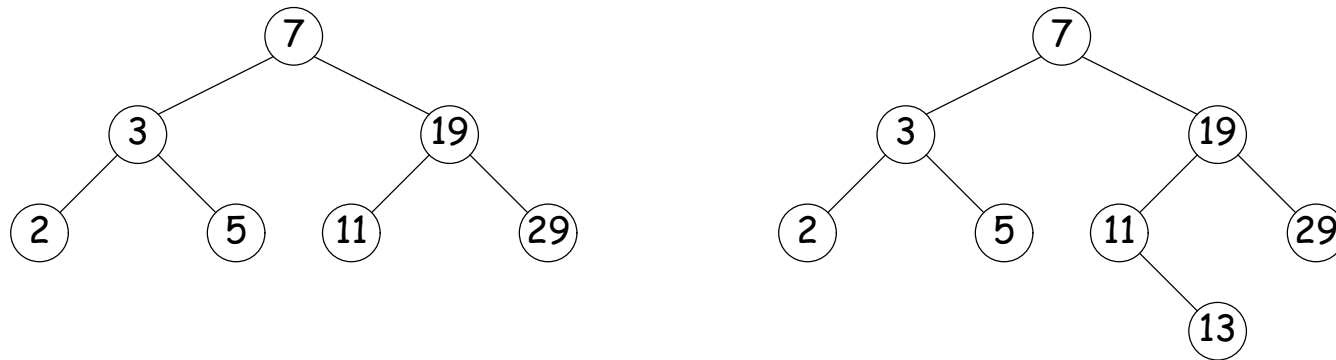


Lecture #22: Search Trees and Sets, Part II

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 _____  
    elif x == T.label:  
        return _  
    elif x < T.label:  
        return _____  
    else:  
        return _____
```

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 BinTree(x)  
    elif x == T.label:  
        return _  
    elif x < T.label:  
        return _____  
    else:  
        return _____
```

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 BinTree(x)  
    elif x == T.label:  
        return T  
    elif x < T.label:  
        return _____  
    else:  
        return _____
```

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 BinTree(x)  
    elif x == T.label:  
        return T  
    elif x < T.label:  
        return BinTree(T.label, tree_add(T.left, x), T.right)  
    else:  
        return BinTree(T.label, T.left, 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."""
        ...

    def set_right(self, newval):
        """Assuming NEWVAL is a BinTree, sets SELF.right to 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 _____  
    elif x == T.label:  
        return _  
    elif x < T.label:  
        _____  
        return _  
    else:  
        _____  
        return _
```


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 BinTree(x)  
    elif x == T.label:  
        return T  
    elif x < T.label:  
  
        _____  
        return _  
    else:  
  
        _____  
        return _
```

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 BinTree(x)  
    elif x == T.label:  
        return T  
    elif x < T.label:  
        T.set_left(dtree_add(T.left, x))  
        return T  
    else:  
        T.set_right(dtree_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: Make a Balanced Tree

- I have a sorted list, and would like to turn it into the best (shallowest) binary search tree that contains the same values.
- Hint: Getting a shallow tree requires making the two child subtrees of each node have equal numbers of values (± 1).

```
def list_to_tree(L):  
    """Assuming L is a sorted list, a (nearly) balanced  
    search tree containing exactly the values in L."""  
    if _____:  
        return _____  
    else:  
        root_index = _____  
        return
```

Problem: Make a Balanced Tree

- I have a sorted list, and would like to turn it into the best (shallowest) binary search tree that contains the same values.
- Hint: Getting a shallow tree requires making the two child subtrees of each node have equal numbers of values (± 1).

```
def list_to_tree(L):  
    """Assuming L is a sorted list, a (nearly) balanced  
    search tree containing exactly the values in L."""  
    if len(L) == 0:  
        return Tree.empty_tree  
    else:  
        root_index = _____  
        return
```

Problem: Make a Balanced Tree

- I have a sorted list, and would like to turn it into the best (shallowest) binary search tree that contains the same values.
- Hint: Getting a shallow tree requires making the two child subtrees of each node have equal numbers of values (± 1).

```
def list_to_tree(L):  
    """Assuming L is a sorted list, a (nearly) balanced  
    search tree containing exactly the values in L."""  
    if len(L) == 0:  
        return Tree.empty_tree  
    else:  
        root_index = len(L) // 2  
        return
```

Problem: Make a Balanced Tree

- I have a sorted list, and would like to turn it into the best (shallowest) binary search tree that contains the same values.
- Hint: Getting a shallow tree requires making the two child subtrees of each node have equal numbers of values (± 1).

```
def list_to_tree(L):  
    """Assuming L is a sorted list, a (nearly) balanced  
    search tree containing exactly the values in L."""  
    if len(L) == 0:  
        return Tree.empty_tree  
    else:  
        root_index = len(L) // 2  
        return BinTree(L[root_index],  
                        list_to_tree(L[:root_index]),  
                        list_to_tree(L[root_index+1:]))
```

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.preorder_values():  
    ...
```

- How would we do that?

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

- Here, ? could be a list of all values in the tree, which we've done already. What else?

Creating an Iterator (Review)

- As we've seen (Lecture 17), an *iterator* is an object that implements a method `__next__` on itself.
- When called, it should either return a value or raise `StopException`.
- An *iterable* is an object that either
 - Implements a method `__iter__(self)` that returns an iterator, or
 - Implements a method `__getitem__(self, k)` that returns item number `k` (or raises an exception).
- Many methods and constructs take iterables, including `for` clauses, `map`, `reduce`, `zip`, and many others.
- When given an iterable, these create a new iterator from it (using `__iter__`), which allows one pass over the data.

Iterating Over a Binary Tree: Strategy

- To create an iterator on a tree, consider this reimplement 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 __iter__(self): return tree_iter(self)
```

```
class tree_iter:
```

```
    def __init__(self, the_tree):
```

```
        self._work_queue = [ the_tree ]
```

```
    ...
```

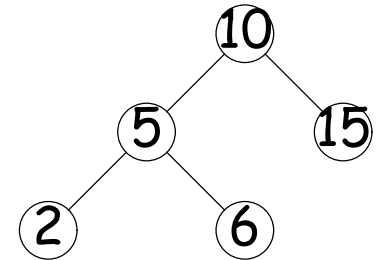
```
    def __next__(self): ?
```

```
# Standard hack: by making iterators implement __iter__, they  
# are themselves iterable, so you can use them in  
# for statements, etc.
```

```
def __iter__(self): return self
```

Iterating Over a Binary Tree: Example

- Suppose that we create `iter = T.__iter__()` 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]