Efficient Nearest Neighbour Search

3007/7059 Artificial Intelligence

Lingqiao Liu

School of Computer Science
The University of Adelaide

Nearest Neighbour Search

Given a set of points $\mathcal{X} = \{\mathbf{x}_i\}_{i=1}^{N}$ and a query point $\mathbf{y}$, both in M-dimensional space, find the nearest neighbour of $\mathbf{y}$ among $\mathcal{X}$.

The concept of "nearness" is based on a pre-defined distance metric, usually the Euclidean distance.

A naive search algorithm scans each and every point in $\mathcal{X}$, incurring a computational cost of $\mathcal{O}(MN)$. 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

2? : <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

2? : <5
2? : <3

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

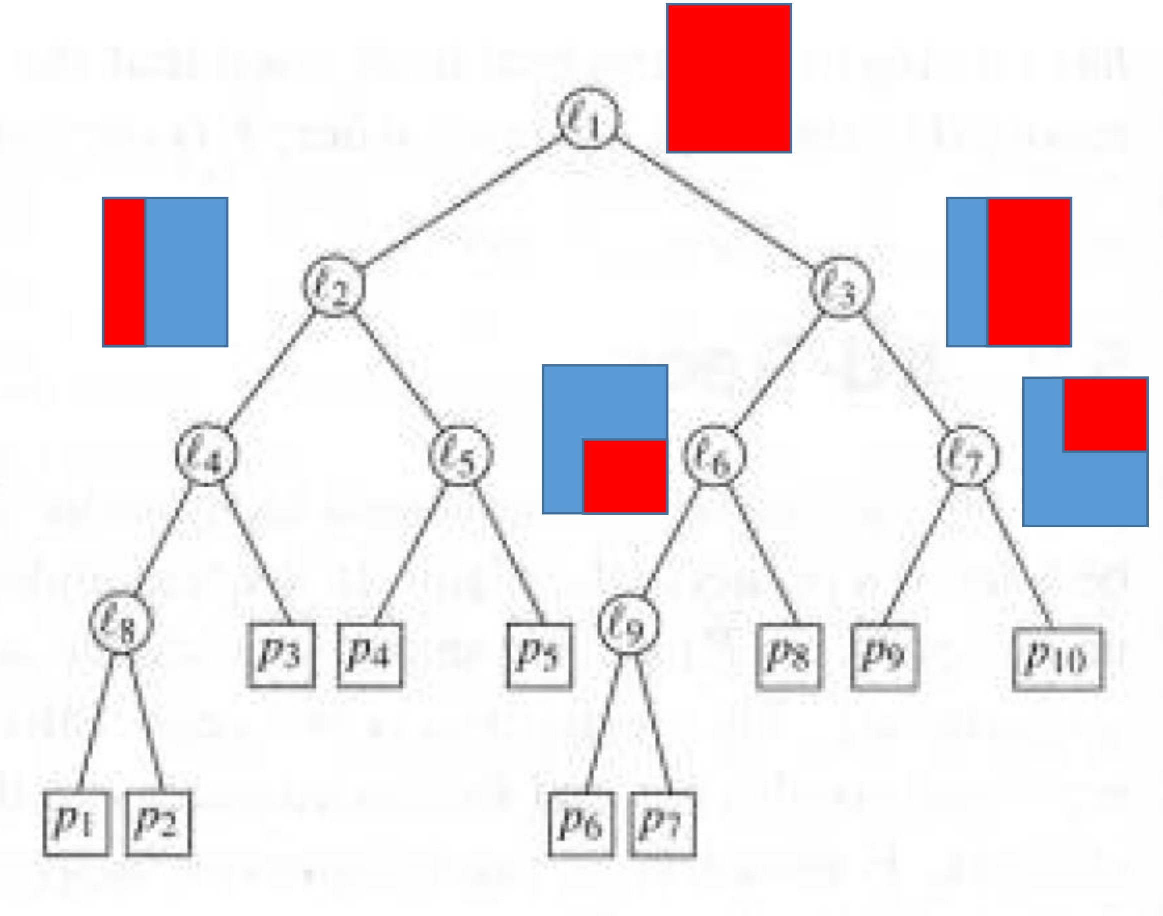
2? : <5
2? : <3

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 X
- A binary tree, i.e. there are 2 child nodes per node.
  - Each node is associated with a subset of the data points

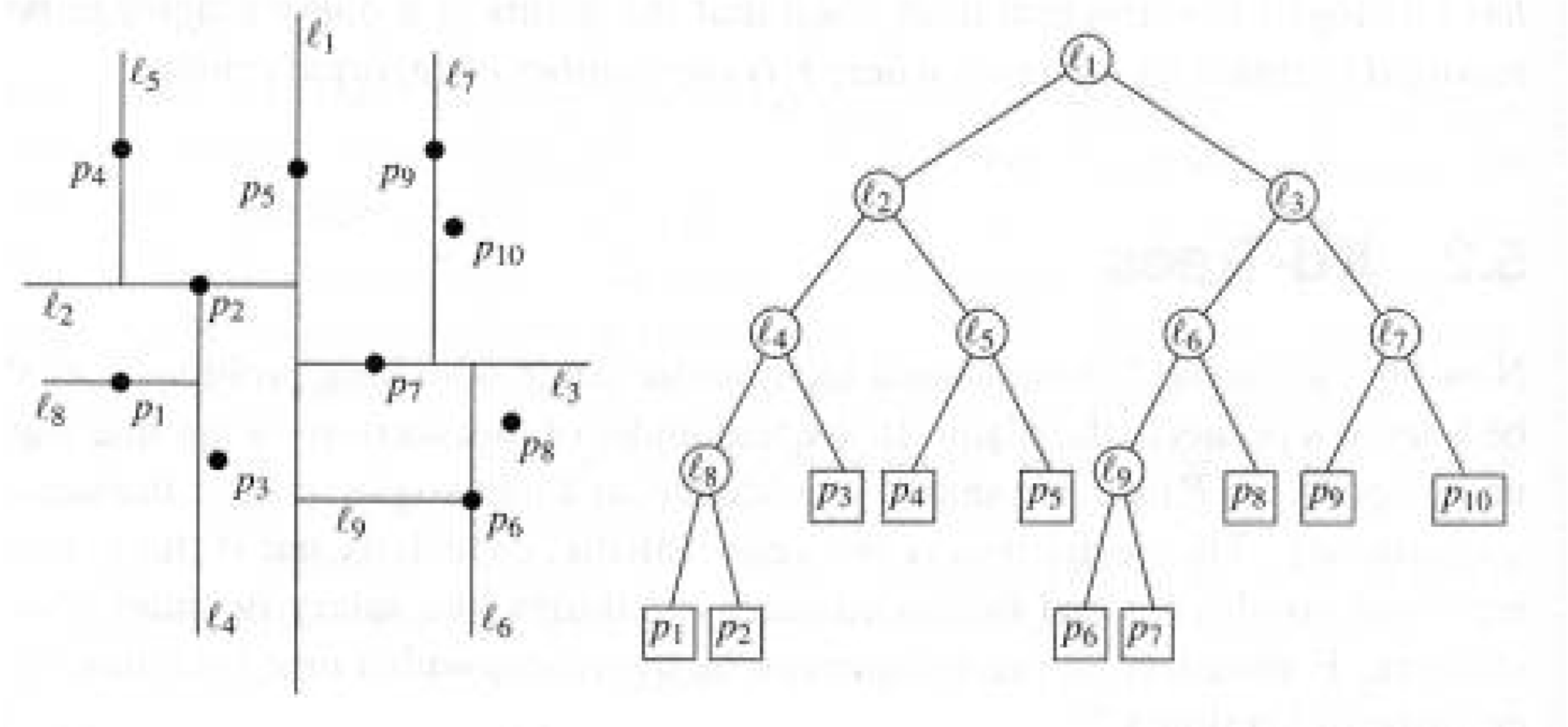


Building a K-d tree

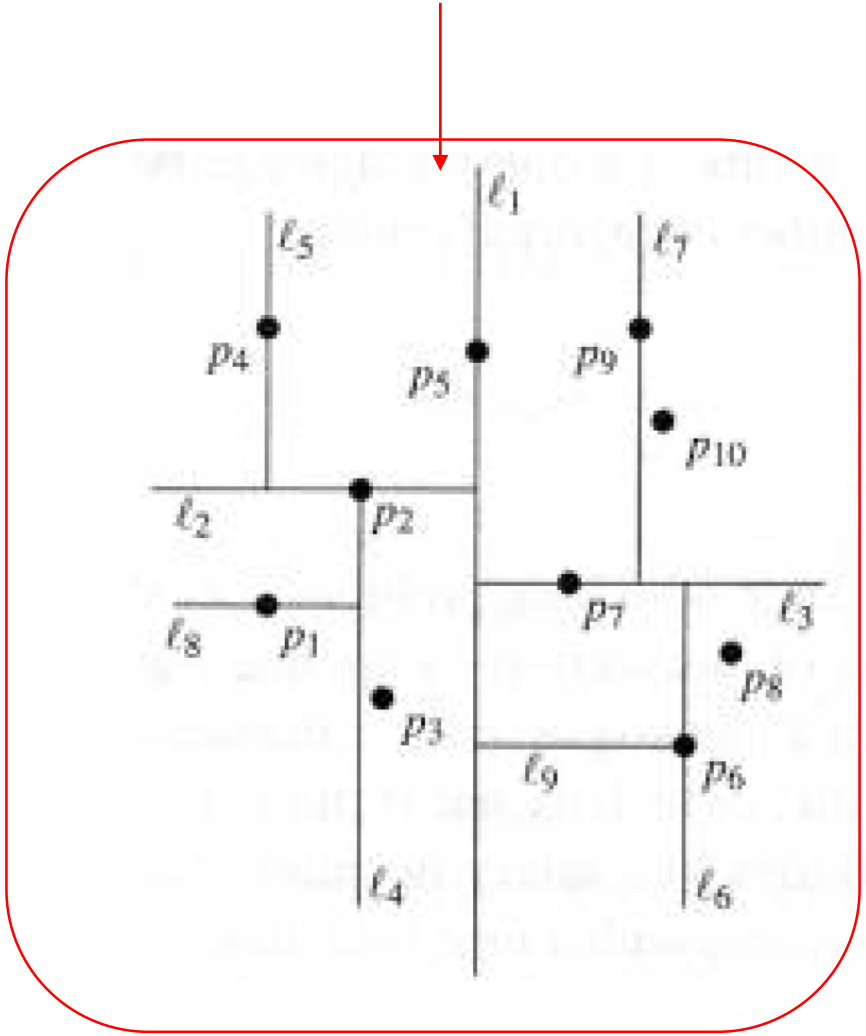
- Start with the root and gradually descend
- Initialization: assign whole X to the root
  - Repeat until each node is a singleton node
    - 1. choose one dimension, find its median value of the points corresponding to the current node
    - 2. split the data points in the current node into the half, 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

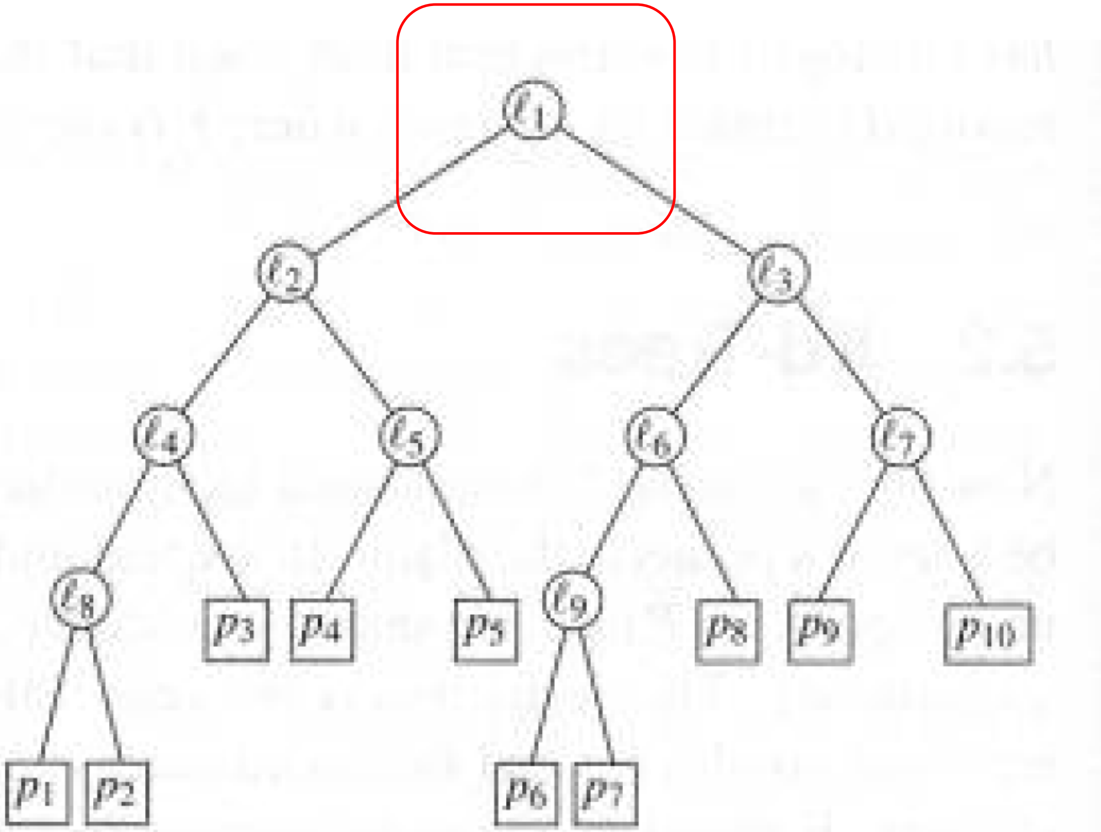
Constructing a Kd-tree

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

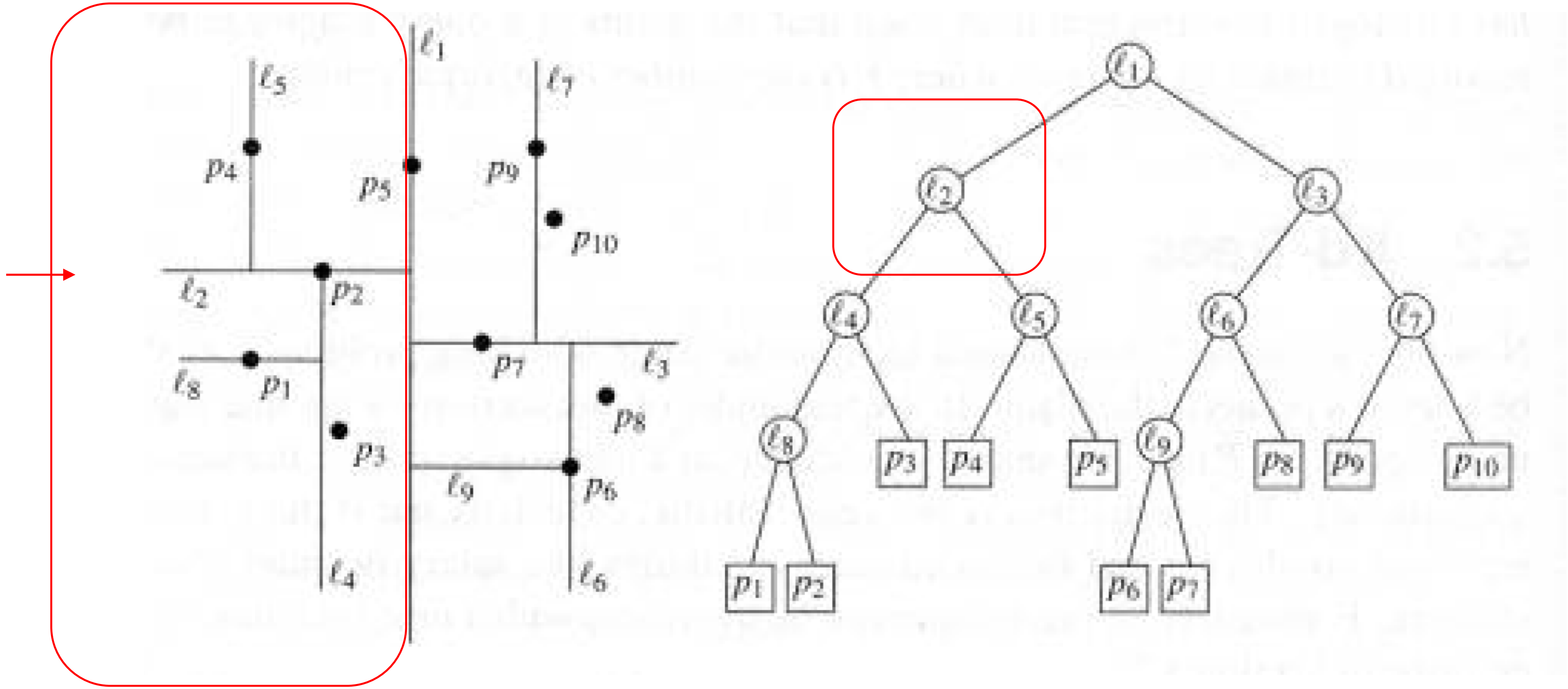


Constructing a Kd-tree

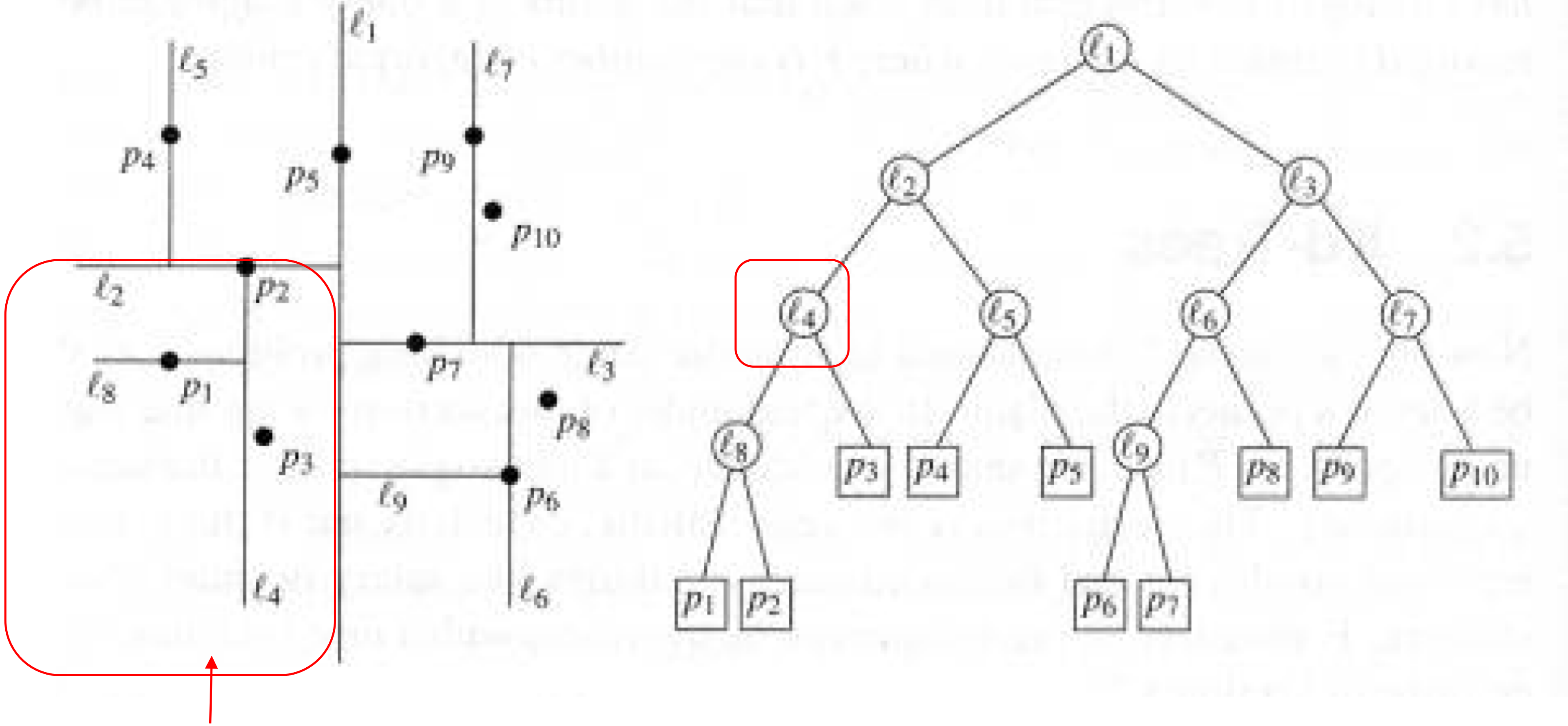




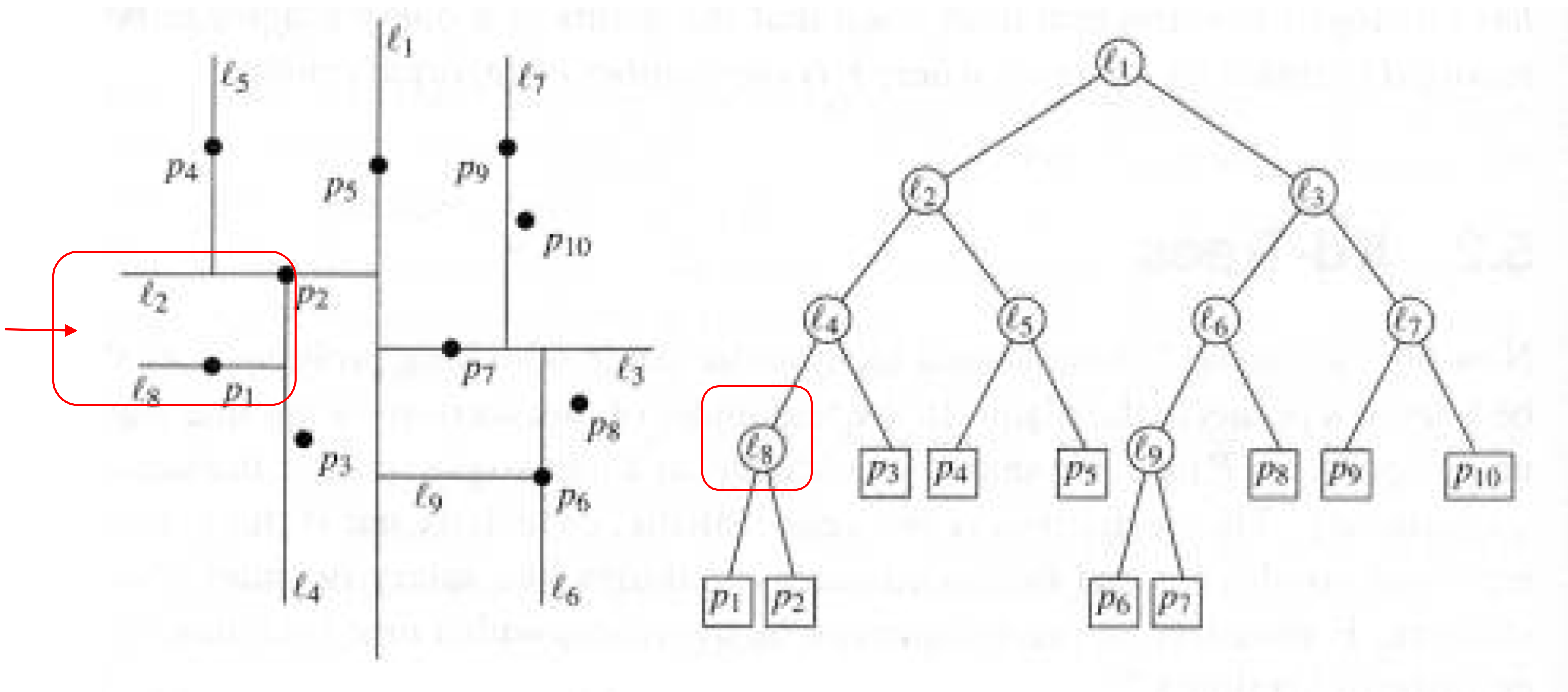
Constructing a Kd-tree



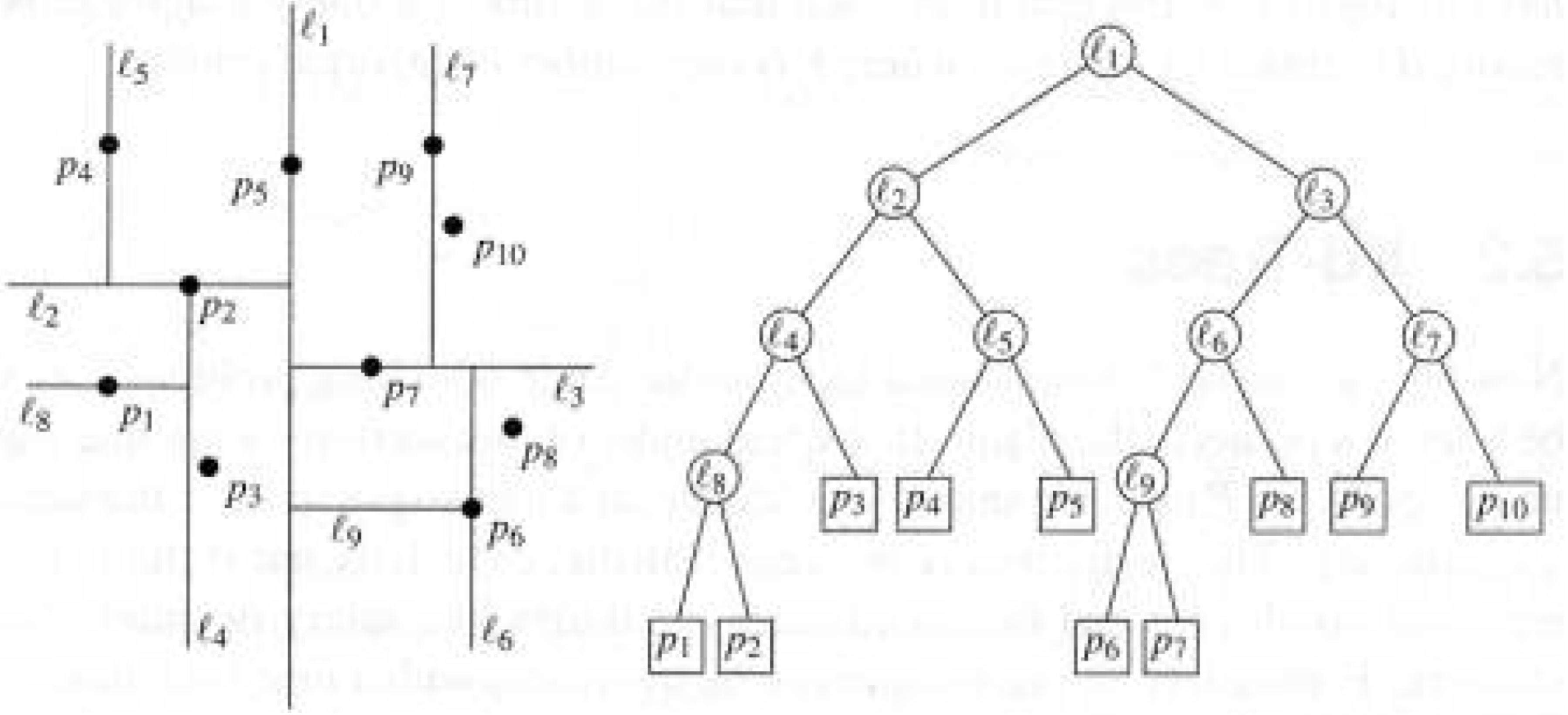
Constructing a Kd-tree



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.

if P is empty **then**
    return null
else
    $d \leftarrow D \mod M$.
    $val \leftarrow$ Median value along dimension d among points in P.
    Create new node $node$.
    $node.d \leftarrow d$.
    $node.val \leftarrow val$.
    $node.point \leftarrow$ Point at the median along dimension d.
    $node.left \leftarrow$ BuildKdTree(points in P for which value at dimension d is less than val, $D+1$).
    $node.right \leftarrow$ BuildKdTree(points in P for which value at dimension d is greater than val, $D+1$).
    Return $node$.
end if

Constructing a Kd-tree

Function BuildKdTree(P,D)

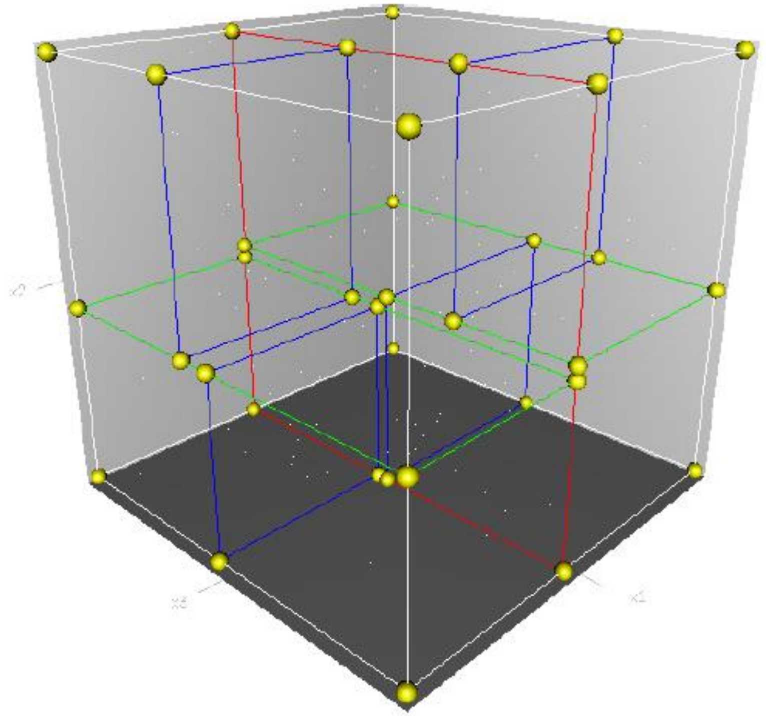
Require: A set of points P of M dimensions and current depth D.

if P is empty **then**
    return null
else
    $d \leftarrow D \mod M$.
    $val \leftarrow$ Median value along dimension d among points in P.
    Create new node $node$.
    $node.d \leftarrow d$.
    $node.val \leftarrow val$.
    $node.point \leftarrow$ Point at the median along dimension d.
    $node.left \leftarrow$ BuildKdTree(points in P for which value at dimension d is less than val, $D+1$).
    $node.right \leftarrow$ BuildKdTree(points in P for which value at dimension d is greater than val, $D+1$).
    Return $node$.
end if

Using recursion to implement it

Example with 3D

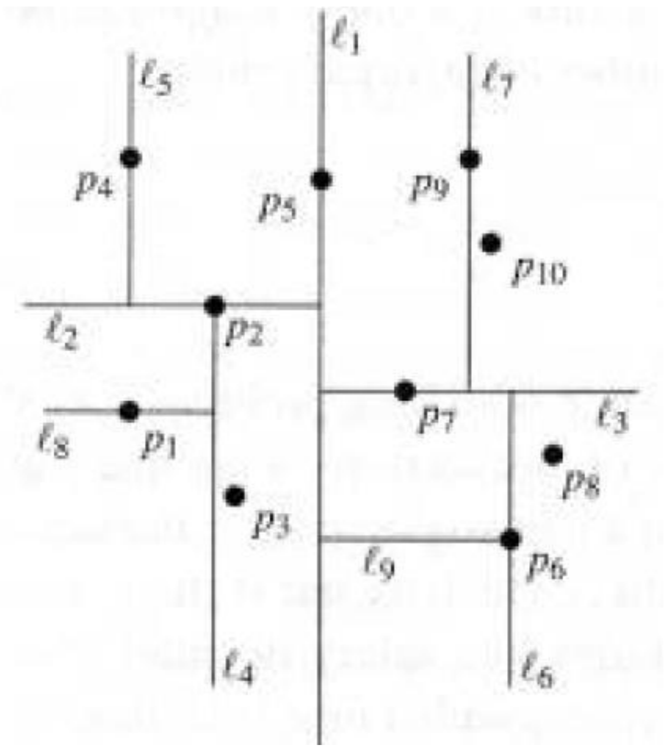
In 3D a particular split defines a separating plane (a 2-dimensional geometric entity).

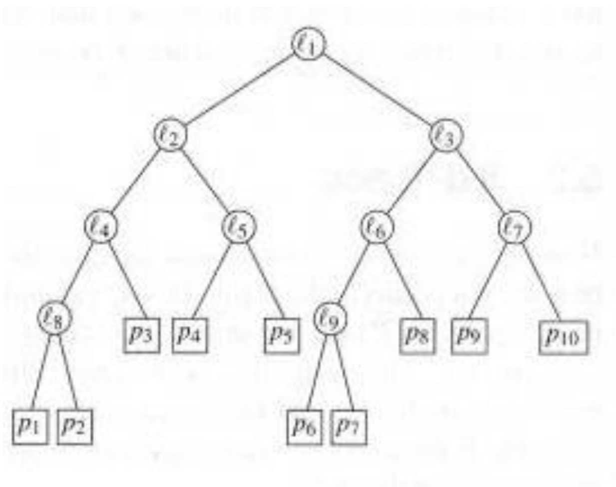


More generally, for M dimensional data, a particular split defines a separating hyperplane of $(M-1)$ dimensions.

Nearest Neighbour search with K-d tree

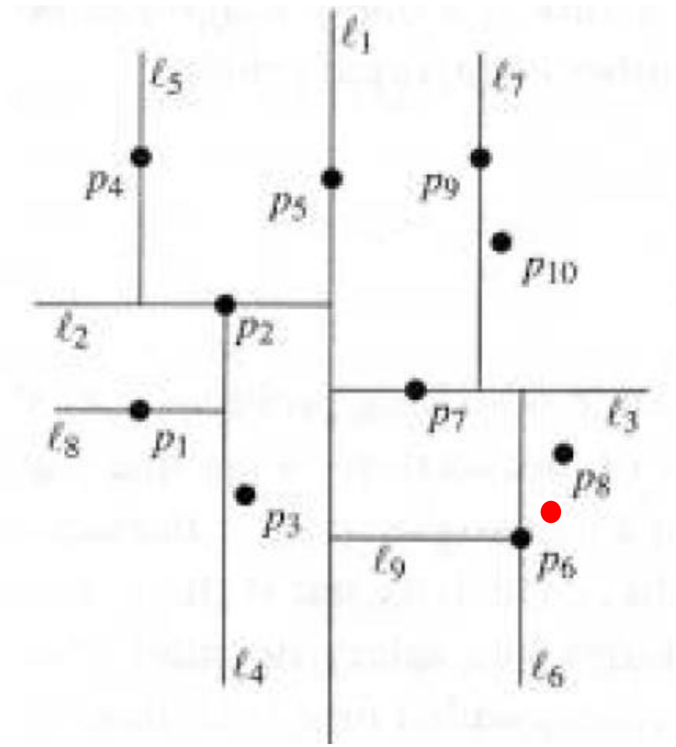
A Kd-tree partitions the M-dimensional space into a number of cells, each corresponding to a particular branch of the tree.

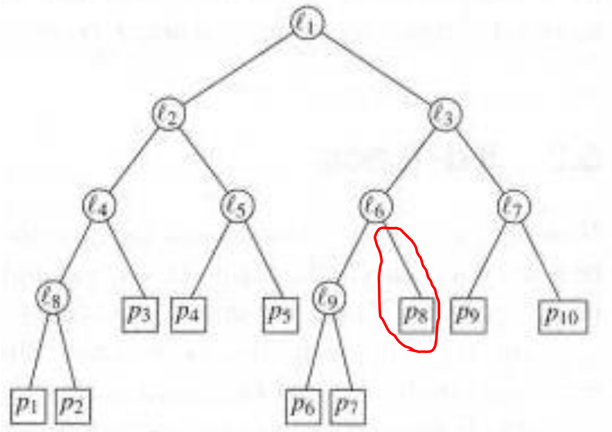




Nearest Neighbour search with K-d tree

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

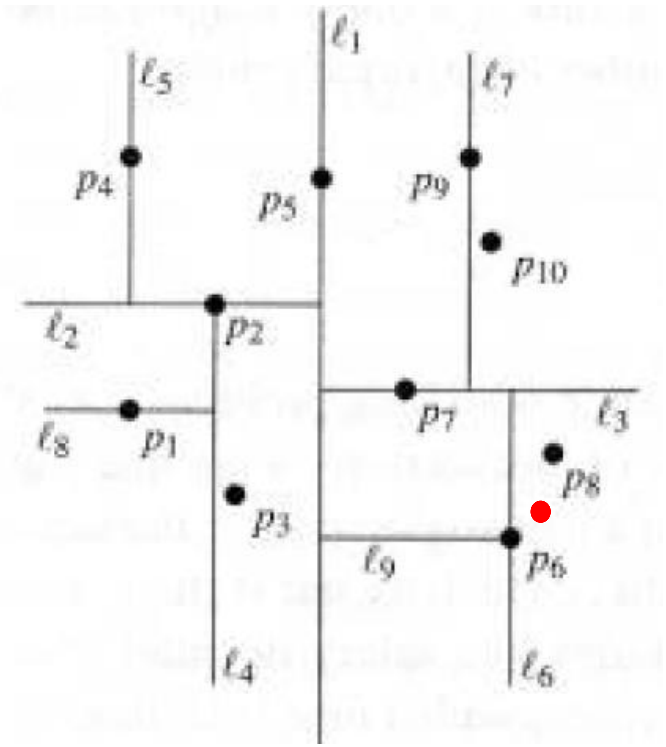


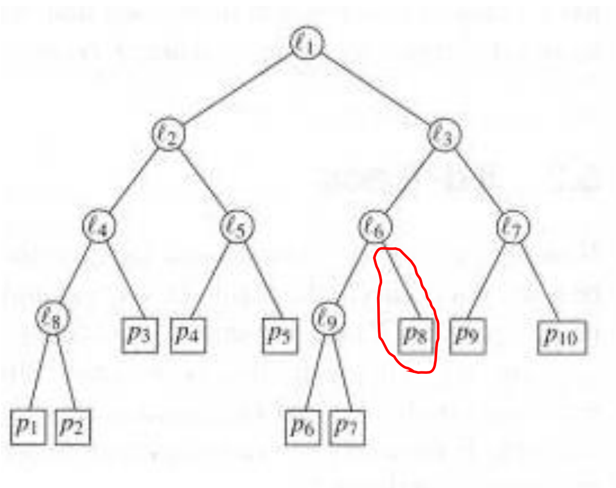


Nearest Neighbour search with K-d tree

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

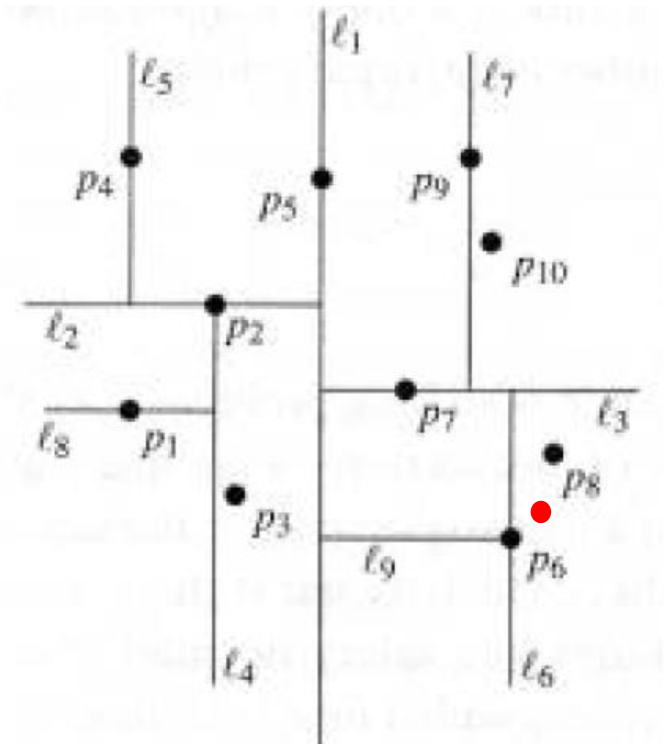
Not true!

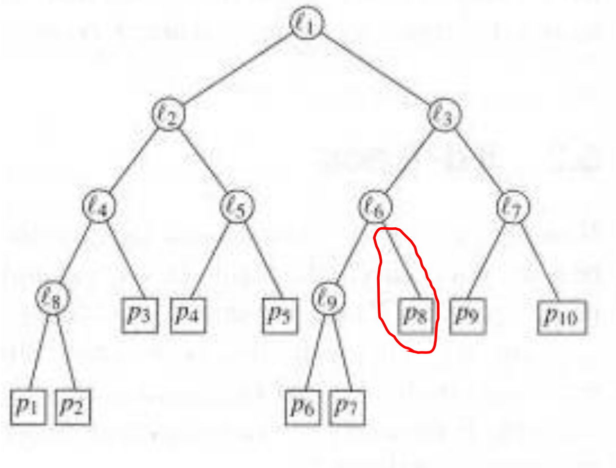




Nearest Neighbour search with K-d tree

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



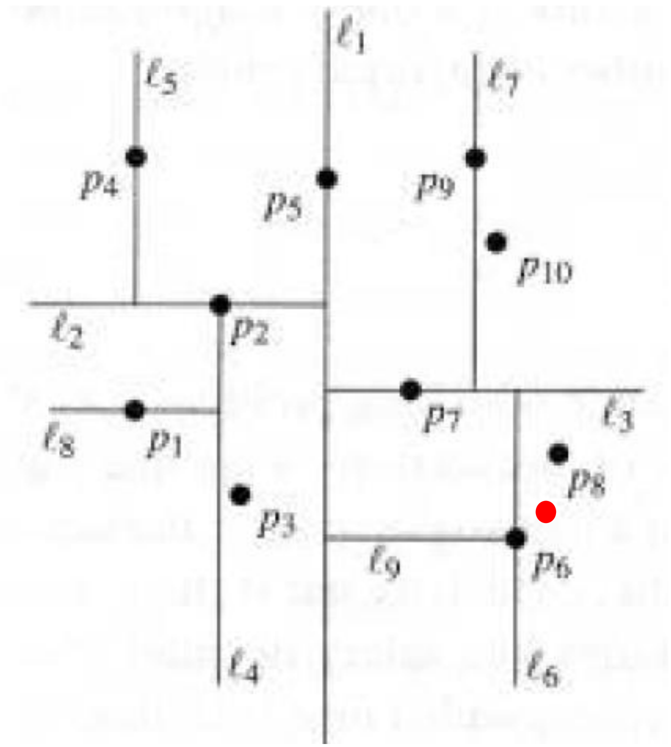


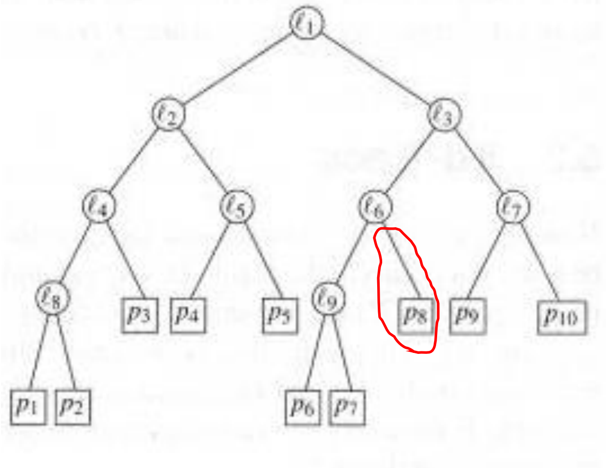
Still need to search other branches!

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 likely to locate around the neighbouring cells

Still need to search other branches!

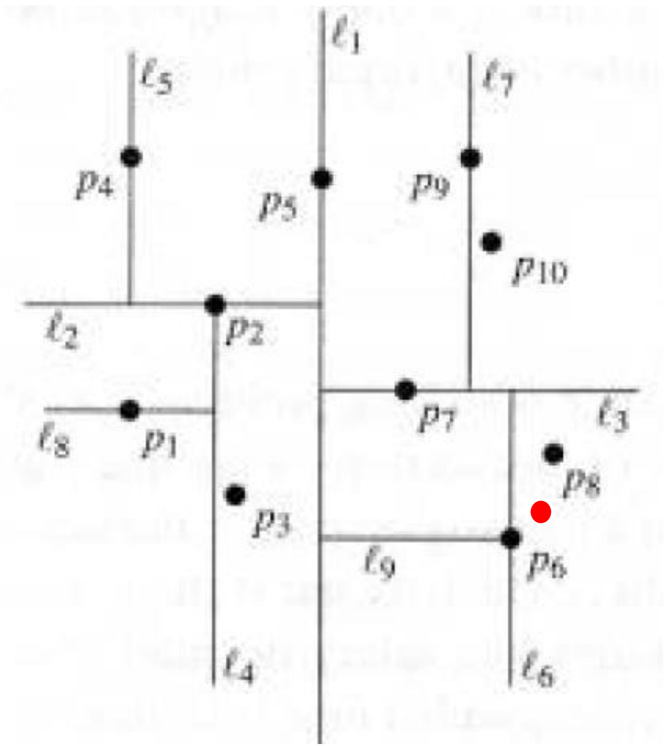


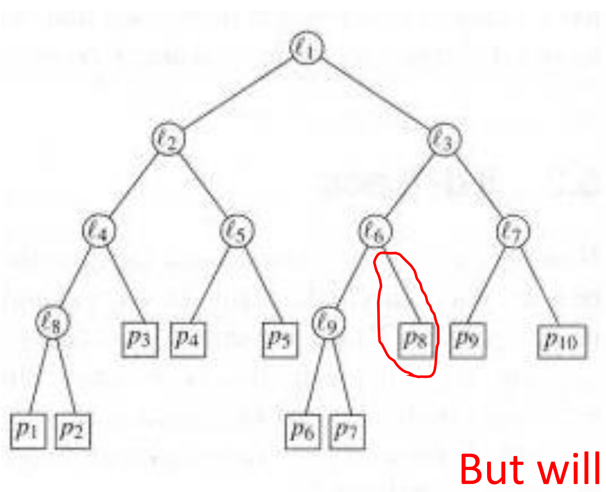


Solution: we will unwind the recursion of the tree and check if the nearest neighbour is located on the other branch of a node

Nearest Neighbour search with K-d tree

Idea: Dropping the test point down the tree until a leaf node is reached, then the nearest neighbour is more likely to locate 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

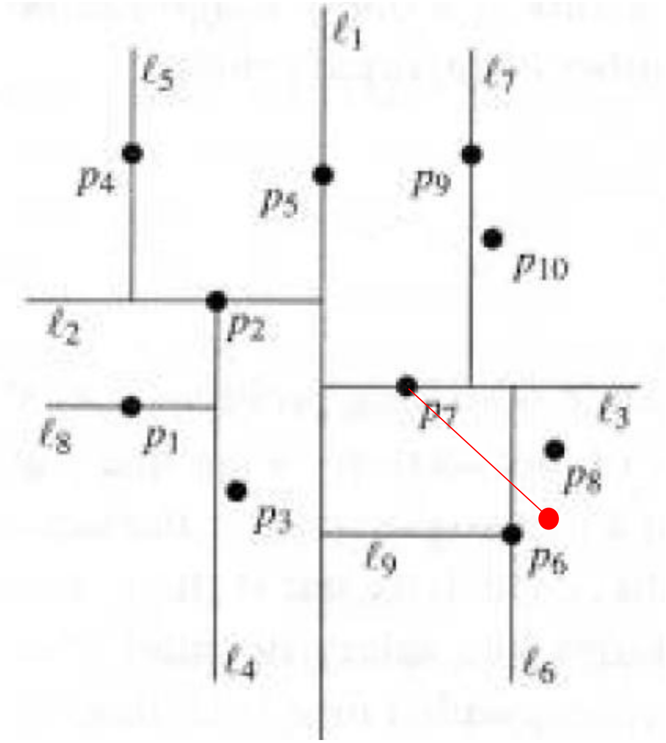
Nearest Neighbour search with K-d tree

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

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

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

$val \leftarrow$ Median value along dimension d among points in
Create new node $node$.
$node.d \leftarrow d$.
$node.val \leftarrow val$.
$node.point \leftarrow$ Point at the median along dimension d.
$node.left \leftarrow$ BuildKdTree(points in P for which value a



Nearest Neighbour search with K-d tree

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

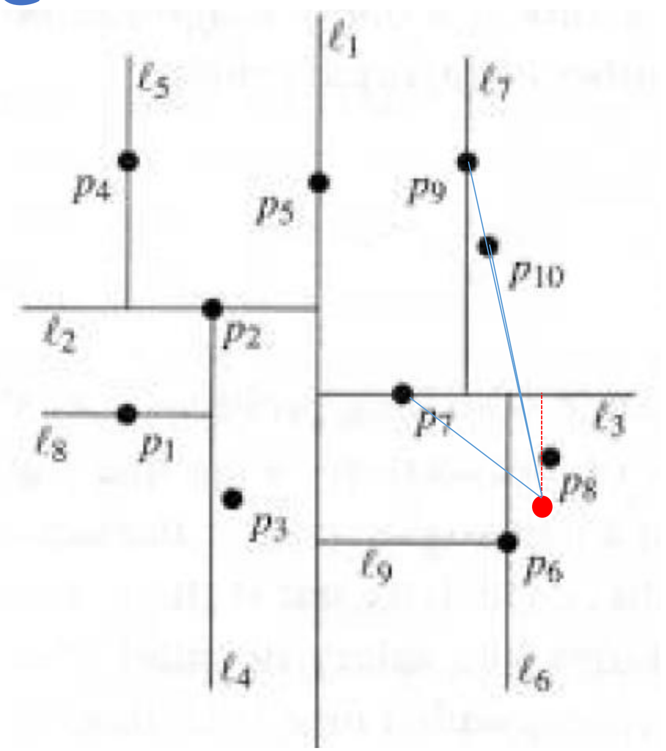
Computes the Euclidean distance of $\mathbf{y}$ to the point contained in node $node$, i.e.,

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

distance_lb($node$,$\mathbf{y}$)

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 $node.d$ to the point contained in $node$, i.e.,

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



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

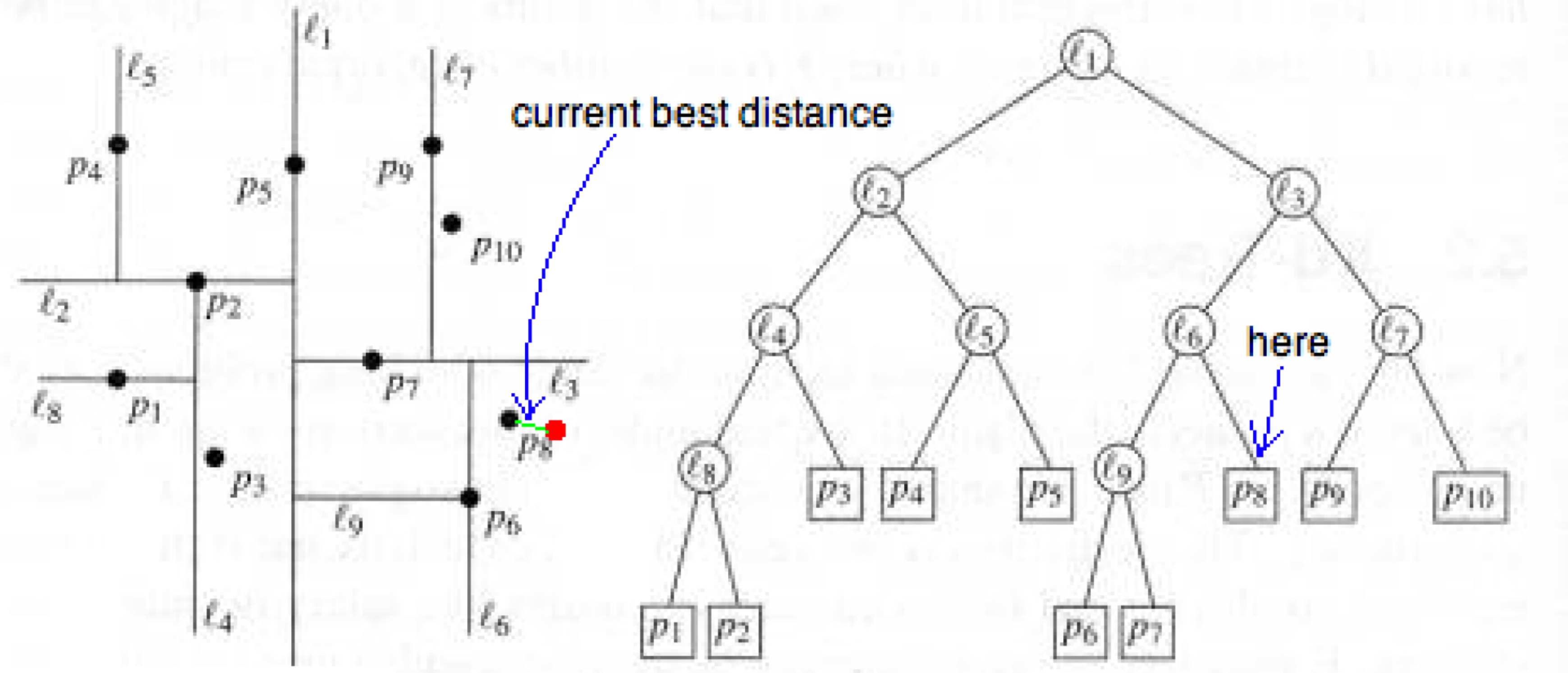
Nearest Neighbour search with K-d tree

Given a query point $\mathbf{y}$, a nearest neighbour search with a Kd-tree is accomplished by

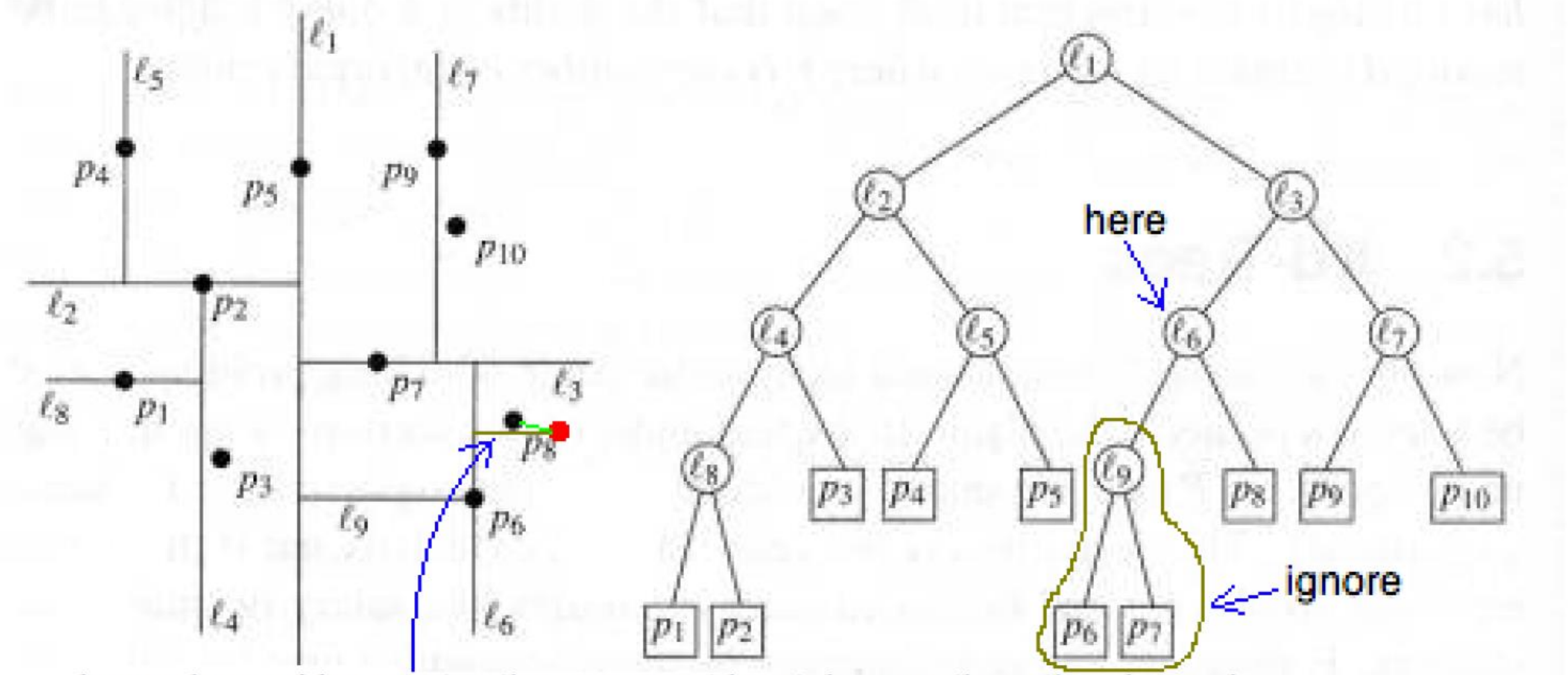
1. Dropping $\mathbf{y}$ down the tree until a leaf node is reached.
2. Record the leaf node as the current best node.
3. Unwind the recursion of the tree, calculating at each node the lower bound of the distance to $\mathbf{y}$ of the other branch.
4. If the lower bound is larger than the current best 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 $\mathbf{y}$.

Example 1

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

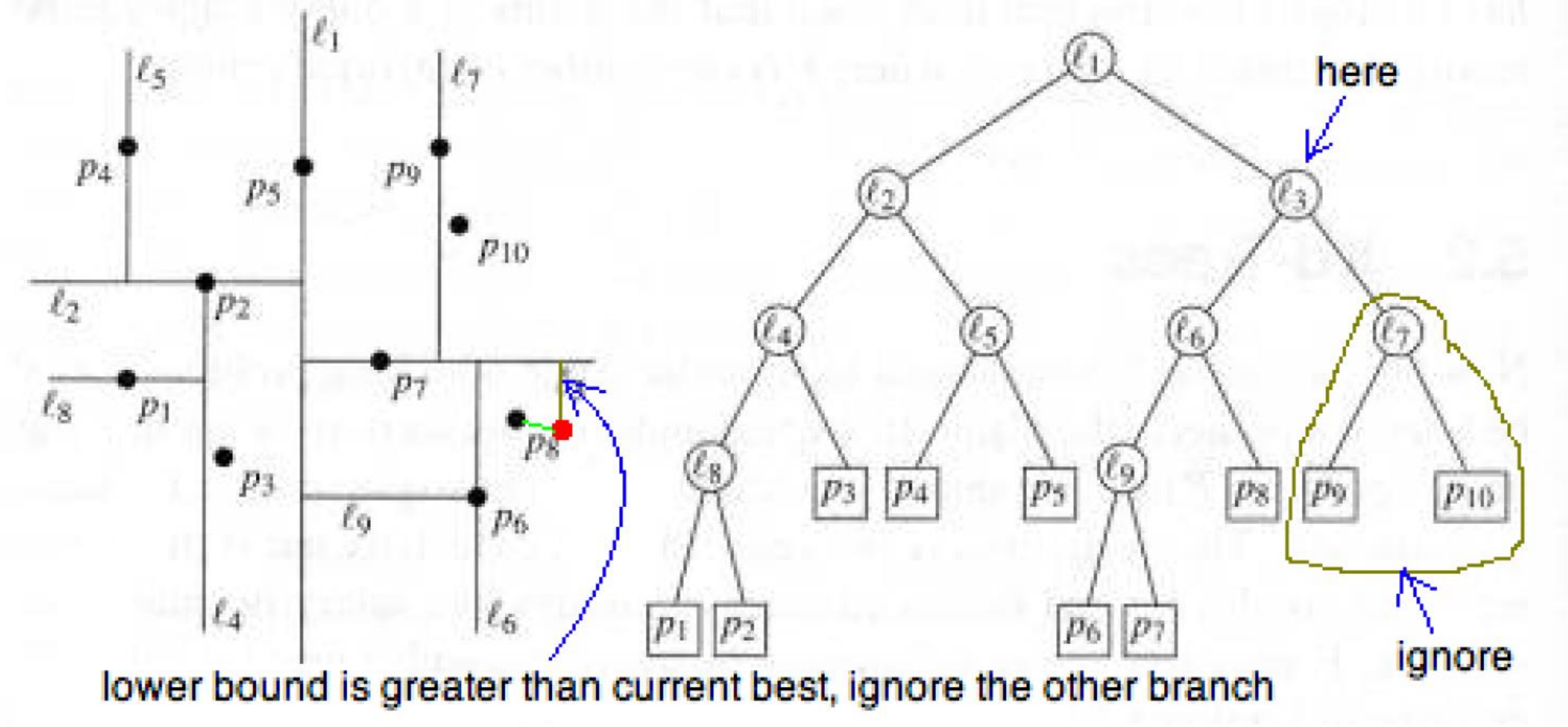


Example 1



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

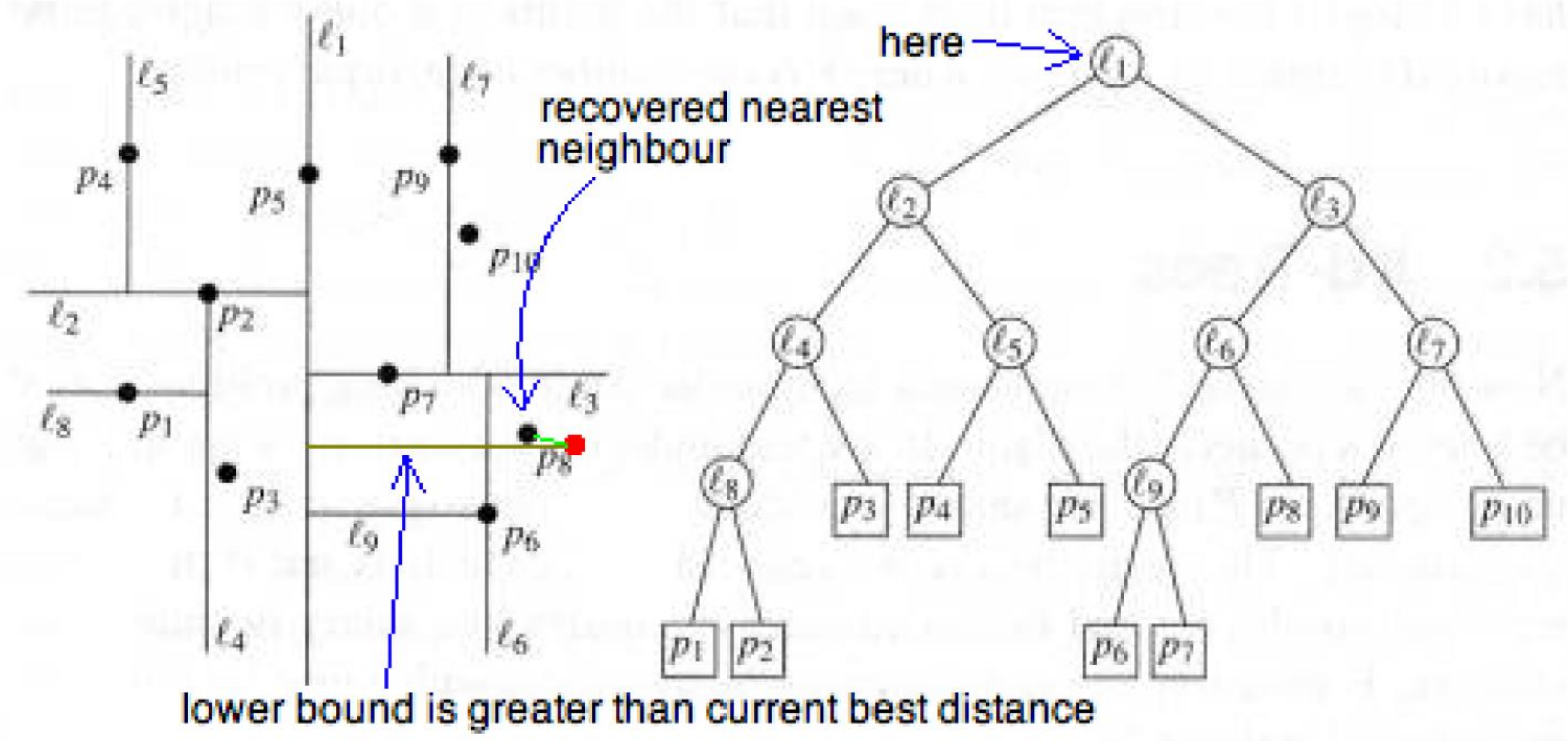
Example 1



ℓ_1
ℓ_5
ℓ_7
p_4
p_5
p_9
p_{10}
ℓ_2
p_2
p_7
ℓ_8
p_1
p_8
p_3
ℓ_9
p_6
ℓ_4
ℓ_6
here
ℓ_1
ℓ_2
ℓ_3
ℓ_4
ℓ_5
ℓ_6
ℓ_7
ℓ_8
p_3
p_4
p_5
ℓ_9
p_8
p_9
p_{10}
p_1
p_2
p_6
p_7
ignore
lower bound is greater than current best, ignore the other branch

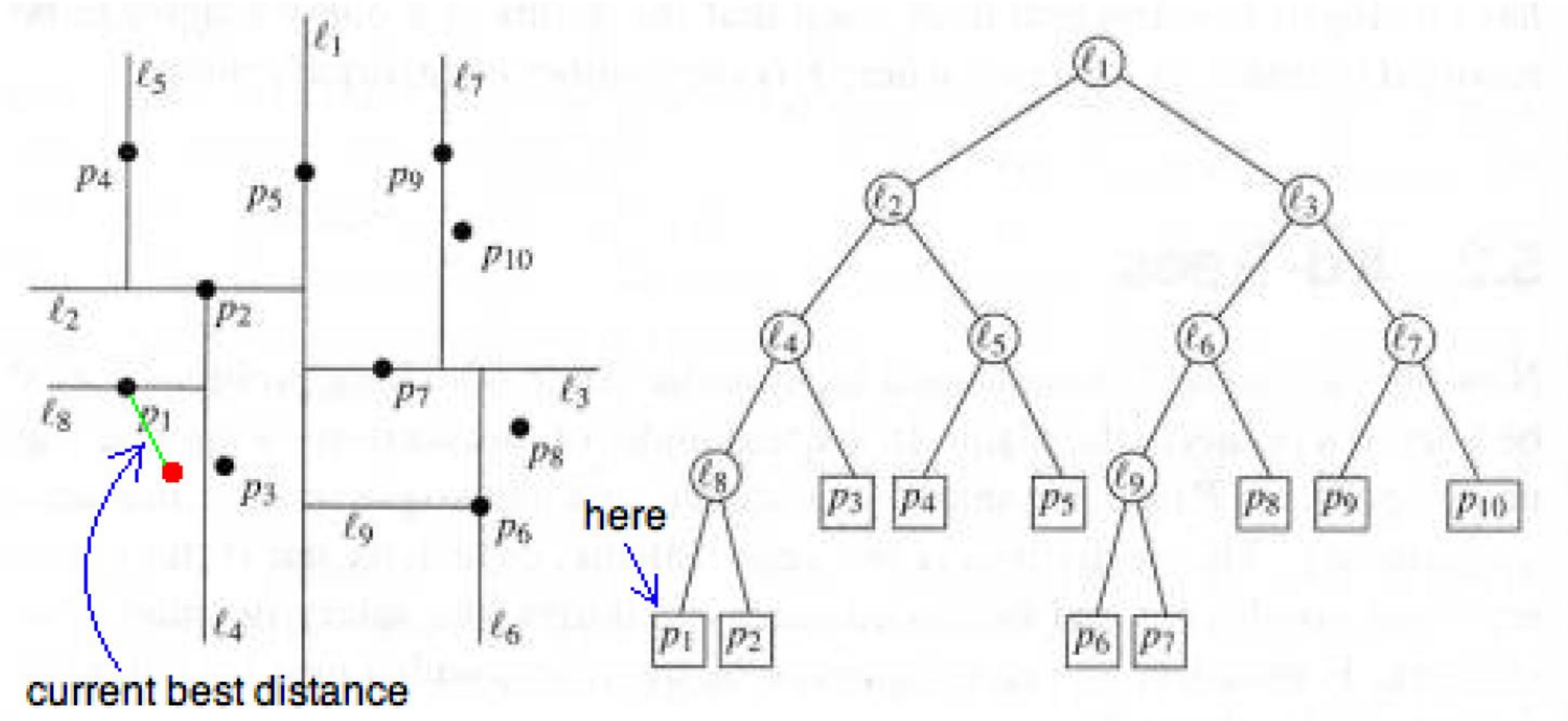
Example 1

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



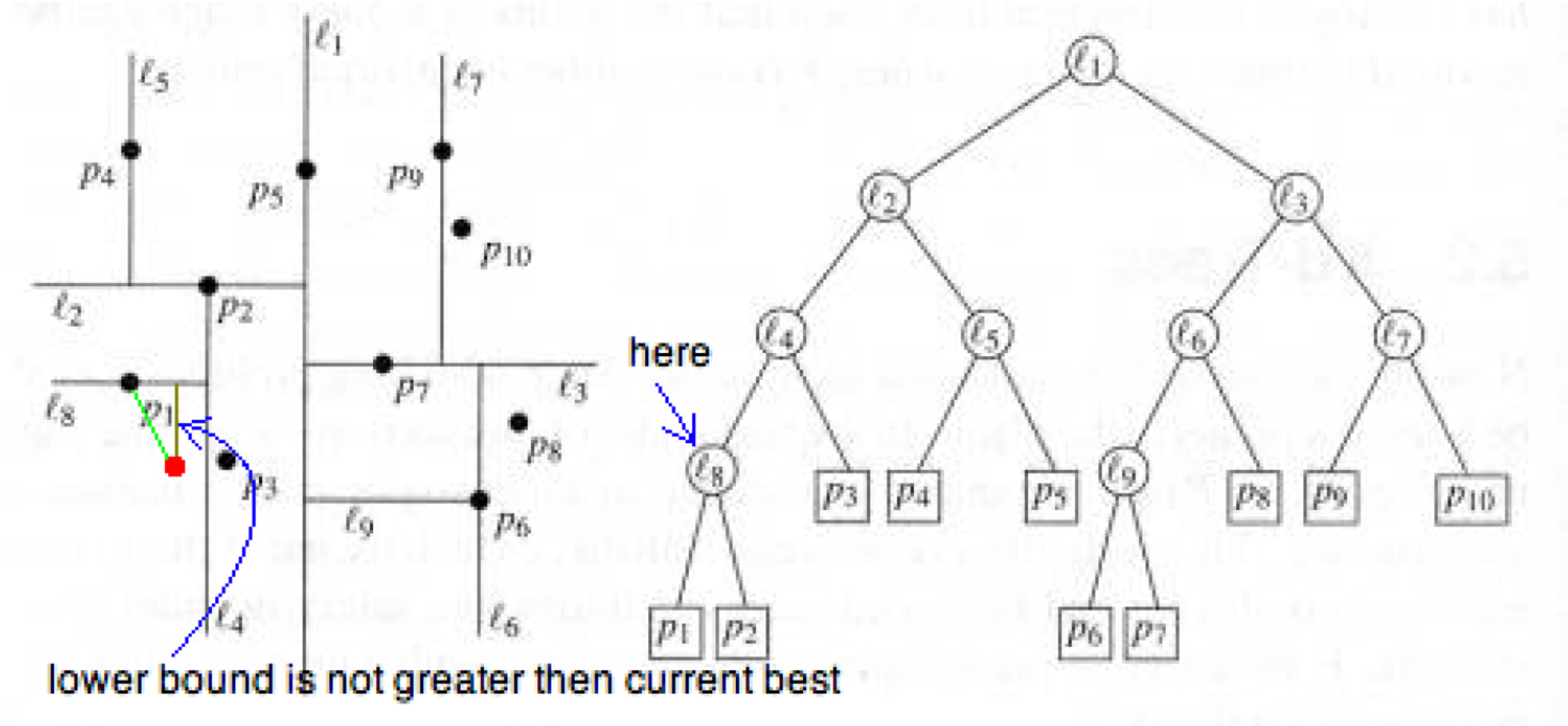
Example 2

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



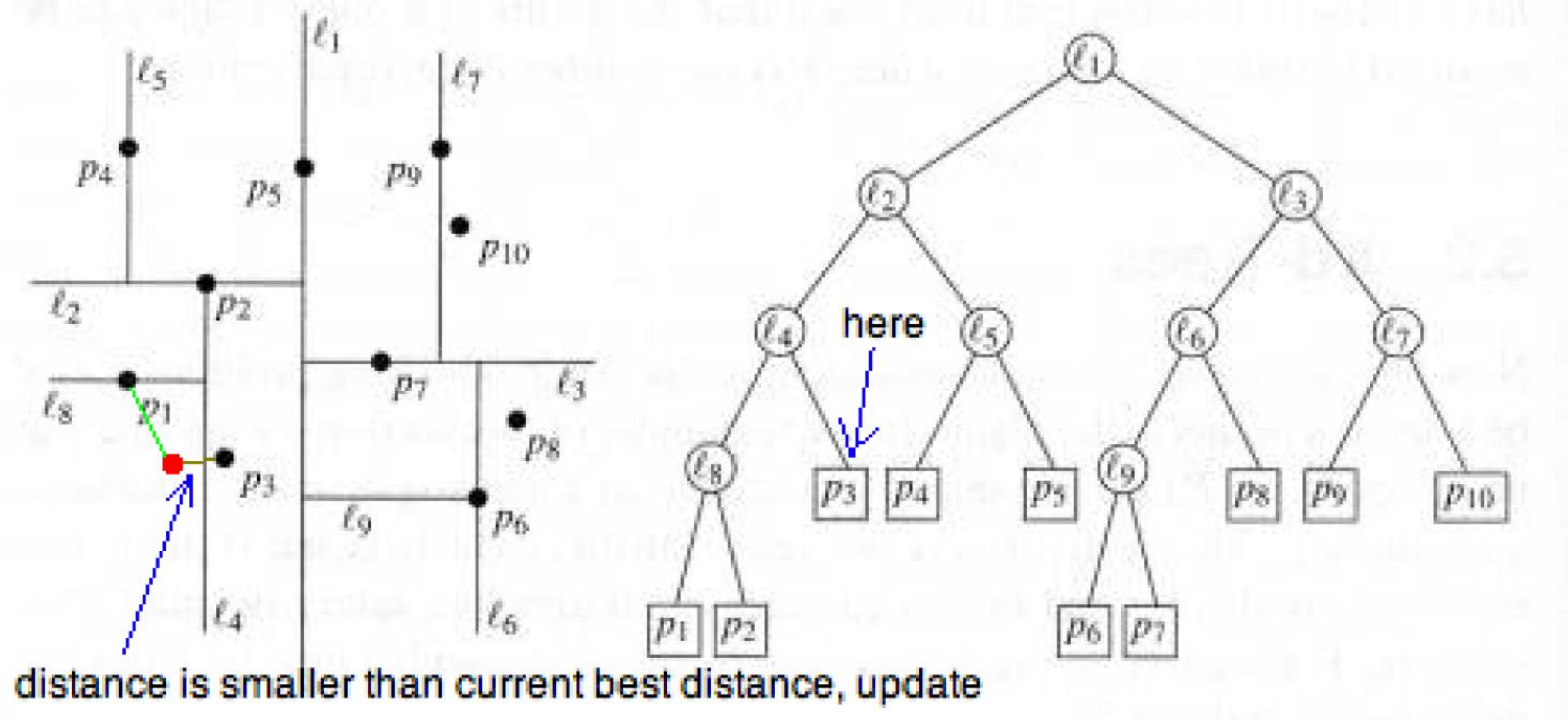
Example 2

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

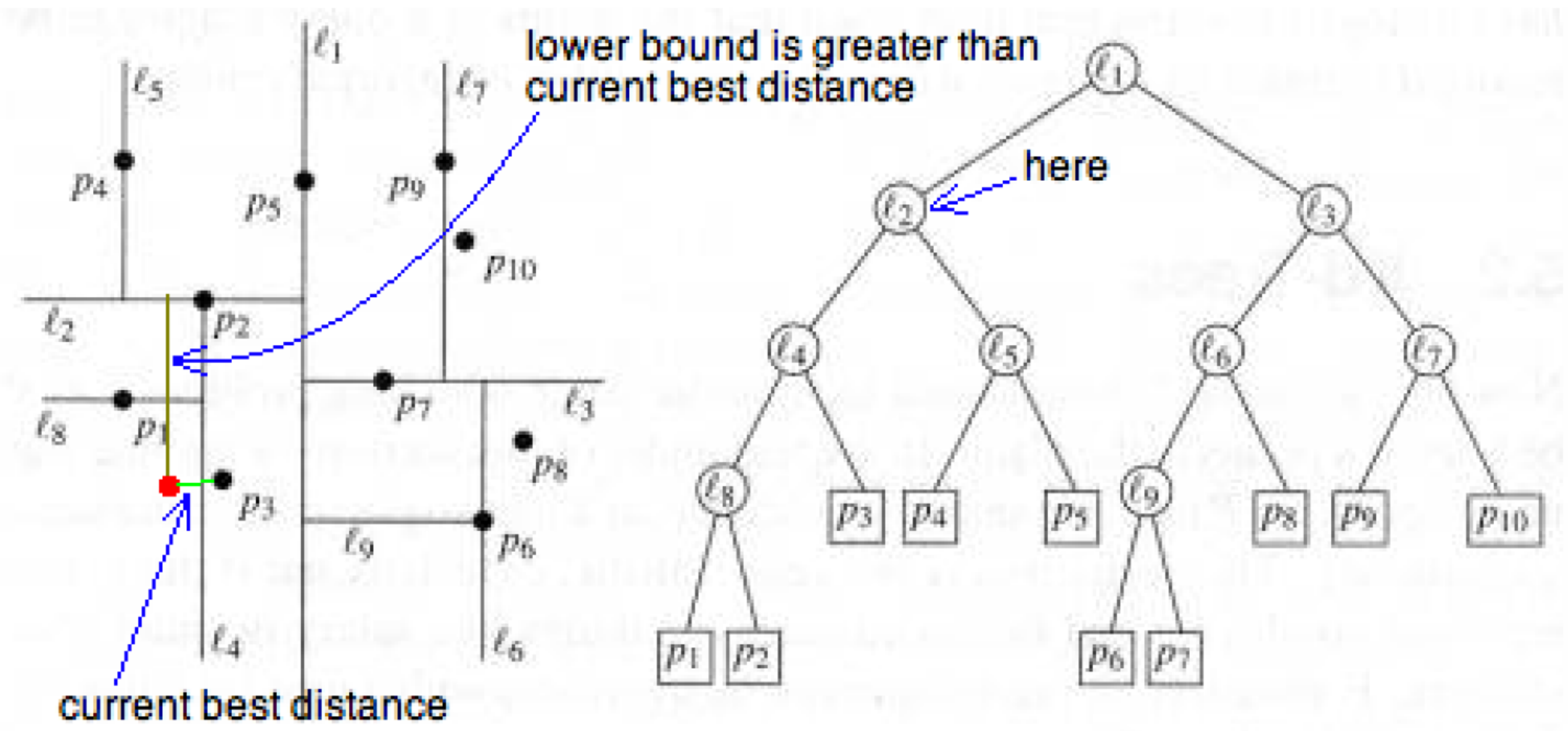


Example 2

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



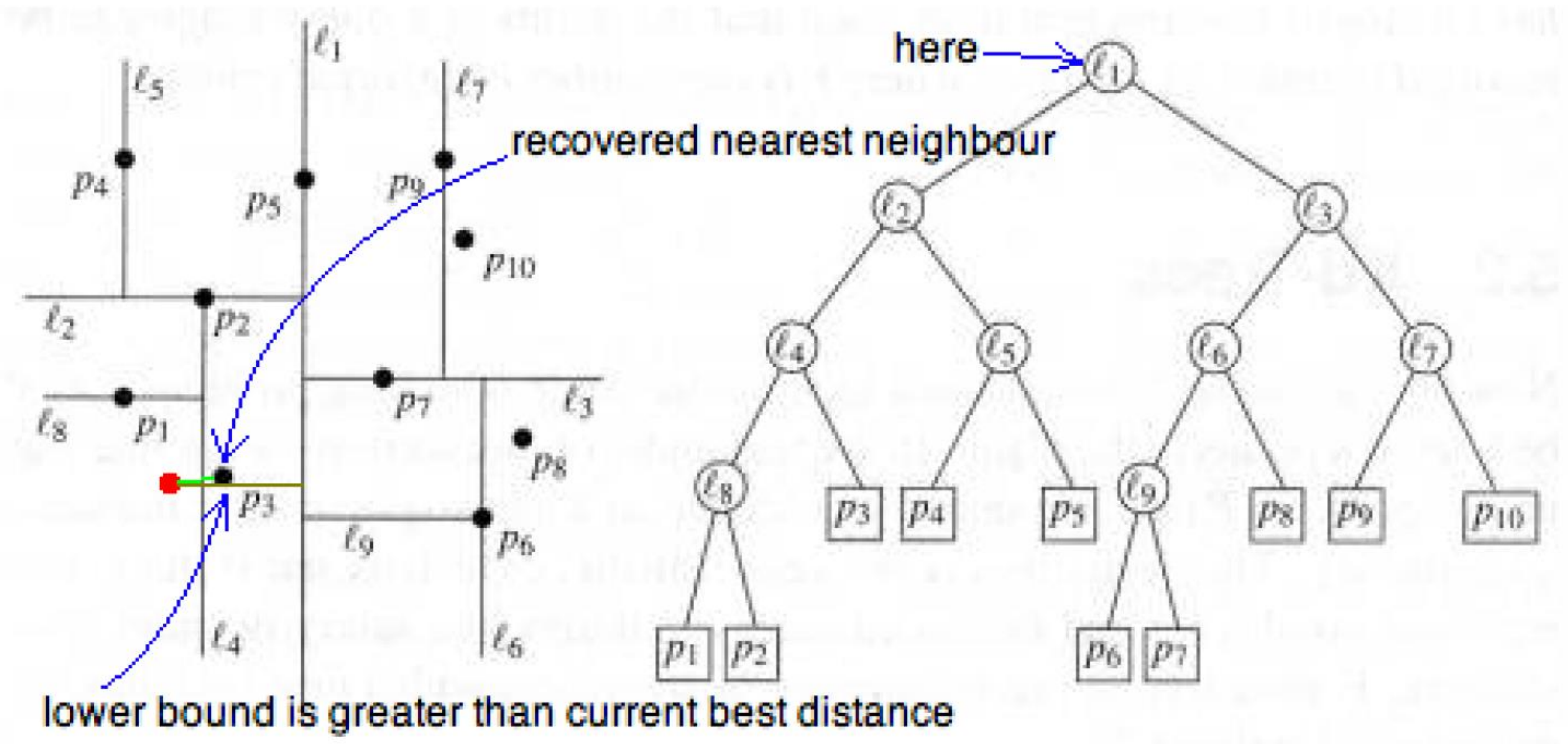
Example 2



lower bound is greater than
current best distance
here
current best distance
ℓ_1
ℓ_2
ℓ_3
ℓ_4
ℓ_5
ℓ_6
ℓ_7
ℓ_8
ℓ_9
p_1
p_2
p_3
p_4
p_5
p_6
p_7
p_8
p_9
p_{10}

Example 2

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



Theoretic behaviour

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

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

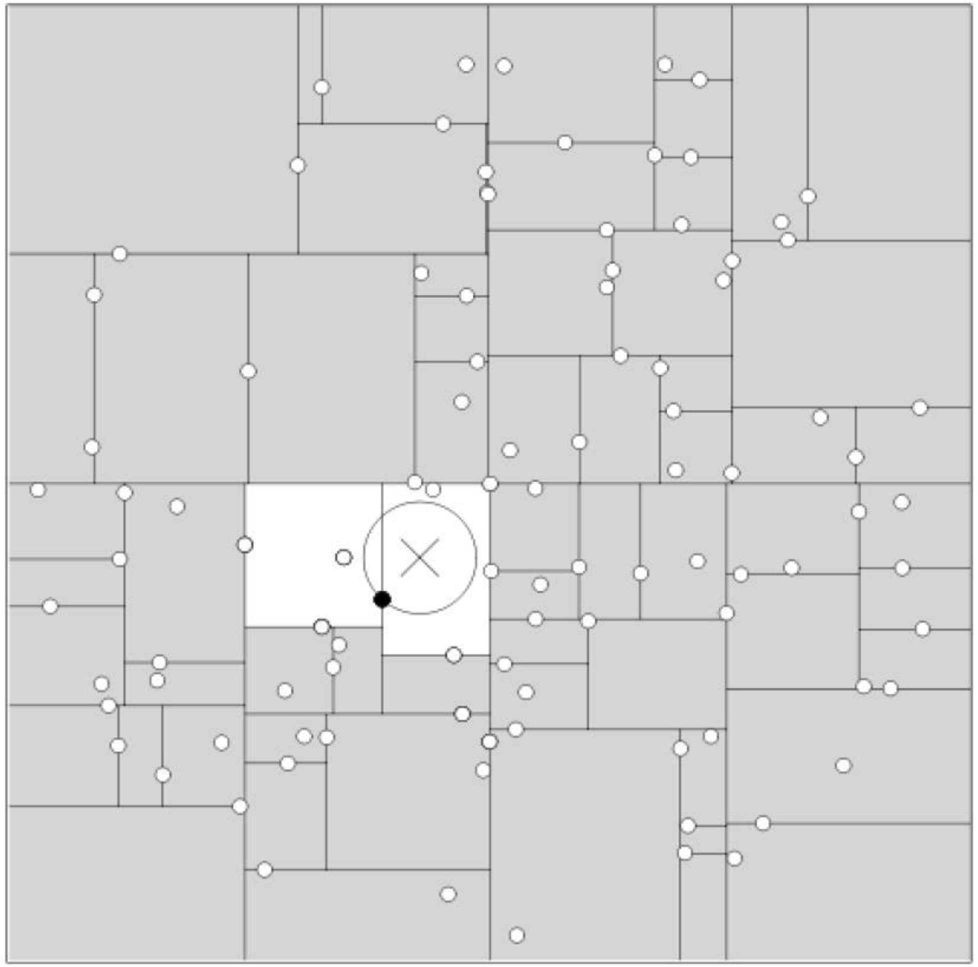
We are more concerned with the search performance— This depends on how many times we need to descend into the separate branches. Unfortunately this is hard to asymptotically analysed since the speed depends on how the data is distributed.

It is clear that at least $\mathcal{O}(\log N)$ 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 searched. In the worst case the algorithm reduces to a naive exhaustive search.

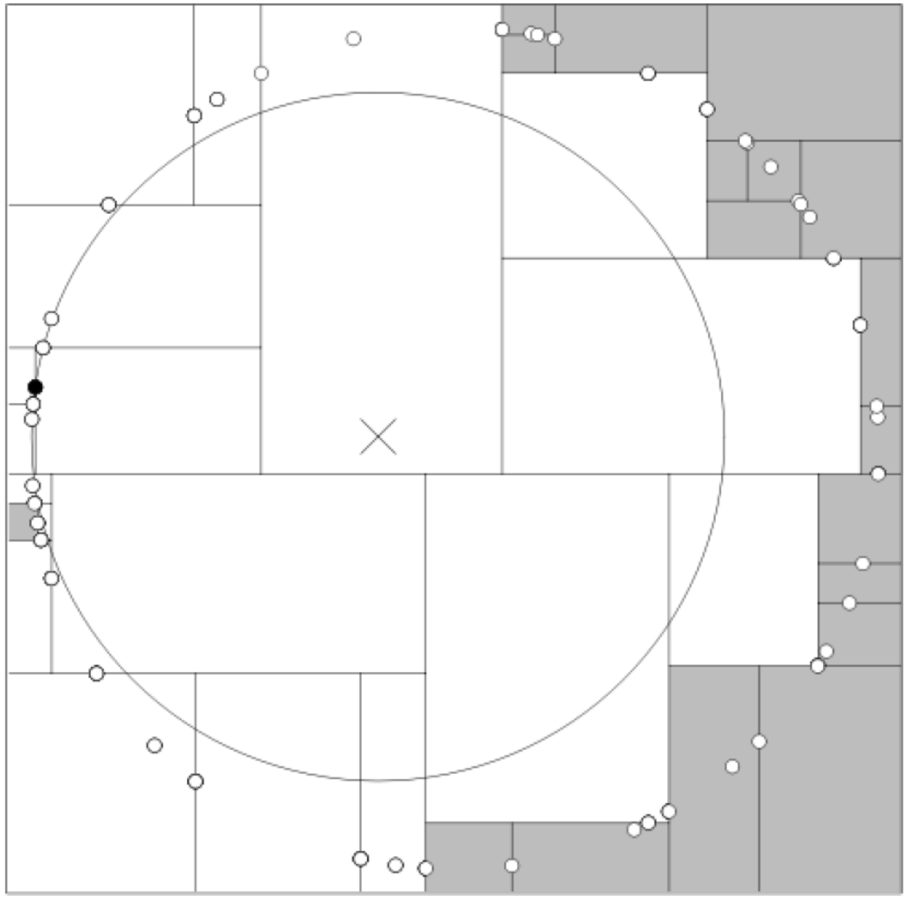
The distribution affects the search performance (cont.)

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-trees can provide the highest gains in efficiency if the number of points N is much greater than 2^M, where M is the dimensionality of the data.

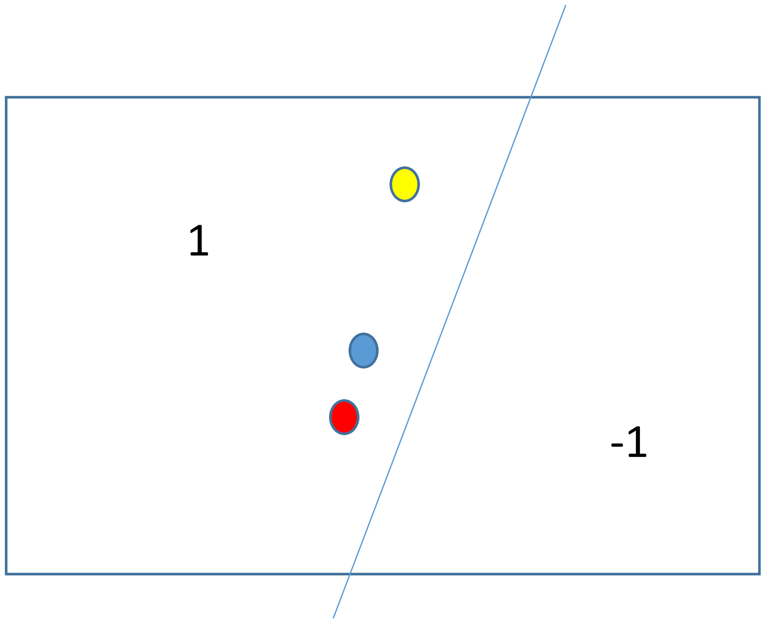
If the rule is not satisfied in most cases the Kd-tree search reduces to an exhaustive search.

Efficient Nearest Neighbour Search [optional]

- Approximate Nearest Neighbour Search
  - Idea: generate key for each data point $x_i \rightarrow s_i$
  - 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)

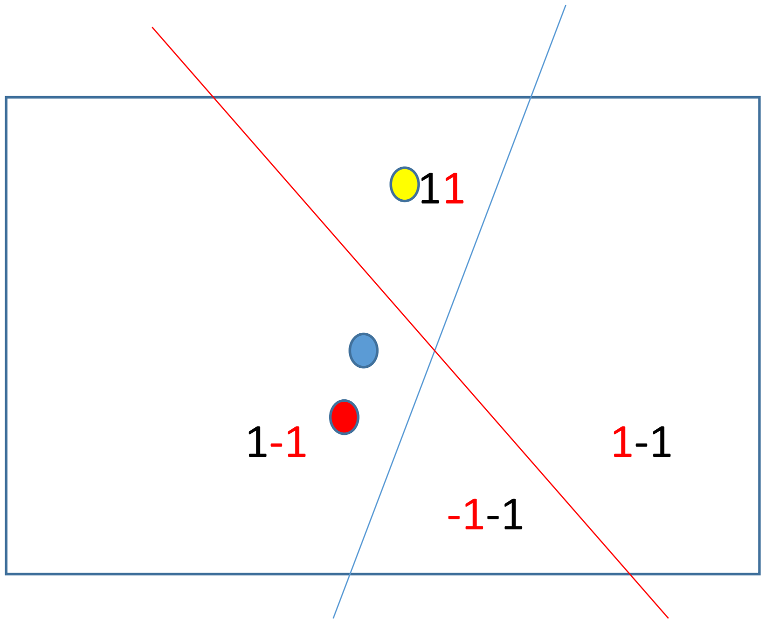
Efficient Nearest Neighbour Search [optional]

- Example: Locality Sensitive Hashing
- $S_k = sign(w_k^T x_i)$



Efficient Nearest Neighbour Search [optional]

- Example: Locality Sensitive Hashing
- $S_k = sign(w_k^T x_i)$



Efficient Nearest Neighbour Search [optional]

- Example: Locality Sensitive Hashing (LSH)
- $S_k = sign\left(w_k^T x_i\right)$
- More projections, finer partition
- Closer points are more likely

to share the same keys