Friday, October 18
Announcements

- Homework 6 is due Tuesday 10/22 @ 11:59pm
Announcements

• Homework 6 is due Tuesday 10/22 @ 11:59pm
  ▪ Includes a mid-semester survey about the course so far
Announcements

• Homework 6 is due Tuesday 10/22 @ 11:59pm
  ▪ Includes a mid-semester survey about the course so far
• Project 3 is due Thursday 10/24 @ 11:59pm
Announcements

- Homework 6 is due Tuesday 10/22 @ 11:59pm
  - Includes a mid-semester survey about the course so far
- Project 3 is due Thursday 10/24 @ 11:59pm
- Midterm 2 is on Monday 10/28 7pm–9pm
Announcements

• Homework 6 is due Tuesday 10/22 @ 11:59pm
  ▪ Includes a mid-semester survey about the course so far
• Project 3 is due Thursday 10/24 @ 11:59pm
• Midterm 2 is on Monday 10/28 7pm–9pm
• Guerrilla section 3 this weekend
Announcements

• Homework 6 is due Tuesday 10/22 @ 11:59pm
  ▪ Includes a mid-semester survey about the course so far
• Project 3 is due Thursday 10/24 @ 11:59pm
• Midterm 2 is on Monday 10/28 7pm–9pm
• Guerrilla section 3 this weekend
  ▪ Object-oriented programming, recursion, and recursive data structures
Announcements

• Homework 6 is due Tuesday 10/22 @ 11:59pm
  ▪ Includes a mid-semester survey about the course so far
• Project 3 is due Thursday 10/24 @ 11:59pm
• Midterm 2 is on Monday 10/28 7pm–9pm
• Guerrilla section 3 this weekend
  ▪ Object-oriented programming, recursion, and recursive data structures
  ▪ 2pm–5pm on Saturday and 10am–1pm on Sunday
Announcements

• Homework 6 is due Tuesday 10/22 @ 11:59pm
  ▪ Includes a mid-semester survey about the course so far
• Project 3 is due Thursday 10/24 @ 11:59pm
• Midterm 2 is on Monday 10/28 7pm–9pm
• Guerrilla section 3 this weekend
  ▪ Object-oriented programming, recursion, and recursive data structures
  ▪ 2pm–5pm on Saturday and 10am–1pm on Sunday
  ▪ Please let us know you are coming by filling out the Piazza poll
Comparing Orders of Growth
Comparing orders of growth (n is the problem size)
Comparing orders of growth (n is the problem size)

\( \Theta(b^n) \)
Comparing orders of growth (n is the problem size)

\[ \Theta(b^n) \quad \text{Exponential growth! Recursive fib takes} \]
\[ \Theta(\phi^n) \text{ steps, where} \quad \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \]
Comparing orders of growth (n is the problem size)

\[ \Theta(b^n) \] Exponential growth! Recursive fib takes

\[ \Theta(\phi^n) \] steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)

Incrementing the problem scales \( R(n) \) by a factor.
Comparing orders of growth (n is the problem size)

\( \Theta(b^n) \) Exponential growth! Recursive fib takes \( \Theta(\phi^n) \) steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)

Incrementing the problem scales R(n) by a factor.

\( \Theta(n^2) \)
Comparing orders of growth (n is the problem size)

- \( \Theta(b^n) \)  Exponential growth! Recursive fib takes \( \Theta(\phi^n) \) steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)
  Incrementing the problem scales \( R(n) \) by a factor.

- \( \Theta(n^2) \)  Quadratic growth. E.g., operations on all pairs.
Comparing orders of growth (n is the problem size)

\( \Theta(b^n) \)  Exponential growth! Recursive fib takes
\[ \Theta(\phi^n) \] steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)
Incrementing the problem scales \( R(n) \) by a factor.

\( \Theta(n^2) \) Quadratic growth. E.g., operations on all pairs.
Incrementing \( n \) increases \( R(n) \) by the problem size \( n \).
Comparing orders of growth ($n$ is the problem size)

$\Theta(b^n)$  Exponential growth!  Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales $R(n)$ by a factor.

$\Theta(n^2)$  Quadratic growth.  E.g., operations on all pairs.

Incrementing $n$ increases $R(n)$ by the problem size $n$.

$\Theta(n)$
Comparing orders of growth (n is the problem size)

\[ \Theta(b^n) \] Exponential growth! Recursive fib takes \[ \Theta(\phi^n) \] steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)

Incrementing the problem scales \( R(n) \) by a factor.

\[ \Theta(n^2) \] Quadratic growth. E.g., operations on all pairs.

Incrementing \( n \) increases \( R(n) \) by the problem size \( n \).

\[ \Theta(n) \] Linear growth. Resources scale with the problem.
Comparing orders of growth (n is the problem size)

\( \Theta(b^n) \)  Exponential growth! Recursive fib takes 
\( \Theta(\phi^n) \) steps, where  \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)
Incrementing the problem scales \( R(n) \) by a factor.

\( \Theta(n^2) \)  Quadratic growth. E.g., operations on all pairs.
Incrementing \( n \) increases \( R(n) \) by the problem size \( n \).

\( \Theta(n) \)  Linear growth. Resources scale with the problem.

\( \Theta(\log n) \)
Comparing orders of growth (n is the problem size)

$\Theta(b^n)$  
Exponential growth! Recursive fib takes
$\Theta(\phi^n)$ steps, where $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828$
Incrementing the problem scales $R(n)$ by a factor.

$\Theta(n^2)$  
Quadratic growth. E.g., operations on all pairs.
Incrementing $n$ increases $R(n)$ by the problem size $n$.

$\Theta(n)$  
Linear growth. Resources scale with the problem.

$\Theta(\log n)$  
Logarithmic growth. These processes scale well.
Comparing orders of growth (n is the problem size)

$\Theta(b^n)$  Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales $R(n)$ by a factor.

$\Theta(n^2)$  Quadratic growth. E.g., operations on all pairs.
Incrementing $n$ increases $R(n)$ by the problem size $n$.

$\Theta(n)$  Linear growth. Resources scale with the problem.

$\Theta(\log n)$  Logarithmic growth. These processes scale well.
Doubling the problem only increments $R(n)$. 
Comparing orders of growth (n is the problem size)

\( \Theta(b^n) \)  Exponential growth! Recursive fib takes 
\( \Theta(\phi^n) \) steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)

Incrementing the problem scales R(n) by a factor.

\( \Theta(n^2) \)  Quadratic growth. E.g., operations on all pairs.
Incrementing n increases R(n) by the problem size n.

\( \Theta(n) \)  Linear growth. Resources scale with the problem.

\( \Theta(\log n) \)  Logarithmic growth. These processes scale well.

Doubling the problem only increments R(n).

\( \Theta(1) \)
Comparing orders of growth (n is the problem size)

\( \Theta(b^n) \) Exponential growth! Recursive fib takes
\( \Theta(\phi^n) \) steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)
Incrementing the problem scales R(n) by a factor.

\( \Theta(n^2) \) Quadratic growth. E.g., operations on all pairs.
Incrementing n increases R(n) by the problem size n.

\( \Theta(n) \) Linear growth. Resources scale with the problem.

\( \Theta(\log n) \) Logarithmic growth. These processes scale well.
Doubling the problem only increments R(n).

\( \Theta(1) \) Constant. The problem size doesn't matter.
Comparing orders of growth (n is the problem size)

\[ \Theta(b^n) \] Exponential growth! Recursive fib takes \( \Theta(\phi^n) \) steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)
Incrementing the problem scales \( R(n) \) by a factor.

\[ \Theta(n^2) \] Quadratic growth. E.g., operations on all pairs.
Incrementing \( n \) increases \( R(n) \) by the problem size \( n \).

\[ \Theta(n) \] Linear growth. Resources scale with the problem.

\[ \Theta(\log n) \] Logarithmic growth. These processes scale well.
Doubling the problem only increments \( R(n) \).

\[ \Theta(1) \] Constant. The problem size doesn't matter.
Comparing orders of growth (n is the problem size)

- \( \Theta(b^n) \): Exponential growth! Recursive fib takes \( \Theta(\phi^n) \) steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)
- \( \Theta(n^6) \): Incrementing the problem scales \( R(n) \) by a factor.
- \( \Theta(n^2) \): Quadratic growth. E.g., operations on all pairs. Incrementing \( n \) increases \( R(n) \) by the problem size \( n \).
- \( \Theta(n) \): Linear growth. Resources scale with the problem.
- \( \Theta(\log n) \): Logarithmic growth. These processes scale well. Doubling the problem only increments \( R(n) \).
- \( \Theta(1) \): Constant. The problem size doesn't matter.
Comparing orders of growth (n is the problem size)

\[ \Theta(b^n) \quad \text{Exponential growth! Recursive fib takes} \]
\[ \Theta(\phi^n) \text{ steps, where } \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \]

\[ \Theta(n^6) \quad \text{Incrementing the problem scales } R(n) \text{ by a factor.} \]

\[ \Theta(n^2) \quad \text{Quadratic growth. E.g., operations on all pairs.} \]
\[ \text{Incrementing } n \text{ increases } R(n) \text{ by the problem size } n. \]

\[ \Theta(n) \quad \text{Linear growth. Resources scale with the problem.} \]

\[ \Theta(\sqrt{n}) \quad \text{Logarithmic growth. These processes scale well.} \]
\[ \Theta(\log n) \quad \text{Doubling the problem only increments } R(n). \]

\[ \Theta(1) \quad \text{Constant. The problem size doesn't matter.} \]
Sets
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
True
```
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})
{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})
{3, 4}
```
Implementing Sets
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
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 set1 or set2
- Intersection: Return a set with any elements in set1 and set2

**Union**

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

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**Intersection**

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
</tr>
</thead>
</table>
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
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
Sets as Unordered Sequences
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.

```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.

```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.

```python
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.

```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.

```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:
```

Sets as Unordered Sequences

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

```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.rest, v)
```
Sets as Unordered Sequences

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

```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.rest, v)
```

(Demo)
Review: Order of Growth
Review: Order of Growth

For a set operation that takes "linear" time, we say that
Review: Order of Growth

For a set operation that takes "linear" time, we say that

\[ n: \text{size of the set} \]
Review: Order of Growth

For a set operation that takes "linear" time, we say that

\[ n: \text{size of the set} \]

\[ R(n): \text{number of steps required to perform the operation} \]
Review: Order of Growth

For a set operation that takes "linear" time, we say that

\[ n: \text{size of the set} \]

\[ R(n): \text{number of steps required to perform the operation} \]

\[ R(n) = \Theta(n) \]
Review: Order of Growth

For a set operation that takes "linear" time, we say that

\[ n: \text{size of the set} \]

\[ R(n): \text{number of steps required to perform the operation} \]

\[ R(n) = \Theta(n) \]

An example \( f(n) \)
Review: Order of Growth

For a set operation that takes "linear" time, we say that

\[ n: \text{size of the set} \]

\[ R(n): \text{number of steps required to perform the operation} \]

which means that there are positive constants \( k_1 \) and \( k_2 \) such that

\[ R(n) = \Theta(n) \]

An example f(n)
Review: Order of Growth

For a set operation that takes "linear" time, we say that

\[ n: \text{size of the set} \]
\[ R(n): \text{number of steps required to perform the operation} \]

\[ R(n) = \Theta(n) \]

which means that there are positive constants \( k_1 \) and \( k_2 \) such that

\[ k_1 \cdot n \leq R(n) \leq k_2 \cdot n \]
Review: Order of Growth

For a set operation that takes "linear" time, we say that

- $n$: size of the set
- $R(n)$: number of steps required to perform the operation

$$R(n) = \Theta(n)$$

which means that there are positive constants $k_1$ and $k_2$ such that

$$k_1 \cdot n \leq R(n) \leq k_2 \cdot n$$

for sufficiently large values of $n$. 
Sets as Unordered Sequences
Sets as Unordered Sequences

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

```python
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
    else:
Sets as Unordered Sequences

def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    else:
        return Rlist(v, s)
Sets as Unordered Sequences

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

Time order of growth
Sets as Unordered Sequences

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

Time order of growth

$\Theta(n)$
Sets as Unordered Sequences

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

def intersect_set(set1, set2):
    in_set2 = 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
    else:
        return Rlist(v, s)

def intersect_set(set1, set2):
    in_set2 = lambda v: set_contains(set2, v)
    return filter_rlist(set1, in_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
    else:
        return Rlist(v, s)

def intersect_set(set1, set2):
    in_set2 = lambda v: set_contains(set2, v)
    return filter_rlist(set1, in_set2)
```

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
    else:
        return Rlist(v, s)

def intersect_set(set1, set2):
    in_set2 = lambda v: set_contains(set2, v)
    return filter_rlist(set1, in_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

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

def intersect_set(set1, set2):
    in_set2 = lambda v: set_contains(set2, v)
    return filter_rlist(set1, in_set2)

def union_set(set1, set2):
```

Time order of growth

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

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
    else:
        return Rlist(v, s)

def intersect_set(set1, set2):
    in_set2 = lambda v: set_contains(set2, v)
    return filter_rlist(set1, in_set2)

def union_set(set1, set2):
    not_in_set2 = 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
    else:
        return Rlist(v, s)

def intersect_set(set1, set2):
    in_set2 = lambda v: set_contains(set2, v)
    return filter_rlist(set1, in_set2)

def union_set(set1, set2):
    not_in_set2 = lambda v: not set_contains(set2, v)
    set1_not_set2 = filter_rlist(set1, not_in_set2)
```

Time order of growth

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

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
    else:
        return Rlist(v, s)

def intersect_set(set1, set2):
    in_set2 = lambda v: set_contains(set2, v)
    return filter_rlist(set1, in_set2)

def union_set(set1, set2):
    not_in_set2 = lambda v: not set_contains(set2, v)
    set1_not_set2 = filter_rlist(set1, not_in_set2)
    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
Sets as Unordered Sequences

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

def intersect_set(set1, set2):
in_set2 = lambda v: set_contains(set2, v)
return filter_rlist(set1, in_set2)

def union_set(set1, set2):
    not_in_set2 = lambda v: not set_contains(set2, v)
    set1_not_set2 = filter_rlist(set1, not_in_set2)
    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

\[ \Theta(n^2) \]
Sets as Unordered Sequences

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

def intersect_set(set1, set2):
in_set2 = lambda v: set_contains(set2, v)
return filter_rlist(set1, in_set2)

def union_set(set1, set2):
    not_in_set2 = lambda v: not set_contains(set2, v)
    set1_not_set2 = filter_rlist(set1, not_in_set2)
    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

\[ \Theta(n^2) \]  

(Demo)
Sets as Ordered Sequences
Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest
Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

```python
def set_contains(s, v):
```
Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

```python
def set_contains(s, v):
    if empty(s) or s.first > v:
        return False
```
Sets as Ordered Sequences

**Proposal 2:** A set is represented by a recursive list with unique elements ordered from least to greatest

```python
def set_contains(s, v):
    if empty(s) or s.first > v:
        return False
    elif s.first == v:
        return True
```
Sets as Ordered Sequences

**Proposal 2:** A set is represented by a recursive list with unique elements ordered from least to greatest

```
def set_contains(s, v):
    if empty(s) or s.first > v:
        return False
    elif s.first == v:
        return True
    else:
```


Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

```python
def set_contains(s, v):
    if empty(s) or s.first > v:
        return False
    elif s.first == v:
        return True
    else:
        return set_contains(s.rest, v)
```
Sets as Ordered Sequences

**Proposal 2**: A set is represented by a recursive list with unique elements ordered from least to greatest

```python
def set_contains(s, v):
    if empty(s) or s.first > v:
        return False
    elif s.first == v:
        return True
    else:
        return set_contains(s.rest, v)
```

Order of growth?
Sets as Ordered Sequences

**Proposal 2:** A set is represented by a recursive list with unique elements ordered from least to greatest

```python
def set_contains(s, v):
    if empty(s) or s.first > v:
        return False
    elif s.first == v:
        return True
    else:
        return set_contains(s.rest, v)
```

Order of growth? $\Theta(n)$
Set Intersection Using Ordered Sequences
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set(set1, set2):
```
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
```
Set Intersection Using Ordered Sequences

This algorithm *assumes* that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
```
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    else:
```
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    else:
        e1, e2 = set1.first, set2.first
```
Set Intersection Using Ordered Sequences

This algorithm *assumes* that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    else:
        e1, e2 = set1.first, set2.first
        if e1 == e2:
```
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    else:
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Rlist(e1, intersect_set(set1.rest, set2.rest))
```
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    else:
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Rlist(e1, intersect_set(set1.rest, set2.rest))
        elif e1 < e2:
```
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    else:
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Rlist(e1, intersect_set(set1.rest, set2.rest))
        elif e1 < e2:
            return intersect_set(set1.rest, set2)
```
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    else:
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Rlist(e1, intersect_set(set1.rest, set2.rest))
        elif e1 < e2:
            return intersect_set(set1.rest, set2)
        elif e2 < e1:
```

```python
e1, e2 = set1.first, set2.first
```
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    else:
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Rlist(e1, intersect_set(set1.rest, set2.rest))
        elif e1 < e2:
            return intersect_set(set1.rest, set2)
        elif e2 < e1:
            return intersect_set(set1, set2.rest)
```
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    else:
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Rlist(e1, intersect_set(set1.rest, set2.rest))
        elif e1 < e2:
            return intersect_set(set1.rest, set2)
        elif e2 < e1:
            return intersect_set(set1, set2.rest)
```

(Demo)
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    else:
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Rlist(e1, intersect_set(set1.rest, set2.rest))
        elif e1 < e2:
            return intersect_set(set1.rest, set2)
        else:
            return intersect_set(set1, set2.rest)
```

(Demo)

Order of growth?
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    else:
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Rlist(e1, intersect_set(set1.rest, set2.rest))
        elif e1 < e2:
            return intersect_set(set1.rest, set2)
        elif e2 < e1:
            return intersect_set(set1, set2.rest)

(Demo)

Order of growth? $\Theta(n)$
Sets as Binary Search Trees
Tree Sets
Tree Sets

**Proposal 3:** A set is represented as a Tree. Each entry is:
Proposal 3: A set is represented as a Tree. Each entry is:
- Larger than all entries in its left branch and
**Proposal 3:** A set is represented as a Tree. Each entry is:

- Larger than all entries in its left branch and
- Smaller than all entries in its right branch
Proposal 3: A set is represented as a Tree. Each entry is:
- Larger than all entries in its left branch and
- Smaller than all entries in its right branch
Proposal 3: A set is represented as a Tree. Each entry is:
• Larger than all entries in its left branch and
• Smaller than all entries in its right branch
**Proposal 3:** A set is represented as a Tree. Each entry is:

- Larger than all entries in its left branch and
- Smaller than all entries in its right branch
Membership in Tree Sets
Membership in Tree Sets

Set membership traverses the tree
Membership in Tree Sets

Set membership traverses the tree
• The element is either in the left or right sub-branch
Membership in Tree Sets

Set membership traverses the tree
- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half
Membership in Tree Sets

Set membership traverses the tree
• The element is either in the left or right sub-branch
• By focusing on one branch, we reduce the set by about half

```python
def set_contains(s, v):
```
Membership in Tree Sets

Set membership traverses the tree
- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```python
def set_contains(s, v):
    if s is None:
```


Membership in Tree Sets

Set membership traverses the tree
- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```python
def set_contains(s, v):
    if s is None:
        return False
    return False
```
Membership in Tree Sets

Set membership traverses the tree
- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```python
def set_contains(s, v):
    if s is None:
        return False
    elif s.entry == v:
```
Membership in Tree Sets

Set membership traverses the tree
• The element is either in the left or right sub-branch
• By focusing on one branch, we reduce the set by about half

```python
def set_contains(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
```
Membership in Tree Sets

Set membership traverses the tree
• The element is either in the left or right sub-branch
• By focusing on one branch, we reduce the set by about half

```python
def set_contains(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
```
Membership in Tree Sets

Set membership traverses the tree
- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```python
def set_contains(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains(s.right, v)
```
Membership in Tree Sets

Set membership traverses the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```python
def setContains(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return setContains(s.right, v)
    elif s.entry > v:
```
Membership in Tree Sets

Set membership traverses the tree
- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```python
def set_contains(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains(s.right, v)
    elif s.entry > v:
        return set_contains(s.left, v)
```
Membership in Tree Sets

Set membership traverses the tree
- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```python
def set_contains(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains(s.right, v)
    else:
        return set_contains(s.left, v)
```
Membership in Tree Sets

Set membership traverses the tree
- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```python
def set_contains(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains(s.right, v)
    elif s.entry > v:
        return set_contains(s.left, v)
```
Membership in Tree Sets

Set membership traverses the tree
• The element is either in the left or right sub-branch
• By focusing on one branch, we reduce the set by about half

def set_contains(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains(s.right, v)
    elif s.entry > v:
        return set_contains(s.left, v)

If 9 is in the set, it is in this branch
Membership in Tree Sets

Set membership traverses the tree
• The element is either in the left or right sub-branch
• By focusing on one branch, we reduce the set by about half

```python
def set_contains(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains(s.right, v)
    elif s.entry > v:
        return set_contains(s.left, v)
```

Order of growth?

If 9 is in the set, it is in this branch
Adjoining to a Tree Set
Adjoining to a Tree Set

```
  8
 /  \\
 5   11
/    |
3     9
/     |
1     7
```
Adjoining to a Tree Set

Right!
Adjoining to a Tree Set

Right!
Adjoining to a Tree Set

Right!
Adjoining to a Tree Set

Right!  Left!
Adjoining to a Tree Set

Right!  Left!
Adjoining to a Tree Set

```
      8
     /\   \\
    5 9
   /\  /\  \\
  3 7 9 7 11
 /\  /\  \\
1 7 11
```

Right!  Left!
Adjoining to a Tree Set

```
Right!  Left!  Right!

1 7 11
3 9
5

7 11
9
8

7
None
None
8

19
```
Adjoining to a Tree Set

Right!  Left!  Right!
Adjoining to a Tree Set

```
  8
 / \
5   9
/   /
3   7
 1   11
```

```
  8
 /   \
9     7
|     /\n7   None
 11 None
```

```
  8
 / \
8   None
```

```
  8
 /   \
None None
```

Right!  Left!  Right!  Stop!
Adjoining to a Tree Set

Right!  Left!  Right!  Stop!

1  3  4  5  6  7  8  9  10  11
Adjoining to a Tree Set

```
  8
 / \
5  9
/ \
3 7
```

```
  8
 / \
9  7
/ \
7 11
```

```
  8
 / \
None
```

```
Right!  Left!  Right!  Stop!
```

8
Adjoining to a Tree Set

```
  8
  / \  \
 5   9
 /   /  \
3  7  11
```

```
  8
  / \  \
 9   7
 /   /  \
7  11 None
```

```
  8
  /     \
None   None
```

```
  8
  /     \
7     8
 /     /  \
8     8
```

Right!  Left!  Right!  Stop!
Adjoining to a Tree Set

Right!

Left!

Right!

Stop!
Adjoining to a Tree Set

```
8
/  
5   
   / 
  3   9
 / 
1  7 11
```

```
8
/  
9   
   / 
  7   11
```

```
8
/  
7   
   / 
None  None
```

```
8
```

```
Right!  Left!  Right!  Stop!
```

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

```
8
```

```
9
 /  
7   11
```

```
7
 /  
8   
```

```
8
```

```
8
```

```
7
```

```
8
```

```
19
```
Adjoining to a Tree Set

Right!  Left!  Right!  Stop!

(Demo)
More Set Operations
What Did I Leave Out?

Sets as ordered sequences:
What Did I Leave Out?

Sets as ordered sequences:
• Adjoining an element to a set
What Did I Leave Out?

Sets as ordered sequences:
• Adjoining an element to a set
• Union of two sets
What Did I Leave Out?

Sets as ordered sequences:
- Adjoining an element to a set
- Union of two sets

Sets as binary trees:
What Did I Leave Out?

Sets as ordered sequences:
• Adjoining an element to a set
• Union of two sets

Sets as binary trees:
• Intersection of two sets
What Did I Leave Out?

Sets as ordered sequences:
• Adjoining an element to a set
• Union of two sets

Sets as binary trees:
• Intersection of two sets
• Union of two sets
What Did I Leave Out?

Sets as ordered sequences:
• Adjoining an element to a set
• Union of two sets

Sets as binary trees:
• Intersection of two sets
• Union of two sets
• Balancing a tree
What Did I Leave Out?

Sets as ordered sequences:
• Adjoining an element to a set
• Union of two sets

Sets as binary trees:
• Intersection of two sets
• Union of two sets
• Balancing a tree

That's all on homework 7!