# Algorithms, C, Systems

**Lecture 4: Memory, malloc, structs, etc.**

**CS 5006**
**Spring 2019**
**Seattle**

Adrienne Slaughter, Ph.D.
ahslaughter@northeastern.edu

Joe Buck
jbuck@northeastern.edu

# ASK ME IF I'm RECORDING!!

# Agenda

- Review from last time/summarize
- More structs
- Malloc & Free
- Review:
  - Abstract Data Types
  - Queues
  - Compare/contrast Linked Lists, Stacks, Queues

# MORE MALLOC & FREE

# Pattern 1: Declare variable, function to initialize

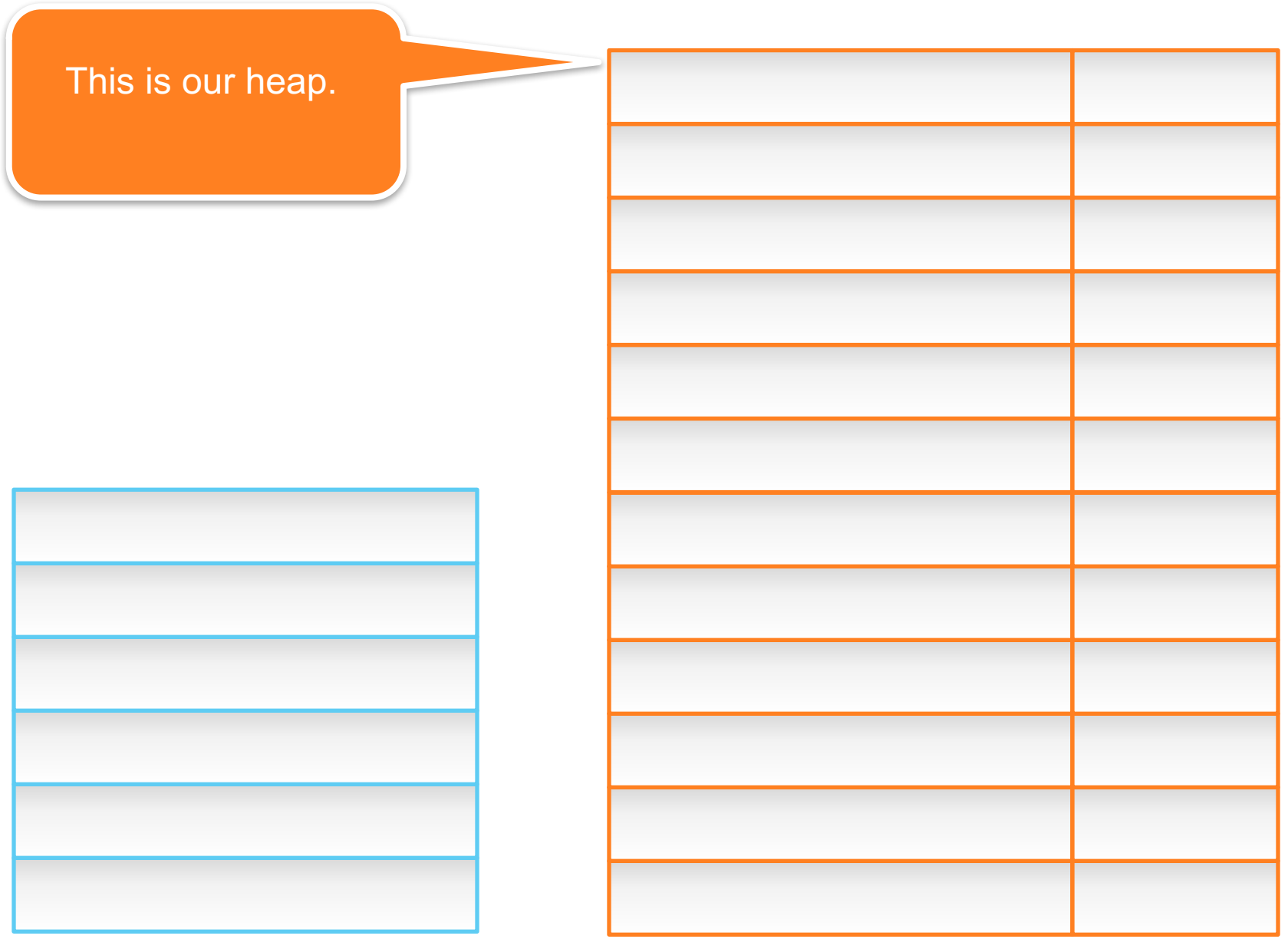
- Pros? Cons?

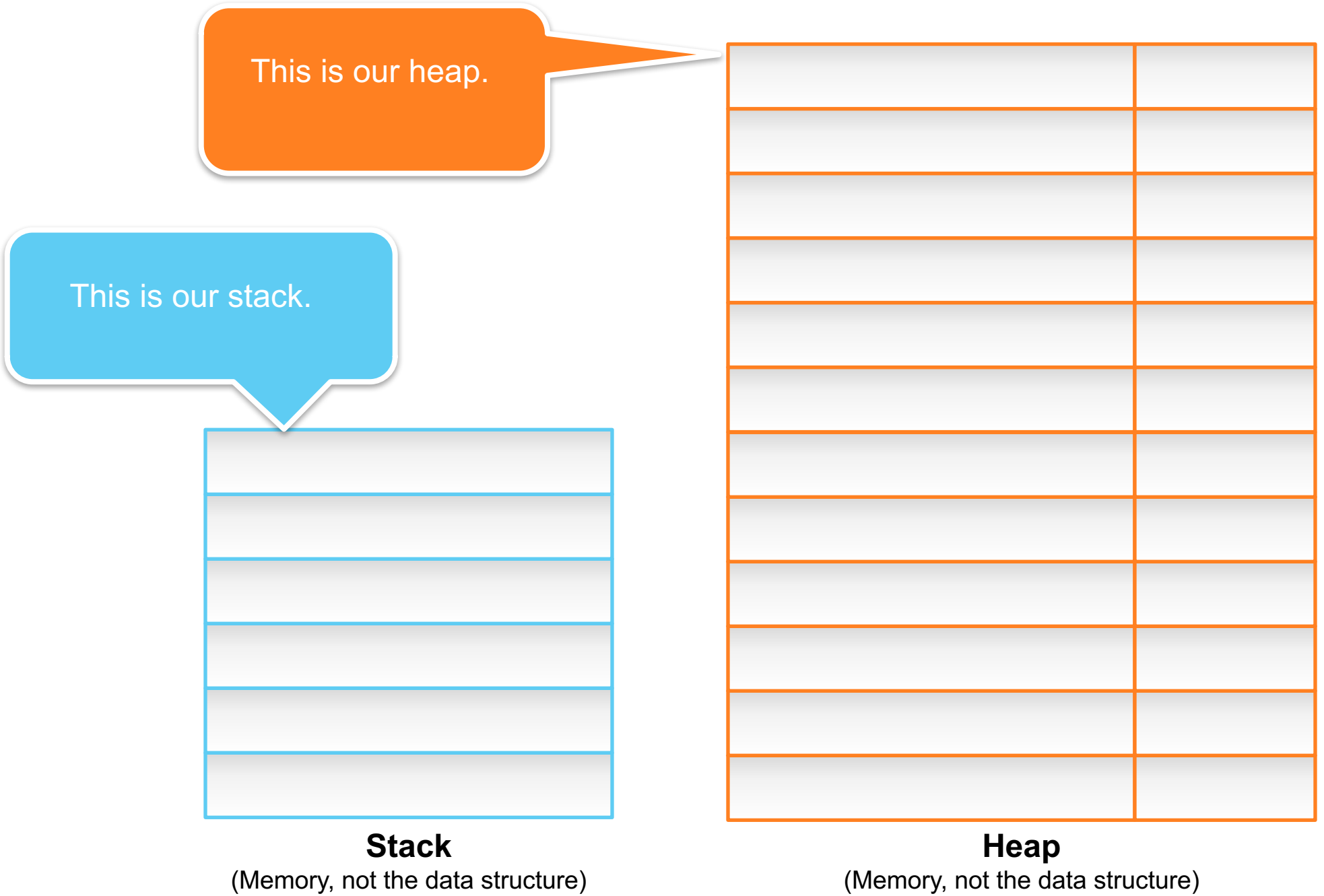
# Pattern 2:

- Write a function to declare and initialize a variable,
- Return a pointer to it
- Free the pointer later

# Memory: Storing Variables

- **Global:** available to the whole program
- **Local (stack):** available "in scope"– in a function
- **Heap:** available to the program

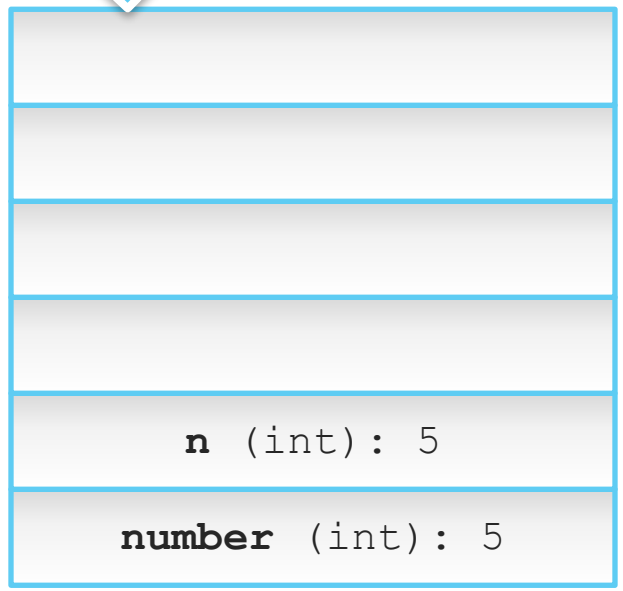

This is our heap.
Stack
(Memory, not the data structure)
Heap
(Memory, not the data structure)


This is our heap.
This is our stack.
Stack
(Memory, not the data structure)
Heap
(Memory, not the data structure)

```
10  int number = 5;
11
12  printf("The original value of the number is %d\n", number);
13  number = cubeByValue(number);
```

This is our stack.

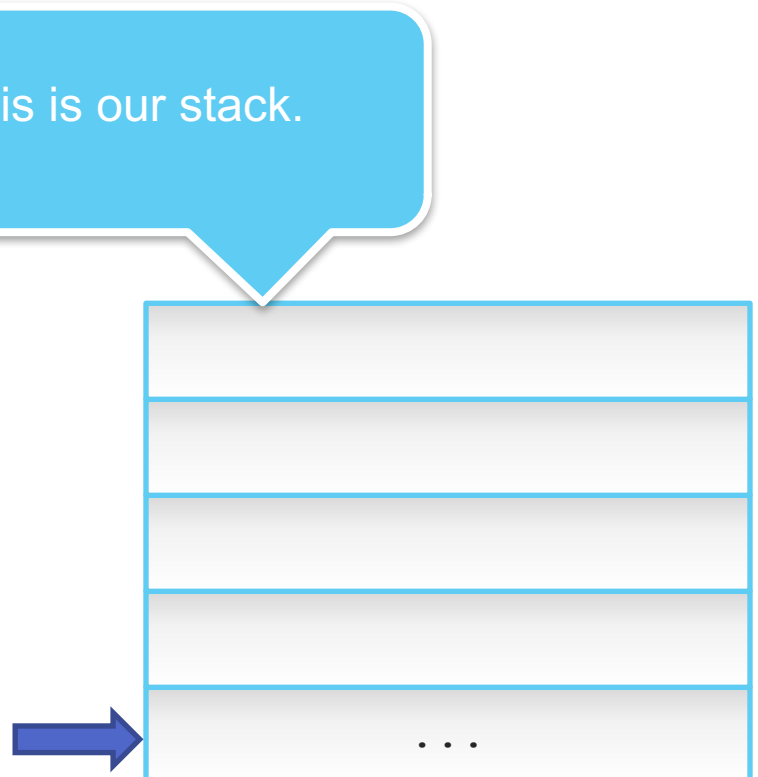
```
19int cubeByValue(int n) {
20   return n*n*n;
21}
```




**Stack**

(Memory, not the data structure)

```
14Thing createThing() {
15  Thing thing;
16  thing.age = 4;
17  thing.value = 750;
18  return thing;
19}
```

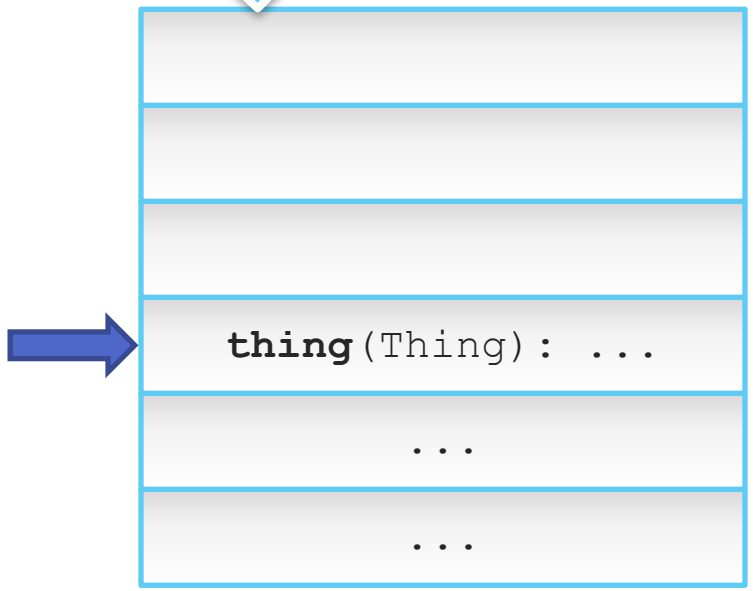



**Stack**

(Memory, not the data structure)

This is our stack.

```
14Thing createThing() {
15  Thing thing;
16  thing.age = 4;
17  thing.value = 750;
18  return thing;
19}
```

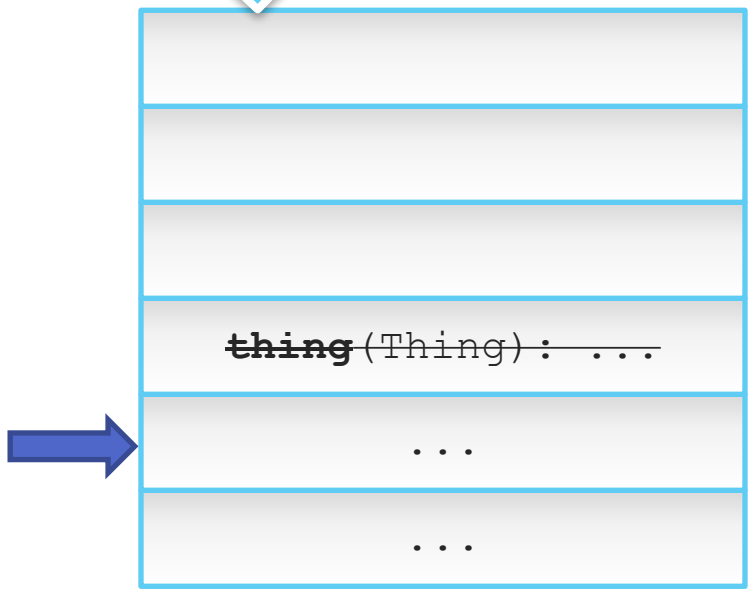



**Stack**

(Memory, not the data structure)

This is our stack.

```
14Thing createThing() {
15  Thing thing;
16  thing.age = 4;
17  thing.value = 750;
18  return thing;
19}
```



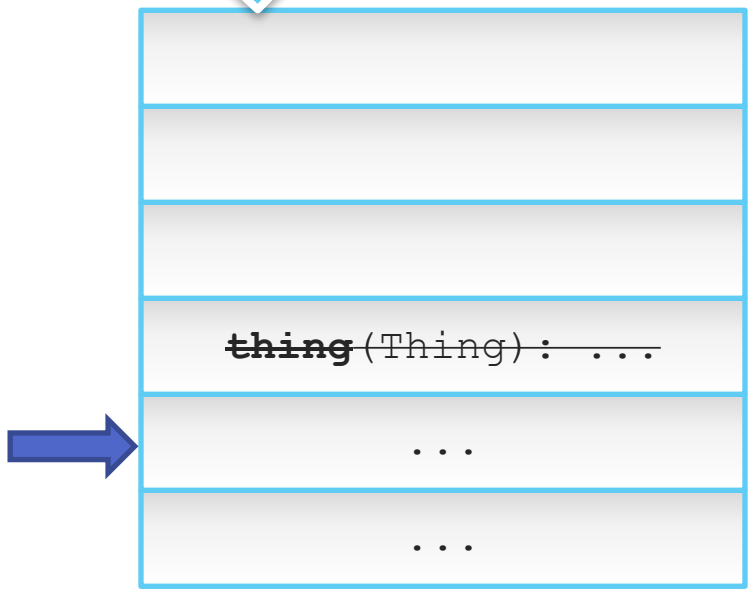

**Stack**

(Memory, not the data structure)

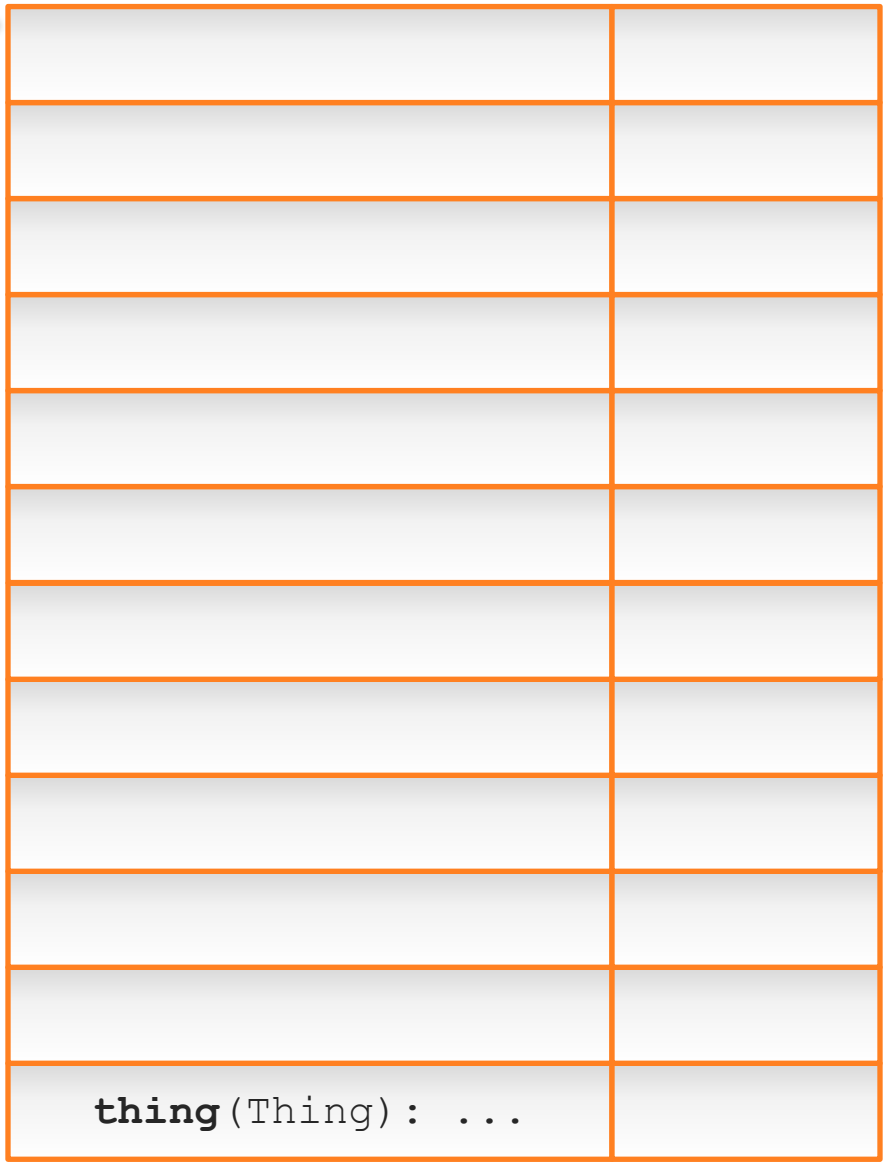
This is our heap.

This is our stack.

```
14Thing createThing() {
15  Thing thing;
16  thing.age = 4;
17  thing.value = 750;
18  return thing;
19}
```




**Stack**
(Memory, not the data structure)




**Heap**
(Memory, not the data structure)

# Malloc:

- Allocates memory "on the heap".
- Returns a value of type void*
- Cast it into the pointer type you care about
- Don't forget to free() the pointer when you're done with it

# A little deeper on malloc & free

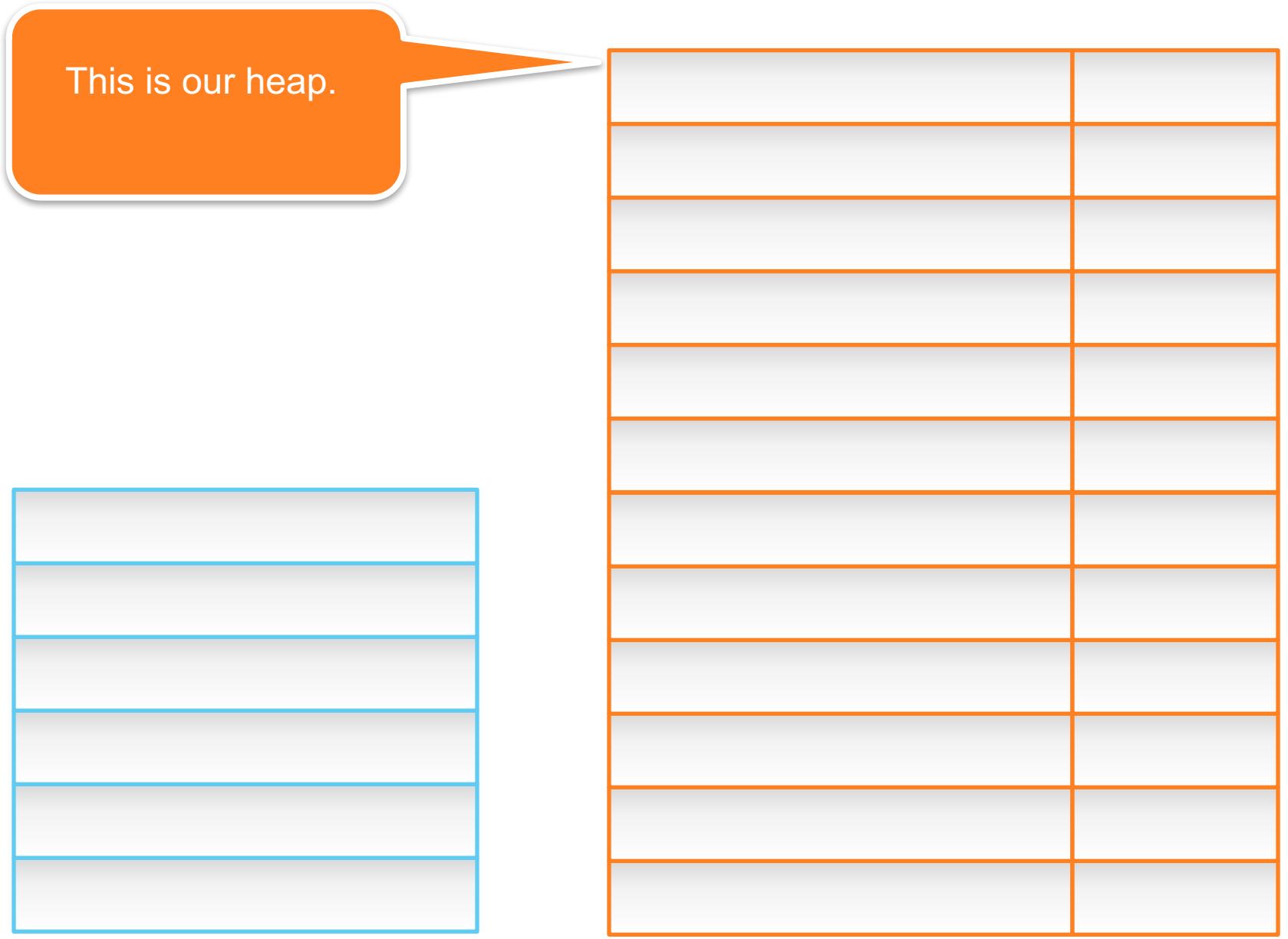
- These slides illustrate what happens on the stack and the heap when running some code

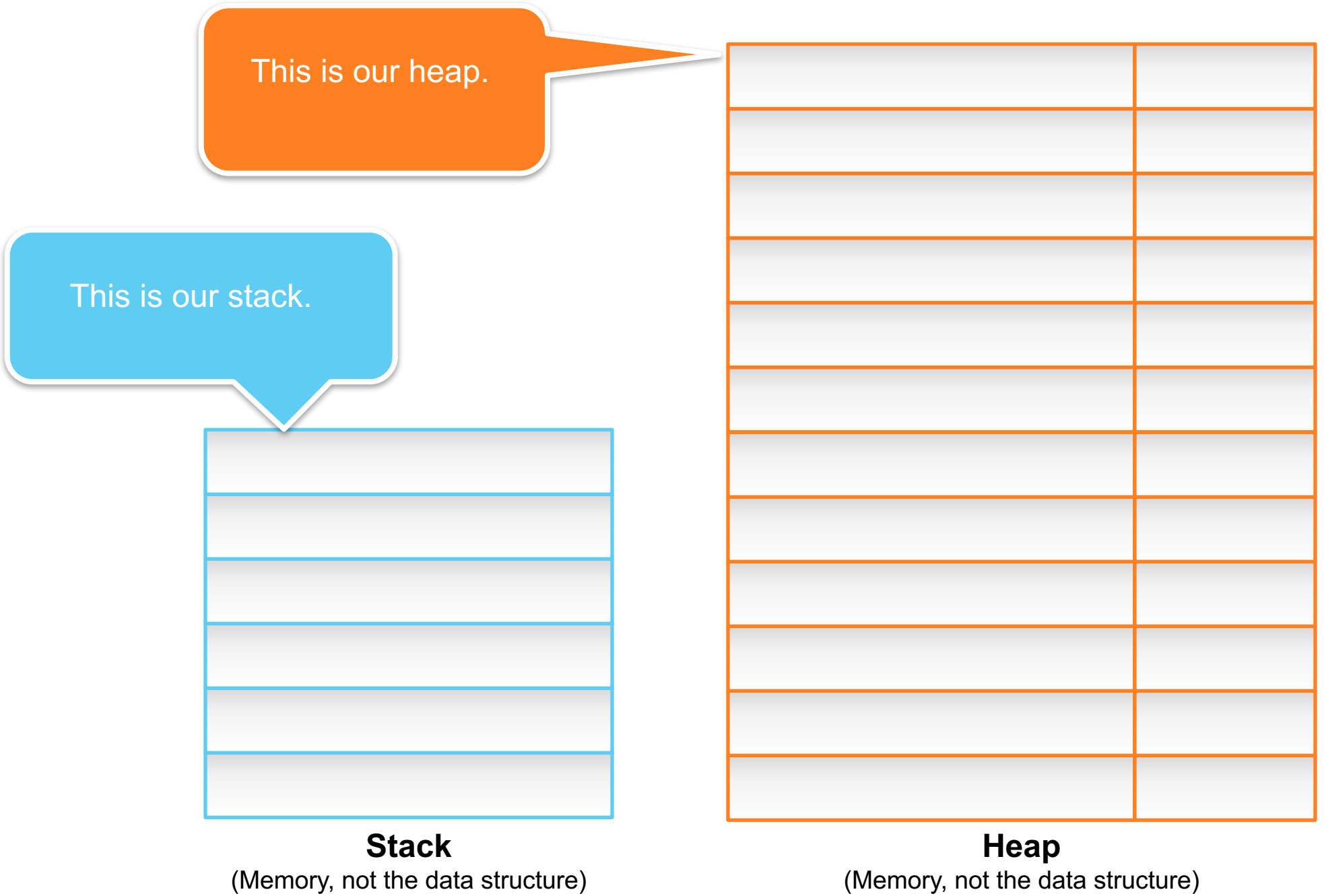
# For reference, here's the code we start with:

```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Node";
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes recently created: \n");
        printLinkedList(list);
        printf("\n-------\n");
}
```

```
struct Node{
    char* data;
    Node* prevNode;
    Node* nextNode;
};
```

```
struct LinkedList{
    Node* firstNode;
};
```


This is our heap.
Stack
(Memory, not the data structure)
Heap
(Memory, not the data structure)

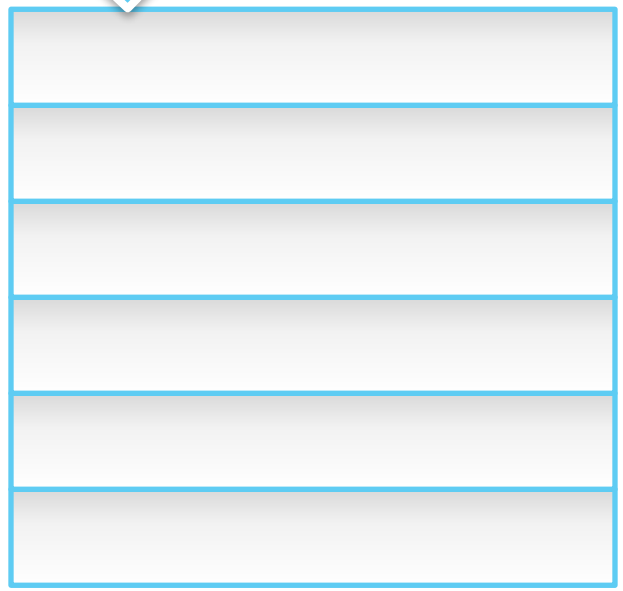

This is our heap.
This is our stack.
Stack
(Memory, not the data structure)
Heap
(Memory, not the data structure)

```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data =
        insert(list, ne
        Node *node3 = c
        node3->data = "
        insert(list, no
        printf("printing the linked li   with 3 nodes
recently created: \n");
        printLinkedList(list);
        printf("\n-------\n");
}
```

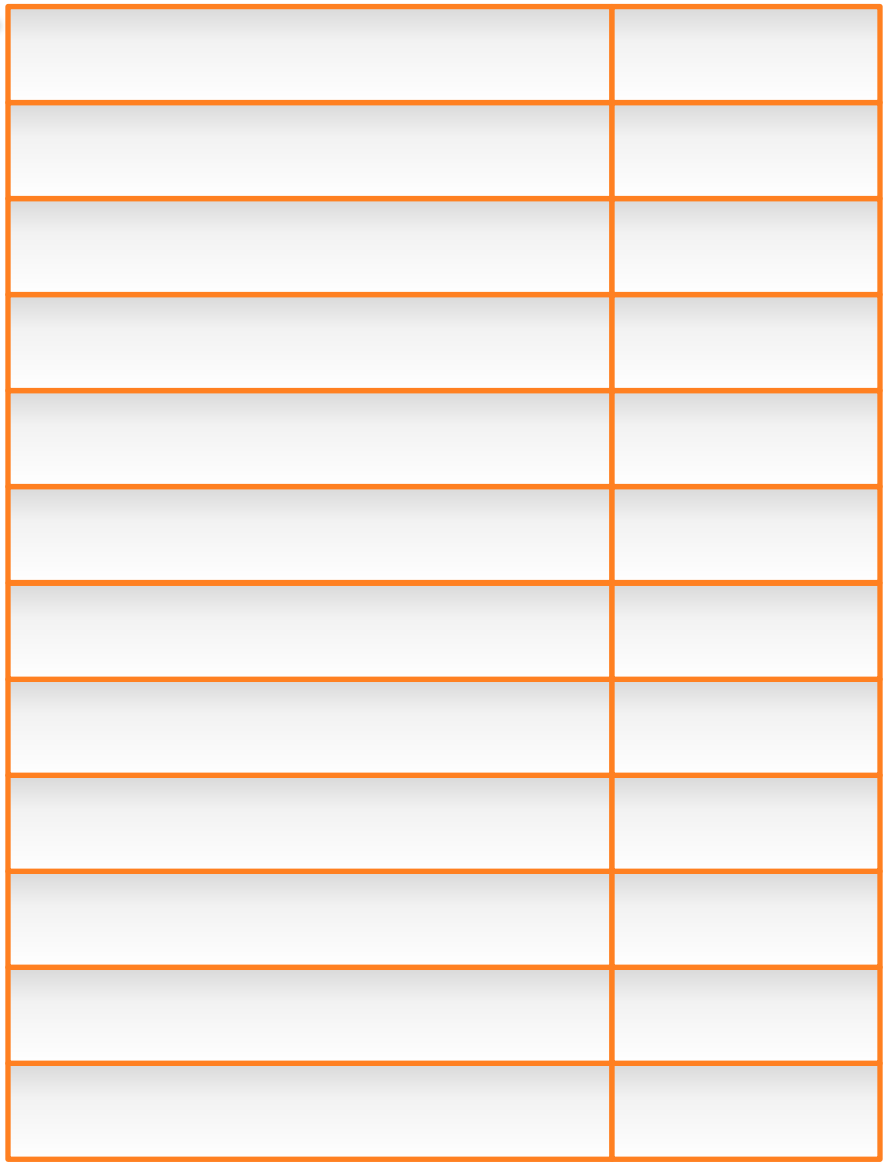
This is our code.

This is our heap.

This is our stack.



**Stack**
(Memory, not the data structure)



**Heap**
(Memory, not the data structure)

```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Nod
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(list);
        printf("\n-------\n");
}
```

When we run some code, the line is highlighted here…


**Stack**
(Memory, not the data structure)


**Heap**
(Memory, not the data structure)

```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Nod
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(list);
        printf("\n-------\n");
}
```

When we run some code, the line is highlighted here… …and the memory changes are highlighted here.


**Stack**
(Memory, not the data structure)

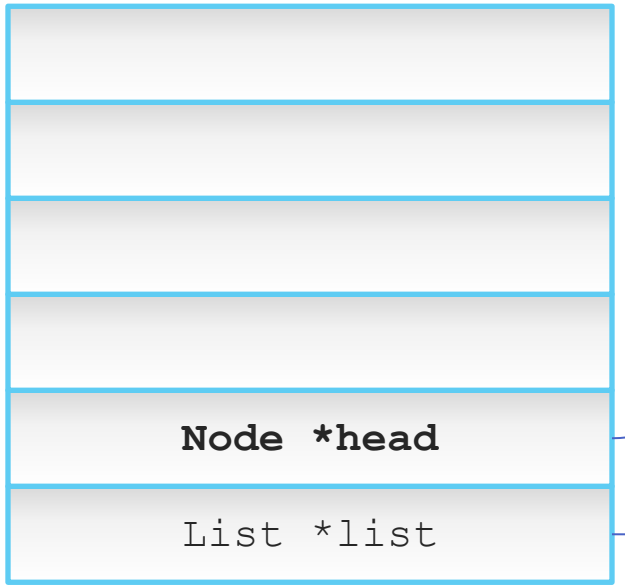

**Heap**
(Memory, not the data structure)

```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Node";
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(list);
        printf("\n-------\n");
}
```

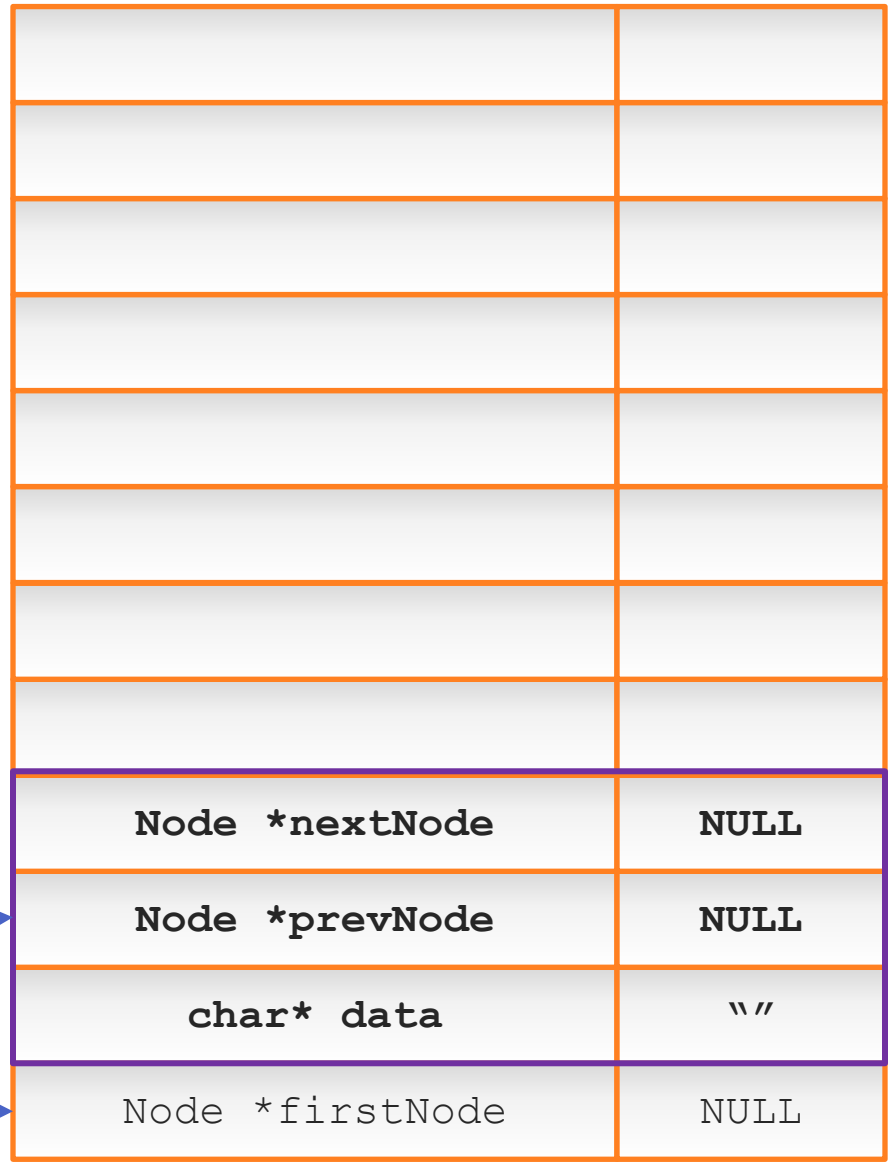

**Stack**
(Memory, not the data structure)


**Heap**
(Memory, not the data structure)

```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Node";
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(list);
        printf("\n-------\n");
}
```

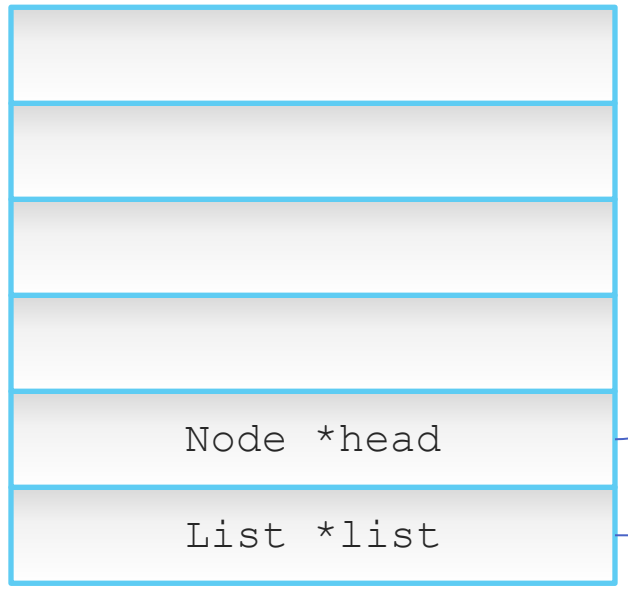



**Stack**
(Memory, not the data structure)

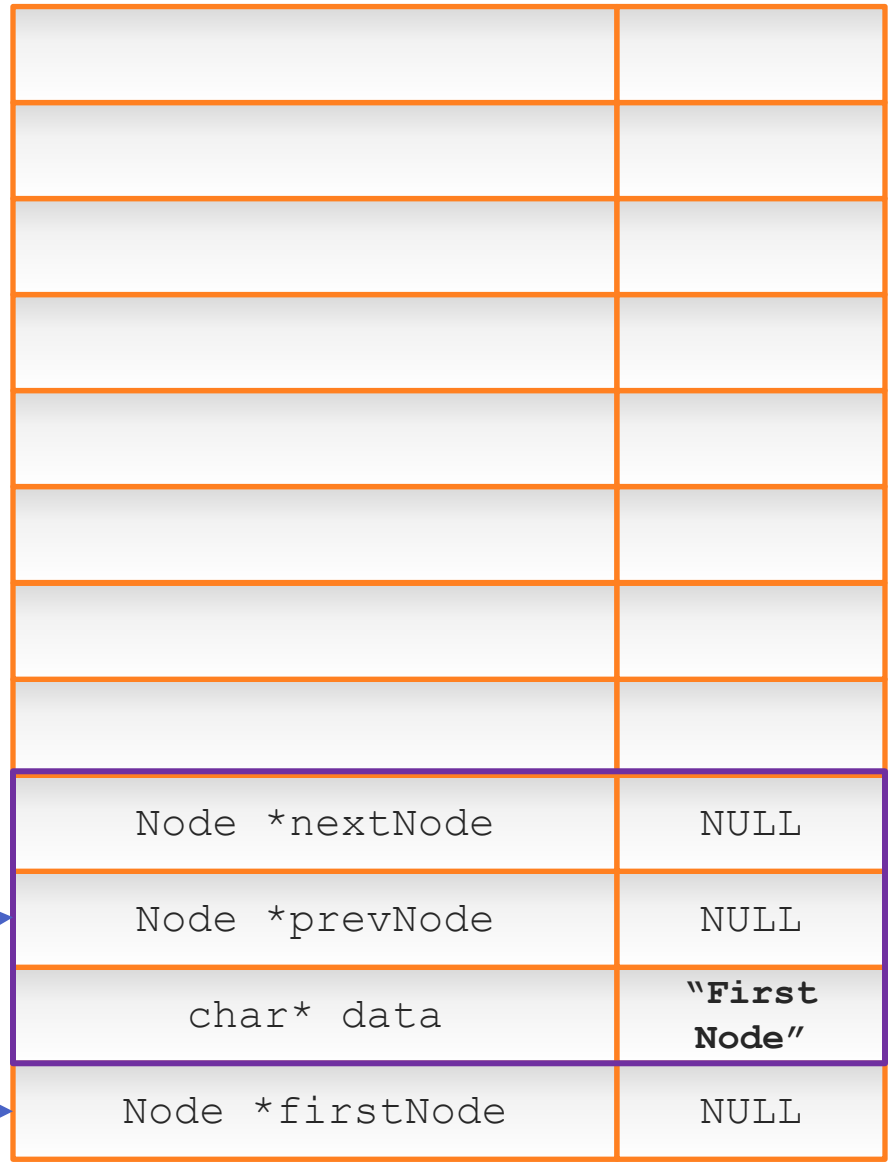



**Heap**
(Memory, not the data structure)

```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Node";
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(list);
        printf("\n-------\n");
}
```

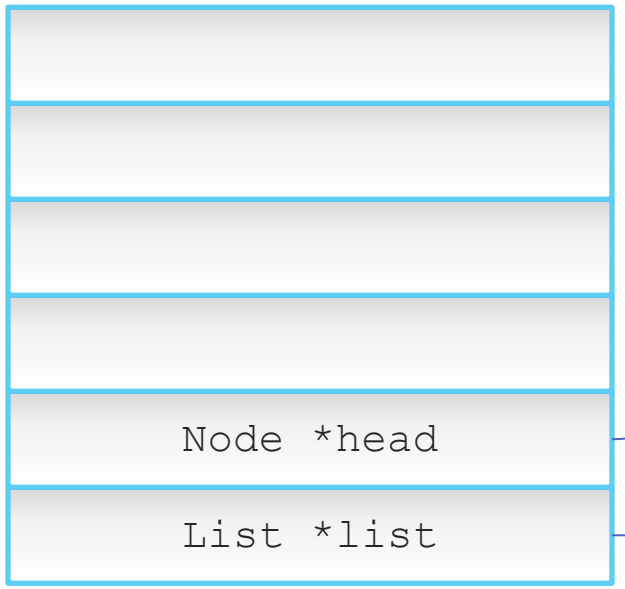



**Stack**
(Memory, not the data structure)

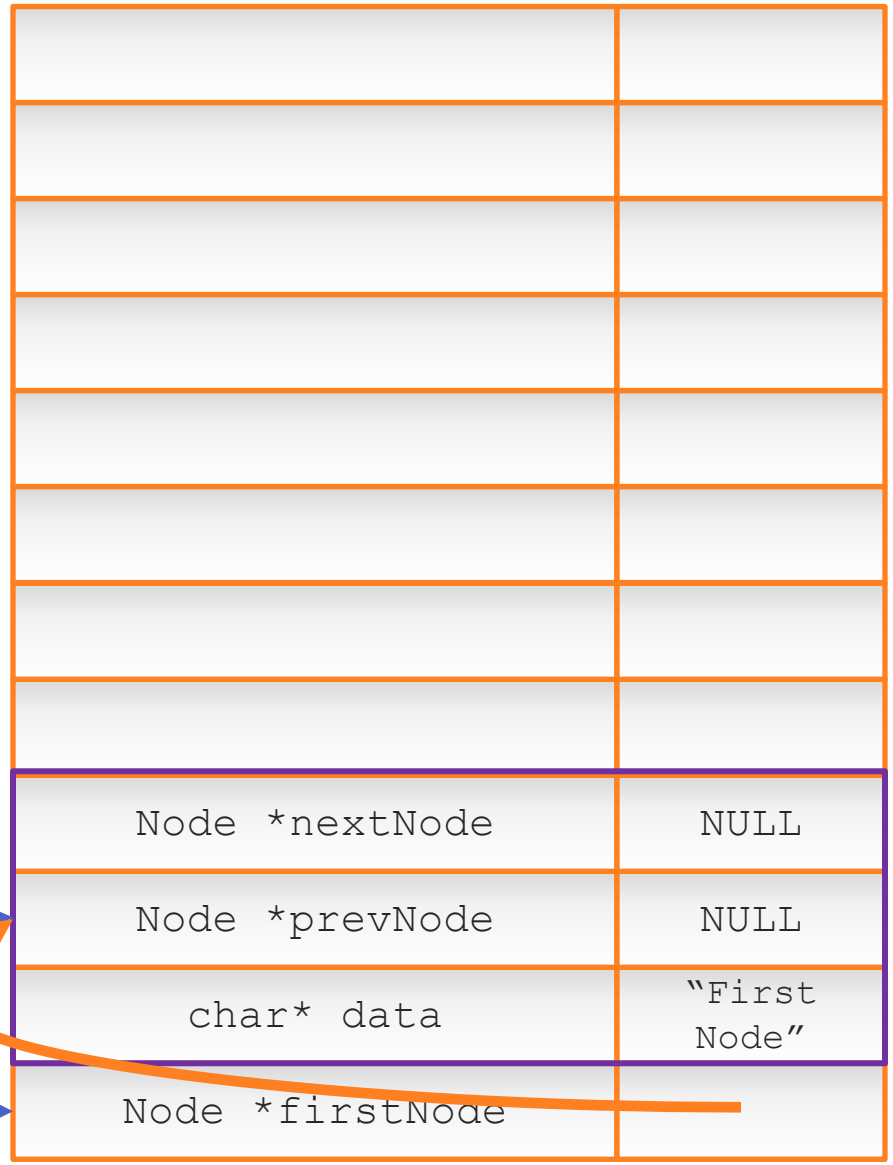



**Heap**
(Memory, not the data structure)

```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Node";
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(list);
        printf("\n-------\n");
}
```

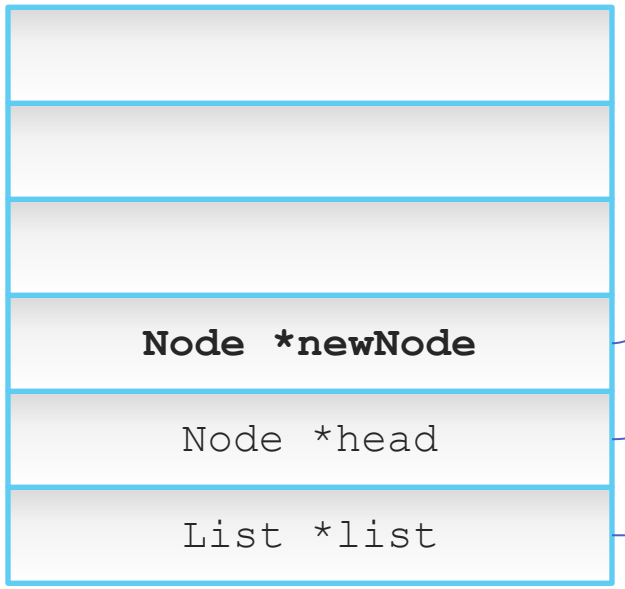



**Stack**
(Memory, not the data structure)

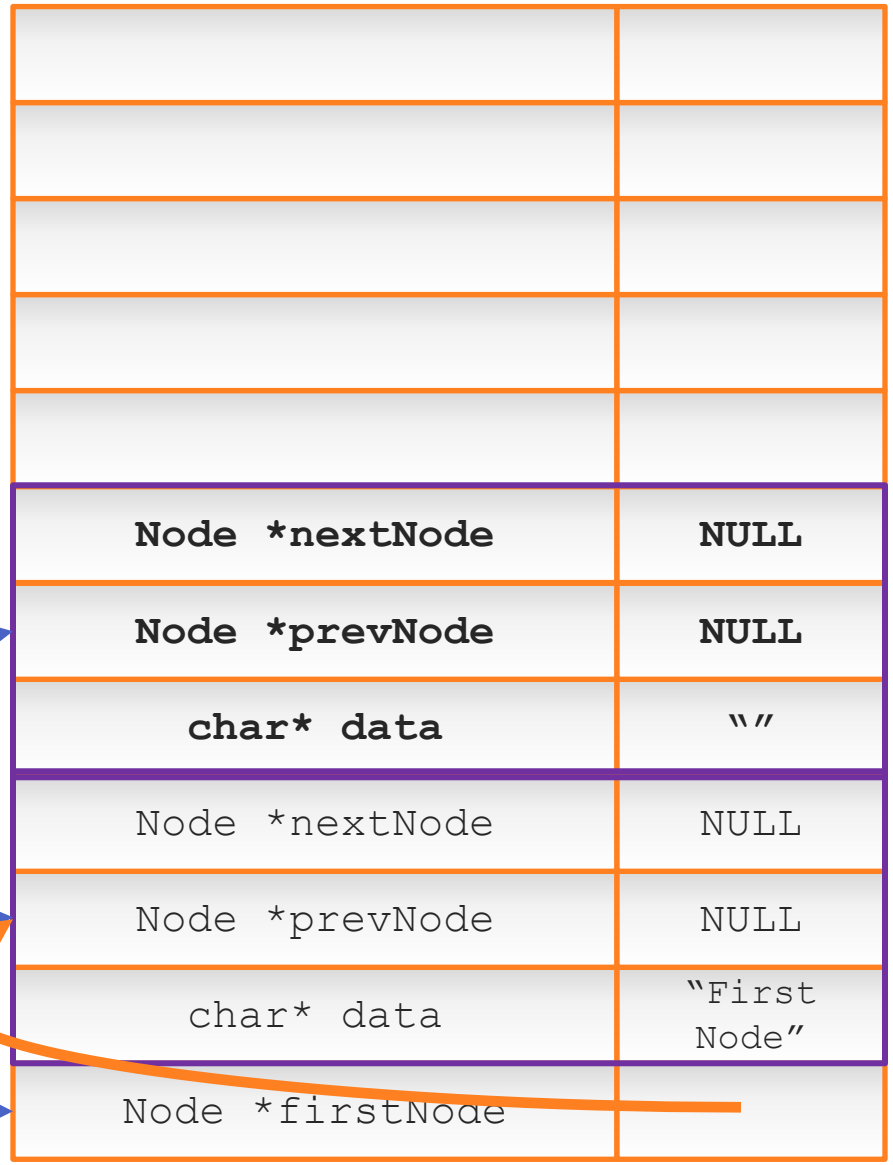



**Heap**
(Memory, not the data structure)

```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Node";
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(list);
        printf("\n-------\n");
}
```

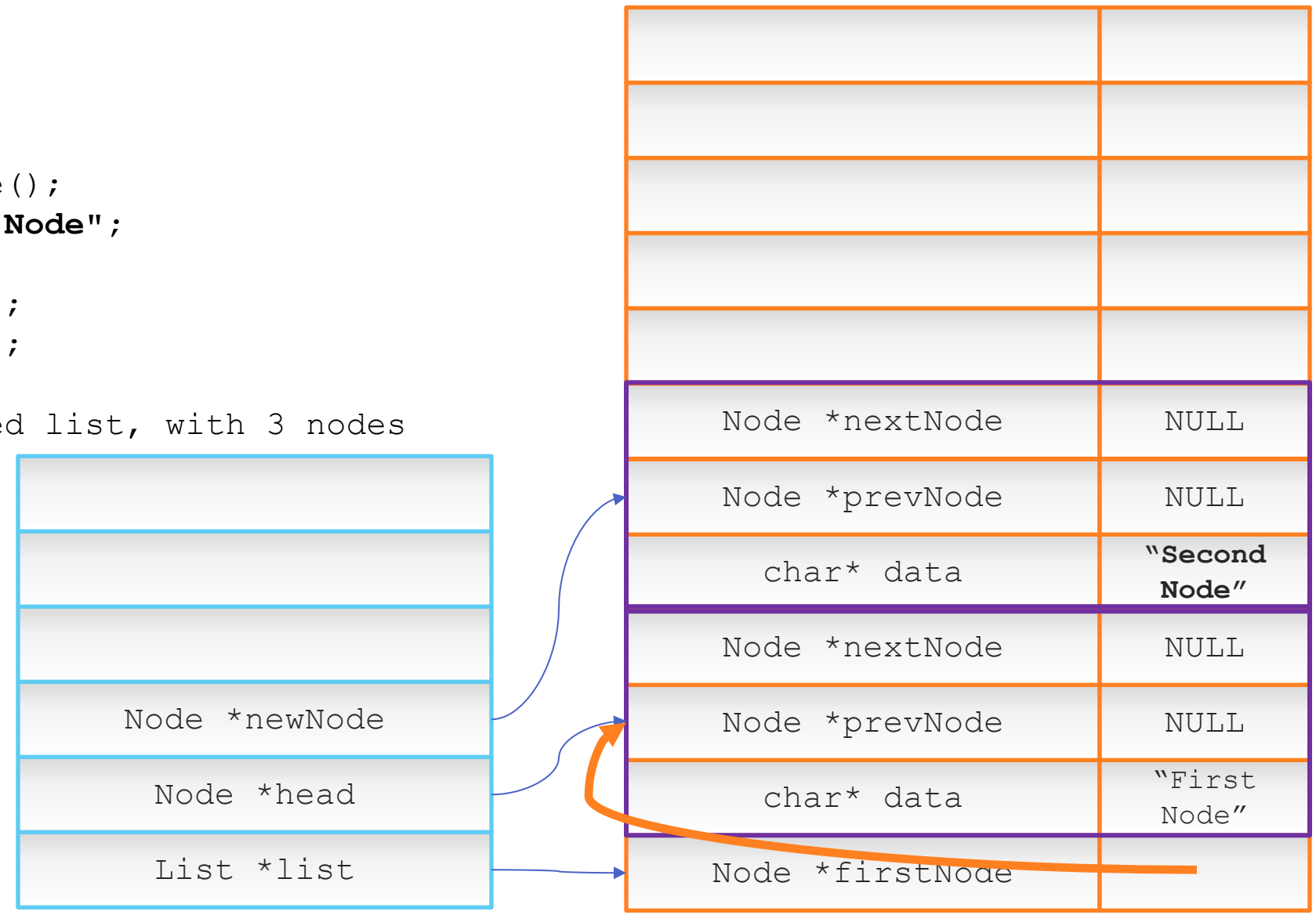



**Stack**
(Memory, not the data structure)




**Heap**
(Memory, not the data structure)

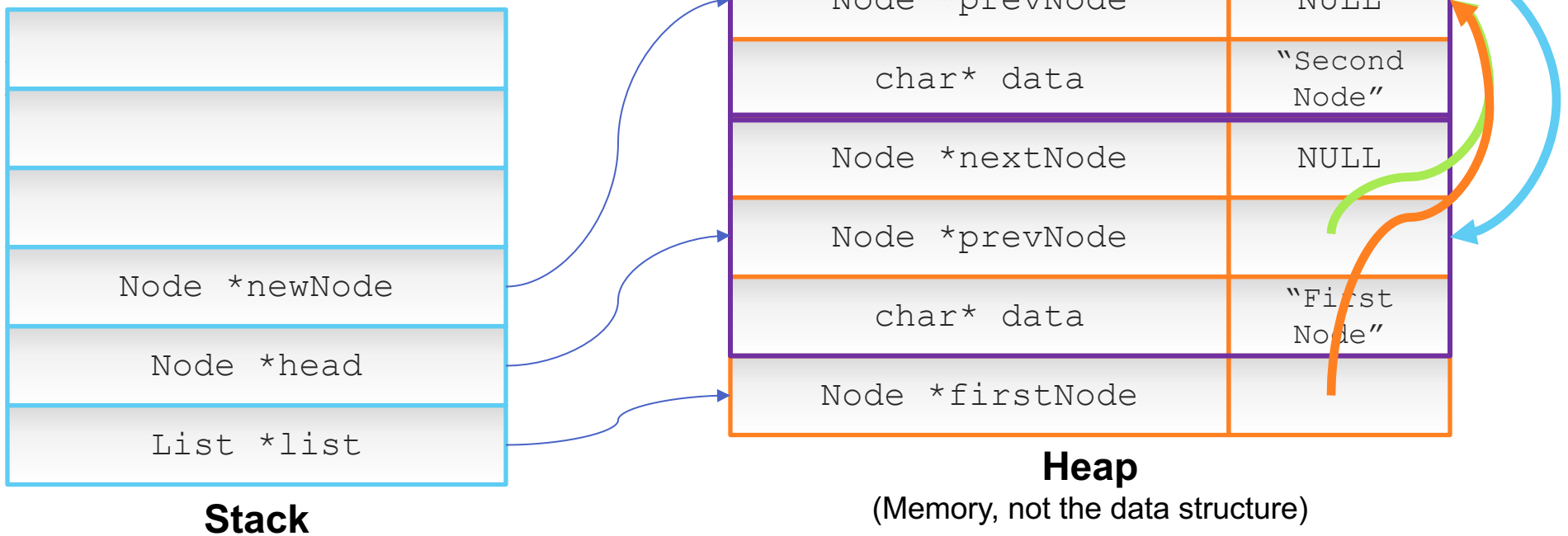
```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Node";
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(list);
        printf("\n-------\n");
}
```



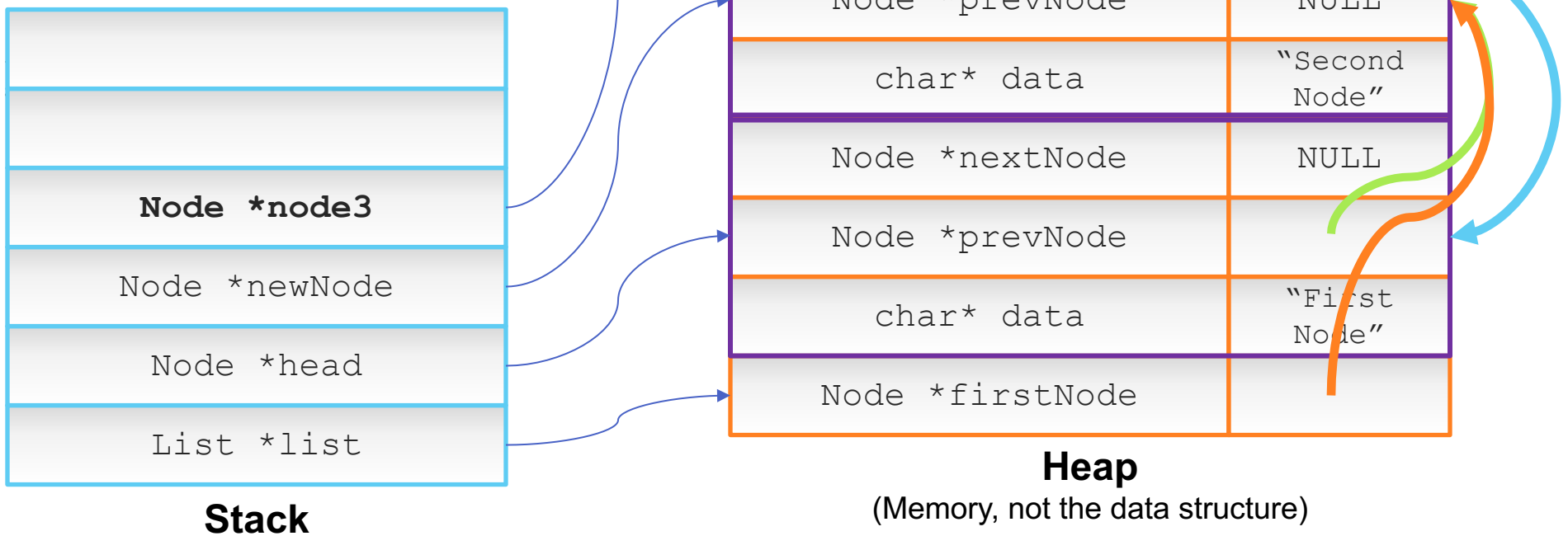

**Stack**
(Memory, not the data structure)

**Heap**
(Memory, not the data structure)

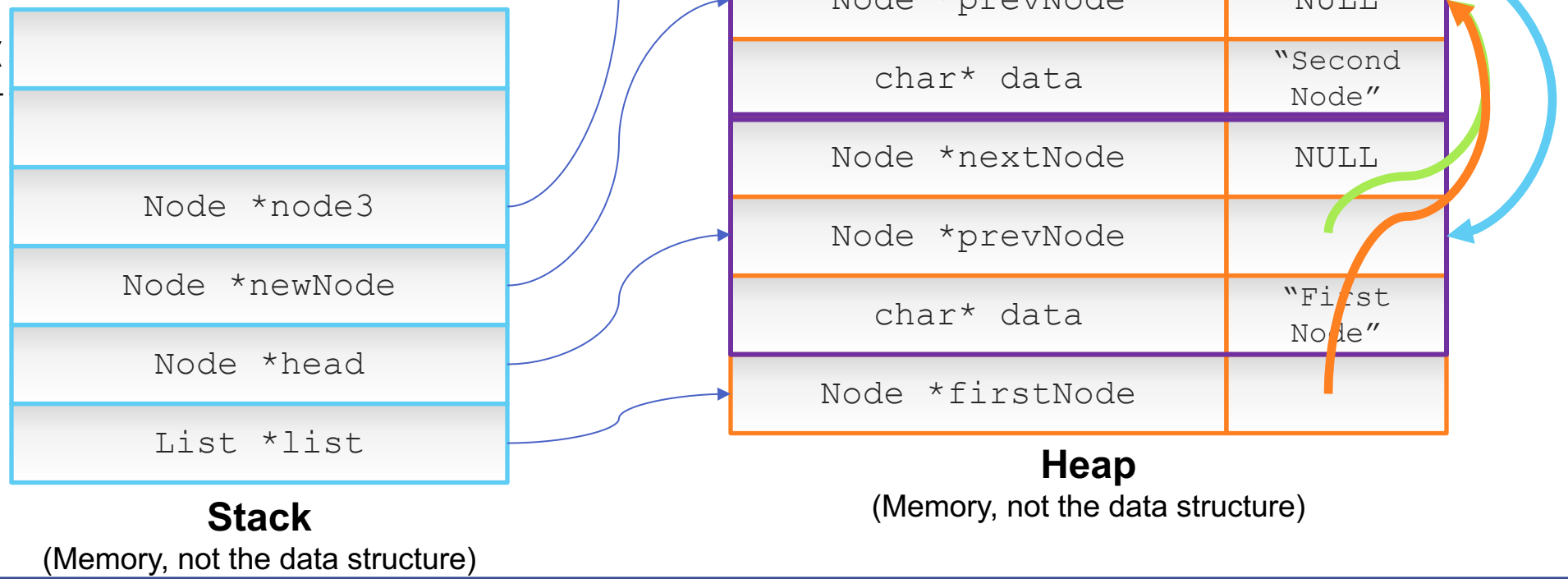
```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Node";
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(
        printf("\n------
}
```



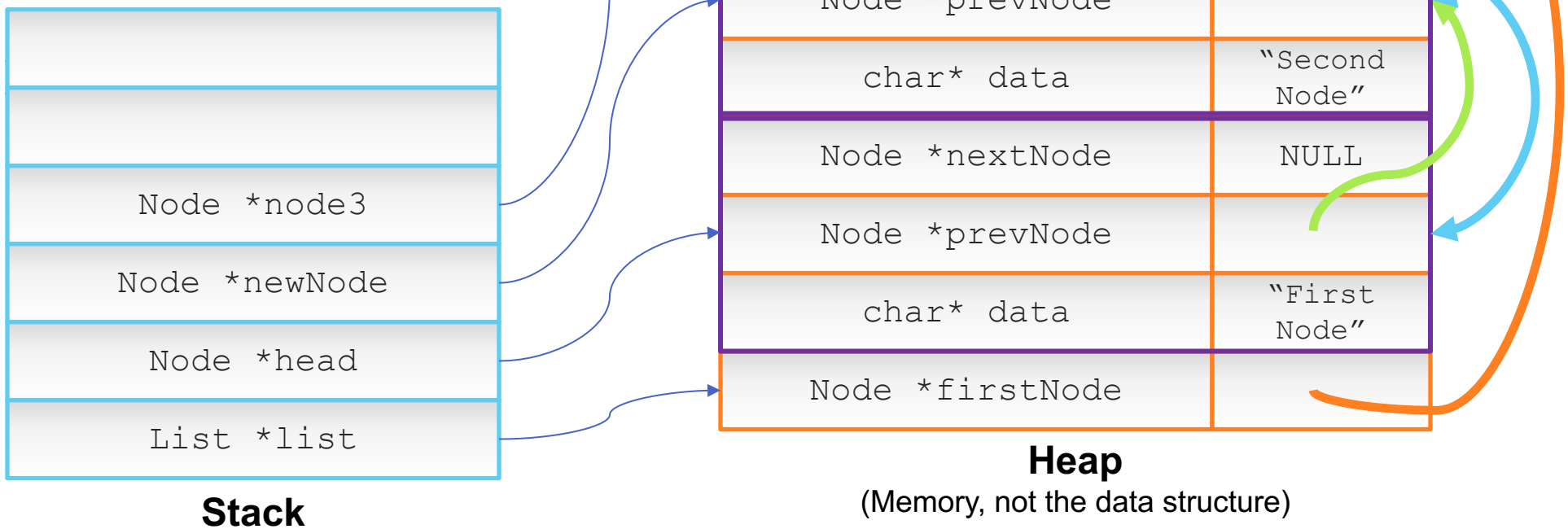
```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Node";
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(
        printf("\n------
}
```



```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Node";
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(
        printf("\n------
}
```



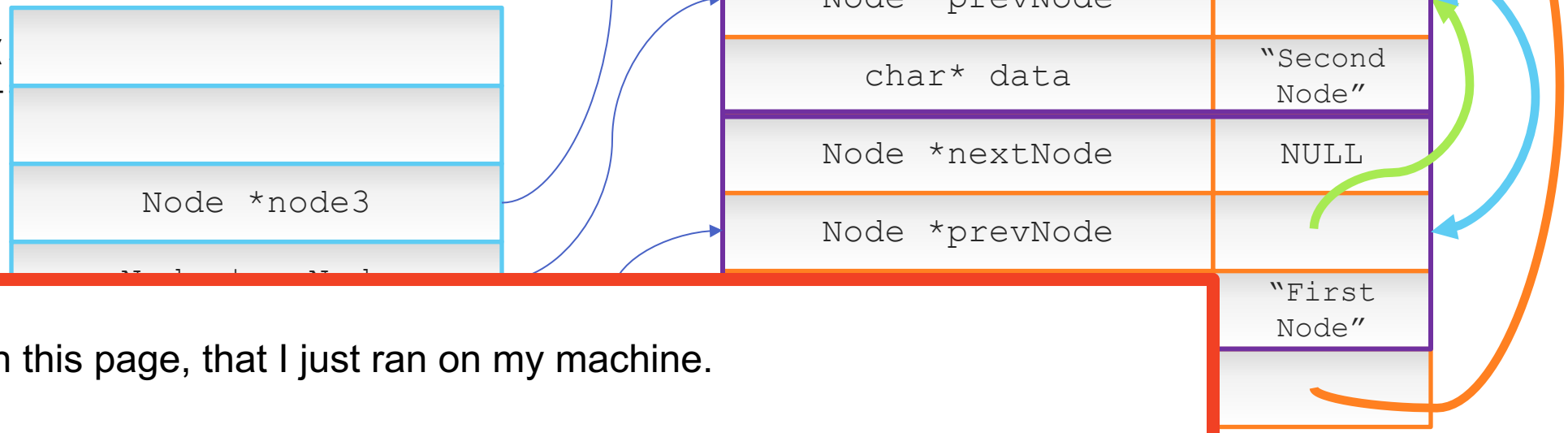
```
int main(){printf("first ");
        List *list = create();

        Node *head = createNode();
        head->data = "First Node";
        insert(list, head);

        Node *newNode = createNode();
        newNode -> data = "Second Node";
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(
        printf("\n------
}
```



```
int main(){printf("first ");
```

```
printing the linked list, with 3 nodes recently created:
0x7ff16cc02720  PrevNode:                0x0|          Third Node| NextNode:  0x7ff16cc02700
0x7ff16cc02700  PrevNode:  0x7ff16cc02720|         Second Node| NextNode:  0x7ff16cc026e0
0x7ff16cc026e0  PrevNode:  0x7ff16cc02700|          First Node| NextNode:            0x0

------
```

```
        Node *newNode = createNode();
        newNode -> data = "Second Node";
        insert(list, newNode);
        Node *node3 = createNode();
        node3->data = "Third Node";
        insert(list, node3);
        printf("printing the linked list, with 3 nodes
recently created: \n");
        printLinkedList(
        printf("\n------
}
```




This is the output of the on this page, that I just ran on my machine.

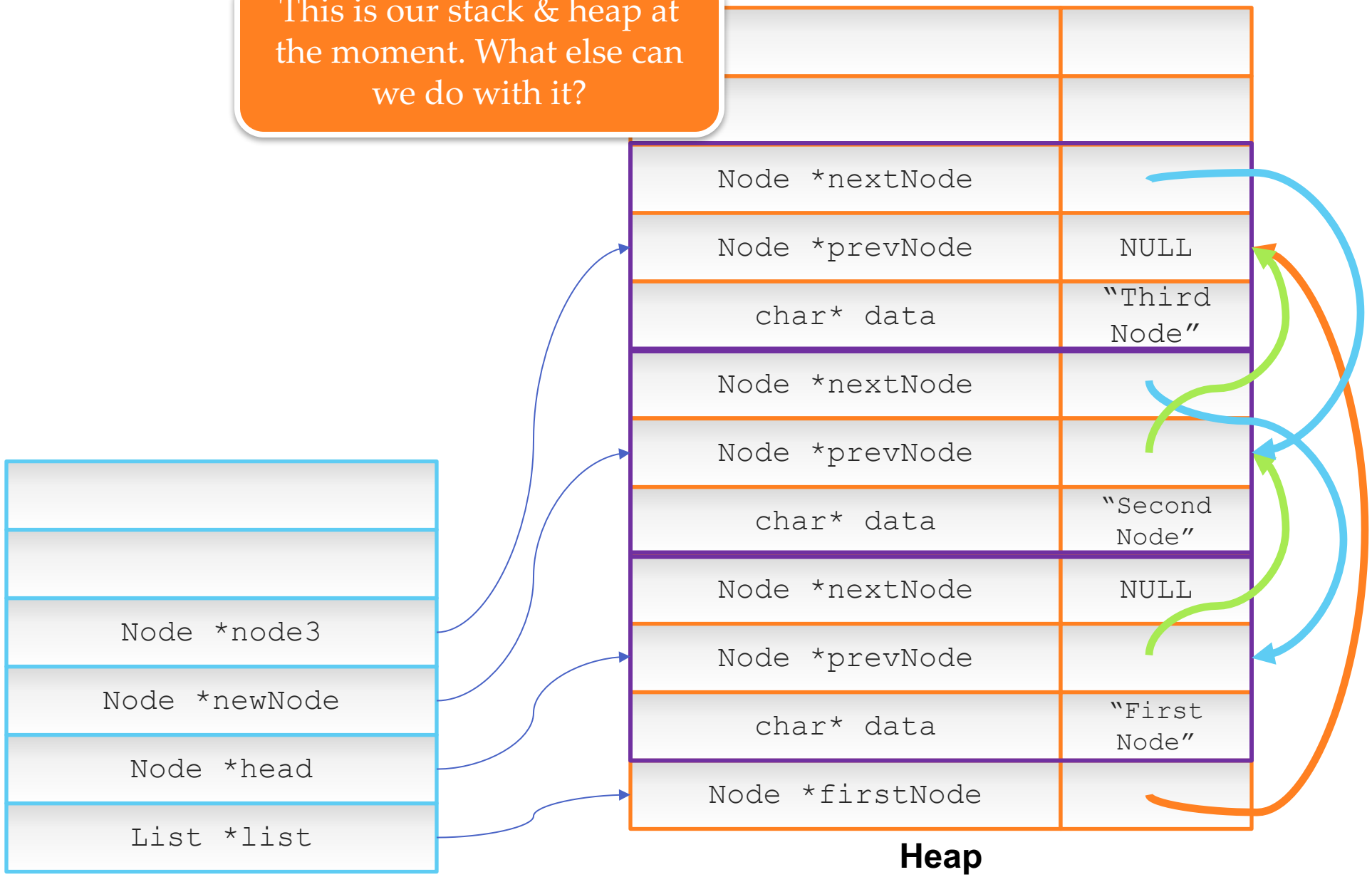
Note that this is what we should expect: Printing our list, with 3 nodes.

cture)

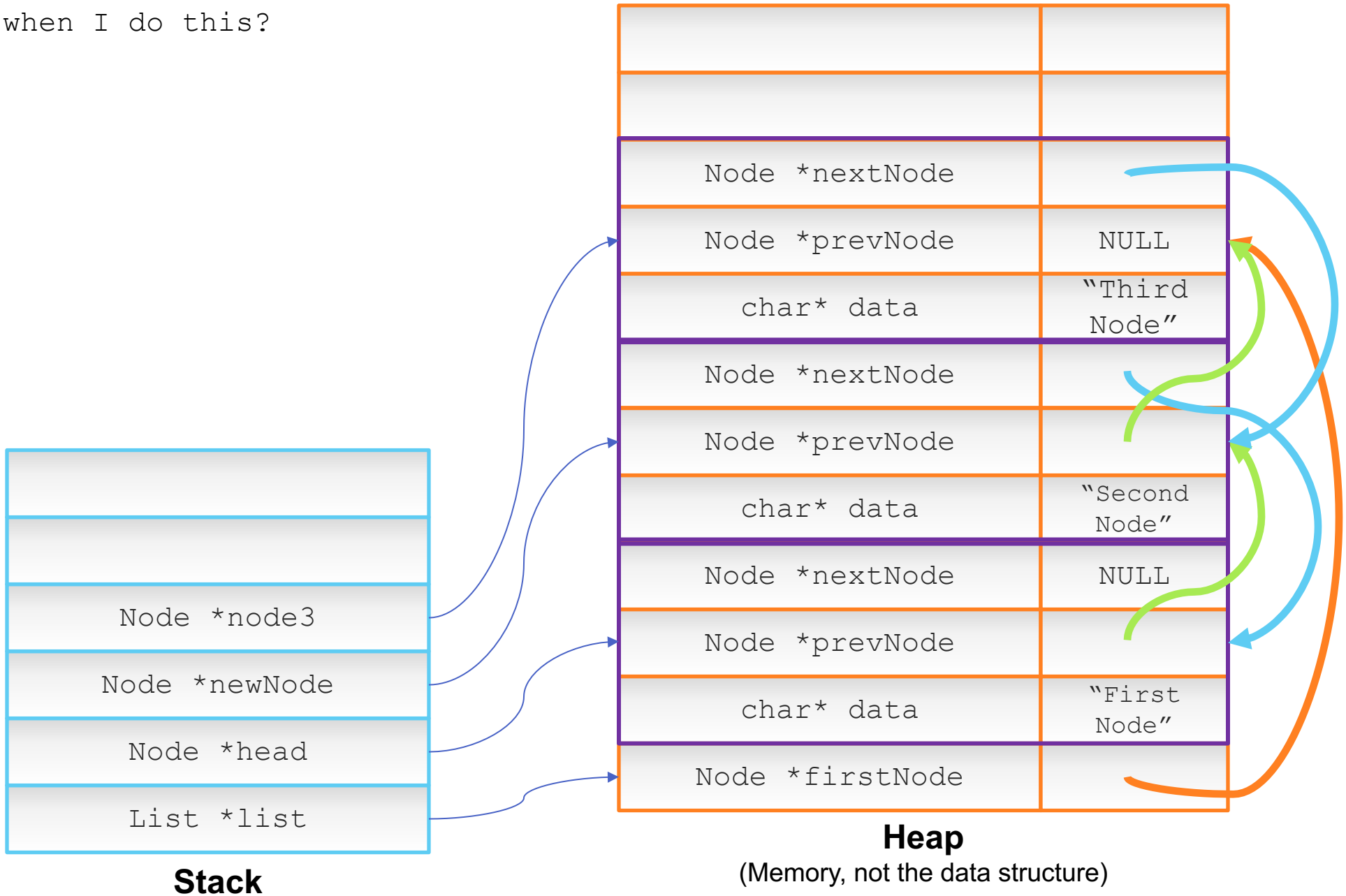
(Memory, not the data structure)

```
    …
    // Continuing on…


}
```

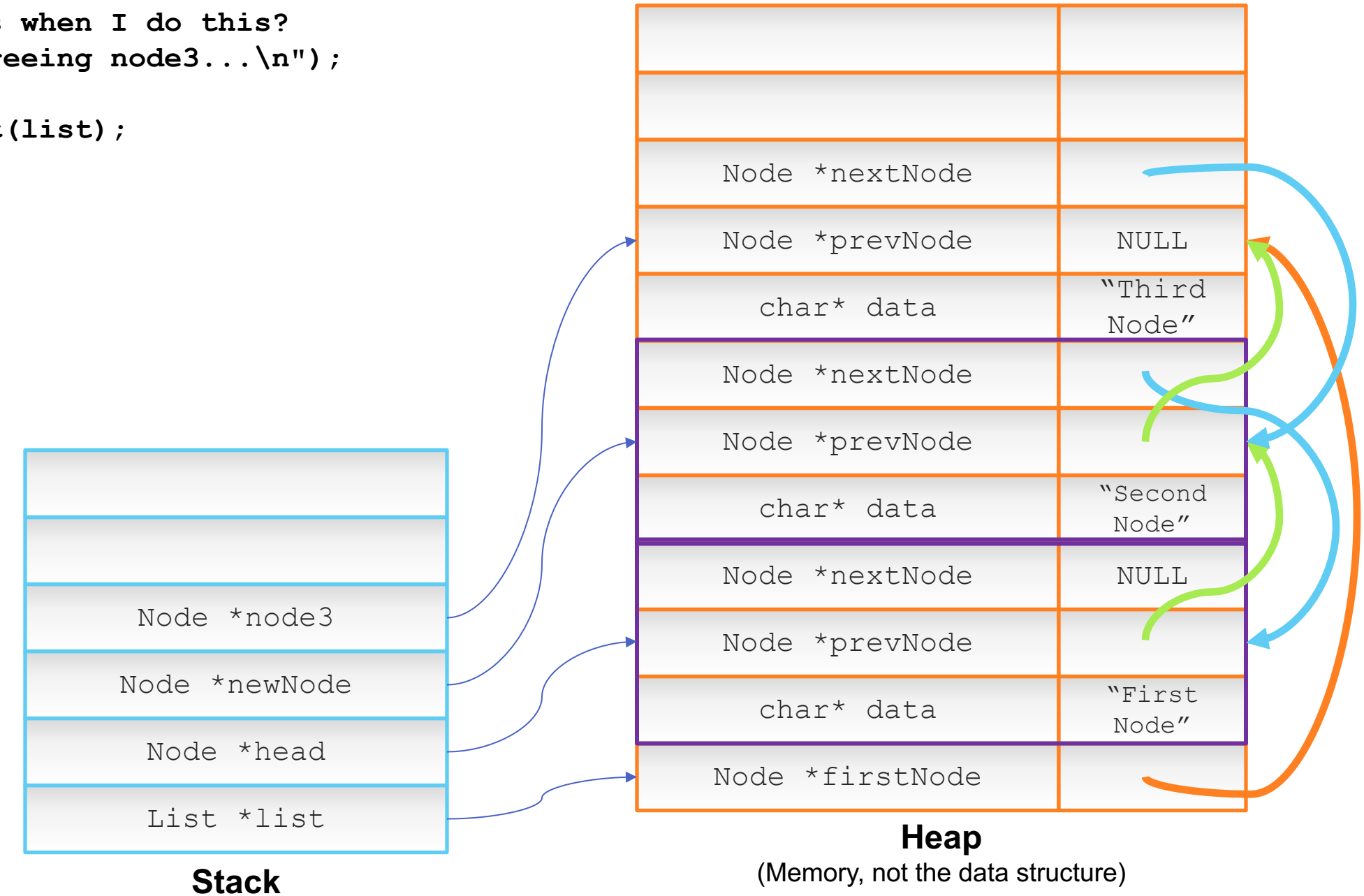
This is our stack & heap at the moment. What else can we do with it?



```
    …
    // Continuing on…
    // What happens when I do this?
    free(node3);


}
```

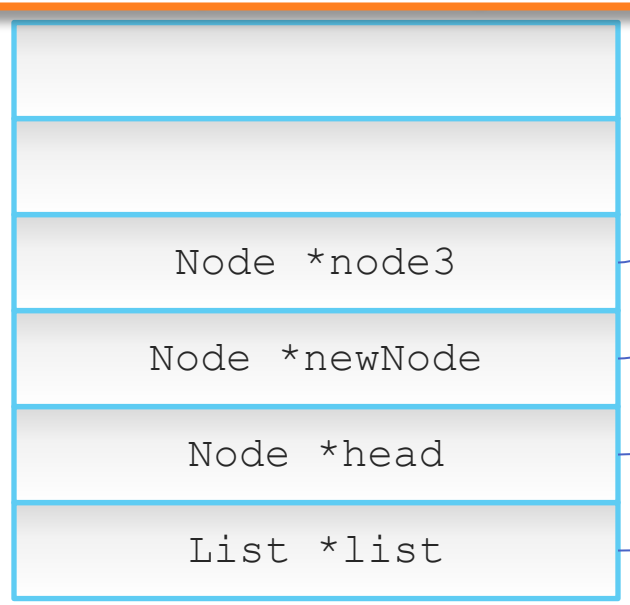


```
…
// Continuing on…
// What happens when I do this?
 printf("I'm freeing node3...\n");
free(node3);
printLinkedList(list);

}
```

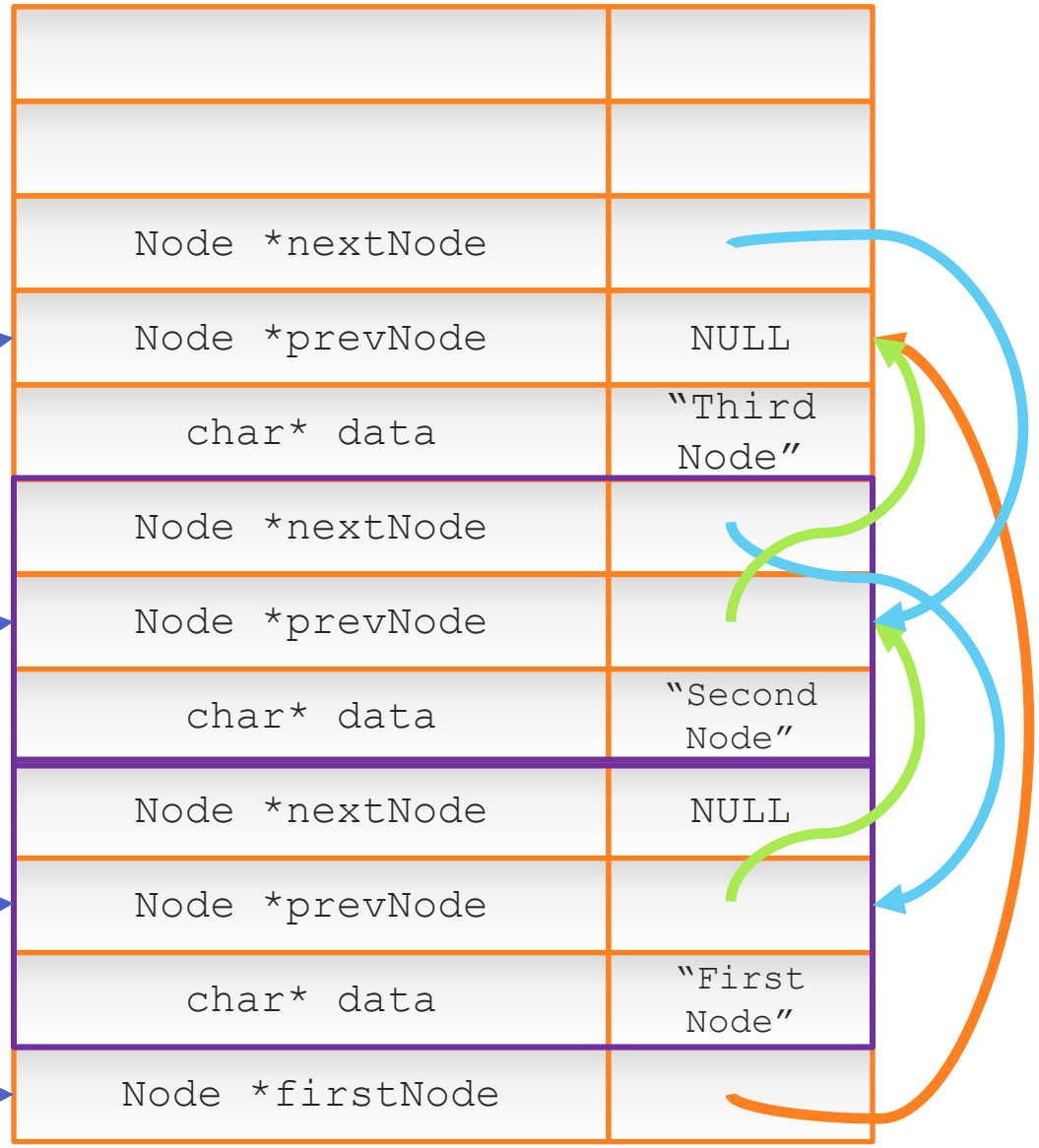


```
…
// Continuing on…
// What happens when I do this?
 printf("I'm freeing node3...\n");
free(node3);
printLinkedList(list);

}
```

→ The blue square goes away.
→ This means that the heap has deallocated the memory, or released it.
→ **NOTE:** Nothing else has changed. There are still pointers everywhere!

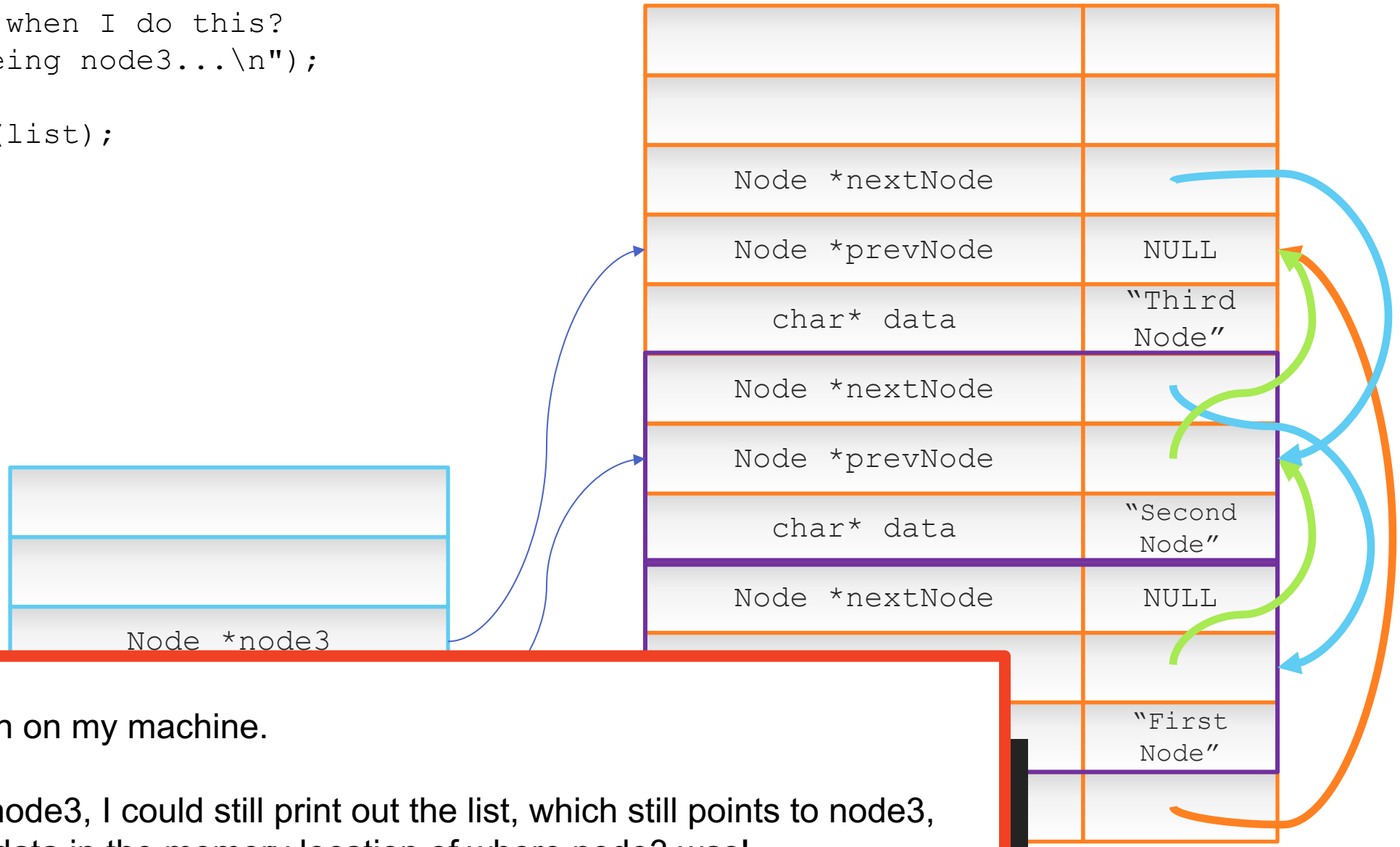



**Stack**
(Memory, not the data structure)




**Heap**
(Memory, not the data structure)

```
        …
        // Continuing on…
        // What happens when I do this?
        printf("I'm freeing node3...\n");
        free(node3);
        printLinkedList(list);

}
```




Once again, the code I ran on my machine.

We see that after I freed node3, I could still print out the list, which still points to node3, and I can still access the data in the memory location of where node3 was!

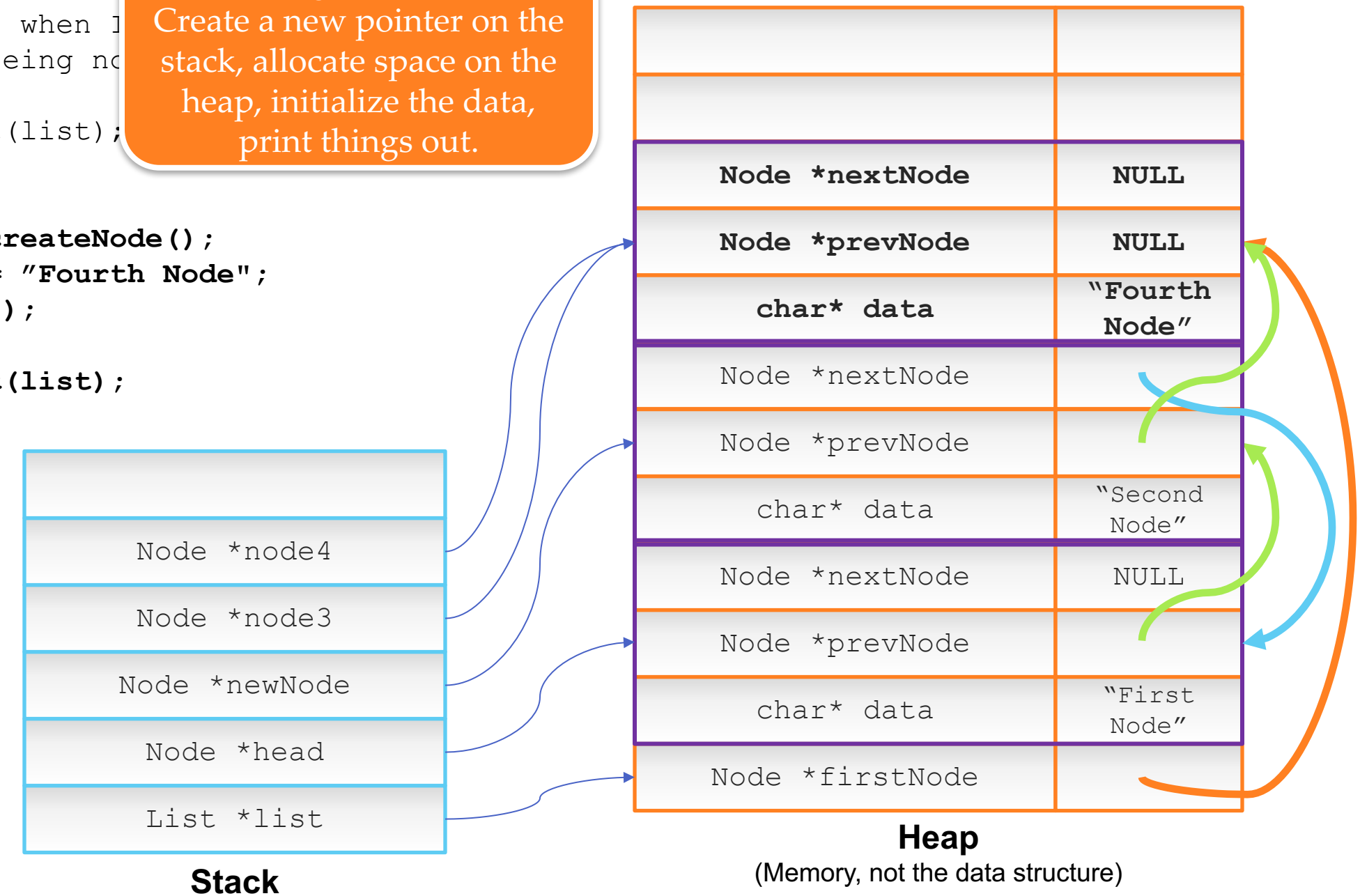
free() is only guaranteed to ***not save that chunk of memory*** any more. It doesn't

structure)

```
…
// Continuing on…
// What happens when I
printf("I'm freeing no
free(node3);
printLinkedList(list);

// But now…
Node *node4 = createNode();
node4 -> data = "Fourth Node";
printNode(node4);

printLinkedList(list);
```

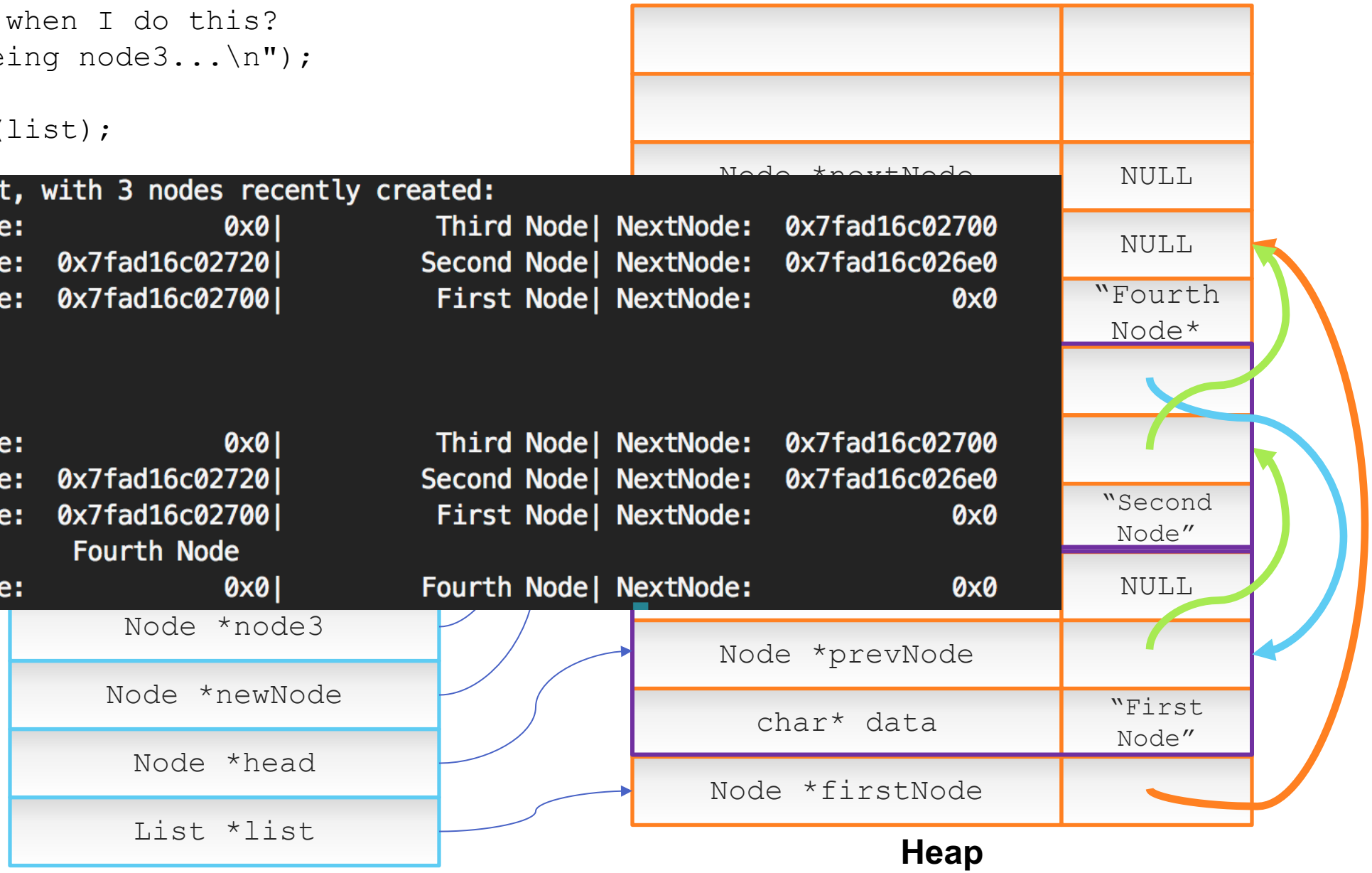
}

Creating a new node. Create a new pointer on the stack, allocate space on the heap, initialize the data, print things out.



```
    …
    // Continuing on…
    // What happens when I do this?
    printf("I'm freeing node3...\n");
    free(node3);
    printLinkedList(list);
}
```

```
printing the linked list, with 3 nodes recently created:
0x7fad16c02720  PrevNode:                0x0|          Third Node| NextNode:  0x7fad16c02700
0x7fad16c02700  PrevNode:  0x7fad16c02720|         Second Node| NextNode:  0x7fad16c026e0
0x7fad16c026e0  PrevNode:  0x7fad16c02700|          First Node| NextNode:              0x0

-------
I'm freeing node3...
0x7fad16c02720  PrevNode:                0x0|          Third Node| NextNode:  0x7fad16c02700
0x7fad16c02700  PrevNode:  0x7fad16c02720|         Second Node| NextNode:  0x7fad16c026e0
0x7fad16c026e0  PrevNode:  0x7fad16c02700|          First Node| NextNode:              0x0
Node: 0x7fad16c02720         Fourth Node
0x7fad16c02720  PrevNode:                0x0|         Fourth Node| NextNode:              0x0
```



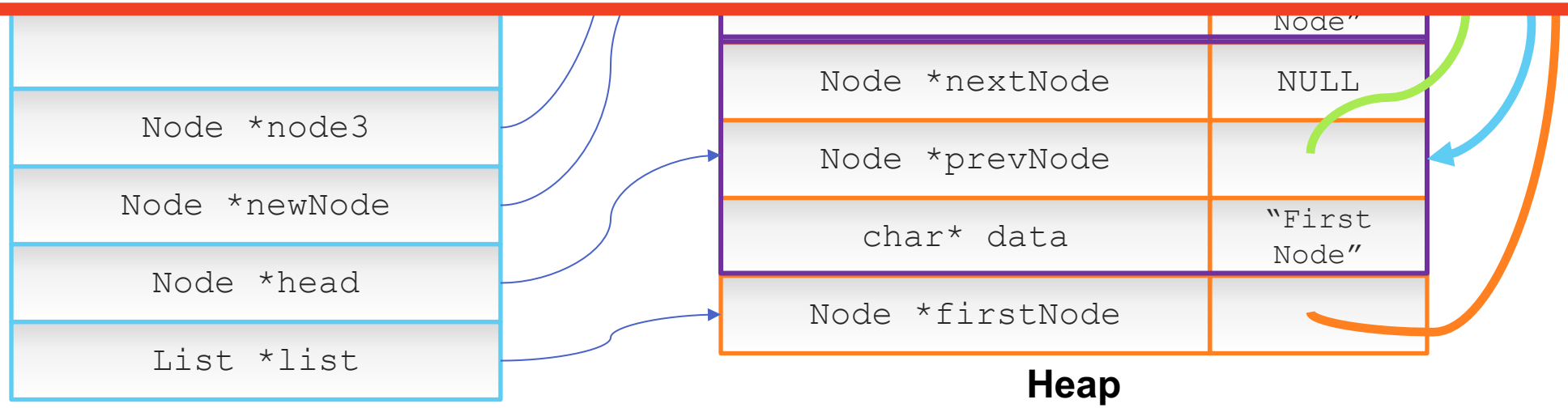
```
…
// Continuing on…
// What happens when I do this?
printf("I'm freeing node3...\n");
free(node3);
printLinkedList(list);

// But now…
```

Now, all of the above I ran on one machine.

I very particularly did this to demonstrate these concepts.

The fact is: you can't assume that all systems will behave like this.

I ran this same code on a different machine/different version of C, and it gave me a seg fault.



# To Summarize:

- **malloc** allocates some space on the heap
  - Until it's been freed, the heap is not going to put anything else in there.
- **free** de-allocates that space on the heap.
  - Nothing else is promised.
  - It does not guarantee that anything is removed from that memory.
  - It does not guarantee that you don't still have pointers to it.
- Unless you specifically clear out the memory, you can still access the data in that memory.
- But you can't ***guarantee*** access to that data.
- *Anyone smell potential threats to data???*

# DATA STRUCTURES INTRO

# What is a Data Structure?

- A structure used to store data.
  - Stores data called elements
  - Typical operations:
    - add()
    - remove()
    - clear()
    - size()
    - contains()
    - isEmpty()

# Examples of data structures

- Stack
- Lists (Array, Linked)
- Queue
- Maps
- Sets
- Trees

**Why so many?**

http://diaryofafitmommy.com/chocolate-chip-cookie-dough-protein-bites/#
https://www.amazon.com/Betty-Crocker-3-in-1-Baking-Rack/dp/B0018DYW84
http://i693.photobucket.com/albums/vv298/theliebertfamily/oct12/chewy-molasses-spice-cookies-400_zpsb89ed2fa.jpg

## Set

## Queue

## Stack

http://diaryofafitmommy.com/chocolate-chip-cookie-dough-protein-bites/#
https://www.amazon.com/Betty-Crocker-3-in-1-Baking-Rack/dp/B0018DYW84
http://i693.photobucket.com/albums/vv298/theliebertfamily/oct12/chewy-molasses-spice-cookies-400_zpsb89ed2fa.jpg

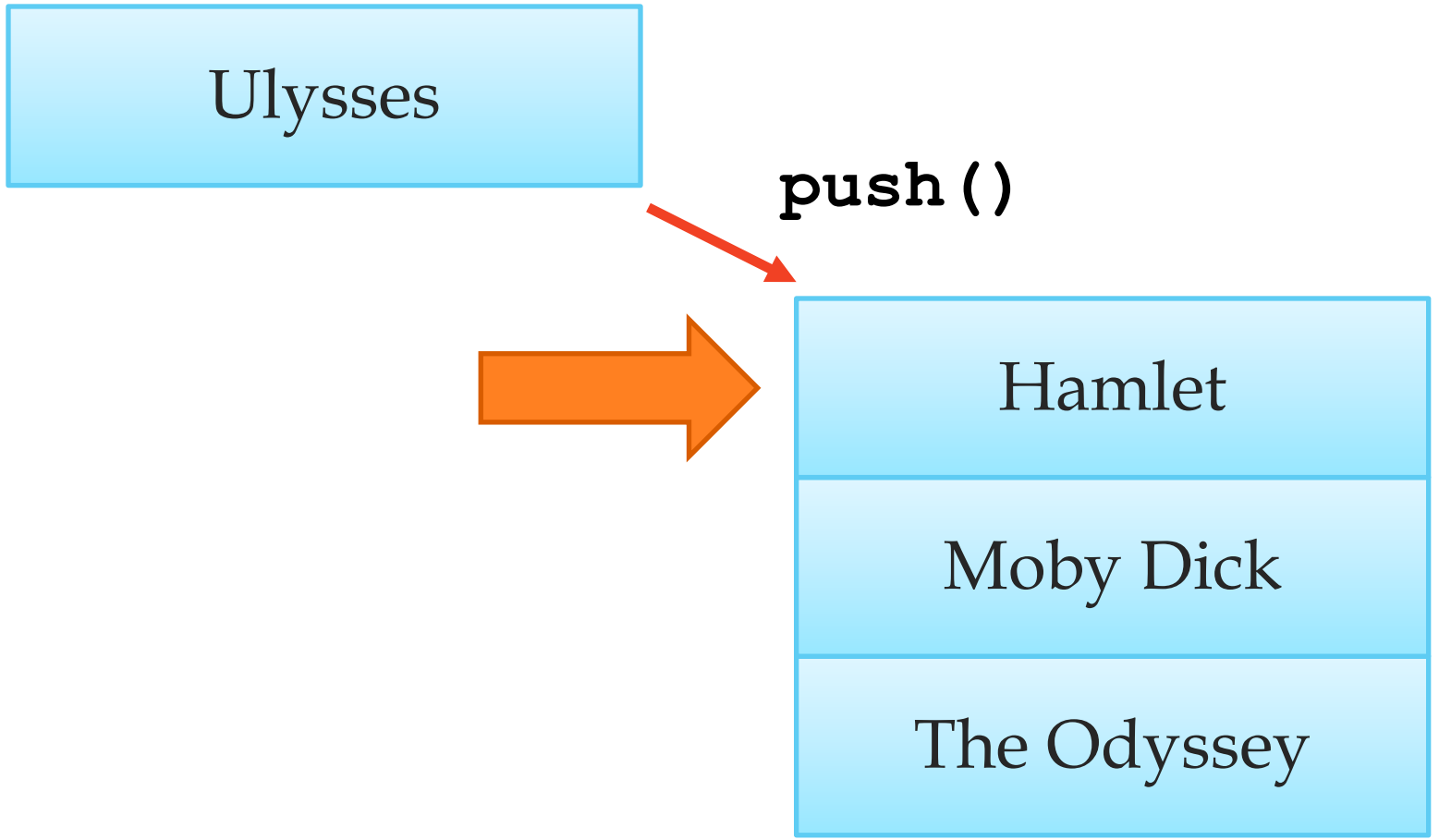
# Eventually, we will:

- Compare data structures based on:
  - **Characteristics**
    - LIFO? FIFO? Random Access? Can there be many copies of an element?
  - **Operations** we can perform on a data structure
  - **Performance and efficiency**:
    - How long does it take to put a piece of data in a data structure?
    - How long does it take to get data out of a data structure?
    - How much memory does the data structure use?

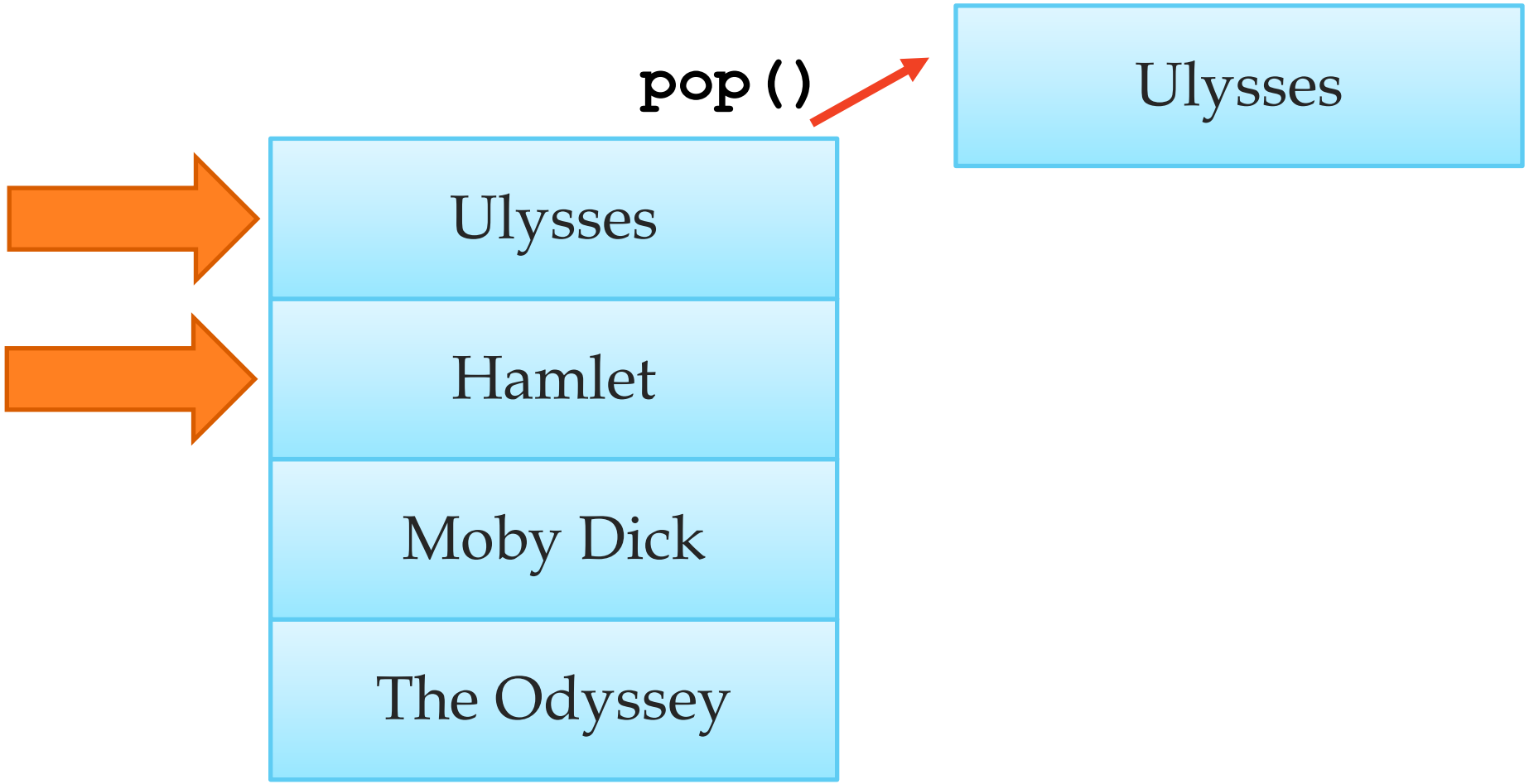
# STARTING WITH STACKS…

https://www.istockphoto.com/photo/stack-of-books-isolated-on-white-background-gm514536119-47608274

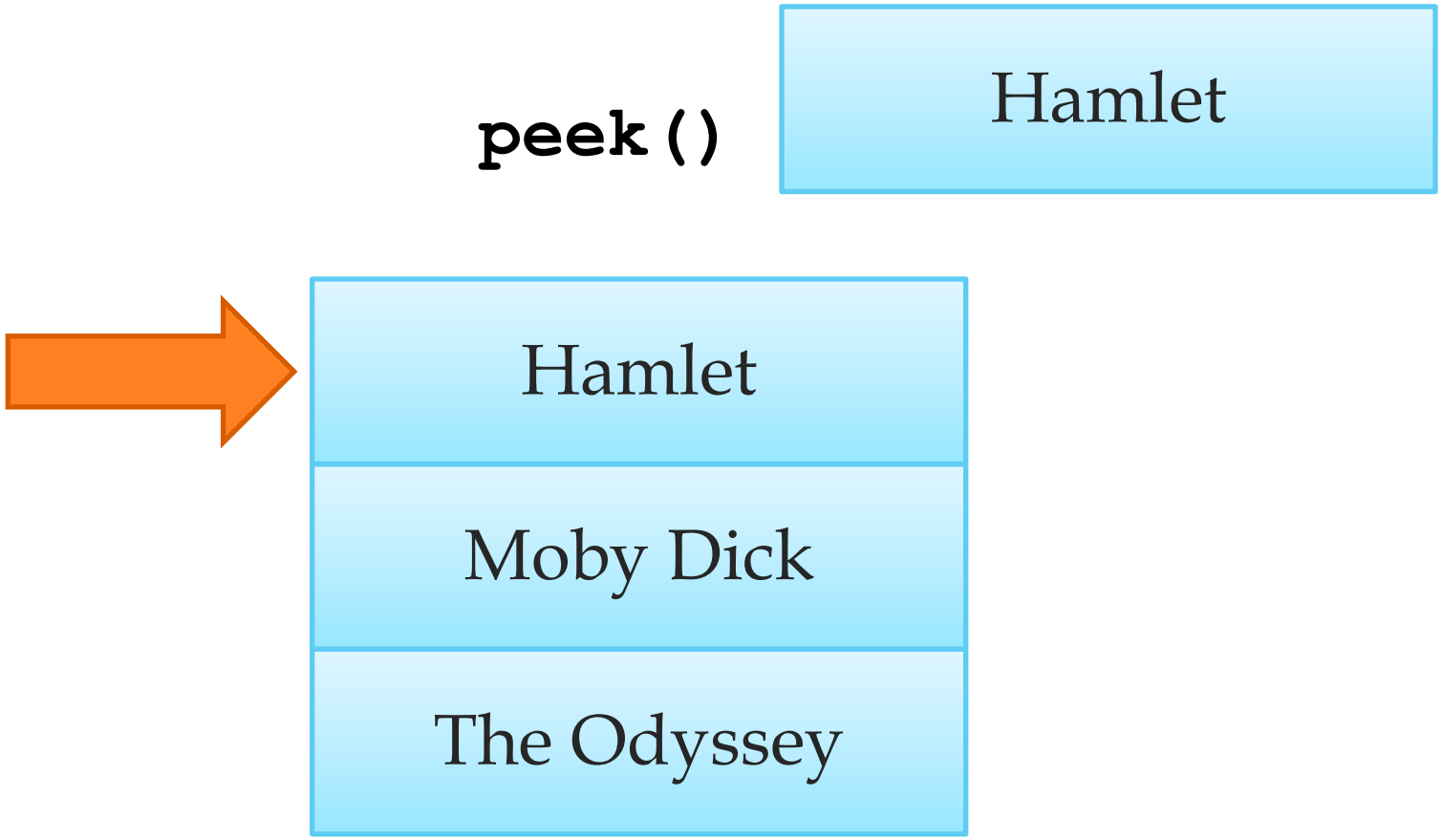
# What is a stack?



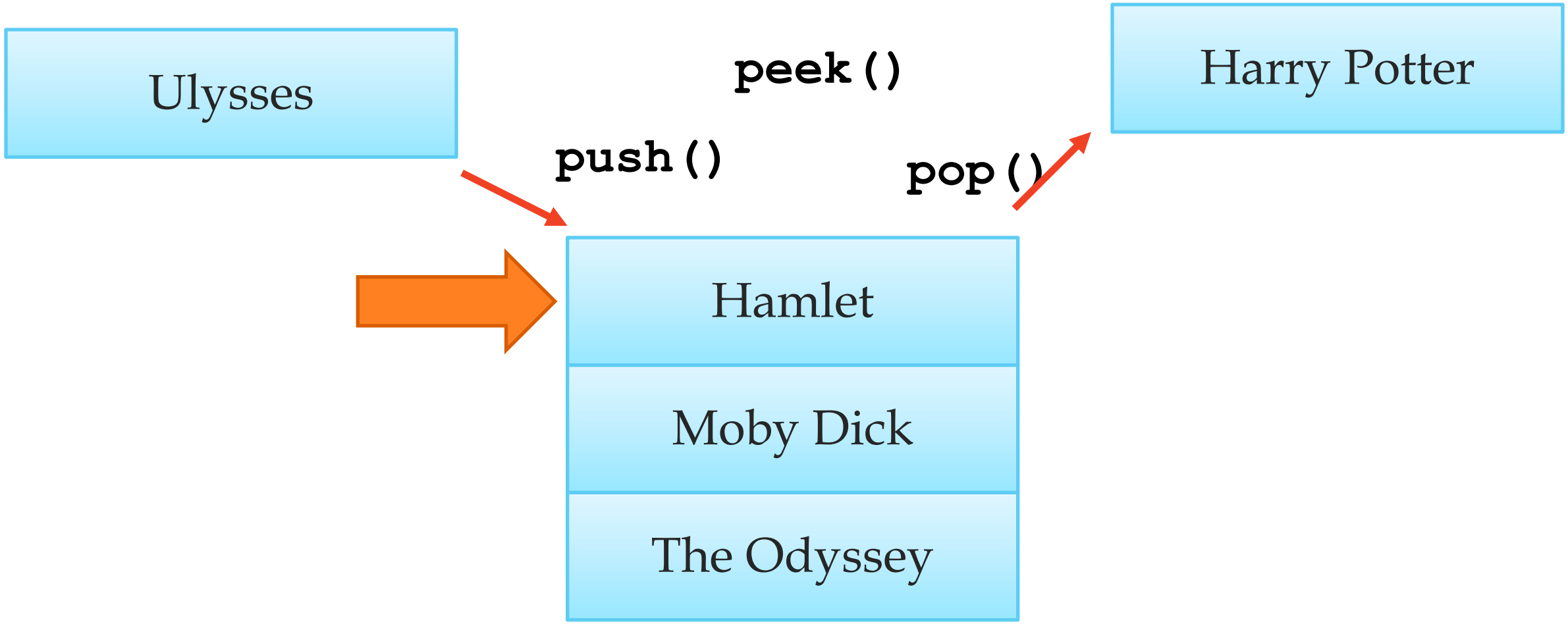
# What is a stack?



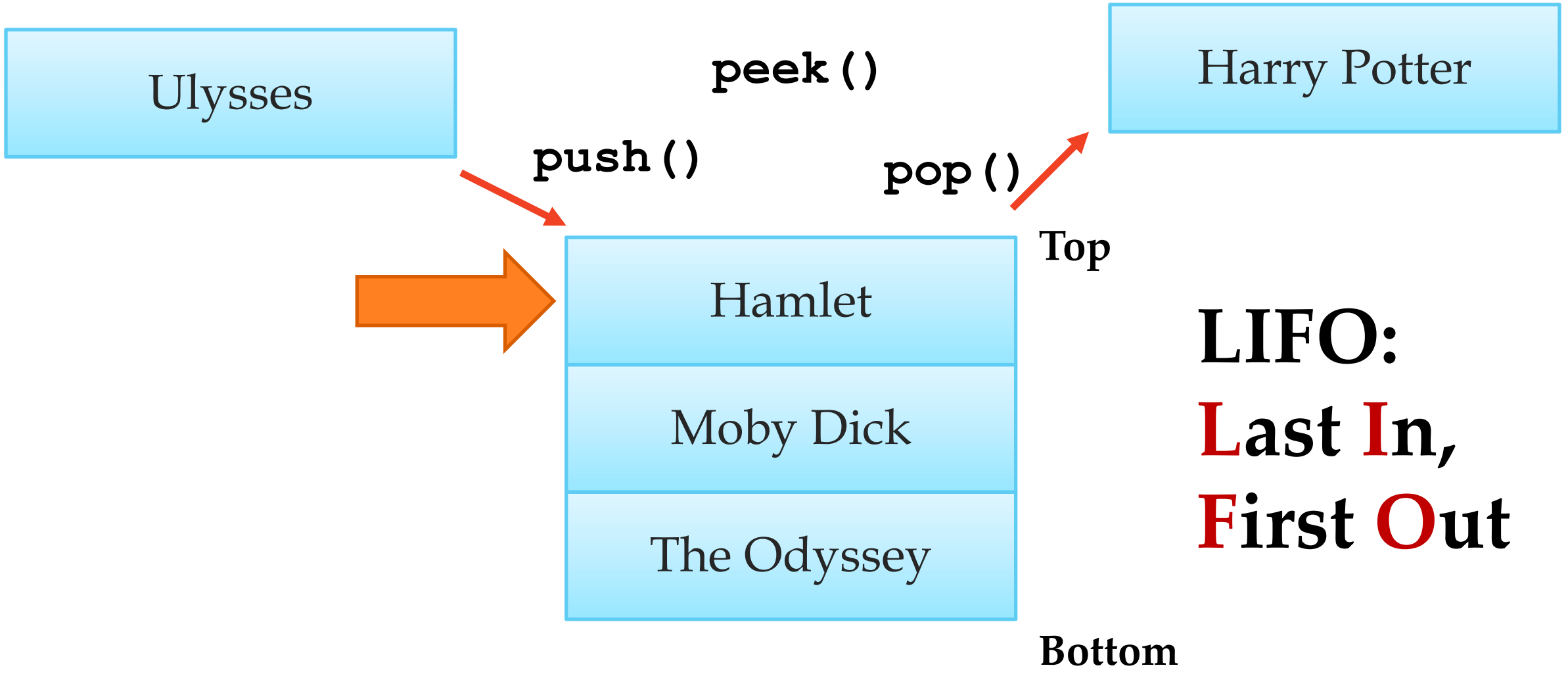
# What is a stack?



# What is a stack?



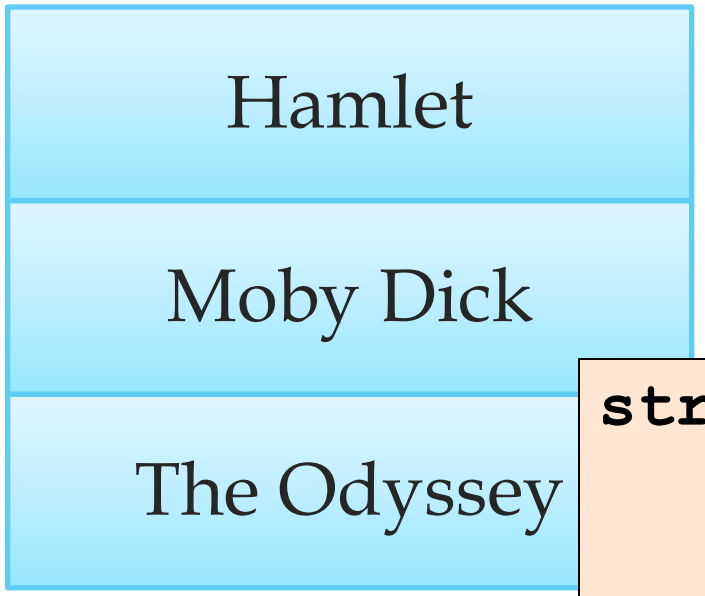
# What is a stack?




**LIFO:**
**Last In,**
**First Out**

# What is a stack in C?

```
int bookAtTop;
```




```
char* booknames[];
```

```
struct StackOfBooks{
    char* booknames[SIZE];
    int bookAtTop;
}
```

# Stack Operations

- pop()
- push()
- peek()
- create()
- destroy()

# Stack Operations in C

- `char* pop(stackOfBooks*);`
- `stackOfBooks* push(char* newBook, stackOfBooks*);`
- `char* peek(stackOfBooks*);`
- `stackOfBooks* create();`
- `void destroy(stackOfBooks*);`

```
struct stackOfBooks{
    char* booknames[SIZE];
    int bookAtTop;
}
```

# Implementing Stack Operations in C: create()

- `stackOfBooks* create();`

```
stackOfBooks* create(){
        // Allocate space on the heap
        stackOfBooks* newStack =
                (stackOfBooks*)malloc(sizeof(stackOfBooks));
        // Initialize anything
        newStack->bookAtTop = -1;
        // Return pointer to new stack
        return newStack;
}
```

```
struct stackOfBooks{
    char* booknames[SIZE];
    int bookAtTop;
}
```

# Implementing Stack Operations in C: destroy()

- `void destroy(stackOfBooks*);`

```
void destroy(stackOfBooks* bookStack){
      // Free anything that hasn't been freed

      // Free the stack
      free(bookStack);
}
```

```
struct stackOfBooks{
   char* booknames[SIZE];
   int bookAtTop;
}
```

# Implementing Stack Operations in C: push()

- `stackOfBooks* push(char*, stackOfBooks*);`

```
stackOfBooks* push(char* newBook, stackOfBooks* bookStack){
        // Update the head index
        bookStack->bookAtTop++;

        // Put the element in the array
        bookStack->booknames[WHAT GOES HERE] = newBook;

        // Return stack (by convention)
        return bookStack;
}
```

```
struct stackOfBooks{
    char* booknames[SIZE];
    int bookAtTop;
}
```

# Implementing Stack Operations in C: push()

- `stackOfBooks* push(char*, stackOfBooks*);`

```
stackOfBooks* push(char* newBook, stackOfBooks* bookStack){
        // Update the head index
        bookStack->bookAtTop++;

        // Put the element in the array
        bookStack->booknames[bookstack->bookAtTop] = newBook;

        // Return stack (by convention)
        return bookStack;
}
```

```
struct stackOfBooks{
    char* booknames[SIZE];
    int bookAtTop;
}
```

# Implementing Stack Operations in C: pop()

- `char* pop(stackOfBooks*);`

```
char* pop(stackOfBooks* bookStack){
      // Get the element from the array
      return bookStack->booknames[WHAT GOES HERE];

      // Update the head index???
}
```

```
struct stackOfBooks{
   char* booknames[SIZE];
   int bookAtTop;
}
```

# Implementing Stack Operations in C: pop()

- `char* pop(stackOfBooks*);`

```
char* pop(stackOfBooks* bookStack){
      // Get the element from the array
      return bookStack->booknames[bookStack->bookAtTop--];

      // Update the head index???
}
```

```
struct stackOfBooks{
   char* booknames[SIZE];
   int bookAtTop;
}
```

# Implementing Stack Operations in C: peek()

- `char* peek(stackOfBooks*);`

```
char* peek(stackOfBooks* bookStack){
      // Get the element from the array
      return bookStack->booknames[WHAT GOES HERE];

      // Update the head index???
}
```

```
struct stackOfBooks{
   char* booknames[SIZE];
   int bookAtTop;
}
```

# Implementing Stack Operations in C: peek()

- `char* peek(stackOfBooks*);`

```
char* peek(stackOfBooks* bookStack){
      // Get the element from the array
      return bookStack->booknames[bookStack->bookAtTop];

      // Update the head index???
}
```

```
struct stackOfBooks{
   char* booknames[SIZE];
   int bookAtTop;
}
```

# Let's play with code…

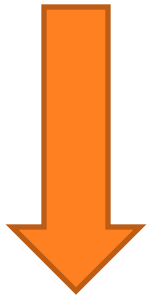
- https://github.ccs.neu.edu/CS5002SEAF17/resources/blob/master/lect9/stack_play.c

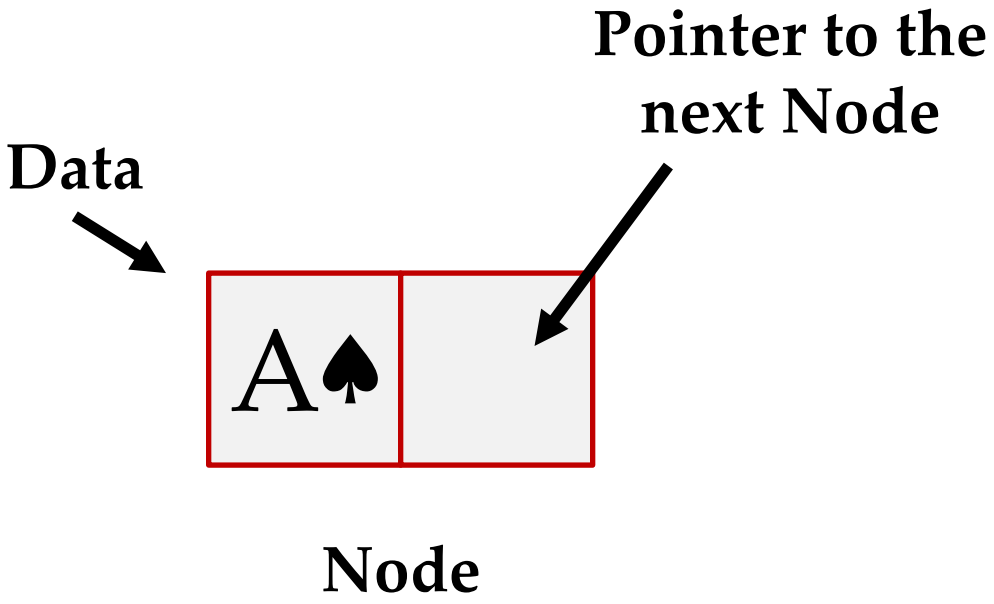
- (CS5002 resources repo on github; lect9)

# PLAYING WITH LINKED LISTS

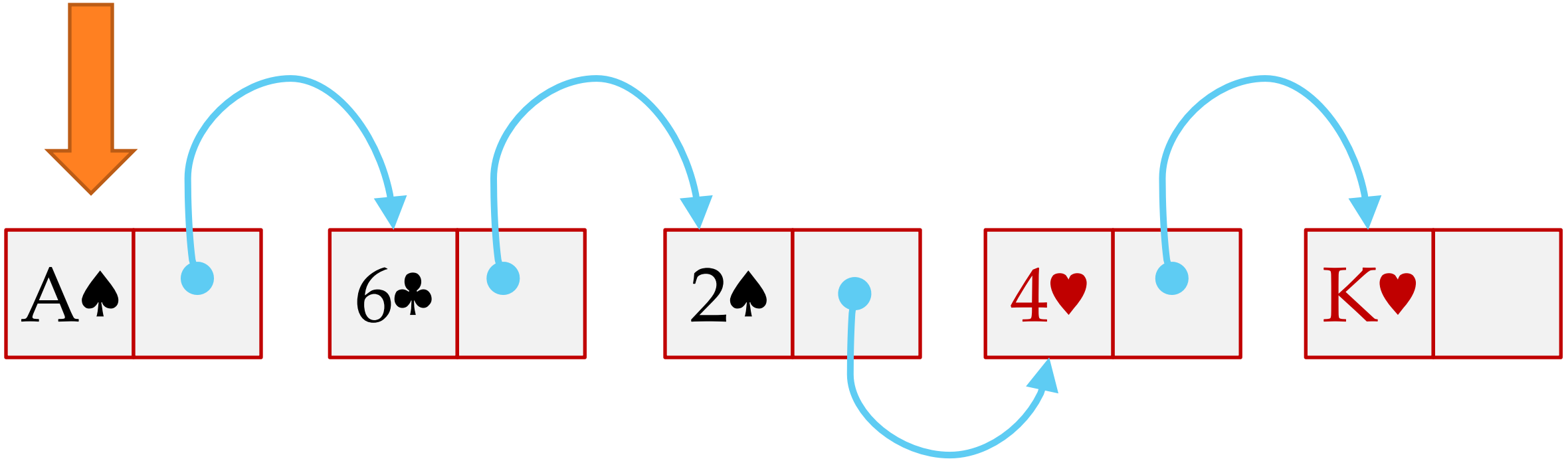
# Okay, back to the fun stuff.

# What is a Linked List?

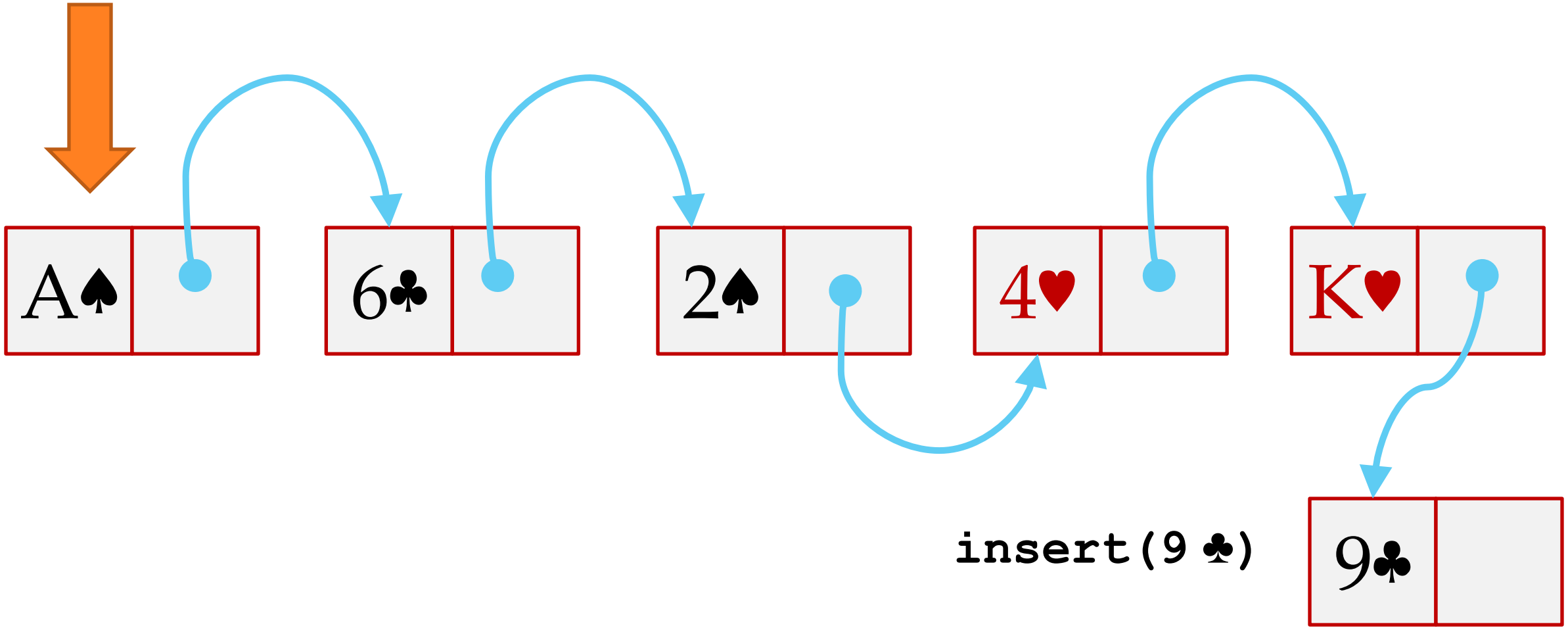




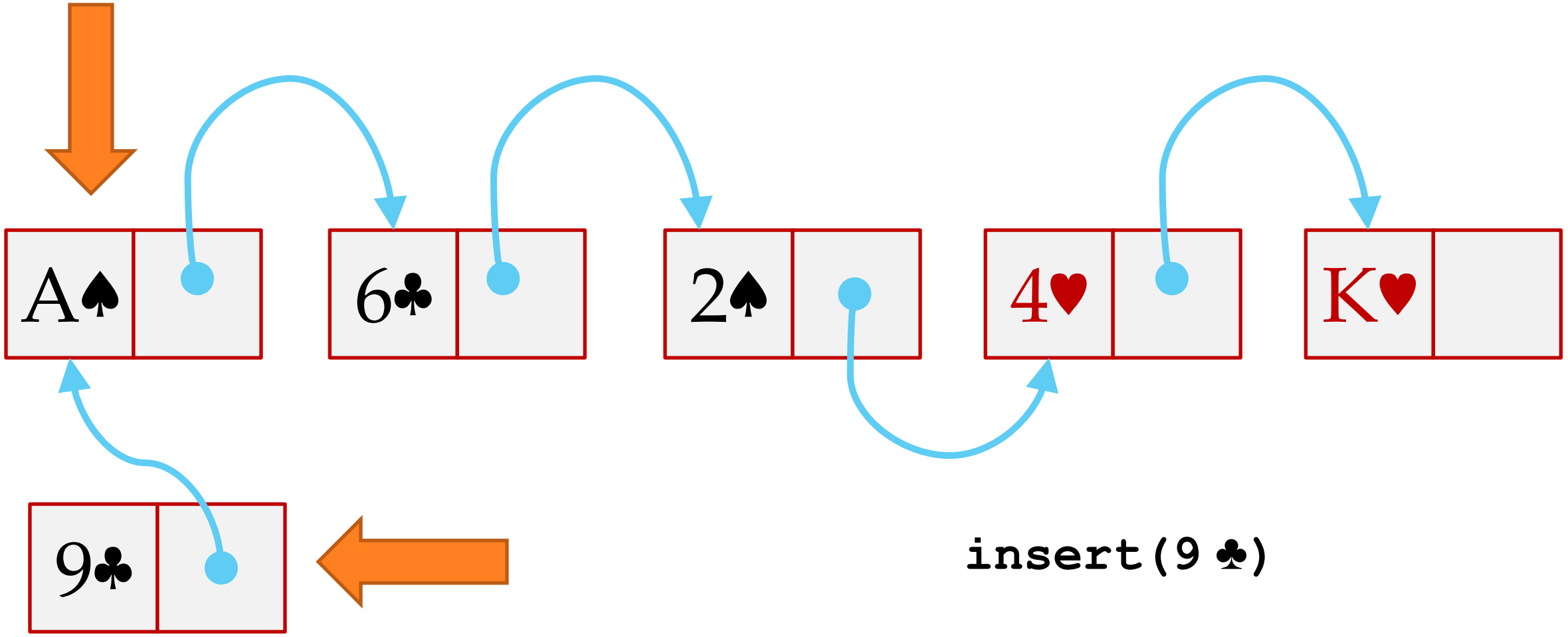
# What is a Linked List?



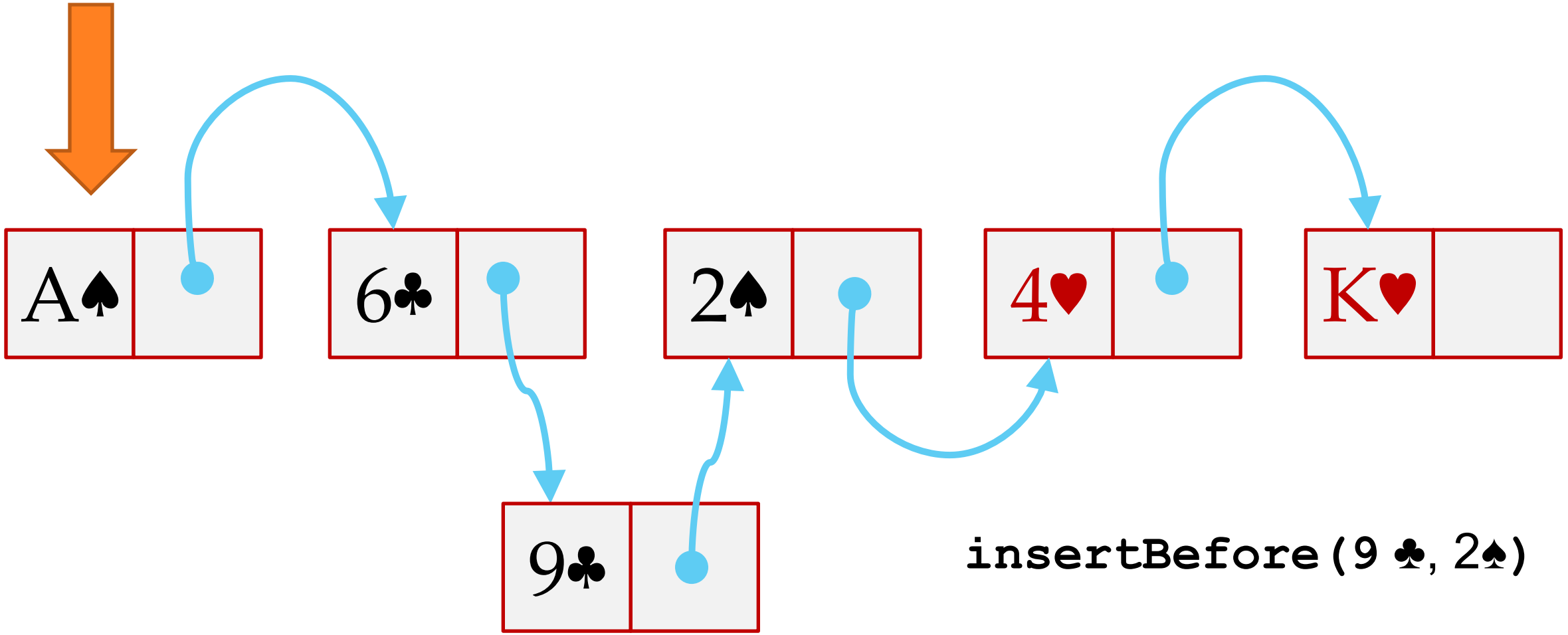
# Adding a Node to a Linked List



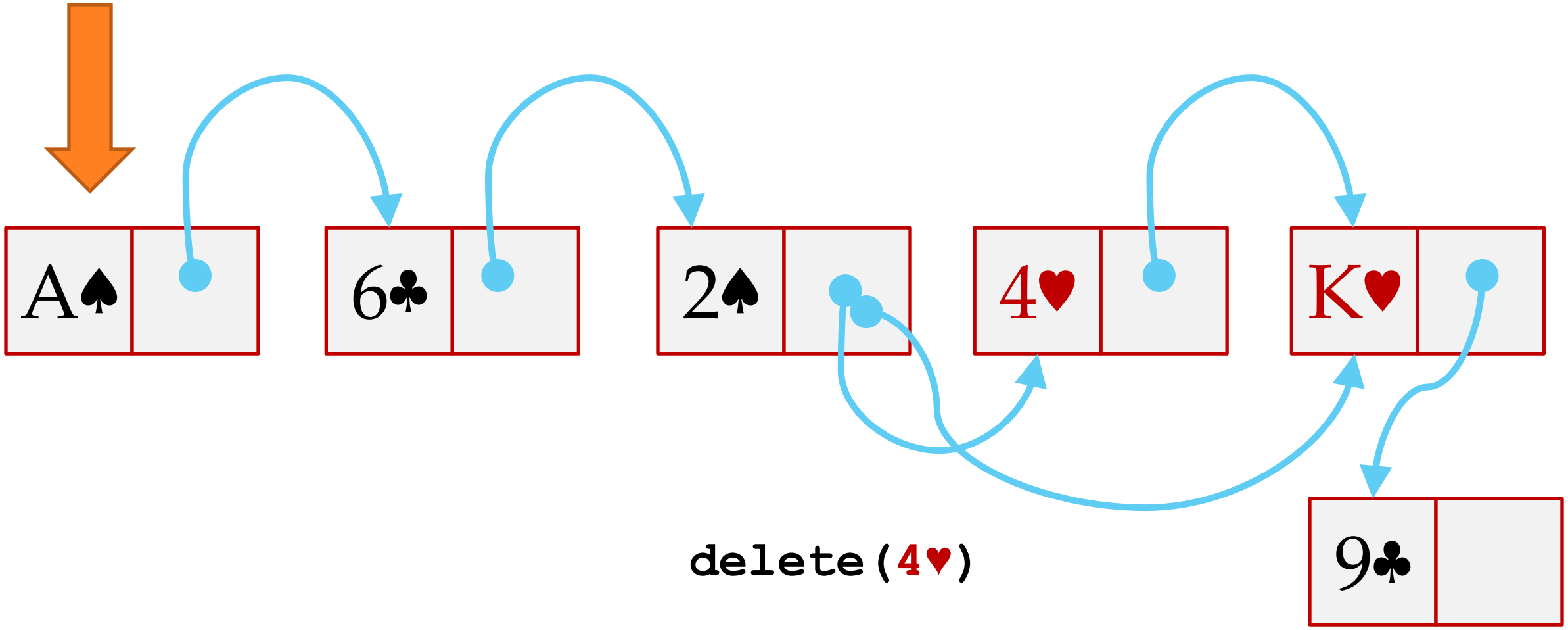
# An Alternative Way to Add a Node



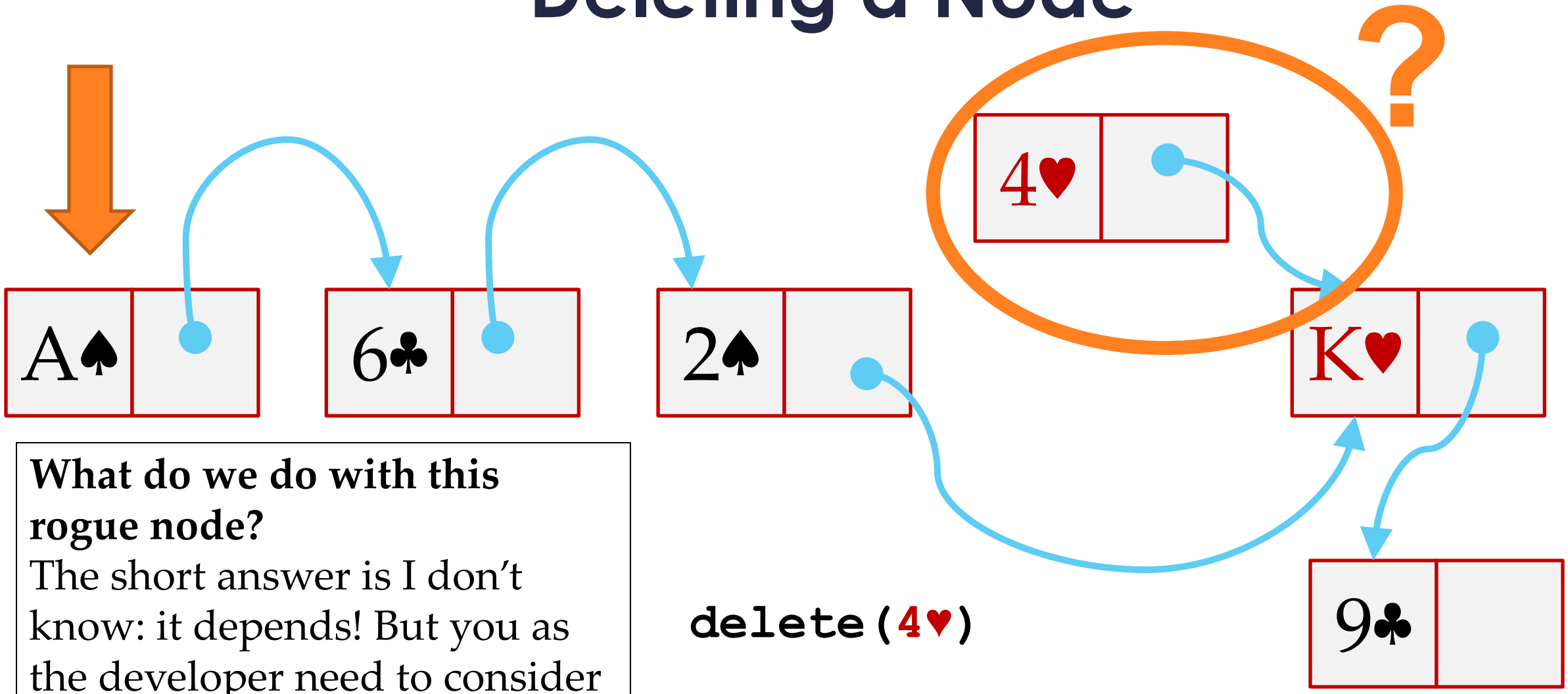
# Another Alternative Way to Add a Node



# Deleting a Node



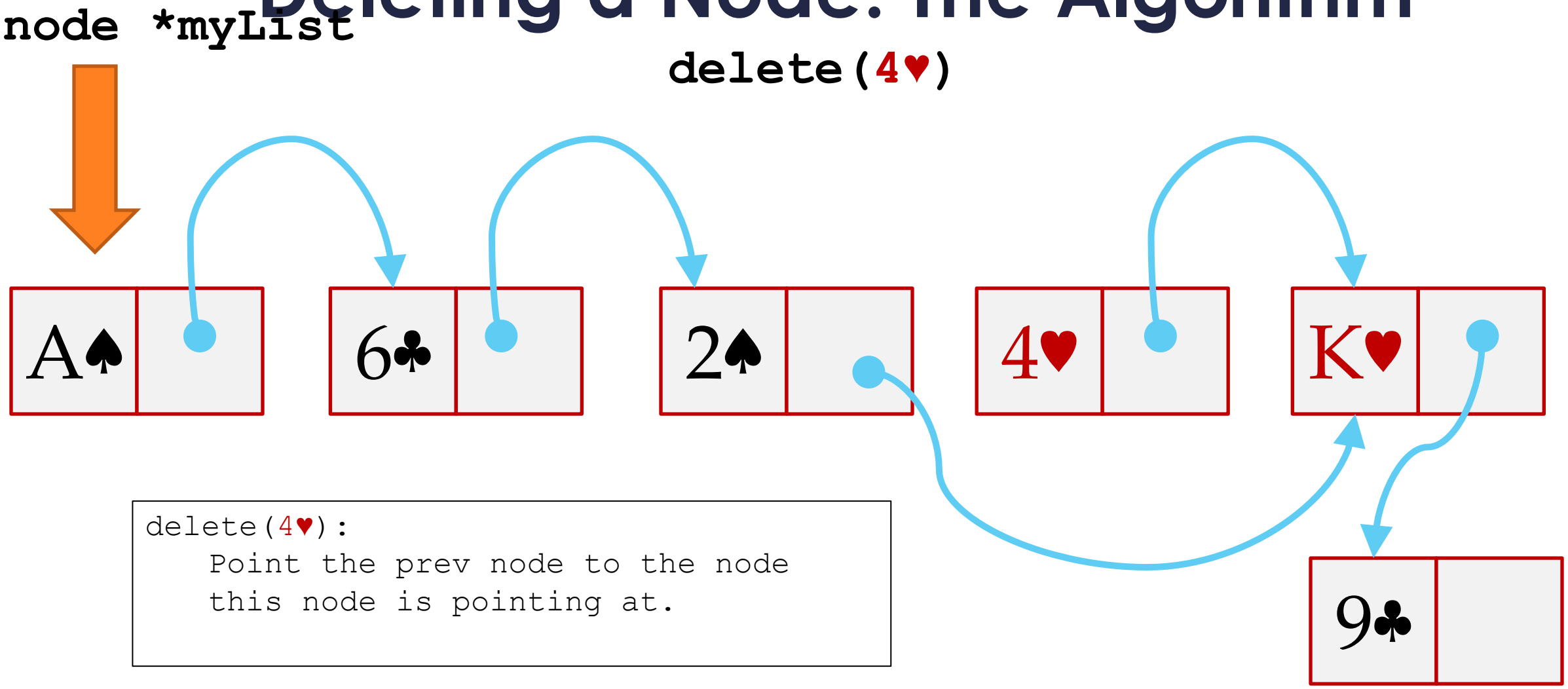
# Deleting a Node




**What do we do with this rogue node?**

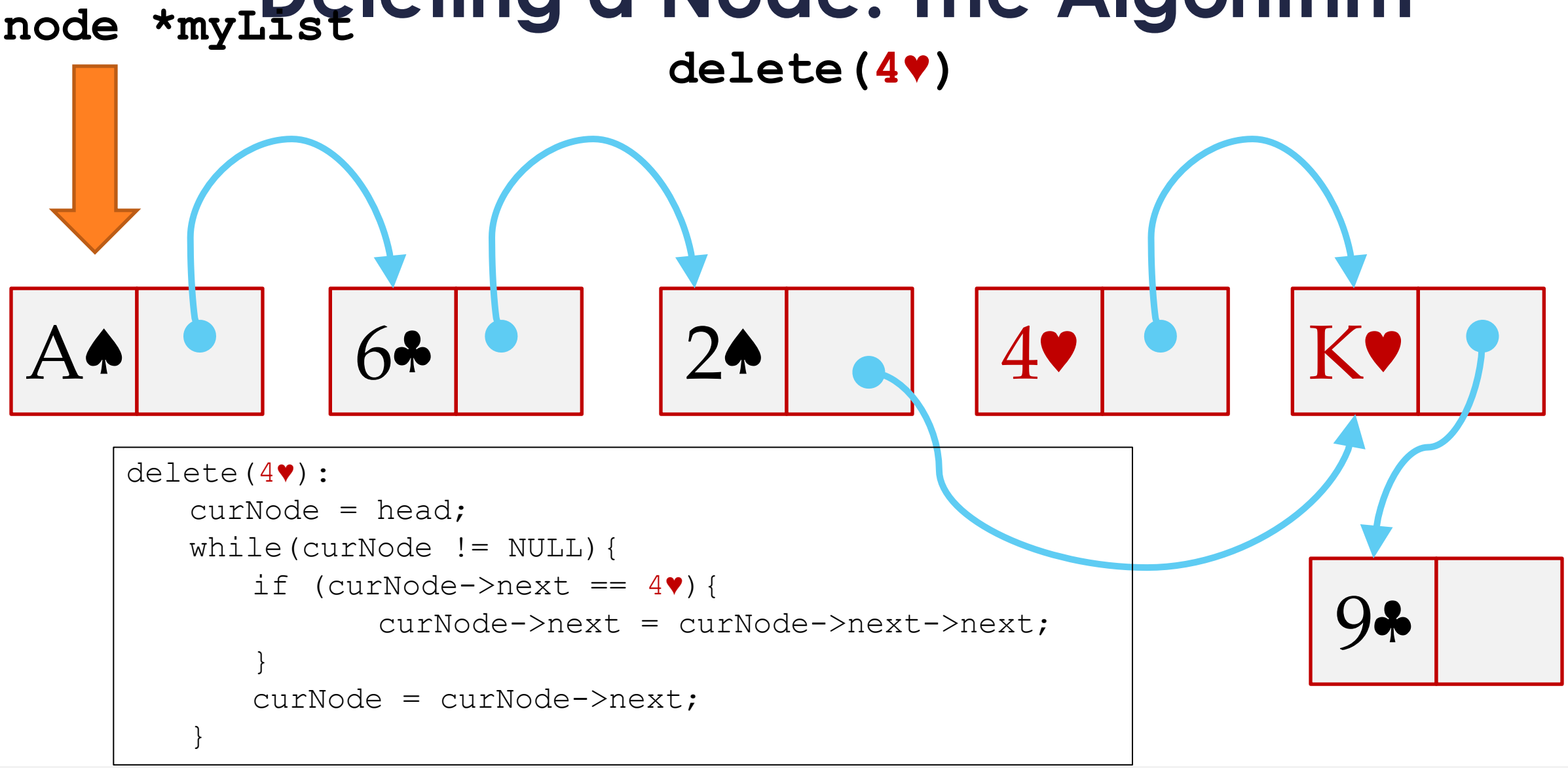
The short answer is I don't know: it depends! But you as the developer need to consider when it gets freed from the heap.

# Deleting a Node: The Algorithm




```
delete(4♥):
    Point the prev node to the node
    this node is pointing at.
```

# Deleting a Node: The Algorithm



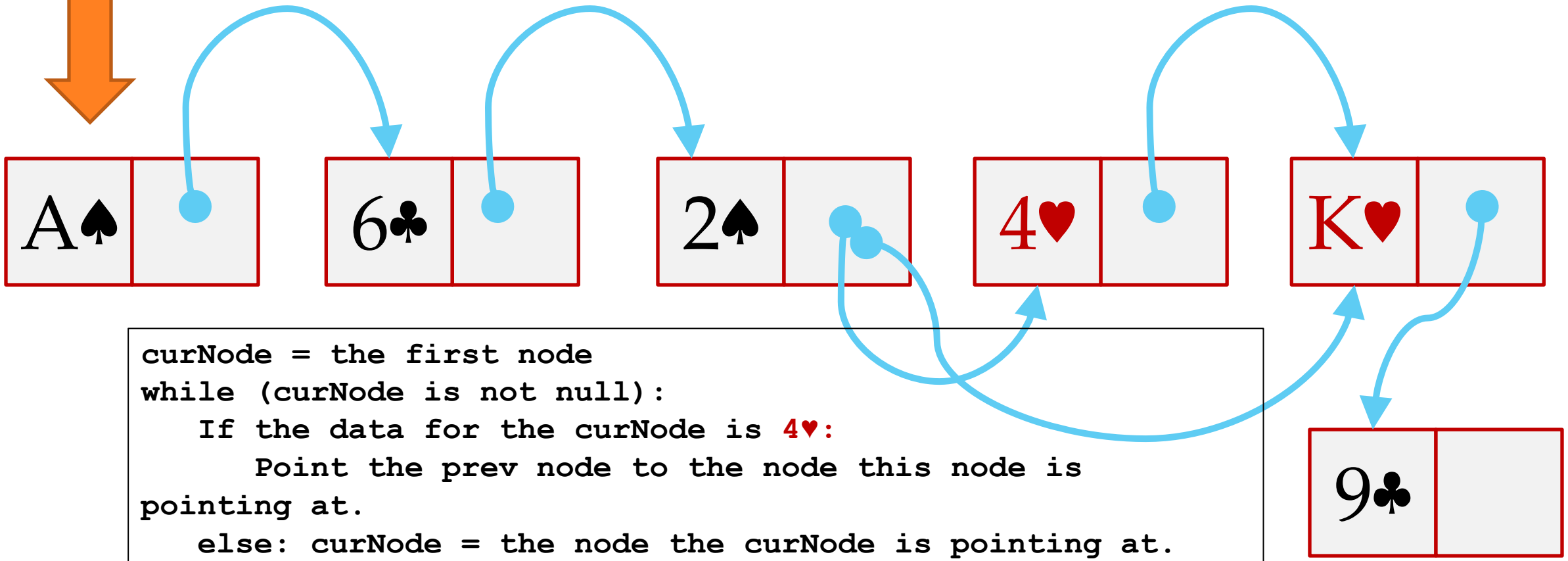

```
delete(4♥):
    curNode = head;
    while(curNode != NULL){
        if (curNode->next == 4♥){
                curNode->next = curNode->next->next;
        }
        curNode = curNode->next;
    }
```

What if I know which
***data*** I want to remove
(4♥),
but I don't know which
***NODE*** it is?

# Deleting a Node: The Algorithm

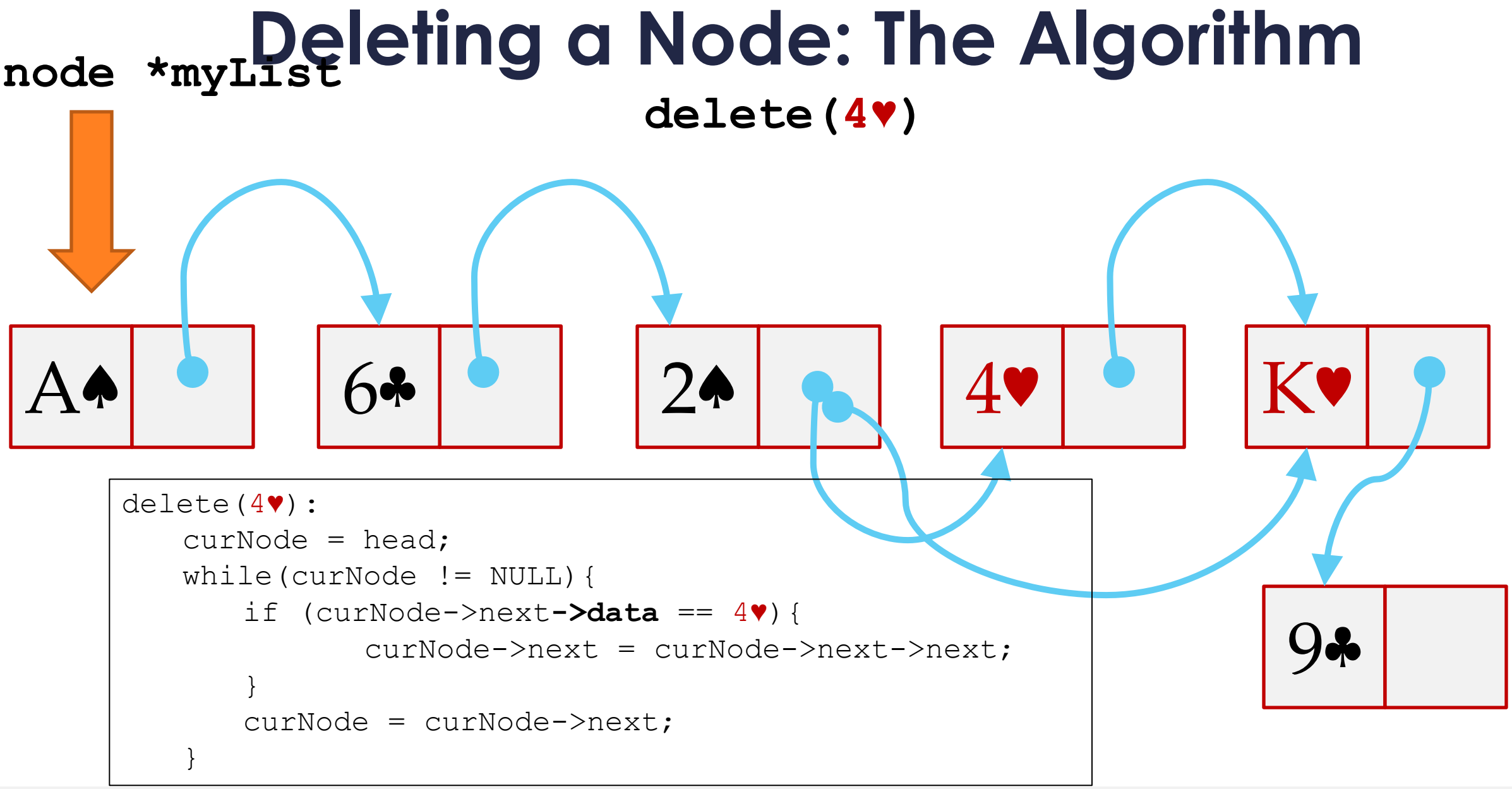
`node *myList`

`delete(4♥)`




```
curNode = the first node
while (curNode is not null):
   If the data for the curNode is 4♥:
      Point the prev node to the node this node is
pointing at.
   else: curNode = the node the curNode is pointing at.
```

# Deleting a Node: The Algorithm

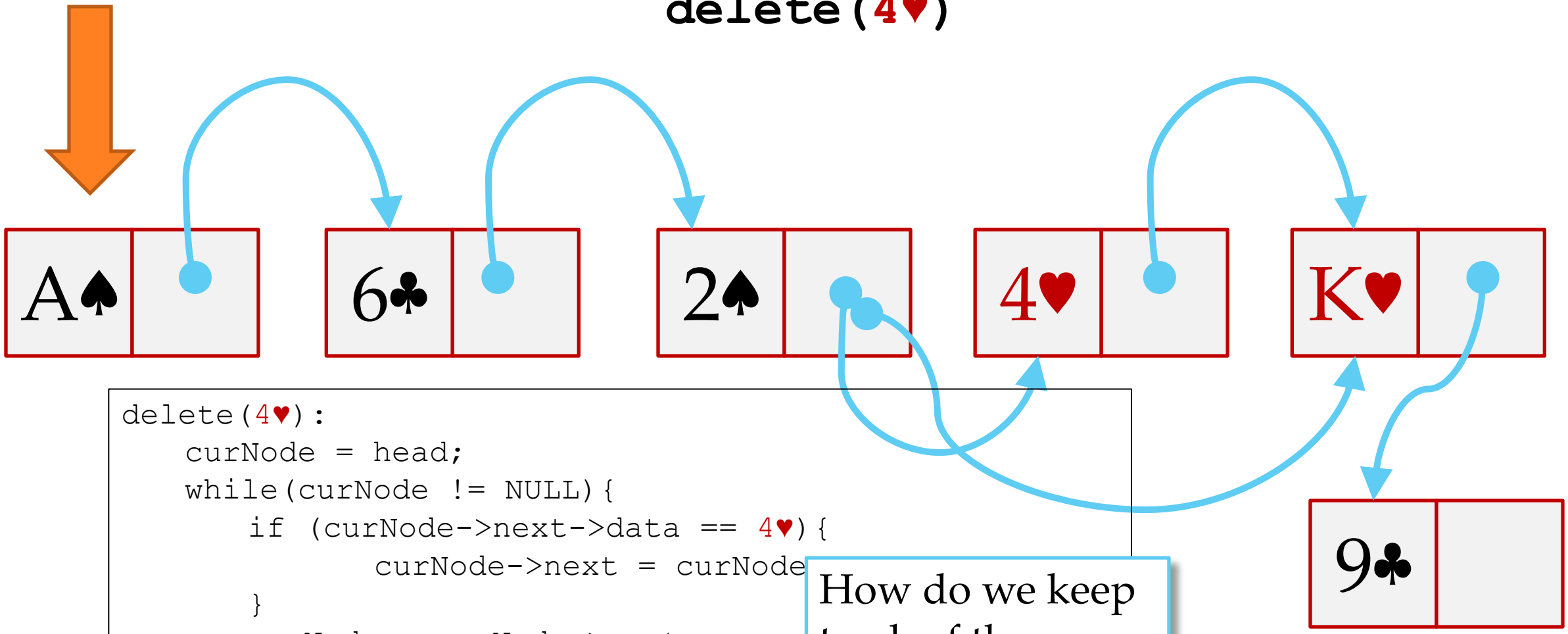



```
delete(4♥):
    curNode = head;
    while(curNode != NULL){
        if (curNode->next->data == 4♥){
                curNode->next = curNode->next->next;
        }
        curNode = curNode->next;
    }
```

# Deleting a Node: The Algorithm

`node *myList`

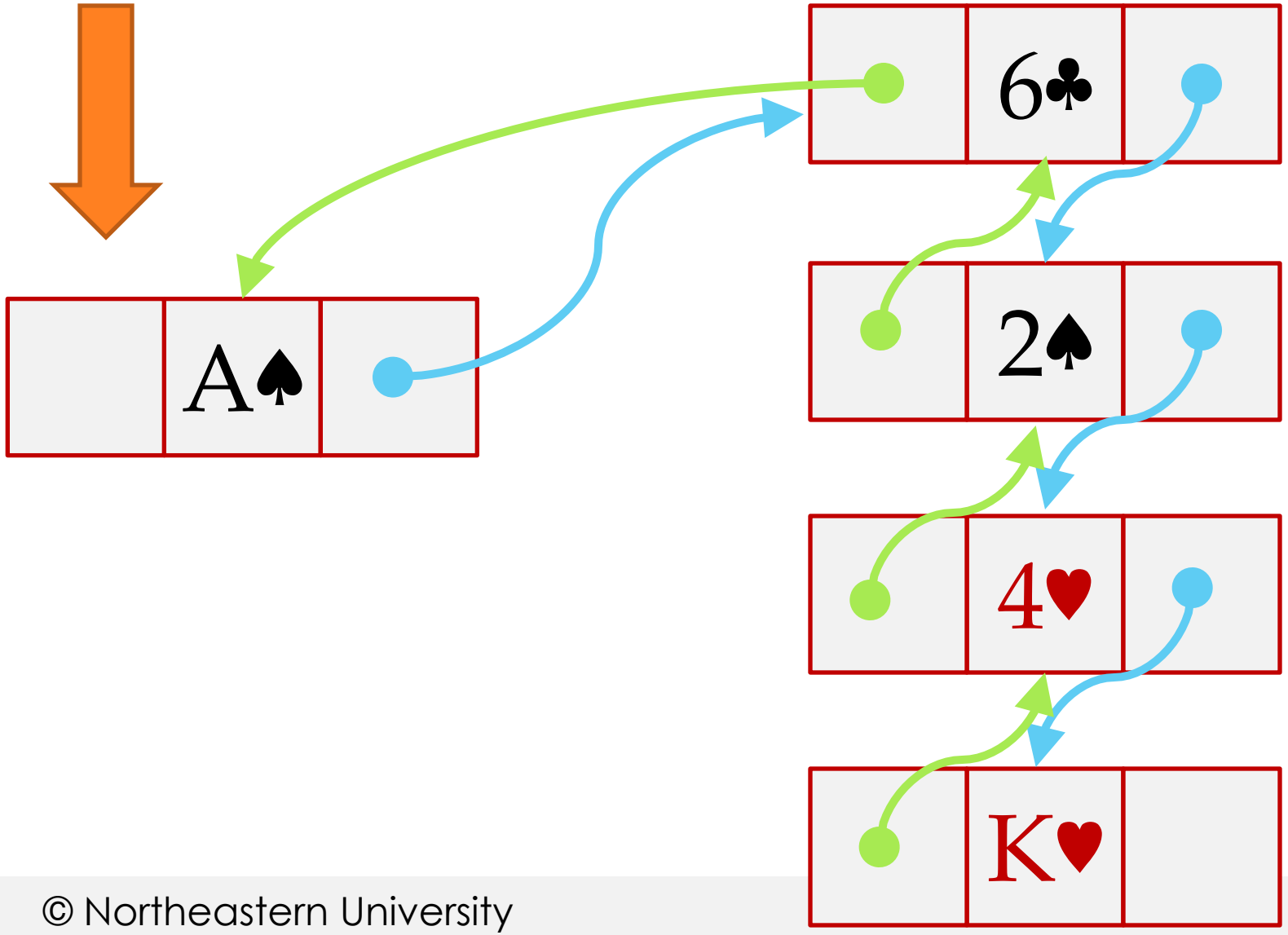
`delete(4♥)`




```
delete(4♥):
    curNode = head;
    while(curNode != NULL){
        if (curNode->next->data == 4♥){
                curNode->next = curNode
        }
        curNode = curNode->next;
    }
```

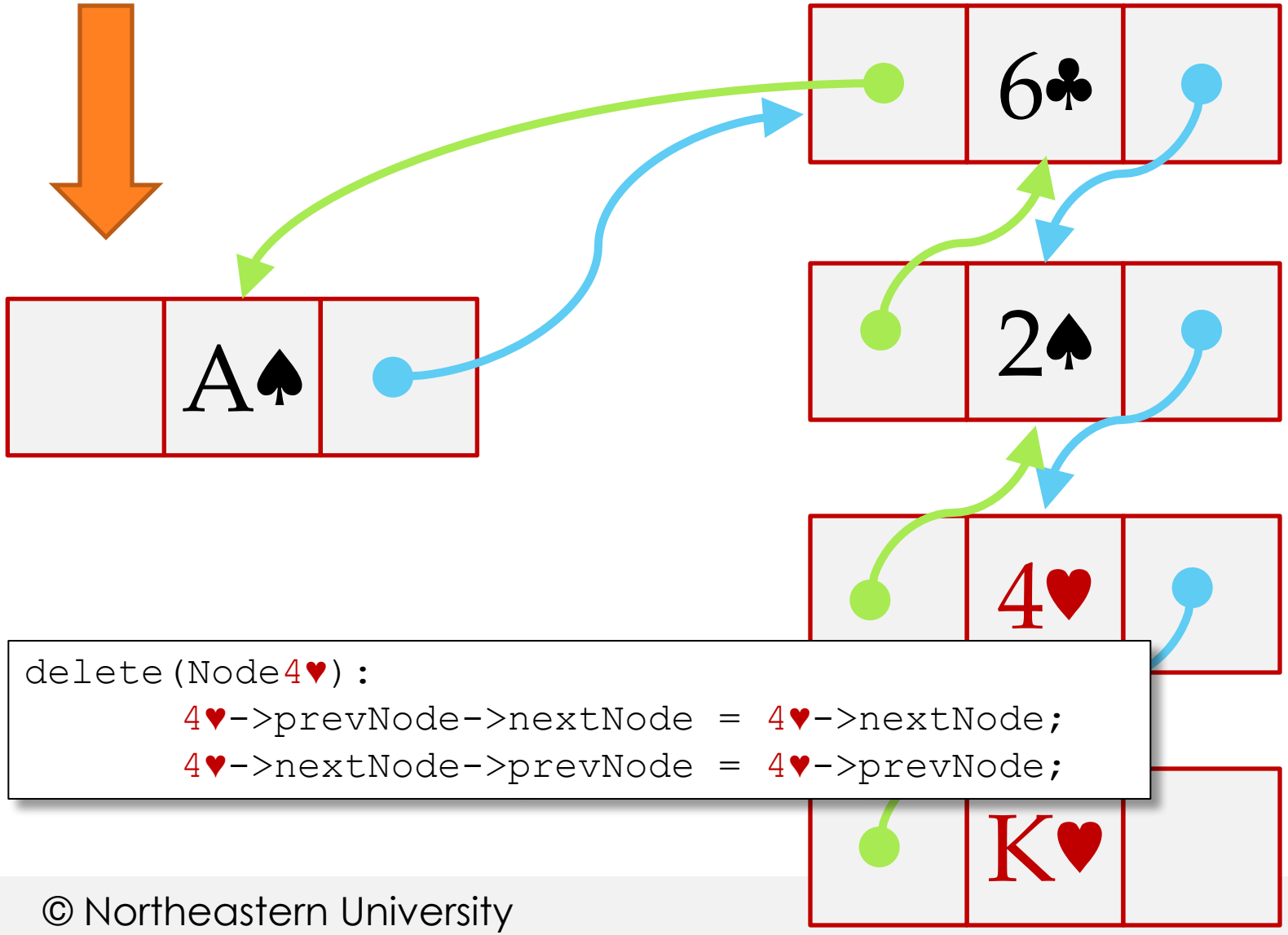
How do we keep track of the previous node??

# Solving some problems: a Doubly Linked List




```
struct Node{
    char* data;
    Node* prevNode;
    Node* nextNode;
};
```

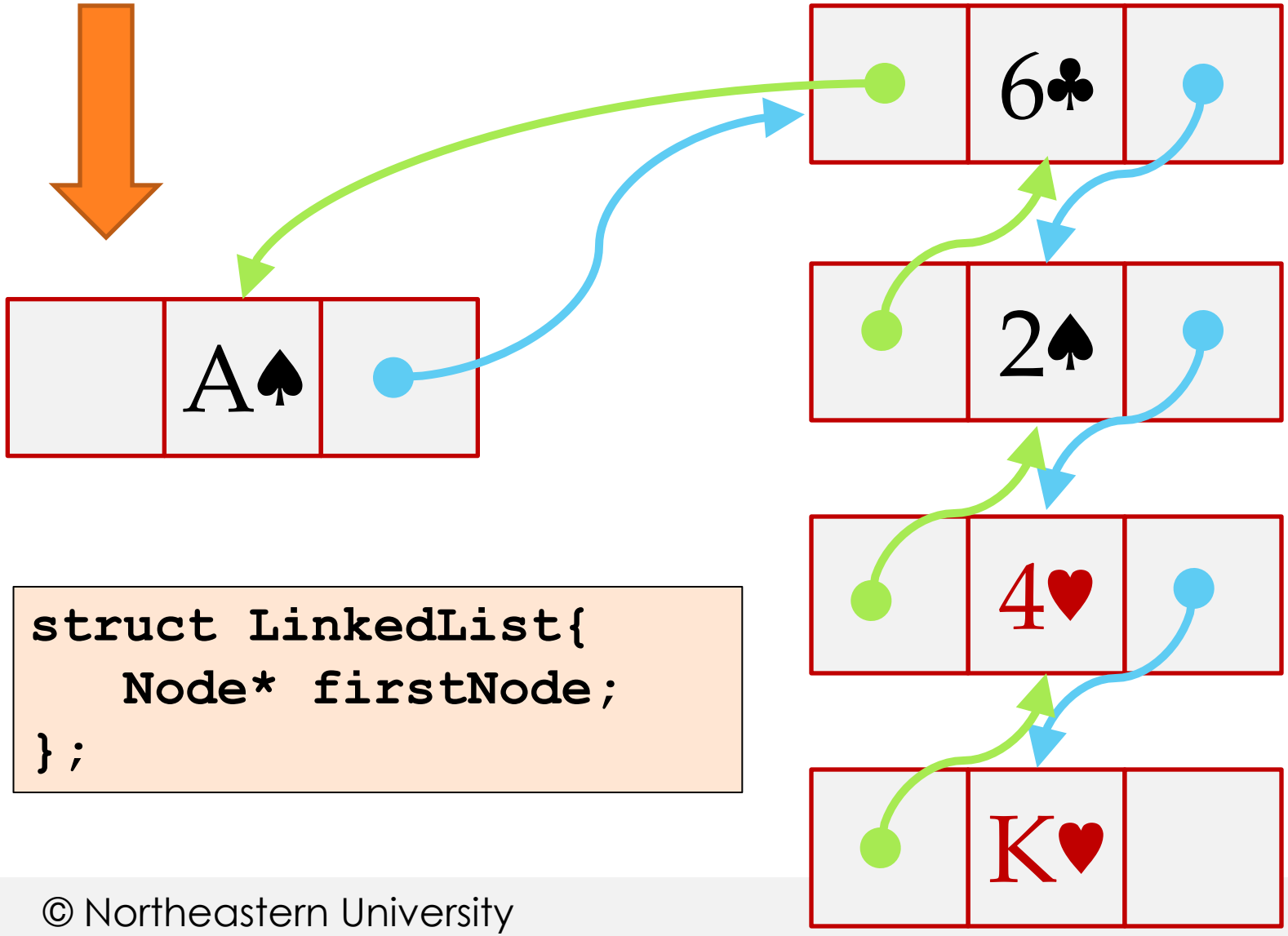
# Solving some problems: a Doubly Linked List




```
delete(Node4♥):
        4♥->prevNode->nextNode = 4♥->nextNode;
        4♥->nextNode->prevNode = 4♥->prevNode;
```

```
struct Node{
    char* data;
    Node* prevNode;
    Node* nextNode;
};
```

# Solving some problems: a Doubly Linked List




```
struct LinkedList{
    Node* firstNode;
};
```

```
struct Node{
    char* data;
    Node* prevNode;
    Node* nextNode;
};
```

**For the rest of this lecture, we're going to focus on:**
*** Doubly Linked List**
*** List is a special struct with the head of the list**

# Linked List Operations

- insert() or add()
- remove()
- create()
- destroy()
- contains()
- isEmpty()
- -----
- iterating: a helpful pattern

# Linked List Operations in C

- `void insert(list*, node*)`
- `void remove(list*, node*)`
- `list* create()`
- `void destroy(list*)`
- `int contains(list*, node*)`
- `int isEmpty(list*)`

# Linked List Operations in C: createList()

- `list* create()`

```
list* create(){
    list* list = (list*)malloc(sizeof(list));
    return list;
}
```

```
struct Node{
    char* data;
    Node* prevNode;
    Node* nextNode;
};
```

# Linked List Operations in C: createNode()

- `node* createNode(char* data)`

```
node* createNode(char* data){
    node* node = (node*)malloc(sizeof(node));
    node->data =  data;
    node->prevNode = NULL;
    node->nextNode = NULL;
    return node;
}
```

```
struct Node{
    char* data;
    Node* prevNode;
    Node* nextNode;
};
```

# Linked List Operations in C: destroy()

- `void destroy(list*)`

```
void destroy(list* aList){
    // What to do with this Node?
    // What if the list is not empty???
    free(aList->firstNode); // Should I do this?
    free(aList);
}
```

```
struct Node{
    char* data;
    Node* prevNode;
    Node* nextNode;
};
```

# Linked List Operations in C: insert ()

```
struct Node{
    char* data;
    Node* prevNode;
    Node* nextNode;
};
```

- `void insert(list*, node*)`

```
// inserting at the front of the list
void insert(list* aList, node *newNode){
        if(aList->firstNode != NULL){
                // Point the new node to the first node
                newNode->nextNode = aList->firstNode;
                // Point it back
                aList->firstNode->prevNode = newNode;
        }
        // Return the now-first element of the list
        aList->firstNode = newNode;
}
```

# Linked List Operations in C: insert ()

- `node* insertAtEnd(list*, node*)`

```
// inserting at the end of the list
node* insertAtEnd(node* aList, node *newNode){
      ** WRITE THIS CODE **
}
```

```
struct Node{
    char* data;
    Node* prevNode;
    Node* nextNode;
};
```

# Linked List Operations in C: remove ()

- `void remove(list*, node*)`

```
void remove(node* aList, node* aNode){
    if(aNode->prevNode != NULL){
        aNode->prevNode->nextNode = aNode->nextNode;
    }
    if(aNode->nextNode != NULL){
        aNode->nextNode->prevNode = aNode->prevNode;
    }
}
```

**What do we do with this rogue node?**
The short answer is I don't know: it depends! But you as the developer need to consider when it gets freed from the heap.

```
struct Node{
    char* data;
    Node* prevNode;
    Node* nextNode;
};
```

# Linked List Operations in C: contains()

- `int contains(list*, node*)`

```
int contains(list* aList, node* aNode){
    ** WRITE THIS CODE **
}
```

```
struct Node{
    char* data;
    Node* prevNode;
    Node* nextNode;
};
```

# Linked List Operations in C: containsValue()

- `int containsValue(list*, data);`

```
int containsValue(list* aList, char* data){
    ** WRITE THIS CODE **
}
```

**What to return?**
0/1 for True/False?
Or the Node containing the data?

```
struct Node{
    char* data;
    Node* prevNode;
    Node* nextNode;
};
```

# Let's Play with Code...

- https://github.ccs.neu.edu/CS5002SEAF17/resources/tree/master/lect9

- Github->CS5002SEAF17-> resources->lect9-> linked_list.c

# Arrays Versus LinkedList

**Array**

- Contiguous Memory
- You have to know the size before you create it
  - If you don't use all the memory allocated, you're wasting it
  - If you don't allocate enough, oops.
- **Time to put an element in the array:**
  -
- **Time to get an element from the array:**
  -

**LinkedList**

- Nodes can be non-contiguous
- You can keep growing the list, taking as much memory as you need but not more than you need
- **Time to put a node into the list:**
  -
- **Time to get a node/data from the list**
  -

**hmmmm, how do we calculate this?**

# Summary

**Stack**

- Primary Operations
  - Push
  - Pop
  - Peek
- Helper Operations
  - Create
  - Destroy
  - IsEmpty
  - IsFull

**Linked List**

- Primary Operations
  - Add
  - Remove
- Helper Operations
  - Create
  - Destroy
  - IsEmpty
  - Contains(Node), Contains(Data)
  - InsertBefore(), InsertAfter()
  - AddAtHead(), AddAtTail()

# Summary

**Stack**

- The abstraction is more important than the implementation
  - We implemented as an array; could also use a list.

**Linked List: Alternative to Array**

- Could be single or double
  - We talked about single
  - We implemented double
  - Code for single is in Github
- The List is really just a pointer to a single node
  - We implemented a wrapper struct that helped us keep track of the head
  - We can add metadata to that struct to help
    - Such as NumItems, ptr to LastNode.

# Next time:

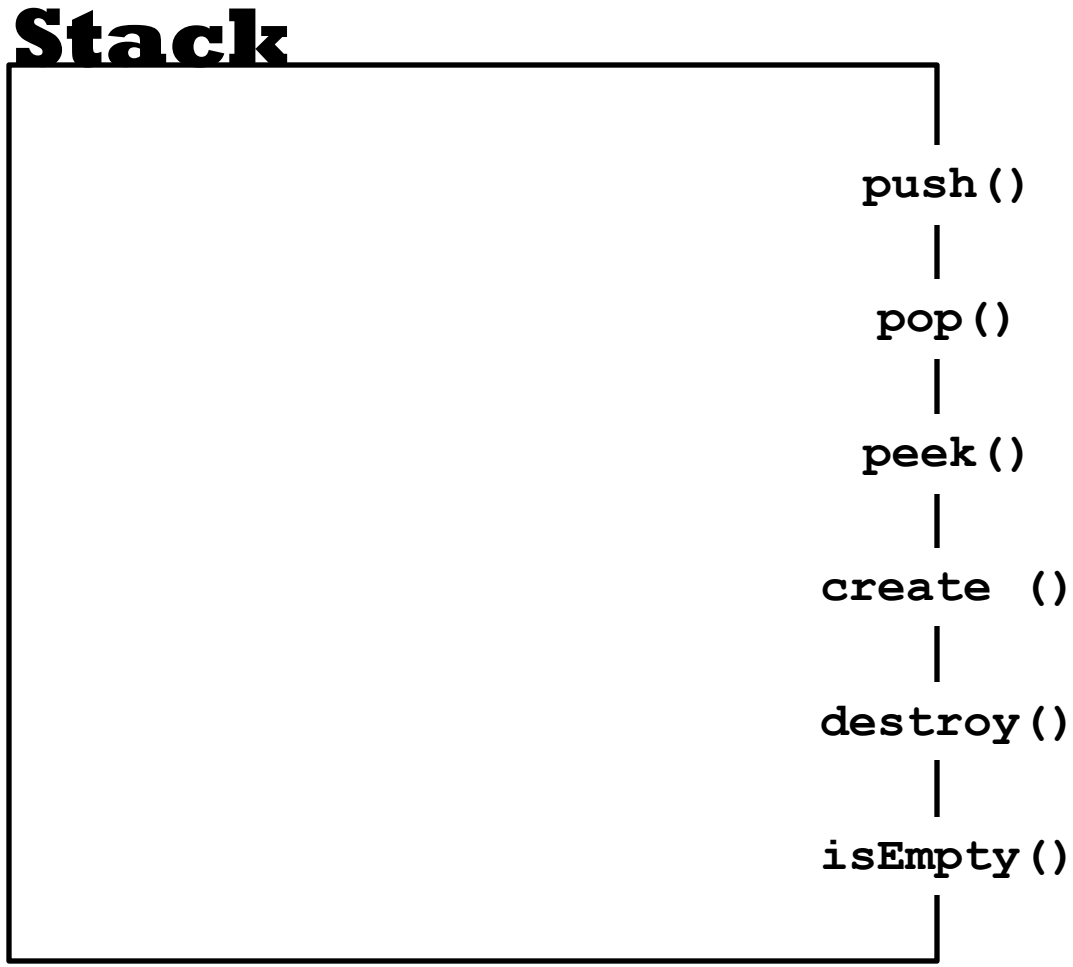
- Finish up with Linked Lists
- Queues
- Big O complexity and performance
  - How to talk about our algorithms.

# ABSTRACT DATA TYPES

# Abstract Data Types

- It's about the interface
- The concrete implementation can vary

# A stack is a black box, with a well-defined interface:



# The implementation (code) is independent of the operations.

## Stack

```
stackOfBooks* create(){
    // Allocate space on the heap
    stackOfBooks* newStack =
        (stackOfBooks*)malloc(sizeof(stackOfBooks));
    // Initialize anything
    newStack->bookAtTop = -1;

    // For purposes of demo, I'm doing this here. It's not always necessary:
    int i=0;
    for (i=0; i<MAXSIZE; i++){
        newStack->booknames[i] = "";
    }

    // Return pointer to new stack
    return newStack;
}


void destroy(stackOfBooks* bookStack){
    // Free anything that hasn't been freed

    // Free the stack
    free(bookStack);
}


stackOfBooks* push(char* newBook, stackOfBooks* bookStack){
    // Update the head index
    bookStack->bookAtTop++;

    // Put the element in the array
    bookStack->booknames[bookStack->bookAtTop] = newBook;

    return bookStack;
}


char* pop(stackOfBooks* bookStack){
    // Get the element from the array
    return bookStack->booknames[bookStack->bookAtTop--];

    // Update the head index???
}


char* peek(stackOfBooks* bookStack){
    // Get the element from the array
    return bookStack->booknames[bookStack->bookAtTop];
```

push()

pop()

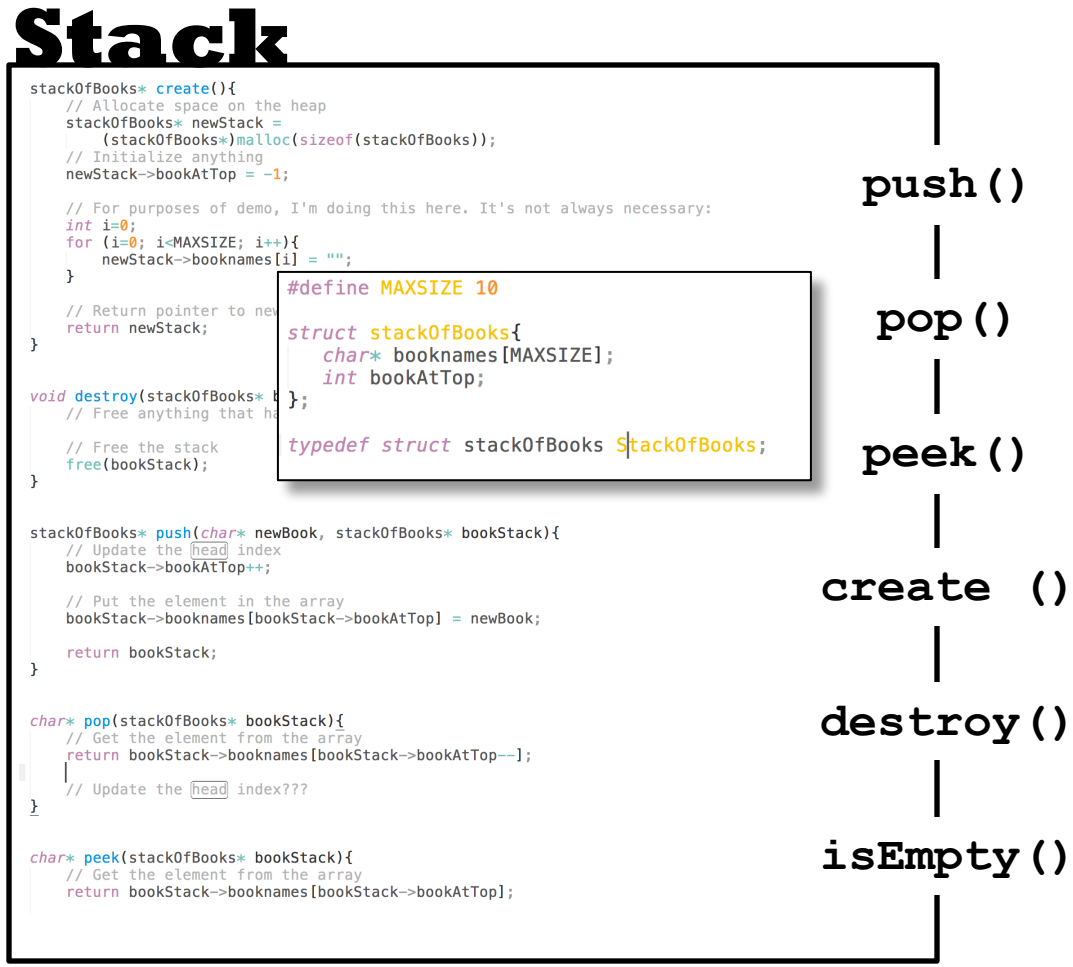
peek()

create ()

destroy()

isEmpty()

# Stack can be implemented with an array…



# …or with a LinkedList.

## Stack

```
stackOfBooks* create(){
    // Allocate space on the heap
    stackOfBooks* newStack =
        (stackOfBooks*)malloc(sizeof(stackOfBooks));
    // Initialize anything
    newStack->bookAtTop = -1;

    // For purposes of demo, I'm doing this here. It's not always necessary:
    int i=0;
    for (i=0; i<MAXSIZE; i++){
        newStack->booknames[i] = "";
    }

    // Retu
    return
}

void destro
    // Free

    // Free
    free(bookStack);
}

stackOfBooks* push(char* newBook, stackOfBooks* bookStack){
    // Update the head index
    bookStack->bookAtTop++;

    // Put the element in the array
    bookStack->booknames[bookStack->bookAtTop] = newBook;

    return bookStack;
}

char* pop(stackOfBooks* bookStack){
    // Get the element from the array
    return bookStack->booknames[bookStack->bookAtTop--];

    // Update the head index???
}

char* peek(stackOfBooks* bookStack){
    // Get the element from the array
    return bookStack->booknames[bookStack->bookAtTop];
```

```
struct stackOfBooks{
    BookNode headOfList;
}

typedef struct stackOfBooks StackOfBooks;
```

push()

pop()

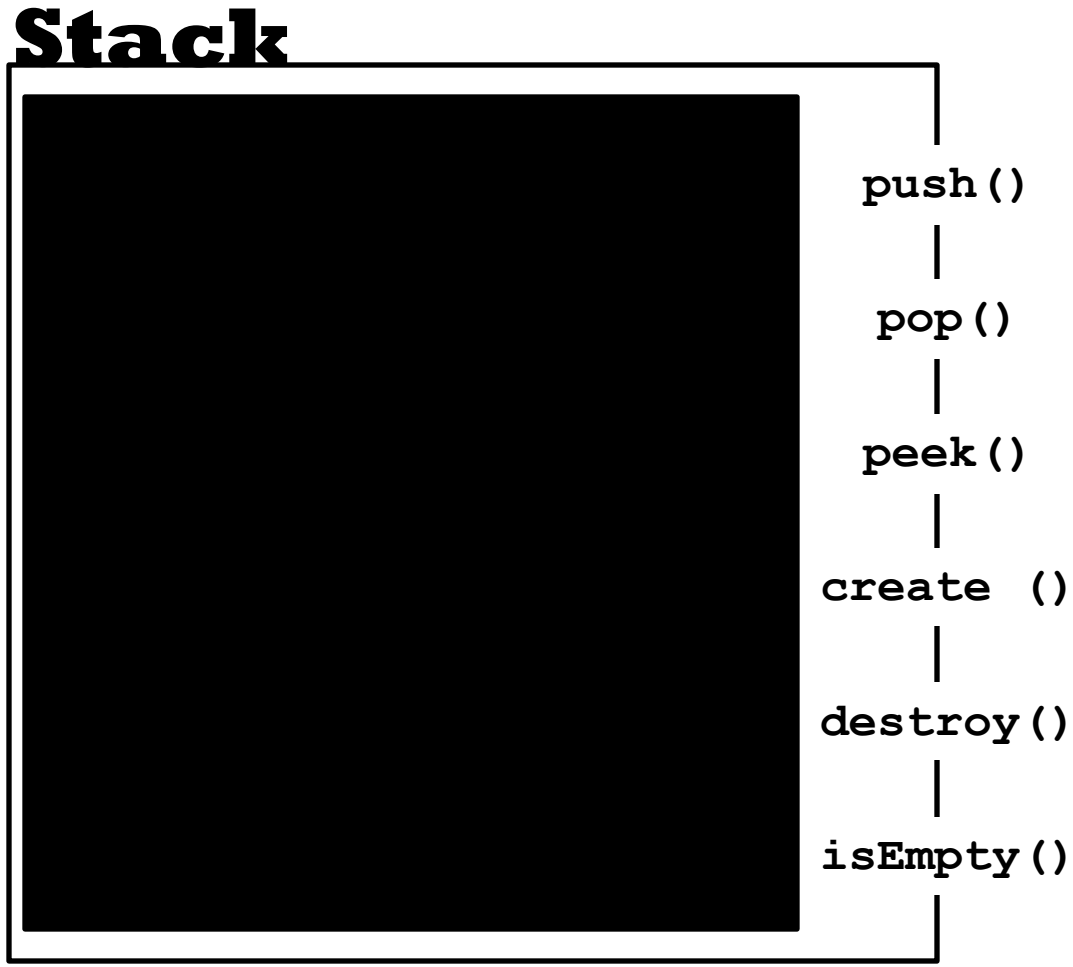
peek()

create ()

destroy()

isEmpty()

# But even as the code changes, we know the operations will stay the same.

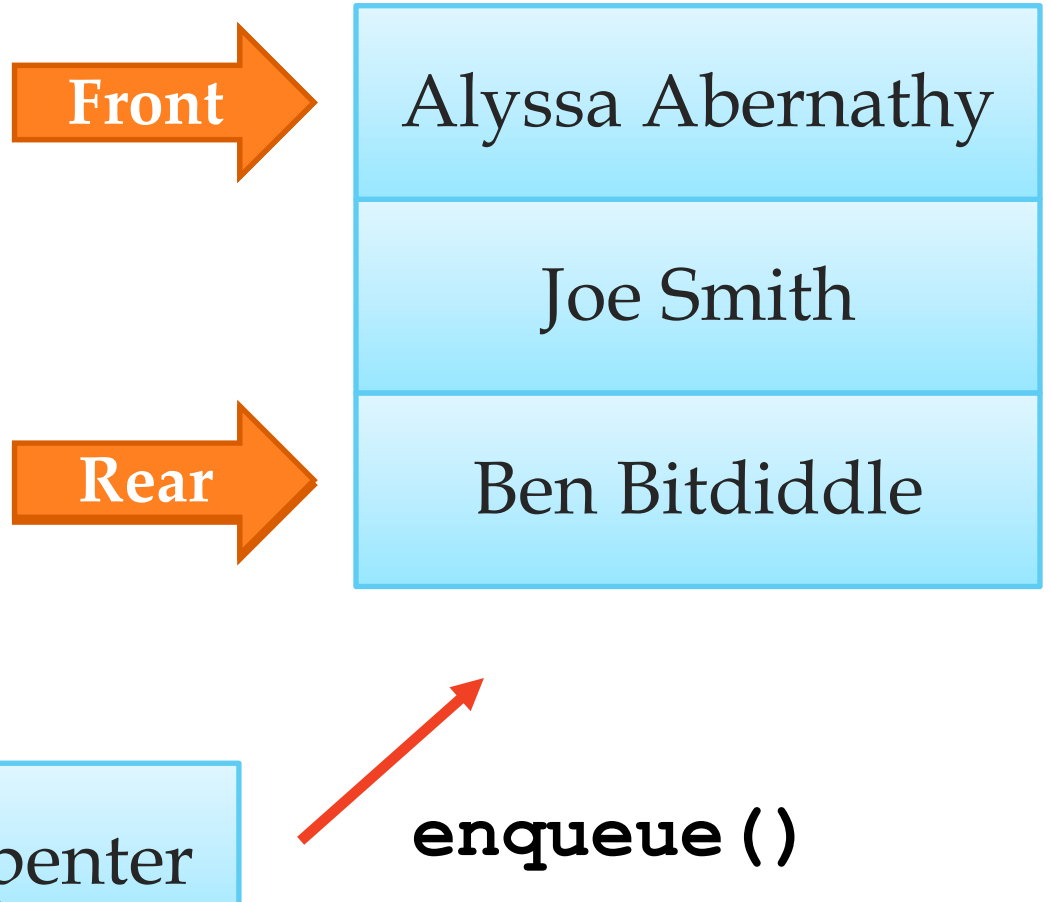


- Abstract Data Types (ADTs) define the operations or interface of a data structure
- The opposite of an ADT is a Concrete Data Type (CDT)
- ADTs demonstrate how defining the interface allows us to decouple the implementation

# QUEUES

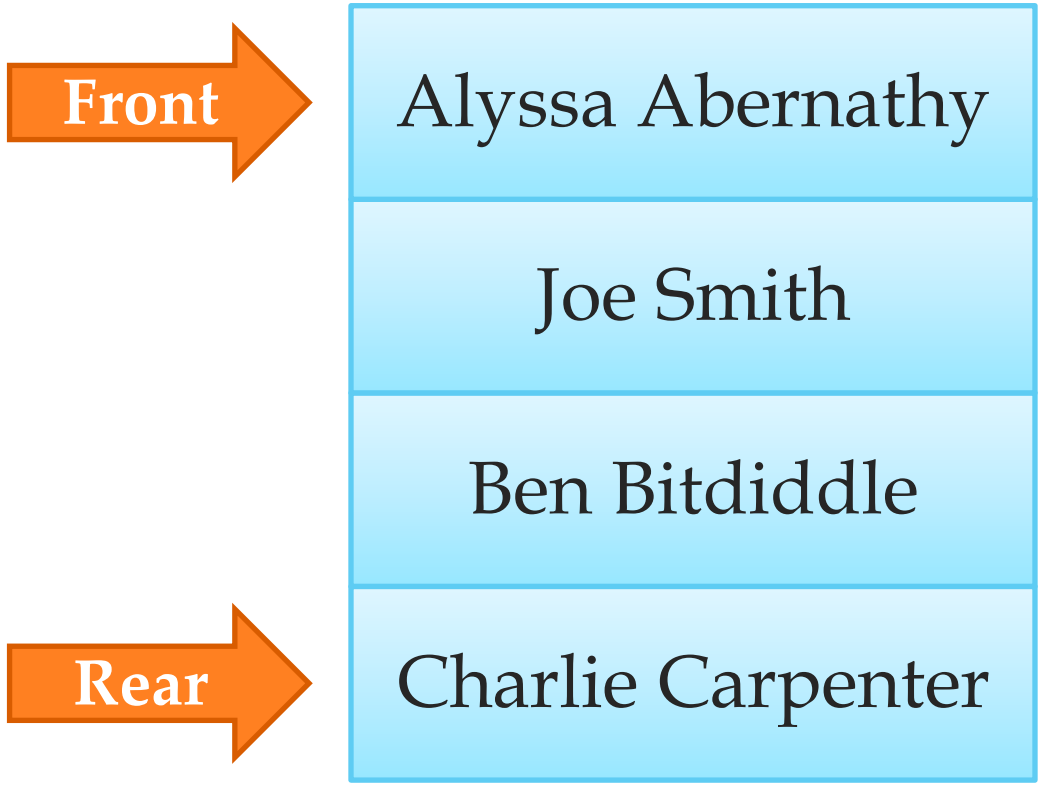
A ***Queue*** allows us to process elements in the order they arrive.

# What is a queue?



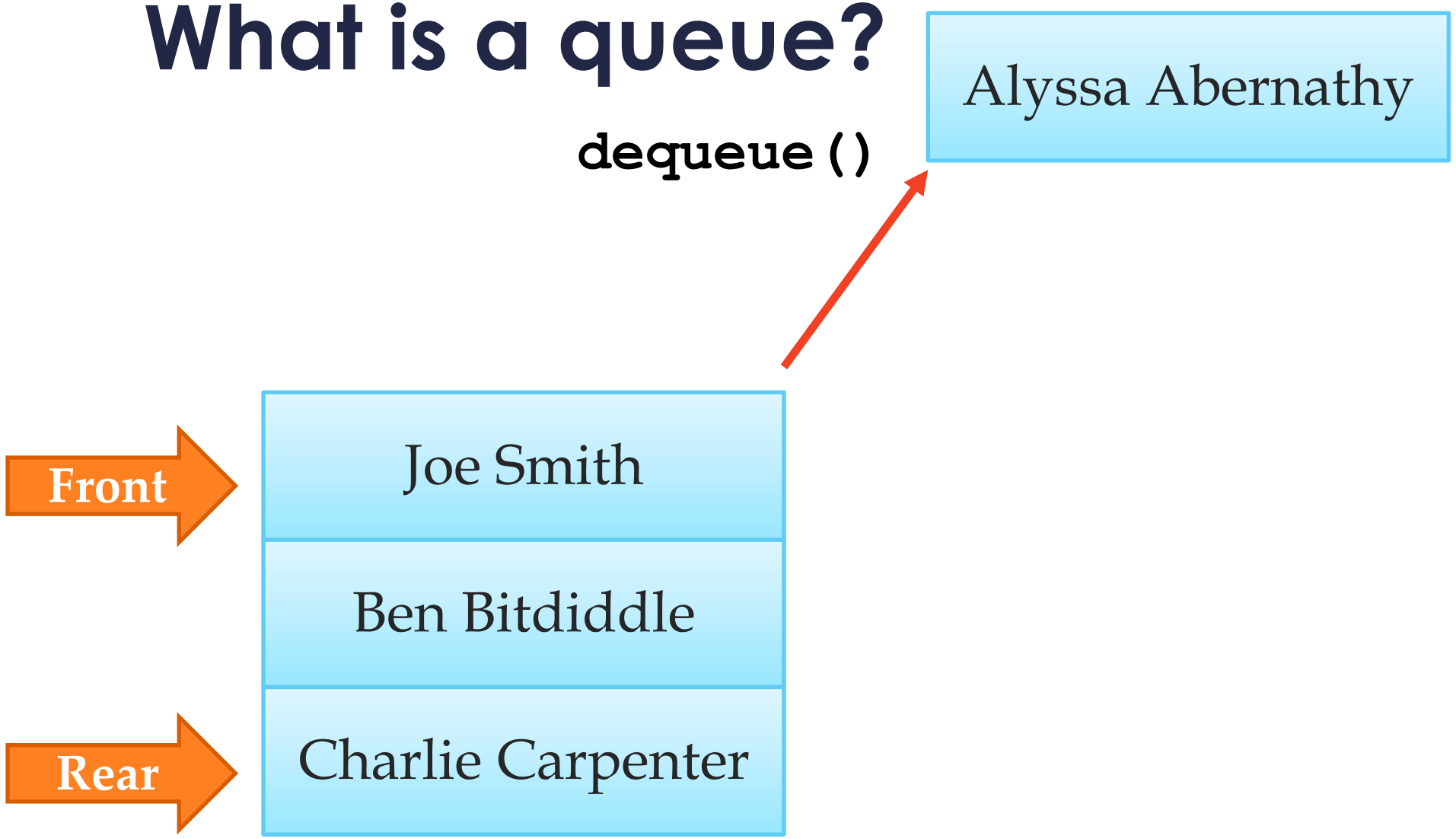
# What is a queue?

`dequeue()`



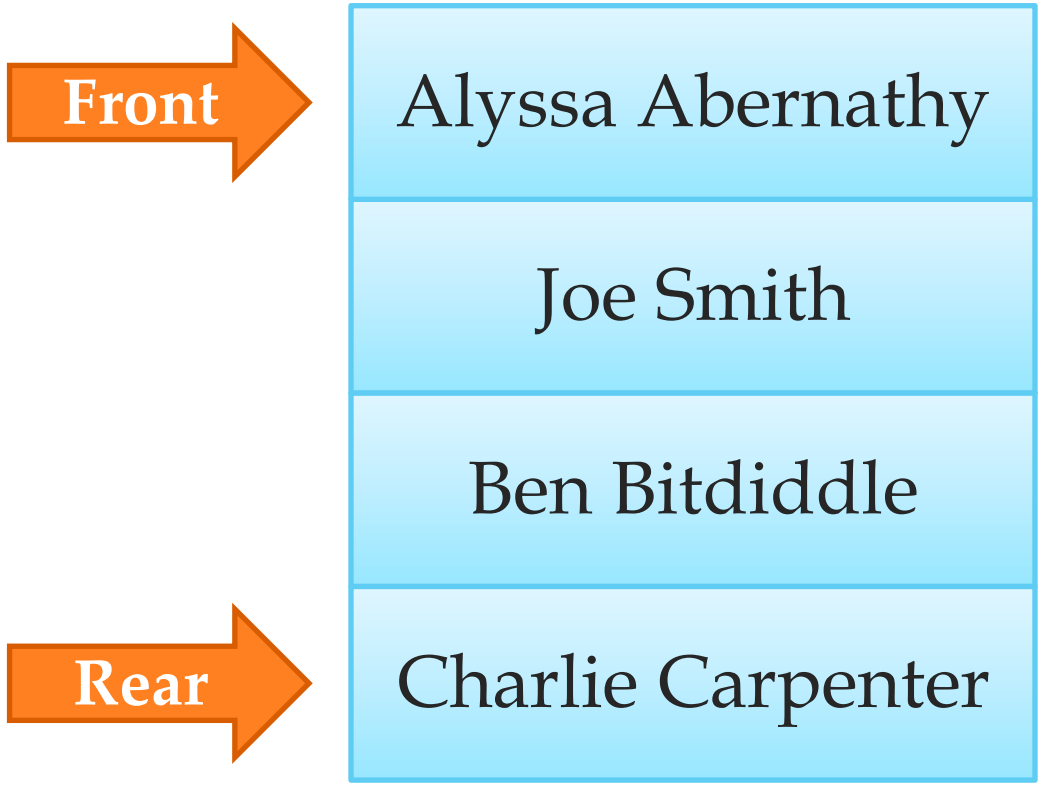
# What is a queue?

`dequeue()`

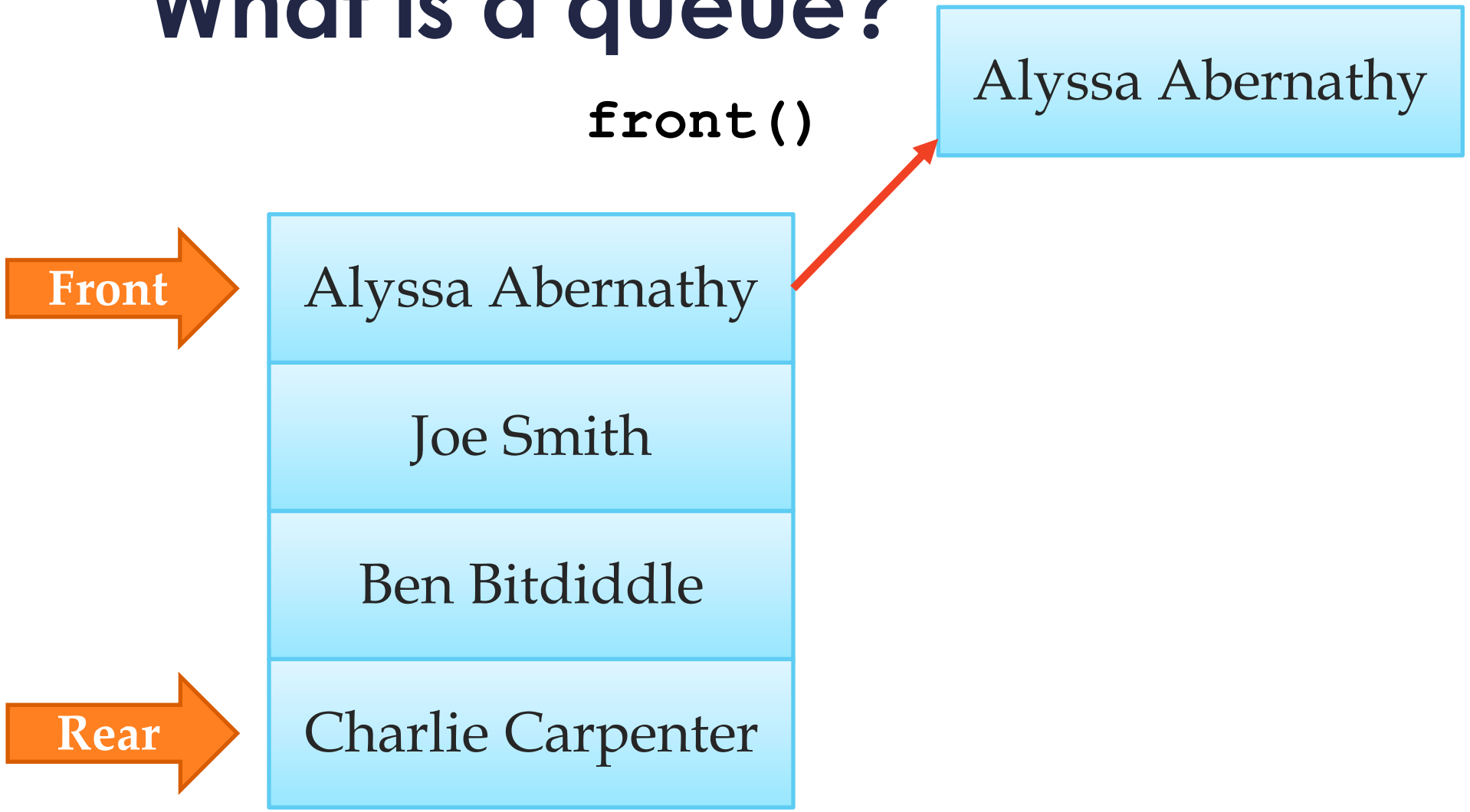


# What is a queue?

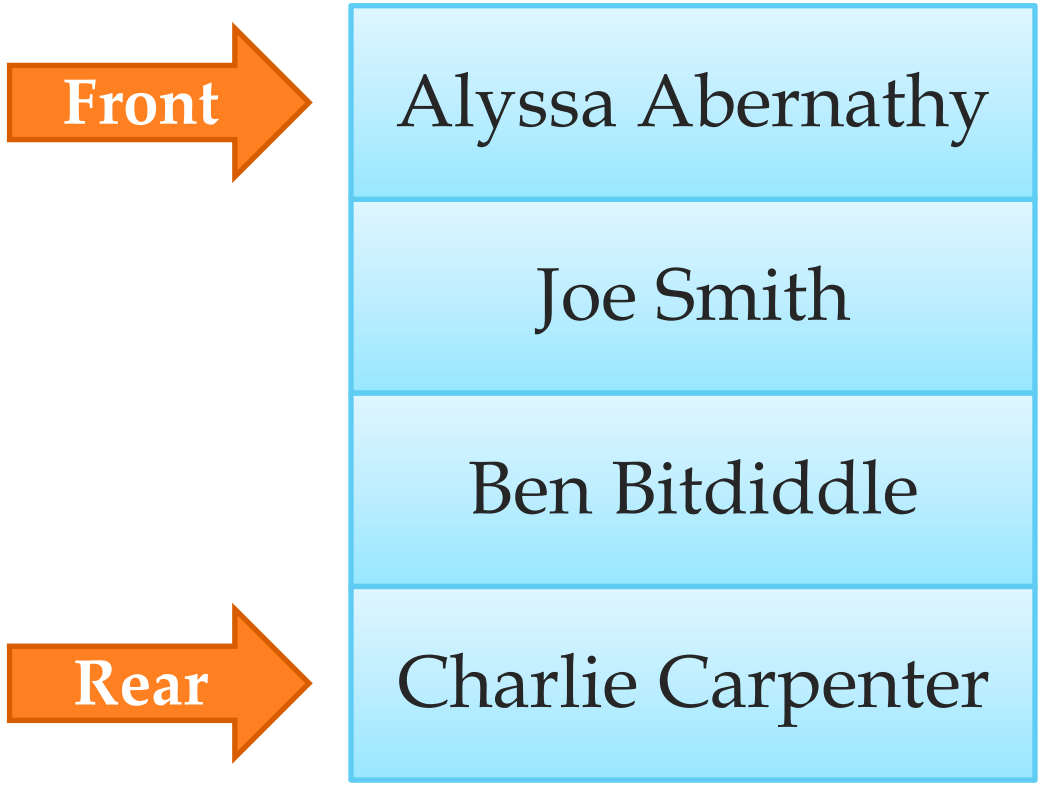
`front()`



# What is a queue?

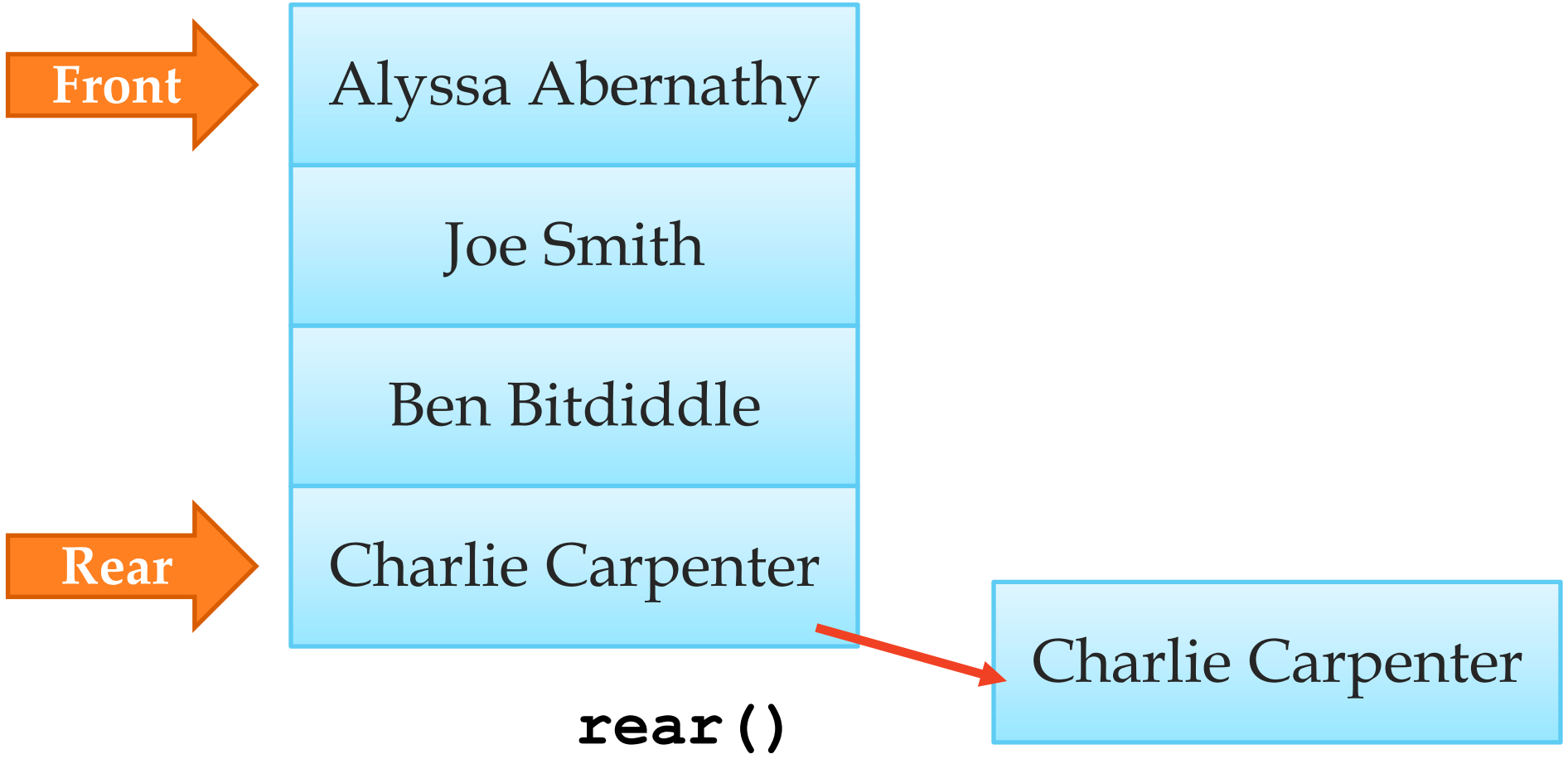


# What is a queue?



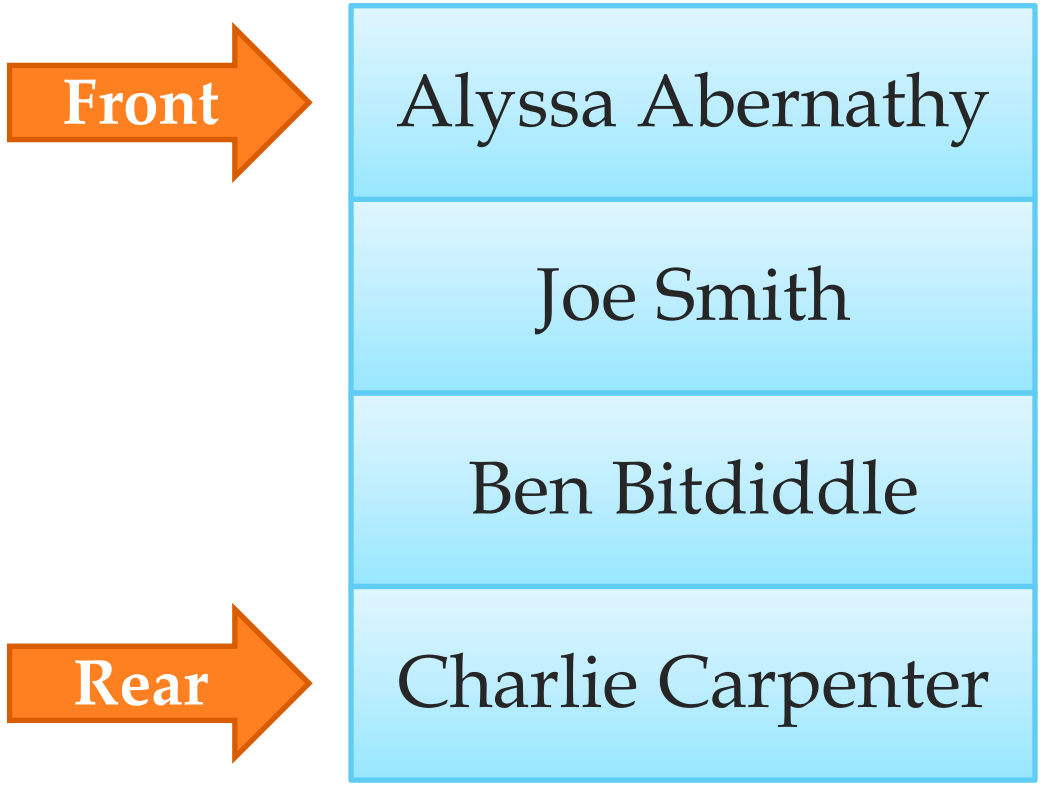

`rear()`

# What is a queue?



# What is a queue?

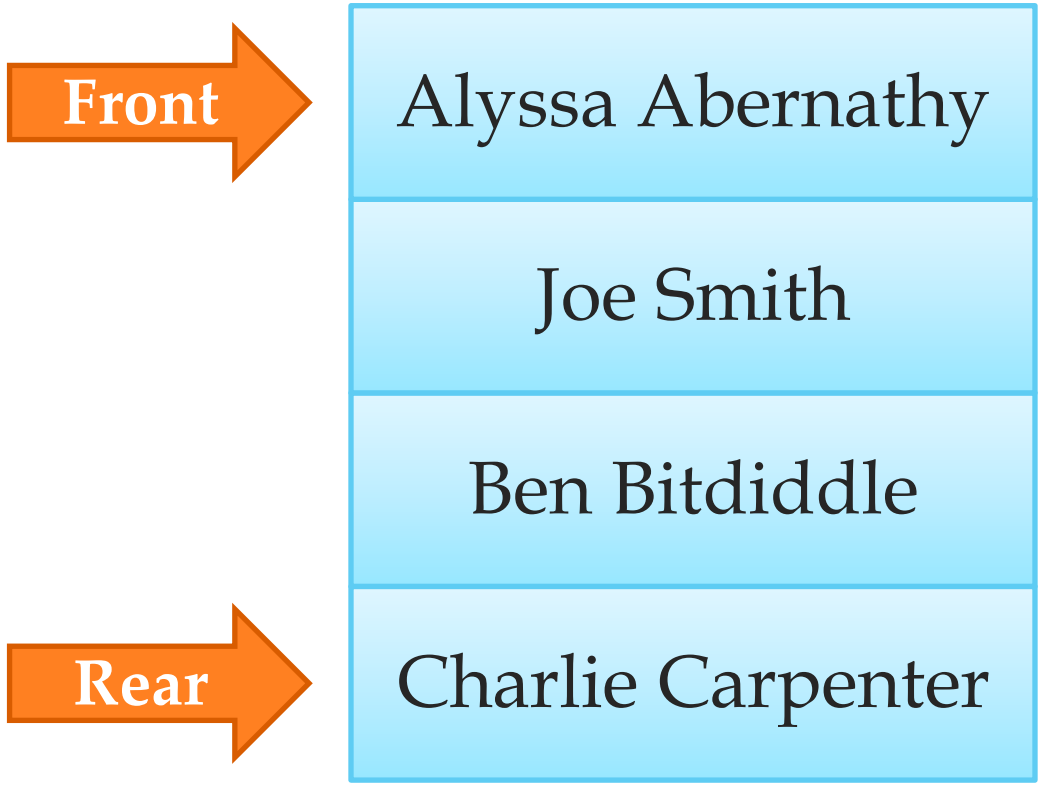
`dequeue()` `front()`




`enqueue()` `rear()`

# What is a queue?

`dequeue()` `front()`



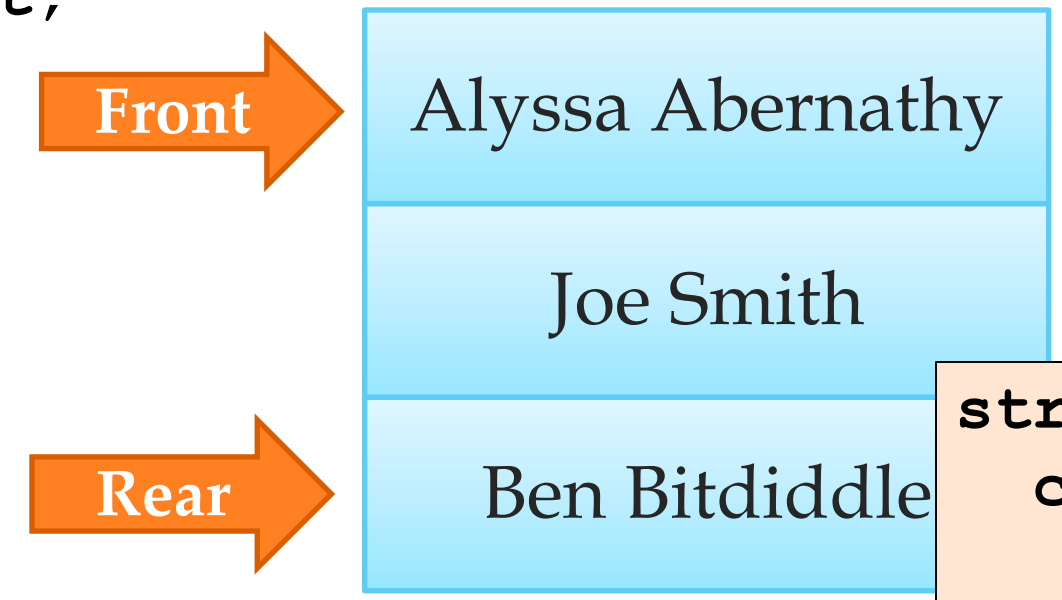

**FIFO:**
**First In,**
**First Out**

`enqueue()` `rear()`

# What is a queue in C?

# What is a queue in C?

```
int front;
```




```
char* customers[];
```

```
int rear;
```

```
struct queueOfCustomers{
   char* customers[SIZE];
    int front;
    int rear;
};
```

# Building a Queue with an array

How many elements does the queue have?

```
size = rear - front;
```

# Enqueue: Adding elements

**What changes with our pointers when we add an element?**

**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

```
enqueue("Ben Bitdiddle");
```

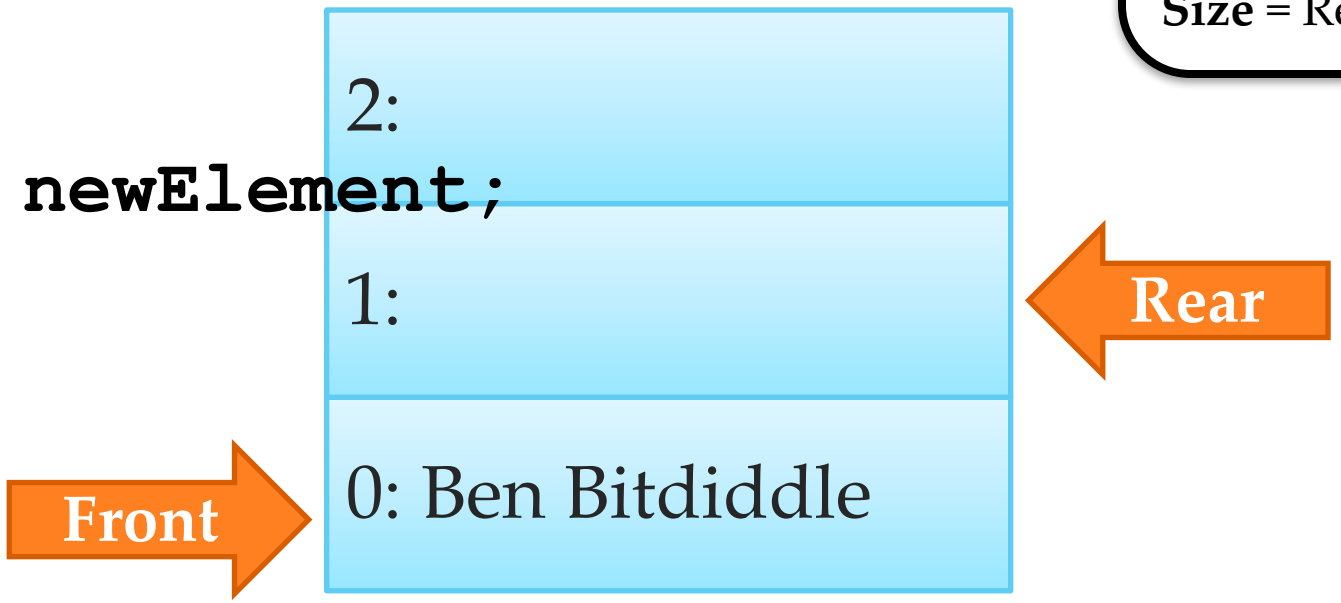
# Enqueue: Adding elements

What changes with our pointers when we add an element?

**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

```
enqueue("Ben Bitdiddle");
```

```
array[rear] = newElement;
rear++;
```



# Enqueue: Adding elements

**What changes with our pointers when we add an element?**

```
enqueue("Joe Smith");
```

```
array[rear] = newElement;
rear++;
```


**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

# Enqueue: Adding elements

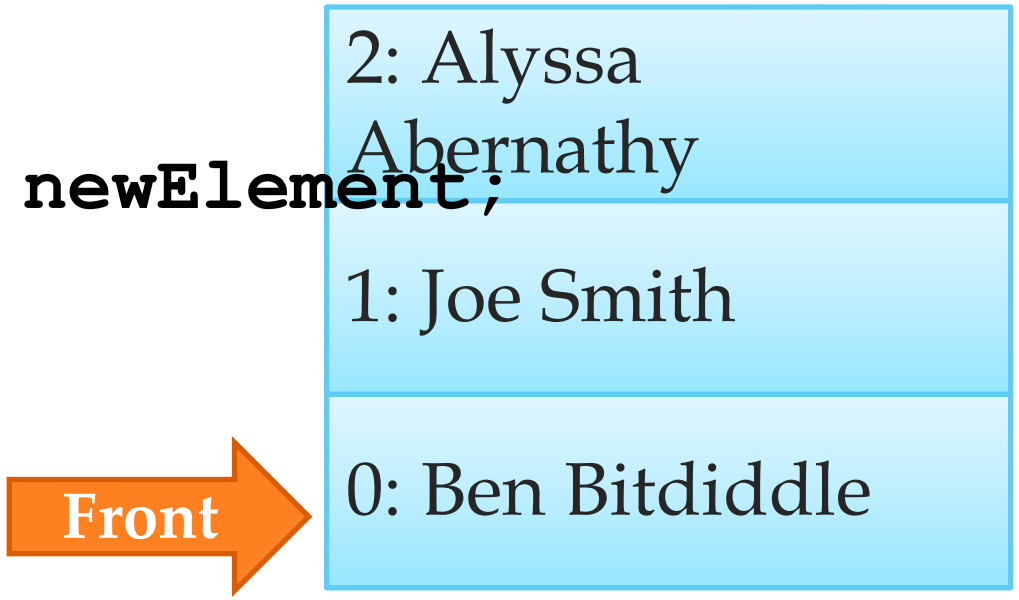
What changes with our pointers when we add an element?

!!!

Rear

```
enqueue("Alyssa Abernathy");
```

```
array[rear] = newElement;
rear++;
```

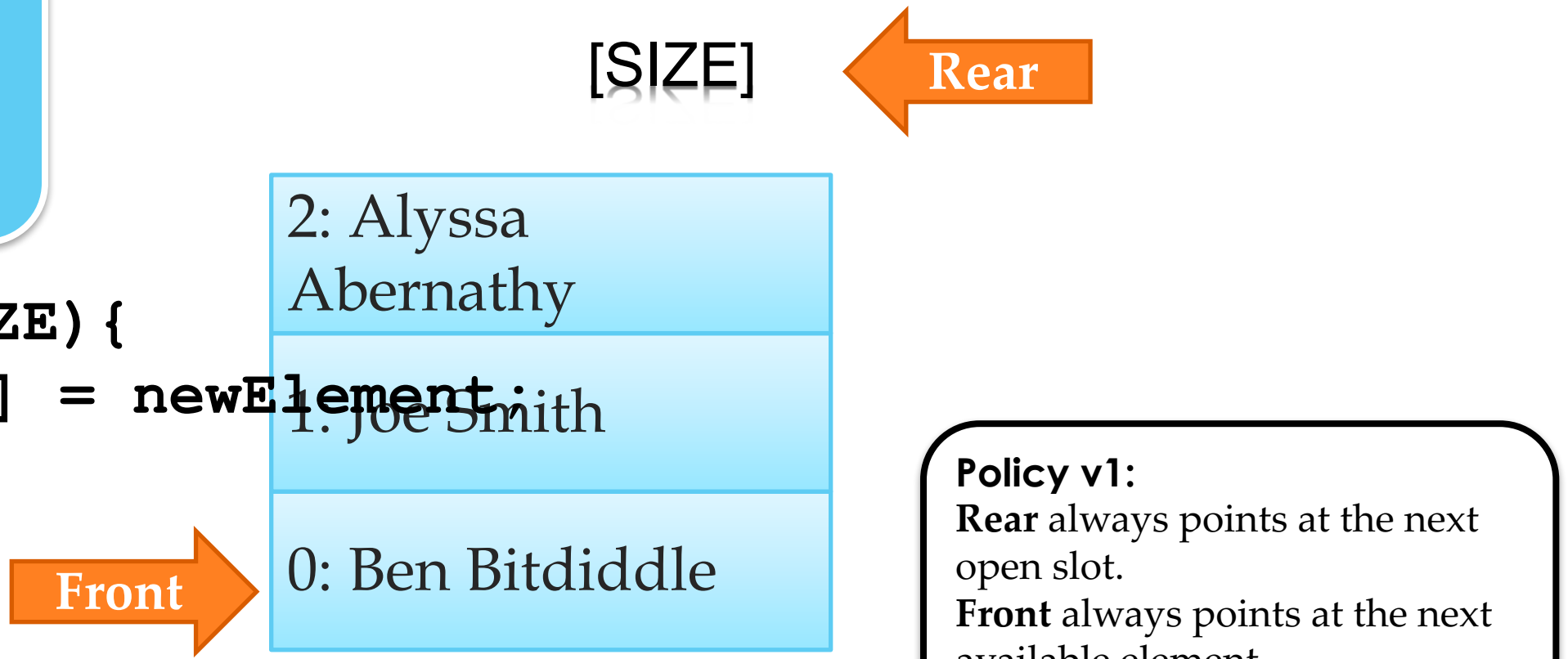



**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

# Enqueue: Adding elements

Add a guard to `enqueue()`, so we don't insert an element into a non-existent slot:

```
if (rear < SIZE){
    array[rear] = newElement;
    rear++;
}
```




**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

# Dequeue: Removing elements

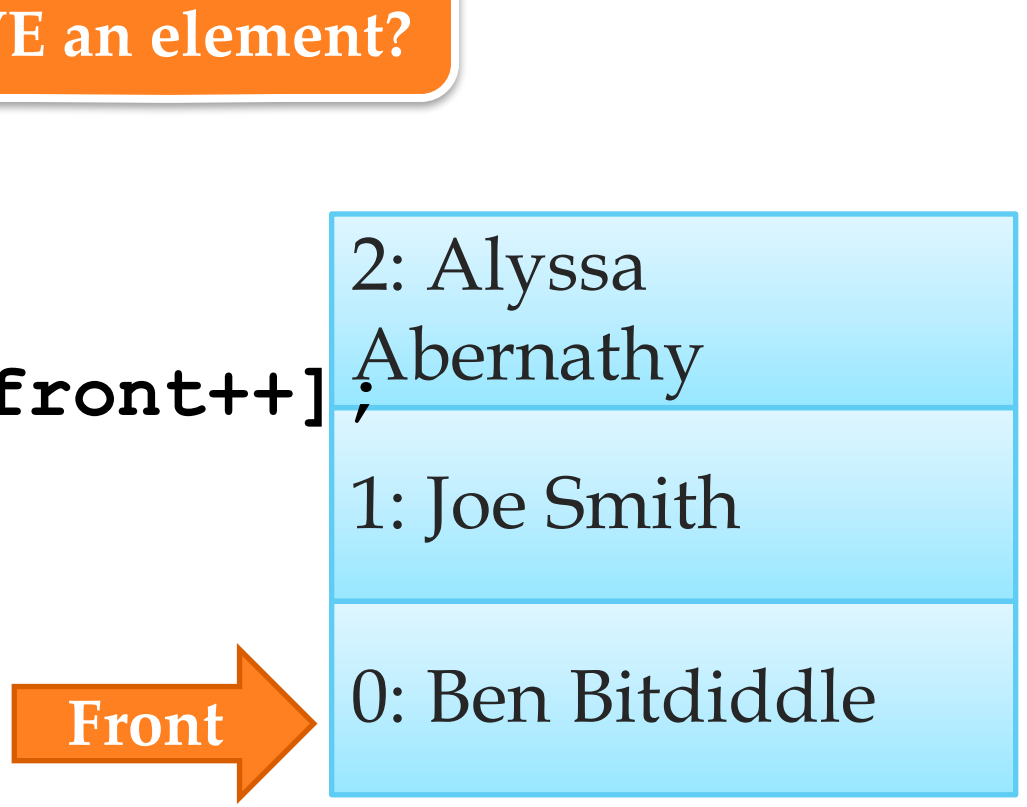
**What changes with our pointers when we REMOVE an element?**


**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

# Dequeue: Removing elements

**What changes with our pointers when we REMOVE an element?**

```
dequeue();

return array[front++];
```

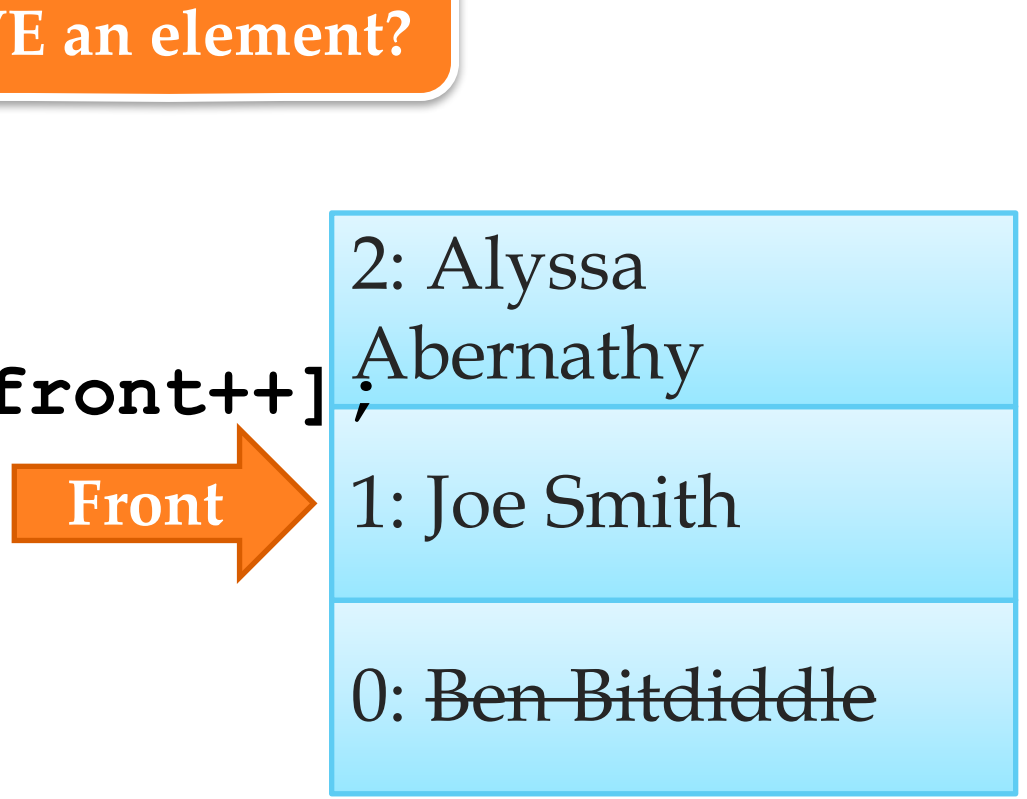



**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

# Dequeue: Removing elements

What changes with our pointers when we REMOVE an element?

```
dequeue();

return array[front++];
```




**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

# Dequeue: Removing elements

What changes with our pointers when we REMOVE an element?

```
dequeue();

return array[front++];
```


**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

# Dequeue: Removing elements

What changes with our pointers when we REMOVE an element?

```
dequeue();

return array[front++];
```

Front

Rear


**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

# Dequeue: Removing elements

What changes with our pointers when we REMOVE an element?

```
dequeue();

return array[front++];
```

Front

Rear

2: ~~Alyssa Abernathy~~

1: ~~Joe Smith~~

0: ~~Ben Bitdiddle~~

Also, add a guard to dequeue(), so we don't try to access an empty slot.

- if queue isEmpty()?
- if front == SIZE?

Coming back …

**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

# Dequeue: Removing elements

What changes with our pointers when we REMOVE an element?

Front

Rear

2: Alyssa

Aaliyah

```
return array[front++];
```

1: Joe Smith

0: ~~Ben Bitdiddle~~

**Now What?**

**Can we reset our queue?**

**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

# Can we re-use empty slots?


**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
**Size** = Rear – Front

# Can we re-use empty slots?

**Policy v1:**
**Rear** always points at the next open slot.
**Front** always points at the next available element.
~~**Size** = Rear – Front~~


```
// Update Enqueue to point
// the rear at zero to reuse
// now-empty slots
    array[rear] = newElement;
    rear++;
if (rear >= SIZE){
    rear = 0;
}
```

# Re-using empty slots: tracking size

**Policy v2:**
**Rear:** always points at the next open slot.
**Front:** always points at the next available element.
**Size:** keep track of separately.


```
// Update Enqueue to point
// the rear at zero to reuse
// now-empty slots
    array[rear] = newElement;
    rear++;
if (rear >= SIZE){
    rear = 0;
}
```

# Re-using empty slots: tracking size

**Policy v2:**
**Rear:** always points at the next open slot.
**Front:** always points at the next available element.
**Size:** keep track of separately.


```
// Update Enqueue to point
// the rear at zero to reuse
// now-empty slots
    array[rear] = newElement;
    rear++;
if (rear >= SIZE){
    rear = 0;
}
```

```
struct queueOfCustomers{
    char* customers[SIZE];
    int front;
    int rear;
    int size;
};
```

# Hmmm… what's happening here?

**Policy v2:**
**Rear:** always points at the next open slot.
**Front:** always points at the next available element.
**Size:** keep track of separately.

```
enqueue("Harry Potter");
```

# Updating the policy again:

**Policy v3:**
**Rear:** always points at the next open slot.
**Front:** always points at the next available element.
**Size:** keep track of separately.
If queue is **full**, can't enqueue.
If queue is **empty**, can't dequeue.


```
// Update Enqueue to check for full
if (rear == front){
    // QUEUE IS FULL!!
    return;
}
if (rear < SIZE){
    array[rear] = newElement;
    rear++;
    size++;
}
if (rear >= SIZE){
    rear = 0;
}
```

…this is getting complicated.

# Can we be smarter about this?

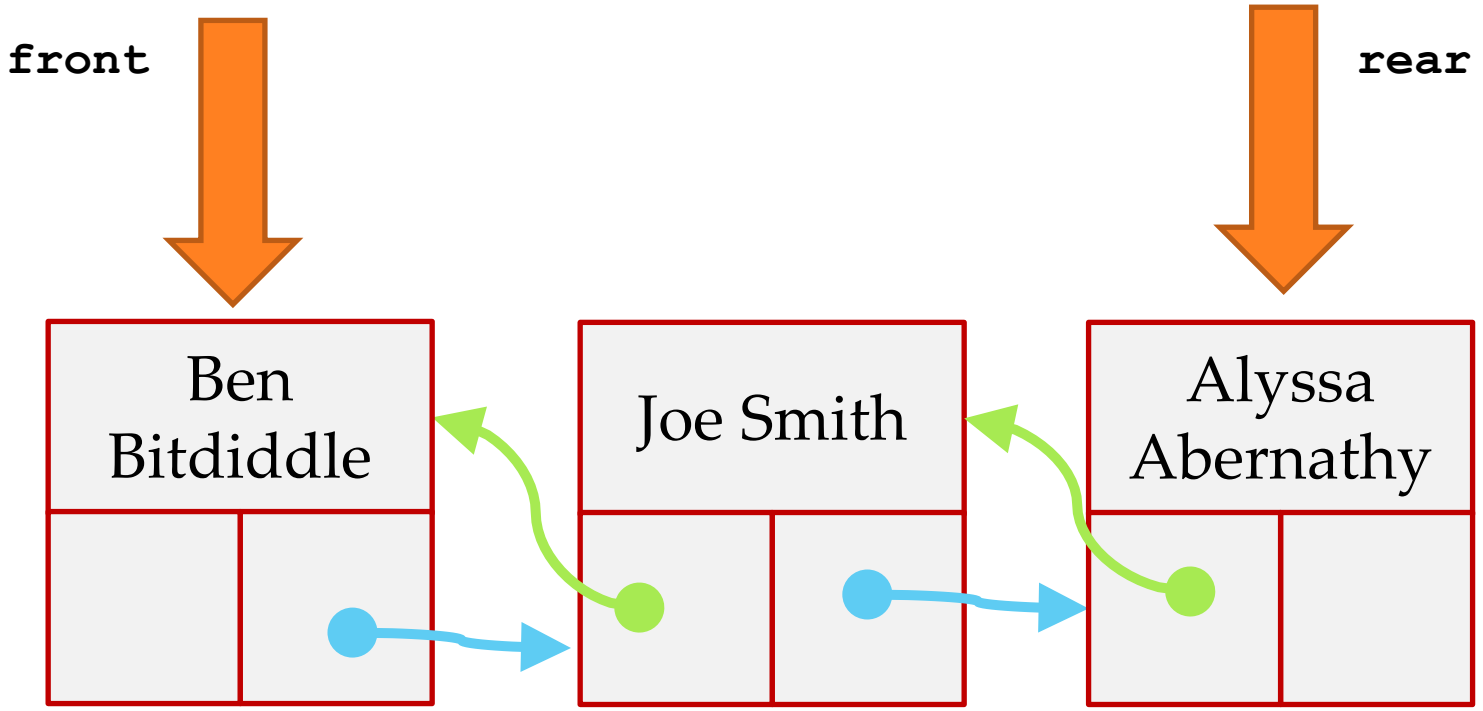
**A queue is an Abstract Data Type. That means we can choose a different implementation if we want!**

**What other data structures can we use?**

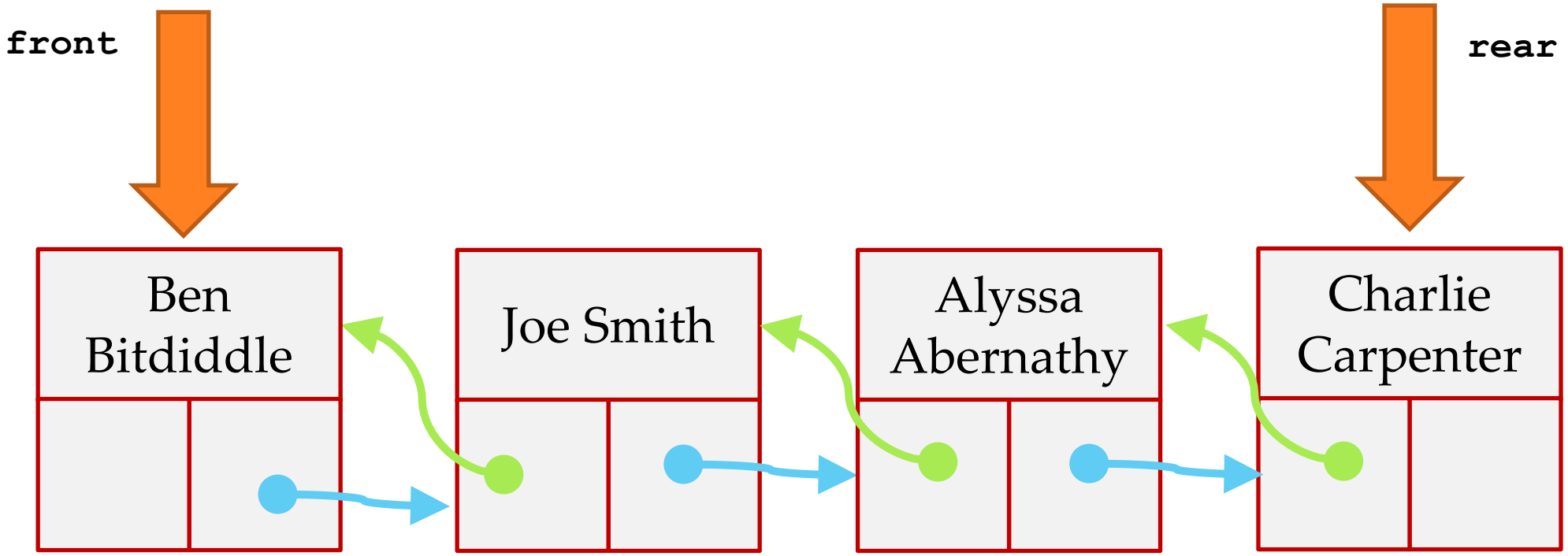
# Linked List?
# Stack?

# Building a Queue with a Linked List

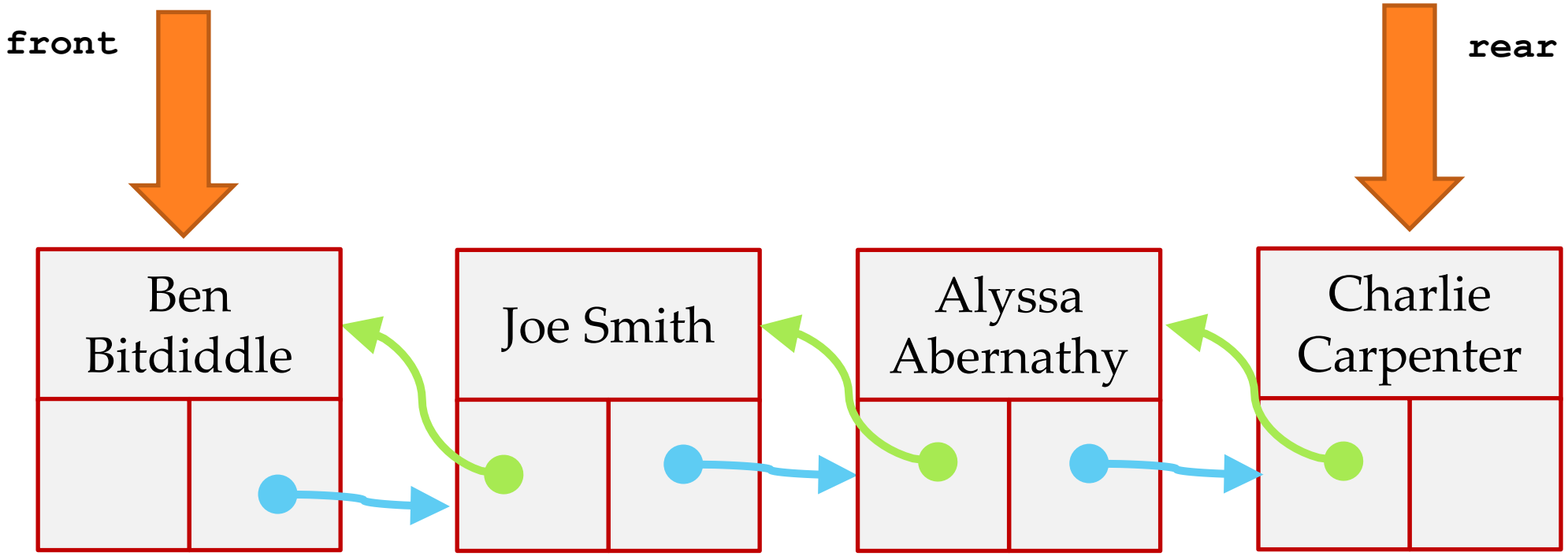
# Building a Queue with Linked List



# Enqueue: Adding an element

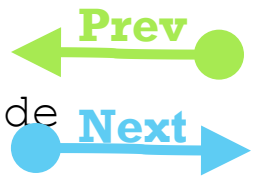



# Enqueue: Adding an element

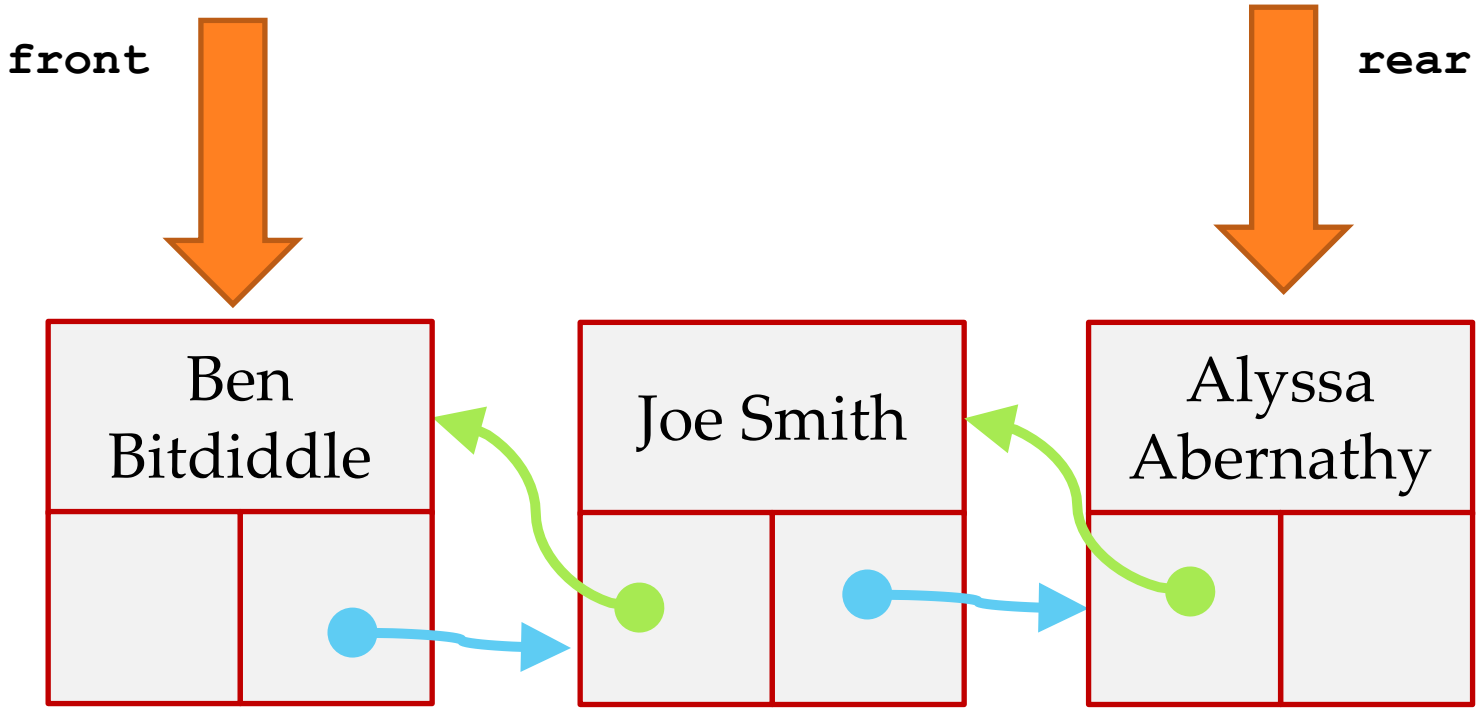



**enqueue():**

```
create a new node
set the next node to null
set the prev node to queue->rear
set queue->rear->nextNode to new node
point rear at new node
```

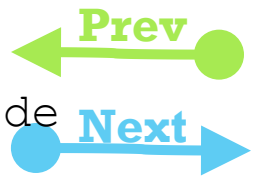


# Enqueue: Adding an element

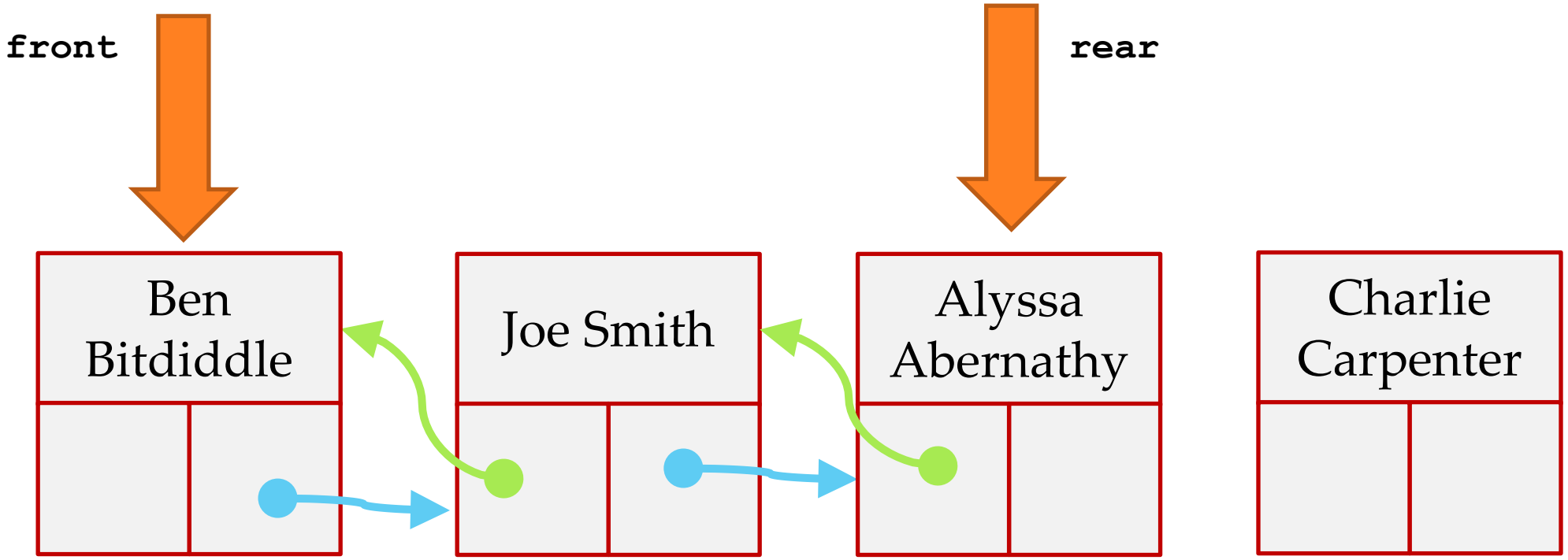



**enqueue("Charlie Carpenter"):**

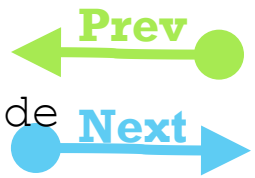
```
create a new node
set the next node to null
set the prev node to queue->rear
set queue->rear->nextNode to new node
point rear at new node
```



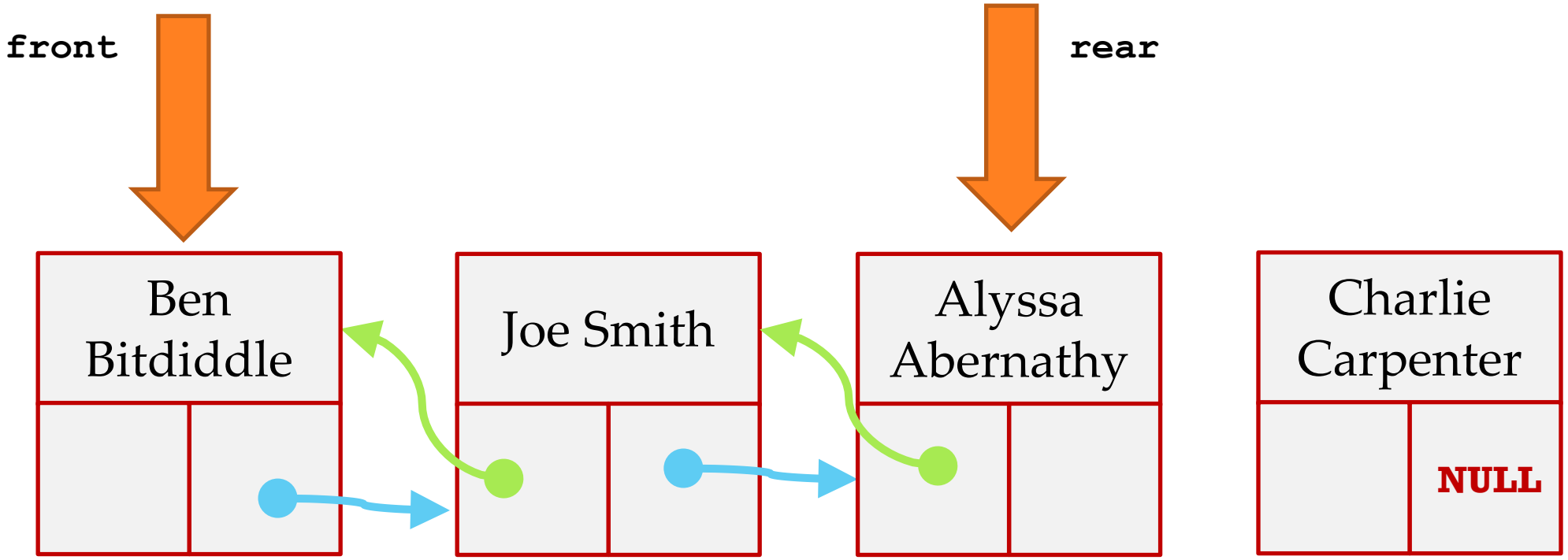
# Enqueue: Adding an element



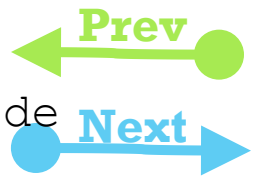

**enqueue("Charlie Carpenter"):**
**create a new node**
set the next node to null
set the prev node to queue->rear
set queue->rear->nextNode to new node
point rear at new node



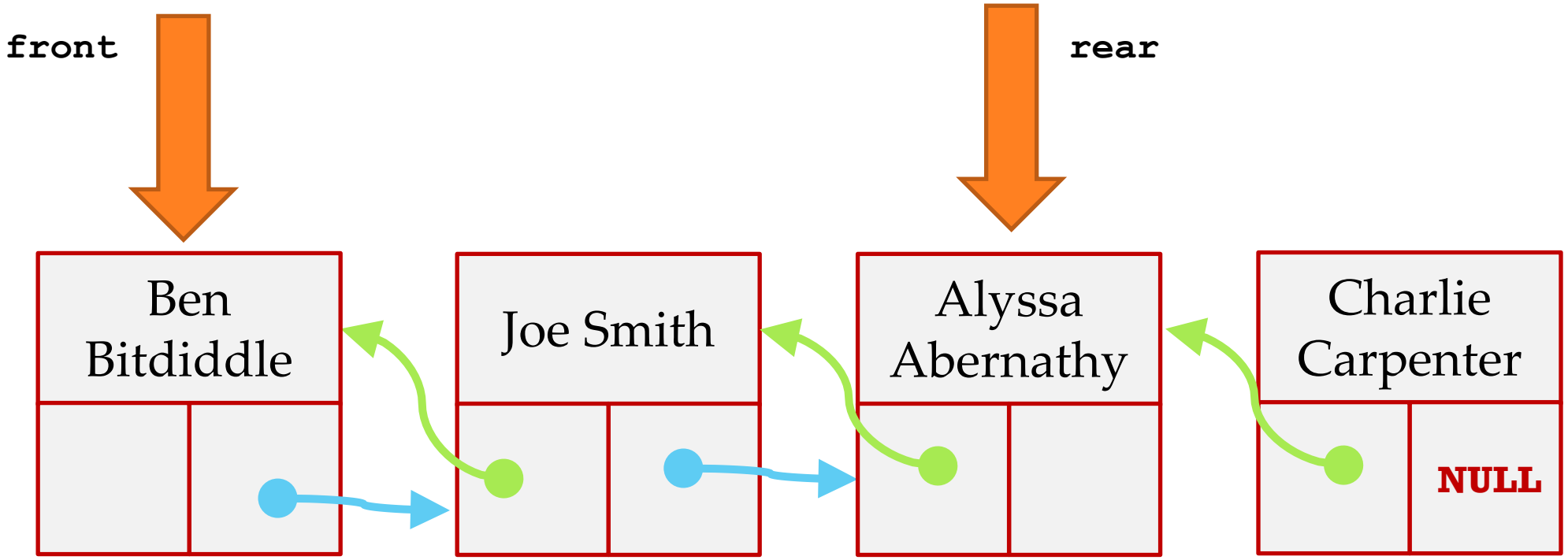
# Enqueue: Adding an element




**enqueue("Charlie Carpenter"):**
create a new node
**set the next node to null**
set the prev node to queue->rear
set queue->rear->nextNode to new node
point rear at new node

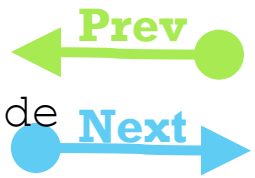


# Enqueue: Adding an element

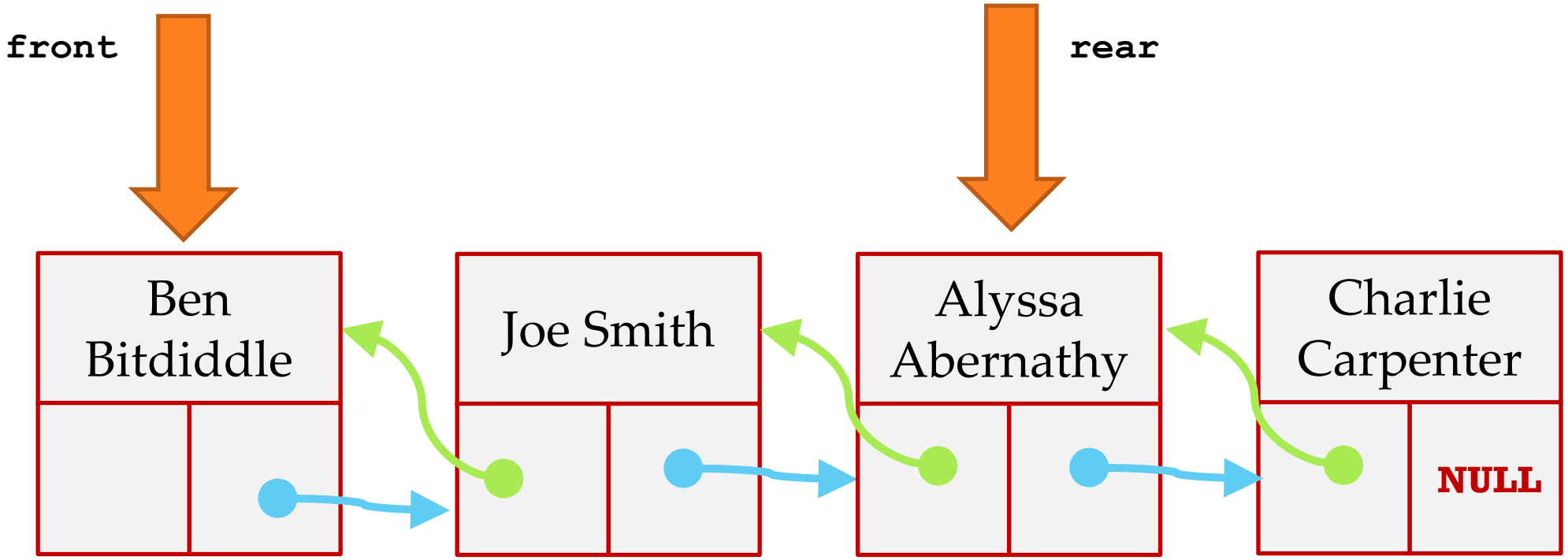



**enqueue("Charlie Carpenter"):**

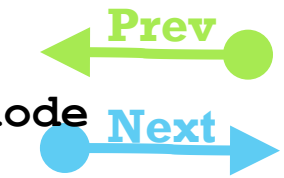
```
create a new node
set the next node to null
set the prev node to queue->rear
set queue->rear->nextNode to new node
point rear at new node
```



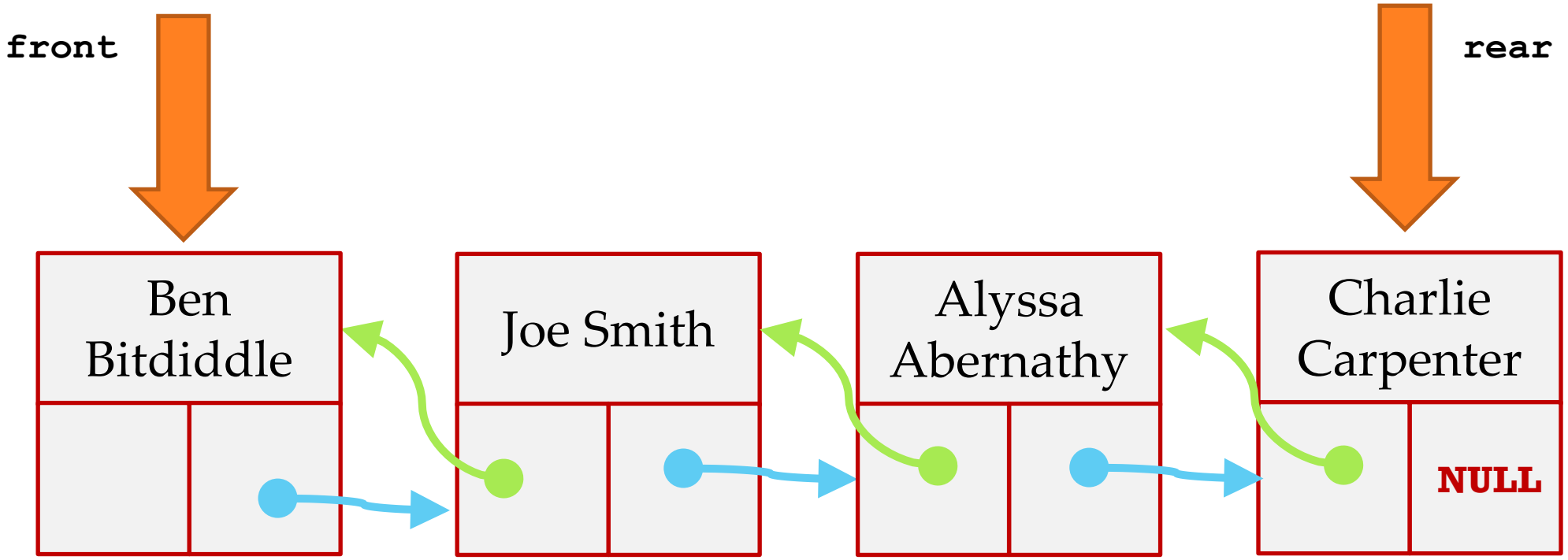
# Enqueue: Adding an element




**enqueue("Charlie Carpenter"):**
create a new node
set the next node to null
set the prev node to queue->rear
**set queue->rear->nextNode to new node**
point rear at new node



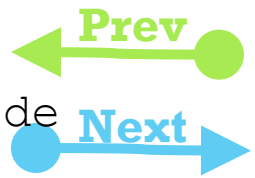
# Enqueue: Adding an element



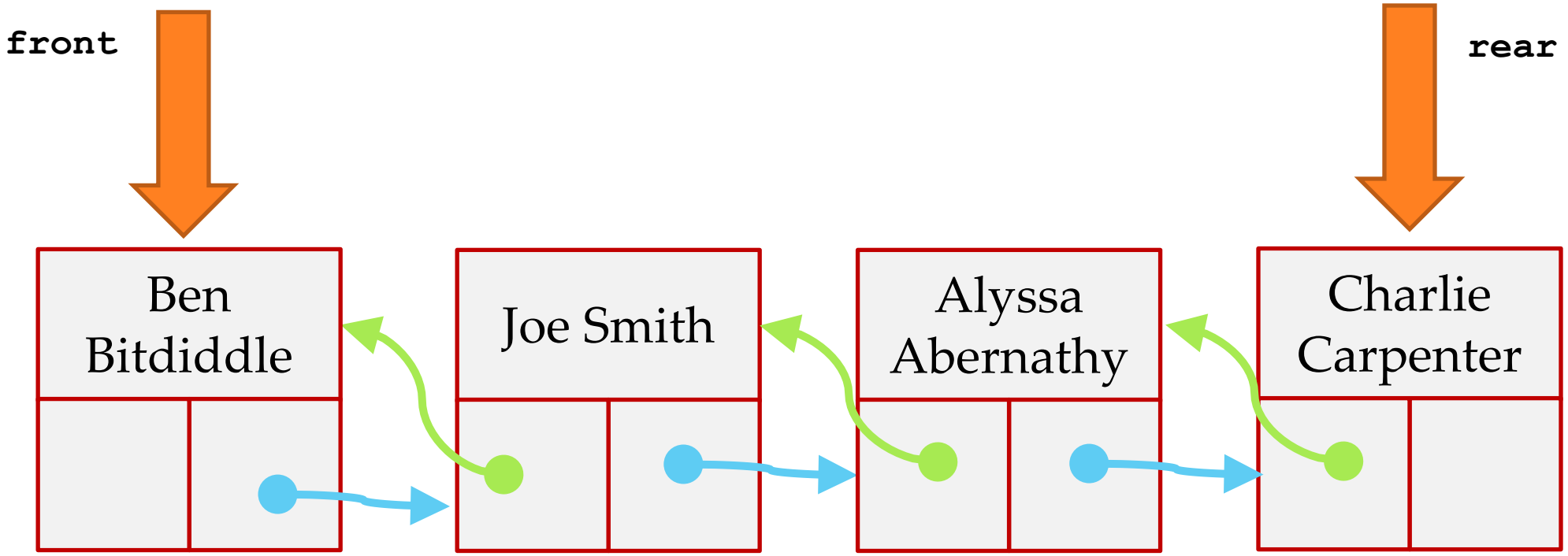

**enqueue("Charlie Carpenter"):**

```
create a new node
set the next node to null
set the prev node to queue->rear
set queue->rear->nextNode to new node
```

**point rear at new node**



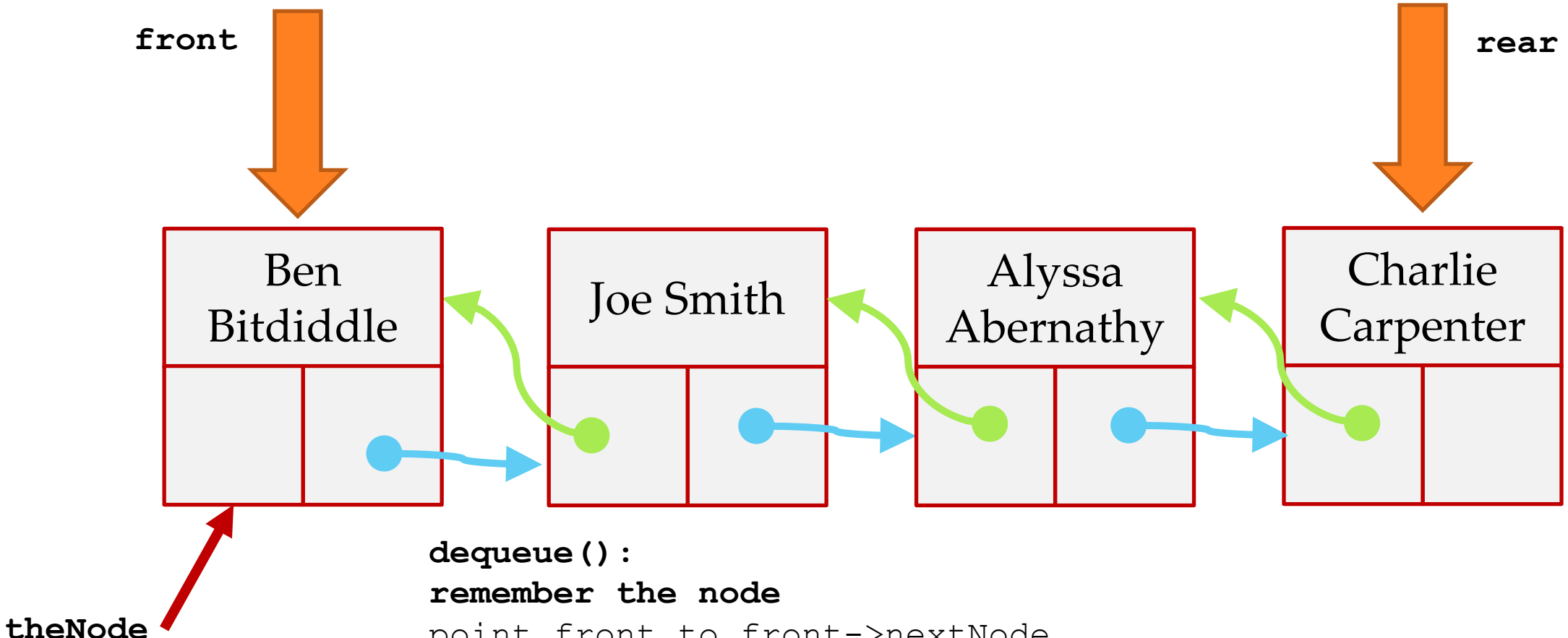
# Dequeue: Removing an element




**dequeue():**

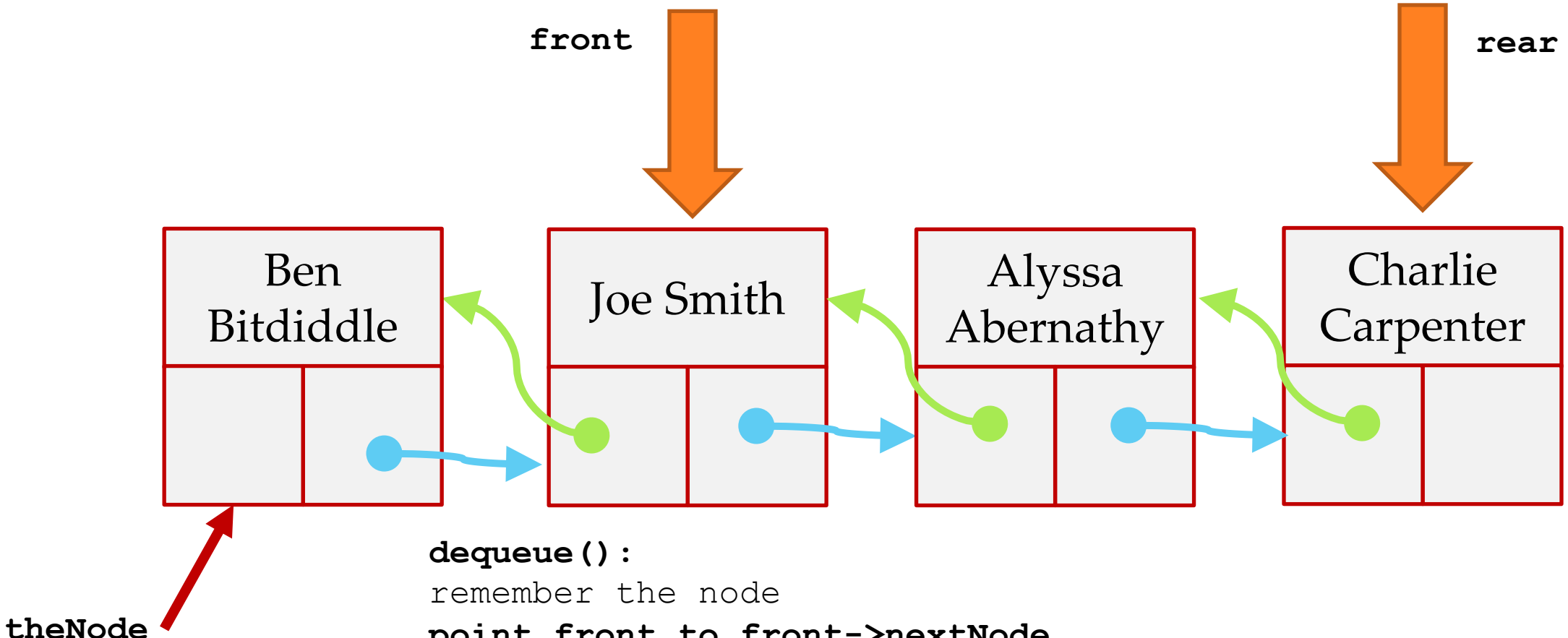
```
remember the node
point front to front->nextNode
point the new front->prevNode to NULL
remember the name
free the node
return the name
```

# Dequeue: Removing an element



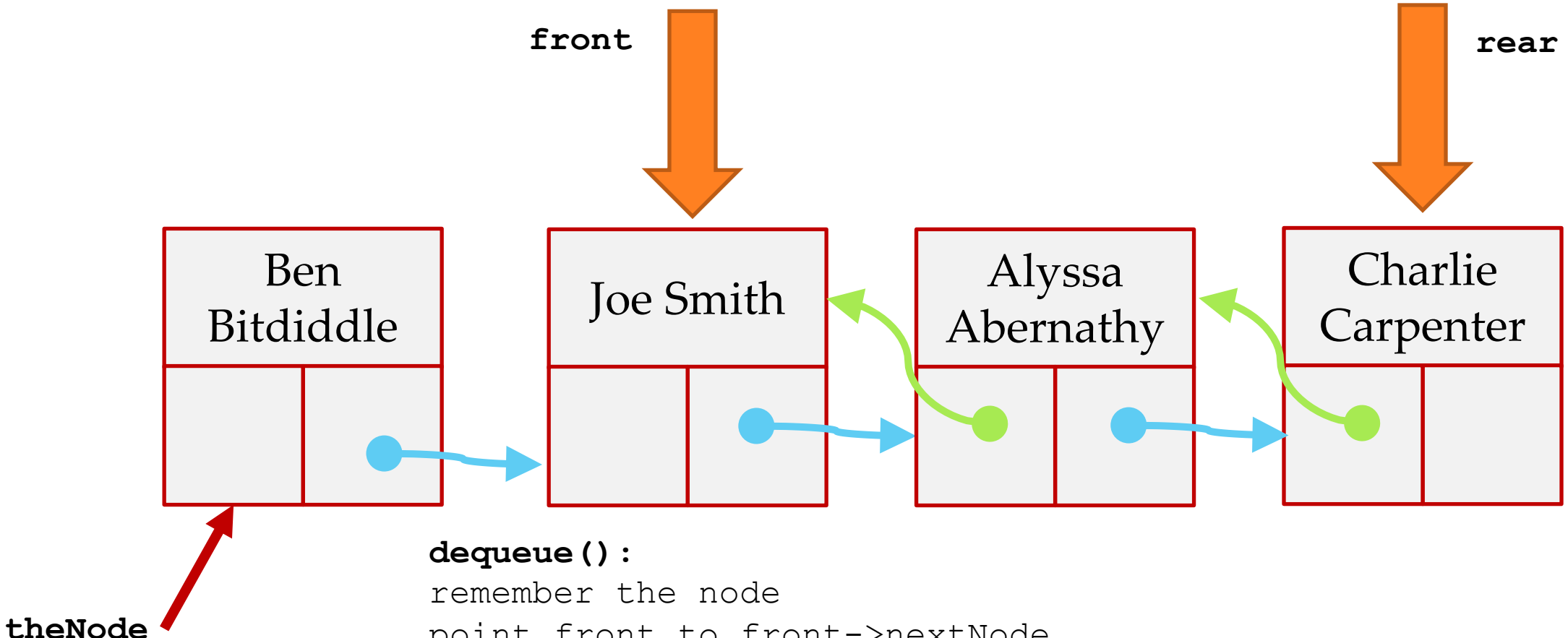

```
dequeue():
remember the node
point front to front->nextNode
point the new front->prevNode to NULL
remember the name
free the node
return the name
```

# Dequeue: Removing an element



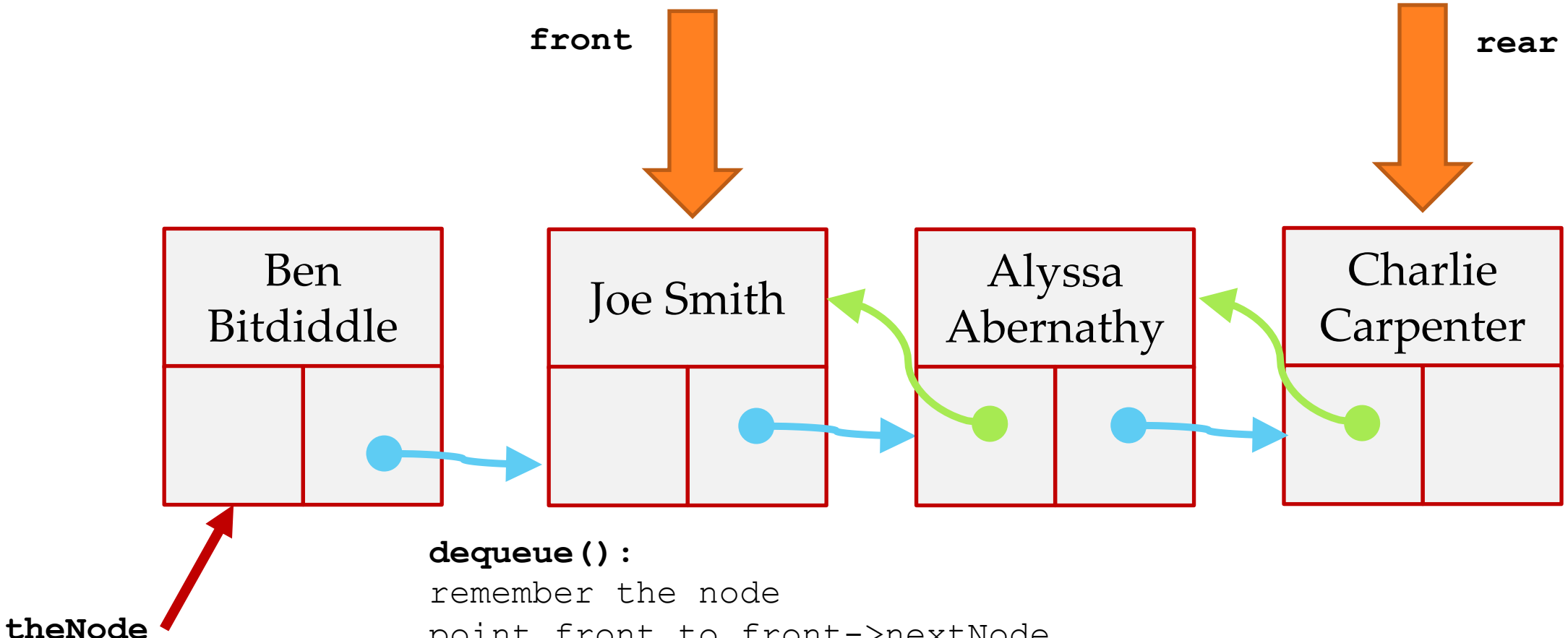

```
dequeue():
remember the node
point front to front->nextNode
point the new front->prevNode to NULL
remember the name
free the node
return the name
```

# Dequeue: Removing an element




```
dequeue():
remember the node
point front to front->nextNode
point the new front->prevNode to NULL
remember the name
free the node
return the name
```

# Dequeue: Removing an element

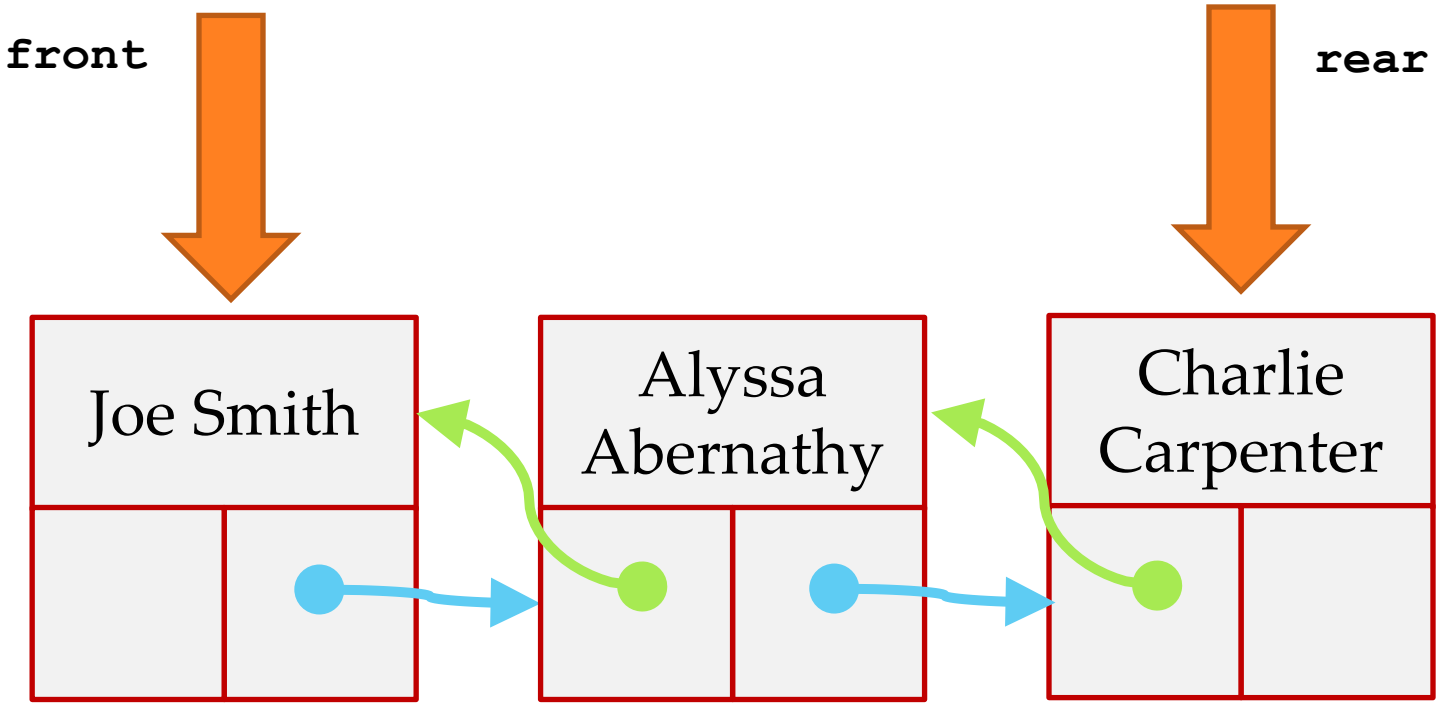



```
dequeue():
remember the node
point front to front->nextNode
point the new front->prevNode to NULL
remember the name          "Ben Bitdiddle"
free the node
return the name
```

Prev

Next

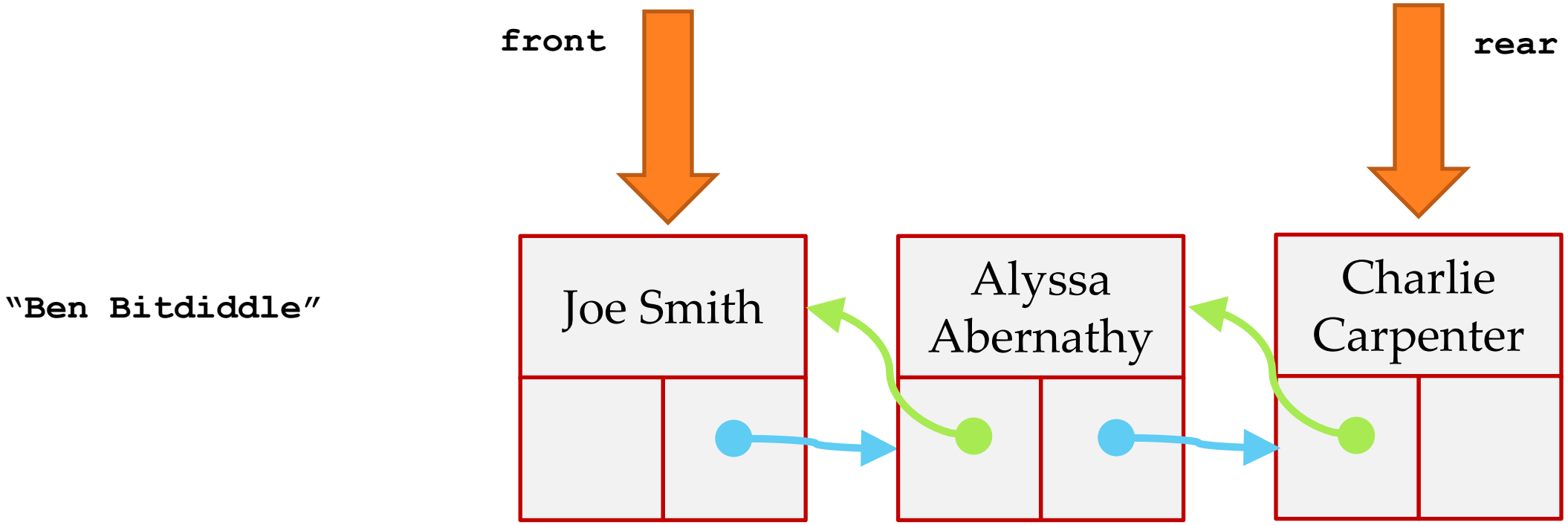
# Dequeue: Removing an element




**dequeue():**

```
remember the node
point front to front->nextNode
point the new front->prevNode to NULL
remember the name          "Ben Bitdiddle"
free the node
return the name
```

# Dequeue: Removing an element




**dequeue():**

```
remember the node
point front to front->nextNode
point the new front->prevNode to NULL
remember the name
free the node
return the name
```

**Now…**

**Do we have the same problems as using an array?**

- **No wrapping around: A linked list can grow [infinitely] in either direction!**

# Queue Operations

- **`enqueue()`**: Put an element in the queue
- **`dequeue()`**: Get the next element out of the queue
- **`front()`**: Look at the element at the front
- **`back()`**: Look at the element at the back/last inserted
- **`create()`**: Create the queue
- **`destroy()`**: Destroy the queue

# Queue Operations in C

- `char* dequeue(queueOfCustomers*);`
- `void enqueue(char* customer, queueOfCustomers*);`
- `char* front(queueOfCustomers*);`
- `char* back(queueOfCustomers*);`
- `queueOfCustomers* create();`
- `void destroy(queueOfCustomers*);`

```
struct queueOfCustomers{
    CustomerNode *front;
    CustomerNode *rear;
    int size;
}
```

# Implementing Queue Operations in C: create()

- `queueOfCustomers* create();`

```
queueOfCustomers* create(){
        // Allocate space on the heap
        // Initialize anything
        // Return pointer to new queue
}
```

```
struct queueOfCustomers{
    CustomerNode *front;
    CustomerNode *rear;
    int size;
}
```

# Implementing Queue Operations in C: destroy()

- `void destroy(queueOfCustomers*);`

```
void destroy(queueOfCustomers* customerQueue){
      // Free anything that hasn't been freed

      // Free the queue
      free(customerQueue);
}
```

```
struct queueOfCustomers{
   CustomerNode *front;
   CustomerNode *rear;
   int size;
}
```

# Implementing Queue Operations in C: enqueue()

- `queueOfCustomers* enqueue(char*, queueOfCustomers*);`

```
void enqueue(char* newCustomer, queueOfCustomers* queue){
        // Update the head index

        // Put the element in the array

        // Return queue (by convention
}
```

```
struct queueOfCustomers{
    CustomerNode *front;
    CustomerNode *rear;
    int size;
}
```

# Implementing Queue Operations in C: dequeue()

- `char* dequeue(queueOfCustomers*);`

```
char* dequeue(queueOfCustomers* queue){

}
```

```
struct queueOfCustomers{
   CustomerNode *front;
   CustomerNode *rear;
   int size;
}
```

# Implementing Queue Operations in C: front()

- `char* front(queueOfCustomers*);`

```
char* front(queueOfCustomers* queue){

}
```

```
struct queueOfCustomers{
   CustomerNode *front;
   CustomerNode *rear;
   int size;
}
```

# Implementing Queue Operations in C: rear()

- `char* rear(queueOfCustomers*);`

```
char* rear(queueOfCustomers* queue){

}
```

```
struct queueOfCustomers{
   CustomerNode *front;
   CustomerNode *rear;
   int size;
}
```

# Let's play with code…

- https://github.ccs.neu.edu/CS5002SEAF17/resources/blob/master/lect…

- (CS5002 resources repo on Github; lect10)

# What about implementing a Queue with a Stack?

# It's possible: think about it, and we'll look at it next time.

# Now, what if our Queue held patients instead of customers?

# Last time…

- Stacks

# Linked Lists

# Summary

**Stack**

- Primary Operations
  - Push
  - Pop
  - Peek
- Helper Operations
  - Create
  - Destroy
  - IsEmpty
  - IsFull

**Linked List**

- Primary Operations
  - Add
  - Remove
- Helper Operations
  - Create
  - Destroy
  - IsEmpty
  - Contains(Node), Contains(Data)
  - InsertBefore(), InsertAfter()
  - AddAtHead(), AddAtTail()

# Summary

**Stack**

- The abstraction is more important than the implementation
  - We implemented as an array; could also use a list.

**Linked List: Alternative to Array**

- Could be single or double linked
  - We talked about single
  - We implemented double
  - Code for single is in Github
- The List is really just a pointer to a single node
  - We implemented a wrapper struct that helped us keep track of the head
  - We can add metadata to that struct to help
    - Such as NumItems, ptr to LastNode.

# Arrays Versus LinkedList

**Array**

- Contiguous Memory
- You have to know the size before you create it
  - If you don't use all the memory allocated, you're wasting it
  - If you don't allocate enough, oops.

**LinkedList**

- Nodes can be non-contiguous
- You can keep growing the list, taking as much memory as you need but not more than you need

# Summary

- Queue
  - LIFO data structure
- Queue Operations:
  - enqueue() or insert(): puts element at back of list
  - dequeue() or remove(): returns element at front of list
  - front() or rear(): looks at the front/rear element
  - create()/destroy()

# Comparing Data Structures

| | Stack | Linked List | Queue | Tree |
|---|---|---|---|---|
| **Type** | Linear | Linear | Linear | Tree/graph |
| **Insert action** | Puts element at top | Inserts element where specified | Puts element at back | … |
| **Insert operation** | push() | insert() | enqueue() | … |
| **Remove action** | Returns element at top | Removes element as specified | Returns element at front | … |
| **Remove operation** | pop() | remove() | dequeue() | … |
| **Notes** | LIFO | Arbitrary length (not constrained by array size) | FIFO | … |
| | | | | |

# Runtime, Memory

| | Stack (Array) | Linked List | Queue (Llist) |
|---|---|---|---|
| **Memory** | O(k >= n)<br>*[k = max size]* | O(n) | O(n) |
| **Insert** | O(1) | O(1) *[insert beginning]*<br>O(n) *[insert elsewhere]* | O(1) |
| **Remove** | O(1) | O(1) | O(1) |
| **Find/ Contains** | N/A | O(n) | N/A |

# Next time:

- Sequences and Summations
- Trees
- More performance & efficiency