Lecture #21: Search Trees, Sets

General Tree Class (From Last Lecture)

```
class Tree:
    """A Tree consists of a label and a sequence
    of 0 or more Trees, called its children."""

    def __init__(self, label, *children):
        """A Tree with given label and children."""

    def __str__(self):  # Used by print(.) and str(.)
    def __repr__(self): # Used by the interpreter

    @property
    def is_leaf(self): return self.arity == 0
    @property
    def label(self): ...

    @property
    def arity(self):
        """The number of my children."""
    def __iter__(self):
        """An iterator over my children."""
    def __getitem__(self, k):
        """My kth child."""
```


A Search

```
def tree_contains(T, x):
    """True iff x is a label in T."""
    if x == T.label:
       return True
    else:
       for c in T:
           if tree_contains(c, x):
               return True
    return False
```

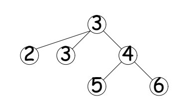
- This particular definition of trees lends itself to Noetherian induction with no explicit base case.

```
def tree_contains(T, x):
    """True iff x is a label in T."""
    return x == T.label or \
           any(map(lambda C: tree_contains(C, x),
                   T))
```


Printing Trees

- The __str__ method lends itself to recursion:

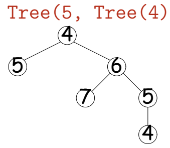
```
class Tree:
    ...
    def __str__(self):
        """My printed string representation (leaves print only
        their labels).
        >>> str(Tree(3, Tree(2), Tree(3), Tree(4, Tree(5), Tree(6))))
        '(3 2 3 (4 5 6))'
        """
        if self.is_leaf:
            return str(self.label)
        return "(" + str(self.label) + " " + \
               " ".join(map(str, self)) + ")"

    def __repr__(self):
        """My string representation for the interpreter, etc.
        >>> Tree(3, Tree(2), Tree(3), Tree(4, Tree(5), Tree(6)))
        Tree:(3 2 3 (4 5 6))"""
        return "Tree:" + str(self)
```



Tree to List

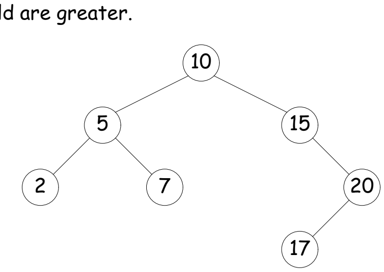
- Another example with no explicit base cases:

```
from functools import reduce
from operator import add
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).
    >>> B = Tree(4, Tree(5), Tree(6, Tree(7), Tree(5, Tree(4))))
    >>> B
    Tree:(4 5 (6 7 (5 4)))
    >>> tree_to_list_preorder(B)
    (4 5 6 7 5 4)
    """
    return sum(map(tree_to_list_preorder, T), (T.label,))
```



Search Trees

- The book talks about *search trees* as implementations of sets of values.
- Here, the purpose of the tree is to divide data into smaller parts.
- In a *binary search tree*, each node is either empty or has two children that are binary search trees such that all labels in the first (left) child are less than the node's label and all labels in the second (right) child are greater.



Search Tree Class

- To work on search trees, it is useful to have a few more methods on trees:

```
class BinTree(Tree):
    @property
    def is_empty(self):
        """This tree contains no labels or children."""

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

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

    """The empty tree"""
    empty_tree = ...
```


Tree Search Program

```
def tree_find(T, x):
    """True iff x is a label in set T, represented as a search tree.
    That is, T
       (a) Is an empty tree if T.is_empty(), or
       (b) Has two children, T.left and T.right, both search trees,
           and all labels in T.left are less than T.label,
           and all labels in T.right are greater than T.label."""
    if T.is_empty:
        return False
    if x == T.label:
        return True
    if x < T.label:
        return tree_find(T.left, x)
    else:
        return tree_find(T.right, x)
```

- Since the values of the only recursive calls are immediately returned, this program is tail-recursive.

Iterative Tree Search Program

```
def tree_find(T, x):
    """True iff x is a label in set T, represented as a search tree.
    That is, T
       (a) Is an empty tree if T.is_empty(), or
       (b) Has two children, T.left and T.right, both search trees,
           and all labels in T.left are less than T.label,
           and all labels in T.right are greater than T.label."""
    while not T.is_empty:
        if x == T.label:
            return True
        elif x < T.label:
            T = T.left
        else:
            T = T.right
    return False
```


Timing

- How long does the tree_find program (search binary tree) take in the worst case,
  1. As a function of H, the height of the tree? (The *height* is the maximum distance from the root to a leaf.) **A:** $\Theta(H)$
  2. As a function of N, the number of keys in the tree? **A:** $\Theta(N)$
  3. As a function of H if the tree is as shallow as possible for the amount of data? **A:** $\Theta(H)$
  4. As a function of N if the tree is as shallow as possible for the amount of data? **A:** $\Theta(\lg N)$