

Linked Lists

Has learning about the abstraction boundary made you think differently about how any apps/programs on your computer work?

(put your answers the chat)



Roadmap

C++ basics

User/client

vectors + grids

stacks + queues

sets + maps

Object-Oriented
Programming

Implementation

arrays

dynamic memory
management

linked data structures

real-world
algorithms

Life after CS106B!

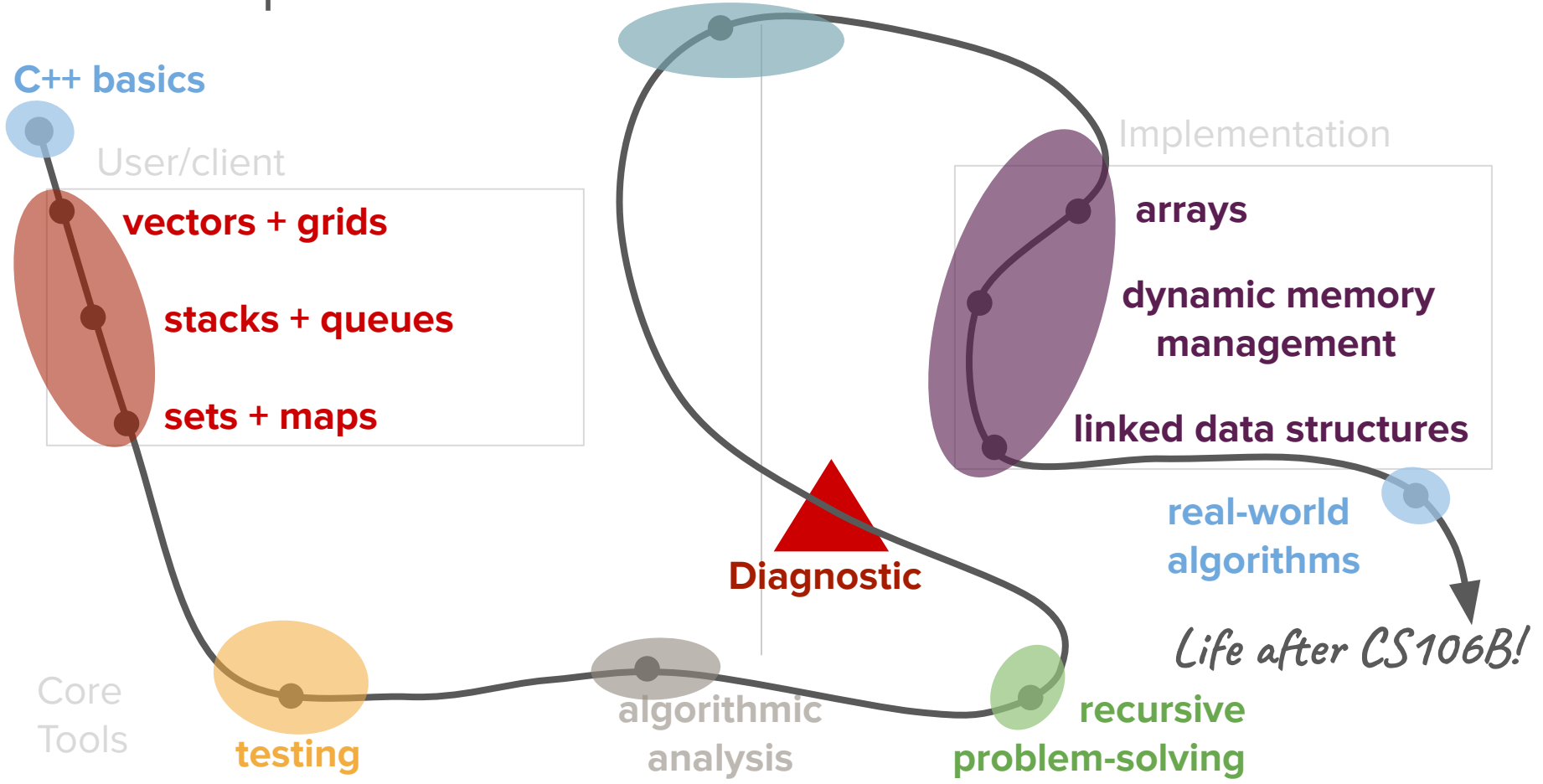
Core
Tools

testing

algorithmic
analysis

recursive
problem-solving

Diagnostic



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

Life after CS106B!

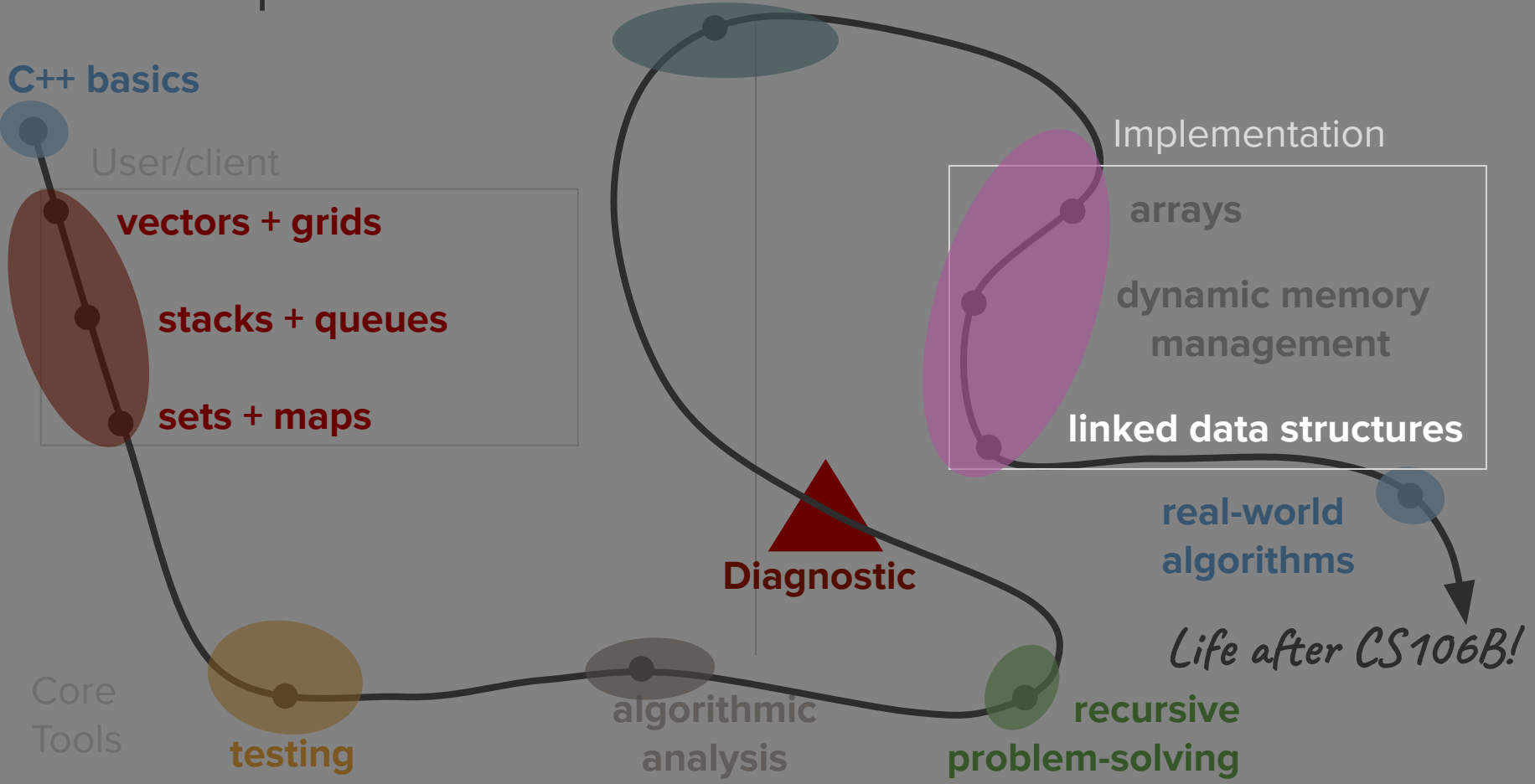
Core Tools

testing

algorithmic analysis

recursive problem-solving

Diagnostic



Today's question

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

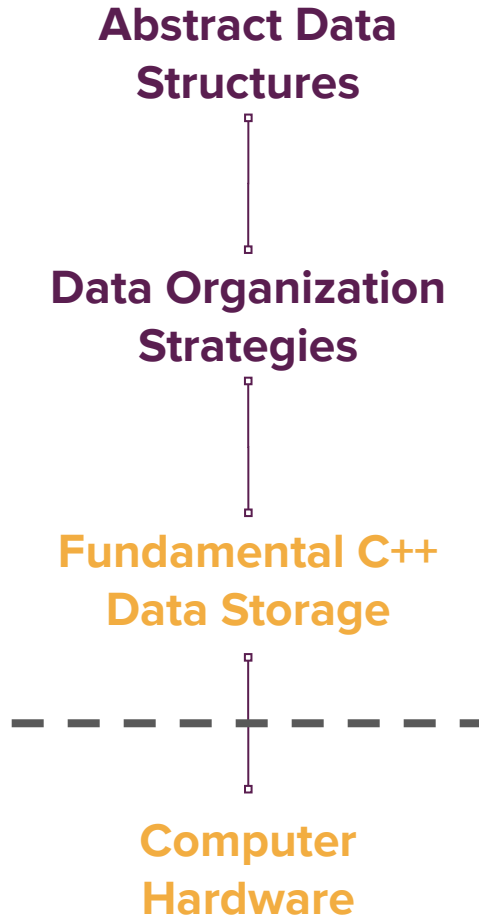
Today's topics

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

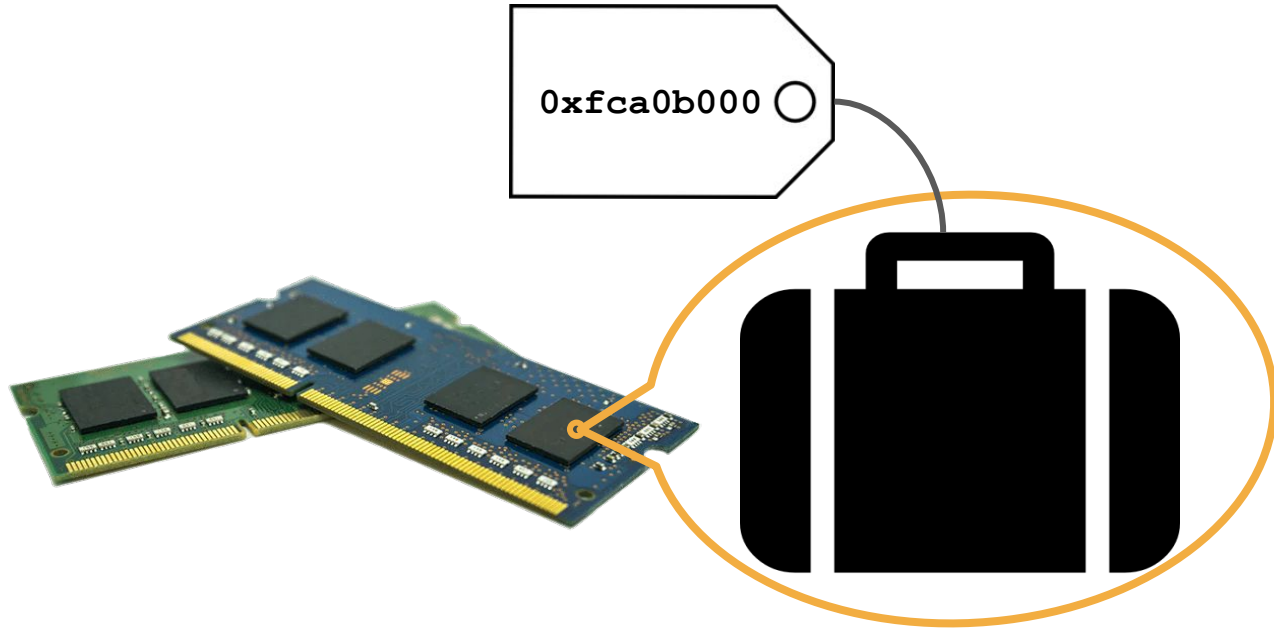
Review

[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;`).

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;`).



`*x = 42;`

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?

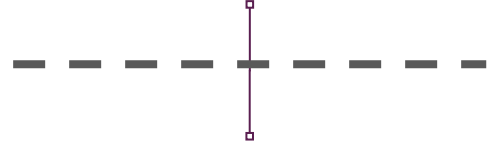


What stores our data?
(arrays, linked lists)



How is data represented electronically?
(RAM)

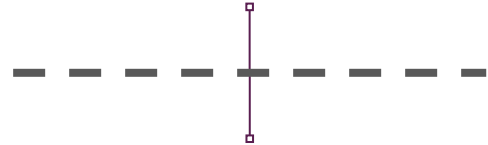
Abstract Data Structures



Data Organization Strategies



Fundamental C++ Data Storage



Computer Hardware

Levels of abstraction

What is the interface for the user?



How is our data organized?

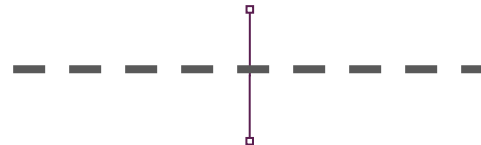


What stores our data?
(arrays, linked lists)



How is data represented electronically?
(RAM)

Abstract Data Structures

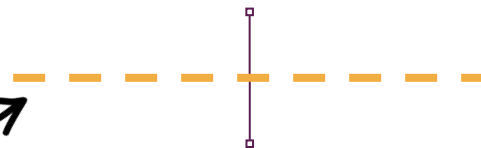


Data Organization Strategies



Fundamental C++ Data Storage

Pointers move us across this boundary!



Computer Hardware

Levels of abstraction

What is the interface for the user?



How is our data organized?



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

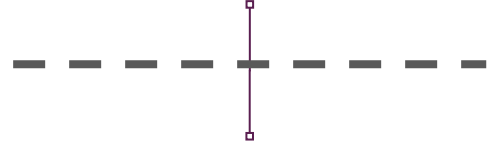


*These are built
on top of
pointers!*



How is data represented electronically?
(RAM)

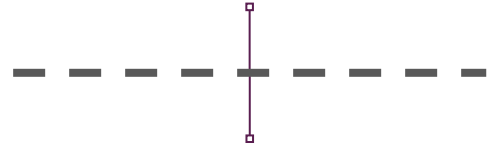
**Abstract Data
Structures**



**Data Organization
Strategies**



**Fundamental C++
Data Storage**



**Computer
Hardware**

Levels of abstraction

What is the interface for the user?



How is our data organized?



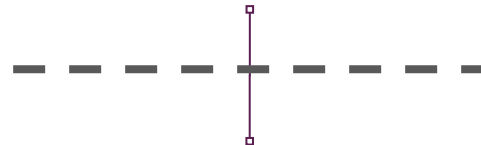
What stores our data?
(arrays, **linked lists**)



How is data represented electronically?
(RAM)

Our focus for today!

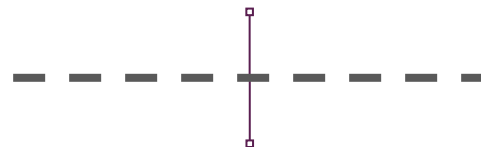
Abstract Data Structures



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**.

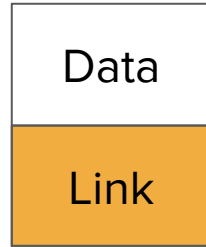
What is a linked list?

- 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

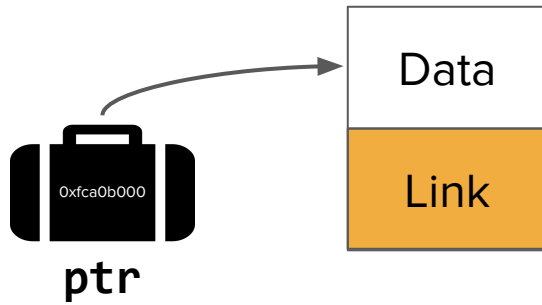
What is a linked list?

- A linked list is a **chain of nodes**.
- Each **node** contains two pieces of information:
 - Some piece of data that is stored in the list
 - 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.

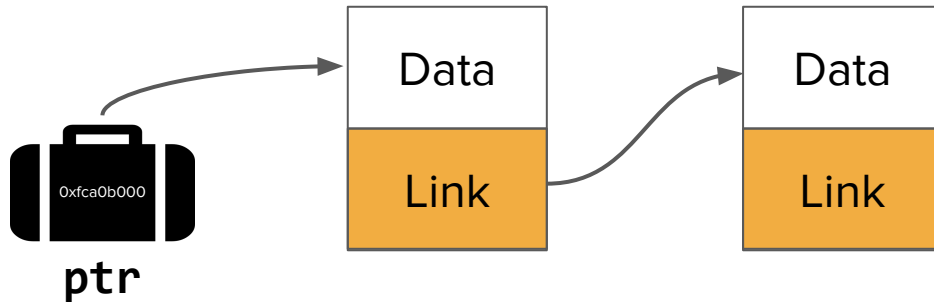
Node



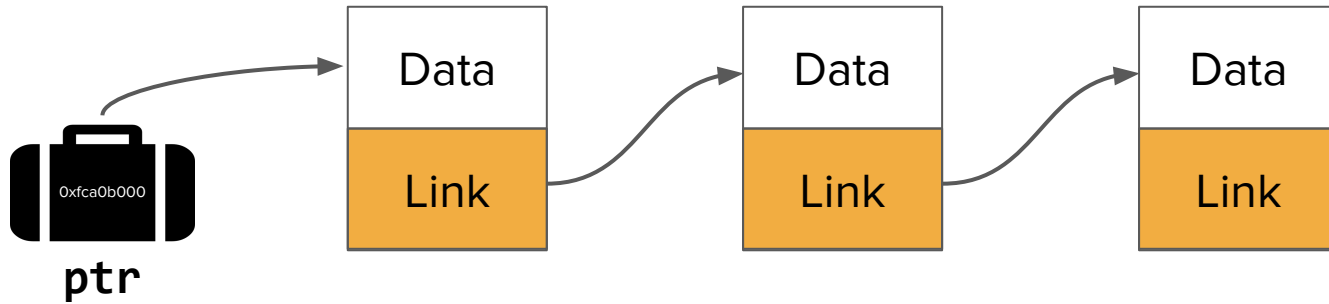
Pointer to a node



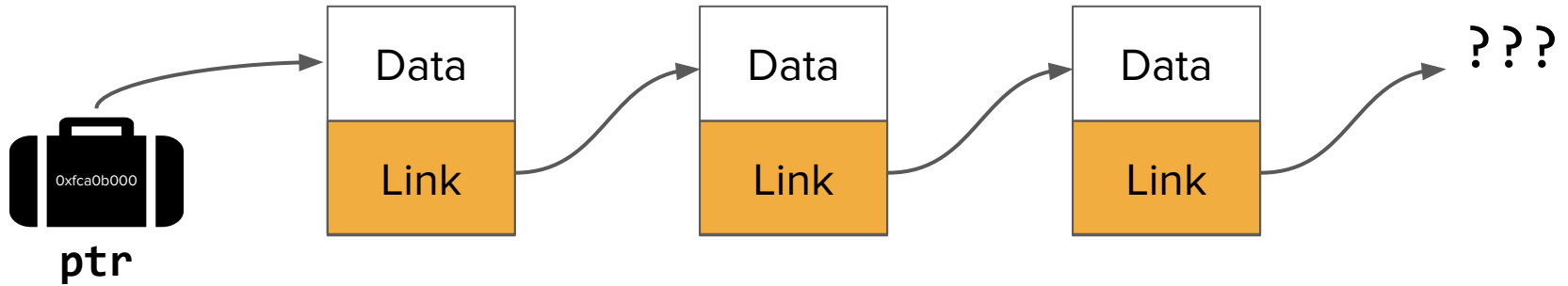
Pointer to a node that points to a node



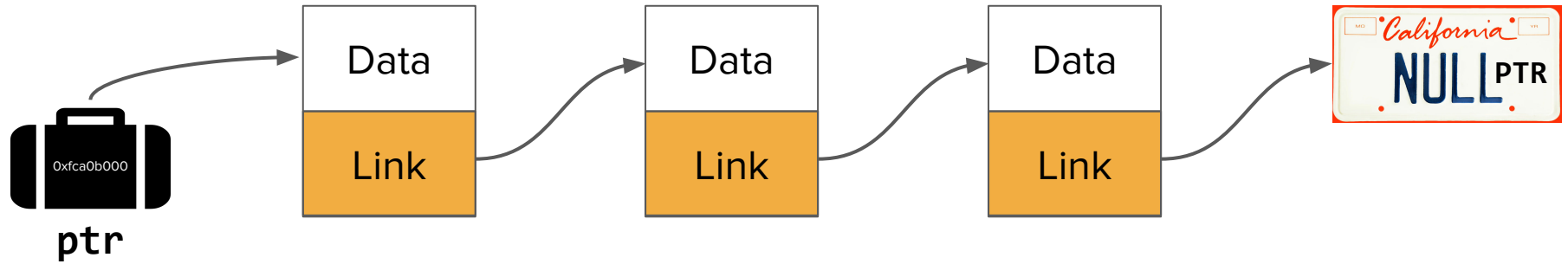
Pointer to a node that points to a node that points to a node



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.



↑ 15.8k



355



Share



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.
- But linked lists still have many tradeoffs and are not always 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;  
}
```

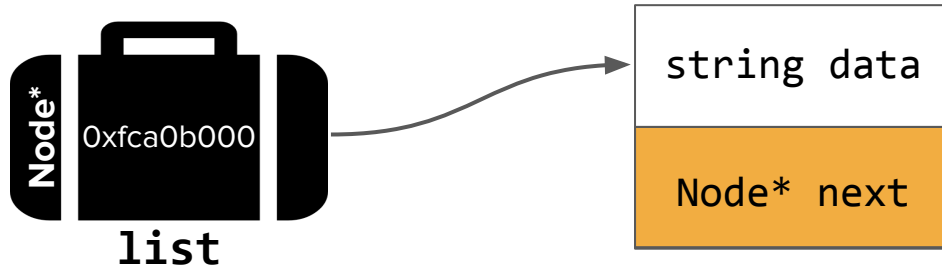
- The structure is defined recursively! (both the Node and the linked list itself)

The **Node** struct

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

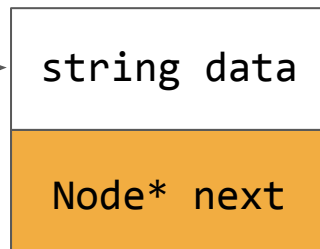
- The structure is defined recursively! (both the **Node** and the linked list itself)
- The compiler can handle the fact that in the definition of the **Node** there is a **Node*** because it knows it is simply a pointer.
 - (It would be impossible to recursively define the **Node** with an actual **Node**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

Pointer to a node



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

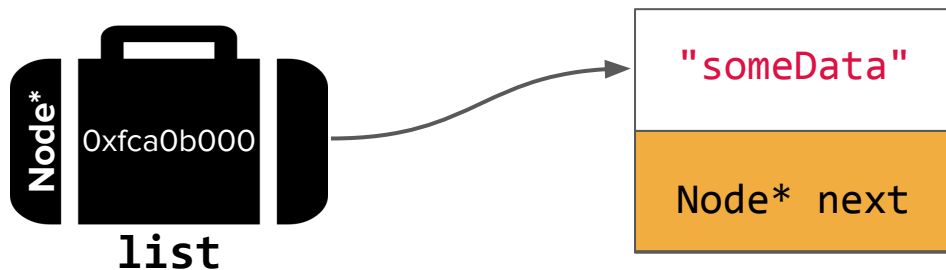
Pointer to a node



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

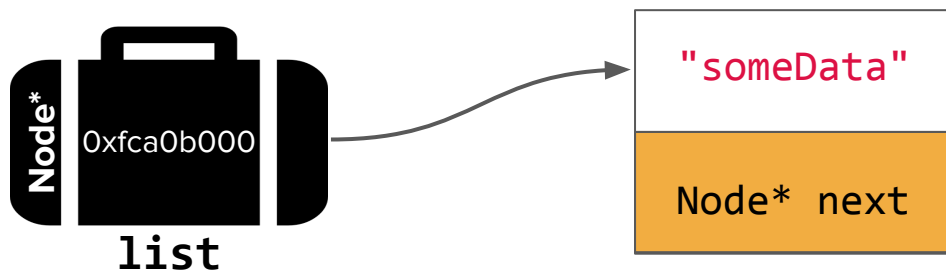
```
Node* list = new Node;
```

Pointer to a node



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

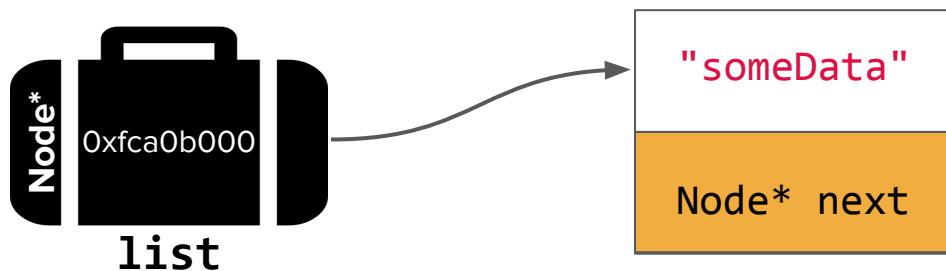
Pointer to a node



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

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

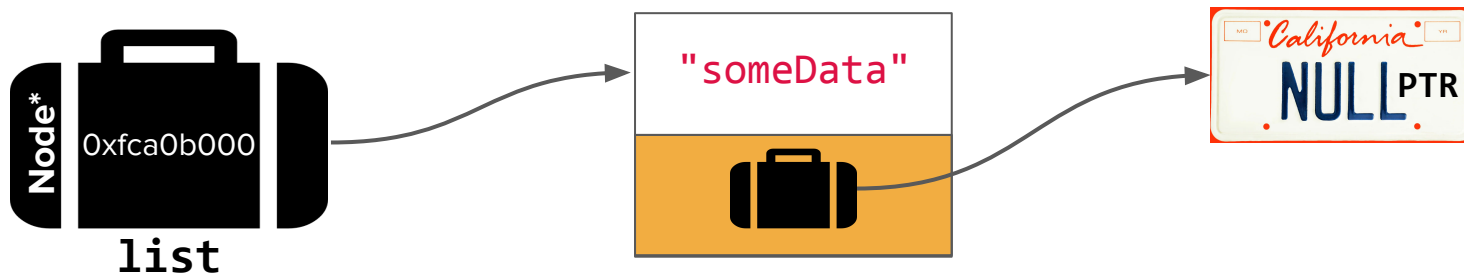
Pointer to a node



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

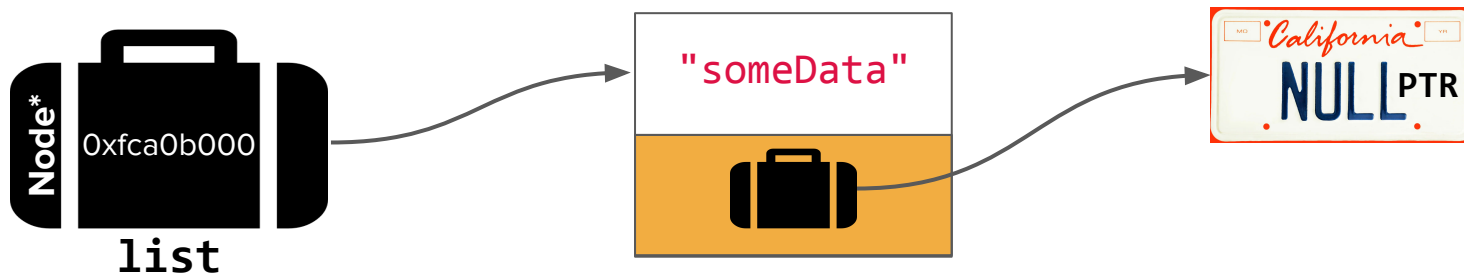
Use dot (.) notation to update the data field of the struct.

Pointer to a node



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

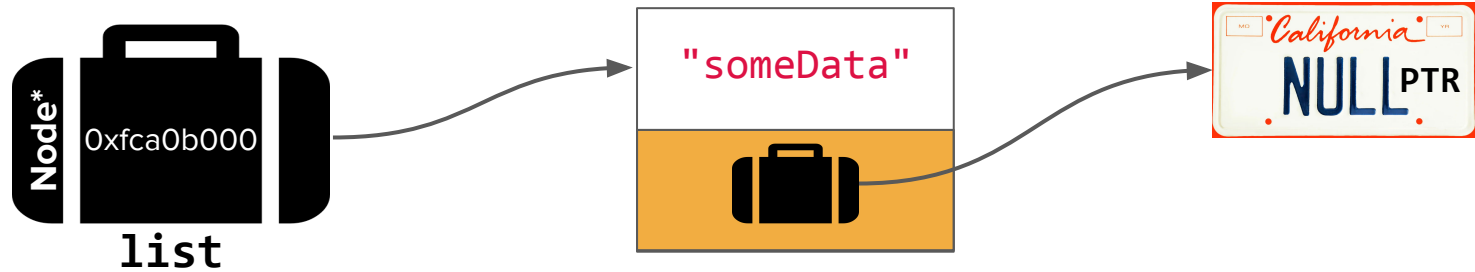
Pointer to a node



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

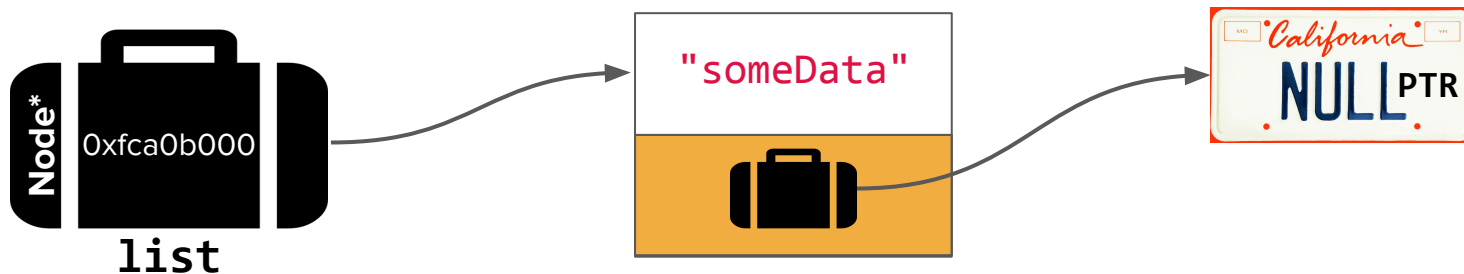
There's an easier way!

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.

Announcements

Announcements

- Final project proposals are due **today at 11:59pm PDT**. Nick and I will try to have feedback to you by early next week.
 - In the meantime, make sure to take a look at the project timeline to stay on track!
- Assignment 5 is due this **Friday, August 6 at 11:59pm PDT**. The grace period will be 48 hours.
 - Good use of the debugger is essential in this assignment. Use the techniques in the warm-up to help you uncover those tricky memory bugs!
- Assignment 6 will also be released on Friday.

How do we use linked lists in a class?

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:

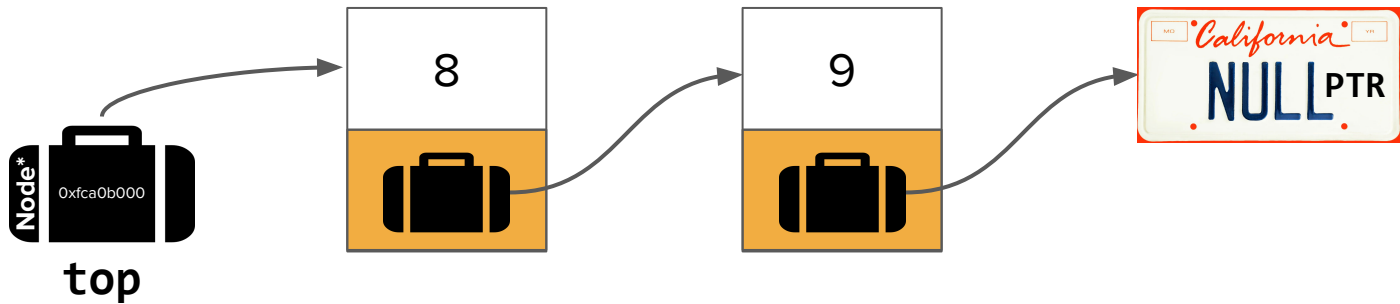
```
Stack myStack = {9, 8}; // 8 is at the "top" of the stack  
myStack.push(7); // we want the result to be {9, 8, 7}
```

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:

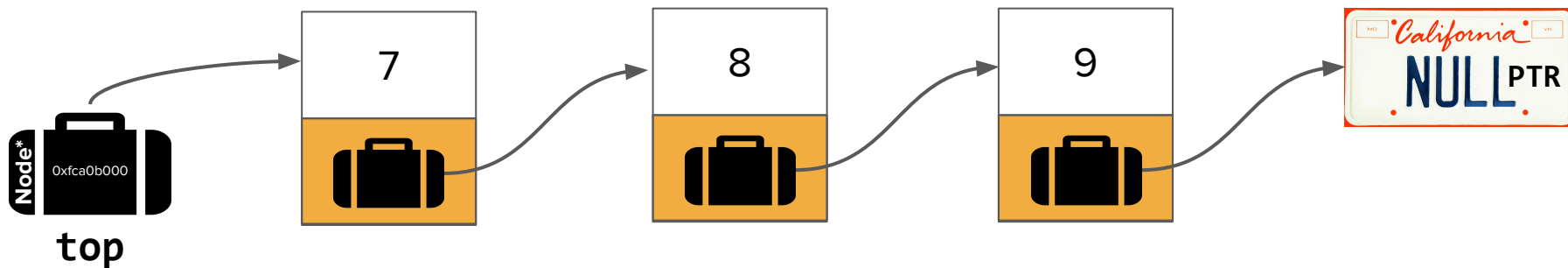


push()

- 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:

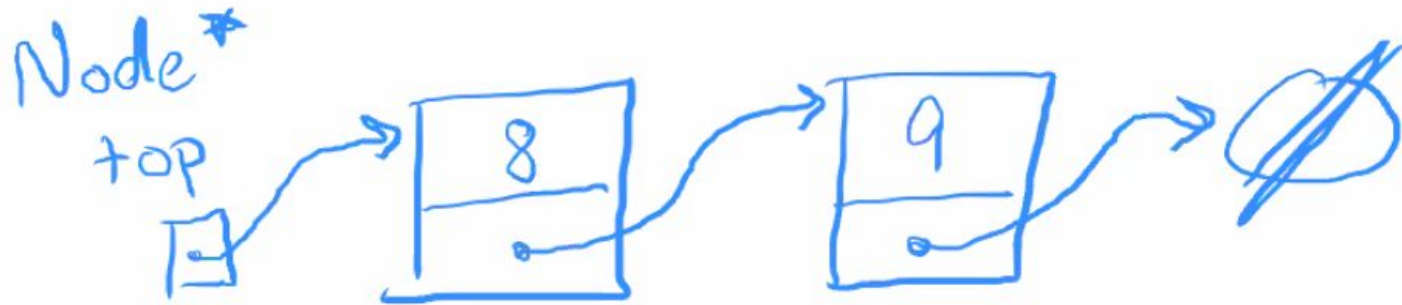


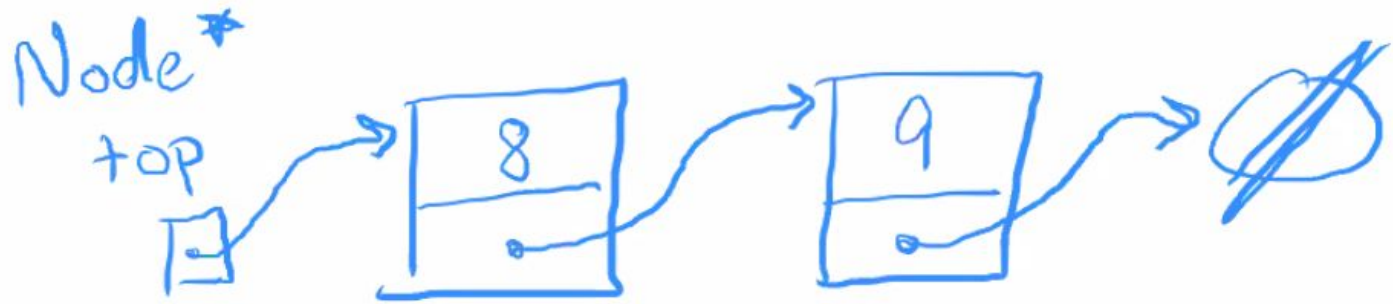
Let's code **push()**!

Live Activity Summary

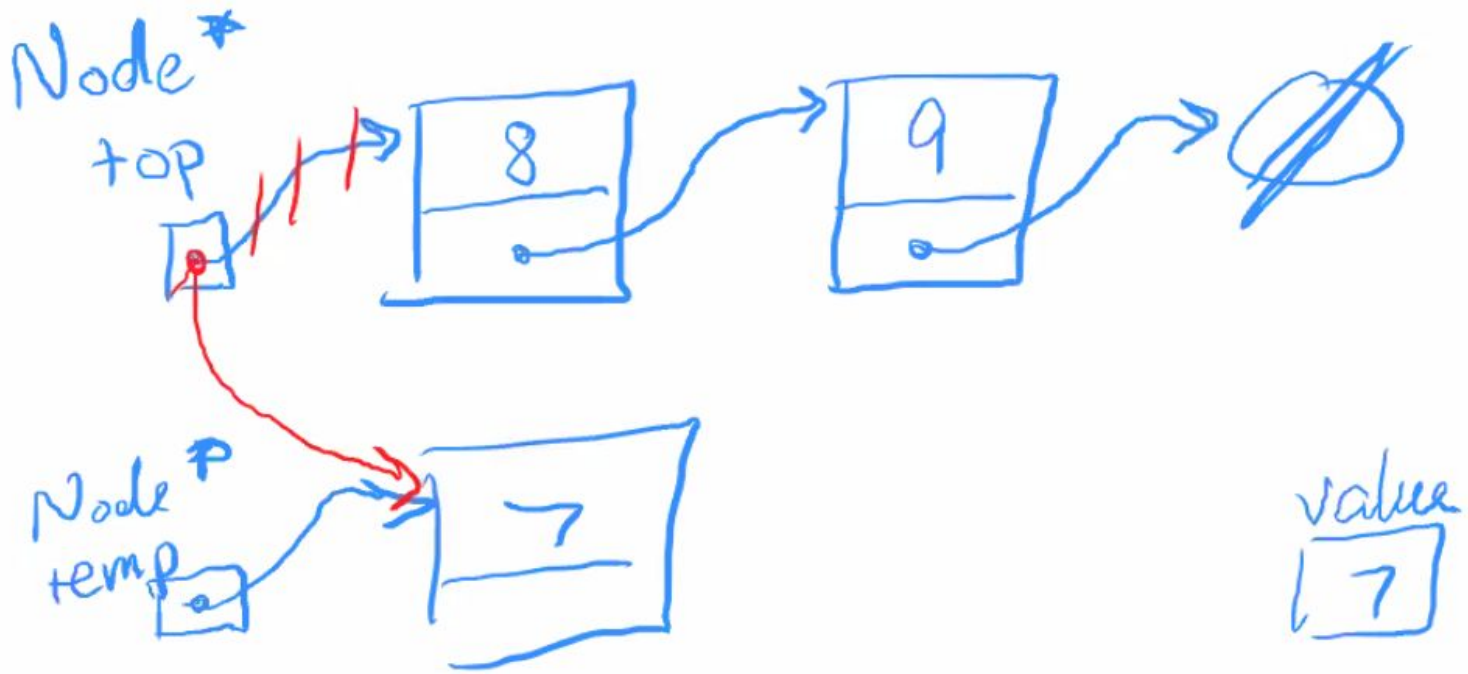
- We strongly recommend watching the live recording of the coding activity, as the code and explanations contextualize the following diagrams

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

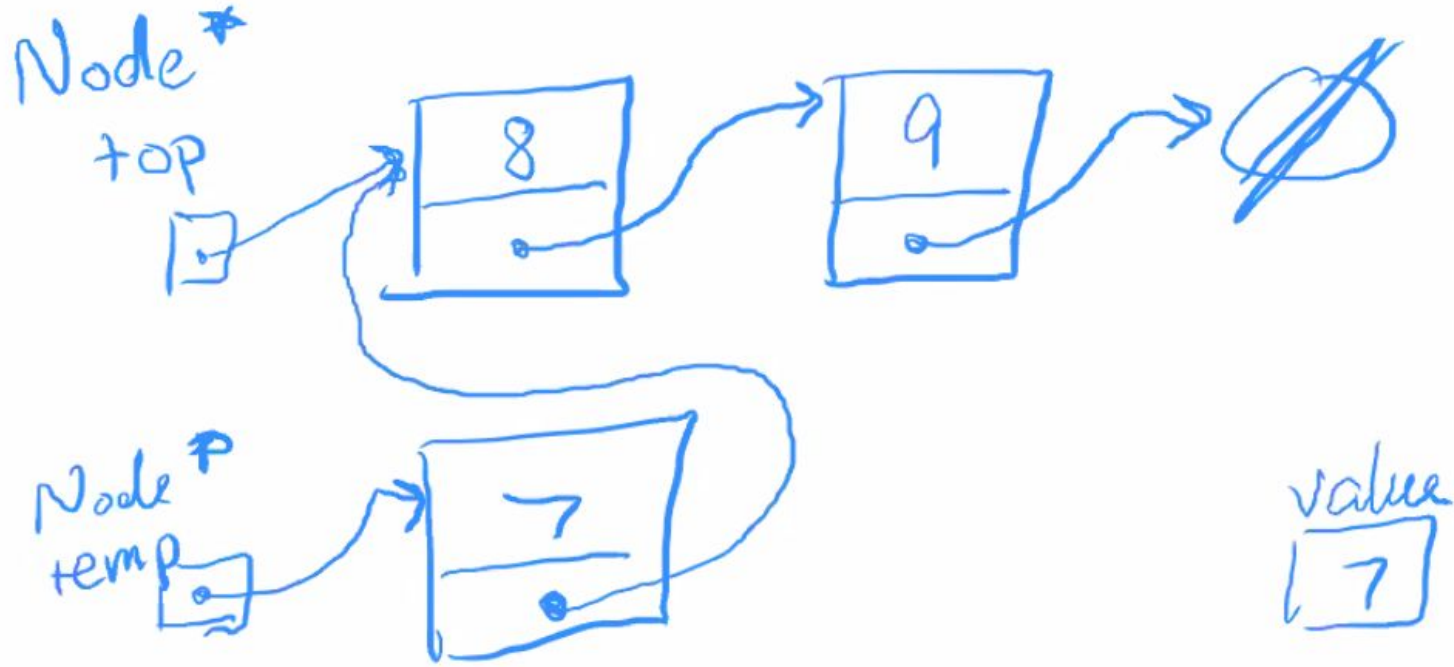




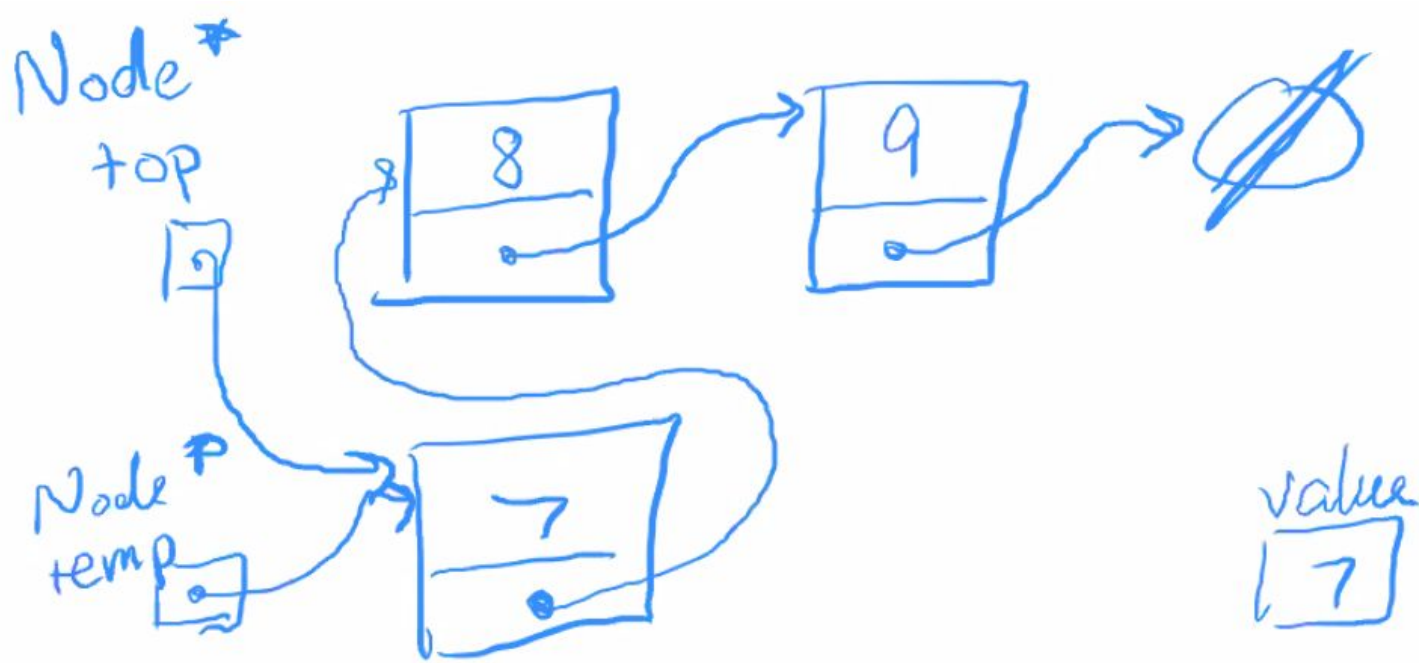
```
Node *temp = new Node;  
temp->data = 7;
```



```
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?

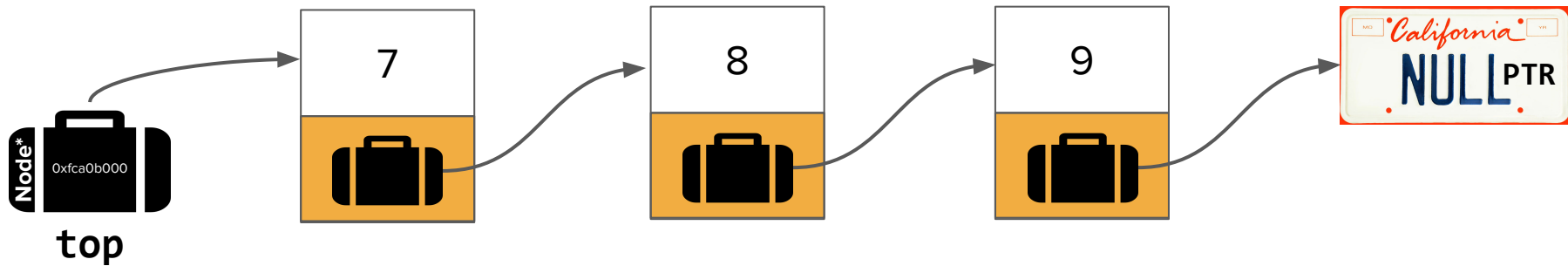
pop()

- Now we want to remove the top value:

...

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

Starting state of the list:



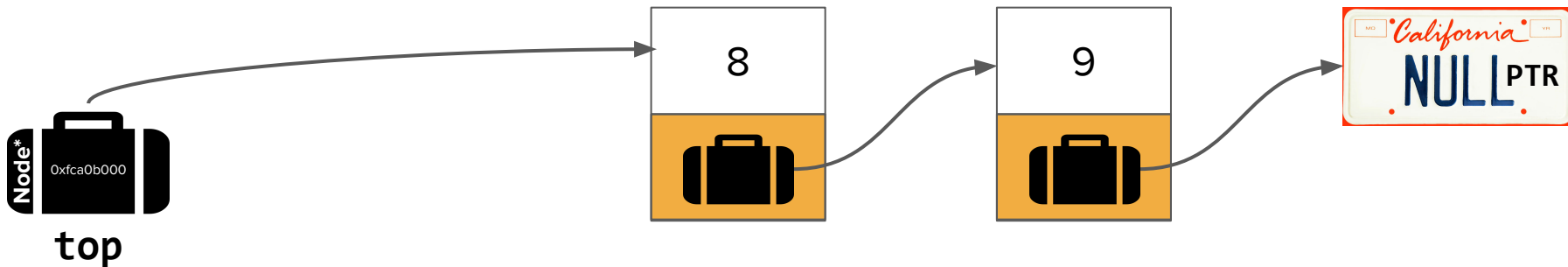
pop()

- Now we want to remove the top value:

...

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

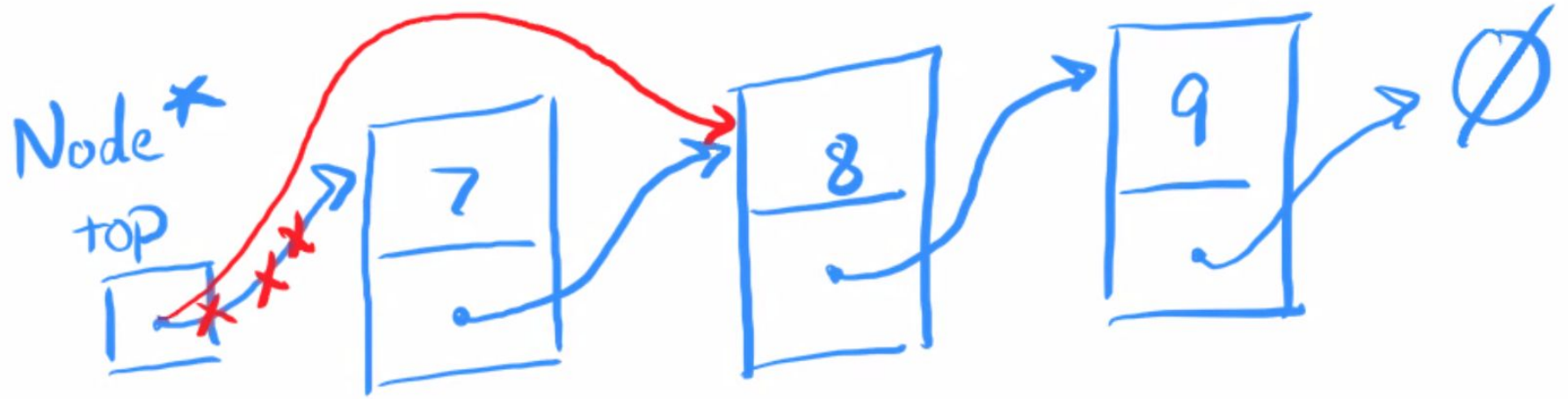
Goal:



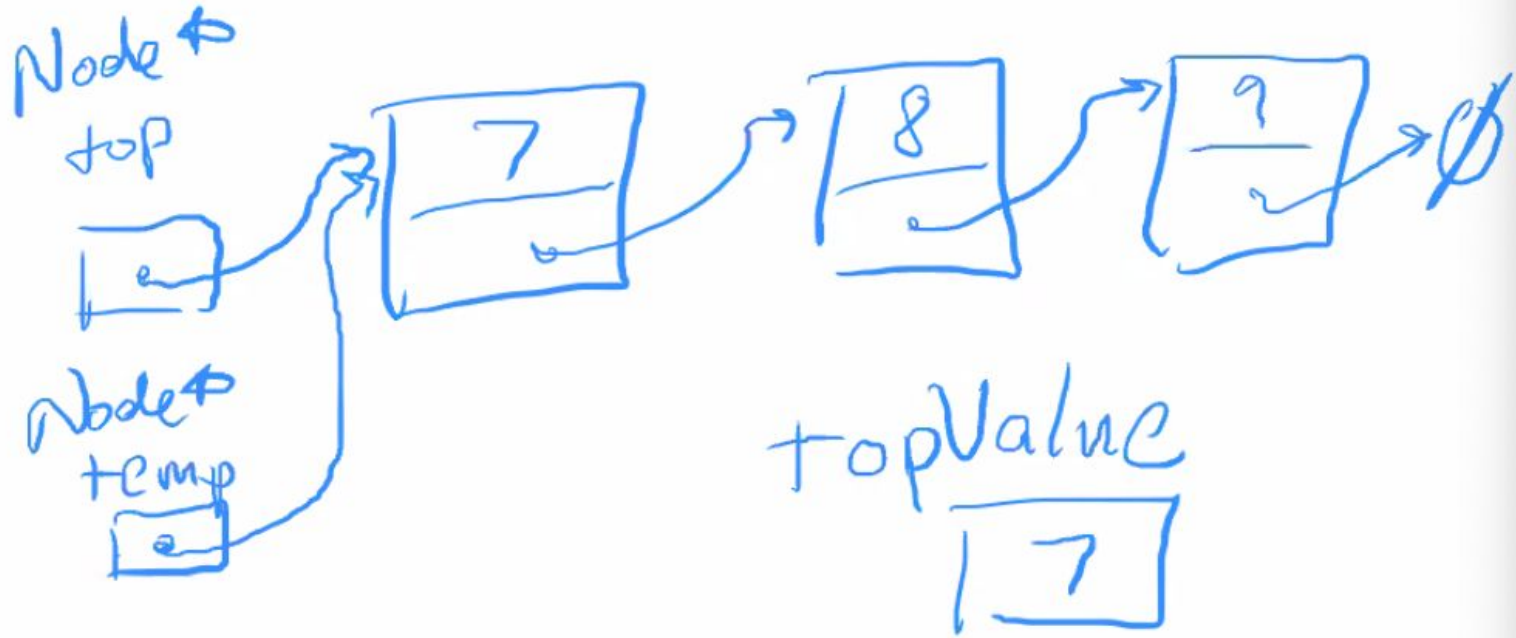
Let's code **pop()**!

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

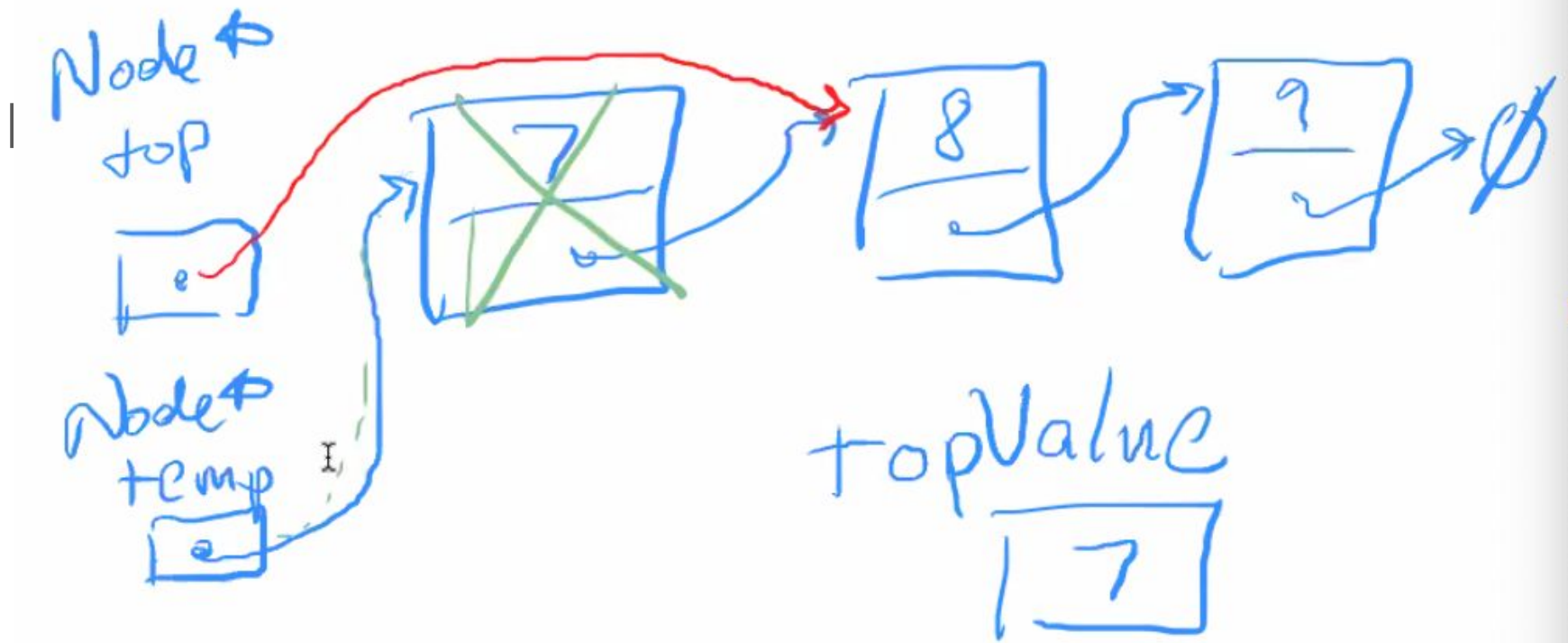




```
top = top->next; // INCORRECT
```



```
Node* temp = top;
```



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

Three Stack operations

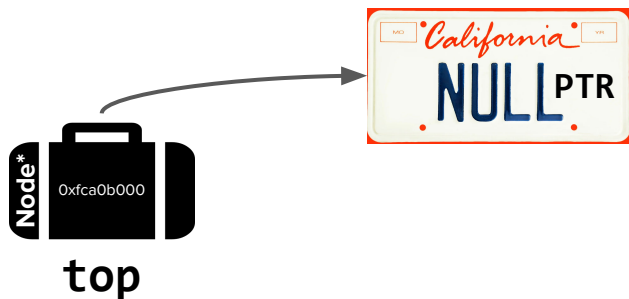
- `push()`
- `pop()`
- **Destructor**

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!

IntStack takeaways

- 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 no longer have a pointer to)

How do we manipulate linked
lists?

Linked list utility functions

- We've now seen linked lists in the context of classes, where we used a linked list as the data storage underlying an implementation of a Stack.

Linked list utility functions

- We've now seen linked lists in the context of classes, where we used a linked list as the data storage underlying an implementation of a Stack.
- However, linked lists are not limited only to use within classes. In fact, the next assignment will ask you to implement "standalone" linked list functions that operate on provided linked lists, outside the context of a class.

Linked list utility functions

- We've now seen linked lists in the context of classes, where we used a linked list as the data storage underlying an implementation of a Stack.
- However, linked lists are not limited only to use within classes. In fact, the next assignment will ask you to implement "standalone" linked list functions that operate on provided linked lists, outside the context of a class.
- This is the paradigm that we will work under for the several functions. In doing so, we'll gain a little more flexibility to get practice with many different linked list operations and build our linked list toolbox!

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?

Linked List Traversal

Traversal utility functions

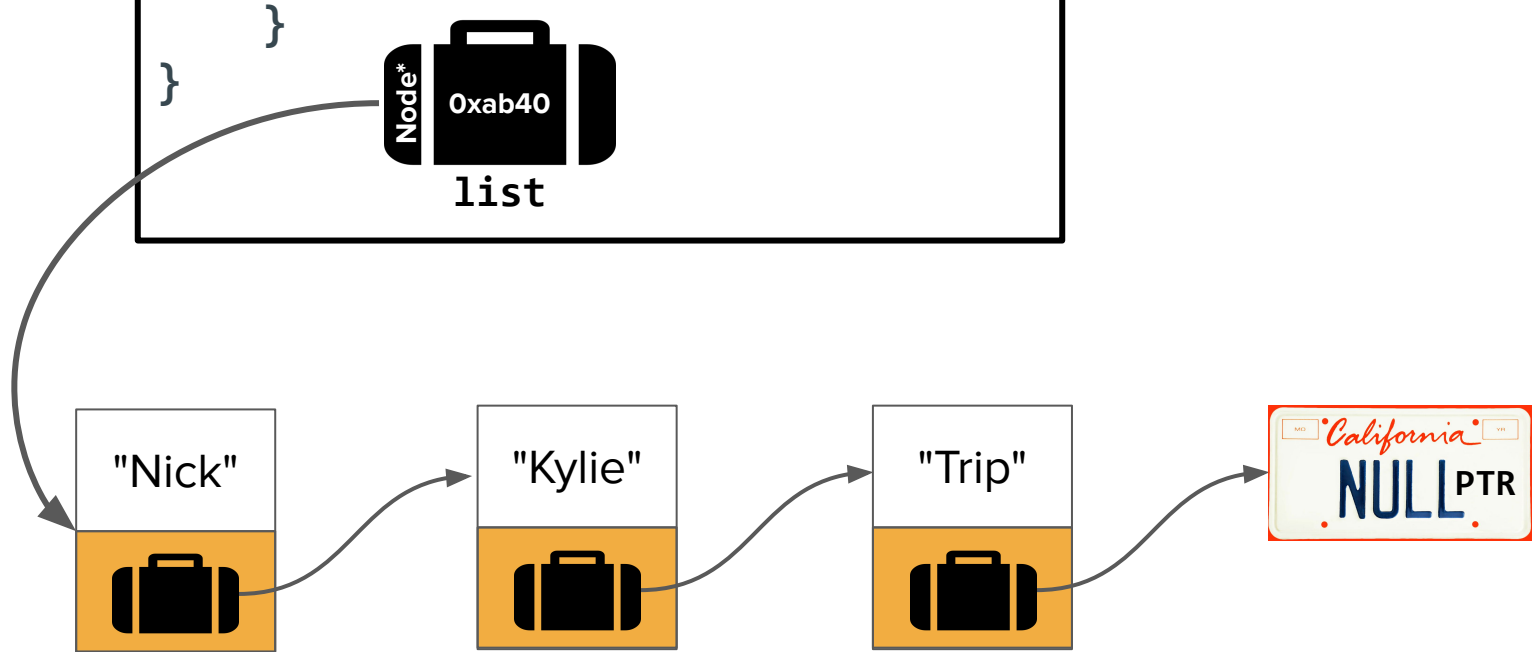
- Freeing a linked list
- Printing a linked list
- Measuring the length of a list

Traversal utility functions

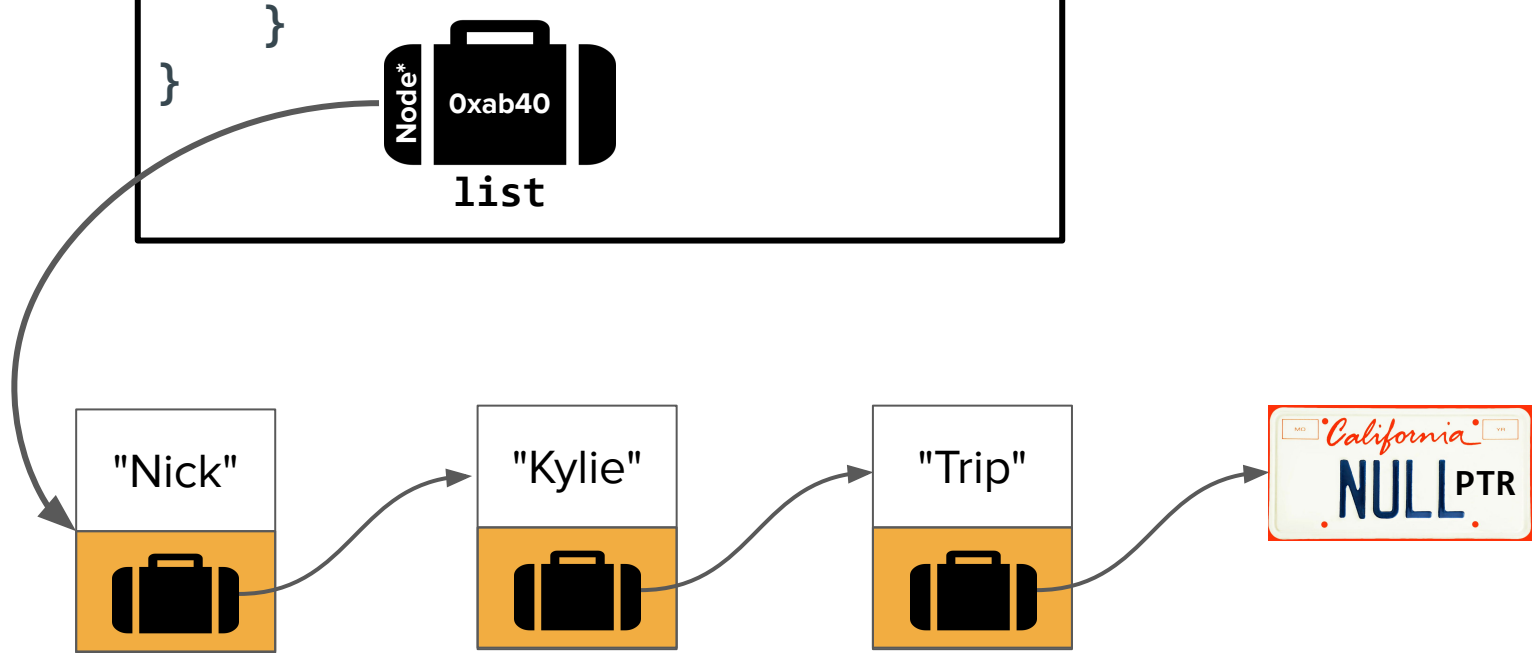
- **Freeing a linked list**
 - Very similar to the destructor we just saw!
- Printing a linked list
- Measuring the length of a list

Freeing linked lists,
the wrong way

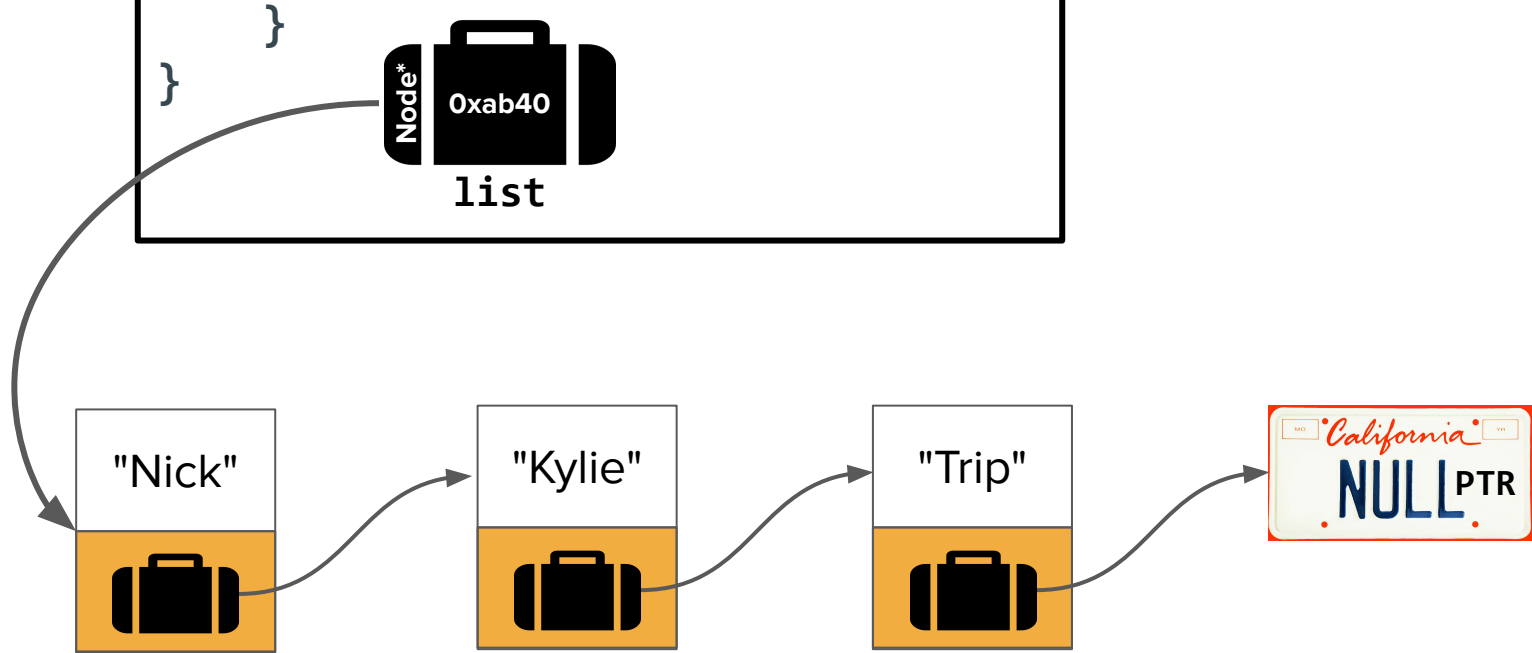
```
void freeList(Node* list) {  
    /* WRONG WRONG WRONG WRONG WRONG */  
    while (list != nullptr) {  
        delete list;  
        list = list->next;  
    }  
}
```



```
void freeList(Node* list) {  
    /* WRONG WRONG WRONG WRONG WRONG */  
    while (list != nullptr) {  
        delete list;  
        list = list->next;  
    }  
}
```

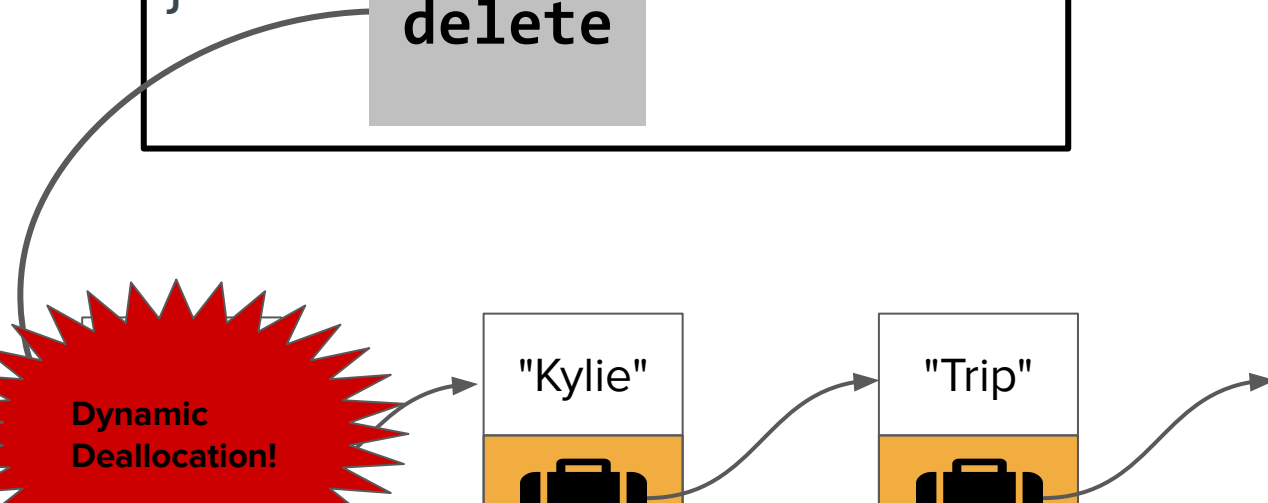


```
void freeList(Node* list) {  
    /* WRONG WRONG WRONG WRONG WRONG */  
    while (list != nullptr) {  
        delete list;  
        list = list->next;  
    }  
}
```

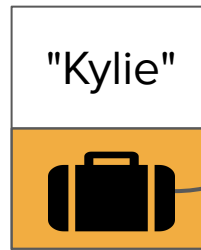


```
void freeList(Node* list) {  
    /* WRONG WRONG WRONG WRONG WRONG */  
    while (list != nullptr) {  
        delete list;  
        list = list->next;  
    }  
}
```

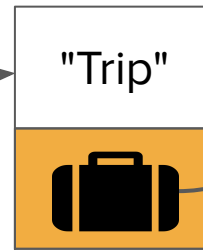
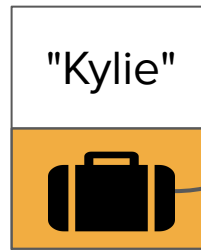
delete



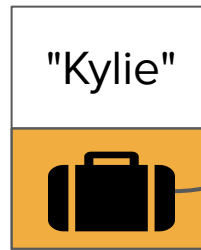
```
void freeList(Node* list) {  
    /* WRONG WRONG WRONG WRONG WRONG */  
    while (list != nullptr) {  
        delete list;  
        list = list->next;  
    }  
}
```



```
void freeList(Node* list) {  
    /* WRONG WRONG WRONG WRONG WRONG */  
    while (list != nullptr) {  
        delete list;  
        list = list->next;  
    }  
}
```

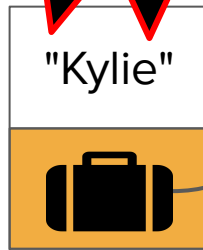


```
void freeList(Node* list) {  
    /* WRONG WRONG WRONG WRONG WRONG */  
    while (list != nullptr) {  
        delete list;  
        list = list->next;  
    }  
}
```



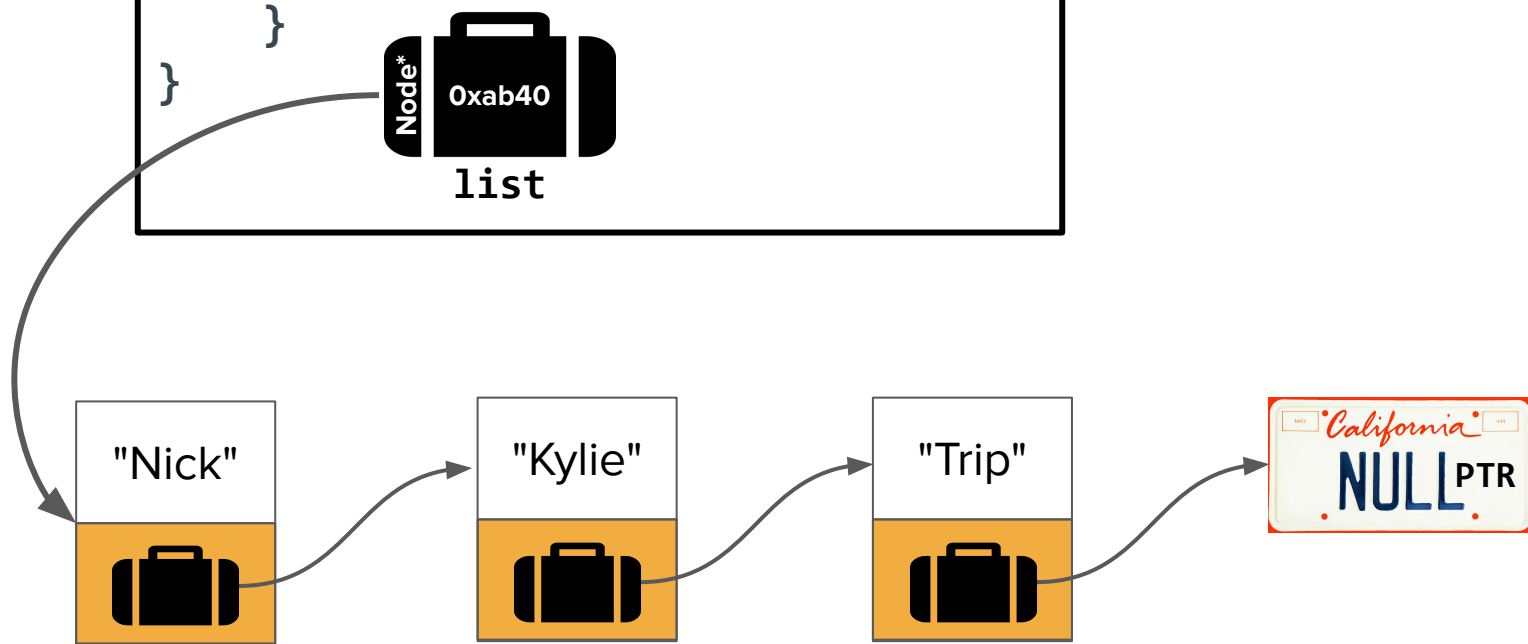
```
void freeList(Node* list) {  
    /* WRONG WRONG WRONG WRONG WRONG */  
    while (list != nullptr) {  
        ...  
    }  
}
```

**Undefined
Behavior!**

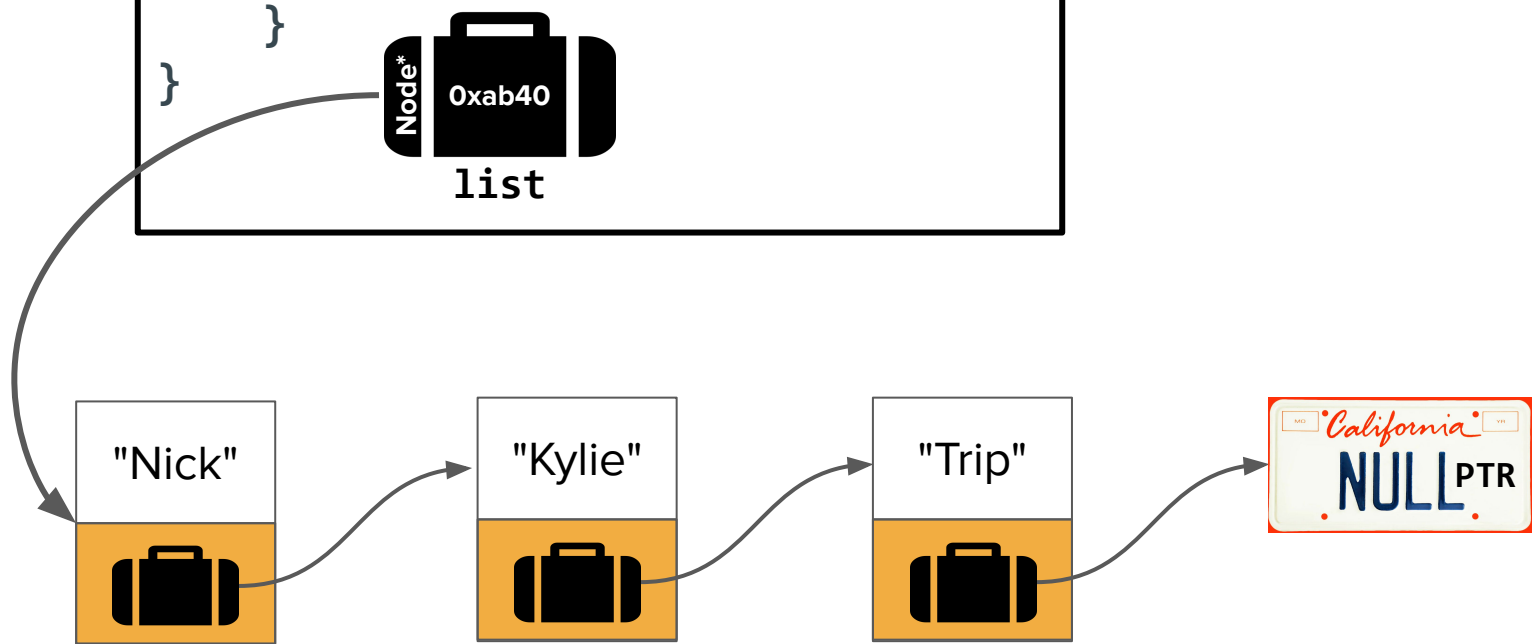


Freeing linked lists,
the right way

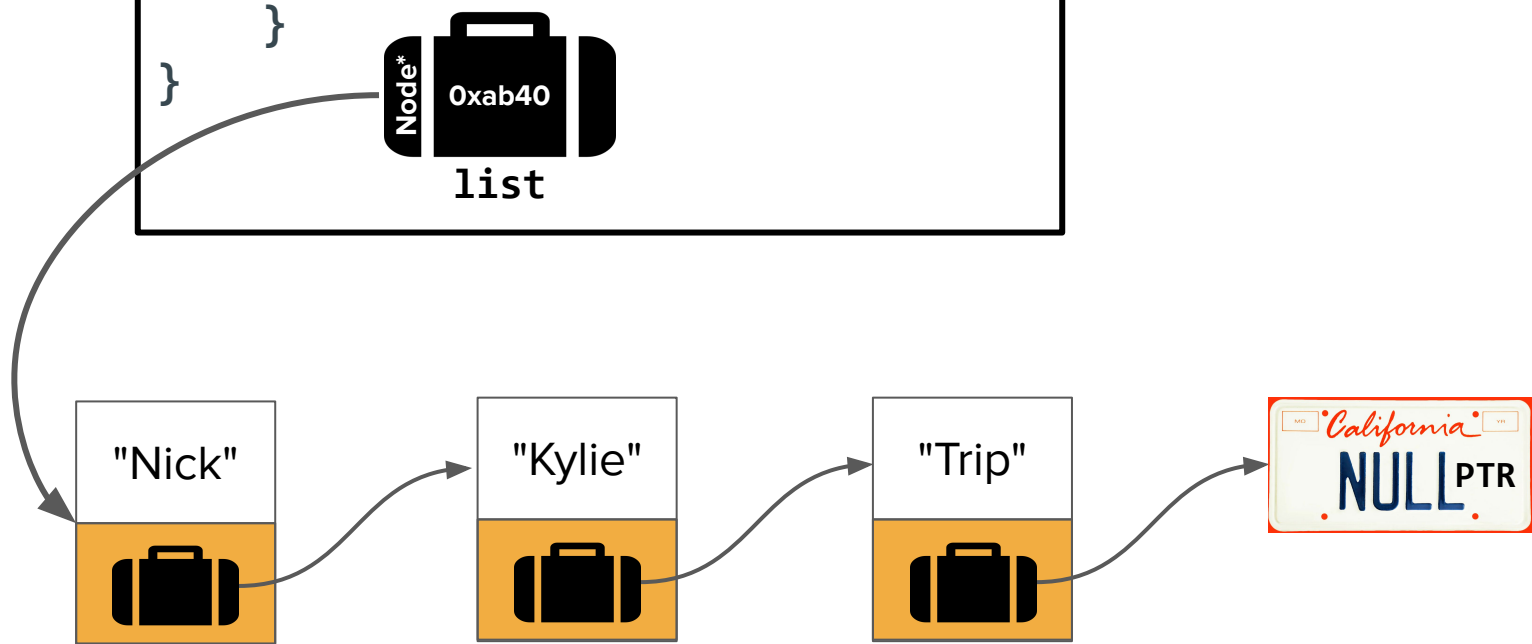
```
void freeList(Node* list) {  
    while (list != nullptr) {  
  
        delete list;  
        list = list->next;  
    }  
}
```



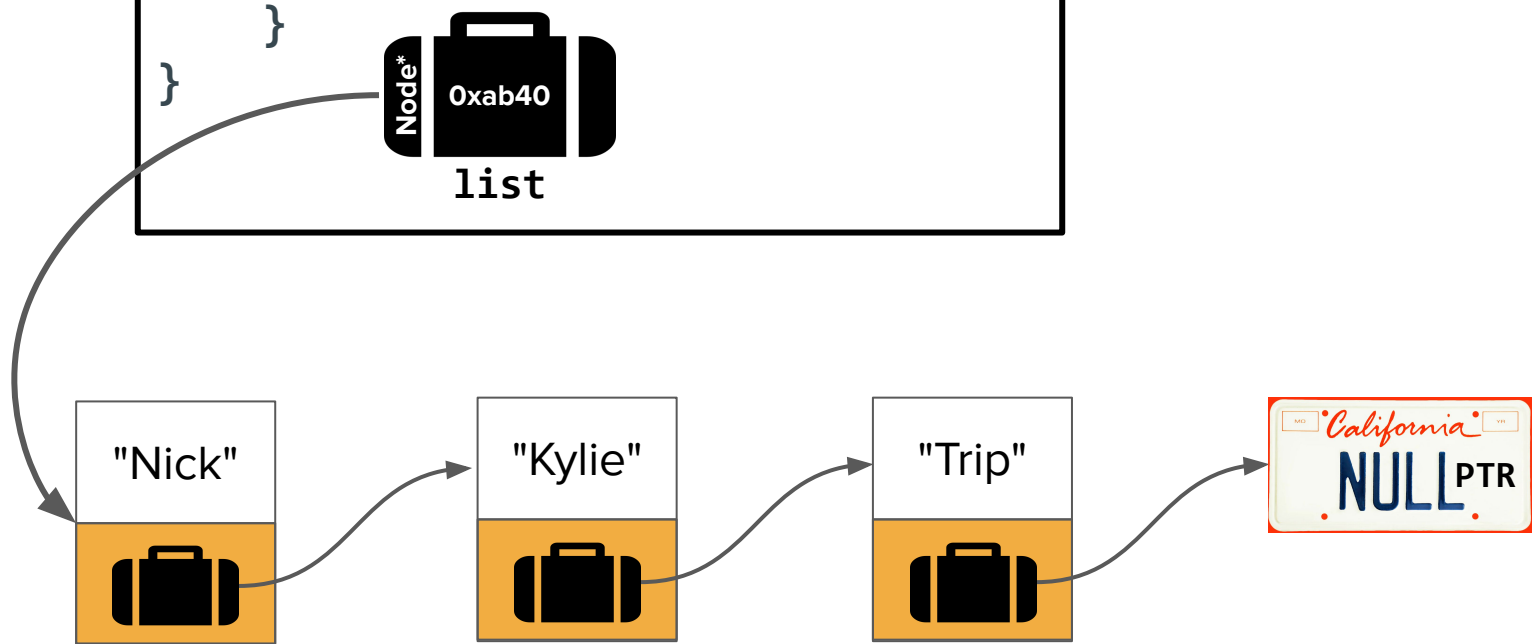
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = list->next;  
    }  
}
```



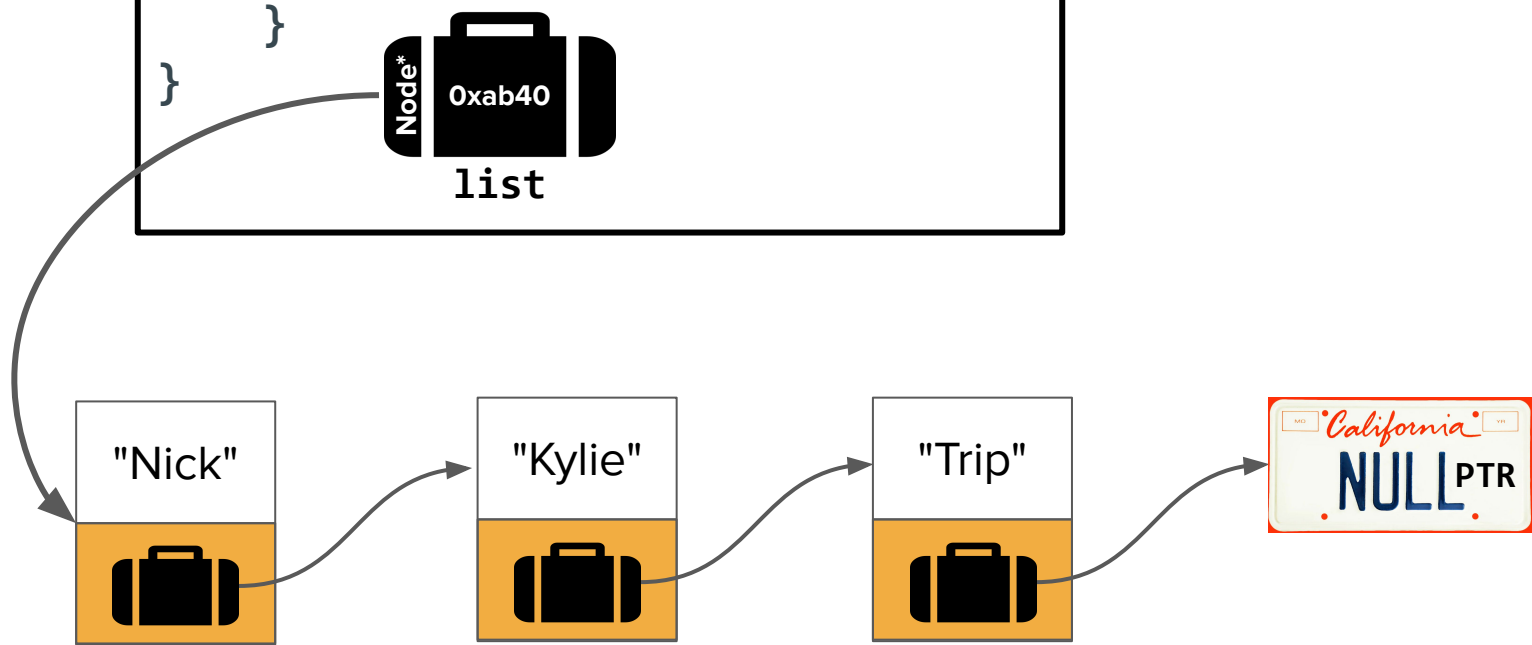
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



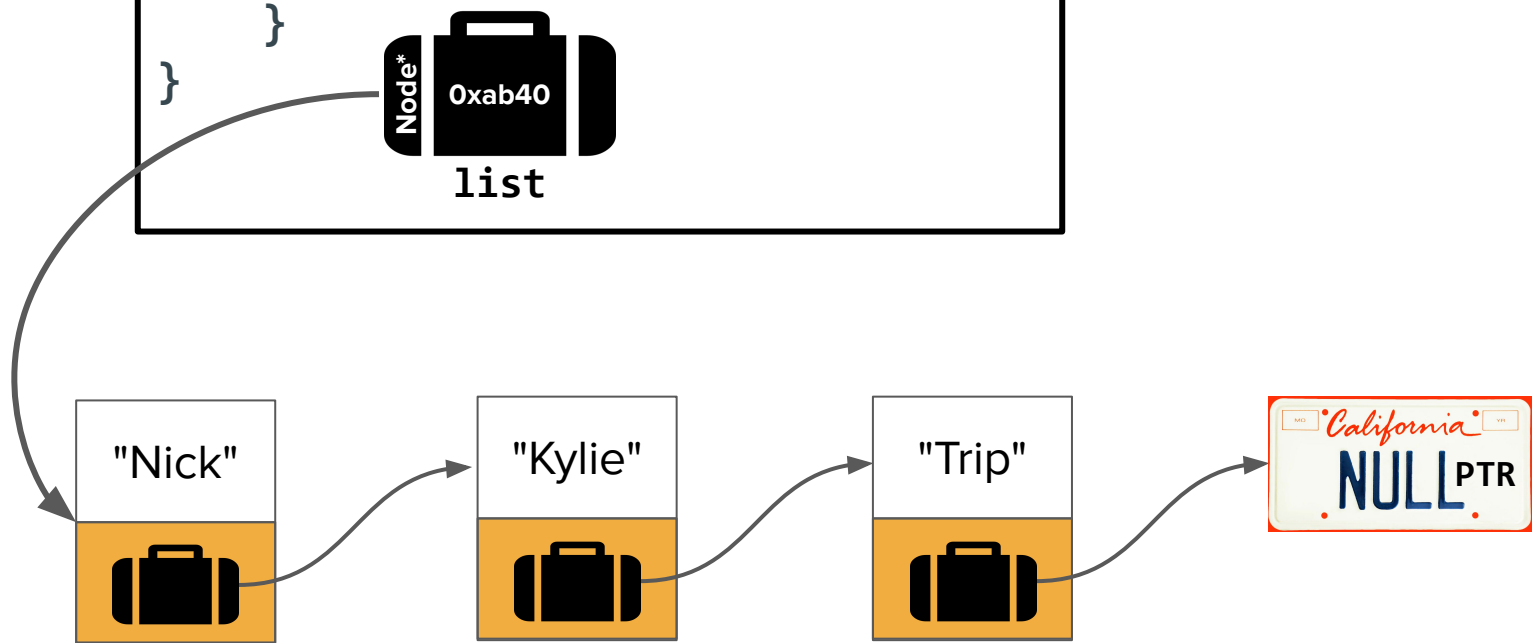
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



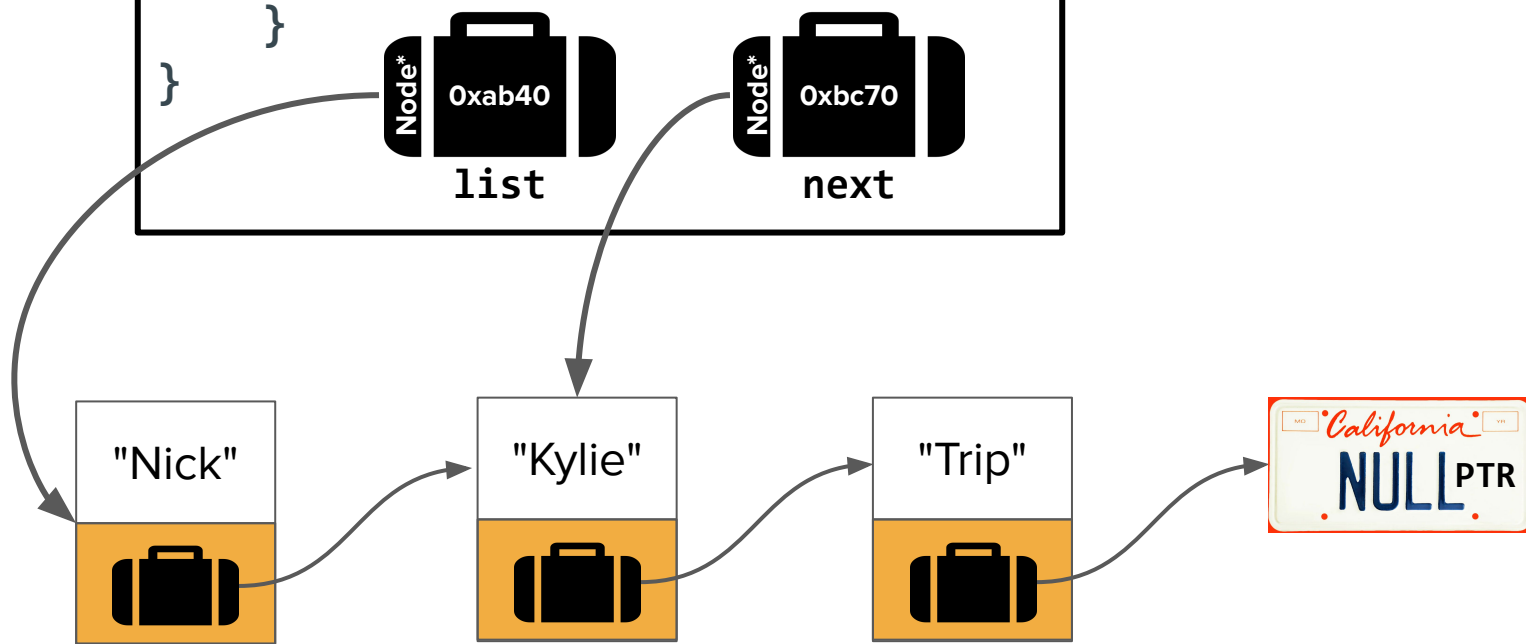
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



list



next



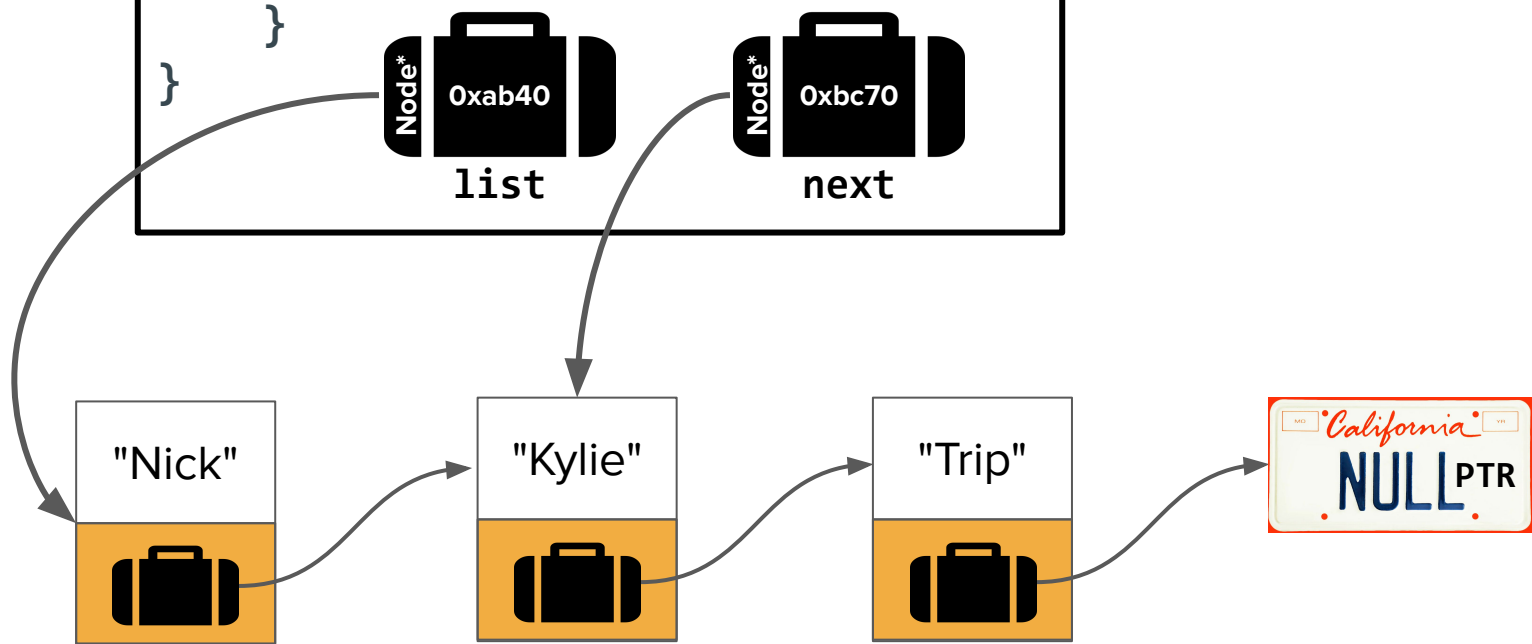
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



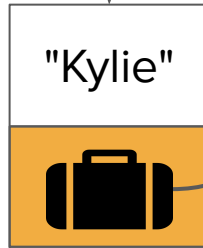
list



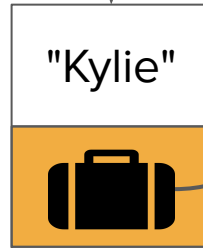
next



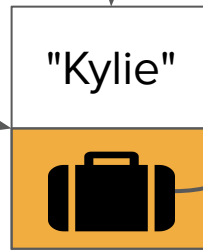
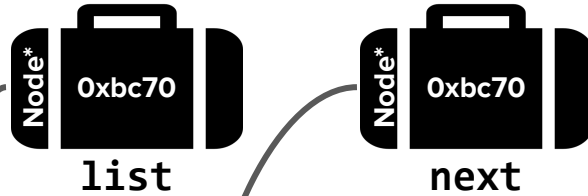
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



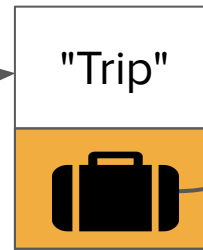
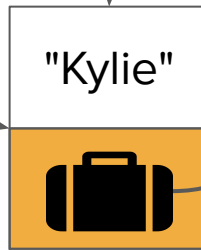
}



list



next



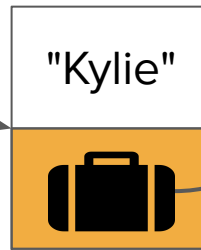
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```

}

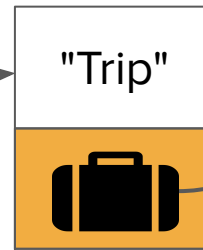
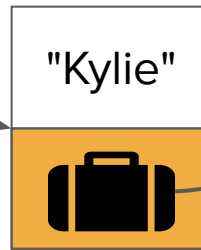
}



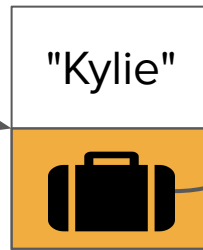
list



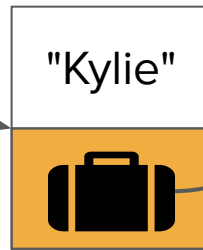
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



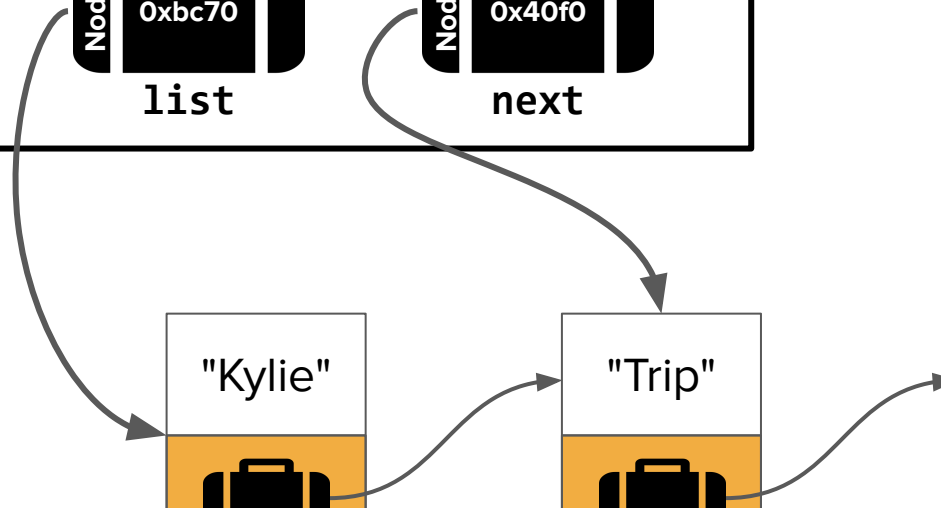
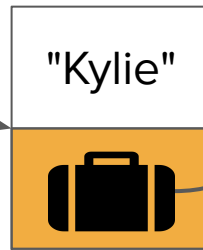
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



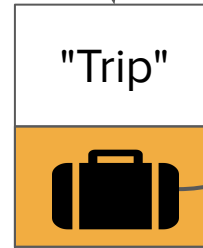
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



list



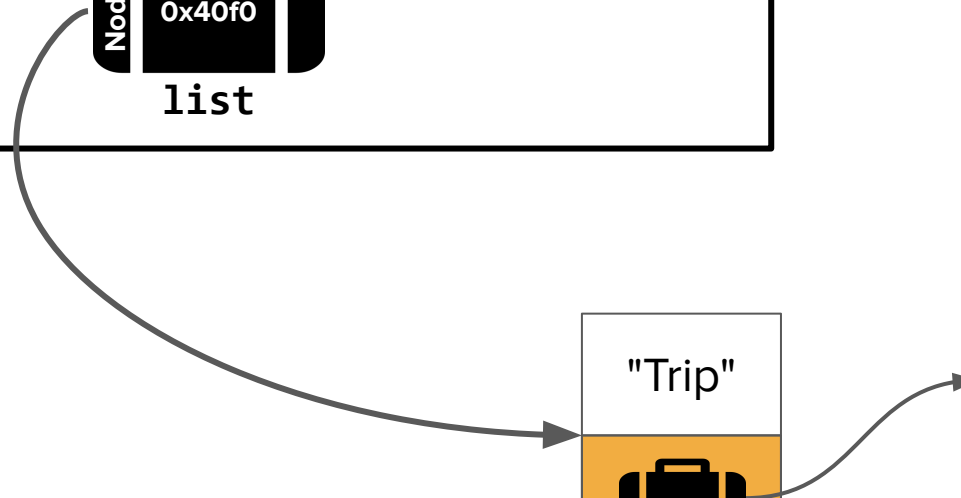
next



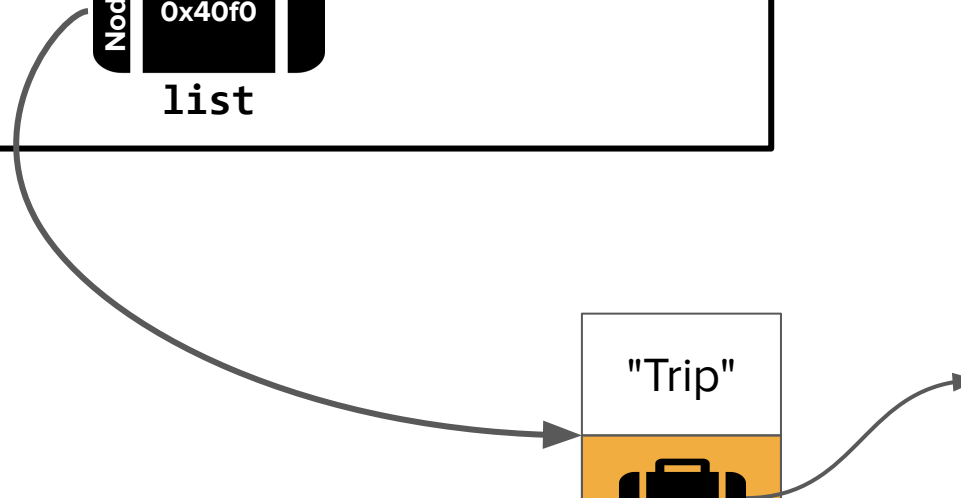
"Trip"



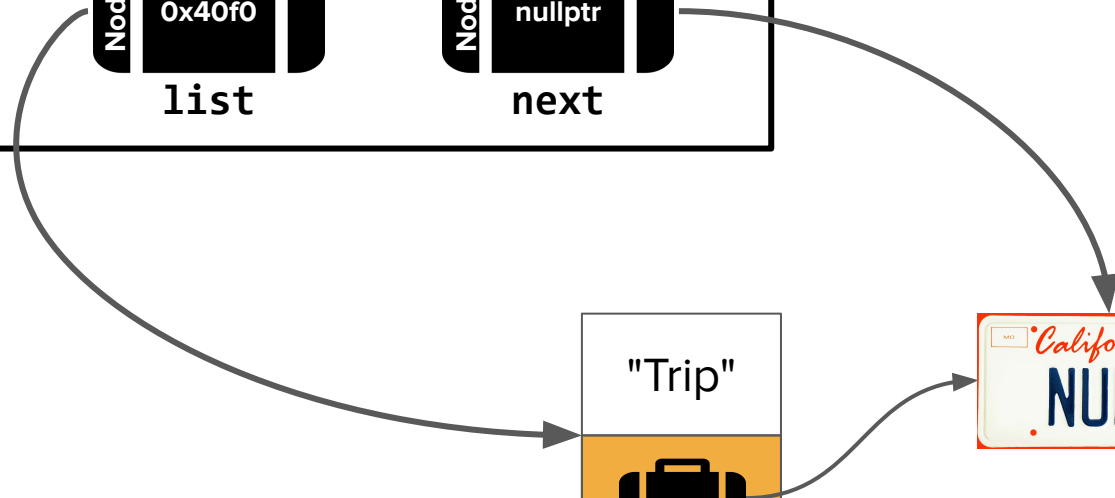
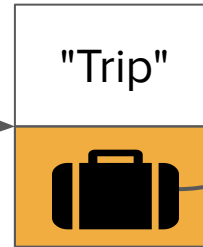
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



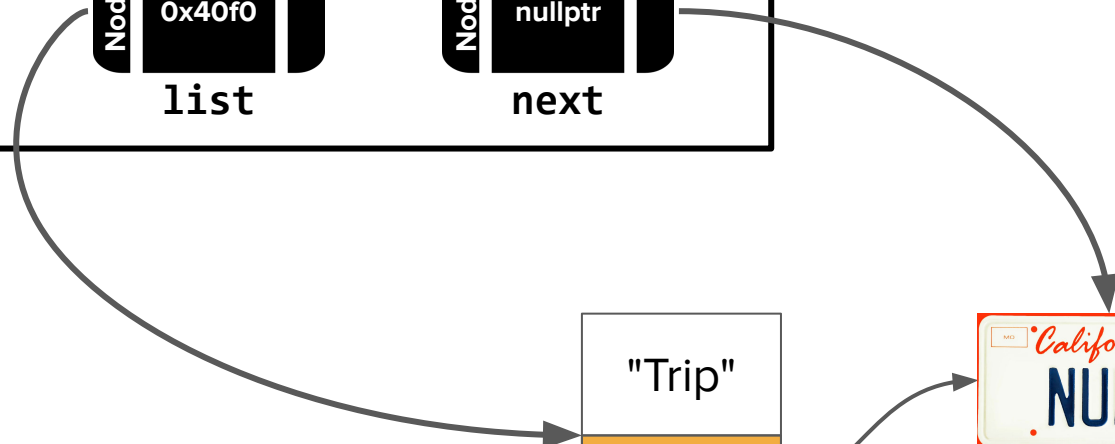
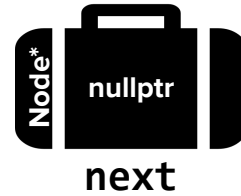
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



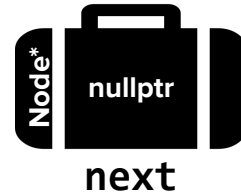
```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* next = list->next;  
        delete list;  
        list = next;  
    }  
}
```



**All memory
freed! Wooo!**

Traversal utility functions

- Freeing a linked list
- **Printing a linked list**
- Measuring the length of a list

Printing a linked list

Inspecting Linked List Contents

- Being able to "see" the contents of a linked list is a really helpful debugging tool!

Inspecting Linked List Contents

- Being able to "see" the contents of a linked list is a really helpful debugging tool!
- There are two main ways to do so: using the **debugger** and printing to the **console**

Inspecting Linked List Contents

- Being able to "see" the contents of a linked list is a really helpful debugging tool!
- There are two main ways to do so: using the **debugger** and printing to the **console**
- First attempt: What is the result of the following code? (Poll)

```
/* Creates a list with contents "Hello" -> "World" -> nullptr */  
Node* list = createList();  
cout << list << endl;
```

Inspecting Linked List Contents

- Being able to "see" the contents of a linked list is a really helpful debugging tool!
- There are two main ways to do so: using the **debugger** and printing to the **console**
- First attempt: What is the result of the following code? (Poll)

```
/* Creates a list with contents "Hello" -> "World" -> nullptr */  
Node* list = createList();  
cout << list << endl;
```

Answer: Some memory address is printed! We can't predict the exact value.

Inspecting Linked List Contents

- Being able to "see" the contents of a linked list is a really helpful debugging tool!
- There are two main ways to do so: using the **debugger** and printing to the **console**
- First attempt (directly printing list pointer) unsuccessful.
- Second attempt: Let's write a function to print the list!

printList()

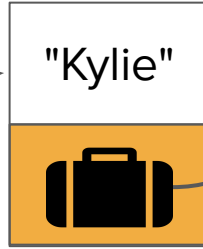
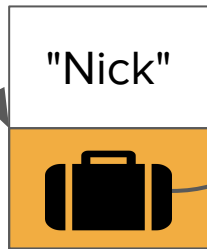
Let's code it!

How does it work?

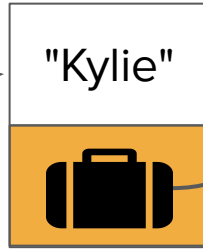
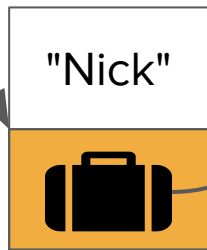
```
int main() {  
    Node* list = readList();  
    printList(list);  
  
    /* other list things happen... */  
}
```

```
int main() {  
    Node* list = readList();  
    printList(list);  
  
    /* other list things happen... */  
}
```

```
int main() {  
    Node* list = readList();  
    printList(list);  
  
    /* other list things happen... */  
}
```

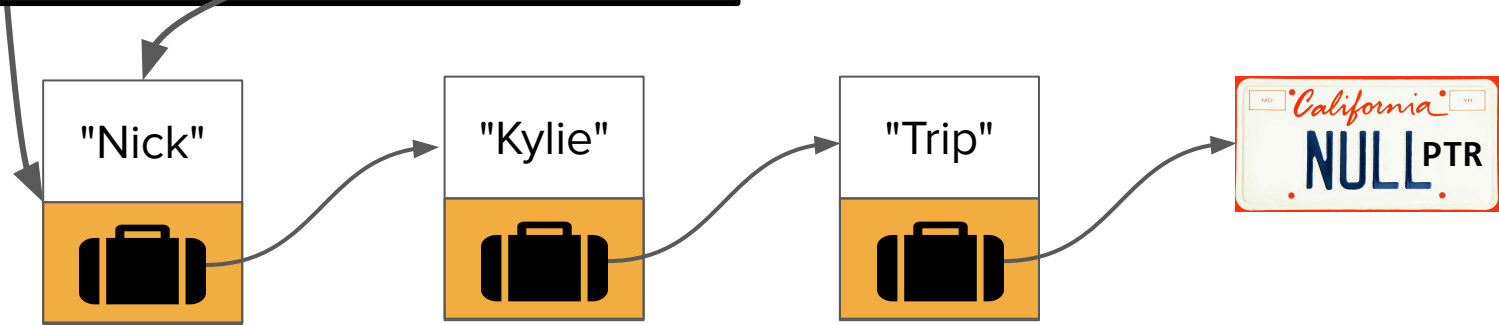


```
int main() {  
    Node* list = readList();  
    printList(list);  
  
    /* other list things happen... */  
}
```



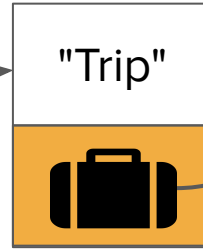
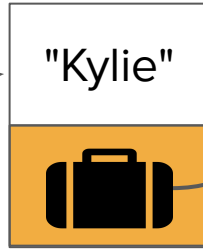
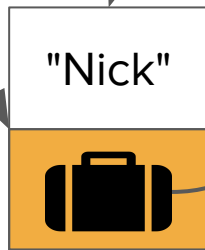
```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```



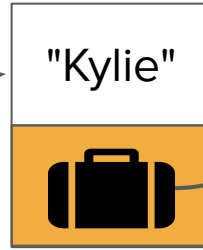
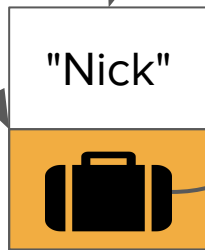
```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```



```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```



```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```



"Nick"

"Kylie"

"Trip"



Nick

```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```



"Nick"

"Kylie"

"Trip"

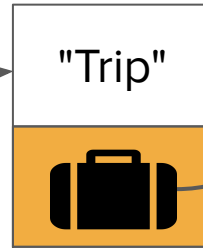
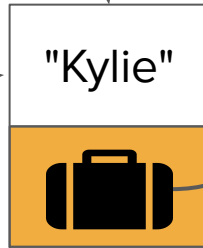
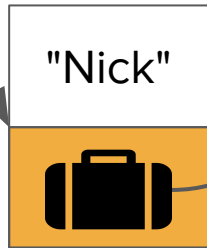


Nick

```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```

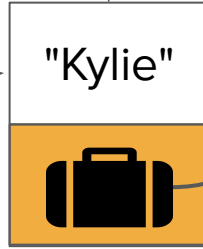
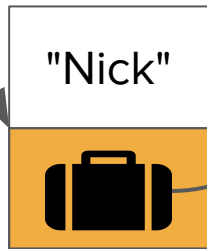
Nick



```
int main() {
```

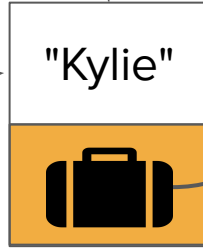
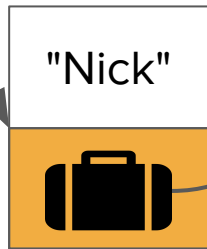
```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```

Nick



```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```

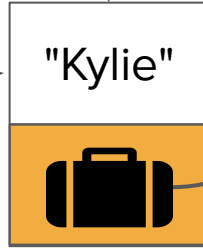
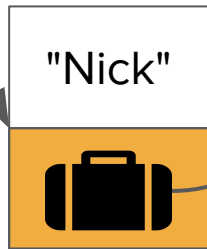


Nick

```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```

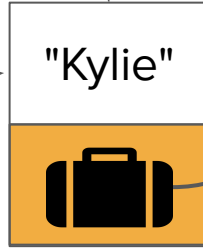
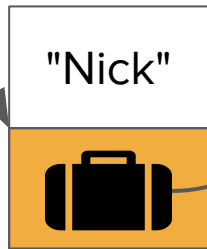
Nick
Kylie



```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```

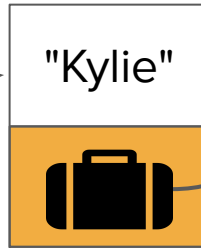
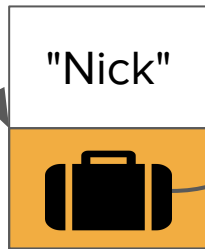
Nick
Kylie



```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```

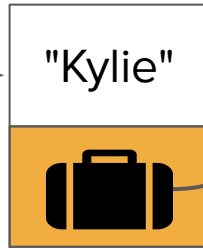
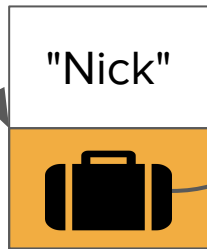
Nick
Kylie



```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```

Nick
Kylie



```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```

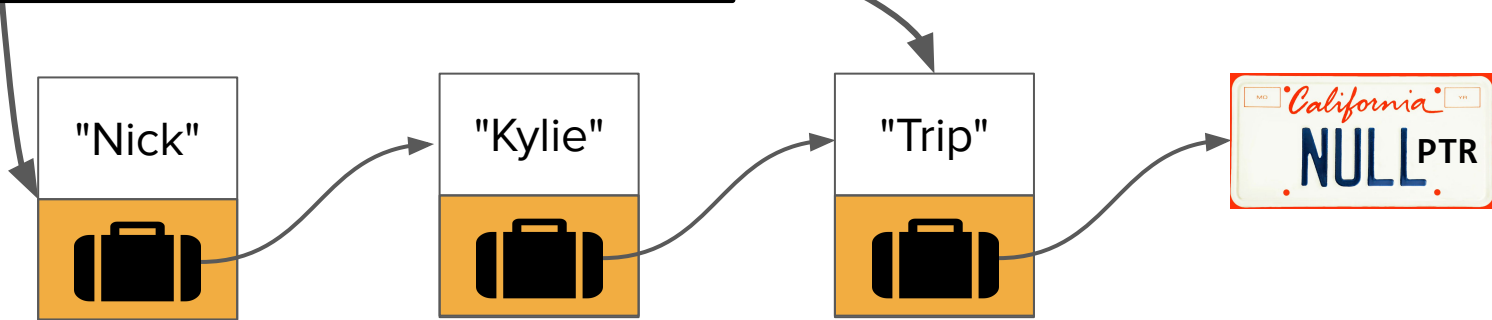
```
}
```

```
}
```

```
}
```



Nick
Kylie



```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```

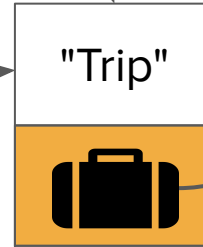
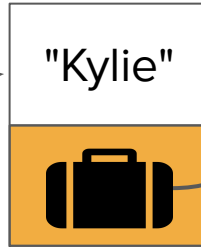
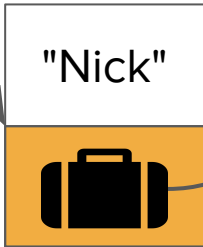
```
}
```

```
}
```

```
}
```



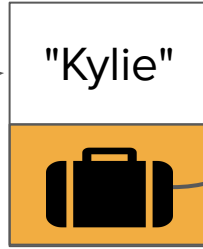
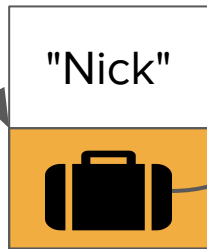
Nick
Kylie
Trip



```
int main() {
```

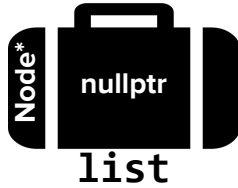
```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```

Nick
Kylie
Trip

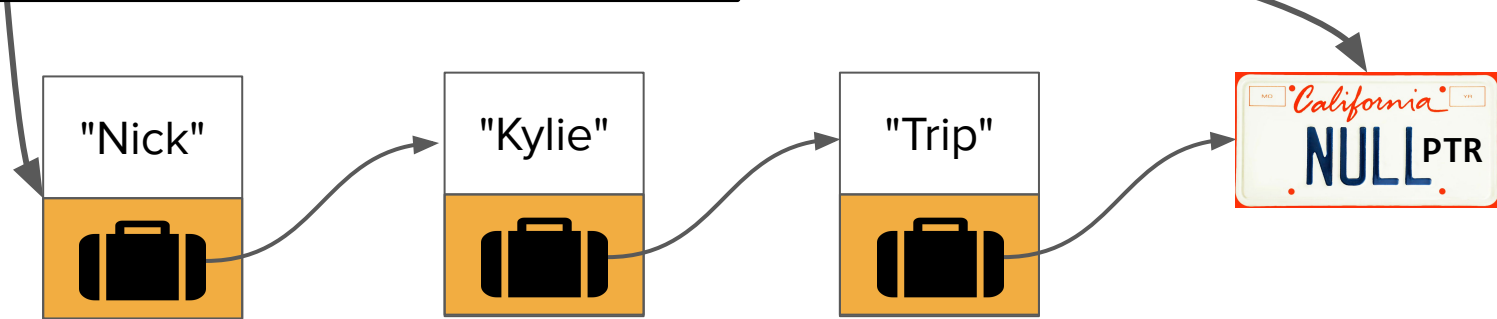


```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```



Nick
Kylie
Trip

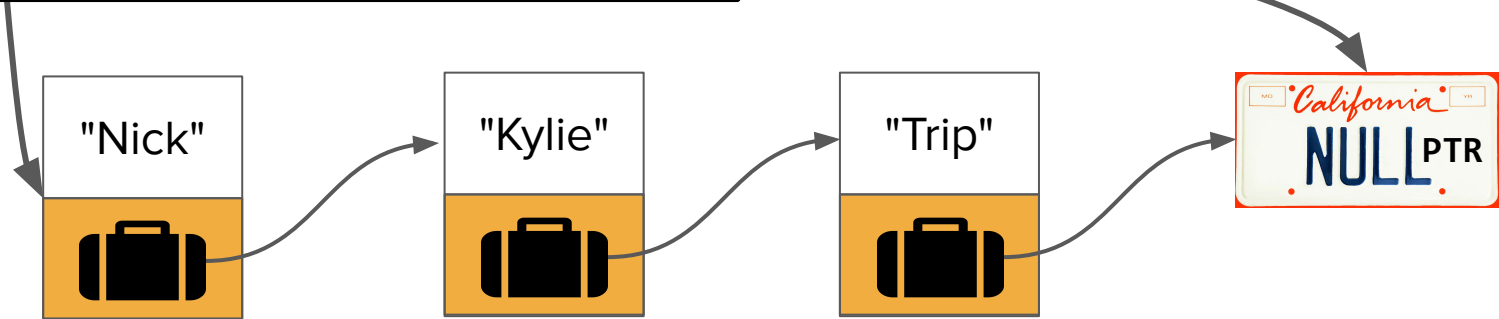


```
int main() {
```

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```



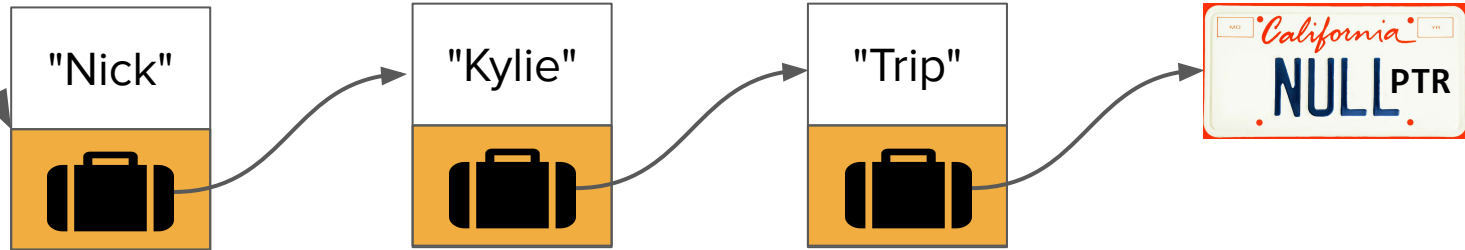
Nick
Kylie
Trip



```
int main() {  
    Node* list = readList();  
    printList(list);  
  
    /* other list things happen... */  
}
```

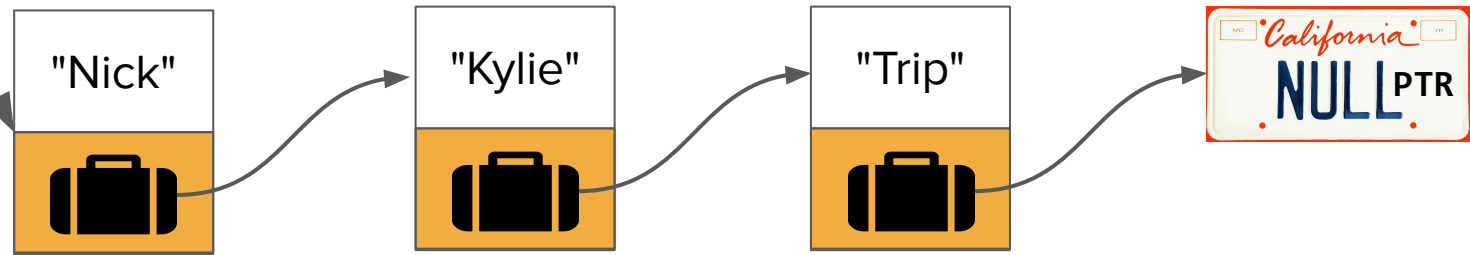


Nick
Kylie
Trip



```
int main() {  
    Node* list = readList();  
    printList(list);  
  
    /* other list things happen... */  
}
```

Nick
Kylie
Trip



Traversal utility functions

- Freeing a linked list
- Printing a linked list
- **Measuring the length of a list**
 - We'll go over this is as a warmup on Friday!

Summary

Linked lists can be used in standalone utility functions or in the context of classes!

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?

Linked list traversal takeaways

- Temporary pointers into lists are very helpful!
 - When processing linked lists iteratively, it's common to introduce pointers that point to cells in multiple spots in the list.
 - This is particularly useful if we're destroying or rewiring existing lists.
- Using a **while** loop with a condition that checks to see if the current pointer is **nullptr** is the prevailing way to traverse a linked list.

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

Life after CS106B!

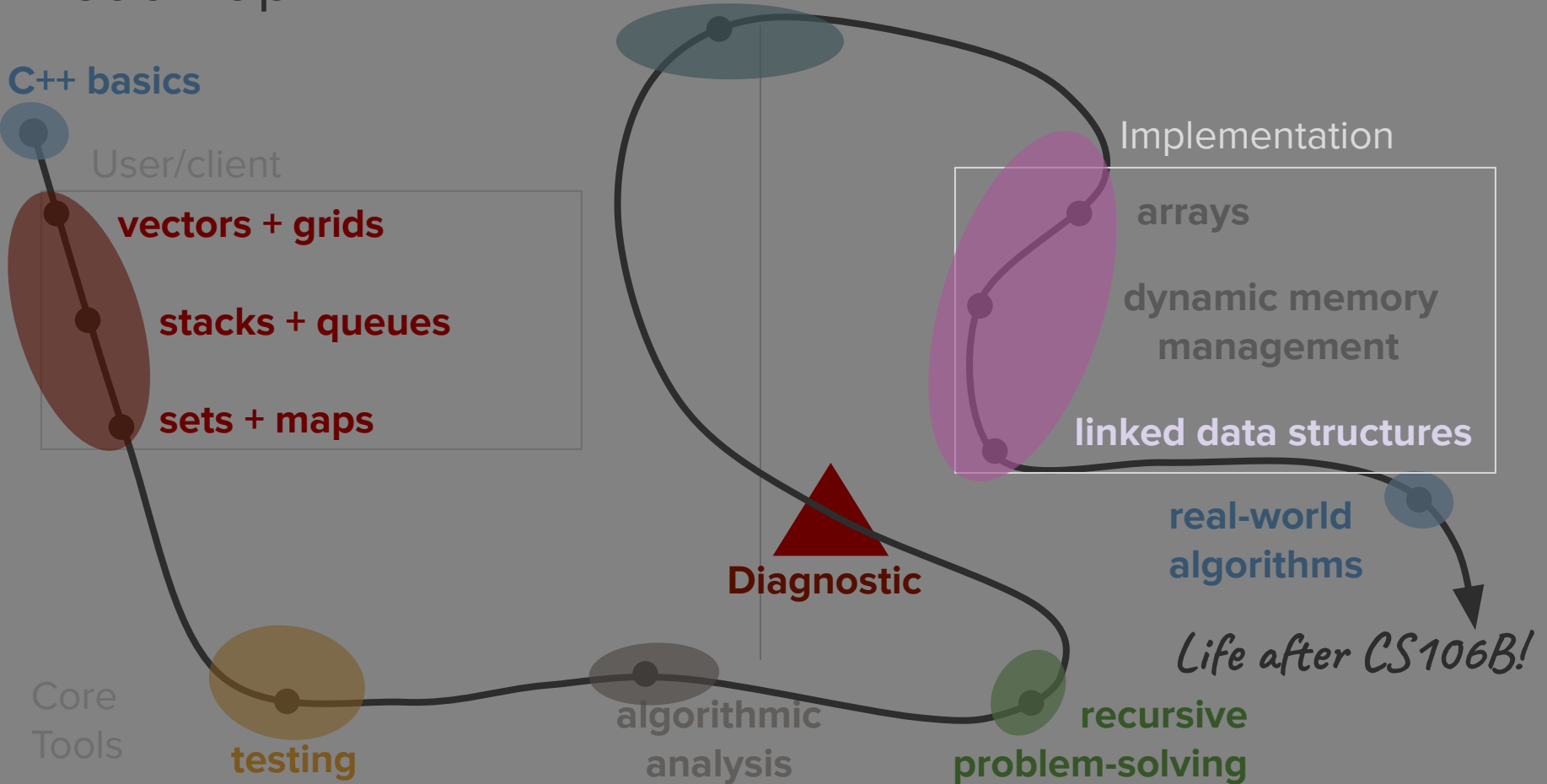
Core Tools

testing

algorithmic analysis

recursive problem-solving

Diagnostic



More on linked lists!



AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?

