```python
"""
    Binary Search Tree
    ------------------
    The Binary Search Tree represents an ordered
symbol table of generic
    key-value pairs.  Keys must be comparable.  Does
not permit duplicate keys.
    When assocating a value with a key already
present in the BST, the previous
    value is replaced by the new one.  This
implementation is for an unbalanced
    BST.

    Pseudo Code: http://algs4.cs.princeton.edu/32bst
"""


class Node(object):
    """
    Implementation of a Node in a Binary Search Tree.
    """

    def __init__(self, key=None, val=None,
size_of_subtree=1):
        self.key = key
        self.val = val
        self.size_of_subtree = size_of_subtree
        self.left = None
        self.right = None


class BinarySearchTree(object):
    """
    Implementation of a Binary Search Tree.
    """

    def __init__(self):
```

```python
        self.root = None

    def _size(self, node):
        if node is None:
            return 0
        else:
            return node.size_of_subtree

    def size(self):
        """
        Return the number of nodes in the BST

        Worst Case Complexity: O(1)

        Balanced Tree Complexity: O(1)
        """
        return self._size(self.root)

    def is_empty(self):
        """
        Returns True if the BST is empty, False
otherwise

        Worst Case Complexity: O(1)

        Balanced Tree Complexity: O(1)
        """
        return self.size() == 0

    def _get(self, key, node):
        if node is None:
            return None

        if key < node.key:
            return self._get(key, node.left)
        elif key > node.key:
            return self._get(key, node.right)
        else:
            return node.val
```

```
    def get(self, key):
        """
        Return the value paired with 'key'

        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(lg N)
        """
        return self._get(key, self.root)

    def contains(self, key):
        """
        Returns True if the BST contains 'key', False
otherwise

        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(lg N)
        """
        return self.get(key) is not None

    def _put(self, key, val, node):

        # If we hit the end of a branch, create a new
node
        if node is None:
            return Node(key, val)

        # Follow left branch
        if key < node.key:
            node.left = self._put(key, val,
node.left)
        # Follow right branch
        elif key > node.key:
            node.right = self._put(key, val,
node.right)
        # Overwrite value
        else:
```

```
            node.val = val

        node.size_of_subtree = self._size(node.left)
+ self._size(node.right)+1
        return node

    def put(self, key, val):
        """
        Add a new key-value pair.

        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(lg N)
        """
        self.root = self._put(key, val, self.root)

    def _min_node(self):
        """
        Return the node with the minimum key in the
BST
        """
        min_node = self.root
        # Return none if empty BST
        if min_node is None:
            return None

        while min_node.left is not None:
            min_node = min_node.left

        return min_node

    def min_key(self):
        """
        Return the minimum key in the BST

        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(lg N)
        """
```

```
        min_node = self._min_node()
        if min_node is None:
            return None
        else:
            return min_node.key

    def _max_node(self):
        """
        Return the node with the maximum key in the
BST
        """
        max_node = self.root
        # Return none if empty BST
        if max_node is None:
            return None

        while max_node.right is not None:
            max_node = max_node.right

        return max_node

    def max_key(self):
        """
        Return the maximum key in the BST

        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(lg N)
        """
        max_node = self._max_node()
        if max_node is None:
            return None
        else:
            return max_node.key

    def _floor_node(self, key, node):
        """
        Returns the node with the biggest key that is
less than or equal to the
```

```
        given value 'key'
        """
        if node is None:
            return None

        if key < node.key:
            # Floor must be in left subtree
            return self._floor_node(key, node.left)

        elif key > node.key:
            # Floor is either in right subtree or is
this node
            attempt_in_right = self._floor_node(key,
node.right)
            if attempt_in_right is None:
                return node
            else:
                return attempt_in_right

        else:
            # Keys are equal so floor is node with
this key
            return node

    def floor_key(self, key):
        """
        Returns the biggest key that is less than or
equal to the given value
        'key'

        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(lg N)
        """
        floor_node = self._floor_node(key, self.root)
        if floor_node is None:
            return None
        else:
            return floor_node.key
```

```
    def _ceiling_node(self, key, node):
        """
        Returns the node with the smallest key that
is greater than or equal to
        the given value 'key'
        """
        if node is None:
            return None

        if key < node.key:
            # Ceiling is either in left subtree or is
this node
            attempt_in_left = self._ceiling_node(key,
node.left)
            if attempt_in_left is None:
                return node
            else:
                return attempt_in_left
        elif key > node.key:
            # Ceiling must be in right subtree
            return self._ceiling_node(key,
node.right)
        else:
            # Keys are equal so ceiling is node with
this key
            return node

    def ceiling_key(self, key):
        """
        Returns the smallest key that is greater than
or equal to the given
        value 'key'

        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(lg N)
        """
        ceiling_node = self._ceiling_node(key,
```

```
self.root)
        if ceiling_node is None:
            return None
        else:
            return ceiling_node.key

    def _select_node(self, rank, node):
        """
        Return the node with rank equal to 'rank'
        """
        if node is None:
            return None

        left_size = self._size(node.left)
        if left_size < rank:
            return self._select_node(rank - left_size
- 1, node.right)
        elif left_size > rank:
            return self._select_node(rank, node.left)
        else:
            return node

    def select_key(self, rank):
        """
        Return the key with rank equal to 'rank'

        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(lg N)
        """
        select_node = self._select_node(rank,
self.root)
        if select_node is None:
            return None
        else:
            return select_node.key

    def _rank(self, key, node):
        if node is None:
```

```
            return None

        if key < node.key:
            return self._rank(key, node.left)
        elif key > node.key:
            return self._size(node.left) +
self._rank(key, node.right) + 1

        else:
            return self._size(node.left)

    def rank(self, key):
        """
        Return the number of keys less than a given
'key'.

        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(lg N)
        """
        return self._rank(key, self.root)

    def _delete(self, key, node):
        if node is None:
            return None
        if key < node.key:
            node.left = self._delete(key, node.left)
        elif key > node.key:
            node.right = self._delete(key,
node.right)

        else:
            if node.right is None:
                return node.left
            elif node.left is None:
                return node.right
            else:
                old_node = node
                node = self._ceiling_node(key,
```

```
node.right)
                node.right =
self._delete_min(old_node.right)
                node.left = old_node.left
        node.size_of_subtree = self._size(node.left)
+ self._size(node.right)+1
        return node

    def delete(self, key):
        """
        Remove the node with key equal to 'key'

        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(lg N)
        """
        self.root = self._delete(key, self.root)

    def _delete_min(self, node):
        if node.left is None:
            return node.right

        node.left = self._delete_min(node.left)
        node.size_of_subtree = self._size(node.left)
+ self._size(node.right)+1
        return node

    def delete_min(self):
        """
        Remove the key-value pair with the smallest
key.


        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(lg N)
        """
        self.root = self._delete_min(self.root)
```

```
    def _delete_max(self, node):
        if node.right is None:
            return node.left

        node.right = self._delete_max(node.right)
        node.size_of_subtree = self._size(node.left)
+ self._size(node.right)+1
        return node

    def delete_max(self):
        """
        Remove the key-value pair with the largest
key.

        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(lg N)
        """
        self.root = self._delete_max(self.root)

    def _keys(self, node, keys):
        if node is None:
            return keys

        if node.left is not None:
            keys = self._keys(node.left, keys)

        keys.append(node.key)

        if node.right is not None:
            keys = self._keys(node.right, keys)

        return keys

    def keys(self):
        """
        Return all of the keys in the BST in
aschending order
```

```
        Worst Case Complexity: O(N)

        Balanced Tree Complexity: O(N)
        """
        keys = []
        return self._keys(self.root, keys)
```