Announcements

- Project 3 is due Thursday 10/23 @ 11:59pm
- Please submit two ways: the normal way and using python3 ok --submit!
- You can view your ok submission on the ok website: http://ok.cs61a.org
- Midterm 2 is on Monday 10/27 7pm-9pm
- Review session on Saturday 10/25 3pm-6pm in 2050 VLSB
- Conflict form submissions are due Wednesday 10/22!

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}
```

Sets as Unordered Sequences

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

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

def set_contains(s, v):
    # Return whether set s contains value v.
    # Time depends on whether v appears in s.
    # Time order of growth is \( \Theta(n) \).
    if empty(s):
        return False
    if s.first == v:
        return True
    else:
        return set_contains(s.rest, v)

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

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

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

Sets as Unordered Sequences

Time order of growth
- \( \Theta(n) \) for the size of the set
- \( \Theta(n^2) \) if sets are the same size
- \( \Theta(n^2) \) if sets are not the same size

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

```
Union

Intersection

Adjoin
```

Sets as Unordered Sequences

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

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

def set_contains(s, v):
    # Return whether set s contains value v.
    # Time depends on whether v appears in s.
    # Time order of growth is \( \Theta(n) \).
    if empty(s):
        return False
    if s.first == v:
        return True
    else:
        return set_contains(s.rest, v)

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

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

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

Sets as Unordered Sequences

Time order of growth
- \( \Theta(n) \) for the size of the set
- \( \Theta(n^2) \) if sets are the same size
- \( \Theta(n^2) \) if sets are not the same size

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

```
Union

Intersection

Adjoin
```
Sets as Ordered Sequences

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

<table>
<thead>
<tr>
<th>Parts of the program that...</th>
<th>Assume that sets are...</th>
<th>Using...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use sets to contain values</td>
<td>Unordered collections</td>
<td>empty, <code>set.contains</code>, <code>adjoint_set</code>, <code>intersect_set</code>, <code>union_set</code></td>
</tr>
</tbody>
</table>

Implement set operations

Doubly-linked lists

First, rest, <, >, ==

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

Sets as 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
- 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:
    empty_is_empty = True
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

    @property
    def left(self):
        return self.branches[0]

    @property
    def right(self):
        return self.branches[1]

Bin = BinaryTree
    left = Bin.empty, Bin(9), Bin(1)
    right = Bin.empty, Bin(1), Bin(9)

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.

```python
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)
    else:
        return set.contains(s.left, v)
```

Order of growth? \( O(h) \) on average. \( \Theta(n) \) on average for a balanced tree.

Adjoining to a Tree Set

Right:

Left:

Right:

Stop!