Range Searching & kd-trees

Dr. Chiranjeev Kumar

ISM Dhanbad

Querying a Databases

- Databases store records or objects.
- Personnel database: Each employee has a name, id code, date of birth, function, salary, start date of employment, ...
- Fields are textual or numerical.

- **Records in a database can be transformed into points in a multi-dimensional space.**

- **The queries about the records can be transformed into queries on this set of points.**

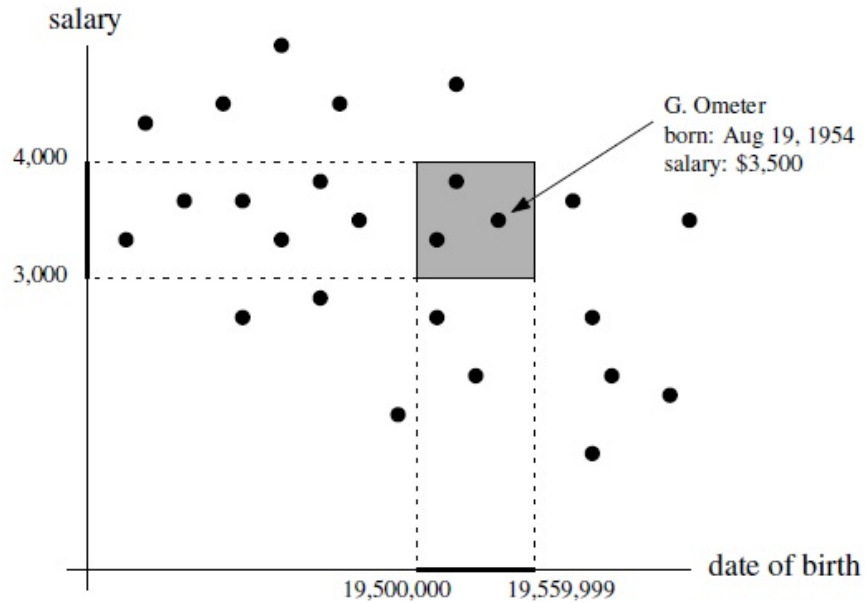
Querying a Databases (contd.)

When we see numerical fields of objects as coordinates, a database stores a point set in higher dimensions.

- **Exact match query:** Asks for the objects whose coordinates match query coordinates exactly.
- **Partial match query:** Same but not all coordinates are specified.
- **Range query:** Asks for the objects whose coordinates lie in a specified query range (interval).

Querying a Databases (contd.)

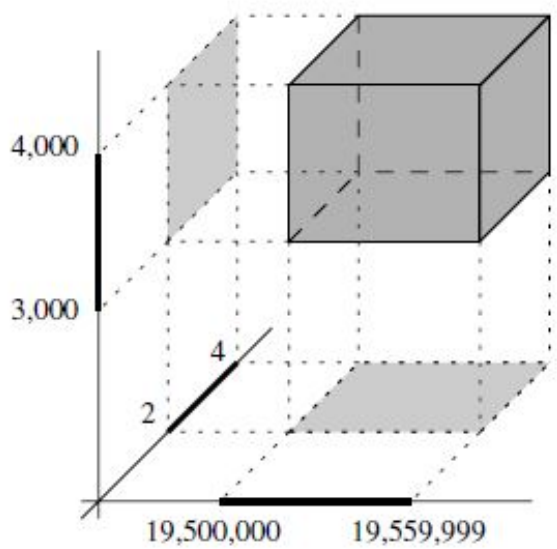
A database query may ask for all employees with age between a_1 and a_2, and salary between s_1 and s_2.



DOB is represented by 10000 * year + 100 * month + day

Example of a 3-dimensional Query:

Children in [2, 4], Salary in [3000, 4000], Date of birth in [19,500,000, 19,559,999]



Such kind of query is called a **Rectangular Range Query**, or an **Orthogonal Range Query**.

1-Dimensional Range Query Problem

1D Range Query Problem: Pre-process a set of *n* points on the real line such that the ones inside a 1-D query range (interval) can be reported fast.

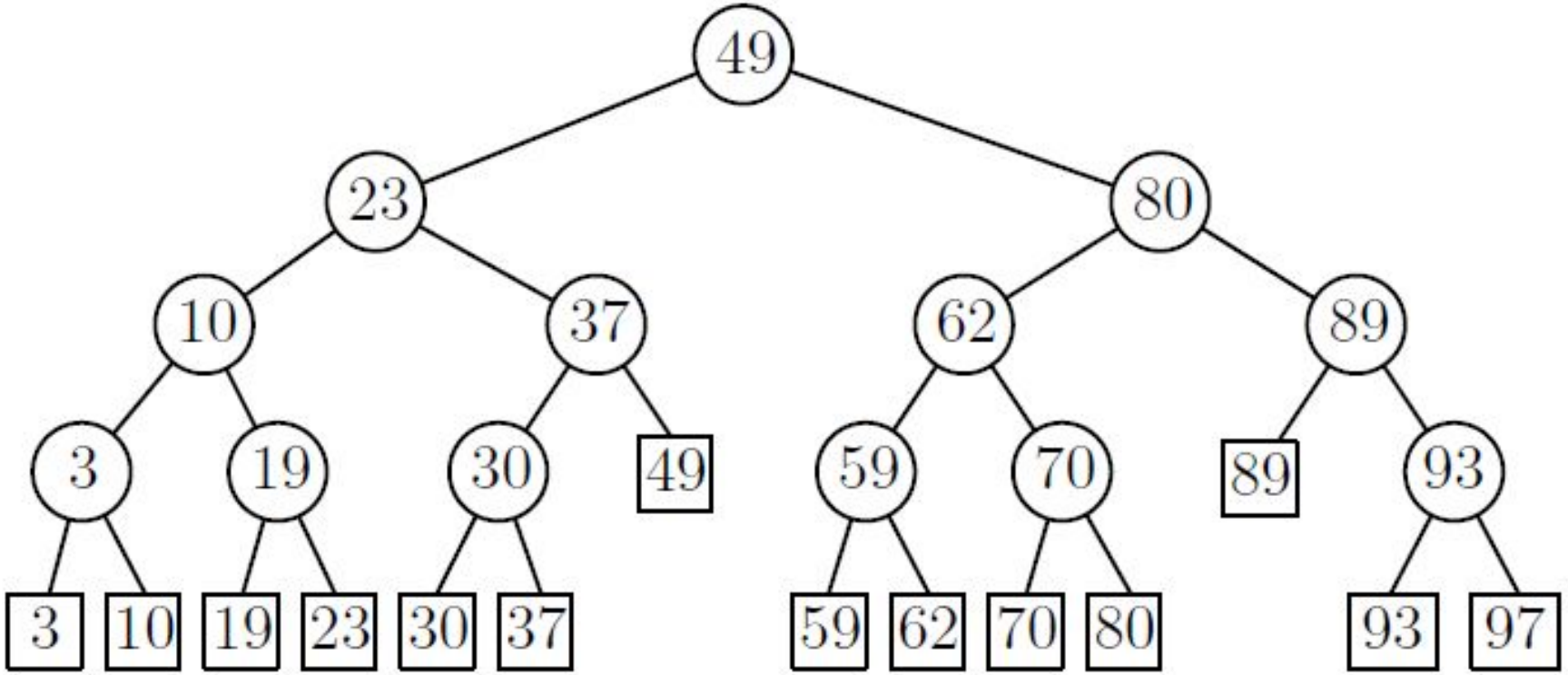
The points $p_1, \ldots, p_n$ are known beforehand, the query $[x, x^{'}]$ only later.

A "solution" to a query problem is a data structure description, a query algorithm, and a construction algorithm.

Question: What are the most important factors for the efficiency of a solution?

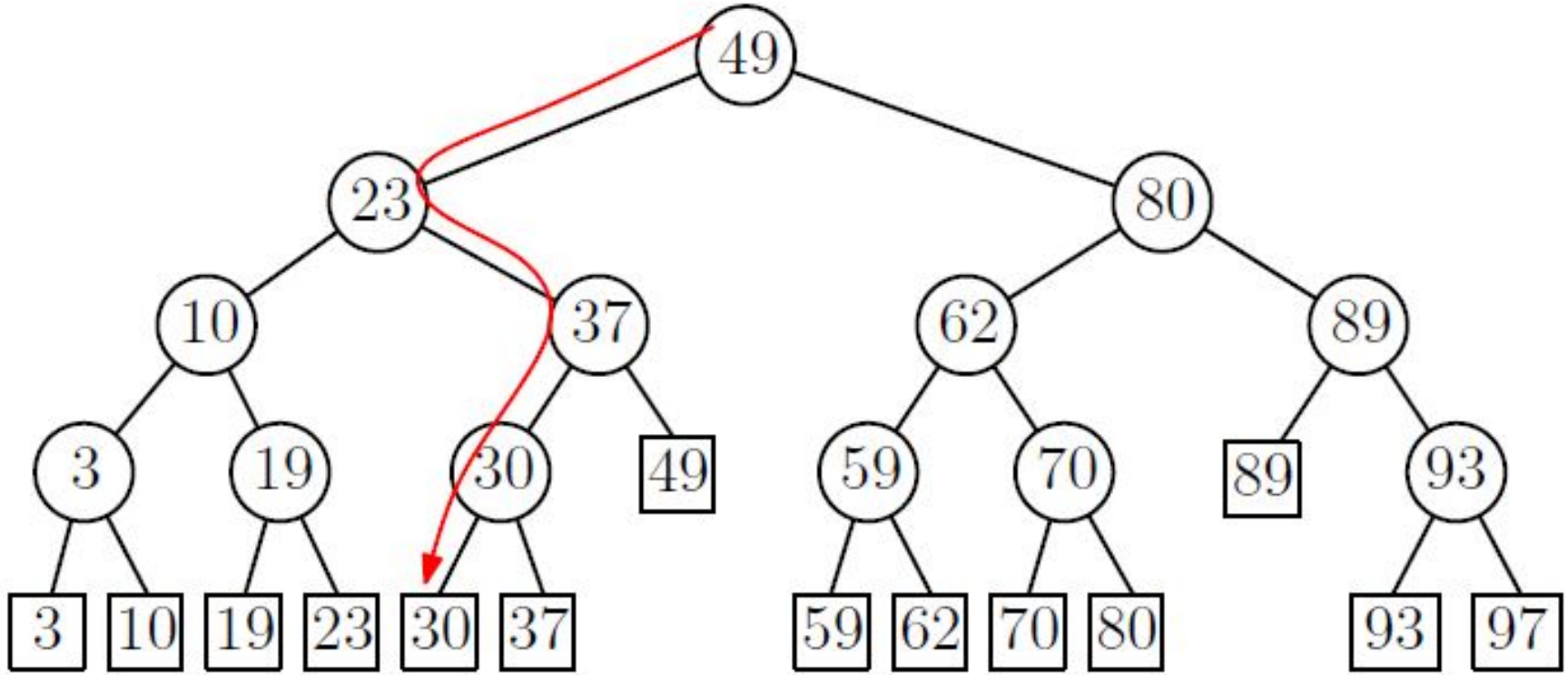
Balanced Binary Search Trees

A balanced binary search tree with the points in the leaves



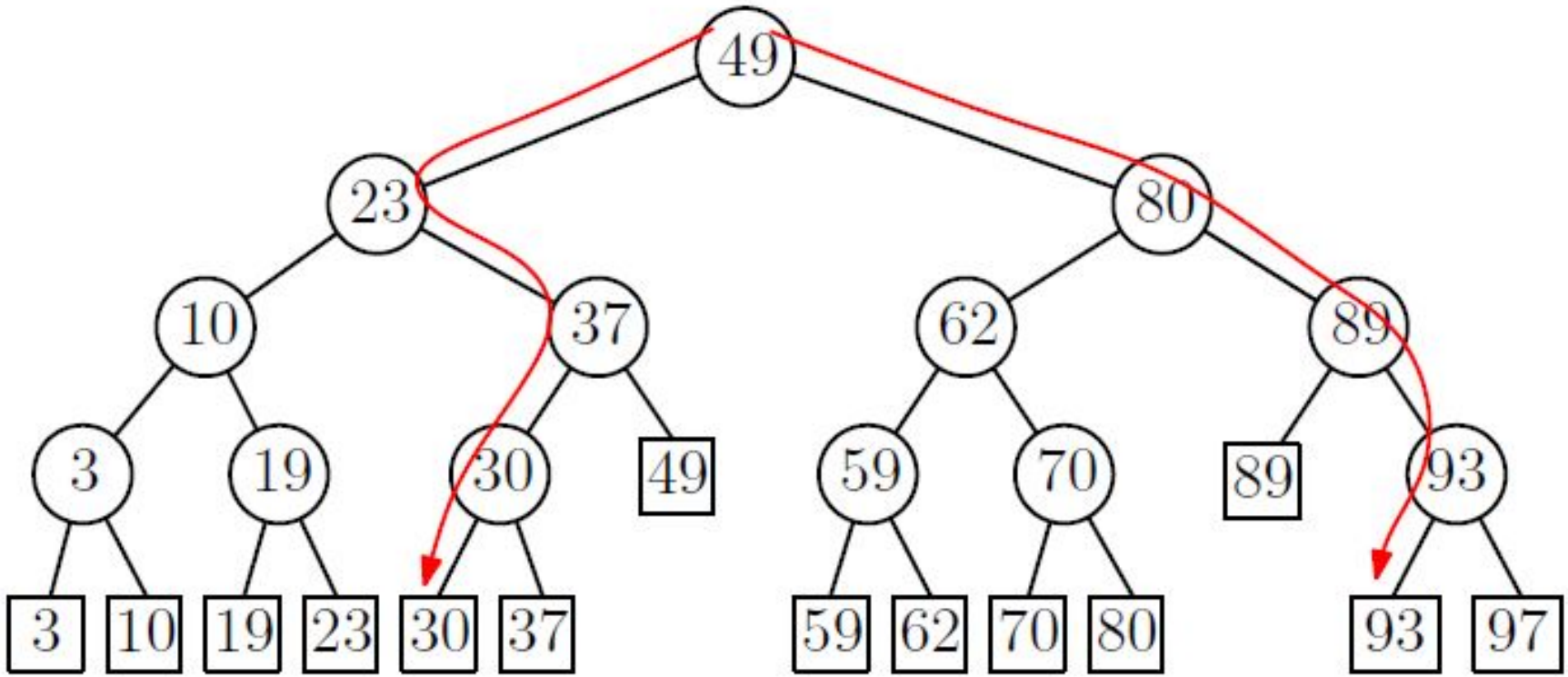
Balanced Binary Search Trees

The search path for 25



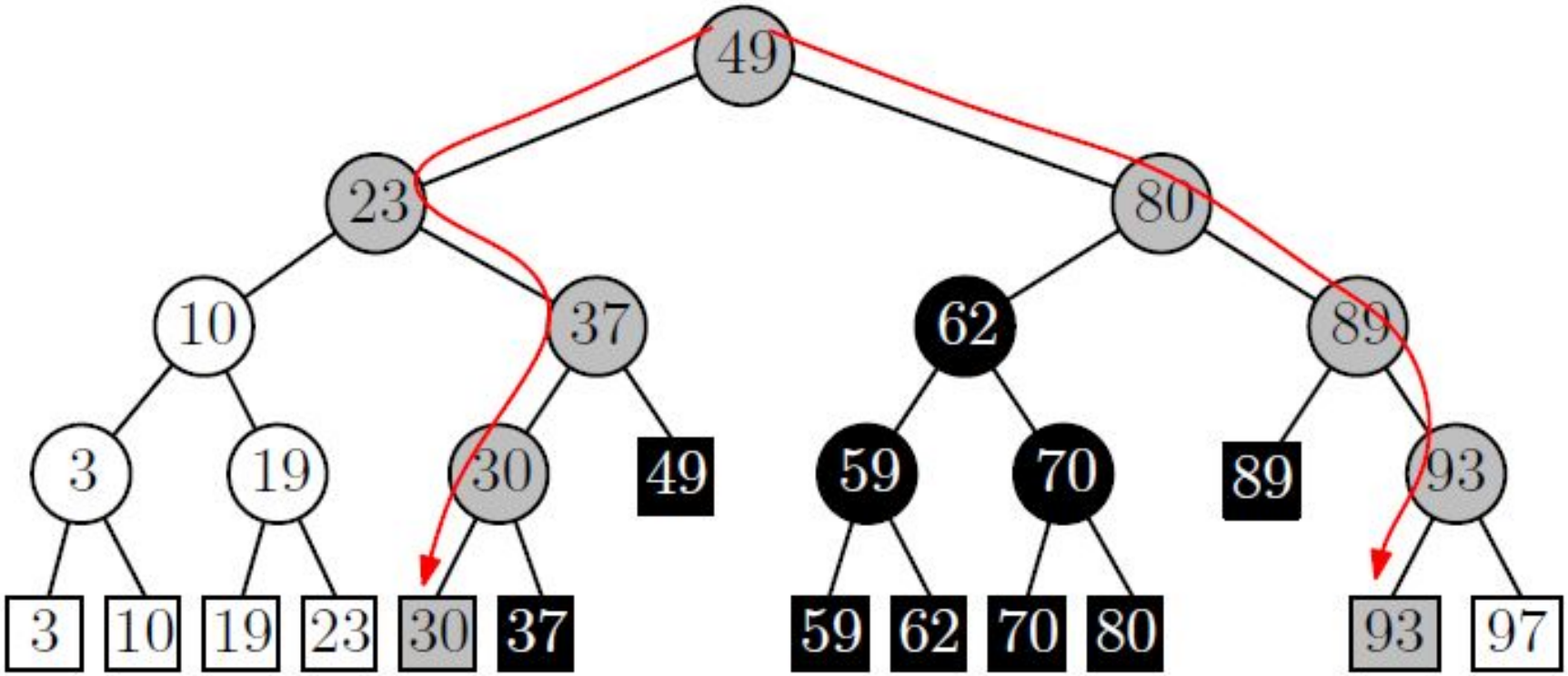
Balanced Binary Search Trees

The search paths for 25 and for 90



Example 1-D Range Query

A 1-dimensional range query with [25, 90]



Node Types for a Query

Three types of nodes for a *given* query.

1. **White nodes:** never visited by the query.

Node Types for a Query

Three types of nodes for a *given* query.

1. **White nodes:** never visited by the query.
2. **Grey nodes:** visited by the query, unclear if they lead to output.

Node Types for a Query

Three types of nodes for a *given* query.

1. **White nodes:** never visited by the query.
2. **Grey nodes:** visited by the query, unclear if they lead to output.
3. **Black nodes:** visited by the query, whole subtree is output.

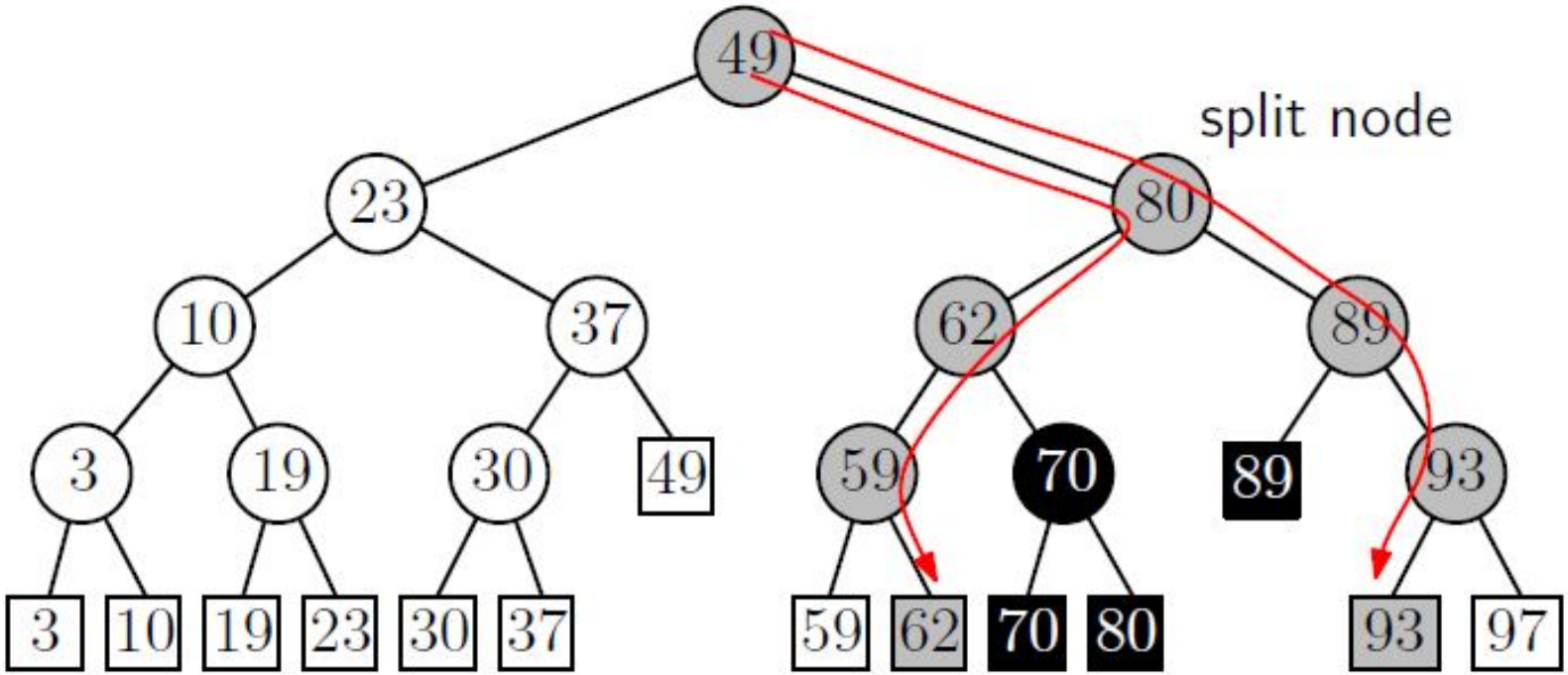
Node Types for a Query

Three types of nodes for a *given* query.

1. **White nodes:** never visited by the query.
2. **Grey nodes:** visited by the query, unclear if they lead to output.
3. **Black nodes:** visited by the query, whole subtree is output.

Example 1-D Range Query

A 1-dimensional range query with [61, 90]



Algorithm to find out the SPLIT NODE

Algorithm : $FINDSPLITNODE(\tau, x, x^{'})$

INPUT: A tree τ and two values x and $x^{'}$ with x $\leq x^{'}$)

OUTPUT: The node v where the paths to x and $x^{'}$ split, or the leaf where both paths end.

1: $v \leftarrow root(\tau)$
2: **while** v is not a leaf and $(x^{'} \leq x_v$ or $x > x_v)$ **do**
3: **IF** $x^{'} \leq x_v$
4: $v \leftarrow lc(v)$
5: **ELSE** $v \leftarrow rc(v)$
6: **end while**
7: **return** v

1-D Range Query Algorithm

Algorithm : $1DRangeQuery(\tau, [x : x^{'}])$

1: $v_{split} \leftarrow FINDSPLITNODE(\tau, x, x^{'})$
2: **if** v_{split} is a leaf **then**
3: Check if the point in v_{split} must be reported.
4: **else**
5: $v \leftarrow lc(v_{split})$
6: **while** v is not a leaf **do**
7: **if** $x \leq x_v$ **then**
8: REPORTSUBTREE $(rc(v))$
9: $v \leftarrow lc(v)$
10: **else**
11: $v \leftarrow rc(v)$
12: **end if**
13: **end while**
14: Check if the point stored in v must be reported.
15: $v \leftarrow rc(v_{split})$
16: **end if**
17: Similarly, follow the path to $x^{'}$, and ...

Query Time Analysis

The *efficiency analysis* is based on counting the numbers of nodes visited for each type

1. **White nodes:** never visited by the query; *no time spent*
2. **Grey nodes:** visited by the query, unclear if they lead to output; *time determines dependency on* n
3. **Black nodes:** visited by the query, whole subtree is output; *time determines dependency on* k*, the output size*

Query Time Analysis

Grey nodes: they occur on only two paths in the tree, and since the tree is balanced, its depth is $O(logn)$

Black nodes: a (sub)tree with m leaves has $m-1$ internal nodes; traversal visits $O(m)$ nodes and finds m points for the output

The time spent at each node is $O(1) \Longrightarrow O(logn + k)$ query time

Storage Requirement and Preprocessing

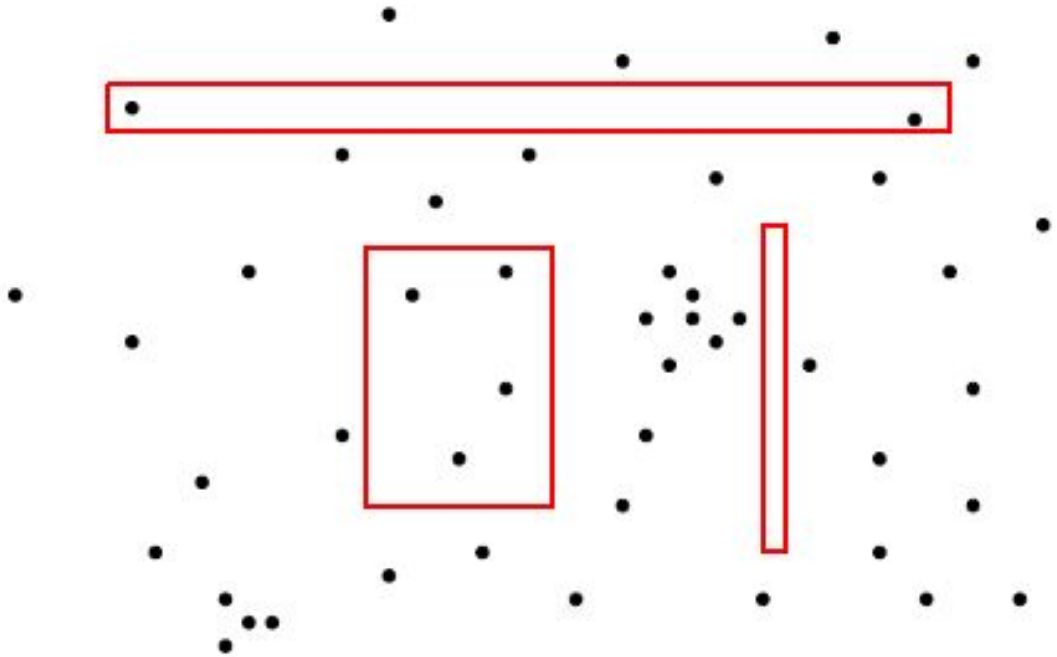
A (balanced) binary search tree storing n points uses $O(n)$ storage

A balanced binary search tree storing n points can be built in $O(n)$ time after sorting, so in $O(nlogn)$ time overall (or by repeated insertion in $O(nlogn)$ time)

Result

Theorem: A set of n points on the real line can be preprocessed in $O(nlogn)$ time into a data structure of $O(n)$ size so that any 1D range query can be answered in $O(logn + k)$ time, where k is the number of answers reported

Range Queries in 2-Dimensional



Range Queries in 2-Dimensional (contd.)

Question: Why can't we simply use a balanced binary tree in x-coordinate?

Or, use one tree on x-coordinate and one on y-coordinate, and query the one where we think querying is more efficient?

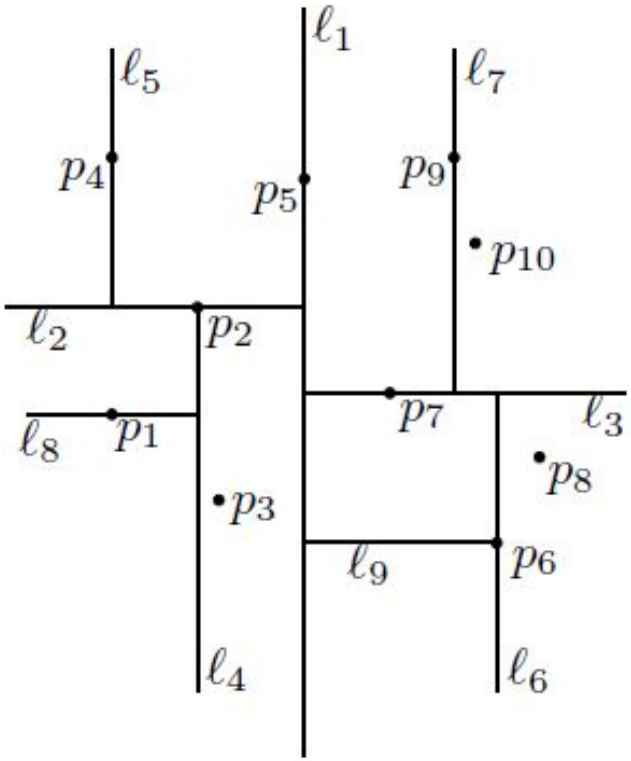
Kd-Trees

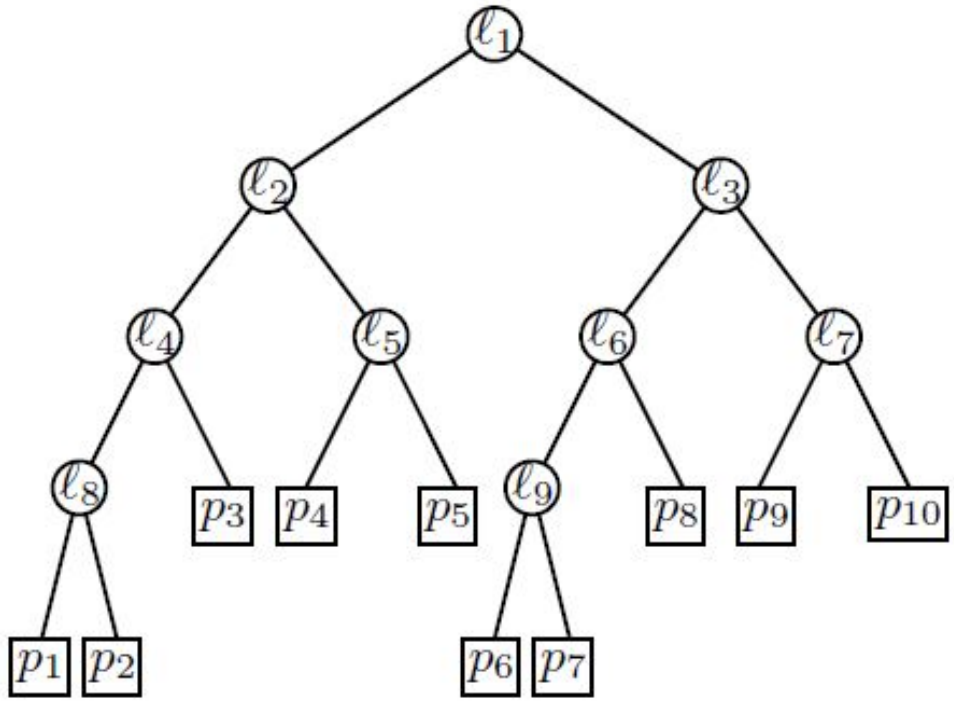
Kd-trees, the idea: Split the point set alternatingly by x-coordinate and by y-coordinate

split by x-coordinate: split by a vertical line that has half the points left and half right

split by y-coordinate: split by a horizontal line that has half the points below and half above

Kd-Trees (contd.)



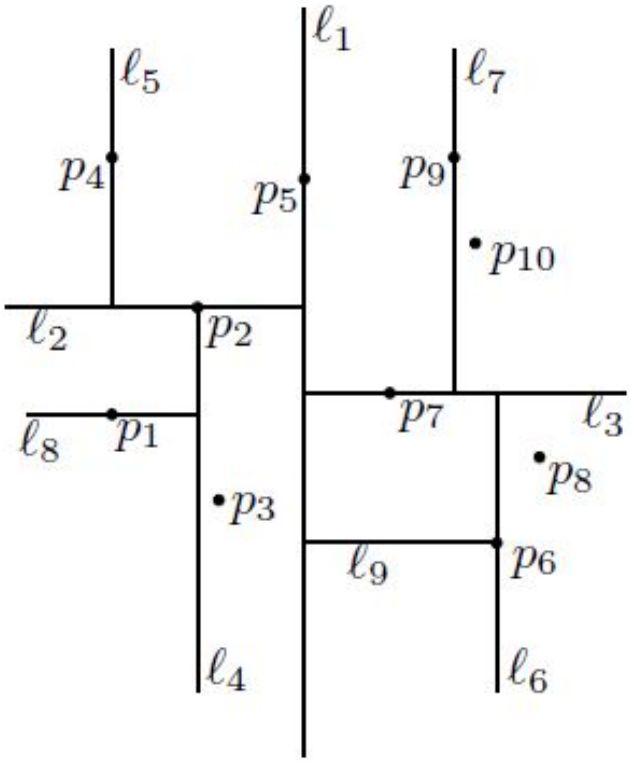


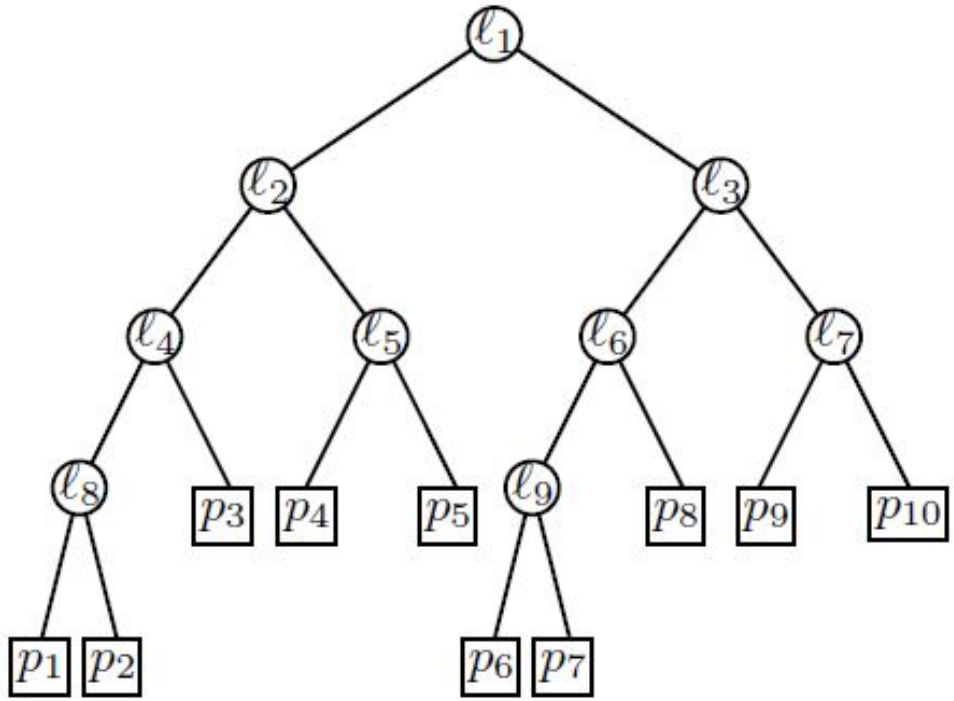
Kd-Trees

At the root, we split the set P with a vertical line l into two subsets of roughly equal size.

- The splitting line is stored at the root.
- P_{left}, the subset of the points to the left or on the splitting line, is stired in the left subtree.
- P_{right}, the subset of the right of it, is stored in the right subtree.

Kd-Trees (contd.)





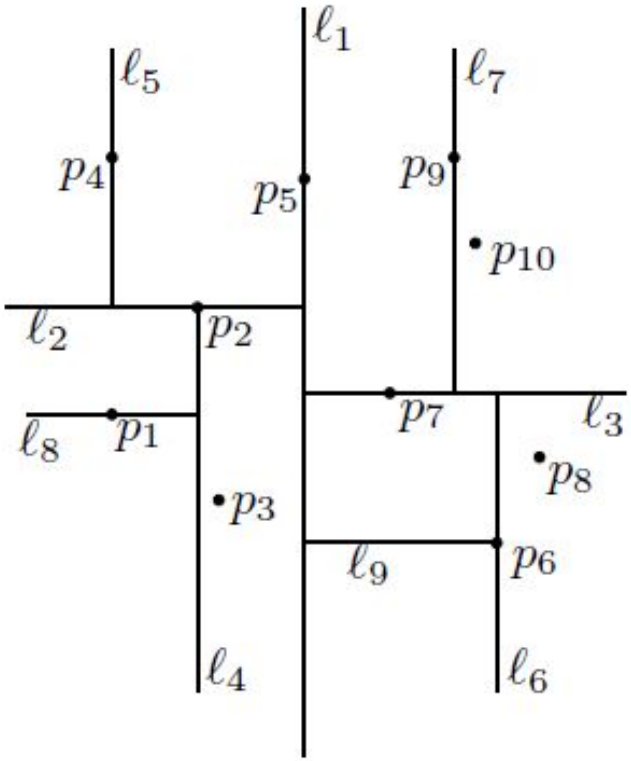
At the left child of the root we split P_{left} into two subsets with a horizontal line:

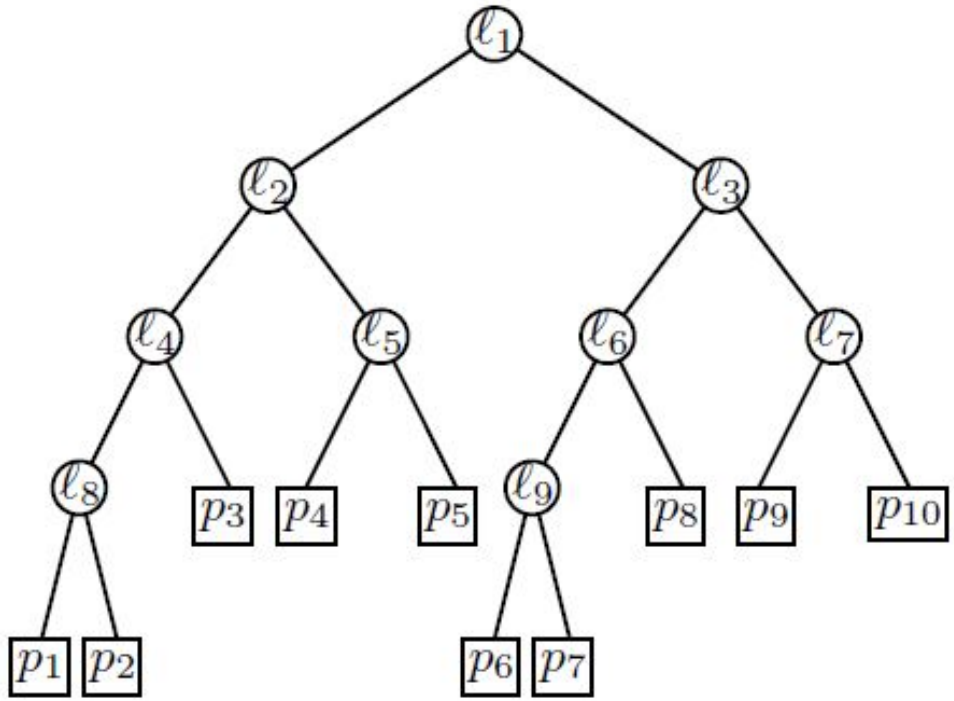
- The point below or on it are stored in the left subtree of the left child.
- The point above it are stored in the right subtree.
- The left child itself stores the splitting line.

Similar steps will be carried out for P_{right}.

In general, we split with a vertical lime at nodes whose depth is even, and we split with a horizontal line at nodes whose depth is odd.

Kd-Trees (contd.)





Kd-Tree Construction

Algorithm: BuildKdTree(P,depth)

1: **if** P contains only one point **then**
2: **return** a leaf storing this point
3: **else**
4: **if** depth is even **then**
5: Split P with a vertical line l through the median x-coordinate into P_1 (left of or on l) and P_2 (right of l)
6: **else**
7: Split P with a horizontal line l through the median y-coordinate into P_1 (below or on l) and P_2 (above l)
8: **end if**
9: **end if**
10: $v_{left} \leftarrow$ BuildKdTree$(P_1, depth + 1)$
11: $v_{right} \leftarrow$ BuildKdTree$(P_2, depth + 1)$
12: Create a node v storing l, make v_{left} the left child of v, and make v_{right} the right child of v.
13: **return** v

Kd-tree construction

The median of a set of n values can be computed in $O(n)$ time (randomized: easy; worst case: much harder)

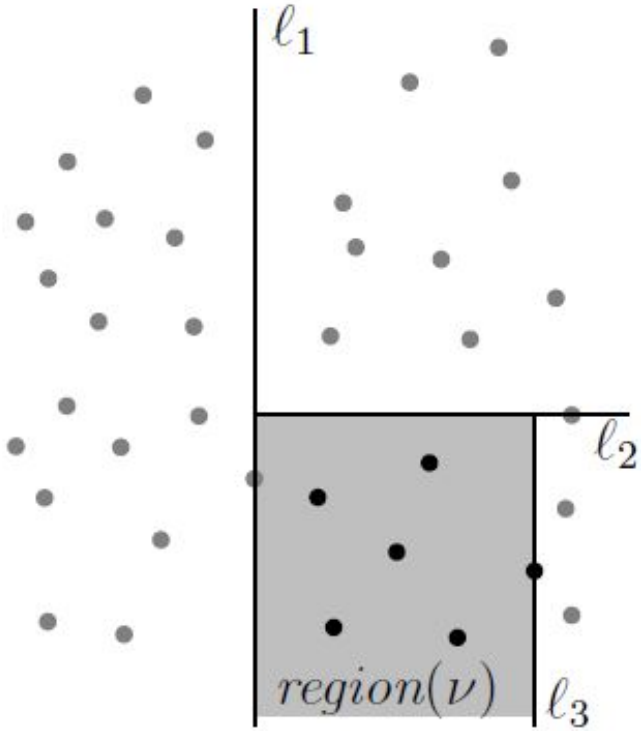
Let $T(n)$ be the time needed to build a kd-tree on n points

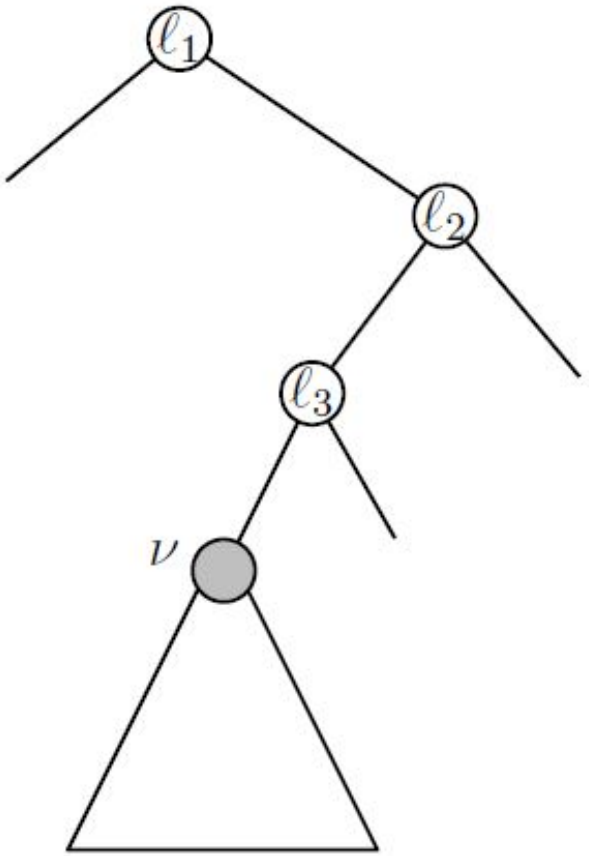
$$T(1) = O(1)$$

$$T(n) = 2.T(n/2) + O(n)$$

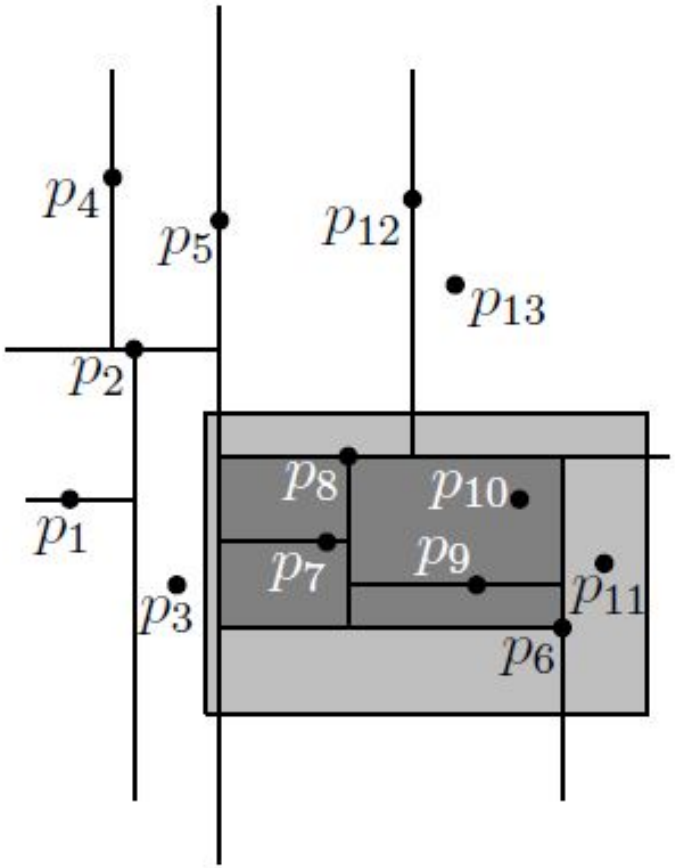
A kd-tree can be built in $O(nlogn)$ time

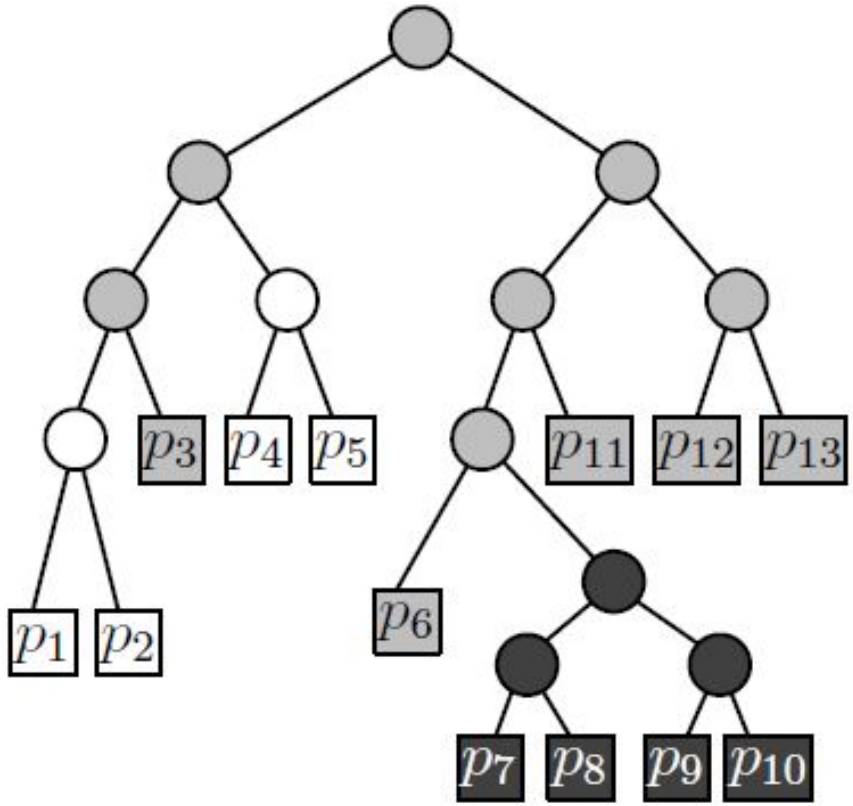
Kd-tree regions of nodes





Kd-tree Querying





Kd-tree Querying

Algorithm: SEARCHKDTree(v, R)

Input: The root of (a subtree of) a kd-tree, and a range R
Output: All points at leaves below v that lie in the range.

```
1: if v is a leaf then
2:     Report the point stored at v if it lies in R
3: else
4:     if region(lc(v)) is fully contained in R then
5:         REPORTSUBTREE(lc(v))
6:     else
7:         if region(lc(v)) intersects R then
8:             SEARCHKDTREE(lc(v), R)
9:         end if
10:        if region(rc(v)) is fully contained in R then
11:            REPORTSUBTREE(rc(v))
12:        else
13:            if region(rc(v)) intersects R then
14:                SEARCHKDTREE(rc(v), R)
15:            end if
16:        end if
17:    end if
18: end if
```

Thank You