Announcements

- Project 3 due Thursday 3/12 @ 11:59pm
Announcements

- Project 3 due Thursday 3/12 @ 11:59pm
- Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
Announcements

- Project 3 due Thursday 3/12 @ 11:59pm
  - Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11
Announcements

• Project 3 due Thursday 3/12 @ 11:59pm
  ▪ Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  ▪ Bonus point for early submission by Wednesday 3/11
• Guerrilla section this weekend (details announced soon)
Announcements

- Project 3 due Thursday 3/12 @ 11:59pm
  - Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11
- Guerrilla section this weekend (details announced soon)
- Homework 6 due Monday 3/16 @ 11:59pm
Announcements

• Project 3 due Thursday 3/12 @ 11:59pm
  ▪ Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  ▪ Bonus point for early submission by Wednesday 3/11

• Guerrilla section this weekend (details announced soon)

• Homework 6 due Monday 3/16 @ 11:59pm

• Midterm 2 is on Thursday 3/19 7pm–9pm
Announcements

• Project 3 due Thursday 3/12 @ 11:59pm
  ▪ Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  ▪ Bonus point for early submission by Wednesday 3/11

• Guerrilla section this weekend (details announced soon)

• Homework 6 due Monday 3/16 @ 11:59pm

• Midterm 2 is on Thursday 3/19 7pm–9pm
  ▪ Fill out conflict form if you cannot attend due to a course conflict
Measuring Efficiency
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

```
fib(5)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 2) + fib(n - 1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

[Diagram of recursive computation]

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

[Diagram of recursive tree for calculating Fibonacci numbers]

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

![Fibonacci Tree](http://en.wikipedia.org/wiki/File:Fibonacci.jpg)
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

Memoization

Idea: Remember the results that have been computed before
Memoization

**Idea:** Remember the results that have been computed before

```
def memo(f):
```
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
```
Memoization

Idea: Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
```

```python
def memo(f):
    cache = {}
    def memoized(n):
```
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
```
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
```

Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
```
**Memoization**

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```
Memoization

Idea: Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```
**Memoization**

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}

def memoized(n):
    if n not in cache:
        cache[n] = f(n)
    return cache[n]
return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

- Keys are arguments that map to return values
- Same behavior as f, if f is a pure function

(Demo)
Memoized Tree Recursion

```
fib(5)
  / 
  |  
  fib(3) fib(4)
  /   /   
  |  |  |  
  fib(1) fib(2) fib(3)
  /   /  
  |  |   
  fib(0) fib(1) fib(0) fib(1)
  |    |    |    
  0    1   0    1
```
Memoized Tree Recursion

Call to \textit{fib}
Memoized Tree Recursion

Call to fib

Found in cache
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to \texttt{fib}  
Found in cache  
Skipped
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

```
Call to fib
- Found in cache
- Skipped

fib(5)
  fib(4)
  fib(3)
    fib(2)
      fib(1)
        fib(0)
          0 1
          fib(1)
            1
            fib(0)
              0 1
              fib(1)
                1
                fib(0)
                  0 1
                  fib(1)
                    1
                    fib(0)
                      0 1
                      fib(1)
                        1
```
Memoized Tree Recursion

Call to $\text{fib}$
- Found in cache
- Skipped
Memoized Tree Recursion

**Call to fib**

**Found in cache**

**Skipped**
Memoized Tree Recursion

- Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped

fib(5)

- fib(3)
  - fib(1)
    - fib(0)
      - 0
      - 1
  - fib(2)
    - fib(0)
      - 0
      - 1
    - fib(1)
      - 1

- fib(4)
  - fib(2)
    - fib(0)
      - 0
      - 1
    - fib(1)
      - 1
  - fib(3)
    - fib(1)
      - fib(0)
        - 0
        - 1
      - fib(1)
        - 1
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

fib(5) → fib(3) → fib(1) → fib(0) → fib(1)

- Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped

```
fib(5)
fib(3)
fib(1)  fib(2)
fib(0)  1
 0
fib(4)
fib(2)
fib(0)  fib(1)
 0  1
fib(3)
fib(1)
fib(0)  fib(1)
 1
fib(2)
 0  1
fib(1)  fib(2)
 0  1
```
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

- Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to fib

Found in cache

Skipped
Memoized Tree Recursion

Call to \texttt{fib} \hspace{1cm} \textcolor{red}{\text{Found in cache}} \hspace{1cm} \textcolor{orange}{\text{Skipped}}

\begin{itemize}
    \item \texttt{fib}(5)
    \item \texttt{fib}(3)
    \item \texttt{fib}(1)
    \item \texttt{fib}(0)
    \item \texttt{fib}(2)
    \item \texttt{fib}(4)
    \item \texttt{fib}(3)
    \item \texttt{fib}(1)
    \item \texttt{fib}(0)
    \item \texttt{fib}(1)
    \item \texttt{fib}(2)
    \item \texttt{fib}(3)
    \item \texttt{fib}(2)
    \item \texttt{fib}(1)
    \item \texttt{fib}(0)
    \item \texttt{fib}(1)
    \item \texttt{fib}(2)
    \item \texttt{fib}(3)
    \item \texttt{fib}(2)
\end{itemize}

- 1
- 1
- 0
- 1
- 0
- 1
- 0
- 1

- 1
- 1
- 0
- 1
- 0
- 1
Tree Class
Tree Class

A Tree has an entry (any value) at its root and a list of branches
Tree Class

A Tree has an entry (any value) at its root and a list of branches

class Tree:
A Tree has an entry (any value) at its root and a list of branches.

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

A Tree has an entry (any value) at its root and a list of branches.

class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
**Tree Class**

A Tree has an entry (any value) at its root and a list of branches

```python
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
```

A Tree has an entry (any value) at its root and a list of branches.
Tree Class

A Tree has an entry (any value) at its root and a list of branches

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

Built-in `isinstance` function: returns True if `branch` has a class that is or `inherits from` `Tree`
A Tree has an entry (any value) at its root and a list of branches

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

Built-in `isinstance` function: returns True if `branch` has a class that is or inherits from `Tree`
Tree Class

A Tree has an entry (any value) at its root and a list of branches

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

def fib_tree(n):
    Built-in `isinstance` function: returns True if `branch` has a class that is or inherits from `Tree`
**Tree Class**

A Tree has an entry (any value) at its root and a list of branches

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

def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
```

Built-in `isinstance` function: returns True if `branch` has a class that is or inherits from `Tree`
A Tree has an entry (any value) at its root and a list of branches

class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
A Tree has an entry (any value) at its root and a list of branches.

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

def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        return Tree(left.entry + right.entry, (left, right))
```

Built-in `isinstance` function: returns True if `branch` has a class that is or inherits from `Tree`
Tree Class

A Tree has an entry (any value) at its root and a list of branches.

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

def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        return Tree(left.entry + right.entry, (left, right))

(Demo)
Hailstone Trees
Hailstone Trees
Hailstone Trees

Pick a positive integer \( n \) as the start
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1
Hailstone Trees

Pick a positive integer \( n \) as the start
If \( n \) is even, divide it by 2
If \( n \) is odd, multiply it by 3 and add 1
Continue this process until \( n \) is 1
Hailstone Trees

Pick a positive integer n as the start
If n is even, divide it by 2
If n is odd, multiply it by 3 and add 1
Continue this process until n is 1
Hailstone Trees

Pick a positive integer n as the start
If n is even, divide it by 2
If n is odd, multiply it by 3 and add 1
Continue this process until n is 1

1
2
4
8
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2

If $n$ is odd, multiply it by 3 and add 1

Continue this process until $n$ is 1

1

2

4

8

16
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1

1
2
4
8
16
32
Hailstone Trees

Pick a positive integer n as the start
If n is even, divide it by 2
If n is odd, multiply it by 3 and add 1
Continue this process until n is 1

1
2
4
8
16
32
64
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1

1
2
4
8
16
32
64
128
### Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1

Continue this process until $n$ is 1
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1
Hailstone Trees

Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1

\[
\begin{align*}
1 & \quad 2 & \quad 4 & \quad 8 & \quad 16 & \quad 32 & \quad 64 & \quad 128 & \quad 21 & \quad 20 & \quad 3 \\
5 & \quad 10 & \quad 3
\end{align*}
\]
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2

If $n$ is odd, multiply it by 3 and add 1

Continue this process until $n$ is 1

All possible $n$ that start a length-8 hailstone sequence
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1

```python
def hailstone_tree(k, n=1):
    """Return a Tree in which the paths from the leaves to the root are all possible hailstone sequences of length \( k \) ending in \( n \)."""
```

All possible \( n \) that start a length-8 hailstone sequence
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1

```python
def hailstone_tree(k, n=1):
    """Return a Tree in which the paths from the leaves to the root are all possible hailstone sequences of length $k$ ending in $n$."""
```

All possible $n$ that start a length-8 hailstone sequence

(Demo)
Binary Tree Class
Binary Tree Class
Binary Tree Class

A binary tree is a tree that has a left branch and a right branch.
Binary Tree Class

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

```
            3
           /|
          / | 
         1  7
           / |
          /   |
         5   9
            /|
           / | 
          E   E
             |
             11
```

E: An empty tree
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 BinaryTree always has exactly two branches

![Binary Tree Diagram](image-url)
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 BinaryTree always has exactly two branches.

```
3
/   \
1   7
/     \
E     E
/     /   \
5     9   E
/   /     \
E   E     E
/   /       \
E   11       E
/           /\
E           E
```
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 BinaryTree always has exactly two branches.

```
class BinaryTree(Tree):
```

```
3
/\  
1 7
/ \  
E E
5 9
/ \ /  
E E E
```

E: An empty tree
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 BinaryTree always has exactly two branches.

```python
class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True
```

![Binary Tree Diagram]

E: An empty tree
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 BinaryTree always has exactly two branches.

```
class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

    def __init__(self, entry, left=empty, right=empty):
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False

E: An empty tree
```

```
3
  /   \
 1     7
 /     /
E      E
  /     /
 5     9
 /     /
E      E
   /     /
  E      E
   /     /
  E      E
```
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 BinaryTree always has exactly two branches.

```
class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

    def __init__(self, entry, left=empty, right=empty):
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False

    @property
    def left(self):
        return self.branches[0]
```
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 BinaryTree always has *exactly* two branches.

class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

def __init__(self, entry, left=empty, right=empty):
    Tree.__init__(self, entry, (left, right))
    self.is_empty = False

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

@property
def right(self):
    return self.branches[1]
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 BinaryTree always has exactly two branches

```python
class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

def __init__(self, entry, left=empty, right=empty):
    Tree.__init__(self, entry, (left, right))
    self.is_empty = False

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

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

Bin = BinaryTree
```

E: An empty tree

```
    3
   / \
  1   7
 / \ /
E E 5 9
/ \ / /
E E E 11
```

```
    E E E
   / \ \\
  E E E
```
**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 BinaryTree always has exactly two branches.

```python
class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

    def __init__(self, entry, left=empty, right=empty):
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False

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

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

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

E: An empty tree

```
3
  /\  
1  7
 / \ / \  
E  E 5  9
 / \ / \  
E  E  E  
  / \  
  E  E
```
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 BinaryTree always has exactly two branches.

```python
class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

    def __init__(self, entry, left=empty, right=empty):
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False

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

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

```python
Bin = BinaryTree

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

(Demo)