Lecture #23: Iterators on Trees

Slight Correction from Last Time

- In the last lecture, I defined

```
class BinTree(Tree):
    def __iter__(self): return tree_iter(self)
```

- However, there is already an __iter__ method on BinTree, inherited from Tree, which iterates over the tree's children.
- So instead, let's define (and document)

```
class BinTree(Tree):
    def preorder_values(self):
        """My labels, delivered in preorder (node label first, then labels
        of left child in preorder, then labels of right child in preorder.
        >>> T = BinTree(10, BinTree(5, BinTree(2), BinTree(6)), BinTree(15))
        >>> for v in T.preorder_values(): print(v, end=" ")
        10 5 2 6 15
        >>> list(T.preorder_values())
        [10, 5, 2, 6, 15]"""
        return tree_iter(self)
```

- The **for** statement above shows why it is useful to have iterators (like tree_iter) have an __iter__ method: it allows a **for** loop to take *either* an iterable or an iterator.

Iterating Over a Binary Tree: Strategy

- To create an iterator on a tree, consider this reimplementation of tree_to_list_preorder from Lecture 21 (for binary trees):

```
def tree_to_list_preorder(T):
    """The list of all labels in T, listing the labels
    of trees before those of their children, and listing their
    children left to right (preorder)."""
    if T.is_empty:
        return ()
    else:
        return (T.label,) + tree_to_list_preorder(T.left) \
                          + tree_to_list_preorder(T.right)
```

- Suppose that we wanted to return just the first item (T's label). What work would be left to do?
- Clearly, returning (iterating through) all the values in the left child and then on the right.
- To get the next value (after T's label), we'll need to *start* iterating through the left child, leaving *its* children to be processed.
- When the next tree in the queue is empty, discard it.

Iterating Over a Binary Tree: Data Structure

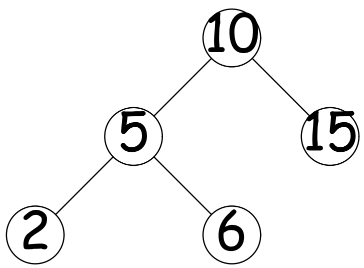
- So, to iterate over a tree, let's have our iterator consist of a *list of subtrees that still need iterating over*.

```
class BinTree(Tree):
    ...
    def preorder_values(self): return tree_iter(self)
class tree_iter:
    def __init__(self, the_tree):
        self._work_queue = [ the_tree ]
    ...
    def __next__(self): ?

    # Have iterator implement __iter__, so that it can
    # be used in for statements, etc.
    def __iter__(self): return self
```

Iterating Over a Binary Tree: Example

- Suppose that we create iter = T.preorder_values() where T is



- Initially, iter._work_queue would contain just the tree rooted at the node labeled 10 (let's just say 'Tree 10' from now on).
- After the first call to iter.__next__(), which returns 10, iter._work_queue would contain [Tree 5, Tree 15]
- After the second call to iter.__next__(), which returns 5, iter._work_queue would contain [Tree 2, Tree 6, Tree 15]
- Then [Empty, Empty, Tree 6, Tree 15]
- Then, throw away the empty trees and process Tree 6, returning 6 and leaving its children: [Empty, Empty, Tree 15]

Iterating Over a Binary Tree: Code

```
class BinTree(Tree):
    ...
    def preorder_values(self): return tree_iter(self)
class tree_iter:
    def __init__(self, the_tree):
        self._work_queue = [ the_tree ]

    def __next__(self):
        while _________________________:
            subtree = self._work_queue.pop(0) # Get first item
            if subtree.is_empty:
                ____
            else:
                _____________________  =  subtree.left, subtree.right
                return _____________

        ___________________

    def __iter__(self): return self
```

Iterating Over a Binary Tree: Code

```
class BinTree(Tree):
    ...
    def preorder_values(self): return tree_iter(self)
class tree_iter:
    def __init__(self, the_tree):
        self._work_queue = [ the_tree ]

    def __next__(self):
        while len(self._work_queue) > 0:
            subtree = self._work_queue.pop(0) # Get first item
            if subtree.is_empty:
                ____
            else:
                _____________________ =  subtree.left, subtree.right
                return _____________
        ___________________

    def __iter__(self): return self
```

Iterating Over a Binary Tree: Code

```
class BinTree(Tree):
    ...
    def preorder_values(self): return tree_iter(self)
class tree_iter:
    def __init__(self, the_tree):
        self._work_queue = [ the_tree ]

    def __next__(self):
        while len(self._work_queue) > 0:
            subtree = self._work_queue.pop(0) # Get first item
            if subtree.is_empty:
                pass
            else:
                _____________________ =  subtree.left, subtree.right
                return _____________

        ___________________

    def __iter__(self): return self
```

Iterating Over a Binary Tree: Code

```
class BinTree(Tree):
    ...
    def preorder_values(self): return tree_iter(self)
class tree_iter:
    def __init__(self, the_tree):
        self._work_queue = [ the_tree ]

    def __next__(self):
        while len(self._work_queue) > 0:
            subtree = self._work_queue.pop(0) # Get first item
            if subtree.is_empty:
                pass
            else:
                self._work_queue[0:0] =  subtree.left, subtree.right
                return _____________

        ___________________

    def __iter__(self): return self
```

Iterating Over a Binary Tree: Code

```
class BinTree(Tree):
    ...
    def preorder_values(self): return tree_iter(self)
class tree_iter:
    def __init__(self, the_tree):
        self._work_queue = [ the_tree ]

    def __next__(self):
        while len(self._work_queue) > 0:
            subtree = self._work_queue.pop(0) # Get first item
            if subtree.is_empty:
                pass
            else:
                self._work_queue[0:0] =  subtree.left, subtree.right
                return subtree.label

        ___________________

    def __iter__(self): return self
```

Iterating Over a Binary Tree: Code

```
class BinTree(Tree):
    ...
    def preorder_values(self): return tree_iter(self)
class tree_iter:
    def __init__(self, the_tree):
        self._work_queue = [ the_tree ]

    def __next__(self):
        while len(self._work_queue) > 0:
            subtree = self._work_queue.pop(0) # Get first item
            if subtree.is_empty:
                pass
            else:
                self._work_queue[0:0] =  subtree.left, subtree.right
                return subtree.label
        raise StopIteration

    def __iter__(self): return self
```

Small Technical Node on Speed

- Inserting and deleting from the beginning of a Python list can be slow (when?).
- So we usually add and delete from the end (reversing the lists):

```
class tree_iter:
    def __init__(self, the_tree):
        self._work_queue = [ the_tree ]

    def __next__(self):
        while len(self._work_queue) > 0:
            subtree = self._work_queue.pop()
            if subtree.is_empty:
                pass
            else:
                self._work_queue += subtree.right, subtree.left
                # Reversed!
                return subtree.label
        raise StopIteration
```

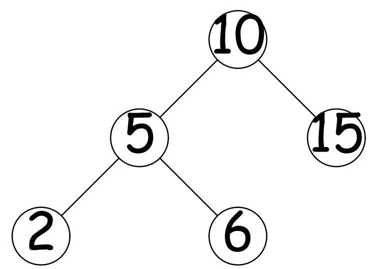
Iterating Over a Binary Search Tree In Order

- The iterator we just defined iterates in *preorder*: first the root's label, then the labels of the left child in preorder, then the labels of the right child in preorder.
- But for a binary search tree, this gives the values out of order.
- Instead, we want the labels of the left child (in order), then the root's label, then those of the right.
- This is known as an *inorder traversal* of a binary tree. For search trees, it gives us the values in order.
- We could get this with a different iterator:

```
class BinTree(Tree):
    ...
    def inorder_values(self):
        """An iterator over my labels in order.
        >>> T = BinTree(10, BinTree(5, BinTree(2), BinTree(6)), BinTree(15))
        >>> for v in T.inorder_values():
        ...     print(v, end=" ")
        2 5 6 10 15"""
        return inorder_tree_iter(self)
```

The Inorder Iterator

- To get this change, we have to put *both* trees and labels in the work queue.
- Let's simplify by assuming that we never use trees as labels (no trees of trees).
- So for the tree we looked at previously:



  we'd start with Tree 10 (as before), and process that by replacing it with Tree 5, 10 (the label), and Tree 15 in the queue.
- When we get to a label in the queue, we return it.

Using Inorder Iterators: A __repr__ Method

- It would be nice to have a specialized way to print binary search trees, which we can do by redefining BinTree.__repr__:

```
class BinTree(Tree):
    ...
    def __repr__(self):
        """A string representing me (used by the interpreter).
       >>> T = BinTree(10, BinTree(5, BinTree(2), BinTree(6)), BinTree(15))
       >>> T
       {2, 5, 6, 10, 15}"""
       result = "{"
       for v in self.inorder_values():
           if result != "{":
               result += ", "
           result += repr(v)
       return result + "}"
       # Can you do it in one line?
       return
```

Using Inorder Iterators: A __repr__ Method

- It would be nice to have a specialized way to print binary search trees, which we can do by redefining BinTree.__repr__:

```
class BinTree(Tree):
    ...
    def __repr__(self):
        """A string representing me (used by the interpreter).
        >>> T = BinTree(10, BinTree(5, BinTree(2), BinTree(6)), BinTree(15))
        >>> T
        {2, 5, 6, 10, 15}"""
        result = "{"
        for v in self.inorder_values():
            if result != "{":
                result += ", "
            result += repr(v)
        return result + "}"
        # Can you do it in one line?
        return "{" + ', '.join(map(repr, self.inorder_values())) + "}"
```

Intersection

- In lab, you looked at intersection between Python sets.
- Since we're using BinTrees as sets, it makes sense to consider the same problem here.
- One approach is brute force, for value in one set, see if it is in the other:

```
def intersection(s1, s2):
    """The intersection of the values in BinTrees S1 and S2."""
    result = BinTree.empty_tree
    for v in s1.preorder_values():
        if tree_find(s2, v):
            result = dtree_add(result, v)
    return result
```

- If our trees remain "bushy" (shallow), how long does this take, as a function of N, the maximum of the sizes of s1 and s2?

Intersection

- In lab, you looked at intersection between Python sets.
- Since we're using BinTrees as sets, it makes sense to consider the same problem here.
- One approach is brute force, for value in one set, see if it is in the other:

```
def intersection(s1, s2):
    """The intersection of the values in BinTrees S1 and S2."""
    result = BinTree.empty_tree
    for v in s1.preorder_values():
        if tree_find(s2, v):
            result = dtree_add(result, v)
    return result
```

- If our trees remain "bushy" (shallow), how long does this take, as a function of N, the maximum of the sizes of s1 and s2? **A:** $O(N \lg N)$
- That's because there are $O(N)$ items in s1; checking for each of them in s2 takes $O(\lg N)$ (if bushy); we add a maximum of N values to the result; and adding each of them also takes $O(\lg N)$.

Using Inorder Iterators for Intersection

- We can avoid doing repeated searches by iterating through both sets of values simultaneously.
- Can use Python's built-in next function: next(an_iterator, default) returns the result of calling an_iterator.__next__(), except that if that causes an exception, next returns default instead.
- Unfortunately, there is a price: resulting tree is not bushy [why?]

```
def intersection(s1, s2):
    it1, it2 = s1.inorder_values(), s2.inorder_values()
    v1, v2 = next(it1, None), next(it2, None)
    result = BinTree.empty_tree
    while v1 is not None and v2 is not None:
        if v1 == v2:
            result = dtree_add(result, v1)
            v1, v2 = next(it1, None), next(it2, None)
        elif _______:
            ____________________
        else:
            ____________________
    return result
```

Using Inorder Iterators for Intersection

- We can avoid doing repeated searches by iterating through both sets of values simultaneously.
- Can use Python's built-in next function: next(an_iterator, default) returns the result of calling an_iterator.__next__(), except that if that causes an exception, next returns default instead.
- Unfortunately, there is a price: resulting tree is not bushy [why?]

```
def intersection(s1, s2):
    it1, it2 = s1.inorder_values(), s2.inorder_values()
    v1, v2 = next(it1, None), next(it2, None)
    result = BinTree.empty_tree
    while v1 is not None and v2 is not None:
        if v1 == v2:
            result = dtree_add(result, v1)
            v1, v2 = next(it1, None), next(it2, None)
        elif v1 < v2:
            ____________________
        else:
            ____________________
    return result
```

Using Inorder Iterators for Intersection

- We can avoid doing repeated searches by iterating through both sets of values simultaneously.
- Can use Python's built-in next function: next(an_iterator, default) returns the result of calling an_iterator.__next__(), except that if that causes an exception, next returns default instead.
- Unfortunately, there is a price: resulting tree is not bushy [why?]

```
def intersection(s1, s2):
    it1, it2 = s1.inorder_values(), s2.inorder_values()
    v1, v2 = next(it1, None), next(it2, None)
    result = BinTree.empty_tree
    while v1 is not None and v2 is not None:
        if v1 == v2:
            result = dtree_add(result, v1)
            v1, v2 = next(it1, None), next(it2, None)
        elif v1 < v2:
            v1 = next(it1, None)
        else:
            v2 = next(it2, None)
    return result
```