Data Intensive Systems (DIS)
KBH-SW7 E25

11. Spatial Data and R-tree

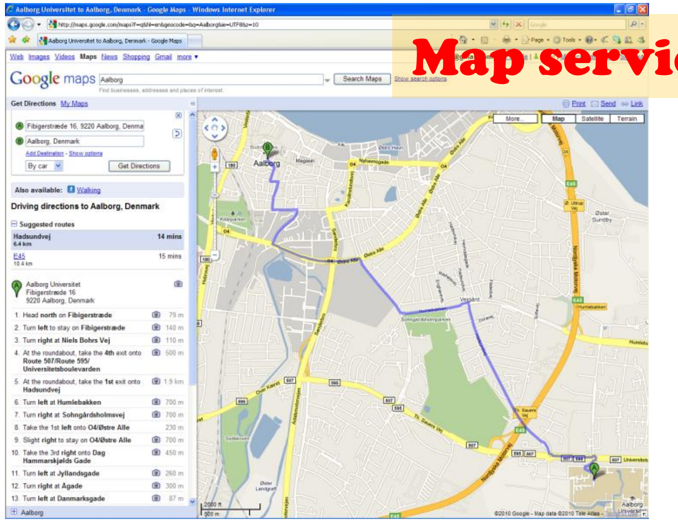
AALBORG UNIVERSITY

Agenda

- Introduction
  - Spatial data model
    - Data types and spatial relationships
  - Spatial queries
- R-tree
- R-tree Variants

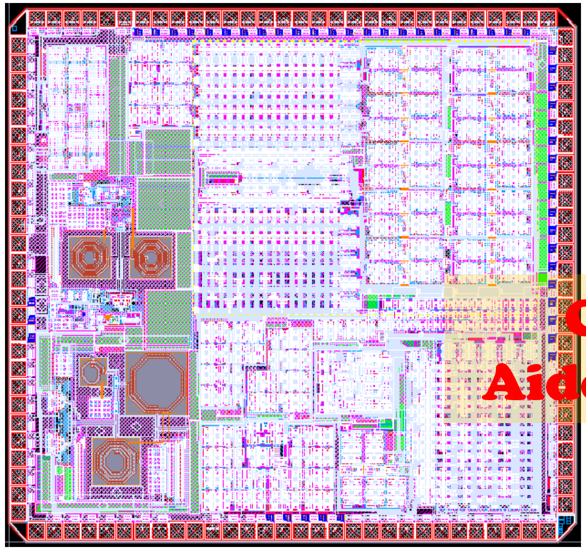
Motivation

- Application scenarios



- Data with locations
  - Special kinds of data

Spatial Data



Data Categorization

- Data
  - Nonspatial data
    - Numeric
    - Text
    - …
  - Spatial data
    - Raster data
      - E.g., remote sensing images
    - Vector data
      - E.g., points, (poly)lines, polygons
    - Graph data
      - E.g., road networks

Relational DB

Multimedia DB

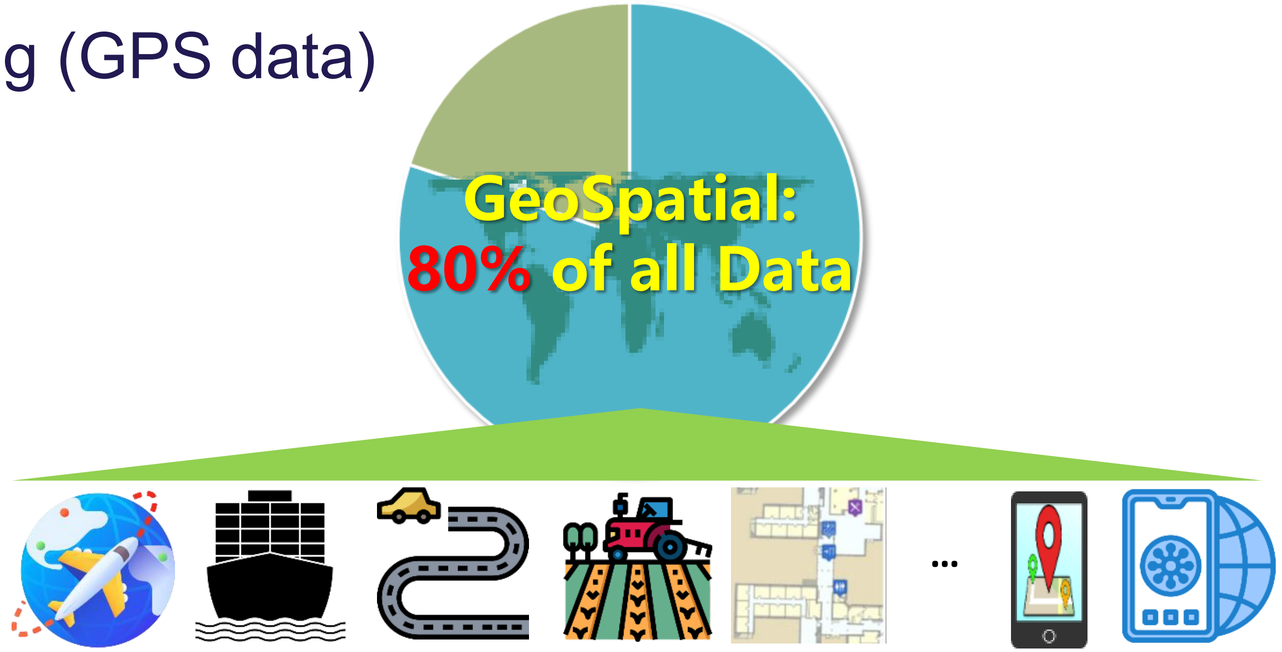
NoSQL

Array DB

Spatial DB

Big Spatial Data

- Air flight tracking
- Ship tracking (AIS data)
- Vehicle tracking (GPS data)
- …

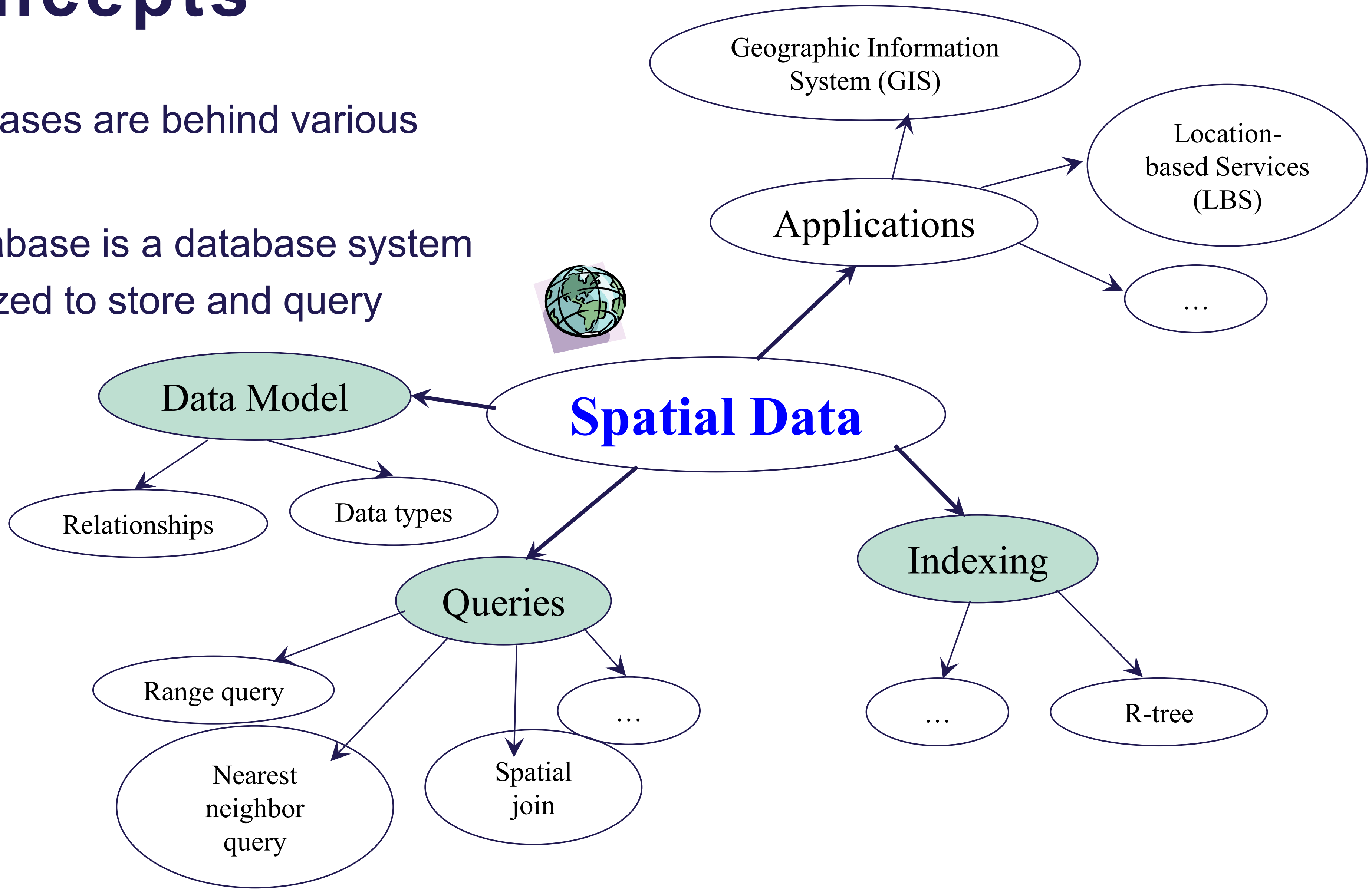


Spatial Data Management

- How to manage spatial data that is everywhere?
  - Modelling, storage, access and process
- Data files
  - No guarantee for logical and physical data independences
- Traditional databases
  - No capabilities for efficient spatial data management
- Spatial Databases
  - For representing, storing, querying and mining spatial data for multiple domains.
  - Advantages of *traditional databases* + *specializations* for spatial data

The Concepts

- Spatial databases are behind various applications.
- A spatial database is a database system that is optimized to store and query **spatial data**.



Spatial Data Management

- A **spatial object** has some spatial attributes that describe its location and geometry
  - Typically 2-dimensional or 3-dimensional data
- A **spatial relation** (dataset, or table) is a collection of spatial objects of the *same* type
  - E.g., rivers, cities, roads
- An example

Road segments of a region in CA

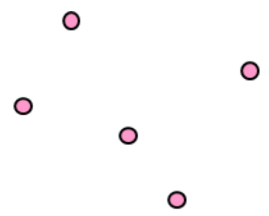
A particular data type supported by a spatial database

ID	Name	Type	Polyline
1	Boulevard	avenue	(10023,1094), (9034,1567), (9020,1610)
2	Leeds	highway	(4240,5910), (4129,6012), (3813,6129), (3602,6129)
…	…	…	…

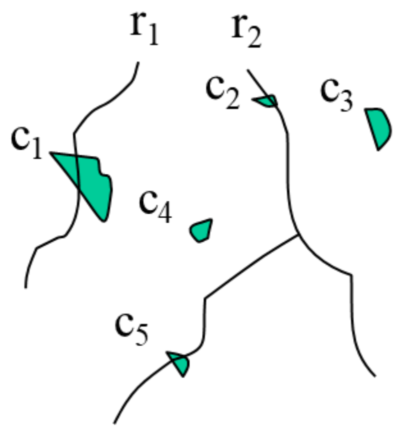
A spatial relation

Fundamental Spatial Object Types

- Points
  - Objects whose dimensionality is not important
  - E.g., cities in a small-scale map
  - E.g., gas stations



- Line segments, and polylines
  - Objects whose lengths are of importance
  - E.g., roads, rivers
- Polygons
  - Objects with extent
  - E.g., forests, districts, a city area.

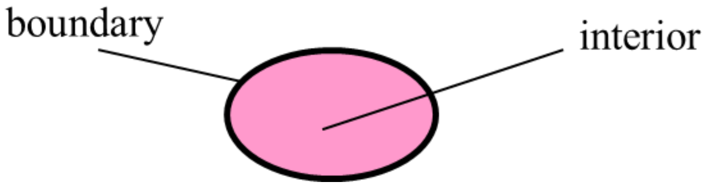


Spatial Relationships

- A spatial relationship links two objects according to their relative location and extent in space
  - Example: “My apartment is close to the little mermaid”
- To be distinguished from “spatial relation”, which means a dataset of spatial objects
- Types of spatial relationships
  - *topological* relationships
  - *distance* relationships
  - *directional* relationships

Topological Concepts

- Each object is characterized by the space it occupies in the universe.
  - a (finite or infinite) set of pixels
- Each object has a boundary and an interior
  - boundary: the set of pixels the object occupies, that are adjacent to at least one pixel not occupied by the object
  - interior: the set of pixels occupied by the object, which are not part of its boundary



- Some objects may not have interior
  - E.g., points, line segments

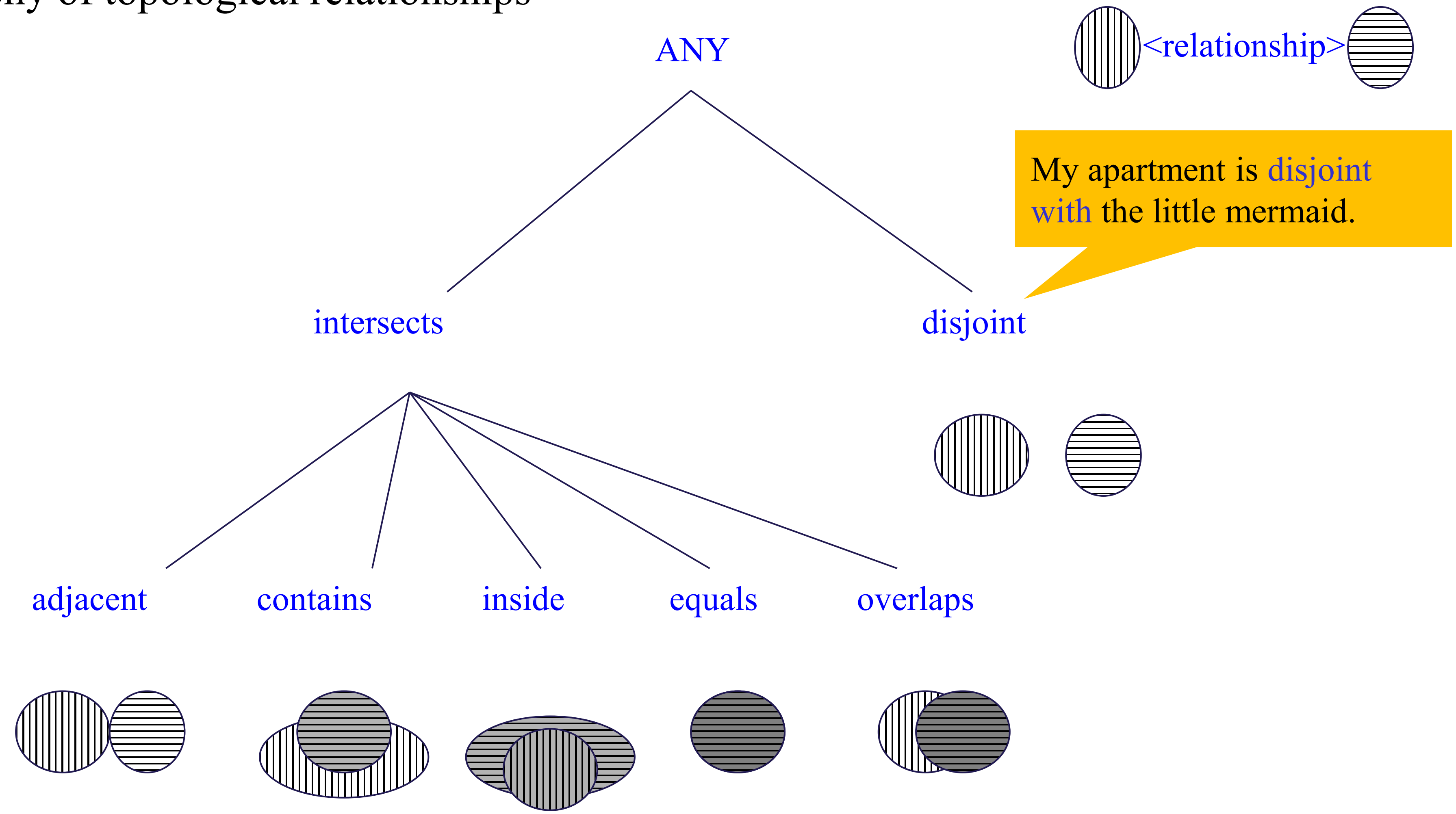
Topological Relationships

- A topological relationship between *two* objects is defined by a set of (set-based) relationships between their boundaries and interiors
  - E.g., o_1 is inside o_2 if interior(o_1) $\subset$ interior(o_2)
- 7 types of topological relationships
  - *intersects* means any of *equals*, *inside*, *contains*, *adjacent*, *overlaps*
  - *intersects* $\Leftrightarrow$ $\neg$*disjoint*

Topological relationship	equivalent boundary/interior relationships
$disjoint(o_1, o_2)$	$(interior(o_1) \cap interior(o_2) = \emptyset) \wedge (boundary(o_1) \cap boundary(o_2) = \emptyset)$
$intersects(o_1, o_2)$	$(interior(o_1) \cap interior(o_2) \neq \emptyset) \vee (boundary(o_1) \cap boundary(o_2) \neq \emptyset)$
$equals(o_1, o_2)$	$(interior(o_1) = interior(o_2)) \wedge (boundary(o_1) = boundary(o_2))$
$inside(o_1, o_2)$	$interior(o_1) \subset interior(o_2)$
$contains(o_1, o_2)$	$interior(o_2) \subset interior(o_1)$
$adjacent(o_1, o_2)$ (or $meets(o_1, o_2)$)	$(interior(o_1) \cap interior(o_2) = \emptyset) \wedge (boundary(o_1) \cap boundary(o_2) \neq \emptyset)$
$overlaps(o_1, o_2)$	$(interior(o_1) \cap interior(o_2) \neq \emptyset) \wedge (\exists p \in o_1 : p \not\subseteq interior(o_2) \wedge p \not\subseteq boundary(o_2))$ $\wedge (\exists p \in o_2 : p \not\subseteq interior(o_1) \wedge p \not\subseteq boundary(o_1))$

Topological Relationship Hierarchy

- Hierarchy of topological relationships

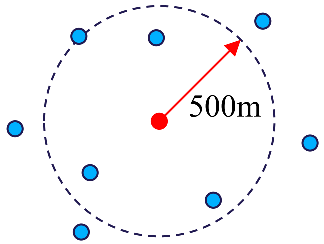


Spatial Queries

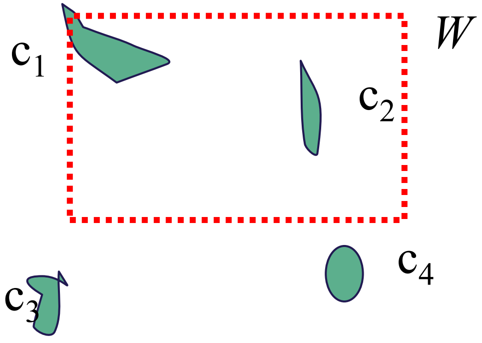
- A spatial query is a type of database query
  - Applied on one (or more) spatial relation/table(s)
  - To retrieve spatial objects (or combinations of spatial objects) that satisfy some user-specified spatial relationships
    - with a reference query object, or
    - between two spatial relations
- Typical spatial query types
  - Range query (*with a query object*)
  - Nearest neighbor query (*with a query object*)
  - Spatial join (*between two spatial relations*)

Range Query

- A.k.a spatial selection
- The query range can be a window or a circle (or a distance)
  - Window: top-left and bottom-right corners
  - Circle: a query point as the center, and a distance as the radius.
  - Different types of spatial relationships can be used.
- E.g., find all restaurants within *500 meters* from my apartment.
  - Depending on distance type, query result can differ.
  - E.g., in a city Manhattan distance makes more sense



- E.g., find all cities that *intersect* window *W*
  - *Result set*: $\{c_1, c_2\}$
  - If a different predicate is used, the query result could be different.
    - E.g., all cities contained in *W*



Spatial Query Processing

- The process of finding the result for a given spatial query.
- The process can be time-consuming if no appropriate data structure or algorithm is used.
- A *naïve* query processing approach: **sequential scan**
  - Given a spatial relation *R* and a query *Q*, check each spatial object *o* in *R* to see if *o* satisfies the query condition in *Q*.
  - The query result is the *complete* set of all such objects.
  - This approach is time-consuming as every single object is checked.
- To speed up spatial query processing, we need spatial access methods.

Problem Setting

- Large spatial dataset(s), stored on disk
  - points, lines, polygons, ...
- We want to process different spatial queries efficiently
  - Range, NN, spatial join
  - In query processing, we need to load data into memory from disk
  - Query processing cost: IOs >> CPU time
- Object clustering: Spatially close objects are stored also in the same or adjacent disk pages
  - This is to reduce IO cost in query processing
  - This is also preferred in relational databases, but what's special about spatial data?

What Is Special About Spatial?

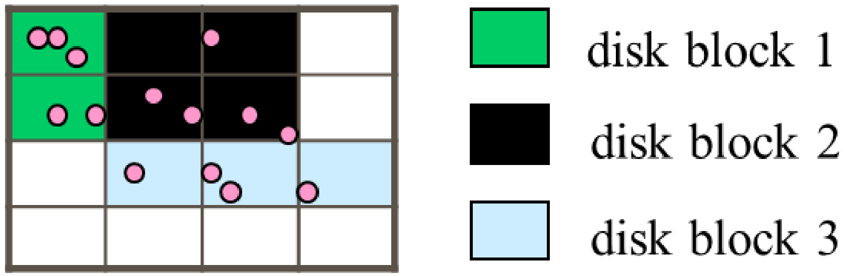
- Dimensionality
  - In the 2D or 3D space, There is no total ordering of objects that preserves spatial proximity
- Complex spatial extent
  - The spatial extent adds to the complexity of clustering objects into disk pages imposed by dimensionality
- Implication
  - Relational indexes (e.g. B+-trees) and query processing methods (e.g. sort-merge join, hash-join) are not readily applicable to spatial data
  - New, spatial access methods (SAMs) for spatial data have to be defined

Problem of Spatial Access Methods

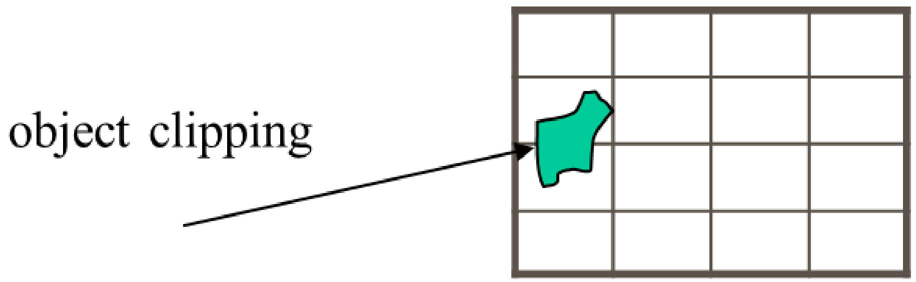
- How to index spatial data such that spatial queries can be answered correctly and efficiently?
  - **Correctly**: Query result should be the same as the counterpart sequential scan
  - **Efficiently**: Query processing should be fast, aiming at the minimization of the expected I/O cost
- Early SAMs index multidimensional points by dividing the space into *disjoint* partitions
  - Examples: the grid file, the k-d-B-tree
- Other SAMs partition data instead
  - E.g., R-tree

Point Access Methods

- Point access methods divide the space into disjoint partitions and group the points according to the regions they belong
  - E.g., the grid-file



- Drawback: point access methods are not effective for extended objects
  - Objects may be *clipped* into several parts ➔ data redundancy and affects performance negatively



AALBORG UNIVERSITET

Space Partitioning SAMs

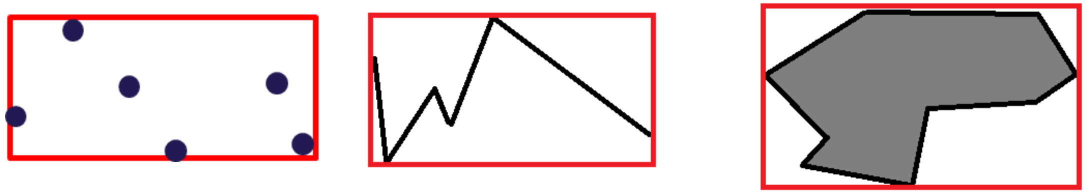
- Space partitioning methods can handle points well
- But they have difficulty in indexing other objects (e.g., polylines and polygons)
  - E.g., object clipping problem
- An alternative is to partition the data instead of the space
  - R-tree
    - 'R' for Rectangle
    - Based on MBR (minimum bounding rectangles)

Agenda

- Introduction
- R-tree
  - Basic idea and structure
  - Insertion and split
  - Deletion
- R-tree Variants

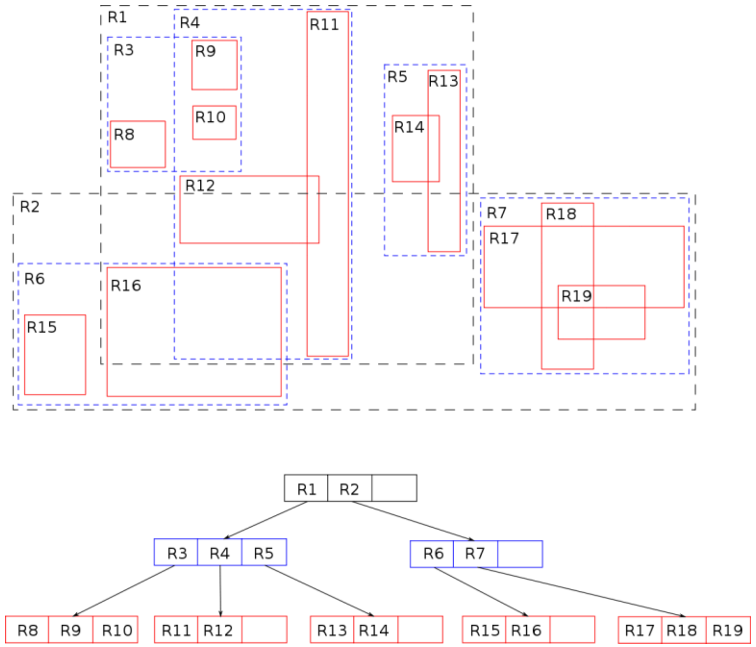
R-tree Concepts

- Minimum Bounding Rectangles (MBRs)



MBRs of points, a polyline and a polygon

- Tree Architecture



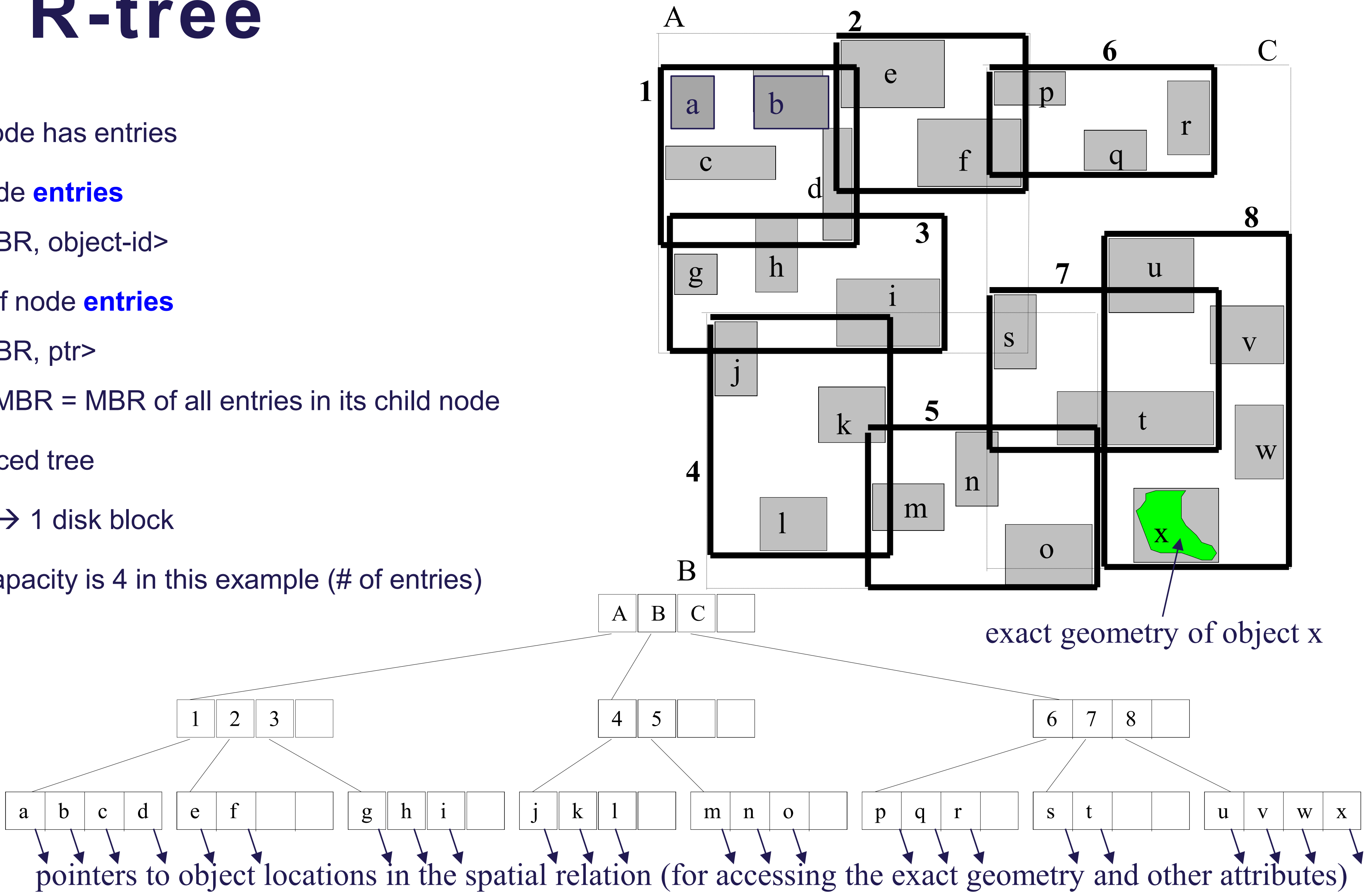
The R-tree

- Groups object MBRs to disk blocks hierarchically
- Each group of objects (a disk block) is a leaf of the tree
  - Objects in the same leaf node should be physically close together in one disk block
- The MBRs in a leaf node are grouped to form nodes at the next upper level
  - MBRs close together are grouped
- Grouping is recursively applied at each level until a single group (the root) is formed

- A generalization of the B^+-tree

The R-tree

- Each node has entries
- Leaf node **entries**
  - <MBR, object-id>
- Non-leaf node **entries**
  - <MBR, ptr>
  - Its MBR = MBR of all entries in its child node
- A balanced tree
- 1 node → 1 disk block
- Node capacity is 4 in this example (# of entries)



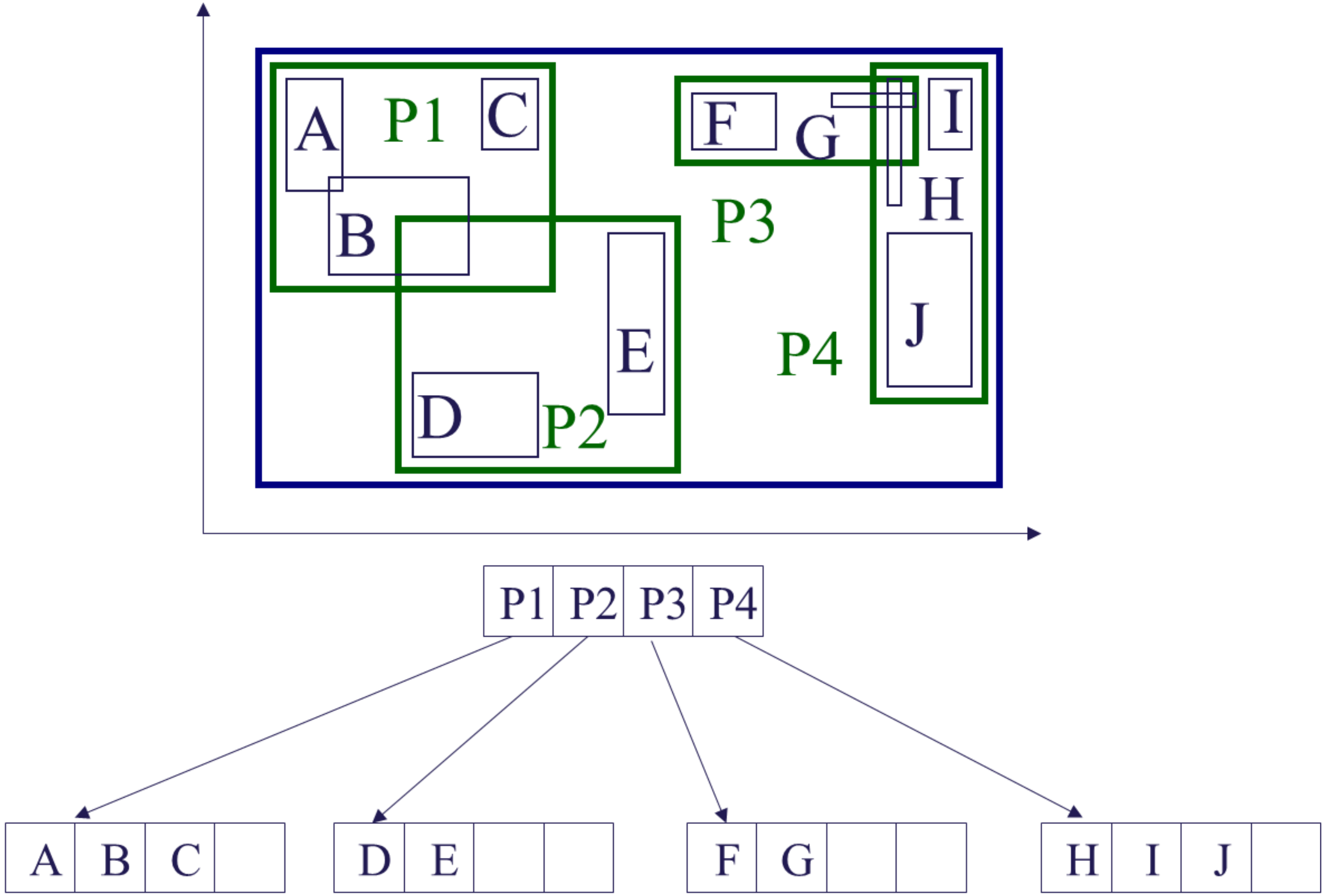
R-tree Properties

- A multi-way external memory tree
- Index nodes and data (leaf) nodes
- All leaf nodes are on the same level
- Every node contains between **m** (≤M/2) and **M** entries.
  - Sometimes M is a.k.a fanout (F).
- The root node has at least 2 entries (children)

- Every parent node completely covers all its ‘children’ in terms of MBRs
- A child MBR may be covered by also its parent’s sibling. However, it is stored under its parent.

Example

- Node capacity M = 4



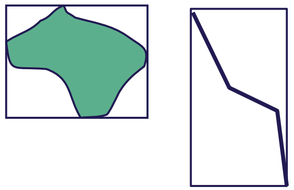
R-tree Family

- R-tree
  - The fundamental one
- R*-tree
  - Optimized insertion and deletion algorithms
- Search algorithms are the same to R-tree and R*-tree
- R^+-tree
  - Object splitting to avoid overlap between MBRs
  - Different algorithms are used compared to the former two

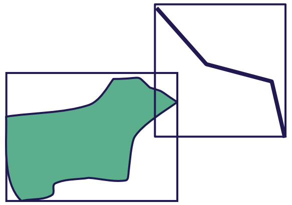
Two-step Spatial Query Processing

- Evaluating spatial relationships on geometric data is slow
- A spatial object is approximated by its MBR
- A spatial query is usually processed in two steps:
  - 1. **Filter step**: The MBR is tested against the query predicate. → Fast!
    Often, a particular spatial index is used in this step.
  - 2. **Refinement step**: The exact geometry of objects that pass the filter step is tested for qualification → Slow!

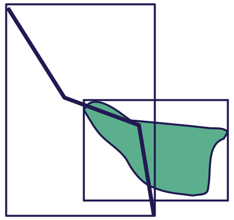
- Example: spatial intersection joins between forests and rivers



filtered pair



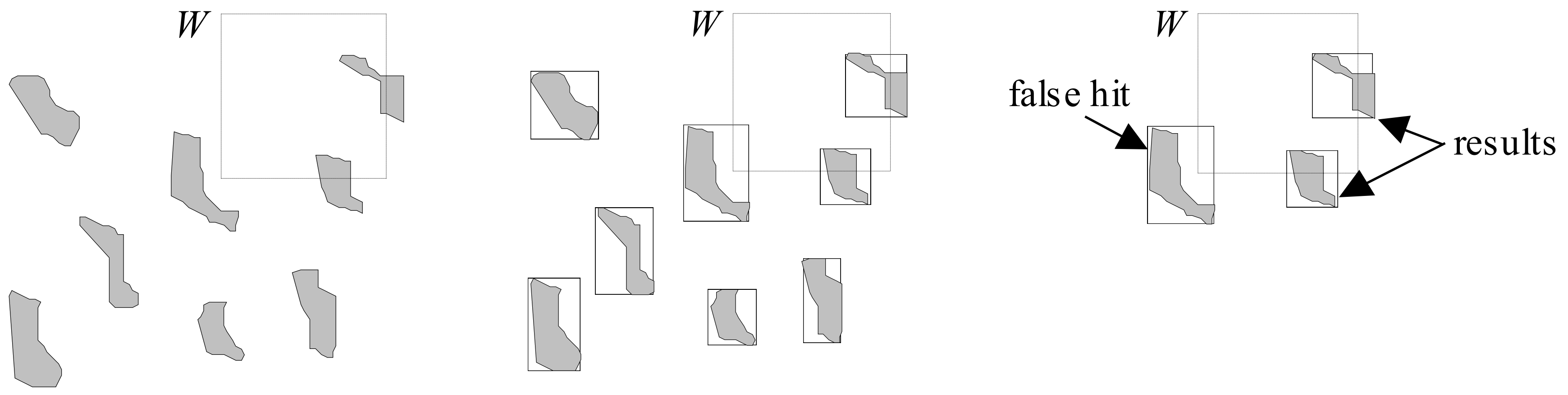
non-qualifying pair that passes the filter step (false hit or false positive)



qualifying pair

Two-step Range Query Processing

- Problem: Find all objects that intersect the query window *W*
- Comparison cost ≈ number of examined vertices
- Suppose that each polygon has 100 vertices
  - Comparisons in Figure (a) ≈ 100*8 = 800
  - Comparisons in Figure (b) ≈ 4*8 + 100*3 = 332



(a) objects and a query (b) object MBRs (c) candidates and results

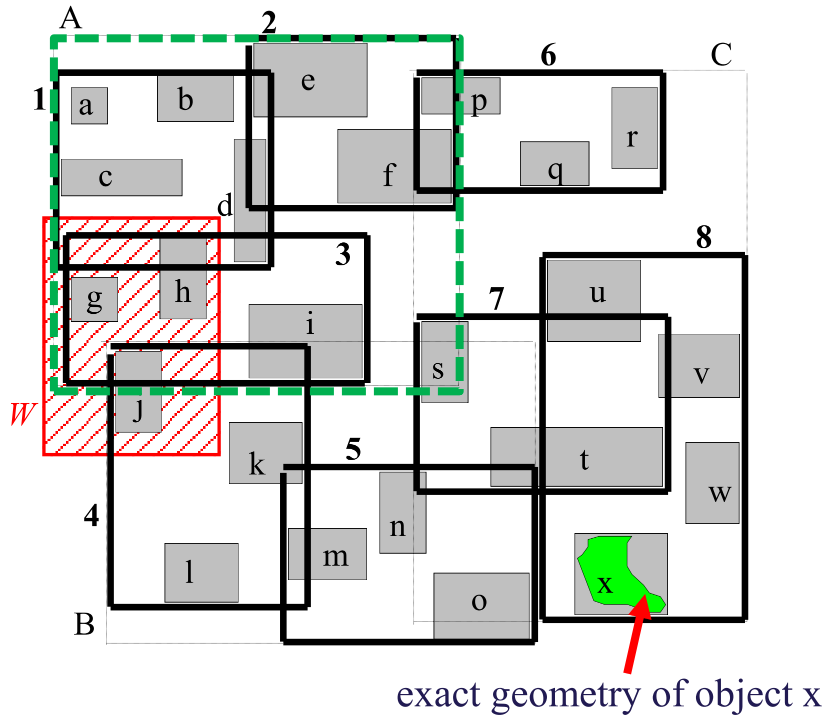
Range Search via R-tree

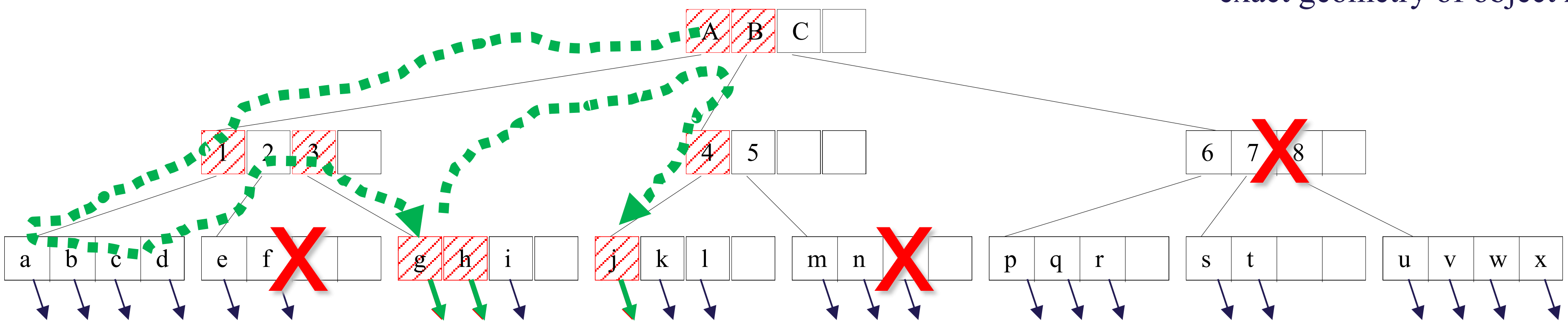
- **Range_query**(query *W*, R-tree node *n*):
  - If *n* is not a leaf node
    - For each index entry *e* in *n* such that e.MBR intersects *W* → Filter
      - visit node *n′* pointed by *e.ptr*
      - **Range_query**(*W*, *n´*)
  - If *n* is a leaf
    - For each index entry *e* in *n* such that e.MBR intersects *W* → Refinement
      - visit object *o* pointed by *e.object-id*
      - test range query against exact geometry of *o*; if *o* intersects W, report *o*

- The recursive search starts from the R-tree root node
- May follow multiple paths during search

Range Search via R-tree

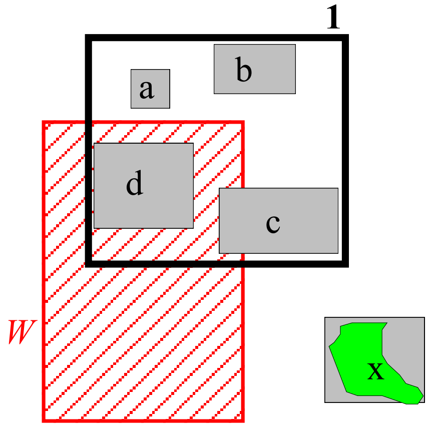
- *W* is a range query (window intersection query)
  - Multiple search paths are involved.
  - Many tree nodes (and thus objects) are pruned.
  - MBRs are used for filter
  - Exact geometries are fetched for refinement
    - E.g., object x





Range Search with Other Predicates

- The search algorithm considers the intersection predicate, i.e., finding all objects intersecting the query window W
- Modify the algorithm for other search predicates?
- E.g., find all objects inside W
  - Search predicate for the object o
    - inside(o, W)
  - Search predicate for a non-leaf MBR R
    - intersect(R, W). Why?
  - Search predicate for an object's MBR R'
    - inside(R', W).



Agenda

- R-tree
  - Basic idea and structure
  - Insertion and split
  - Deletion
- R-tree Variants

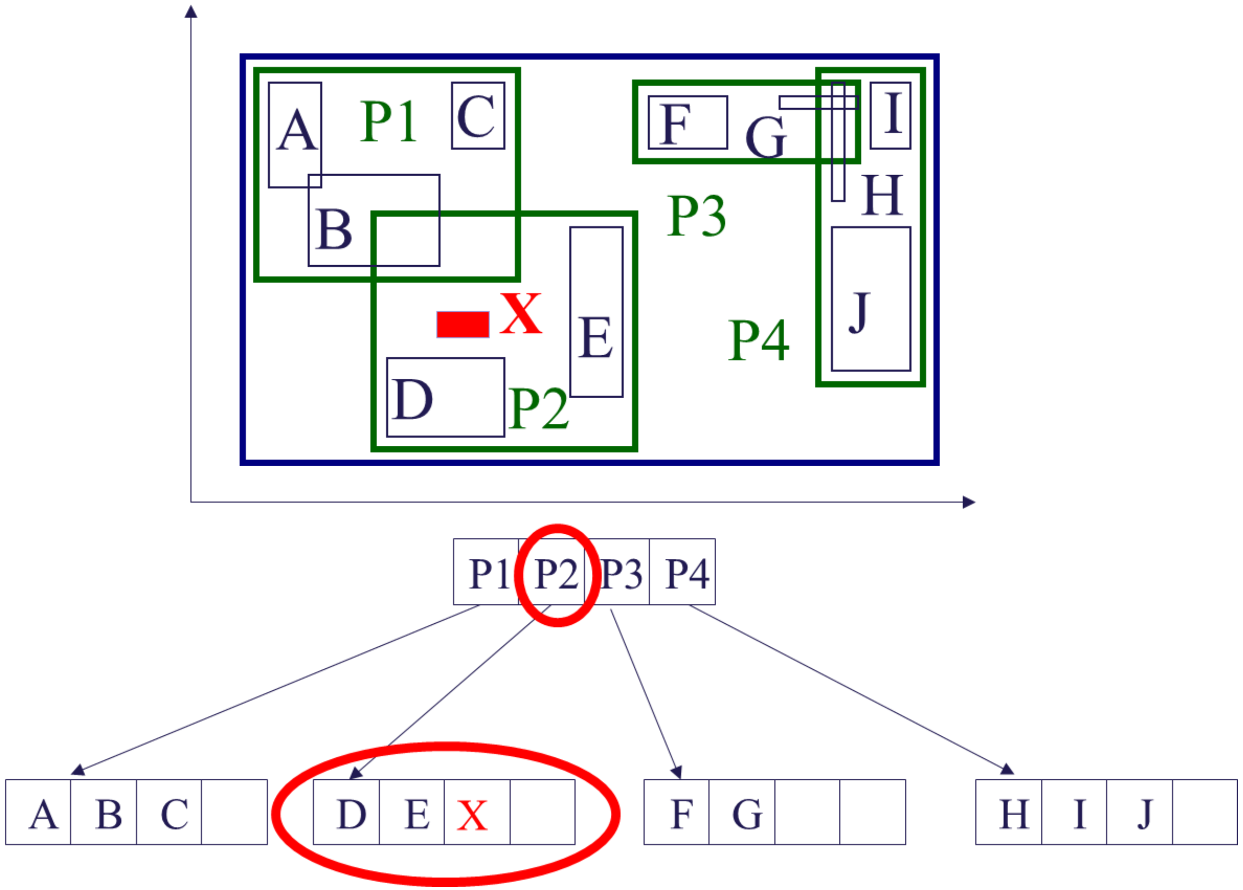
Insert Object *o* to R-tree

- Start at root and go down to "best-fit" leaf node *L*.
  - In each step downwards, go to the child whose MBR needs the least area enlargement to include *o*. Resolve ties by going to the smallest area child.
  - This is encapsulated in Algorithm **ChooseLeaf**
- If the 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 MBR now covers (only) *L1*.
  - Add an entry (in the parent node of *L*) for *L2*. (This could cause the parent node to recursively split.)
- Propagate changes upward by Algorithm AdjustTree.

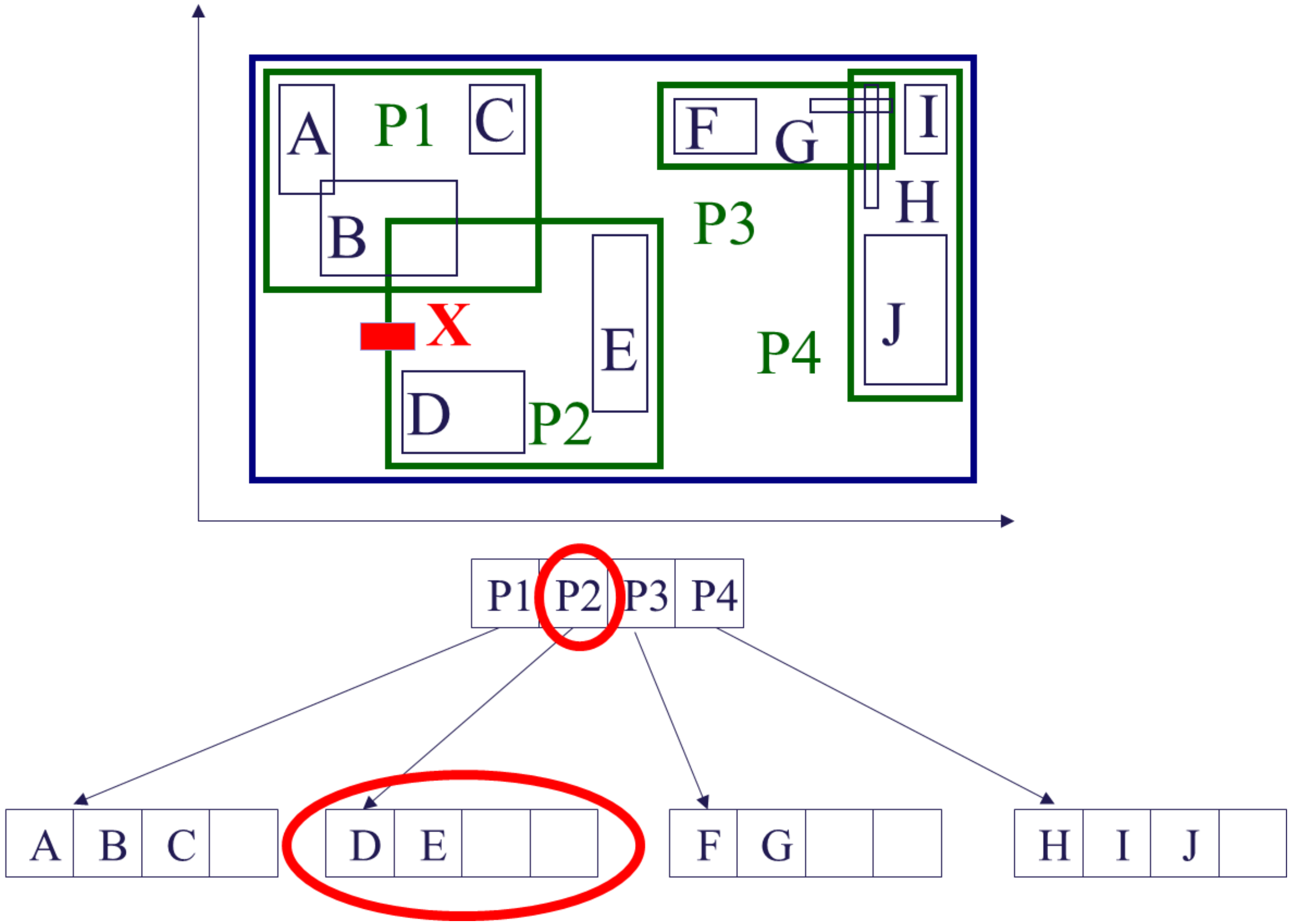
Antonin Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. ACM SIGMOD Conference 1984: 47-57

Insert: No Split, No Enlargement

- X is inserted to the child node that P2 points to.

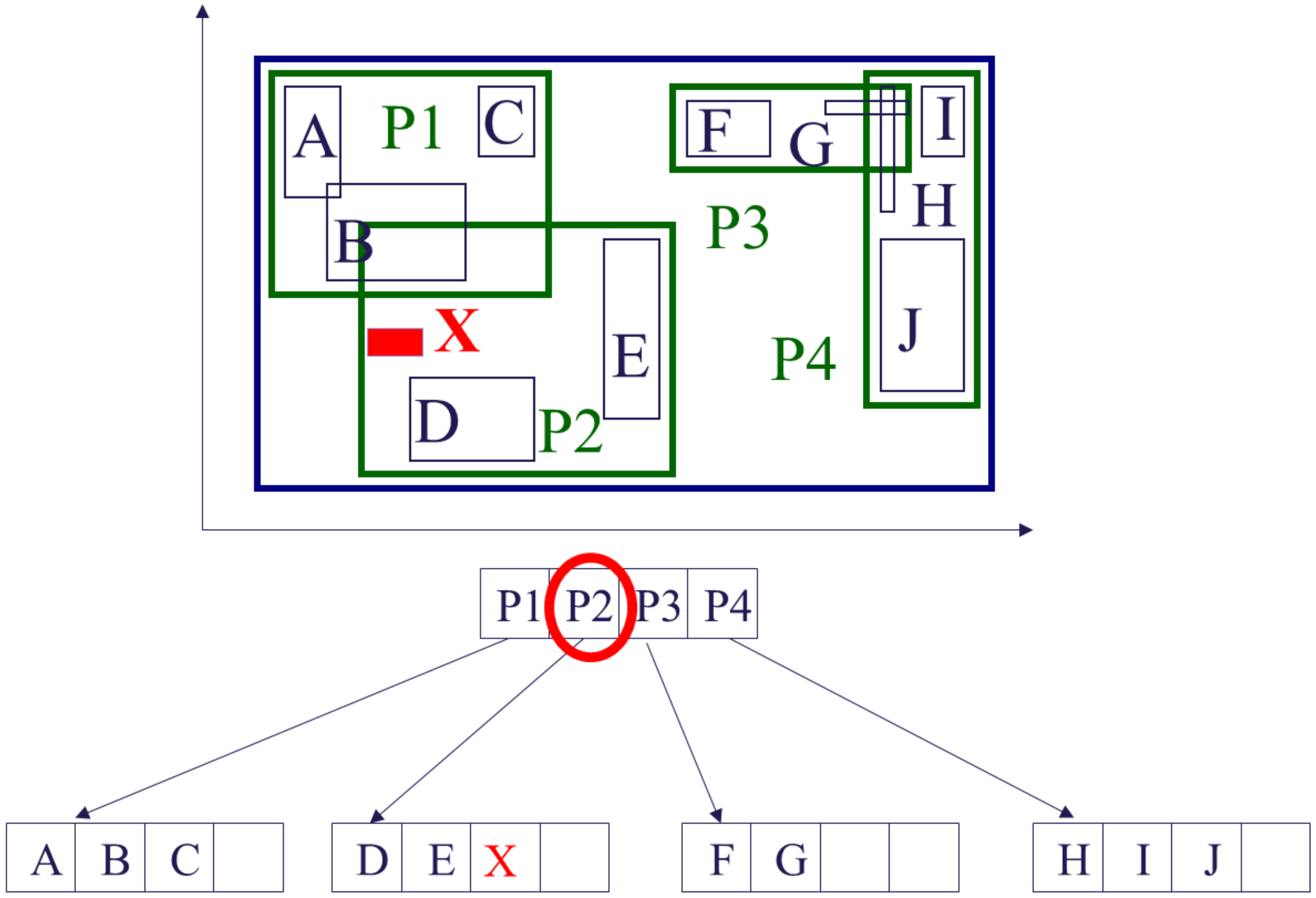


Insert: No Split, But Enlargement



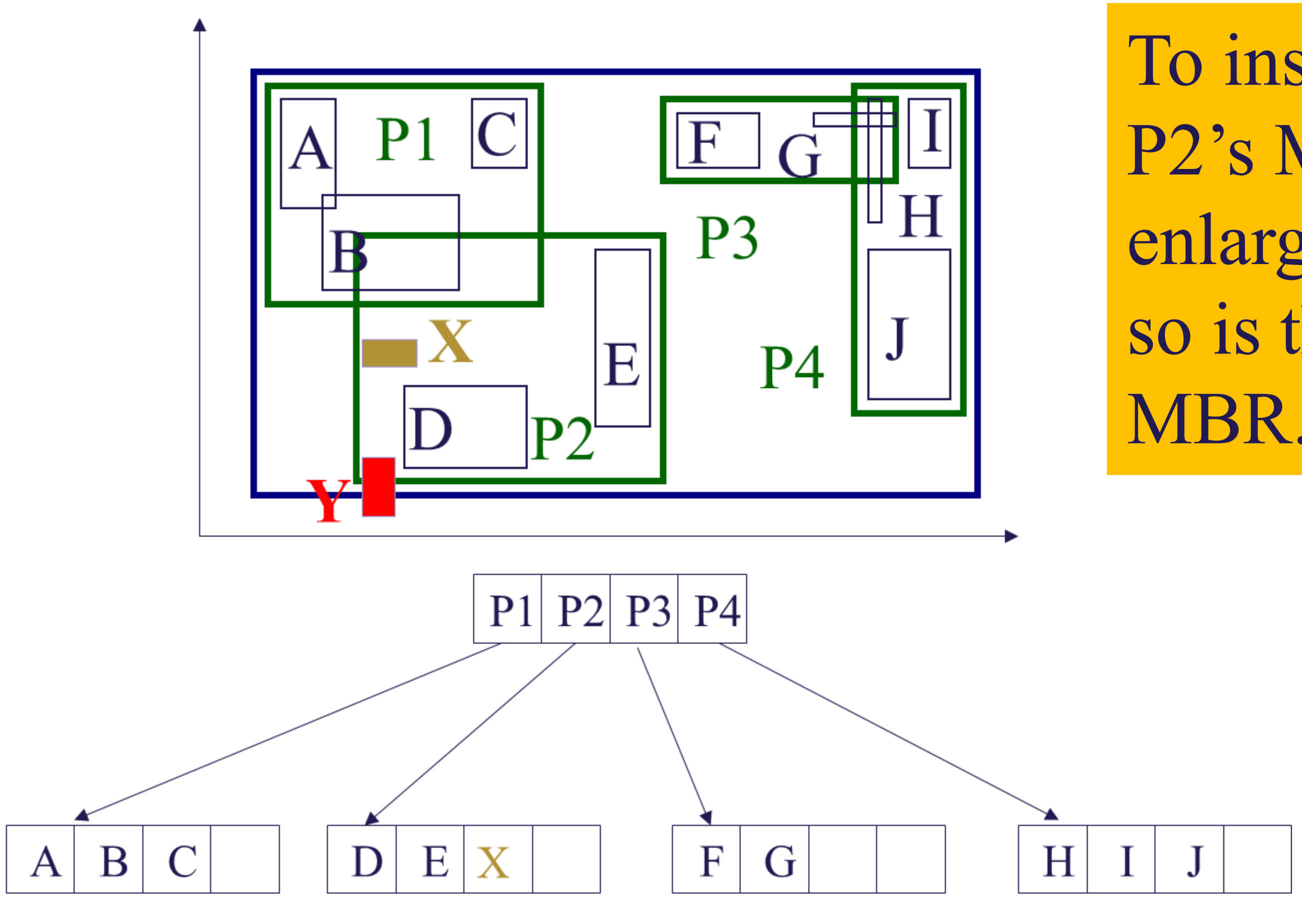
Insert: No Split, But Enlargement

- P2's MBR is enlarged.



Insert: No Split, But Enlargement

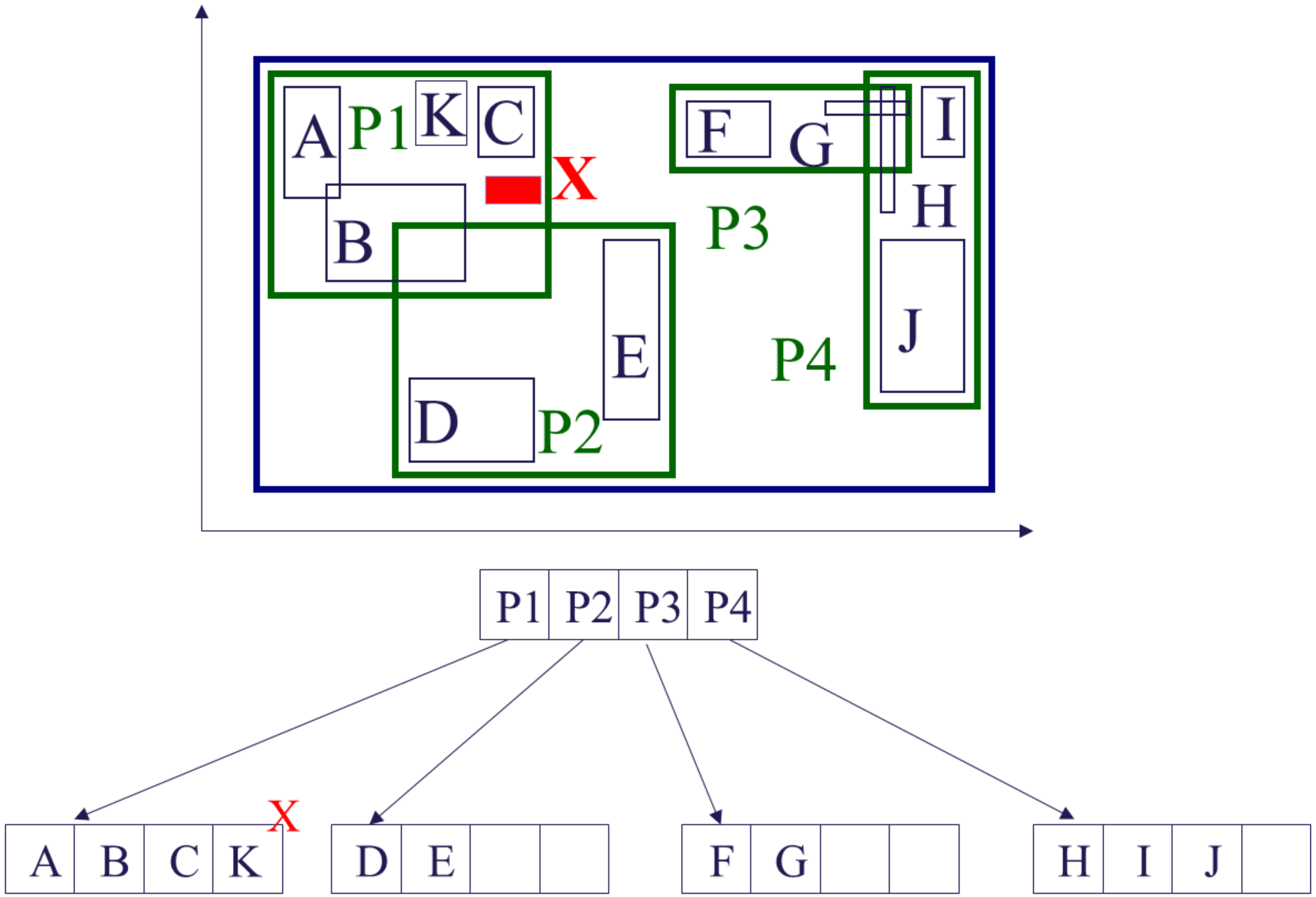
- Enlargement may propagate upward



To insert Y, P2's MBR is enlarged. And so is the overall MBR.

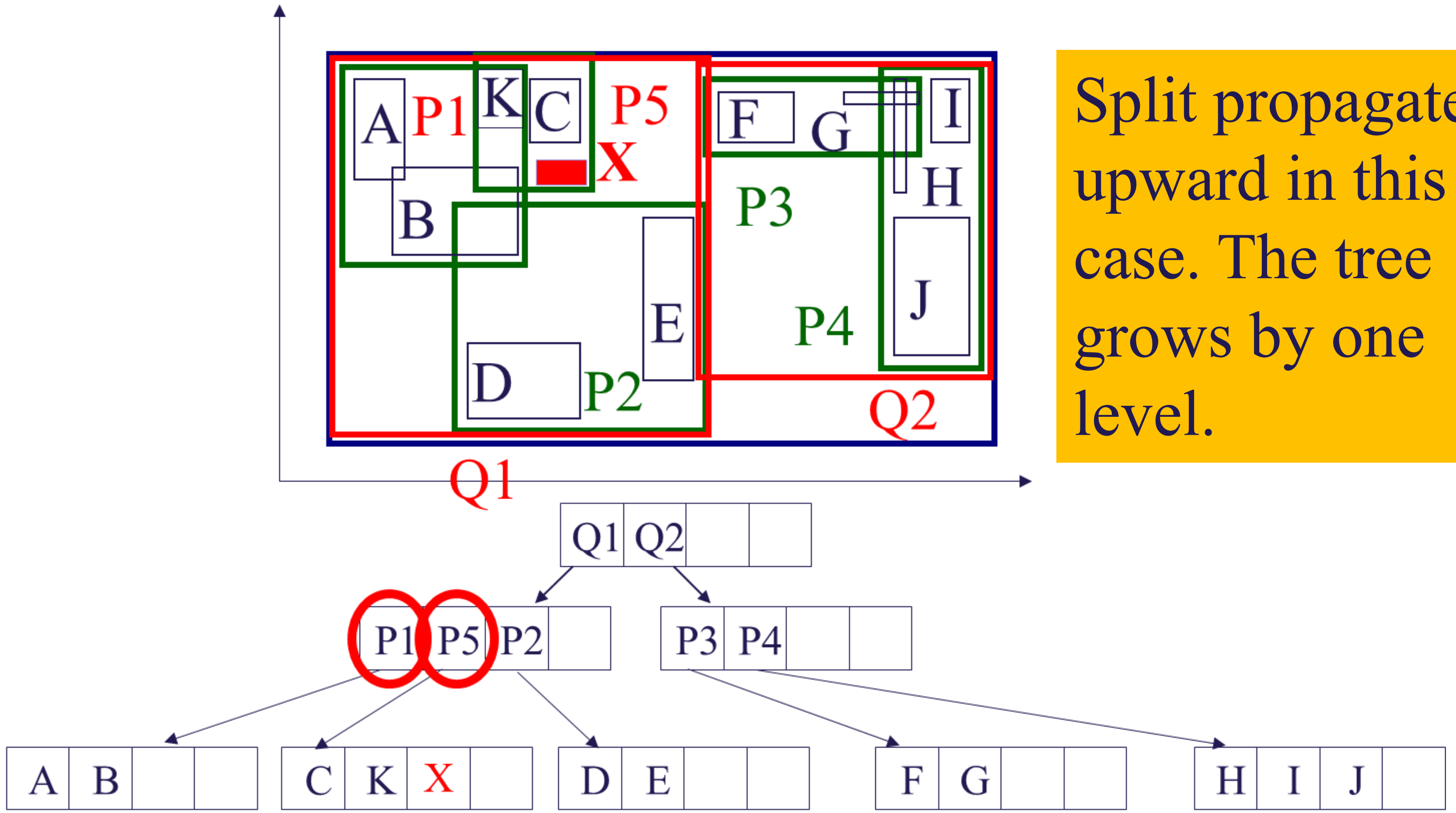
Insert: Split

- P1 is to be split when X is being inserted.



Insert: Split

- P1 is split to P1 and P5.

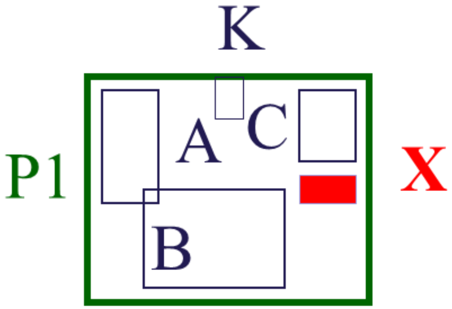


Split propagates upward in this case. The tree grows by one level.

Node Splitting in Insertion

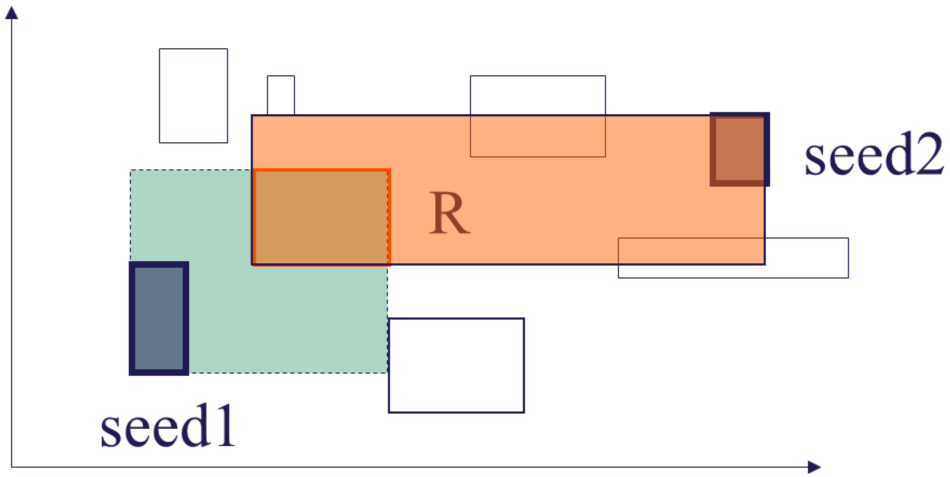
- The entries in node *L* plus the newly inserted entry must be (evenly) distributed between *L1* and *L2*.
- The goal is to reduce likelihood of both *L1* and *L2* being searched on subsequent queries.
- Idea: Redistribute so as to minimize the sum of *L1*'s area and *L2*'s area.
- Split node P1: partition the MBRs into two groups.
  - Option 1: exponential split; 2M-1 choices
  - Option 2: plane sweep, until 50% of MBRs
  - Option 3: quadratic split
  - Option 4: 'linear' split

Pick two seeds to initiate two groups.



R-trees: Split

- Pick two rectangles as ‘seeds’ and put them into two groups.
  - Option 3 and Option 4 differ only in this step.
    - Next slide
- Assign each MBR ‘R’ to its ‘closest’ ‘seed’.
  - ‘closest’: the smallest increase in area



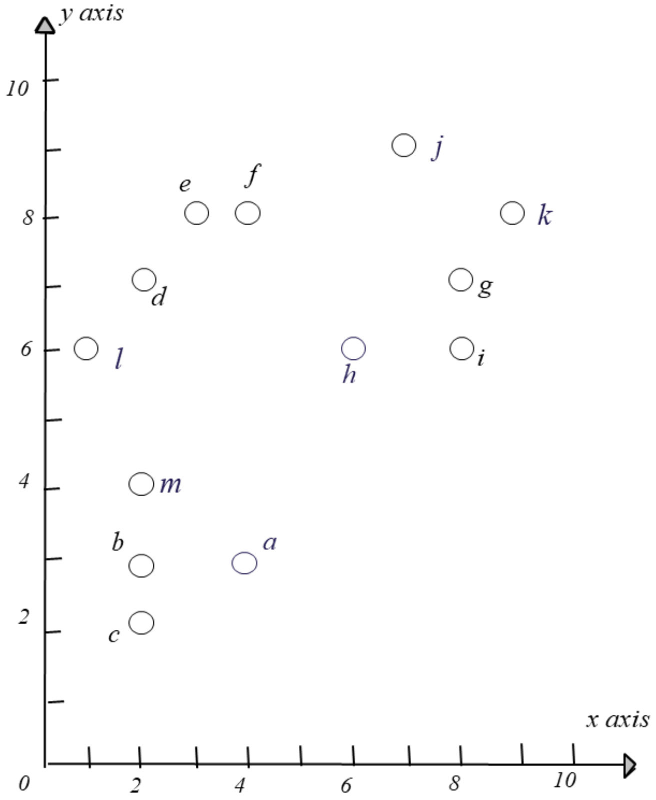
How to Pick Seeds in Split?

- Option 3: Quadratic split
  - For each pair E1 and E2, calculate
    - The rectangle **J = MBR(E1, E2)**
    - **d = area(J) - area(E1) - area(E2)**.
  - Choose the pair with the largest d.

- Option 4: Linear split
  - Along each dimension
    - Find the entry with the *highest low* side, and the one with the *lowest high* side.
    - Normalize the separation along the corresponding dimension.
  - Choose the pair with the greatest normalized separation.

Exercise

- Create an R-tree for the following data points.
  - Each node contains at most 3 entries.
  - Insertions are in alphabetic order



Agenda

- R-tree
  - Basic idea and structure
  - Insertion and split
  - Deletion
- R-tree Variants

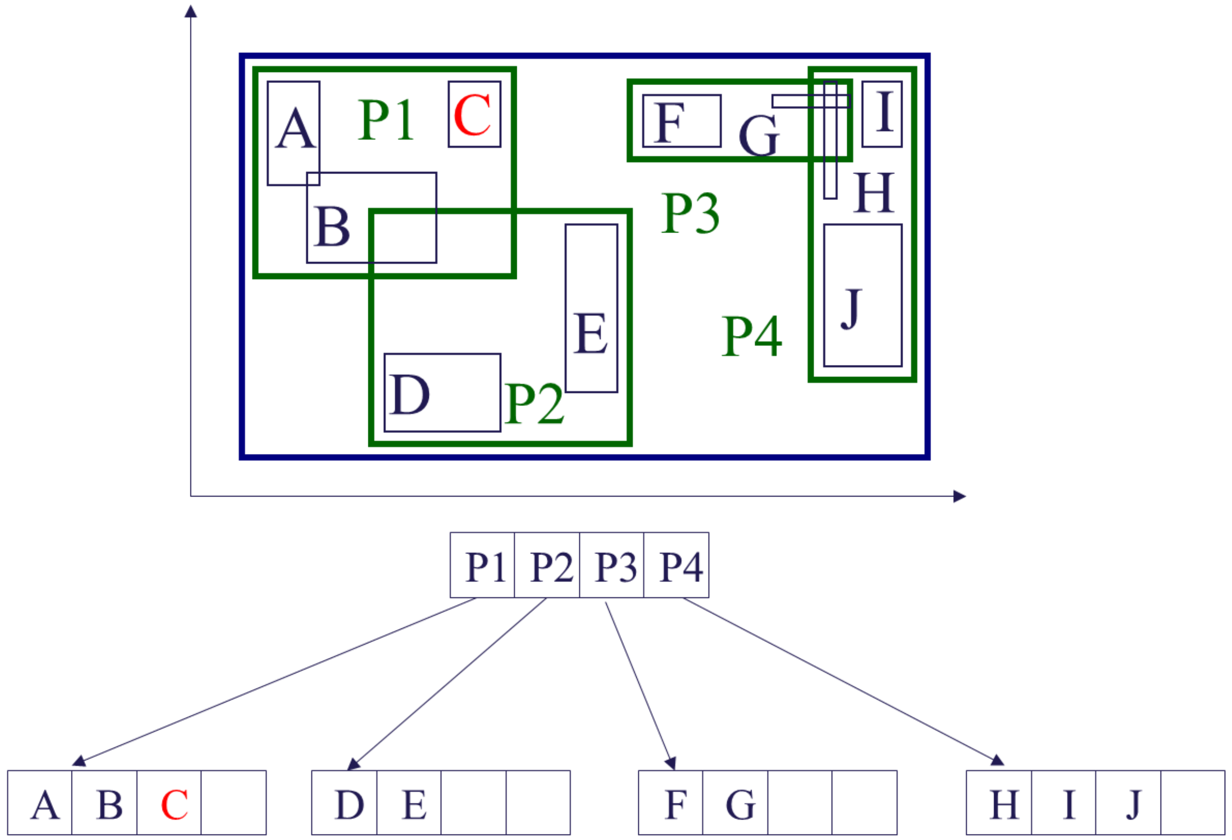
R-Trees: Deletion

- Find the leaf node *N* that contains the entry *E*
- Remove *E* from this node *N*
- If node *N* goes underflow:
  - Eliminate node *N* by removing its entry E_N from its parent node and add *N* to *Q*
- Otherwise, shrink *N*'s MBR accordingly
- Propagation upward
  - Reinsert all entries of nodes in *Q* into the tree using Algorithm **Insert**

Antonin Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. ACM SIGMOD Conference 1984: 47-57

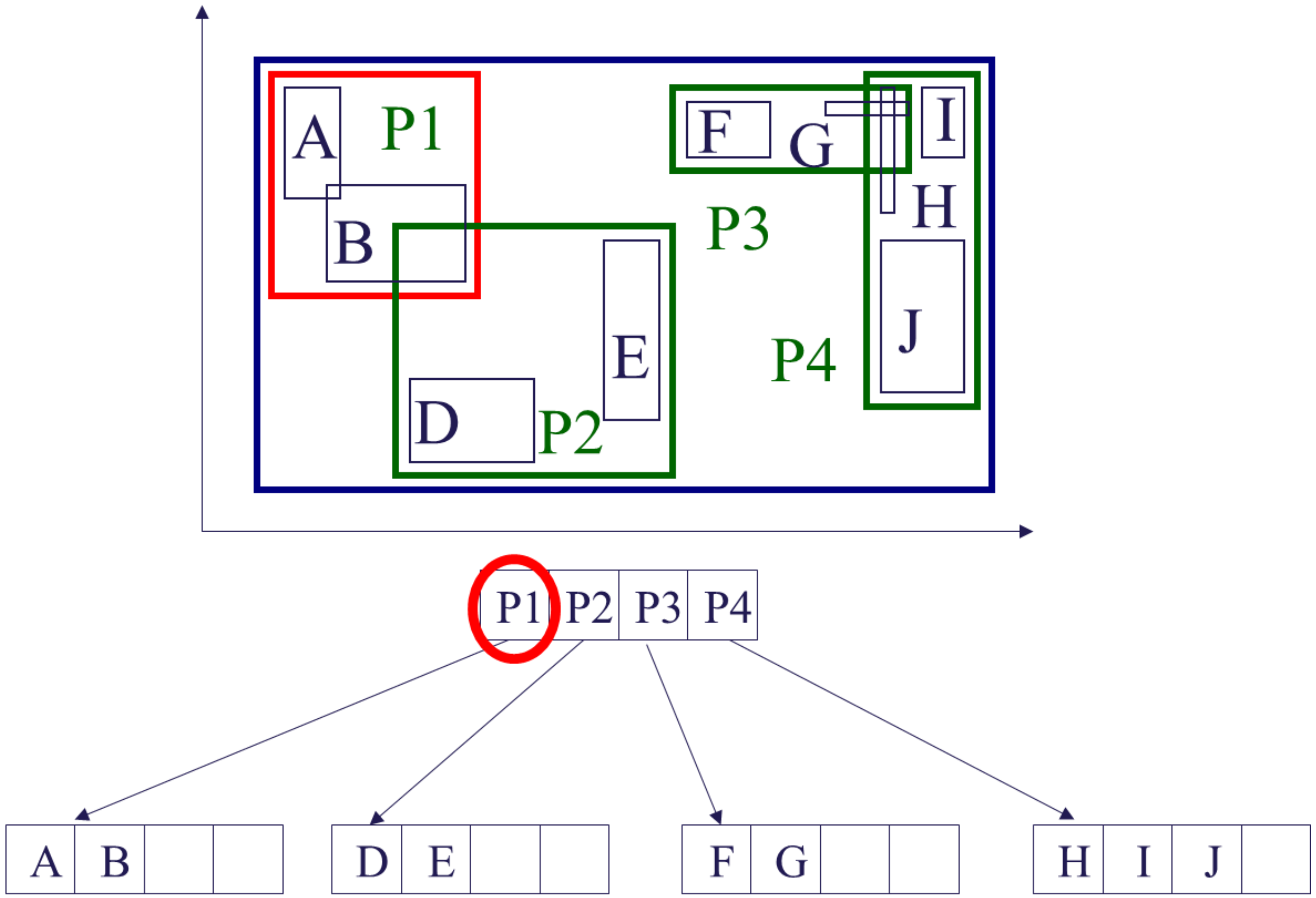
Delete: Shrink MBR

- To delete C



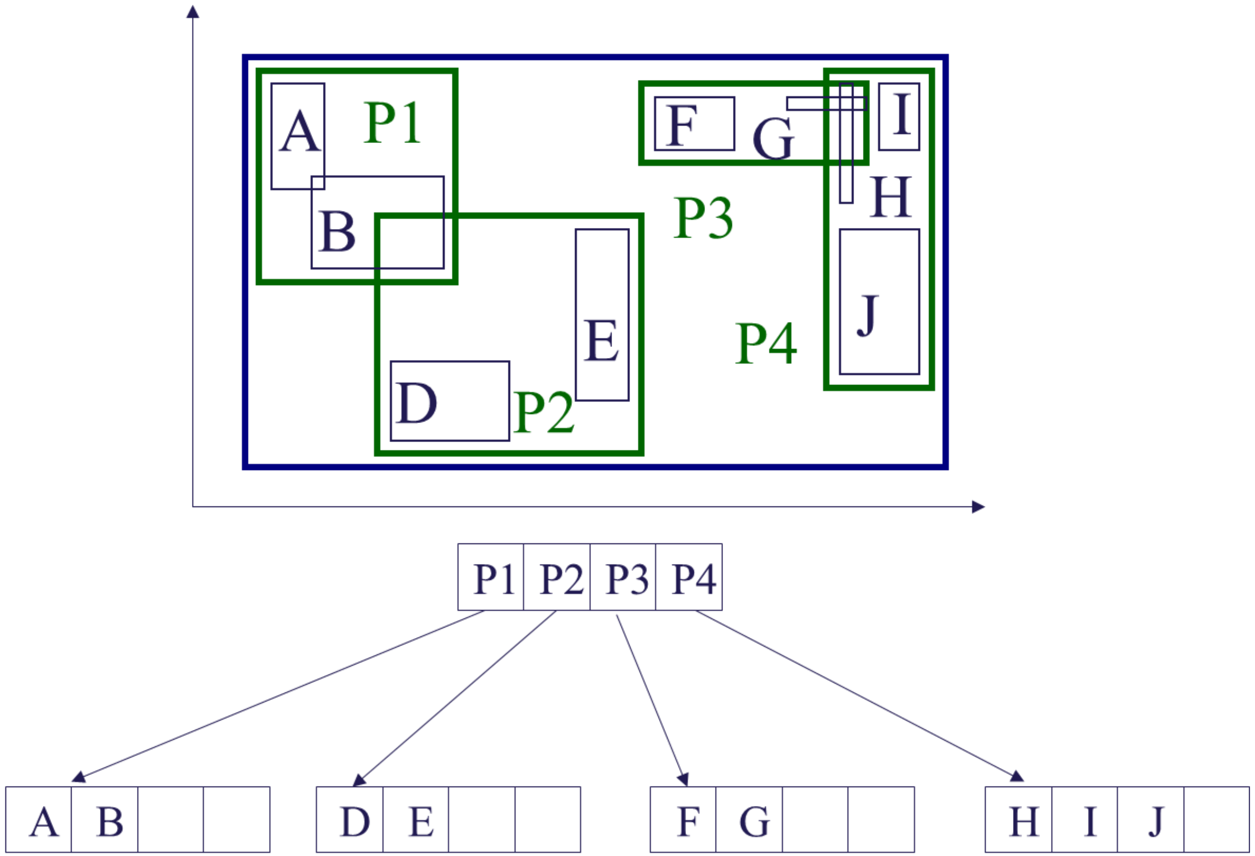
Delete: Shrink MBR, cont.

- After deleting C, P1's MBR has been shrunk.



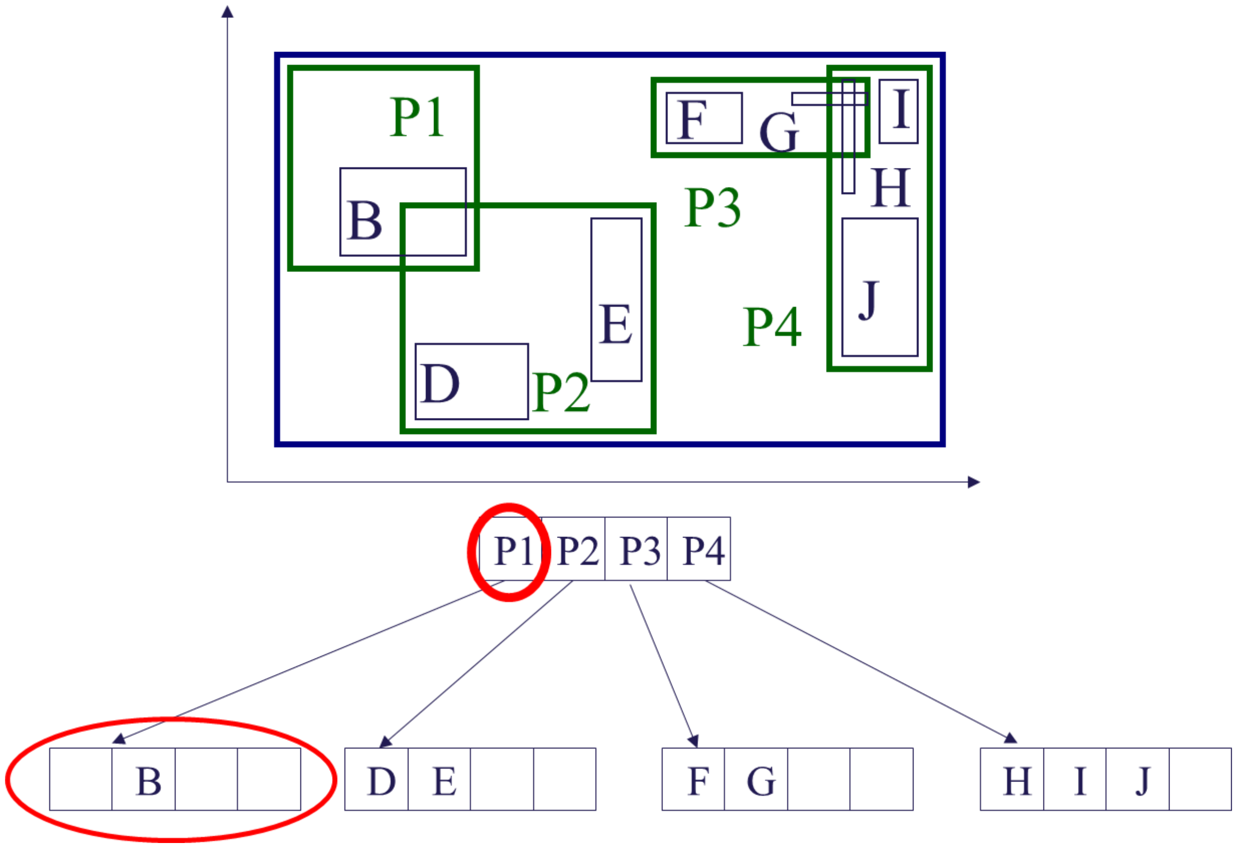
Delete: Node Underflow

- To delete A.



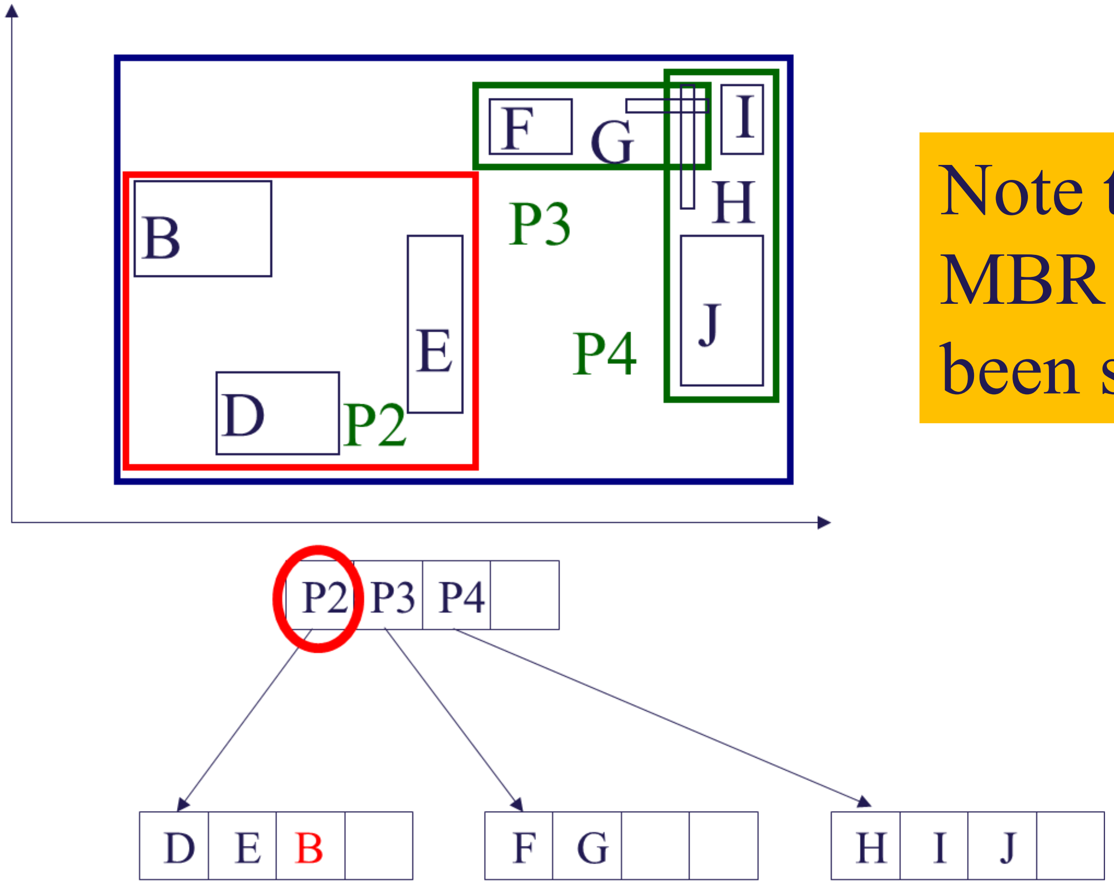
Delete: Node Underflow, cont.

- After deleting A, P1 will become underflow.



Delete: Node Underflow, cont.

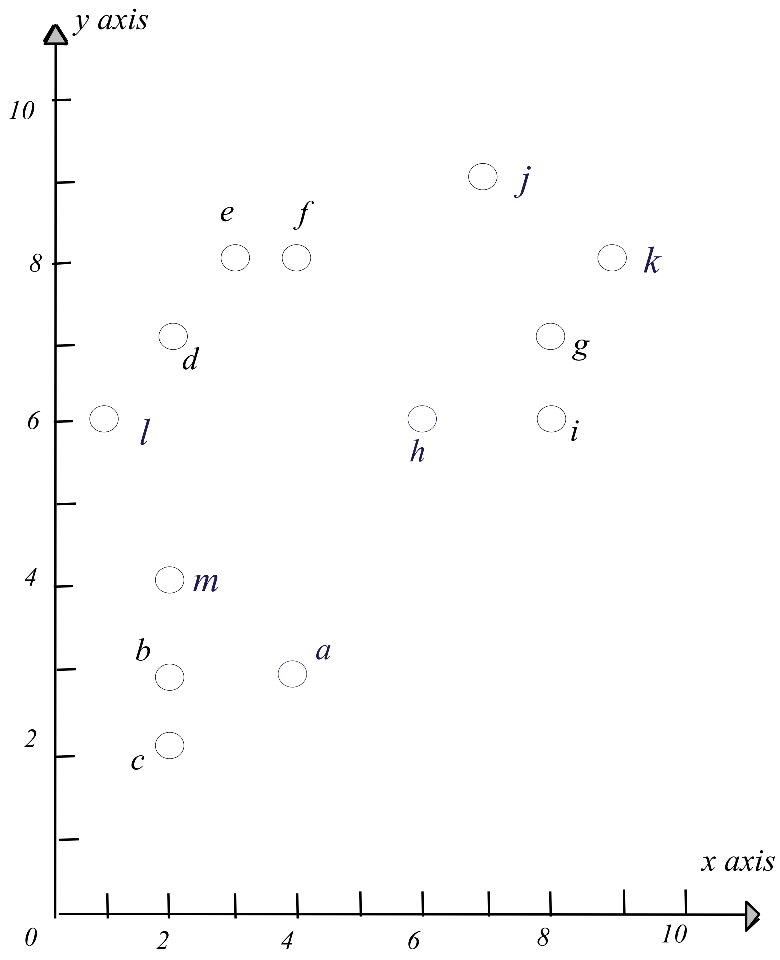
- P1 is removed from root, and B is re-inserted (to P2).



Note the overall MBR has also been shrunk.

Exercise

- On the R-tree you've created
  - Delete point *d*.
  - Delete point *m*.
  - Delete point a.
  - Delete point c.



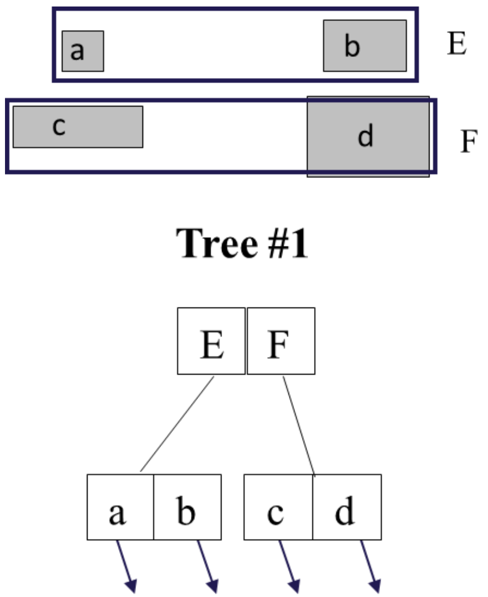
Agenda

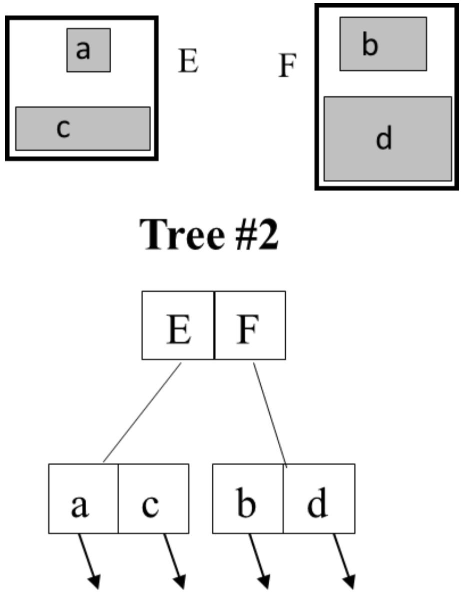
- R-tree
- R-tree Variants
  - R*-tree
  - Bulk loading

R-trees: Variations

- R*-tree
  - Change the insertion, deletion algorithms
  - More criteria to optimize in insertions and deletions
  - *Forced re-insertion* instead of splitting for node overflows
    - Remove some (e.g., 30%) entries and reinsert them into the tree.
    - Could result in all reinserted entries fitting on some existing pages, avoiding a node split. Overall, node splits are reduced and deferred.
- Hilbert R-tree
  - Uses the Hilbert-curve values to insert objects into the tree
- R^+-tree
  - Avoids overlap by splitting and inserting an object into multiple leaves if necessary.
  - Searches now may take a single path (or less paths) to a leaf, at cost of redundancy.

Which R-tree is better? Why?



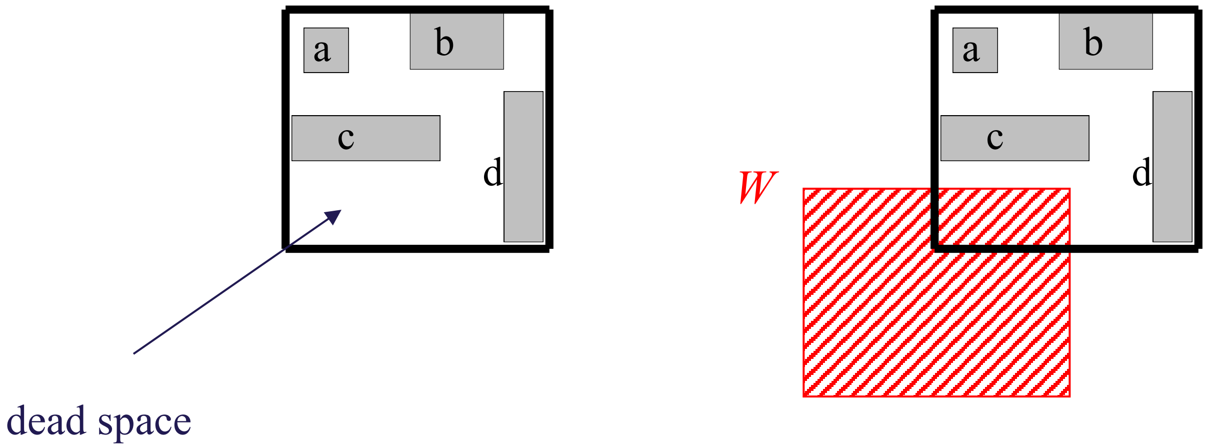


- A "good" tree should have
  - nodes with small MBRs
  - nodes with small MBR overlap
  - nodes that look like squares
  - nodes as full as possible

Sometimes it is impossible to optimize all these criteria at the same time!

R*-tree Optimization Criteria (1)

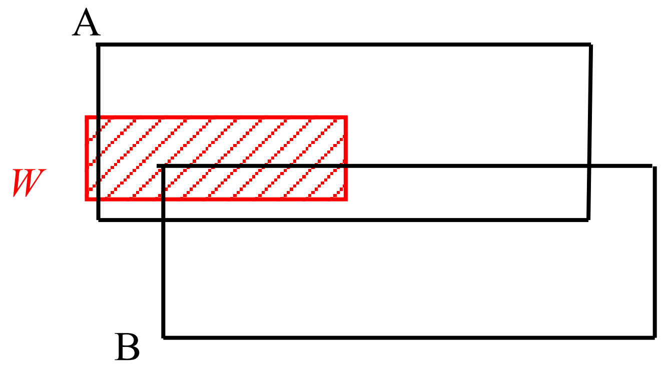
- Minimize the area covered by an index MBR
  - → Small area means small dead space



window query intersects node MBR, but no object!

R*-tree Optimization Criteria (2)

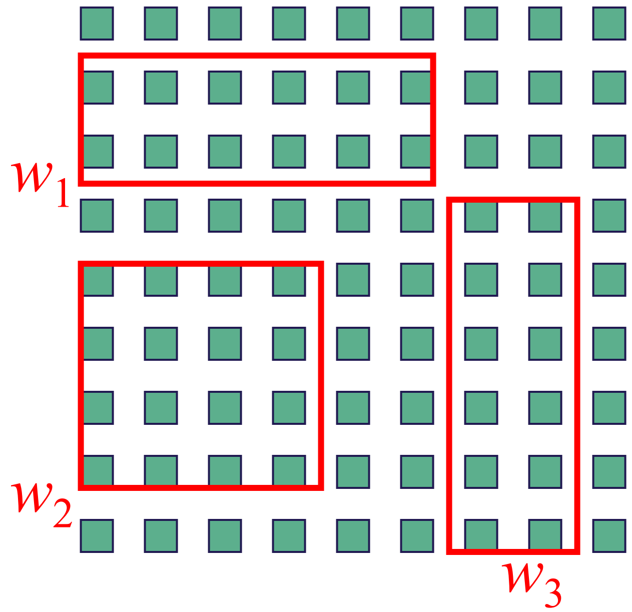
- Minimize overlap between node MBRs
  - → Minimizes the number of traversed paths



Both nodes intersect query!

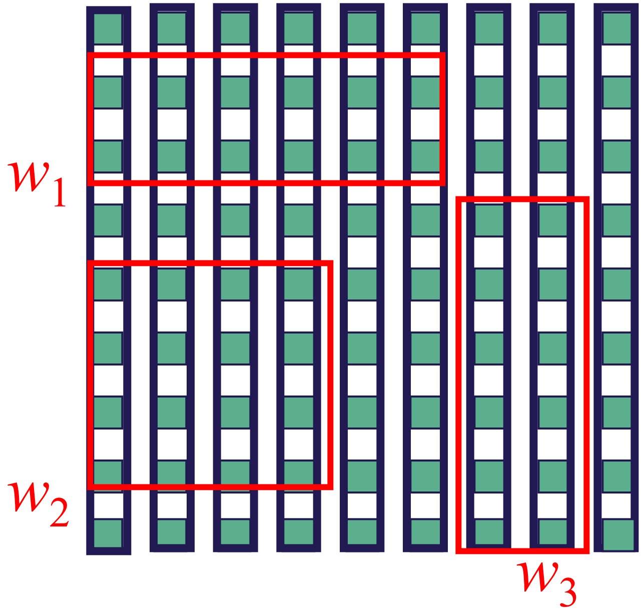
R*-tree Optimization Criteria (3)

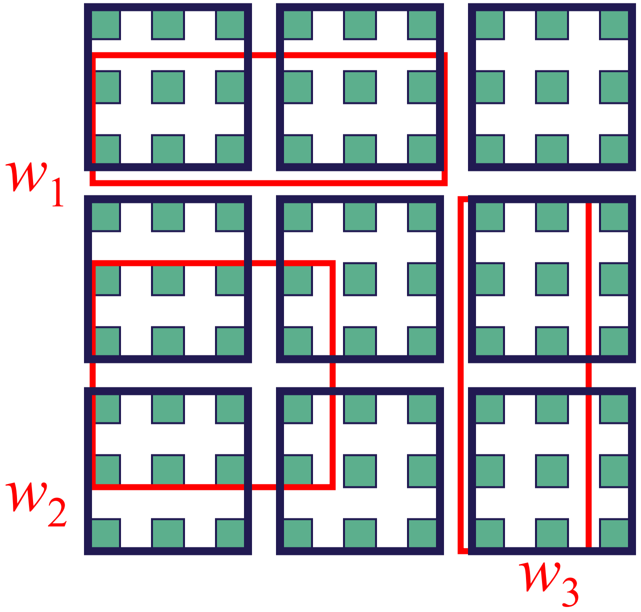
- Minimize the margins of node MBRs
  - → Margin is the sum of the lengths of the edges of a rectangle
  - → Square-like nodes, smaller number of intersections for a random query, better structure
- Problem: group the rectangles into groups of 9, such that the expected number of group MBRs intersected by a (random) query is minimized



Effect of Margins Minimization

- Grouping 1 (minimal areas of group MBRs)
  - Bad grouping for queries w_1, w_2
- Grouping 2 (minimization of margins)
  - Minimizes the expected number of groups touching a random query





R*-tree Optimization Criteria (4)

- Optimize the storage utilization
  - Nodes in tree should be filled as much as possible
  - Minimizes tree height and potentially decreases dead space
- For update-frequent scenarios, this criteria may not be desirable
- This criteria may be in conflict with the first optimization criteria
- Sometimes it is impossible to optimize all these criteria at the same time!

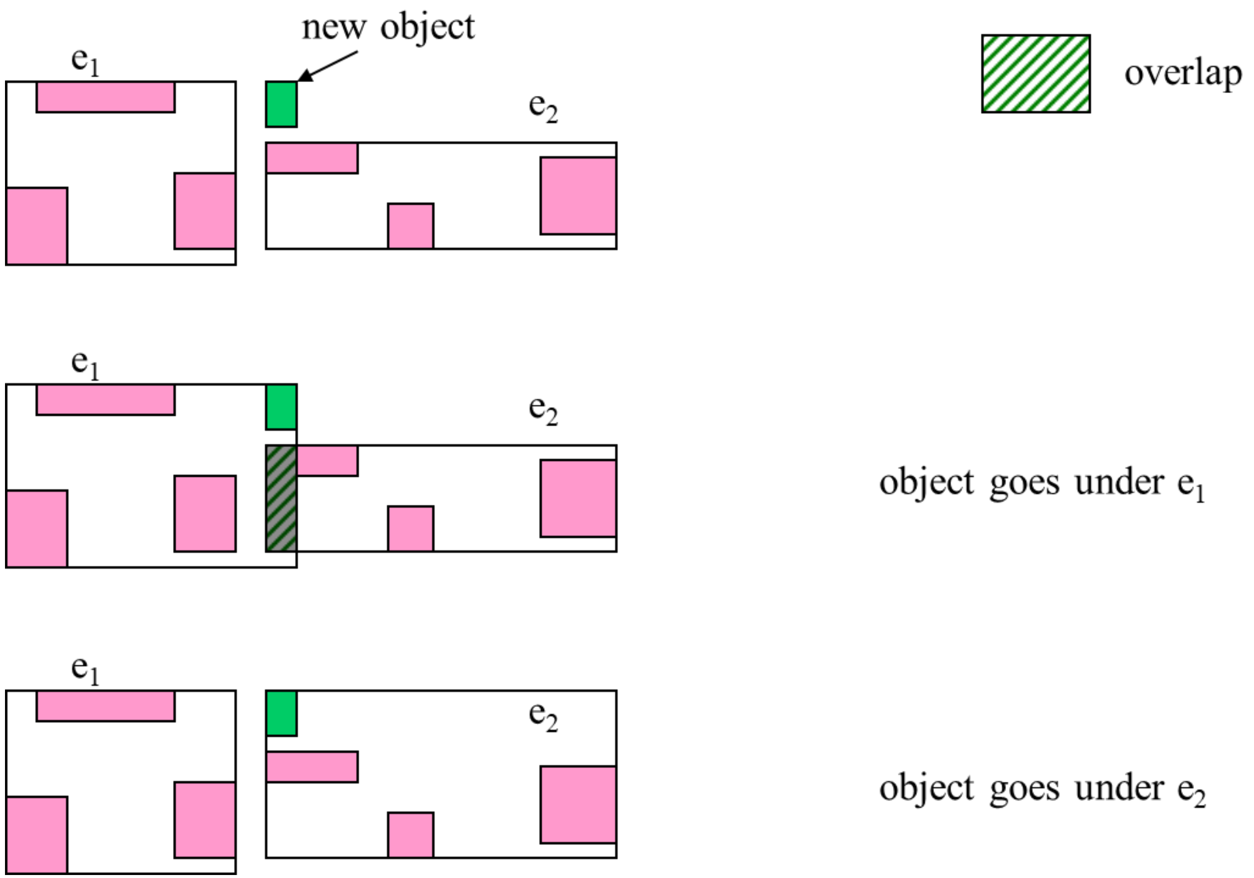
R*-tree Insertion Heuristics

- When inserting a new entry *e* into the tree we follow one path from the root to a leaf
- The entry is then inserted to the leaf

- Issue: How to choose the path (subtree) from node *N*?
  - If *N*'s child pointers point to leaves, choose the leaf node using the following criteria:
    - Needs *the least overlap enlargement* to include the new entry
    - Needs *the least area enlargement* to include the new entry
    - Has *the smallest MBR area* to include the new entry
  - Else, choose the sub-node
    - Needs *the least area enlargement* to include the new entry
    - Has *the smallest MBR area* to include the new entry

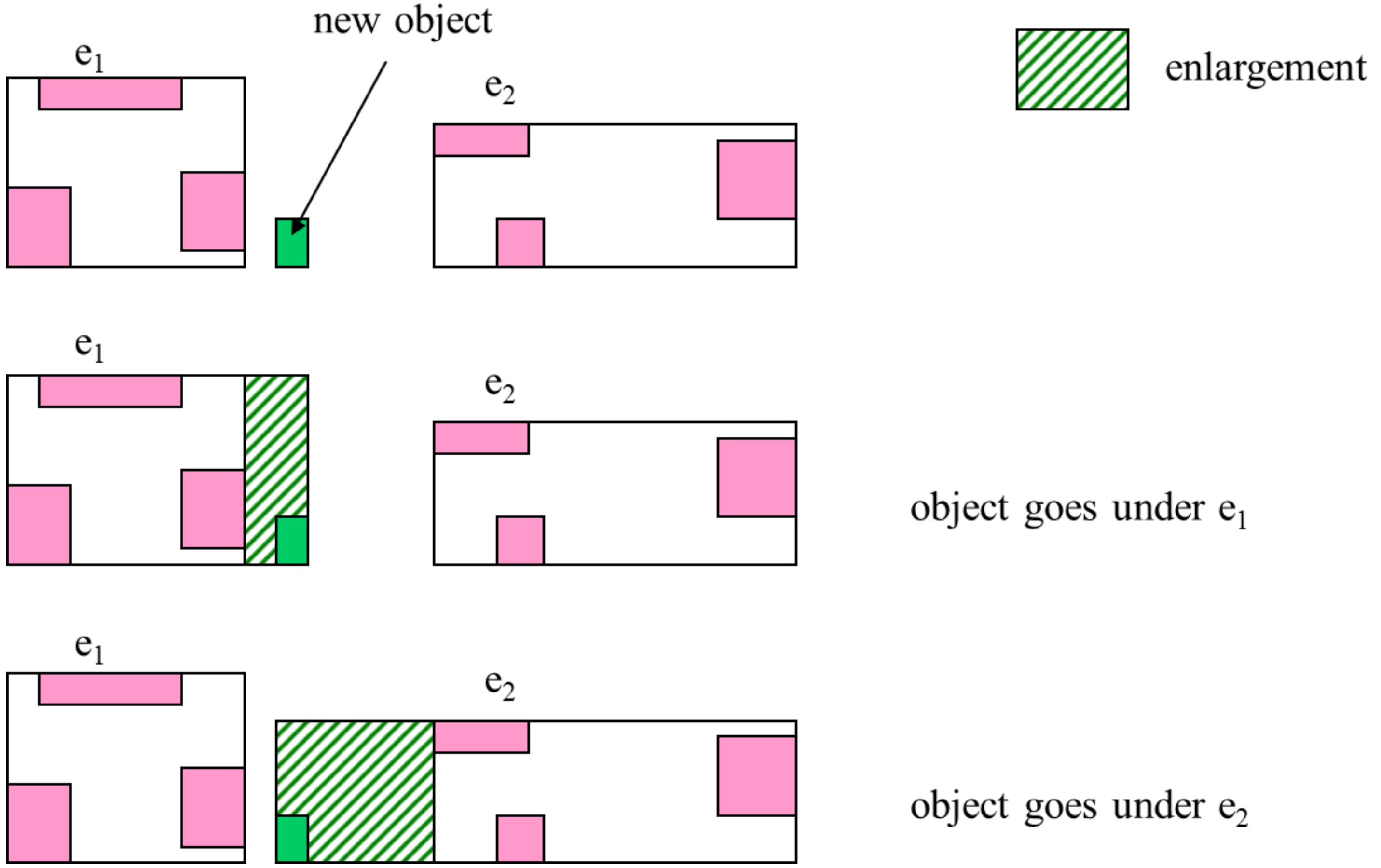
R*-tree Insertion Heuristics, cont.

- The least MBR overlap enlargement after insertion
  - Break ties by choosing MBR with minimum area



R*-tree Insertion Heuristics, cont.

- The least MBR enlargement after insertion
  - Used for interal (non-leaf) nodes



R*-tree Node Splitting

- If a node overflows we need to split it
- Issue: distribute (fast!) a set of rectangles into two nodes such that the areas, overlap, and margins are minimized.
  - Have to give weight on some optimization criteria (conflicting)
  - Distribution may not be even
- Sort the rectangles with respect to one axis (x or y) and find the best split distribution from the sorted lists

Determine the split axis

- For each axis (i.e., x and y axis)
  - Sum=0;
  - sort entries by the lower value, then by upper value
  - for each sorting (e.g. lower value)
    - for k=m to M+1-m
      - place first k entries in group A, and the remaining ones in group B
      - Sum = Sum + margin(A) + margin(B)
- Choose axis with minimum Sum

Distribute entries along the axis

- Along the split axis, choose the distribution with minimum overlap
- If there are multiple groupings with minimal overlap choose <A, B> such that area(A)+area(B) is minimized

Agenda

- R-tree
- R-tree Variants
  - R*-tree
  - Bulk loading

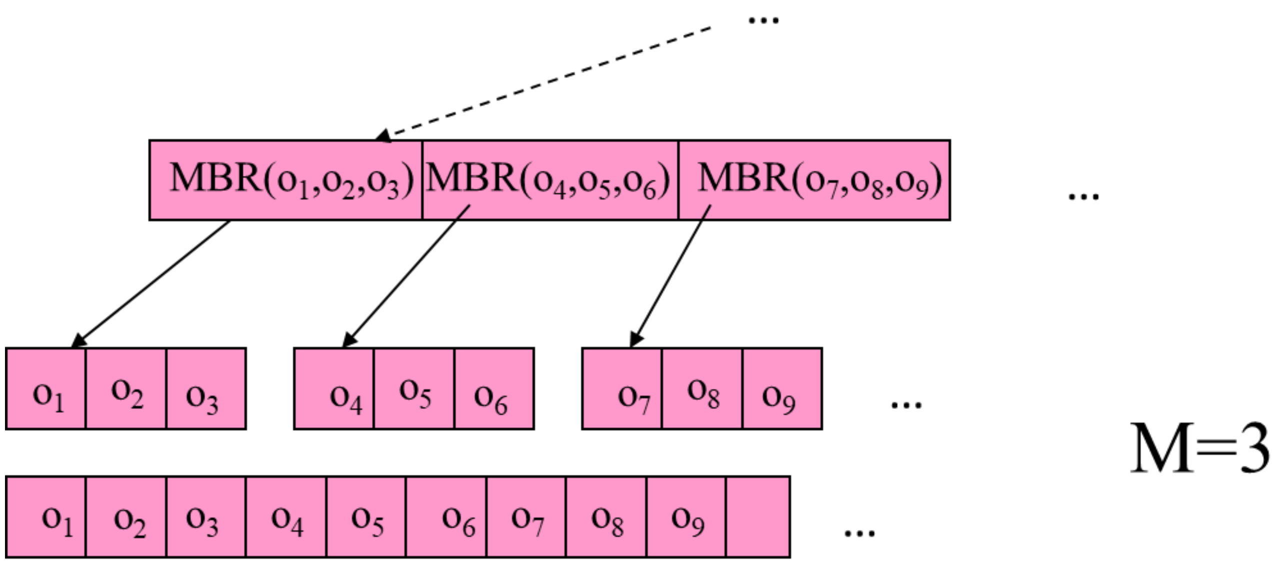
Bulking Loading R-tree

- Given a fixed set of spatial objects, how to create an “optimal” R-tree?

- Nearest-X (x-sorting)
  - Rectangles are sorted on the x-coordinate and nodes are created.
- Hilbert R-Tree
  - Uses the Hilbert value of the center of a rectangle to sort the leaf nodes and recursively builds the tree.
- Sort-Tile-Recursive (STR)

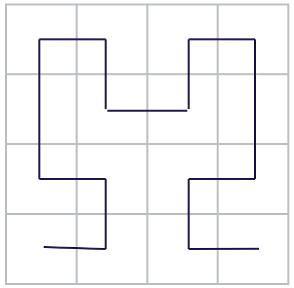
X-Sorting

- Sort using only one axis
  - sort rectangles using the x-coordinate of their center
  - pack M consecutive rectangles in leaf nodes
  - build tree bottom-up

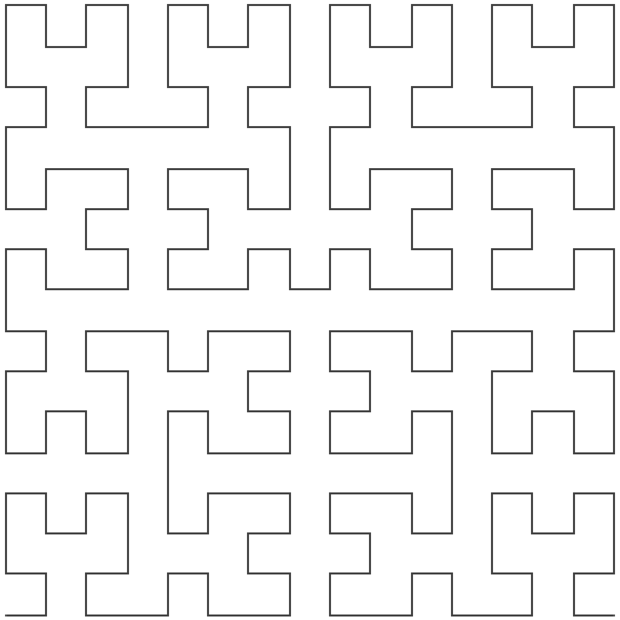


Hilbert R-Tree

- X-sorting results in leaf nodes that are have long stripes as MBRs
- Hilbert R-Tree: use a space-filling curve to order the rectangles
  - much better structure, but still the nodes have large overlap

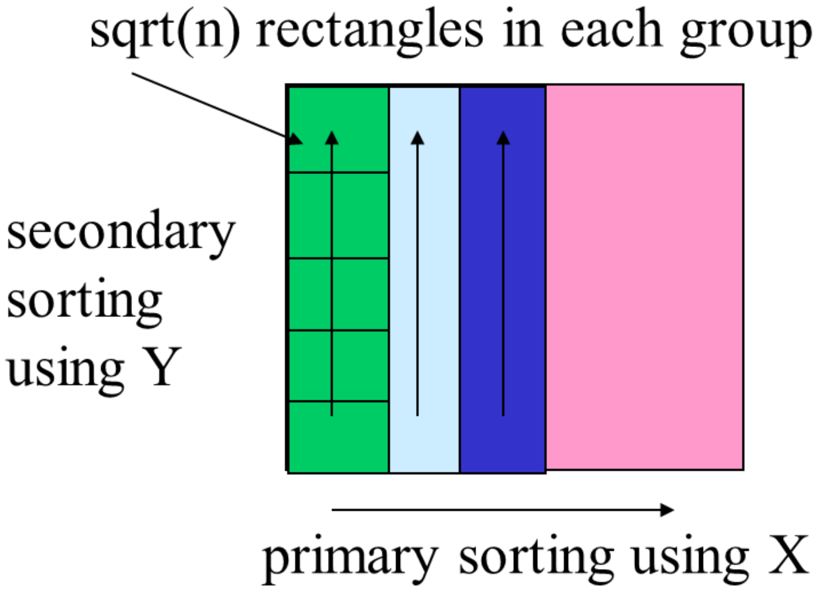


Hilbert-curve

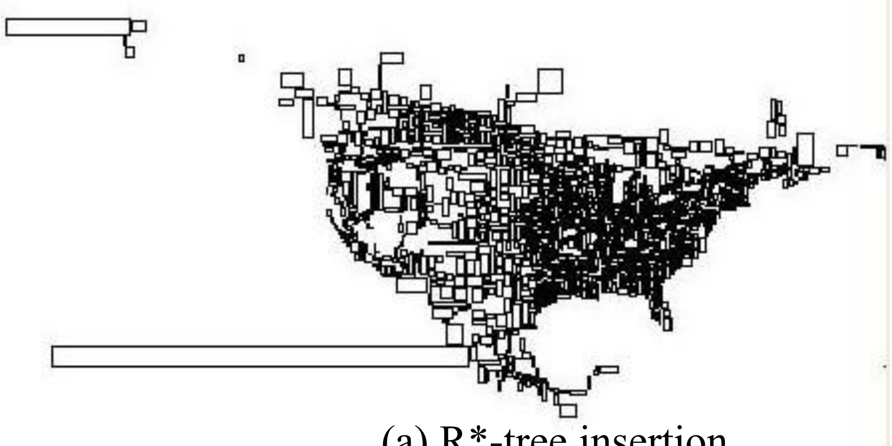


Sort-Tile-Recursive (STR)

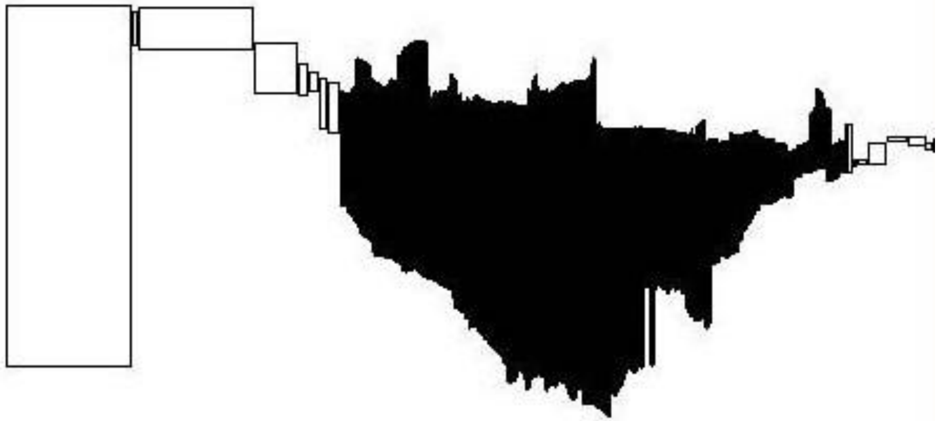
- Sort using one axis first, create sqrt(n) tiles, and then groups of sqrt(n) rectangles in each tile using the other axis (n = #leaves = #objects/capacity) [why sqrt?]
- Usually better compared to other bulk-loading methods



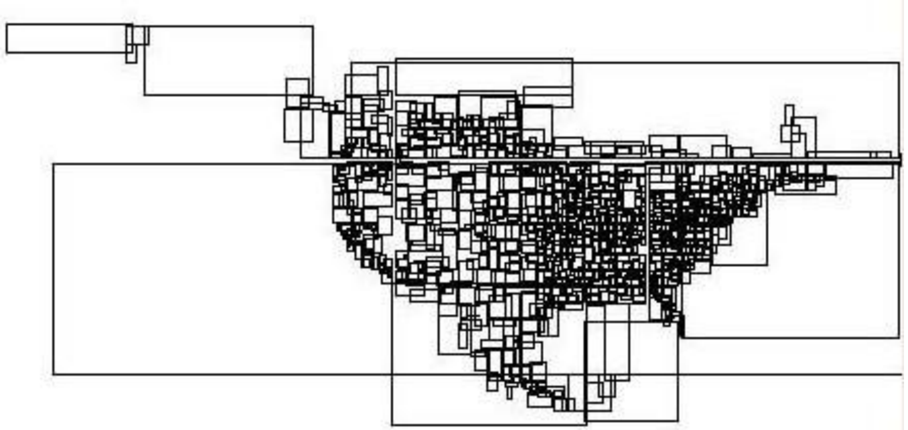
R-tree leaf nodes by different construction methods



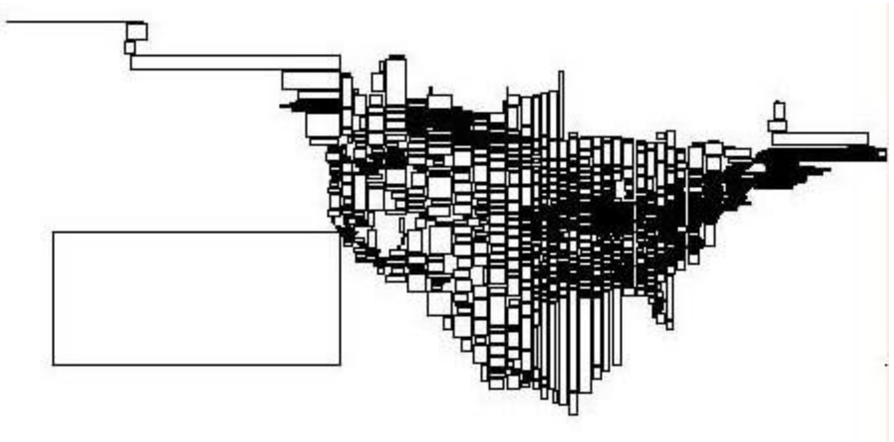
(a) R*-tree insertion



(b) *x*-sorting



(c) Hilbert sorting



(d) Sort-tile recursive

Summary

- Spatial data
  - Data types
  - Spatial relationships
  - Spatial queries
- R-tree
  - R-tree structure
  - R-tree insertions
  - R-tree deletions
- Variants
  - R*-tree optimization criteria
  - Bulk loading

Readings and Exercises

- Mandatory reading
  - A. Silberschatz, H. F. Korth, S. Sudarshan: Database System Concepts (7th edition), McGraw-Hill.
    - 8.4 Spatial Data
    - 14.10 Indexing of Spatial and Temporal Data
  - Antonin Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Conference 1984: 47-57
- Further readings
  - Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD Conference 1990: 322-331
- Exercises
  - Those in the slides
  - The document given in Moodle