# Geometric Range Searching
# Kinetic Data Structures
# Clustering Mobile Nodes

Leonidas J. Guibas
Stanford University

# Geometric Range Searching

- Database

| employee |
|---|
| • age<br>• salary<br>• start date<br>• city address |

Database Record

1. Suppose we want to know all employees with Salary $\in$ [40K, 50K]

➔ Scan records & pick out those in range *slow*

Adapted from N. Amato

# Motivation

- Database

| employee |
| --- |
| • age<br>• salary<br>• start date<br>• city address |

Database Record

2. Suppose we want to know all employees with Salary $\in$ [40K, 50K] AND Age $\in$ [25, 40]

   ➔ Scan records & check each one *slow*

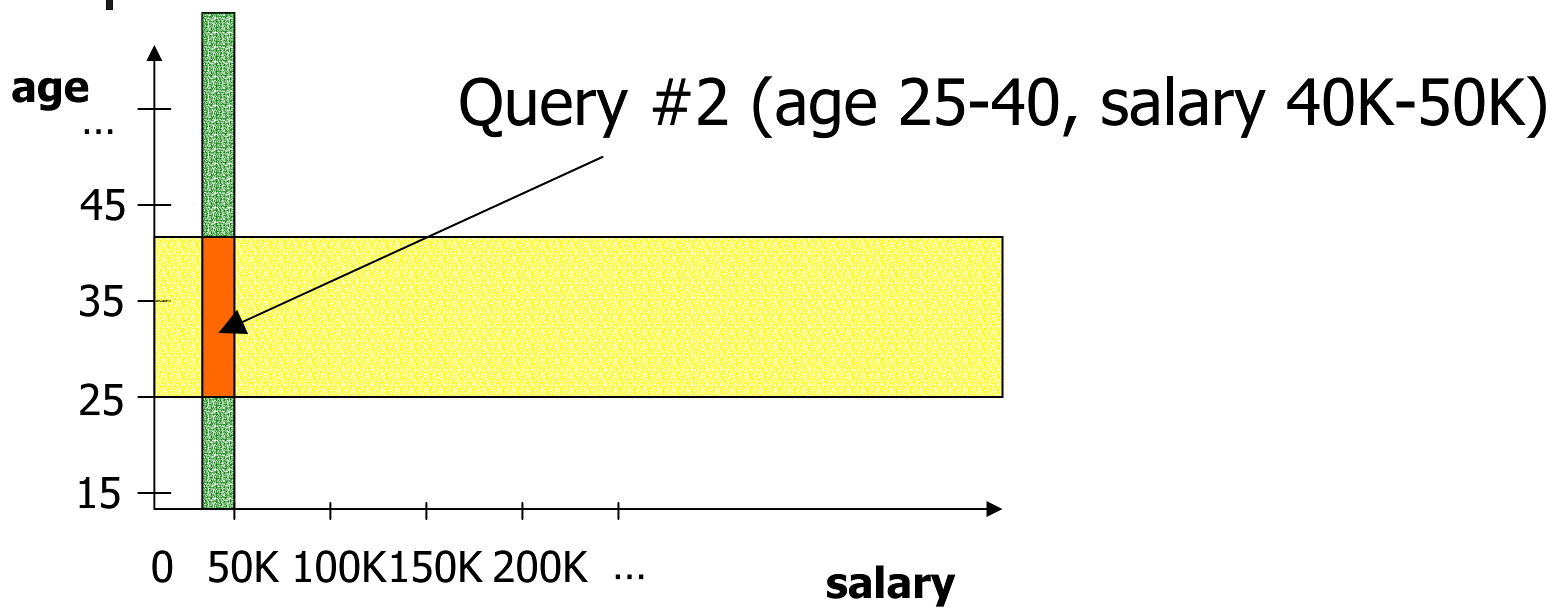
# Motivation, Cnt'd.

- Alternative : View each employee is a point in space

4D

  - age range [15, 75]
  - salary range [0K, 500K]
  - start date [1/1/1900, today]
  - city/address [College Station, Bryan, Austin, …]

# Motivation, Cnt'd.




Orthogonal Range Query (Rectangular)

- Want all points in the orthogonal range
- Faster than linear scan, if good data structures are used.
- Query time O( f(n)+k ) ; where k= # of points reported

# Range searching desiderata

Because may queries will be made, it pays to preprocess the data and build an index. We desire:

- Low index storage cost
- Fast index construction
- Low query overhead [f(n)]
- Reasonably efficient dynamic DB modifications (insertions/deletions)

# 1-D Range Searching

- Data:Points $P=\{p_1, p_2, \ldots p_n\}$ in 1-D space (set of real numbers)
- Query: Which points are in 1-D query rectangle (in interval [x, x'])

Data structure 1: Sorted Array

- A=

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 9 | 27 | 28 | 29 | 98 | 141 | 187 | 200 | 201 | 202 | 999 |

- Query: Search for x & x' in A by binary search O(logn)
  Output all points between them. O(k)
  Total O(logn+k)
- Update: Hard to insert points. Add point p', locate it n A by binary search. Shift elements in A to make room. O(n) on average
- Storage Cost: O(n)
- Construction Cost: O(nlogn)

# 1-D Range Searching Ctnd.

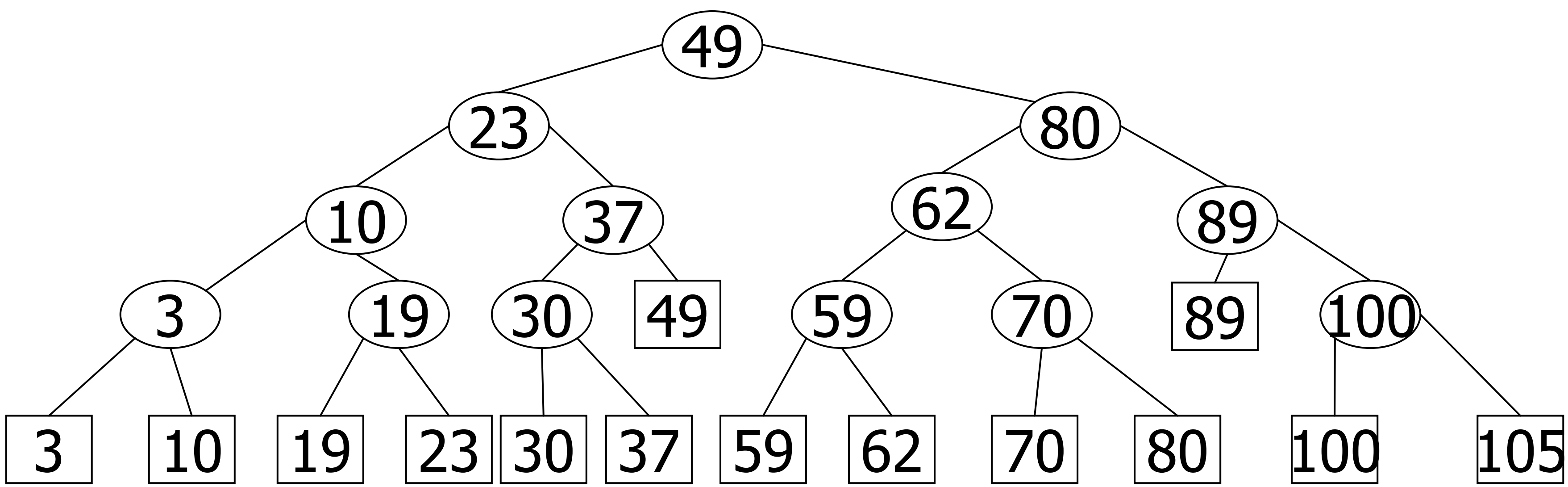
Data structure 2: Balanced Binary Search Tree

- Leaves store points in P (in left to right order)
- Internal nodes are splitting values. $x_V$ used to guide search.
  - Left sub tree of V contains all values $\leq x_V$
  - Right sub tree of V contains all values $> x_V$
- Query: [x, x′]
  - Locate x & x′ in T (search ends at leaves u & u′)
  - Points we want are located in leaves
    - In between u & u′
    - Possibly in u (if $x=u_V$)
    - Possibly in u′ (if $x=u'_V$)

Leaves of sub trees rooted at nodes V s.t. parent (v) is on search path root to u (or root to u′)

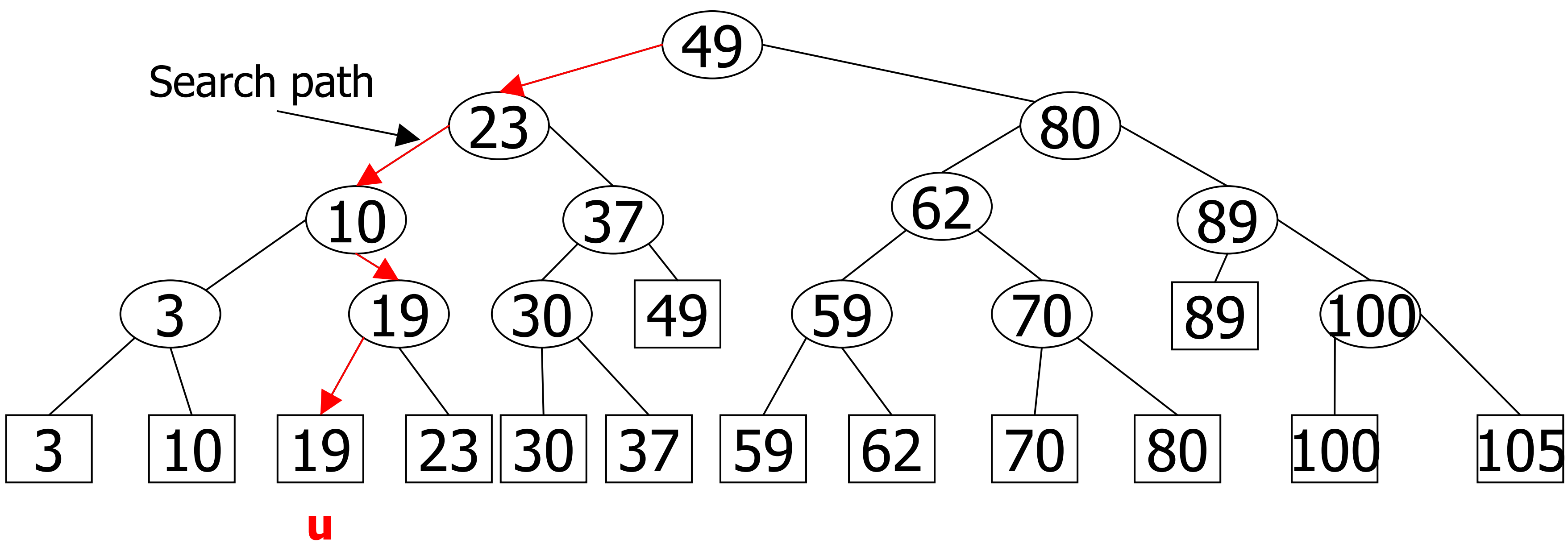
# 1-D Range Searching Ctnd.

- Look for node $V_{split}$ where search paths for x & x′ split
  - Report all values in right sub tree on search path for x′
  - Report all values in left sub tree on search path for x
- Query: [18:77]



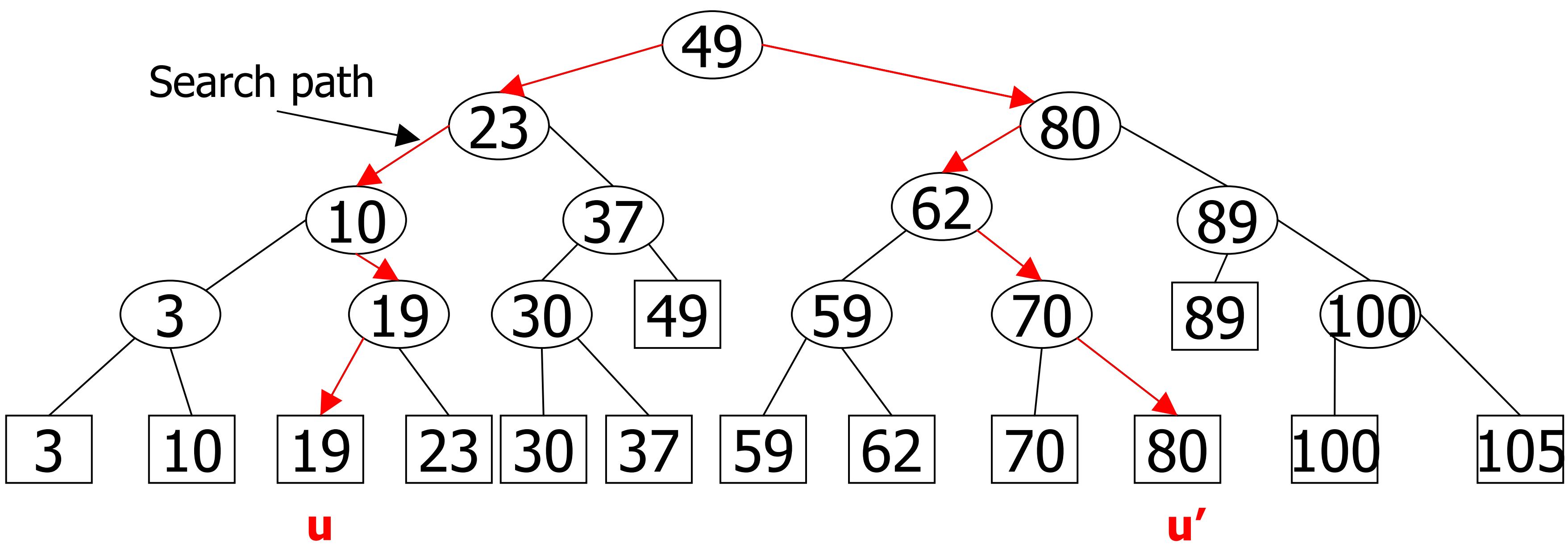
# 1-D Range Searching Ctnd.

- Look for node $V_{split}$ where search paths for x & x′ split
  - Report all values in right sub tree on search path for x′
  - Report all values in left sub tree on search path for x
- Query: [18:77]



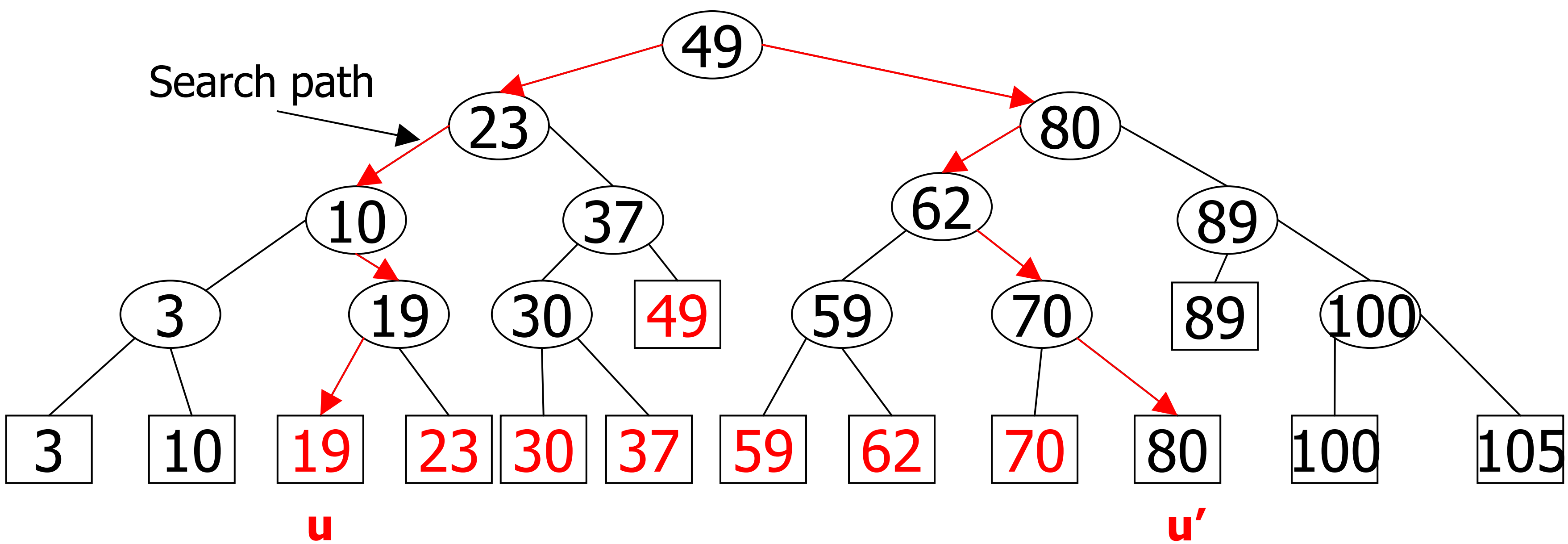
# 1-D Range Searching Ctnd.

- Look for node $V_{split}$ where search paths for x & x′ split
  - Report all values in right sub tree on search path for x′
  - Report all values in left sub tree on search path for x
- Query: [18:77]



# 1-D Range Searching Ctnd.

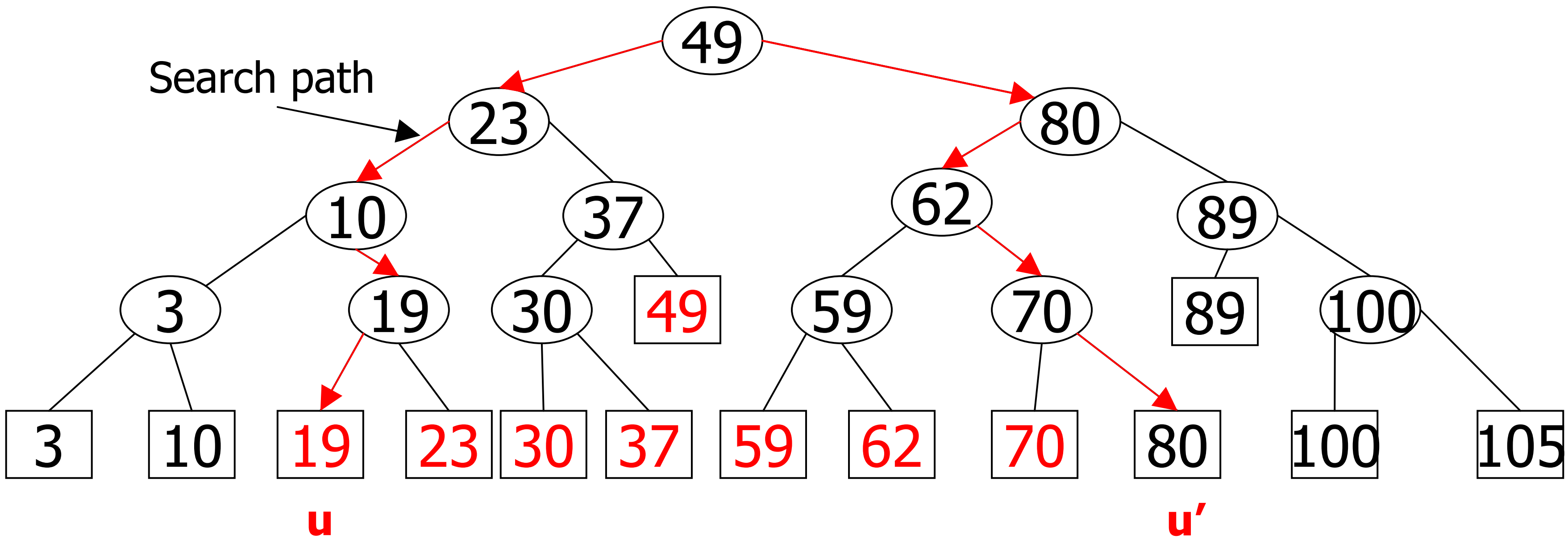
- Look for node $V_{split}$ where search paths for x & x′ split
  - Report all values in right sub tree on search path for x′
  - Report all values in left sub tree on search path for x
- Query: [18:77]



# 1-D Range Searching Ctnd.

- Update Cost O(logn)
- Storage Cost O(n)
- Construction Cost O(nlogn)

Query Overhead O(logn)




1-D Range Tree

# Key ideas

Canonical subsets

- Pre-store the answer to certain queries, in a hierarchical fashion
  - the binary tree defines canonical intervals
- Assemble the answer to the an actual query by combining answers to pre-stored queries
  - any other interval is the disjoint union of canonical intervals
- How many answers to canonical sub-problems do we pre-store? Storage vs. query-time trade-off.

# KD-Trees

(Higher dimensional generalization of 1D-Range Tree.)

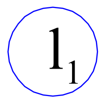
- e.g. for 2-dimensions

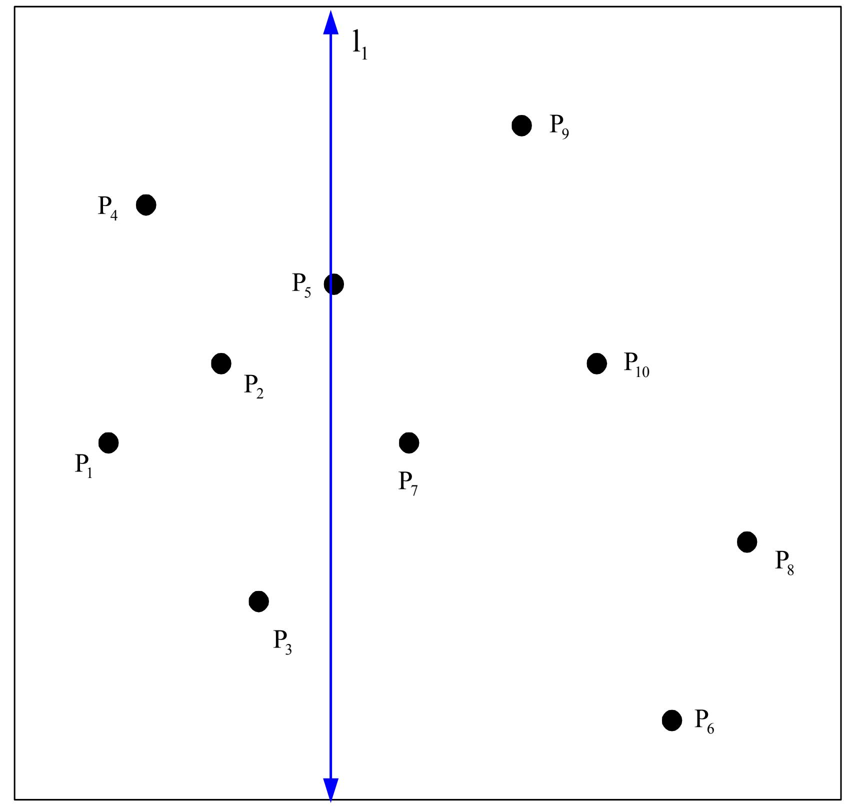
  idea:first split on x-coord (even levels)
  next split on y-coord (odd levels)
  repeat

  levels : store pts

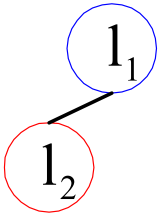
  internal nodes : spilitting lines (as opposed to values)

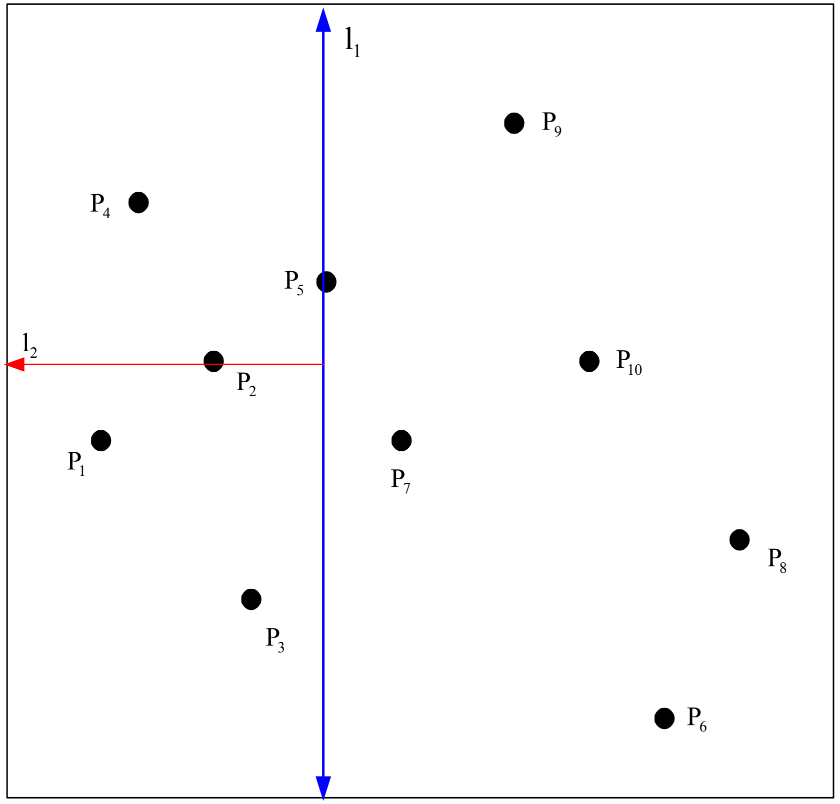
# Build KD-Tree



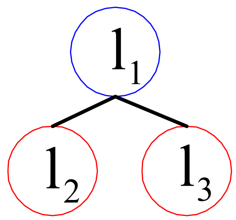



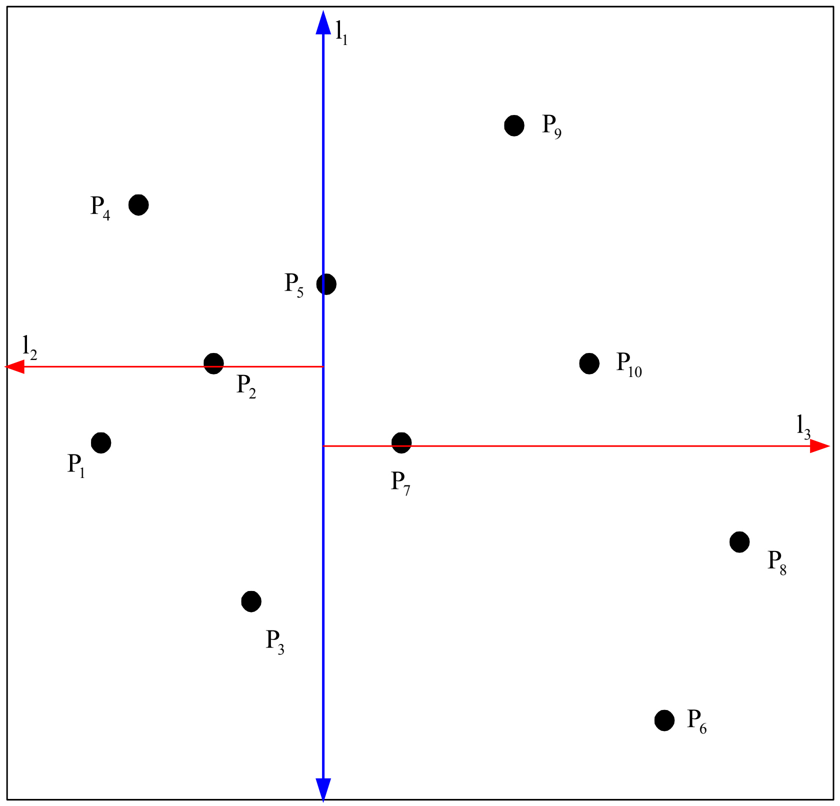
# Build KD-Tree



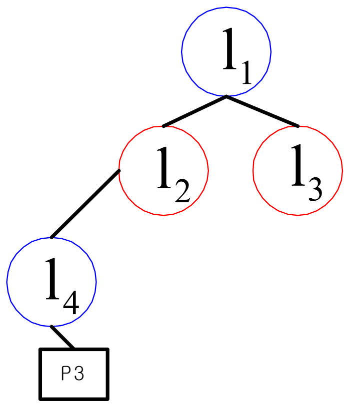



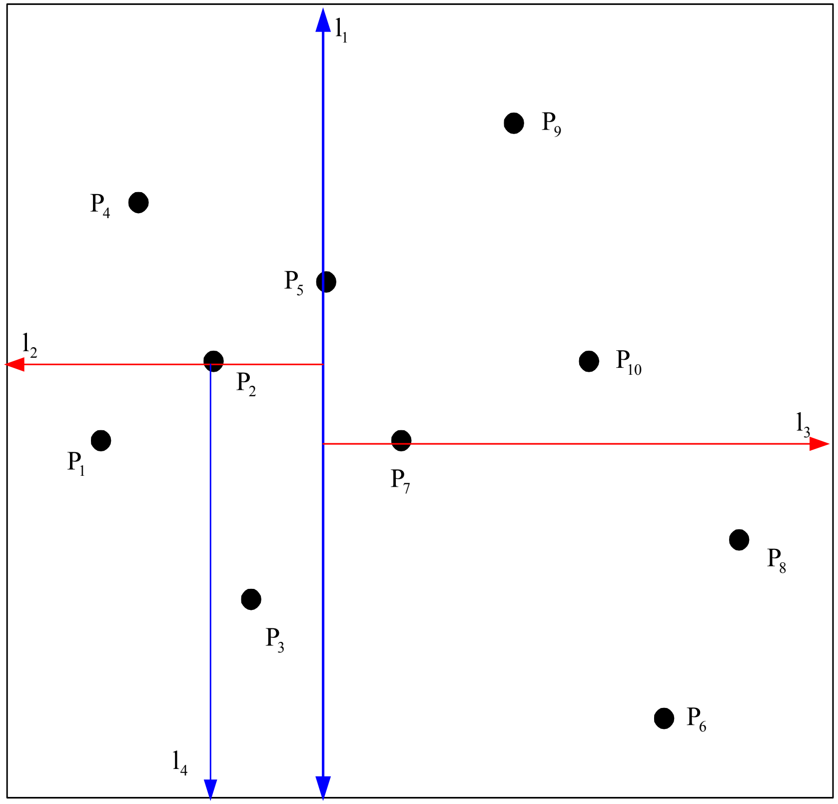
# Build KD-Tree



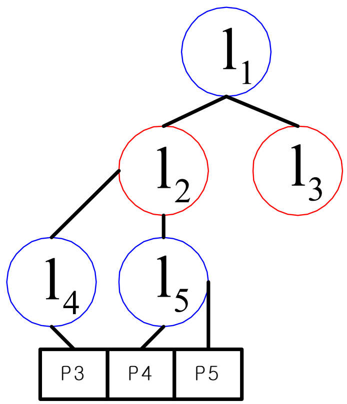



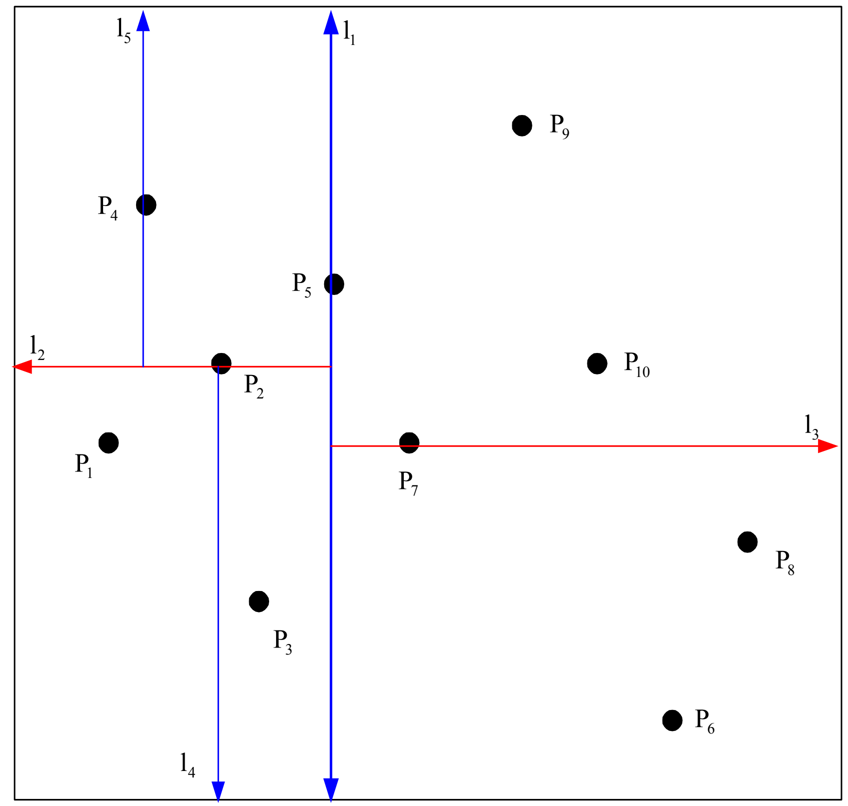
# Build KD-Tree



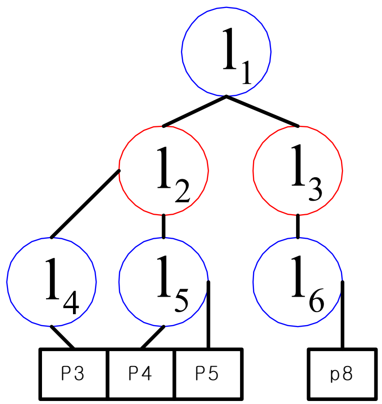



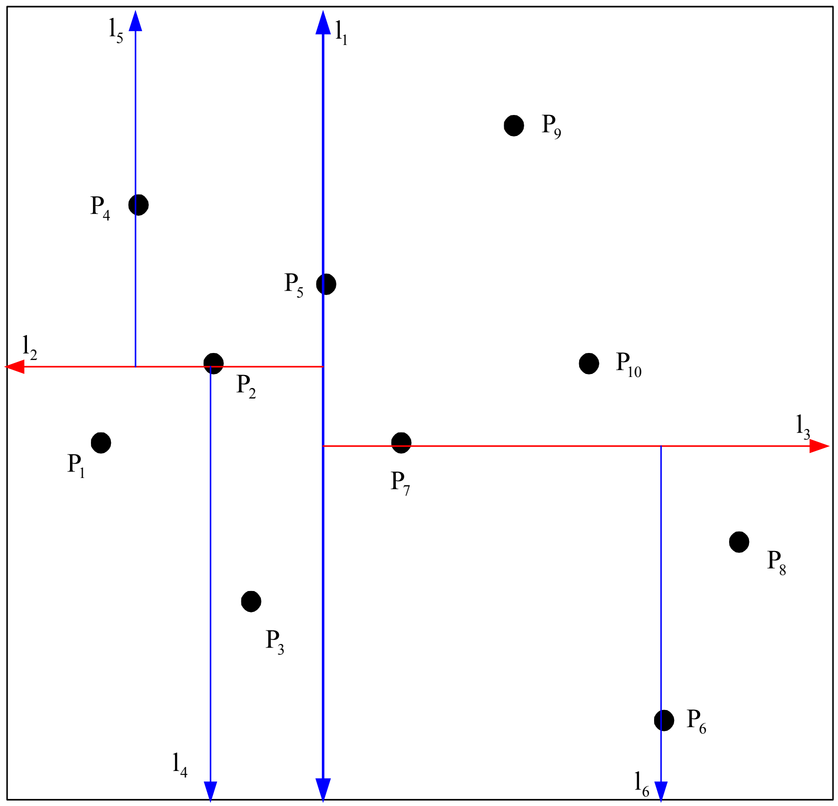
# Build KD-Tree



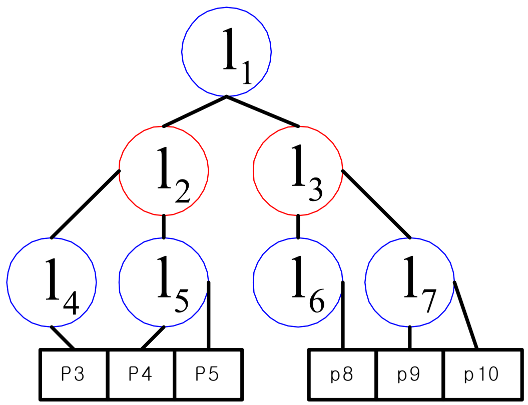



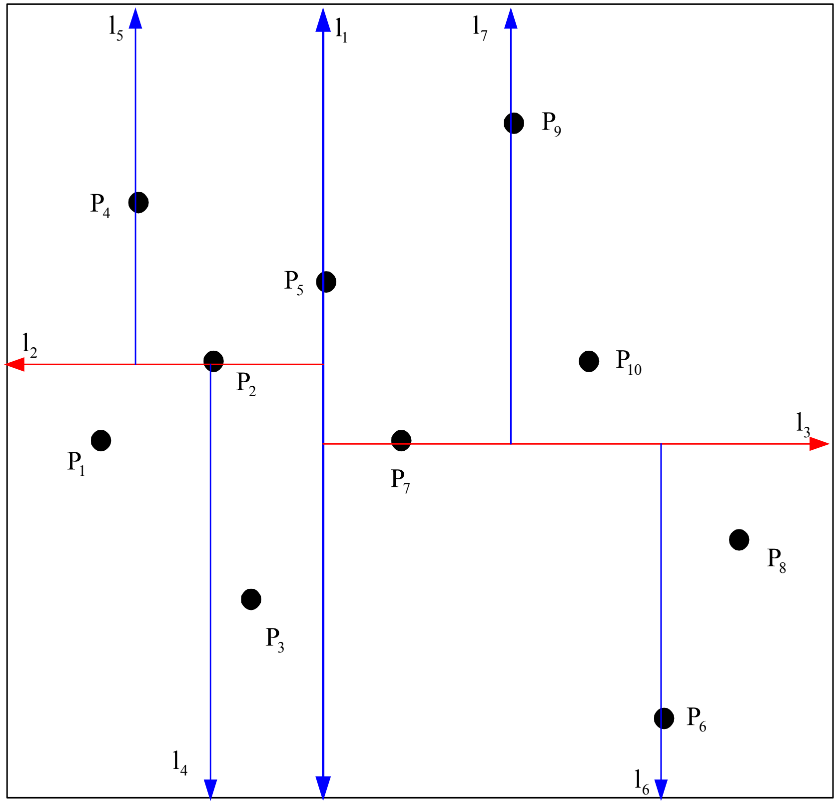
# Build KD-Tree



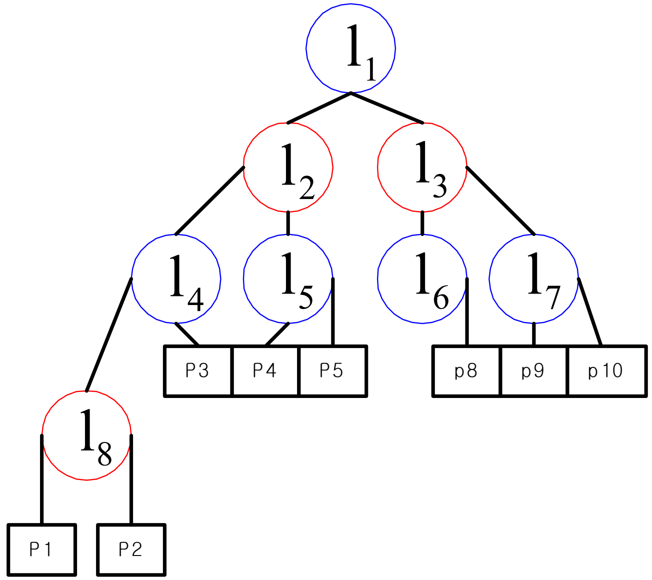



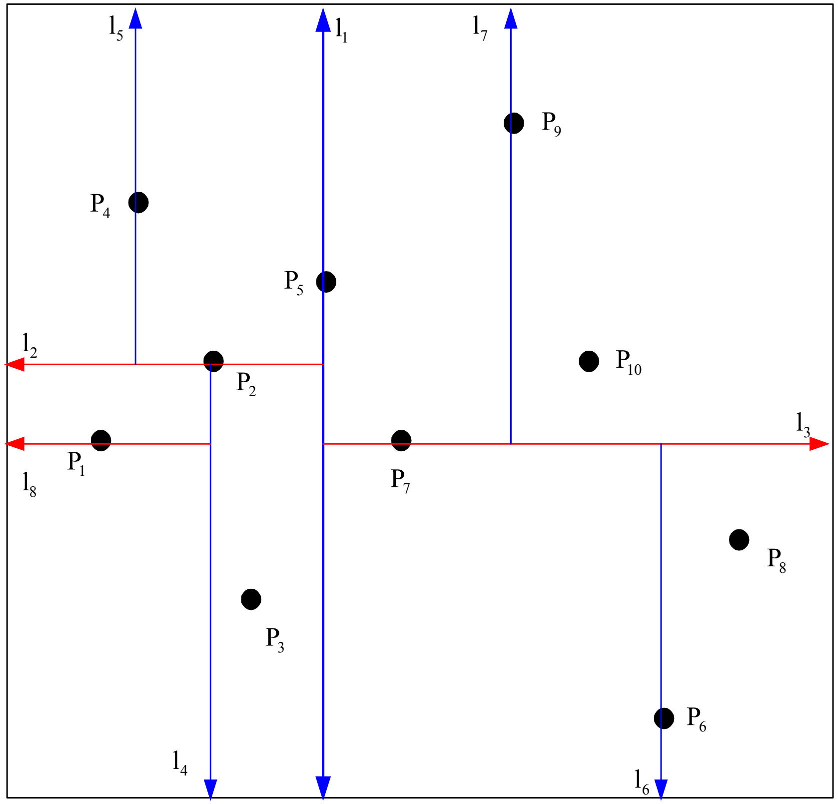
# Build KD-Tree



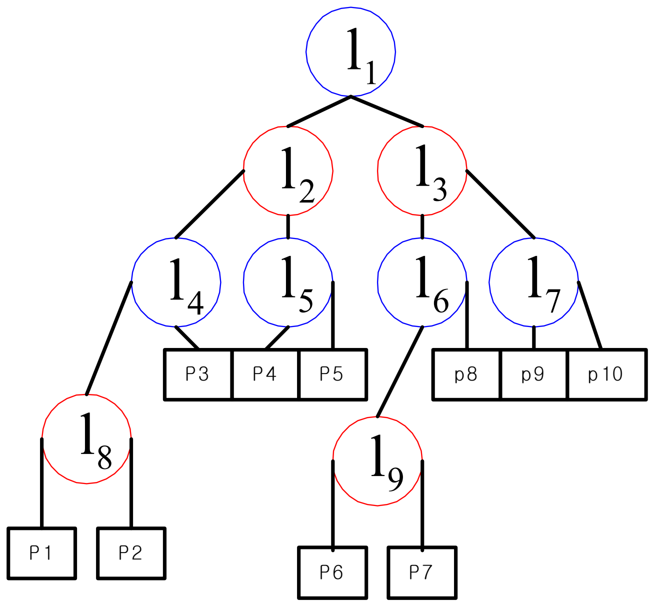



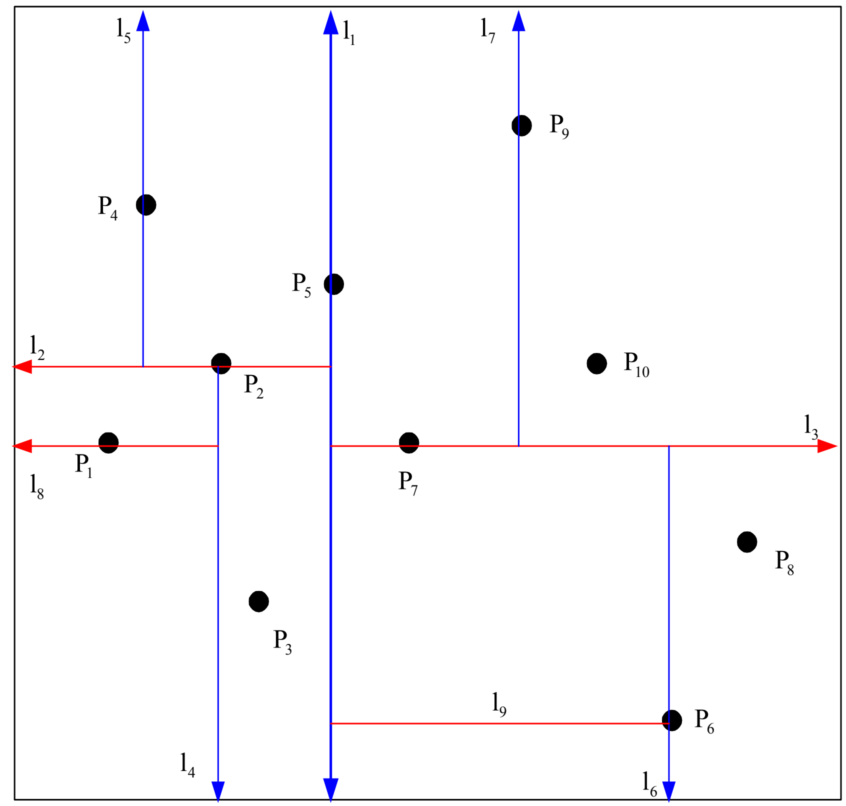
# Build KD-Tree



# Build KD-Tree



# Complexity

## Construction time

- Expensive operation: determining splitting line (median finding)
  - Can use linear time median finding algorithm ➔ O(n log n) time.

$$T(n) = O(n) + 2T(\frac{n}{2}) = O(n \log n)$$

  - but can obtain this time without fancy median finding
    ➔ Presort points by x-coord and by y-coord  (O(nlogn))

    Each time find median in O(1) and partition lists and update x and y ordering by scan in O(n) time

$$T(n) = O(n) + 2T(\frac{n}{2}) = O(n \log n)$$

# Complexity

## Storage

Number of leaves = n (one per point)

Still binary tree ➔ O(n) storage total

## Queries

- each node corresponds to a region in plane
- Need only search nodes whose region intersects query region
- Report all points in subtrees whose regions contained in query range
- When reach leaf, check if point in query region

# Algorithm: Search KD-Tree (v, R)

- Input: root of a subtree of a KD-tree and a range R
  Output: All points at leaves below v that lie in the range

```
If (v = leaf)
  then report v's point if in R
  else if (region (lc(v)) fully contained in R)
    then ReportSubtree (Rc(v))
    else if (region (lc(v)) intersects R)
      then SearchKdTree(lc(v), R)
    if (region(rc(v)) fully contained in R)
      then ReportSubtree(rc(v))
      else if (region(rc(v)) intersects R)
        then SearchKdtree(rc(v), R)
Endif
```

Note: need to know region(v)

- can precompute and store

- Computer during recursive calls, e.g.,

$$\text{region}(\text{lc}(\text{v}) = \text{region}(\text{v}) \cap \text{l}(\text{v})^{\text{left}}$$

L(v) is v's splitting line and $\text{l}(\text{v})^{\text{left}}$ is left halfpland of l(v)

# Query time

**Lemma** *A query with an axis parallel rectangle in a Kd-tree storing* $n$ *points can be performed in* $O(\sqrt{n}+k)$ *time where* $k$ *is the number of reported points.*

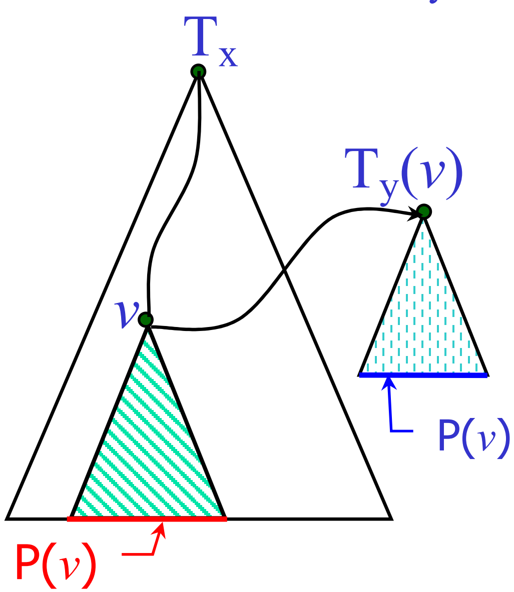
# Query time: Generalization to Higher dimensions

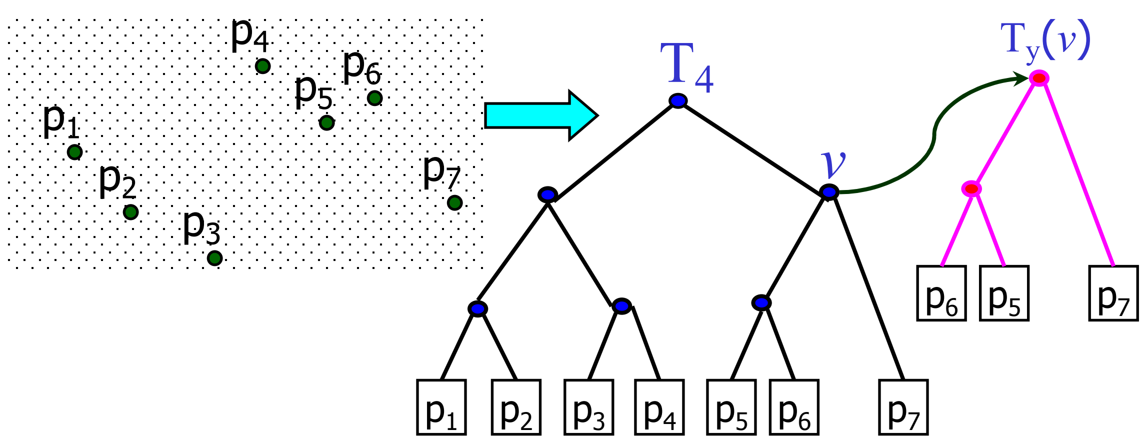
| | |
|---|---|
| Construction is similar: | one level for each dimension |
| Storage: | $O(d \cdot n)$ |
| Time: | $O(d \cdot n \log n)$ |
| Query time: | $O(n^{1-1/d}+k)$ |

# Range trees

- For each internal node $v$ $T_x$ let P($v$) be set of points stored in leaves of subtree rooted at $v$.

Set P($v$) is stored with $v$ as another balanced binary search tree $T_y(v)$ (second level tree) on y-coordinate. (have pointer from $v$ to $T_y(v)$)



# Range trees

Lemma: A 2D-range tree with n points uses O(n log n) storage

Lemma: A query with axis-parallel rectangle in range tree for n points takes $O(\log^2 n + k)$ time, where k = # reported points

# Higher Dimensional Range Trees

- 1st level tree is balanced binary search tree on 1st coordinate
- 2nd level tree is (d-1) dimensional range tree for P(v)
- \- restricted to last (d-1)-coordinates of points

  \- this tree constructed recursively

 \- last tree is 1D balanced binary search tree on $d^{th}$ - coordinates