Lecture 18: Mutable Trees

Mitas Ray
07/21/2016
Announcements
Trees
Terminology
Terminology
Terminology

• **Node**: single unit containing an entry
Terminology

- **Node**: single unit containing an entry
- **Root**: top node
Terminology

- **Node**: single unit containing an entry
- **Root**: top node
- **Leaf**: a node with no children
Terminology

- **Node**: single unit containing an entry
- **Root**: top node
- **Leaf**: a node with no children
- **Children**: subtree with a parent
Tree Class
class Tree:
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        self.entry = entry
        self.children = children
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children
class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children
class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

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

![Diagram of tree structure](tree.png)
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.entry
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.entry
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.entry
4
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.entry
3
class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.entry
3
>>> t.children[0].entry
1
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.entry
3
>>> t.children[0].entry
1
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.entry
3
>>> t.children[0].entry
1
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.entry
3
>>> t.children[0].entry
2
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.entry
3
>>> t.children[0].entry
2
>>> t.children[1].is_leaf()
Tree Class

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.entry
3
>>> t.children[0].entry
2
>>> t.children[1].is_leaf()
Tree Class

```python
class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.entry
3
>>> t.children[0].entry
2
>>> t.children[1].is_leaf()
True
```
Comparison to ADT
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

def tree(entry, children=[]):
    return [entry, children]

def entry(tree):
    return tree[0]

def children(tree):
    return tree[1]
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ...
... [Tree(1)]), Tree(4)])

def tree(entry, children=[]):
    return [entry, children]

def entry(tree):
    return tree[0]

def children(tree):
    return tree[1]
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ...					[Tree(1)]), Tree(4)])

def tree(entry, children=[]):
    return [entry, children]

def entry(tree):
    return tree[0]

def children(tree):
    return tree[1]
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ... [Tree(1)]), Tree(4)])
>>> t_adt = tree(3, [tree(2, ... [tree(1)]), tree(4)])
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ...
...     [Tree(1)]), Tree(4)])
>>> t_adt = tree(3, [tree(2, ...
...     [tree(1)]), tree(4)])

```python
def tree(entry, children=[]):
    return [entry, children]
def entry(tree):
    return tree[0]
def children(tree):
    return tree[1]
```
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ...
...     [Tree(1)]), Tree(4)])
>>> t_adt = tree(3, [tree(2, ...
...     [tree(1)]), tree(4)])
>>> t_class.entry == entry(t_adt)
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ...
...    [Tree(1)], Tree(4))])
>>> t_adt = tree(3, [tree(2, ...
...    [tree(1)], tree(4))])
>>> t_class.entry == entry(t_adt)
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ...
                     [Tree(1)]), Tree(4)])
>>> t_adt = tree(3, [tree(2, ...
                     [tree(1)]), tree(4)])
>>> t_class.entry == entry(t_adt)
True
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ...
...   [Tree(1)]), Tree(4)])
>>> t_adt = tree(3, [tree(2, ...
...   [tree(1)]), tree(4)])
>>> t_class.entry == entry(t_adt)
True
>>> t_class.entry = 5

def tree(entry, children=[]):
    return [entry, children]
def entry(tree):
    return tree[0]
def children(tree):
    return tree[1]
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ...
    [Tree(1)]), Tree(4)])
>>> t_adt = tree(3, [tree(2,
    ...
    [tree(1)]), tree(4)])
>>> t_class.entry == entry(t_adt)
True
>>> t_class.entry = 5
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ... [Tree(1)]), Tree(4)])
>>> t_adt = tree(3, [tree(2, ... [tree(1)]), tree(4)])
>>> t_class.entry == entry(t_adt)
True
>>> t_class.entry = 5
Comparison to ADT

class Tree:
def __init__(self, entry,
    children=[]):
  for c in children:
    assert isinstance(c, Tree)
  self.entry = entry
  self.children = children

>>> t_class = Tree(3, [Tree(2, ...
    ...
      [Tree(1)]), Tree(4)])
>>> t_adt = tree(3, [tree(2, ...
    ...
      [tree(1)]), tree(4)])
>>> t_class.entry == entry(t_adt)
True
>>> t_class.entry = 5
>>> entry(t_adt) = 5
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ...
...    [Tree(1)]), Tree(4)])

>>> t_adt = tree(3, [tree(2, ...
...    [tree(1)]), tree(4)])

>>> t_class.entry == entry(t_adt)
True

>>> t_class.entry = 5
>>> entry(t_adt) = 5
SyntaxError: can't assign ...

---

def tree(entry, children=[]):
    return [entry, children]

def entry(tree):
    return tree[0]

def children(tree):
    return tree[1]
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ... [Tree(1)]), Tree(4)])
>>> t_adt = tree(3, [tree(2, ... [tree(1)]), tree(4)])
>>> t_class.entry == entry(t_adt)
True
>>> t_class.entry = 5
>>> entry(t_adt) = 5
SyntaxError: can't assign ...
>>> t_class.entry == entry(t_adt)
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ...
...  [Tree(1)]), Tree(4)])
>>> t_adt = tree(3, [tree(2,
...  [tree(1)]), tree(4)])
>>> t_class.entry == entry(t_adt)
True
>>> t_class.entry = 5
>>> entry(t_adt) = 5
SyntaxError: can't assign ...
>>> t_class.entry == entry(t_adt)
Comparison to ADT

class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

>>> t_class = Tree(3, [Tree(2, ... [Tree(1)]), Tree(4)])
>>> t_adt = tree(3, [tree(2, ... [tree(1)]), tree(4)])
>>> t_class.entry == entry(t_adt)
True
>>> t_class.entry = 5
>>> entry(t_adt) = 5
SyntaxError: can't assign ...
>>> t_class.entry == entry(t_adt)
False
Map
Map

• Want to apply a function \( fn \) to each element in the tree

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
```
Map

• Want to apply a function \( fn \) to each element in the tree

• Main Ideas

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

def map(self, fn):
```
Map

• Want to apply a function $\text{fn}$ to each element in the tree

• Main Ideas
  - Apply $\text{fn}$ to current node (mutate tree)

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
```
Map

• Want to apply a function \( \text{fn} \) to each element in the tree

• Main Ideas
  - Apply \( \text{fn} \) to current node (mutate tree)
  - Call \text{map} on children

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
        ...
```
Map

- Want to apply a function $\mathbf{fn}$ to each element in the tree

Main Ideas
- Apply $\mathbf{fn}$ to current node (mutate tree)
- Call $\mathbf{map}$ on children

class Tree:
    def __init__(self, entry, children=[]):
        ...

def map(self, fn):
    self.entry = fn(self.entry)
Map

• Want to apply a function \( \text{fn} \) to each element in the tree

• Main Ideas
  - Apply \( \text{fn} \) to current node (mutate tree)
  - Call \text{map} on children

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
        self.entry = fn(self.entry)
```
Map

• Want to apply a function $fn$ to each element in the tree

• Main Ideas
  - Apply $fn$ to current node (mutate tree)
  - Call $map$ on children

```python
class Tree:
  def __init__(self, entry, children=[]): ...

  def map(self, fn):
    self.entry = fn(self.entry)
    for c in self.children:
```

Map

- Want to apply a function \( \text{fn} \) to each element in the tree

Main Ideas
- Apply \( \text{fn} \) to current node (mutate tree)
- Call \texttt{map} on children

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)
```
Map

• Want to apply a function $\text{fn}$ to each element in the tree

• Main Ideas
  - Apply $\text{fn}$ to current node (mutate tree)
  - Call $\text{map}$ on children

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)
```
**Map**

- Want to apply a function $\text{fn}$ to each element in the tree

**Main Ideas**
- Apply $\text{fn}$ to current node (mutate tree)
- Call $\text{map}$ on children

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)
```
class Tree:
    def __init__(self, entry, children=[]): ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)

>>> square = lambda x: x * x
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)

>>> square = lambda x: x * x
>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
Map

class Tree:
    def __init__(self, entry, children=[]): ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)

>>> square = lambda x: x * x
>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.map(square)
Map

class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)

>>> square = lambda x: x * x
>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.map(square)
Map

class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)

>>> square = lambda x: x * x
>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.map(square)
Map

class Tree:
    def __init__(self, entry, children=[]): ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)

>>> square = lambda x: x * x
>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.map(square)
Map

class Tree:
    def __init__(self, entry, children=[]): ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)

>>> square = lambda x: x * x
>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.map(square)
Map

class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)

>>> square = lambda x: x * x
>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.map(square)
class Tree:
    def __init__(self, entry, children=[]): ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)

>>> square = lambda x: x * x
>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.map(square)
class Tree:
    def __init__(self, entry, children=[]): ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)

>>> square = lambda x: x * x
>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.map(square)
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def map(self, fn):
        self.entry = fn(self.entry)
        for c in self.children:
            c.map(fn)

>>> square = lambda x: x * x
>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> t.map(square)
Existence
Existence

• Does the tree contain element $e$?

class Tree:
    def __init__(self, entry, children=[]):
        ... 

    def __contains__(self, e):
        ...
Existence

• Does the tree contain element e?
• Main Ideas

```python
class Tree:
    def __init__(self, entry, children=[]): ...

def __contains__(self, e):
```
Existence

• Does the tree contain element e?

• Main Ideas
  - Check entry of current node

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        ...
```
Existence

• Does the tree contain element e?

• Main Ideas
  - Check entry of current node
  - Otherwise, check children

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        ...
```

Existence

• Does the tree contain element e?

• Main Ideas
  - Check entry of current node
  - Otherwise, check children
    ▸ If no children to investigate, return False

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...
    def __contains__(self, e):
        ...
```
Existence

• Does the tree contain element \( e \)?

• Main Ideas
  - Check entry of current node
  - Otherwise, check children
    ‣ If no children to investigate, return False

```python
class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
```
Existence

• Does the tree contain element $e$?

• Main Ideas
  - Check entry of current node
  - Otherwise, check children
    ‣ If no children to investigate, return False

```python
class Tree:
    def __init__(self, entry, children=[]):
        ... 

    def __contains__(self, e):
        if self.entry == e:
            return True
```

Existence

- Does the tree contain element $e$?

Main Ideas

- Check entry of current node
- Otherwise, check children
  - If no children to investigate, return False

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            ...
```

Existence

• Does the tree contain element e?

• Main Ideas
  - Check entry of current node
  - Otherwise, check children
    ‣ If no children to investigate, return False

class Tree:
    def __init__(self, entry, children=[]):
      ...

    def __contains__(self, e):
      if self.entry == e:
        return True
      for c in self.children:
        if c.__contains__(e):
          return True
Existence

- Does the tree contain element e?

Main Ideas

- Check entry of current node
- Otherwise, check children
  - If no children to investigate, return False

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...  # Initialize the tree

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if c.__contains__(e):
                return True
```

Existence

• Does the tree contain element e?

• Main Ideas
  - Check entry of current node
  - Otherwise, check children
    ‣ If no children to investigate, return False

class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if c.__contains__(e):
                return True
        return False
Existence

• Does the tree contain element $e$?

• Main Ideas
  - Check entry of current node
  - Otherwise, check children
    ‣ If no children to investigate, return False

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if c.__contains__(e):
                return True
        return False
```
Existence

- Does the tree contain element $e$?

Main Ideas

- Check entry of current node
- Otherwise, check children
  - If no children to investigate, return False

```python
class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if c.__contains__(e):
                return True
        return False
```
Existence

• Does the tree contain element e?

• Main Ideas
  - Check entry of current node
  - Otherwise, check children
    ‣ If no children to investigate, return False

```python
class Tree:
    def __init__(self, entry, children=[]): ...  

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False
```
Existence

• Does the tree contain element e?

• Main Ideas
  - Check entry of current node
  - Otherwise, check children
    ‣ If no children to investigate, return False

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False
```
class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False
Existence

class Tree:
    def __init__(self, entry, children=[]):
        ...

def __contains__(self, e):
    if self.entry == e:
        return True
    for c in self.children:
        if e in c:
            return True
    return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
Existence

class Tree:
    def __init__(self, entry, children=[]): ...  
    
    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> 8 in t
class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> 8 in t
Existence

class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> 8 in t
Existence

class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> 8 in t
Existence

class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> 8 in t
Existence

class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> 8 in t
Existence

class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> 8 in t
False
Existence

```python
class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> 8 in t
False
>>> 2 in t
```
Existence

class Tree:
    def __init__(self, entry, children=[]):

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> 8 in t
False
>>> 2 in t

**Existence**

```python
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> 8 in t
False
>>> 2 in t
```
class Tree:
    def __init__(self, entry, children=[]): ...  

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> 8 in t
False
>>> 2 in t
Existence

class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(3, [Tree(2, [Tree(1)]), Tree(4)])
>>> 8 in t
False
>>> 2 in t
True
Binary Search Tree
Definition
Definition

• Each node has at most 2 children, left and right
Definition

- Each node has at most 2 children, left and right
- Left child elements are all less than or equal to entry
Definition

• Each node has at most 2 children, left and right

• Left child elements are all less than or equal to entry

• Right child elements are all greater than entry
Definition

• Each node has at most 2 children, left and right

• Left child elements are all less than or equal to entry

• Right child elements are all greater than entry

• Left child and right child are also BSTs
Definition

• Each node has at most 2 children, left and right

• Left child elements are all less than or equal to entry

• Right child elements are all greater than entry

• Left child and right child are also BSTs

• Only contains numbers!
Definition

- Each node has at most 2 children, left and right
- Left child elements are all less than or equal to entry
- Right child elements are all greater than entry
- Left child and right child are also BSTs
- Only contains numbers!
Definition

- Each node has at most 2 children, left and right
- Left child elements are all less than or equal to entry
- Right child elements are all greater than entry
- Left child and right child are also BSTs
- Only contains numbers!
Definition

- Each node has at most 2 children, left and right
- Left child elements are all less than or equal to entry
- Right child elements are all greater than entry
- Left child and right child are also BSTs
- Only contains numbers!
Definition

• Each node has at most 2 children, left and right

• Left child elements are all less than or equal to entry

• Right child elements are all greater than entry

• Left child and right child are also BSTs

• Only contains numbers!
Definition

- Each node has at most 2 children, left and right
- Left child elements are all less than or equal to entry
- Right child elements are all greater than entry
- Left child and right child are also BSTs
- Only contains numbers!
Definition

• Each node has at most 2 children, left and right
• Left child elements are all less than or equal to entry
• Right child elements are all greater than entry
• Left child and right child are also BSTs
• Only contains numbers!
Binary Search Tree Class
class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children
class BST:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children
class BST:
    empty = ()
    def __init__(self, entry, left=empty, right=empty):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children
class BST:
    empty = ()
    def __init__(self, entry, left=empty, right=empty):
        for c in self.children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.left, self.right = left, right
Binary Search Tree Class

class BST:
    empty = ()
def __init__(self, entry, left=empty, right=empty):
    assert left is BST.empty or isinstance(left, BST)
    assert right is BST.empty or isinstance(right, BST)

    self.entry = entry
    self.left, self.right = left, right
class BST:
    empty = ()
    def __init__(self, entry, left=empty, right=empty):
        assert left is BST.empty or isinstance(left, BST)
        assert right is BST.empty or isinstance(right, BST)
        self.entry = entry
        self.left, self.right = left, right

@property
def max(self): ...  # Returns the maximum element in the BST

@property
def min(self): ...  # Returns the minimum element in the BST
class BST:
    empty = ()
    def __init__(self, entry, left=empty, right=empty):
        assert left is BST.empty or isinstance(left, BST)
        assert right is BST.empty or isinstance(right, BST)

        self.entry = entry
        self.left, self.right = left, right

        if left is not BST.empty:
            assert left.max <= entry
        if right is not BST.empty:
            assert entry < right.min

@property
def max(self): ...  # Returns the maximum element in the BST

@property
def min(self): ...  # Returns the minimum element in the BST
Binary Search Tree Class

class BST:
    empty = ()
    def __init__(self, entry, left=empty, right=empty):
        assert left is BST.empty or isinstance(left, BST)
        assert right is BST.empty or isinstance(right, BST)

        self.entry = entry
        self.left, self.right = left, right

        if left is not BST.empty:
            assert left.max <= entry
        if right is not BST.empty:
            assert entry < right.min

@property
def max(self): ...  # Returns the maximum element in the BST

@property
def min(self): ...  # Returns the minimum element in the BST
Existence
Existence

• Does the tree contain element e?

• Main Ideas

  - Check entry of current node
  
  - Otherwise, check children
    ‣ If no children to investigate, return False

```python
class BST:
    def __init__(self, entry, left=empty, right=empty):
        ...

    def __contains__(self, e):
        ...
```
Existence

• Does the BST contain element e?

• Main Ideas
  - Check entry of current node
  - Otherwise, check children
    ‣ If no children to investigate, return False

```python
class BST:
    def __init__(self, entry, left=empty, right=empty):
        ...

def __contains__(self, e):
    ...
```
Existence

• Does the BST contain element e?

• Main Ideas
  - Check entry of current node
  - Otherwise, check left or right
    • If no children to investigate, return False

```python
class BST:
    def __init__(self, entry, left=empty, right=empty):
        ...
    def __contains__(self, e):
        ...
```
Existence

• Does the BST contain element \( e \)?

• Main Ideas

  - Check entry of current node
  
  - Otherwise, check left or right
    
    ▪ If no children to investigate, return False

```python
class BST:
    def __init__(self, entry, left=empty, right=empty):
        ...

    def __contains__(self, e):
```

Existence

• Does the BST contain element e?

• Main Ideas
  - Check entry of current node
  - Otherwise, check left or right
    - If no children to investigate, return False

```python
class BST:
    def __init__(self, entry, left=empty, right=empty):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False
```
Existence

- Does the BST contain element e?

Main Ideas
- Check entry of current node
- Otherwise, check left or right
  - If no children to investigate, return False
Existence

• Does the BST contain element e?

• Main Ideas
  - Check entry of current node
  - Otherwise, check left or right
    • If no children to investigate, return False

```python
class BST:
    def __init__(self, entry, left=empty, right=empty): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        elif e < self.entry and self.left is not BST.empty:
            return e in self.left
        elif e > self.entry and self.right is not BST.empty:
            return e in self.right
        return False
```
Existence

• Does the BST contain element e?

• Main Ideas
  - Check entry of current node
  - Otherwise, check left or right
    ‣ If no children to investigate, return False

```python
class BST:
    def __init__(self, entry, left=empty, right=empty): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        elif e < self.entry and self.left is not BST.empty:
            return e in self.left
        elif e > self.entry and self.right is not BST.empty:
            return e in self.right
        return False
```
Runtime Comparison
Runtime Comparison

• Is there a difference in runtime when we check existence in a tree versus a BST?
Runtime Comparison

• Is there a difference in runtime when we check existence in a tree versus a BST?

• Runtime in terms of $n$, the number of nodes
Runtime Comparison

• Is there a difference in runtime when we check existence in a tree versus a BST?

• Runtime in terms of $n$, the number of nodes
Runtime Comparison

class Tree:
    def __init__(self, entry, children=[]):
        ...  
    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(7, [Tree(3, [Tree(2), ... Tree(5)]), Tree(13, ... [Tree(11), Tree(17)])[)])
class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(7, [Tree(3, [Tree(2), ...
    ...    Tree(5)]), Tree(13,
    ...    [Tree(11), Tree(17)]))]
>>> 11 in t
class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(7, [Tree(3, [Tree(2), ...
...     Tree(5)]), Tree(13, ...
...     [Tree(11), Tree(17)]]))
>>> 11 in t
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(7, [Tree(3, [Tree(2),
...    Tree(5)]), Tree(13,
...    [Tree(11), Tree(17)]]])

>>> 11 in t
class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(7, [Tree(3, [Tree(2), ... Tree(5)]), Tree(13, ... [Tree(11), Tree(17)]]))

>>> 11 in t
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(7, [Tree(3, [Tree(2), ...
        ...    Tree(5)]), Tree(13,
        ...    ... [Tree(11), Tree(17)]])
>>> 11 in t
Runtime Comparison

class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(7, [Tree(3, [Tree(2), ...
...    Tree(5)]), Tree(13, ...
...    [Tree(11), Tree(17)]]))
>>> 11 in t
class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(7, [Tree(3, [Tree(2), ...
                Tree(5)]), Tree(13, ...
                [Tree(11), Tree(17)]]))
>>> 11 in t
Runtime Comparison

class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(7, [Tree(3, [Tree(2), ...
...     Tree(5)]), Tree(13,
...     ...
...     [Tree(11), Tree(17)]))
>>> 11 in t
class Tree:
    def __init__(self, entry, children=[]):
        ...

def __contains__(self, e):
    if self.entry == e:
        return True
    for c in self.children:
        if e in c:
            return True
    return False

>>> t = Tree(7, [Tree(3, [Tree(2), ... Tree(5)]), Tree(13, ... [Tree(11), Tree(17)]]))
>>> 11 in t
class Tree:
    def __init__(self, entry, children=[]): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(7, [Tree(3, [Tree(2), ..., Tree(5)]), Tree(13, ..., [Tree(11), Tree(17)]))]
>>> 11 in t
```
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(7, [Tree(3, [Tree(2), ... Tree(5)]), Tree(13, ... [Tree(11), Tree(17)]]]
>>> 11 in t
True
```
class Tree:
    def __init__(self, entry, children=[]):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        for c in self.children:
            if e in c:
                return True
        return False

>>> t = Tree(7, [Tree(3, [Tree(2), ...
        ...    Tree(5)]), Tree(13,
        ...    [Tree(11), Tree(17)]))]
>>> 11 in t
True
class BST:
    def __init__(self, entry, left=empty, right=empty):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        elif e < self.entry and self.left is not BST.empty:
            return e in self.left
        elif e > self.entry and self.right is not BST.empty:
            return e in self.right
        return False
class BST:
    def __init__(self, entry, left=empty, right=empty):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        elif e < self.entry and self.left is not BST.empty:
            return e in self.left
        elif e > self.entry and self.right is not BST.empty:
            return e in self.right
        return False

>>> bst = BST(7,
...    ...    BST(3, BST(2), BST(5)),
...    ...    BST(13, BST(11), BST(17)))
class BST:
    def __init__(self, entry, left=empty, right=empty): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        elif e < self.entry and self.left is not BST.empty:
            return e in self.left
        elif e > self.entry and self.right is not BST.empty:
            return e in self.right
        return False

>>> bst = BST(7, ...
...    BST(3, BST(2), BST(5)), ...
...    BST(13, BST(11), BST(17)))
>>> 11 in bst
class BST:
    def __init__(self, entry, left=empty, right=empty):
        ... 

    def __contains__(self, e):
        if self.entry == e:
            return True
        elif e < self.entry and self.left is not BST.empty:
            return e in self.left
        elif e > self.entry and self.right is not BST.empty:
            return e in self.right
        return False

>>> bst = BST(7,
...    ...    BST(3, BST(2), BST(5)),
...    ...    BST(13, BST(11), BST(17)))
>>> 11 in bst
class BST:
    def __init__(self, entry, left=empty, right=empty):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        elif e < self.entry and self.left is not BST.empty:
            return e in self.left
        elif e > self.entry and self.right is not BST.empty:
            return e in self.right
        return False

>>> bst = BST(7,
...    ...BST(3, BST(2), BST(5)),
...    ...BST(13, BST(11), BST(17))
>>> 11 in bst
class BST:
    def __init__(self, entry, left=empty, right=empty): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        elif e < self.entry and self.left is not BST.empty:
            return e in self.left
        elif e > self.entry and self.right is not BST.empty:
            return e in self.right
        return False

>>> bst = BST(7, ...
...    BST(3, BST(2), BST(5)), ...
...    BST(13, BST(11), BST(17)))
>>> 11 in bst
class BST:
    def __init__(self, entry, left=empty, right=empty): ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        elif e < self.entry and self.left is not BST.empty:
            return e in self.left
        elif e > self.entry and self.right is not BST.empty:
            return e in self.right
        return False

>>> bst = BST(7,
...    BST(3, BST(2), BST(5)),
...    BST(13, BST(11), BST(17)))
>>> 11 in bst
class BST:
    def __init__(self, entry, left=empty, right=empty):
        ...

    def __contains__(self, e):
        if self.entry == e:
            return True
        elif e < self.entry and self.left is not BST.empty:
            return e in self.left
        elif e > self.entry and self.right is not BST.empty:
            return e in self.right
        return False

>>> bst = BST(7, ...
    ...     BST(3, BST(2), BST(5)), ...
    ...     BST(13, BST(11), BST(17)))
>>> 11 in bst
class BST:
    def __init__(self, entry, left=empty, right=empty): ...
    
def __contains__(self, e):
        if self.entry == e:
            return True
        elif e < self.entry and self.left is not BST.empty:
            return e in self.left
        elif e > self.entry and self.right is not BST.empty:
            return e in self.right
        return False

>>> bst = BST(7,
...    BST(3, BST(2), BST(5)),
...    BST(13, BST(11), BST(17)))
>>> 11 in bst
True
**Runtime Comparison**

class BST:
    def __init__(self, entry, left=empty, right=empty): ...

def __contains__(self, e):
    if self.entry == e:
        return True
    elif e < self.entry and self.left is not BST.empty:
        return e in self.left
    elif e > self.entry and self.right is not BST.empty:
        return e in self.right
    return False

>>> bst = BST(7, ...
...    BST(3, BST(2), BST(5)), ...
...    BST(13, BST(11), BST(17)))

>>> 11 in bst
True
Summary
Summary
Summary

• Trees created with a class are mutable!
Summary

- Trees created with a class are mutable!

- BSTs allow us to organize our data in left child and right child based on value
Summary

- Trees created with a class are mutable!
- BSTs allow us to organize our data in left child and right child based on value
- BST allows for more efficient search
Summary

• Trees created with a class are mutable!

• BSTs allow us to organize our data in left child and right child based on value

• BST allows for more efficient search
  - $\Theta(n)$ in regular tree
Summary

- Trees created with a class are mutable!

- BSTs allow us to organize our data in left child and right child based on value

- BST allows for more efficient search
  - \( \Theta(n) \) in regular tree
  - \( \Theta(\log n) \) in BST