61A Lecture 21

Announcements

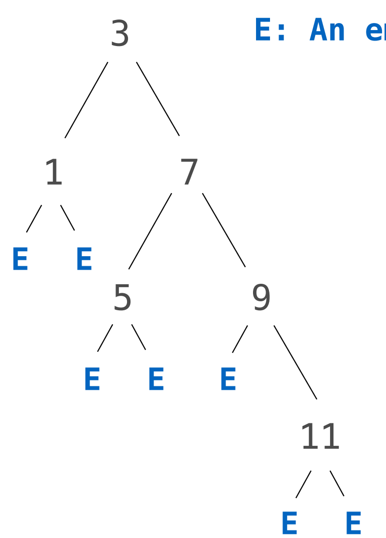
Binary Trees

Binary Tree Class

A binary tree is a tree that has a left branch and a right branch

Idea: Fill the place of a missing left branch with an empty tree

Idea 2: An instance of BTree always has *exactly* two branches



E: An empty tree

```
class BTree(Tree):
    empty = Tree(None)

    def __init__(self, label, left=empty, right=empty):
        Tree.__init__(self, label, [left, right])

    @property
    def left(self):
        return self.branches[0]

    @property
    def right(self):
        return self.branches[1]


t = BTree(3, BTree(1),
             BTree(7, BTree(5),
                      BTree(9, BTree.empty,
                               BTree(11))))
```

(Demo)

```
~/lec$ python3 -i ex.py
>>> BTree(3)
BTree(3)
>>> BTree(3).is_leaf()
True
>>> BTree(3, BTree(1), BTree(5))
BTree(3, BTree(1), BTree(5))
>>> t = BTree(3, BTree(1), BTree(5))
>>> t.left
BTree(1)
>>> t.right
BTree(5)
>>> t.label
3
>>> t.left.label
1
>>>
```

```
class BTree(Tree):
    """A tree with exactly two branches, which m
    empty = Tree(None)

    def __init__(self, label, left=empty, right=
        for b in (left, right):
            assert isinstance(b, BTree) or b is BTree.empty
        Tree.__init__(self, label, (left, right))

    @property
    def left(self):
        return self.branches[0]

    @property
    def right(self):
        return self.branches[1]

    def is_leaf(self):
        return [self.left, self.right] == [BTree.empty] * 2

    def __repr__(self):
        if self.is_leaf():
            return 'BTree({0})'.format(self.label)
        elif self.right is BTree.empty:
            left = repr(self.left)
            return 'BTree({0}, {1})'.format(self.label, left)
        else:
            left, right = repr(self.left), repr(self.right)
            if self.left is BTree.empty:
                left = 'BTree.empty'
            template = 'BTree({0}, {1}, {2})'
            return template.format(self.label, left, right)
```

```
~/lec$ python3 -i ex.py
>>> fib_tree(3)
BTree(2, BTree(1), BTree(1, BTree(0), BTree(1)))
>>> contents(fib_tree(3))
[1, 2, 0, 1, 1]
>>>
```

```
def fib_tree(n):
    """A Fibonacci binary tree."""
    if n == 0 or n == 1:
        return BTree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = left.label + right.label
        return BTree(fib_n, left, right)

def contents(t):
    if t is BTree.empty:
        return []
    else:
        return contents(t.left) + [t.label] + contents(t.right)
```

Binary Search Trees

Binary Search

A strategy for finding a value in a sorted list: check the middle and eliminate half

```
20 in [1, 2, 4, 8, 16, 32, 64]
      [1, 2, 4, 8, 16, 32, 64]
      [1, 2, 4, 8, 16, 32, 64]

False
```

```
4 in [1, 2, 4, 8, 16, 32]
     [1, 2, 4, 8, 16, 32]
     [1, 2, 4, 8, 16, 32, 64]

True
```

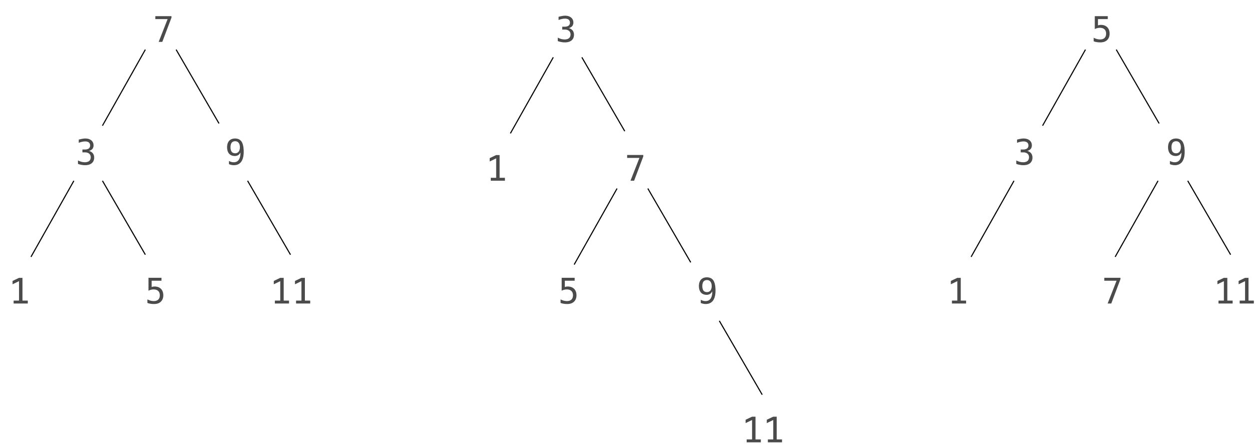
For a sorted list of length n, what Theta expression describes the time required? $\Theta(\log n)$

Binary Search Trees

advantage of storing labels in binary search trees over sorted lists:
it is quicker to add new labels in trees than in sorted lists

A binary search tree is a binary tree where each node’s label is:

- Larger than all node labels in its left branch and
- Smaller than all node labels in its right branch



(Demo)

the best tree: balanced tree -- left and right branches have similar numbers of labels

Binary Search Tree

```
~/lec$ python3 -i ex.py
>>> balanced_bst([3])
BTree(3)
>>> balanced_bst([3, 4, 5])
BTree(4, BTree(3), BTree(5))
>>> balanced_bst(range(10))
BTree(5, BTree(2, BTree(1, BTree(0)), BTree(4, BTree(3))), BTre
e(8, BTree(7, BTree(6)), BTree(9)))
>>> pretty(balanced_bst(range(10)))
                             5

             2                             8

     1               4               7            9

  0               3               6
>>> balanced_bst(range(10)).left.right.label
4
```

```
def balanced_bst(s):
    """Construct a binary search tree from a sor
    if not s:
        return BTree.empty
    else:
        mid = len(s) // 2
        left = balanced_bst(s[:mid])
        right = balanced_bst(s[mid+1:])
        return BTree(s[mid], left, right)
```

not s -- empty

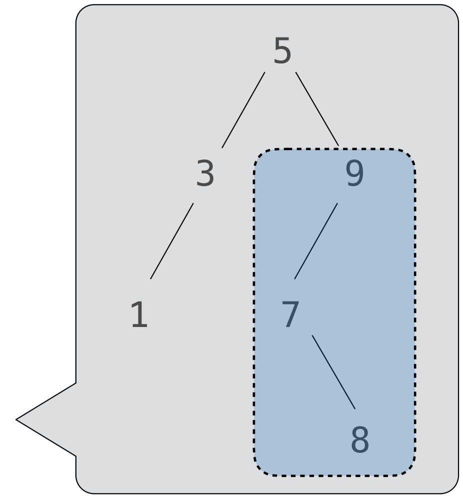
Discussion Questions

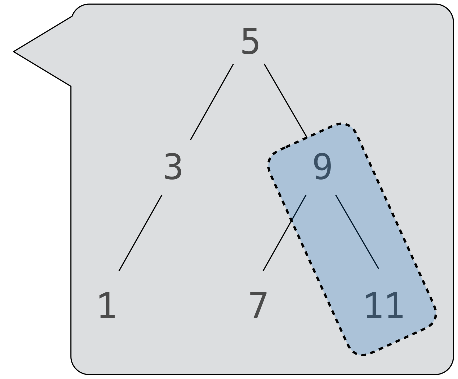
What's the largest element in a binary search tree?

```
def largest(t):
    if t.right is BTree.empty:
        return t.label
    else:
        return largest(t.right)
```

What's the second largest element in a binary search tree?

```
def second(t):
    if t.is_leaf():
        return None
    elif t.right is BTree.empty:
        return largest(t.left)
    elif t.right.is_leaf():
        return t.label
    else:
        return second(t.right)
```





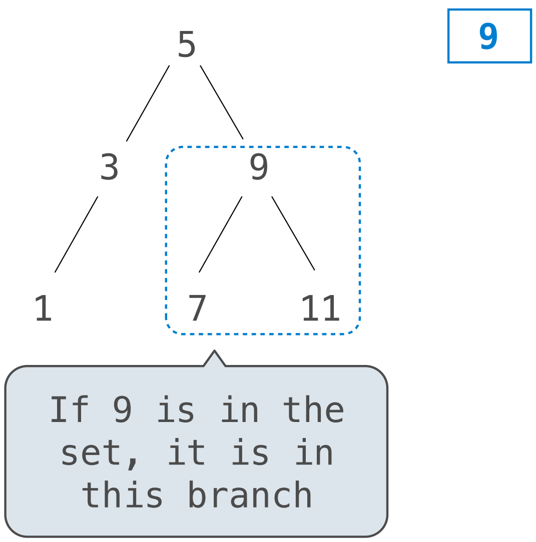
Sets as Binary Search Trees

Membership in Binary Search Trees

contains traverses the tree

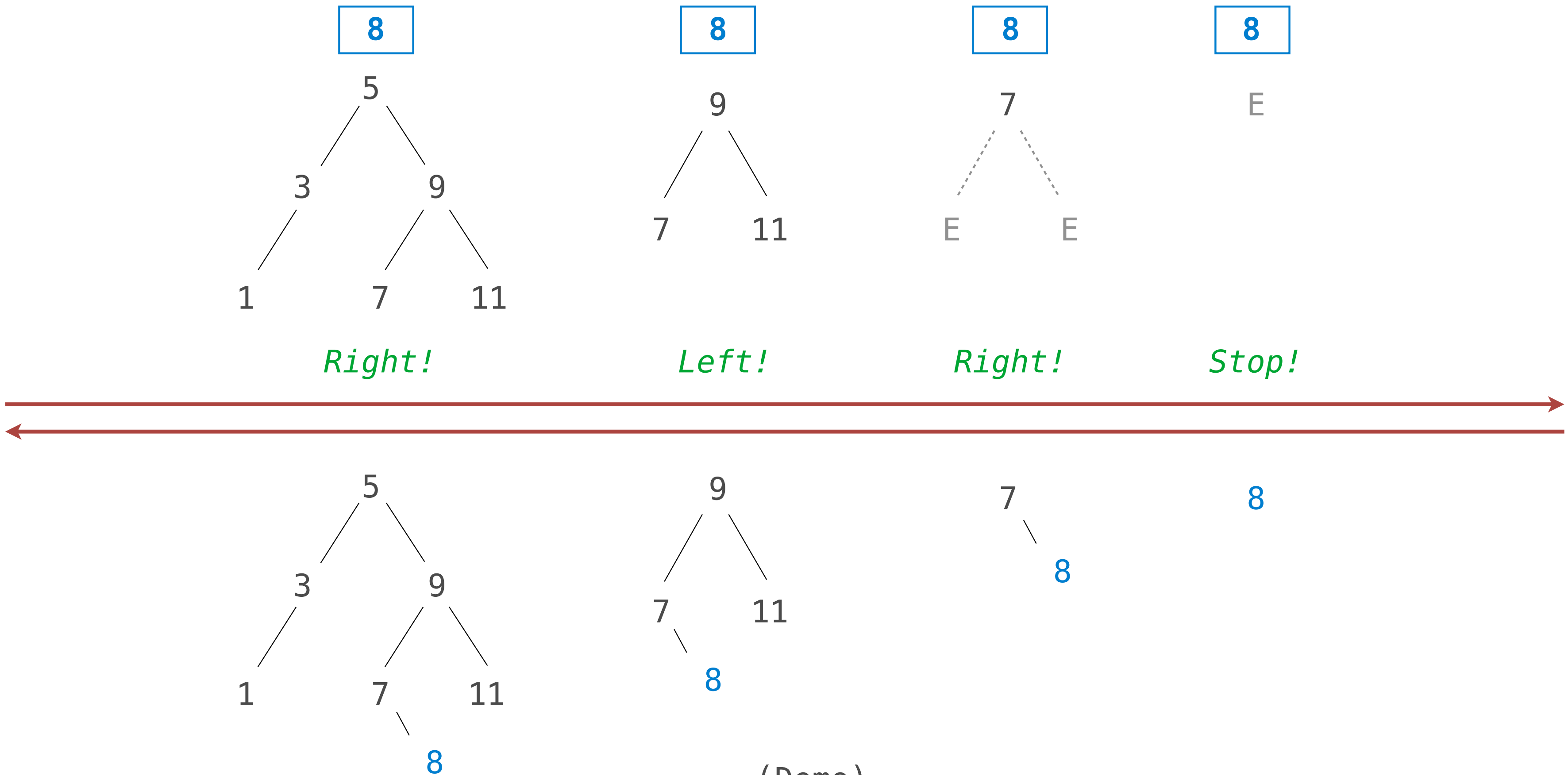
- If the element is not at the root, it can only be in either the left or right branch
- By focusing on one branch, we reduce the set by the size of the other branch

```
def contains(s, v):
    if s is BTree.empty:
        return False
    elif s.label == v:
        return True
    elif s.label < v:
        return contains(s.right, v)
    elif s.label > v:
        return contains(s.left, v)
```



Order of growth? $\Theta(h)$ on average $\Theta(\log n)$ on average for a balanced tree

Adjoining to a Tree Set



(Demo)

```
~/lec$ python3 -i ex.py
>>> odds = [2*n+1 for n in range(6)]
>>> odds
[1, 3, 5, 7, 9, 11]
>>> t = bst(odds)
>>> t
BTree(7, BTree(3, BTree(1), BTree(5)), BTree(11, BTree(9)))
>>> adjoin(t, 8)
BTree(7, BTree(3, BTree(1), BTree(5)), BTree(11, BTree(9, BTree(8))))
>>> adjoin(t, 3)
BTree(7, BTree(3, BTree(1), BTree(5)), BTree(11, BTree(9)))
```

```python
def adjoin(s, v):
    if s is BTree.empty:
        return BTree(v)
    elif s.root == v:
        return s
    elif s.root < v:
        return BTree(s.root, s.left, adjoin(s.right, v))
    elif s.root > v:
        return BTree(s.root, adjoin(s.left, v), s.right)
```

however, adjoin doesn't guarantee a result as a balanced tree

```
>>> t = BTree.empty
>>> for k in range(20):
...     t = adjoin(t, k)
...
>>> t
BTree(0, BTree.empty, BTree(1, BTree.empty, BTree(2, BTree.empty, BTree(3, BTree.empty, BTree(4, BTree.empty, BTree(5, BTree.empty, BTree(6, BTree.empty, BTree(7, BTree.empty, BTree(8, BTree.empty, BTree(9, BTree.empty, BTree(10, BTree.empty, BTree(11, BTree.empty, BTree(12, BTree.empty, BTree(13, BTree.empty, BTree(14, BTree.empty, BTree(15, BTree.empty, BTree(16, BTree.empty, BTree(17, BTree.empty, BTree(18, BTree.empty, BTree(19))))))))))))))))))))
>>> print(t)
```

```
>>> print(t)
0
  None
  1
    None
    2
      None
      3
        None
        4
          None
          5
            None
            6
              None
              7
                None
                8
                  None
                  9
                    None
                    10
                      None
                      11
```

Terminal Shell Edit View Window Help

lec — Python -i ex.py — 52×27

~/lec — Python -i ex.py

```
~/lec$ python3 -i ex.py
>>> odds = [2*n+1 for n in range(6)]
>>> odds
[1, 3, 5, 7, 9, 11]
>>> t = bst(odds)
>>> t
BTree(7, BTree(3, BTree(1), BTree(5)), BTree(11, BTr
ee(9)))
>>> adjoin(t, 8)
BTree(7, BTree(3, BTree(1), BTree(5)), BTree(11, BTr
ee(9, BTree(8))))
>>> adjoin(t, 3)
BTree(7, BTree(3, BTree(1), BTree(5)), BTree(11, BTr
ee(9)))
```

ex.py (~/Documents/workspace/61a_lectures/

```
def adjoin(s, v):
    if s is BTree.empty:
        return BTree(v)
    elif s.root == v:
        return s
    elif s.root < v:
        return BTree(s.root, s.left, adjoin(s.right, v))
    elif s.root > v:
        return BTree(s.root, adjoin(s.left, v), s.right)
```

one solution is to randomize the order when adjoining elements

```
>>> s = list(range(20))
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1
5, 16, 17, 18, 19]
>>> from random import shuffle
>>> shuffle(s)
>>> s
[5, 11, 14, 19, 0, 10, 6, 16, 13, 2, 9, 3, 1, 17, 7,
 4, 15, 18, 12, 8]
>>> t = BTree.empty
>>> for k in s:
...     t = adjoin(t, k)
...
>>> t
BTree(5, BTree(0, BTree.empty, BTree(2, BTree(1), BT
ree(3, BTree.empty, BTree(4)))), BTree(11, BTree(10,
 BTree(6, BTree.empty, BTree(9, BTree(7, BTree.empty
, BTree(8))))), BTree(14, BTree(13, BTree(12)), BTre
e(19, BTree(16, BTree(15), BTree(17, BTree.empty, BT
ree(18))))))
```

```
>>> print(t)
5
  0
    None
    2
      1
        None
        None
      3
        None
        4
          None
          None
  11
    10
      6
        None
        9
          7
            None
            8
              None
```

how to keep a binary search tree balanced is an interesting topic in computer science