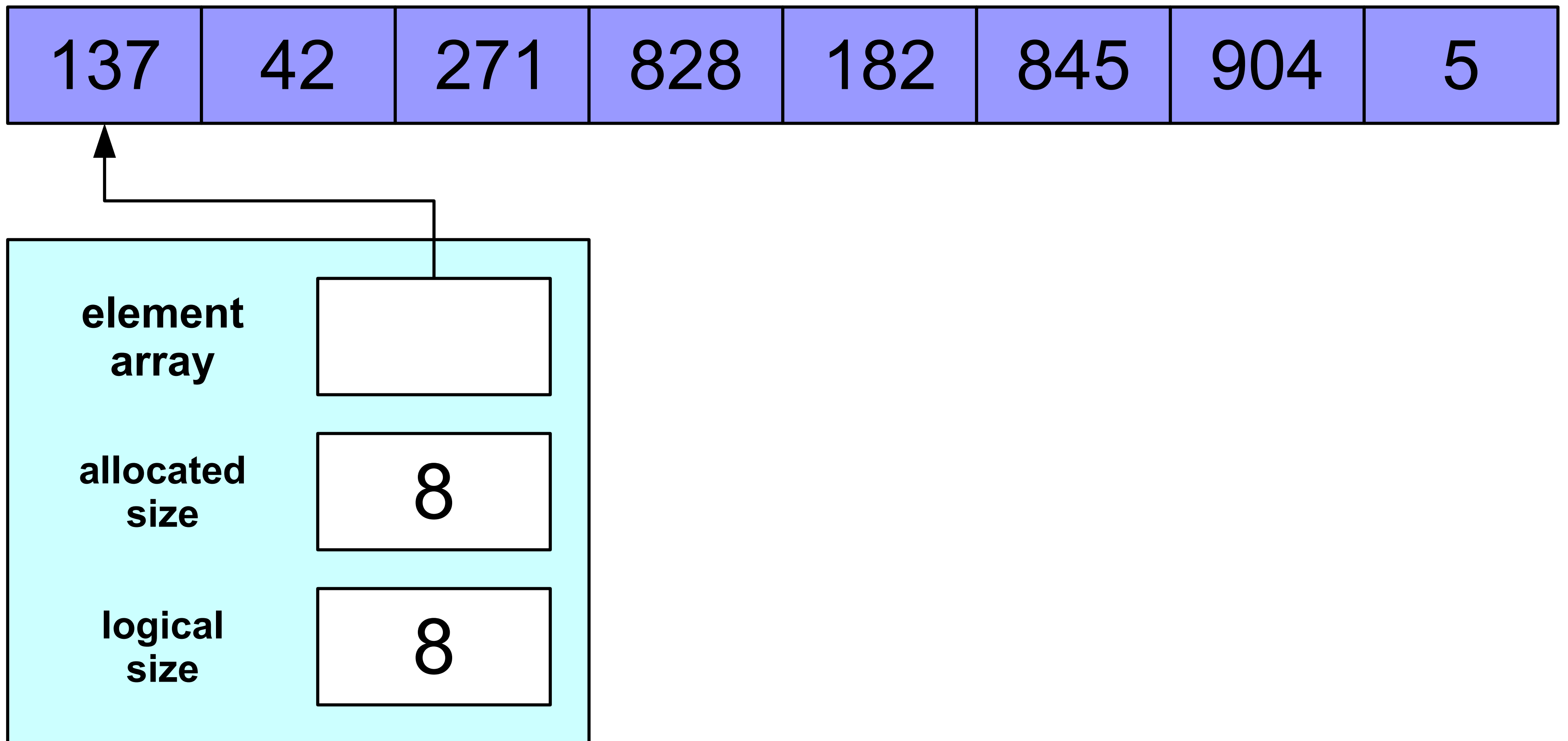
A Much Better Idea



A Much Better Idea



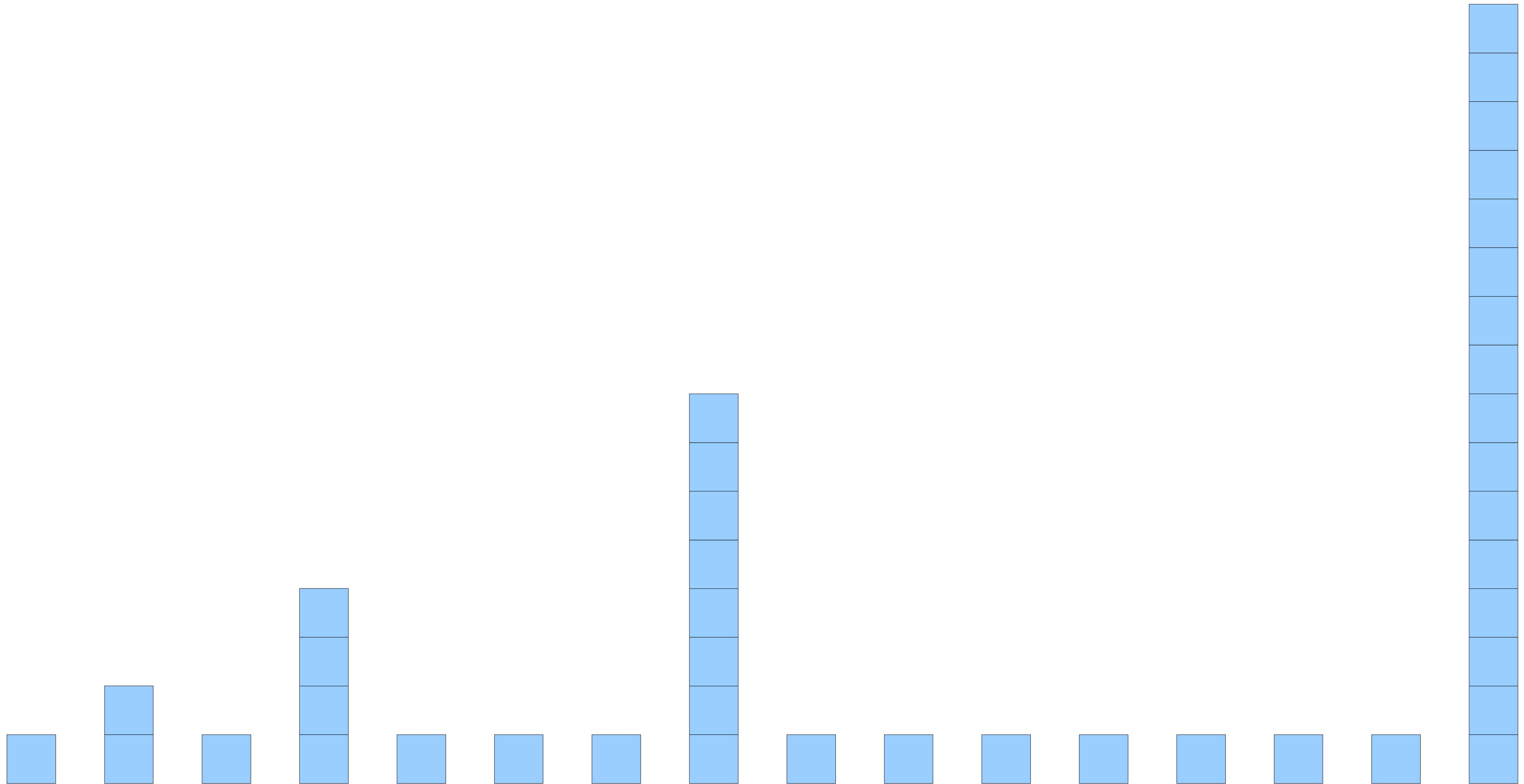
A Much Better Idea



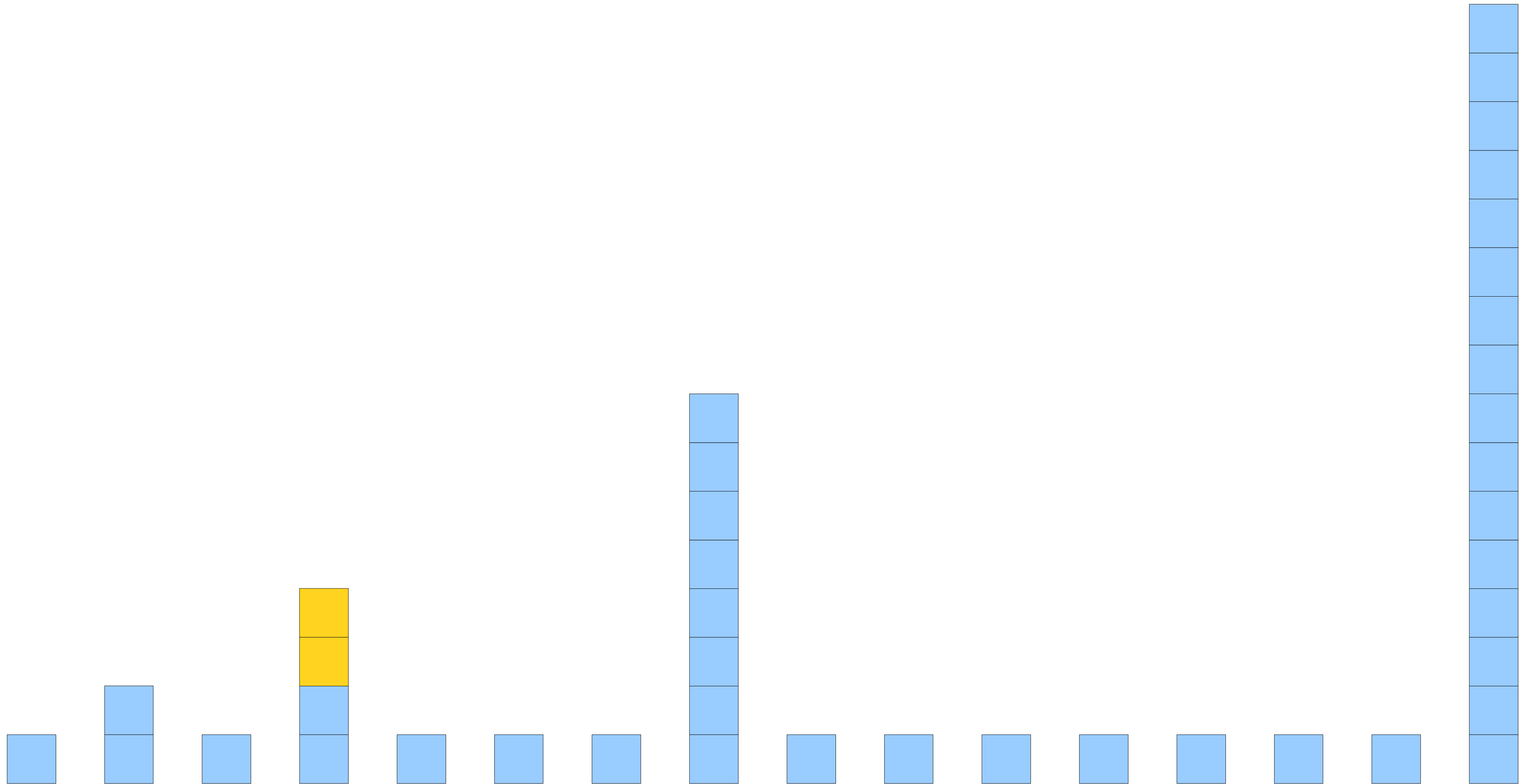
Let's Give it a Try!

How do we analyze this?

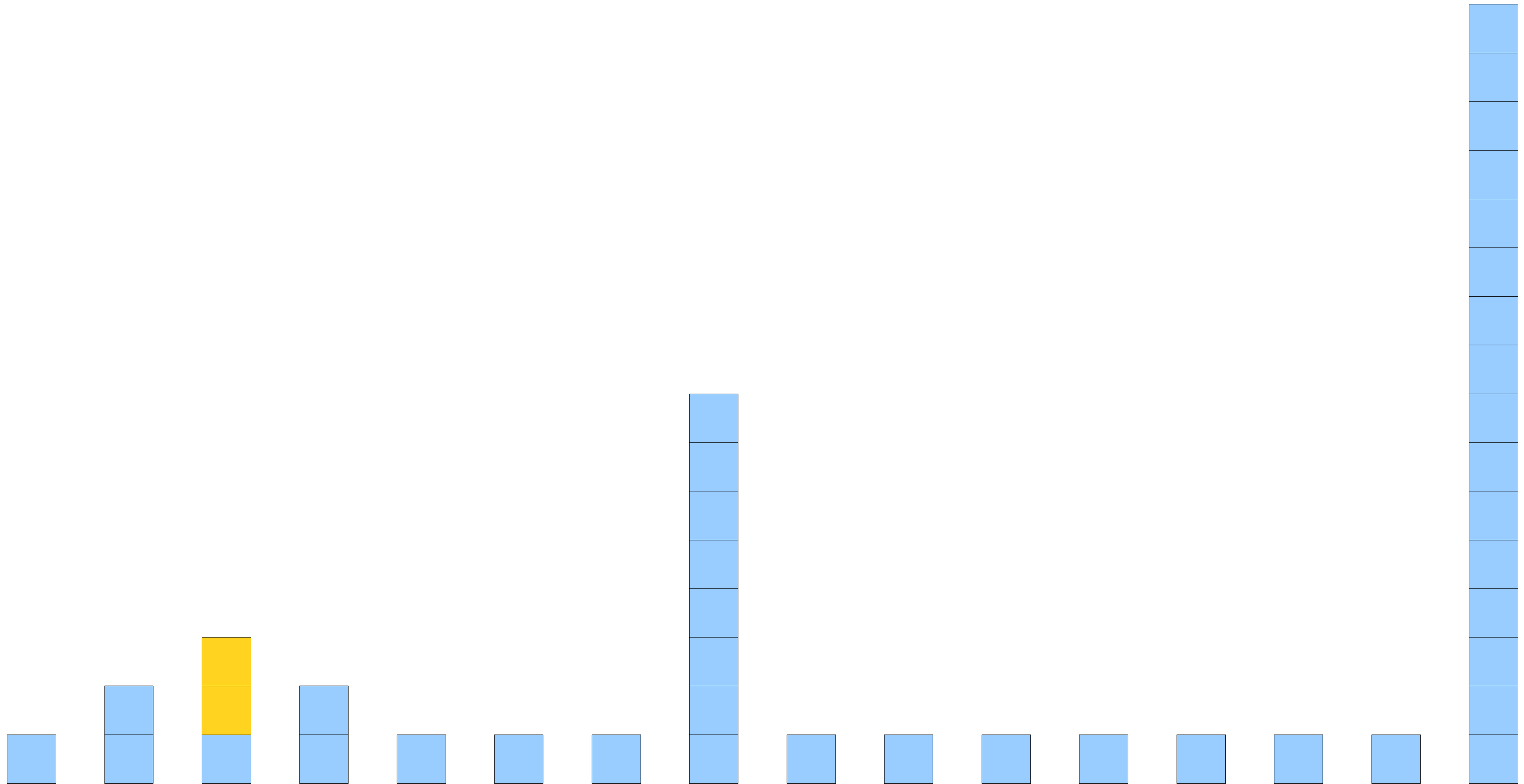
Spreading the Work



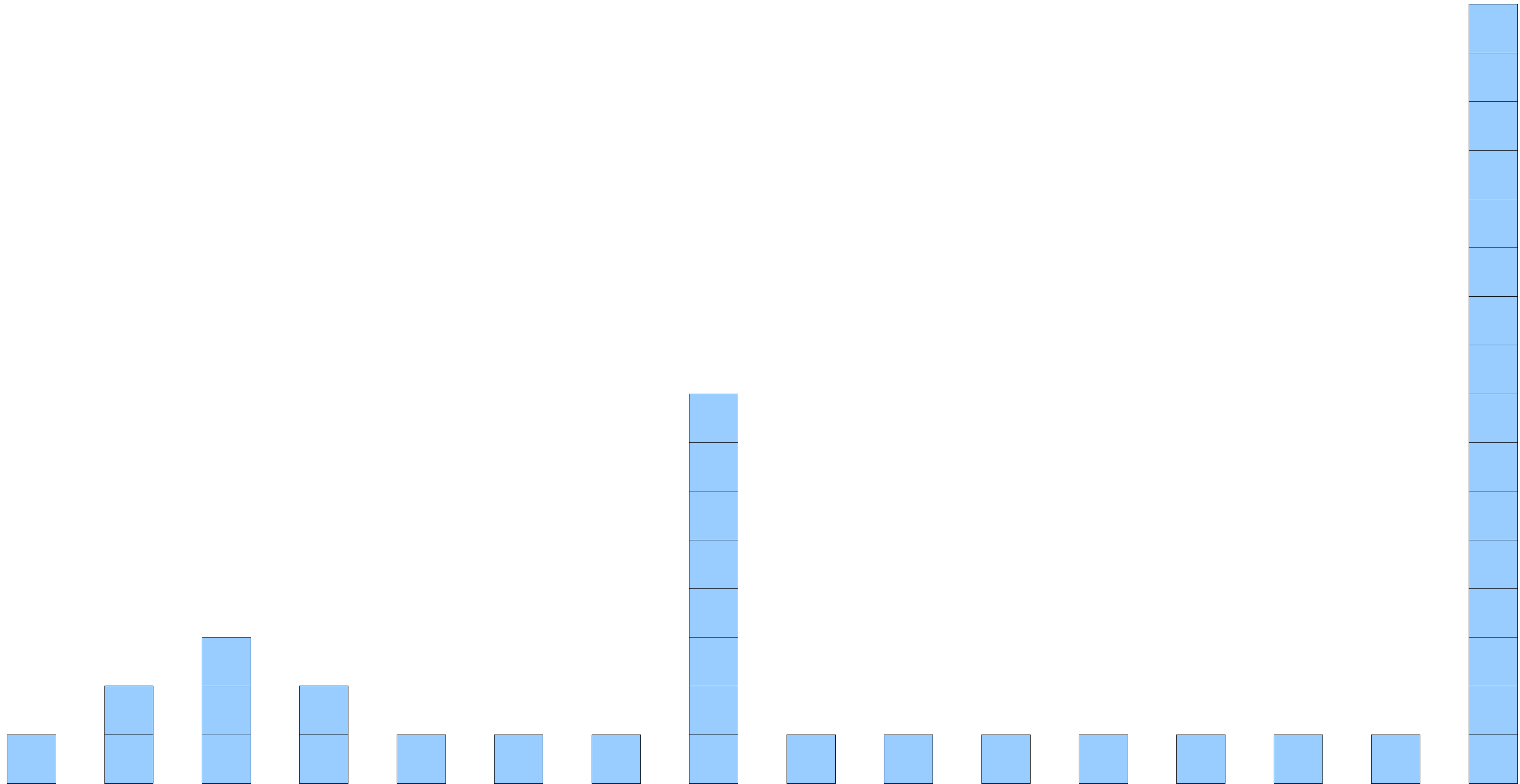
Spreading the Work



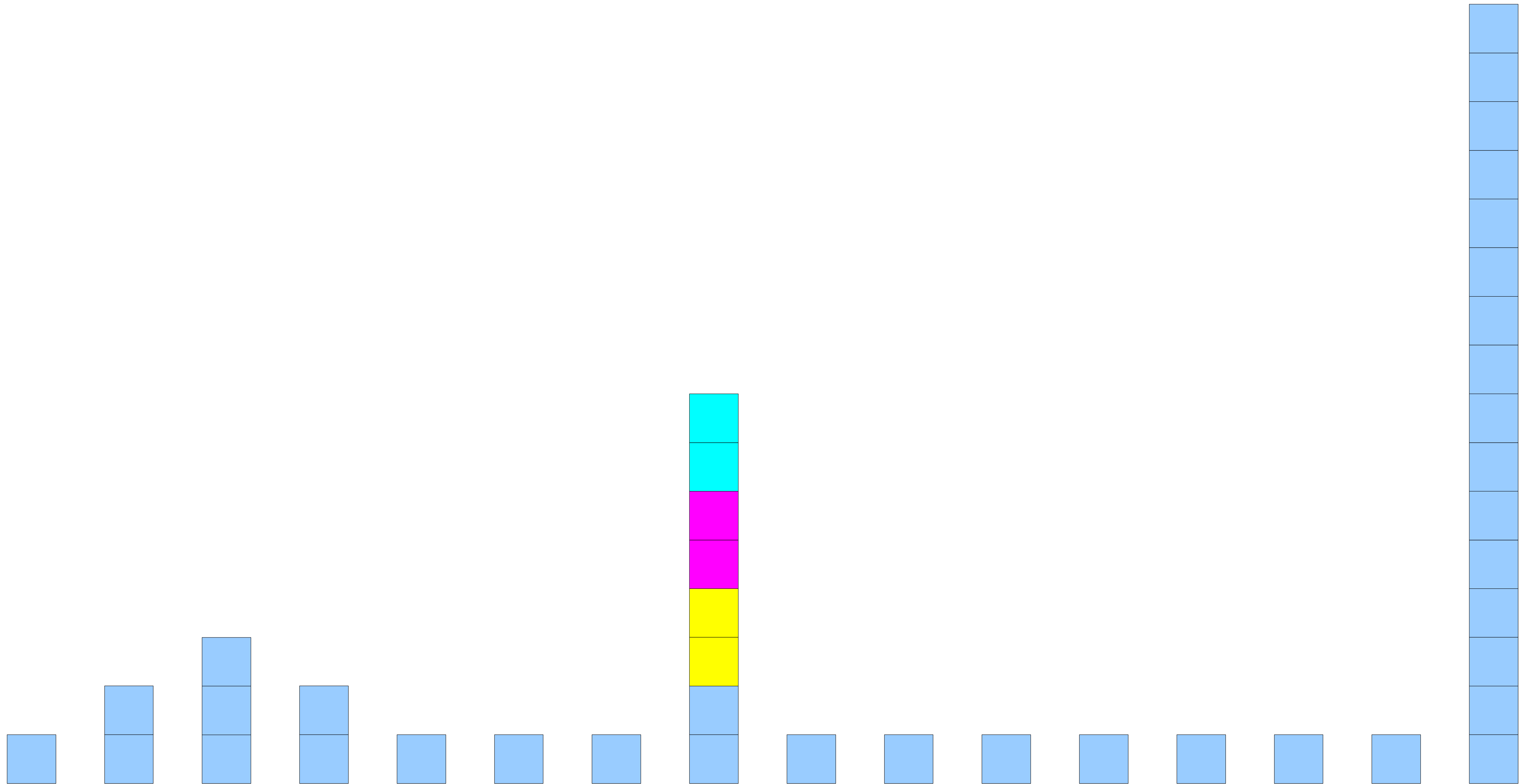
Spreading the Work



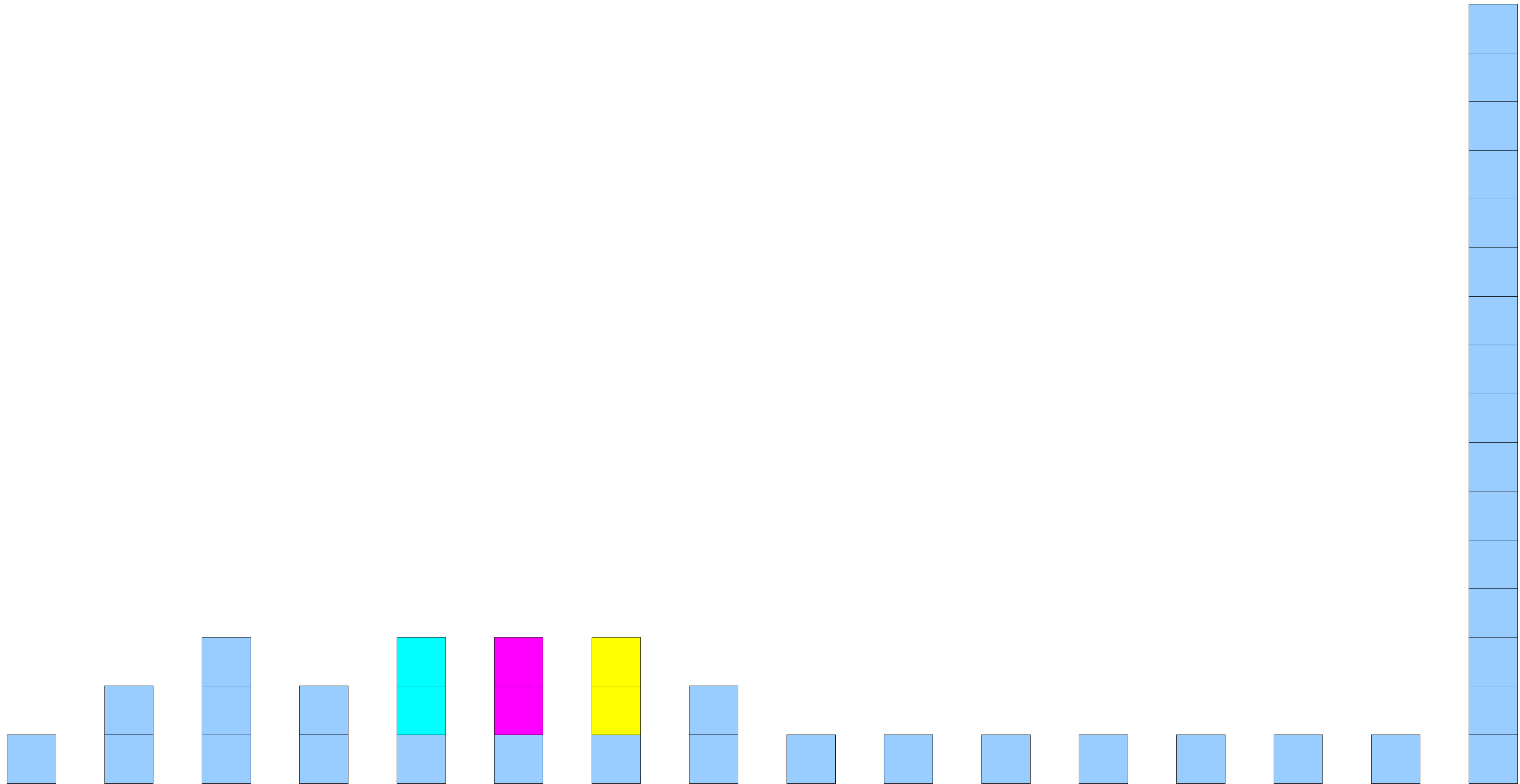
Spreading the Work



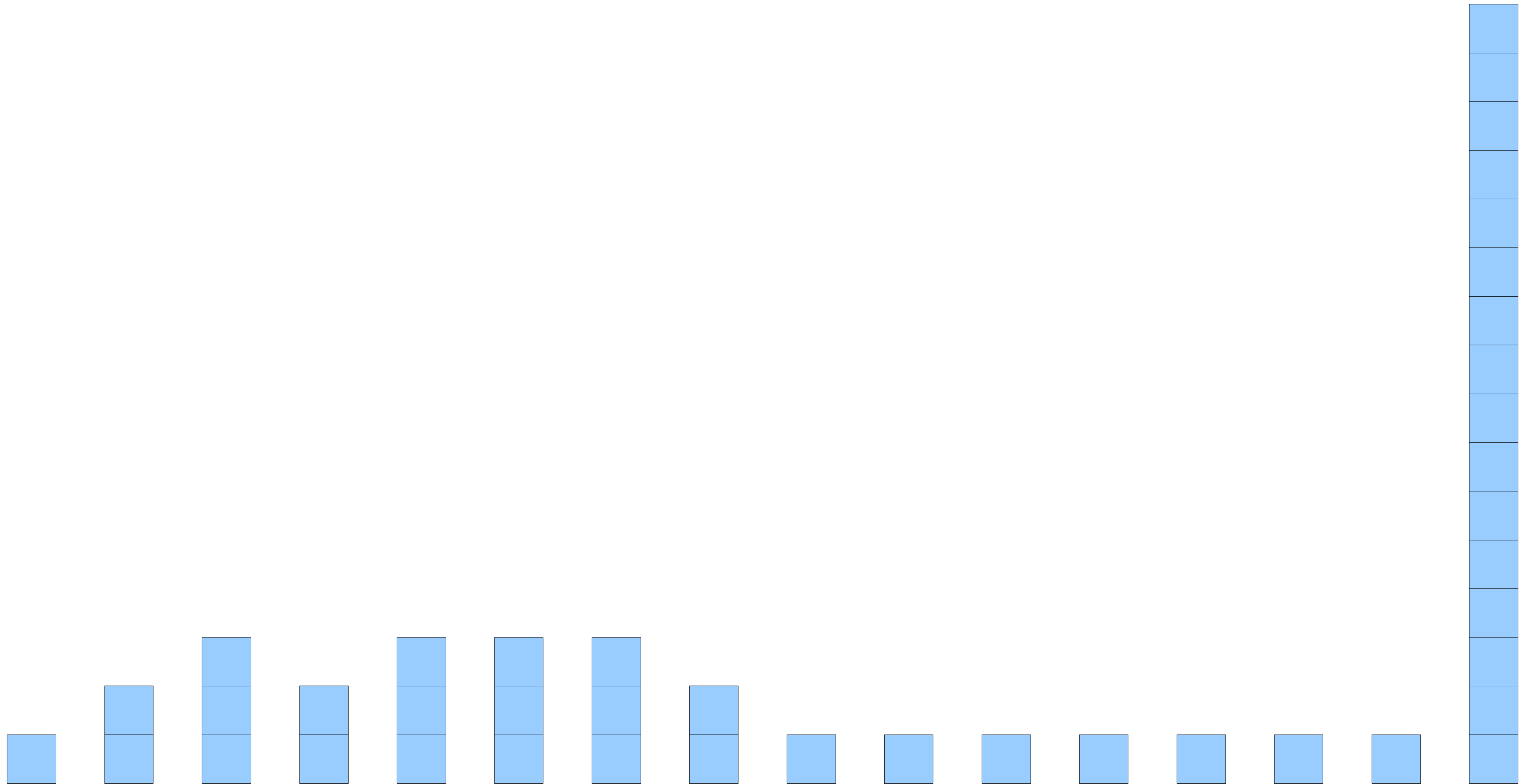
Spreading the Work



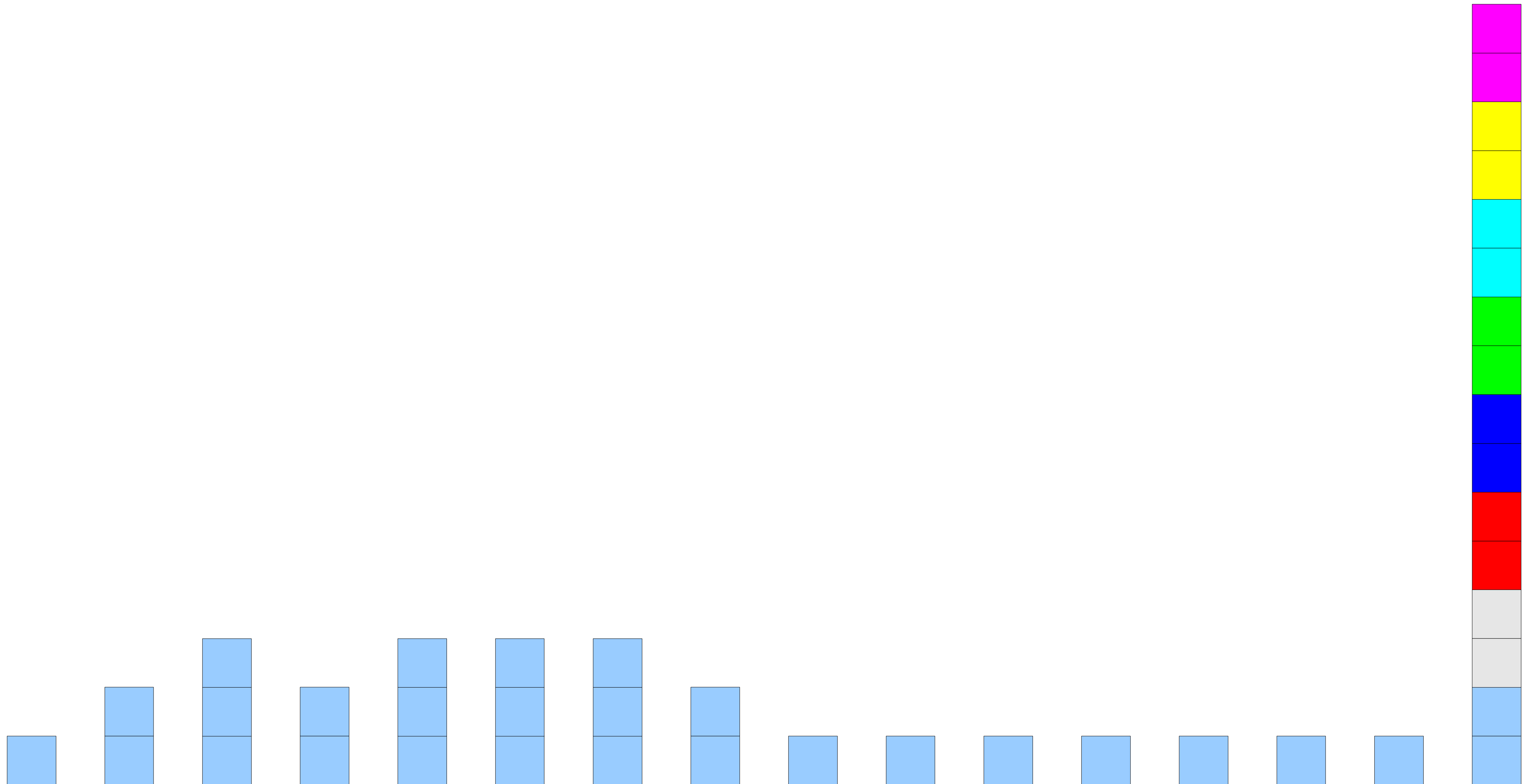
Spreading the Work



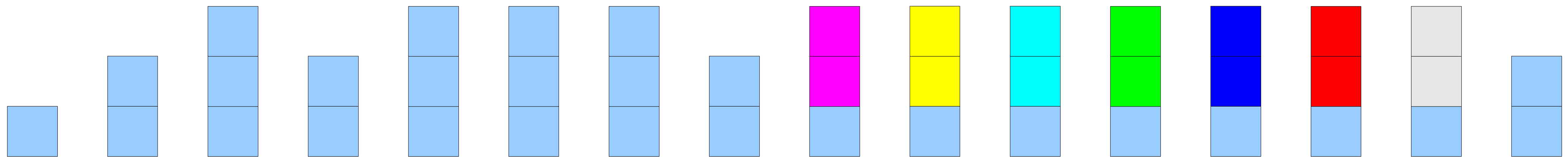
Spreading the Work



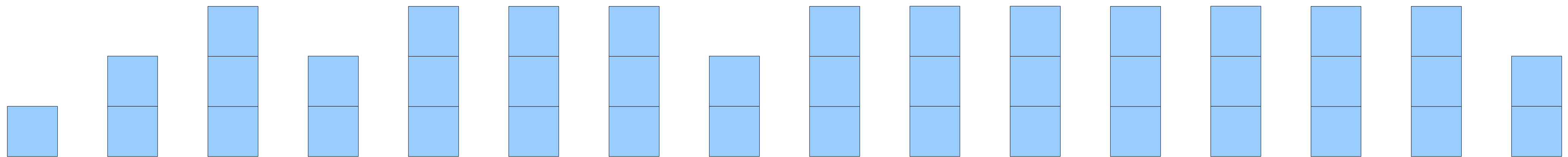
Spreading the Work



Spreading the Work



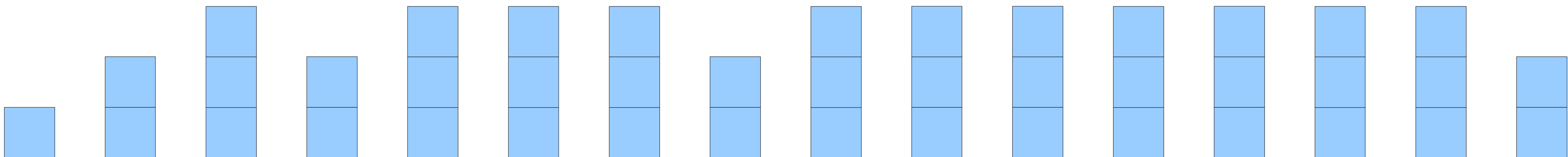
Spreading the Work



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).
- Heavy pushes become so rare that the ***average*** time for a push is O(1).
- Cost of n pushes:
  - 1 + 1 + 1 + ... + 1 = **O(*n*)**.
- Cost of n pops:
  - 1 + 1 + 1 + ... + 1 = **O(*n*)**.
- Total work done: **O(*n*)**.
- Can we confirm this?

Amortized Analysis

- The analysis we have just done is called an ***amortized analysis***.
- We reason about the total 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`.

Your Action Items

- ***Keep working on Assignment 5***
  - Haven't started yet? Not a problem! You've got time if you make slow and steady progress from here on out.
  - Need help? Stop by the LaIR!

Next Time

- ***Hash Functions***
  - A magical and wonderful gift from the world of mathematics.
- ***Hash Tables***
  - How do we implement `HashMap` and `HashSet`?