# Implementation of KD Trees and KNN Algorithm

Presented by:

Team Unsupervised Learning

[Atharv Srivastava (2022MEB1301)

Arjun Rana (2022EEB1370)

Atharva Sharma (2022MEB1302)]

Instructor: Dr. Swapnil Dhamal

Teaching Assistant: Miss Shradha Sharma

# Introduction

- **What is a KD Tree?**
- A KD (K-dimensional) tree is a data structure used for organizing and searching points in a multi-dimensional space.
- It’s a binary tree that partitions space into half-spaces, useful for efficiently indexing k-dimensional data.
- **Applications:**
- Used in computer graphics, vision, data mining, spatial databases, and other fields.

# Problem and Objectives

- **Problem Statement:**
- With the growth of digital data, efficiently retrieving relevant information has become challenging.
- Nearest Neighbor (NN) search is a simple, effective way to locate data points closest to a query in high-dimensional space.

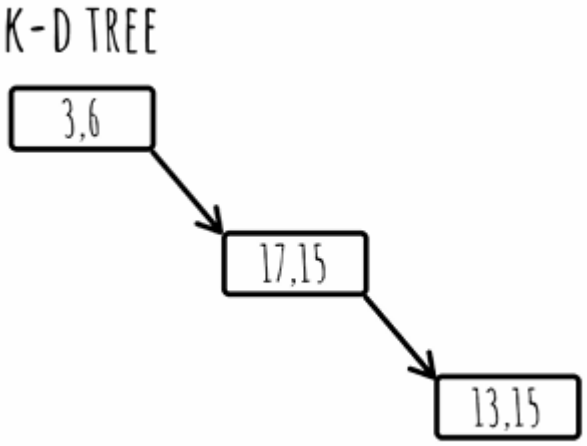
- **Objectives:**
- Implement the KD Tree data structure.
- Perform nearest neighbor searches for efficient data retrieval.

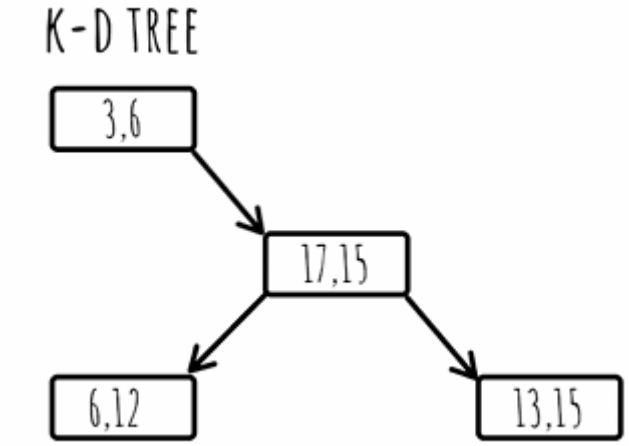
# Insertion and Search in KD Trees

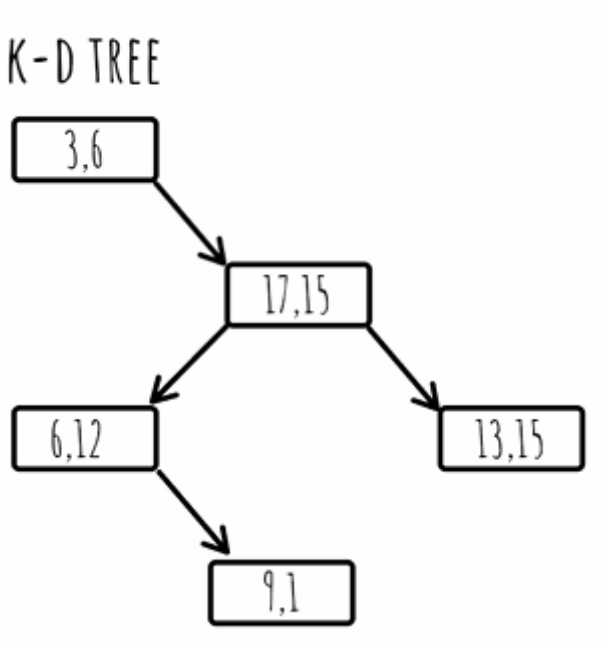
- **Insertion:**
- The tree is partitioned based on alternating dimensions (e.g., x and y for a 2D KD Tree).
- Each node splits space based on its dimension's coordinate value, creating left and right subtrees.
- **Search:**
- Traverses similar to a binary tree, switching between dimensions at each level.
- Efficiently reduces search space by excluding large portions of irrelevant data points.

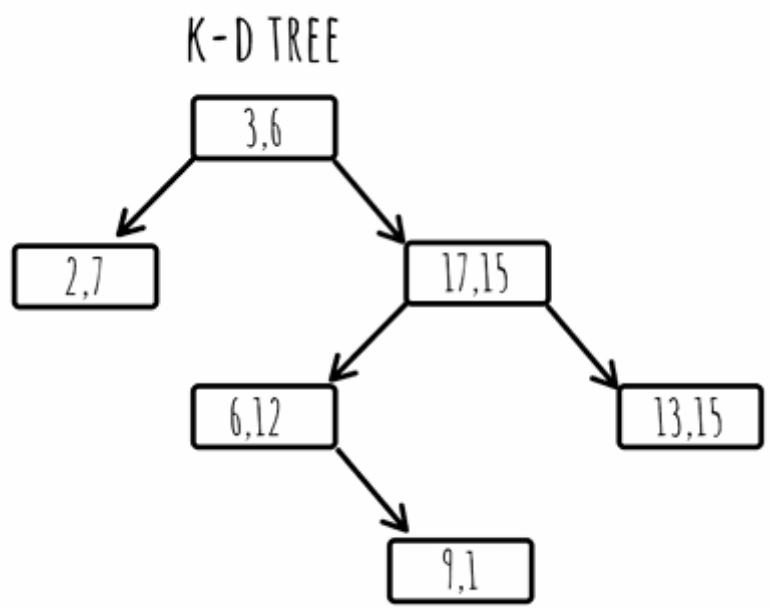
Consider an example of insert in the 2-D plane:
(3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19)

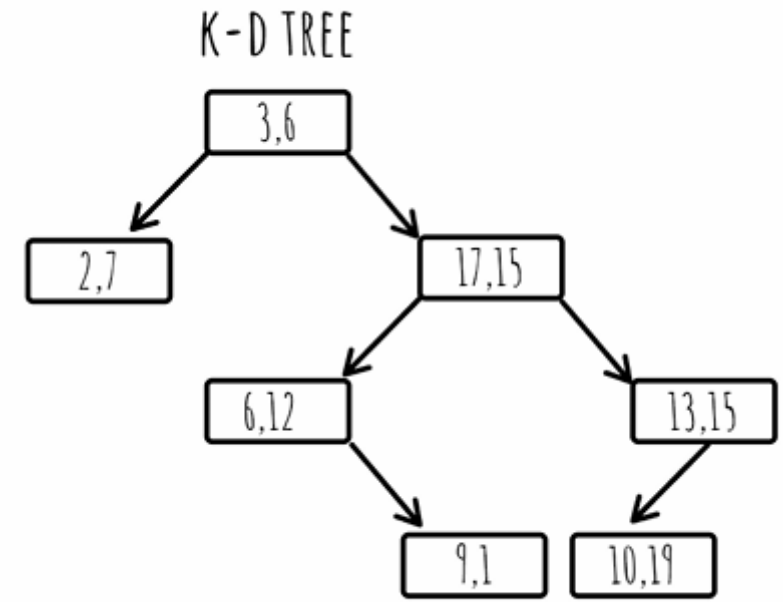





K-D TREE
3,6
17,15
6,12
13,15
9,1


K-D TREE
3,6
2,7
17,15
6,12
13,15
9,1


K-D TREE
3,6
2,7
17,15
6,12
13,15
9,1
10,19

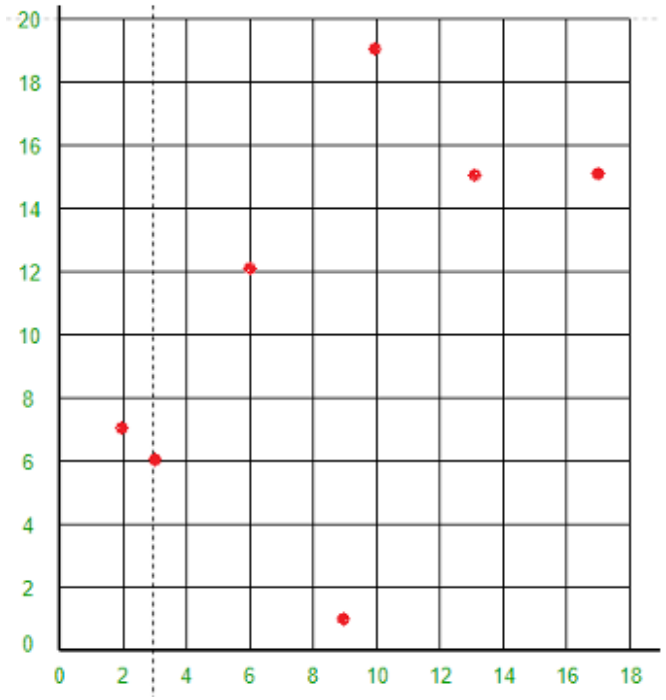
# Insertion Pseudo code

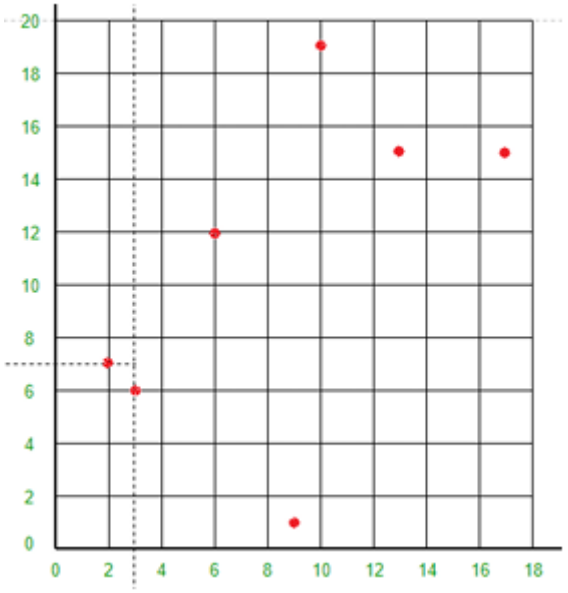
**Algorithm 1** Insertion

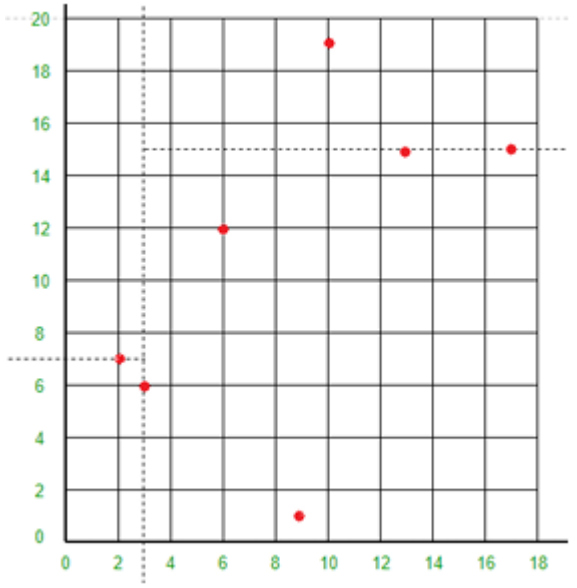
```
1: node* Insert(node* x,int point[ ],int depth)
2: if x is NULL then
3:    temp=Allocate memory()
4:    for i = 0 upto k − 1 do
5:       temp.point_i=point_i
6:    end for
7:    x=temp
8:    return temp
9: end if
10: if point_{depth%k}<x.point_{depth%k} then
11:    return Insert(x.left,point,depth)
12: else
13:    return Insert(x.right,point,depth)
14: end if
15: return x
```

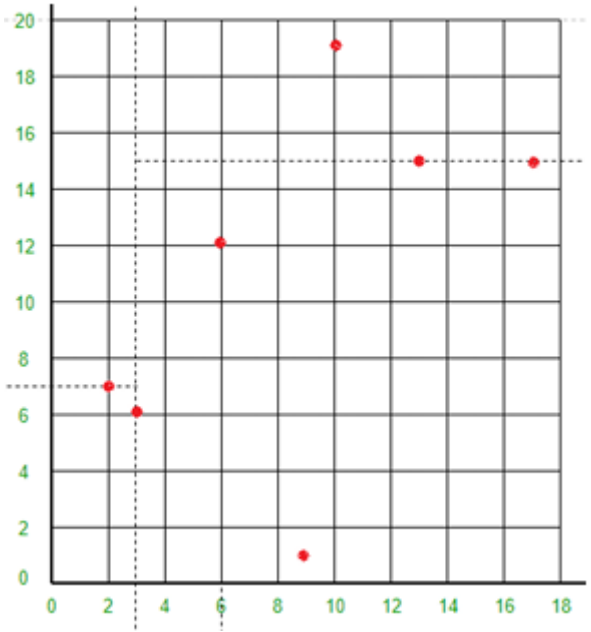
It has an average of O(log n) time complexity and O(n) space complexity.

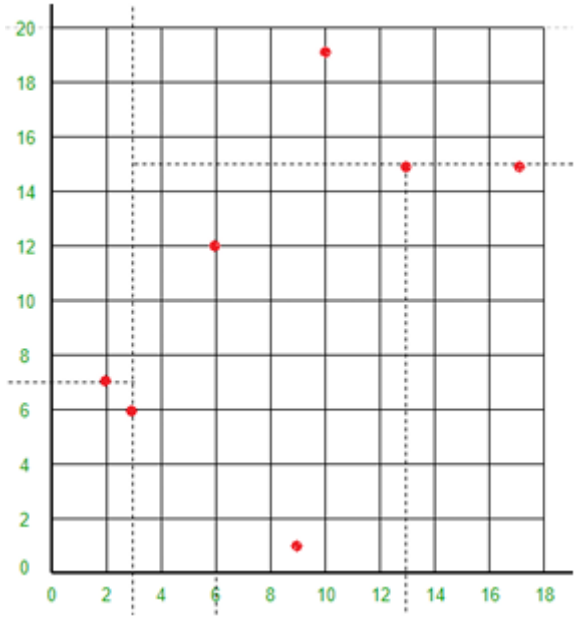
# Search

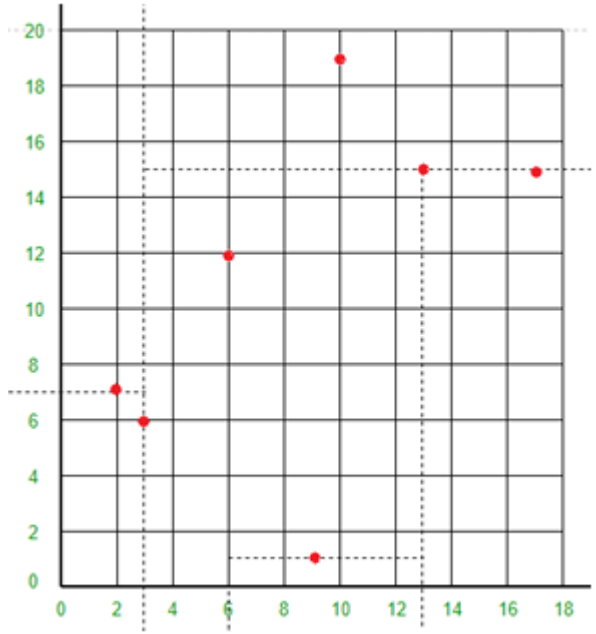

20
18
16
14
12
10
8
6
4
2
0
0 2 4 6 8 10 12 14 16 18

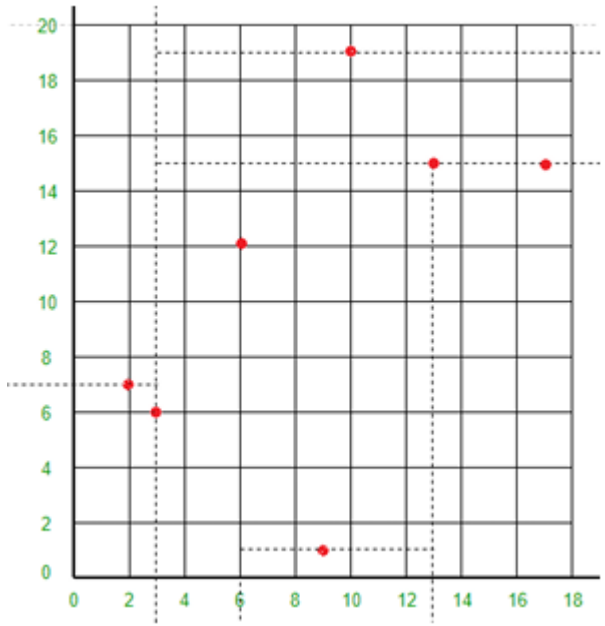

20
18
16
14
12
10
8
6
4
2
0
0 2 4 6 8 10 12 14 16 18


20
18
16
14
12
10
8
6
4
2
0
0 2 4 6 8 10 12 14 16 18


0 2 4 6 8 10 12 14 16 18
0 2 4 6 8 10 12 14 16 18 20


0 2 4 6 8 10 12 14 16 18
0 2 4 6 8 10 12 14 16 18 20


0 2 4 6 8 10 12 14 16 18
0 2 4 6 8 10 12 14 16 18 20


0 2 4 6 8 10 12 14 16 18
0 2 4 6 8 10 12 14 16 18 20

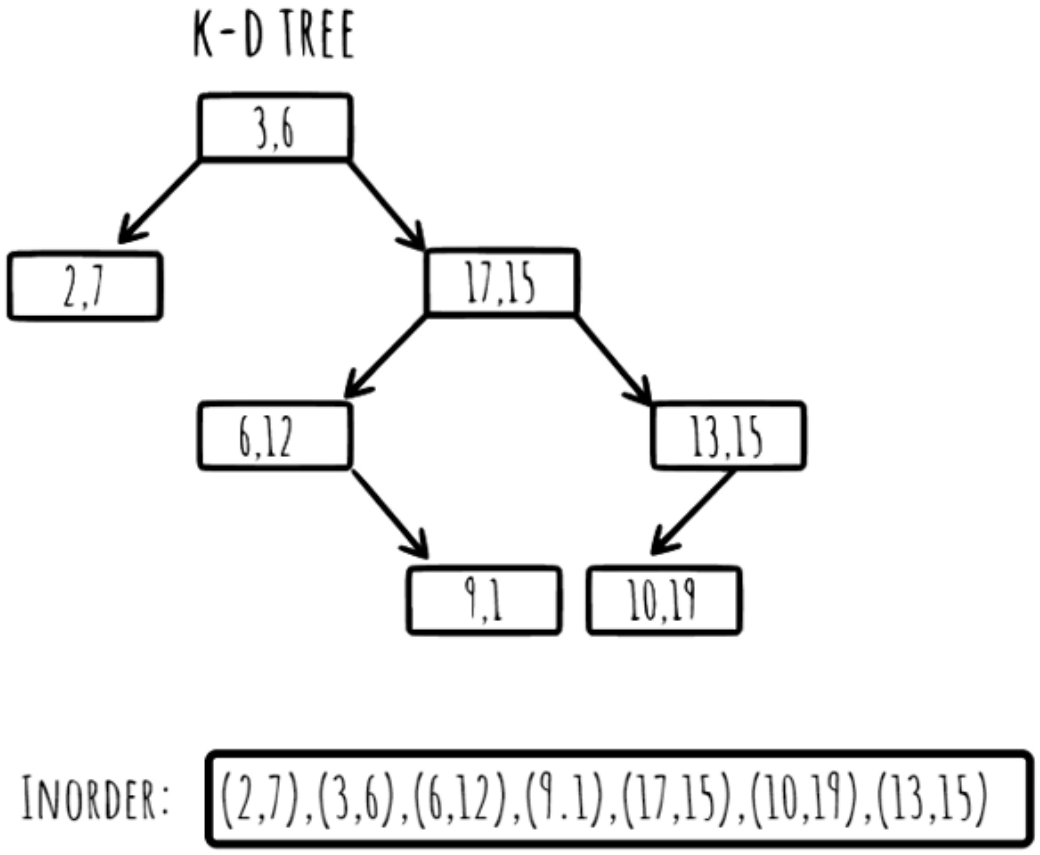
# Search Pseudocode

Algorithm 2 Search

```
1: node* Search(node* x,int point[ ],int depth)
2: if x is NULL then
3:    return NULL
4: end if
5: if point_{depth%k}=x.point_{depth%k} then
6:    if point=x.point then
7:       return x
8:    else if point_{(depth+1)%k}<x.point_{(depth+1)%k} then
9:       return Search(x.left,point,depth+1)
10:   else
11:      return Search(x.right,point,depth+1
12:   end if
13: else if point_{depth%k}<x.point_{depth%k} then
14:   return Search(x.left,point,depth)
15: else
16:   return Search(x.right,point,depth)
17: end if
```

It has an average of O(log n) time complexity.

# Traversal

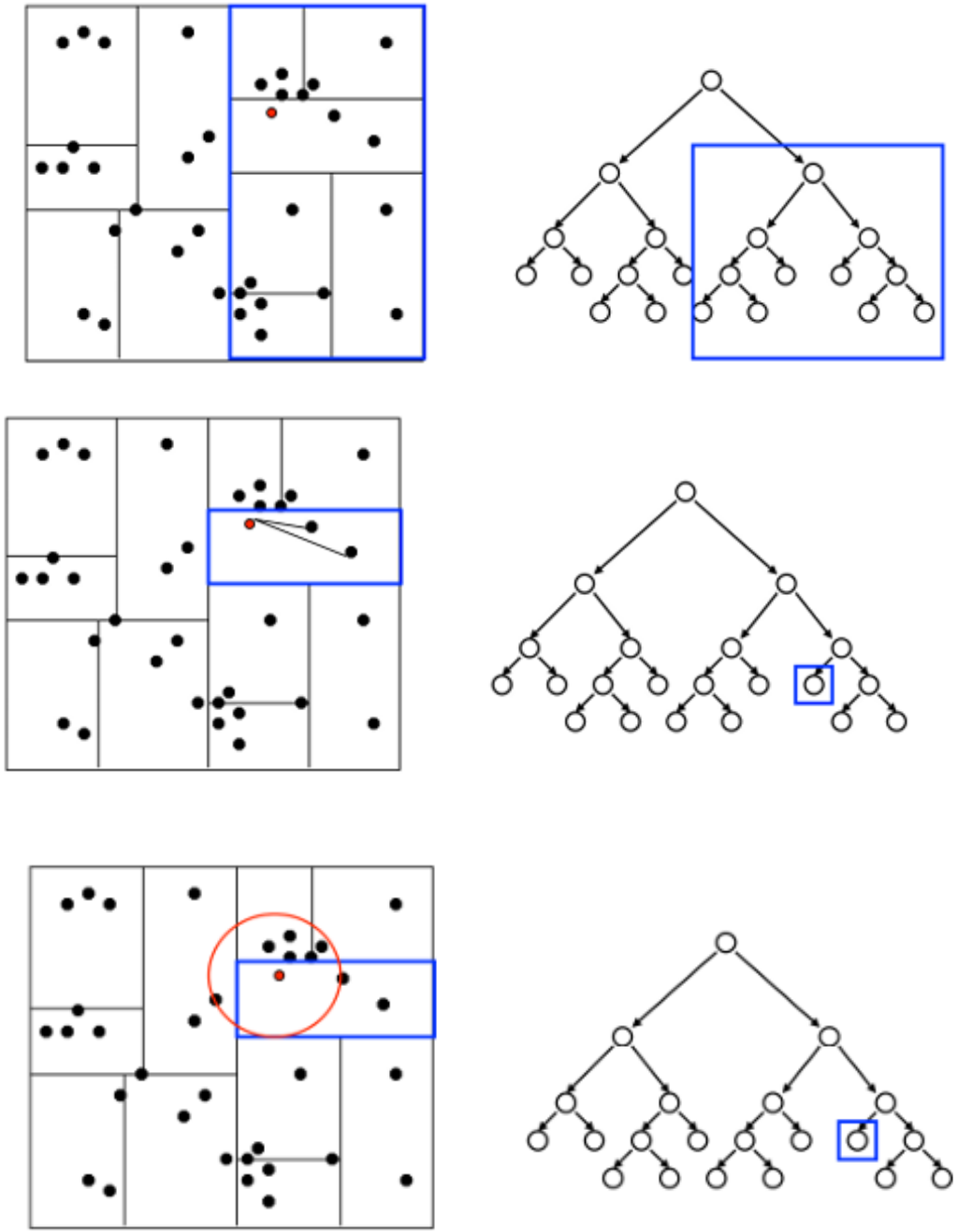



Algorithm 3 Inorder Traversal

```
1: void Inorder(node* x)
2: if x is NULL then
3:     return NULL
4: end if
5: Inorder(x.left)
6: print x.point
7: Inorder(x.right)
```

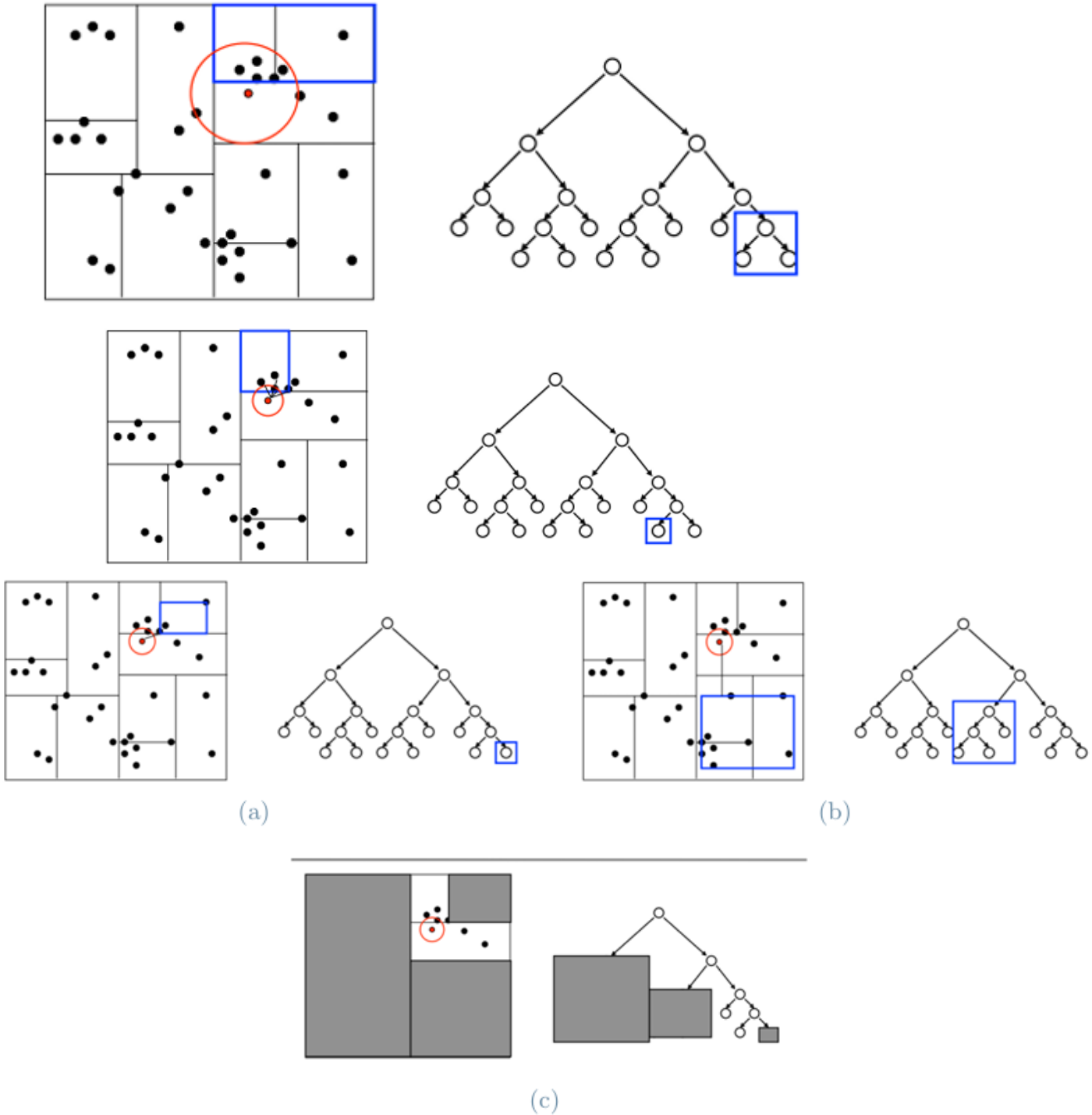
# Nearest Neighbour

- 1.We traverse the tree looking for the nearest neighbor of the query point.
- 2.Explore the branch of the tree that is closest to the query point first.
- 3.When we reach a leaf node, compute the distance to each point in the node.



# Nearest Neighbour

- 4.Then we can backtrack and try the other branch at each node visited.
- 5.Each time a new closest node is found, we can update the distance bounds.
- 6.Using the distance bounds and the bounds of the data below each node, we can prune parts of the tree that could NOT include the nearest neighbour.



# Time Complexity Analysis

- **Insertion**
  - Similar to a Binary Tree: Compares with root and inserts in left/right subtree.
  - **Time Complexity:** O(logN) on average, where N is the number of nodes.
- **Search**
  - Similar to insertion; traverses tree to locate point if it exists.
  - **Time Complexity:** O(logN) on average.
- **Traversal**
  - In-order traversal visits each node once.
  - **Time Complexity:** O(N).
- **Nearest Neighbor (NNS)**
  - Traverses to leaf node, backtracks for pruning.
  - **Complexity:** Ranges from O(logN) (best case, with pruning) to O(N) (worst case, no pruning).

# Test Case