6.1

Stacks and Queues

Icebreaker

Lesson Plan

- [10] Icebreaker
- [10] LinkedList Recursion
- [20] Stacks and Queues

- [35] Practice: LinkedList Recursion
- [20] Practice: Stacks

Recursion

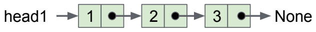
Recursion

LinkedLists are recursive data structures. The entire list is defined by just the head!

Every node is the head of its own list!

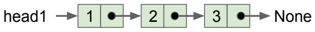
Recursion

Find the length of this list recursively

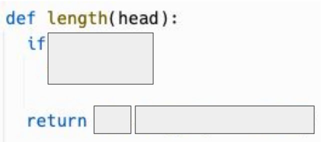


Recursion

Find the length of this list recursively



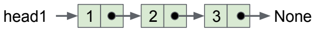
What's our base case? What's the simplest possible list?



```
def length(head):
  if □

  return □ □
```

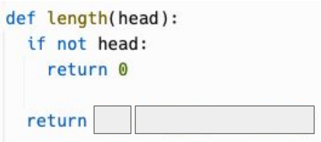
Recursion

Find the length of this list recursively



What's our base case? What's the simplest possible list?

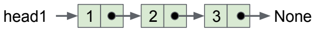
None



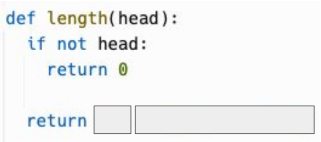
```
def length(head):
  if not head:
    return 0

  return □ □
```

Recursion

Find the length of this list recursively



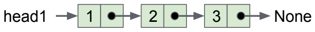
What's a list that's just a little bit smaller than the current list?



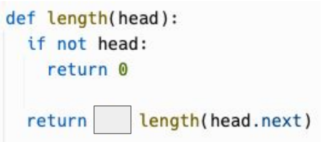
```
def length(head):
  if not head:
    return 0

  return □ □
```

Recursion

Find the length of this list recursively



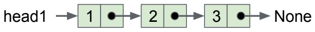
What's a list that's just a little bit smaller than the current list?



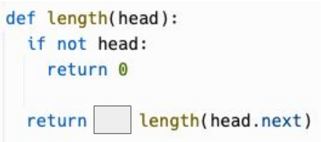
```
def length(head):
  if not head:
    return 0

  return □ length(head.next)
```

Recursion

Find the length of this list recursively



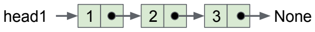
What's the relationship between the size of head and the size of head.next?



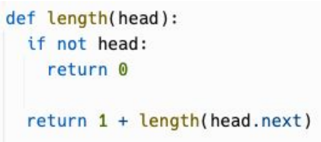
```
def length(head):
  if not head:
    return 0

  return □ length(head.next)
```

Recursion

Find the length of this list recursively



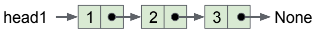
What's the relationship between the size of head and the size of head.next?



```
def length(head):
  if not head:
    return 0

  return 1 + length(head.next)
```

Recursion

How could we append to the end of this list recursively?

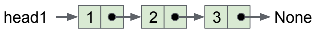


Recursion

How could we append to the end of this list recursively?

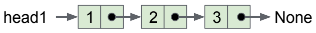
Append the node onto the end of head.next!

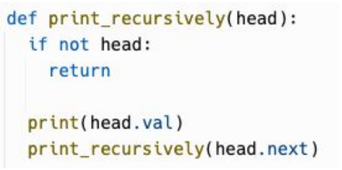
(You'll get to try this today)



Recursion

Recursion also lets us access the elements of a LinkedList in reverse order…

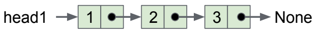


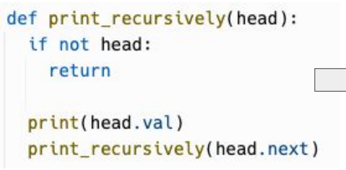


```
def print_recursively(head):
  if not head:
    return

  print(head.val)
  print_recursively(head.next)
```

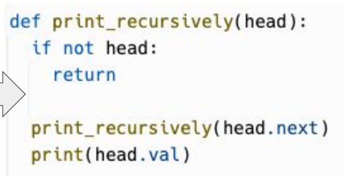
Recursion

Recursion also lets us access the elements of a LinkedList in reverse order…





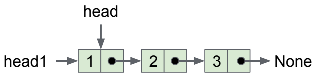
```
def print_recursively(head):
  if not head:
    return

  print(head.val)
  print_recursively(head.next)
```

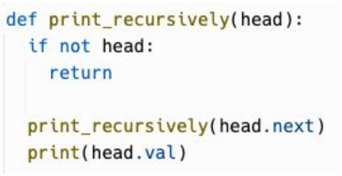


```
def print_recursively(head):
  if not head:
    return

  print_recursively(head.next)
  print(head.val)
```

Recursion

Recursion also lets us access the elements of a LinkedList in reverse order…

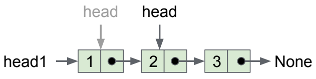


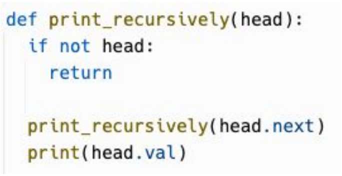


```
def print_recursively(head):
  if not head:
    return

  print_recursively(head.next)
  print(head.val)
```

Recursion

Recursion also lets us access the elements of a LinkedList in reverse order…

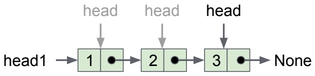


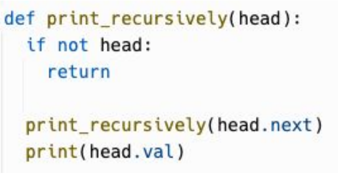


```
def print_recursively(head):
  if not head:
    return

  print_recursively(head.next)
  print(head.val)
```

Recursion

Recursion also lets us access the elements of a LinkedList in reverse order…

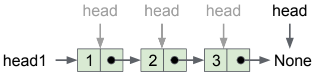


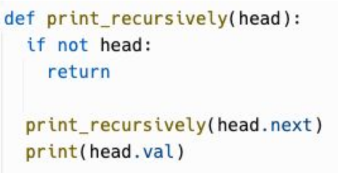


```
def print_recursively(head):
  if not head:
    return

  print_recursively(head.next)
  print(head.val)
```

Recursion

Recursion also lets us access the elements of a LinkedList in reverse order…



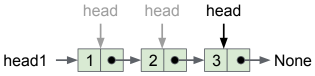


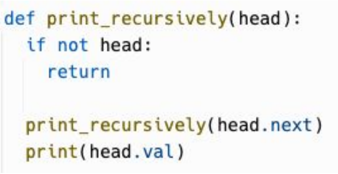
```
def print_recursively(head):
  if not head:
    return

  print_recursively(head.next)
  print(head.val)
```

Recursion

Recursion also lets us access the elements of a LinkedList in reverse order…

Prints 3!



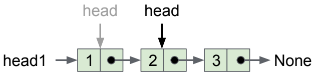


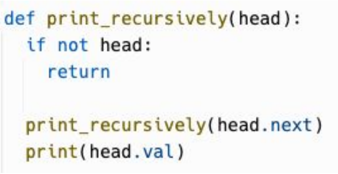
```
def print_recursively(head):
  if not head:
    return

  print_recursively(head.next)
  print(head.val)
```

Recursion

Recursion also lets us access the elements of a LinkedList in reverse order…

Prints 3!
Prints 2!



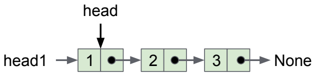


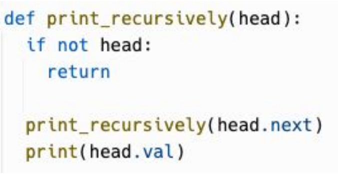
```
def print_recursively(head):
  if not head:
    return

  print_recursively(head.next)
  print(head.val)
```

Recursion

Recursion also lets us access the elements of a LinkedList in reverse order…

Prints 3!
Prints 2!
Prints 1!





```
def print_recursively(head):
  if not head:
    return

  print_recursively(head.next)
  print(head.val)
```

Practice Problems - LinkedList Recursion [repl.it]

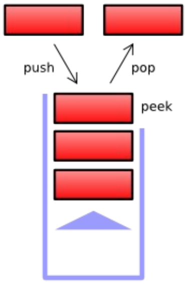
Stacks & Queues

Stack ADT

Supported Operations:

- Push
- Pop
- Peek
- Empty?

LIFO = **L**ast **I**n **F**irst **O**ut

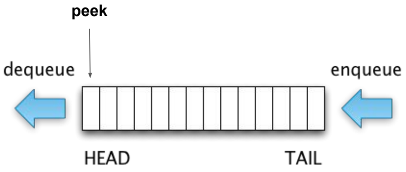


Queue ADT

Supported Operations:

- Enqueue
- Dequeue
- Peek
- Empty?

FIFO = **F**irst **I**n **F**irst **O**ut



Stack (LIFO) or Queue (FIFO)?

1. Ticket line at the movie theater **Q**
2. Putting on several bracelets and taking them off **S**
3. Interrupting your story with a brief tangent and then resuming it **S**
4. Waitlist for enrolling in a course **Q**
5. People riding an escalator **Q**
6. Wrapping a gift in many layers of wrapping paper **S**
7. Going down a bad path in a maze and retracing your steps **S**
8. People going down a waterslide **Q**

Stack & Queue Applications

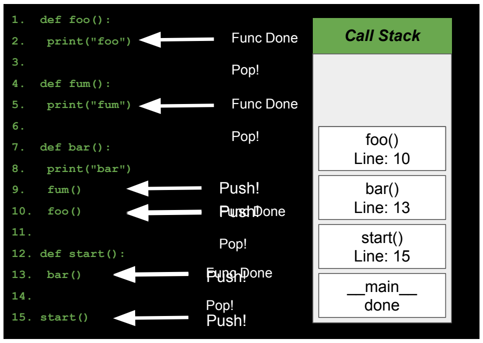
Why are Stacks useful?

- Browser History
- Function Call stack
- “Undo” in Text Editors
- Parsing Computer Programs
- Backtracking & Depth-first search

Browser History

Review: Function Call Stack

- Stack Frames are stored in a program's **Call Stack**
- Calling a function **pushes** a Stack Frame onto the Call Stack
- Returning from a function **pops** that Stack Frame off of the Call Stack



```
1.  def foo():
2.   print("foo")
3.
4.  def fum():
5.   print("fum")
6.
7.  def bar():
8.   print("bar")
9.   fum()
10.  foo()
11.
12. def start():
13.  bar()
14.
15. start()
```

Why are Queues useful?

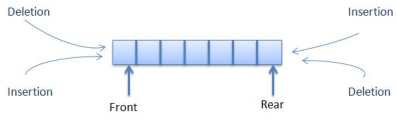
- Event processing
- Buffering
- Breadth-first search

Deque ADT

Deque ADT

Supported Operations:

- Back
  - Add
  - Remove
  - Peek
- Front
  - Add
  - Remove
  - Peek
- IsEmpty

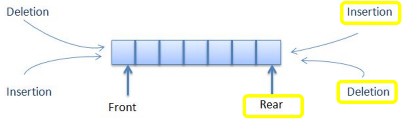


Deque is pronounced like "deck"

Using a Deque as a Stack

Supported Operations:

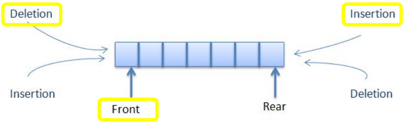
- Back
  - Add
  - Remove
  - Peek
- Front
  - Add
  - Remove
  - Peek
- IsEmpty



Using a Deque as a Queue

Supported Operations:

- Back
  - Add
  - Remove
  - Peek
- Front
  - Add
  - Remove
  - Peek
- IsEmpty



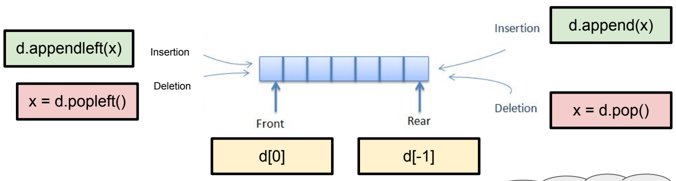
Python deque

[Python deque docs]

Python deque

```
from collections import deque

d = deque()
```

Same module as defaultdict

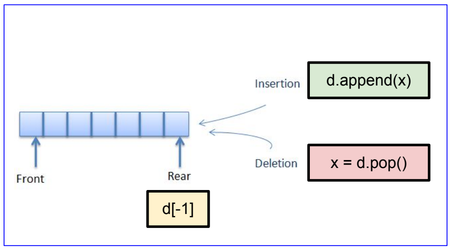


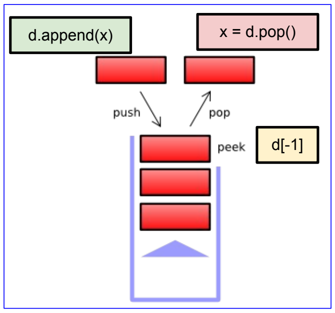
Python will let you access d[anything], but it isn't really done!

```
# If the deque is empty
if not d:
  # do something
```

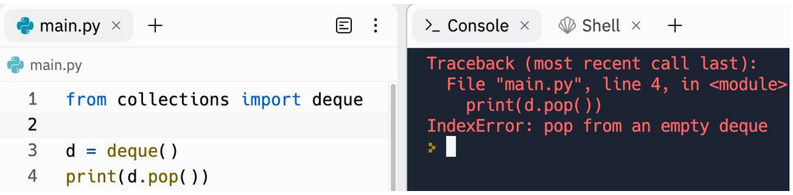
Same as checking if a list is empty

Python deque as a Stack





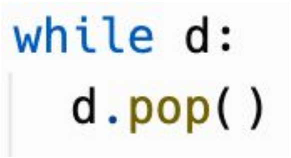
What happens if you try to remove from an empty deque?



```
from collections import deque

d = deque()
print(d.pop())
```

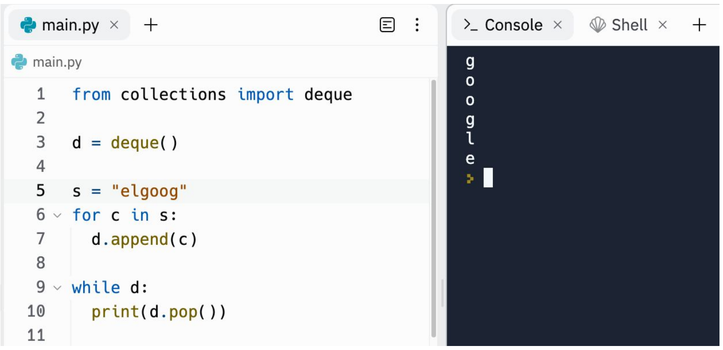
```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    print(d.pop())
IndexError: pop from an empty deque
```

Popping Everything Out of a Stack



```
while d:
    d.pop()
```

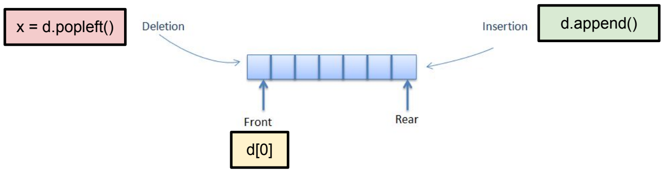
Using a Stack to Reverse Order



```python
from collections import deque

d = deque()

s = "elgoog"
for c in s:
    d.append(c)

while d:
    print(d.pop())
```

```
g
o
o
g
l
e
```

Python deque as a Queue



Function Return Types

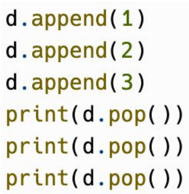
pop()/popleft() remove the element AND return it

append()/appendleft() add the element and return NONE

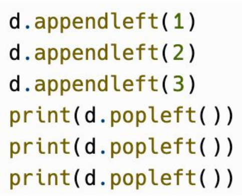
Deques are Symmetric!

You can use either the right or left as the top of the stack! You can use either the right or left as the front/back of the queue!

Just be consistent.



```python
d.append(1)
d.append(2)
d.append(3)
print(d.pop())
print(d.pop())
print(d.pop())
```

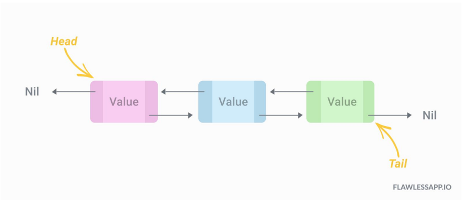


```python
d.appendleft(1)
d.appendleft(2)
d.appendleft(3)
print(d.popleft())
print(d.popleft())
print(d.popleft())
```

Runtime of deque operations

Python deques are implemented using a **Doubly-Linked List** with **Head and Tail**

- O(1) to insert, remove, or peek at the front or back
- O(n) to peek at an arbitrary element at a given index in the middle



Remember lists?

If you want fast add / remove to front, use a deque!

	Add to back	**Remove from back**	**Add to front**	**Remove from front**	**Peek at back**	**Peek at front**	**Peek at middle**
Python deque	?	?	?	?	?	?	?
Python list	?	?	?	?	?	?	?

If you want fast random access, use a list!

Practice using deque

deque docs

Practice Problems - Stacks [repl.it]

1. Write a function that takes a sequence of parentheses and returns True if they are balanced.
    a. Example: (()(()))() → True
    b. Example: (()() → False

Extension
Can you make your program handle other types of paired characters, like **[]{}<>** without duplicating a bunch of logic?