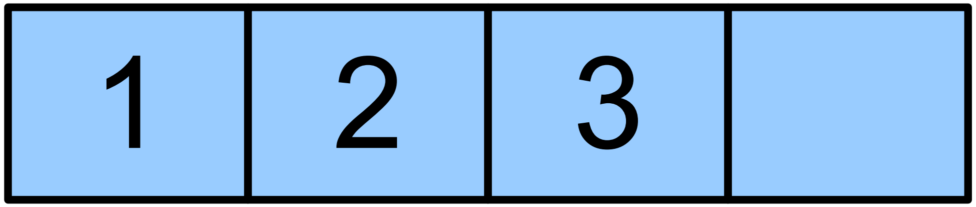
Linked Lists

Part One

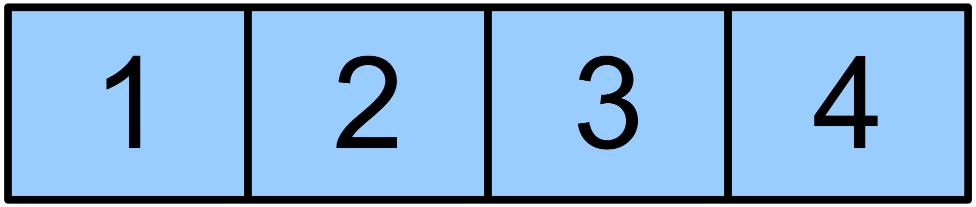
Array-Based Allocation

- Our current implementation of **`Stack`** uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a new array and move everything over.



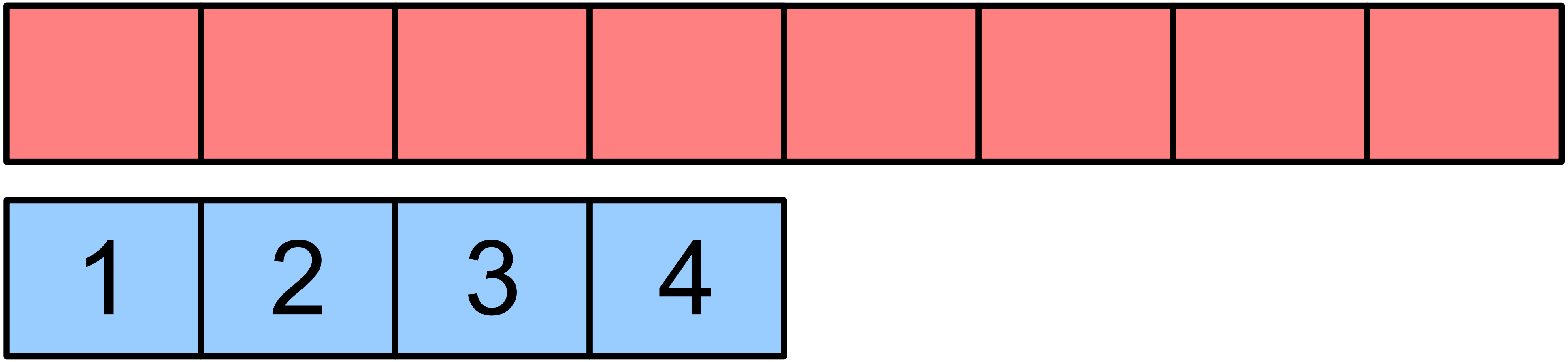
Array-Based Allocation

- Our current implementation of **`Stack`** uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a new array and move everything over.



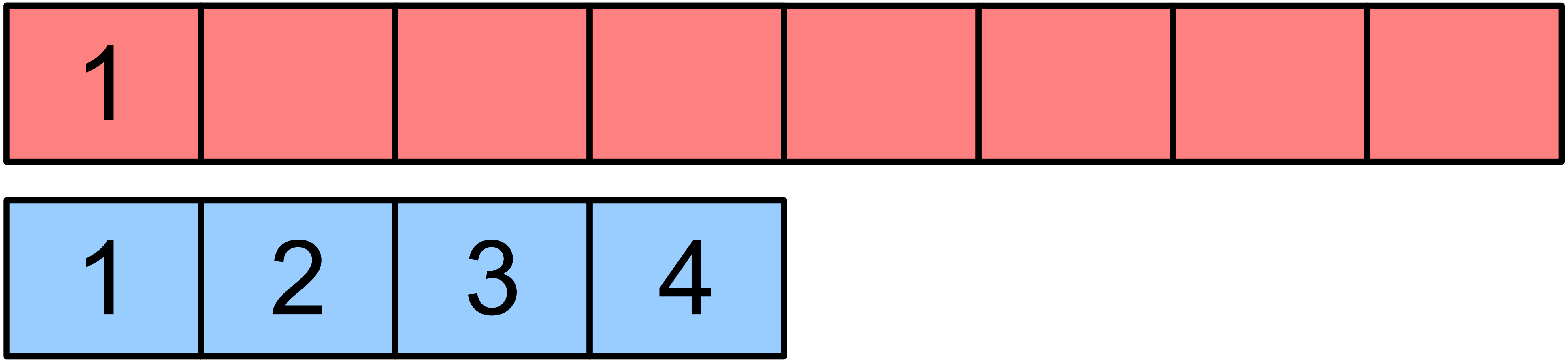
Array-Based Allocation

- Our current implementation of **`Stack`** uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a new array and move everything over.



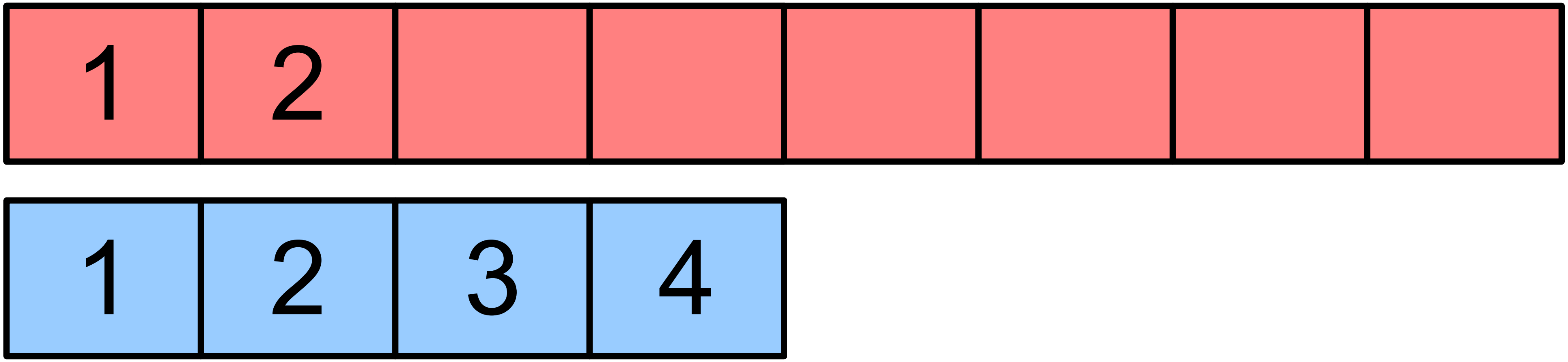
Array-Based Allocation

- Our current implementation of **`Stack`** uses dynamically-allocated arrays.
- To append an element:
    - If there is free space, put the element into that space.
    - Otherwise, get a new array and move everything over.



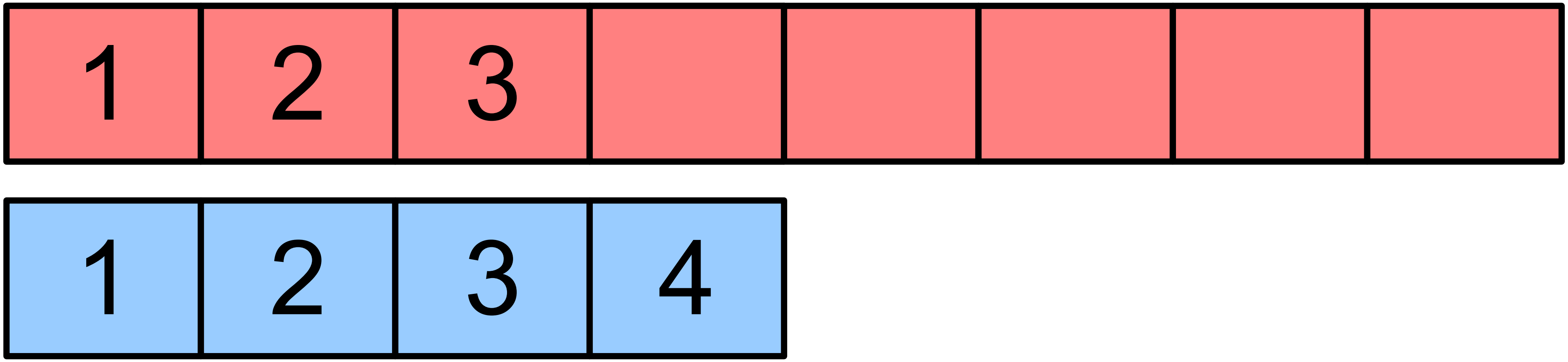
Array-Based Allocation

- Our current implementation of **`Stack`** uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a new array and move everything over.



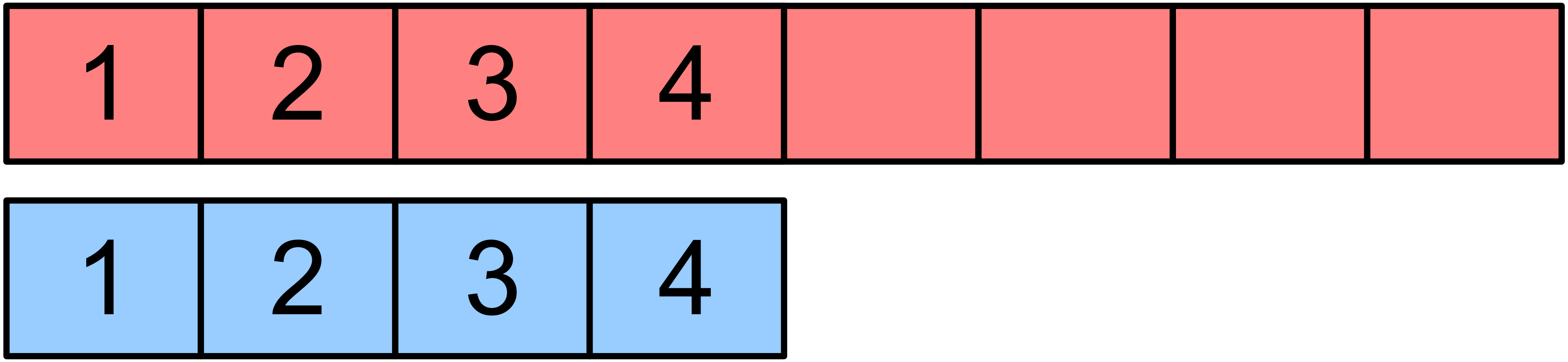
Array-Based Allocation

- Our current implementation of **`Stack`** uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a new array and move everything over.



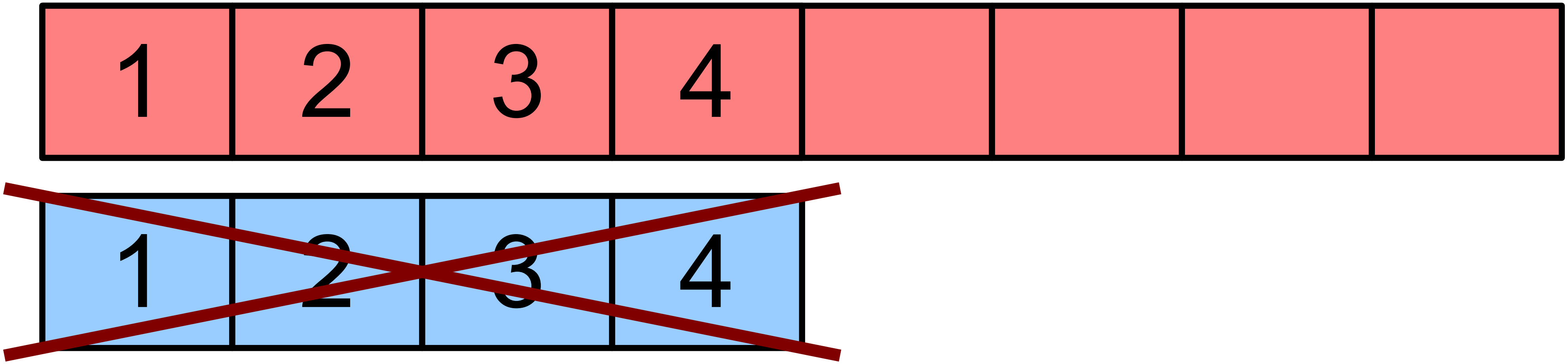
Array-Based Allocation

- Our current implementation of **`Stack`** uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a new array and move everything over.



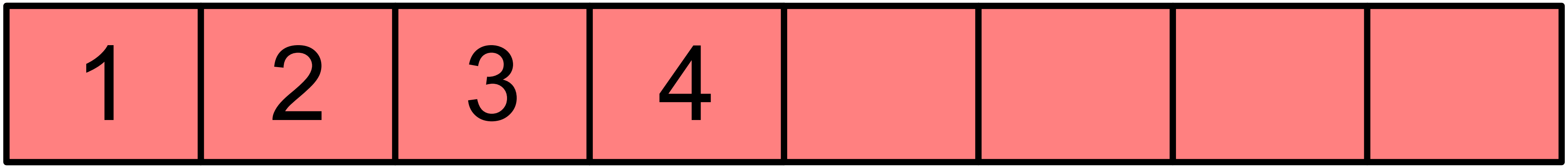
Array-Based Allocation

- Our current implementation of **`Stack`** uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a new array and move everything over.



Array-Based Allocation

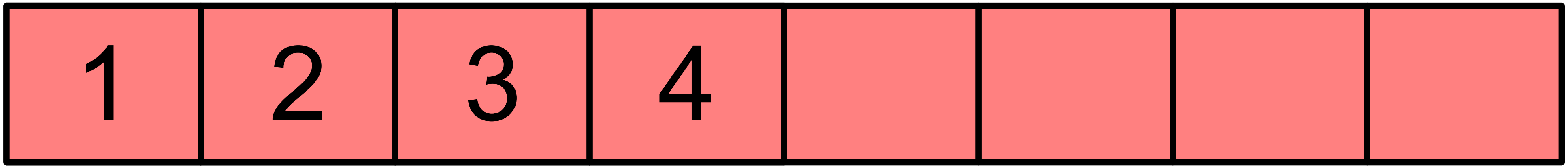
- Our current implementation of **`Stack`** uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a new array and move everything over.



1	2	3	4				

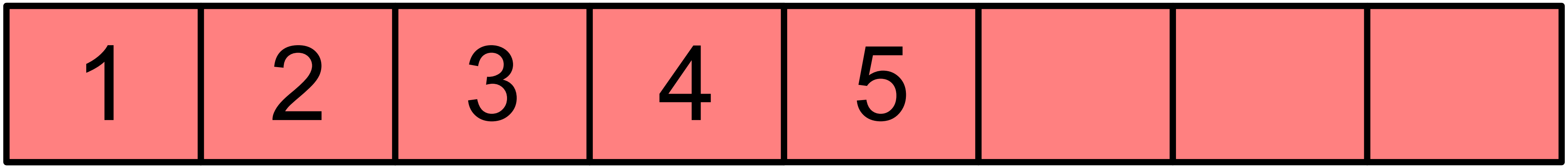
Array-Based Allocation

- Our current implementation of **`Stack`** uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a new array and move everything over.



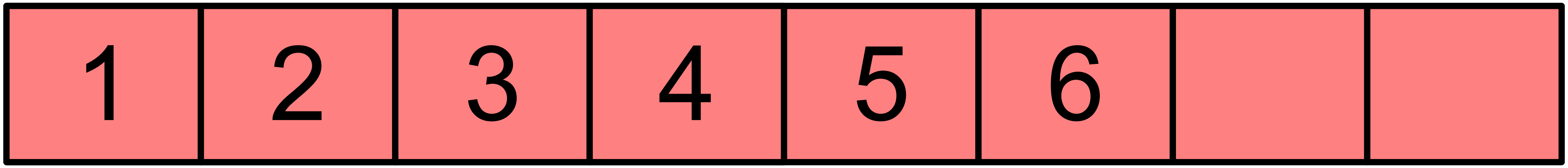
Array-Based Allocation

- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a new array and move everything over.

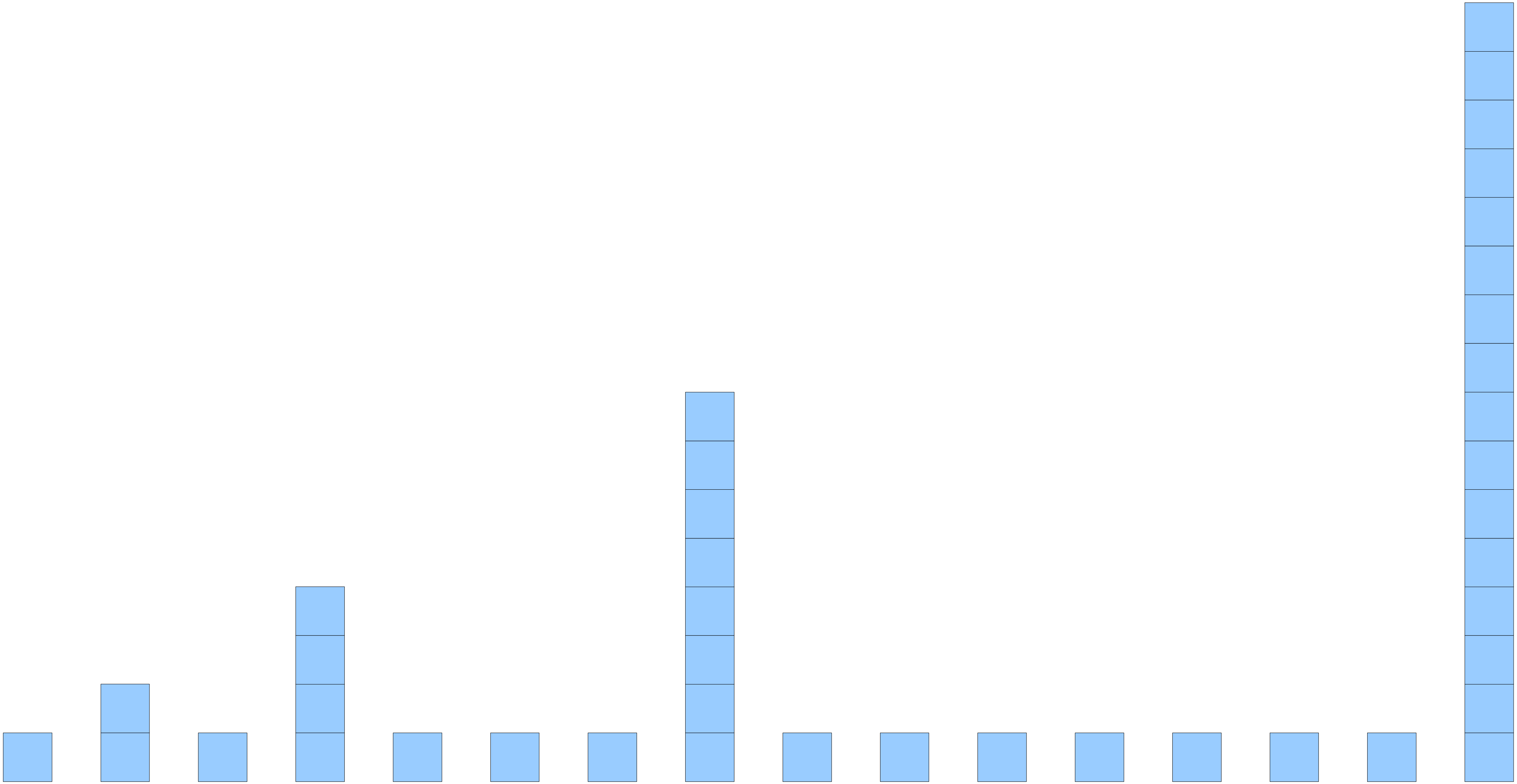


Array-Based Allocation

- Our current implementation of **`Stack`** uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a new array and move everything over.



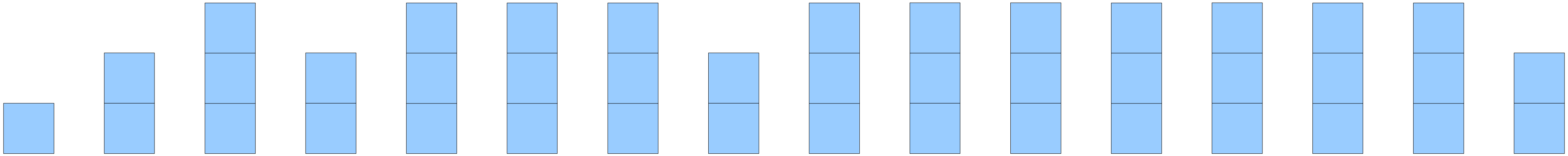
Spreading the Work



Spreading the Work

On average, we do just 3 units of work!

This is O(1) work on average!



`Vector` is implemented in a similar manner.

Data Structures

- Last Lecture: **`Stack`**, **`Vector`**
- Today: **`Queue`**

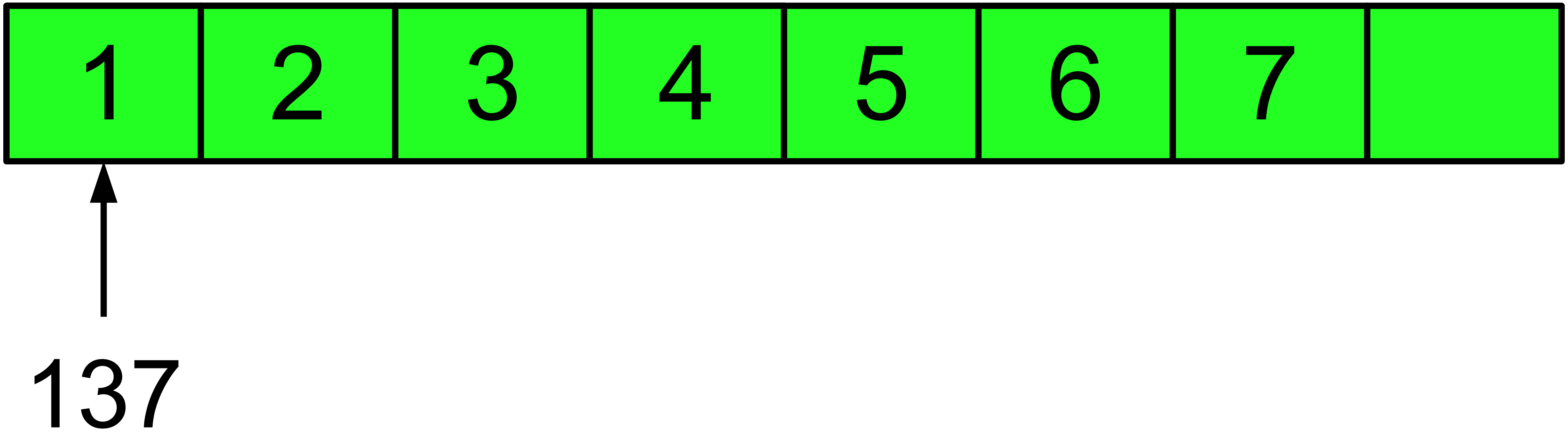
Array Based Queue?

- Instead of reallocating a huge array to get the space we need, why not just get a tiny amount of extra space for the next element?
- Taking notes – when you run out of space on a page, you just get a new page. You don't copy your entire set of notes onto a longer sheet of paper!

Excuse Me, Coming Through...

1	2	3	4	5	6	7	

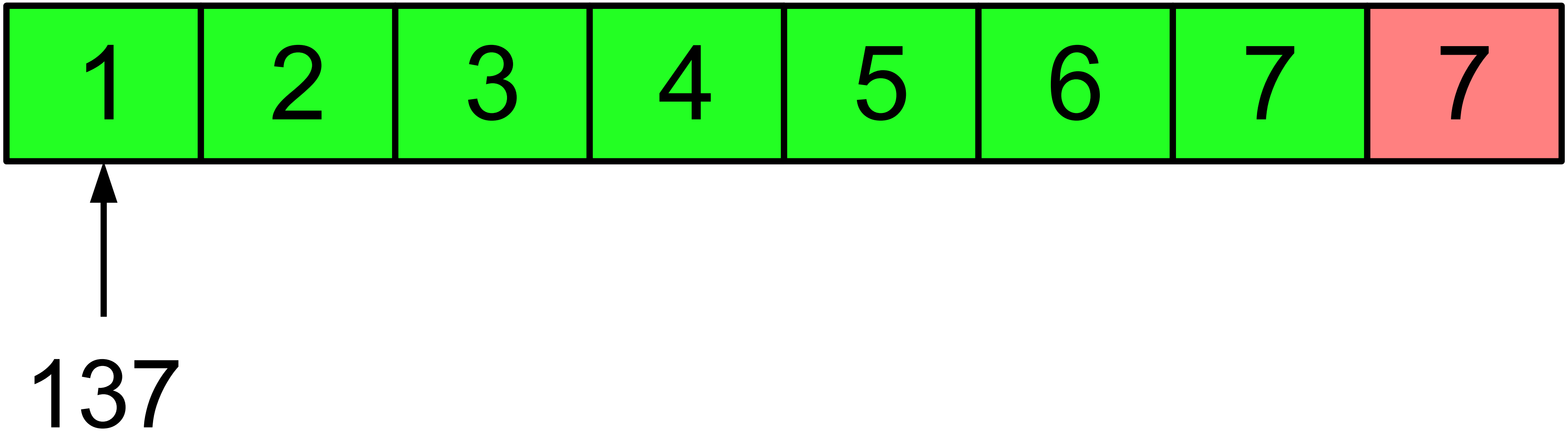
Excuse Me, Coming Through...



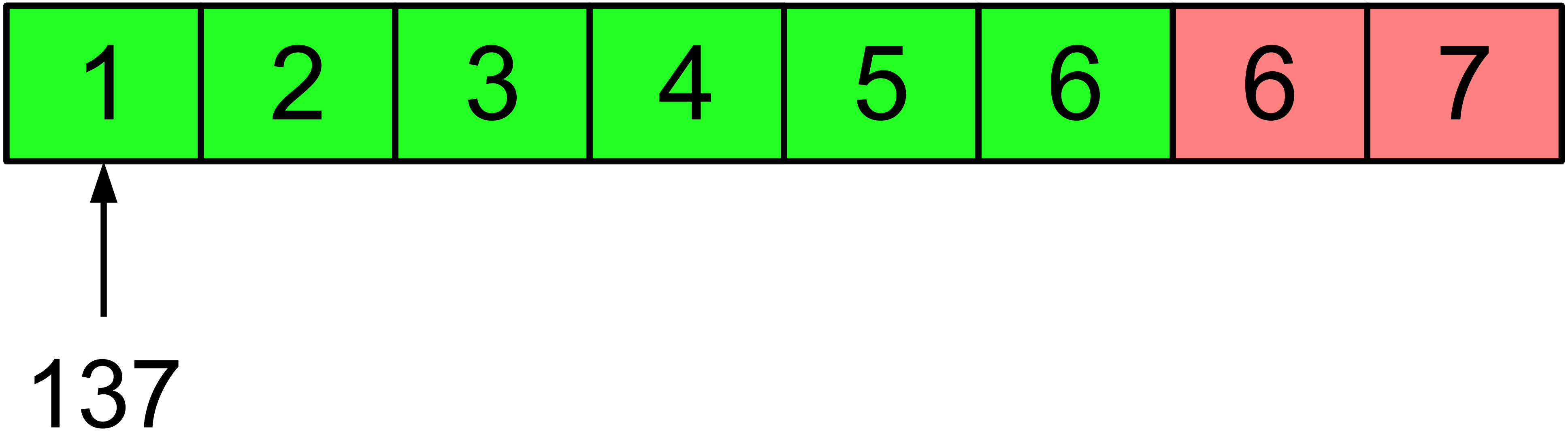
1 2 3 4 5 6 7

137

Excuse Me, Coming Through...



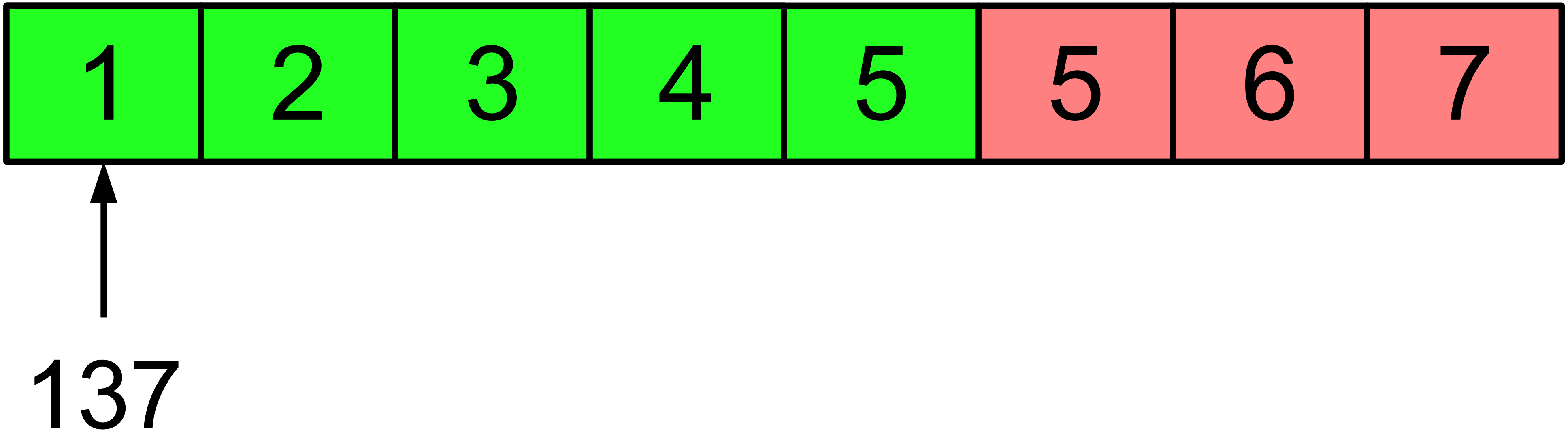
Excuse Me, Coming Through...



1 2 3 4 5 6 6 7

137

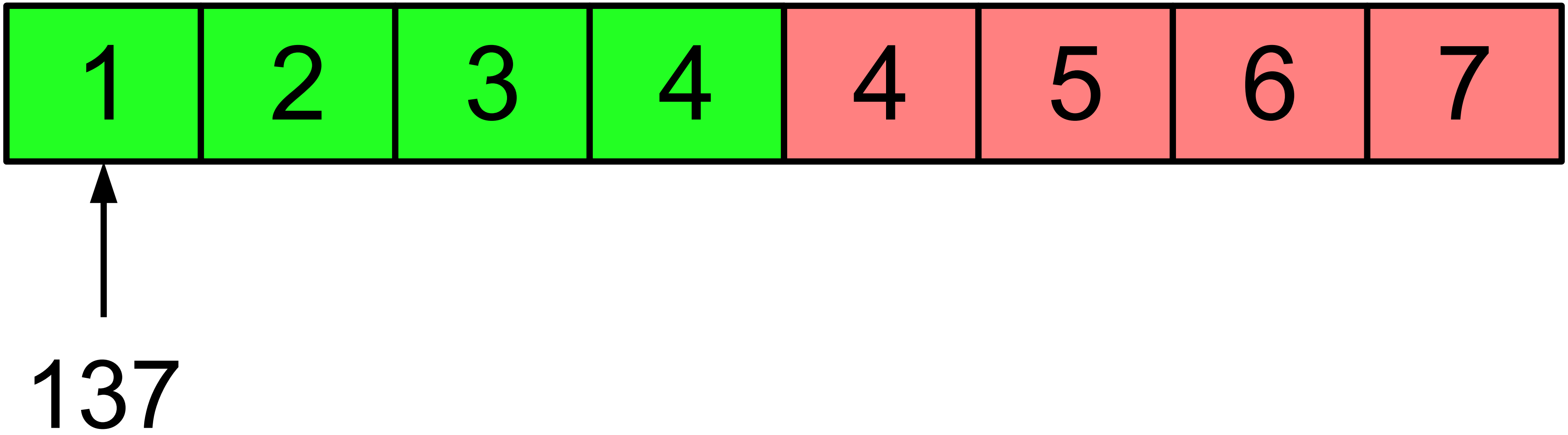
Excuse Me, Coming Through...



1	2	3	4	5	5	6	7

137

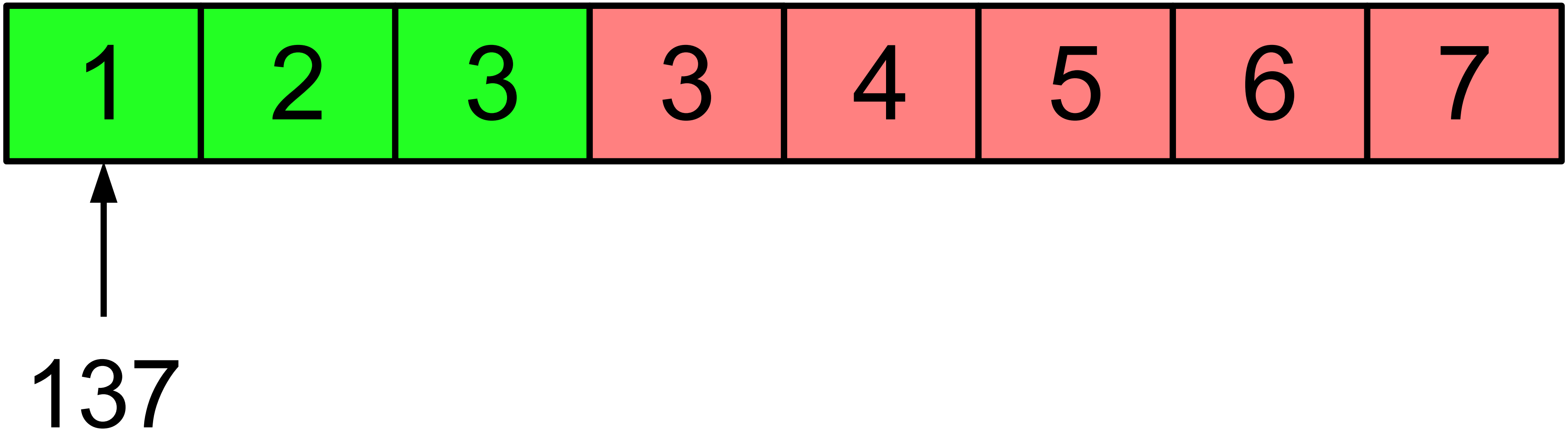
Excuse Me, Coming Through…



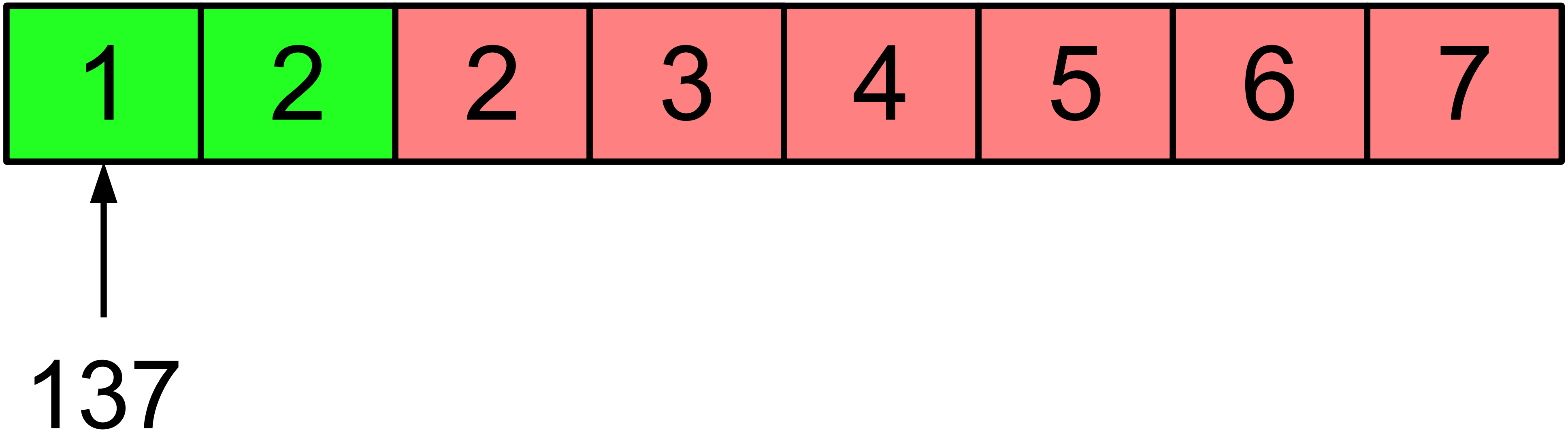
1	2	3	4	4	5	6	7

137

Excuse Me, Coming Through...

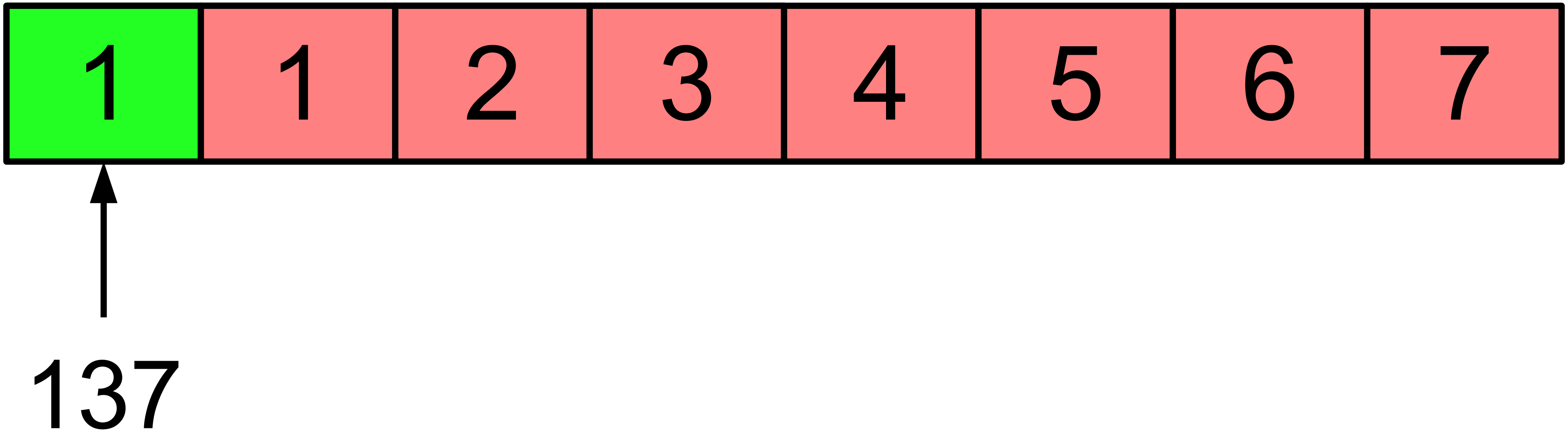


Excuse Me, Coming Through...

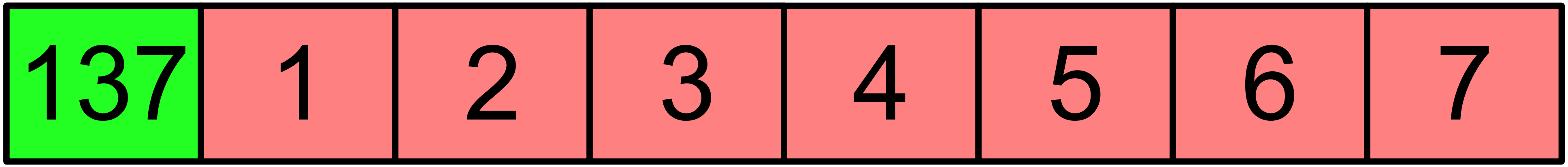


1	2	2	3	4	5	6	7

Excuse Me, Coming Through...



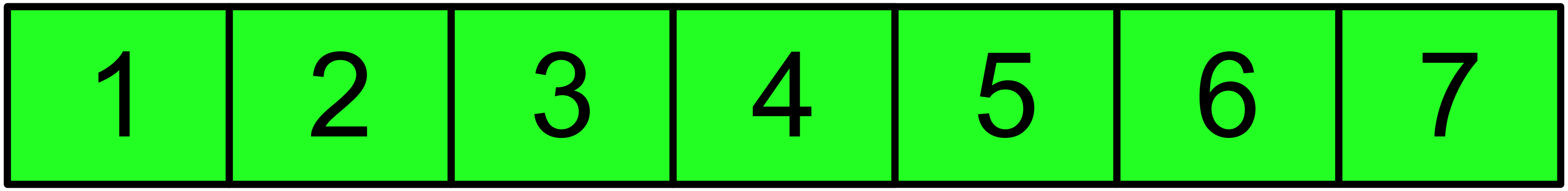
Excuse Me, Coming Through...



137	1	2	3	4	5	6	7

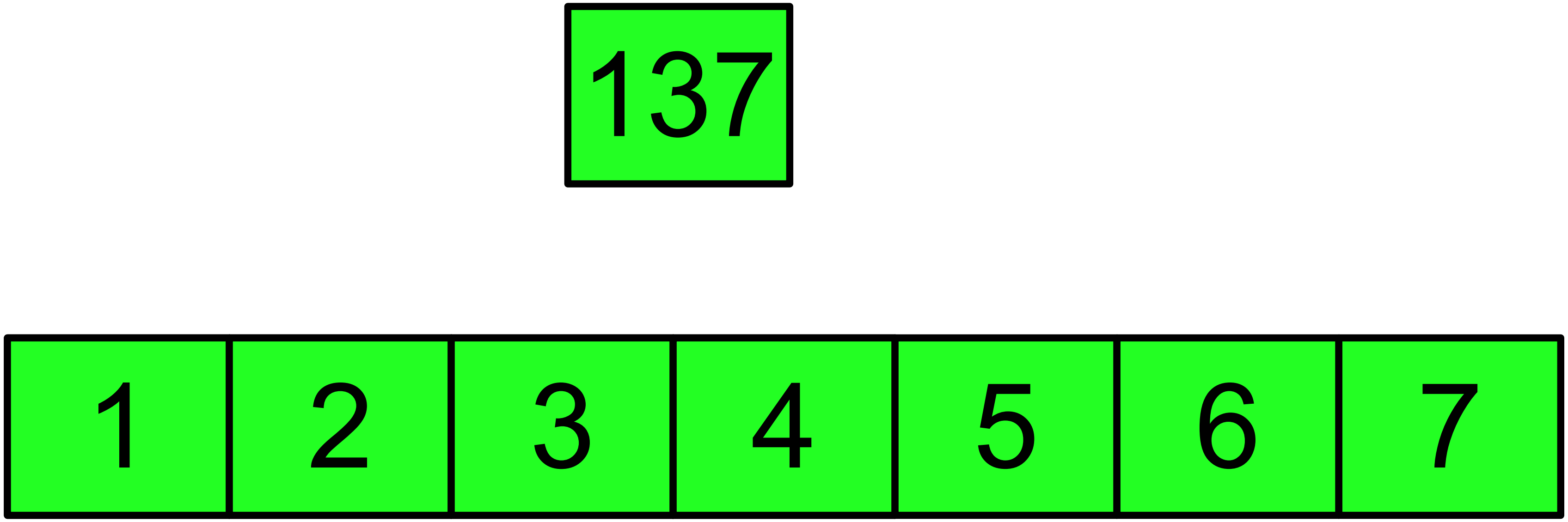
Shoving Things Over

- Right now, inserting an element into a middle of a **`Vector`** can be very costly.
- Couldn't we just do something like this?



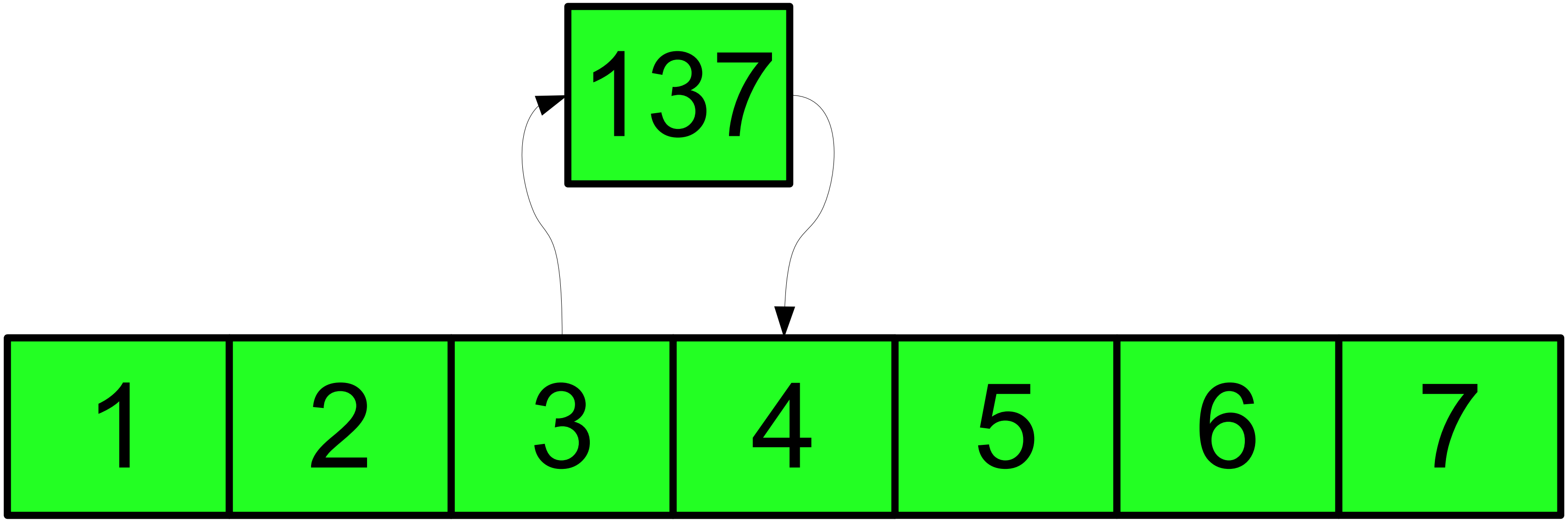
Shoving Things Over

- Right now, inserting an element into a middle of a **`Vector`** can be very costly.
- Couldn't we just do something like this?



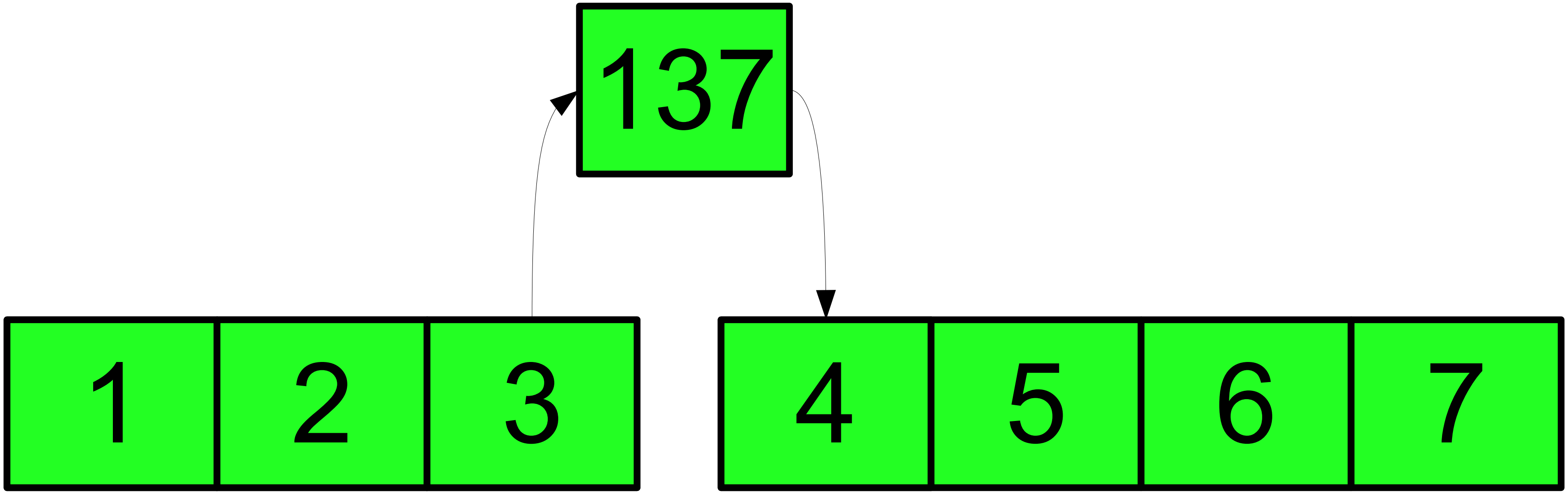
Shoving Things Over

- Right now, inserting an element into a middle of a **`Vector`** can be very costly.
- Couldn't we just do something like this?



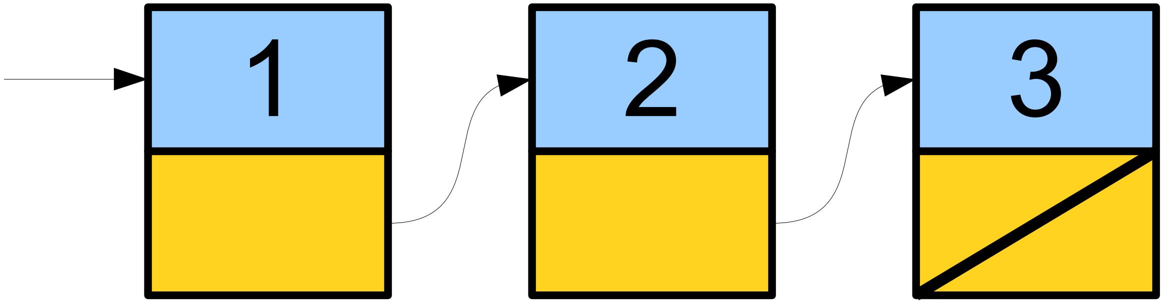
Shoving Things Over

- Right now, inserting an element into a middle of a **`Vector`** can be very costly.
- Couldn't we just do something like this?



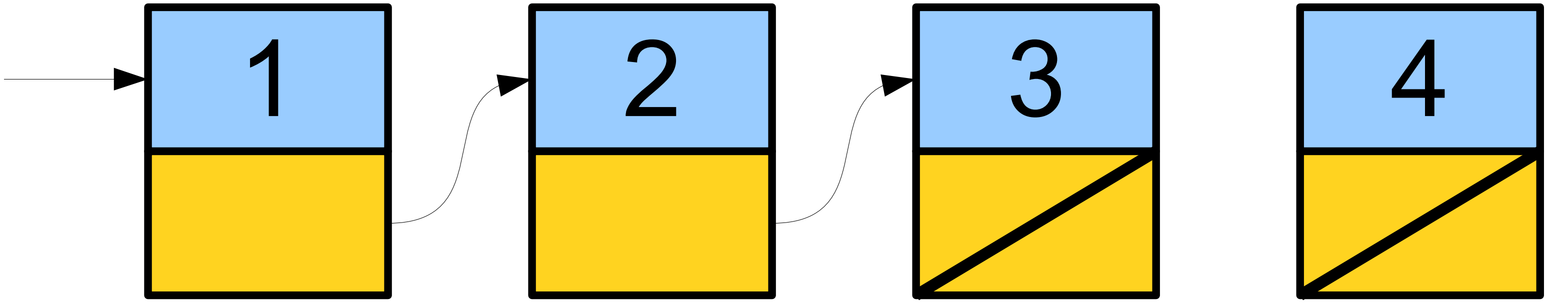
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



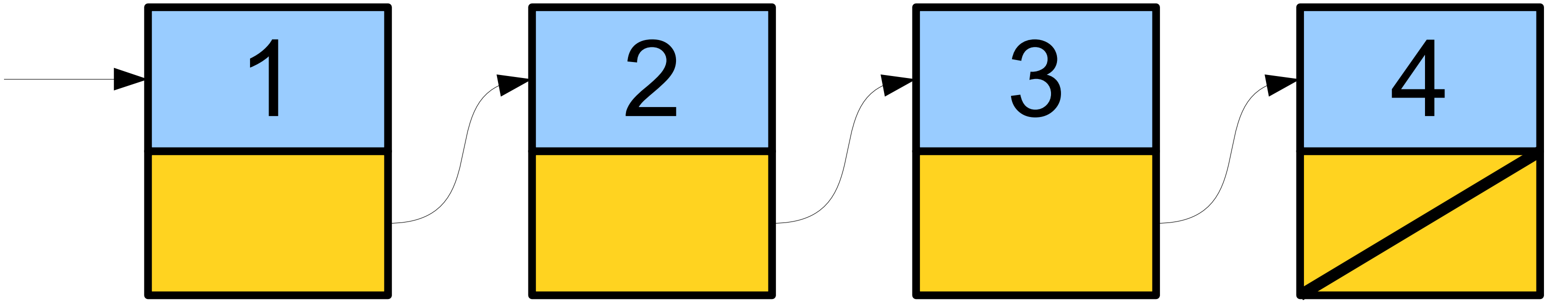
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



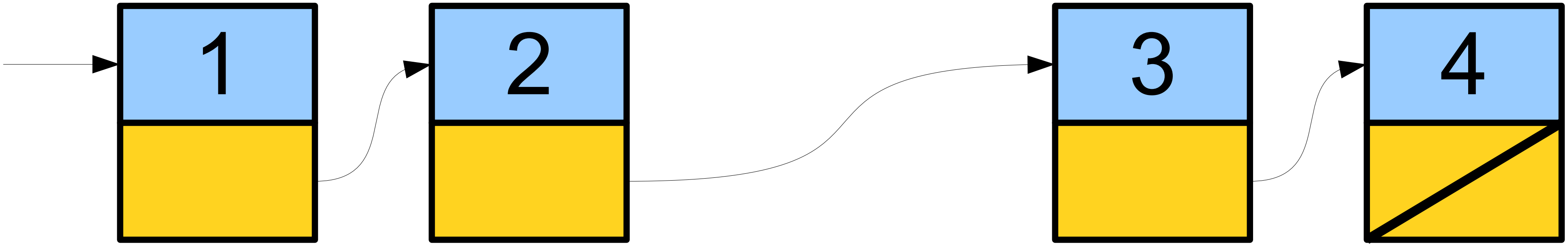
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



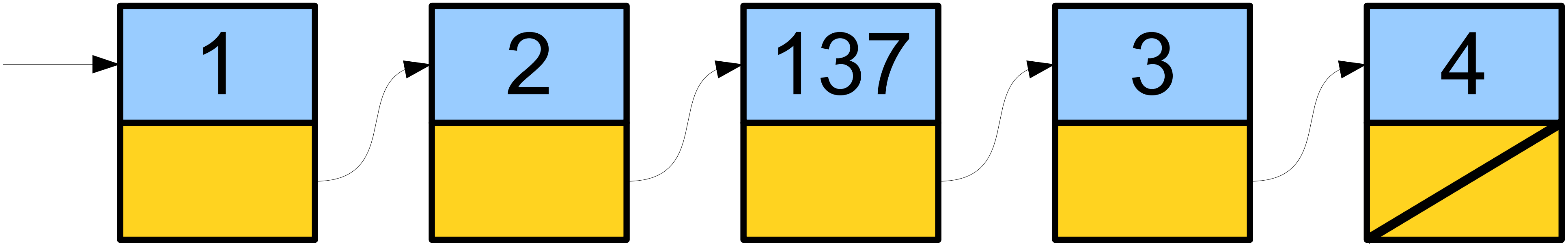
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



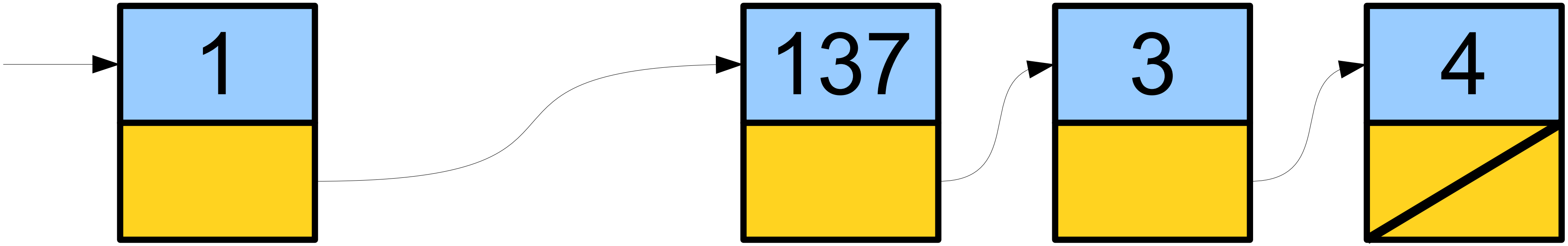
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



Linked Lists at a Glance

- Can efficiently splice new elements into the list or remove existing elements anywhere in the list.
- Never have to do a massive copy step; insertion is efficient in the worst-case.
- Has some tradeoffs; we'll see this later.

Building our Vocabulary

- In order to use linked lists, we will need to introduce or revisit several new language features:
  - Structures
  - Dynamic allocation
  - Null pointers

Building our Vocabulary

In order to use linked lists, we will need to introduce or revisit several new language features:

- Structures

Dynamic allocation

Null pointers

Structures

- In C++, a **structure** is a type consisting of several individual variables all bundled together.
- To create a structure, we must
  - Define what fields are in the structure, then
  - Create a variable of the appropriate type.
- Similar to using classes – need to define and implement the class before we can use it.

Defining Structures

- You can define a structure by using the `struct` keyword:

```
struct TypeName {
   /* … field declarations … */
};
```

- For those of you with a C background: in C++, “`typedef struct`” is not necessary.

A Simple Structure

```
struct Tribute {
    string name;
    int districtNumber;
};
```

A Simple Structure

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute t;
```

A Simple Structure

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute t;
t.name = "Katniss Everdeen";
t.districtNumber = 1;
```

A Simple Structure

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute t;
t.name = "Katniss Everdeen";
t.districtNumber = 12;
```

`struct`s and **`class`**es

- In C++, a **`class`** is a pair of an interface and an implementation.
  - Interface controls how the class is to be used.
  - Implementation specifies how it works.
- A **`struct`** is *usually* a stripped-down version of a **`class`**:
  - Purely implementation, no interface.
  - Primarily used to bundle information together when no interface is needed.

Building our Vocabulary

- In order to use linked lists, we will need to introduce or revisit several new language features:
    - Structures
    - Dynamic allocation
    - Null pointers

Building our Vocabulary

In order to use linked lists, we will need to introduce or revisit several new language features:

Structures

- Dynamic allocation

Null pointers

Dynamic Memory Allocation

- We have seen the `new` keyword used to allocate arrays, but it can also be used to allocate single objects.
- The syntax

  ```
  new T(args)
  ```

  creates a new object of type ***T*** passing the appropriate arguments to the constructor, then returns a pointer to it.

Dynamic Memory Allocation

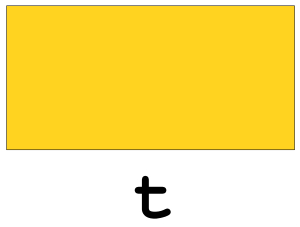
```
struct Tribute {
    string name;
    int districtNumber;
};
```

Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```

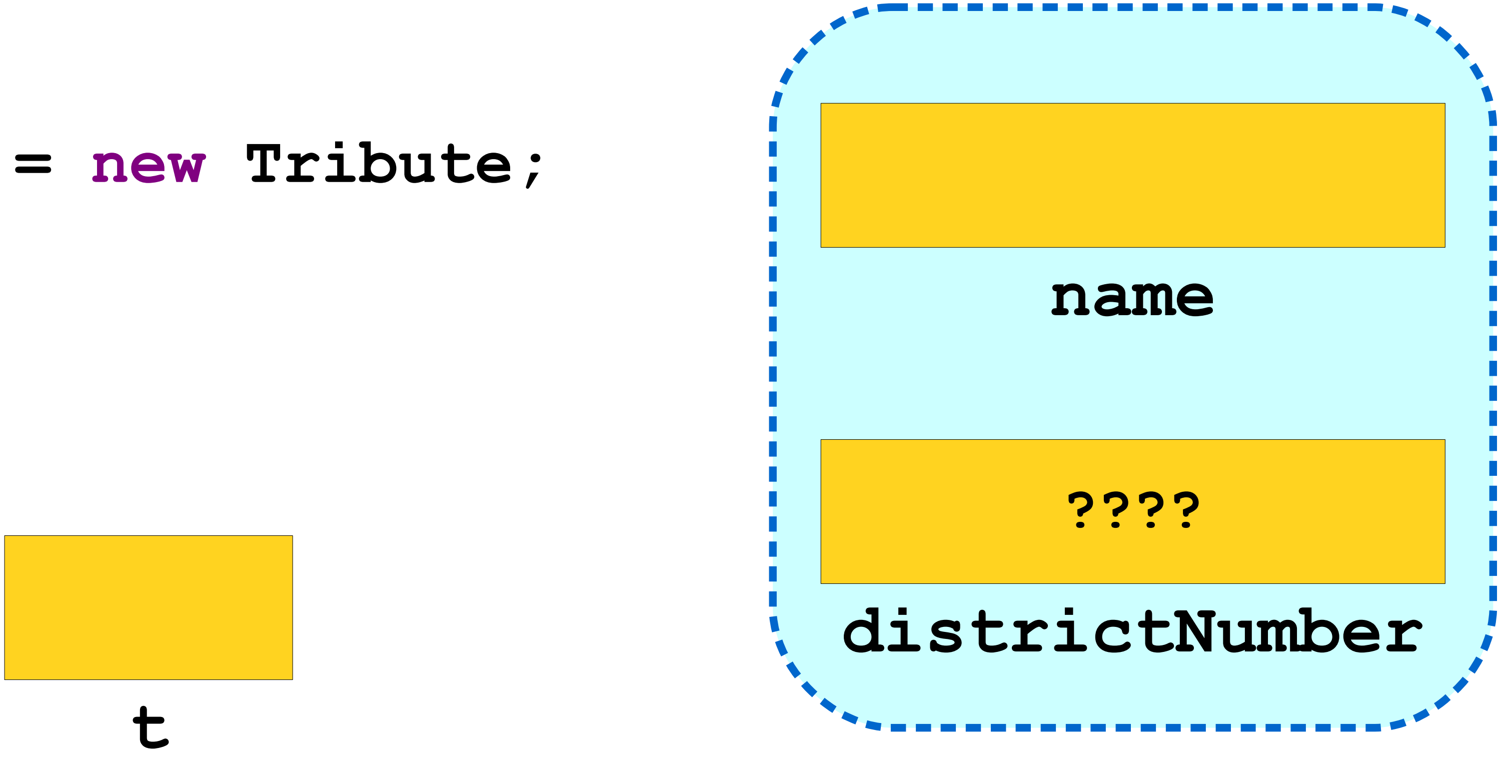
Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```



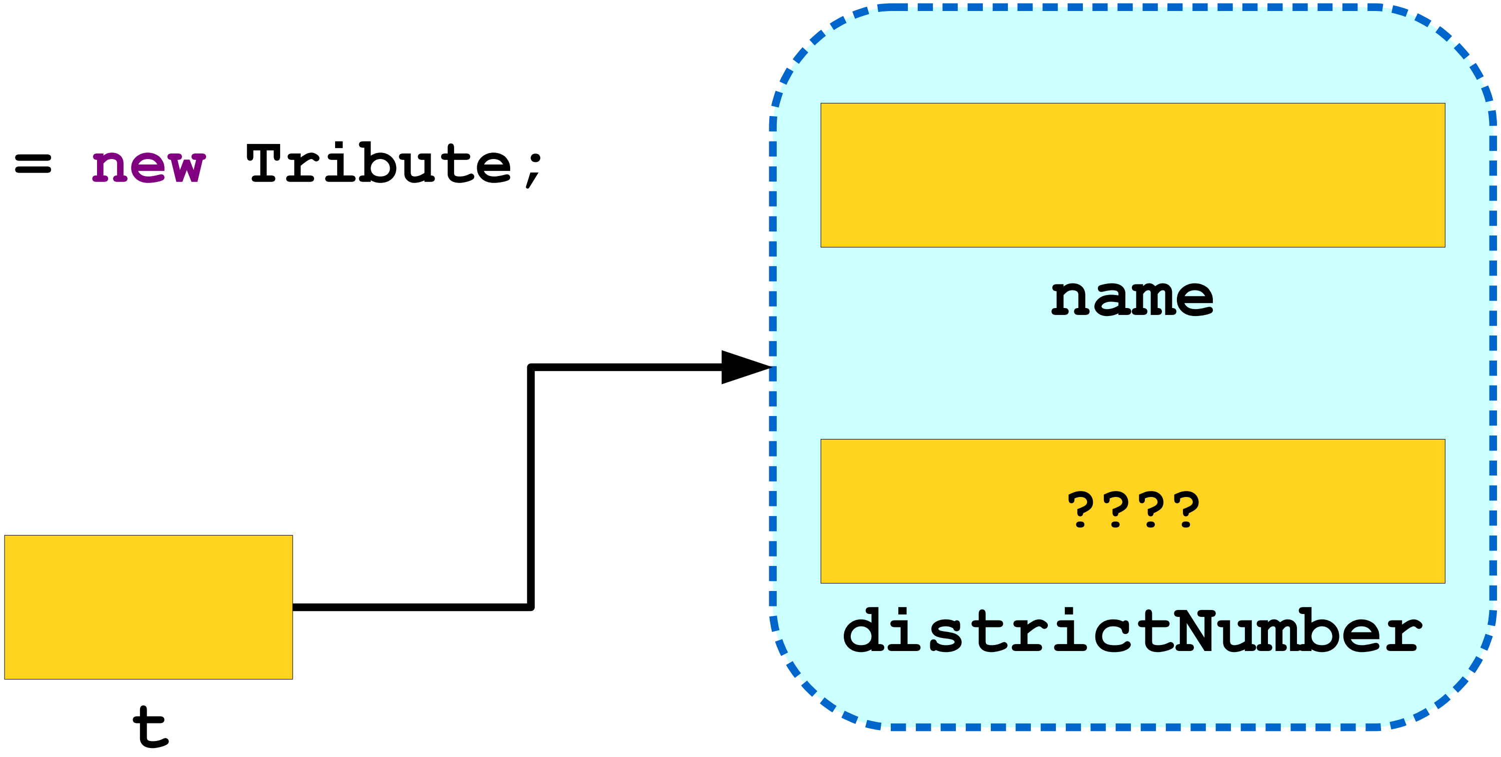
Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```



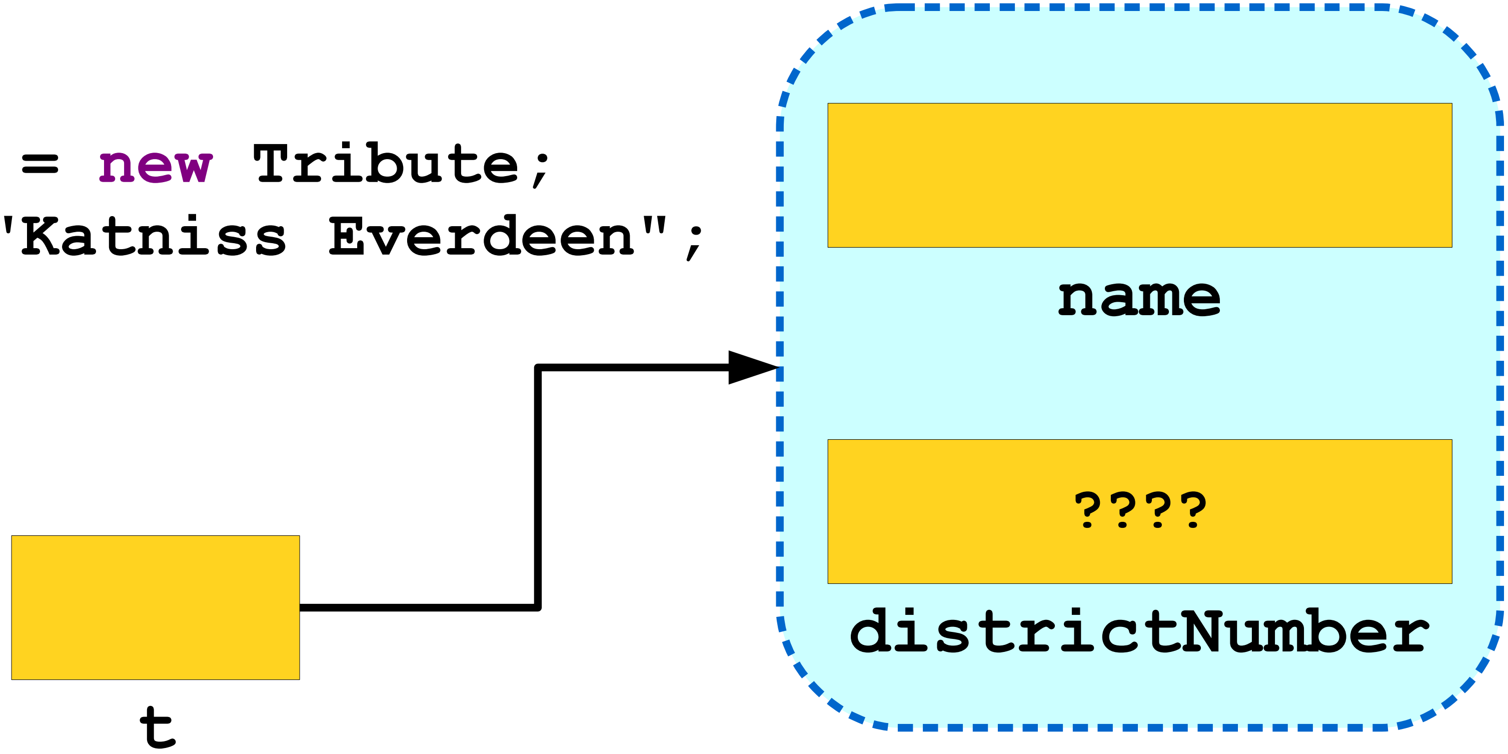
Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```



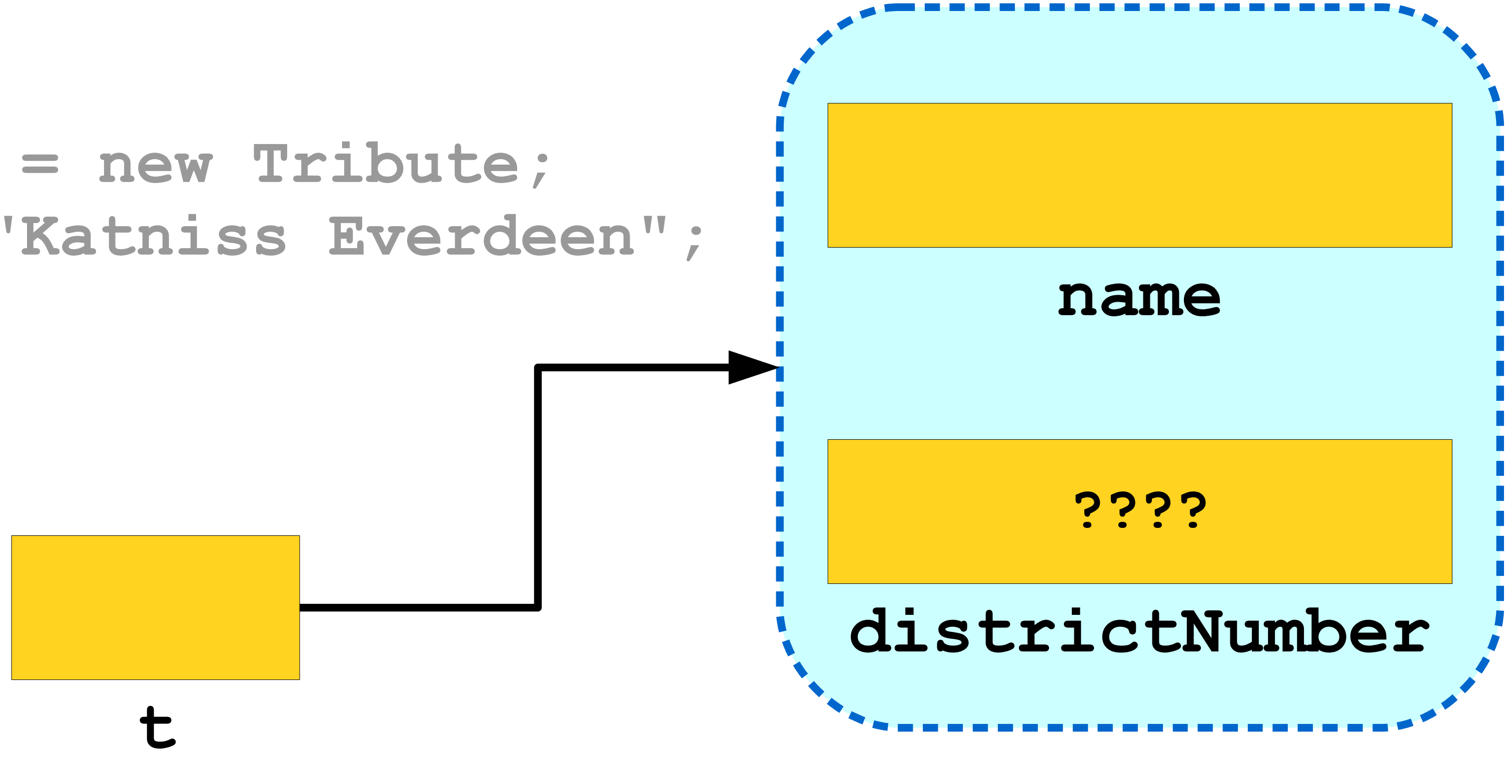
Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
```



Dynamic Memory Allocation

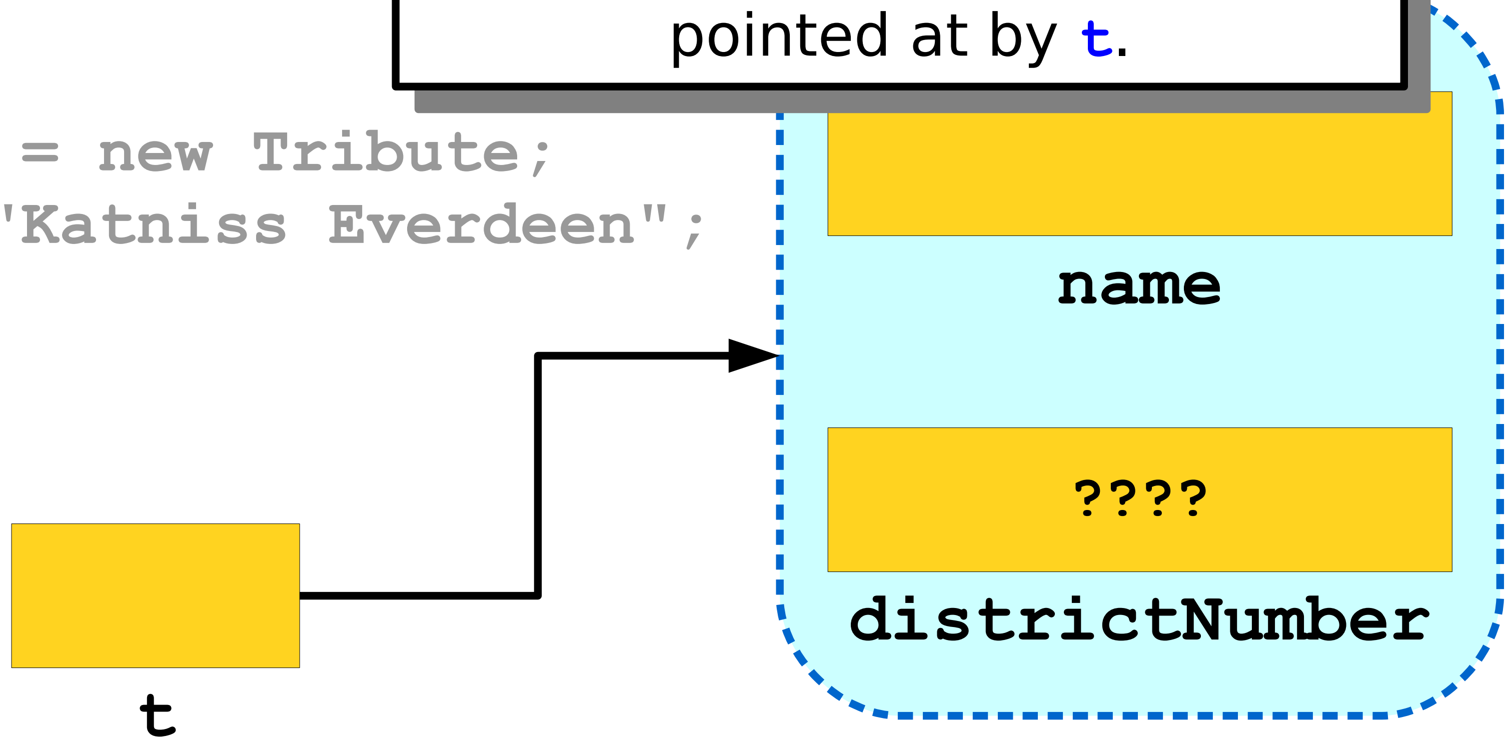
```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
```



Dynamic Memory Allocation

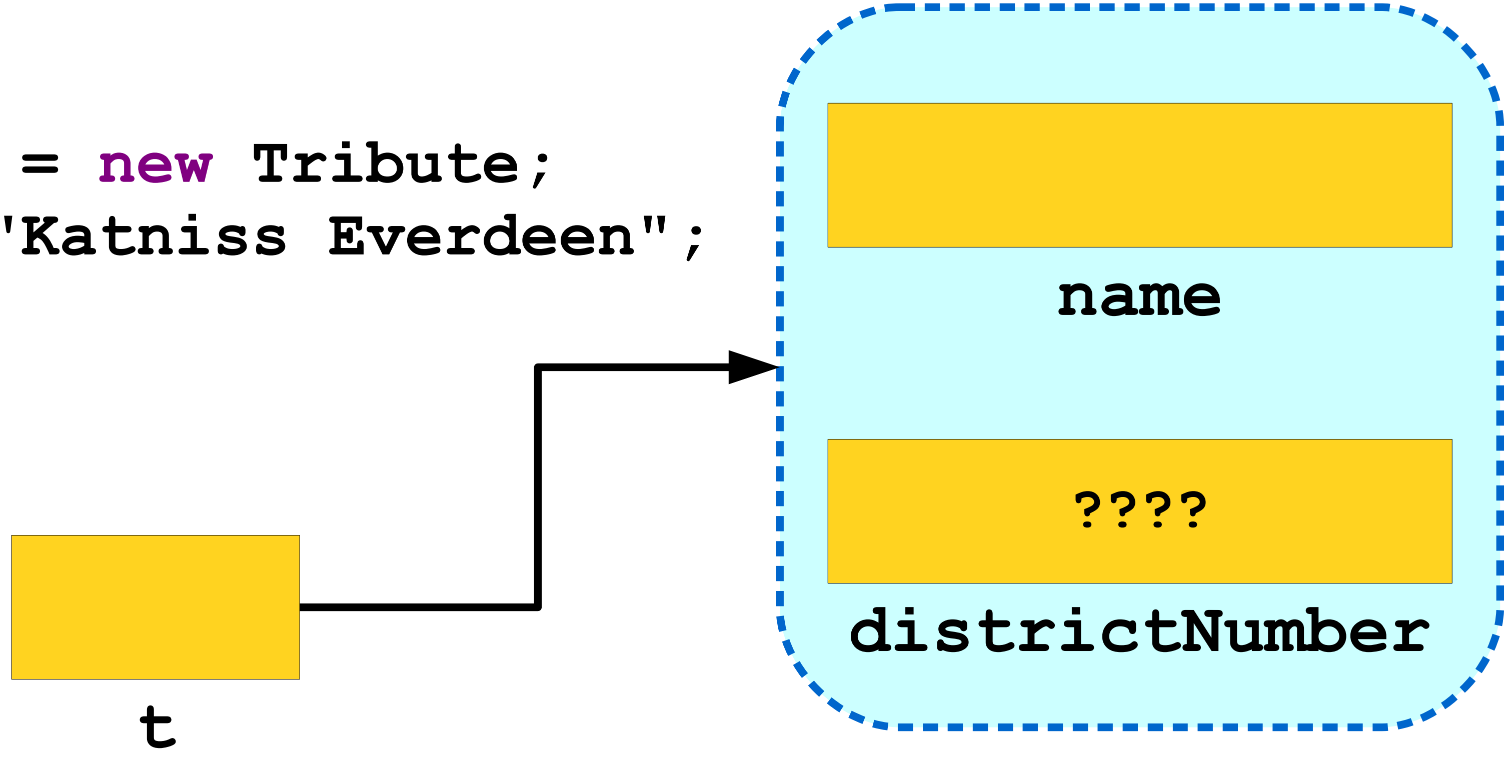
```
struct Tribute {
    string name;
    int districtNumbe
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
```

Because **t** is a pointer to a **Tribute**, not an actual **Tribute**, we have to use the arrow operator to access the fields pointed at by **t**.



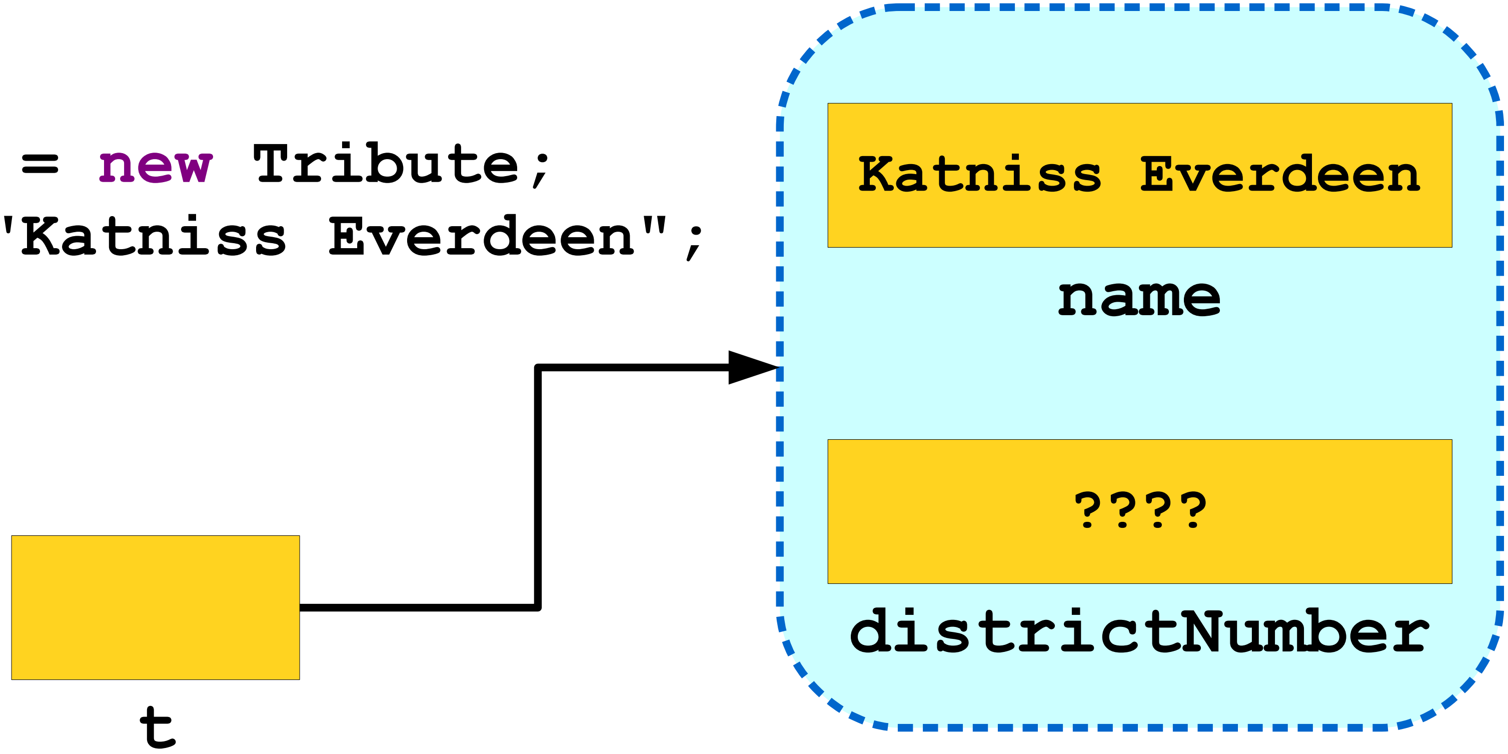
Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
```



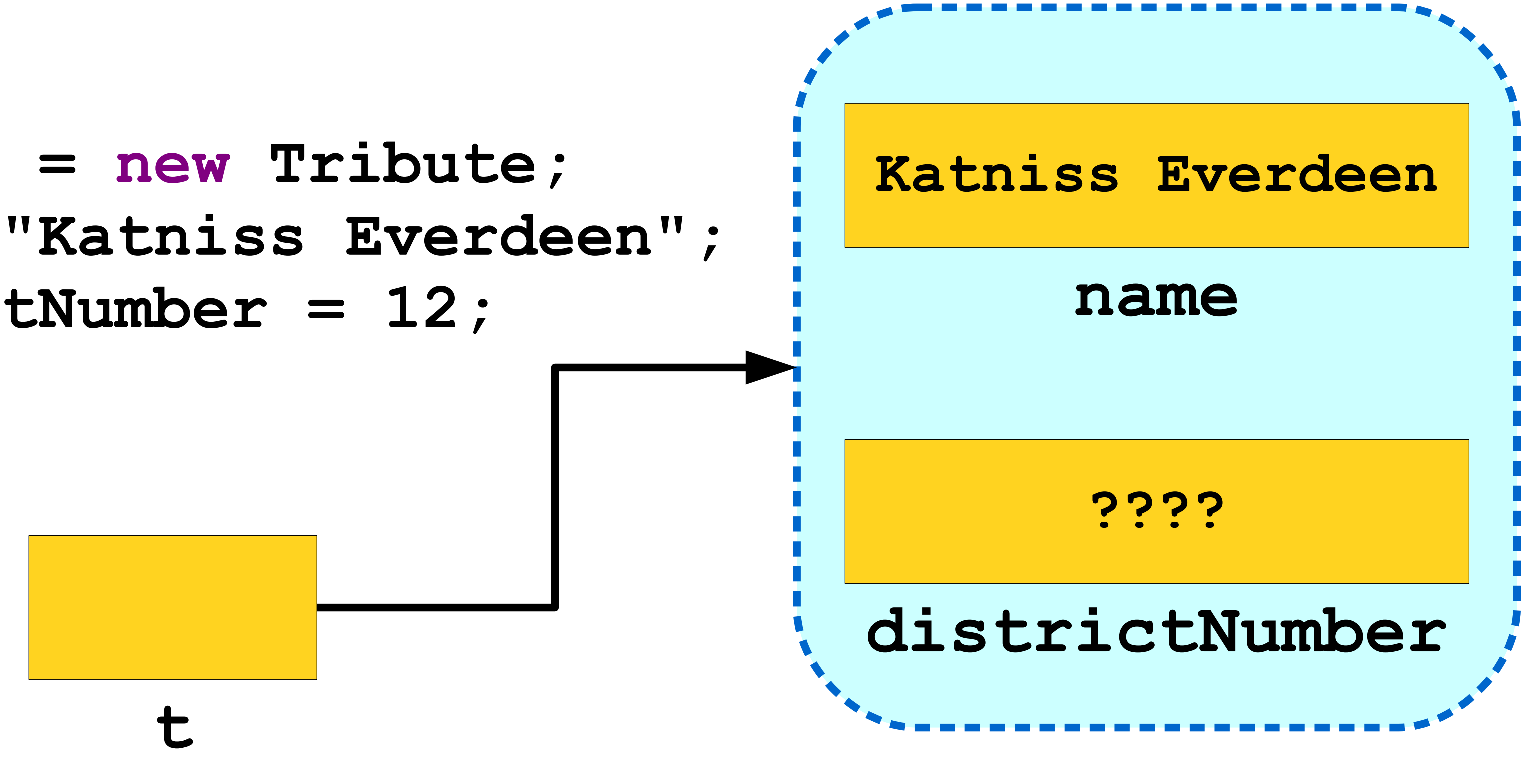
Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
```



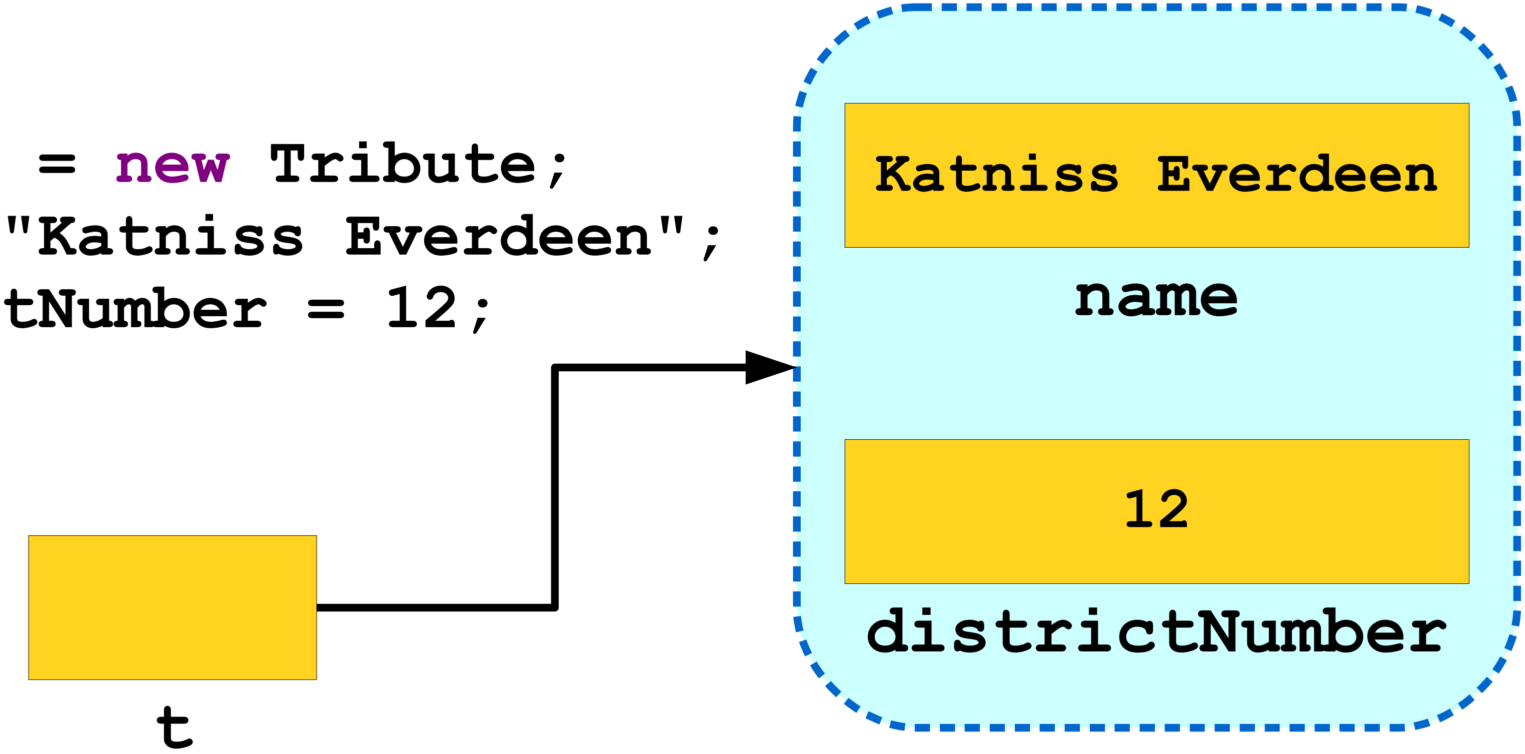
Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
t->districtNumber = 12;
```



Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
t->districtNumber = 12;
```



Cleaning Up

- As with dynamic arrays, you are responsible for cleaning up memory allocated with **new**.
- You can deallocate memory with the **delete** keyword:

```
delete ptr;
```

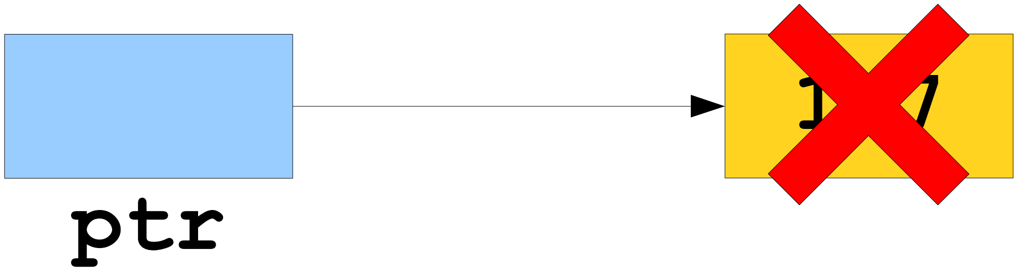
- This destroys the object pointed at by the given pointer, not the pointer itself.

Cleaning Up

- As with dynamic arrays, you are responsible for cleaning up memory allocated with **`new`**.
- You can deallocate memory with the **`delete`** keyword:

```
delete ptr;
```

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



Cleaning Up

- As with dynamic arrays, you are responsible for cleaning up memory allocated with **`new`**.
- You can deallocate memory with the **`delete`** keyword:

```
delete ptr;
```

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

Unfortunately...

- In C++, all of the following result in undefined behavior:
  - Deleting an object with **`delete[]`** that was allocated with **`new`**.
  - Deleting an object with **`delete`** that was allocated with **`new[]`**.
- Although it is not always an error, it is usually a Very Bad Idea to treat an array like a single object or vice-versa.

Building our Vocabulary

- In order to use linked lists, we will need to introduce or revisit several new language features:
  - Structures
  - Dynamic allocation
  - Null pointers

Building our Vocabulary

In order to use linked lists, we will need to introduce or revisit several new language features:

Structures

Dynamic allocation

- Null pointers

A Pointless Exercise

- When working with pointers, we sometimes wish to indicate that a pointer is not pointing to anything.
- In C++, you can set a pointer to **`NULL`** to indicate that it is not pointing to an object:

```
ptr = NULL;
```

- This is **not** the default value for pointers; by default, pointers default to a garbage value.

Building our Vocabulary

- In order to use linked lists, we will need to introduce or revisit several new language features:
  - Structures
  - Dynamic allocation
  - Null pointers

Building our Vocabulary

In order to use linked lists, we will need to introduce or revisit several new language features:

- Structures
- Dynamic allocation
- Null pointers

And now... linked lists!

Linked List Cells

- A linked list is a chain of **cells**.
- Each cell contains two pieces of information:
  - Some piece of data that is stored in the sequence, and
  - A **link** to the next cell in the list.
- We can traverse the list by starting at the first cell and repeatedly following its link.

Representing a Cell

- For simplicity, let's assume we're building a linked list of **`string`**s.
- We can represent a cell in the linked list as a structure:

```
struct Cell {
    string value;
    /* ? */ next;
};
```

Representing a Cell

- For simplicity, let's assume we're building a linked list of **string**s.
- We can represent a cell in the linked list as a structure:

```
struct Cell {
    string value;
    Cell* next;
};
```

Representing a Cell

- For simplicity, let's assume we're building a linked list of **`string`**s.
- We can represent a cell in the linked list as a structure:

```
struct Cell {
    string value;
    Cell* next;
};
```

- **The structure is defined recursively!**

Creating a Linked List
(Pseudocode + Drawing)

Building Linked Lists