



THE UNIVERSITY  
of ADELAIDE



CRICOS PROVIDER 00123M

Faculty of SET / School of Computer and Mathematical Sciences

# COMP SCI 3007&7059 Artificial Intelligence Efficient KNN

adelaide.edu.au

*seek* LIGHT

# Nearest Neighbour Search

- Given a set of points  $X = \{x_i\}_{i=1}^N$  and a query point  $\mathbf{z}$ , both in  $M$ -dimension space, find the nearest neighbour of  $\mathbf{z}$  among  $X$ .
  - The concept of “nearness” is based on a pre-defined distance metric, usually the Euclidean distance.
  - A naïve search algorithm scans each and every point in  $X$ , incurring a computational cost of  $O(MN)$ . This is not scalable to large  $N$ 's (realistically  $N$  can be in the range of millions).
-

# Efficient Nearest Neighbour Search

- How to improve the search efficiency?
    - Exact methods:
      - K-d tree
    - Approximate methods
      - Locality sensitive hashing
      - Vector quantization
      - Randomized k-d tree forest
-

# Exact method: General idea

- Recall binary search
  - Find exact match
  - Direct search requires  $O(N)$  calculation
  - Binary search reduces the computational cost

Input: 1,15,5,2,3,7,19

---

# Exact method: General idea

- Recall binary search
  - Find exact match
  - Direct search requires  $O(N)$  calculation
  - Binary search reduces the computational cost

Input: 1,15,5,2,3,7,19

sort: 1,2,3,5,7,15,19

---



# Exact method: General idea

- Recall binary search
  - Find exact match
  - Direct search requires  $O(N)$  calculation
  - Binary search reduces the computational cost

Input: 1,15,5,2,3,7,19

sort: 1,2,3,5,7,15,19

3? : < 5

---

# Exact method: General idea

- Recall binary search
  - Find exact match
  - Direct search requires  $O(N)$  calculation
  - Binary search reduces the computational cost

Input: 1,15,5,2,3,7,19

sort: 1,2,3,5,7,15,19

3? : < 5

3? : > 2

---

# Exact method: General idea

- Recall binary search
  - Find exact match
  - Direct search requires  $O(N)$  calculation
  - Binary search reduces the computational cost

Input: 1,15,5,2,3,7,19

sort: 1,2,3,5,7,15,19

3? : < 5

3? : > 2

---

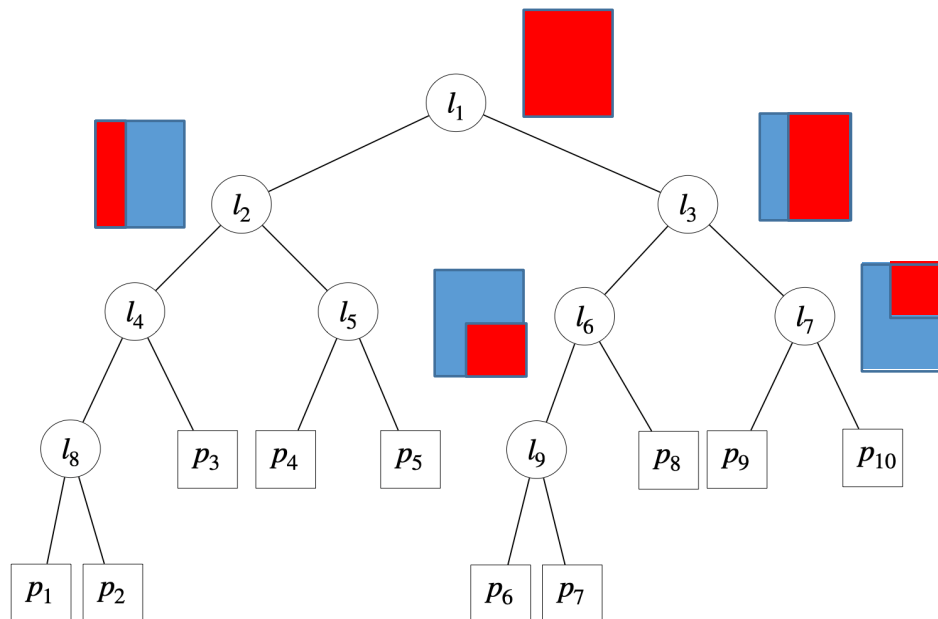


# Exact method: General idea

- Idea
    - Organize data points first
    - Search by conducting a sequence of tests
    - Quickly discard a large portion of data points from the search space
-

# K-d tree

- Given a set of d-dimensional point set  $\mathbf{X}$ .
- A binary tree, i.e. there are 2 child nodes per node.
  - Each node is associated with a subset of the data points

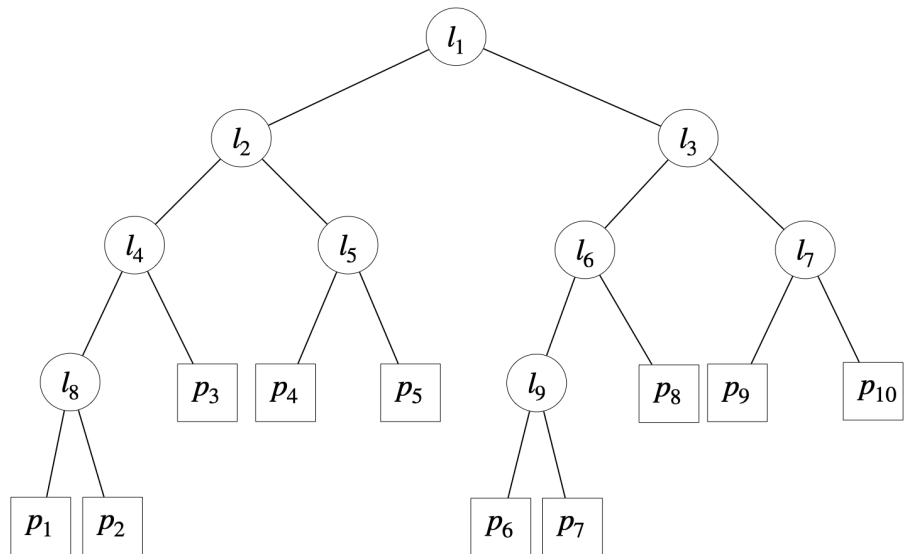
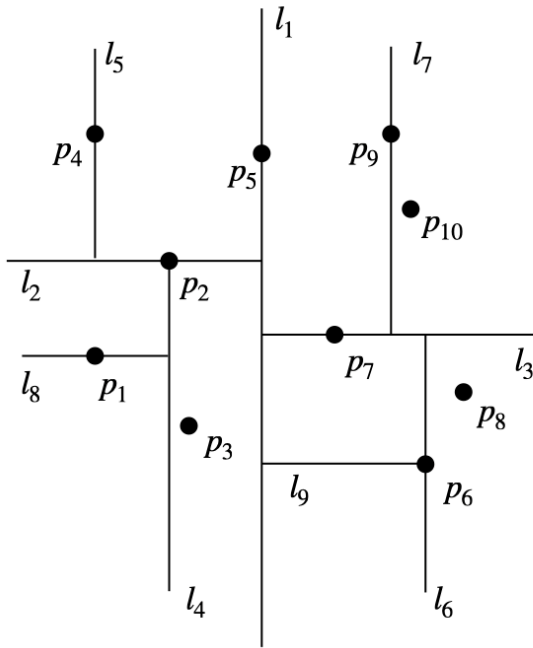


# Building a K-d tree

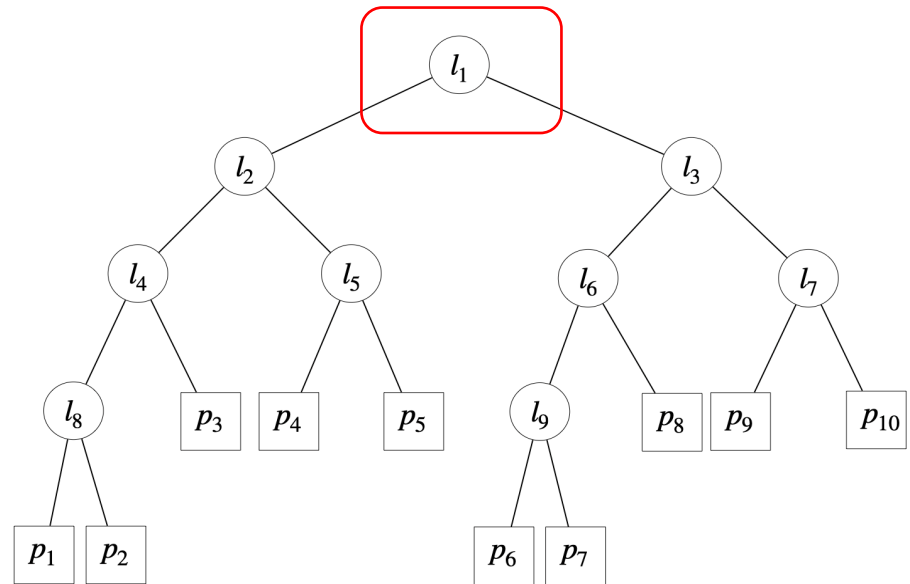
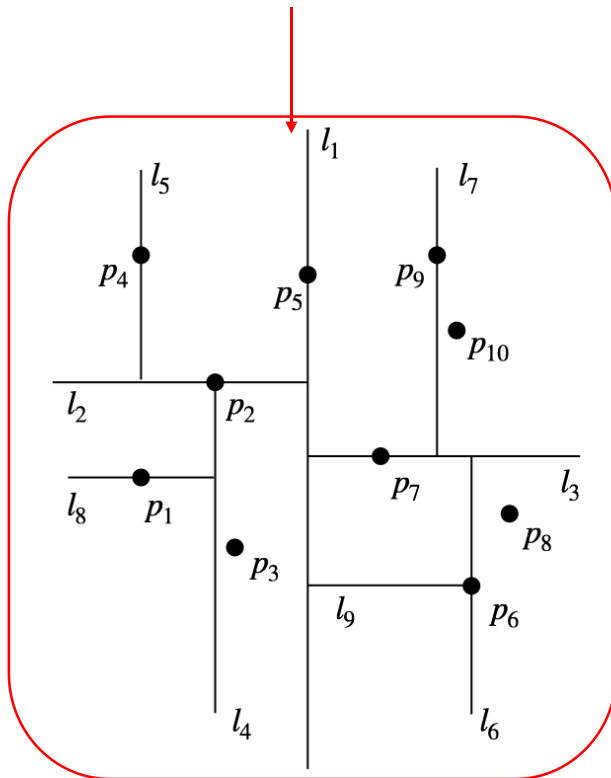
- Start with the root and gradually descend
    - Initialization: assign whole  $X$  to the root
    - Repeat until each node is a singleton node:
      - 1) choose one dimension, find its median value of the points corresponding to the current node
      - 2) split the data points in the current node into the half, assign the split to one of the child: e.g. those with chosen dimension values larger than the median value goes to right, otherwise left
  - The above steps lead to a balanced tree
  - Variants differ in choosing dimensions or in choosing split values
-

# Constructing a Kd-tree

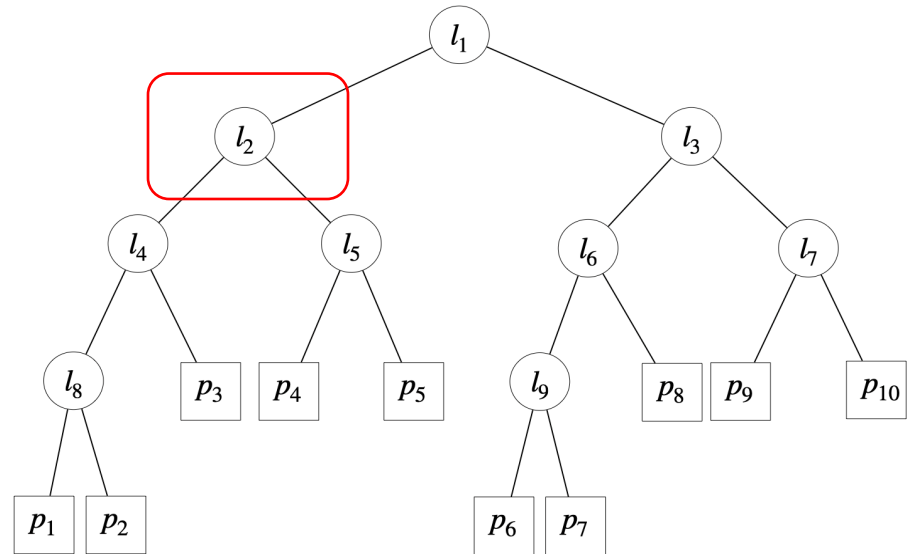
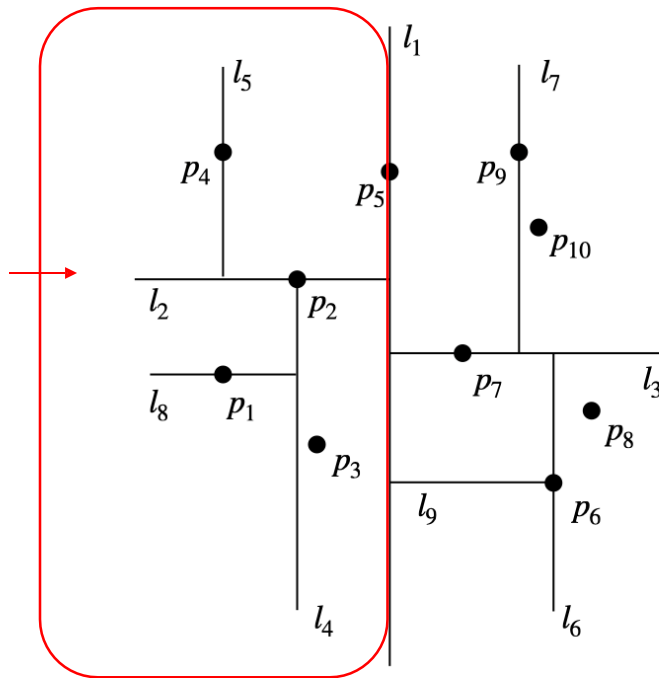
In 2D, a particular split defines a separating **line** (a 1-dimensional geometric entity).



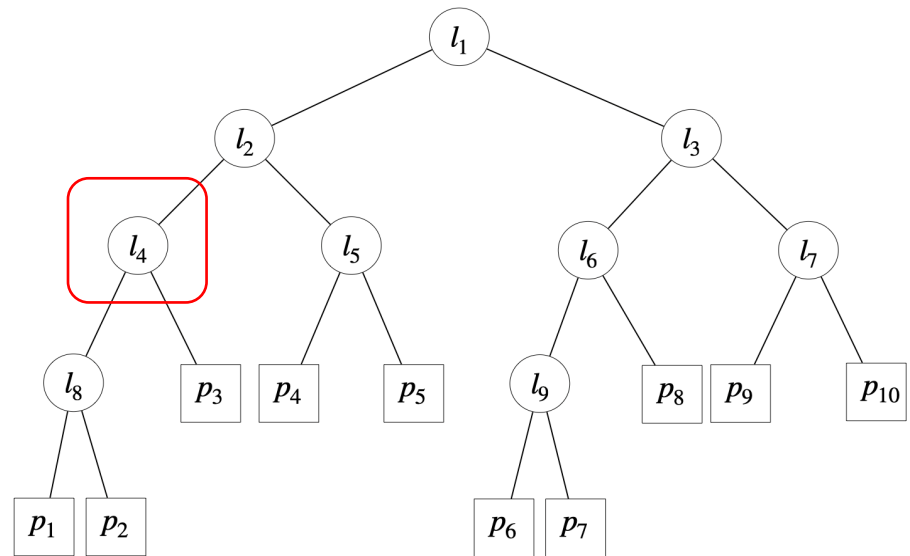
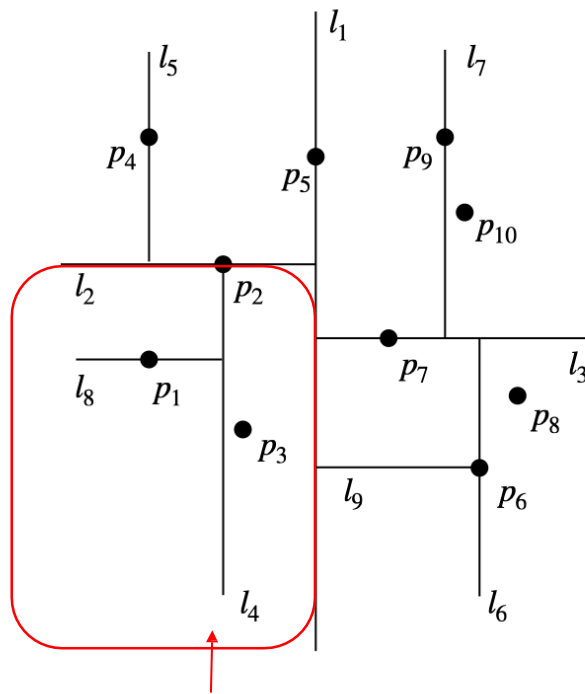
# Constructing a Kd-tree



# Constructing a Kd-tree

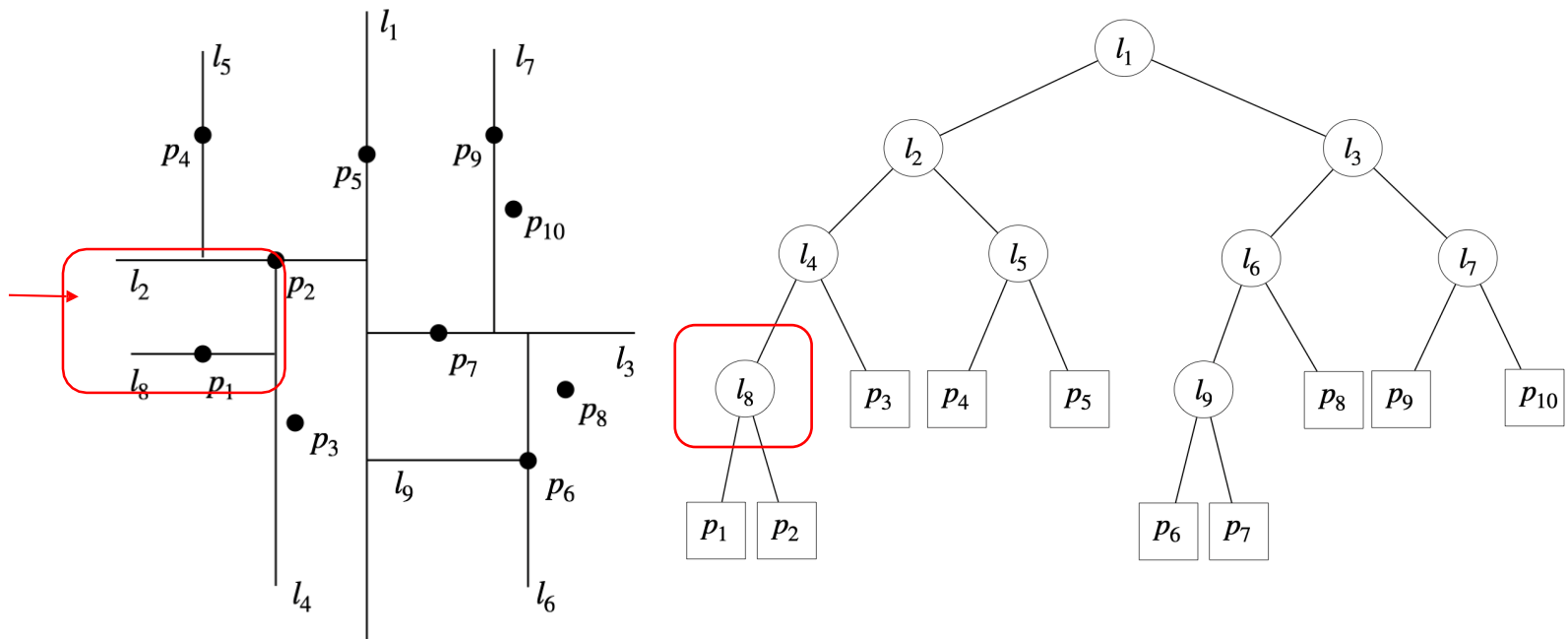


# Constructing a Kd-tree

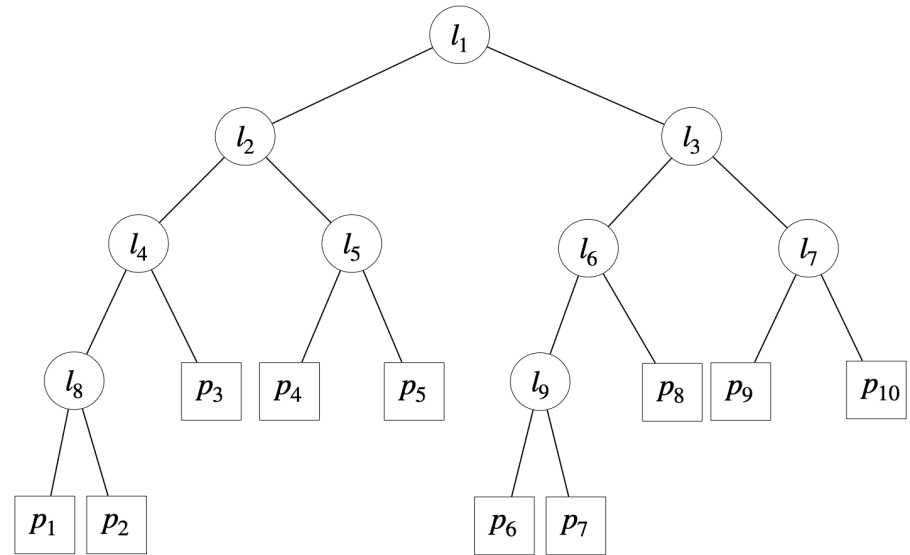
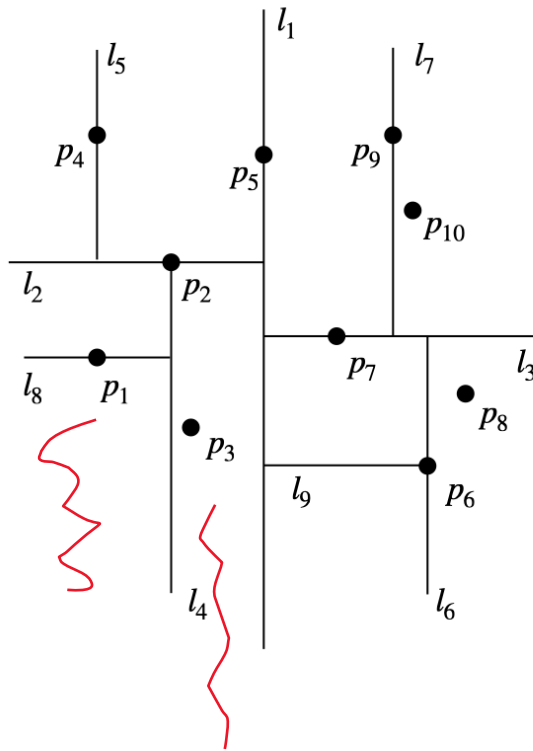




# Constructing a Kd-tree



# Constructing a Kd-tree




# Constructing a Kd-tree

Function BuildKdTree( $P, D$ )

**Require:** A set of points  $P$  of  $M$  dimensions and current depth  $D$ .

```
1: if  $P$  is empty then
2:   return null
3: else
4:    $d \leftarrow D \bmod M$ .
5:    $val \leftarrow$  Median value along dimension  $d$  among points in  $P$ .
6:   Create new node  $node$ .
7:    $node.d \leftarrow d$ .
8:    $node.val \leftarrow val$ .
9:    $node.point \leftarrow$  Point at the median along dimension  $d$ .
10:   $node.left \leftarrow$  BuildKdTree(points in  $P$  for which value at dimension  $d$  is less
    than  $val$ ,  $D + 1$ ).
11:   $node.right \leftarrow$  BuildKdTree(points in  $P$  for which value at dimension  $d$  is
    greater than  $val$ ,  $D + 1$ ).
12:  Return  $node$ .
13: end if
```

Need to sort first!



# Constructing a Kd-tree

Function BuildKdTree( $P, D$ )

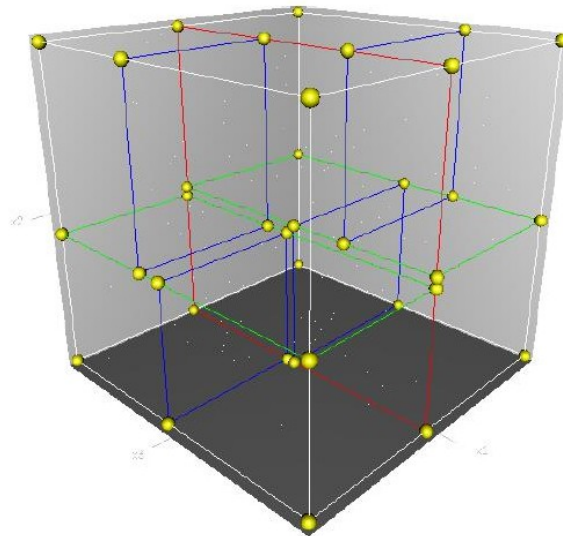
**Require:** A set of points  $P$  of  $M$  dimensions and current depth  $D$ .

```
1: if  $P$  is empty then
2:   return null
3: else
4:    $d \leftarrow D \bmod M$ .
5:    $val \leftarrow$  Median value along dimension  $d$  among points in  $P$ .
6:   Create new node  $node$ .
7:    $node.d \leftarrow d$ .
8:    $node.val \leftarrow val$ .
9:    $node.point \leftarrow$  Point at the median along dimension  $d$ .
10:   $node.left \leftarrow$  BuildKdTree(points in  $P$  for which value at dimension  $d$  is less
    than  $val$ ,  $D + 1$ ).
11:   $node.right \leftarrow$  BuildKdTree(points in  $P$  for which value at dimension  $d$  is
    greater than  $val$ ,  $D + 1$ ).
12:  Return  $node$ .
13: end if
```

Using recursion to implement it

# Example with 3D

In 3D, a particular split defines a separating **plane** (a 2-dimensional geometric entity).

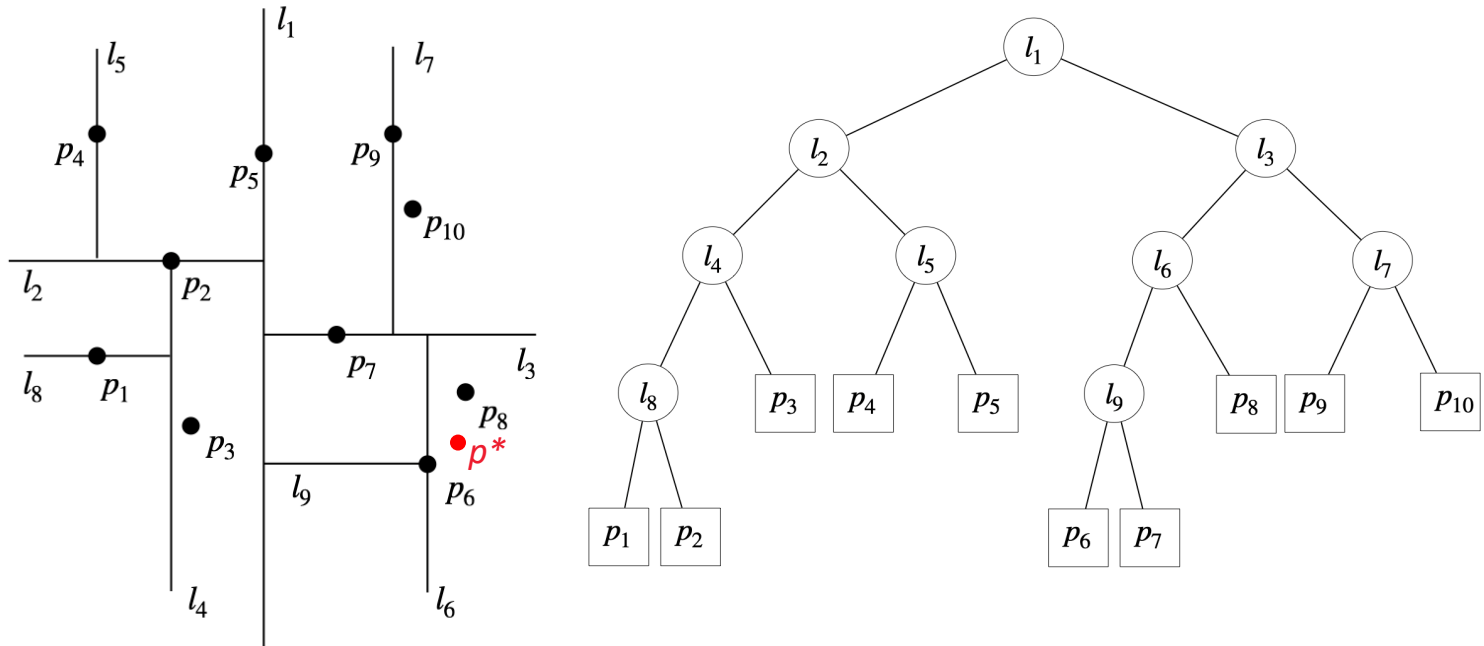


More generally, for  $M$  dimensional data, a particular split defines a separating **hyperplane** of  $(M-1)$  dimensions.

---

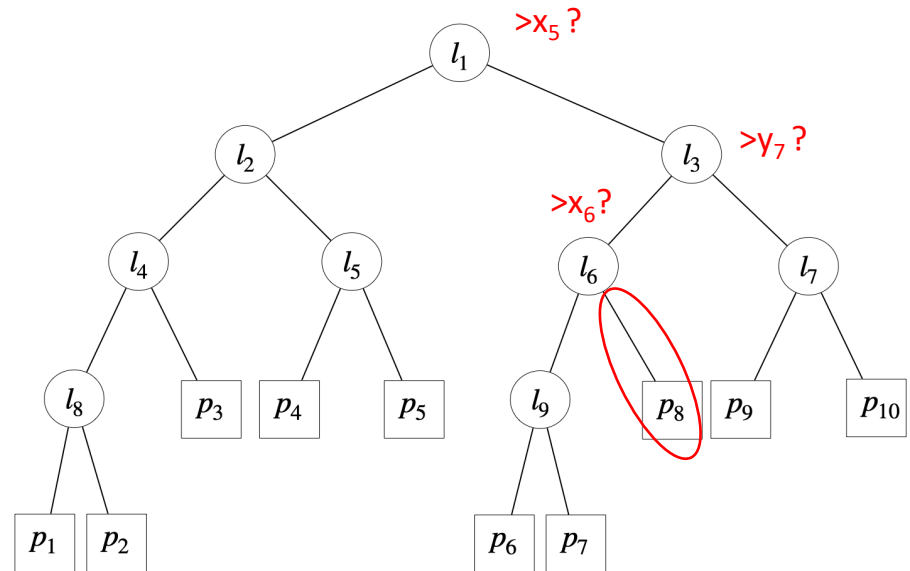
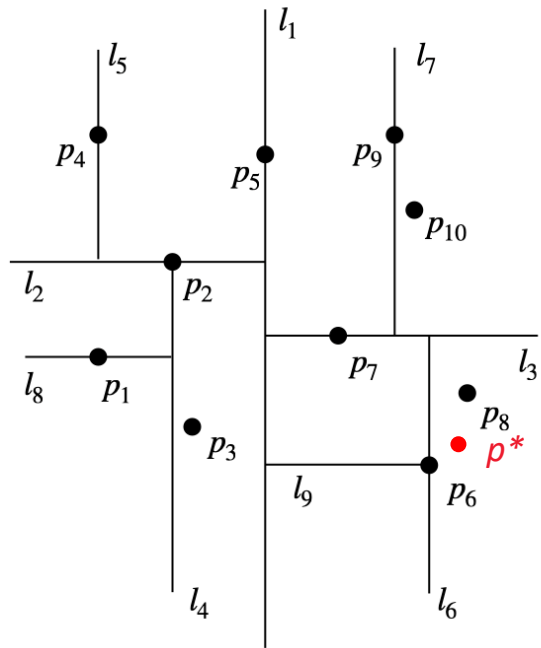
# Nearest Neighbour search with K-d tree

A Kd-tree partitions the  $M$ -dimensional space into a number of **cells**, each corresponding to a particular **branch** of the tree.



# Nearest Neighbour search with K-d tree

Idea 1: Dropping the test point down the tree until a leaf node is reached, then the nearest neighbour is located in the leaf node

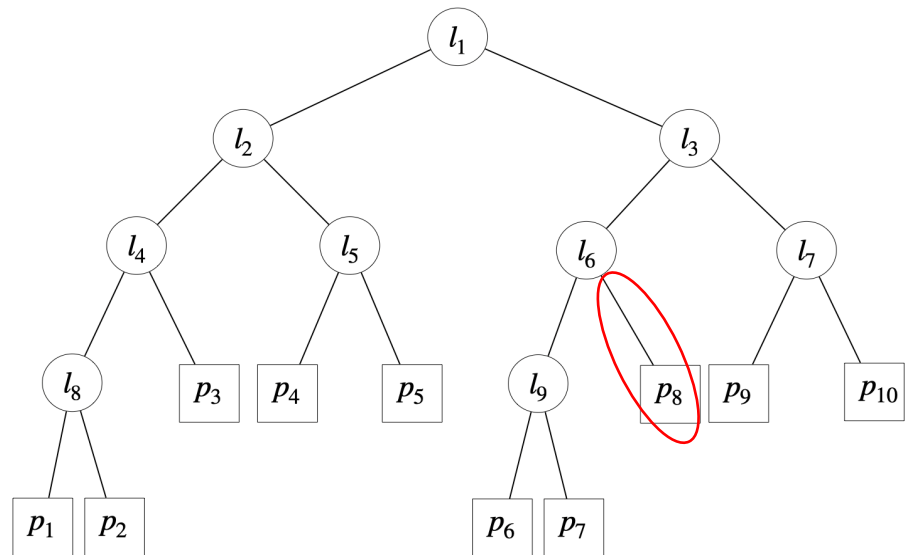
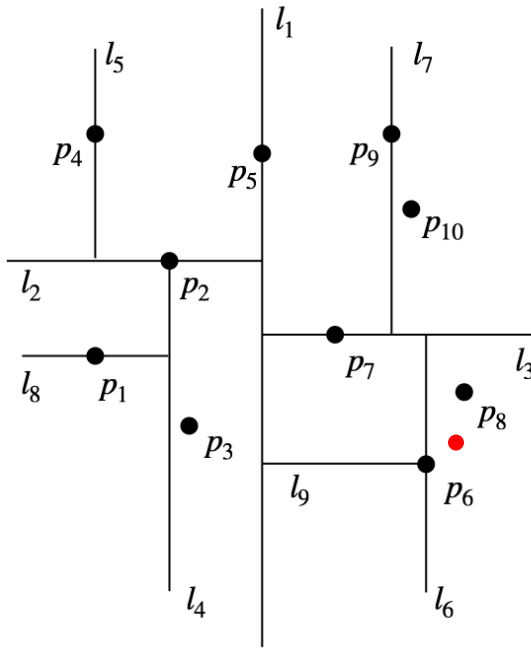




# Nearest Neighbour search with K-d tree

Idea 1: Dropping the test point down the tree until a leaf node is reached, then the nearest neighbour is located in the leaf node

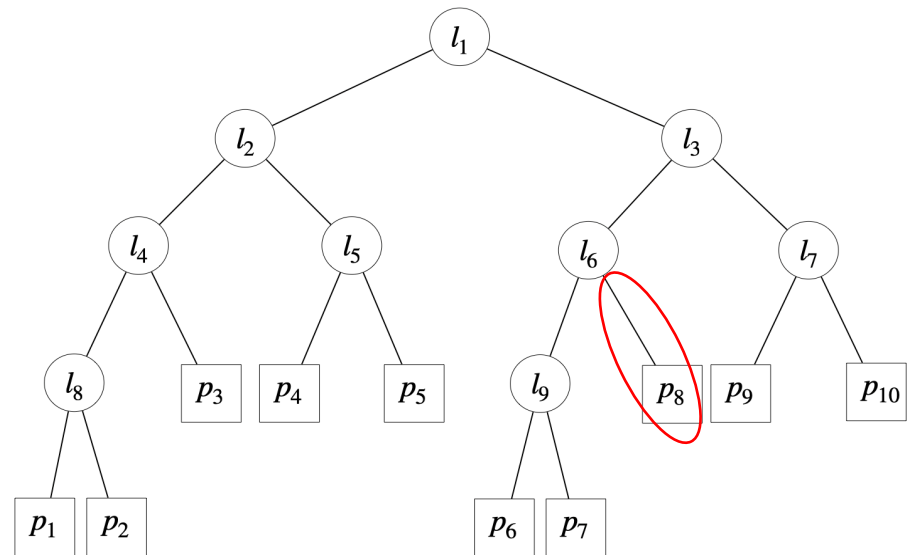
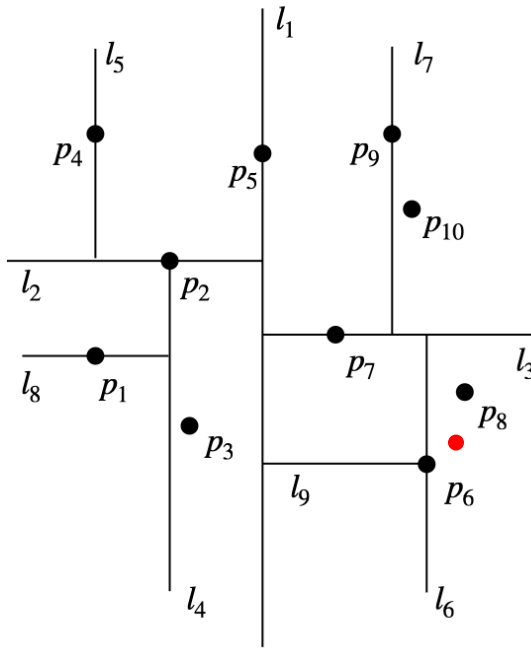
Not true!



# Nearest Neighbour search with K-d tree

Idea: Dropping the test point down the tree until a leaf node is reached, then the nearest neighbour is **more located around the neighbouring cells**

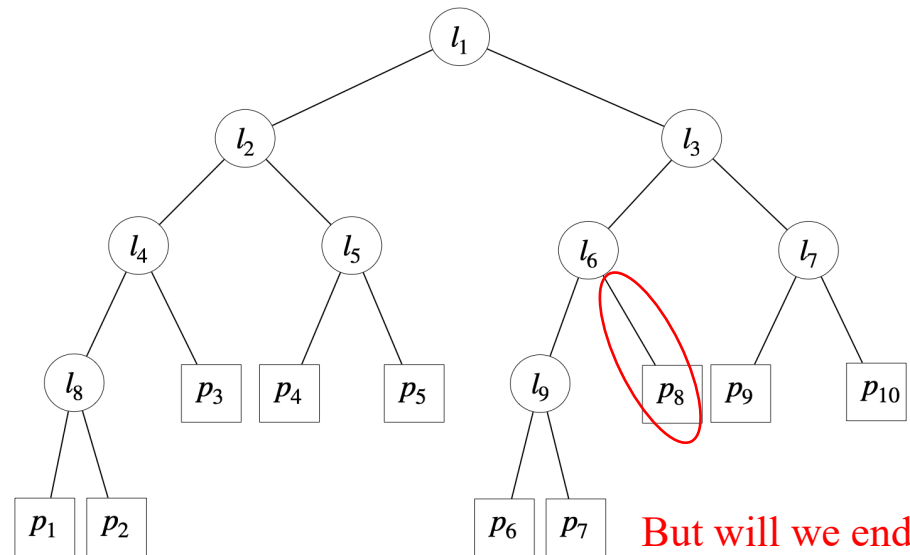
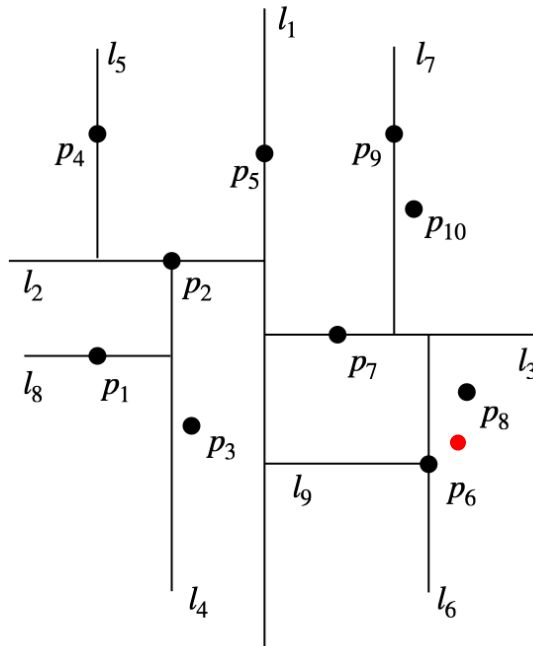
still need to search other branches!



# Nearest Neighbour search with K-d tree

Idea: Dropping the test point down the tree until a leaf node is reached, then the nearest neighbour is **more located around the neighbouring cells**.

still need to search other branches!



But will we end up with doing exhaustive search?

Solution: we will unwind the recursion of the tree and check if the nearest neighbour is located on the other branch of a node

# Nearest Neighbour search with K-d tree

$\text{distance}(\text{node}, \mathbf{y})$

Computes the Euclidean distance of  $\mathbf{y}$  to the point contained in node  $\text{node}$ , i.e.,

$$\text{distance}(\text{node}, \mathbf{y}) = \sqrt{\sum_{j=1}^M (\text{node.point}[j] - \mathbf{y}[j])^2}$$

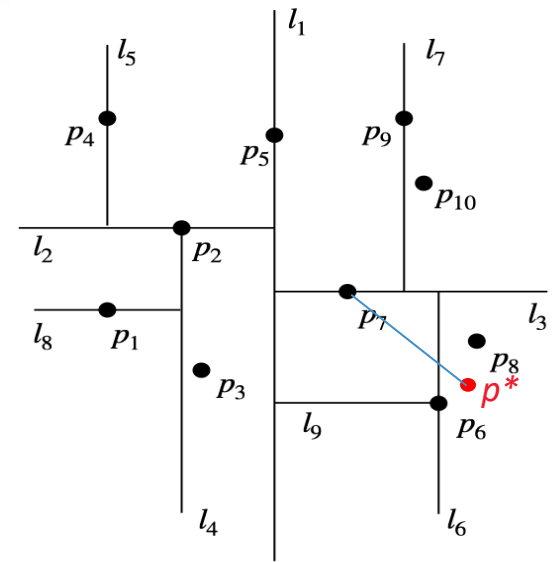
$\text{val} \leftarrow$  Median value along dimension  $d$  among points in  
Create new node  $\text{node}$ .

$\text{node.d} \leftarrow d$ .

$\text{node.val} \leftarrow \text{val}$ .

$\text{node.point} \leftarrow$  Point at the median along dimension  $d$ .

$\text{node.left} \leftarrow \text{BuildKdTree}(\text{points in } P \text{ for which value a}$



# Nearest Neighbour search with K-d tree

$\text{distance}(\text{node}, \mathbf{y})$

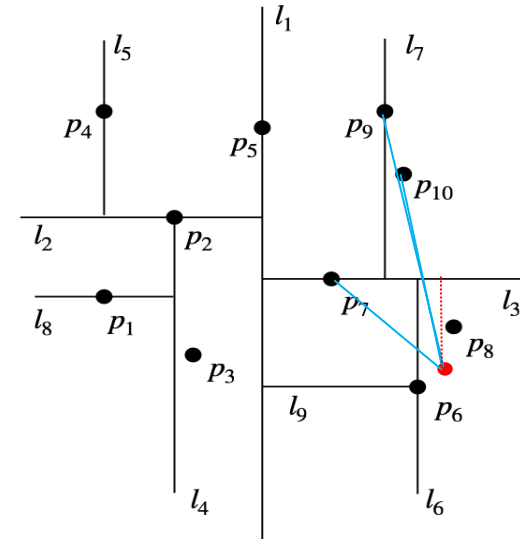
Computes the Euclidean distance of  $\mathbf{y}$  to the point contained in node  $\text{node}$ , i.e.,

$$\text{distance}(\text{node}, \mathbf{y}) = \sqrt{\sum_{j=1}^M (\text{node.point}[j] - \mathbf{y}[j])^2}$$

$\text{distance\_lb}(\text{node}, \mathbf{y})$

Computes the **lower bound** of the Euclidean distances of  $\mathbf{y}$  to points on the **other branch**. The lower bound is obtained as the distance along the dimension  $\text{node.d}$  to the point contained in  $\text{node}$ , i.e.,

$$\text{distance\_lb}(\text{node}, \mathbf{y}) = \text{abs}(\text{node.point}[\text{node.d}] - \mathbf{y}[\text{node.d}])$$



Any point within the other side of the node definitely has larger distance than the lower bound

# Nearest Neighbour search with K-d tree

- Record current best (smallest) distance
  - Check the if the distance lower bound of one branch is greater than the current best distance
    - If yes, it means that it is impossible to find a closer point in that branch, then we can safely discard the branch
    - If no, descend the node to continue searching
-

# Nearest Neighbour search with K-d tree

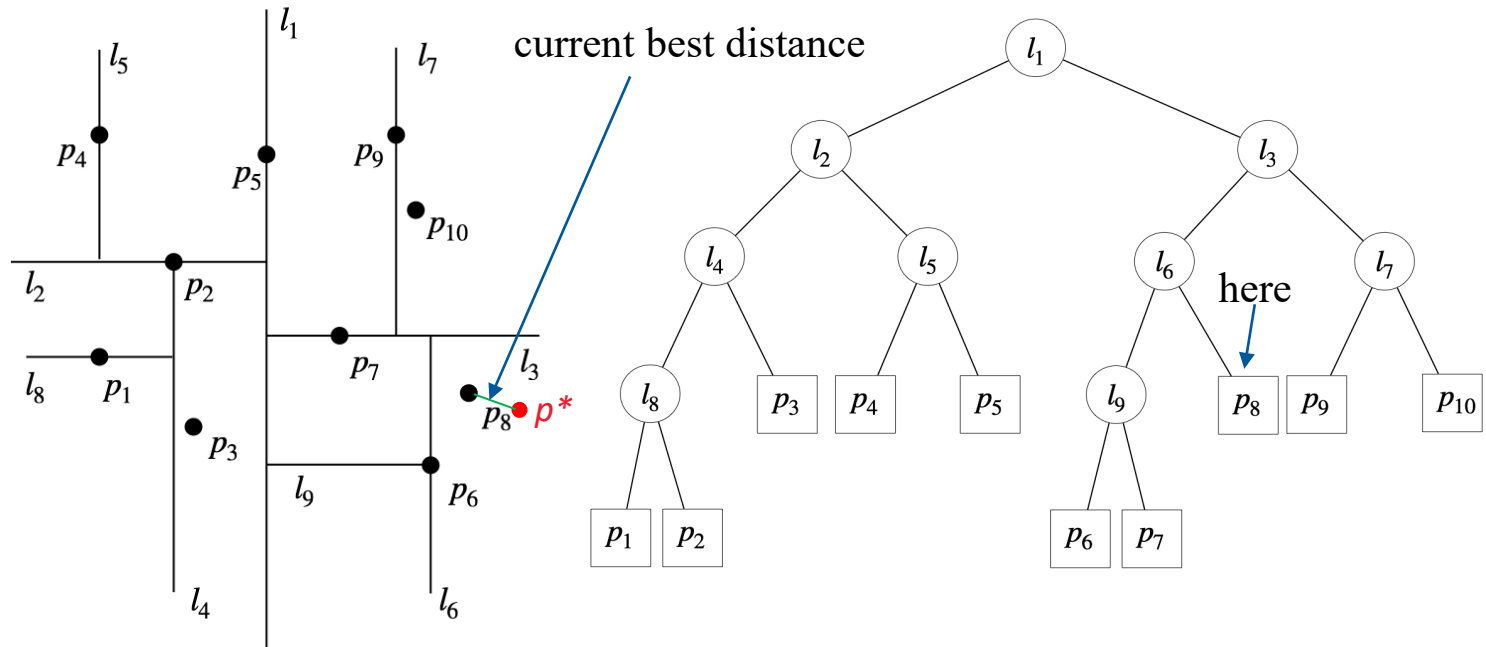
Given a query point  $y$ , a nearest neighbour search with a Kd-tree is accomplished by:

1. Dropping  $y$  down the tree until a leaf node is reached.
  2. Record the leaf node as the current best node.
  3. Unwind the recursion of the tree, calculating at each node the lower bound of the distance to  $y$  of the other branch.
  4. If the lower bound is ~~not~~ larger than the current best distance, continue ascending the tree.
  5. If the lower bound is not larger than the current best distance, descend the other branch until a leaf node is reached. Then update the best node if necessary.
  6. Once the root node is reached, terminate the search and return the point in the best node as the nearest neighbour of  $y$ .
-



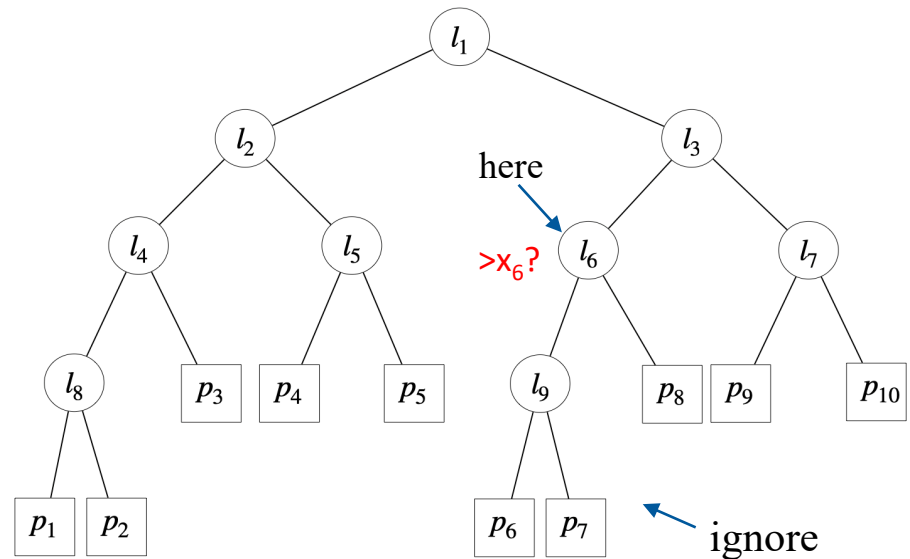
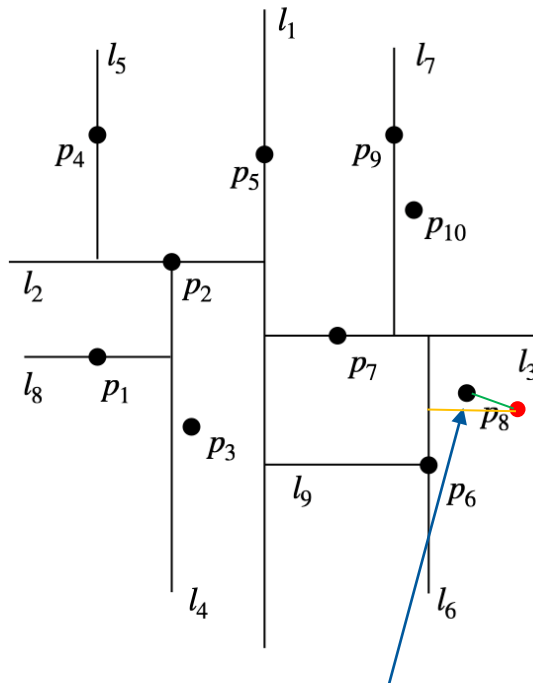
## Example 1

The query point is dropped down the tree until it reaches a leaf node.



Tentative nearest neighbor:  $p_8$

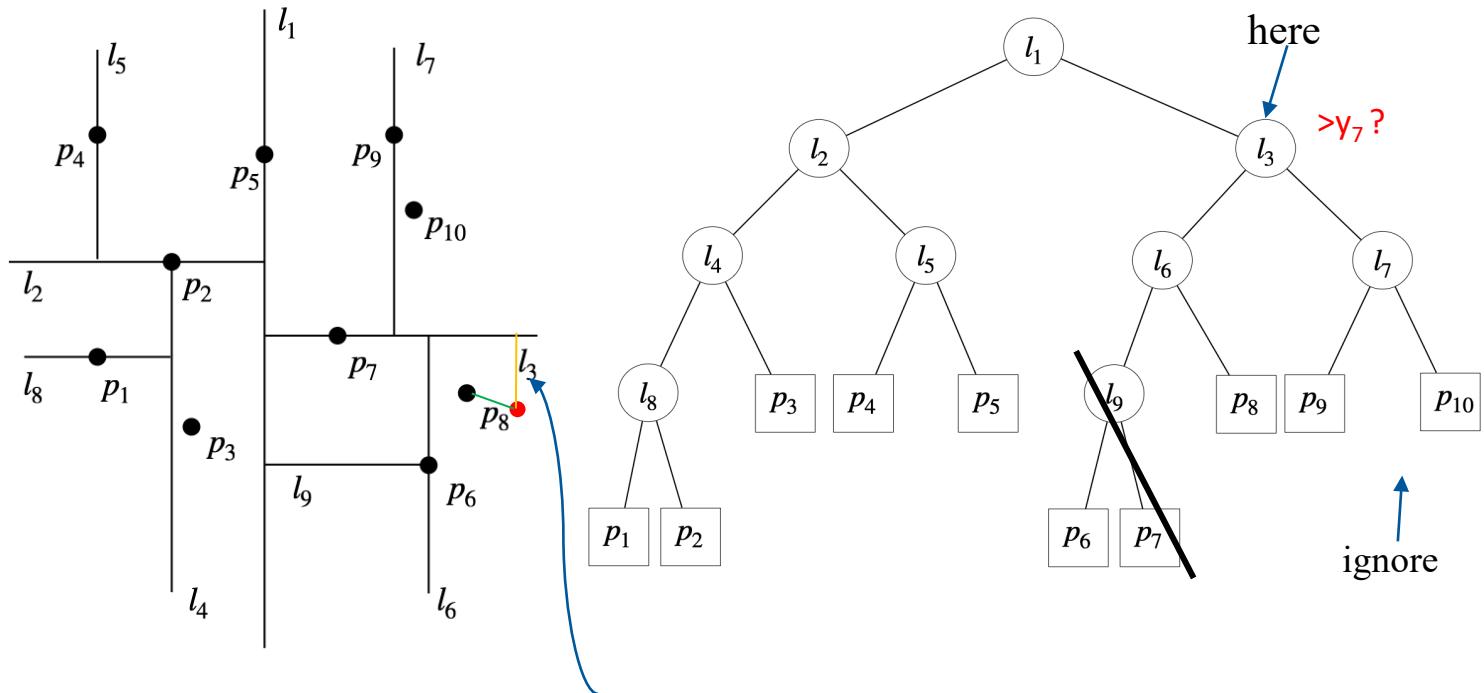
# Example 1



lower bound is greater than current best, ignore the other branch

Tentative nearest neighbor:  $p_8$

# Example 1

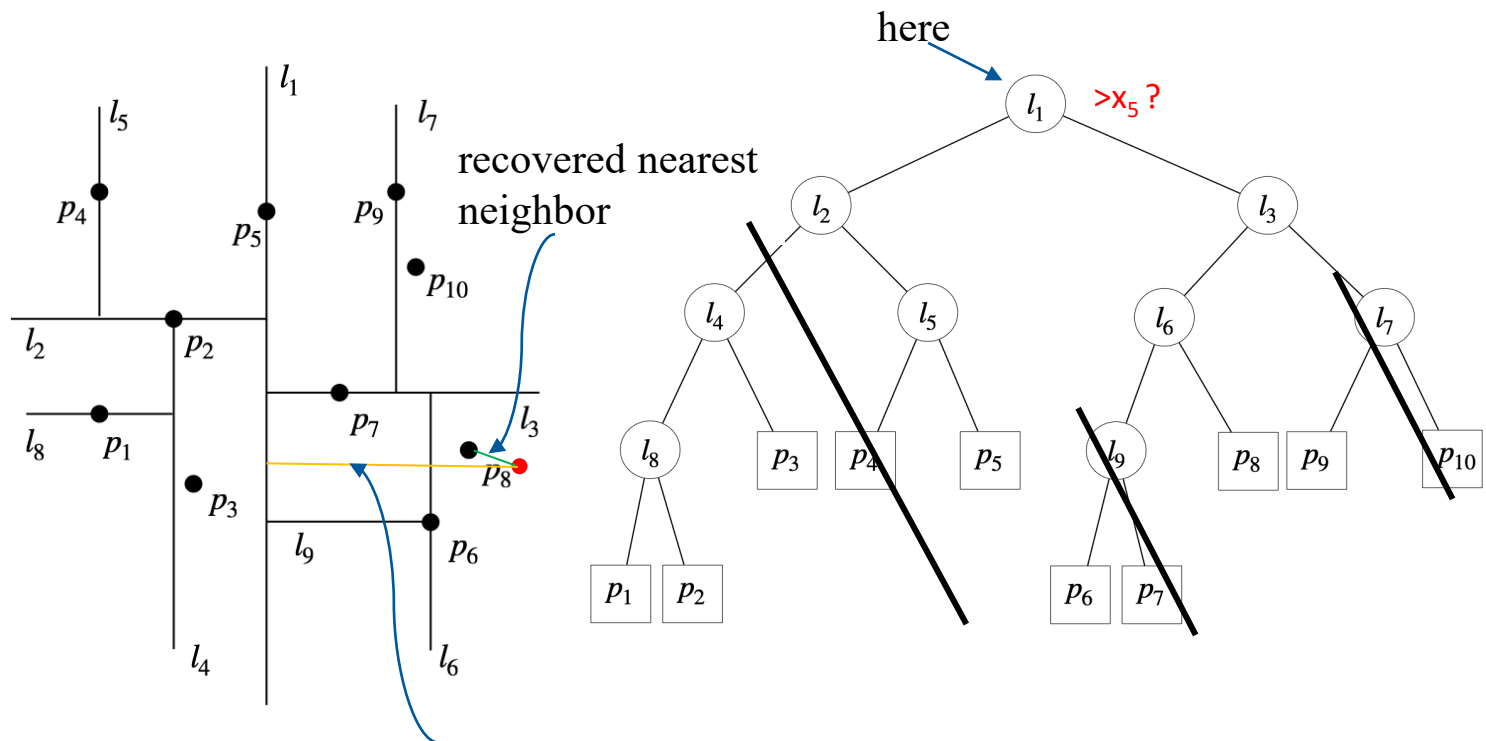


lower bound is greater than current best, ignore the other branch

Tentative nearest neighbor:  $p_8$

# Example 1

Search terminates. Number of **distinct** full Euclidean distances computed is 4.



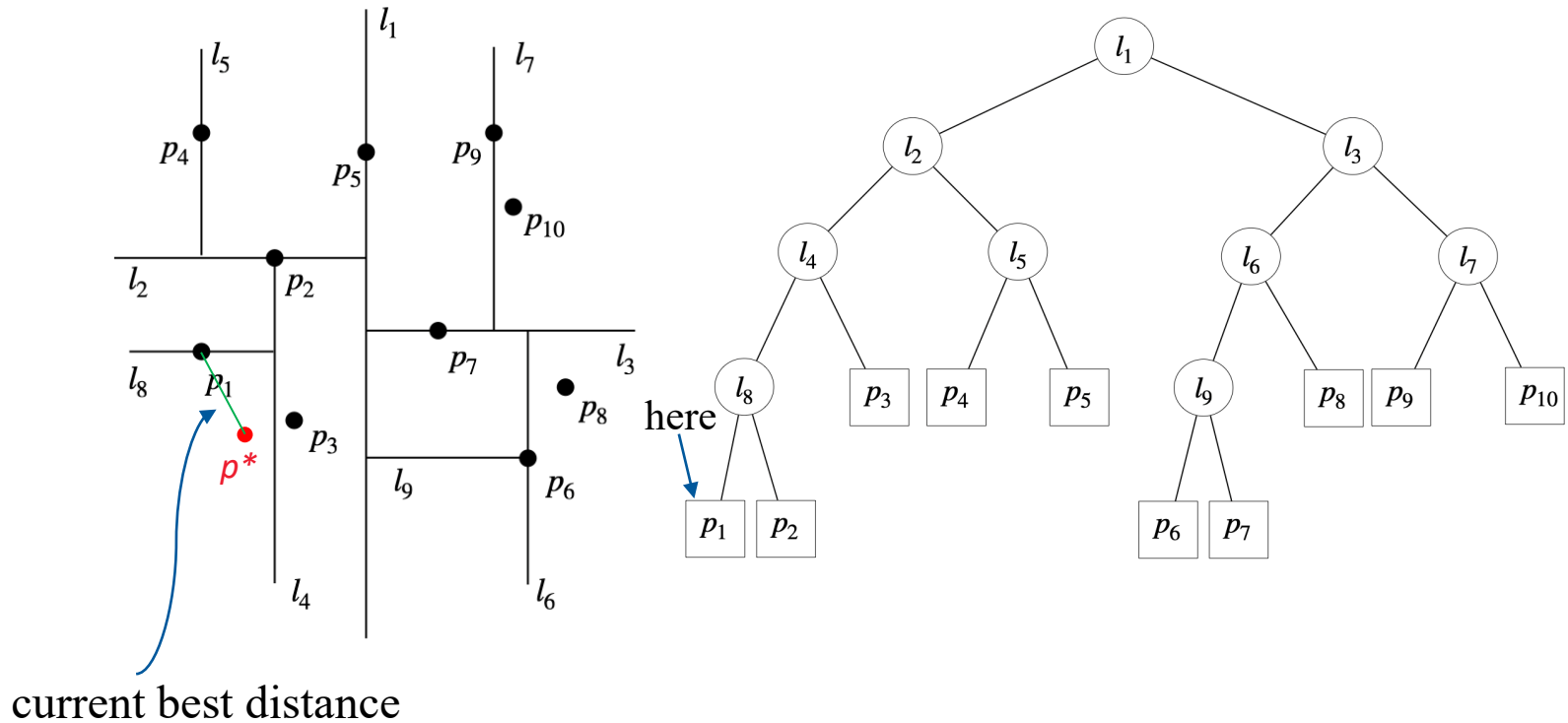
lower bound is greater than current best distance

~~Tentative nearest neighbor:  $p_8$~~

Nearest neighbor:  $p_8$

## Example 2

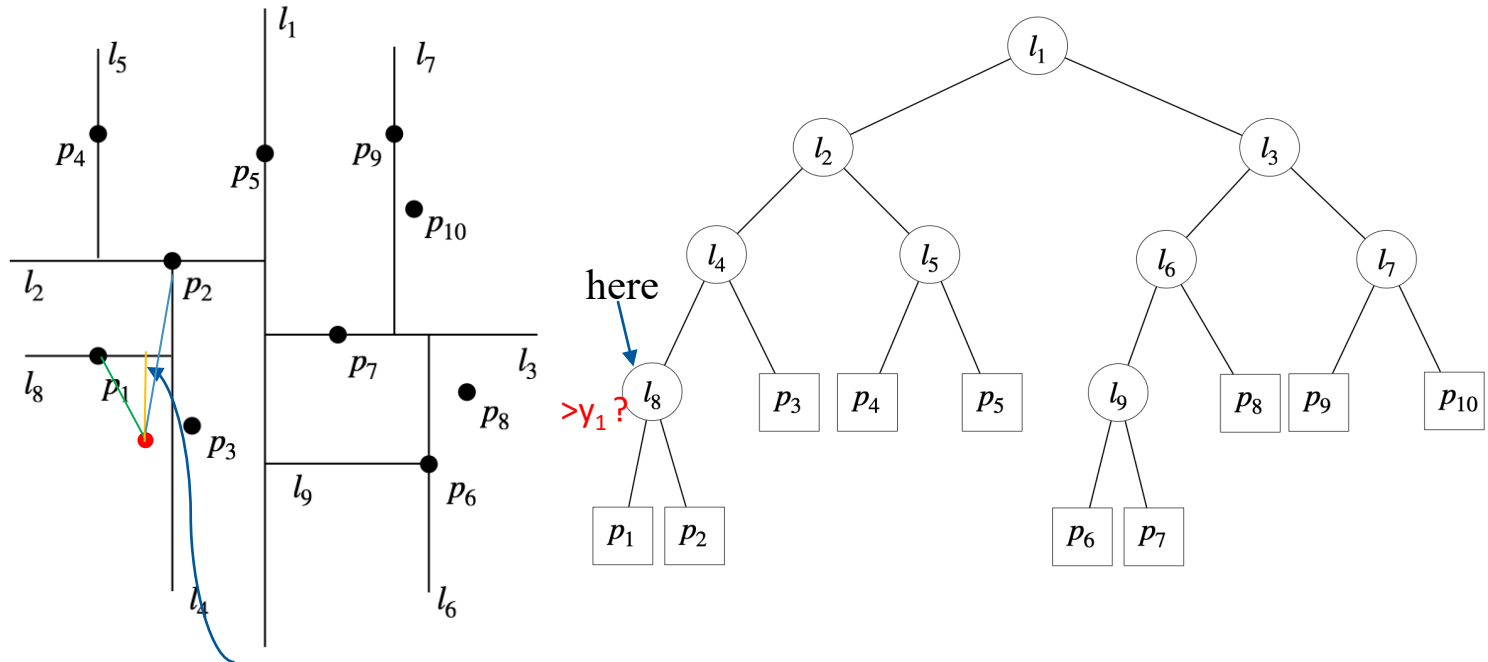
The query point is dropped down the tree until it reaches a leaf node.



Tentative nearest neighbor:  $p_1$

## Example 2

If a lower bound does not exceed the current best distance, we have to descend to the other branch.

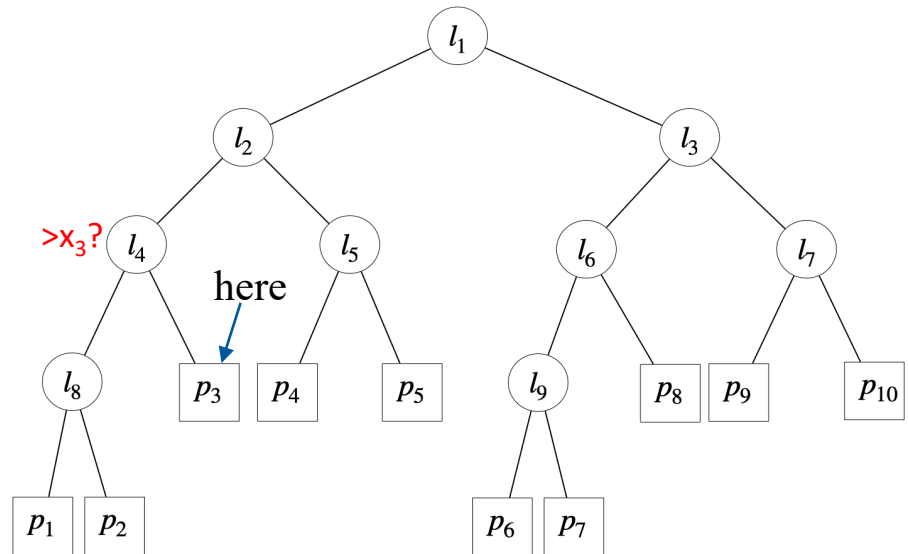
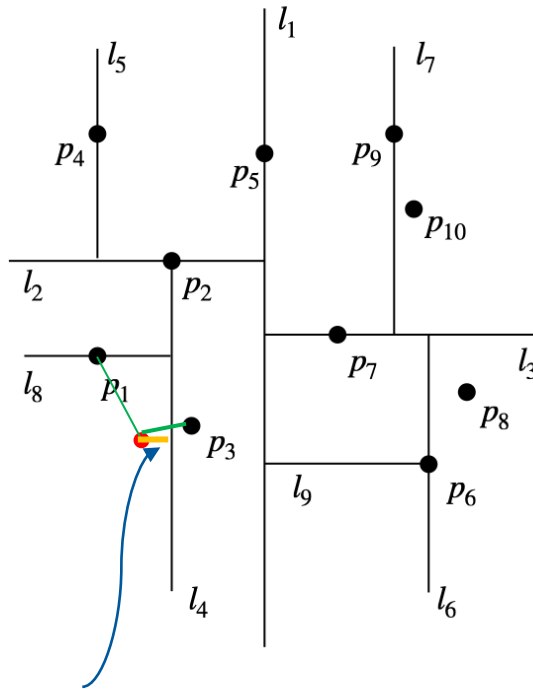


lower bound is not greater than current best

Tentative nearest neighbor:  $p_1$

## Example 2

If a better solution is found, update the current best solution.

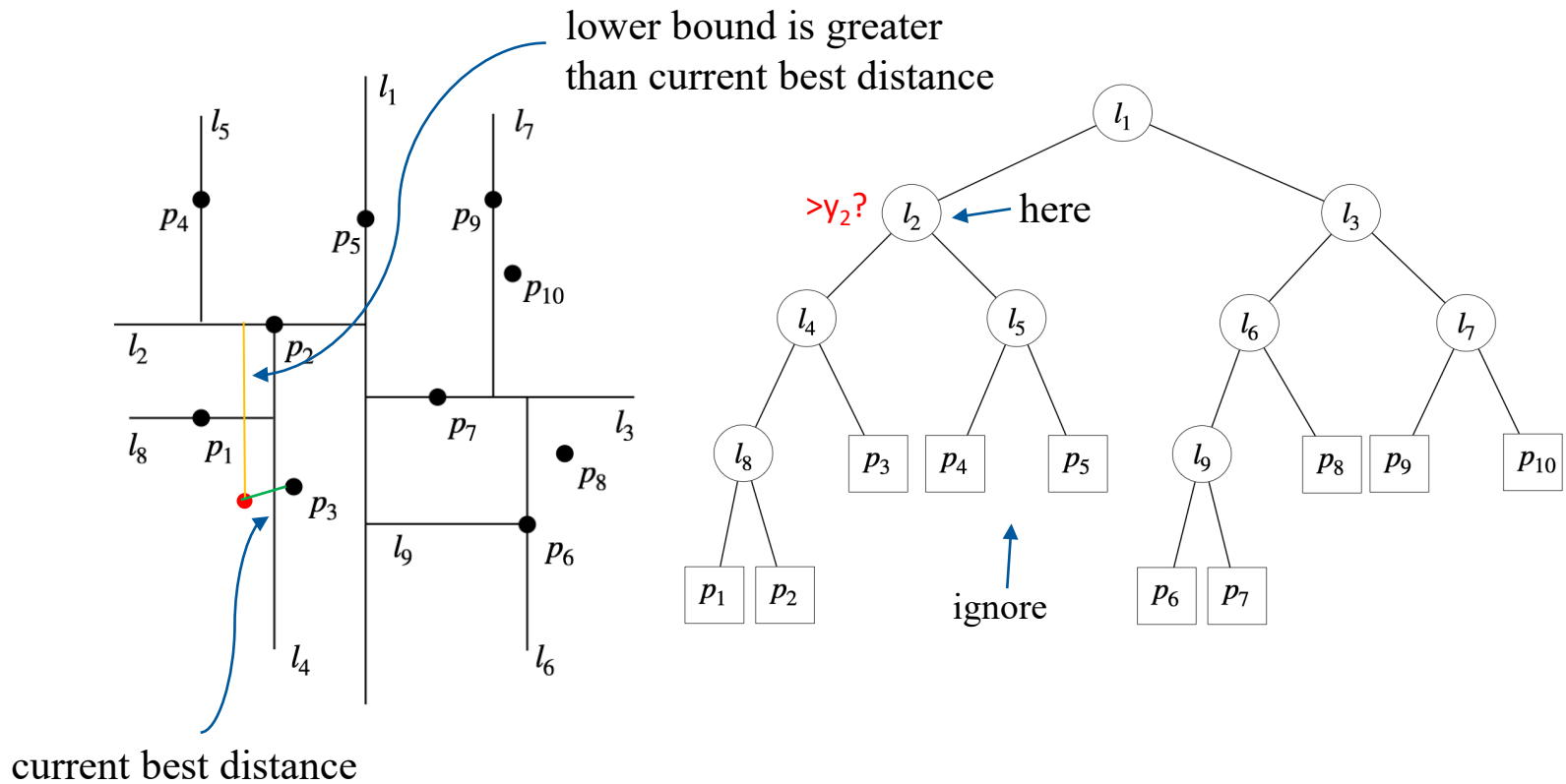


distance is smaller than current best distance, update

Tentative nearest neighbor:  ~~$p_7$~~   $p_3$



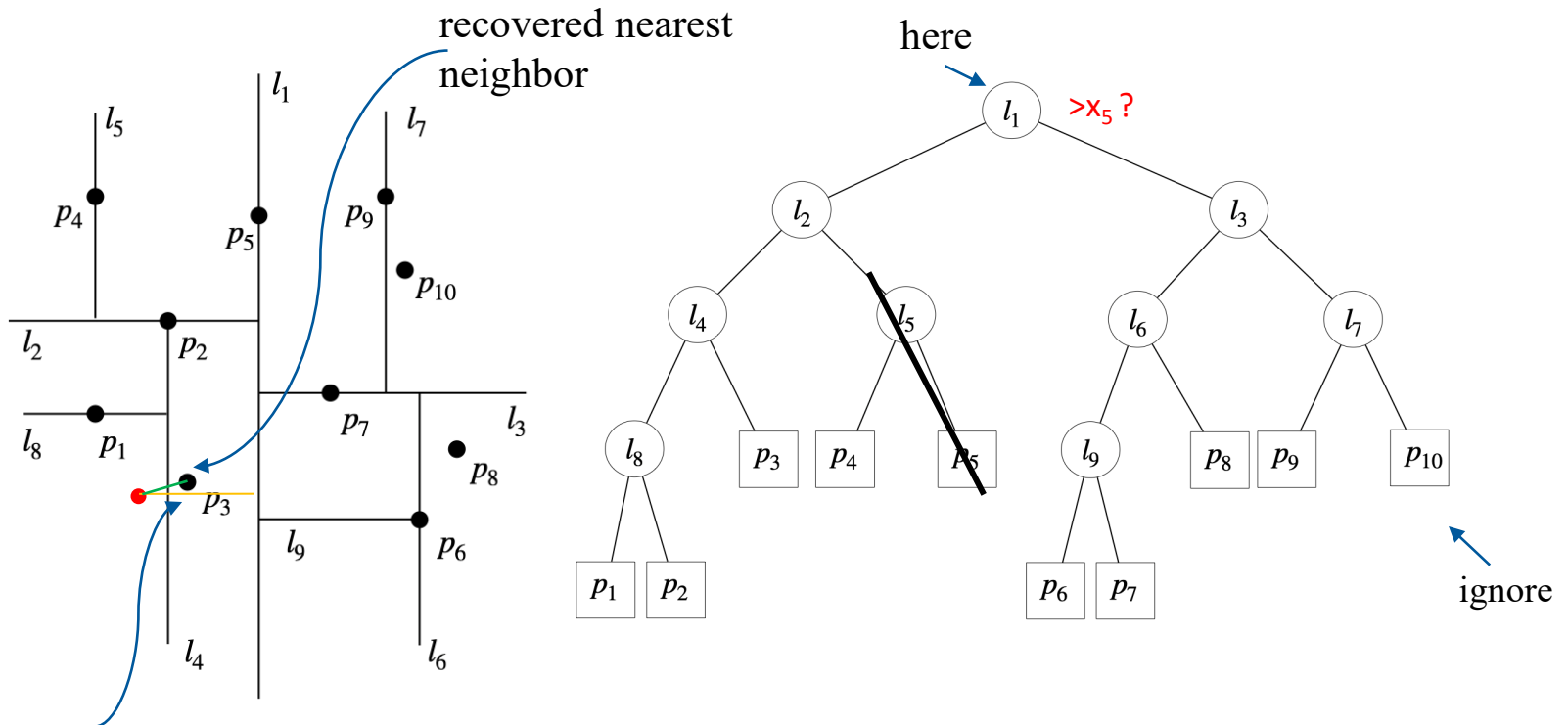
## Example 2



Tentative nearest neighbor:  $p_3$

## Example 2

Search terminates. Number of **distinct** full Euclidean distances computed is 7.



lower bound is greater than current best distance

~~Tentative nearest neighbor:  $p_3$~~

**Nearest neighbor:  $p_3$**

# Theoretic behaviour

Building a balanced  $Kd$ -tree for  $N$  points takes  $O(N \log^2 N)$  if a  $O(N \log N)$  sort is used to compute the media at each level.

This can be improved if a more efficient sort procedure is used. However, we can generally not concern with this since building the tree can be done offline.

We are more concerned with the search performance – This depends on how many times we need to descend into the separate branches. Unfortunately, this is hard to asymptotically analysed since the speed depends on **how the data is distributed**.

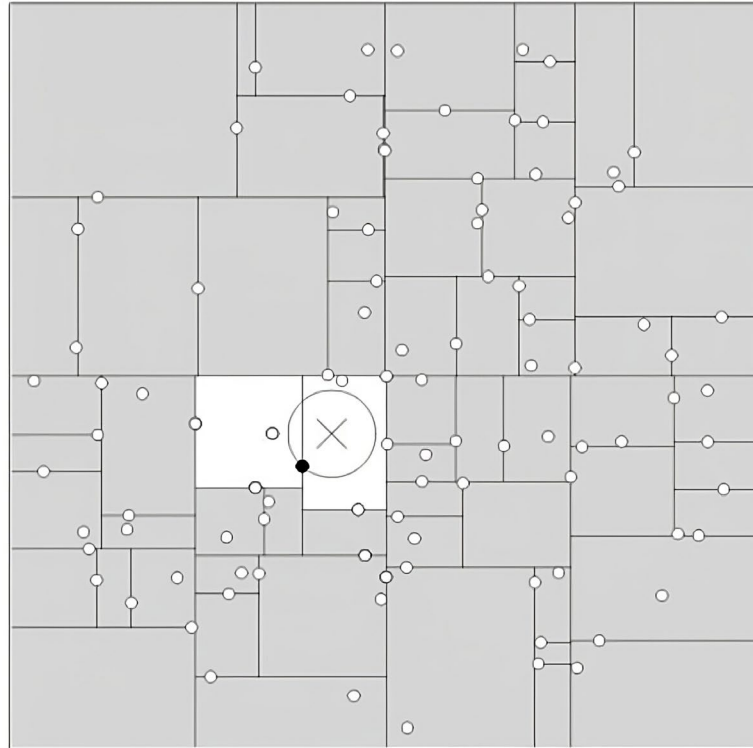
It is clear that at least  $O(N \log N)$  computations of distances are necessary since we need to drop to at least one leaf node.

It is also clear that no more than  $N$  nodes are search. In the worst case the algorithm reduces to a naive exhaustive search.

---

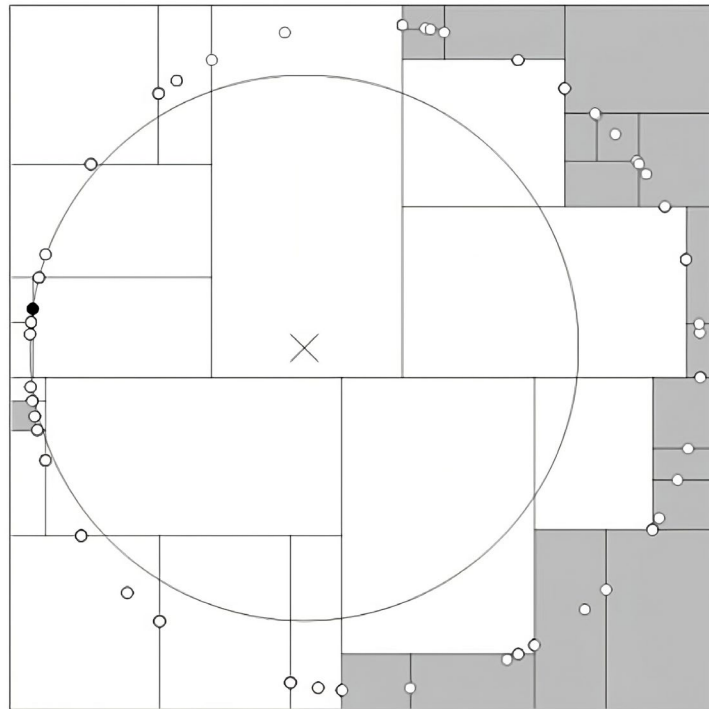
# The distribution affects the search performance (cont.)

A case where most of the branches are ignored (the grey boxes).



# The distribution affects the search performance (cont.)

A case which forces almost all branches to be inspected.



# Performance on high-dimensional data

The *Kd*-tree method for nearest neighbour search has a serious weakness – it is not efficient for high-dimensional data.

As a general rule *Kd*-tree can provide the highest gains in efficiency if the number of points  $N$  is much greater than  $2^M$ , where  $M$  is the dimension of the data.

If the rule is not satisfied in most cases the *Kd*-tree search reduces to an exhaustive search.

---

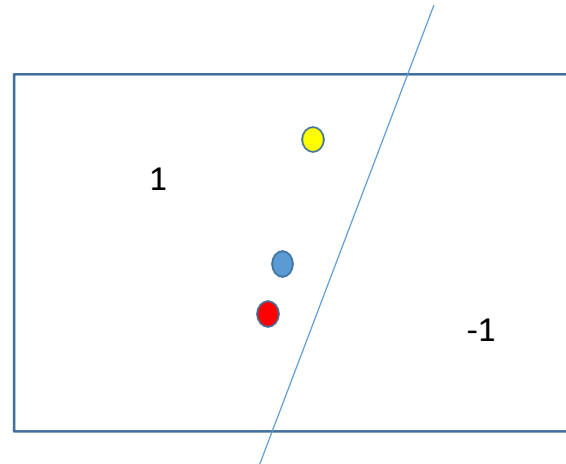
# Efficient Nearest Neighbour Search [optional]

- Approximate Nearest Neighbour Search
    - Idea: generate key for each data point  $x_i$  -->  $s_i$
    - Searching on keys is much simpler and efficient e.g. keys are binary vectors
    - If two points share the same key, they are more likely to be similar (to be the nearest neighbour)
-

# Efficient Nearest Neighbour Search [optional]

- Example: Locality Sensitive Hashing

$$S_1 = \text{sign}(\mathbf{w}_1^T \mathbf{x}_i)$$



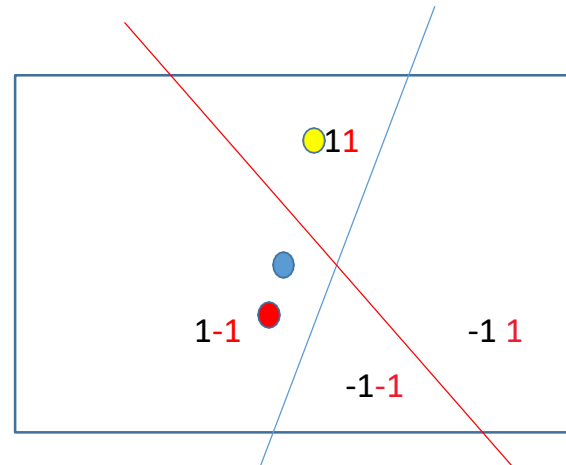


# Efficient Nearest Neighbour Search [optional]

- Example: Locality Sensitive Hashing

$$S_1 = \text{sign}(\mathbf{w}_1^T \mathbf{x}_i)$$

$$S_2 = \text{sign}(\mathbf{w}_2^T \mathbf{x}_i)$$



# Efficient Nearest Neighbour Search [optional]

- Example: Locality Sensitive Hashing (LSH)

$$S_1 = \text{sign}(\mathbf{w}_1^T \mathbf{x}_i)$$

$$S_2 = \text{sign}(\mathbf{w}_2^T \mathbf{x}_i)$$

...

$$S_k = \text{sign}(\mathbf{w}_k^T \mathbf{x}_i)$$

- More projections, finer partition
- Closer points are more likely to share the same keys

