Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — September 26, 2018
Instructors: Zachary Abel, Erik Demaine, Jason Ku — Recitation 5b

Recitation 5b

Binary Search Trees

A **binary search tree** (BST) is a binary tree that satisfies the BST Property.

> **BST Property:** Every node in a BST contains a key that is at least as large as any key in the node's left subtree, and no greater than any key in the node's right subtree.

As a data structure, a BST stores a set of keys, and supports three types of operations on those keys: **search** (e.g. `find`), **dynamic** (e.g. `insert`, `delete`), and **order** (e.g. `find_min`, `find_next`). Each of these operations can be performed in time proportional to the tree's height. This is linear if $h = O(n)$, but will be logarithmic if $h = O(\log n)$. Next lecture, we will show how to enforce logarithmic height by performing some maintenance after dynamic operations. Below are a couple BST examples, though having students generate keys will be more engaging.

```
____5__
0__  _7_
 _2_ 6 8_
 1 4    9
```

```
  ____6_________________
_1_       __________17__________
0 2_    __11_____          _____25__
   4 _8_  __13__  __21__   27__
      7 9  12  15  20  23    29
```

Nodes: How do we represent a binary tree in code? Last week, we showed how to use an array to store a complete binary tree with left-aligned lowest level. For BSTs, we will want to be able to represent all possible binary trees, not just complete ones. To do this, we will represent the tree as a collection of linked node objects, each containing a constant number of attributes: the node's key, and pointers to its parent, left child, and right child. Using a linked pointer-based container will allow us to re-position many nodes in constant time. We also define a helper function `_is_empty()` to tell us whether the BST contains any items.

```
class BST:
    def __init__(self, item = None, parent = None):
        self.item   = item
        self.parent = parent
        self.left   = None
        self.right  = None
```

```
    def _is_empty(self): return self.item is None
```

Recursive Traversal

If you were to loop through and print successive keys of a BST, by the BST Property you would print the keys in sorted order. Building a BST and then printing keys while traversing nodes in order from minimum to maximum, is a sorting algorithm we call **BST sort**.

```
def iter_recursive(self):
    if self.left:
        yield from self.left.iter_recursive()
    yield self.item
    if self.right:
        yield from self.right.iter_recursive()
```

Search/Order Operations

The BST Property makes it easy to perform search and order operations for keys within the BST. In our implementation, we will have two types of operations. The first are data structure internal node operations, which return and act on nodes of the tree; by convention, these operations have been prefixed with an underscore and suffixed with `_node`. In addition, we support the ordered dynamic set interface API as discussed in Lecture 4.

Minimum: Consider BST node A. To find the node having an item with minimum key in a A's subtree, return A if A has no left child, or recursively find the minimum of A's left child. Returning the item given that node is then trivial. **Exercise:** demonstrate finding the minimum of your example tree.

```
def find_min(self):
    if self._is_empty(): return None
    node = self._min_node()
    return node.item if node else None
```

```
def _min_node(self):  # assumes self has item
    if self.left:
        return self.left._min_node()
    return self
```

Find: To find a node in A's subtree containing a queried key, return A if A's key is the same as the query, or recursively search in A's left or right subtree. If you reach the bottom without finding a node containing the key, then you know the key is not in the tree. **Exercise:** demonstrate finding keys that are both contained and not contained in your example tree.

```
def find(self, k):
    if self._is_empty(): return None
    node = self._find_node(k)
    return node.item if node else None
```

```python
def _find_node(self, k):  # assumes self has item
    if k == self.item.key:
        return self                      # keys are equal
    if k < self.item.key and self.left:
        return self.left._find_node(k)   # recursive call in left subtree
    if k > self.item.key and self.right:
        return self.right._find_node(k)  # recursive call in right subtree
    return None                          # key not found
```

Find Next: Given a key k, `find_next(k)` asks us to return the stored item having the smallest key strictly greater than k. To do this, we first search for a node containing a key close to k: either k, the largest key smaller than k, or the smallest key larger than k (`_close_node` implements this behavior). In the latter two cases, we will then find the item stored in the next node in an in-order traversal of the tree (`_successor_node` implements this behavior). To find the **successor** of node A, i.e. the node containing the next larger key in the BST, return the minimum of A's right subtree, or A's lowest ancestor having A in its left subtree. **Exercise:** demonstrate the successor of: a node with a right child, a node without a right child, and the right-most node.

```python
def find_next(self, k):
    if self._is_empty(): return None
    node = self._close_node(k)
    if node.item.key < k:
        node = node._successor_node()
    return node.item if node else None
```

```python
def _close_node(self, k):
    if k < self.item.key and self.left:
        return self.left._close_node(k)
    if k > self.item.key and self.right:
        return self.right._close_node(k)
    return self
```

```python
def _successor_node(self):
    if self.right:
        return self.right._min_node()   # minimum of right subtree
    node = self
    while node.parent and (node.parent.right is node):
        node = node.parent
    return node.parent                  # None if self contains largest item
```

Dynamic Operations

A BST is a **dynamic** data structure, meaning that it supports insertion and deletion of keys over time. When performing dynamic operations, the subtrees of all ancestors of the inserted or deleted node will change. We include a **maintenance** function which is called by the lowest node whose subtree is modified by a dynamic operation. Right now this function is a stub, but we will use this function to maintain subtree properties later.

```
def _maintain(self):
    pass
```

Insert: To insert a key into A's subtree, recursively insert into A's left or right subtree depending on whether A's key is larger or smaller. When A is missing the relevant child, link a new BST node as a child containing the inserted key. **Exercise:** Insert some keys into your example BST.

```
def insert(self, x):
    if self._is_empty():
        self.item = x                               # insert key
        self._maintain()
    elif x.key < self.item.key:
        if self.left is None:
            self.left = self.__class__(None, self)  # make new left child
        self.left.insert(x)                         # recursive call on left
    else:
        if self.right is None:
            self.right = self.__class__(None, self) # make new right child
        self.right.insert(x)                        # recursive call on right
```

Delete: To delete A's key from its subtree, we must first find the key to delete, and then remove the key (and a node) from the tree. To remove a node from the tree, we consider three cases:

1. If A has two children, find the node B in A's right subtree having minimum key, which cannot have two children. Swap the keys of A and B, and recursively delete B.

2. If A has only one child, replace A's key and child pointers with those of its child B, while re-linking the parent pointers of B's children.

3. If A has no children, remove A by removing the child pointer from its parent.

Exercise: Delete some keys from your example BST, specifically keys contained in nodes that: have no children, have one child, have two children.

```
def delete(self, k):
    if self._is_empty():
        raise IndexError('delete from empty tree')
    node = self._find_node(k)      # find node
    if node is None:
        return None
    item = node.item
    node._delete_node()            # remove the node
    return item
```

```
def _delete_node(self):
    node = self
    if self.left and self.right:                  # has two children
        node = self.right._min_node()
        self.item = node.item
    if  node.right:    node._replace(node.right)   # has one child
    elif node.left:    node._replace(node.left)
    else:                                         # has no children
        if node.parent is None:
            node.item = None
            return
        if node.parent.right is node:
            node.parent.right = None
        else:
            node.parent.left  = None
        node = node.parent
    node._maintain()
```

This BST implementation is designed so that the root node persists as the root throughout all operations. Because of this, we need to avoid removing nodes that may be the root, or else we would lose access to the BST! To support this behavior, our `_delete_node` function calls a a `replace(a, b)` auxiliary procedure, which clobbers the properties of a node with another node, and re-links its children to point to the new parent. **Exercise:** have students work out which pointers need re-linking before showing them this.

```
def replace(self, node):
    self.item  = node.item
    self.left  = node.left
    self.right = node.right
    if self.left:   self.left.parent  = self
    if self.right:  self.right.parent = self
```

Iterative Traversal

Here is an iterative version of the in-order traversal that uses two of the node operations presented above. In your problem set, we will ask you to prove that traversals takes linear time.

```
def iter_iterative(self):
    node = self._min_node()
    while node:
        yield node.item
        node = node._successor_node()
```

Exercise: Our iterative and recursive traversal code produces the same result when called by the root of a BST. How do they differ when called by a node that is **not** the root?