

# Implementing an ADT

Amrita Kaur

July 25, 2023

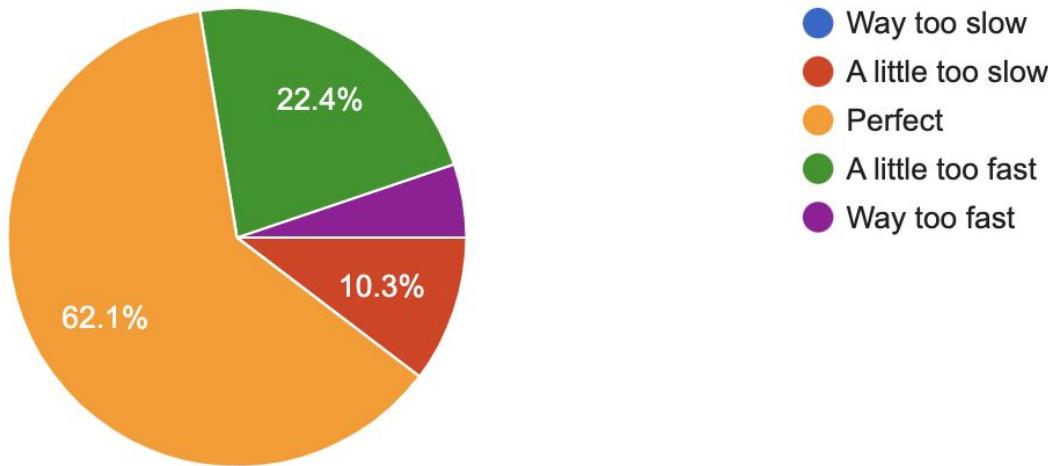
# Announcements

- Assignment 3 due Wednesday at 11:59pm
- Minor change to our grading scheme
  - Attendance bonus can be applied to either the final or the midterm
    - Downweight final from 25% to as low as 20%, **OR**
    - Downweight midterm from 10% to as low as 5%
  - Will calculate both grades and take the better of the two
- Any talking in this room gets amplified, and that's by design! So please don't talk during lecture, because it can be distracting :)

# Week 4 Feedback

Rate the pace of lecture

58 responses



# Week 4 Feedback

Things you liked:

“I've really enjoyed **working through examples in class**, even if it's pseudo code”

“Live examples on the **white board!!!**”

“**Walking through code** step by step”

“**Very understanding** to any situation”

“Really appreciate you all **taking the feedback into account** and improving the course structure/format based on students' comments”

# Week 4 Feedback

Places we can improve:

“On longer examples, a little extra time perhaps with a **couple baby steps along the way** would go a long way in being able to fully create and understand examples.”

“**it's hard to see the whiteboard**”

“Maybe **more real-life examples** to show how the basic concepts we are learning”

# Week 4 Feedback

We hear you...

“Allocating **more time** to doing problems together as a class.”

“I think maybe have **more time** for us to do the problems ourselves”

“I would enjoy a little less lecturing and **more coding examples.**”

“Speed of the lecture could **slow down**”

“I wish that there could be opportunities on the homework for us to have some space to think for ourselves and practice our problem-solving strategies...In my opinion, there's something to be said for **learning by doing**, as opposed to learning by guided walkthrough.” - **Try out some extensions!**

# Week 4 Feedback

Anything else you would like us to know:

“Thank you for listening to our feedback!”

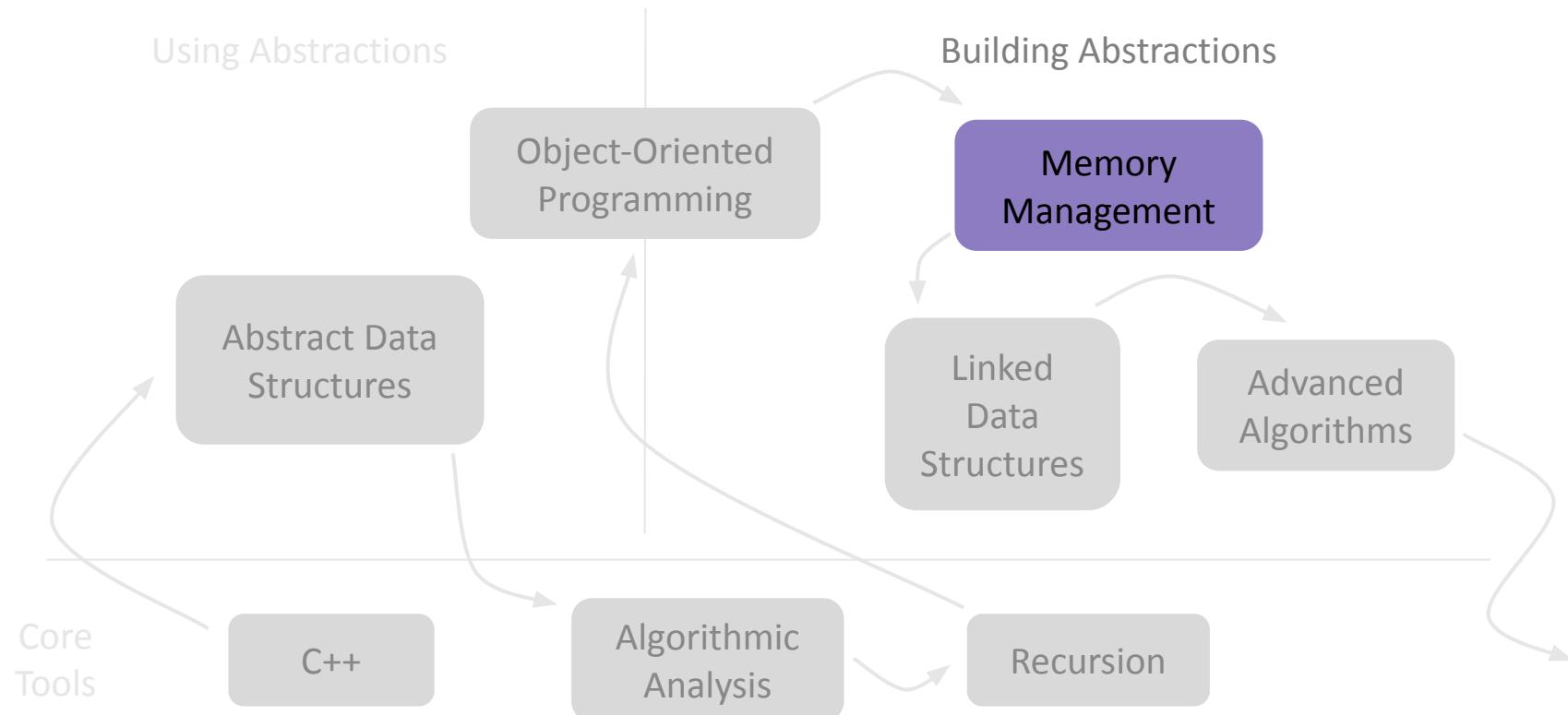
“I almost lost my water bottle 3 times in 106B section and lecture this week, each incident was uniquely stressful.”

“Homesickness sucks.”

“Watch the Barbie Movie!”

“ty”

# Roadmap



# Question from Last Class

- In an implicit constructor, does an `int` default initialize to 0?
  - Nope!
  - You will need to set the value explicitly to 0 in a constructor if that is what you want

# Memory on Stack vs Heap

```
Vector<string> varOnStack;
```

- Until today, all variables we've created get defined on the **stack**
- This is static memory allocation
- Variables on the stack are stored directly to the memory and access to this memory is very fast
- We don't have to worry about memory management

# Memory on Stack vs Heap

```
Vector<string> varOnStack;
```

- Until today, all variables we've created get defined on the **stack**
- This is static memory allocation
- Variables on the stack are stored directly to the memory and access to this memory is very fast
- We don't have to worry about memory management

```
string* arr = new string[numValues];
```

- We can now request memory from the **heap**
- This is dynamic memory allocation
- We have more control over variables on the heap
- But this means that we also have to handle the memory we're using carefully and properly clean it up when done

# Dynamic Memory Allocation: `new`

- To request memory from the heap to allocate one element:

```
type* variable = new type;
```

- To allocate multiple (n) elements on the heap:

```
type* variable = new type[n];
```

# Dynamic Memory Allocation: new

```
type* variable = new type;
```

Declaring a variable that will point at our newly-allocated memory

- Name is variable
- Type is type\* (match the type of the element)

Allocating heap memory with the new keyword

Assigning the pointer to point to the heap memory

# Dynamic Memory Allocation: Examples

```
int* anInteger = new int;
```

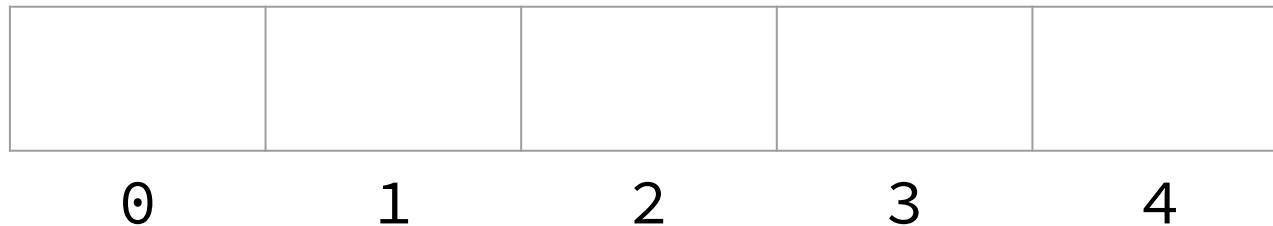
```
int* tenInts = new int[10];
```

# Pointers

- Data type, that like all other data types, takes up space in memory and stores specific values
- **Always stores a memory address**, which is like the specific coordinates of where a piece of memory exists on the computer
- Quite literally "points" to another location on your computer

# Arrays

- Lower-level and more limited than Vectors
- A contiguous chunk of space in the computer's memory, split into slots, each of which can contain one piece of information
  - Contiguous means that each slot is located directly next to the others (There are no "gaps")
  - Have a specific type which dictates what information can be held in each slot
  - Each slot has an "index" by which we can refer to it



# Arrays

```
int firstTen[10];  
int* secondTen = new int[10];  
// fill memory with values  
for (int i = 0; i < 10; i++) {  
    firstTen[i] = i * 2; // evens  
    secondTen[i] = i * 2 + 1; // odds  
}  
int len = firstTen.length(); // ERROR! No functions!  
firstTen.add(42); // ERROR! No functions!
```

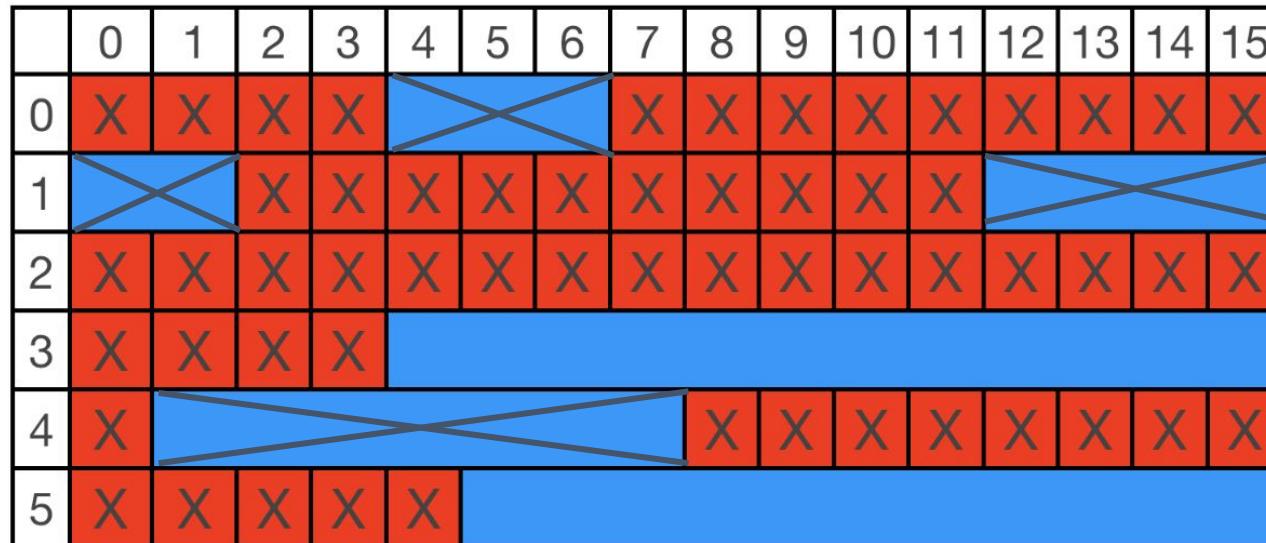
# Under the Hood

```
int* tenInts = new int[10];
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0	X	X	X	X					X	X	X	X	X	X	X	X	
1				X	X	X	X	X	X	X	X	X	X				
2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
3	X	X	X	X													
4	X							X	X	X	X	X	X	X	X	X	
5	X	X	X	X	X												

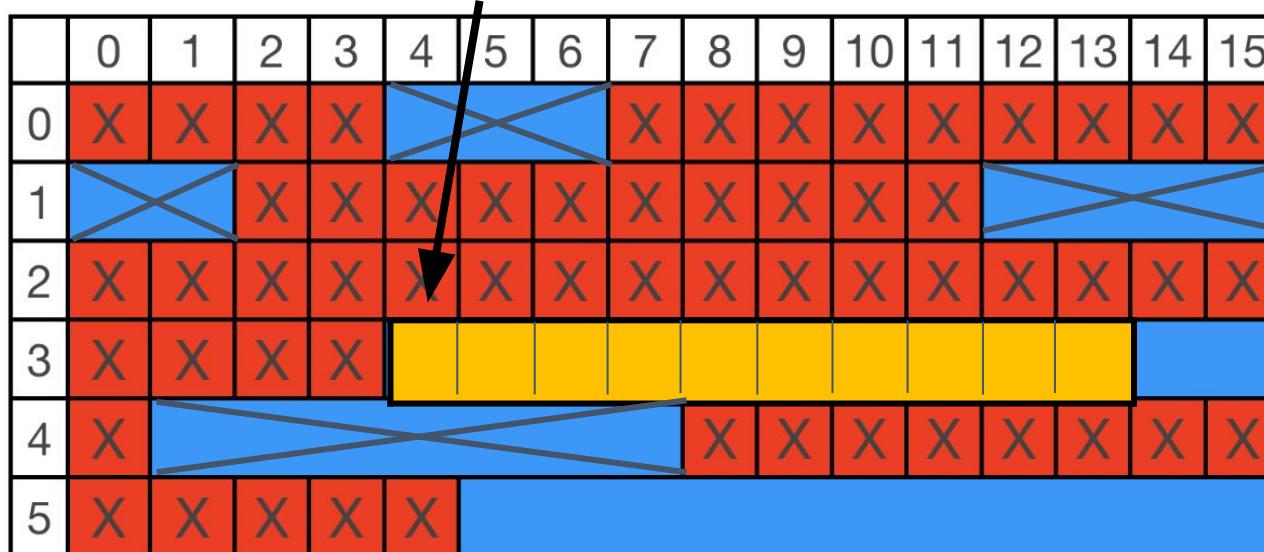
# Under the Hood

```
int* tenInts = new int[10];
```



# Under the Hood

```
int* tenInts = new int[10];  
(Col 4, Row 3)
```



# Pitfalls and Dangers

- The array you get from `new[]` is **fixed-size**: it can neither grow nor shrink once it's created
- The array you get from `new[]` has **no bounds-checking**: accessing anything past the beginning or end of an array triggers undefined behavior

# Cleaning Up

- When declaring local variables or parameters, C++ automatically handles memory allocation and deallocation for you
- When using `new`, you are responsible for deallocated the memory you allocate
- If you don't, you get a memory leak
  - Your program will never be able to use that memory again
  - Too many leaks can cause a program to crash – it's important to not leak memory!

# Cleaning Up: delete

- You can deallocate (free) memory with the `delete` keyword
- To deallocate a single element:

```
delete var;
```

- To deallocate an array of elements:

```
delete[] arr;
```

# Cleaning Up: delete

- This destroys the array pointed to by the given pointer, not the pointer itself
  - You can think of this operation as relinquishing control over the memory back to the computer
- Once you've deleted the memory pointed at by a pointer, you have a dangling pointer and shouldn't read or write from it.

# Designing OurVector



# What is OurVector?

- Goal: Let's make our very own version of the Stanford C++ Vector that we've been using all quarter long
  - It all will feel so much cooler when we've built it ourselves!
- We will only implement a subset of the functionality that the Stanford Vector provides
  - OurVector will **only store integers** and will not be configurable to store other types
    - Generic, or "templated" classes that allow the client to specify the data type that is stored, are possible in C++, but they are beyond the scope of this class.
  - At first, OurVector will be limited to **storing a fixed number of elements**, but we will lift this restriction by the end of class. For now, if we run out space we'll just throw an error.

# Three Main Parts

- Member variables (*What subvariables make up this new variable type?*)
- Member functions (*What functions can you call on a variable of this type?*)
- Constructor (*What happens when you make a new instance of this type?*)

# Three Main Parts

- Member variables (*What subvariables make up this new variable type?*)
  - What private information will we need to store in order to keep track of the data stored in OurVector?
- Member functions (*What functions can you call on a variable of this type?*)
  - What public interface should OurVector support? What functions might a client want to call?
- Constructor (*What happens when you make a new instance of this type?*)
  - How are the member variables initialized when a new instance of OurVector is created?

# Three Main Parts

- Member variables (*What subvariables make up this new variable type?*)
  - What private information will we need to store in order to keep track of the data stored in `OurVector`?
- Member functions (*What functions can you call on a variable of this type?*)
  - What public interface should `OurVector` support? What functions might a client want to call?
- Constructor (*What happens when you make a new instance of this type?*)
  - How are the member variables initialized when a new instance of `OurVector` is created?

# OurVector Header File

```
class OurVector {  
public:  
    /* What goes here? */  
  
private:  
    /* To be defined soon! */  
};
```

# OurVector Header File

```
class OurVector {  
public:  
    void add(int value);  
    void insert(int index, int value);  
    int get(int index);  
    void remove(int index);  
    int size();  
    bool isEmpty();  
private:  
    /* To be defined soon! */  
};
```

# Three Main Parts

- **Member variables** (*What subvariables make up this new variable type?*)
  - **What private information will we need to store in order to keep track of the data stored in OurVector?**
- **Member functions** (*What functions can you call on a variable of this type?*)
  - What public interface should OurVector support? What functions might a client want to call?
- **Constructor** (*What happens when you make a new instance of this type?*)
  - How are the member variables initialized when a new instance of OurVector is created?

# OurVector Member Variables

# OurVector Member Variables

- `int* elements;`
  - A pointer to an array of integers, which will act as our underlying data storage mechanism

# OurVector Member Variables

- `int* elements;`
  - A pointer to an array of integers, which will act as our underlying data storage mechanism
- `int allocatedCapacity;`
  - An integer that stores the size of the allocated elements array.
  - Arrays don't have any conception/knowledge of their own size, so we must manually track this!

# OurVector Member Variables

- `int* elements;`
  - A pointer to an array of integers, which will act as our underlying data storage mechanism
- `int allocatedCapacity;`
  - An integer that stores the size of the allocated elements array.
  - Arrays don't have any conception/knowledge of their own size, so we must manually track this!
- `int numItems;`
  - An integer that stores the number of elements currently stored in the vector

# OurVector Header File

```
class OurVector {  
public:  
    void add(int value);  
    void insert(int index, int value);  
    int get(int index);  
    void remove(int index);  
    int size();  
    bool isEmpty();  
private:  
    int* elements;  
    int allocatedCapacity;  
    int numItems;  
};
```

# Three Main Parts

- Member variables (*What subvariables make up this new variable type?*)
  - What private information will we need to store in order to keep track of the data stored in OurVector?
- Member functions (*What functions can you call on a variable of this type?*)
  - What public interface should OurVector support? What functions might a client want to call?
- Constructor (*What happens when you make a new instance of this type?*)
  - How are the member variables initialized when a new instance of OurVector is created?

# OurVector Header File

```
class OurVector {  
public:  
    // Constructor  
  
    void add(int value);  
    void insert(int index, int value);  
    int get(int index);  
    void remove(int index);  
    int size();  
    bool isEmpty();  
private:  
    int* elements;  
    int allocatedCapacity;  
    int numItems;  
};
```

# OurVector Header File

```
class OurVector {  
public:  
    OurVector();  
  
    void add(int value);  
    void insert(int index, int value);  
    int get(int index);  
    void remove(int index);  
    int size();  
    bool isEmpty();  
private:  
    int* elements;  
    int allocatedCapacity;  
    int numItems;  
};
```

# OurVector Constructor

Must initialize all the values of our member variables to be things that initially make sense

- elements
- allocatedCapacity
- numItems

# OurVector Constructor

Must initialize all the values of our member variables to be things that initially make sense

- `elements`: should be allocated using the `new[]` keyword
- `allocatedCapacity`
- `numItems`

# OurVector Constructor

Must initialize all the values of our member variables to be things that initially make sense

- `elements`: should be allocated using the `new[]` keyword
- `allocatedCapacity`: should be set to some small integer
- `numItems`

# OurVector Constructor

Must initialize all the values of our member variables to be things that initially make sense

- `elements`: should be allocated using the `new[]` keyword
- `allocatedCapacity`: should be set to some small integer
- `numItems`: should be initialized to 0

# Let's Code It Up!

Member Variables and Constructor

# Destructor

- Specially defined method for classes
  - Special naming convention defined as `~ClassName()`
- Does not take in parameters and does not return anything
- Automatically called when the object's lifetime ends (for example, if it's a local variable that goes out of scope)
- Responsible for cleaning up an object's memory

# OurVector Header File

```
class OurVector {  
public:  
    OurVector();  
    // Destructor  
    void add(int value);  
    void insert(int index, int value);  
    int get(int index);  
    void remove(int index);  
    int size();  
    bool isEmpty();  
private:  
    int* elements;  
    int allocatedCapacity;  
    int numItems;  
};
```

# OurVector Header File

```
class OurVector {  
public:  
    OurVector();  
    ~OurVector();  
    void add(int value);  
    void insert(int index, int value);  
    int get(int index);  
    void remove(int index);  
    int size();  
    bool isEmpty();  
private:  
    int* elements;  
    int allocatedCapacity;  
    int numItems;  
};
```

# OurVector Destructor

- Take responsibility for freeing any allocated memory currently in use by an instance of the class
- This means calling the `delete[]` operator on the `elements` array to officially give that memory back to the computer and avoid any memory leaks
- The other member variables (`allocatedCapacity` and `numItems`) are both simple stack-allocated variables, so nothing special is needed to clean them up

# Let's Code It Up!

Destructor

# Takeaways

- **Member variables** define the key data storage components of a class implementation.
- The **constructor** is the special method that gets called when a new instance of a class is declared. In this method, we initialize all of our member variables to the appropriate values, including allocating any necessary memory.
- The **destructor** is a special method that gets called when an instance of a class goes out of scope and thus is destroyed. In this method, we most often are responsible for freeing any dynamically allocated memory used by the instance.

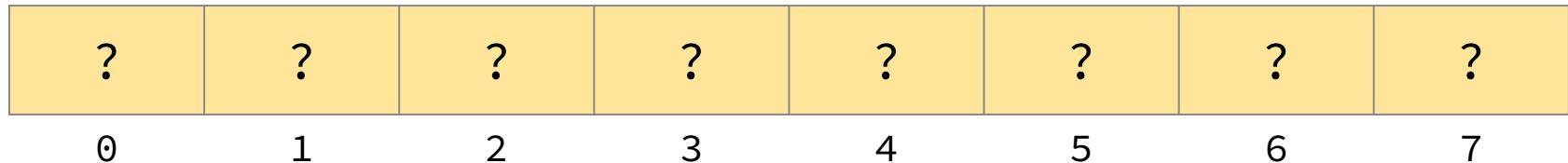
# Visualizing Operations

# Initialization via the Constructor

# Initialization via the Constructor

```
// client code  
  
OurVector vec;
```

# Initialization via the Constructor

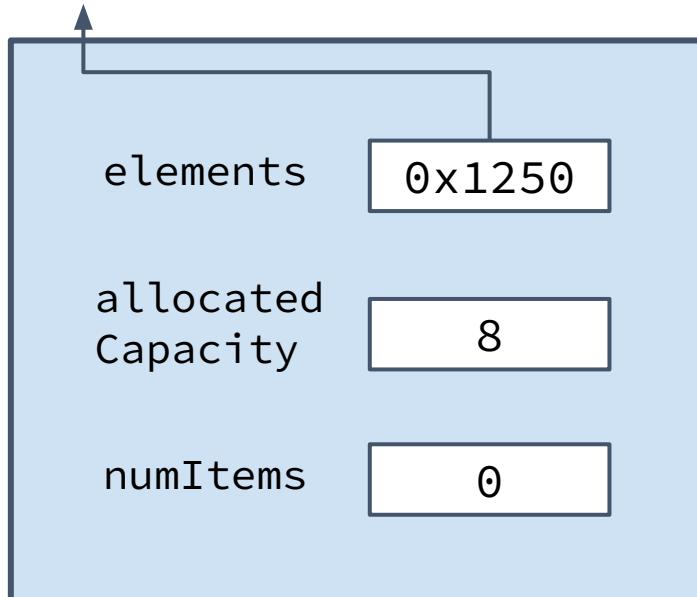
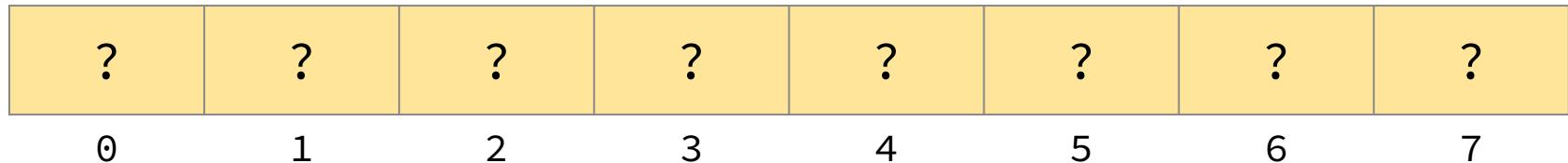


```
// client code  
  
OurVector vec;
```

# Adding Elements

`add()` : takes a specified element and adds it to the first open spot at the end of the vector

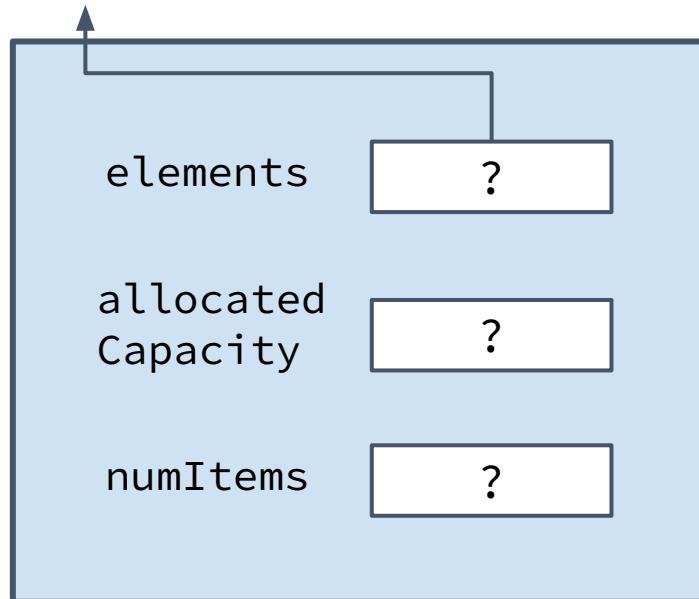
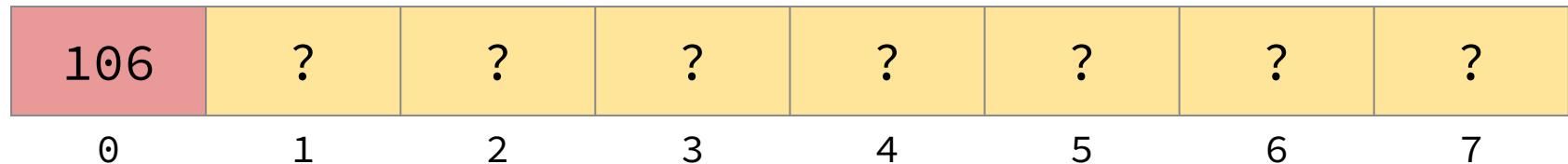
# Adding Elements



```
// client code
```

```
OurVector vec;  
vec.add(106);
```

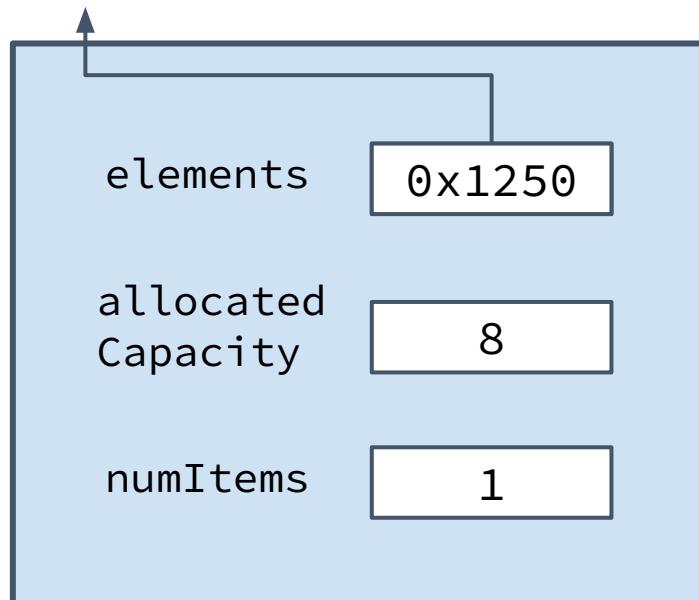
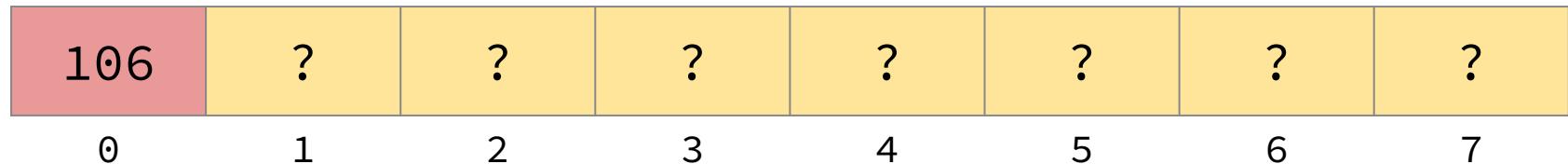
# Adding Elements



```
// client code
```

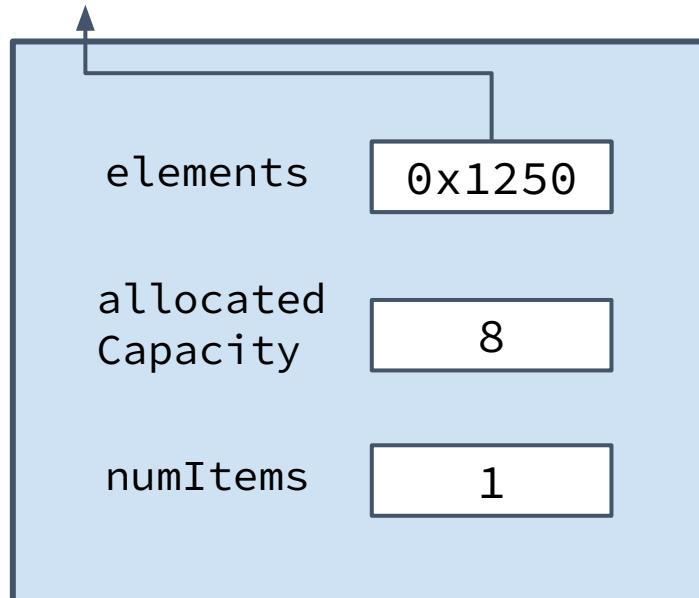
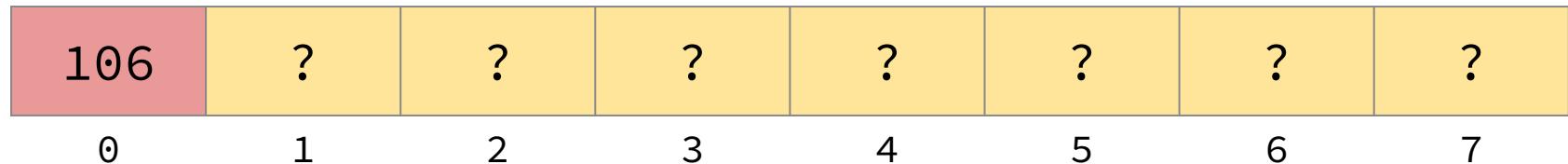
```
OurVector vec;  
vec.add(106);
```

# Adding Elements



```
// client code  
  
OurVector vec;  
vec.add(106);
```

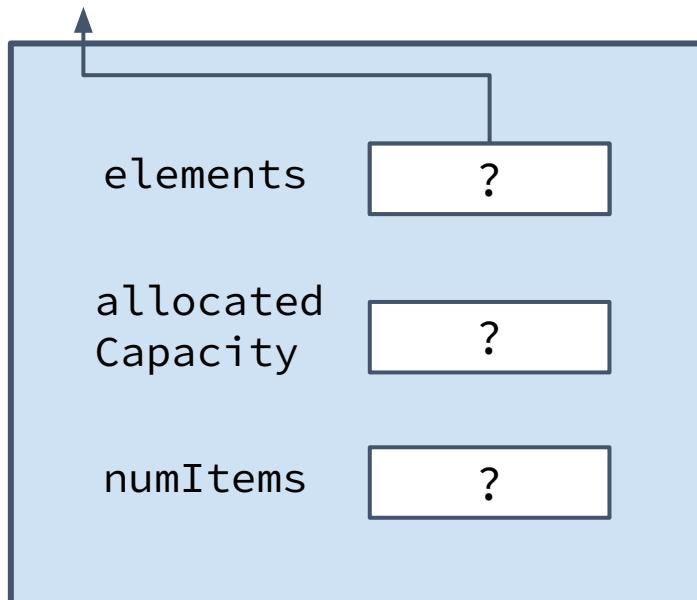
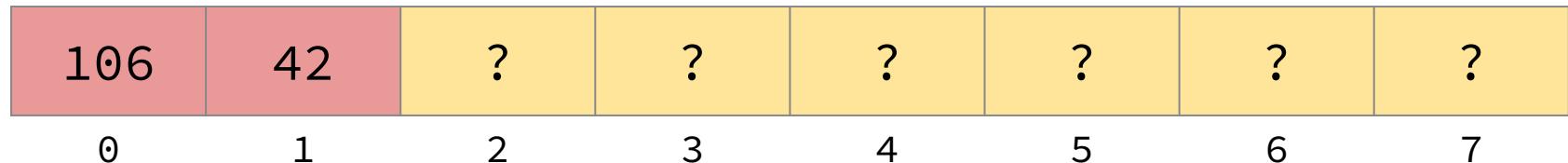
# Adding Elements



// client code

```
OurVector vec;  
vec.add(106);  
vec.add(42);
```

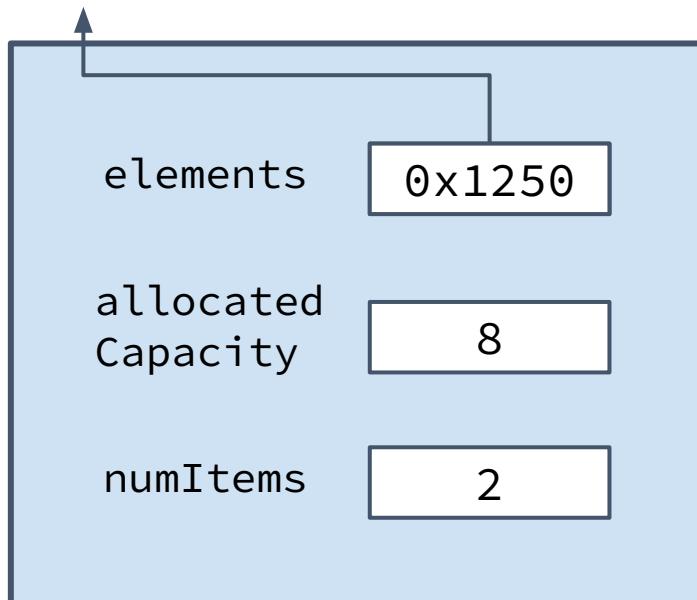
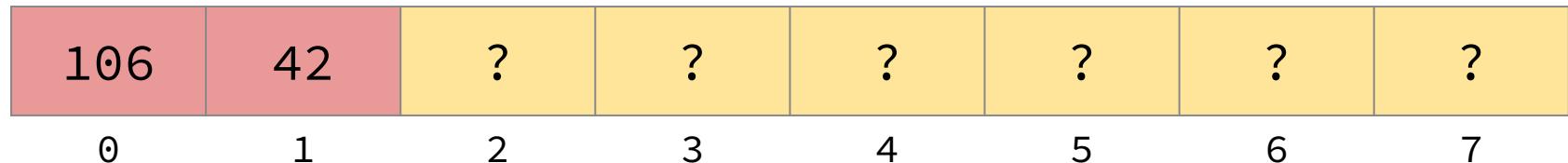
# Adding Elements



// client code

```
OurVector vec;  
vec.add(106);  
vec.add(42);
```

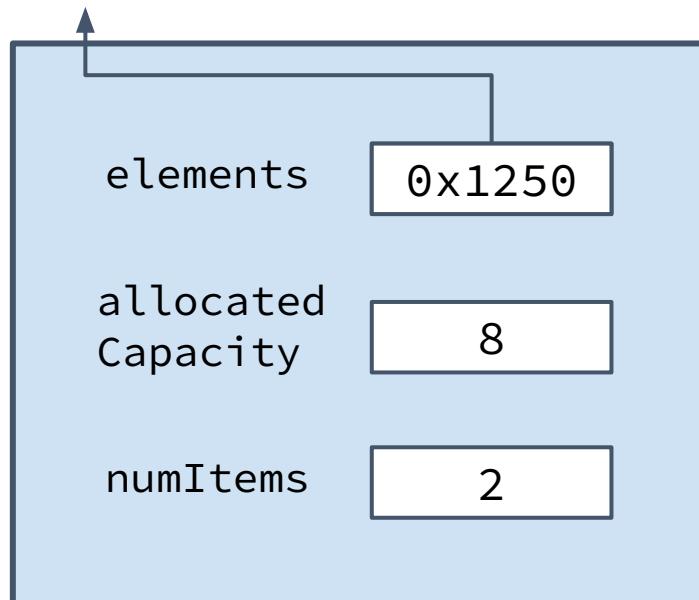
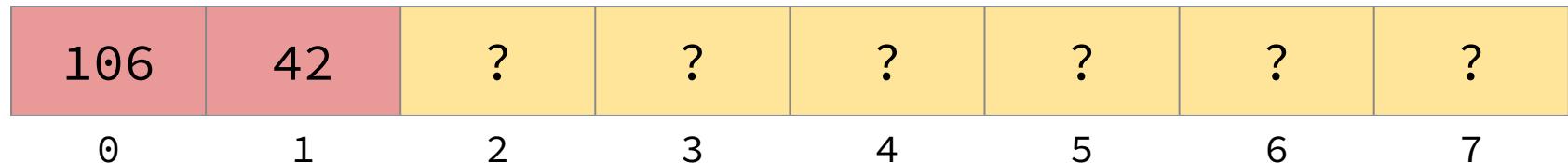
# Adding Elements



```
// client code
```

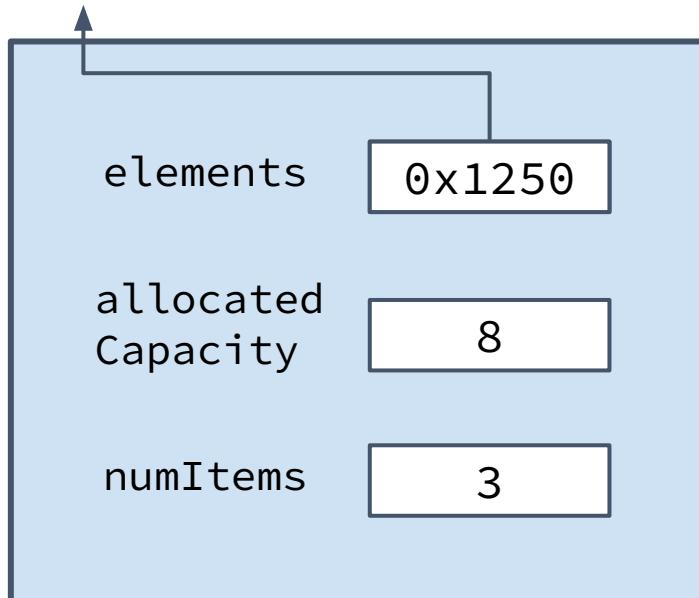
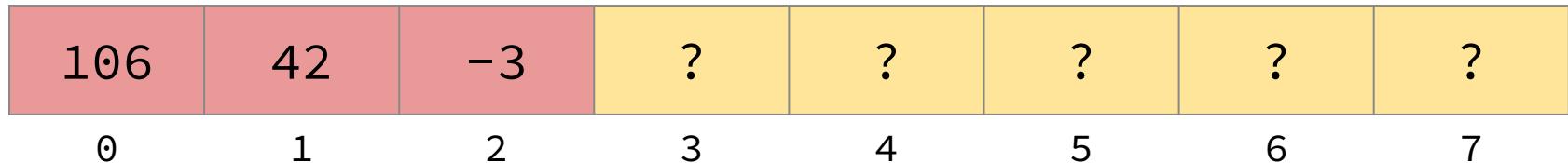
```
OurVector vec;  
vec.add(106);  
vec.add(42);
```

# Adding Elements



```
// client code  
  
OurVector vec;  
vec.add(106);  
vec.add(42);  
vec.add(-3);
```

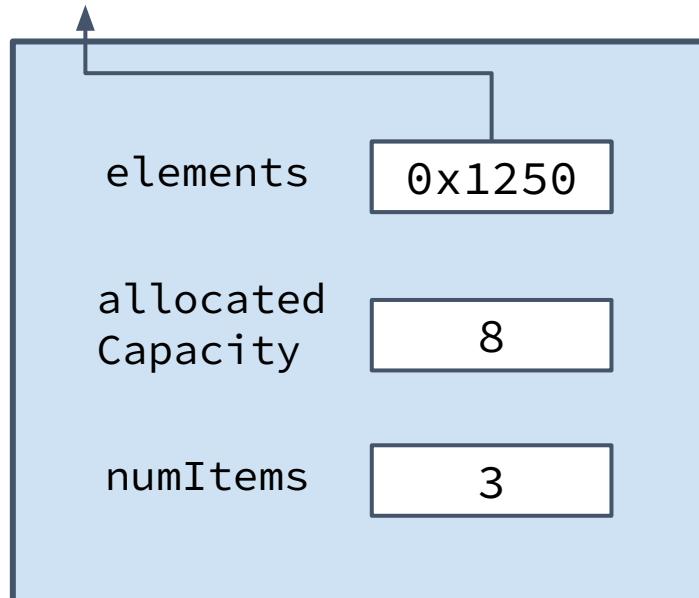
# Adding Elements



```
// client code
```

```
OurVector vec;  
vec.add(106);  
vec.add(42);  
vec.add(-3);
```

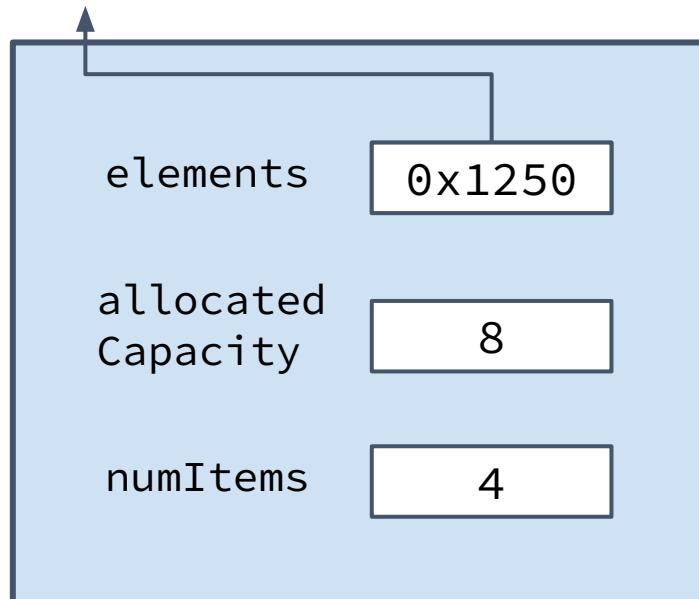
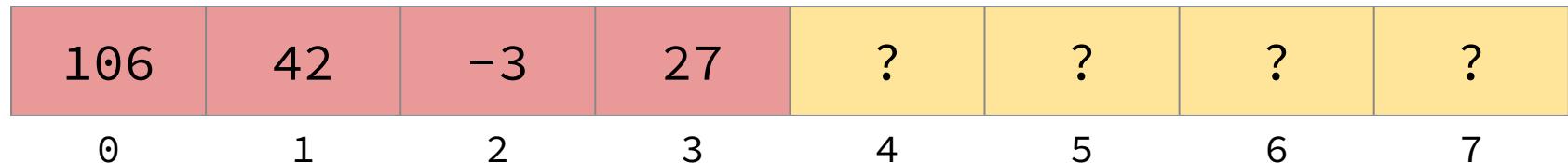
# Adding Elements



```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);
```

# Adding Elements



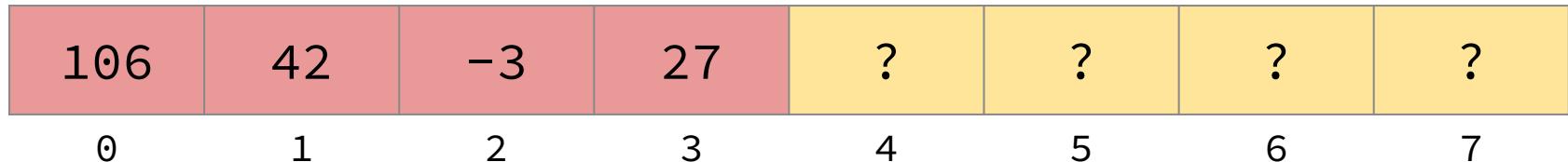
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);
```

# Removing Elements

`remove()` : allows the client to specify an index at which to remove an element, and then removes the value at that index

# Removing Elements

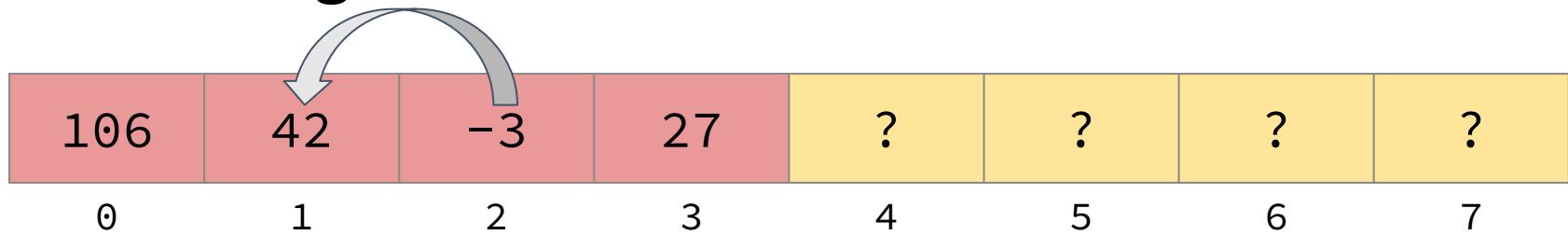


// client code

```
OurVector vec;  
vec.add(106);  
vec.add(42);  
vec.add(-3);  
vec.add(27);
```

```
vec.remove(1);
```

# Removing Elements

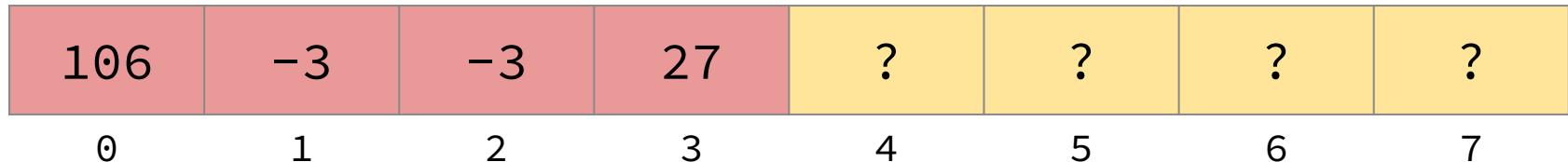


```
// client code
```

```
OurVector vec;  
vec.add(106);  
vec.add(42);  
vec.add(-3);  
vec.add(27);
```

```
vec.remove(1);
```

# Removing Elements

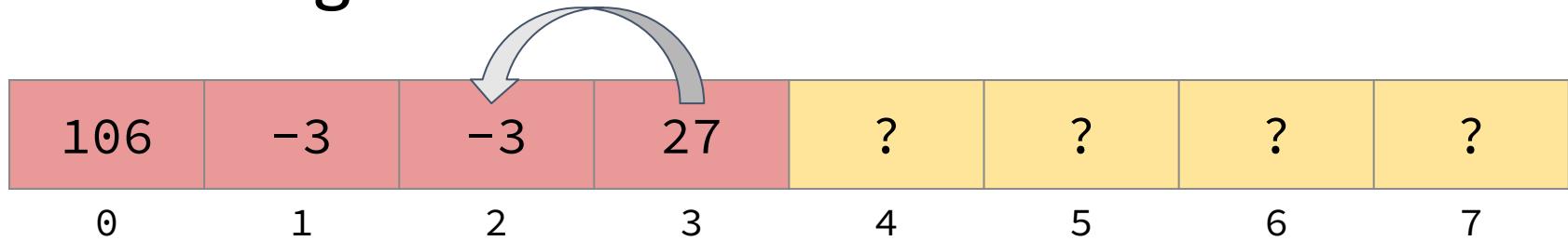


// client code

```
OurVector vec;  
vec.add(106);  
vec.add(42);  
vec.add(-3);  
vec.add(27);
```

**vec.remove(1);**

# Removing Elements

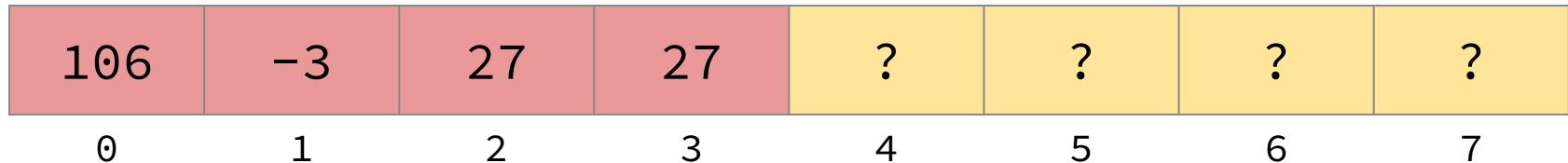


```
// client code
```

```
OurVector vec;  
vec.add(106);  
vec.add(42);  
vec.add(-3);  
vec.add(27);
```

```
vec.remove(1);
```

# Removing Elements

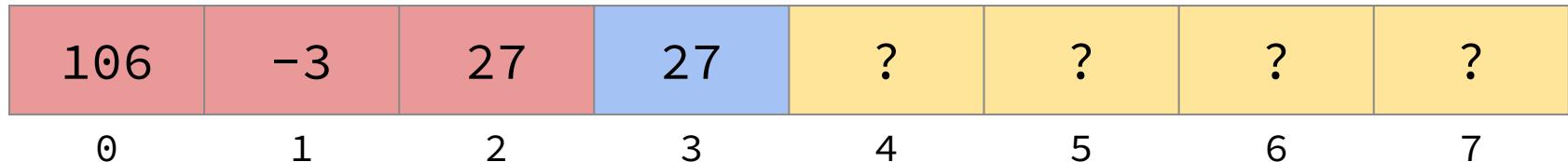


// client code

```
OurVector vec;  
vec.add(106);  
vec.add(42);  
vec.add(-3);  
vec.add(27);
```

**vec.remove(1);**

# Removing Elements

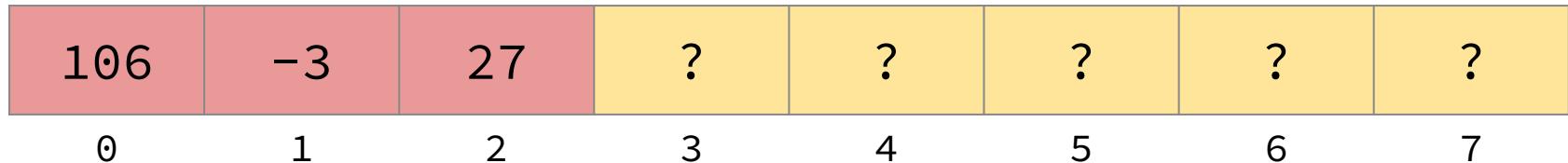


```
// client code
```

```
OurVector vec;  
vec.add(106);  
vec.add(42);  
vec.add(-3);  
vec.add(27);
```

```
vec.remove(1);
```

# Removing Elements



// client code

```
OurVector vec;  
vec.add(106);  
vec.add(42);  
vec.add(-3);  
vec.add(27);
```

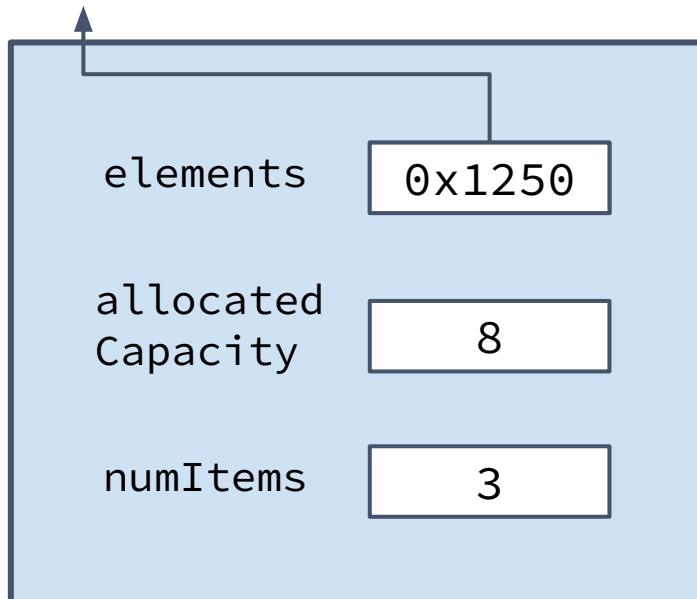
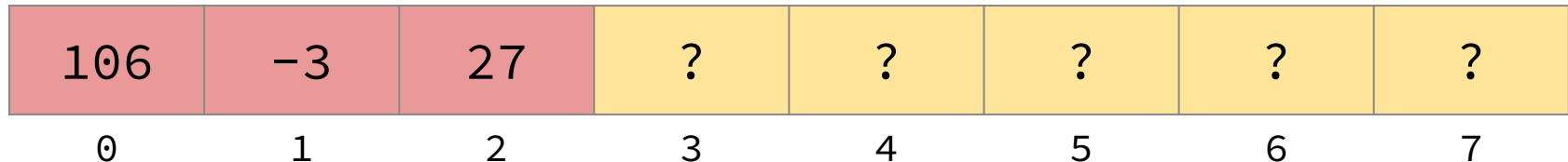
```
vec.remove(1);
```

# Inserting Elements

`insert()`: allows the client to specify which index they want the value to be inserted at

- similar to `add()`, but doesn't have to add to back

# Inserting Elements

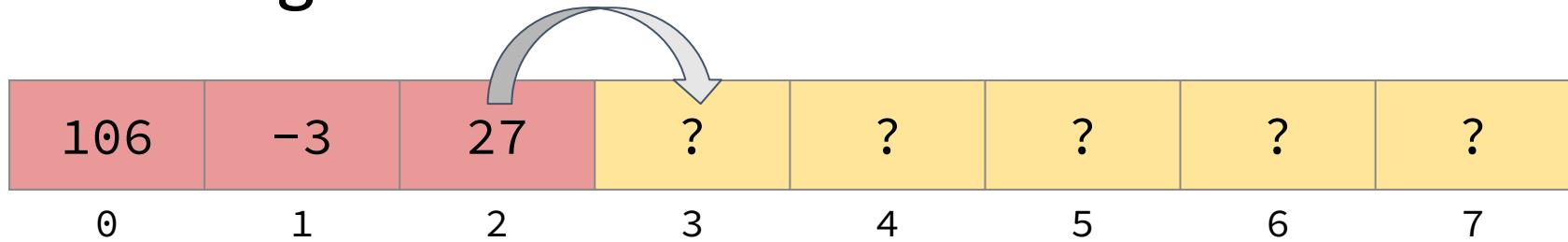


```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

# Inserting Elements

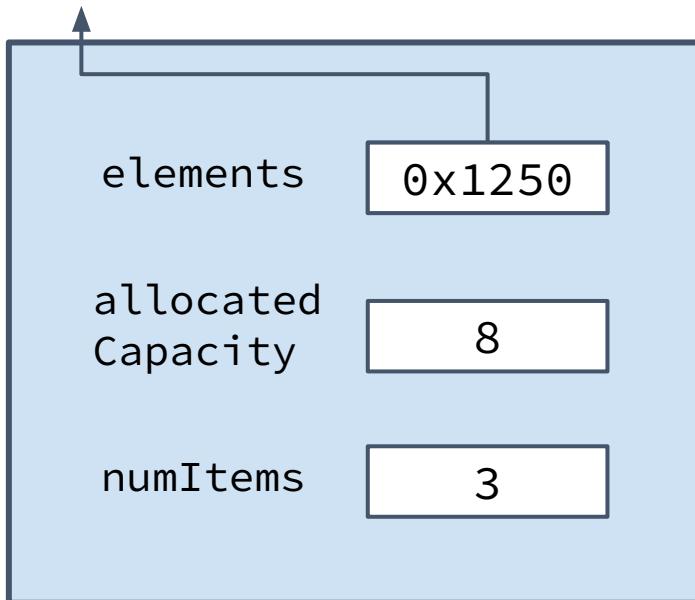
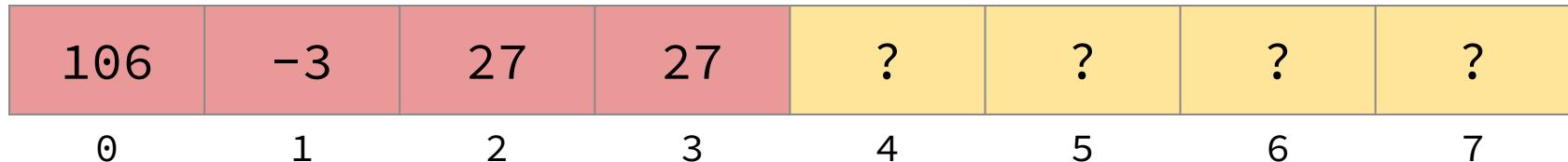


// client code

```
OurVector vec;  
vec.add(106);  
vec.add(42);  
vec.add(-3);  
vec.add(27);
```

```
vec.remove(1);  
vec.insert(0, 198);
```

# Inserting Elements

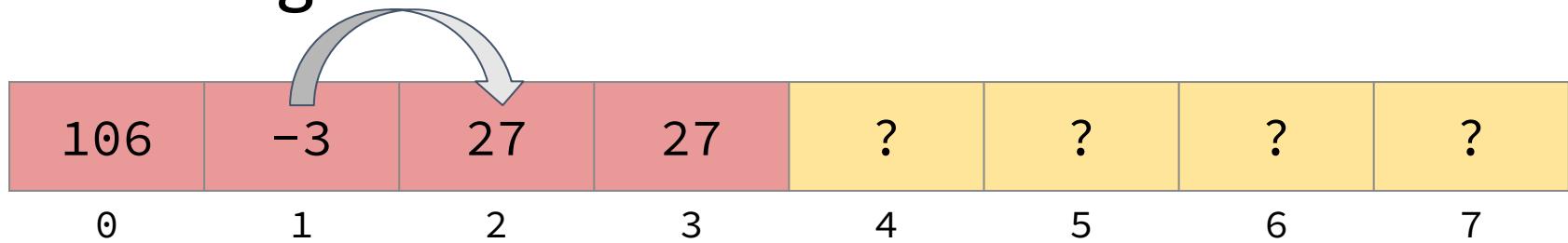


```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

# Inserting Elements

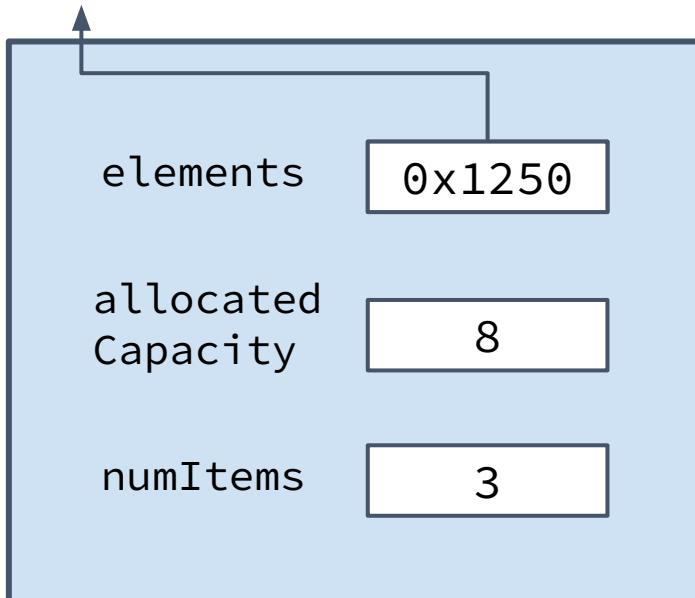
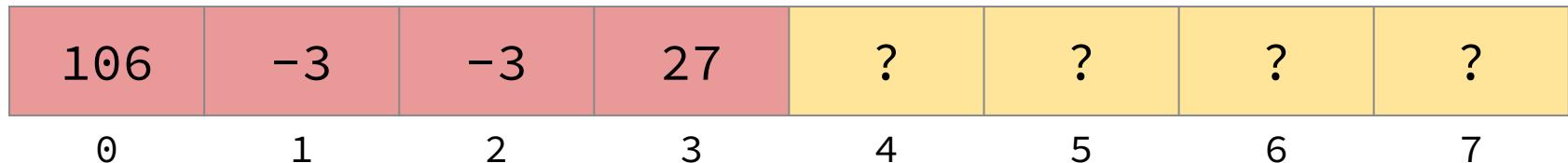


// client code

```
OurVector vec;  
vec.add(106);  
vec.add(42);  
vec.add(-3);  
vec.add(27);
```

```
vec.remove(1);  
vec.insert(0, 198);
```

# Inserting Elements

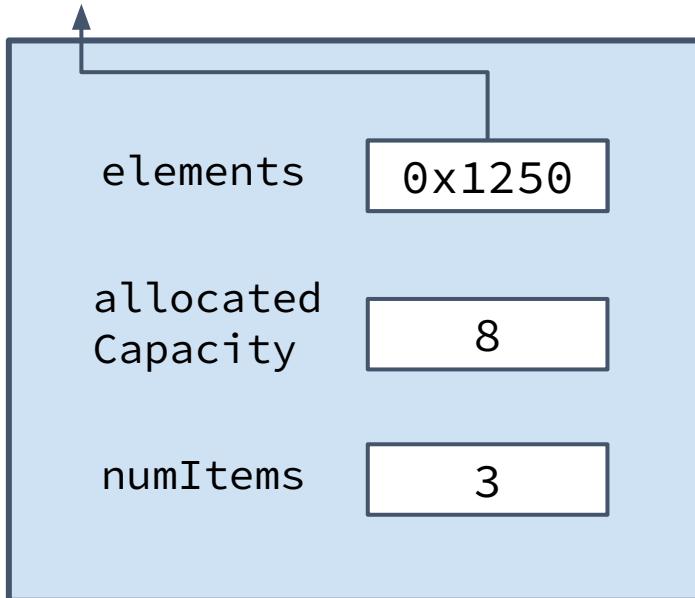
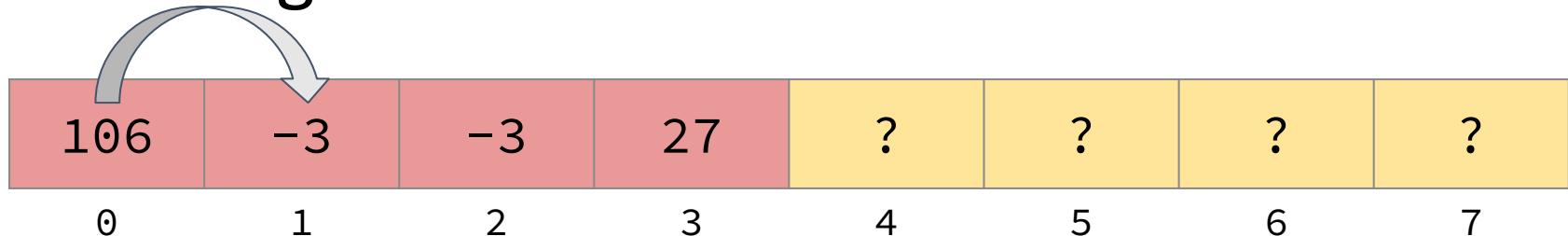


```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

# Inserting Elements

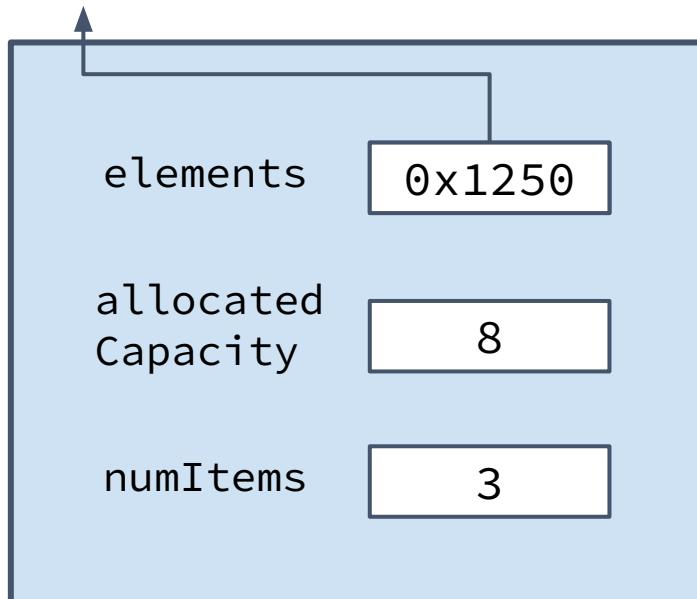
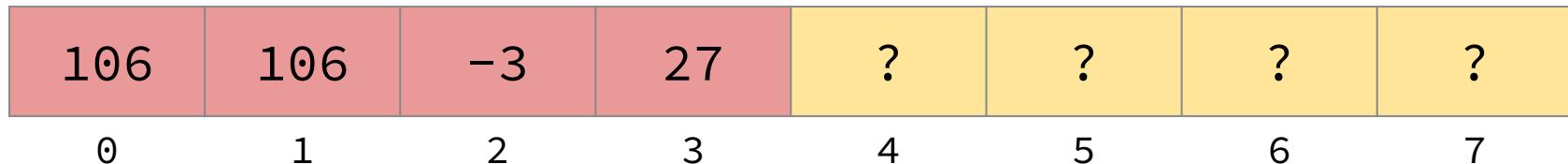


```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

# Inserting Elements

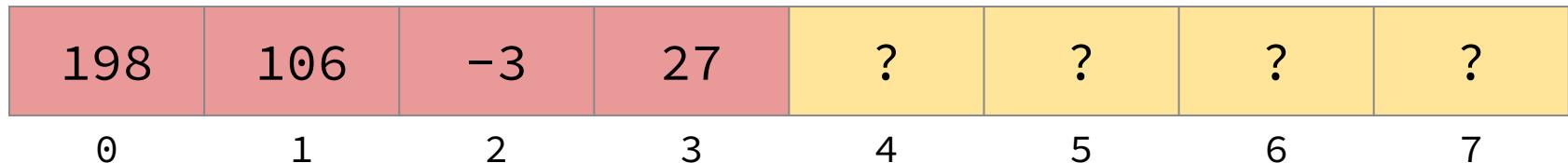


```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

# Inserting Elements



// client code

```
OurVector vec;  
vec.add(106);  
vec.add(42);  
vec.add(-3);  
vec.add(27);
```

```
vec.remove(1);  
vec.insert(0, 198);
```

# Remaining Operations

- `get()`: return the array element at a specified index
- `size()`: return the number of items the array currently holds
- `isEmpty()`: returns true if the number of items is 0

# Let's Code It Up!

Member functions

# Takeaways

- Using an array as a backing store of data involves shifting elements around – this kind of code is ripe for off-by-one errors!
- With good member variable member choices, most public methods are relatively straightforward to implements.
- We've now gained an appreciation for why insertion/removal on Vectors is an "expensive"  $O(n)$  operation.

# Running Out of Space

- Our current implementation very quickly runs out of space to store elements.
- What should we do when this happens?
  - Currently, we just throw an error.
  - Instead, we need a way to dynamically resize (grow) our internal data storage mechanism.

# Dynamic Array Growth

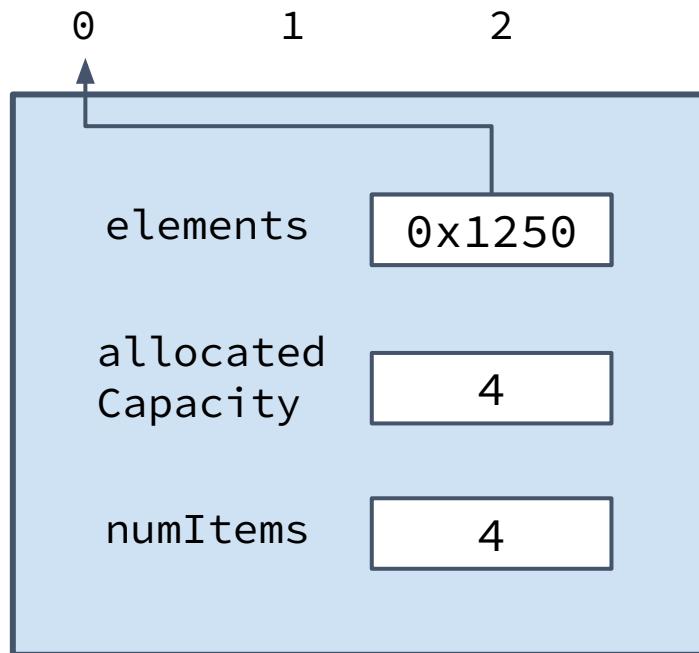
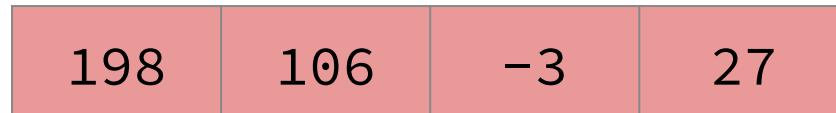


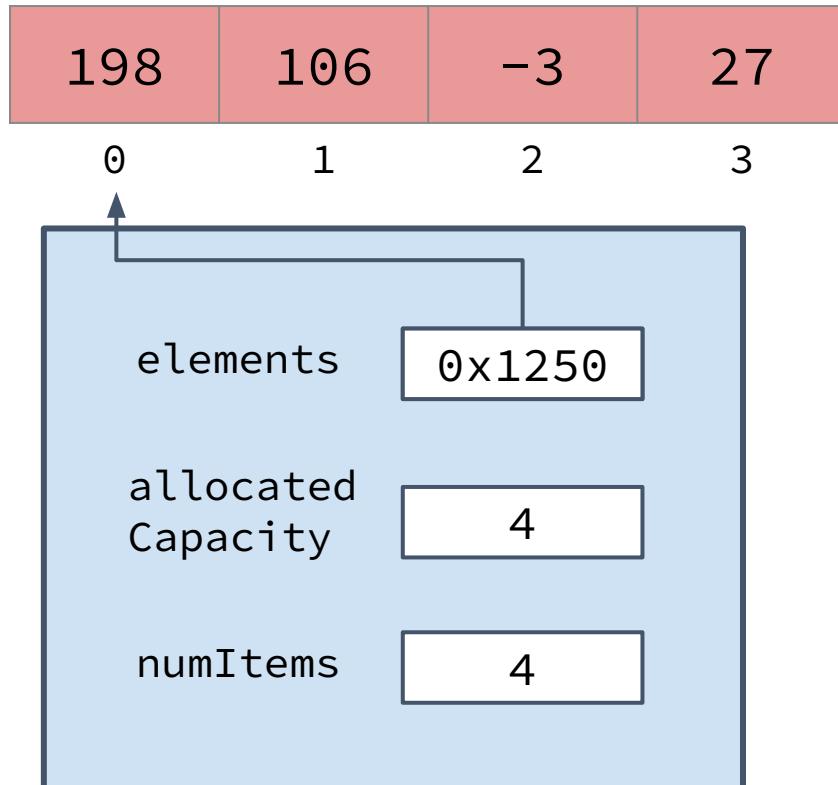
# Day in the Life of a Hermit Crab

- Hermit crabs live in scavenged shells that they find on the seafloor. Once in a shell, this is their lifestyle (with a bit of poetic license):
  - Grow until they have outgrown their current shell. Then, follow these 5 steps.
    - Find another, larger shell
    - Move all their stuff into the new shell
    - Leave the old shell on the seafloor
    - Update their address with the Hermit Crab Postal Service
    - Make note of their new shell's spacious capacity
- While this is purposefully a bit of a silly analogy, this process models almost exactly what we need to do in order to dynamically resize our internal data storage mechanism

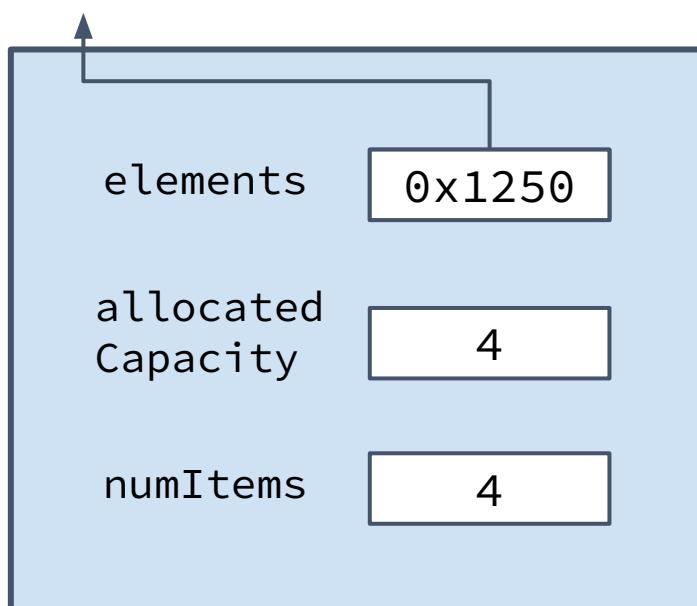
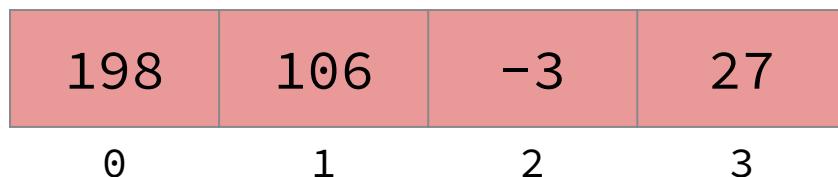
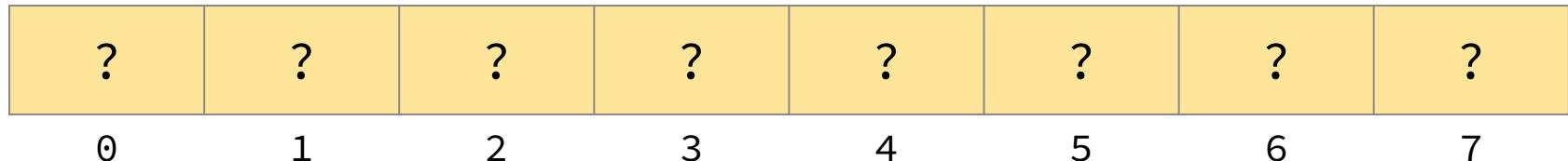
# Day in the Life of a Growable Array

- When we run out of space in our array, we want to allocate a new array that is bigger than our old array so we can store the new data and keep growing.
- These "growable arrays" follow a five-step expansion that mirrors the hermit crab model (with poetic license).
  - Grow the array until we run out of space (how can we tell if we've run out of space?)
    - Create a new, larger array (usually we choose to double the current size)
    - Copy the old array elements to the new array
    - Delete (free) the old array
    - Point the old array variable to the new array
    - Update the associated capacity variable for the array

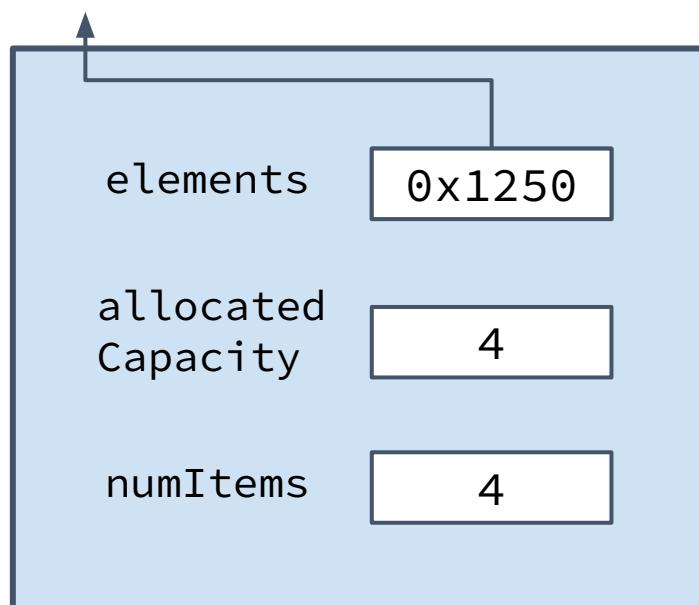
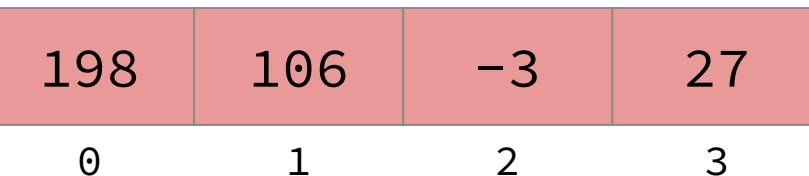
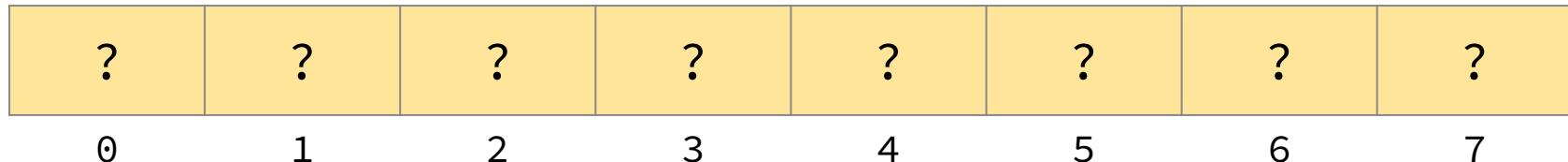




1. Create a new, larger array (usually we choose to double the current size)



1. Create a new, larger array (usually we choose to double the current size)



1. Create a new, larger array (usually we choose to double the current size)
2. Copy the old array elements to the new array

198	106	-3	27	?	?	?	?
-----	-----	----	----	---	---	---	---

0

1

2

3

4

5

6

7

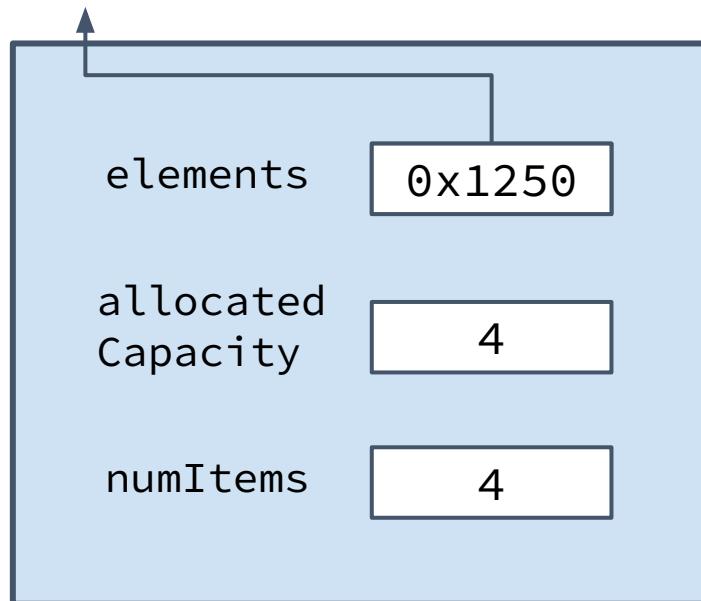
198	106	-3	27
-----	-----	----	----

0

1

2

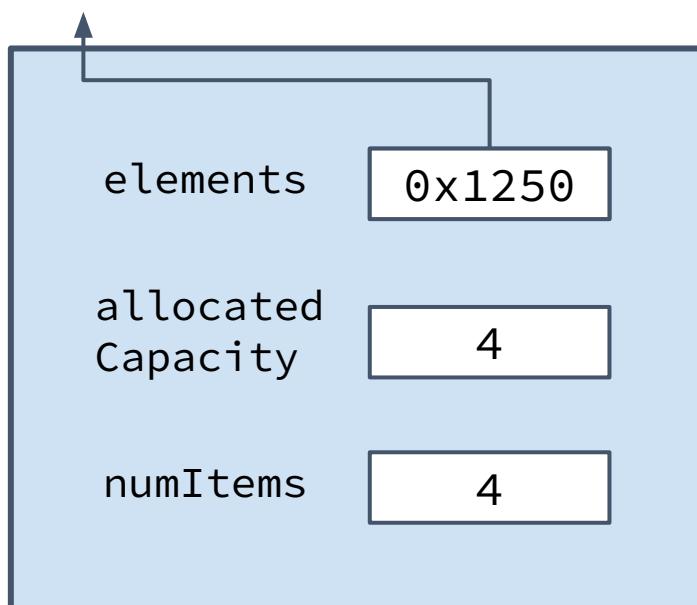
3



1. Create a new, larger array (usually we choose to double the current size)
2. Copy the old array elements to the new array

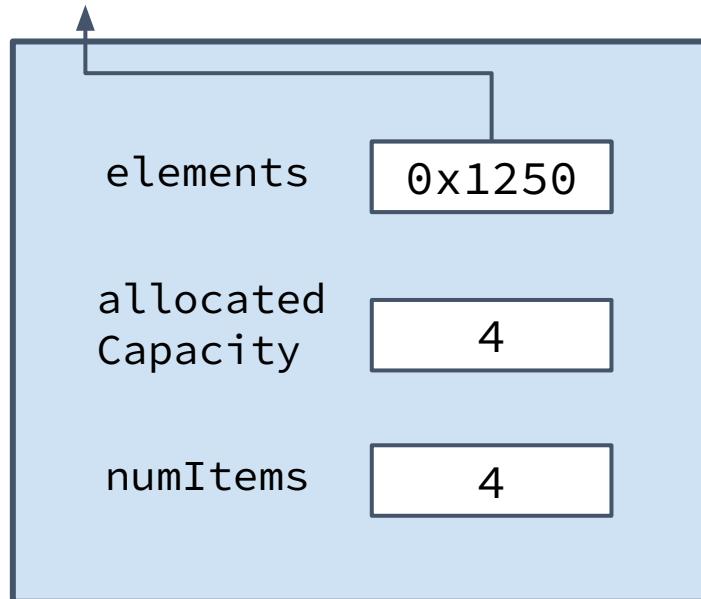
198	106	-3	27	?	?	?	?
0	1	2	3	4	5	6	7

198	106	-3	27
0	1	2	3



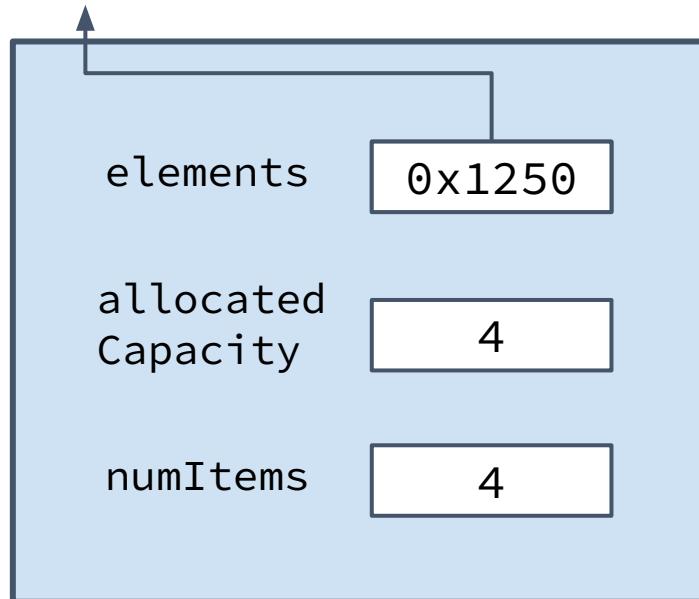
1. Create a new, larger array (usually we choose to double the current size)
2. Copy the old array elements to the new array
3. Delete (free) the old array

198	106	-3	27	?	?	?	?
0	1	2	3	4	5	6	7



1. Create a new, larger array (usually we choose to double the current size)
2. Copy the old array elements to the new array
3. Delete (free) the old array

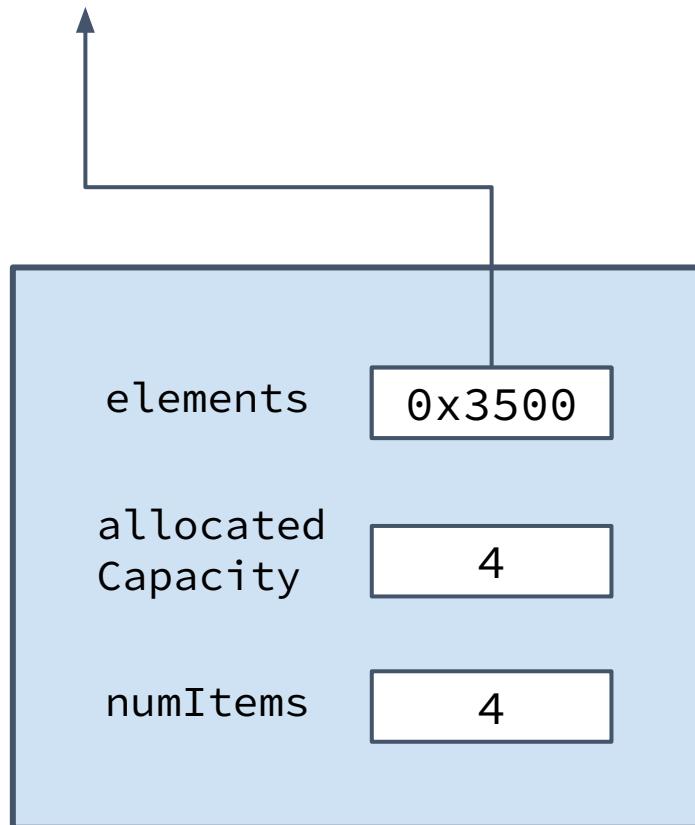
198	106	-3	27	?	?	?	?
0	1	2	3	4	5	6	7



1. Create a new, larger array (usually we choose to double the current size)
2. Copy the old array elements to the new array
3. Delete (free) the old array
4. Point the old array variable to the new array

198	106	-3	27	?	?	?	?
-----	-----	----	----	---	---	---	---

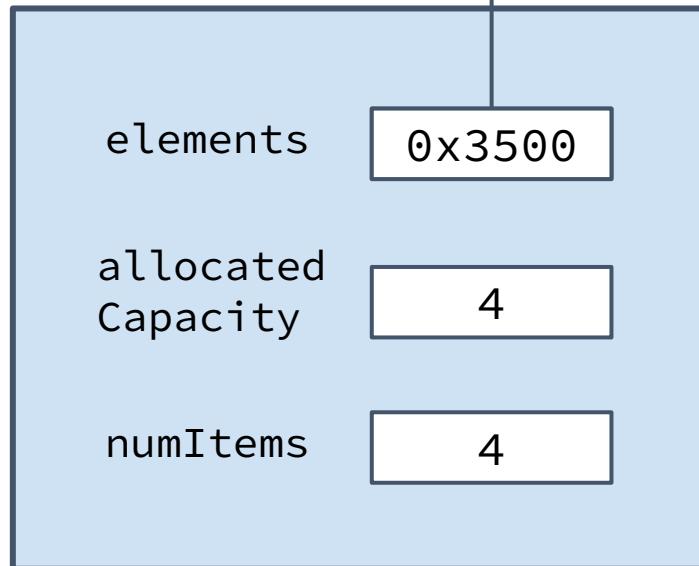
0 1 2 3 4 5 6 7



1. Create a new, larger array (usually we choose to double the current size)
2. Copy the old array elements to the new array
3. Delete (free) the old array
4. Point the old array variable to the new array

198	106	-3	27	?	?	?	?
-----	-----	----	----	---	---	---	---

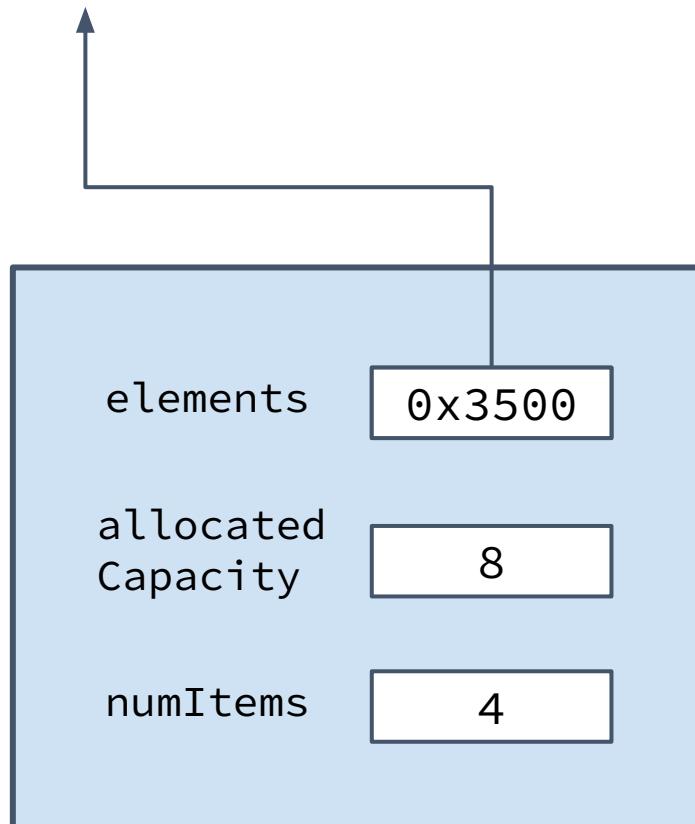
0 1 2 3 4 5 6 7



1. Create a new, larger array (usually we choose to double the current size)
2. Copy the old array elements to the new array
3. Delete (free) the old array
4. Point the old array variable to the new array
5. Update the associated capacity variable for the array

198	106	-3	27	?	?	?	?
-----	-----	----	----	---	---	---	---

0 1 2 3 4 5 6 7



1. Create a new, larger array (usually we choose to double the current size)
2. Copy the old array elements to the new array
3. Delete (free) the old array
4. Point the old array variable to the new array
5. Update the associated capacity variable for the array

# Let's Code It Up!

Dynamic Array Growth

# Summary

- The first step of implementing an ADT class (as with any class) is answering the three important questions regarding its member variables, member functions, and initialization procedures.
- Most ADT classes will need to store their data in an underlying array. The organizational patterns of data in that array may vary, so it is important to illustrate and visualize the contents and any operations that may be done.
- The paradigm of "growable" arrays allows for fast and flexible containers with dynamic resizing capabilities that enable storage of large amounts of data.