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
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 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.

```
    fib(6)
   /     \
  fib(4)  fib(5)
 /     /     /
fib(2) fib(3) fib(3)
   /     /     /
  1 fib(1) fib(1) fib(1)
 /     /     /
0 fib(2) fib(2) fib(2)
   /     /     /
  0 1 1 1
 /     /
0 fib(1) fib(1)
 /     /
1 fib(2) fib(2)
 /     /
1 1
 /     /
0 fib(3) fib(3)
 /     /
0 fib(4) fib(4)
 /     /
0 0
 /     /
0 0
 /     /
0 1
```
Trees with Internal Node Values

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

class Tree(object):
Trees can have values at internal nodes as well as their leaves.

class Tree(object):
    def __init__(self, entry, left=None, right=None):
Trees with Internal Node Values

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

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

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

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

```python
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)
```
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
Memoization

Tree recursive functions can compute the same thing many times
Memoization

Tree recursive functions can compute the same thing many times.

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

Tree recursive functions can compute the same thing many times

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

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

\[
\text{fib(6)}
\]

\[
\text{fib(4)}
\]

\[
\text{fib(2)} \quad \text{fib(3)}
\]

\[
1 \quad \text{fib(1)} \quad \text{fib(2)}
\]

\[
0 \quad 1
\]

\[
\text{fib(5)}
\]

\[
\text{fib(3)}
\]

\[
\text{fib(2)}
\]

\[
1 \quad \text{fib(1)} \quad \text{fib(2)}
\]

\[
0 \quad 1
\]

\[
\text{fib(4)}
\]

\[
\text{fib(3)}
\]

\[
\text{fib(2)}
\]

\[
1 \quad \text{fib(1)} \quad \text{fib(2)}
\]

\[
0 \quad 1 \quad 0 \quad 1
\]
Memoized Tree Recursion

Call to \texttt{fib}

\begin{itemize}
  \item \texttt{fib(6)}
  \item \texttt{fib(4)}
    \begin{itemize}
      \item \texttt{fib(2)}
      \begin{itemize}
        \item 1
        \item \texttt{fib(1)}
          \begin{itemize}
            \item 0
          \end{itemize}
        \item \texttt{fib(2)}
          \begin{itemize}
            \item 1
          \end{itemize}
      \end{itemize}
      \item \texttt{fib(3)}
        \begin{itemize}
          \item \texttt{fib(1)}
            \begin{itemize}
              \item 0
            \end{itemize}
          \item \texttt{fib(2)}
            \begin{itemize}
              \item 1
            \end{itemize}
        \end{itemize}
    \end{itemize}
  \item \texttt{fib(5)}
    \begin{itemize}
      \item \texttt{fib(3)}
        \begin{itemize}
          \item \texttt{fib(1)}
            \begin{itemize}
              \item 0
            \end{itemize}
          \item \texttt{fib(2)}
            \begin{itemize}
              \item 1
            \end{itemize}
        \end{itemize}
      \item \texttt{fib(4)}
        \begin{itemize}
          \item 1
          \item \texttt{fib(1)}
            \begin{itemize}
              \item 0
            \end{itemize}
          \item \texttt{fib(2)}
            \begin{itemize}
              \item 1
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}
Memoized Tree Recursion

```
fib(6)
  fib(4)  fib(5)
    fib(2)  fib(3)  fib(3)  fib(4)
      fib(1) fib(2) fib(1) fib(2)
```

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

Call to fib
Found in cache

fib(6) → fib(4) → fib(2) → fib(1)
fib(5) → fib(3) → fib(2) → fib(1)

fib(2) → fib(1)
fib(3) → fib(2) → fib(1)

0 1 1 0 1 1 0 1 1 0 1
Memoized Tree Recursion

Call to \texttt{fib}

Found in cache
Memoized Tree Recursion

Call to $\text{fib}$

Found in cache

$\text{fib}(6)$

$\text{fib}(4)$

$\text{fib}(2)$ $\text{fib}(3)$

$\text{fib}(1)$ $\text{fib}(2)$

$\text{fib}(5)$

$\text{fib}(3)$

$\text{fib}(1)$ $\text{fib}(2)$

$\text{fib}(4)$

$\text{fib}(2)$ $\text{fib}(3)$

$\text{fib}(1)$ $\text{fib}(2)$
Memoized Tree Recursion

Call to fib
• Found in cache
Memoized Tree Recursion

- $\text{fib}(6)$
- $\text{fib}(5)$
- $\text{fib}(4)$
- $\text{fib}(3)$
- $\text{fib}(2)$
- $\text{fib}(1)$

- $\text{fib}(6)$
  - $\text{fib}(4)$
    - $\text{fib}(2)$
      - $\text{fib}(1)$
      - $\text{fib}(2)$
    - $\text{fib}(3)$
  - $\text{fib}(5)$
    - $\text{fib}(4)$
      - $\text{fib}(3)$
      - $\text{fib}(2)$
      - $\text{fib}(1)$
      - $\text{fib}(2)$

- $\text{fib}(6)$
- Call to $\text{fib}$
- Found in cache
Memoized Tree Recursion

Call to fib
- Found in cache

```
fib(6)
/   
|    |
sib(4) sib(5)
/   /   /
|   |   |
sib(2) sib(3) sib(1)
/ 
1 0 1
```

```python
fib(6)
/   
|    |
sib(4) sib(5)
/   /   /
|   |   |
sib(2) sib(3) sib(1)
/ 
1 0 1
```
Memoized Tree Recursion

- **fib(6)**
  - **fib(4)**
    - **fib(2)**
      - **fib(1)**: 0
      - **fib(2)**: 1
    - **fib(3)**
  - **fib(5)**
    - **fib(3)**
      - **fib(1)**: 0
      - **fib(2)**: 1
    - **fib(2)**
  - **fib(4)**
    - **fib(2)**
      - **fib(1)**: 0
      - **fib(2)**: 1
    - **fib(3)**
  - **fib(5)**
    - **fib(3)**
      - **fib(1)**: 0
      - **fib(2)**: 1
    - **fib(2)**

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

- fib(6)
  - fib(4)
    - fib(2)
      - fib(1)
      - fib(2)
    - fib(3)
      - fib(1) fib(2)
  - fib(5)
    - fib(3)
      - fib(1) fib(2)
    - fib(4)
      - fib(2)
      - fib(3)
Memoized Tree Recursion

Call to `fib`
- Found in cache

```
fib(6)  
  /   
fib(4)  fib(5)
  /   
fib(2) fib(3)  
  /   
0   fib(1) fib(2)
  /   
1   0 1
```
Memoized Tree Recursion

$\text{fib}(35)$

<table>
<thead>
<tr>
<th>Calls to $\text{fib}$ with memoization:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls to $\text{fib}$ without memoization:</td>
</tr>
</tbody>
</table>
Memoized Tree Recursion

Call to \texttt{fib}

\begin{itemize}
\item fib(6)
\item fib(5)
\item fib(4)
\item fib(3)
\item fib(2)
\item fib(1)
\end{itemize}

\begin{itemize}
\item fib(3)
\item fib(2)
\item fib(1)
\end{itemize}

\begin{itemize}
\item fib(35)
\end{itemize}

\textbf{Calls to \texttt{fib} with memoization:}

\textbf{35}

\textbf{Calls to \texttt{fib} without memoization:}
Memoized Tree Recursion

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

Calls to \texttt{fib} with memoization: \texttt{35}

Calls to \texttt{fib} without memoization: \texttt{18,454,929}
Orders of Growth

Iterative, recursive, and memoized implementations are not the same.

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

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

fib = memo(fib)
```
Orders of Growth

Iterative, recursive, and memoized implementations are not the same.

\[
\begin{array}{ll}
\text{Time} & \Theta(n) \\
\end{array}
\]

```python
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

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

fib = memo(fib)
```
Orders of Growth

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>
</tbody>
</table>

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

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

fib = memo(fib)
```
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>
</tbody>
</table>

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

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

fib = memo(fib)
```
Orders of Growth

Iterative, recursive, and memoized implementations are not the same.

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

def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

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

fib = memo(fib)
Orders of Growth

Iterative, recursive, and memoized implementations are not the same.

\[
\begin{array}{ccc}
\text{Time} & \text{Space} \\
\Theta(n) & \Theta(1) \\
\Theta(\phi^n) & \Theta(n) \\
\Theta(n) & \\
\end{array}
\]

```python
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

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

fib = memo(fib)
```
Orders of Growth

Iterative, recursive, and memoized implementations are not the same.

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

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

fib = memo(fib)
```

<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
Sets

One more built-in Python container type
Sets

One more built-in Python container type

• Set literals are enclosed in braces
Sets

One more built-in Python container type

• Set literals are enclosed in braces
• Duplicate elements are removed on construction
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
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}
```
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
```
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
```
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)
```
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
```
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})
```
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}
```
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})
```
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
Implementing Sets

What we should be able to do with a set:
Implementing Sets

What we should be able to do with a set:

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

Union

\[
\begin{array}{|c|c|}
\hline
1 & 2 \\
4 & 5 \\
\hline
\end{array}
\begin{array}{|c|c|}
\hline
3 & 3 \\
\hline
\end{array}
\]
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 set1 or set2
- Intersection: Return a set with any elements in set1 and set2
- Adjunction: Return a set with all elements in s and a value v

<table>
<thead>
<tr>
<th>Union</th>
<th>Intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 4 3</td>
<td>1 2 4 5 3</td>
</tr>
<tr>
<td>2 3</td>
<td>1 3 2 5 3</td>
</tr>
</tbody>
</table>

Union

Intersection
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 set1 or set2
• Intersection: Return a set with any elements in set1 and set2
• Adjunction: Return a set with all elements in s and a value v

Union

\[
\begin{array}{ccc}
1 & 2 & \\ \\
4 & 5 & 3
\end{array}
\quad \Rightarrow \quad
\begin{array}{ccc}
1 & 2 & \\ \\
4 & 5 & 3
\end{array}
\]

Intersection

\[
\begin{array}{ccc}
1 & 2 & \\ \\
4 & 5 & 3
\end{array}
\quad \Rightarrow \quad
\begin{array}{ccc}
1 & 2 & \\ \\
4 & 5 & 3
\end{array}
\]

Adjunction

\[
\begin{array}{ccc}
1 & 2 & \\ \\
4 & 3 & \\ \\
\end{array}
\quad \Rightarrow \quad
\begin{array}{ccc}
1 & 2 & \\ \\
4 & 3 & \\ \\
\end{array}
\]

\[
\begin{array}{ccc}
1 & 3 & \\ \\
4 & 2 & \\ \\
\end{array}
\]
Sets as Unordered Sequences

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

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

def set_contains(s, v):
    if empty(s):
        return False
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
    else:
        return set_contains(s.tail, v)
```
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
```
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
def adjoin_set(s, v):
Sets as Unordered Sequences

```python
def adjoin_set(s, v):
    if set_contains(s, v):
```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
Sets as Unordered Sequences

def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)

Time order of growth
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
Sets as Unordered Sequences

```python
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

Time order of growth

$\Theta(n)$

The size of the set
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):
```

Time order of growth

$\Theta(n)$

The size of the set
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)
```

Time order of growth

$\Theta(n)$

The size of the set
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)
```

Time order of growth

\( \Theta(n) \)

The size of the set
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)
```

Time order of growth

- $\Theta(n)$
- $\Theta(n^2)$

The size of the set
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)
```

Time order of growth

\[ \Theta(n) \]

The size of the set

\[ \Theta(n^2) \]

Assume sets are the same size
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):
```

Time order of growth

\[ \Theta(n) \]

The size of the set

\[ \Theta(n^2) \]

Assume sets are the same size
Sets as Unordered Sequences

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)

Time order of growth

$\Theta(n)$

The size of the set

$\Theta(n^2)$

Assume sets are the same size
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)
```

Time order of growth

\[ \Theta(n) \]

The size of the set

Assume sets are the same size

\[ \Theta(n^2) \]
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
  - Assume sets are the same size
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

\( \Theta(n^2) \)

The size of the set

Assume sets are the same size

\( \Theta(n^2) \)