**K-D tree**

1.

This code uses namedtuple to define the point class, inherits the Rectangle from namedtuple, and also inherits the Node class and KDTree class from namedtuple. The Point class and Rectangle class have the repr method to return the string. Rectangle has an is\_ Contains method to return the object of Point as a Boolean value and determine whether it is within the rectangle. The Node class has three attributes: location, left, and right. The KDTree class has methods to insert a list of points and search the range of points within a rectangle. Then define range\_ Test function to test the range search method. Performance\_test is defined to test the time required for KDTree method to perform range search.

2.

The insert() method is to add points to the K-D tree. The method of median recursion is used to divide points. The first step is to determine the current depth and dimension, such as x and y, based on recursion. Then sort to find the median point, and take the median point as the root of the current subtree, and then use the insert method to create the left and right subtrees through the recursive call.

def insert(self, p: List[Point], depth=0):

if not p:

return None

dimension = depth % 2

p.sort(key=lambda x: x[dimension])

median = len(p) // 2

left\_subtree = self.insert(p[:median], depth+1)

right\_subtree = self.insert(p[median+1:], depth+1)

node = Node(location=p[median], left=left\_subtree, right=right\_subtree)

return node

Range() is used to find points within the rectangle. Search the K-D tree recursively from the root. First, for each node in the tree, check whether it contains this point. Second, check whether it is on the left side. If it is, search the left subtree recursively. If it is on the right side, search the right subtree recursively.

def range(self, rectangle: Rectangle, node=None) -> List[Point]:

if node is None:

node = self.\_root

if node is None:

return []

result = []

if rectangle.is\_contains(node.location):

result.append(node.location)

if rectangle.lower.x <= node.location.x:

result += self.range(rectangle, node.left)

if rectangle.upper.x >= node.location.x:

result += self.range(rectangle, node.right)

return result

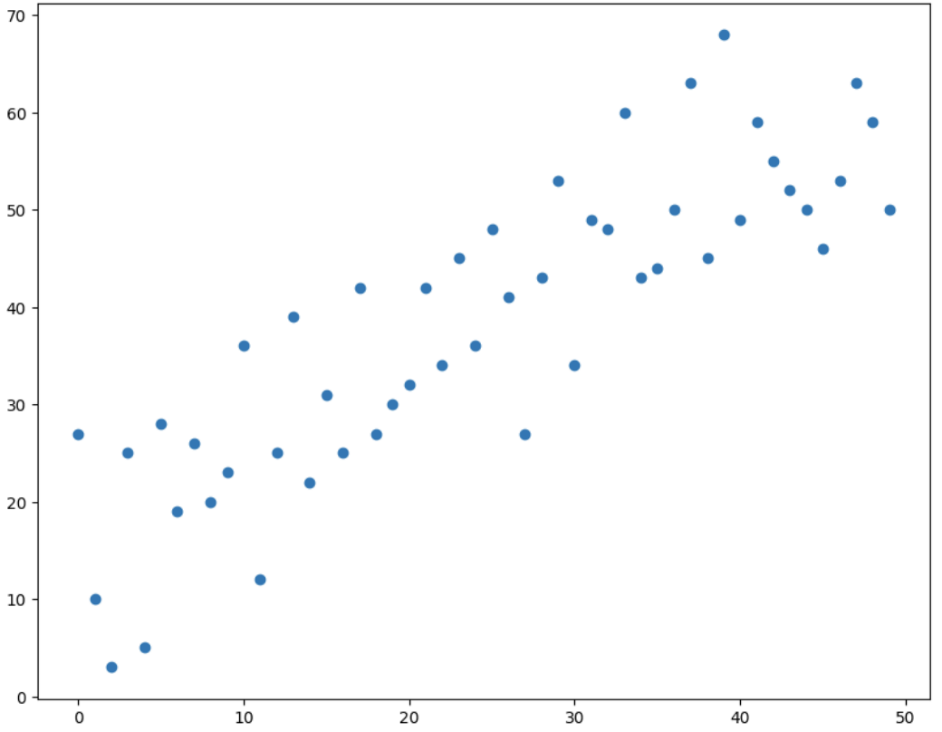
3.

In the worst case, all points are on the same line, the tree becomes a linked list, the time complexity is O(n).

In the best case, when the tree is balanced, the number of left and right subtrees of each node is roughly equal, the time complexity is O(log n).

4.

Create a function to measure the time required to execute the range query using these two methods in the point set. Then we can use the plug-in of python to draw the image, and use the x-axis and y-axis to represent the number of points and the time required.



5.

We can build a function to find the nearest point to the current node, and then enter the while loop to keep going except for None. Check whether the current point is smaller than the node. If yes, set it to the left child node, otherwise set it to the right child node. After updating the node, the function will check whether the distance between the point and the current node position is smaller than the distance between the point and the current optimal node. If yes, the function updates the nearest point to the position of the current node. Once the loop exits, the function returns the nearest point.

def nearestneighbor(self, point: Point, node=None):

if node is None:

node = self.\_root

if node is None:

return None

nearest = node.location

while node:

dimension = depth % 2

if point[dimension] < node.location[dimension]:

node = node.left

else:

node = node.right

if node and self.distance(point, node.location) < self.distance(point, best):

nearest = node.location

return nearest