Linked Lists

Chris Piech

CS 106B
Lecture 14
Feb 8, 2016

恭禧发财

Midterm


Functions


Collections


Recursion


Exploration


Big O


Reference Sheet


Exam Strategies

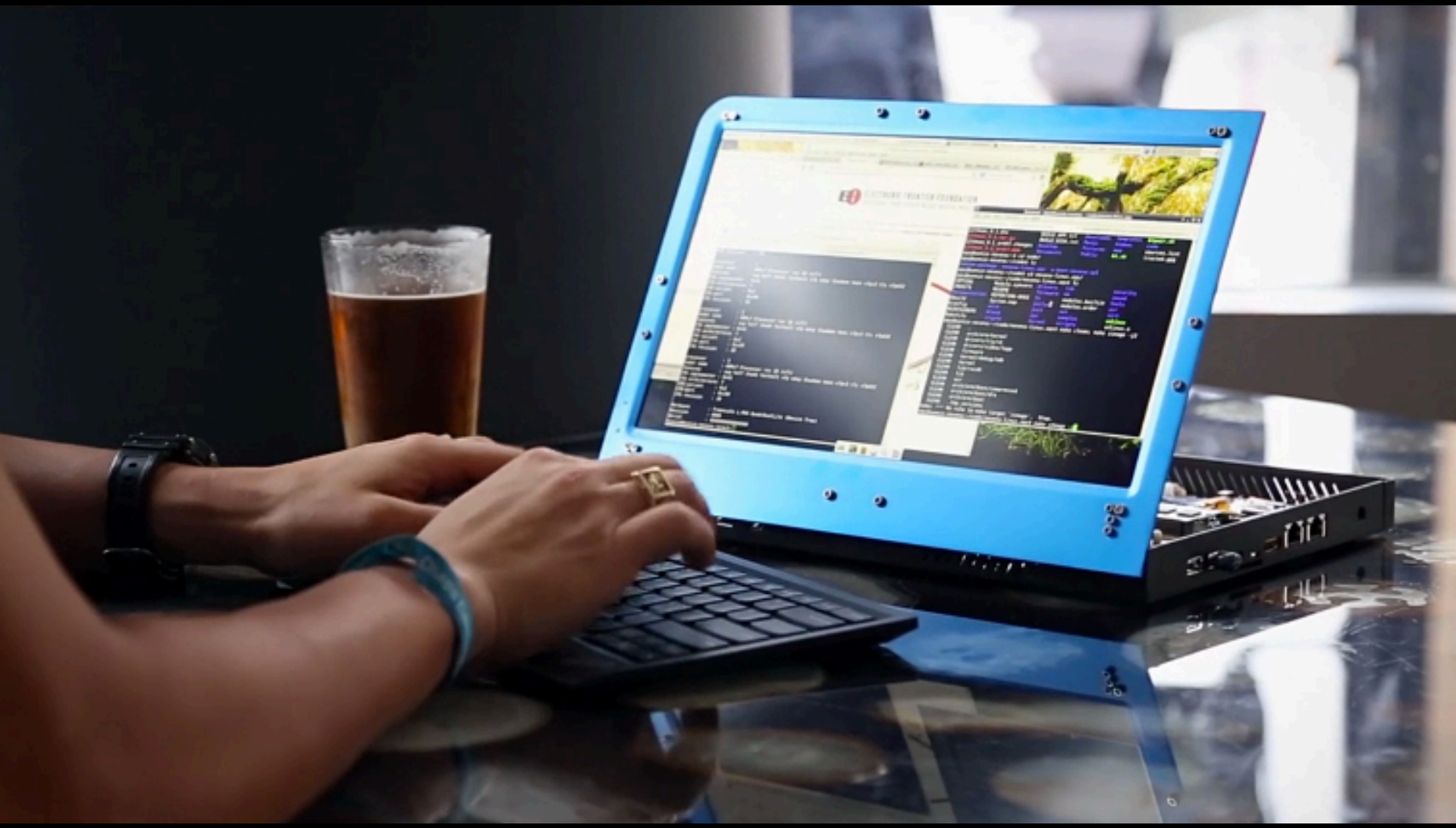

Practice #1


Practice Soln #1

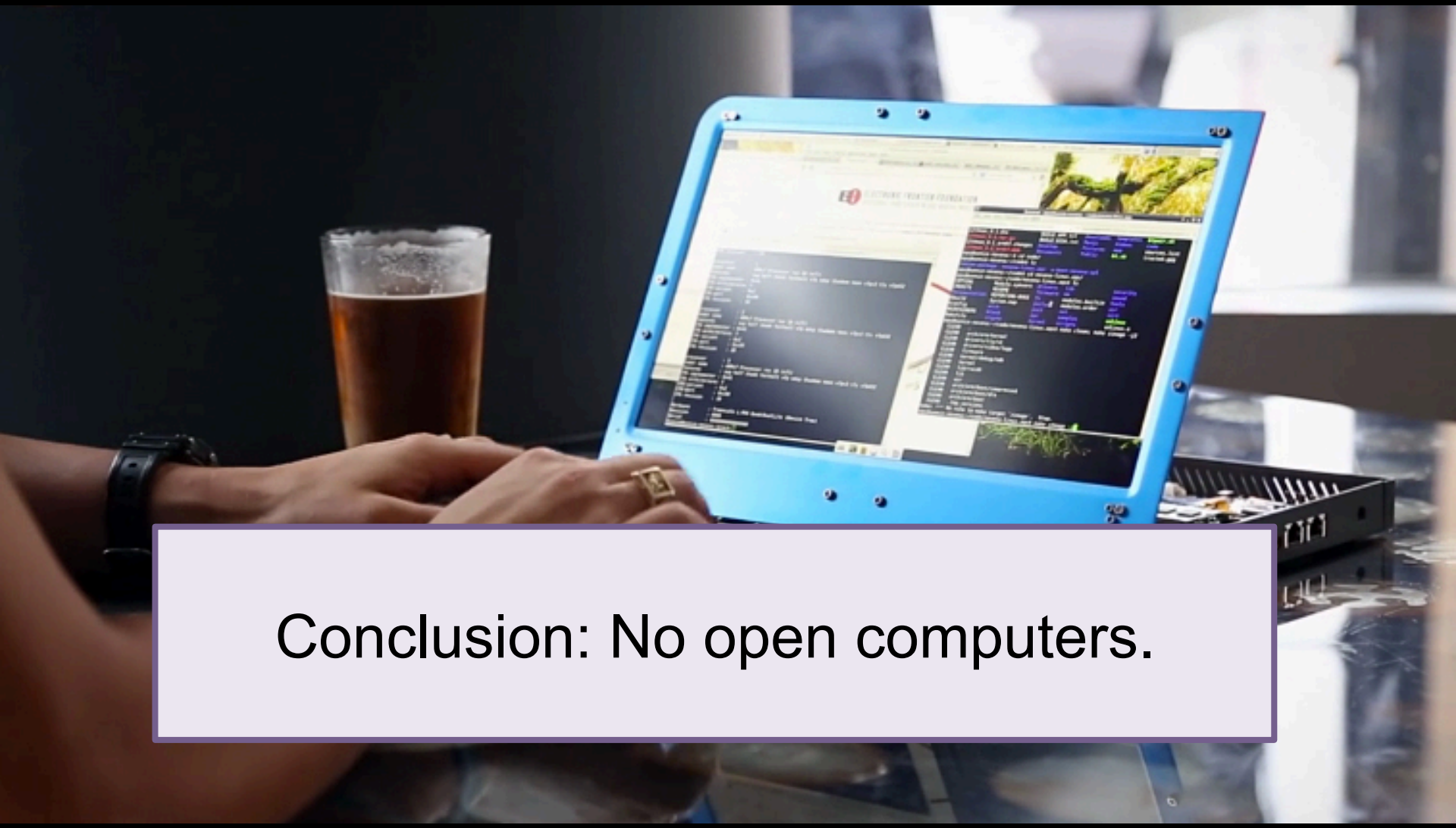

Practice #2


Practice Soln #2

Open Computer?



Open Computer?

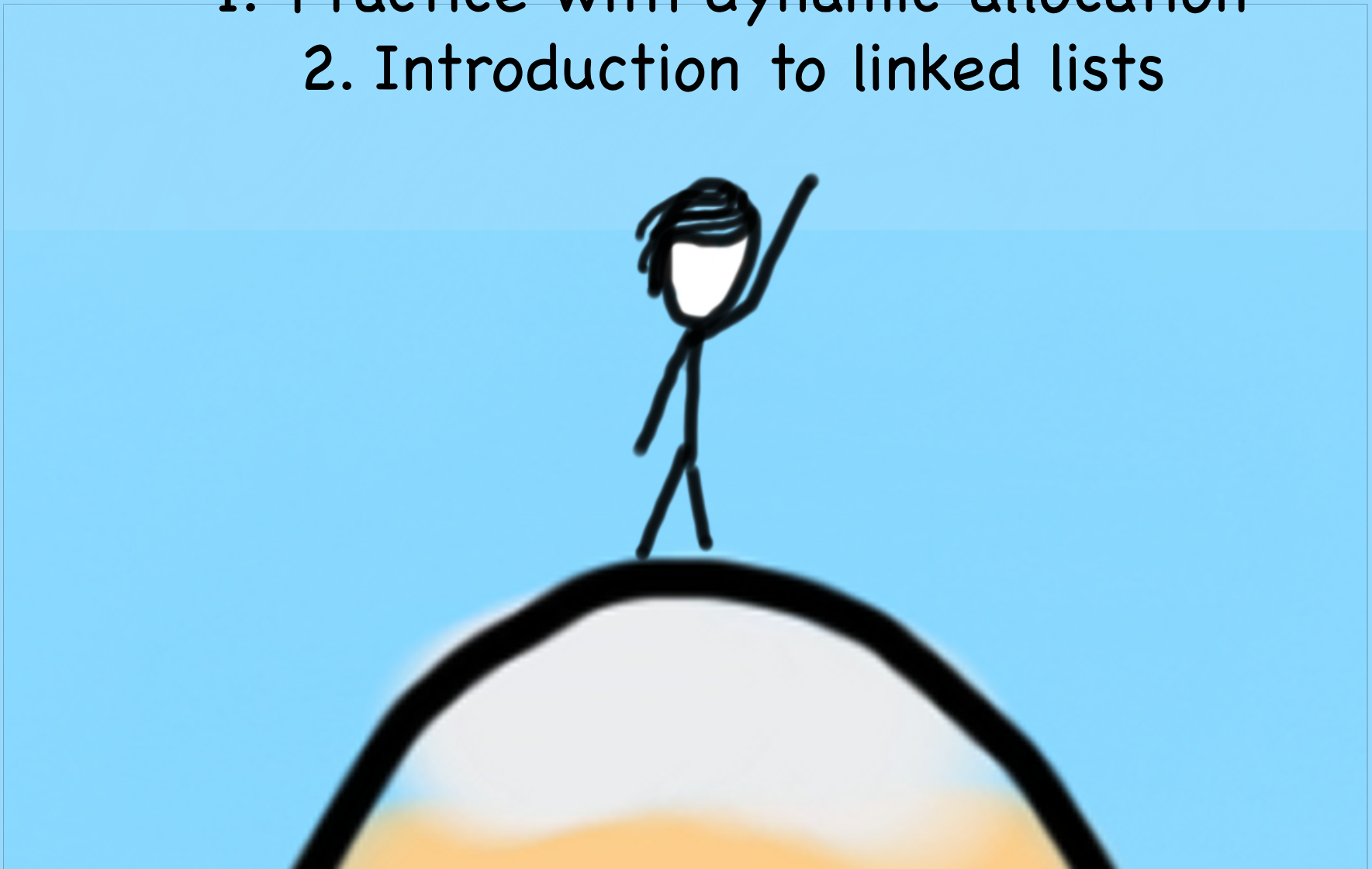


Conclusion: No open computers.

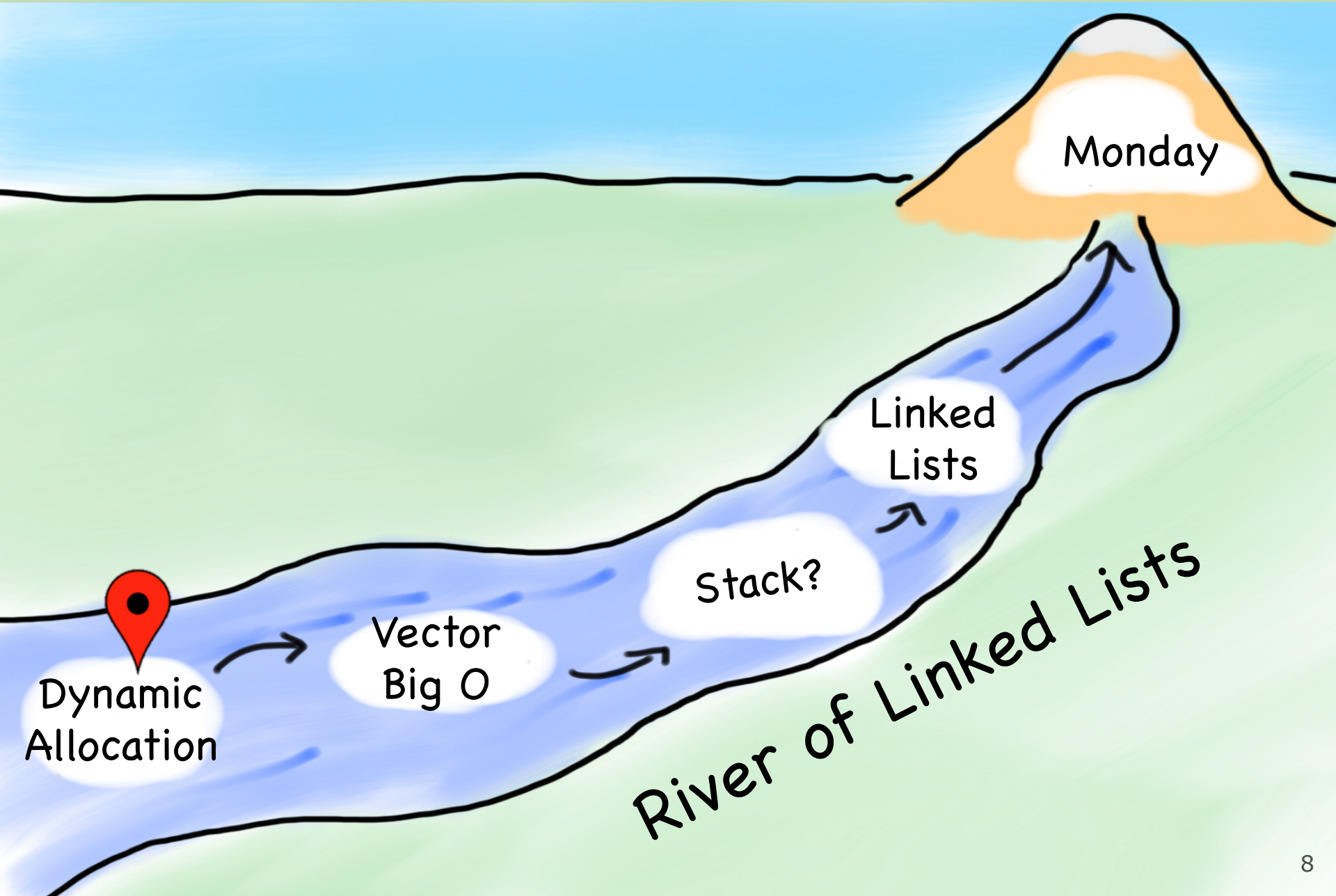
Midterm Questions?

Today's Goals

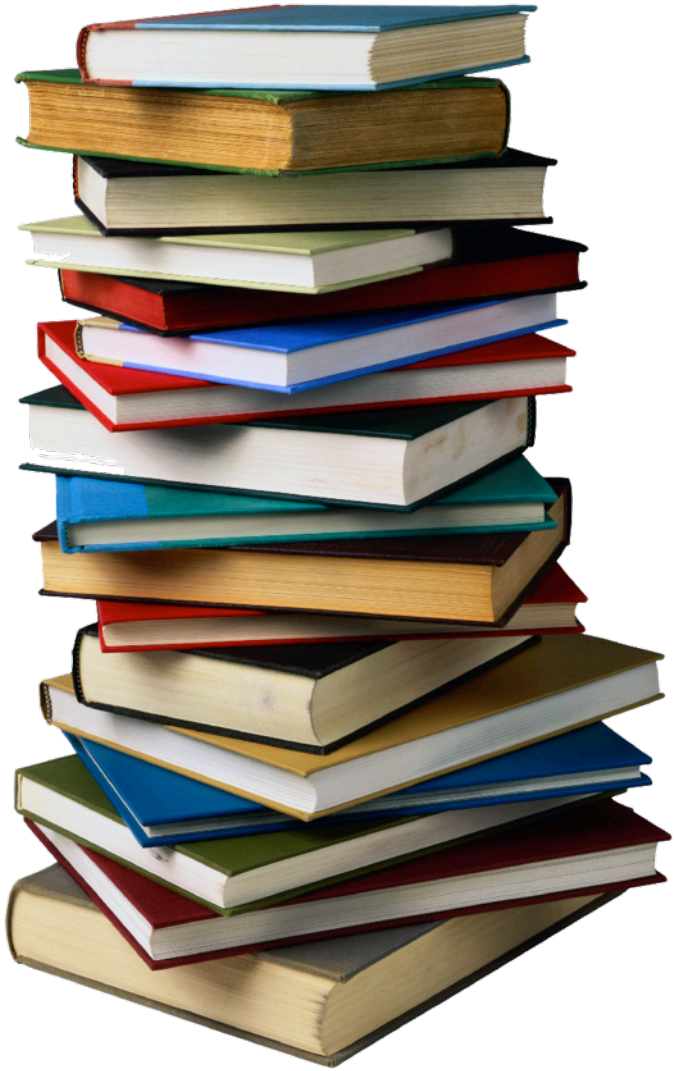
1. Practice with dynamic allocation
2. Introduction to linked lists



Today's Goals



How is the Stack Implemented?



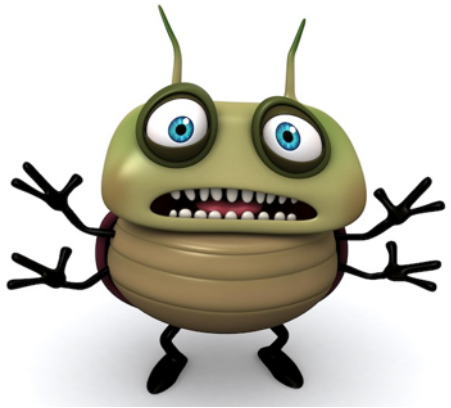
Lets Write Vector

VectorInt

```
class VectorInt {          // in VectorInt.h
public:
    VectorInt();           // constructor

    void add(int value); // append a value to the end
    int get(int index);  // return the value at index

private:
    type name;      // member variables
    type name;      // (data inside each object)
};
```

Buggy VectorInt the First

```
class VectorInt {          // in VectorInt.h
public:
    VectorInt();           // constructor

    void add(int value); // append a value to the end
    int get(int index);  // return the value at index

private:
    int value0;      // member variables
    int value1;      // (data inside each object)
};
```



Problems with Stack Variables

Variables have to be known at compile time. Not runtime.

Problems with Stack Variables

Variables have to be known at compile time. Not runtime.

Persistence is out of our control.

Its hard to share a single large object between classes.

Dynamic Allocation!

Pointers

```
// dynamically request memory for a new GImage.
// calls the constructor.
// store the address of the new GImage.

GImage * image = new GImage("cat.png");
```

Pointers

```
// dynamically request memory for a new GImage.
// calls the constructor.
// store the address of the new GImage.

GImage * image = new GImage("cat.png");
```

Pointers

```
// dynamically request memory for a new GImage.
// calls the constructor.
// store the address of the new GImage.

GImage * image = new GImage("cat.png");
```

124134

Pointers

```
// dynamically request memory for a new GImage.
// calls the constructor.
// store the address of the new GImage.

GImage * image = new GImage("cat.png");
```

124134

124134

Pointers

```
// dynamically request memory for a new GImage.
// calls the constructor.
// store the address of the new GImage.
```

124134

```
GImage * image = new GImage("cat.png");
```

124134

Pointers

```
// dynamically request memory for a new GImage.
// calls the constructor.
// store the address of the new GImage.
```

124134

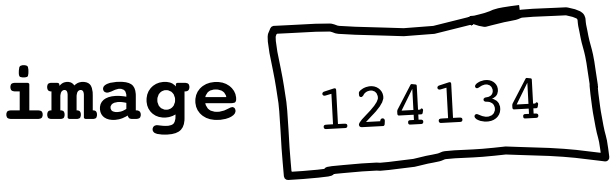
```
GImage * image = new GImage("cat.png");
```

image 124134

124134

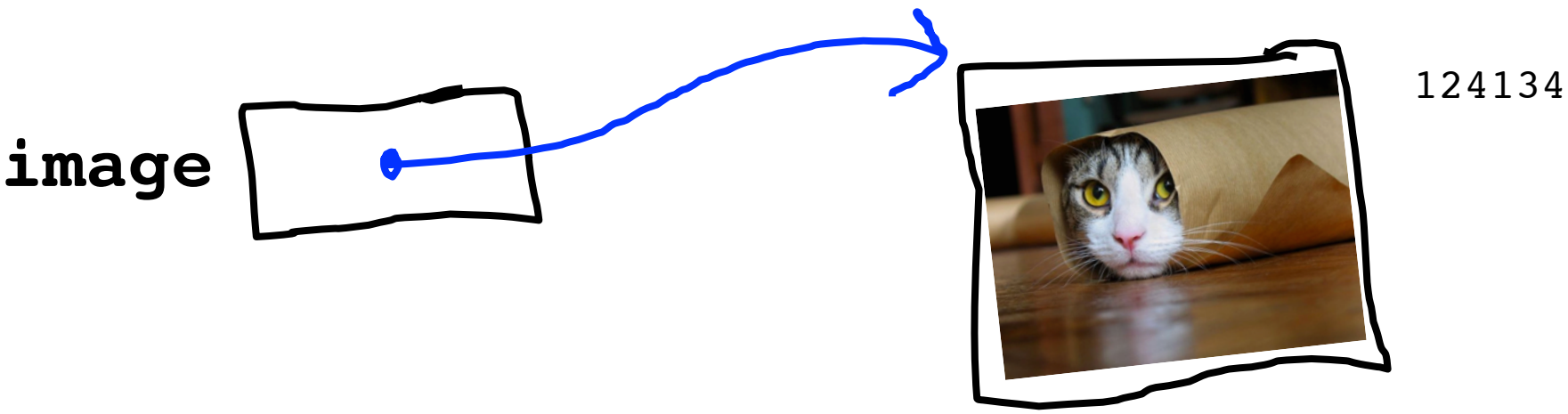
Pointers

```
// dynamically request memory for a new GImage.
// calls the constructor.
// store the address of the new GImage.

GImage * image = new GImage("cat.png");
```

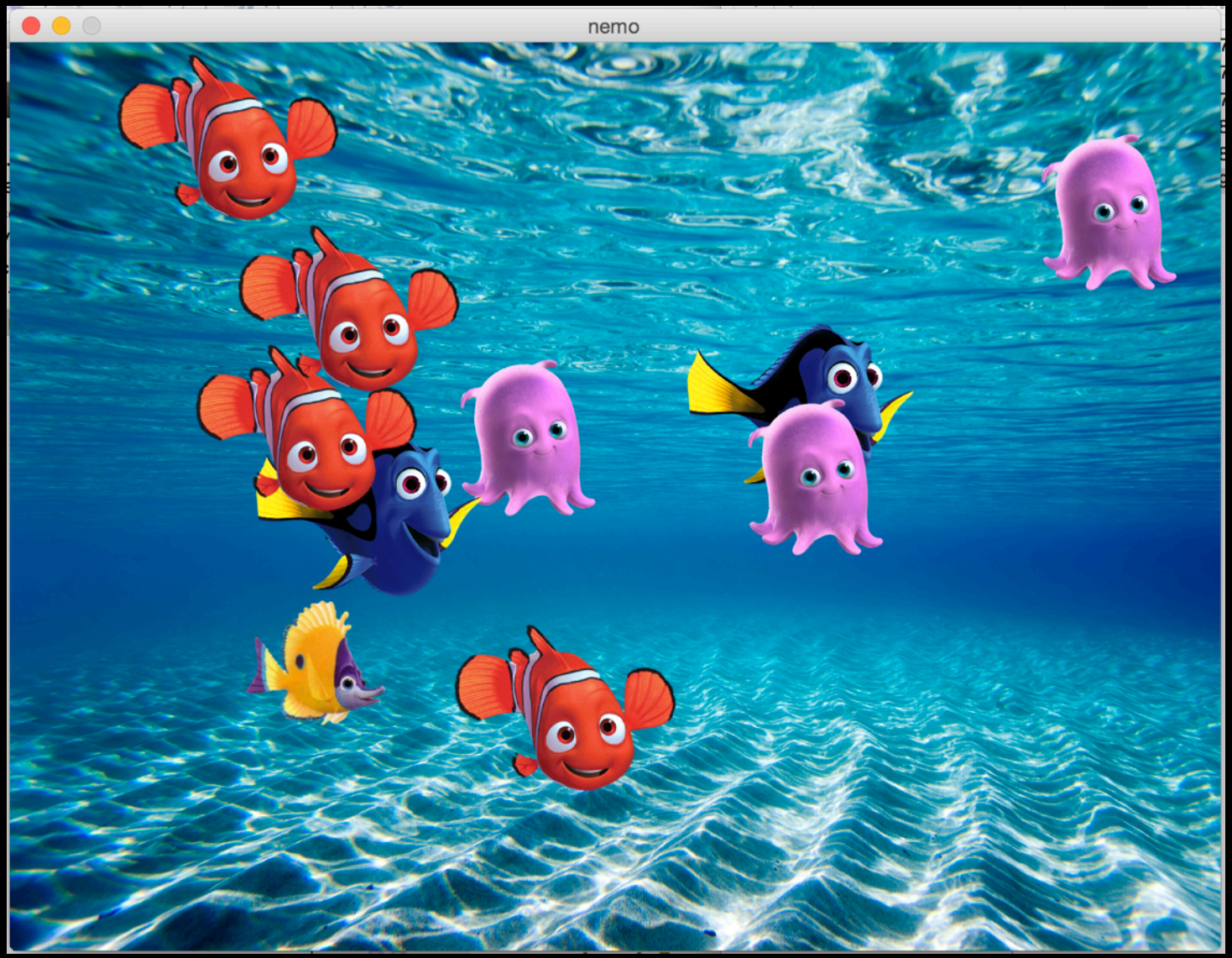


Pointers

```
// dynamically request memory for a new GImage.
// calls the constructor.
// store the address of the new GImage.

GImage * image = new GImage("cat.png");
```



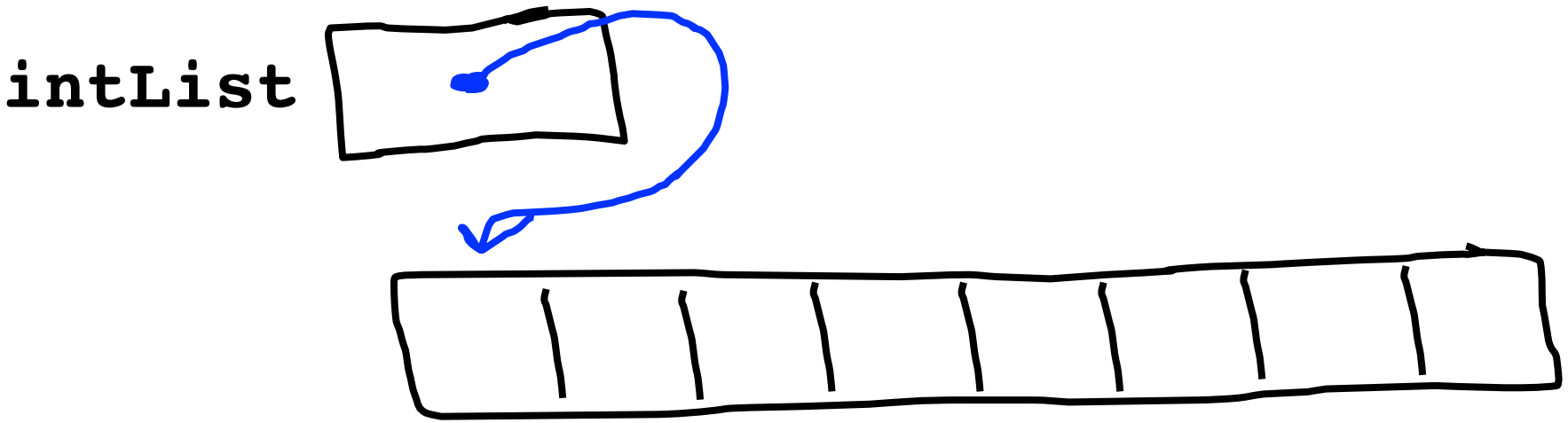
Pointers Example



Pointers

```
// dynamically request memory for n integers.
// store the address of the provided ints.
```

```
int * intList = new int[n];
```



How Does this Help?

Variables have to be known at compile time. Not runtime.

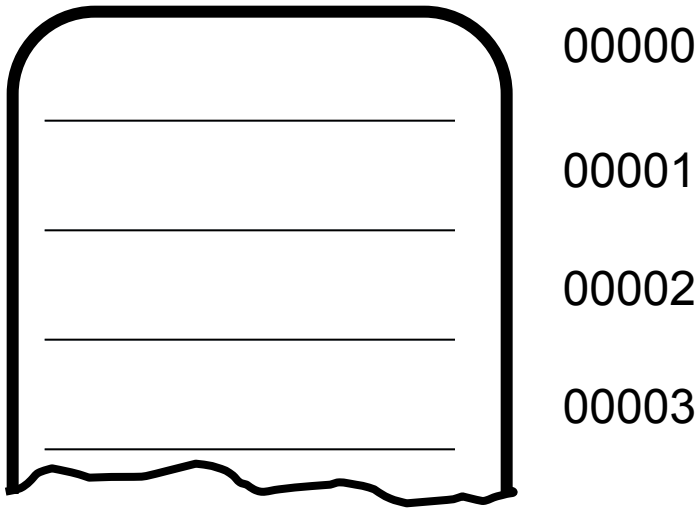
Persistence is out of our control.

Its hard to share a single large object between classes.

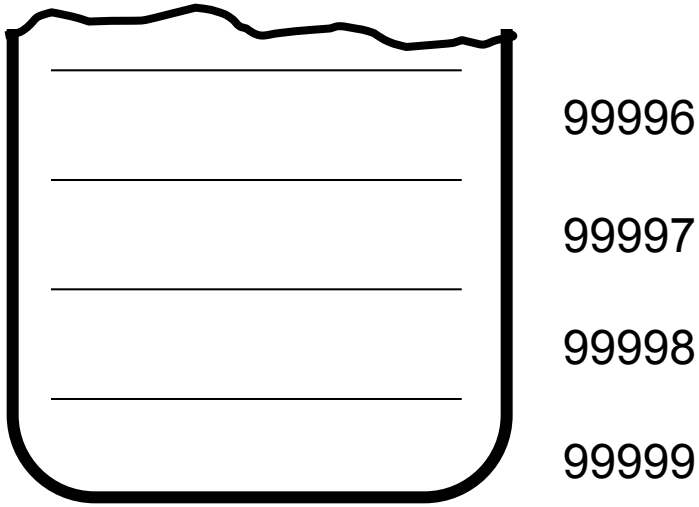
Dig deeper

All Memory Has an Address

RAM not disk



Each program gets it's own



URL Metaphore

A pointer is like a URL. Not the actual page, the address of the page

Socrative

```
Point * megan = new Point();
megan->setX(10);
Point * student = megan;
student->setY(7);

cout << student->getX();
cout << " ";
cout << student->getY();
```

a) 10, 7

b) crashes

c) ?, 7

d) ? ?

Socrative

```
Point * megan = new Point();
megan->setX(10);
Point * student = megan;
student->setY(7);

cout << student->getX();
cout << " ";
cout << student->getY();
```

a) 10, 7

b) crashes

c) ?, 7

d) ? ?

Socrative

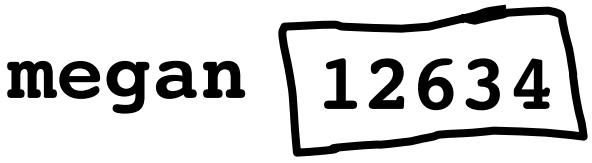
```
Point * megan = new Point();
megan->setX(10);
Point * student = megan;
student->setY(7);

cout << student->getX();
cout << " ";
cout << student->getY();
```

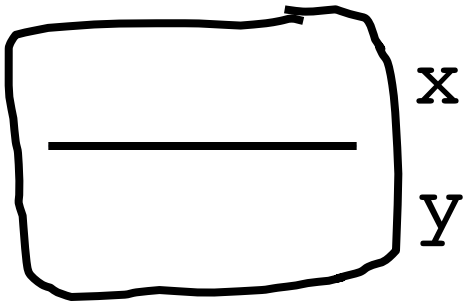
a) 10, 7

b) crashes

c) ?, 7

d) ? ?





12634

Socrative

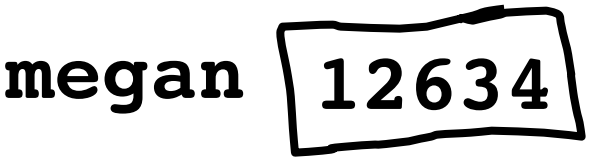
```
Point * megan = new Point();
megan->setX(10);
Point * student = megan;
student->setY(7);

cout << student->getX();
cout << " ";
cout << student->getY();
```

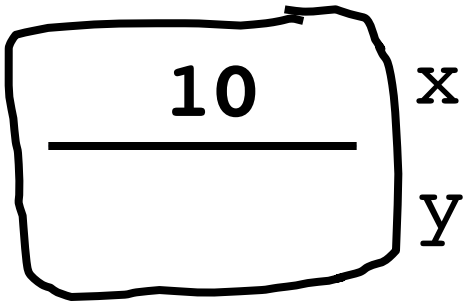
a) 10, 7

b) crashes

c) ?, 7

d) ? ?





12634

Socrative

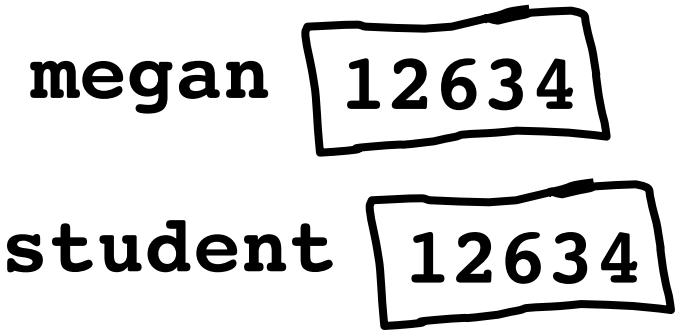
```
Point * megan = new Point();
megan->setX(10);
Point * student = megan;
student->setY(7);

cout << student->getX();
cout << " ";
cout << student->getY();
```

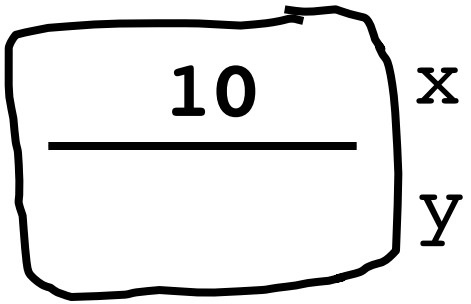
a) 10, 7

b) crashes

c) ?, 7

d) ? ?





12634

Socrative

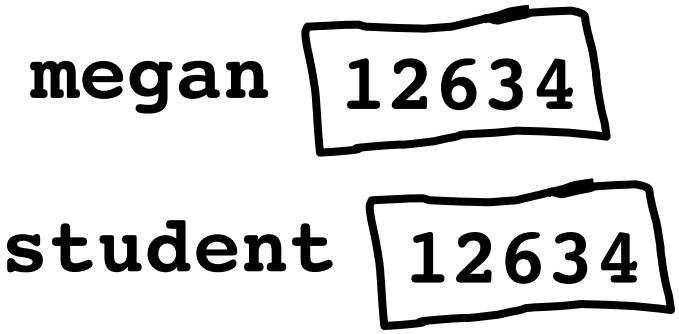
```
Point * megan = new Point();
megan->setX(10);
Point * student = megan;
student->setY(7);

cout << student->getX();
cout << " ";
cout << student->getY();
```

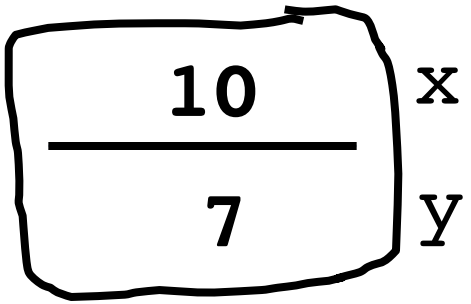
a) 10, 7

b) crashes

c) ?, 7

d) ? ?





12634

Socrative

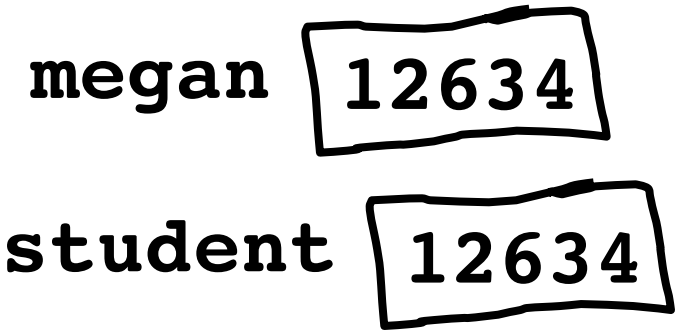
```
Point * megan = new Point();
megan->setX(10);
Point * student = megan;
student->setY(7);

cout << student->getX();
cout << " ";
cout << student->getY();
```

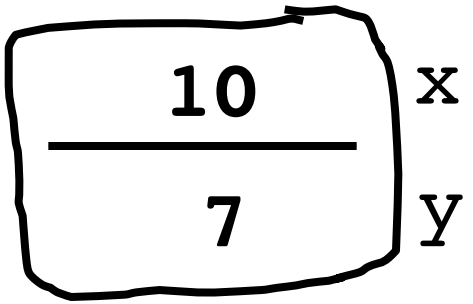
a) 10, 7

b) crashes

c) ?, 7

d) ? ?





12634

Socrative

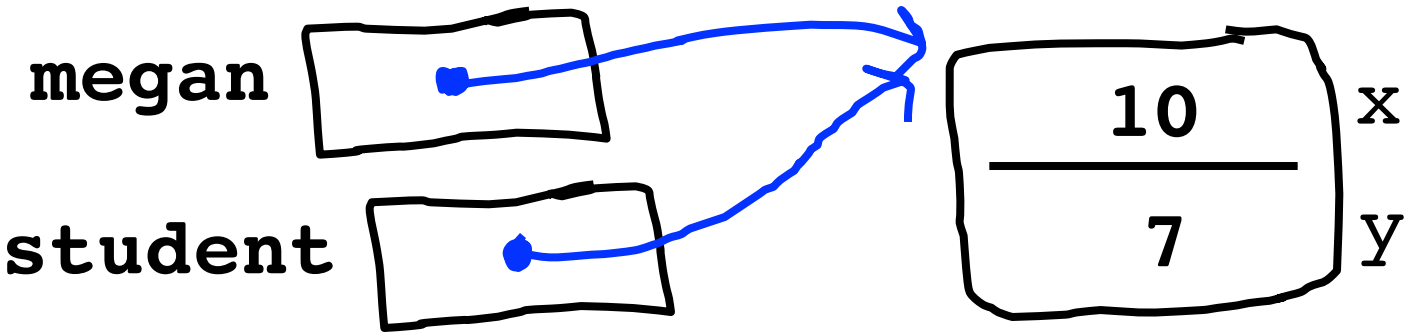
```
Point * megan = new Point();
megan->setX(10);
Point * student = megan;
student->setY(7);

cout << student->getX();
cout << " ";
cout << student->getY();
```

a) 10, 7

b) crashes

c) ?, 7

d) ? ?



12634

VectorInt

```
class VectorInt {          // in VectorInt.h
public:
    VectorInt();           // constructor

    void add(int value); // append a value to the end
    int get(int index);  // return the value at index

private:
    type name;      // member variables
    type name;      // (data inside each object)
};
```

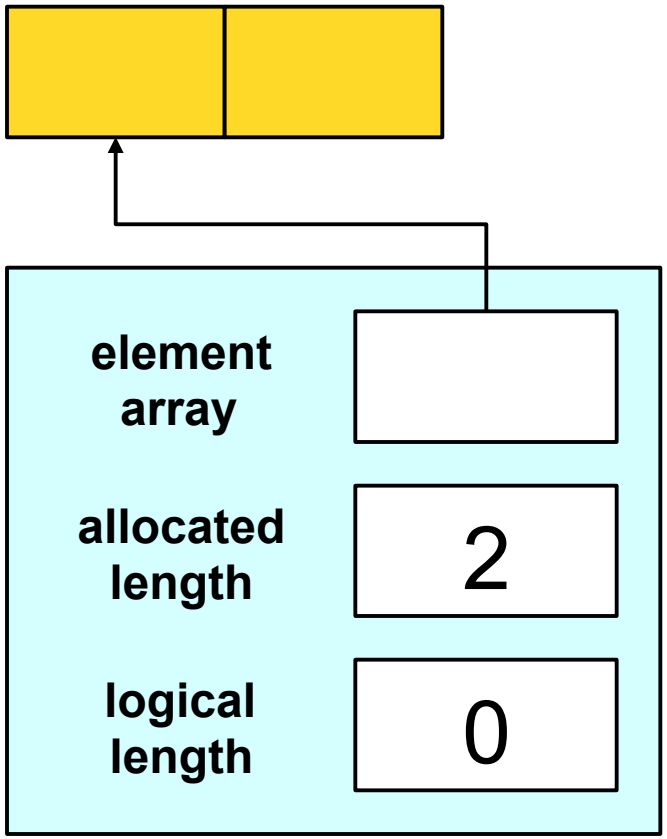
Actual Vector

```
class VectorInt {          // in VectorInt.h
public:
    VectorInt();           // constructor

    void add(int value); // append a value to the end
    int get(int index);  // return the value at index

private:
    int * data;
};
```

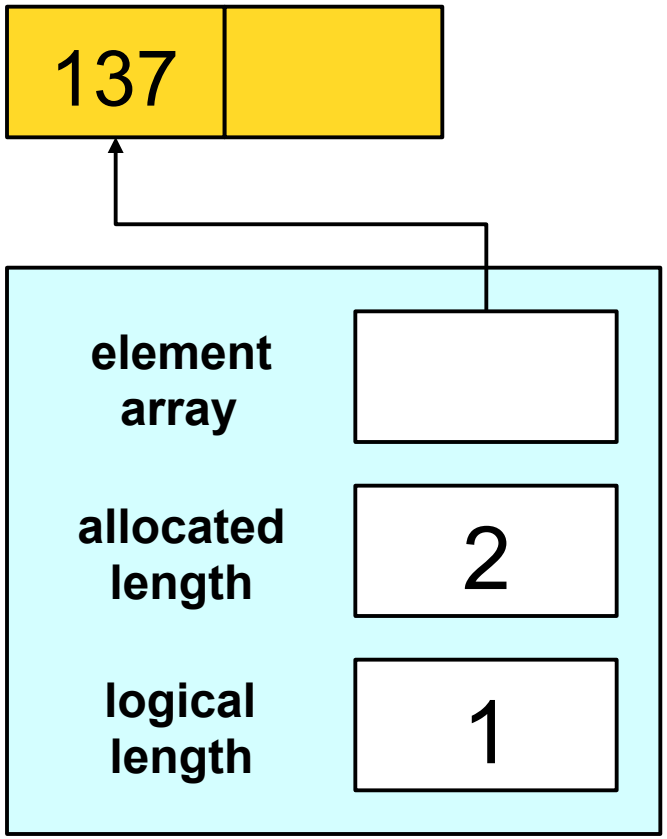
Actual Vector

```
class VectorInt {          // in VectorInt.h
public:
    VectorInt();           // constructor

    void add(int value); // append a value to the end
    int get(int index);  // return the value at index

private:
    int * data;
    int size;
    int allocatedSize;
};
```

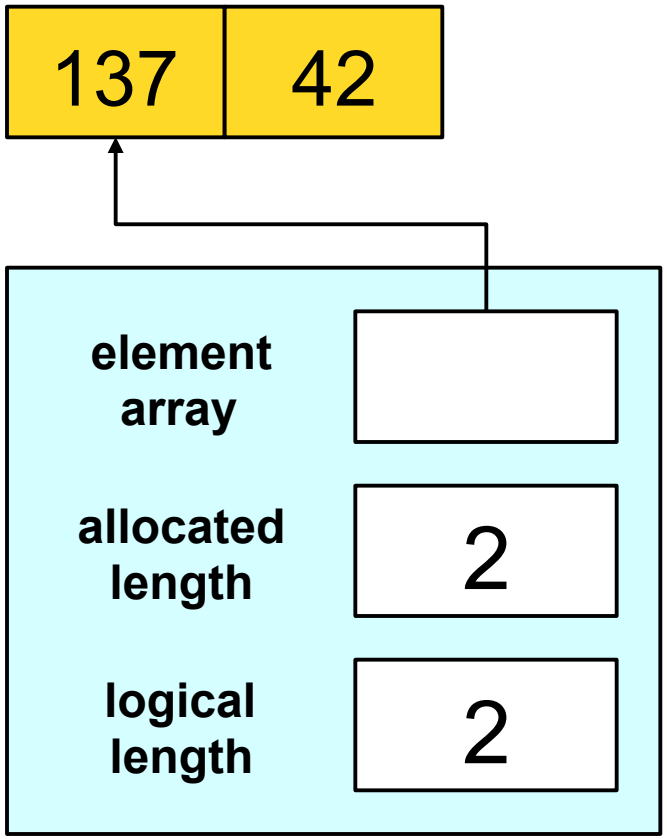
Actual Vector



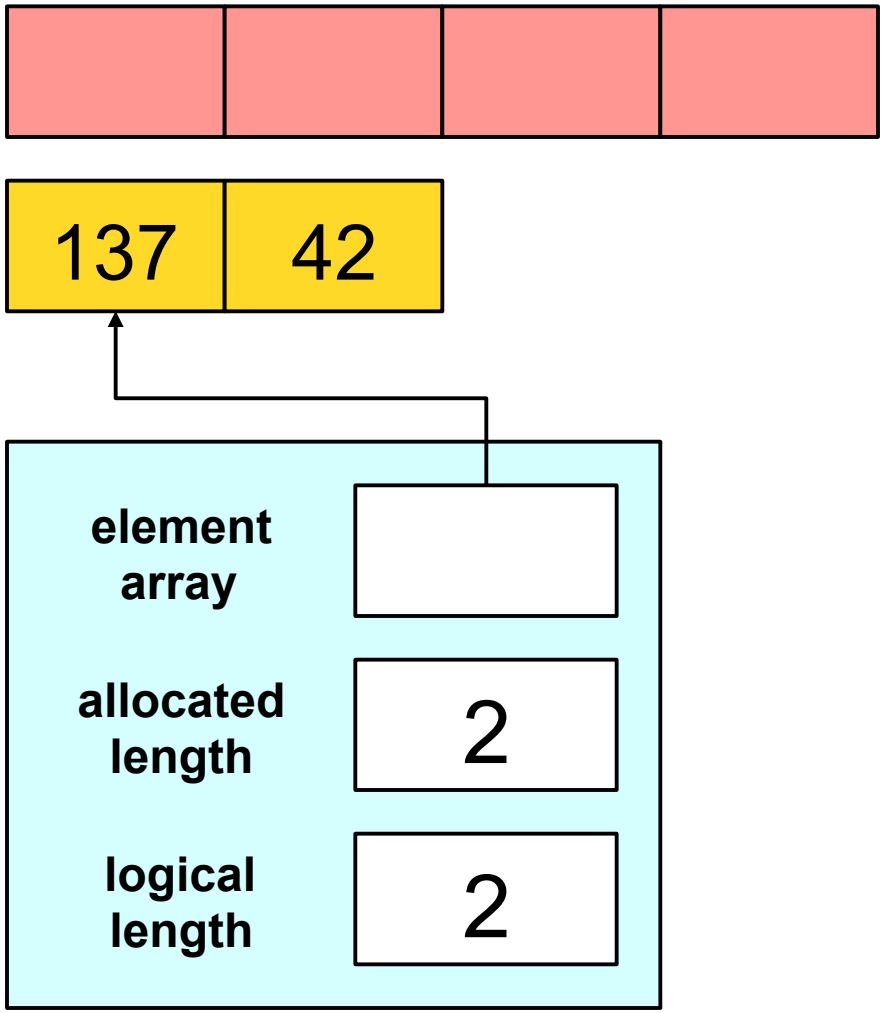
Actual Vector



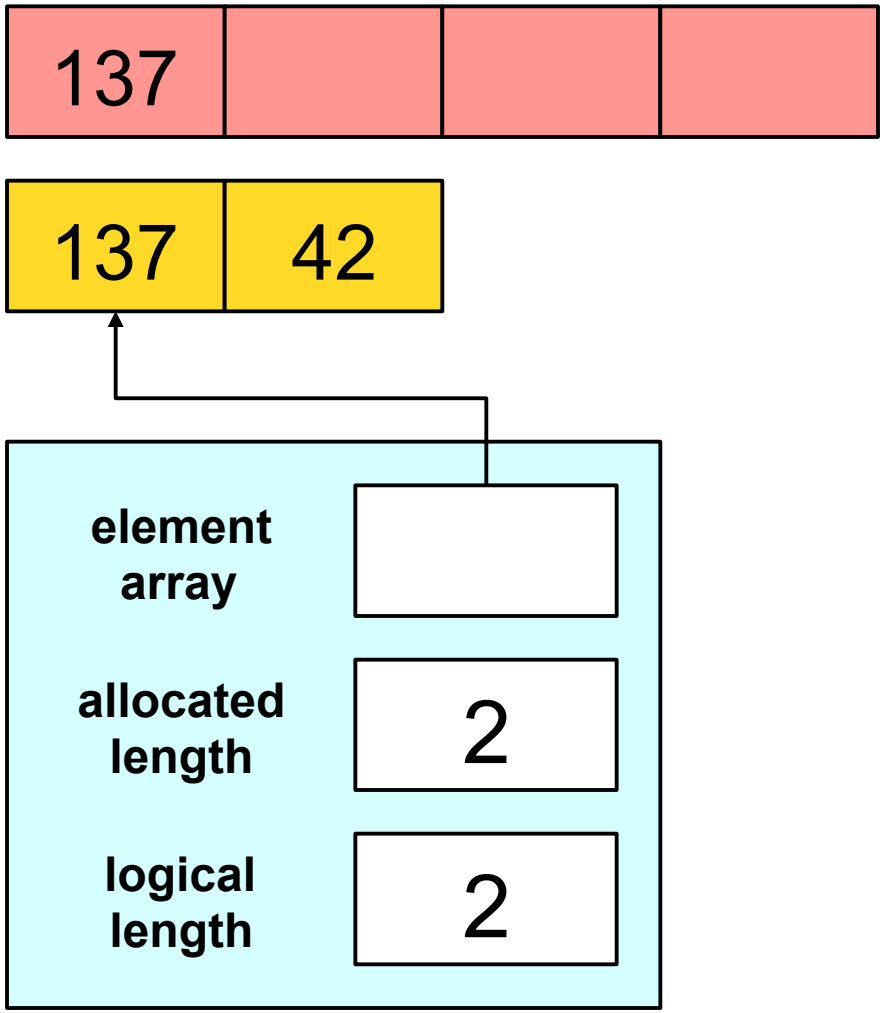
Actual Vector



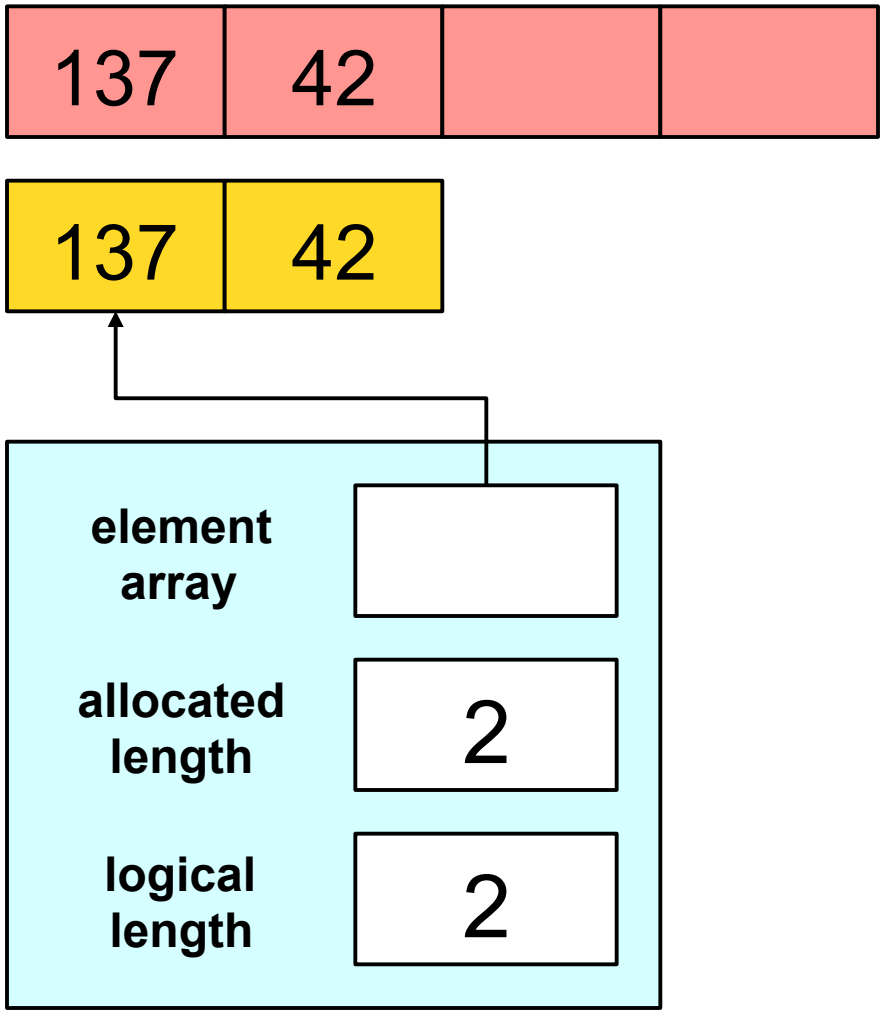
Actual Vector



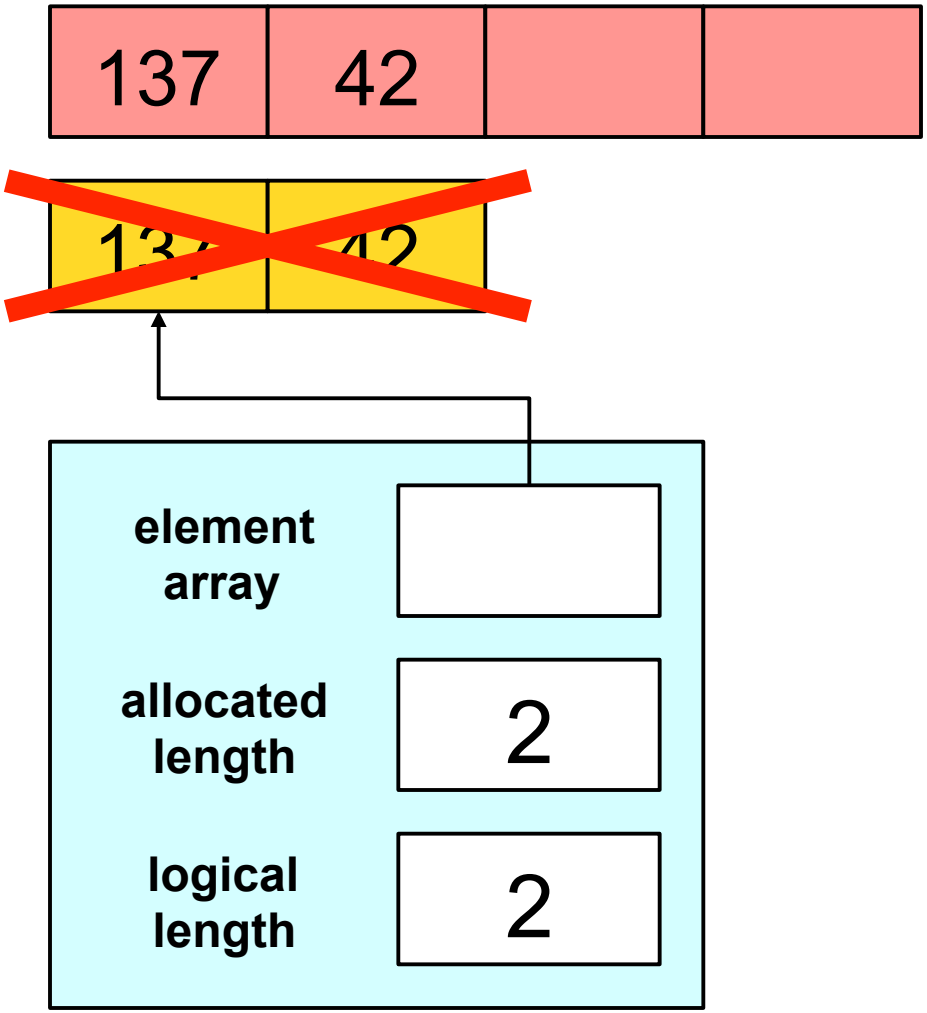
Actual Vector



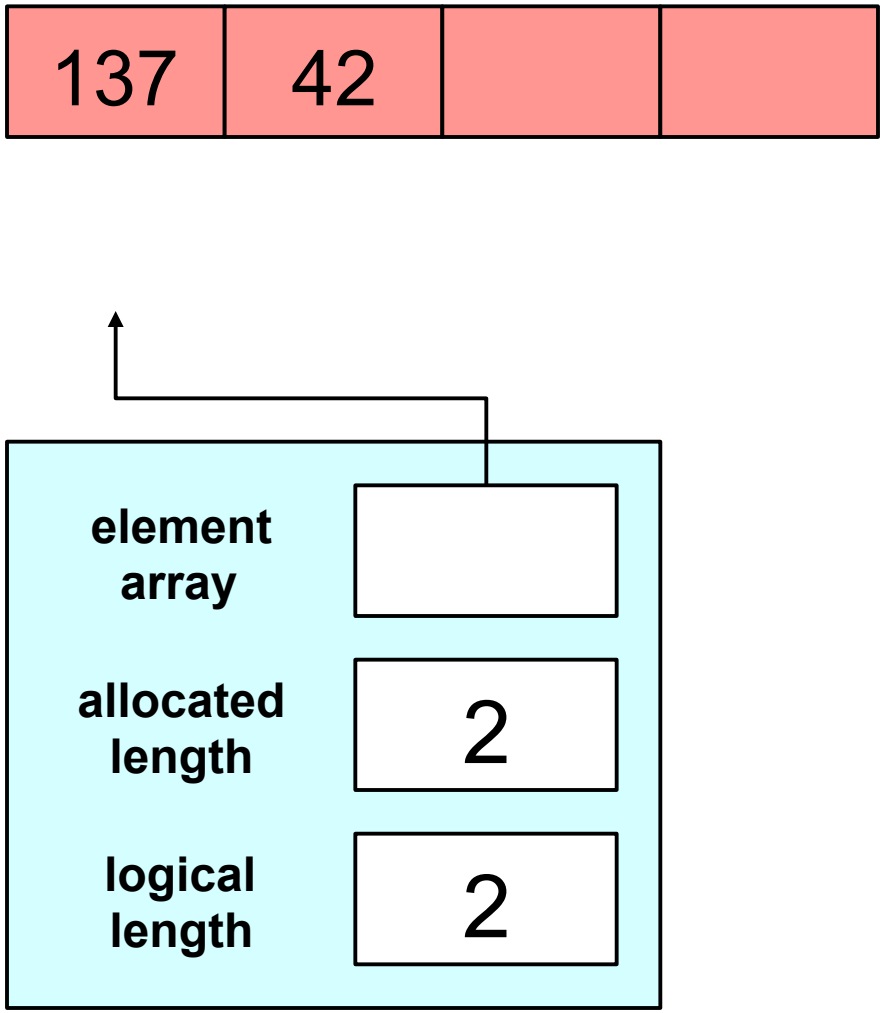
Actual Vector



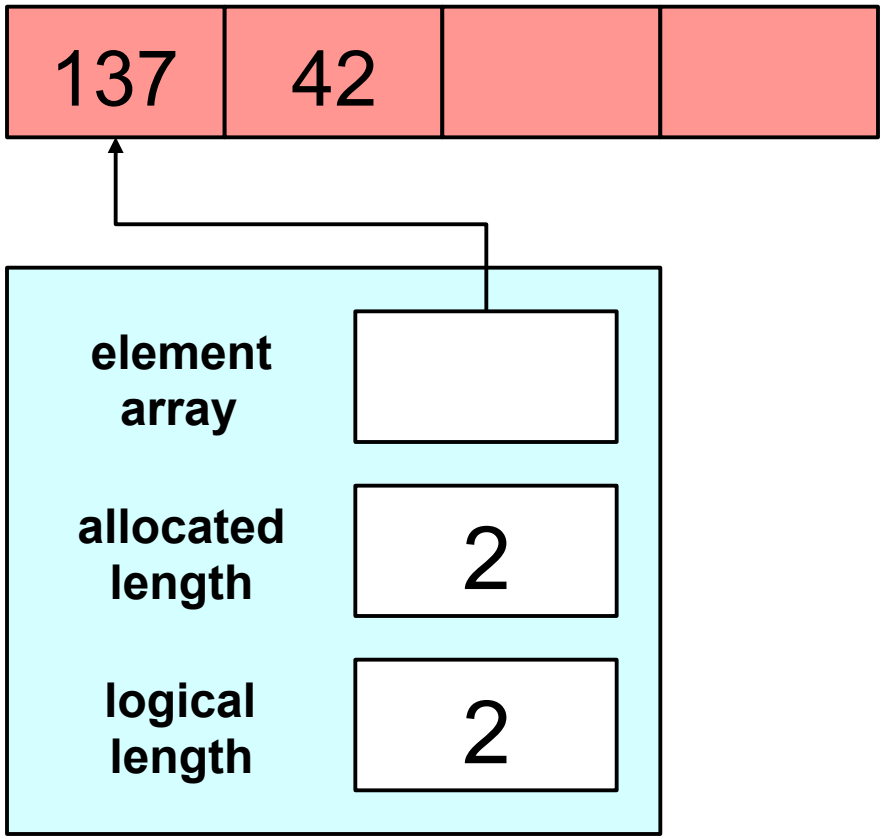
Actual Vector



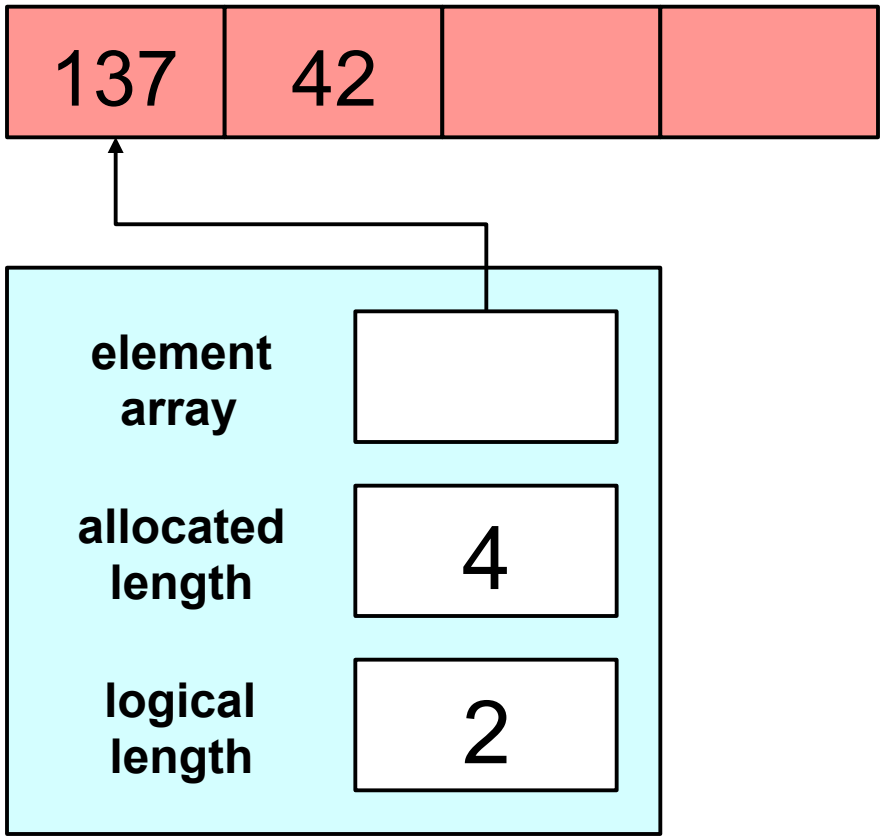
Actual Vector



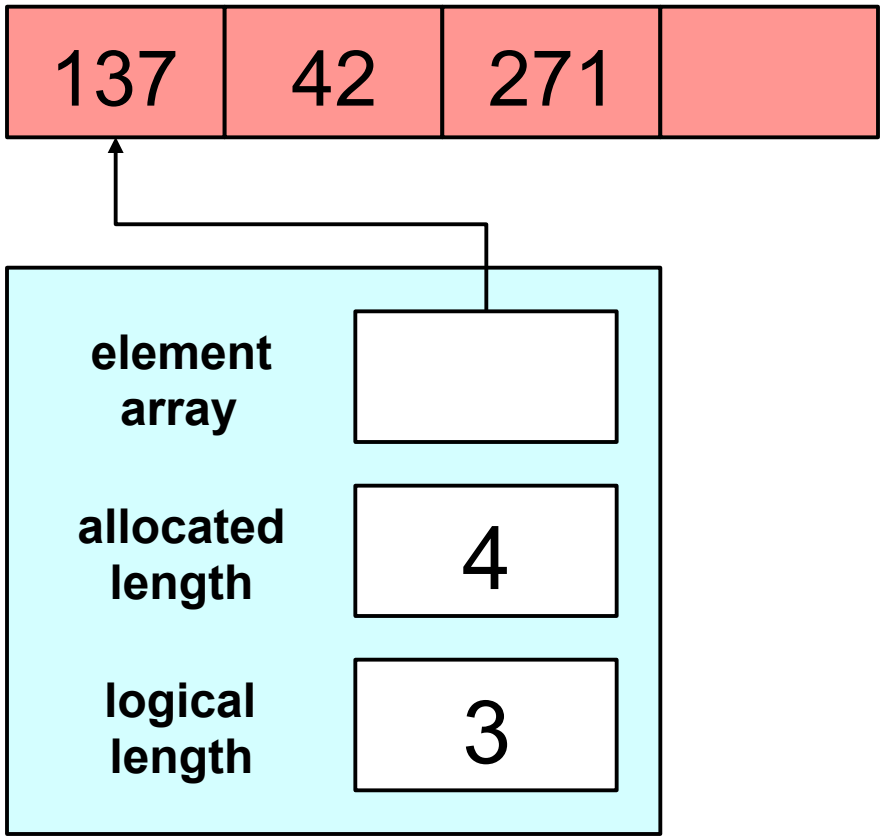
Actual Vector



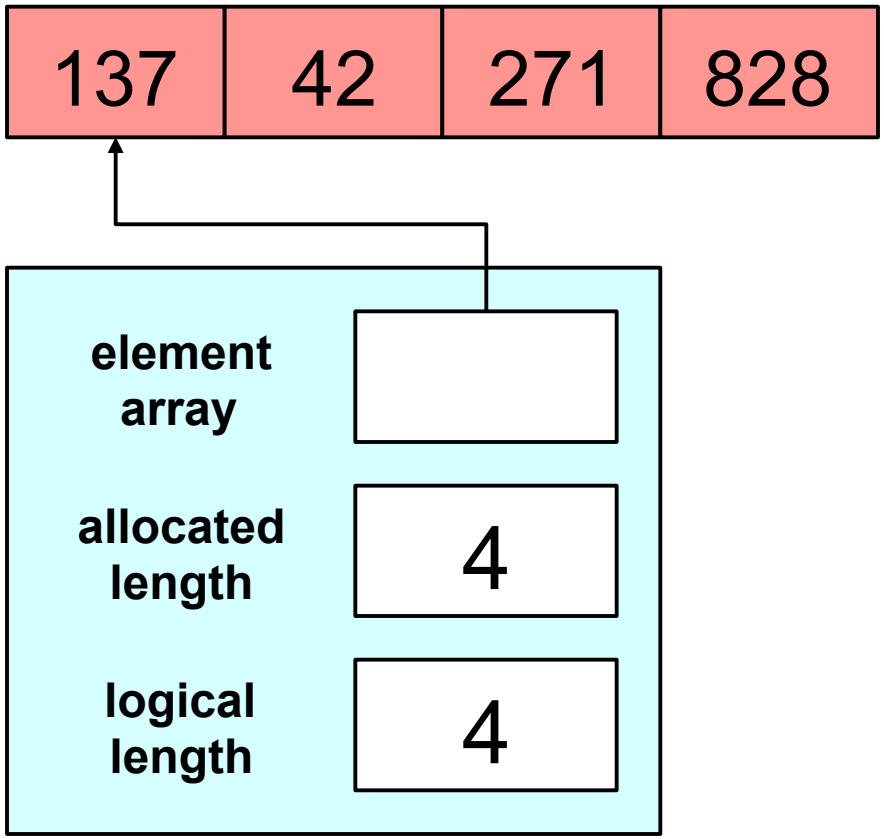
Actual Vector



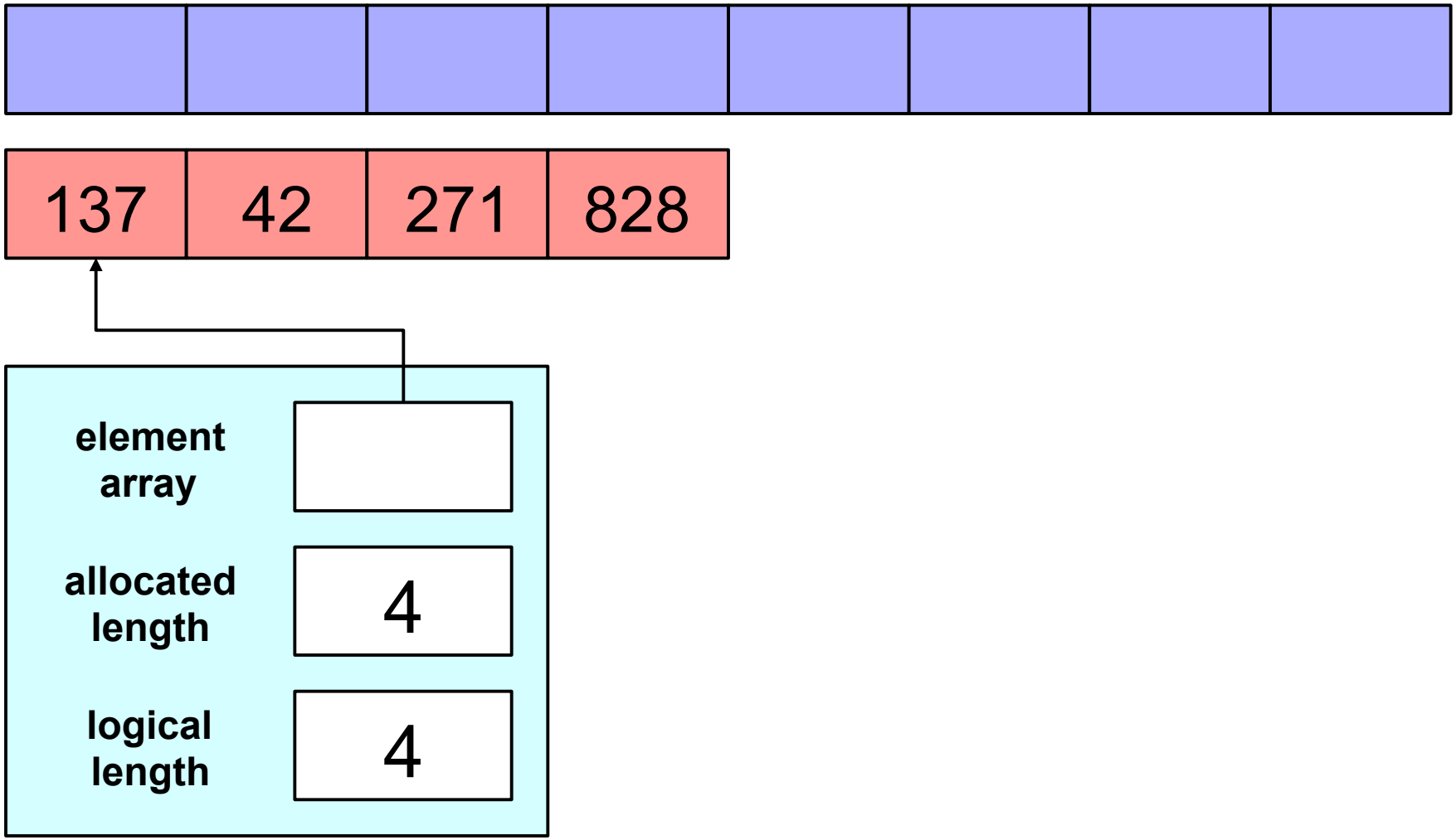
Actual Vector



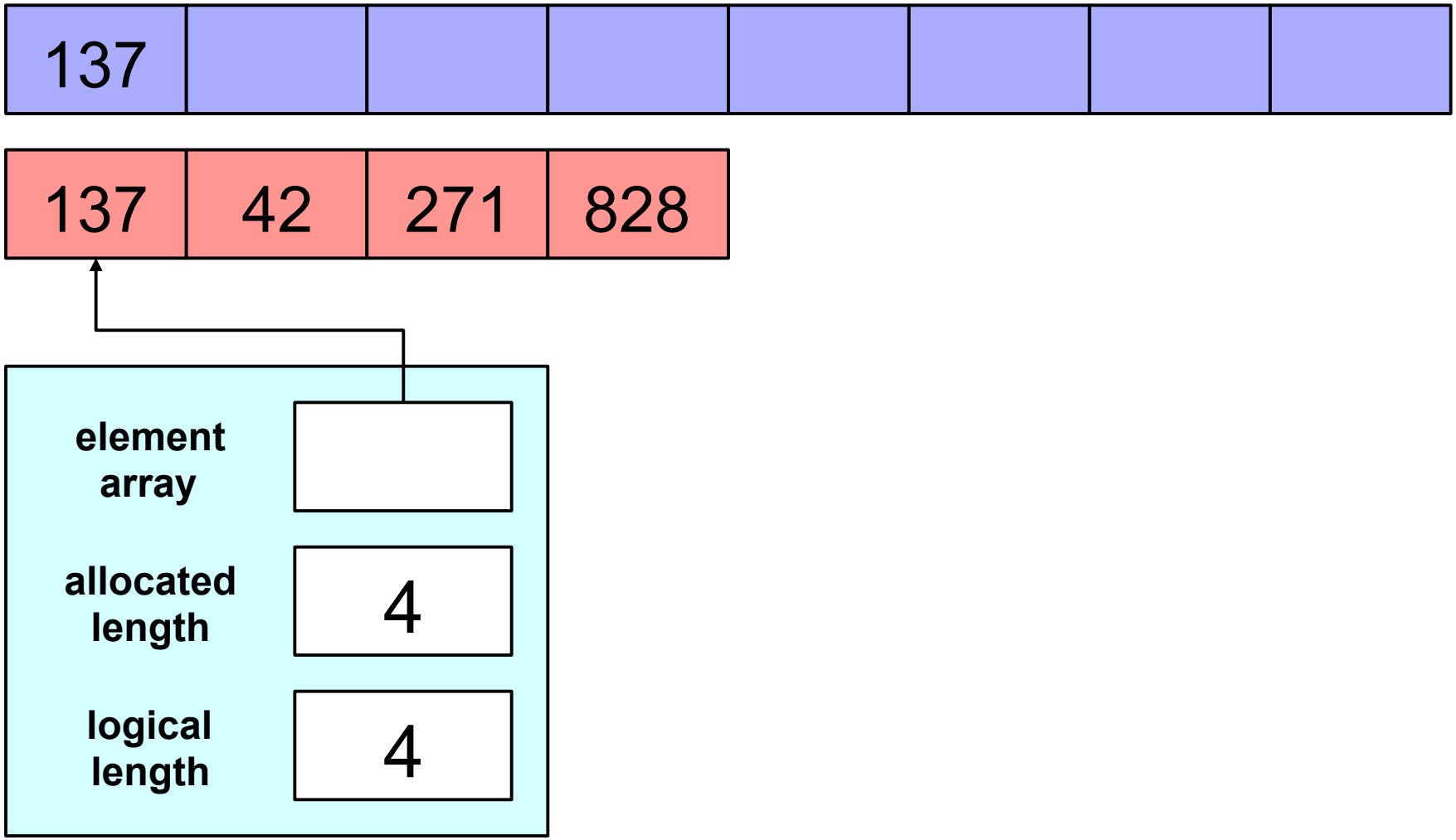
Actual Vector



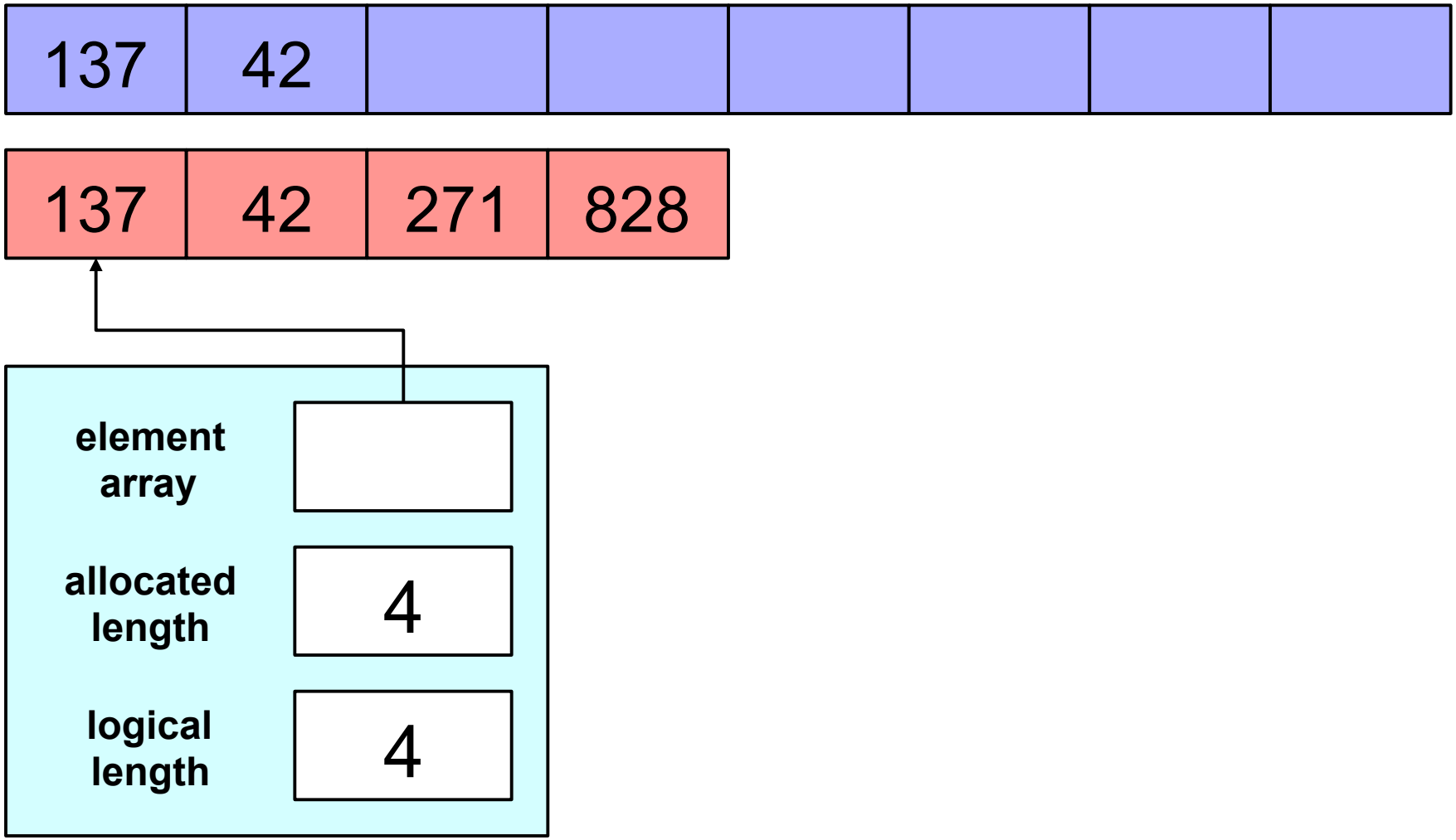
Actual Vector



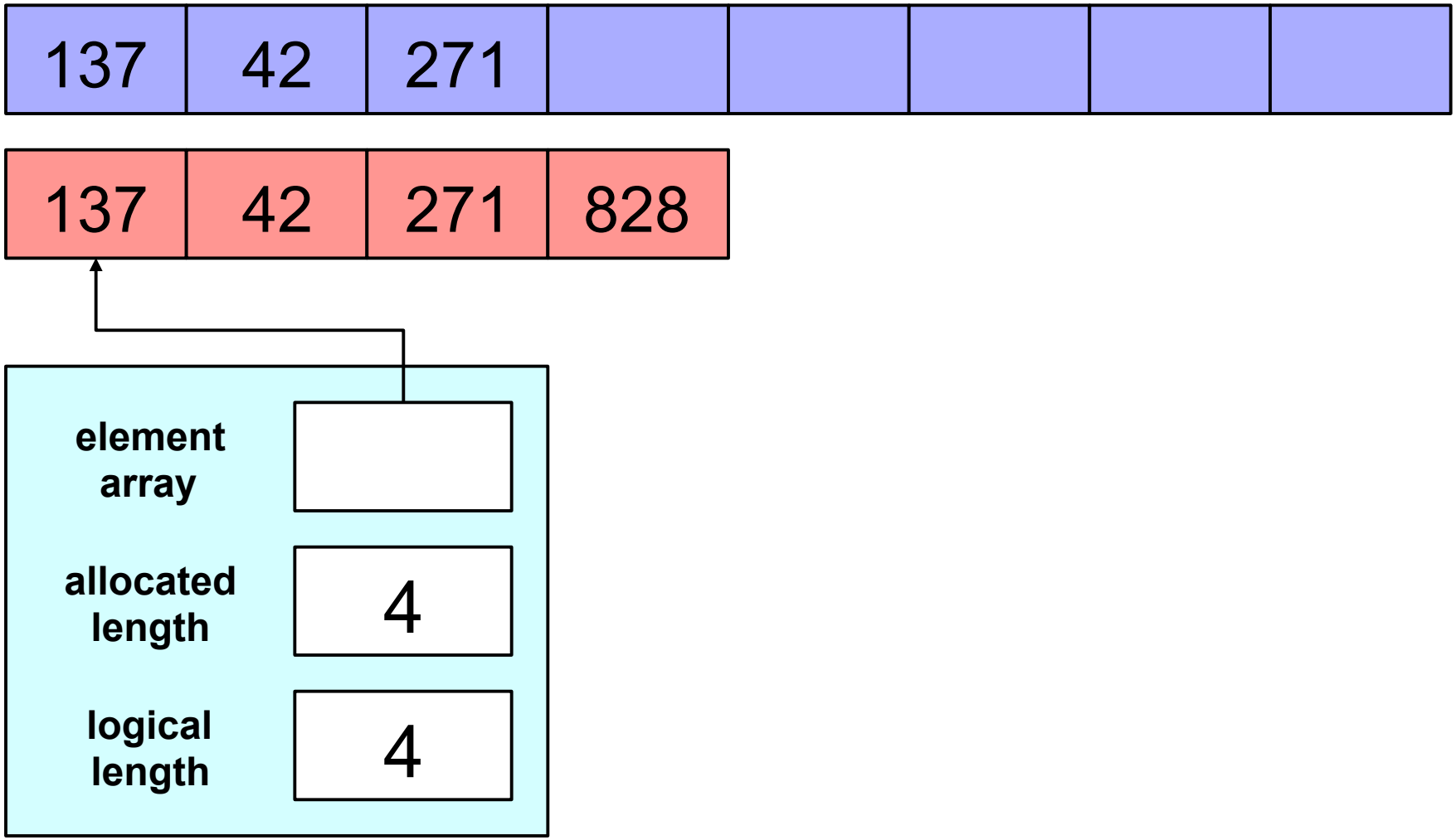
Actual Vector



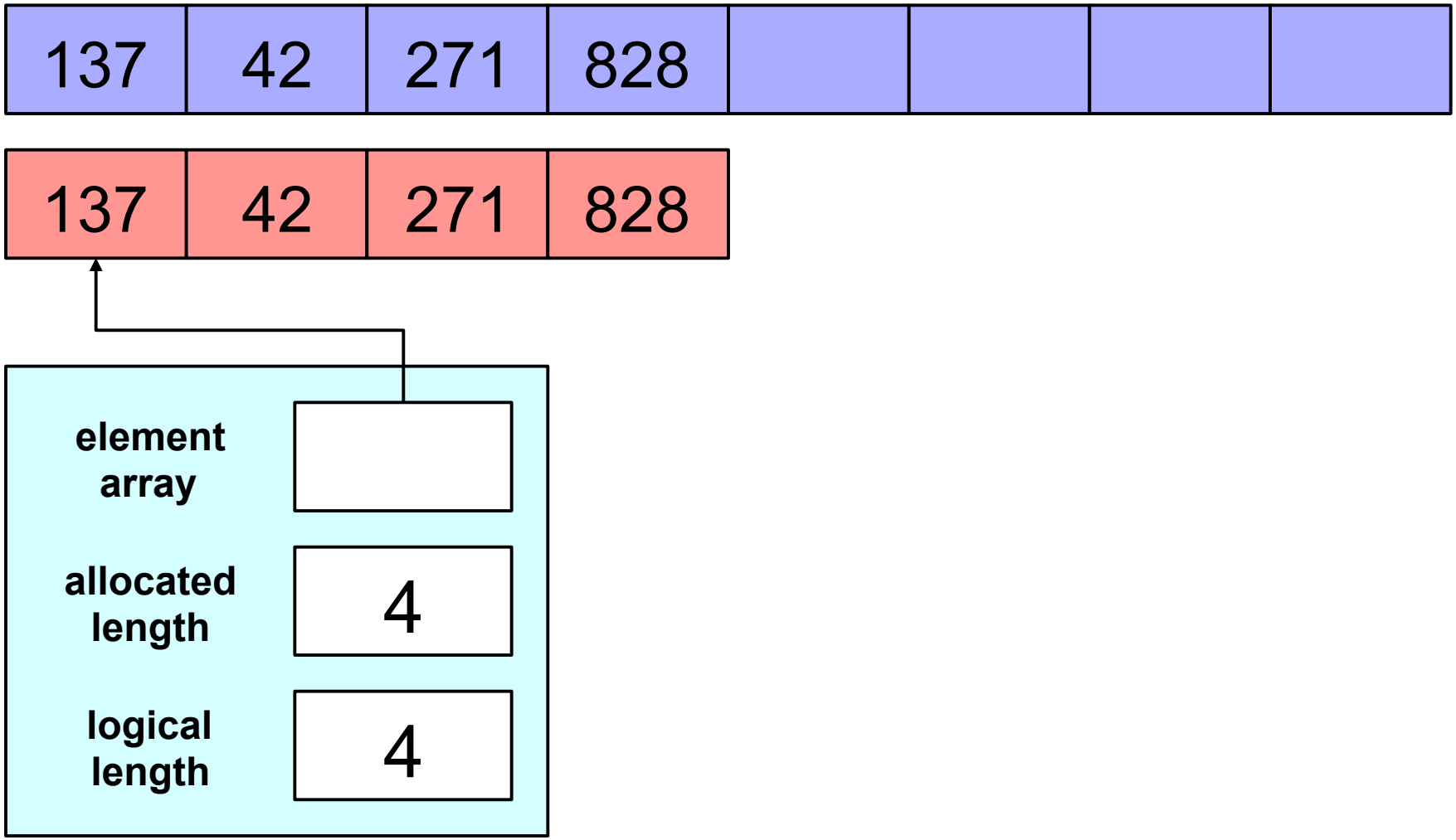
Actual Vector



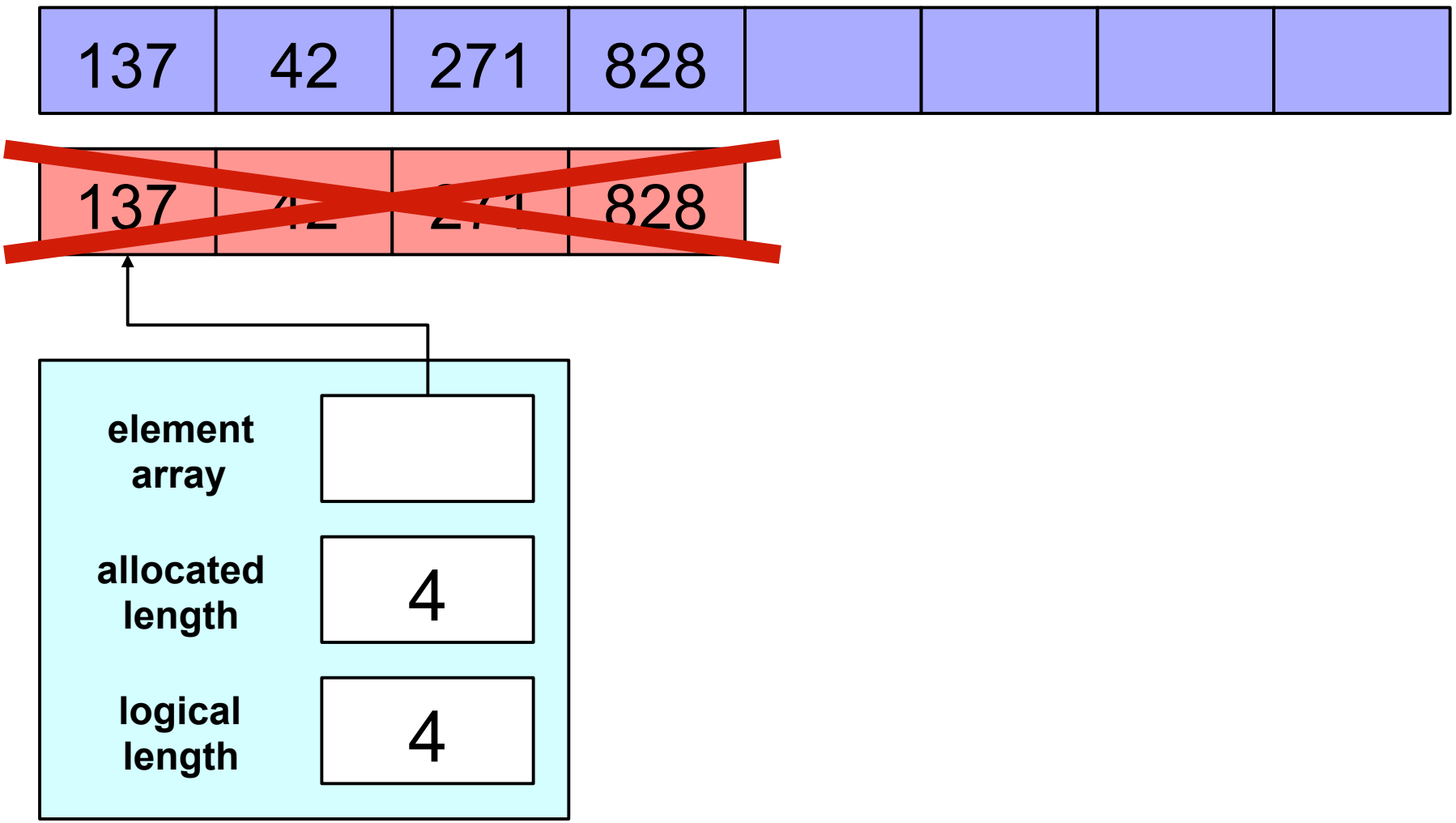
Actual Vector



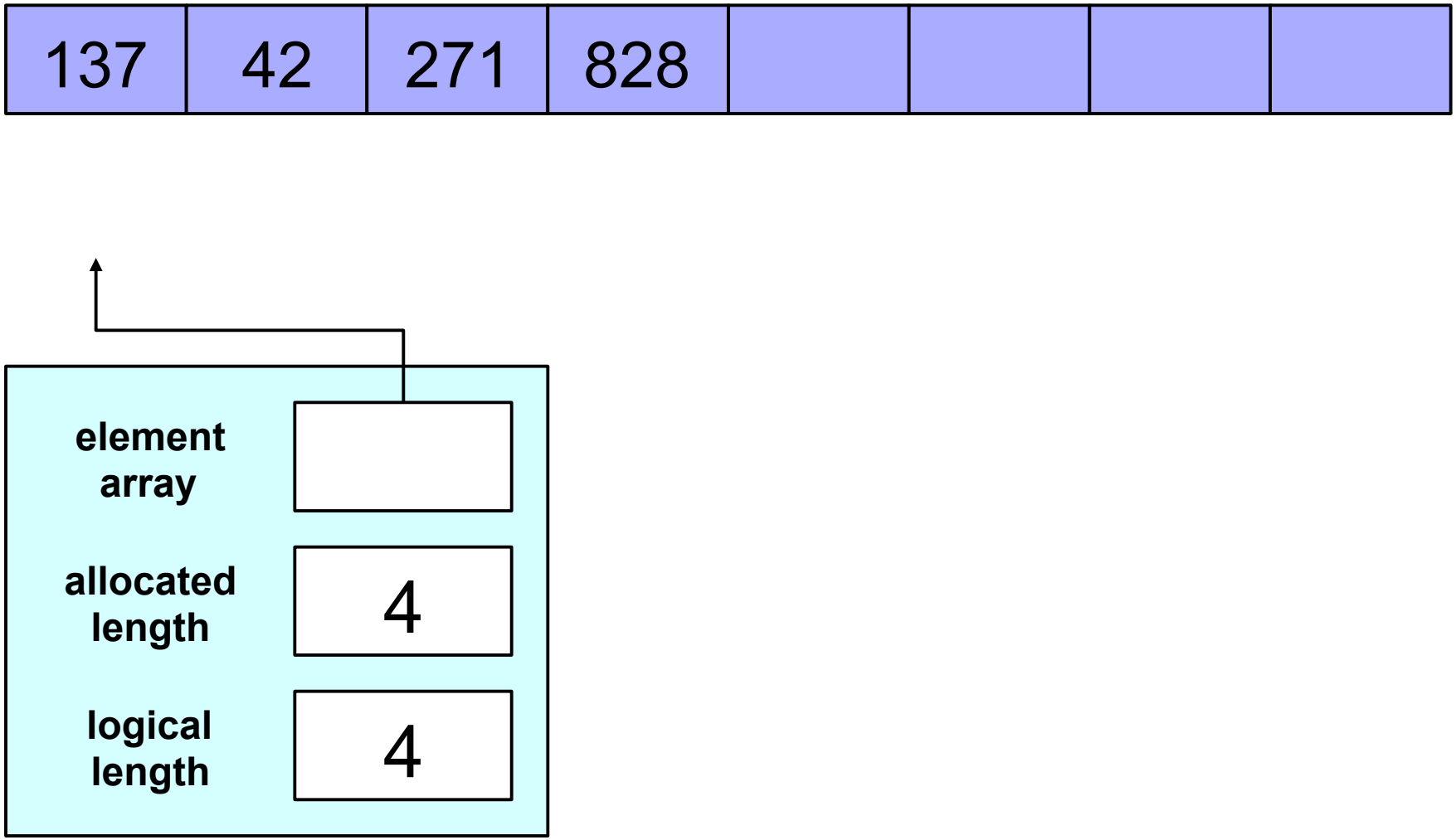
Actual Vector



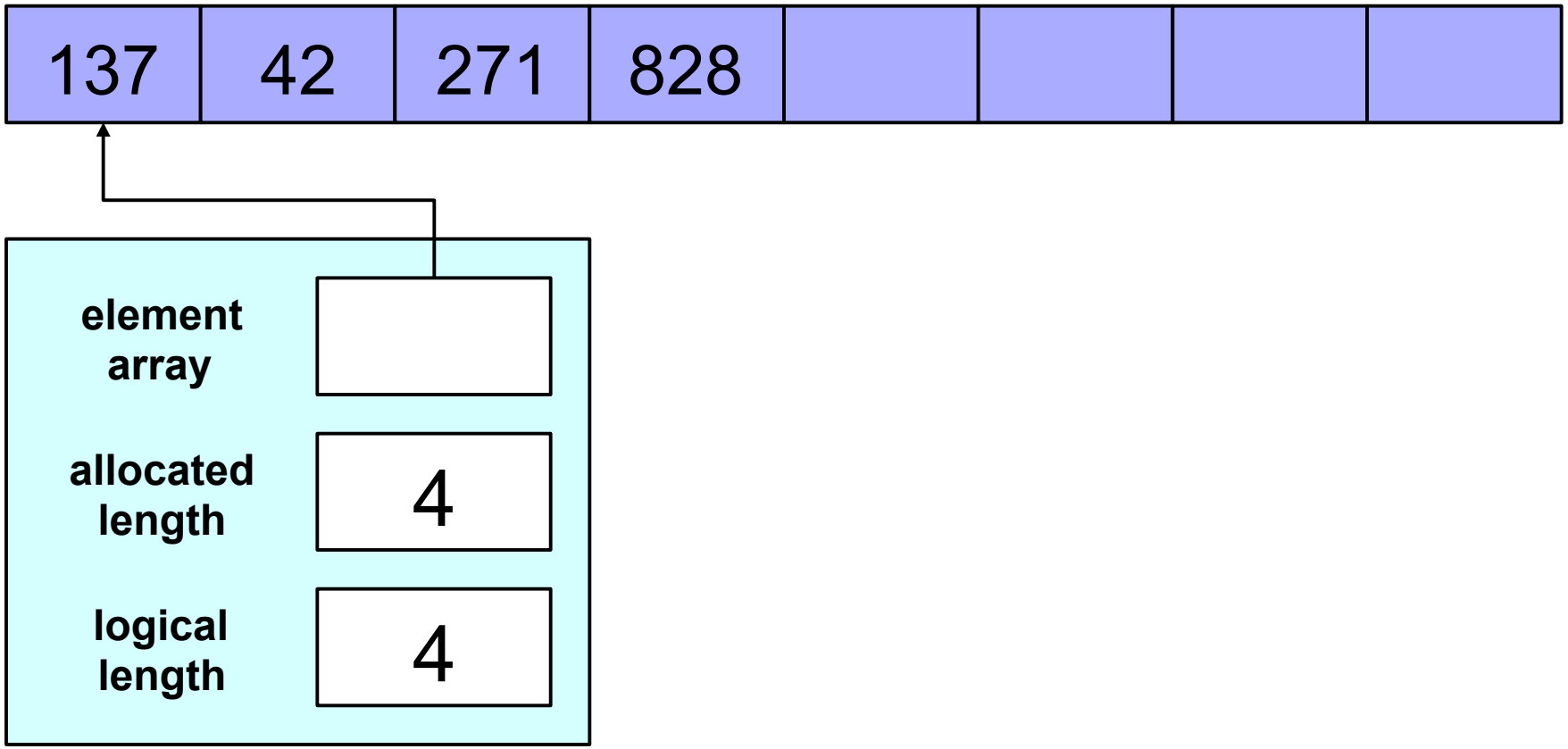
Actual Vector



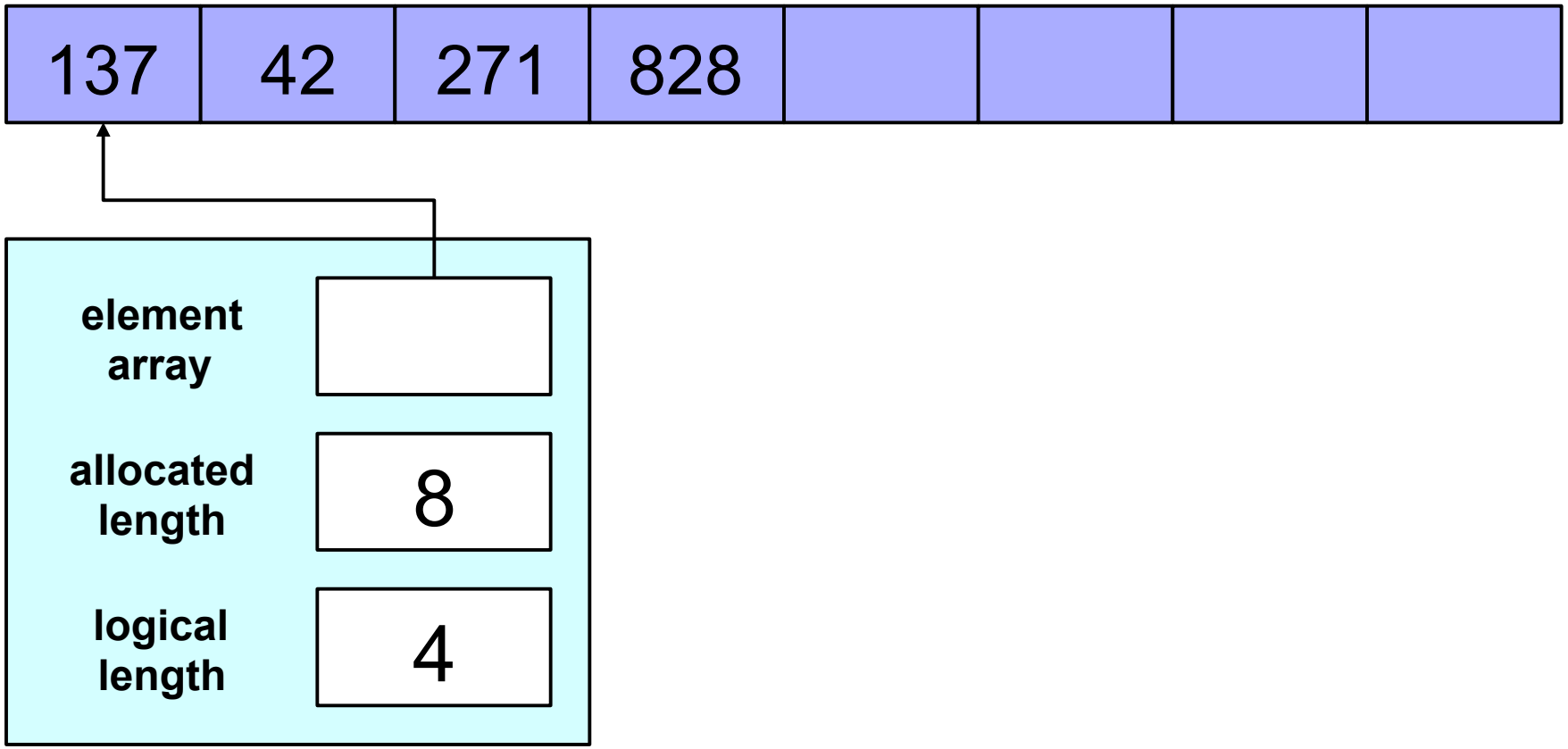
Actual Vector



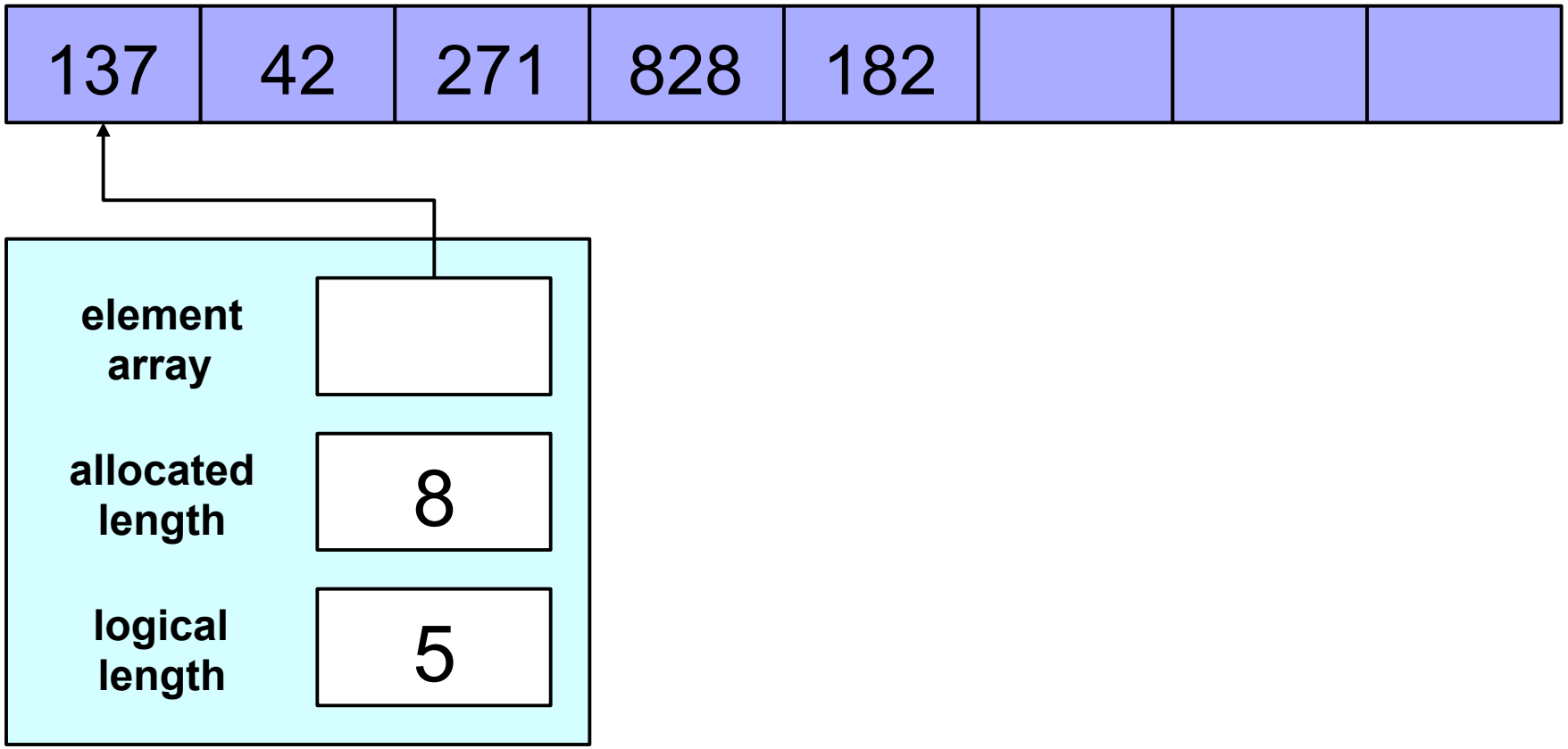
Actual Vector



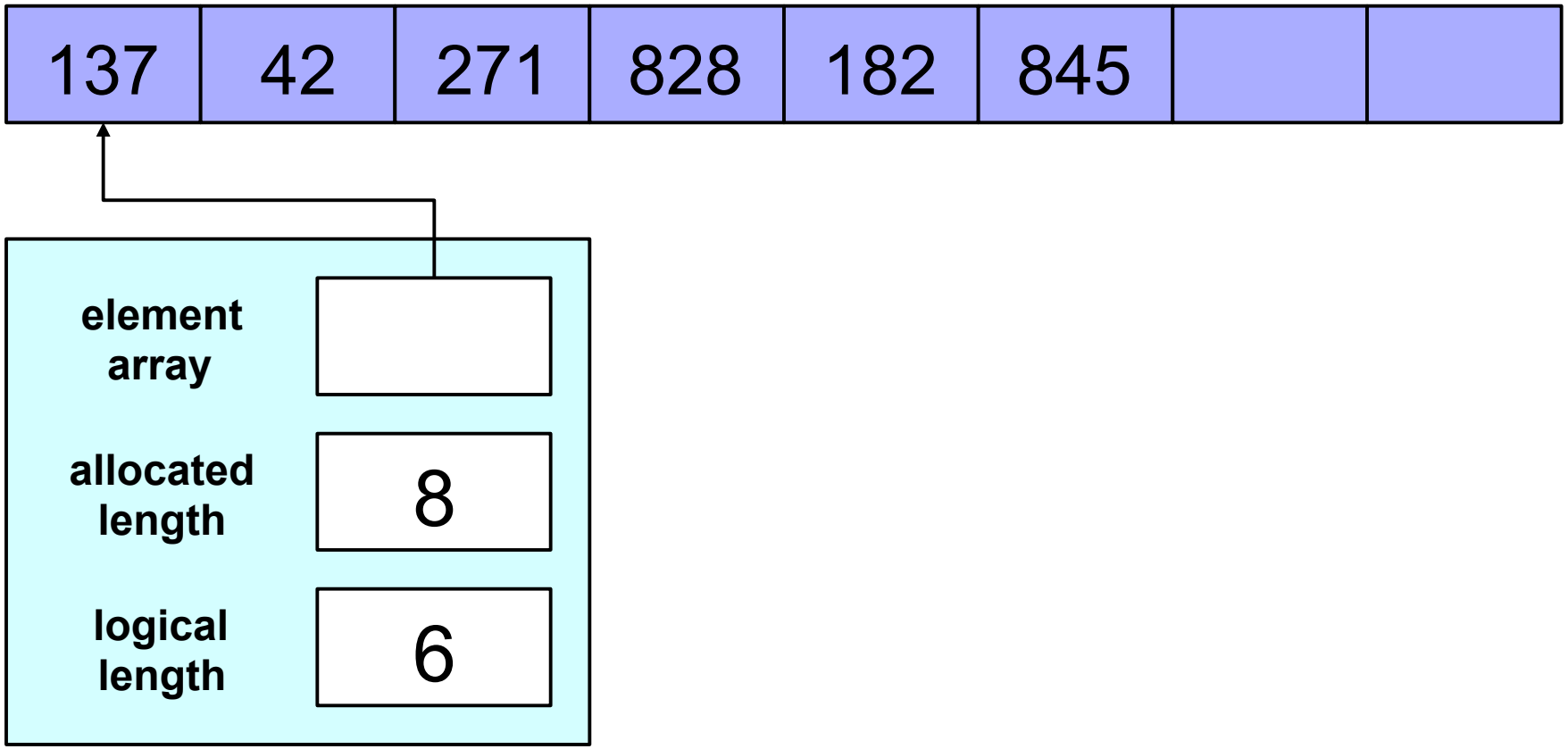
Actual Vector



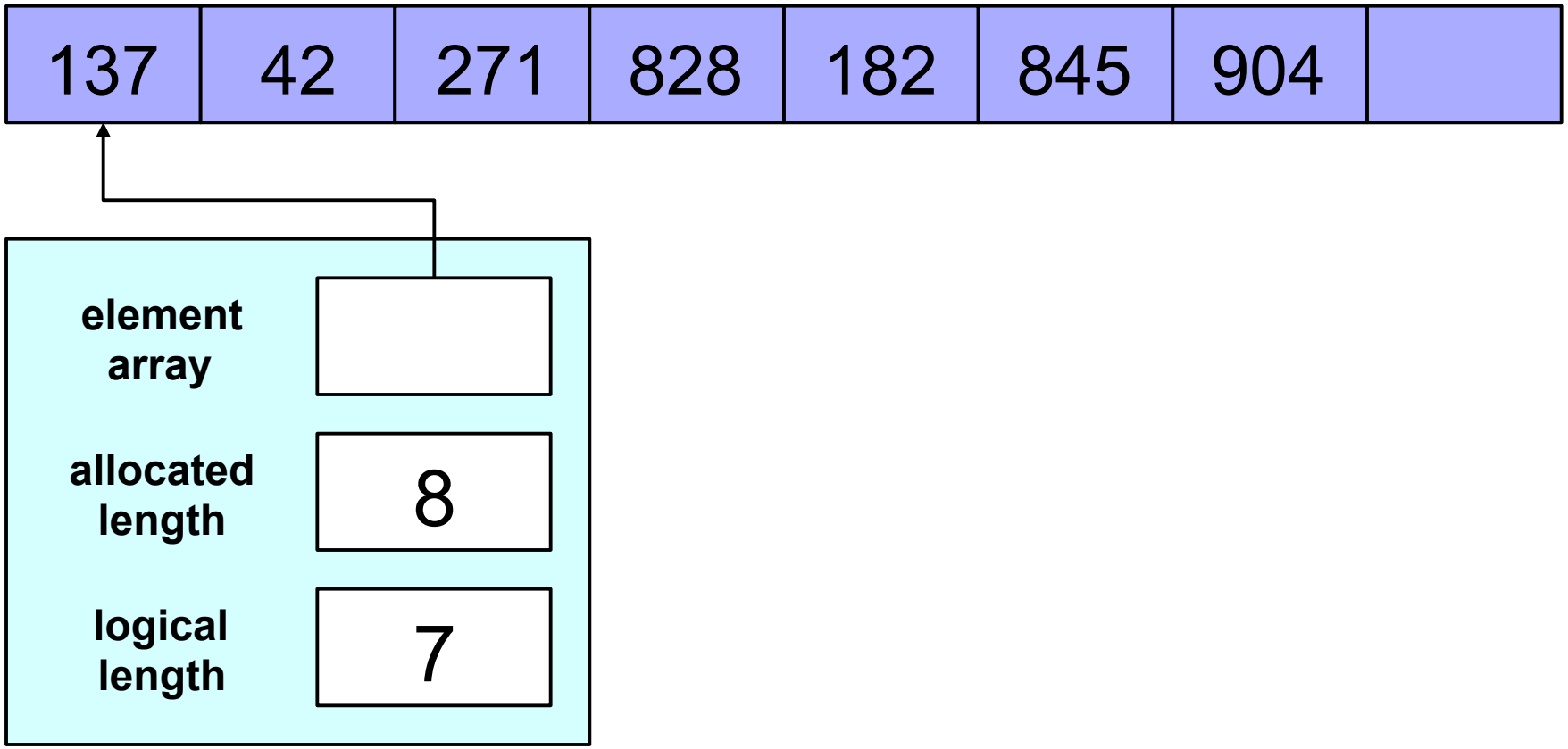
Actual Vector



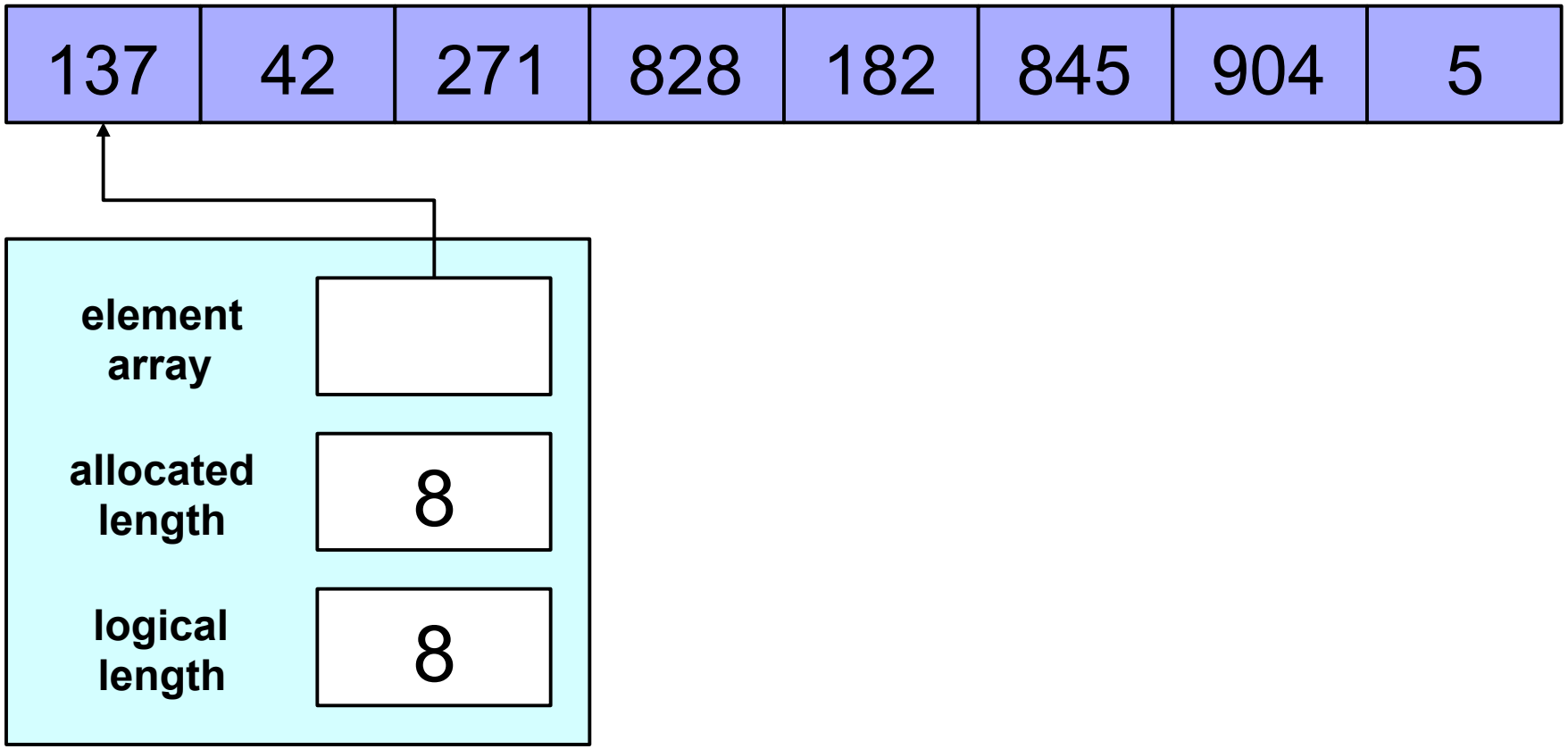
Actual Vector



Actual Vector



Actual Vector



Dynamic allocation was necessary

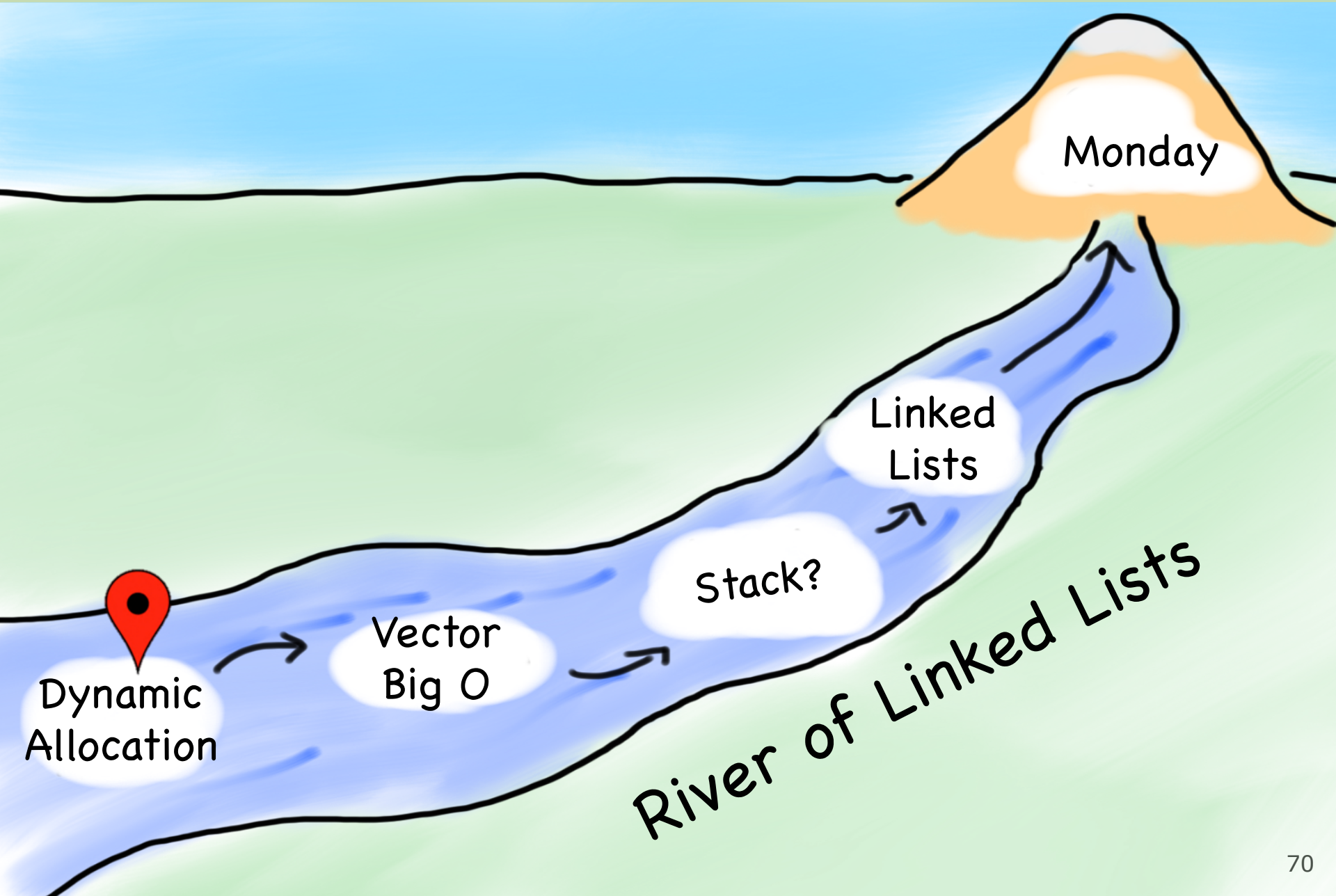
Benefits of Dynamic Allocation

Can decide on the number of variables at run time.

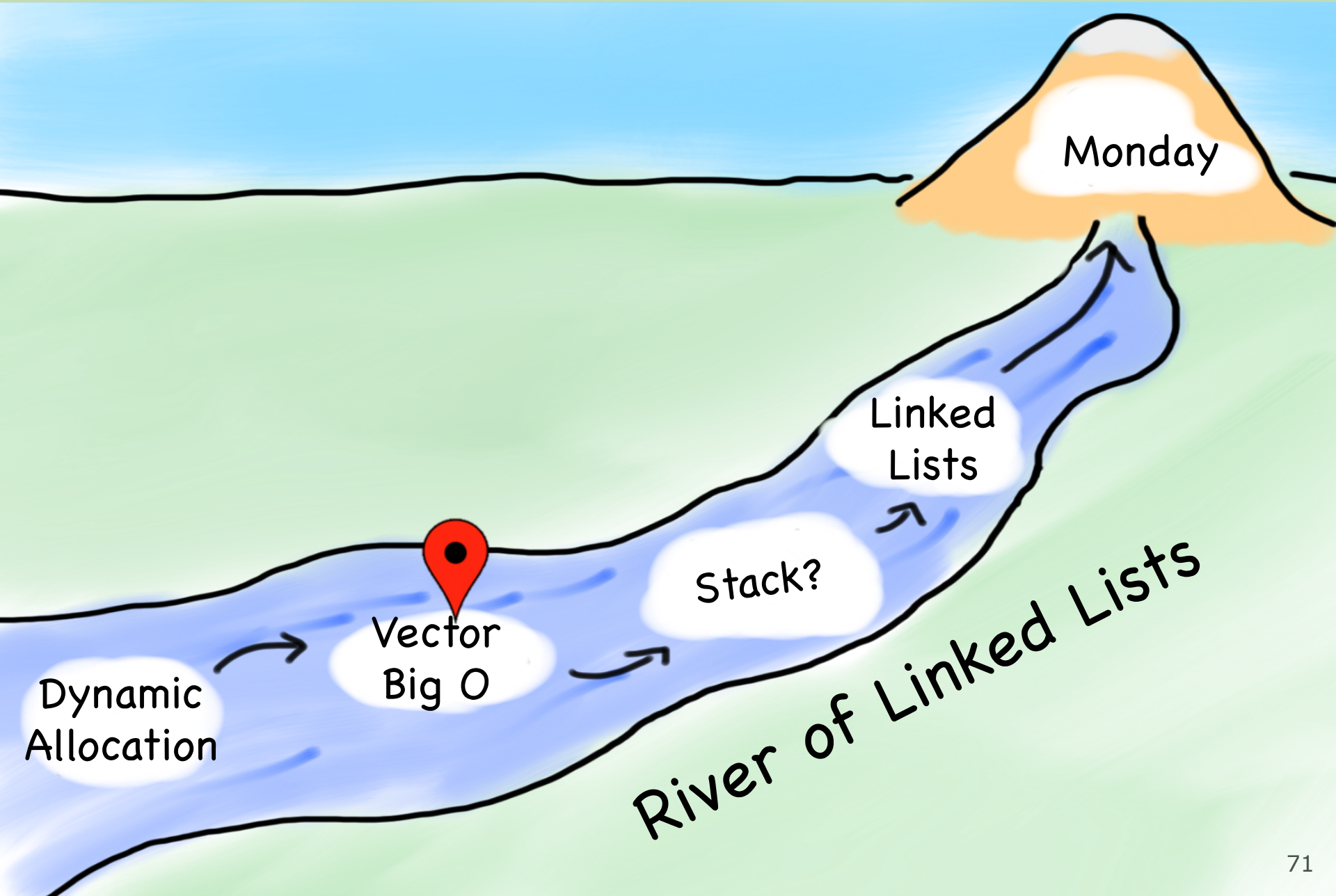
Control over variable lifespan

Its easy to share a single large object.

Today's Goals



Today's Goals



Vector Big O?

Big O of Get?

```
int VecInt::get(int index){
    return data[index];
}
```

Big O of Get?

$$\mathcal{O}(1)$$

Big O of Get?

Add?

Big O of Add?

```
void VecInt::add(int v){
    data[usedSize] = v;
    usedSize++;
    if(usedSize == allocatedSize) {
        doubleAllocation();
    }
}

void VecInt::doubleAllocation() {
    allocatedSize *= 2;
    int * newData = new int[allocatedSize];
    for(int i = 0; i < usedSize; i++) {
        newData[i] = data[i];
    }
    delete[] data;
    data = newData;
}
```

Big O of Add?

```
void VecInt::add(int v){
    data[usedSize] = v;
    usedSize++;
    if(usedSize == allocatedSize) {
        doubleAllocation();
    }
}

void VecInt::doubleAllocation() {
    allocatedSize *= 2;
    int * newData = new int[allocatedSize];
    for(int i = 0; i < usedSize; i++) {
        newData[i] = data[i];
    }
    delete[] data;
    data = newData;
}
```

Worst Case

Big O of Add?

```
void VecInt::add(int v){
    data[usedSize] = v;
    usedSize++;
    if(usedSize == allocatedSize) {
        doubleAllocation();
    }
}

void VecInt::doubleAllocation() {
    allocatedSize *= 2;
    int * newData = new int[allocatedSize];
    for(int i = 0; i < usedSize; i++) {
        newData[i] = data[i];
    }
    delete[] data;
    data = newData;
}
```

Worst Case

Big O of Add?

```
void VecInt::add(int v){
    data[usedSize] = v;
    usedSize++;
    if(usedSize == allocatedSize) {
        doubleAllocation();
    }
}

void VecInt::doubleAllocation() {
    allocatedSize *= 2;
    int * newData = new int[allocatedSize];
    for(int i = 0; i < usedSize; i++) {
        newData[i] = data[i];
    }
    delete[] data;
    data = newData;
}
```

Worst Case

Big O of Add?

```
void VecInt::add(int v){
    data[usedSize] = v;
    usedSize++;
    if(usedSize == allocatedSize) {
        doubleAllocation();
    }
}

void VecInt::doubleAllocation() {
    allocatedSize *= 2;
    int * newData = new int[allocatedSize];
    for(int i = 0; i < usedSize; i++) {
        newData[i] = data[i];
    }
    delete[] data;
    data = newData;
}
```

Worst Case

Big O of Add?

```
void VecInt::add(int v){
    data[usedSize] = v;
    usedSize++;
    if(usedSize == allocatedSize) {
        doubleAllocation();
    }
}

void VecInt::doubleAllocation() {
    allocatedSize *= 2;
    int * newData = new int[allocatedSize];
    for(int i = 0; i < usedSize; i++) {
        newData[i] = data[i];
    }
    delete[] data;
    data = newData;
}
```

Worst Case

Big O of Add?

$$\mathcal{O}(n)$$

Remove?

Big O of Remove?

```
void VecInt::remove(int index){
    for(int i = index; i < usedSize - 1; i++) {
        data[i] = data[i + 1];
    }
    usedSize--;
}
```

Big O of Remove?

$$\mathcal{O}(n)$$

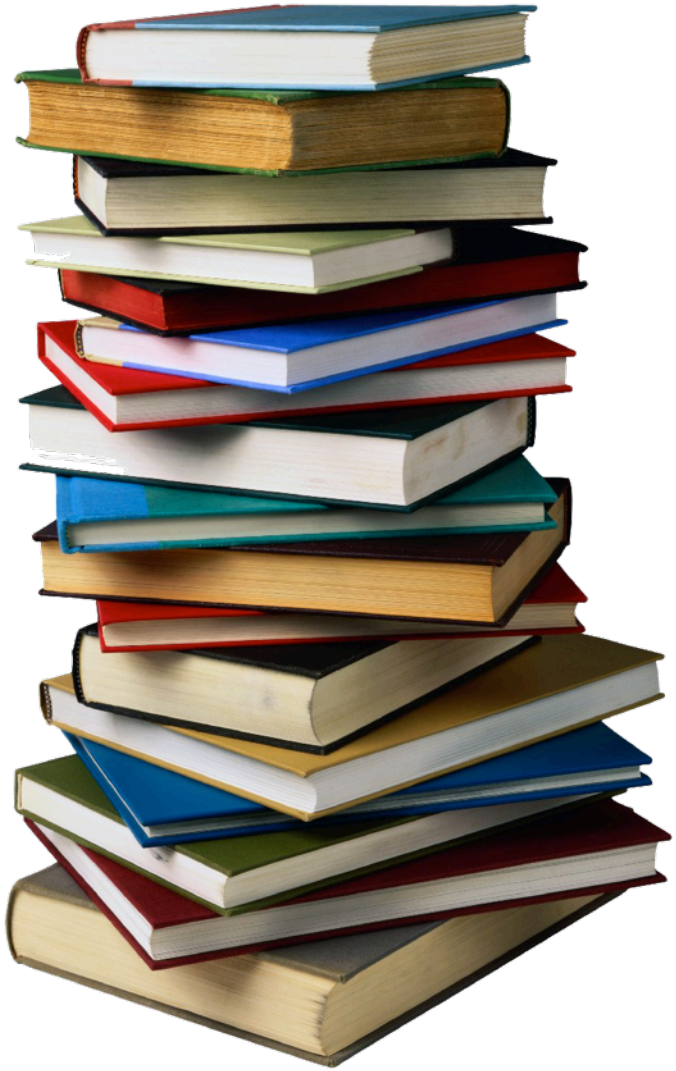
Summary?

Vector Big O

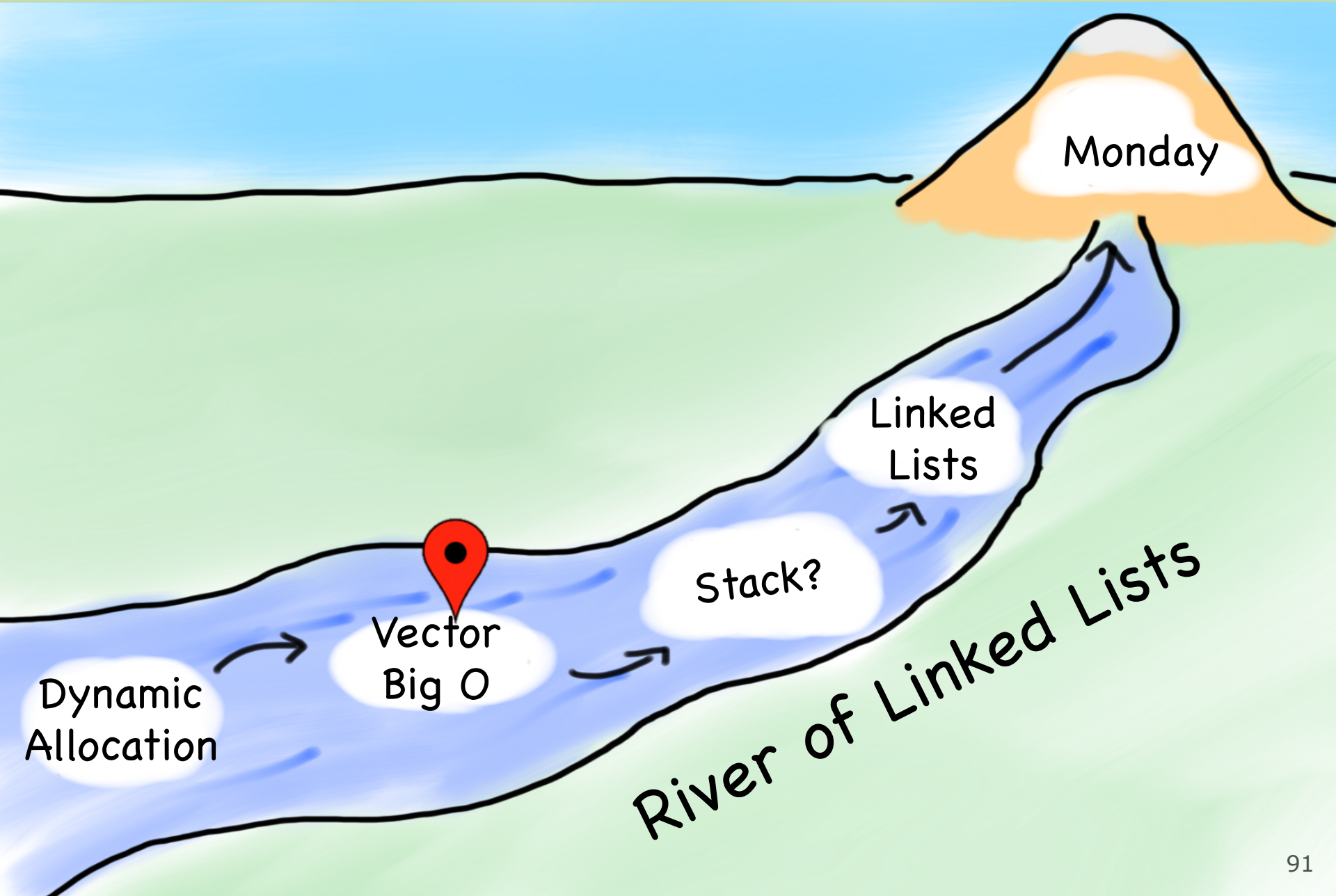
Get	$\mathcal{O}(1)$
Add	$\mathcal{O}(n)$
Remove	$\mathcal{O}(n)$

That’s fine.

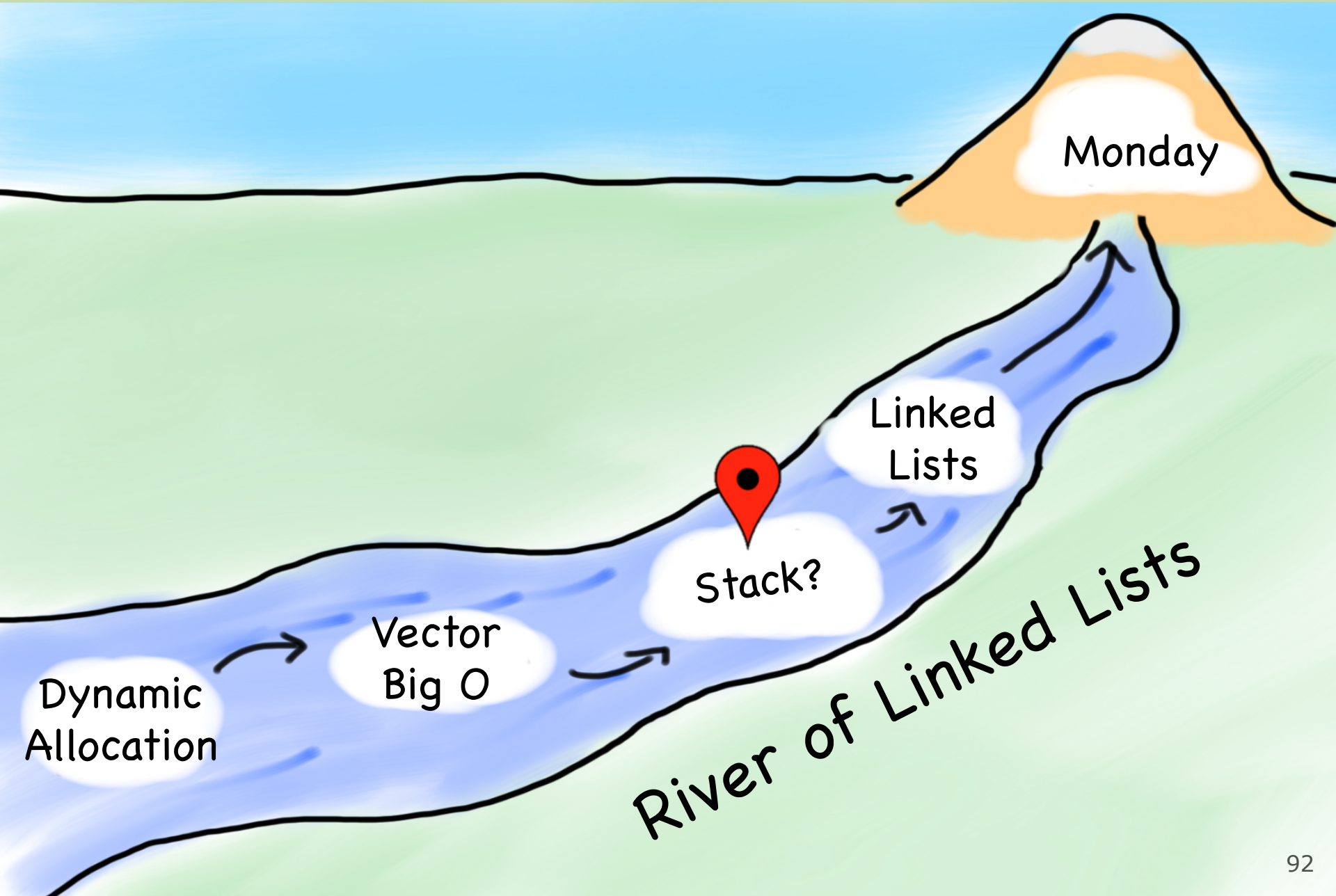
How is the Stack Implemented?



Today's Goals



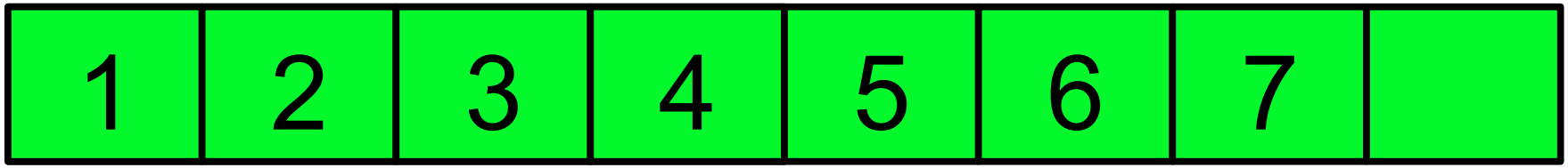
Today's Goals



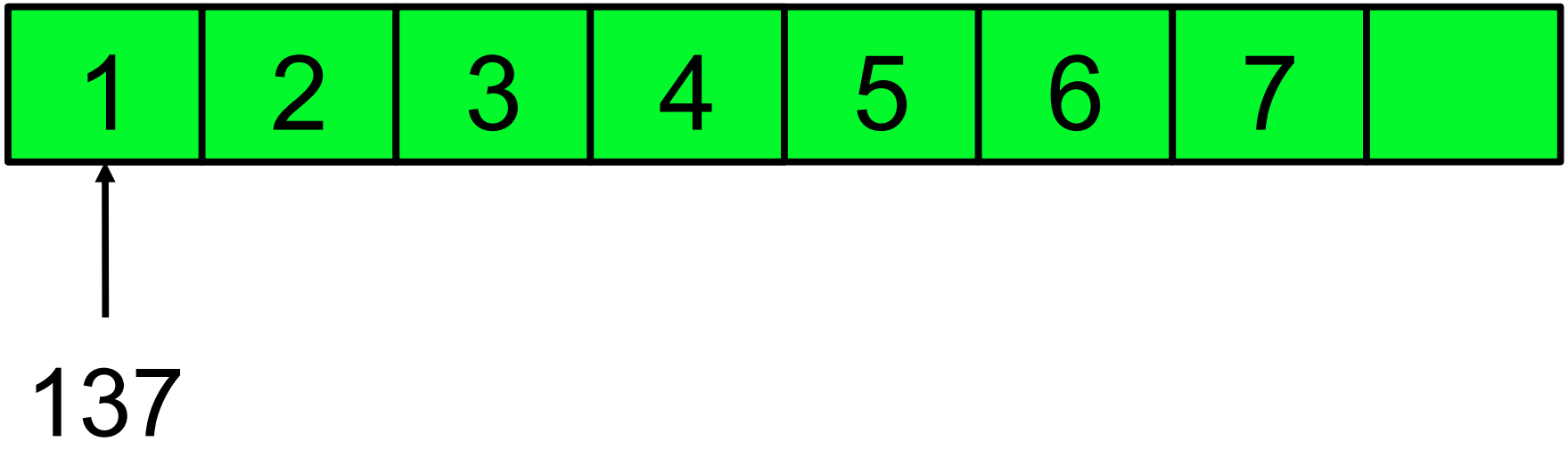
VectorInt

```
class StackInt {          // in VectorInt.h
public:
    StackInt ();            // constructor

    void push(int value); // append a value to the end
    int pop();              // return the value at index

private:
    VectorInt data;       // member variables
};
```

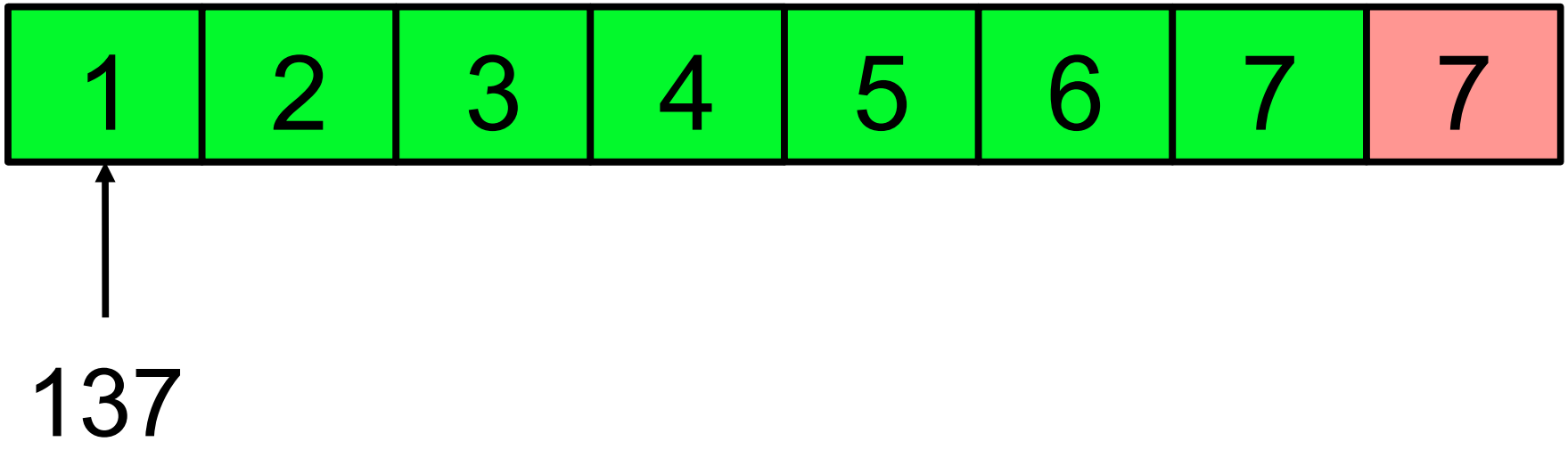
Excuse Me, Coming Through



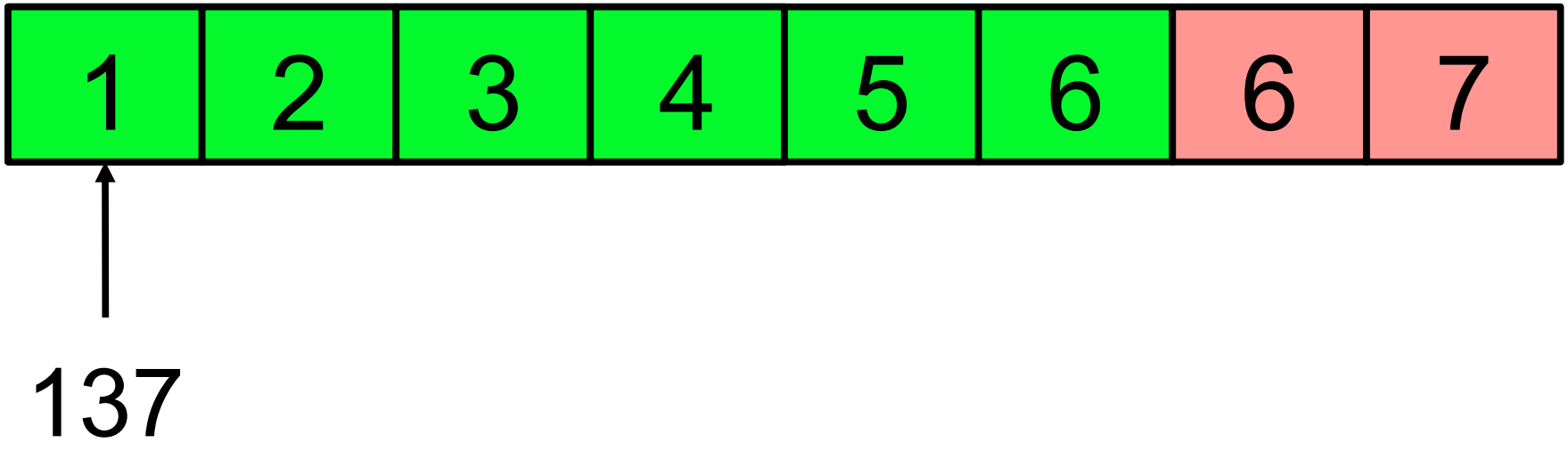
Excuse Me, Coming Through



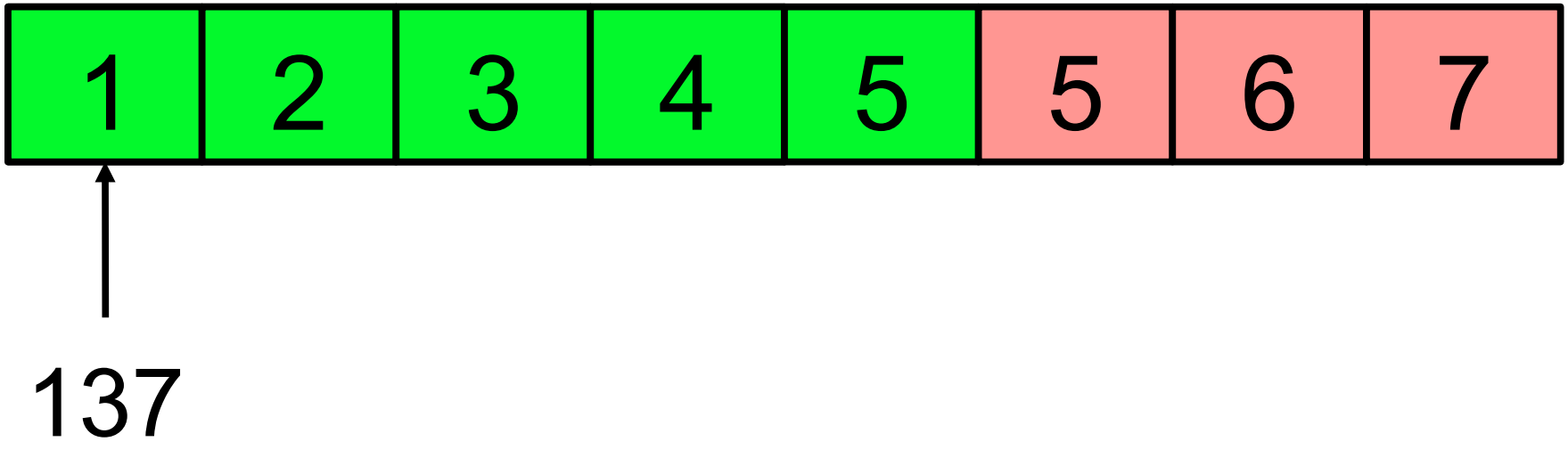
Excuse Me, Coming Through



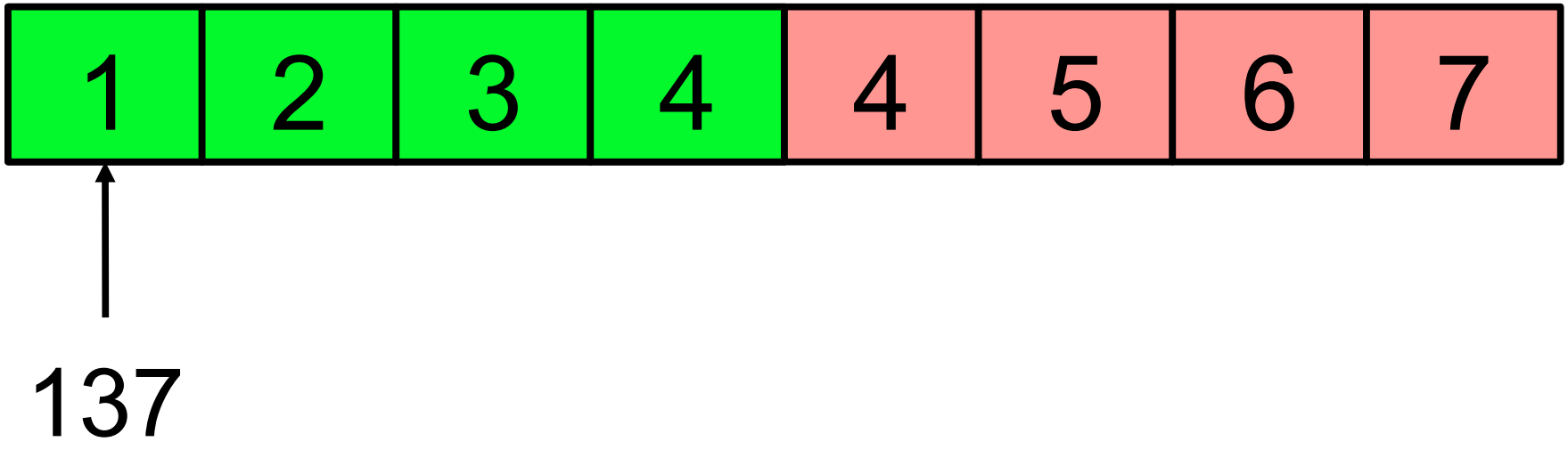
Excuse Me, Coming Through



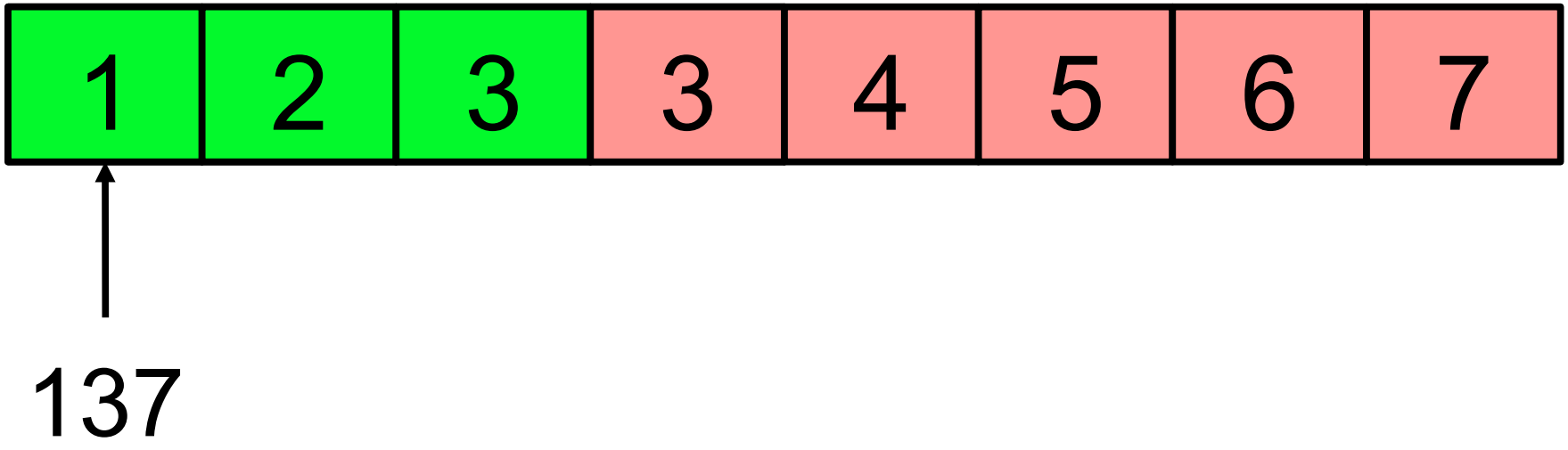
Excuse Me, Coming Through



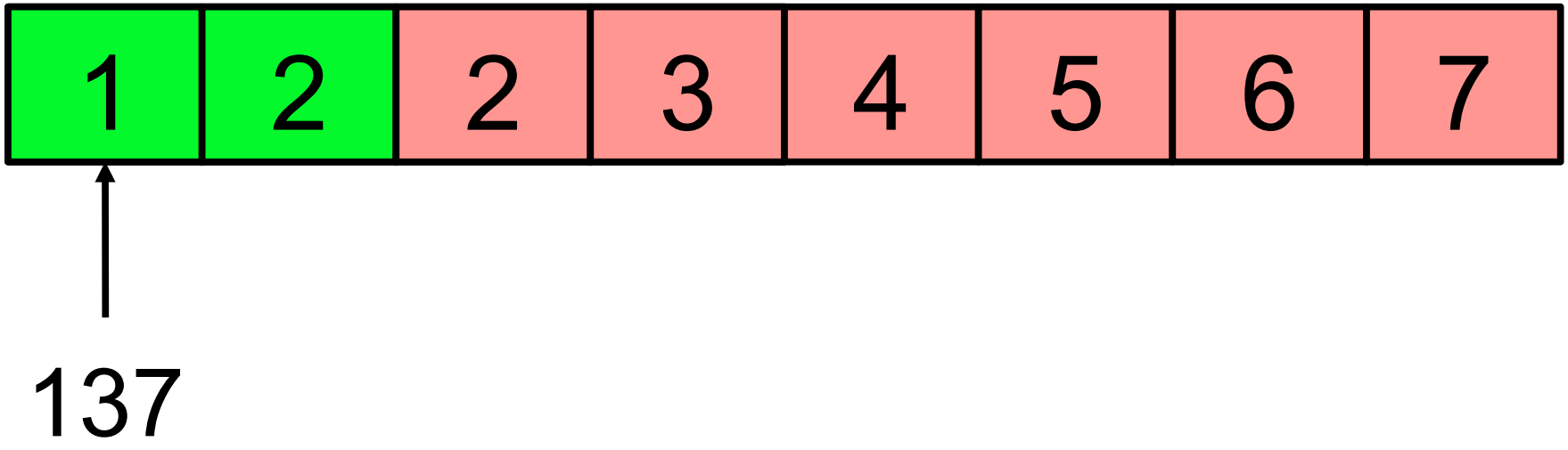
Excuse Me, Coming Through



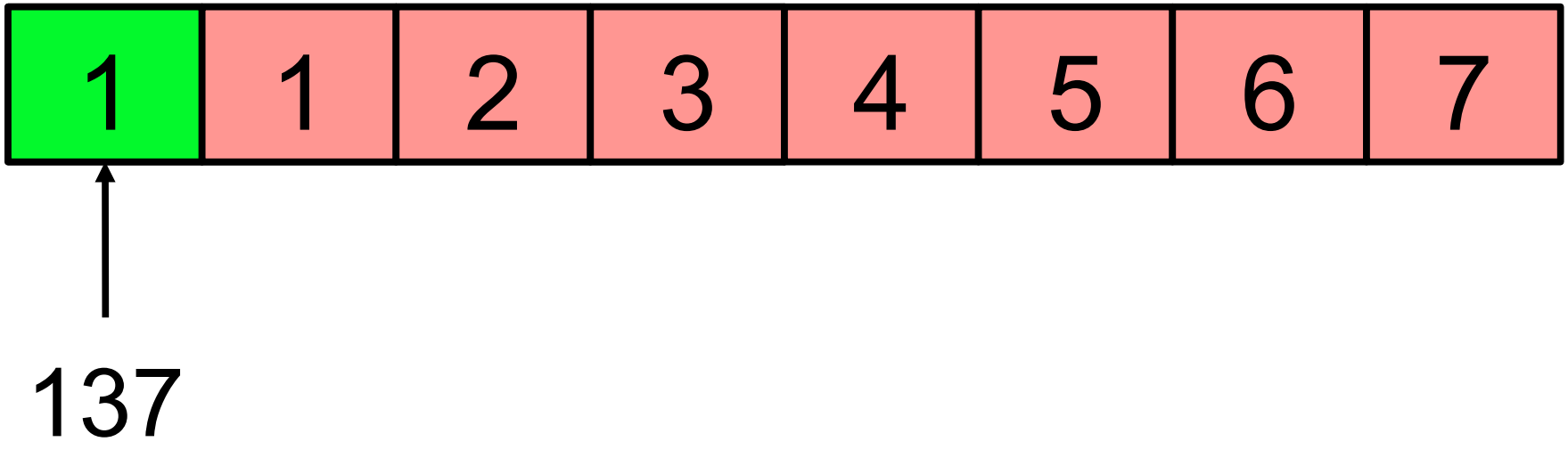
Excuse Me, Coming Through



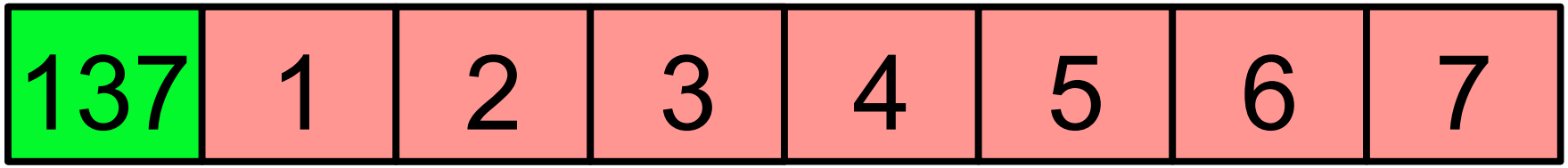
Excuse Me, Coming Through



Excuse Me, Coming Through



Excuse Me, Coming Through



Stack as Vector Big O

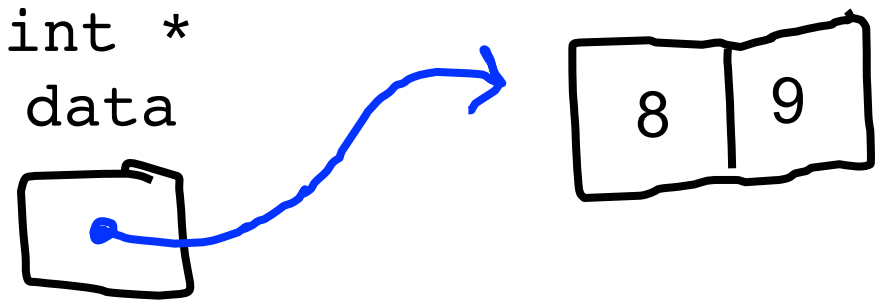
Push $\mathcal{O}(n)$

Pop $\mathcal{O}(1)$

There’s always a better way

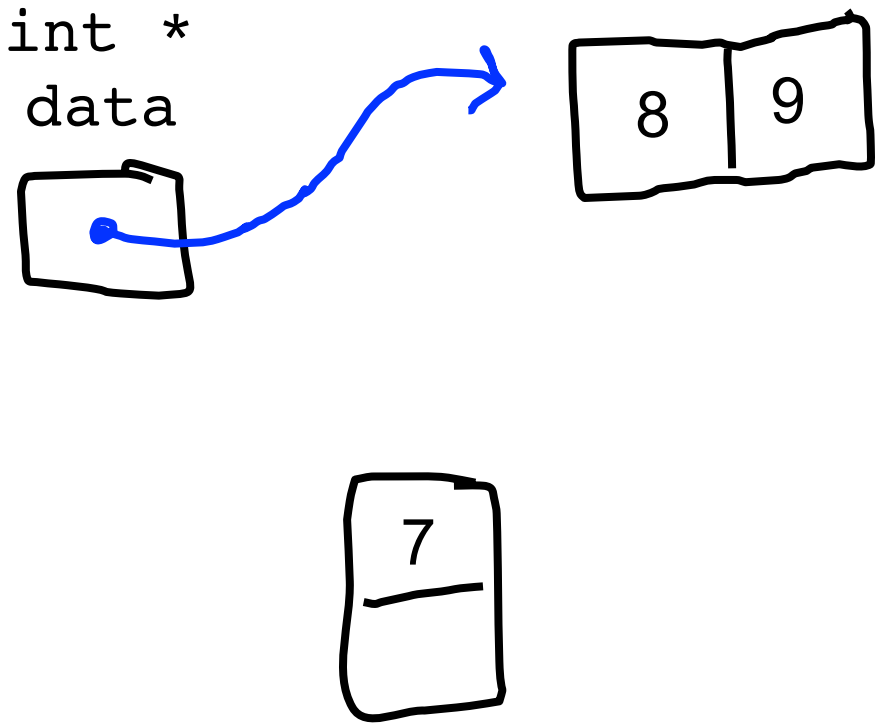
What About This?

```
push(7);
```



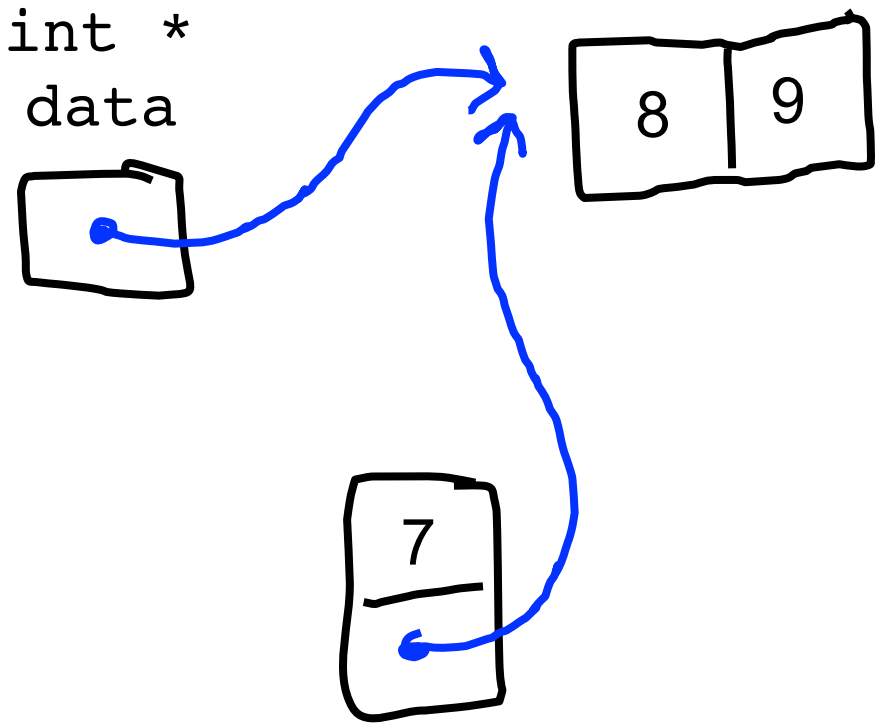
What About This?

```
push(7);
```



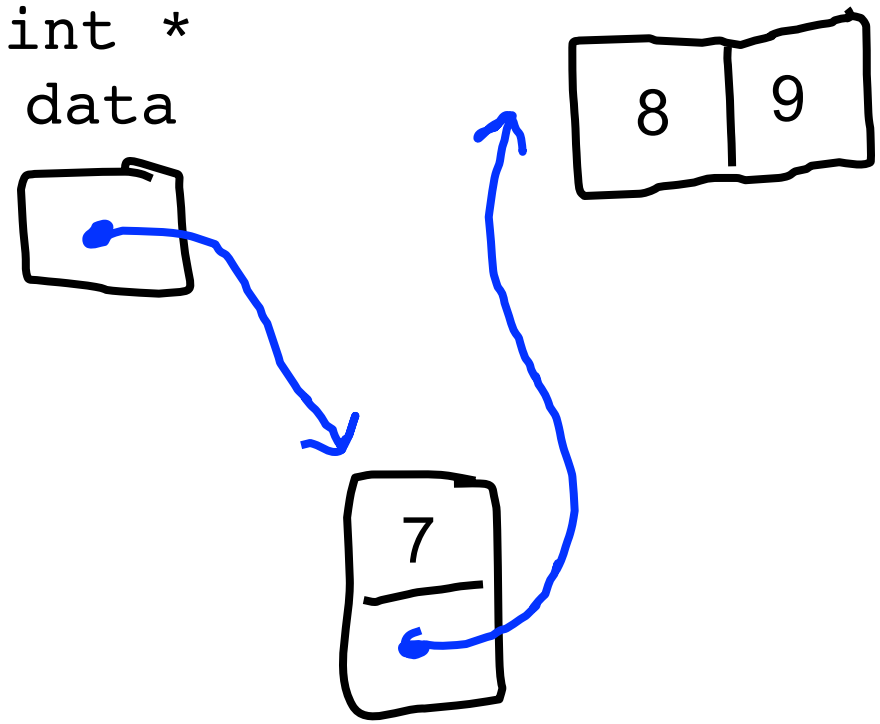
What About This?

```
push(7);
```



What About This?

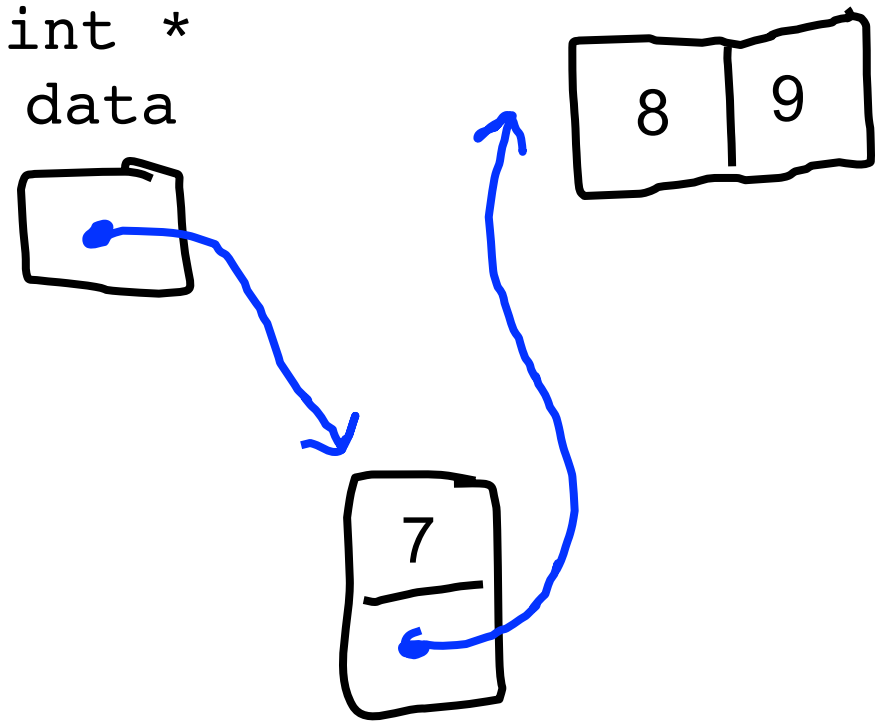
```
push(7);
```



Oh Cool

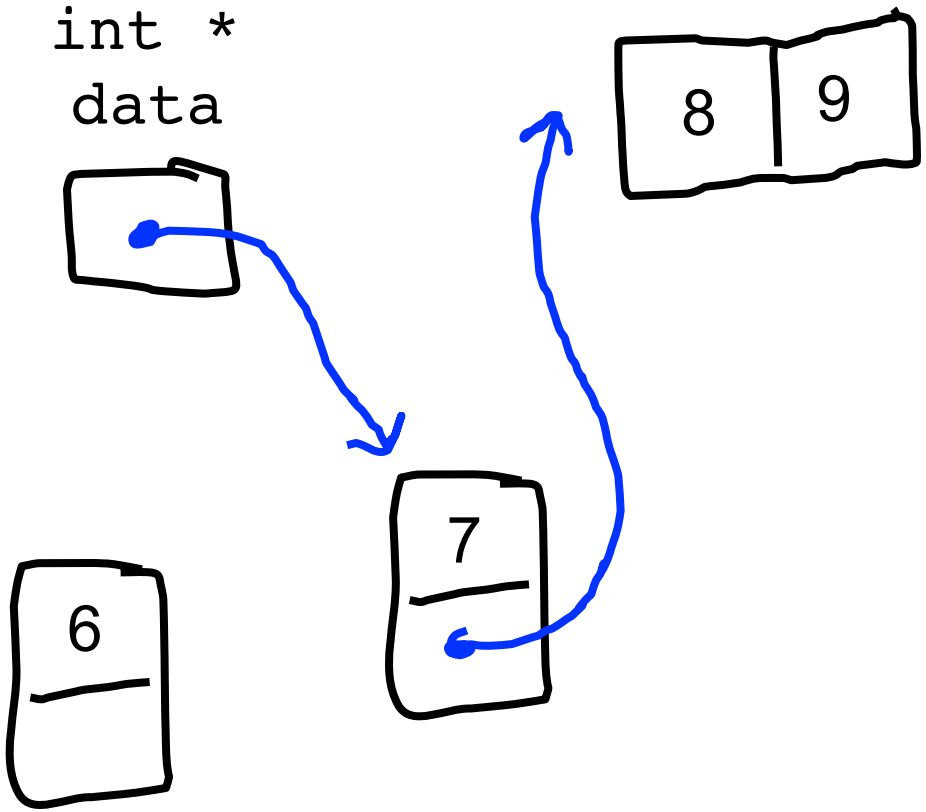
What About This?

```
push(6);
```



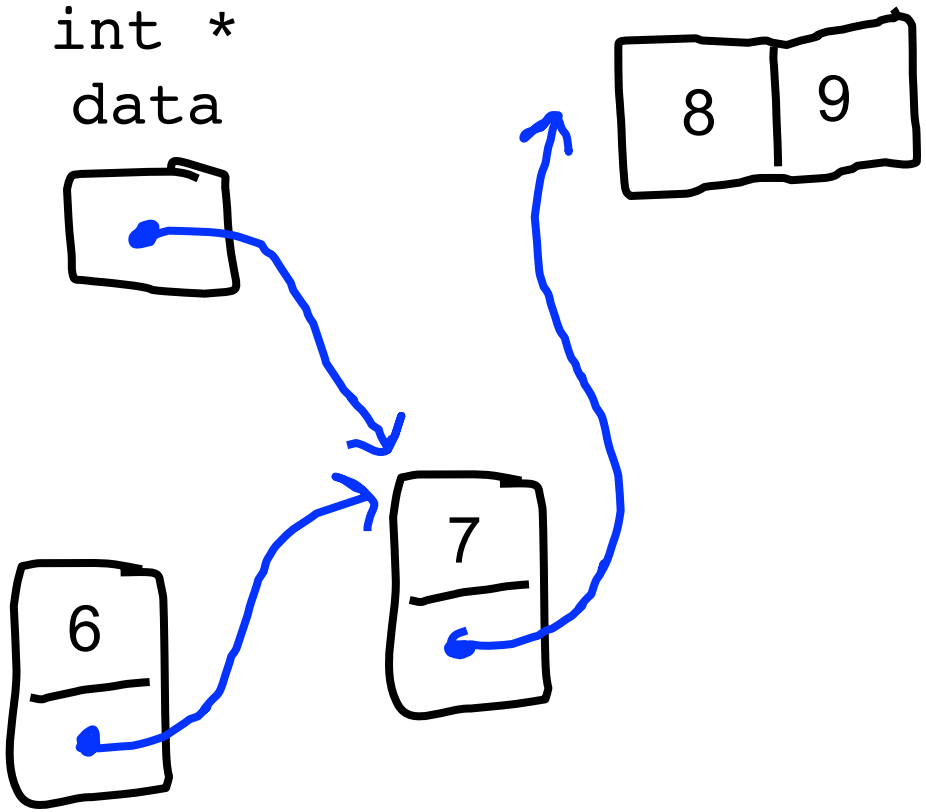
What About This?

```
push(6);
```



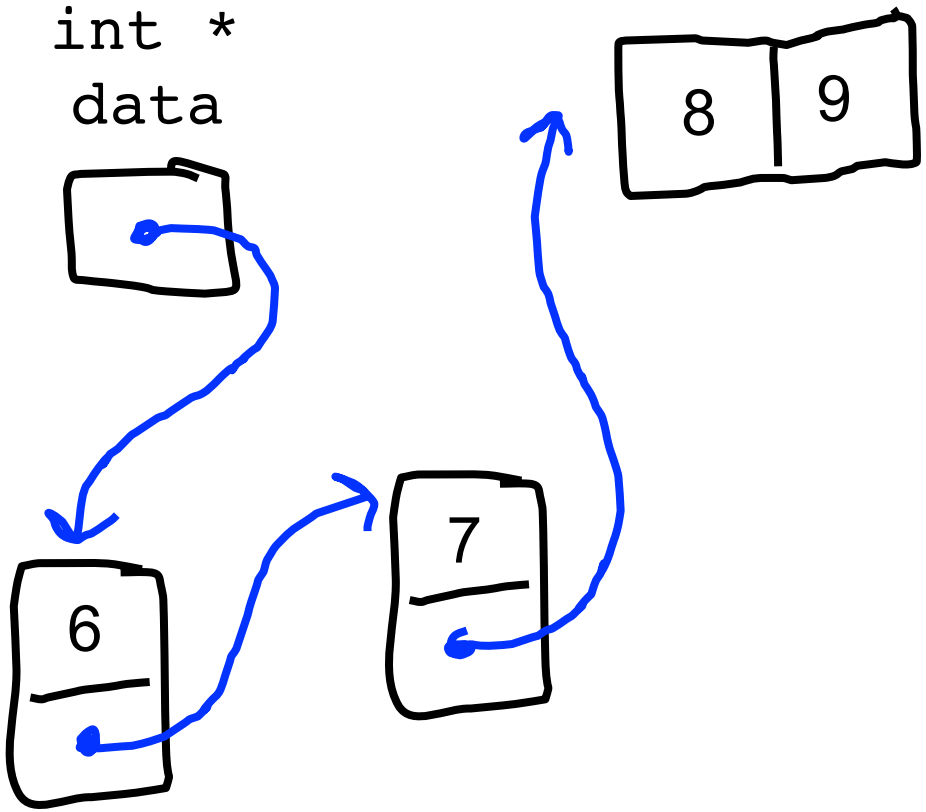
What About This?

```
push(6);
```



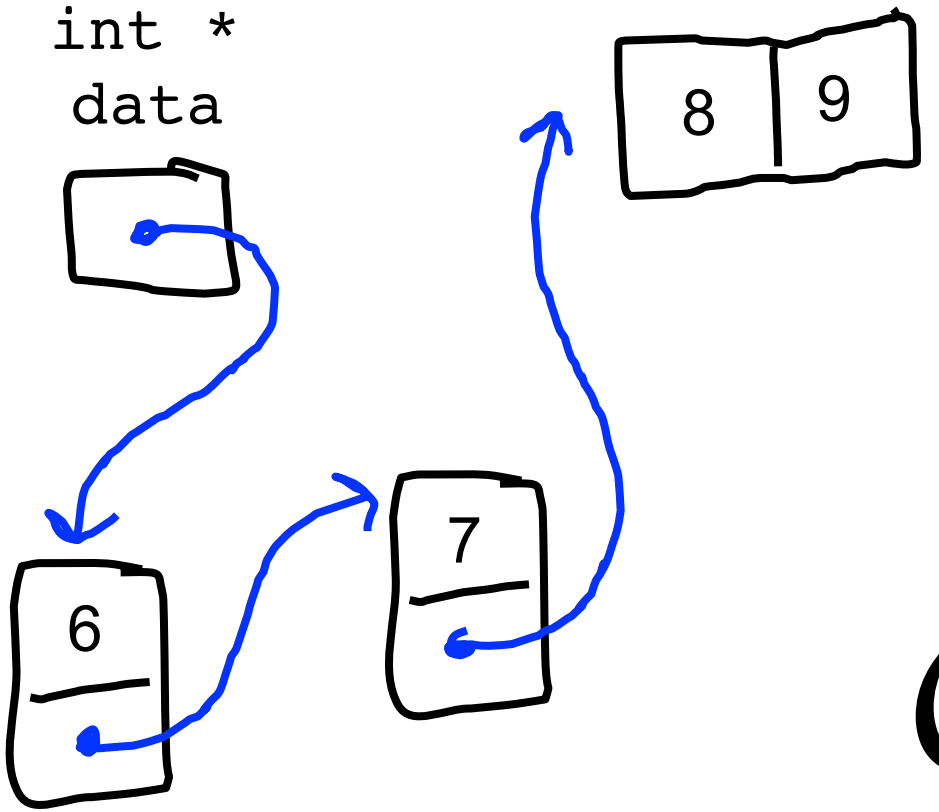
What About This?

```
push(6);
```



What About This?

```
push(6);
```

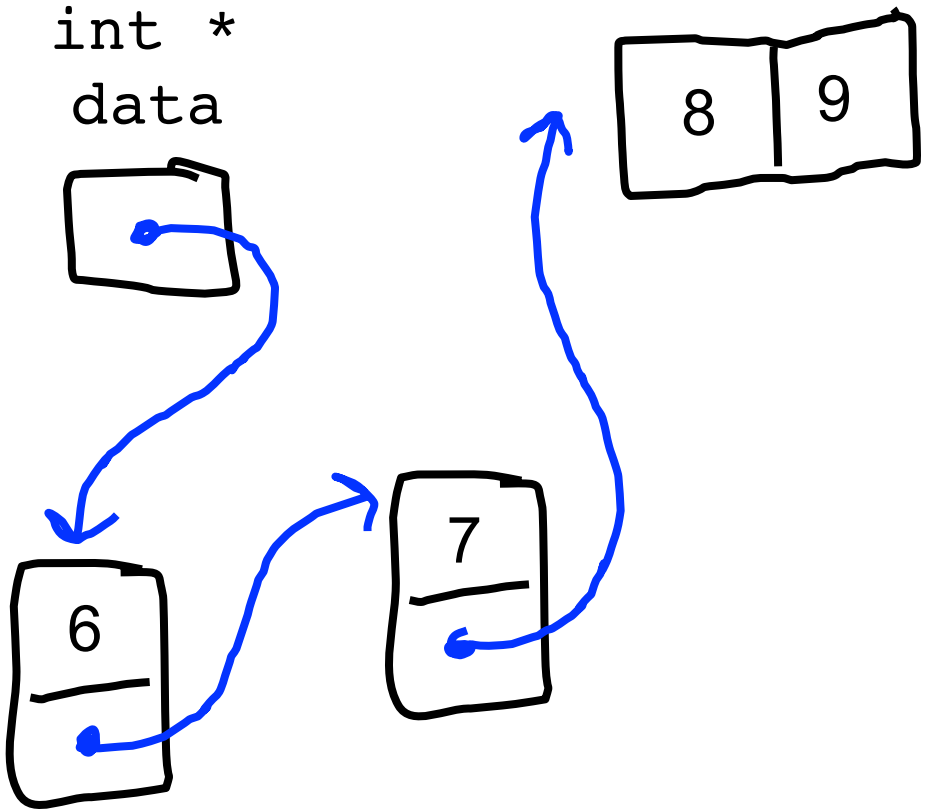


$$\mathcal{O}(1)$$

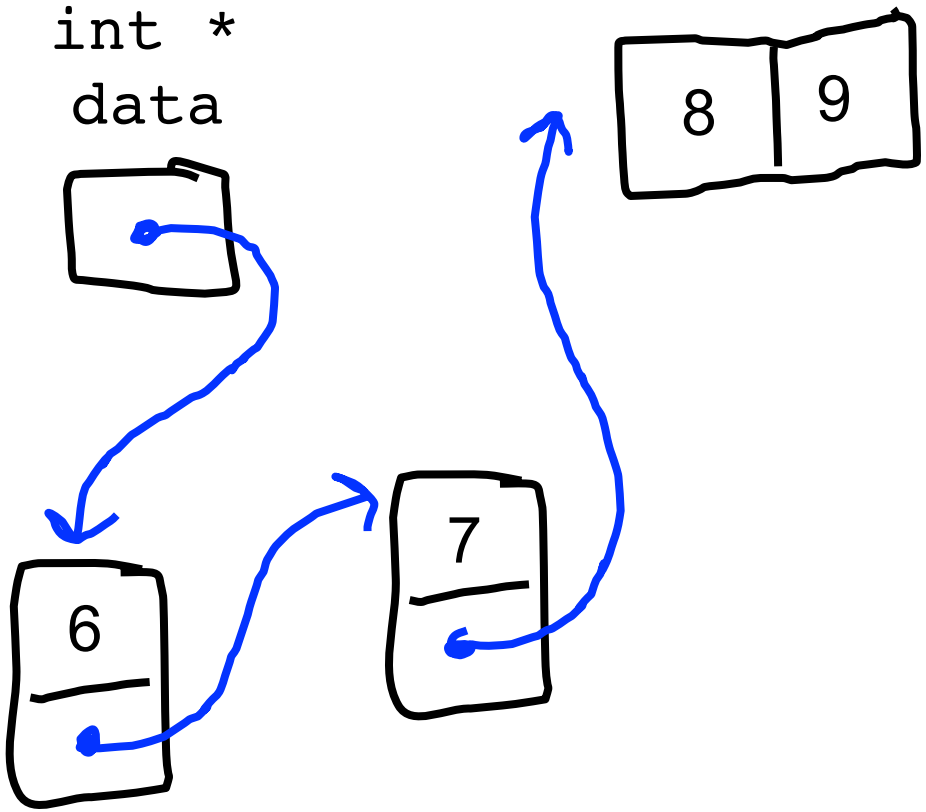
And Pop?

What About This?

```
pop();
```



What About This?

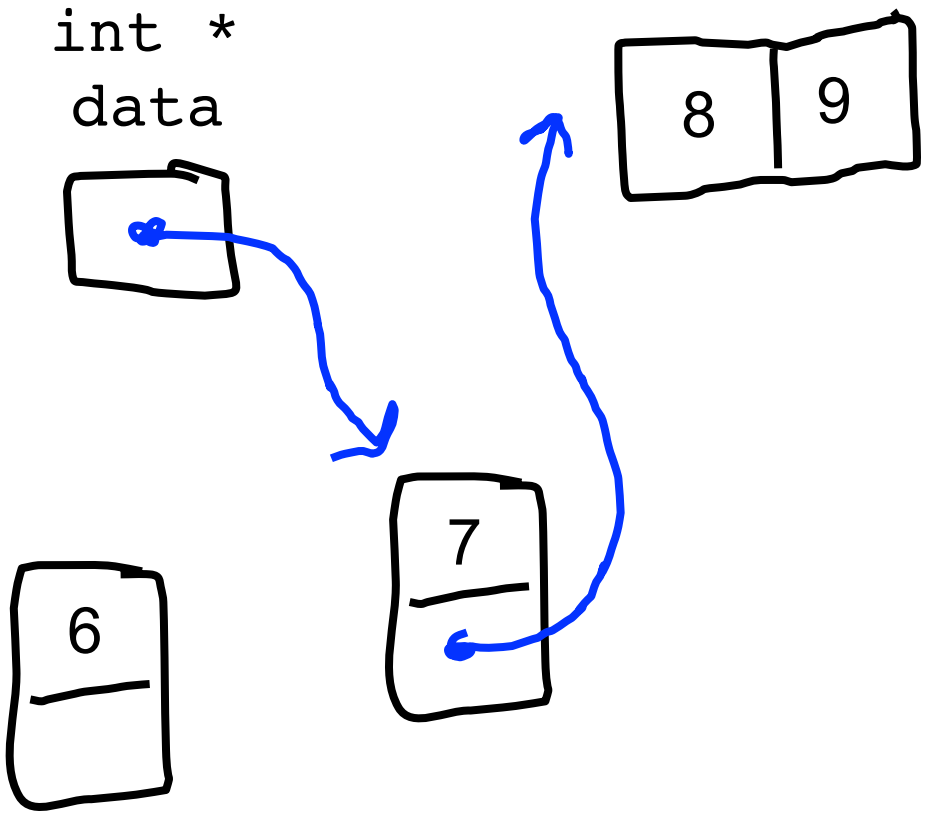



```
pop();
```

```
int return
```

6

What About This?

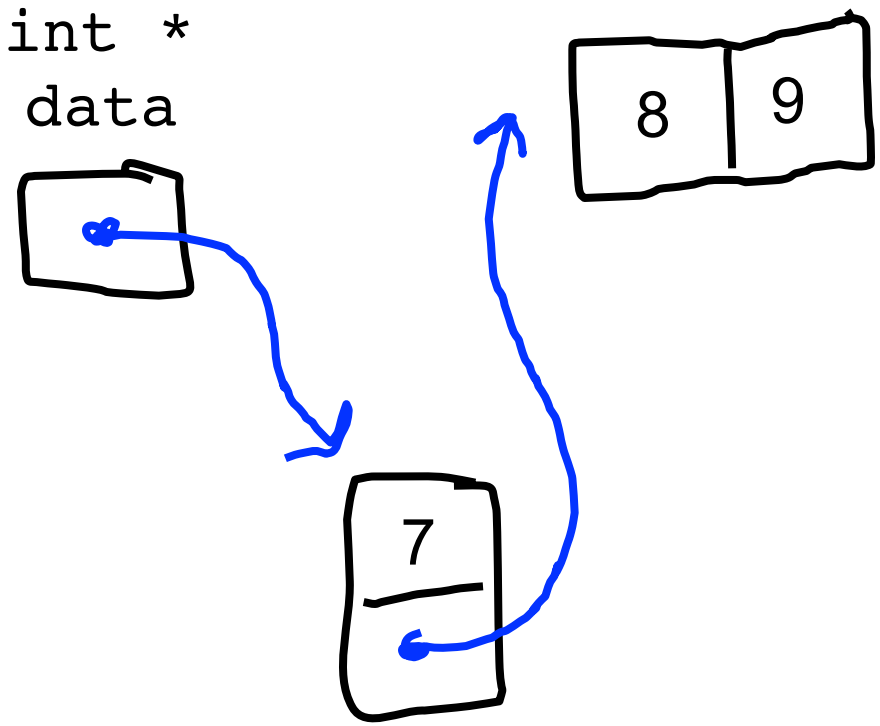



```
pop();
```

```
int return
```

6

What About This?



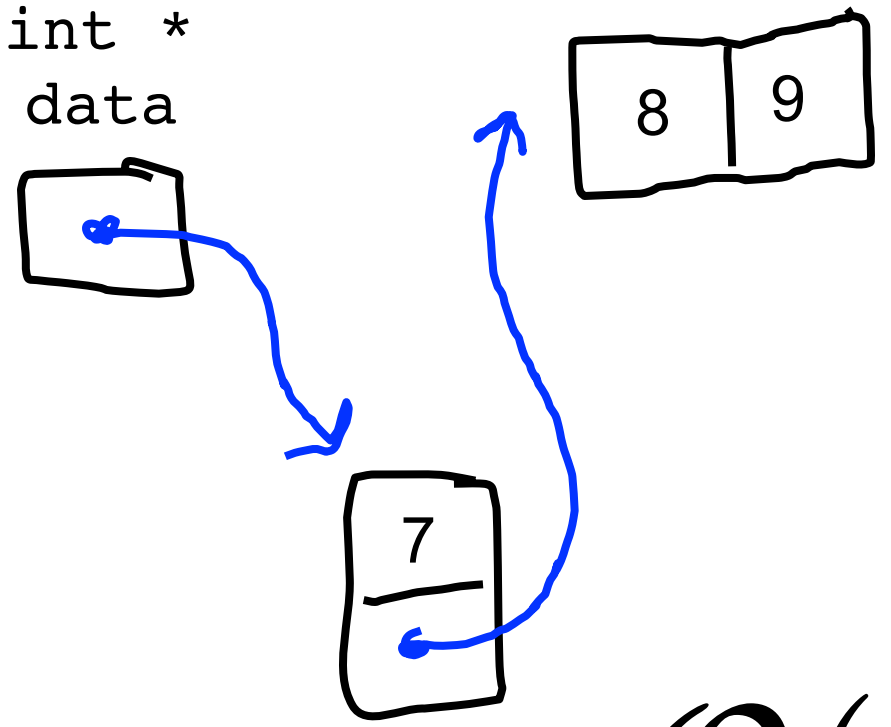

```
pop();
```

```
int return
```

6

What About This?

```
pop();
```



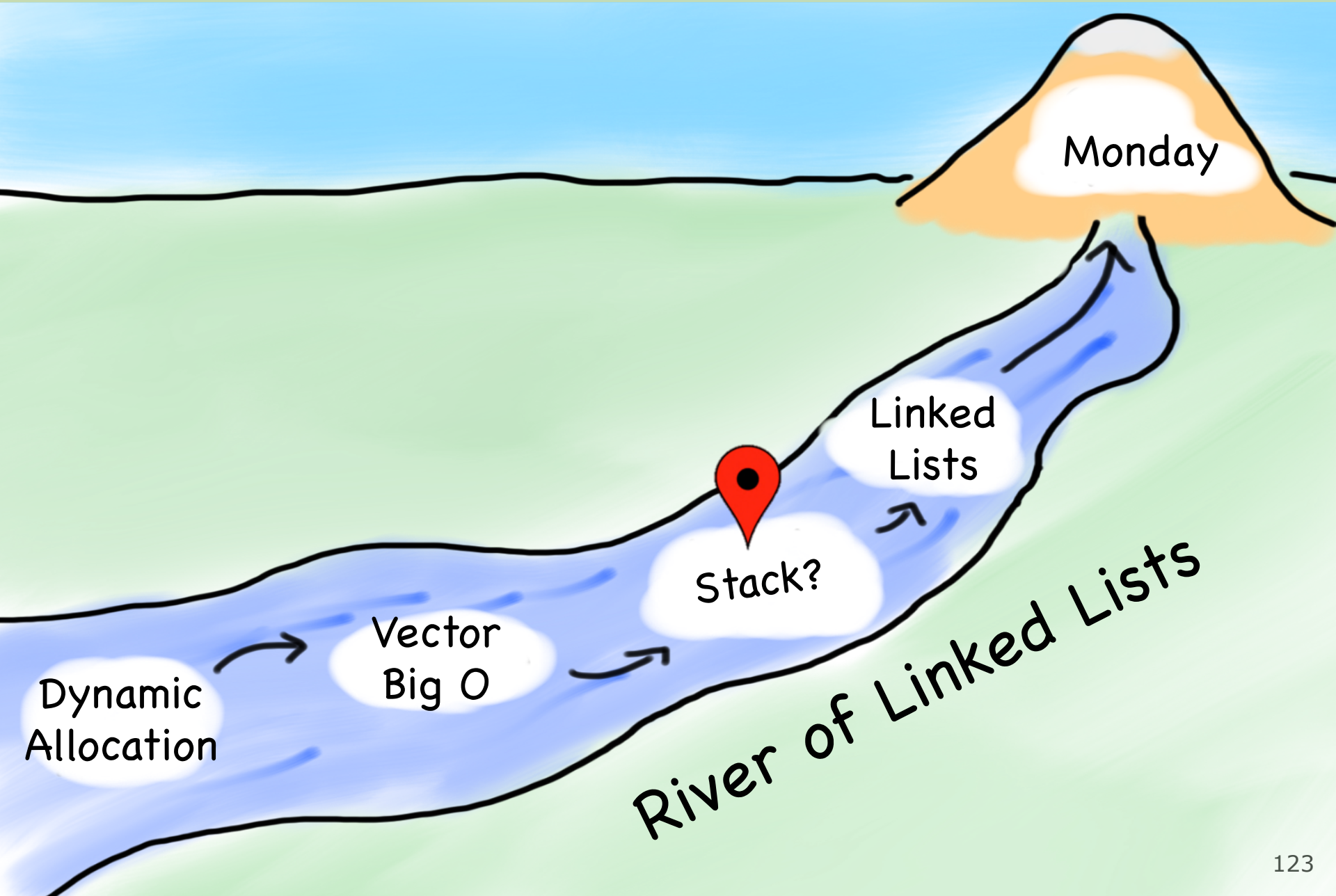

```
int return
```

6

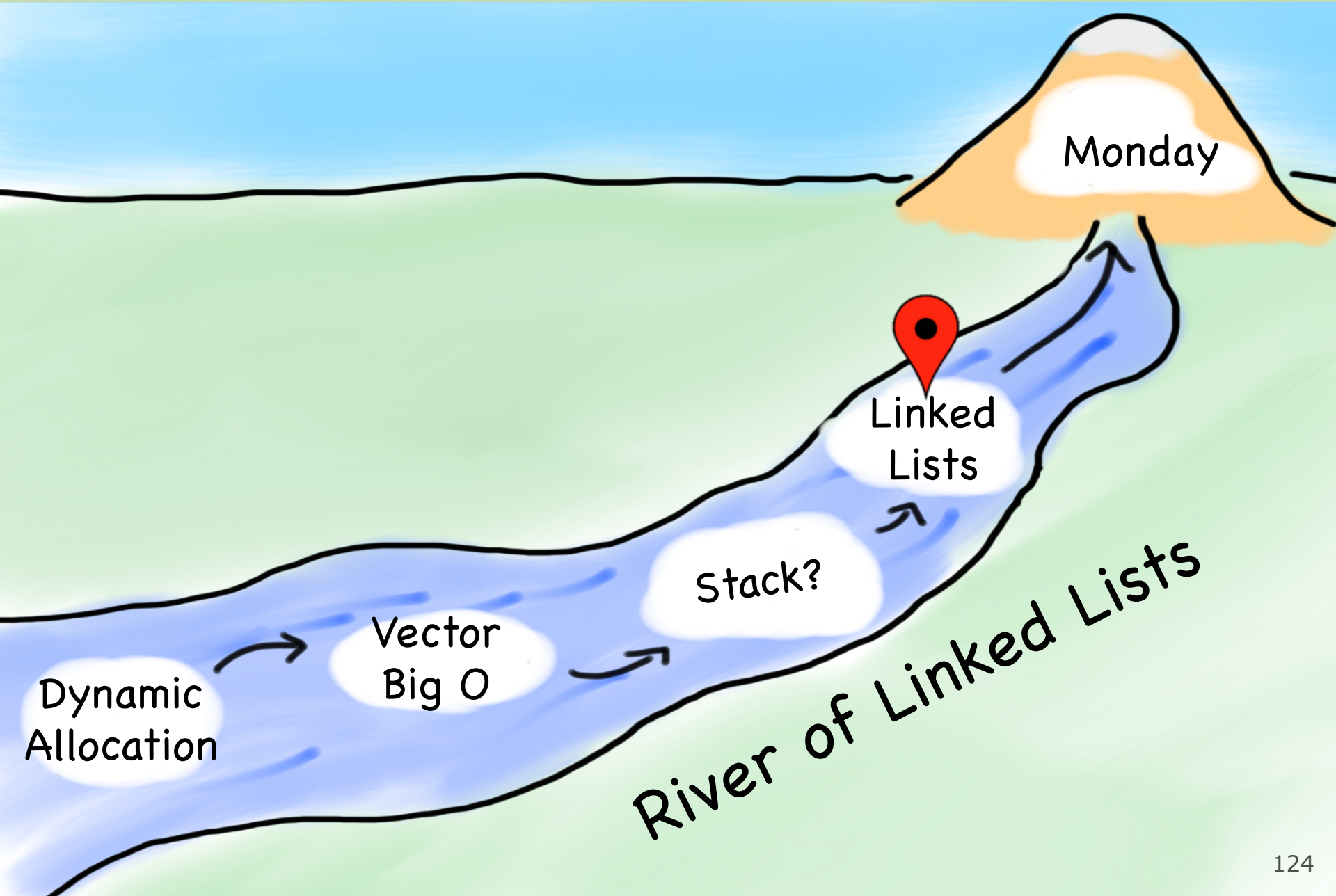
$$\mathcal{O}(1)$$

Linked Lists!

Today's Goals

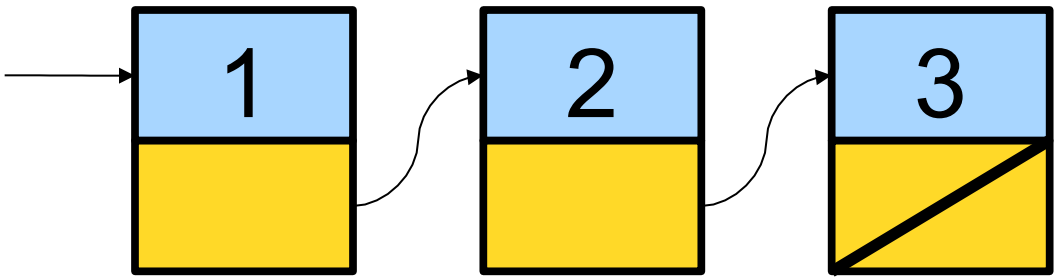


Today's Goals



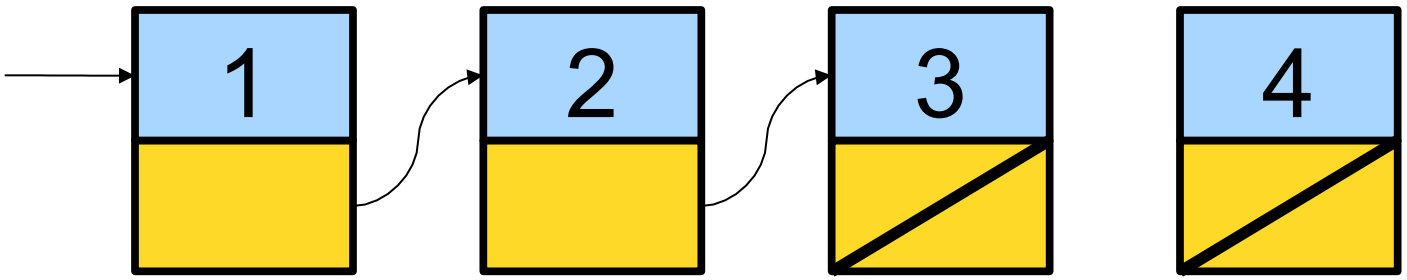
Linked Lists

- 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.



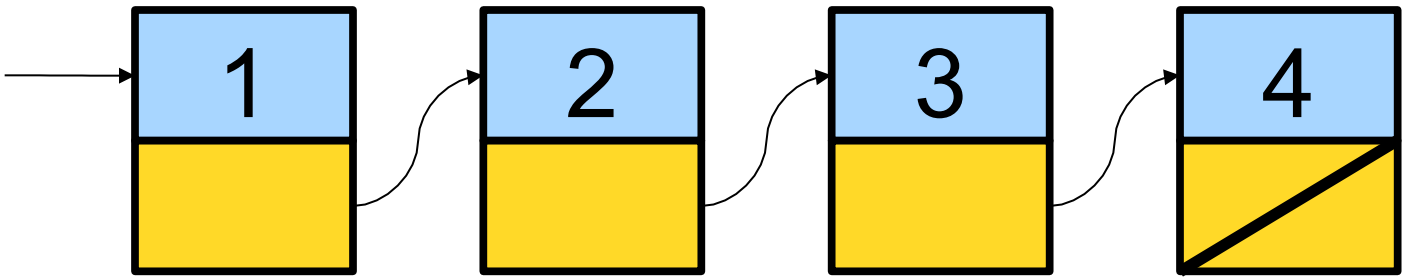
Linked Lists

- 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.



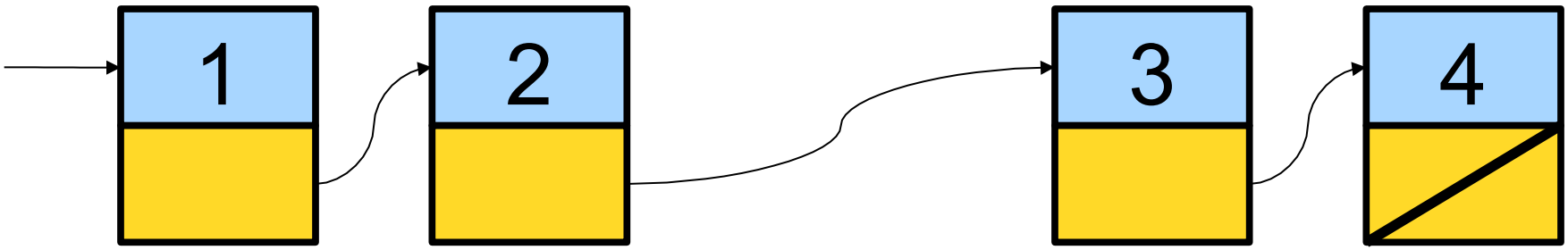
Linked Lists

- 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.



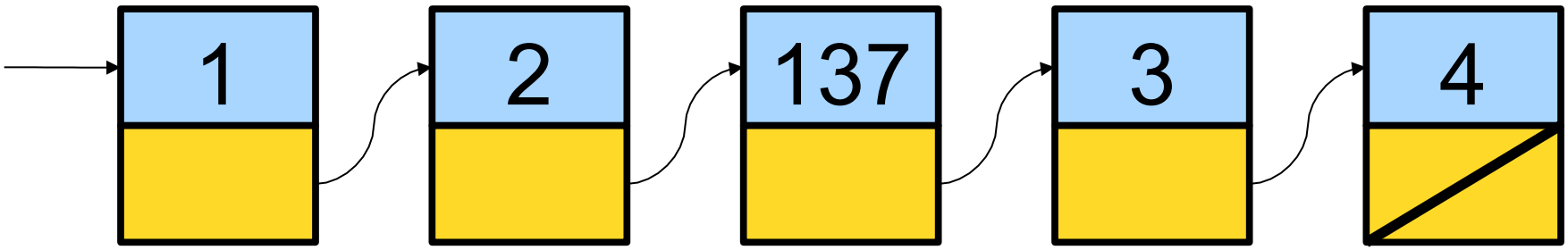
Linked Lists

- 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.



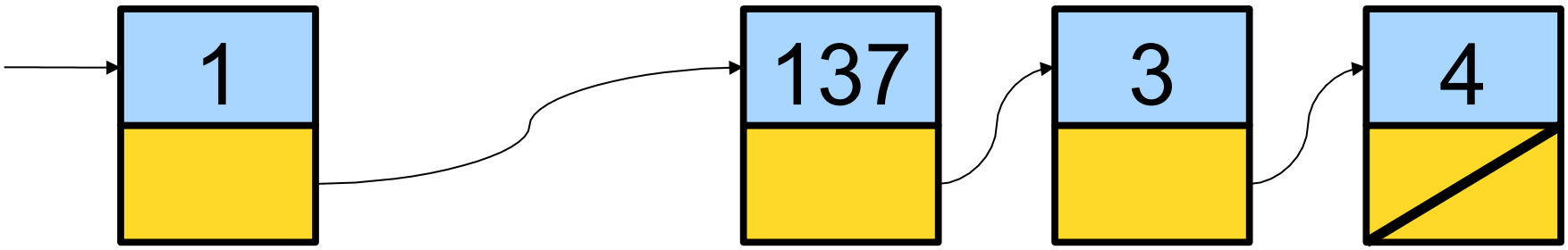
Linked Lists

- 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

- 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

- 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.

Linked Lists

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

- Structures
- Dynamic allocation
- Null pointers

Linked Lists

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

- Structures
- Dynamic allocation
- Null pointers

Structs

- 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.

Structs

- 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.

Structs

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

Structs

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

Tribute t;
```

Structs

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

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

Structs

- 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.

Structs

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

- Structures
- Dynamic allocation
- Null pointers

Structs

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

- Structures
- **Dynamic allocation**
- Null pointers

Structs

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.

Structs

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

Structs

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

Tribute* t = new Tribute;
```

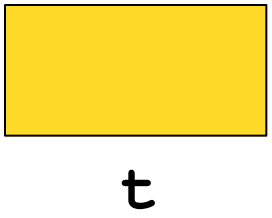
Structs

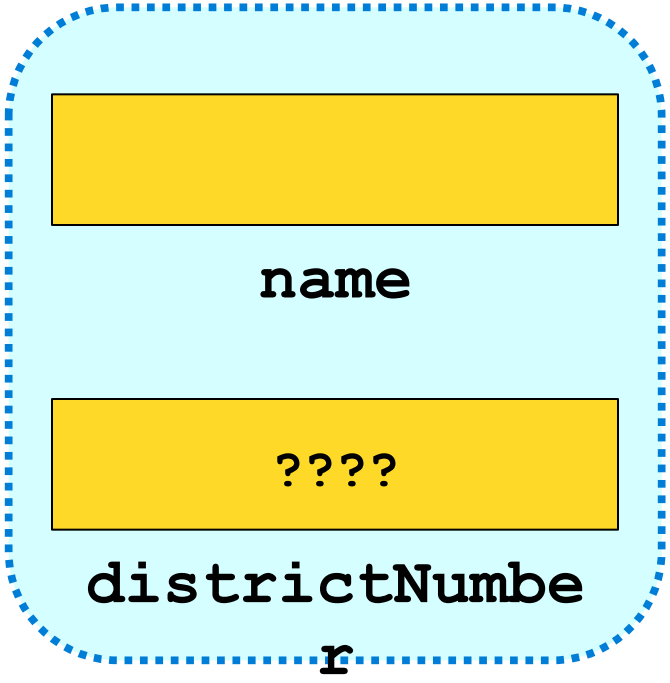
```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```

t

Structs

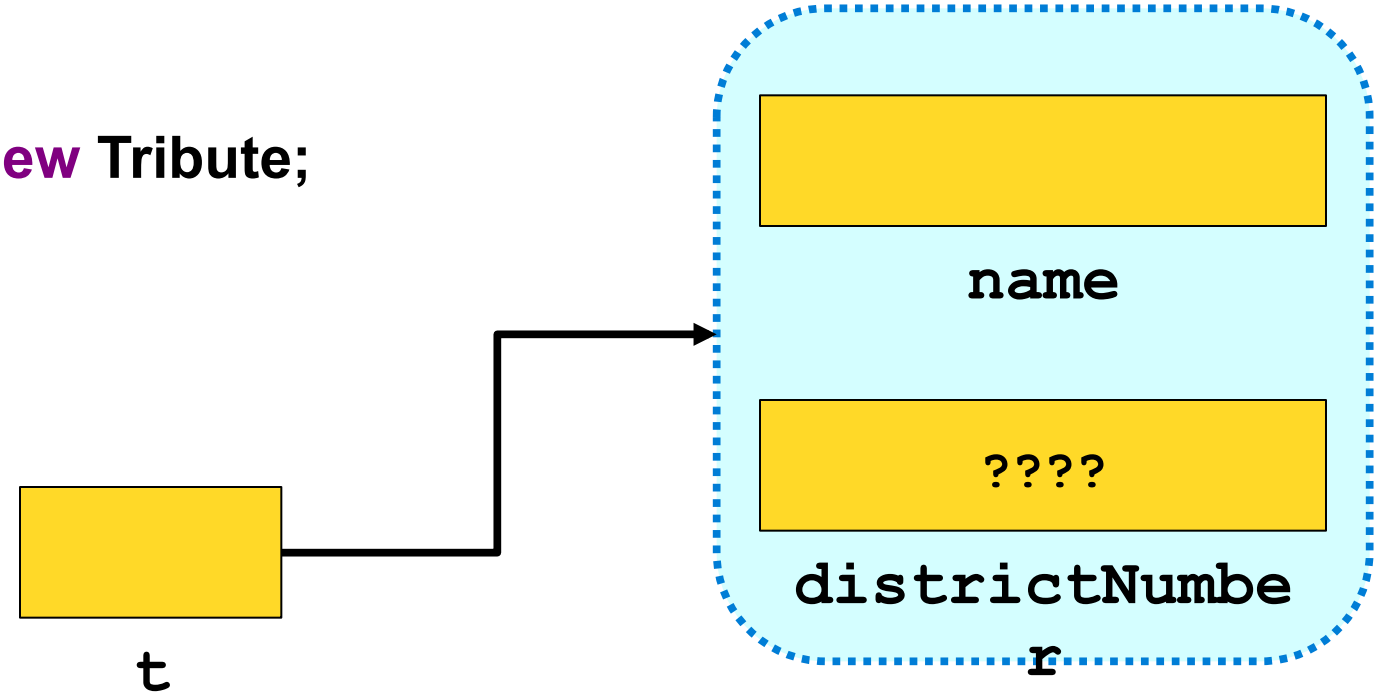
```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```





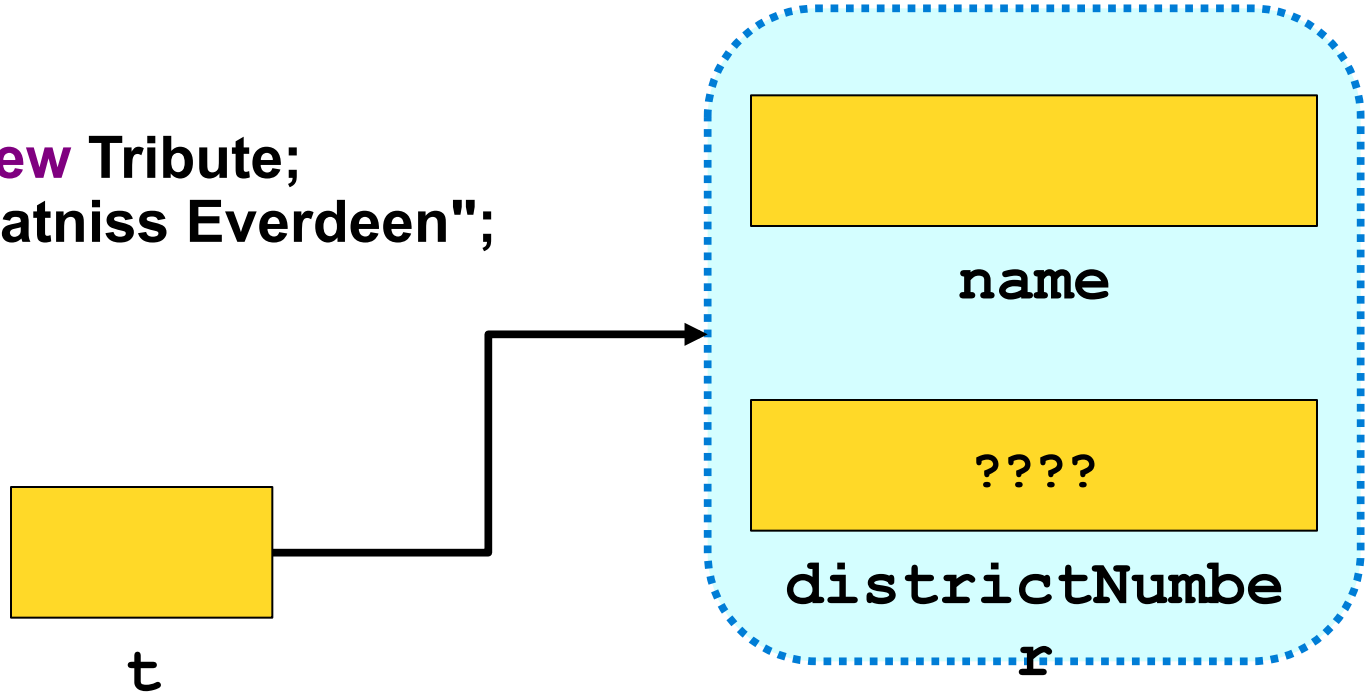
Structs

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

Tribute* t = new Tribute;
```



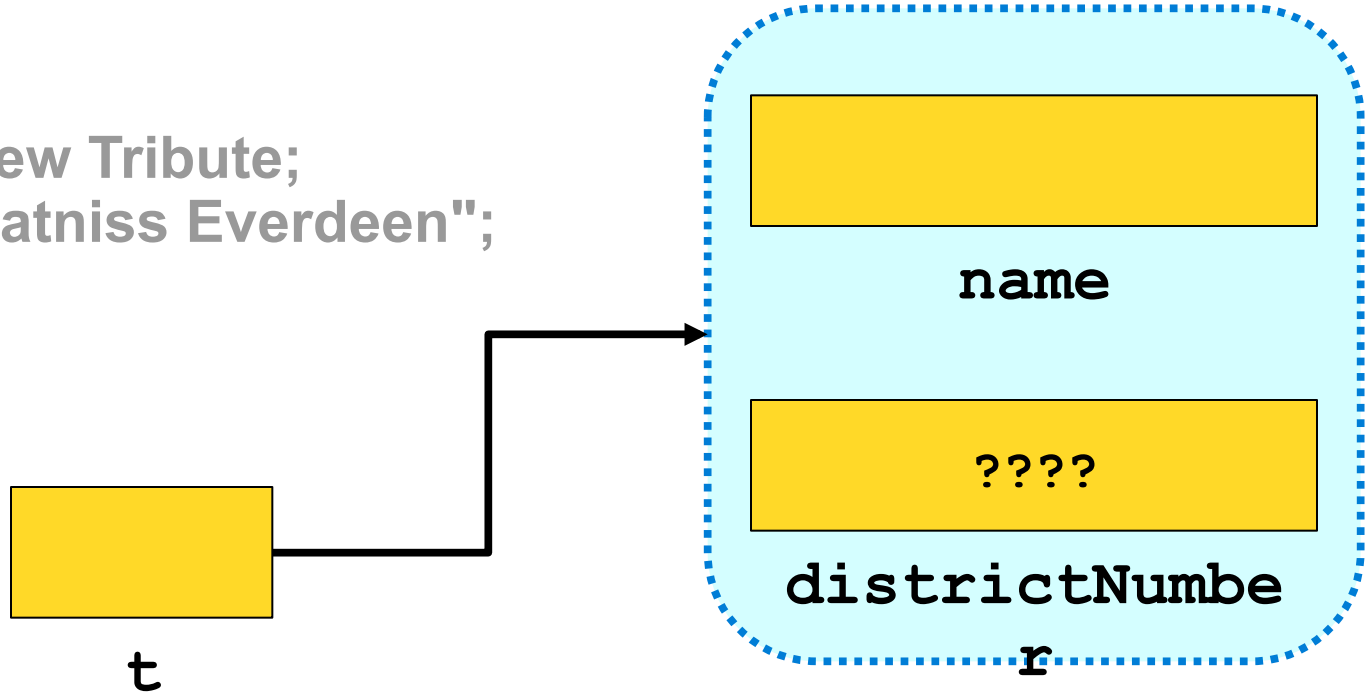
Structs

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

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



Structs

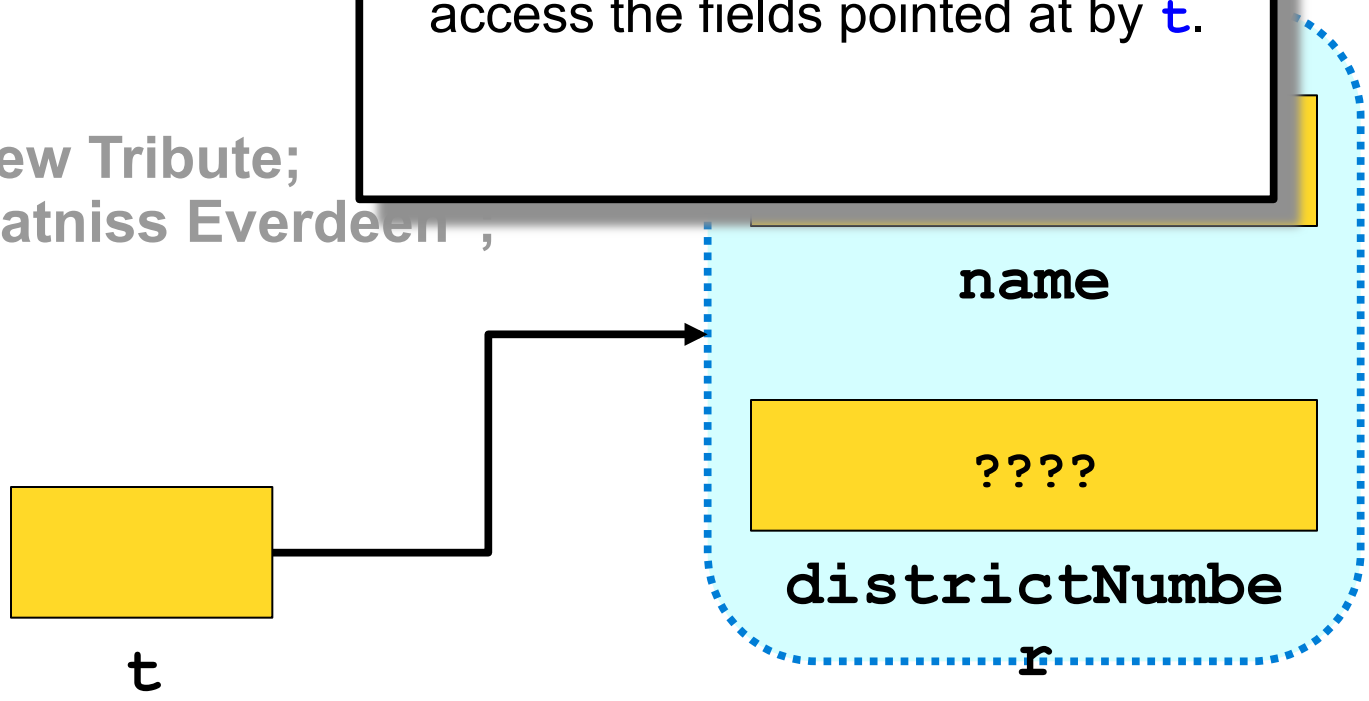
```
struct Tribute {
    string name;
    int districtNumber;
};

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



Structs

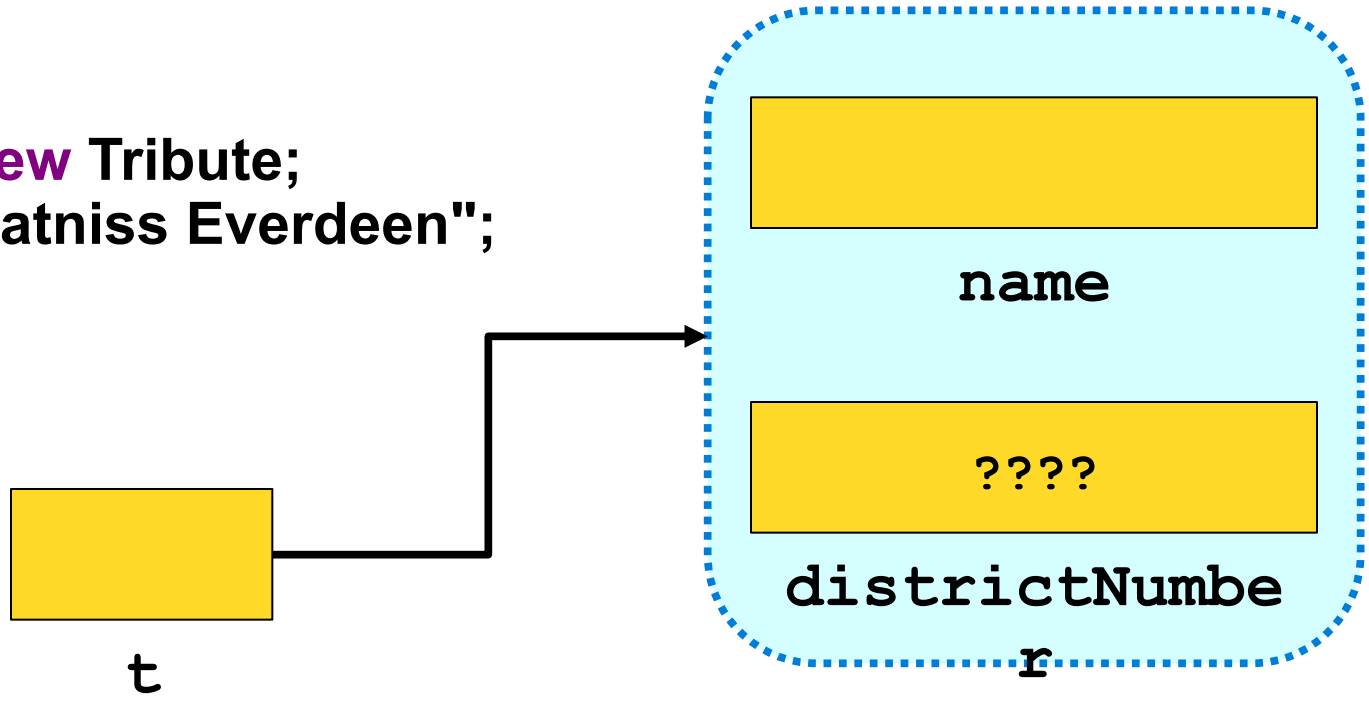
```
struct Tribute {
    string name;
    int districtNumber;
};

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`.



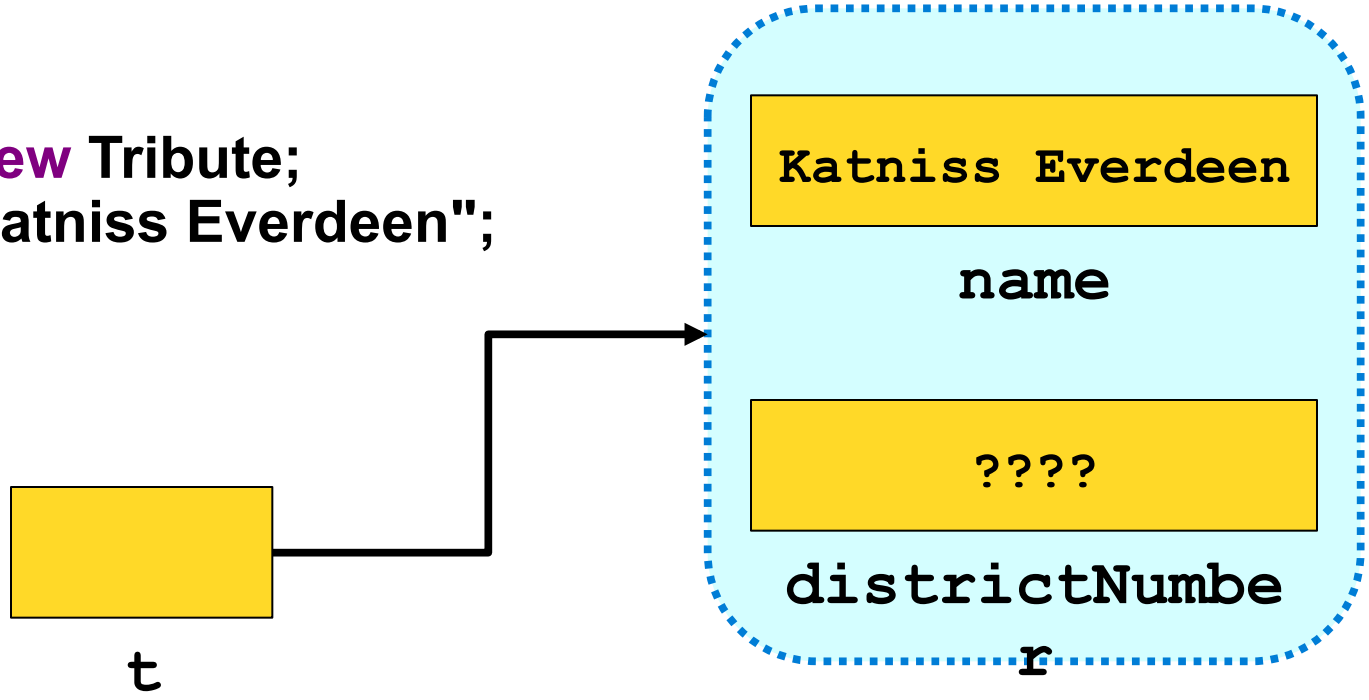
Structs

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

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



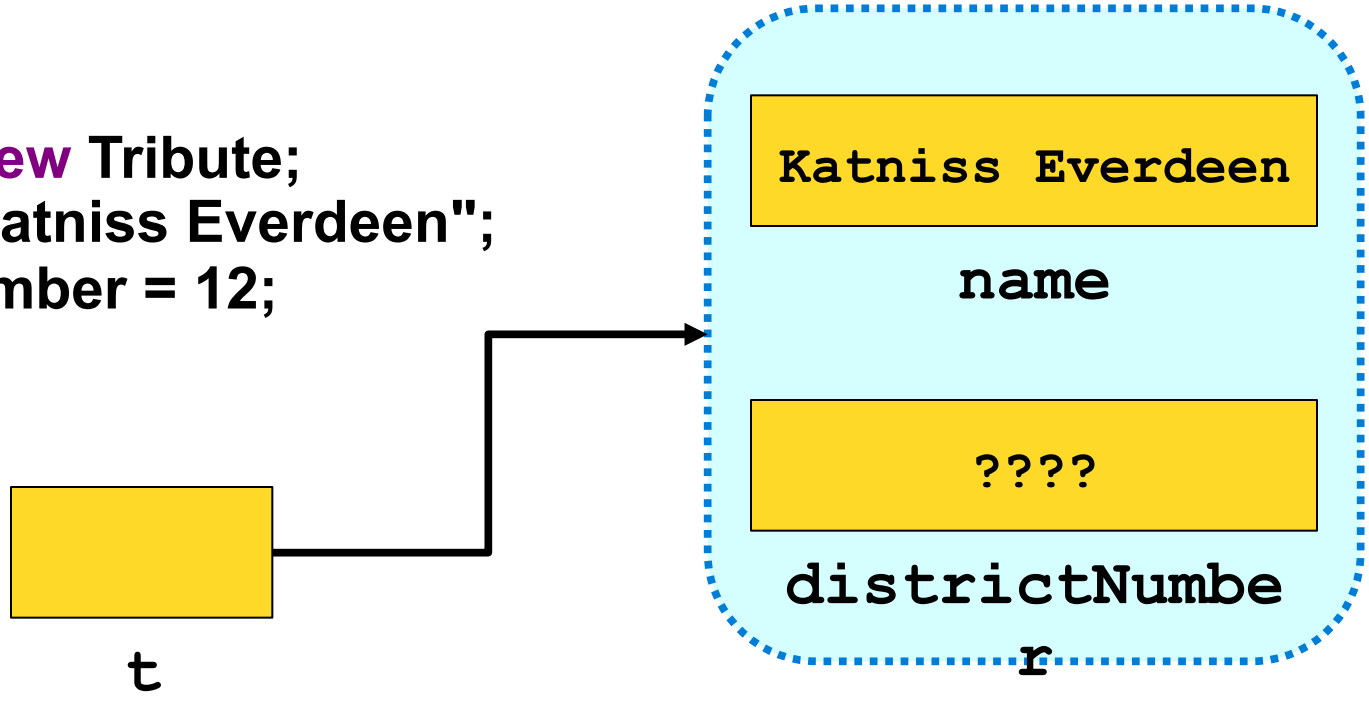
Structs

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

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



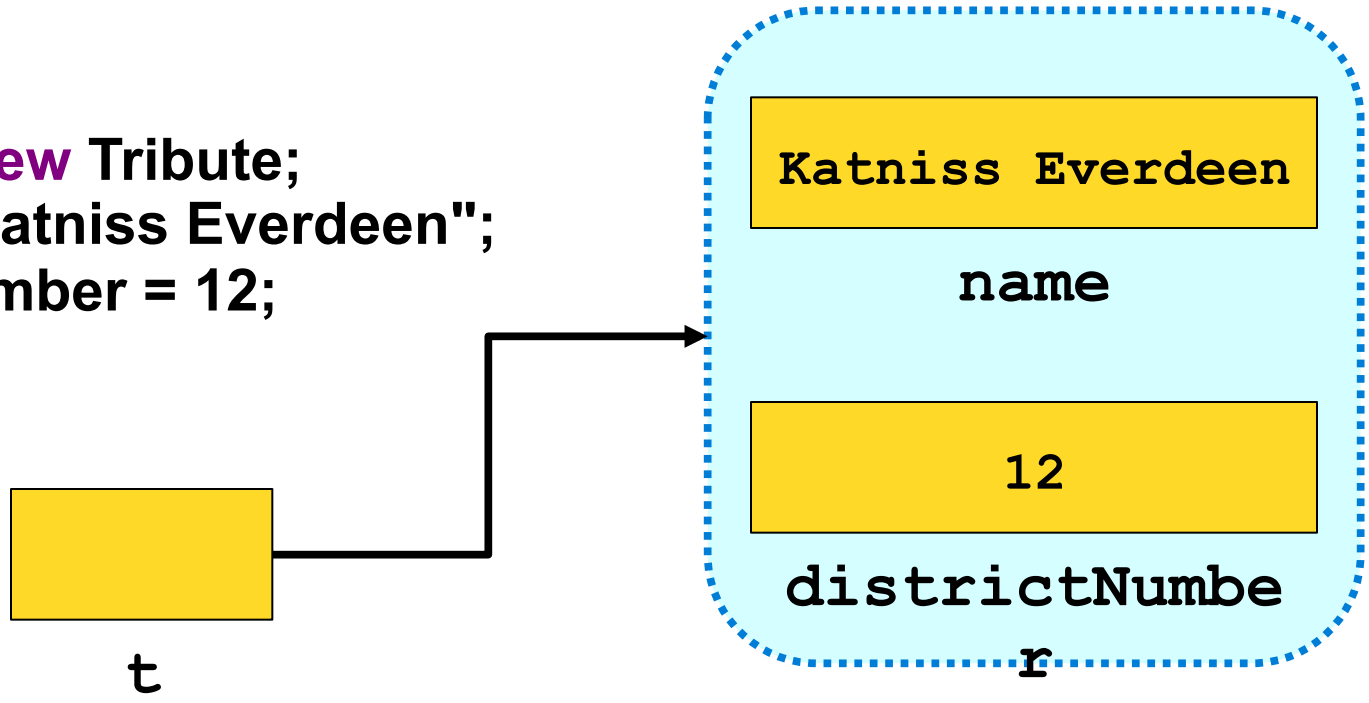
Structs

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

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



Structs

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

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



Structs

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

delete *ptr*;

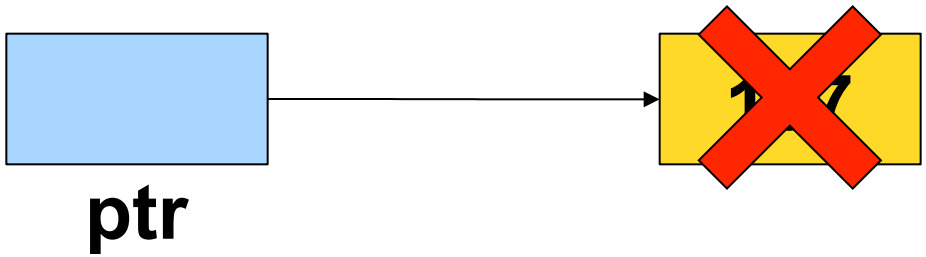
- This destroys the object pointed at by the given pointer, not the pointer itself.

Structs

- 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.



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**

The Null Pointer

- 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 Lists

- 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.

Linked Lists

- 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;
};
```

Linked Lists

- 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;
};
```

Linked Lists

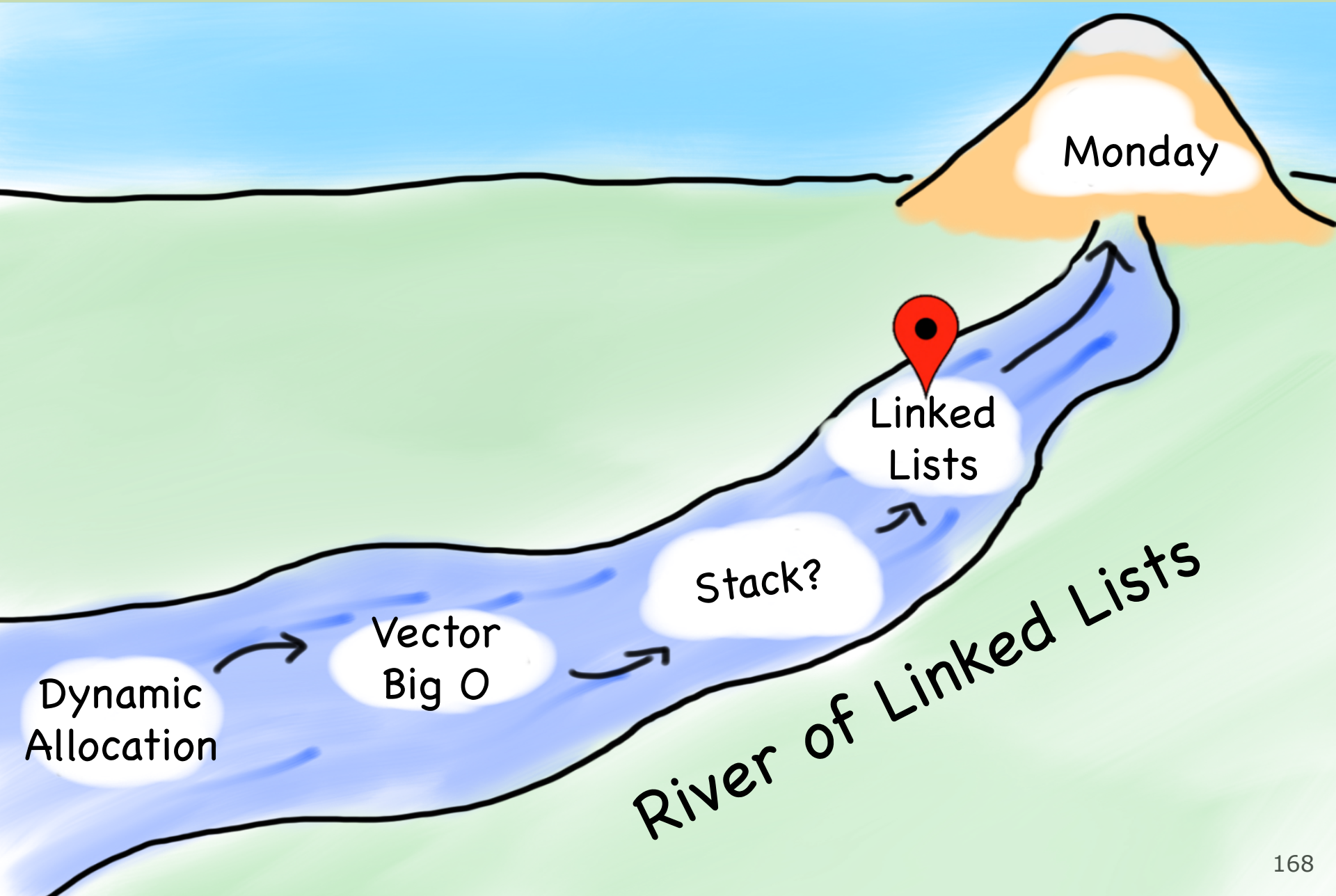
- For simplicity, let's assume we're building a linked list of **`string`**s.
- We can represent a cell in the linked list as a structure:

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

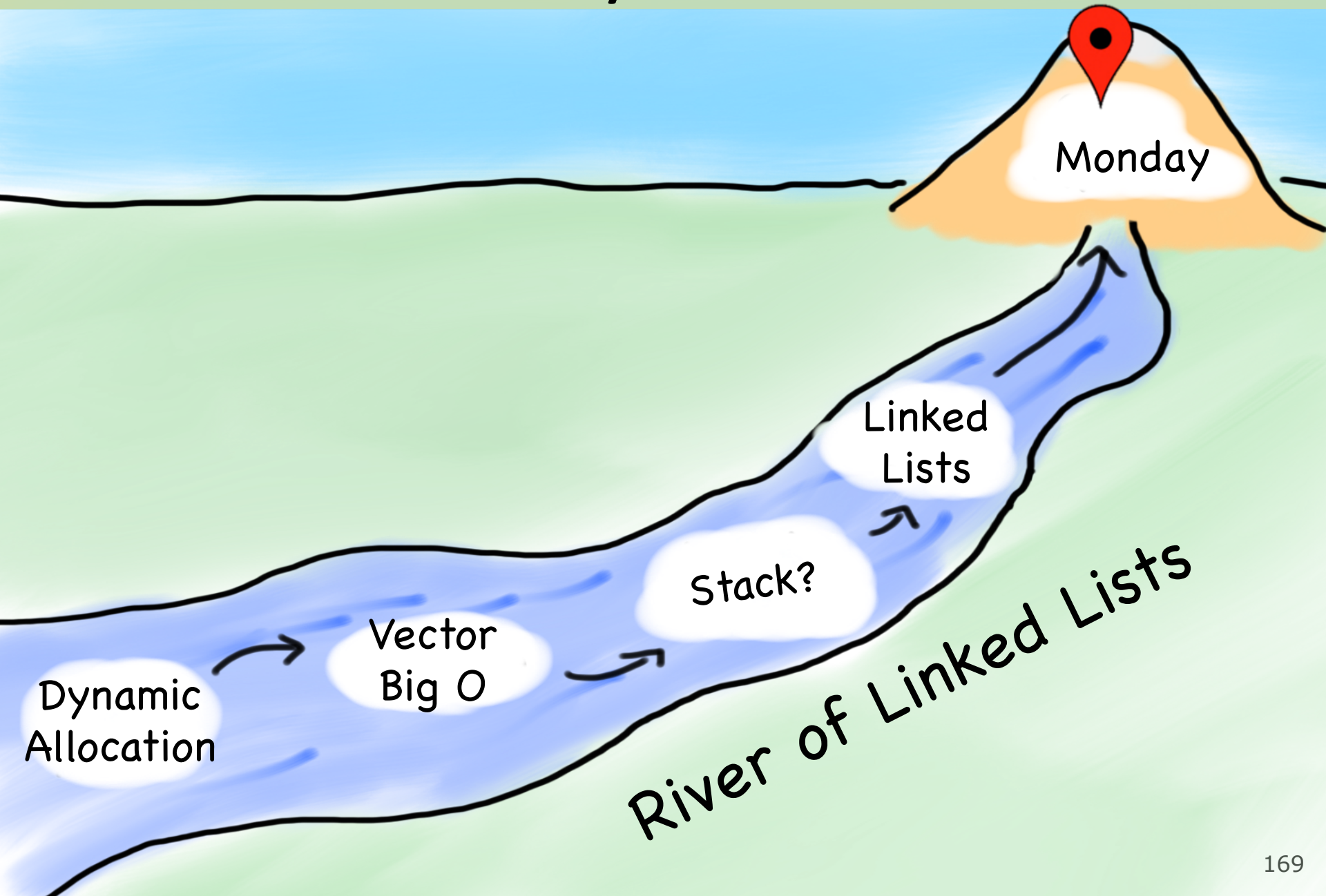
- **The structure is defined recursively!**

Building Linked Lists!

Today's Goals



Today's Goals



Today's Goals

1. Practice with dynamic allocation
2. Introduction to linked lists