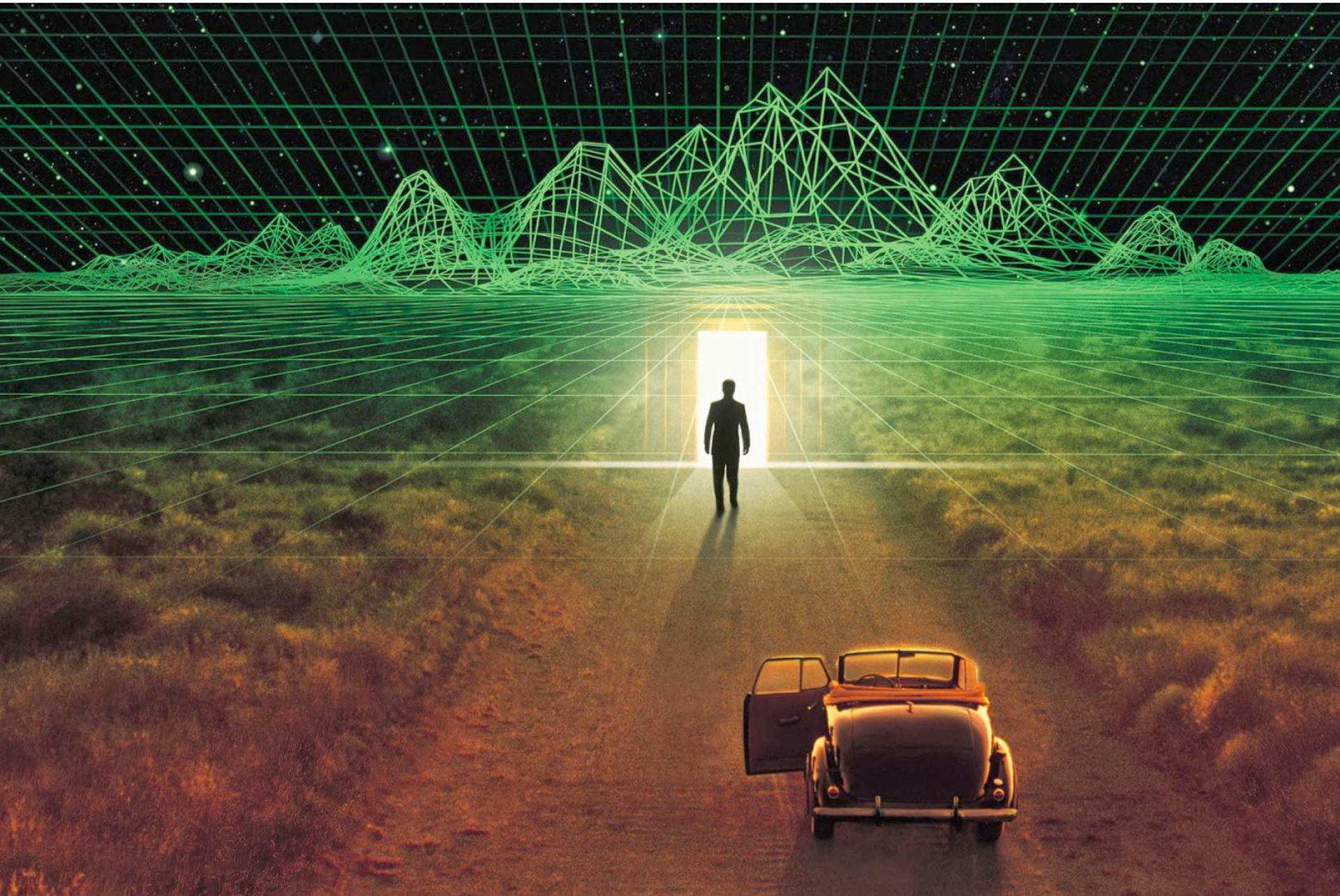
Linked Lists II

Chris Piech

CS 106B
Lecture 14
Feb 8, 2016

Contest



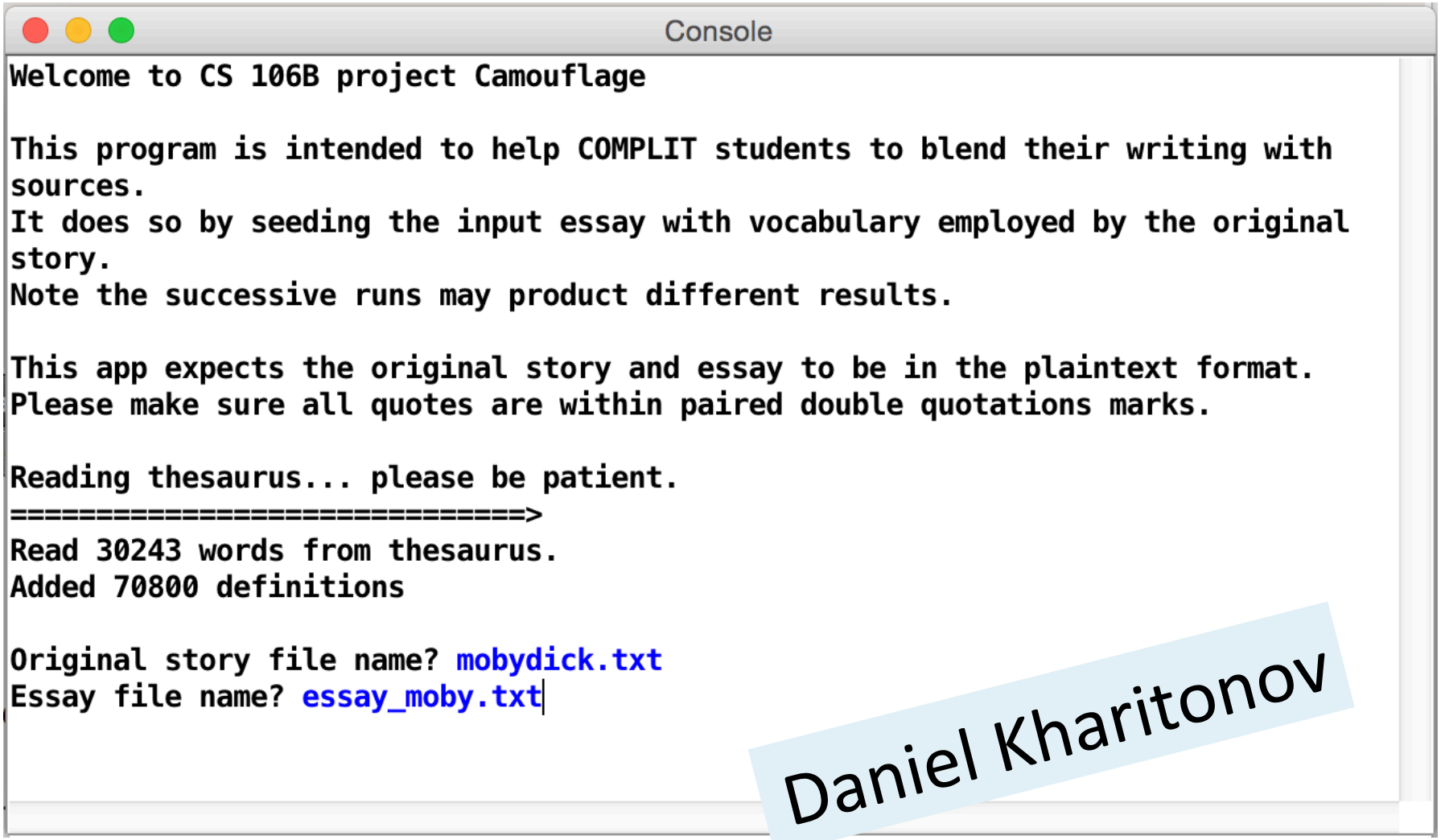
Thanks to all of those who submitted

Finalists

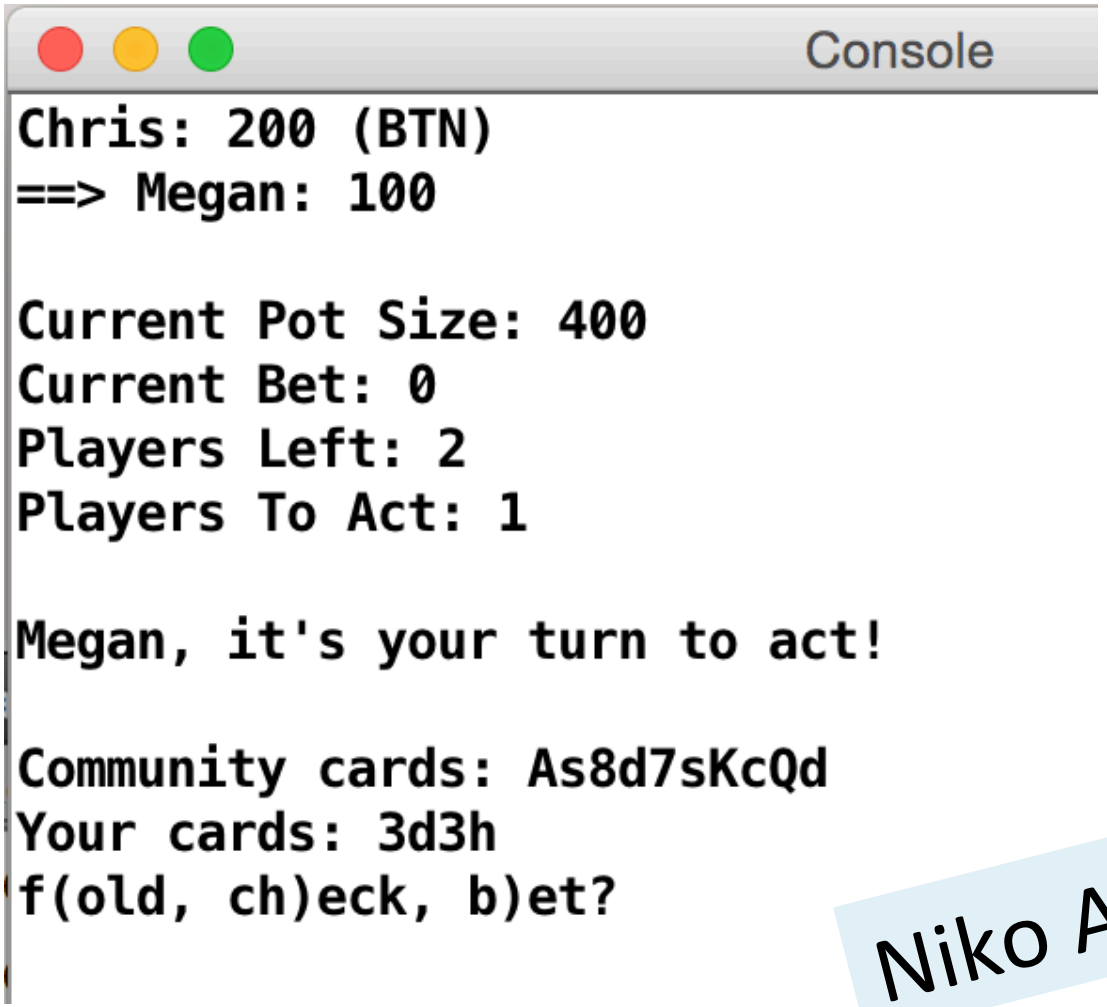
68^2	29^2	41^2	37^2
17^2	31^2	79^2	32^2
59^2	28^2	23^2	61^2
11^2	77^2	8^2	49^2

Robert Haag

Finalist

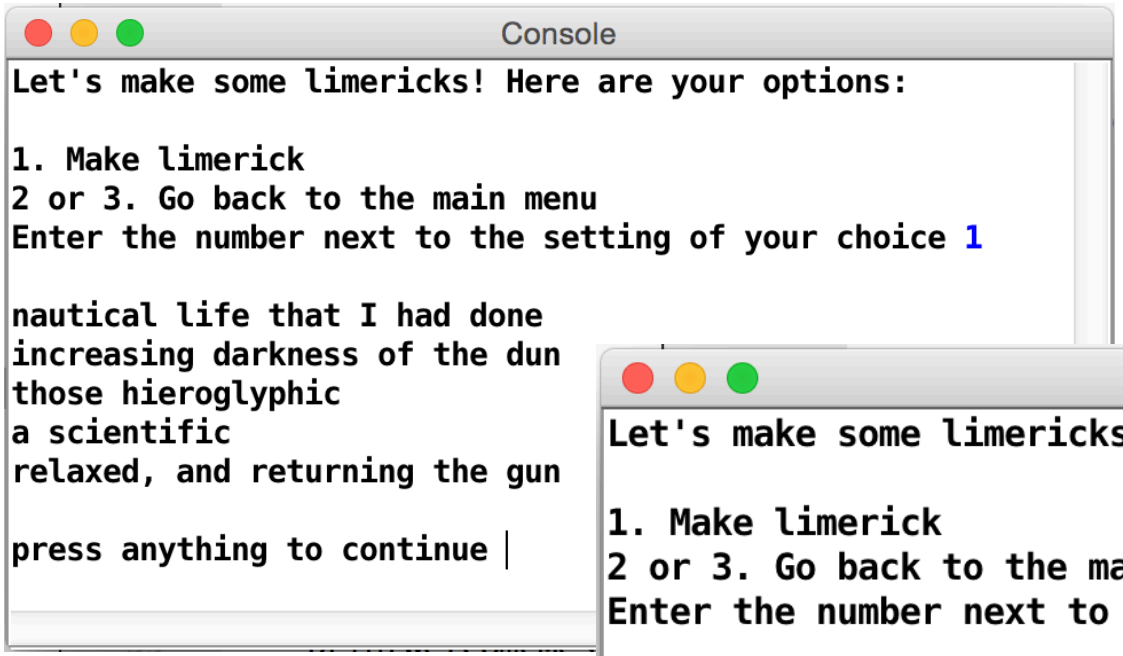

Console
Welcome to CS 106B project Camouflage
This program is intended to help COMPLIT students to blend their writing with sources.
It does so by seeding the input essay with vocabulary employed by the original story.
Note the successive runs may product different results.
This app expects the original story and essay to be in the plaintext format.
Please make sure all quotes are within paired double quotations marks.
Reading thesaurus... please be patient.
==============================>
Read 30243 words from thesaurus.
Added 70800 definitions
Original story file name? mobydick.txt
Essay file name? essay_moby.txt
Daniel Kharitonov

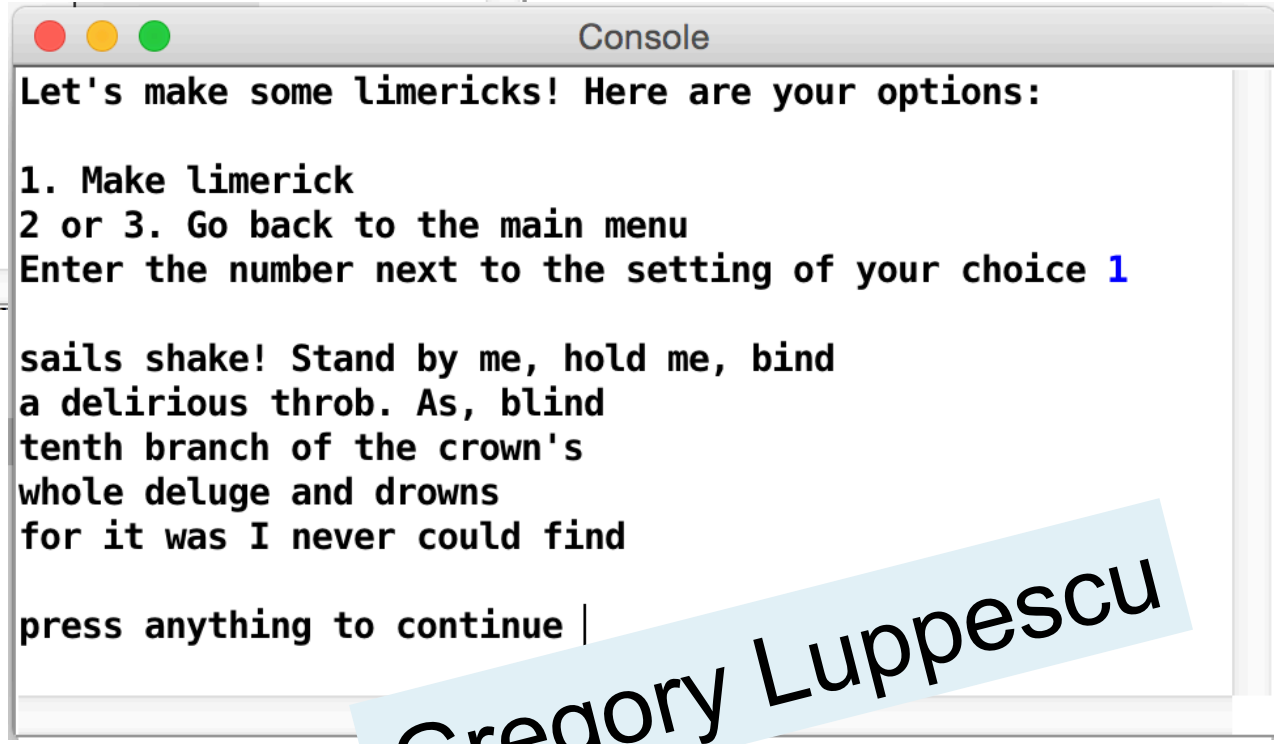
Finalist



Niko Alino

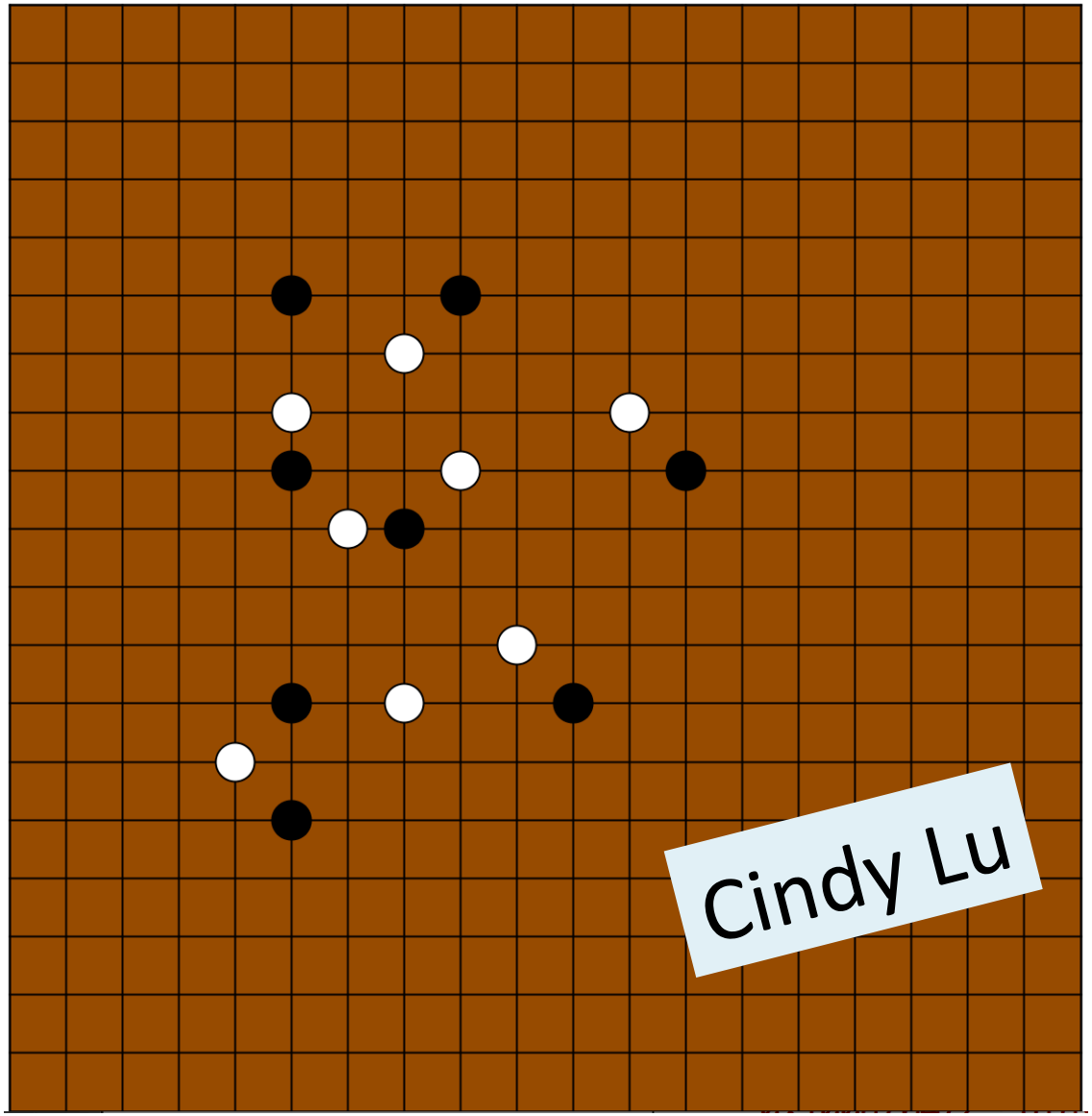
Runner Up - Creativity



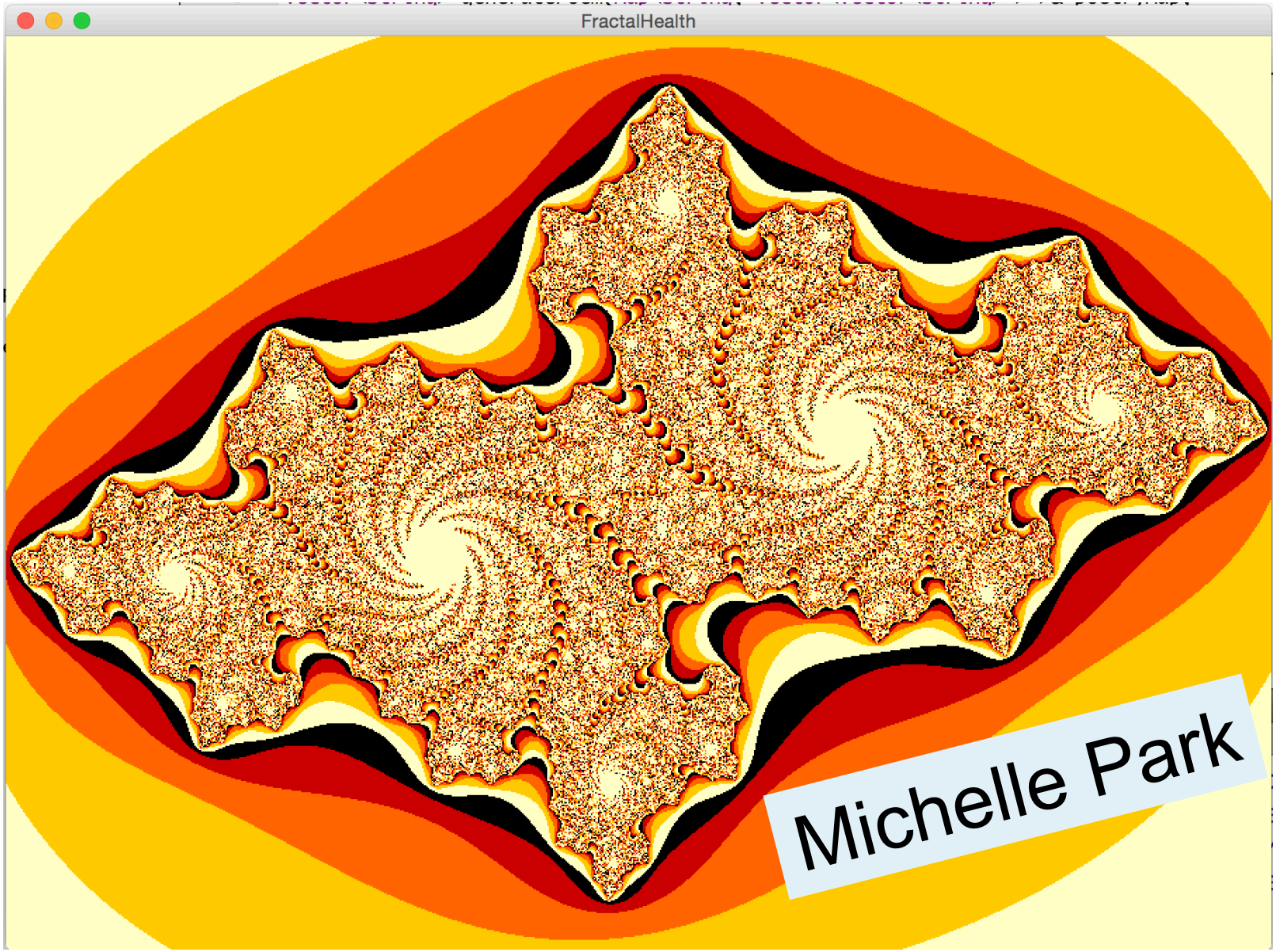


Gregory Luppescu

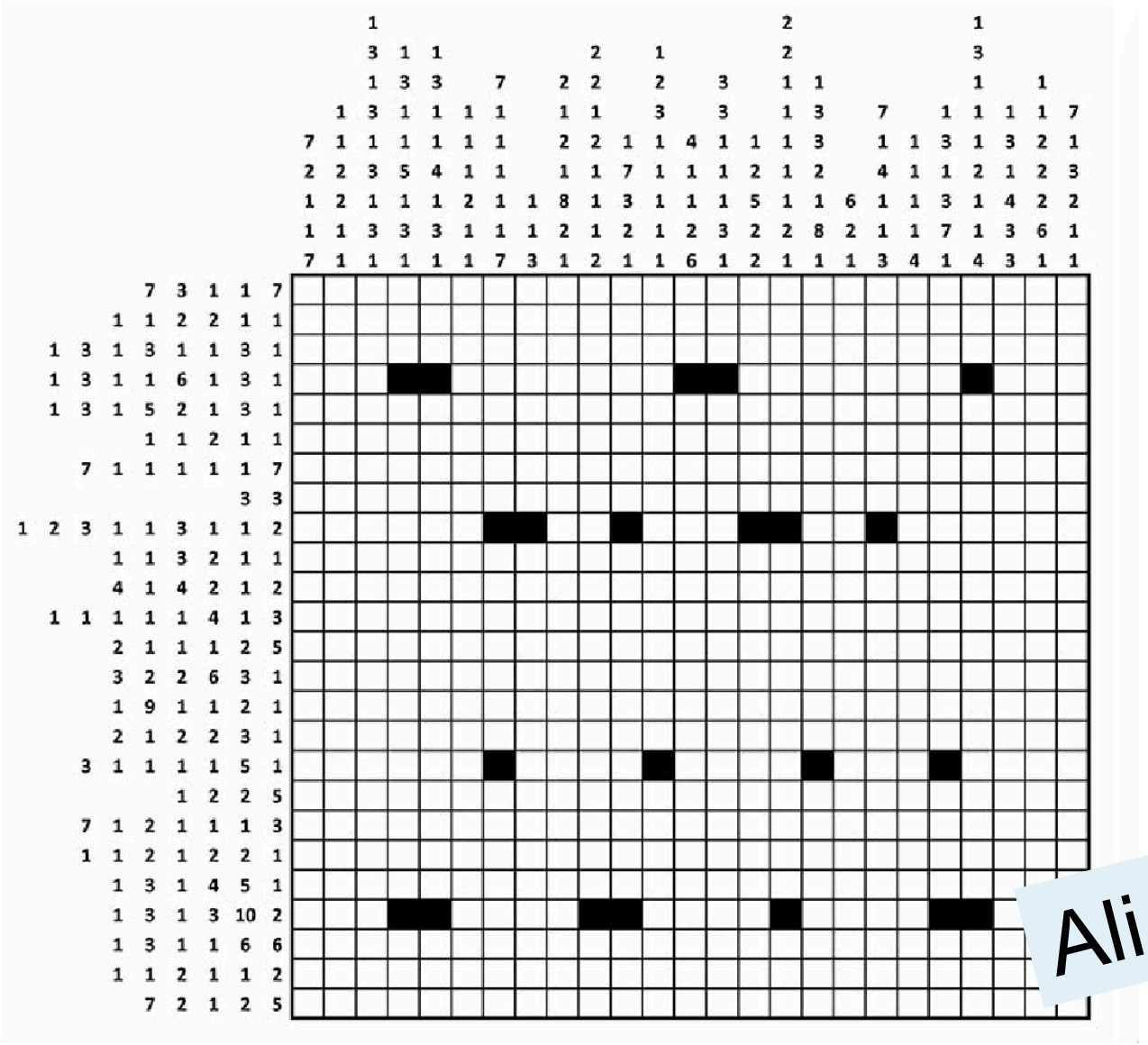
Runner Up - Algorithmic



Winner - Creativity



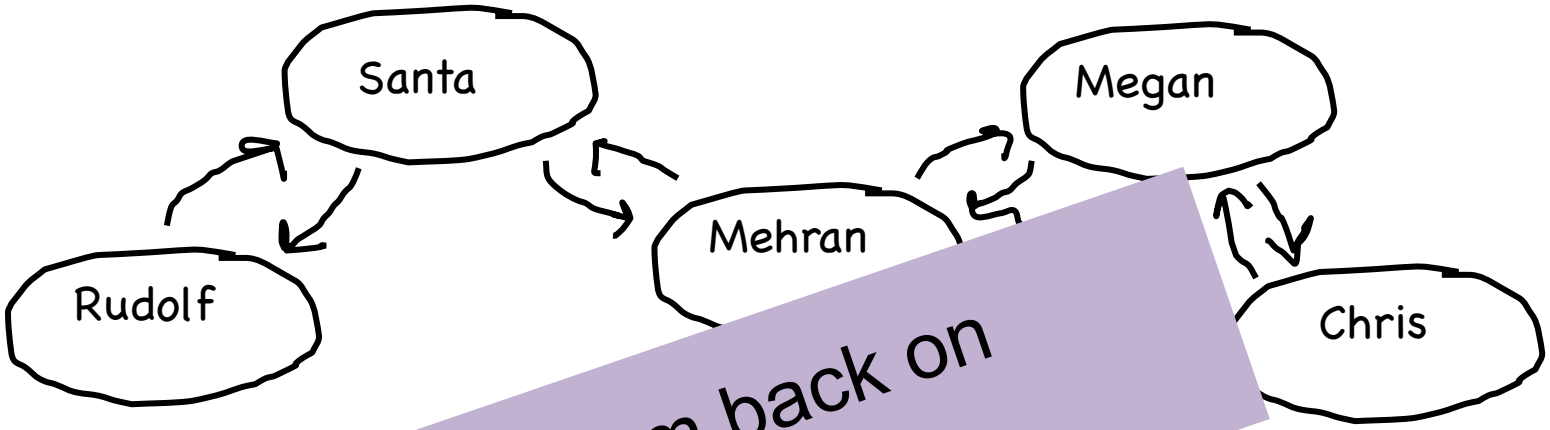
Winner - Algorithmic



Ali Malik

Thanks for playing

Midterm



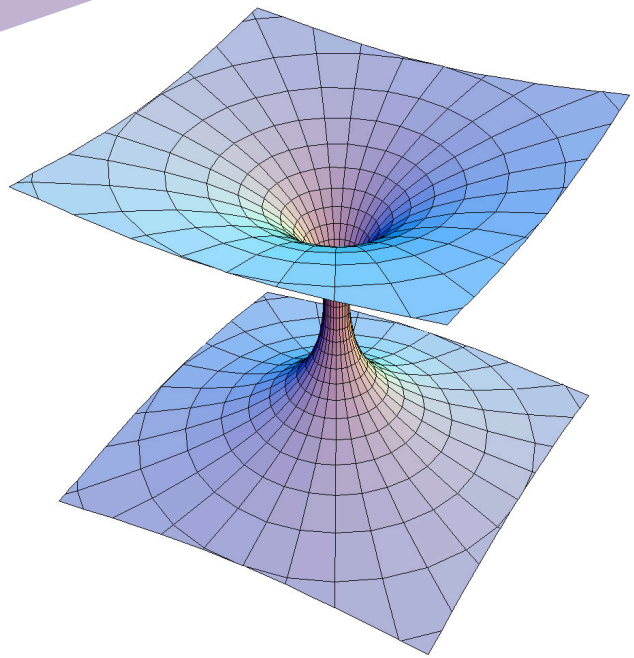
Get them back on Wednesday

PANDORA
internet radio

Tell us your favorite song and we'll create a station that explores that part of the music universe.

Enter Song

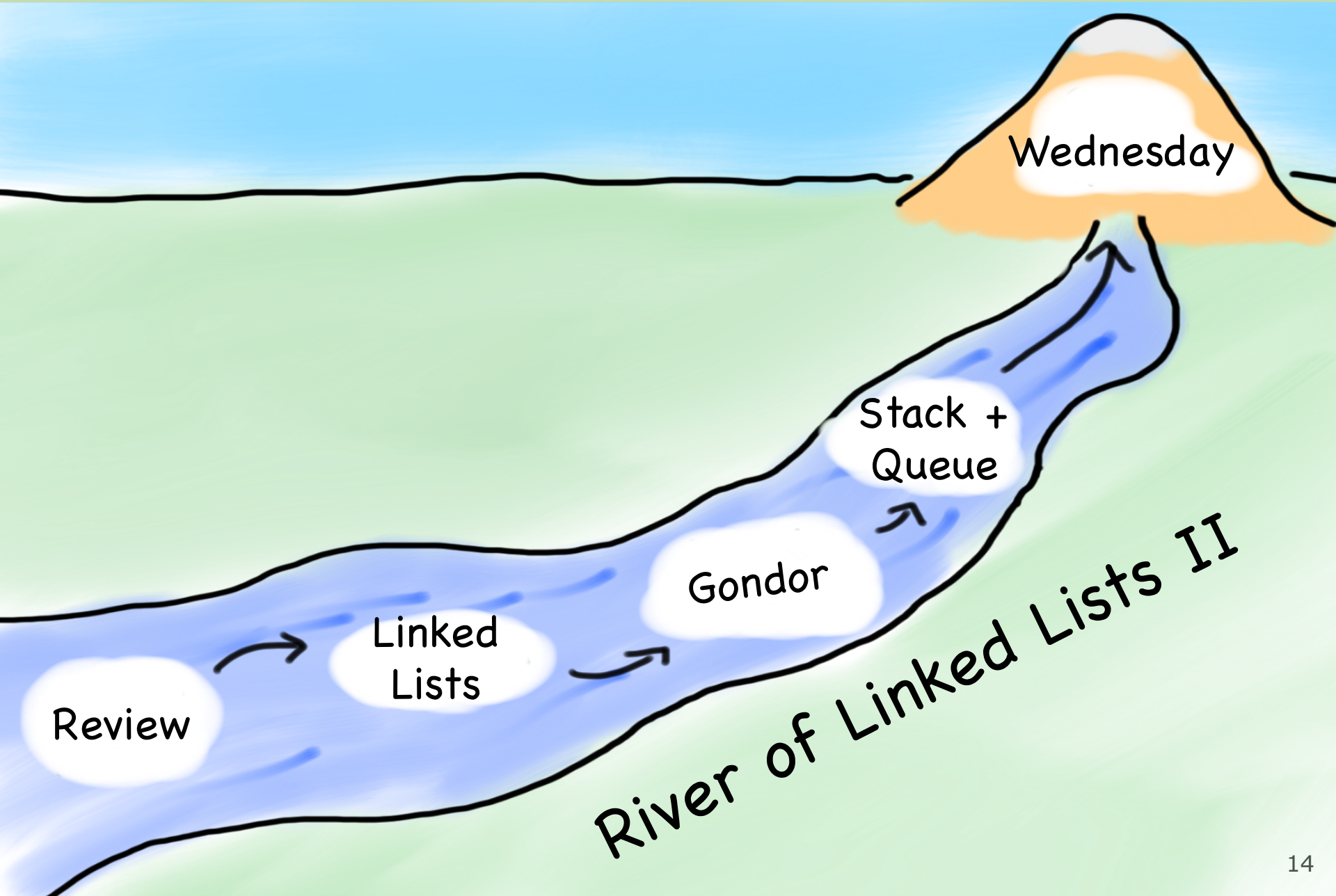
Create



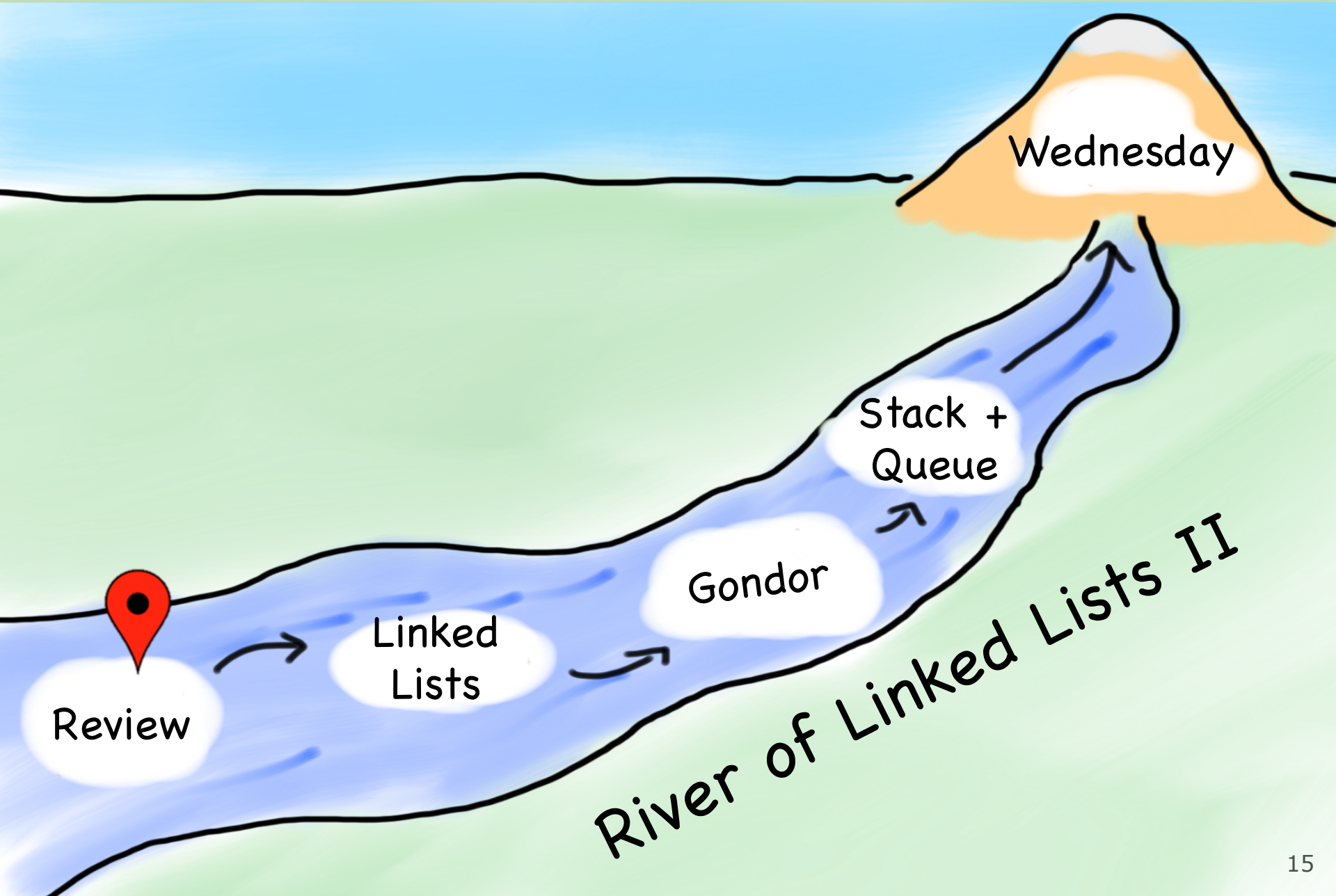
Today's Goals

1. Round out knowledge of linked lists
2. See how Stack + Queue work

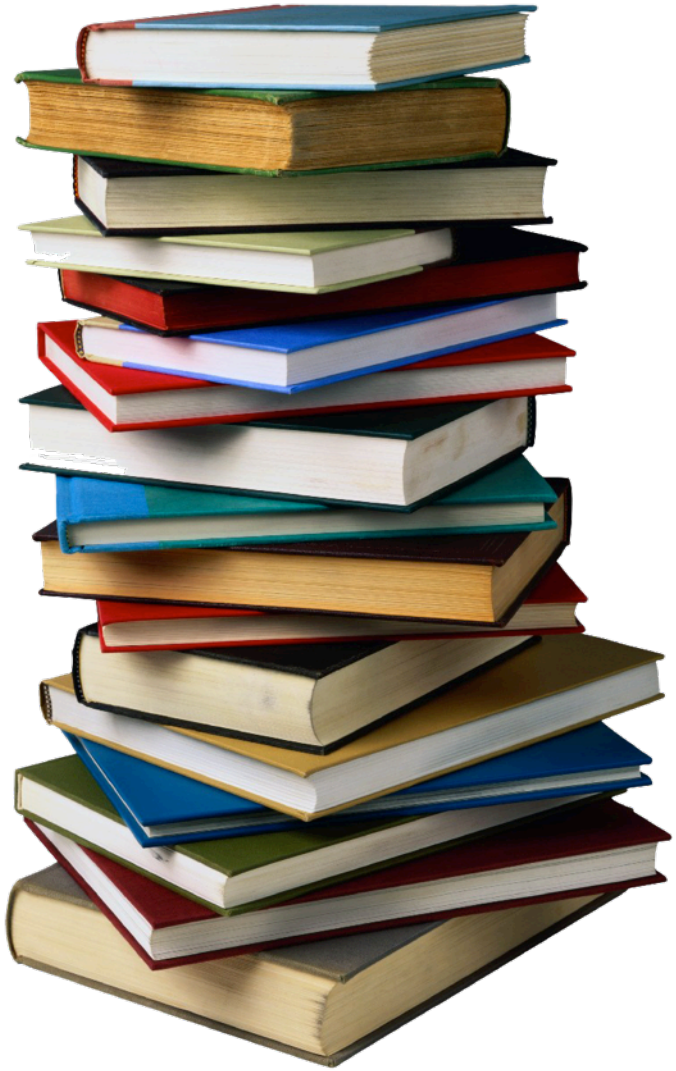
Today's Goals



Today's Goals



How is the Stack Implemented?



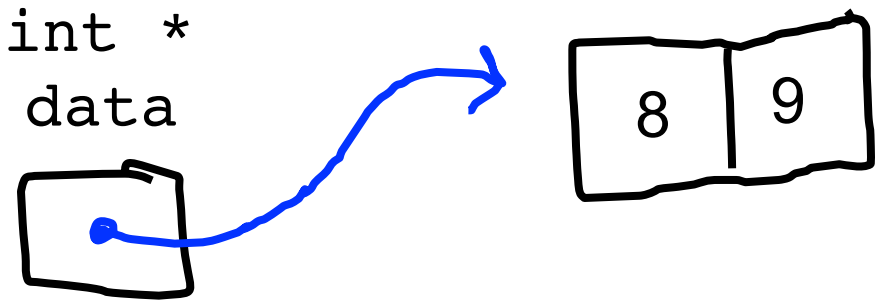
VectorInt

```
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
};
```

There’s always a better way

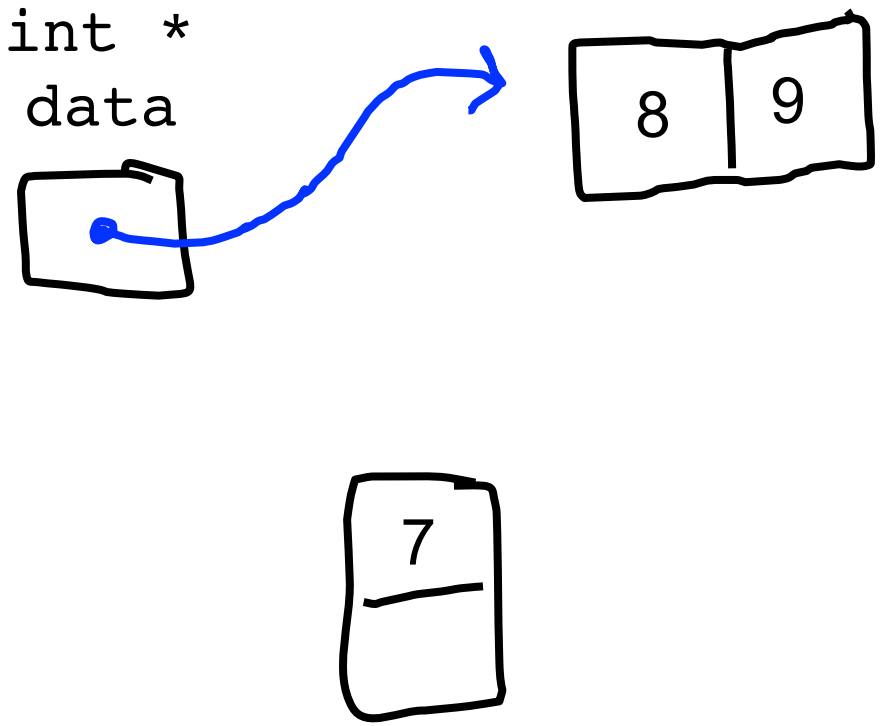
What About This?

```
push(7);
```



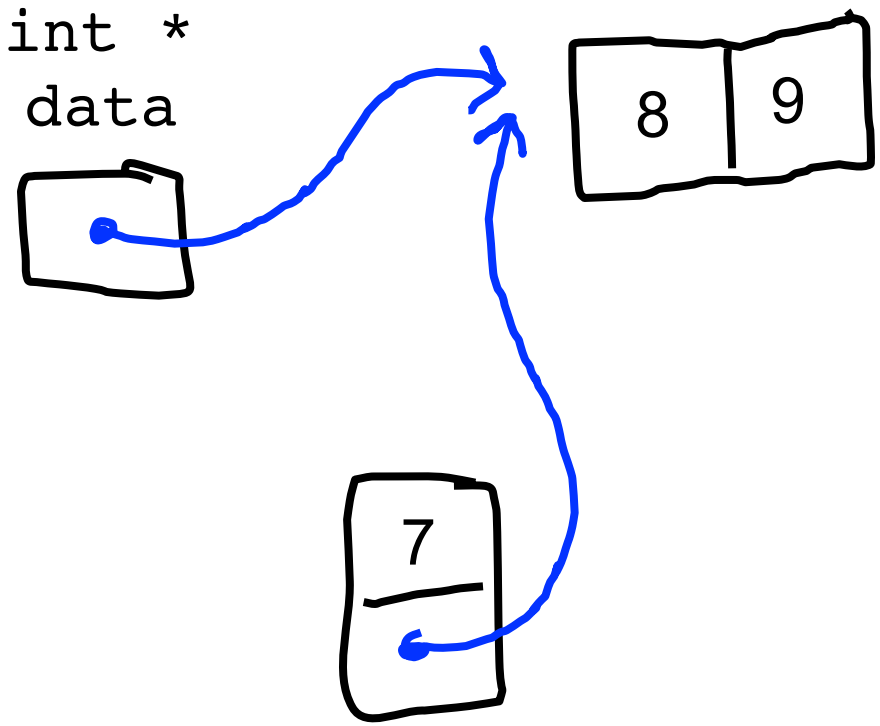
What About This?

```
push(7);
```



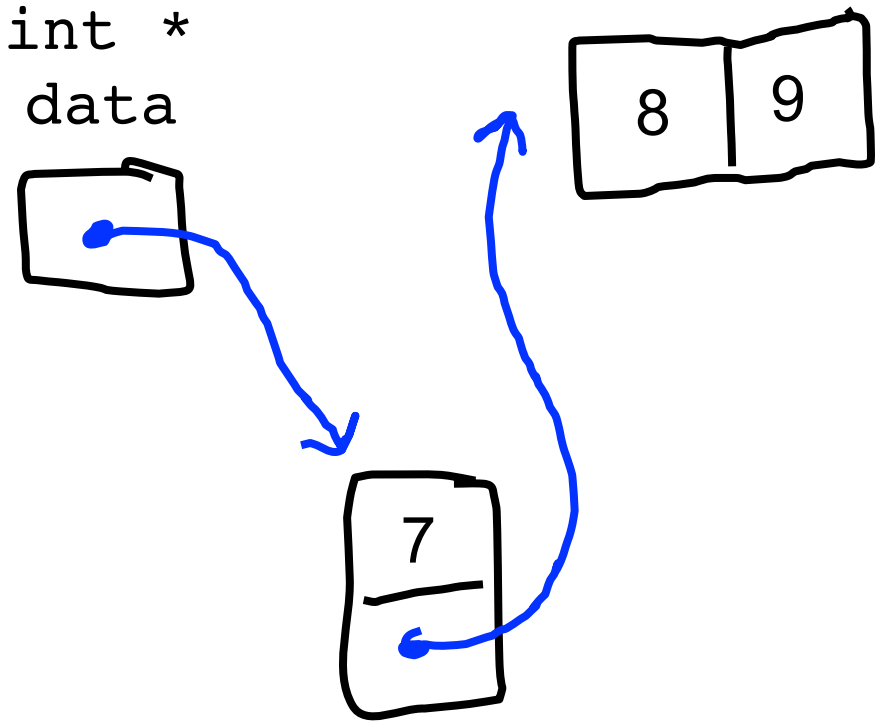
What About This?

```
push(7);
```



What About This?

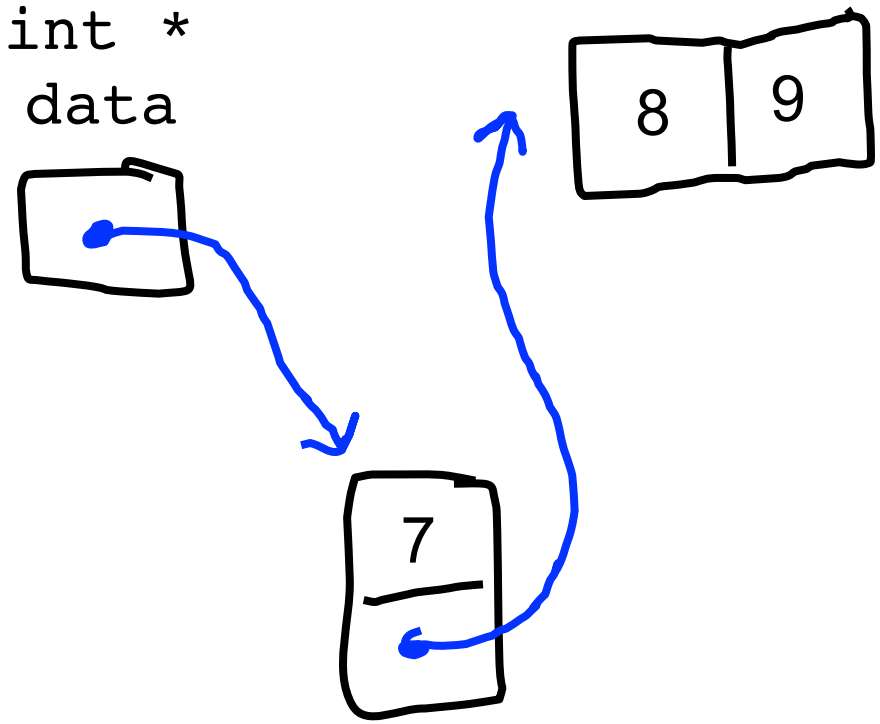
```
push(7);
```



Oh Cool

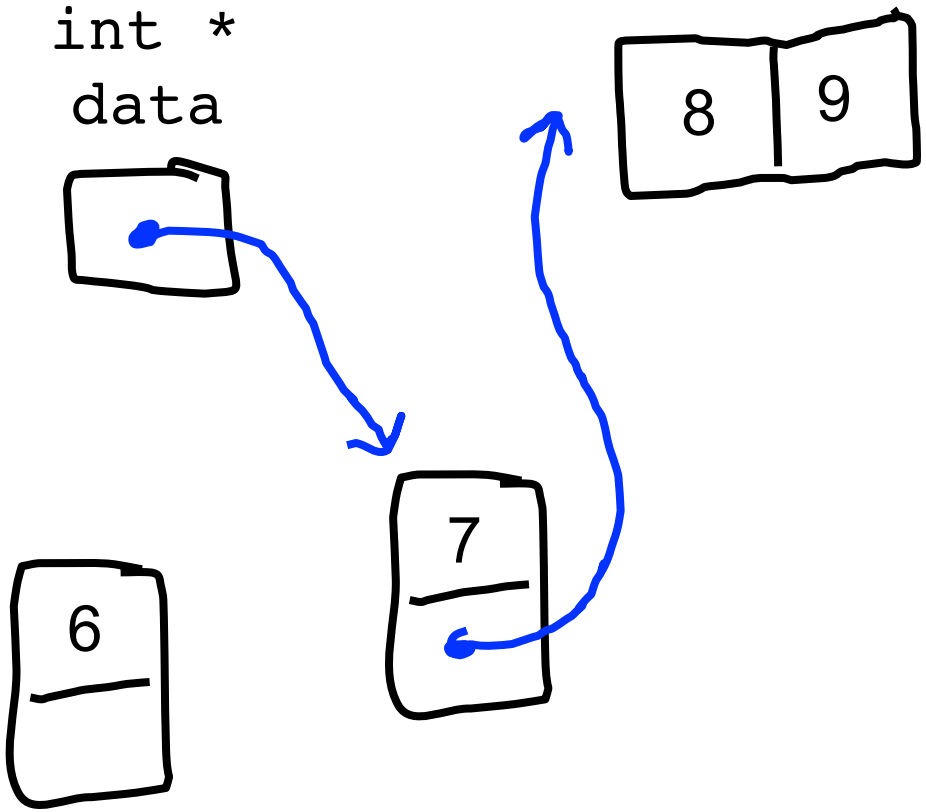
What About This?

```
push(6);
```



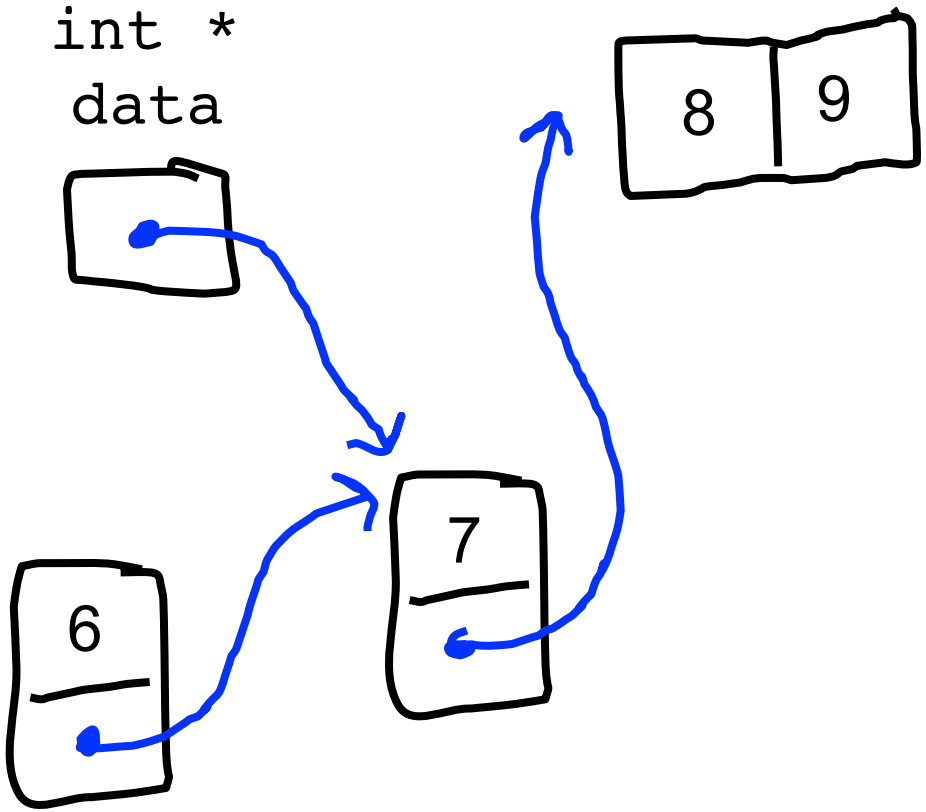
What About This?

```
push(6);
```



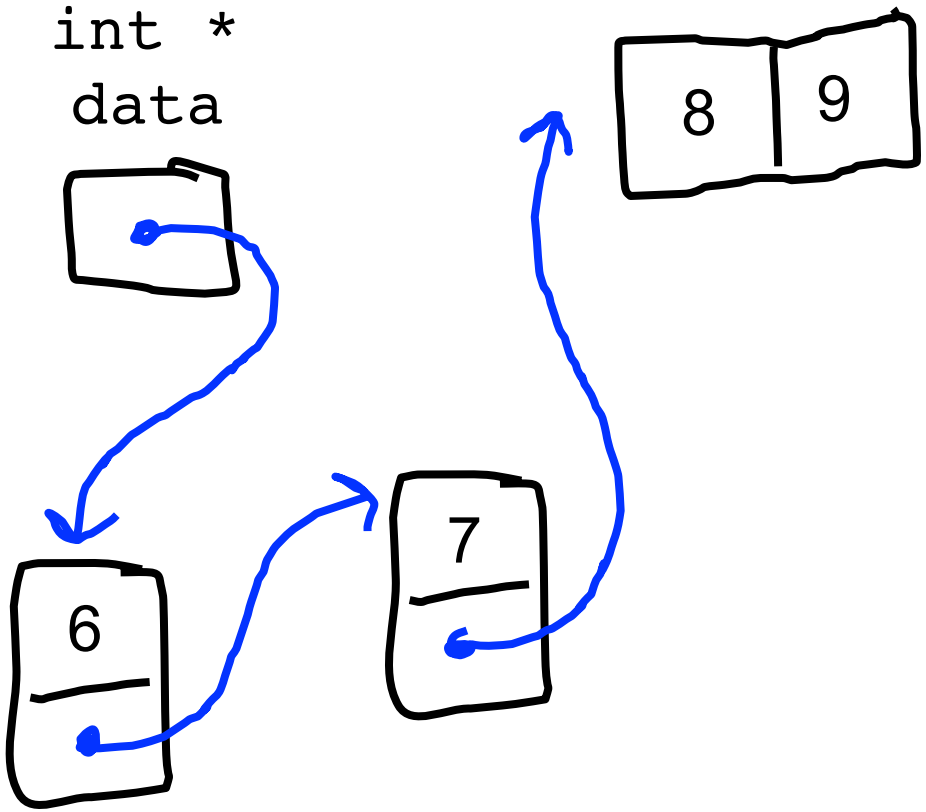
What About This?

```
push(6);
```



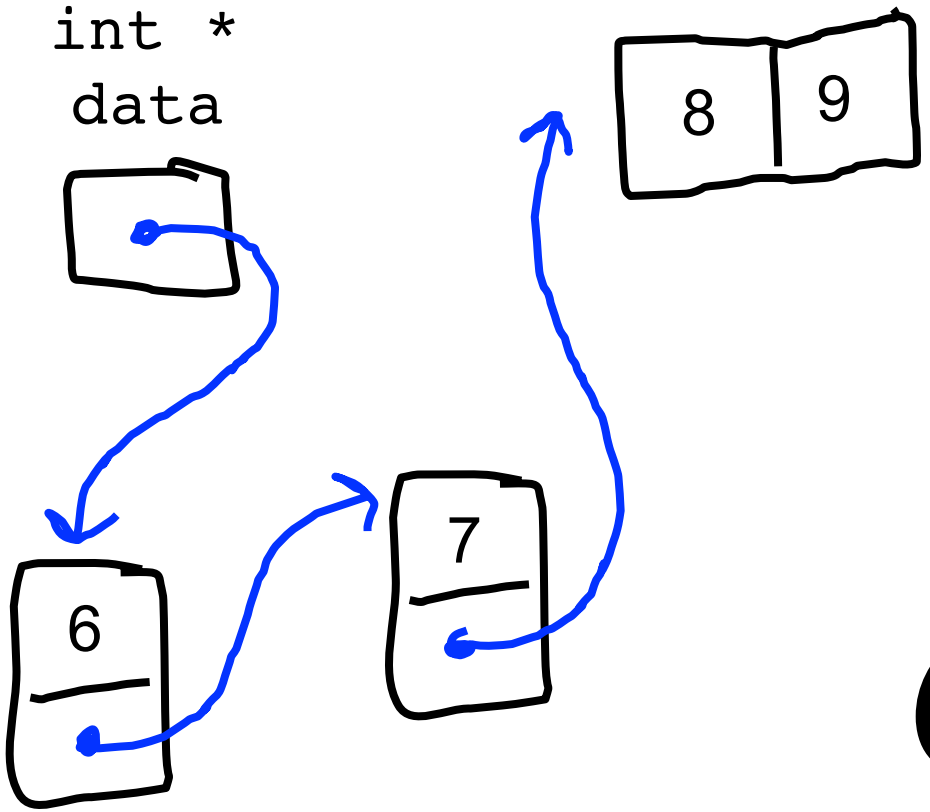
What About This?

```
push(6);
```



What About This?

```
push(6);
```

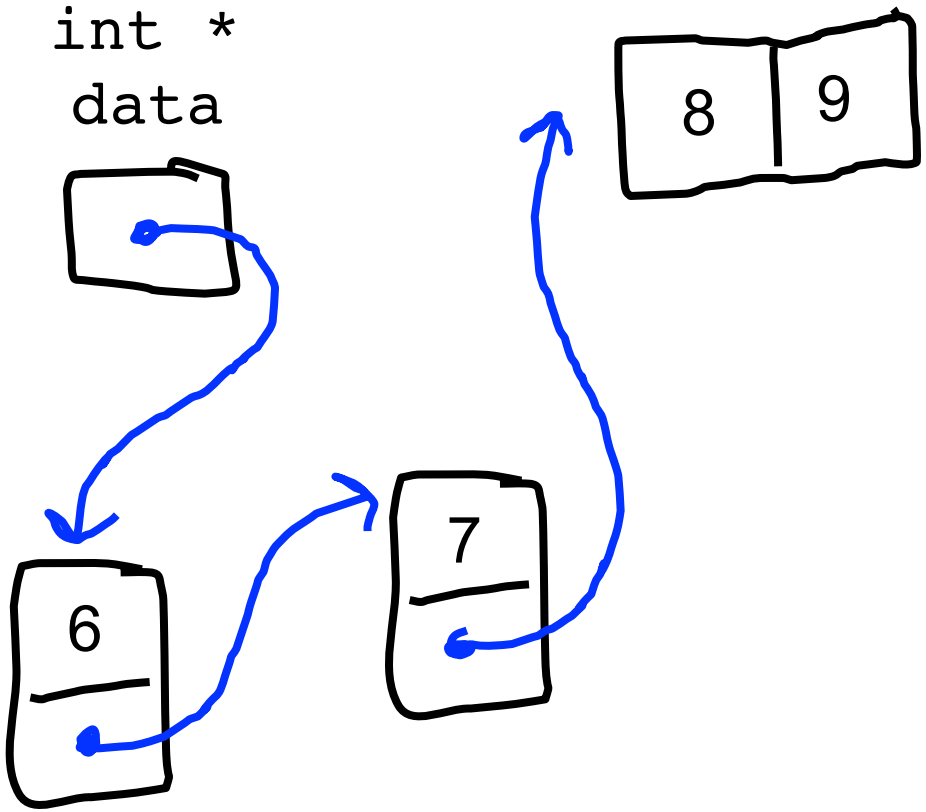


$$\mathcal{O}(1)$$

And Pop?

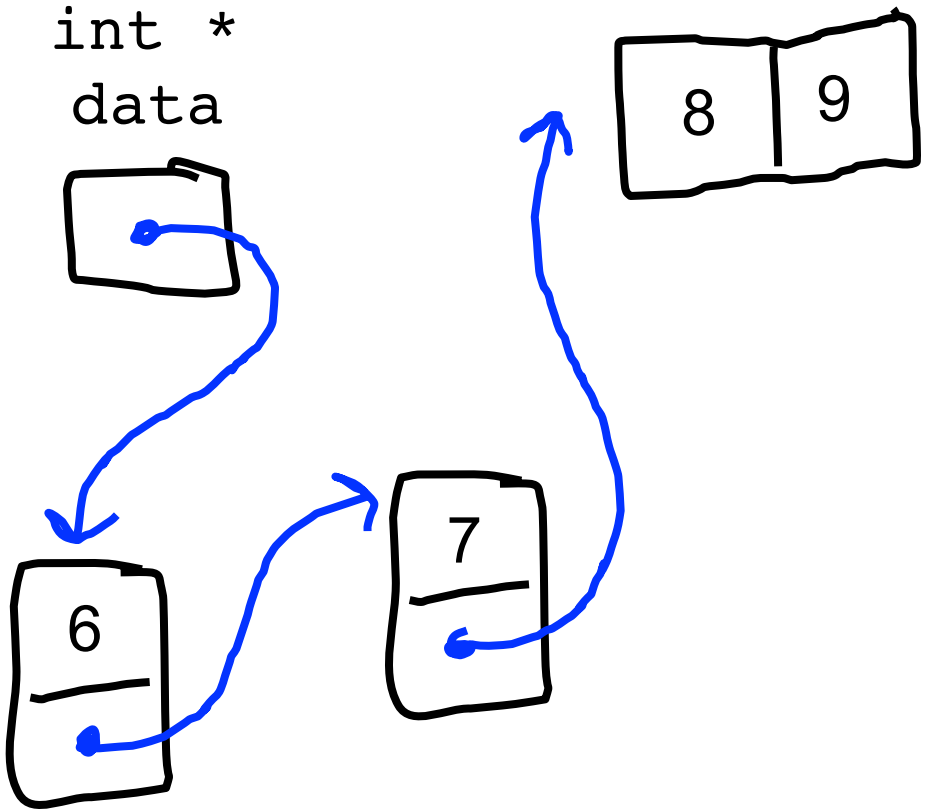
What About This?

```
pop();
```



What About This?

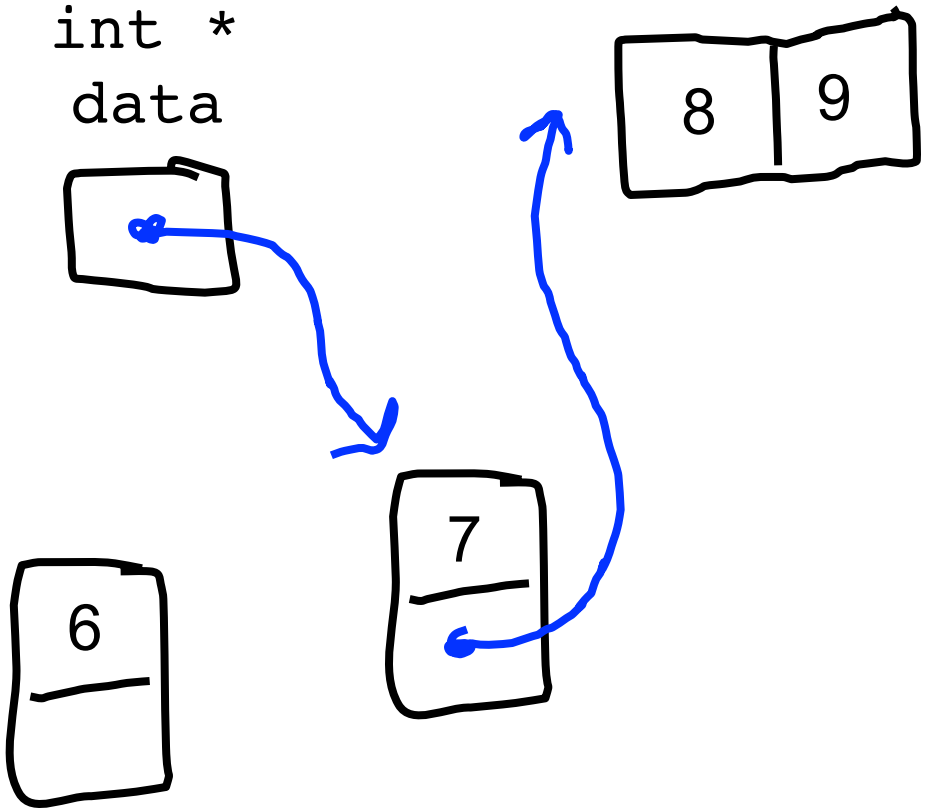
```
pop();
```




```
int return
```

```
6
```

What About This?



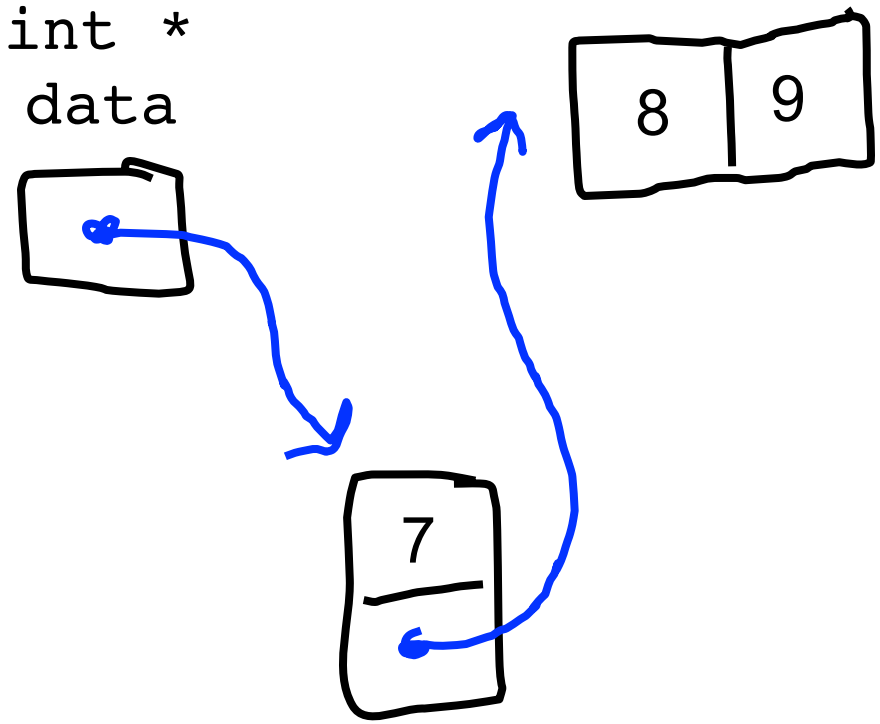

```
pop();
```

```
int return
```

6

What About This?

```
pop();
```

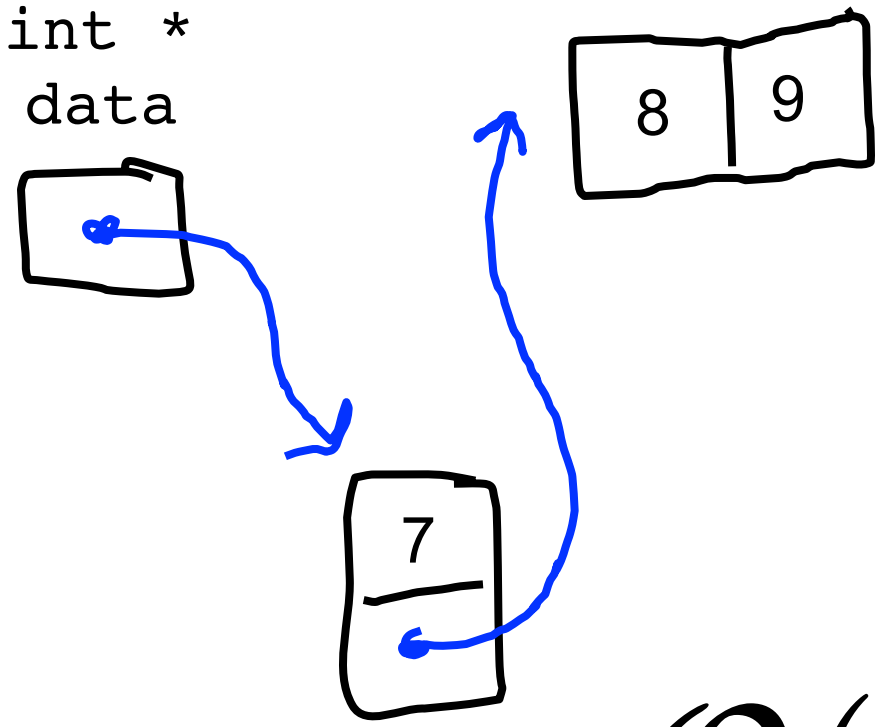



```
int return
```

```
6
```

What About This?

```
pop();
```



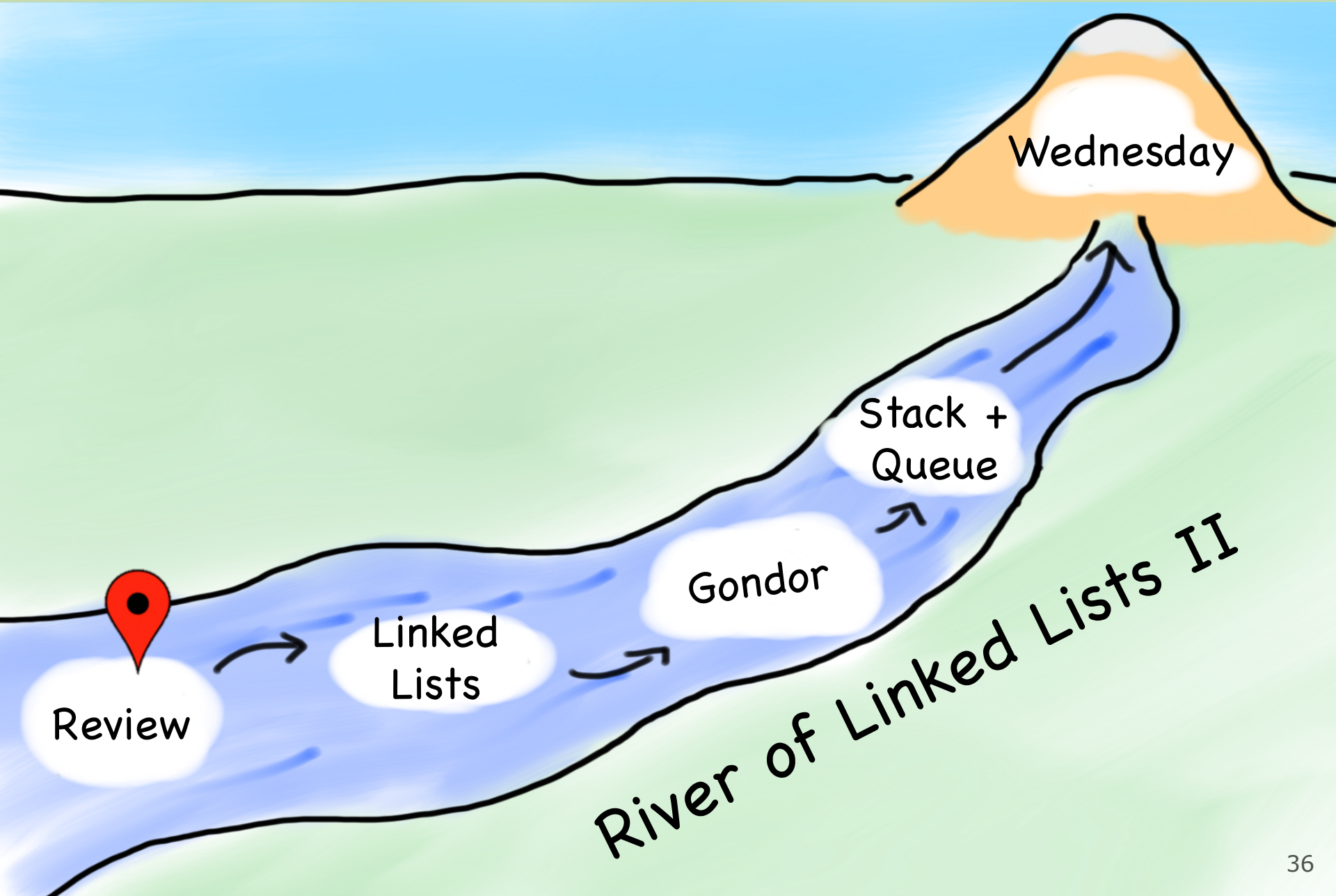

```
int return
```

6

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

Linked Lists!

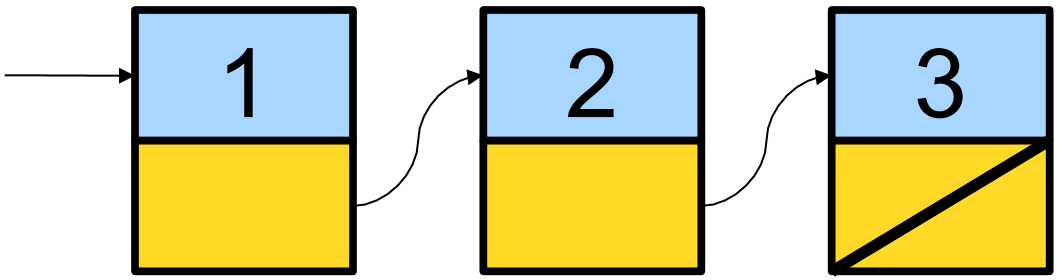
Today's Goals



Today's Goals

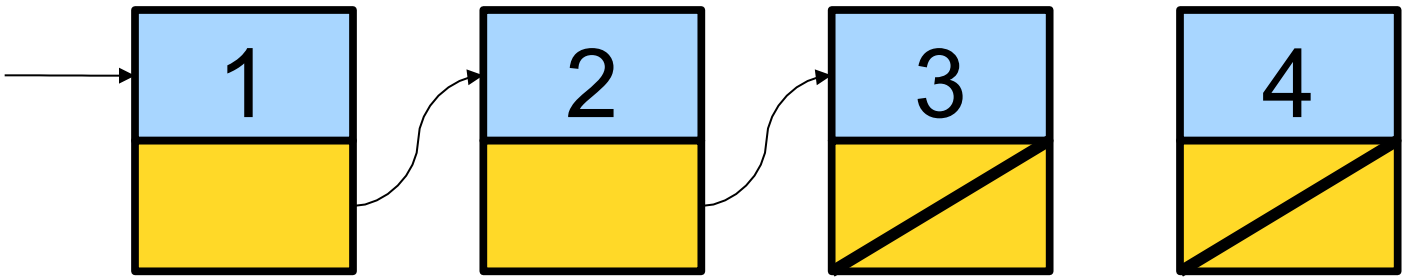
Linked Lists

- 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.



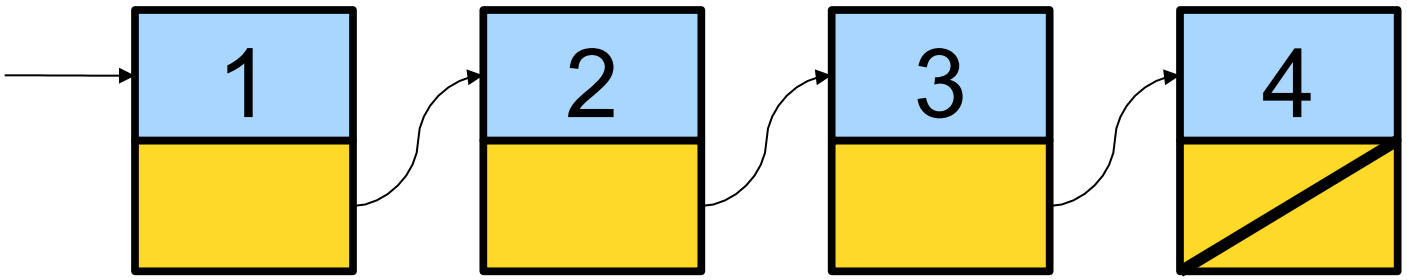
Linked Lists

- 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.



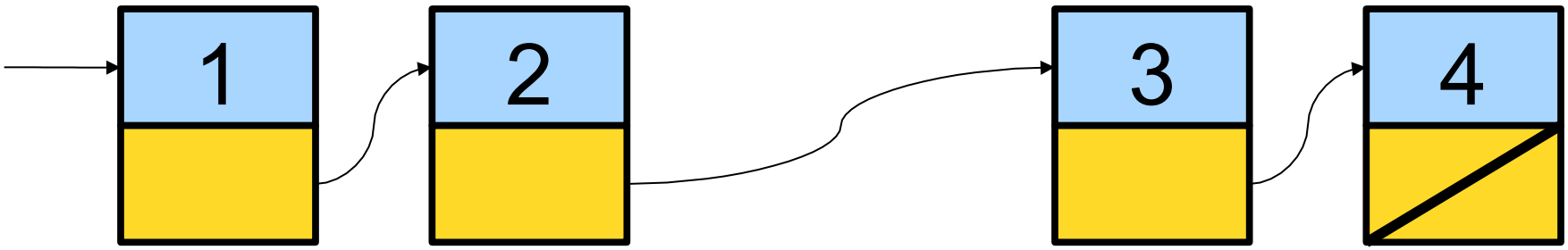
Linked Lists

- 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.



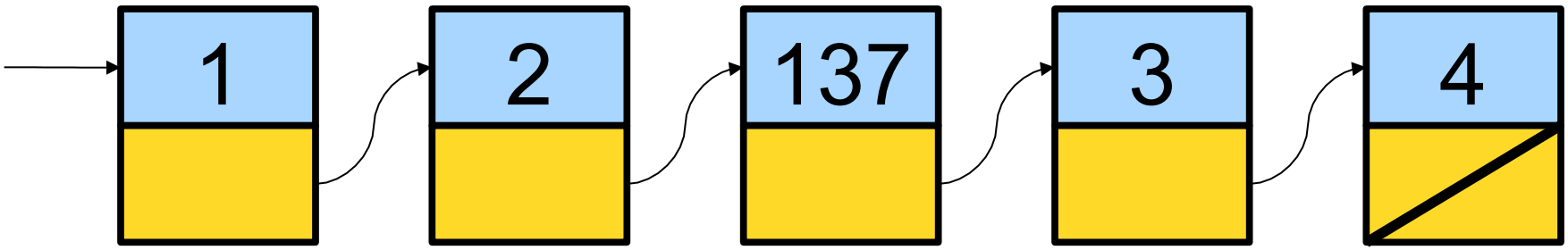
Linked Lists

- 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.



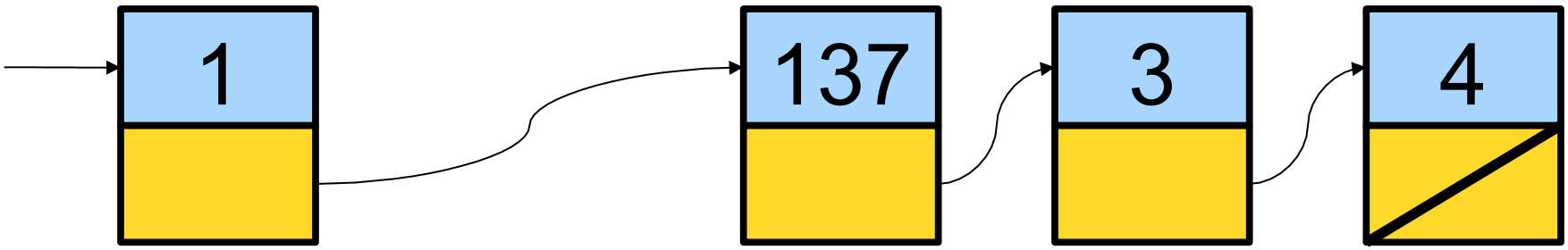
Linked Lists

- 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

As a reminder we can use the **`new`** keyword 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

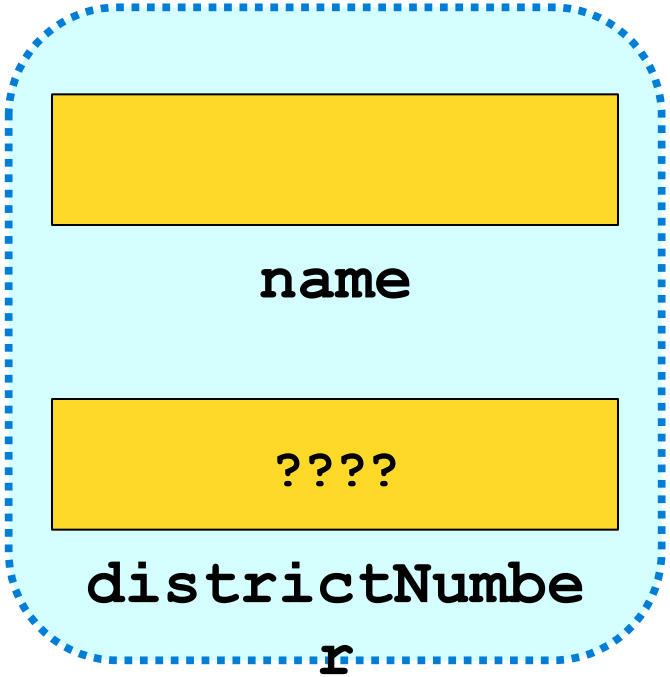
```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```

t

Structs

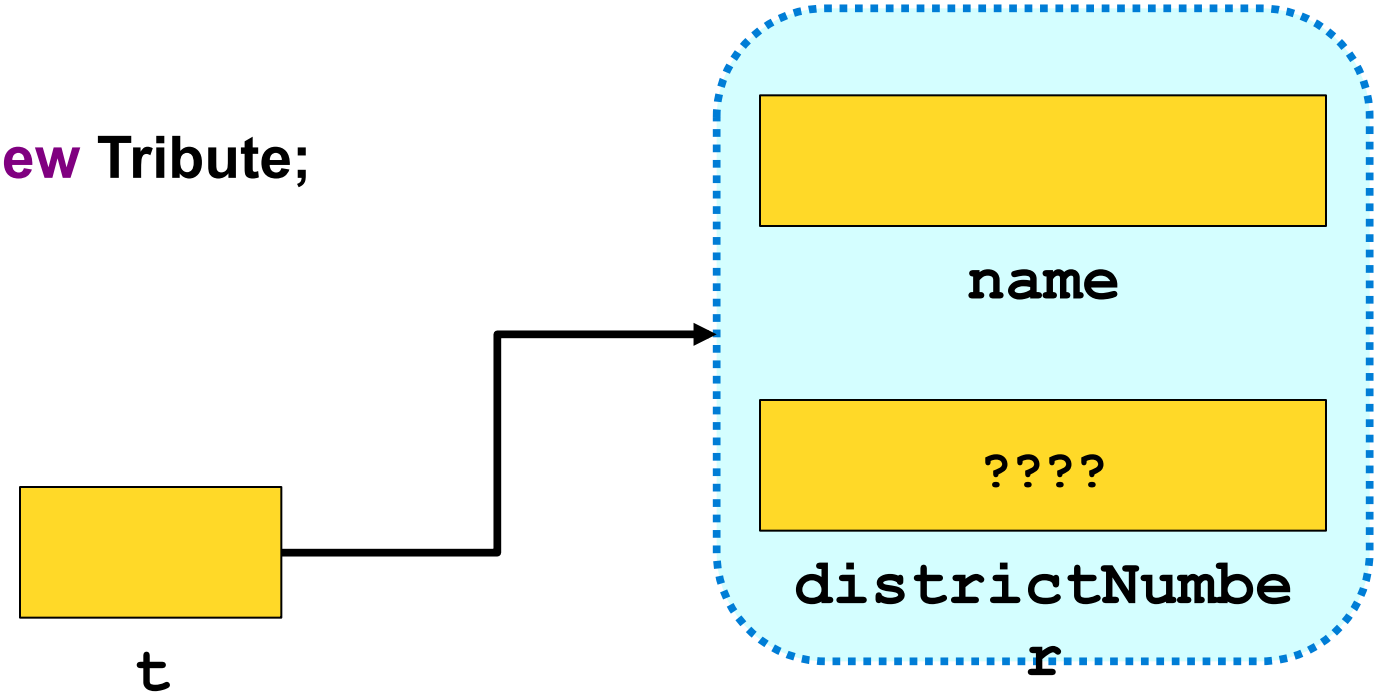
```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```

t



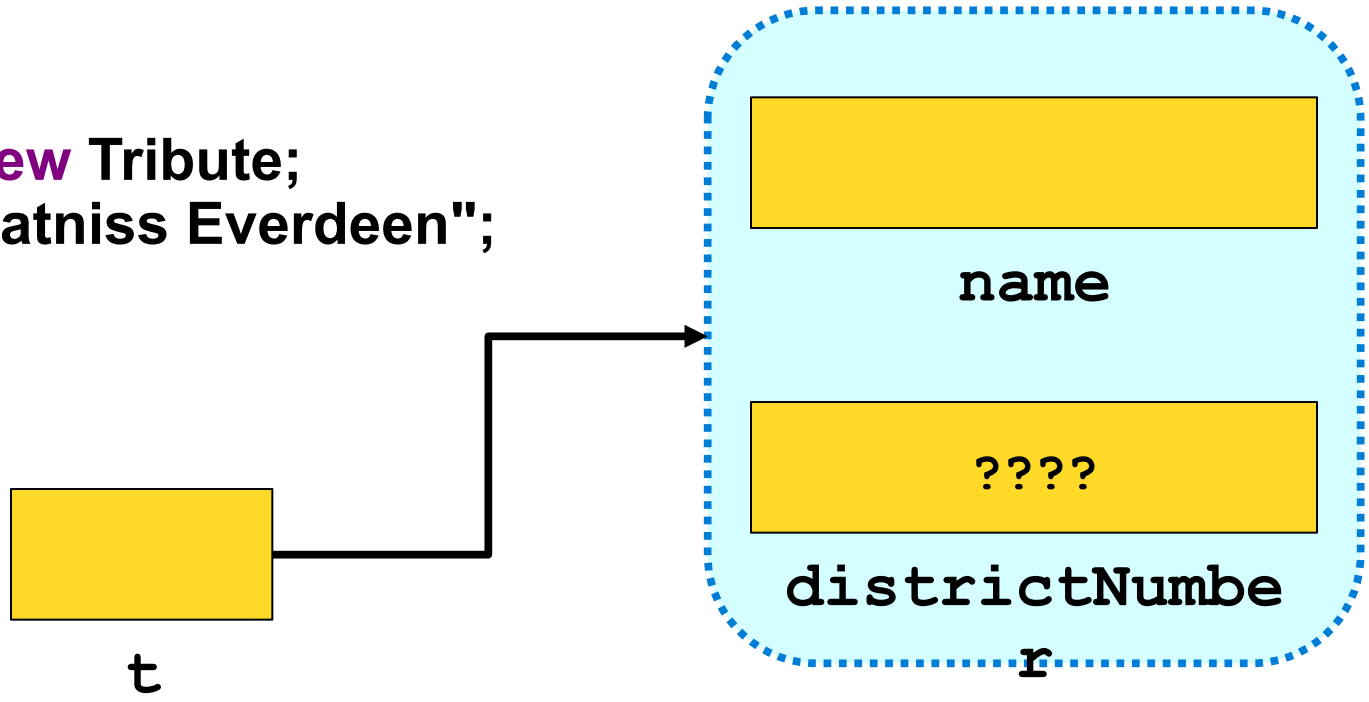
Structs

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

Tribute* t = new Tribute;
```



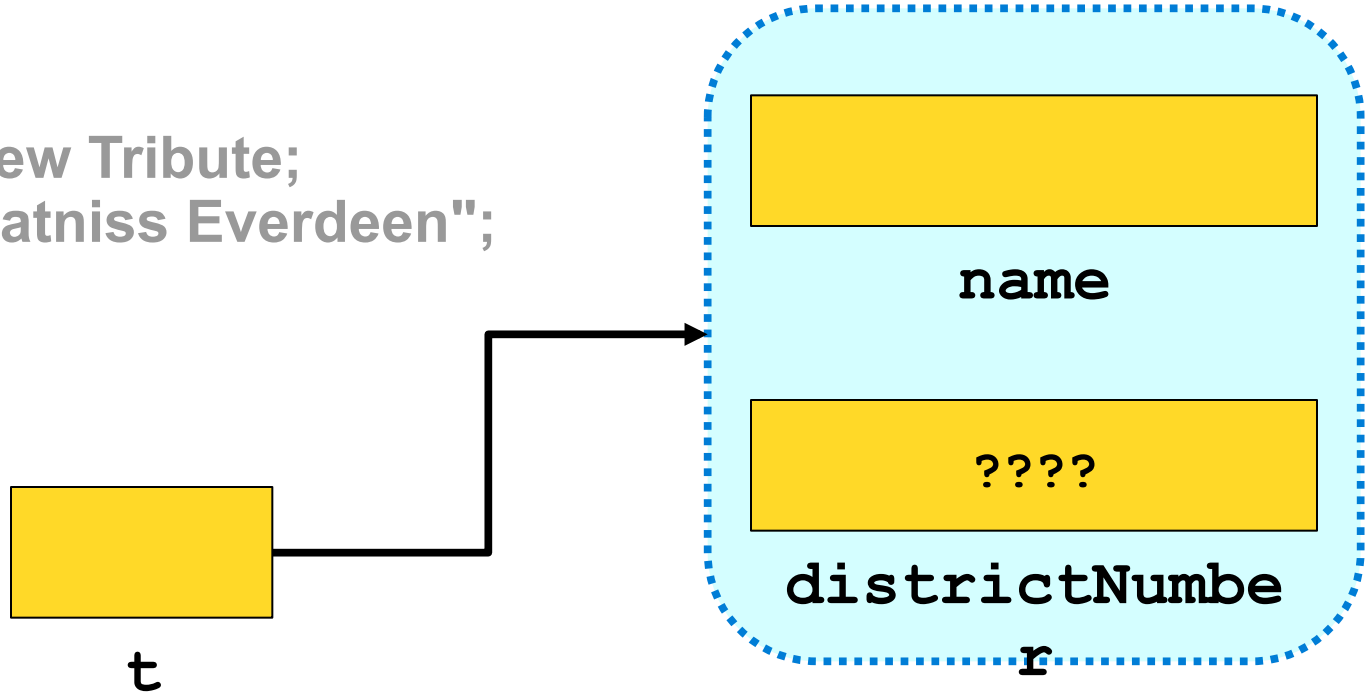
Structs

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

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



Structs

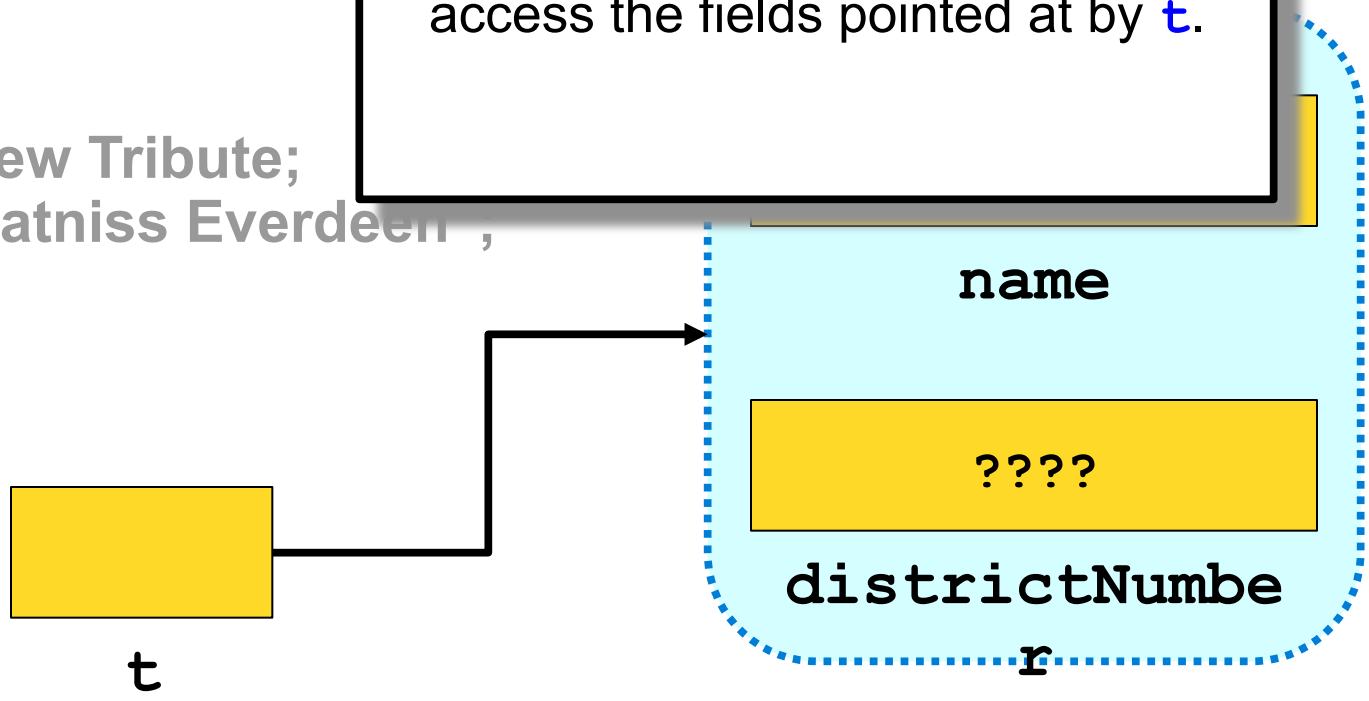
```
struct Tribute {
    string name;
    int districtNumber;
};

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



Structs

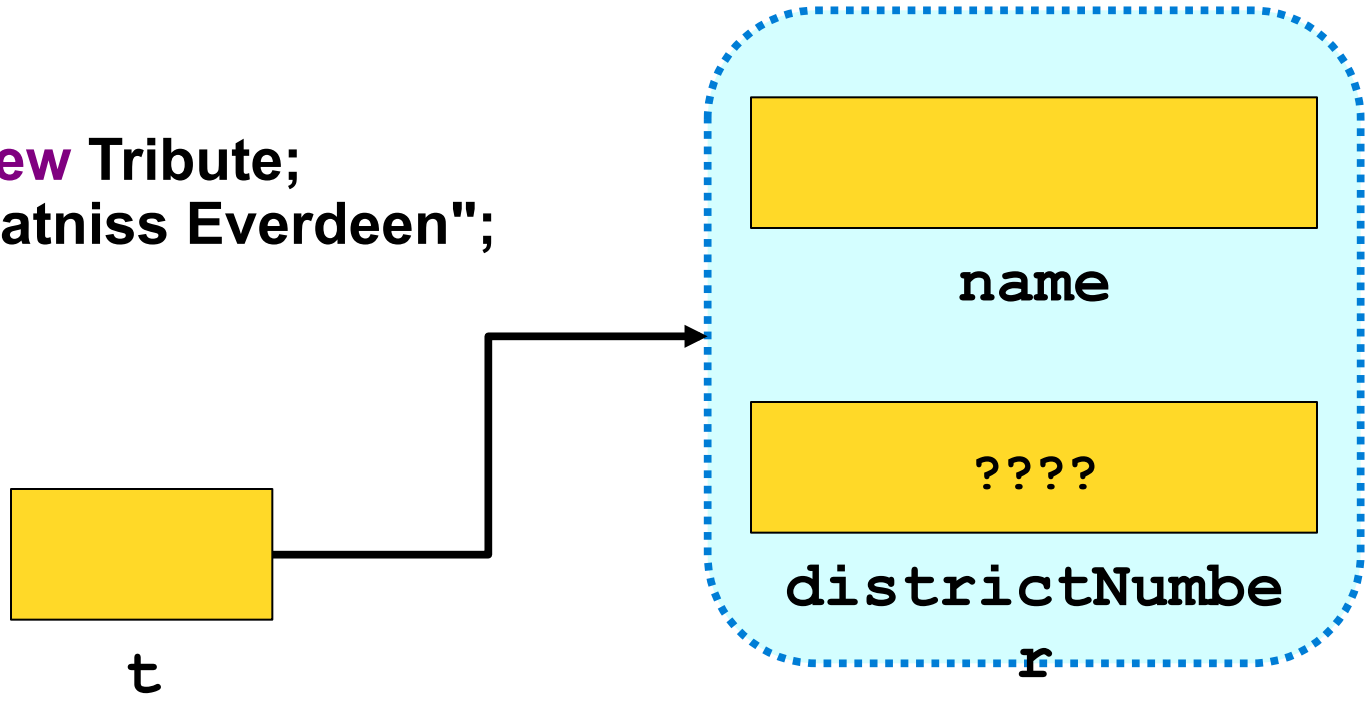
```
struct Tribute {
    string name;
    int districtNumber;
};

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`.



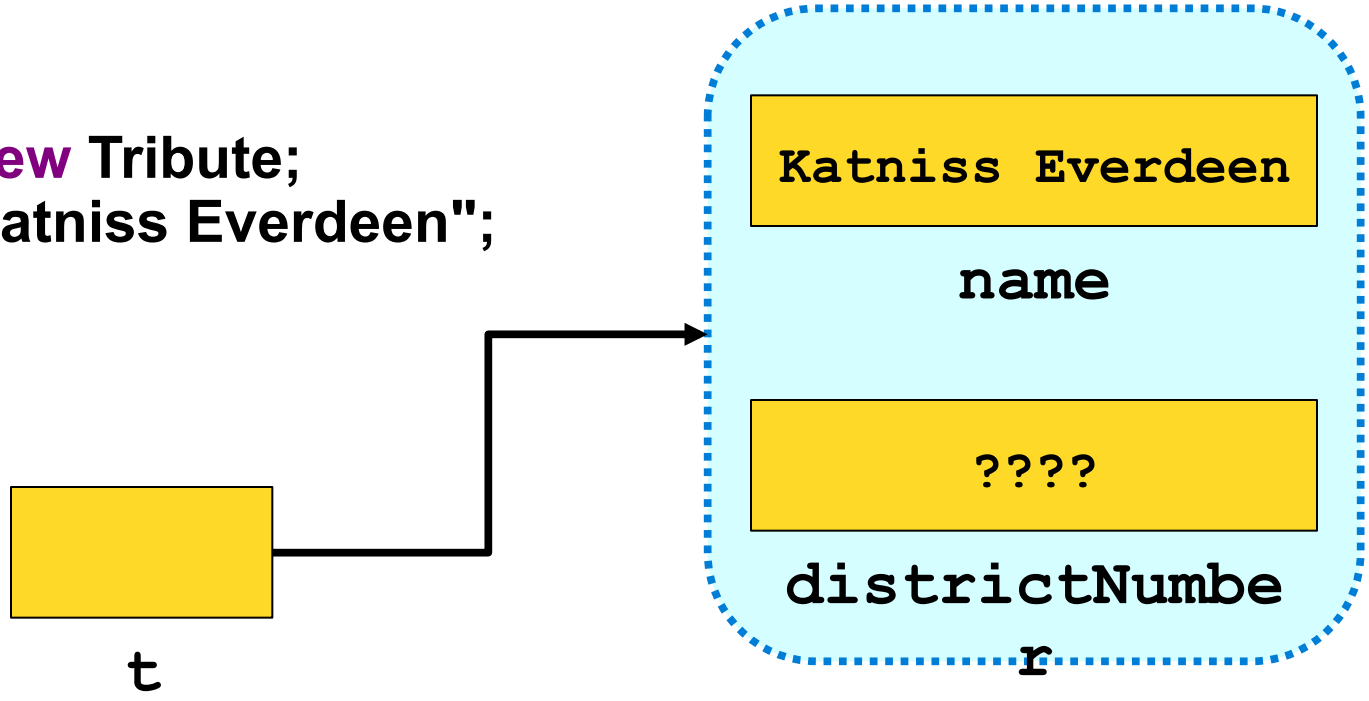
Structs

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

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



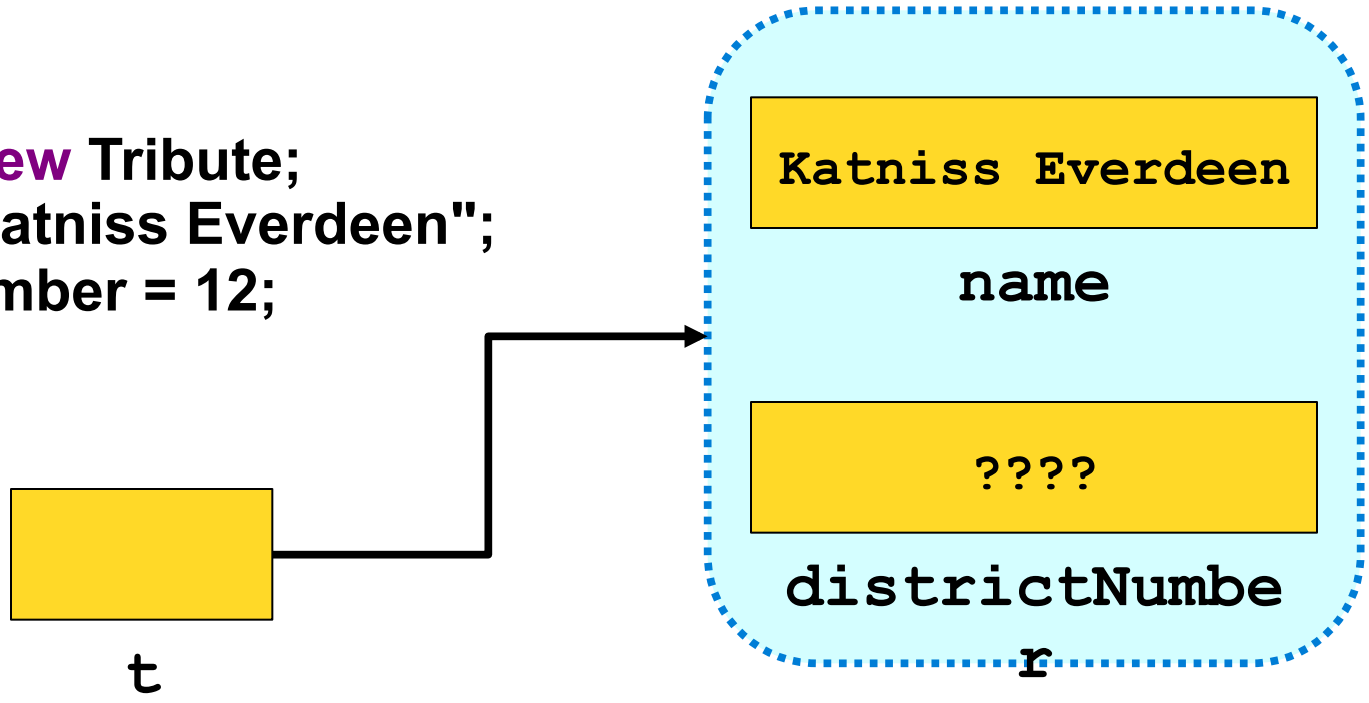
Structs

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

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



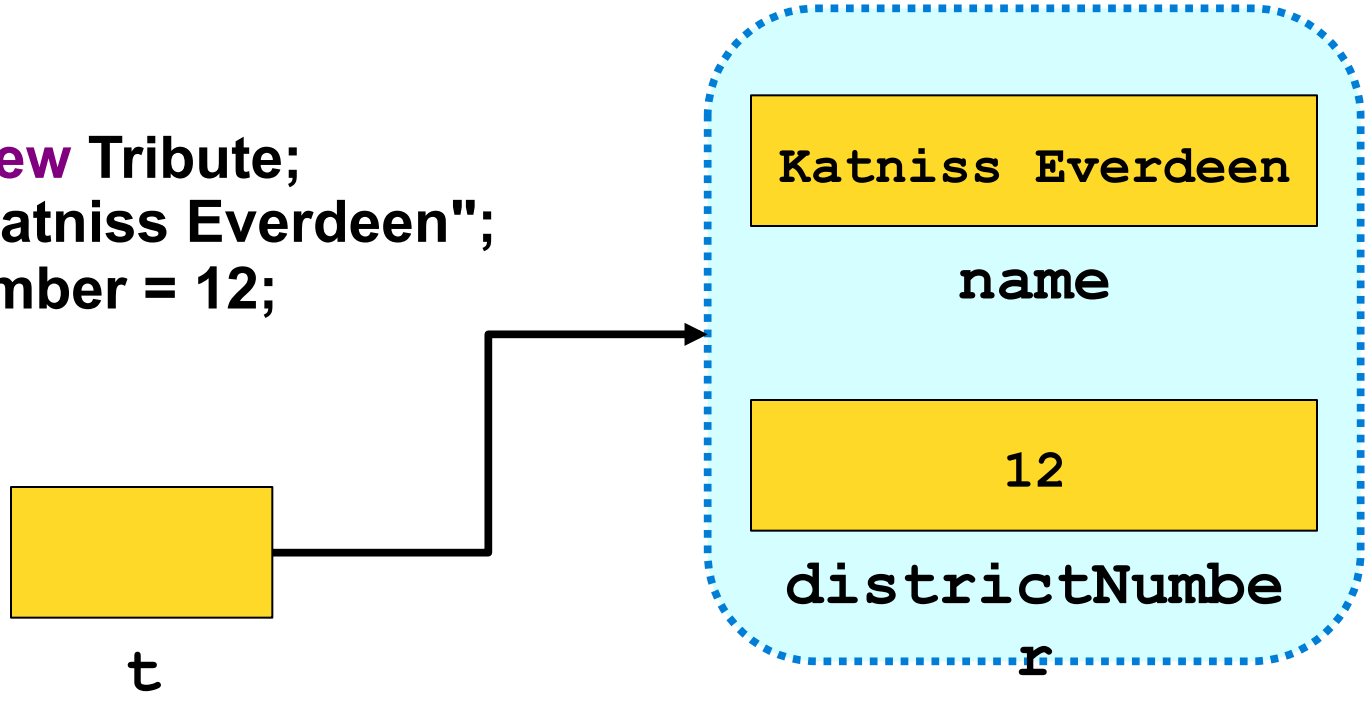
Structs

```
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*;

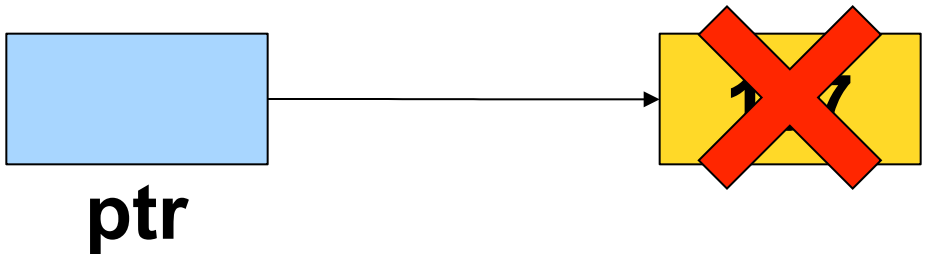
- This destroys the object pointed at by the given pointer, not the pointer itself.

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 **nodes**.
- Each cell contains two pieces of information:
  - Some piece of data that is stored in the sequence, and
  - 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

- 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 Lists

- 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 Lists

- 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

First Rule of Linked List Club

Draw a picture

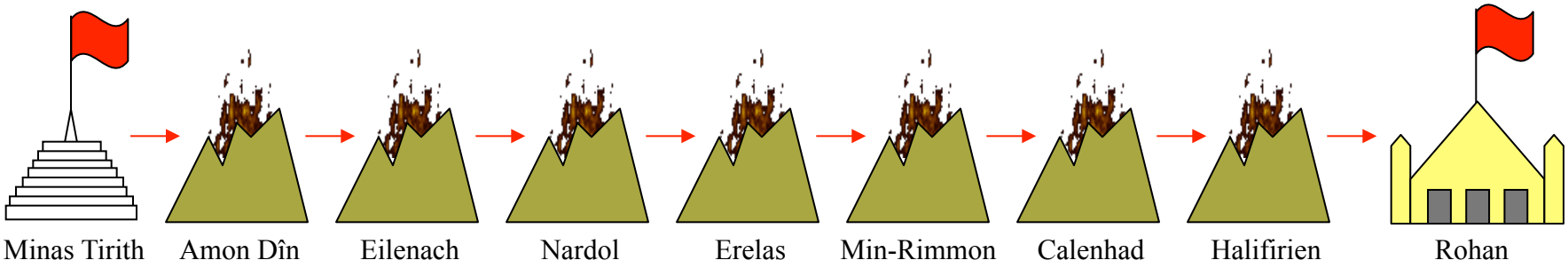
Today's Goals

Today's Goals

Beacons of Gondor

> *For answer Gandalf cried aloud to his horse. "On, Shadowfax! We must hasten. Time is short. See! The beacons of Gondor are alight, calling for aid. War is kindled. See, there is the fire on Amon Dîn, and flame on Eilenach; and there they go speeding west: Nardol, Erelas, Min-Rimmon, Calenhad, and the Halifirien on the borders of Rohan."*
>
> —J. R. R. Tolkien, *The Return of the King,* 1955

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 Beacons of Gondor

Volunteer

Beacons of Gondor

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

Beacons of Gondor

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

Beacons of Gondor

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

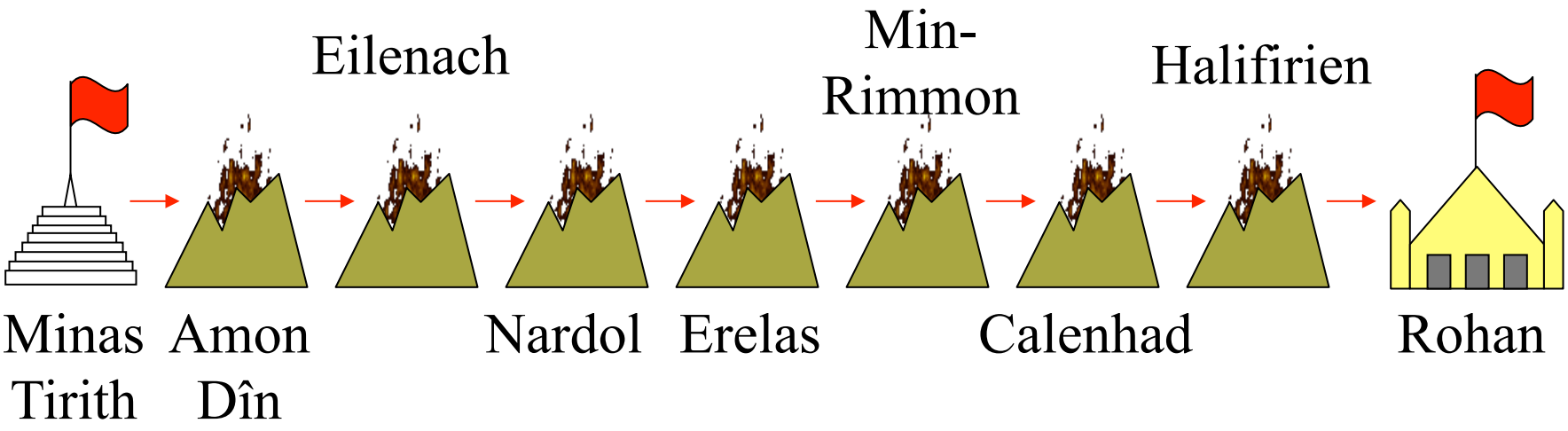
Beacons of Gondor

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

Beacons of Gondor

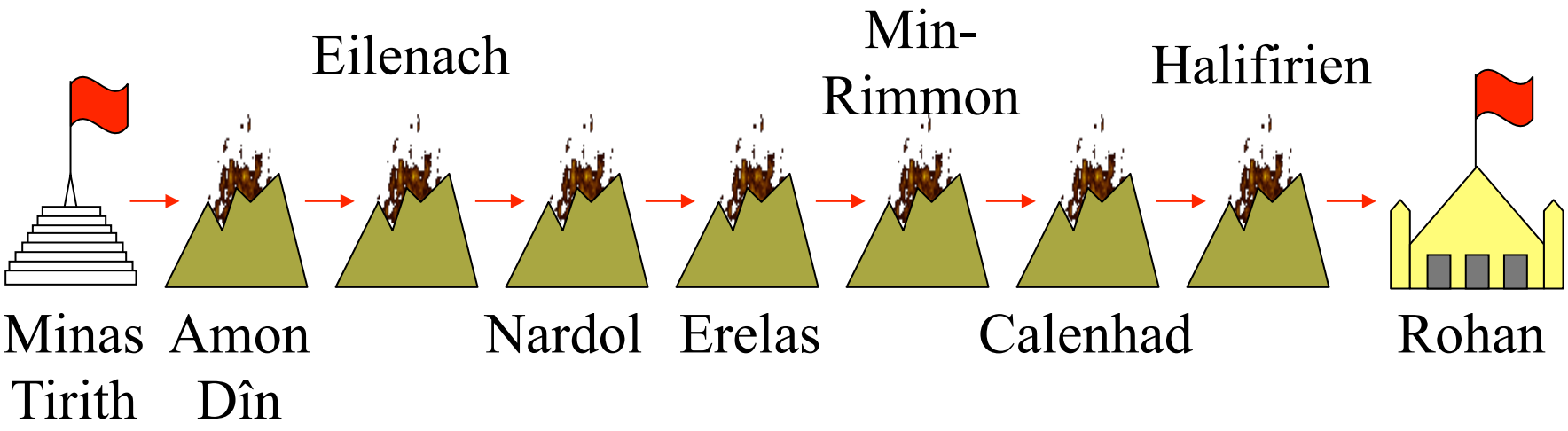
```
// add the first tower
Tower * head = new Tower;
head->name = "Rohan";
head->link = NULL;
```

Beacons of Gondor



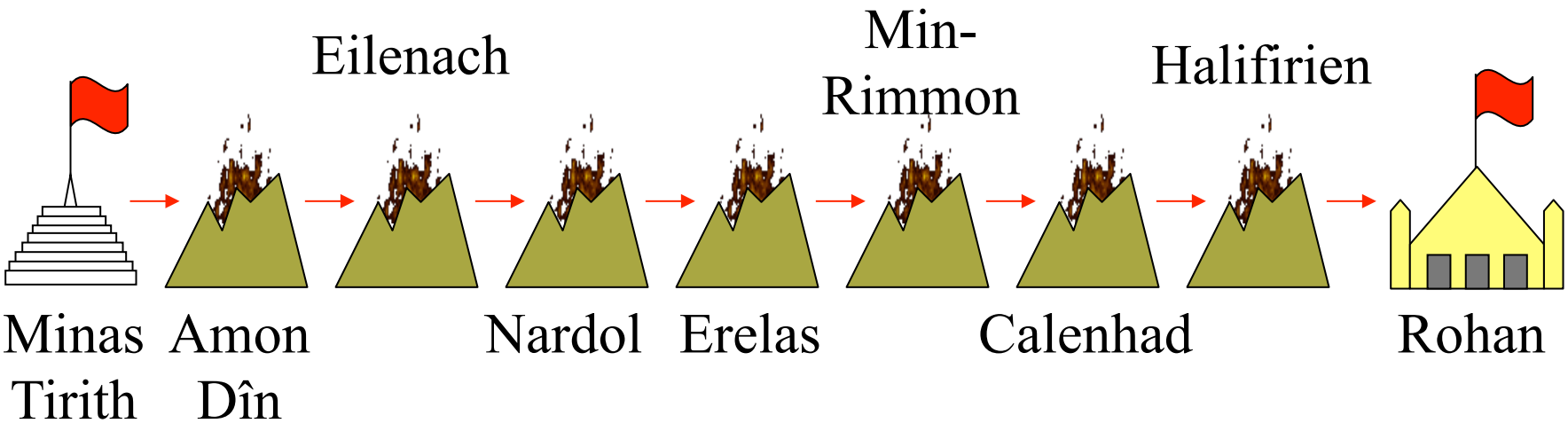

```
// add a new tower
Tower * temp = new Tower;
temp->name = towerName;
temp->link = head;
head = temp;
```

Beacons of Gondor



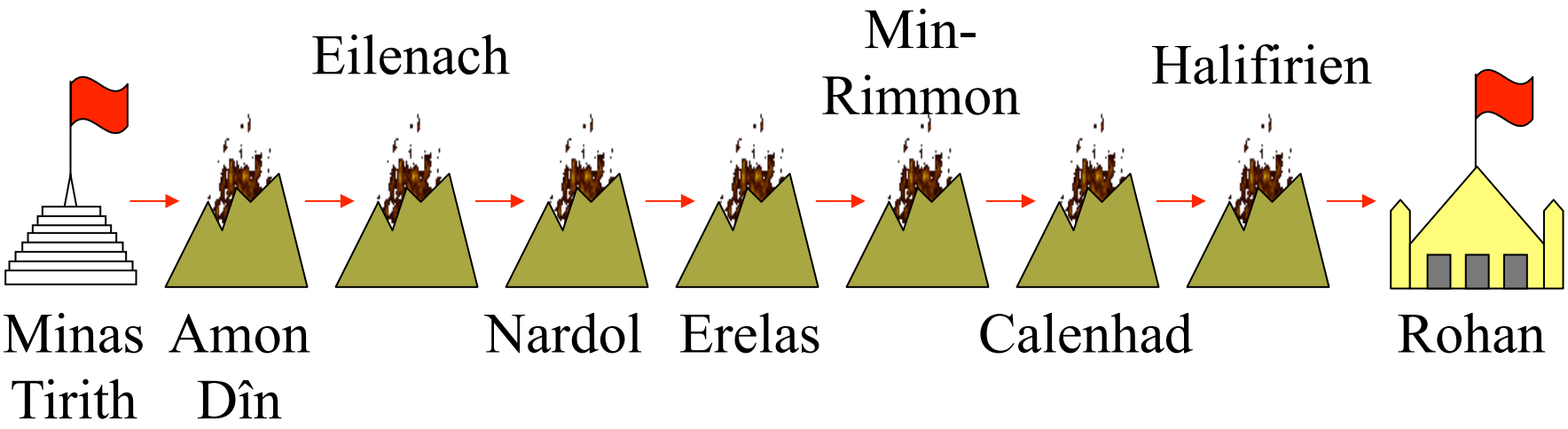

```
// add a new tower
Tower * temp = new Tower;
temp->name = towerName;
temp->link = head;
head = temp;
```

Beacons of Gondor



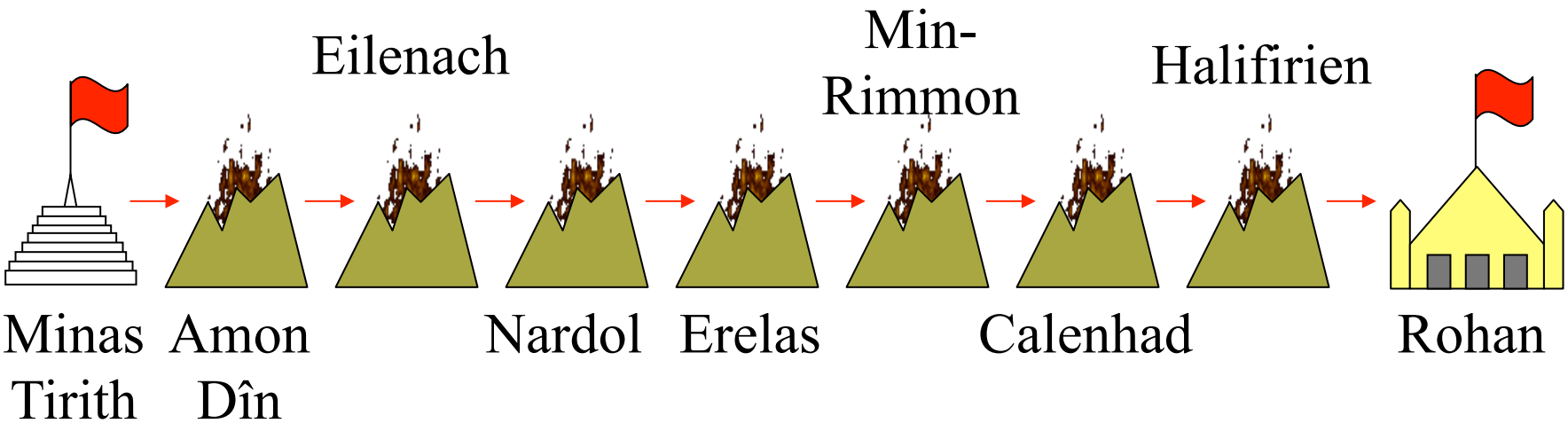

```
// add a new tower
Tower * temp = new Tower;
temp->name = towerName;
temp->link = head;
head = temp;
```

Beacons of Gondor




```
// add a new tower
Tower * temp = new Tower;
temp->name = towerName;
temp->link = head;
head = temp;
```

Beacons of Gondor



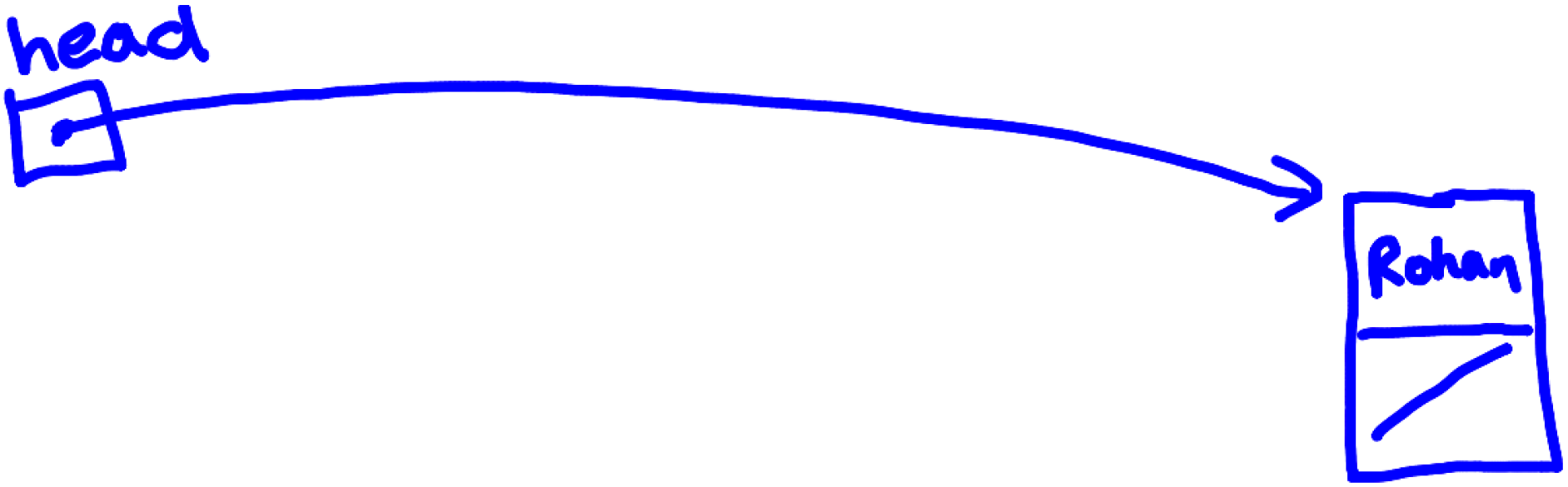

```
// add a new tower
Tower * temp = new Tower;
temp->name = towerName;
temp->link = head;
head = temp;
```

Light the Fire

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

```
signal(head);
```

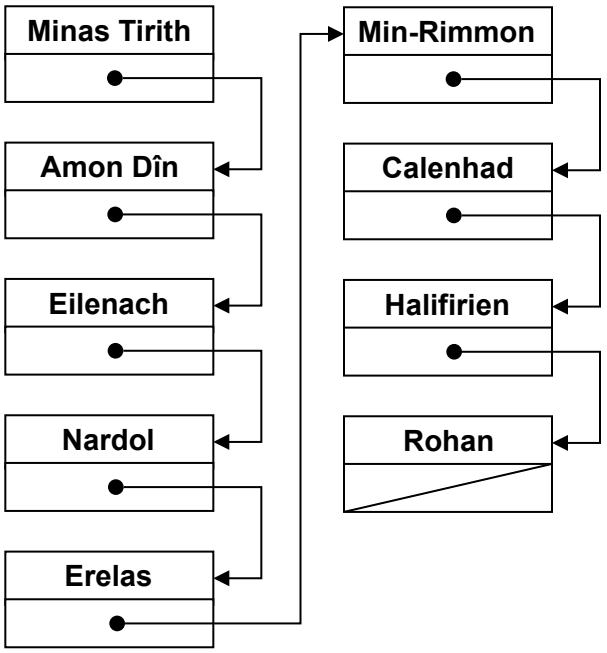
Lets Look at What Happens



Message Passing in Linked Lists

To represent this message-passing image, you might use a definition such as the one shown on the right.

You can then initialize a chain of `Tower` structures, like this:



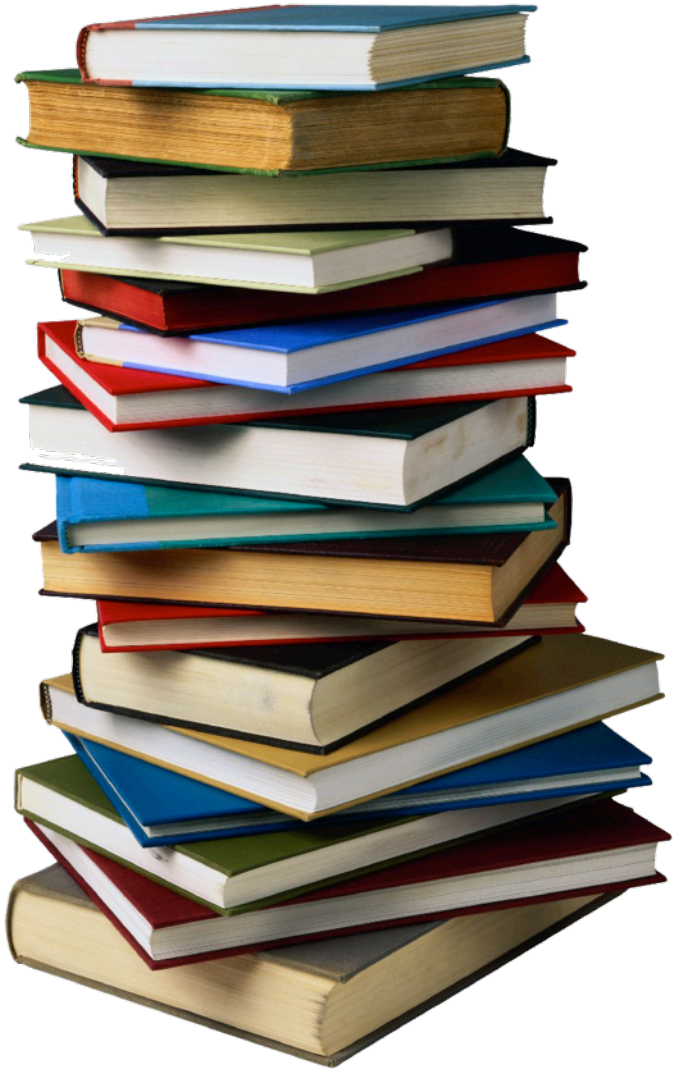
Calling `signal` on the first tower sends a message down the chain.

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

/*
 * Function: createTower(name, link);
 * ----------------------------------
 * Creates a new Tower with the specified values.
 */

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

/*
 * Function: signal(start);
 * ------------------------
 * Generates a signal beginning at start.
 */

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

Today's Goals

Today's Goals

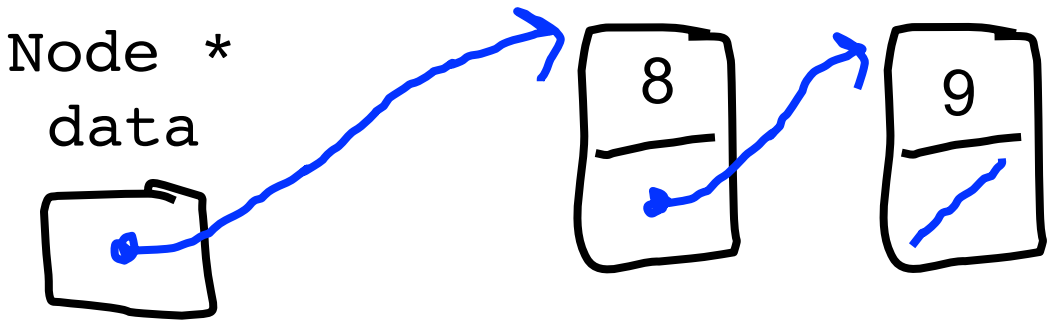
How is the Stack Implemented?



Beacons of Gondor

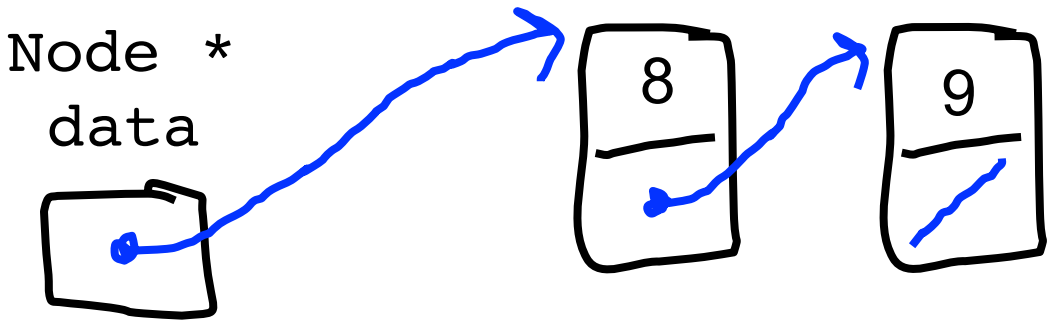
```
struct Node{
   int value;     /* The value of this elem    */
   Node *link;    /* Pointer to the next node */
};
```

Stack is a Linked List



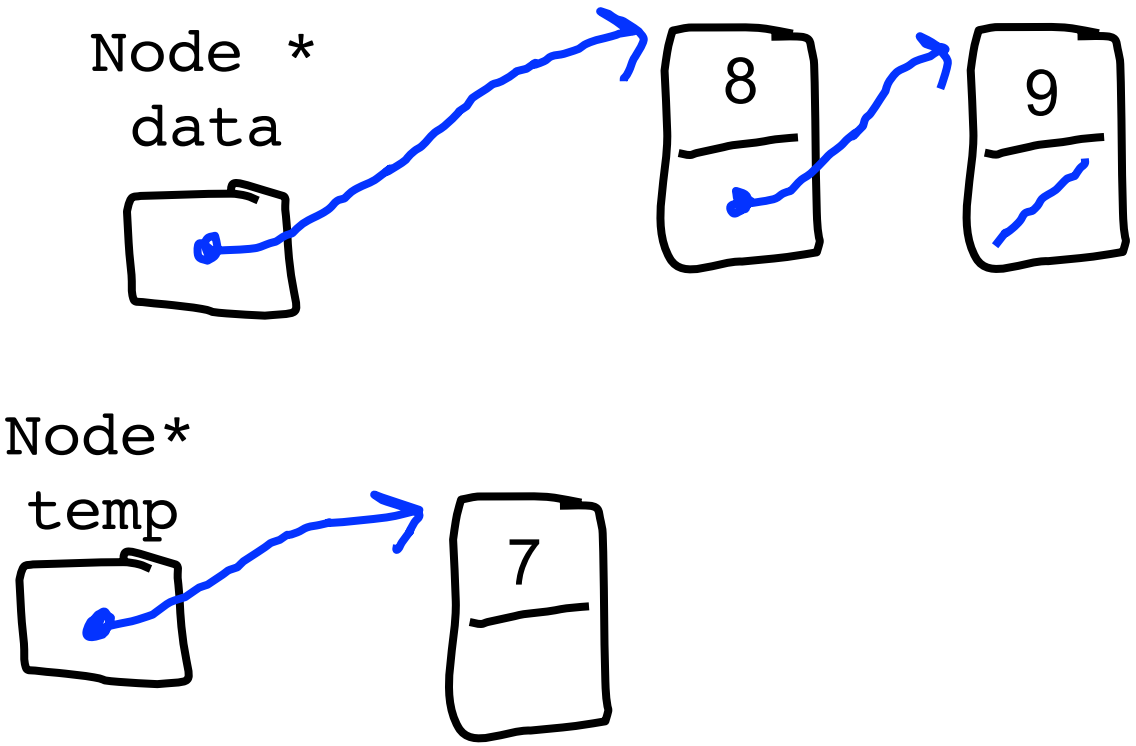
Stack is a Linked List

```
push(7);
```



Stack is a Linked List

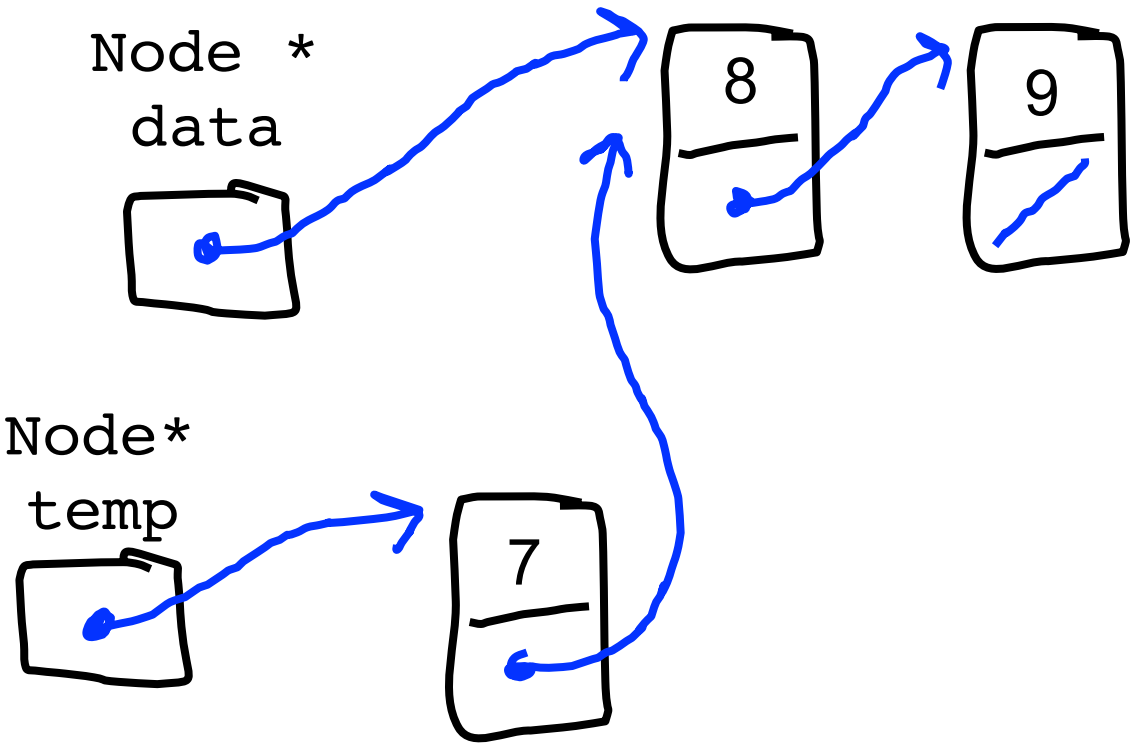
```
push(7);
```




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

Stack is a Linked List

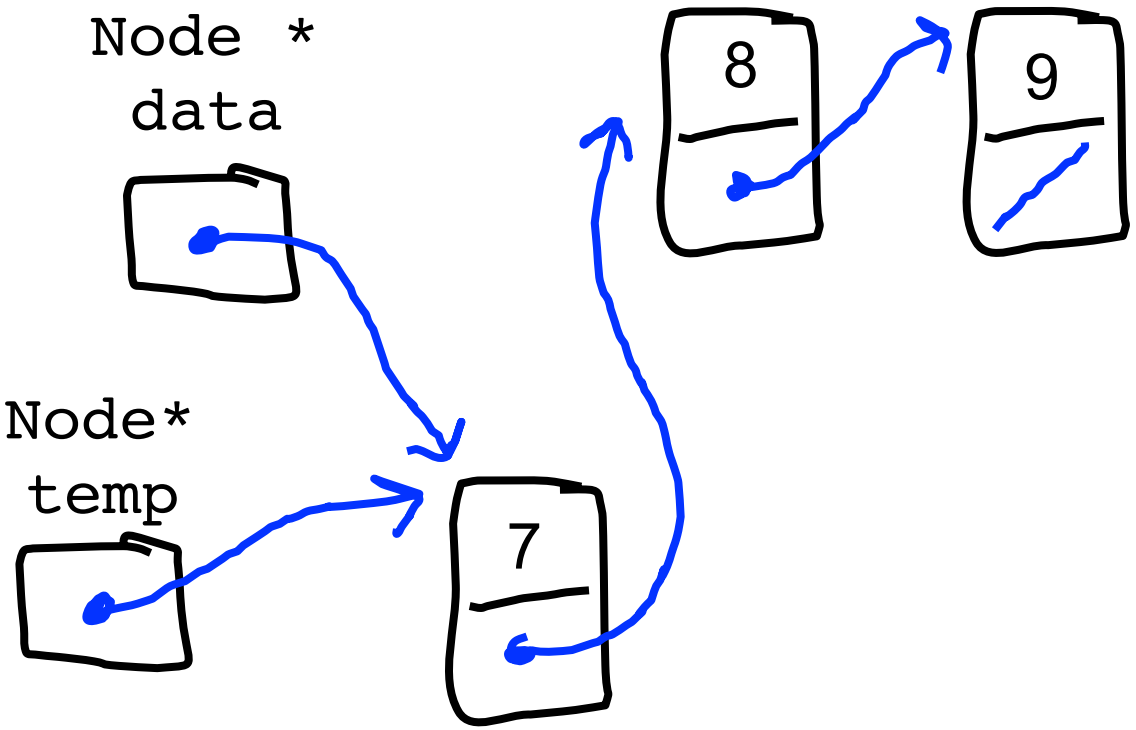
```
push(7);
```




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

Stack is a Linked List

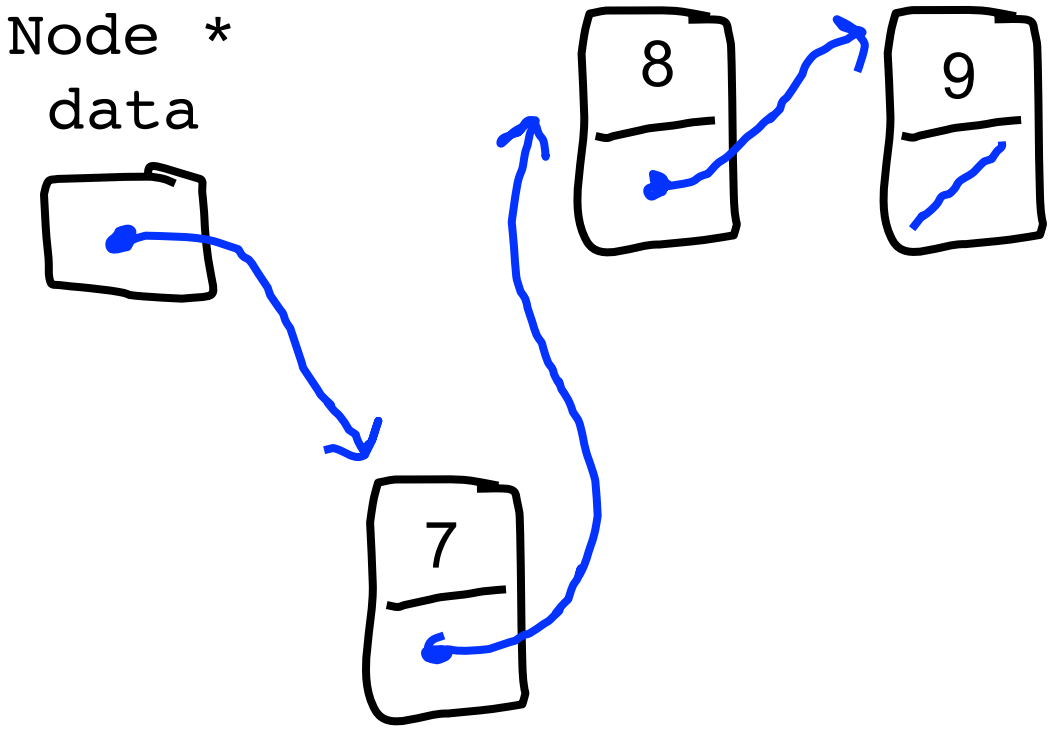
```
push(7);
```




```
data = temp;
```

Stack is a Linked List

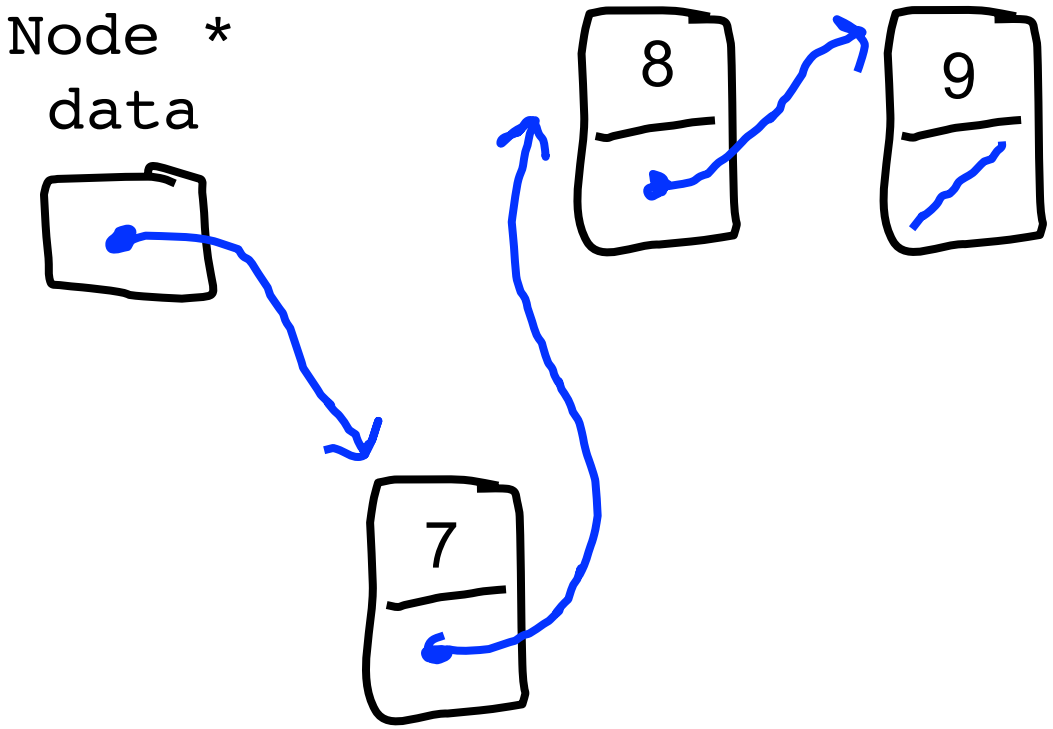
```
push(7);
```



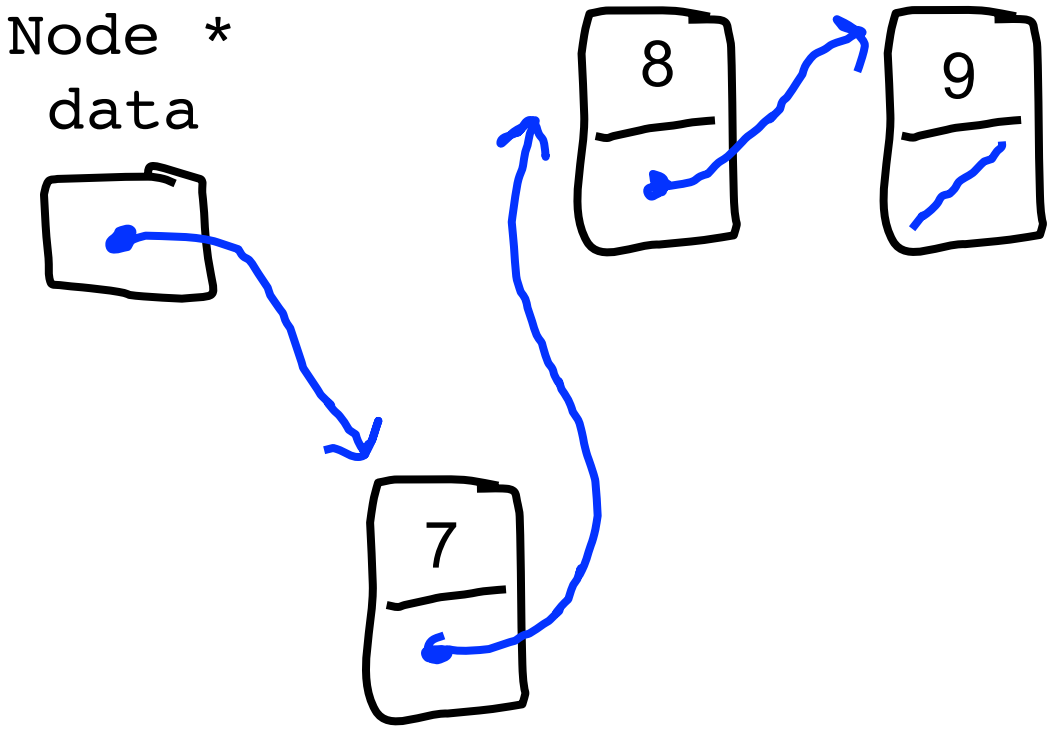
exit function

Stack is a Linked List

```
pop();
```



Stack is a Linked List



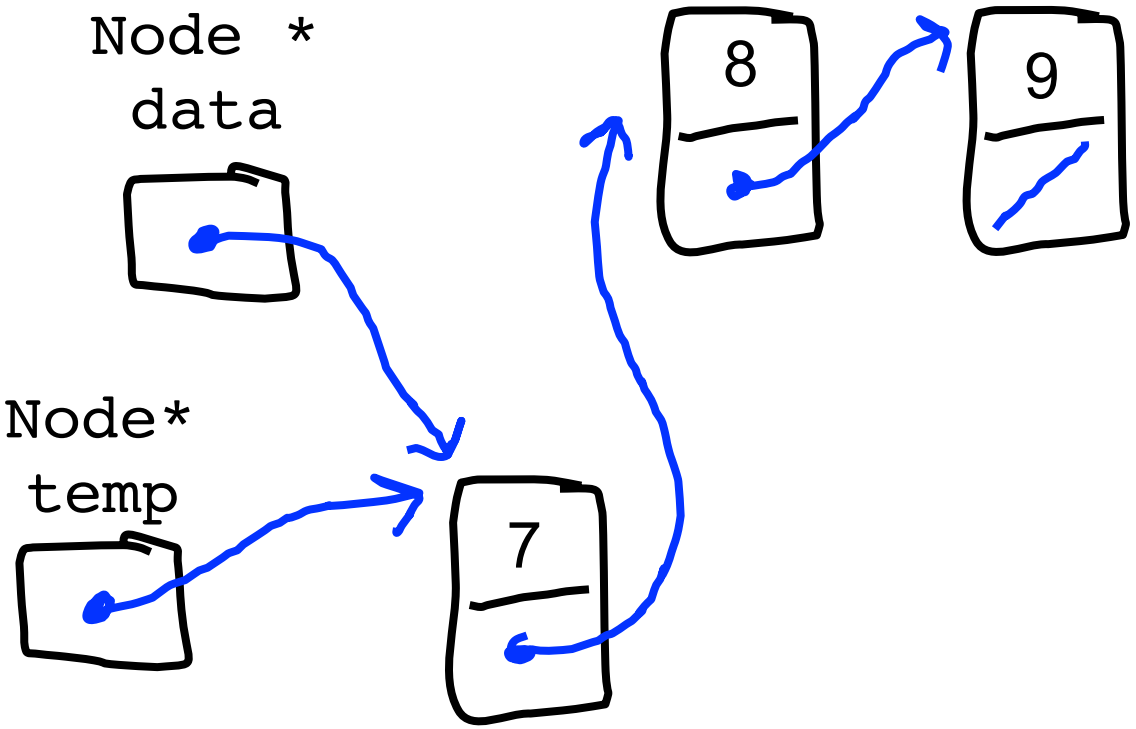

```
pop();
```

```
int return
```

7

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

Stack is a Linked List



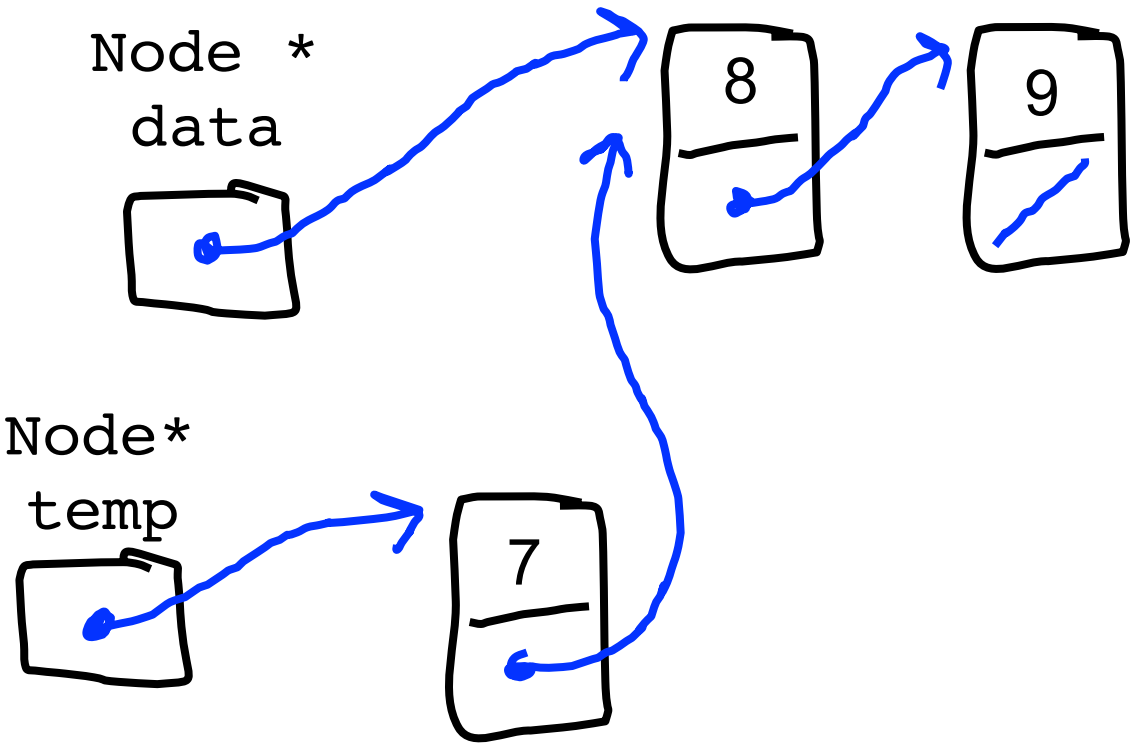

```
pop();
```

```
int return
```

```
7
```

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

Stack is a Linked List




```
pop();
```

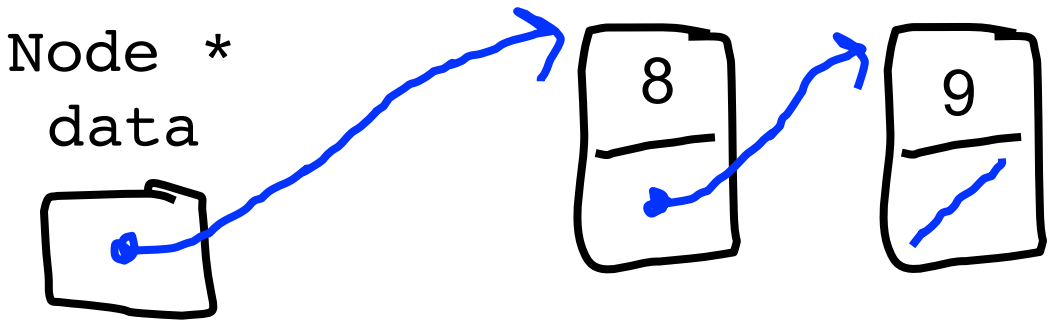
```
int return
```

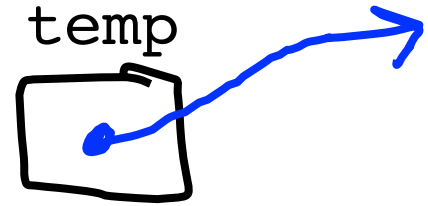
7

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

Stack is a Linked List

```
pop();
```



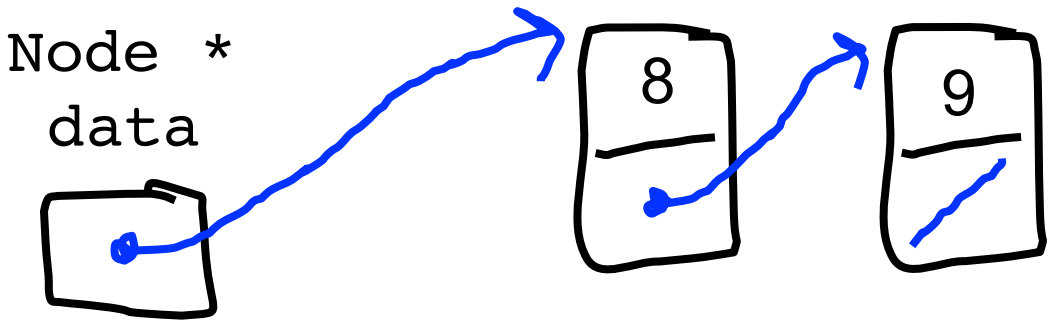



```
int return
```

```
7
```

```
delete temp;
```

Stack is a Linked List




```
pop();
```

```
int return
```

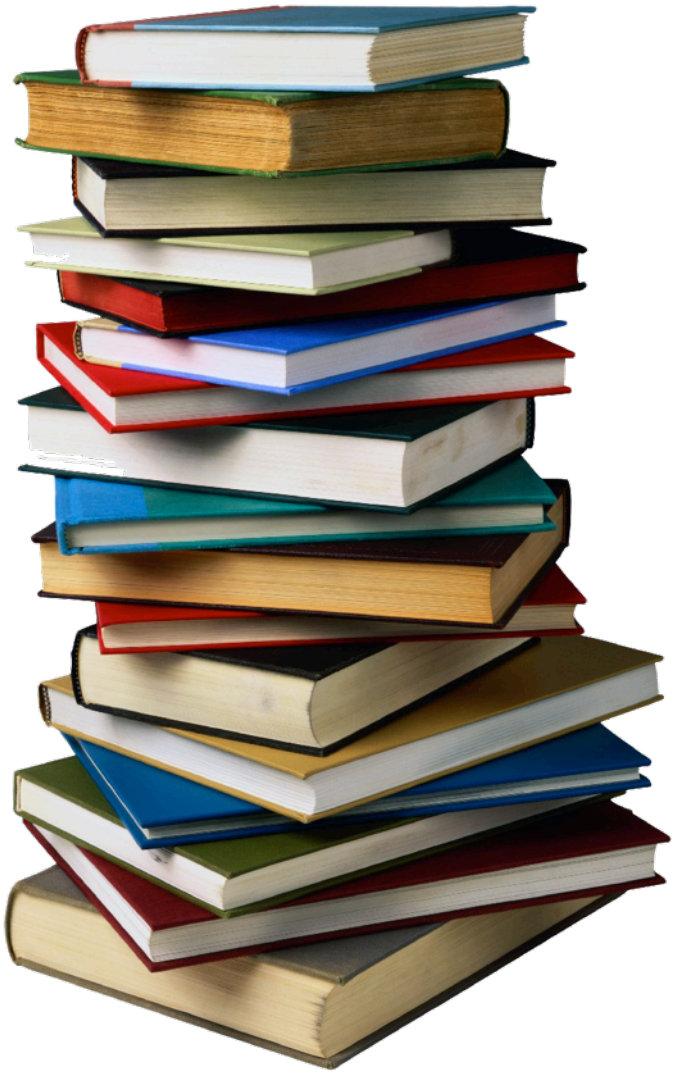
7

exit function + return

Big O of Push?

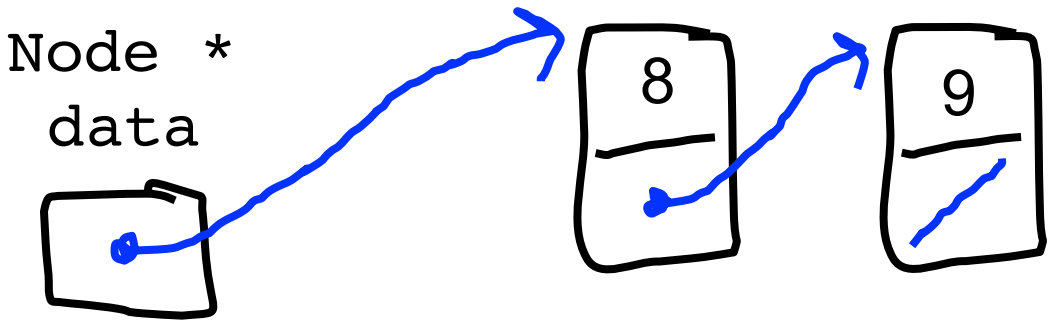
Big O of Pop?

Stack in C++

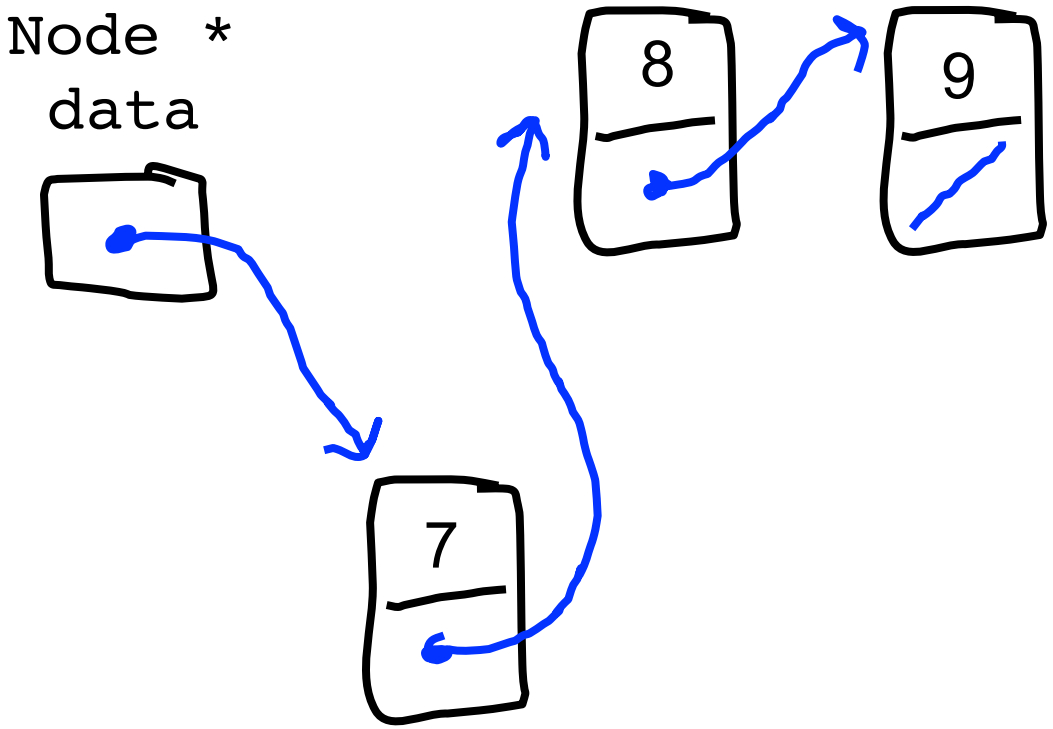


How is the Queue Implemented?

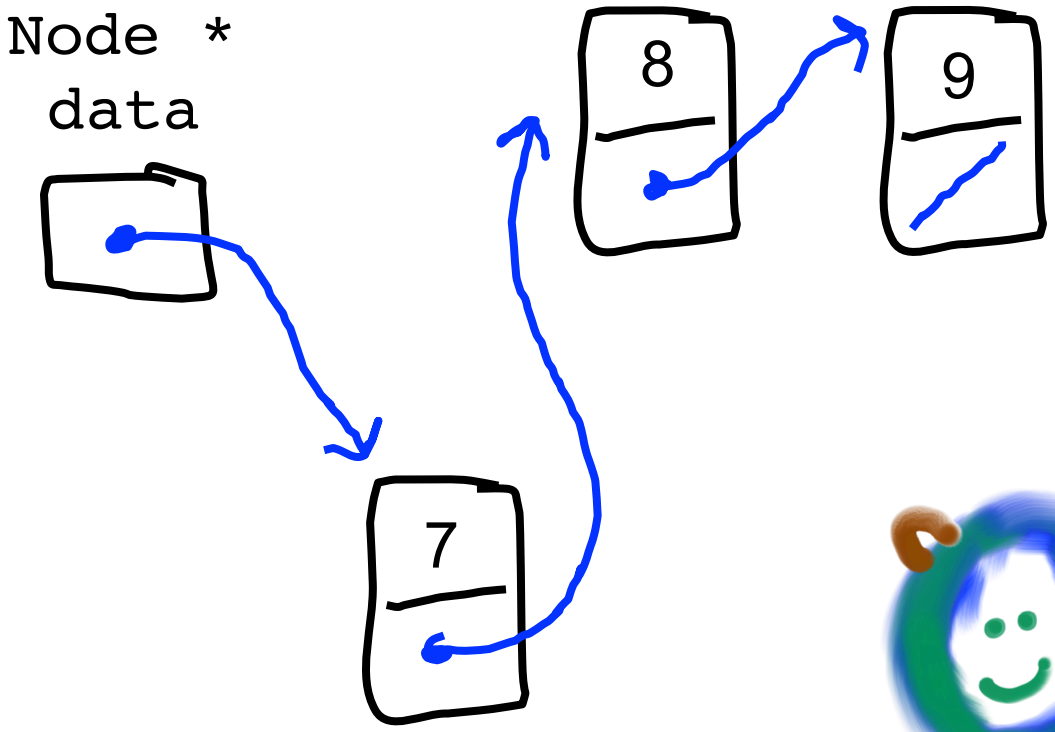
Queue?



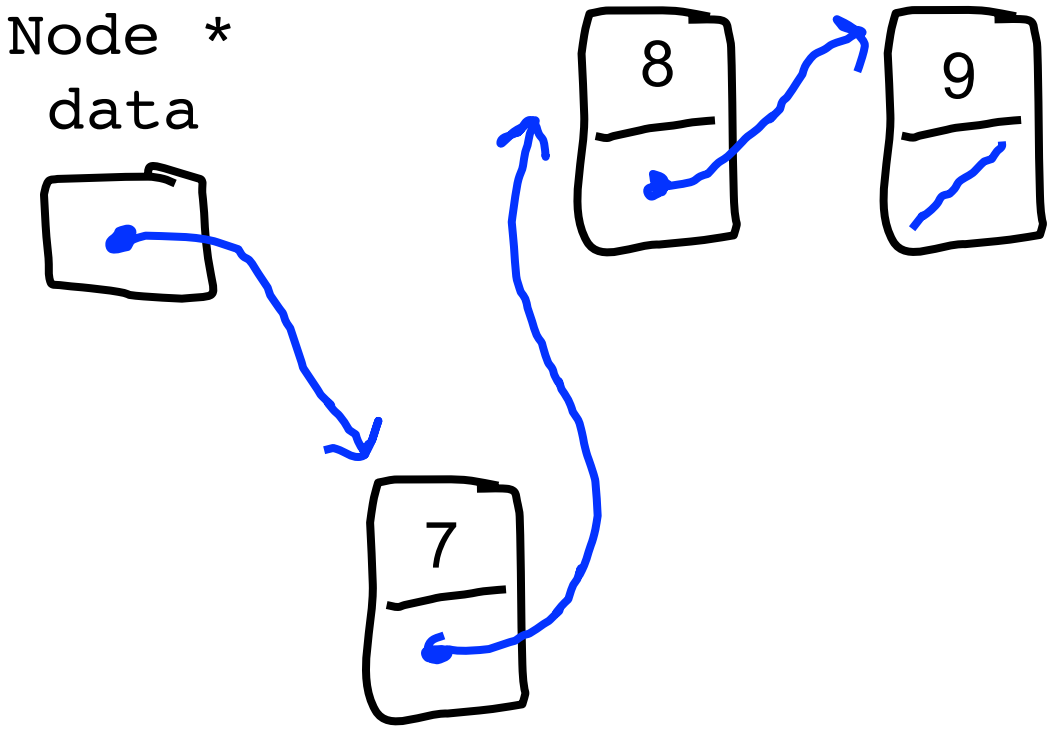
Queue Enqueue?



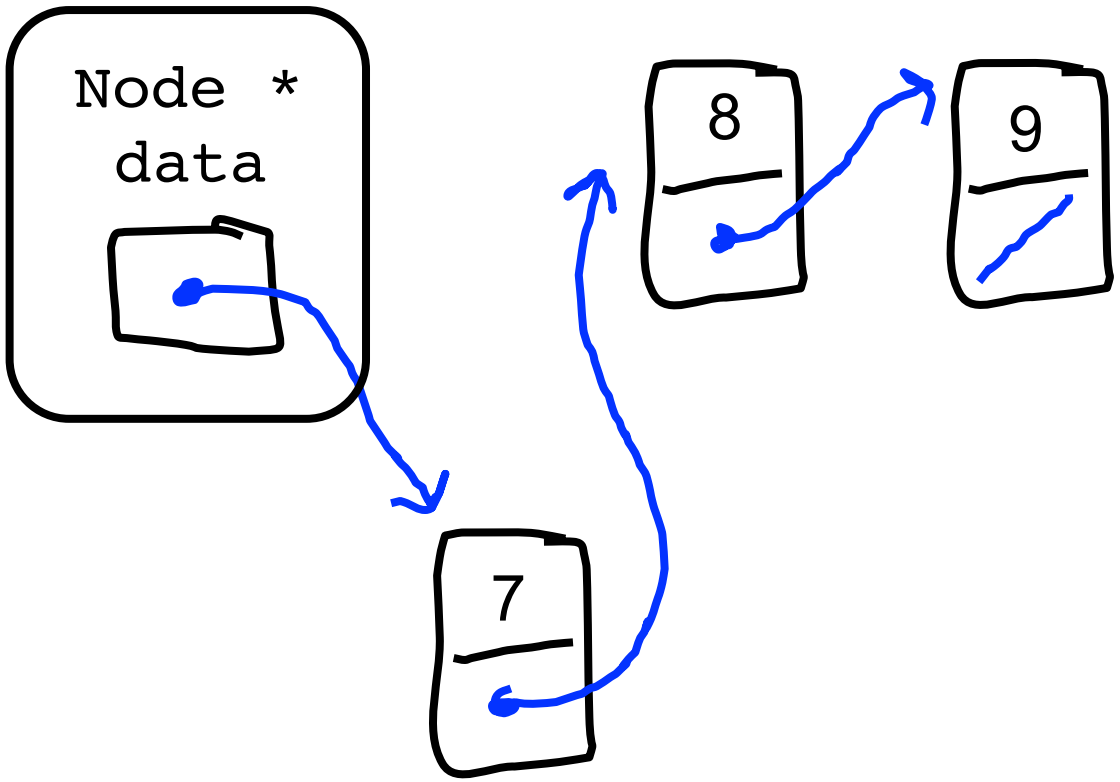
Queue Enqueue?



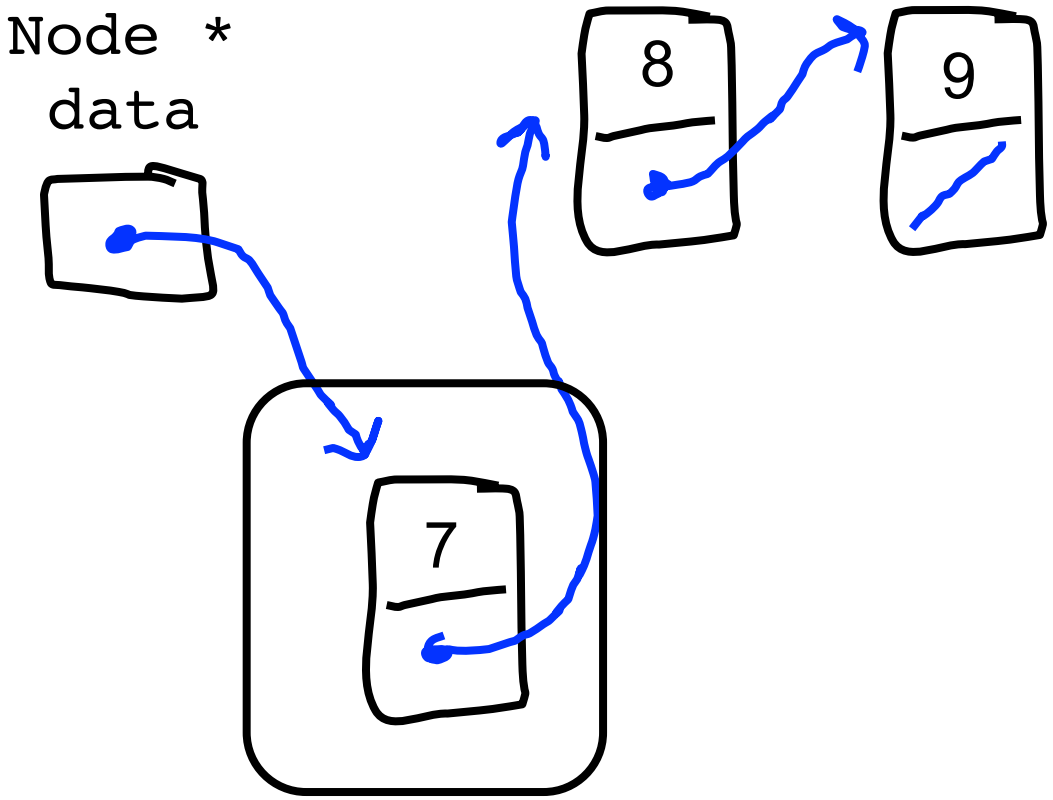
Queue Dequeue?



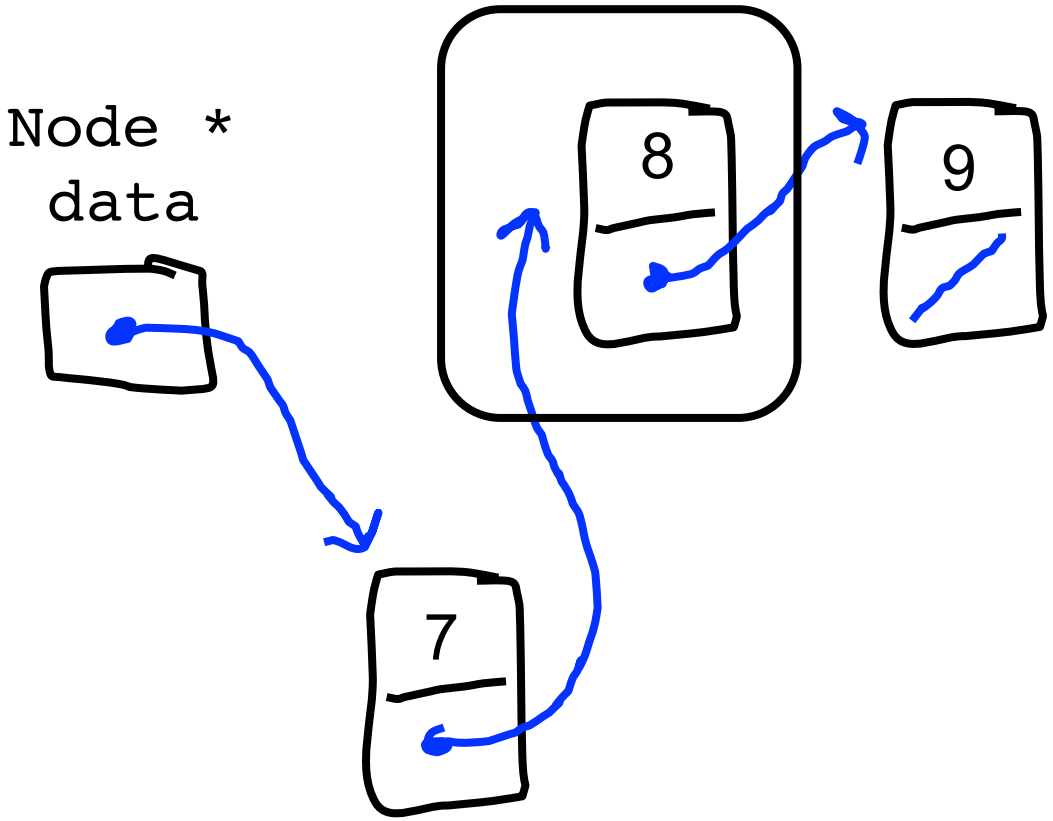
Queue Dequeue?



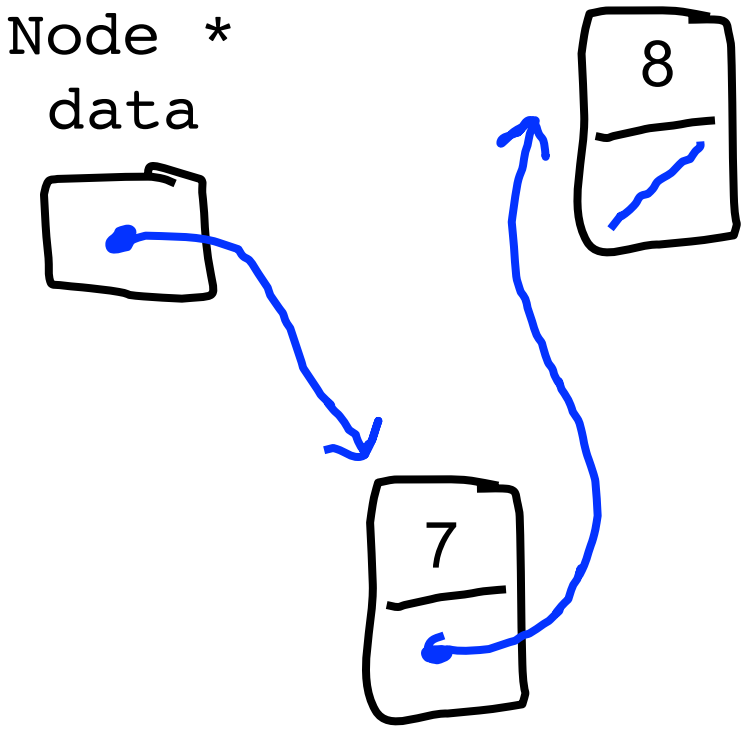
Queue Dequeue?



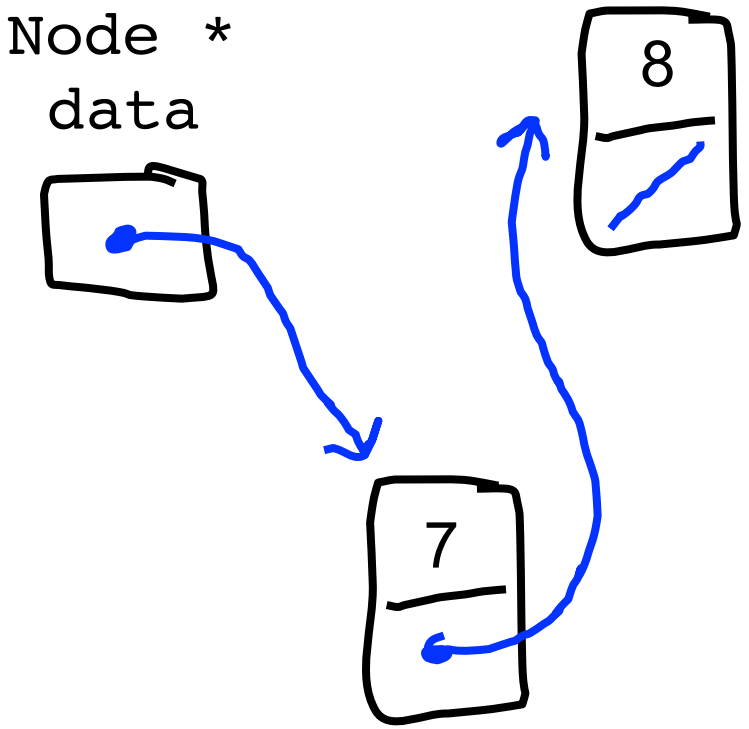
Queue Dequeue?



Queue Dequeue?



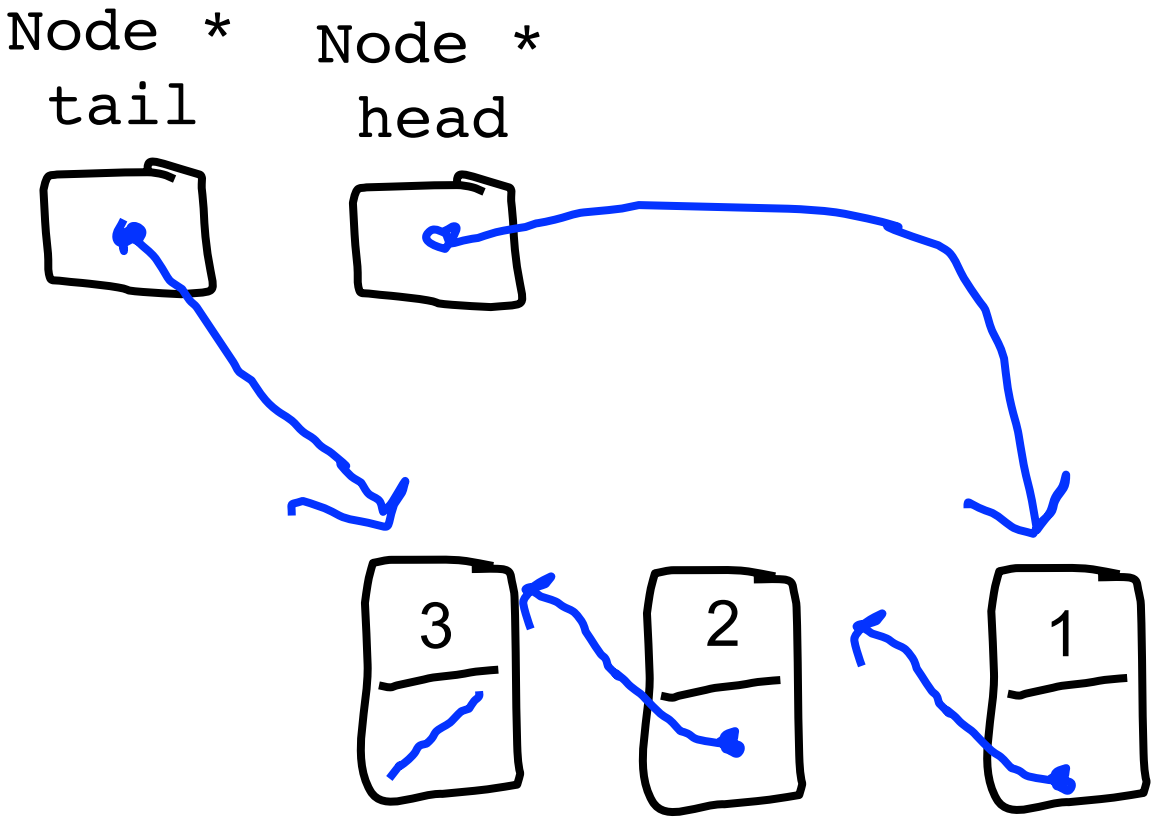
Queue Dequeue?



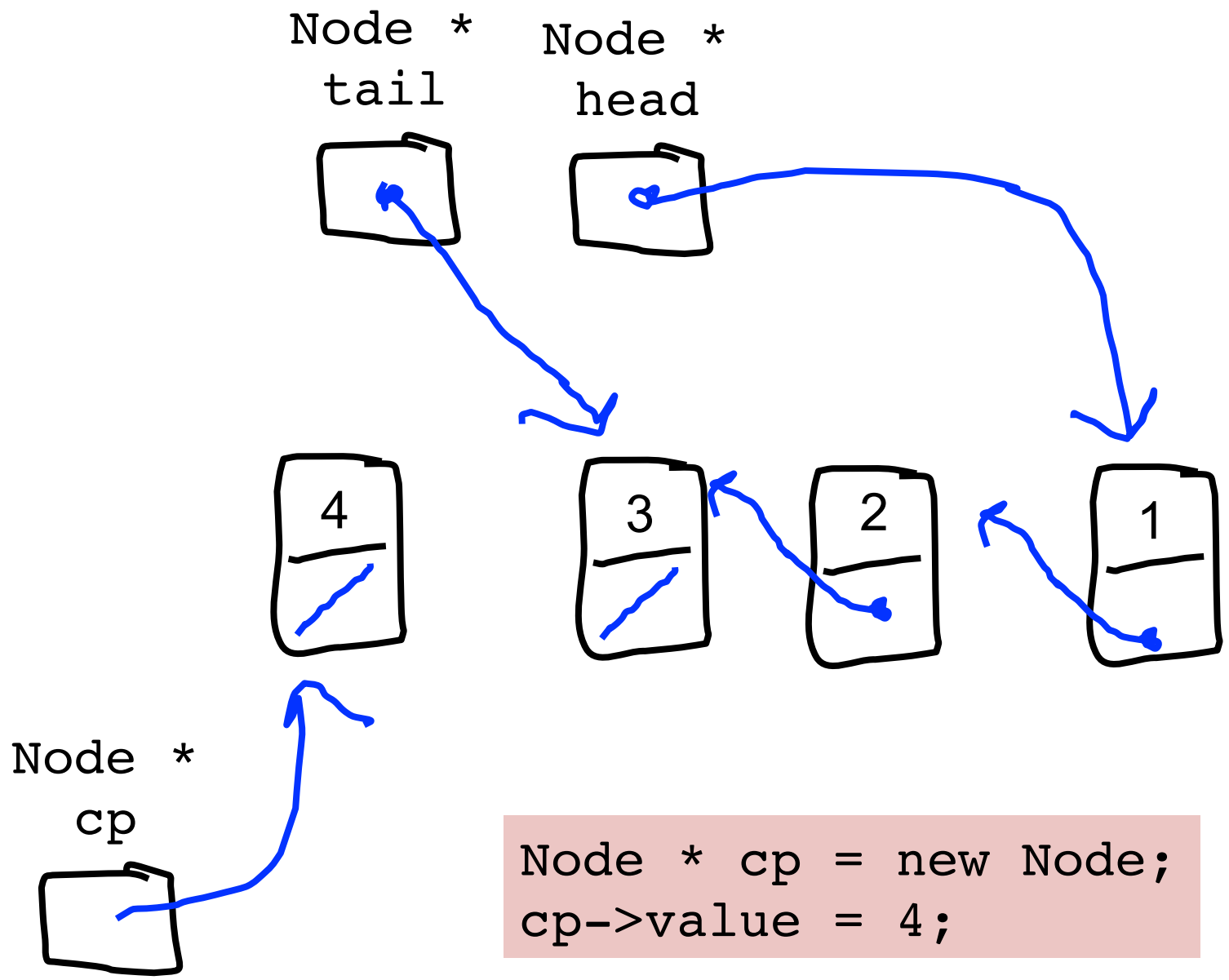
$$O(n)$$

Always a Better Way

Queue Enqueue?

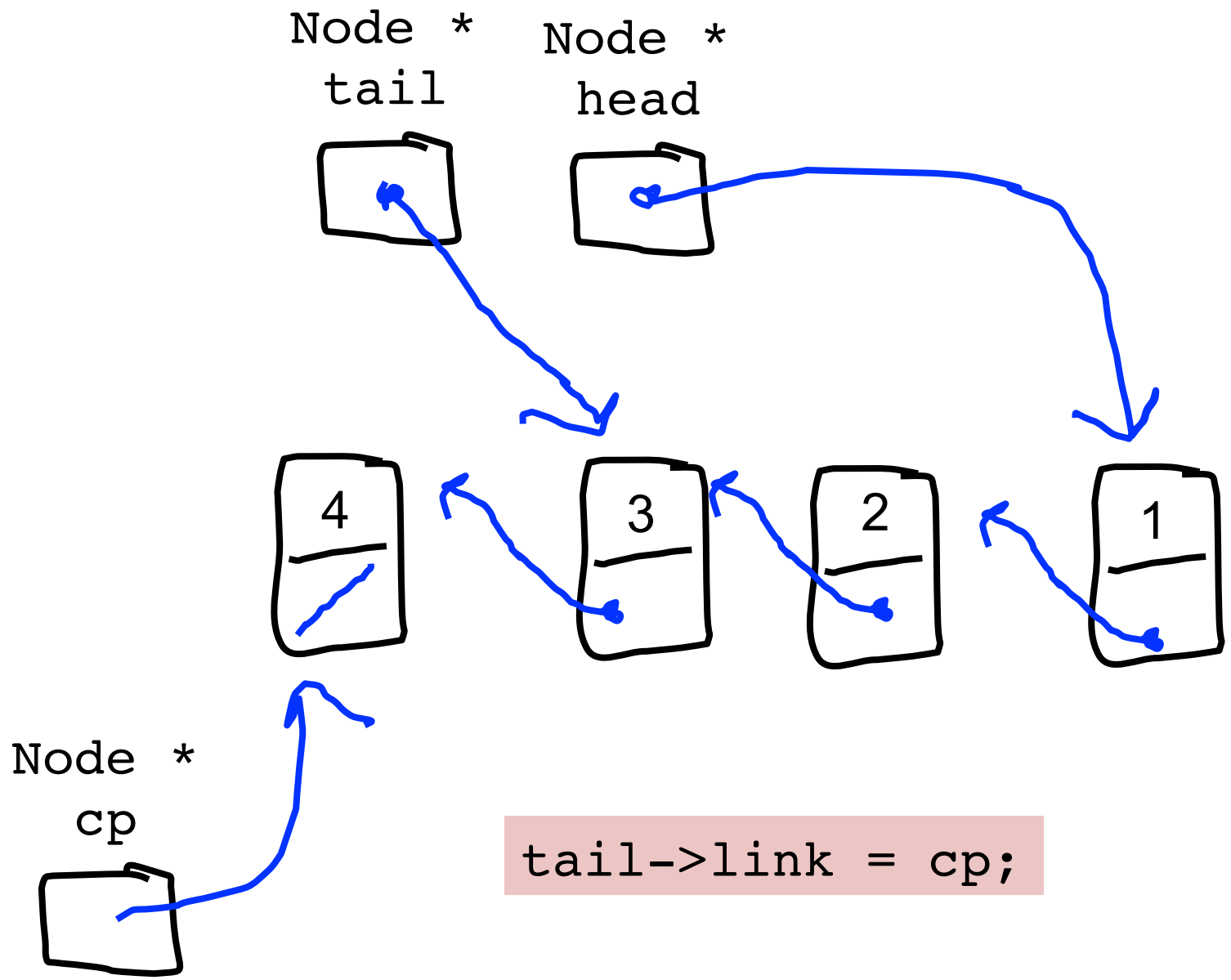


Queue Enqueue?



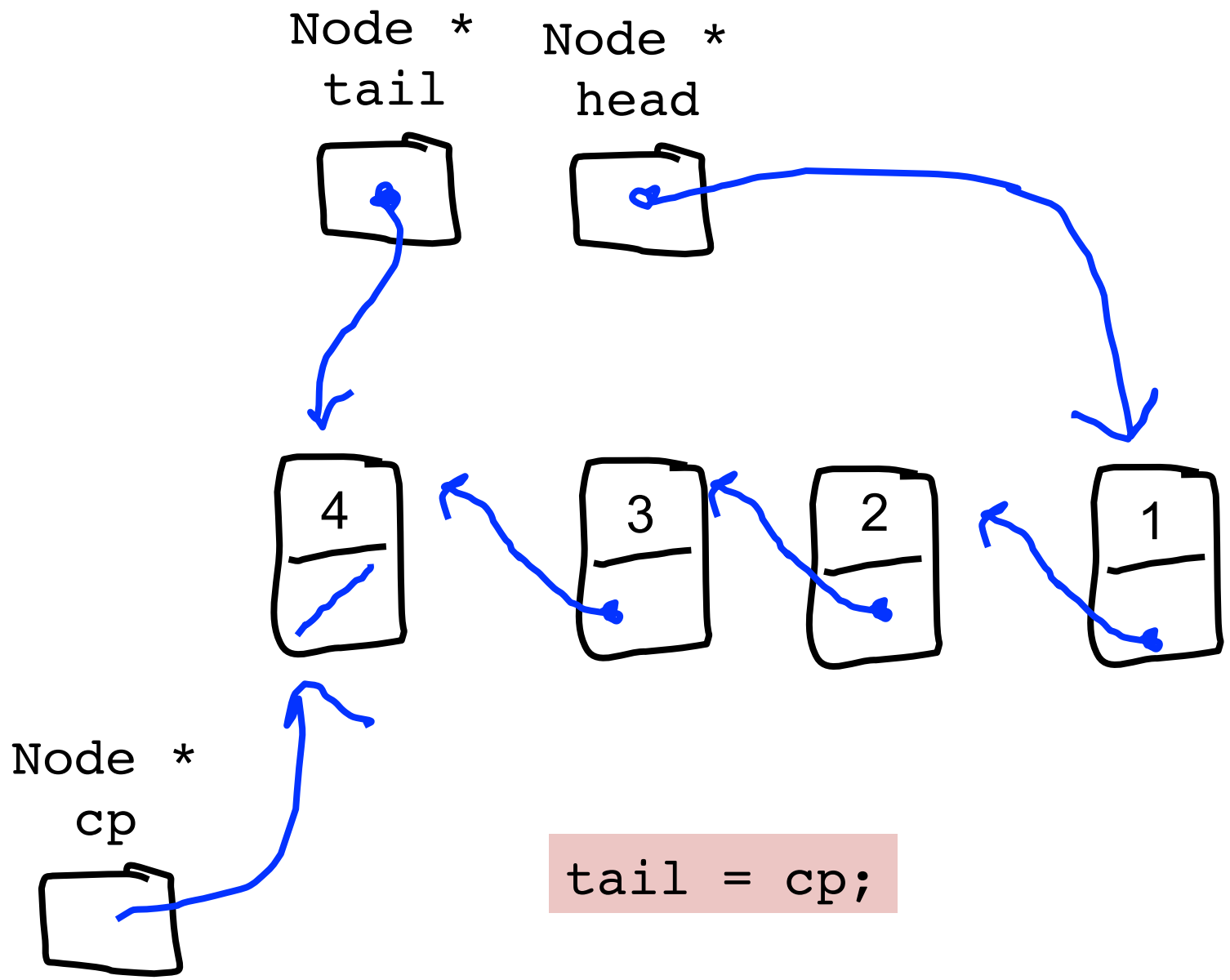

```
Node * cp = new Node;
cp->value = 4;
```

Queue Enqueue?

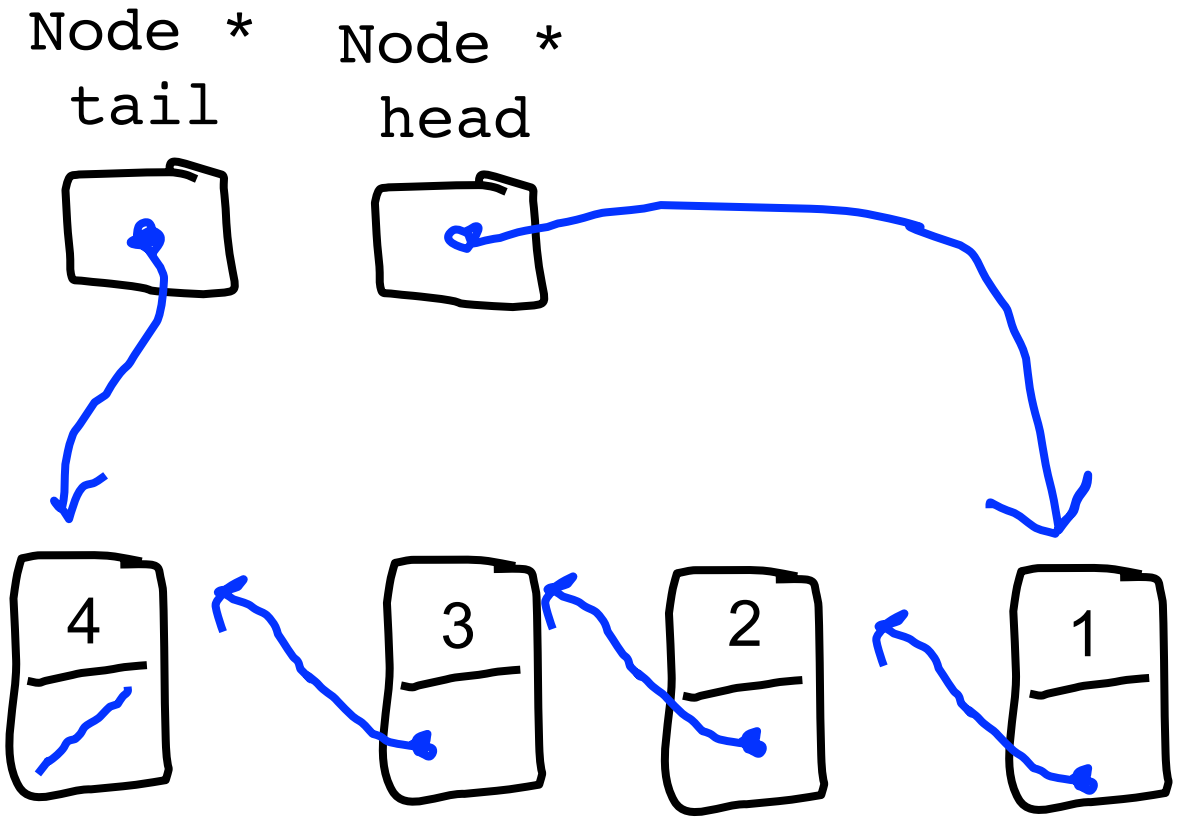



```
tail->link = cp;
```

Queue Enqueue?

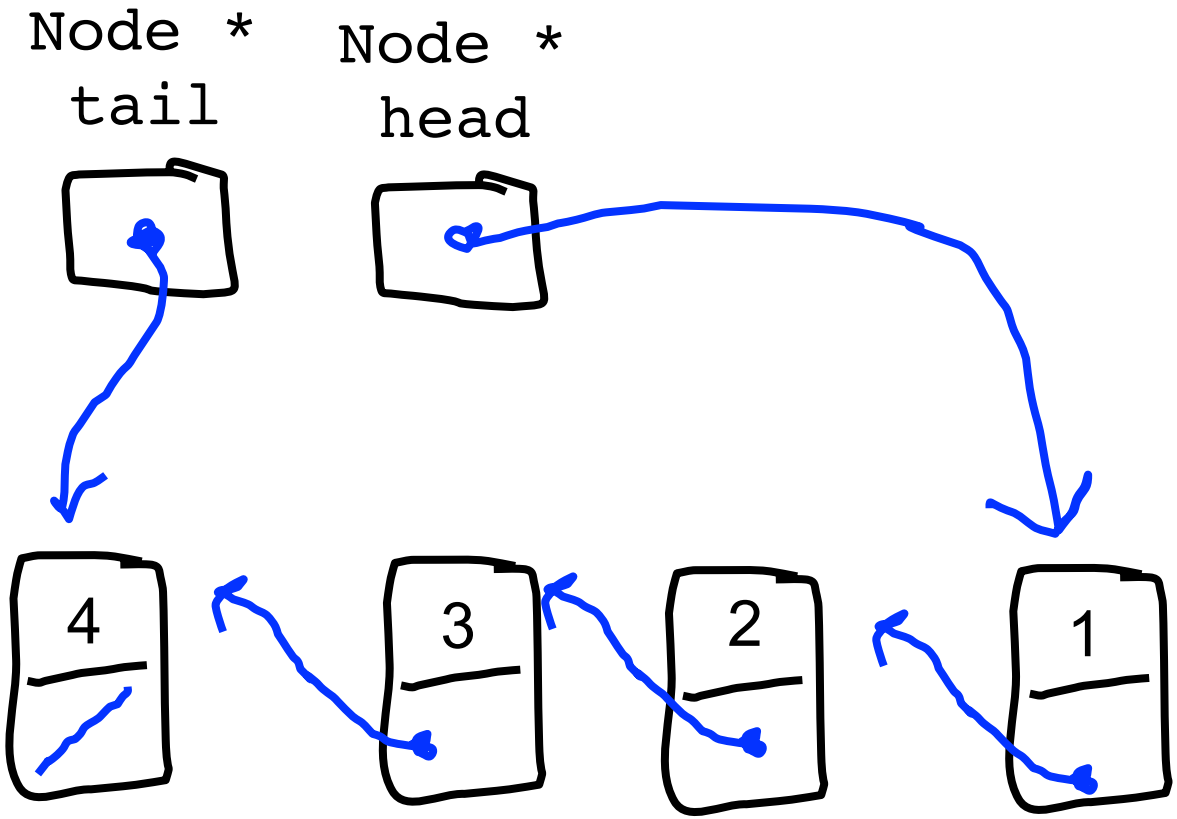


Queue Enqueue?



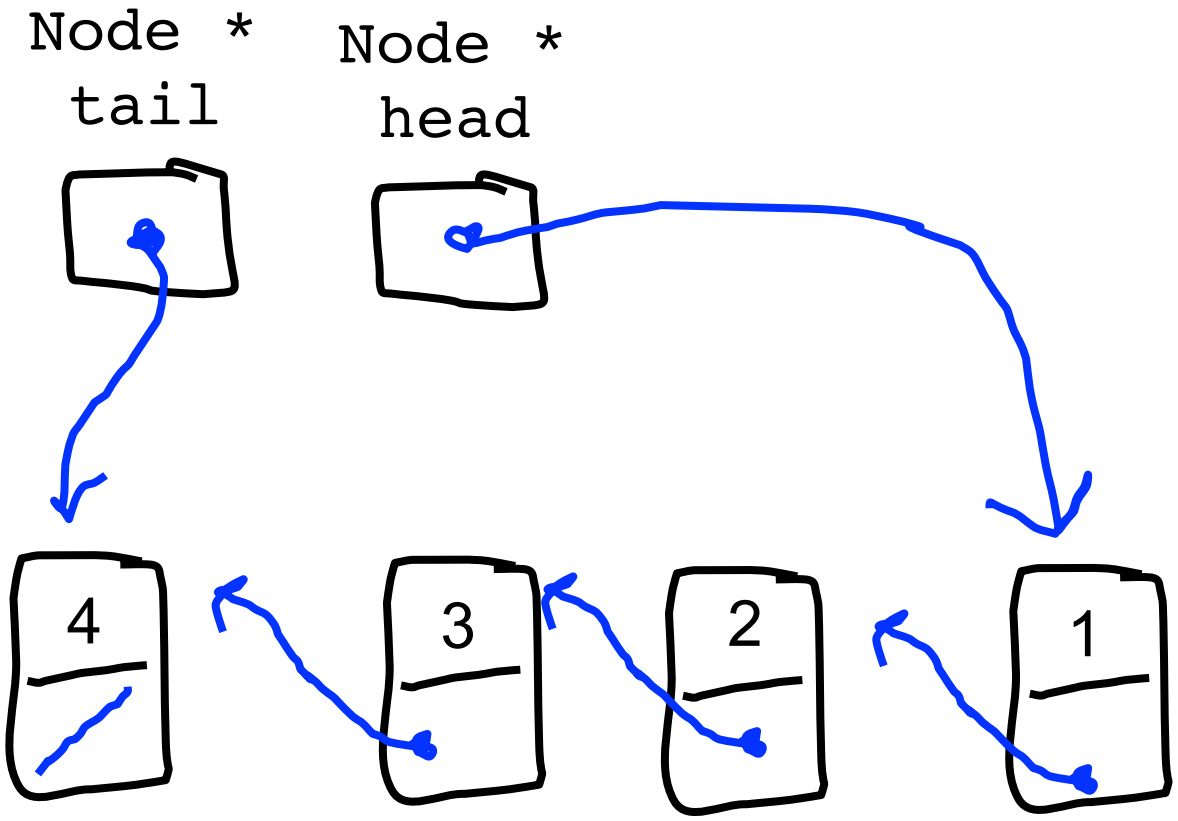

```
return
```

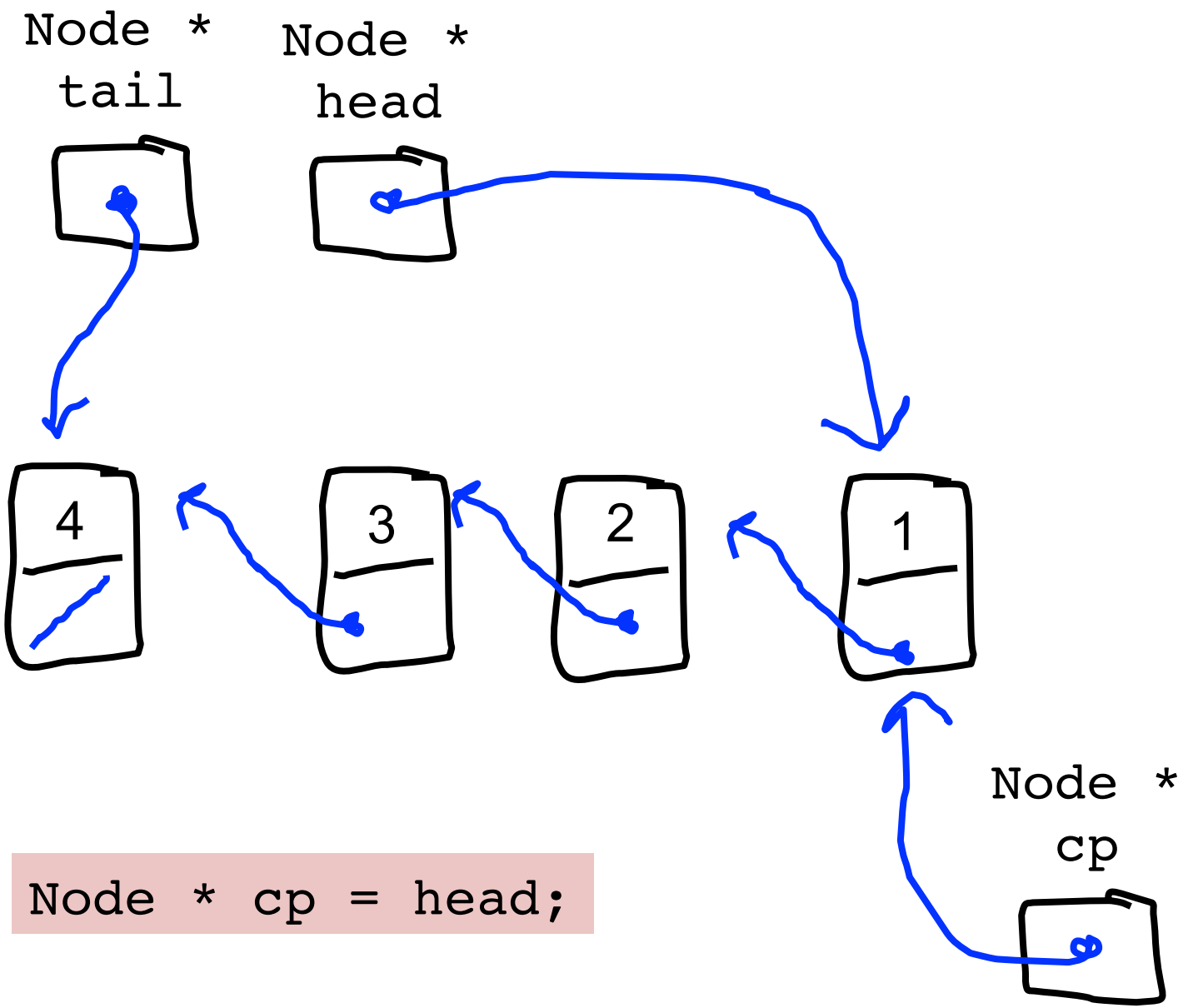
Queue Enqueue?



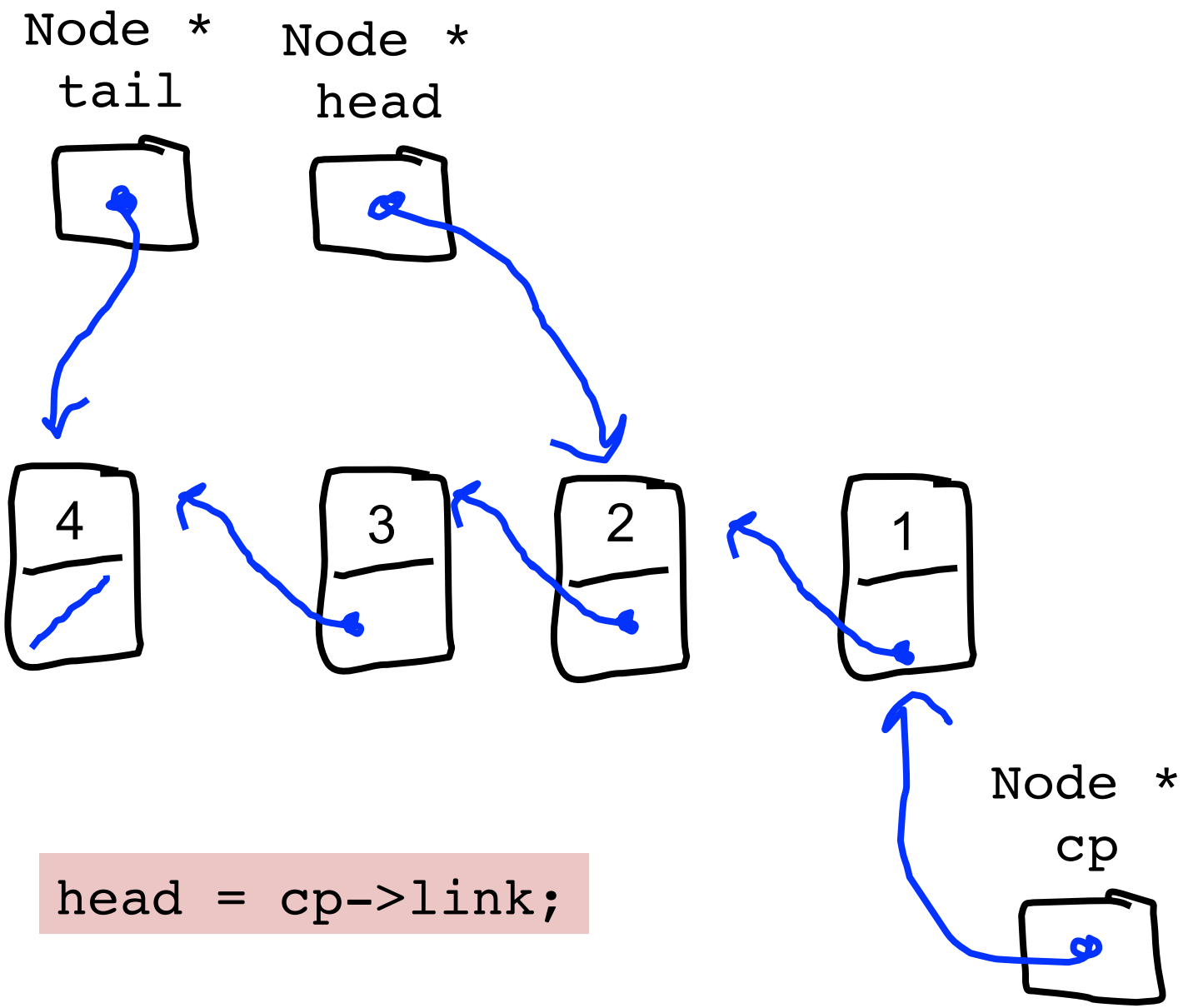
Dequeue

Queue Enqueue?

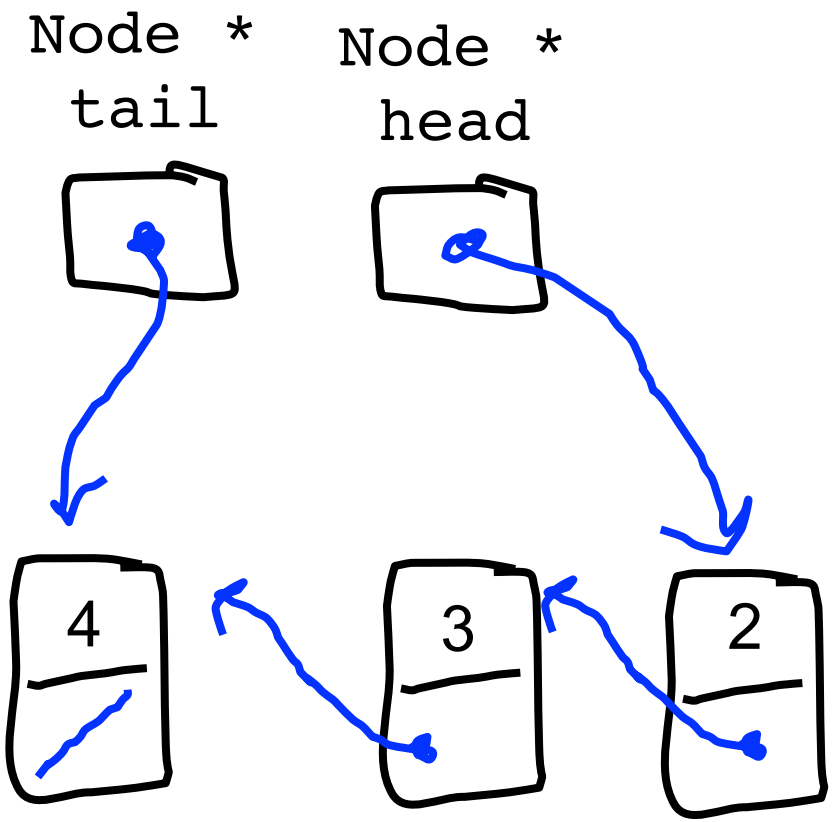




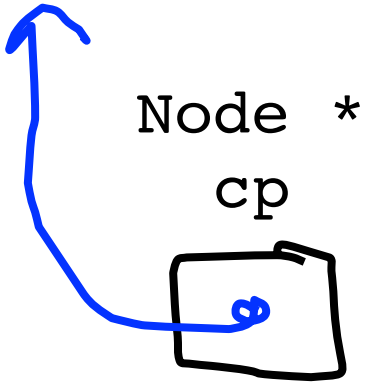

```
Node * cp = head;
```

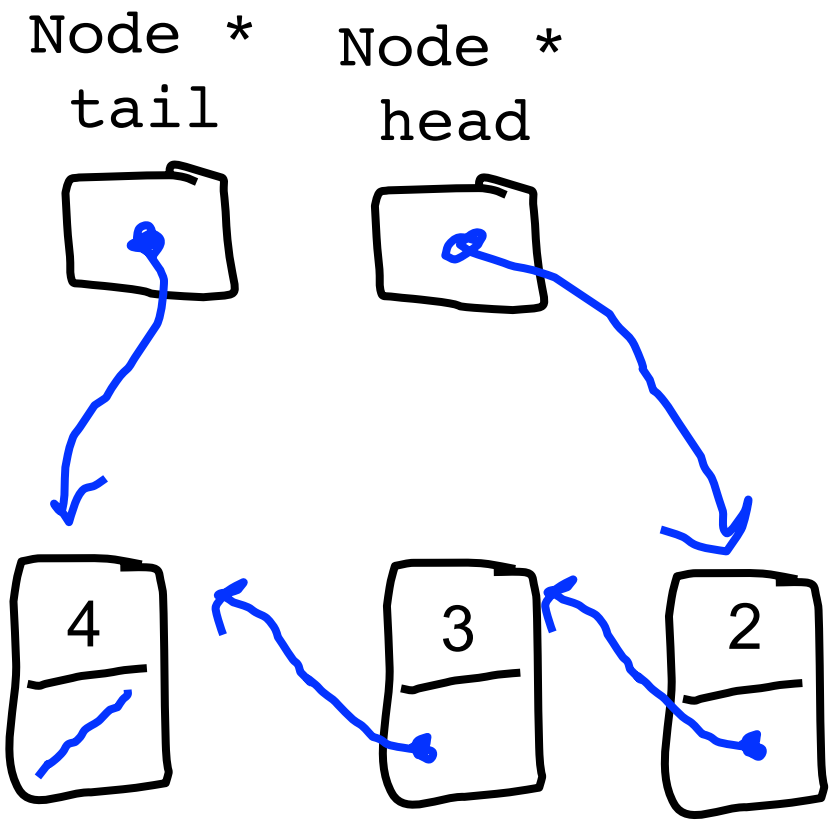



```
head = cp->link;
```

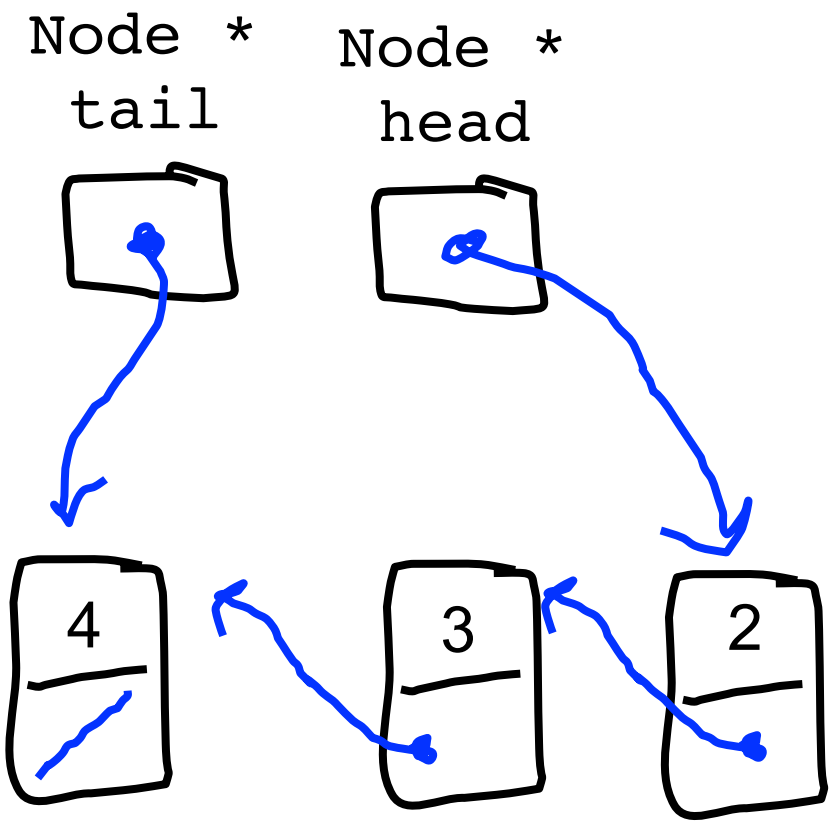



```
delete cp;
```






```
return 1
```

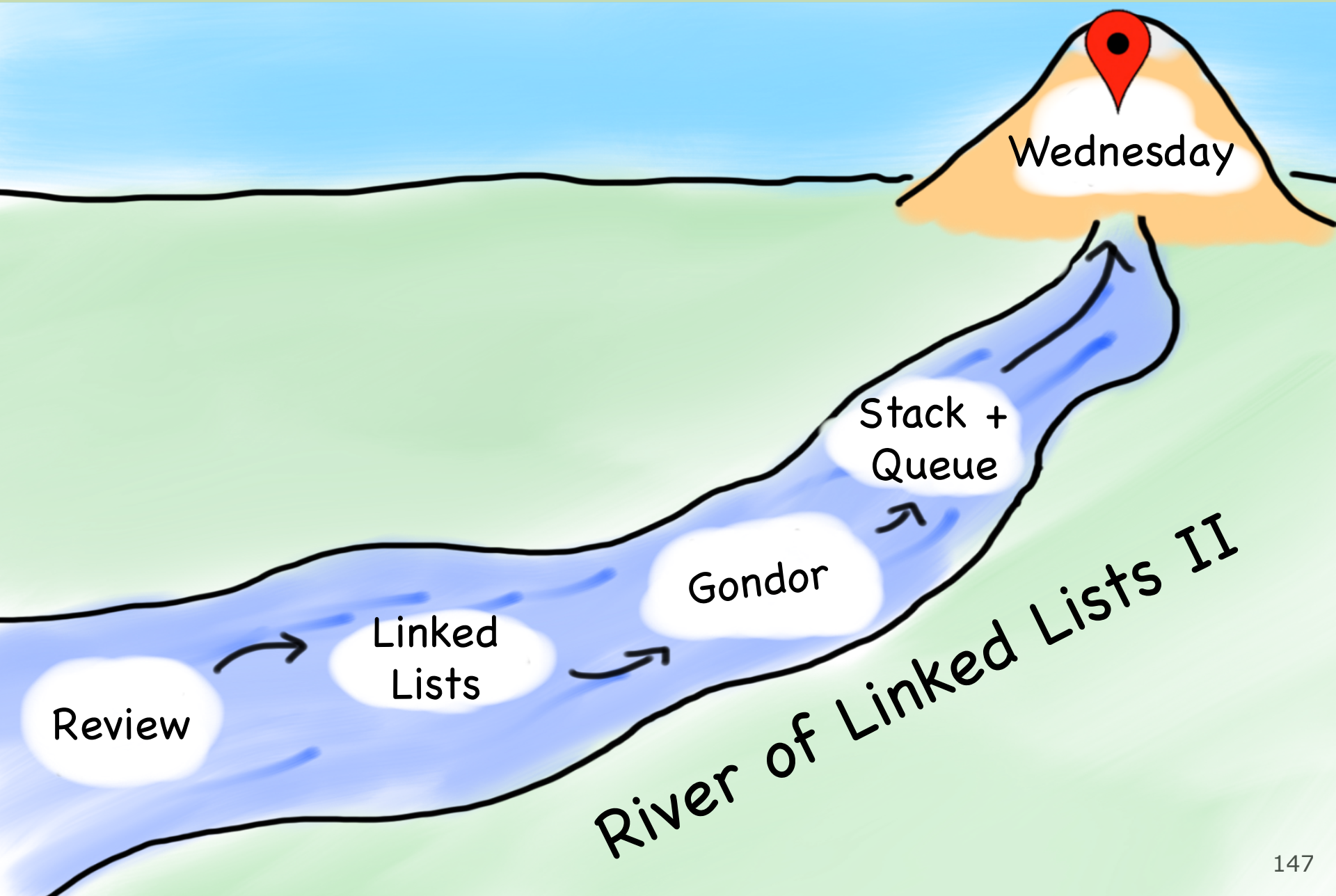

Node *
tail
Node *
head
4
3
2

Summary

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

Today's Goals

Today's Goals



Today's Goals

1. Round out knowledge of linked lists
2. See how Stack + Queue work