Vorlesung Algorithmische Geometrie

Bereichsabfragen

INSTITUT FÜR THEORETISCHE INFORMATIK · FAKULTÄT FÜR INFORMATIK

Martin Nöllenburg

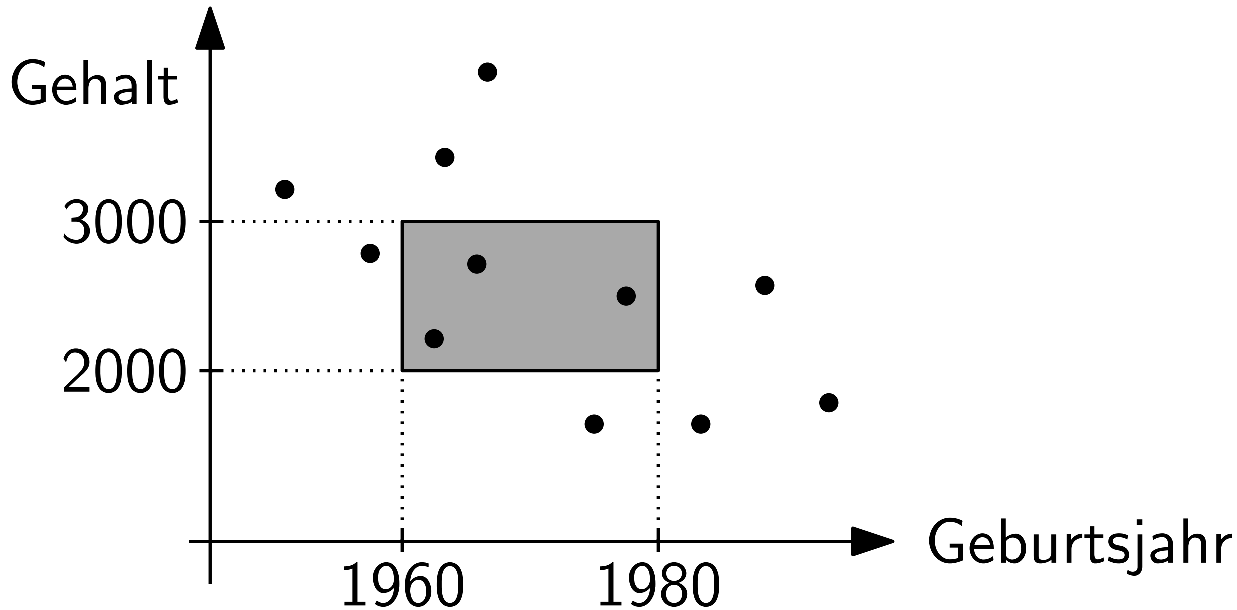
15.05.2012

Geometrie in Datenbanken

In einer Personaldatenbank werden die Mitarbeiter einer Firma erfasst und u.a. die Attribute Monatseinkommen und Geburtsjahr gespeichert. Es sollen nun alle Mitarbeiter mit einem Einkommen zwischen 2000 und 3000 EUR, die zwischen 1960 und 1980 geboren sind gesucht werden.

Geometrie in Datenbanken

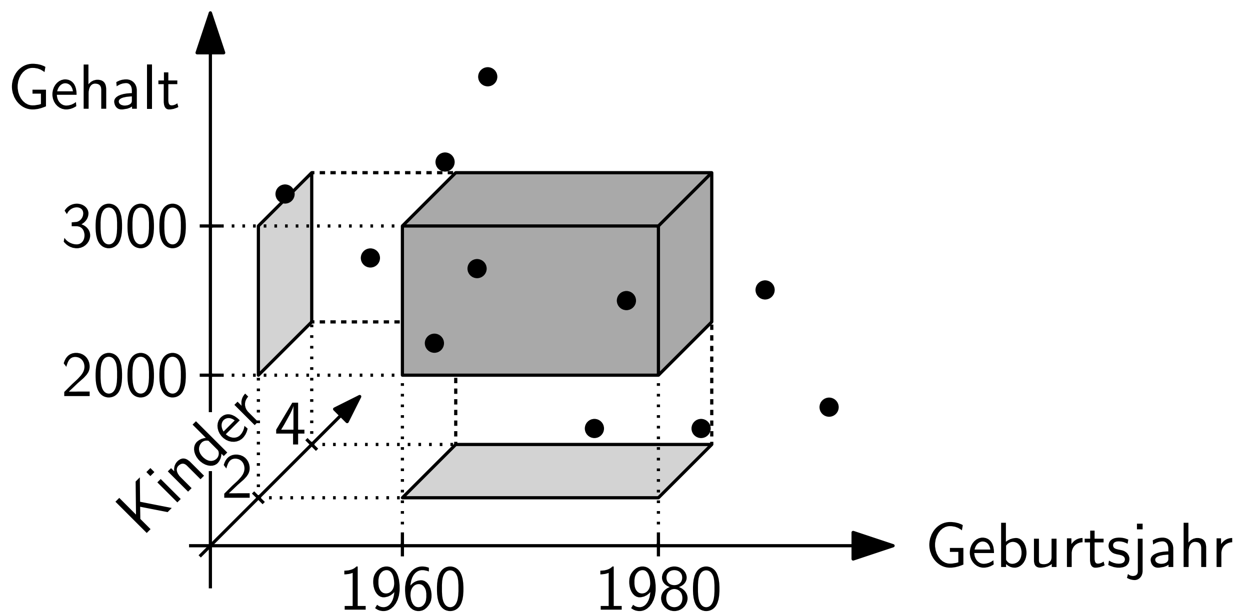
In einer Personaldatenbank werden die Mitarbeiter einer Firma erfasst und u.a. die Attribute Monatseinkommen und Geburtsjahr gespeichert. Es sollen nun alle Mitarbeiter mit einem Einkommen zwischen 2000 und 3000 EUR, die zwischen 1960 und 1980 geboren sind gesucht werden.



Geometrische Interpretation:
Einträge sind Punkte in der Gehalt-Geburtsjahr-Ebene und die Anfrage ist ein achsenparalleles Rechteck.

Geometrie in Datenbanken

In einer Personaldatenbank werden die Mitarbeiter einer Firma erfasst und u.a. die Attribute Monatseinkommen und Geburtsjahr gespeichert. Es sollen nun alle Mitarbeiter mit einem Einkommen zwischen 2000 und 3000 EUR, die zwischen 1960 und 1980 geboren sind gesucht werden.



Lässt sich problemlos auf d Dimensionen verallgemeinern.

Orthogonale Bereichsabfragen

Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Orthogonale Bereichsabfragen

Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Orthogonale Bereichsabfragen

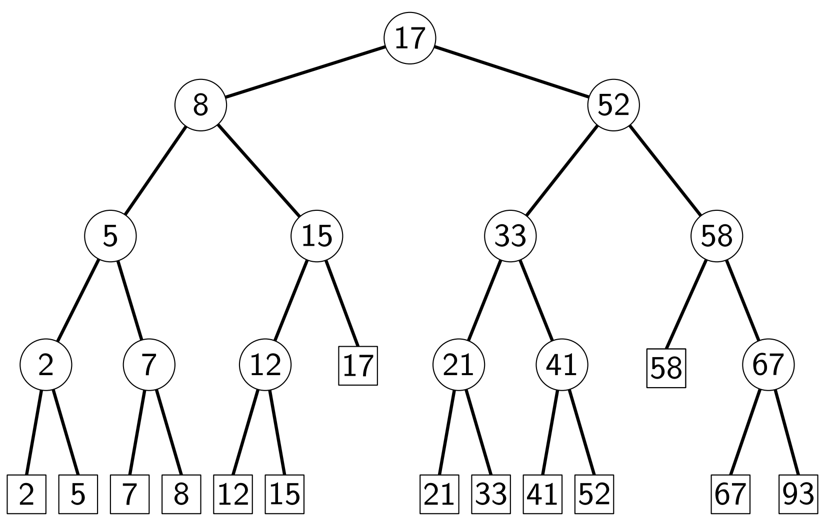
Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Lösung: balancierter binärer Suchbaum:

- Blätter speichern die Eingabepunkte
- innerer Knoten v speichert Pivotwert x_v



Orthogonale Bereichsabfragen

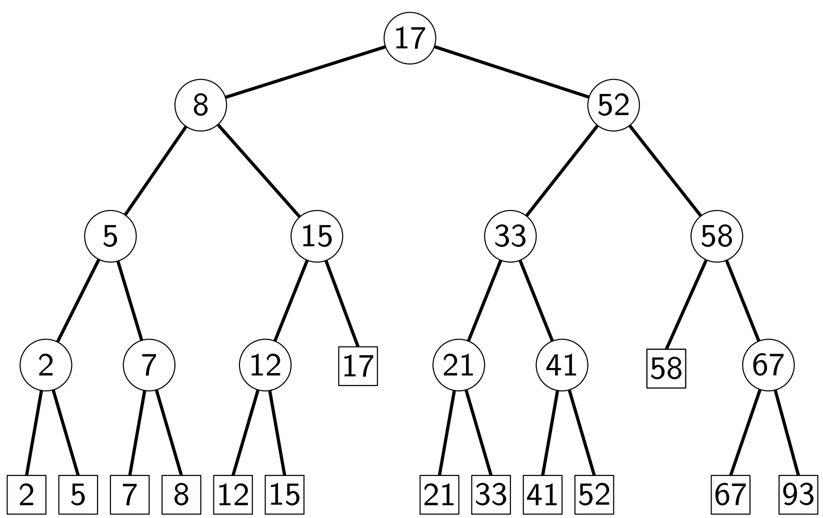
Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Lösung: balancierter binärer Suchbaum:

- Blätter speichern die Eingabepunkte
- innerer Knoten v speichert Pivotwert x_v



Beispiel:

Suche alle Punkte in [6,50]

Orthogonale Bereichsabfragen

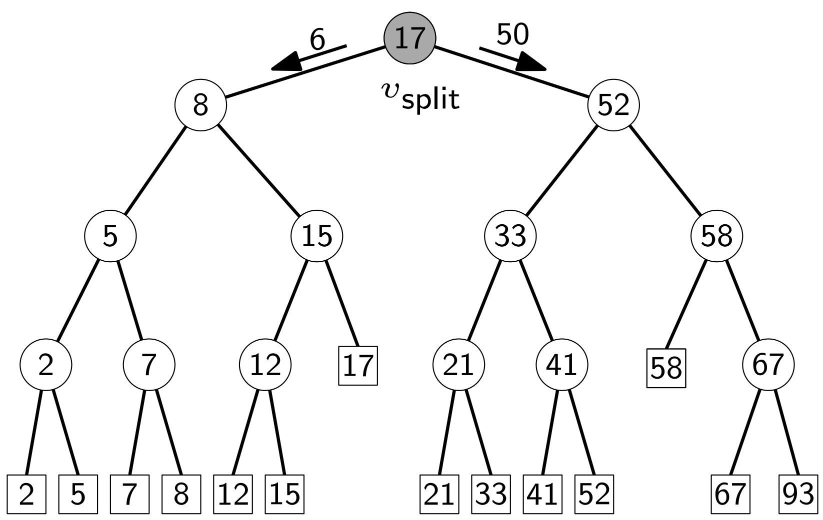
Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Lösung: balancierter binärer Suchbaum:

- Blätter speichern die Eingabepunkte
- innerer Knoten v speichert Pivotwert x_v



Beispiel:

Suche alle Punkte in [6,50]

Orthogonale Bereichsabfragen

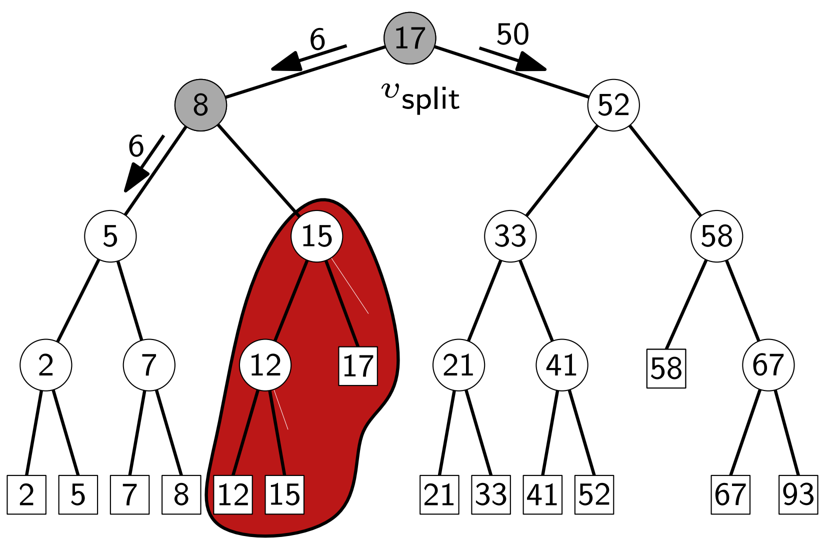
Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Lösung: balancierter binärer Suchbaum:

- Blätter speichern die Eingabepunkte
- innerer Knoten v speichert Pivotwert x_v



Beispiel:
Suche alle Punkte in [6,50]

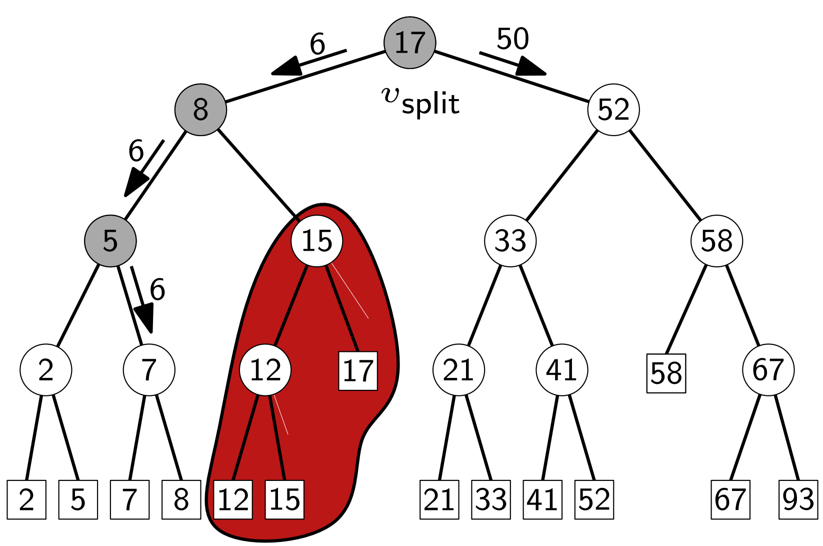
Orthogonale Bereichsabfragen

Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Lösung: balancierter binärer Suchbaum:
- Blätter speichern die Eingabepunkte
- innerer Knoten v speichert Pivotwert x_v



Beispiel:
Suche alle Punkte in [6,50]

Orthogonale Bereichsabfragen

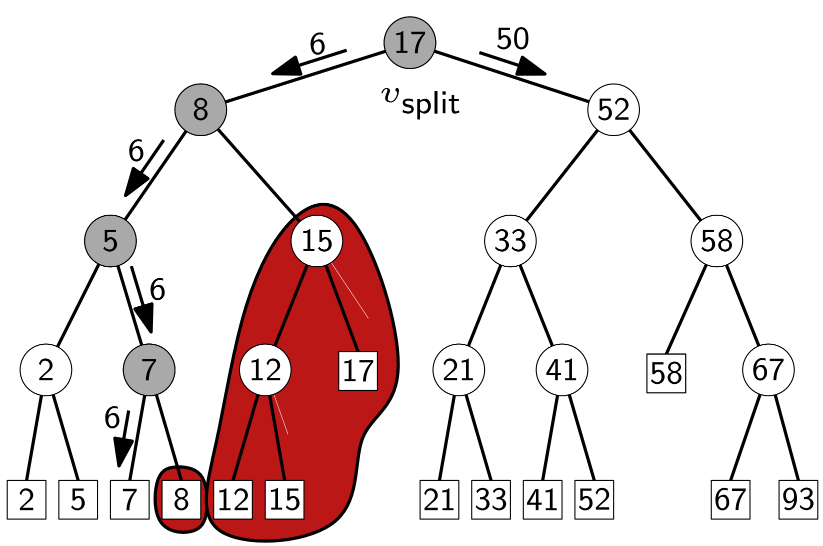
Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Lösung: balancierter binärer Suchbaum:

- Blätter speichern die Eingabepunkte
- innerer Knoten v speichert Pivotwert x_v



Beispiel:

Suche alle Punkte in [6,50]

Orthogonale Bereichsabfragen

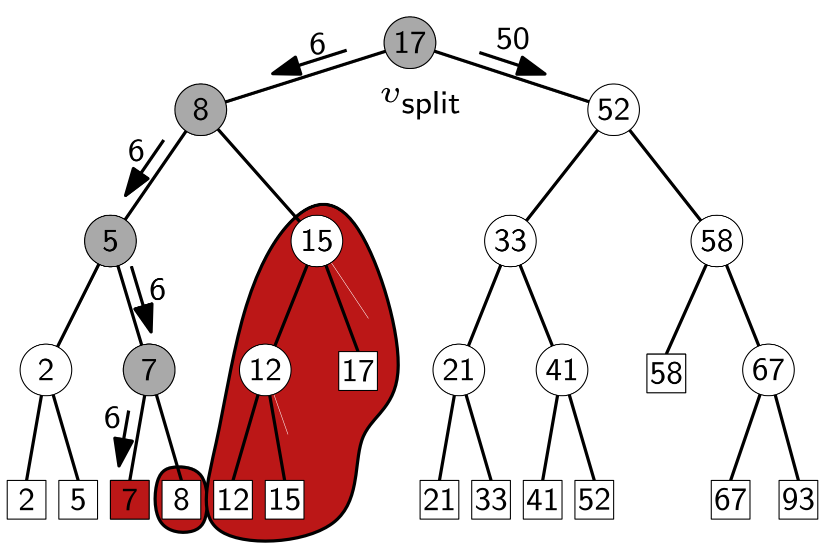
Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Lösung: balancierter binärer Suchbaum:

- Blätter speichern die Eingabepunkte
- innerer Knoten v speichert Pivotwert x_v



Beispiel:

Suche alle Punkte in [6,50]

Orthogonale Bereichsabfragen

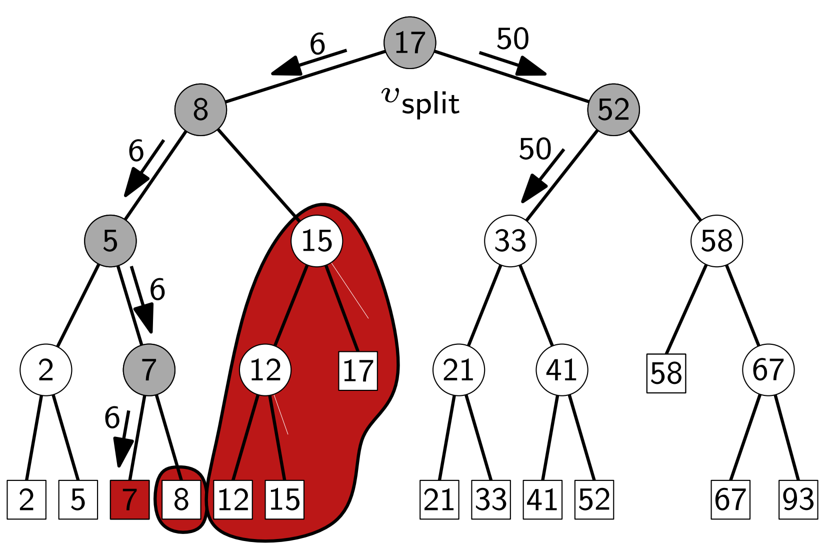
Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Lösung: balancierter binärer Suchbaum:

- Blätter speichern die Eingabepunkte
- innerer Knoten v speichert Pivotwert x_v



Beispiel:

Suche alle Punkte in [6,50]

Orthogonale Bereichsabfragen

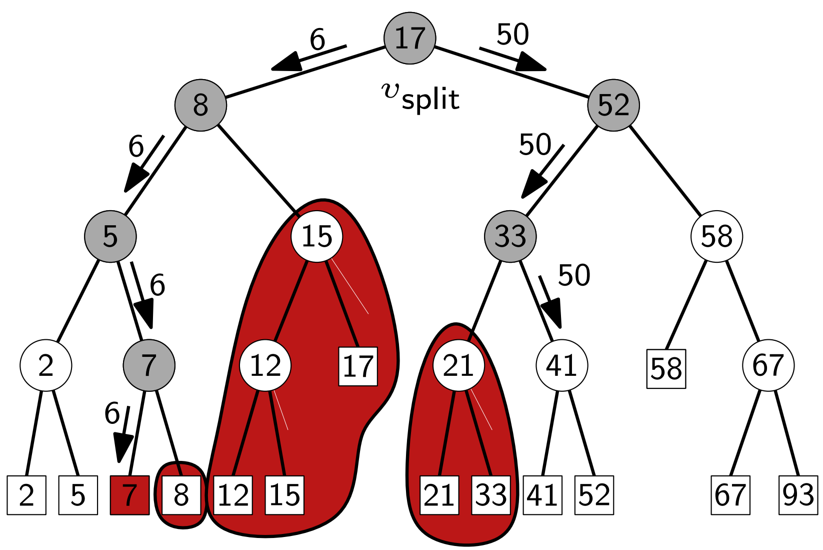
Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Lösung: balancierter binärer Suchbaum:

- Blätter speichern die Eingabepunkte
- innerer Knoten v speichert Pivotwert x_v



Beispiel:
Suche alle Punkte in [6,50]

Orthogonale Bereichsabfragen

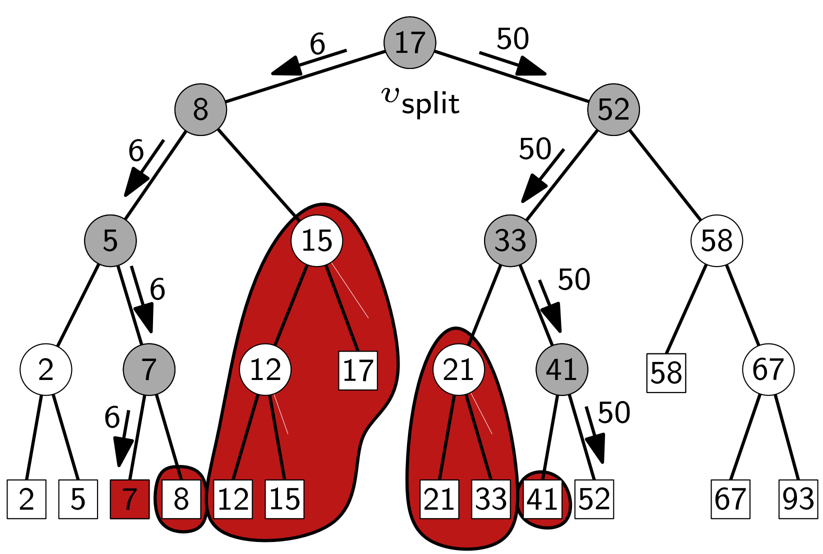
Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Lösung: balancierter binärer Suchbaum:

- Blätter speichern die Eingabepunkte
- innerer Knoten v speichert Pivotwert x_v



Beispiel:

Suche alle Punkte in [6,50]

Orthogonale Bereichsabfragen

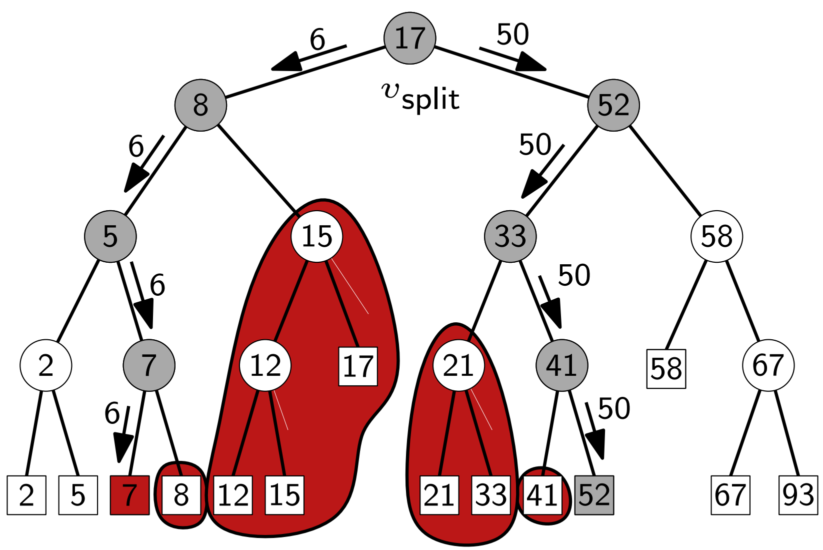
Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Lösung: balancierter binärer Suchbaum:

- Blätter speichern die Eingabepunkte
- innerer Knoten v speichert Pivotwert x_v



Beispiel:

Suche alle Punkte in [6,50]

Orthogonale Bereichsabfragen

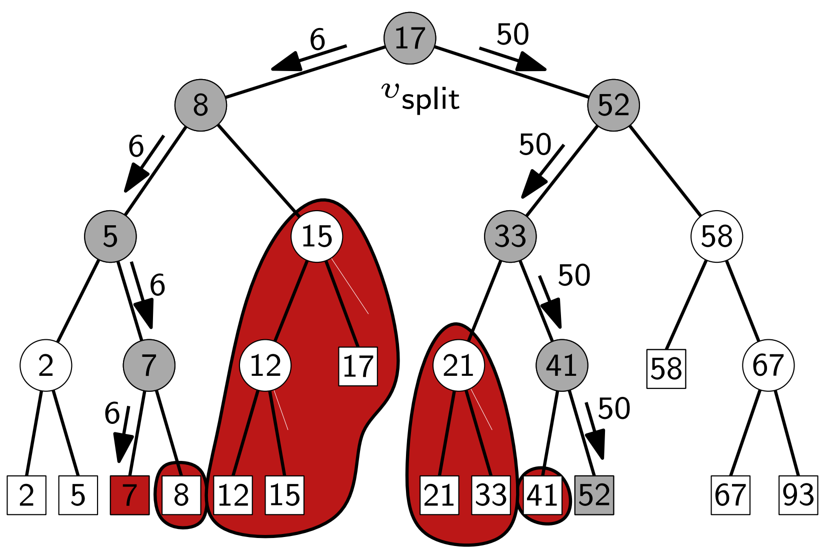
Geg: n Punkte im $\mathbb{R}^d$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Aufgabe: Entwerfen Sie eine Datenstruktur für den Fall $d = 1$.

Lösung: balancierter binärer Suchbaum:

- Blätter speichern die Eingabepunkte
- innerer Knoten v speichert Pivotwert x_v



Beispiel:
Suche alle Punkte in [6,50]

Antwort:
Blätter der Teilbäume zwischen den beiden Suchpfaden, d.h. {7,8,12,15,17,21,33,41}

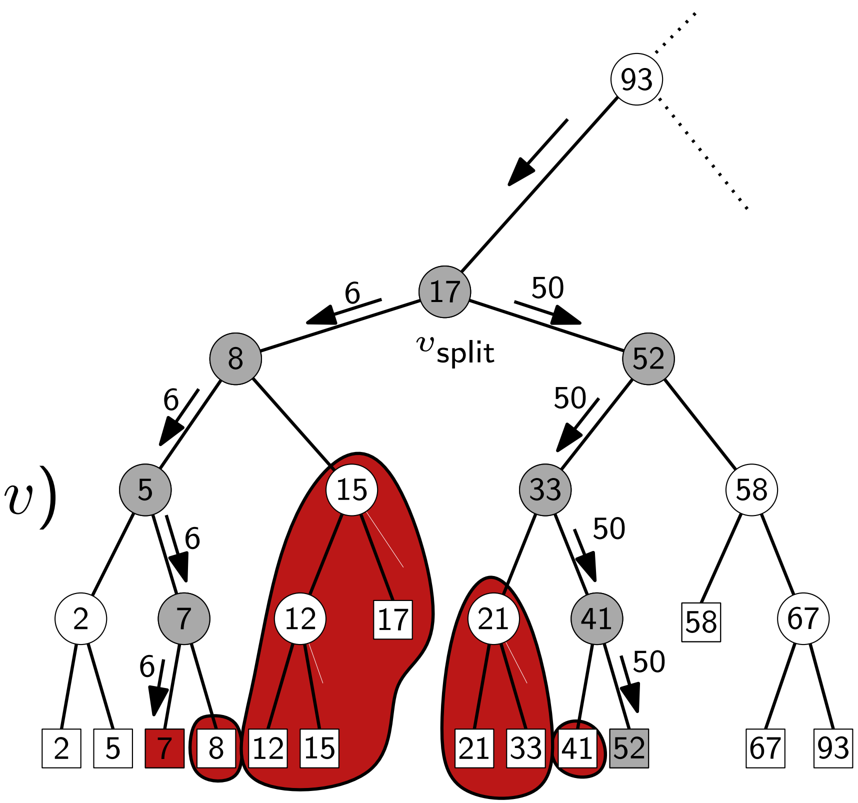
1dRangeQuery

FindSplitNode(T, x, x')

$v \leftarrow$ root(T)
while v kein Blatt und $(x' \leq x_v$ oder $x > x_v)$ **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} ist Blatt **then** prüfe v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v kein Blatt **do**
    if $x \leq x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  prüfe v
  `// analog für` x' `und rc(`v_{split}`)`



1dRangeQuery

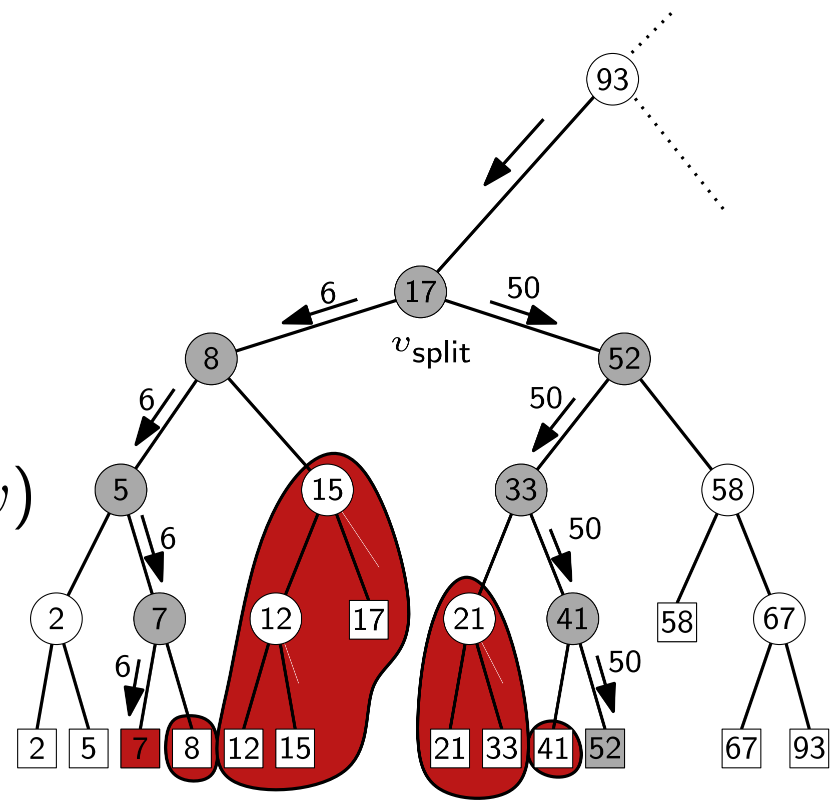
FindSplitNode(T, x, x')

$v \leftarrow$ root(T)
while v kein Blatt und $(x' \leq x_v$ oder $x > x_v)$ **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} ist Blatt **then** prüfe v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v kein Blatt **do**
    if $x \leq x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  prüfe v
  `// analog für` x' `und rc(`v_{split}`)`

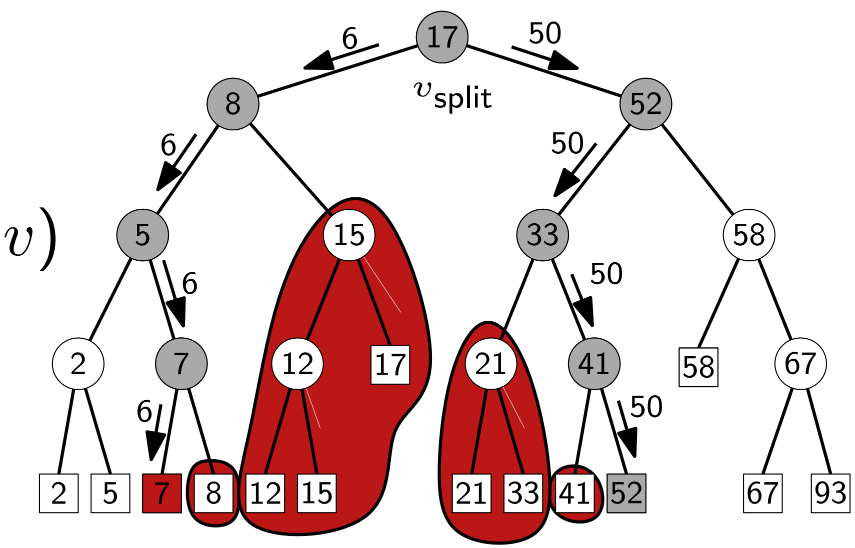
gibt *kanonische Blattmenge* in linearer Zeit aus



Analyse von 1dRangeQuery

1dRangeQuery(T, x, x')

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} ist Blatt **then** prüfe v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v kein Blatt **do**
    if $x \leq x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  prüfe v
  `// analog für` x' `und rc(`v_{split}`)`



Satz: Eine Menge von n Punkten in $\mathbb{R}$ kann in $O(n \log n)$ Zeit und $O(n)$ Platz so vorverarbeitet werden, dass eine Bereichsanfrage $O(k + \log n)$ Zeit benötigt, wobei k die Antwortgröße ist.

Orthogonale Bereichsabfragen für $d = 2$

Geg: Menge P von n Punkten in $\mathbb{R}^2$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $R = [x, x'] \times [y, y']$

Orthogonale Bereichsabfragen für $d = 2$

Geg: Menge P von n Punkten in $\mathbb{R}^2$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $R = [x, x'] \times [y, y']$

Ideen zur Verallgemeinerung des 1d Falls?

Orthogonale Bereichsabfragen für $d = 2$

Geg: Menge P von n Punkten in $\mathbb{R}^2$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $R = [x, x'] \times [y, y']$

Ideen zur Verallgemeinerung des 1d Falls?

Lösungsansätze:

- *ein* Suchbaum, der abwechselnd nach x- und y-Koordinaten trennt $\rightarrow$ kd-**Tree**
- *ein* Suchbaum für x-Koordinaten,
  mehrere untergeordnete Suchbäume für y-Koordinaten
  $\rightarrow$ **Range-Tree**

Orthogonale Bereichsabfragen für $d = 2$

Geg: Menge P von n Punkten in $\mathbb{R}^2$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $R = [x, x'] \times [y, y']$

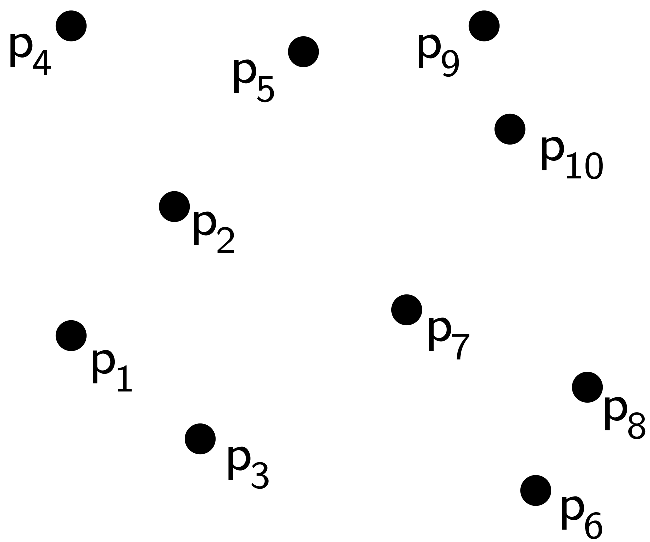
Ideen zur Verallgemeinerung des 1d Falls?

Lösungsansätze:

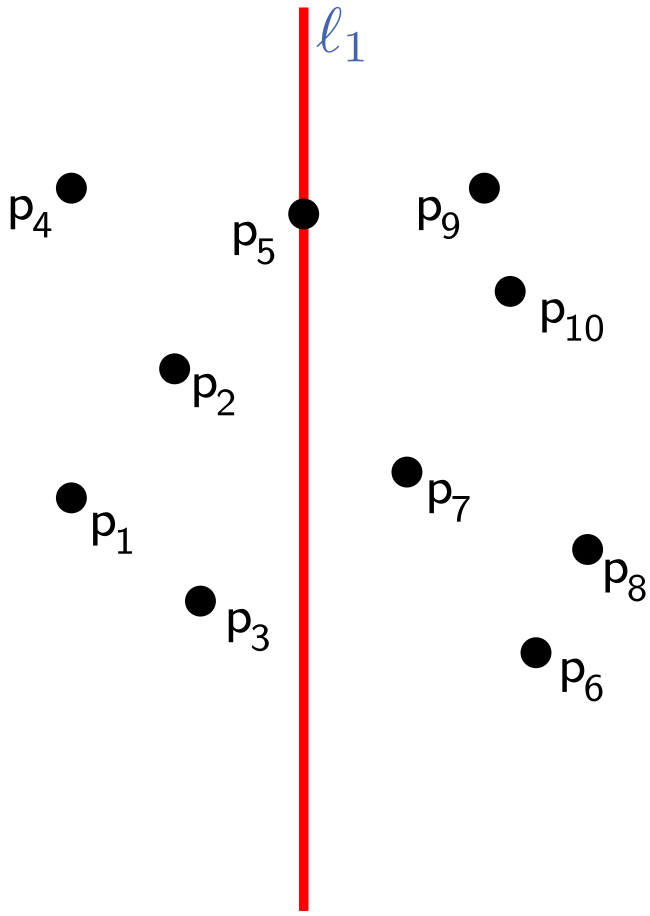
- *ein* Suchbaum, der abwechselnd nach x- und y-Koordinaten trennt $\rightarrow$ kd-**Tree**
- *ein* Suchbaum für x-Koordinaten, *mehrere* untergeordnete Suchbäume für y-Koordinaten $\rightarrow$ **Range-Tree**

vorübergehende Annahme: allgemeine Lage, d.h. keine zwei Punkte haben gleiche x- oder y-Koordinate

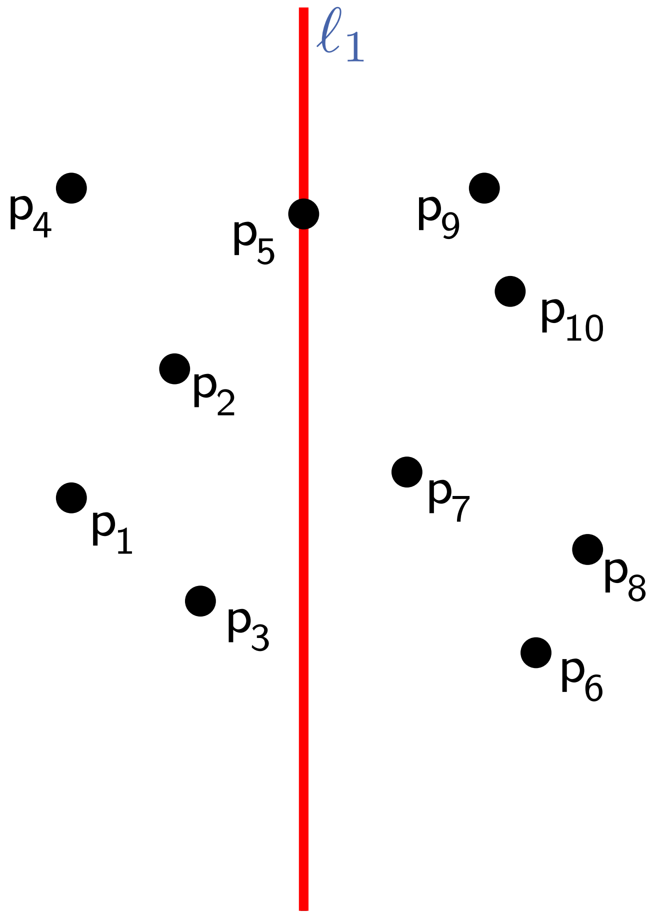
kd-Trees: Beispiel

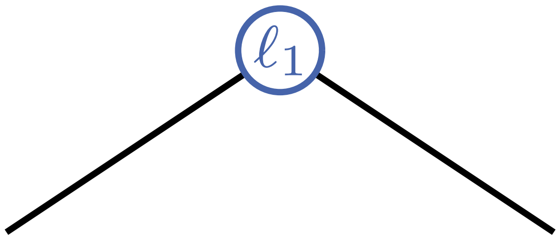


kd-Trees: Beispiel

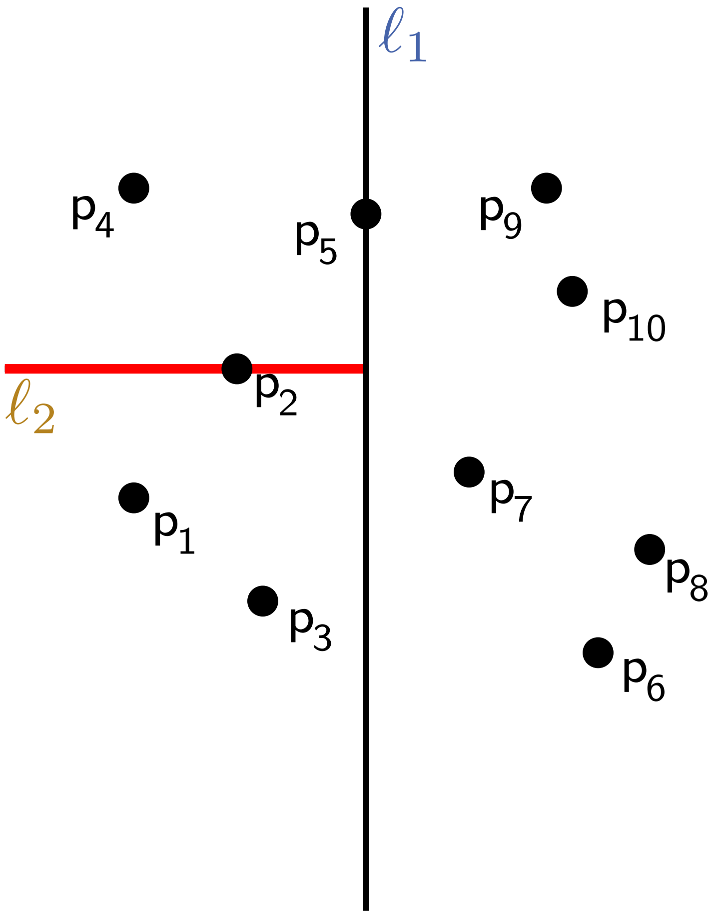


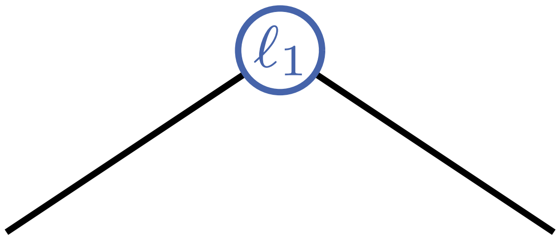
kd-Trees: Beispiel



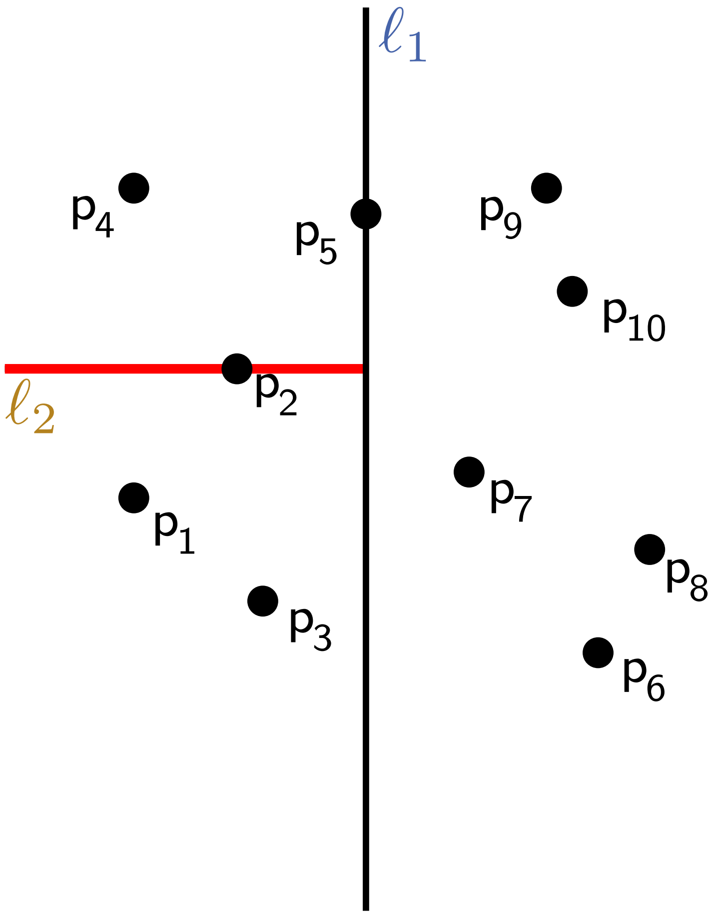


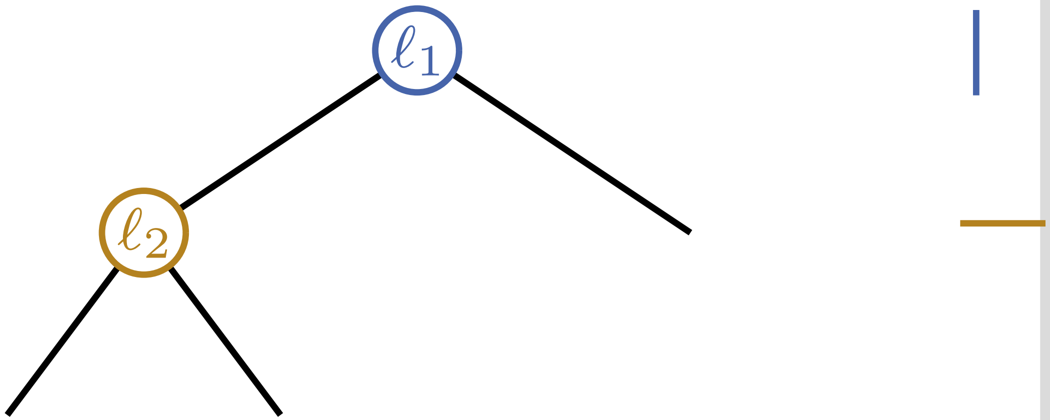
kd-Trees: Beispiel



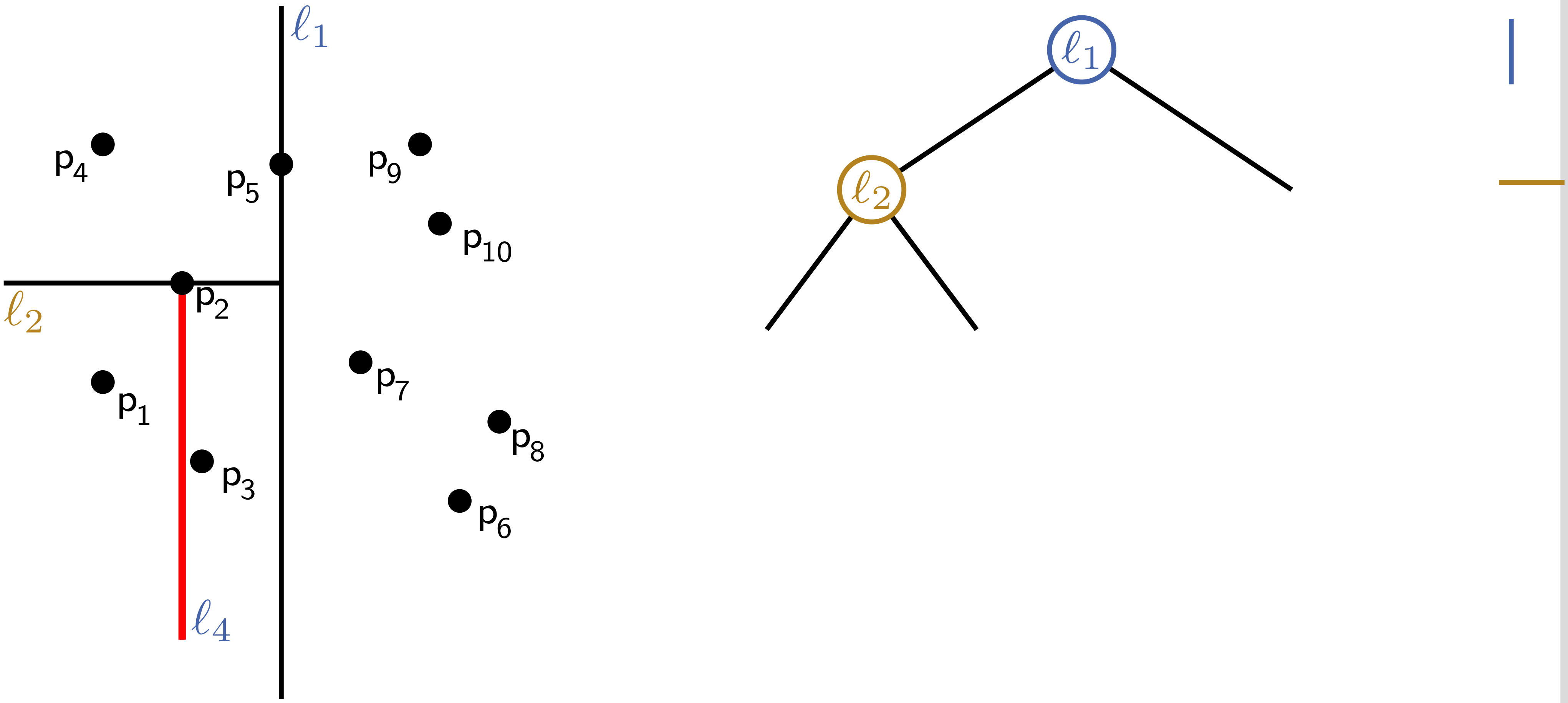


kd-Trees: Beispiel

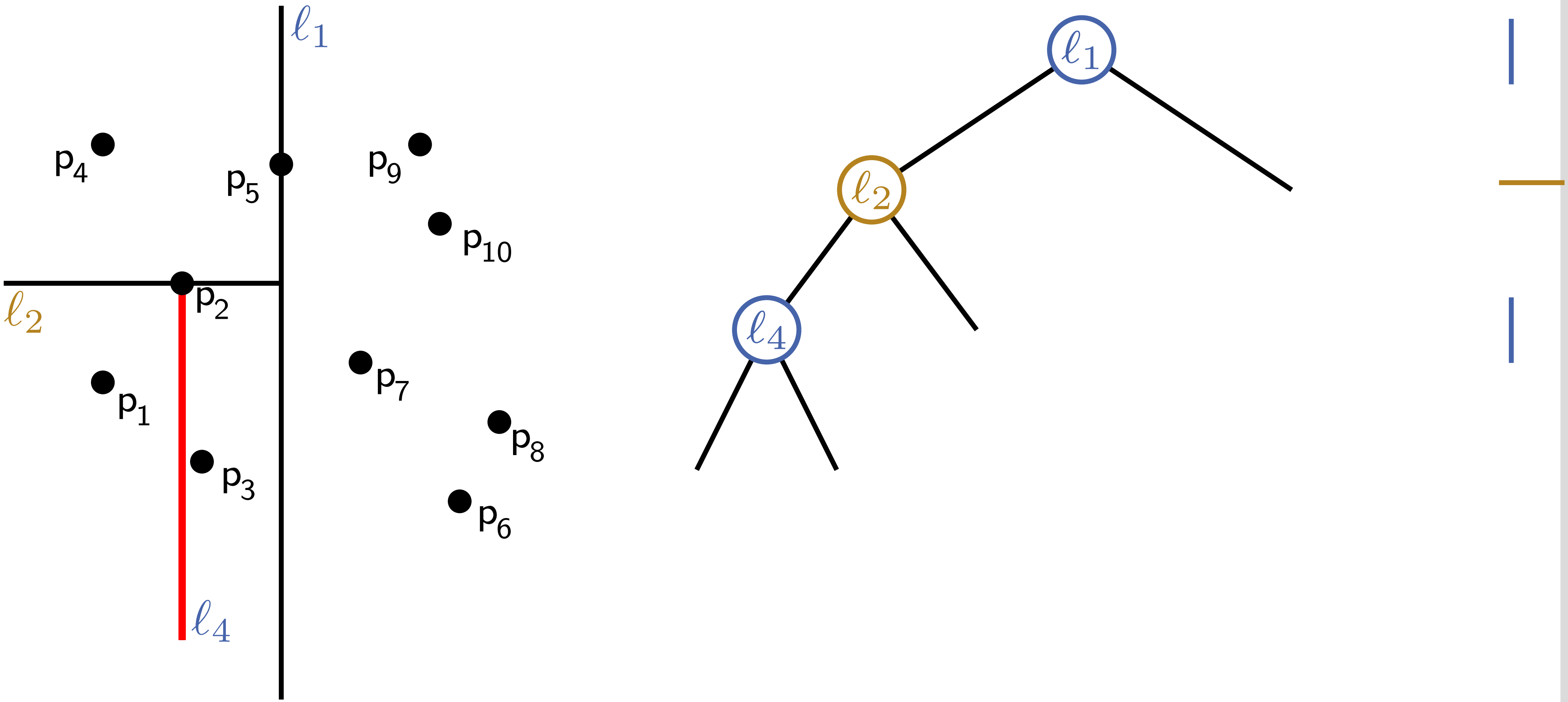




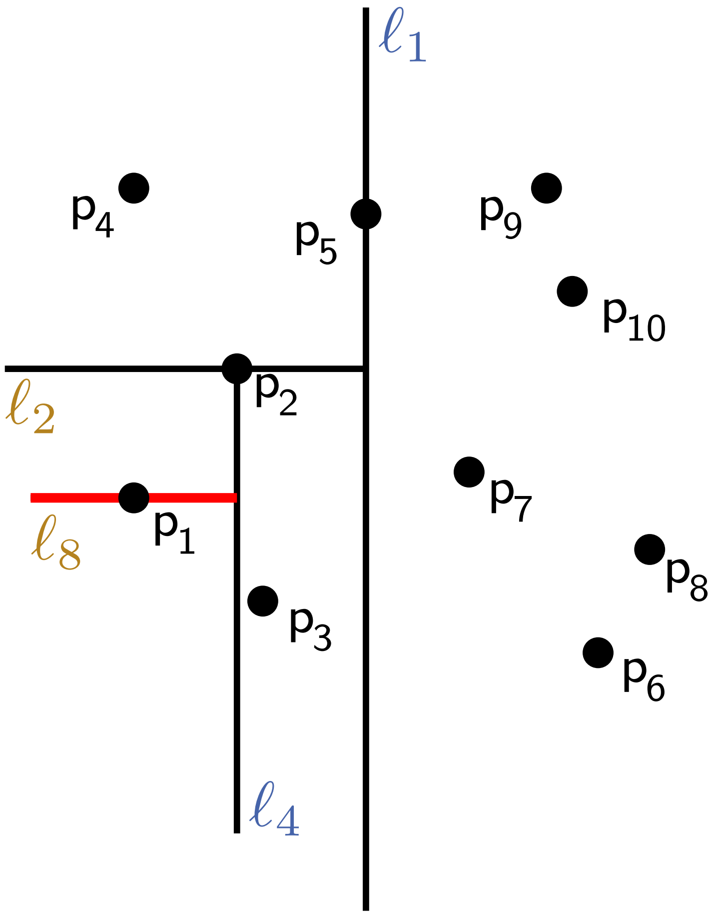
kd-Trees: Beispiel

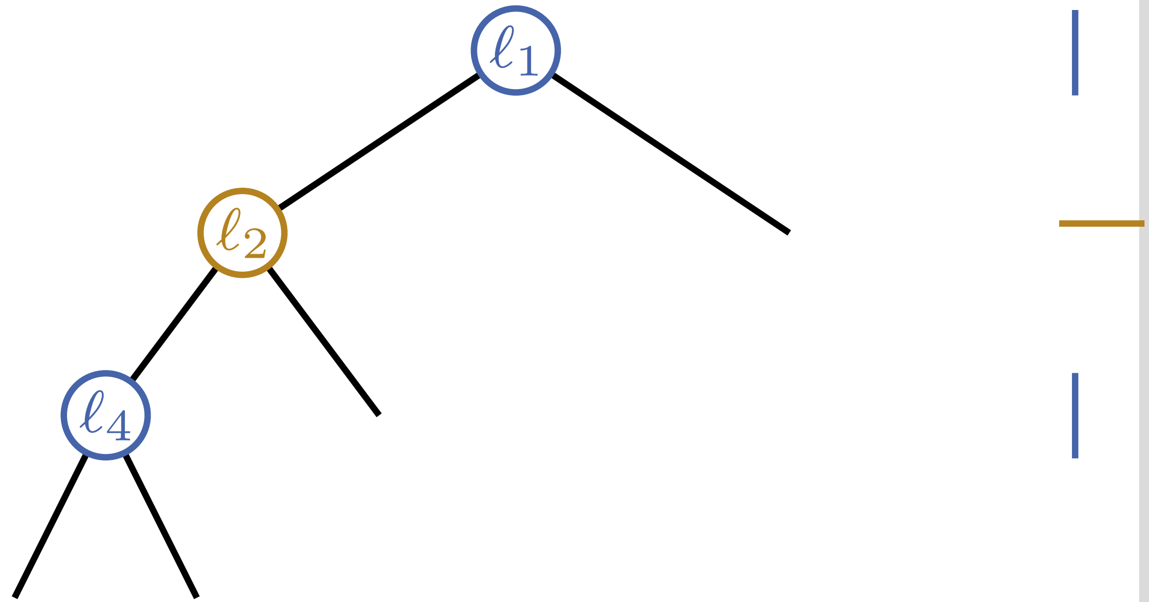


kd-Trees: Beispiel

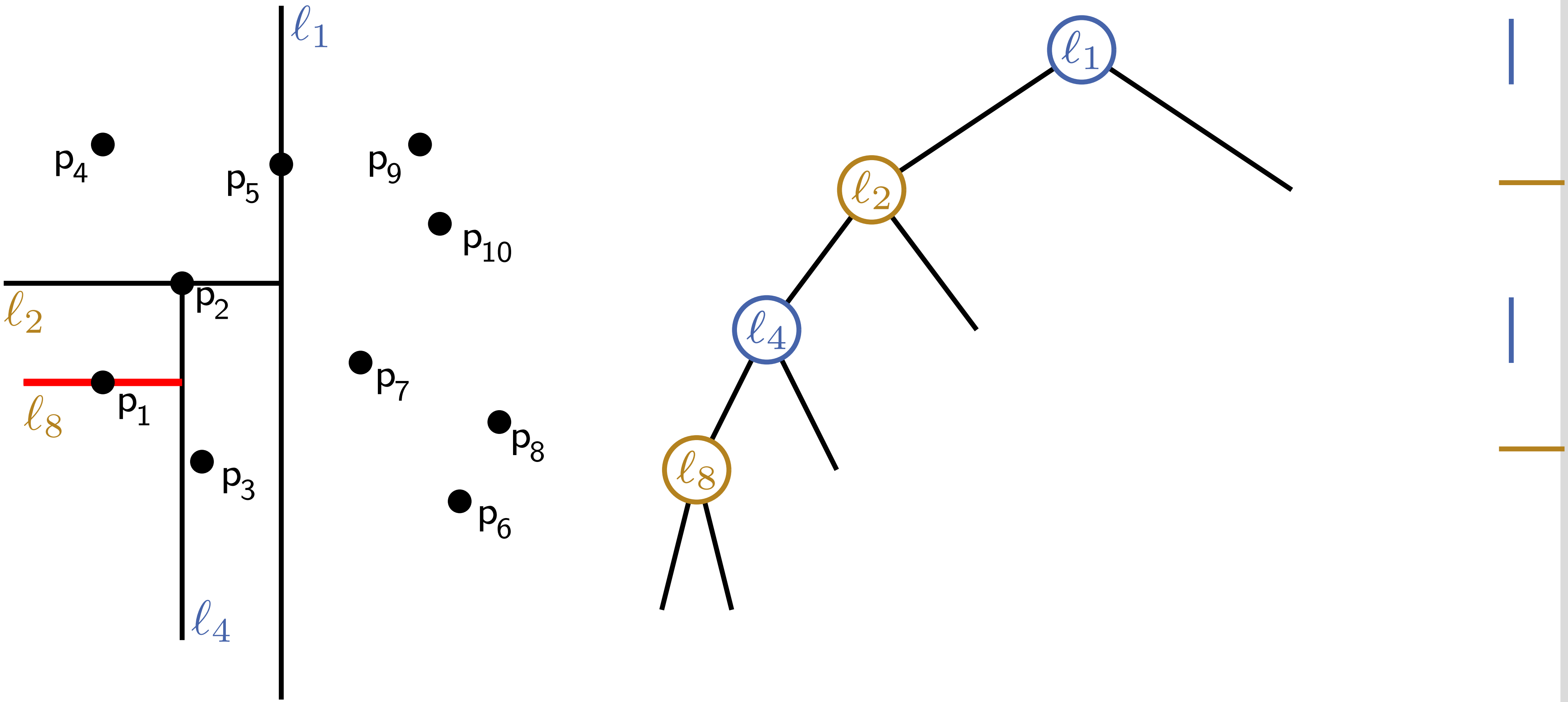


kd-Trees: Beispiel

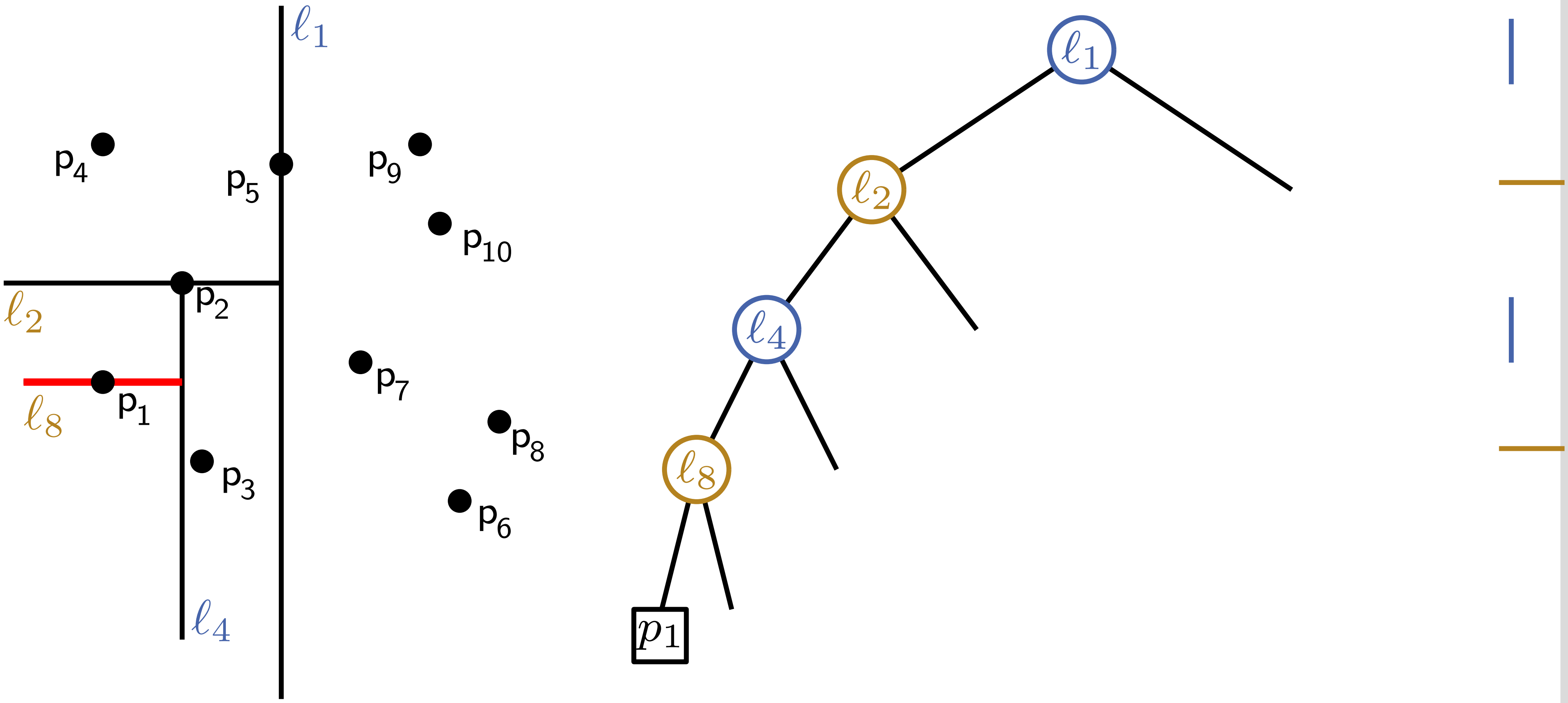




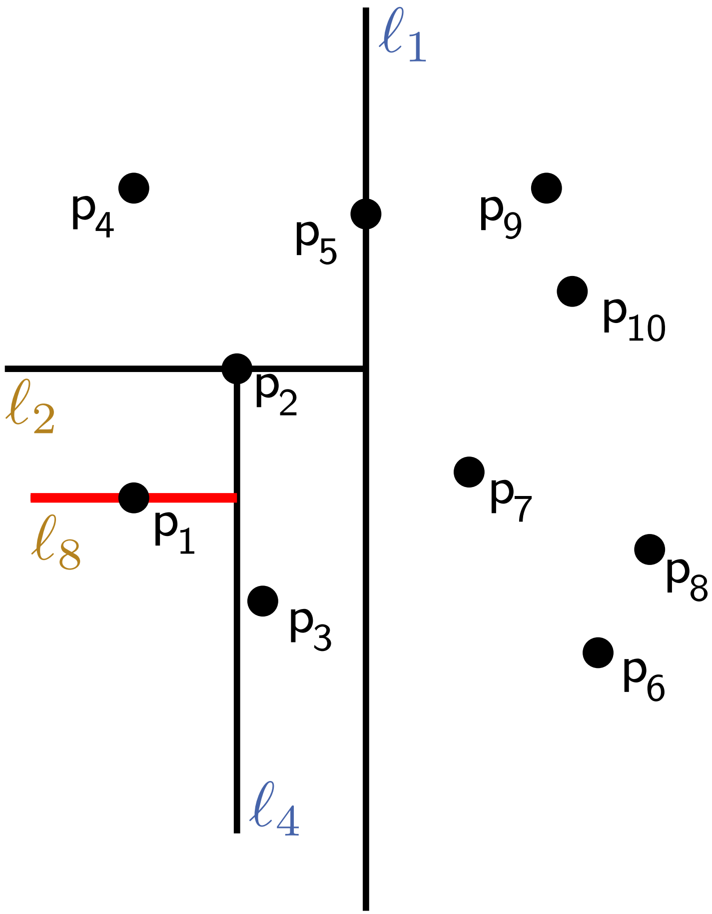
kd-Trees: Beispiel

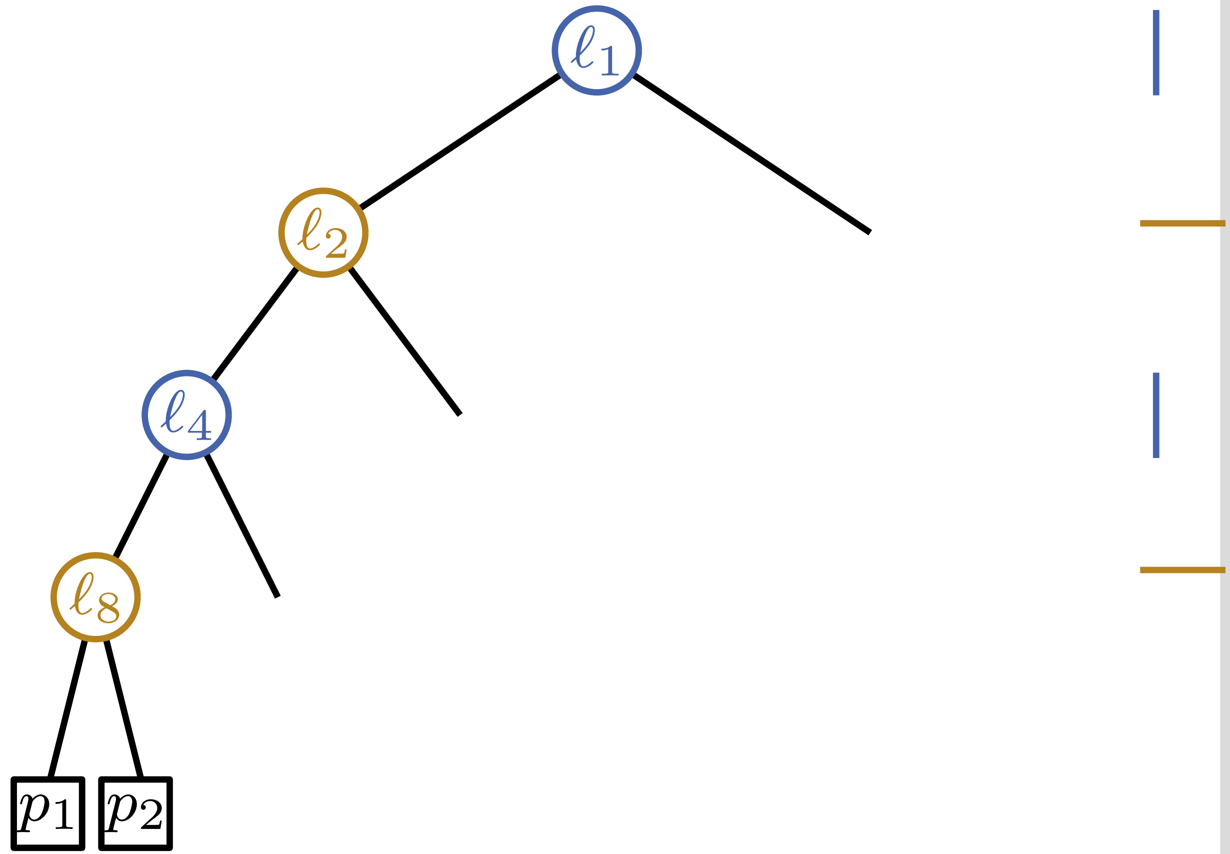


kd-Trees: Beispiel

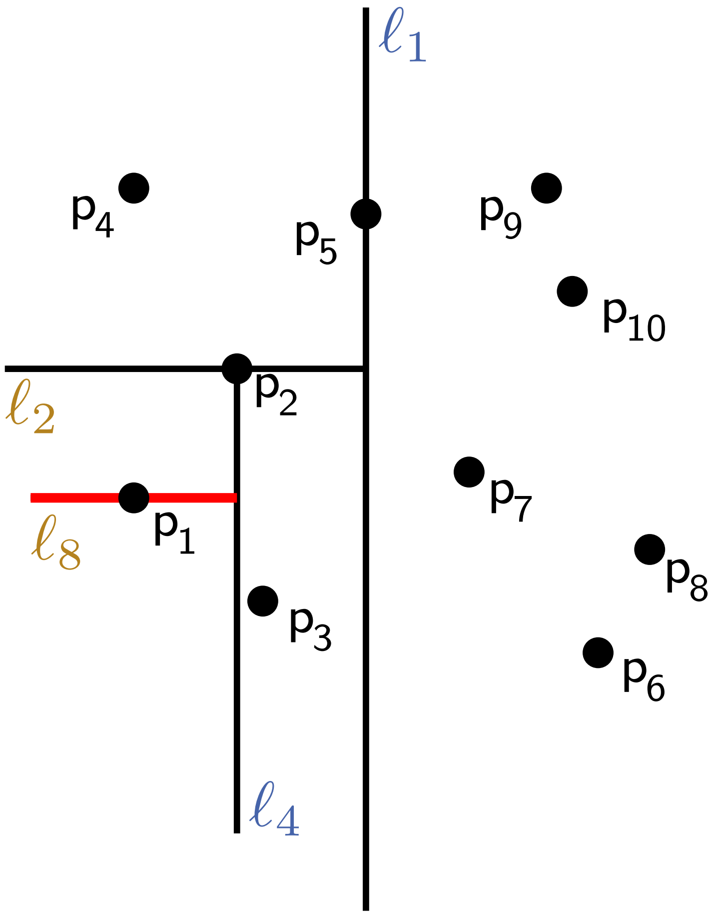


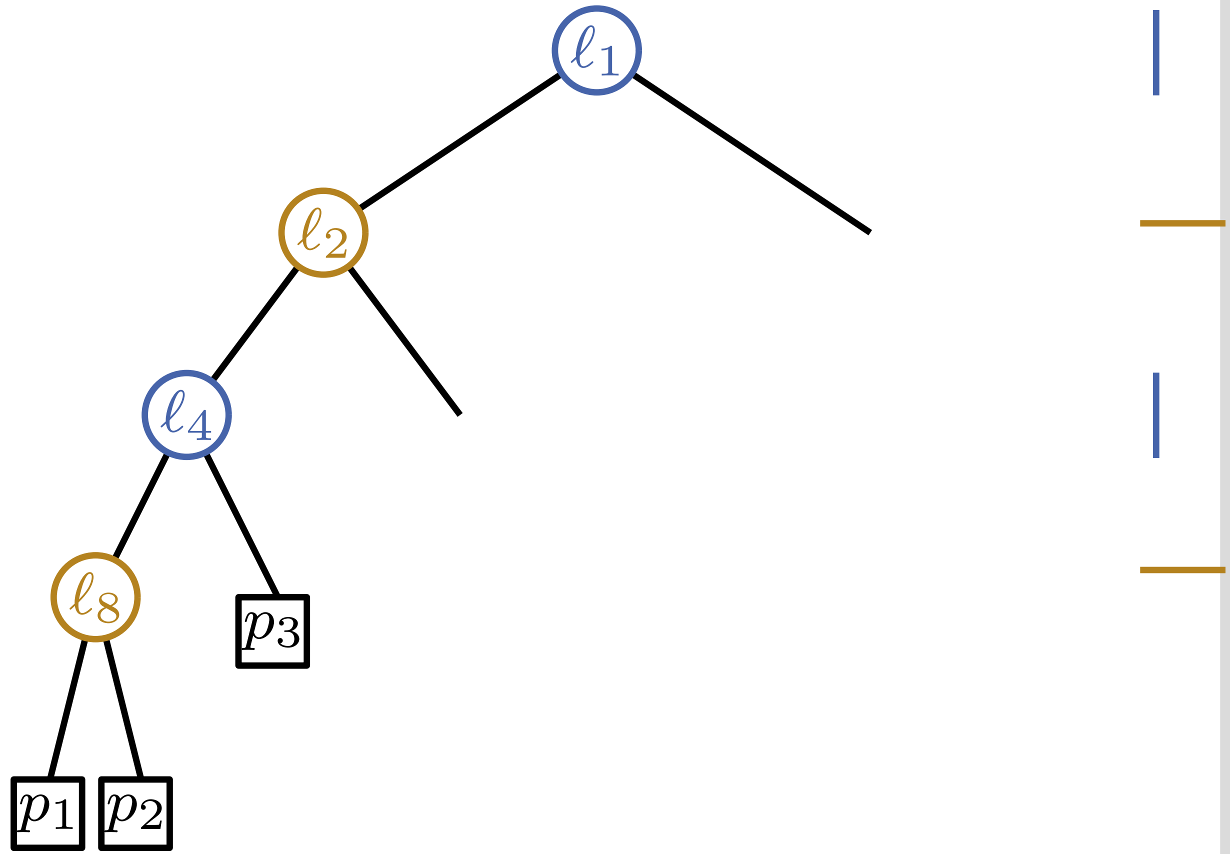
kd-Trees: Beispiel



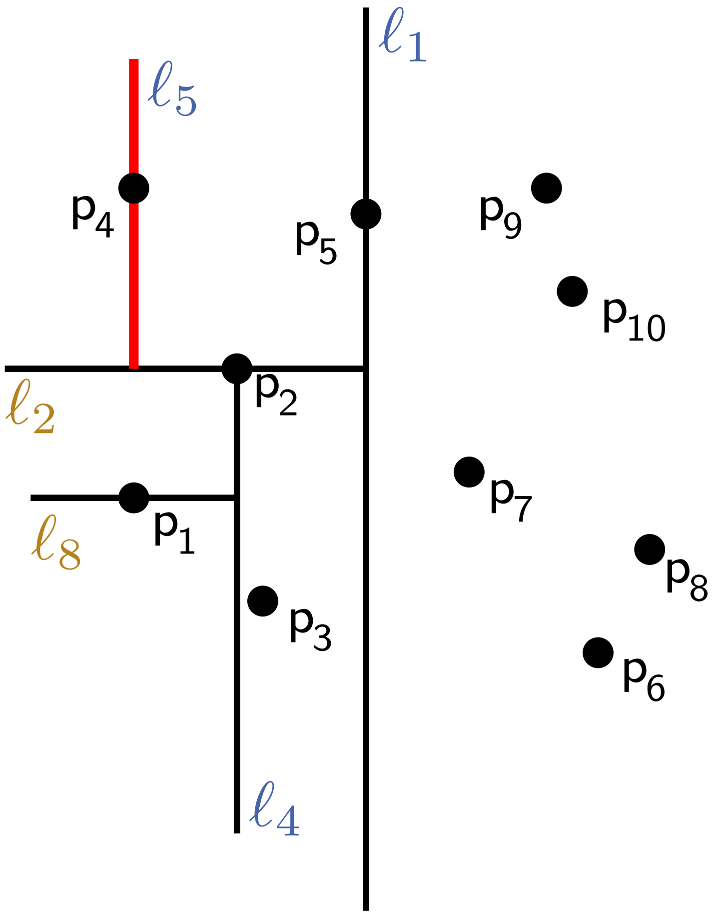


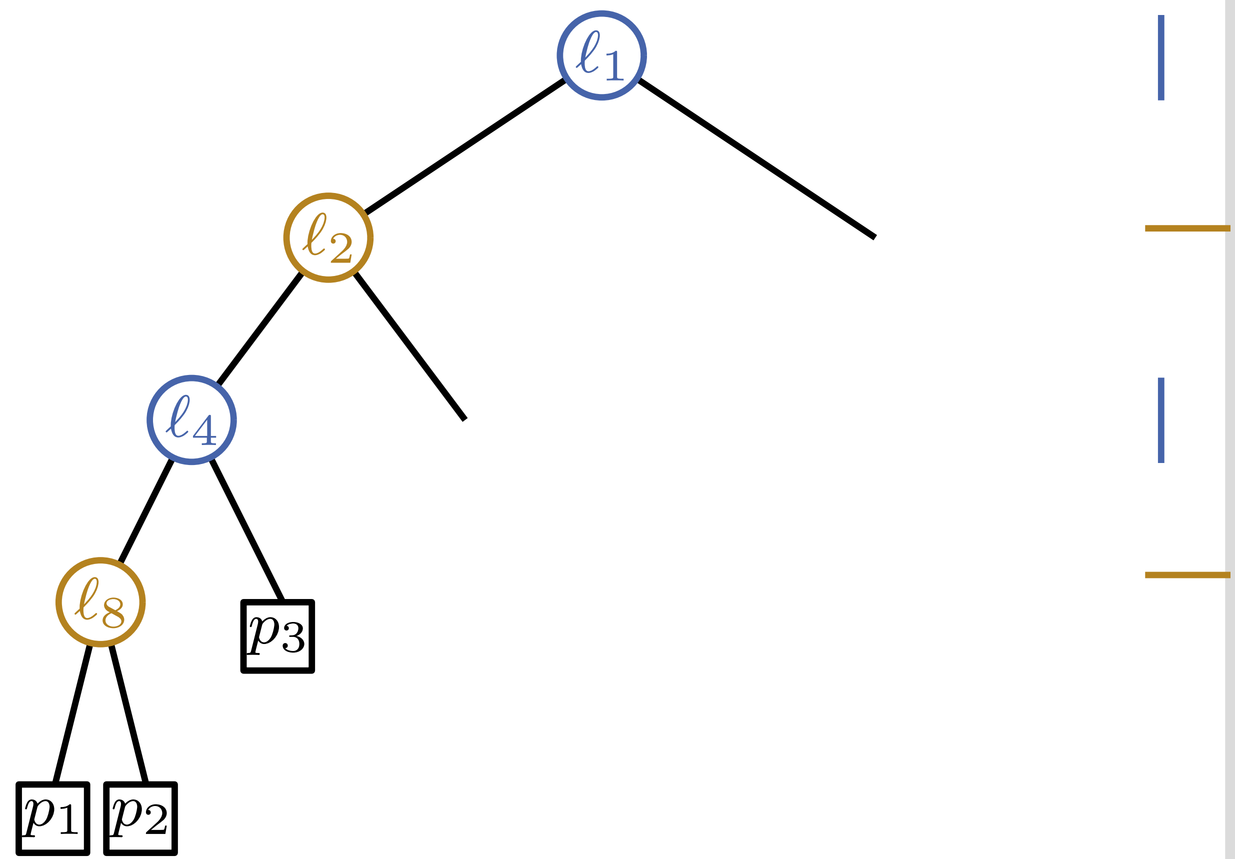
kd-Trees: Beispiel



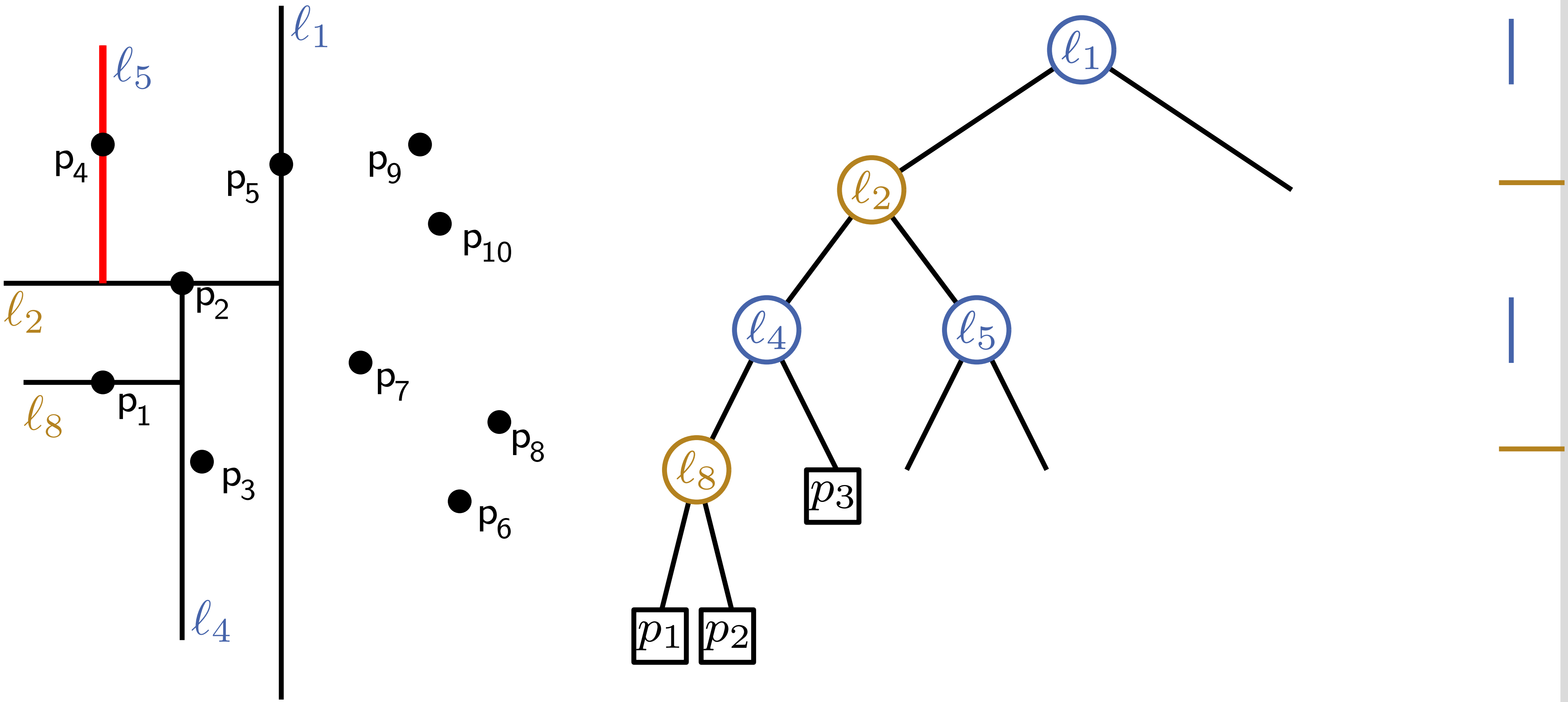


kd-Trees: Beispiel

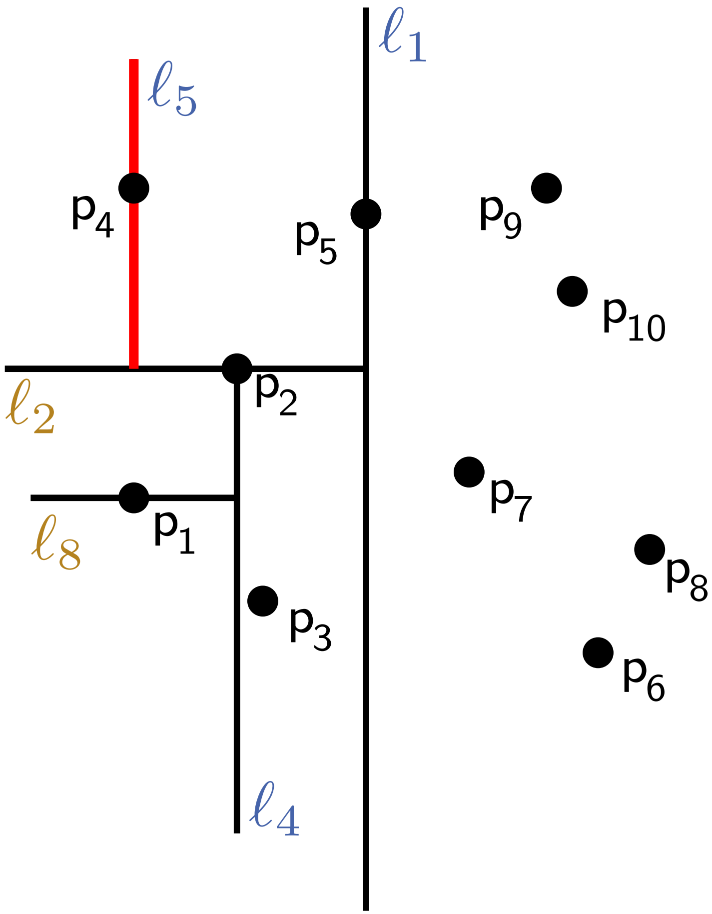


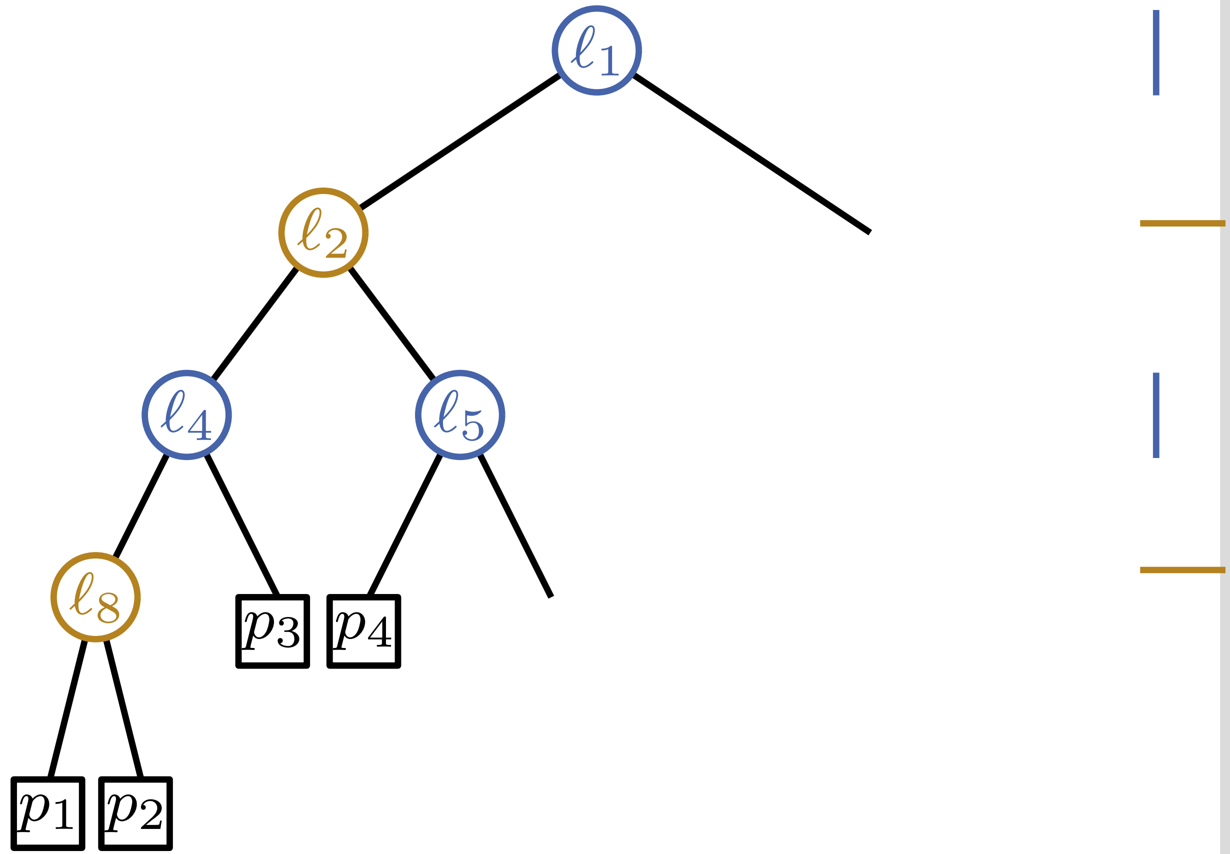


kd-Trees: Beispiel

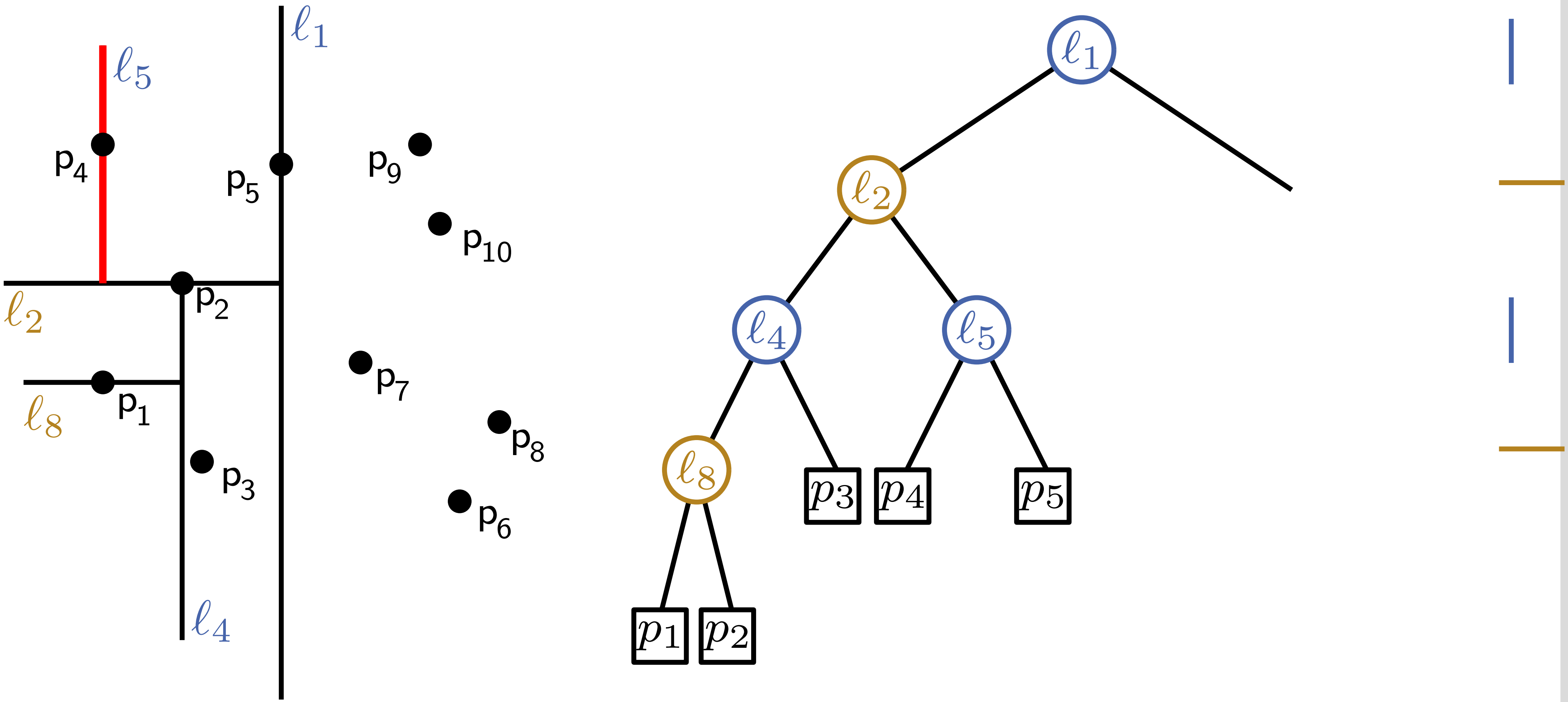


kd-Trees: Beispiel

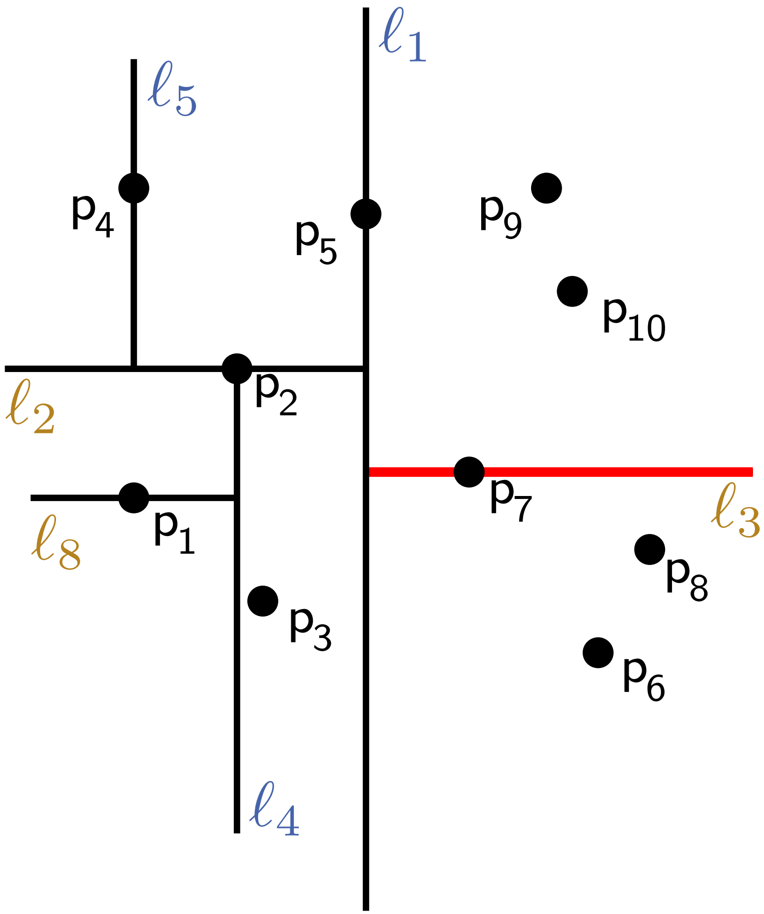


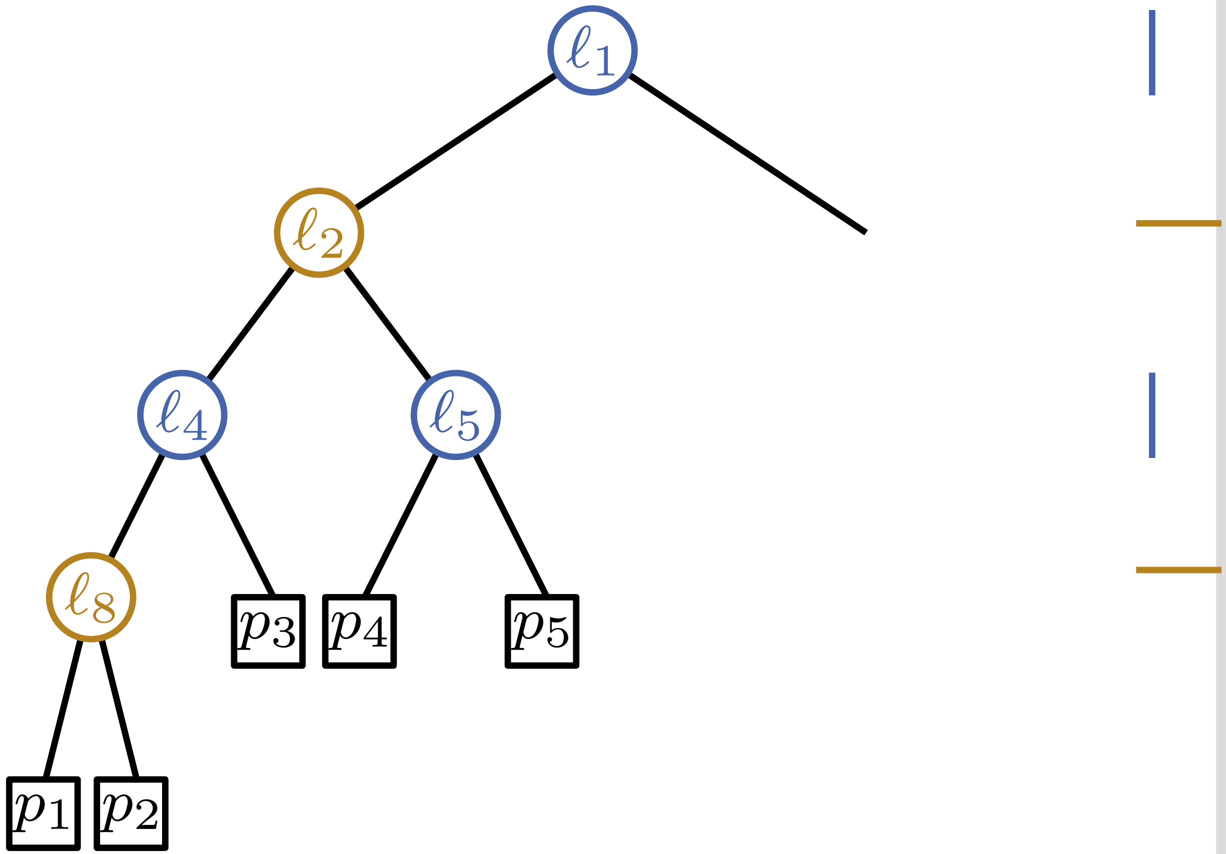


kd-Trees: Beispiel

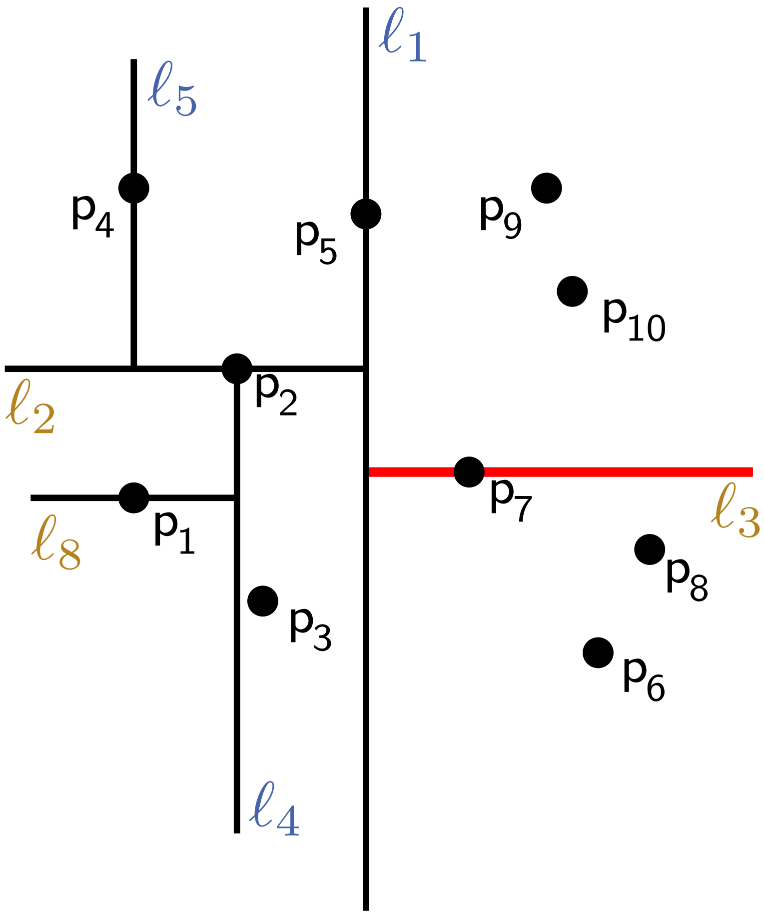


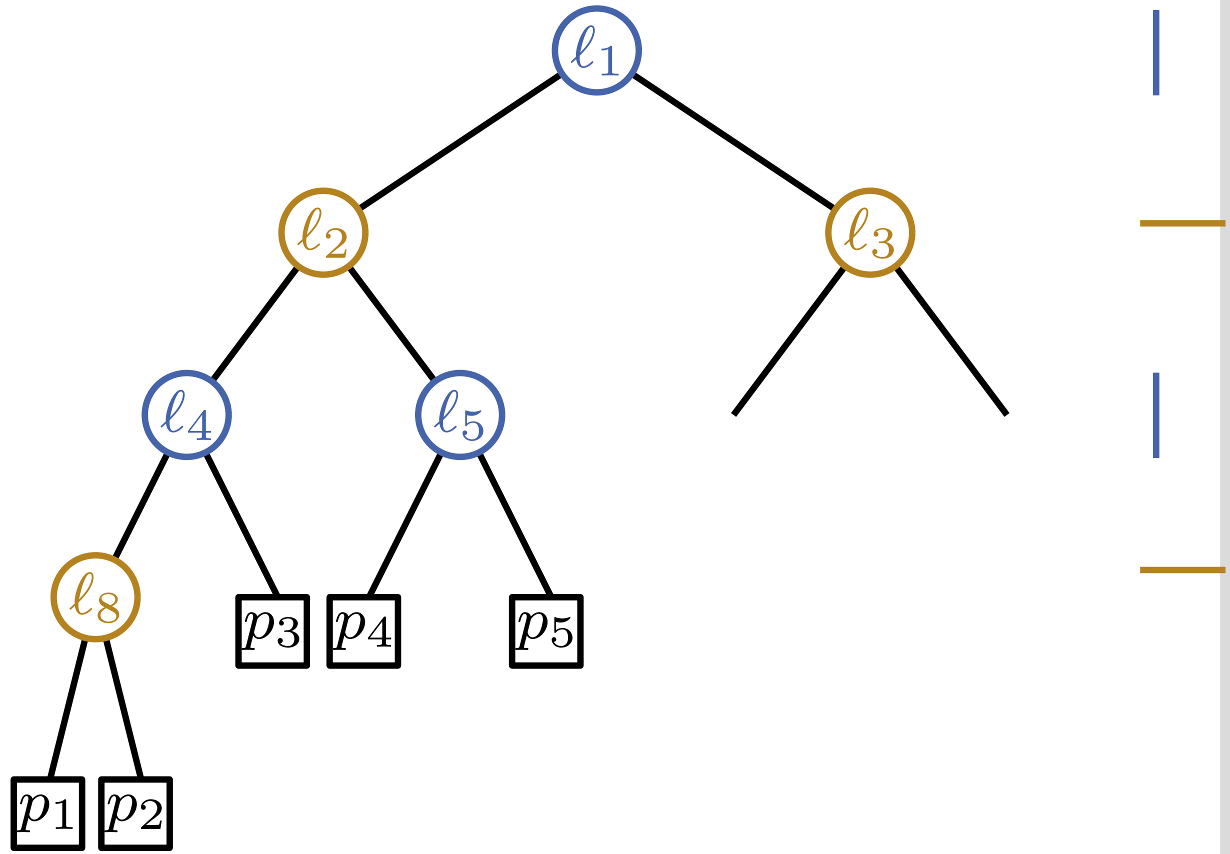
kd-Trees: Beispiel



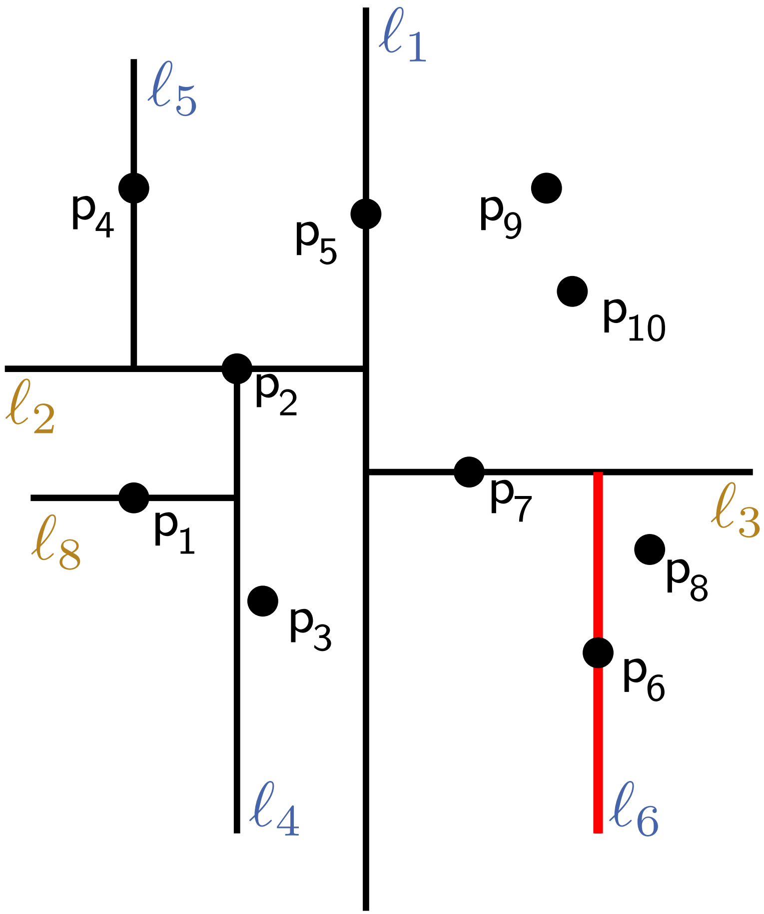


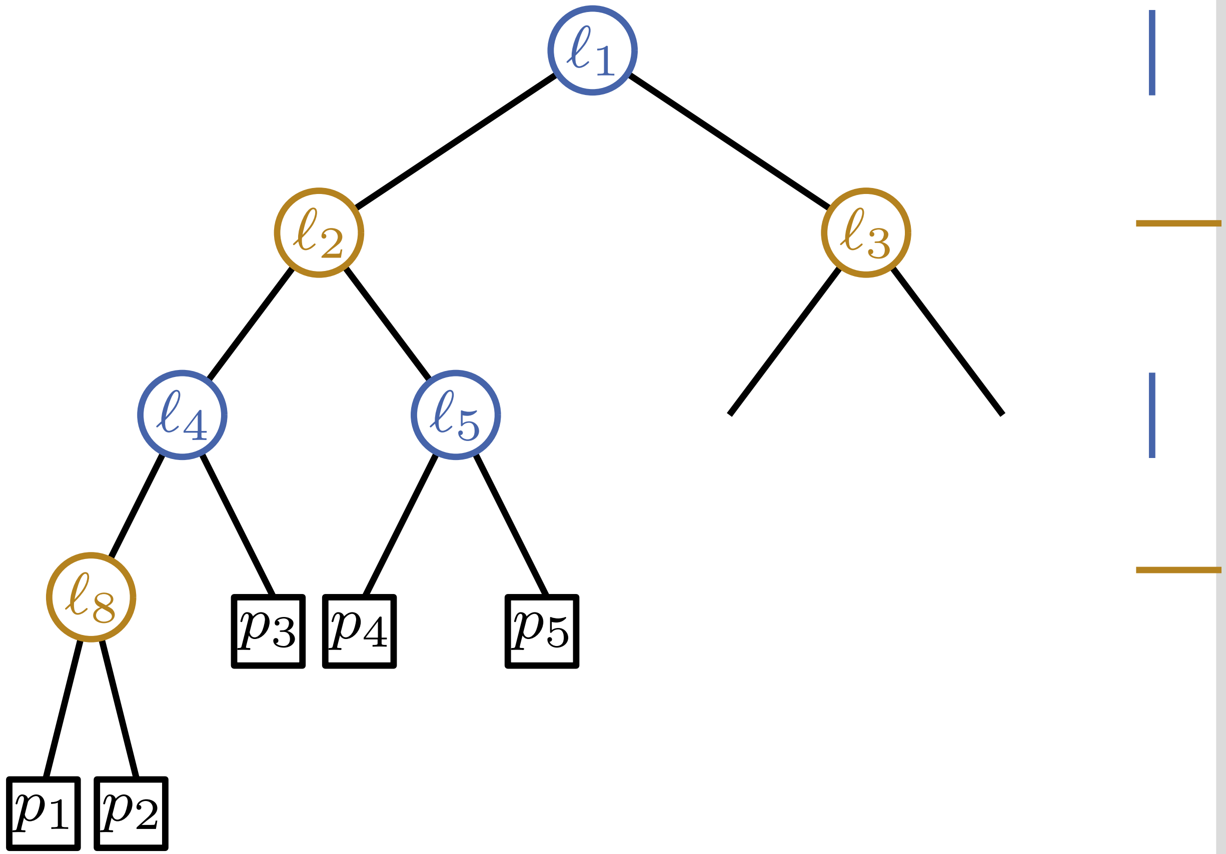
kd-Trees: Beispiel



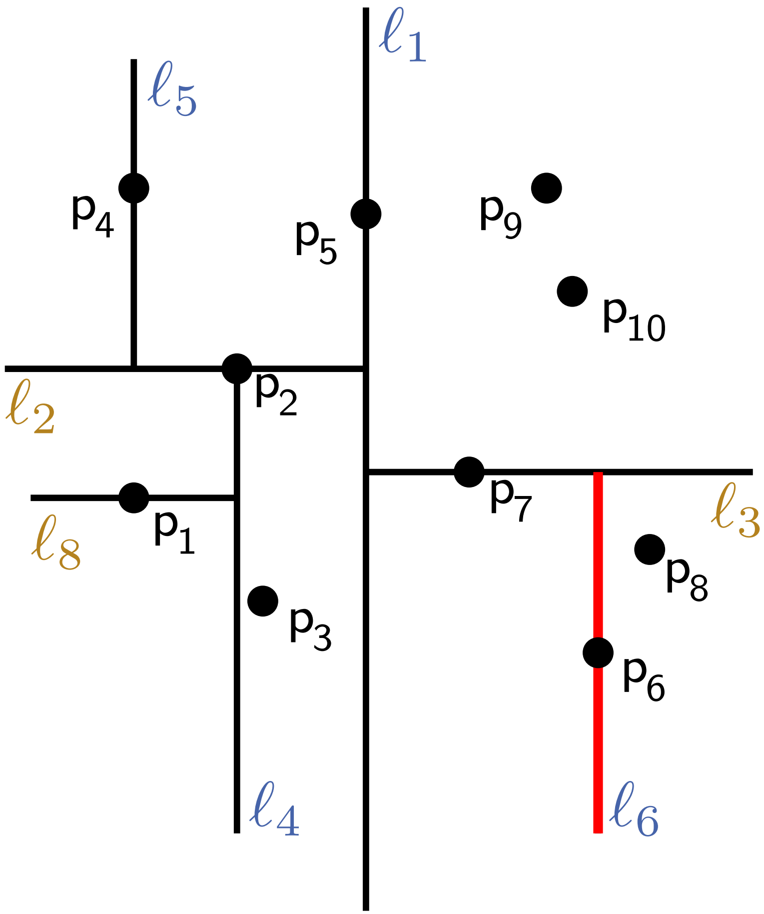


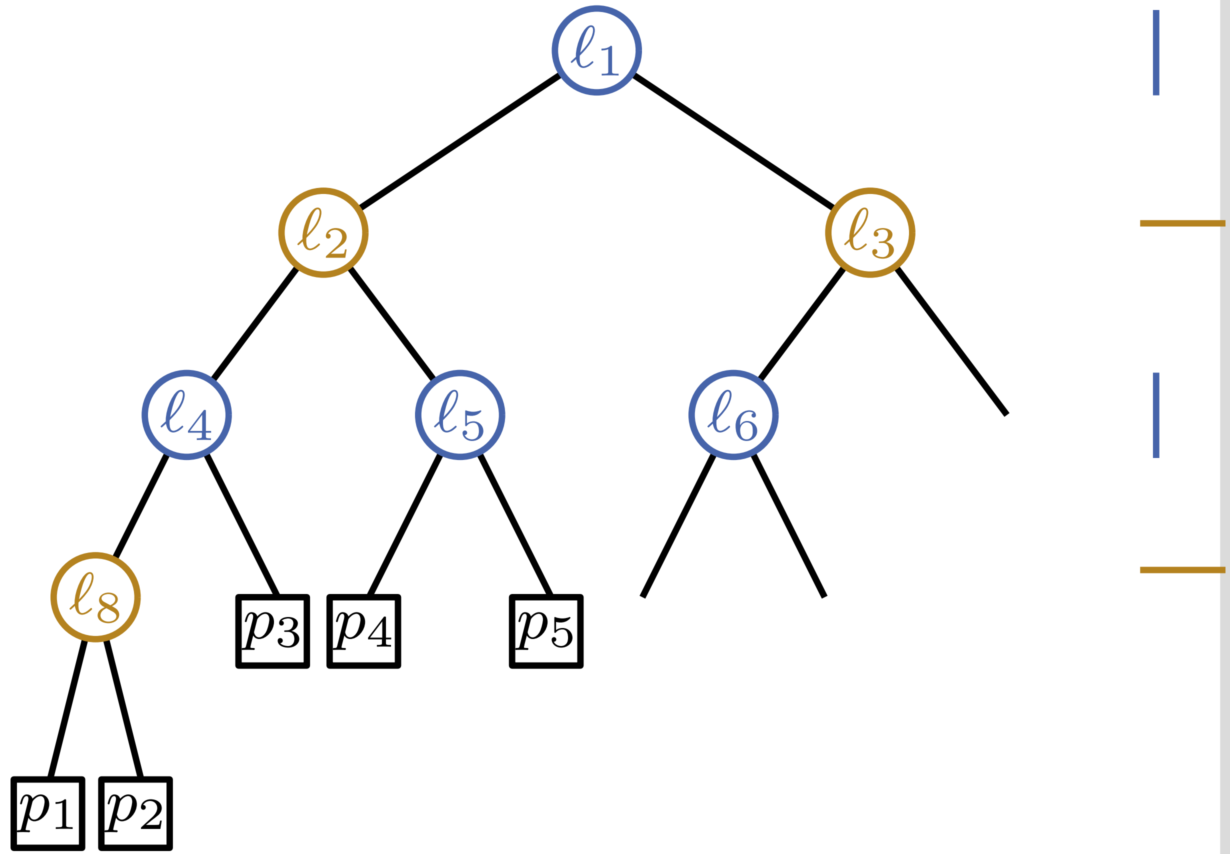
kd-Trees: Beispiel



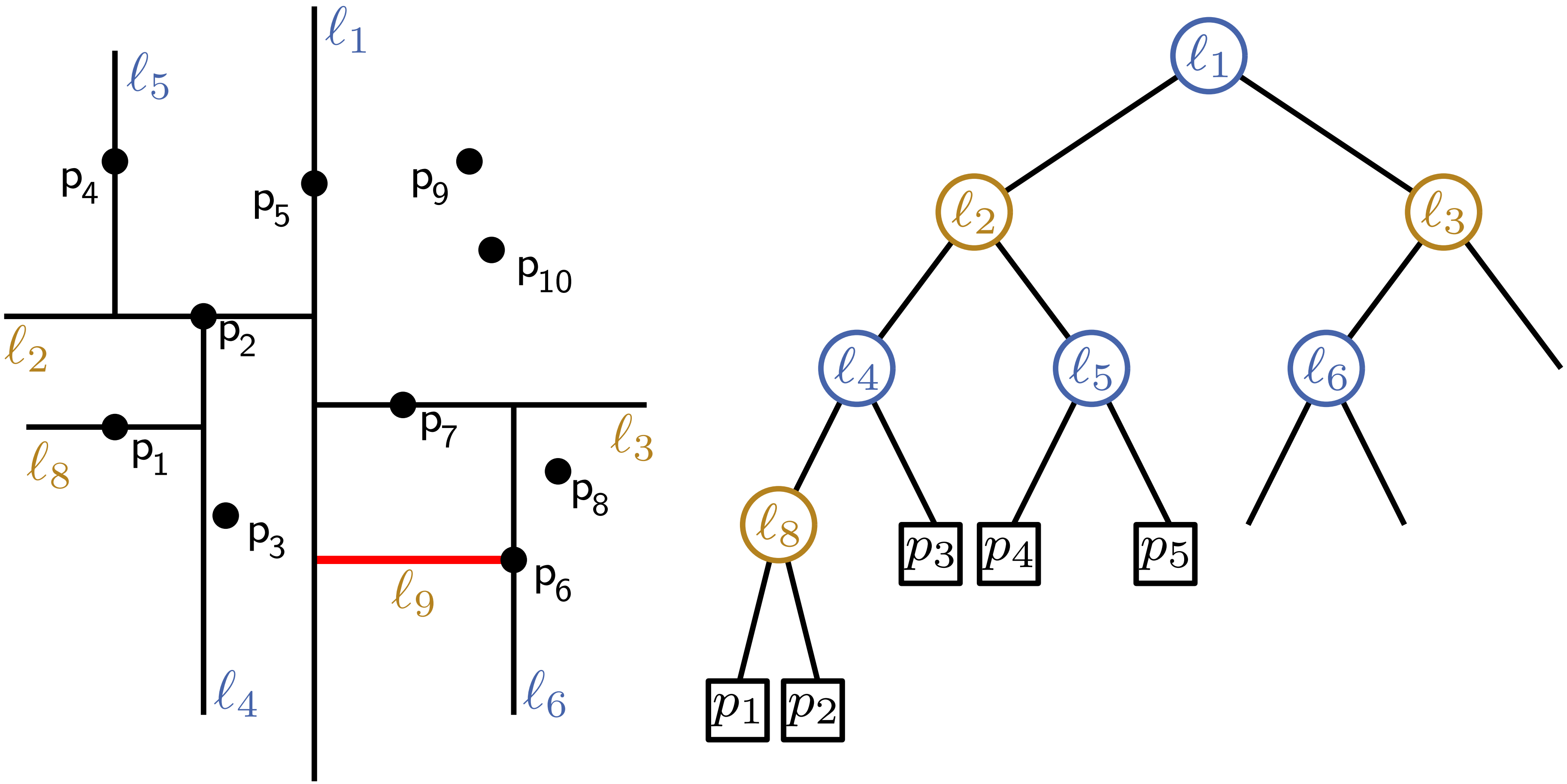


kd-Trees: Beispiel

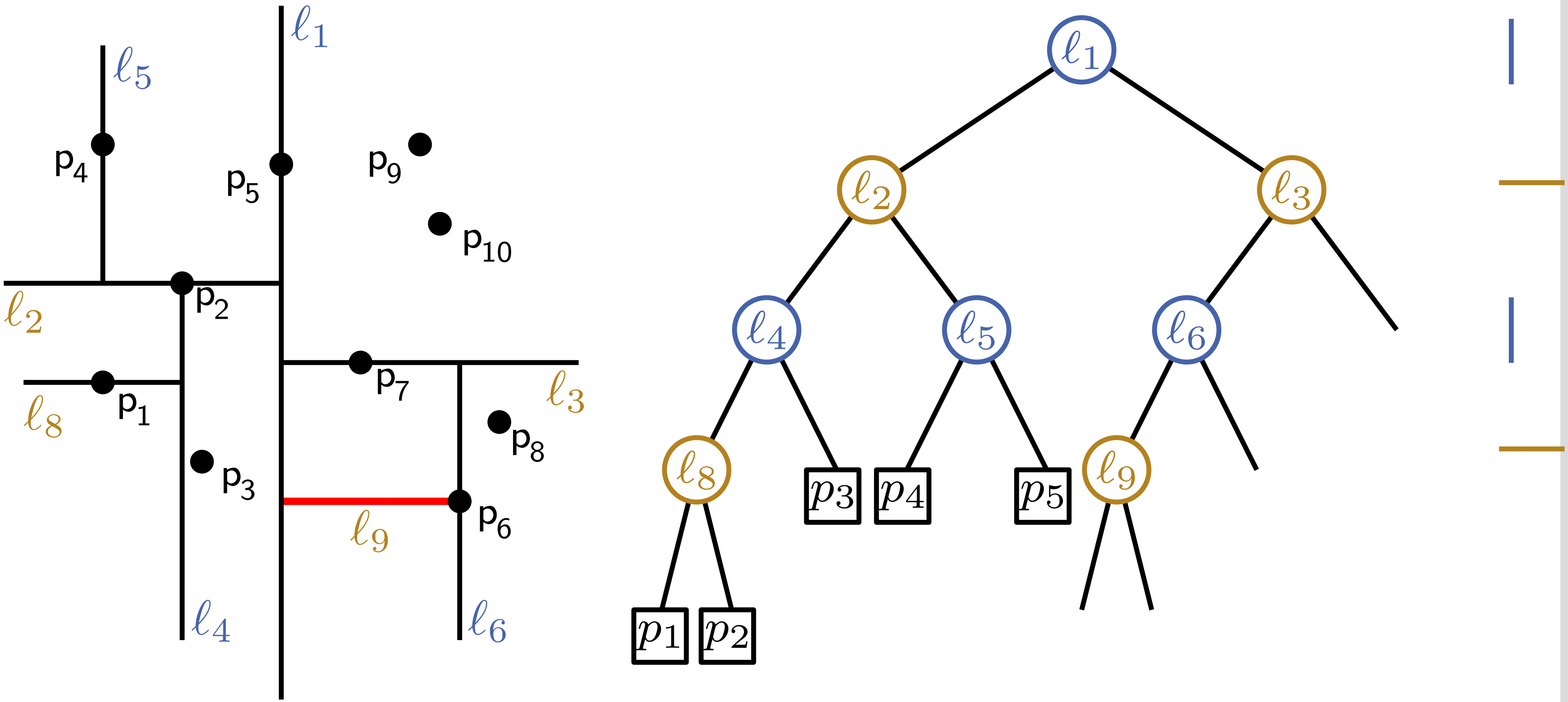




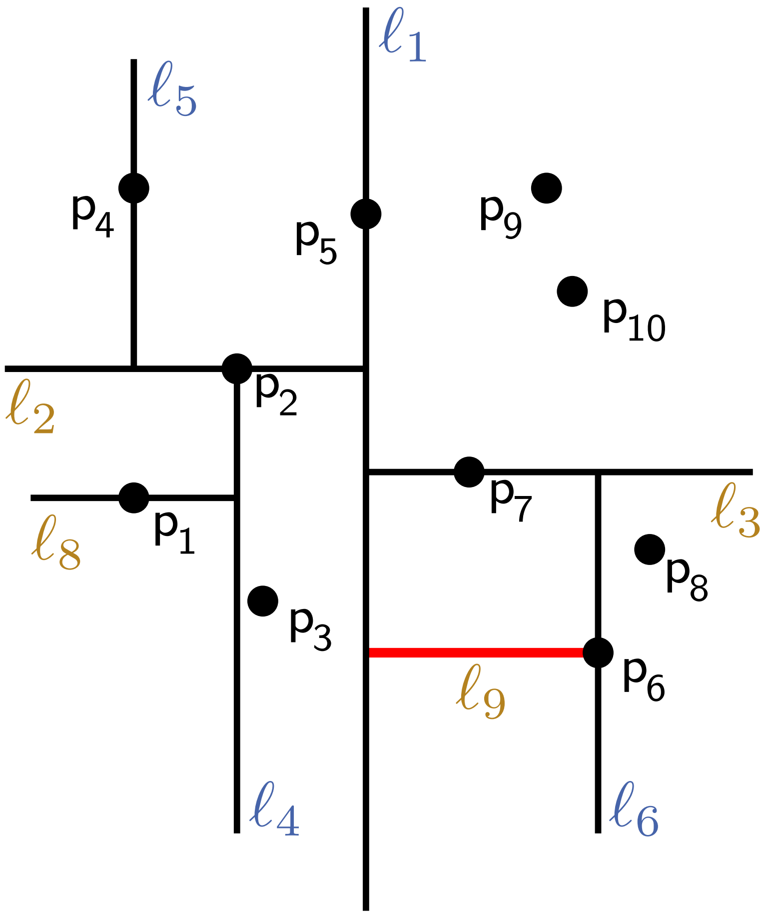
kd-Trees: Beispiel

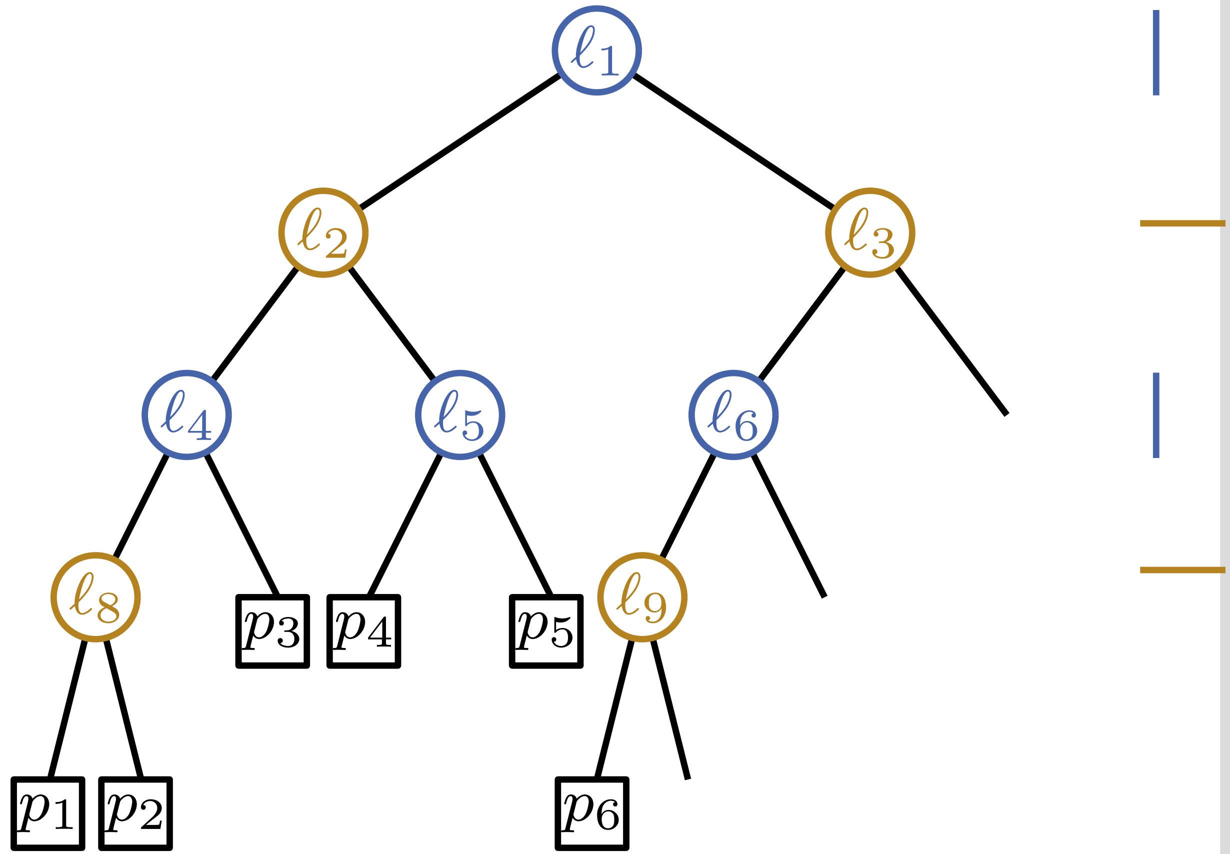


kd-Trees: Beispiel

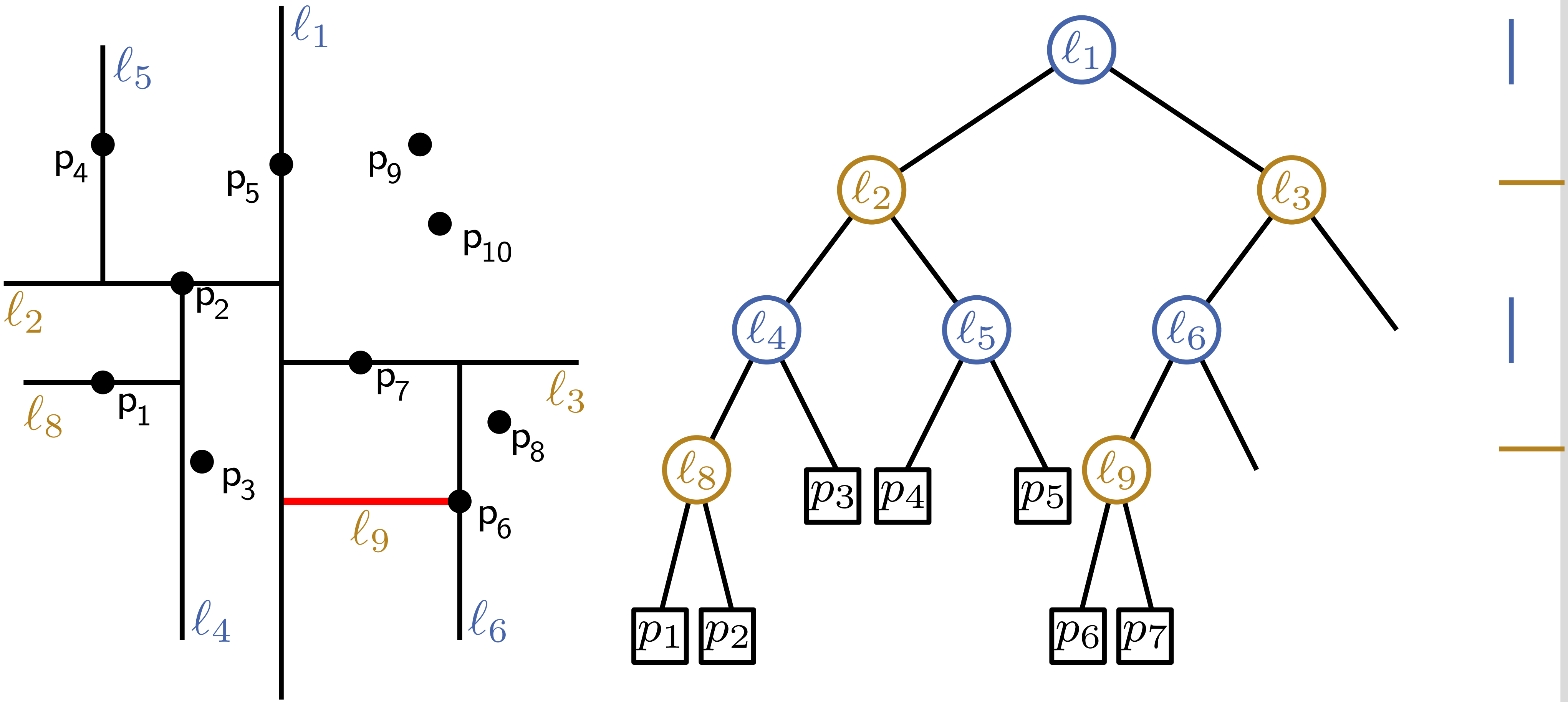


kd-Trees: Beispiel

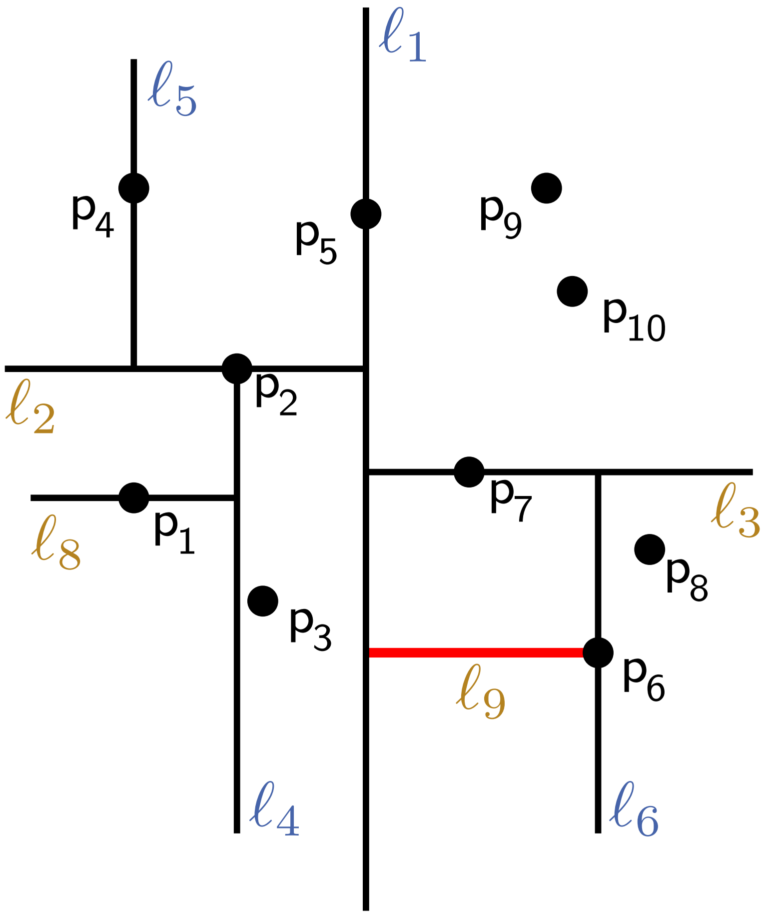


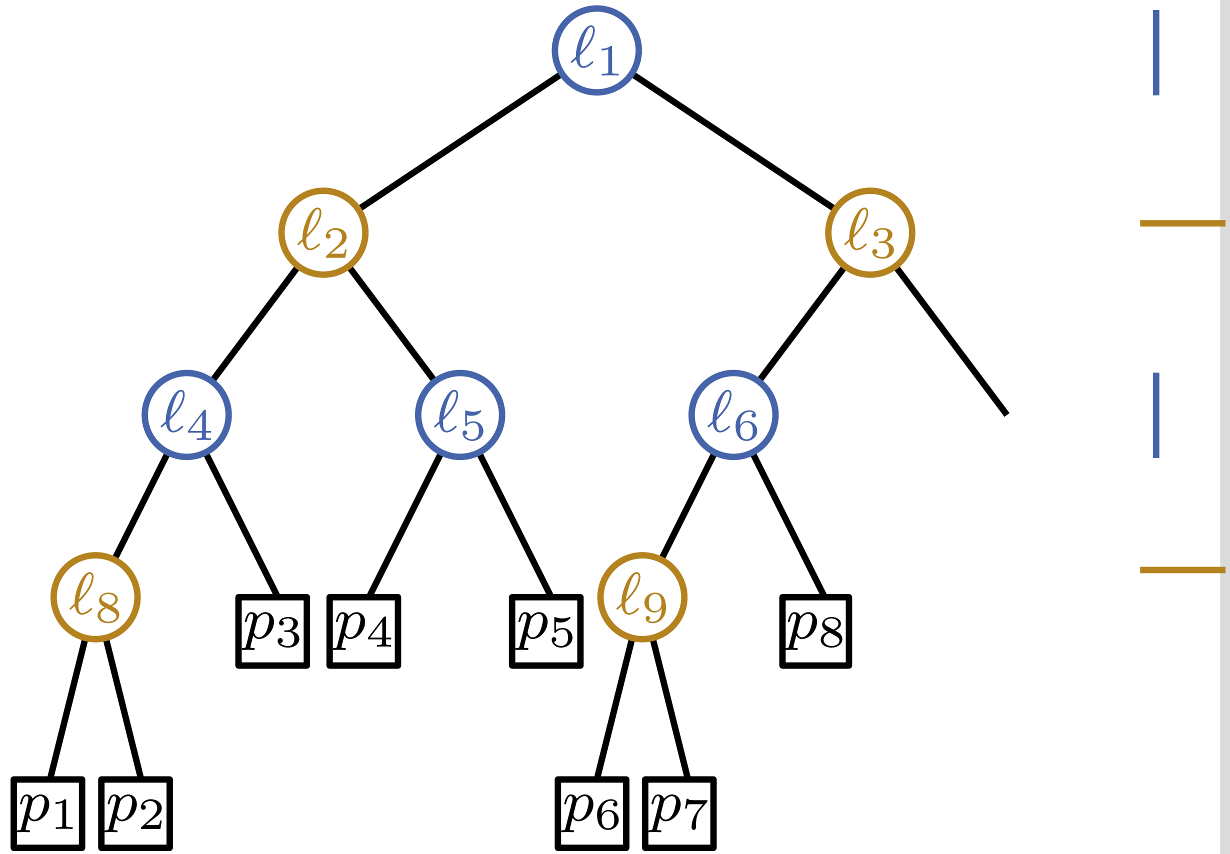


kd-Trees: Beispiel

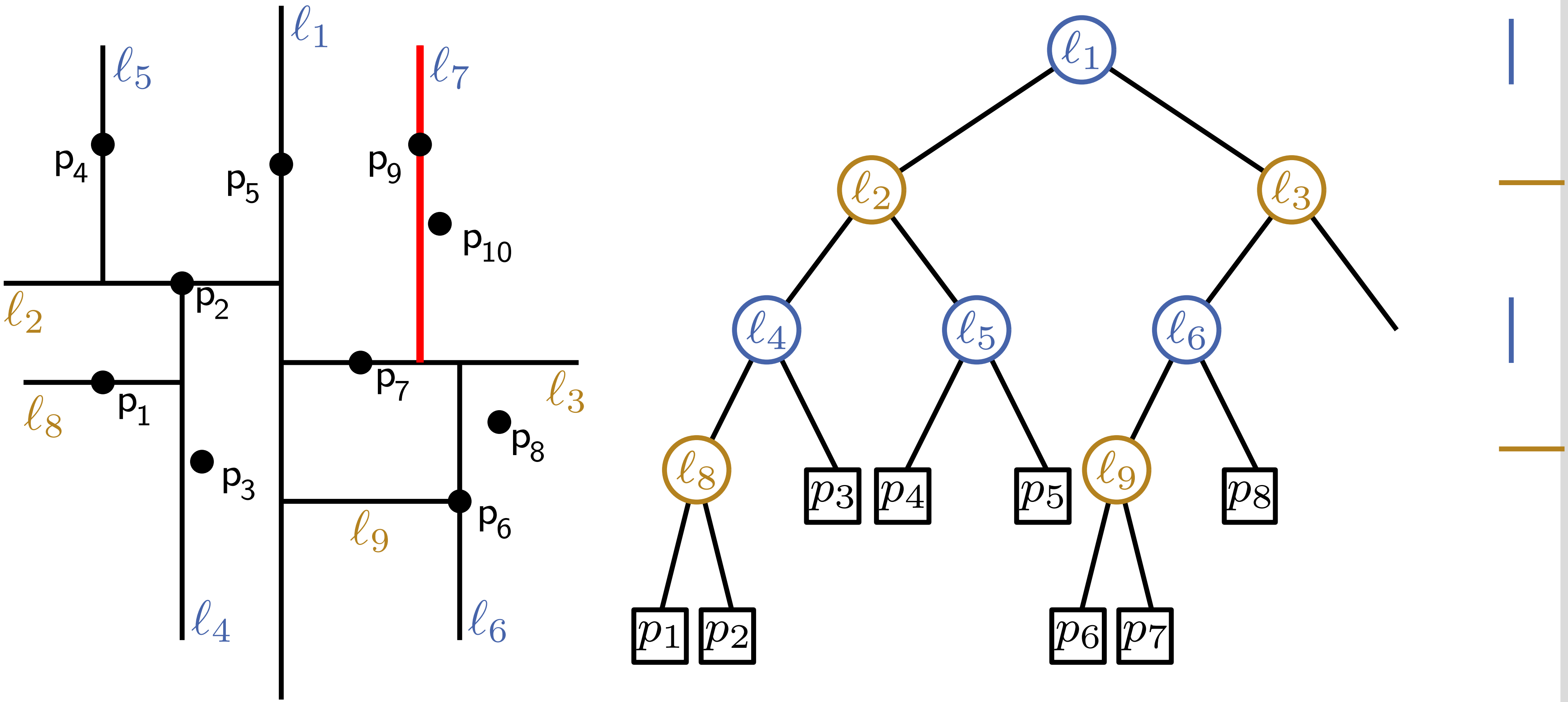


kd-Trees: Beispiel

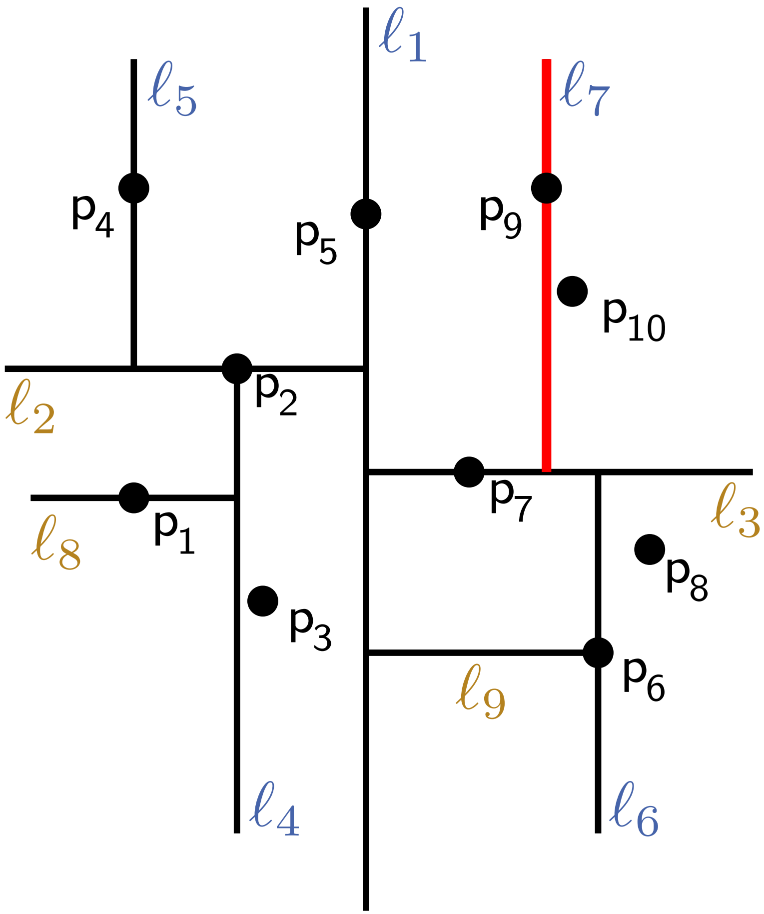


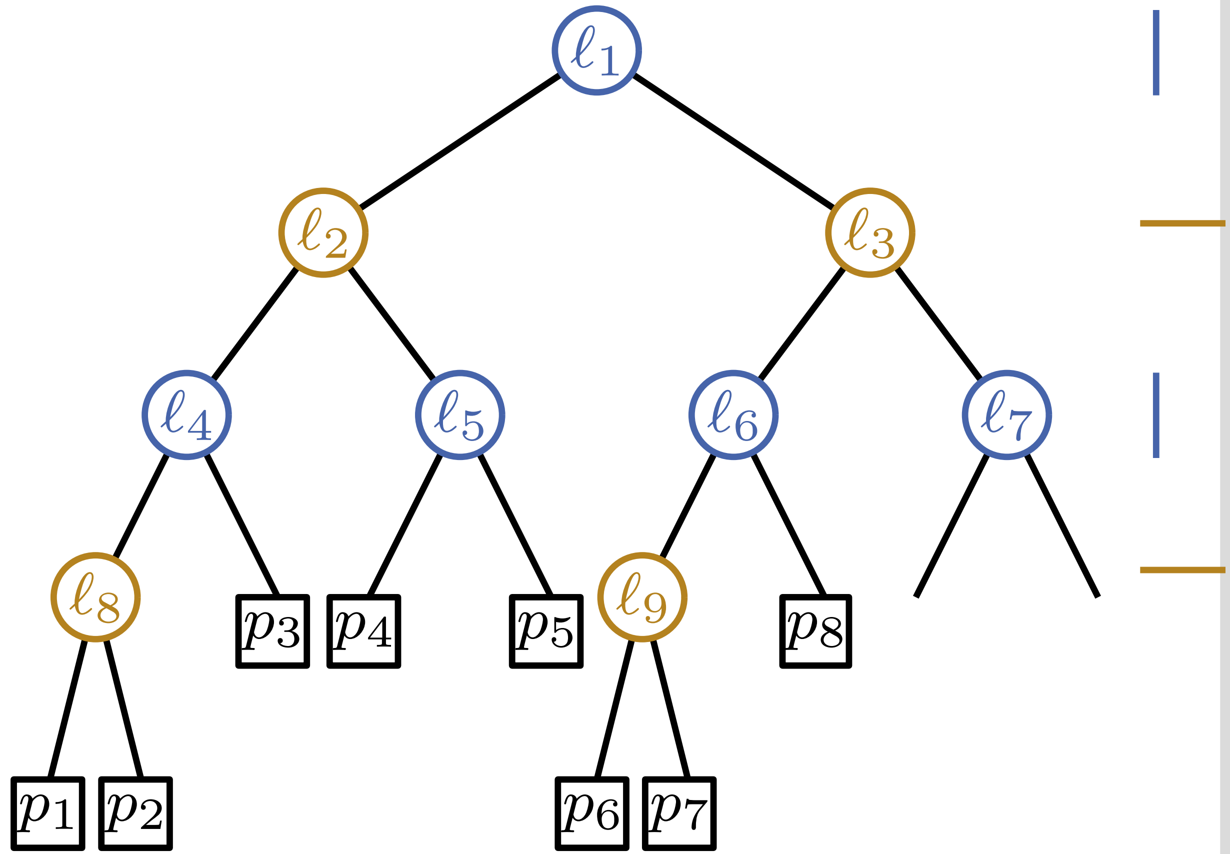


kd-Trees: Beispiel

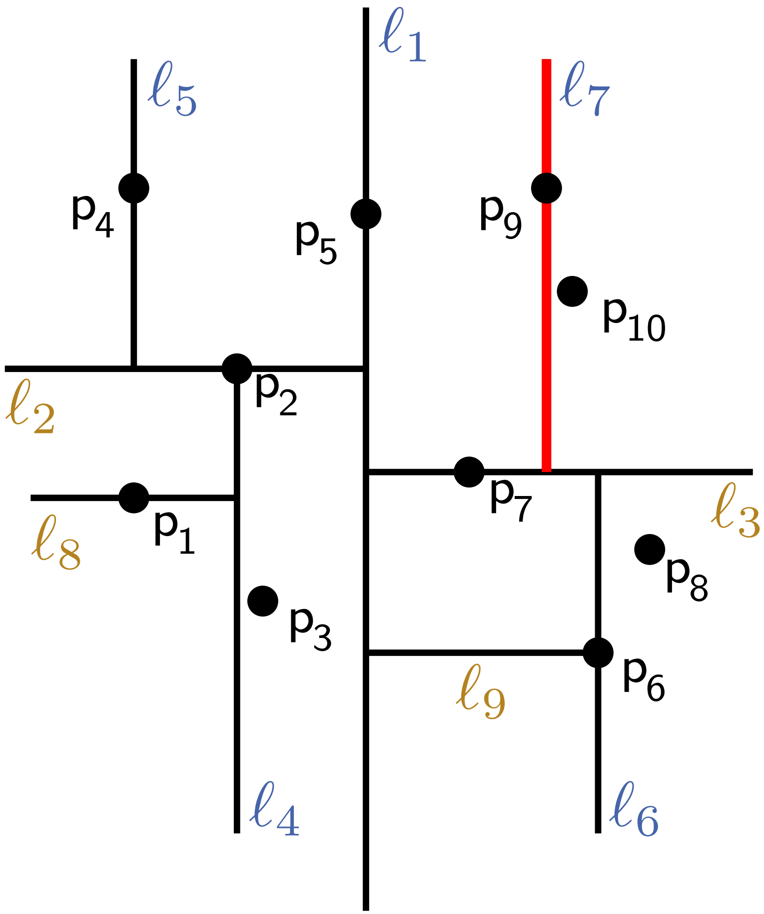


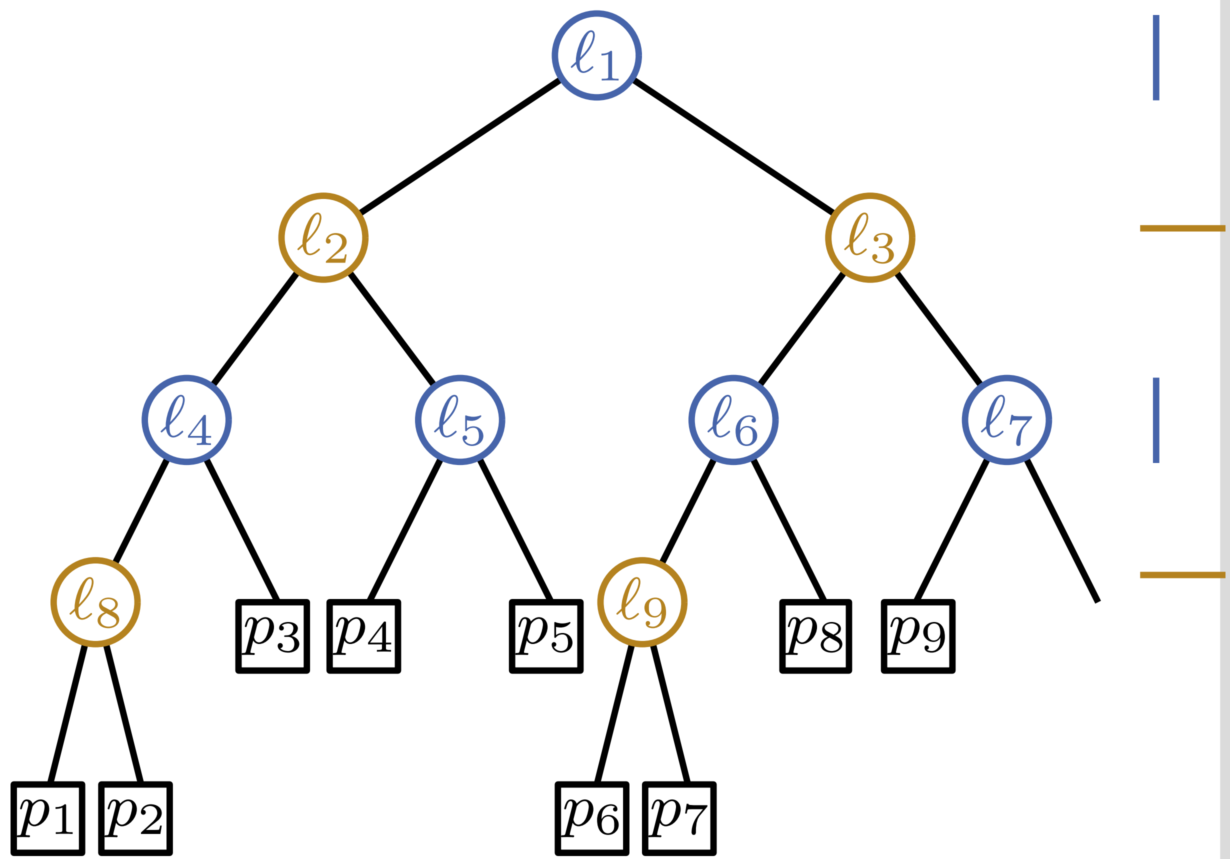
kd-Trees: Beispiel



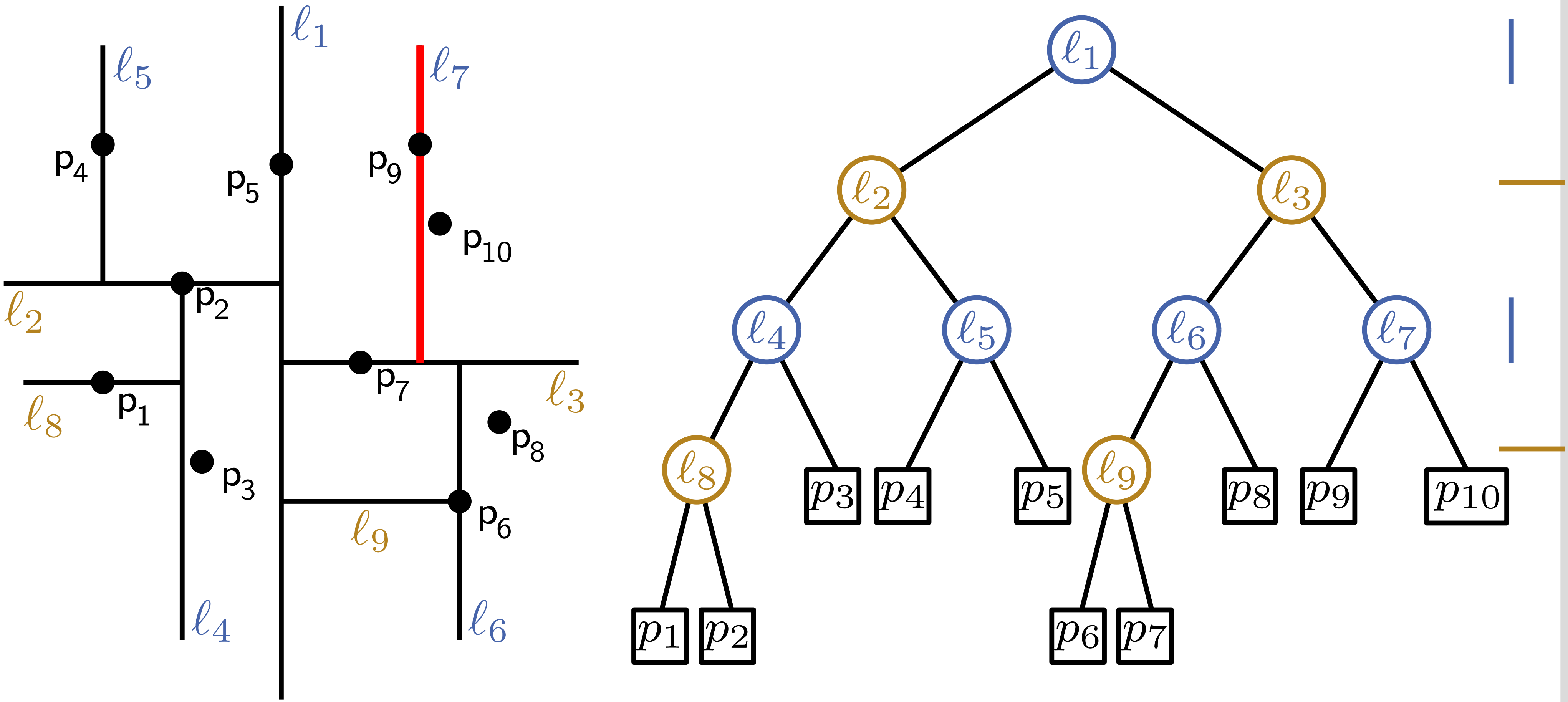


kd-Trees: Beispiel



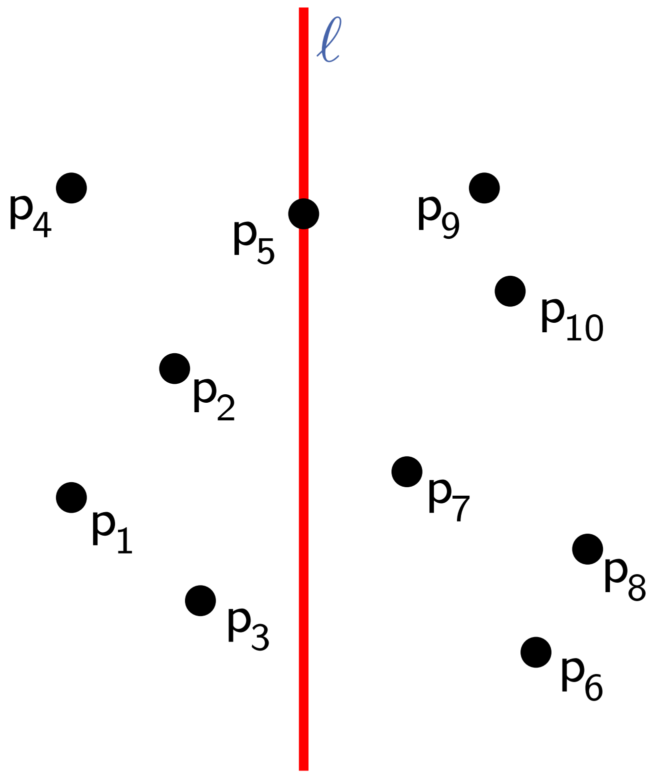


kd-Trees: Beispiel

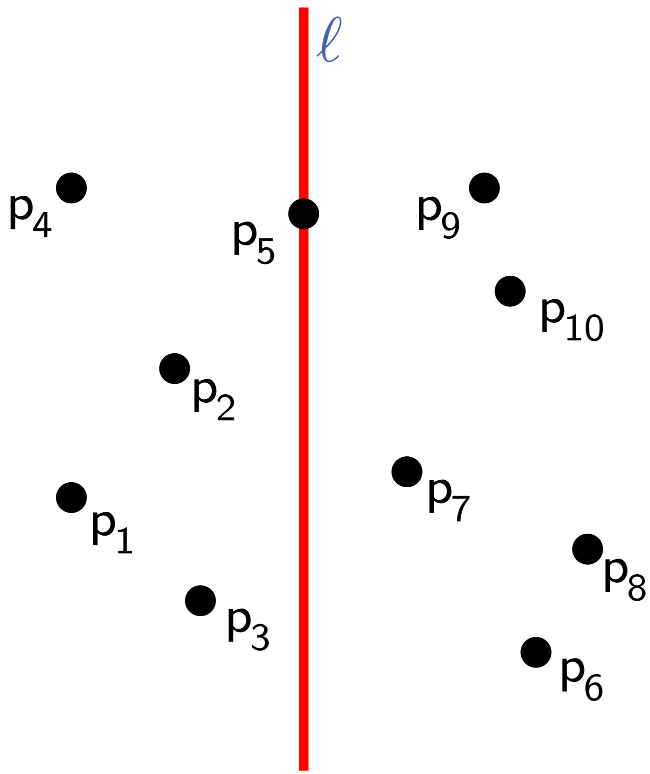


kd-Trees: Konstruktion

BuildKdTree(P, $depth$)



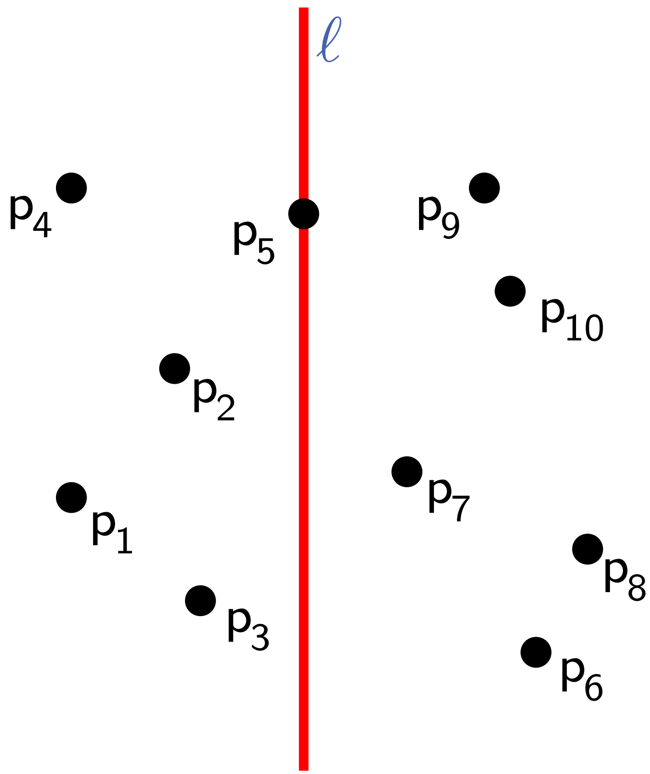
kd-Trees: Konstruktion



BuildKdTree$(P, depth)$

if $|P| = 1$ **then**
  return Blatt mit dem Punkt in P
else

kd-Trees: Konstruktion

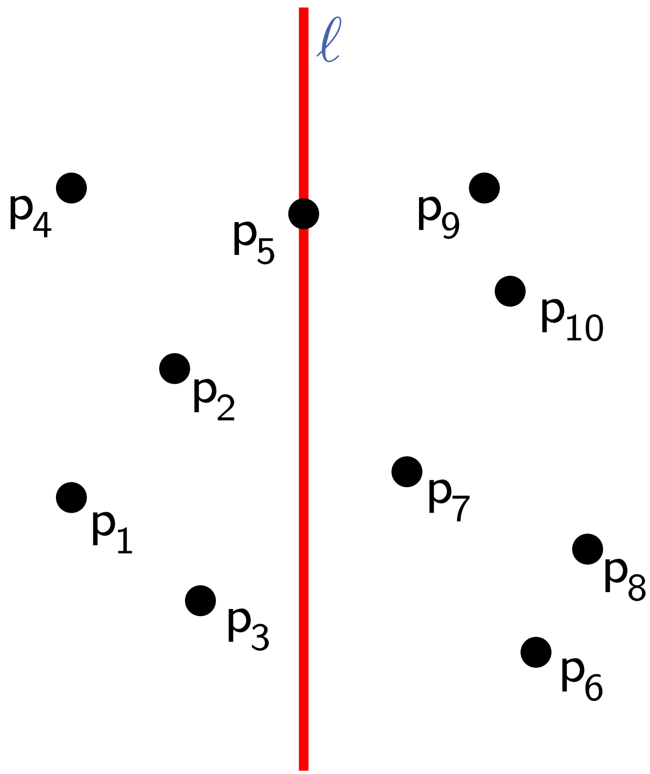


BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
  return Blatt mit dem Punkt in P
else
  if $depth$ gerade **then**

  else

kd-Trees: Konstruktion

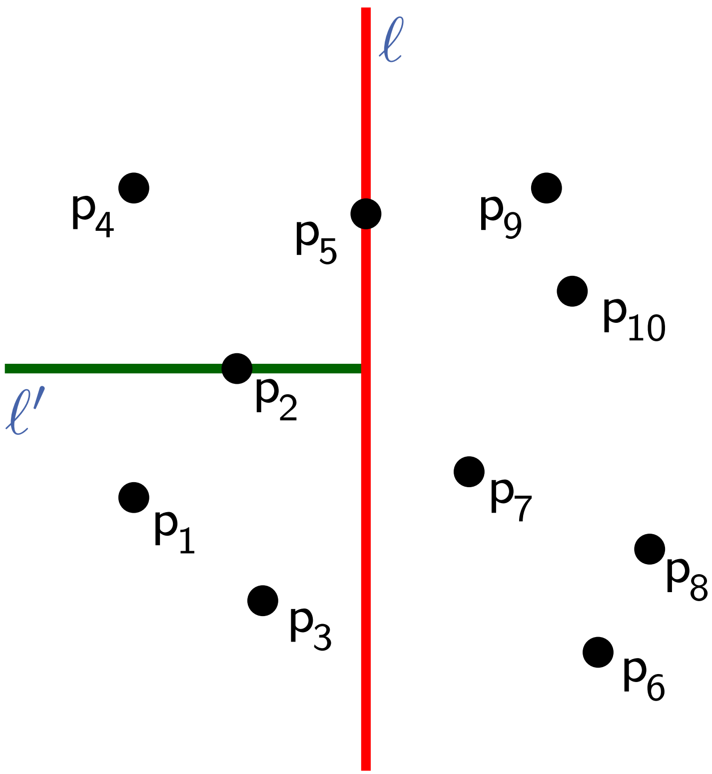


BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
  return Blatt mit dem Punkt in P
else
  if $depth$ gerade **then**
    teile P vertikal an
    $\ell : x = x_{\text{median}(P)}$ in
    P_1 (Punkte links von oder auf ℓ)
    und $P_2 = P \setminus P_1$
  else

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

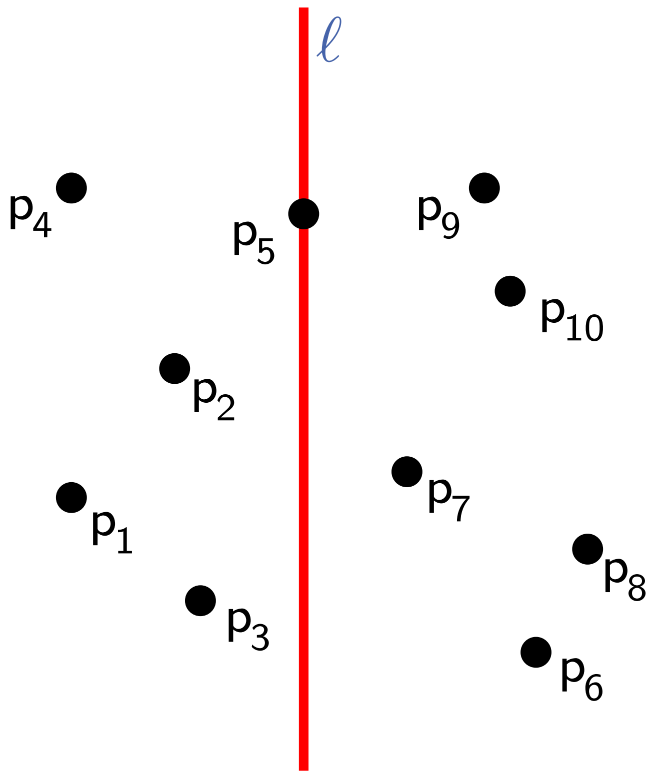
kd-Trees: Konstruktion



BuildKdTree$(P, depth)$

if $|P| = 1$ **then**
  return Blatt mit dem Punkt in P
else
  if $depth$ gerade **then**
    teile P vertikal an
    $\ell : x = x_{\text{median}(P)}$ in
    P_1 (Punkte links von oder auf ℓ)
    und $P_2 = P \setminus P_1$
  else
    teile P horizontal an
    $\ell : y = y_{\text{median}(P)}$ in
    P_1 (Punkte unter oder auf ℓ)
    und $P_2 = P \setminus P_1$

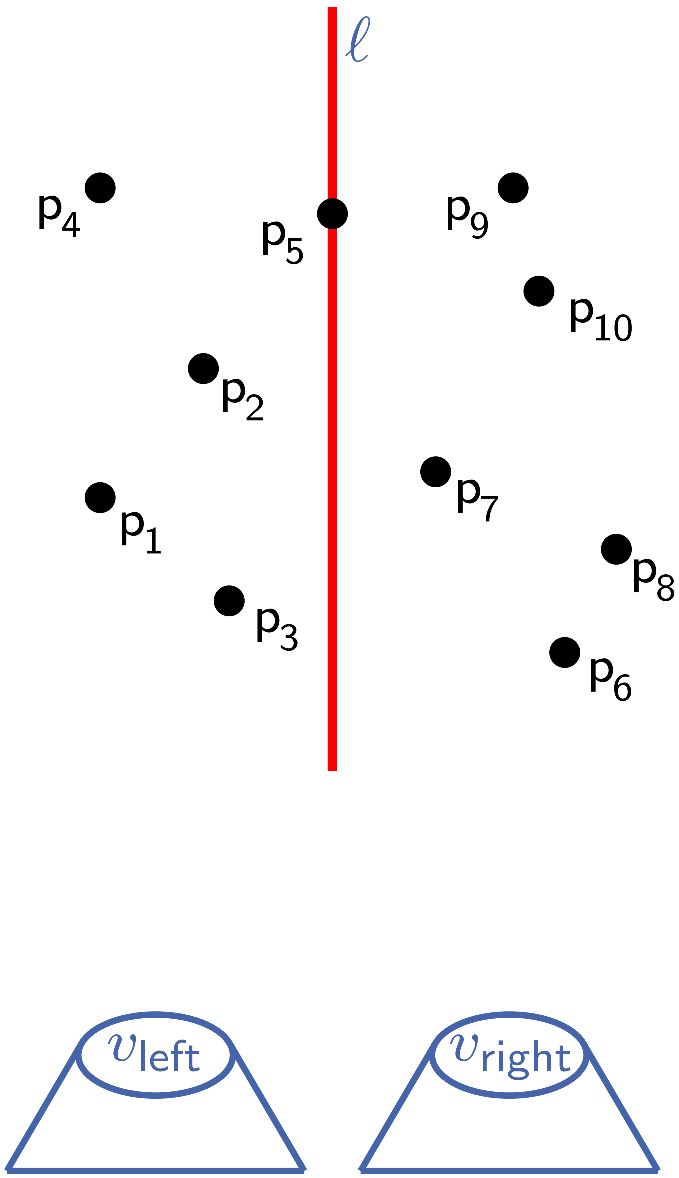
kd-Trees: Konstruktion



BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
  return Blatt mit dem Punkt in P
else
  if $depth$ gerade **then**
    teile P vertikal an
    $\ell : x = x_{\text{median}(P)}$ in
    P_1 (Punkte links von oder auf ℓ)
    und $P_2 = P \setminus P_1$
  else
    teile P horizontal an
    $\ell : y = y_{\text{median}(P)}$ in
    P_1 (Punkte unter oder auf ℓ)
    und $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildKdTree(P_1, $depth + 1$)
  $v_{\text{right}} \leftarrow$ BuildKdTree(P_2, $depth + 1$)

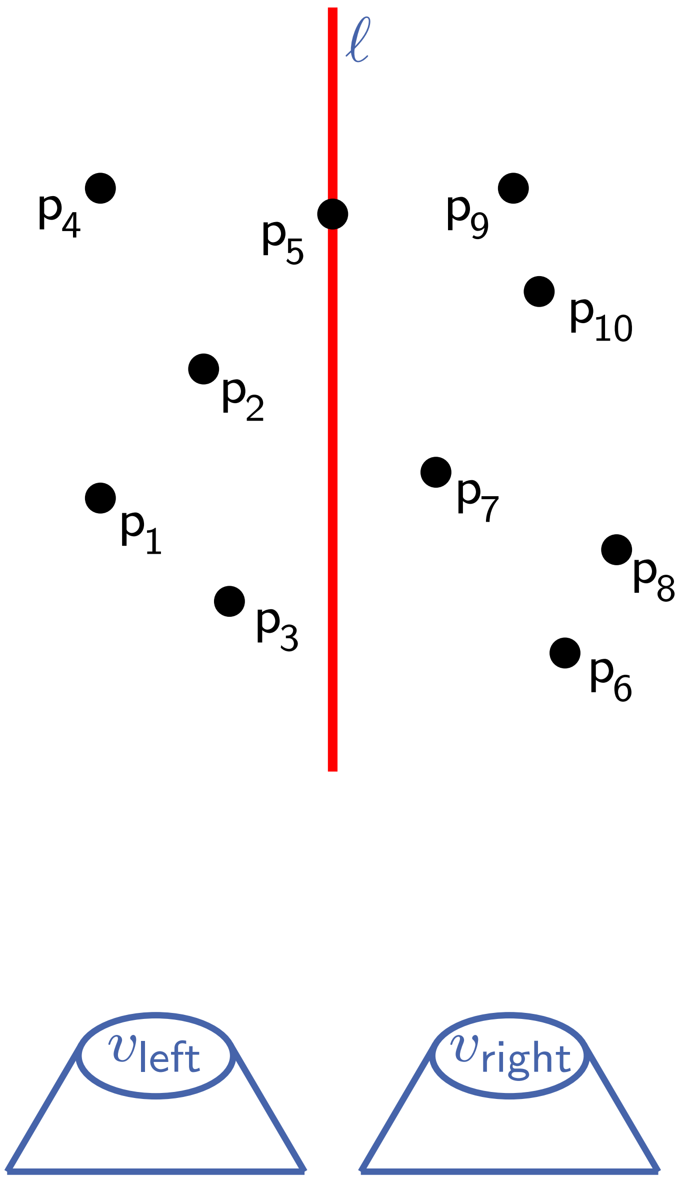
kd-Trees: Konstruktion



BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
  return Blatt mit dem Punkt in P
else
  if $depth$ gerade **then**
    teile P vertikal an
    $\ell : x = x_{\text{median}(P)}$ in
    P_1 (Punkte links von oder auf ℓ)
    und $P_2 = P \setminus P_1$
  else
    teile P horizontal an
    $\ell : y = y_{\text{median}(P)}$ in
    P_1 (Punkte unter oder auf 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: Konstruktion



BuildKdTree(P, $depth$)

if $|P| = 1$ **then**

  return Blatt mit dem Punkt in P

else

  if $depth$ gerade **then**

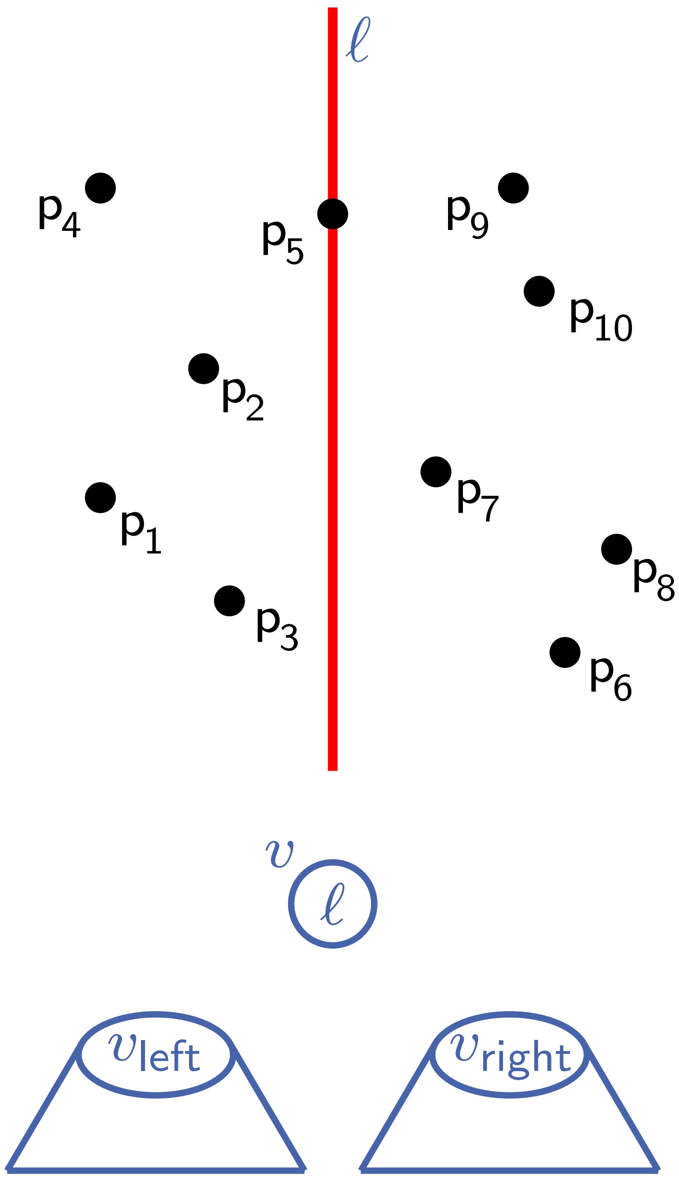
    teile P vertikal an
    $\ell : x = x_{\text{median}(P)}$ in
    P_1 (Punkte links von oder auf ℓ)
    und $P_2 = P \setminus P_1$

  else

    teile P horizontal an
    $\ell : y = y_{\text{median}(P)}$ in
    P_1 (Punkte unter oder auf ℓ)
    und $P_2 = P \setminus P_1$

  $v_{\text{left}} \leftarrow$ BuildKdTree(P_1, $depth + 1$)
  $v_{\text{right}} \leftarrow$ BuildKdTree(P_2, $depth + 1$)
  erzeuge Knoten v, der ℓ speichert

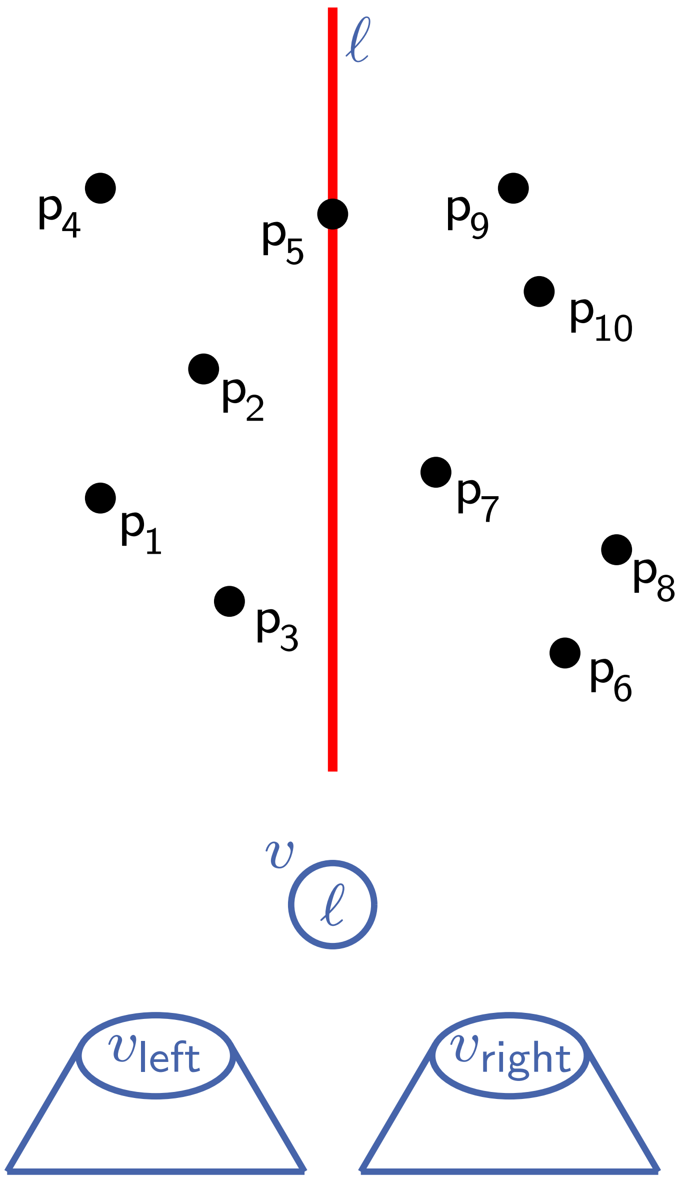
kd-Trees: Konstruktion



BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
  return Blatt mit dem Punkt in P
else
  if $depth$ gerade **then**
    teile P vertikal an
    $\ell : x = x_{\text{median}(P)}$ in
    P_1 (Punkte links von oder auf ℓ)
    und $P_2 = P \setminus P_1$
  else
    teile P horizontal an
    $\ell : y = y_{\text{median}(P)}$ in
    P_1 (Punkte unter oder auf ℓ)
    und $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildKdTree(P_1, $depth + 1$)
  $v_{\text{right}} \leftarrow$ BuildKdTree(P_2, $depth + 1$)
  erzeuge Knoten v, der ℓ speichert

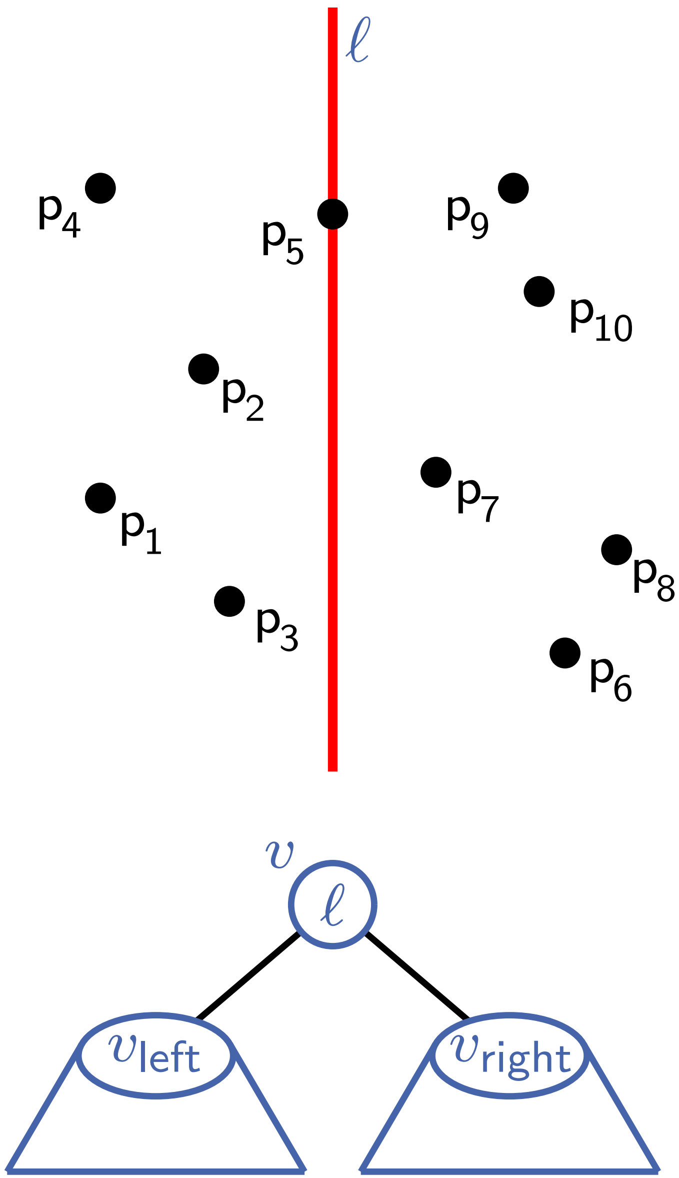
kd-Trees: Konstruktion



BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
  return Blatt mit dem Punkt in P
else
  if $depth$ gerade **then**
    teile P vertikal an
    $\ell : x = x_{\text{median}(P)}$ in
    P_1 (Punkte links von oder auf ℓ)
    und $P_2 = P \setminus P_1$
  else
    teile P horizontal an
    $\ell : y = y_{\text{median}(P)}$ in
    P_1 (Punkte unter oder auf ℓ)
    und $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildKdTree(P_1, $depth + 1$)
  $v_{\text{right}} \leftarrow$ BuildKdTree(P_2, $depth + 1$)
  erzeuge Knoten v, der ℓ speichert
  setze v_{left} und v_{right} als Kinder von v

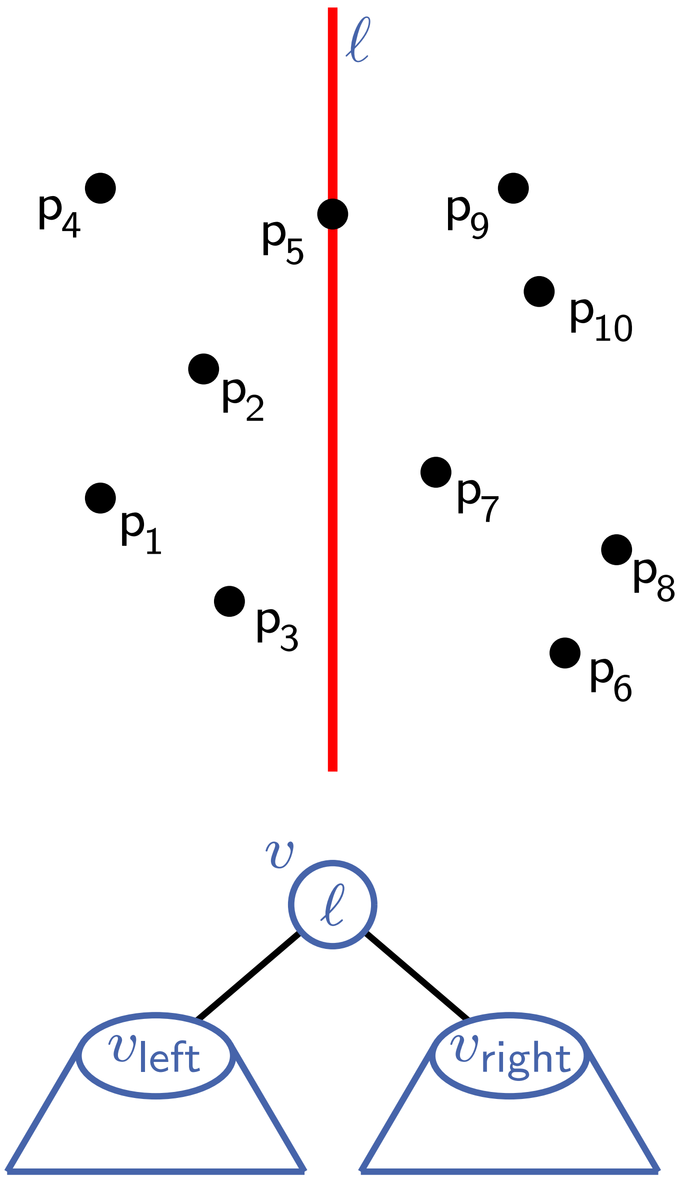
kd-Trees: Konstruktion



BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
  return Blatt mit dem Punkt in P
else
  if $depth$ gerade **then**
    teile P vertikal an
    $\ell : x = x_{\text{median}(P)}$ in
    P_1 (Punkte links von oder auf ℓ)
    und $P_2 = P \setminus P_1$
  else
    teile P horizontal an
    $\ell : y = y_{\text{median}(P)}$ in
    P_1 (Punkte unter oder auf ℓ)
    und $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildKdTree(P_1, $depth + 1$)
  $v_{\text{right}} \leftarrow$ BuildKdTree(P_2, $depth + 1$)
  erzeuge Knoten v, der ℓ speichert
  setze v_{left} und v_{right} als Kinder von v

kd-Trees: Konstruktion



BuildKdTree(P, $depth$)

if $|P| = 1$ **then**
    return Blatt mit dem Punkt in P
else
    if $depth$ gerade **then**
        teile P vertikal an
        $\ell : x = x_{\text{median}(P)}$ in
        P_1 (Punkte links von oder auf ℓ)
        und $P_2 = P \setminus P_1$
    else
        teile P horizontal an
        $\ell : y = y_{\text{median}(P)}$ in
        P_1 (Punkte unter oder auf ℓ)
        und $P_2 = P \setminus P_1$
    $v_{\text{left}} \leftarrow$ BuildKdTree(P_1, $depth + 1$)
    $v_{\text{right}} \leftarrow$ BuildKdTree(P_2, $depth + 1$)
    erzeuge Knoten v, der ℓ speichert
    setze v_{left} und v_{right} als Kinder von v
    return v

Analyse Konstruktion kd-Trees

Lemma: Ein kd-Tree für n Punkte in $\mathbb{R}^2$ kann in $O(n \log n)$ Zeit konstruiert werden und benötigt $O(n)$ Platz.

Analyse Konstruktion kd-Trees

Lemma: Ein kd-Tree für n Punkte in $\mathbb{R}^2$ kann in $O(n \log n)$ Zeit konstruiert werden und benötigt $O(n)$ Platz.

Beweisskizze:

- Median bestimmen:
  initial zwei sortierte Listen nach x- und y-Koordinaten
  dann in jedem Schritt Median suchen und Listen aufteilen

Analyse Konstruktion kd-Trees

Lemma: Ein kd-Tree für n Punkte in $\mathbb{R}^2$ kann in $O(n \log n)$ Zeit konstruiert werden und benötigt $O(n)$ Platz.

Beweisskizze:

- Median bestimmen:
  initial zwei sortierte Listen nach x- und y-Koordinaten
  dann in jedem Schritt Median suchen und Listen aufteilen
- damit folgende Laufzeit-Rekurrenz:

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

Analyse Konstruktion kd-Trees

Lemma: Ein kd-Tree für n Punkte in $\mathbb{R}^2$ kann in $O(n \log n)$ Zeit konstruiert werden und benötigt $O(n)$ Platz.

Beweisskizze:

- Median bestimmen:
  initial zwei sortierte Listen nach x- und y-Koordinaten
  dann in jedem Schritt Median suchen und Listen aufteilen
- damit folgende Laufzeit-Rekurrenz:

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

- wird gelöst zu $T(n) = O(n \log n)$ (analog MergeSort o.ä.)

Analyse Konstruktion kd-Trees

Lemma: Ein kd-Tree für n Punkte in $\mathbb{R}^2$ kann in $O(n \log n)$ Zeit konstruiert werden und benötigt $O(n)$ Platz.

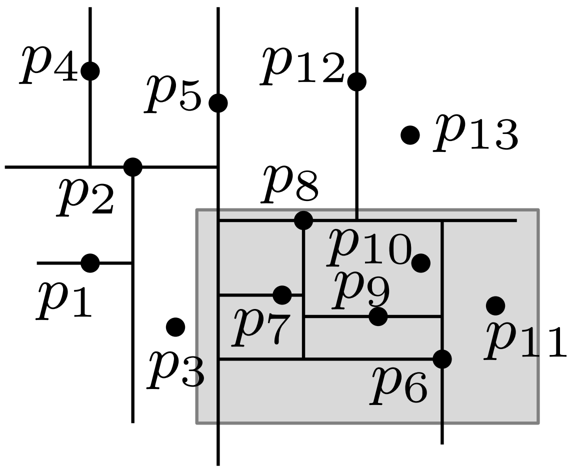
Beweisskizze:

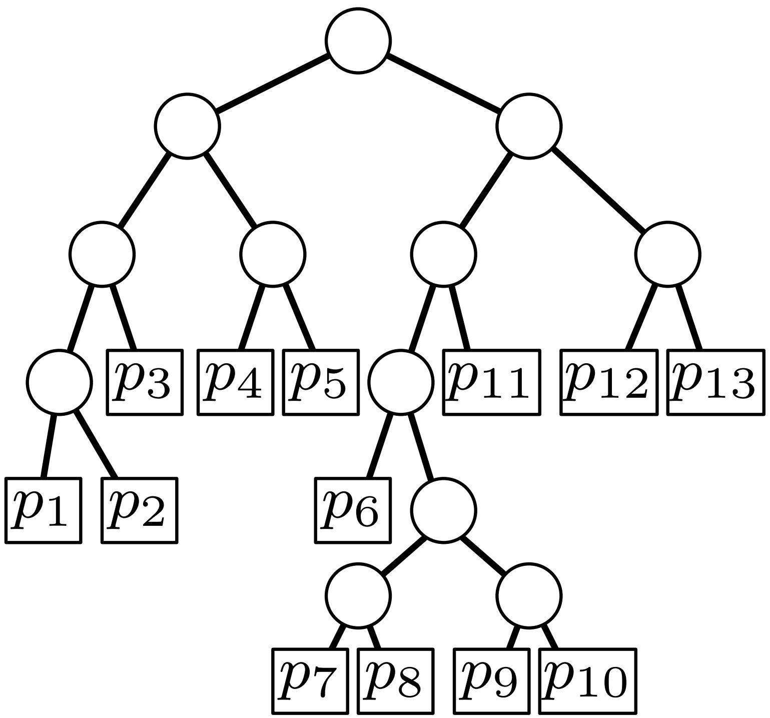
- Median bestimmen:
  initial zwei sortierte Listen nach x- und y-Koordinaten
  dann in jedem Schritt Median suchen und Listen aufteilen
- damit folgende Laufzeit-Rekurrenz:

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

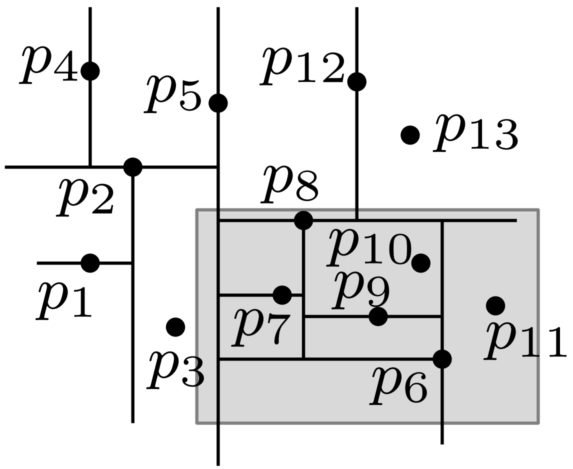
- wird gelöst zu $T(n) = O(n \log n)$ (analog MergeSort o.ä.)
- Platzbedarf linear da Binärbaum mit n Blättern

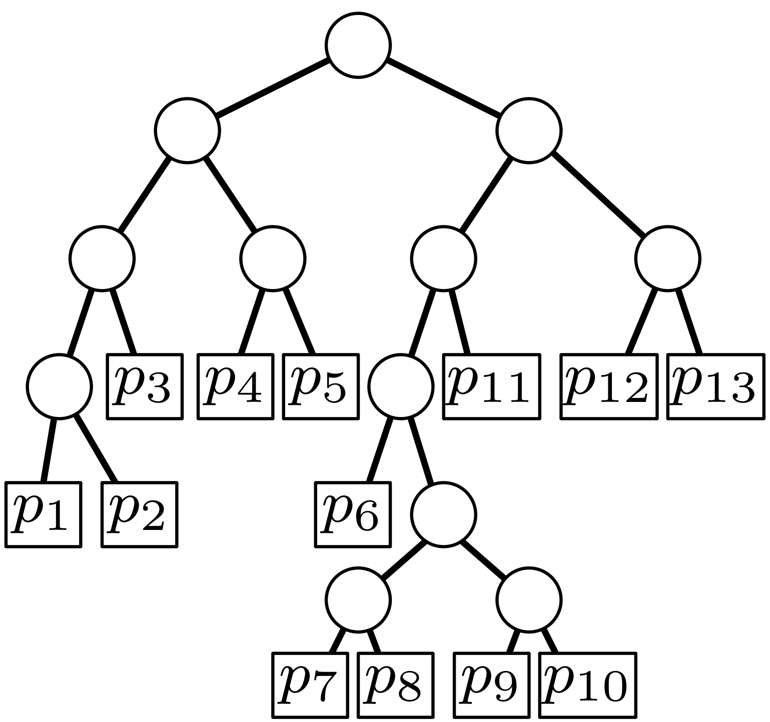
Bereichsabfrage in einem kd-Tree





Bereichsabfrage in einem kd-Tree

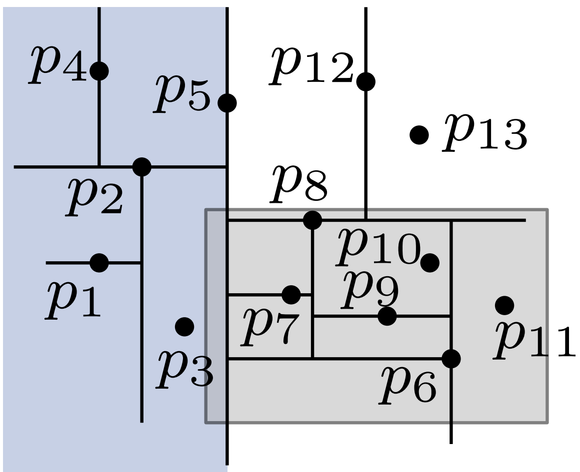


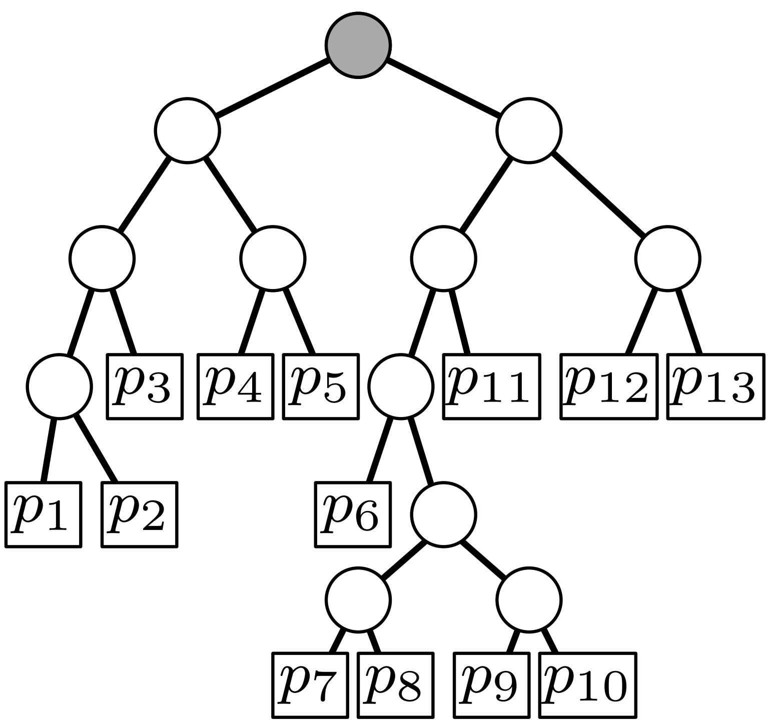


SearchKdTree(v, R)

- **if** v Blatt **then**
  - prüfe Punkt p in v auf $p \in R$
- **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)

Bereichsabfrage in einem kd-Tree

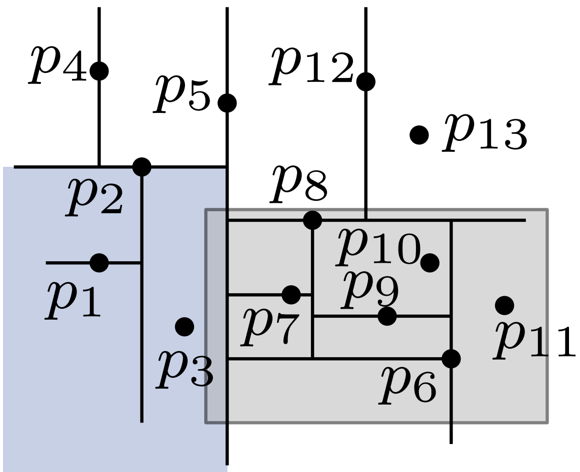


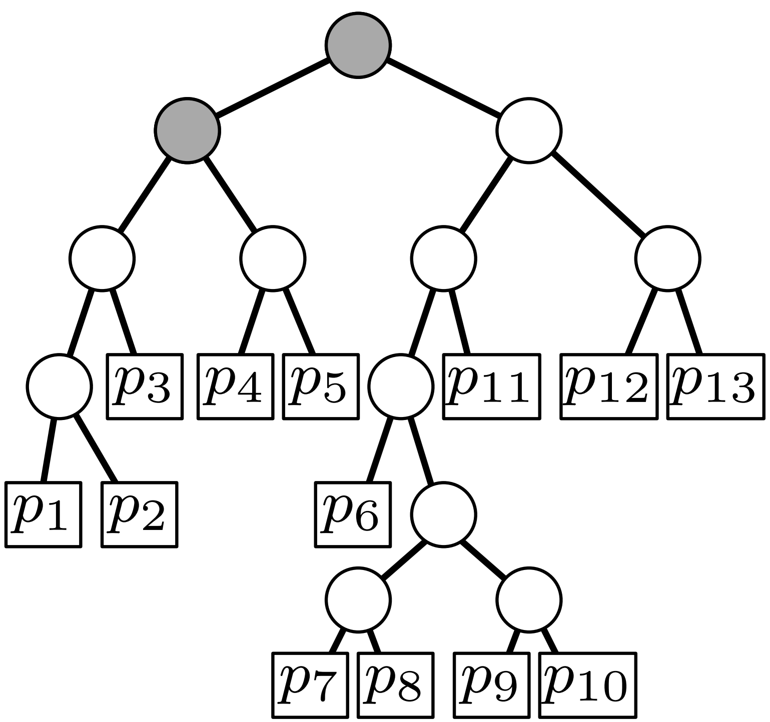


SearchKdTree(v, R)

- **if** v Blatt **then**
  - prüfe Punkt p in v auf $p \in R$
- **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)

Bereichsabfrage in einem kd-Tree

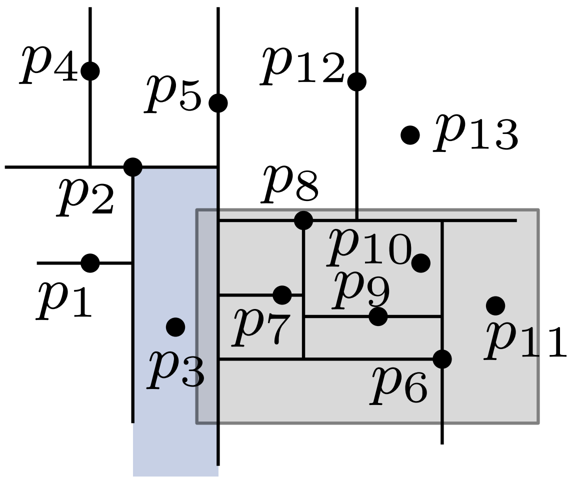


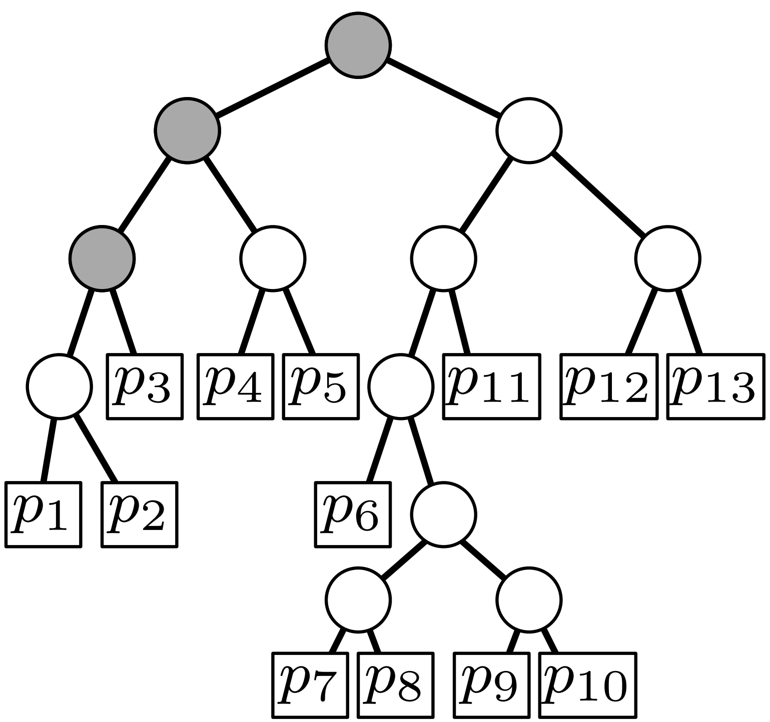


SearchKdTree(v, R)

- **if** v Blatt **then**
  - prüfe Punkt p in v auf $p \in R$
- **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)

Bereichsabfrage in einem kd-Tree

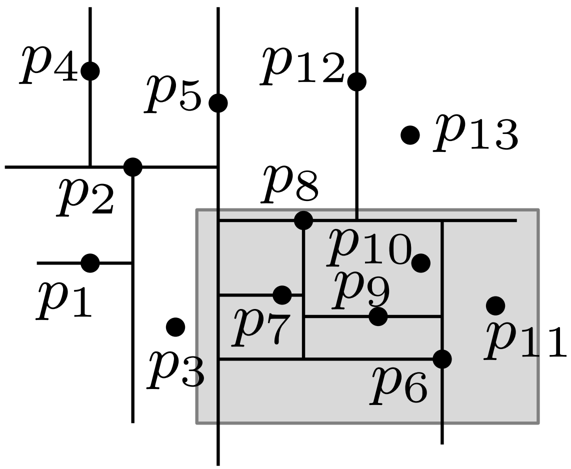


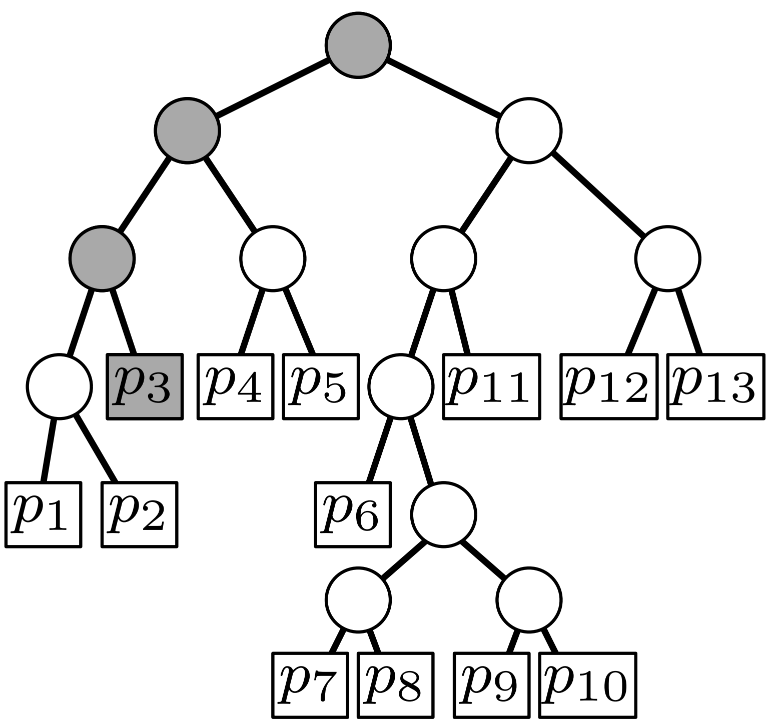


SearchKdTree(v, R)

if v Blatt **then**
  prüfe Punkt p in v auf $p \in R$
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)

Bereichsabfrage in einem kd-Tree

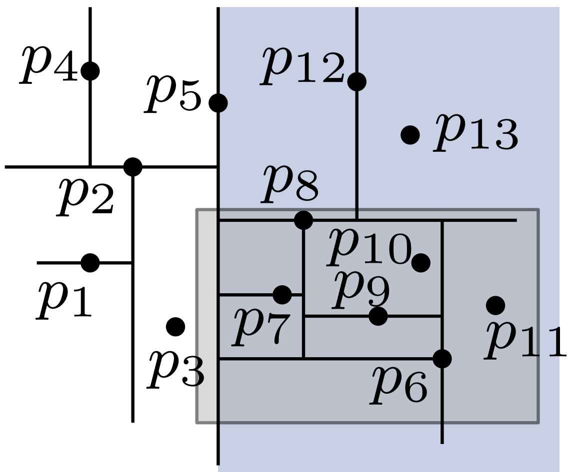


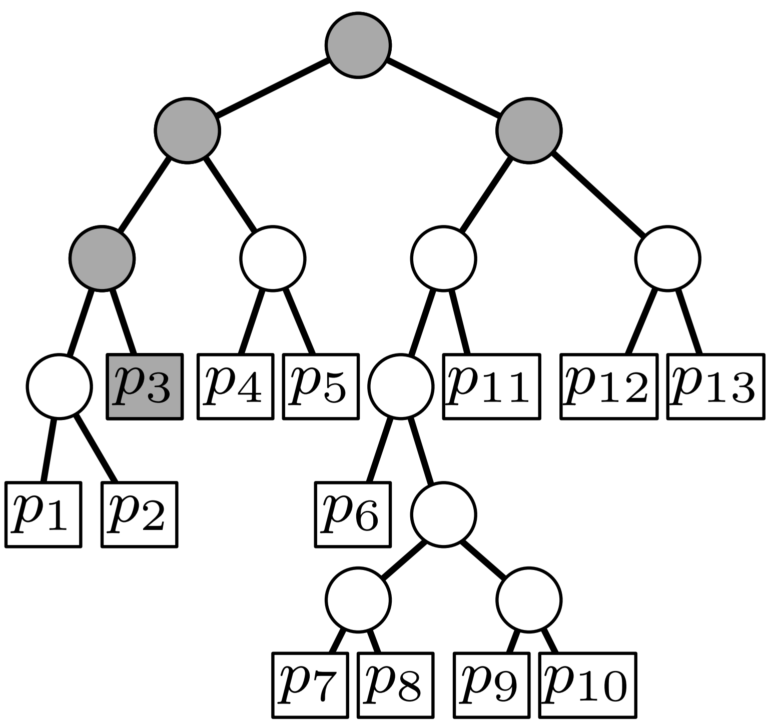


SearchKdTree(v, R)

if v Blatt **then**
  prüfe Punkt p in v auf $p \in R$
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)

Bereichsabfrage in einem kd-Tree

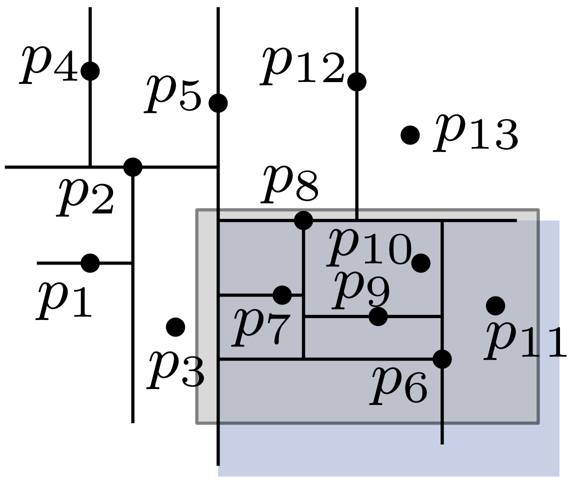


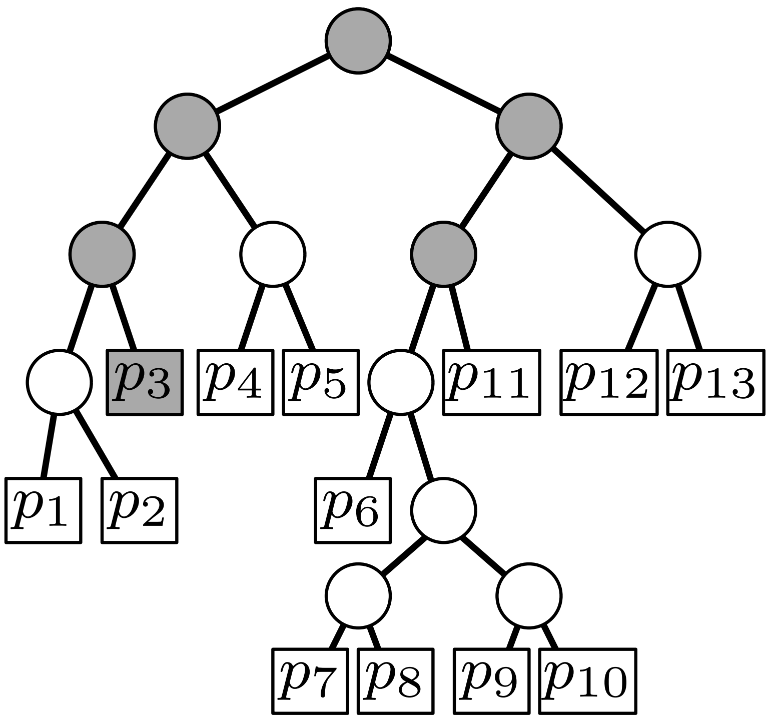


SearchKdTree(v, R)

- **if** v Blatt **then**
  - prüfe Punkt p in v auf $p \in R$
- **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)

Bereichsabfrage in einem kd-Tree

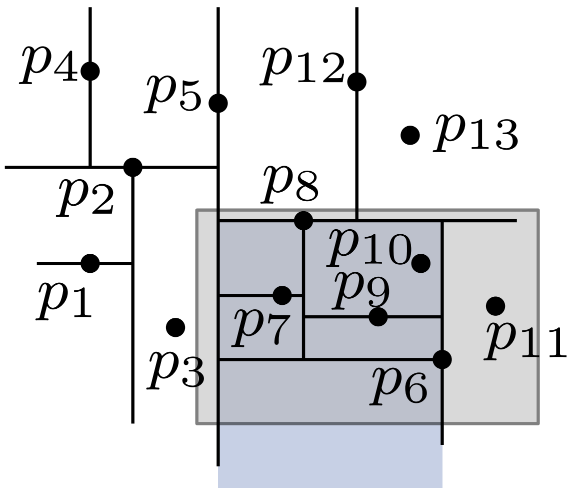


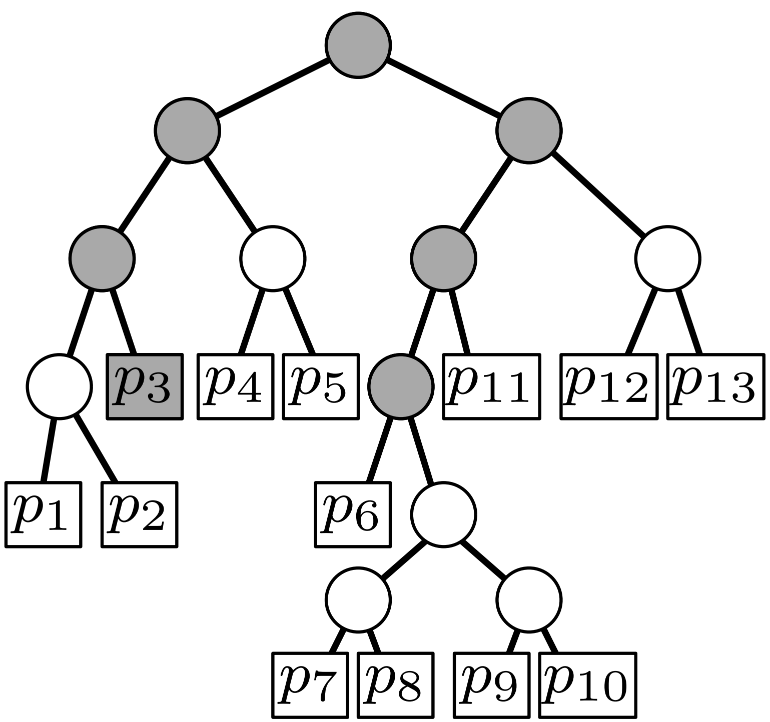


SearchKdTree(v, R)

- **if** v Blatt **then**
  - prüfe Punkt p in v auf $p \in R$
- **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)

Bereichsabfrage in einem kd-Tree

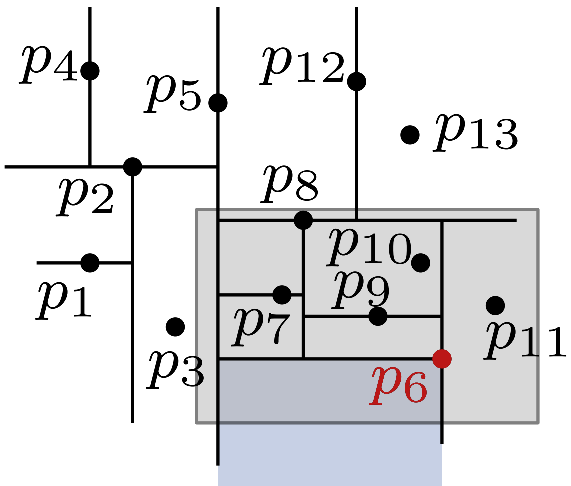


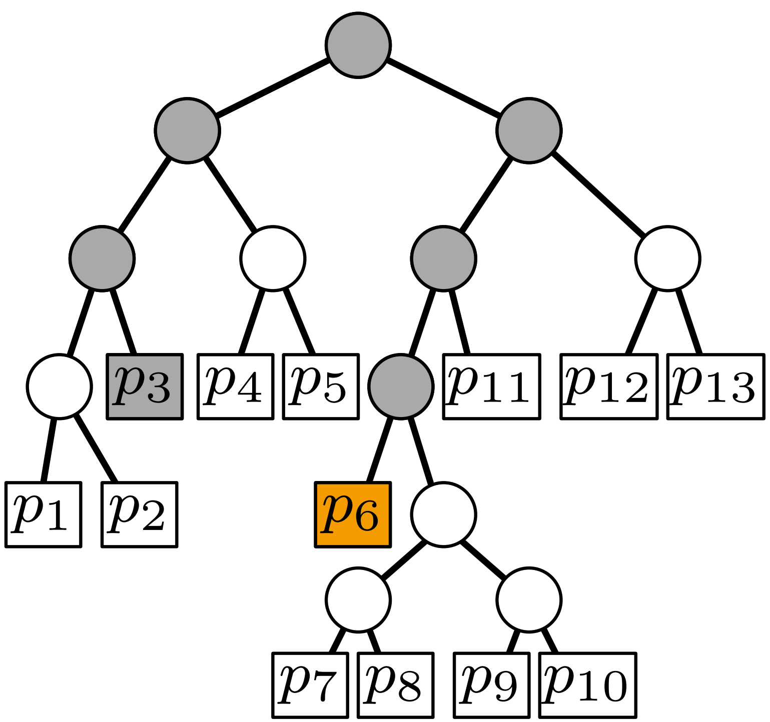


SearchKdTree(v, R)

if v Blatt **then**
  prüfe Punkt p in v auf $p \in R$
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)

Bereichsabfrage in einem kd-Tree

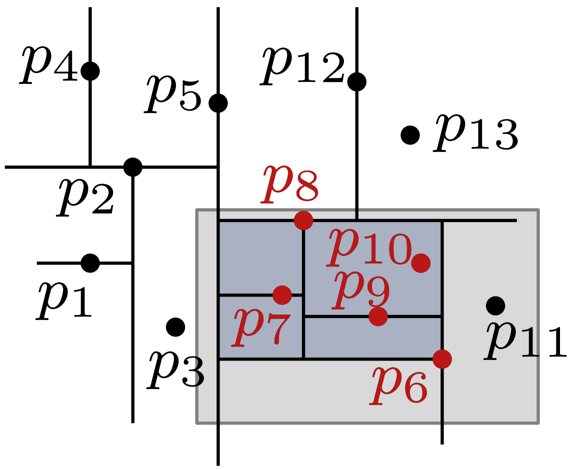


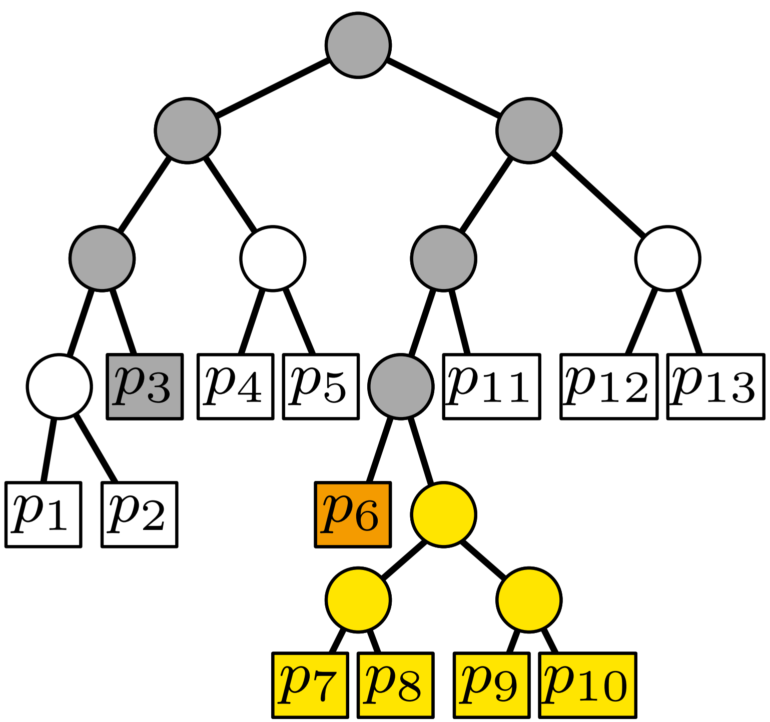


SearchKdTree(v, R)

- **if** v Blatt **then**
  - prüfe Punkt p in v auf $p \in R$
- **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)

Bereichsabfrage in einem kd-Tree

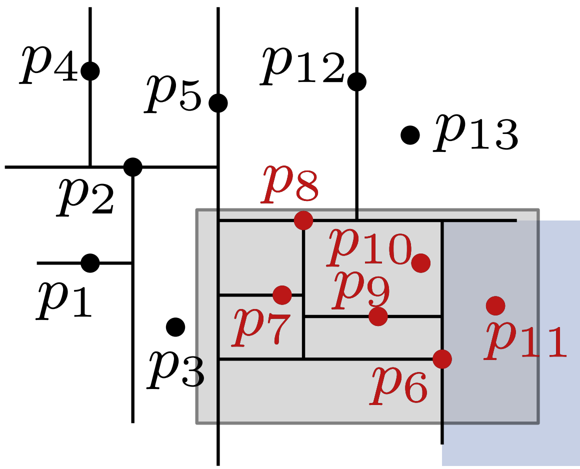


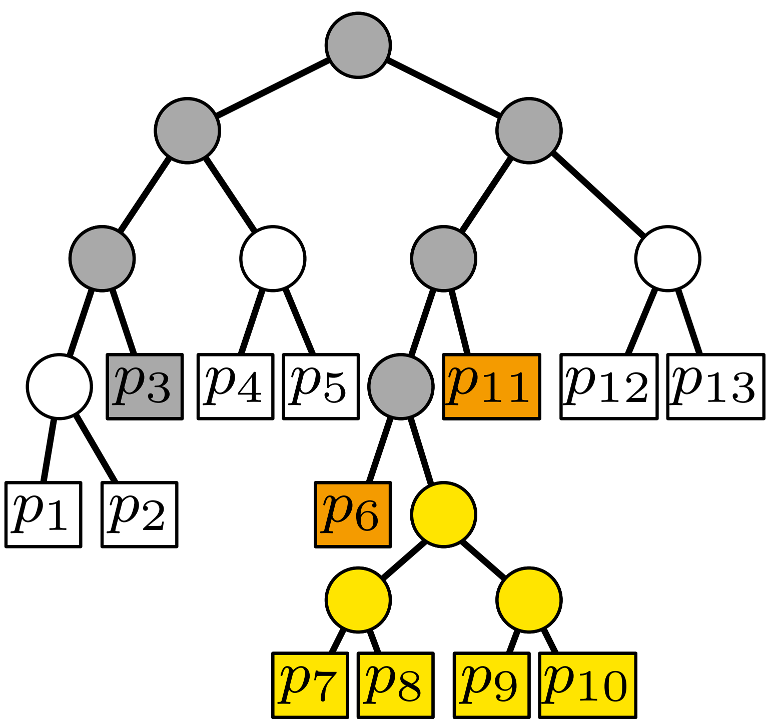


SearchKdTree(v, R)

- **if** v Blatt **then**
  - prüfe Punkt p in v auf $p \in R$
- **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)

Bereichsabfrage in einem kd-Tree

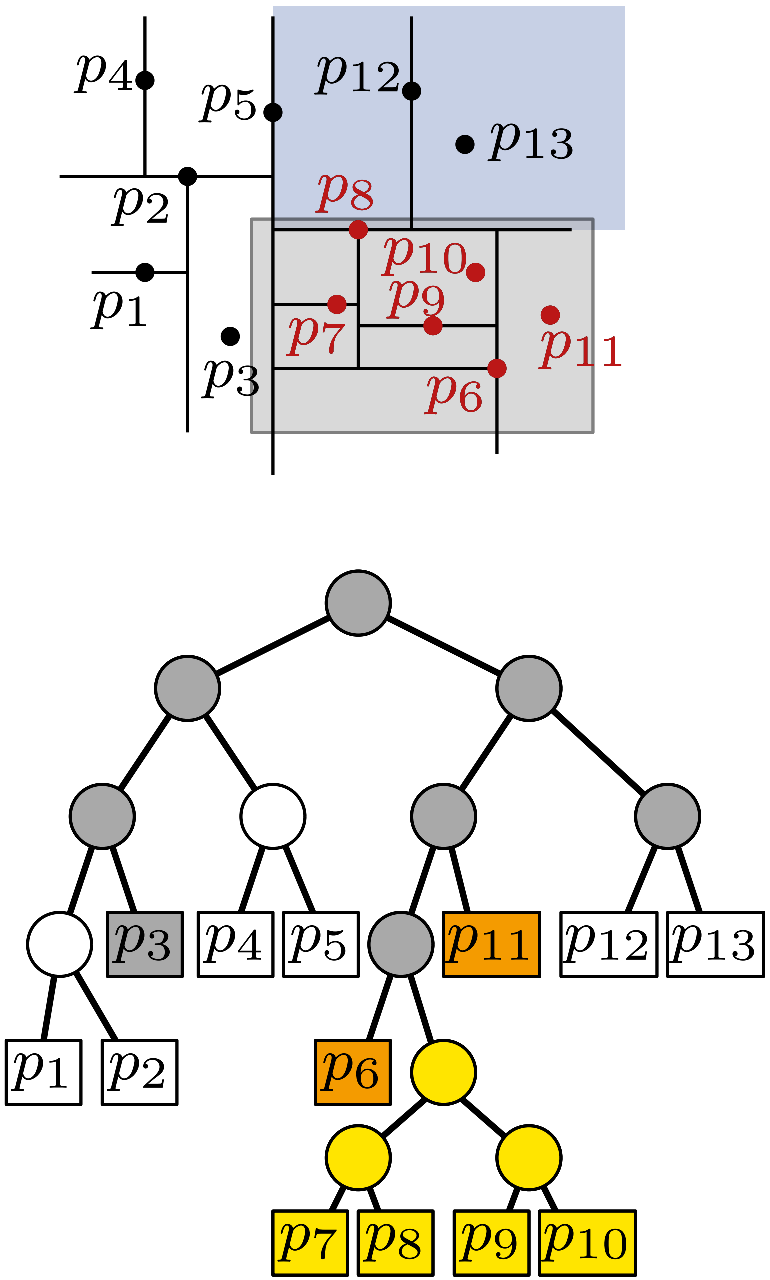




SearchKdTree(v, R)

- **if** v Blatt **then**
  - prüfe Punkt p in v auf $p \in R$
- **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)

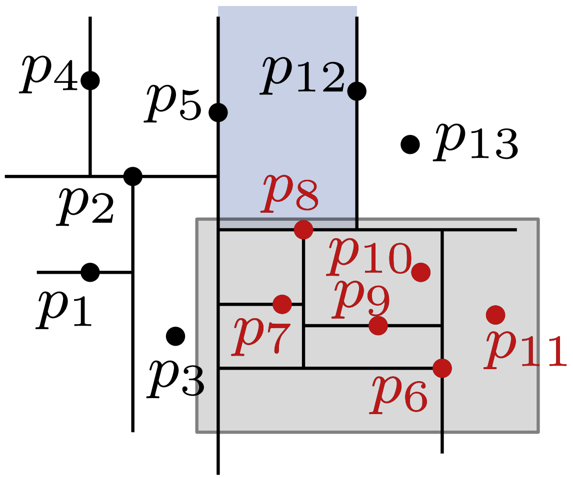
Bereichsabfrage in einem *kd*-Tree

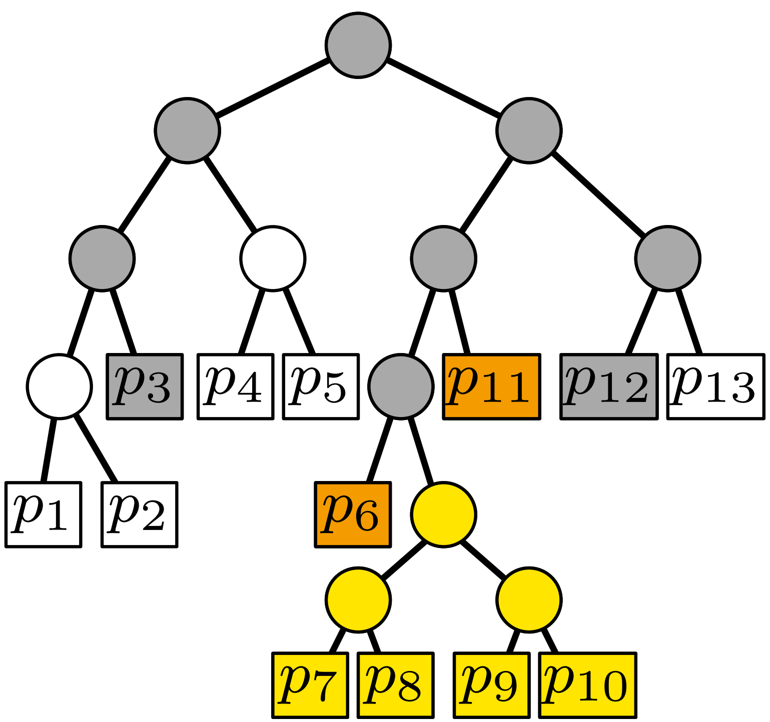


SearchKdTree(v, R)

- **if** v Blatt **then**
  - prüfe Punkt p in v auf $p \in R$
- **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)

Bereichsabfrage in einem kd-Tree

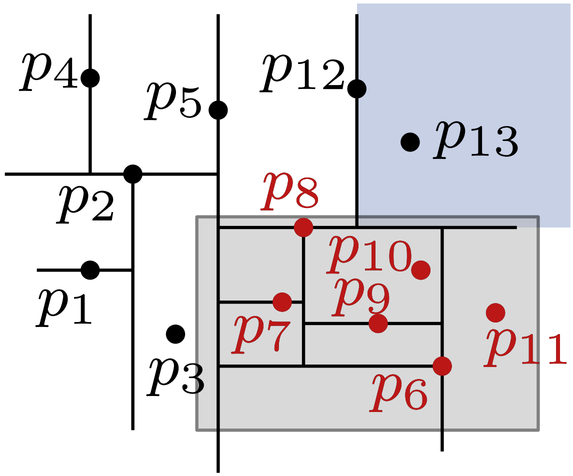


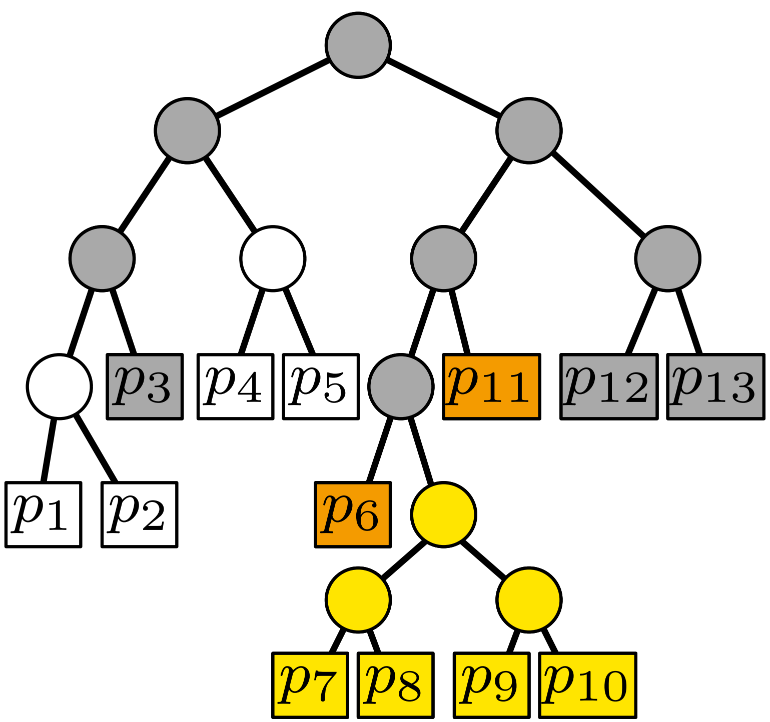


SearchKdTree(v, R)

if v Blatt **then**
  prüfe Punkt p in v auf $p \in R$
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)

Bereichsabfrage in einem kd-Tree





SearchKdTree(v, R)

if v Blatt **then**
  prüfe Punkt p in v auf $p \in R$
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)

Analyse Abfrage kd-Tree

Lemma: Eine Bereichsabfrage mit einem achsenparallelen Rechteck R in einem kd-Tree für n Punkte benötigt $O(\sqrt{n} + k)$ Zeit, wobei k die Antwortgröße ist.

Analyse Abfrage kd-Tree

Lemma: Eine Bereichsabfrage mit einem achsenparallelen Rechteck R in einem kd-Tree für n Punkte benötigt $O(\sqrt{n} + k)$ Zeit, wobei k die Antwortgröße ist.

Beweisskizze:

- Aufrufe von ReportSubtree benötigen insgesamt $O(k)$ Zeit

Analyse Abfrage kd-Tree

Lemma: Eine Bereichsabfrage mit einem achsenparallelen Rechteck R in einem kd-Tree für n Punkte benötigt $O(\sqrt{n} + k)$ Zeit, wobei k die Antwortgröße ist.

Beweisskizze:

- Aufrufe von ReportSubtree benötigen insgesamt $O(k)$ Zeit
- fehlt noch:
  Anzahl der übrigen besuchten Knoten abschätzen
  $\rightarrow$ Übungsblatt

Orthogonale Bereichsabfragen für $d = 2$

Geg: Menge P von n Punkten in $\mathbb{R}^2$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $R = [x, x'] \times [y, y']$

Ideen zur Verallgemeinerung des 1d Falls?

Lösungsansätze:

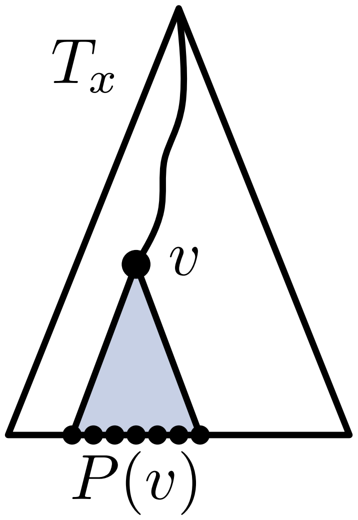
- *ein* Suchbaum, der abwechselnd nach x- und y-Koordinaten trennt $\rightarrow$ kd-**Tree** ✓
- *ein* Suchbaum für x-Koordinaten, *mehrere* untergeordnete Suchbäume für y-Koordinaten $\rightarrow$ **Range-Tree**

vorübergehende Annahme: allgemeine Lage, d.h. keine zwei Punkte haben gleiche x- oder y-Koordinate

Range Trees

Idee: Nutze eindimensionale binäre Suchbäume auf zwei Ebenen:

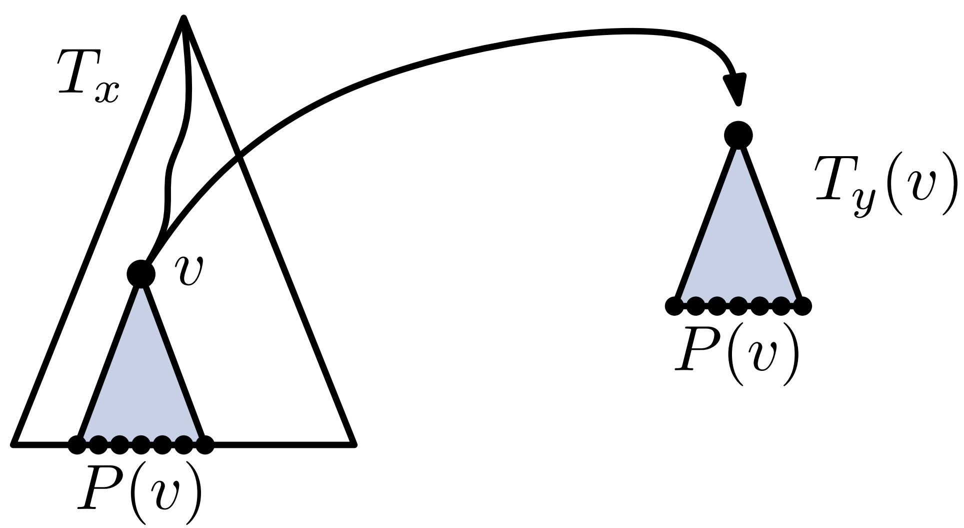
- ein 1d Suchbaum T_x bzgl. x-Koordinaten



Range Trees

Idee: Nutze eindimensionale binäre Suchbäume auf zwei Ebenen:

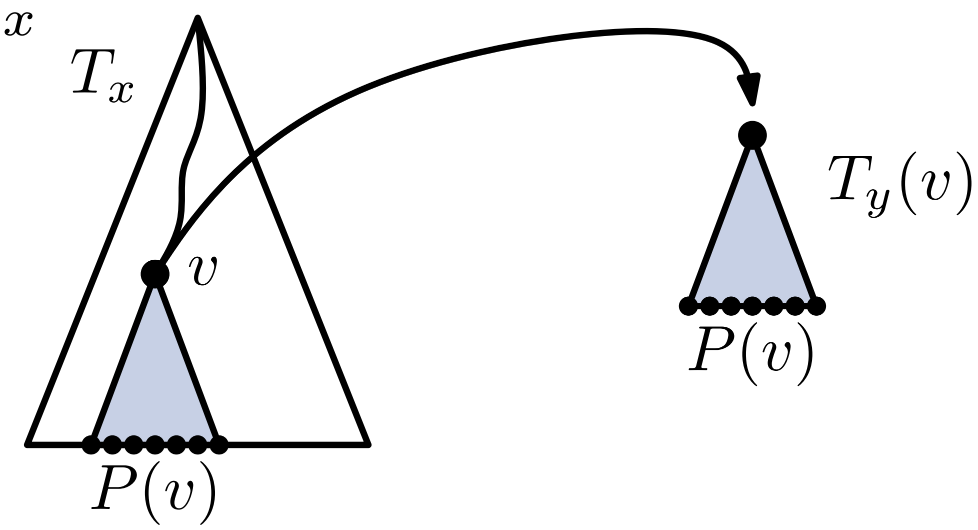
- ein 1d Suchbaum T_x bzgl. x-Koordinaten
- in jedem Knoten v von T_x einen 1d Suchbaum $T_y(v)$ zum Speichern der kanonischen Blattmenge $P(v)$ bzgl. y-Koordinaten



Range Trees

Idee: Nutze eindimensionale binäre Suchbäume auf zwei Ebenen:

- ein 1d Suchbaum T_x bzgl. x-Koordinaten
- in jedem Knoten v von T_x einen 1d Suchbaum $T_y(v)$ zum Speichern der kanonischen Blattmenge $P(v)$ bzgl. y-Koordinaten
- bestimme Lösungsmenge durch x-Abfrage in T_x und anschließender y-Abfrage in den Hilfsstrukturen T_y der Teilbäume in T_x



Range Trees: Konstruktion

BuildRangeTree(P)

if $|P| = 1$ **then**
  erzeuge Blatt v für den Punkt in P
else
  teile P an x_{median} in $P_1 = \{p \in P \mid p_x \leq x_{\text{median}}\}$ und $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildRangeTree(P_1)
  $v_{\text{right}} \leftarrow$ BuildRangeTree(P_2)
  erzeuge Knoten v mit Pivot x_{median} und Kindern v_{left} und v_{right}
$T_y(v) \leftarrow$ binärer Suchbaum für P bzgl. y-Koordinaten
return v

Range Trees: Konstruktion

BuildRangeTree(P)

if $|P| = 1$ **then**
- erzeuge Blatt v für den Punkt in P

else
- teile P an x_{median} in $P_1 = \{p \in P \mid p_x \leq x_{\text{median}}\}$ und $P_2 = P \setminus P_1$
- $v_{\text{left}} \leftarrow$ BuildRangeTree(P_1)
- $v_{\text{right}} \leftarrow$ BuildRangeTree(P_2)
- erzeuge Knoten v mit Pivot x_{median} und Kindern v_{left} und v_{right}

$T_y(v) \leftarrow$ binärer Suchbaum für P bzgl. y-Koordinaten

return v

Aufgabe: Wieviel Speicher und Laufzeit benötigt BuildRangeTree?

Range Trees: Konstruktion

BuildRangeTree(P)

if $|P| = 1$ **then**
  erzeuge Blatt v für den Punkt in P
else
  teile P an x_{median} in $P_1 = \{p \in P \mid p_x \leq x_{\text{median}}\}$ und $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildRangeTree(P_1)
  $v_{\text{right}} \leftarrow$ BuildRangeTree(P_2)
  erzeuge Knoten v mit Pivot x_{median} und Kindern v_{left} und v_{right}
$T_y(v) \leftarrow$ binärer Suchbaum für P bzgl. y-Koordinaten
return v

Aufgabe: Wieviel Speicher und Laufzeit benötigt BuildRangeTree?

Lemma: Ein Range Tree für n Punkte in $\mathbb{R}^2$ benötigt $O(n \log n)$ Platz und kann in $O(n \log n)$ Zeit konstruiert werden.

Bereichsabfrage in einem Range Tree

Erinnerung:

1dRangeQuery(T, x, x')

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} ist Blatt **then** prüfe v_{split}
else

$v \leftarrow$ lc(v_{split})
while v kein Blatt **do**
  if $x \leq x_v$ **then**
    ReportSubtree(rc(v))
    $v \leftarrow$ lc(v)
  else $v \leftarrow$ rc(v)
prüfe v
`// analog für` x' `und rc(`$v_{\texttt{split}}$`)`

Bereichsabfrage in einem Range Tree

Erinnerung:

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

$v_{\text{split}} \leftarrow \text{FindSplitNode}(T, x, x')$
if v_{split} ist Blatt **then** prüfe v_{split}
else
  $v \leftarrow \text{lc}(v_{\text{split}})$
  while v kein Blatt **do**
    if $x \leq x_v$ **then**
      ~~ReportSubtree(rc(v))~~ 1dRangeQuery($T_y(\text{rc}(v)), y, y'$)
      $v \leftarrow \text{lc}(v)$
    else $v \leftarrow \text{rc}(v)$
  prüfe v
  `// analog für` x' `und rc(`v_{split}`)`

Bereichsabfrage in einem Range Tree

Erinnerung:

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

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} ist Blatt **then** prüfe v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v kein Blatt **do**
    if $x \leq x_v$ **then**
      ~~ReportSubtree(rc(v))~~ 1dRangeQuery(T_y(rc(v))$, y, y'$)
      $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  prüfe v
  `// analog für` x' `und rc(`v_{split}`)`

Lemma: Eine Bereichsabfrage in einem Range Tree benötigt $O(\log^2 n + k)$ Zeit, wobei k die Antwortgröße ist.

Beschleunigung durch Fractional Cascading

Beob.: Bereichsabfrage in Range Tree führt $O(\log n)$ 1d Abfragen in jeweils $O(\log n + k_v)$ Zeit durch. Das Abfrageintervall $[y, y']$ ist immer gleich!

Beschleunigung durch Fractional Cascading

Beob.: Bereichsabfrage in Range Tree führt $O(\log n)$ 1d Abfragen in jeweils $O(\log n + k_v)$ Zeit durch. Das Abfrageintervall $[y, y']$ ist immer gleich!

Idee: Nutze diese Eigenschaft aus um 1d-Abfragen auf $O(1 + k_v)$ Zeit zu beschleunigen

Beschleunigung durch Fractional Cascading

Beob.: Bereichsabfrage in Range Tree führt $O(\log n)$ 1d Abfragen in jeweils $O(\log n + k_v)$ Zeit durch. Das Abfrageintervall $[y, y']$ ist immer gleich!

Idee: Nutze diese Eigenschaft aus um 1d-Abfragen auf $O(1 + k_v)$ Zeit zu beschleunigen

Beispiel: zwei Mengen $B \subseteq A \subseteq \mathbb{R}$ in sortierten Arrays

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

B	10	19	30	62	70	80	100

Beschleunigung durch Fractional Cascading

Beob.: Bereichsabfrage in Range Tree führt $O(\log n)$ 1d Abfragen in jeweils $O(\log n + k_v)$ Zeit durch. Das Abfrageintervall $[y, y']$ ist immer gleich!

Idee: Nutze diese Eigenschaft aus um 1d-Abfragen auf $O(1 + k_v)$ Zeit zu beschleunigen

Beispiel: zwei Mengen $B \subseteq A \subseteq \mathbb{R}$ in sortierten Arrays

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

B	10	19	30	62	70	80	100

Suchintervall [20,65]

Beschleunigung durch Fractional Cascading

Beob.: Bereichsabfrage in Range Tree führt $O(\log n)$ 1d Abfragen in jeweils $O(\log n + k_v)$ Zeit durch. Das Abfrageintervall $[y, y']$ ist immer gleich!

Idee: Nutze diese Eigenschaft aus um 1d-Abfragen auf $O(1 + k_v)$ Zeit zu beschleunigen

Beispiel: zwei Mengen $B \subseteq A \subseteq \mathbb{R}$ in sortierten Arrays

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

Geht es besser als zwei binäre Suchen?

B	10	19	30	62	70	80	100

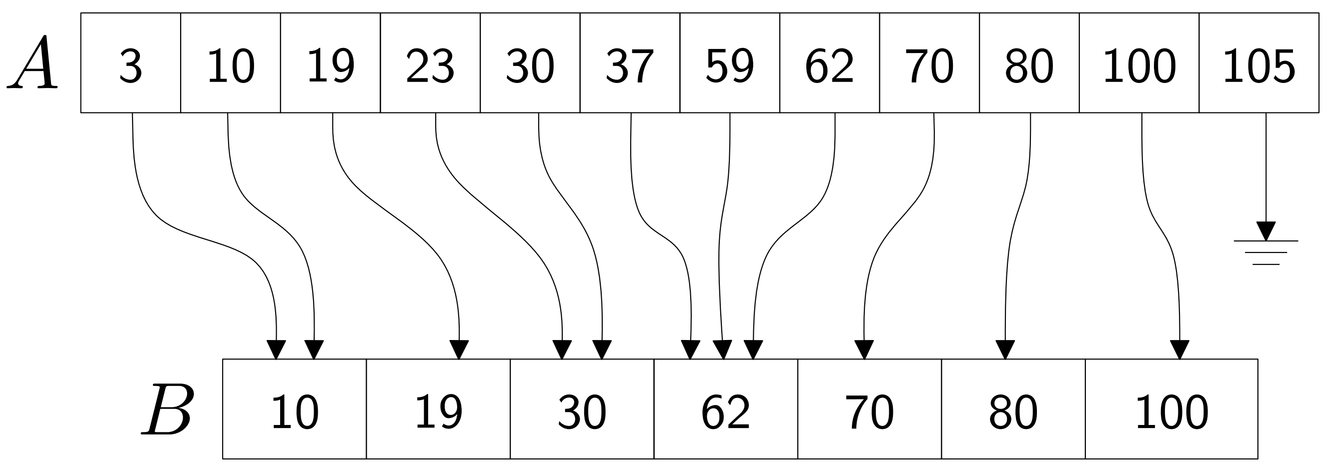
Suchintervall [20,65]

Beschleunigung durch Fractional Cascading

Beob.: Bereichsabfrage in Range Tree führt $O(\log n)$ 1d Abfragen in jeweils $O(\log n + k_v)$ Zeit durch. Das Abfrageintervall $[y, y']$ ist immer gleich!

Idee: Nutze diese Eigenschaft aus um 1d-Abfragen auf $O(1 + k_v)$ Zeit zu beschleunigen

Beispiel: zwei Mengen $B \subseteq A \subseteq \mathbb{R}$ in sortierten Arrays



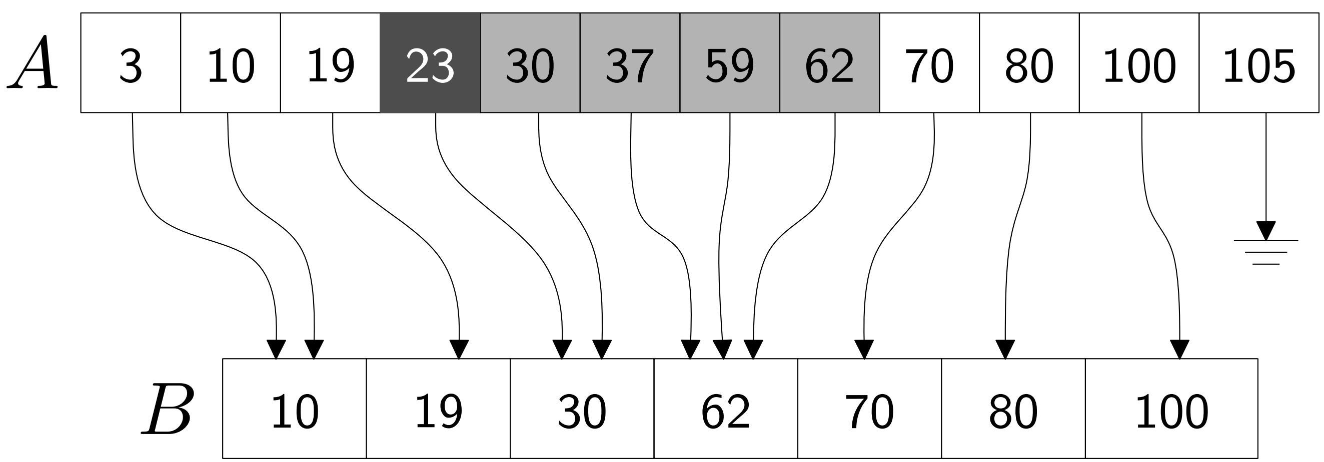
verknüpfe $a \in A$ mit kleinstem $b \geq a$ in B

Beschleunigung durch Fractional Cascading

Beob.: Bereichsabfrage in Range Tree führt $O(\log n)$ 1d Abfragen in jeweils $O(\log n + k_v)$ Zeit durch. Das Abfrageintervall $[y, y']$ ist immer gleich!

Idee: Nutze diese Eigenschaft aus um 1d-Abfragen auf $O(1 + k_v)$ Zeit zu beschleunigen

Beispiel: zwei Mengen $B \subseteq A \subseteq \mathbb{R}$ in sortierten Arrays



verknüpfe $a \in A$ mit kleinstem $b \geq a$ in B

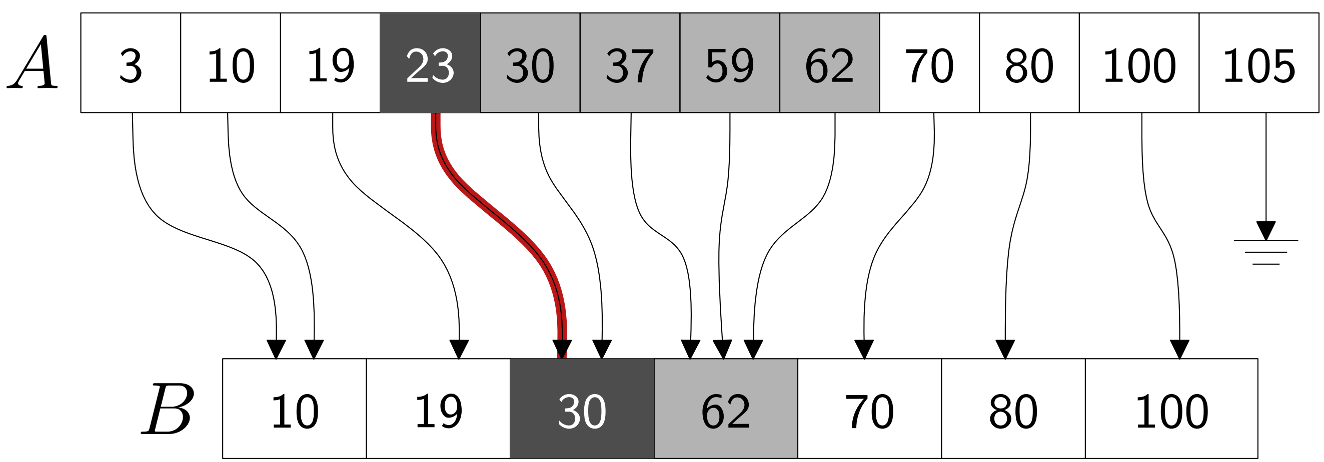
Suchintervall [20,65]

Beschleunigung durch Fractional Cascading

Beob.: Bereichsabfrage in Range Tree führt $O(\log n)$ 1d Abfragen in jeweils $O(\log n + k_v)$ Zeit durch. Das Abfrageintervall $[y, y']$ ist immer gleich!

Idee: Nutze diese Eigenschaft aus um 1d-Abfragen auf $O(1 + k_v)$ Zeit zu beschleunigen

Beispiel: zwei Mengen $B \subseteq A \subseteq \mathbb{R}$ in sortierten Arrays



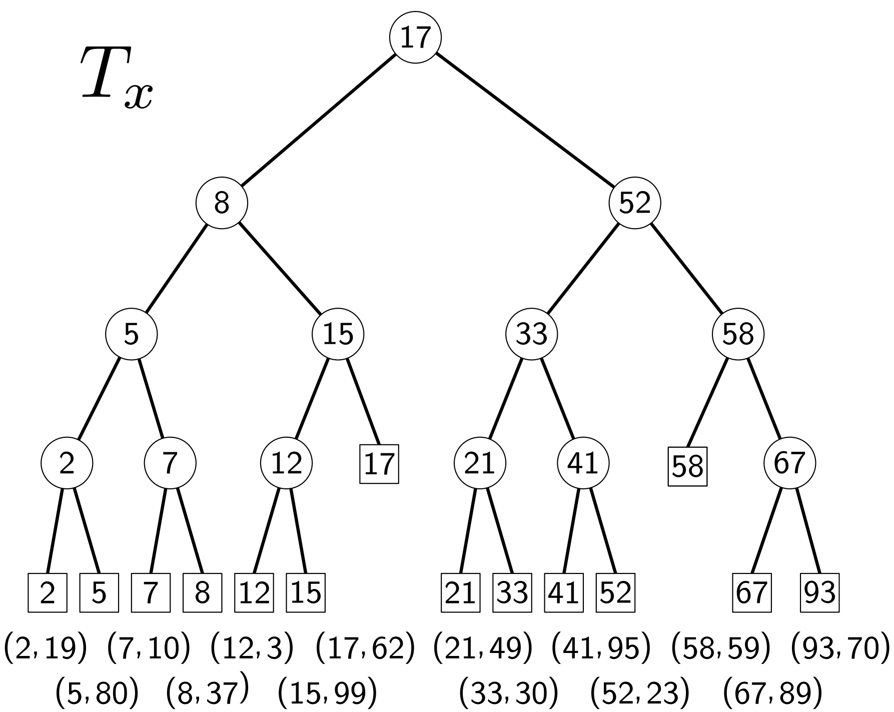
verknüpfe $a \in A$ mit kleinstem $b \geq a$ in B

Suchintervall [20,65]

Pointer liefert Startpunkt für zweite Suche in $O(1)$ Zeit

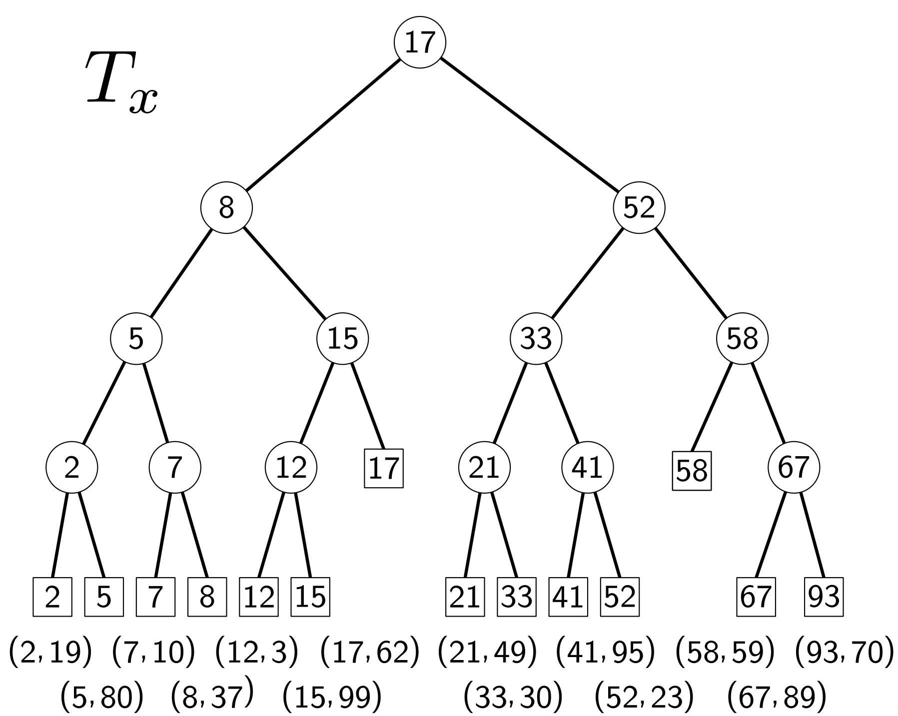
Beschleunigung durch Fractional Cascading

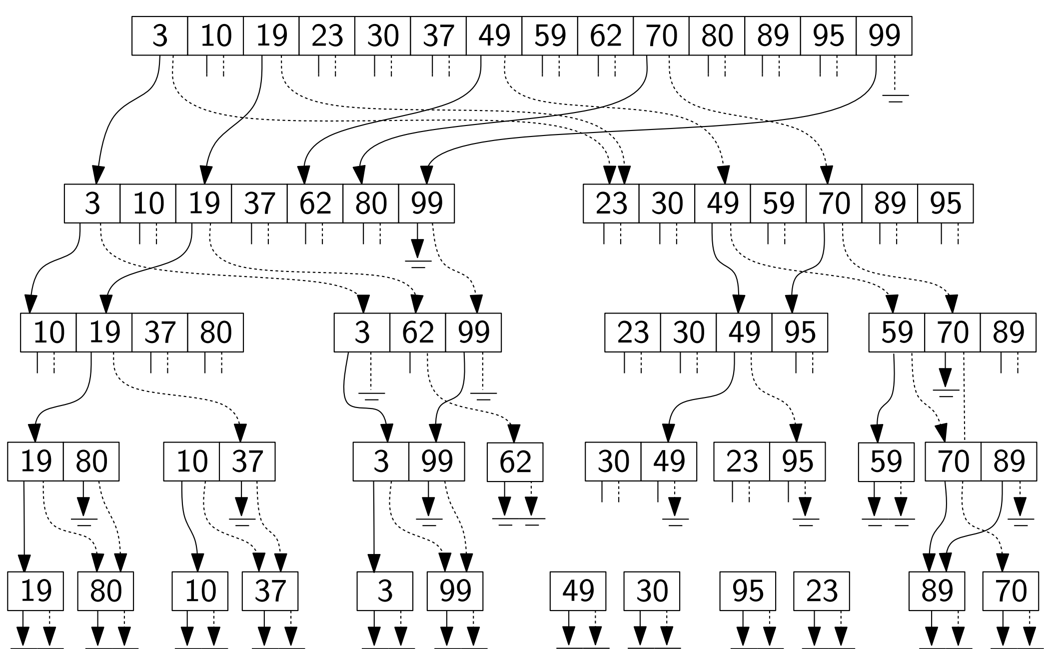
- In Range Trees gilt für die kanonischen Blattmengen $P(\mathsf{lc}(v)) \subseteq P(v)$ und $P(\mathsf{rc}(v)) \subseteq P(v)$.



Beschleunigung durch Fractional Cascading

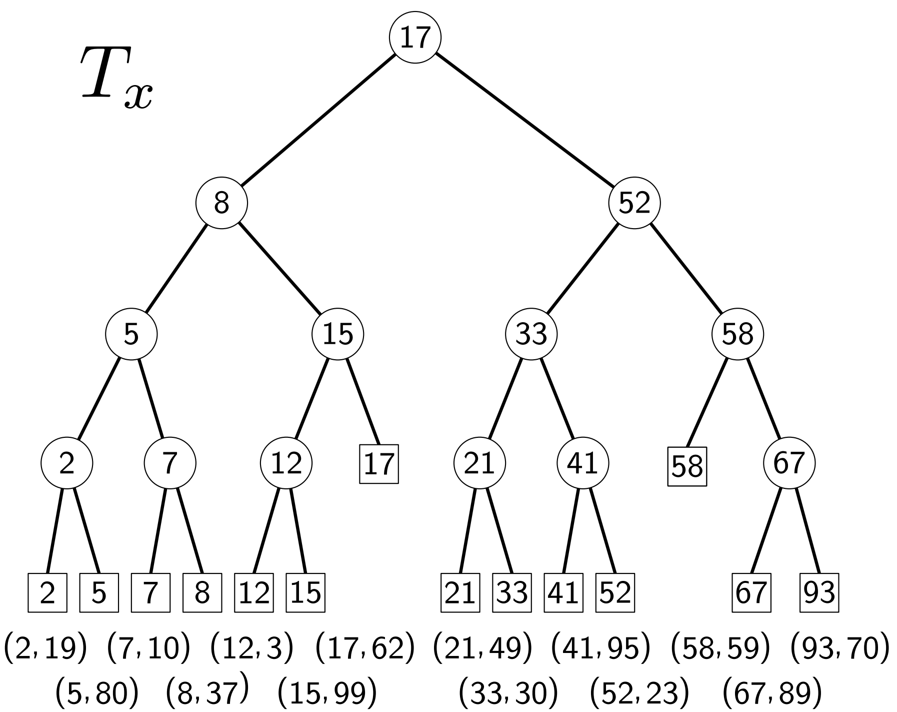
- In Range Trees gilt für die kanonischen Blattmengen $P(\mathrm{lc}(v)) \subseteq P(v)$ und $P(\mathrm{rc}(v)) \subseteq P(v)$.
- definiere für jedes Array-Element $A(v)[i]$ zwei entsprechende Pointer in die Arrays $A(\mathrm{lc}(v))$ und $A(\mathrm{rc}(v))$
  $\rightarrow$ **Layered Range Tree**

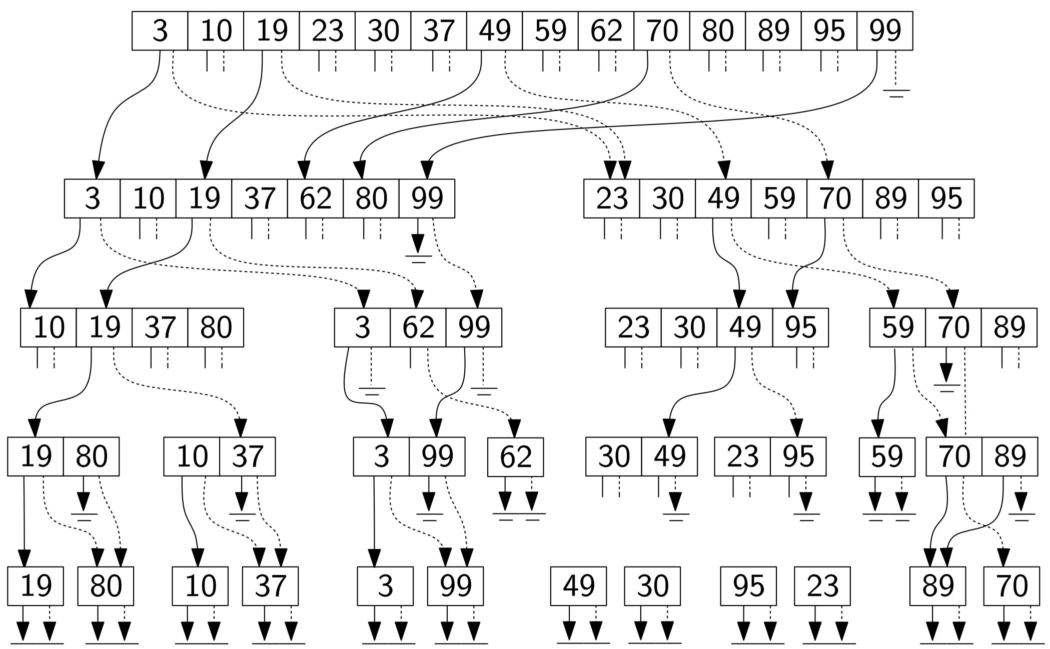




Beschleunigung durch Fractional Cascading

- In Range Trees gilt für die kanonischen Blattmengen $P(\mathrm{lc}(v)) \subseteq P(v)$ und $P(\mathrm{rc}(v)) \subseteq P(v)$.
- definiere für jedes Array-Element $A(v)[i]$ zwei entsprechende Pointer in die Arrays $A(\mathrm{lc}(v))$ und $A(\mathrm{rc}(v))$
  $\rightarrow$ **Layered Range Tree**
- im Splitknoten eine binäre Suche in $O(\log n)$ Zeit, danach in Kindern in $O(1)$ Zeit die Pointer mitverfolgen





Beschleunigung durch Fractional Cascading

- In Range Trees gilt für die kanonischen Blattmengen $P(\mathsf{lc}(v)) \subseteq P(v)$ und $P(\mathsf{rc}(v)) \subseteq P(v)$.
- definiere für jedes Array-Element $A(v)[i]$ zwei entsprechende Pointer in die Arrays $A(\mathsf{lc}(v))$ und $A(\mathsf{rc}(v))$
  $\rightarrow$ **Layered Range Tree**
- im Splitknoten eine binäre Suche in $O(\log n)$ Zeit, danach in Kindern in $O(1)$ Zeit die Pointer mitverfolgen

Satz: Ein Layered Range Tree für n Punkte im $\mathbb{R}^2$ lässt sich in $O(n \log n)$ Zeit und Platz konstruieren. Bereichsabfragen benötigen $O(\log n + k)$ Zeit, wobei k die Antwortgröße ist.

Beliebige Punktmengen

Bisher: Punkte in allgemeiner Lage, d.h. keine zwei Punkte mit gleicher x- oder y-Koordinate

Idee: benutze statt $\mathbb{R}$ Zahlenpaare $(a|b)$ mit lexikographischer Ordnung

Beliebige Punktmengen

Bisher: Punkte in allgemeiner Lage, d.h. keine zwei Punkte mit gleicher x- oder y-Koordinate

Idee: benutze statt $\mathbb{R}$ Zahlenpaare $(a|b)$ mit lexikographischer Ordnung

$$p = (p_x, p_y)$$

Beliebige Punktmengen

Bisher: Punkte in allgemeiner Lage, d.h. keine zwei Punkte mit gleicher x- oder y-Koordinate

Idee: benutze statt $\mathbb{R}$ Zahlenpaare $(a|b)$ mit lexikographischer Ordnung

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

Beliebige Punktmengen

Bisher: Punkte in allgemeiner Lage, d.h. keine zwei Punkte mit gleicher x- oder y-Koordinate

Idee: benutze statt $\mathbb{R}$ Zahlenpaare $(a|b)$ mit lexikographischer Ordnung

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

eindeutige Koord.

Beliebige Punktmengen

Bisher: Punkte in allgemeiner Lage, d.h. keine zwei Punkte mit gleicher x- oder y-Koordinate

Idee: benutze statt $\mathbb{R}$ Zahlenpaare $(a|b)$ mit lexikographischer Ordnung

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

eindeutige Koord.

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

Beliebige Punktmengen

Bisher: Punkte in allgemeiner Lage, d.h. keine zwei Punkte mit gleicher x- oder y-Koordinate

Idee: benutze statt $\mathbb{R}$ Zahlenpaare $(a|b)$ mit lexikographischer Ordnung

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

eindeutige Koord.

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

Beliebige Punktmengen

Bisher: Punkte in allgemeiner Lage, d.h. keine zwei Punkte mit gleicher x- oder y-Koordinate

Idee: benutze statt $\mathbb{R}$ Zahlenpaare $(a|b)$ mit lexikographischer Ordnung

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

eindeutige Koord.

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

$$\downarrow$$

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

Beliebige Punktmengen

Bisher: Punkte in allgemeiner Lage, d.h. keine zwei Punkte mit gleicher x- oder y-Koordinate

Idee: benutze statt $\mathbb{R}$ Zahlenpaare $(a|b)$ mit lexikographischer Ordnung

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

eindeutige Koord.

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

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

Zeige:

Beliebige Punktmengen

Bisher: Punkte in allgemeiner Lage, d.h. keine zwei Punkte mit gleicher x- oder y-Koordinate

Idee: benutze statt $\mathbb{R}$ Zahlenpaare $(a|b)$ mit lexikographischer Ordnung

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

eindeutige Koord.

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

$$\downarrow$$

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

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

Zusammenfassung

Geg: Menge P von n Punkten in $\mathbb{R}^2$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $R = [x, x'] \times [y, y']$

$\rightarrow$ haben zwei Alternativen gesehen

	kd-Tree	Range-Tree
Aufbau		
Speicherplatz		
Anfragezeit		

Zusammenfassung

Geg: Menge P von n Punkten in $\mathbb{R}^2$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $R = [x, x'] \times [y, y']$

$\rightarrow$ haben zwei Alternativen gesehen

	kd-Tree	Range-Tree
Aufbau	$O(n \log n)$	$O(n \log n)$
Speicherplatz	$O(n)$	$O(n \log n)$
Anfragezeit	$O(\sqrt{n} + k)$	$O(\log^2 n + k)$

Zusammenfassung

Geg: Menge P von n Punkten in $\mathbb{R}^2$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $R = [x, x'] \times [y, y']$

$\rightarrow$ haben zwei Alternativen gesehen

	kd-Tree	Range-Tree
Aufbau	$O(n \log n)$	$O(n \log n)$
Speicherplatz	$O(n)$	$O(n \log n)$
Anfragezeit	$O(\sqrt{n} + k)$	$O(\log^{\cancel{2}} n + k)$

Zusammenfassung

Geg: Menge P von n Punkten in $\mathbb{R}^2$

Ziel: Datenstruktur zur effizienten Beantwortung von Bereichsabfragen der Form $R = [x, x'] \times [y, y']$

$\rightarrow$ haben zwei Alternativen gesehen

	kd-Tree	Range-Tree
Aufbau	$O(n \log n)$	$O(n \log n)$
Speicherplatz	$O(n)$	$O(n \log n)$
Anfragezeit	$O(\sqrt{n} + k)$	$O(\log^{\cancel{2}} n + k)$

Diskussion

Wie lassen sich die Datenstrukturen auf den d-dimensionalen Fall verallgemeinern?

Diskussion

Wie lassen sich die Datenstrukturen auf den d-dimensionalen Fall verallgemeinern?

- kd-Trees funktionieren ganz analog und trennen die Punkte alternierend in den d Koordinaten. Speicherplatz bleibt $O(n)$, Konstruktion $O(n \log n)$ und Abfragezeit ist $O(n^{1-1/d} + k)$.

Diskussion

Karlsruhe Institute of Technology

Wie lassen sich die Datenstrukturen auf den d-dimensionalen Fall verallgemeinern?

- kd-Trees funktionieren ganz analog und trennen die Punkte alternierend in den d Koordinaten. Speicherplatz bleibt $O(n)$, Konstruktion $O(n \log n)$ und Abfragezeit ist $O(n^{1-1/d} + k)$.
- Range Trees lassen sich ebenfalls rekursiv aufbauen: Die Hilfsstruktur im Suchbaum der ersten Koordinate ist ein $(d-1)$-dimensionaler Range Tree. Damit wachsen Speicherbedarf und Konstruktionszeit auf $O(n \log^{d-1} n)$; eine Abfrage benötigt $O(\log^d n + k)$ Zeit, mit Fractional Cascading $O(\log^{d-1} n + k)$.

Diskussion

Wie lassen sich die Datenstrukturen auf den d-dimensionalen Fall verallgemeinern?

- kd-Trees funktionieren ganz analog und trennen die Punkte alternierend in den d Koordinaten. Speicherplatz bleibt $O(n)$, Konstruktion $O(n \log n)$ und Abfragezeit ist $O(n^{1-1/d} + k)$.
- Range Trees lassen sich ebenfalls rekursiv aufbauen: Die Hilfsstruktur im Suchbaum der ersten Koordinate ist ein $(d-1)$-dimensionaler Range Tree. Damit wachsen Speicherbedarf und Konstruktionszeit auf $O(n \log^{d-1} n)$; eine Abfrage benötigt $O(\log^d n + k)$ Zeit, mit Fractional Cascading $O(\log^{d-1} n + k)$.

Lassen sich auch Abfragen für andere Objekte (z.B. Polygone) mit den Datenstrukturen beantworten?

Diskussion

Wie lassen sich die Datenstrukturen auf den d-dimensionalen Fall verallgemeinern?

- kd-Trees funktionieren ganz analog und trennen die Punkte alternierend in den d Koordinaten. Speicherplatz bleibt $O(n)$, Konstruktion $O(n \log n)$ und Abfragezeit ist $O(n^{1-1/d} + k)$.
- Range Trees lassen sich ebenfalls rekursiv aufbauen: Die Hilfsstruktur im Suchbaum der ersten Koordinate ist ein $(d-1)$-dimensionaler Range Tree. Damit wachsen Speicherbedarf und Konstruktionszeit auf $O(n \log^{d-1} n)$; eine Abfrage benötigt $O(\log^d n + k)$ Zeit, mit Fractional Cascading $O(\log^{d-1} n + k)$.

Lassen sich auch Abfragen für andere Objekte (z.B. Polygone) mit den Datenstrukturen beantworten?

Ja, das geht durch eine geeignete Transformation von Polygonen in Punkte in einem 4d-Raum (s. Übung) bzw. mit Windowing Queries (kommt evtl. in späterer Vorlesung).