Announcements

- HW8 due tonight at 7pm

- Midterm 2 Thursday at 7pm
  - See course website for more information
Tree Structured Data

Nested Sequences are Hierarchical Structures.

$((1, 2), (3, 4), 5)$

In every tree, a vast forest

Example: [http://goo.gl/0h6n5](http://goo.gl/0h6n5)
Recursive Tree Processing

Tree operations typically make recursive calls on branches

```python
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))

def map_tree(tree, fn):
    if type(tree) != tuple:
        return fn(tree)
    return tuple(map_tree(branch, fn)
                for branch in tree)
```
Trees with Internal Node Values

Trees can have values at internal nodes as well as their leaves.
Trees with Internal Node Values

Trees can have values at internal nodes as well as their leaves.

class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

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

Tree recursive functions can compute the same thing many times

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
Memoized Tree Recursion

\[ \text{fib}(35) \]

Calls to \texttt{fib} with memoization: \hspace{1cm} 35

Calls to \texttt{fib} without memoization: \hspace{1cm} 18,454,929
Iterative, recursive, and memoized implementations are not the same.

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>$\Theta(\phi^n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>
Sets

One more built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets are unordered, just like dictionary entries

```python
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}
>>> 3 in s
True
>>> len(s)
4
>>> s.union({1, 5})
{1, 2, 3, 4, 5}
>>> s.intersection({6, 5, 4, 3})
{3, 4}
```
Implementing Sets

What we should be able to do with a set:

- Membership testing: Is a value an element of a set?
- Union: Return a set with all elements in \textit{set1 or set2}
- Intersection: Return a set with any elements in \textit{set1 and set2}
- Adjunction: Return a set with all elements in \textit{s} and a value \textit{v}

\begin{align*}
\text{Union} & : 1 2 3 4 \\
\text{Intersection} & : 1 3 2 5 \\
\text{Adjunction} & : 1 2 3 4
\end{align*}
Sets as Unordered Sequences

Proposal 1: A set is represented by a recursive list that contains no duplicate items

This is how we implemented dictionaries

```python
def empty(s):
    return s is Rlist.empty

def set_contains(s, v):
    if empty(s):
        return False
    elif s.first == v:
        return True
    return set_contains(s.rest, v)
```
Sets as Unordered Sequences

```python
def adjoin_set(s, v):
    if set.contains(s, v):
        return s
    return Rlist(v, s)

def intersect_set(set1, set2):
    f = lambda v: set.contains(set2, v)
    return filter_rlist(set1, f)

def union_set(set1, set2):
    f = lambda v: not set.contains(set2, v)
    set1_not_set2 = filter_rlist(set1, f)
    return extend_rlist(set1_not_set2, set2)
```

Time order of growth

- \( \Theta(n) \) - The size of the set
- \( \Theta(n^2) \) - Assume sets are the same size