Arrays and Linked Lists

Download Demo Code

Goals

- Describe what an “abstract data type” means
- Compare different types of arrays
- Define singly and doubly linked lists
- Compare performance characteristics of arrays and lists
- Implement linked lists in JavaScript

Lists

A *list* is an *abstract data type*

It describes a *set of requirements*, not an exact implementation.

- Keep multiple items
- Can insert or delete items at any position
- Can contain duplicates
- Preserves order of items

Arrays

Arrangement of items at equally-spaced addresses in memory

```
[3, 7, 2, 4, 1, 2]
```

In memory:

3	7	2	4	1	2				

Therefore, inserting or deleting an item requires moving everything after it.

Array Runtimes

- Retrieving by index
  - ***O(1)***
- Finding
  - ***O(n)***
- General insertion
  - ***O(n)***
- General deletion
  - ***O(n)***

Direct Arrays / Vectors

This kind of array is often called a *direct array* or *vector*

Direct arrays only work if items are same size:

- all numbers
- all same-length strings

Don’t work well when items are varied sizes:

- different length strings
- subarrays or objects

They’re not commonly used, but JavaScript provides these as Typed Arrays

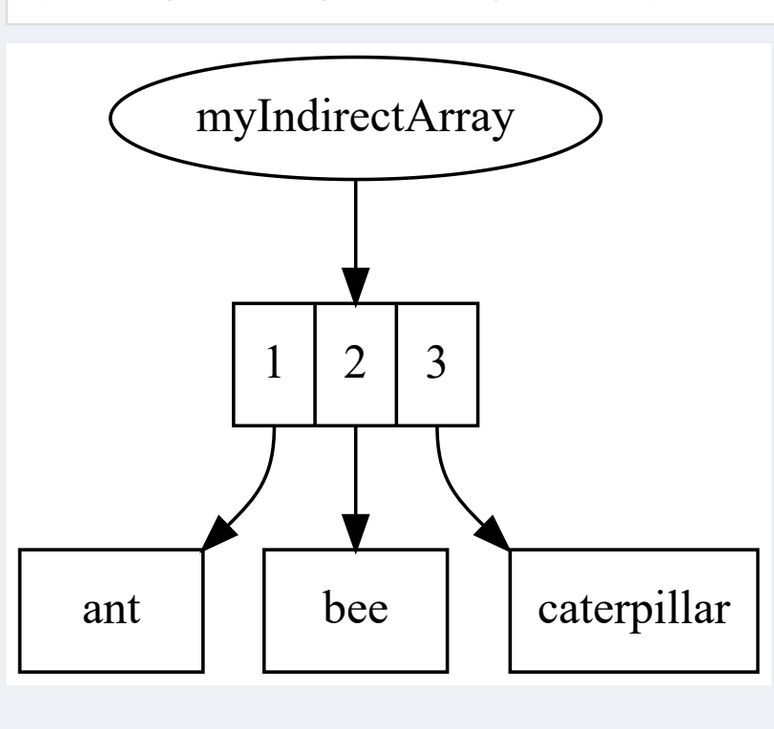
Indirect Arrays

In any *indirect array*, the array doesn’t directly hold the value.

It holds the memory address of the real value.

This lets an array store different types of data, or different length data.

```
["ant", "bee", "caterpillar"]
```

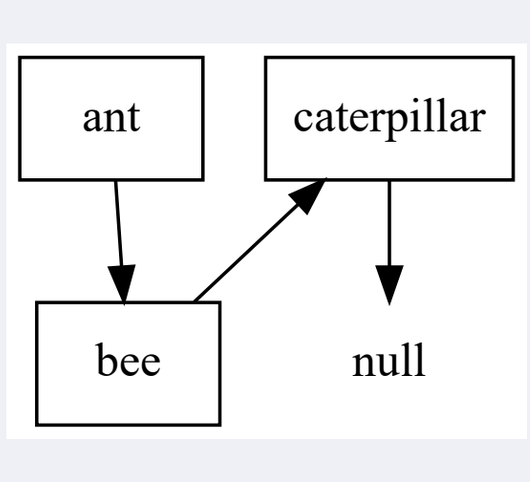


What Does JavaScript Use?

Indirect arrays — since you can store different-length things in them

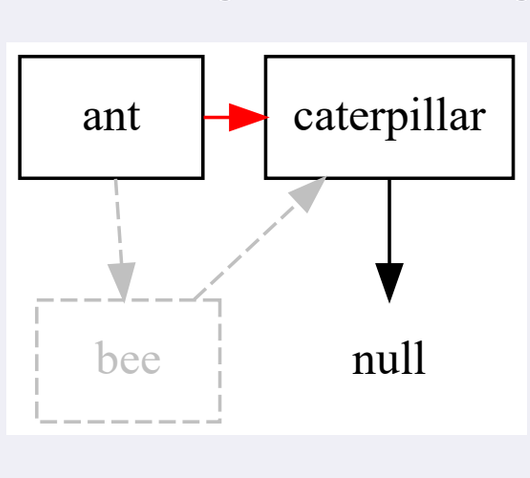
It’s complicated, though: some implementations have specialized or adaptive structures to handle edge cases like sparse arrays

Linked Lists



Items aren’t stored in contiguous memory; instead, each item references the next item in the sequence.

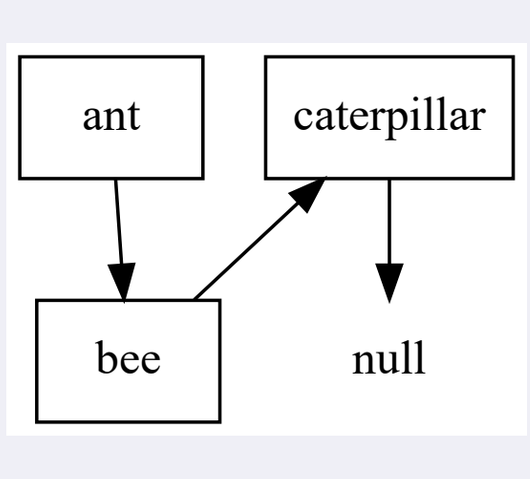
Can rearrange without having to move other items in memory.



This is a lot faster than having to move everything around in a big list.

A Node

The basic unit of a linked list is a node.

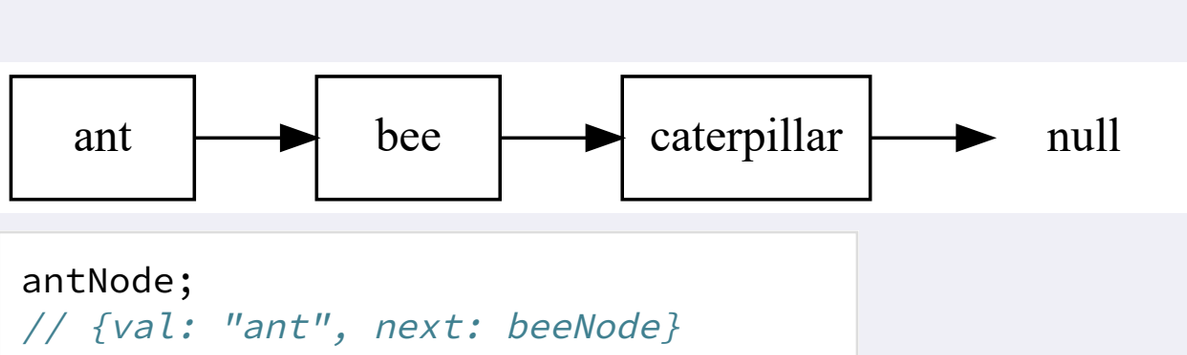


ant, ***bee***, and ***caterpillar*** are nodes.

A basic Node has two attributes:

val
: the information the node contains (could be string, int, instance, etc)

next
: reference to next node (for last item, this is ***null***)



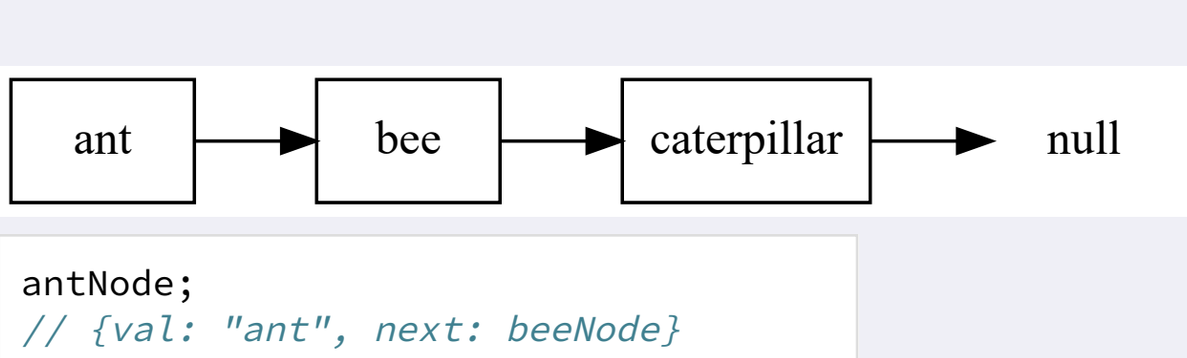

```
antNode;
// {val: "ant", next: beeNode}

beeNode;
// {val: "bee", next: caterpillarNode}

caterpillarNode;
// {val: "caterpillar", next: null}
```

The Node Class

demo/linkedlist.js

```
/** Node class for item in linked list. */

class Node {
  constructor(val) {
    this.val = val;
    this.next = null;
  }
}
```

```
let antNode = new Node("ant");
let beeNode = new Node("bee");
let caterpillarNode = new Node("caterpillar");

antNode.next = beeNode;
beeNode.next = caterpillarNode;
```




```
antNode;
// {val: "ant", next: beeNode}

beeNode;
// {val: "bee", next: caterpillarNode}

caterpillarNode;
// {val: "caterpillar", next: null}
```

Smarter Node Class

Some people make a ***Node*** class which accepts optional ***next*** argument:

```
class Node {
  constructor(val, next=null) {
    this.val = val;
    this.next = next;
  }
}
```

Then you can add a chain of nodes:

```
let antNode = new Node("ant",
                new Node("bee",
                  new Node("caterpillar")));
```

This ends up exactly the same, but can be harder to read at first.

LinkedList Class

A Linked List is just a bunch of nodes linked sequentially.

The only attribute it must have is a reference to its first node, called the ***head***.

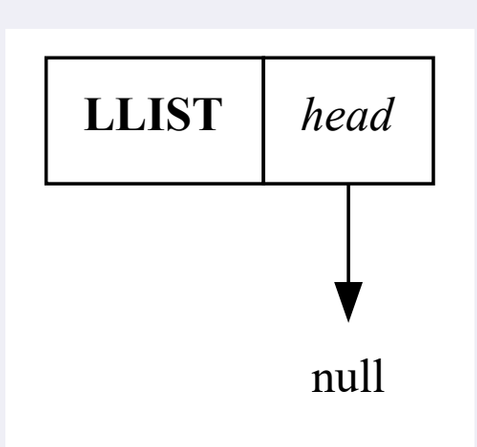
Since the list starts empty, the head is initially ***null***.

```
class LinkedList {
  constructor() {
    this.head = null;
  }
}
```

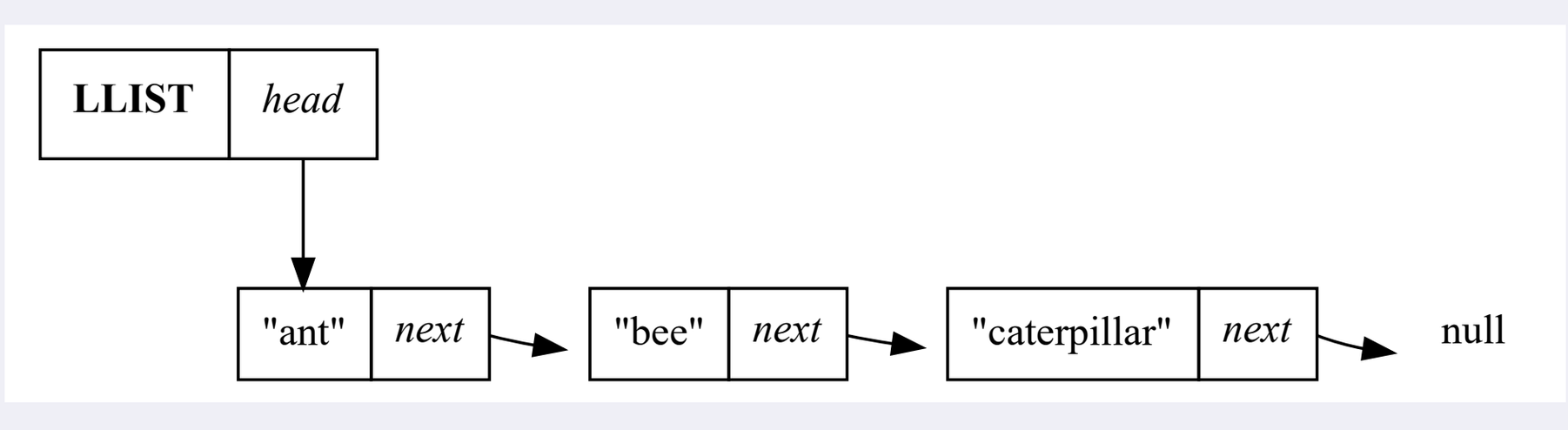
```
let insects = new LinkedList();
```

In Pictures…

An empty Linked List:



A Linked List with nodes in it:



Things you might want to do

- Print each node
- Find a node by its data
- Append to end
- Insert at specific position
- Remove a node

Traversing

Assumption: we’ve already built list, leaving the actual construction for later.

We’re just going to traverse the list and print it.

demo/linkedlist.js

```
  /** print(): traverse & console.log each item. */

  print() {
    let current = this.head;

    while (current !== null) {
      console.log(current.val);
      current = current.next;
    }
  }
```

Searching

Like printing—but stop searching once we find what we’re looking for.

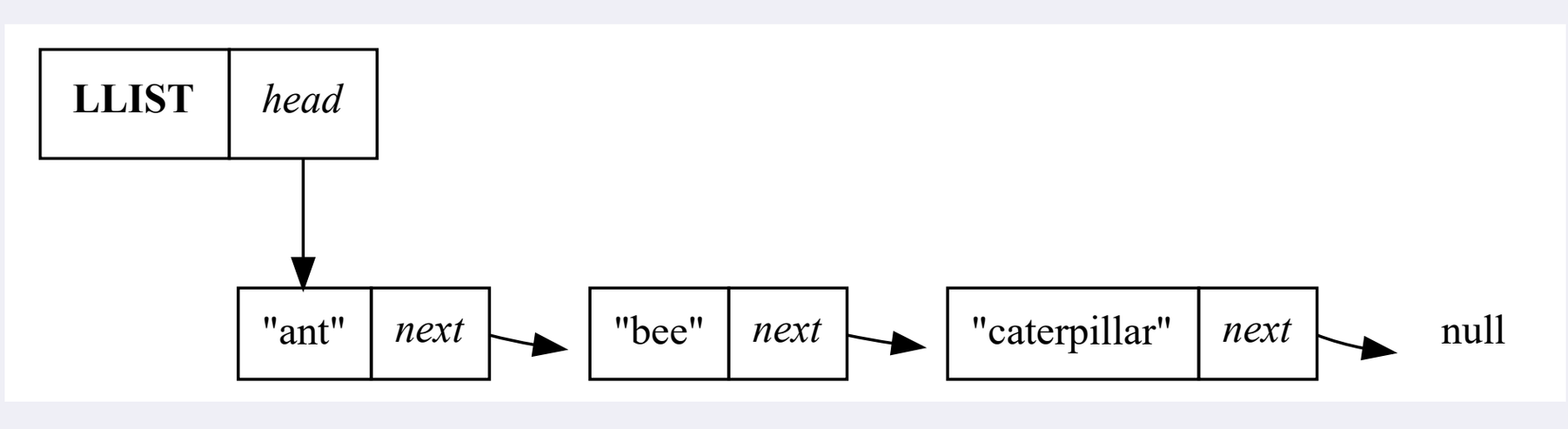
demo/linkedlist.js

```
  /** find(val): is val in list? */

  find(val) {
    let current = this.head;

    while (current !== null) {
      if (current.val === val) return true;

      current = current.next;
    }

    return false;
  }
```

Appending/Removing Nodes

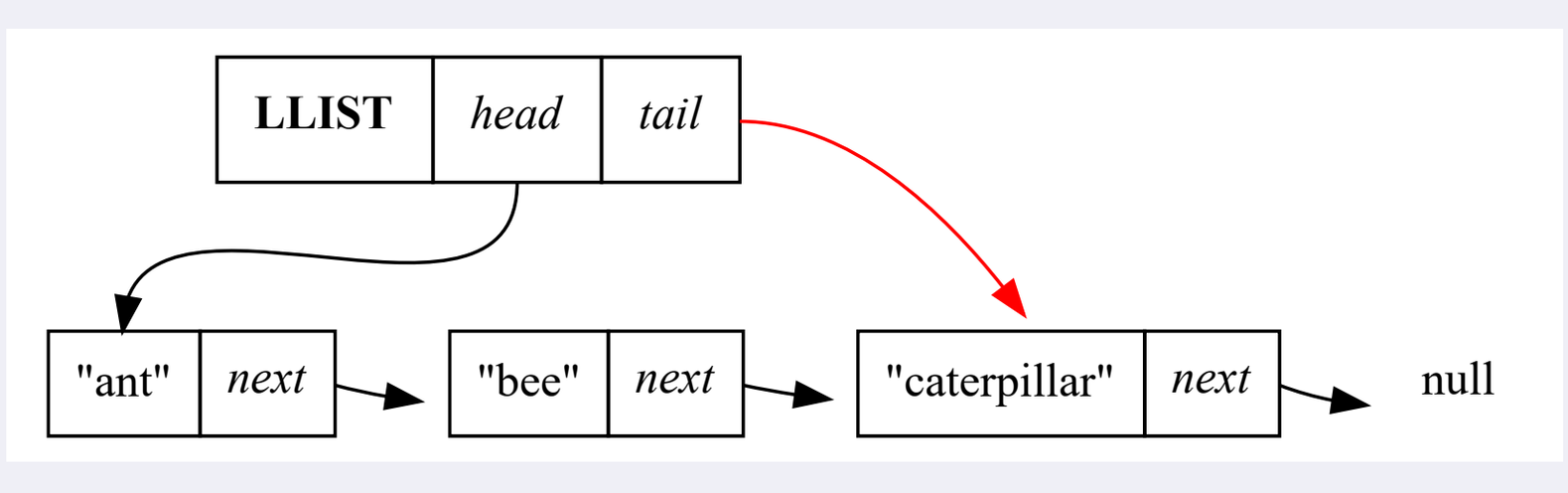
Append a Node

Q: How do we append a node to the end of a linked list?



A: Walk to the end and add it there.

(But wouldn’t it be faster to append if we “know” the end?)

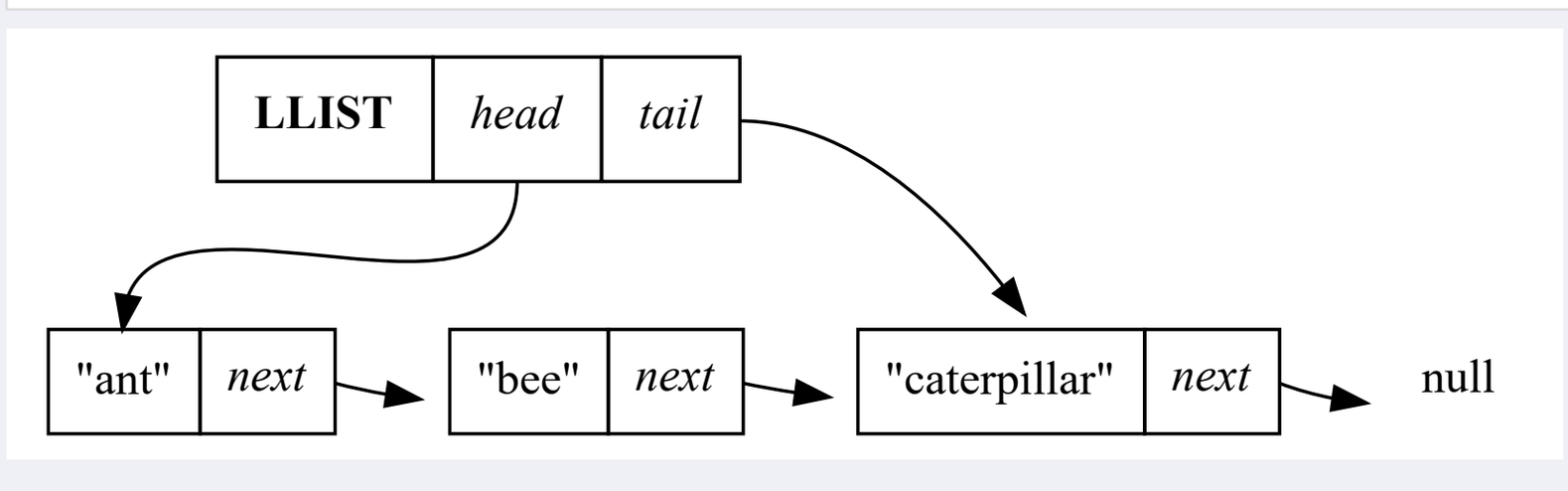


This way, appending is always ***O(1)***

This becomes easier if we add a ***tail*** attribute onto our list. This way, we don’t have to traverse the list every time we add a node.

We can do this with just ***head***, but why if we can add a ***tail***?

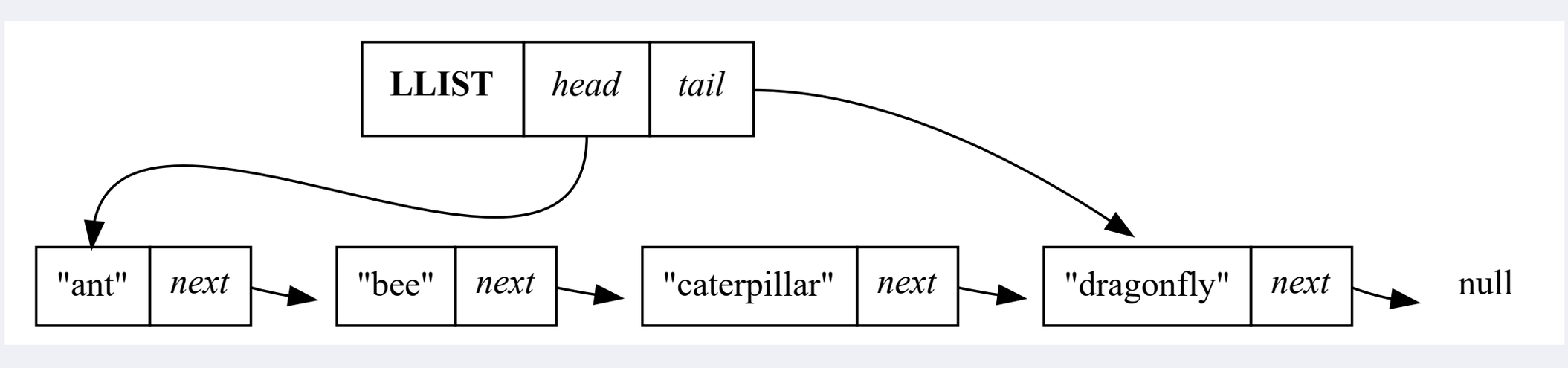
```
class LinkedList {
 constructor() {
   this.head = null;
   this.tail = null;
 }
```



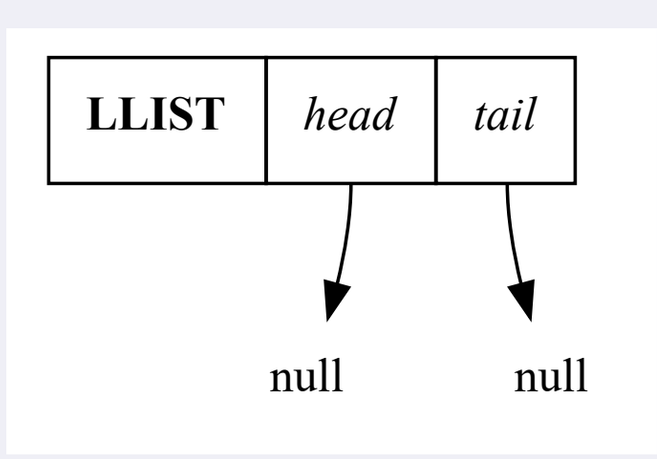
What do we need to do to add “dragonfly”?

- make new node ***dragonfly***
- make ***caterpillar.next*** a reference to ***dragonfly***
- make ***list.tail*** a reference to ***dragonfly***

Success!



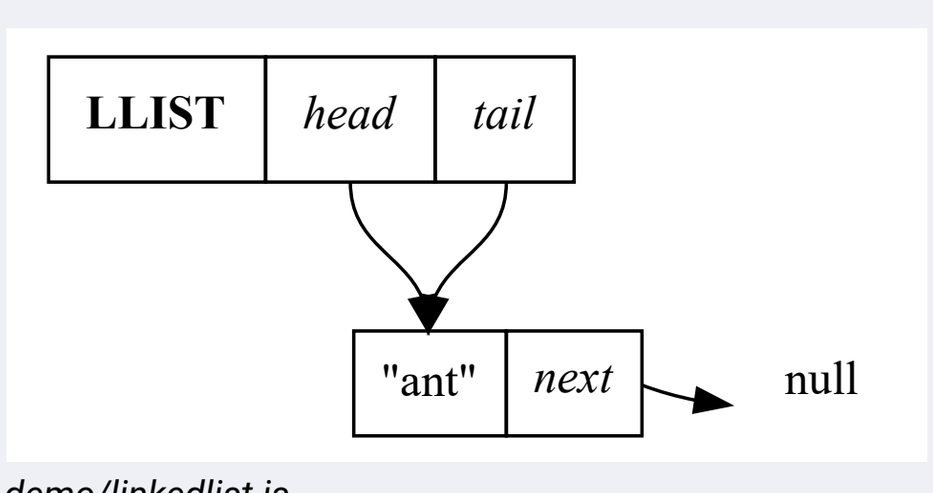
Don’t forget to handle case of an empty list!



What do we need to do to add “ant”?

- make new node ***ant***
- make ***list.head*** a reference to ***ant***
- make ***list.tail*** a reference to ***ant***

Success!



demo/linkedlist.js

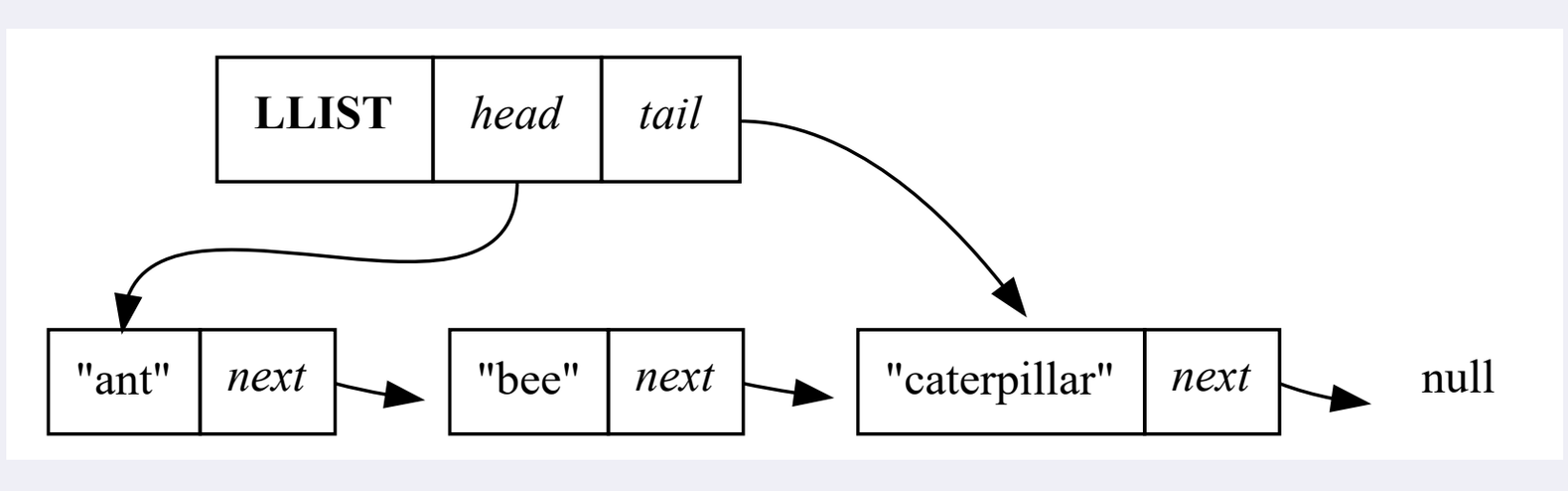
```
  /** push(val): add node w/val to end of list. */

  push(val) {
    let newNode = new Node(val);

    if (this.head === null) this.head = newNode;

    if (this.tail !== null) this.tail.next = newNode;

    this.tail = newNode;
  }
```

```
let insects = new LinkedList();

insects.push("ant");
insects.push("bee");
insects.push("caterpillar");
```

Remove a Node (by value)

What would you need to change to remove:

- “ant”
- “bee”
- “caterpillar”



All we are doing to “remove” a node from the list is redirecting the reference (or ***next***) of a node to the one after the node we’re looking for.

There are many tricky ways of doing this.

We’re going to rely on a “daisy-chaining” effect and the fact that any given node’s ***next*** is just a node, which has its own ***val*** and ***next***.

The code is a bit complex, since we need to handle:

- removing only item in linked list
  - Don’t forget to update head *and* tail to ***null***
- removing first item
  - Don’t forget to update the head!
- removing the last item
  - Don’t forget to update the tail!
- removing an item in the middle

Runtime of Linked Lists

- Going to “next” item
  - ***O(1)***
- Going to item by arbitrary index
  - ***O(n)***
- Searching for value
  - ***O(n)***
- General insertion or deletion
  - ***O(n)***
- Adding to start
  - ***O(1)***
- Appending to end
  - ***O(1)*** if know tail; ***O(n)*** if don’t
- Deleting at start
  - ***O(1)***

How do these compare to arrays?

Code Implementation

Can write with classic OO:

```
class Node { /* ... */ }

class LinkedList {
  constructor() {
    this.head = null;
    this.tail = null;
  }

  find(val) { /* ... */ }
}


let antNode = new Node("ant");
let insects = new LinkedList();

insects.find("ant");
```

Can write using plain JS objects:

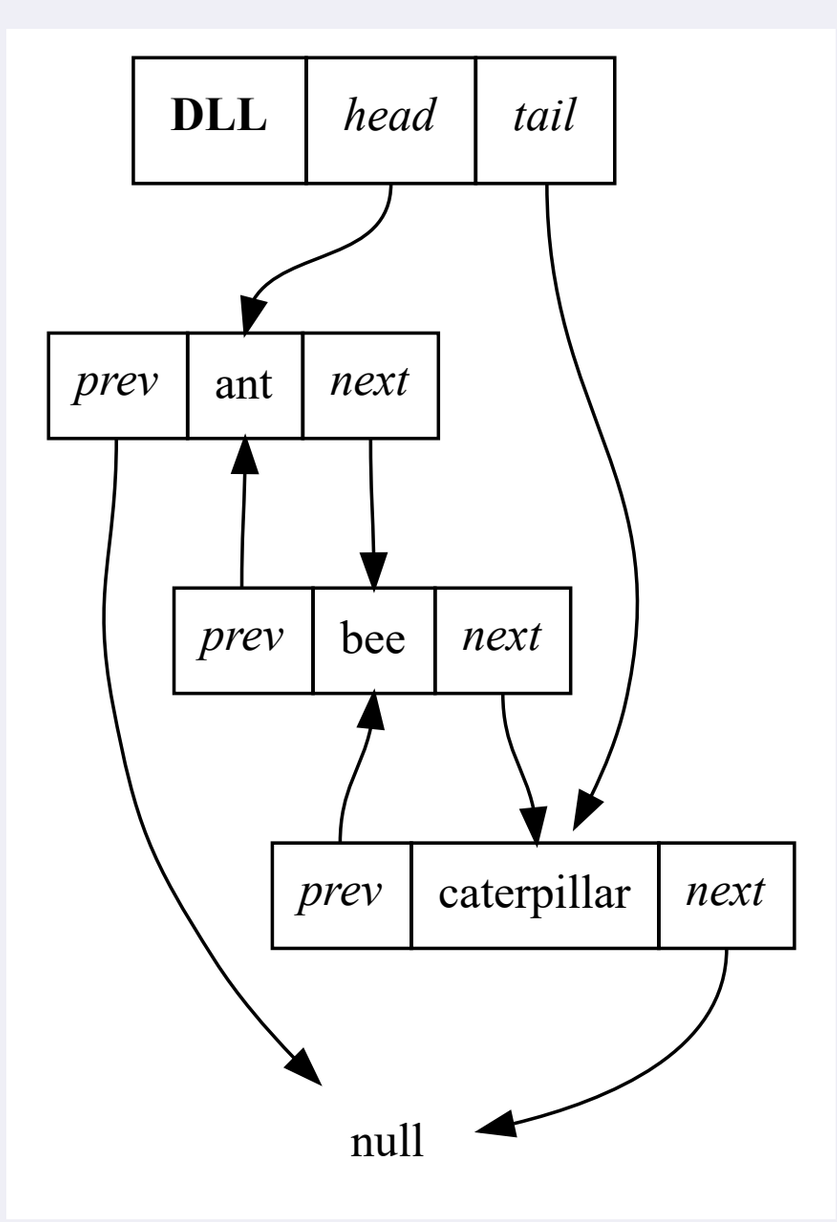
```
function find(insects, val) { /* ... */ }


antNode = {val: "ant", next: null};
insects = {head: antNode, tail: antNode};

find(insects, "ant");
```

Note: Other Possibilities, too!

Less commonly, you may see implementations that use arrays or tuples to hold nodes, such that the linked list is series of nested arrays or tuples. These tend to be more common in languages without OO, and tend to be more complex to visualize or understand.

Doubly-Linked Lists

Sometimes, linked lists have ***next*** and a ***prev*** (the “previous node”)



n.b. While doubly-linked lists are relatively common and useful in actual programming, most interview questions are asking about a singly-linked list.

Resources

What’s a Linked List, Anyway? [Base CS]