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):
        """My string representation."""

    @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

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

Printing Trees

- The `__str__` method lends itself to recursion:

```
class Tree:
    ...
    def __str__(self):
        """My printed string representation.
        >>> str(Tree(3, Tree(2), Tree(3), Tree(4, Tree(5), Tree(6)))
        Tree:(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."""
        return "Tree:" + str(self)
```

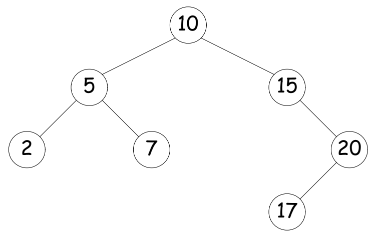
Tree to List

- 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, 5, Tree(6, 7, Tree(5, 4)))
    >>> B
    Tree:(4 5 (6 7 (5 4)))
    >>> tree_to_list_preorder(B)
    (4 5 6 7 5 4)
    """
    return (T.label, ) + \
           reduce(add, map(tree_to_list_preorder, T, ()))
```

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) Represents an empty tree if its label is None, or
       (b) has two children, both search trees, and all labels in
           T[0] are less than T.label, and all labels in T[1] 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)
```

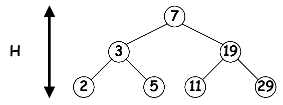
- Since the values of the only recursive calls are immediately returned, this program is tail-recursive.

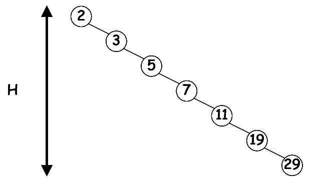
Iterative Tree Search Program

```
def tree_find(T, x):
    """True iff x is a label in set T, represented as a search tree.
    That is, T
       (a) Represents an empty tree if its label is None, or
       (b) has two children, both search trees, and all labels in
           T[0] are less than T.label, and all labels in T[1] 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

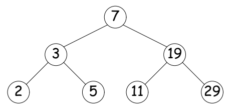
- How long does the tree_find program (search binary tree) take in the worst case,
  1. As a function of H, the height of the tree? (The *height* is the maximum distance from the root to a leaf.) **A:** $\Theta(H)$
  2. As a function of N, the number of keys in the tree? **A:** $\Theta(N)$
  3. As a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

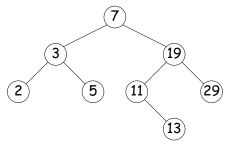




Adding (Adjoining) a Value

- Must add values to a search tree in the right place: the place tree_find would try to find them.
- For example, if we add 17 to the search tree on left, we get the one on the right:





- Simplest always to add at the bottom (leaves) of the tree.

Non-destructive Add

- Broadly, there are two styles for dealing with structures that change over time:
  - *Non-destructive* operations preserve the prior state of the structure and create a new one.
  - *Destructive* operations, as a side effect, *may* modify the previous structure, losing information about its previous contents.

```
def tree_add(T, x):
    """Assuming T is a binary search tree, a new binary search tree
    that contains all previous values in T, plus X
    (if not previously present)."""
    if T.is_empty:
        return Tree(x)
    elif x == T.label:
        return T
    elif x < T.label:
        return tree_add(T.left, x)
    else:
        return tree_add(T.right, x)
```

Destructive Operations

- Destructive operations can be appropriate in circumstances where
  - We want speed: avoid the work of creating new structures.
  - The same data structure is referenced from multiple places, and we want all of them to be updated.
- First requires that we add capabilities to our class:

```
class BinTree(Tree):
    def set_left(self, newval):
        """Assuming NEWVAL is a BinTree, sets SELF.left to NEWVAL."""
        assert type(newval) is BinTree
        self[0] = newval

    def set_right(self, newval):
        """Assuming NEWVAL is a BinTree, sets SELF.right to NEWVAL."""
        assert type(newval) is BinTree
        self[1] = newval
```

Destructive Add

- Destructive add looks very much like the non-destructive variety.

```
def dtree_add(T, x):
    """Assuming T is a binary search tree, a binary search tree
    that contains all previous values in T, plus X
    (if not previously present). May destroy the initial contents
    of T."""
    if T.is_empty:
        return Tree(x)
    elif x == T.label:
        return T
    elif x < T.label:
        set_left(tree_add(T.left, x)
        return T
    else:
        set_right(tree_add(T.right, x)
        return T
```

Binary Search Trees as Sets

- For data that has a well-behaved ordering relation (a *total ordering*), BinTree provides a possible implementation of Python's set type.
- x in S corresponds to tree_find(S, x)
- S.union({x}) or S + {x} correspond to tree_add(S, x)
- S.add(x) or S += {x} correspond to dtree_add(S, x)
- Actually, Python uses *hash tables* for its sets, which you'll see in CS61B (plug).

Problem: Iterating Through All Values

- Iterating over a tree gives us only the children, at present.
- Could we get *all* the nodes or labels in a tree,
- ...and for binary search trees, could we get them in sorted order?
- All it takes is a method that returns an appropriate iterator or iterable, and we can write, e.g.,

```
for val in T.inorder_values():
    ...
```

- How would we do that?

```
class Tree:
   ...
   def inorder_values(self):
       return ?
```

- Here, ? could be a list of all values in the tree. What else?