Linked Lists

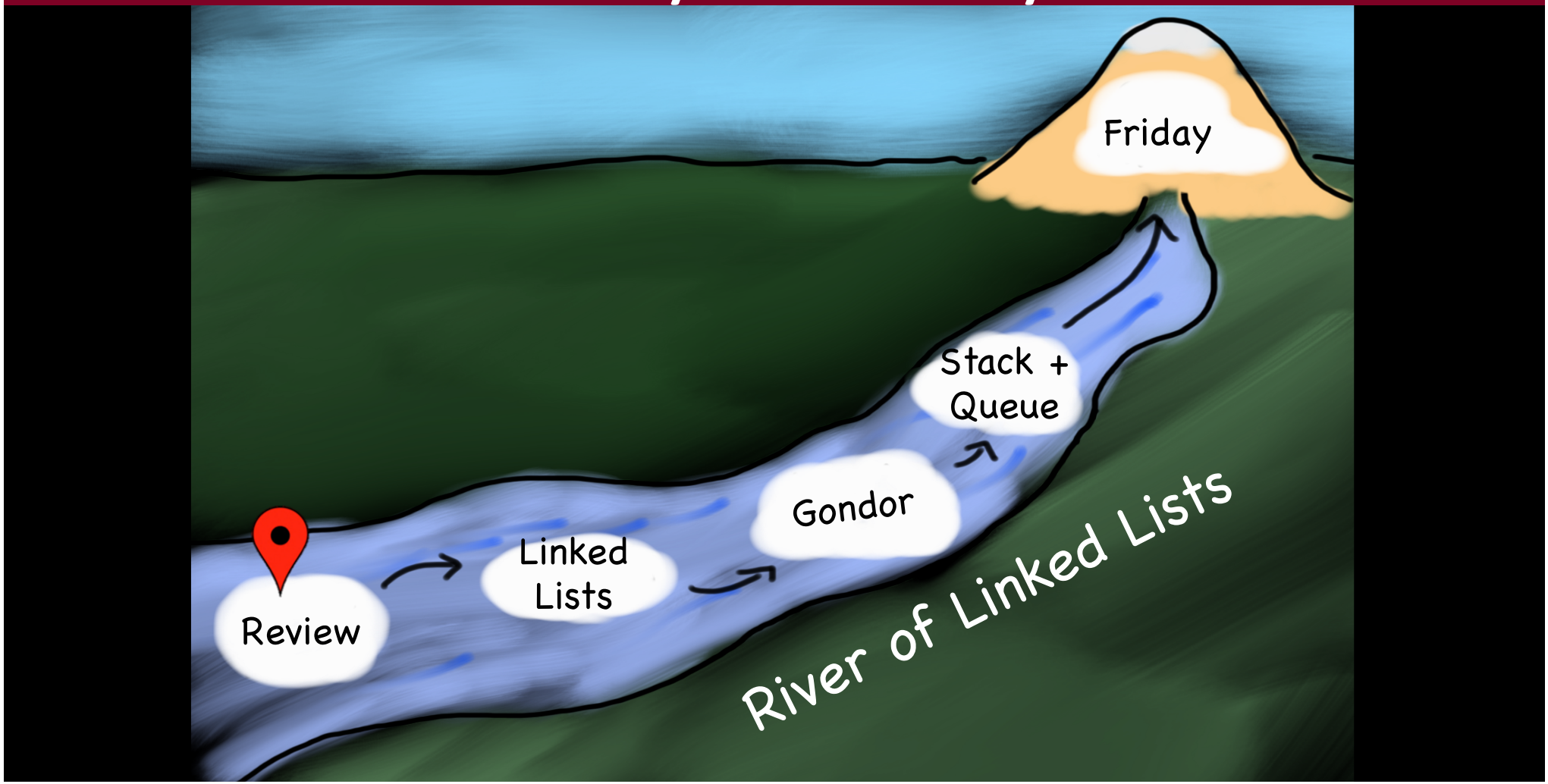
CS 106B

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department

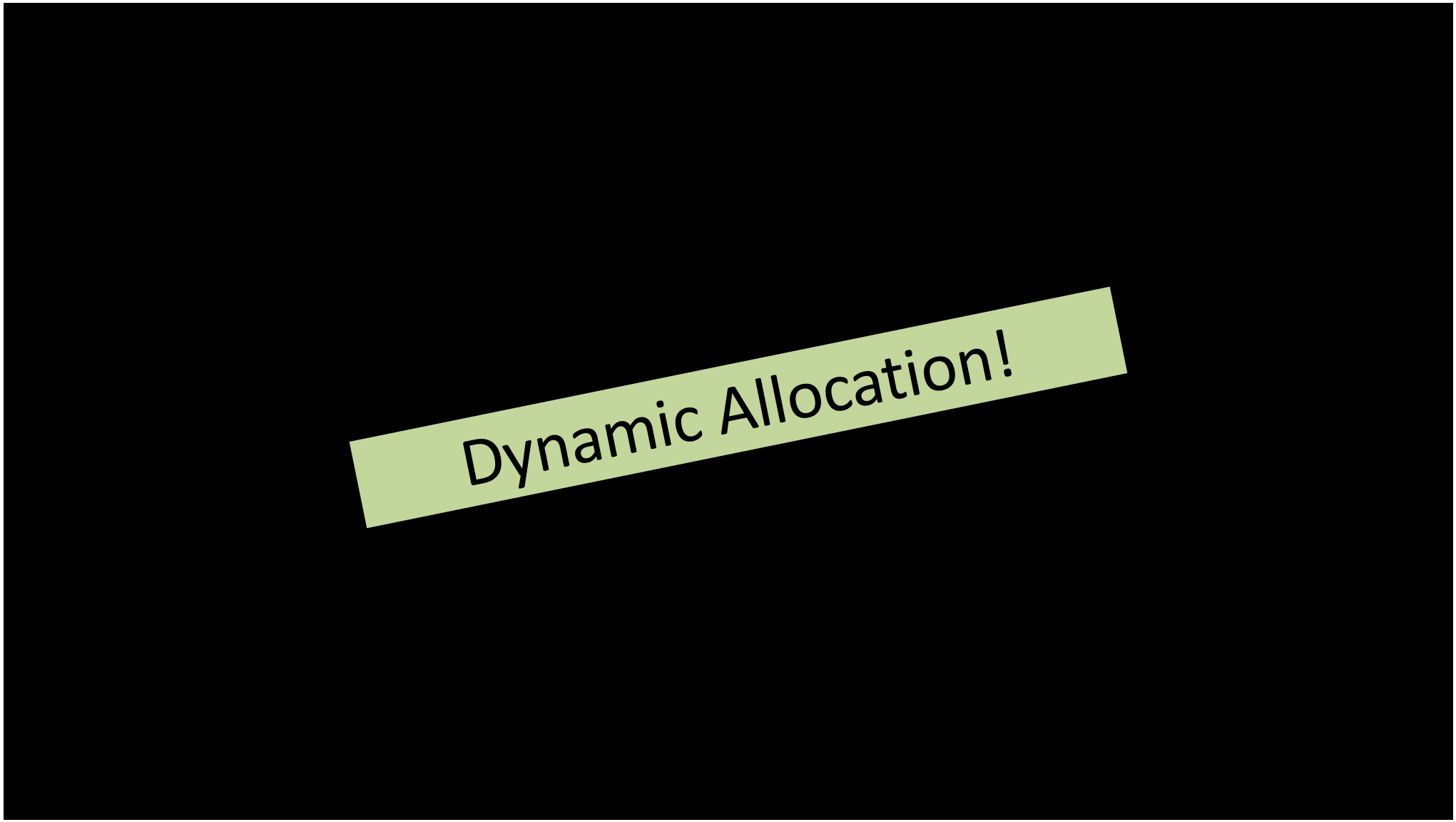
Today's Goals

1. Learn linked lists
2. See how Stack + Queue work
3. Get ready for trees…

Today's Journey



Announcement


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");
```

Dynamic Allocation

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

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

Dynamic Allocation

```
// 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.
```

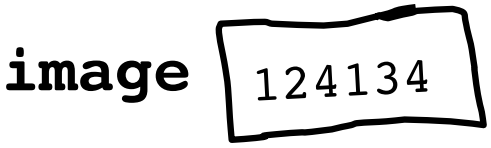
124134

```
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");
```



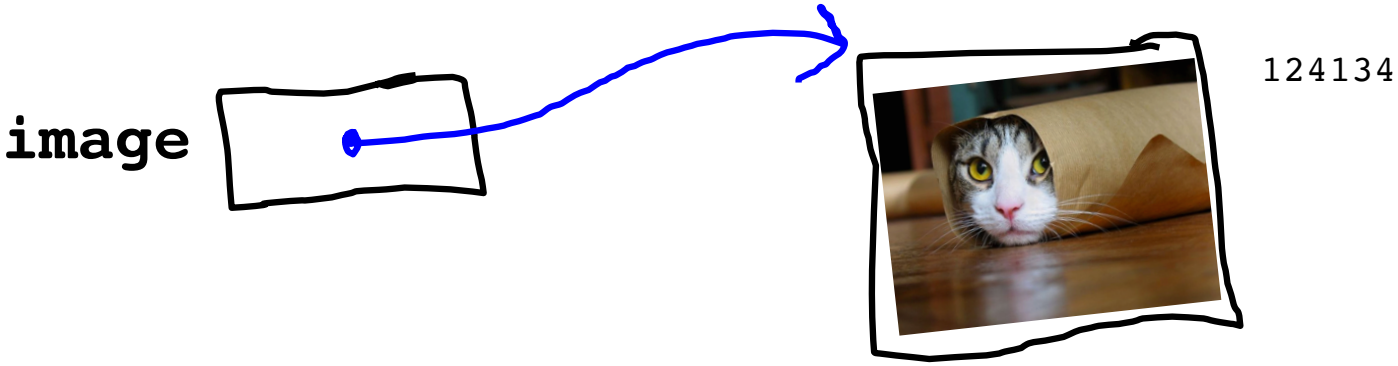
124134

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");
```



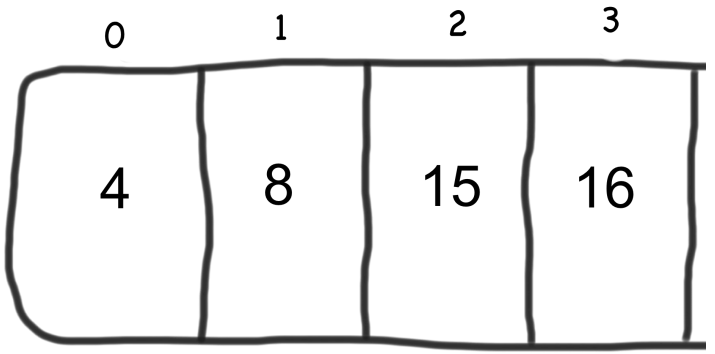
Pointers

Its easy to share large objects.

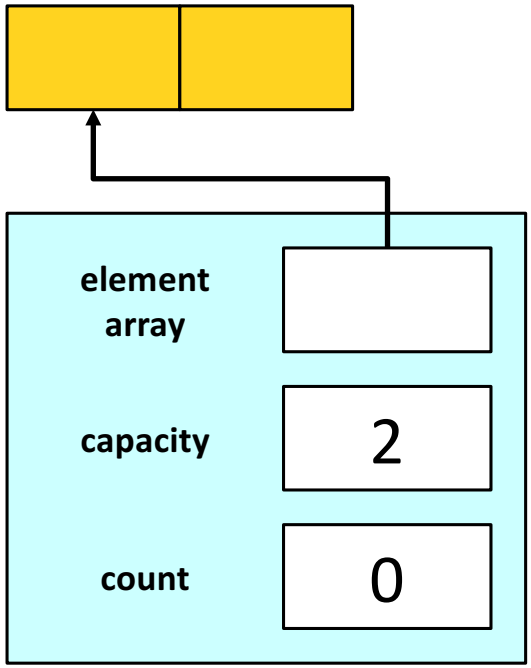
A pointer is like a URL. Not the actual page, the address of the page

Control over variable lifespan.

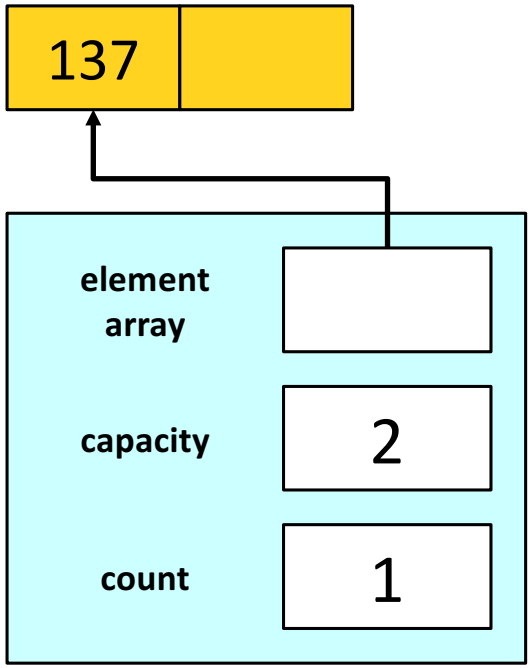
While the program is running, you can request more space.



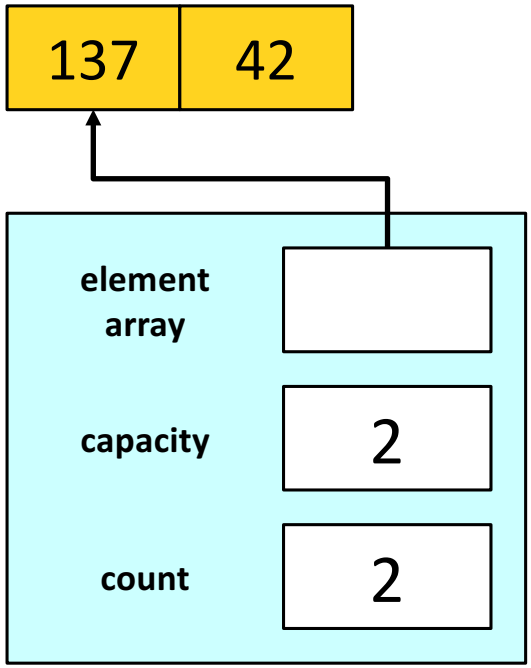
The Actual Vector



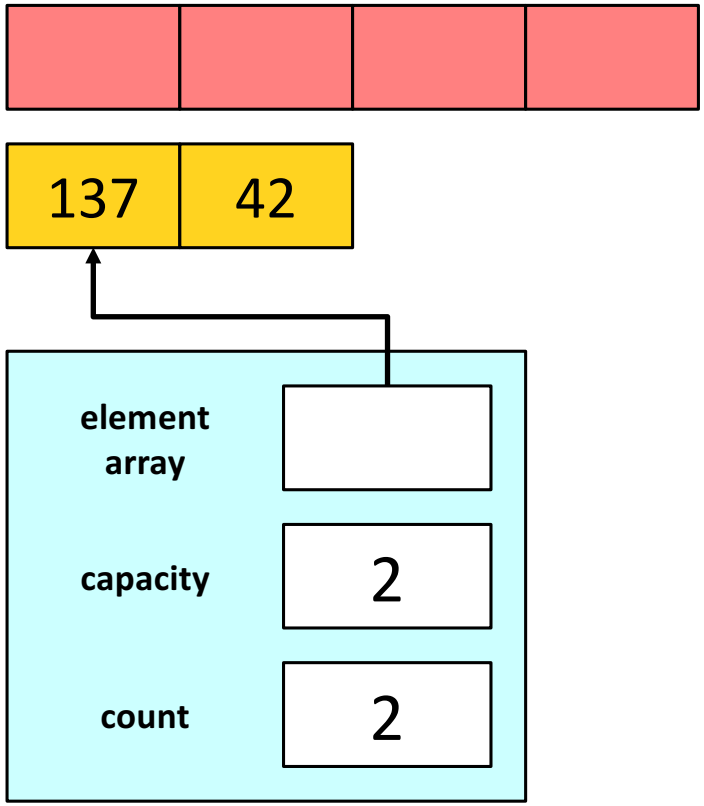
The Actual Vector



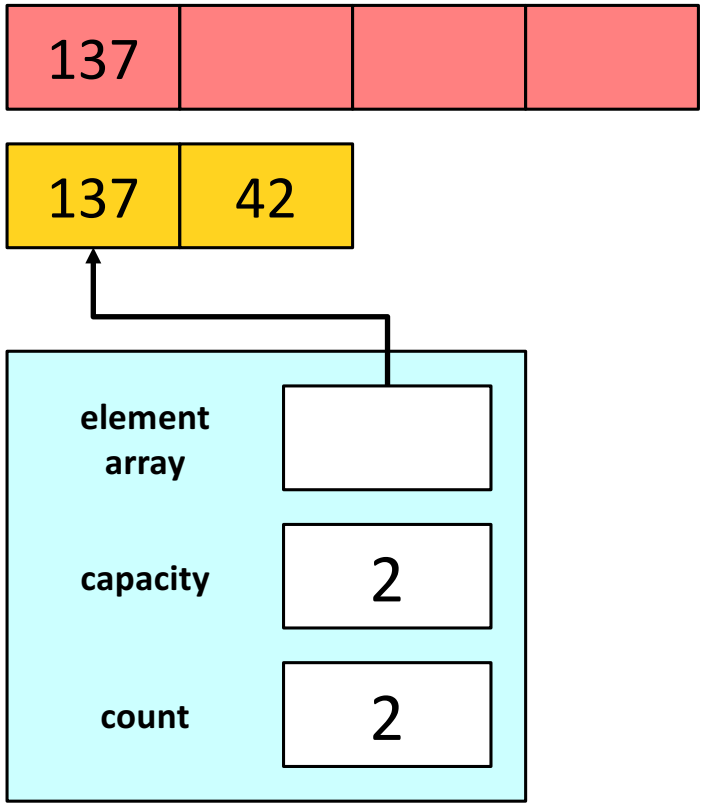
The Actual Vector



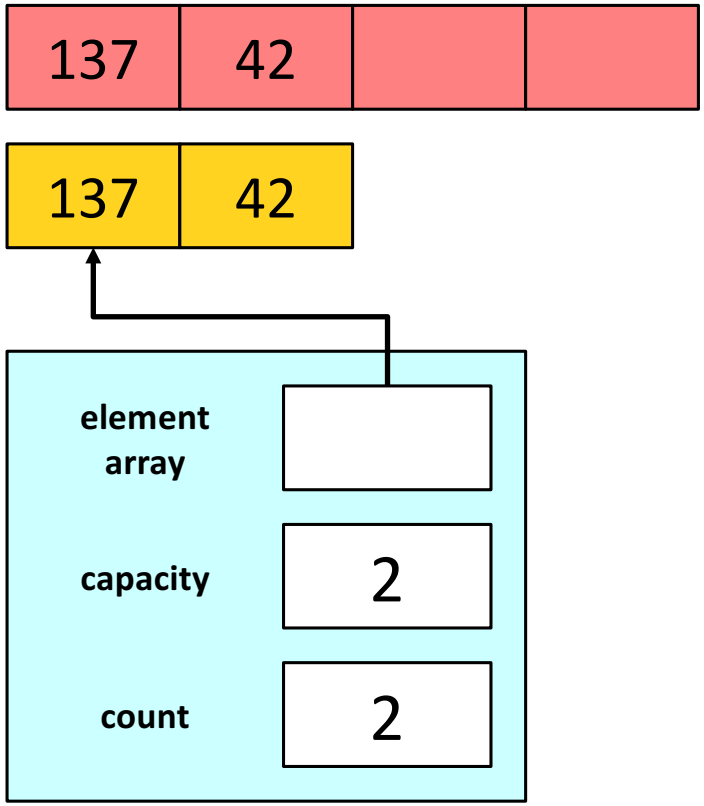
The Actual Vector



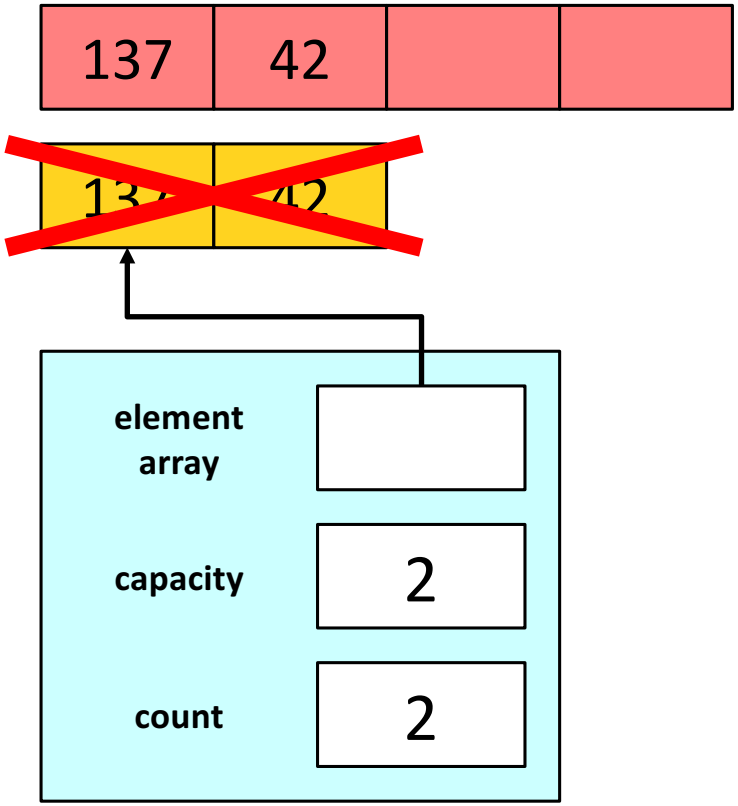
The Actual Vector



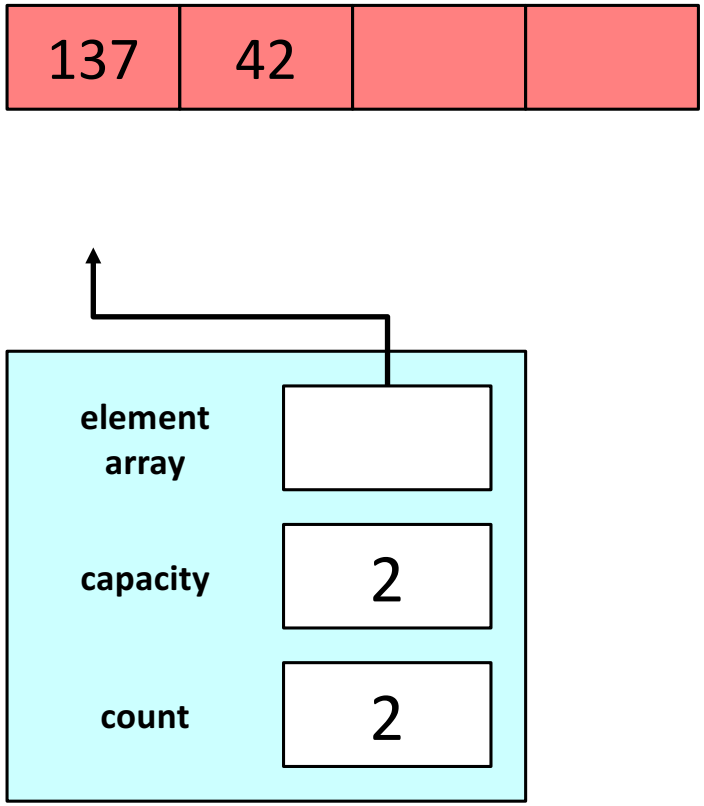
The Actual Vector



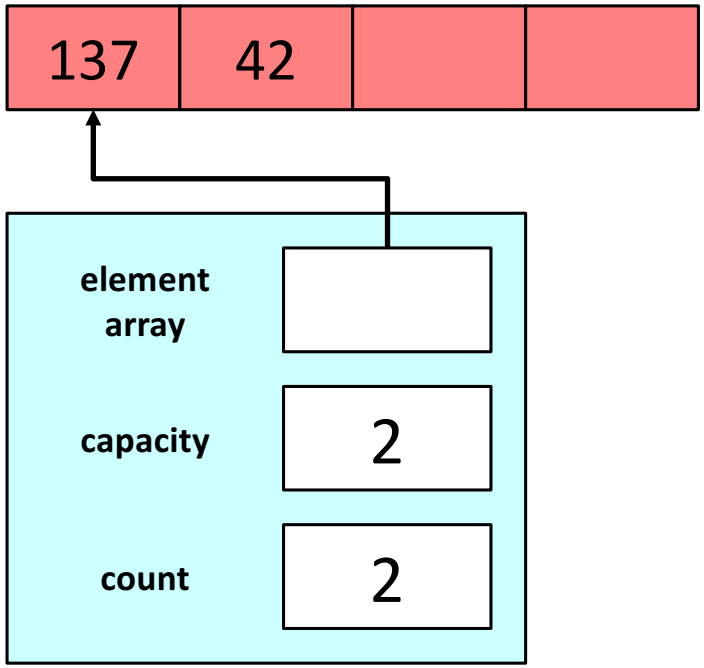
The Actual Vector



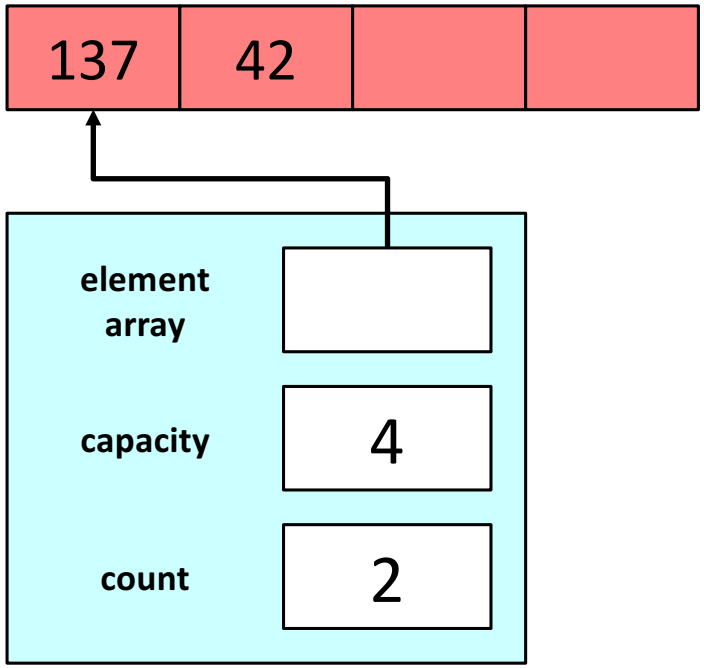
The Actual Vector



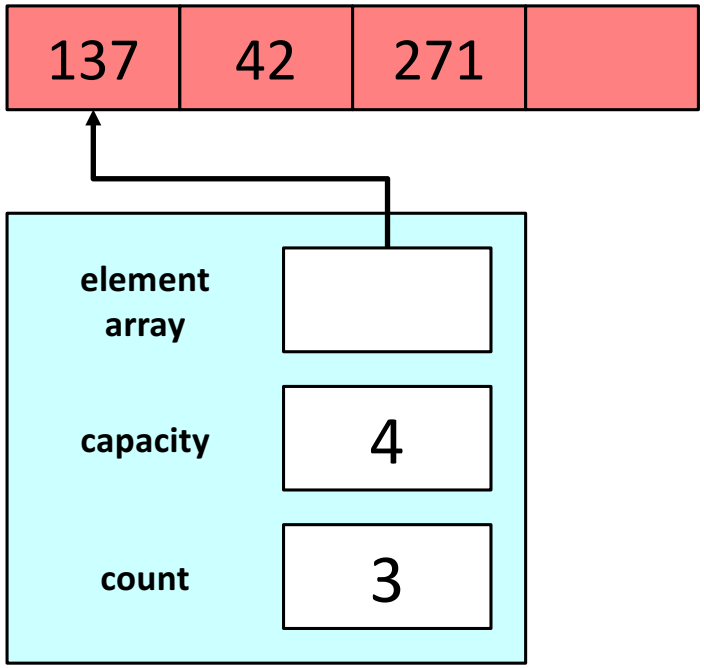
The Actual Vector



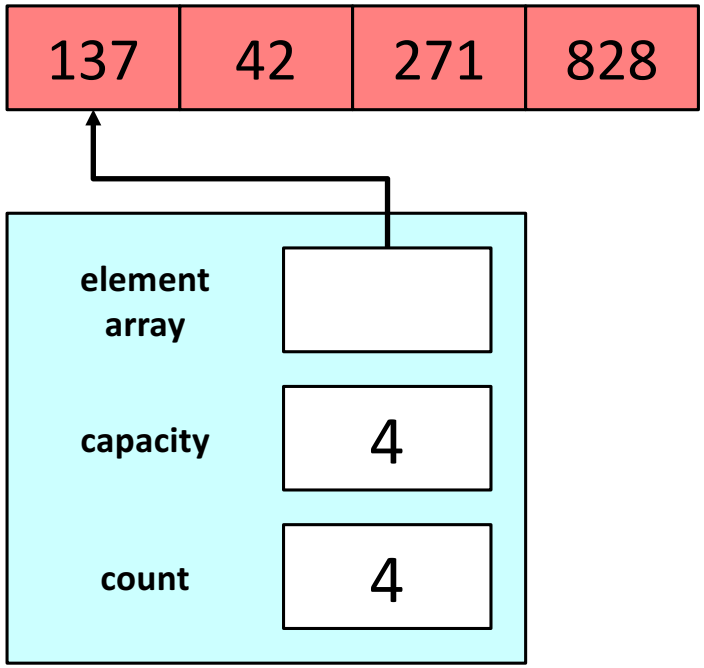
The Actual Vector



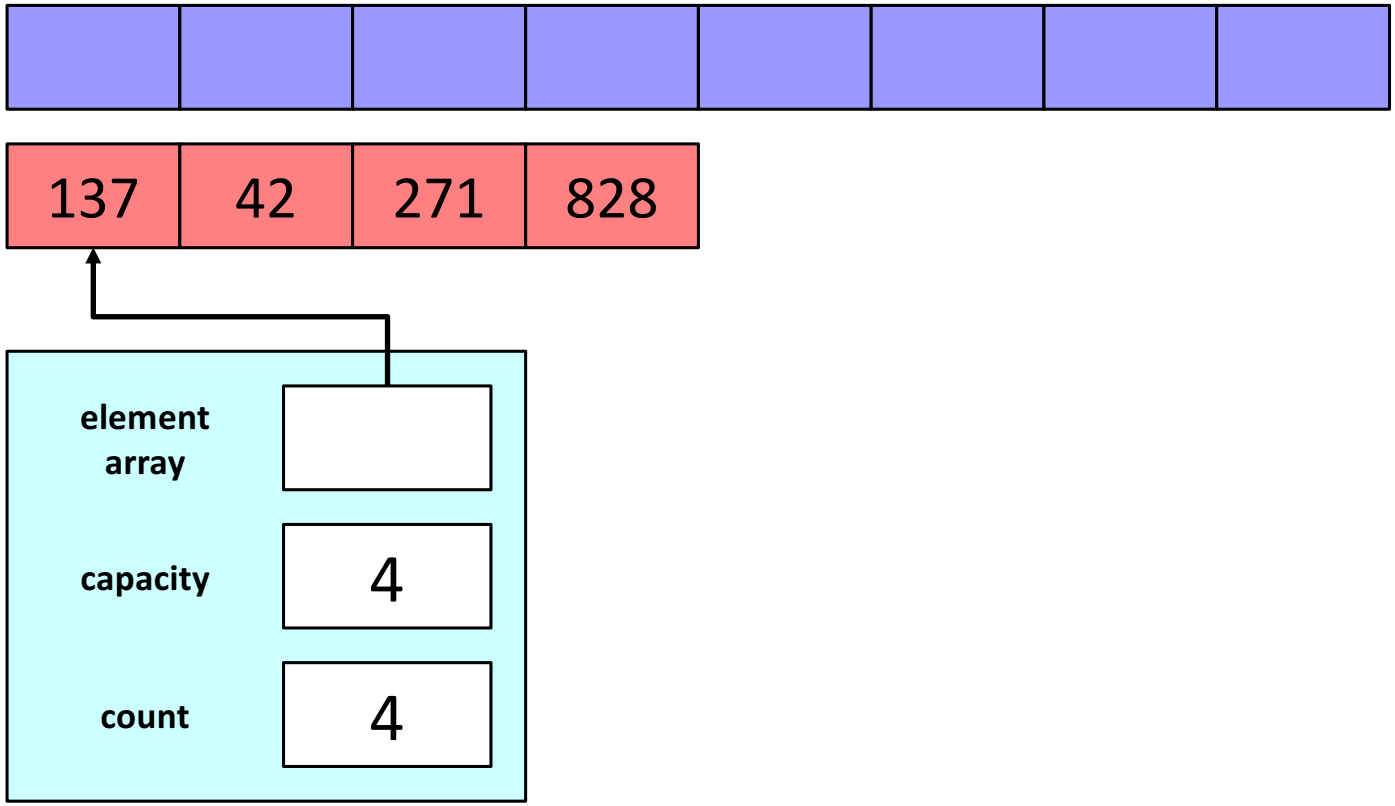
The Actual Vector



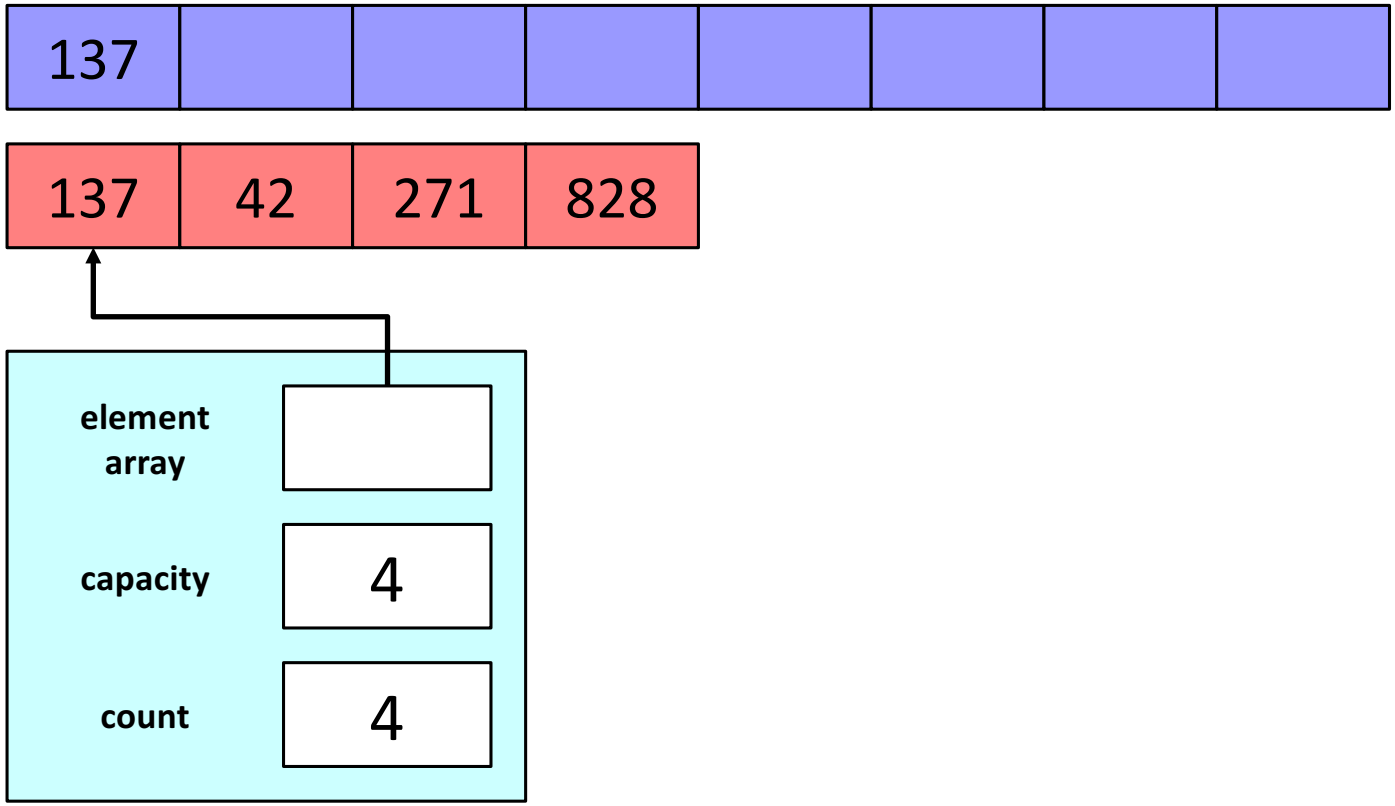
The Actual Vector



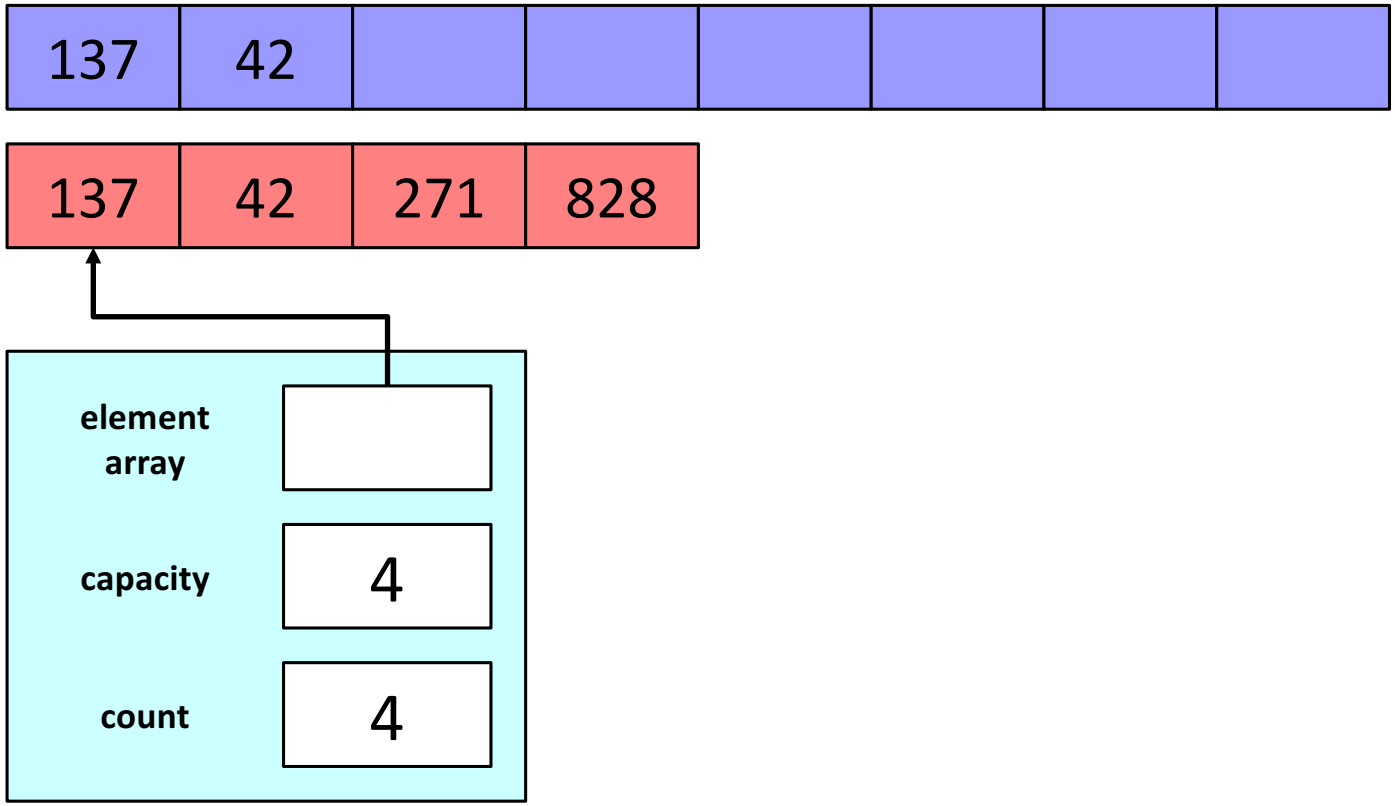
The Actual Vector



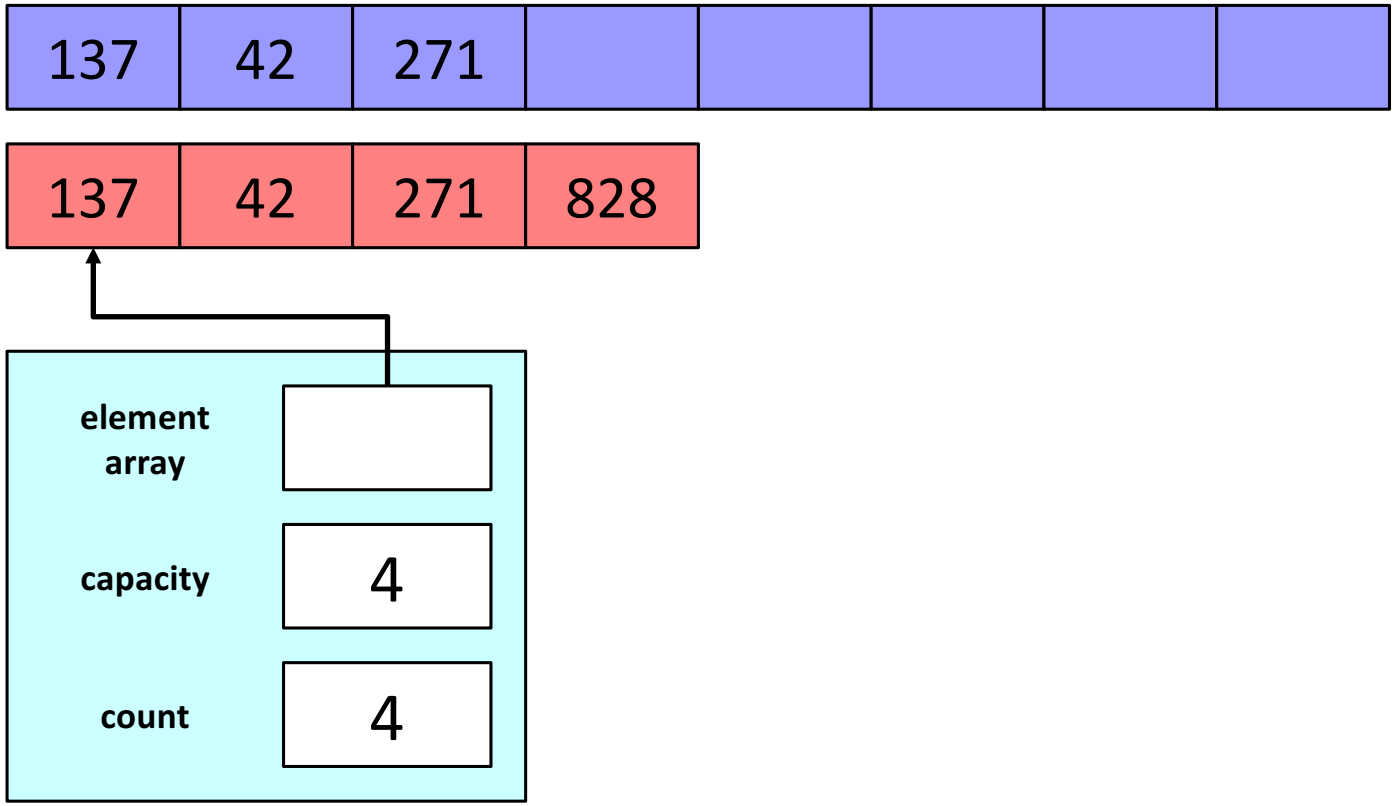
The Actual Vector



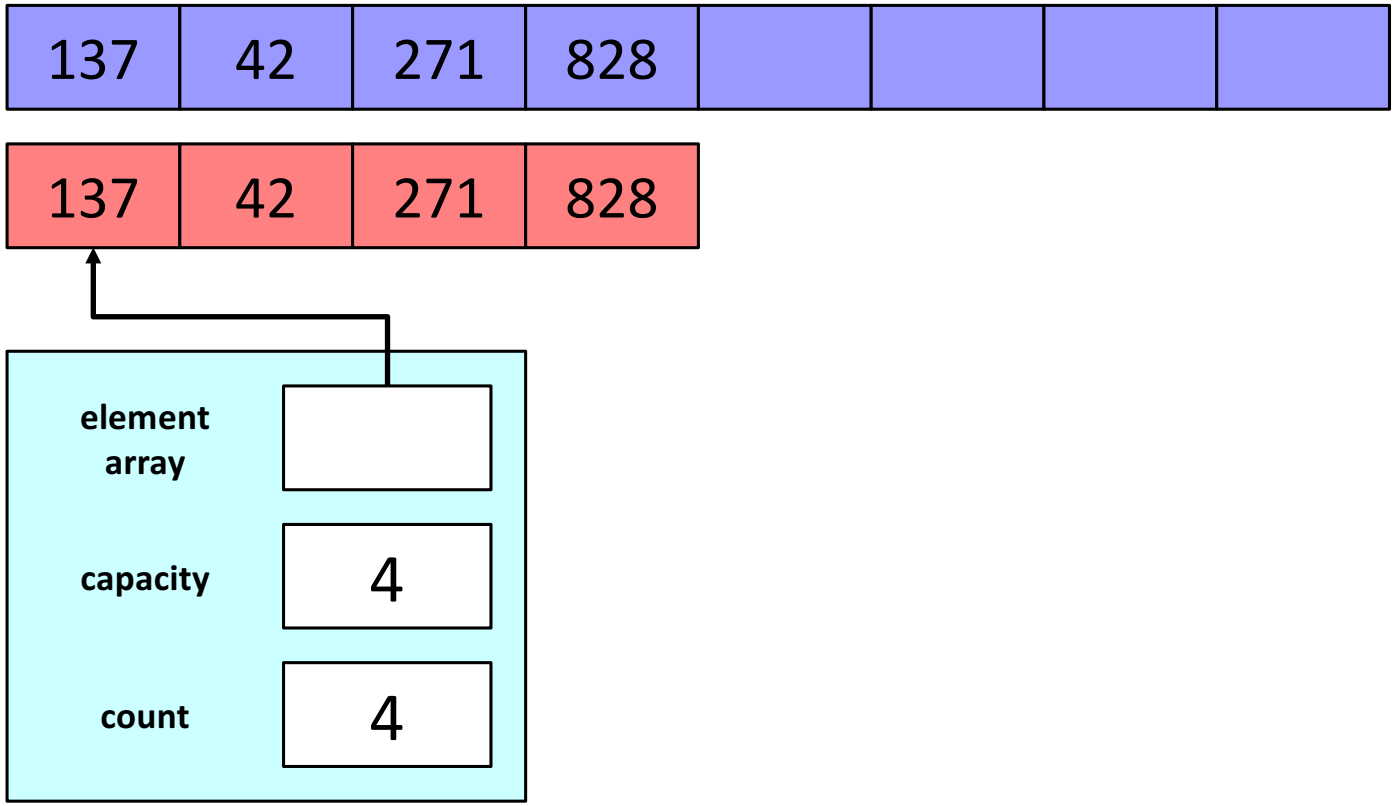
The Actual Vector



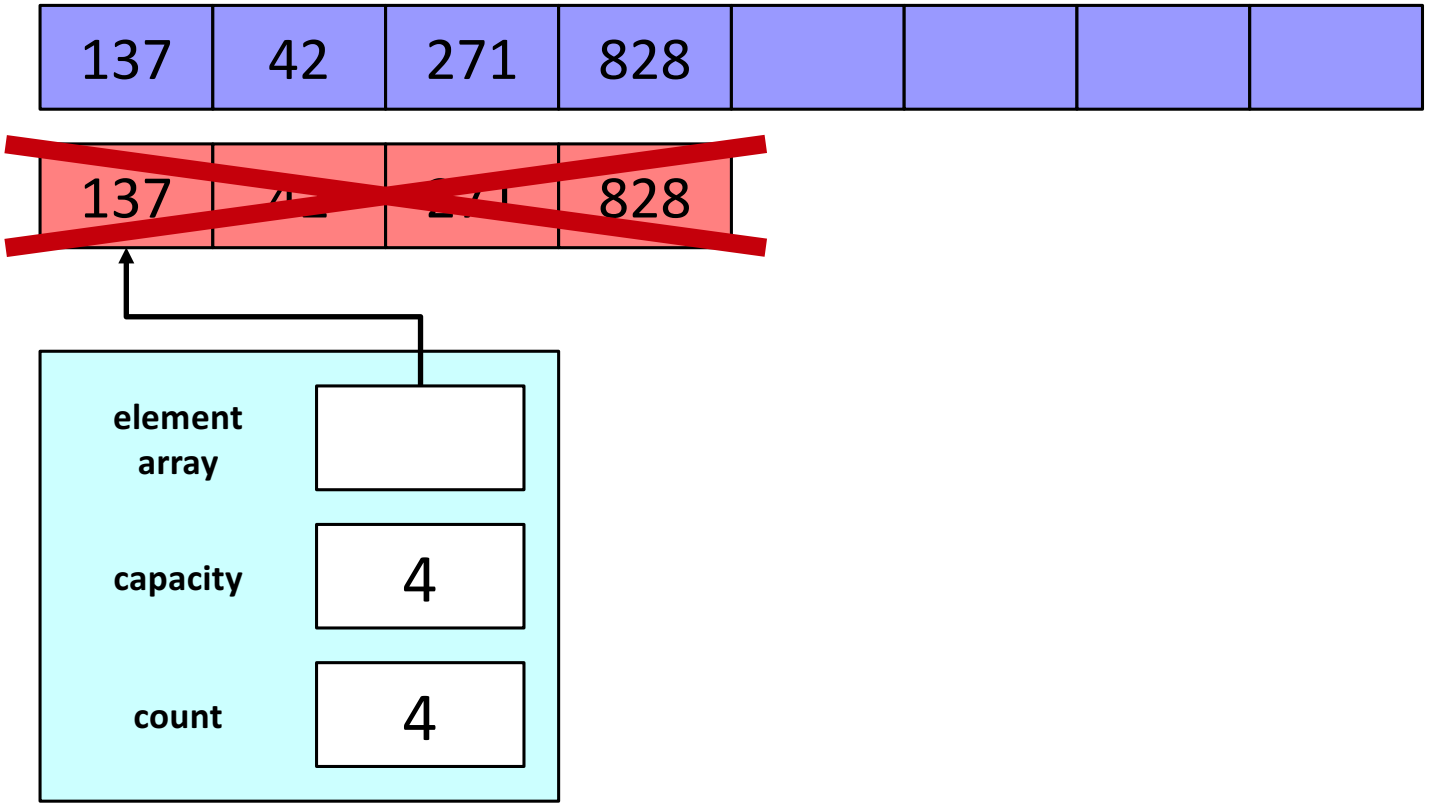
The Actual Vector



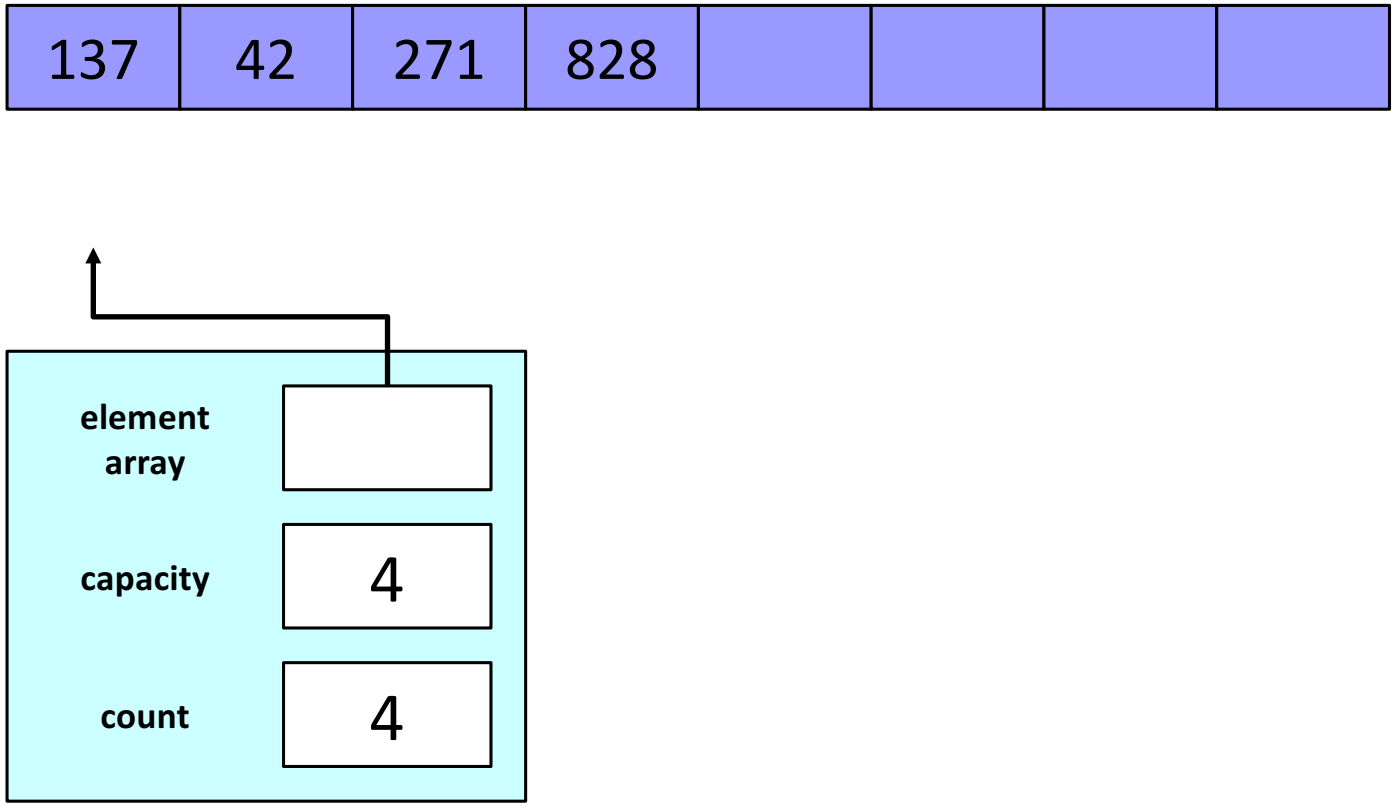
The Actual Vector



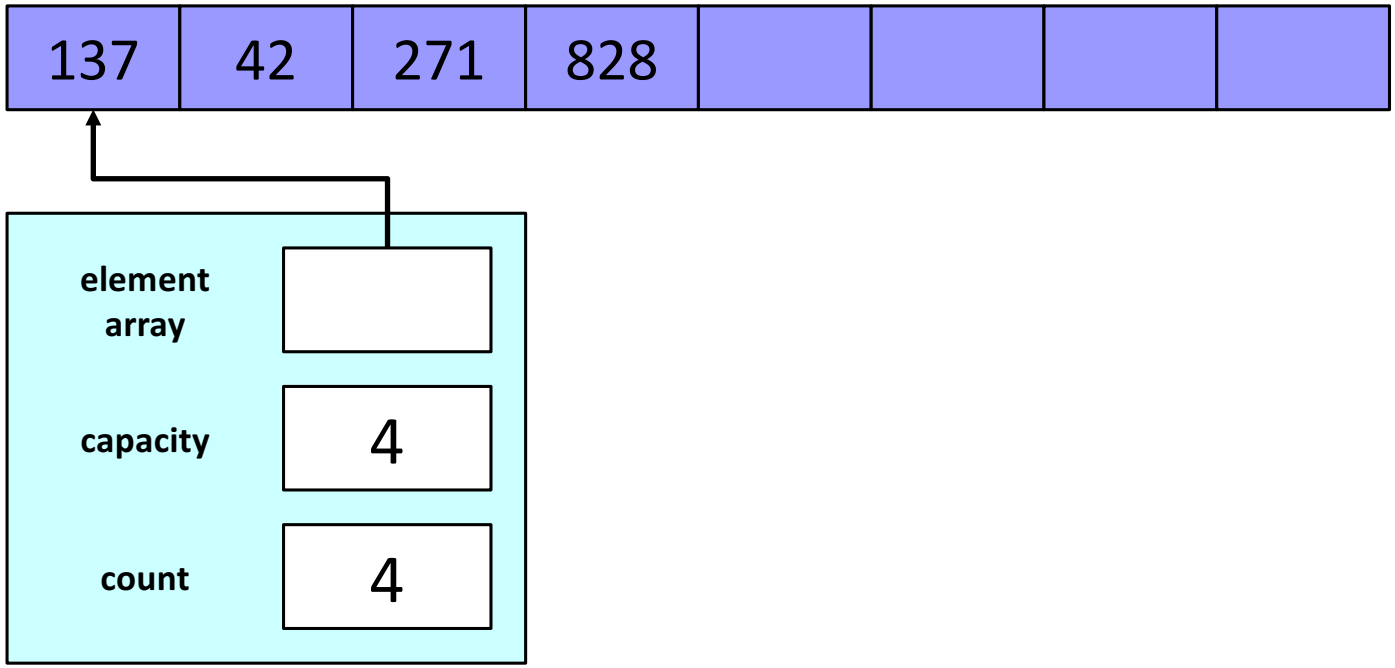
The Actual Vector



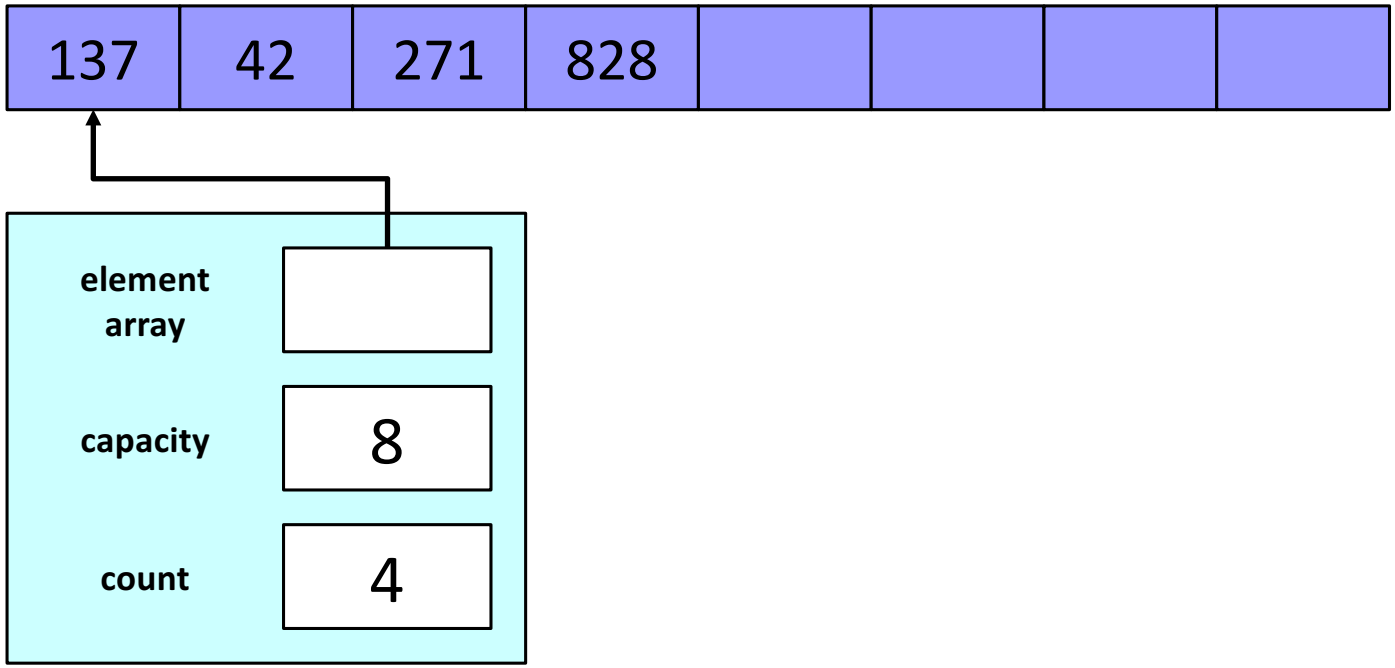
The Actual Vector



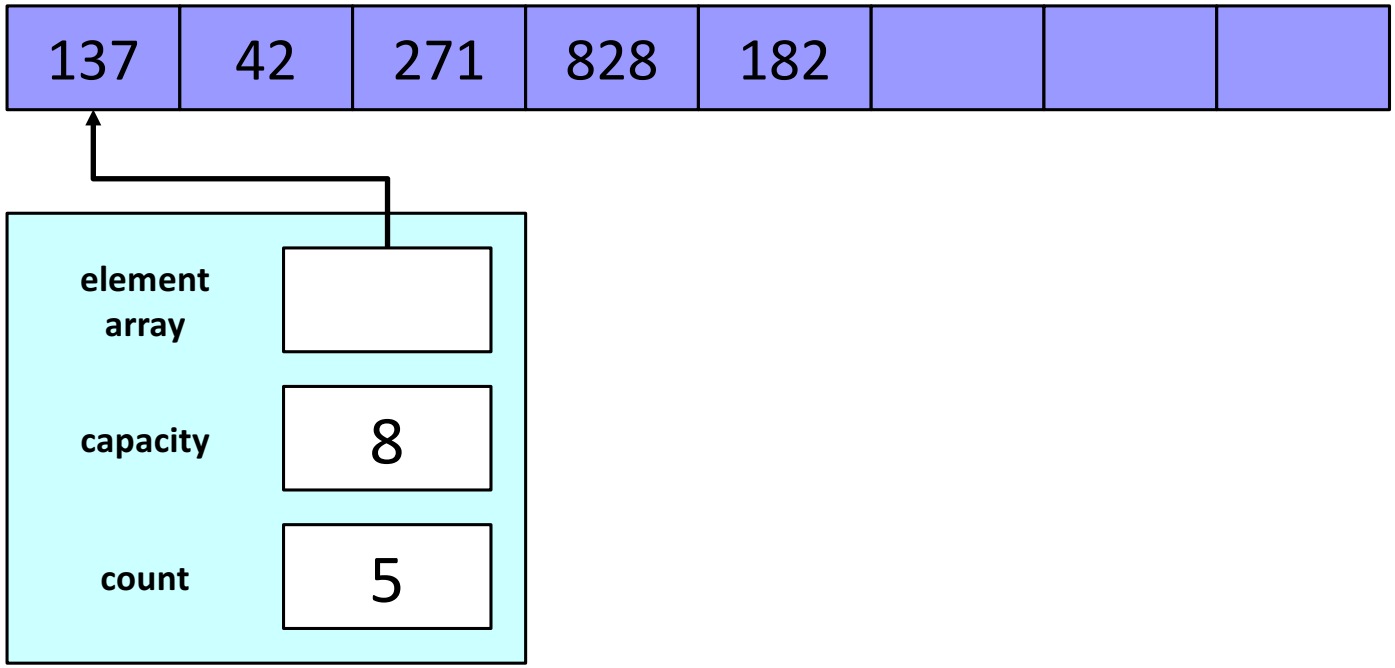
The Actual Vector



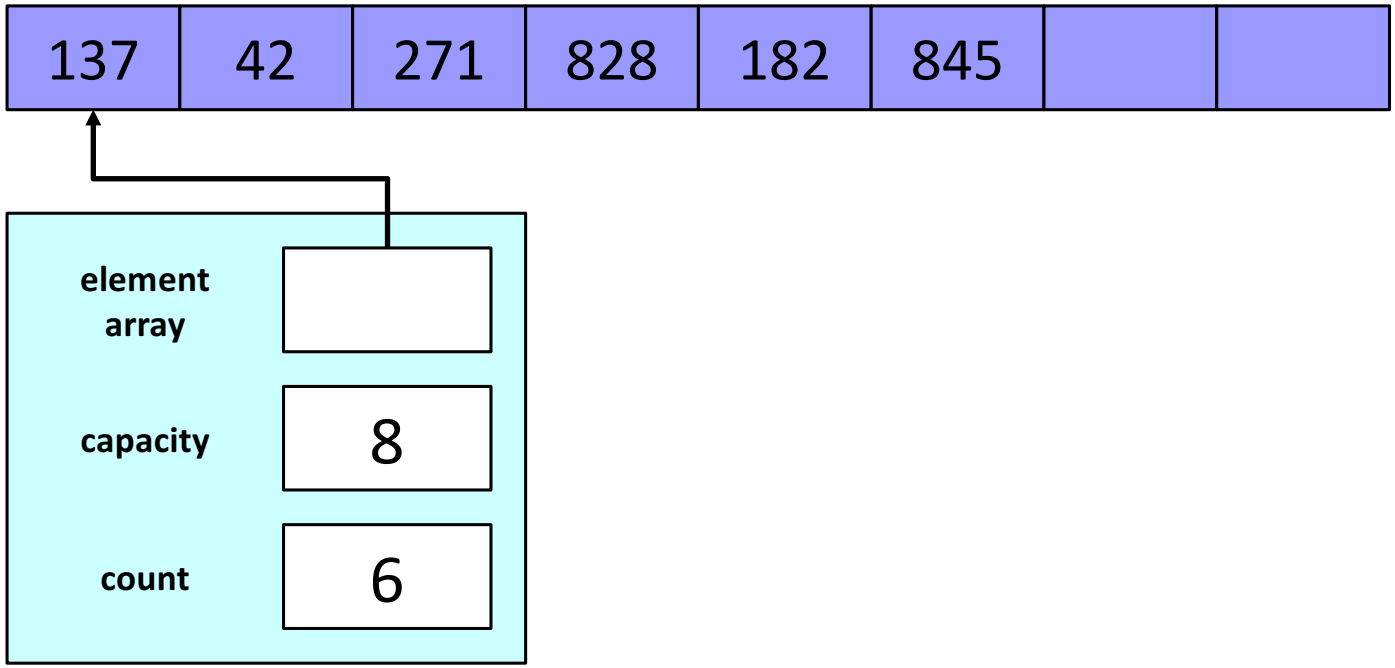
The Actual Vector



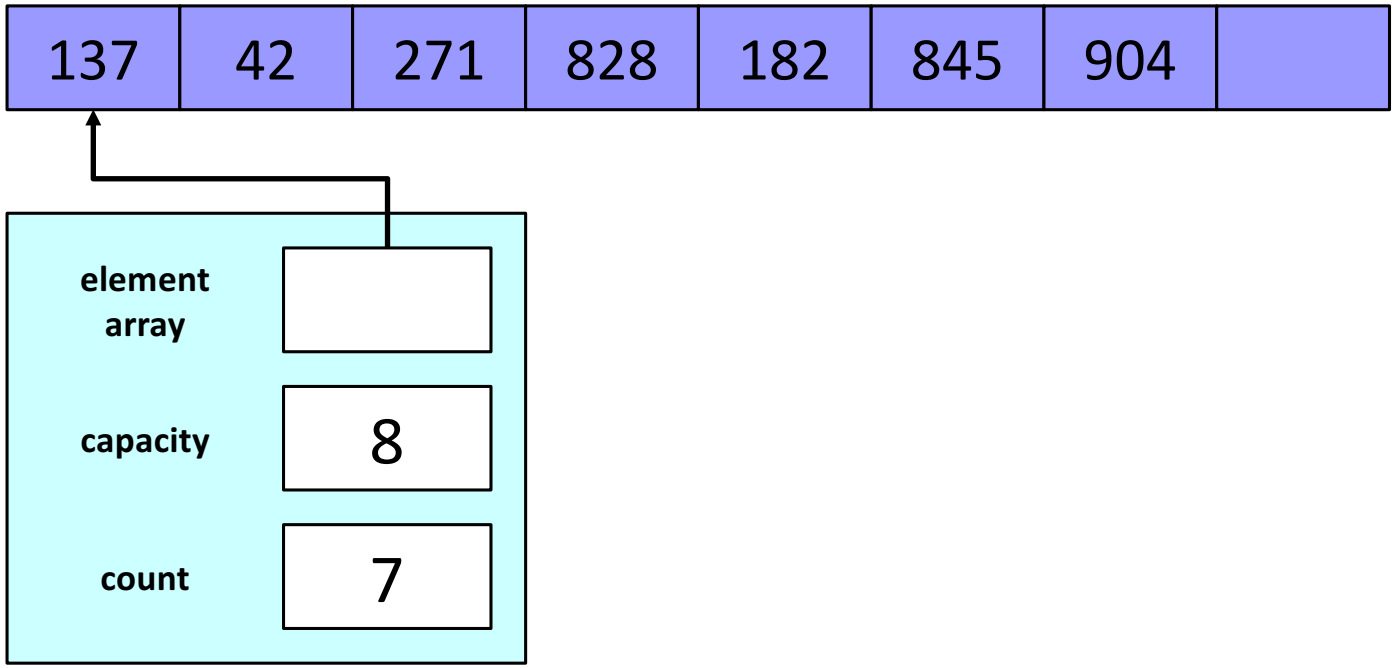
The Actual Vector



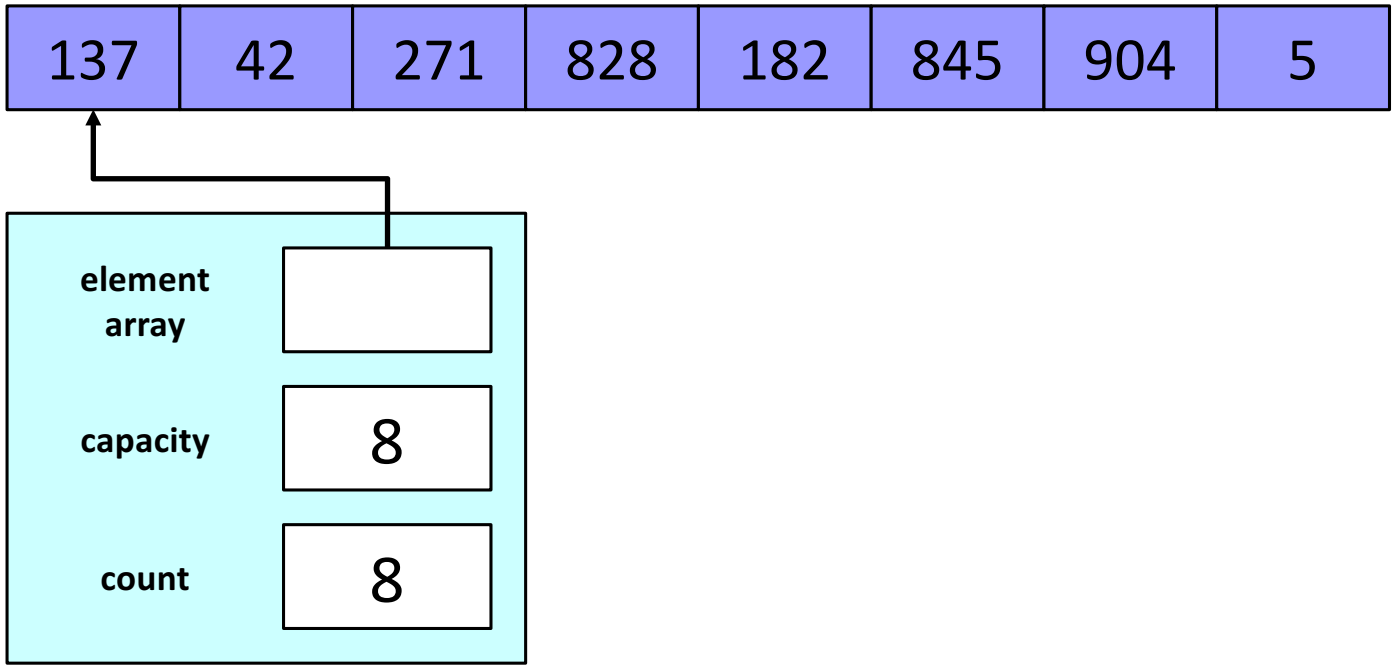
The Actual Vector



The Actual Vector



The Actual Vector



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){
    if(count == capacity) {
        expand();
    }
    elements[count] = v;
    count++;
}

void VecInt::expand() {
    capacity *= 2;
    int * newData = new int[capacity];
    for(int i = 0; i < count; i++) {
        newData[i] = elements[i];
    }
    delete[] data;
    elements = newData;
}
```

Big O of Add?

```
void VecInt::add(int v){
    if(count == capacity) {
        expand();
    }
    elements[count] = v;
    count++;
}

void VecInt::expand() {
    capacity *= 2;
    int * newData = new int[capacity];
    for(int i = 0; i < count; i++) {
        newData[i] = elements[i];
    }
    delete[] data;
    elements = newData;
}
```

Worst Case

Big O of Add?

```
void VecInt::add(int v){
    if(count == capacity) {
        expand();
    }
    elements[count] = v;
    count++;
}

void VecInt::expand() {
    capacity *= 2;
    int * newData = new int[capacity];
    for(int i = 0; i < count; i++) {
        newData[i] = elements[i];
    }
    delete[] data;
    elements = newData;
}
```

Worst Case

Big O of Add?

```
void VecInt::add(int v){
    if(count == capacity) {
        expand();
    }
    elements[count] = v;
    count++;
}

void VecInt::expand() {
    capacity *= 2;
    int * newData = new int[capacity];
    for(int i = 0; i < count; i++) {
        newData[i] = elements[i];
    }
    delete[] data;
    elements = newData;
}
```

Worst Case

Big O of Add?

```
void VecInt::add(int v){
    if(count == capacity) {
        expand();
    }
    elements[count] = v;
    count++;
}
```

```
void VecInt::expand() {
    capacity *= 2;
    int * newData = new int[capacity];
    for(int i = 0; i < count; i++) {
        newData[i] = elements[i];
    }
    delete[] data;
    elements = newData;
}
```

Worst Case

Big O of Add?

```
void VecInt::add(int v){
    if(count == capacity) {
        expand();
    }
    elements[count] = v;
    count++;
}

void VecInt::expand() {
    capacity *= 2;
    int * newData = new int[capacity];
    for(int i = 0; i < count; i++) {
        newData[i] = elements[i];
    }
    delete[] data;
    elements = newData;
}
```

Worst Case

Big O of Add?

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

Remove?

Big O of Remove?

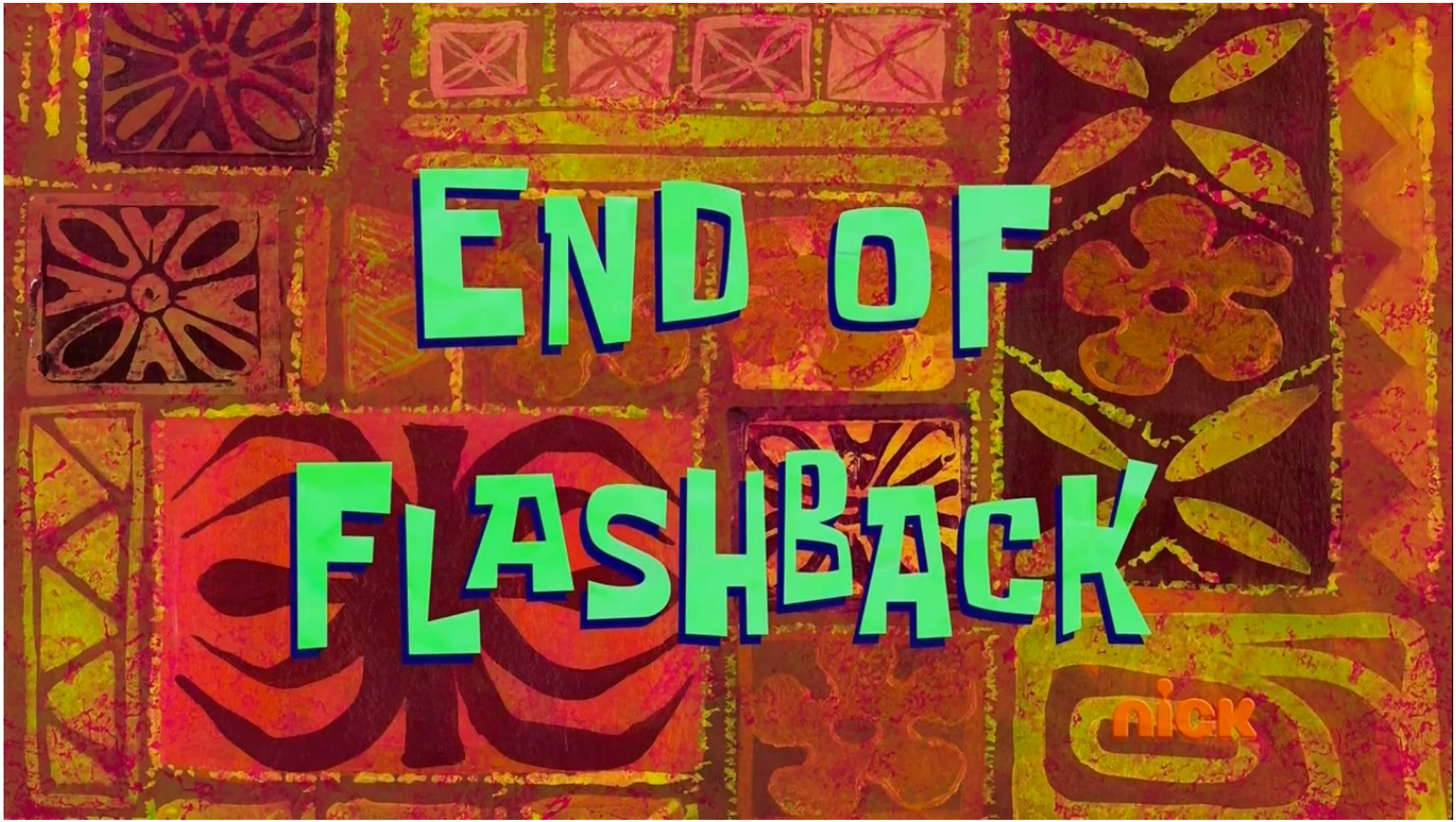
```
void VecInt::remove(int index){
    for(int i = index; i < count - 1; i++) {
        elements[i] = elements[i + 1];
    }
    count--;
}
```

Big O of Remove?

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

Vector Big O Summary

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

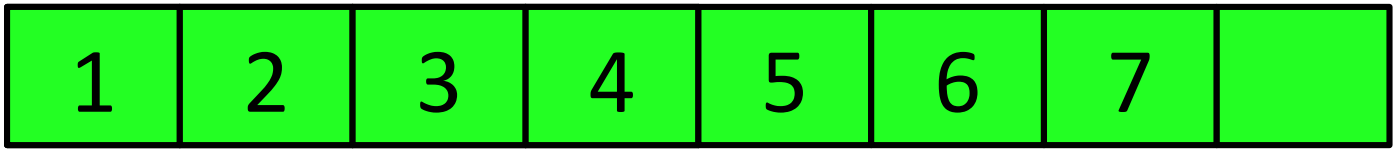

END OF
FLASHBACK
nick

Your job: Architect of Queue

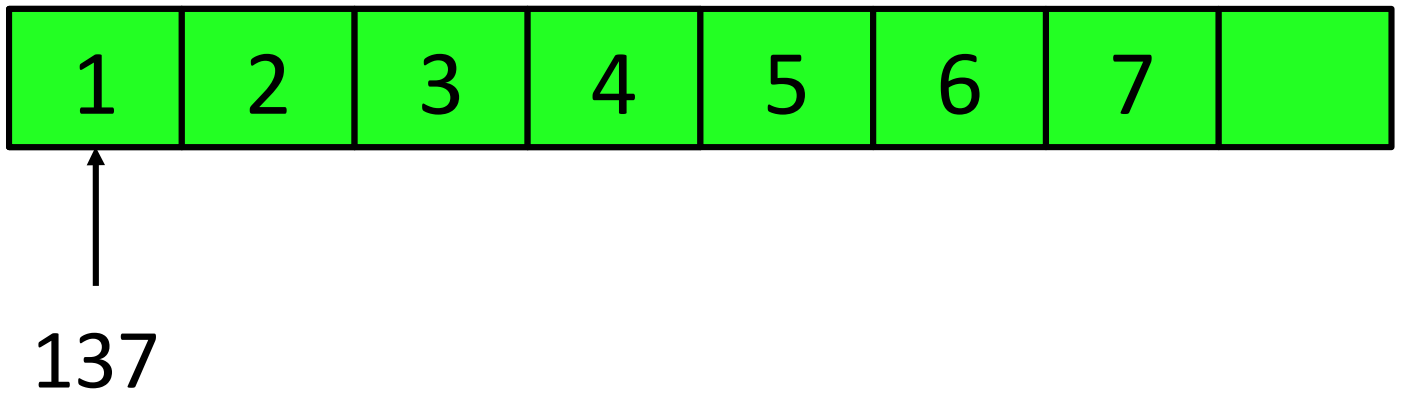
Easiest solution…

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

    void enqueue(int value); // append a value
    int dequeue();            // return the first-in value

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

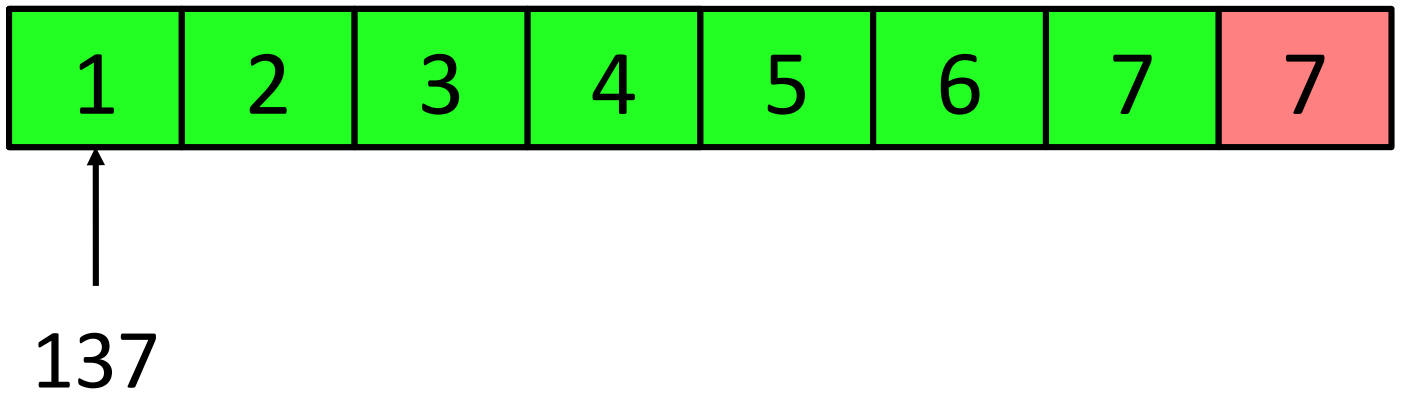
Excuse Me, Coming Through



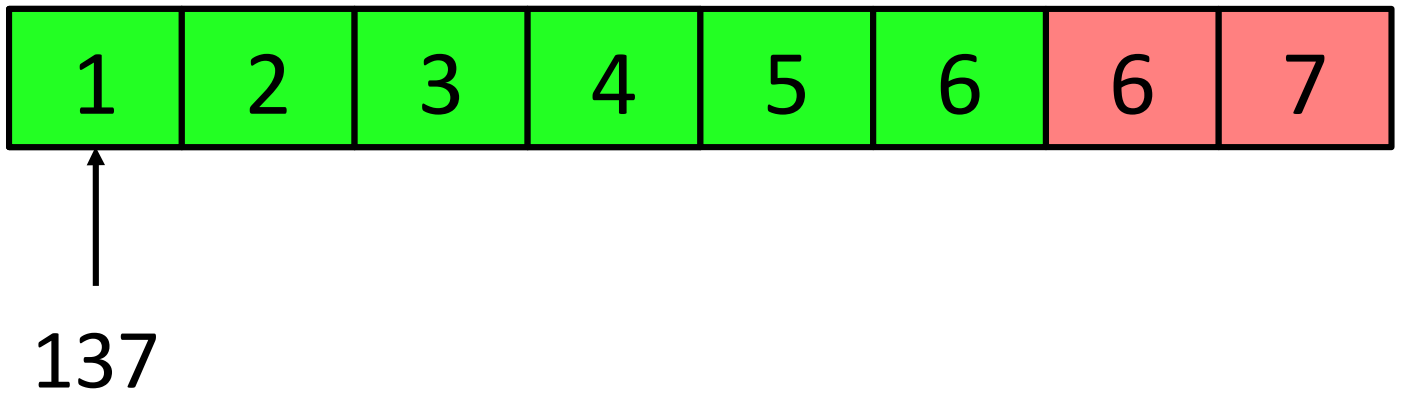
Excuse Me, Coming Through



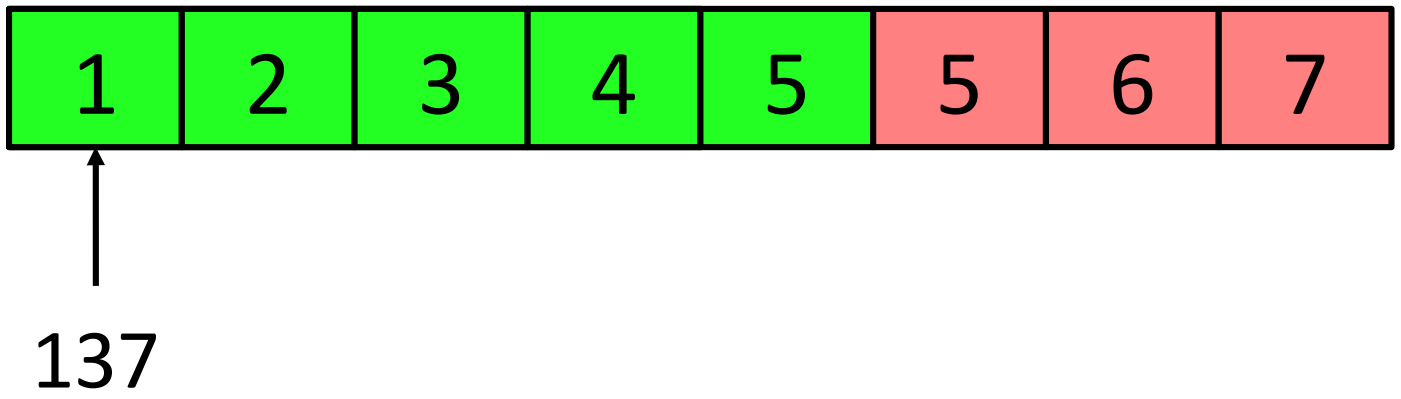
Excuse Me, Coming Through



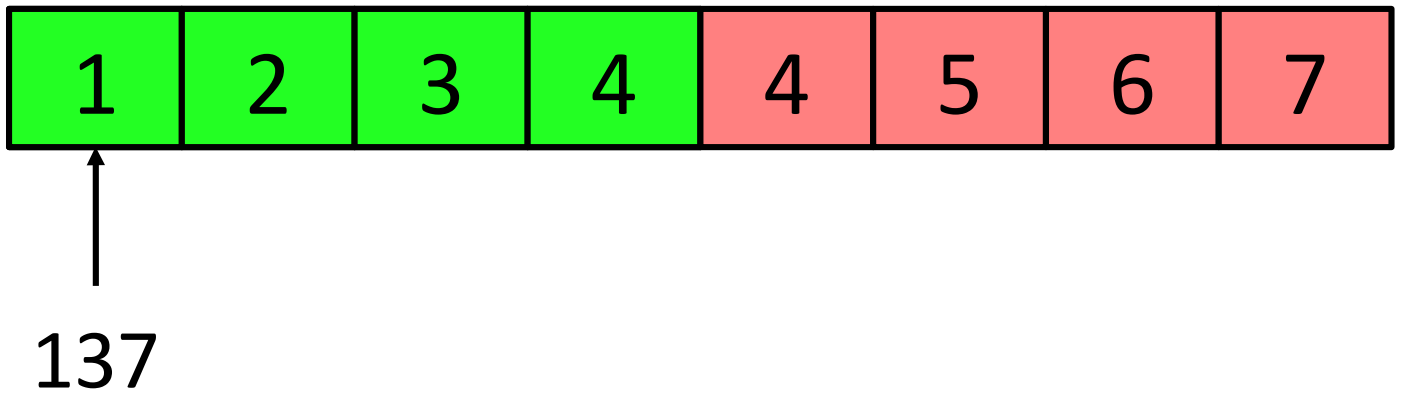
Excuse Me, Coming Through



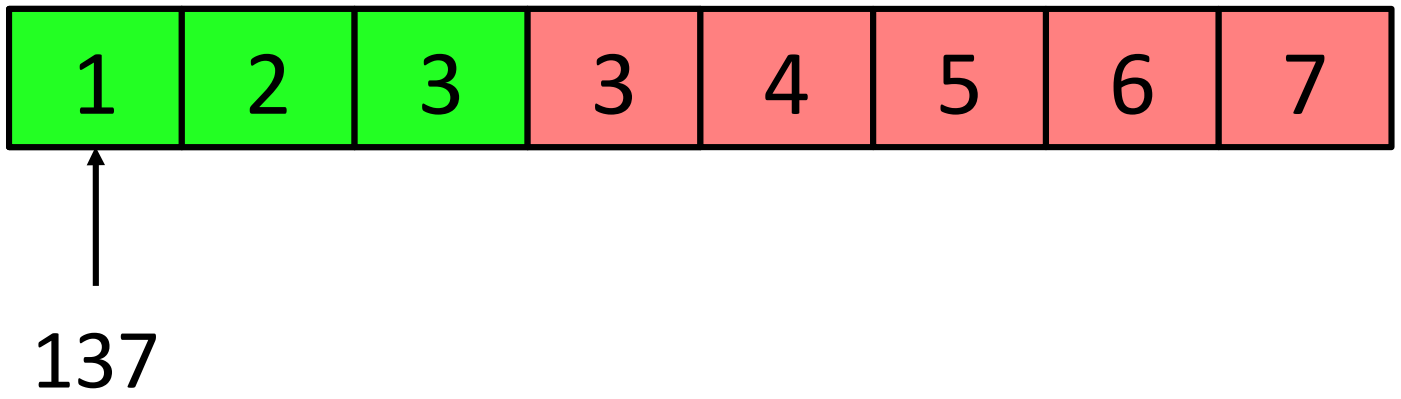
Excuse Me, Coming Through



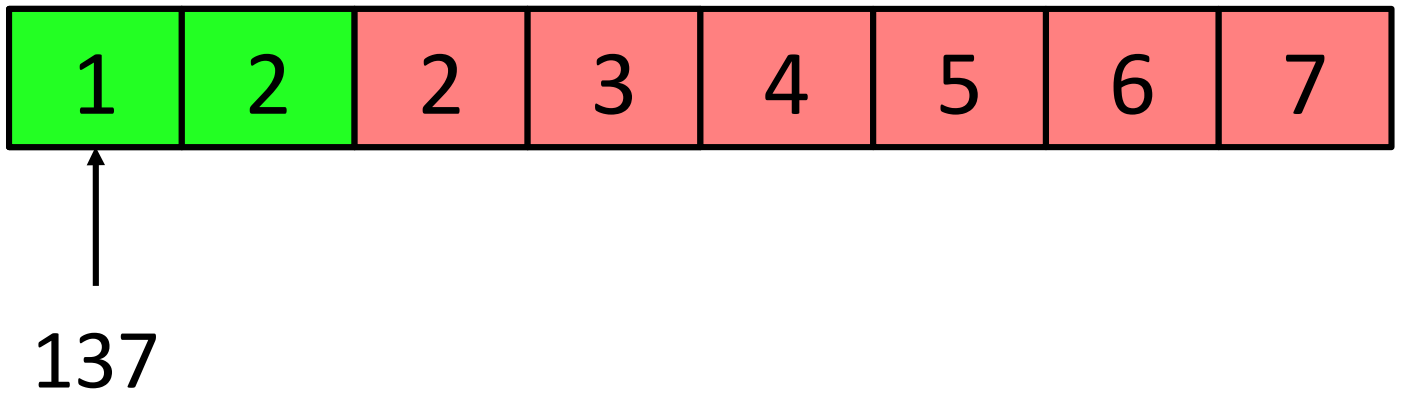
Excuse Me, Coming Through



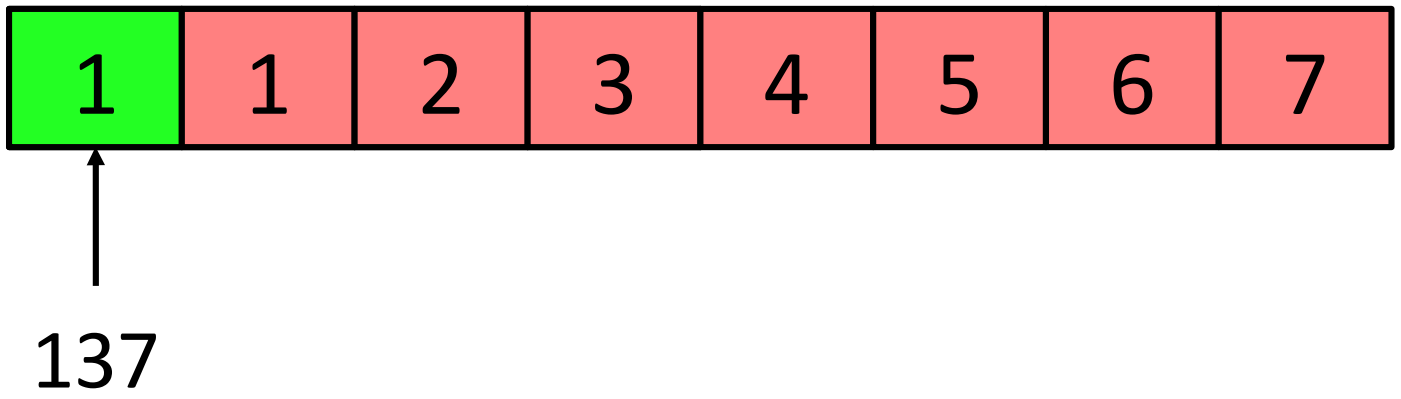
Excuse Me, Coming Through



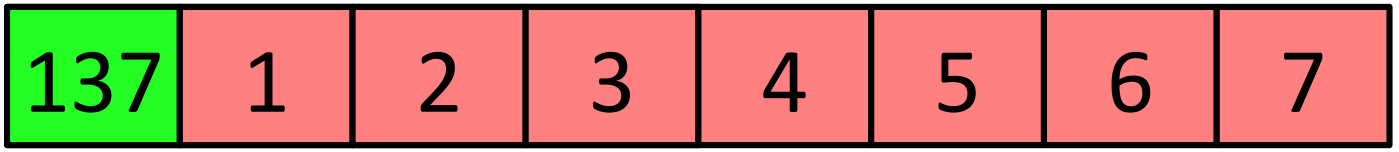
Excuse Me, Coming Through



Excuse Me, Coming Through



Excuse Me, Coming Through



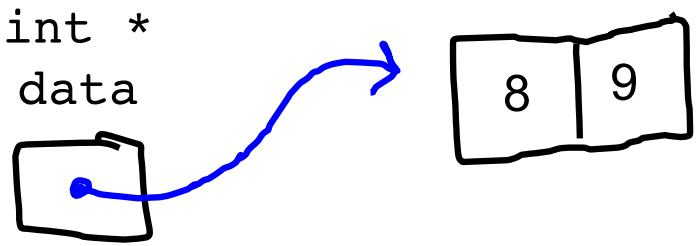
Queue as Vector: Big O

Enqueue	$\mathcal{O}(n)$
Dequeue	$\mathcal{O}(1)$

There’s always a better way

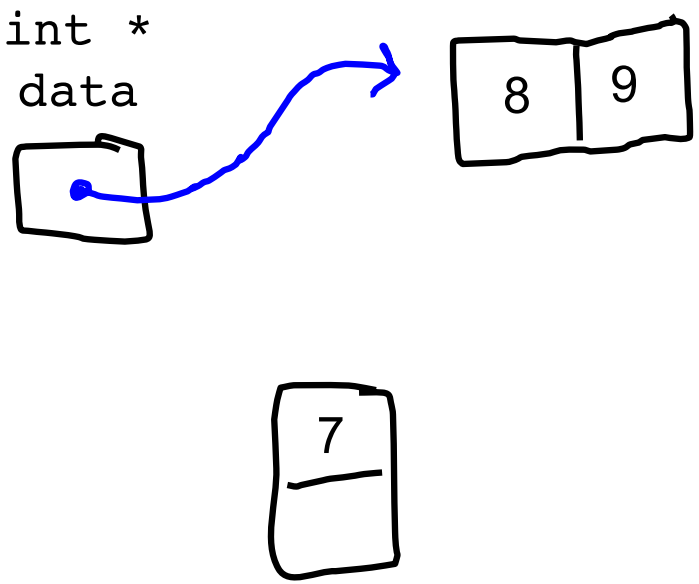
What About This?

```
enqueue(7);
```



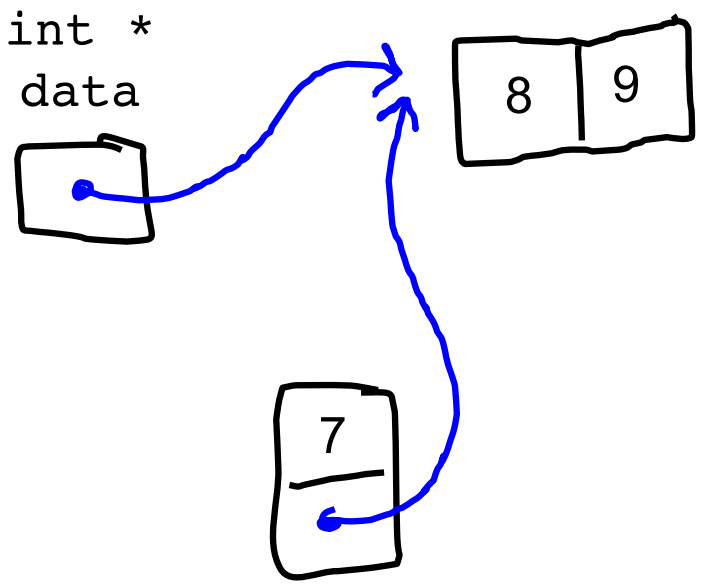
What About This?

```
enqueue(7);
```



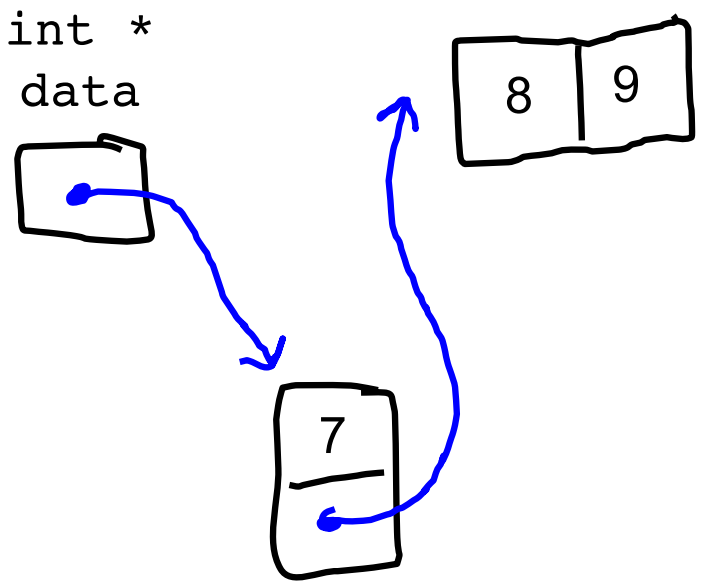
What About This?

```
enqueue(7);
```



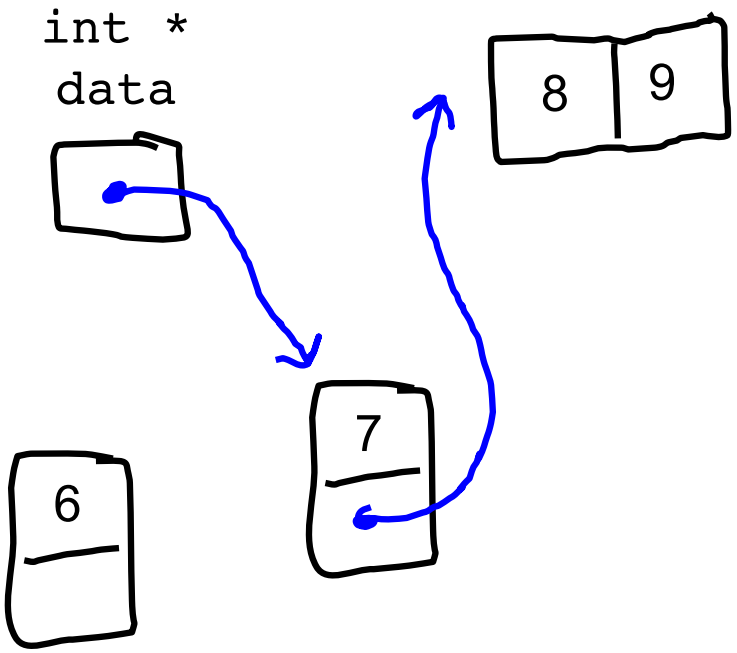
What About This?

```
enqueue(7);
```



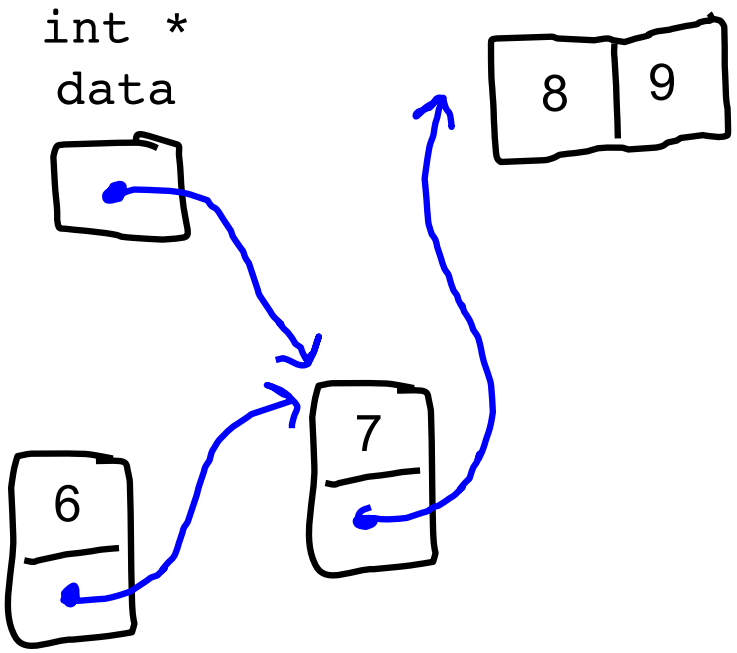
What About This?

```
enqueue(6);
```



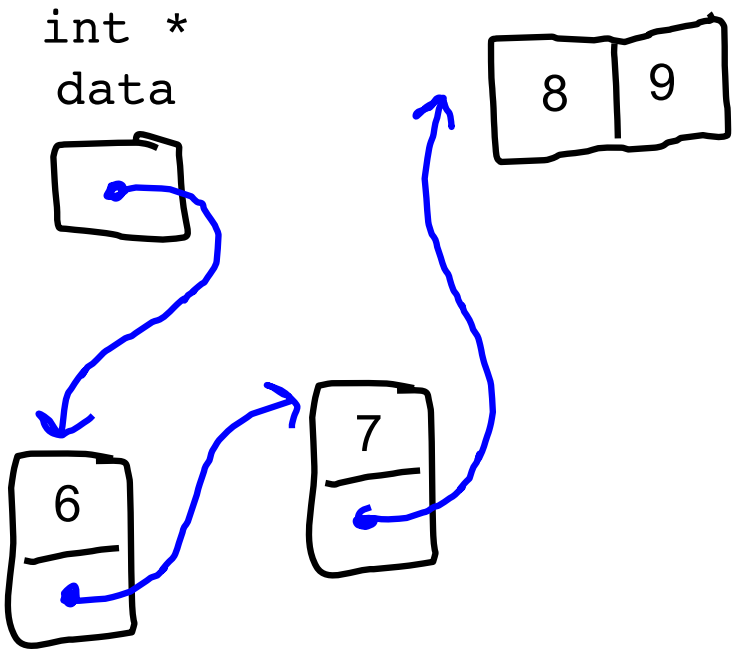
What About This?

```
enqueue(6);
```

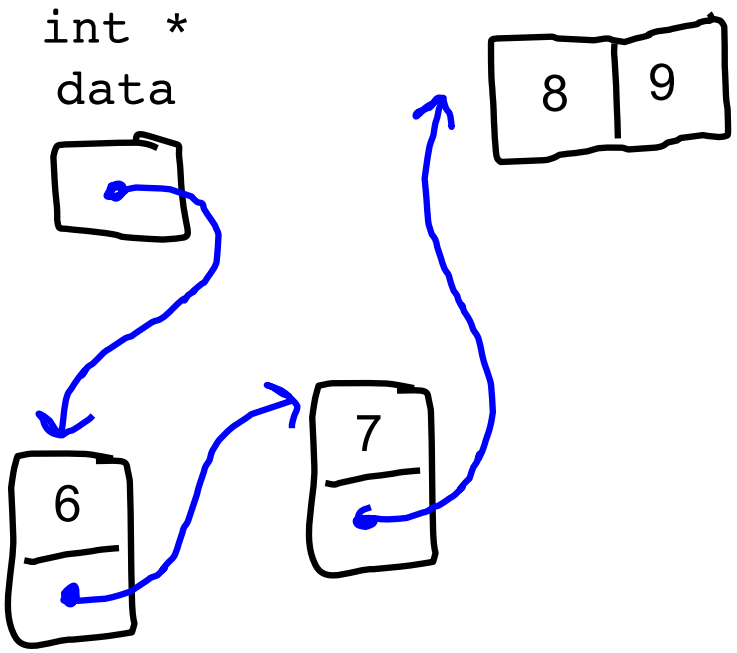


What About This?

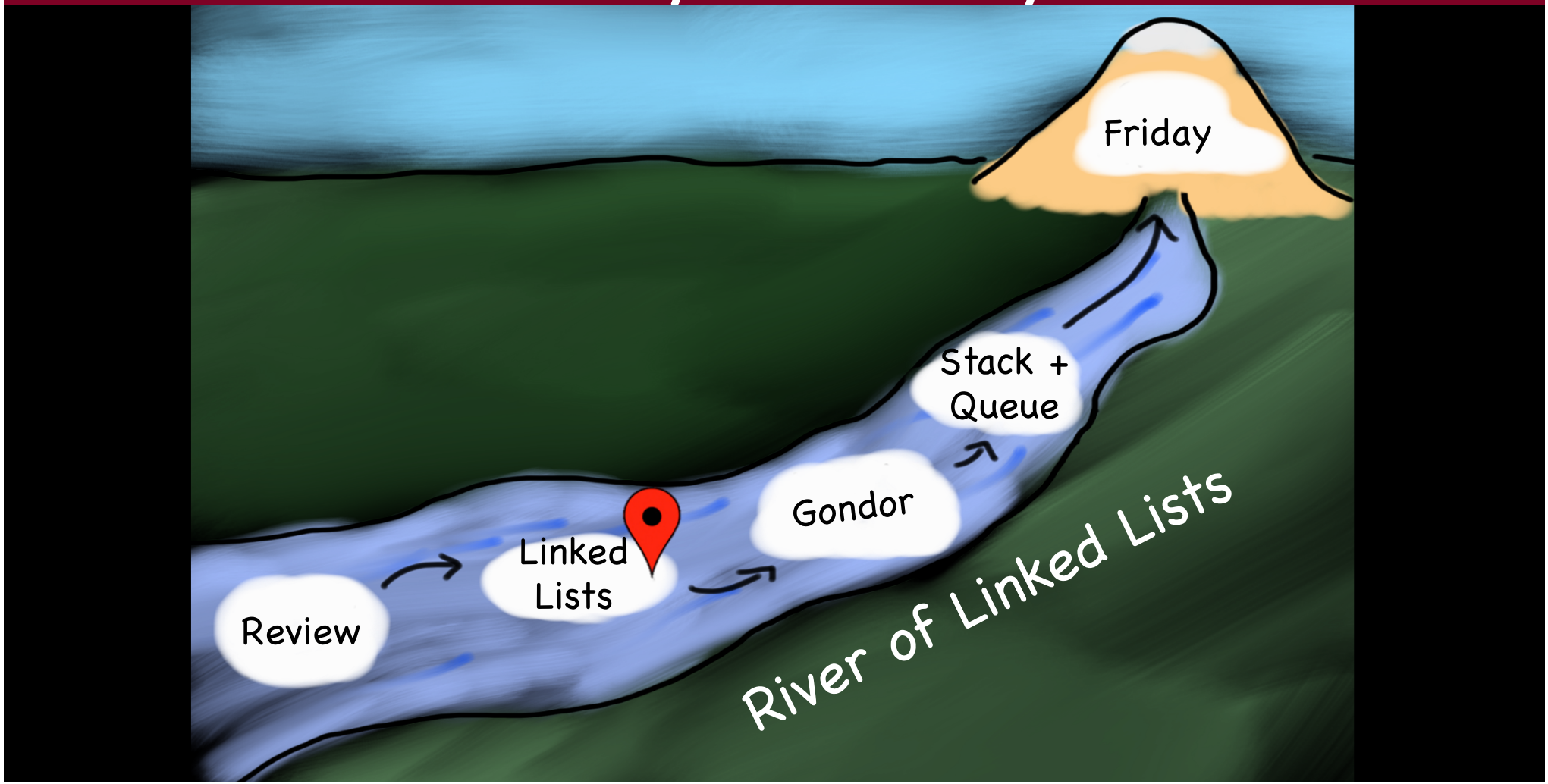
```
enqueue(6);
```



What About This?

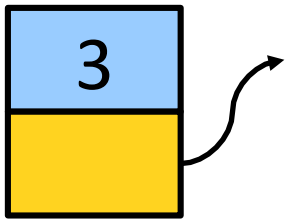


Today's Journey



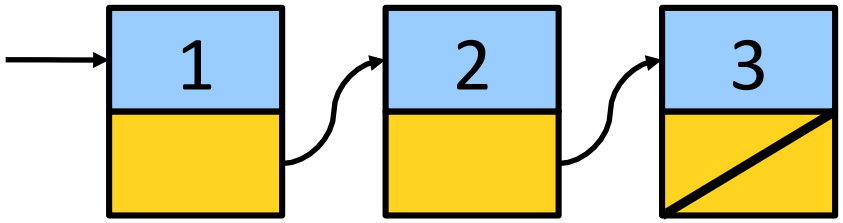
Linked List

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



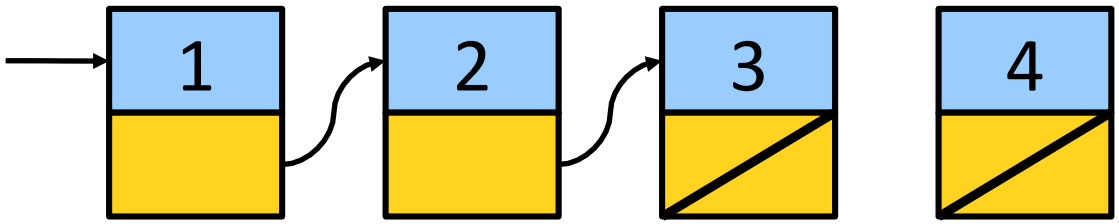
Linked Lists

- 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.



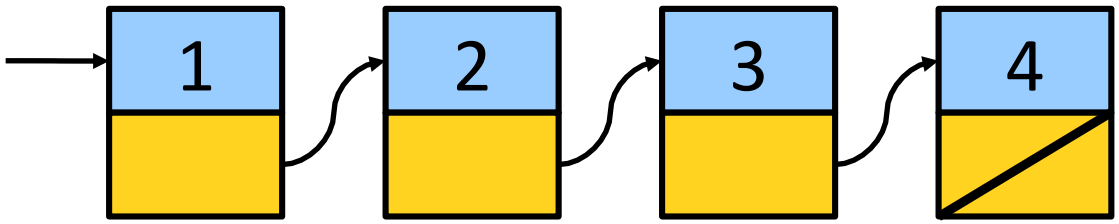
Linked Lists

- 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.



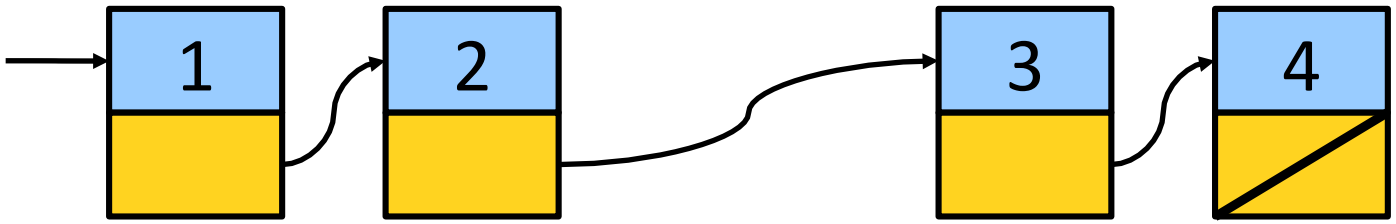
Linked Lists

- 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.



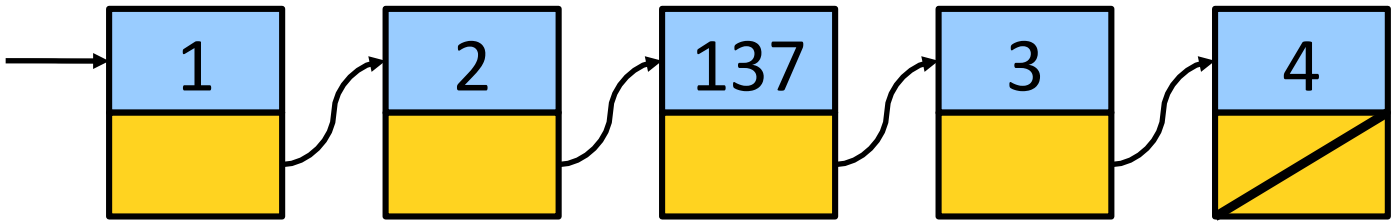
Linked Lists

- 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.



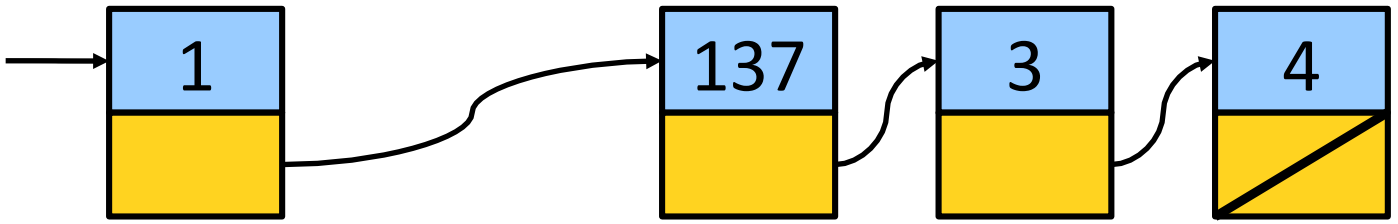
Linked Lists

- 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.



Why 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;
- Has some tradeoffs; we'll see this later.

Linked List Structure

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

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

Linked List of Strings

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

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

Linked List of Strings

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

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

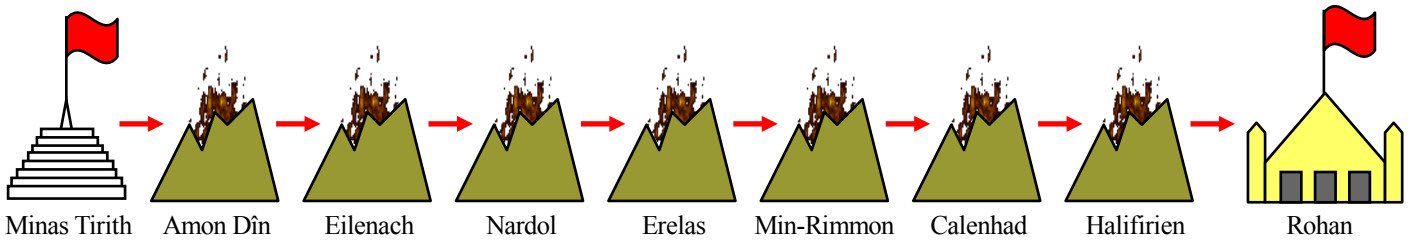
- **The structure is defined recursively!**

First Rule of Linked List Club

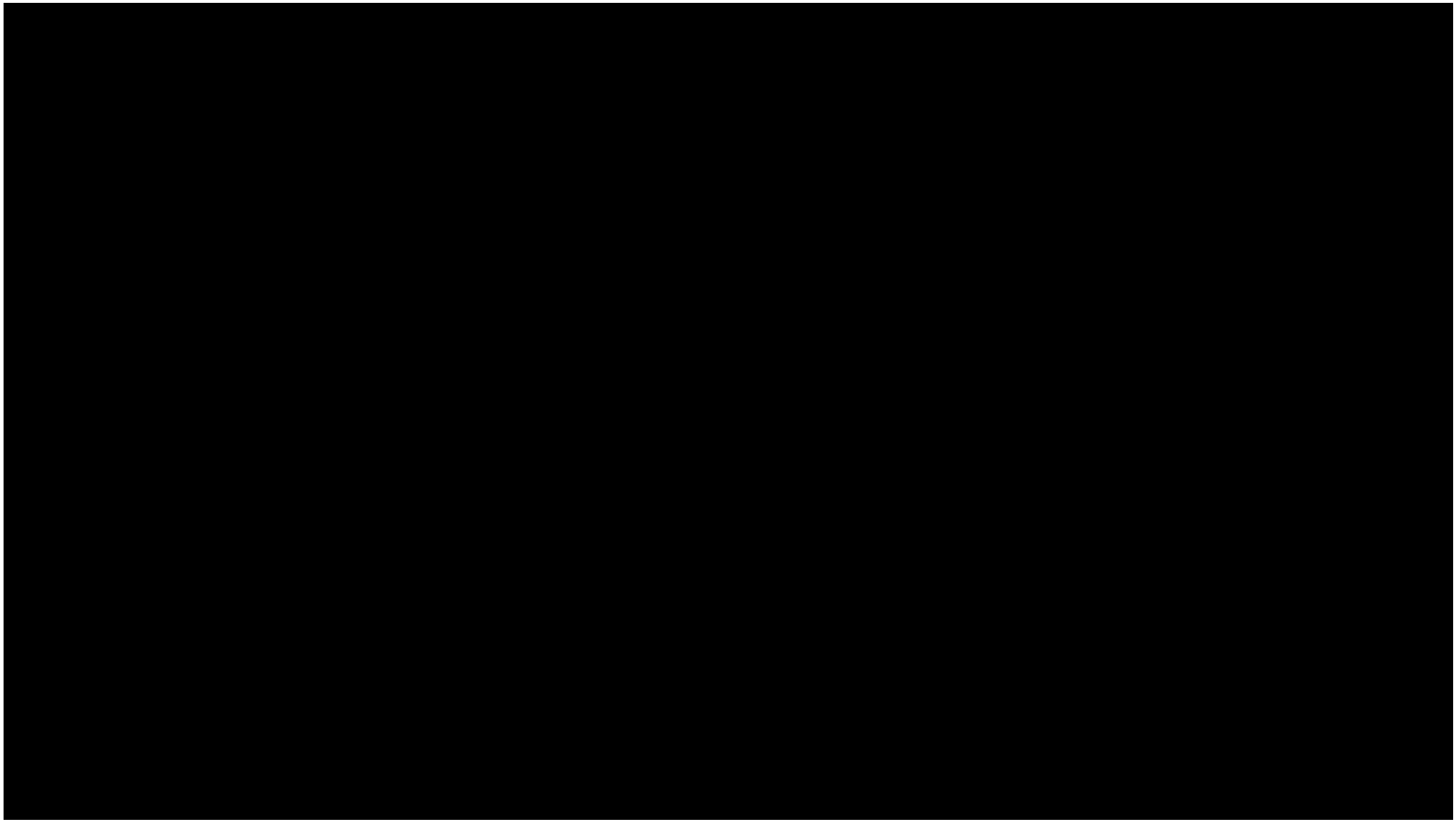
Draw a picture

Lord of the Linked Lists

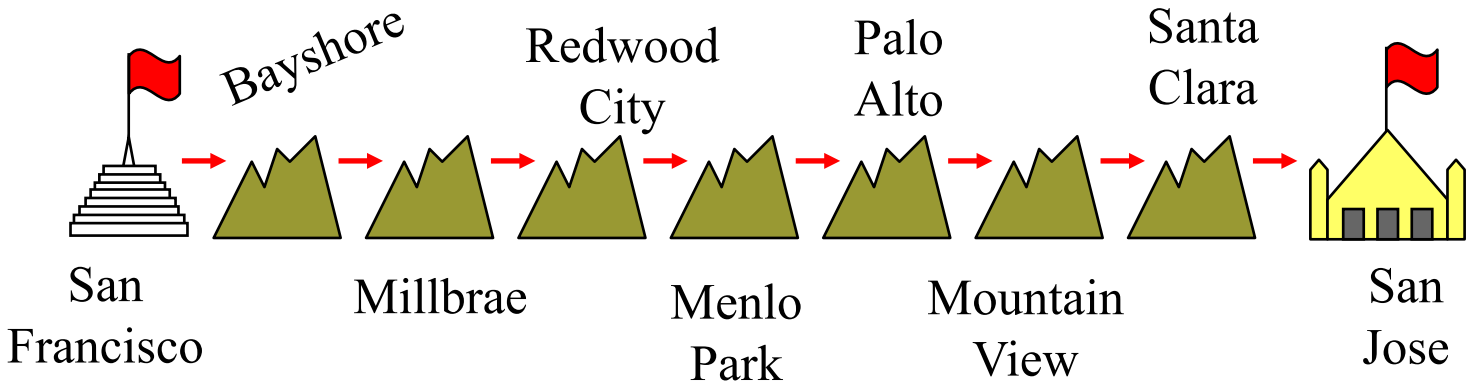
In a scene that was brilliantly captured in Peter Jackson's film adaptation of *The Return of the King,* Rohan is alerted to the danger to Gondor by a succession of signal fires moving from mountain top to mountain top. This scene is a perfect illustration of the idea of message passing in a linked list.




THE
LORD OF THE RINGS
THE RETURN OF THE KING



Step 1: Make this linked list



Step 2: Light the fires….

```
Lighting the fire of San Francisco!
```

Lord of the Linked Lists

```
struct Tower {
   string name;   /* The name of this tower    */
   Tower *link;   /* Pointer to the next tower */
};
```

```
// add the first tower
Tower * head = new Tower;
head->name = “San Jose”;
head->link = NULL;
```

```
// add the first tower
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;
```

```
// add the first tower
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;
```

```
// add the first tower
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;
```


Qt

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

Linked List Trace

```
// main
Tower * head = new Tower;
head->name = "San Jose";
head->link = NULL;

head = createTower("Santa Clara", head);
head = createTower("Mountain View", head);
head = createTower("Palo Alto", head);
head = createTower("Menlo Park", head);
head = createTower("Redwood City", head);
head = createTower("Millbrae", head);
head = createTower("Bayshore", head);
head = createTower("San Francisco", head);
```

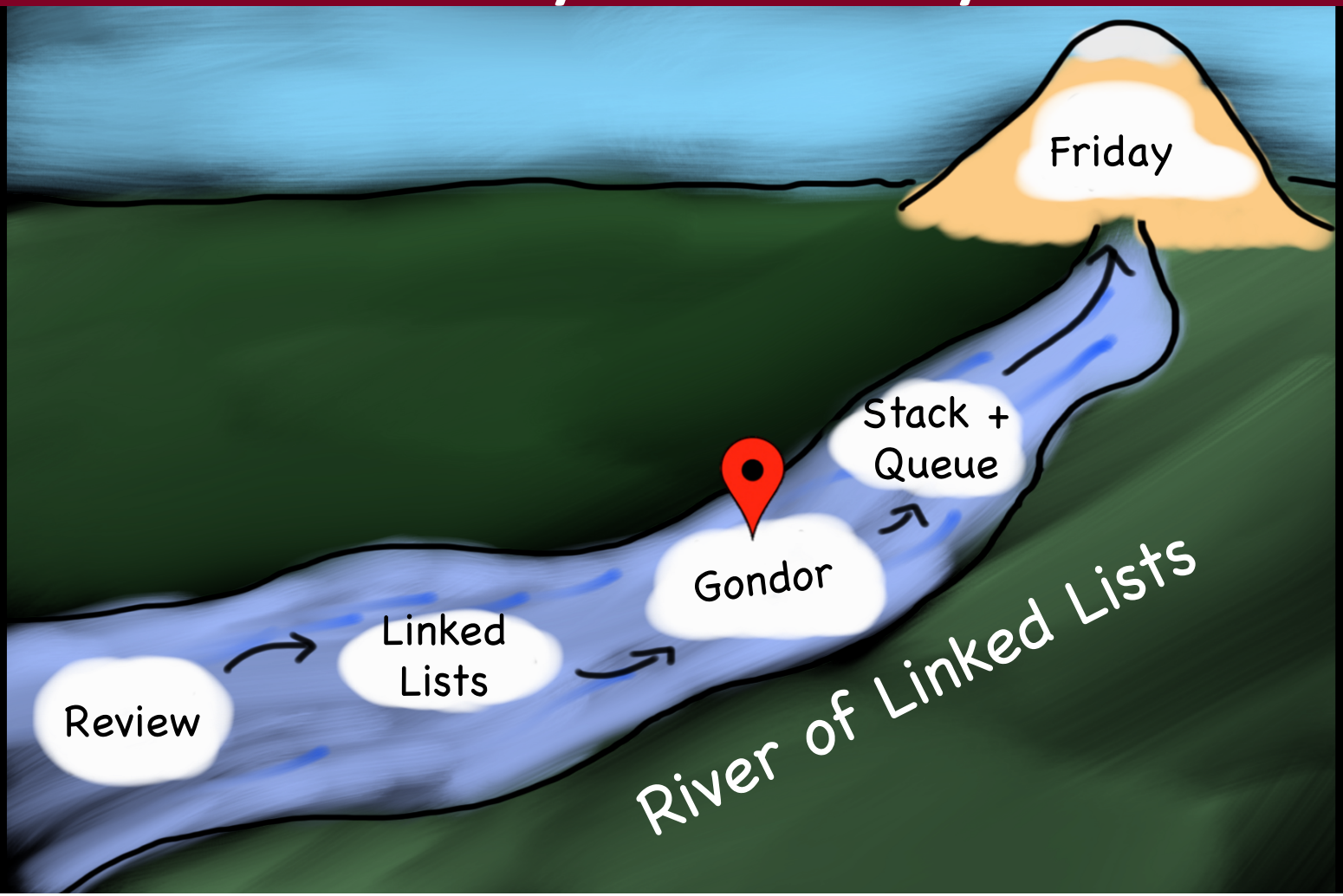
```
struct Tower{
    string name;
    Tower * link;
};
```

```
Tower *createTower(string name, Tower *link) {
    Tower *tp = new Tower;
    tp->name = name;
    tp->link = link;
    return tp;
}
```

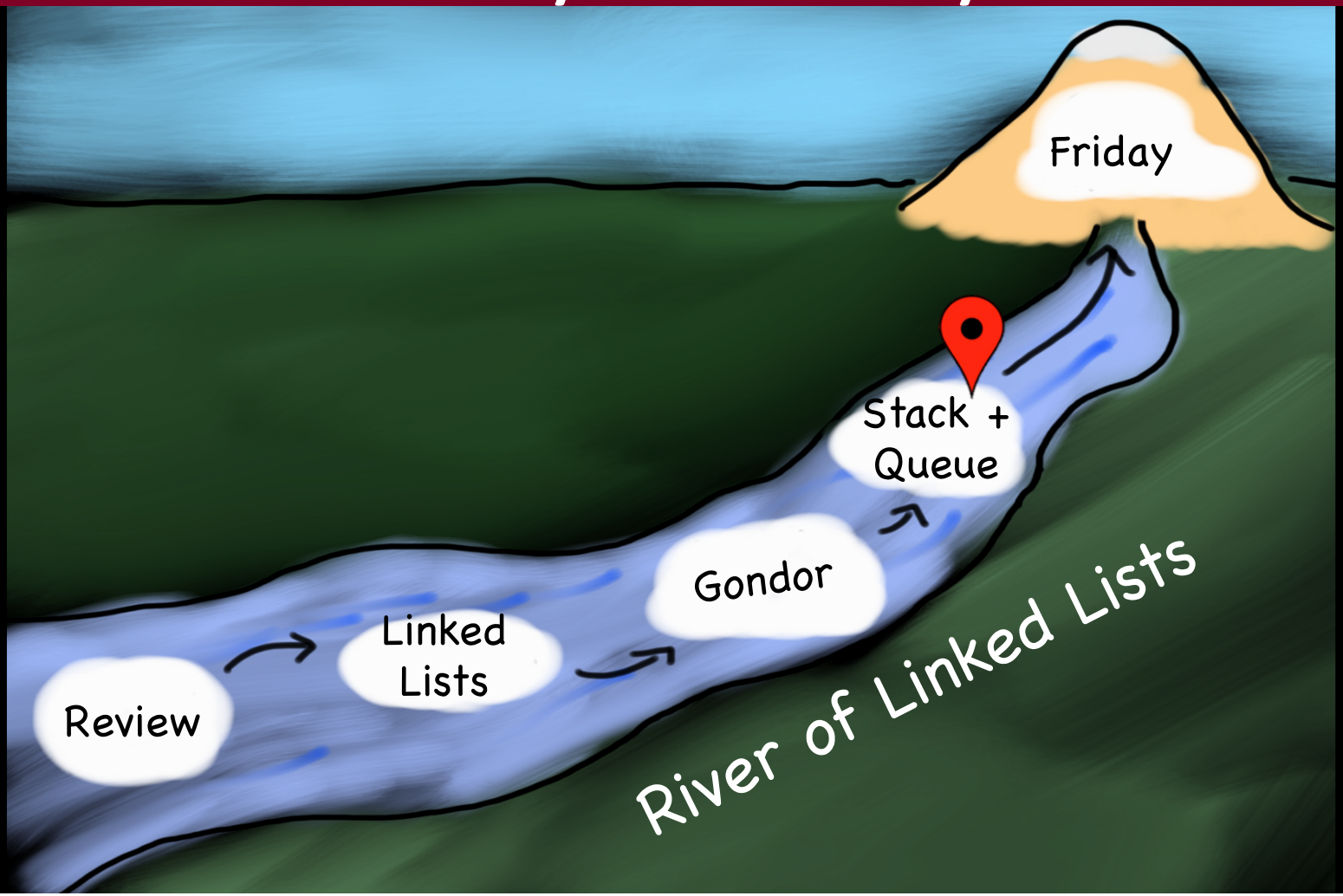
```
void signal(Tower *start) {
    if (start != NULL) {
        cout << "Lighting " << start->name << endl;
        signal(start->link);
    }
}
```

```
signal(head);
```

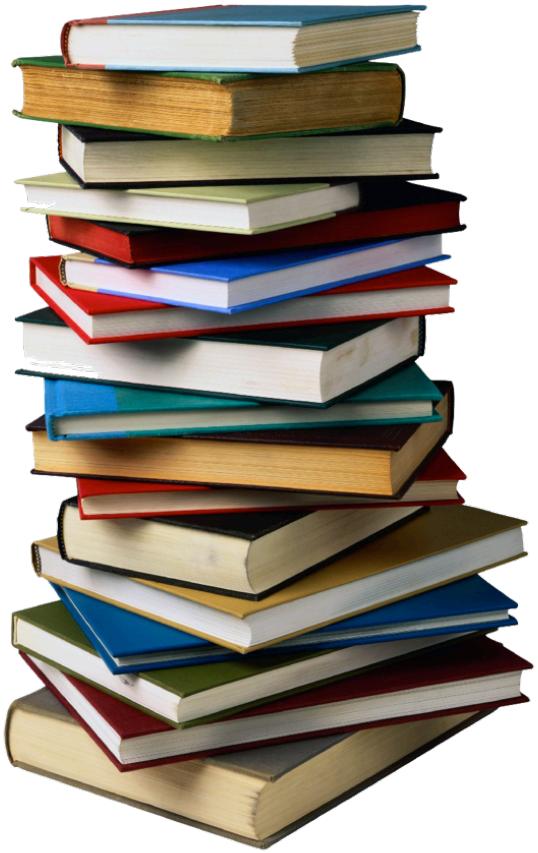
Today's Journey



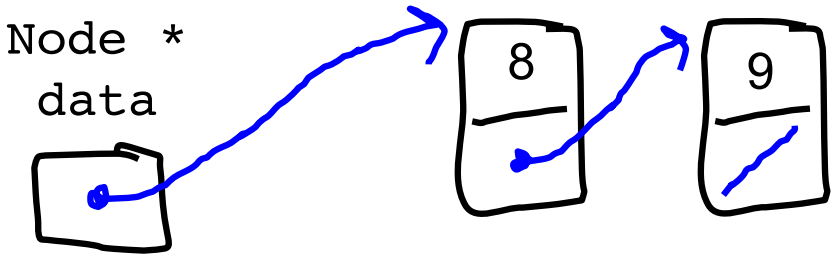
Today's Journey



How is the Stack Implemented?



```
struct Node{
   int value;     /* The value of this elem    */
   Node *link;    /* Pointer to the next node */
};
```

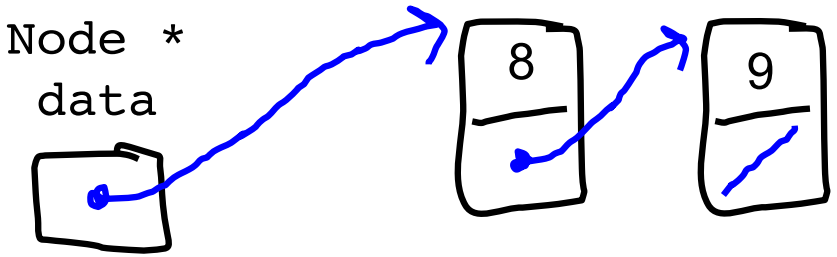

Node *
data
8
9

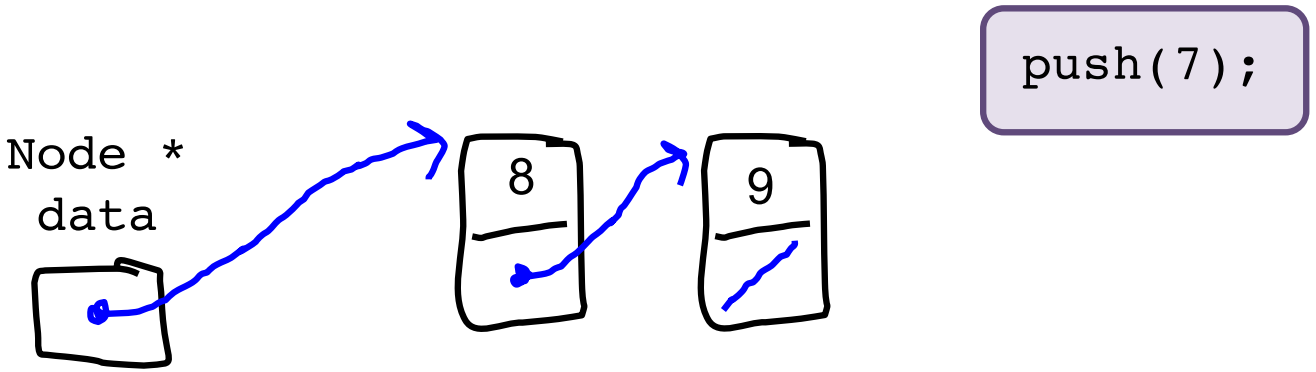
Stack

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

    void push(int value); // append a value
    int pop();                // return the first-in value

private:
    struct Node {
      int value;
      Node * link;
    };
    Node * data;      // member variables
};
```

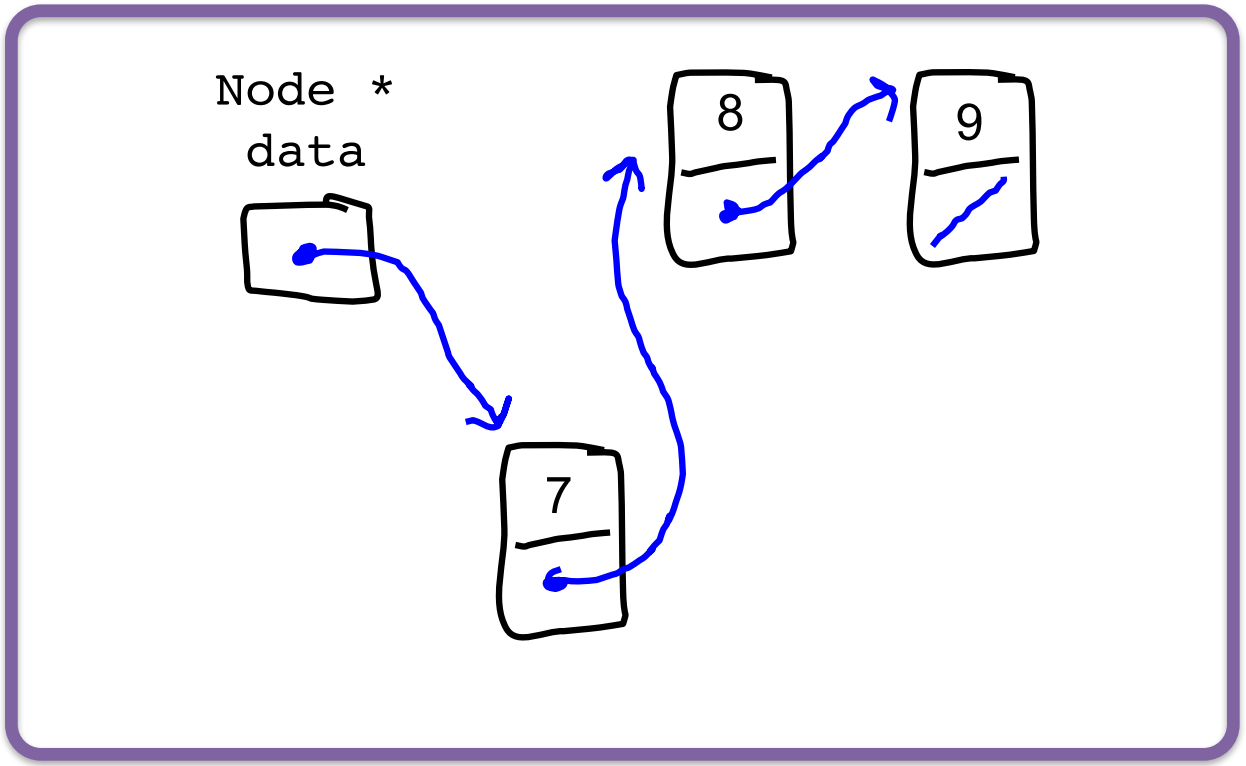
Stack Implementation

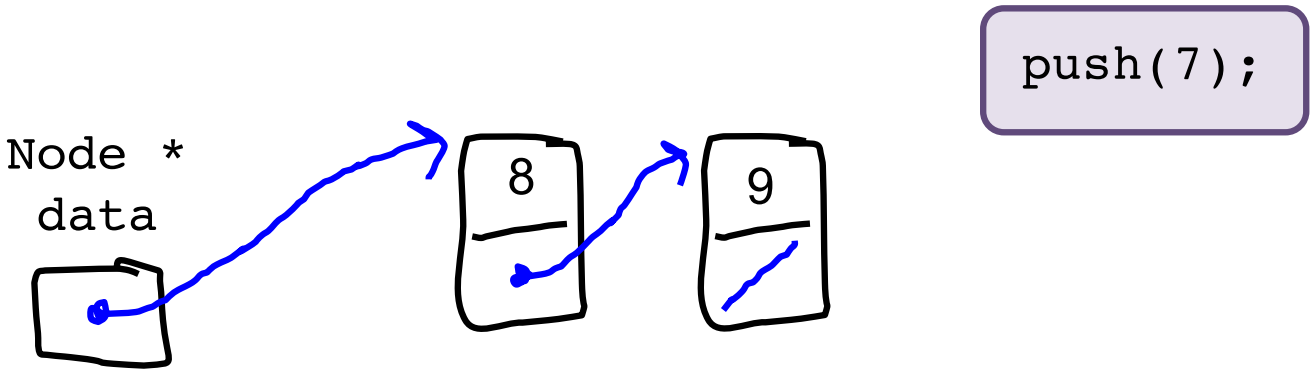
```
void StackInt::push(int v) {
    Node * temp = new Node;
    temp->value = v;
    temp->link = data;
    data = temp;
}

int StackInt::pop() {
    int toReturn = data->value;
    Node * temp = data;
    data = temp->link;
    delete data;
}
```


Node *
data
8
9


Node *
data
8
9
push(7);

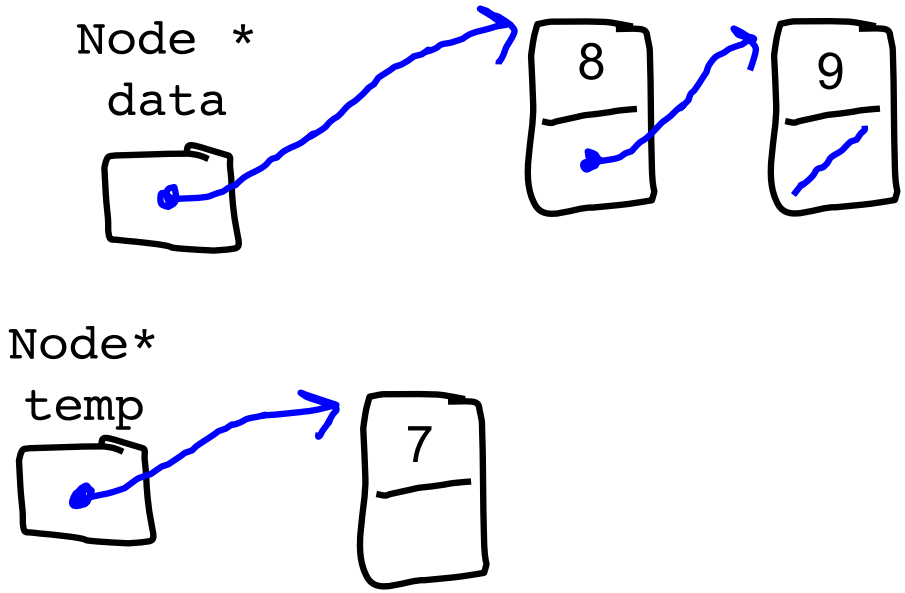
Goal of Push




Node *
data
8
9
push(7);

Stack is a Linked List

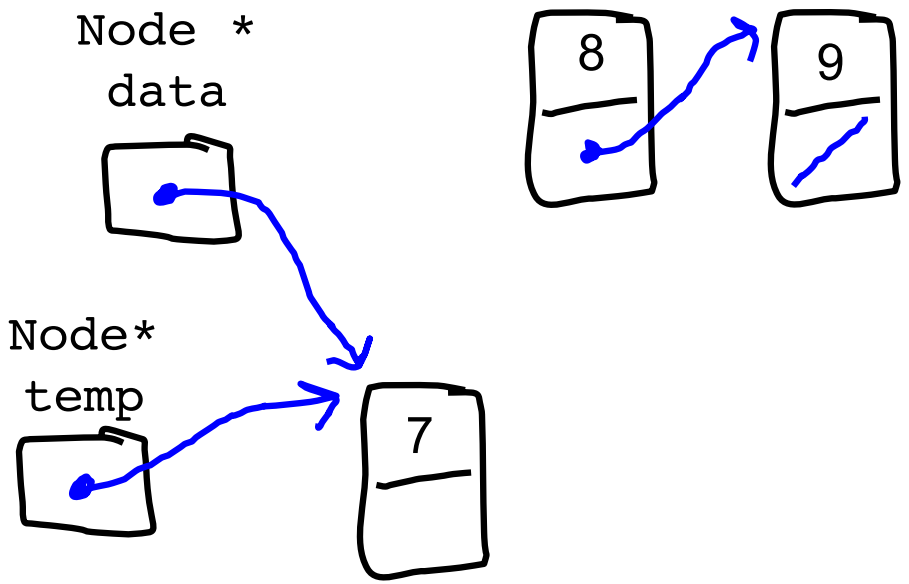
```
push(7);
```



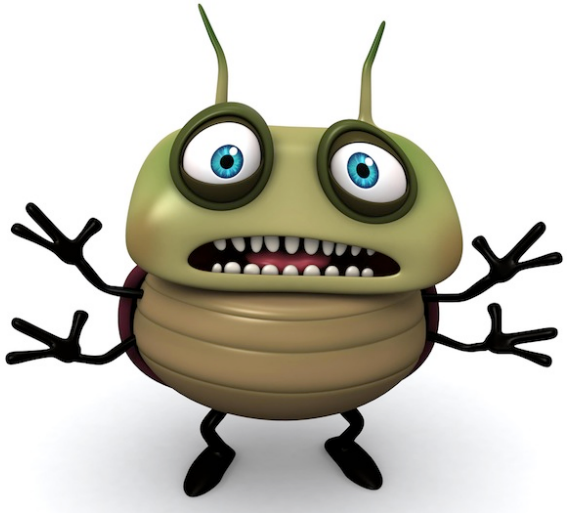

```
Node * temp =  new Node;
temp -> value = 7;
```

Stack is a Linked List

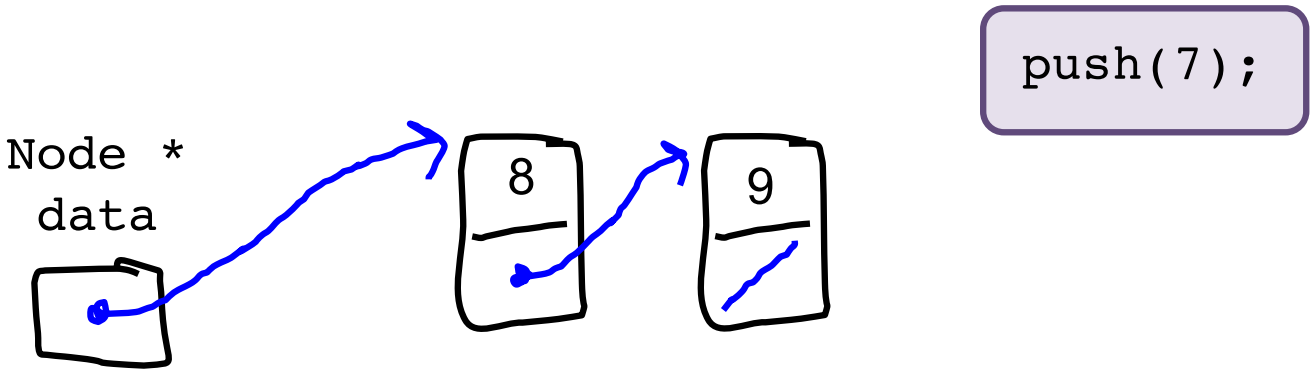
```
push(7);
```



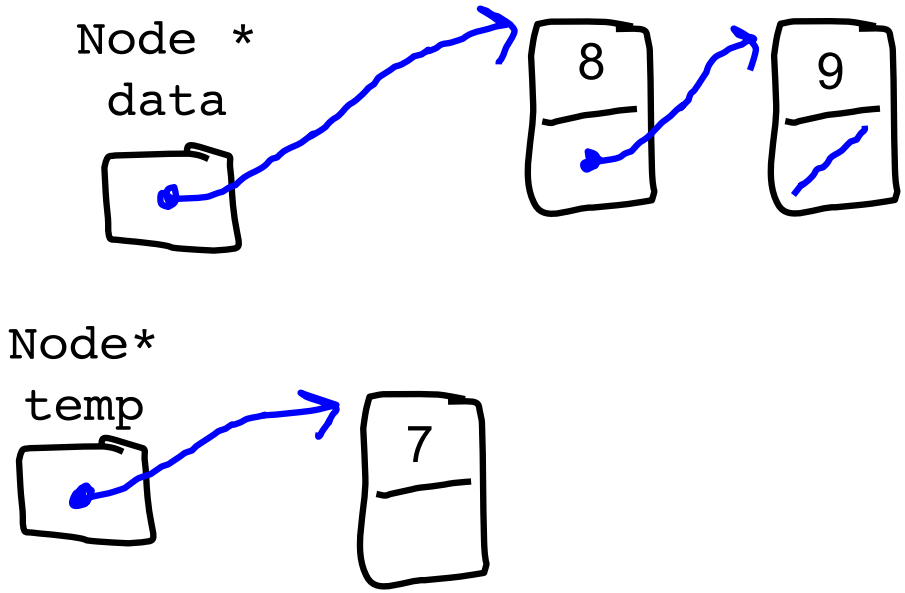

```
data = temp;
```



BRAAWWRRRR!

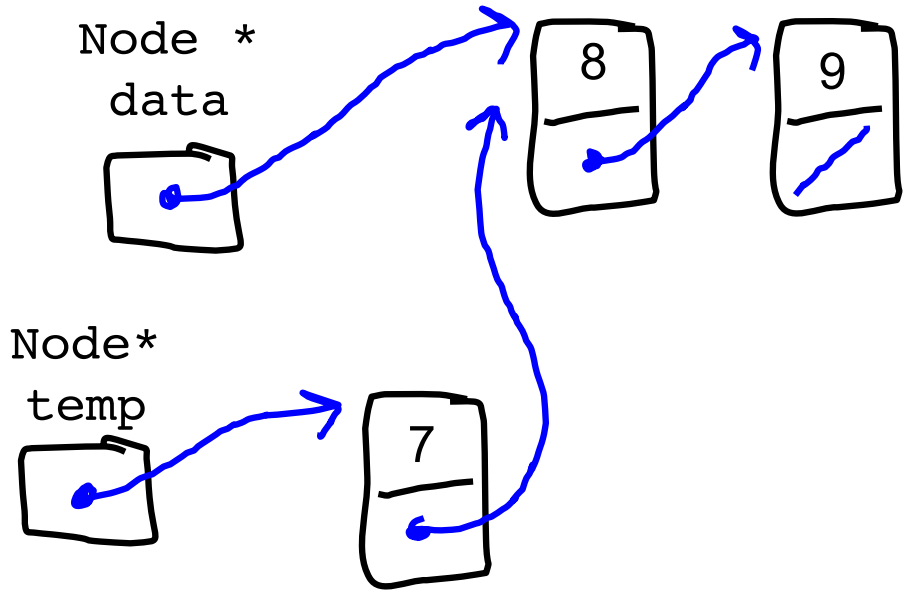

Node *
data
8
9
push(7);

```
push(7);
```



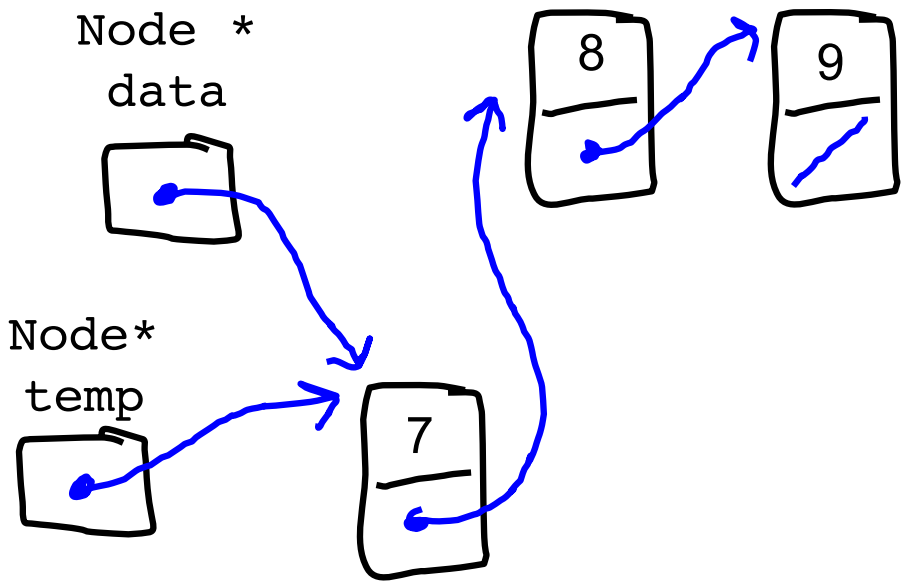

```
Node * temp =  new Node;
temp -> value = 7;
```

```
push(7);
```



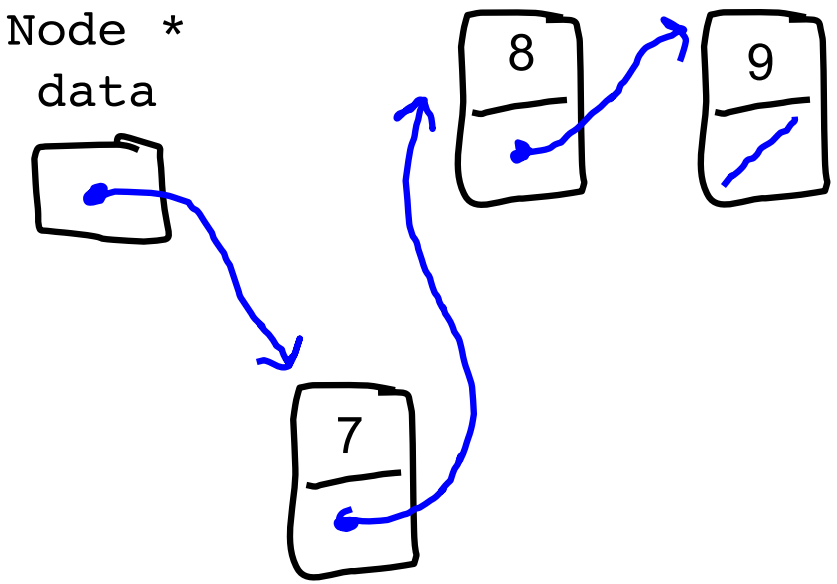

```
temp -> link = data;
```

```
push(7);
```

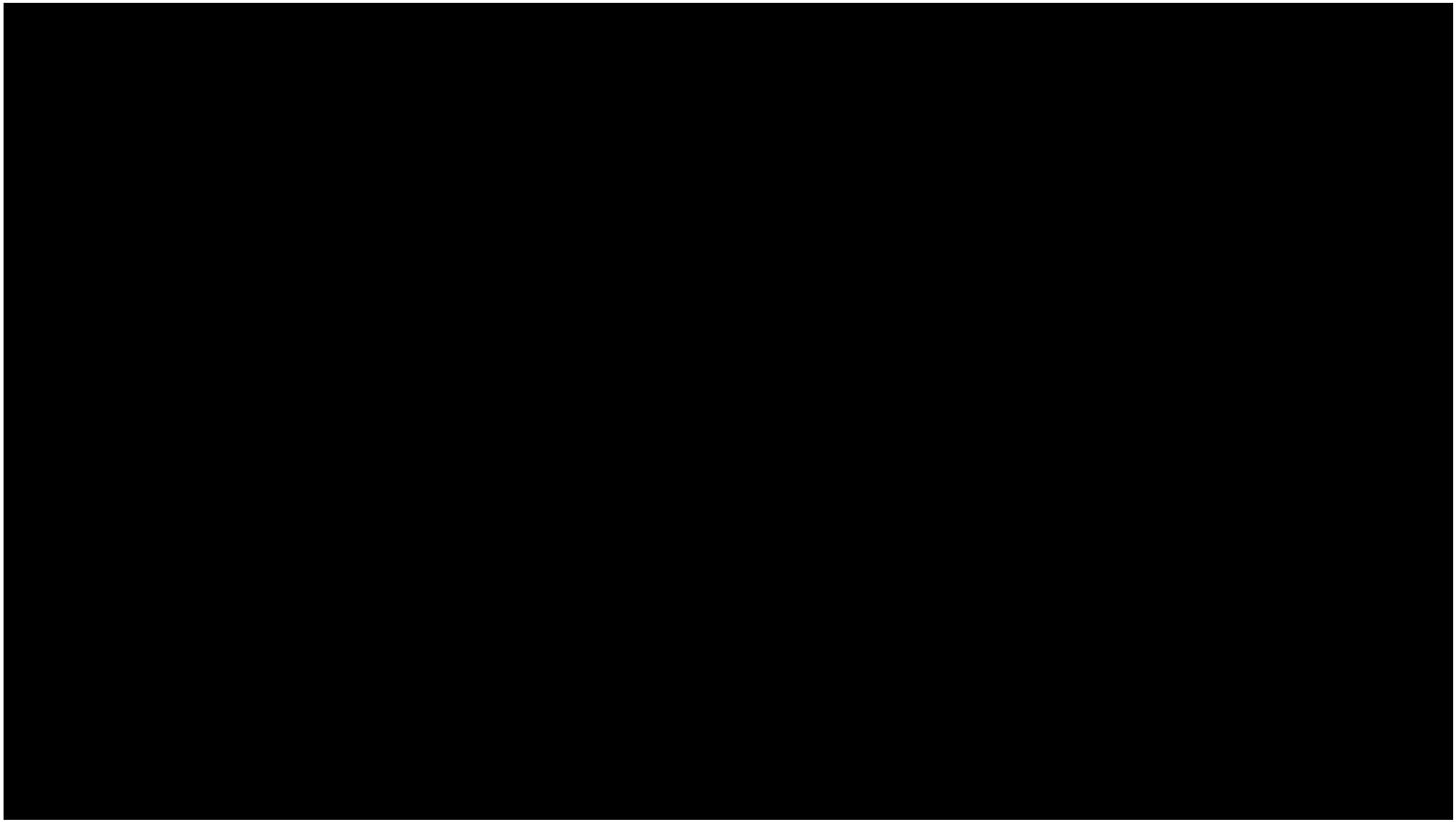



```
data = temp;
```

```
push(7);
```

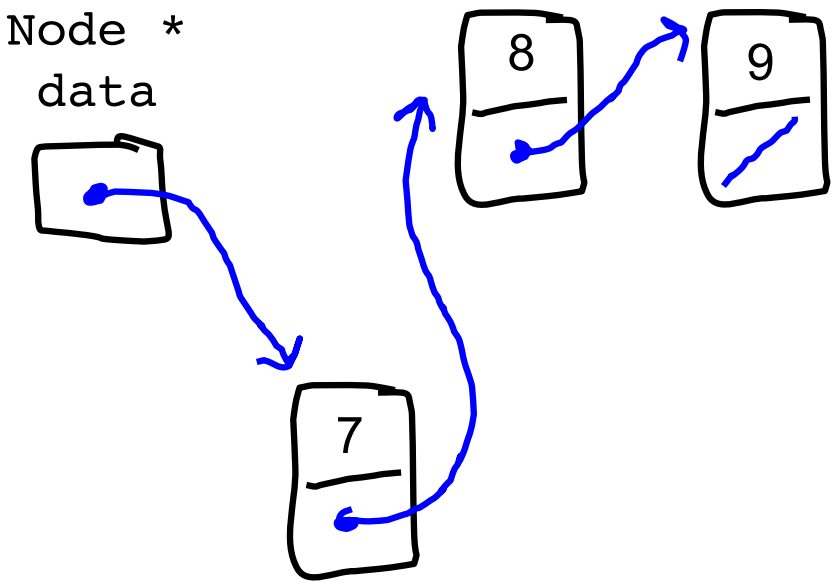


exit function

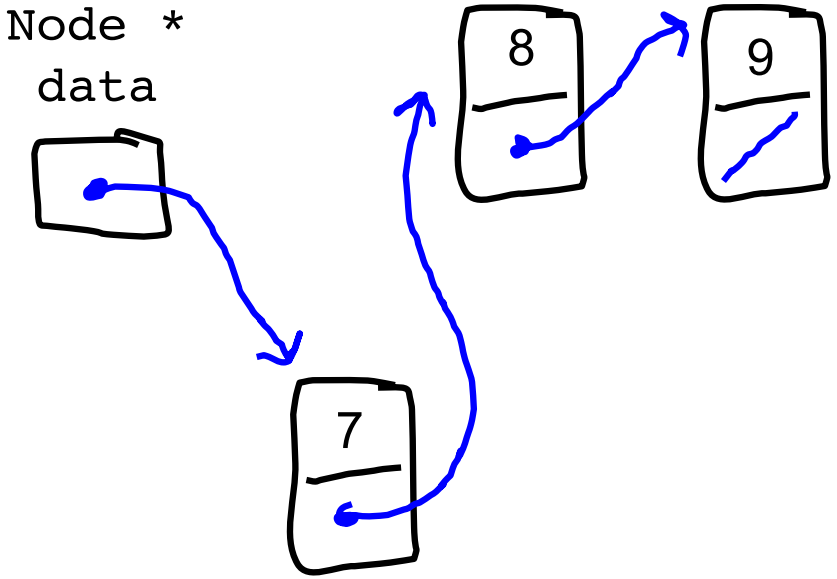


Stack is a Linked List

```
pop();
```



Stack is a Linked List



pop();

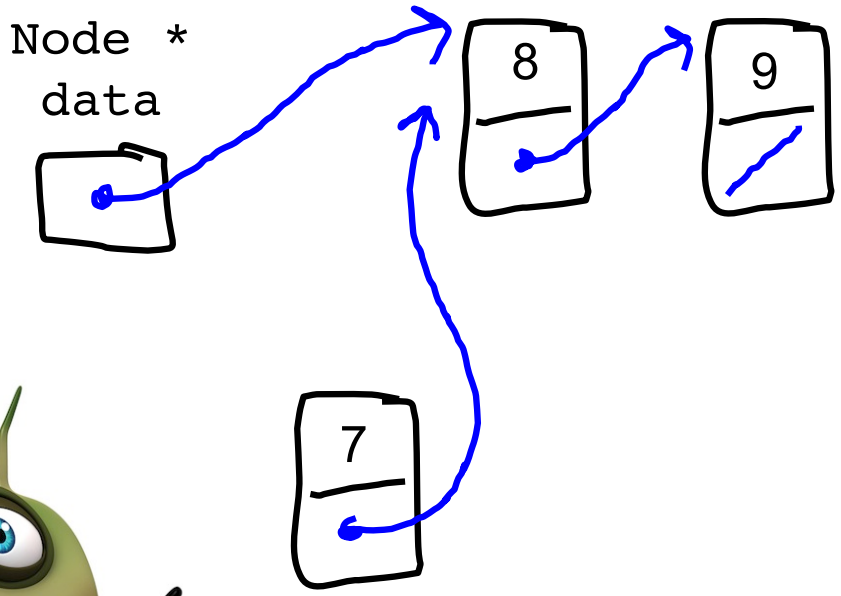
int toReturn

7

```
int toReturn = data->value;
```

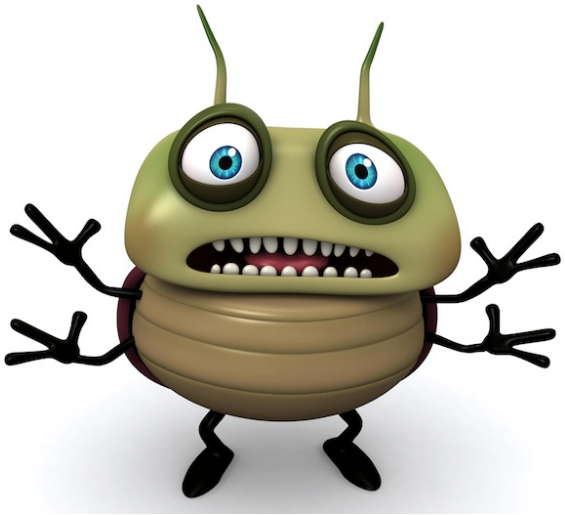
Stack is a Linked List

```
pop();
```




```
int toReturn
```

```
7
```

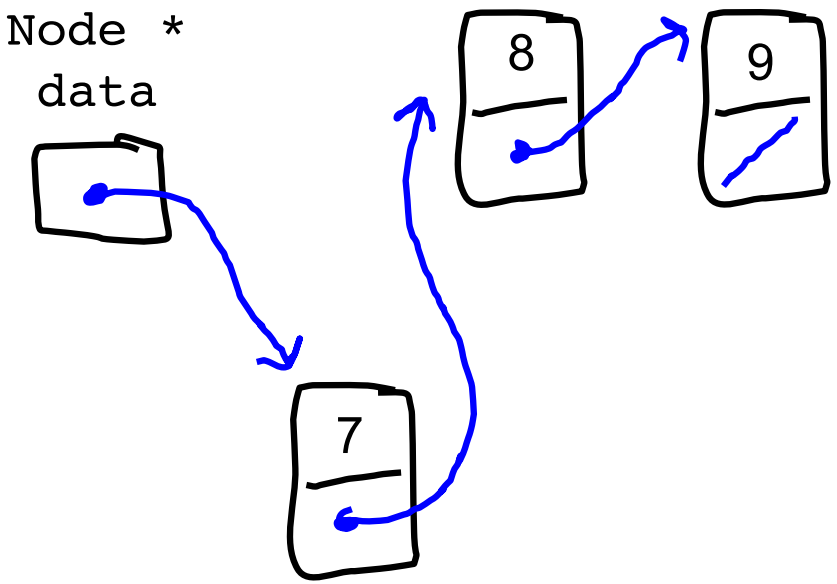


```
data = data->link;
```

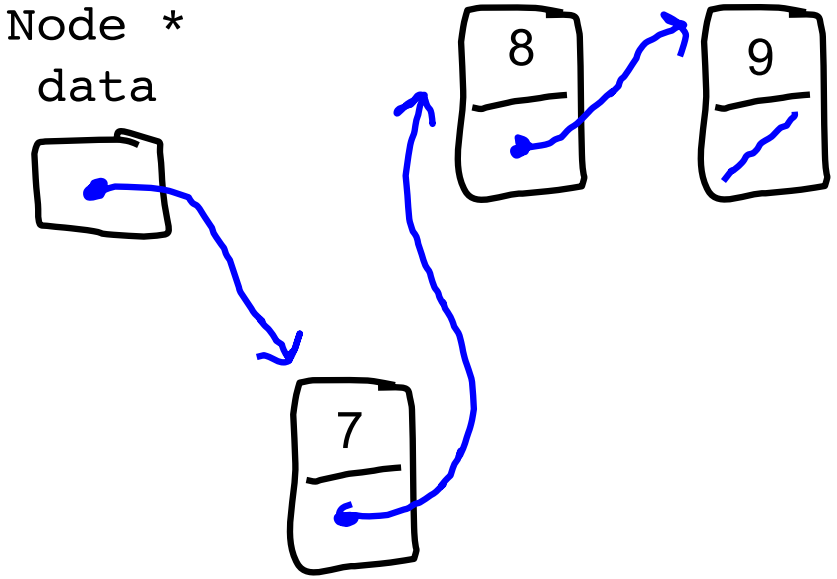
That didn’t work. Let’s try again…

Stack is a Linked List

```
pop();
```



Stack is a Linked List



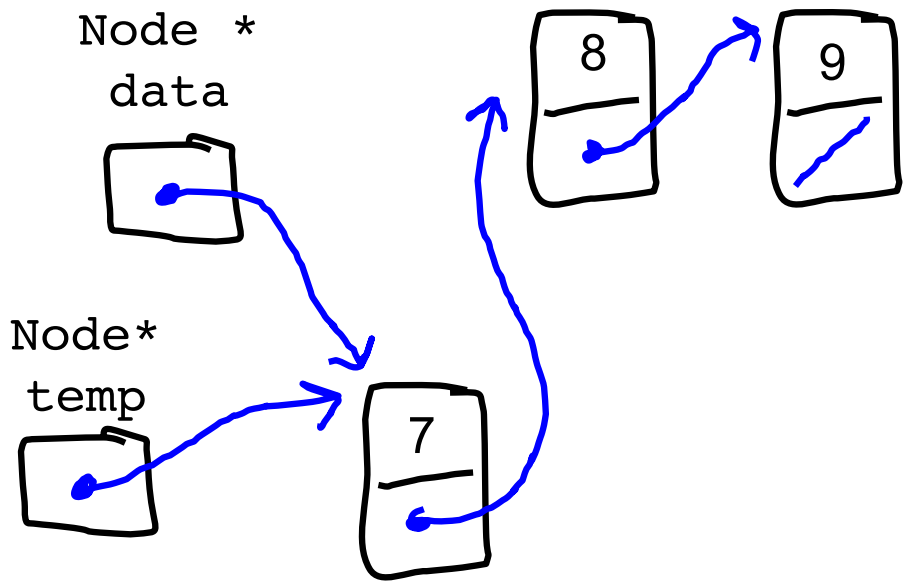
pop();

int toReturn

7

```
int toReturn = data->value;
```

Stack is a Linked List



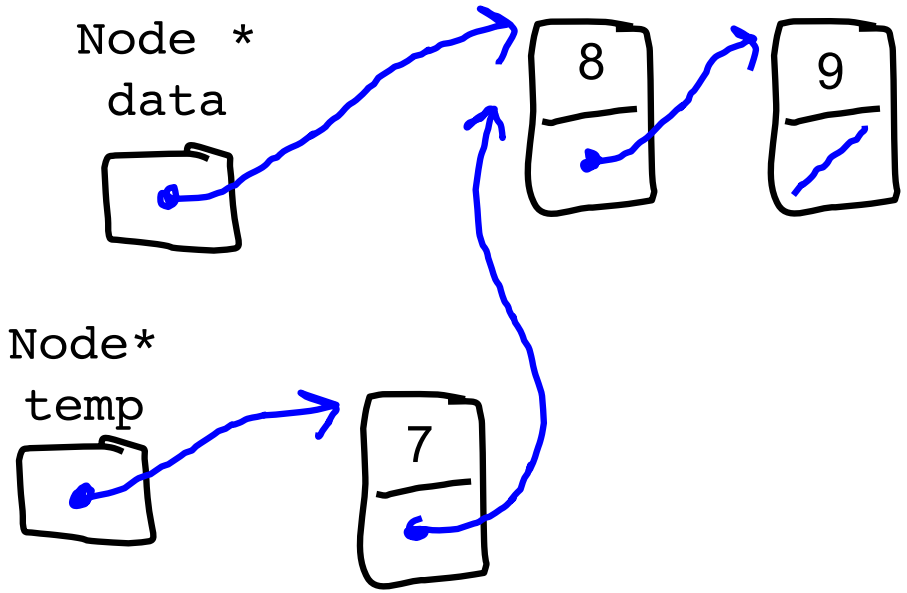

```
pop();
```

```
int toReturn
```

```
7
```

```
Node * temp = data;
```

Stack is a Linked List



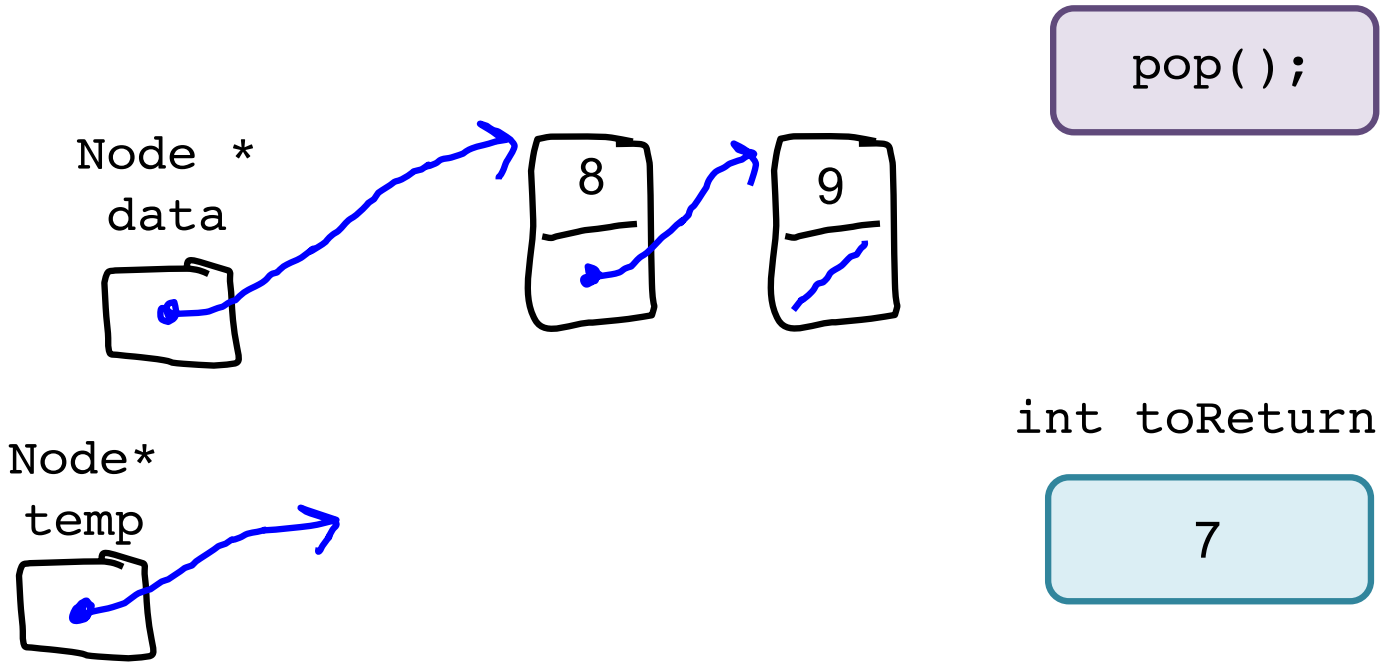

```
pop();
```

```
int toReturn
```

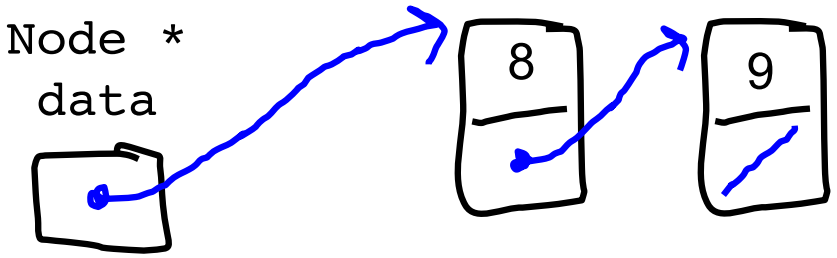
```
7
```

```
data = temp->link;
```

Stack is a Linked List



Stack is a Linked List




```
pop();
```

```
int toReturn
```

7

```
return toReturn;
```

Stack

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

    void push(value);          // append a value
    int pop();                // return the first-in value

private:
    struct Node {
      int value;
      Node * link;
    };
    Node * data;      // member variables
};
```

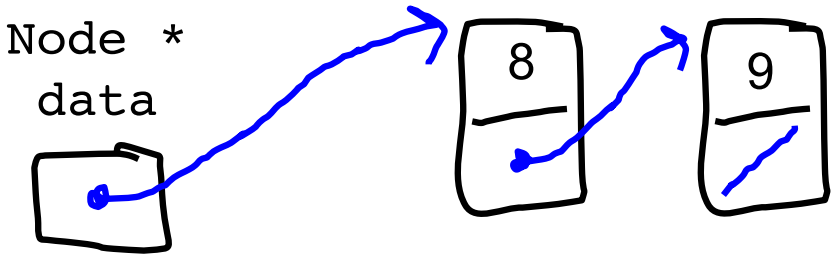
Stack Implementation

```
void StackInt::push(int v) {
    Node * temp = new Node;
    temp->value = v;
    temp->link = data;
    data = temp;
}

int StackInt::pop() {
    int toReturn = data->value;
    Node * temp = data;
    data = temp->link;
    delete temp;
    return toReturn;
}
```

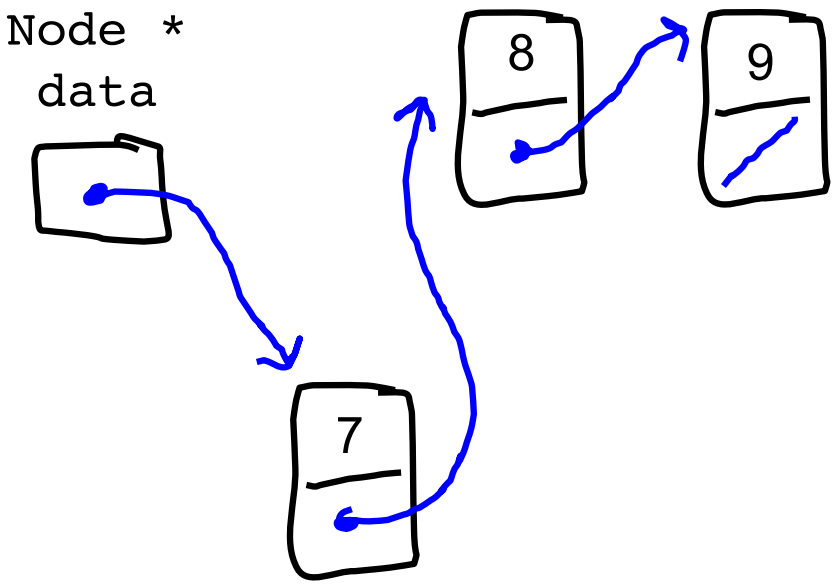
Stack: Big O of Push?

Stack: Big O of Pop?

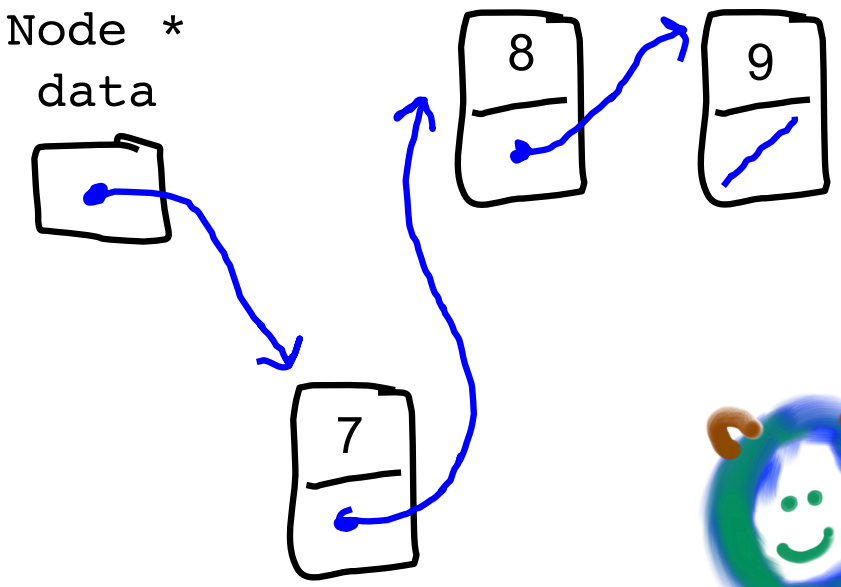
Queue?



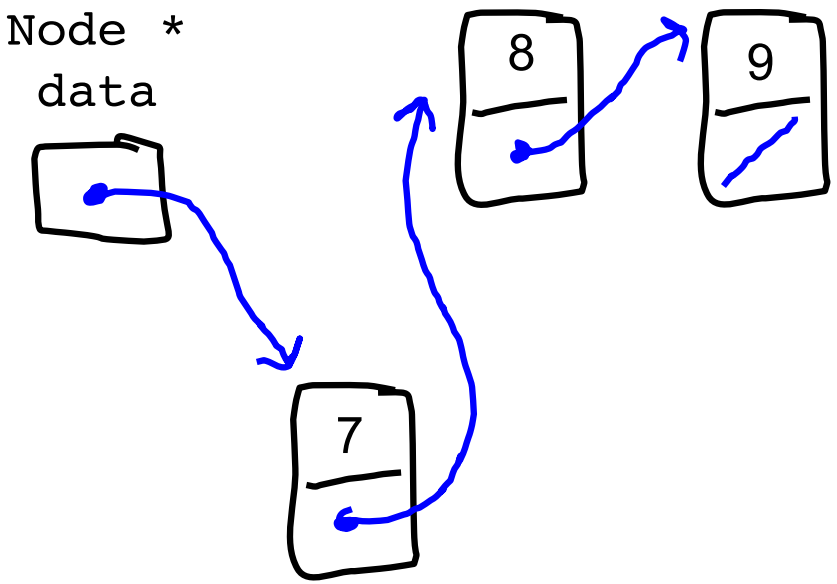
Queue Enqueue?



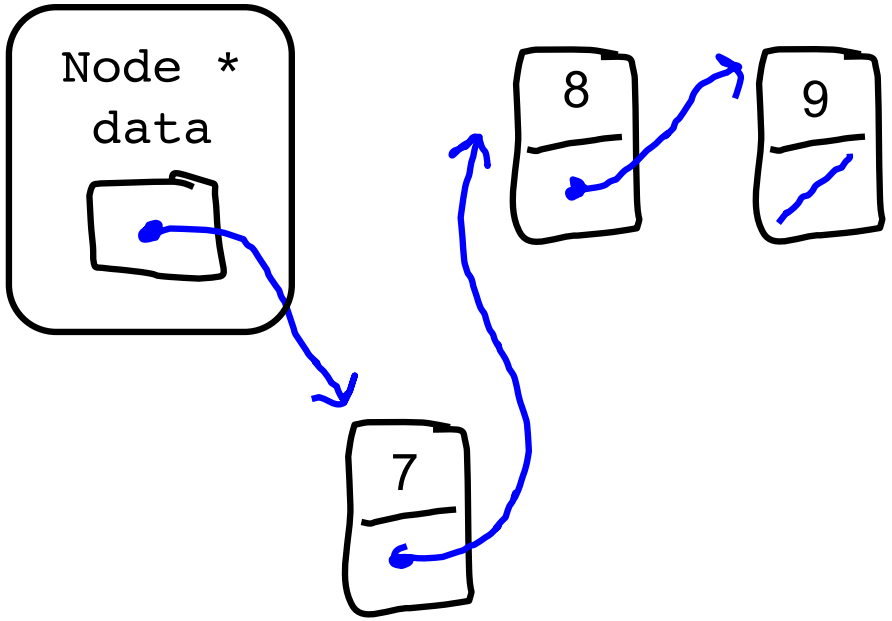
Queue Enqueue?



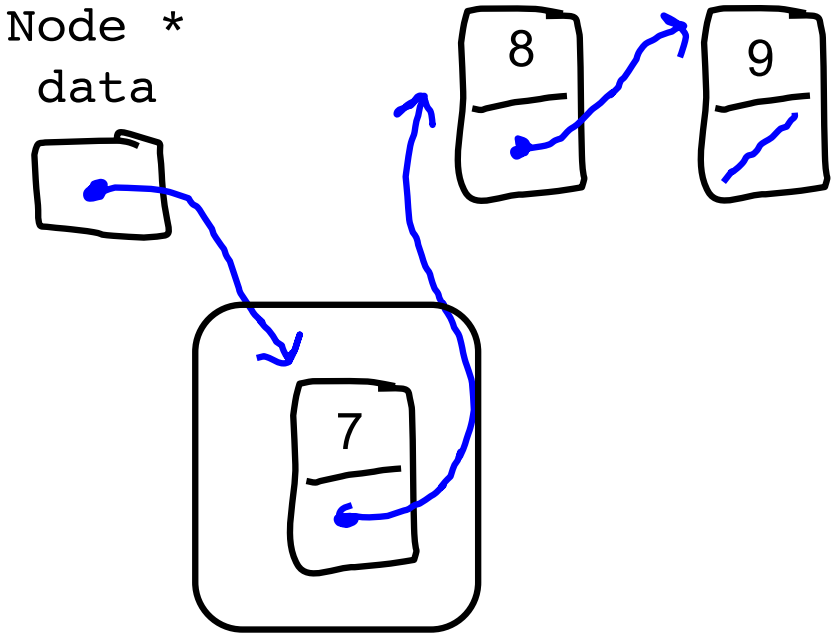
Queue Dequeue?



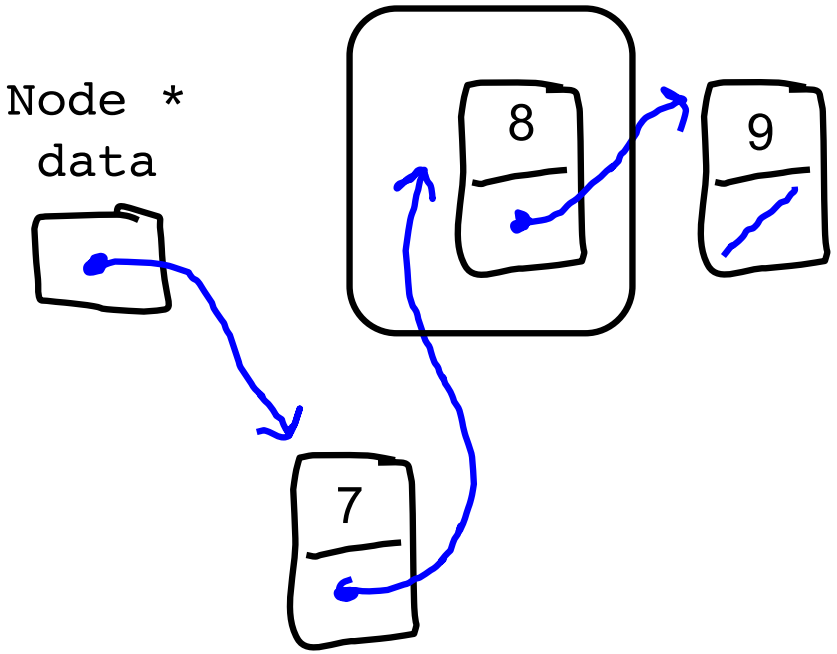
Queue Dequeue?



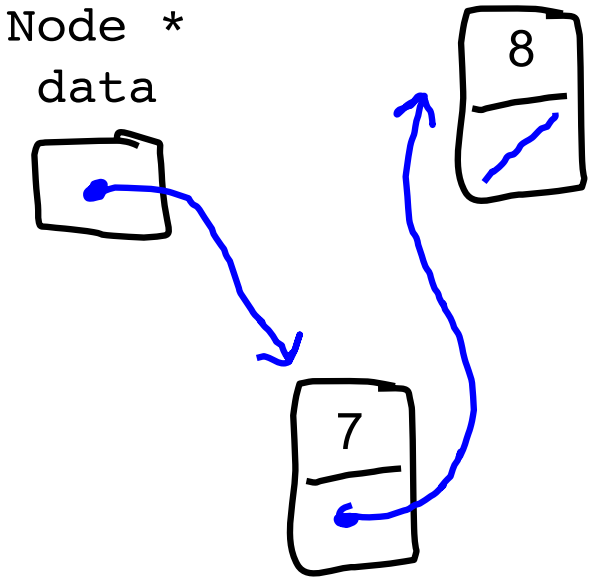
Queue Dequeue?



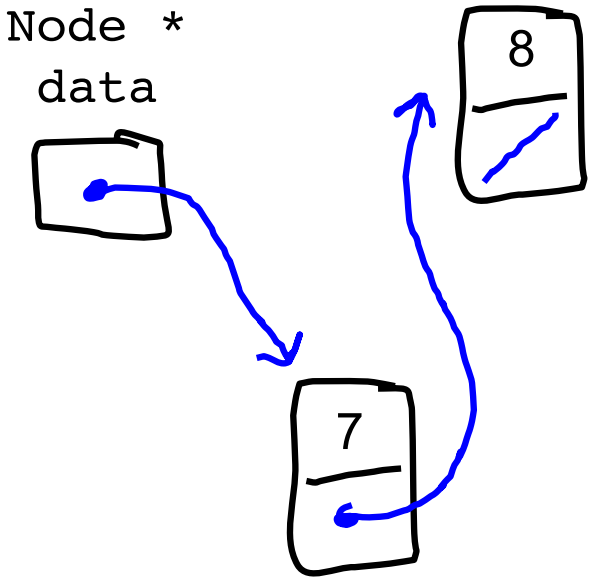
Queue Dequeue?



Queue Dequeue?



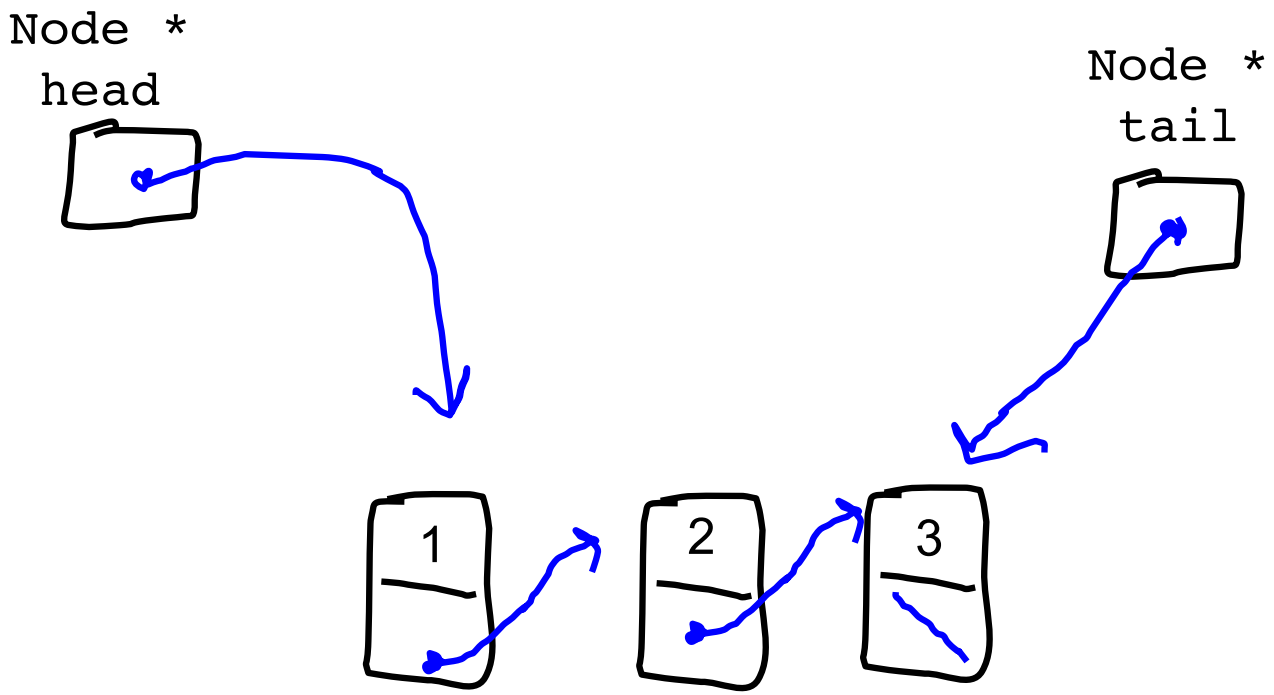
Queue Dequeue?



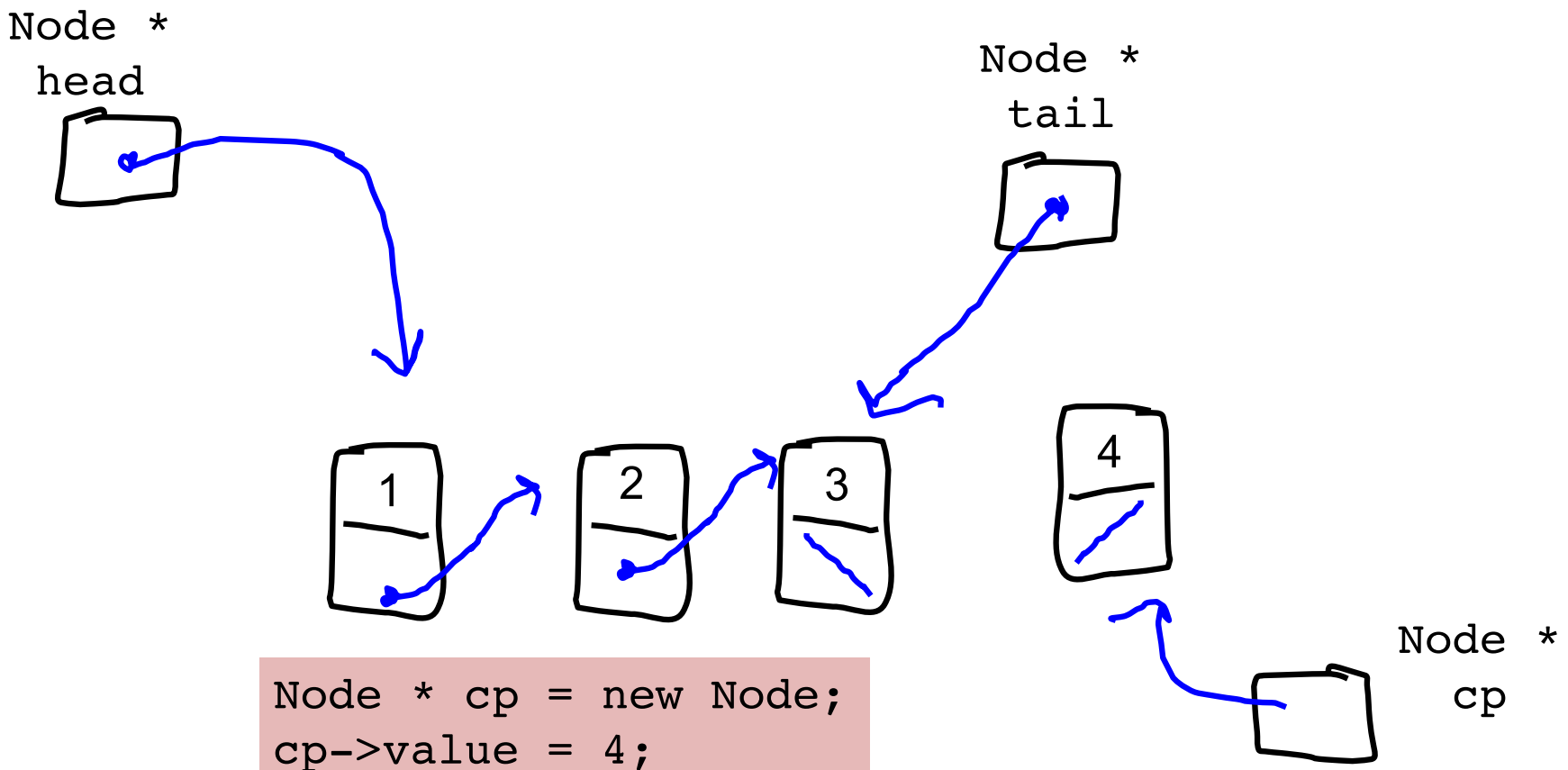
$$\mathcal{O}(n)$$

Always a Better Way

Actual Queue: Enqueue

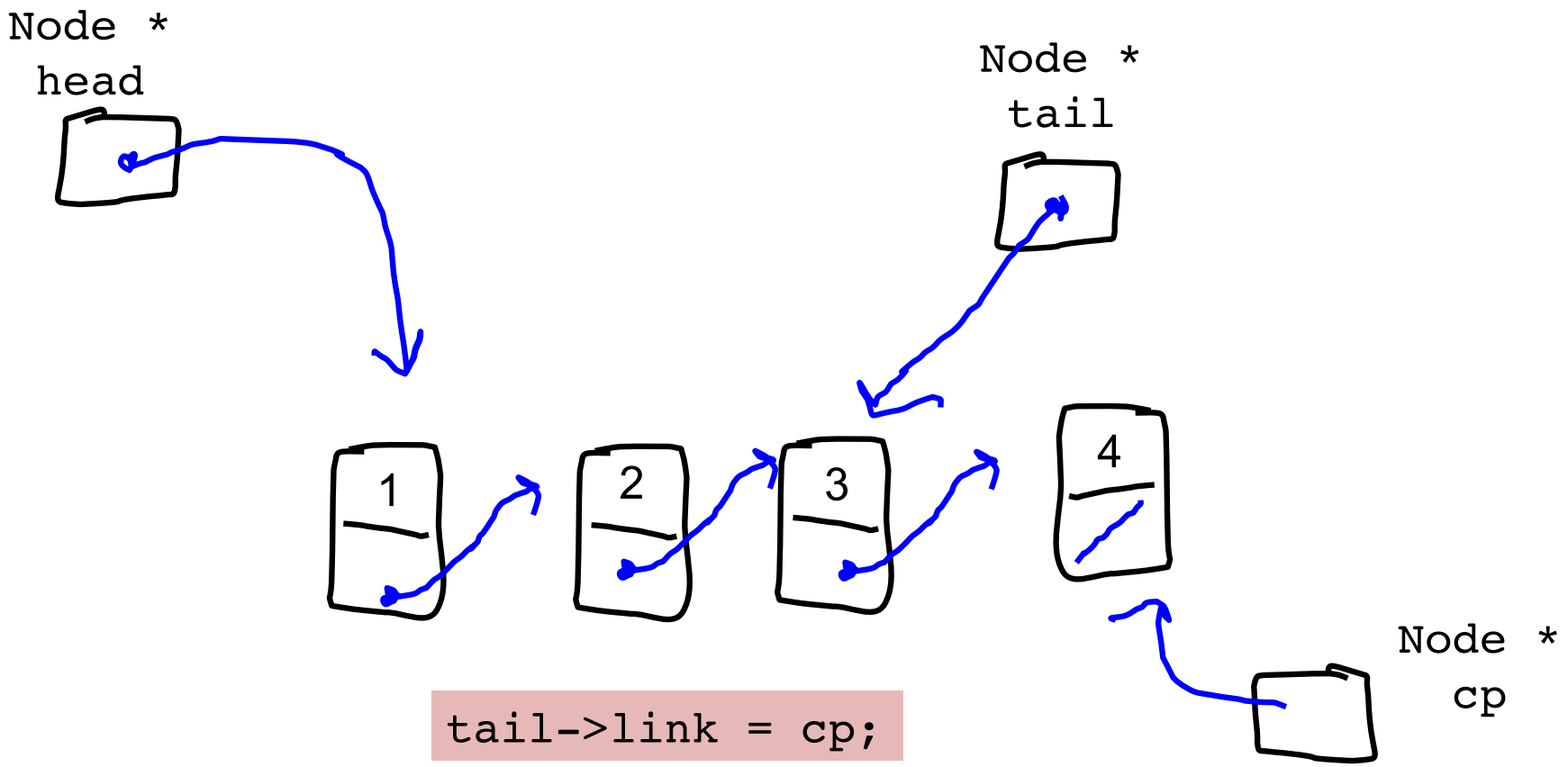


Actual Queue: Enqueue

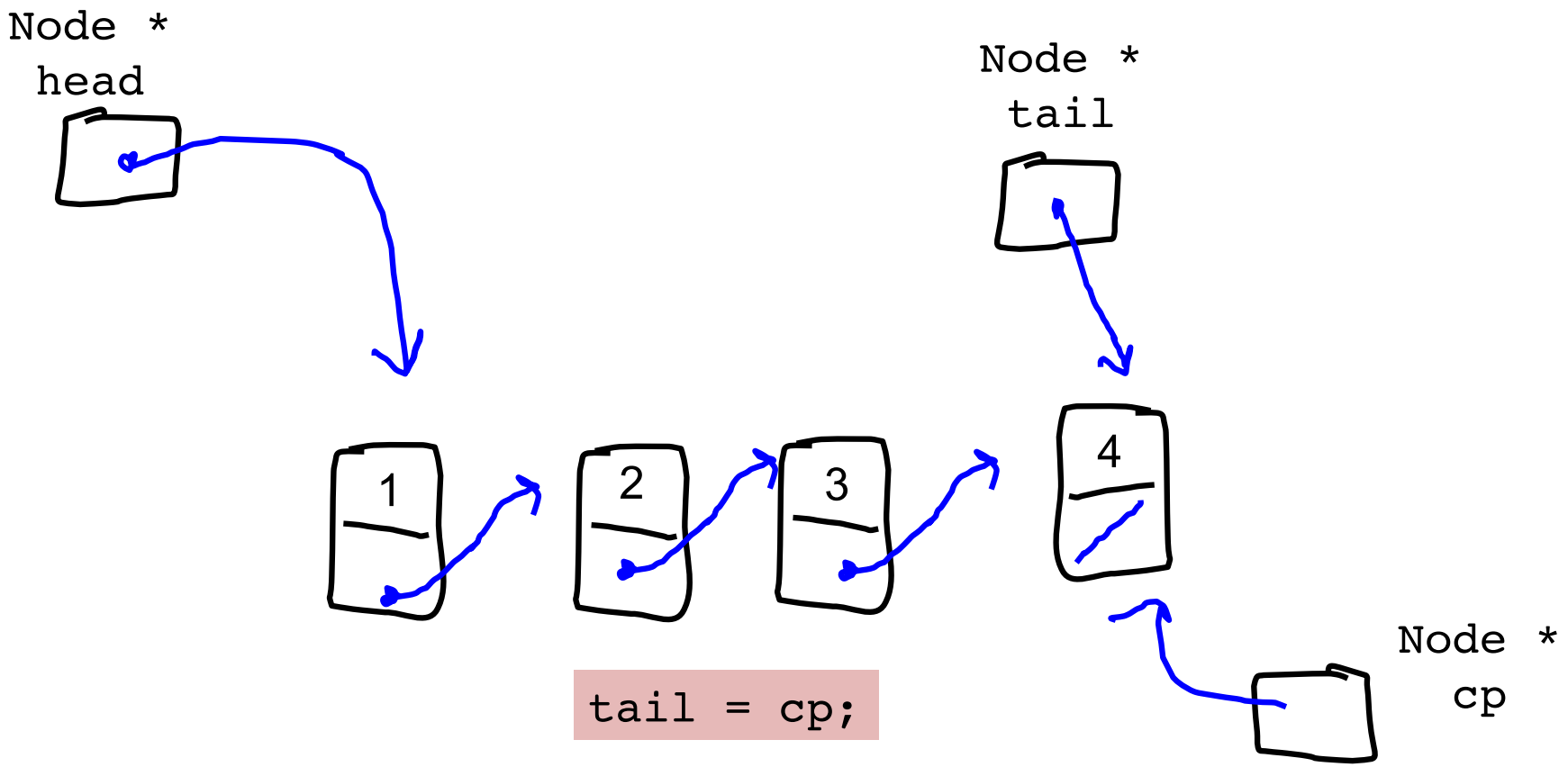



```
Node * cp = new Node;
cp->value = 4;
```

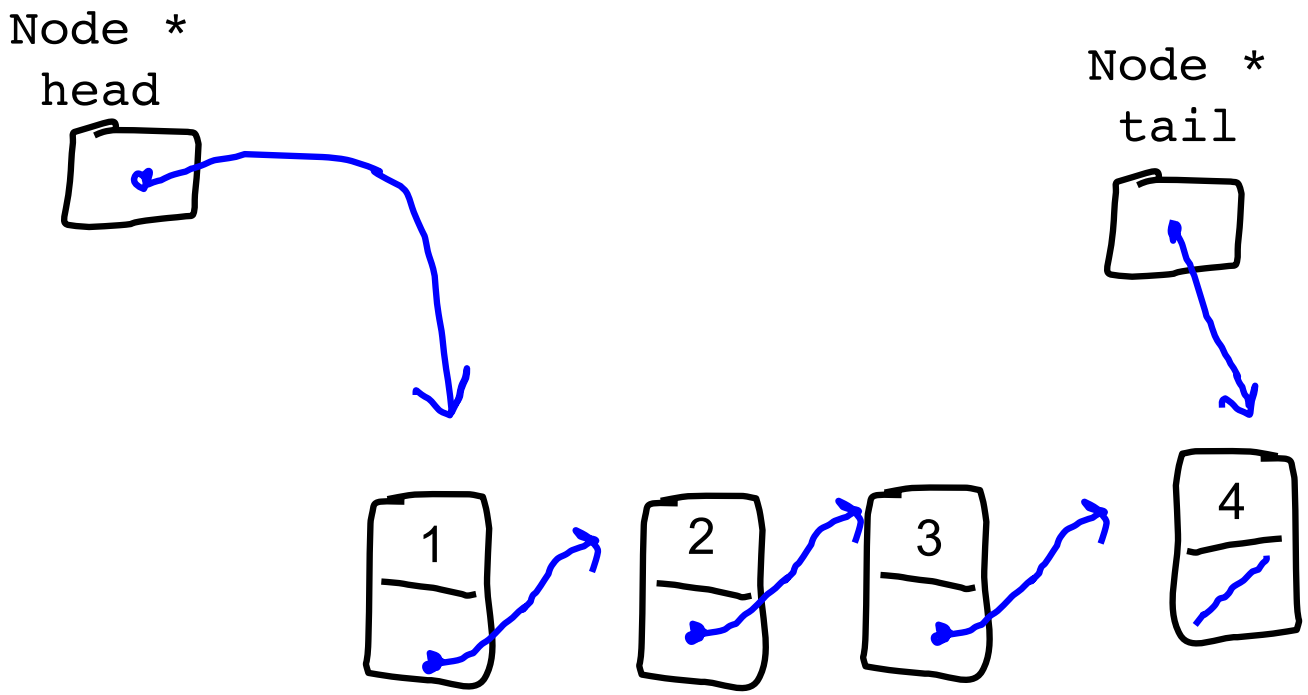
Actual Queue: Enqueue



Actual Queue: Enqueue

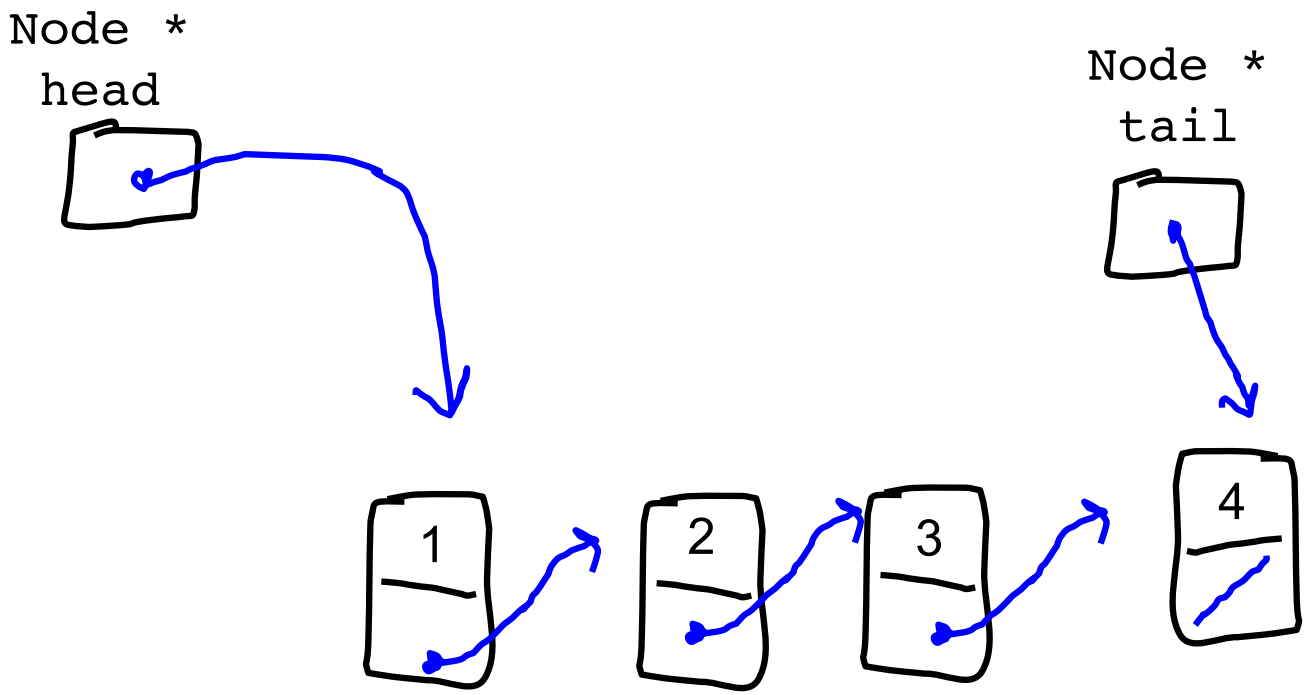


Actual Queue: Enqueue



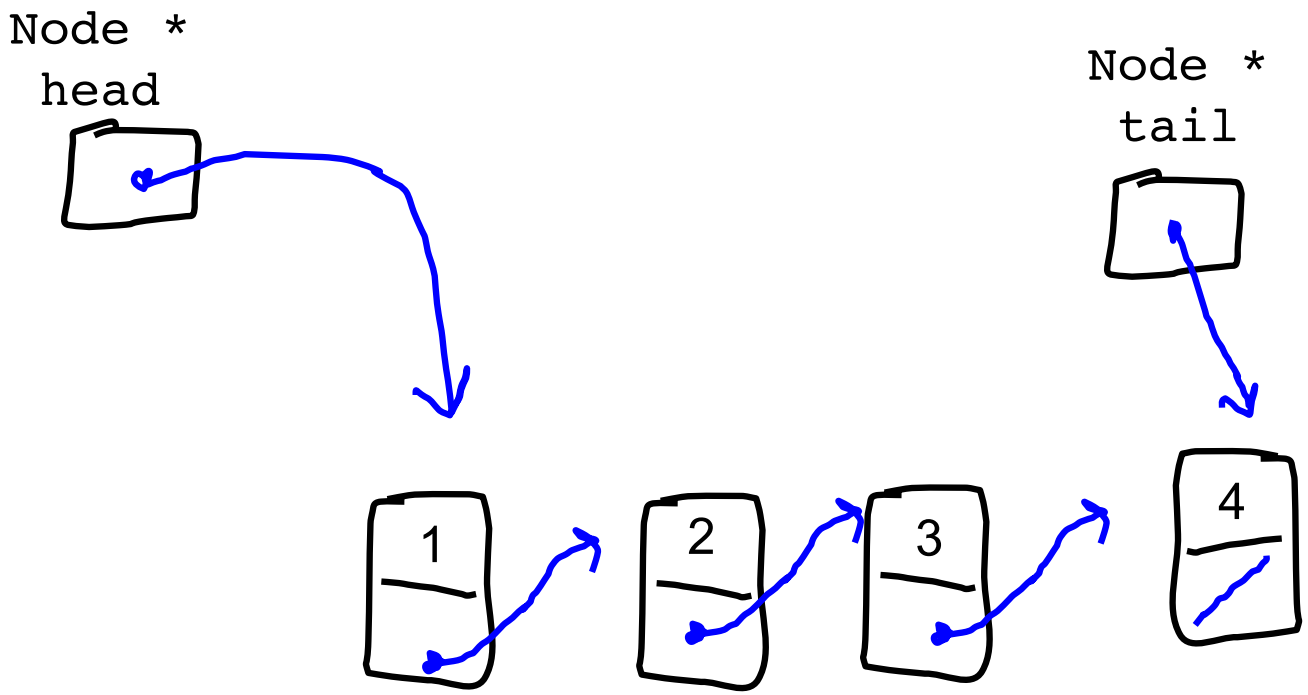

```
return;
```

Actual Queue: Enqueue

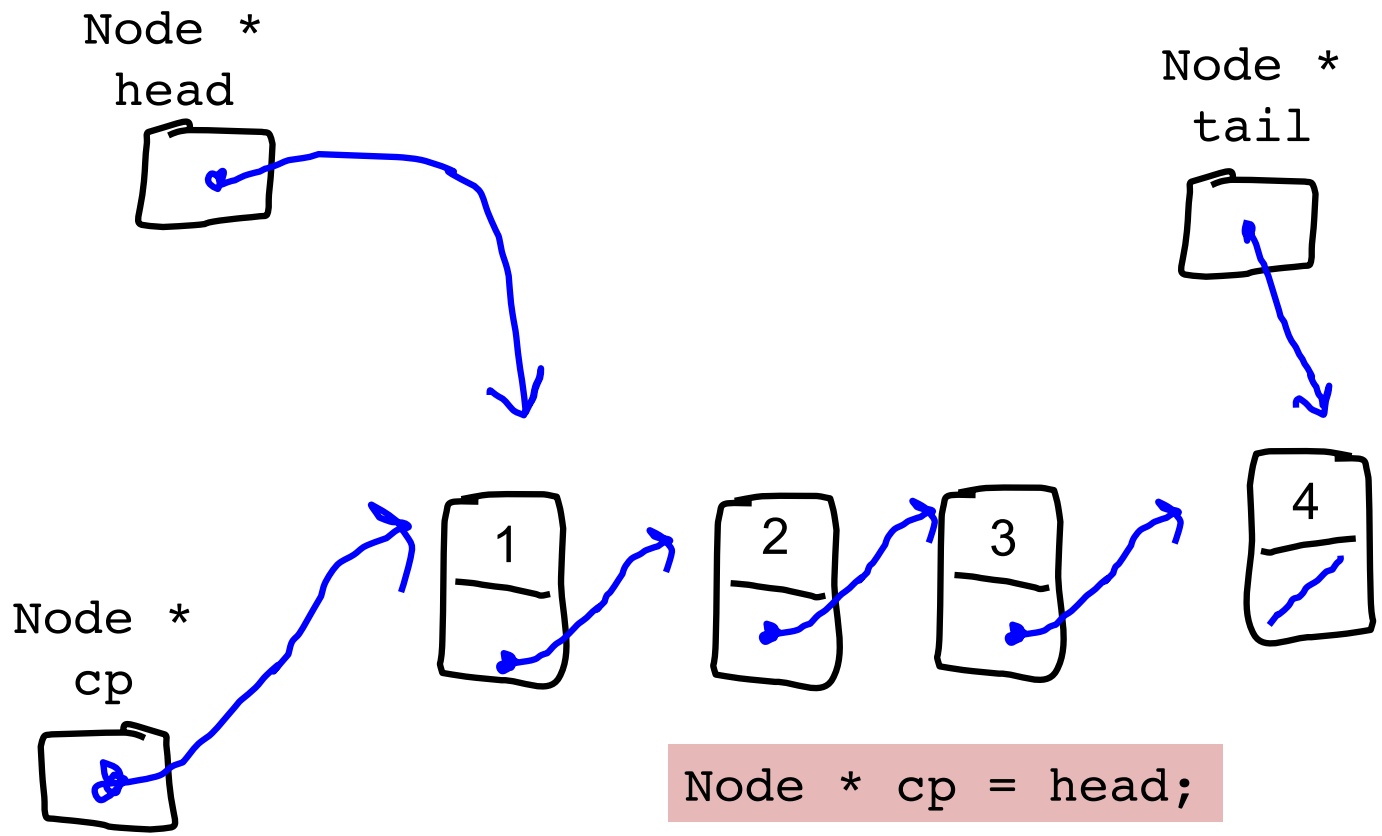


Dequeue

Actual Queue: Dequeue

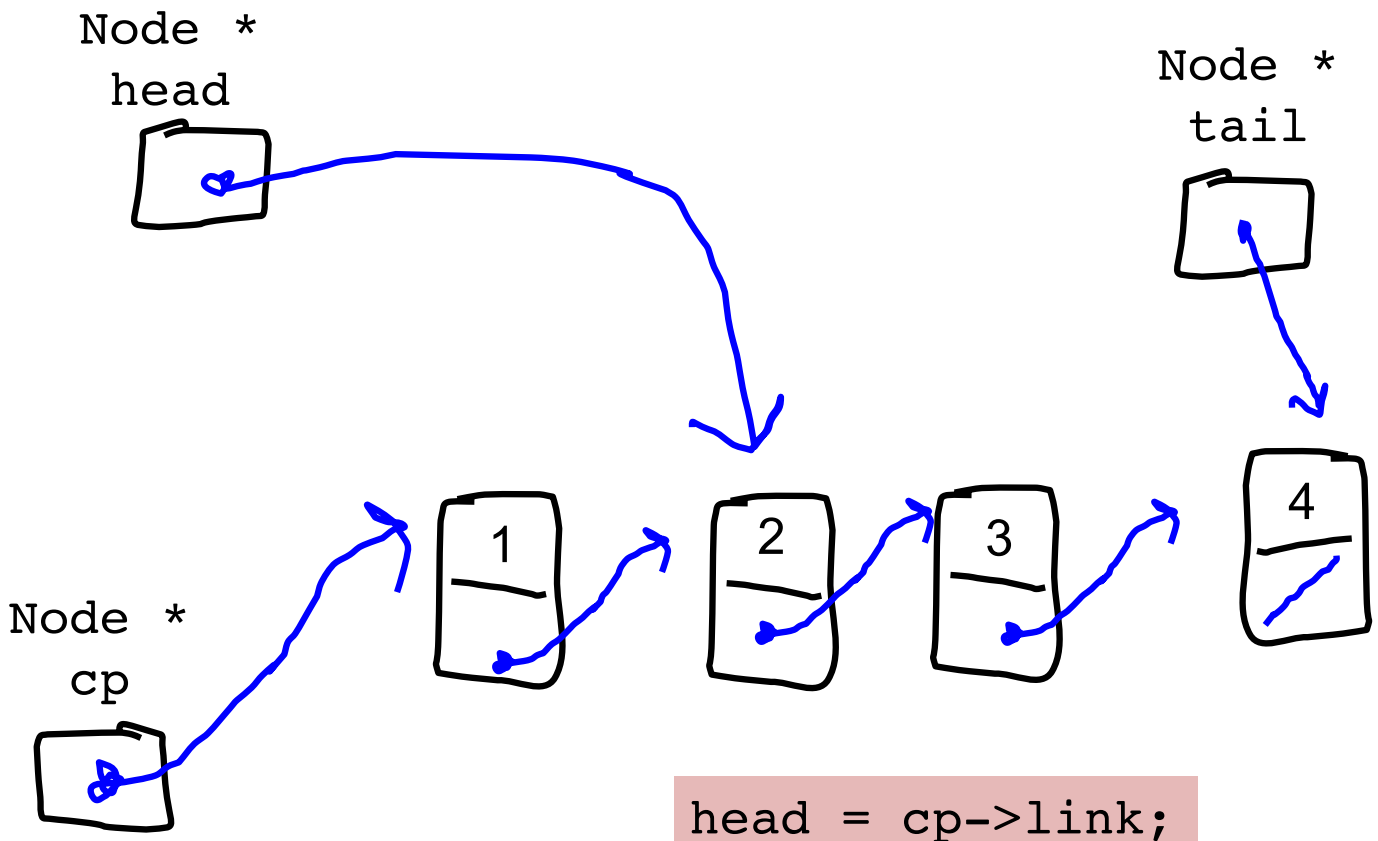


Actual Queue: Dequeue



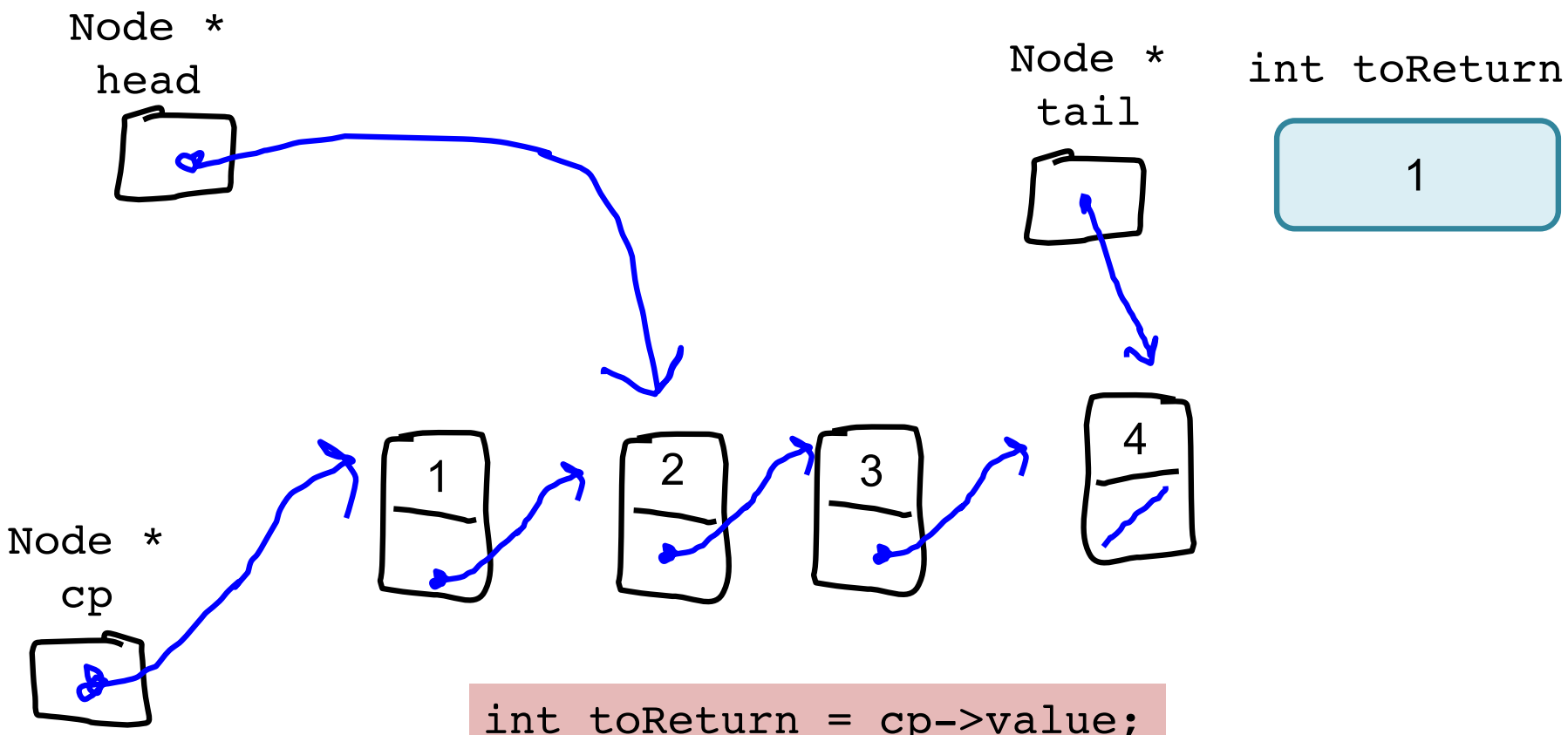

```
Node * cp = head;
```

Actual Queue: Dequeue

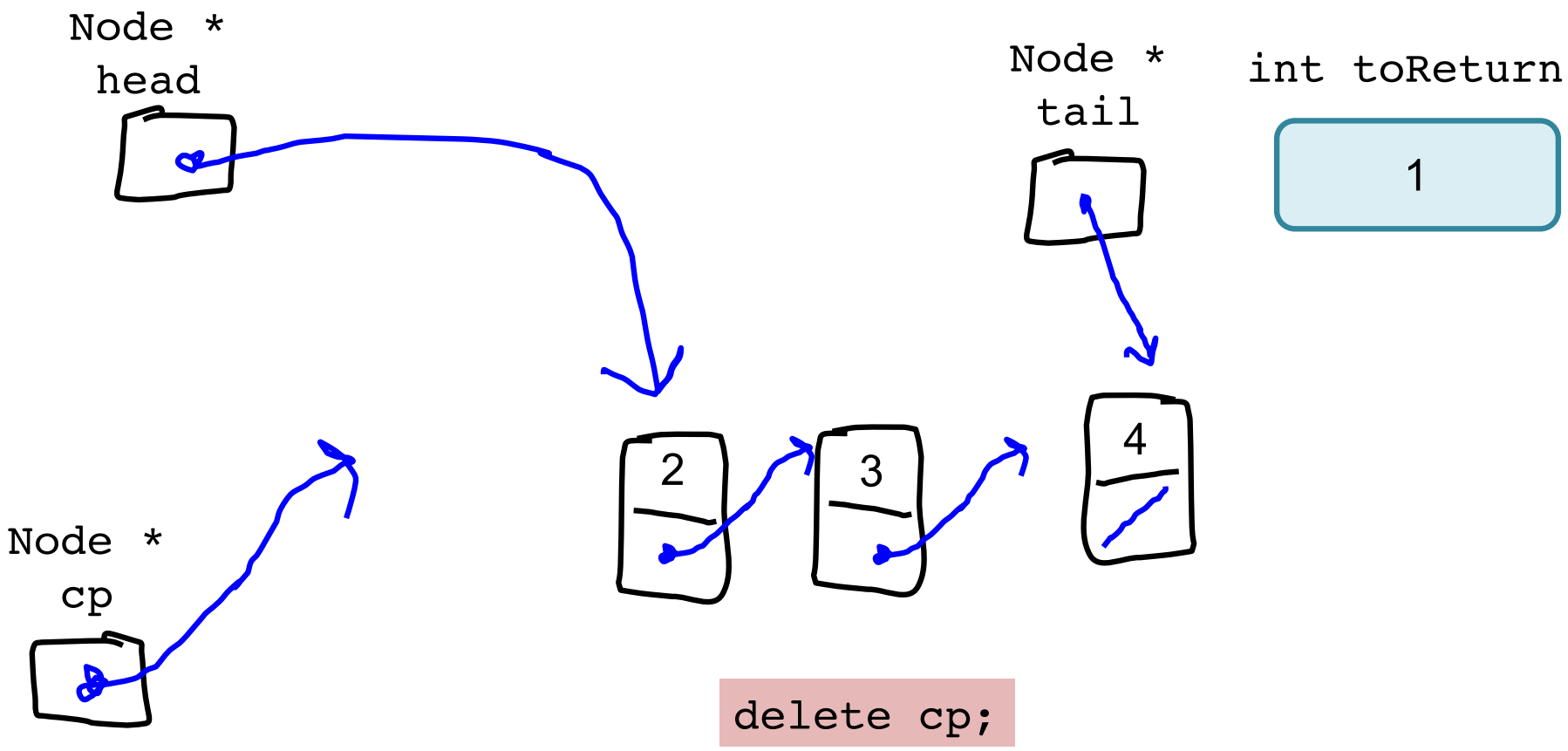



```
head = cp->link;
```

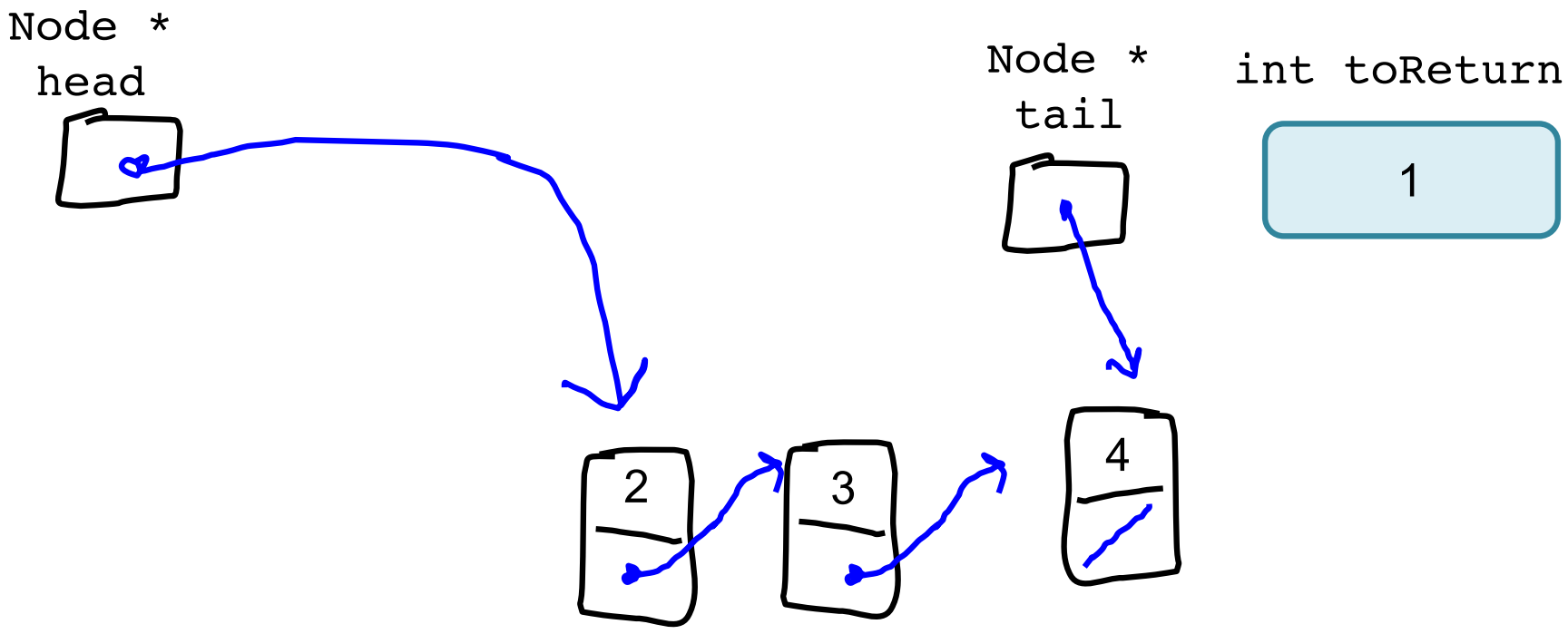
Actual Queue: Dequeue



Actual Queue: Dequeue

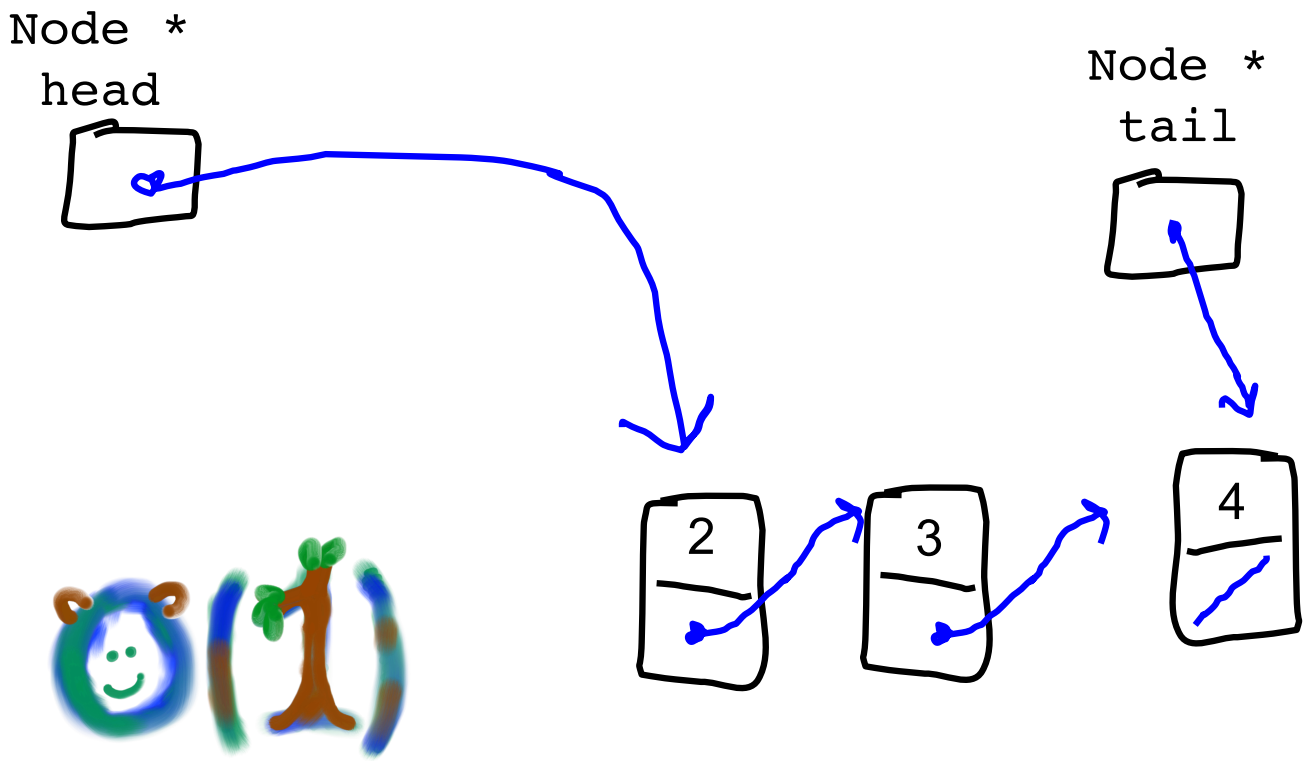


Actual Queue: Dequeue




```
return toReturn;
```

Actual Queue: Dequeue



Queue

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

    void enqueue(int value); // append a value
    int dequeue();             // return the first-in value

private:
    struct Node {
      int value;
      Node * link;
    };
    Node * head;      // has a pointer to the first node
    Node * tail;      // and a pointer to the last node
};
```

Queue Implementation

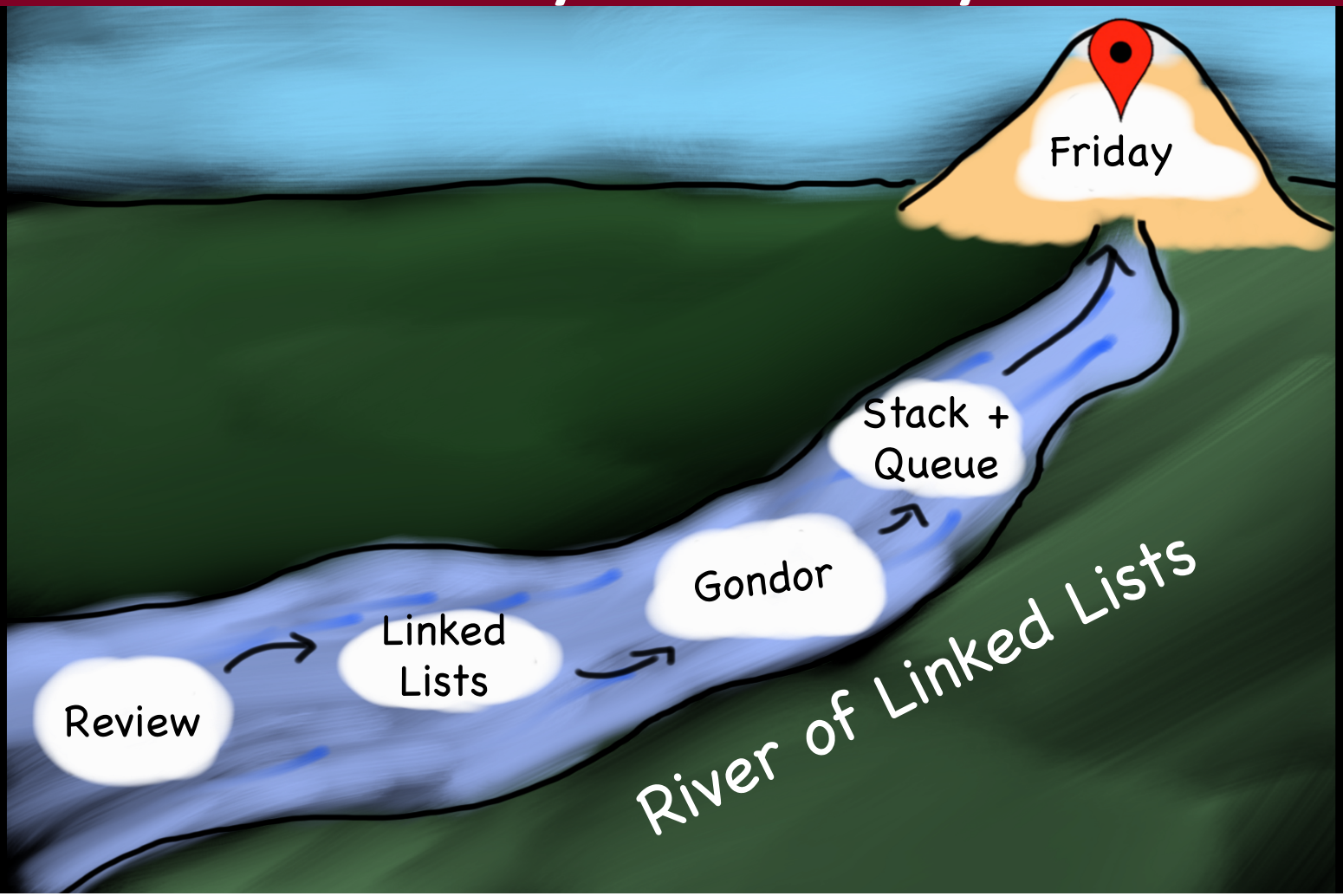
```
void QueueInt::enqueue(int v) {
    Node * temp = new Node;
    temp->value = v;
    tail->link = temp;
    tail = temp;
}

int QueueInt::dequeue() {
    int toReturn = head->value;
    Node * temp = head;
    head = temp->link;
    delete temp;
    return toReturn;
}
```

Linked Lists are Excellent

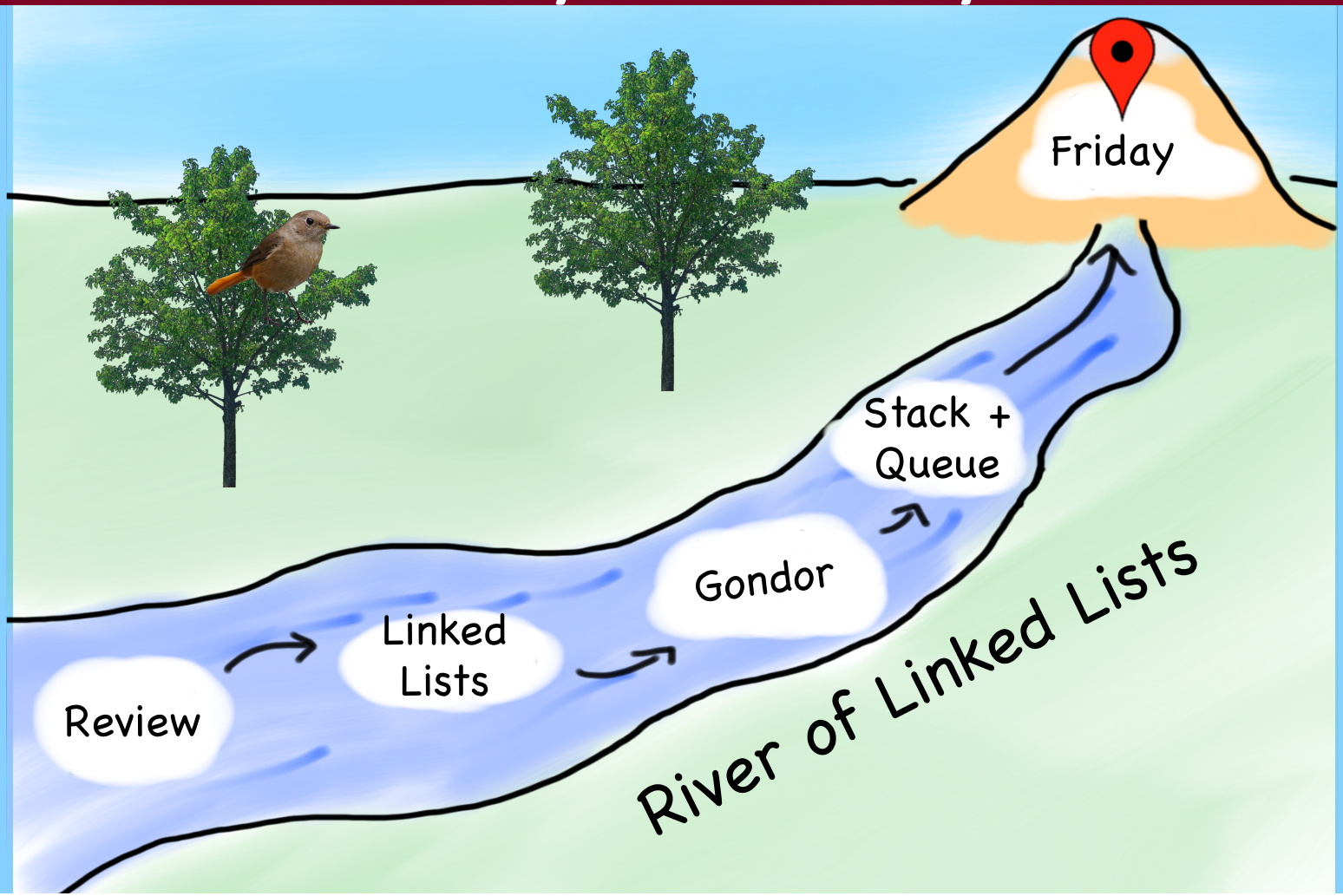
	Worst
Stack Push	$\mathcal{O}(1)$
Stack Pop	$\mathcal{O}(1)$
Queue Enqueue	$\mathcal{O}(1)$
Queue Dequeue	$\mathcal{O}(1)$

Today's Journey

Today's Journey



Today's Journey



Today's Goals

1. Learn linked lists
2. See how Stack + Queue work
3. Get ready for trees…