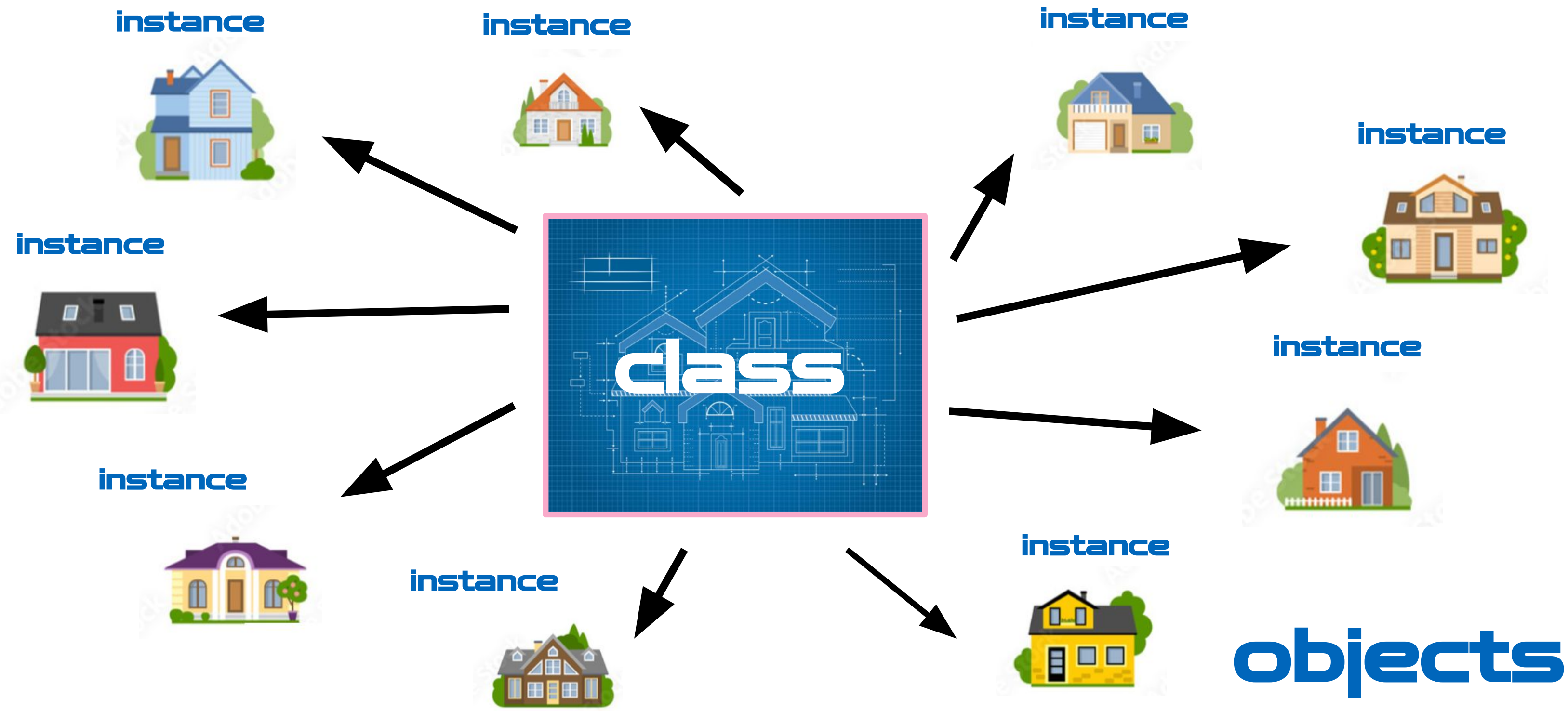
Lecture 16: Mutable Trees

CS 61A - Summer 2024
Raymond Tan

Review: Objects + Inheritance

Objects



instance
instance
instance
instance
instance
class
instance
instance
instance
instance
instance
objects

Definitions

- Class
  - A template for creating objects
- Instance
  - A single object created from a class
- Instance Variable
  - A data attribute of an object, specific to an instance
- Class Variable
  - A data attribute of an object, shared by all instances of a class
- Instance method
  - A function that operates on individual instances of a class
- Constructor
  - A method that specifies how to initialize an individual instance

Example of A Complete Class Definition

class declaration

class variable

constructor

instance variable

instance method

```
class Album():
    albums_by_label = {}

    def __init__(self, name, release_year, label):
        self.name = name
        self.release_year = release_year
        self.label = label
        self.sales = 0
        self.streams = 0

        if label not in Album.albums_by_label:
            Album.albums_by_label[label] = [self]
        else:
Album.albums_by_label[label].append(self)

    def sell(self, amount):
        self.sales += amount

    def stream(self):
        self.streams += 1

    def is_platinum(self):
        return self.sales + 150 * self.streams >
1000000
```

Inheritance vs Composition

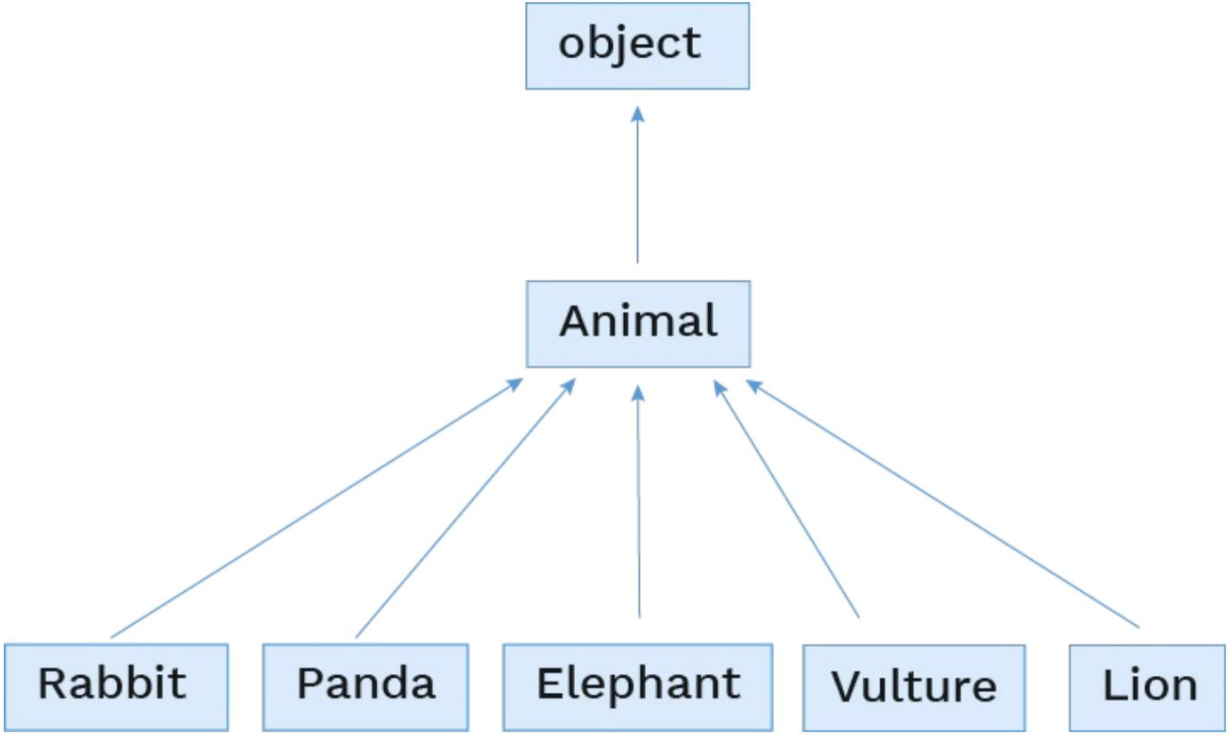
OOP is best when we think about the following metaphor:

Inheritance is best for representing **is-a** relationships

- A panda *is* an animal so Panda inherits from Animal

Composition is best for representing **has-a** relationships

- An album *has* listeners
- A zoo *has* animals


object
Animal
Rabbit
Panda
Elephant
Vulture
Lion

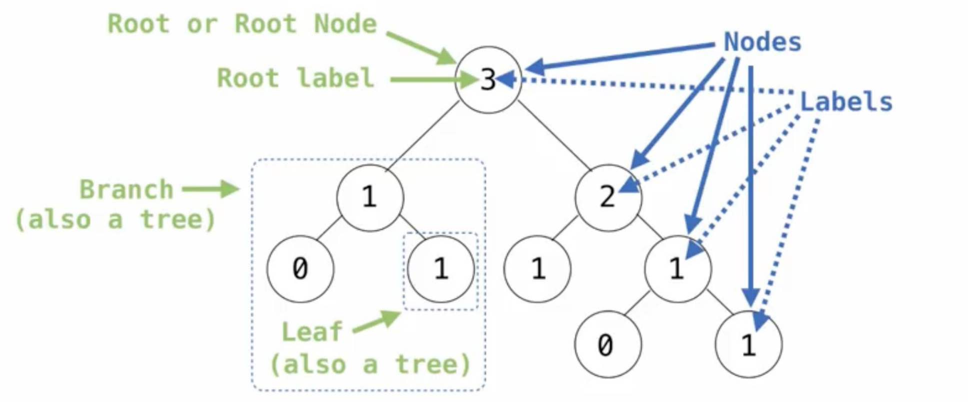
The Album Class + The Listener Class

```
class Album():
      albums_by_label = {}

      def __init__(self, name, release_year, label):
            self.name = name
            self.release_year = release_year
            self.label = label
            self.sales = 0
            self.streams = 0

            if label not in Album.albums_by_label:
                  Album.albums_by_label[label] = [self]
            else:

Album.albums_by_label[label].append(self)

      def sell(self, amount):
            self.sales += amount

      def stream(self):
            self.streams += 1

      def is_platinum(self):
            return self.sales + 150 * self.streams >
1000000
```

```
class Listener():
      def __init__(self, name,
favorite_album):
            self.name = name
            self.favorite_album = favorite_album
            self.albums = []
            self.buy(favorite_album)

      def buy(self, album):
      album.sell(1)
      self.albums.append(album)
```

Mutable Trees

Review: ADT Trees



Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

A **tree** starts at the **root**

Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **label** that can be any value

One node can be the **parent**/**child** of another

The top node is the **root node**

Mutable Trees

- Now that we've learned about the idea of using classes to create custom objects, let's revisit the Tree ADT and see how we can convert that to a formal Python class
- Why?
  - Using formal classes translate more directly to how we would create custom objects in the real world
    - Introducing ADTs was a delayed introduction to OOP
  - These new trees are mutable, which unlock different methods of problem solving

Mutable Trees vs Tree ADT

```python
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
```

Tree class

```python
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
```

Tree ADT

Note that the concepts are still the same!

Drawing out diagrams for trees, terminology associated with trees, etc

is_leaf

`is_leaf` is a ***function*** that returns whether the tree is a leaf (a node with no branches)

```
def is_leaf(self):
    return not self.branches
```

Tree class

```
def is_leaf(tree):
    return not branches(tree)
```

Tree ADT

Note that is_leaf is **not an attribute** of the Tree class!

count_leaves: Revisited

```
def count_leaves(t):
    if is_leaf(t):
        return 1
    else:
        branch_counts = 0
        for b in branches(t):
            branch_counts += count_leaves(b)
        return branch_counts
```

Let’s try to implement this using the Tree class!

count_leaves: Revisited

```python
def count_leaves_2(t):
    if t.is_leaf():
        return 1
    else:
        branch_counts = 0
        for b in t.branches:
            branch_counts += count_leaves_2(b)
        return branch_counts
```

As shown: We can apply a majority of the concepts we've already learned to solving problems using the Tree class.

String Representation of Trees

String Representations

- Strings are important - they represent language and programs
- Printing strings is exactly how the interpreter communicates back to us
    - We should be able to use strings to represent objects!
- Python gives us two ways/functions to represent objects
    - **str**: returns a representation that a **human** should be able to read
        - used by the print function
    - **repr**: returns a representation that the **interpreter** should be able to read
        - used by the interpreter for printing results
    - these are usually the same, but not always!

String representation of trees

- When creating tree objects using code, it often is hard to visualize the properties of each node, which make up the overall tree structure
- The String representation of trees makes visualizing trees easier!

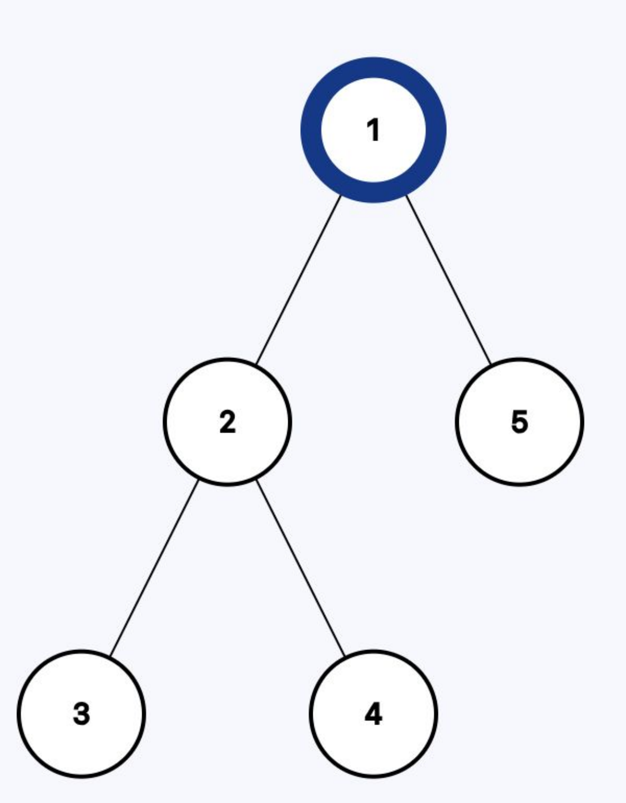
```python
def __repr__(self):
    if self.branches:
        branch_str = ', ' + repr(self.branches)
    else:
        branch_str = ''
    return 'Tree({0}{1})'.format(self.label, branch_str)
```

`repr` method
"Computer readable"

```python
def __str__(self):
    def print_tree(t, indent=0):
        tree_str = '  ' * indent + str(t.label) + "\n"
        for b in t.branches:
            tree_str += print_tree(b, indent + 1)
        return tree_str
    return print_tree(self).rstrip()
```

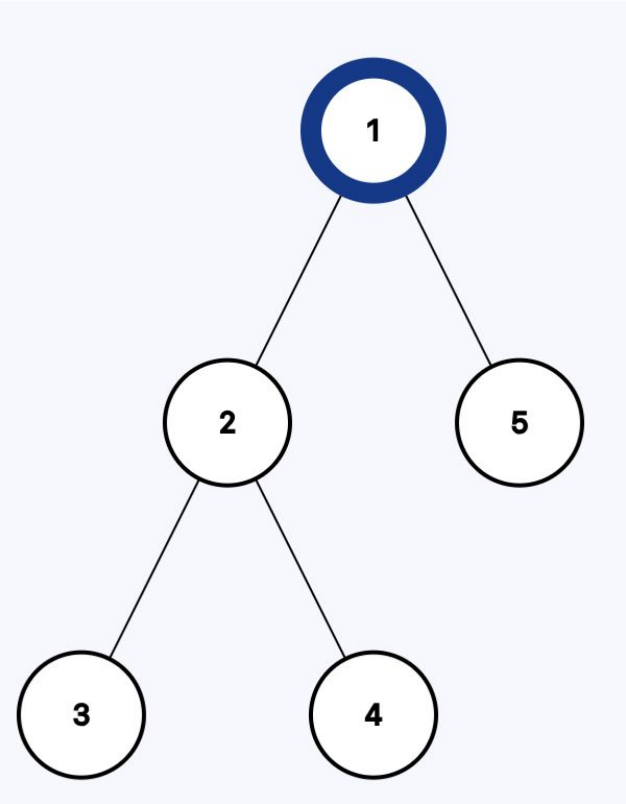
`str` method
"Human readable"

Examples




```
>>> t = Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(5)])
>>> t
Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(5)])
>>> print(t)
1
  2
    3
    4
  5
```

Examples



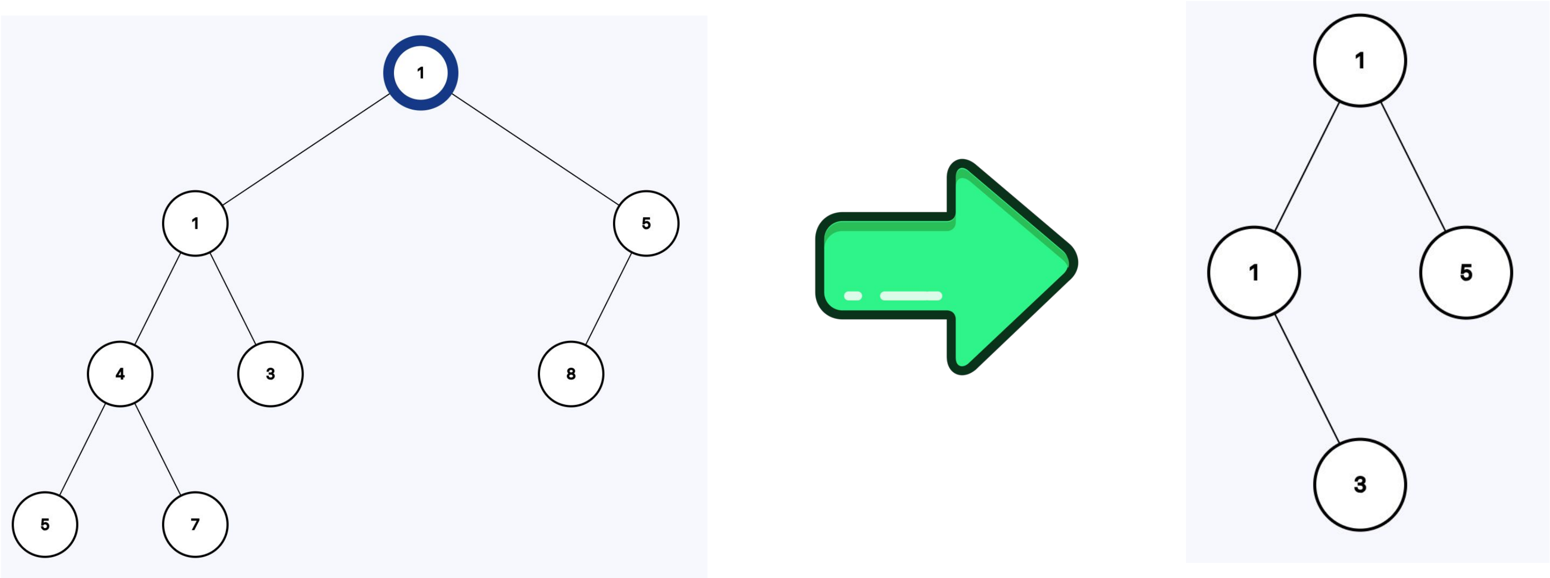

```
>>> t = Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(5)])
>>> repr(t)
'Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(5)])'
>>> str(t)
'1\n  2\n    3\n    4\n  5'
```

Invoking an object directly in the interpreter calls repr on the object, and prints out the output
Printing a string calls str on the object, and prints out the output

Examples

Example: prune_evens

- Problem: Given an input tree `t`, ***mutate*** the tree such that all labels with an even value are removed from the tree.
    - If a node containing an even value is removed, all branches belonging to that node are removed as well



prune_evens: Solution

`is_even` is a function we defined in lecture that takes in a value and returns whether it is even

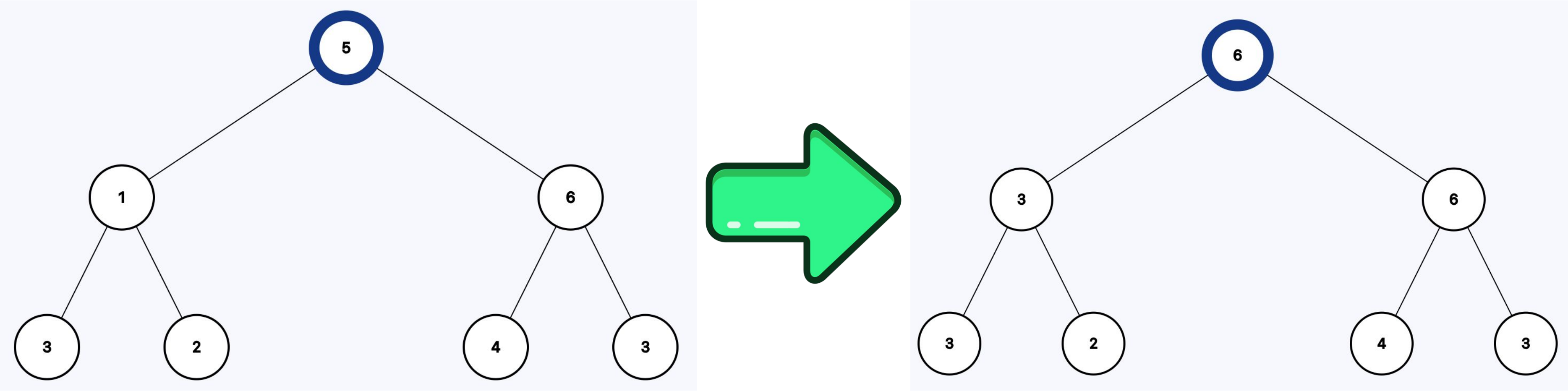
```
def prune_even(t):
    """Mutate the tree t such that all nodes containing even values are removed."""
    if is_even(t.label):
        return
    new_branches = []
    for b in t.branches:
        if not is_even(b.label):
            prune_even(b)
            new_branches.append(b)
    t.branches = new_branches # This is where the mutative step happens!
```

To mutate an object, we need to modify one of its attributes (in this case, modifying the `branches` attribute)

The return in the base case is used to stop the recursion, not to explicitly return a value used by a previous frame

Example: largest_of_group

- Problem: Given an input tree `t`, ***mutate*** each node such that the label of a node is the largest among all of its children plus its original value label.



largest_of_group: Solution #1

```
def largest_of_group(t):
    largest_labels = [t.label]
    for b in t.branches:
        largest_of_group(b)
        largest_labels.append(b.label)
    t.label = max(largest_labels)
```

largest_of_group: Solution #2

```
def largest_of_group(t):
    for b in t.branches:
        largest_of_group(b)
    t.label = max([t] + t.branches, key=lambda n : n.label).label
```

Here, we were taking the max over a list of tree nodes. Python doesn't know how to compare tree nodes directly, so we pass in a `key` argument which specifies what attribute of tree nodes we use during comparison.
Note that max still evaluates to a tree node, and not the attribute that the `key` function was using!

Break

Example: keep_k_largest

- Problem: Given a tree t and a value k, ***mutate*** the tree such that each node has no more than k branches. If a node has more than k branches, keep the k largest ones.

keep_k_largest: Solution

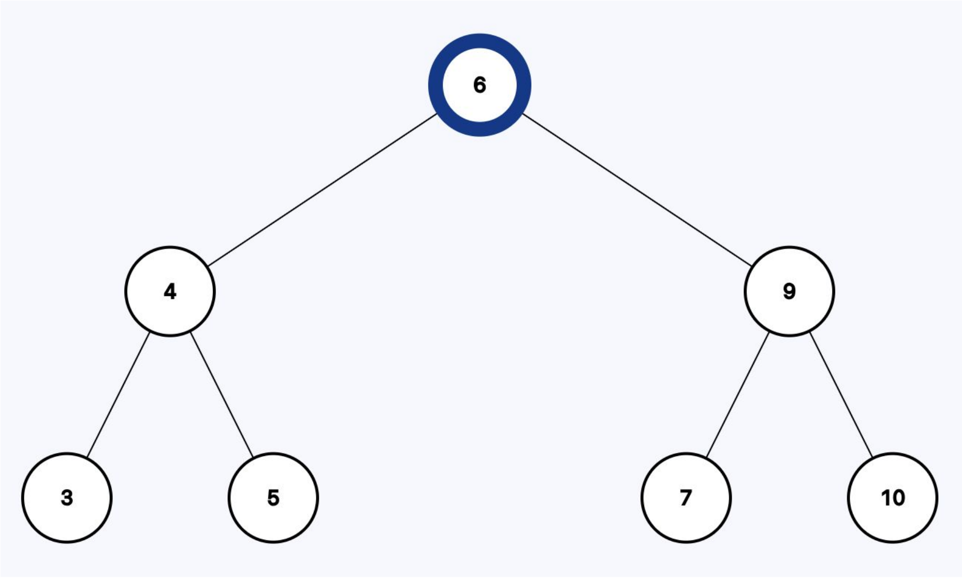
```
def keep_k_largest(t, k):
    n = len(t.branches)
    counter = k
    while counter > n:
        smallest = min(t.branches, key=lambda n : n.label)
        t.branches.remove(smallest)
        counter -= 1
    for b in t.branches:
        keep_k_largest(b, k)
```

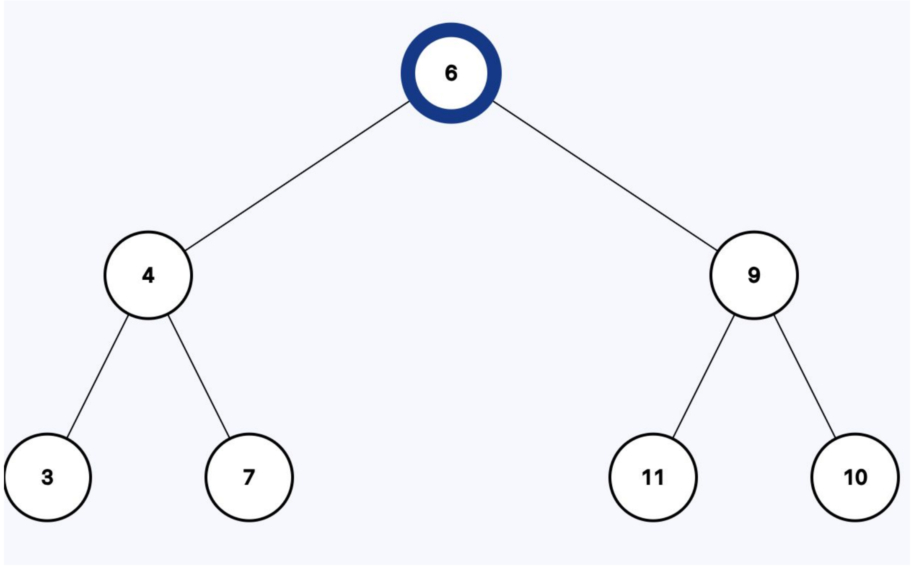
Extra: Binary Search Trees

Binary Search Trees

- Binary Search Trees (BSTs) are a specific type of tree with the following properties:
  - Every node has 0, 1, or 2 branches (binary)
  - Every value to the **left** of a node is ***less*** than the label of the current node, and every value to the **right** of a node is ***greater*** than the label of the current node
    - Definitions vary, but oftentimes BSTs do not allow duplicate values. We'll assume this implementation
- We'll also assume that with our Tree class, if a node has exactly one branch, it is the left branch
- These properties of Binary Search Trees make them very efficient to search for values

BSTs vs Non-BSTs





BST
✅

Not a BST
❌

Example: search_tree

- Problem: Given a BST `t` and a value `x`, return whether or not `x` exists as a label in `t`.

search_tree: Solution #1

```python
def search_tree(t, x):
    if t.label == x:
        return True
    for b in t.branches:
        found = search_tree(b, x)
        if found:
            return True
    return False
```

With this approach, are we taking advantage of the properties of BSTs?

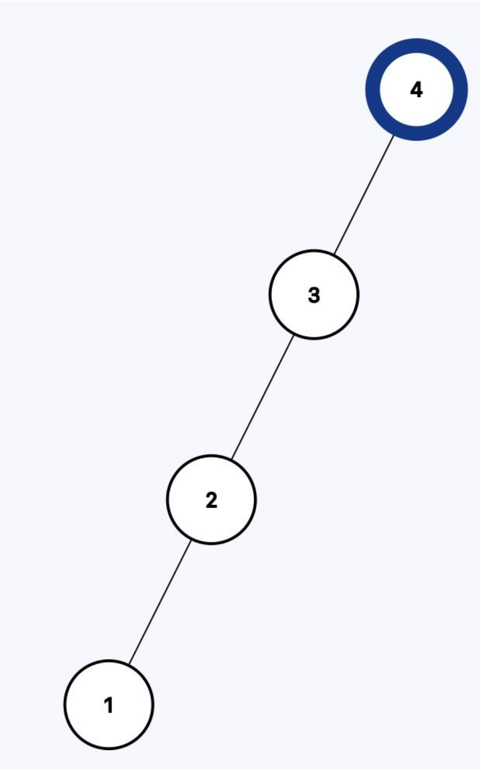
search_tree: Solution #2

```python
def search_tree(t, x):
    if t.label == x:
        return True
    n = len(t.branches)
    if n == 0: # Leaf node
        return False
    elif n == 1: # Only has left branch
        if x < t.label:
            return search_tree(t.branches[0])
        else:
            return False
    else: # Has both left and right branch
        if x < t.label:
            return search_tree(t.branches[0])
        else: # x > t.label
            return search_tree(t.branches[1])
```

Assuming we have a “balanced” BST, searching for a value is logarithmic/O(log n) runtime with this approach.

Is searching BSTs always logarithmic?

- No. Consider the case of a “stringy” BST
- This is a valid BST!
- However, it doesn’t allow us to get the behavior of halving all values at each step
- In fact, this structure is closer to a Linked List (our topic for tomorrow’s lecture!)



Summary

- The Tree class is the OOP version of the Tree ADT
  - ADTs are a delayed introduction to OOP
- Same concepts from Tree ADT can be applied to Tree OOP!
- Trees created from the Tree class are mutable
  - No longer have to create new trees to solve problems
- Extra: Binary Search Trees are an efficient data structure for lookup of data