61A Lecture 21

Friday, March 13
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 Thursday 3/19 7pm–9pm
  ▪ Review session on Tuesday 3/17 5pm–6:30pm in 2050 VLSB
  ▪ HKN review session on Sunday 3/15 12–3pm in 10 Evans
  ▪ Conflict form submissions are due Friday 3/13!
  ▪ 1 2-sided sheet of hand-written notes created by you + 2 official study guides
  ▪ Same exam location as midterm 1. See http://cs61a.org/exams/midterm2.html
  ▪ Today's lecture contains the last of the Midterm 2 material (Monday is just examples)

• No lecture next Wednesday 3/18
• No discussion sections next Thursday 3/19 or Friday 3/20
• Lecture next Friday 3/20 is a video (but a great one)
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

```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

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.

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

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

    >>> 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)
```

Time order of growth

\( \Theta(1) \)

Time depends on whether & where \( v \) appears in \( s \)

\( \Theta(n) \)

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

(Demo)
Sets as Unordered Sequences

```python
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)
```

Time order of growth

- \(\Theta(n)\)
  - The size of the set
  - If sets are the same size
  - \(\Theta(n^2)\) (Demo)

Need a new version defined for Link instances
Sets as Ordered Sequences
Sets as Ordered Sequences

Proposal 2: A set is represented by a linked list with unique elements that is \textit{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, set_contains, adjoin_set, intersect_set, union_set</td>
</tr>
<tr>
<td>Implement set operations</td>
<td>Ordered linked lists</td>
<td>first, rest, &lt;, &gt;, ==</td>
</tr>
</tbody>
</table>

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

```python
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
**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

```python
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(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

```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)
    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

Right!

Left!

Right!

Stop!

(Demo)