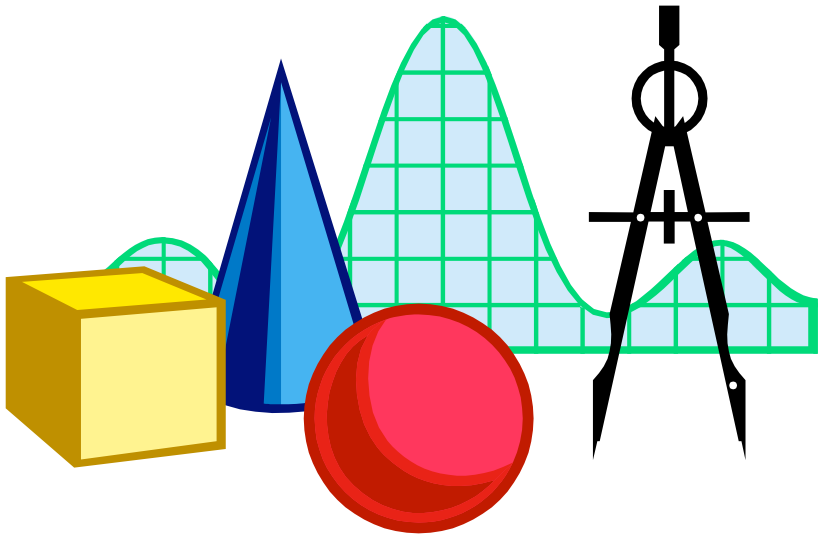
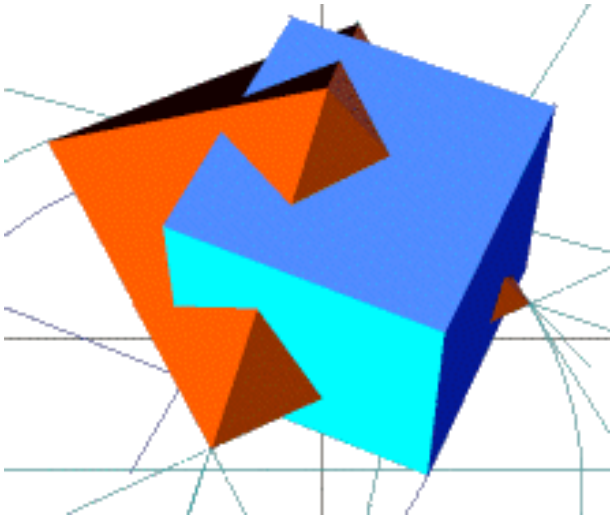
Algoritmisk geometri



Intervalsøgning



Motivation for intervaltræer

Lad der være givet en database over ansatte i en virksomhed

Ansat
• Alder • Løn • Ansættelsesdato

post i databasen

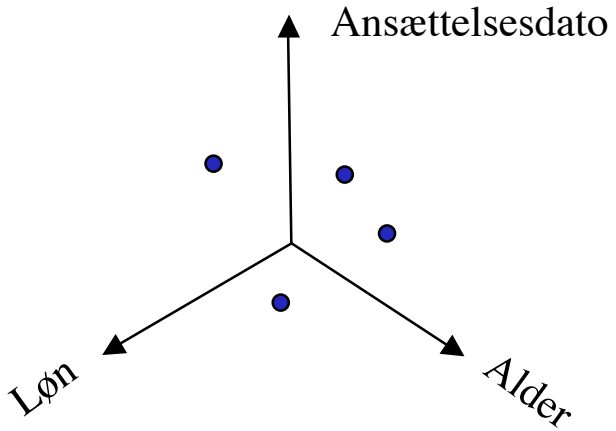
Antag, at vi ønsker at finde alle ansatte med en løn i intervallet [20000, 50000]
Vi kan undersøge samtlige poster og udtage dem, hvor lønnen ligger i dette interval
Men hvis databasen er stor, er denne fremgangsmåde for langsom
Det samme gør sig gældende, hvis vi ønsker at finde alle ansatte, for hvilke

$$\text{Løn} \in [20000, 50000] \wedge \text{Alder} \in [25, 40]$$

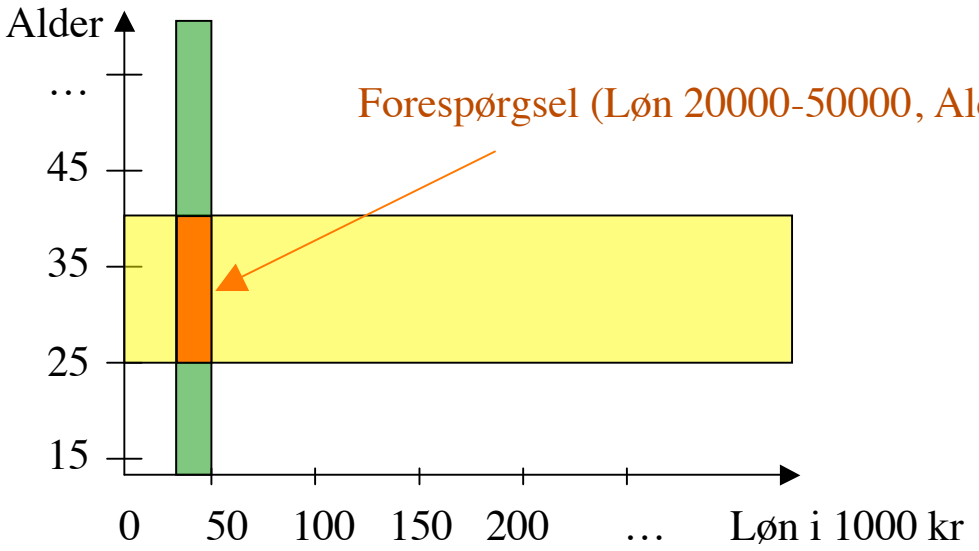
Intervalsøgning

Vi kan opfatte en ansat som et punkt i et 3-dimensionalt rum

Alder:	[15, 67]
Løn:	[0, 500000]
Ansættelsesdato:	[1/1 1900, 1/1 2100]



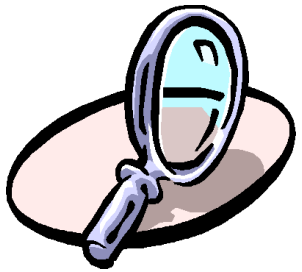
Ortogonal intervalsøgning



Vi kan opnå hurtigere søgning end lineært gennemløb ved hjælp af passende datastrukturer

Vi vil tilstræbe $O(\log n + s)$ køretid, hvor s er antallet af fundne elementer

1-dimensional intervalsøgning



Data: Punkter $P = \{p_1, p_2, .., p_n\}$ i 1 dimension (mængde af reelle tal)

Forespørgsel: Find alle punkter i et 1-dimensionalt interval, $[x_1, x_2]$

Datastruktur 1: Sorteret array

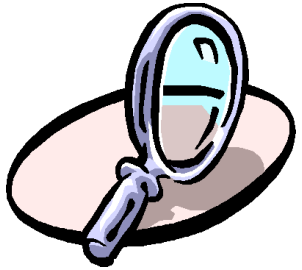
Algoritme:

1. Søg efter x_1 og x_2 ved binær søgning. Køretid $O(\log n)$
2. Udskriv alle punkter imellem dem. Køretid $O(s)$

Samlet køretid $O(\log n + s)$

1-dimensional intervalsøgning

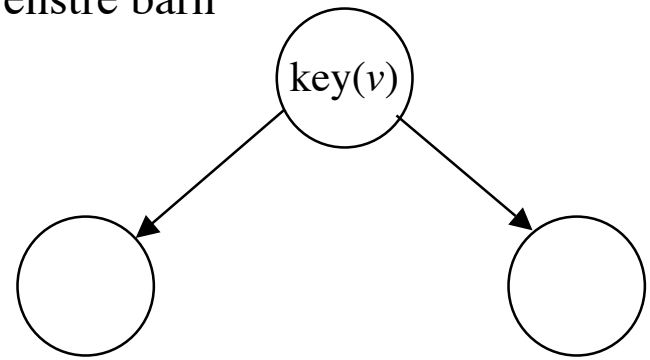
(fortsat)



Datastruktur 2: Balanceret binært søgetræ (f.eks. et rød-sort-træ)
De interne knuder indeholder nøglerne og bruges til at styre søgningen

Algoritme:
Benyt rekursion startende i roden
Når en intern knude v mødes gøres følgende:

(1) hvis key(v) < x_1, så søg rekursivt i v's højre barn
(2) hvis $x_1 \leq$ key(v) $\leq x_2$, så søg rekursivt i begge v's børn
(3) hvis key(v) > x_2, så søg rekursivt i v's venstre barn

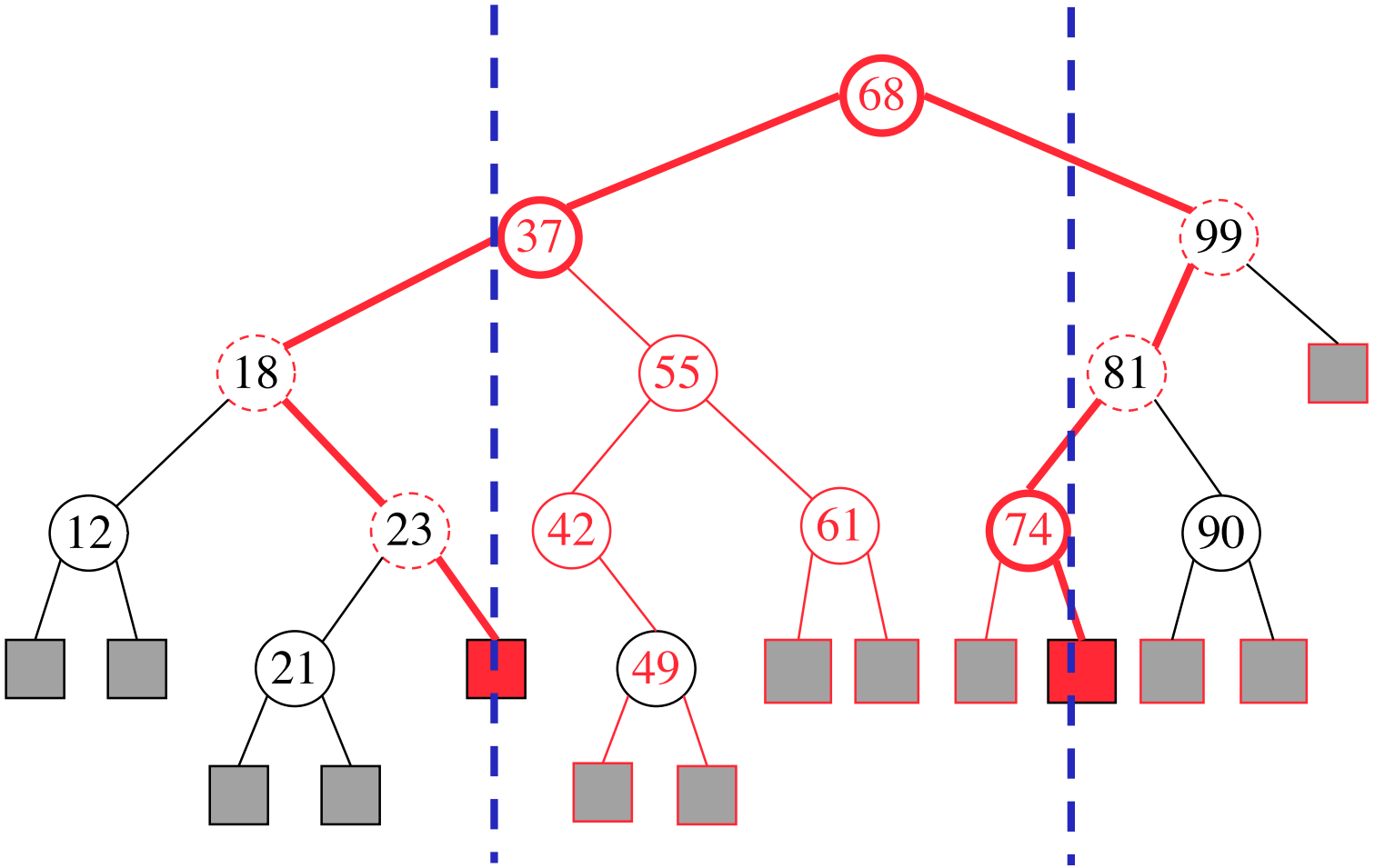


Algoritme til 1-dimensional intervalsøgning

Algorithm *1DTreeRangeSearch*(x_1, x_2, v)
  Input Search keys x_1 and x_2, and a node v of a binary search tree T
  Output The elements stored in the subtree of T rooted at v, whose keys are greater than or equal to x_1 and less than or equal to x_2
  if *T.isExternal*(v) **then**
    return $\emptyset$
  if $x_1 \leq key(v) \leq x_2$ **then**
    $L \leftarrow$ *1DTreeRangeSearch*(x_1, x_2, *T.leftChild*(v))
    $R \leftarrow$ *1DTreeRangeSearch*(x_1, x_2, *T.rightChild*(v))
    return $L \cup \{element(v)\} \cup R$
  else if $key(v) < x_1$ **then**
    return *1DTreeRangeSearch*(x_1, x_2, *T.rightChild*(v))
  else if $x_2 < key(v)$ **then**
    return *1DTreeRangeSearch*(x_1, x_2, *T.leftChild*(v))

1-dimensional intervalsøgning

(Eksempel: søgning for $x_1 = 30$ og $x_2 = 80$)



Analyse

Balanceret binært søgetræ:

Pladsforbrug: $O(n)$

Tid for konstruktion: $O(n \log n)$

Tid for forespørgsel:

Vi besøger s knuder inden for intervallet

Vi besøger højst $2h$ knuder, som ikke er i intervallet (de stiplede knuder på forrige figur), hvor h er træets højde

Da træet er balanceret, h er $O(\log n)$, er køretiden for søgning $O(\log n + s)$

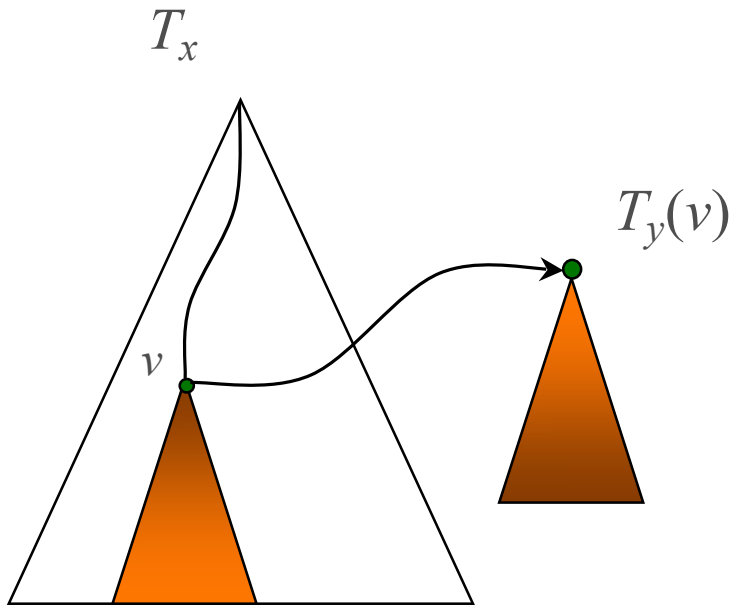
2-dimensional intervalsøgning

Lad T_x være et 1-dimensionalt intervaltræ baseret på punkternes x-koordinater

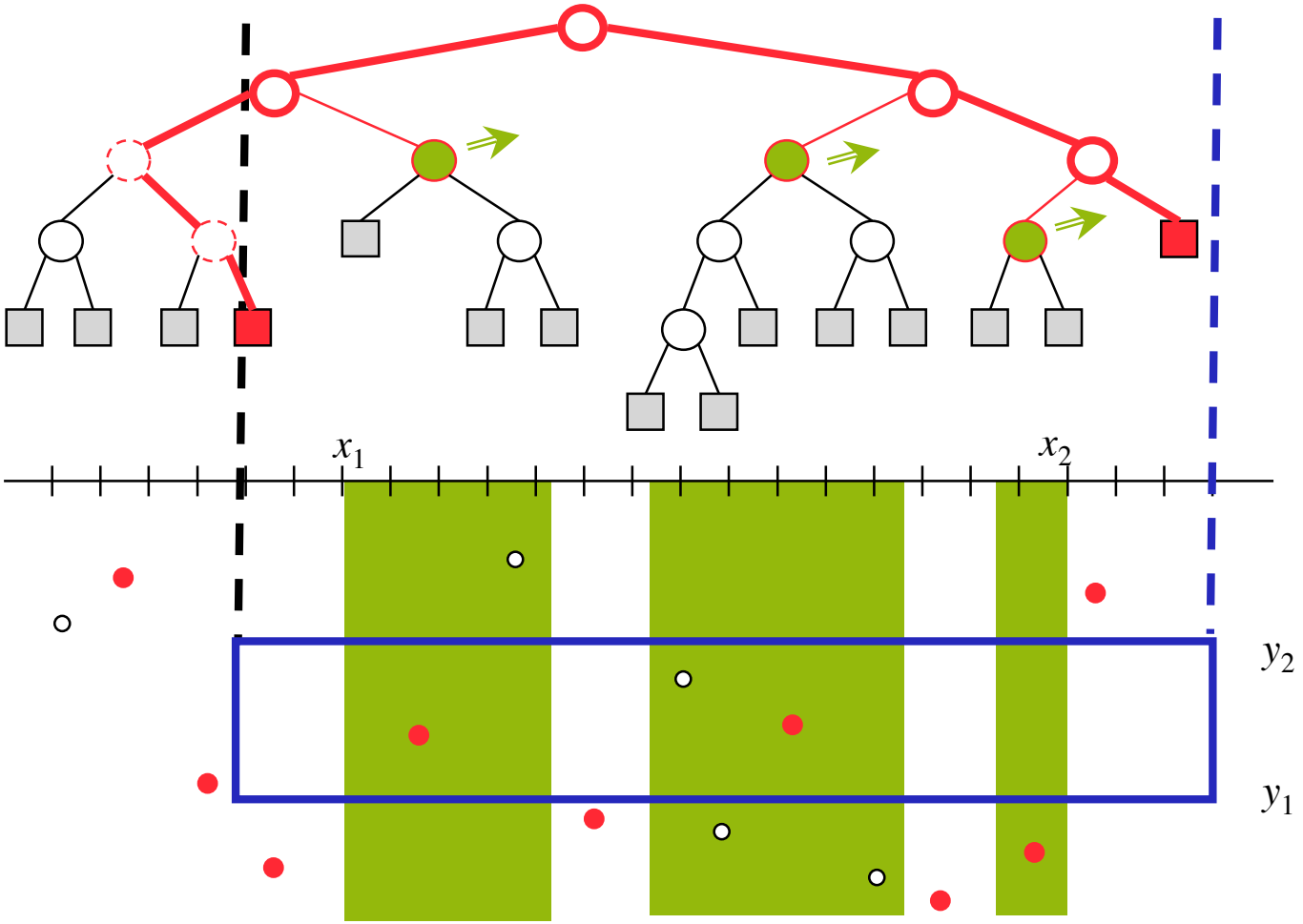
Knyt til hver interne knude v i T_x et 1-dimensionalt træ $T_y(v)$, der indeholder knuderne i undertræet for v og er ordnet efter disses y-koordinat

Søg i T_x efter punkter i intervallet $[x_1, x_2]$
Undervejs opsamles de knuder, der tilhører $[x_1, x_2]$ x $[y_1, y_2]$

Når algoritmen møder en knude, v, hvor alle knuder i dens undertræ tilhører $[x_1, x_2]$, søges i det tilknyttede hjælpetræ, $T_y(v)$
Efterkommere til v i T_x undersøges ikke



2-dimensional intervalsøgning



Algoritme til 2-dimensional intervalsøgning

Algorithm *2DTreeRangeSearch*(x_1, x_2, y_1, y_2, v, t)

Input Search keys x_1, x_2, y_1, and y_2; node v in the primary structure T of a two-dimensional range tree; type t of node v

Output The items in the subtree rooted at v, whose coordinates are in the x-range $[x_1,x_2]$ and in the y-range $[y_1,y_2]$

```
if T.isExternal(v) then
    return Ø
if x1 ≤ x(v) ≤ x2 then
    if y1 ≤ y(v) ≤ y2 then
        M ← {element(v)}
    else
        M ← Ø
    { to be continued on next page }
```

```
    if t = "left" then
        L ← 2DRangeSearch(x1, x2, y1, y2, T.leftChild(v), "left")
        R ← 1DRangeSearch(y1, y2, T.rightChild(v))
    else if t = "right" then
        L ← 1DRangeSearch(y1, y2, T.leftChild(v))
        R ← 2DRangeSearch(x1, x2, y1, y2, T.rightChild(v), "right")
    else if t = "middle" then
        L ← 2DRangeSearch(x1, x2, y1, y2, T.leftChild(v), "left")
        R ← 2DRangeSearch(x1, x2, y1, y2, T.rightChild(v), "right")
else { x(v) < x1 ∨ x(v) > x2 }
    M ← Ø
    if x(v) < x1 then
        L ← Ø
        R ← 2DRangeSearch(x1, x2, y1, y2, T.rightChild(v), t)
    else { x(v) > x2 }
        L ← 2DRangeSearch(x1, x2, y1, y2, T.leftChild(v), t)
        R ← Ø
return L ∪ M ∪ R
```

Analyse
(plads)

Et 2D-intervaltræ bruger $O(n \log n)$ plads

Bevis:
Et punkt, der lagres i T_x, lagres også i enhver hjælpestruktur $T_y(u)$, hvor u er en forfader til v i T_x

Der er $O(\log n)$ forfædre for hver knude, og derfor $O(\log n)$ kopier af ethvert punkt

Analyse
(tid)

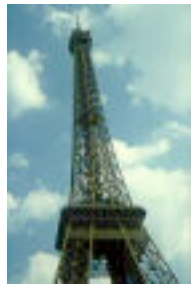
Søgning i et 2D-intervaltræ bruger $O(\log^2 n + s)$ tid, hvor s er antallet af fundne punkter

Bevis:
Tidsforbruget for knuder på de to søgeveje er $O(\log n)$

For hver af de indvendige knuder, der er børn af knuder på de to søgeveje, besøges deres hjælpetræ

Der er $O(\log n)$ af denne type knuder, og hver søgning i et hjælpetræ bruger $O(\log n)$ tid

Prioritetsøgetræer



Et **prioritetsøgetræ** er et træ, der er et binært søgetræ med hensyn til punkternes *x*-koordinat og en max-hob med hensyn punkternes *y*-koordinat

I et sådan træ indeholder roden (1) det punkt, der har den største *y*-koordinat samt (2) medianen for punkternes *x*-koordinater (bemærk, at denne værdi ikke behøver at være *x*-koordinaten for punktet)

Det venstre og højre undertræ er begge prioritetstræer og repræsenterer punkter med *x*-koordinat mindre end, henholdsvis større end, medianen

Prioritetsøgetræer
Konstruktion

Et priorioritetsøgetræ kan konstrueres i $O(n \log n)$ tid

Bevis:
Punkterne sorteres først i forhold til deres x-koordinat. Det kan gøres i $O(n \log n)$ tid
Medianen for punkternes x-kordinater kan bestemmes i $O(1)$ tid
Punktet med størst y-koordinat kan bestemmes i $O(n)$ tid
Dermed er roden bestemt, og dennes venstre og højre undertræ kan bestemmes på samme måde (rekursivt, ud fra to punktmængder, der hver er cirka halvt så stor som den oprindelige punktmængde)
Køretiden for konstruktion $T(n)$ kan derfor udtrykkes ved rekursionsligningen

$$T(n) = 2T(n/2) + bn,$$
$$T(1) = b$$

Ved brug af mestersætningen får vi, at

$T(n)$ er $O(n \log n)$

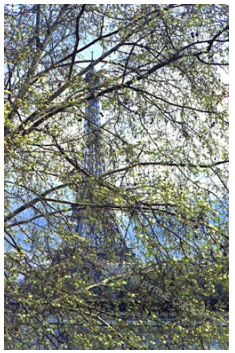
Prioritetsøgetræer
Søgning

Et prioritetsøgetræ kan benyttes til at finde alle punkter i et område af formen $[x_1, x_2]$ x $[y_1,\infty)$

Søgningen foretages som en normal 1-dimensional intervalsøgning efter $[x_1, x_2]$, men et undertræ for en knude v undersøges kun, hvis $y(v) \geq y_1$

Søgningen bruger $O(\log n + s)$ tid, hvor s er antallet af fundne punkter

Prioritetintervaltræ



Hvad gør vi, hvis ønsker at søge i et område af formen $[x_1, x_2]$ x $[y_1, y_2]$?

Benyt et **prioritetintervaltræ**, d.v.s. et binært søgetræ for y-koordinaterne, hvor der til alle interne knuder, undtagen roden, er knyttet et prioritettræ

Et venstre barn har et prioritetsøgetræ, der ikke er opad begrænset
Et højre barn har et et prioritetsøgetræ, der ikke er nedad begrænset

Forælderen tjener som den manglende grænse ved søgningen,
så den 3-sidede søgning er tilstrækkelig

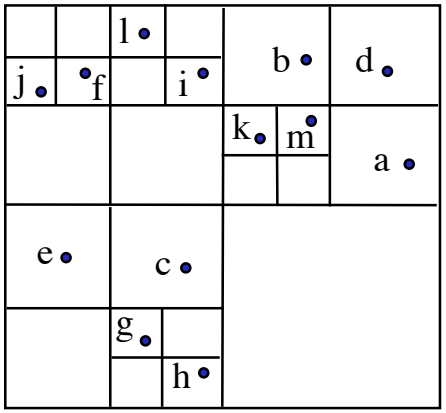
Søgning bruger $O(\log n + s)$ tid, hvor s er antallet af fundne punkter

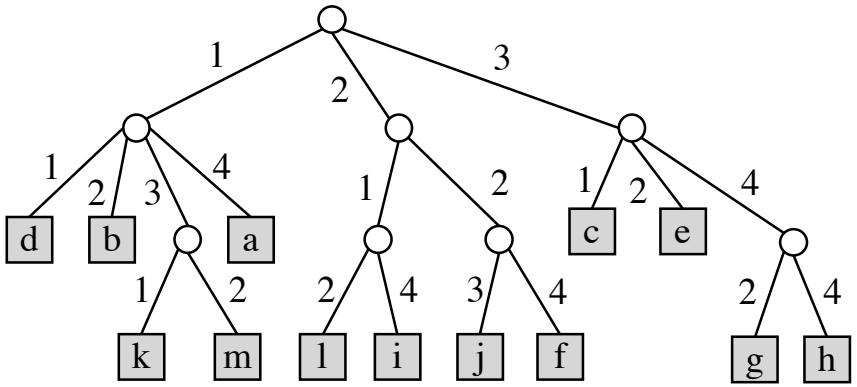
Quad-træer

Et quad-træ kan bruges til at gemme punkter i planet, således at intervalsøgning bliver let

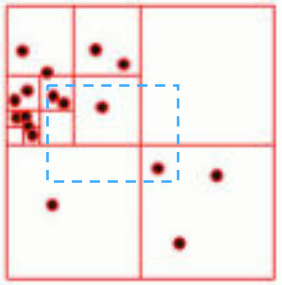
Der benyttes et 4-vejs træ, der repræsenterer kvadranter, delkvadranter, o.s.v.

Da træets dybde afhænger af punkternes nærhed, kan det være *meget* dybt. Sædvanligvis angives derfor en maksimal dybde, *D*, for træet





Søgning i quad-træer



Ved søgning efter punkter i et område R undersøges hvert delkvadrant, der overlapper R

Når R omslutter et delkvadrant, opregner vi blot alle de eksterne knuder, der er efterkommere af knuden

Når quad-træet har begrænset dybde, D, er køretiden for søgning $O(Dn)$. Hele træet traverseres i værste tilfælde

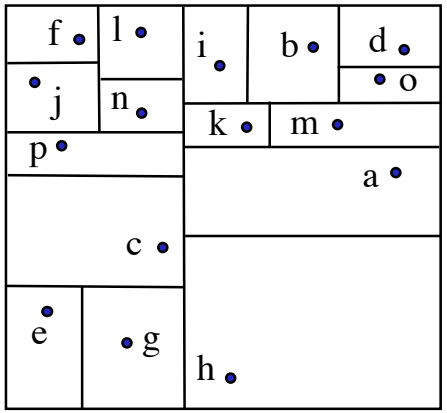
k-d-træer

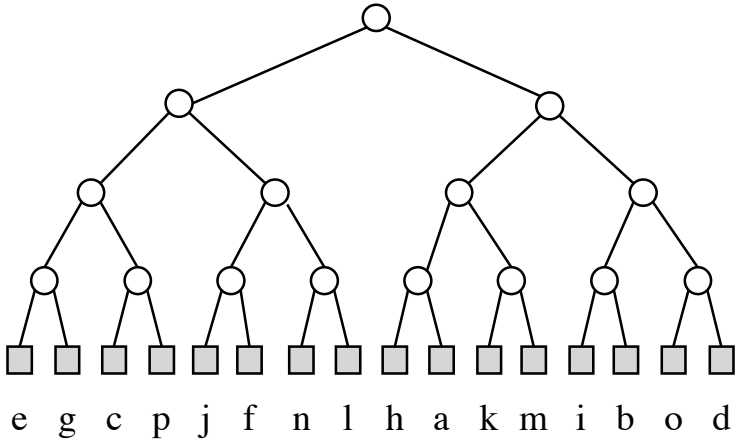
Et *k*-d-træ er en generalisering af et 1D-områdetræ til højere dimensioner

Et *k*-d-træ er et binært træ, hvor hver interne knuder repræsenterer en opsplitning af punkterne i en given dimension, og de eksterne knuder indeholder punkterne

To typer af *k*-d-træer:

- **Interval-baserede** (opsplitning i forhold til den længste side i rektanglet)
- **Punkt-baserede** (opsplitning i forhold til fordeling af punkter i rektanglet)

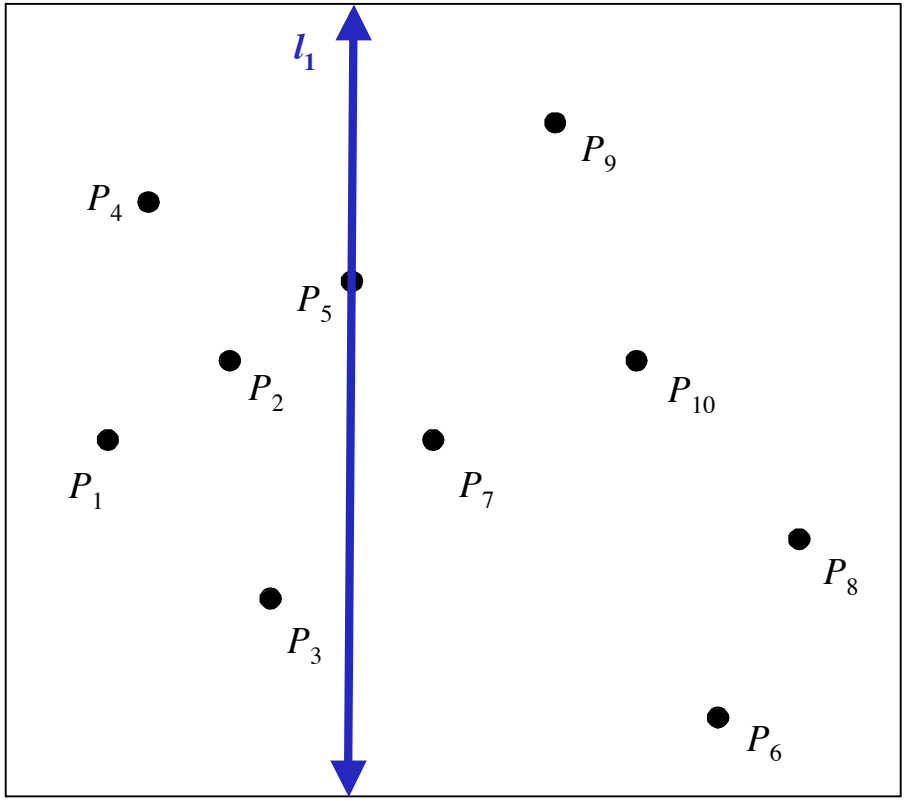




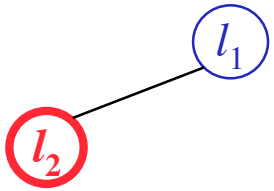
Konstruktion af *k*-d-Tree

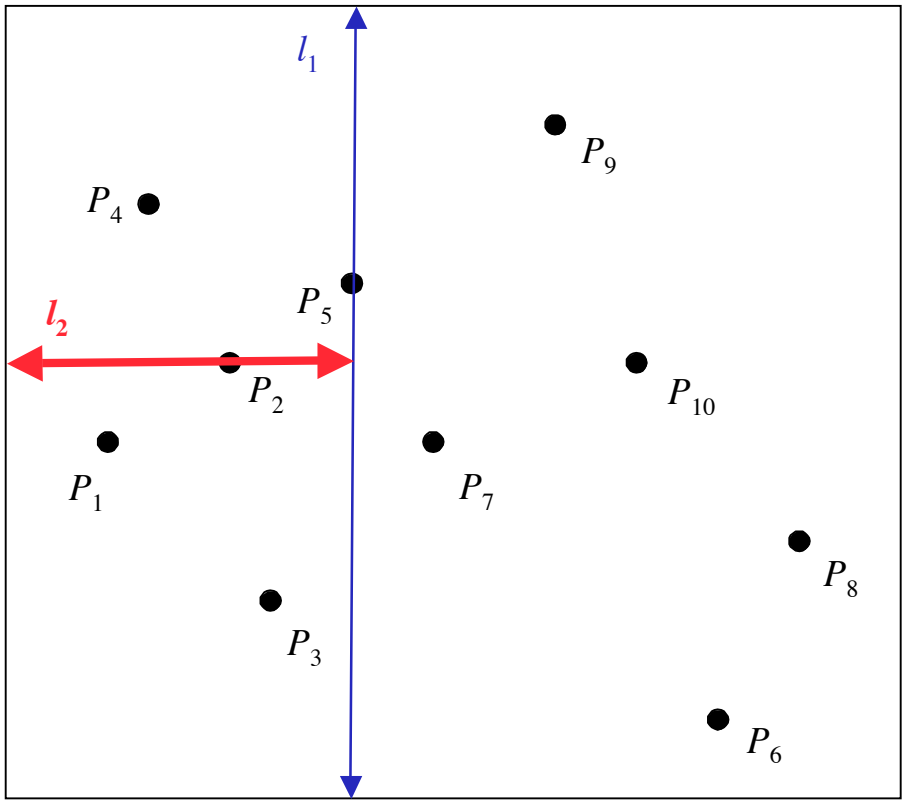
(punktbaseret)

l_1

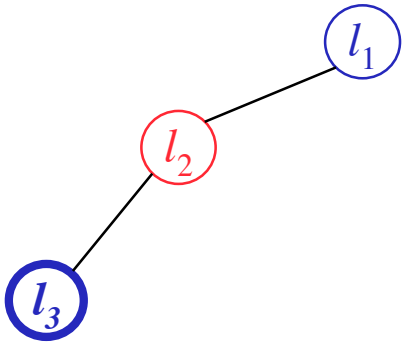


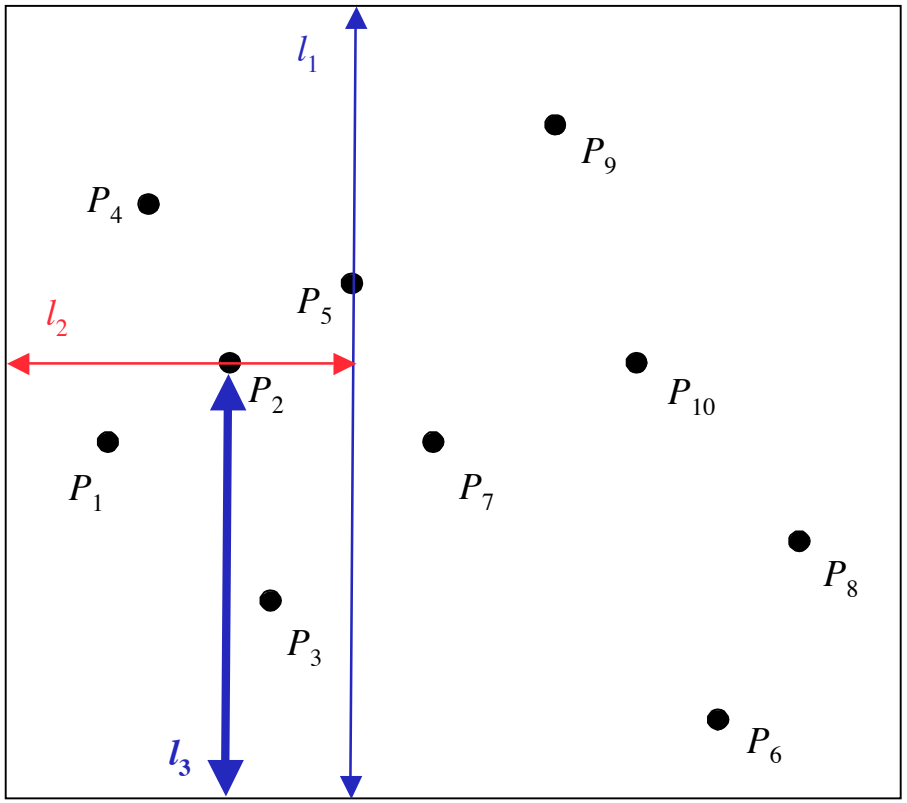
Konstruktion af *k*-d-Tree



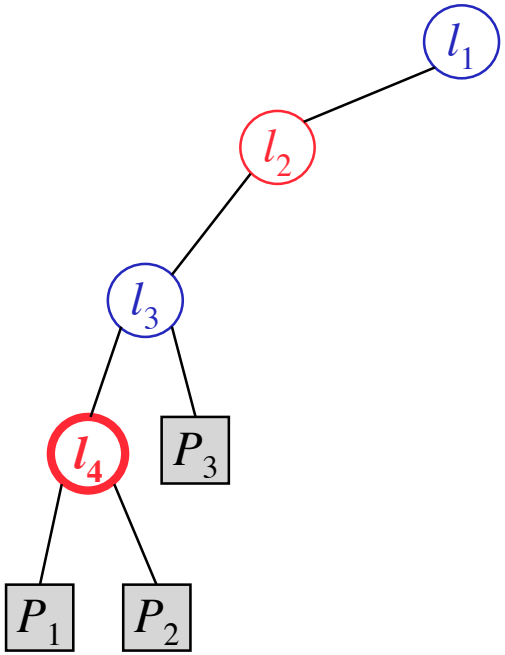


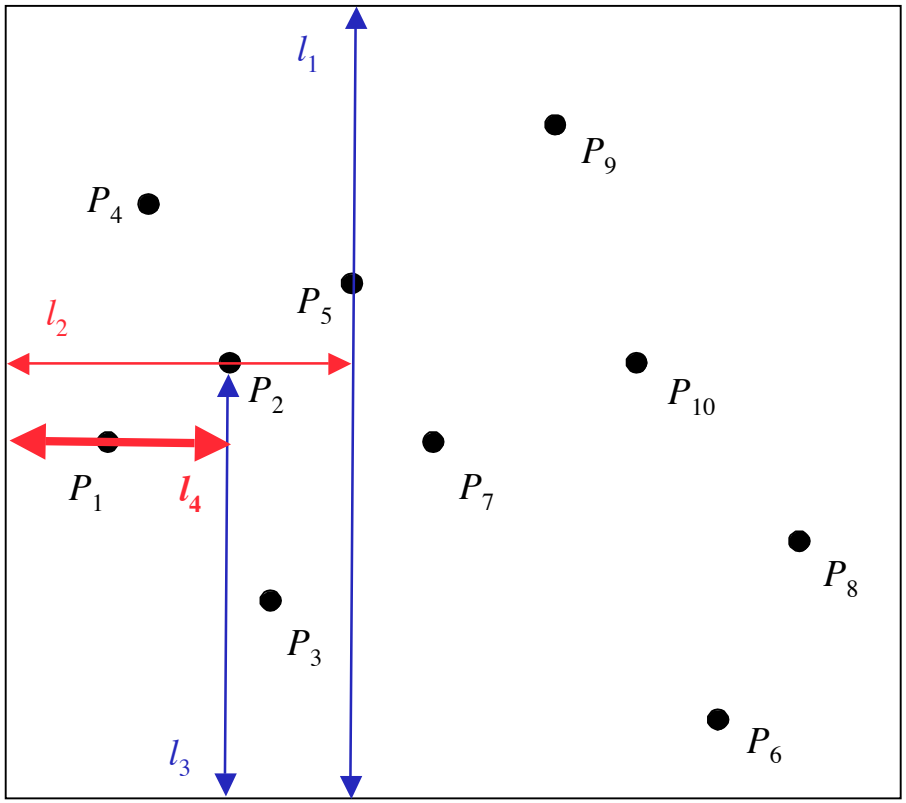
Konstruktion af *k*-d-Tree



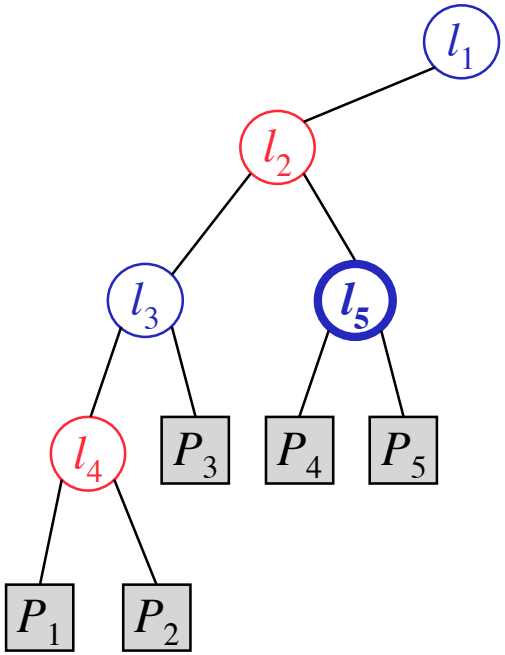


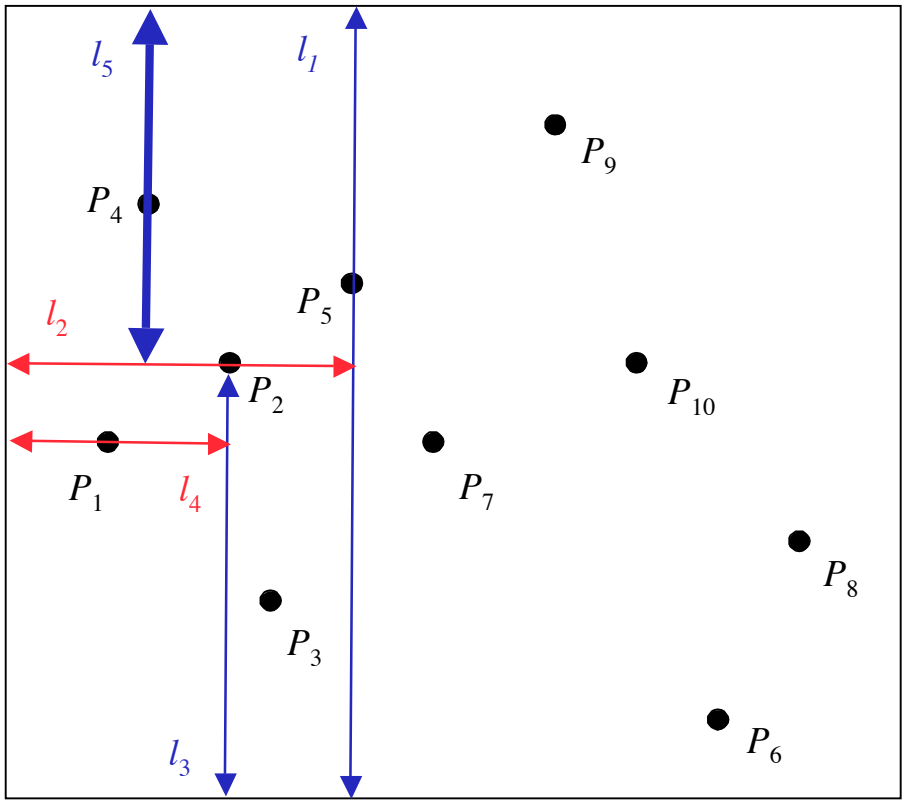
Konstruktion af *k*-d-Tree



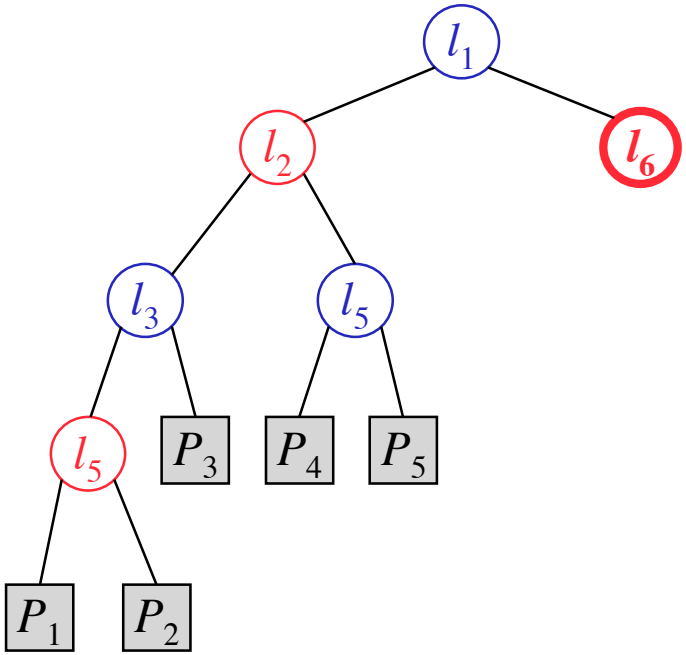


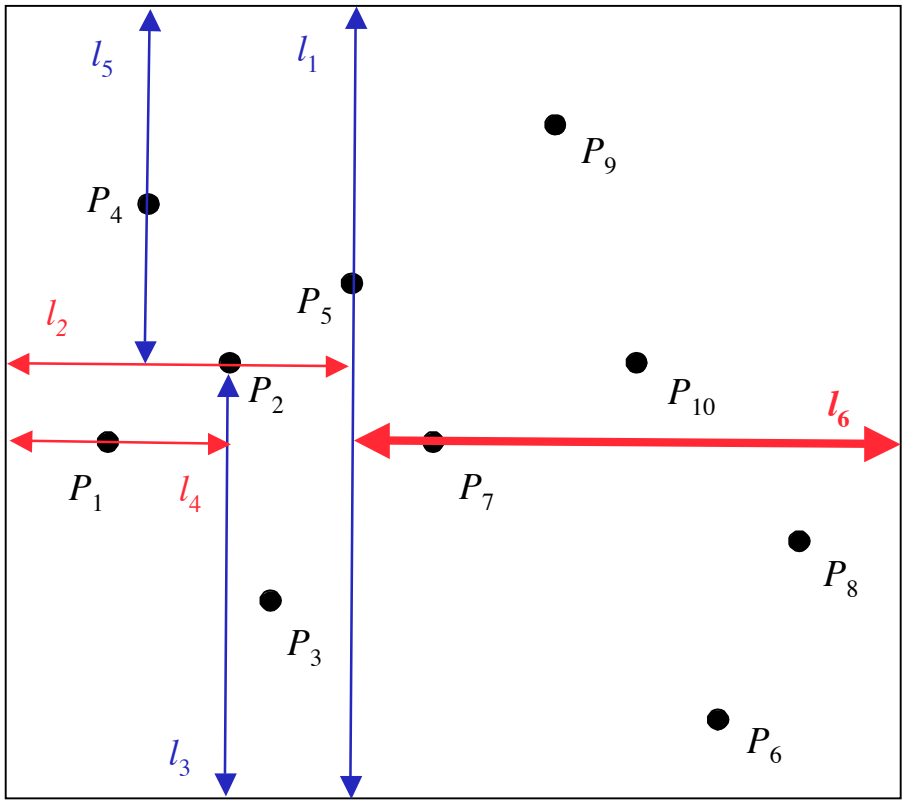
Konstruktion af *k*-d-Tree



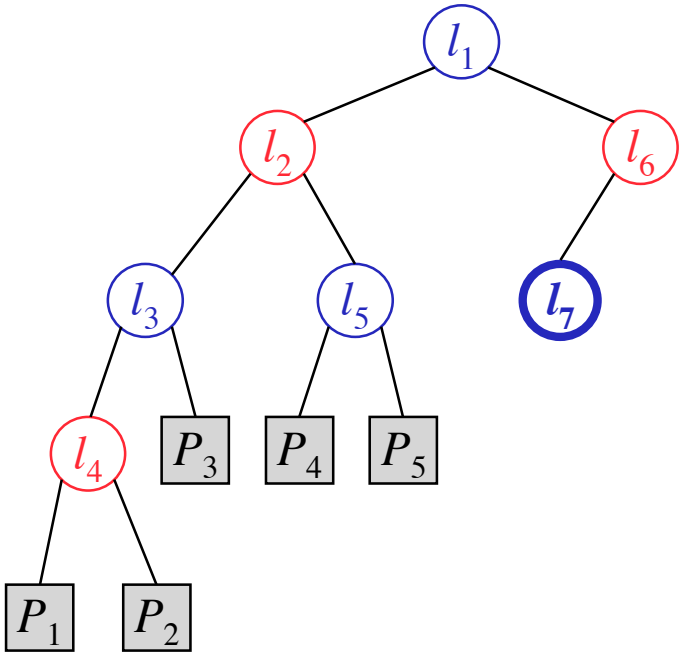


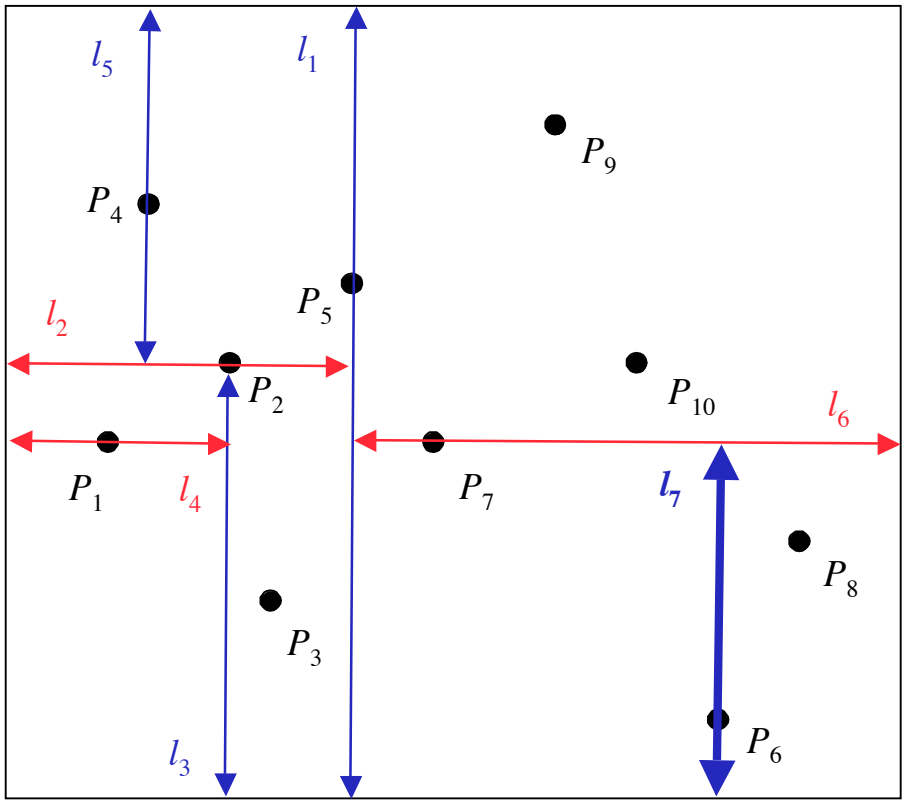
Konstruktion af *k*-d-Tree



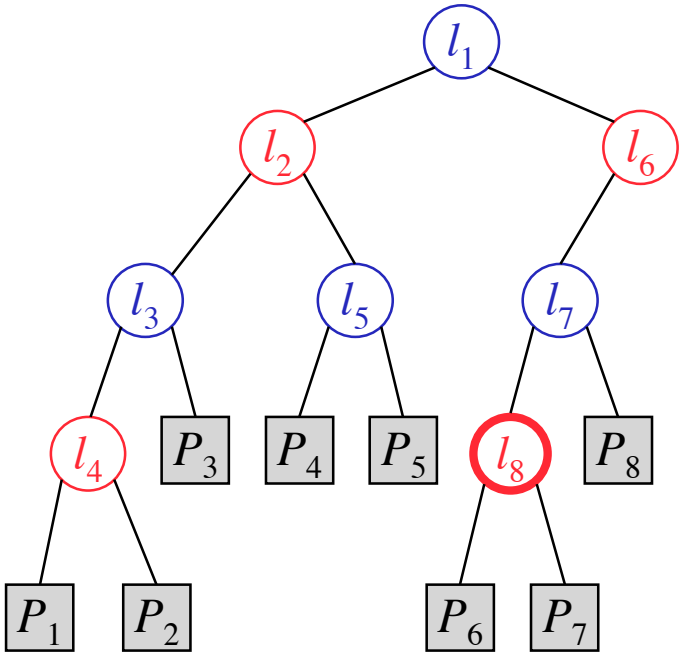


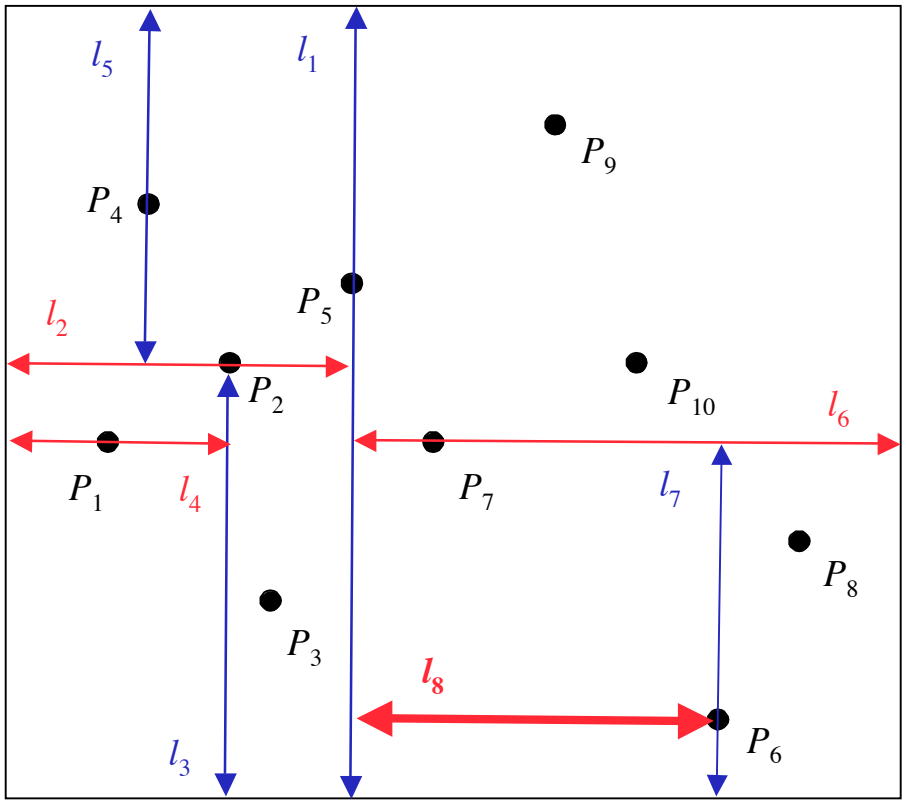
Konstruktion af *k*-d-Tree



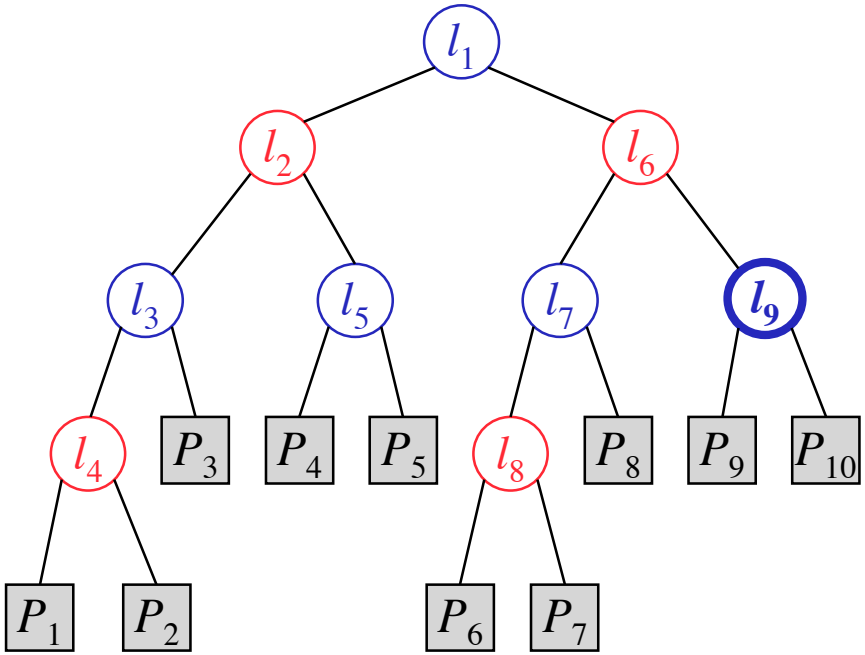


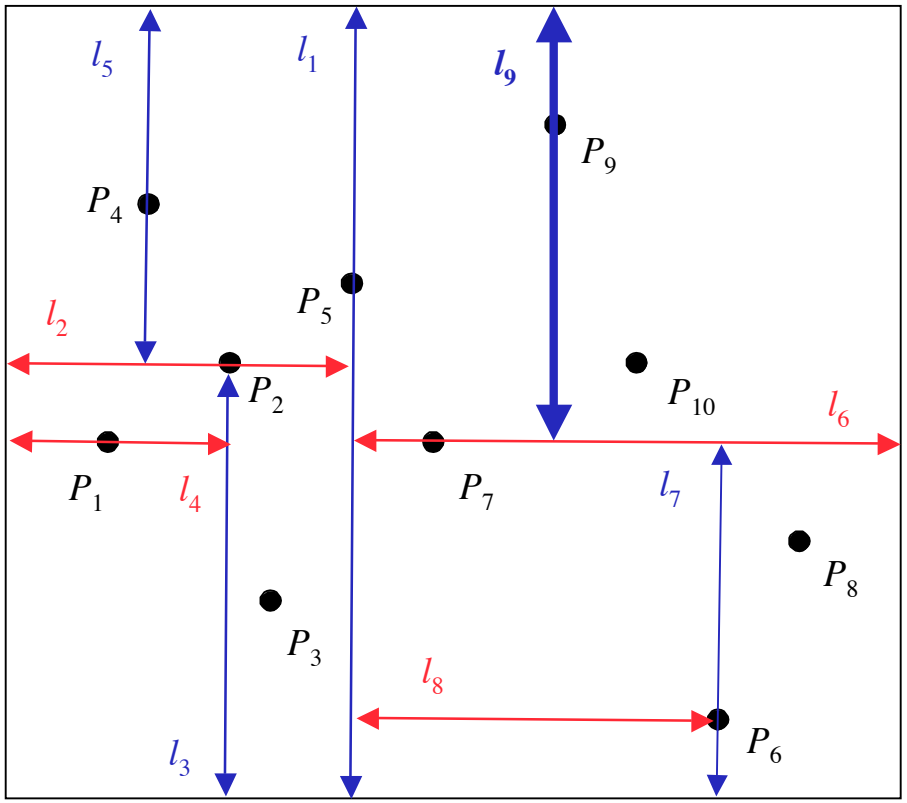
Konstruktion af *k*-d-Tree



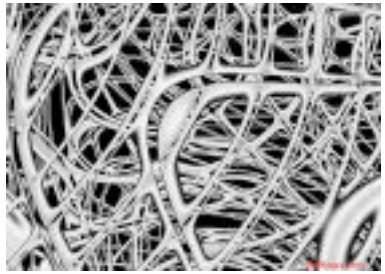


Konstruktion af *k*-d-Tree





Kompleksitet



Konstruktion af et k-d-træ kan foretages i $O(n \log n)$ tid
Medianbestemmelse kan ske med Quickselect i lineær tid

Søgning kan i værste tilfælde være $O(dn)$, men i praksis er søgning hurtig

Et k-d-træ kan benyttes til at finde nærmeste nabo til et punkt:

(1) Find ved søgning efter punktet et lille rektangel, der indeholder punktet
(2) Benyt dette rektangel til en intervalsøgning.
Formindsk om muligt rektanglet undervejs
Køretiden er typisk $O(\log n)$