Data Organization and Processing

Indexing Spatial Data

(NDBI007)

David Hoksza, Škoda Petr
http://siret.ms.mff.cuni.cz/hoksza

Lecture Outline

- Spatial DBMS
- 2D → 1D mapping
- Spatial indexing
  - tree-based structures (k-d tree, R-tree, UB-tree, …)

Motivation

- **Management and manipulation of data related to multidimensional spaces**
  - **Spatial access methods and databases** deal with objects in **2D** or **3D** space
    - generalization to higher dimensions is straightforward but usually inefficient
  - examples
    - **2D**
      - Geographical Information Systems (**GIS**)
      - Computer Aided Design (**CAD**)
    - **2.5D**
      - location in a 2D space with a dimensional attribute attached to it (elevation)
    - **3D**
      - 3D modeling, …
      - special - molecular structures, models of various systems (e.g., brain), …
    - …
      - Multimedia: images, video, sound, text

Spatial / Geo-spatial Data

- **Spatial data**
  - data with assigned **location**
    - the spatial component of an object consist of a **location** and an **extent** *(rozsah)*
    - contains also non-spatial component
- **Geographic/geo-spatial data**
  - spatial data with assigned **location with Earth's surface** as the **reference frame**

Frame of Reference

- Every domain dealing with spatial data needs a **spatial frame of reference** *(referenční rámec)* to **anchor** the **objects** to be able to process them

- Examples
  - GIS, satellite images
    - **Earth's surface** (coordination system + 2D projection + geocentric datums)
  - CAD
    - e.g., **building's layout**
  - medical imaging
    - e.g., **human body**

Spatial Databases (1)

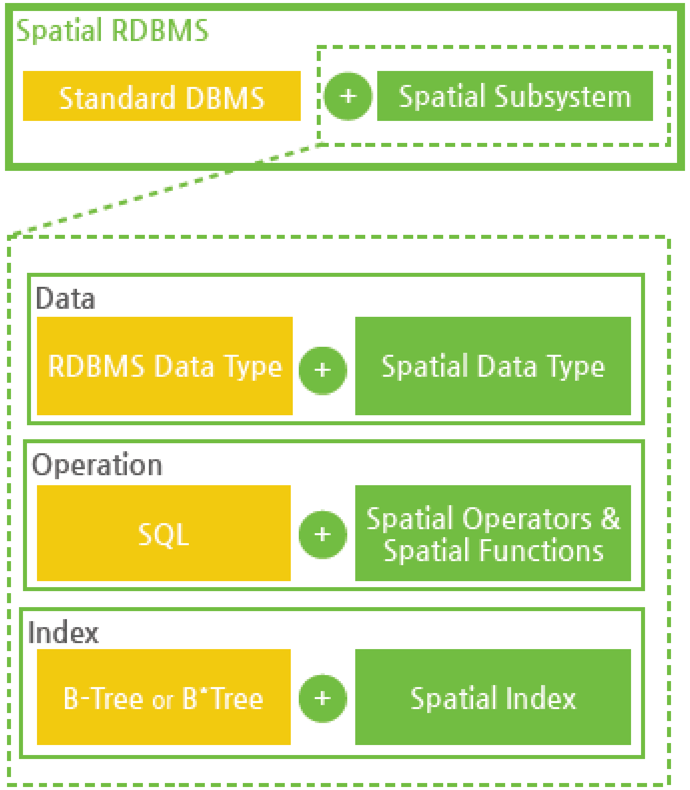
- Spatial database
  - collection of **objects with location and extent**
    - i.e., an image itself is not a spatial database unless some extraction process takes place
  - **objects**
    - **point**
      - also a set of related points such as polyline
    - **region**

Spatial Databases (2)

- **Required operations**
  - **standard update and search operations** (insert, delete, update, search)
  - operations related to **spatial objects**
    - **finding objects in/near** another objects
    - **intersection**
    - **spatial join**
      - determines for two sets of spatial objects all objects related by a spatial predicate
    - …

Spatial DBMS (SDBMS)

- **Extension of DBMS**
  - **DBMS + spatial subsystem**
    - **spatial data types**
      - points, lines, polylines, areas, …
    - **spatial query language**
      - incorporation into its inherent query language, e.g. SQL
    - **spatial indexing**
      - efficient techniques for spatial operations (e.g., spatial join)



source: http://www.cubrid.org/wp-content/uploads/2011/09/conceptual-diagram-of-a-spatial-rdbms.png

Spatial Data Types

- **Point data**
  - n-dimensional **points** in m-dimensional **space**
  - e.g., locations in the modeled domain (houses, poles, …), points of a raster image, feature vectors
- **Line data**
  - **connected set of line segments**
  - e.g., rivers, roads, sequence of events located in (real or feature) space
- **Region data (polygon)**
  - **point data with spatial extent** defined by its boundaries
    - boundaries are defined by connected line segments (vectors) with common beginning and end
  - e.g., countries, census blocks, …

Examples of Spatial Queries

- List the names of all cinemas within 10 kilometers from the city center.
- List all rivers crossing at least two states.
- List all customers living in the Czech Republic and the neighboring countries.
- Find all cities within 15 miles of highway I170.
- Find the five cities nearest to the interstate highway I170.
- Aggregate all the counties in Texas, producing the boundary for the state of Texas.

Spatial Queries/Operations

Queries

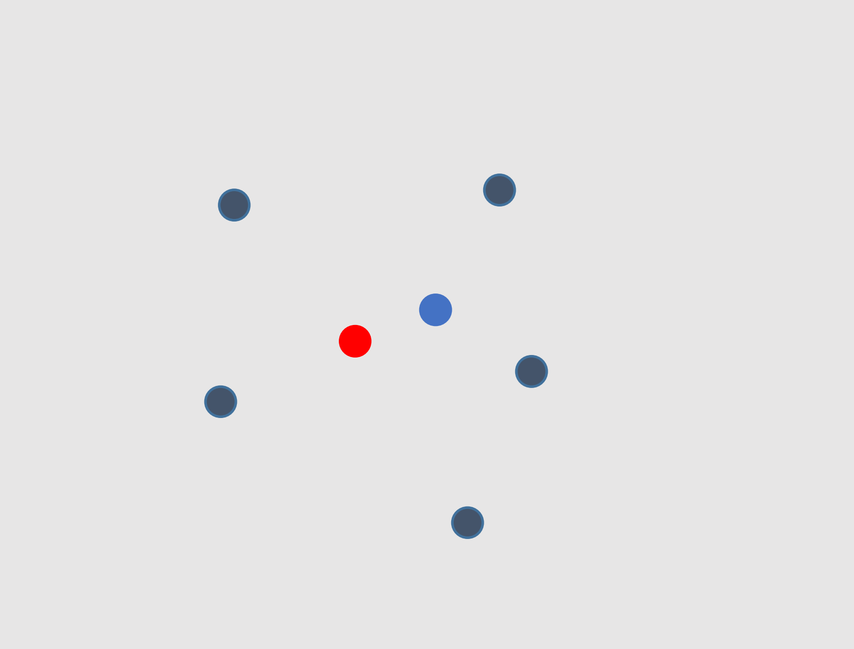
- Containment query
- Region query
- Enclosure query
- Line intersection query
- Adjacency query
- Metric (proximity) queries

Operations

- Clipping
- Spatial join
- Map overlay
- Merge/Aggregation

Nearest Neighbor Query

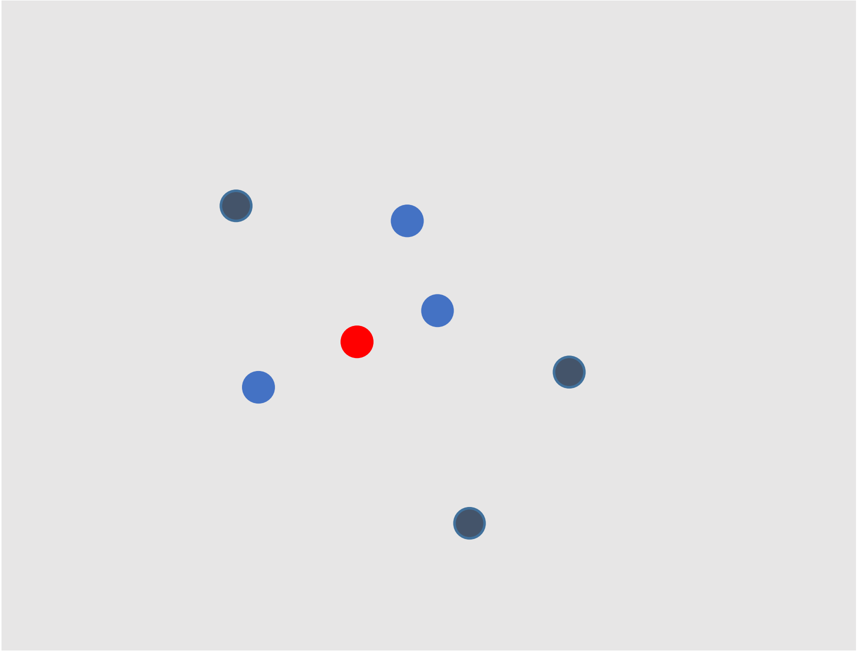
- *Dotaz na nejbližšího souseda*
- Given an object ***O***, find the **object** in the map that is **closest** to ***O***
  - very common query in any space containing any metric



k-Nearest Neighbor Query

- *Dotaz na k nejbližších sousedů*

- Given an object ***O***, find **k objects** in the map that are **closest** to ***O***
  - very common query in any space containing any metric

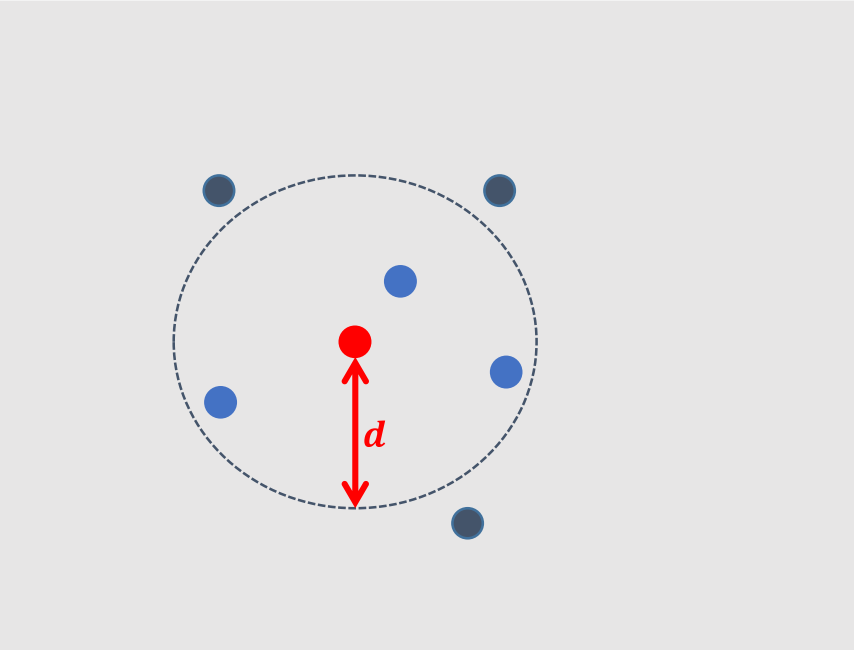


3-NN

Range Query

- *Rozsahový dotaz*

- Given an object ***O*** and a distance ***d***, find all **objects** in the data set that are **within distance** ***d*** from ***O***



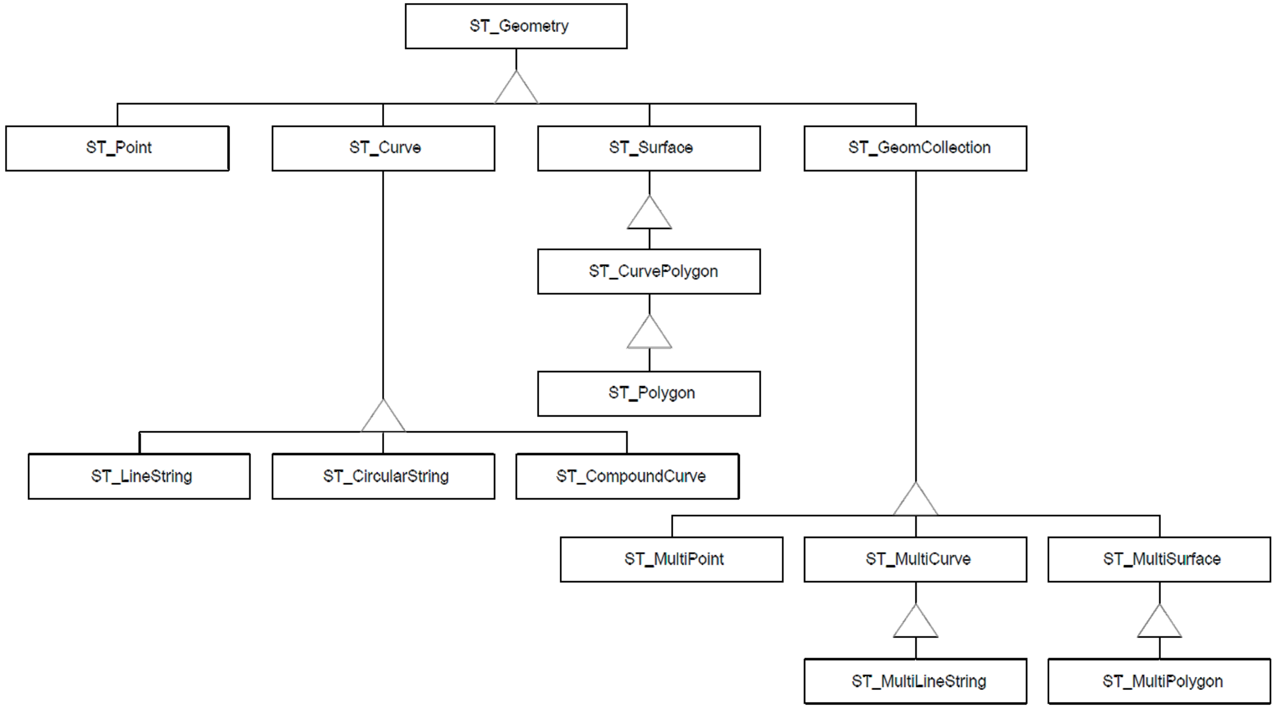
Spatial Join

- *Prostorové spojení*

- Given two sets of spatial objects, spatial join **pairs the sets' objects** based on a given spatial predicate
  - **intersection**
    - identify pairs of objects from the two sets which intersect
    - "Find all pairs of rivers and cities that intersect."
  - **distance**
    - identify pairs of objects from the two sets which are within given distance
  - **… any relation including a pair of spatial objects**

SQL Extension

- **SQL Multimedia and Application P**
- Standard being **part of SQL:1999**
  - ISO SQL Geometry Specification



  - set of **user-defined types** (UDT) and **us**
    - Area(POLYGON), Distance(GEOMETRY,GEOMETRY), Contains(GEOMETRY,GEOMETRY), Intersection(GEOMETRY,GEOMETRY), Intersects(GEOMETRY,GEOMETRY), Union(GEOMETRY,GEOMETRY), Buffer(GEOMETRY,double), ConvexHull(GEOMETRY), Perimeter(GEOMETRY), Crosses(GEOMETRY,GEOMETRY), Transform(GEOMETRY,integerSRID), Dimension(GEOMETRY), AsText(GEOMETRY), ST_X(POINT), ST_Y(POINT), NumPoints(GEOMETRY), PointN(GEOMETRY,integer), NumGeometries(GEOMETRY), GeometryN(GEOMETRY,integer), GeometryType(GEOMETRY)

PostGIS implementation

SQL Extension - Examples

- Insert a road and its coordinates

```
INSERT INTO roads
(road_id, road_geom, road_name)
VALUES
(1, GeomFromText('LINESTRING(19123 24311,19110 23242)',
242),'Jeff Rd.')
```

- How many people live within 5 miles of the toxic gas leak

```
SELECT sum(population)
FROM census_tracks
WHERE distance(census_geom, 'POINT(210030 3731201)') < (5 * 1609.344)
```

- What is the area of municipal parks inside the Westside neighbourhood

```
SELECT sum(area(park_geom))
FROM parks, nhoods
WHERE contains(nd_geom,park_geom) AND nhood_name = 'Westside'
```

Spatial Objects Representation

- When dealing with spatial objects in terms of **storage and manipulation** we can either
  - **project (serialize**) them **into 1D** space and employ **existing single-dimensional methods**
  - utilize the **full spatial information** with **specialized techniques** for spatial management

One-Dimensional Embedding of Spatial Objects (1)

- **Data files** are **linear** (1-D) which is **not natural for spatial data** (n-D)

- We would like to **cluster together data** to maintain **locality**
  - data accessed by a query are grouped together
  - reduced I/O cost for queries

- We assume a **space representable as a grid**
  - every space can be expressed as a sufficiently granular grid

One-Dimensional Embedding of Spatial Objects (2)

- **Space filling curve**
  - **a curve visiting cells of the grid** representing the space; **each cell is visited exactly once**
  - the points on the line are ordered thus giving the points in the space (grid) **linear ordering**

- **Typical** approaches for space filling curves
  - Naïve curve, spiral curve, **Z-curve, Hilbert curve**, …

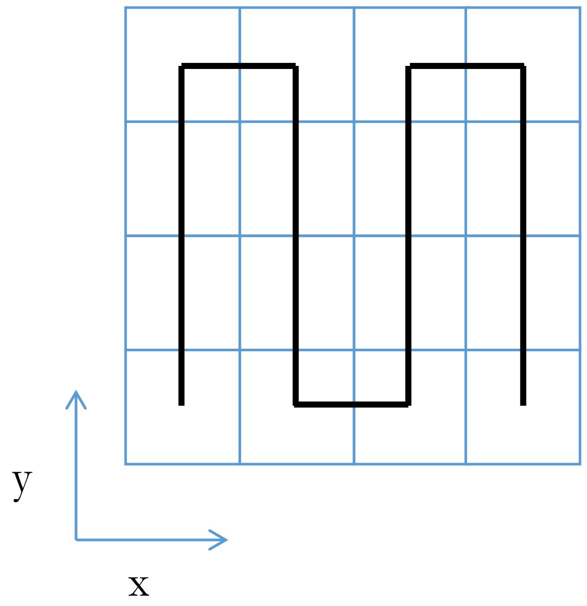
- With space filling curves one can implement file operations similarly to standard ordered files

Naïve representation

- 2D representation is projected into 1D so that **points with smaller x coordinates precede those with larger x coordinates**

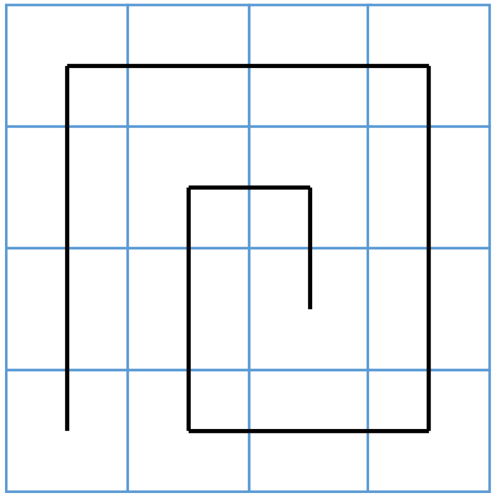
- The goal is to find out 2D → 1D mapping **preserving locality** as best as possible

  - in bigger grid the locality of neighboring points in the x direction is quite poor



Spiral Representation

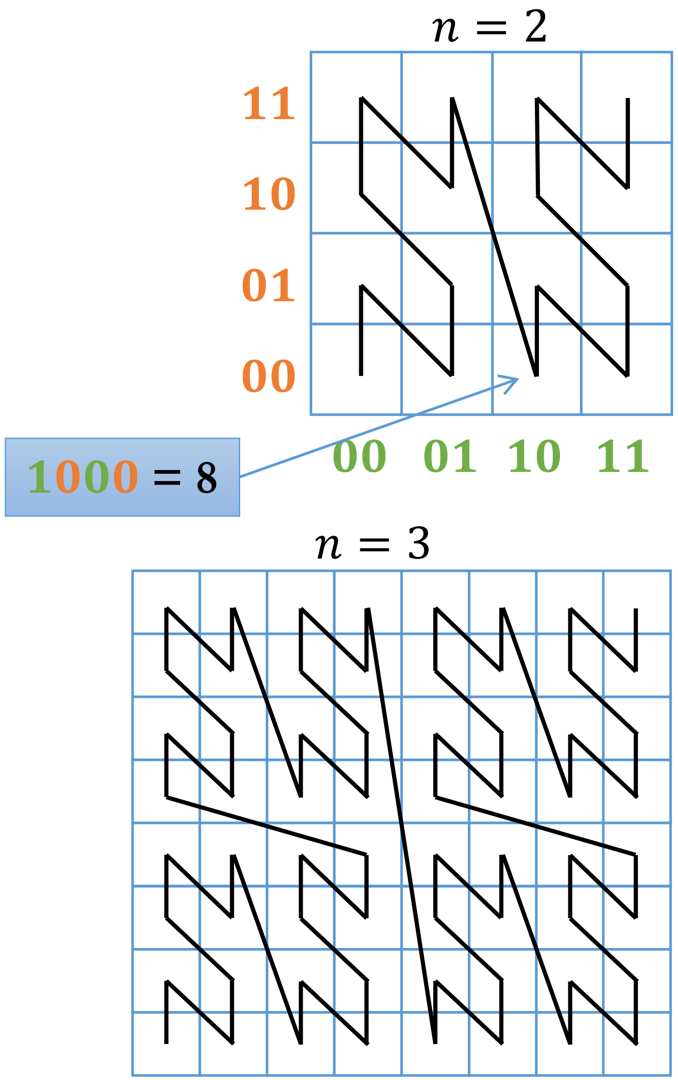
- **Spiral representation** assigns addresses in a **spiral fashion starting from the middle of the grid**



- Another naïve representation in the sense of **poor maintenance of the locality**
  - favors objects in the middle of the grid but is not suitable for "moderately" off-centered objects

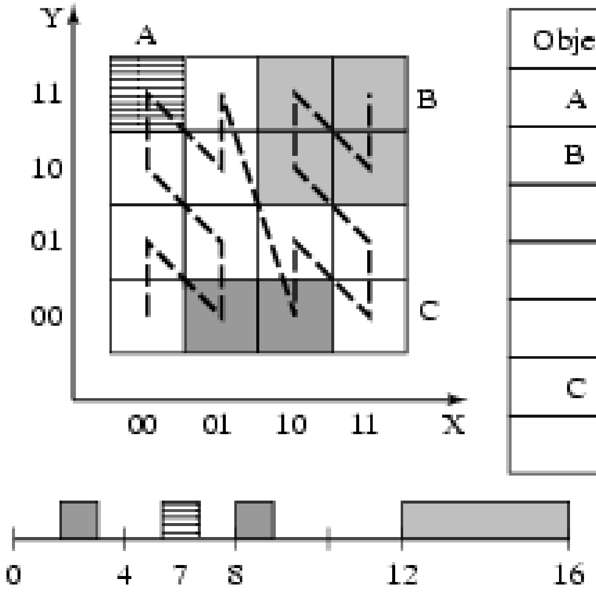
Z-Curve/Z-Ordering/Pean Curve

- **Connecting** points **by z-order** (connecting Zs)
  - Recursive representation
- **Address** is formed **by interleaving the bits** in bit representations of x and y coordinates
  - 1. position corresponds to 1. bit of the x coordinate bit representation
  - 2. position corresponds to 1. bit of the y coordinate bit representation
  - 3. position corresponds to 2. bit of the x coordinate bit representation
  - …



Z-value Example

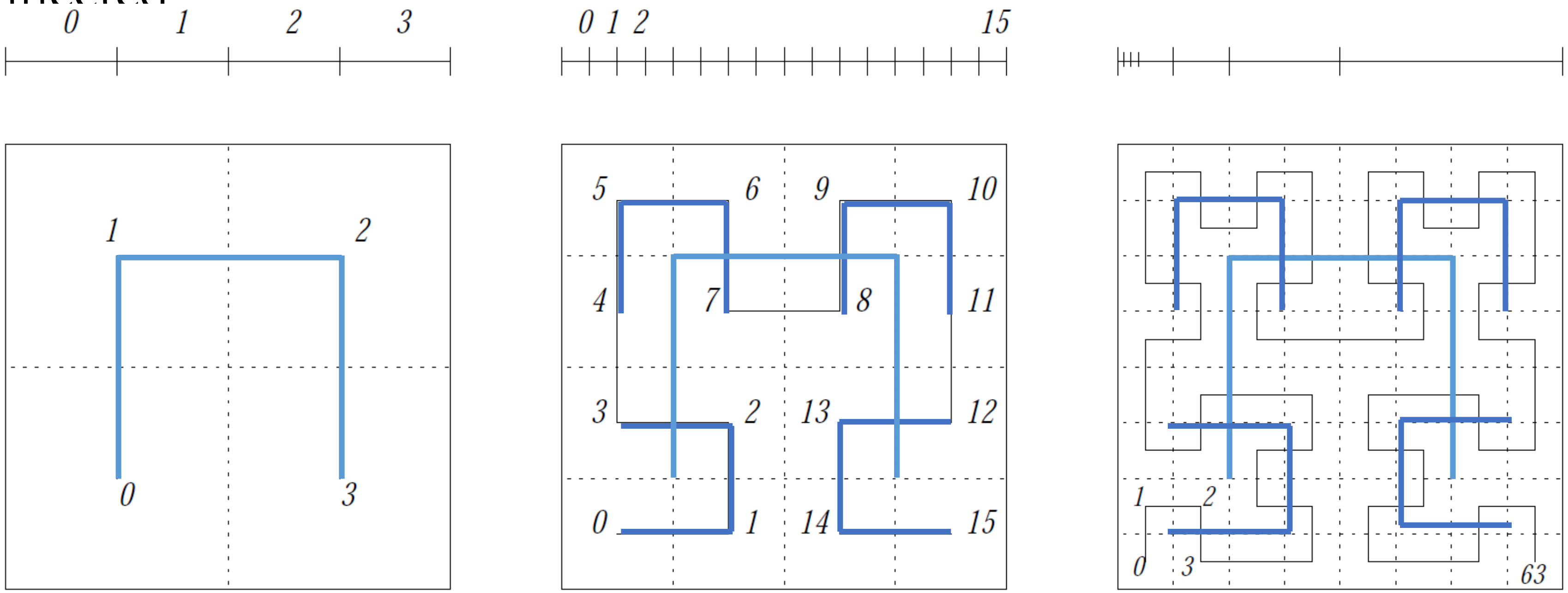
- Map contains **3 object** A, B, C
- **Points of object *C*** obtain addresses which are **not close**
- On the other hand, object ***B*** obtains **neighboring addresses**



Object	Points	x	y	interleave	z-value
A	1	00	11	0101	5
B	1				
	2				
	3				
	4				
C	1	01	00	0010	2
	2	10	00	1000	8

Hilbert Curve

- Recursive representation of the space
  - space is divided into four parts and their ordering is given by the “cup”-like curve
  - every square is divided into another four parts using another cup-like curve which needs to be rotated so that neighboring squares in higher level ordering are connected



First Order Hilbert Curve

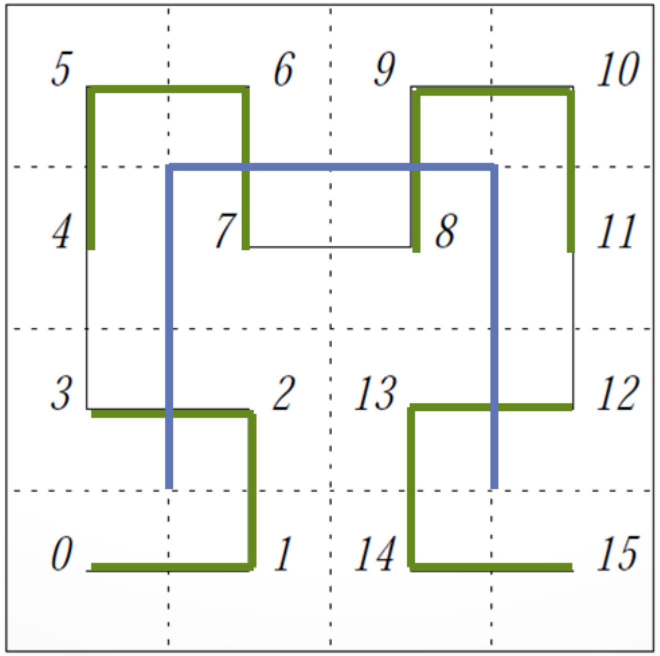
Second Order Hilbert Curve

Third Order Hilbert Curve

Hilbert Curve (cont.)

- 2D → 1D mapping

```
int xy2d (int n, int x, int y) {
    int rx, ry, s, d=0;
    for (s=n/2; s>0; s/=2) {
        rx = (x & s) > 0;
        ry = (y & s) > 0;
        d += s * s * ((3 * rx) ^ ry);
        rot(s, &x, &y, rx, ry);
    }
    return d;
}
```




```
void rot(int n, int *x, int *y, int rx, int ry) {
    if (ry == 0) {
        if (rx == 1) {
            *x = n-1 - *x;
            *y = n-1 - *y;
        }

        //Swap x and y
        int t  = *x;
        *x = *y;
        *y = t;
    }
}
```

Spatial Indexing

- Methods for **efficient search in multidimensional data**; i.e., spatial queries should access as few pages as possible
- **B+-trees** are usable but they are **basically single-domain** indexes, although they can support multi-dimensional data/queries (e.g. using space-filling curves)
- To efficiently index multi-dimensional data we need **multi-dimensional indexes**
  - Quadtree, k-d-tree
  - R-tree, R+-tree, R*-tree
  - Hilbert R-tree, X-tree, …
  - UB-tree, …
  - …

Single Dimension-Based Indexing (1)

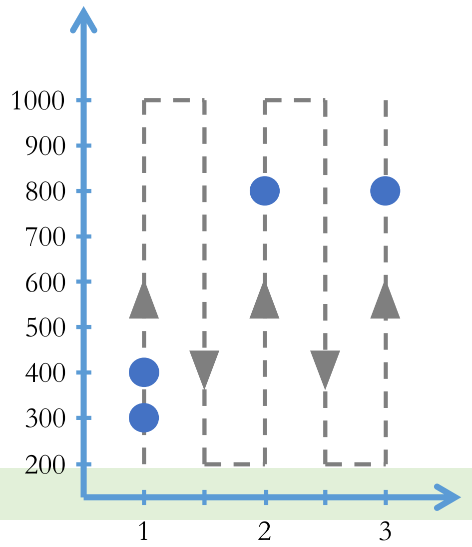
- **B+Trees** are capable of storing **multi-dimensional information** in form of an **ordered tuple** – **compound (chained) search keys** *(složený klíč)*
    - the tuples are **ordered first based on the first element**, then on the second and so on (lexicographical order)
        - the standard ordering of tuples in a B+-tree resambles naïve space-filling curve
        - the way in which we define ordering on the tuples defines the type of space-filling curve

Single Dimension-Based Indexing (2)

- Let us have a **composite index** on **<quality,price>** of a product, then we can **visualize the indexed tuples as points** in 2-dimensional space
  - the index can be a standard B-tree where the ordering of the 2D points defines the ordering based on which the tree is built
  - composite indexes **with more than two attributes** form **points in higher-dimensional space**

- $< 1; 300 >, < 1; 400 >$
- $< 2; 800 >, < 3; 800 >$

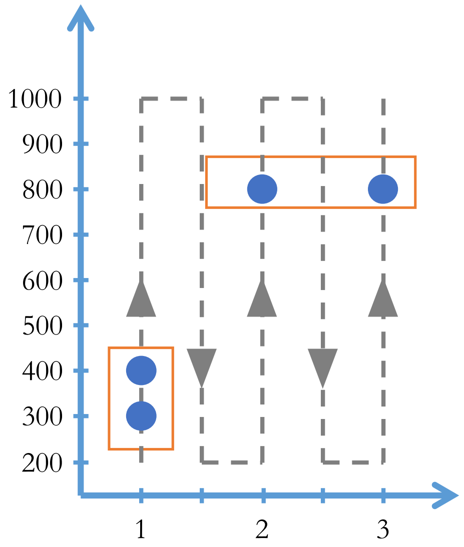
The composite index specifies the ordering – first quality (x axis), second price (y axis).



Single Dimension-Based Indexing (3)

- For real spatial data, one would intuitively group together objects $< 1; 300 >, < 1; 400 >$ and $< 2; 800 >, < 3; 800 >$, but since the index is one-dimensional, only the first pair of objects is close

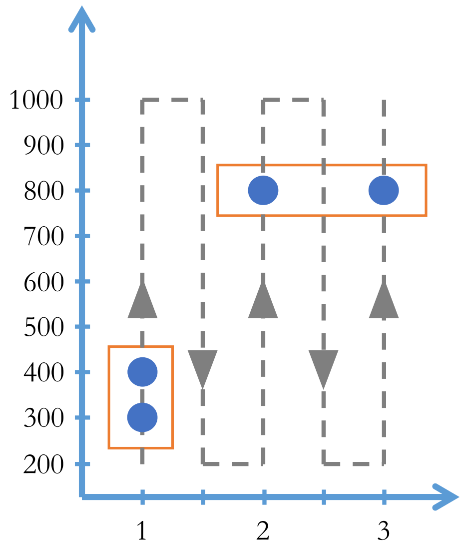
- For real spatial data, we need a way to **group spatially close objects** together in the indexing structure → **all the dimensions** need to be **directly involved** in the structure



Multidimensional Indexing

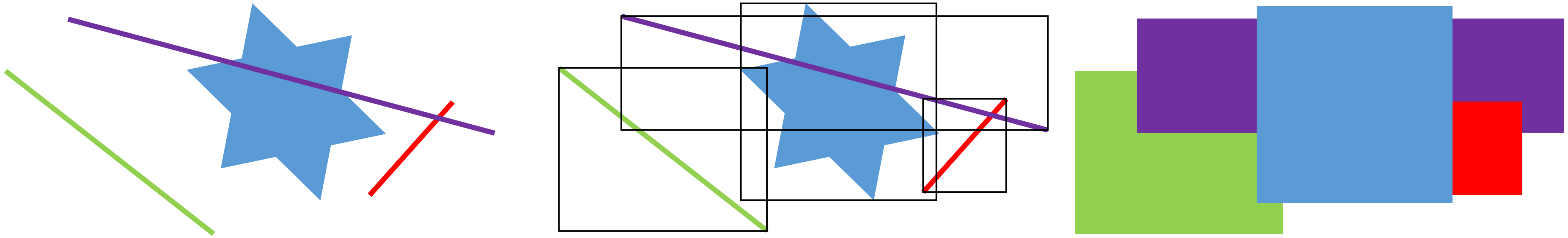
- **Multidimensional indexes** focus on storing spatial objects in such a way that objects close to each other in the space are also close in the structure and on the disk, i.e., **maintain locality**

- As in single dimensional indexing we are interested in **tree or hash structures to avoid sequential scan** of every record in the database



Objects Approximations

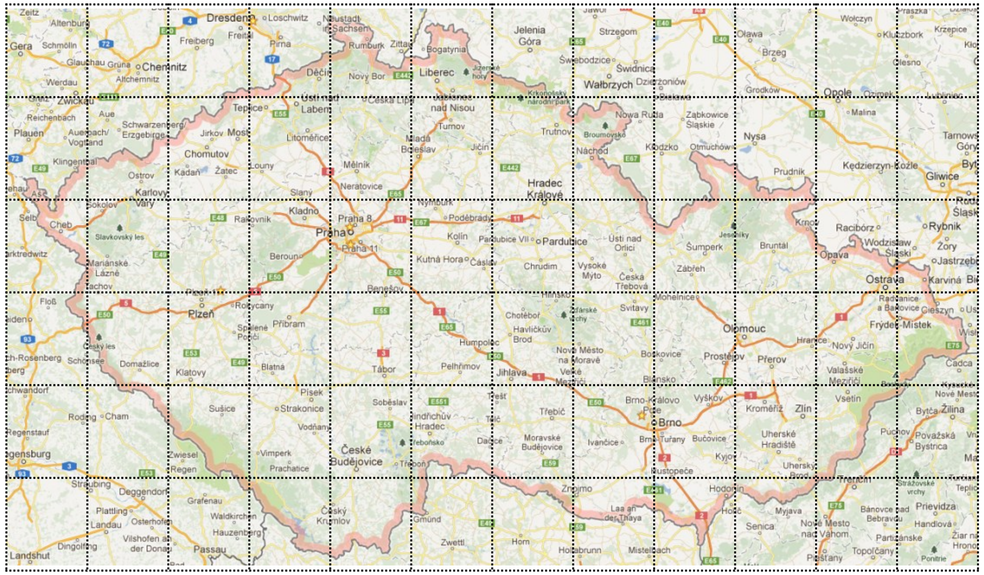
- **General spatial objects** are **more complex** than simple points
- Representing every single point of every single object can be too difficult/expensive for searching
- To easily represent a possibly complex spatial object we use **approximation expressed by (Minimum) Bounding Rectangle/cube/box/object (MBR)** *(minimální ohraničující obdelník – MOO)*
- Comparison of objects is the reduced to the comparison of their MBRs



Grid-Based Indexing

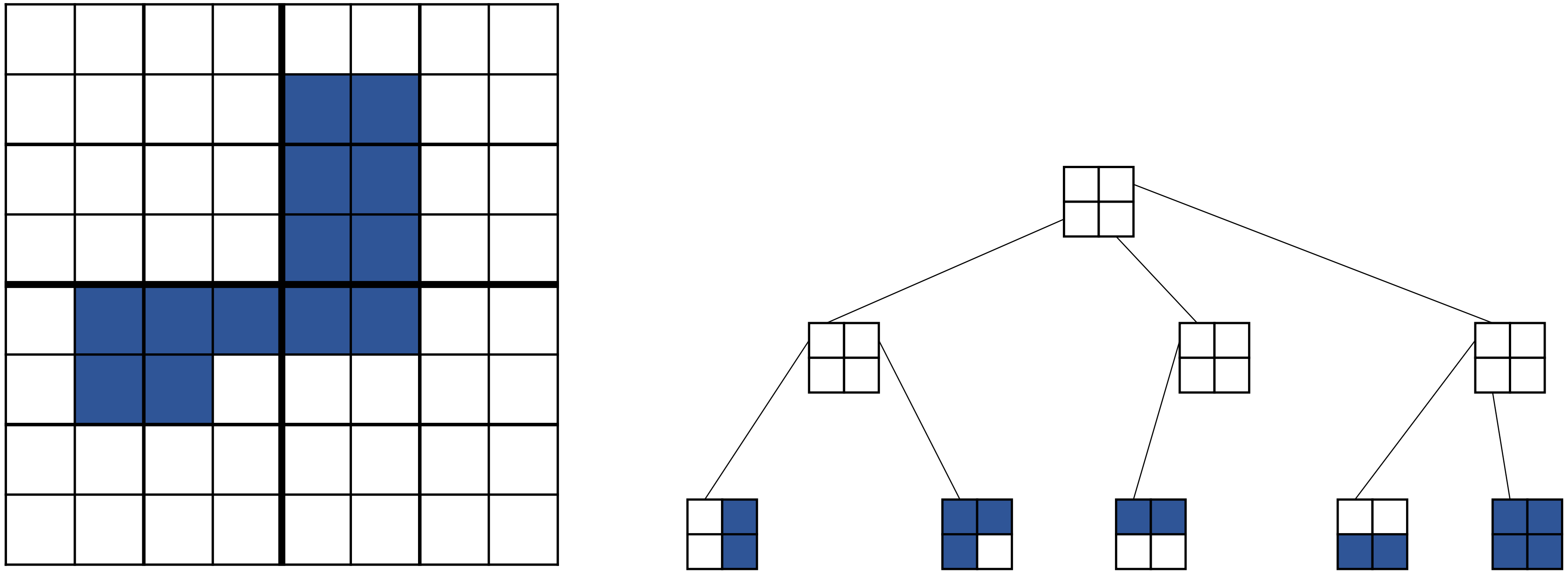
- **N-dimensional grid** covers the space and is **not dependent on the data distribution** in any way, i.e. the grid is formed in advance
  - every **point** object can be **addressed by the grid address**
  - basically **corresponds to hashing** where a grid cell corresponds to a bucket

- **Objects distribution** in the grid **does not have to be uniform** → **retrieval times** for different grid cells **may differ** substantially for different parts of the space



Quad-Tree

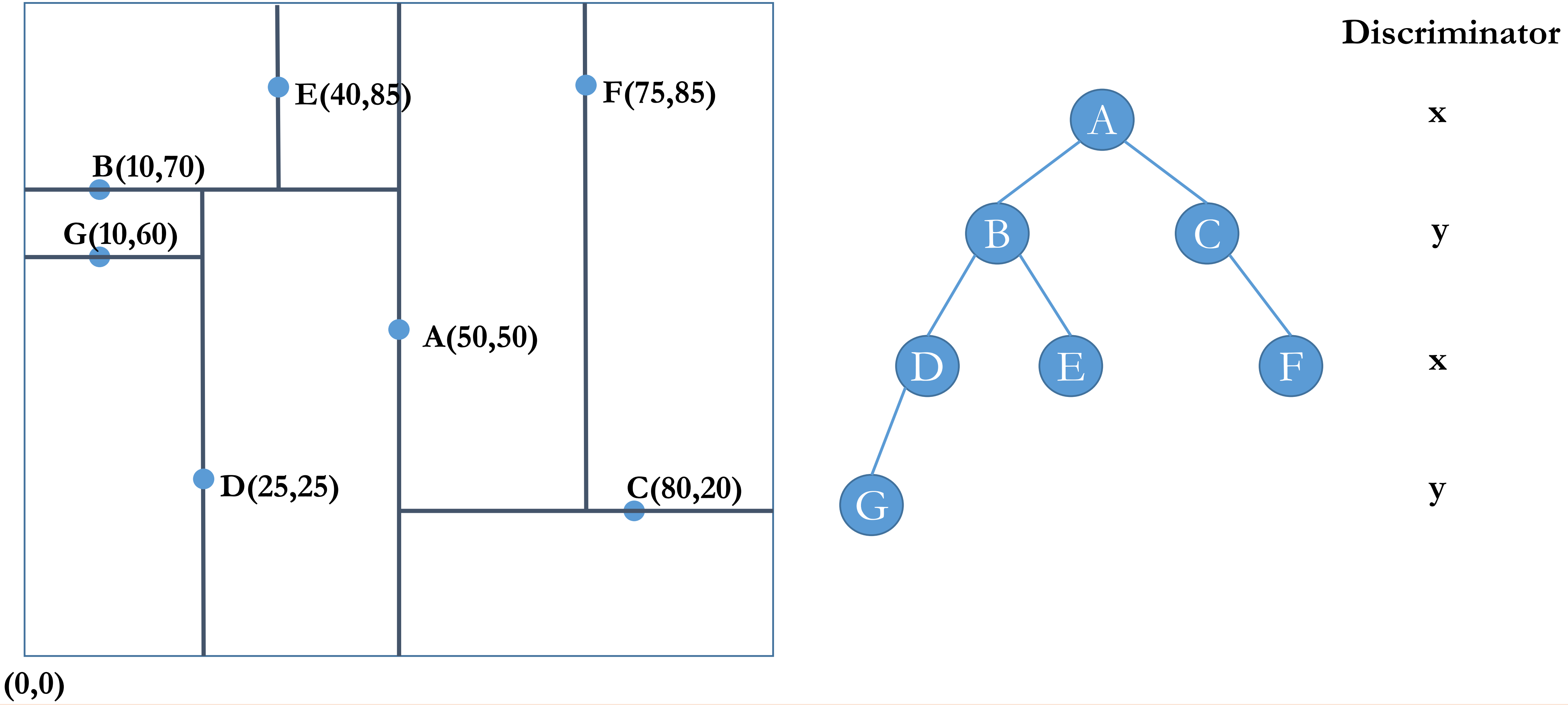
- [Finkel, Bentley; 1974]
- Tree structure representing recursive splitting of a space into quadrants
  - each node has from zero to four children
  - typically the regions are squares (although any arbitrary shape is possible)



k-d-Tree

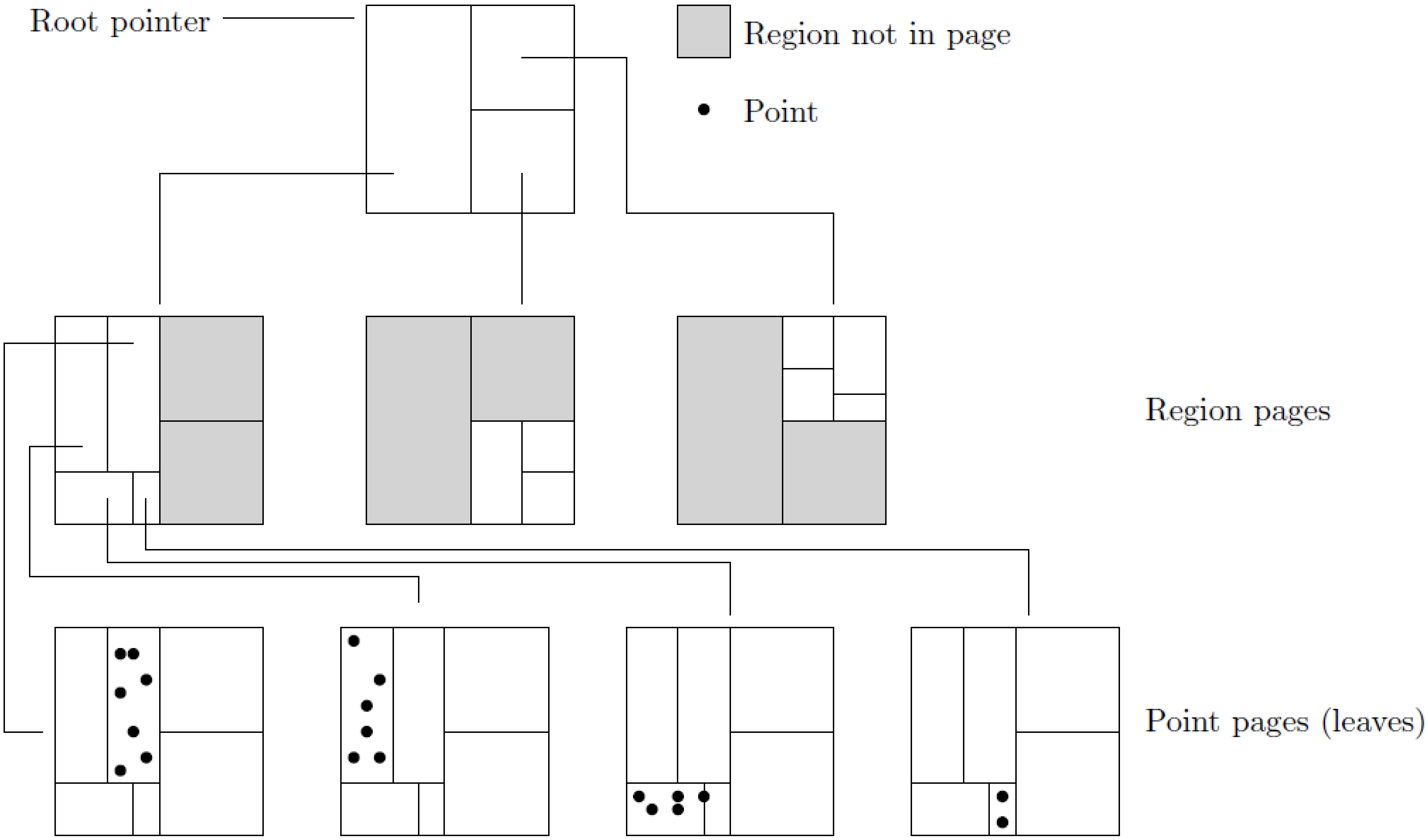
- [Bentley; 1975]
- **k-dimensional tree**
- **Binary tree** where **inner nodes** consist of a **point, an axis identification (hyperplane in nD) and two pointers**
  - **inner nodes** correspond to **hyper planes splitting space into two parts** where the location of the hyper plane is defined by the point
  - points in one part are pointed to by one pointer and the other part by the second one

k-d-Tree Example



K-D-B-Tree

- [Robinson; 1981]
- **Combination of K-D-Tree and B-Tree**
- **k-d-tree** is designed for **main memory**
- In case when the dataset does not fit in main memory it is not clear how to **group nodes into pages** on the disk
- **multiway balanced tree**
- each **node stored as a page** but unlike B-trees **50% utilization can not be guaranteed**
- each **inner node** contains **multiple split axis** to fill the node's capacity
- leaf nodes contain indexed records (points)
- **splitting and merging** happens **analogously to B-trees**



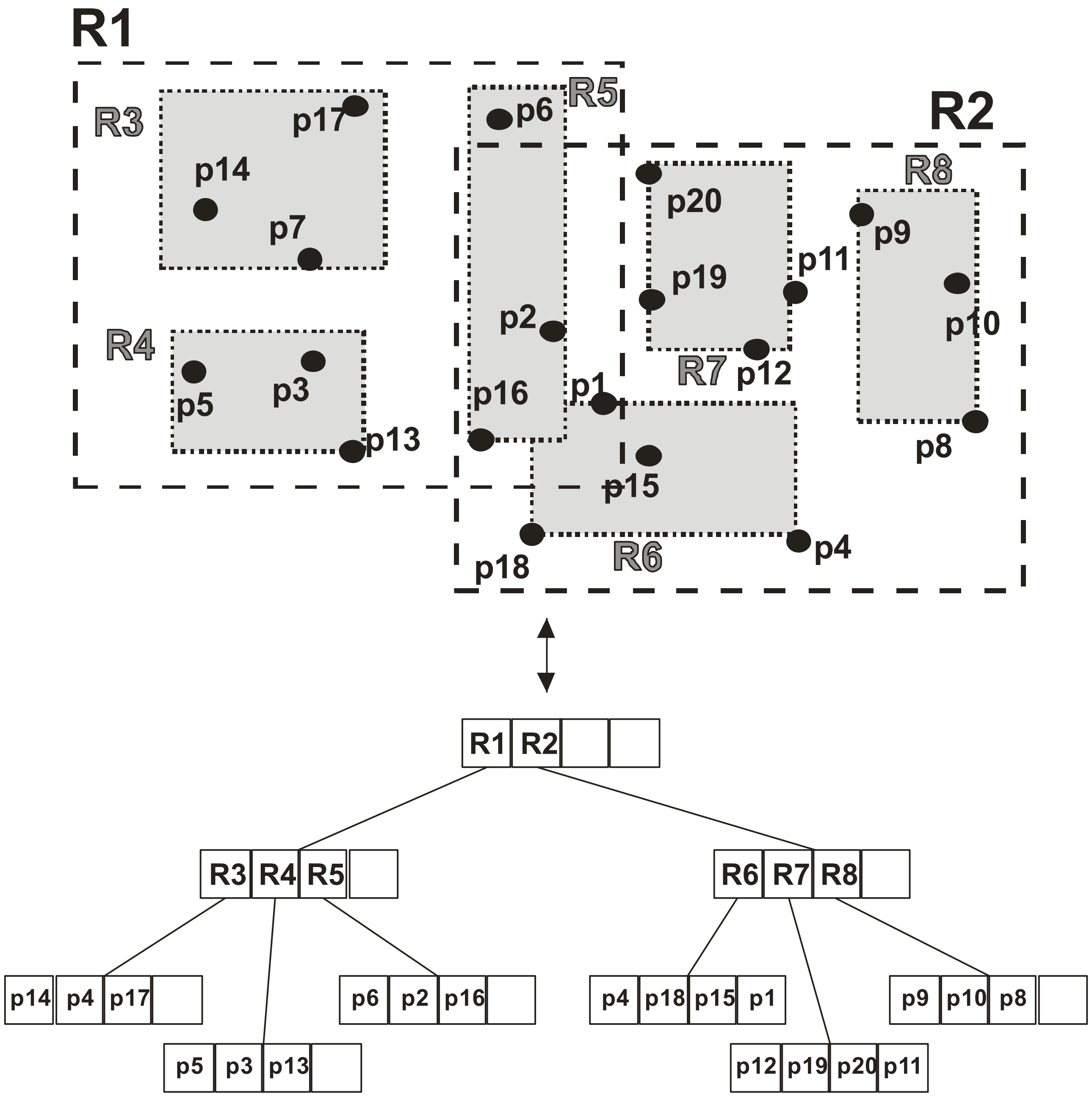
https://en.wikipedia.org/wiki/K-D-B-tree#/media/File:KBDtreeStructure.png

R-Tree

- [Guttman, 1984]
- **Quad-trees and k-d-trees** are not suitable for large collections since they **do not take paging** of secondary memory **into account**
- **K-D-B-tree** is capable of storing **point objects only** and **might not be ballanced**

- **R-Tree** can be viewed as a direct **multidimensional extension of the B+-tree**
- **Leaf records** contain **pointers to the spatial objects**
- **Inner records** contain **MBRs** of the underlying MBRs (or objects)
  - an **MBR corresponding to a node *N* covers** all the objects (MBRs and spatial objects) in **all the descendants of *N***

R-Tree Example



In reality the MBRs (the rectangles) enclose the bounded objects as close as possible.

R-Tree Definition (1)

- **Height-balanced** tree
- **Nodes** correspond to **disk pages**
- Each **node** contains a set of **entries** $\boldsymbol{E}$ consisting of
  - $\boldsymbol{E.p}$ – **pointer to the child node** (inner node) or **spatial object identifier** (leafs)
  - $\boldsymbol{E.I}$ – $\boldsymbol{n}$**-dimensional bounding box** $I = (I_0, I_1, \dots, I_{n-1})$, where I_j corresponds to the extent of the object I along j-th dimension
- Let $\boldsymbol{M}$ be the **maximum number of entries** in a node and let $\boldsymbol{m} \leq \frac{\boldsymbol{M}}{\boldsymbol{2}}$

R-Tree Definition (2)

- Given the labeling from previous slide, R-Tree is an M-ary tree fulfilling the following conditions

  - Every **leaf** contains **between $\boldsymbol{m}$ and $\boldsymbol{M}$ index records**.
  - Every **non-leaf** node other than root contains **between $\boldsymbol{m}$ and $\boldsymbol{M}$ entries**.
  - The **root** has **at least 2 children** unless it is a leaf.
  - For each **record $\boldsymbol{E}$, $\boldsymbol{E.I}$** is the **minimum bounding rectangle**.
  - **All leaves** appear on the **same level**.

- It follows that height of an R-tree with n index records $\leq \log_m n$

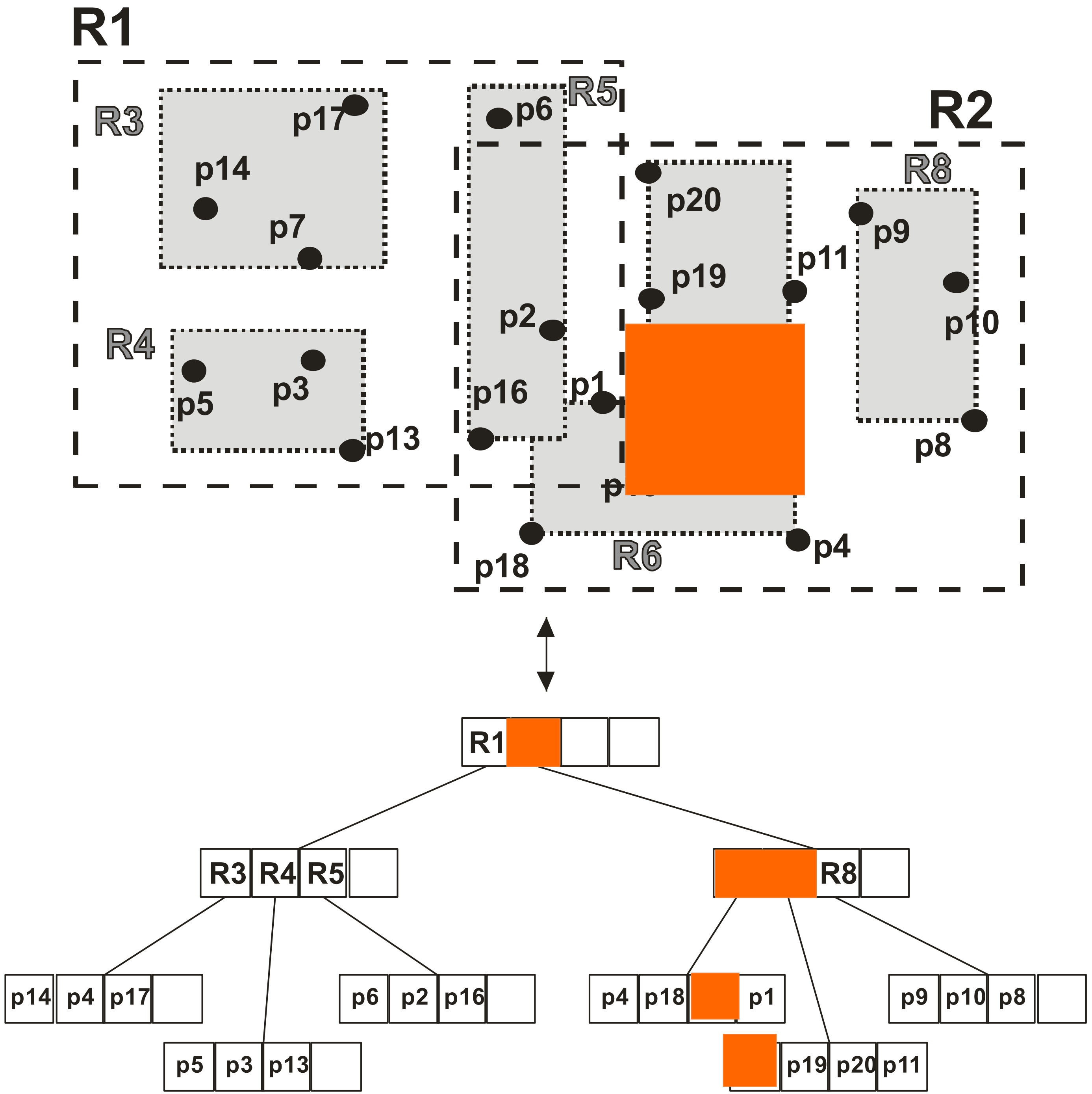
Searching in R-trees (1)

- **Result** of a search is a set of **objects intersecting the query object**
- **Search key** is represented by the **bounding box of a query object**

```
Search_R(T,S)
Input: R-tree with a root T, rectangle S
Output: identifiers of objects overlapping S
  IF T != leaf THEN
    FOR EACH E ∈ T DO
      IF E.I ∩ S THEN Search_R(E.p,S);
  ELSE
    FOR EACH E ∈ T DO
      IF E.I ∩ S THEN Output(E.p,S);
```

Searching in R-Trees (2)

- Unlike in B-trees, the **search procedure can follow multiple paths** → worst-case performance cannot be guaranteed
  - the more the MBRs intersect the worse the performance
- **Update algorithms** force the **bounding rectangles** to be **as much separated as possible** allowing efficient filtering while searching


R1
R2
R3
R4
R5
R6
R8
p17
p14
p7
p6
p20
p9
p11
p19
p2
p10
p5
p3
p16
p1
p13
p8
p4
p18
R1
R3 R4 R5
R8
p14 p4 p17
p6 p2 p16
p4 p18 p1
p9 p10 p8
p5 p3 p13
p19 p20 p11

Inserting into R-trees (1)

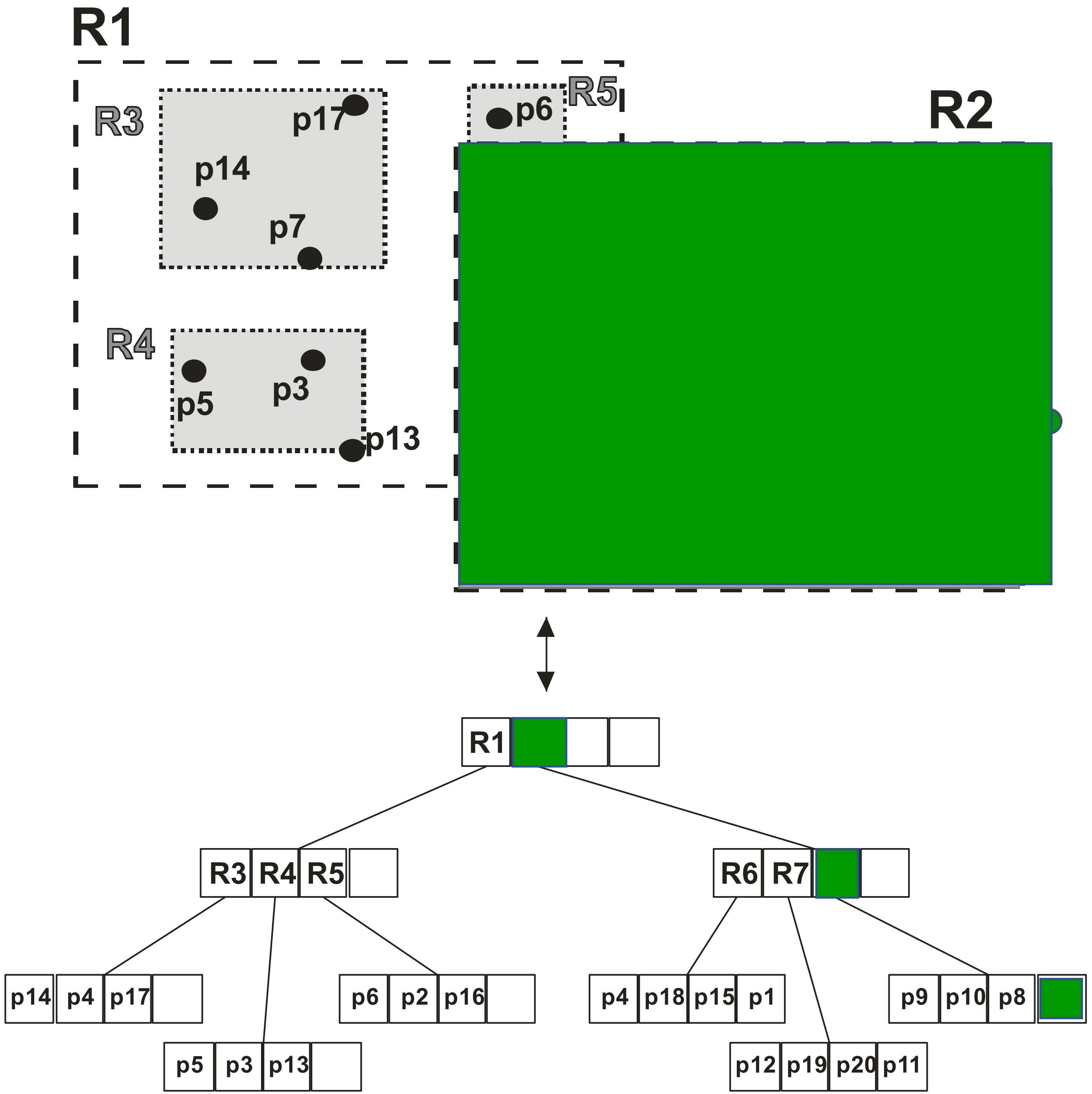
```
Insert_R(T,E)
Input: R-tree with a root T, index record E
Output: updated R-tree

  ChooseLeaf(T,L,E); {chooses leaf L for E}
  IF E fits in L THEN
    Insert(L,E); LL ← NIL;
  ELSE
    SplitNode(L,LL,E)
  AdjustTree(L,LL,T); {propagates changes
upwards}
  IF T was split THEN
    install a new root;
```

Inserting into R-trees (2)

```
ChooseLeaf(T,L,E)
Input: R-tree with a root T, index record E
Output: leaf L

   N ← T;
   WHILE N ≠ leaf DO
      chose such entry F from N whose F.I needs
      least enlargement to include E.I in case
      of tie choose F.I with smallest area;
      N ← F.p;
   L ← N;
```

Inserting into R-trees (3)

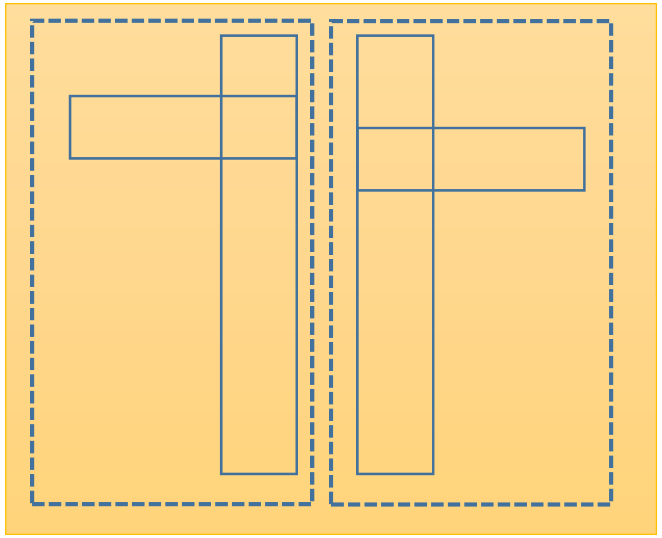
```
AdjustTree(L,LL,T)
Input: R-tree with a root T, leafs L and LL
Output: updated R-tree

   N ← L; NN ← LL;
   WHILE N ≠ T DO
      P ← Parent(N); PP ← NIL;
      modify E_N.I in P so that it contains all rectangles
      in N;
      IF NN != NIL THEN
         create E_NN, where E_NN.p = NN and
         E_NN.I covers all rectangles in NN;
         IF E_NN fits in P THEN Insert(P, E_NN); PP ← NIL
         ELSE SplitNode(P,PP, E_NN)
      N ← P; NN ← PP
   LL ← NN;
```



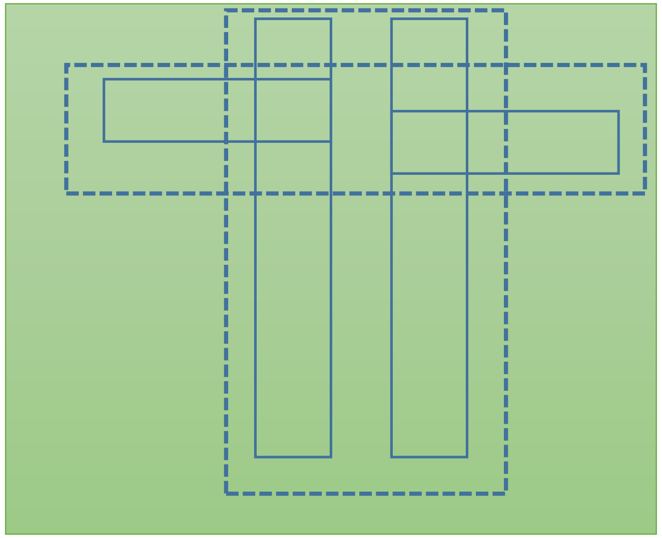
R1
R3
p17
p14
p7
R5
p6
R2
R4
p5
p3
p13
R1
R3
R4
R5
R6
R7
p14
p4
p17
p5
p3
p13
p6
p2
p16
p4
p18
p15
p1
p12
p19
p20
p11
p9
p10
p8

Splitting in R-Tree (1)

- $M + 1$ entries need to be split between two nodes in an **efficient** way
- A good split should **minimize the probability of searching in too much nodes** $\rightarrow$ **total covering area should be minimized = dead space (space not covering any objects) should be minimized**
  - **overlaps are, however, problematic as well**



Bad split



Good split

Splitting in R-Tree (2)

- Approaches to **minimization** of the **total dead space**
  - **Exhaustive algorithm**
    - exhaustive generation of all possible divisions
    - 2^{M-1}
  - **Quadratic cost algorithm**
    - first, a best seed (pair of MBRs) is picked and remaining MBRs are added one by one
    - details in the following slides
  - **Linear-cost algorithm**
    - the seed picking is based on finding rectangles with the greatest normalized separation along each dimension

Splitting in R-Tree (3)

```
SplitNode(P,PP,E)
Input: node P, new node PP, m original entries, new entry E
Output: modified P, PP

   PickSeeds(); {chooses first Ei and Ej for P and PP}
   WHILE not assigned entry exists DO
      IF remaining entries need to be assigned to P or PP in
      order to have the minimum number of entries m THEN
         assign them;
      ELSE
         Ei ← PickNext() {choose where to assign next entry}
         Add Ei into group that will have to be enlarged least
         to accommodate it. Resolve ties by adding the entry
         to the group with smaller area, then to the one with
         fewer entries;
```

Splitting in R-Tree (4)

PickSeeds

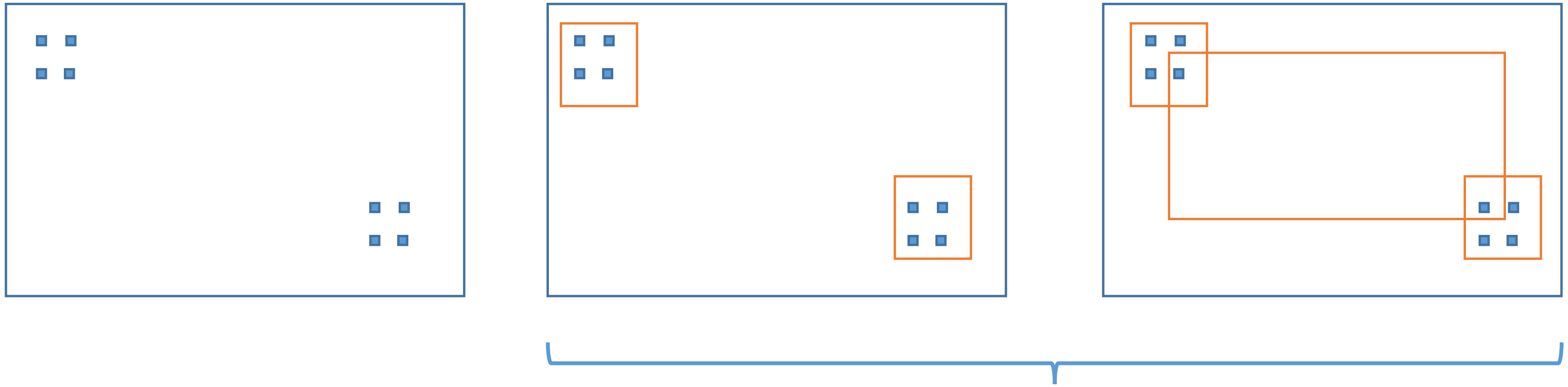
```
FOREACH Ei, Ej (i ≠ j) DO
    dij ← area(J) – area(Ei.I) – area(Ej.I) (J is the
    MBR covering Ei and Ej);
pick Ei and Ej with maximal dij;
```

PickNext

```
FOREACH remaining Ei DO
    d1 ← area increase required for MBR of P and
    Ei.I;
    d2 ← area increase required for MBR of PP and
    Ei.I;

pick Ei with maximal |d1 – d2|;
```

Theoretical Problems with R-Trees

- **Between an object and its MBR can be large dead space**

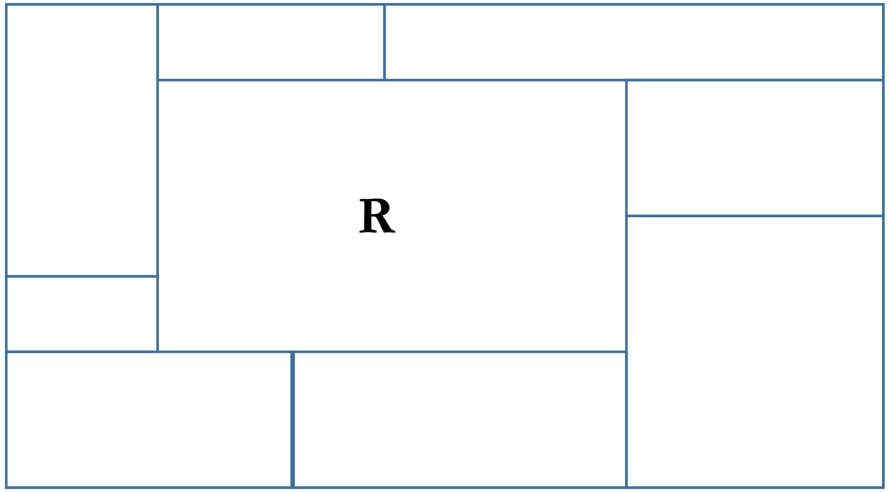


The insert procedure can lead to both arrangements.

- **Is there a way to arrange objects in leaf nodes in a way removing overlap of leafs' MBRs completely?**

Theoretical Problems with R-Trees

- Theorem:
  - For any finite set S of disjoint regions in plane there does not always exist such a set of MBRs where:
    - every region resides in exactly one MBR
    - every MBR bounds n regions where $1 < n < m$
    - intersection of all the MBRs is empty
- Proof:

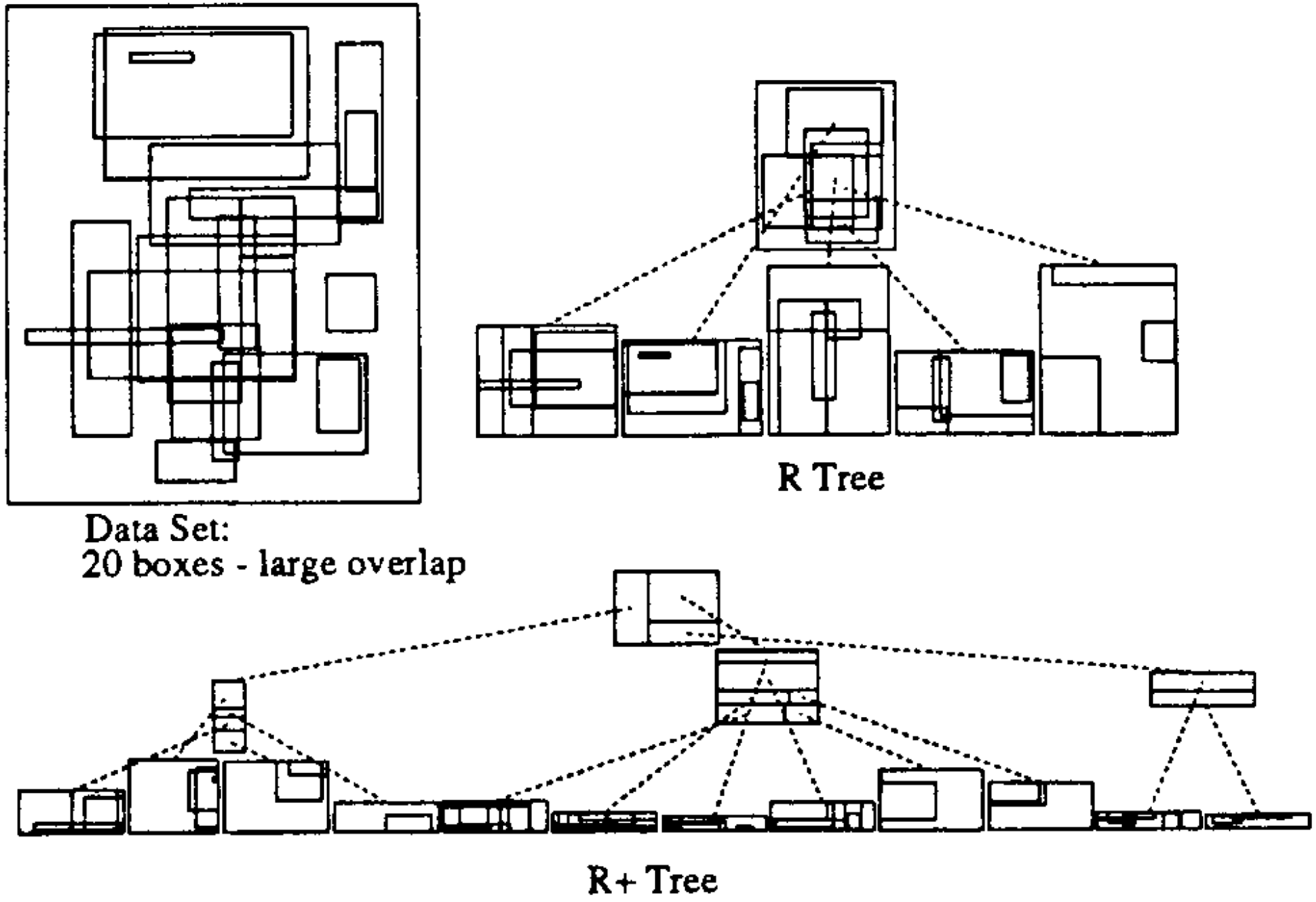


If $m = 8$, given set of regions can not be covered with MBRs meeting the requirements.

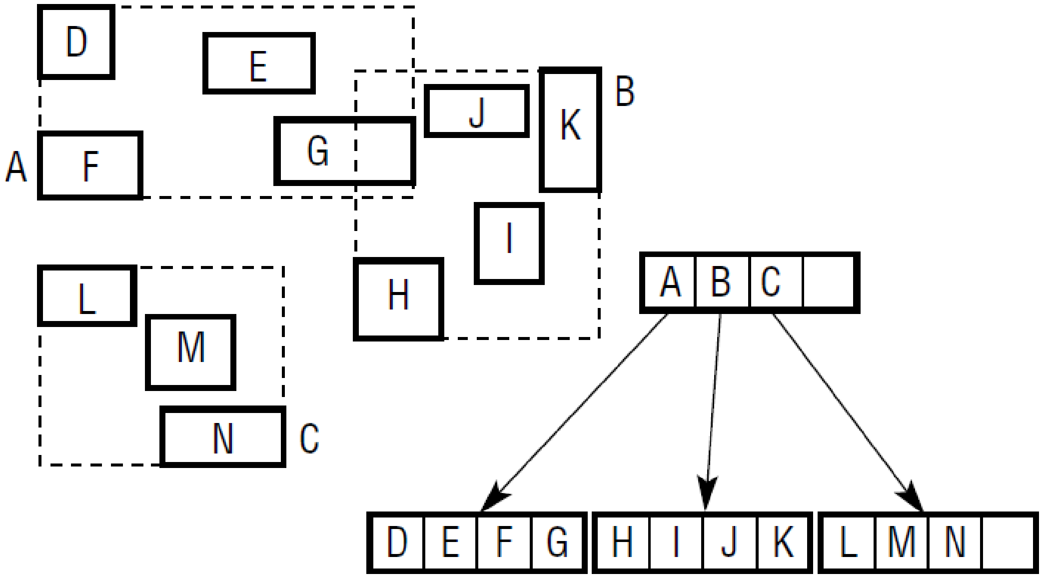
R+-Tree

- [Sellis et al.; 1987]
- MBRs of R+-tree have **zero overlap** while allowing **underfilled nodes** and **duplication** of MBRs in the nodes
  - achieved by splitting an object and placing it into multiple leaves if necessary
- Takes into account not only **coverage** (total area of a covering rectangle) but also **overlap** (area existing in one or more rectangles)

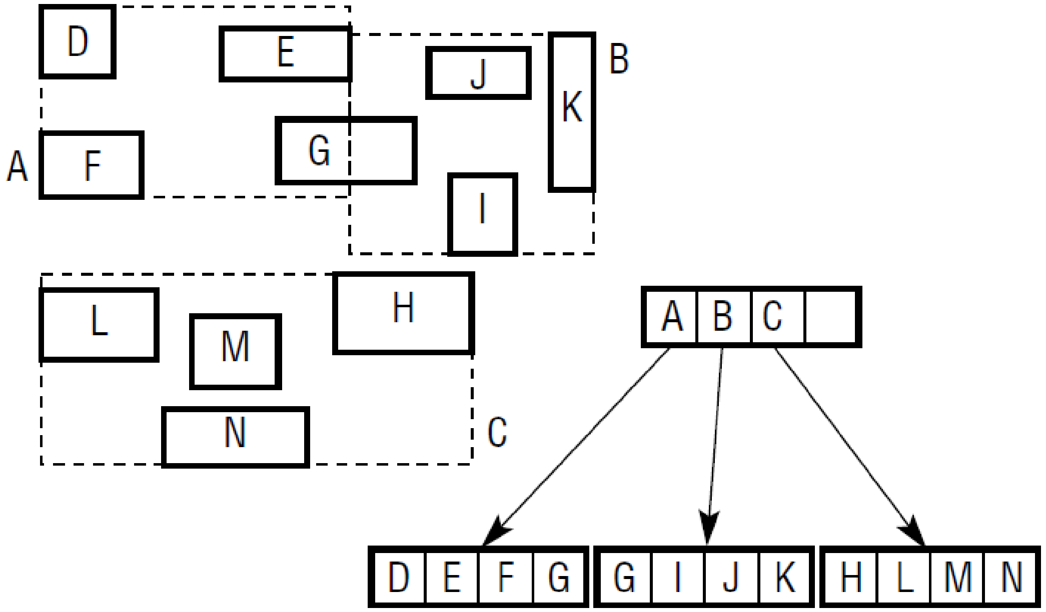
Pros	Cons
• fewer paths are explored when searching • point queries go along one path only	• Overlapping rectangles need to be split → more frequent splitting → higher tree → slower queries



source: Green, D.:;An Implementation and Performance Analysis of Spatial Data Access Methods; 1989



Branching factor M=4



R-Tree (Green) (1)

- [Green; 1989]
- **Modification of the split** algorithm of the original R-tree
- Splitting is based on a **hyperplane** which defines in which node the objects will fall

```
SplitNode_G(P,PP,E)
  ChooseAxis()
  Distribute()
```

R-Tree (Green) (2)

ChooseAxis

PickSeads; {from Guttman's version – returns seeds E_i and E_j}
For every axis compute the distance between MBRs E_i, E_j;
Normalize the distances by the respective edge length of the bounding rectangle of the original node.
Pick the axis with **greatest normalized separation;**

Distribute

Sort E_is in the chosen axis j based on the j-th coordinate.
Add first $\lceil (M+1)/2 \rceil$ records into P and rest of them into PP.

R*-Tree (1)

- [Beckmann et al.; 1990]

- **R*-tree** tries to **minimize coverage(area)** and **overlap** by adding another criterion - **margin**

- Overlap defined as

$$overlap(E_k) = \sum_{i=1, i \neq k}^{n} area(E_k.I \cap E_i.I)$$

R*-Tree (2)

```
ChooseLeaf_RS(T,L,E)
Input: R-tree with a root T, index record E
Output: leaf L

    N ← T;
    WHILE N ≠ leaf DO
        IF following level contains leaves THEN
            choose F from N minimizing overlap(F ∪ E) and solve ties by
            picking F whose F.I needs minimal extension or having
            minimal volume;
        ELSE
            choose F from N where F.I needs minimal extension to I'
            while E.I ⊂ F.I' and area(F.I') is minimal
        N:=F.p
    L:= N
```

R*-Tree Splitting (1)

- Exhaustive algorithm where **entries** are **sorted** first **based on $\boldsymbol{x1}$** axis and second on $\boldsymbol{x2}$

- **For each axis, $\boldsymbol{M - 2m + 2}$ distributions of $\boldsymbol{M + 1}$ entries into 2 groups are determined**
  - in k-th distribution, the first group contains $(m - 1) + k$ entries and the second group the rest, $k = 1, \ldots, M - 2m + 2$

R*-tree Splitting (2)

- For each distribution following so-called **goodness** values are computed (G_i denotes i-th group)
  - **area-value** *(h-objem)*
    - $area\big(MBR(G_1)\big) + area\big(MBR(G_2)\big)$
  - **margin-value** *(h-okraj)*
    - $margin\big(MBR(G_1)\big) + margin\big(MBR(G_2)\big)$
  - **overlap-value** *(h-překrytí)*
    - $area\big(MBR(G_1) \cap MBR(G_2)\big)$

R*-tree Splitting (3)

```
Split_RS
    ChooseSplitAxis(); {Determines the axis perpendicular to which the
    split is performed}
    ChooseSplitIndex(); {Determines the distribution}
    Distribute the entries into two groups;

ChooseSplitAxis
    FOREACH axis DO
        Sort the entries along given axis;
        S ← sum of all margin-values of the different distributions;
    Choose the axis with the minimum S as split axis;

ChooseSplitIndex
    Along the split axis, choose the distribution with minimum overlap-
    value.
    Resolve ties by choosing the distribution with minimum area-value;
```

R*-Tree - Forced Reinsert (1)

- When **inserting into rectangles created long in the past** it can happen that these rectangles **cannot guarantee good retrieval performance in the current situation**

- **Standard split** causes only **local reorganization** of the rectangles

- To achieve **dynamic reorganizations** R*-tree forces entries to be **reinserted during the insertion routine**

R*-Tree - Forced Reinsert (2)

```
OverflowTreatment
    IF the level is not the root level AND this is the first call of
    OverflowTreatment within this Insert THEN Reinsert;
    ELSE Split;

Reinsert
    FOREACH M + 1 entries of a node N DO
        Compute the distance between the centers of their rectangles and
        the center of the bounding rectangle of N;
    Sort the entries in decreasing order of their distances;
    P := first p entries from N; {p is a parameter which can differ for
    leaf and non-leaf node}
    FOREACH E ∈ P DO remove E from N; {Includes shrink of the bounding
    rectangle}
    FOREACH E ∈ P DO Insert(E);
```

Hilbert R-Tree

- [Kamel&Faloutsos; 1994]

- Idea
  - facilitates **deferred splitting** in R-tree
  - **ordering is defined on the R-tree nodes** which enables to define sibling of a node in given order
    - **Hilbert space filling curves**
  - when a split is needed, the overflown entries can be moved to their neighboring nodes thus deferring the split
  - **search procedure identical** to ordinary **R-tree** (i.e. the idea of a MBR covering its descendants' MBRs still holds)

Hilbert R-Tree Definition

- **Hilbert value** of a rectangle – **Hilbert value of its center**

- **Hilbert R-tree**
  - behaves exactly the **same as R-tree on search**
  - on **insertion** supports **deferred splitting** using the Hilbert values

  - **leaf nodes** contain pairs **(R, obj_id)**
    - **R** - MBR of the indexed object; **obj_id** – indexed object's id (pointer)
  - **non-leaf nodes** contain triplets **(R, ptr, LHV)**
    - **R** - MBR of the region corresponding to the entry; **ptr** – pointer to subtree; **LHV** – largest Hilbert value among the data rectangles enclosed by R

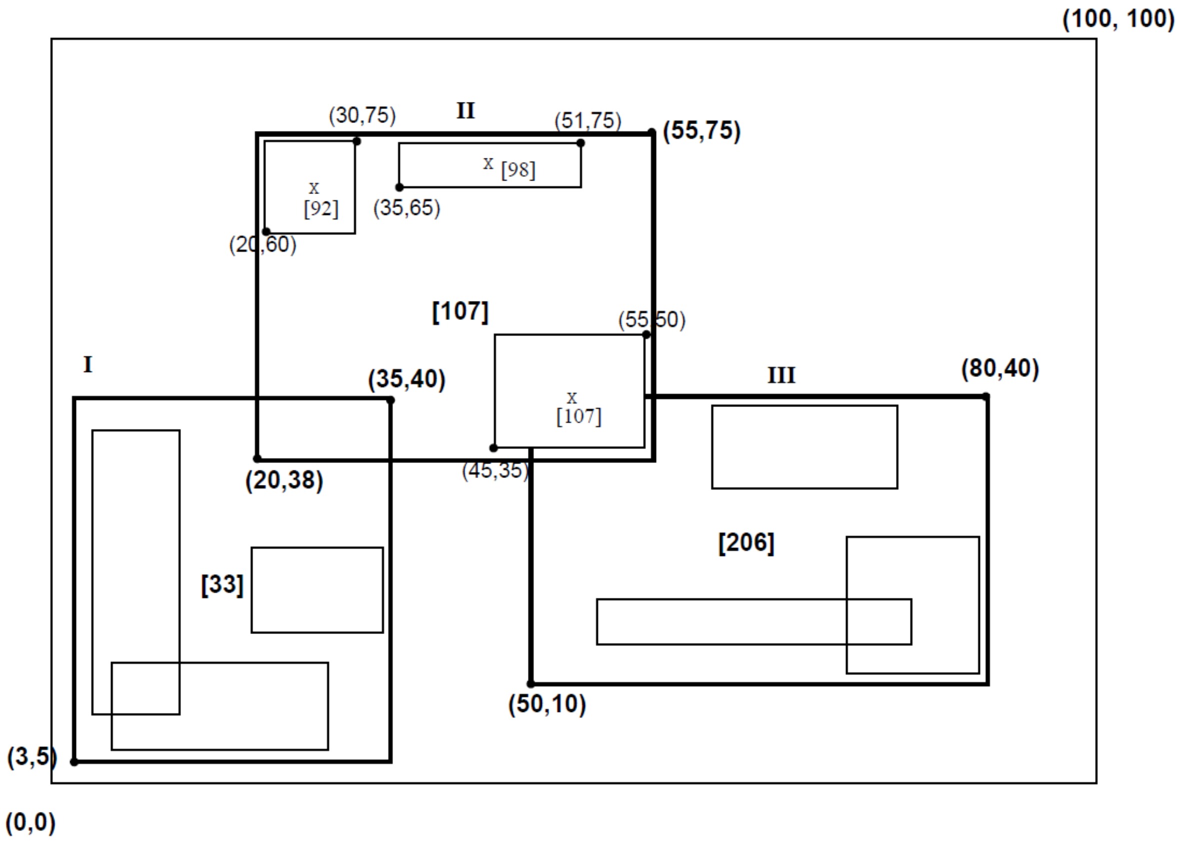
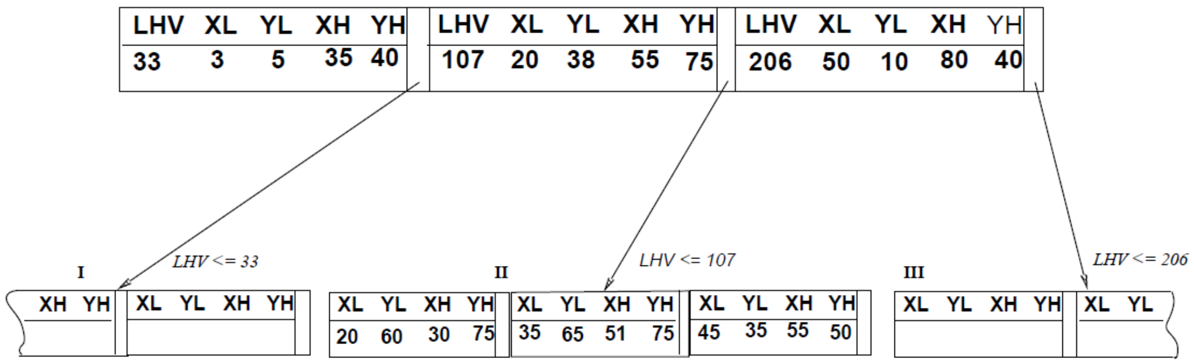


Figure 2: *Data rectangles organized in a Hilbert R-tree*



Hilbert R-Tree - Splitting

- Hilbert R-tree can **parameterize the splitting procedure** by setting **order of the splitting policy**
  - **normal** R-tree splits **1 node into 2** → **1-to2 splitting policy**
  - when we defer the split up to the point when 2 nodes are full we get 2-to-3 policy

  - **order of the splitting policy** s $= s$ nodes split to $s + 1$

- **Splitting procedure**
  - When a **node overflows** it tries to **push** some of its entries to one of its s-1 siblings called **cooperating siblings**
  - if all of them are **full, s-to-(s+1) split** occurs

Hilbert R-Tree Insertion (1)

```
Insert(T,R)
Input: R-tree with a root T, rectangle R
Output: Modified tree

    LL ← NIL;
    L ← ChooseLeaf(T, R, Hilbert(R));
    IF L has empty slot THEN Insert(L, R);
    ELSE LL ← HandleOverflow(L, R);

    S ← L U cooperating siblings of L U LL;

    AdjustTree(S);
    IF root was split THEN Install new root;
```

Hilbert R-Tree Insertion (2)

```
ChooseLeaf(T, R, h)
Input: R-tree with a root T, rectangle R and its
Hilbert value h
Output: Leaf where R should be inserted

   N ← T;
   WHILE N is not leaf DO
      E ← entry in N with minimum LHV greater
      than h;
      N ← E.ptr;
   RETURN N;
```

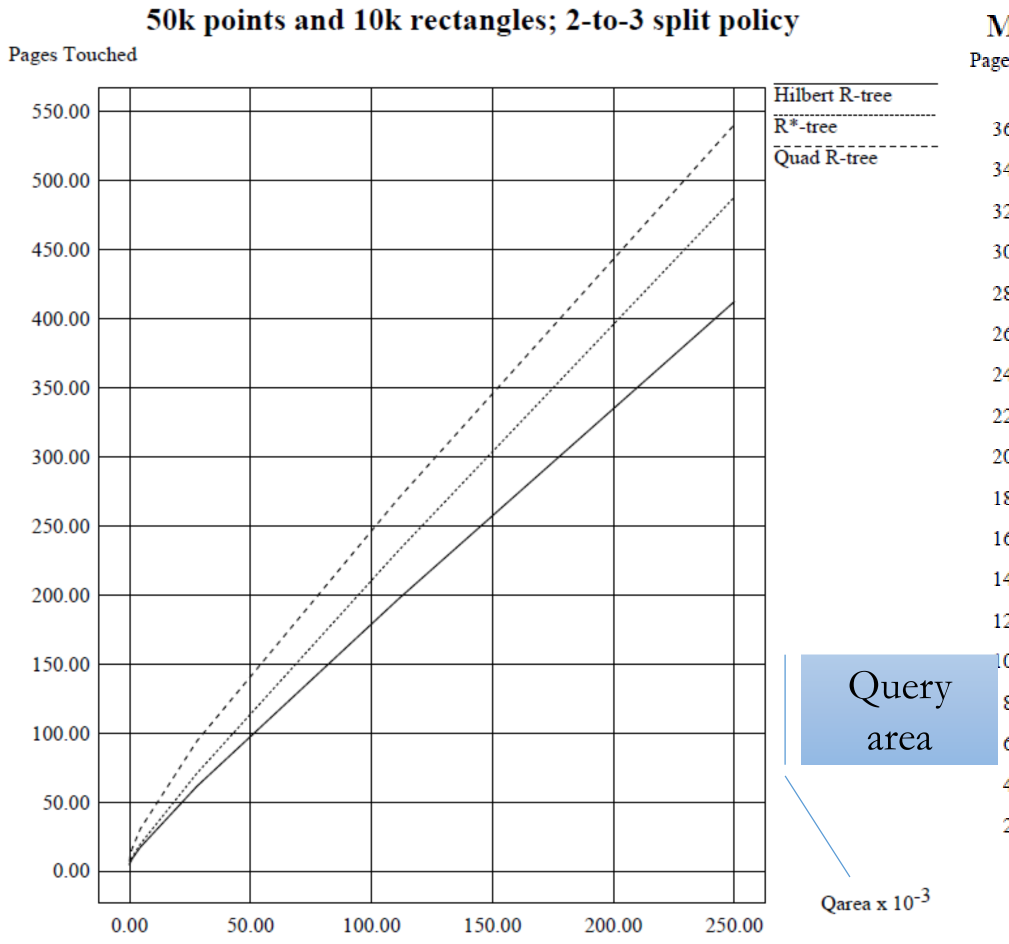
Hilbert R-Tree Insertion (3)

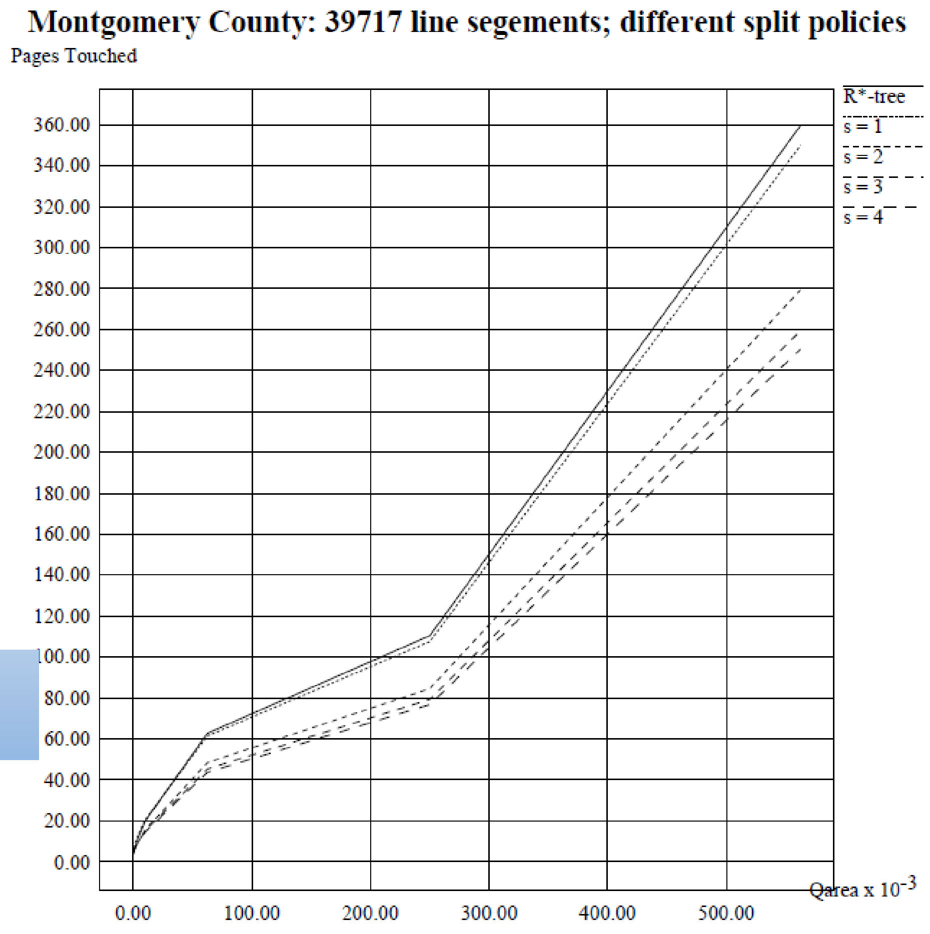
```
HandleOverflow(N, R)
Input: Overflown node N, rectangle R
Output: Modified tree

   NN ← NIL;
   A ← set of all entries of N and its s−1 cooperating siblings;
   IF at least one of the s−1 siblings has a free slot THEN
      Distribute A evenly among the s nodes according to the
      Hilbert value;
   ELSE
      NN ← CreateNode();
      Distribute A evenly among the s+1 nodes according to the
      Hilbert value;
   RETURN NN;
```

Hilbert R-Tree Insertion (4)

```
AdjustTree(S)
Input: Set of affected nodes
Output: Modified tree
   WHILE S ≠ root DO
      PP ← NIL;
      FOREACH N ∈ S DO
         NP ← Parent(N);
         IF N has been split THEN
            NN ← the new node;
            IF NP has an empty slot THEN
               Insert NN in NP in the correct order according to its
               Hilbert value;
            ELSE
               PP ← HandleOverflow(NP, NN.R);
         Adjust MBRs and LHVs in NP based on values in N;
      S ← Parents(S) ∪ PP;
```

Experimental Results





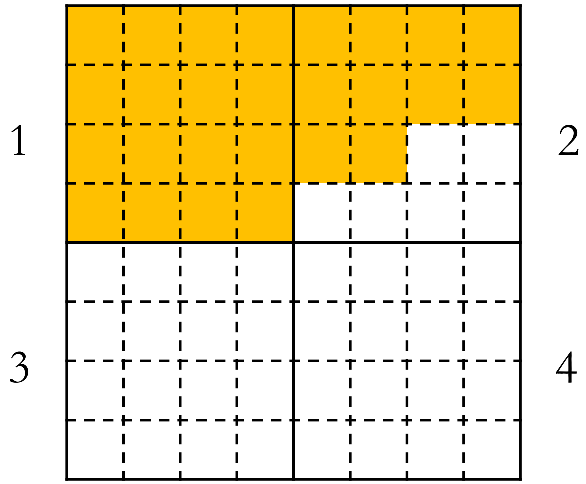
UB-Tree

- [Bayer; 1996,1997]
- Idea
  - **combination of B-tree and Z-curve**
  - multidimensional **space** is **partitioned into Z-regions** being mapped to **one page** of the secondary storage
  - unlike Hilbert R-tree, **both insert and search procedures work with single dimension (defined by the Z-curve)**
  - **better than R-tree for high-dimensional data**

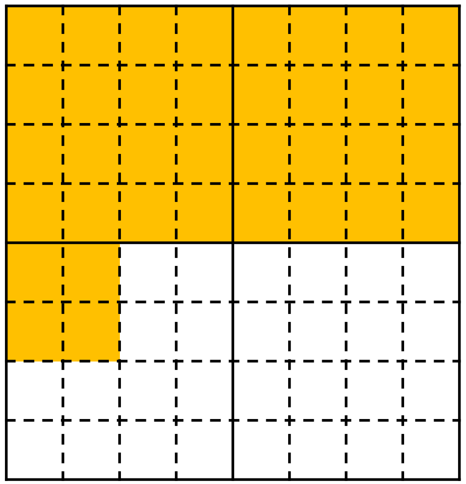
UB-Tree – Address (1)

- **Address of an area** starting at position $(0; \dots; 0)$ in the "upper left corner" corresponds to the **Z-value of its "lower right corner"**
  - **Z-address of an area $\boldsymbol{\alpha}$** ($area(\alpha)$) is a **sequence $\boldsymbol{i_1 i_2 \dots i_l}$**, where $\boldsymbol{i_j \in <0; 2^n - 1>, j < l}$ and $\boldsymbol{i_j \in <1; 2^n - 1>, j = l}$, n being dimension of the space
    - we can construct Z-address of α recursively dividing the space in half along each of the axis (forming 2^n) cubes and **at level $\boldsymbol{j}$, $\boldsymbol{i_j}$ corresponds to the number of subcubes fully covered by $\boldsymbol{\alpha}$ in the z-order**

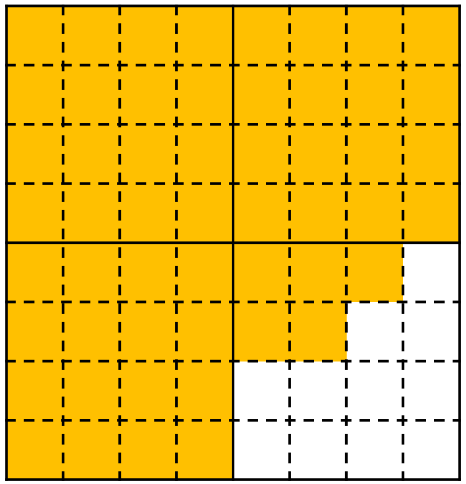
UB-Tree – Address (2)



Address: 1.2.2



Address: 2.1

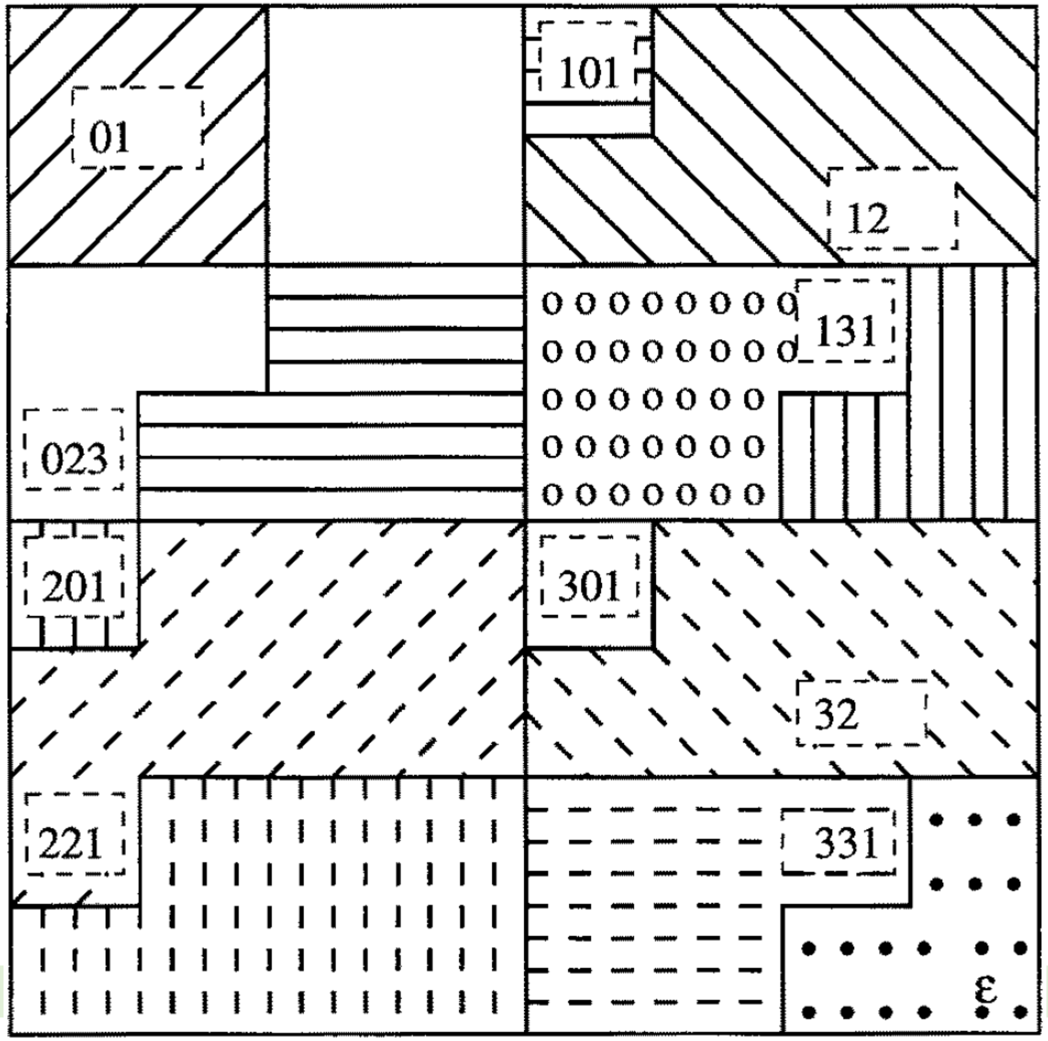


Address: 3.1.1

UB-Tree - Regions

- A **region** is a **difference of two areas** α, β $(\alpha \subseteq \beta)$, i.e

$$region(\alpha, \beta) = area(\beta) \setminus area(\alpha)$$



- Each **page $\boldsymbol{p}$ holds objects between successive addresses $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$**, i.e.

$$content(P) = set\ of\ objects\ in\ region(\alpha, \beta)$$

- **Ordering of pages** is defined by **ordering of the regions**

Regions bounded by two addresses
(0, 01), (01,023), (023, 101), (101, 12), (12, 131)
(131, 201), (201,221), (221,301), (301, 32)
(32, 331), (331, ∞)

UB-Tree - Insert

- **Points**
  - a **point** can be defined as a **region on the highest level of resolution** (smallest possible subcube), i.e., a **point is a region γ belonging to a unique region (β, δ)** which defines the page where the point will be inserted
  - the correct region can be found using point query (follows)

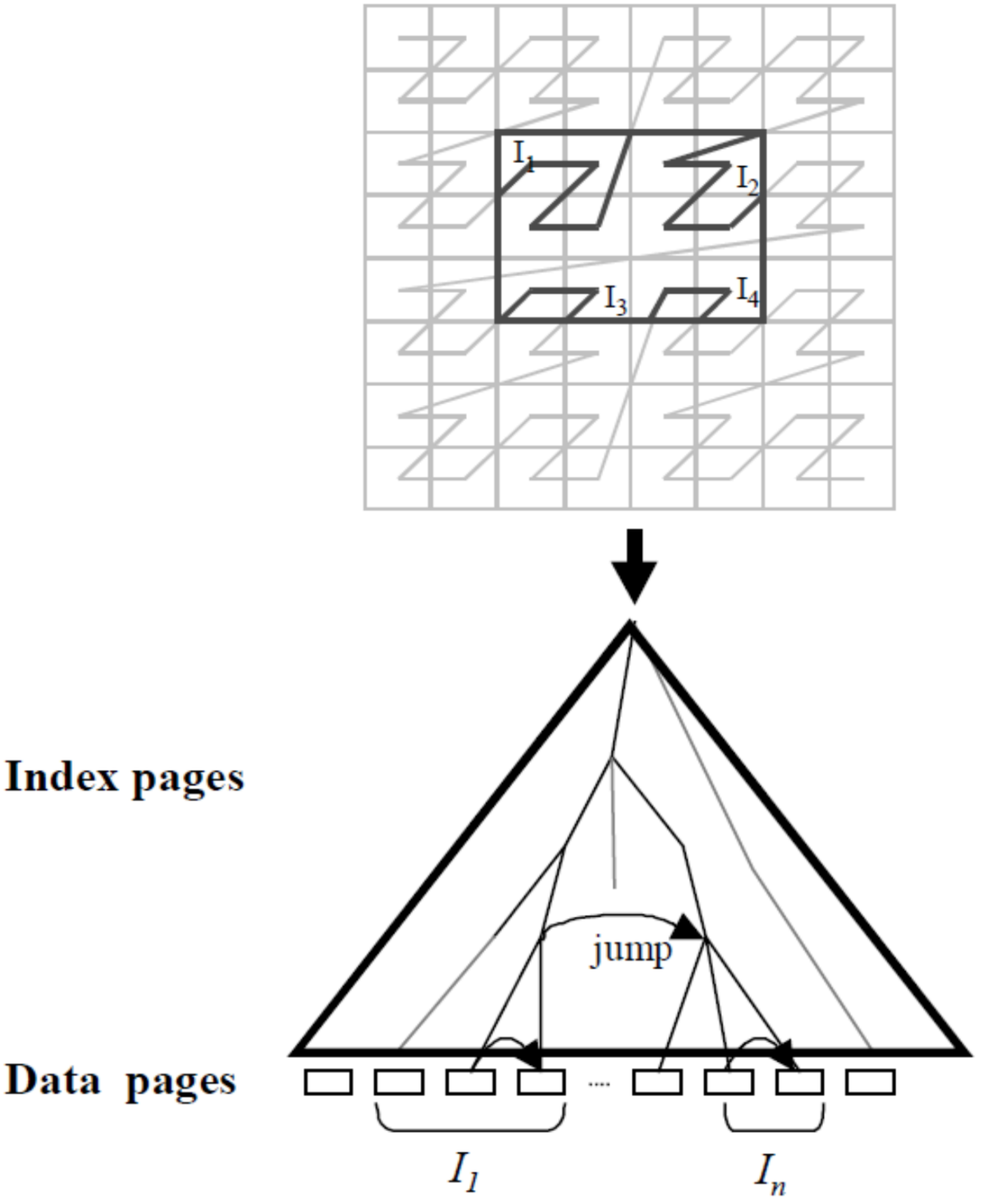
- **Complex objects**
  - a complex object **intersects several regions**
  - **id of an object O** to be inserted is inserted **into every region** (=page) **which intersect O**
  - insertion of a complex object **can lead to multiple region** (=page) **splits**

UB-Tree - Split

- Pages can store **maximum number of $\boldsymbol{M}$ entries**
- **Splitting** of a region (α, γ) **introduces a new area** with address $\boldsymbol{\beta}$ such that $\alpha \subseteq \beta \subseteq \gamma$
- Region $(\boldsymbol{\alpha}, \boldsymbol{\gamma})$ **is split into regions** $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ **and** $(\boldsymbol{\beta}, \boldsymbol{\gamma})$ and **objects** are **evenly distributed** between the regions

UB-Tree - Search

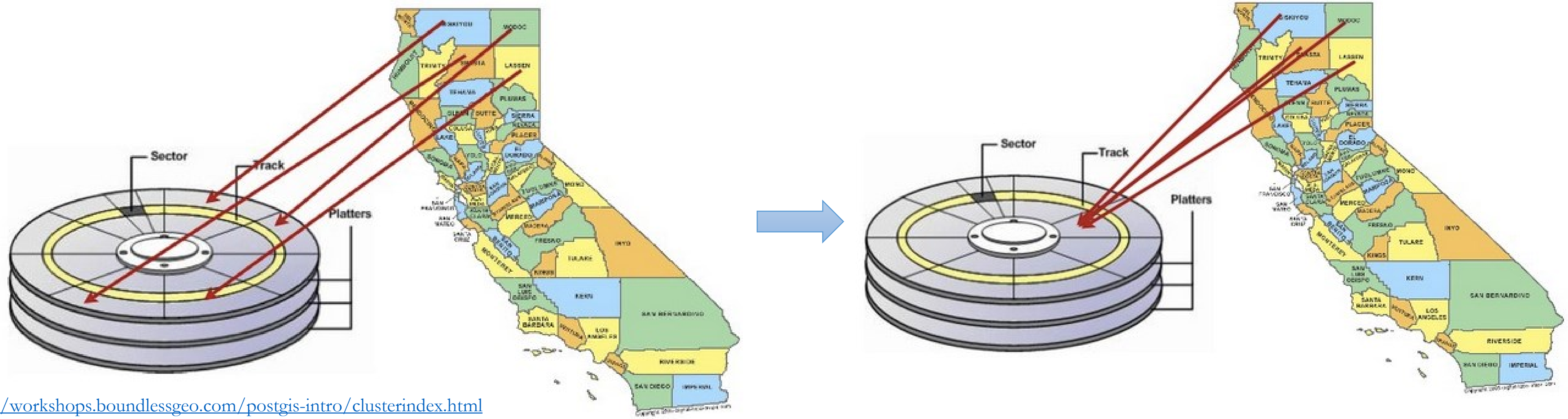
- **Point query**
  - address of the query is computed and page containing this address is identified (B-tree search)
  - the found page is checked for the existence of the query
- **Range query**
  - let the **query rectangle** $\boldsymbol{q}$ be defined by its low and high **bounding points** $(\boldsymbol{ql_1}, \boldsymbol{ql_2}, \dots, \boldsymbol{ql_n}), (\boldsymbol{qh_1}, \boldsymbol{qh_2}, \dots \boldsymbol{qh_n})$ and let $\boldsymbol{\lambda} = \boldsymbol{address}(\boldsymbol{ql}) \in \boldsymbol{region}(\boldsymbol{\alpha_{j-1}}, \boldsymbol{\alpha_j})$
  - we fetch **all objects in** $\boldsymbol{region}(\boldsymbol{\alpha_{j-1}}, \boldsymbol{\alpha_j})$ and **check** their intersection with q
  - let $\boldsymbol{\beta}$ be the address of the **last subcube** of $region(\alpha_{j-1}, \alpha_j)$ **which intersects** $\boldsymbol{q}$
  - we must **check all older brothers of** $\boldsymbol{\beta}$ **to exhaust the father of** $\boldsymbol{\beta}$ which comprises of possible descent to the lowest level
  - then we **check grandfather**, …



I1
I2
I3
I4
Index pages
jump
Data pages
....
I1
In

Clustering on Spatial Indexes

- Having a **spatial index does not enforce** the data to be stored on disk in any special way
  - **spatial data** tend to be **accessed** in a way **corresponding to their spatial arrangement** due to the use of spatial queries
- **Clustering** – data to be retrieved together are located in similar physical positions on the disk
  - **spatial index defines an ordering scheme** which can be used when retrieving data



source: http://workshops.boundlessgeo.com/postgis-intro/clusterindex.html