Name: Jigar Siddhpura **SAPID:** 60004200155

DIV: C/C2 **Branch:** Computer Engineering

AA - Experiment 5 - KD Tree

60004210155
Jigar Siddhpura
C22

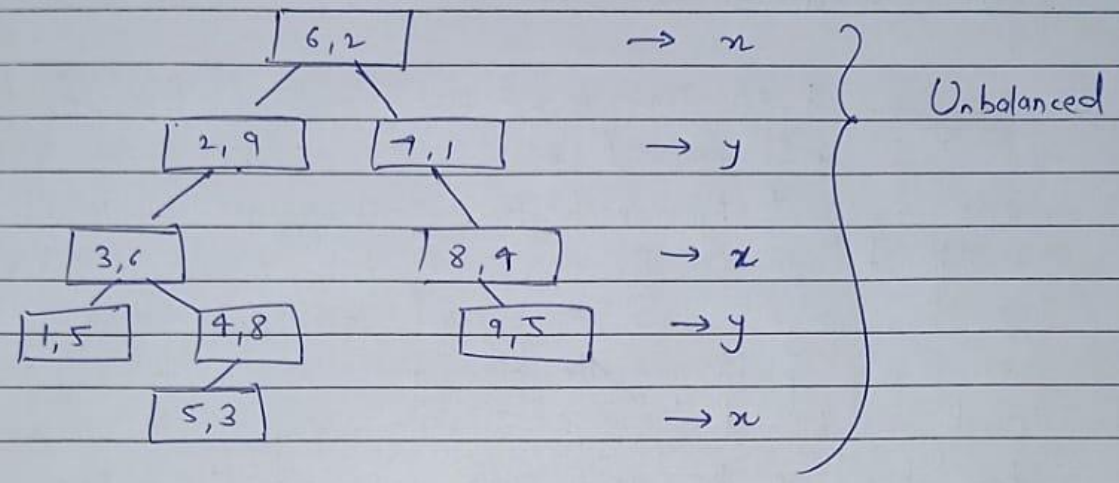
AA - Experiment 6 - KD Tree

Aim: To implement KD Tree

Theory: A KD-Tree is a binary Tree used for efficient multidimensional search. It partitions the space into axis-aligned hyperrectangles, splitting at each level along a different axis. This allows for fast nearest neighbor search, range search & spatial queries. Construction involves recursively partitioning the data along the median of each dimension, creating a balanced tree. Searching involves traversing the tree based on the query points co-ordinates, pruning branches cannot contain the nearest neighbors. KD trees are beneficial for high dimensional data, reducing search time compared to linear search. However they become inefficient with skewed data distribution / high dimensions.

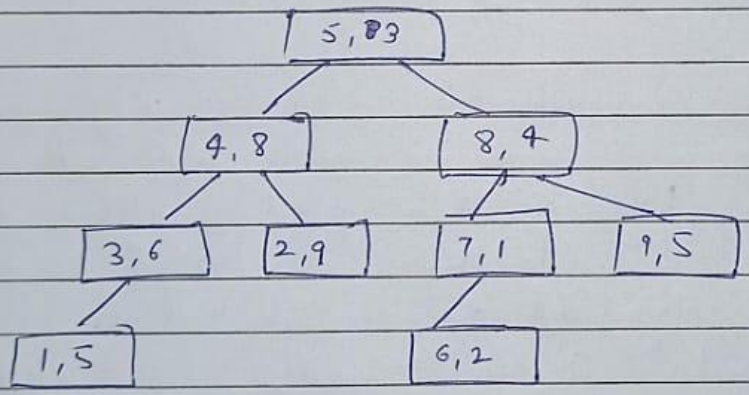
The first code inserts points into a KD Tree using a specified axis for splitting. It recursively inserts points into left or right subtree based on their position relative to their current node. 2ND code builds a balanced KD Tree by sorting pts along diff axes at each level & choosing the medians as pivot. It can recursively construct left & right subtree. Both codes allow for efficient spatial search operations on multidimensional data.

Q- (6,2), (7,1), (2,9), (3,6), (9,8), (8,9), (5,3), (1,5), (9,5)



Right bias (balanced)

~~(1,5), (3,6), (~~



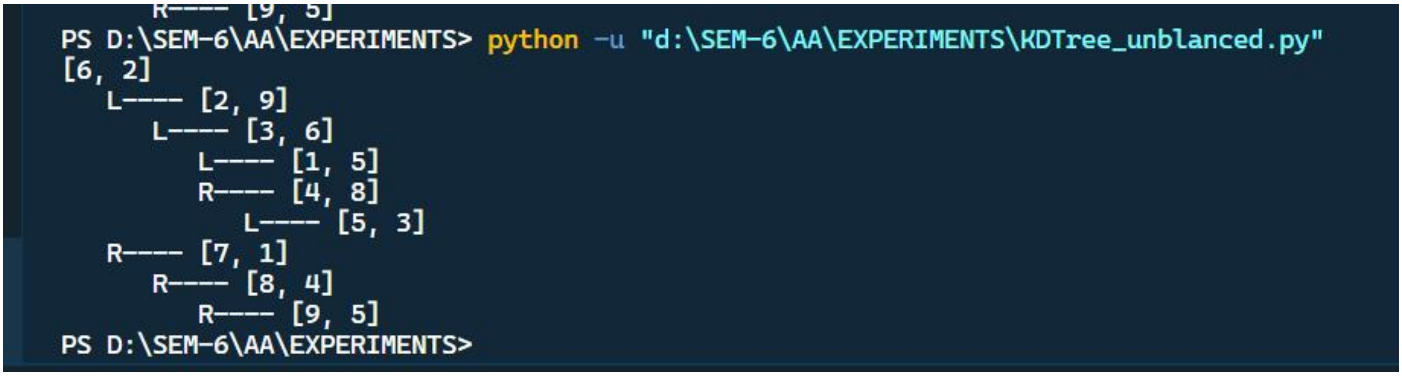
Conclusion: Thus we implemented KD Tree.

CODE :

UNBALANCED KD TREE:

```
class Node:
    def __init__(self, point, axis):
        self.point = point
        self.axis = axis
        self.left = None
        self.right = None

def insert(root, point, axis=0):
    if root is None:
        return Node(point, axis)
    if point[axis] < root.point[axis]:
        root.left = insert(root.left, point, (axis + 1) % len(point))
    else:
        root.right = insert(root.right, point, (axis + 1) % len(point))
    return root

def print_tree(node, level=0, side=None):
    if node is not None:
        prefix = ""
        if side is not None:
            prefix = side + "---- "
        print("    " * level + prefix + str(node.point))
        print_tree(node.left, level + 1, "L")
        print_tree(node.right, level + 1, "R")

# Sample data points
points = [[6, 2], [7, 1], [2, 9], [3, 6], [4, 8], [8, 4], [5, 3], [1, 5], [9, 5]]
# points = [[6, 2, 9], [7, 1, 2], [2, 9, 6], [3, 6, 1], [4, 8, 5], [8, 4, 4], [5, 3, 7], [1, 5, 1], [9, 5, 4]]

# Build KDTree
root = None
for point in points:
    root = insert(root, point)

# Print the tree
print_tree(root)
```

OUTPUT :



BALANCED KD TREE (Right biased):

```
class Node:
    def __init__(self, point, axis):
        self.point = point
        self.axis = axis
        self.left = None
        self.right = None

def build_kdtree(points, depth=0):
    if not points:
        return None

    # Select axis based on depth so that axis cycles through all valid values
    k = len(points[0])  # Dimension of the points
    axis = depth % k

    # Sort points based on the axis and choose median as pivot element
    points.sort(key=lambda x: x[axis])
    median = len(points) // 2

    # Create node and construct subtrees
    node = Node(points[median], axis)
    node.left = build_kdtree(points[:median], depth + 1)
    node.right = build_kdtree(points[median + 1:], depth + 1)

    return node

def print_tree(node, level=0, side=None):
    if node is not None:
        prefix = ""
        if side is not None:
            prefix = side + "---- "
        print("    " * level + prefix + str(node.point))
        print_tree(node.left, level + 1, "L")
        print_tree(node.right, level + 1, "R")
```

```
# Sample data points
points = [[6, 2], [7, 1], [2, 9], [3, 6], [4, 8], [8, 4], [5, 8], [1, 5], [9, 5]]
# points = [[6, 2, 9], [7, 1, 2], [2, 9, 6], [3, 6, 1], [4, 8, 5], [8, 4, 4], [5, 3, 7], [1, 5, 1], [9, 5, 4]]
# Build balanced KDTree
root = build_kdtree(points)
# Print the tree
print_tree(root)
```

OUTPUT :