Lecture 18: Mutable Trees

Mitas Ray
07/21/2016
Announcements
Trees
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:
    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

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

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

Main Ideas
- Apply \( fn \) 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

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

• 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**

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

• 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 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

• 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

• 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
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

Θ(n)
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

- 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