BBM371- Data Management

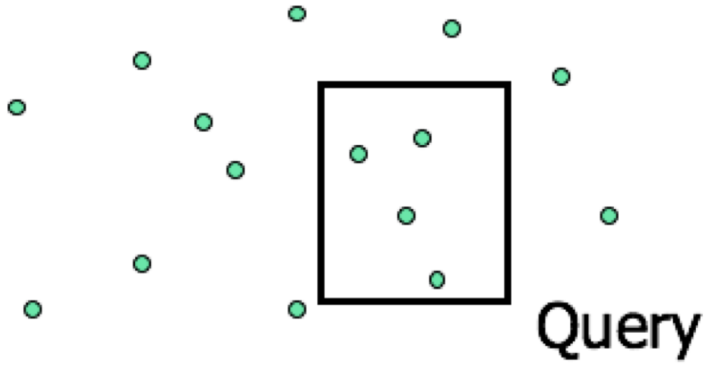
Spatial Data Management

Spatial and Geographic Databases

- Spatial databases store information related to spatial locations, and support efficient storage, indexing and querying of spatial data.
- Special purpose index structures are important for accessing spatial data, and for processing spatial queries.
- Computer Aided Design (CAD) databases store design information about how objects are constructed E.g.: designs of buildings, aircraft, layouts of integrated-circuits
- Geographic databases store geographic information (e.g., maps): often called geographic information systems or GIS.

The Problem

- Given a point set and a rectangular query, find the points enclosed in the query
- We allow insertions/deletions online

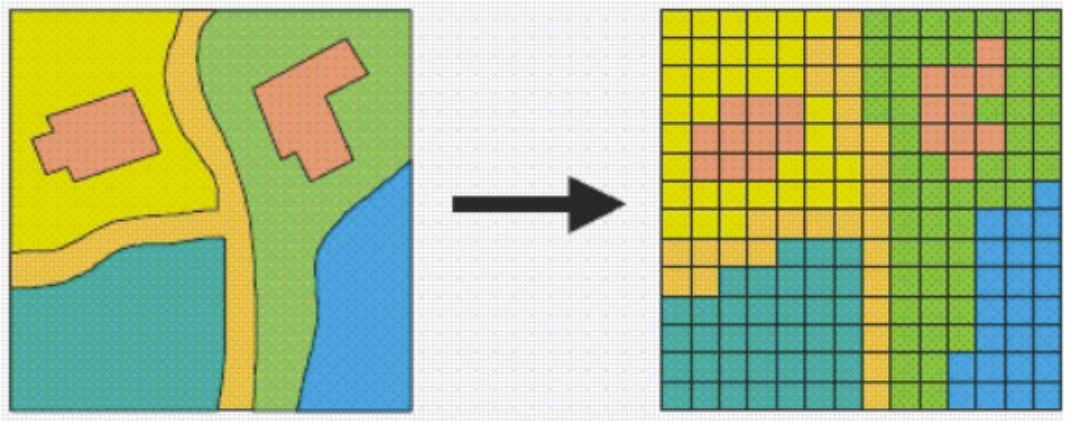


Types of Spatial Data

- Raster Data
  - Points in a multidimensional space
- Vector Data
  - Objects have spatial extent with location and boundary
  - DB typically uses geometric approximations constructed using line segments, polygons

Raster is faster but vector is corrector - old GIS adage

- The vector model uses points and line segments to identify locations on the earth while the raster model uses a series of cells to represent locations on the earth.
- The figure represents vector (left) versus raster (right) data.



https://www.esri.com/content/dam/esrisites/en-us/media/pdf/teach-with-gis/raster-faster.pdf

Raster Data

- **Raster data** consist of bit maps or pixel maps, in two or more dimensions.
  - Example 2-D raster image: satellite image of cloud cover, where each pixel stores the cloud visibility in a particular area.
  - Additional dimensions might include the temperature at different altitudes at different regions, or measurements taken at different points in time.
- Design databases generally do not store raster data.

Vector Data

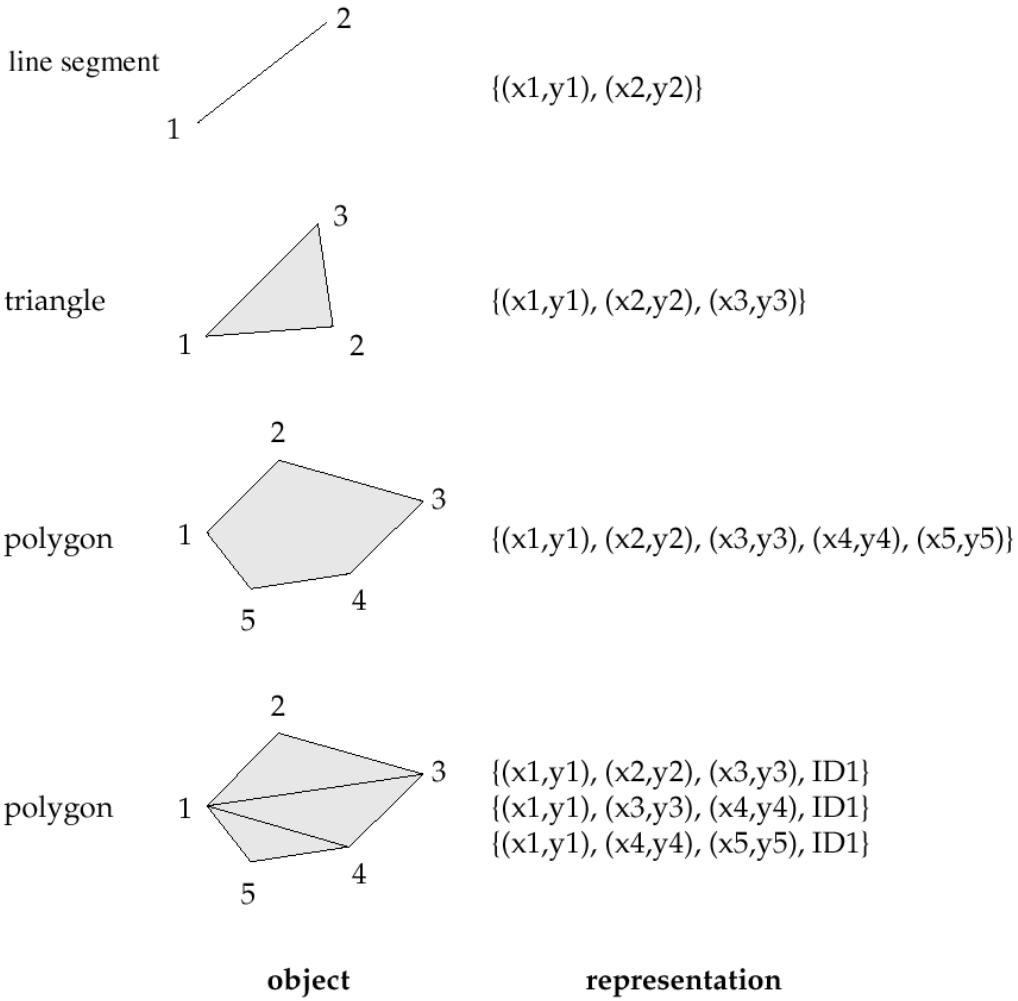
- **Vector data** are constructed from basic geometric objects: points, line segments, triangles, and other polygons in two dimensions, and cylinders, spheres, cuboids, and other polyhedrons in three dimensions.

- Vector format often used to represent map data.
  - Roads can be considered as two-dimensional and represented by lines and curves.
  - Some features, such as rivers, may be represented either as complex curves or as complex polygons, depending on whether their width is relevant.
  - Features such as regions and lakes can be depicted as polygons.

Representation of Vector Information

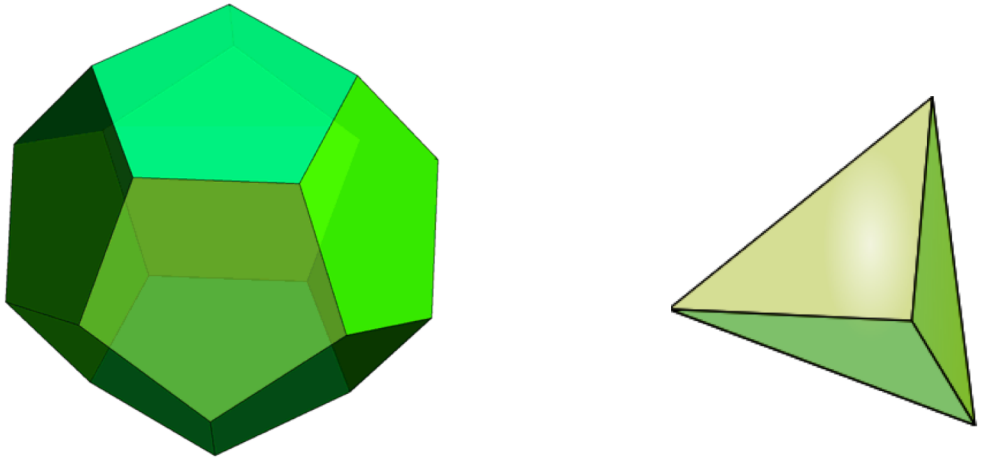
- Various geometric constructs can be represented in a database in a normalized fashion.
- Represent a line segment by the coordinates of its endpoints.
- Approximate a curve by partitioning it into a sequence of segments
  - Create a list of vertices in order, or
  - Represent each segment as a separate tuple that also carries with it the identifier of the curve (2D features such as roads).
- Closed polygons
  - List of vertices in order, starting vertex is the same as the ending vertex, or
  - Represent boundary edges as separate tuples, with each containing identifier of the polygon, or
  - Use **triangulation** — divide polygon into triangles
    - Note the polygon identifier with each of its triangles.

Representation of Vector Data



Representation of Vector Data (Cont.)

- Representation of points and line segments in 3-D similar to 2-D, except that points have an extra z component
- Represent arbitrary polyhedra by dividing them into tetrahedrons, like triangulating polygons.

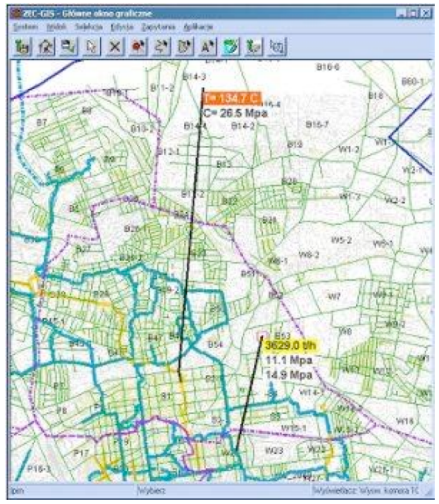


Types of Spatial Queries

- Spatial Range Queries
  - *"Find all cities within 50 miles of Madison"*
    - Query has associated region (location, boundary)
    - Answer includes overlapping or contained data regions
- Nearest-Neighbor Queries
  - *"Find the 10 cities nearest to Madison"*
    - Results must be ordered by proximity
- Spatial Join Queries
  - *"Find all cities near a lake"*
    - Expensive, join condition involves regions and proximity
- Region queries deal with spatial regions.
  - e.g., ask for objects that lie partially or fully inside a specified region.

Applications of Spatial Data

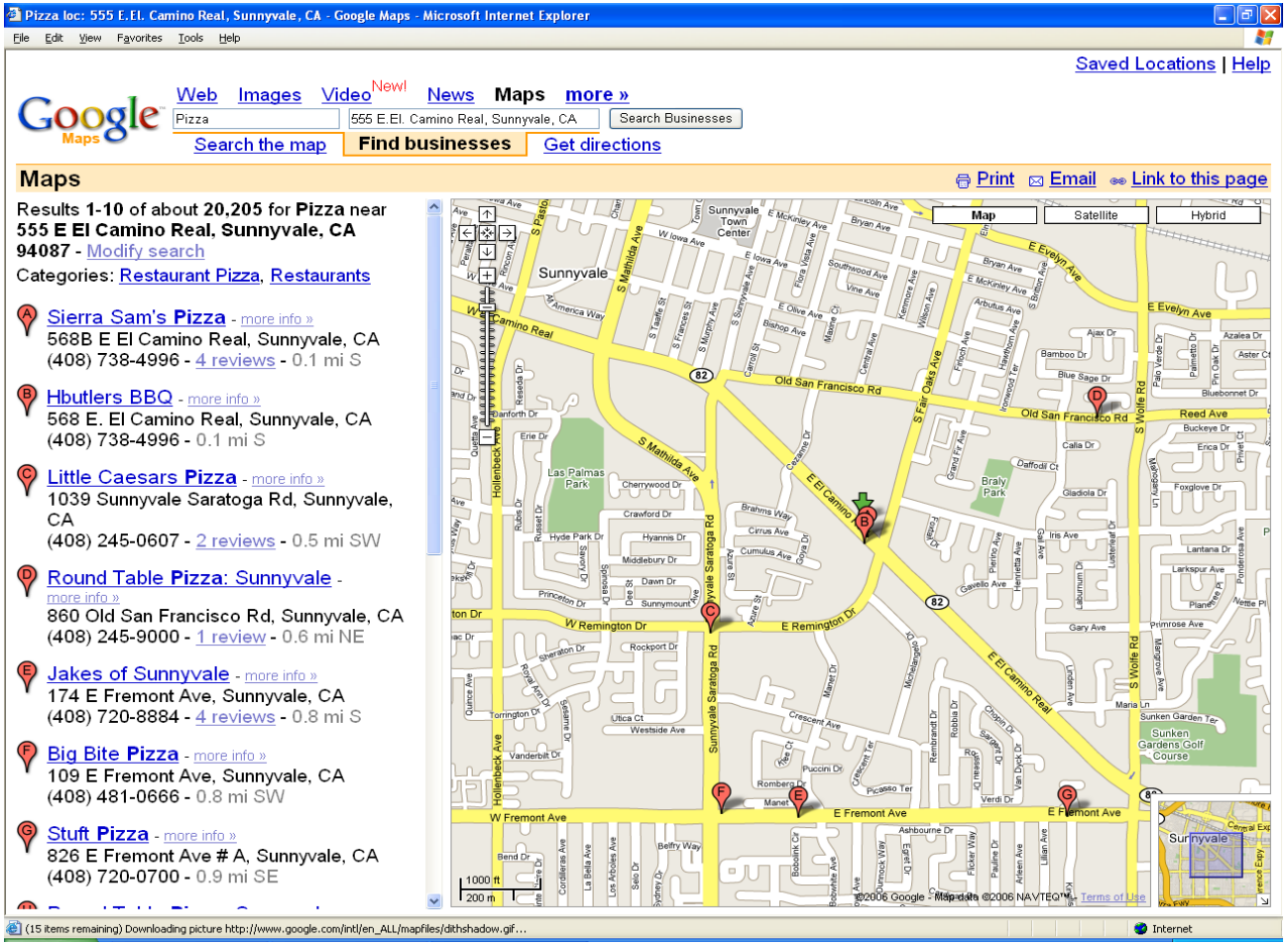
- Geographic Information Systems (GIS)
  - E.g., ESRI's ArcInfo; OpenGIS Consortium
  - Geospatial information
  - All classes of spatial queries and data are common



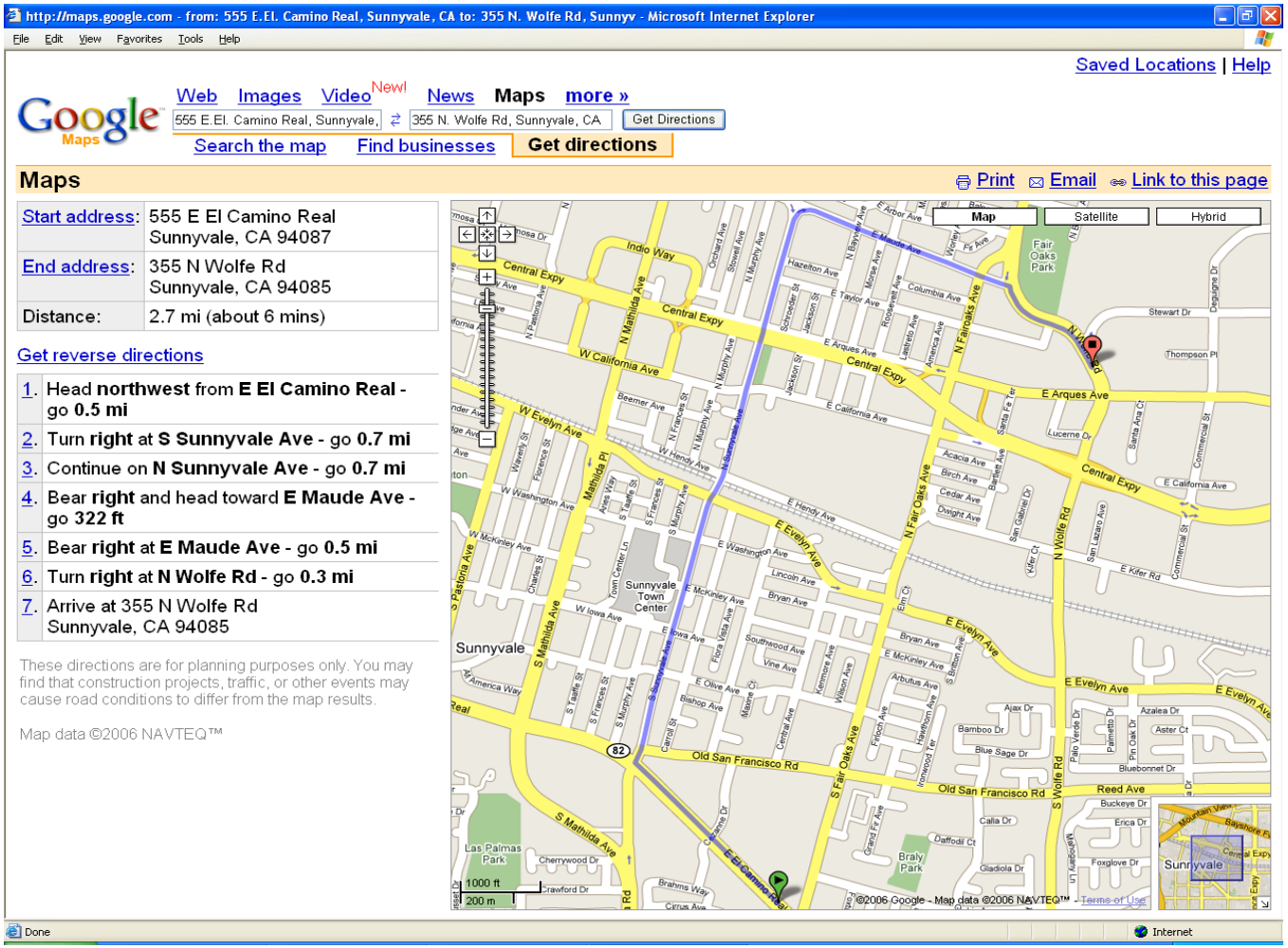
- Computer-Aided Design/Manufacturing
  - Store spatial objects such as surface of airplane fuselage
  - Range queries and spatial join queries are common

- Multimedia Databases
  - Images, video, text, etc. stored and retrieved by content
  - First converted to *feature vector* form; high dimensionality
  - Nearest-neighbor queries are the most common

Applications of Spatial Data



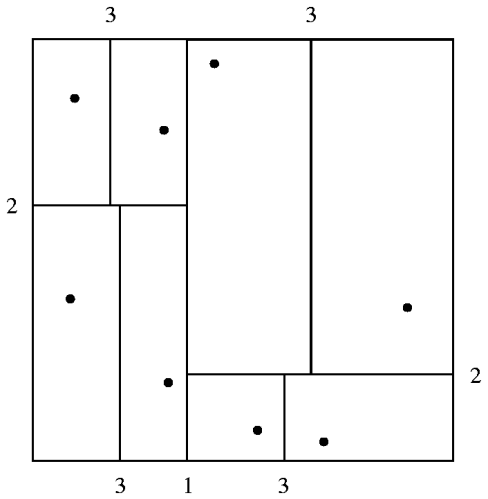
Applications of Spatial Data



Indexing of Spatial Data

- **k-d tree** - early structure used for indexing in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
  - choose one dimension for partitioning at the root level of the tree.
  - choose another dimensions for partitioning in nodes at the next level and so on, cycling through the dimensions.
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
- Partitioning stops when a node has less than a given maximum number of points.
- The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.

Division of Space by a k-d Tree

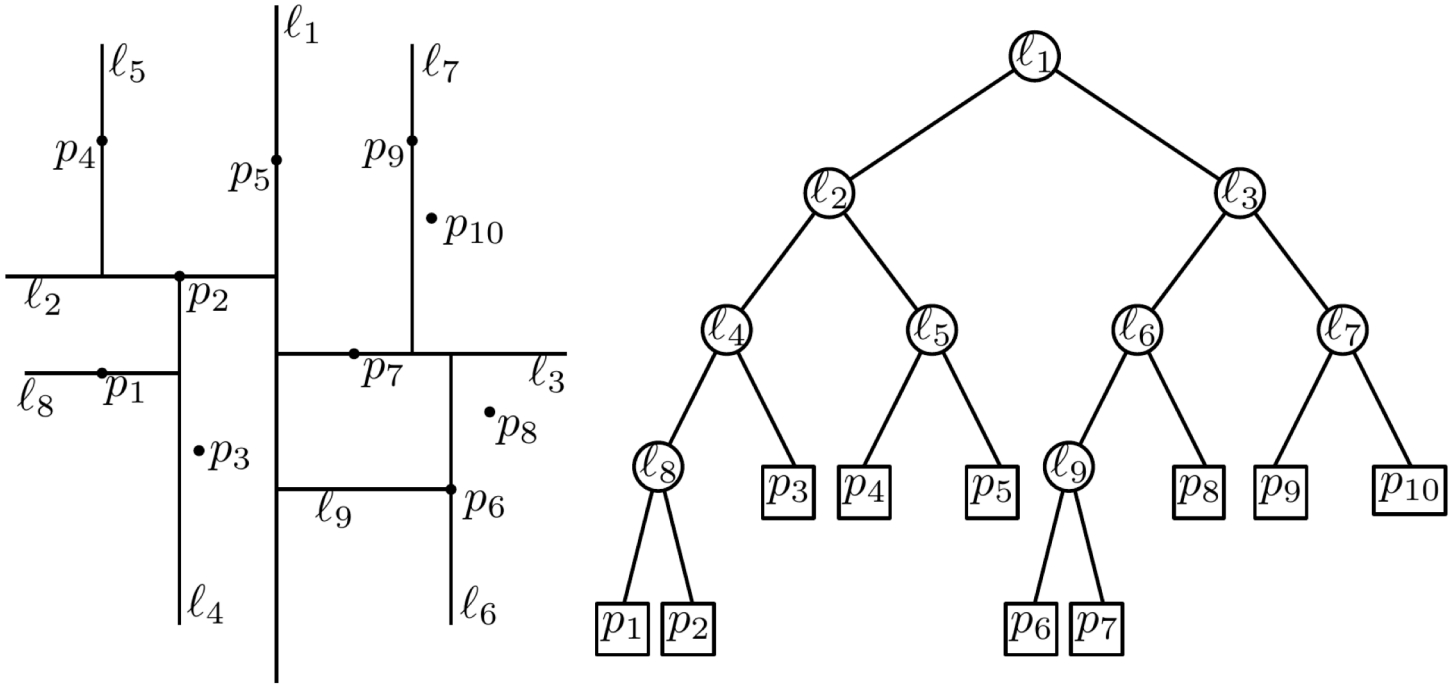


- Each line in the figure (other than the outside box) corresponds to a node in the *k-d* tree
  - the maximum number of points in a leaf node has been set to 1.

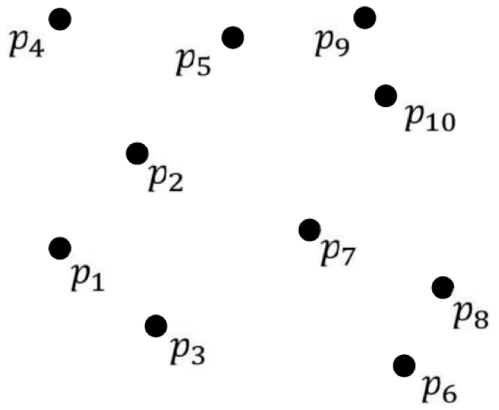
- The numbering of the lines in the figure indicates the level of the tree at which the corresponding node appears.

Kd-tree

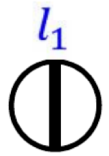
- Alternate between coordinates and split the point set

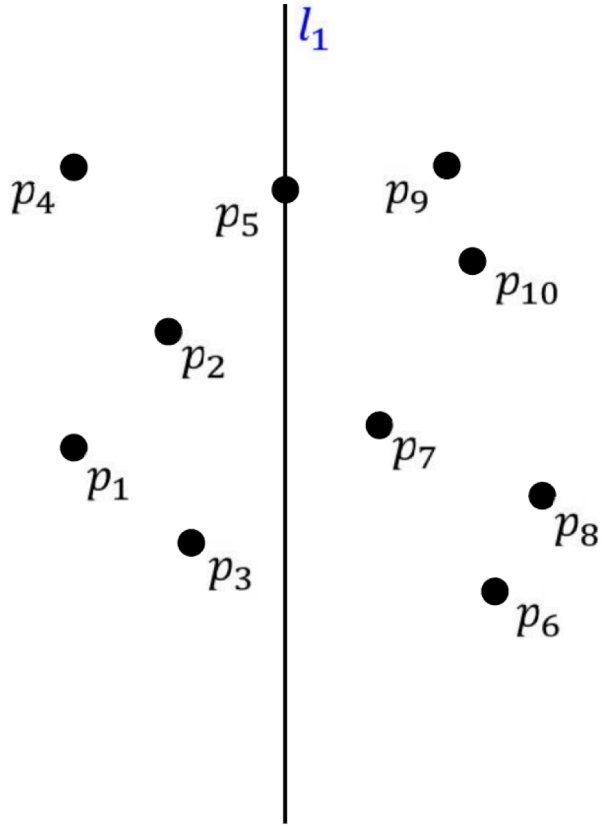


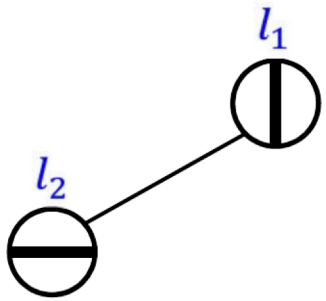
K-d Tree

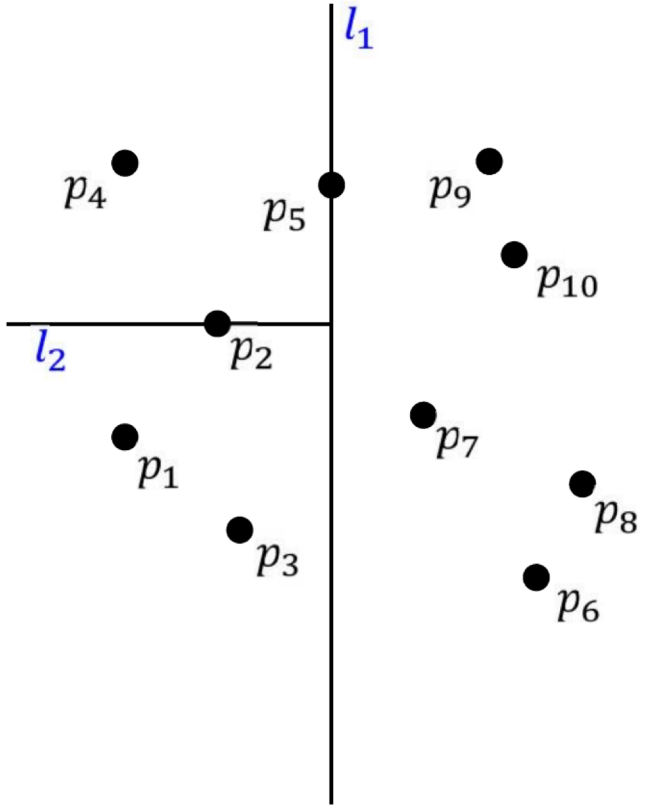


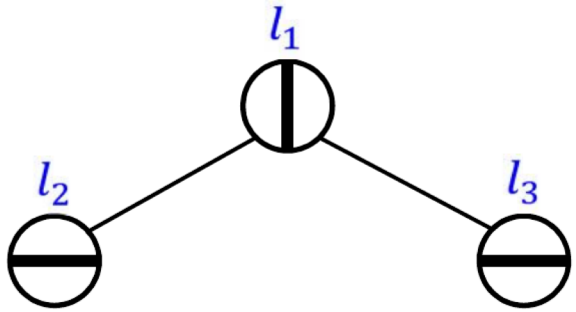
K-d tree

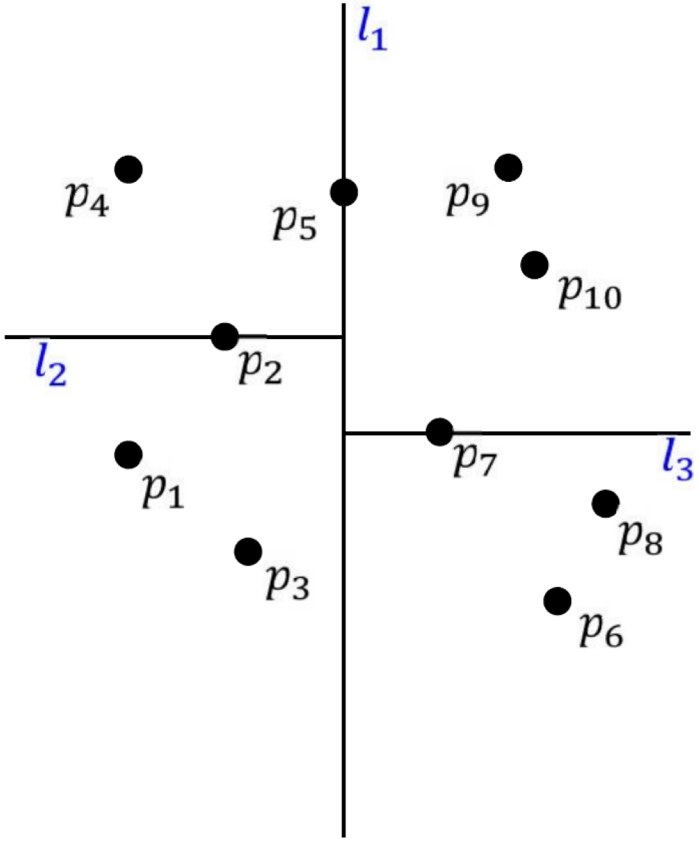


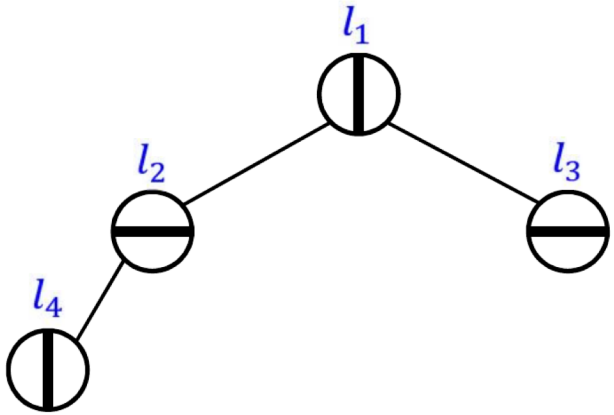


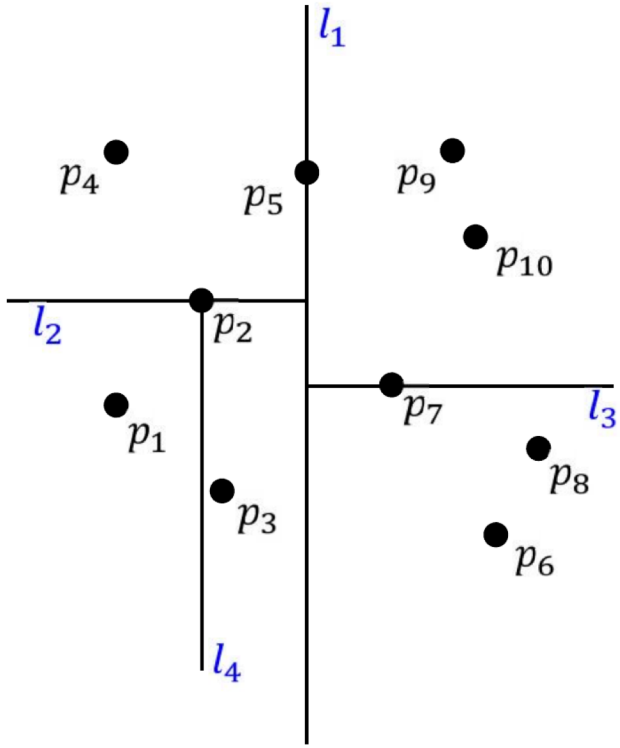

l_1
l_2

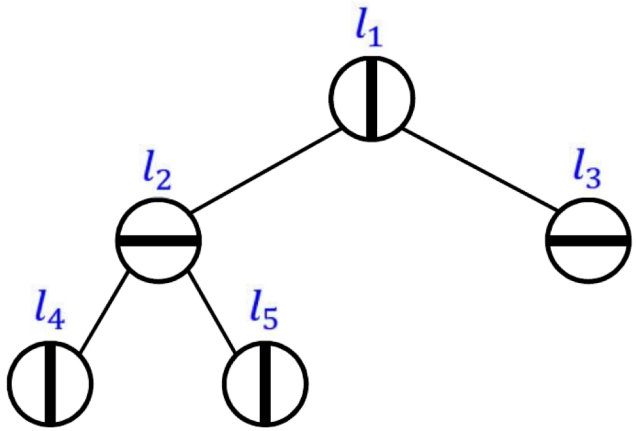

l_1
p_4
p_5
p_9
p_{10}
l_2
p_2
p_7
p_1
p_8
p_3
p_6

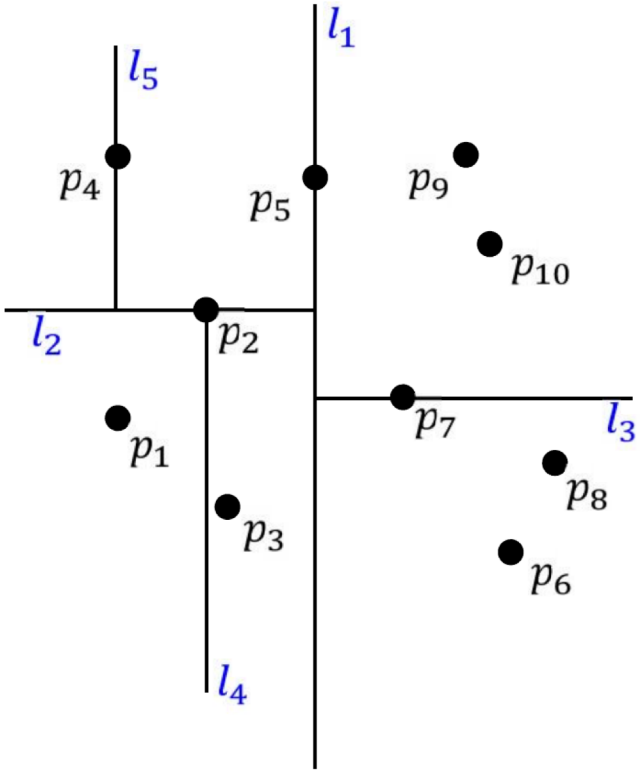

l_1
l_2
l_3

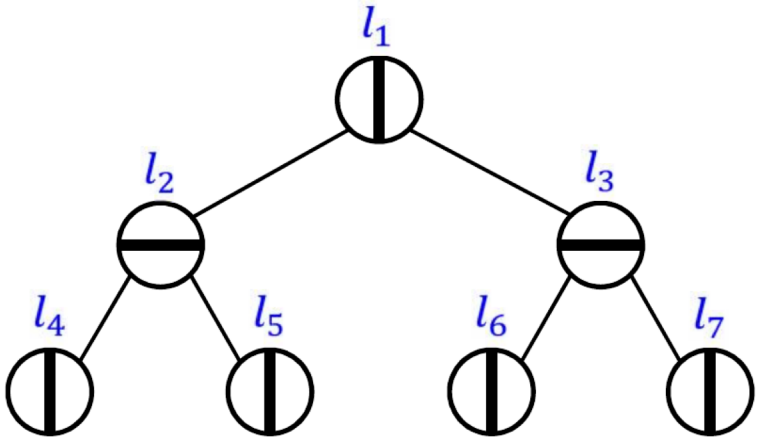

l_1
p_4
p_5
p_9
p_{10}
l_2
p_2
p_7
l_3
p_1
p_8
p_3
p_6

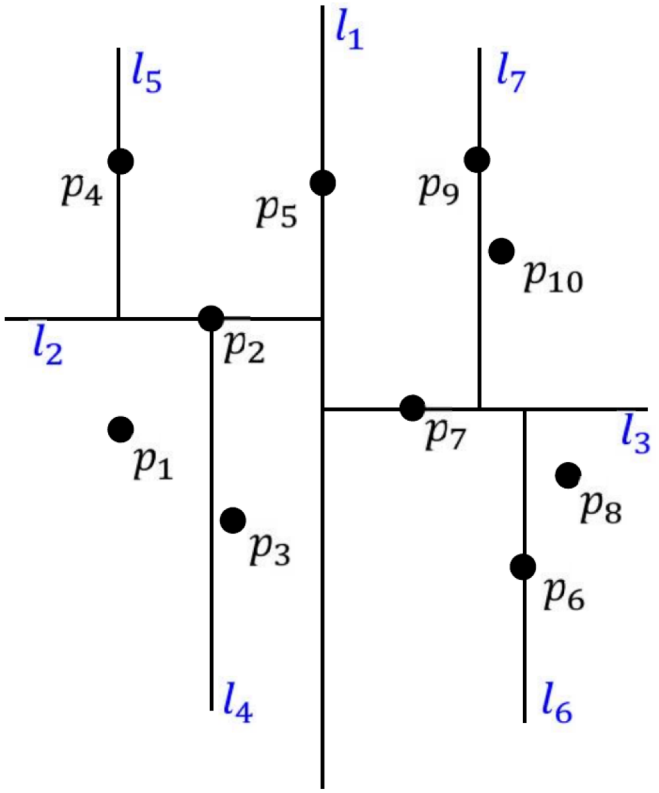

l_1
l_2
l_3
l_4

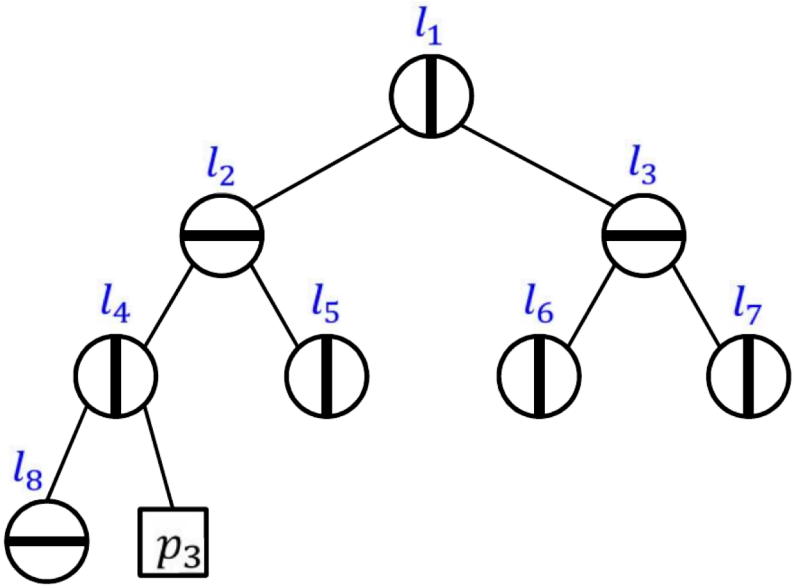

l_1
p_4
p_5
p_9
p_{10}
l_2
p_2
p_7
l_3
p_1
p_8
p_3
p_6
l_4

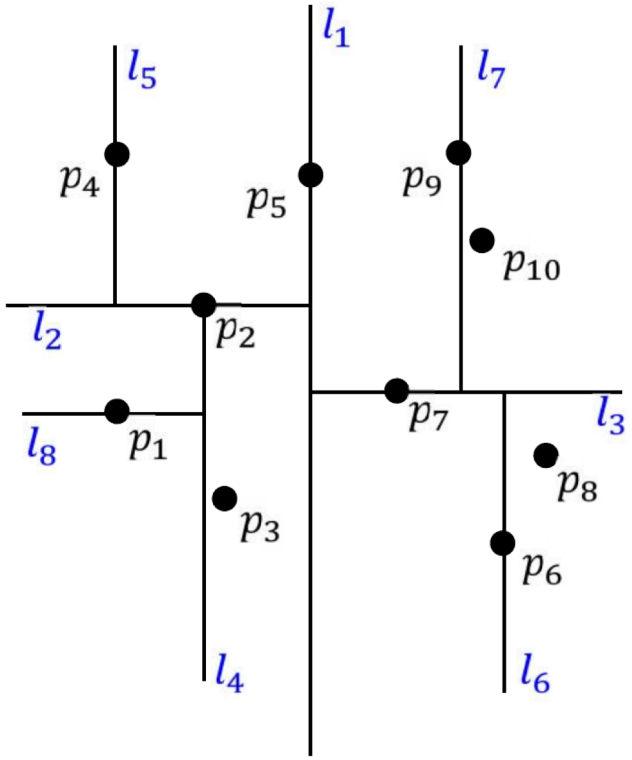

l_1
l_2
l_3
l_4
l_5

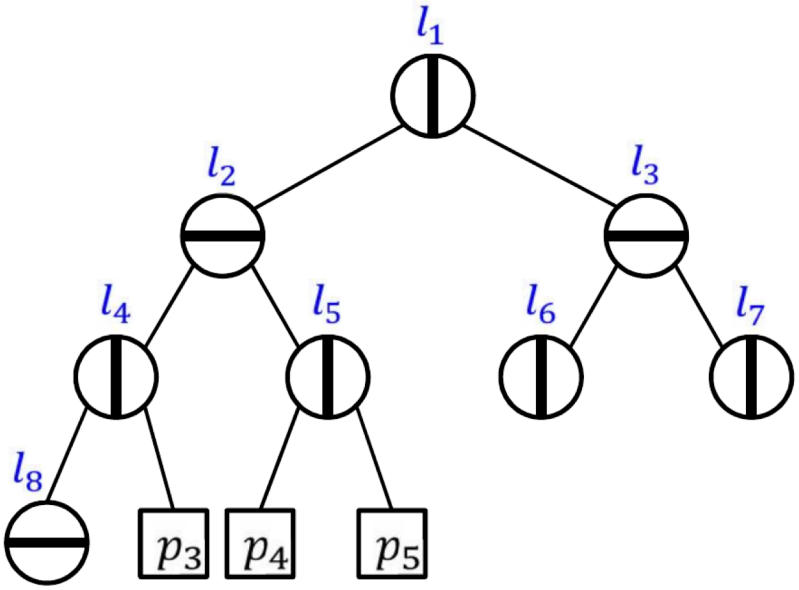

l_1
l_5
p_4
p_5
p_9
p_{10}
l_2
p_2
p_7
l_3
p_1
p_8
p_3
p_6
l_4

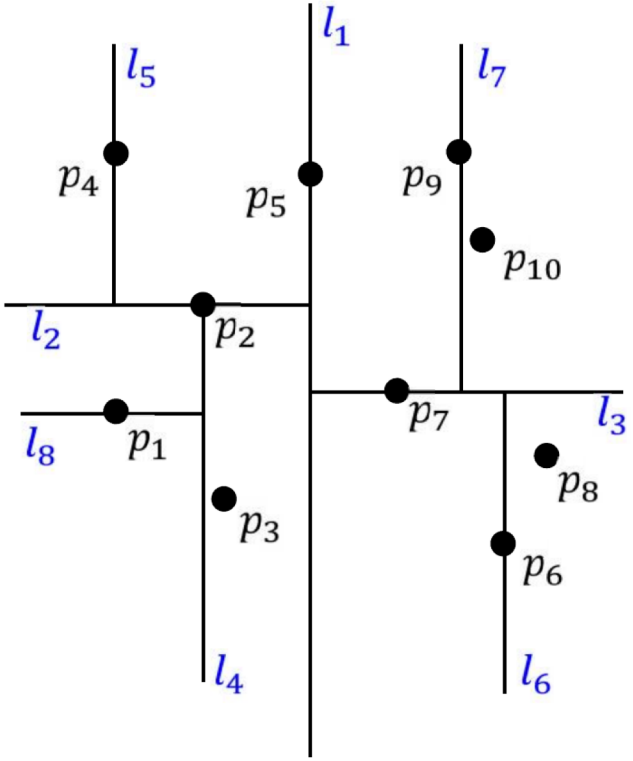

l_1
l_2
l_3
l_4
l_5
l_6
l_7

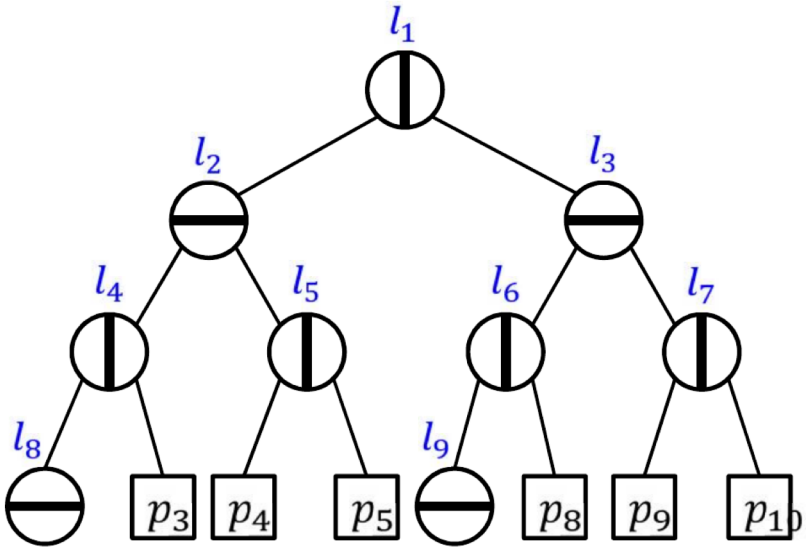

l_1
l_5
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
p_7
l_3
p_1
p_8
p_3
p_6
l_4
l_6

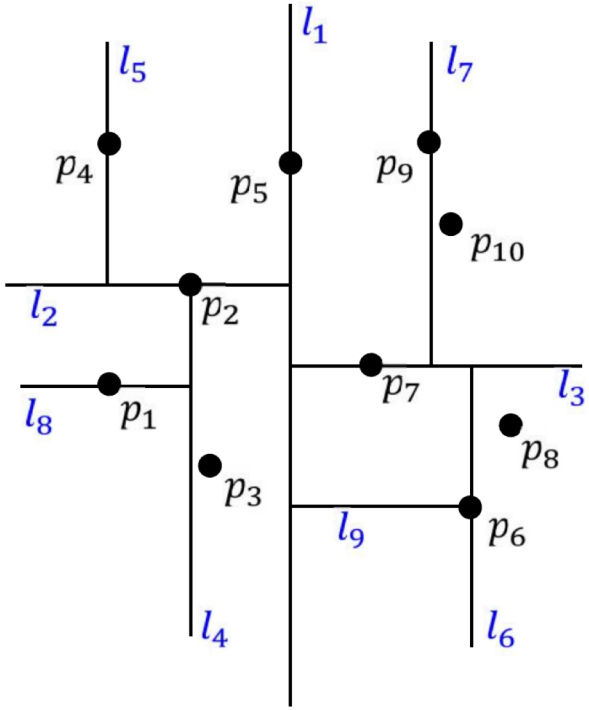

l_1
l_2
l_3
l_4
l_5
l_6
l_7
l_8
p_3

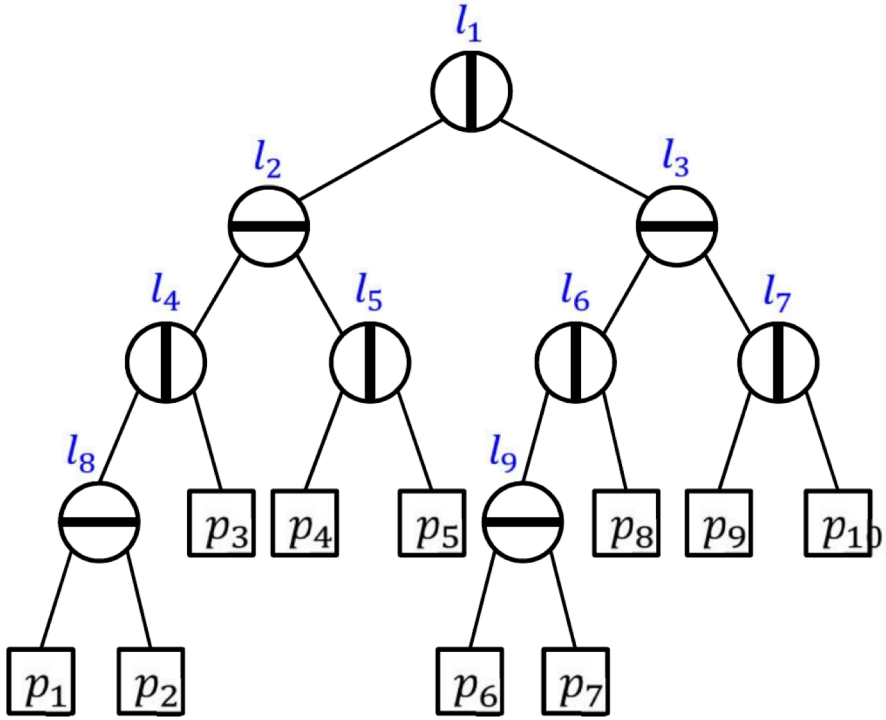

l_1
l_5
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
p_7
l_3
l_8
p_1
p_8
p_3
p_6
l_4
l_6

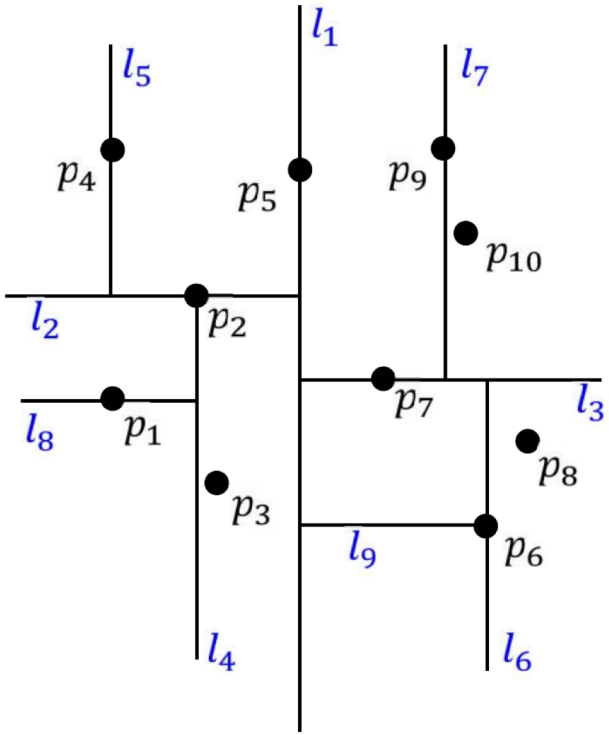

l_1
l_2
l_3
l_4
l_5
l_6
l_7
l_8
p_3
p_4
p_5


l_1
l_5
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
p_7
l_3
l_8
p_1
p_8
p_3
p_6
l_4
l_6


l_1
l_2
l_3
l_4
l_5
l_6
l_7
l_8
l_9
p_3
p_4
p_5
p_8
p_9
p_{10}


l_1
l_5
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
p_7
l_3
l_8
p_1
p_8
p_3
l_9
p_6
l_4
l_6

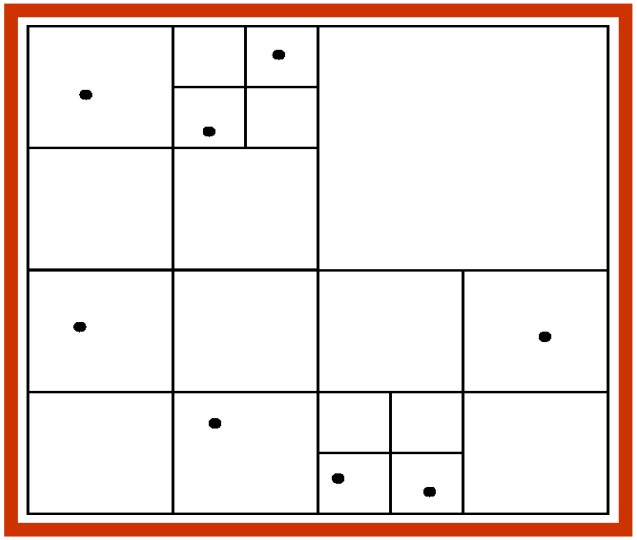

l_1
l_2
l_3
l_4
l_5
l_6
l_7
l_8
l_9
p_1
p_2
p_3
p_4
p_5
p_6
p_7
p_8
p_9
p_{10}


l_1
l_5
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
p_7
l_3
l_8
p_1
p_8
p_3
l_9
p_6
l_4
l_6

Division of Space by Quadtrees

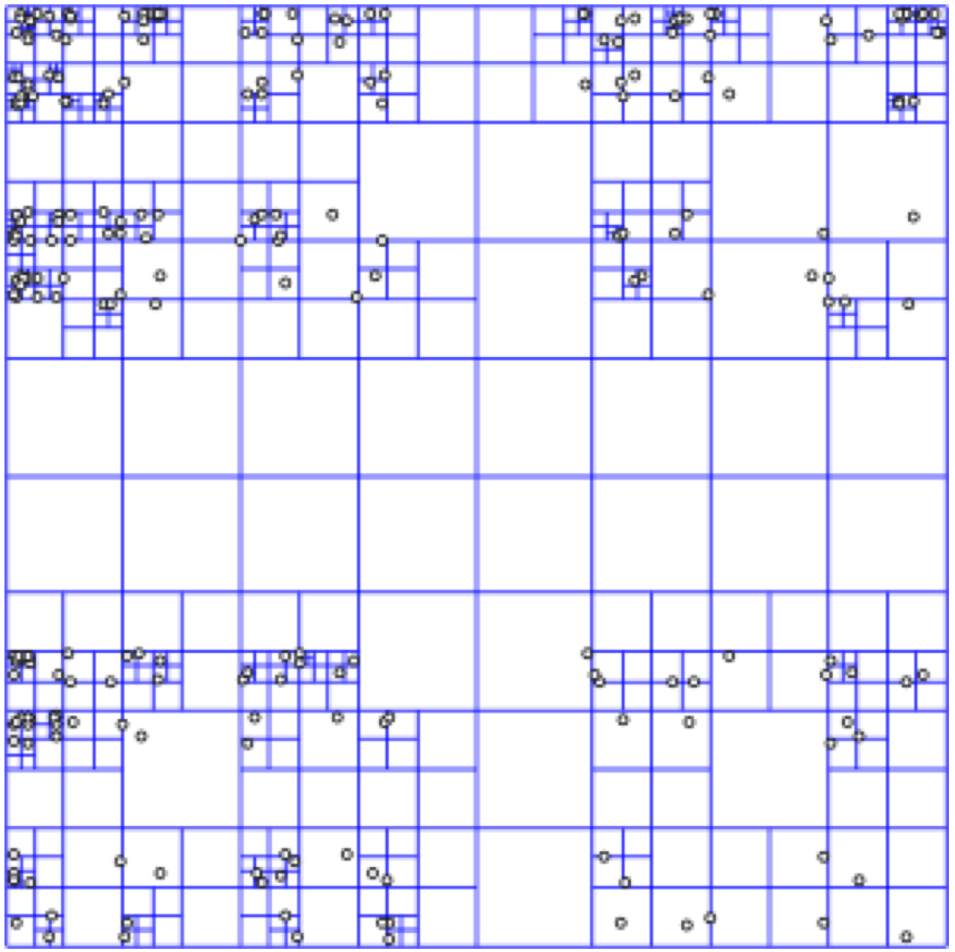
Quadtrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf node divides its region into four equal sized quadrants
  - correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have between zero and some fixed maximum number of points (set to 1 in example).



Quadtree

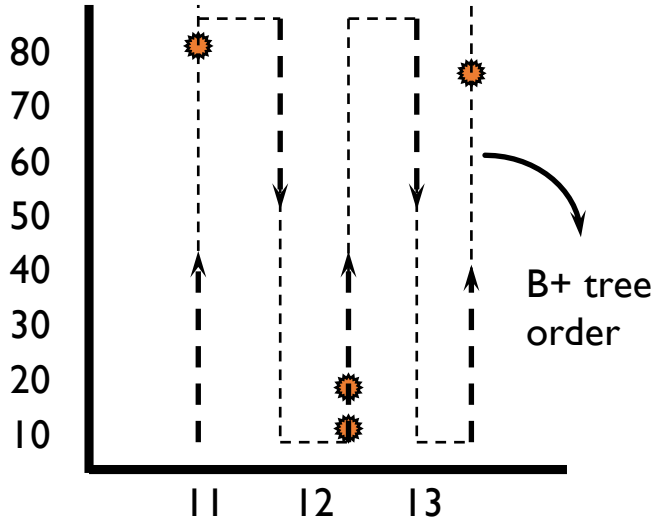
- Quadtree has more nodes for regions with more points
- The tree is not balanced
  - Depth of leaves is not the same
  - A region with more points will be in a larger depth



Single Dimensional Indexes

- B+ trees are fundamentally single-dimensional indexes.
- When we create a composite search key B+ tree, e.g., an index on <age, sal>, we effectively linearize the 2-dimensional space since we sort entries first by age and then by sal.

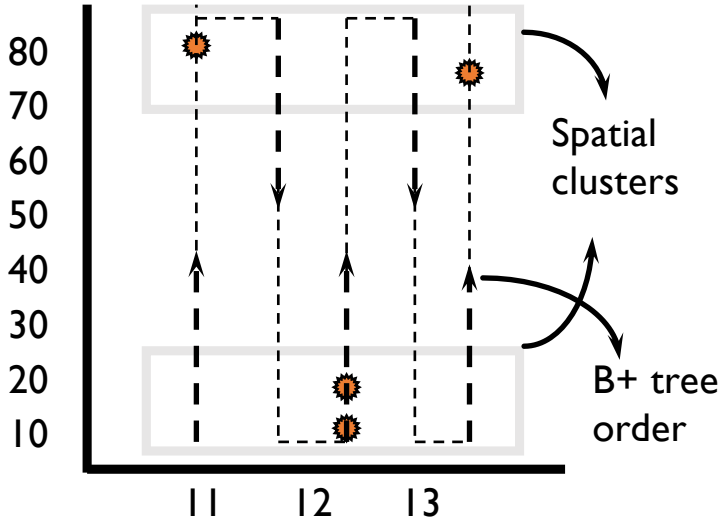
Consider entries:
<11, 80>, <12, 10>
<12, 20>, <13, 75>



Multidimensional Indexes

- A multidimensional index clusters entries so as to exploit "nearness" in multidimensional space.
- Keeping track of entries and maintaining a balanced index structure presents a challenge!

Consider entries:
<11, 80>, <12, 10>
<12, 20>, <13, 75>



Motivation for Multidimensional Indexes

- Spatial queries (GIS, CAD).
  - Find all hotels within a radius of 5 miles from the conference venue.
  - Find the city with population 500,000 or more that is nearest to Kalamazoo, MI.
  - Find all cities that lie on the Nile in Egypt.
  - Find all parts that touch the fuselage (in a plane design).
- Similarity queries (content-based retrieval).
  - Given a face, find the five most similar faces.
- Multidimensional range queries.
  - 50 < age < 55 AND 80K < sal < 90K

What is the difficulty?

- An index based on spatial location needed.
  - One-dimensional indexes don't support multidimensional searching efficiently.
  - Hash indexes only support point queries; want to support range queries as well.
  - Must support inserts and deletes gracefully.
- Ideally, want to support non-point data as well (e.g., lines, shapes).
- The R-tree meets these requirements, and variants are widely used today.
  - Since 1984. Thanks to Guttman

R-TREES: A DYNAMIC INDEX STRUCTURE FOR SPATIAL SEARCHING

Antonin Guttman
University of California
Berkeley

Abstract

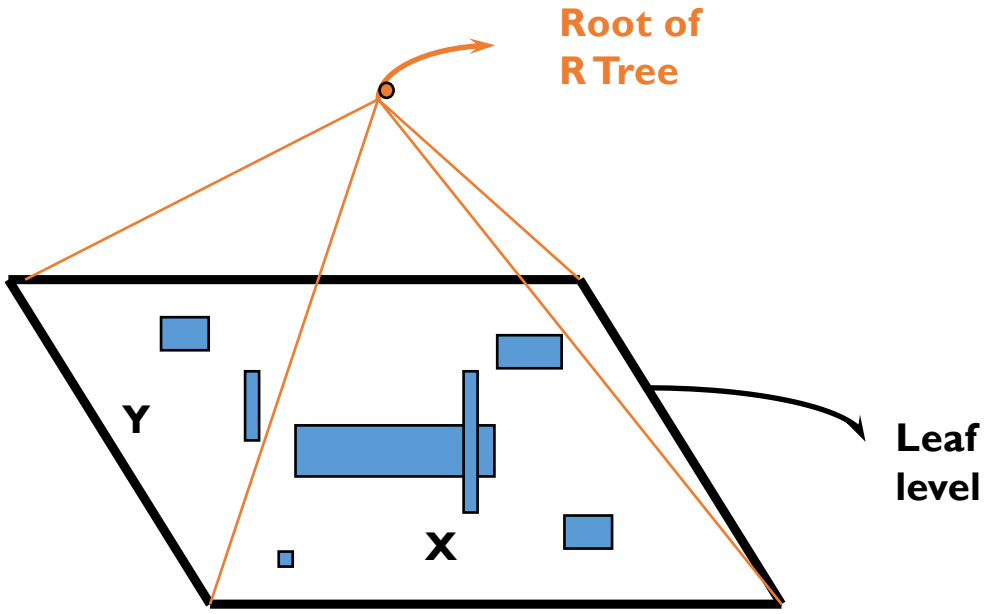
In order to handle spatial data efficiently, as required in computer aided design and geo-data applications, a database system needs an index mechanism that will help it retrieve data items quickly according to their spatial locations. However, traditional indexing methods are not well suited to data objects of non-zero size located in multi-dimensional spaces. In this paper we describe a dynamic index structure called an R-tree which meets this need, and give algorithms for searching and updating it. We present the results of a series of tests which indicate that the structure performs well, and conclude that it is useful for current database systems in spatial applications.

Proceedings of the 1984 ACM SIGMOD international conference on Management of data

Special Interest Group on Management of Data

R-Trees

- The R-tree is a tree-structured index that remains balanced on inserts and deletes.
- Each key stored in a leaf entry is intuitively a box, or collection of intervals, with one interval per dimension.
- Example in 2-D:



R-Trees

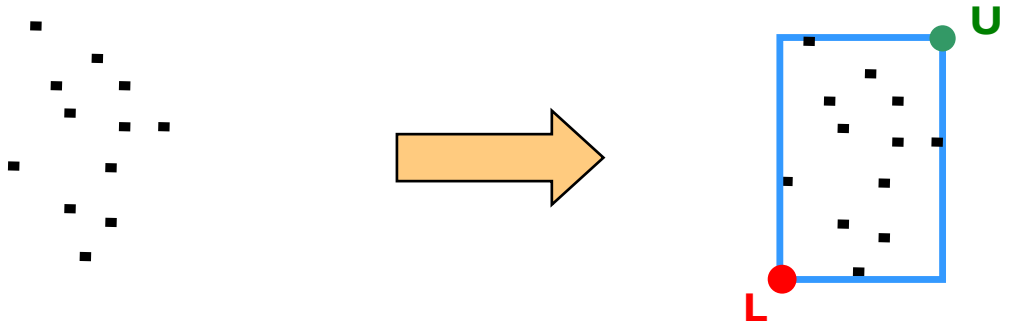
- R-Trees can organize any-dimensional data by representing the data by a minimum bounding box.
- Each node bounds it's children. A node can have many objects in it
- The leaves point to the actual objects (stored on disk probably)
- The height is always log n (it is height balanced)

R Tree Properties

- Leaf entry = < n-dimensional box, rid >
  - This is Alternative (2), with *key value* being a box.
  - Box is the tightest bounding box for a data object.
- Non-leaf entry = < n-dimensional box, ptr to child node >
  - Box covers all boxes in child node (in fact, subtree).
- All leaves at same distance from root.
- Nodes can be kept 50% full (except root).
  - Can choose a parameter *m* that is <= 50%, and ensure that every node is at least *m*% full.

Bounding Rectangle

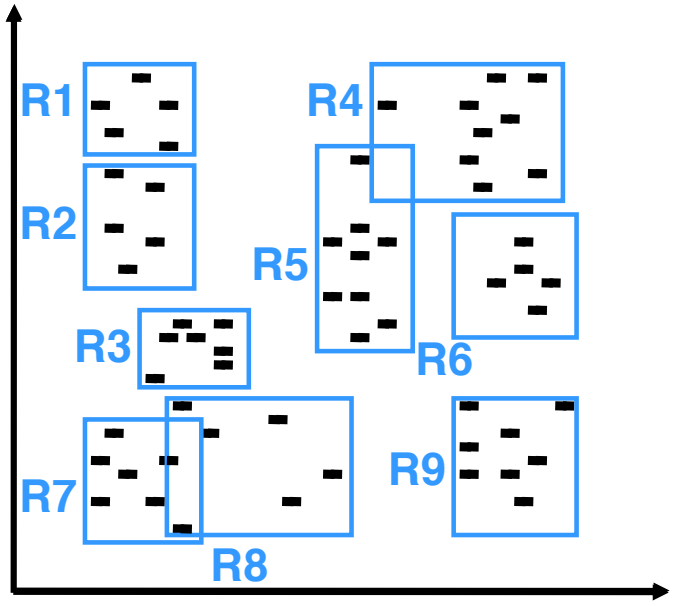
- Suppose we have a cluster of points in 2-D space...
  - We can build a "box" around points. The smallest box (which is axis parallel) that contains all the points is called a Minimum Bounding Rectangle (MBR)
    - also known as minimum bounding box



MBR = {(L.x, L.y) (U.x, U.y)}

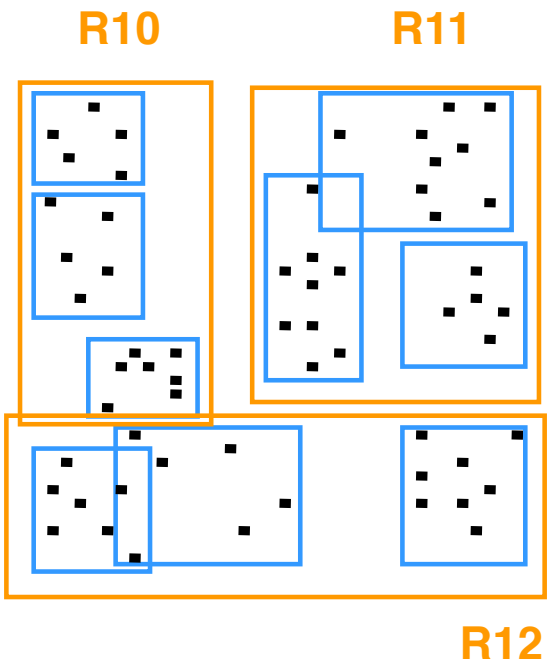
Clustering Points

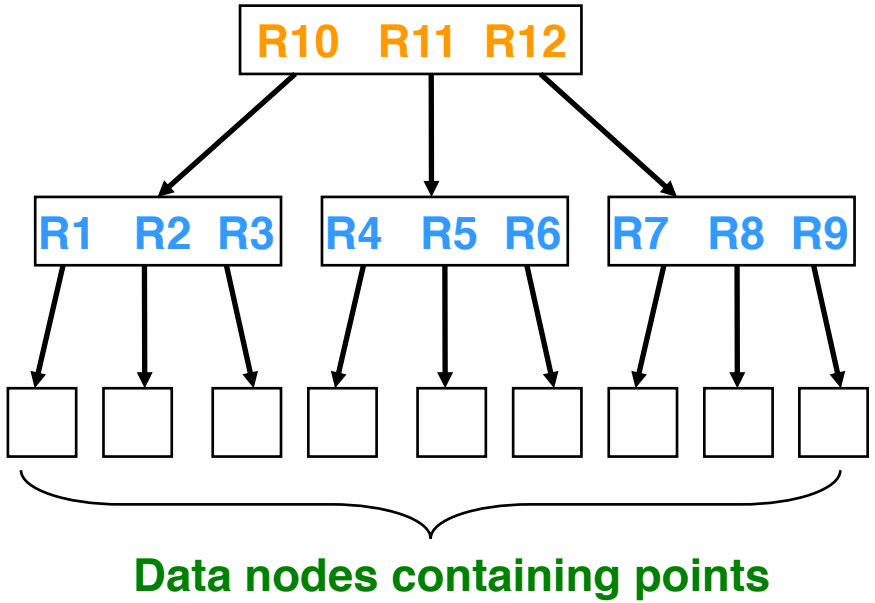
- We can group clusters of datapoints into MBRs
  - Can also handle line-segments, rectangles, polygons, in addition to points



We can further recursively group MBRs into larger MBRs….

R-Tree Structure





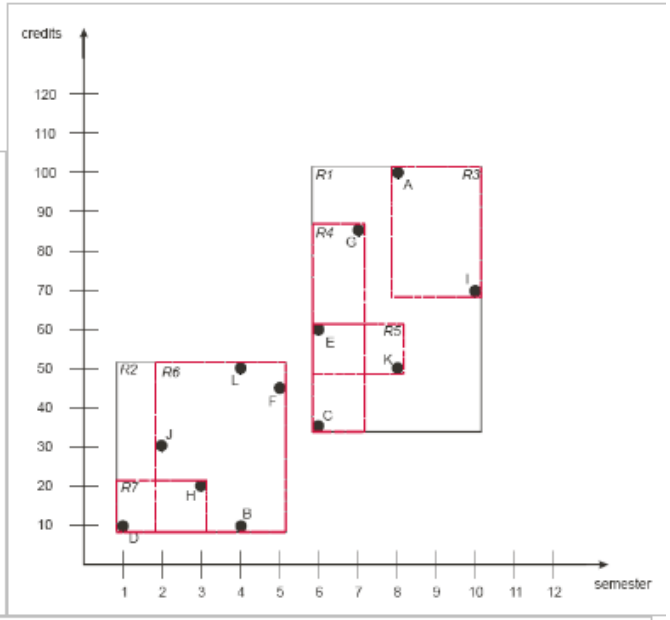
R Trees

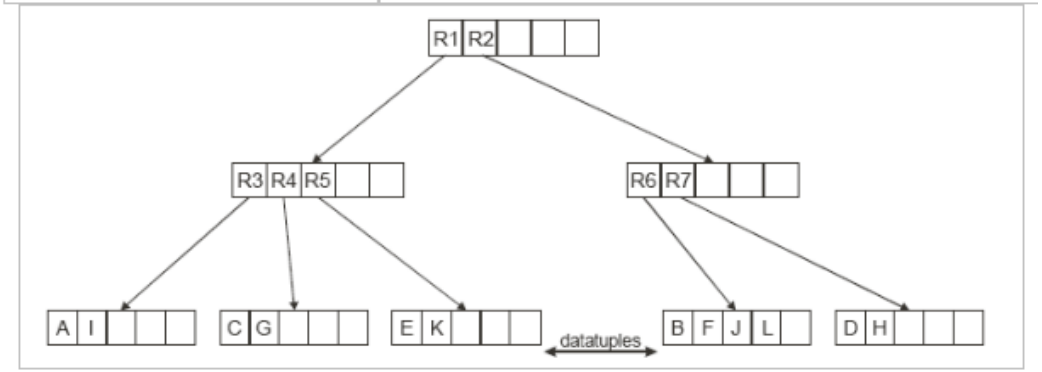
- A rectangular **bounding box (MBR)** is associated with each tree node.
  - Bounding box of a leaf node is a minimum sized rectangle that contains all the rectangles/polygons associated with the leaf node.
  - The bounding box associated with a non-leaf node contains the bounding box associated with all its children.
  - Bounding box of a node serves as its key in its parent node (if any)
  - *Bounding boxes of children of a node are allowed to overlap*

- A polygon is stored only in one node, and the bounding box of the node must contain the polygon
  - The storage efficiency or R-trees is better than that of k-d trees or quadtrees since a polygon is stored only once

R-Tree Example

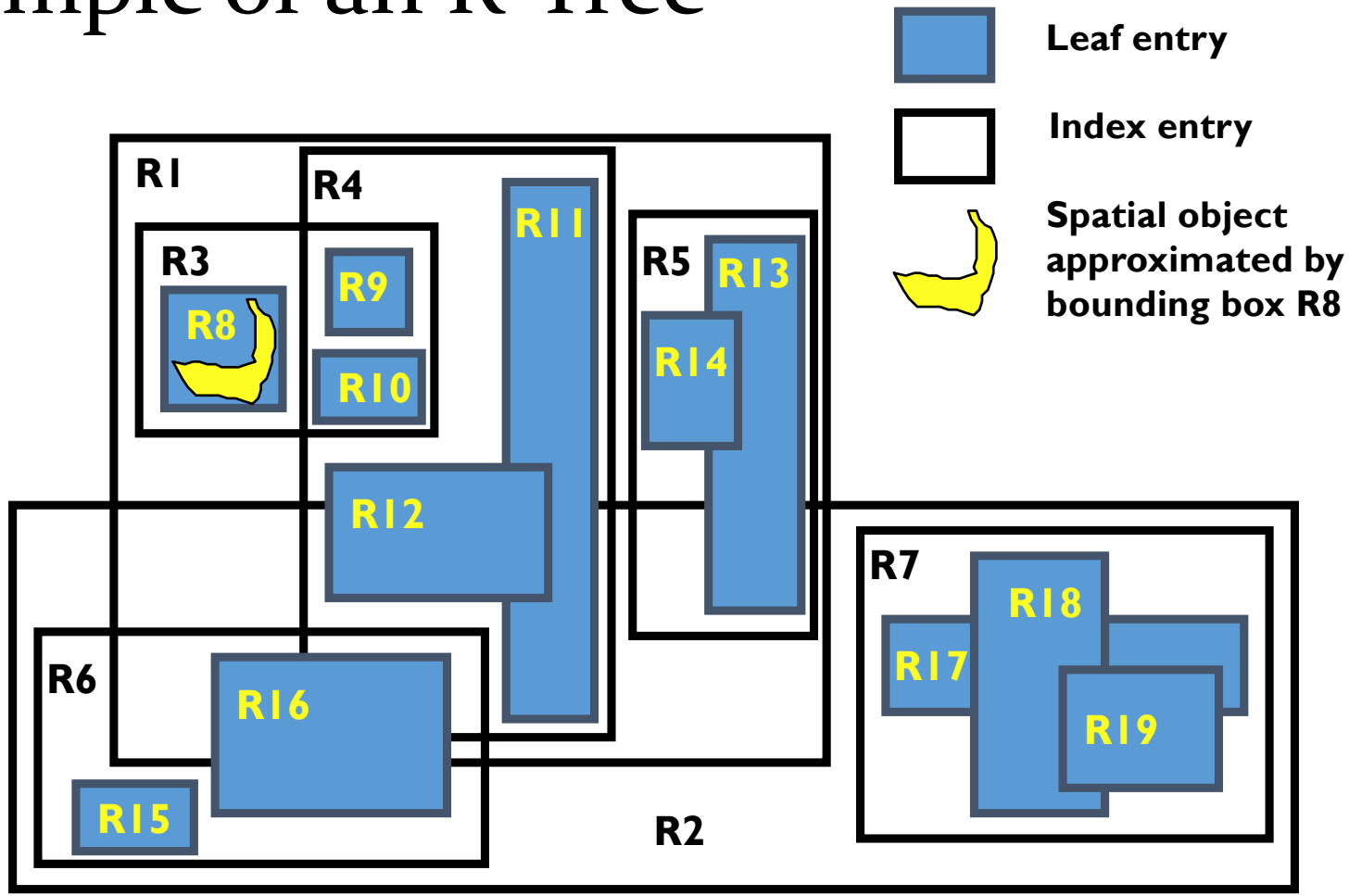
name	semester	credits
A	8	100
B	4	10
C	6	35
D	1	10
E	6	40
F	5	45
G	7	85
H	3	20
I	10	70
J	2	30
K	8	50
L	4	50



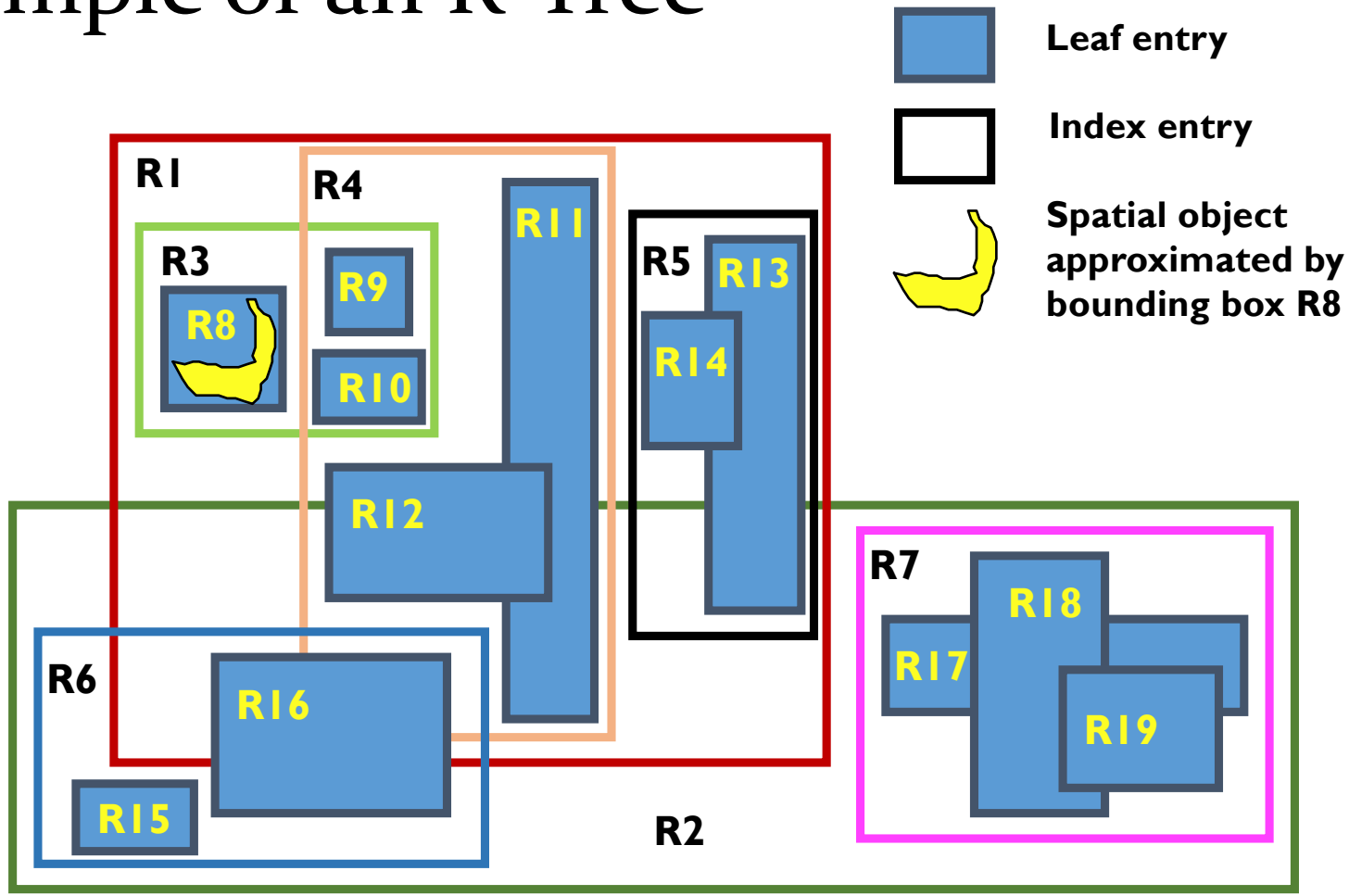


From http://lacot.org/public/enst/bda/img/schema1.gif

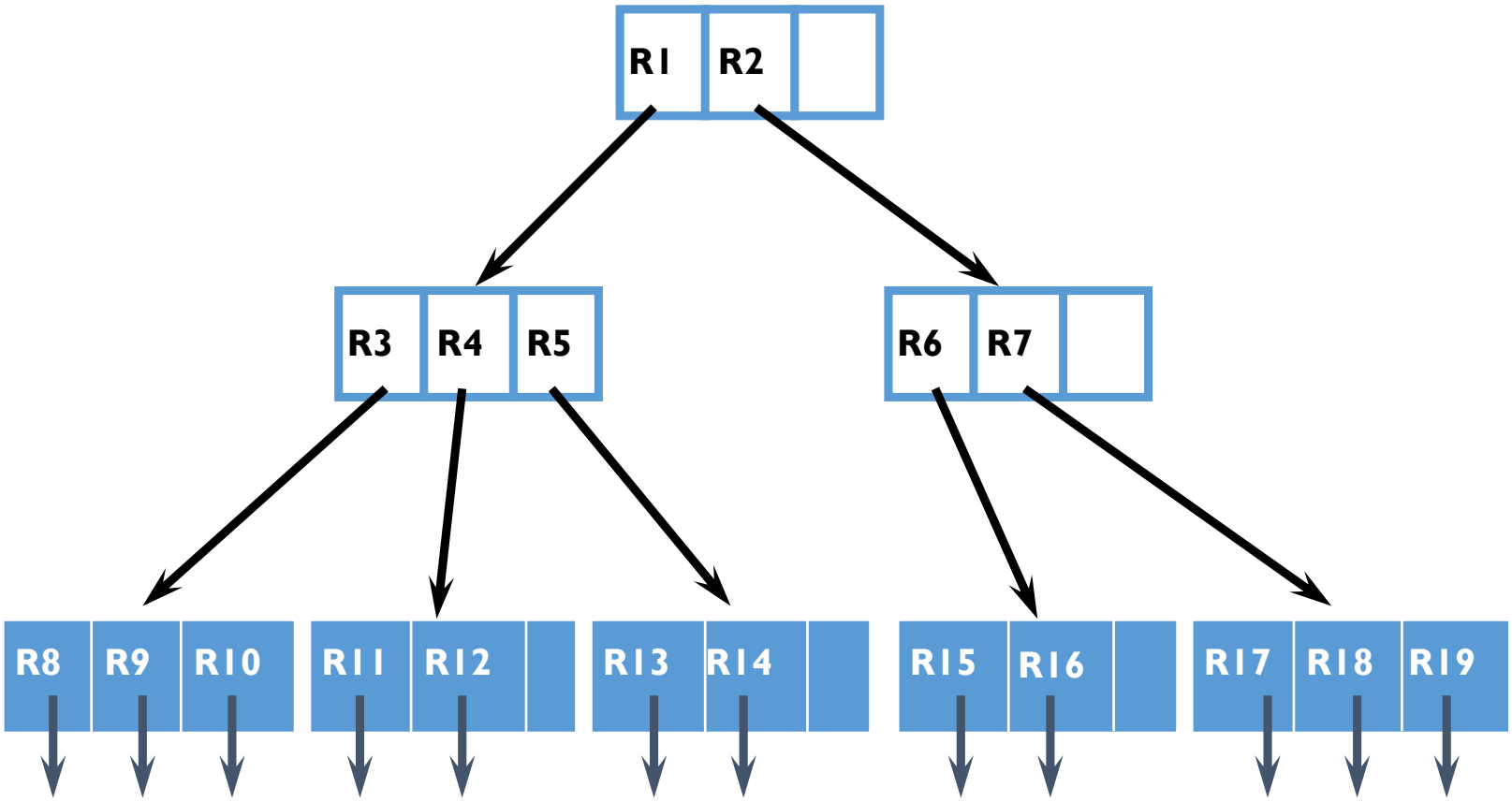
Example of an R-Tree



Example of an R-Tree

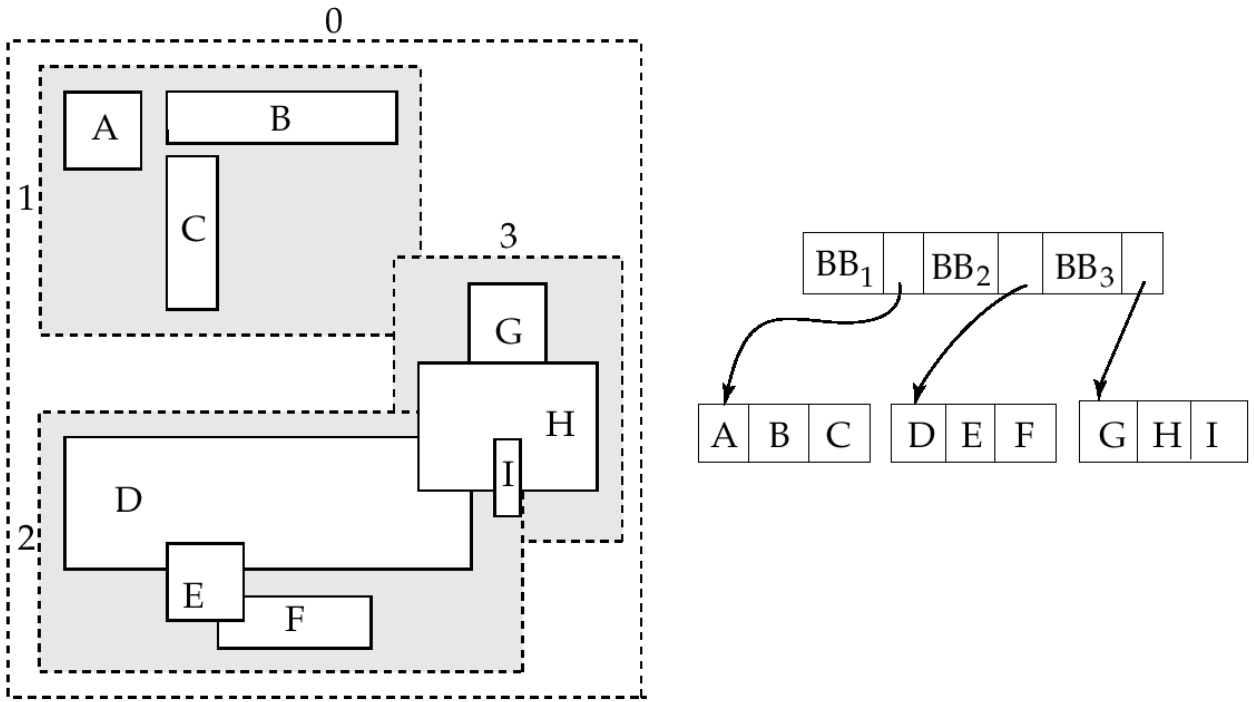


Example R-Tree (Contd.)



Example R-Tree

- A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles. The R-tree is shown on the right.



Search for Objects Overlapping

Start at root.

1. If current node is non-leaf, for each entry <E, ptr>, if ***box*** E overlaps Q, search subtree identified by ptr.

2. If current node is leaf, for each entry <E, rid>, if E overlaps Q, rid identifies an object that might overlap Q.

Note: May have to search several subtrees at each node!
(In contrast, a B-tree equality search goes to just one leaf.)

Improving Search Using Constraints

- It is convenient to store boxes in the R-tree as approximations of arbitrary regions, because boxes can be represented compactly.

- But why not use convex polygons to approximate query regions more accurately?
  - Will reduce overlap with nodes in tree, and reduce the number of nodes fetched by avoiding some branches altogether.
  - Cost of overlap test is higher than bounding box intersection, but it is a main-memory cost, and can actually be done quite efficiently. Generally a win.

Insert Entry <B, ptr>

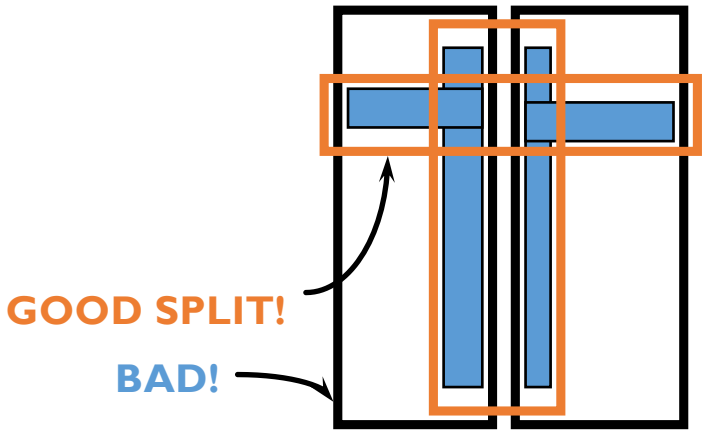
- Start at root and go down to "best-fit" leaf L.
  - Go to child whose box needs least enlargement to cover B; resolve ties by going to smallest area child.

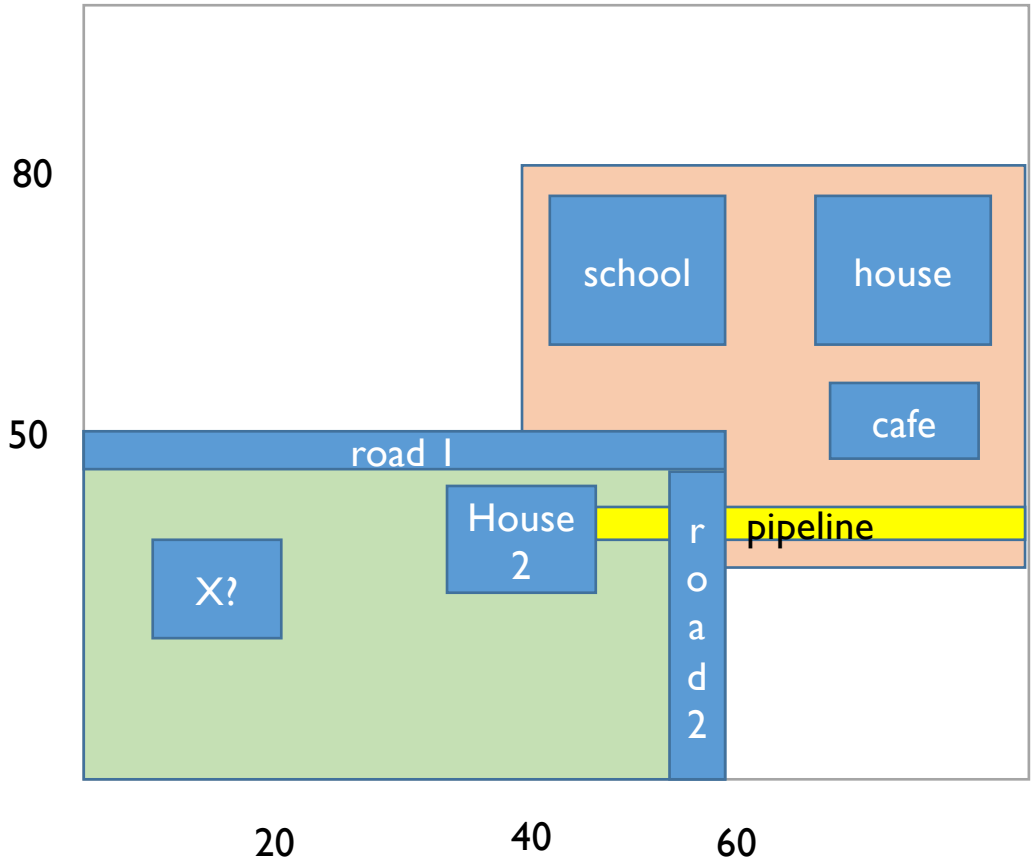
- If best-fit leaf L has space, insert entry and stop. Otherwise, split L into L1 and L2.
  - Adjust entry for L in its parent so that the box now covers (only) L1.
  - Add an entry (in the parent node of L) for L2. (This could cause the parent node to recursively split.)

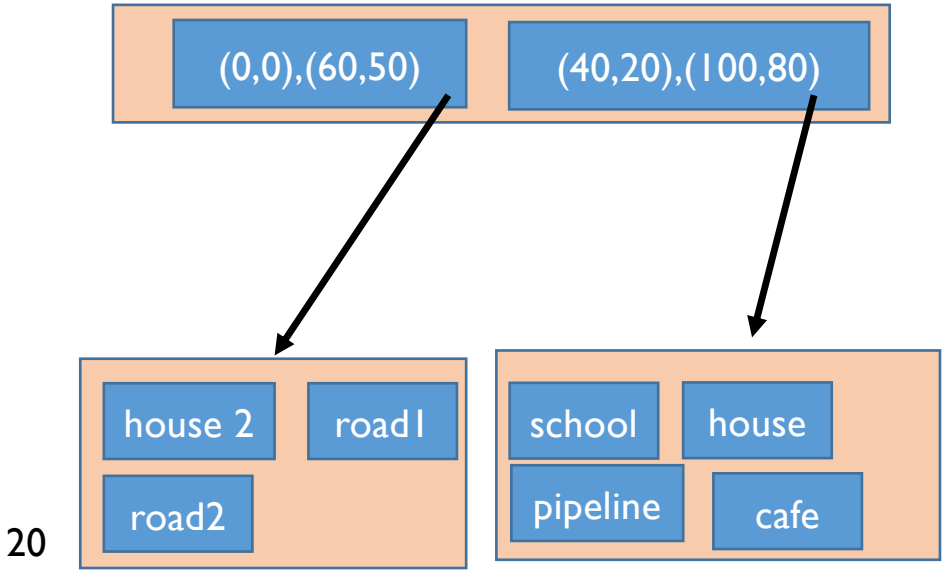
Splitting a Node During Insertion

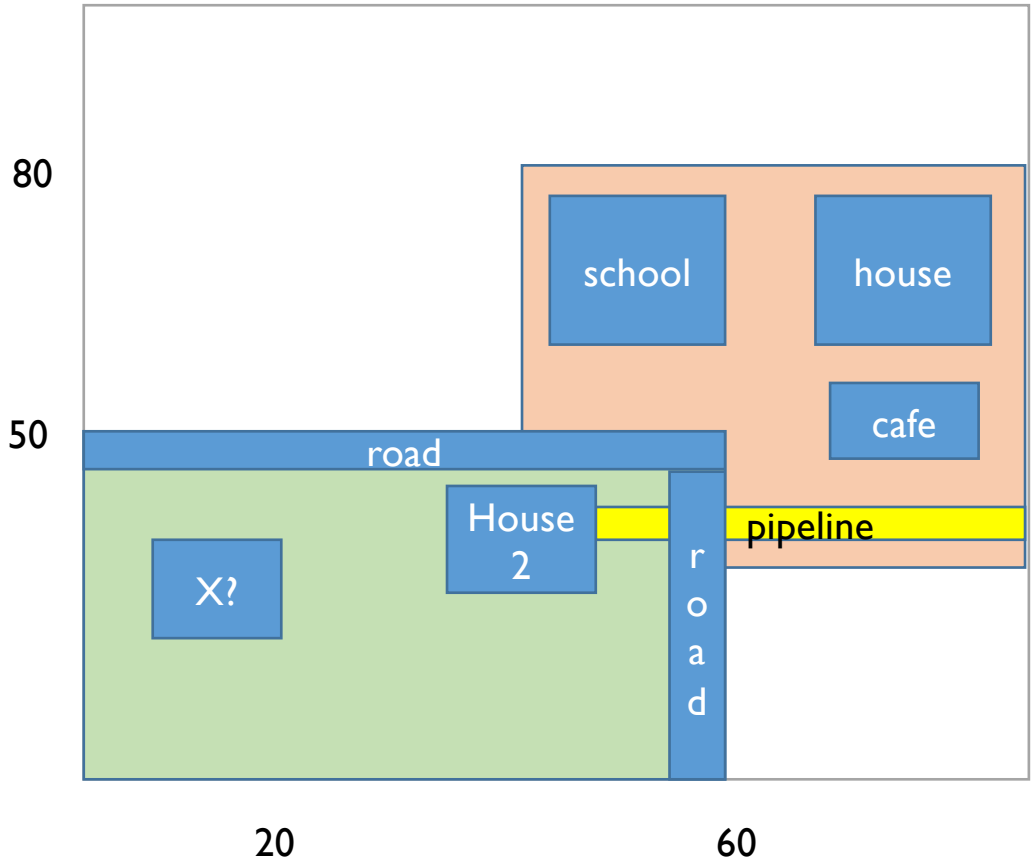
- The entries in node L plus the newly inserted entry must be distributed between L1 and L2.
- Goal is to reduce likelihood of both L1 and L2 being searched on subsequent queries.
- Idea: Redistribute so as to minimize area of L1 plus area of L2.

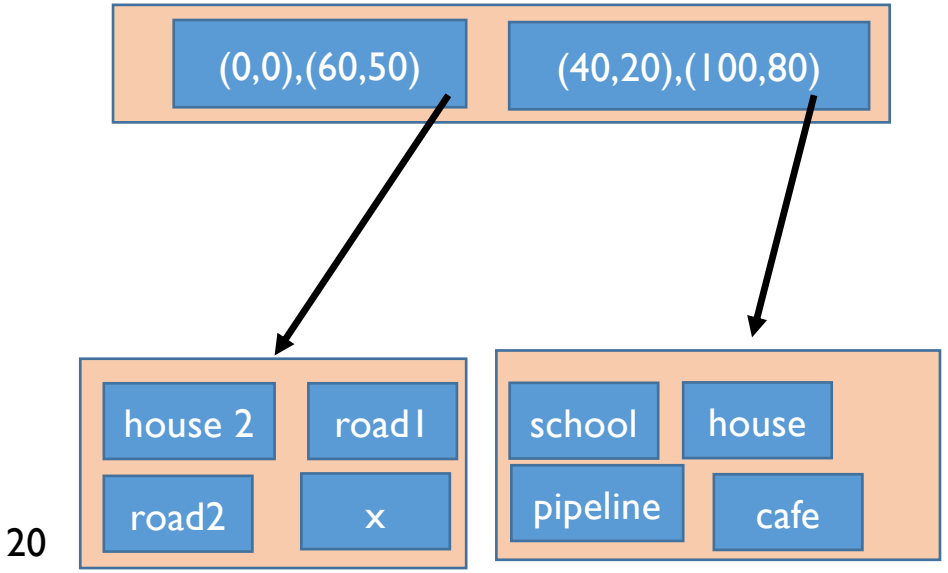
Exhaustive algorithm is too slow; quadratic and linear heuristics are described in the paper.

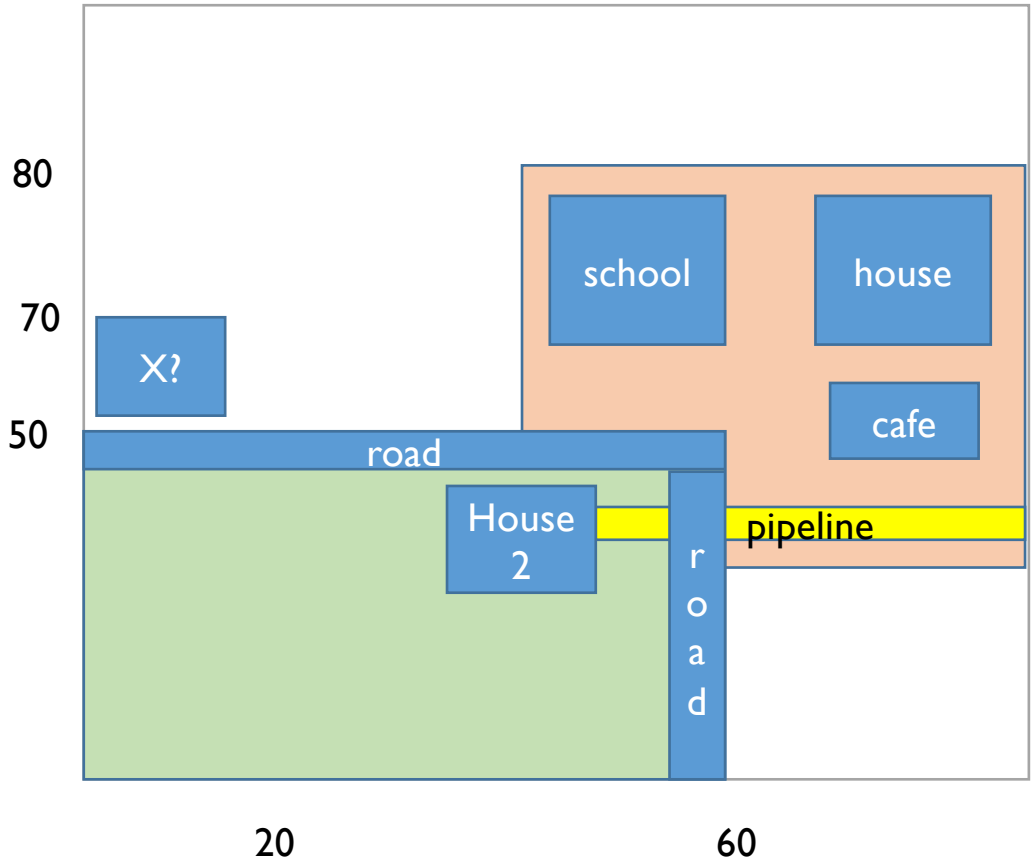


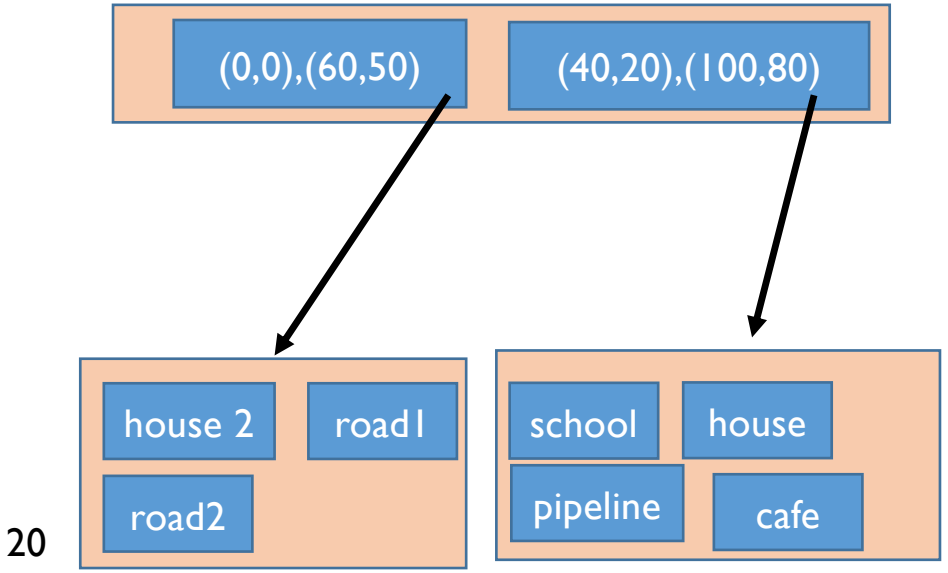

80
50
school
house
cafe
road 1
House 2
pipeline
X?
r o a d 2
20
40
60

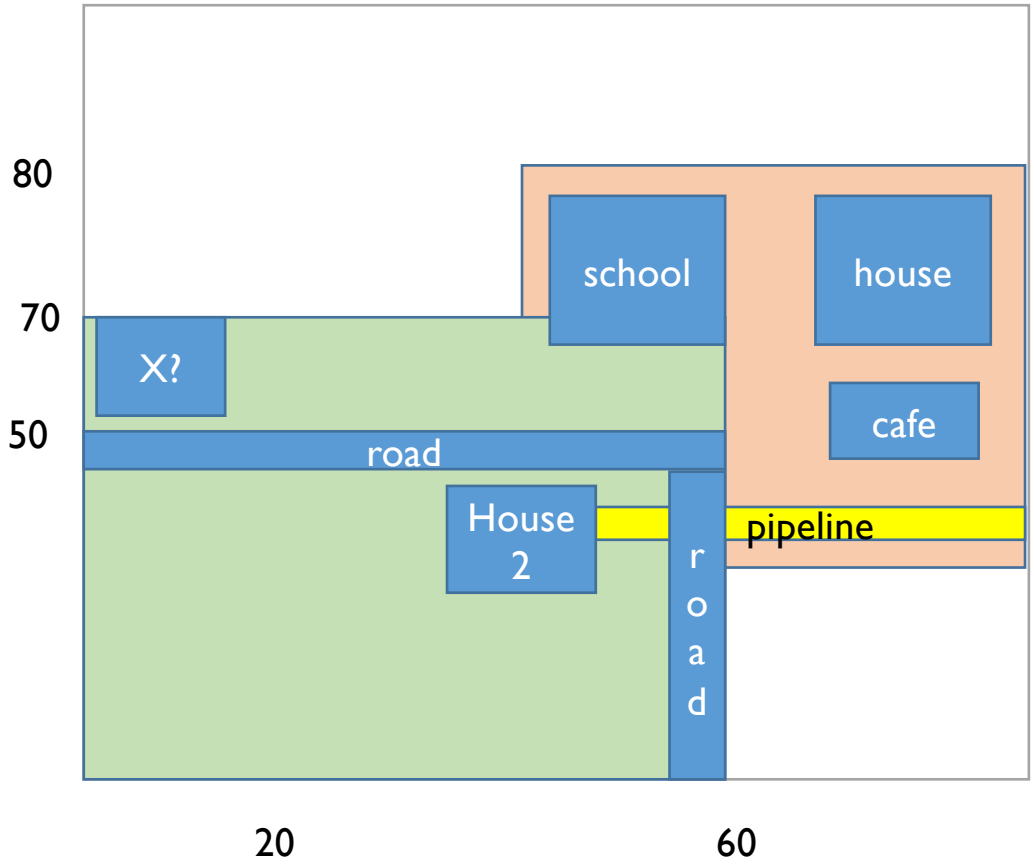

(0,0),(60,50)
(40,20),(100,80)
house 2
road1
road2
school
house
pipeline
cafe
20

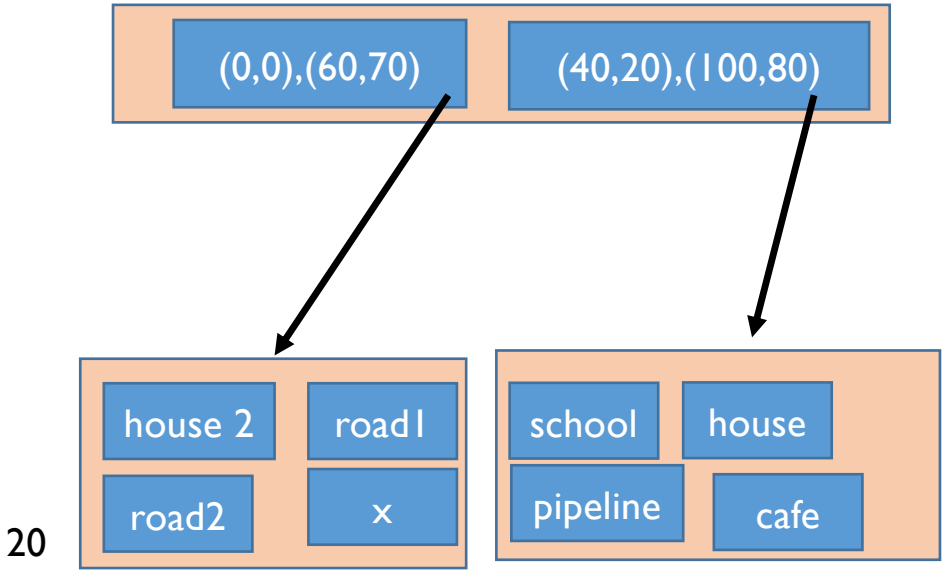

80
50
school
house
cafe
road
House 2
pipeline
X?
road
20
60

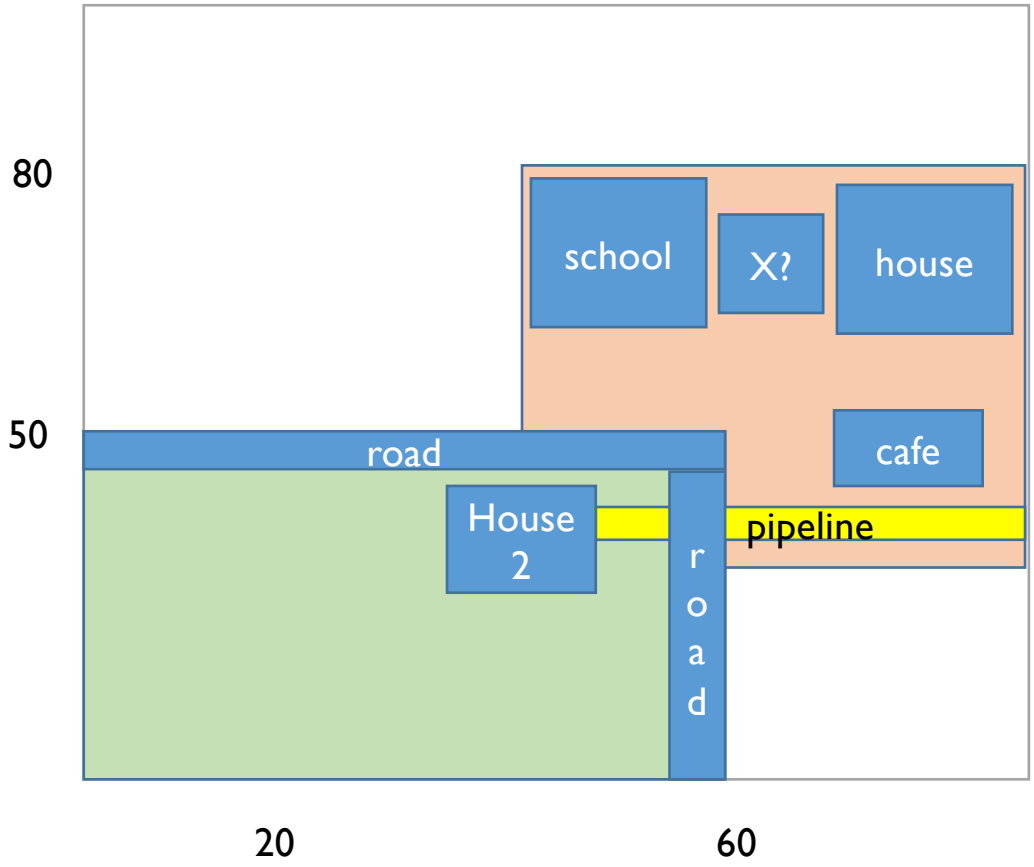

(0,0),(60,50)
(40,20),(100,80)
house 2
road1
road2
x
school
house
pipeline
cafe
20

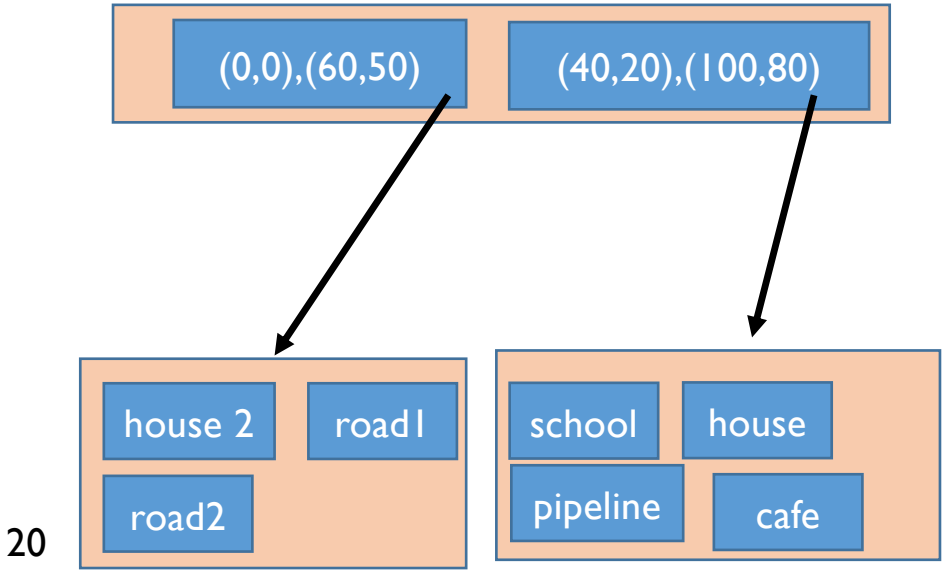

80
70
50
X?
school
house
cafe
road
House 2
pipeline
road
20
60

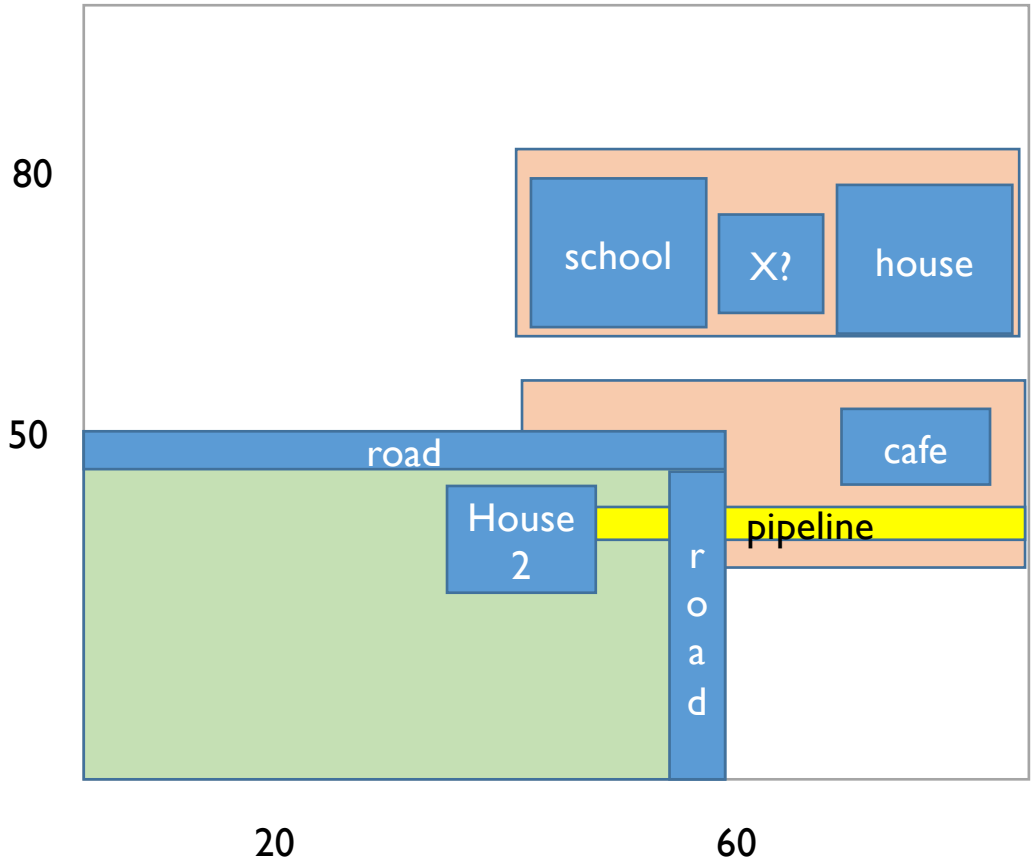

(0,0),(60,50)
(40,20),(100,80)
house 2
road1
road2
school
house
pipeline
cafe
20

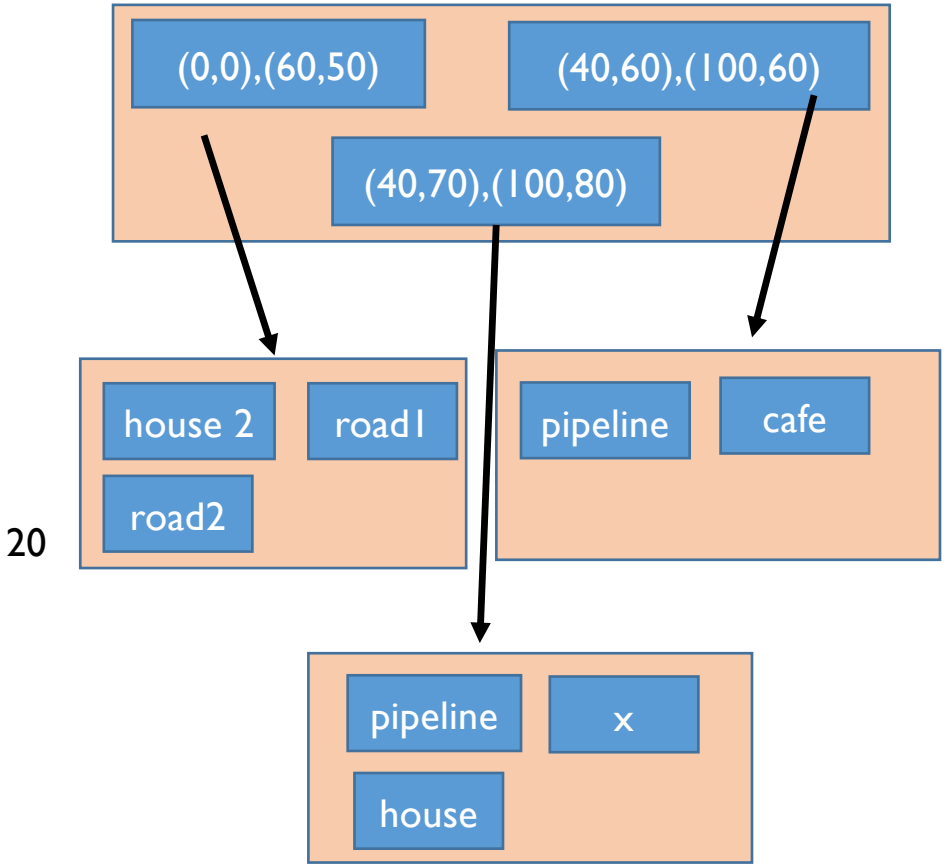

80
70
50
X?
school
house
cafe
road
House 2
road
pipeline
20
60


(0,0),(60,70)
(40,20),(100,80)
house 2
road1
road2
x
school
house
pipeline
cafe
20


80
50
school
X?
house
cafe
road
House
2
pipeline
road
20
60


(0,0),(60,50)
(40,20),(100,80)
20
house 2
road1
road2
school
house
pipeline
cafe


80
50
school
X?
house
cafe
road
House 2
pipeline
road
20
60


(0,0),(60,50)
(40,60),(100,60)
(40,70),(100,80)
20
house 2
road1
road2
pipeline
cafe
pipeline
x
house

R-Tree Variants

- The R* tree uses the concept of forced reinserts to reduce overlap in tree nodes. When a node overflows, instead of splitting:
  - Remove some (say, 30% of the) entries and reinsert them into the tree.
  - Could result in all reinserted entries fitting on some existing pages, avoiding a split.

- R* trees also use a different heuristic, minimizing box perimeters rather than box areas during insertion.

Indexing High-Dimensional Data

- Typically, high-dimensional datasets are collections of points, not regions.
  - E.g., Feature vectors in multimedia applications.
  - Very sparse
- Nearest neighbor queries are common.
  - R-tree becomes worse than sequential scan for most datasets with more than a dozen dimensions

Comments on R-Trees

- Deletion consists of searching for the entry to be deleted, removing it, and if the node becomes under-full, deleting the node and then re-inserting the remaining entries.

- Overall, works quite well for 2 and 3 D datasets. Several variants (notably, R+ and R* trees) have been proposed; widely used.

- Can improve search performance by using a convex polygon to approximate query shape (instead of a bounding box) and testing for polygon-box intersection.