Computational Geometry Lecture
Range Searching

INSTITUT FÜR THEORETISCHE INFORMATIK · FAKULTÄT FÜR INFORMATIK

Chih-Hung Liu · Tamara Mchledidze

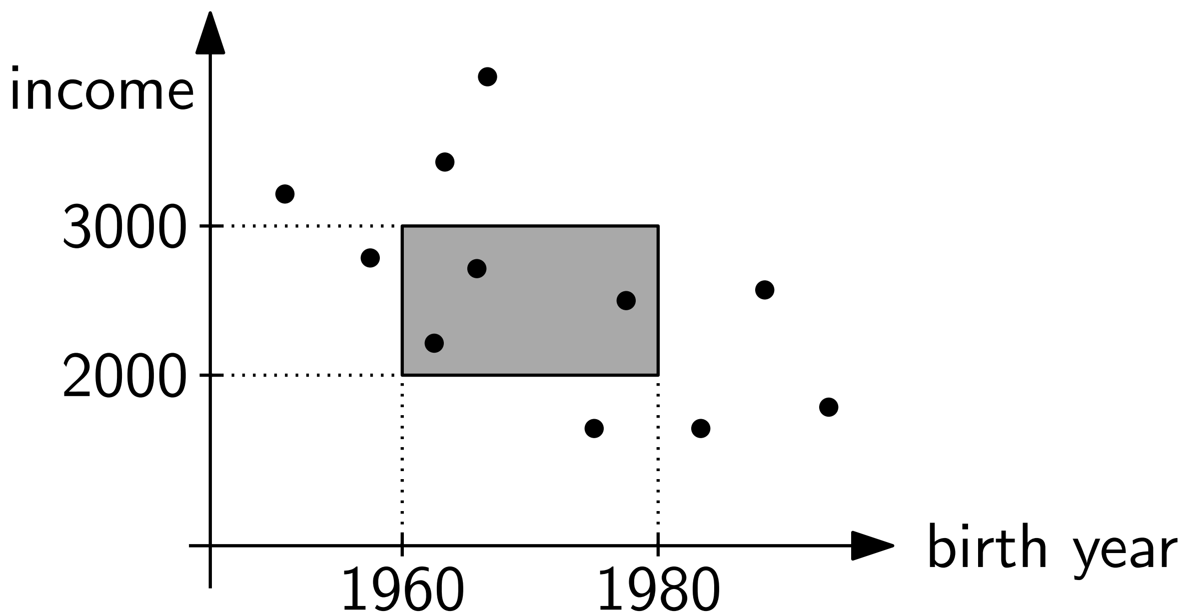
20.06.2018

Geometry in Databases

In a personnel database, the employees of a company are anonymized and their monthly income and birth year are saved. We now want to perform a search: which employees have an income between 2,000 and 3,000 Euro and were born between 1960 and 1980?

Geometry in Databases

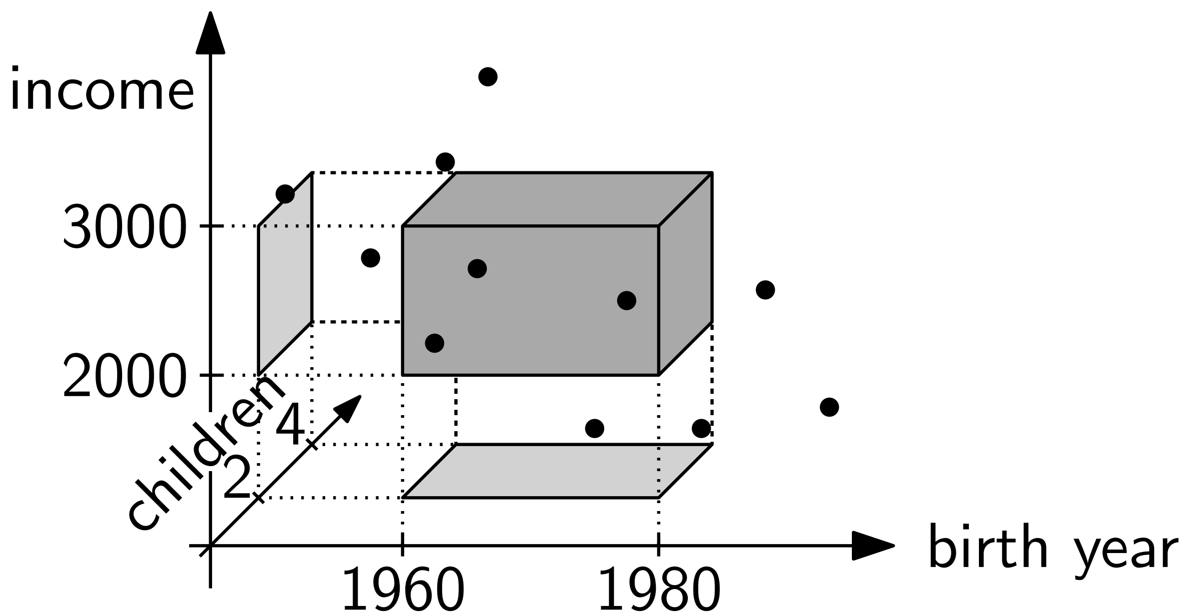
In a personnel database, the employees of a company are anonymized and their monthly income and birth year are saved. We now want to perform a search: which employees have an income between 2,000 and 3,000 Euro and were born between 1960 and 1980?



Geometric Interpretation:
Entries are points: (birth year, income level) and the query is an axis-parallel rectangle

Geometry in Databases

In a personnel database, the employees of a company are anonymized and their monthly income and birth year are saved. We now want to perform a search: which employees have an income between 2,000 and 3,000 Euro and were born between 1960 and 1980?



This problem can easily be generalized to d dimensions.

Orthogonal Range Queries

Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Orthogonal Range Queries

Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Orthogonal Range Queries

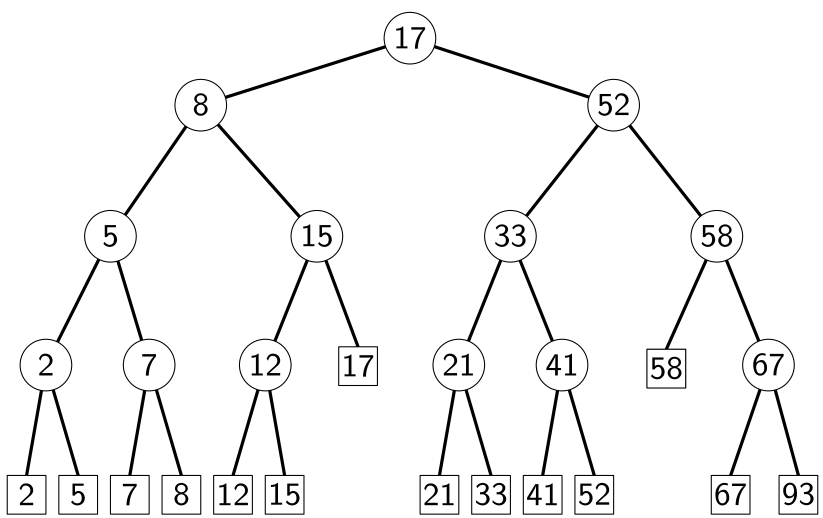
Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Solution: Balanced binary search tree:

- Stores points in the leaves
- Internal node v stores pivot value x_v



Orthogonal Range Queries

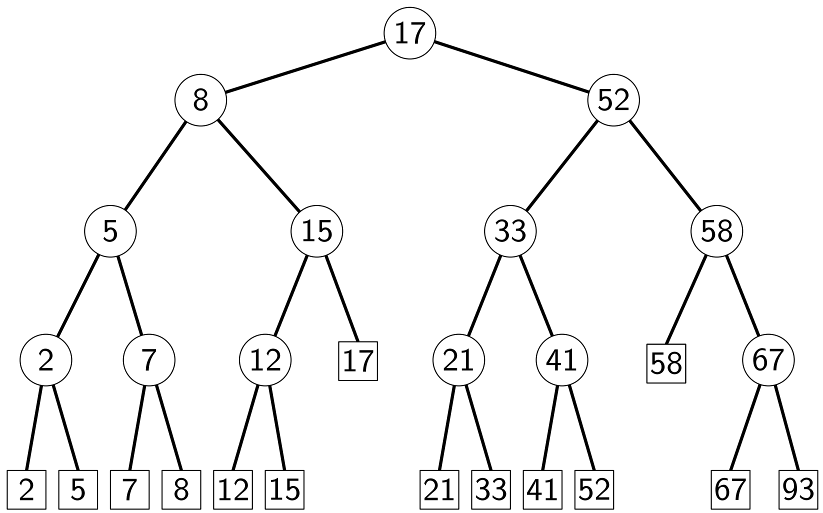
Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Solution: Balanced binary search tree:

- Stores points in the leaves
- Internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

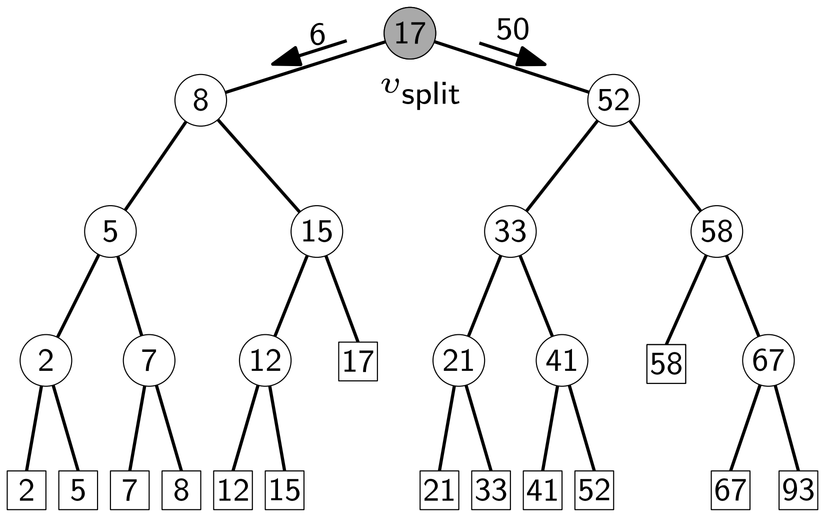
Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Solution: Balanced binary search tree:

- Stores points in the leaves
- Internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

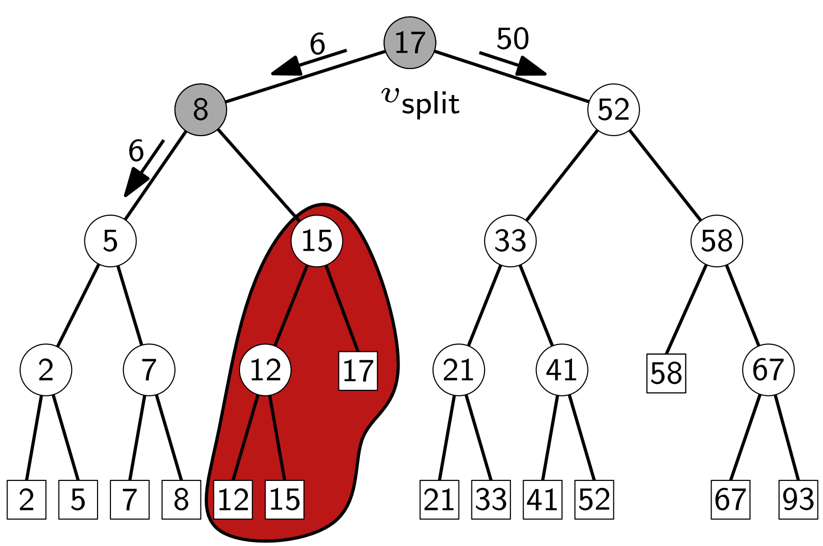
Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Solution: Balanced binary search tree:

- Stores points in the leaves
- Internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

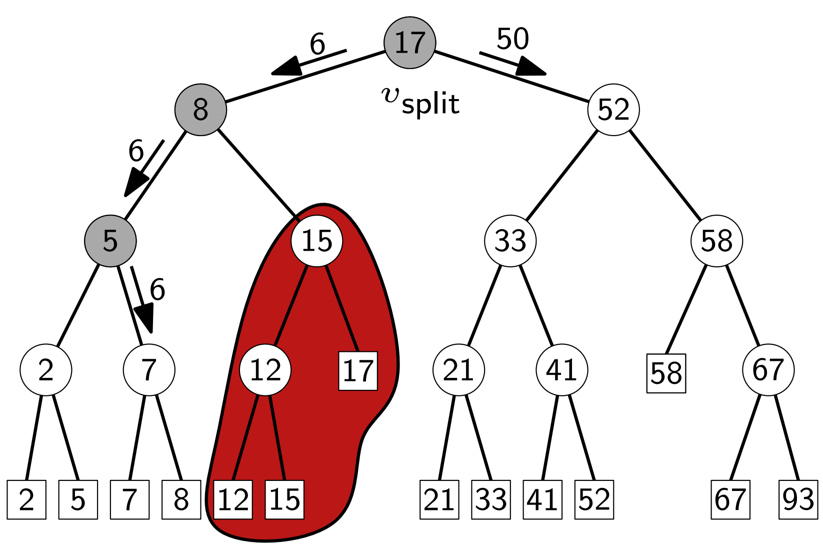
Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Solution: Balanced binary search tree:

- Stores points in the leaves
- Internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

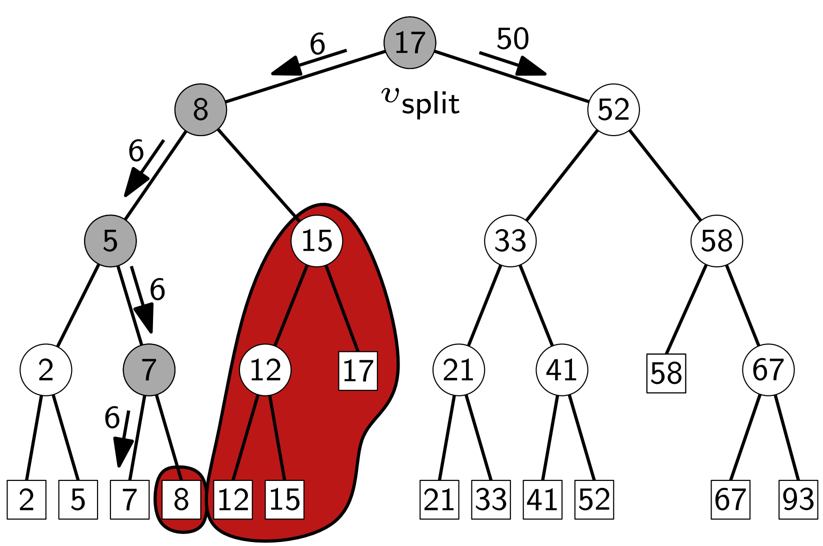
Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Solution: Balanced binary search tree:

- Stores points in the leaves
- Internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

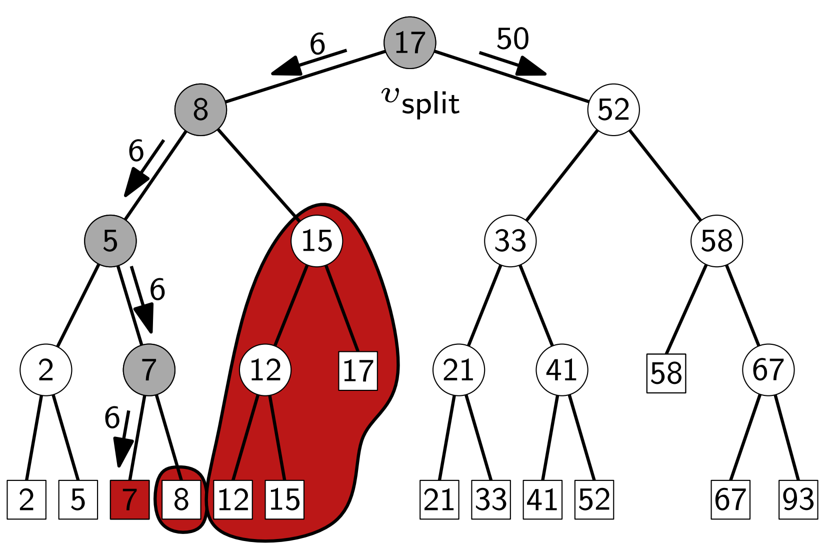
Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Solution: Balanced binary search tree:

- Stores points in the leaves
- Internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

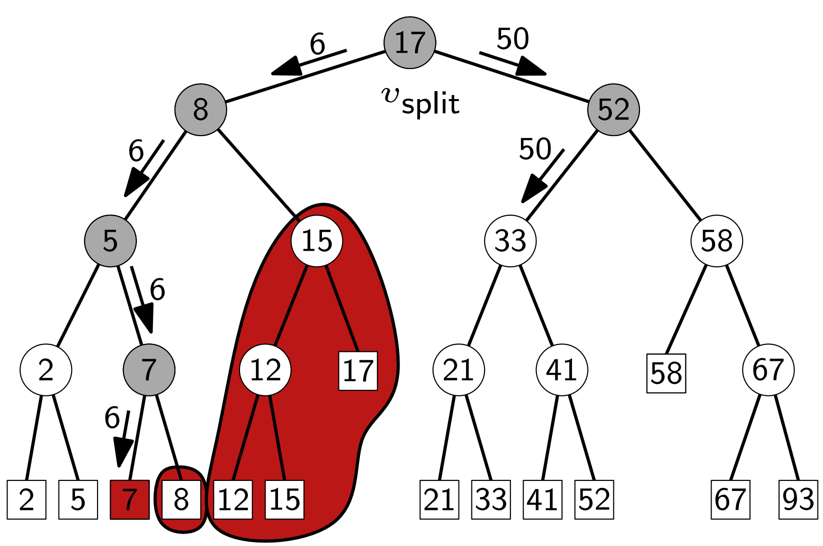
Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Solution: Balanced binary search tree:

- Stores points in the leaves
- Internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

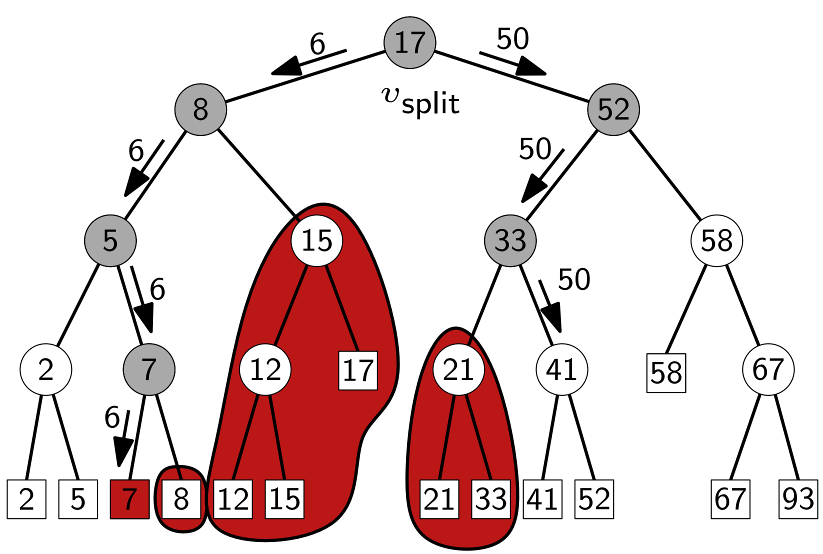
Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Solution: Balanced binary search tree:

- Stores points in the leaves
- Internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

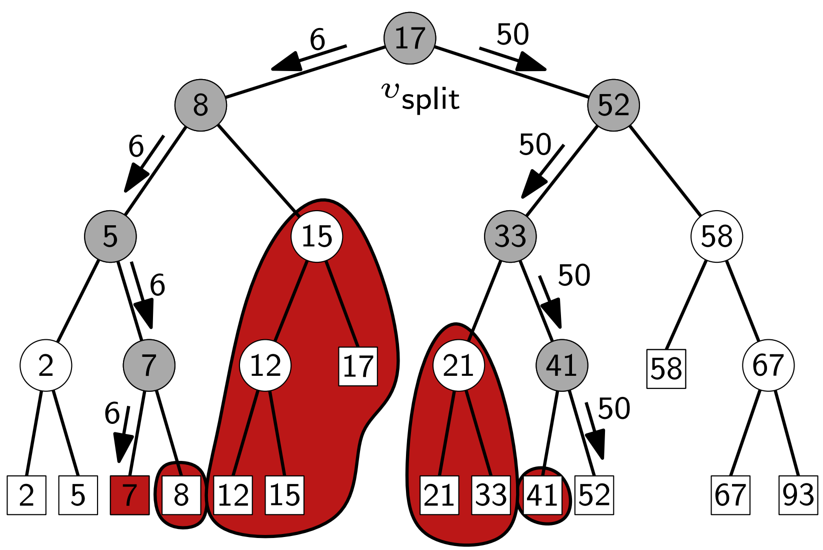
Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Solution: Balanced binary search tree:

- Stores points in the leaves
- Internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

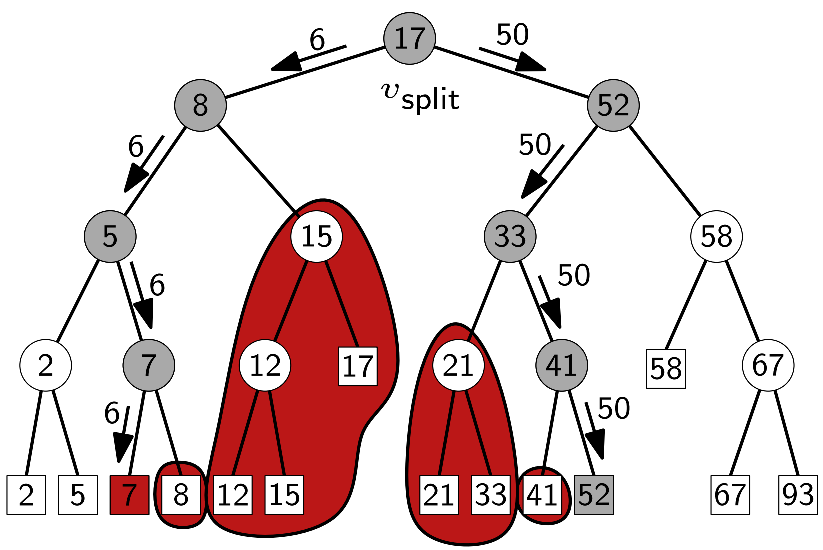
Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Solution: Balanced binary search tree:

- Stores points in the leaves
- Internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

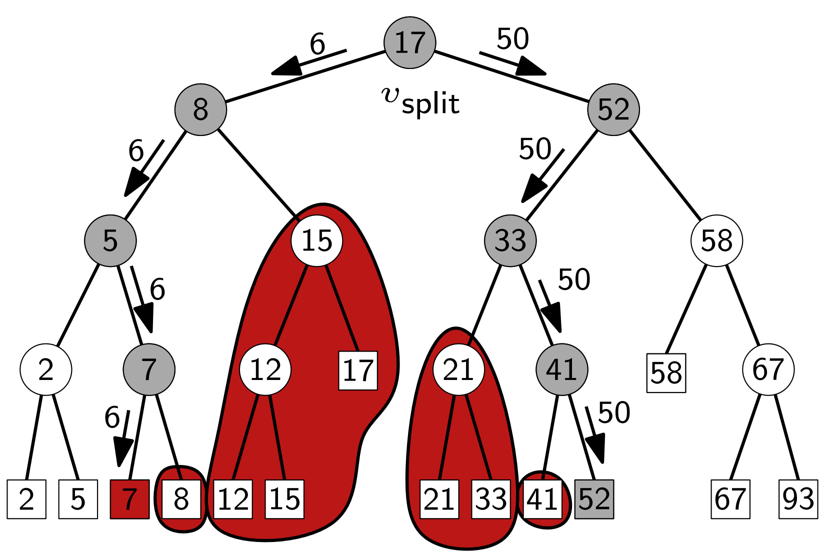
Given: n points in $\mathbb{R}^d$

Output: A data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Problem: Design a data structure for the case $d = 1$.

Solution: Balanced binary search tree:

- Stores points in the leaves
- Internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Answer:

Points in the leaves between the search paths, (i.e., {7,8,12,15,17,21,33,41})

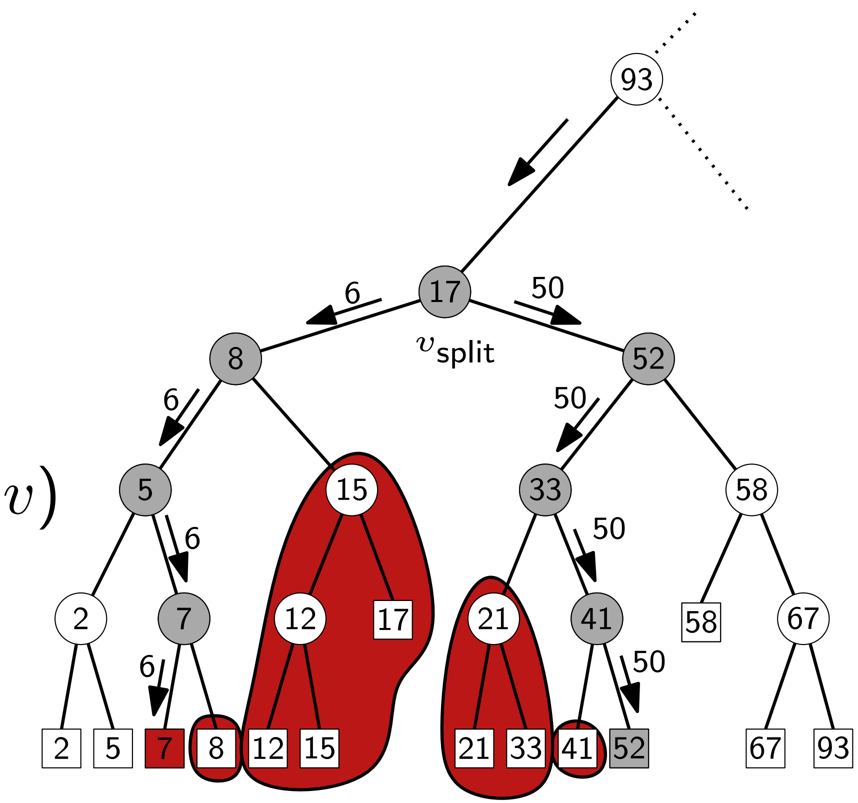
1dRangeQuery

FindSplitNode(T, x, x')

$v \leftarrow$ root(T)
while v not a leaf and $(x' \leq x_v$ or $x > x_v)$ **do**
  if $x' \leq x_v$ **then** $v \leftarrow$ lc(v) **else** $v \leftarrow$ rc(v)
return v

1dRangeQuery(T, x, x')

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** report v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not a leaf **do**
    if $x \leq x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  report v
  `// analog. for` x' `and rc(`v_{split}`)`



1dRangeQuery

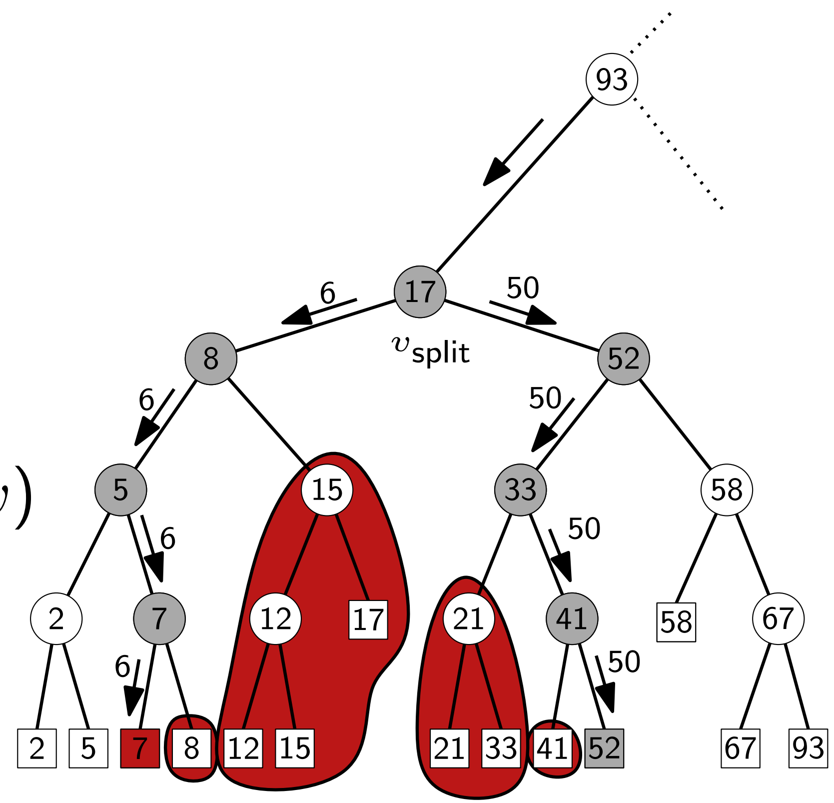
FindSplitNode(T, x, x')

$v \leftarrow$ root(T)
while v not a leaf and $(x' \leq x_v$ or $x > x_v)$ **do**
  if $x' \leq x_v$ **then** $v \leftarrow$ lc(v) **else** $v \leftarrow$ rc(v)
return v

1dRangeQuery(T, x, x')

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** report v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not a leaf **do**
    if $x \leq x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  report v
  `// analog. for` x' `and rc(`v_{split}`)`

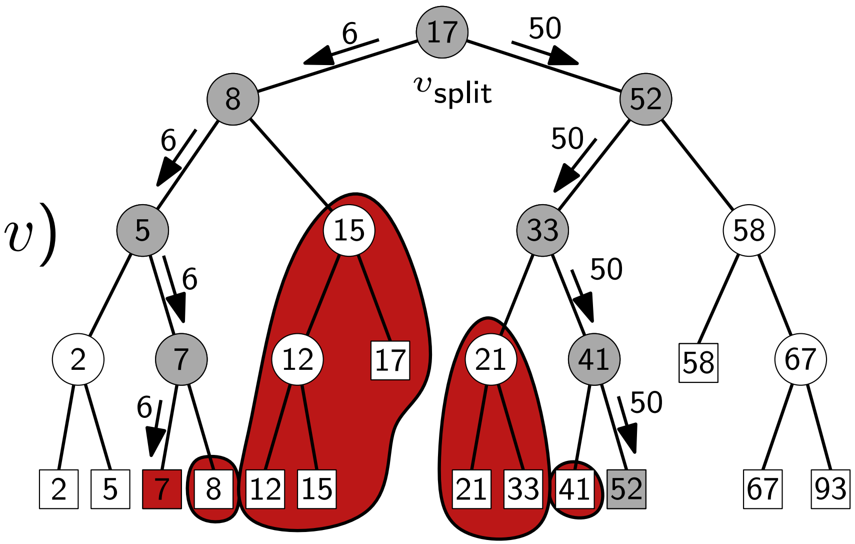
Can find *canonical subset* in linear time



Analysis of 1dRangeQuery

1dRangeQuery(T, x, x')

$v_{\mathsf{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** report v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not a leaf **do**
    if $x \leq x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  report v
  `// analog. for` x' `and rc(`v_{split}`)`



Theorem 1: A set of n points in $\mathbb{R}$ can preprocessed in $O(n \log n)$ time and stored in $O(n)$ space so that we can answer range queries in $O(k + \log n)$ time, where k is the number of reported points.

Orthogonal Range Queries for $d = 2$

Given: Set P of n points in $\mathbb{R}^2$

Goal: A data structure to efficiently answer range queries of the form $R = [x, x'] \times [y, y']$

Orthogonal Range Queries for $d = 2$

Given: Set P of n points in $\mathbb{R}^2$

Goal: A data structure to efficiently answer range queries of the form $R = [x, x'] \times [y, y']$

Ideas for generalizing the 1d case?

Orthogonal Range Queries for $d = 2$

Karlsruhe Institute of Technology

Given: Set P of n points in $\mathbb{R}^2$

Goal: A data structure to efficiently answer range queries of the form $R = [x, x'] \times [y, y']$

Ideas for generalizing the 1d case?

Solutions:

- *one* search tree, alternate search for x and y coordinates
  $\rightarrow$ kd-**Tree**
- *primary* search tree on x-coordinates,
  several *secondary* search trees on y-coordinates
  $\rightarrow$ **Range Tree**

Orthogonal Range Queries for $d = 2$

Karlsruhe Institute of Technology

Given: Set P of n points in $\mathbb{R}^2$

Goal: A data structure to efficiently answer range queries of the form $R = [x, x'] \times [y, y']$

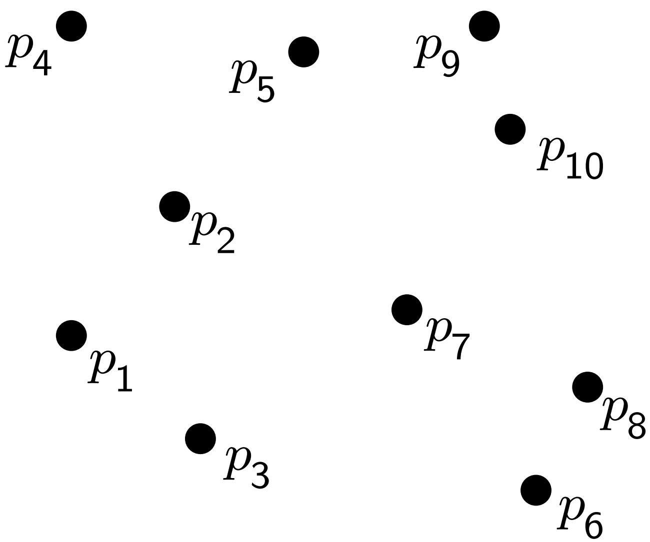
Ideas for generalizing the 1d case?

Solutions:

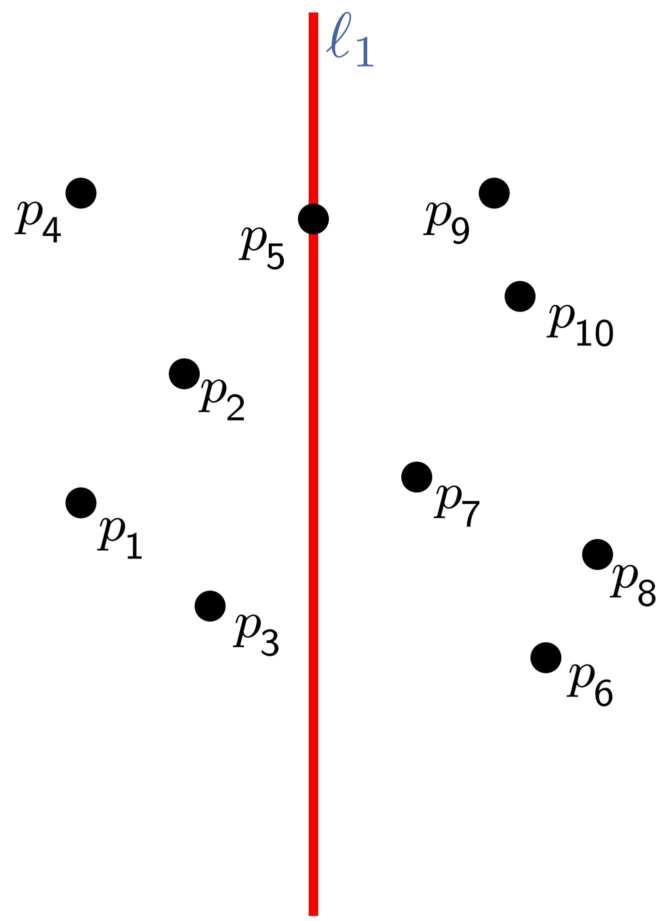
- *one* search tree, alternate search for x and y coordinates
  $\rightarrow$ kd-**Tree**
- *primary* search tree on x-coordinates,
  several *secondary* search trees on y-coordinates
  $\rightarrow$ **Range Tree**

Temporary assumption: general position, that is no two points have the same x- or y-coordinates

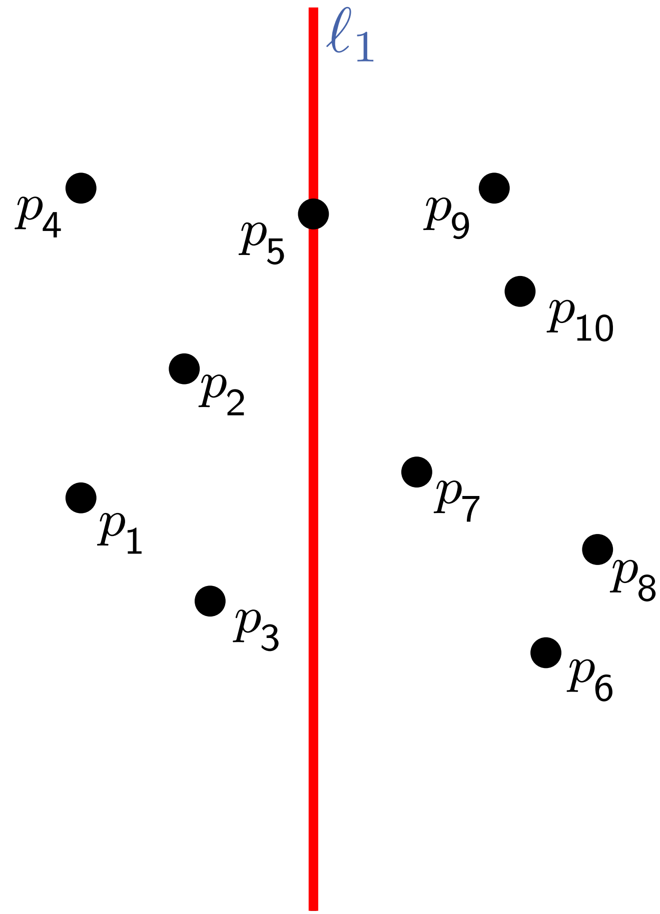
kd-Trees: Example

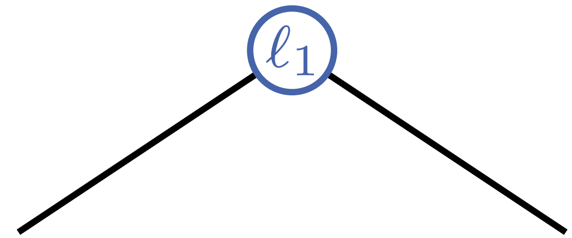


kd-Trees: Example

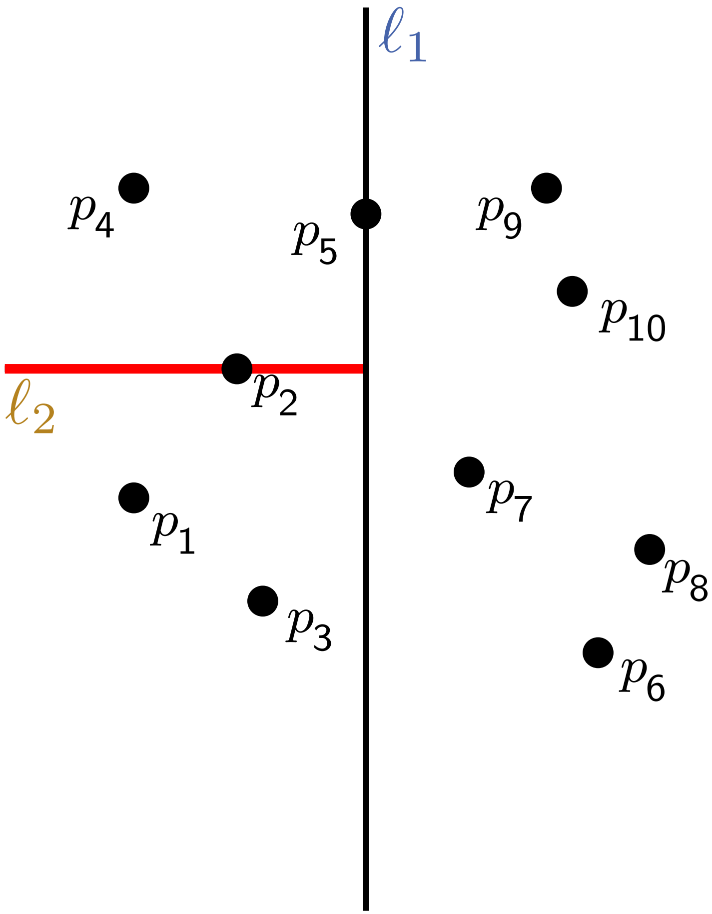


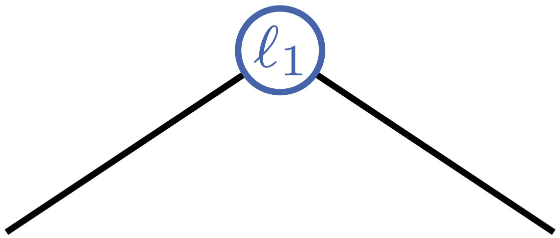
kd-Trees: Example



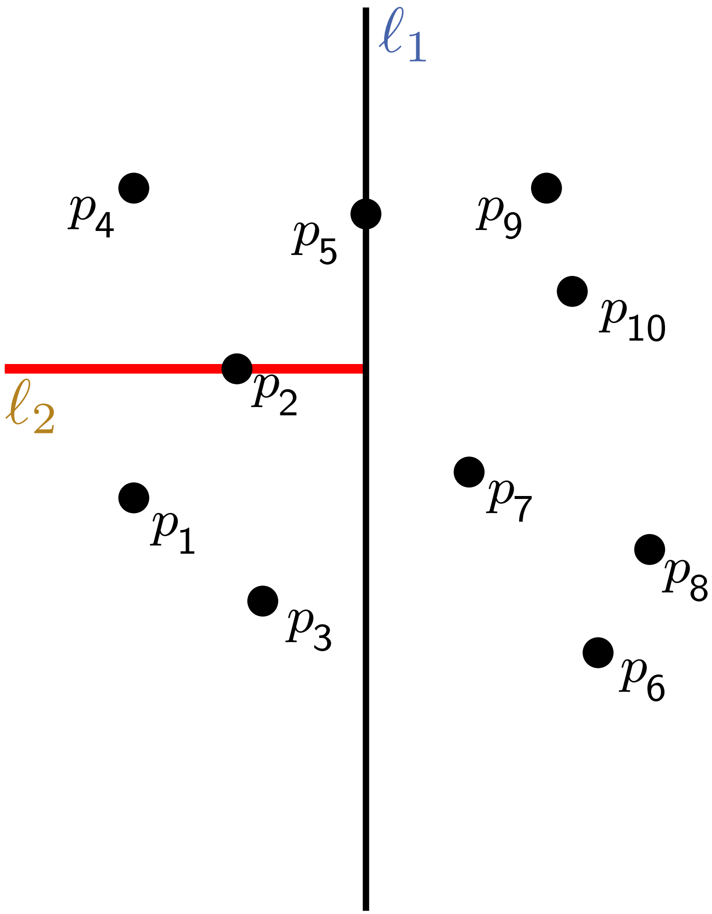


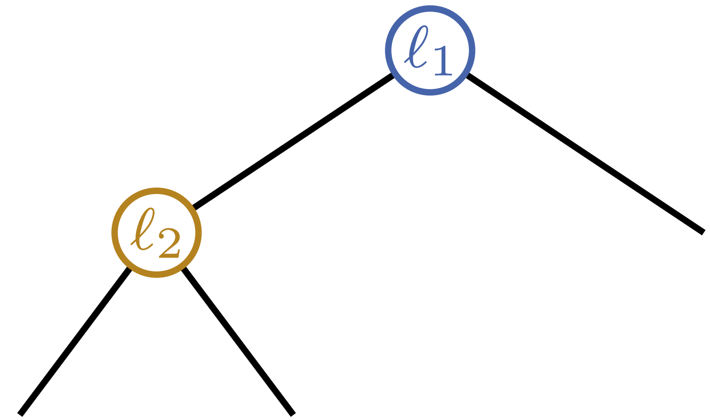
kd-Trees: Example



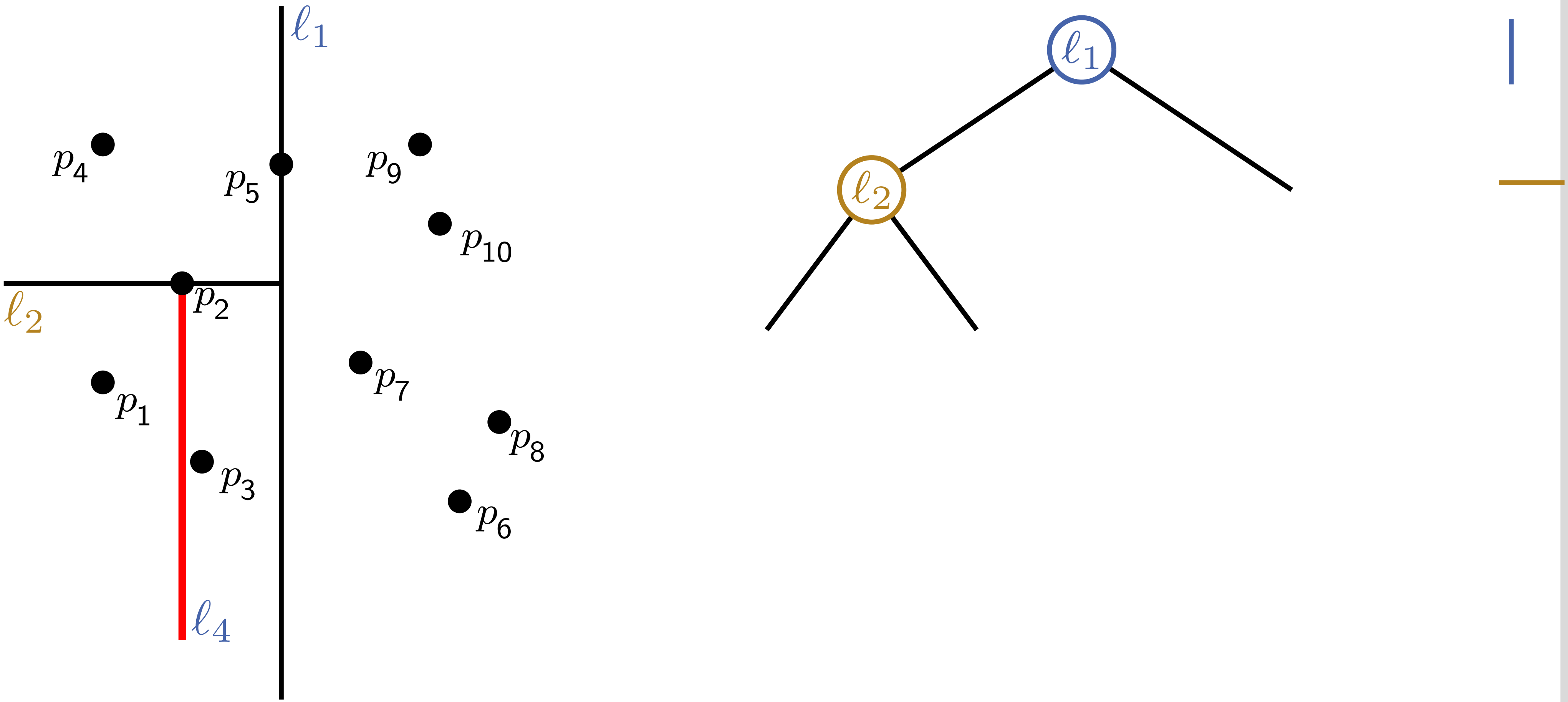


kd-Trees: Example

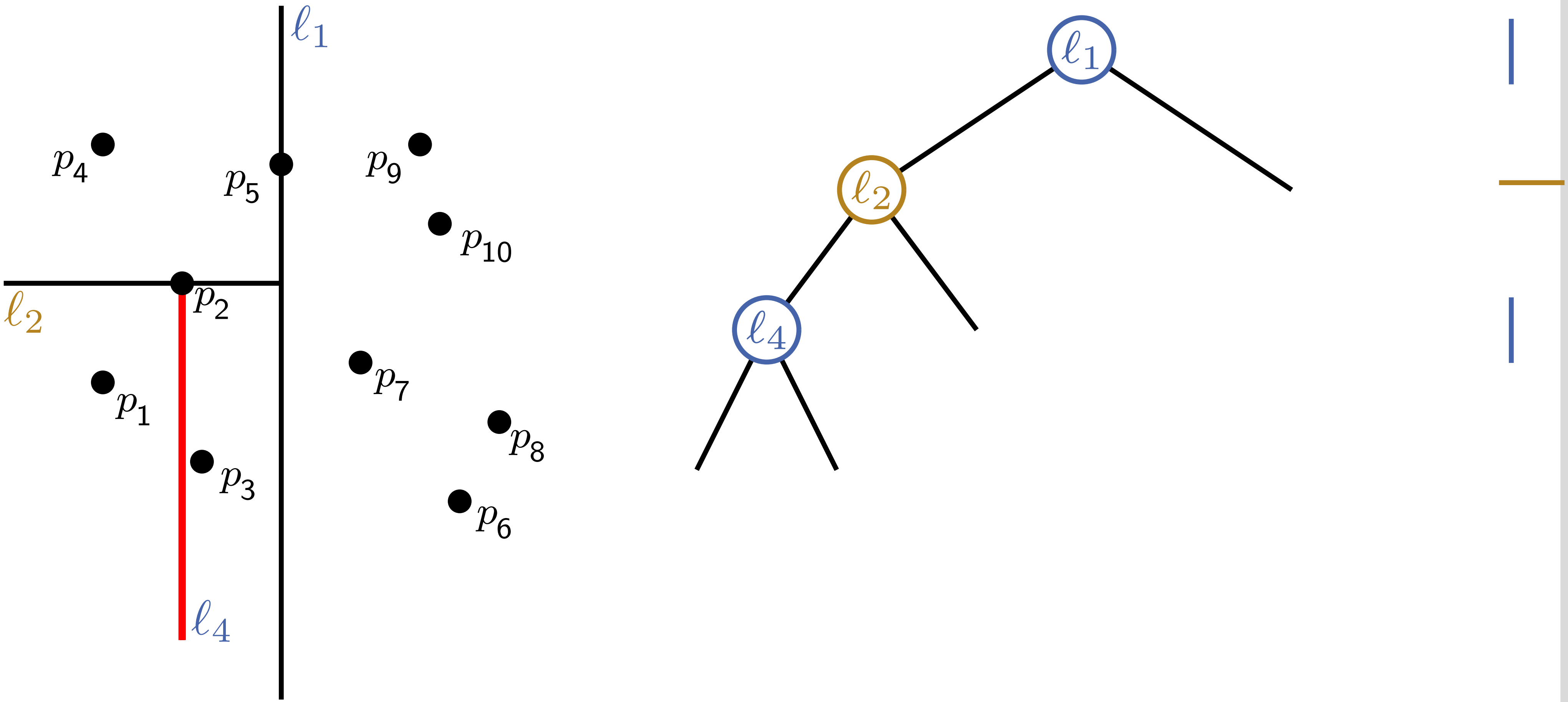




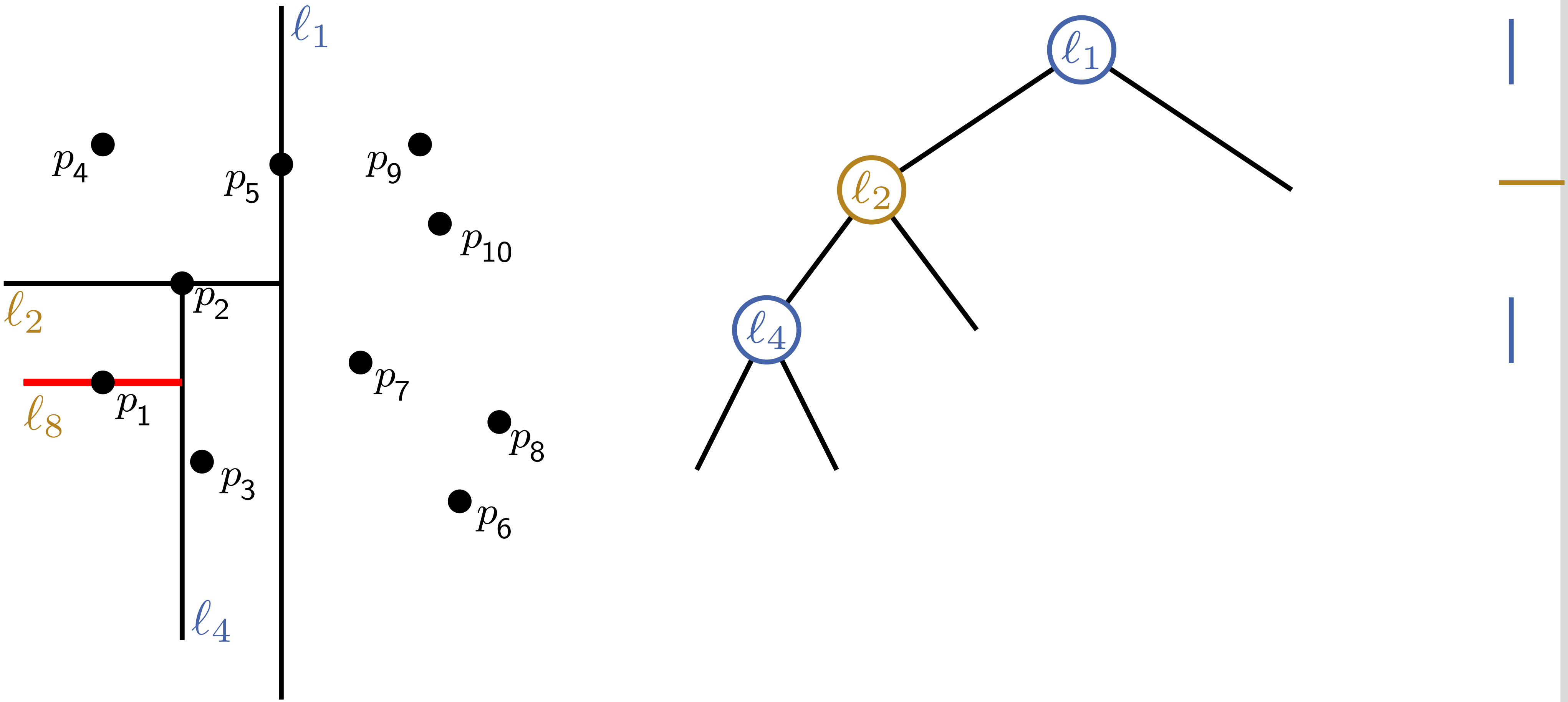
kd-Trees: Example



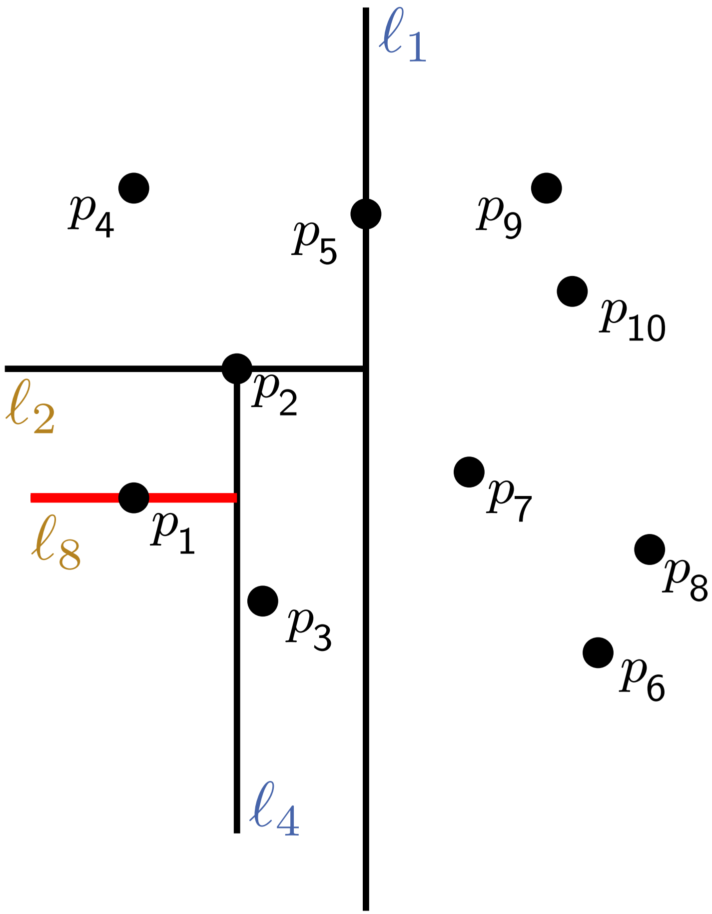
kd-Trees: Example

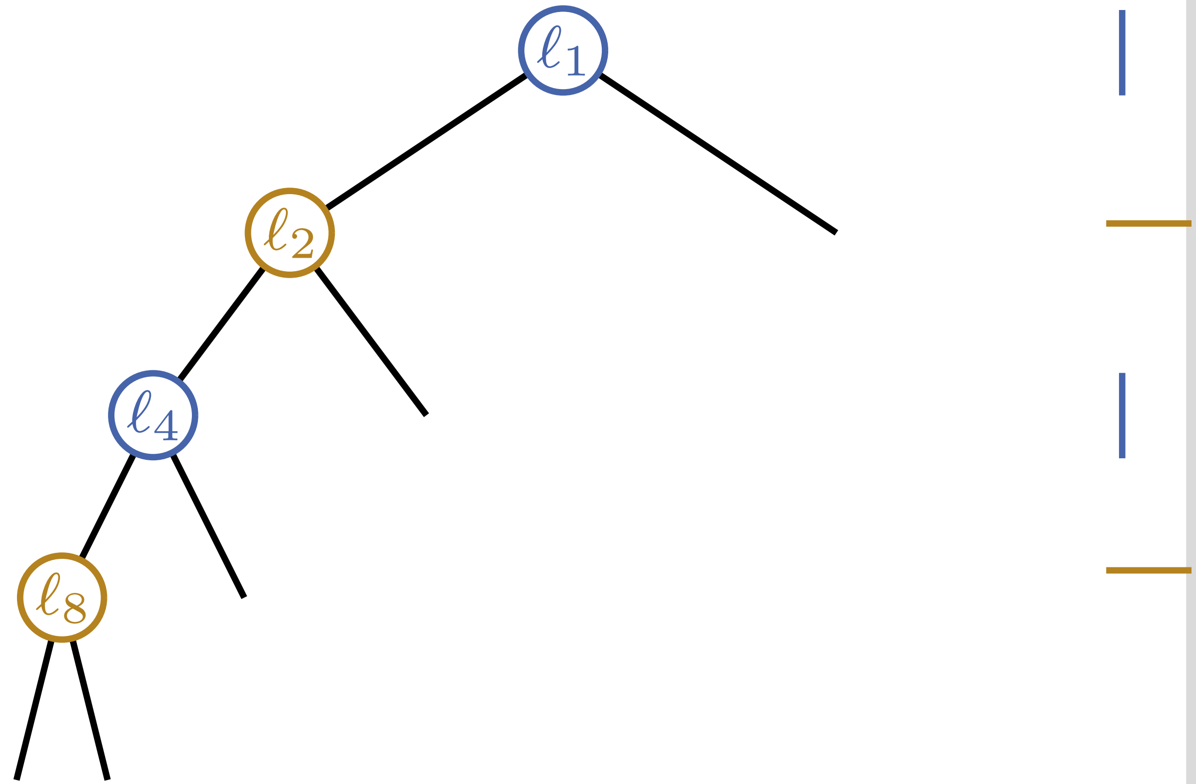


kd-Trees: Example

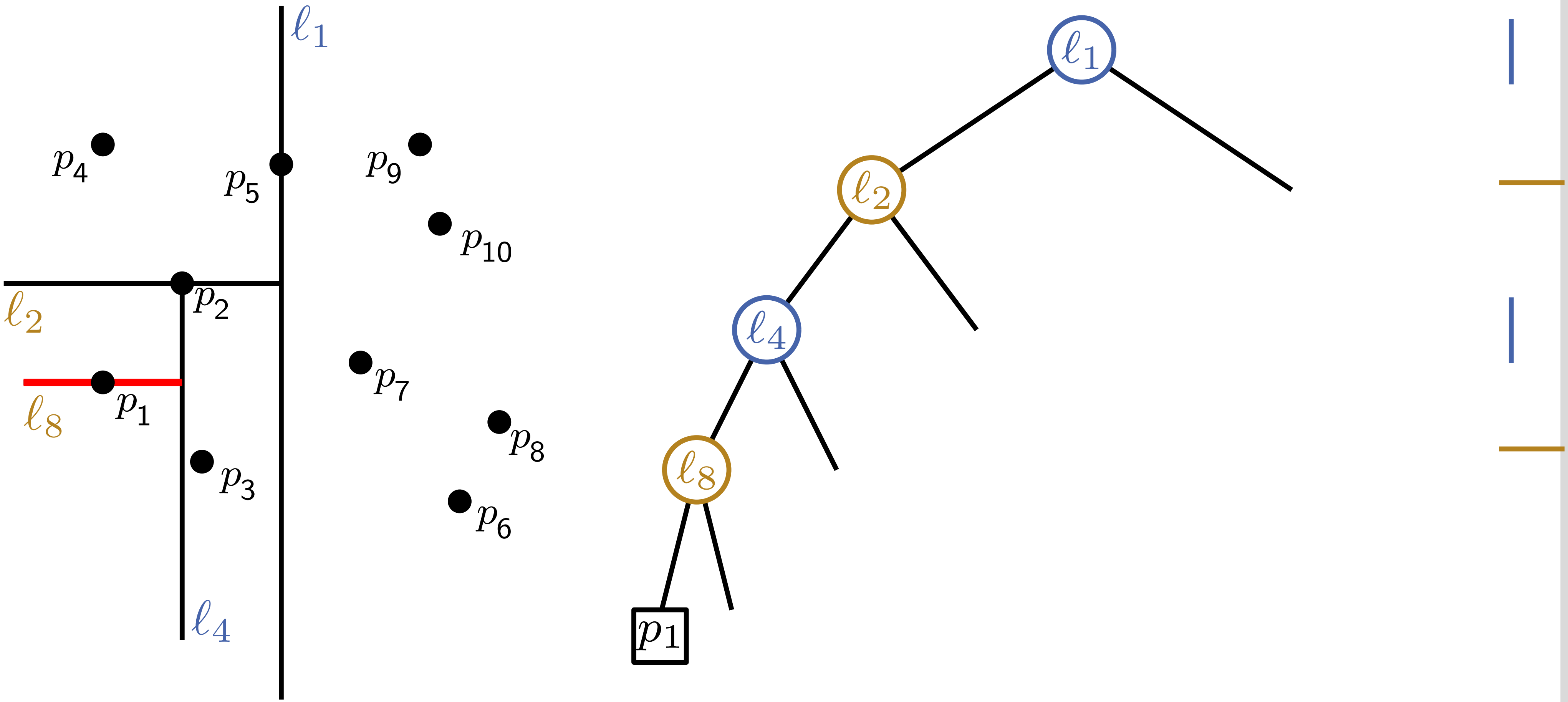


kd-Trees: Example

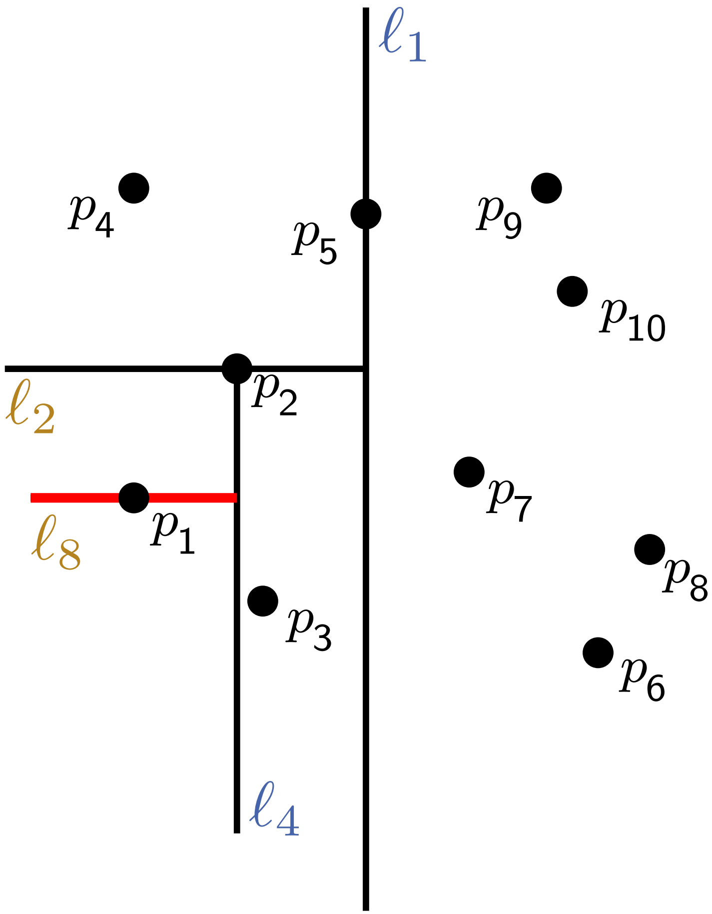


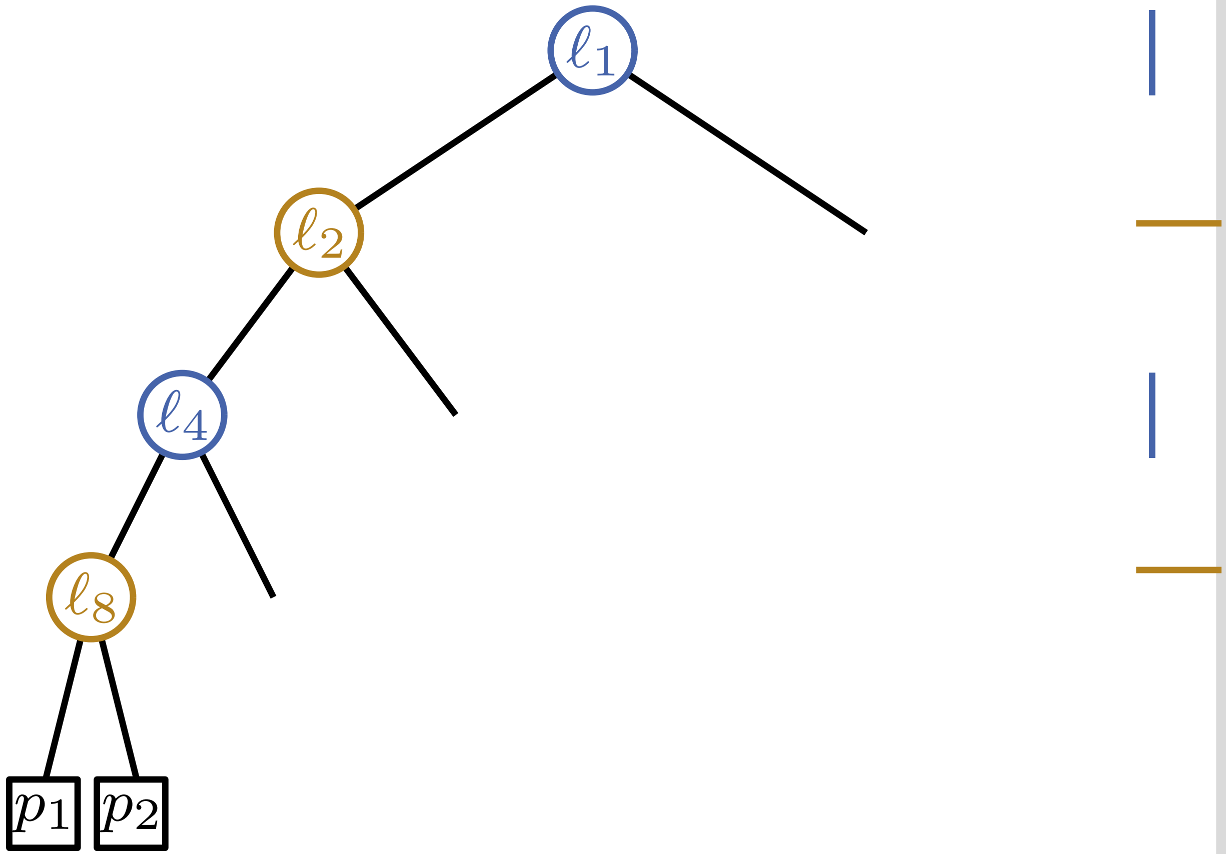


kd-Trees: Example

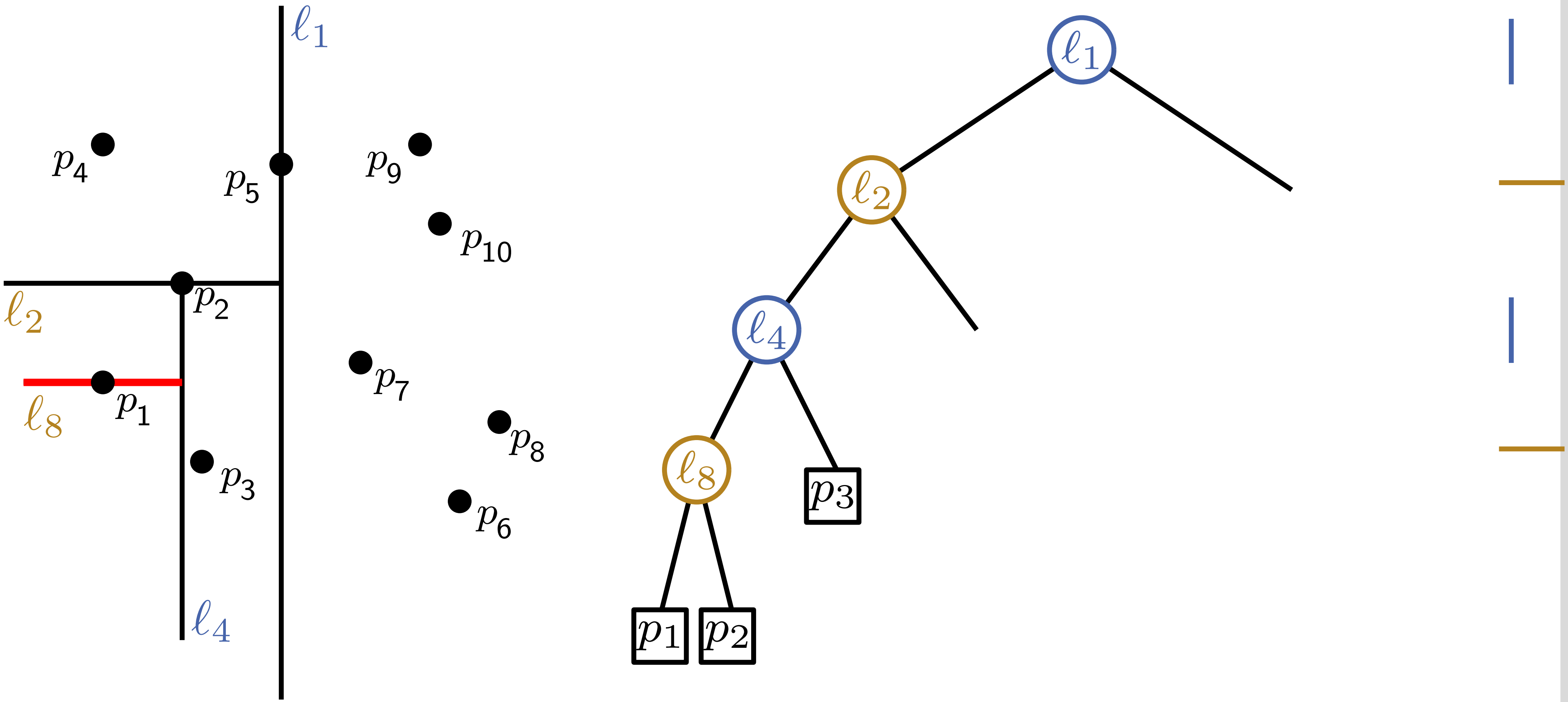


kd-Trees: Example

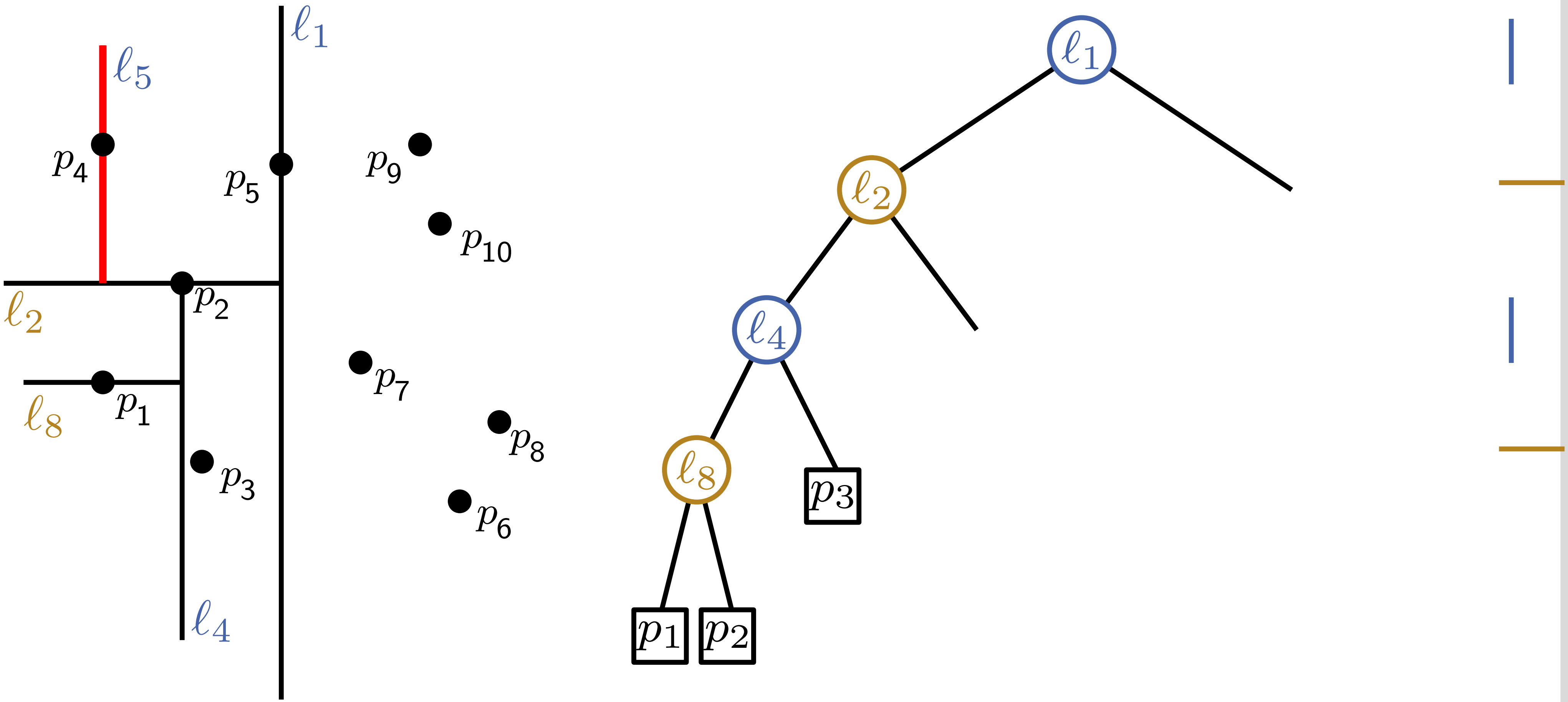




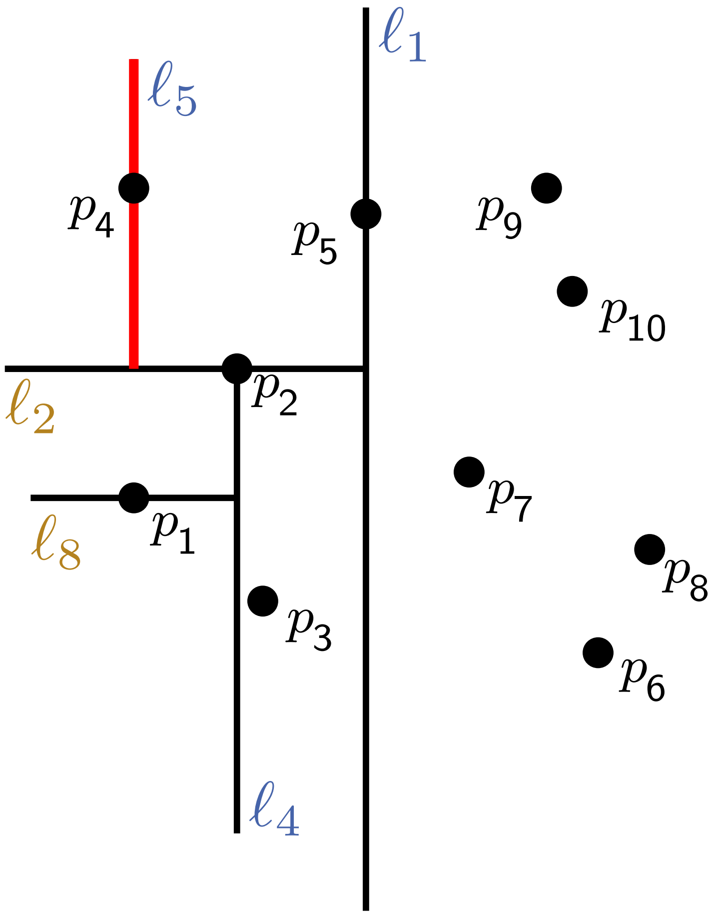
kd-Trees: Example

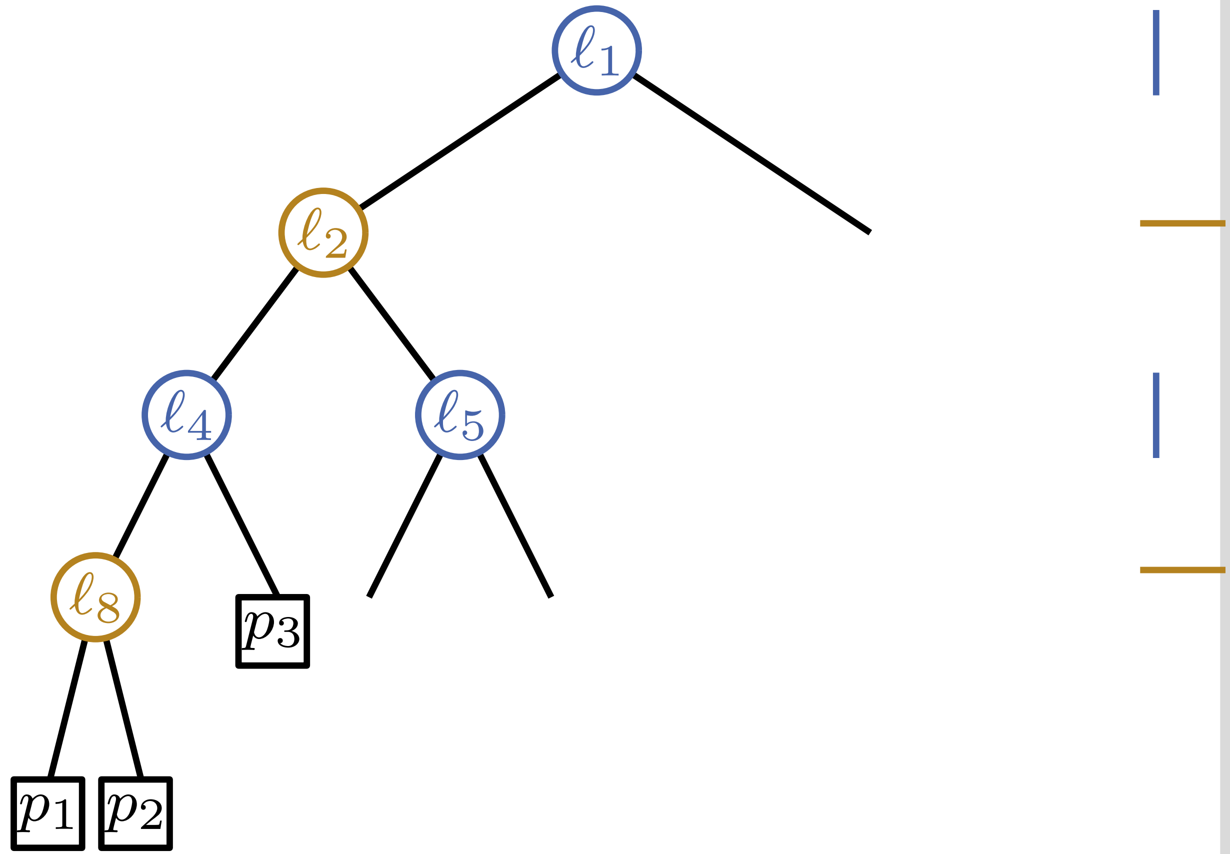


kd-Trees: Example

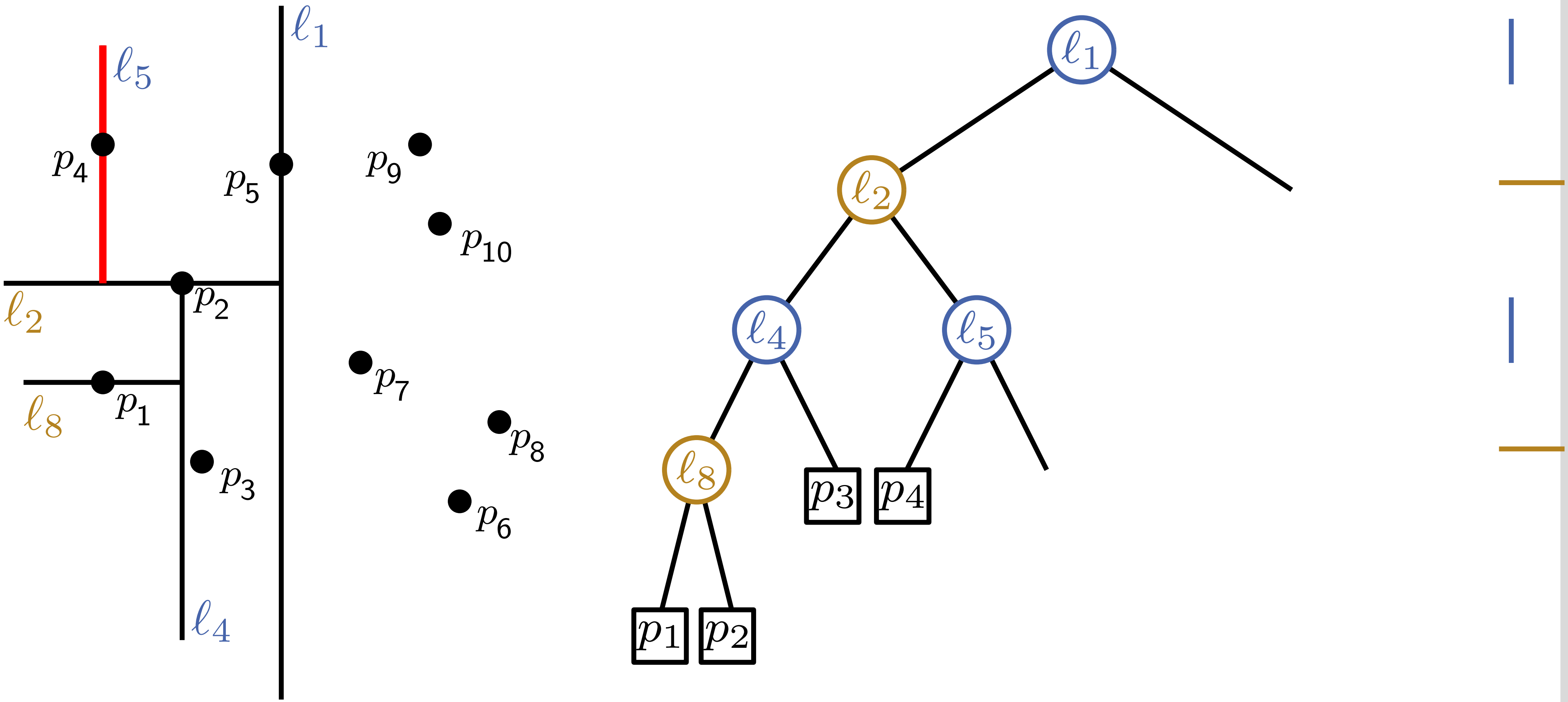


kd-Trees: Example

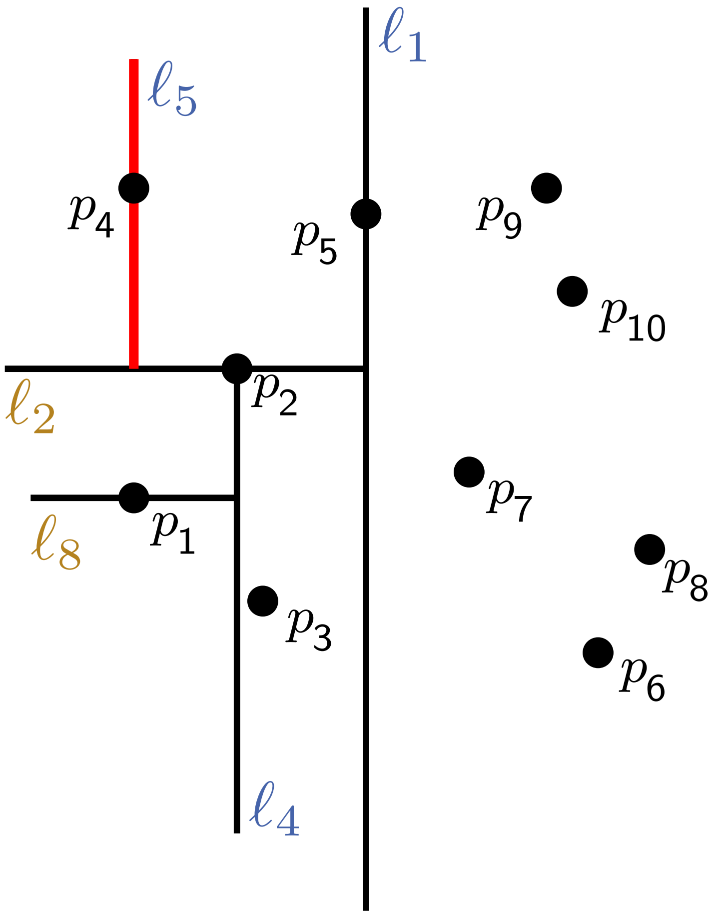


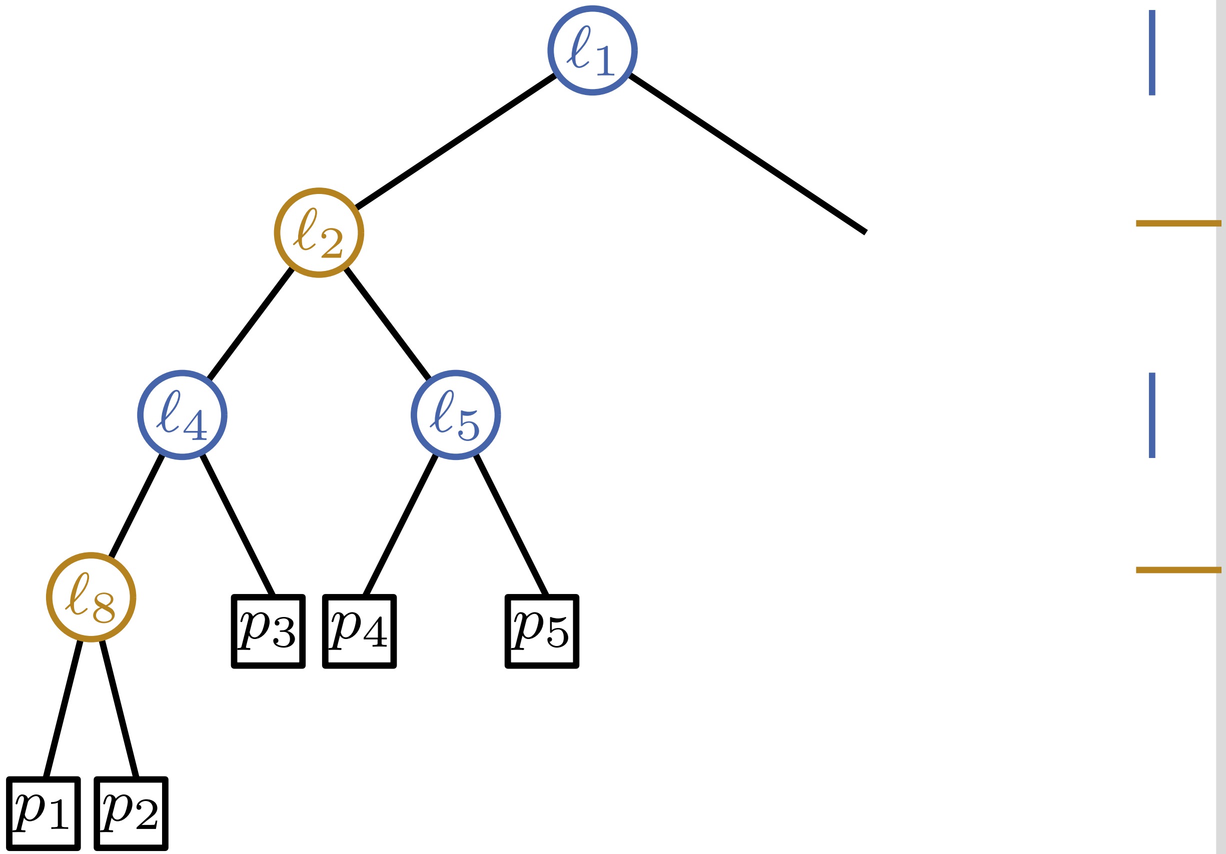


kd-Trees: Example

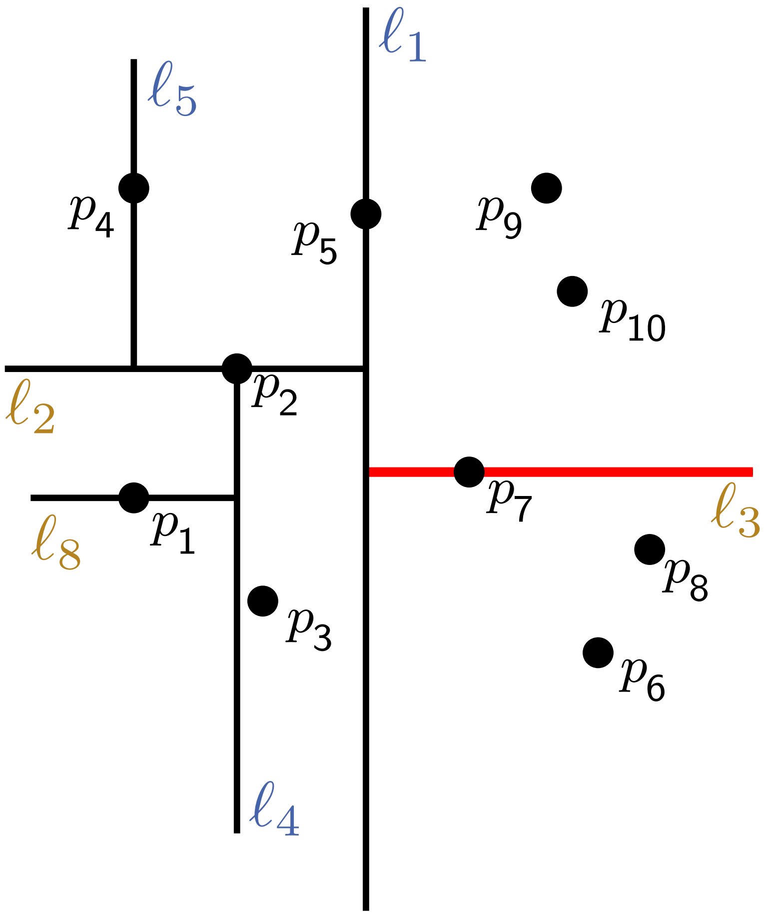


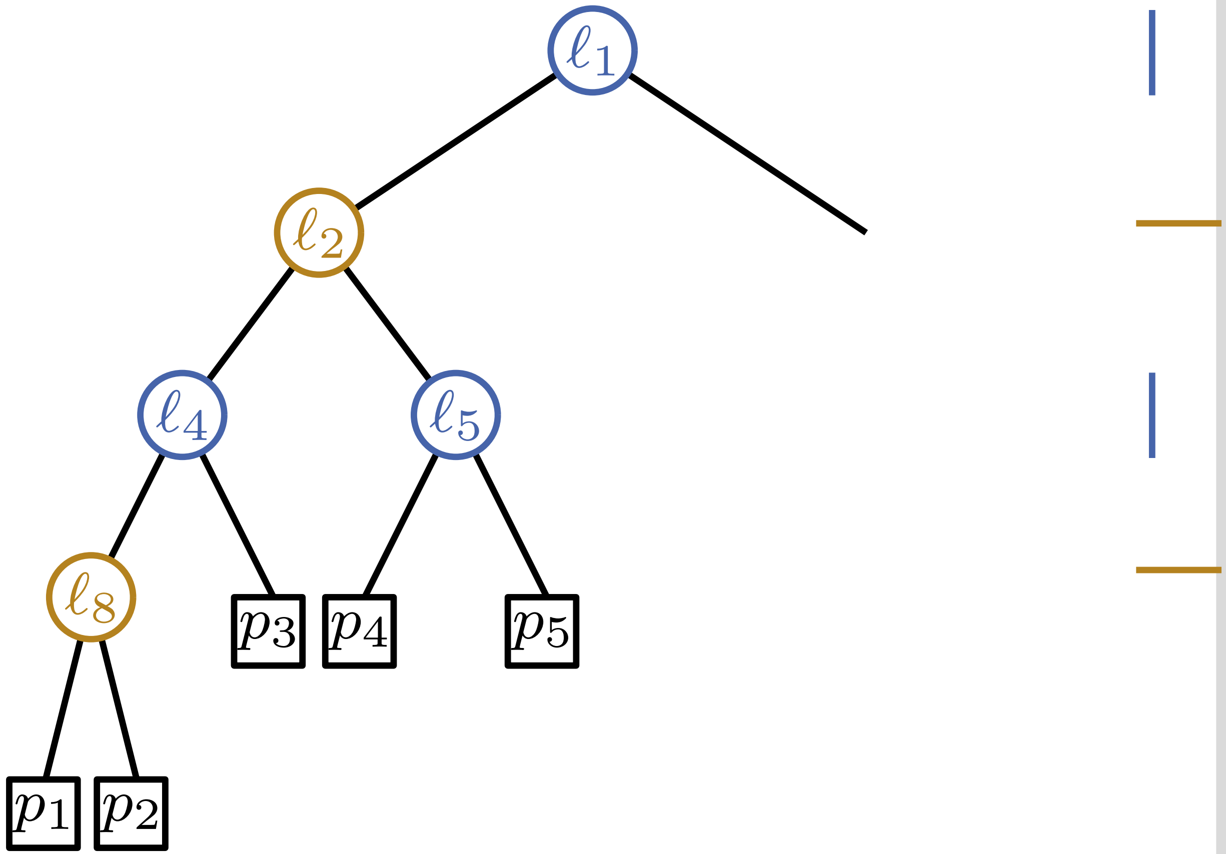
kd-Trees: Example



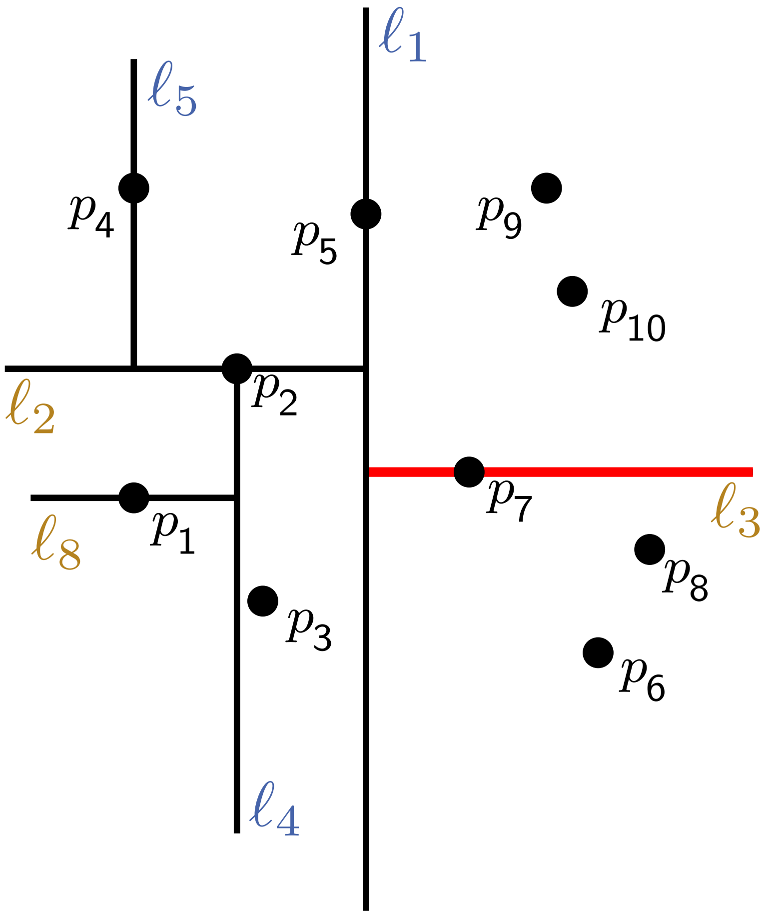


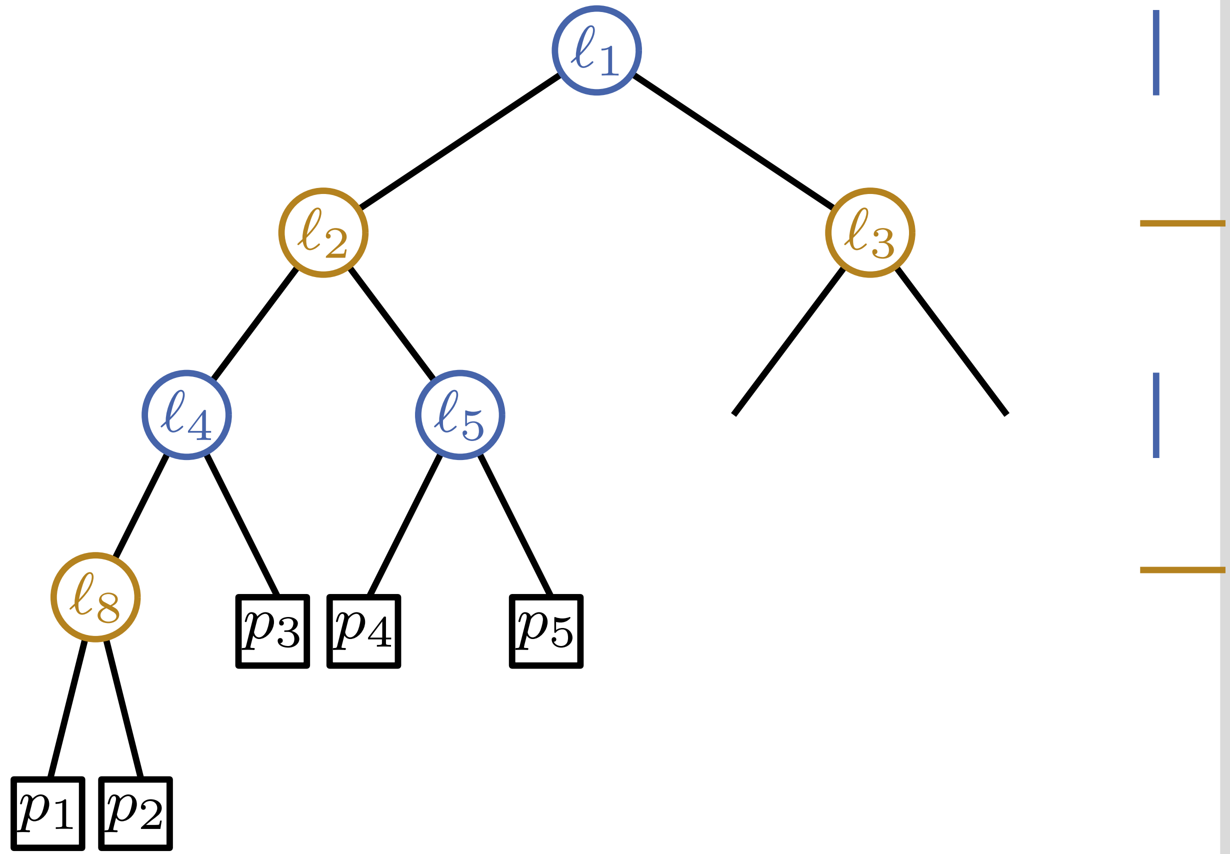
kd-Trees: Example



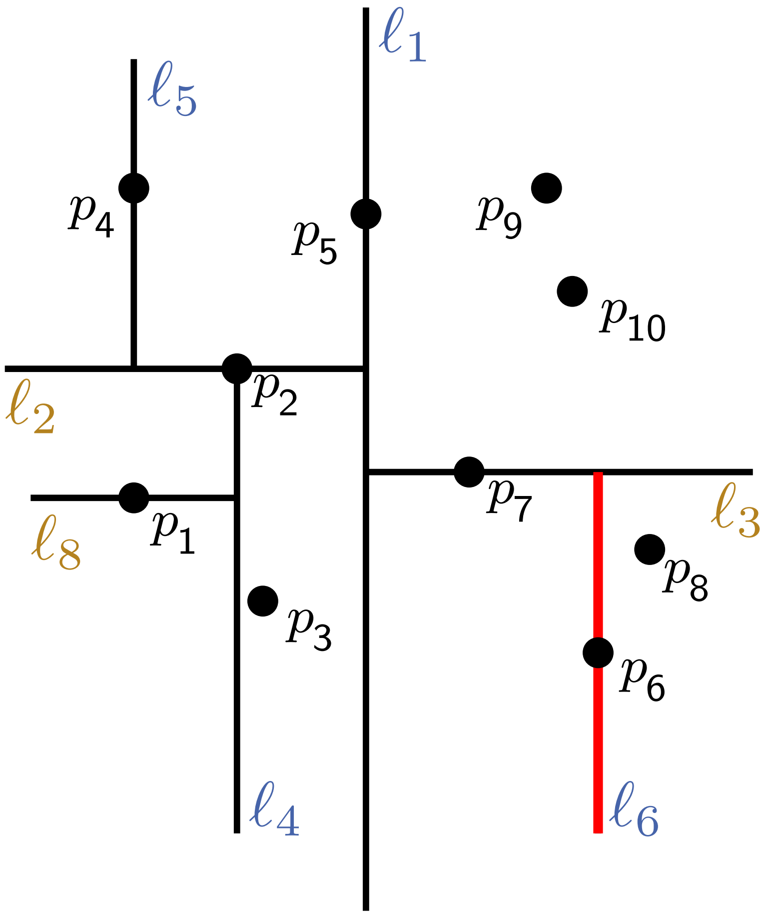


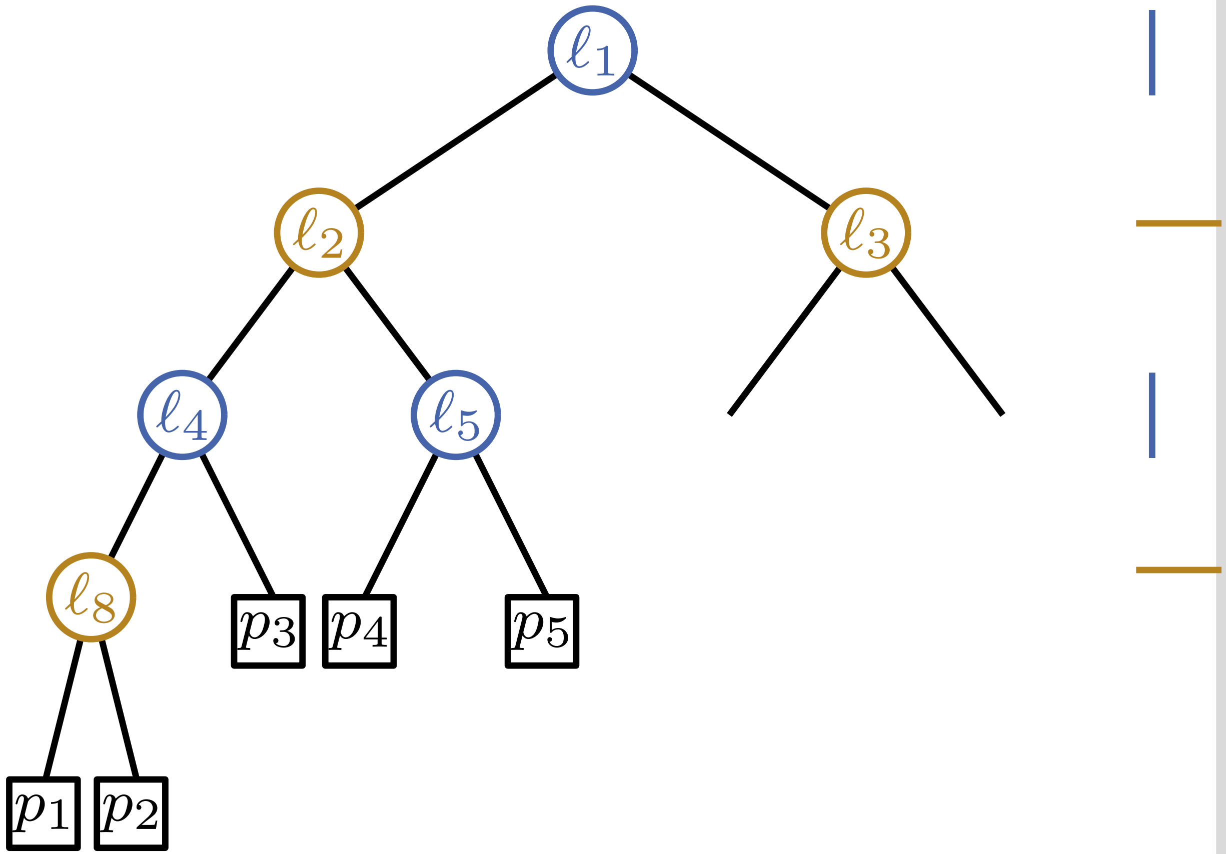
kd-Trees: Example



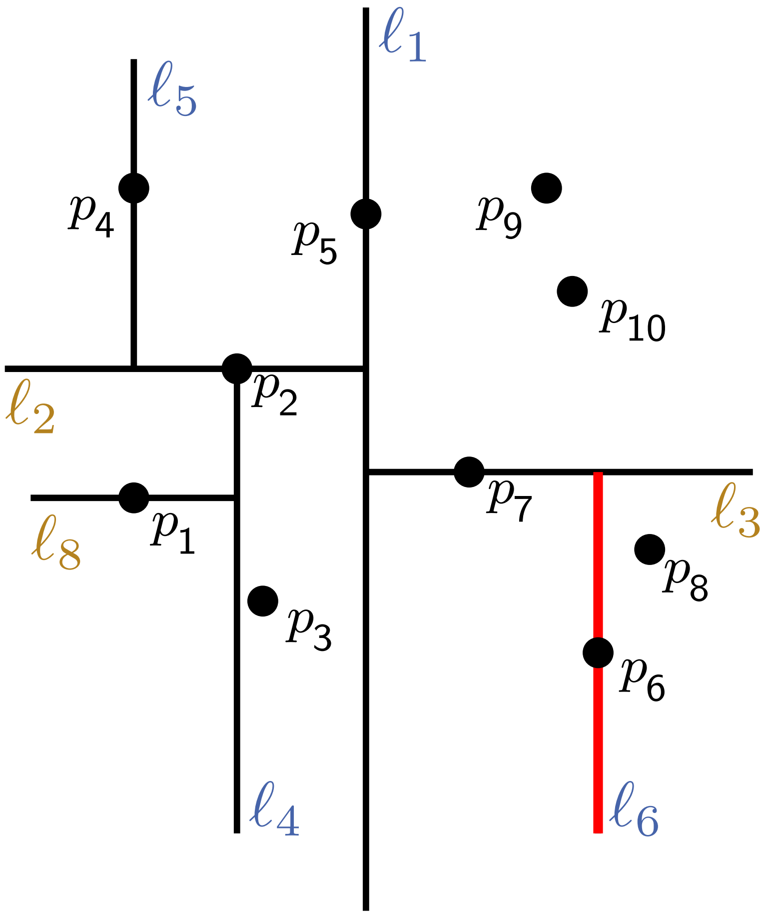


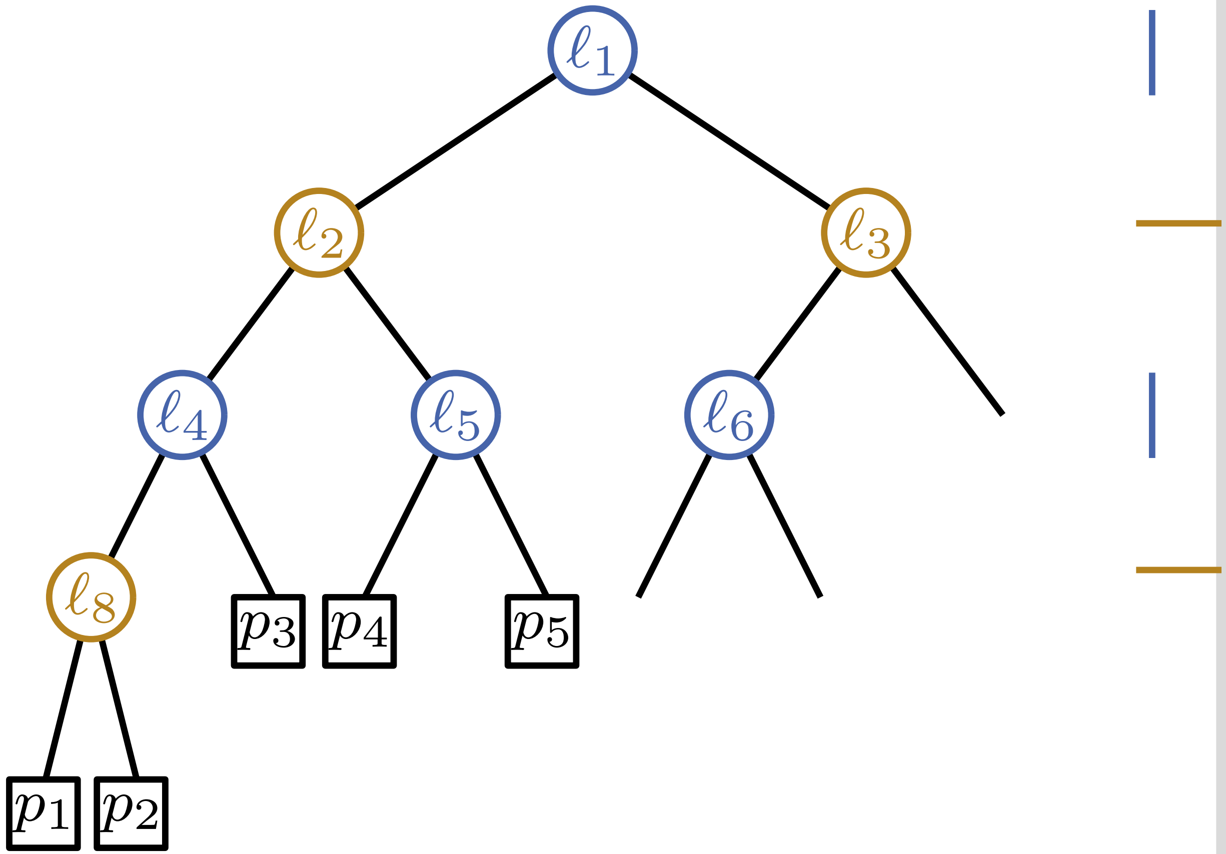
kd-Trees: Example



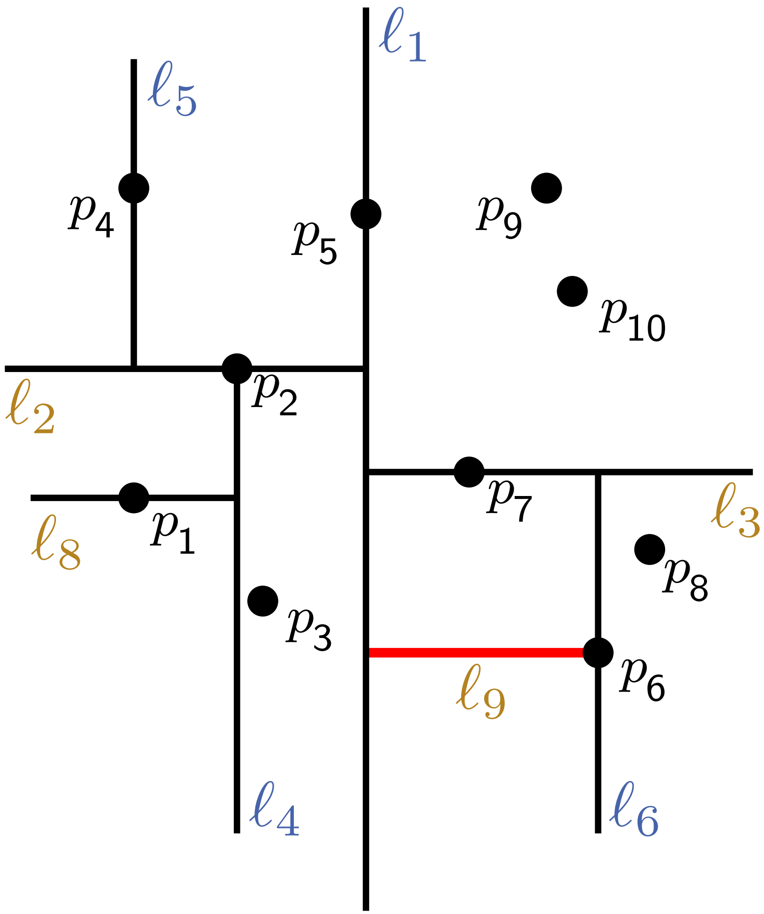


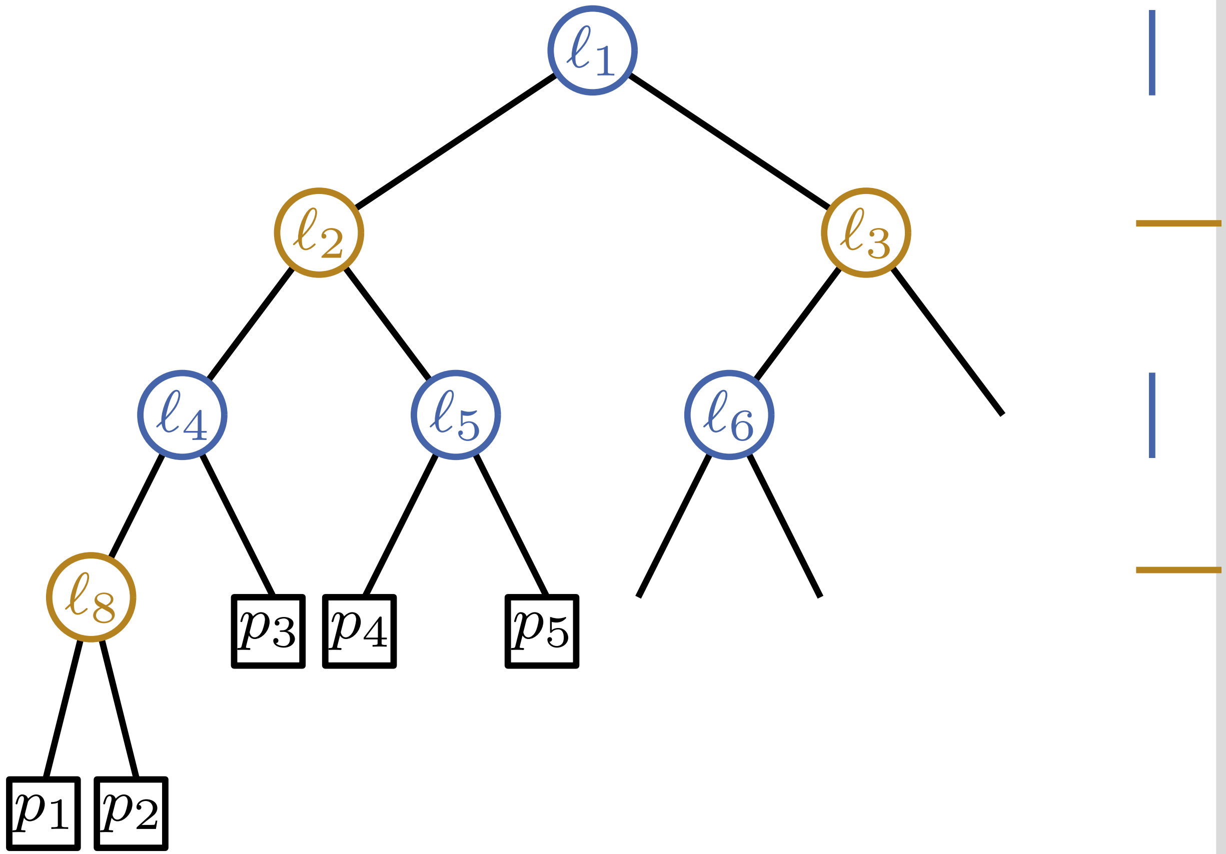
kd-Trees: Example



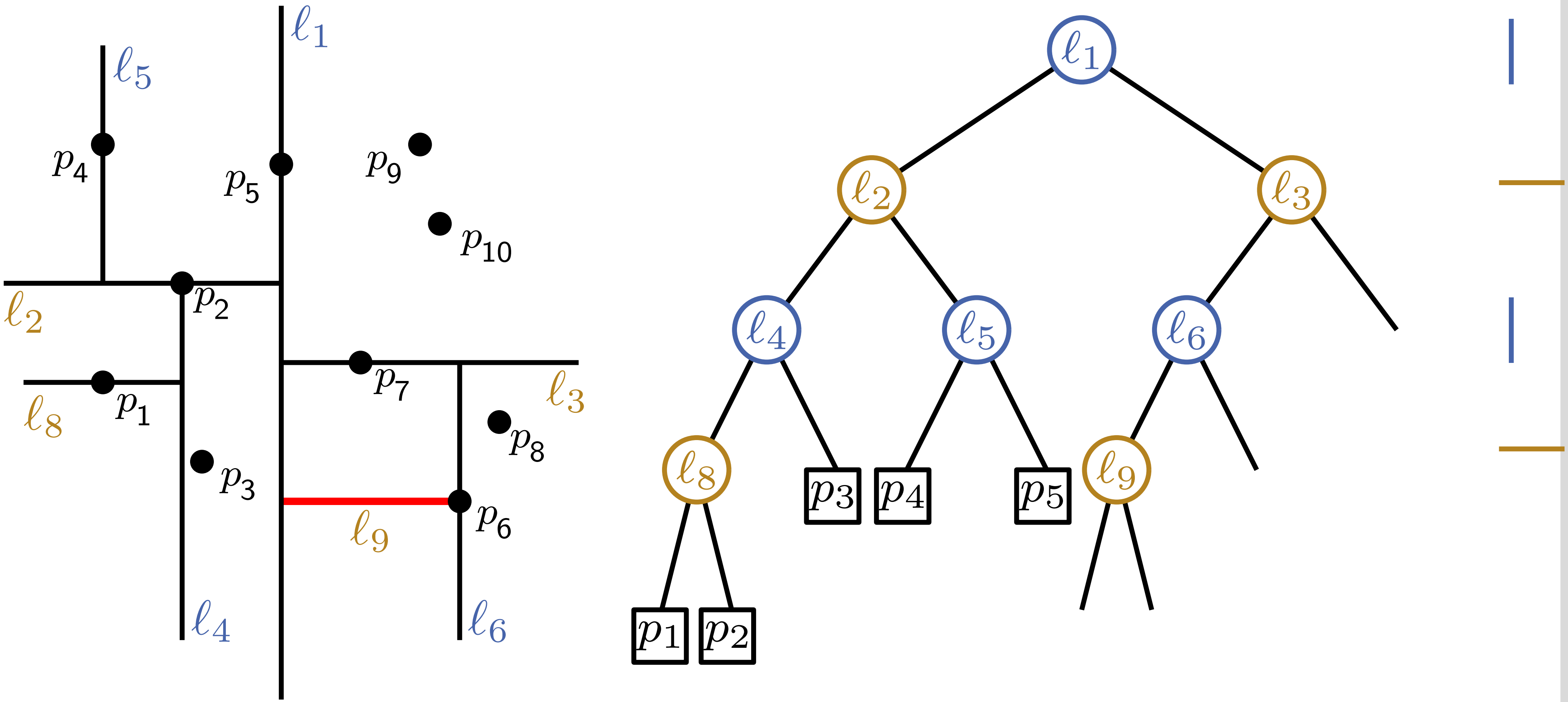


kd-Trees: Example

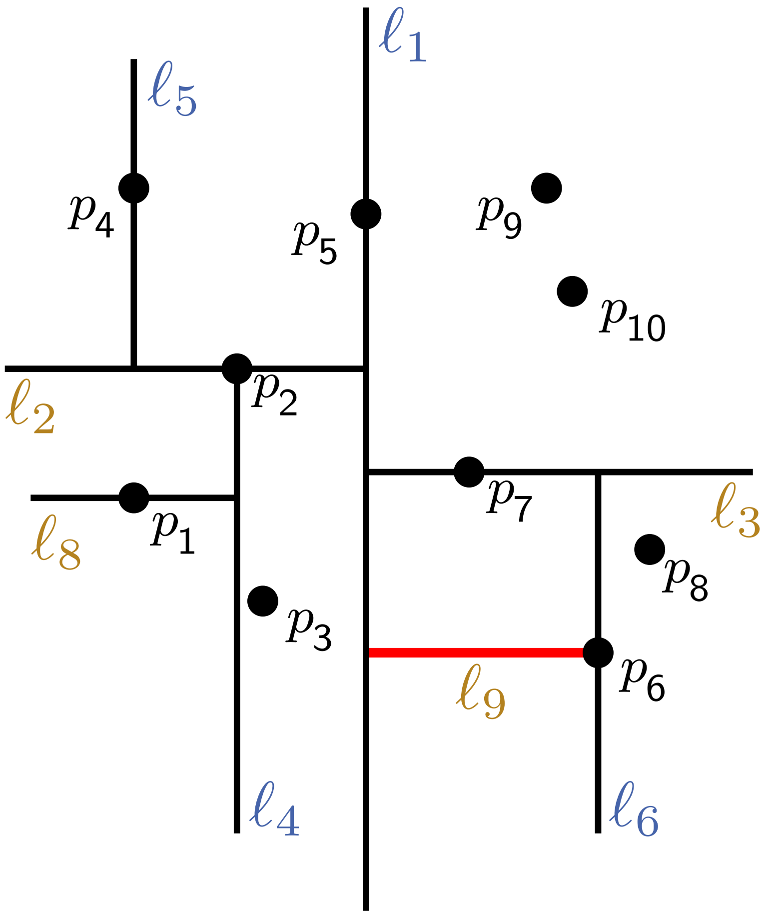


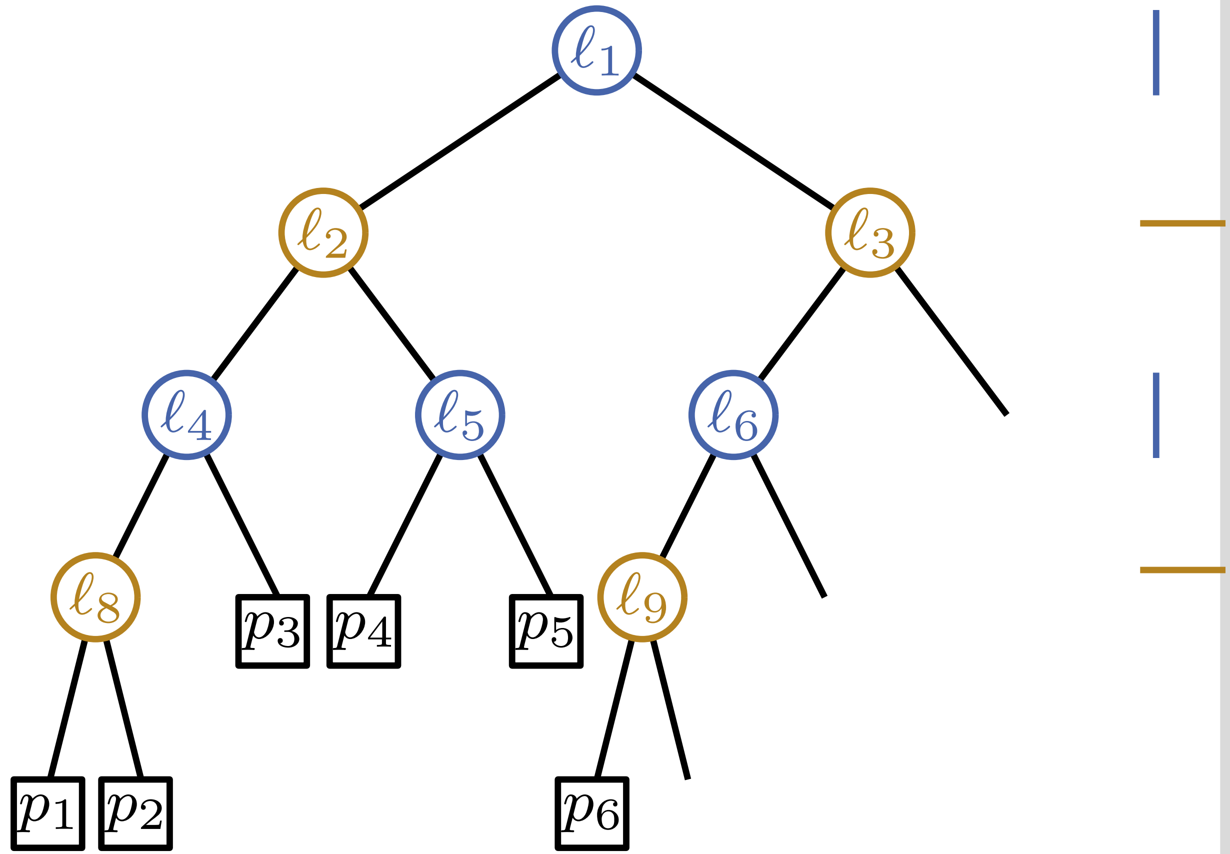


kd-Trees: Example

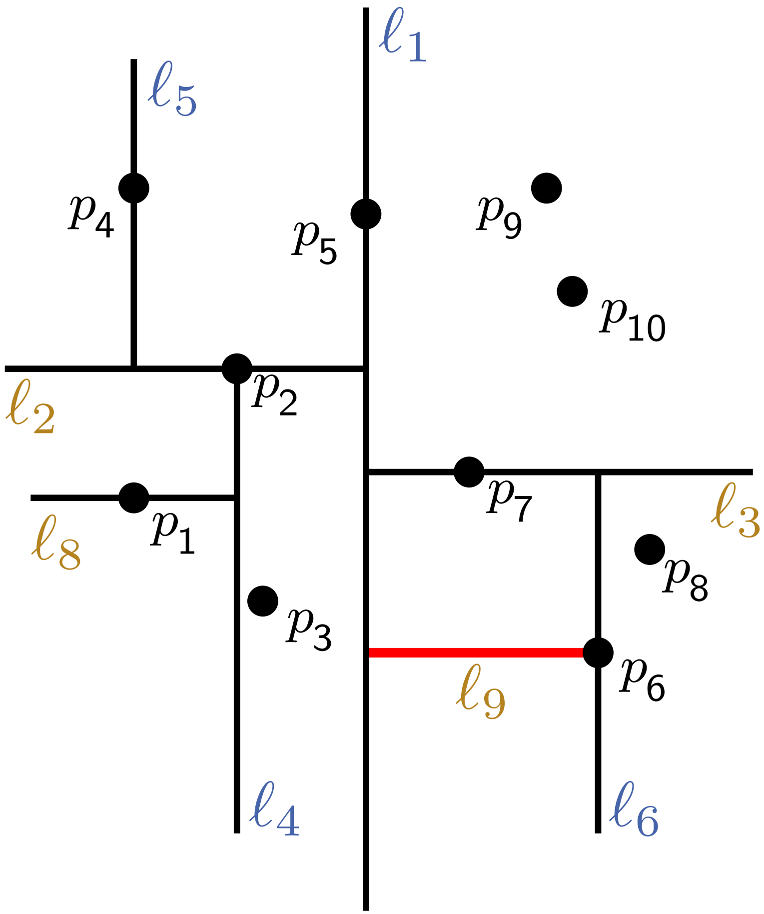


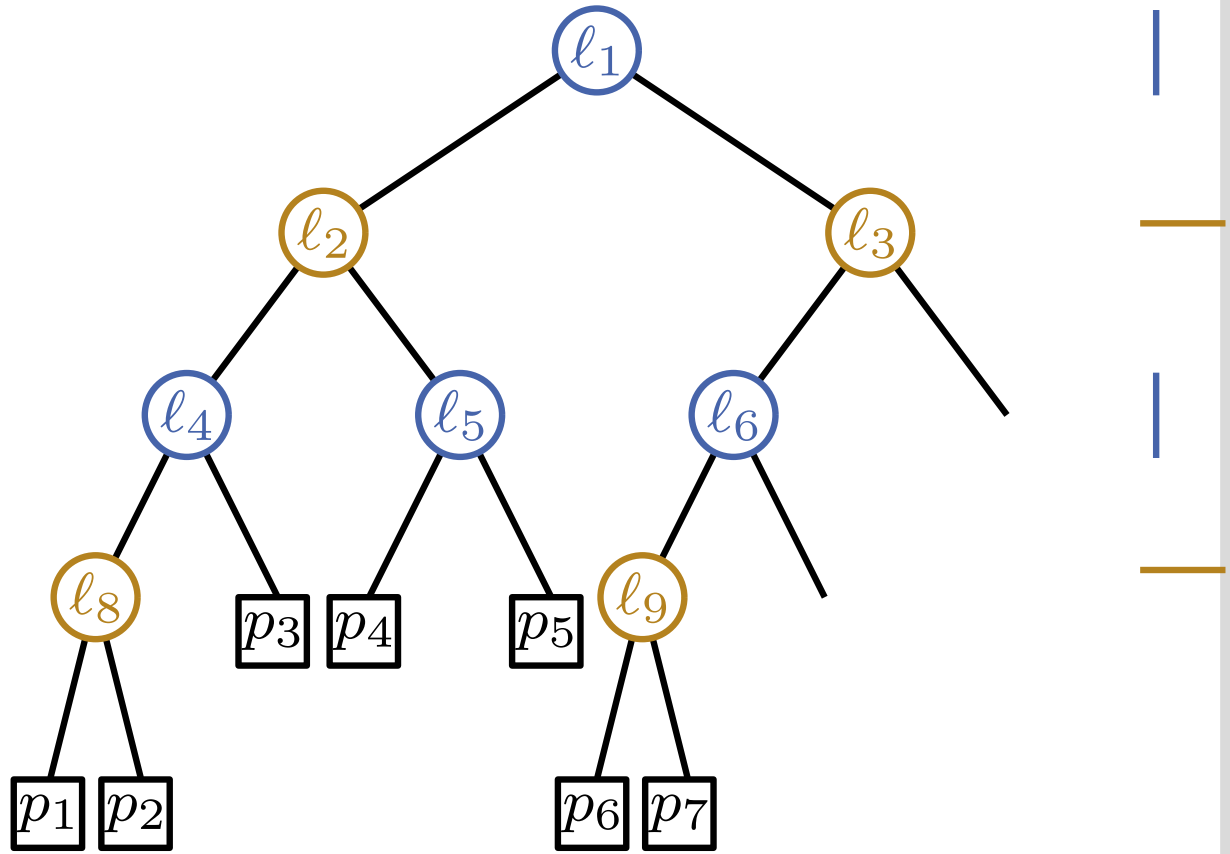
kd-Trees: Example



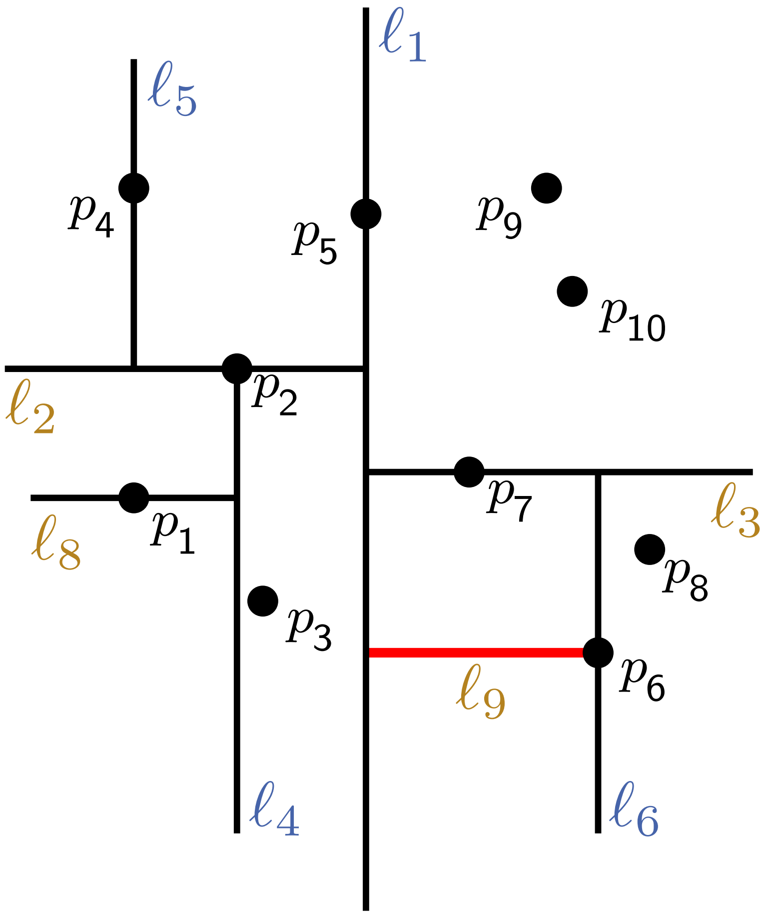


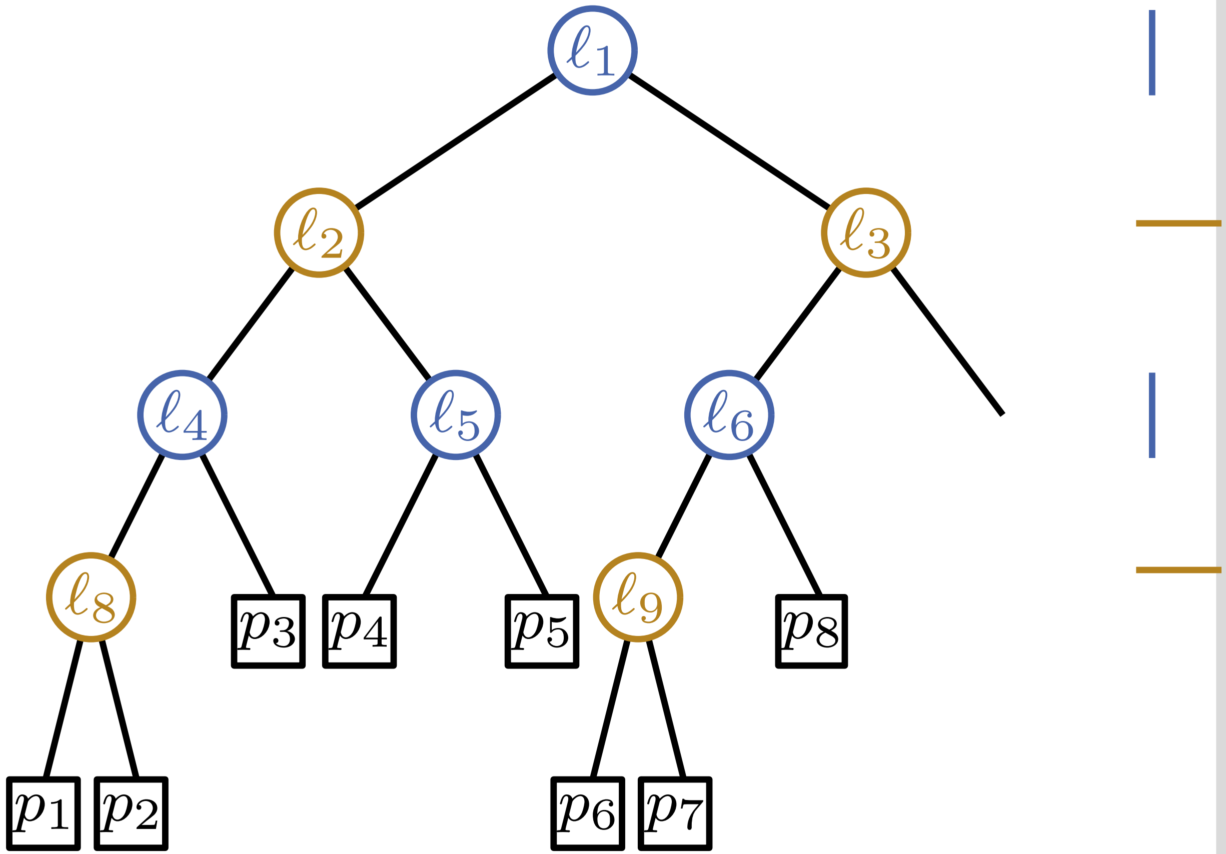
kd-Trees: Example



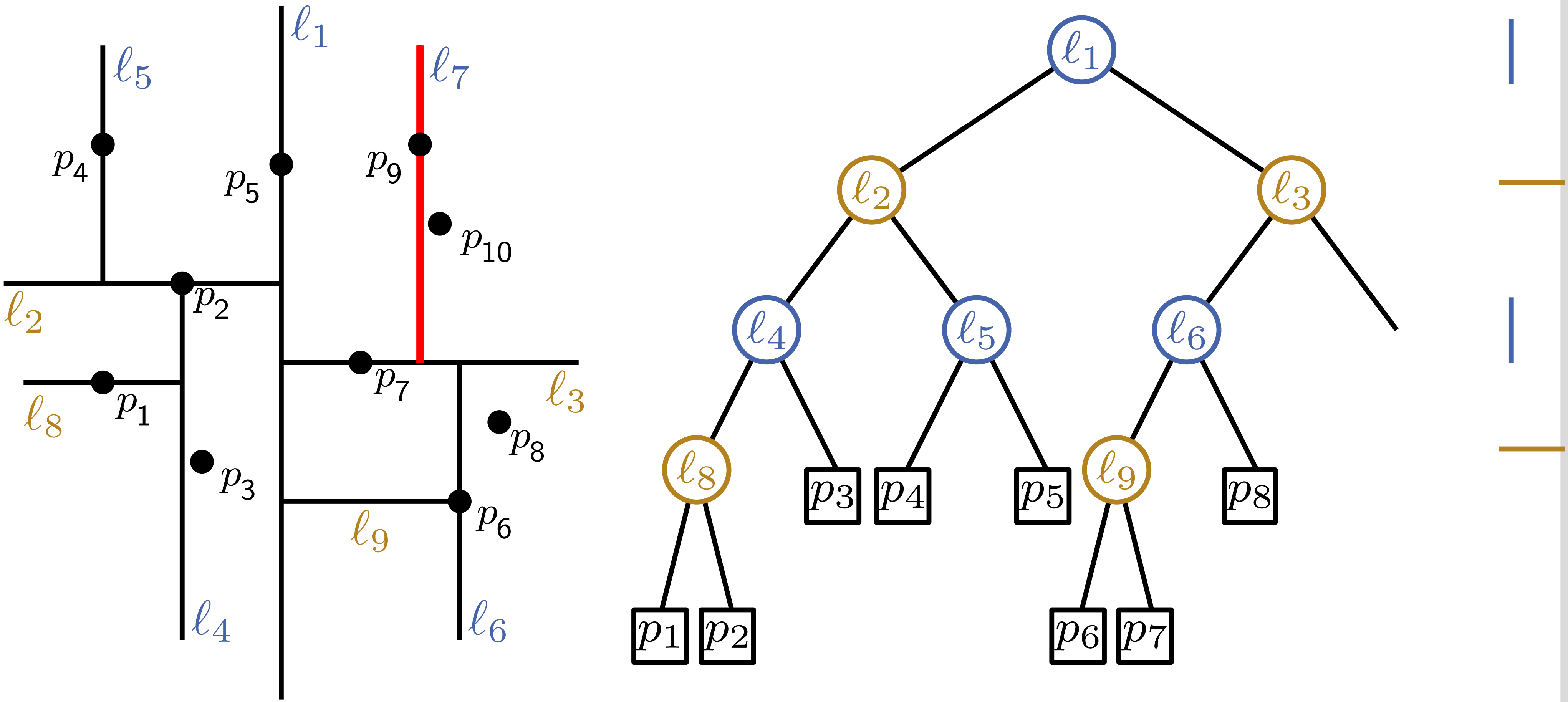


kd-Trees: Example

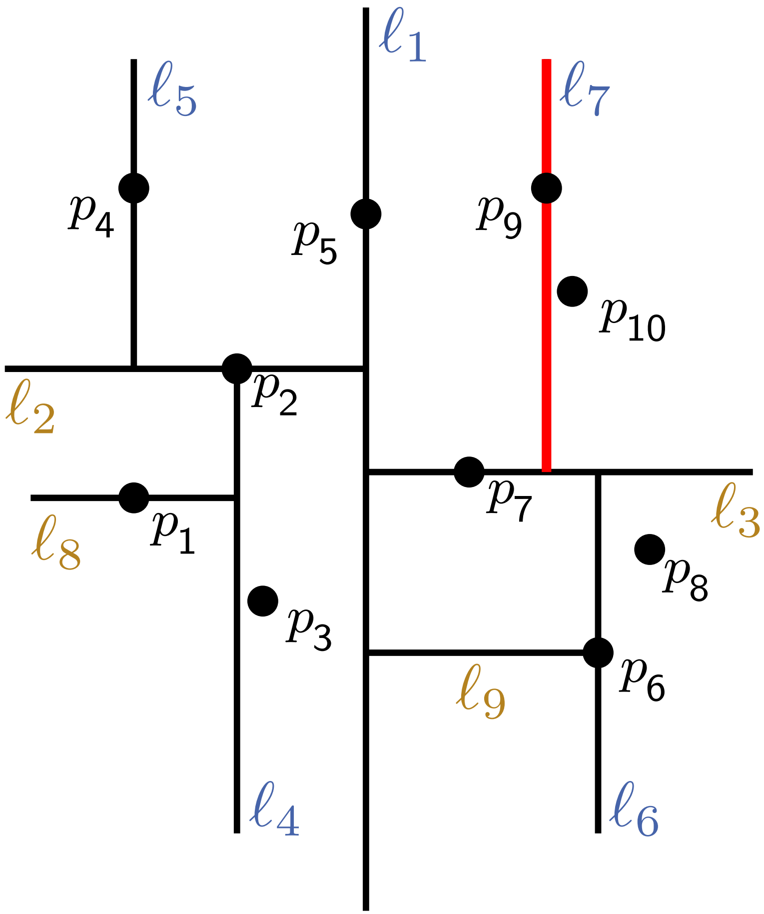


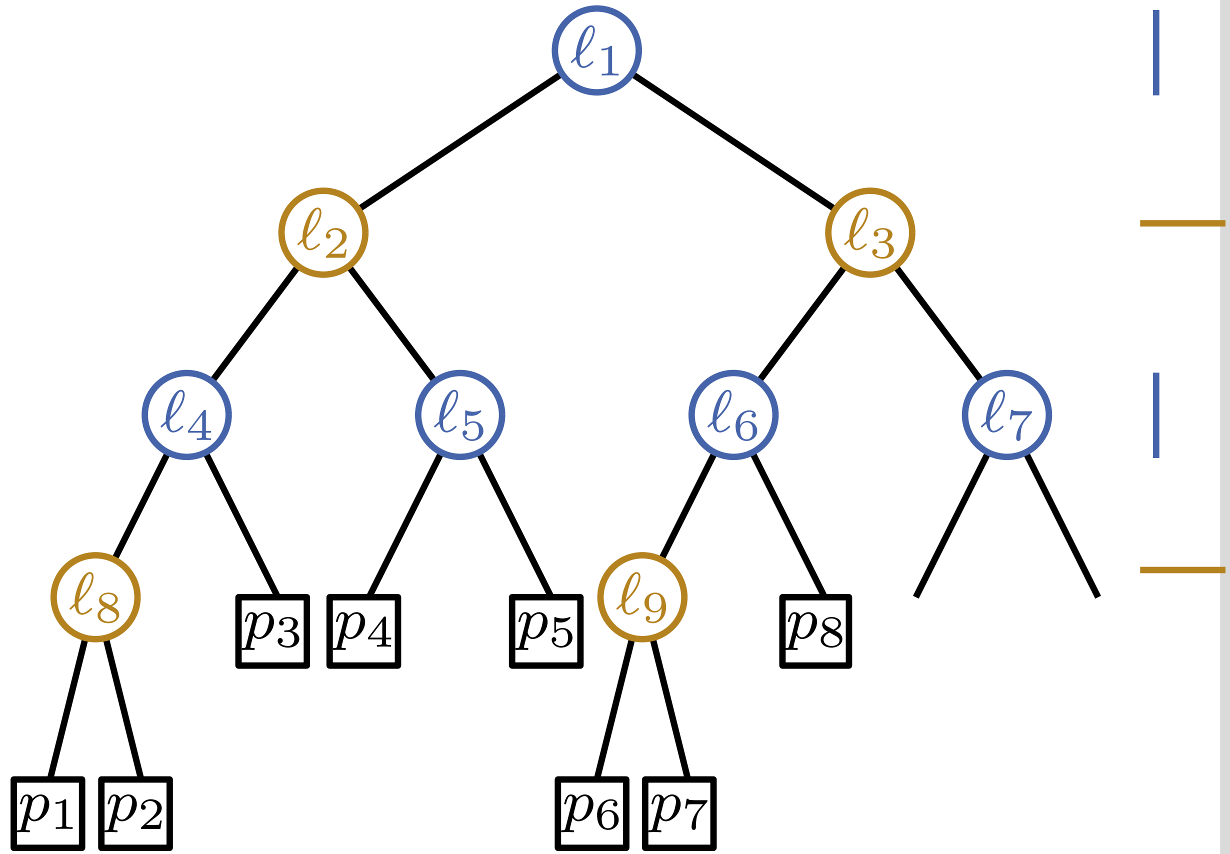


kd-Trees: Example

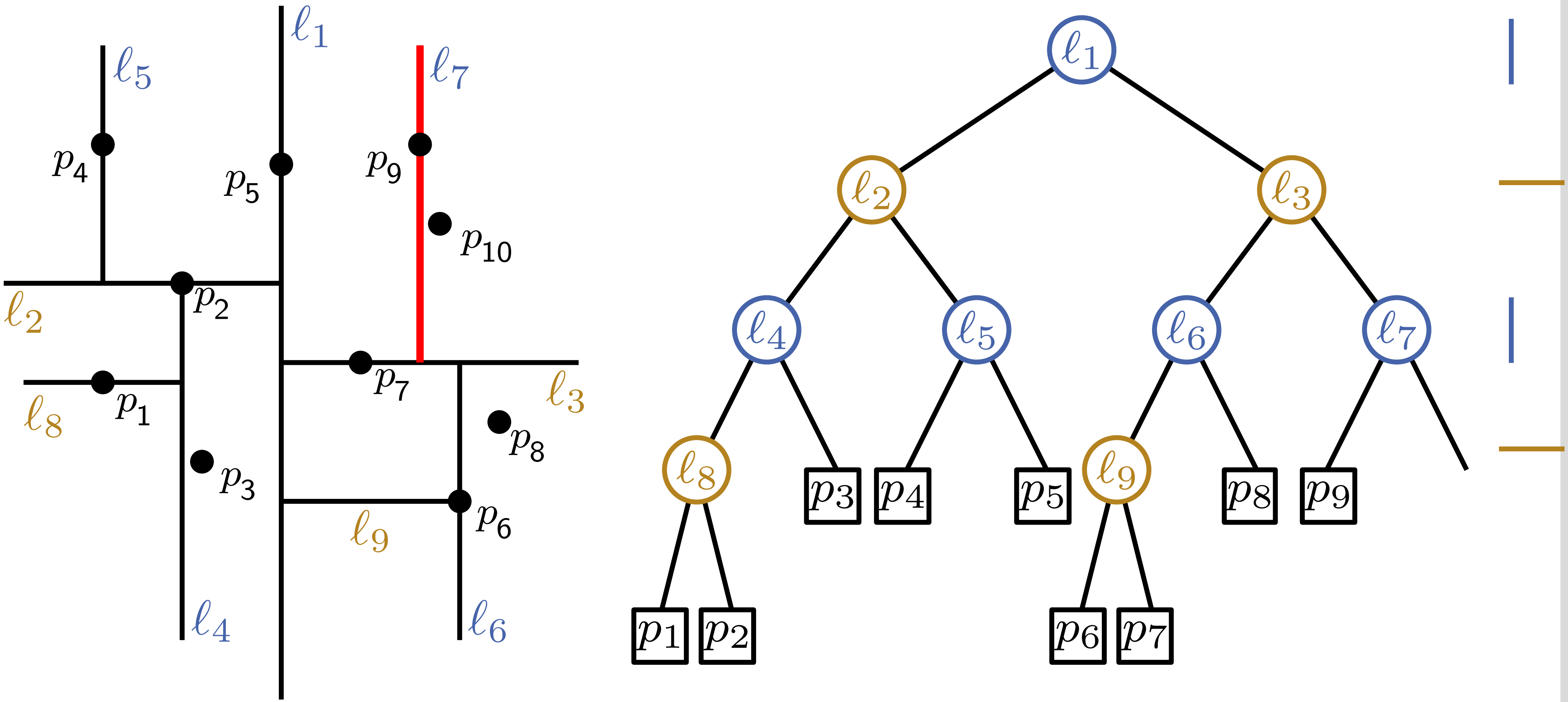


kd-Trees: Example

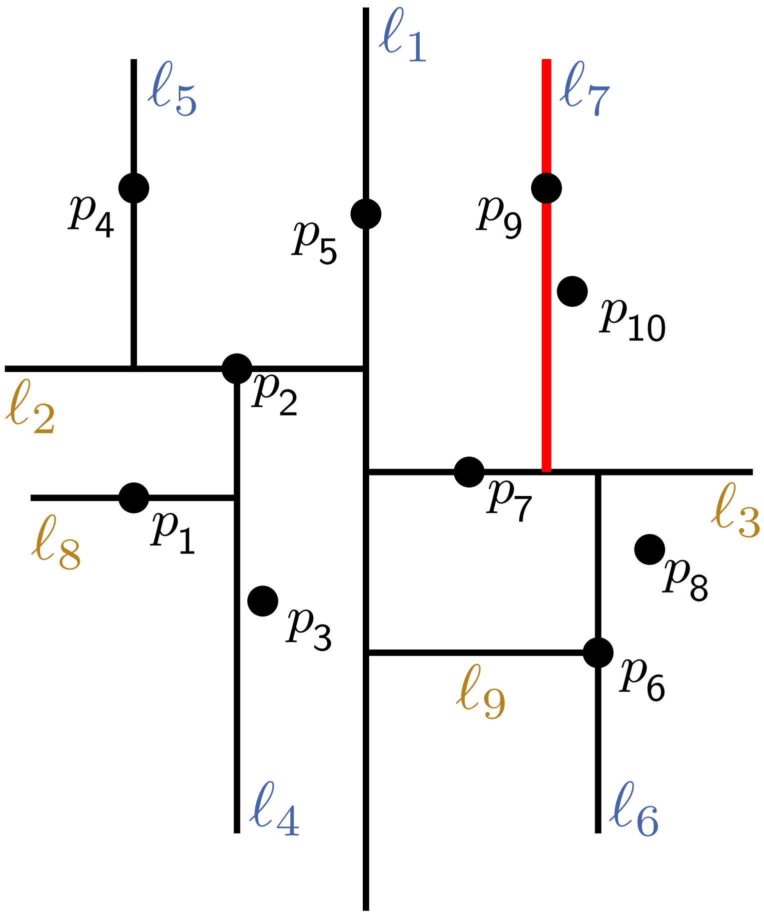


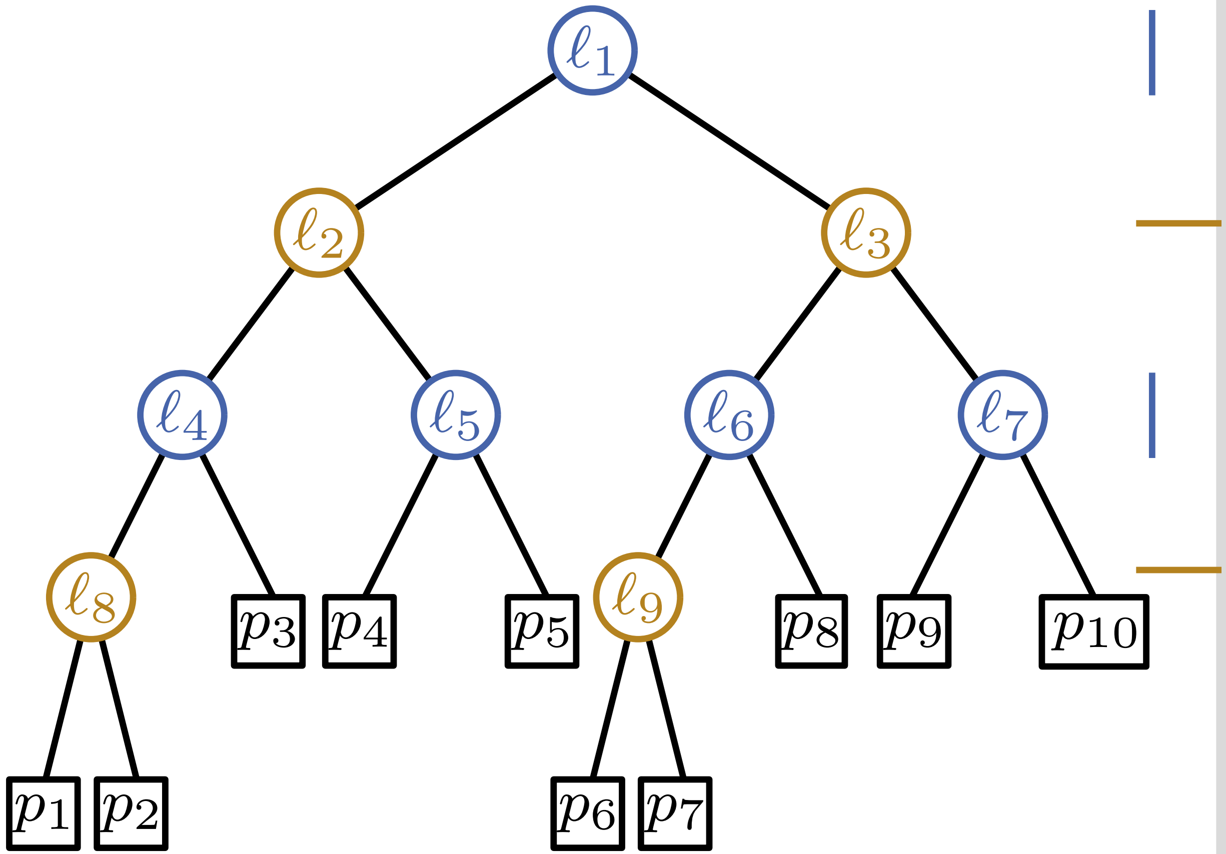


kd-Trees: Example

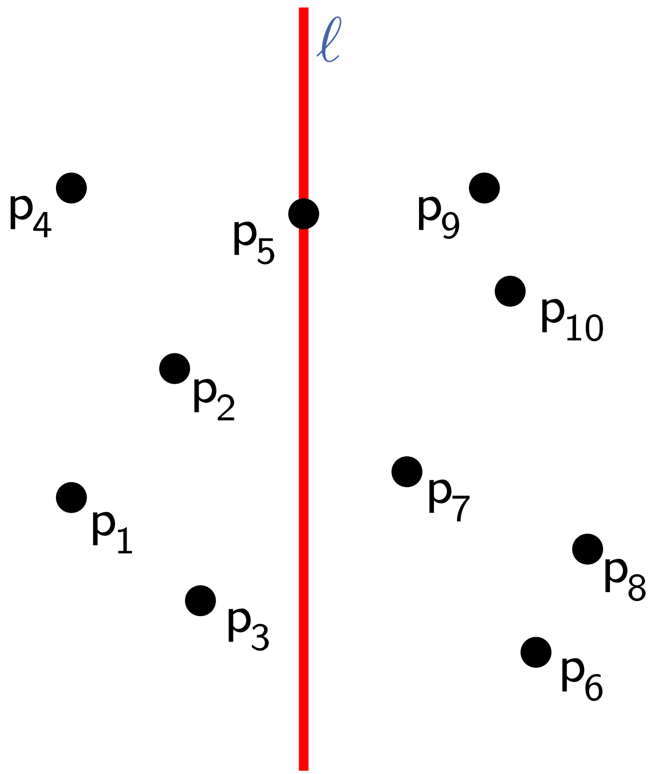


kd-Trees: Example



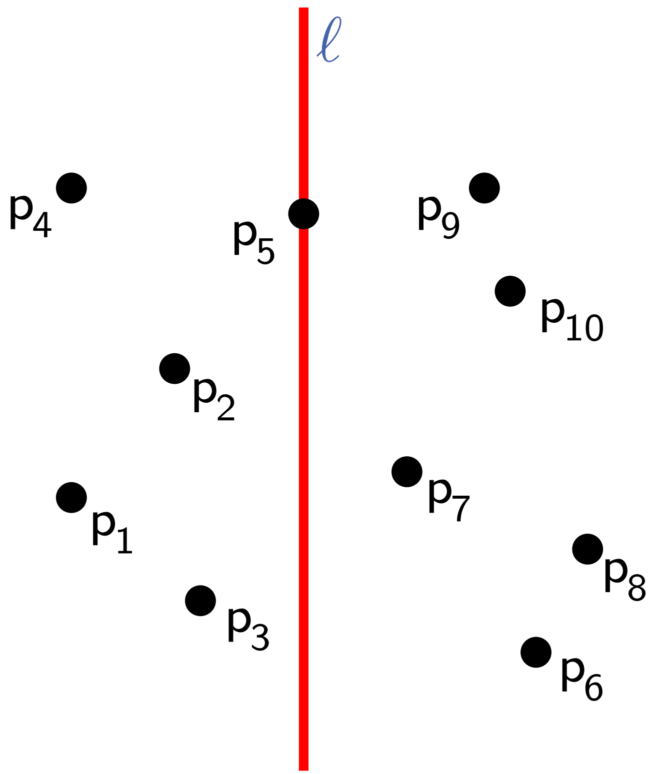


kd-Trees: Construction



BuildKdTree(P, $depth$)

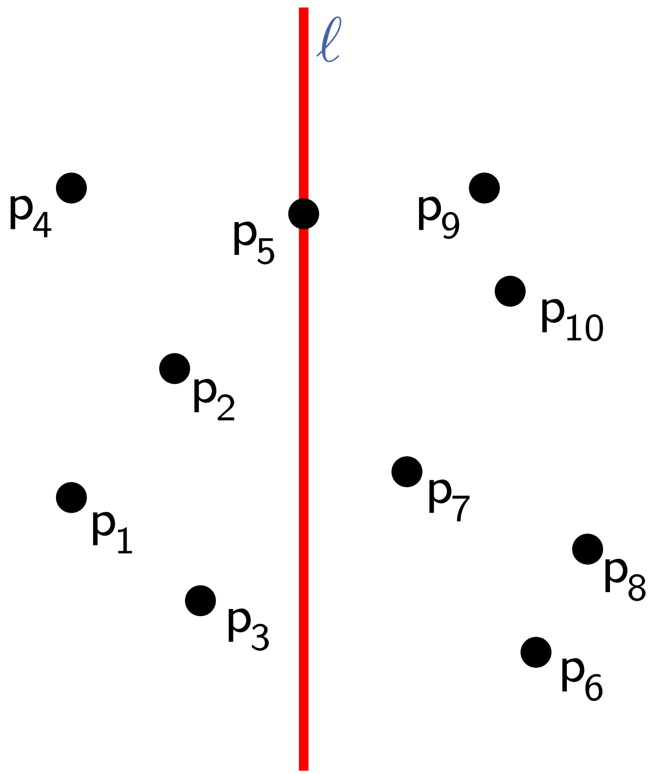
kd-Trees: Construction



BuildKdTree$(P, depth)$

if $|P| = 1$ **then**
  return leaf with the point in P
else

kd-Trees: Construction

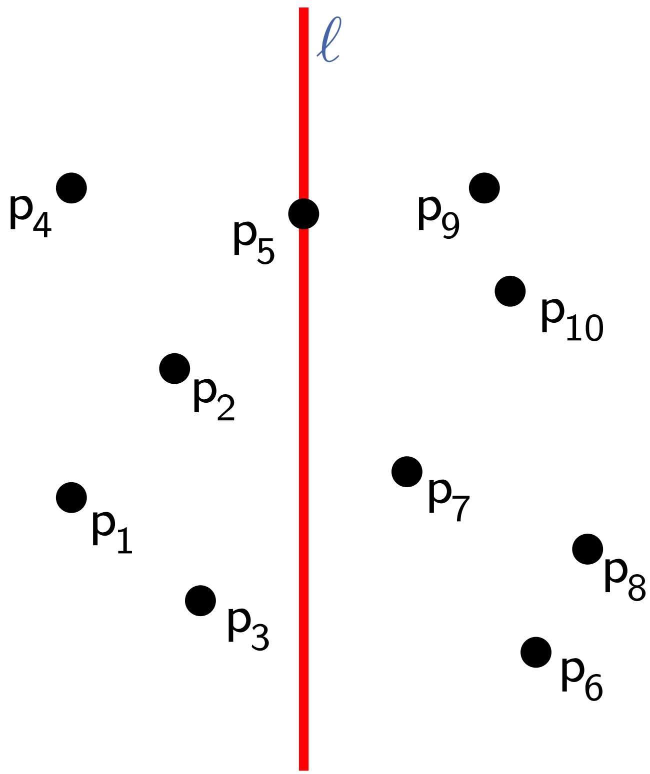


BuildKdTree$(P, depth)$

if $|P| = 1$ **then**
  return leaf with the point in P
else
  if $depth$ is even **then**

  else

kd-Trees: Construction

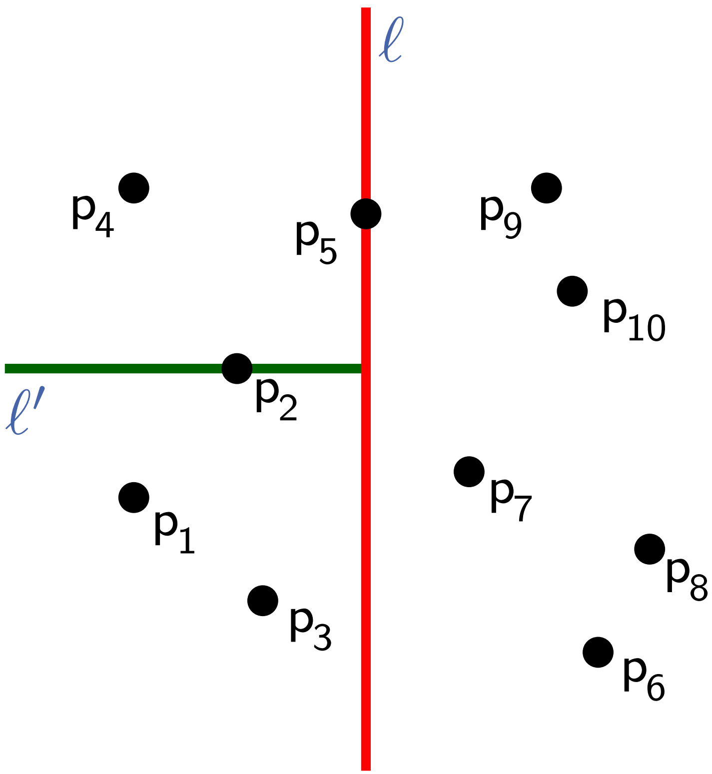


BuildKdTree($P, depth$)

if $|P| = 1$ **then**
  return leaf with the point in P
else
  if $depth$ is even **then**
    divide P vertically at $\ell : x = x_{\text{median}(P)}$ in P_1 (Points left to or on ℓ) and $P_2 = P \setminus P_1$
  else

Point $\lceil |P|/2 \rceil$

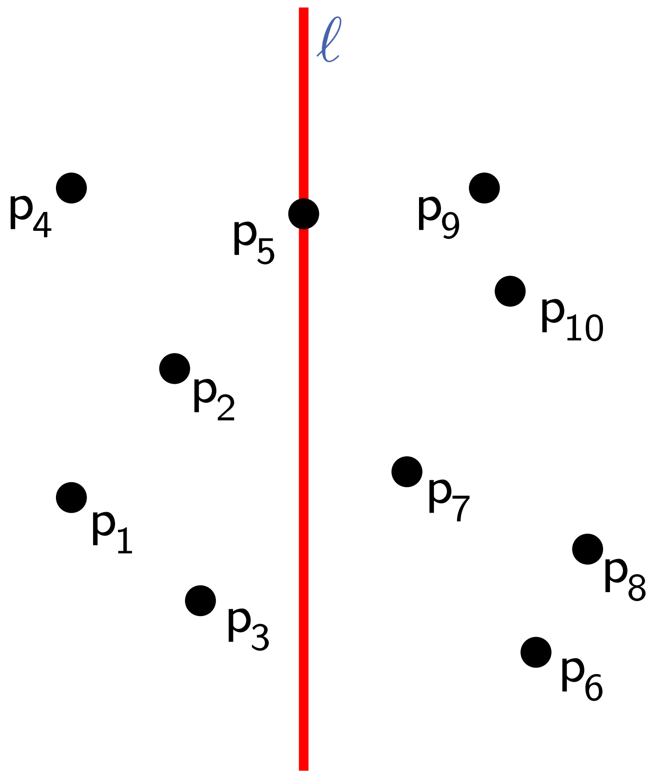
kd-Trees: Construction



BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
  return leaf with the point in P
else
  if $depth$ is even **then**
    divide P vertically at $\ell : x = x_{\text{median}(P)}$ in P_1 (Points left to or on ℓ) and $P_2 = P \setminus P_1$
  else
    divide P horizontally at $\ell : y = y_{\text{median}(P)}$ in P_1 (Points below or on ℓ) und $P_2 = P \setminus P_1$

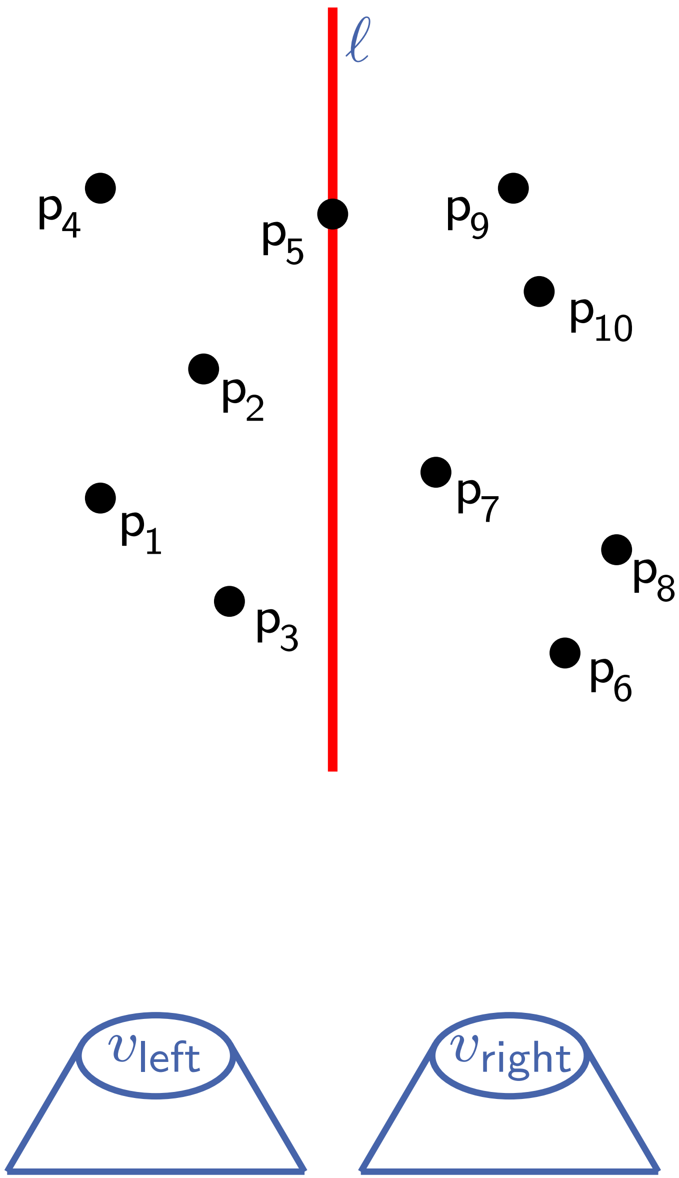
kd-Trees: Construction



BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
  return leaf with the point in P
else
  if $depth$ is even **then**
    divide P vertically at $\ell : x = x_{\text{median}(P)}$ in P_1 (Points left to or on ℓ) and $P_2 = P \setminus P_1$
  else
    divide P horizontally at $\ell : y = y_{\text{median}(P)}$ in P_1 (Points below or on ℓ) und $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildKdTree(P_1, $depth + 1$)
  $v_{\text{right}} \leftarrow$ BuildKdTree(P_2, $depth + 1$)

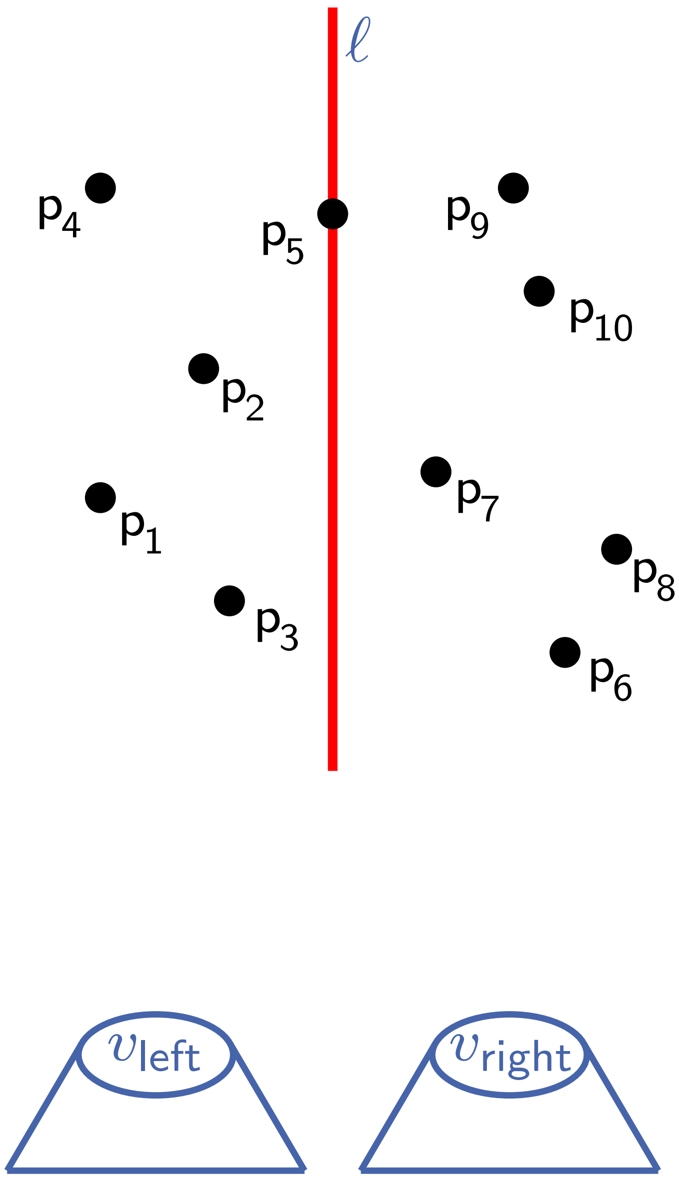
kd-Trees: Construction



BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
  return leaf with the point in P
else
  if $depth$ is even **then**
    divide P vertically at $\ell : x = x_{\text{median}(P)}$ in P_1 (Points left to or on ℓ) and $P_2 = P \setminus P_1$
  else
    divide P horizontally at $\ell : y = y_{\text{median}(P)}$ in P_1 (Points below or on ℓ) und $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildKdTree(P_1, $depth + 1$)
  $v_{\text{right}} \leftarrow$ BuildKdTree(P_2, $depth + 1$)

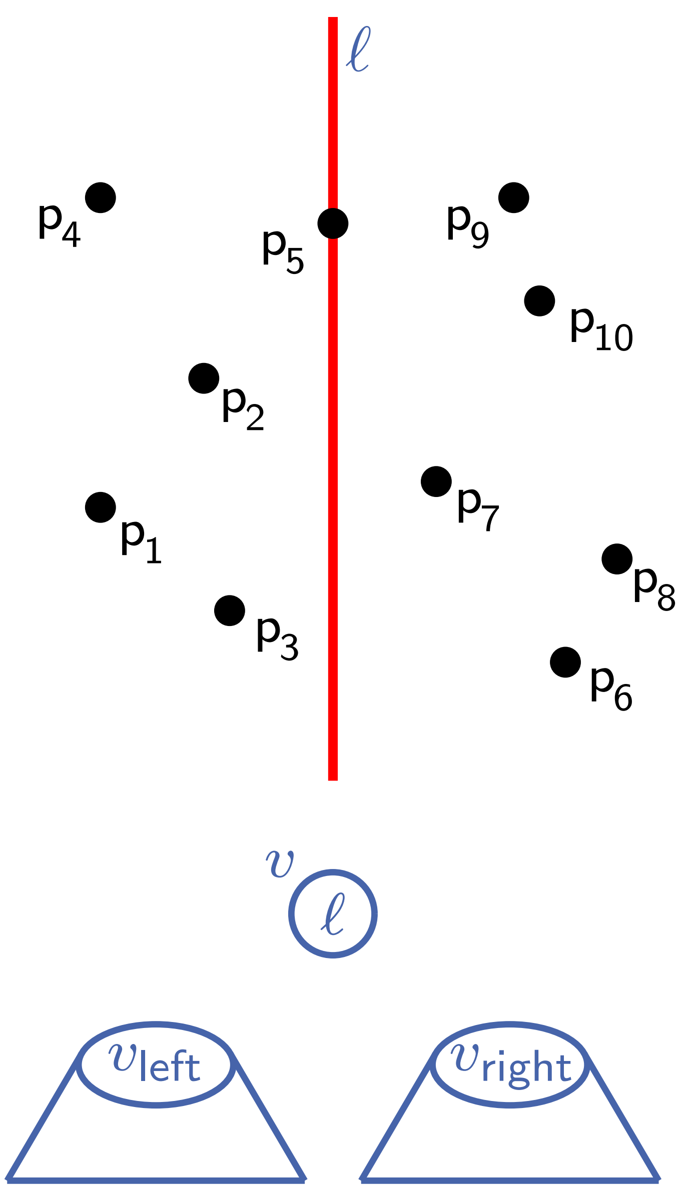
kd-Trees: Construction



BuildKdTree(P, $depth$)

```
if |P| = 1 then
    return leaf with the point in P
else
    if depth is even then
        divide P vertically at
         ℓ : x = x_median(P) in
        P1 (Points left to or on ℓ) and
         P2 = P \ P1
    else
        divide P horizontally at
         ℓ : y = y_median(P) in
        P1 (Points below or on ℓ) und
         P2 = P \ P1
    v_left ← BuildKdTree(P1, depth + 1)
    v_right ← BuildKdTree(P2, depth + 1)
    Create node v, which stores ℓ
```

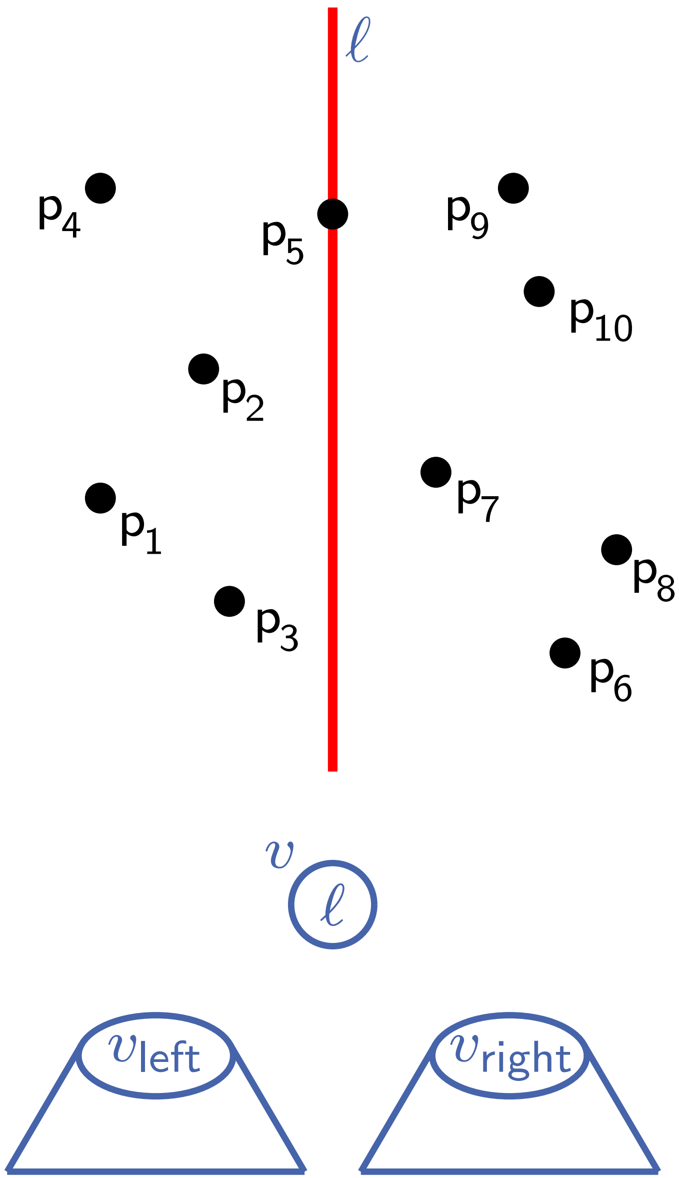
kd-Trees: Construction



BuildKdTree($P, depth$)

if $|P| = 1$ **then**
  return leaf with the point in P
else
  if $depth$ is even **then**
    divide P vertically at $\ell : x = x_{\text{median}(P)}$ in P_1 (Points left to or on ℓ) and $P_2 = P \setminus P_1$
  else
    divide P horizontally at $\ell : y = y_{\text{median}(P)}$ in P_1 (Points below or on ℓ) und $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildKdTree($P_1, depth + 1$)
  $v_{\text{right}} \leftarrow$ BuildKdTree($P_2, depth + 1$)
  Create node v, which stores ℓ

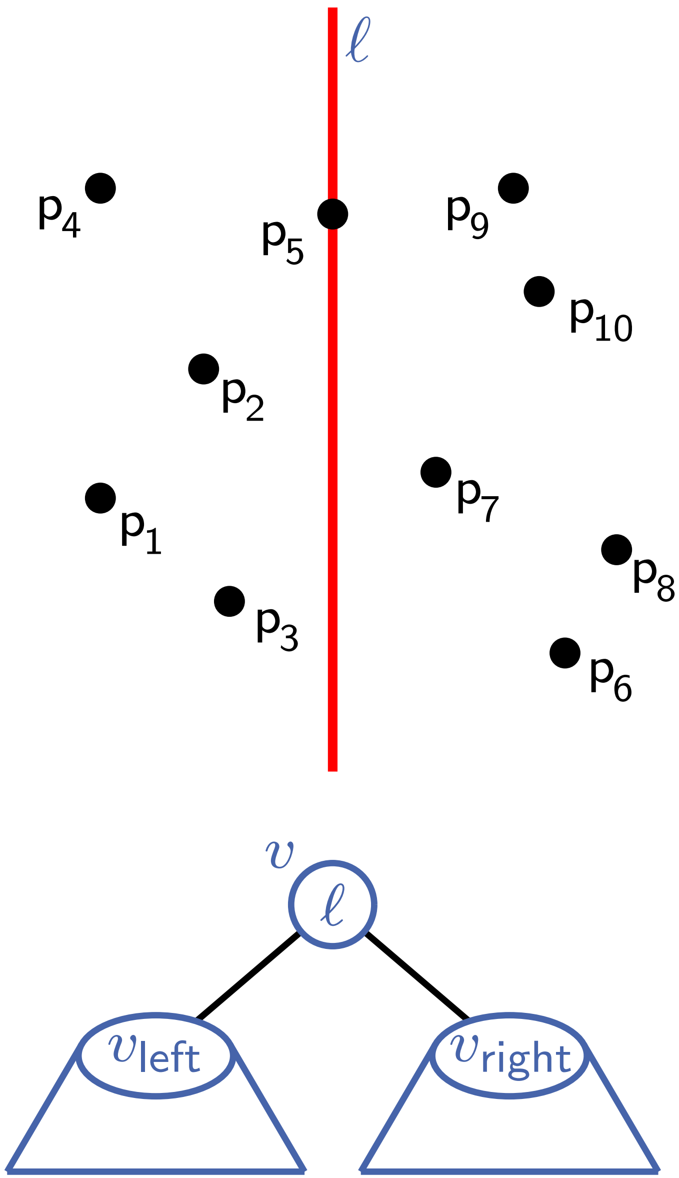
kd-Trees: Construction



BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
  return leaf with the point in P
else
  if $depth$ is even **then**
    divide P vertically at $\ell : x = x_{\text{median}(P)}$ in P_1 (Points left to or on ℓ) and $P_2 = P \setminus P_1$
  else
    divide P horizontally at $\ell : y = y_{\text{median}(P)}$ in P_1 (Points below or on ℓ) und $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildKdTree(P_1, $depth + 1$)
  $v_{\text{right}} \leftarrow$ BuildKdTree(P_2, $depth + 1$)
  Create node v, which stores ℓ
  Make v_{left} und v_{right} children of v

kd-Trees: Construction

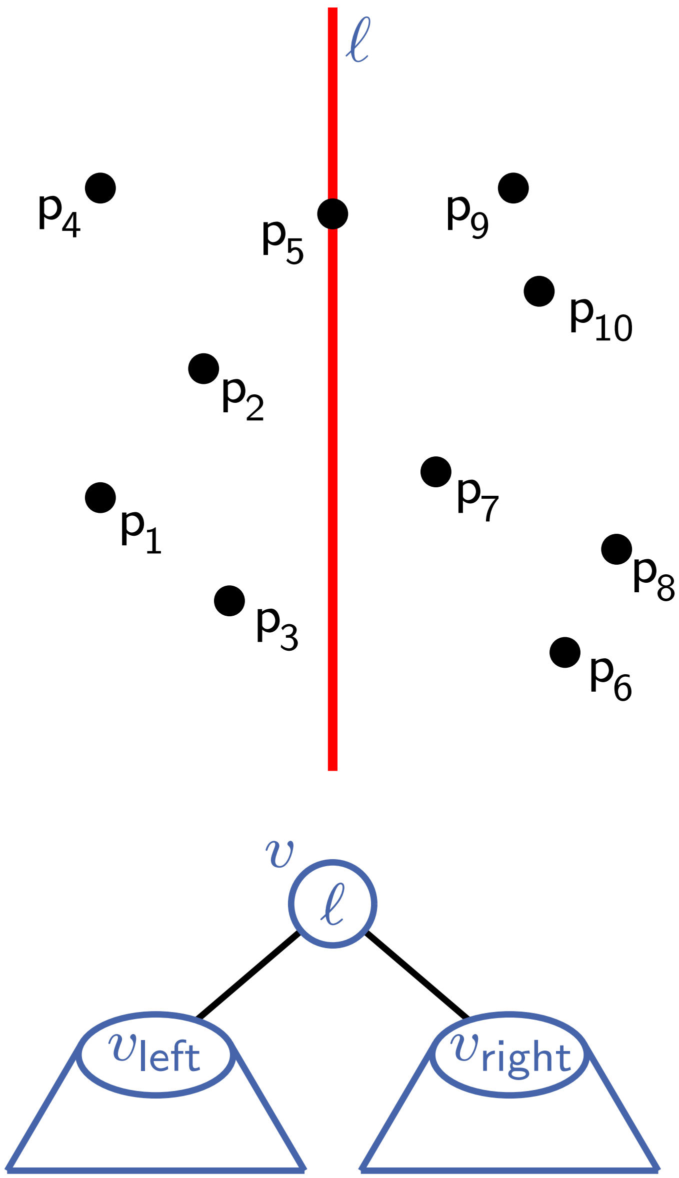


BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
- **return** leaf with the point in P

else
- **if** $depth$ is even **then**
  - divide P vertically at $\ell : x = x_{\text{median}(P)}$ in P_1 (Points left to or on ℓ) and $P_2 = P \setminus P_1$
- **else**
  - divide P horizontally at $\ell : y = y_{\text{median}(P)}$ in P_1 (Points below or on ℓ) und $P_2 = P \setminus P_1$
- $v_{\text{left}} \leftarrow$ BuildKdTree(P_1, $depth + 1$)
- $v_{\text{right}} \leftarrow$ BuildKdTree(P_2, $depth + 1$)
- Create node v, which stores ℓ
- Make v_{left} und v_{right} children of v

kd-Trees: Construction



BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
  return leaf with the point in P
else
  if $depth$ is even **then**
    divide P vertically at $\ell : x = x_{\text{median}(P)}$ in P_1 (Points left to or on ℓ) and $P_2 = P \setminus P_1$
  else
    divide P horizontally at $\ell : y = y_{\text{median}(P)}$ in P_1 (Points below or on ℓ) und $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildKdTree(P_1, $depth + 1$)
  $v_{\text{right}} \leftarrow$ BuildKdTree(P_2, $depth + 1$)
  Create node v, which stores ℓ
  Make v_{left} und v_{right} children of v
  return v

Analysis of kd-Tree Construction

Lemma 1: A kd-tree for n points in $\mathbb{R}^2$ can be constructed in $O(n \log n)$ time, using $O(n)$ space.

Analysis of kd-Tree Construction

Lemma 1: A kd-tree for n points in $\mathbb{R}^2$ can be constructed in $O(n \log n)$ time, using $O(n)$ space.

Proof sketch:

- Determine median:
  - make two lists sorted on x- and y-coordinates
  - at each step, determine median and divide the lists

Analysis of kd-Tree Construction

Lemma 1: A kd-tree for n points in $\mathbb{R}^2$ can be constructed in $O(n \log n)$ time, using $O(n)$ space.

Proof sketch:

- Determine median:
  – make two lists sorted on x- and y-coordinates
  – at each step, determine median and divide the lists
- We get the following recurrence:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n) + 2T(\lceil n/2 \rceil) & \text{otherwise} \end{cases}$$

Analysis of kd-Tree Construction

Lemma 1: A kd-tree for n points in $\mathbb{R}^2$ can be constructed in $O(n \log n)$ time, using $O(n)$ space.

Proof sketch:

- Determine median:
  - make two lists sorted on x- and y-coordinates
  - at each step, determine median and divide the lists
- We get the following recurrence:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n) + 2T(\lceil n/2 \rceil) & \text{otherwise} \end{cases}$$

- Solves to $T(n) = O(n \log n)$ (analogous to MergeSort)

Analysis of kd-Tree Construction

Lemma 1: A kd-tree for n points in $\mathbb{R}^2$ can be constructed in $O(n \log n)$ time, using $O(n)$ space.

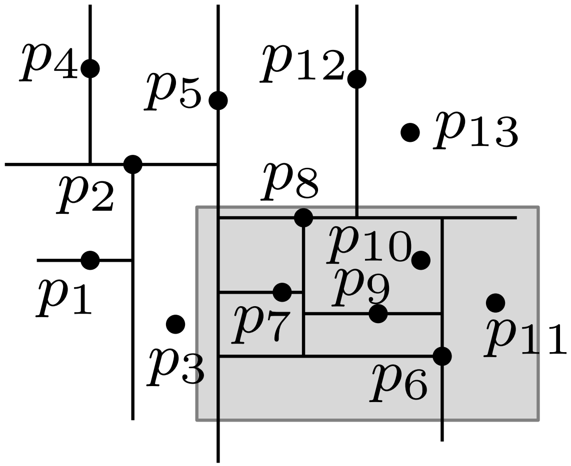
Proof sketch:

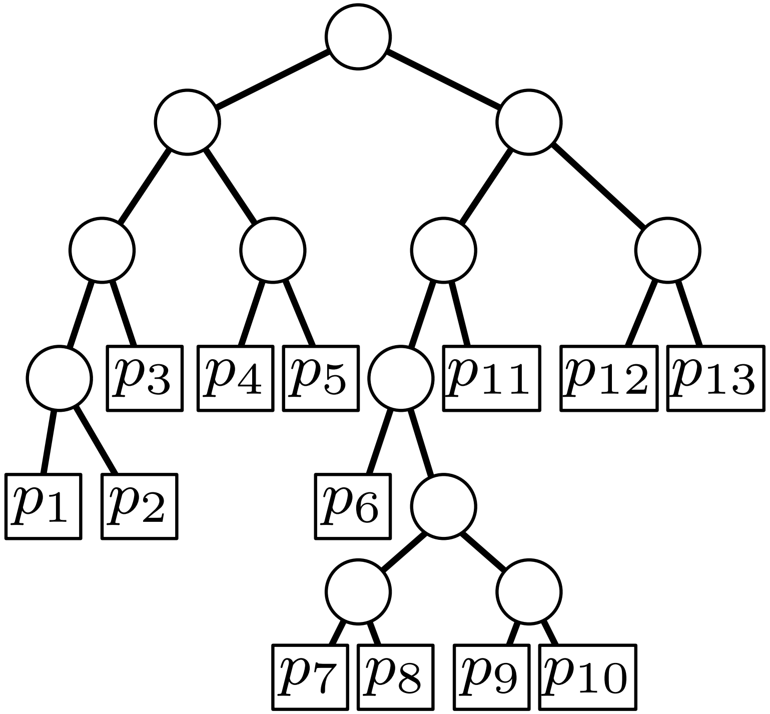
- Determine median:
  - make two lists sorted on x- and y-coordinates
  - at each step, determine median and divide the lists
- We get the following recurrence:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n) + 2T(\lceil n/2 \rceil) & \text{otherwise} \end{cases}$$

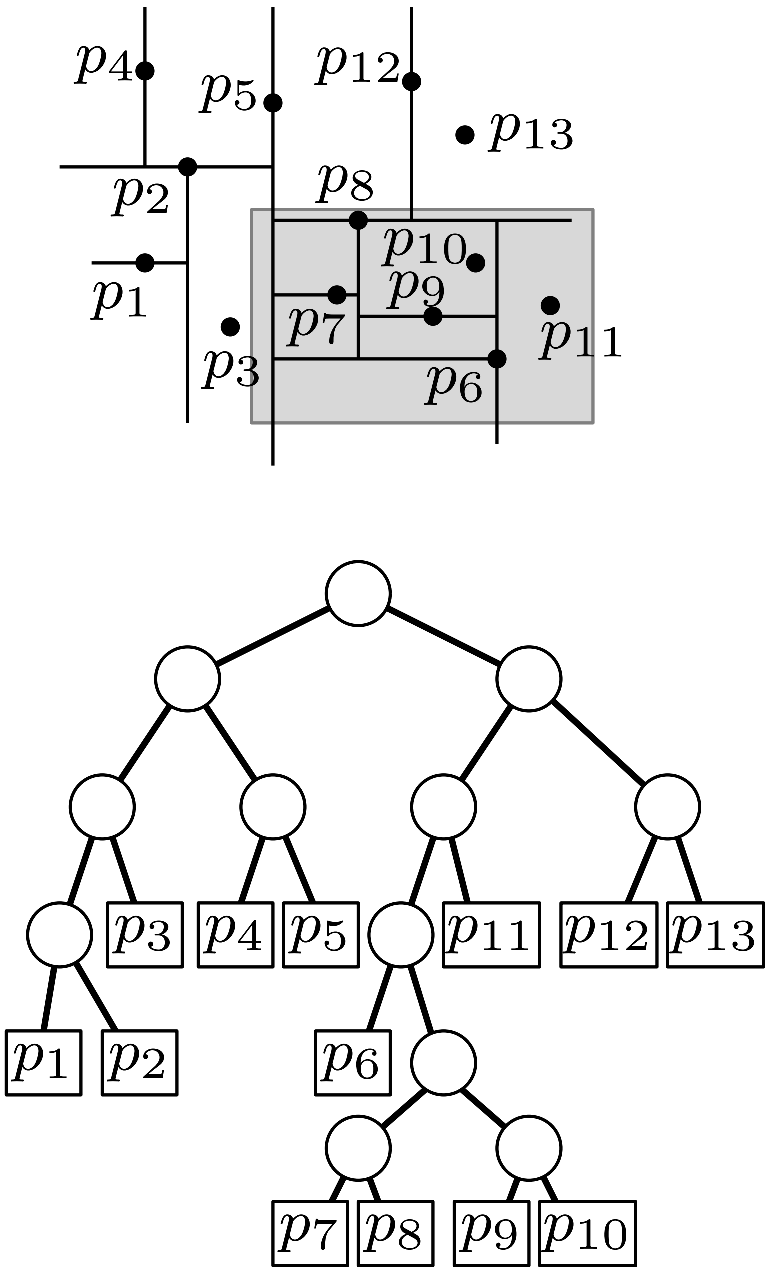
- Solves to $T(n) = O(n \log n)$ (analogous to MergeSort)
- Linear space, since we are using a binary tree with n leaves.

Range Queries in a kd-Tree





Range Queries in a kd-Tree

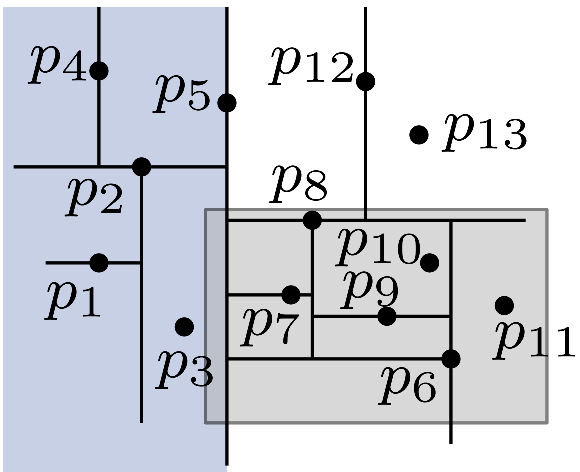


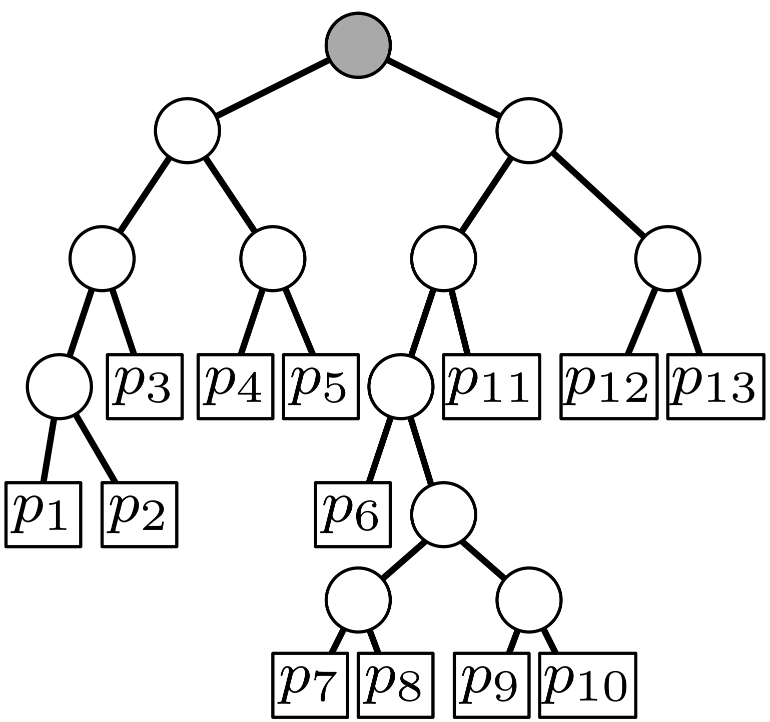
SearchKdTree(v, R)

if v leaf **then**
- report point p in v when $p \in R$

else
- **if** region(lc(v)) $\subseteq R$ **then**
  - ReportSubtree(lc(v))
- **else**
  - **if** region(lc(v))$\cap R \neq \emptyset$ **then**
  - SearchKdTree(lc(v), R)
- **if** region(rc(v)) $\subseteq R$ **then**
  - ReportSubtree(rc(v))
- **else**
  - **if** region(rc(v))$\cap R \neq \emptyset$ **then**
  - SearchKdTree(rc(v), R)

Range Queries in a kd-Tree

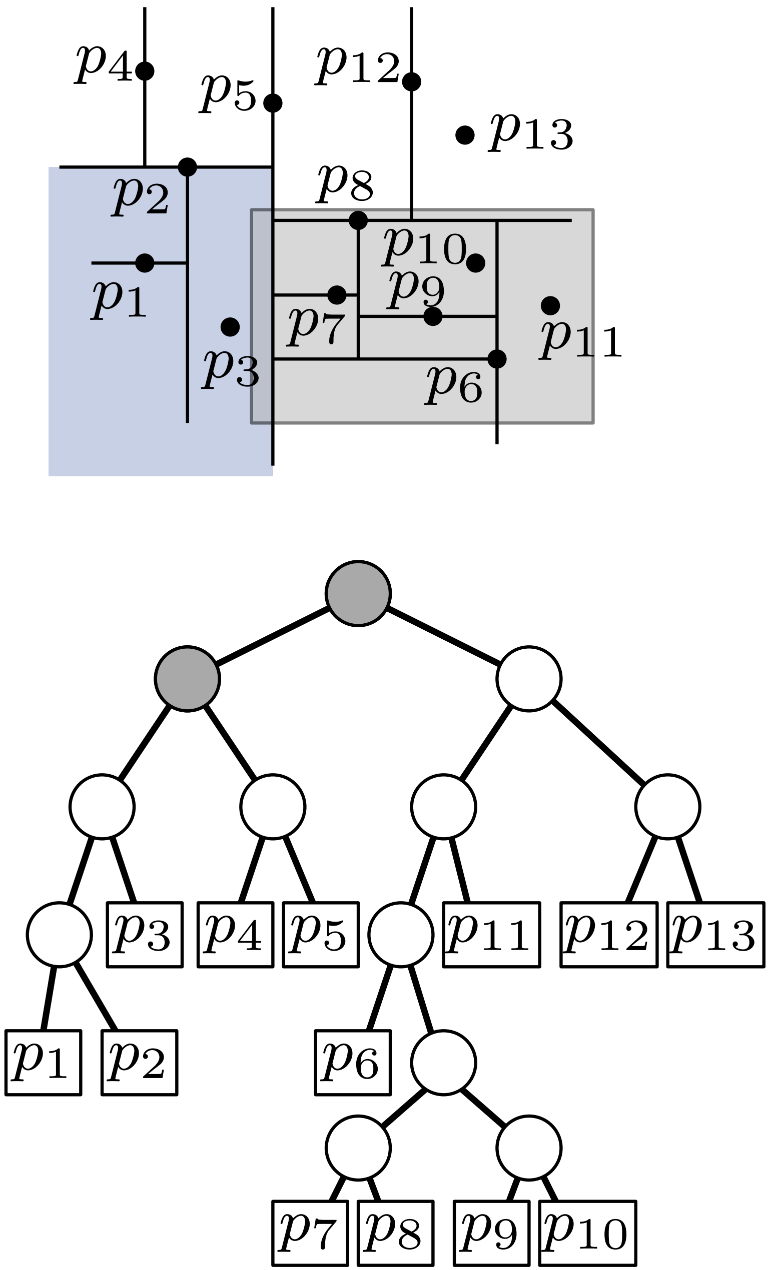




SearchKdTree(v, R)

- **if** v leaf **then**
  - report point p in v when $p \in R$
- **else**
  - **if** region(lc(v)) $\subseteq R$ **then**
    - ReportSubtree(lc(v))
  - **else**
    - **if** region(lc(v))$\cap R \neq \emptyset$ **then**
    - SearchKdTree(lc(v), R)
  - **if** region(rc(v)) $\subseteq R$ **then**
    - ReportSubtree(rc(v))
  - **else**
    - **if** region(rc(v))$\cap R \neq \emptyset$ **then**
    - SearchKdTree(rc(v), R)

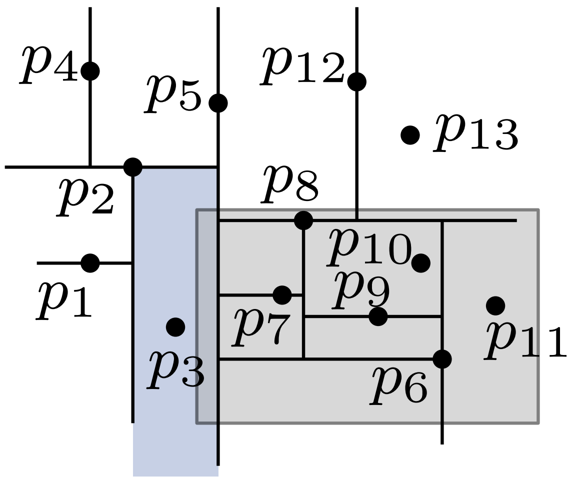
Range Queries in a kd-Tree

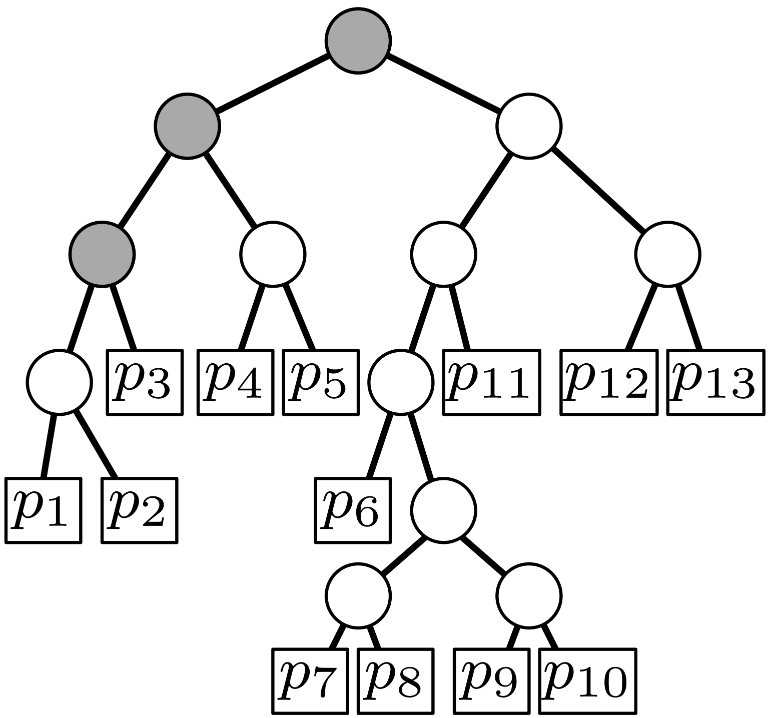


SearchKdTree(v, R)

- **if** v leaf **then**
  - report point p in v when $p \in R$
- **else**
  - **if** region(lc(v)) $\subseteq R$ **then**
    - ReportSubtree(lc(v))
  - **else**
    - **if** region(lc(v))$\cap R \neq \emptyset$ **then**
    - SearchKdTree(lc(v), R)
  - **if** region(rc(v)) $\subseteq R$ **then**
    - ReportSubtree(rc(v))
  - **else**
    - **if** region(rc(v))$\cap R \neq \emptyset$ **then**
    - SearchKdTree(rc(v), R)

Range Queries in a kd-Tree



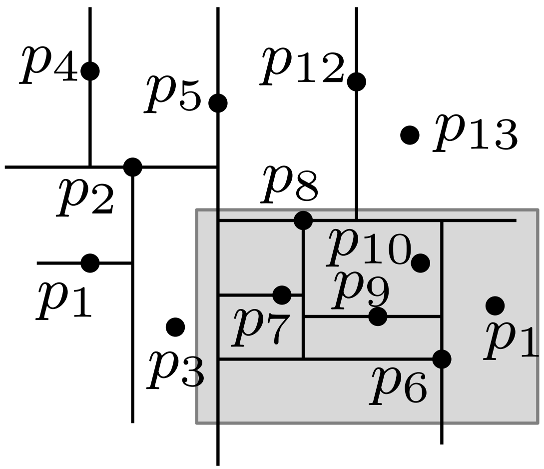


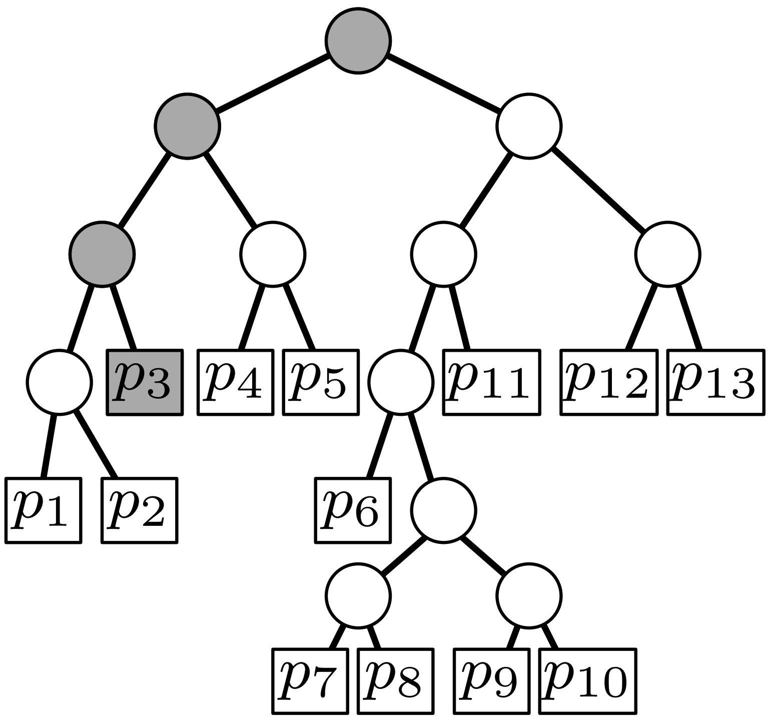
SearchKdTree(v, R)

if v leaf **then**
- report point p in v when $p \in R$

else
- **if** region(lc(v)) $\subseteq R$ **then**
  - ReportSubtree(lc(v))
- **else**
  - **if** region(lc(v))$\cap R \neq \emptyset$ **then**
  - SearchKdTree(lc(v), R)
- **if** region(rc(v)) $\subseteq R$ **then**
  - ReportSubtree(rc(v))
- **else**
  - **if** region(rc(v))$\cap R \neq \emptyset$ **then**
  - SearchKdTree(rc(v), R)

Range Queries in a kd-Tree

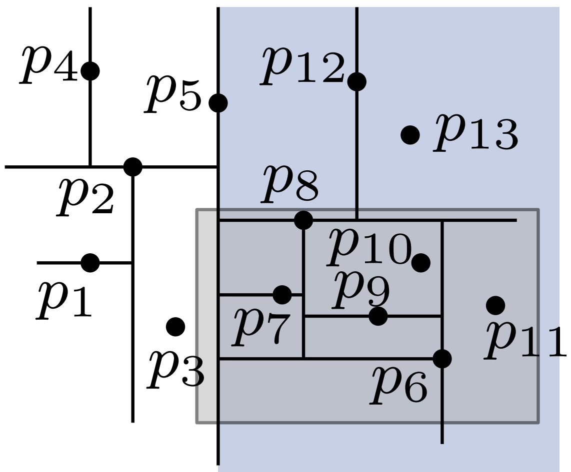


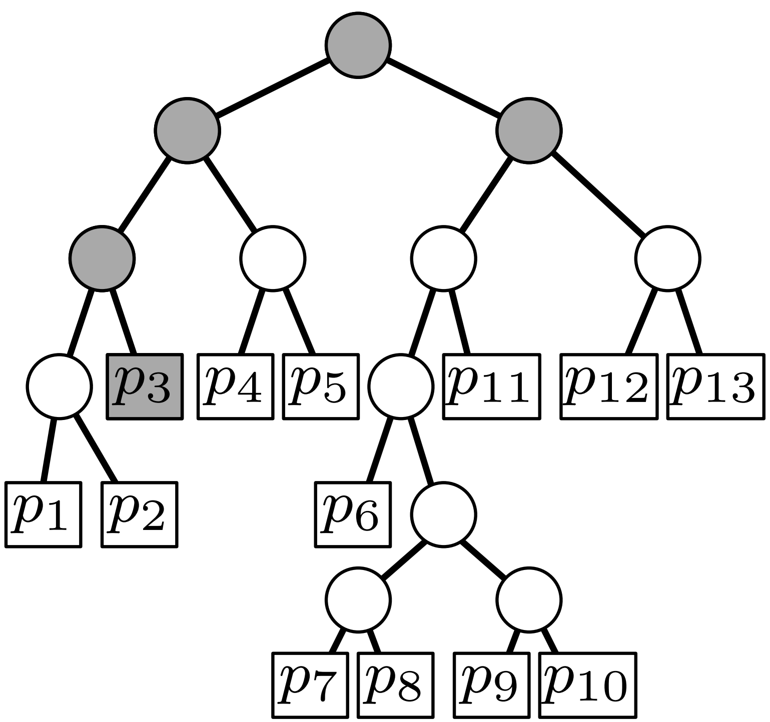


SearchKdTree(v, R)

- **if** v leaf **then**
  - report point p in v when $p \in R$
- **else**
  - **if** region(lc(v)) $\subseteq R$ **then**
    - ReportSubtree(lc(v))
  - **else**
    - **if** region(lc(v))$\cap R \neq \emptyset$ **then**
    - SearchKdTree(lc(v), R)
  - **if** region(rc(v)) $\subseteq R$ **then**
    - ReportSubtree(rc(v))
  - **else**
    - **if** region(rc(v))$\cap R \neq \emptyset$ **then**
    - SearchKdTree(rc(v), R)

Range Queries in a kd-Tree

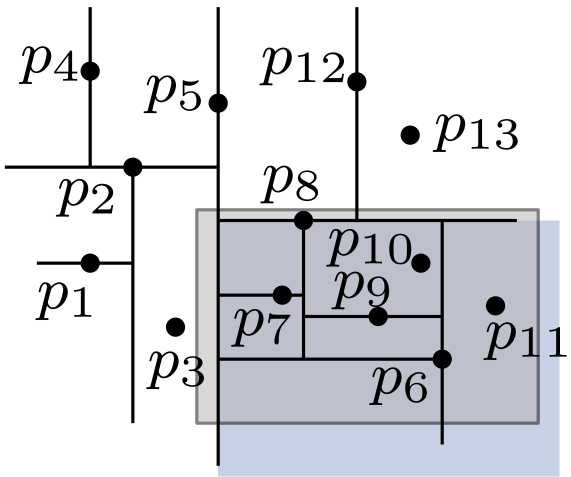


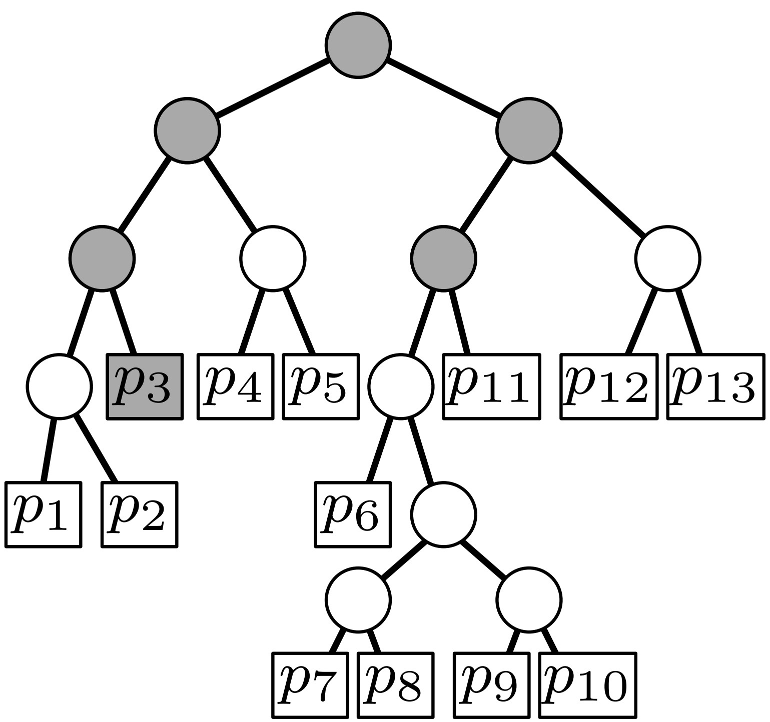


SearchKdTree(v, R)

if v leaf **then**
  report point p in v when $p \in R$
else
  if region(lc(v)) $\subseteq R$ **then**
    ReportSubtree(lc(v))
  else
    if region(lc(v))$\cap R \neq \emptyset$ **then**
    SearchKdTree(lc(v), R)
  if region(rc(v)) $\subseteq R$ **then**
    ReportSubtree(rc(v))
  else
    if region(rc(v))$\cap R \neq \emptyset$ **then**
    SearchKdTree(rc(v), R)

Range Queries in a kd-Tree

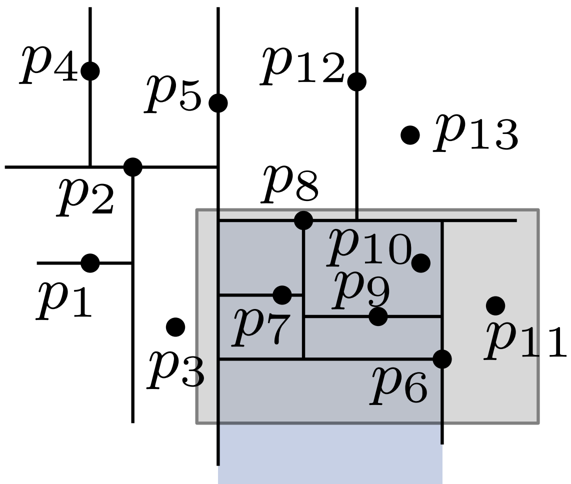


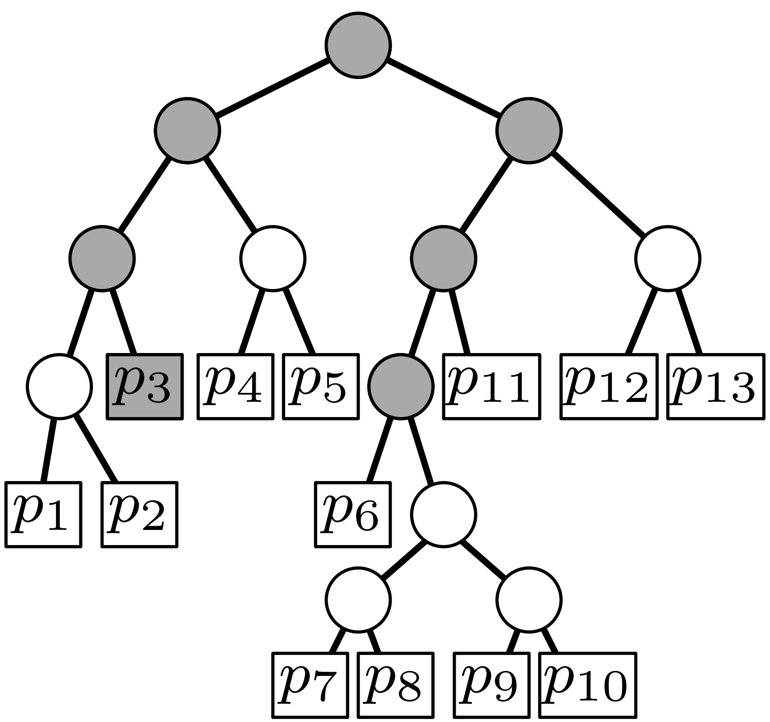


SearchKdTree(v, R)

- **if** v leaf **then**
  - report point p in v when $p \in R$
- **else**
  - **if** region(lc(v)) $\subseteq R$ **then**
    - ReportSubtree(lc(v))
  - **else**
    - **if** region(lc(v))$\cap R \neq \emptyset$ **then**
    - SearchKdTree(lc(v), R)
  - **if** region(rc(v)) $\subseteq R$ **then**
    - ReportSubtree(rc(v))
  - **else**
    - **if** region(rc(v))$\cap R \neq \emptyset$ **then**
    - SearchKdTree(rc(v), R)

Range Queries in a kd-Tree

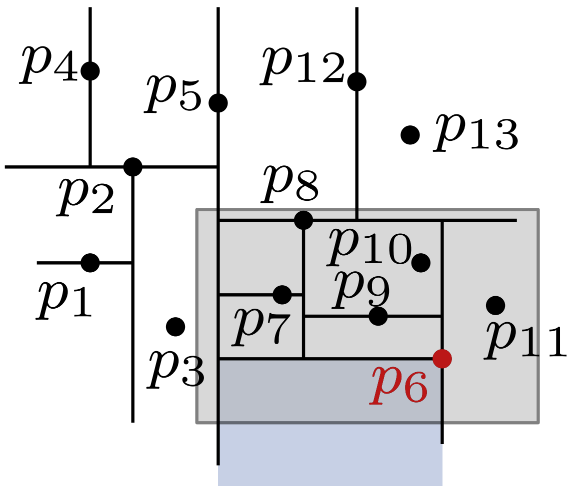


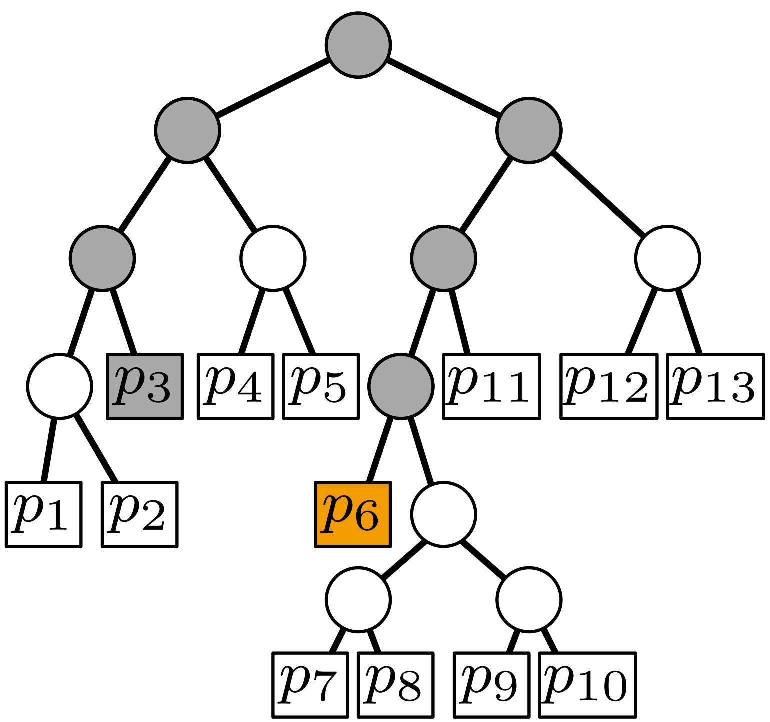


SearchKdTree(v, R)

if v leaf **then**
    report point p in v when $p \in R$
else
    if region(lc(v)) $\subseteq R$ **then**
        ReportSubtree(lc(v))
    else
        if region(lc(v))$\cap R \neq \emptyset$ **then**
        SearchKdTree(lc(v), R)
    if region(rc(v)) $\subseteq R$ **then**
        ReportSubtree(rc(v))
    else
        if region(rc(v))$\cap R \neq \emptyset$ **then**
        SearchKdTree(rc(v), R)

Range Queries in a kd-Tree





SearchKdTree(v, R)

if v leaf **then**

  report point p in v when $p \in R$

else

  if region(lc(v)) $\subseteq R$ **then**

    ReportSubtree(lc(v))

  else

    if region(lc(v))$\cap R \neq \emptyset$ **then**

    SearchKdTree(lc(v), R)

  if region(rc(v)) $\subseteq R$ **then**

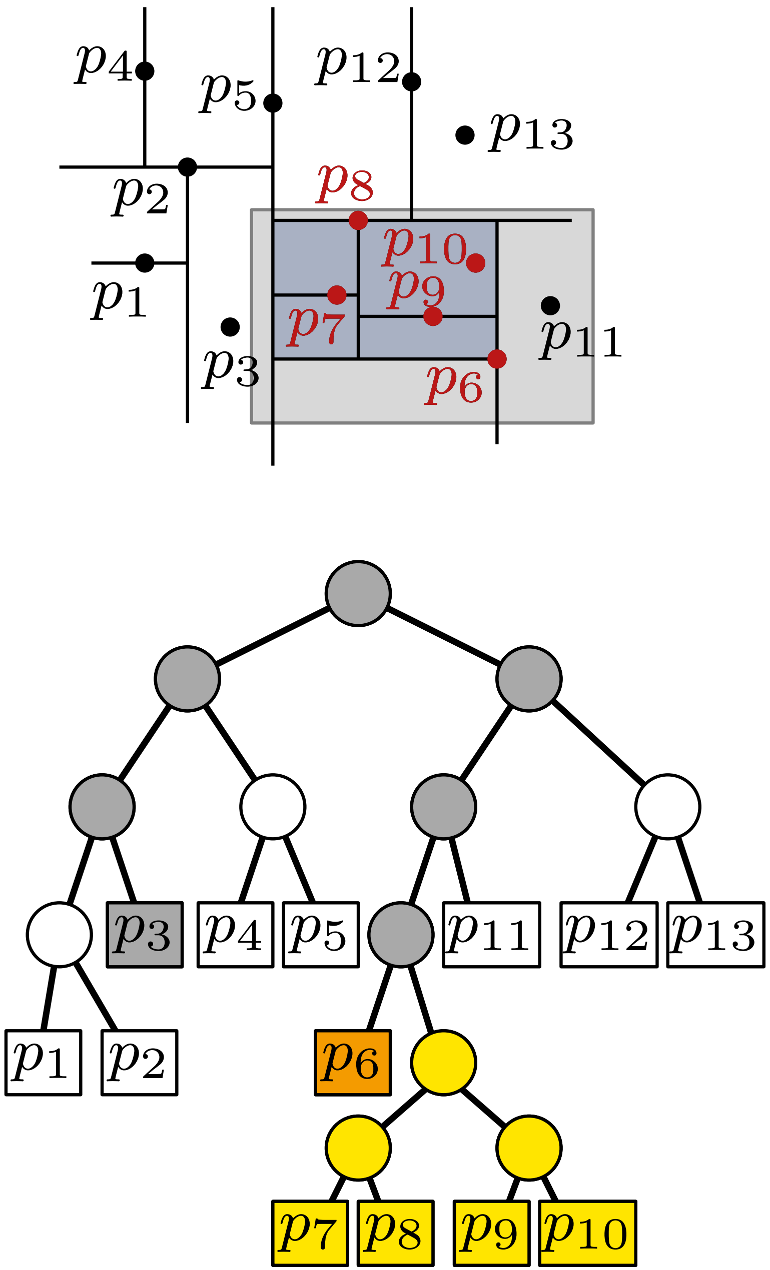
    ReportSubtree(rc(v))

  else

    if region(rc(v))$\cap R \neq \emptyset$ **then**

    SearchKdTree(rc(v), R)

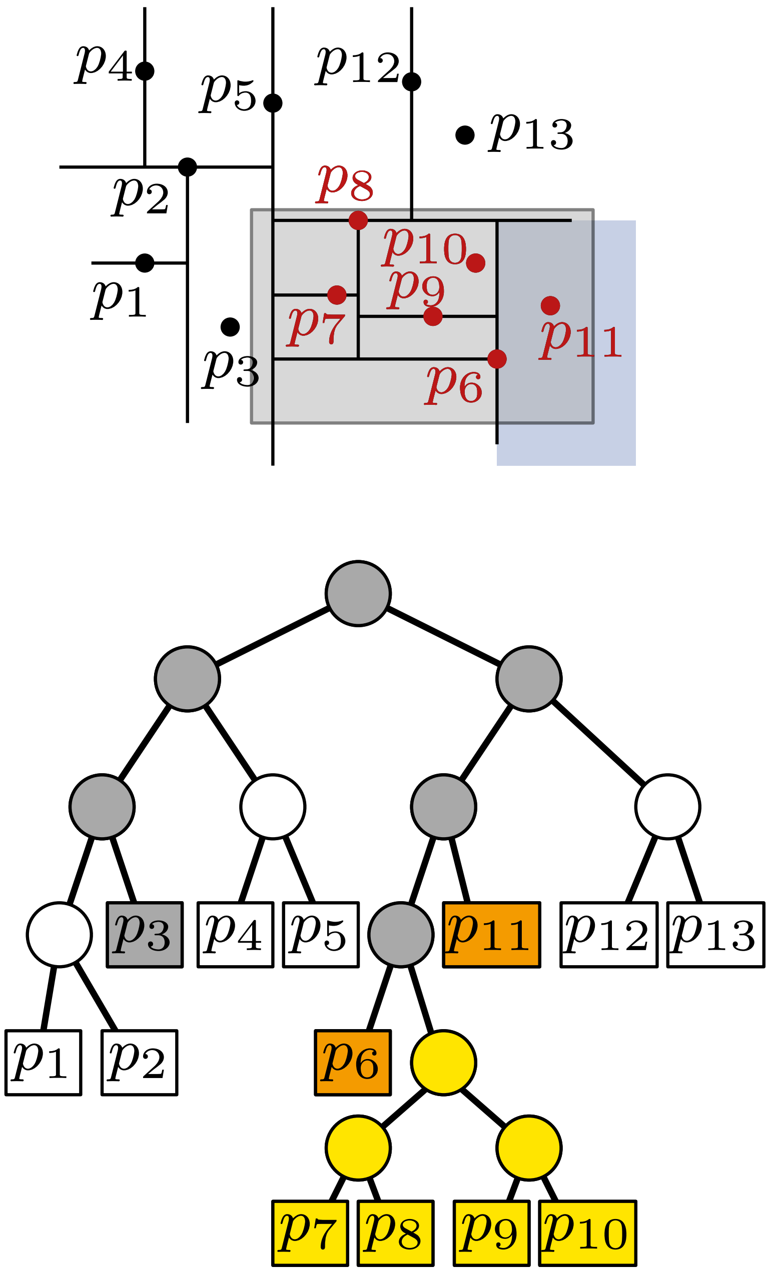
Range Queries in a kd-Tree



SearchKdTree(v, R)

if v leaf **then**
  report point p in v when $p \in R$
else
  if region(lc(v)) $\subseteq R$ **then**
    ReportSubtree(lc(v))
  else
    if region(lc(v))$\cap R \neq \emptyset$ **then**
    SearchKdTree(lc(v), R)
  if region(rc(v)) $\subseteq R$ **then**
    ReportSubtree(rc(v))
  else
    if region(rc(v))$\cap R \neq \emptyset$ **then**
    SearchKdTree(rc(v), R)

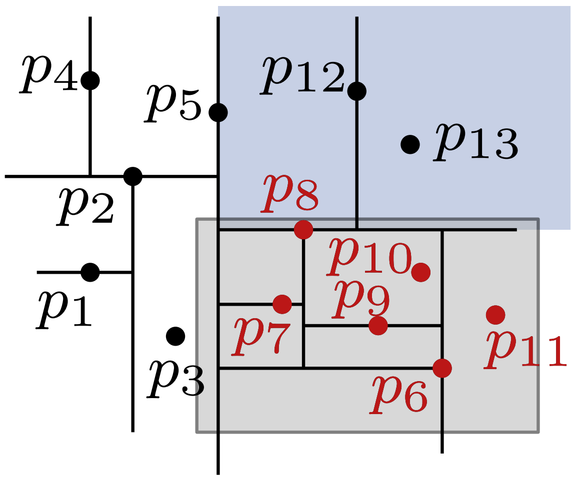
Range Queries in a kd-Tree

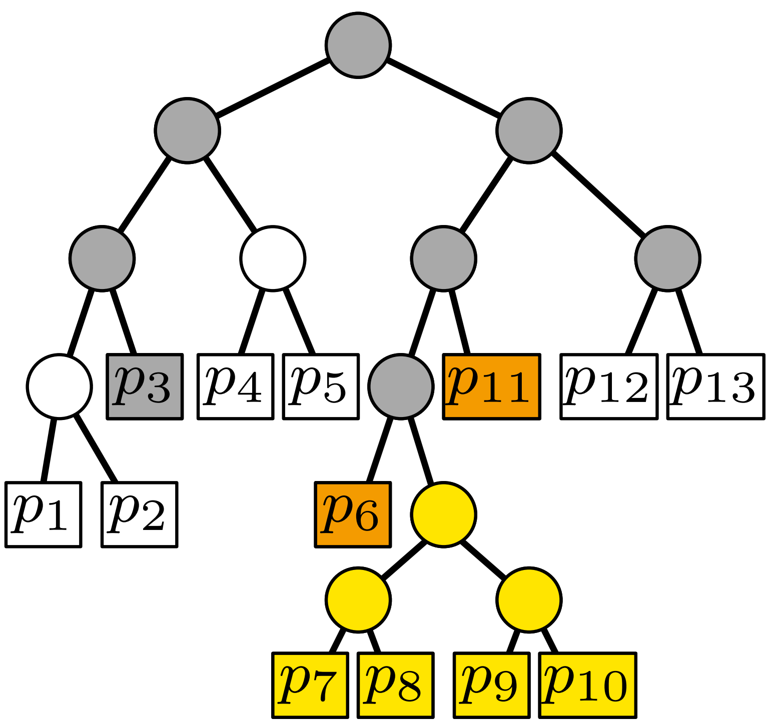


SearchKdTree(v, R)

- **if** v leaf **then**
  - report point p in v when $p \in R$
- **else**
  - **if** region(lc(v)) $\subseteq R$ **then**
    - ReportSubtree(lc(v))
  - **else**
    - **if** region(lc(v))$\cap R \neq \emptyset$ **then**
    - SearchKdTree(lc(v), R)
  - **if** region(rc(v)) $\subseteq R$ **then**
    - ReportSubtree(rc(v))
  - **else**
    - **if** region(rc(v))$\cap R \neq \emptyset$ **then**
    - SearchKdTree(rc(v), R)

Range Queries in a kd-Tree

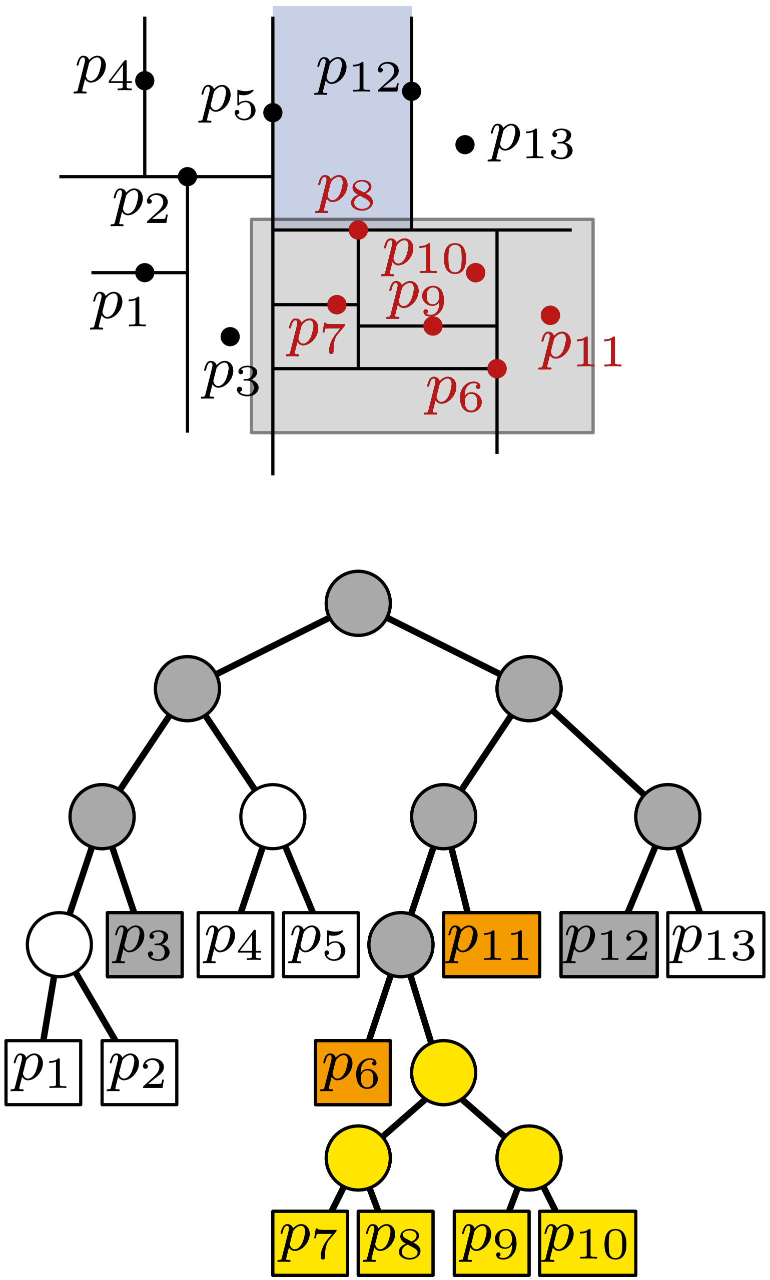




SearchKdTree(v, R)

if v leaf **then**
    report point p in v when $p \in R$
else
    if region(lc(v)) $\subseteq R$ **then**
        ReportSubtree(lc(v))
    else
        if region(lc(v))$\cap R \neq \emptyset$ **then**
        SearchKdTree(lc(v), R)
    if region(rc(v)) $\subseteq R$ **then**
        ReportSubtree(rc(v))
    else
        if region(rc(v))$\cap R \neq \emptyset$ **then**
        SearchKdTree(rc(v), R)

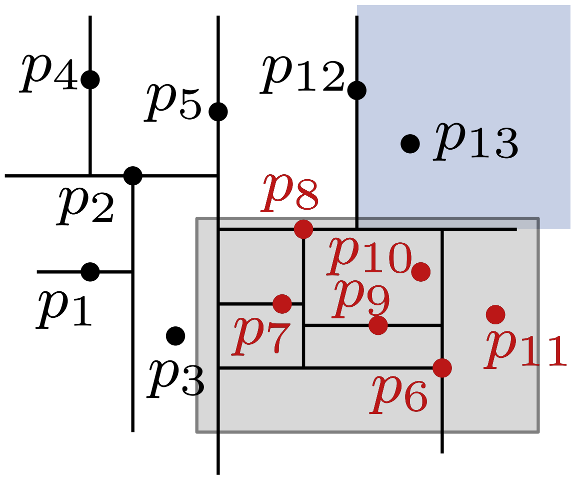
Range Queries in a kd-Tree

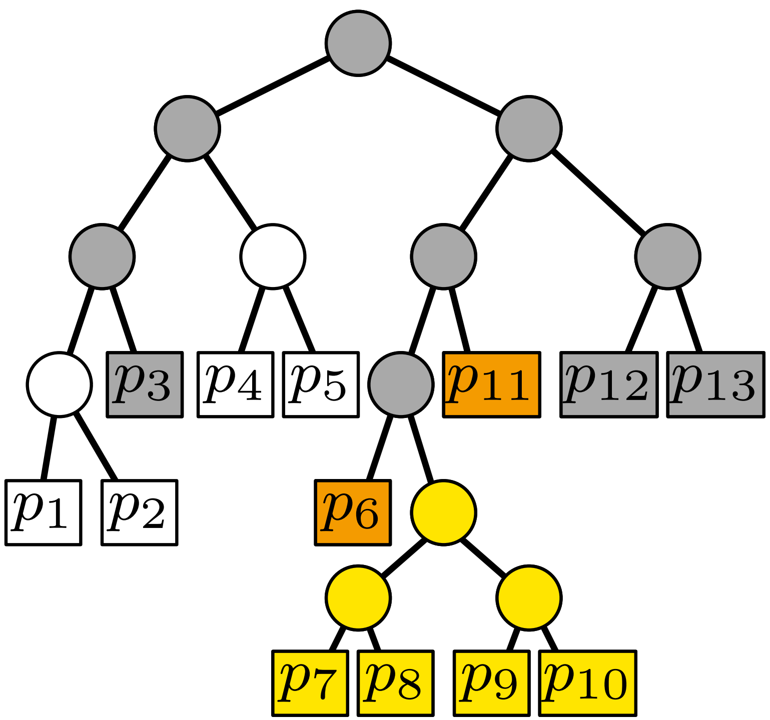


SearchKdTree(v, R)

if v leaf **then**
  report point p in v when $p \in R$
else
  if region(lc(v)) $\subseteq R$ **then**
    ReportSubtree(lc(v))
  else
    if region(lc(v))$\cap R \neq \emptyset$ **then**
    SearchKdTree(lc(v), R)
  if region(rc(v)) $\subseteq R$ **then**
    ReportSubtree(rc(v))
  else
    if region(rc(v))$\cap R \neq \emptyset$ **then**
    SearchKdTree(rc(v), R)

Range Queries in a kd-Tree





SearchKdTree(v, R)

- **if** v leaf **then**
  - report point p in v when $p \in R$
- **else**
  - **if** region(lc(v)) $\subseteq R$ **then**
    - ReportSubtree(lc(v))
  - **else**
    - **if** region(lc(v))$\cap R \neq \emptyset$ **then**
      - SearchKdTree(lc(v), R)
  - **if** region(rc(v)) $\subseteq R$ **then**
    - ReportSubtree(rc(v))
  - **else**
    - **if** region(rc(v))$\cap R \neq \emptyset$ **then**
      - SearchKdTree(rc(v), R)

Analysis of Queries in kd-Trees

Lemma 2: A range query with an axis-aligned rectangle R in a kd-tree on n points may use $O(\sqrt{n} + k)$ time, where k is the number of reported points.

Analysis of Queries in kd-Trees

Lemma 2: A range query with an axis-aligned rectangle R in a kd-tree on n points may use $O(\sqrt{n} + k)$ time, where k is the number of reported points.

Proof sketch:

- Calls to ReportSubtree take $O(k)$ time in total

Analysis of Queries in kd-Trees

Lemma 2: A range query with an axis-aligned rectangle R in a kd-tree on n points may use $O(\sqrt{n} + k)$ time, where k is the number of reported points.

Proof sketch:

- Calls to ReportSubtree take $O(k)$ time in total
- Still missing:
  Number of remaining nodes visited
  $\rightarrow$ Exercise

Orthogonal Range Queries for $d = 2$

Given: Set P of n points in $\mathbb{R}^2$

Goal: A data structure to efficiently answer range queries of the form $R = [x, x'] \times [y, y']$

Ideas for generalizing the 1d case?

Solutions:

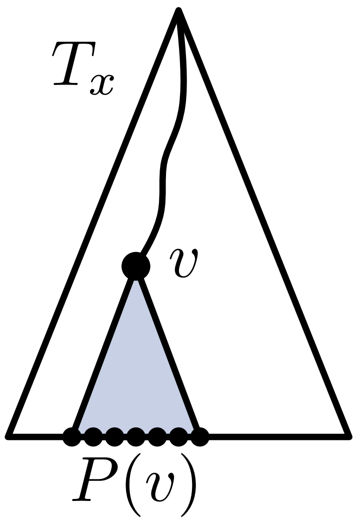
- *one* search tree, alternate search for x and y coordinates
  $\rightarrow$ kd-**Tree** ✓
- *primary* search tree on x-coordinates,
  several *secondary* search trees on y-coordinates
  $\rightarrow$ **Range Tree**

Temporary assumption: general position, that is no two points have the same x- or y-coordinates

Range Trees

Idea: Use 1-dimensional search trees on two levels:

- a 1d search tree T_x on x-coordinates

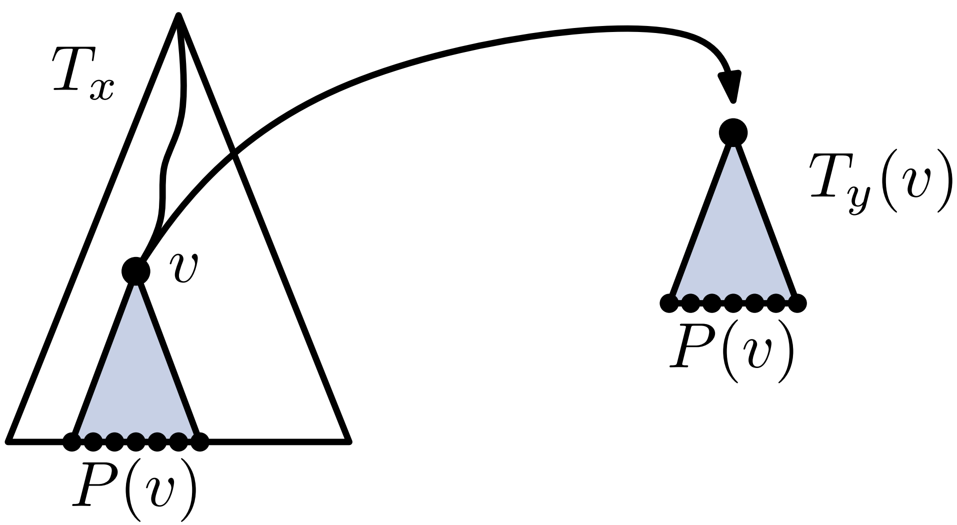


Range Trees

Idea: Use 1-dimensional search trees on two levels:

- a 1d search tree T_x on x-coordinates

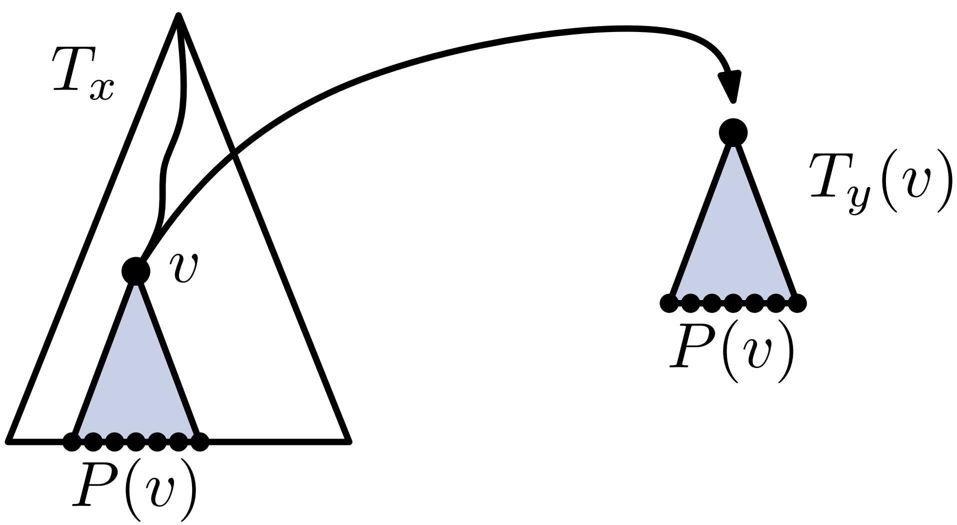
- in each node v of T_x a 1d search tree $T_y(v)$ stores the canonical subset $P(v)$ on y-coordinates



Range Trees

Idea: Use 1-dimensional search trees on two levels:

- a 1d search tree T_x on x-coordinates
- in each node v of T_x a 1d search tree $T_y(v)$ stores the canonical subset $P(v)$ on y-coordinates
- compute the points by x-query in T_x and subsequent y-queries in the auxiliary structures T_y for the subtrees in T_x



Range Trees: Construction

BuildRangeTree(P)

if $|P| = 1$ **then**
    Create leaf v for the point in P
else
    Split P at x_{median} into $P_1 = \{p \in P \mid p_x \leq x_{\text{median}}\}$, $P_2 = P \setminus P_1$
    $v_{\text{left}} \leftarrow$ BuildRangeTree(P_1)
    $v_{\text{right}} \leftarrow$ BuildRangeTree(P_2)
    Create node v with pivot x_{median} and children v_{left} and v_{right}
$T_y(v) \leftarrow$ binary search tree for P w.r.t y-coordinates
return v

Range Trees: Construction

BuildRangeTree(P)

if $|P| = 1$ **then**
  Create leaf v for the point in P
else
  Split P at x_{median} into $P_1 = \{p \in P \mid p_x \leq x_{\text{median}}\}$, $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildRangeTree(P_1)
  $v_{\text{right}} \leftarrow$ BuildRangeTree(P_2)
  Create node v with pivot x_{median} and children v_{left} and v_{right}
$T_y(v) \leftarrow$ binary search tree for P w.r.t y-coordinates
return v

Problem: How much space and runtime does BuildRangeTree use?

Range Trees: Construction

BuildRangeTree(P)

if $|P| = 1$ **then**
  Create leaf v for the point in P
else
  Split P at x_{median} into $P_1 = \{p \in P \mid p_x \leq x_{\text{median}}\}$, $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildRangeTree(P_1)
  $v_{\text{right}} \leftarrow$ BuildRangeTree(P_2)
  Create node v with pivot x_{median} and children v_{left} and v_{right}
$T_y(v) \leftarrow$ binary search tree for P w.r.t y-coordinates
return v

Problem: How much space and runtime does BuildRangeTree use?

Lemma 3: A Range Tree for n points in $\mathbb{R}^2$ uses $O(n \log n)$ space and can be constructed in $O(n \log n)$ time.

Range Queries in a Range Tree

Reminder:

1dRangeQuery(T, x, x')

```
v_split ← FindSplitNode(T, x, x')
if v_split is leaf then report v_split
else
    v ← lc(v_split)
    while v not leaf do
        if x ≤ x_v then
            ReportSubtree(rc(v))
            v ← lc(v)
        else  v ← rc(v)
    report v
    // analogous for x' and rc(v_split)
```

Range Queries in a Range Tree

Reminder:

~~**1dRangeQuery**(T, x, x')~~ **2dRangeQuery**$(T, [x, x'] \times [y, y'])$

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** report v_{split}
else
  $v \leftarrow \text{lc}(v_{\text{split}})$
  while v not leaf **do**
    if $x \leq x_v$ **then**
      ~~ReportSubtree(rc(v))~~ 1dRangeQuery$(T_y(\text{rc}(v)), y, y')$
      $v \leftarrow \text{lc}(v)$
    else $v \leftarrow \text{rc}(v)$
  report v
  `// analogous for` x' `and rc(`v_{split}`)`

Range Queries in a Range Tree

Reminder:

~~**1dRangeQuery**(T, x, x')~~ **2dRangeQuery**$(T, [x, x'] \times [y, y'])$

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** report v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not leaf **do**
    if $x \leq x_v$ **then**
      ~~ReportSubtree(rc(v))~~ 1dRangeQuery$(T_y(\text{rc}(v)), y, y')$
      $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  report v
  `// analogous for` x' `and rc(`v_{split}`)`

Lemma 4: A range query in a Range Tree takes $O(\log^2 n + k)$ time, where k is the number of reported points.

Range Queries with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Range Queries with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Range Queries with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Example: Two sets $B \subseteq A \subseteq \mathbb{R}$ in sorted arrays

A	3	10	19	23	30	37	59	62	70	80	100	105

B	10	19	30	62	70	80	100

Range Queries with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Example: Two sets $B \subseteq A \subseteq \mathbb{R}$ in sorted arrays

A	3	10	19	23	30	37	59	62	70	80	100	105

B	10	19	30	62	70	80	100

Search interval [20,65]

Range Queries with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Example: Two sets $B \subseteq A \subseteq \mathbb{R}$ in sorted arrays

A	3	10	19	23	30	37	59	62	70	80	100	105

Can we do better than two binary searches?

B	10	19	30	62	70	80	100

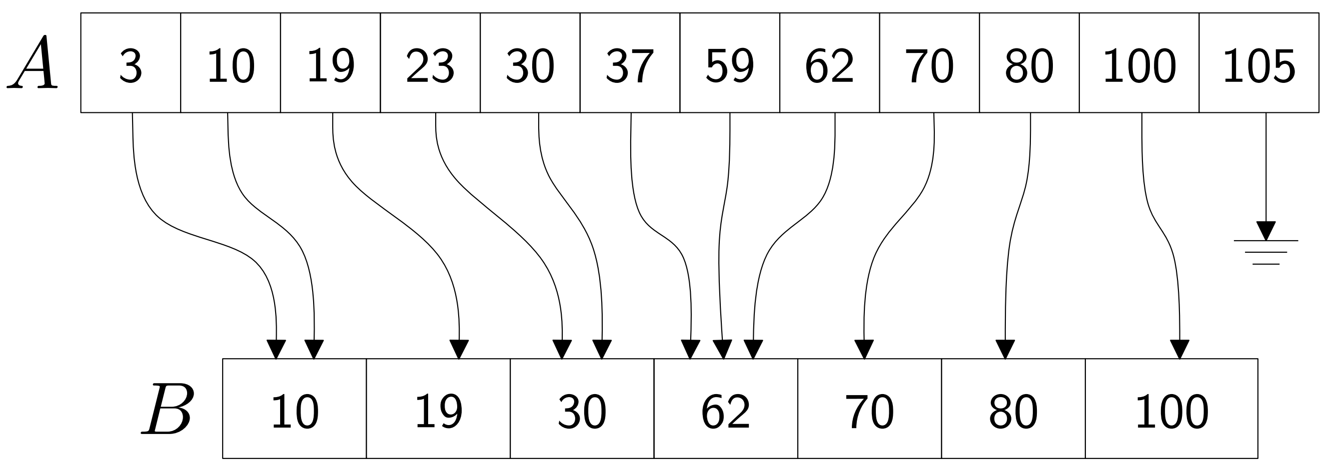
Search interval [20,65]

Range Queries with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Example: Two sets $B \subseteq A \subseteq \mathbb{R}$ in sorted arrays



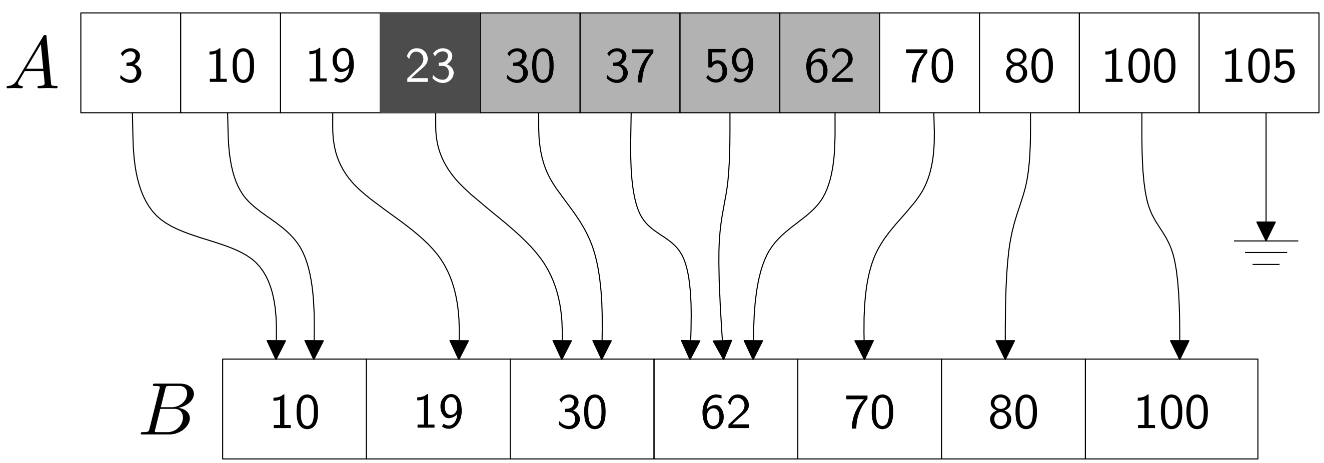
link $a \in A$ with smallest $b \geq a$ in B

Range Queries with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Example: Two sets $B \subseteq A \subseteq \mathbb{R}$ in sorted arrays



link $a \in A$ with smallest $b \geq a$ in B

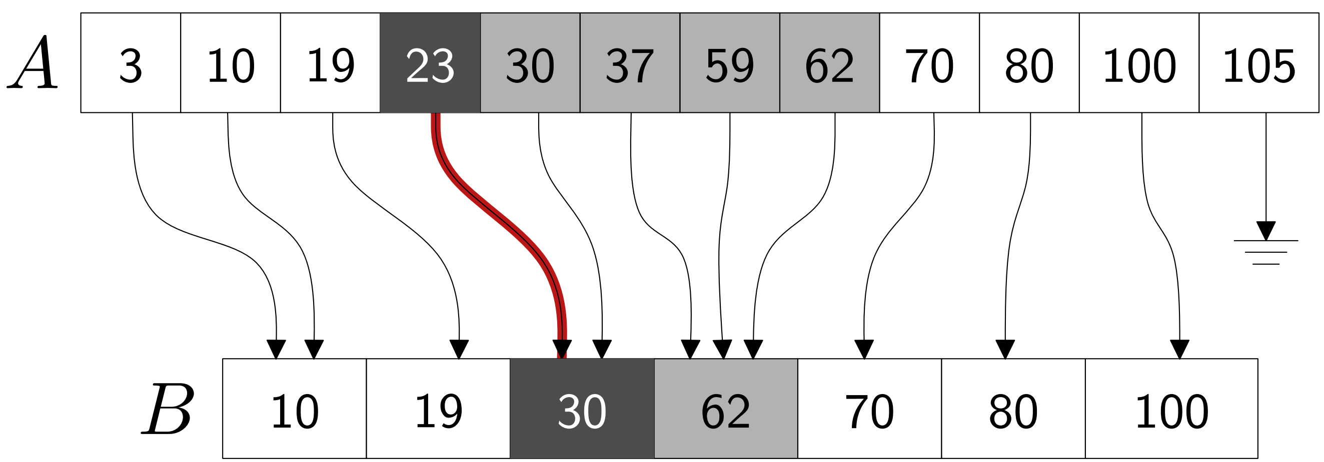
Search interval [20,65]

Range Queries with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Example: Two sets $B \subseteq A \subseteq \mathbb{R}$ in sorted arrays



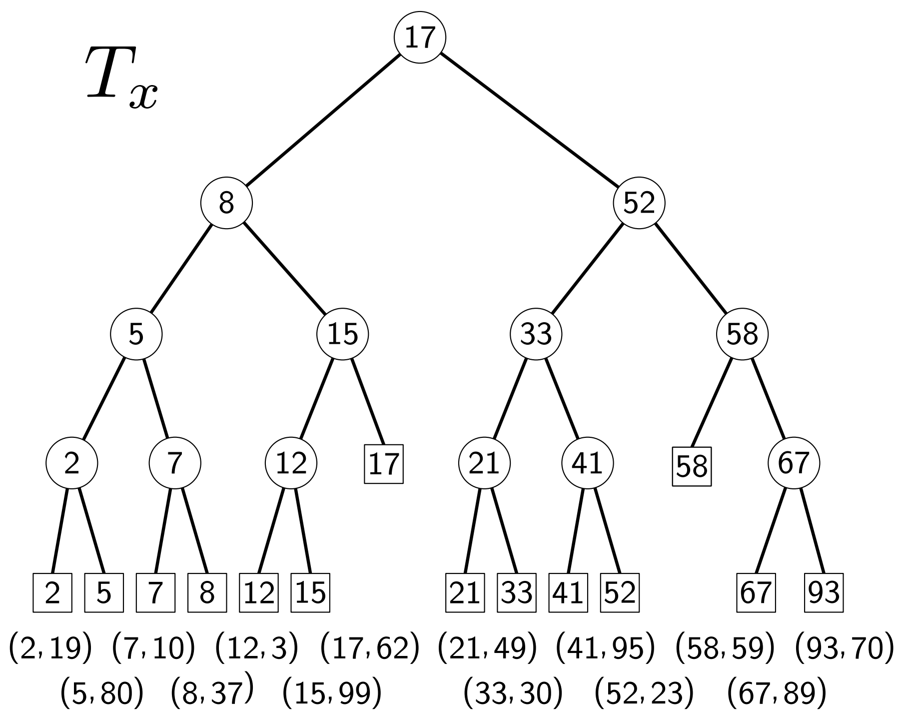
link $a \in A$ with smallest $b \geq a$ in B

Search interval [20,65]

Pointer yields starting point for second search in $O(1)$ time

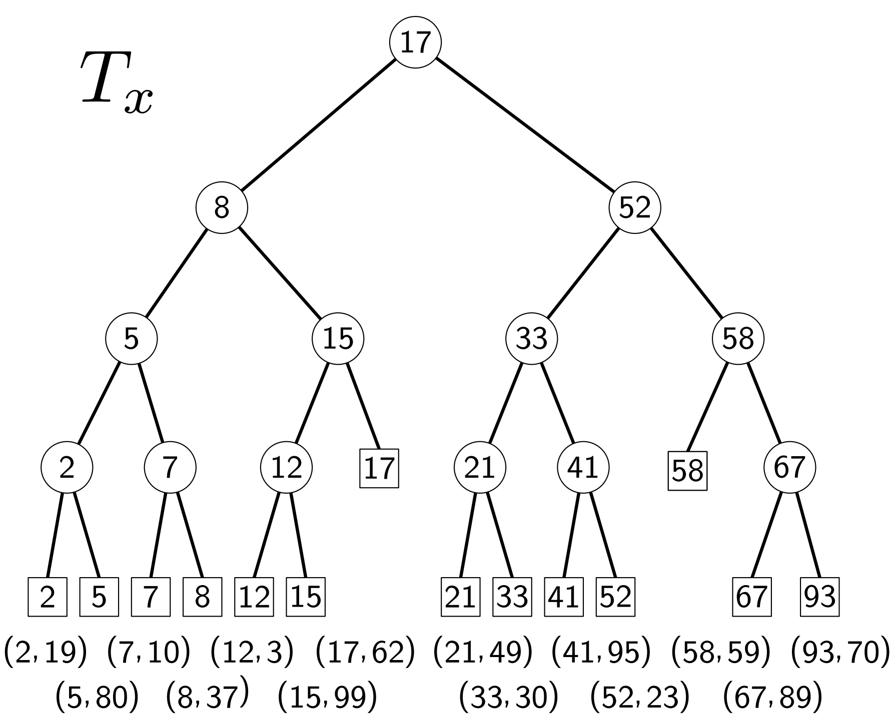
Speed-up with Fractional Cascading

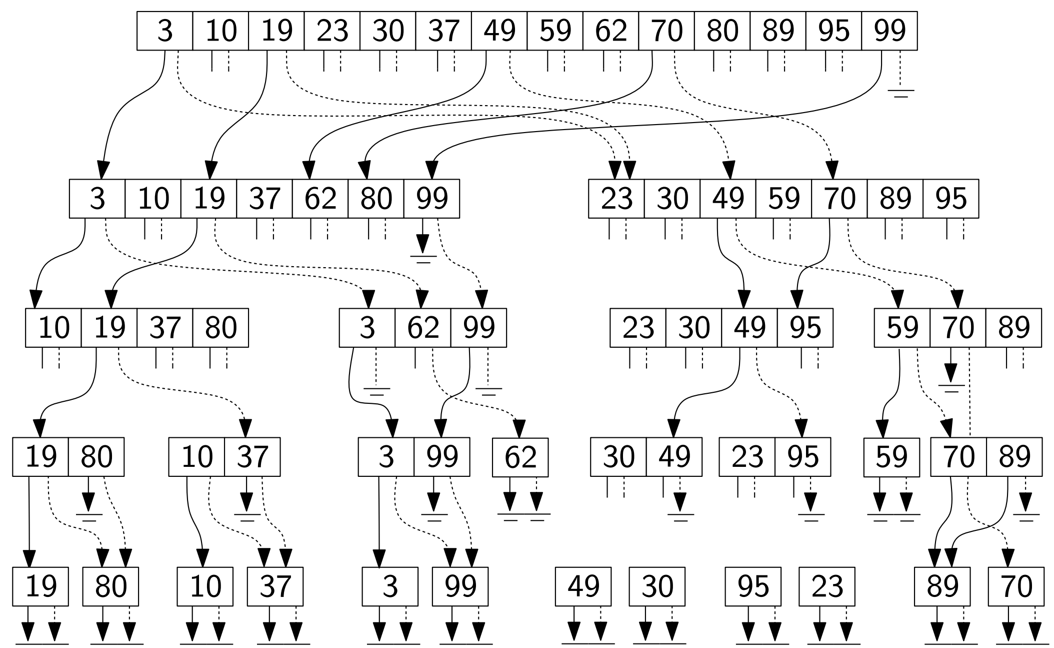
- In Range Trees we have $P(\mathrm{lc}(v)) \subseteq P(v)$ and $P(\mathrm{rc}(v)) \subseteq P(v)$ as the canonical sets.



Speed-up with Fractional Cascading

- In Range Trees we have $P(\mathrm{lc}(v)) \subseteq P(v)$ and $P(\mathrm{rc}(v)) \subseteq P(v)$ as the canonical sets.
- Define for each array element $A(v)[i]$ two pointers into the arrays $A(\mathrm{lc}(v))$ and $A(\mathrm{rc}(v))$
  $\rightarrow$ **Layered Range Tree**

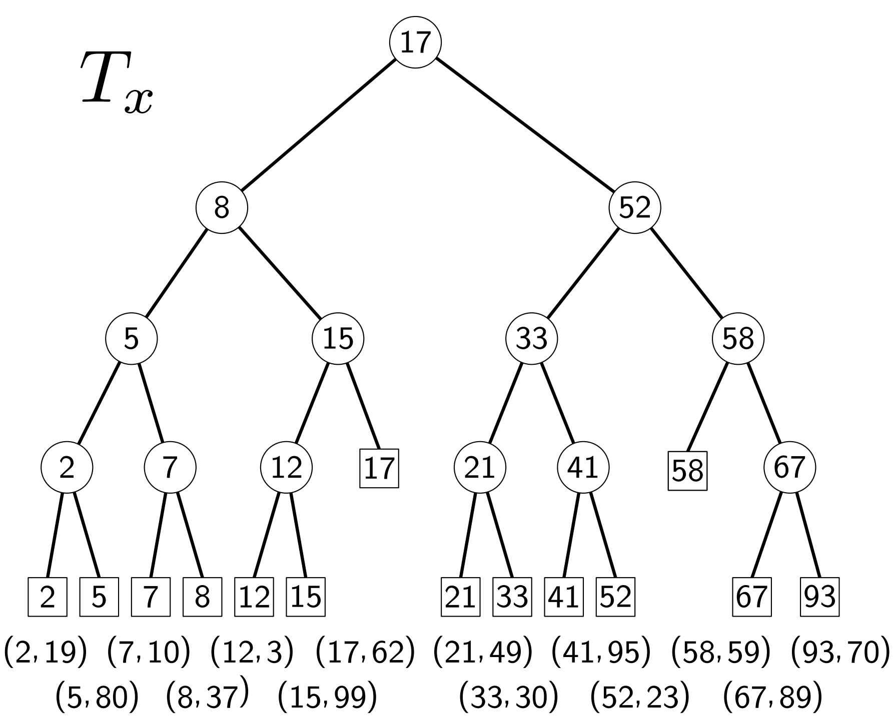


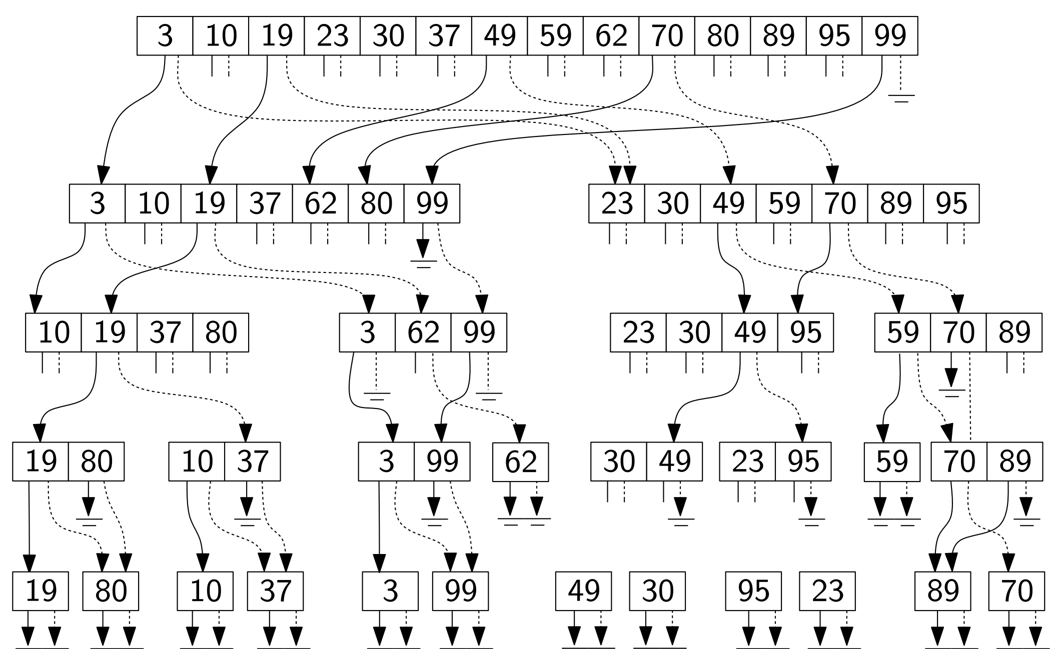


Speed-up with Fractional Cascading

Karlsruhe Institute of Technology

- In Range Trees we have $P(\mathrm{lc}(v)) \subseteq P(v)$ and $P(\mathrm{rc}(v)) \subseteq P(v)$ as the canonical sets.
- Define for each array element $A(v)[i]$ two pointers into the arrays $A(\mathrm{lc}(v))$ and $A(\mathrm{rc}(v))$
  $\rightarrow$ **Layered Range Tree**
- In the split node a binary search takes $O(\log n)$ time, then it takes $O(1)$ time to follow the pointers in the children





Speed-up with Fractional Cascading

- In Range Trees we have $P(\mathsf{lc}(v)) \subseteq P(v)$ and $P(\mathsf{rc}(v)) \subseteq P(v)$ as the canonical sets.
- Define for each array element $A(v)[i]$ two pointers into the arrays $A(\mathsf{lc}(v))$ and $A(\mathsf{rc}(v))$
  $\rightarrow$ **Layered Range Tree**
- In the split node a binary search takes $O(\log n)$ time, then it takes $O(1)$ time to follow the pointers in the children

Theorem 2: A Layered Range Tree on n points in $\mathbb{R}^2$ can be constructed in $O(n \log n)$ time and space. Range queries take $O(\log n + k)$ time, where k is the number of reported points.

Arbitrary Point Sets

So far: Points in general position, where no two points have the same x- or y-coordinate

Idea: Instead of $\mathbb{R}$, use pairs of numbers $(a|b)$ with total order $\leftrightarrow$ lexicographic order

Arbitrary Point Sets

So far: Points in general position, where no two points have the same x- or y-coordinate

Idea: Instead of $\mathbb{R}$, use pairs of numbers $(a|b)$ with total order $\leftrightarrow$ lexicographic order

$$p = (p_x, p_y)$$

Arbitrary Point Sets

So far: Points in general position, where no two points have the same x- or y-coordinate

Idea: Instead of $\mathbb{R}$, use pairs of numbers $(a|b)$ with total order $\leftrightarrow$ lexicographic order

$$p = (p_x, p_y) \longrightarrow \hat{p} = \big((p_x|p_y),\ (p_y|p_x)\big)$$

Arbitrary Point Sets

So far: Points in general position, where no two points have the same x- or y-coordinate

Idea: Instead of $\mathbb{R}$, use pairs of numbers $(a|b)$ with total order $\leftrightarrow$ lexicographic order

$$p = (p_x, p_y) \longrightarrow \hat{p} = \big((p_x|p_y),\ (p_y|p_x)\big)$$

unique coord.

Arbitrary Point Sets

So far: Points in general position, where no two points have the same x- or y-coordinate

Idea: Instead of $\mathbb{R}$, use pairs of numbers $(a|b)$ with total order $\leftrightarrow$ lexicographic order

$$p = (p_x, p_y) \rightarrow \hat{p} = \big((p_x|p_y),\ (p_y|p_x)\big)$$

unique coord.

Rectangle $R = [x, x'] \times [y, y']$

Arbitrary Point Sets

So far: Points in general position, where no two points have the same x- or y-coordinate

Idea: Instead of $\mathbb{R}$, use pairs of numbers $(a|b)$ with total order $\leftrightarrow$ lexicographic order

$$p = (p_x, p_y) \longrightarrow \hat{p} = \big((p_x|p_y),\ (p_y|p_x)\big)$$

unique coord.

Rectangle $R = [x, x'] \times [y, y']$

Arbitrary Point Sets

So far: Points in general position, where no two points have the same x- or y-coordinate

Idea: Instead of $\mathbb{R}$, use pairs of numbers $(a|b)$ with total order $\leftrightarrow$ lexicographic order

$$p = (p_x, p_y) \longrightarrow \hat{p} = \big((p_x|p_y),\ (p_y|p_x)\big)$$

unique coord.

Rectangle $R = [x, x'] \times [y, y']$

$$\downarrow$$

$$\hat{R} = [(x| - \infty), (x'| + \infty)] \times [(y| - \infty), (y'| + \infty)]$$

Arbitrary Point Sets

So far: Points in general position, where no two points have the same x- or y-coordinate

Idea: Instead of $\mathbb{R}$, use pairs of numbers $(a|b)$ with total order $\leftrightarrow$ lexicographic order

$$p = (p_x, p_y) \longrightarrow \hat{p} = \big((p_x|p_y),\ (p_y|p_x)\big)$$

unique coord.

Rectangle $R = [x, x'] \times [y, y']$

$$\downarrow$$

$$\hat{R} = [(x|-\infty), (x'|+\infty)] \times [(y|-\infty), (y'|+\infty)]$$

Then:

Arbitrary Point Sets

So far: Points in general position, where no two points have the same x- or y-coordinate

Idea: Instead of $\mathbb{R}$, use pairs of numbers $(a|b)$ with total order $\leftrightarrow$ lexicographic order

$$p = (p_x, p_y) \longrightarrow \hat{p} = \big((p_x|p_y),\ (p_y|p_x)\big)$$

unique coord.

Rectangle $R = [x, x'] \times [y, y']$

$$\downarrow$$

$$\hat{R} = [(x|-\infty), (x'|+\infty)] \times [(y|-\infty), (y'|+\infty)]$$

Then: $p \in R \ \Leftrightarrow\ \hat{p} \in \hat{R}$

Summary

Given: Set P of n points in $\mathbb{R}^2$

Construct: Data structures with efficient range queries of the form $R = [x, x'] \times [y, y']$

$\rightarrow$ We have seen two alternatives

	kd-Tree	Range Tree
Preprocessing		
Space		
Query time		

Summary

Given: Set P of n points in $\mathbb{R}^2$

Construct: Data structures with efficient range queries of the form $R = [x, x'] \times [y, y']$

$\rightarrow$ We have seen two alternatives

	kd-Tree	Range Tree
Preprocessing	$O(n \log n)$	$O(n \log n)$
Space	$O(n)$	$O(n \log n)$
Query time	$O(\sqrt{n} + k)$	$O(\log^2 n + k)$

Summary

Given: Set P of n points in $\mathbb{R}^2$

Construct: Data structures with efficient range queries of the form $R = [x, x'] \times [y, y']$

$\rightarrow$ We have seen two alternatives

	kd-Tree	Range Tree
Preprocessing	$O(n \log n)$	$O(n \log n)$
Space	$O(n)$	$O(n \log n)$
Query time	$O(\sqrt{n} + k)$	$O(\log^{\cancel{2}} n + k)$

Summary

Given: Set P of n points in $\mathbb{R}^2$

Construct: Data structures with efficient range queries of the form $R = [x, x'] \times [y, y']$

$\rightarrow$ We have seen two alternatives

	kd-Tree	Range Tree
Preprocessing	$O(n \log n)$	$O(n \log n)$
Space	$O(n)$	$O(n \log n)$
Query time	$O(\sqrt{n} + k)$	$O(\log^{\cancel{2}} n + k)$

Discussion

How can the data structures generalize to d-dimensions?

Discussion

How can the data structures generalize to d-dimensions?

- kd-Trees function analogously and by dividing the points alternately on d coordinates. Space is still $O(n)$, construction $O(n \log n)$ and the query time is $O(n^{1-1/d} + k)$.

Discussion

How can the data structures generalize to d-dimensions?

- kd-Trees function analogously and by dividing the points alternately on d coordinates. Space is still $O(n)$, construction $O(n \log n)$ and the query time is $O(n^{1-1/d} + k)$.
- Range Trees can be built recursively: the auxiliary search tree on the first coordinate is a $(d-1)$-dimensional Range Tree. The construction and space takes $O(n \log^{d-1} n)$ time; a query takes $O(\log^d n + k)$ time, and with fractional cascading, $O(\log^{d-1} n + k)$ time.

Discussion

How can the data structures generalize to d-dimensions?

- kd-Trees function analogously and by dividing the points alternately on d coordinates. Space is still $O(n)$, construction $O(n \log n)$ and the query time is $O(n^{1-1/d} + k)$.
- Range Trees can be built recursively: the auxiliary search tree on the first coordinate is a $(d-1)$-dimensional Range Tree. The construction and space takes $O(n \log^{d-1} n)$ time; a query takes $O(\log^d n + k)$ time, and with fractional cascading, $O(\log^{d-1} n + k)$ time.

Is it possible to query for other objects (e.g., polygons) with these data structures?

Discussion

How can the data structures generalize to d-dimensions?

- kd-Trees function analogously and by dividing the points alternately on d coordinates. Space is still $O(n)$, construction $O(n \log n)$ and the query time is $O(n^{1-1/d} + k)$.
- Range Trees can be built recursively: the auxiliary search tree on the first coordinate is a $(d-1)$-dimensional Range Tree. The construction and space takes $O(n \log^{d-1} n)$ time; a query takes $O(\log^d n + k)$ time, and with fractional cascading, $O(\log^{d-1} n + k)$ time.

Is it possible to query for other objects (e.g., polygons) with these data structures?

Yes, we can transform any polygon into a point in 4d space (exercise) or we can use windowing queries (comes in a later lecture).

Dynamic Range Queries

Question: Can we adapt these data structures for dynamic point sets?

- Inserting points
- Removing points

Dynamic Range Queries

Question: Can we adapt these data structures for dynamic point sets?

- Inserting points
- Removing points

1) **Divided kd-trees** [van Kreveld, Overmars '91]
support updates in $O(\log n)$ time, but the query time is
$O(\sqrt{n \log n} + k)$

Dynamic Range Queries

Question: Can we adapt these data structures for dynamic point sets?

- Inserting points
- Removing points

1) **Divided kd-trees** [van Kreveld, Overmars '91]
support updates in $O(\log n)$ time, but the query time is $O(\sqrt{n \log n} + k)$

2) **Augmented dynamic range trees** [Mehlhorn, Näher '90]
support updates in $O(\log n \log\log n)$ time and queries in $O(\log n \log\log n + k)$ time