

## 61A Lecture 21

## Announcements

## Sets

## Sets

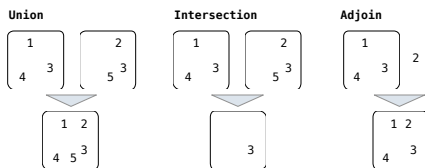
- One more built-in Python container type
- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets are unordered, just like dictionary entries

```
>>> 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
- Adjoin:** 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 linked list that contains no duplicate items.

```
def empty(s):
    return s is Link.empty
```

Time order of growth  
 $\Theta(1)$

```
def set_contains(s, v):
    """Return whether set s contains value v.
```

Time depends on whether  
& where v appears in s

```
>>> s = Link(1, Link(2, Link(3)))
>>> set_contains(s, 2)
True
"""
if empty(s):
    return False
elif s.first == v:
    return True
else:
    return set_contains(s.rest, v)
```

$\Theta(n)$   
Assuming v either  
does not appear in s  
or  
appears in a uniformly  
distributed random location

(Demo)

## Sets as Unordered Sequences

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

Time order of growth

$\Theta(n)$

The size of the set

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

$\Theta(n^2)$

If sets are  
the same size

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

$\Theta(n^2)$

(Demo)

## Sets as Ordered Sequences

## Sets as Ordered Sequences

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

Parts of the program that...	Assume that sets are...	Using...
Use sets to contain values	Unordered collections	<code>empty</code> , <code>set_contains</code> , <code>adjoin_set</code> , <code>intersect_set</code> , <code>union_set</code>
Implement set operations	Ordered linked lists	<code>first</code> , <code>rest</code> , <code>&lt;</code> , <code>&gt;</code> , <code>==</code>

*Different parts of a program may make different assumptions about data*

## Sets as Ordered Sequences

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

```
def intersect_set(set1, set2):
    if empty(set1) or empty(set2):
        return Link.empty
    else:
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Link(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)
```

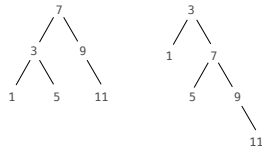
Order of growth?  $\Theta(n)$

## Sets as Binary Search Trees

## Binary Search Trees

**Proposal 3:** A set is represented as a Tree with two branches. Each entry is:

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

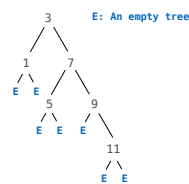


## Binary Tree Class

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

**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]

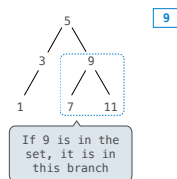
Bin = BinaryTree
t = Bin(3, Bin(1),
        Bin(7, Bin(5), Bin.empty,
             Bin(9, Bin.empty,
                  Bin(11))))
```

## Membership in Binary Search Trees

`set_contains` traverses the tree

- If the element is not the entry, it can only be in either the left or right branch
- By focusing on one branch, we reduce the set by about half with each recursive call

```
def set_contains(s, v):
    if s.is_empty:
        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?  $\Theta(h)$  on average  $\Theta(\log n)$  on average for a balanced tree

## Adjoining to a Tree Set

