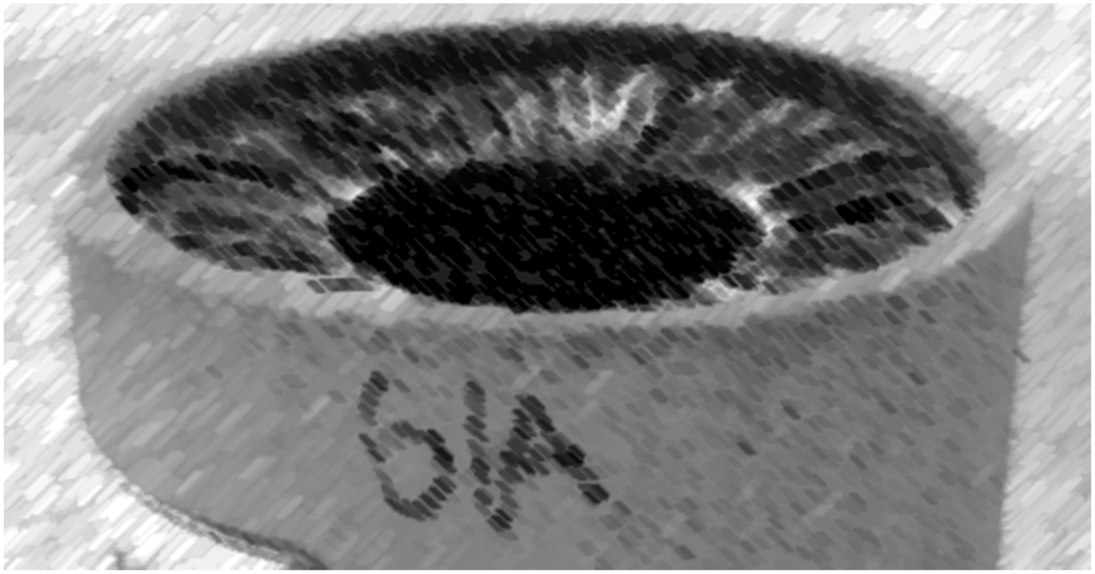


CS61A Lecture 25

Amir Kamil

UC Berkeley

March 20, 2013

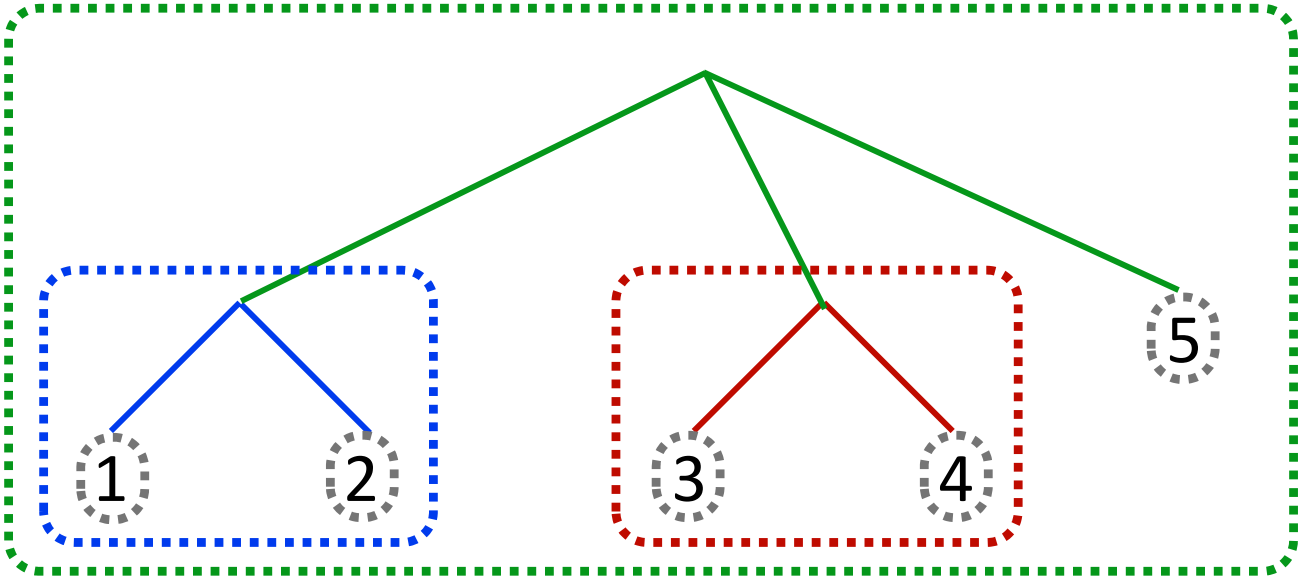
Announcements

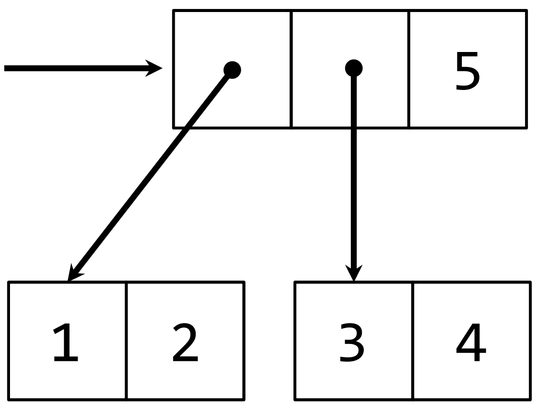
- HW8 due tonight at 7pm
- Midterm 2 Thursday at 7pm
  - See course website for more information

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

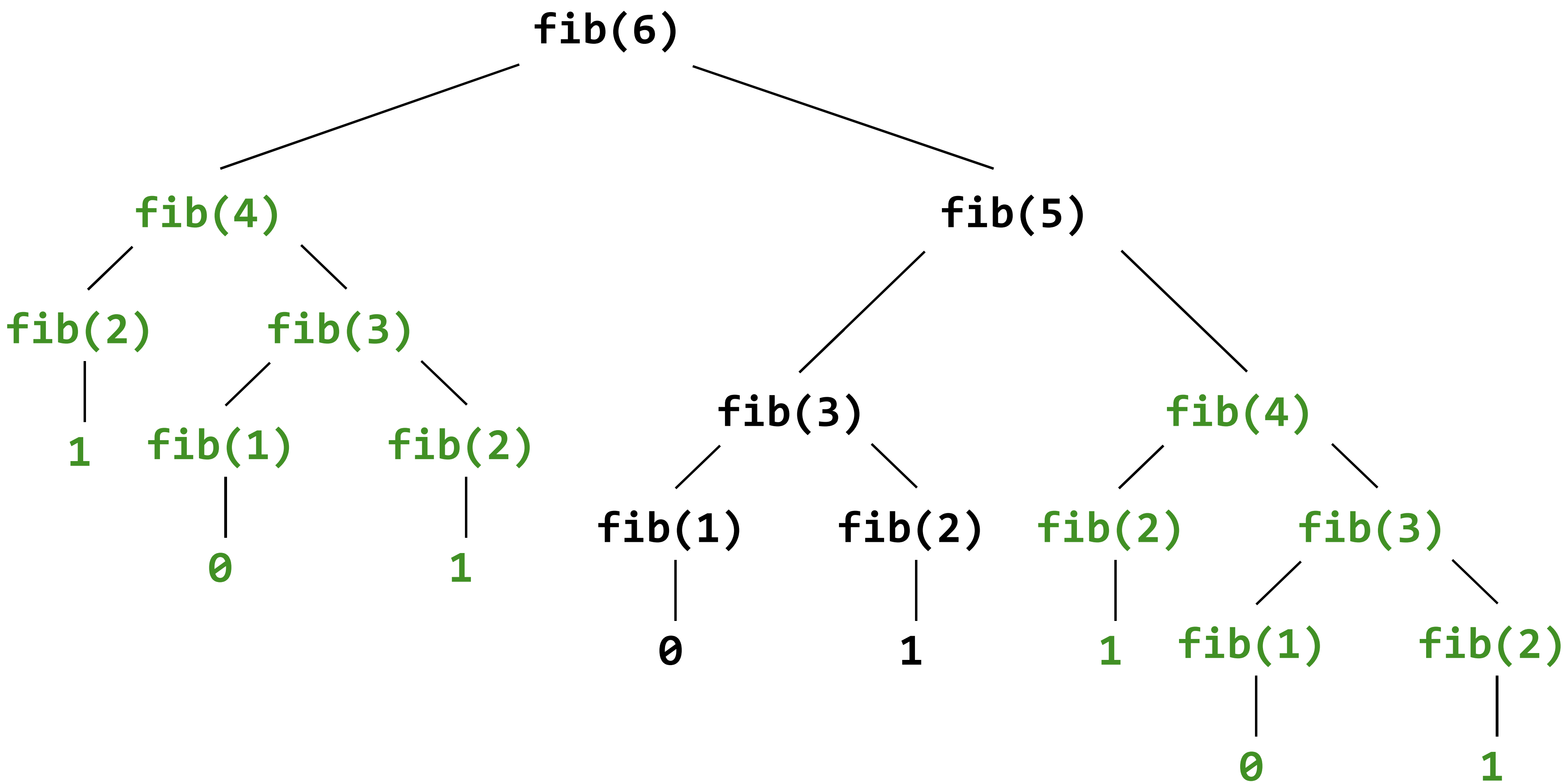
Tree operations typically make recursive calls on branches

```
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

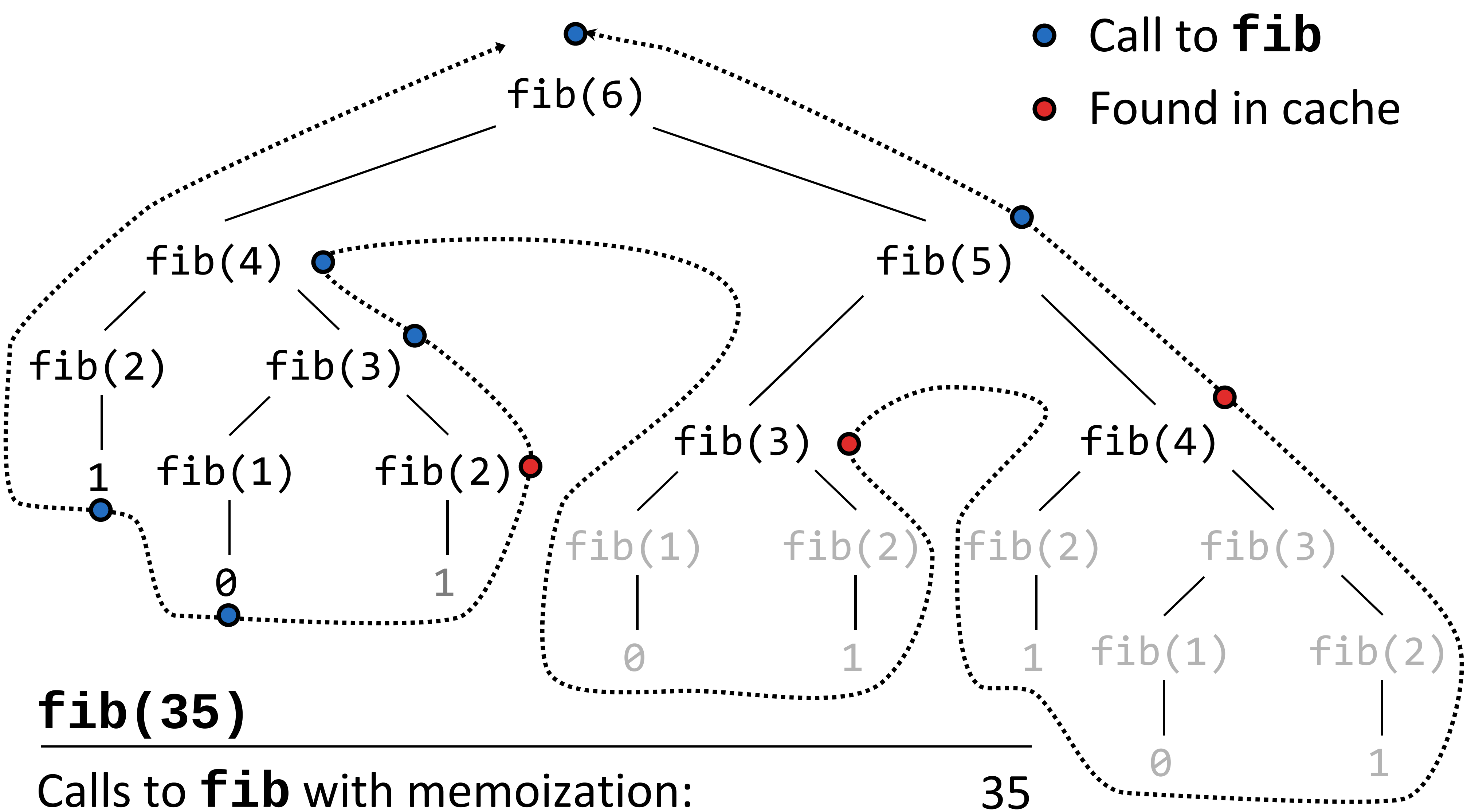
Idea: Remember the results that have been computed before

```
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



fib(35)

Calls to **fib** with memoization:	35
Calls to **fib** without memoization:	18,454,929

Orders of Growth

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

	Time	Space
`fib_iter`	$\Theta(n)$	$\Theta(1)$
`fib`	$\Theta(\phi^n)$	$\Theta(n)$
`fib = memo(fib)`	$\Theta(n)$	$\Theta(n)$

```
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)
```

Sets

One more built-in Python container type

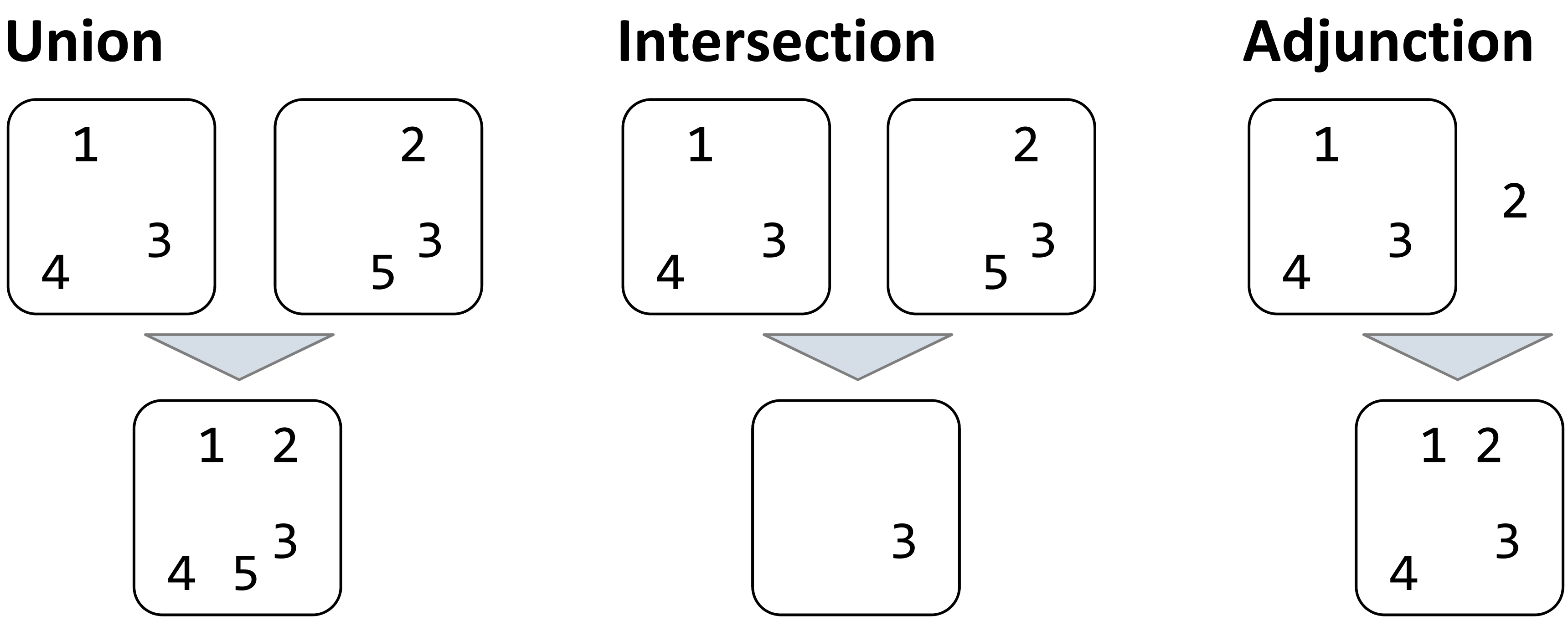
- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets are unordered, just like dictionary entries

```
>>> 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 *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

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
    elif s.first == v:
        return True
    return set_contains(s.rest, v)
```

Sets as Unordered Sequences

Time order of growth

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

$\Theta(n)$

The size of the set

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

$\Theta(n^2)$

Assume sets are the same size

```
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)
```

$\Theta(n^2)$