Announcements

● Midterm remarks due Friday @ 11PM
● A1 remarks due July 16th @ 11PM
  ○ Read the e-mail/Piazza post before submitting a remark!
● A2 will be released on Saturday
● Additional recursion practice problems
● Lab 6 and Exercise 6 are out
Review: Recursion

- get_values(x)
  - x: Anything (might be a list of nested lists)
  - Return all of the values in x
Review: Recursion

- get_values(x)
  - x: Anything (might be a list of nested lists)
  - Return all of the values in x

get_values([1, [[2], 'cat'], ['a', [['b', [3]], 4], 5]])
Review: Recursion

- get-values(x)
  - x: Anything (might be a list of nested lists)
  - Return all of the values in x

get-values([1, [[2], 'cat'], ['a', [['b', [3]], 4], 5]])

[1, 2, 'cat', 'a', 'b', 3, 4, 5]
Review: Recursion

- `get_values(x)`
  - `x`: Anything (might be a list of nested lists)
  - Return all of the values in `x`

`get_values(5)`
Review: Recursion

- get_values(x)
  - x: Anything (might be a list of nested lists)
  - Return all of the values in x

get_values(5)

  5
get_values

1. **Base case:** When don't we need recursion?
get_values

1. **Base case:** When don't we need recursion?

   When x is not a list, return [x]
get_values

if not isinstance(x, list):
    return [x]
get_values

1. **Base case:** When don't we need recursion?
   
   When x is not a list, return [x]

2. **Recursive step:** What recursive calls do we make and how do we use them?
get_values

[1, [[2], 'cat'], ['a', [['b', [3]], 4], 5]]
get_values

[1, [[2], 'cat'], ['a', [['b', [3]], 4], 5]]

get_values(1)

get_values([[2], 'cat'])

get_values(['a', [['b', [3]], 4], 5])
get_values

[1, [[2], 'cat'], ['a', [['b', [3]], 4], 5]]

[1]

[2, 'cat']

['a', 'b', 3, 4, 5]
get_values

Have:

[1]
[2, 'cat']
['a', 'b', 3, 4, 5]

Want: [1, 2, 'cat', 'a', 'b', 3, 4, 5]
get_values

1. **Base case:** When don't we need recursion?
   When x is not a list, return [x]

2. **Recursive step:** What recursive calls do we make and how do we use them?
   Add the results of the recursive calls to each item in x together.
get_values

if not isinstance(x, list):
    return [x]

return sum([get_values(item) for item in x], [])
get_values

if not isinstance(x, list):
    return [x]

to_return = []
for item in x:
    to_return += get_values(item)
return to_return
Review: Recursion (+ Linked List)

- Can use recursion to traverse a LinkedList.
- get_values_linked_list(lnk)
  - Ink is None or a LinkedListNode
  - Ignore front/back/size (since it's a node).
get_values_linked_list

Ink = 3 → 1 → 4 → 7 → 2

get_values_linked_list(Ink) == [3, 1, 4, 7, 2]
get_values_linked_list(None) == []
get_values_linked_list

1. Base case?

2. Recursive step?
   a. What recursive call(s) do we make?
   b. What do we expect back?
   c. How do we get the result we want to return?

(LinkedListNodes only have a .value and .next_!)
get_values_linked_list

1. **Base case?** lnk is None: return []

2. Recursive step?
   a. What recursive call(s) do we make?
   b. What do we expect back?
   c. How do we get the result we want to return?

(LinkedListNodes only have a .value and .next_)
get_values_linked_list

if lnk is None:
    return []
get\_values\_linked\_list

1. Base case? Ink is None: return []

2. Recursive step?
   a. What recursive call(s) do we make? 
      get\_values\_linked\_list(Ink.next)
   b. What do we expect back?
   c. How do we get the result we want to return?
get_values_linked_list

Ink = 3

Ink.next_ = 1

Diagram of linked list:

- ink = 3
- ink.next_ = 1

Nodes: 1 -> 4 -> 7 -> 2
get_values_linked_list

Ink = 3 → 1 → 4 → 7 → 2

Ink.next_ = 1 → 4 → 7 → 2

get_values_linked_list(Ink.next_) == [1, 4, 7, 2]
`get_values_linked_list`\[1, 4, 7, 2]\n
Add `Ink.value` to the front of the list!
def get_values_linked_list:
    if lnk is None:
        return []
    return [lnk.value] +
    get_values_linked_list(lnk.next_)

28
Multiple LinkedLists 'next' 

- What if we had a class with a value and children?
  - Children is a list of LinkedLists
- LinkedLists only point to 1 item after, but this points to more.
Multiple LinkedLists 'next'

1 -> 4 -> 3
2 -> 8
9 -> 7 -> 5 -> 6
Multiple LinkedLists 'next'
Multiple LinkedLists 'next'

```
self.value

0

self

1 -> 4 -> 3

2 -> 8

9 -> 7 -> 5 -> 6

self.children = [lnk1, lnk2, lnk3]
```
get_values

- Return a list of all of the values in self and its children.
  - Use get_values_linked_list(lnk)
get_values

self.value

self.children = [lnk1, lnk2, lnk3]

[0, 1, 4, 3, 2, 8, 9, 7, 5, 6]
get_values

- Add our own value to the start of the list
- Add results of calling get_values_linked_list on each child

return [self.value] +
    sum([[get_values_linked_list(child) for child in self.children], []])
Children having multiple children

```
0
   1  2
   |   |
   4  8
   |   |
   3  5
   |   |
   6
```

Children having multiple children
class Tree:
    def __init__(self, value, children = None):
        self.value = value
        self.children = children[:]
        if children
        else []
class Tree:

def __init__(self, value,
    children = None):
    self.value = value
    self.children = children[:]

    if children
    else []

Make a copy of children so it doesn't get mutated if they modify that list elsewhere.

Try to not use mutable types as default parameters.
Tree.get_values(self)

- Get all the values in a Tree (self)

[0, 1, 3, 4, 2, 8, 9, 7, 5] (or some order)
Tree.get_values(self)

return [self.value] ...
Tree.get_values(self)

[1, 3, 4]
Tree.get_values(self)

[1, 3, 4]

[2, 8]
Tree.get_values(self)

[1, 3, 4]
[2, 8]
[9, 7, 5]
Tree.get_values(self)

return [self.value] +

    sum([child.get_values()
         for child in
         self.children], []


Tree.get_values(self)

to_return = [self.value]
for child in self.children:
    to_return += child.get_values()
return to_return

Same thing but with a for-loop
Trees.get_values(self)

- Base case?
Trees.get_values(self)

- **Base case?** Not needed.
  - If children == [], we don't make any recursive calls (for-loop doesn't run).

```python
if children == []:
    return [self.value]
```
Terminology

Tree
Terminology

Subtrees
Node

Code-wise, Nodes and Trees are the same.
Terminology
"The node at the start (root) of the tree"
Terminology

Leaf

"A node with no children."
**Terminology**

**Internal Node**

"A non-leaf node (a node with children)"
Terminology

"Connects 2 nodes"

Edge
**Terminology**

Path
A sequence of nodes (where one node has an edge to the next)

The path 1 -> 3 -> 4
Terminology

Path
A sequence of nodes (where one node has an edge to the next)

0
/
/  /
1 2 3
/
/  /
4 8 9
/
/  /
5 7

The path 9 -> 7
The height of this tree is 4

Terminology

Height

"How many 'levels' there are in the tree"
Terminology

Level 1
Level 2
Level 3
Level 4

Height

"How many 'levels' there are in the tree"
Terminology

Depth "How far something is from the root"
**Terminology**

Depth = 0

Depth = 1

Depth = 2

Depth = 3

Depth "How far something is from the root"
Terminology

The max depth of a Tree is the height - 1

Depth: "How far something is from the root"
Terminology

"How much a tree branches out"

Arity ("Branching Factor")
Terminology

Arity = 3

"How much a tree branches out"

Arity ("Branching Factor")
"How much a tree branches out"

**Terminology**

- **Arity** = 3
- **Arity** = 1

Arity = 3

Arity = 1

("Branching Factor")

"How much a tree branches out"
Terminology

Arity = 3

Arity = 1

Arity = 1

Arity ("Branching Factor")

"How much a tree branches out"
Terminology

Arity = 3

Arity = 1

Arity = 1

Arity = 0

"Branching Factor"

"How much a tree branches out"
**Terminology**

How much a tree branches out

Terminology

<table>
<thead>
<tr>
<th>Arity</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

Arity = 3

Arity = 1

Arity = 1

Arity = 1

Arity = 0

Arity = 2

("Branching Factor")

"How much a tree branches out"
What is the height of this tree?

A) 2   B) 3   C) 4   D) 5
Answer:

D) 5
What is the depth of the node with the value 16?

A) 2  B) 3  C) 4  D) 5
Answer:

C) 3
Example: list_leaves

- Return a list of all the leaves in our Tree.
Example: list_leaves

- Return a list of all the leaves in our Tree.

```
[2, 3]
```
[9, 4, 9, 10, 13, 18, 20, 16, 17]
Example: list_leaves

- Base case
- Recursive step
Example: list_leaves

● Base case
  ○ If there are no children (we're at a leaf), return [self.value]

● Recursive step
if self.children == []:
    return [self.value]
Example: list_leaves

- Base case
  - If there are no children (we're at a leaf), return [self.value]

- Recursive step
  - Get the leaves in each child
  - Return all of those leaves
```
list_leaves

if self.children == []:
    return [self.value]

return sum([child.get_leaves() for child in self.children], [])
```
Example: `get_height`

- Return the height of our Tree.

```plaintext
Level 1 5

Height == 1
```
Example: get_height

- Return the height of our Tree.

Level 1

Level 2

Height == 2
Height == 5
Example: get_height

- Base case
- Recursive step
Example: get_height

- Base case
  - If we're at a leaf: Return 1
- Recursive step
get_height

if self.children == []:
    return 1
Example: get_height

- Base case
  - If we're at a leaf: Return 1
- Recursive step
  - Get the height of all of the children
Height is 3
Height is 4
Height is 3

Height is 4
Example: get_height

- **Base case**
  - If we're at a leaf: Return 1

- **Recursive step**
  - Get the height of all of the children
  - Get the max of those heights
  - Add 1 to it
get_height

if self.children == []:
    return 1

return 1 + max([child.get_height() for child in self.children])
Break!

For 10 minutes. :)
Family Tree

Anne

Bob
  Dot
  Iris

Edna

Carol

Frank
  Jack
  Hank
  Mary

Gina

Kate

Lisa
Example: contains(name)

- Return whether our Tree contains name.
Example: contains(value)

- Return whether our Tree contains value.
- Base case?
- Recursive step?
  - What are we recursing on?
  - What do we do with the results of our recursive call(s)?
contains

Anne

Bob

Dot

Iris

Edna

Frank

Jack

Hank

Mary

Carol

Gina

Lisa

Iris?

Sophia?
contains

Anne

Bob

Dot

Iris

Edna

Frank

Jack

Hank

Mary

Carol

Bob?

True

Gina

Lisa
contains

Bob

Dot

Edna

Iris

Anne

Carol

Frank

Kate

Gina

Lisa

Bob?
contains

Bob
- Dot
- Edna
- Iris

Bob?
True
contains

Anne

Bob

Bob?

Dot

Edna

Iris

Carol

Frank

Jack

Hank

Kate

Mary

Gina

Lisa
contains

Anne

Bob

Dot

Iris

Edna

Bob?
False

Carol

Frank

Jack

Hank

Mary

Kate

Gina

Lisa
contains

- Base case
- Recursive step
contains

● Base case
  ○ If value is the value of our Tree (root node) return True

● Recursive step
contains

if self.value == value:
    return True
contains

● Base case
  ○ If value is the value of our Tree (root node) return True

● Recursive step
  ○ Return True if any of the subtrees return True
contains

if self.value == value:
    return True

return any([child.contains(value)
              for child in
              self.children])
Find closest common ancestor

- Return the closest common ancestor
  - The value of the node that contains both of them as a descendant (child, or child of child, etc.)
- Return None if no such ancestor exists
get_closest_common_ancestor

- Will want to use the contains() method somewhere.
get_closest_common_ancestor on 1st subtree returns what?
get_closest_common_ancestor on 2nd subtree returns what?
get_closest_common_ancestor
on 1st subtree returns what?
get_closest_common_ancestor on 2nd subtree returns what?
get_closest_common_ancestor

- Return the closest common ancestor
  - The value of the node that contains both of them as a descendant (child, or child of child, etc.)
- Return None if no such ancestor exists
- You'll want to use contains().
get_closest_common_ancestor on 1st subtree returns what?
get_closest_common_ancestor on 1st subtree returns what? None
get_closest_common_ancestor on 2nd subtree returns what?
get_closest_common_ancestor on 2nd subtree returns what?

None
get_closest_common_ancestor

- If all recursive calls return None:
  - Use contains()! If we can find both values somewhere amongst our children, then **we're** an ancestor of both!
  - Otherwise, return None.
get_closest_common_ancestor on 1st subtree returns what?
get_closest_common_ancestor on 1st subtree returns what?

Bob
get_closest_common_ancestor

- If all recursive calls return None:
  - Use contains()! If we can find both values somewhere amongst our children, then we're an ancestor of both!
  - Otherwise, return None.
- Otherwise: Return the non-None value returned by one of the recursive calls.
get_closest_common_ancestor

recursive_calls =
[child.get_closest_common_ancestor(value1, value2)
  for child in self.children]

Make all of the recursive calls and store their results
get_closest_common_ancestor

```
recursive_calls =
[child.get_closest_common_ancestor(value1, value2)
  for child in self.children]

for result in recursive_calls:
    if result is not None:
        return result
```

If there's a non-None result, return it.
get_closest_common_ancestor

found_value1 = any([child.contains(value1) for child in self.children])
found_value2 = any([child.contains(value2) for child in self.children])

return self.value if (found_value1 and found_value2) else None

Return self.value if we can find value1 and value2. Otherwise, return None.
def get_values(x):
    if not isinstance(x, list):
        return [x]

    return sum([[get_values(item) for item in x]], [[]])
Traversals

- We can process nodes in different orders.
  - For example: Print the root, and then the children.
  - Or... print the children, and then the root.
Traversals
Pre-order Traversal

Root first, then children
Pre-order Traversal

Root first, then children
Pre-order Traversal

Root first, then children

1, 2
Pre-order Traversal

Root first, then children
1, 2, 4
Pre-order Traversal

Root first, then children

1, 2, 4, 5
Pre-order Traversal

Root first, then children

1, 2, 4, 5, 9
Pre-order Traversal

Root first, then children

1, 2, 4, 5, 9, 10
Pre-order Traversal

Root first, then children
1, 2, 4, 5, 9, 10, 3
Pre-order Traversal

Root first, then children
1, 2, 4, 5, 9, 10, 3, 6
Pre-order Traversal

Root first, then children
1, 2, 4, 5, 9, 10, 3, 6, 11
Pre-order Traversal

Root first, then children
1, 2, 4, 5, 9, 10, 3, 6, 11, 7
Pre-order Traversal

Root first, then children
1, 2, 4, 5, 9, 10, 3, 6, 11, 7, 8
Pre-order Traversal

Root first, then children
1, 2, 4, 5, 9, 10, 3, 6, 11, 7, 8, 12
Pre-order Traversal

Root first, then children

1, 2, 4, 5, 9, 10, 3, 6, 11, 7, 8, 12, 13
def print_preorder(self):
    print(self.value)
    for child in self.children:
        child.print_preorder()
Pre-order Traversal

Root first, then children
Pre-order Traversal

Root first, then children
Post-order Traversal

Children first, then root
Post-order Traversal

Children first, then root
Post-order Traversal

Children first, then root

4
Post-order Traversal

Children first, then root

4, 9
Post-order Traversal

Children first, then root
4, 9, 10
Post-order Traversal

Children first, then root
4, 9, 10, 5
Post-order Traversal

Children first, then root
4, 9, 10, 5, 2
Post-order Traversal

Children first, then root
4, 9, 10, 5, 2, 11
Post-order Traversal

Children first, then root
4, 9, 10, 5, 2, 11, 6
Post-order Traversal

Children first, then root
4, 9, 10, 5, 2, 11, 6, 7
Post-order Traversal

Children first, then root
4, 9, 10, 5, 2, 11, 6, 7, 12
Post-order Traversal

Children first, then root
4, 9, 10, 5, 2, 11, 6, 7, 12, 13
Post-order Traversal

Children first, then root

4, 9, 10, 5, 2, 11, 6, 7, 12, 13, 8
Post-order Traversal

Children first, then root
4, 9, 10, 5, 2, 11, 6, 7, 12, 13, 8, 3
Post-order Traversal

Children first, then root

4, 9, 10, 5, 2, 11, 6, 7, 12, 13, 8, 3, 1
for child in self.children:
    child.print_postorder()

print(self.value)
Post-order Traversal

Children first, then root
Post-order Traversal

Children first, then root
Level-over Traversal

By level
Level-over Traversal

By level 1
Level-over Traversal

By level 1, 2
Level-over Traversal

By level 1, 2, 3
Level-over Traversal

By level 1, 2, 3, 4
Level-over Traversal

By level
1, 2, 3, 4, 5
Level-over Traversal

By level
1, 2, 3, 4, 5, 6
Level-over Traversal

By level
1, 2, 3, 4, 5, 6, 7
Level-over Traversal

By level
1, 2, 3, 4, 5, 6, 7, 8
Level-over Traversal

By level
1, 2, 3, 4, 5, 6, 7, 8, 9
Level-over Traversal

By level
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Level-over Traversal

By level
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
Level-over Traversal

By level

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
Level-over Traversal

By level
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
Level-order Traversal

By level
Level-order Traversal

● Use a Queue!
  ○ Recall the 'level order' example for nested lists from the Stack/Queue Lecture
Queue Example

[1, [2, 3, [4, 5], 6], 7, [[8], 9]]
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Top
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]\]

Top

1

[2, 3, [4, 5], 6]
Queue Example

\[ [1, [2, 3, [4, 5], 6], 7, [[8], 9]] \]

Top
Queue Example

\[ [1, [2, 3, [4, 5], 6], 7, [[8], 9]] \]

Top
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Top

Printed:
Queue Example

\[\langle 1, [2, 3, [4, 5], 6], 7, [[8], 9]\rangle\]

Top

Printed: 1
Queue Example

\[ [1, [2, 3, [4, 5], 6], 7, [[8], 9]] \]

Printed: 1
Queue Example

\[ [1, [2, 3, [4, 5], 6], 7, [[8], 9]] \]
Queue Example

\[ [1, [2, 3, [4, 5], 6], 7, [[8], 9]] \]

Printed: 1
Queue Example

\[ [1, [2, 3, [4, 5], 6], 7, [[8], 9]] \]

Top

Printed: 1
# Queue Example

\[ [1, [2, 3, [4, 5], 6], 7, [[8], 9]] \]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>[[8], 9]</td>
<td>2</td>
<td>3</td>
<td>[4, 5]</td>
</tr>
</tbody>
</table>

Top

Printed: 1
Queue Example

\[[1, [2, 3, [4, 5], 6], 7, [[8], 9]]\]

Printed: 1
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Top

Printed: 1, 7
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Printed: 1, 7
Queue Example

\[[1, [2, 3, [4, 5], 6], 7, [[8], 9]]\]

Printed: 1, 7
Queue Example

[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Printed: 1, 7
Queue Example

[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Printed: 1, 7
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Top

Printed: 1, 7
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Top

Printed: 1, 7, 2
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Printed: 1, 7, 2, 3
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Printed: 1, 7, 2, 3
Queue Example

\[ [1, [2, 3, [4, 5], 6], 7, [[8], 9]] \]

Printed: 1, 7, 2, 3
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]\]

Top

Printed: 1, 7, 2, 3
Queue Example

\[[1, [2, 3, [4, 5], 6], 7, [[8], 9]]\]

Top

Printed: 1, 7, 2, 3
Queue Example

\([1, [2, 3, [4, 5], 6], 7, [[8], 9]]\)

Top

Printed: 1, 7, 2, 3, 6
Queue Example

\[[1, [2, 3, [4, 5], 6], 7, [[8], 9]]\]

Printed: 1, 7, 2, 3, 6
Queue Example

\[ [1, [2, 3, [4, 5], 6], 7, [[8], 9]] \]

Printed: 1, 7, 2, 3, 6
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Top

Printed: 1, 7, 2, 3, 6
Queue Example

\[ [1, [2, 3, [4, 5], 6], 7, [[8], 9]] \]

Top

Printed: 1, 7, 2, 3, 6
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Top

Printed: 1, 7, 2, 3, 6, 9
Queue Example

\[\{1, [2, 3, [4, 5], 6], 7, [[8], 9]\}\]

Printed: 1, 7, 2, 3, 6, 9, 4
Queue Example

[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Top

Printed: 1, 7, 2, 3, 6, 9, 4, 5
Queue Example

\[1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Top

Printed: 1, 7, 2, 3, 6, 9, 4, 5, 8
Queue Example

\[[1, [2, 3, [4, 5], 6], 7, [[8], 9]]\]

Printed: 1, 7, 2, 3, 6, 9, 4, 5, 8
Queue Example

[1, [2, 3, 4, 5, 6], 7, [8, 9]]

Printed: 1, 7, 2, 3, 6, 9, 4, 5, 8
Level-order Traversal

- Works the same:
  - Make a Queue
  - Add our Tree to it
  - Remove our Tree
  - Print the value
  - Add all of the children
Level-order Traversal

The code....

Is one of your lab questions!
Homework

● Exercise 6 (due next Thursday @ 11PM)
● Assignment 2 out on Saturday
● Midterm Remarks due Friday @ 11PM
● Assignment 1 Remarks due July 16th