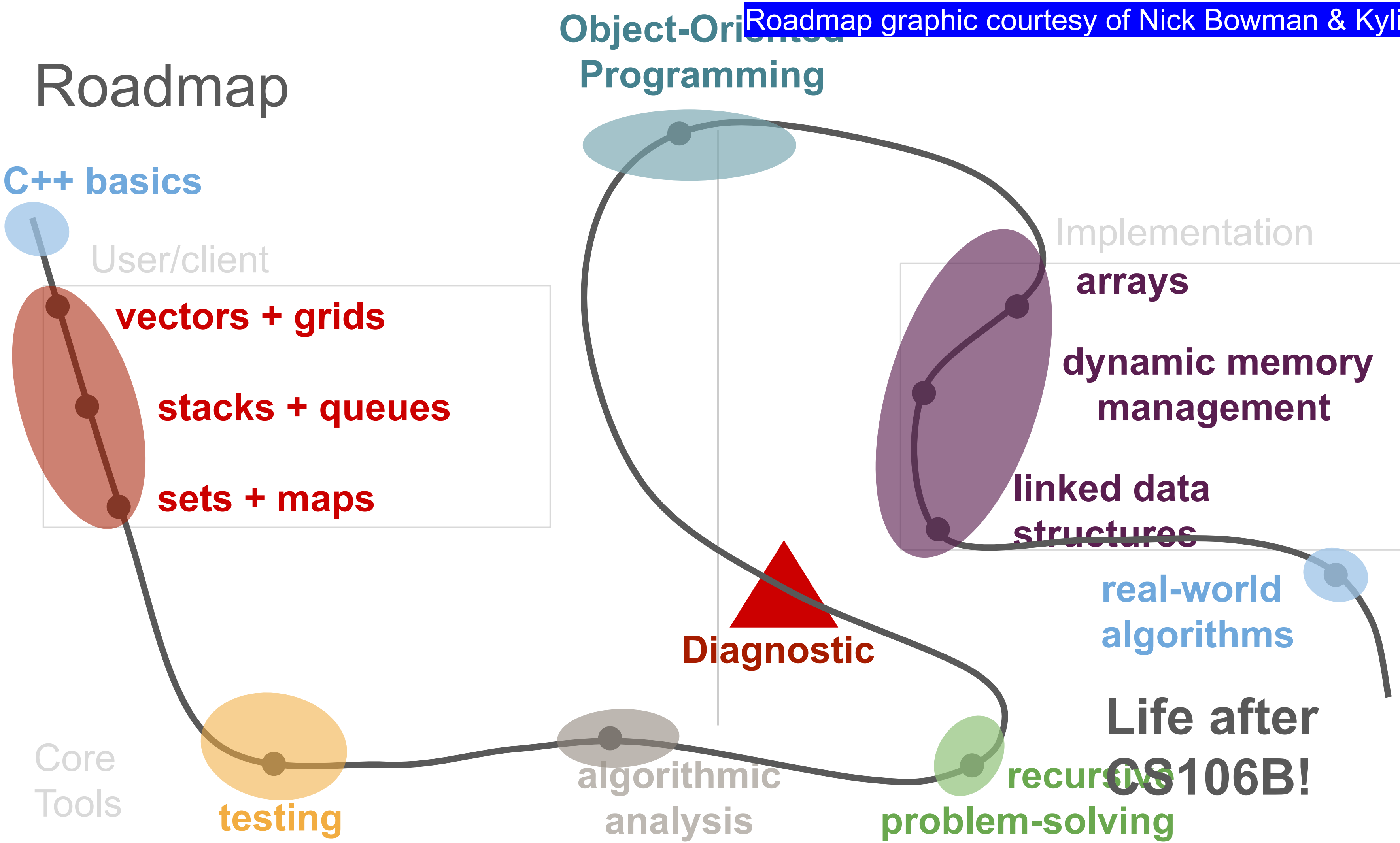
Linked Lists

For something very important to you, would you rather rely on your own memory or computer memory?

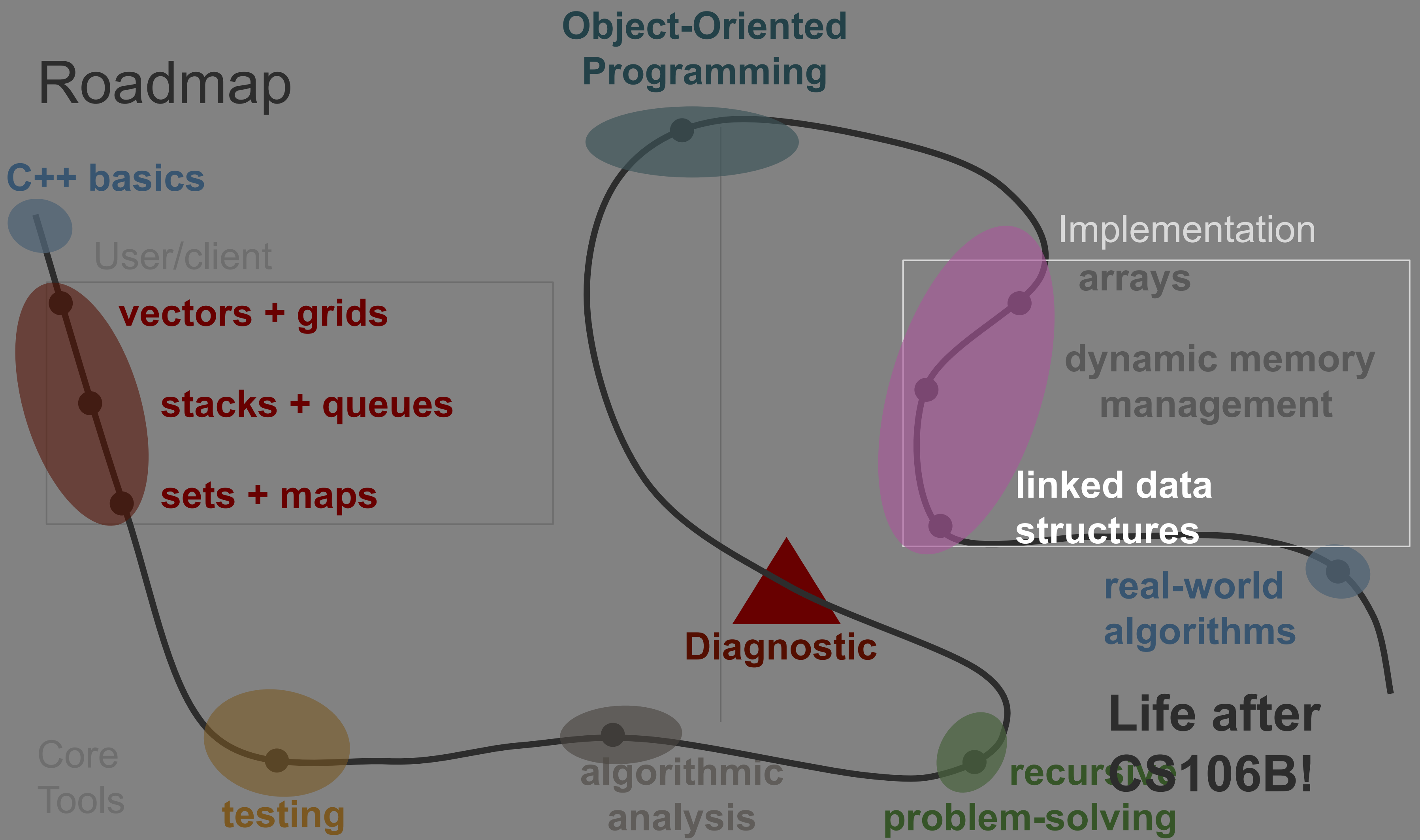
(put your answers in the chat)

Roadmap

Roadmap graphic courtesy of Nick Bowman & Kylie Jue



Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Today's question

How can we use pointers to organize non-contiguous memory on the heap?

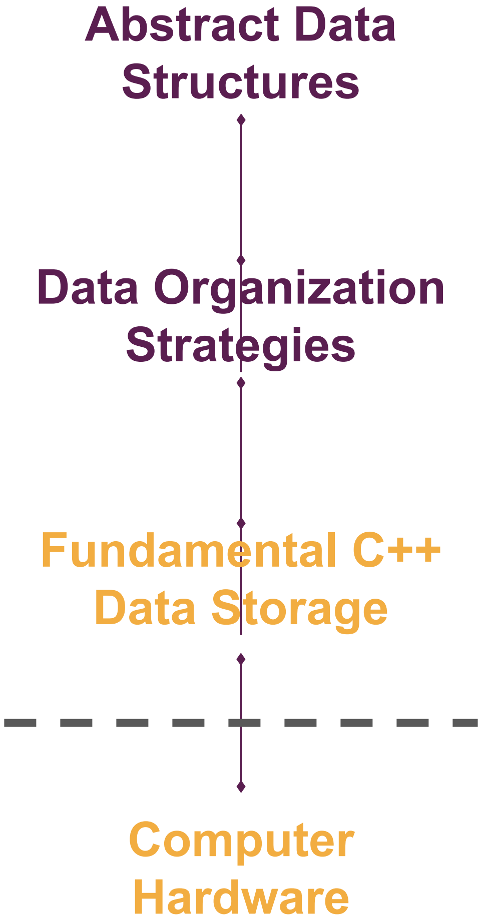
Today's topics

1. Review
2. What is a linked list?
3. How do we manipulate linked lists?

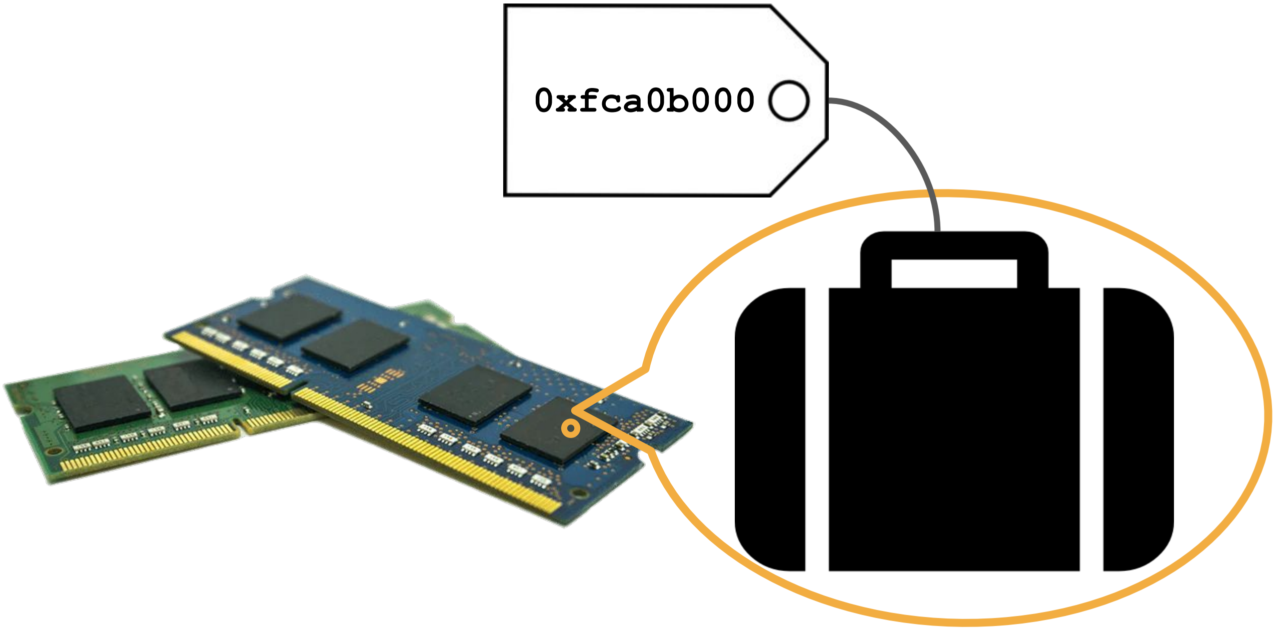
Review

[memory and pointers]

Levels of abstraction



How is computer memory organized?



Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap).

Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap).

- A pointer is just a type of variable that stores a memory address!

Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap).

- A pointer is just a type of variable that stores a memory address!
  - You specify the type of the variable that it points to so that C++ knows how much space the value its pointing to is taking up (e.g. **string*** or **int*** or **Vector***).

Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap).

- A pointer is just a type of variable that stores a memory address!
  - You specify the type of the variable that it points to so that C++ knows how much space the value its pointing to is taking up (e.g. **string*** or **int*** or **Vector***).
  - But remember that pointers and what they point to (e.g. **string** vs. **string***) are two completely different data types!

Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap)

- A pointer is just a type of variable that stores a memory address!

- When you **dynamically allocate** variables on the heap, you must use the keyword **new** (or **new[]** for arrays) and must store the address in a pointer to keep track of it.
  - E.g. **int* number = new int;**

Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap)

- A pointer is just a type of variable that stores a memory address!

- When you **dynamically allocate** variables on the heap, you must use the keyword **new** (or **new[]** for arrays) and must store the address in a pointer to keep track of it.
    - E.g. **int* number = new int;**

Dynamically allocated variables
are the only reason we'll use pointers in this class!

Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap)
- A pointer is just a type of variable that stores a memory address!
- When you **dynamically allocate** variables on the heap, you must use the keyword **new** (or **new[]** for arrays) and must store the address in a pointer to keep track of it.
- To get the value located at the memory address stored in a pointer, you must **dereference** the pointer using the * operator (e.g. **cout << *number << endl;**).

Pointer Fun with Binky: a Stanford CS106 Throwback

- Nick Parlante has been teaching intro CS classes at Stanford for many years.
- In 1999, he created a stop-motion claymation video starring a character named Binky that has been a staple of explaining pointers in intro CS classes at Stanford ever since.


```
*y = 13;
```

Today: Using pointers in practice

Today: Using pointers in practice

How can we use pointers to organize non-contiguous memory on the heap?

Today: Using pointers in practice

How can we use pointers to organize
non-contiguous memory on the heap?

Not arrays!

Levels of abstraction

What is the interface for the user?

How is our data organized?

What stores our data?
(arrays, linked lists)

How is data represented electronically?
(RAM)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Computer Hardware

Levels of abstraction

What is the interface for the user?

Abstract Data Structures

How is our data organized?

Data Organization Strategies

What stores our data?
(arrays, linked lists)

Fundamental C++ Data Storage

Pointers move us across this boundary!

How is data represented electronically?
(RAM)

Computer Hardware

Levels of abstraction

What is the interface for the user?

How is our data organized?

What stores our data?
(**arrays**, **linked lists**)

These are built on top of pointers!

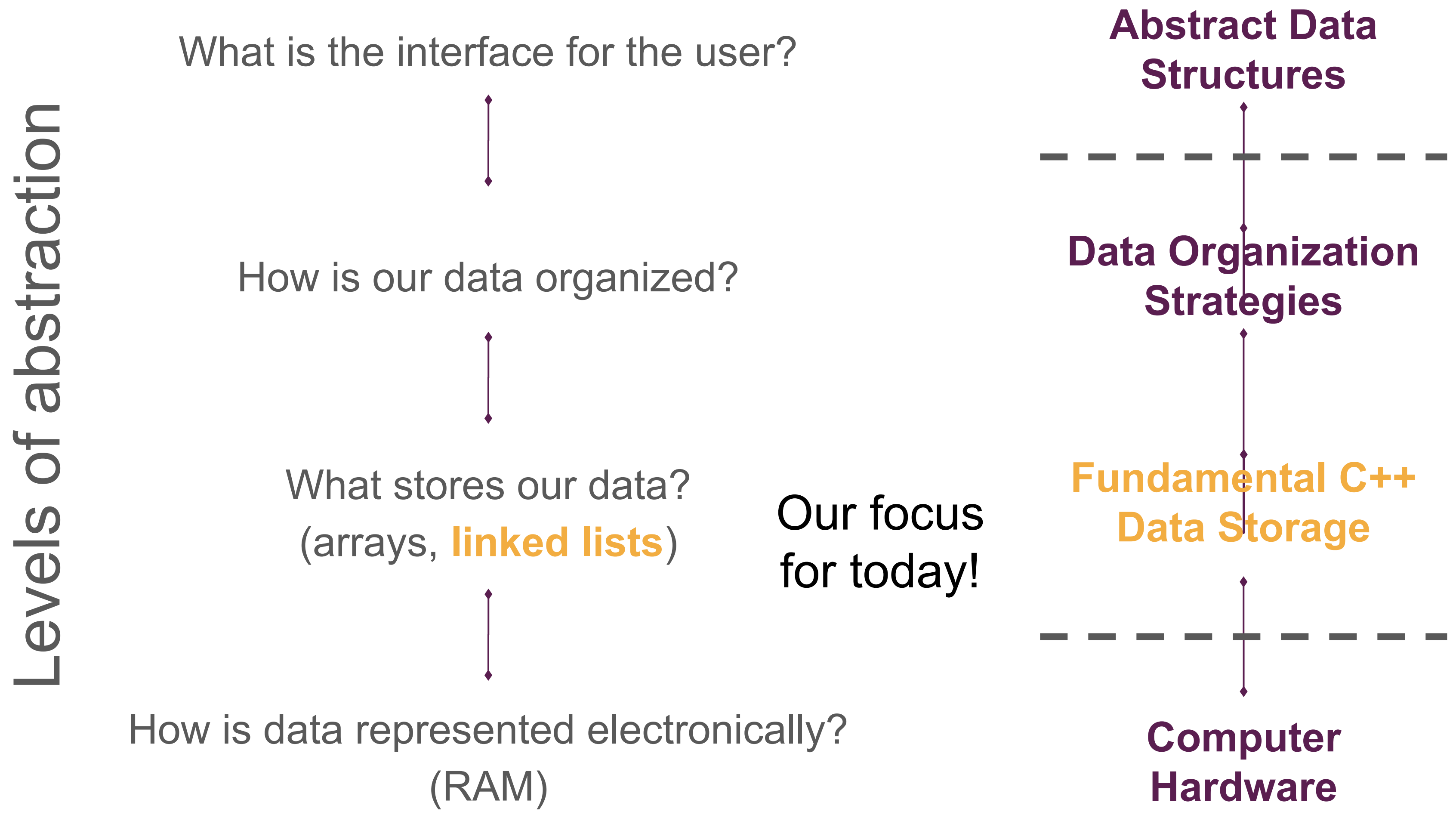
How is data represented electronically?
(RAM)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Computer Hardware


Levels of abstraction
What is the interface for the user?
How is our data organized?
What stores our data?
(arrays, linked lists)
How is data represented electronically?
(RAM)
Our focus
for today!
Abstract Data
Structures
Data Organization
Strategies
Fundamental C++
Data Storage
Computer
Hardware

What is a linked list?

What is a linked list?

- A linked list is a **chain of nodes**.

What is a 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

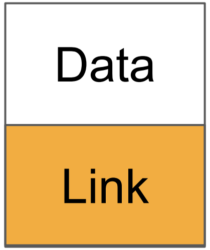
What is a linked list?

- A linked list is a **chain of nodes**.

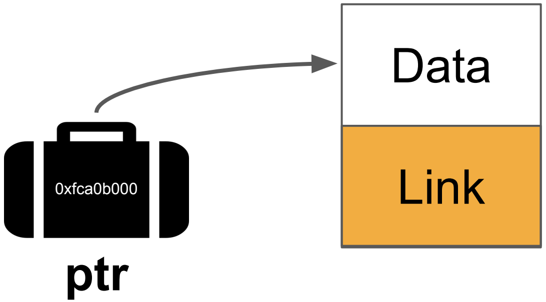
- Each **node** contains two pieces of information:
    - Some piece of data that is stored in the sequence
    - A link to the next node in the list

- We can traverse the list by starting at the first node and repeatedly following its link.

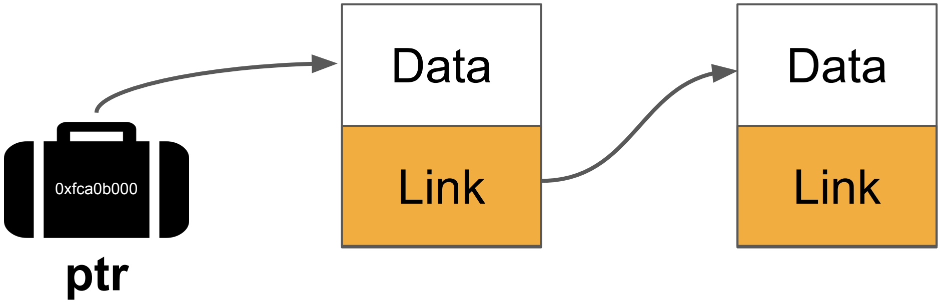
Node



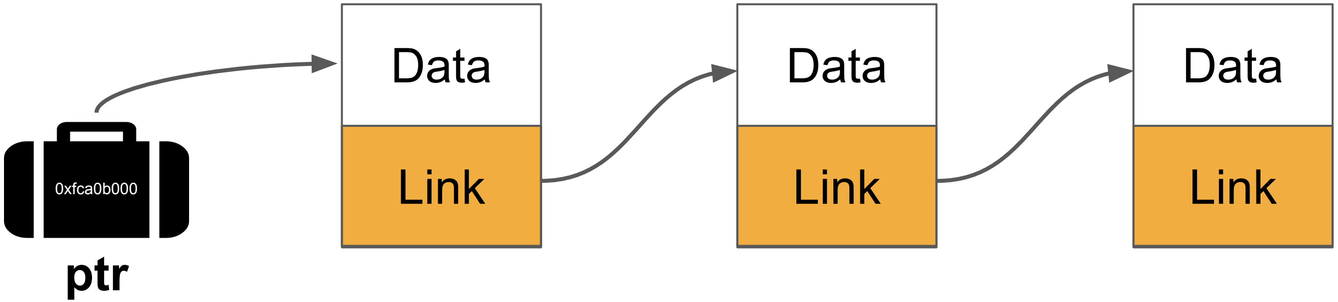
Pointer to a node



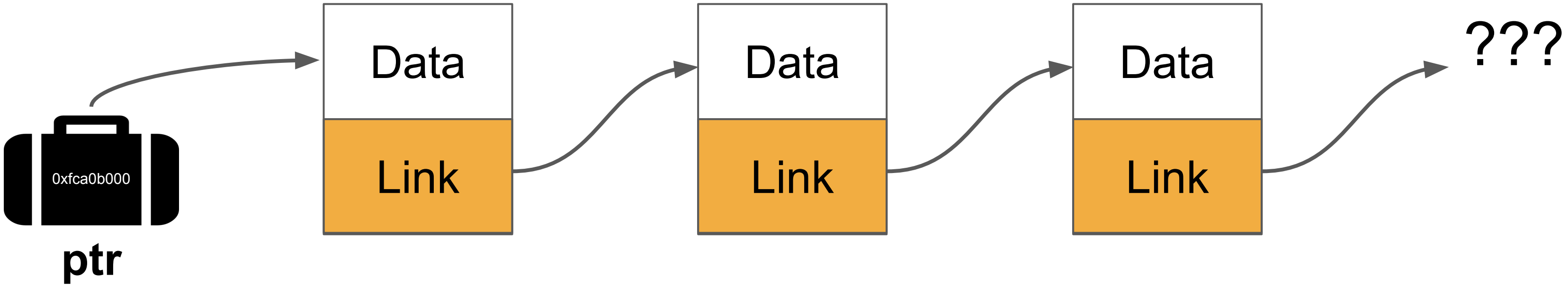
Pointer to a node that points to a node



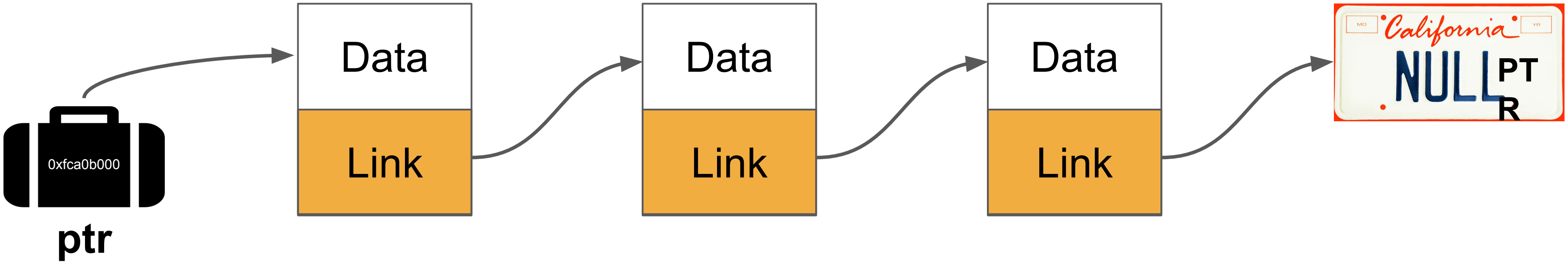
Pointer to a node that points to a node that points to a node



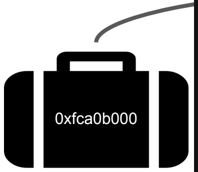
Pointer to a node that points to a node that points to a node



A linked list!



A link



ptr


r/todayilearned
Posted by u/shaka_sulu • 8h
TIL a California man got 'NULL' as a personalized license plate hoping that 'NULL' would confuse the computer system. Instead, when cops left the plate number info empty on a ticket or citation, the fine went to him. He got over $12k fines sent to him his first year.

Why use linked lists?

- More flexible than arrays!
  - Since they're not contiguous, they're easier to rearrange.

- We can efficiently splice new elements into the list or remove existing elements anywhere in the list. (We'll see how shortly!)

- We never have to do a massive copy step.

- Linked lists have many tradeoffs, and are not often the best data structure!

Linked lists in C++

The **Node** struct

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

The **Node** struct

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

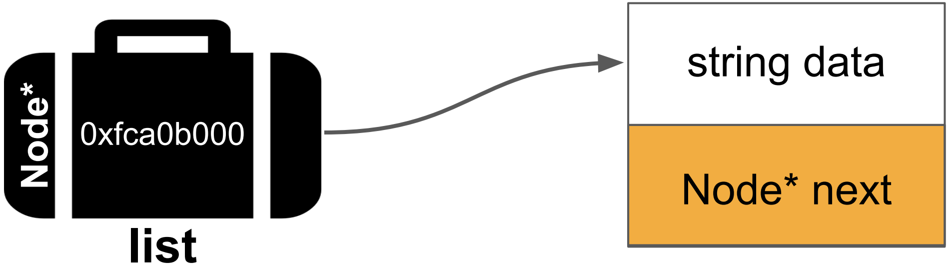
- The structure is defined recursively! (both the Node and the linked list itself)

The **Node** struct

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

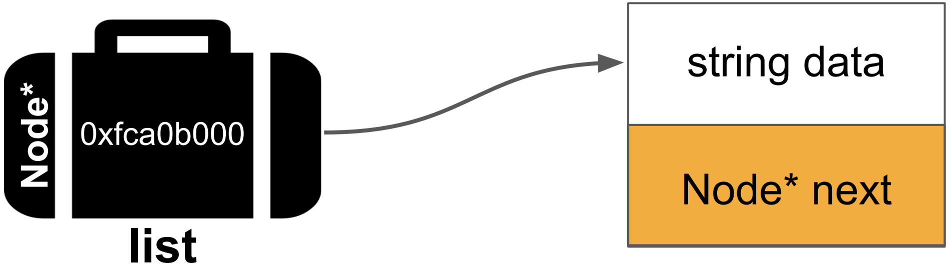
- The structure is defined recursively! (both the Node and the linked list itself)
- The compiler can handle the fact that in the definition of the **Node** there is a **Node*** because it knows it is simply a pointer.
    - (It would be impossible to recursively define the **Node** with an actual **Node** object inside the struct.)

Pointer to a node




```
Node* list = new Node;
```

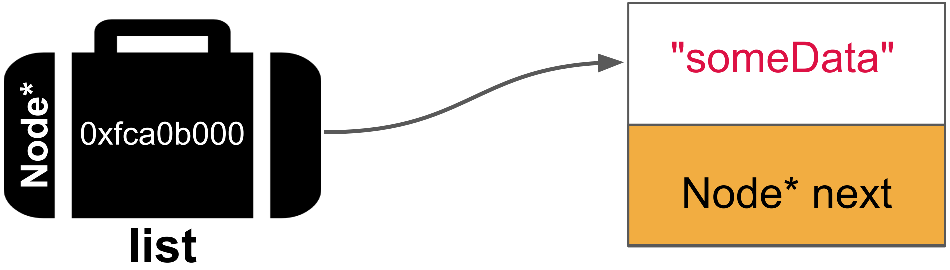
Pointer to a node



Node* list = **new** Node;

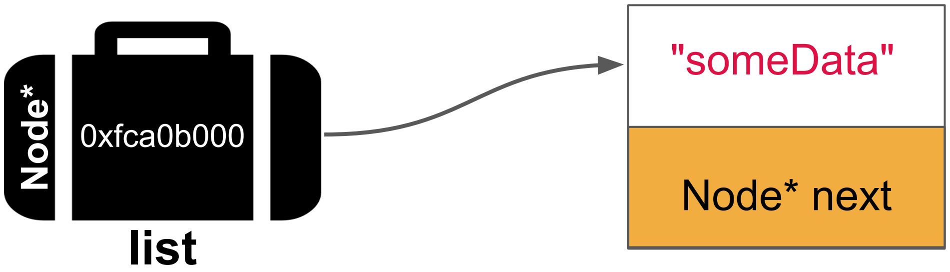
How do we update these values (i.e., the Node itself)?

Pointer to a node



Node* list = **new** Node;
(*list).data = "someData";

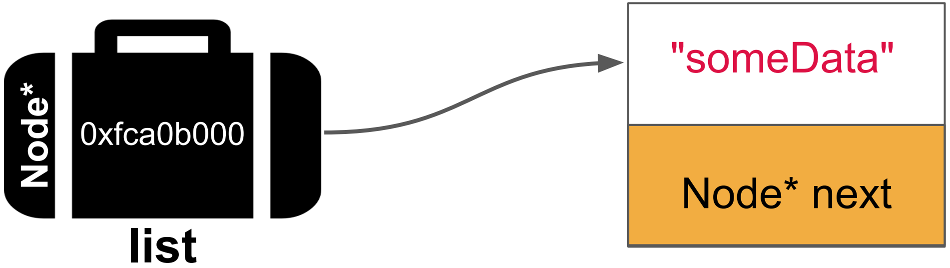
Pointer to a node




```
Node* list = new Node;
(*list).data = "someData";
```

Use * to dereference the pointer to get the Node struct.

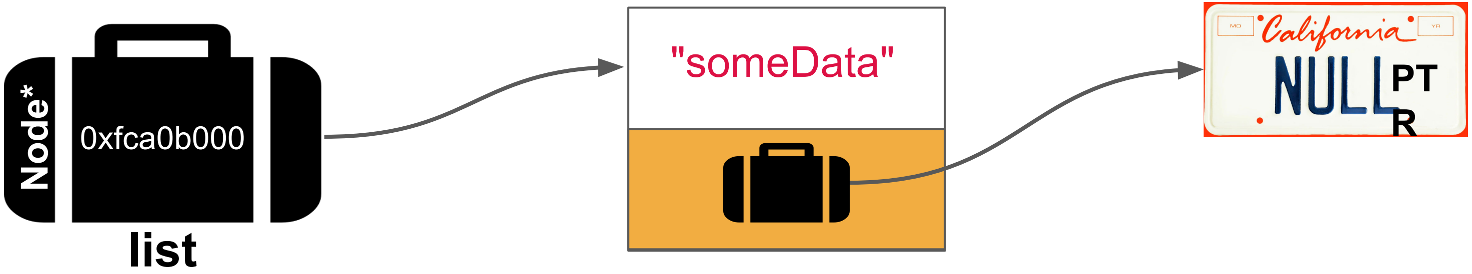
Pointer to a node




```
Node* list = new Node;
(*list).data = "someData";
```

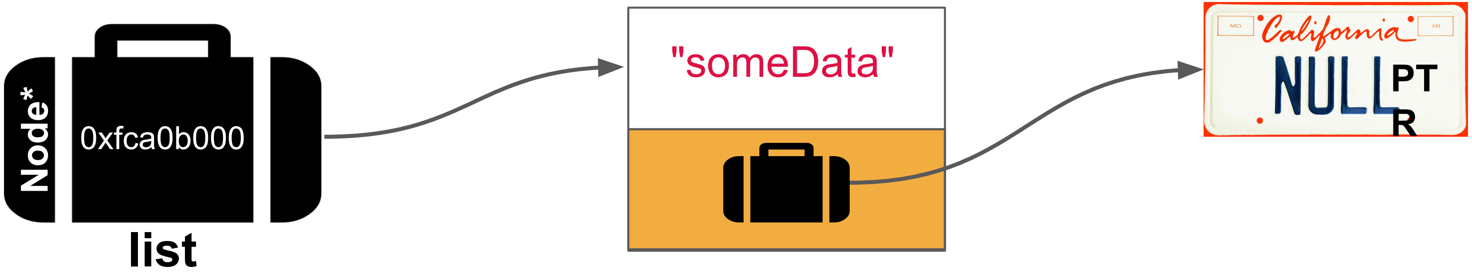
Use dot (.) notation to update the data field of the struct.

Pointer to a node




```
Node* list = new Node;
(*list).data = "someData";
(*list).next = nullptr;
```

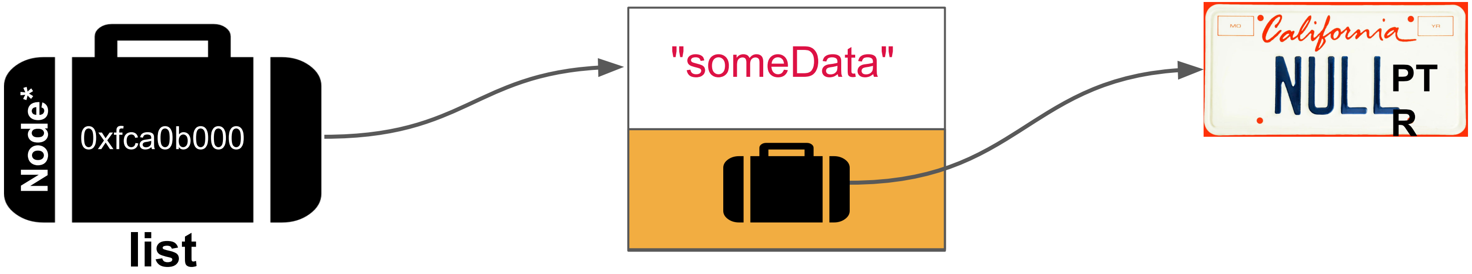
Pointer to a node




```
Node* list = new Node;
(*list).data = "someData";
(*list).next = nullptr;
```

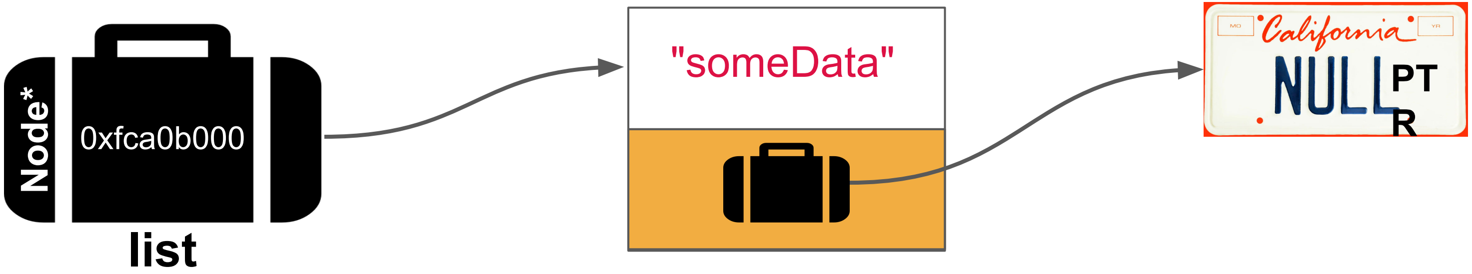
There's an easier way!

Pointer to a node




```
Node* list = new Node;
list->data = "someData";
list->next = nullptr;
```

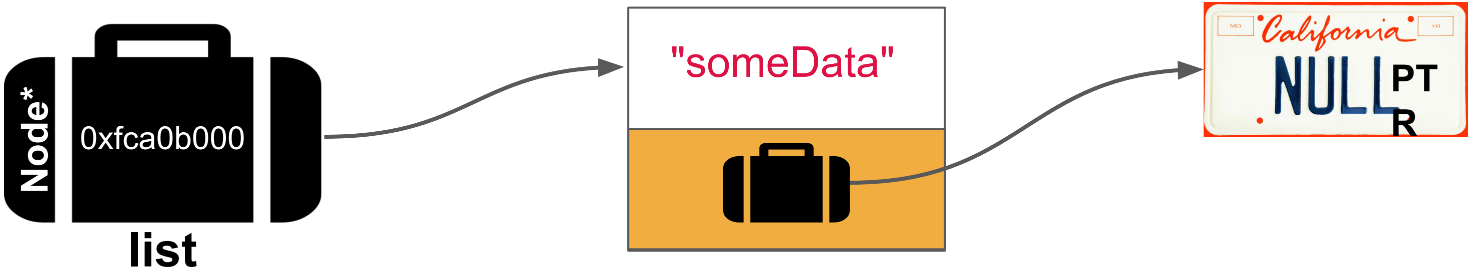
Pointer to a node




```
Node* list = new Node;
list->data = "someData";
list->next = nullptr;
```

The arrow notation (**->**) dereferences AND accesses the field for pointers that point to structs specifically.

Pointer to a node



Node* list = **new** Node;

(*list).data = "someData";

(*list).next = nullptr;

Node* list = **new** Node;

list->data = "someData";

list->next = nullptr;

Announcements

Announcements

- Assignment 4 is due tomorrow **Tuesday, July 27 at 11:59pm PDT.**
    - As a reminder, LaIR is happening today from **5-7pm PDT and Tuesday 7-9pm PDT.** If you've encountered any bugs in A4, we encourage you to come to LaIR tonight or tomorrow!
    - When you submit A4, you'll be redirected to our **Mid-Quarter Evaluation**. This is a comprehensive form that will ask for your feedback about CS106B and the course staff.

- Diagnostic regrade requests are due **today at 11:59pm PDT.**

How do we manipulate linked lists?

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

Implementing a Stack

Note: You could do this with an array! This is just for the sake of getting practice with linked lists.

Stack as a linked list

- We’ll keep a pointer **Node* top** that points to the “top” element in our stack.
  - This member var will get initialized to **nullptr** when our stack is empty!

Stack as a linked list

- We'll keep a pointer **Node* top** that points to the "top" element in our stack.
    - This member var will get initialized to **nullptr** when our stack is empty!

- Our linked list nodes will be connected from the top to the bottom of our stack.

Stack as a linked list

- We'll keep a pointer **Node* top** that points to the "top" element in our stack.
  - This member var will get initialized to **nullptr** when our stack is empty!

- Our linked list nodes will be connected from the top to the bottom of our stack.

- Our stack will specifically hold integers, so our **Node** struct will hold an **int** type for our **data** field:

```
struct Node {
    int data;
    Node* next;
}
```

Three Stack operations

- **push()**
- **pop()**
- Destructor

Three Stack operations

- **push()**
- **pop()**
- Destructor

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion (at the front)**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

push()

- Suppose we have the following Stack we want to push to:

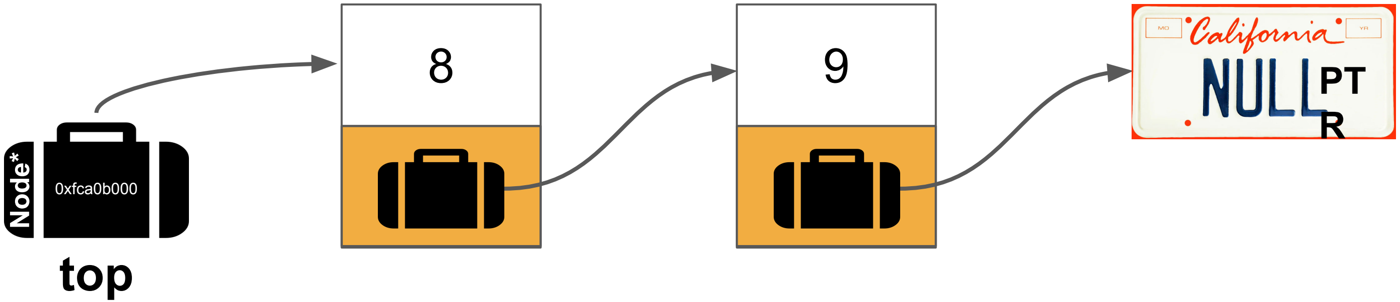
```
Stack myStack = {9, 8}; // 8 is at the "top" of the stack
myStack.push(7); // we want the result to be {9, 8, 7}
```

push()

- Suppose we have the following Stack we want to push to:

  Stack myStack = {9, 8}; *// 8 is at the "top" of the stack*
  myStack.push(7); *// we want the result to be {9, 8, 7}*

How our linked list starts:

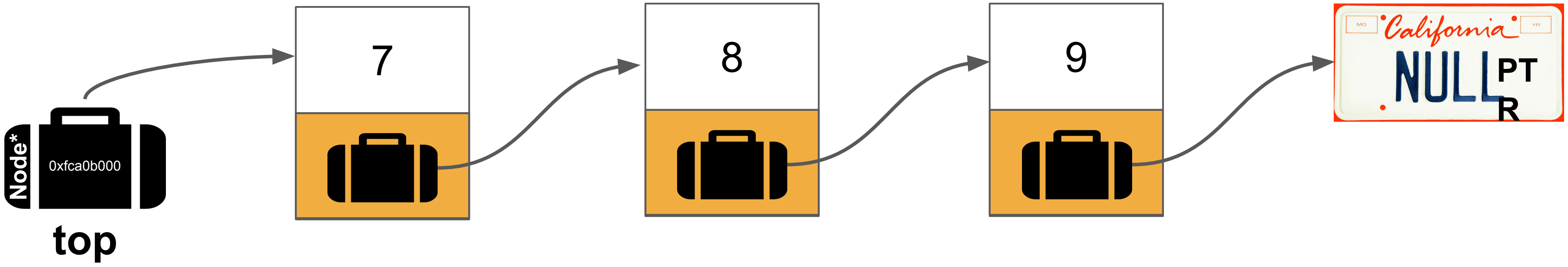


push()

- Suppose we have the following Stack we want to push to:

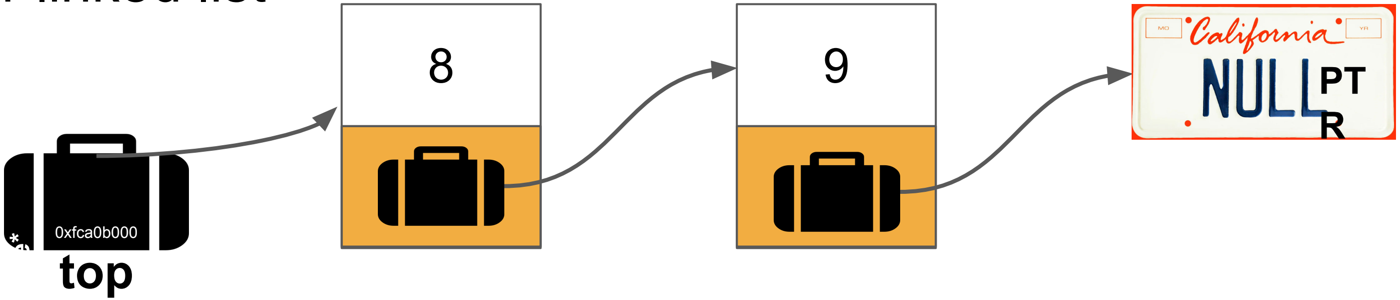
  Stack myStack = {9, 8}; *// 8 is at the "top" of the stack*
  myStack.push(7); *// we want the result to be {9, 8, 7}*

Goal:

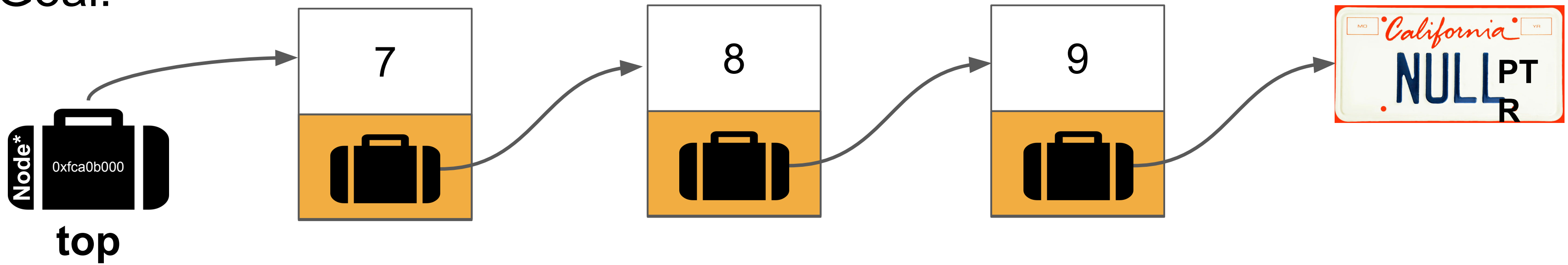


push()

How our linked list starts:

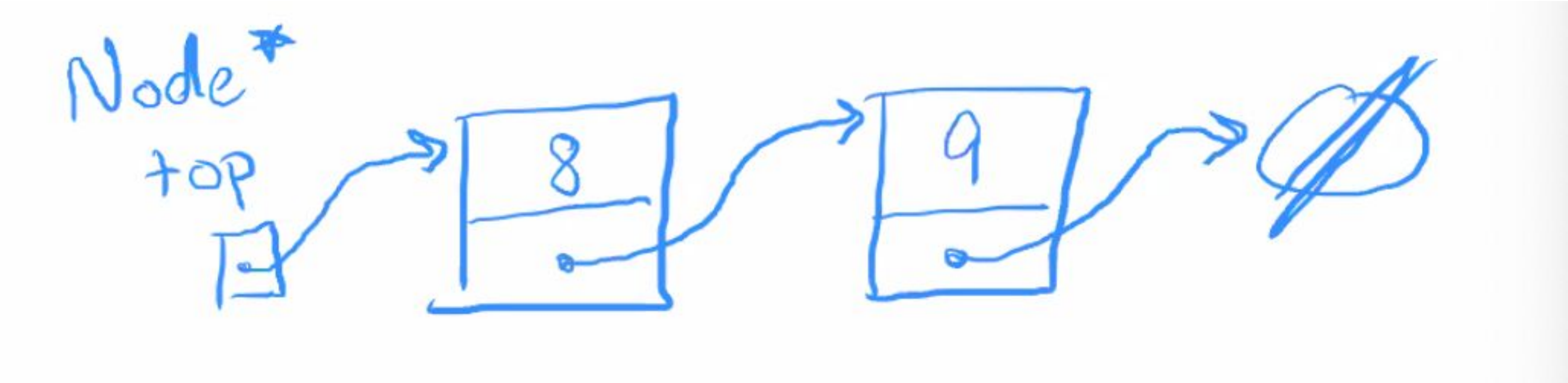


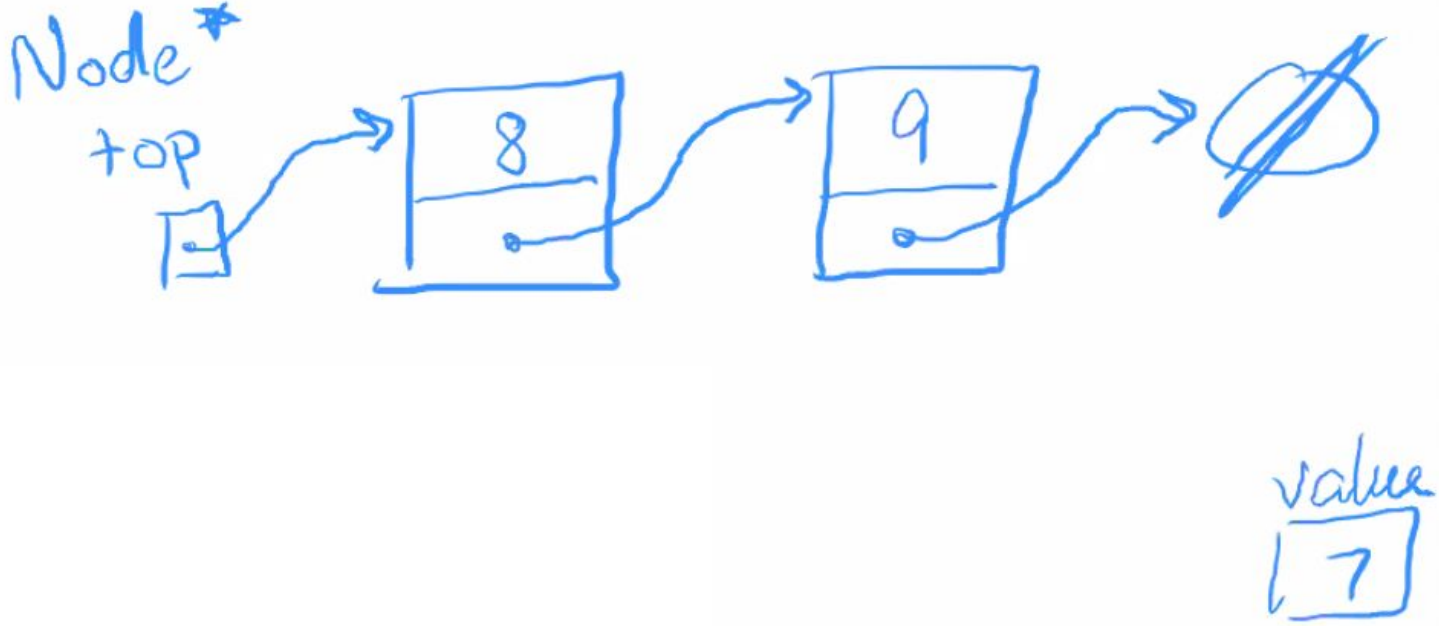
Goal:

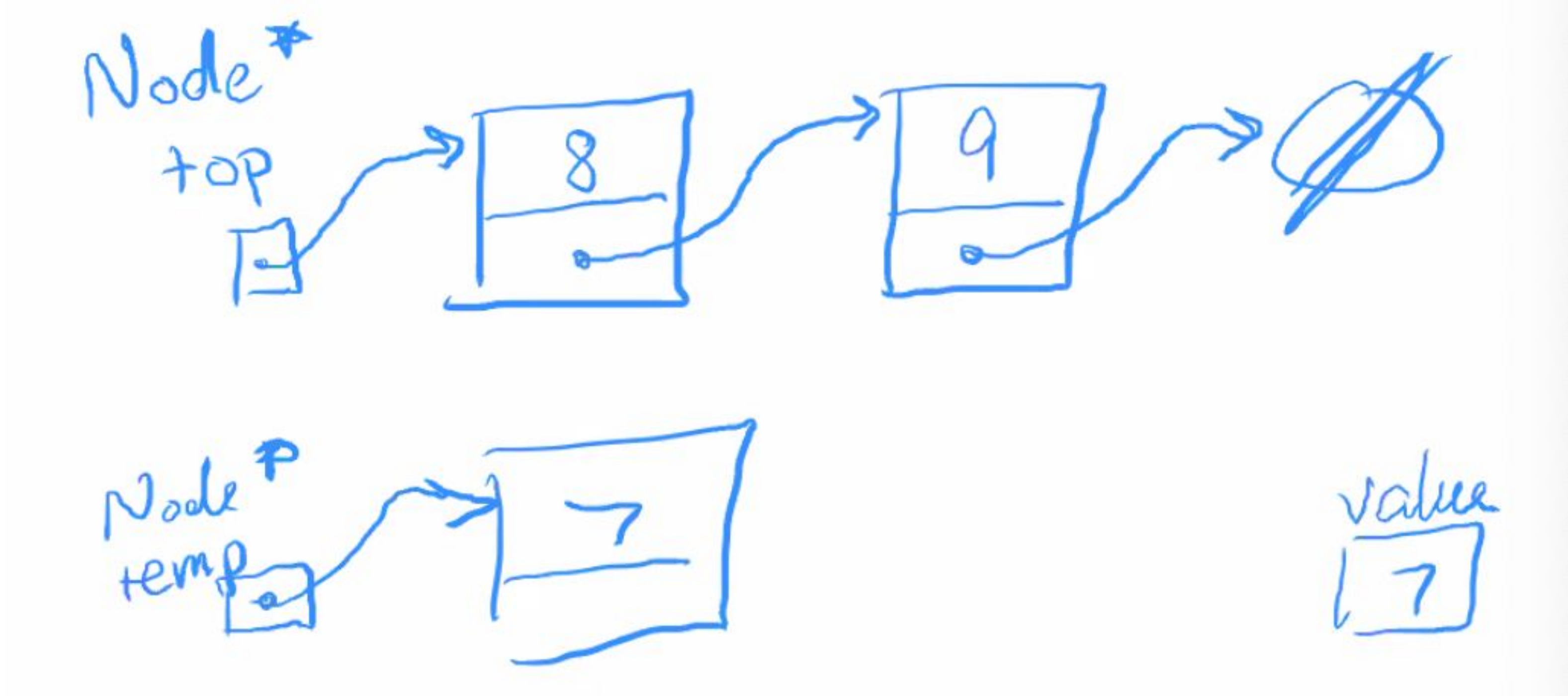


Let’s code **push()**!

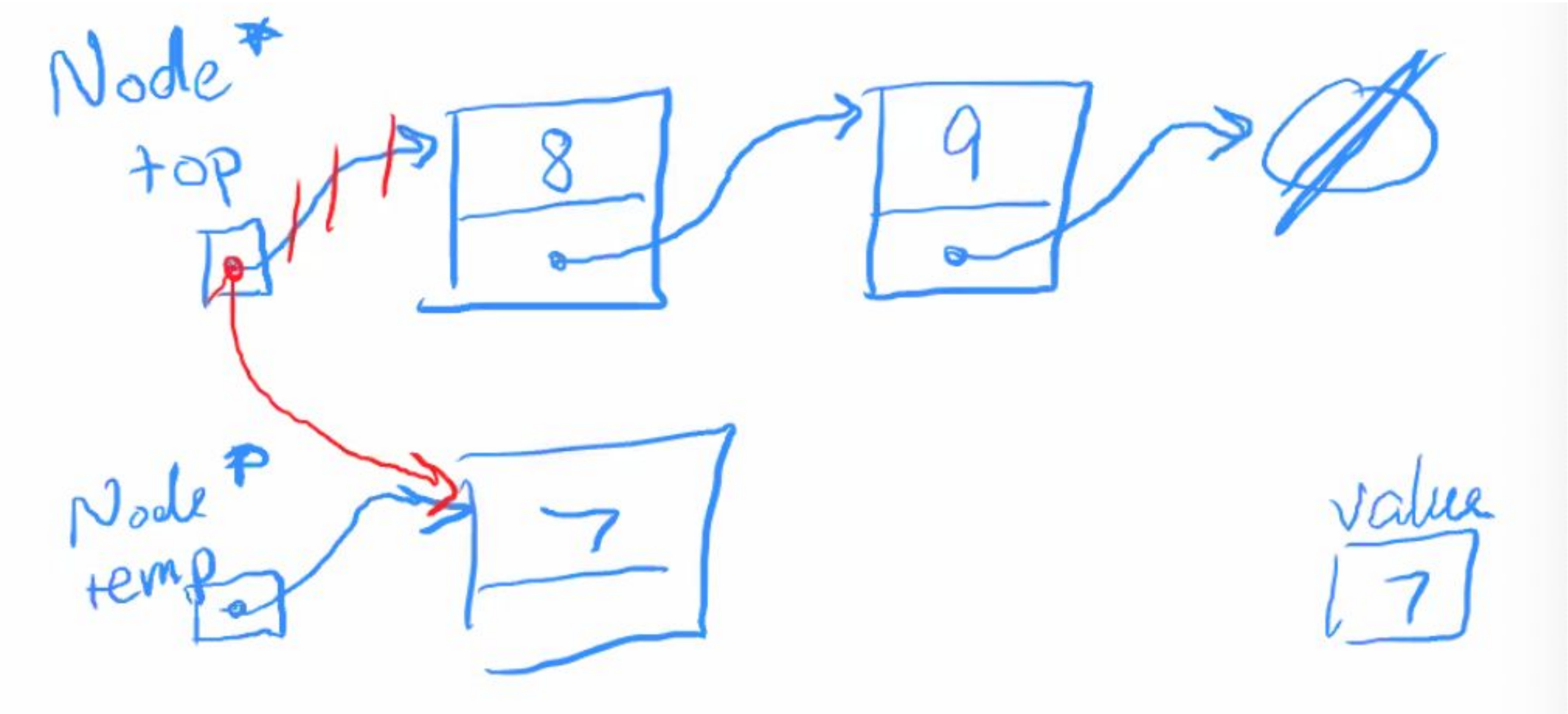
Initial State (beginning of **push()** function)



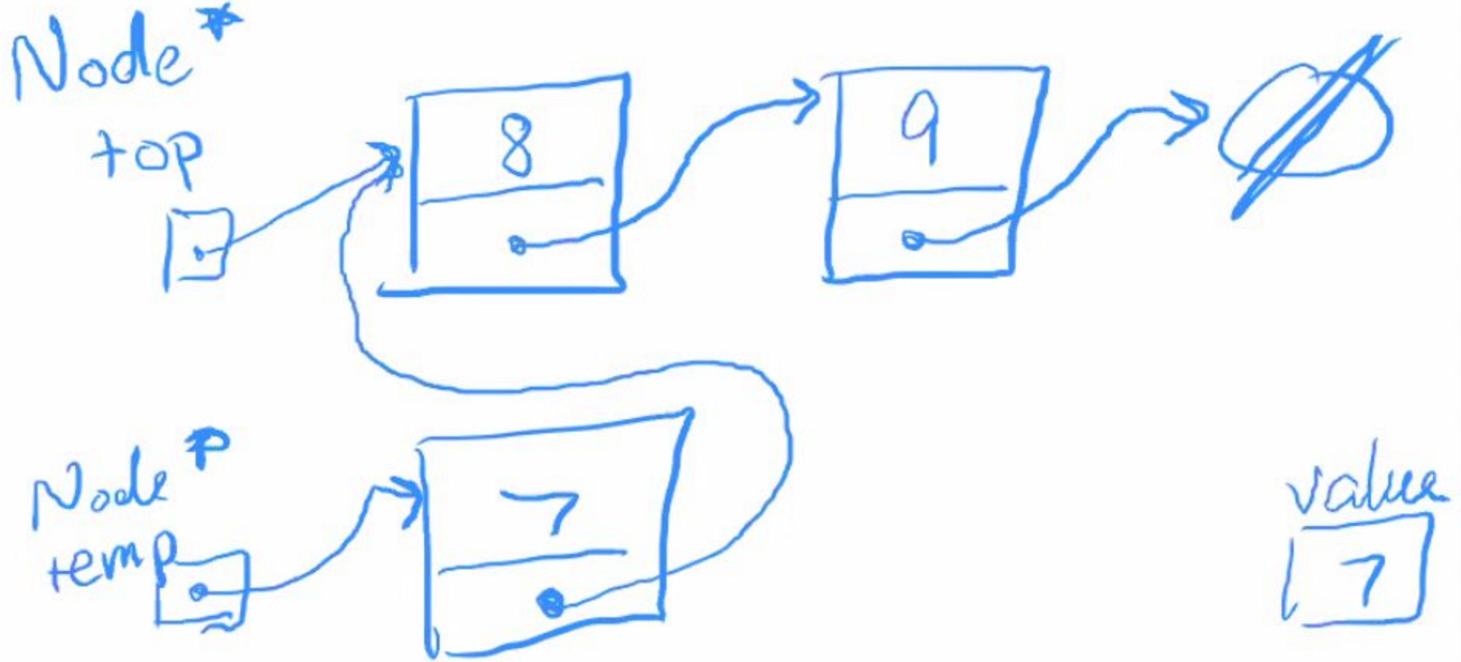

Node*
top
8
9
value
7



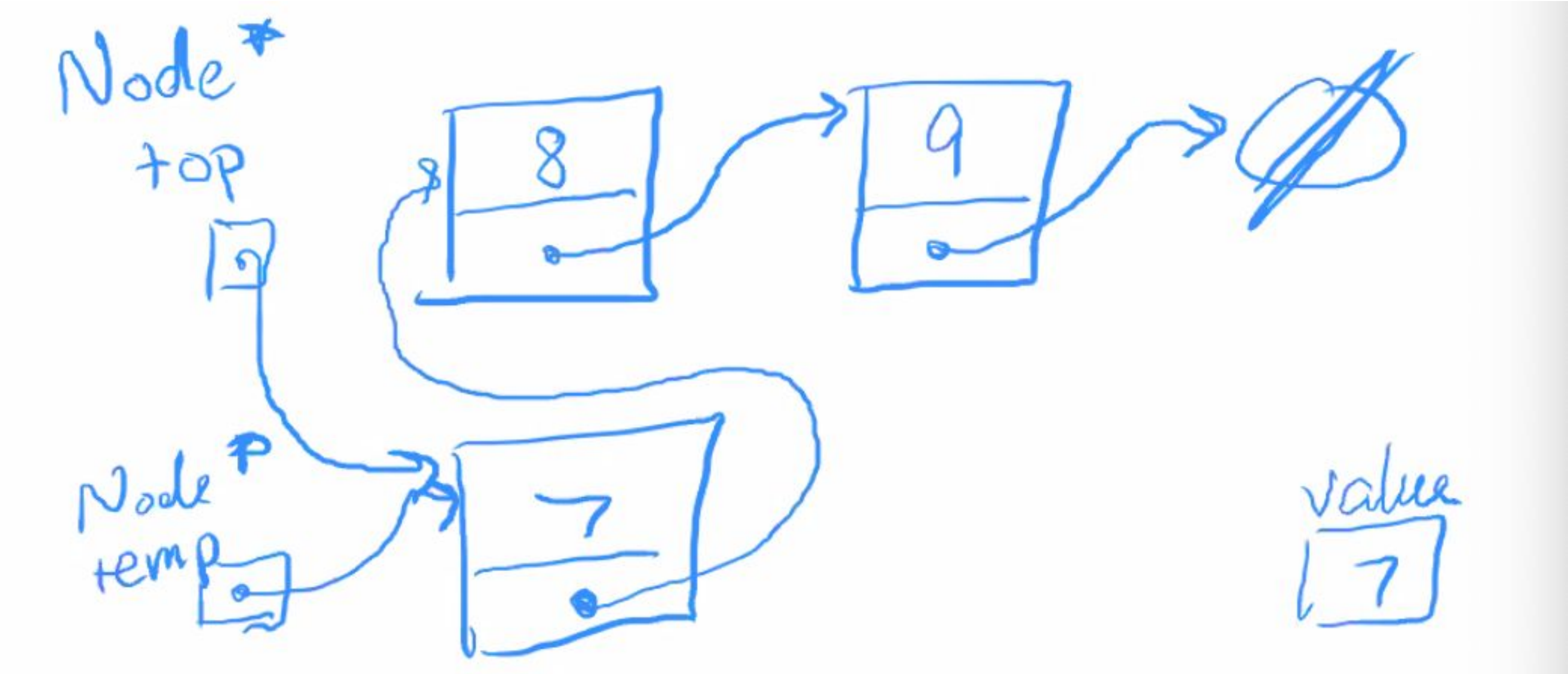

```
Node *temp = new Node;
temp->data = 7;
```




```
Node *temp = new Node;
temp->data = 7;
top = temp; // INCORRECT
```



Node *temp = new Node;
temp->data = 7;
temp->next = top;




```
Node *temp = new Node;
temp->data = 7;
temp->next = top;
top = temp;
```

Three Stack operations

- **push()**
- **pop()**
- Destructor

Common linked lists operations

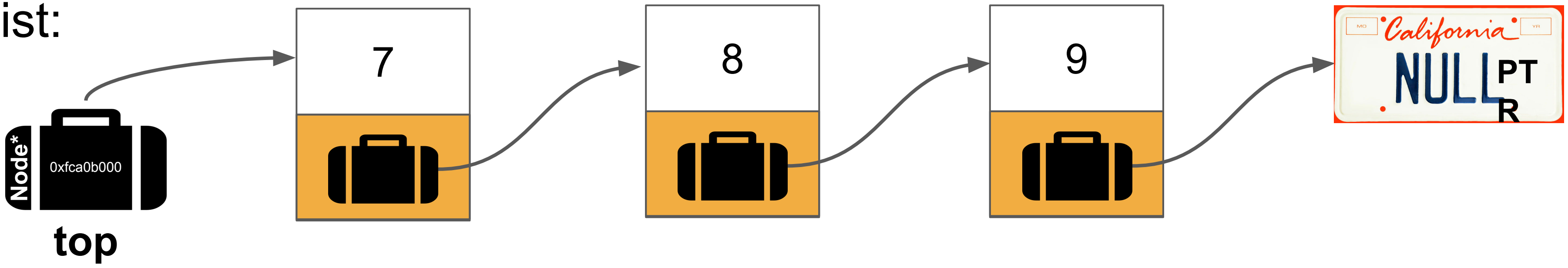
- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

pop()

- Now we want to remove the top value:

```
...
myStack.pop(); // we want the result to be {9, 8}
```

Starting state of the list:

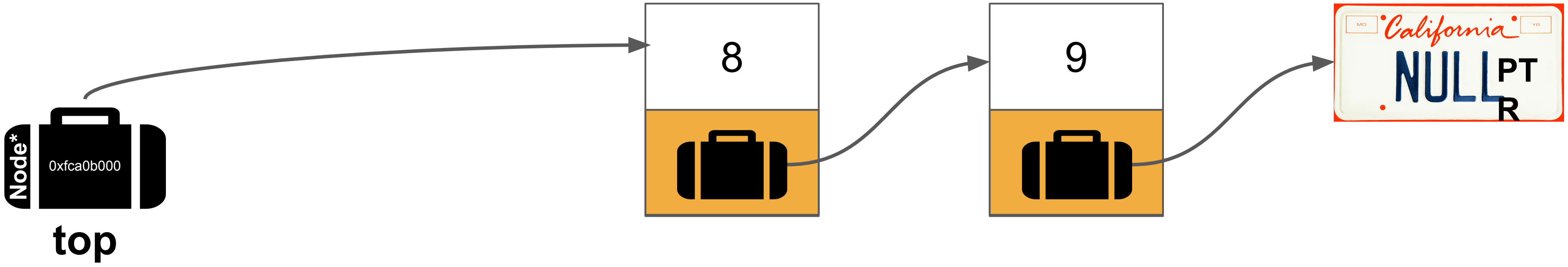


pop()

- Now we want to remove the top value:

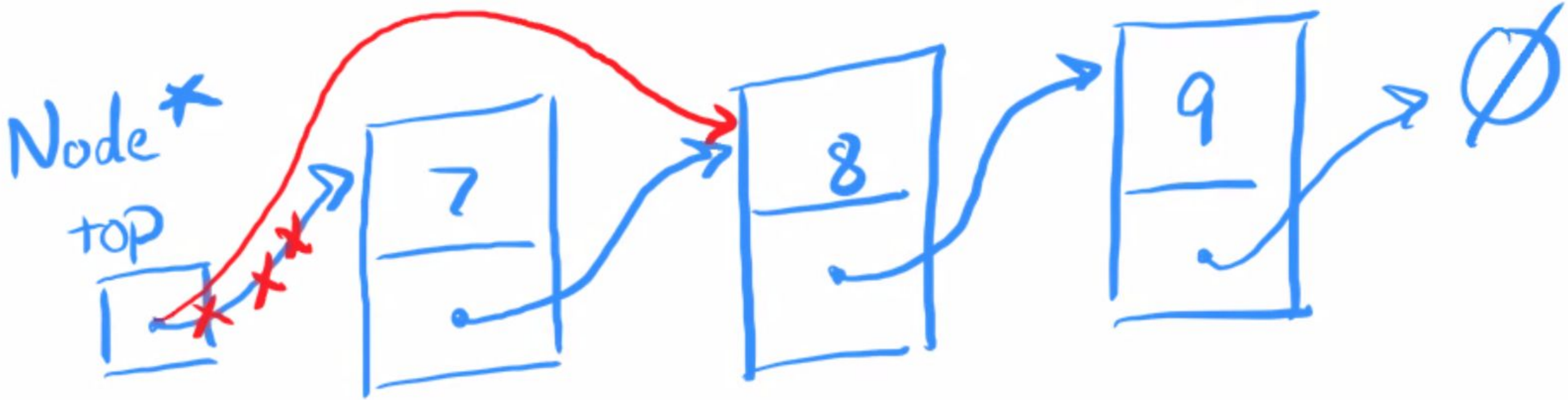
```
...
myStack.pop(); // we want the result to be {9, 8}
```

Goal:

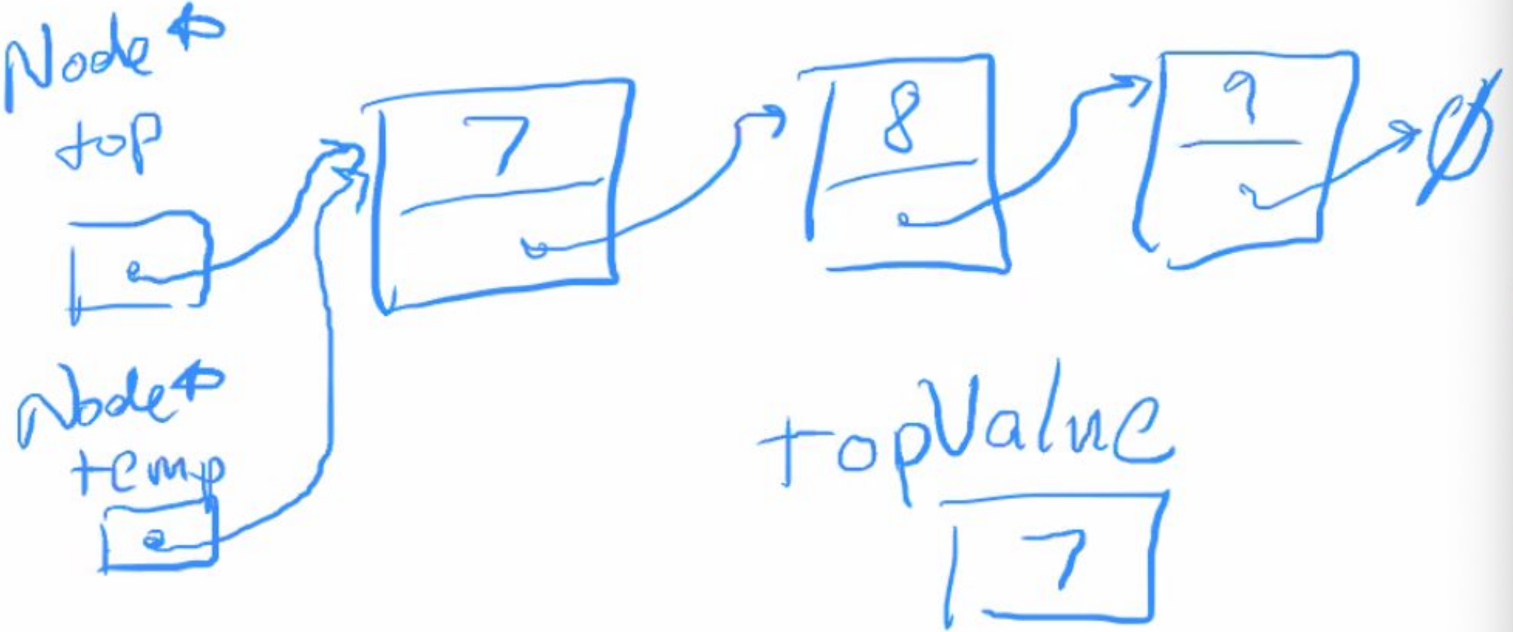


Let’s code **pop()**!

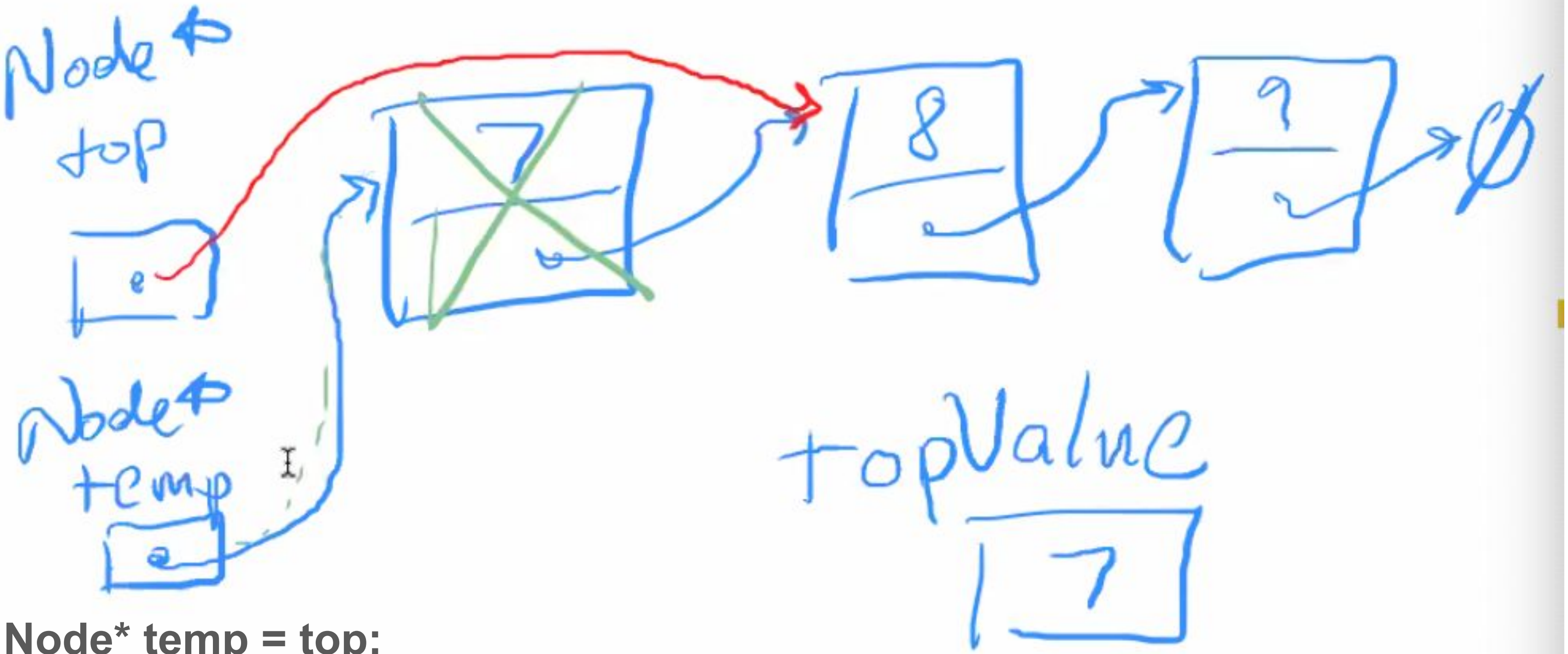
Initial State (beginning of **pop()** function)



top = top->next; // INCORRECT



Node* temp = top;




```
Node* temp = top;
top = top->next;
delete temp;
```

Three Stack operations

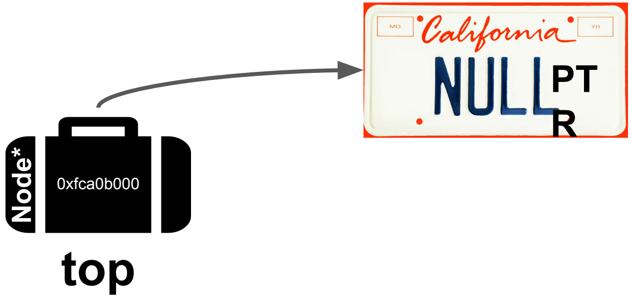
- **push()**
- **pop()**
- **Destructor**

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

Destructor

- We have to make sure we delete all of the **Nodes**.
- The **top** pointer should be **nullptr** when we’re done.



Let’s code the destructor!

Summary

Linked lists summary

- Linked lists are chains of Node structs, which are connected by pointers.
  - Since the memory is not contiguous, they allow for fast rewiring between nodes (without moving all the other Nodes like an array might).

Linked lists summary

- Linked lists are chains of Node structs, which are connected by pointers.
  - Since the memory is not contiguous, they allow for fast rewiring between nodes (without moving all the other Nodes like an array might).

- Common traversal strategy
  - While loop with a pointer that starts at the front of your list
  - Inside the while loop, reassign the pointer to the next node

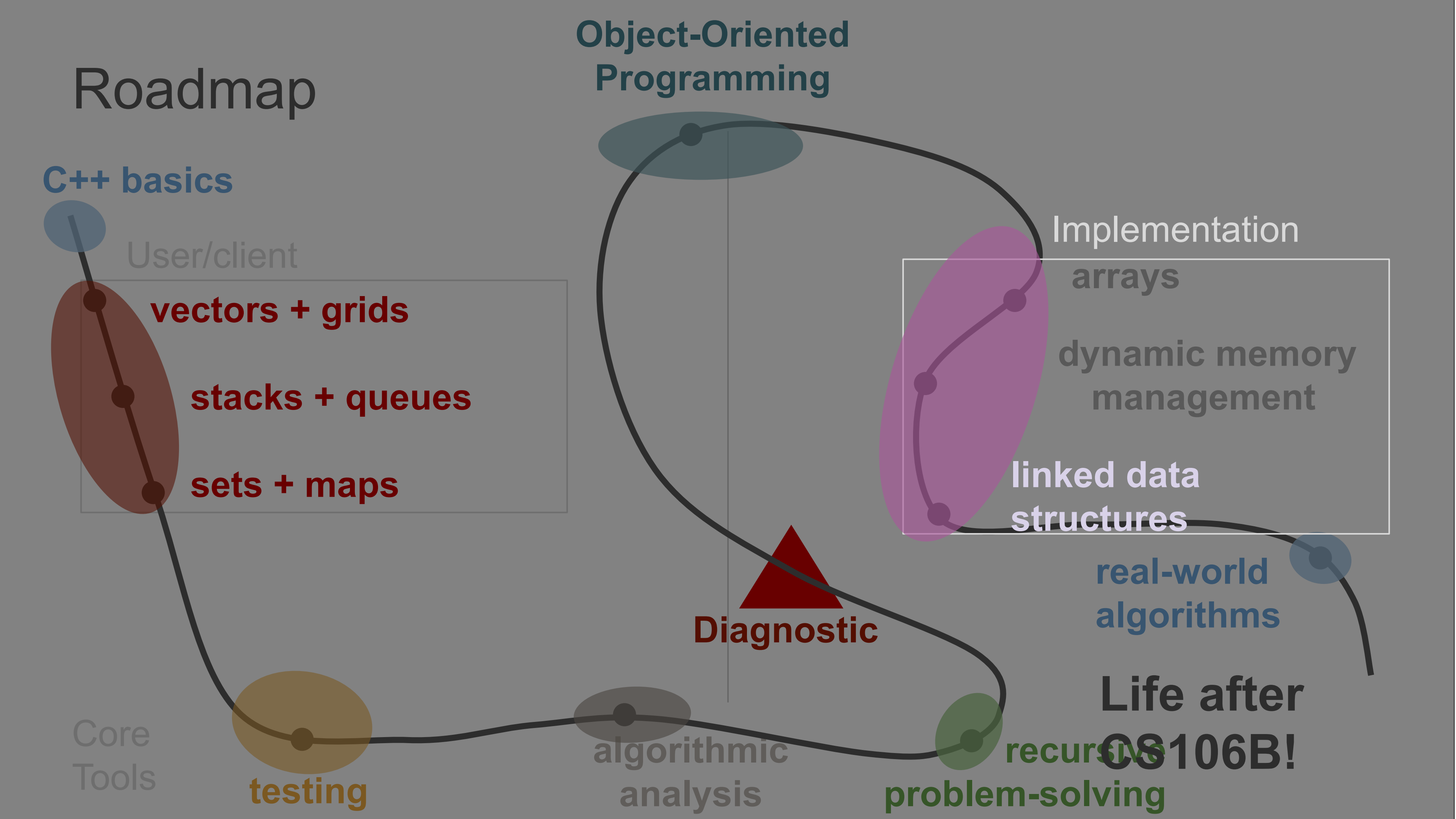
Linked lists summary

- Linked lists are chains of Node structs, which are connected by pointers.
    - Since the memory is not contiguous, they allow for fast rewiring between nodes (without moving all the other Nodes like an array might).

- Common traversal strategy
    - While loop with a pointer that starts at the front of your list
    - Inside the while loop, reassign the pointer to the next node

- Common bugs
    - Be careful about the order in which you delete and rewire pointers!
    - It’s easy to end up with dangling pointers or memory leaks (memory that hasn’t been deallocated but that you not longer have a pointer to)

What’s next?



Roadmap
Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

More on linked lists!