Midterm 2 Review

Announcements

Plan for today

Warmup: Python lists and list mutation

Recursion strategy and practice

    Tree tree recursion

    Not a tree tree recursion

Class practice: Mic and Speakers

Bonus: Generator Where's Waldo?

Warmup: List Practice

```
def prefixes(s):
    """Return a list of all of the list prefixes of s.

    >>> prefixes([1, 2, 3])
    [[1], [1, 2], [1, 2, 3]]
    """

    so_far = []

    result = []

    for x in s:

        so_far.append(x)

        result.append(so_far)

    return result
```

pollev.com/cs61a

```
>>> prefixes([1, 2, 3])
```

Warmup: List Practice

```
def prefixes(s):
    """Return a list of all of the list prefixes of s.

    >>> prefixes([1, 2, 3])
    [[1], [1, 2], [1, 2, 3]]
    """

    so_far = []

    result = []

    for x in s:

        so_far.append(x)

        result.append(so_far)

    return result
```

pollev.com/cs61a

```
>>> prefixes([1, 2, 3])

[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

How can we fix the implementation?

pollev.com/cs61a

Warmup: List Practice

```
def prefixes(s):
    """Return a list of all of the list prefixes of s.

    >>> prefixes([1, 2, 3])
    [[1], [1, 2], [1, 2, 3]]
    """

    so_far = []

    result = []

    for x in s:

        so_far.append(x)
                      list(so_far)
        result.append(so_far)

    return result
```

pollev.com/cs61a

```
>>> prefixes([1, 2, 3])

[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

How can we fix the implementation?

pollev.com/cs61a

Recursion Recipe

- Write down an **example input**
- What **small initial choice** can I make?
- What **recursive call for each option**?
  - Write down the recursive call(s) for your example input
  - Write down what each of those calls returns for your example input

Tree Recursion Practice

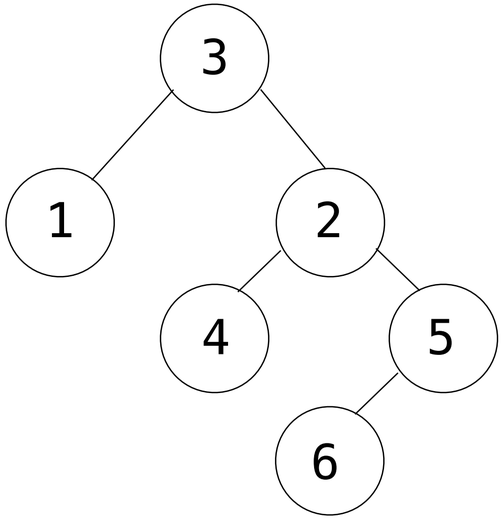
```
def trade_up(t, v):
    """
    Takes a tree with unique values, and trades the node with
    value v with its parent until v is at the top of the tree.

    >>> t = Tree(3, [Tree(2), Tree(1)])
    >>> trade_up(t, 2)
    >>> t
    Tree(2, [Tree(3), Tree(1)])
    >>> t = Tree(3, [Tree(1), Tree(2, [Tree(4), Tree(5, [Tree(6)])])])
    >>> trade_up(t, 5)
    >>> t
    Tree(5, [Tree(1), Tree(3, [Tree(2, [Tree(6)]), Tree(4)]), Tree(1)])
    """
```

Write down an example input (draw the tree!)

Tree Recursion Practice

```
def trade_up(t, v):
    """
    Takes a tree with unique values, and trades the node with
    value v with its parent until v is at the top of the tree.

    >>> t = Tree(3, [Tree(2), Tree(1)])
    >>> trade_up(t, 2)
    >>> t
    Tree(2, [Tree(3), Tree(1)])
    >>> t = Tree(3, [Tree(1), Tree(2, [Tree(4), Tree(5, [Tree(6)])])])
    >>> trade_up(t, 5)
    >>> t
    Tree(5, [Tree(1), Tree(3, [Tree(2, [Tree(6)]), Tree(4)]), Tree(1)])
    """
```

Write down an example input (draw the tree!)

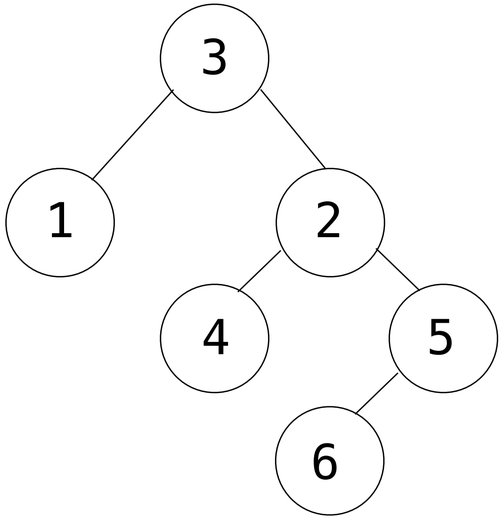


Write the recursive calls and return values

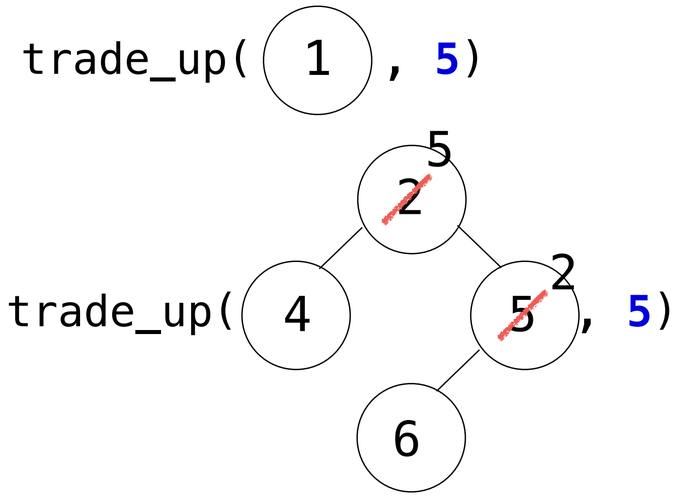
Tree Recursion Practice

```
def trade_up(t, v):
    """
    Takes a tree with unique values, and trades the node with
    value v with its parent until v is at the top of the tree.

    >>> t = Tree(3, [Tree(2), Tree(1)])
    >>> trade_up(t, 2)
    >>> t
    Tree(2, [Tree(3), Tree(1)])
    >>> t = Tree(3, [Tree(1), Tree(2, [Tree(4), Tree(5, [Tree(6)])])])
    >>> trade_up(t, 5)
    >>> t
    Tree(5, [Tree(1), Tree(3, [Tree(2, [Tree(6)]), Tree(4)]), Tree(1)])
    """
```

Write down an example input (draw the tree!)



Write the recursive calls and return values



What do we do after making the recursive call?

```
call trade_up(b, v) for each branch b

if any branch has a label of 5:
    trade that branch's label with the root
```

Tree Recursion Practice

```
def trade_up(t, v):
    """
    Takes a tree with unique values, and trades the node with
    value v with its parent until v is at the top of the tree.

    >>> t = Tree(3, [Tree(2), Tree(1)])
    >>> trade_up(t, 2)
    >>> t
    Tree(2, [Tree(3), Tree(1)])
    >>> t = Tree(3, [Tree(1), Tree(2, [Tree(4), Tree(5, [Tree(6)])])])
    >>> trade_up(t, 5)
    >>> t
    Tree(5, [Tree(1), Tree(3, [Tree(2, [Tree(6)]), Tree(4)]), Tree(1)])
    """
    if ___________:
        return
    for b in t.branches:
        ________________
        if _______________:
            ________________
            ________________
```

```
call trade_up(b, v) for each branch v

if any branch has a label of 5:
    trade that branch's label with the root
```

pollev.com/cs61a

Tree Recursion Practice

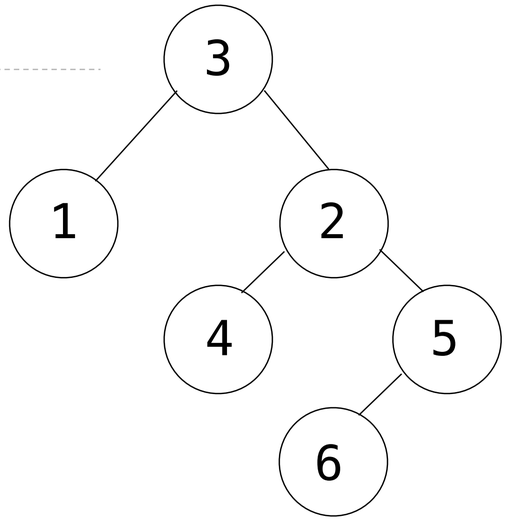
```
def trade_up(t, v):
    """
    Takes a tree with unique values, and trades the node with
    value v with its parent until v is at the top of the tree.

    >>> t = Tree(3, [Tree(2), Tree(1)])
    >>> trade_up(t, 2)
    >>> t
    Tree(2, [Tree(3), Tree(1)])
    >>> t = Tree(3, [Tree(1), Tree(2, [Tree(4), Tree(5, [Tree(6)])])])
    >>> trade_up(t, 5)
    >>> t
    Tree(5, [Tree(1), Tree(3, [Tree(2, [Tree(6)]), Tree(4)]), Tree(1)])
    """
    if t.is_leaf():
        return
    for b in t.branches:
        trade_up(b, v)
        if b.label == v:
            b.label = t.label
            t.label = v
```

call trade_up(b, v) for each branch v

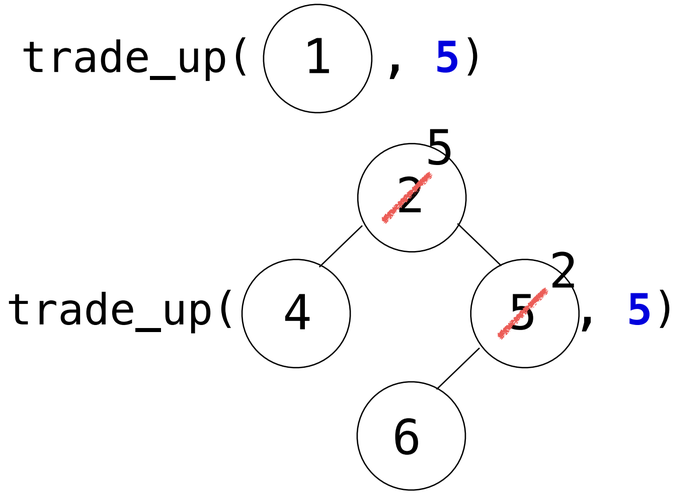
if any branch has a label of 5:
    trade that branch's label with the root

pollev.com/cs61a

Recursion Recipe



- Write down an **example input**
- What **small initial choice** can I make?
- What **recursive call for each option**?
  - Write down the recursive call(s) for your example input
  - Write down what each of those calls returns for your example input




```
call trade_up(b, v) for each branch v

if any branch has a label of 5:
    trade that branch's label with the root
```

Dictionary/Recursion Practice

Make Change

coins is a dictionary from denominations to counts. Two nickels and a quarter is {5: 2, 25: 1}

remove_one(coins, amount) returns a dictionary with one fewer count:

remove_one({5: 2, 25: 1}, 5) -> {5: 1, 25: 1} **remove_one**({5: 2, 25: 1}, 25) -> {5: 2}

```
def make_change(amount, coins):
    """Return a list of coins that sum to amount, preferring the smallest coins
    available and placing the smallest coins first in the returned list."""
```

- What **small initial choice** can I make?
- What **recursive call for each option**?

Use a 2 or don't use a 2

```
>>> coins = {2: 2, 3: 2, 4: 3, 5: 1}
>>> make_change(8, coins)
[2, 2, 4]
>>> make_change(25, coins)
[2, 3, 3, 4, 4, 4, 5]
```

make_change(25, {2: 2, 3: 2, 4: 3, 5: 1})

Returns [2, 3, 3, 4, 4, 4, 5]

use a 2

make_change(____, _____)

Returns _____

Make Change

coins is a dictionary from denominations to counts. Two nickels and a quarter is {5: 2, 25: 1}

remove_one(coins, amount) returns a dictionary with one fewer count:

remove_one({5: 2, 25: 1}, 5) -> {5: 1, 25: 1} **remove_one**({5: 2, 25: 1}, 25) -> {5: 2}

```
def make_change(amount, coins):
    """Return a list of coins that sum to amount, preferring the smallest coins
    available and placing the smallest coins first in the returned list."""
```

- What **small initial choice** can I make?
- What **recursive call for each option**?

Use a 2 or don't use a 2

```
>>> coins = {2: 2, 3: 2, 4: 3, 5: 1}
>>> make_change(8, coins)
[2, 2, 4]
>>> make_change(25, coins)
[2, 3, 3, 4, 4, 4, 5]
```

make_change(**25**, {**2**: **2**, **3**: **2**, **4**: **3**, **5**: **1**})

Returns [2, 3, 3, 4, 4, 4, 5]

use a 2

don't use a 2

make_change(**23**, {**2**: **1**, **3**: **2**, **4**: **3**, **5**: **1**})

Returns [3, 3, 4, 4, 4, 5]

make_change(**25**, {**3**: **2**, **4**: **3**, **5**: **1**})

Returns

Make Change

coins is a dictionary from denominations to counts. Two nickels and a quarter is {5: 2, 25: 1}

remove_one(coins, amount) returns a dictionary with one fewer count:

remove_one({5: 2, 25: 1}, 5) -> {5: 1, 25: 1} **remove_one**({5: 2, 25: 1}, 25) -> {5: 2}

```
def make_change(amount, coins):
    """Return a list of coins that sum to amount, preferring the smallest coins
    available and placing the smallest coins first in the returned list."""
    if not coins:
        return None
    smallest = min(coins)
    rest = remove_one(coins, smallest)
    if amount < smallest:
        return None
    elif amount == smallest:

        return ___________
    else:
        result = make_change(________________, rest)
        if result:
            return ____________________
        else:
            return make_change(amount, rest)
```

```
>>> coins = {2: 2, 3: 2, 4: 3, 5: 1}
>>> make_change(8, coins)
[2, 2, 4]
>>> make_change(25, coins)
[2, 3, 3, 4, 4, 4, 5]
```

Make Change

coins is a dictionary from denominations to counts. Two nickels and a quarter is {5: 2, 25: 1}

remove_one(coins, amount) returns a dictionary with one fewer count:
  remove_one({5: 2, 25: 1}, 5) -> {5: 1, 25: 1} **remove_one**({5: 2, 25: 1}, 25) -> {5: 2}

25 {2: 2, 3: 2, 4: 3, 5: 1}

```
def make_change(amount, coins):
    """Return a list of coins that sum to amount, preferring the smallest coins
    available and placing the smallest coins first in the returned list."""
    if not coins:
        return None
    smallest = min(coins)
    rest = remove_one(coins, smallest)
    if amount < smallest:
        return None
    elif amount == smallest:
        return [smallest]
    else:
        result = make_change(amount-smallest, rest)
        if result:
            return [smallest] + result
        else:
            return make_change(amount, rest)
```

```
>>> coins = {2: 2, 3: 2, 4: 3, 5: 1}
>>> make_change(8, coins)
[2, 2, 4]
>>> make_change(25, coins)
[2, 3, 3, 4, 4, 4, 5]
```

smallest is 2

rest is {2: 1, 3: 2, 4: 3, 5: 1}

Use a 2

23

result is [3, 3, 4, 4, 4, 5]

[2] + [3, 3, 4, 4, 4, 5] -> [2, 3, 3, 4, 4, 4, 5]

pollev.com/cs61a

Why??

Don't use a 2

rest is {2: 1, 3: 2, 4: 3, 5: 1}

Mic and Speakers (Spring 2024 Midterm)

```
class Mic:
    """A microphone connected to speakers.

    >>> m = Mic()  # Front is connected automatically
    >>> m.sing('Is this thing on?')
    Front - Is this thing on?
    >>> Speaker(str.lower).connect(m, 'Left Side')
    >>> m.sing("You belong with me.")
    Front - You belong with me.
    Left Side - you belong with me.
    """
    def __init__(self):
        self.speakers = _______

    def sing(self, lyrics: str):
        for k in self.speakers.keys():
            print(k, '-', self.speakers[k].repeat(lyrics))

class Speaker:
    def __init__(self, transform):
        self.transform = transform

    def connect(self, m: Mic, location: str):
        _______

    def repeat(self, s: str) -> str:
        return _______
```

pollev.com/cs61a

If you sing lyrics into a mic, every connected speaker repeats them.

A **Mic** instance has a dictionary **speakers** containing **Speaker** instances as values, each with its **location** (**str**) as its key. Its **sing** method takes a string **lyrics** and invokes the **repeat** method of each **Speaker** instance connected to it.

A **Speaker** takes a **transform** function that takes and returns a string. To **connect** a **Speaker** instance to **m** (**Mic**) in a **location** (**str**), add that instance to the **speakers** dictionary of **m** in that **location.** To **repeat** a signal **s** (**str**), return the result of calling the speaker's **transform** function on **s.**

Every **Mic** starts connected to a **Speaker** in the **Front** location that repeats the exact same signal it receives.

Mic and Speakers (Spring 2024 Midterm)

```
class Mic:
    """A microphone connected to speakers.

    >>> m = Mic()  # Front is connected automatically
    >>> m.sing('Is this thing on?')
    Front - Is this thing on?
    >>> Speaker(str.lower).connect(m, 'Left Side')
    >>> m.sing("You belong with me.")
    Front - You belong with me.
    Left Side - you belong with me.
    """
    def __init__(self):
        self.speakers = {'Front': Speaker(lambda x: x)}

    def sing(self, lyrics: str):
        for k in self.speakers.keys():
            print(k, '-', self.speakers[k].repeat(lyrics))

class Speaker:
    def __init__(self, transform):
        self.transform = transform

    def connect(self, m: Mic, location: str):
        m.speakers[location] = self

    def repeat(self, s: str) -> str:
        return self.transform(s)
```

pollev.com/cs61a

If you sing lyrics into a mic, every connected speaker repeats them.

A **Mic** instance has a dictionary **speakers** containing **Speaker** instances as values, each with its **location** (**str**) as its key. Its **sing** method takes a string **lyrics** and invokes the **repeat** method of each **Speaker** instance connected to it.

A **Speaker** takes a **transform** function that takes and returns a string. To **connect** a **Speaker** instance to **m** (**Mic**) in a **location** (**str**), add that instance to the **speakers** dictionary of **m** in that **location.** To **repeat** a signal **s** (**str**), return the result of calling the speaker's **transform** function on **s.**

Every **Mic** starts connected to a **Speaker** in the **Front** location that repeats the exact same signal it receives.

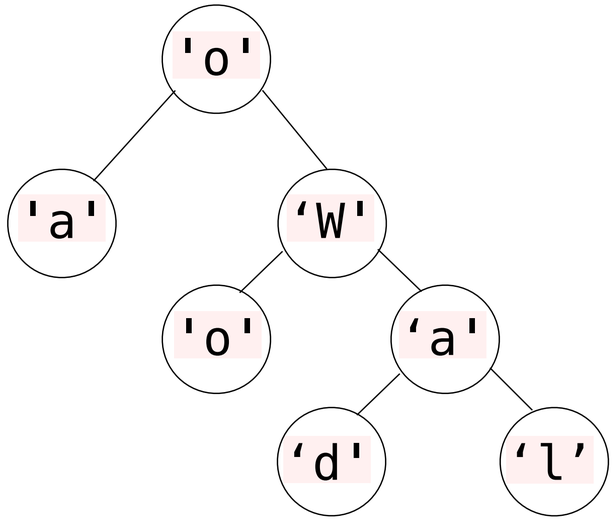
Generator Where's Waldo

```
def all_elements(t: Tree):
    for b in t.branches:
        yield from all_elements(b)
    yield t.label
```

```
>>> t = Tree('o', [Tree('a'), Tree('W', [Tree('o'), Tree('a', [Tree('d'), Tree('l')])])])
```

Generator Where's Waldo

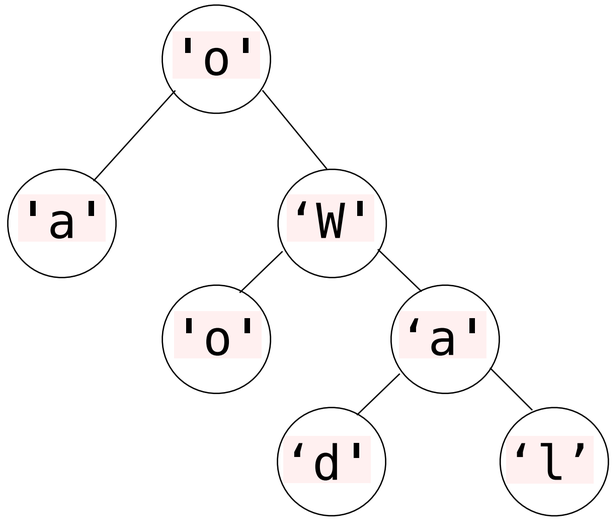
```
def all_elements(t: Tree):
    for b in t.branches:
        yield from all_elements(b)
    yield t.label
```




```
>>> t = Tree('o', [Tree('a'), Tree('W', [Tree('o'), Tree('a', [Tree('d'), Tree('l')])])])
>>> all = all_elements(t)
>>> all
```

Generator Where's Waldo

```
def all_elements(t: Tree):
    for b in t.branches:
        yield from all_elements(b)
    yield t.label
```




```
>>> t = Tree('o', [Tree('a'), Tree('W', [Tree('o'), Tree('a', [Tree('d'), Tree('l')])])])
>>> all = all_elements(t)
>>> all
<generator object all_elements at 0x104e4ddd0>
>>> next(all)
```

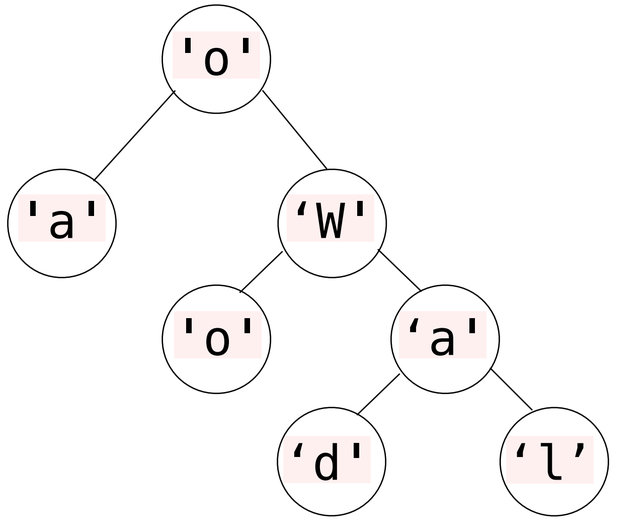
pollev.com/cs61a

Write an expression that uses all and evaluates to ['W', 'a', 'l', 'd', 'o']

pollev.com/cs61a

Generator Where's Waldo

```
def all_elements(t: Tree):
    for b in t.branches:
        yield from all_elements(b)
    yield t.label
```




```
>>> t = Tree('o', [Tree('a'), Tree('W', [Tree('o'), Tree('a', [Tree('d'), Tree('l')])])])
>>> all = all_elements(t)
>>> all
<generator object all_elements at 0x104e4ddd0>
>>> next(all)
'a'

>>> list(reversed(list(all)[:-1]))
['W', 'a', 'l', 'd', 'o']
```

pollev.com/cs61a

Write an expression that uses all and evaluates to ['W', 'a', 'l', 'd', 'o']

pollev.com/cs61a