Name : JAY PATEL (RedID :824672518)

```python
from abc import abstractmethod, ABC
import unittest


# Base Class for defining the node and its functions
class Node(ABC):
    def __init__(self, key):
        pass

    @abstractmethod
    def insert_node(self, value):
        pass

    @abstractmethod
    def accept_node(self, traversal):
        pass


# Concrete Null Node Class for handling the null node in Tree
class NullNodeHandler(Node):

    # constructor has one argument which contains the strategy to be used
    def __init__(self, strategy_for_ordering):
        self.strategy_for_ordering = strategy_for_ordering

    # creates new Node and overright the null Node where needed
    def insert_node(self, value):
        return BinarySearchTree(value, self.strategy_for_ordering)

    def accept_node(self, traversal):
        pass


# Concrete Binary Search Tree for handling other nodes other than Null Nodes
class BinarySearchTree(Node):
    # the constructor takes 2 arguments: value of node, and strategy
    # it also defines the left and right node to be type of NullNode
    def __init__(self, key, strategy_for_ordering):
        self.value = key
        self.strategy_for_ordering = strategy_for_ordering
        self.left = NullNodeHandler(strategy_for_ordering)
        self.right = NullNodeHandler(strategy_for_ordering)

    # it adds the node to tree and uses NullObject Pattern for
    # implementing the Null Handler
    # the current tree node value and new node value are passed to Strategy Class
    def insert_node(self, value):
        if self.strategy_for_ordering.strategy_for_inserting(value, self.value):
            self.left = self.left.insert_node(value)
        else:
            self.right = self.right.insert_node(value)
        return self

    # this function defines the way for traversing the nodes in binary search tree
    def accept_node(self, traversal):
        return traversal.traverse_tree(self)


# abstract class for implementing Strategy Pattern
class Strategy(ABC):
    def __init__(self):
        pass

    @abstractmethod
    def strategy_for_inserting(self):
        pass


# Concrete Strategy Class which defines the way of normal lexicographic order
class Sorting(Strategy):
    def __init__(self):
        pass

    # it compares the string of new node and returns true if new node has lower value
    # compared to the value of current node in the tree
    def strategy_for_inserting(self, value_of_node, value_in_tree):
        if value_of_node < value_in_tree:
            return True
        else:
            return False


# Concrete Strategy class which defines the way of reverse lexicographic order
class ReverseSorting(Strategy):
    def __init__(self):
        pass

    # it compares the reverse string of new node and returns true if new node has
    # lower value compared to the reverse value of current node in tree
    def strategy_for_inserting(self, value_of_node, value_in_tree):
        if value_of_node[::-1] < value_in_tree[::-1]:
            return True
        else:
            return False
```

Name : JAY PATEL (RedID :824672518)

```
# Base class for implementing Visitor Pattern
class Traversal(ABC):
    def __init__(self):
        pass

    @abstractmethod
    def traverse_tree(self, node):
        pass


# Concrete Class for traversing the Binary Search Tree
class PreOrderTraversal(Traversal):
    def __init__(self):
        pass

    # it checks whether current node is of type NullNodeHandler
    # if yes then will return something, otherwise something different
    def traverse_tree(self, node):
        if isinstance(node, NullNodeHandler):
            return "()"
        else:
            return "(" + node.value + self.traverse_tree(node.left) \
                   + self.traverse_tree(node.right) + ")"


# UNIT TESTING

# Test the validation of inserting the node in the Binary Tree
class InsertingNodeClass(unittest.TestCase):
    def test_insert_node(self):
        sorting = Sorting()
        node = BinarySearchTree("xyz", sorting)
        self.assertEqual("xyz", node.value)


# Test wheather the visitor pattern works and display perfectly or not
class DisplayingNodes(unittest.TestCase):
    def test_display_node(self):
        sorting = Sorting()
        node = BinarySearchTree("xyz", sorting)
        node.insert_node("abc")
        node.insert_node("zz")
        pre_order = PreOrderTraversal()
        Traversing = node.accept_node(pre_order)
        self.assertEqual("(xyz(abc()())(zz()()))", Traversing)

# checks if the Null Node Handler works and returns BinaryTreeNode Object
class TypeOfNode(unittest.TestCase):
    def test_type_of_node(self):
        sorting = Sorting()
        node = BinarySearchTree("xyz", sorting)
        self.assertEqual(str, type(node.value))
        self.assertEqual(NullNodeHandler, type(node.left))
        self.assertEqual(NullNodeHandler, type(node.right))


# Test the one of the Strategy that is been implemented via ReverseSorting
class ReverseSortingClass(unittest.TestCase):
    def test_sorting(self):
        sorting = ReverseSorting()
        node = BinarySearchTree("xyz", sorting)
        node.insert_node("az")
        node.insert_node("zz")
        pre_order = PreOrderTraversal()
        Traversing = node.accept_node(pre_order)
        self.assertEqual("(xyz(abc()())(zz()()))", Traversing)


# Test the one of the Strategy that is been implemented via ReverseSorting
class SortingClass(unittest.TestCase):
    def test_sorting(self):
        sorting = Sorting()
        node = BinarySearchTree("xyz", sorting)
        node.insert_node("abc")
        node.insert_node("zz")
        pre_order = PreOrderTraversal()
        Traversing = node.accept_node(pre_order)
        self.assertEqual("(xyz(abc()())(zz()()))", Traversing)d


"""
Unit Testing Driver Program
"""
def main():
    # UnitTest start
    unittest.main()


if __name__ == "__main__":
    main()
```