BSTNode

class BSTNode(object):
    """A node in the vanilla BST tree."""

def __init__(self, parent, k):
    """Creates a node.
    
    Args:
        parent: The node's parent.
        k: The key of the node.
    """
    self.key = k
    self.parent = parent
    self.left = None
    self.right = None


def find(self, k):
    """Finds and returns the node with key k from the subtree
    rooted at this
    node.
    
    Args:
        k: The key of the node we want to find.
    """
    if k == self.key:
        return self
    elif k < self.key:
        if self.left is None:
            return None
        else:
            return self.left.find(k)
    else:
        if self.right is None:
            return None
        else:
            return self.right.find(k)


def find_min(self):
    """Finds the node with the minimum key in the subtree rooted
    at this
    node.
    
    Returns:
        The node with the minimum key.
    """
    current = self
    while current.left is not None:
        current = current.left
    return current
def next_larger(self):
    """Returns the node with the next larger key (the successor) in the BST."
    
    if self.right is not None:
        return self.right.find_min()
    current = self
    while current.parent is not None and current is current.
        parent.right:
            current = current.parent
    return current.parent

def insert(self, node):
    """Inserts a node into the subtree rooted at this node.
    
    Args:
        node: The node to be inserted.
    """
    if node is None:
        return
    if node.key < self.key:
        if self.left is None:
            node.parent = self
            self.left = node
        else:
            self.left.insert(node)
    else:
        if self.right is None:
            node.parent = self
            self.right = node
        else:
            self.right.insert(node)

def delete(self):
    """Deletes and returns this node from the BST."""
    if self.left is None or self.right is None:
        if self is self.parent.left:
            self.parent.left = self.left or self.right
        if self.parent.left is not None:
            self.parent.left.parent = self.parent
        else:
            self.parent.right = self.left or self.right
        if self.parent.right is not None:
            self.parent.right.parent = self.parent
        return self
    else:
        s = self.next_larger()
        self.key, s.key = s.key, self.key
        return s.delete()
**BST**

```python
class BST(object):
    def __init__(self):
        self.root = None

    def find(self, k):
        return self.root and self.root.find(k)

    def find_min(self):
        """Returns the minimum node of this BST."""
        return self.root and self.root.find_min()

    def insert(self, k):
        node = BSTNode(None, k)
        if self.root is None:
            # The root’s parent is None.
            self.root = node
        else:
            self.root.insert(node)

    def delete(self, k):
        """Deletes and returns a node with key k if it exists from
        the BST.

        Args:
            k: The key of the node that we want to delete.
        """
        node = self.find(k)
        if node is None:
            return None
        if node is self.root:
            pseudoroot = BSTNode(None, 0)
            pseudoroot.left = self.root
            self.root.parent = pseudoroot
            deleted = self.root.delete()
            self.root = pseudoroot.left
            if self.root is not None:
                self.root.parent = None
                return deleted
        else:
            return node.delete()
```
```python
def next_larger(self, k):
    """Returns the node that contains the next larger (the successor) key in
    the BST in relation to the node with key k.

    Args:
        k: The key of the node of which the successor is to be found.

    Returns:
        The successor node.
    ""
    node = self.find(k)
    return node and node.next_larger()
```

**MinBSTNode**

```python
class MinBSTNode(BSTNode):
    """A node in BST which is augmented to keep track of the node
    with the minimum key in the subtree rooted at this node.
    ""
    __init__(self, parent, key):
        super(MinBSTNode, self).__init__(parent, key)
        self.min = self
```