# Computational Geometry

## Orthogonal Range Searching

**Efi Fogel**

Tel Aviv University 

CS School, Spring 2012

# Outline



# Range Searching Definition

## Definition (Range Searching)

Given a set $S$ of $n$ points in $\mathbb{R}^d$, preprocess $S$ into a data structure (if necessary) and answer a series of queries about the data. Queries include a $d$-dimensional region $r$ and may be any of the following types:

**Emptiness:** Does $r \cap S = \emptyset$?

**Reporting:** Output $r \cap S$.

**Counting:** Compute $|r \cap S|$.

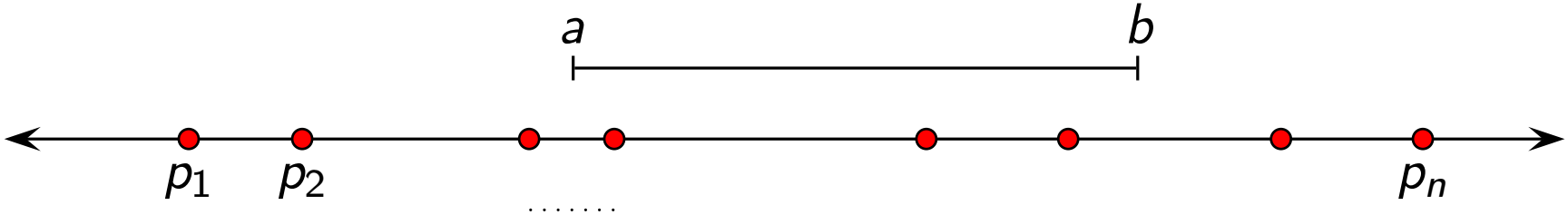
# Orthogonal Range Searching

- Queries are limited to $d$-dimensional axis-aligned rectangles
  - referred to as orthogonal (rectangular) range queries
- Input: $r = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d]$
- The naive query time is $O(n)$
- Many data structures can be used to speed up the process

# 1-Dimensional Orthogonal Range Searching

- The input $S$ is a set of points on the real line
- A query rectangle is an interval $r = [a, b]$
  - $k$ — number of points in the interval

- Preprocessing: Sort the points into an array
  - $O(n \log n)$ time, $O(n)$ space
- Query: $O(\log n + k)$ time, optimal
  - Perform a binary search for the first item larger than $a$, $O(\log n)$ time
  - Output every item until a point larger than $b$ is reached, $O(k)$ time




# 2-Dimensional Orthogonal Range Searching

- The input $S$ is a set of points in the plane
- A query is 2-dimensional axis-aligned rectangle $r = [a_1, b_1] \times [a_2, b_2]$
- Goal: maintain an output-sensitive query time of $O(\log n + k)$
- Two data structures are particularly useful in solving this problem:
  1. Kd-tree
  2. Range tree

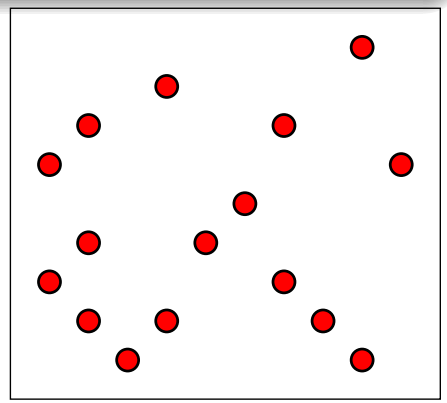
# 2-Dimensional Orthogonal Range Searching

- The input $S$ is a set of points in the plane
- A query is 2-dimensional axis-aligned rectangle $r = [a_1, b_1] \times [a_2, b_2]$
- Goal: maintain an output-sensitive query time of $O(\log n + k)$
- Two data structures are particularly useful in solving this problem:
  1. Kd-tree
  2. Range tree
     - Useful for higher dimension as well

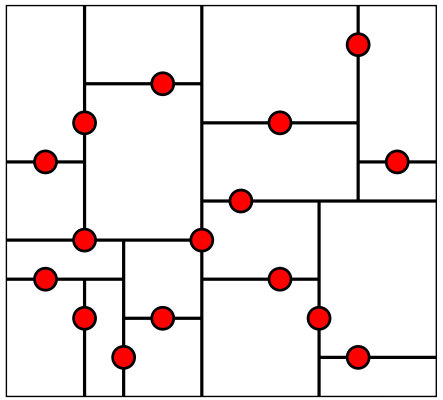
# Kd-trees

## Definition (Kd-tree)

A Kd-tree is a space-partitioning data structure for organizing points in a $k$-dimensional space.



- Each node stores a single point
- The dimension in which the median is found changes at each step
- In the plane:
  - Enclose all points with a rectangle
  - Divide the rectangle in the $x$-direction by median
  - Divide each region in the $y$-direction by median
  - Repeat until there is only one point in each region



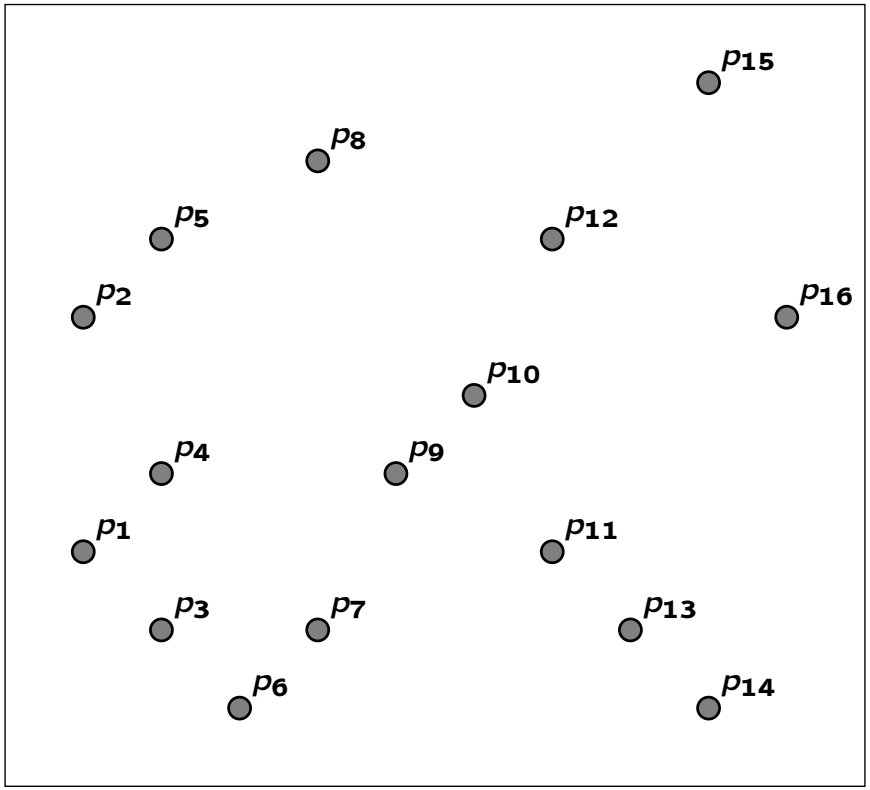
# d-Dimensional Kd-tree Preprocessing

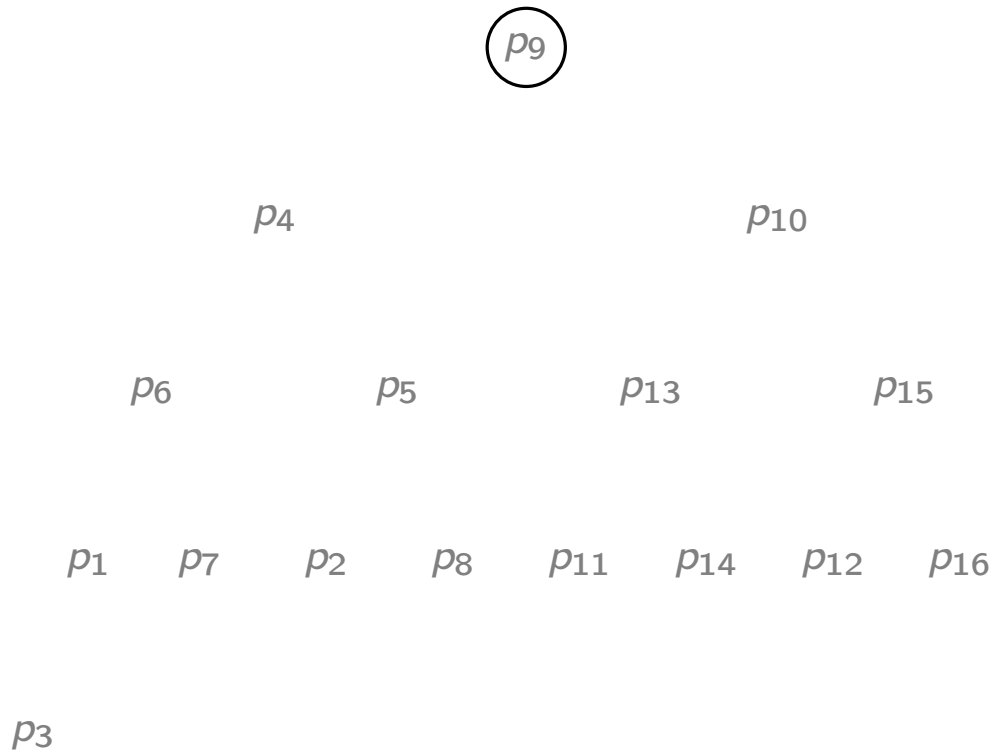
Store a set $S$ of points in a $d$-dimensional Kd-tree

$\mathrm{kdtree}(S, i, d)$

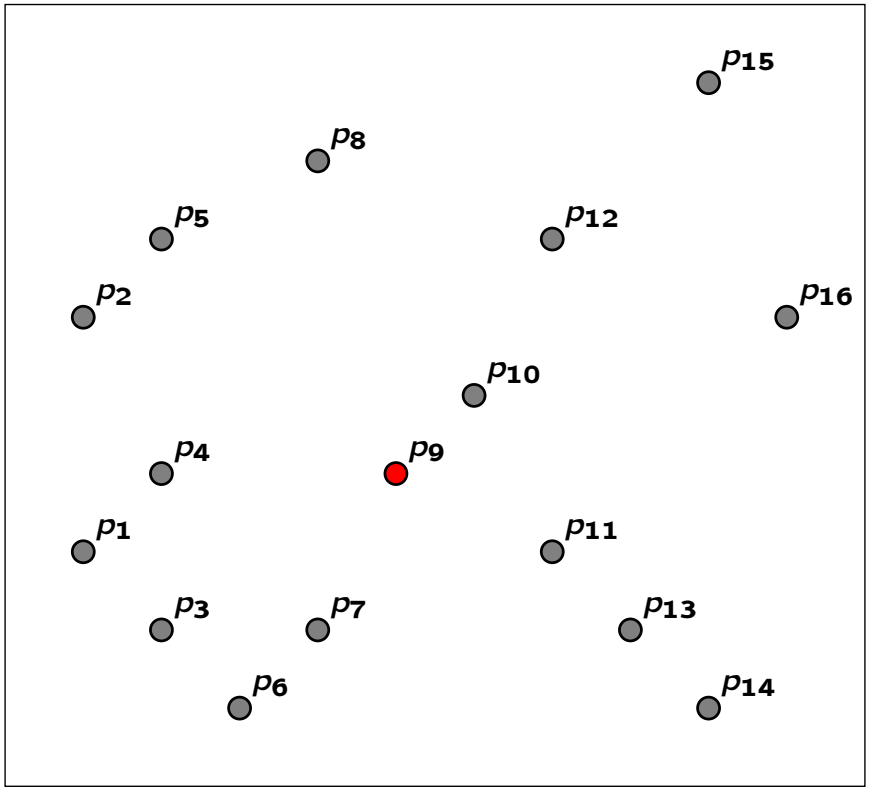
1. **if** $S = \emptyset$, **return null**
2. $p \leftarrow$ point in $S$ with median $x_i$-coordinate
3. Create a node $v$
4. $p(v) \leftarrow p$
5. $S_- \leftarrow \{q \in S \mid x_i(q) < x_i(p)\}$
6. $S_+ \leftarrow \{q \in S \mid x_i(q) > x_i(p)\}$
7. $\mathrm{left}(v) \leftarrow \mathrm{kdtree}(S_-, (i+1) \mod d, d)$
8. $\mathrm{right}(v) \leftarrow \mathrm{kdtree}(S_+, (i+1) \mod d, d)$
9. **return** $v$

# 2-Dimensional Kd-tree Construction



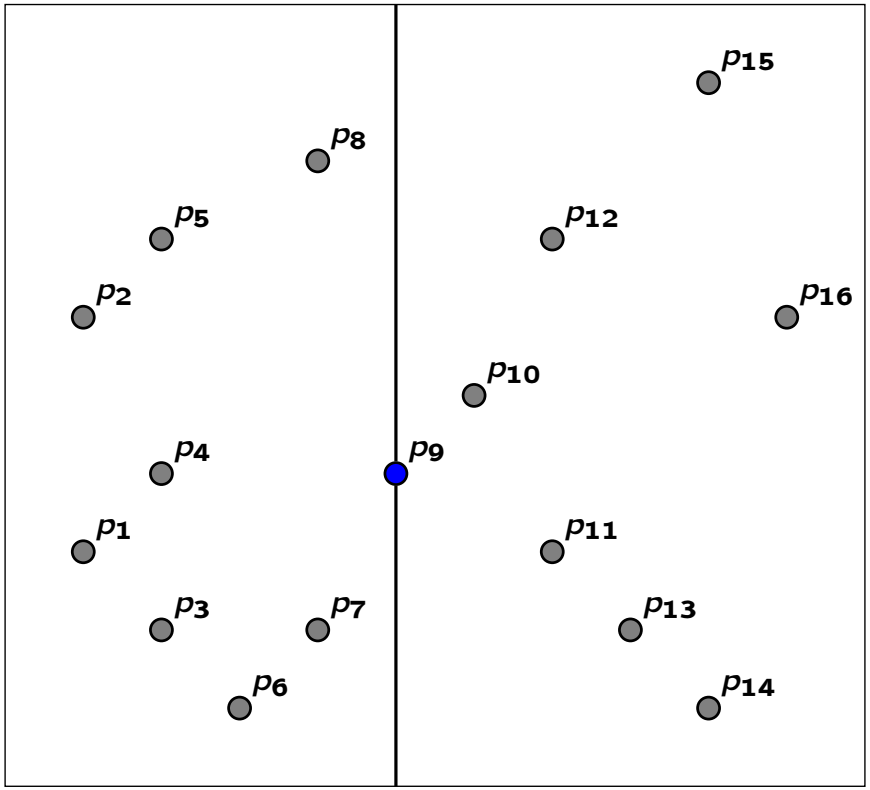



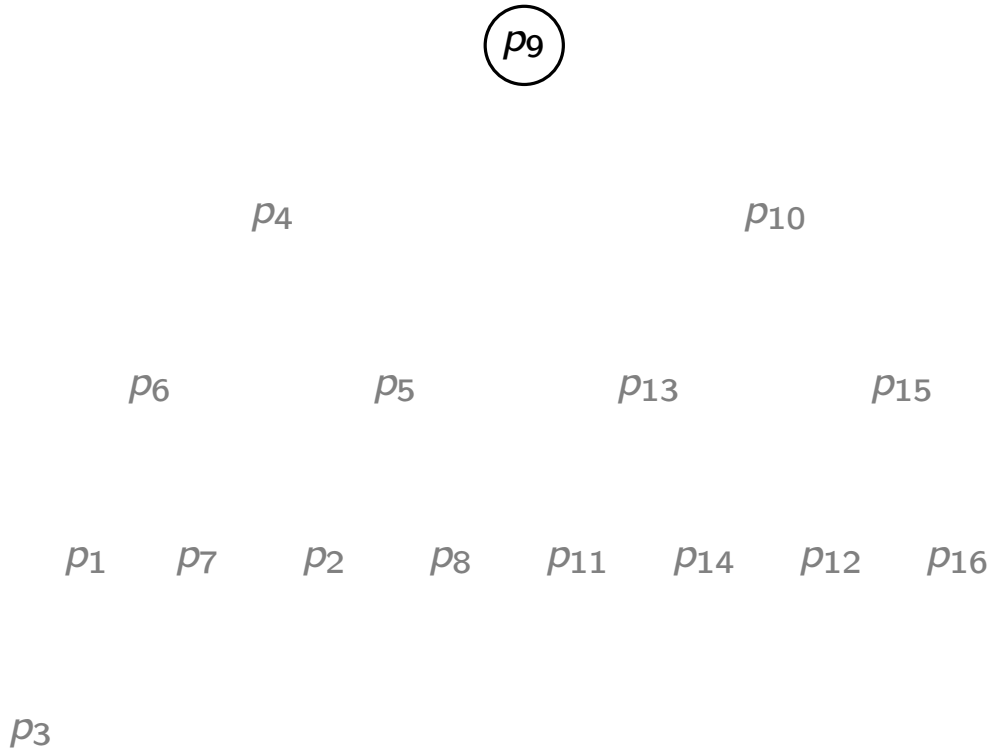
# 2-Dimensional Kd-tree Construction



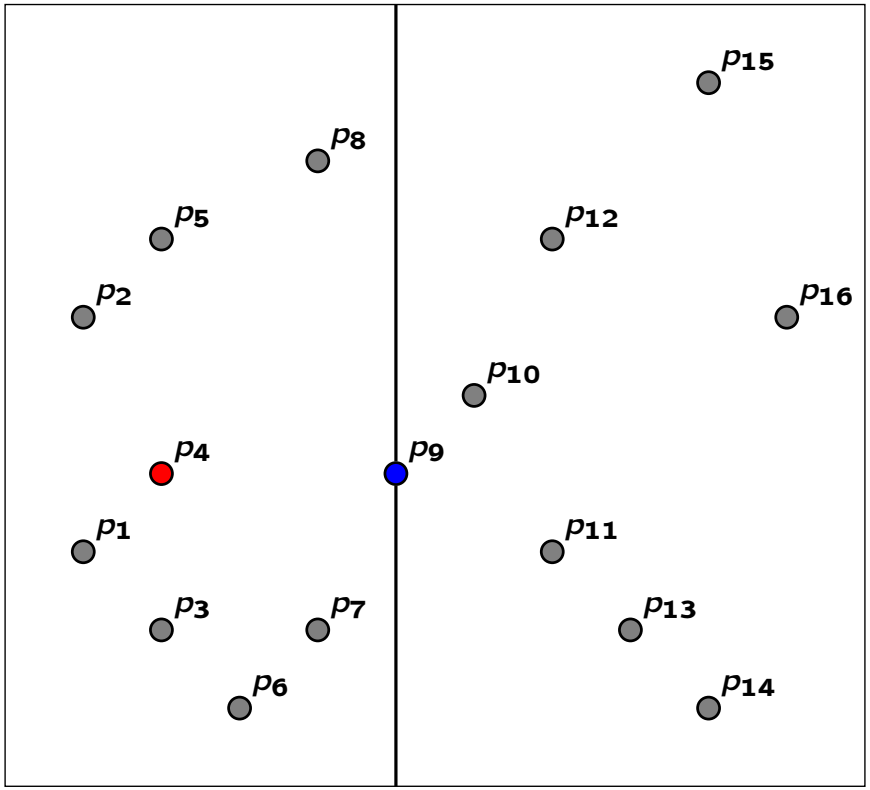


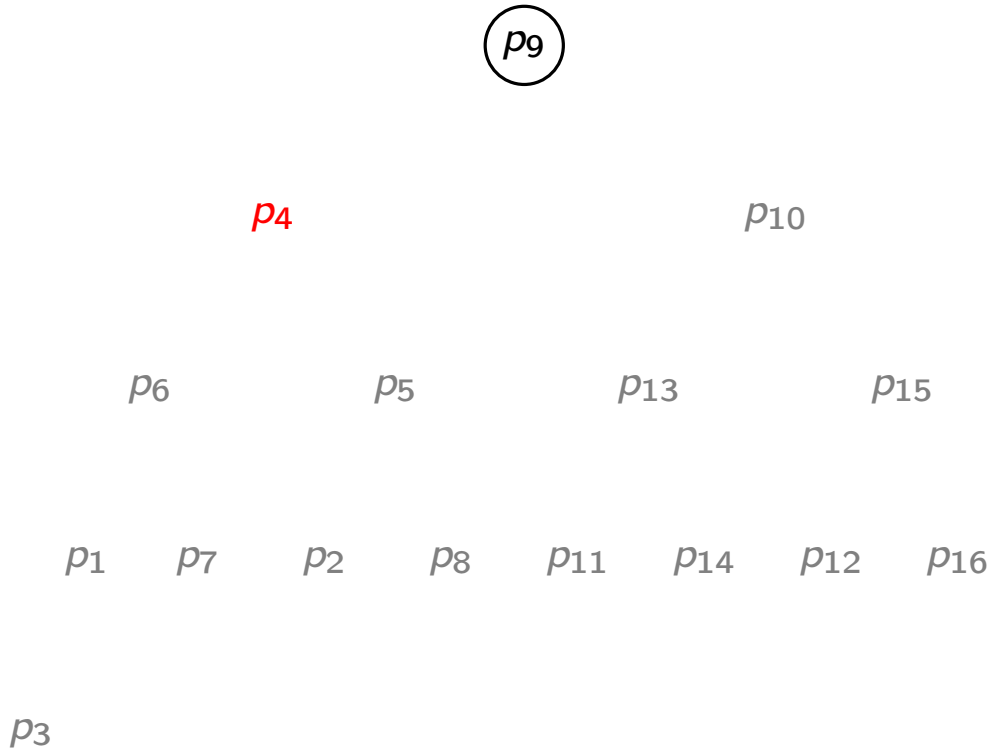
# 2-Dimensional Kd-tree Construction



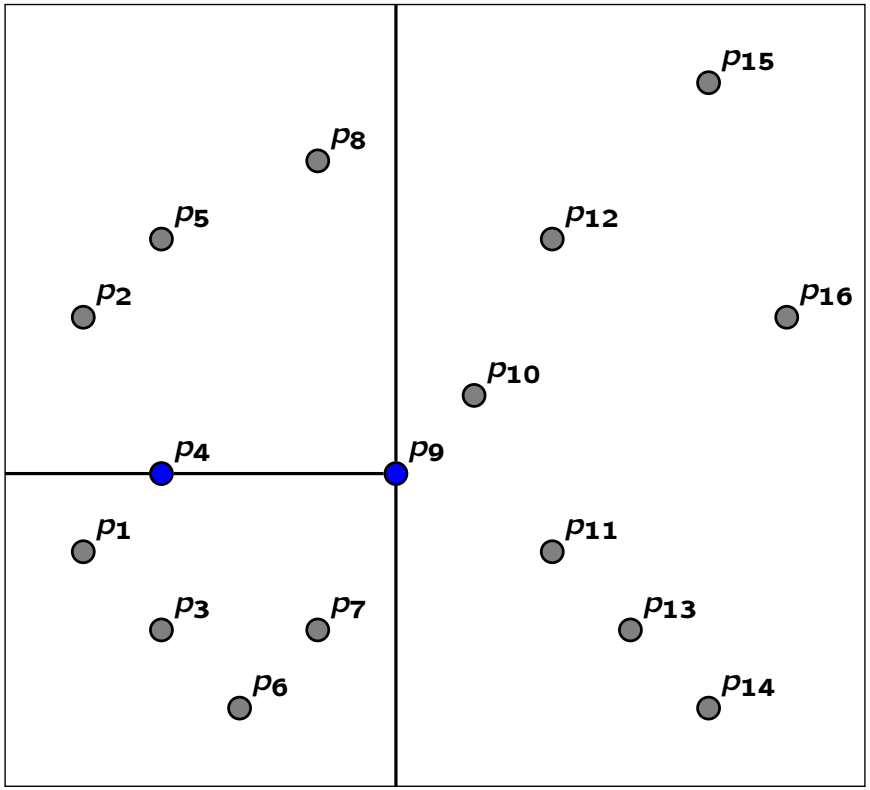



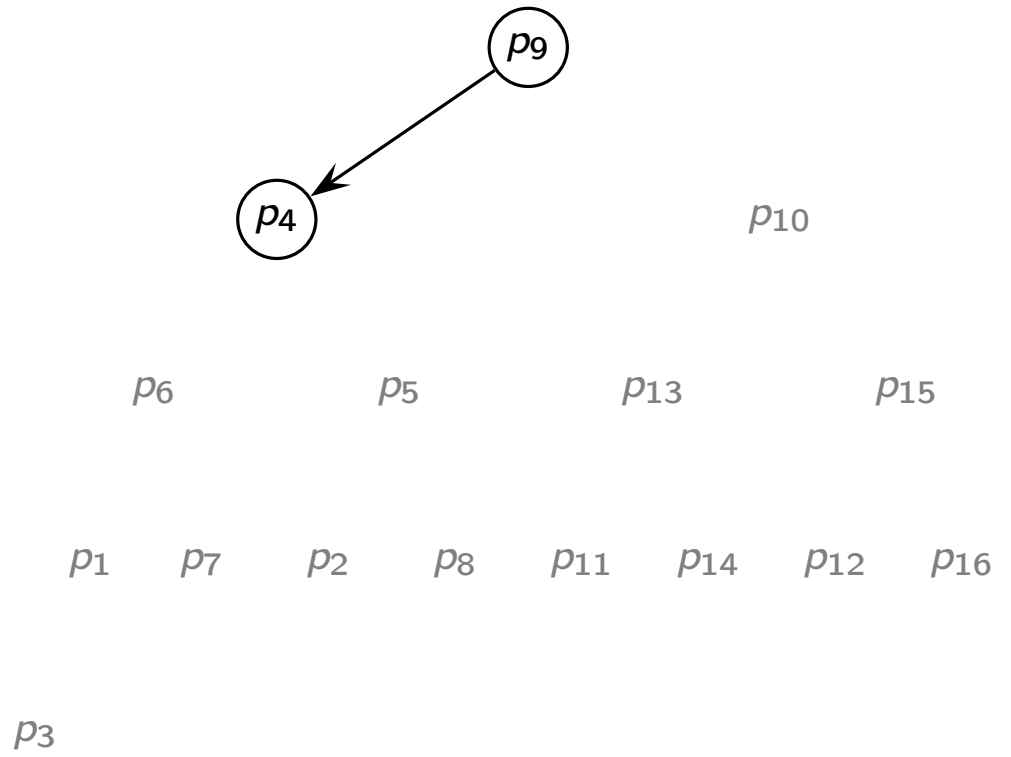
# 2-Dimensional Kd-tree Construction



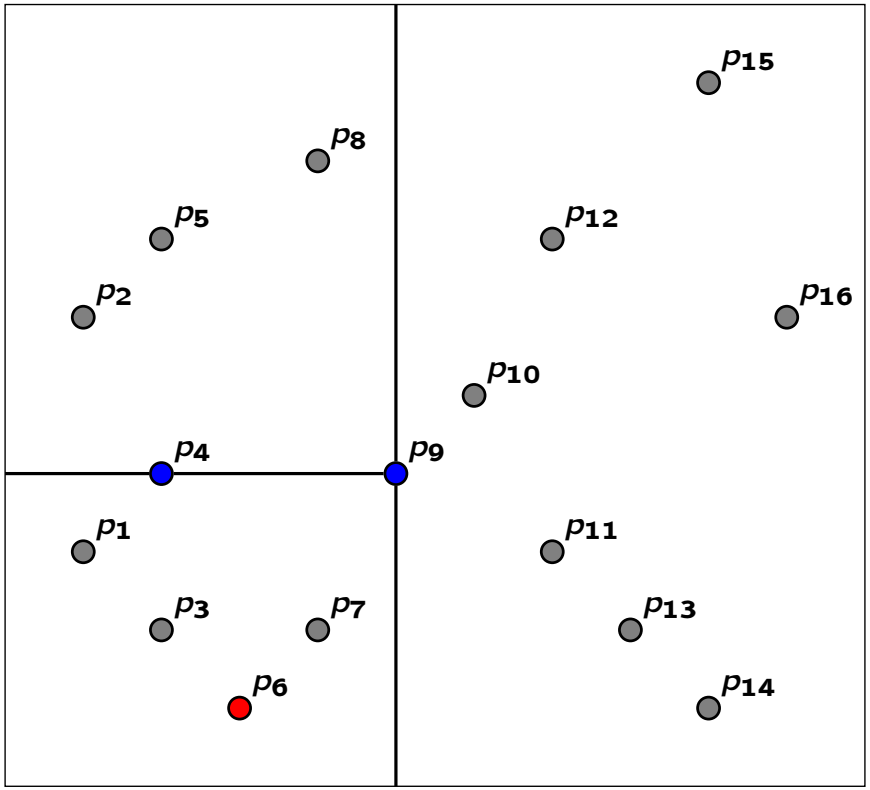



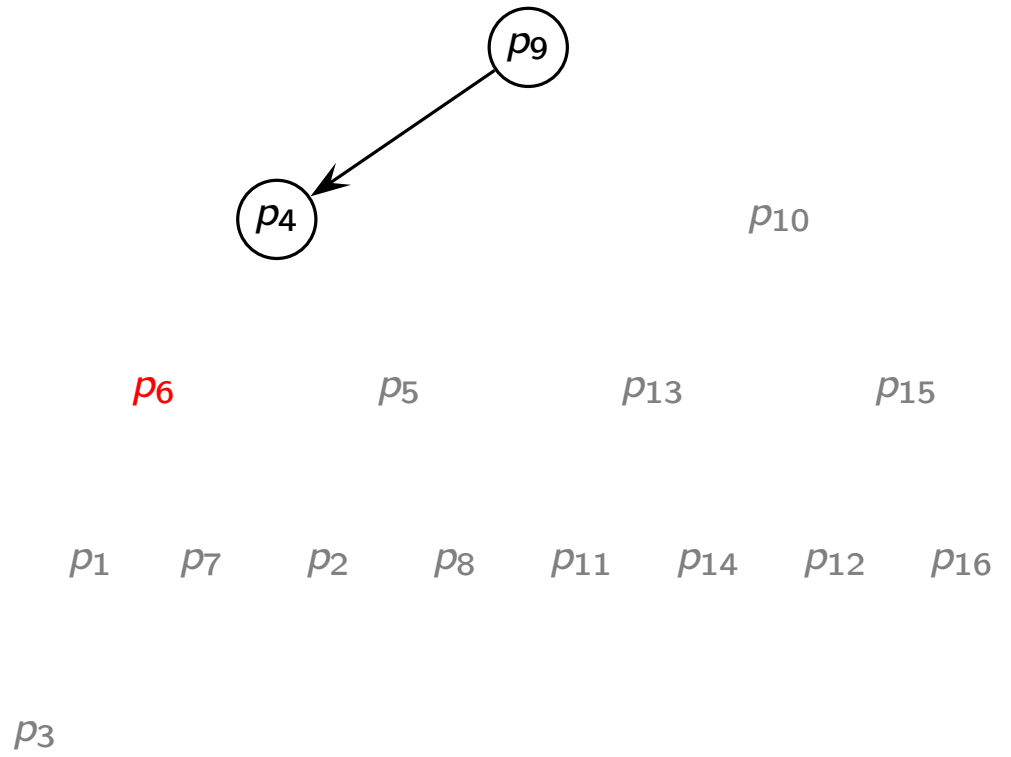
# 2-Dimensional Kd-tree Construction



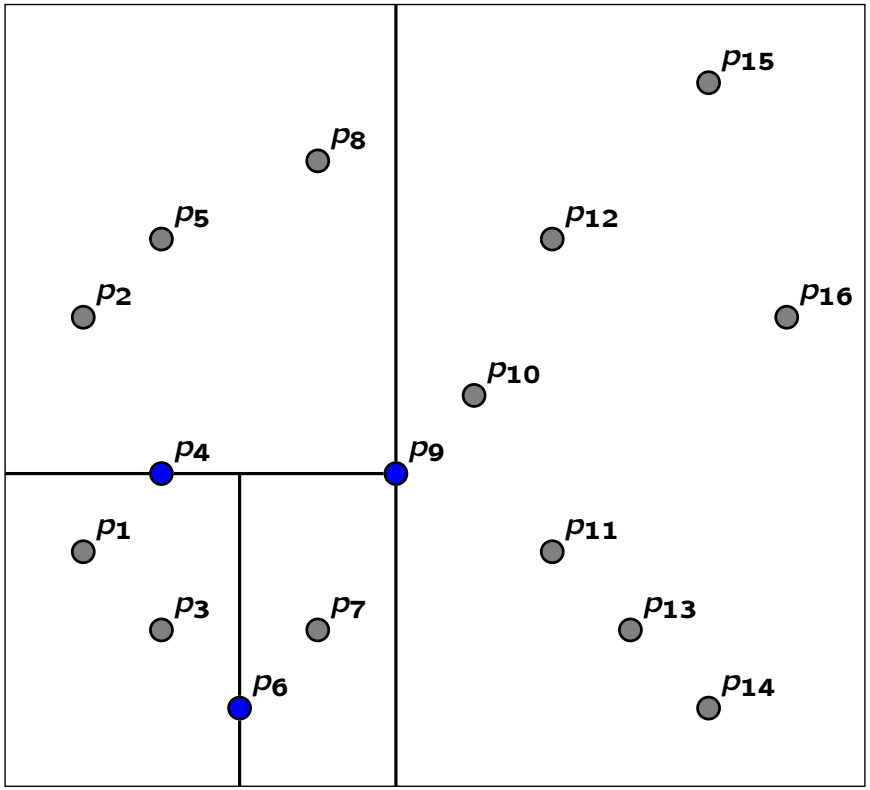



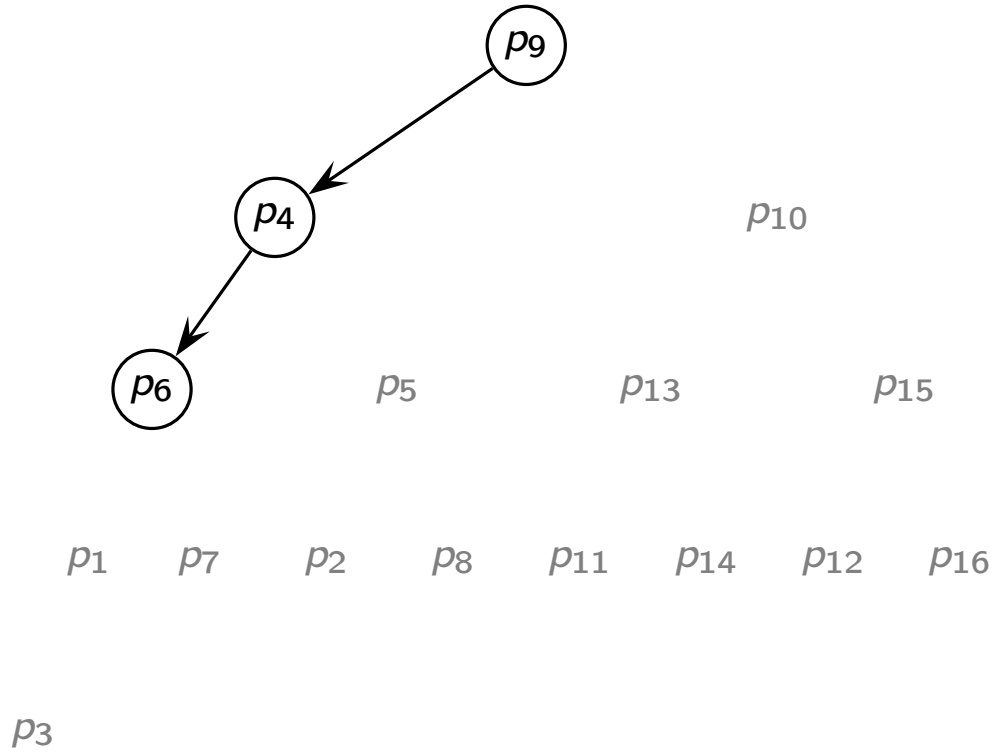
# 2-Dimensional Kd-tree Construction



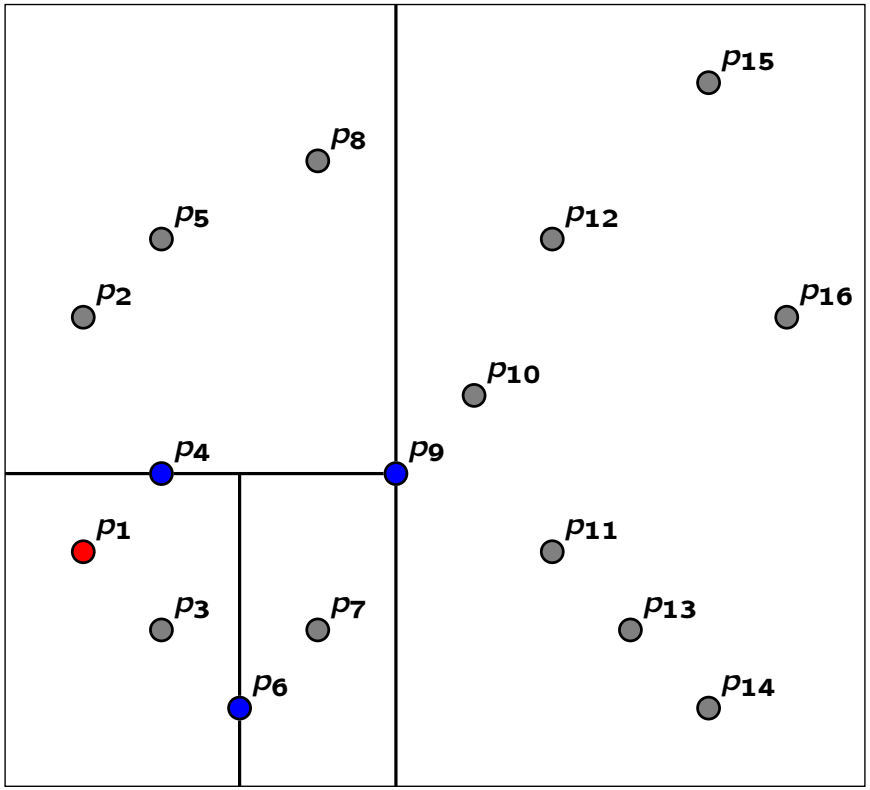



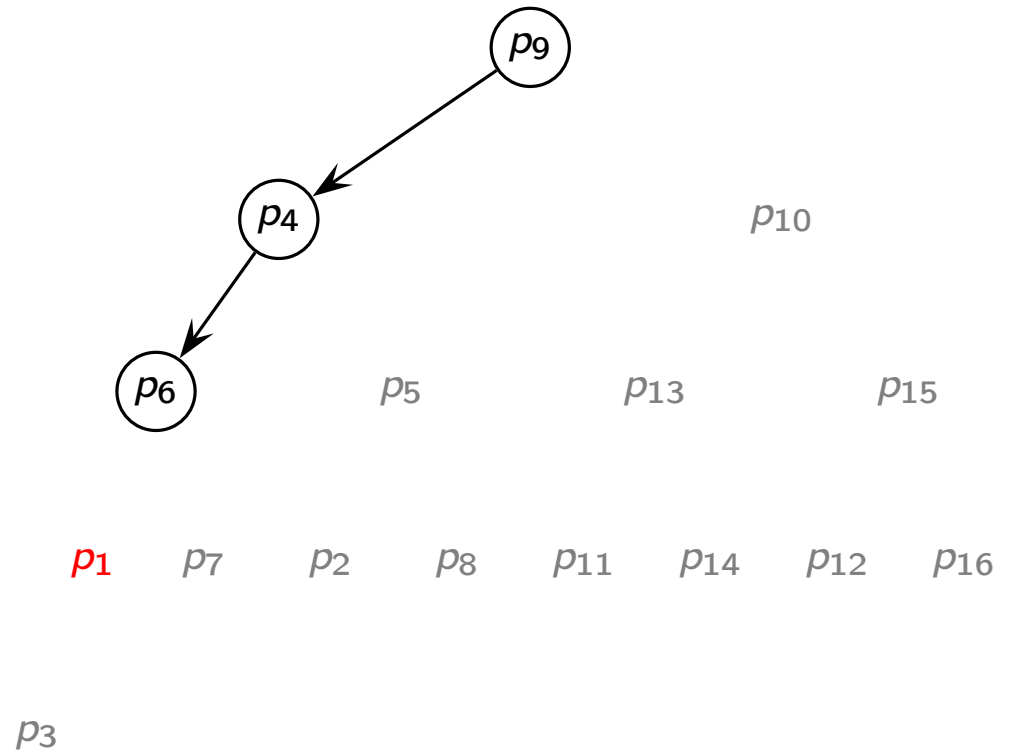
# 2-Dimensional Kd-tree Construction



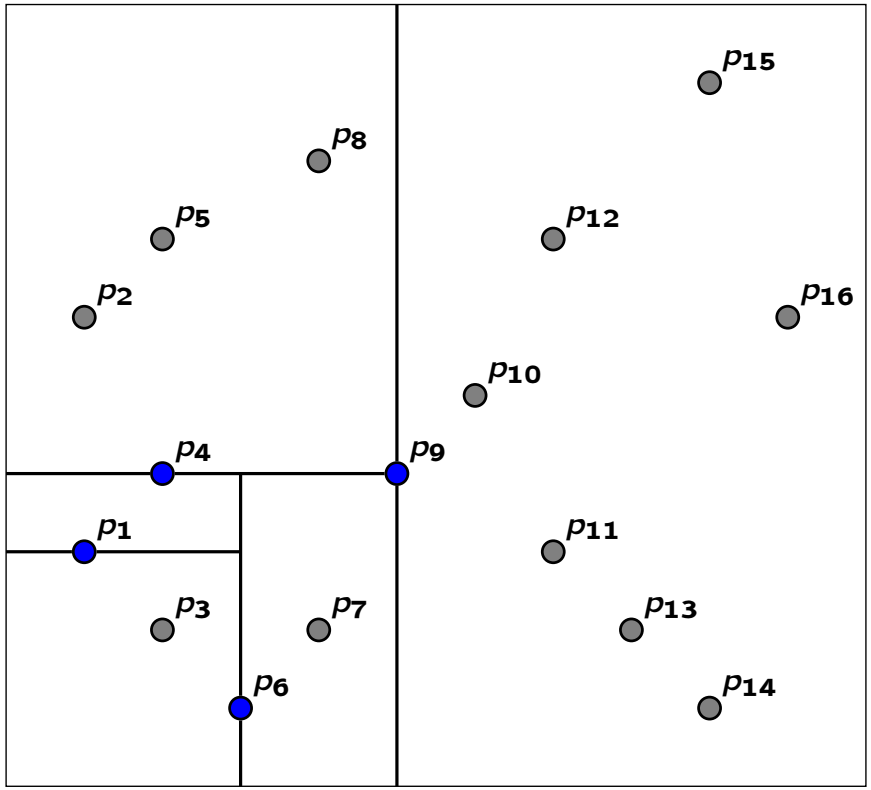



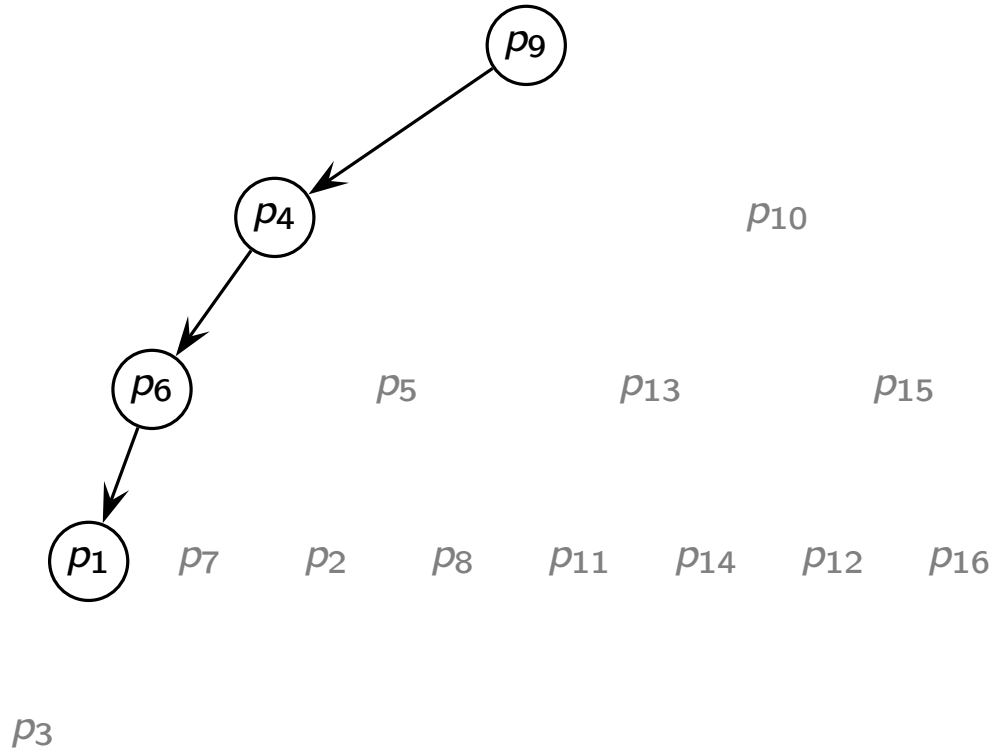

# 2-Dimensional Kd-tree Construction



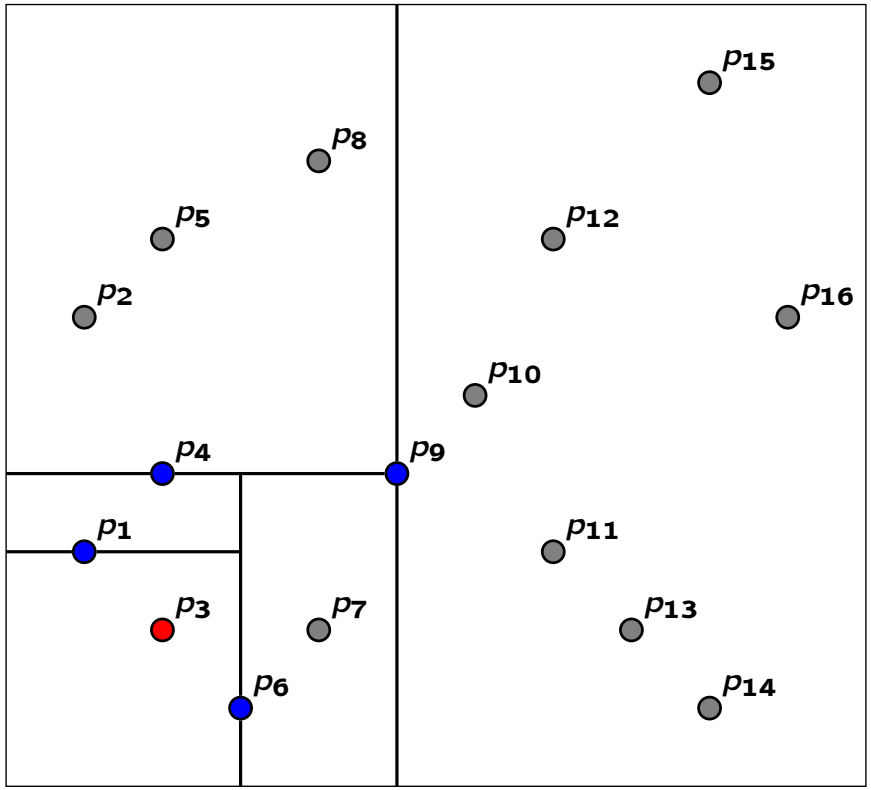



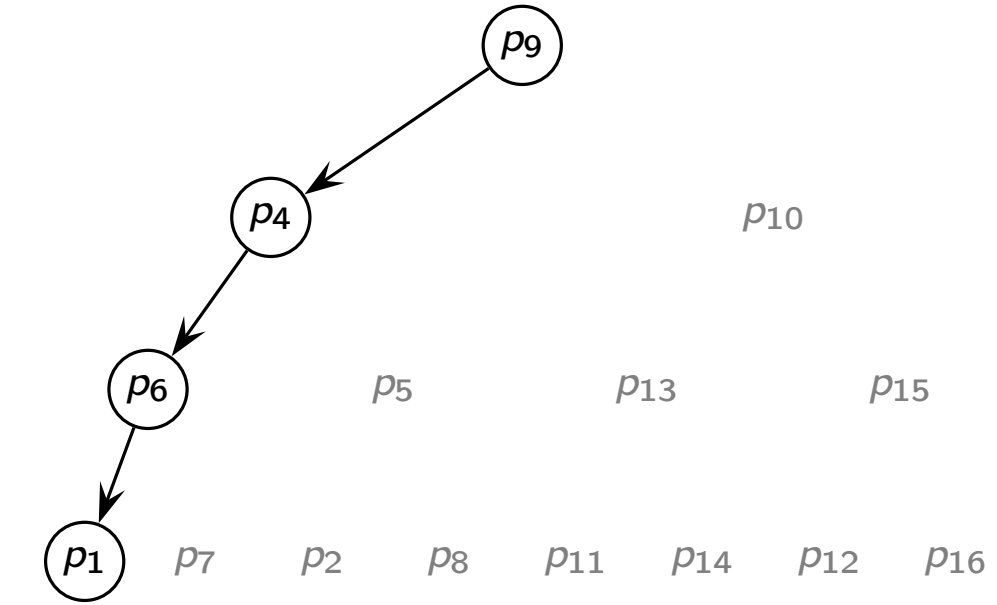
# 2-Dimensional Kd-tree Construction



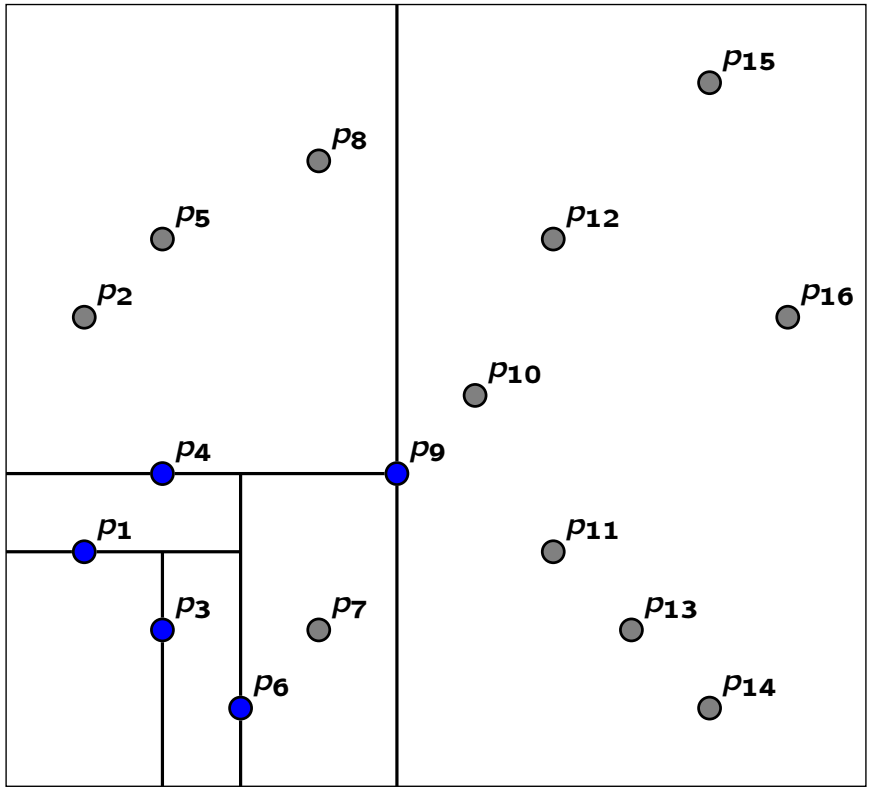
# 2-Dimensional Kd-tree Construction

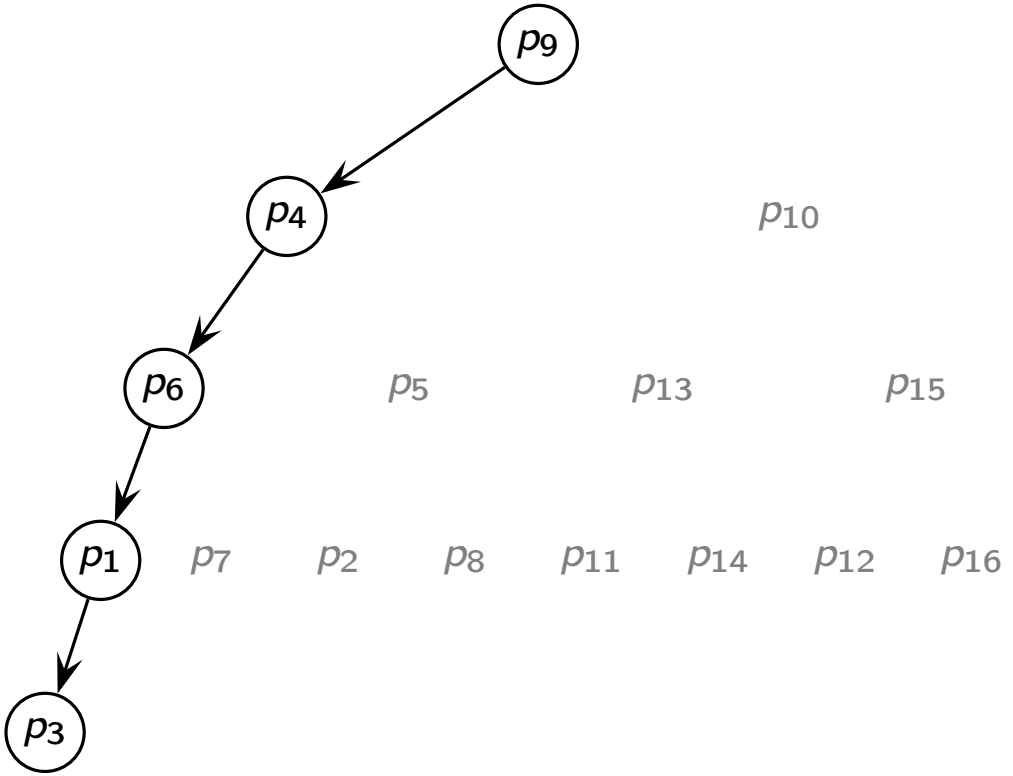


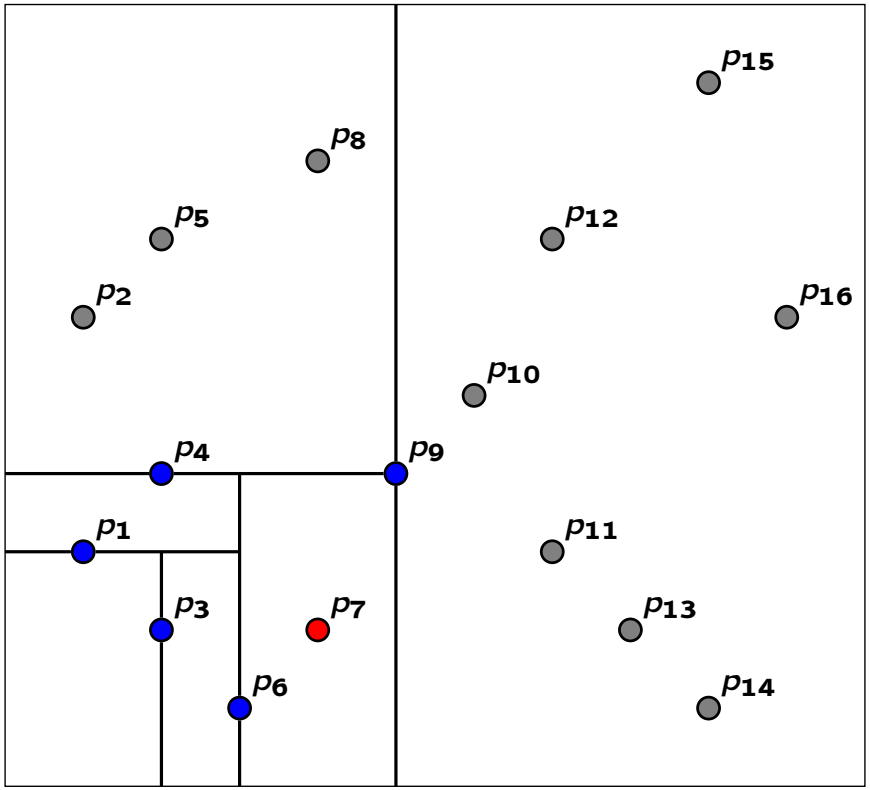

$p_3$

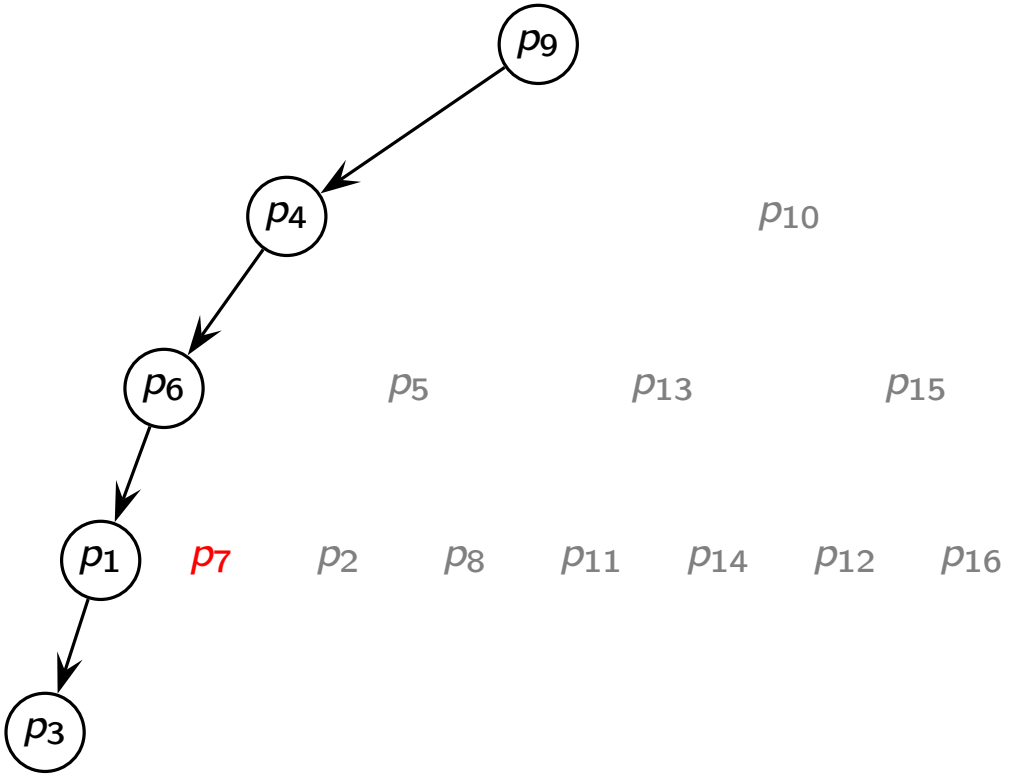
# 2-Dimensional Kd-tree Construction



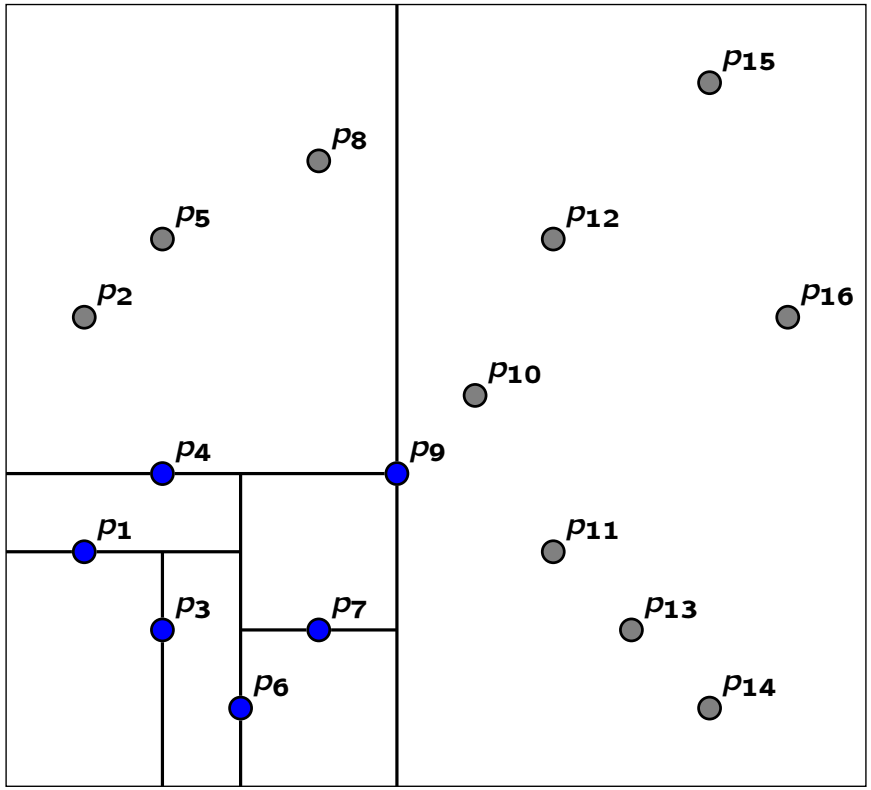



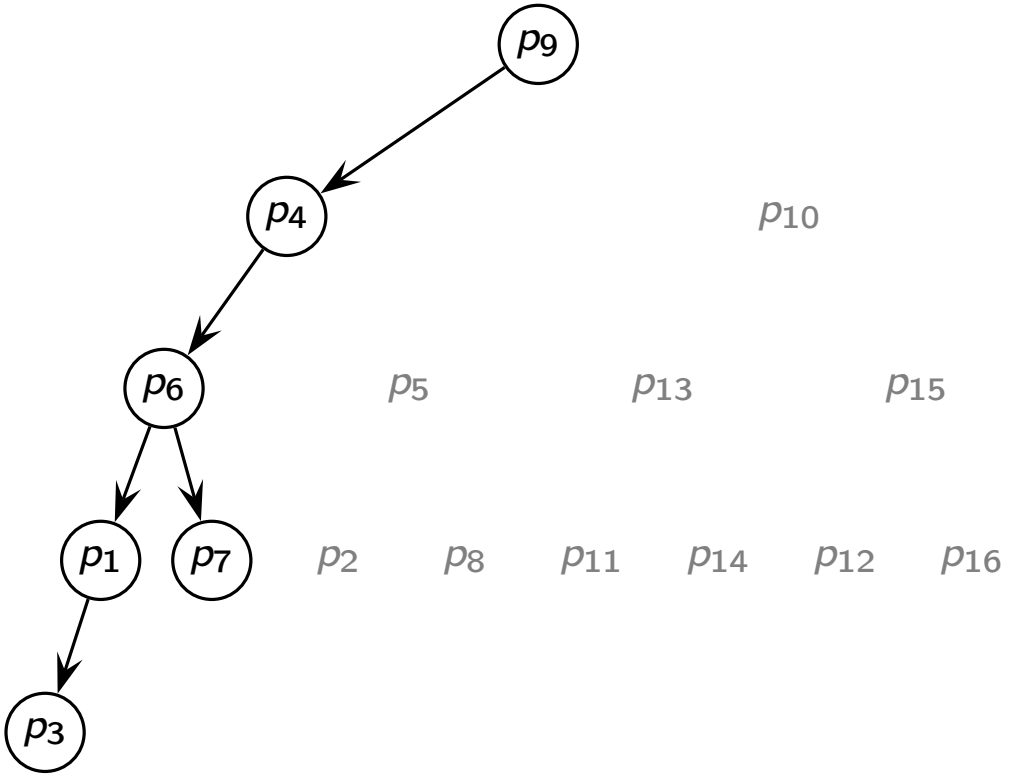

# 2-Dimensional Kd-tree Construction



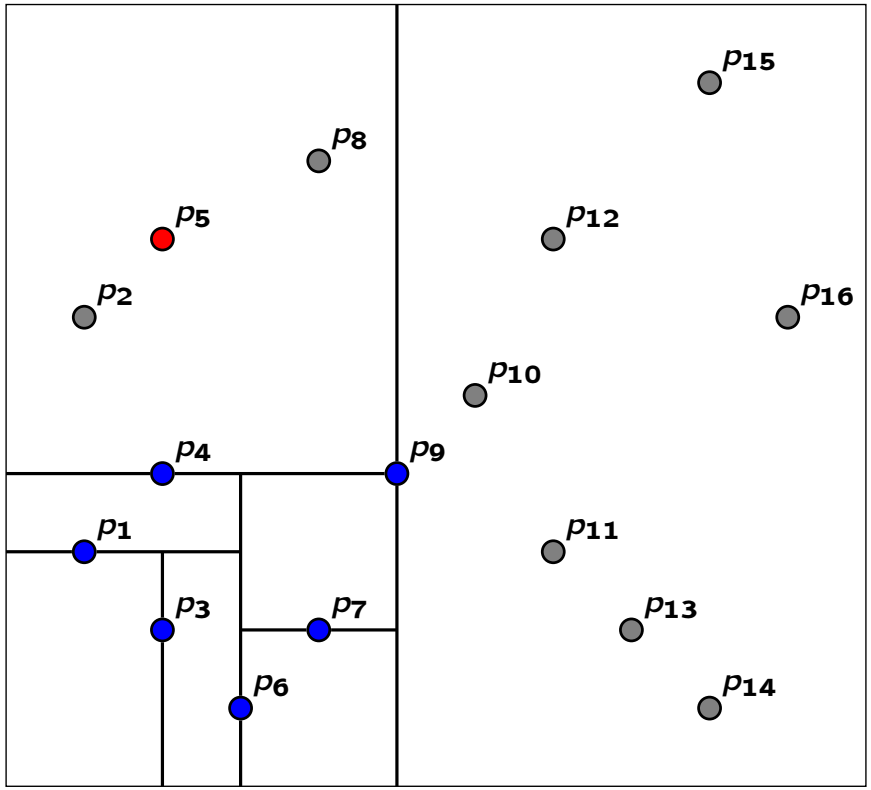



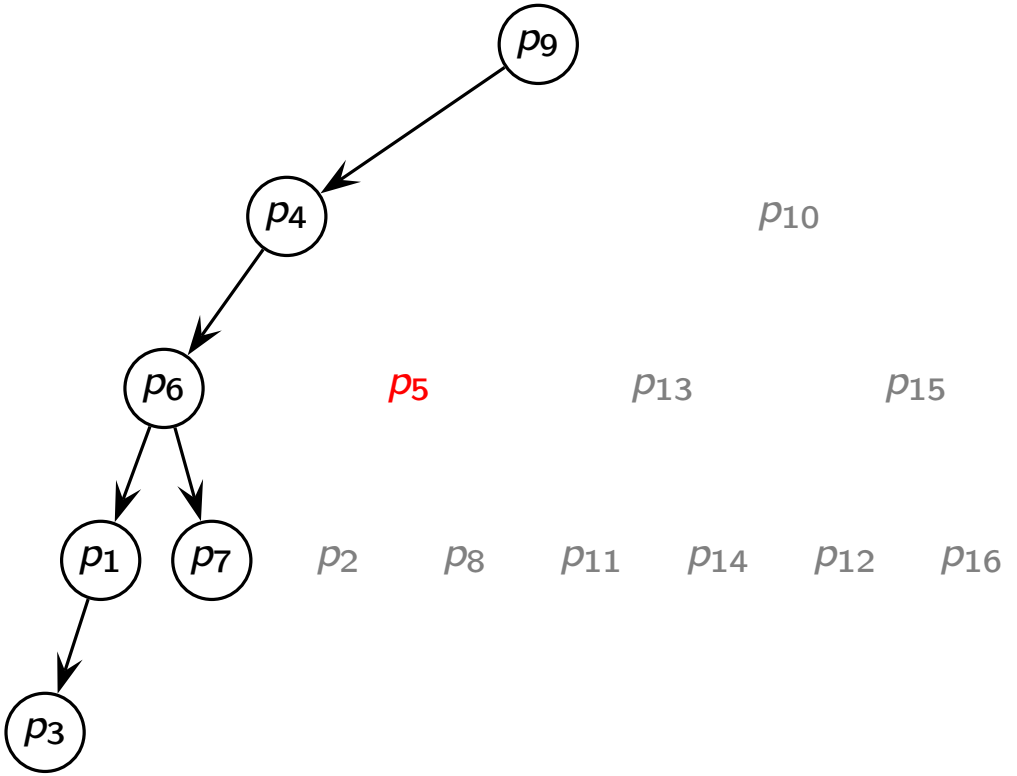
# 2-Dimensional Kd-tree Construction



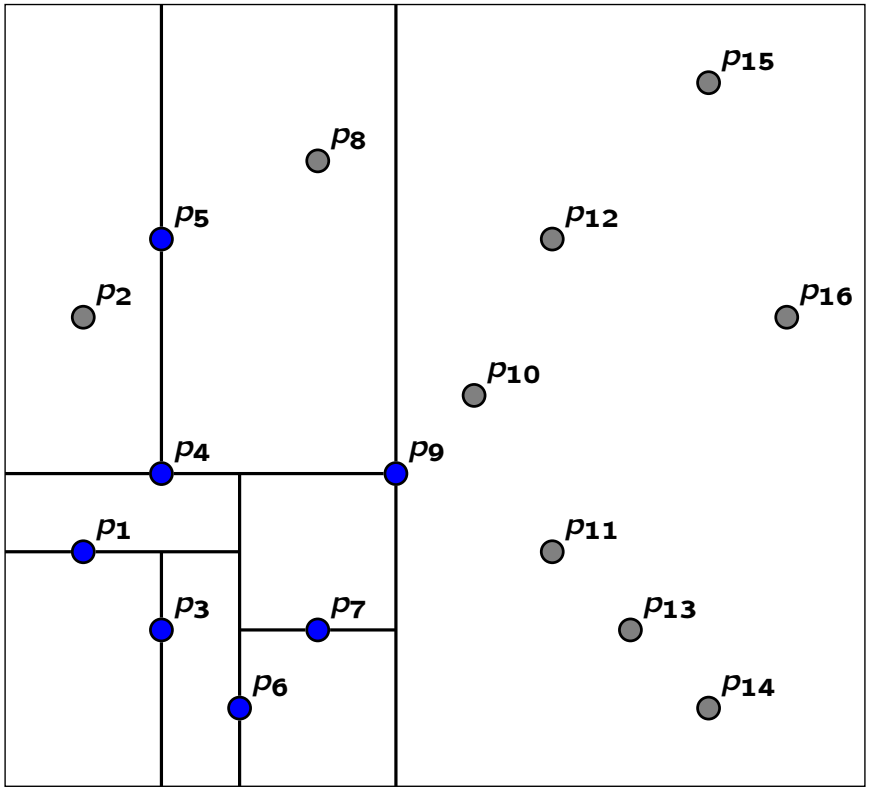



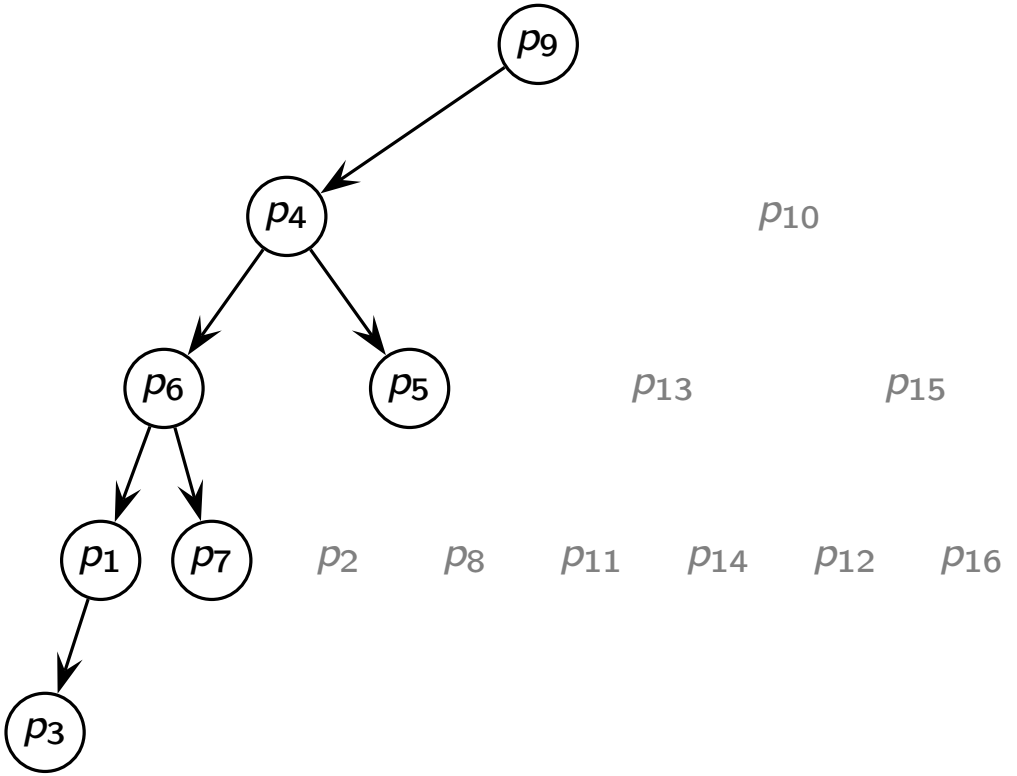
# 2-Dimensional Kd-tree Construction



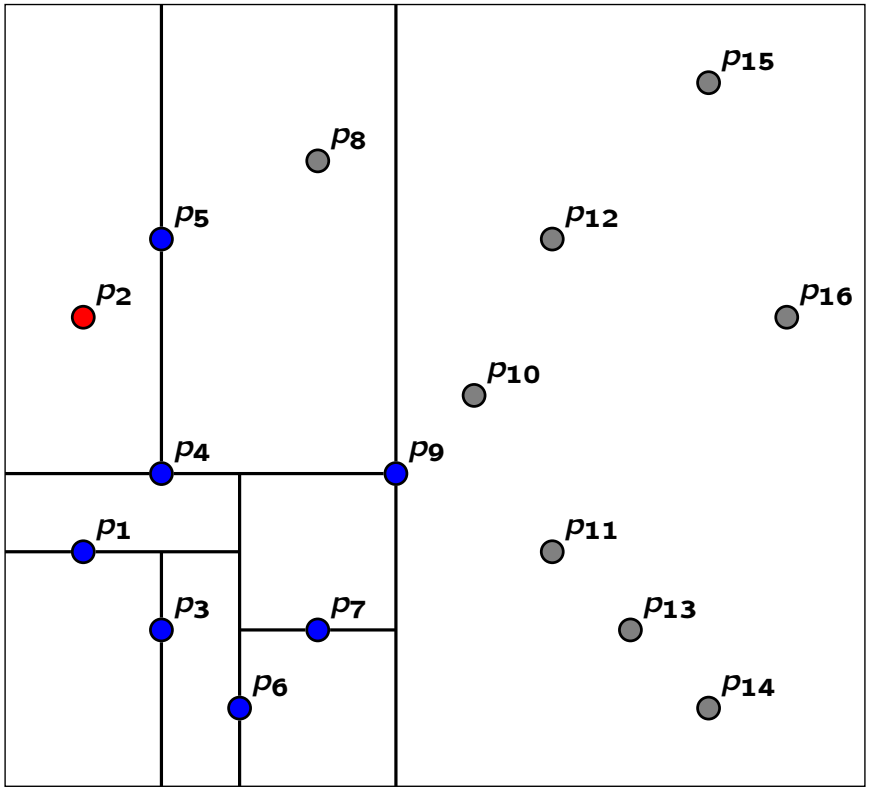



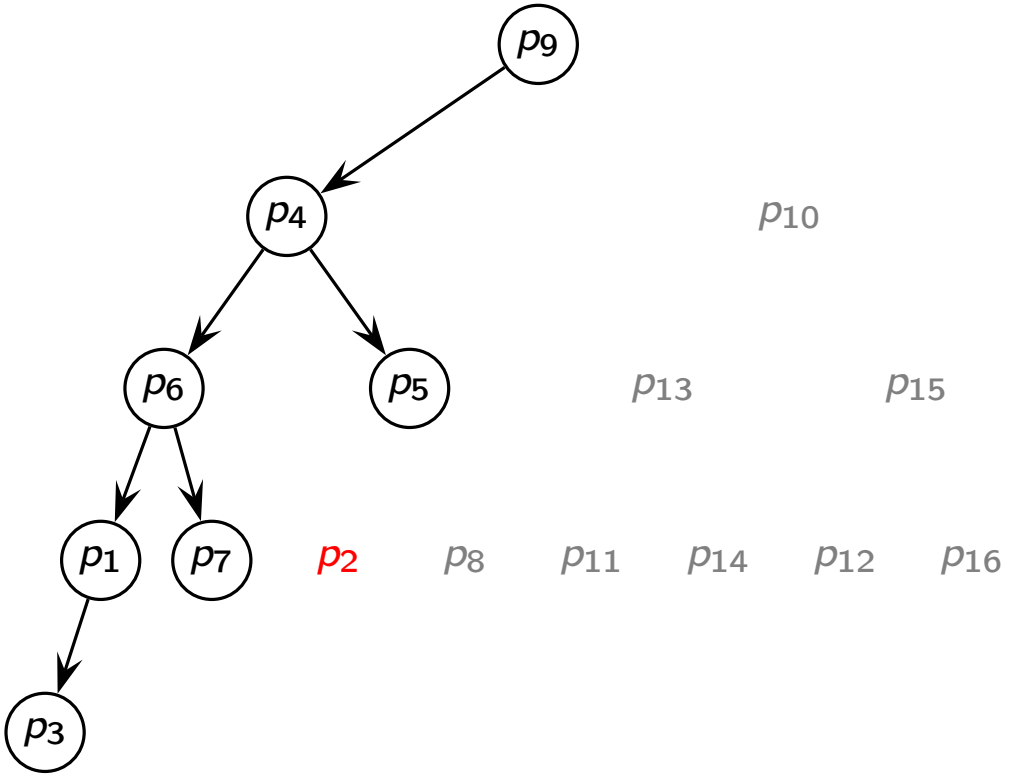
# 2-Dimensional Kd-tree Construction



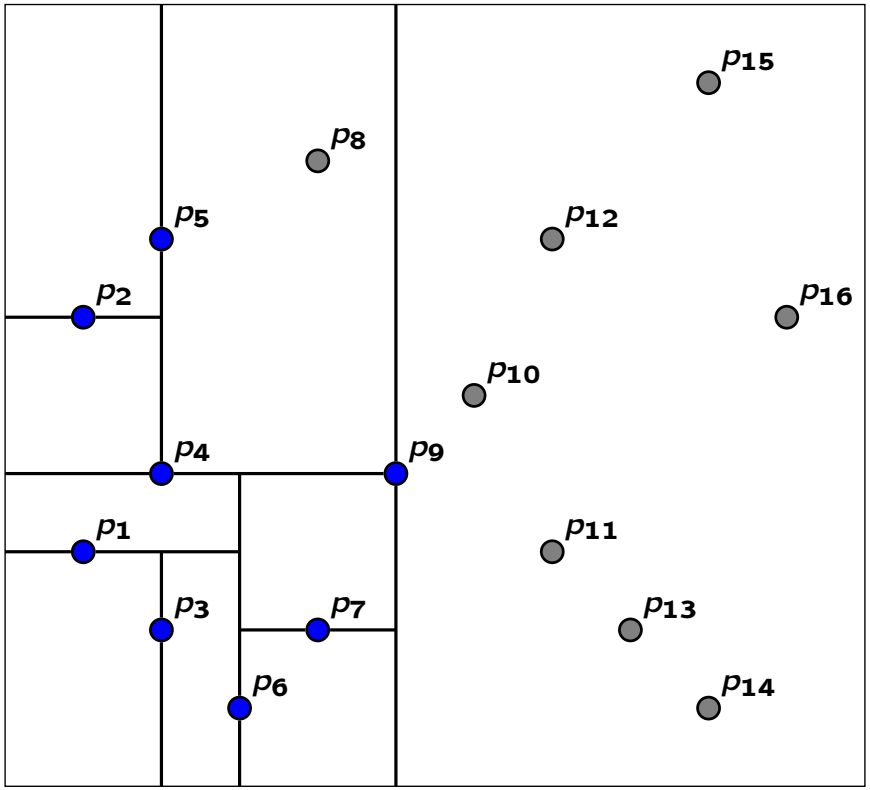



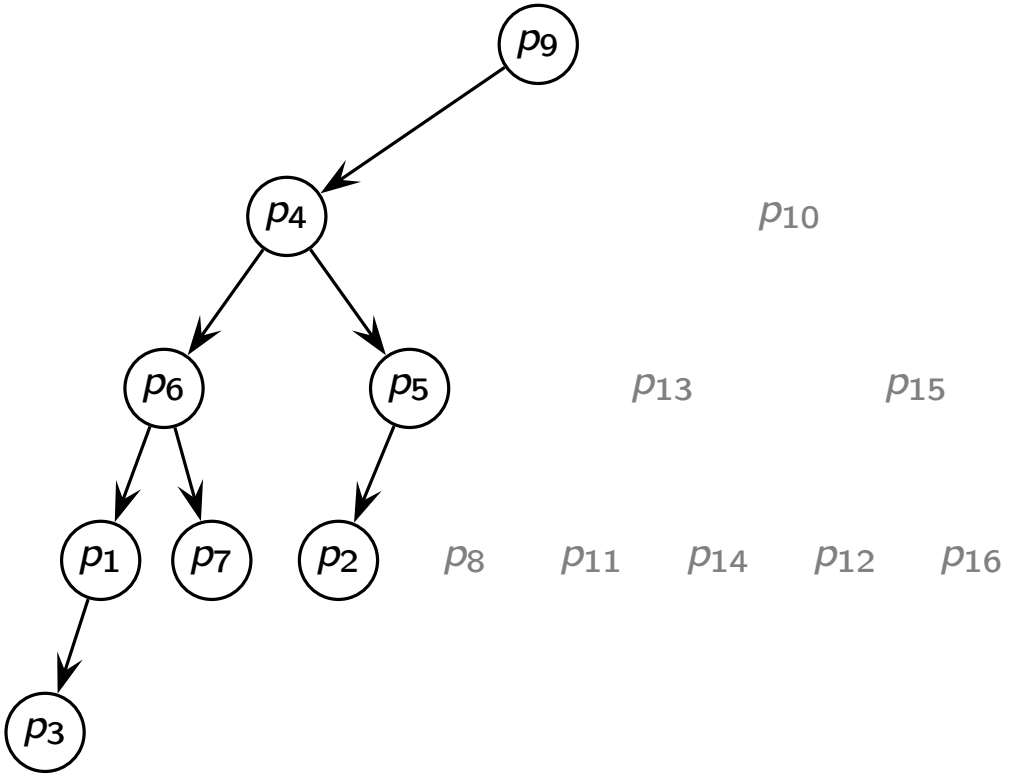
# 2-Dimensional Kd-tree Construction



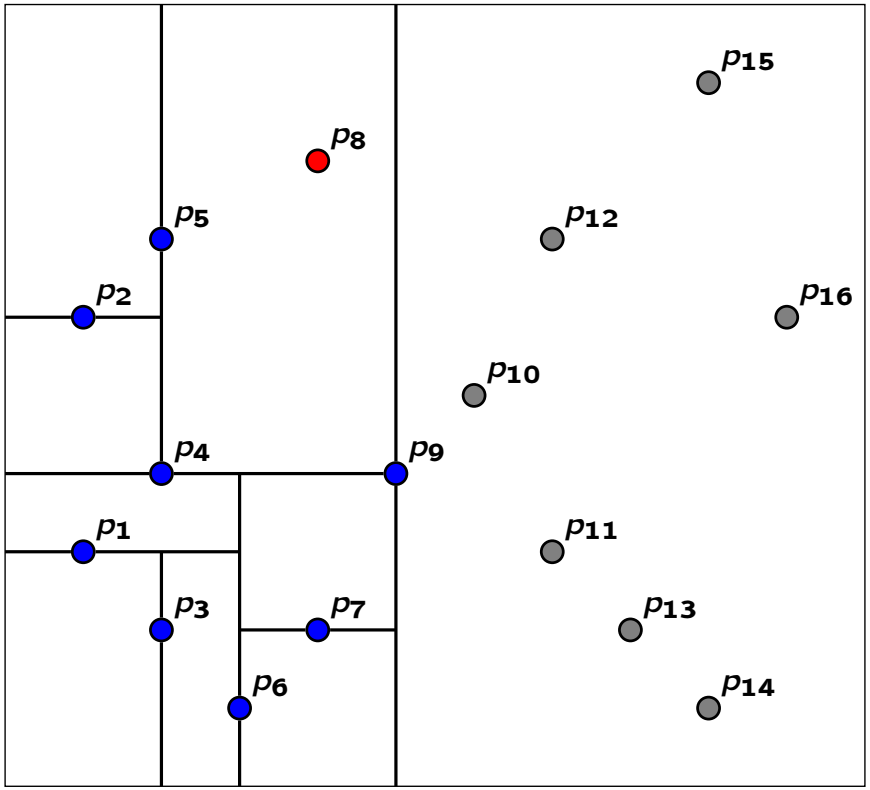



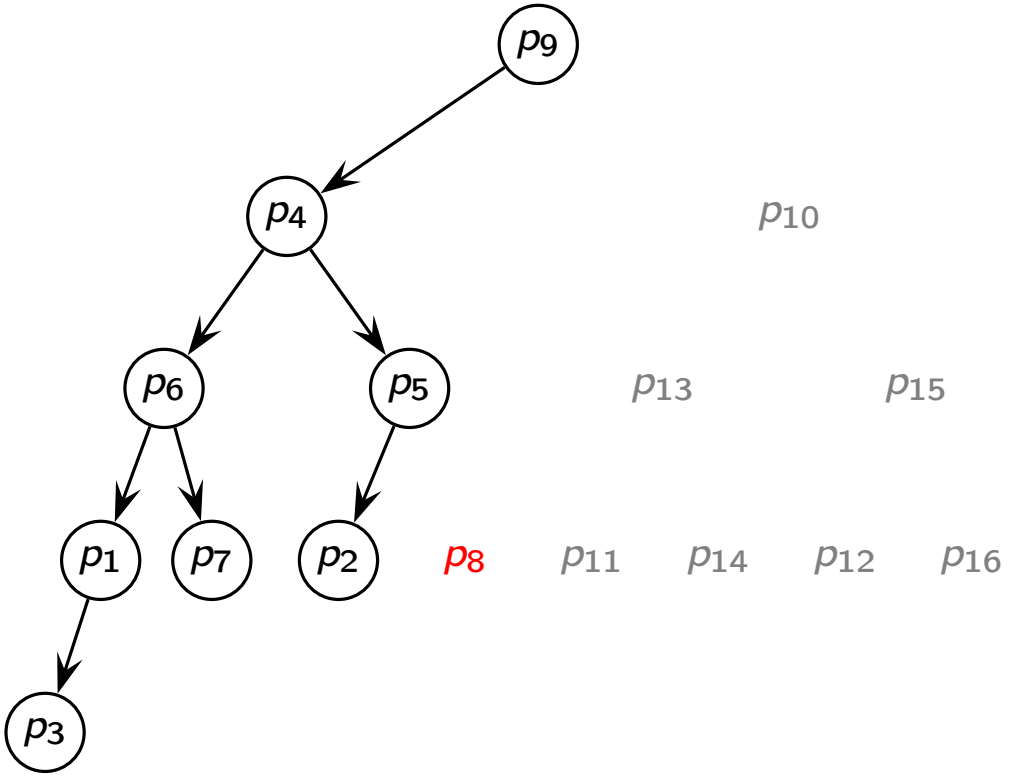
# 2-Dimensional Kd-tree Construction



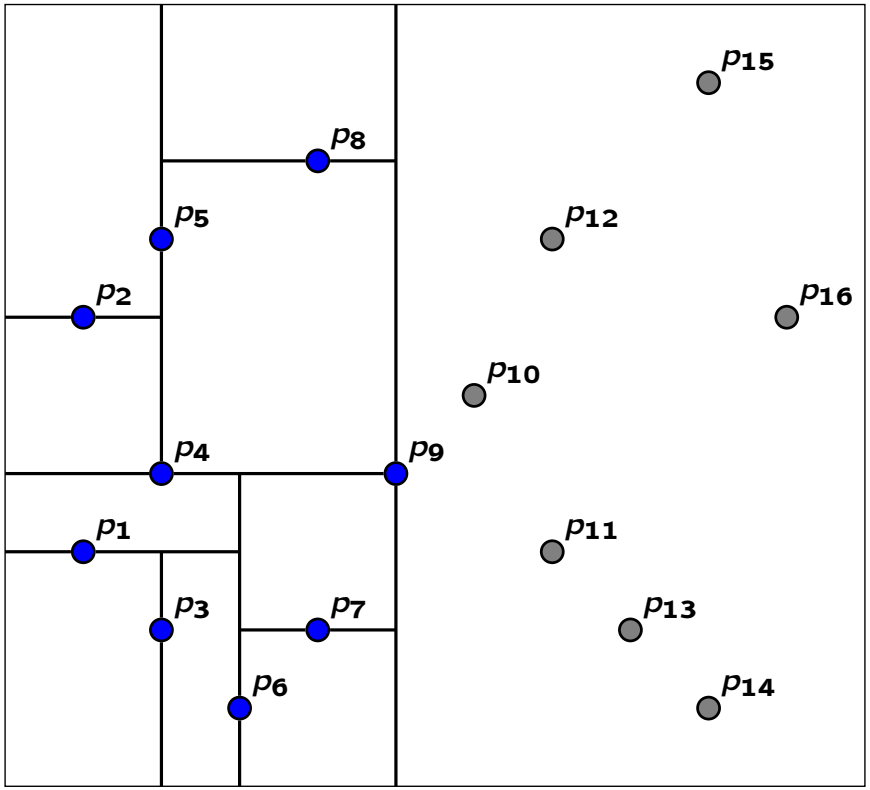



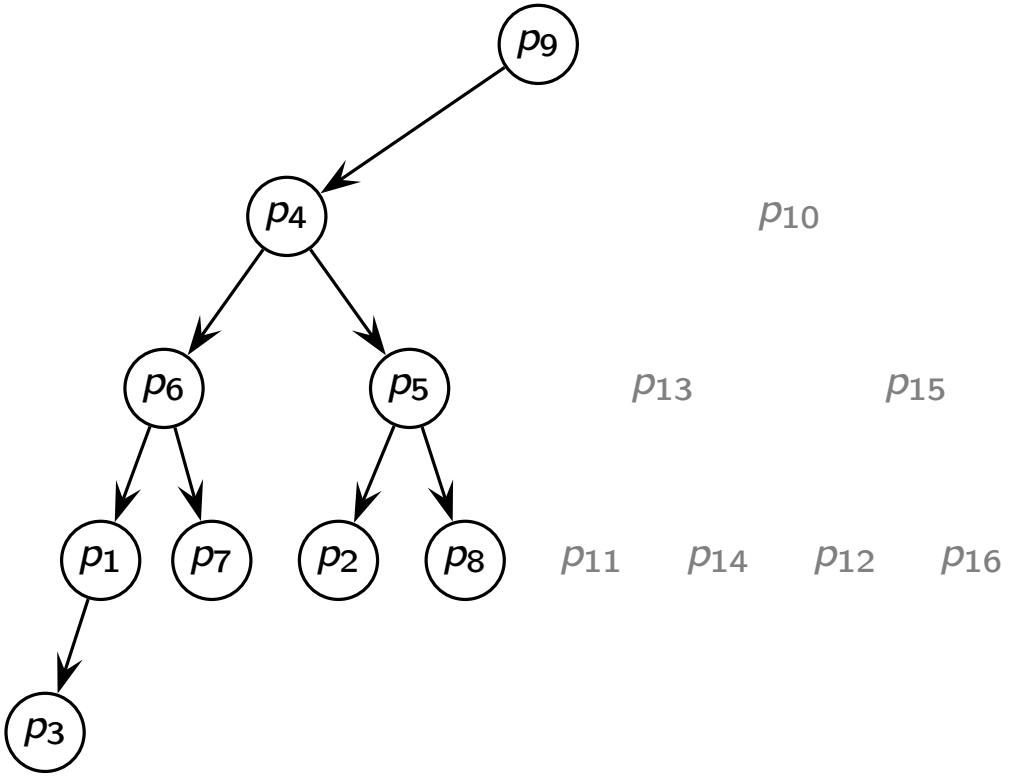
# 2-Dimensional Kd-tree Construction



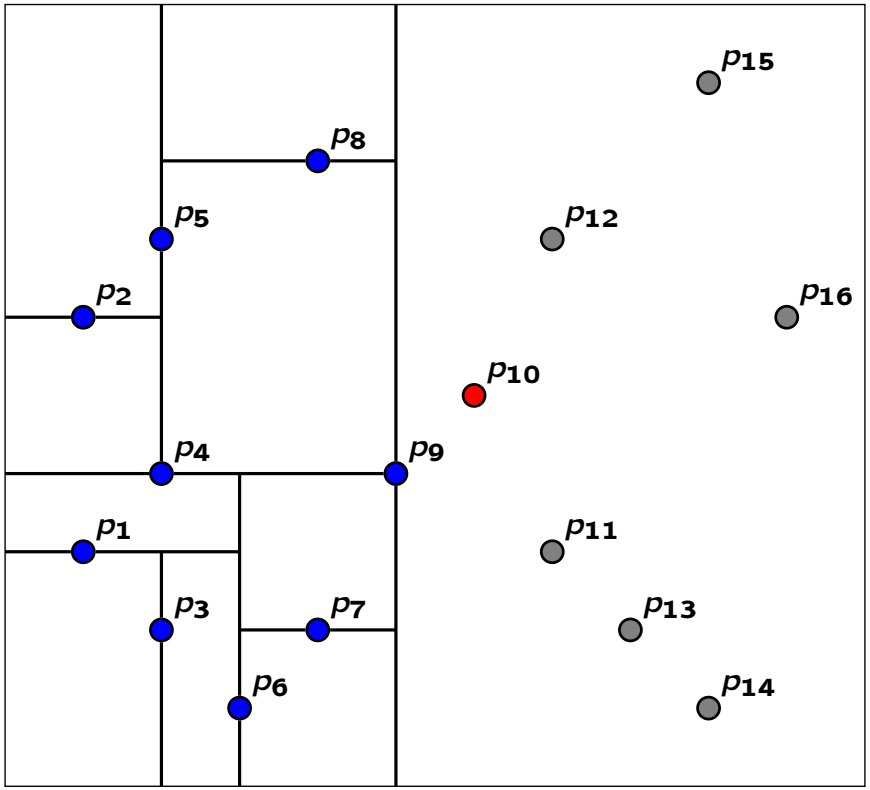



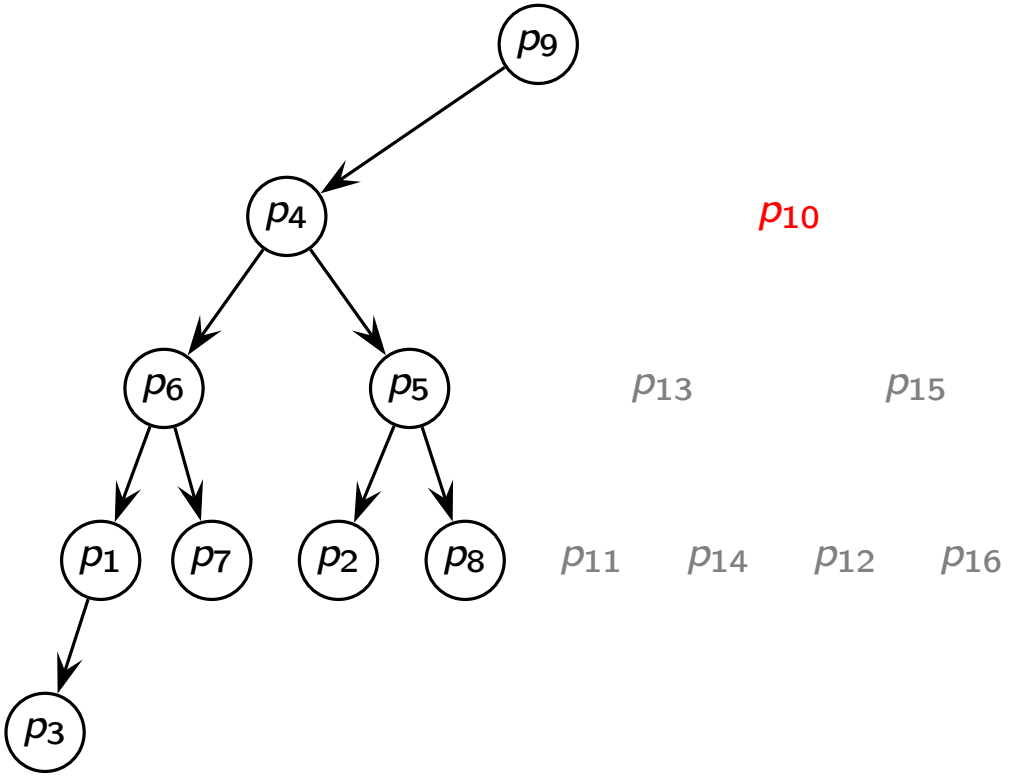
# 2-Dimensional Kd-tree Construction



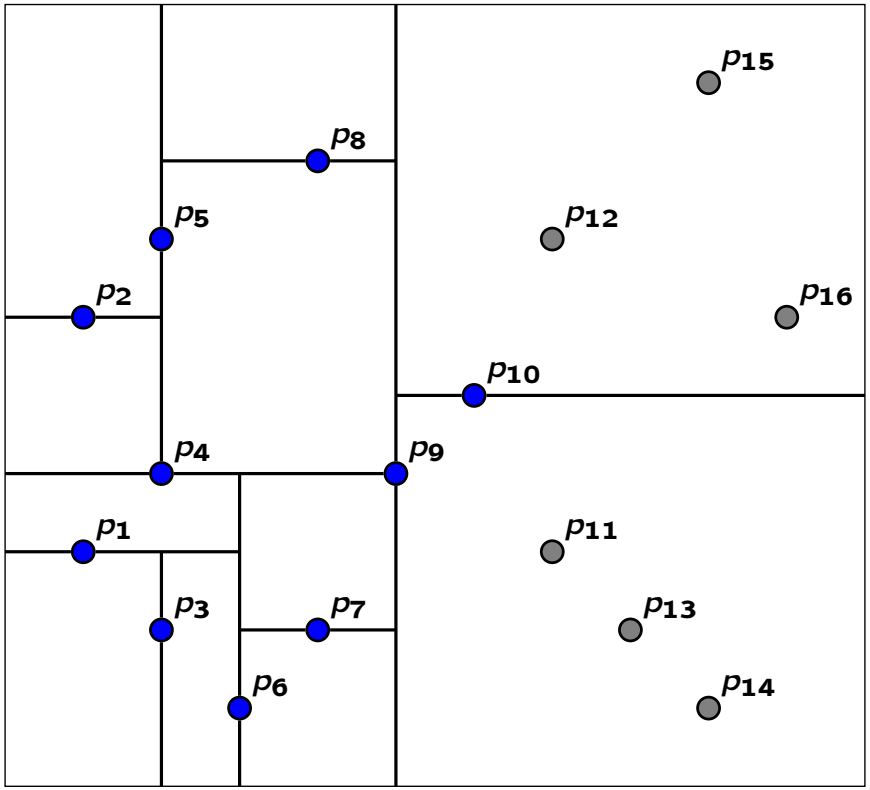



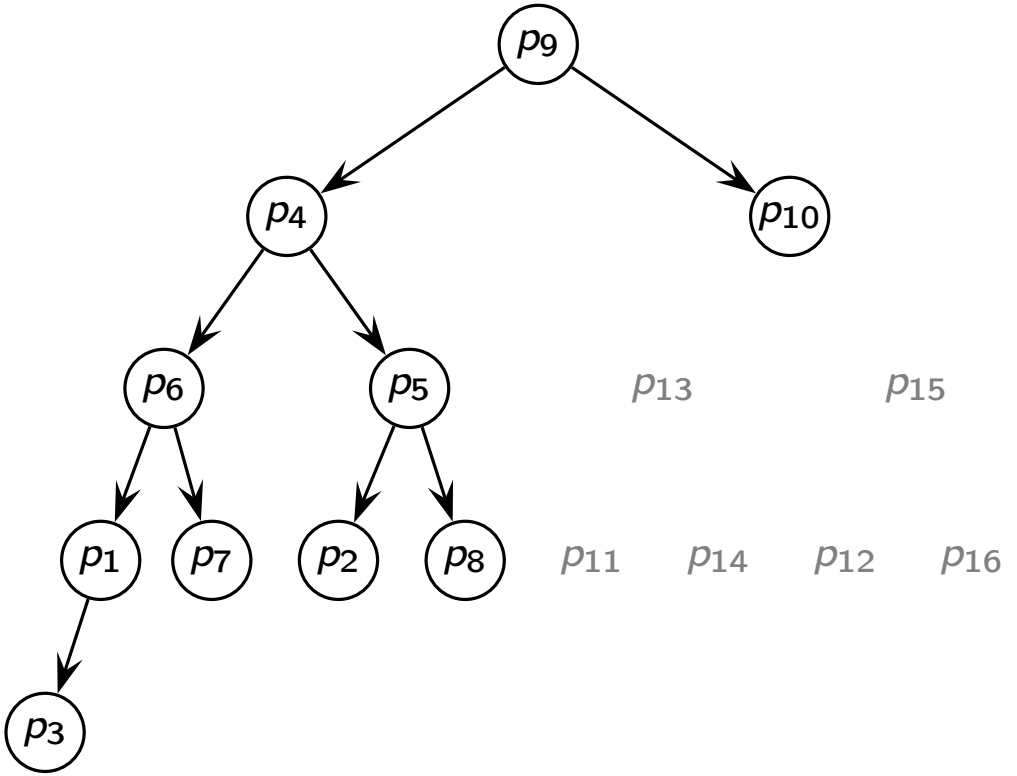
# 2-Dimensional Kd-tree Construction



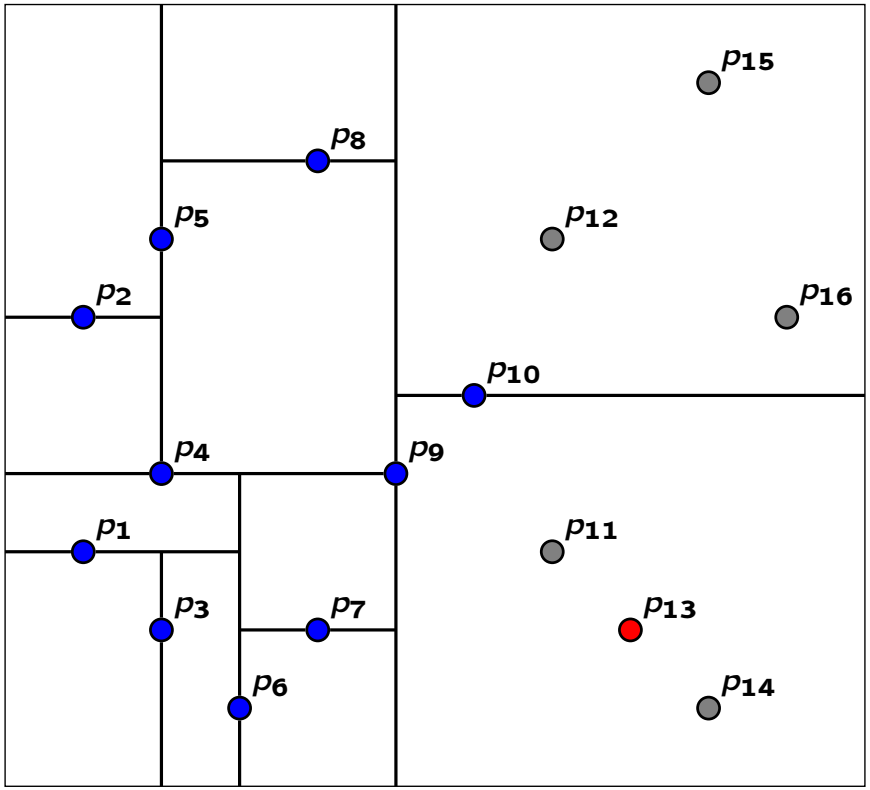



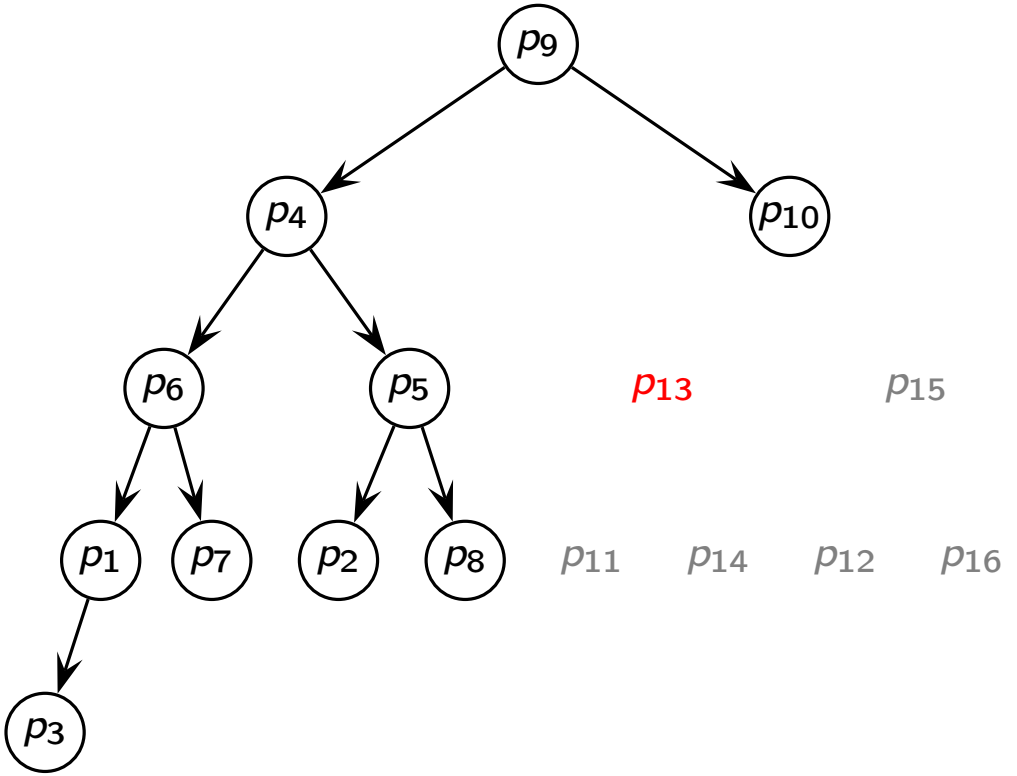
# 2-Dimensional Kd-tree Construction



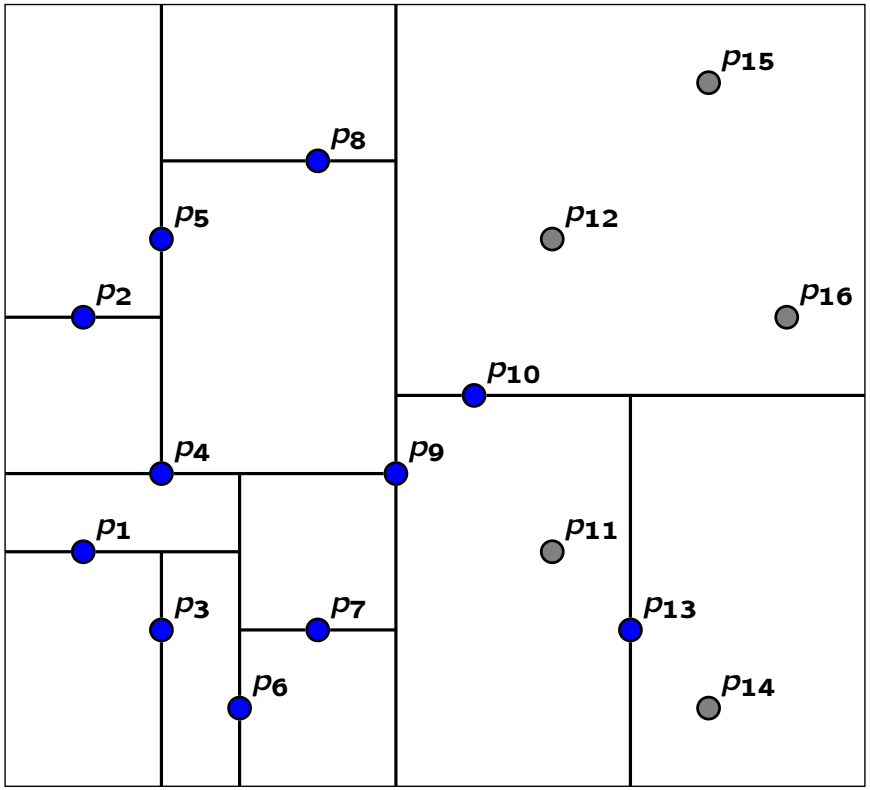



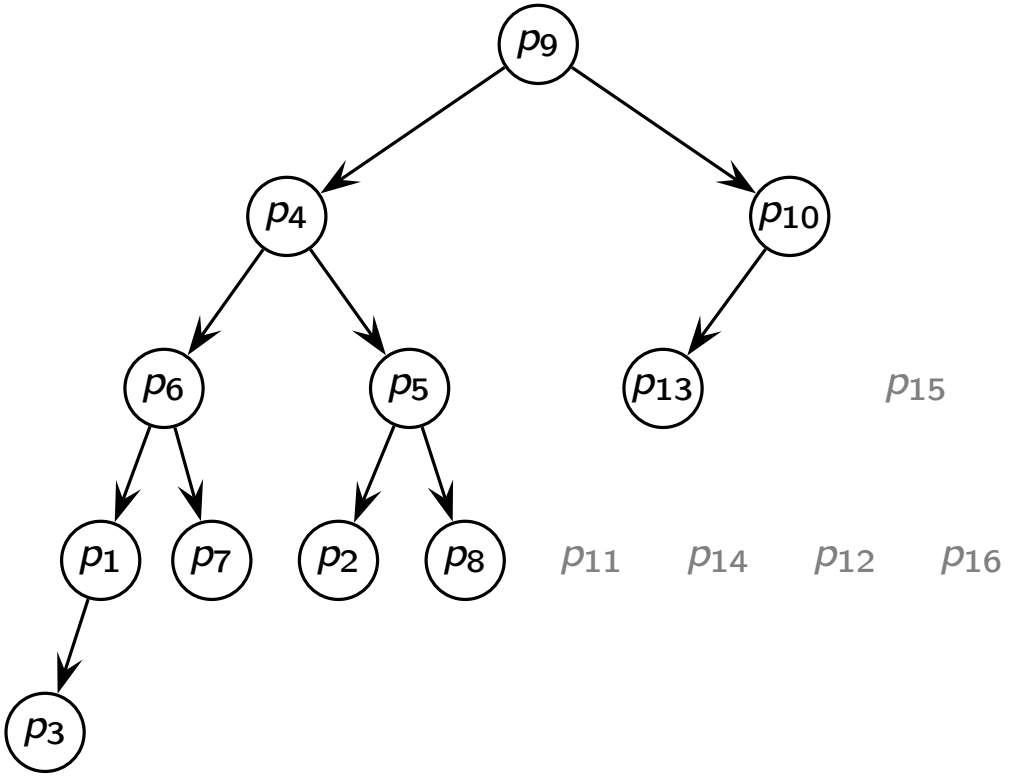

# 2-Dimensional Kd-tree Construction



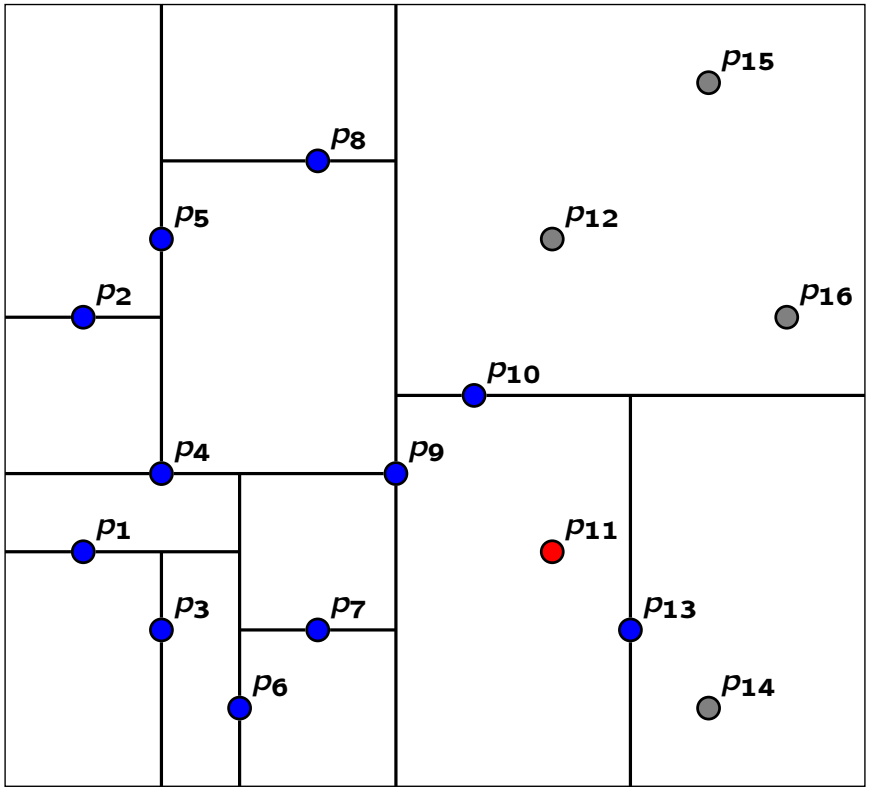



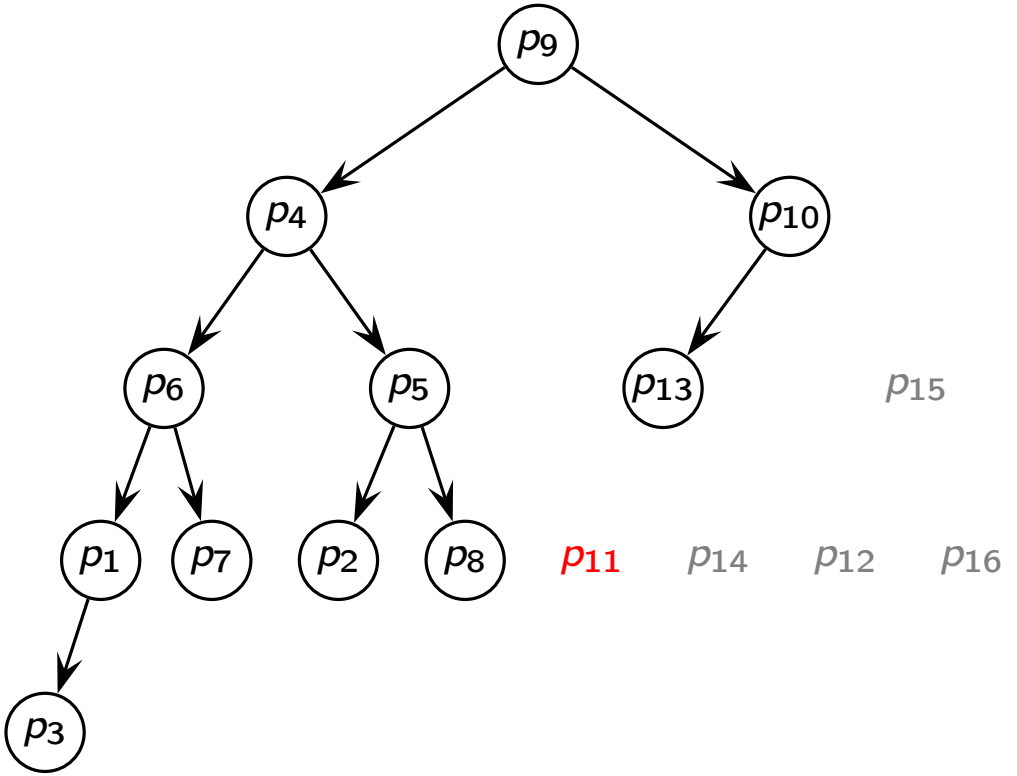

# 2-Dimensional Kd-tree Construction



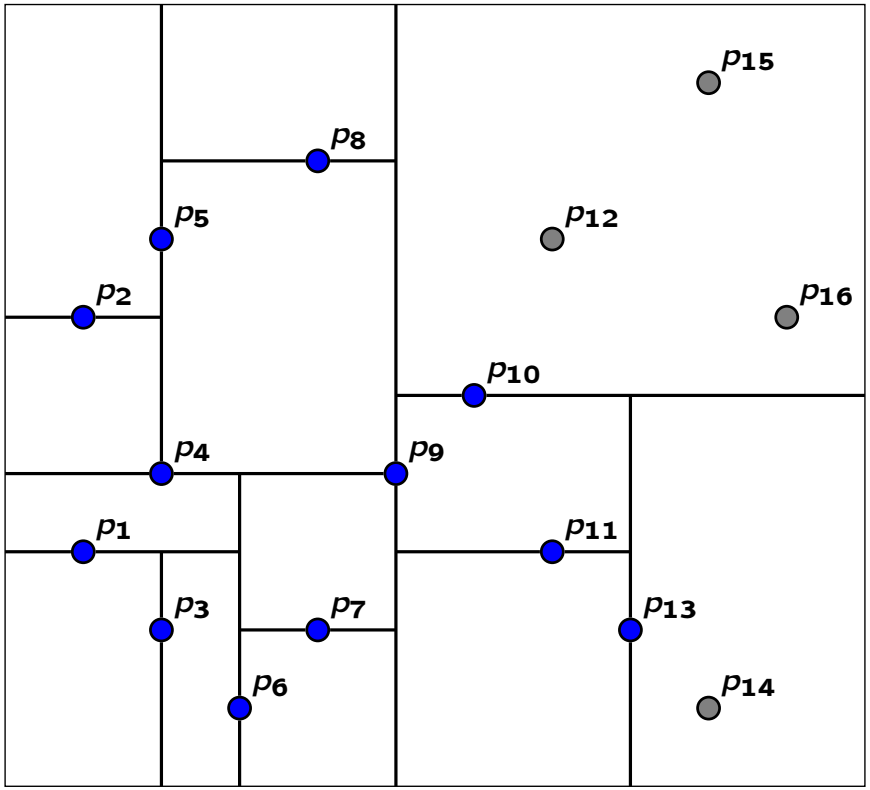



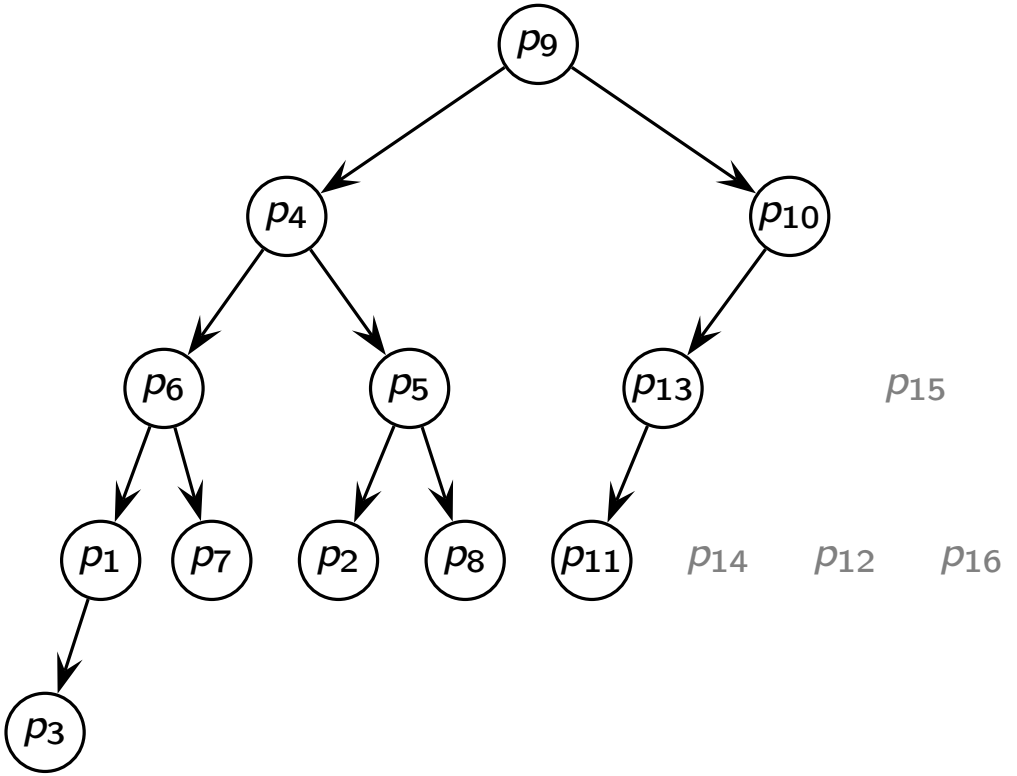

# 2-Dimensional Kd-tree Construction



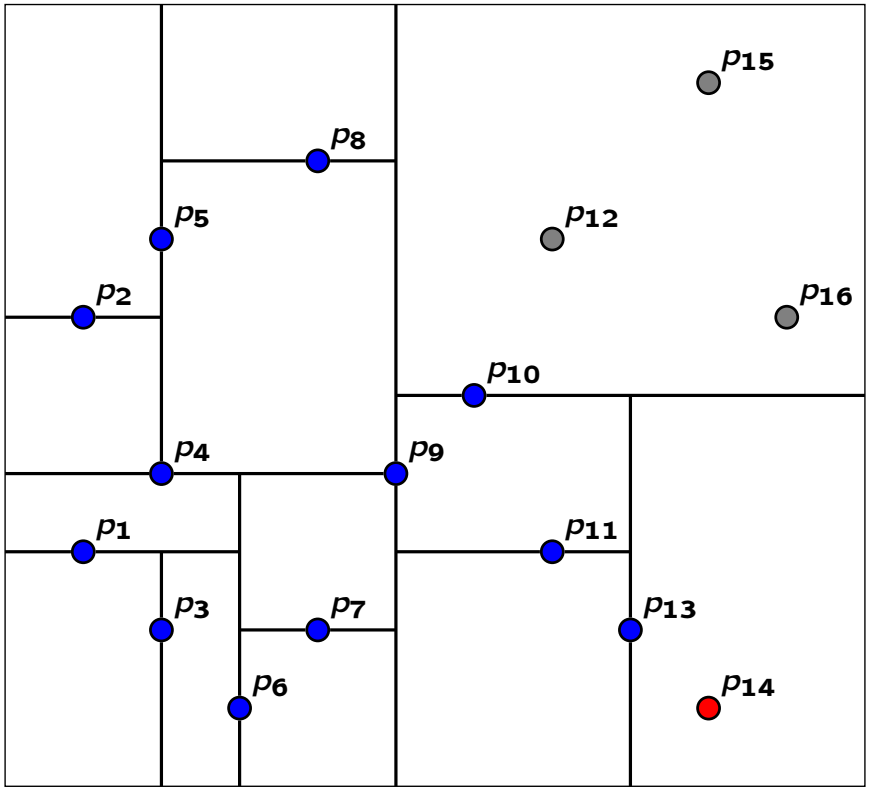



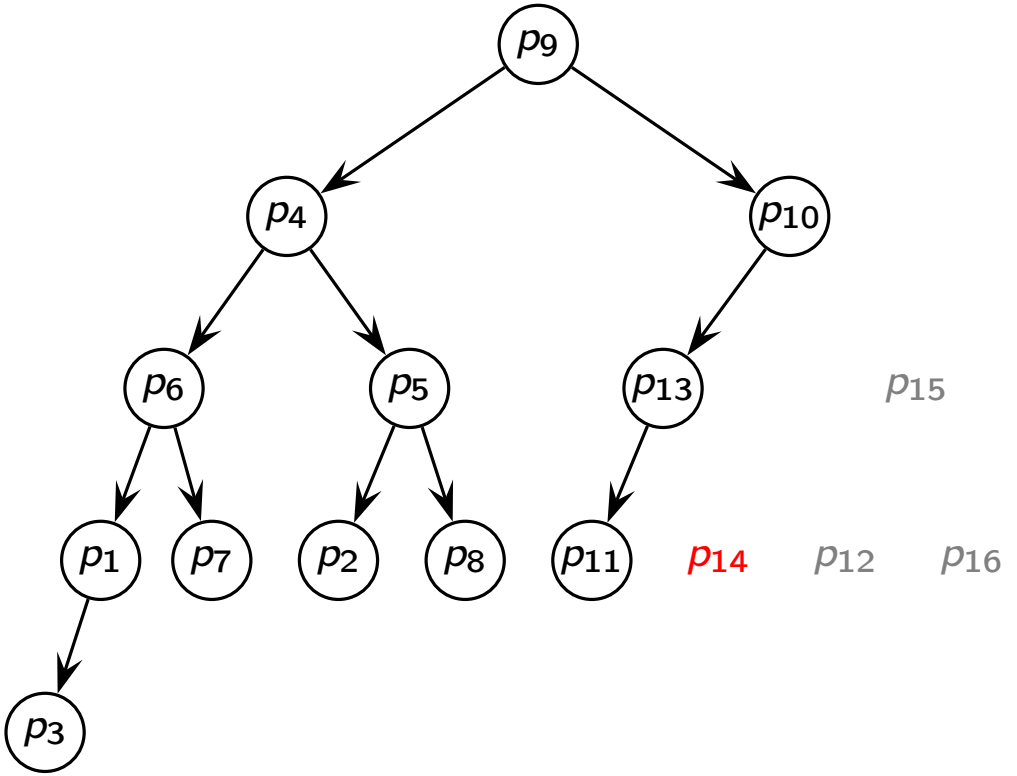

# 2-Dimensional Kd-tree Construction



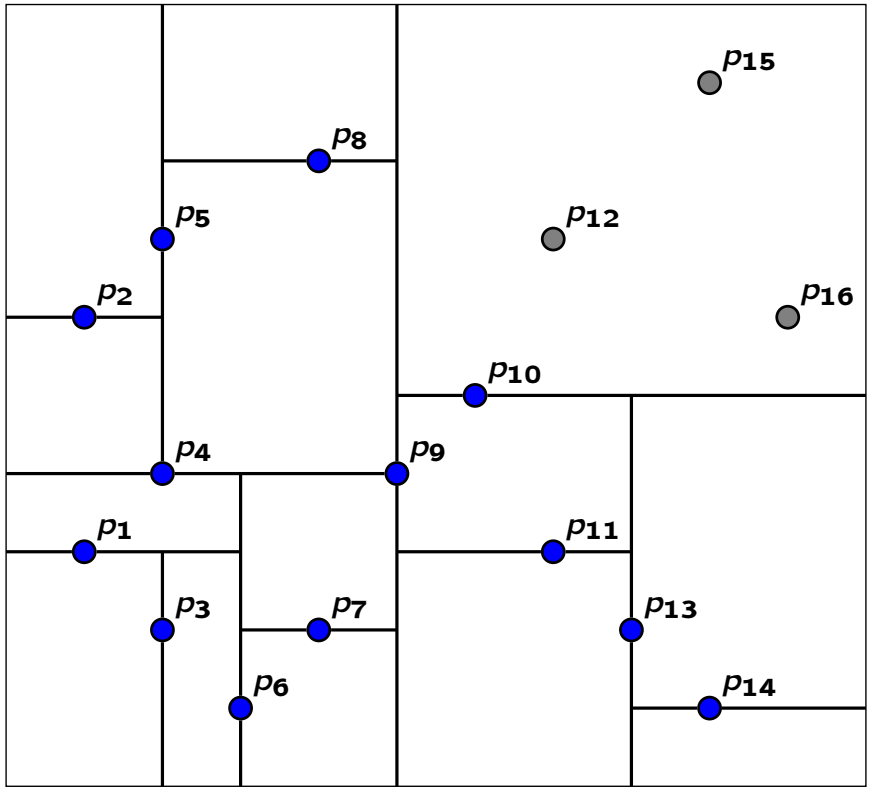



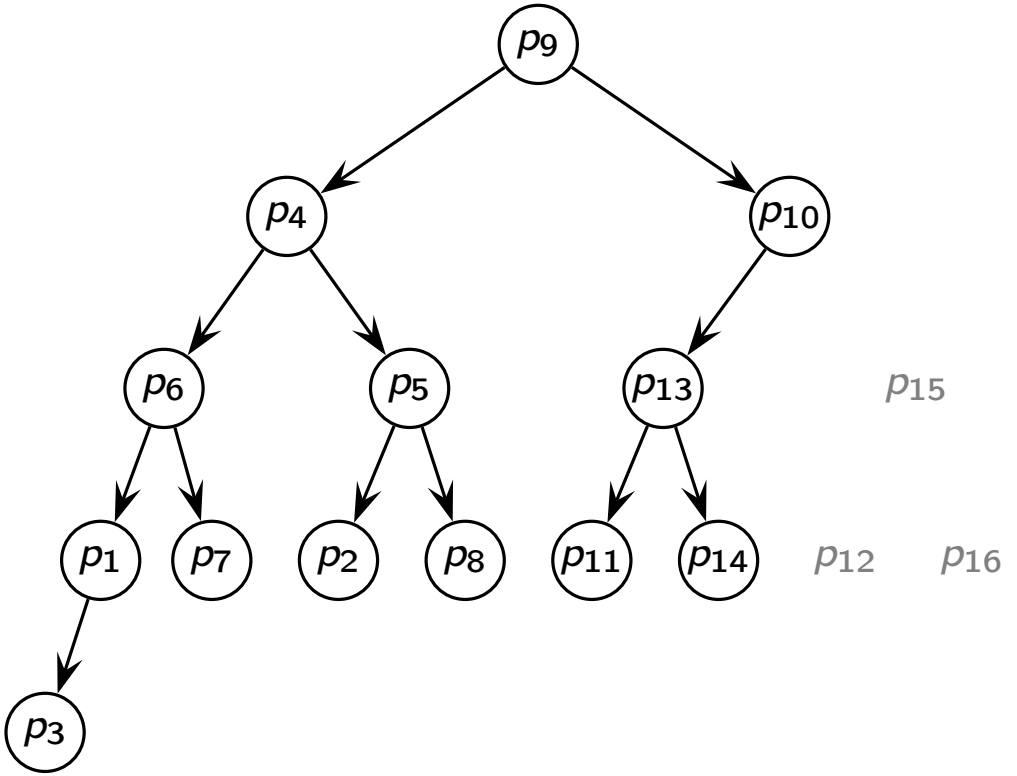

# 2-Dimensional Kd-tree Construction



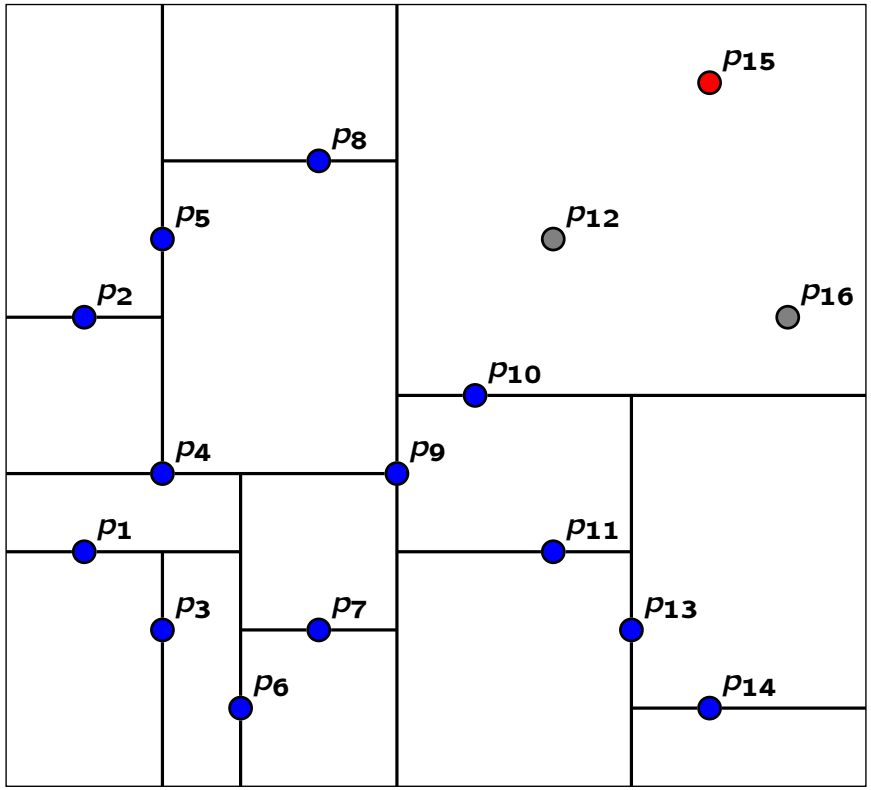



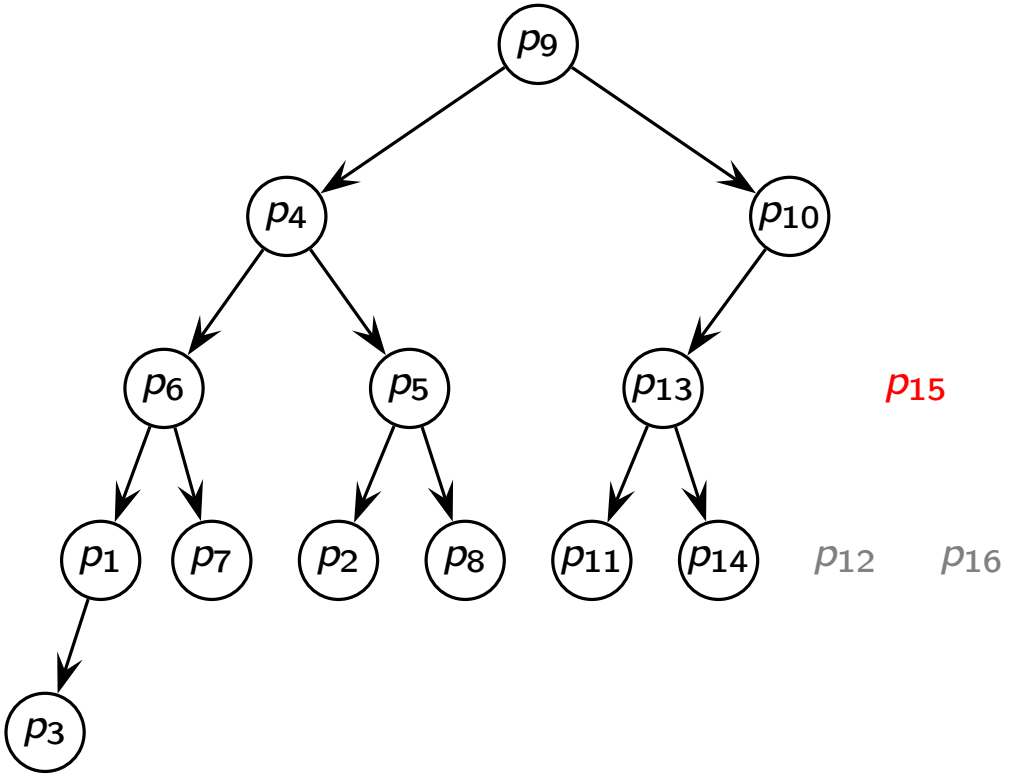

# 2-Dimensional Kd-tree Construction



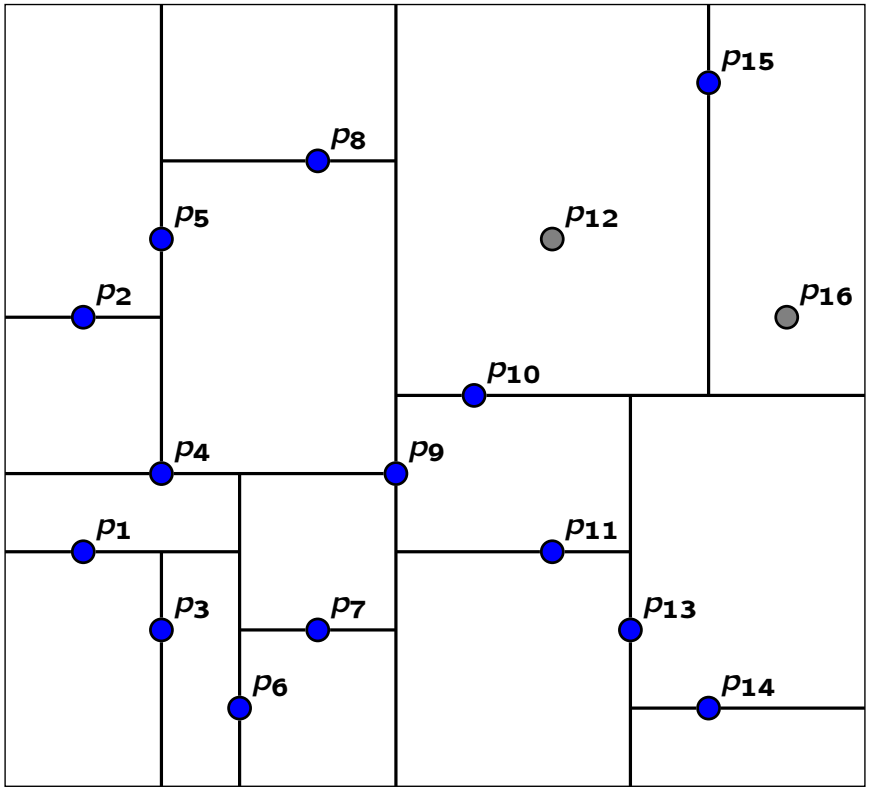



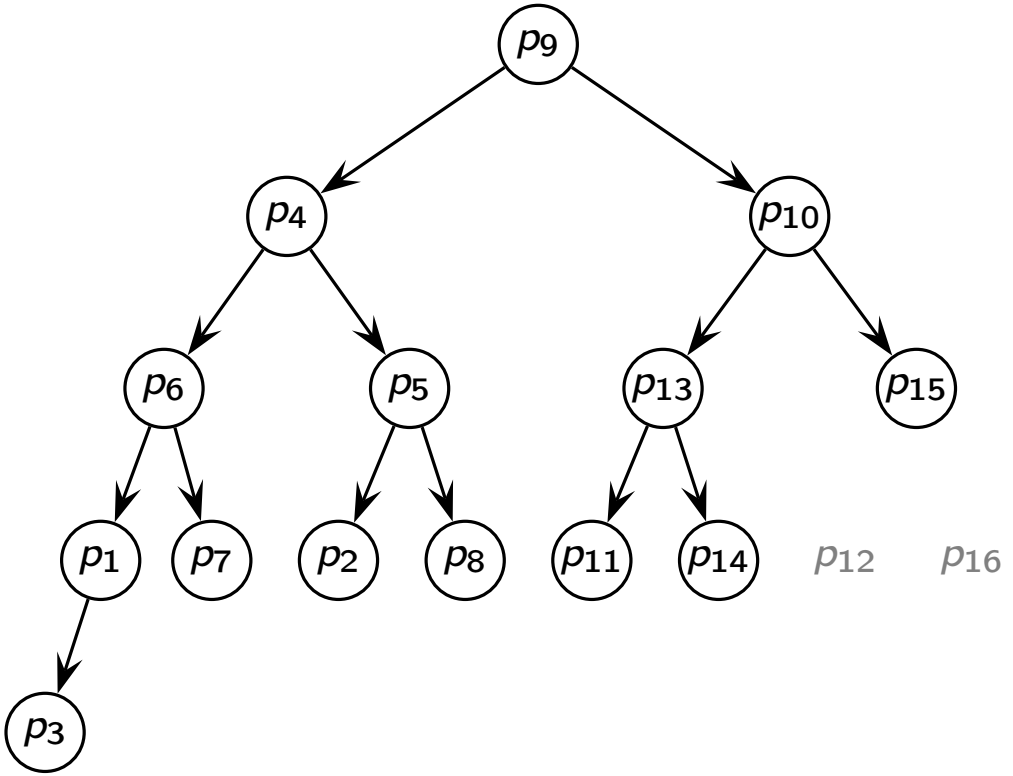

# 2-Dimensional Kd-tree Construction



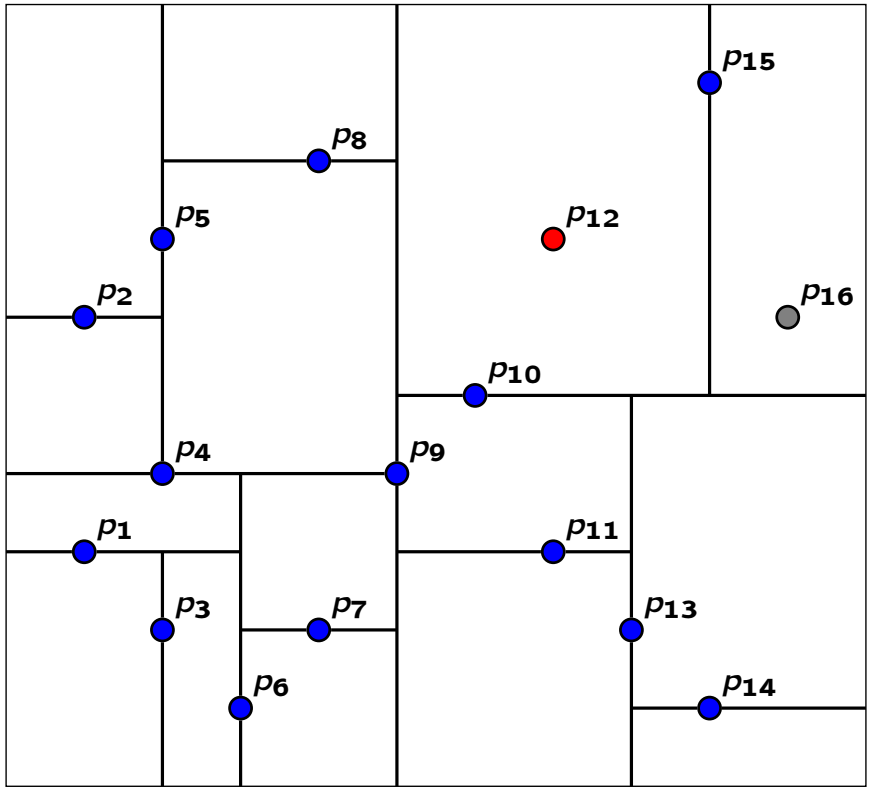



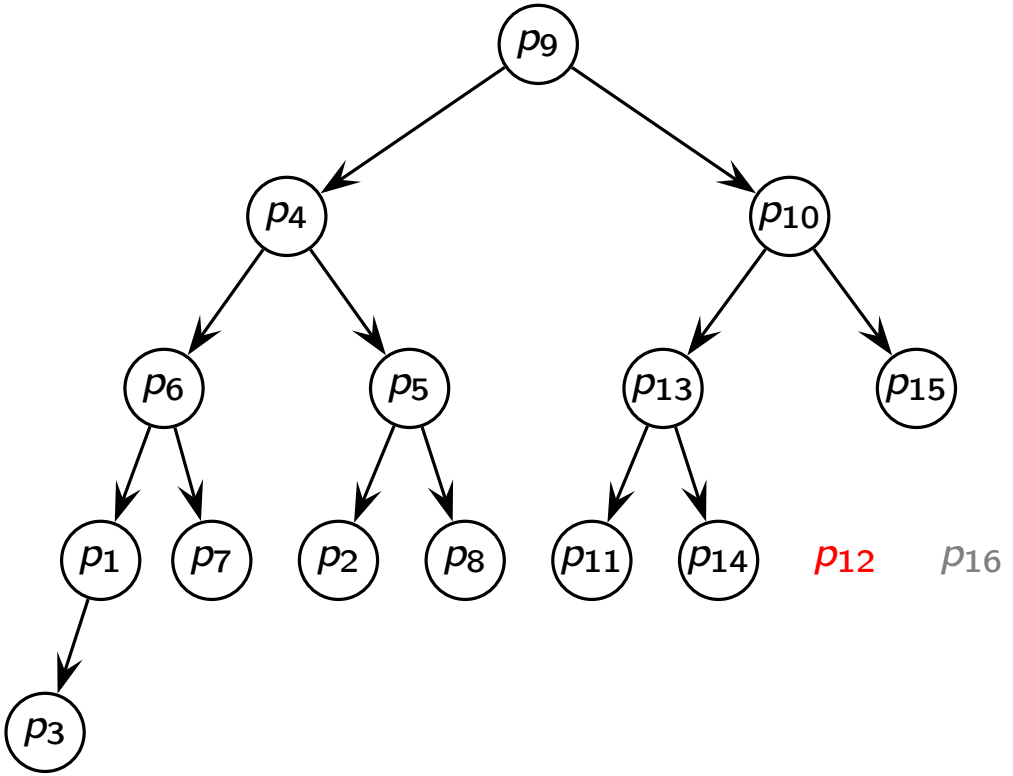

# 2-Dimensional Kd-tree Construction



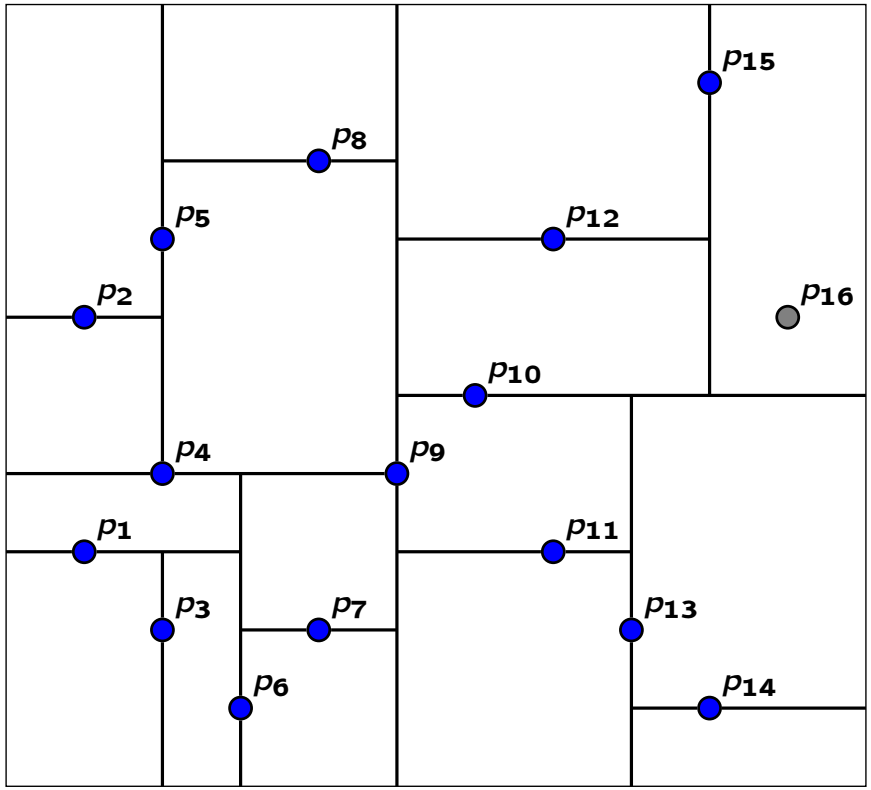



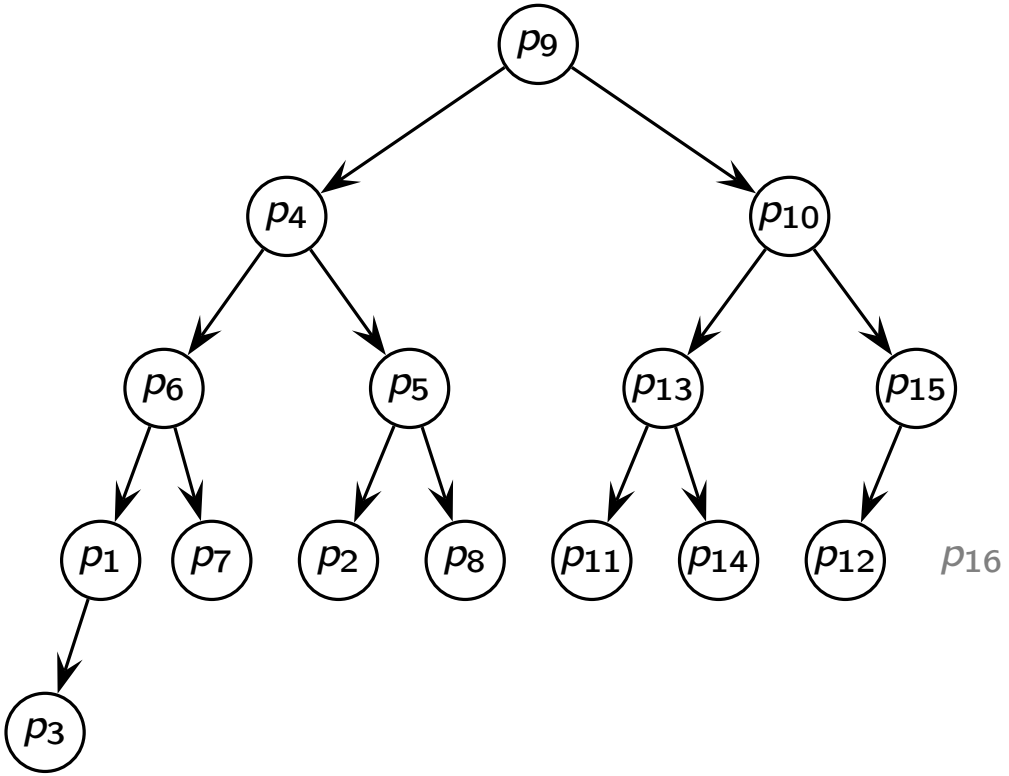

# 2-Dimensional Kd-tree Construction



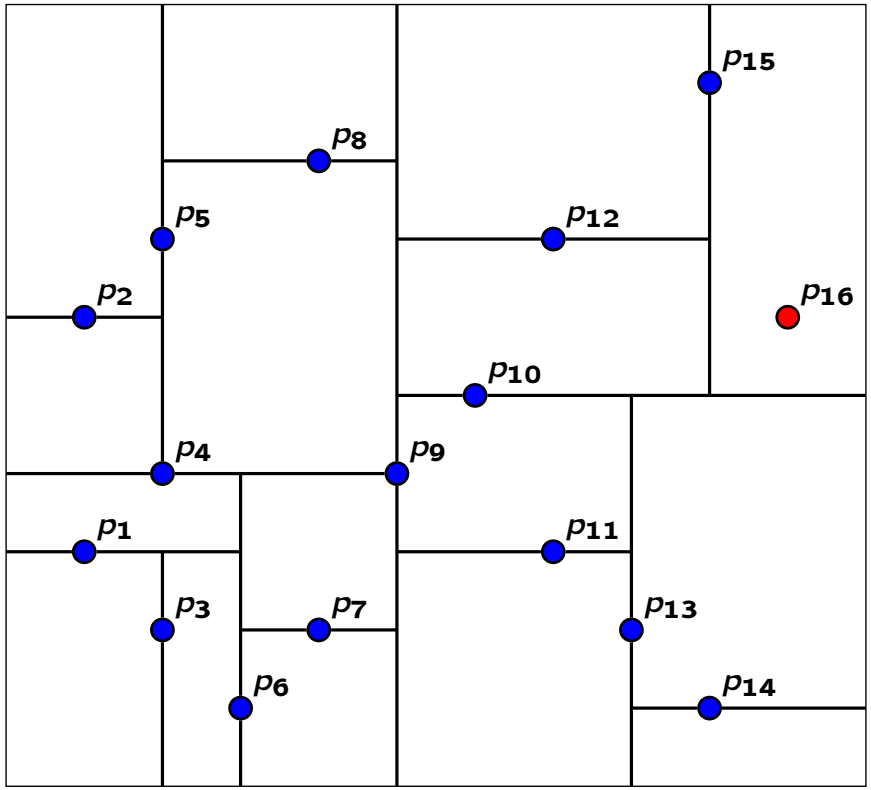



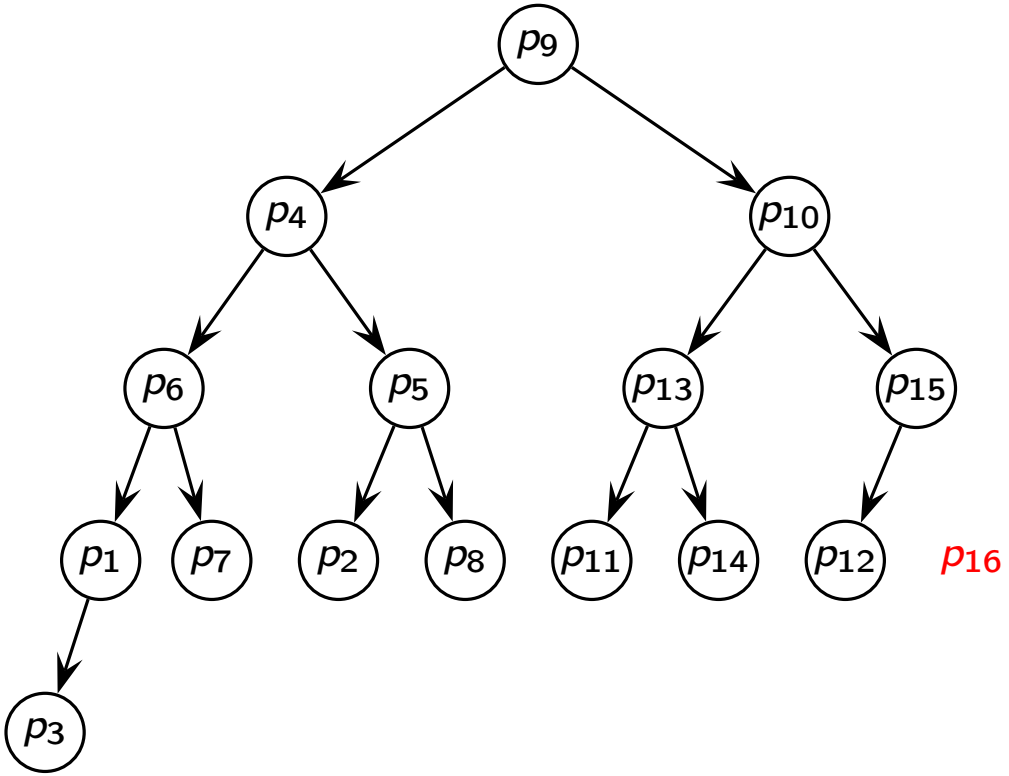

# 2-Dimensional Kd-tree Construction



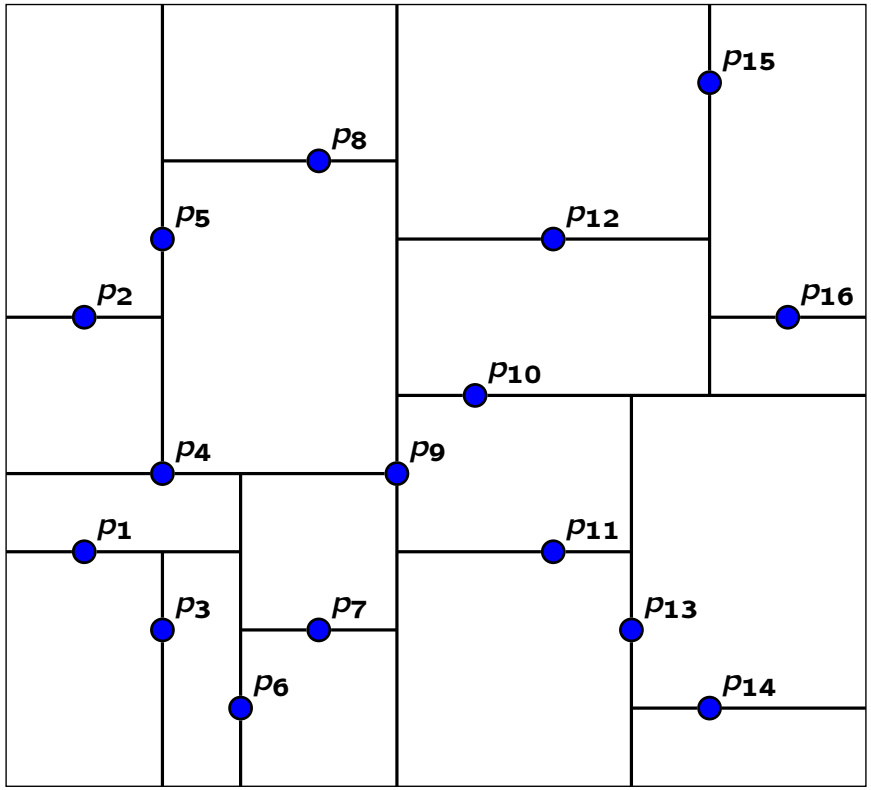



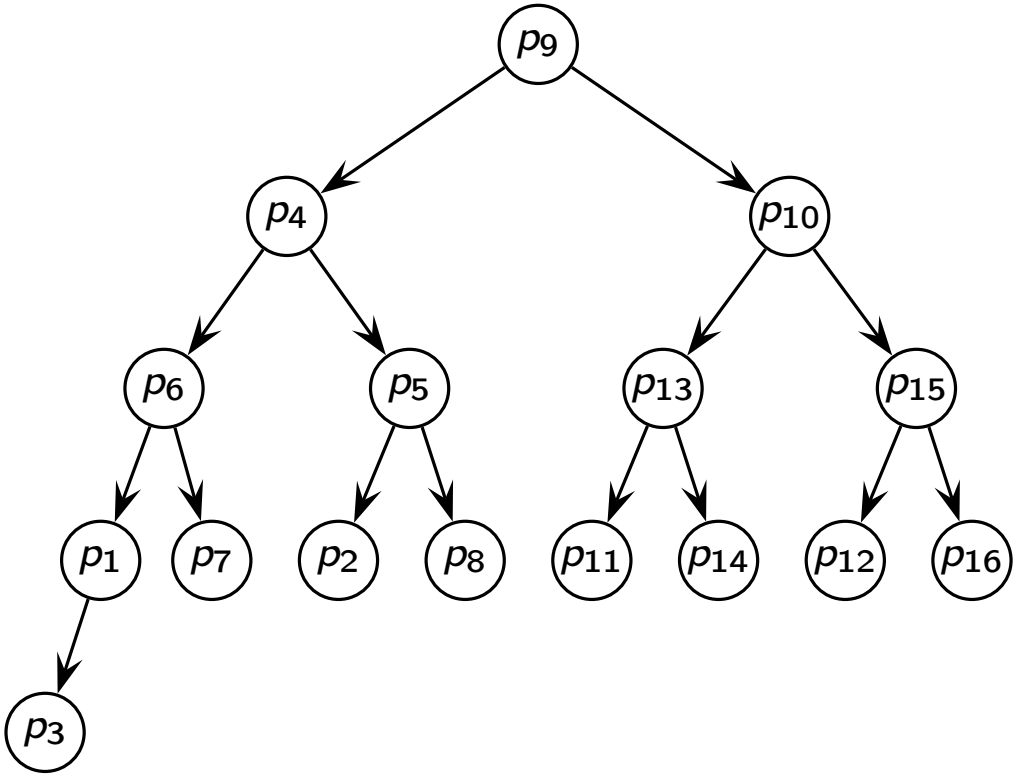

# 2-Dimensional Kd-tree Construction



# 2-Dimensional Kd-tree Construction




# d-Dimensional Kd-tree Preprocessing Complexity

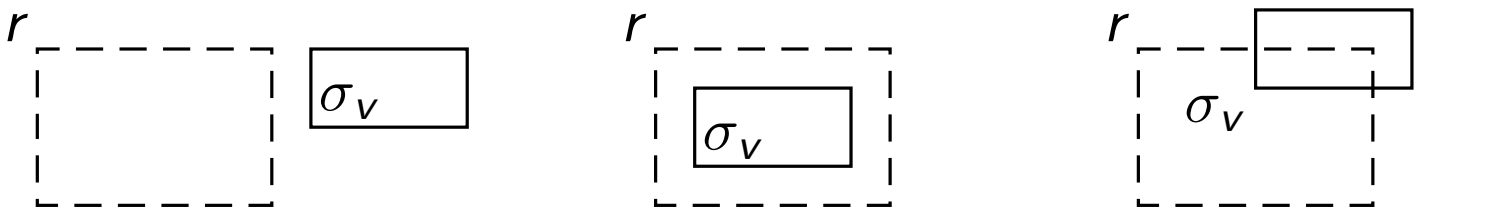
- $O(n)$ — the time required to find the median
  - The selection algorithm [CT+01]
  - Maintain $d$ lists of pointers to the points such that the $i$th list is ordered by the $i$th coordinate
- The two recursive calls are on point sets half the size of the previous level
- $T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$
- Each node contains exactly one point $\Rightarrow S(n) = O(n)$

# d-Dimensional Kd-tree Query

Query a $d$-dimensional Kd-tree, $r = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d]$

$\text{query}(r, v, i, d)$

1. **if** $v = \text{null}$, **return** $\emptyset$
2. **if** $p(v) \in r$, $O \leftarrow p(v)$
3. **if** $a_i < x_i(p(v))$, $O \leftarrow \text{query}(r, \text{left}(v), (i+1) \mod d, d)$
4. **if** $b_i > x_i(p(v))$, $O \leftarrow \text{query}(r, \text{right}(v), (i+1) \mod d, d)$
5. **return** $O$

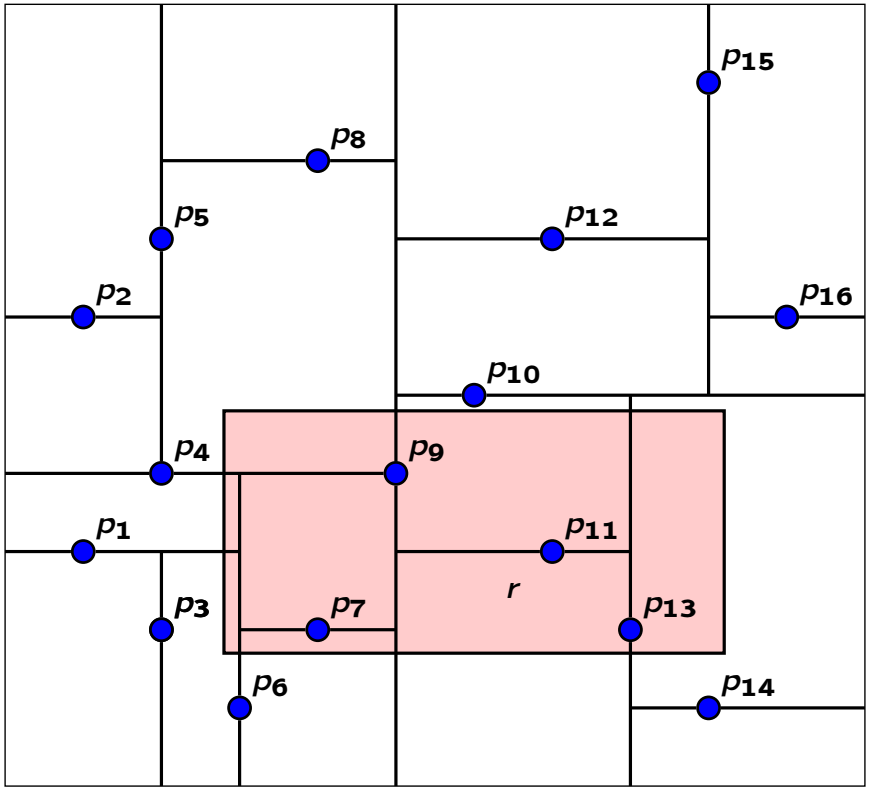


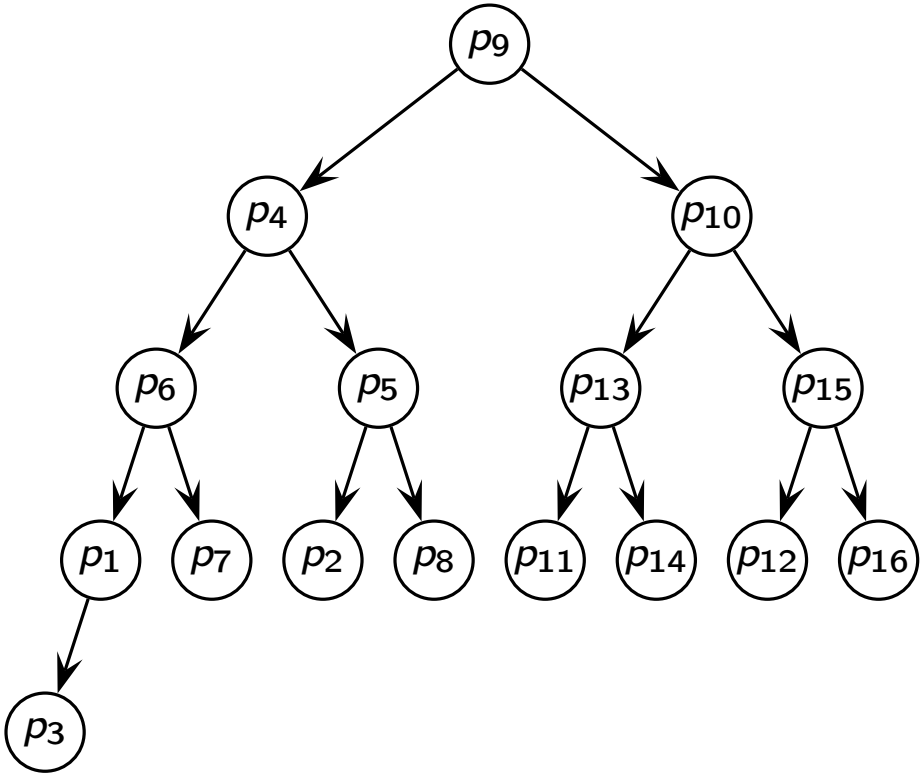

Case (a) — White  Case (a) — Black  Case (c) — Gray

$\sigma_v$ — the rectangle representing the current node

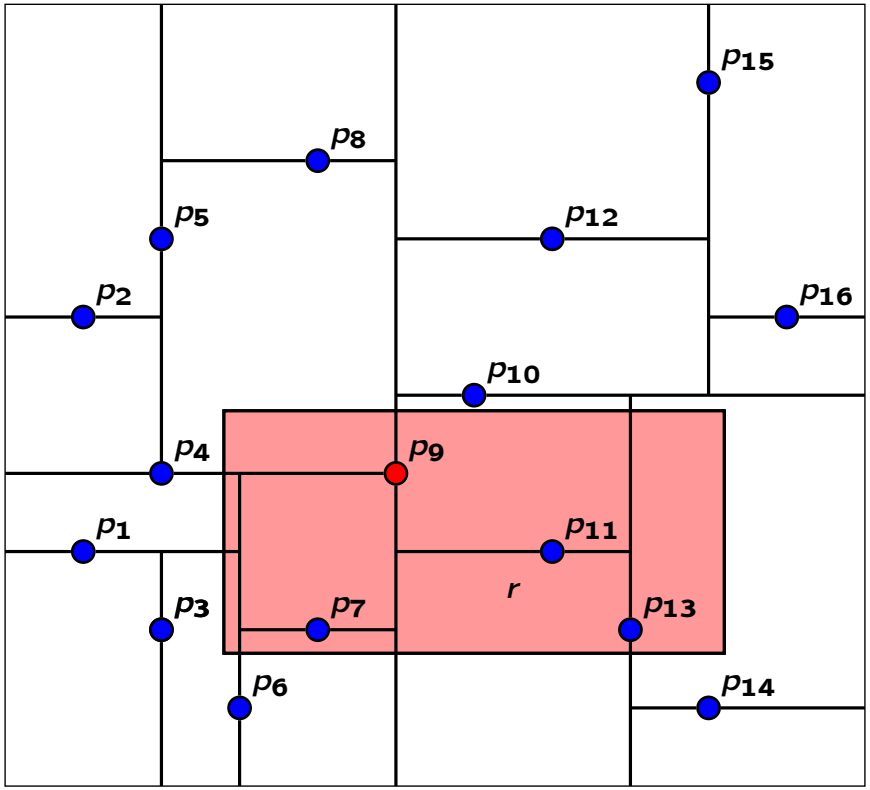
- If a node is white (black), all of its children are white (black)
- If a node is gray, any of its children can be of any color

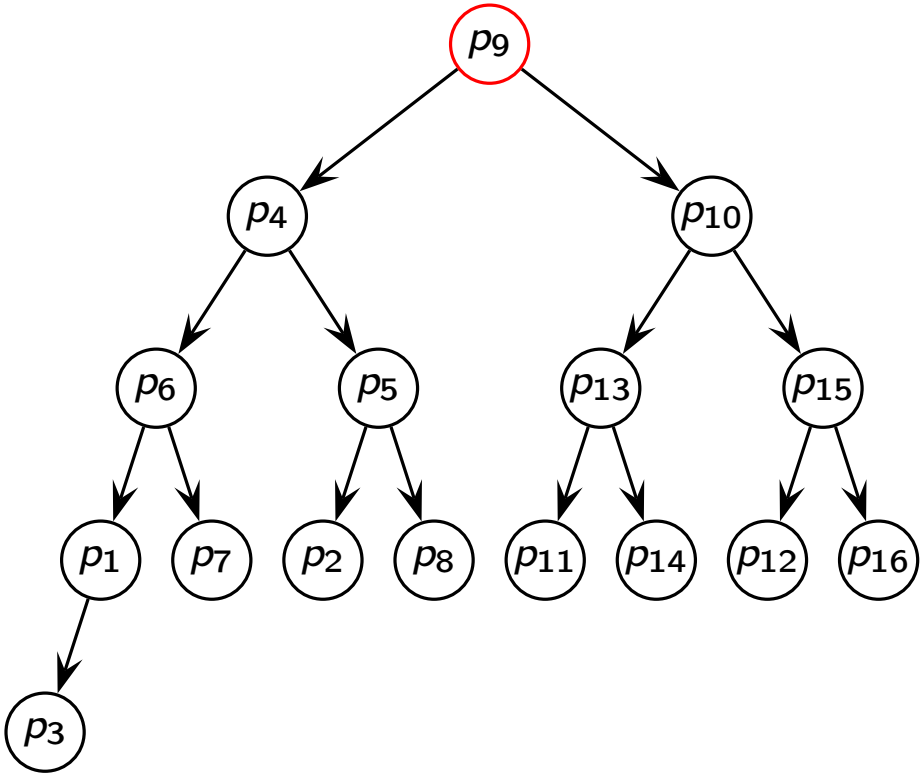
# 2-Dimensional Kd-tree Query Example



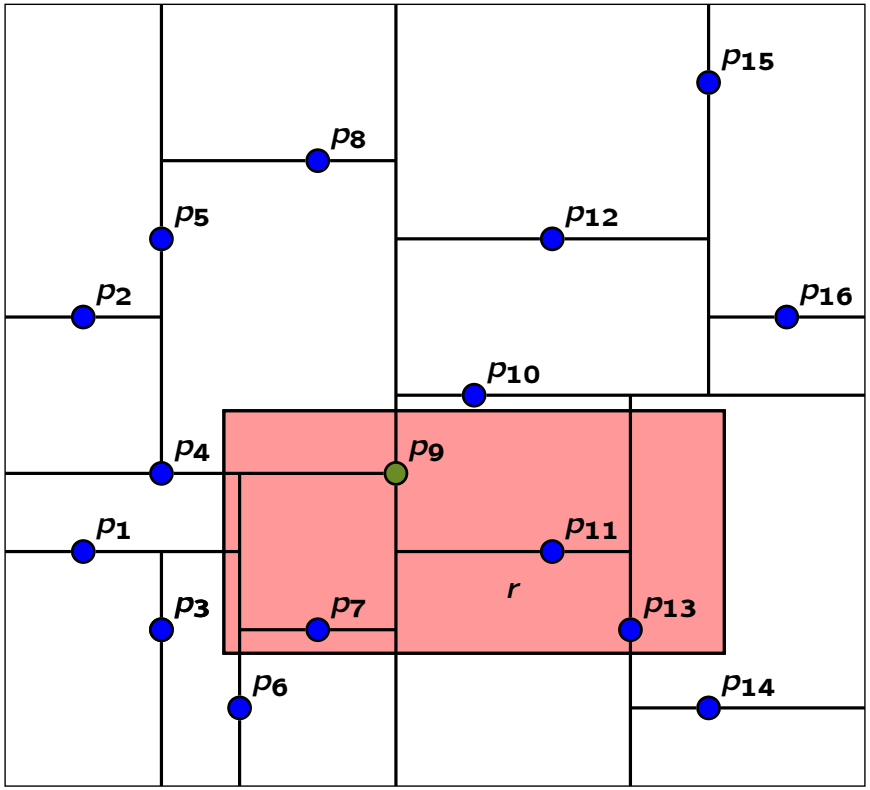



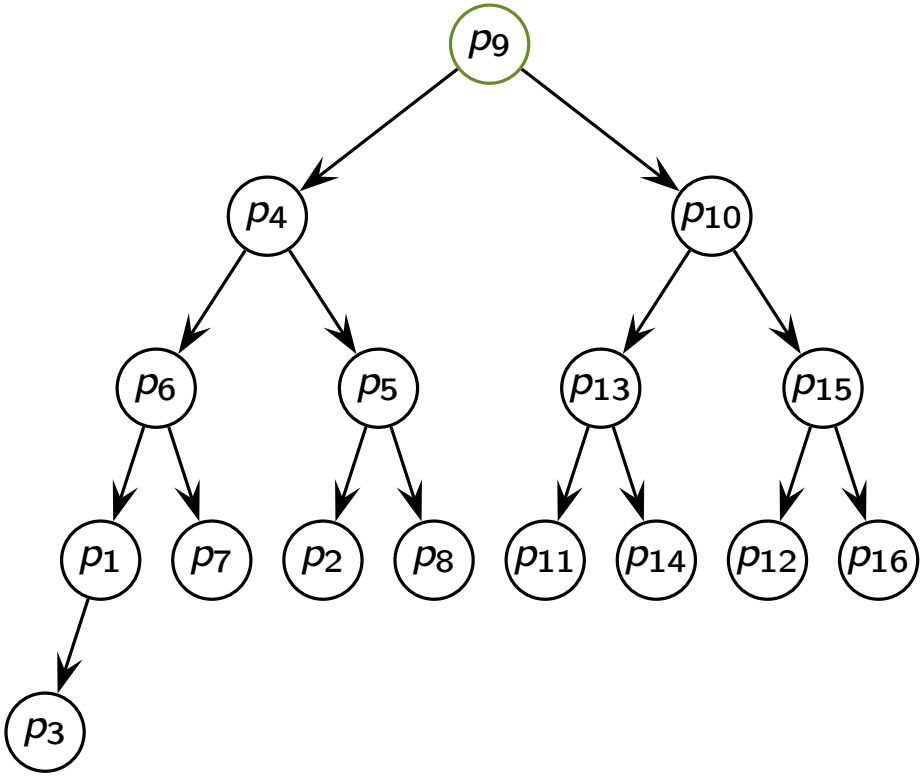
# 2-Dimensional Kd-tree Query Example



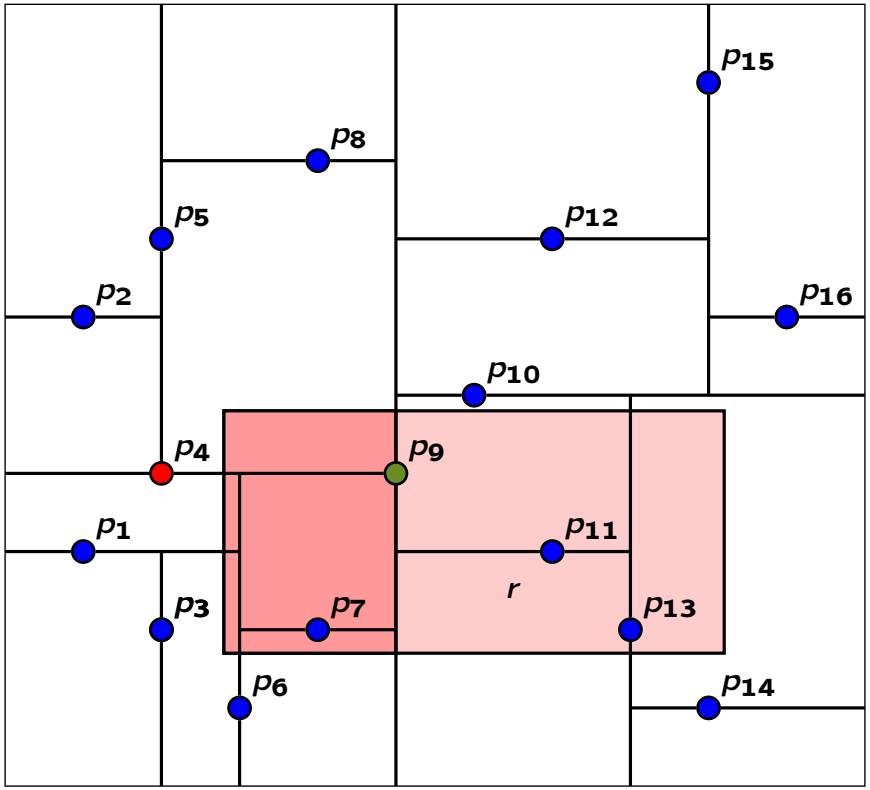



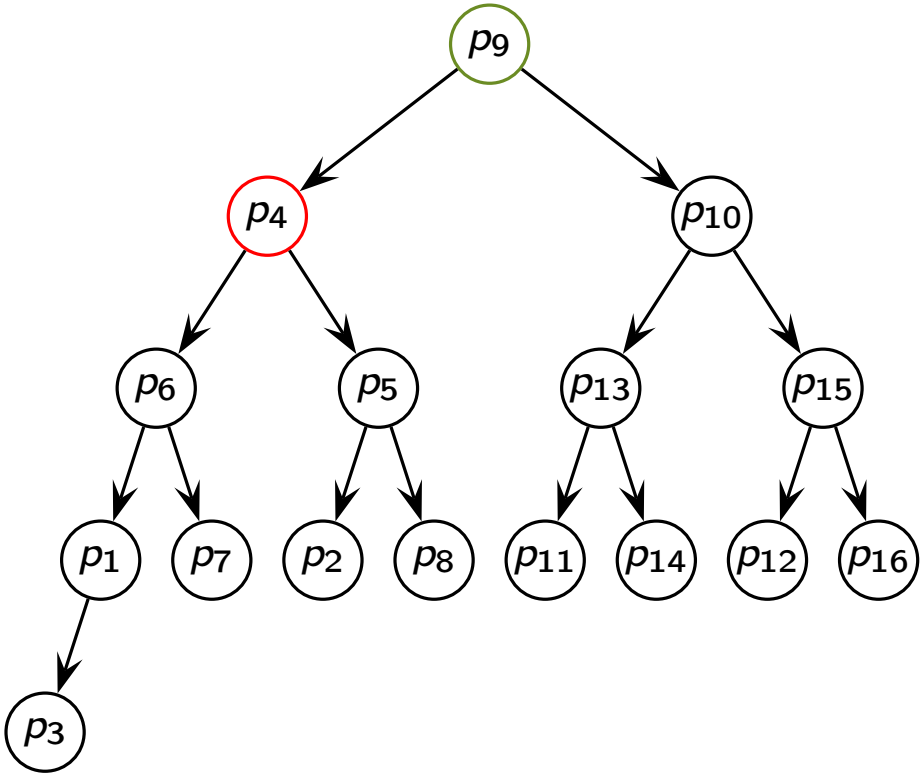
# 2-Dimensional Kd-tree Query Example



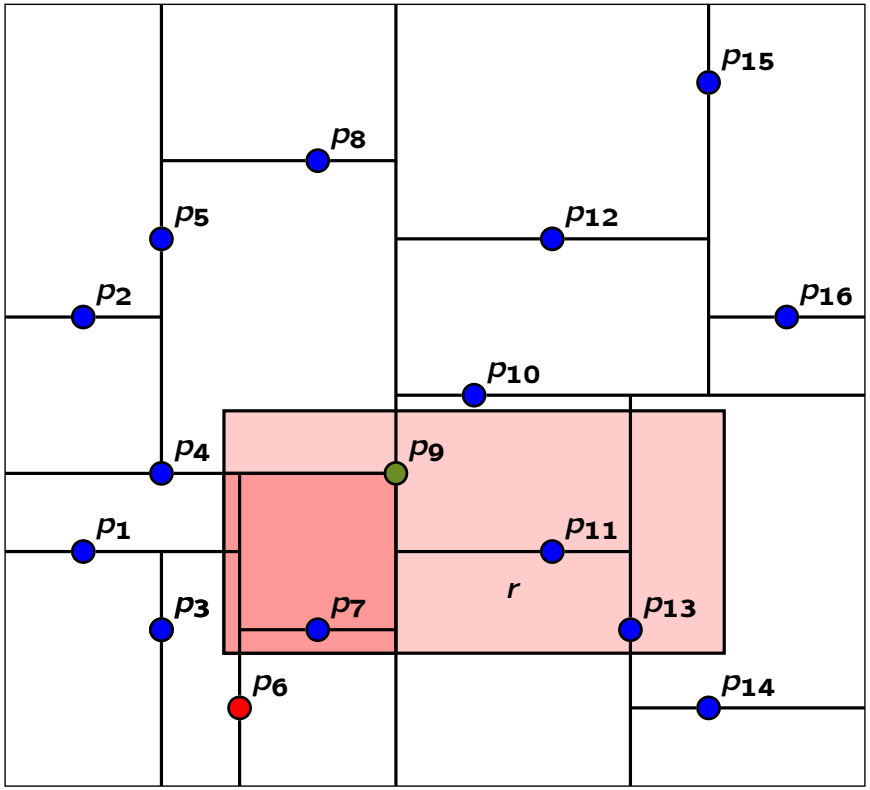



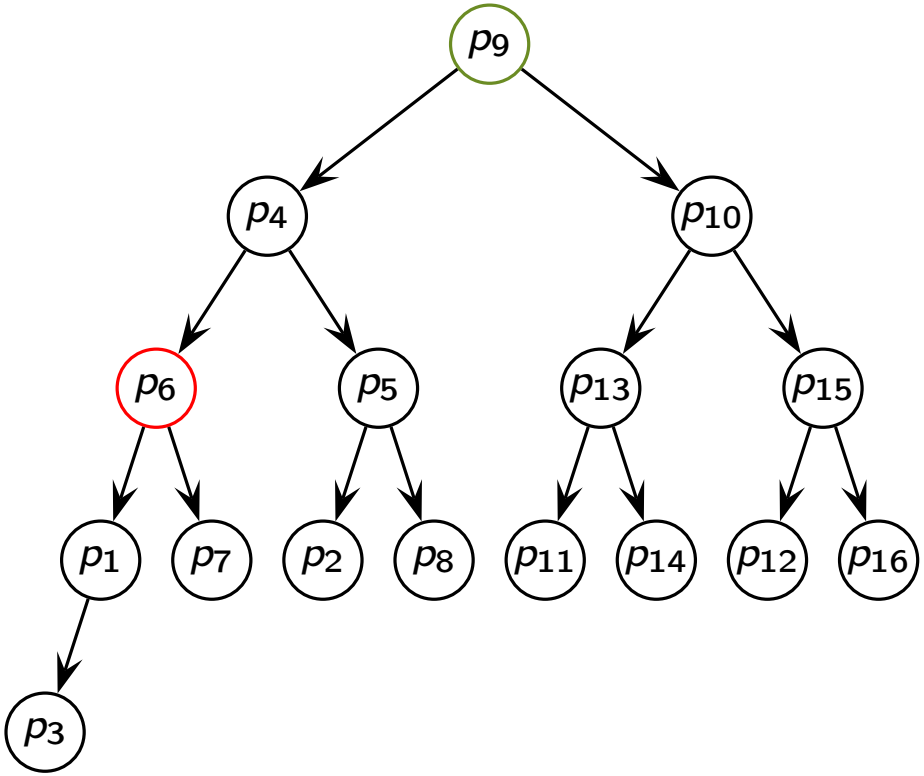
# 2-Dimensional Kd-tree Query Example



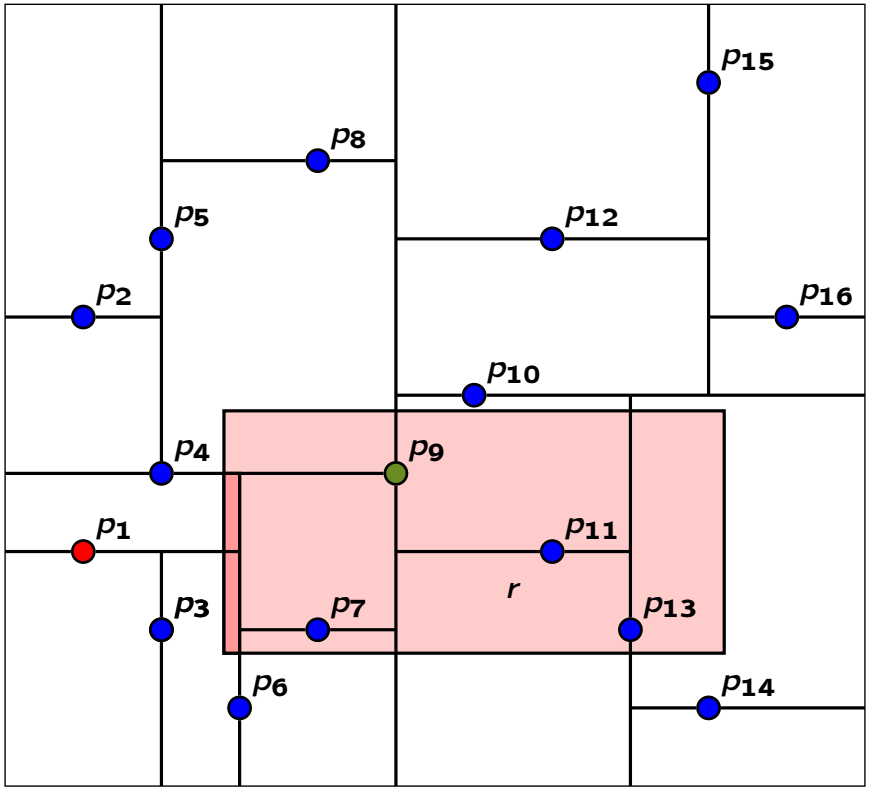



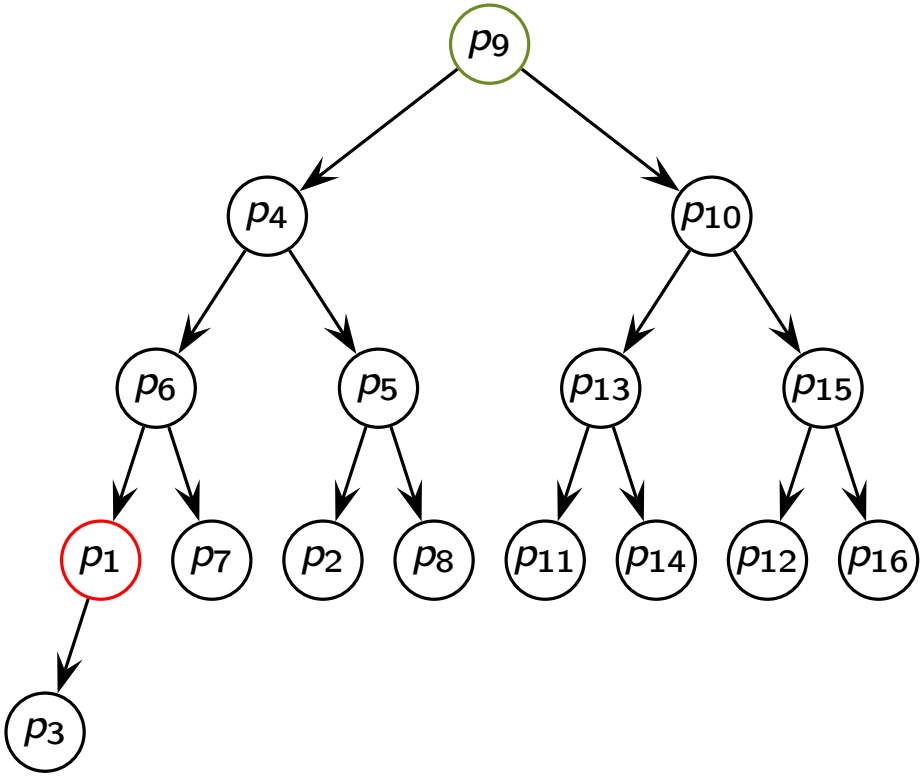
# 2-Dimensional Kd-tree Query Example



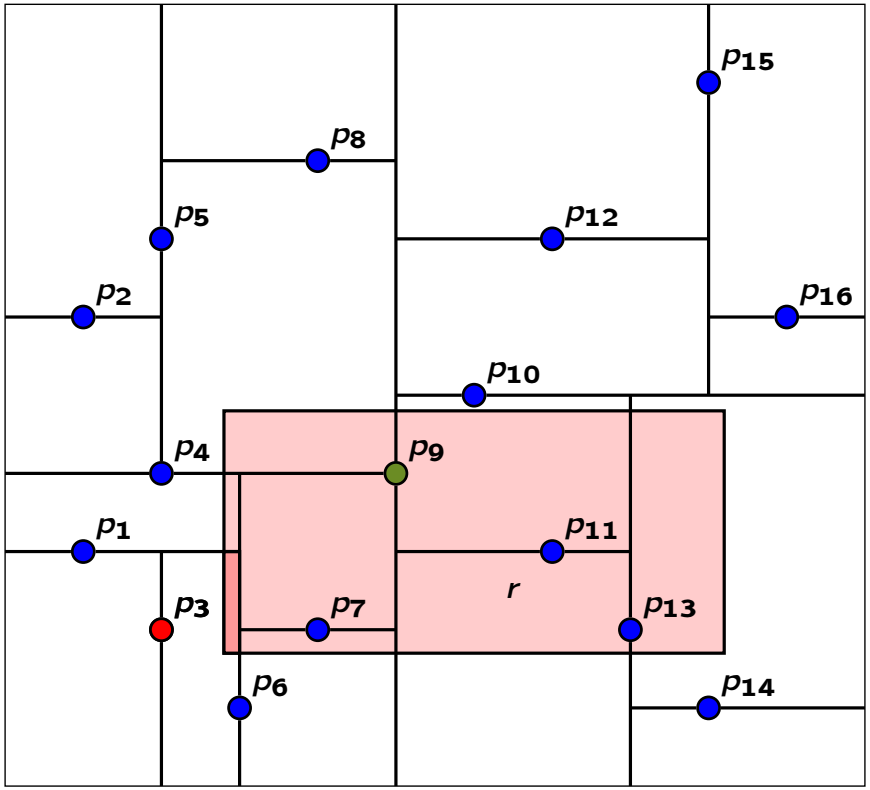



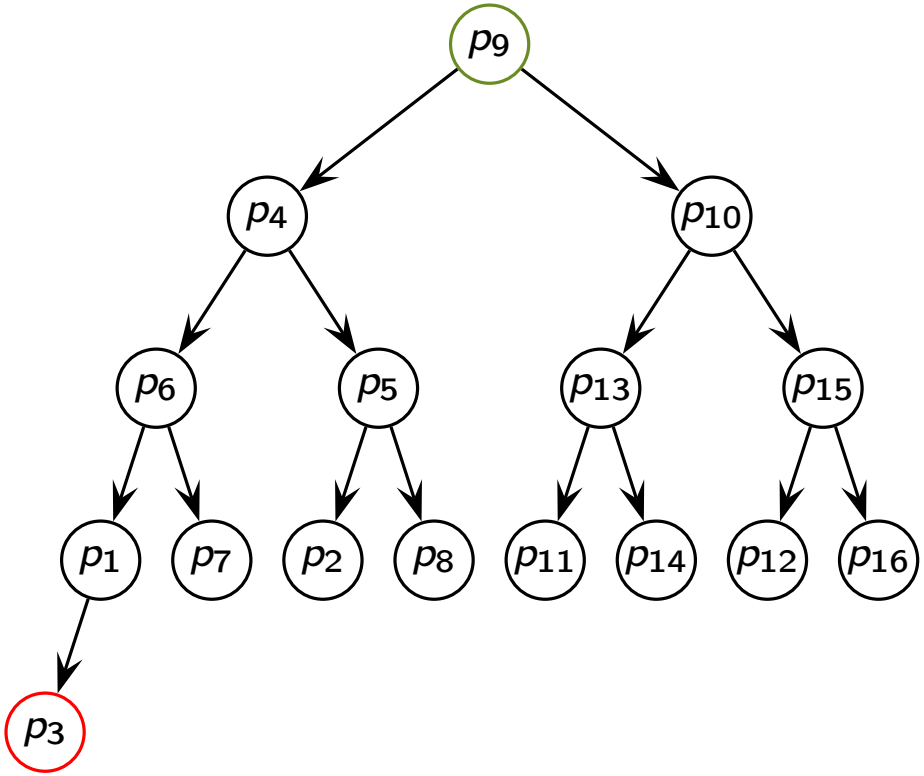
# 2-Dimensional Kd-tree Query Example



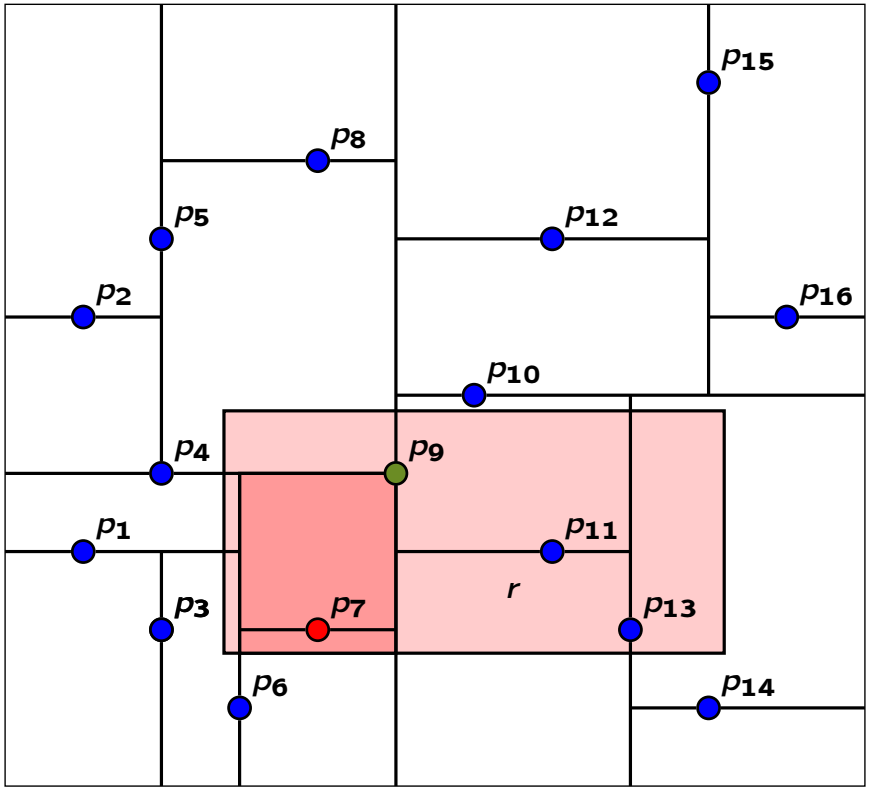



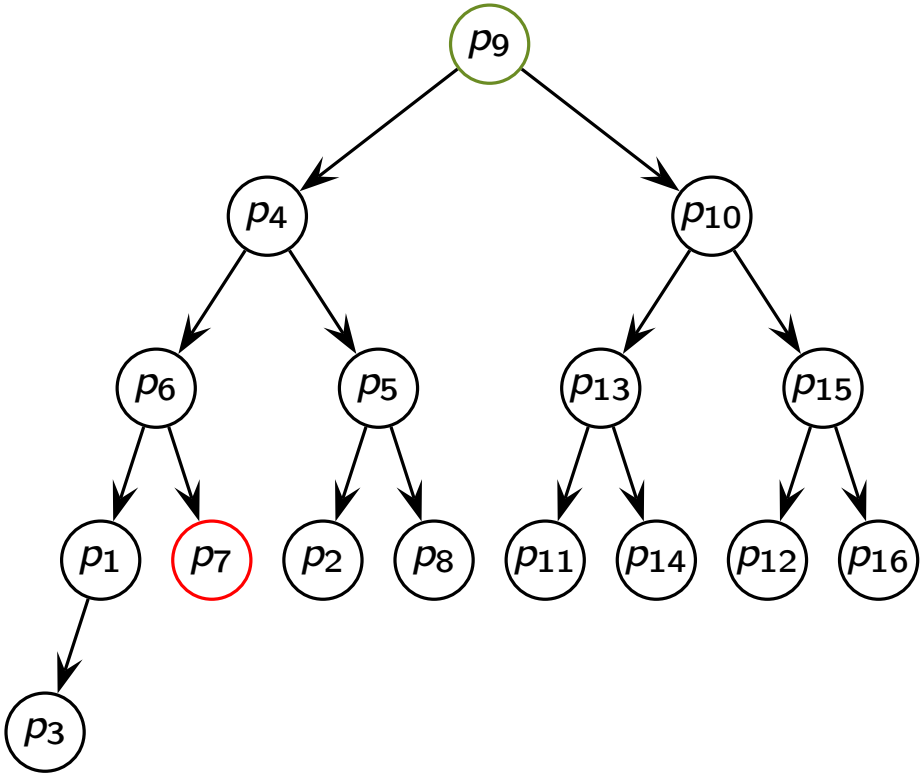
# 2-Dimensional Kd-tree Query Example



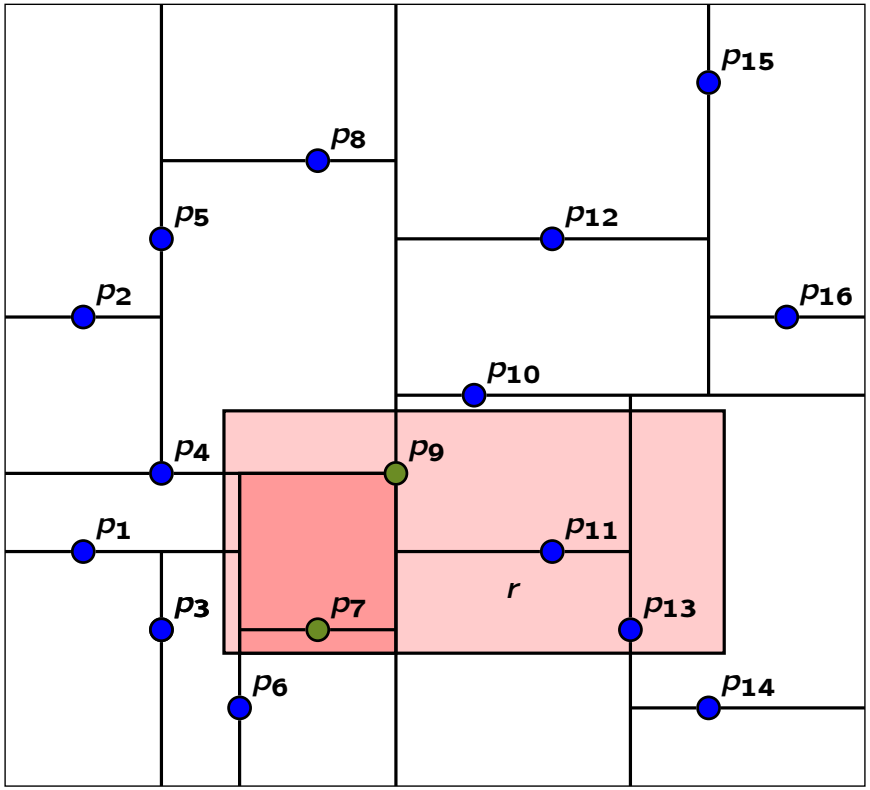



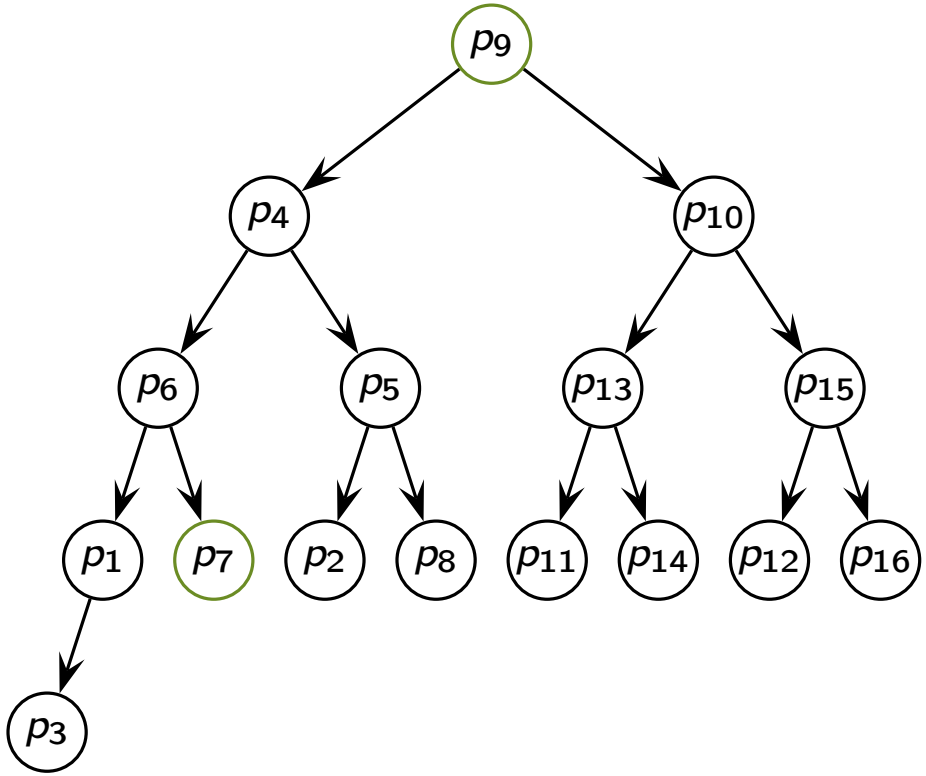
# 2-Dimensional Kd-tree Query Example



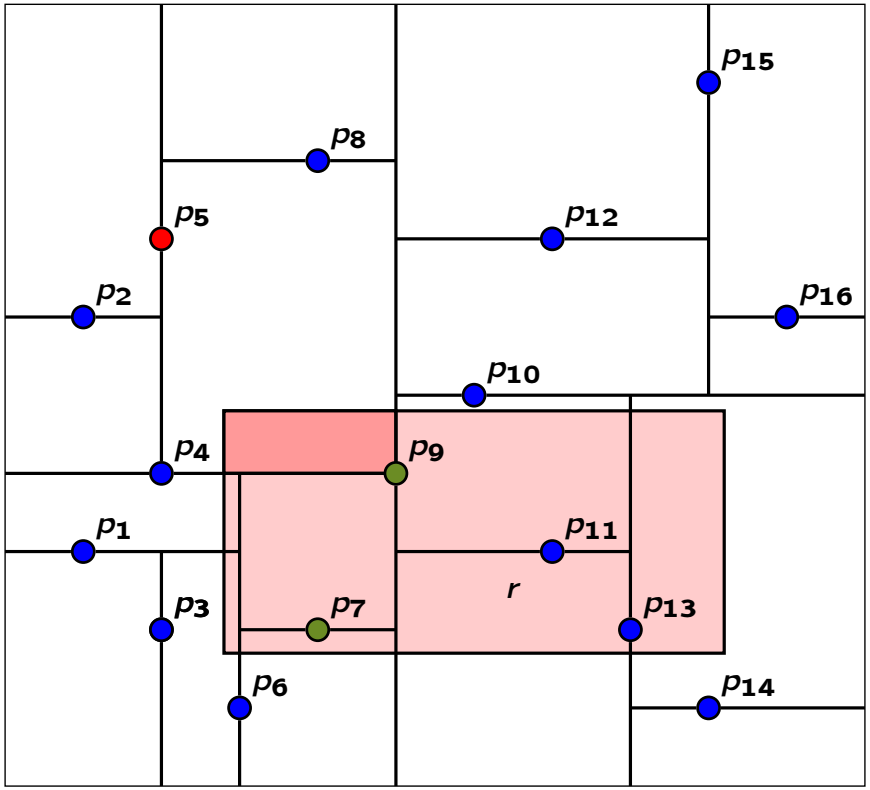
# 2-Dimensional Kd-tree Query Example

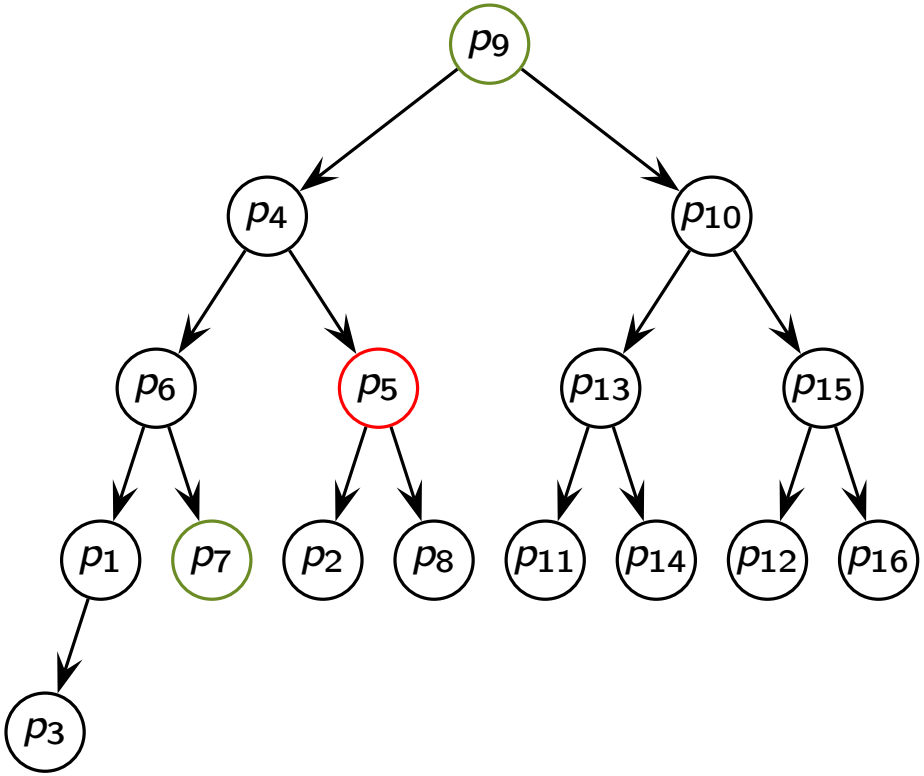



- All nodes rooted at $p_7$ right child, had existed, are rendered black

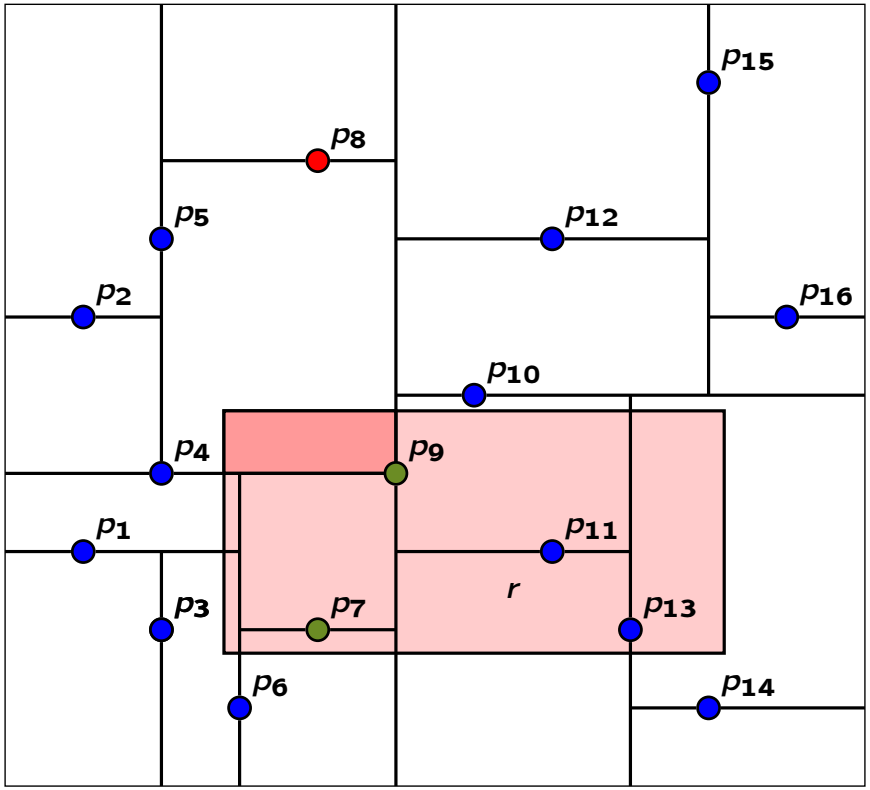
# 2-Dimensional Kd-tree Query Example

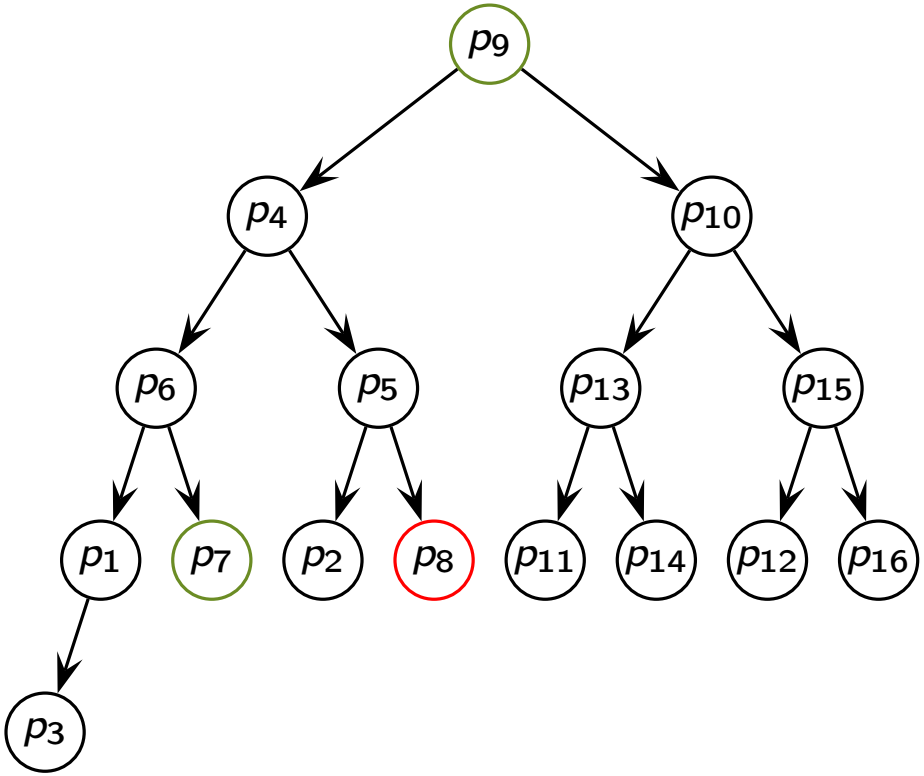



- All nodes rooted at $p_7$ right child, had existed, are rendered black

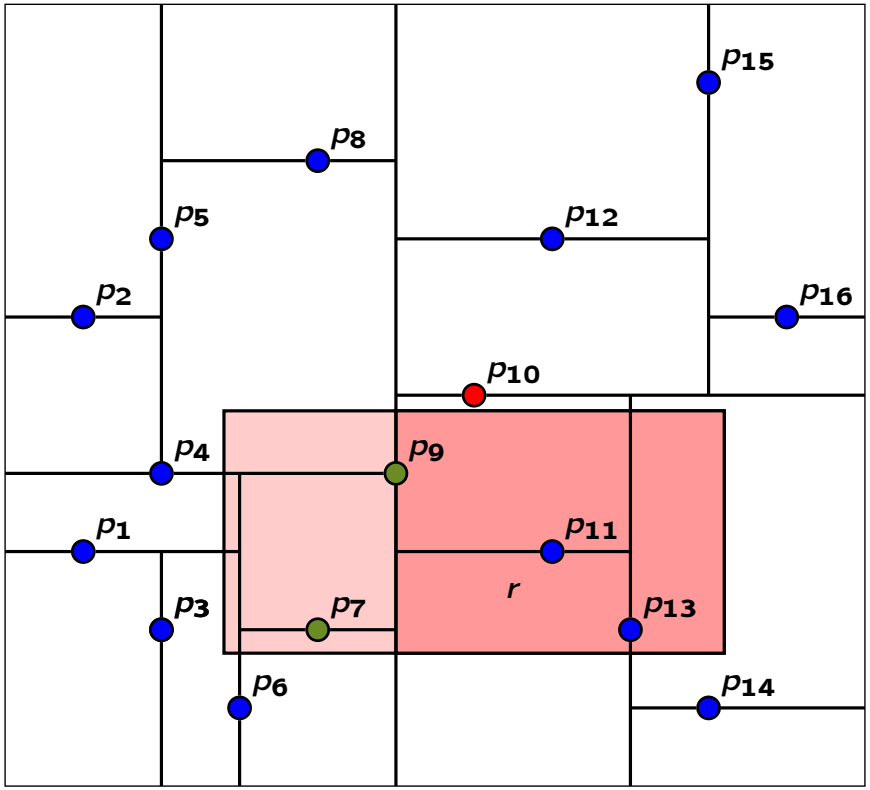
# 2-Dimensional Kd-tree Query Example

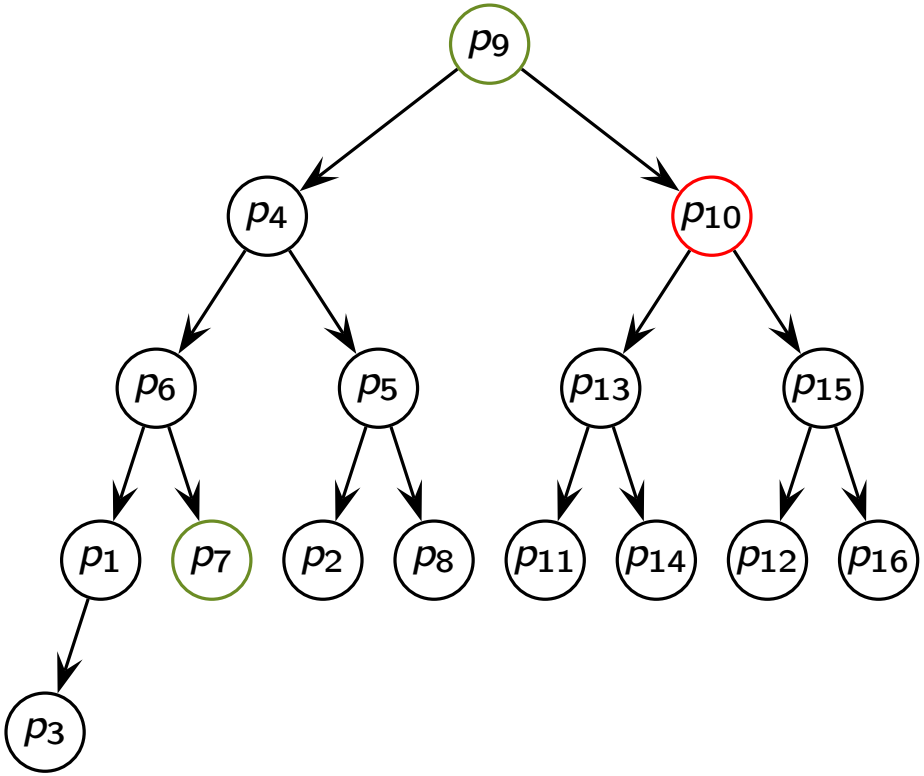



- All nodes rooted at $p_7$ right child, had existed, are rendered black

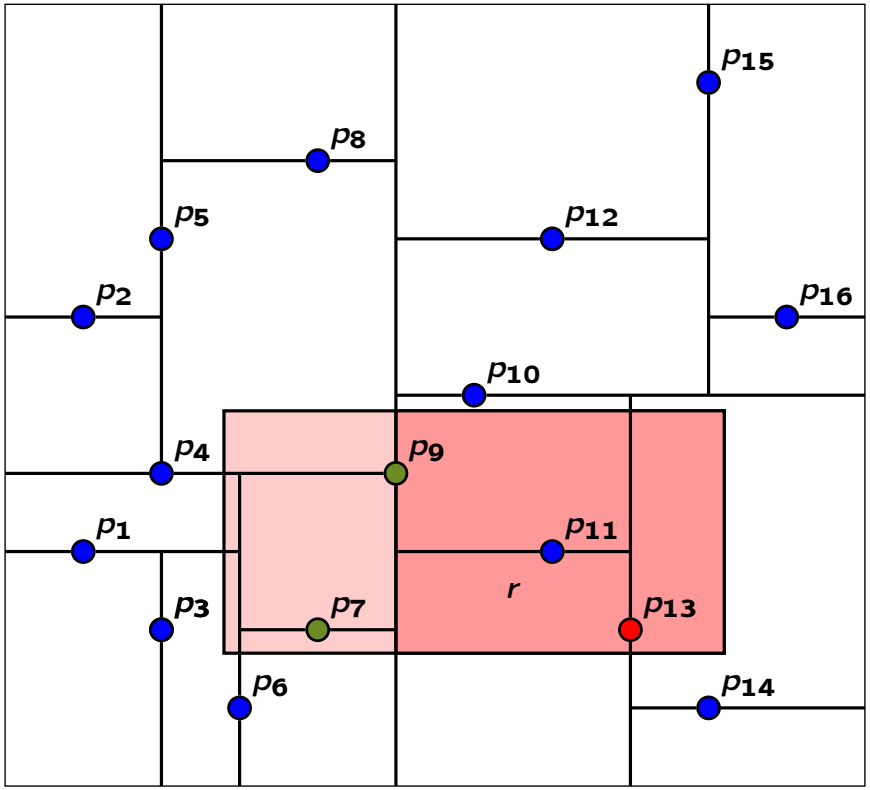
# 2-Dimensional Kd-tree Query Example

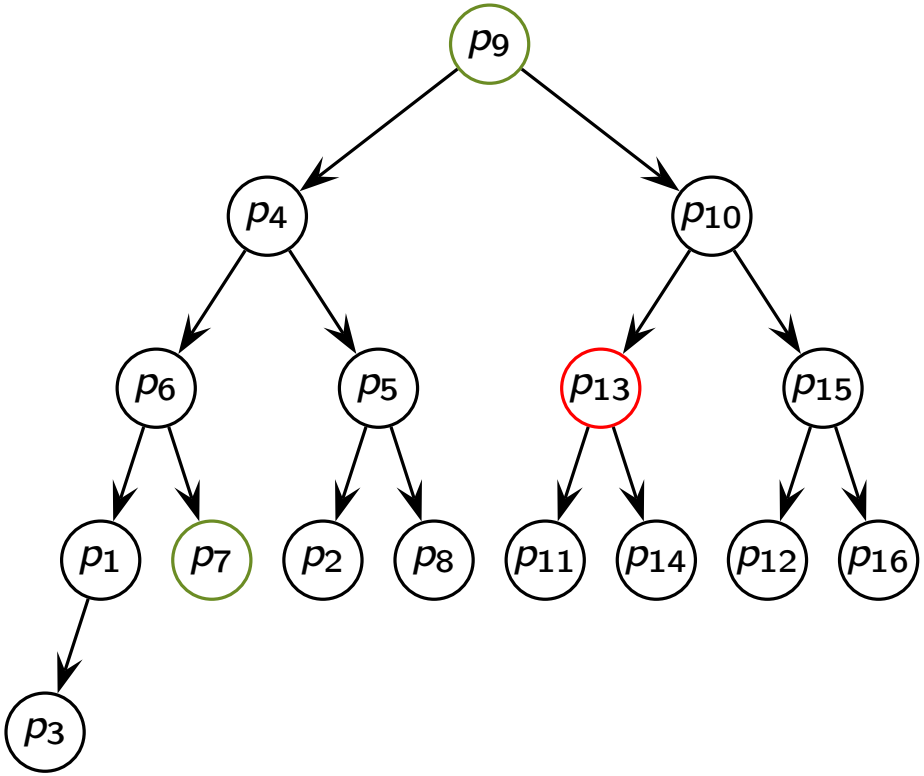



- All nodes rooted at $p_7$ right child, had existed, are rendered black

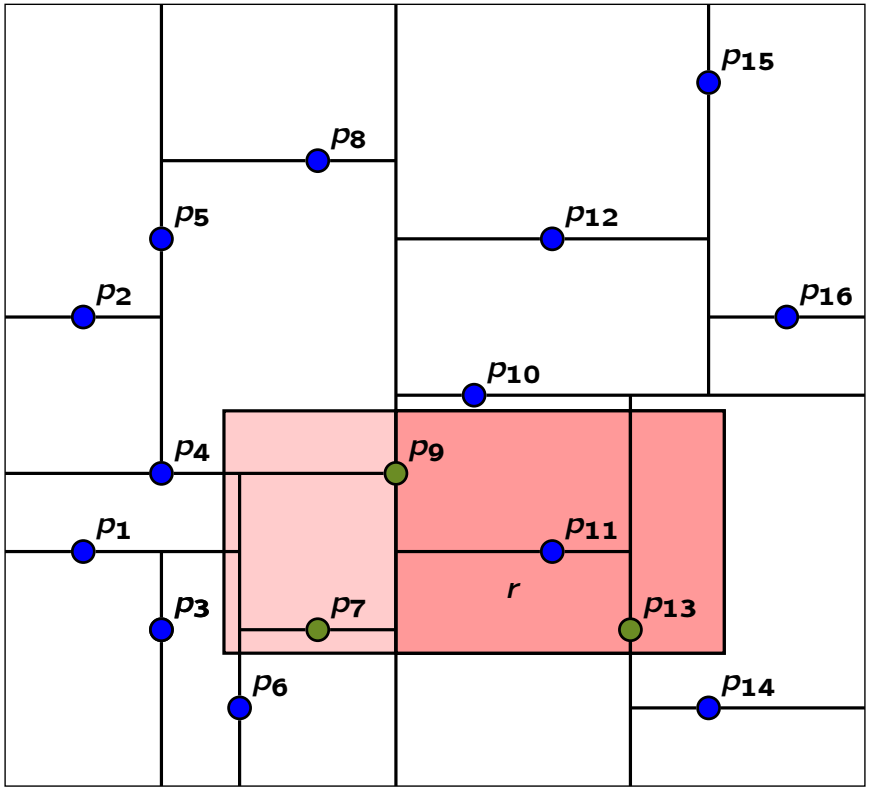
# 2-Dimensional Kd-tree Query Example

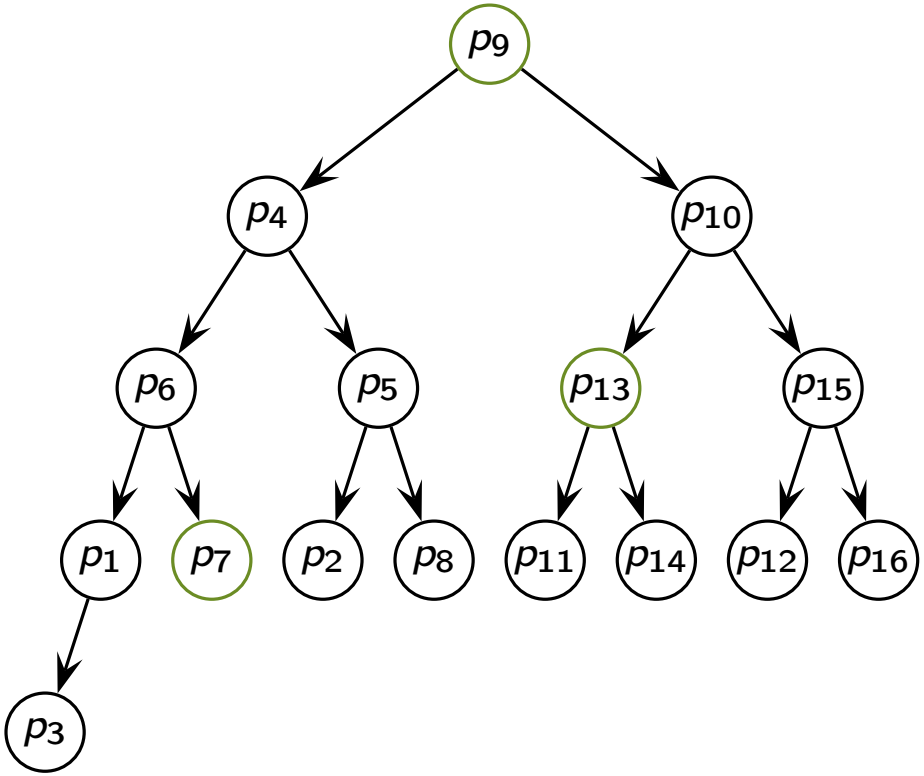



- All nodes rooted at $p_7$ right child, had existed, are rendered black

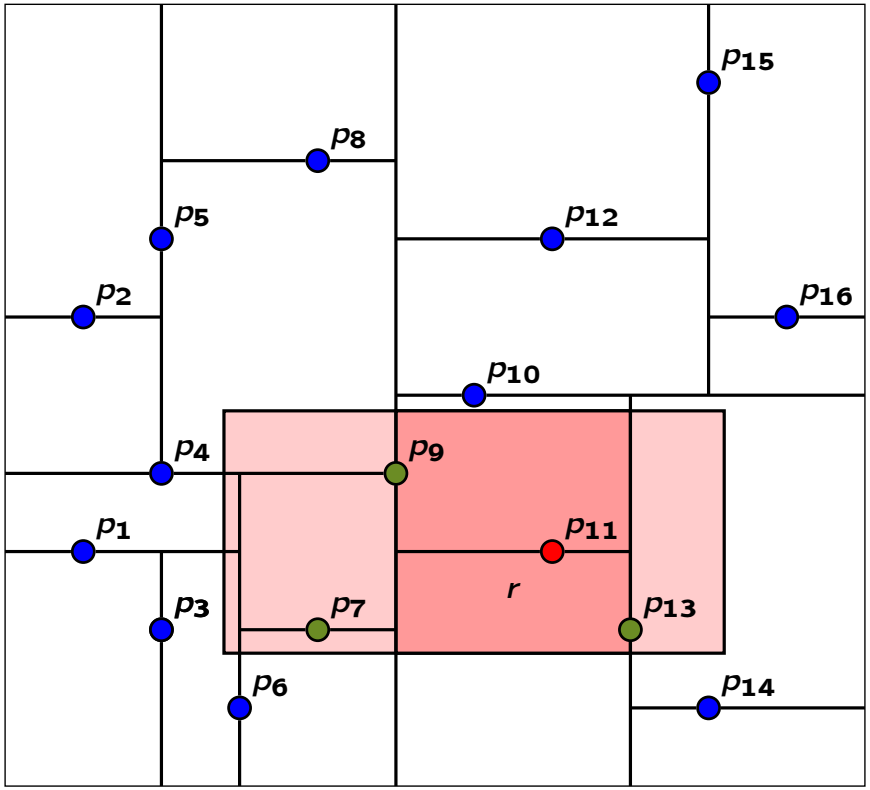
# 2-Dimensional Kd-tree Query Example

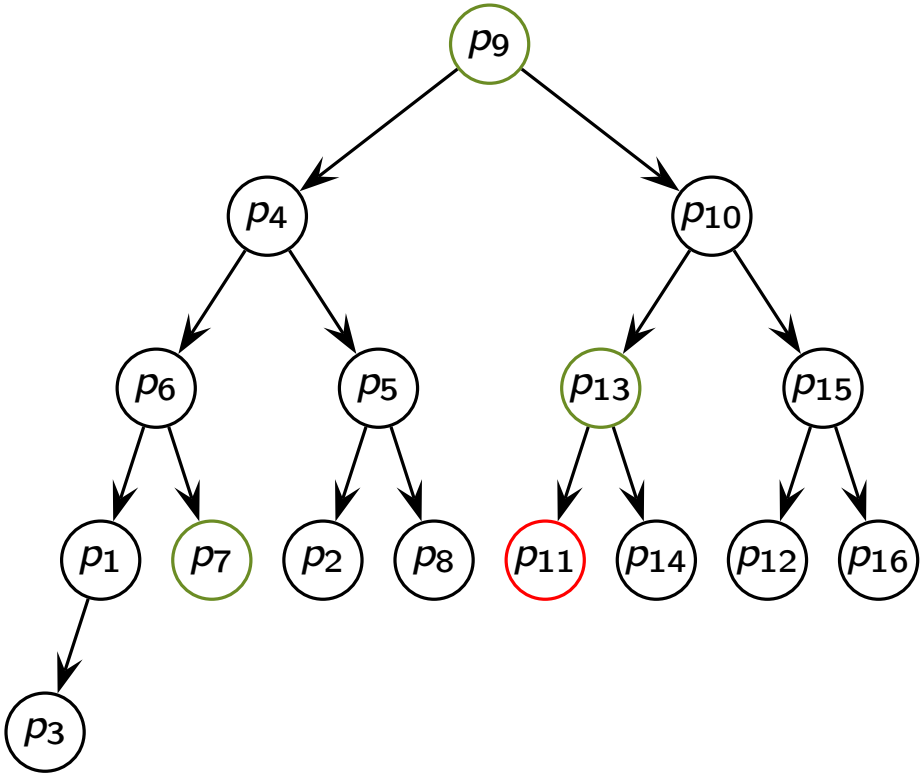



- All nodes rooted at $p_7$ right child, had existed, are rendered black

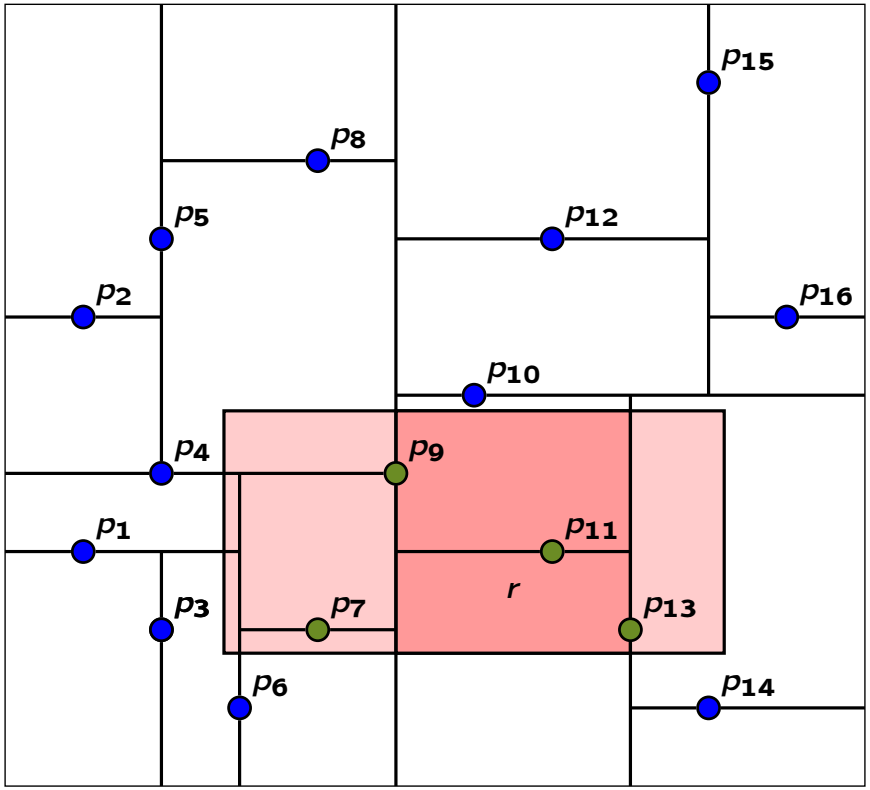
# 2-Dimensional Kd-tree Query Example

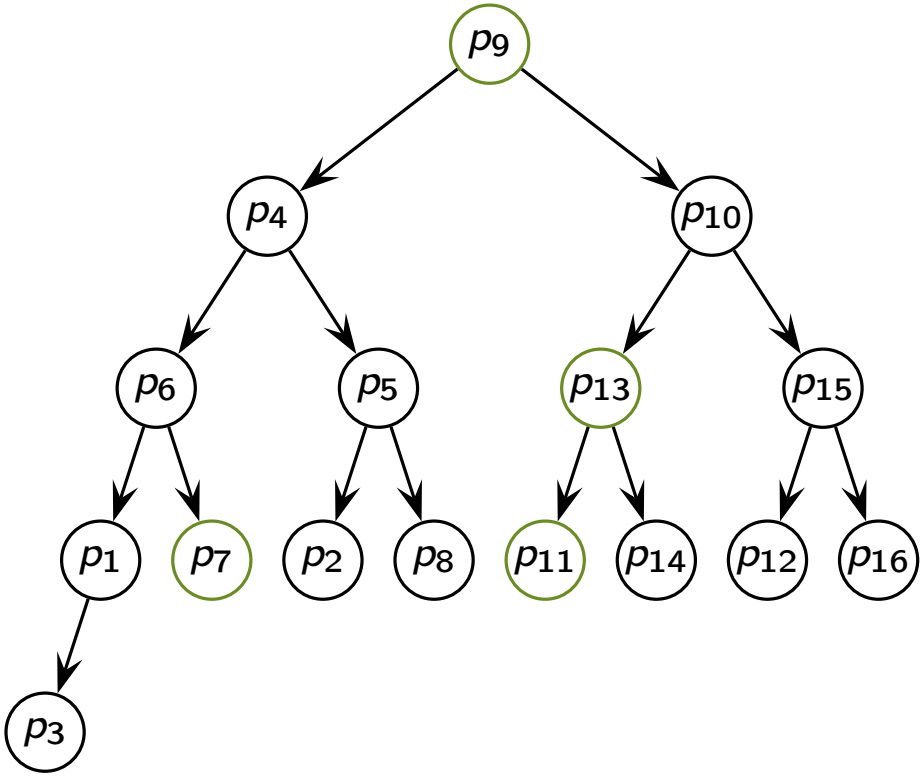



- All nodes rooted at $p_7$ right child, had existed, are rendered black

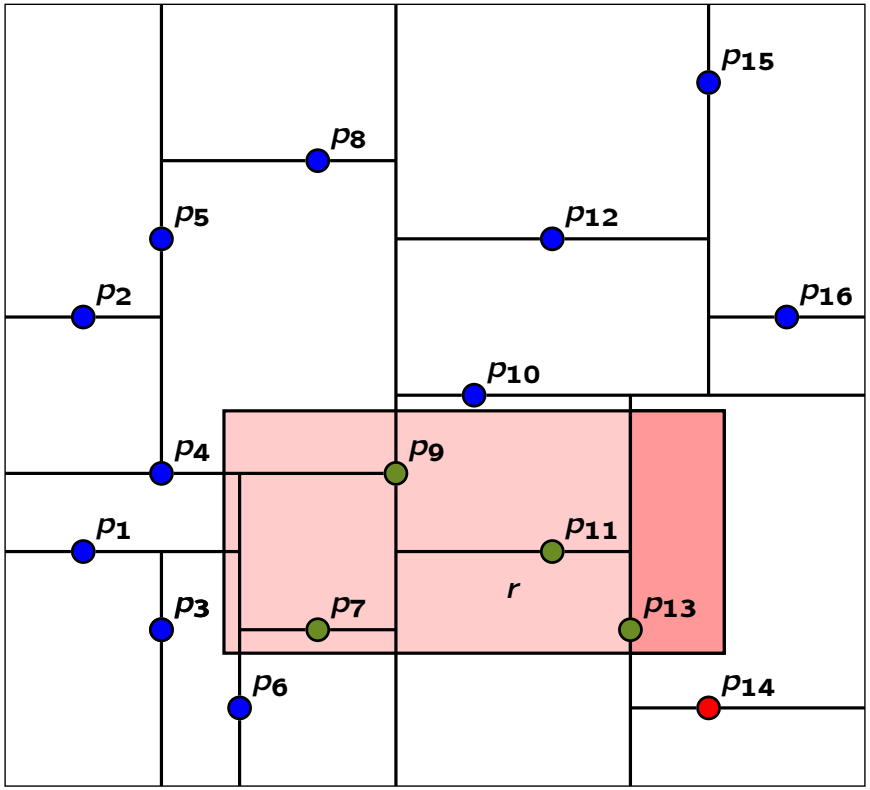
# 2-Dimensional Kd-tree Query Example

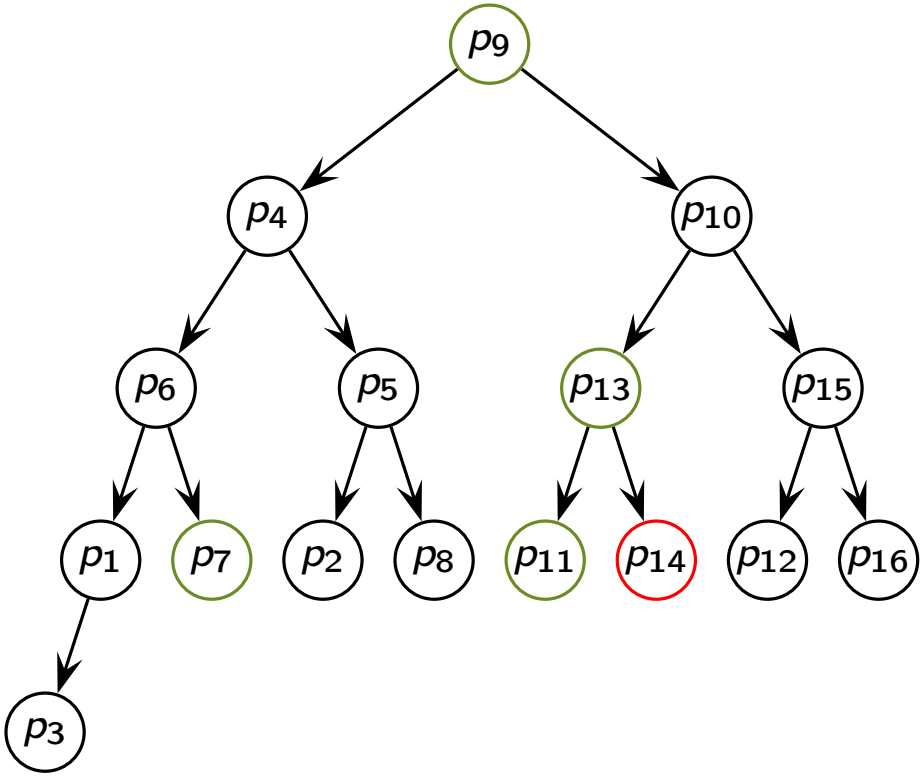



- All nodes rooted at $p_7$ right child, had existed, are rendered black

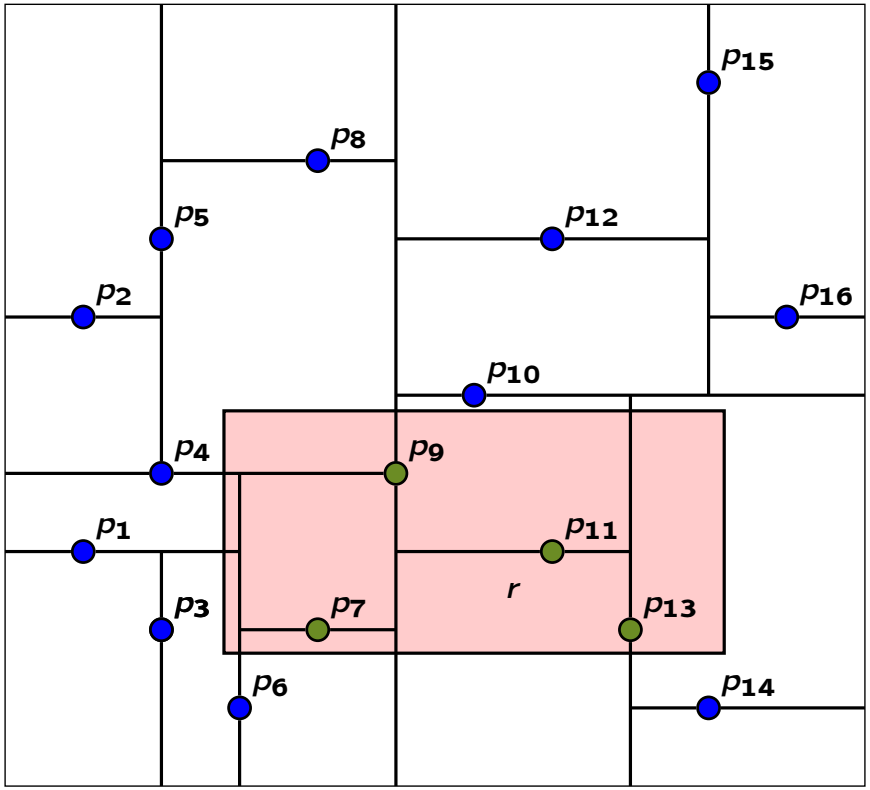
# 2-Dimensional Kd-tree Query Example

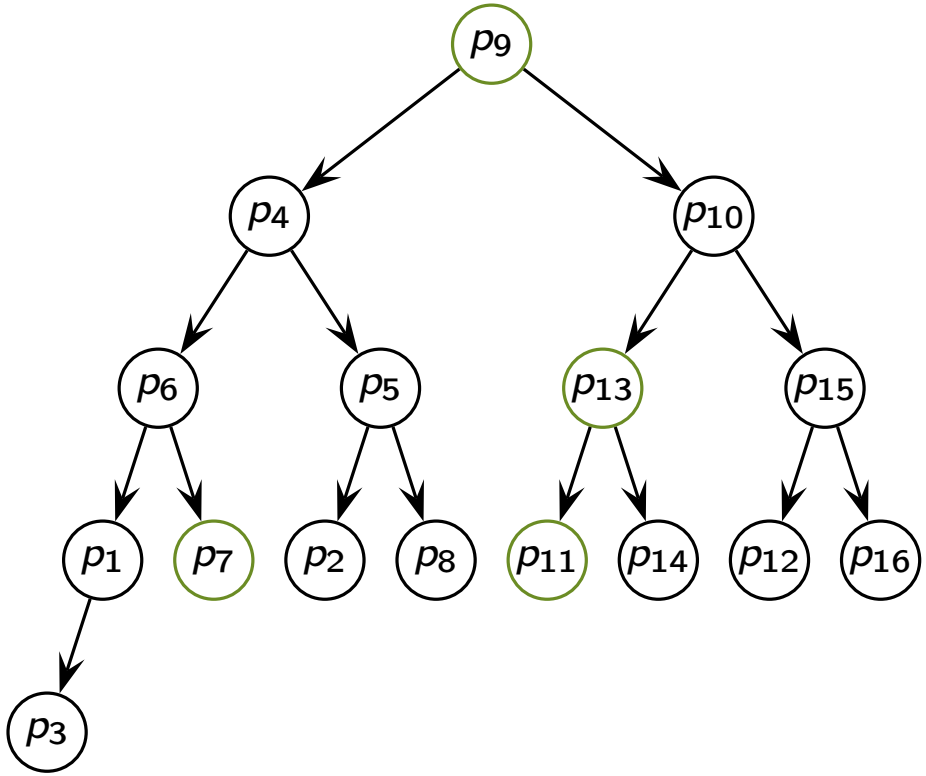



- All nodes rooted at $p_7$ right child, had existed, are rendered black

# 2-Dimensional Kd-tree Query Example

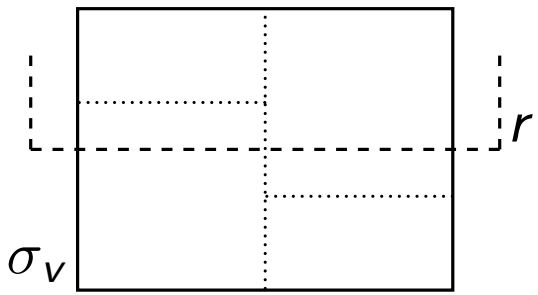


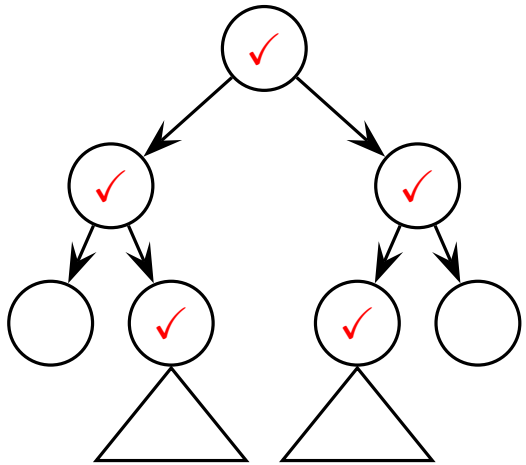




- All nodes rooted at $p_7$ right child, had existed, are rendered black
- Exercise: eliminate the geometric tests in lines 1., 2., and 3. for black nodes using state variables

# Kd-tree Query Complexity

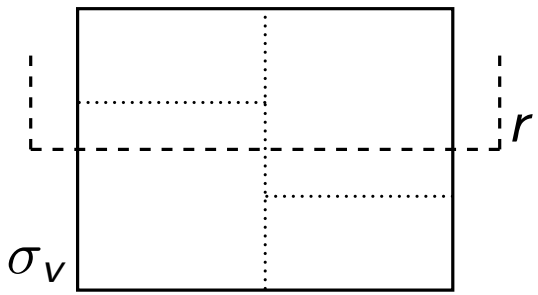
- Query time is proportional to # of black nodes + # of gray nodes
- $k$ — # of points in the rectangle $r$
- The number of black nodes is bounded by $k$
- $\phi(n)$ — # of nodes $v'$ such that a horizontal edge of $r$ intersects $\sigma_{v'}$
- $\phi(n) = 2\phi(n/4) + 3 \Rightarrow \phi(n) = O(\sqrt{n}) \Rightarrow T(n) = O(\sqrt{n} + k)$

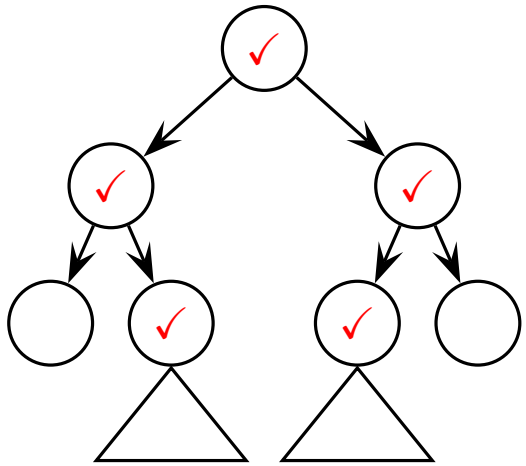


# Kd-tree Query Complexity

- Query time is proportional to # of black nodes + # of gray nodes
- $k$ — # of points in the rectangle $r$
- The number of black nodes is bounded by $k$
- $\phi(n)$ — # of nodes $v'$ such that a horizontal edge of $r$ intersects $\sigma_{v'}$
- $\phi(n) = 2\phi(n/4) + 3 \Rightarrow \phi(n) = O(\sqrt{n}) \Rightarrow T(n) = O(\sqrt{n} + k)$



- Querying a d-dimensional rectangle takes $O(n^{1-1/d} + k)$ time
  - In many practical situations the query rectangle is small and intersects much fewer regions

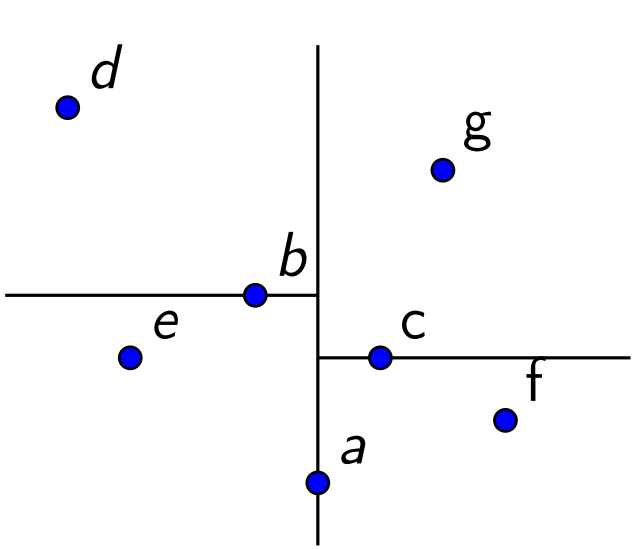
# Finding the Nearest Neighbor using Kd-trees

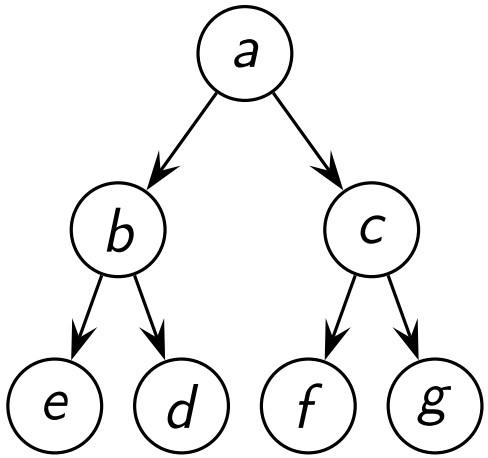
- A Kd-tree can be used to efficiently find the nearest neighbor of a target point, $t$, in $S$

- Find a first approximation for the nearest neighbor going down
- Use the point $p$ contained therein as the "current best"
  - $p$ may not be the nearest neighbor
  - The nearest neighbor lies within the circle defined by $t$ and $p$
- Back up to the parent of the current node
- Does a closer point exist in the parent-node other child?
  - If so, recompute the circle, proceed down the tree and repeat
- Move up the tree and repeat

# Kd-tree Nearest-Neighbor Complexity

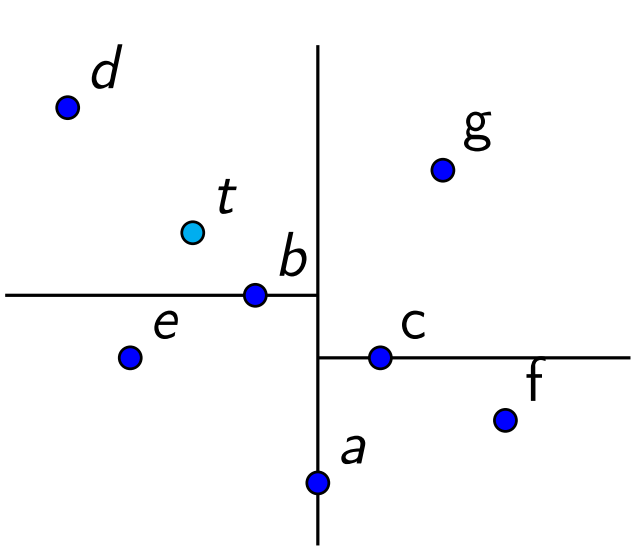
- At least $\Omega(\log n)$ nodes are inspected
  - At least one leaf node is examined
- The number of nodes examined can not exceed $n$
- Under certain assumptions the expected time for finding the nearest neighbor is $O(\log n)$
  - Depends on the expected distribution of the points and on the distribution of the target points [FBP80]

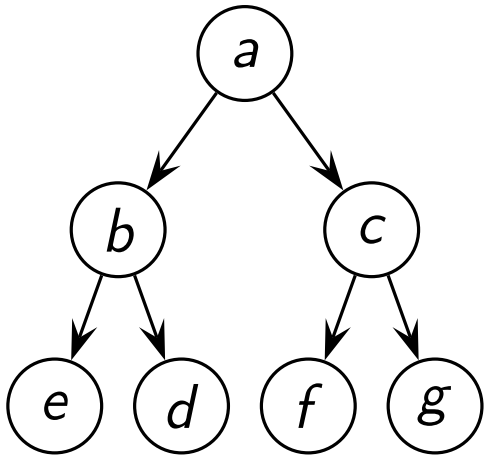



# Kd-tree Nearest-Neighbor Complexity

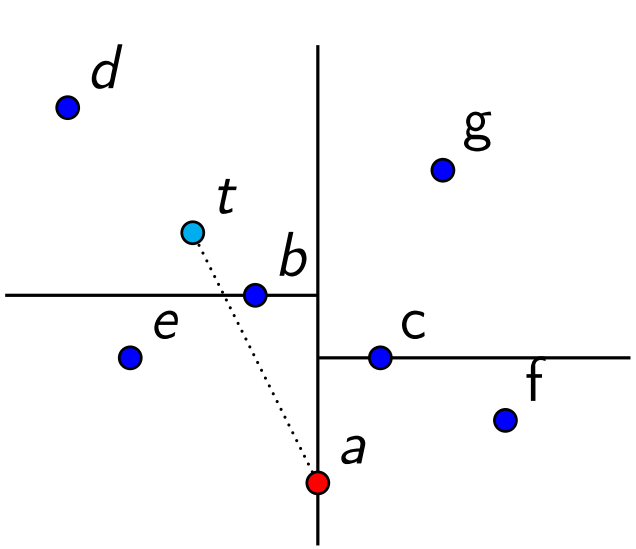
- At least $\Omega(\log n)$ nodes are inspected
  - At least one leaf node is examined
- The number of nodes examined can not exceed $n$
- Under certain assumptions the expected time for finding the nearest neighbor is $O(\log n)$
  - Depends on the expected distribution of the points and on the distribution of the target points [FBP80]

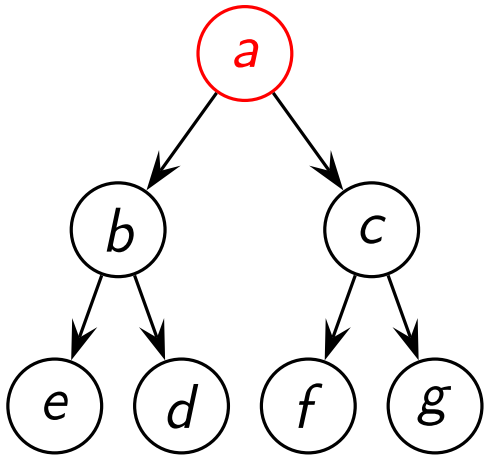



# Kd-tree Nearest-Neighbor Complexity

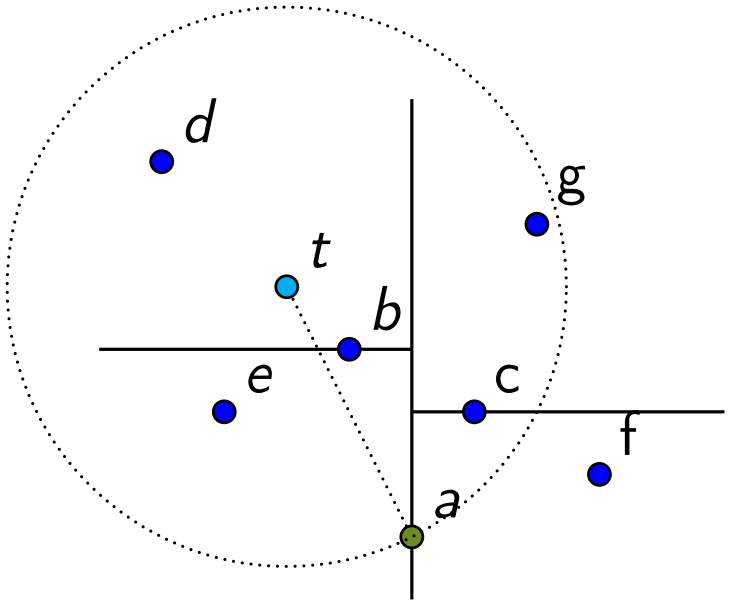
- At least $\Omega(\log n)$ nodes are inspected
  - At least one leaf node is examined
- The number of nodes examined can not exceed $n$
- Under certain assumptions the expected time for finding the nearest neighbor is $O(\log n)$
  - Depends on the expected distribution of the points and on the distribution of the target points [FBP80]

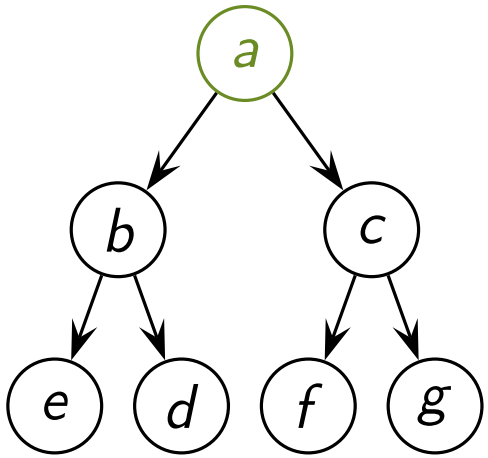



# Kd-tree Nearest-Neighbor Complexity

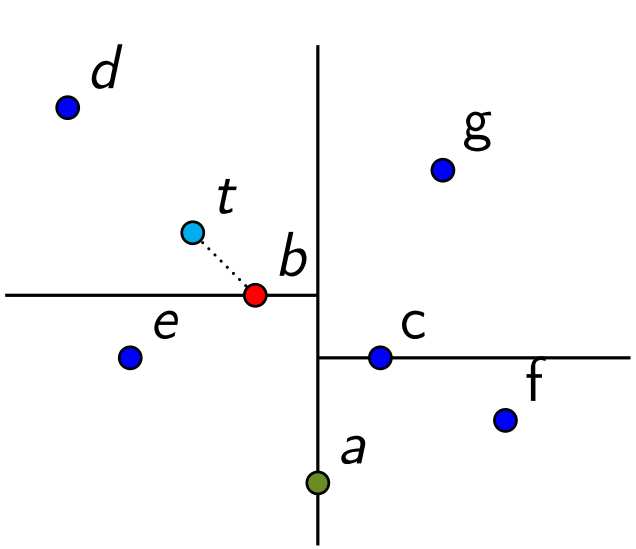
- At least $\Omega(\log n)$ nodes are inspected
  - At least one leaf node is examined
- The number of nodes examined can not exceed $n$
- Under certain assumptions the expected time for finding the nearest neighbor is $O(\log n)$
  - Depends on the expected distribution of the points and on the distribution of the target points [FBP80]

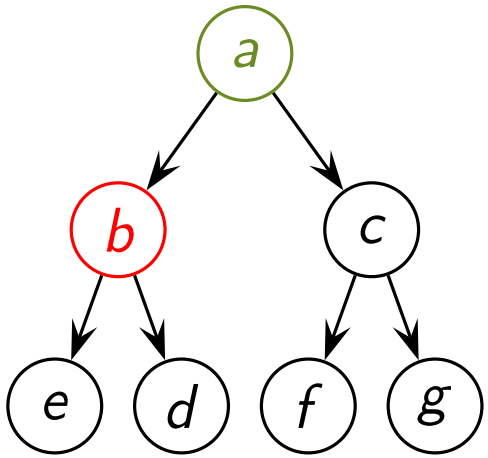



# Kd-tree Nearest-Neighbor Complexity

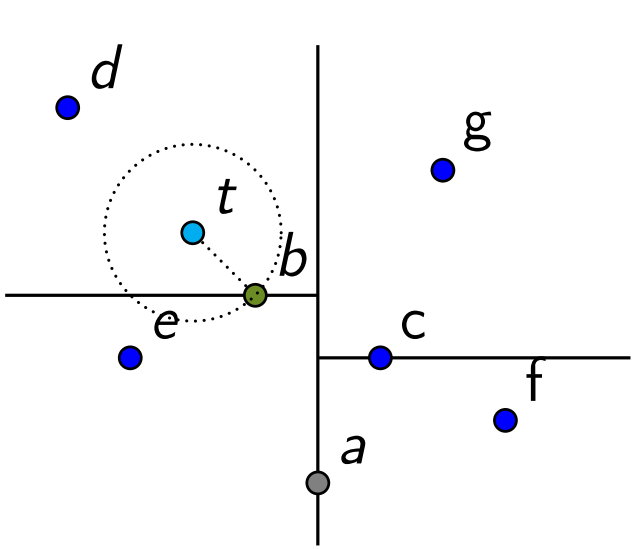
- At least $\Omega(\log n)$ nodes are inspected
  - At least one leaf node is examined
- The number of nodes examined can not exceed $n$
- Under certain assumptions the expected time for finding the nearest neighbor is $O(\log n)$
  - Depends on the expected distribution of the points and on the distribution of the target points [FBP80]

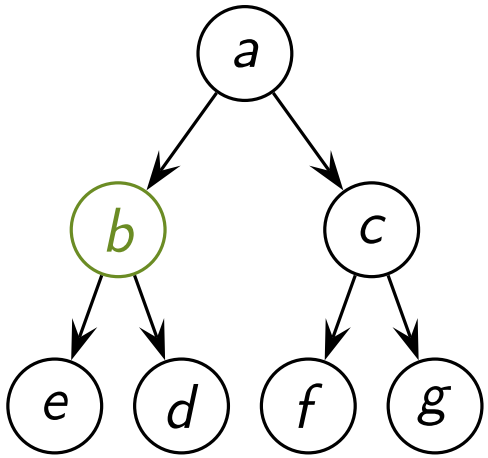



# Kd-tree Nearest-Neighbor Complexity

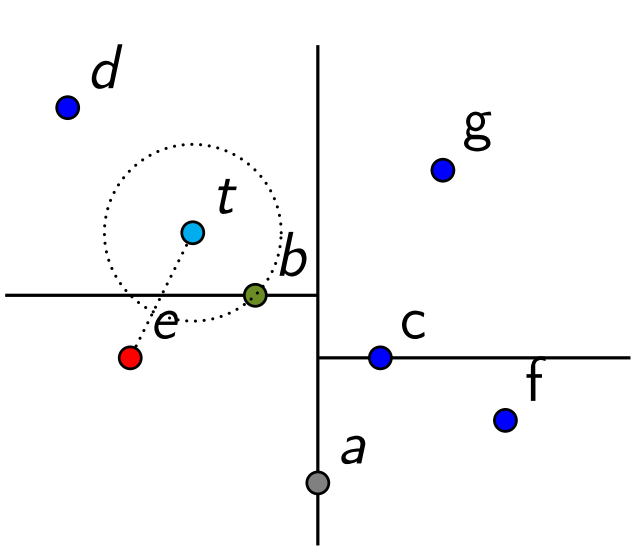
- At least $\Omega(\log n)$ nodes are inspected
  - At least one leaf node is examined
- The number of nodes examined can not exceed $n$
- Under certain assumptions the expected time for finding the nearest neighbor is $O(\log n)$
  - Depends on the expected distribution of the points and on the distribution of the target points [FBP80]

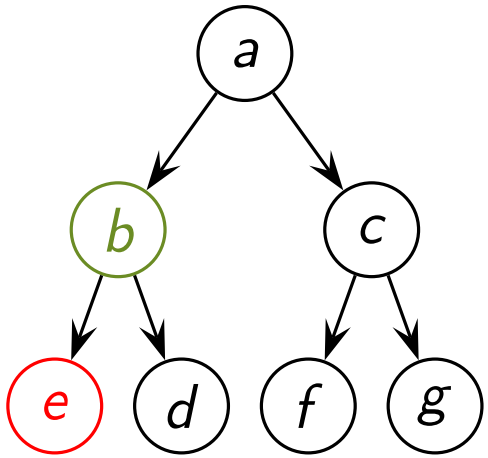



# Kd-tree Nearest-Neighbor Complexity

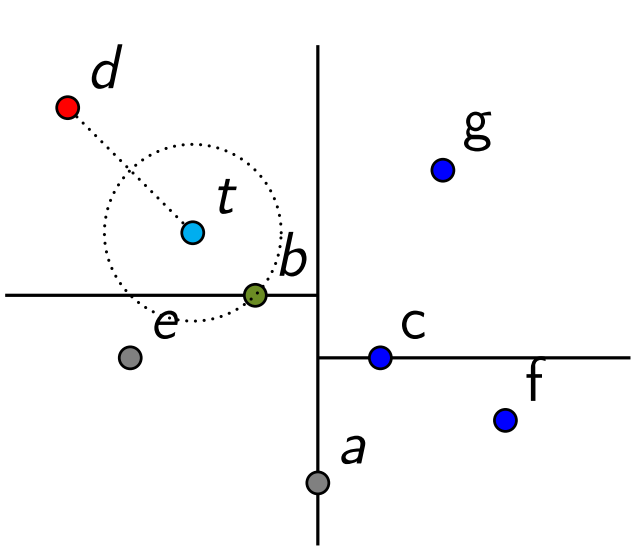
- At least $\Omega(\log n)$ nodes are inspected
  - At least one leaf node is examined
- The number of nodes examined can not exceed $n$
- Under certain assumptions the expected time for finding the nearest neighbor is $O(\log n)$
  - Depends on the expected distribution of the points and on the distribution of the target points [FBP80]

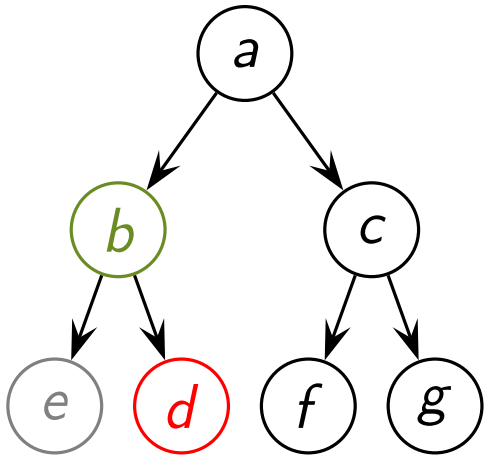



# Kd-tree Nearest-Neighbor Complexity

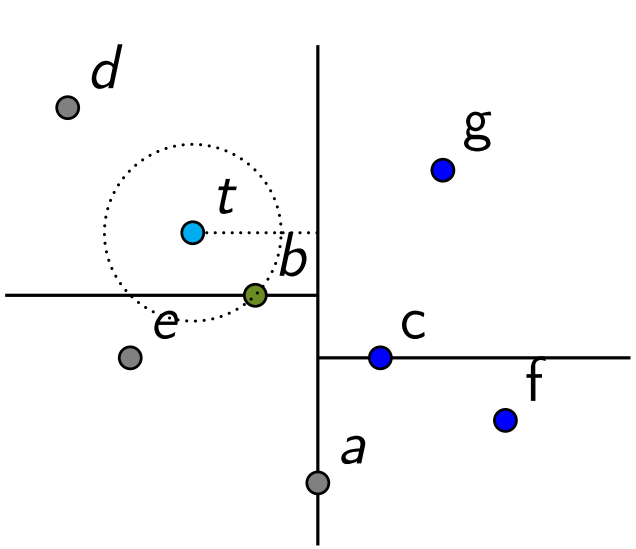
- At least $\Omega(\log n)$ nodes are inspected
  - At least one leaf node is examined
- The number of nodes examined can not exceed $n$
- Under certain assumptions the expected time for finding the nearest neighbor is $O(\log n)$
  - Depends on the expected distribution of the points and on the distribution of the target points [FBP80]

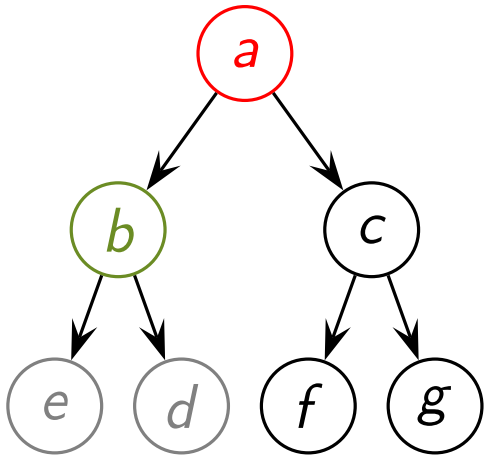


# Kd-tree Nearest-Neighbor Complexity

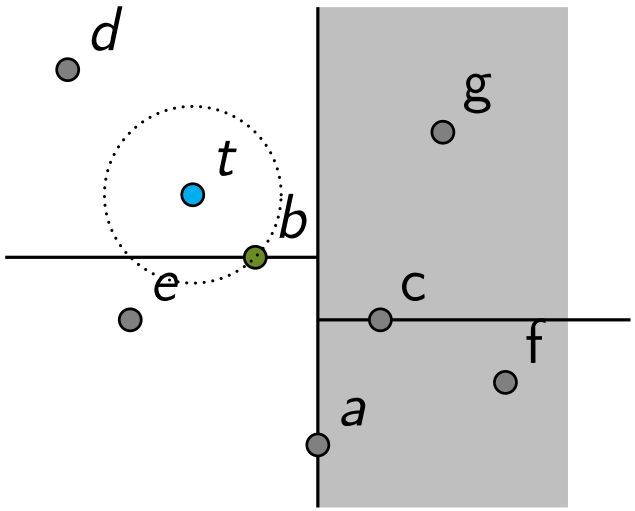
- At least $\Omega(\log n)$ nodes are inspected
  - At least one leaf node is examined
- The number of nodes examined can not exceed $n$
- Under certain assumptions the expected time for finding the nearest neighbor is $O(\log n)$
  - Depends on the expected distribution of the points and on the distribution of the target points [FBP80]

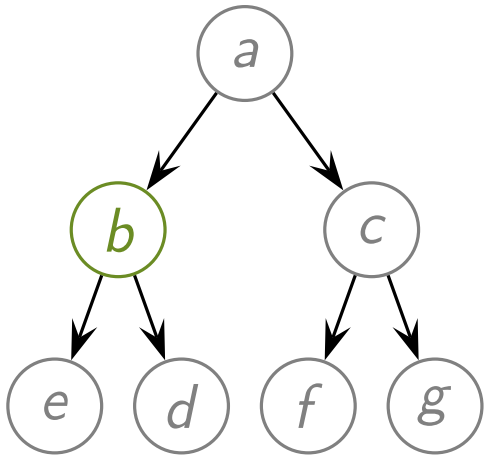



# Kd-tree Nearest-Neighbor Complexity

- At least $\Omega(\log n)$ nodes are inspected
  - At least one leaf node is examined
- The number of nodes examined can not exceed $n$
- Under certain assumptions the expected time for finding the nearest neighbor is $O(\log n)$
  - Depends on the expected distribution of the points and on the distribution of the target points [FBP80]




# Kd-trees Variations and Implementations

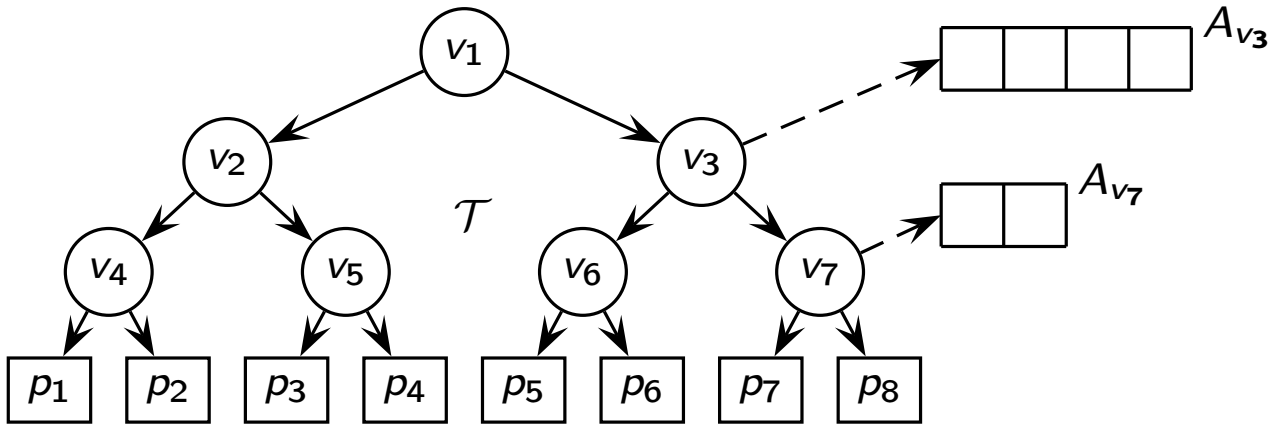
- Choosing the cutting dimension
  - Cycle through the dimensions one by one
    - ★ The cutting dimension need not be stored explicitly in each node
    - ★ May produce very skinny (elongated) cells
  - Select the cutting dimension with the greatest spread (difference between the largest and smallest coordinates)
- Implementations
  - Cgal Kd-tree
  - ANN Library [MA10]
- Applies only to points or $d$-dimensional axis-aligned rectangles
- Is a special case of Binary Space Partition (BSP) trees
  - The splitting lines (or hyperplanes in general) may be oriented in any direction
  - The cells are convex polygons

# Range Trees

## Definition (Range Tree)

A range tree is multi-level data structure.

- The primary structure is a balanced binary tree $\mathcal{T}$ ordered by $x$-coordinate such that each point is stored at a leaf node
- $S_v$ — the set of points in the subtree rooted at an internal node $v$
  - $S_{\text{root}} = S$
- $A_v$ — an array, stored at an internal node $v$, containing $S_v$ ordered by $y$-coordinate



# Range Tree Preprocessing Complexity

- Each point is stored only once at a given depth
- The total depth is $O(\log n) \Rightarrow S(n) = O(n \log n)$
- A range tree can be created by joining nodes from the bottom-up, while also merging ordered $A_v$ lists at each level
- The time spend at a node $v$ in $\mathcal{T}$ is linear in the size of $S_v \Rightarrow T(n) = O(n \log n)$

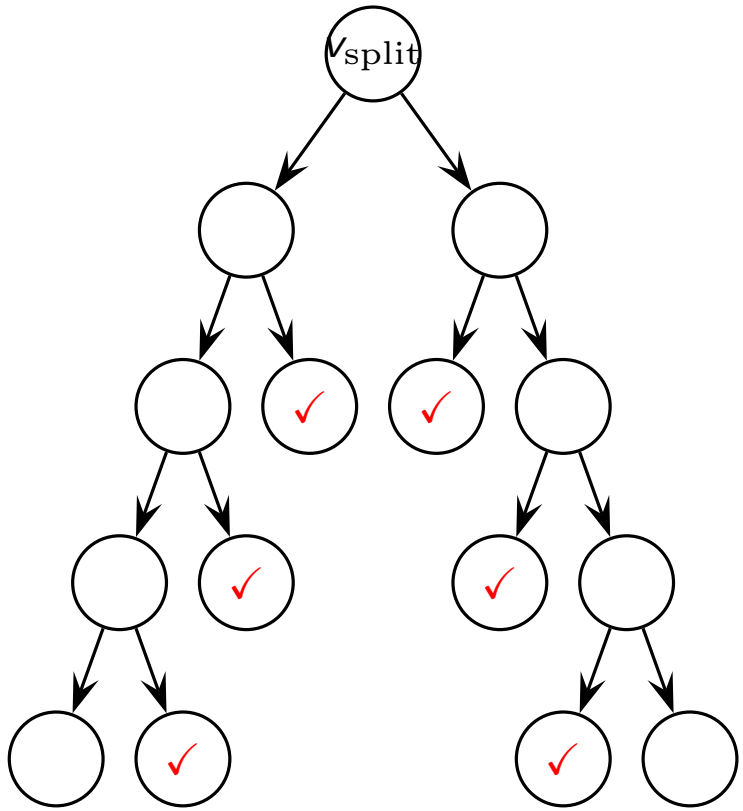
# Range Tree Query

2DRangeQuery($\mathcal{T}, [a_1, b_1] \times [a_2, b_2]$)

1. $v_{\text{split}} \leftarrow \text{FindSplitNode}(\mathcal{T}, a_1, b_1)$
2. **if** $v_{\text{split}}$ is a leaf, check whether $p(v_{\text{split}})$ must be reported
3. **else**
4. $\quad v \leftarrow \text{lc}(v_{\text{split}})$
5. $\quad$ **while** $v$ is not a leaf **do**
6. $\quad\quad$ **if** $a_1 \leq x(p(v))$
7. $\quad\quad\quad \text{1DRangeQuery}(A_{\text{rc}(v)}, a_2, b_2)$
8. $\quad\quad\quad v \leftarrow \text{lc}(v)$
9. $\quad\quad$ **else**
10. $\quad\quad\quad v \leftarrow \text{rc}(v)$
11. $\quad$ check whether $p(v)$ must be reported
12. $\quad v \leftarrow \text{rc}(v_{\text{split}})$
13. $\quad$ **while** $v$ is not a leaf **do**
14. $\quad\quad$ **if** $b_1 \geq x(p(v))$
15. $\quad\quad\quad \text{1DRangeQuery}(A_{\text{lc}(v)}, a_2, b_2)$
16. $\quad\quad\quad v \leftarrow \text{rc}(v)$
17. $\quad\quad$ **else**
18. $\quad\quad\quad v \leftarrow \text{lc}(v)$
19. $\quad$ check whether $p(v)$ must be reported

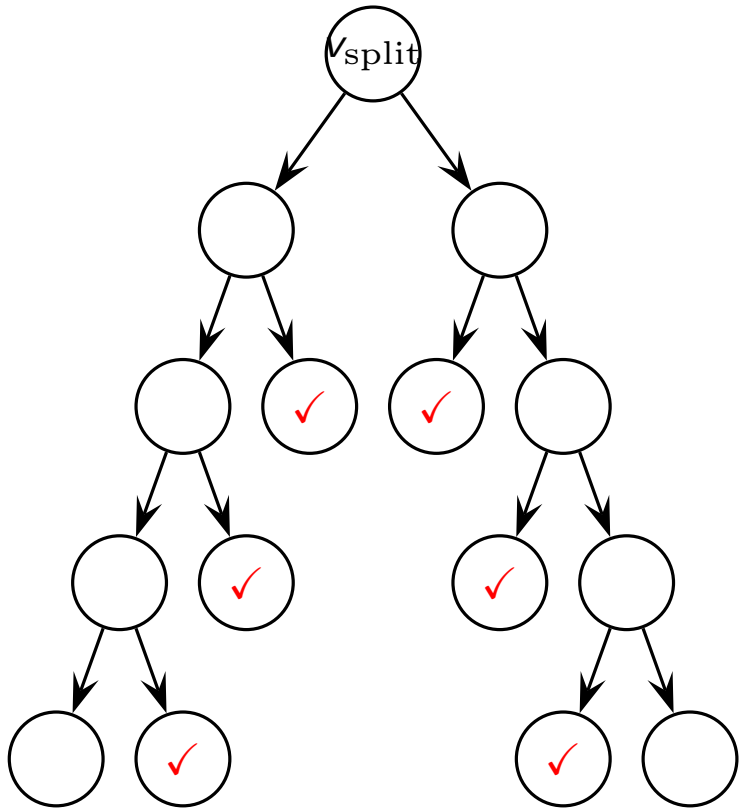
# Range Tree Query Complexity

- Queries are performed by searching for $a_1$ and $b_1$ in $\mathcal{T}$
- The $A_v$ lists of nodes that fall "inside" the query paths (checked) are searched for points in the interval $[a_2, b_2]$
- At most $2 \log n$ $A_v$ lists are searched, each in at most $O(\log n)$ time
  $\Rightarrow T(n) = O(\log^2 n + k)$




# Range Tree Query Complexity

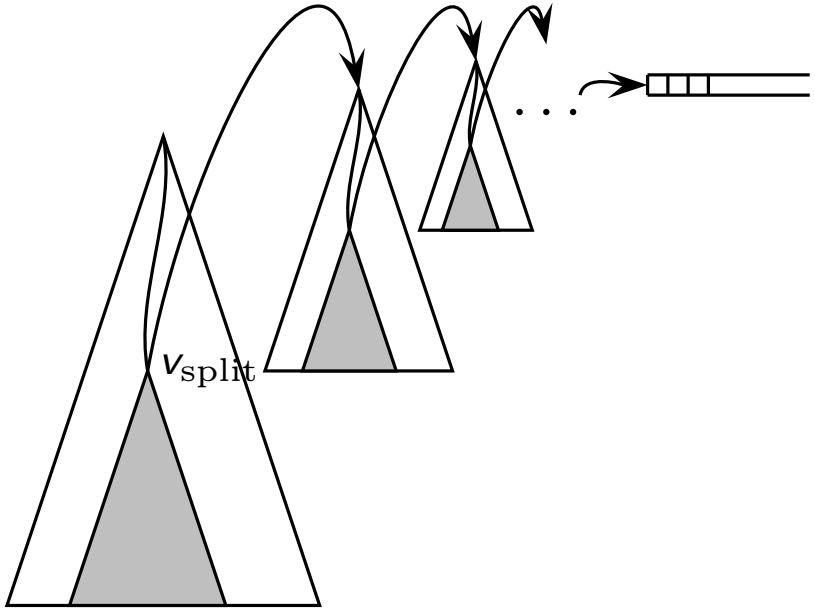
- Queries are performed by searching for $a_1$ and $b_1$ in $\mathcal{T}$
- The $A_v$ lists of nodes that fall "inside" the query paths (checked) are searched for points in the interval $[a_2, b_2]$
- At most $2 \log n$ $A_v$ lists are searched, each in at most $O(\log n)$ time
  $\Rightarrow T(n) = O(\log^2 n + k)$




- The query time can be improved to $O(\log n + k)$

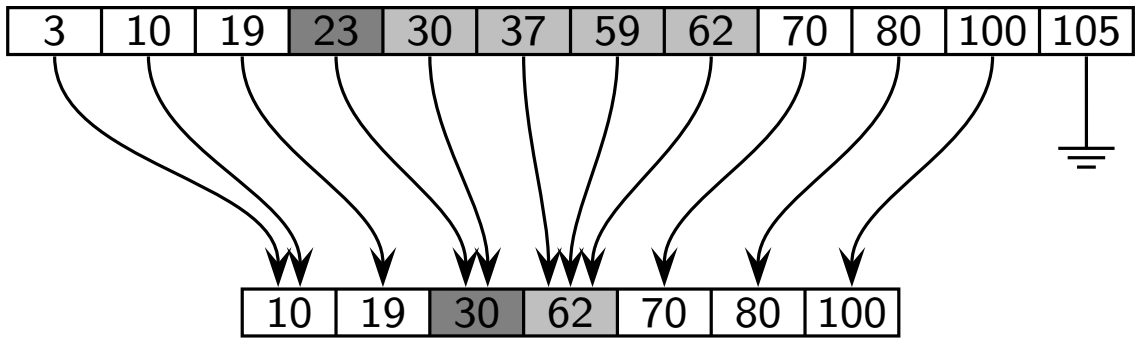
# $d$-Dimensional Range Trees

- $O(n \log^{d-1} n)$ storage
- $O(n \log^{d-1} n)$ construction time
- $O(\log^d n + k)$ query time.

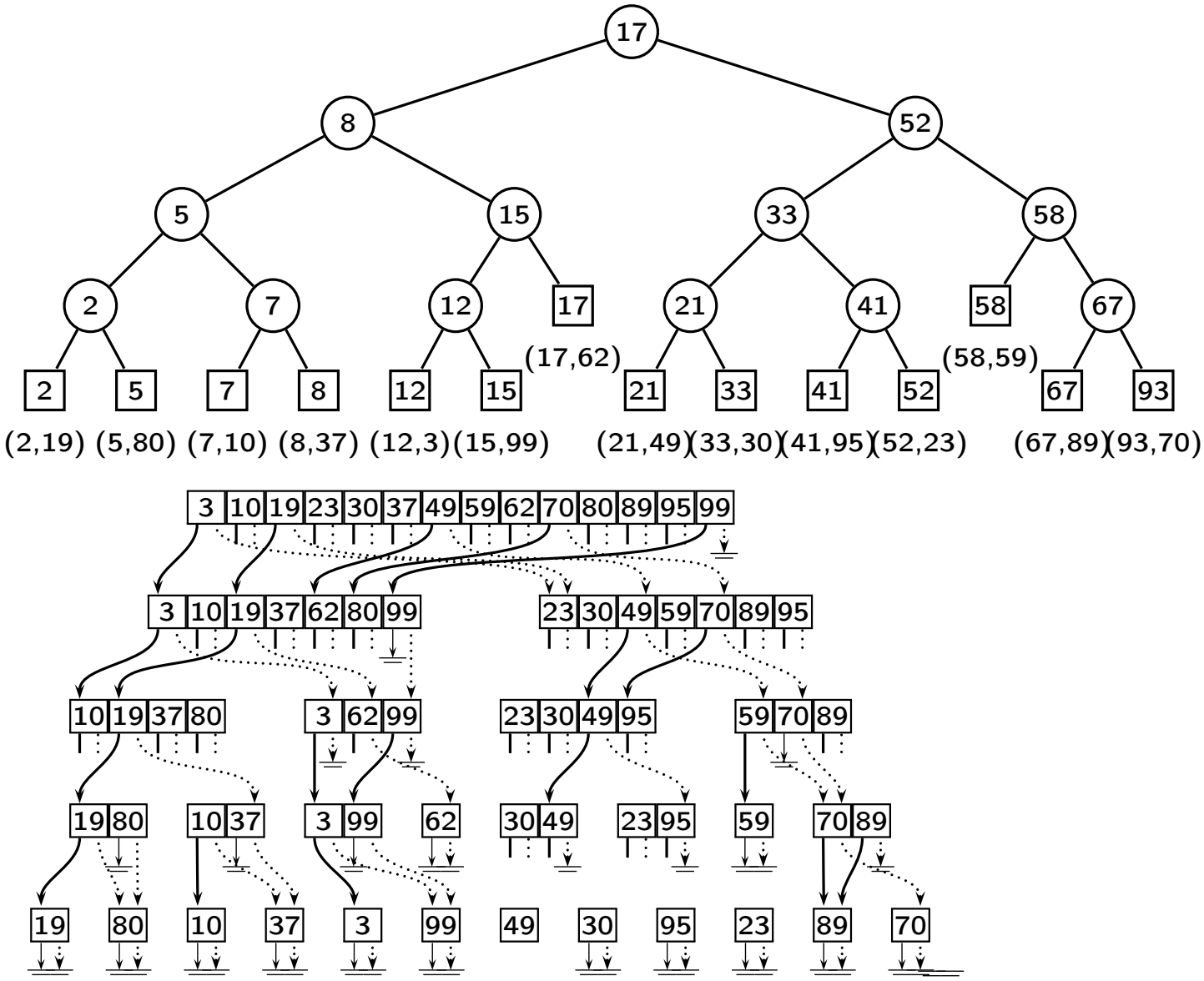



# Fractional Cascading

- $S_1$, $S_2$ — two sets of objects with real number keys
- The problem: report all objects in $S_1$ and $S_2$ whose keys lie in $[b_1, b_2]$
- The keys are ordered in arrays $A_1$ and $A_2$
- Solution: two binary searches in $A_1$ and $A_2$
- $S_2 \subseteq S_1 \Rightarrow$ the binary search in $A_2$ can be avoided




# Layered Range Trees




# Layered Range Tree Query

- $[a_1, b_1] \times [a_2, b_2]$ — the query range
- $k_v$ — the number of reported points at node $v$
- At $v_{\text{split}}$ find the entry in $A_{v_{\text{split}}}$ whose $y$-coordinate is larger than or equal to $a_2$ in $O(\log n)$ time
- For each node $v$ on the paths to $a_1$ and $b_1$ maintain the pointer from the entry in $A_{\text{parent}(v)}$ to the entry in $A_v$ whose $y$-coordinate is larger than or equal to $a_2$ in $O(1)$ time.
- Report the points of $A_v$ in $O(1 + k_v)$ time
- There are $O(\log n)$ nodes on the paths to $a_1$ and $b_1$
- Total query time becomes $O(\log n + k)$
- Fractional cascading also imporves the query time of higher-dimensional range trees by a logarithmic factor

# Range Searching Bibliography I



Mark de Berg, Mark van Kreveld, Mark H. Overmars, and Otfried Cheong.
*Computational Geometry: Algorithms and Applications*.
Springer, 3$^{rd}$ edition, 2008.


Jon Louis Bentley.
Multidimensional binary search trees used for associative searching.
*Communications of the ACM*, 18(9): 509–517, 1975.


J.H. Friedman, Jon Louis Bentley, and R.A. Finkel.
An algorithm for finding best matches in logarithmic expected time.
*ACM Transactions on Mathematical Software*, 3(3):209-226, 1977.


Jon Louis Bentley.
Multidimensional divide and conquer.
*Communications of the ACM*, 23(4):214-229, 1980.


A. Moore.
A tutorial on kd-trees.
University of Cambridge Computer Laboratory Technical Report No. 209, 1991.


David M. Mount and Sunil Arya.
ANN: A Library for Approximate Nearest Neighbor Searching.
Version 1.1.2, 2010


Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson.
Introduction to Algorithms.
2nd edition, McGraw-Hill Higher Education, 2001