- **RED and BLACK tree:**

```
class Node:
    def __init__(self, value, color='red'):
        self.value = value
        self.color = color
        self.left = None
        self.right = None
        self.parent = None

class RedBlackTree:
    def __init__(self):
        self.NIL = Node(None, color='black')
        self.root = self.NIL

    def insert(self, value):
        new_node = Node(value)
        new_node.left = self.NIL
        new_node.right = self.NIL

        y = None
        x = self.root

        while x != self.NIL:
            y = x
            if new_node.value < x.value:
                x = x.left
            else:
                x = x.right

        new_node.parent = y

        if y is None:
            self.root = new_node
        elif new_node.value < y.value:
            y.left = new_node
        else:
            y.right = new_node

        new_node.color = 'red'
        self.fix_insert(new_node)

    def fix_insert(self, node):
        while node != self.root and node.parent.color == 'red':
            if node.parent == node.parent.parent.left:
                y = node.parent.parent.right
                if y.color == 'red':
                    node.parent.color = 'black'
                    y.color = 'black'
                    node.parent.parent.color = 'red'
                    node = node.parent.parent
                else:
                    if node == node.parent.right:
                        node = node.parent
                        self.left_rotate(node)
                    node.parent.color = 'black'
                    node.parent.parent.color = 'red'
                    self.right_rotate(node.parent.parent)
            else:
                y = node.parent.parent.left
                if y.color == 'red':
                    node.parent.color = 'black'
                    y.color = 'black'
```

```
            node.parent.parent.color = 'red'
                    node = node.parent.parent
                else:
                    if node == node.parent.left:
                        node = node.parent
                        self.right_rotate(node)
                    node.parent.color = 'black'
                    node.parent.parent.color = 'red'
                    self.left_rotate(node.parent.parent)
        self.root.color = 'black'
    def left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left != self.NIL:
            y.left.parent = x
        y.parent = x.parent
        if x.parent is None:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y
    def right_rotate(self, x):
        y = x.left
        x.left = y.right
        if y.right != self.NIL:
            y.right.parent = x
        y.parent = x.parent
        if x.parent is None:
            self.root = y
        elif x == x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y
        y.right = x
        x.parent = y

    def inorder_traversal(self, node, result=None):
        if result is None:
            result = []
        if node != self.NIL:
            self.inorder_traversal(node.left, result)
            result.append((node.value, node.color))
            self.inorder_traversal(node.right, result)
        return result

    def display_tree(self):
        result = self.inorder_traversal(self.root)
        for value, color in result:
            print(f"Value: {value}, Color: {color}")

# Example usage:
rbt = RedBlackTree()
values = [20, 15, 25, 10, 5, 1]
for value in values:
    rbt.insert(value)

rbt.display_tree()
```

- **Output:**

```
Value: 1, Color: red
```

Value: 5, Color: black
Value: 10, Color: red
Value: 15, Color: black
Value: 20, Color: black
Value: 25, Color: black

- **SPLAY:**

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.parent = None

class SplayTree:
    def __init__(self):
        self.root = None

    def _right_rotate(self, x):
        y = x.left
        x.left = y.right
        if y.right:
            y.right.parent = x
        y.parent = x.parent
        if not x.parent:
            self.root = y
        elif x == x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y
        y.right = x
        x.parent = y

    def _left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left:
            y.left.parent = x
        y.parent = x.parent
        if not x.parent:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y

    def _splay(self, node):
        while node.parent:
            if node.parent.parent is None:
                if node == node.parent.left:
                    self._right_rotate(node.parent)
                else:
                    self._left_rotate(node.parent)
            elif node == node.parent.left and node.parent == node.parent.parent.left:
                self._right_rotate(node.parent.parent)
                self._right_rotate(node.parent)
            elif node == node.parent.right and node.parent == node.parent.parent.right:
                self._left_rotate(node.parent.parent)
                self._left_rotate(node.parent)
            else:
                if node == node.parent.left:
                    self._right_rotate(node.parent)
                self._left_rotate(node.parent)

    def insert(self, value):
        if not self.root:
            self.root = Node(value)
            return
```

```
        node = self._insert(self.root, value)
        self._splay(node)

    def _insert(self, root, value):
        if value < root.value:
            if root.left:
                return self._insert(root.left, value)
            else:
                root.left = Node(value)
                root.left.parent = root
                return root.left
        else:
            if root.right:
                return self._insert(root.right, value)
            else:
                root.right = Node(value)
                root.right.parent = root
                return root.right

    def search(self, value):
        node = self._search(self.root, value)
        if node:
            self._splay(node)
        return node is not None

    def _search(self, root, value):
        if not root or root.value == value:
            return root
        if value < root.value:
            return self._search(root.left, value)
        return self._search(root.right, value)

    def inorder_traversal(self, node, result=None):
        if result is None:
            result = []
        if node:
            self.inorder_traversal(node.left, result)
            result.append(node.value)
            self.inorder_traversal(node.right, result)
        return result

    def display_tree(self):
        print("Inorder traversal:", self.inorder_traversal(self.root))

# Example usage:
splay_tree = SplayTree()
for value in [10, 20, 30, 40, 50]:
    splay_tree.insert(value)

splay_tree.display_tree()
splay_tree.search(30)
splay_tree.display_tree()
```

- **Output:**

Inorder traversal: [10, 20, 30, 40, 50]
Inorder traversal: [10, 20, 30, 40, 50]