explain below code of insertion in binary search tree by visualization:

define a binary search tree node

class Node:
def **init**(self, key):
self.left = None
self.right = None
self.val = key

define a function to insert a new node with the given key

def insert(root, key):
if root is None:
return Node(key)
if root.val == key:
return root
if root.val < key:
root.right = insert(root.right, key)
else:
root.left = insert(root.left, key)

```
return root
```

def inorder(root):
if root:
inorder(root.left)
print(root.val, end=' ')
inorder(root.right)

Here's a step-by-step explanation of BST insertion with visual examples:

Binary Search Tree Insertion Process

1. Node Structure

```
class Node:
    def __init__(self, key):
        self.left = None   # Left child
        self.right = None  # Right child
        self.val = key     # Node value
```

2. Insertion Logic

```
def insert(root, key):
    if root is None:          # Base case: Empty spot found
        return Node(key)
    if root.val == key:       # Duplicate: Do nothing
        return root
    if root.val < key:        # Go right if key is larger
        root.right = insert(root.right, key)
    else:                     # Go left if key is smaller
        root.left = insert(root.left, key)
    return root
```

3. Visualization Example

Let's insert values

Step 1: Insert 50 (root)

```
    50
```

Step 2: Insert 30 (<50 → left)

```
    50
   /
 30
```

Step 3: Insert 70 (>50 → right)

```
    50
   /  \
 30   70
```

Step 4: Insert 20 (<30 → left of 30)

```
    50
   /  \
 30   70
 /
20
```

Step 5: Insert 40 (>30 → right of 30)

```
    50
   /  \
 30   70
/  \
20 40
```

Step 6: Insert 60 (<70 → left of 70)

```
    50
   /  \
 30   70
/  \  /
20 40 60
```

Step 7: Insert 80 (>70 → right of 70)

```
    50
   /  \
 30   70
/  \  / \
20 40 60 80
```

4. Inorder Traversal

```
def inorder(root):
    if root:
        inorder(root.left)
        print(root.val, end=' ')
        inorder(root.right)
```

Output: `20 30 40 50 60 70 80` (Sorted order)

5. Key Characteristics

- **Left Subtree:** Always contains smaller values
- **Right Subtree:** Always contains larger values
- **Time Complexity:** O(h) where h = tree height
- **Duplicates:** Ignored (existing value returns same node)

6. Recursion Flow (Insert 60 in final tree)

```
insert(50,60) → 60>50 → call insert(70,60)
               insert(70,60) → 60<70 → call insert(None,60)
               create node(60) ← returns to 70.left
returns updated tree
```

This structure maintains efficient search (O(log n) in balanced trees) while allowing dynamic updates! ▯