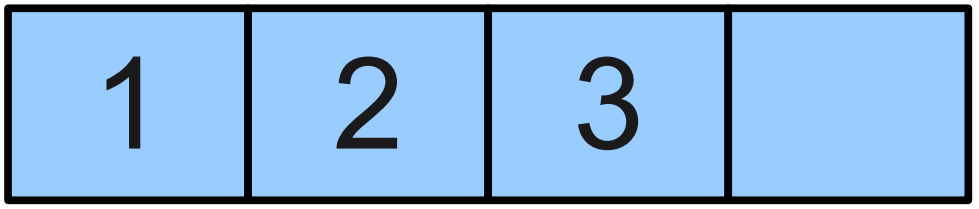
Linked Lists

Apply to Section Lead!

http://cs198.stanford.edu

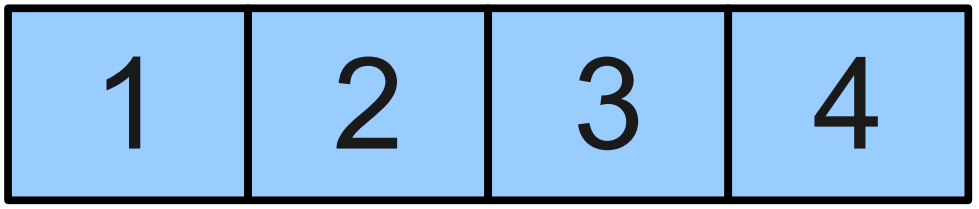
Array-Based Allocation

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



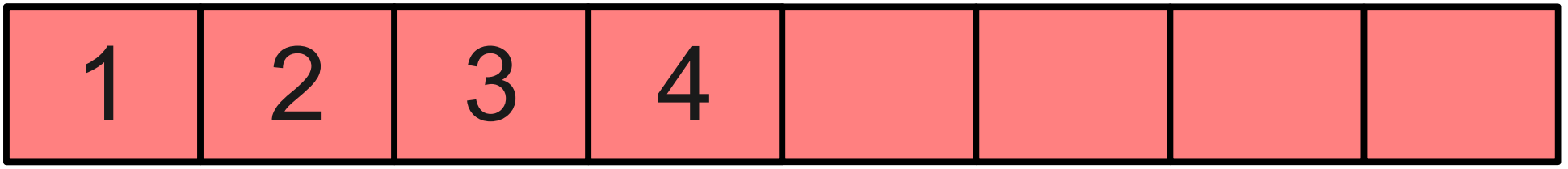
Array-Based Allocation

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



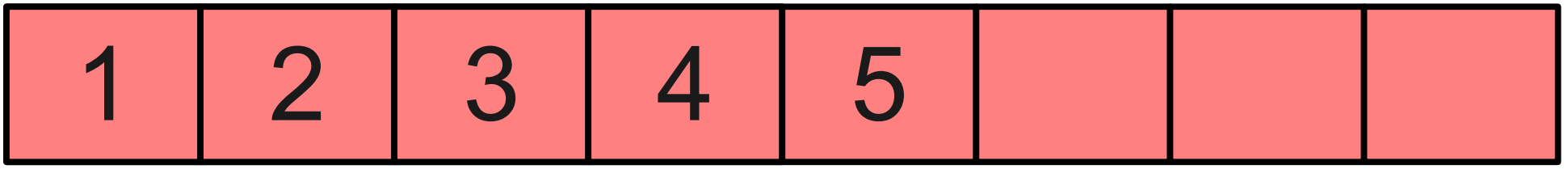
Array-Based Allocation

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



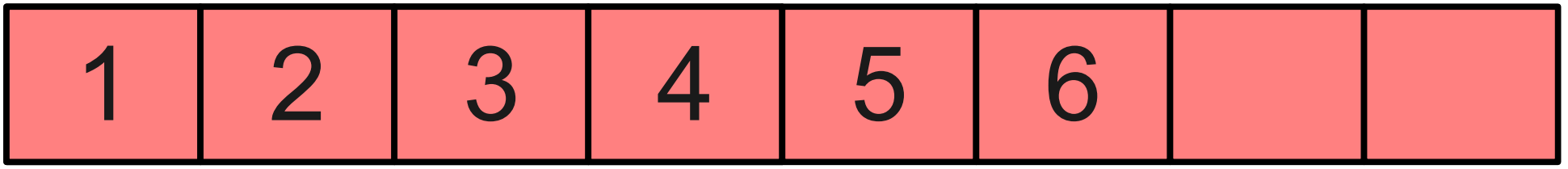
Array-Based Allocation

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



Array-Based Allocation

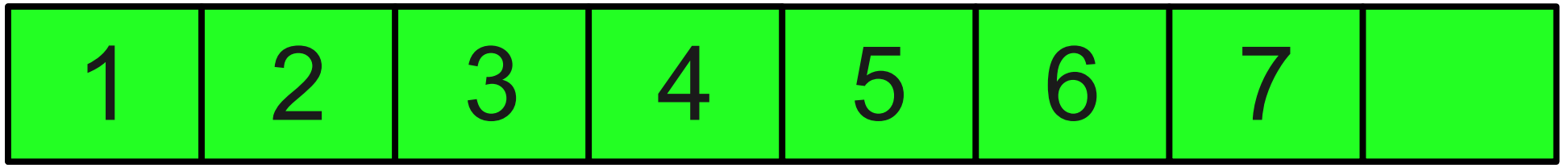
- Our current implementation of Vector and Stack use dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a **huge** new array and move everything over.



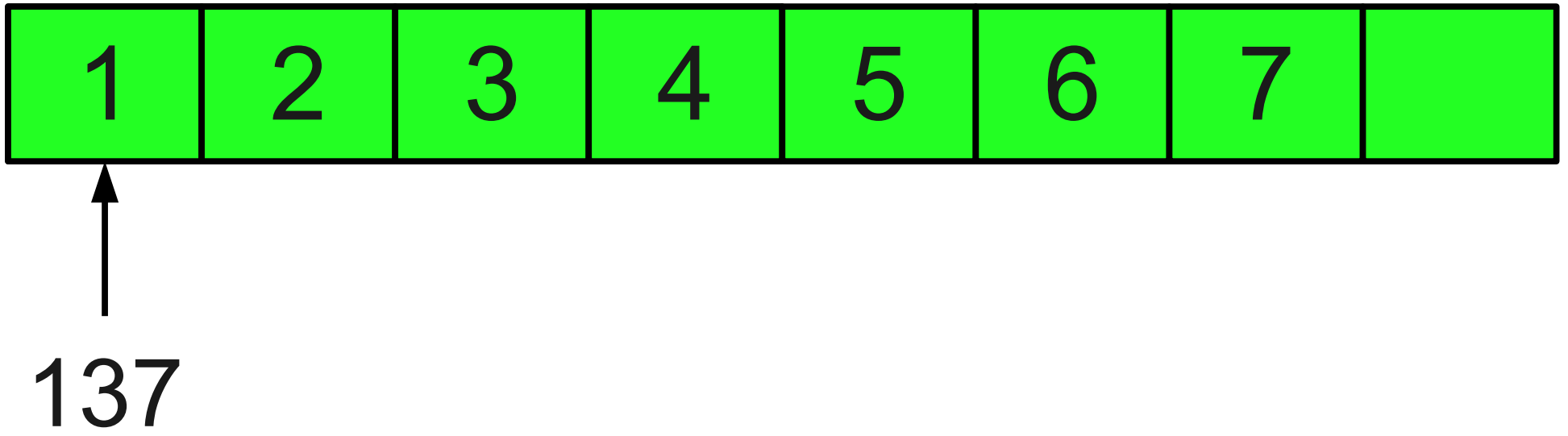
A Different Idea

- 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!

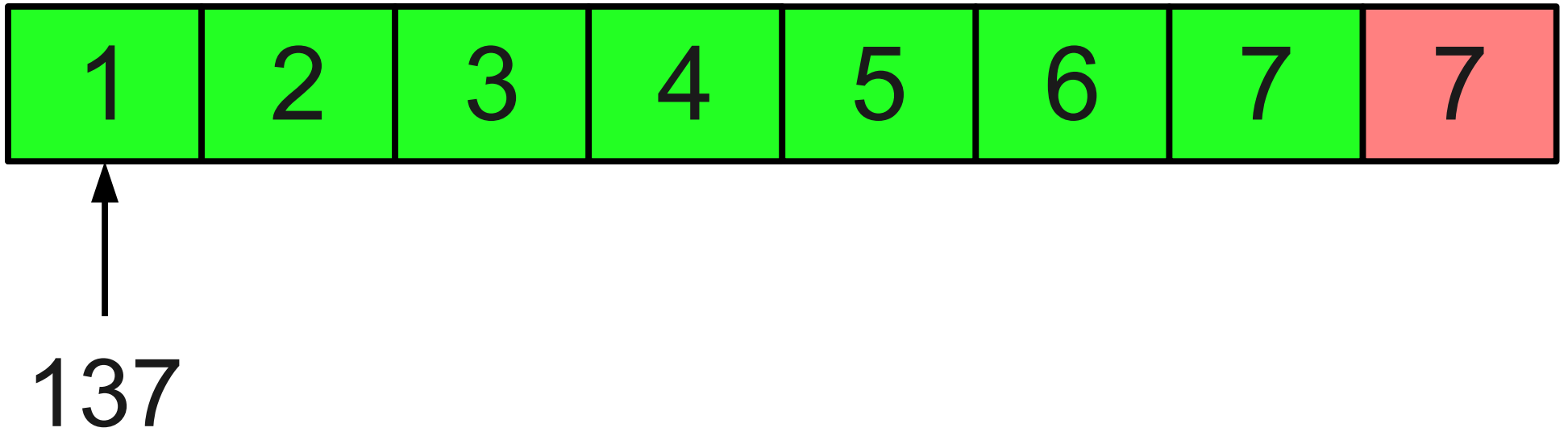
Excuse Me, Coming Through...



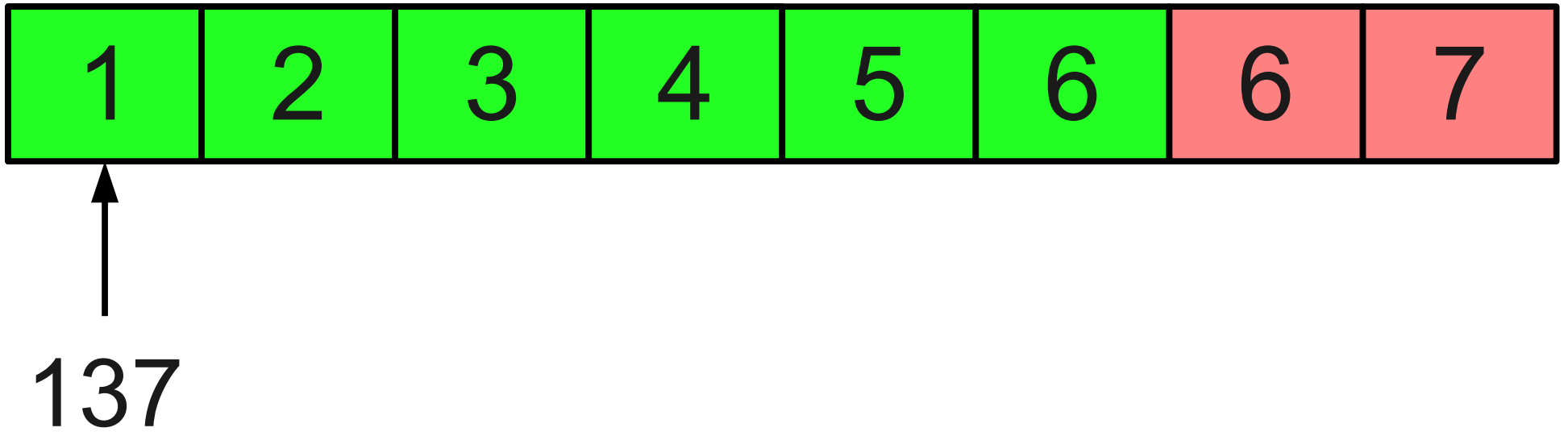
Excuse Me, Coming Through...



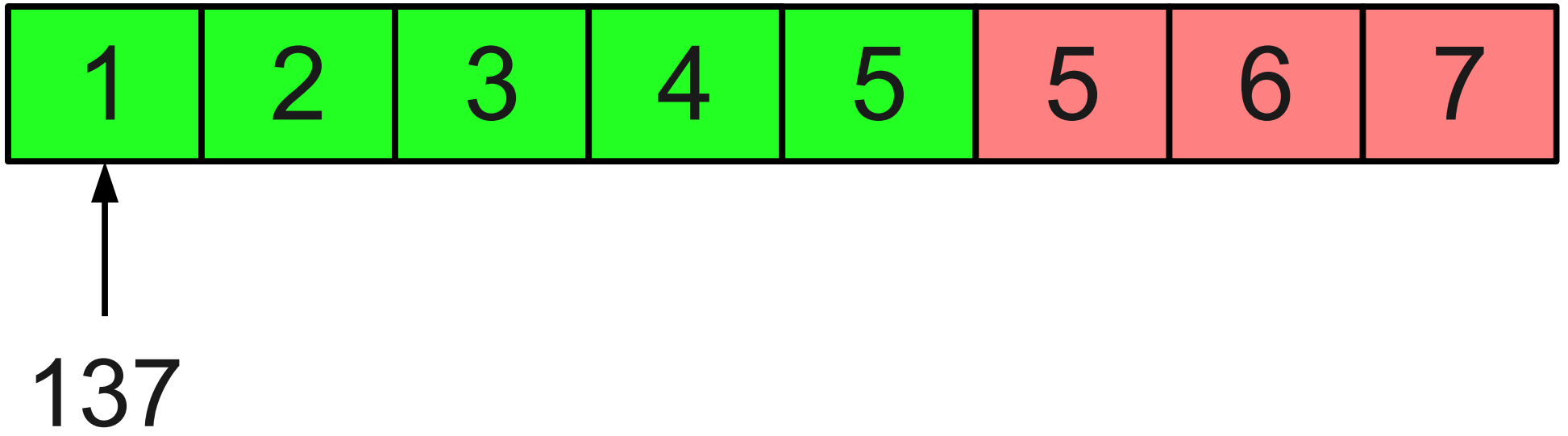
Excuse Me, Coming Through...



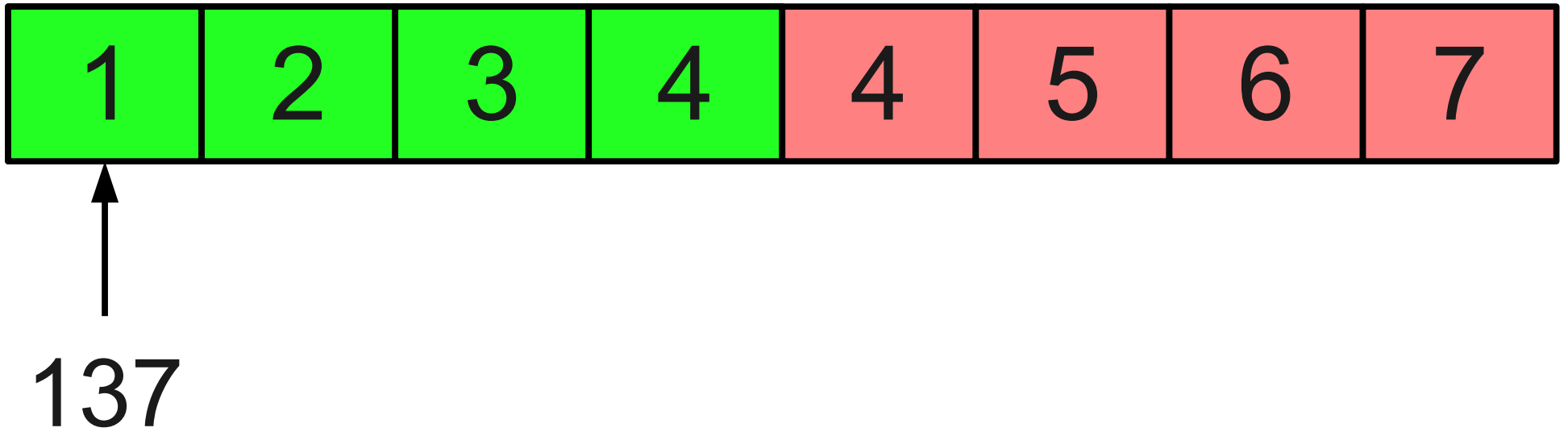
Excuse Me, Coming Through...



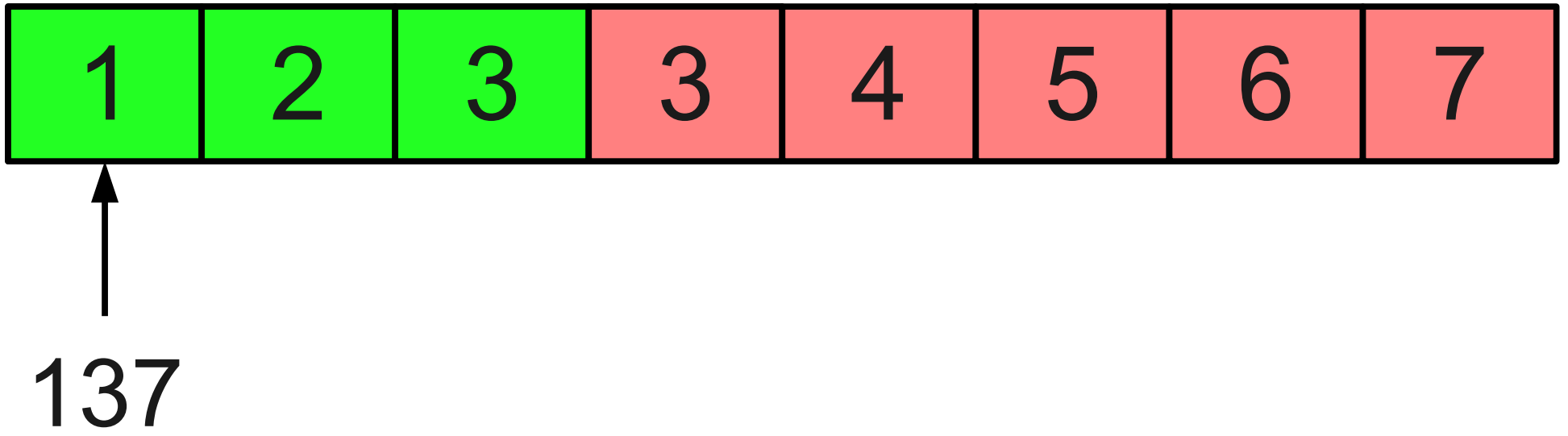
Excuse Me, Coming Through...



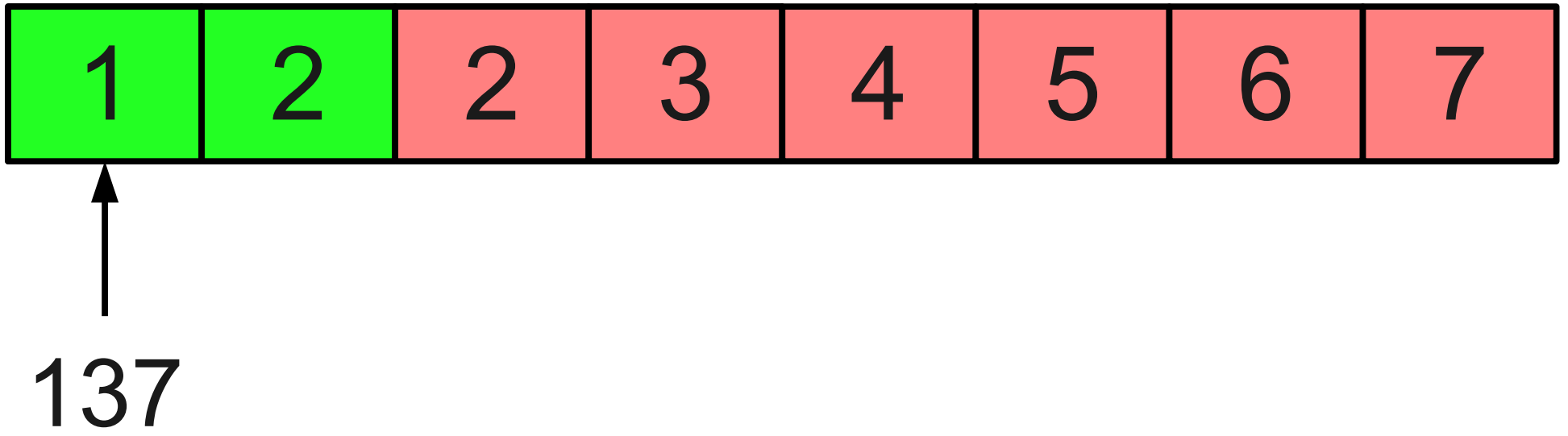
Excuse Me, Coming Through...



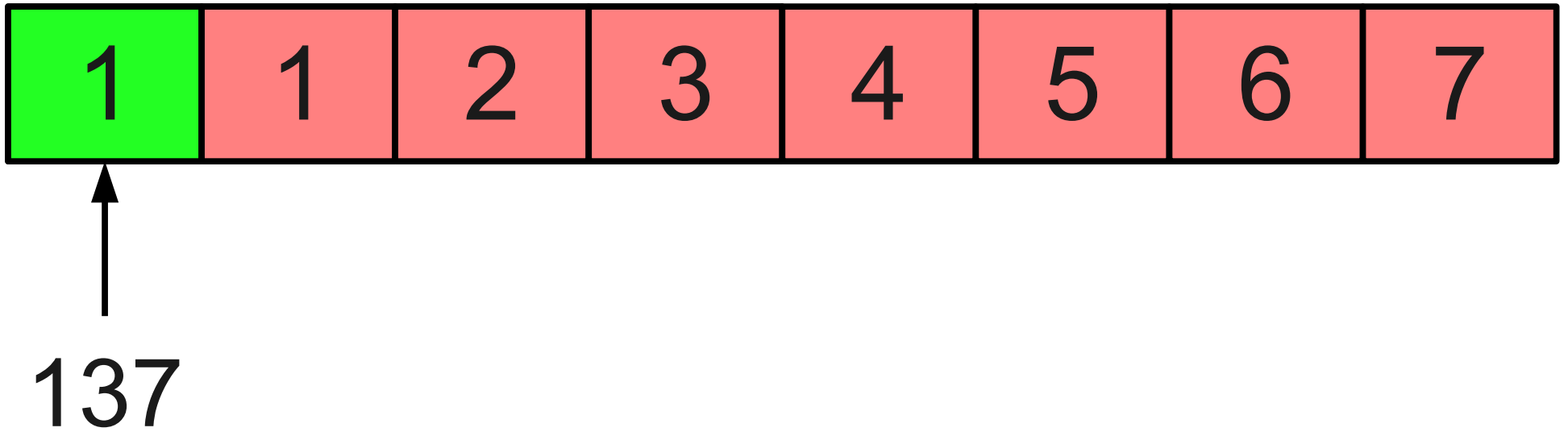
Excuse Me, Coming Through...



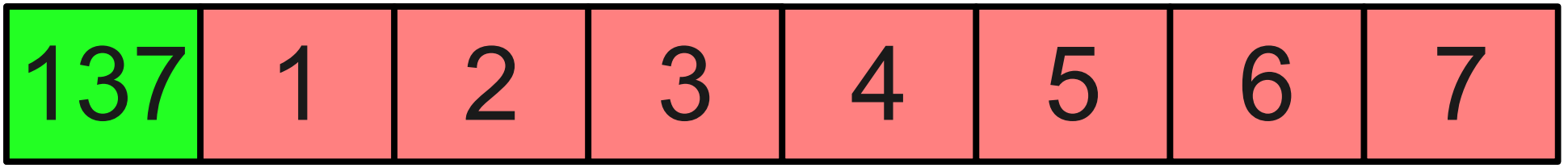
Excuse Me, Coming Through...



Excuse Me, Coming Through...

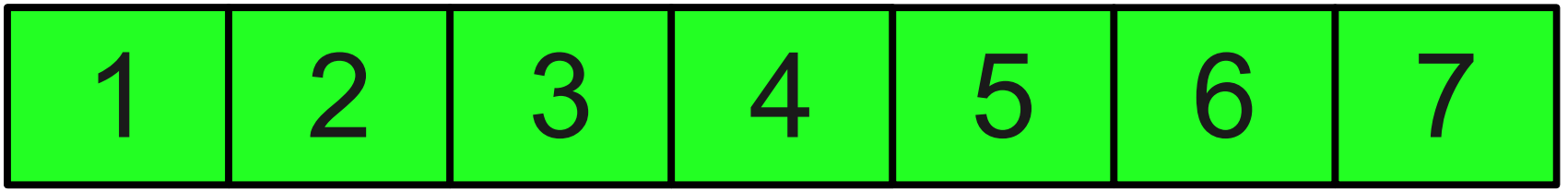


Excuse Me, Coming Through...



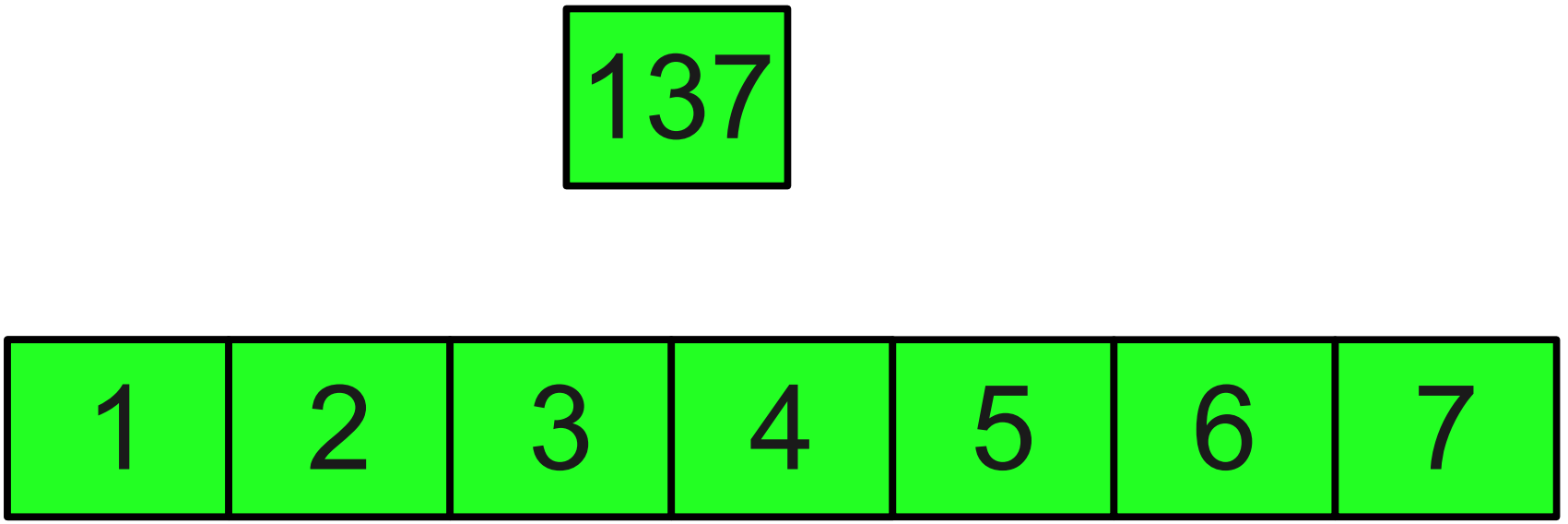
Shoving Things Over

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



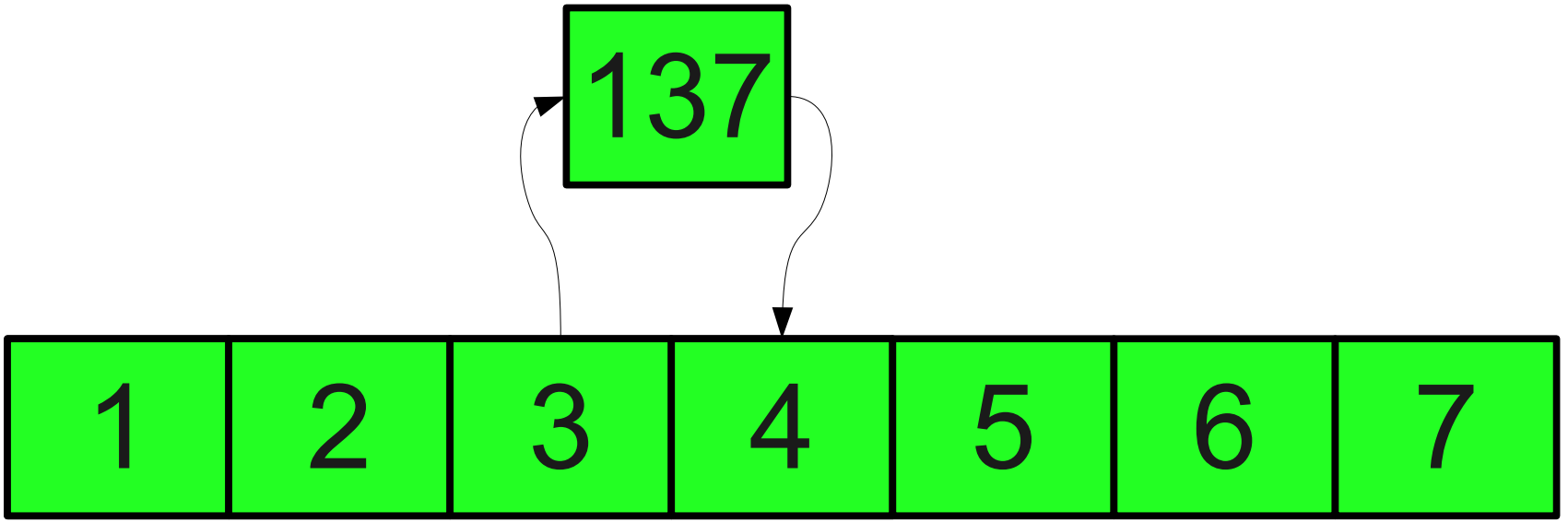
Shoving Things Over

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



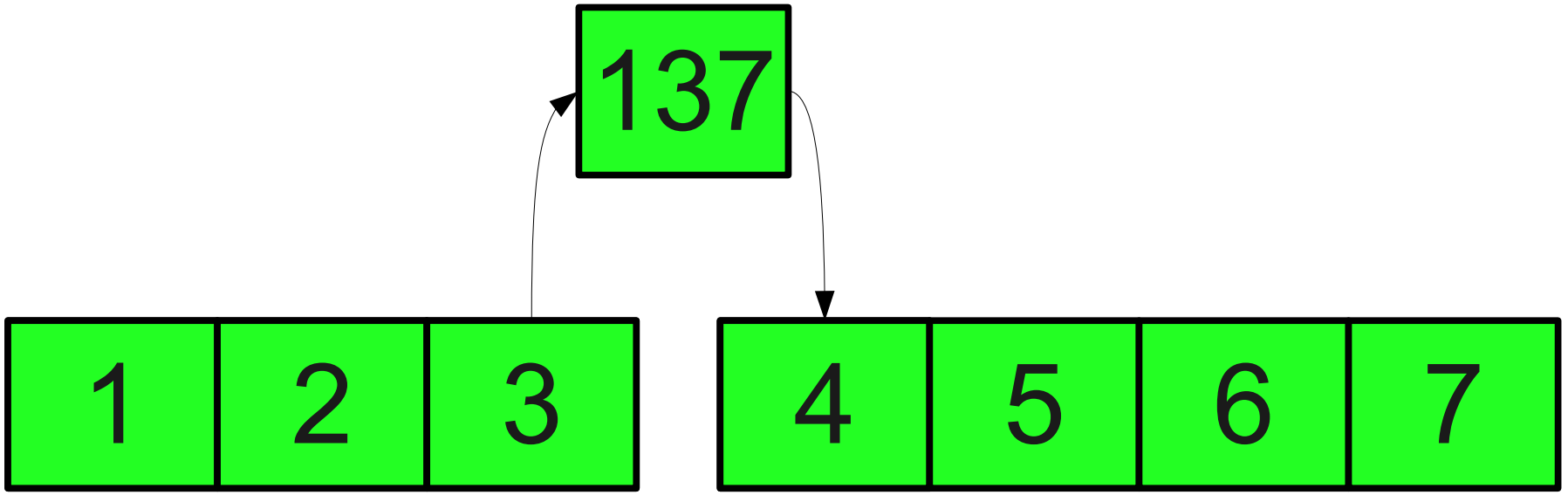
Shoving Things Over

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



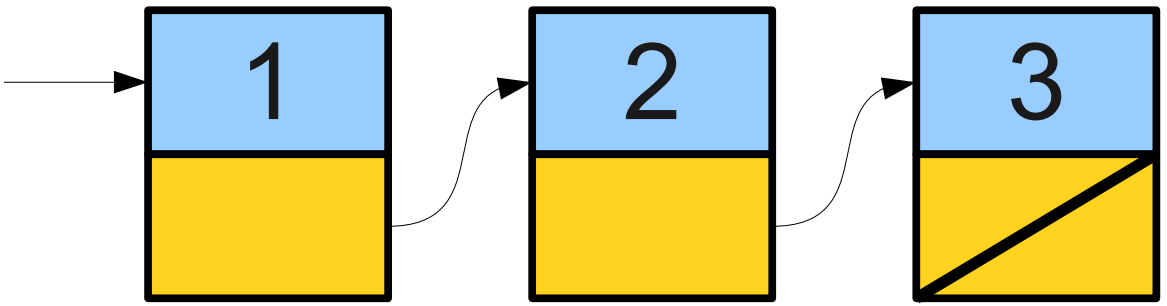
Shoving Things Over

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



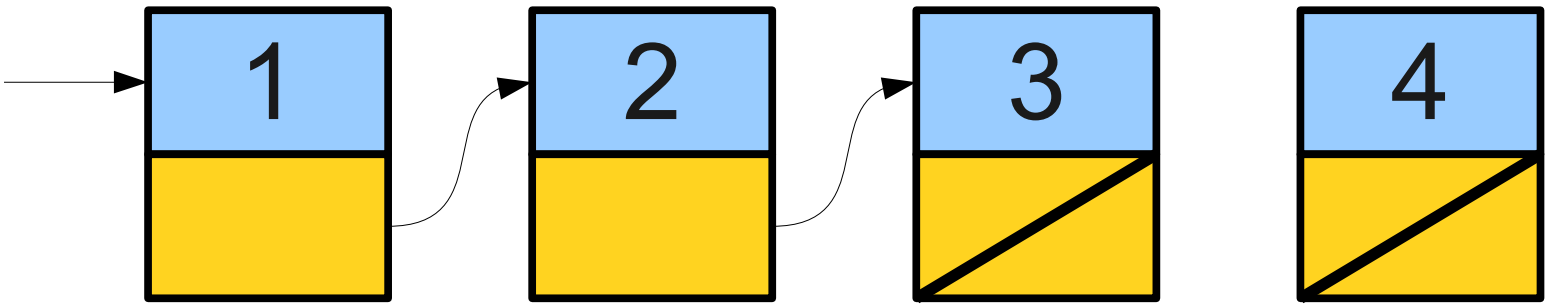
Linked Lists at a Glance

- 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.



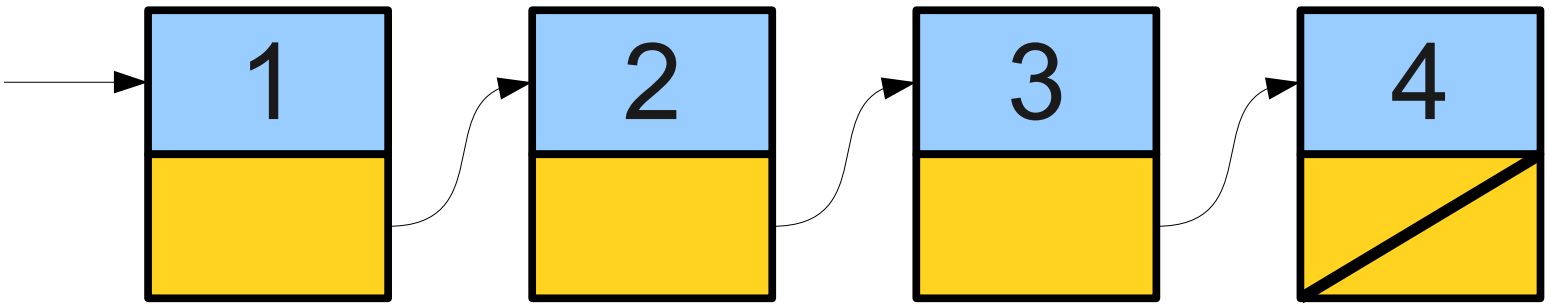
Linked Lists at a Glance

- 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.



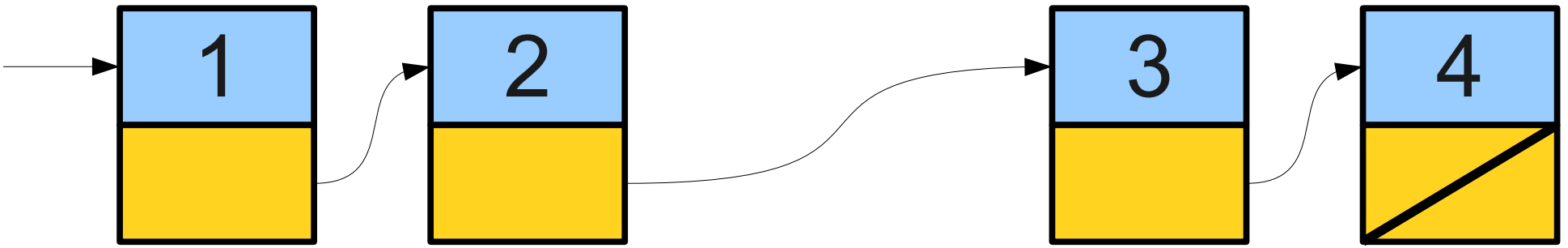
Linked Lists at a Glance

- 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.



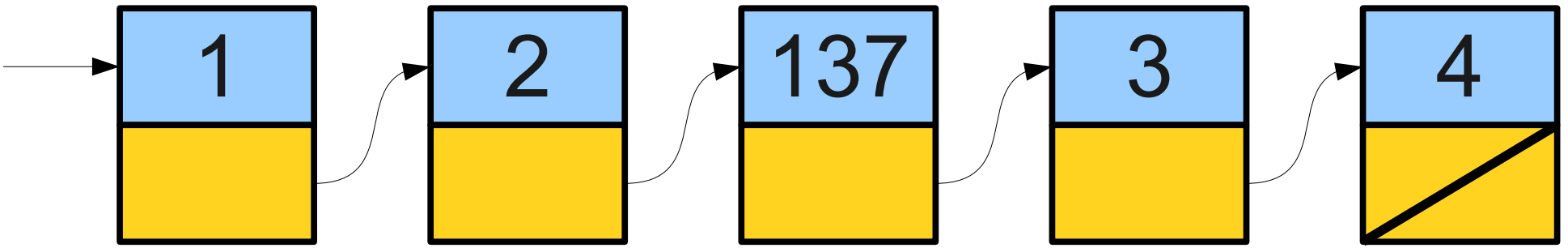
Linked Lists at a Glance

- 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.



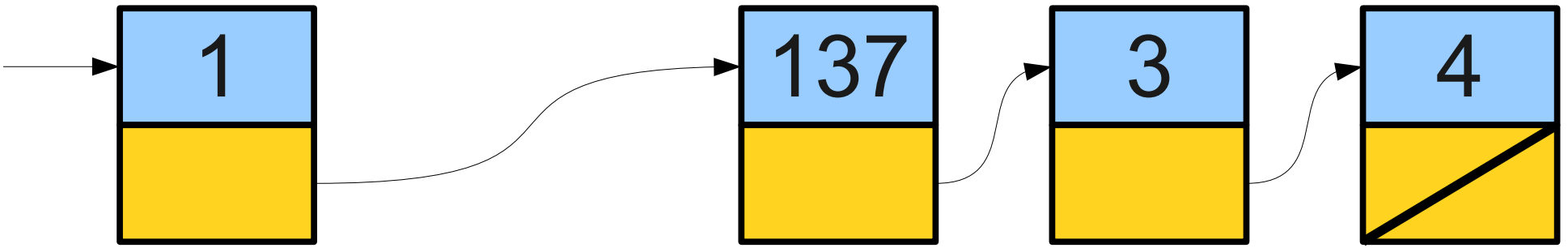
Linked Lists at a Glance

- 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; worst-case insertion is efficient.
- 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 = 12;
```

`structs` and `classes`

- 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 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`*****`)`**

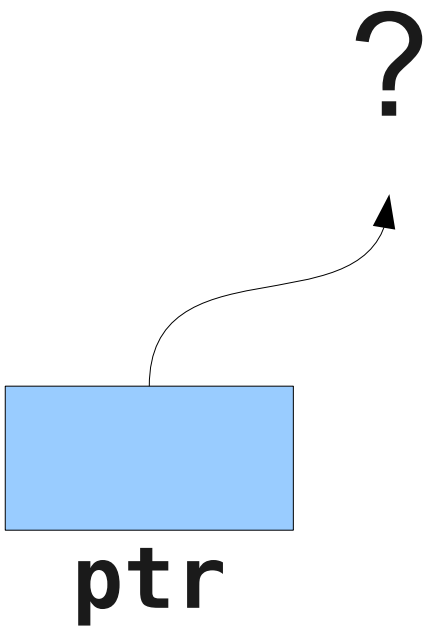
  creates a new object of type ***`T`*** passing the appropriate arguments to the constructor, then returns a pointer to it.

Dynamic Memory Allocation

```
int* ptr;
```

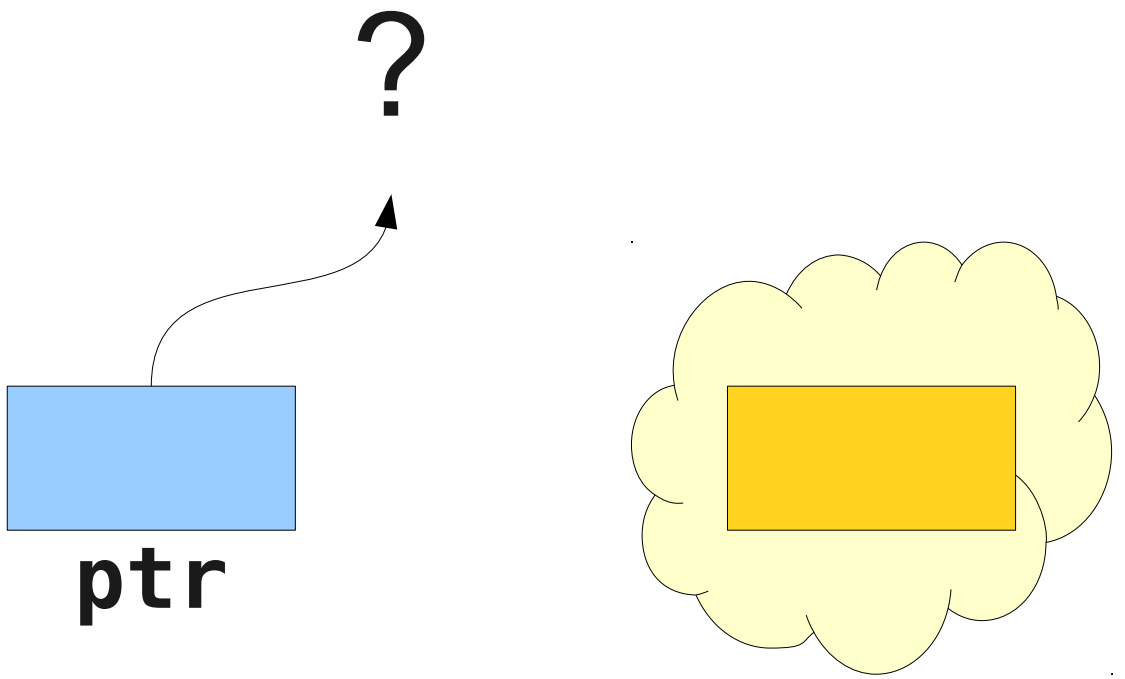
Dynamic Memory Allocation

```
int* ptr;
```



Dynamic Memory Allocation

```
int* ptr;
ptr = new int;
```



Dynamic Memory Allocation

```
int* ptr;
ptr = new int;
```

Dynamic Memory Allocation

```
int* ptr;
ptr = new int;

*ptr = 137;
```

Dynamic Memory Allocation

```
int* ptr;
ptr = new int;

*ptr = 137;
```

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.

Pointers and Structures

A Tale of Dots and Stars

- If we have a pointer to a structure, like this one:

```
Tribute* ptr = new Tribute;
```

- We cannot access the fields by using dot, since **ptr** is not an actual Tribute.
- The following doesn't work either:

```
*ptr.districtNumber = 13;
```

because it's interpreted as

```
*(ptr.districtNumber) = 13
```

and not

```
(*ptr).districtNumber = 13;
```

Arrow to the Rescue

- To access a field in a structure or class through a pointer, you can write

```
(*ptr).districtNumber = 13;
```

- However, it's much easier to use the **arrow operator** (`->`)

```
ptr->districtNumber = 13;
```

- The arrow operator is so convenient that we almost always use it instead of using the parenthesis/star.

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 point to arbitrary locations in memory.

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;
    Cell* next;
};
```

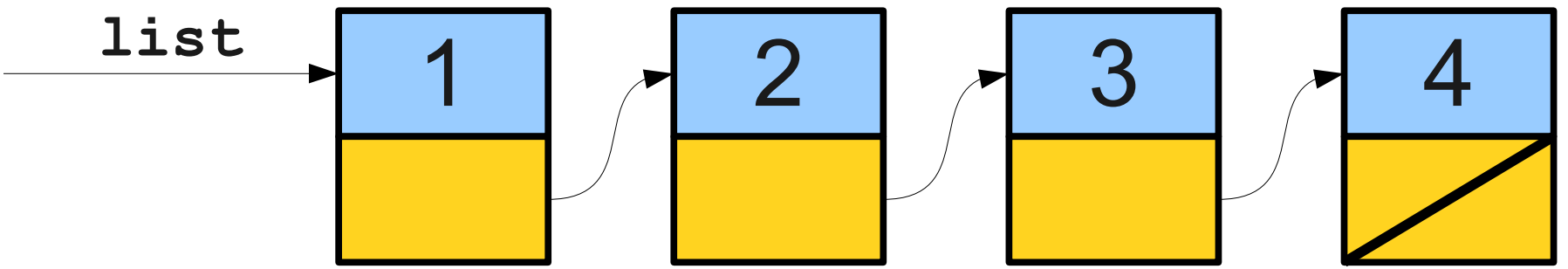
- **The structure is defined recursively!**

Building Linked Lists

Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

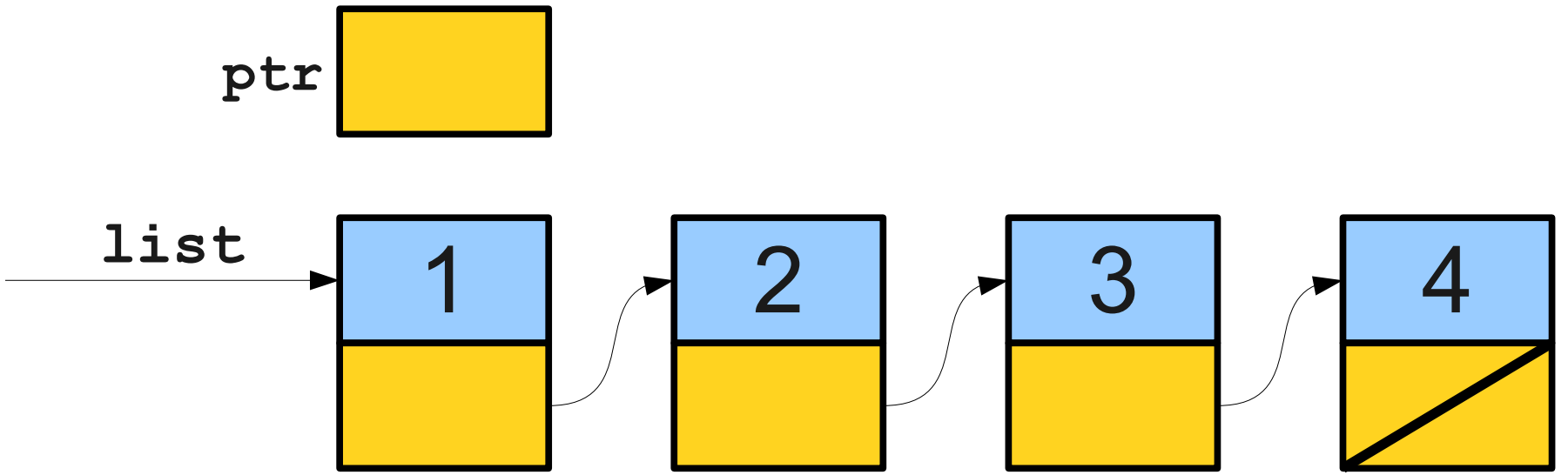
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

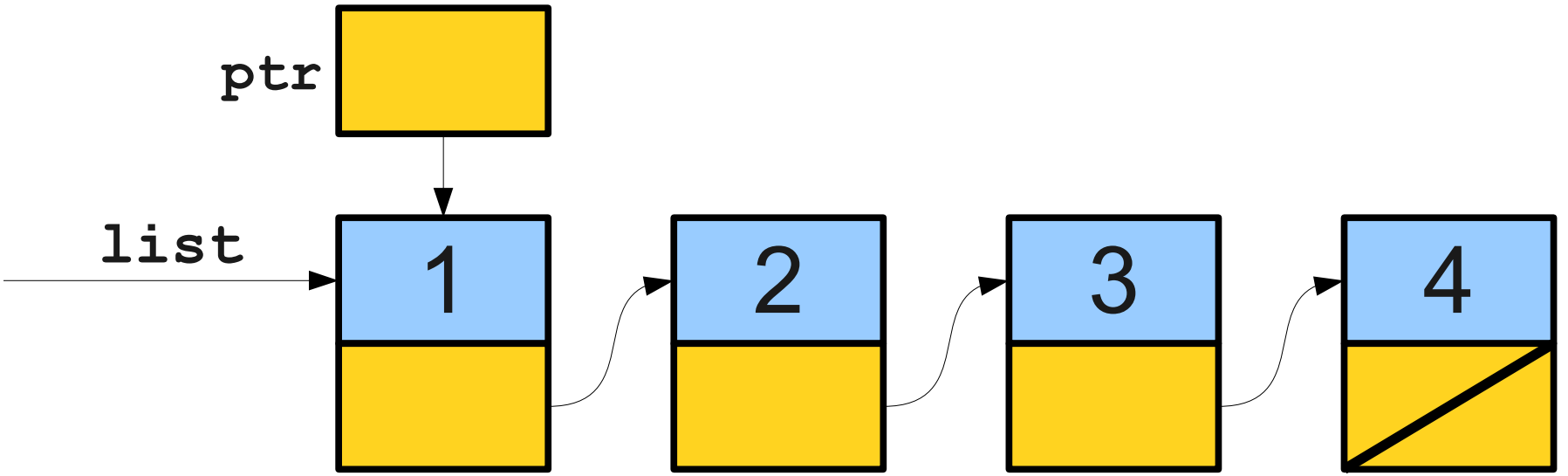
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

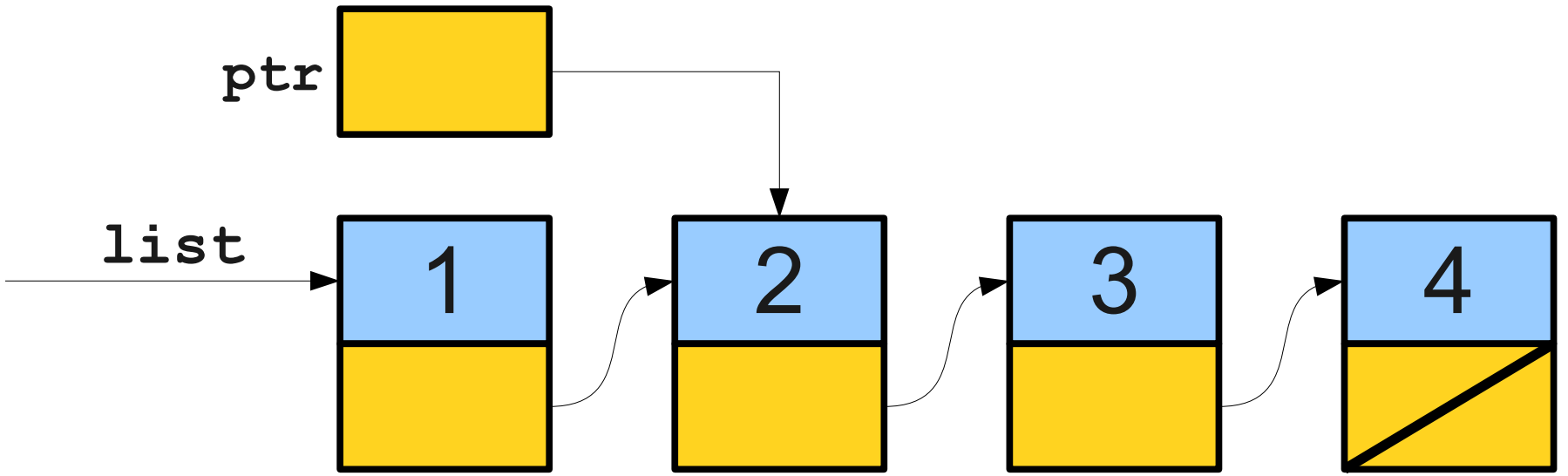
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

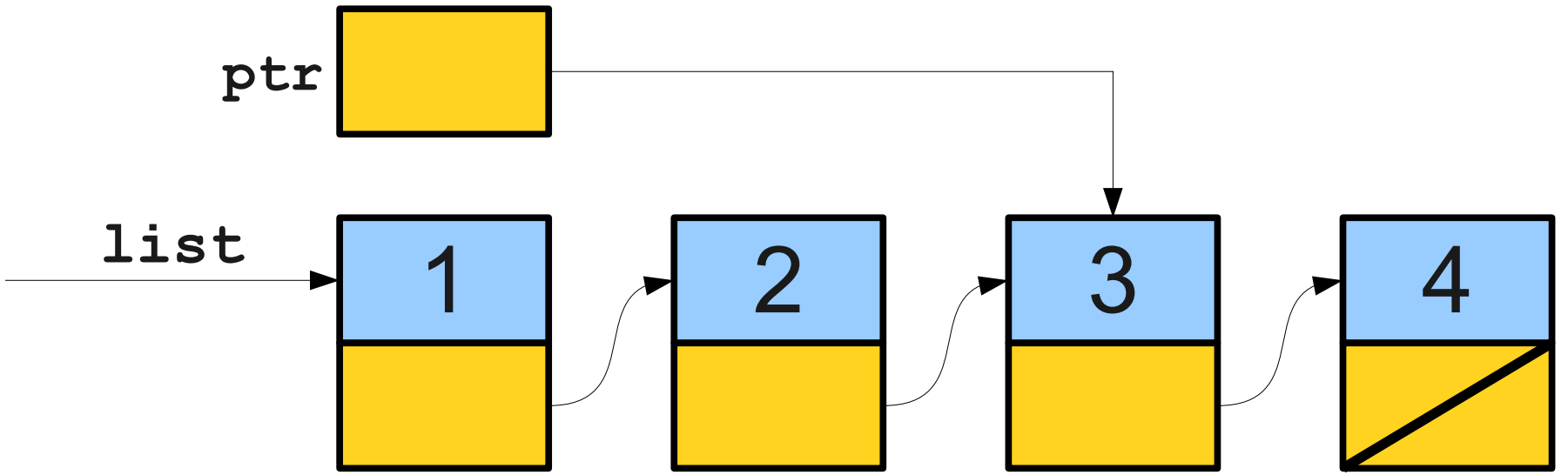
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

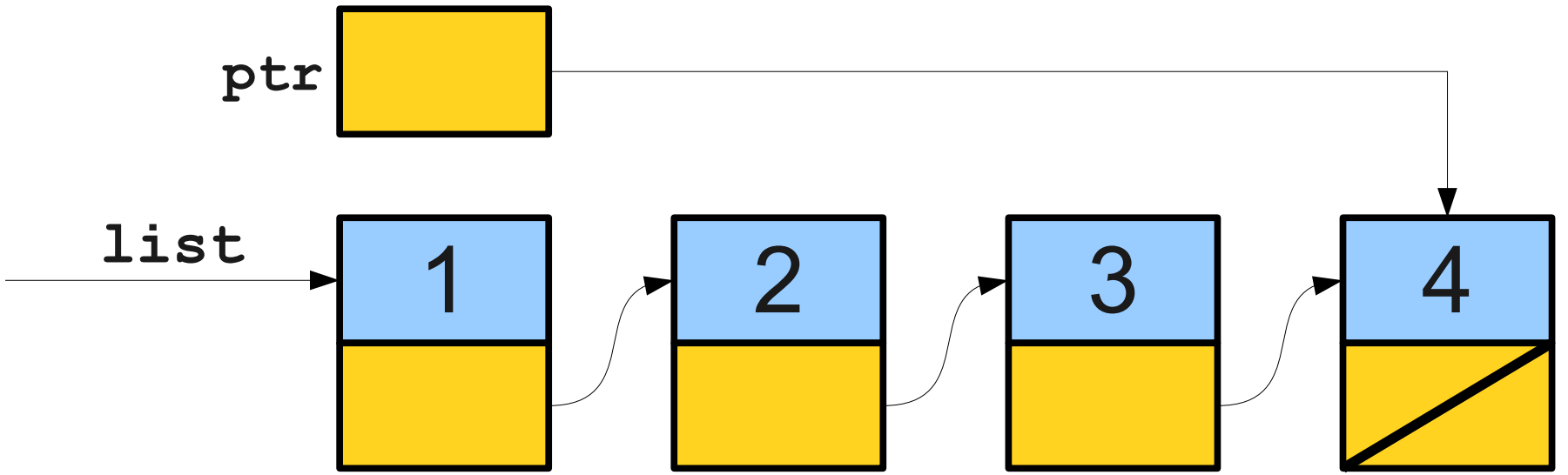
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

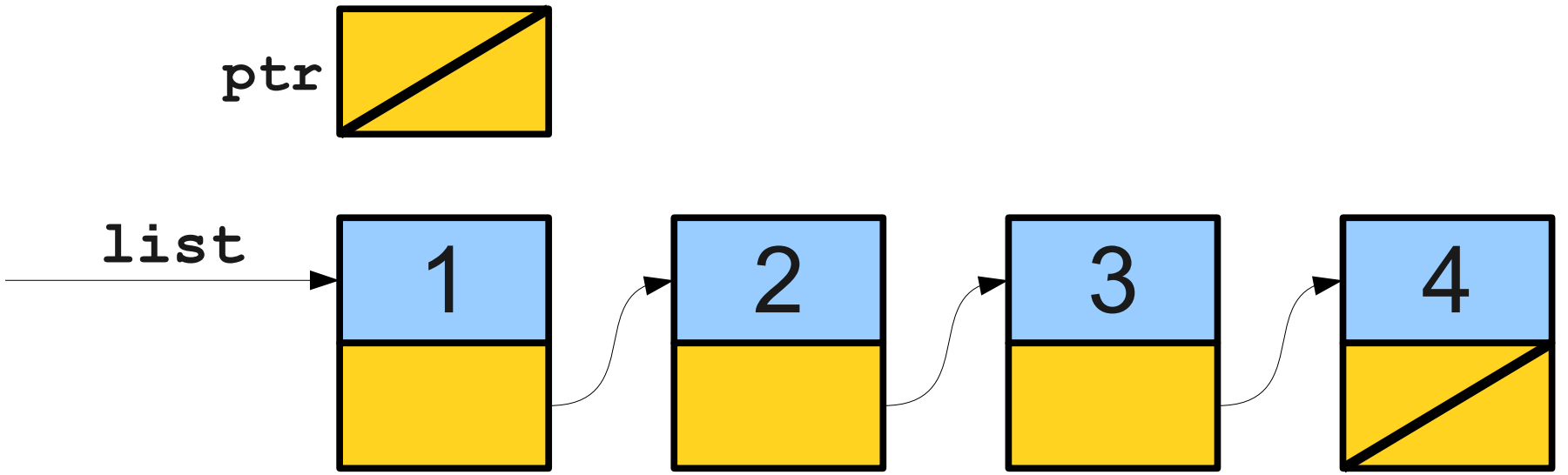
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Once More With Recursion

- Linked lists are defined recursively, and we can traverse them using recursion!

```
void recursiveTraverse(Cell* list) {
    if (list == NULL) return;
    /* … do something with list … */
    recursiveTraverse(list->next);
}
```