

LEC 10

**CSE 123**

# Recursive LinkedIntList

Questions during Class?  
Raise hand or send here

sli.do #cse123



BEFORE WE START

*Talk to your neighbors:*


*What'*

Music: [123 24su Lecture Tunes](#) 

**Instructor:** Joe Spaniac

**TAs:** Andras Eric Sahej Zach  
Daniel Nicole Trien


# Lecture Outline

- **Announcements/Reminders** 
- Recursive Definitions
  - Files
  - LinkedLists
- Recursive Traversals
- LinkedList Modifications
  - Iterative
  - Recursive

# Announcements

- R4 feedback releases sometime after lecture today
- P2 due tonight (7/24) at 11:59pm
  - Submit *something* so we can provide some feedback!
- Creative Project 3 releases tomorrow (7/25)
  - Back to one week turnaround
- Check-in 3 in section tomorrow (7/25)
  - Very, *very* similar problem to what you might see on a quiz
  - Guaranteed to get feedback before the quiz on Tuesday if you attend
- Quiz 2 this upcoming Tuesday (7/30)
  - Topics: Runtime; Recursion
    - Note: Separate topics, we'll never ask you to determine the runtime of a recursive algorithm

# Lecture Outline

- Announcements/Reminders
- **Recursive Definitions** 
  - Files
  - LinkedLists
- Recursive Traversals
- LinkedList Modifications
  - Iterative
  - Recursive

# Files

- We'll say that computer files fall into one of the following categories:



Standard file (.txt, .csv, .java)

```
f.isDirectory() -> false
```



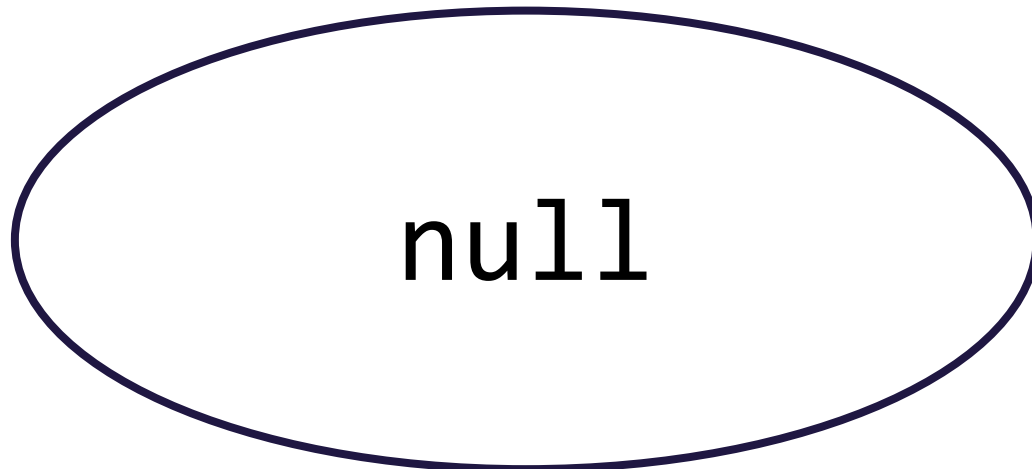
Directory w/ subfiles

```
f.isDirectory() -> true  
File[] subFiles = f.listFiles()
```

*This is a recursive definition! A File is either normal, or a directory with a File[] of subFiles*

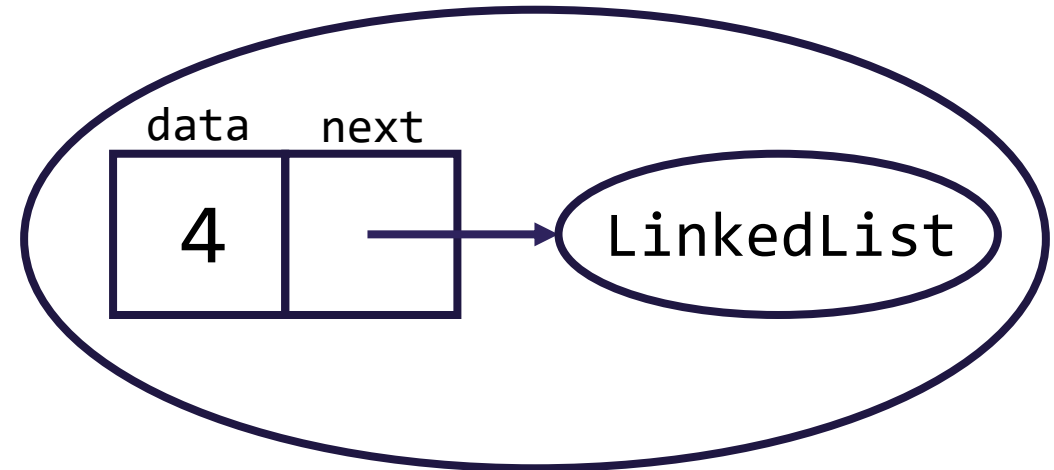
# LinkedLists

- We'll say that any LinkedList falls into one of the following categories:



Empty list

`front == null`




Node w/ another LinkedList

`front != null`  
`front.next = LinkedList`

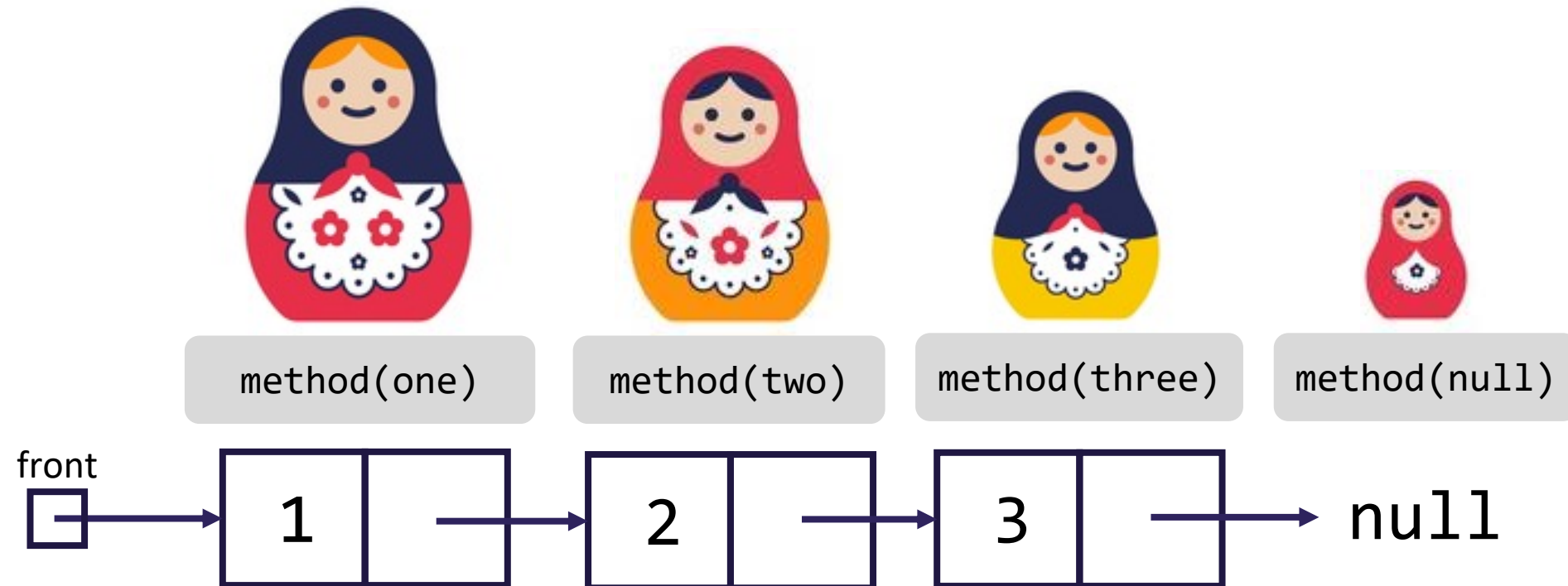
*This is a recursive definition! A sublist is either empty or a node with another sublist!*

# Lecture Outline

- Announcements/Reminders
- Recursive Definitions
  - Files
  - LinkedLists
- **Recursive Traversals** 
- LinkedList Modifications
  - Iterative
  - Recursive


# Recursive Traversals w/ LinkedLists

- Guaranteed base case: empty list
  - Simplest possible input, should immediately know the return
- Guaranteed public / private pair
  - Need to know which sublist you're currently processing (i.e. curr)





# Lecture Outline

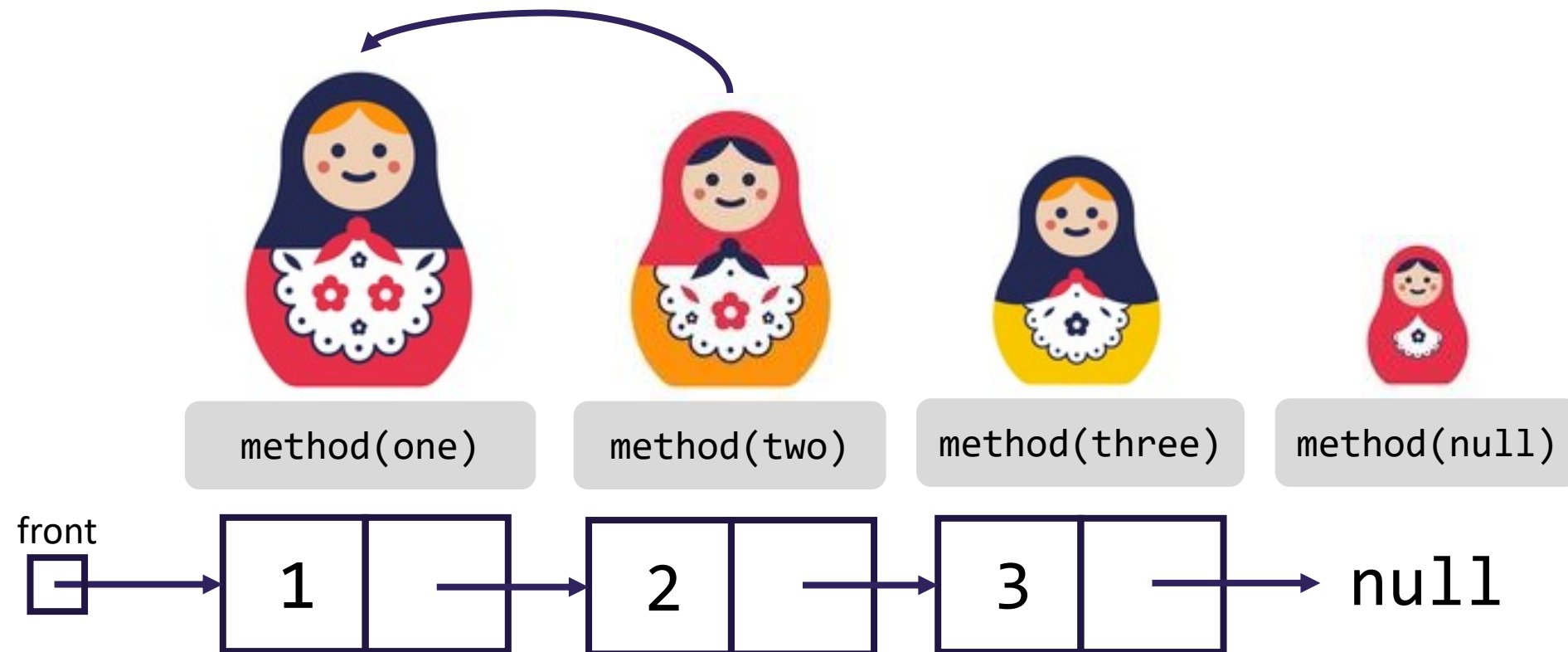
- Announcements/Reminders
- Recursive Definitions
  - Files
  - LinkedLists
- Recursive Traversals
- **LinkedList Modifications** 
  - Iterative
  - Recursive

# Modifying LinkedLists [Review]

- Remember: using a `curr` variable to iterate over nodes
- Does changing `curr` actually update our chain?
  - What will? Changing `curr.next`, changing `front`
  - Need to **stop one early** to make changes
- Often a number of cases to watch out for:
  - M(iddle) – Modifying node in the middle of the list (general)
  - F(ront) – Modifying the first node
  - E(mpty) – What if the list is empty?
  - E(nd) – Rare, do we need to do something with the end of the list?

# Modifying LinkedLists Recursively

- Much easier than iterative solutions!
- No longer need to stop one early
  - Can go right to the point you'd like to make the change



# Modifying LinkedLists Recursively

- Much easier than iterative solutions!
- No longer need to stop one early
  - Can go right to the point you'd like to make the change
- How? Return the updated change and catch it!
  - Private pair returns `ListNode` type
  - `curr.next = change(curr.next) / front = change(front)`
  - Resulting solutions much cleaner than iterative cases
- We call this pattern **`x = change(x)`**