Spatial Databases

DUŠAN KOLÁŘ
KOLAR@FIT.VUT.CZ

Contents

1. What is a spatial database?
2. Key issues connected with spatial data storage
3. How to store the spatial data
4. **Indexing in** (not only) **spatial databases**

What Is a Spatial Database?

Topics

1. Space storage – what do we mean by a space?
2. Tiling the Earth
3. What is the point?
4. Basic data types

Space Storage

Spatial data

- Space around us (streets, woods, cities, roads, power lines, etc.) – 2D/3D
- Spatial structure of chemicals – 3D
- VLSI circuits – 2D/3D
- Space as such (planets, suns, galaxies) – 3D
- Pipes, lines – water, oil, data/telephony – 2D/3D

Virtual spatial data

- Vectors of characteristics (e.g. of text, image, video, sound, etc.) – nD ($n >> 3$)
- Vectors of numerical attributes (e.g. date, price, size, etc.) – nD (tpy.: $2 < n < 8$)

Space Storage

Is it sufficient to store the space?

Is it sufficient to index the data?

What is so interesting and important about spatial data storage?

Relations among data in the database!

- All notebooks not more expensive than …, with memory at least … GB, weight at most …
- All cities/towns/villages close to lake …
- All chemicals with similar spatial structure to …
- All roads in the district … that are closer to power lines than …

Tiling the Earth – Example

1. Let’s make orthophotos and put them to tiles to cover the district (Earth)
2. Store them to the database (correct numbering – which one ???)
3. Let’s make an application to show the tiles correctly

Is that database spatial?

4. Convert the photos to pictures (just a few colors – roads, buildings, water, other)
5. Store these pictures next to orthophotos
6. Adjust the application

Is the database spatial?

Tiling the Earth – Example

7. Take the pictures and extract “vector” features
   I. Roads as lines with capacity
   II. Buildings as their area with border
   III. Rivers as lines with flow, lakes as an area
   IV. Etc.
8. Store the “vector” data
9. Adapt application to show features, calculate routes, sizes, showing surroundings, filtering features, etc.

Is the database spatial?

What Is the Point?

Data showing a space may not be spatial!

Spatial data carry spatial information that can be manipulated by a computer

Spatial database management system recognizes spatial data as a special kind of data and provides operations over these spatial data (e.g. size, length, distance, mutual relationship, difference, computation of convex hull)

THUS

Application showing some of these features may not be working on top of spatial database system – this is definitely NOT a good approach.

Application relying on spatial database system not performing computations on its own, but exploiting spatial database system, IS a good approach. This holds for data in virtual spaces too.

What Is the Point?

Space, which one wants to describe and store, is decomposed to rather simple elements

- Abstraction

During the process some information is lost (neighbors, relations, containment, etc.)

- Consequence of the abstraction

Required information can be computed based on spatial information

- Reconstruction of information lost during abstraction, some “new” may be discovered

Computation is performed in the database management system

- Otherwise the DBMS is not spatial ☺

Other database operations are available too, various kinds of information may be combined, interleaved, etc.

Basic Data Types

Points

- Cities/towns/…
- Endpoints of vectors in virtual data space
- Planets, suns, …
- Atoms in chemicals

At least 2D (for 1D we can handle it)

Storage of uniform data kinds

Not sufficient for many applications

Intervals define points

Basic Data Types

Lines, poly-lines

- Roads/routes
- Pipe-, power- lines
- (Inter-)connections

At least two points in higher dimensional space

Variable length

Should describe uniform data kinds (within database table)

Not sufficient for many applications (even if points are taken into account)

Basic Data Types

Polygons

- Borders of cities, districts, states, etc.

At least three points

Variable length

Should describe uniform data kinds (within database table)

Not sufficient for many applications (even if points and (poly)lines are taken into account)

Basic Data Types

Areas/volumes

- Woods, fields, …
- Lakes, seas, oceans, …
- Masses

At least 3 points (in volume 4)

Variable length

Should describe uniform data kinds (within database table)

Not sufficient for many applications (even if preceding data types taken into account)

So What Is a Spatial Database?

Such a database management system that can efficiently store, query, and manipulate spatial data (such as points, lines, polygons, areas, volumes, …). Such data are natural part of the DBMS and all operations available for basic (e.g. plain relational) data are available for spatial data too (if it makes sense).

Key Issues of Spatial DBMS Implementation

Topics

1. Continuous and discrete space
2. Discrete arithmetic operations
3. Reality mapping
4. Mutual relation among spatial elements
5. Operations over spatial data
6. Storage and indexing – see following sections

Continuous and Discrete Space

Usually, we use real numbers to represent space coordinates and, thus, elements in a space

- $(x, y) \in \mathfrak{R}^2$ $(x, y, z) \in \mathfrak{R}^3$ Euclidean space

Even if we round computation to some value, we do perform computation in real numbers

Do computers perform computation in real numbers?

- NO! Usually, they compute with some small subset of rational numbers only (real numbers model is not used).

Why?

- Computers are finite and quite small… And there are real numbers, which do not have finite representation as a number, only as an expression, e.g. $\sqrt{2}$

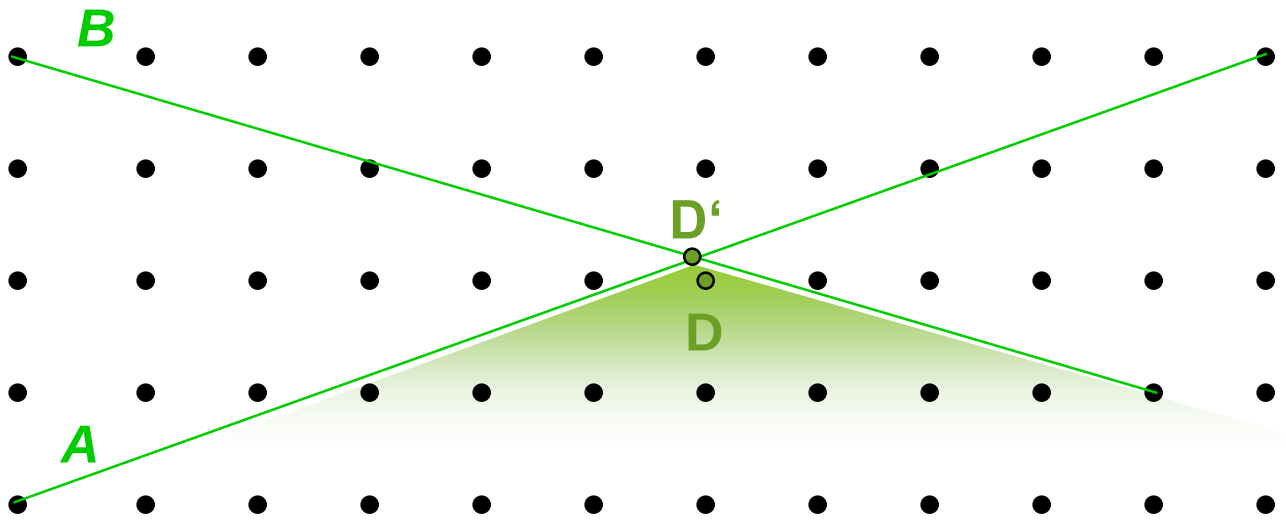
Thus…

Continuous and Discrete Space

We use some finite, discrete approximation of real numbers, e.g. 000 000.000

Well, fine, where's the problem?

- Results of some operations cannot be represented within the discrete space approximation
- Rounding error may change result of some queries…
  - Is point D on the line segment A?
  - Is point D in the sub-space delimited by line segments A and B? (shaded quadrant)



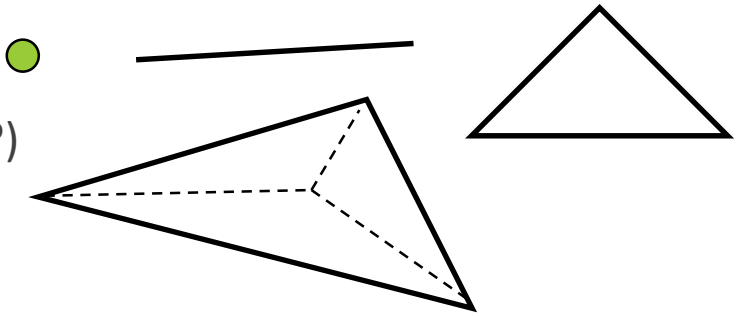
Continuous and Discrete Space

How to handle the discrete space?

- Models for representation
- Hiding the computation and problems from a user (not possible always)

Various approaches

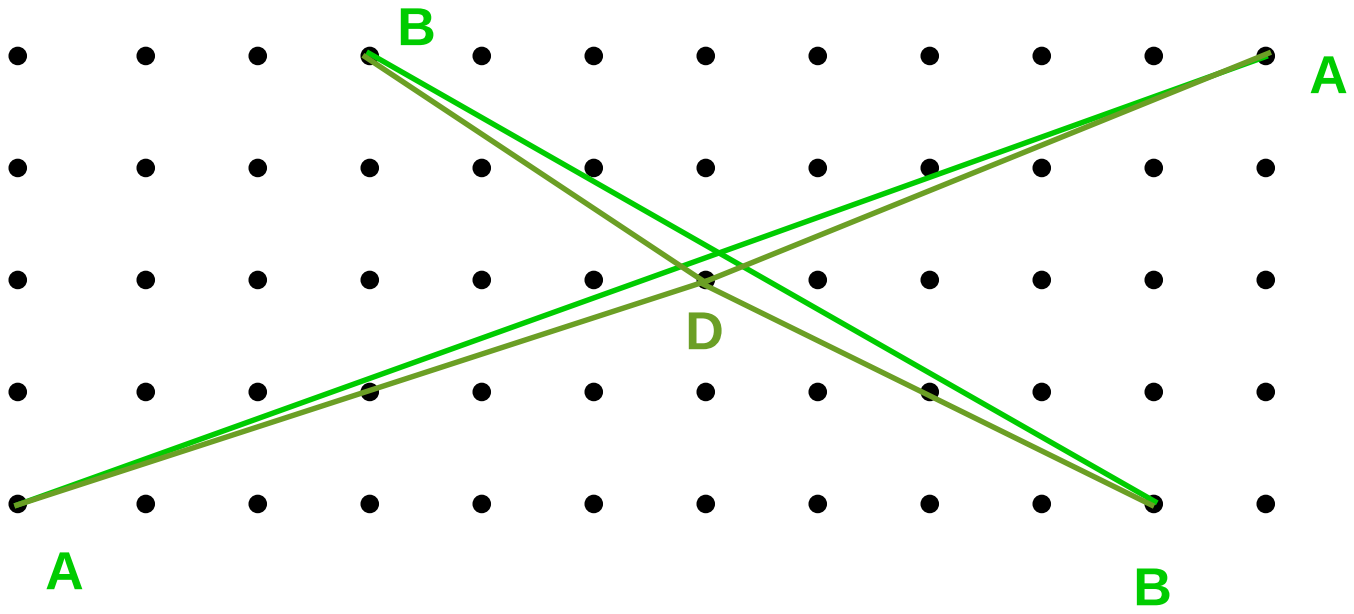
- Allow only certain shapes, from which we can build the spatial elements
  (Usually not feasible; who will break the spatial elements to match the rules?)



- Define conditions, which must be obeyed, if the condition is to be broken,
  modify the spatial elements to match the condition, keeping original values…
  (Simple to implement, nevertheless, problems with finite discrete representation is not resolved well.)

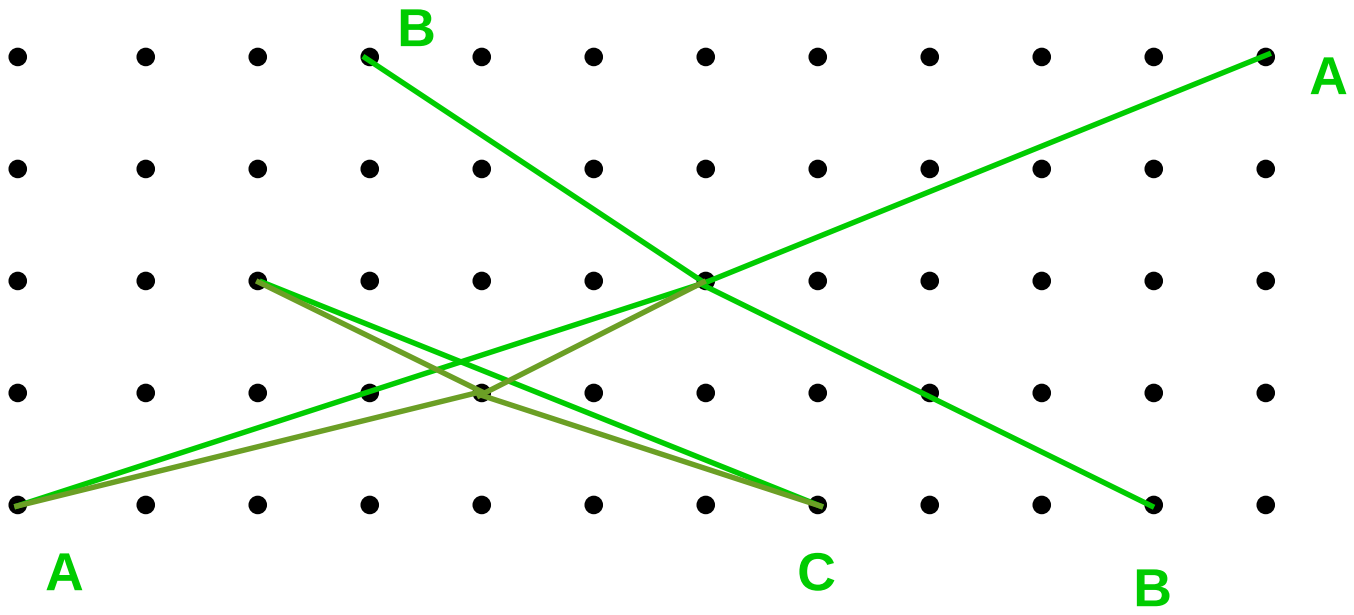
Discrete Arithmetic Operations

Demonstration – slight dislocation of line segments



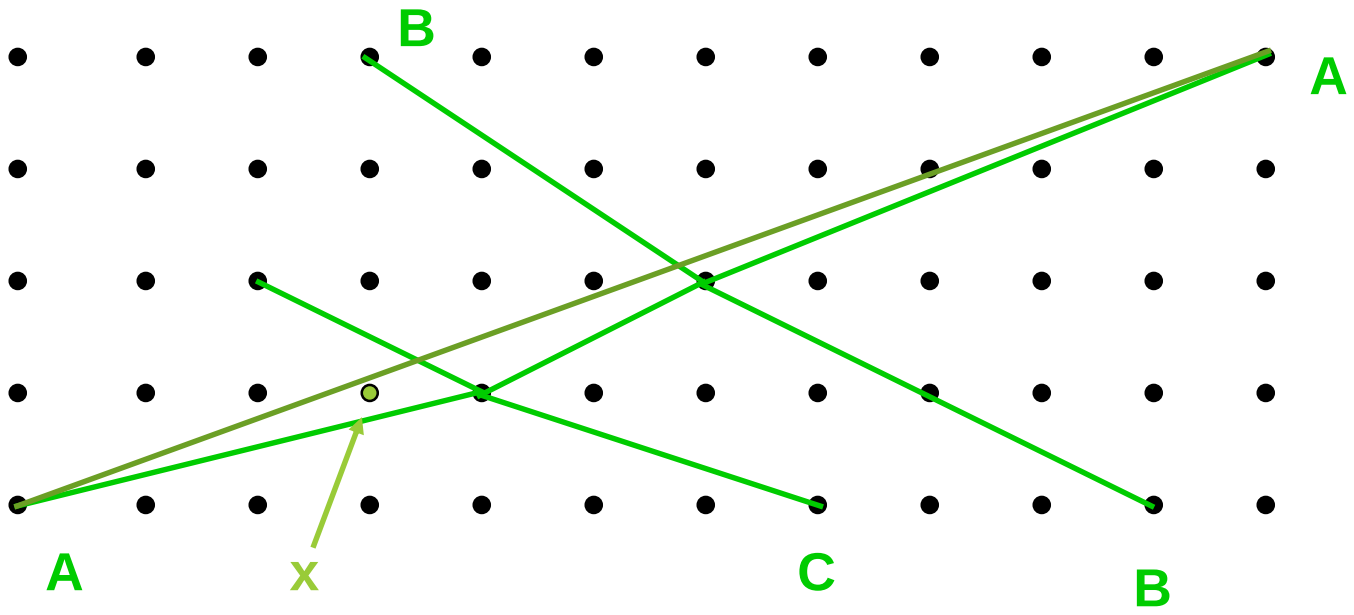
Discrete Arithmetic Operations

Another line segment added…



Discrete Arithmetic Operations

And, now, compare with the original line segment…
See point **x**



Discrete Arithmetic Operations

So, is there a solution at all?

- Yes, but usually complicated
- We should carefully design the discrete space – fine grained grid

Other possible problems

- Computation of area, volume, etc.
- Detection of centers, centers of gravity, and other points of interest

Reality Mapping

Let’s take into account just Earth surface and its mapping into 2D space – just a map of some particular region/district/etc.

What do we need to cover?

- Points of interest – just points with precise coordinates
- Power lines, pipes, etc. – lines and poly-lines are fine for representation
  Really? What about arcs?
- Buildings – rectangles, polygons are fine, or areas of the same shape
- Woods, fields, lakes, cities – polygons/areas seems to be fine
  But! Building in a city is an area inside an area (lake in the field), which is not true, areas cannot overlap

In reality, we either have to handle some problems on application level (area inside area) or we need proper spatial elements/model for representation, e.g. areas with holes…

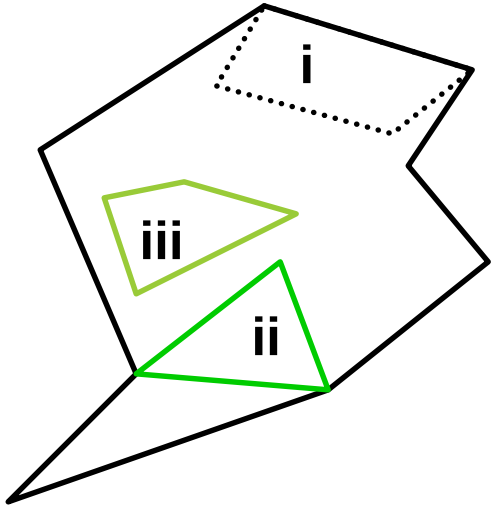
Reality Mapping – Regions

Definition:

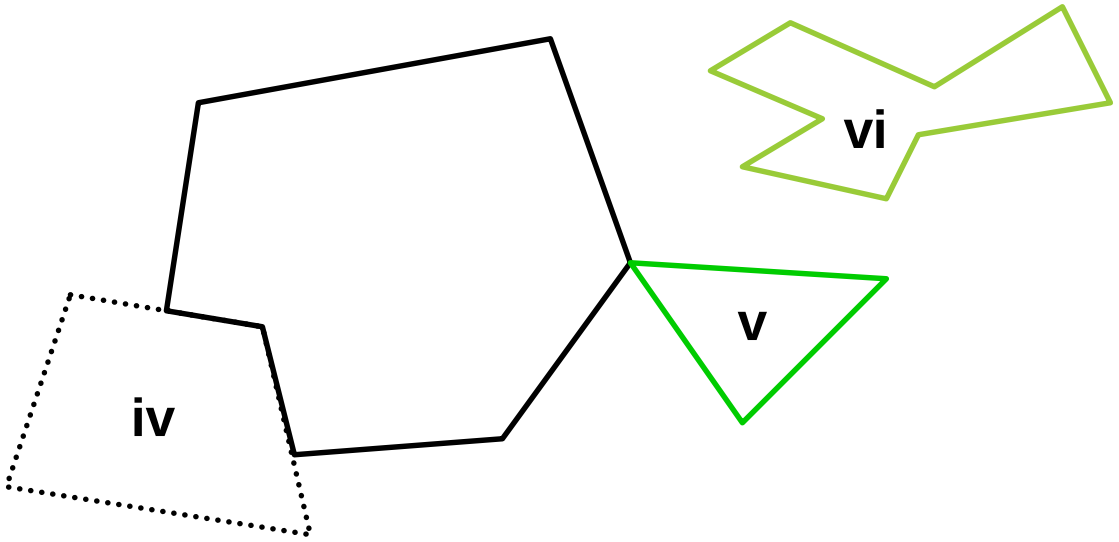
- **R-cycle** is a polygon stored in a discrete space representation. Such a polygon is composed by sequence of *n* line segments $s_1, \ldots, s_n$ and, for each line segment, it must hold that endpoint of line segment s_i is equal with starting point of line segment $s_{(i+1) \bmod n}$. Moreover, any two line segments s_i, s_j, where $i, j \in \{1, \ldots, n\}$ cannot have any intersection.

Reality Mapping – Regions

The R-cycle is
- area-nested (inside) - i, ii, iii
- edge-nested (inside) - ii, iii
- vertex/completely-nested (inside) - iii



The R-cycles are
- area-disjointed - iv, v, vi
- edge-disjointed - v, vi
- vertex/completely disjointed - vi



Reality Mapping – Regions

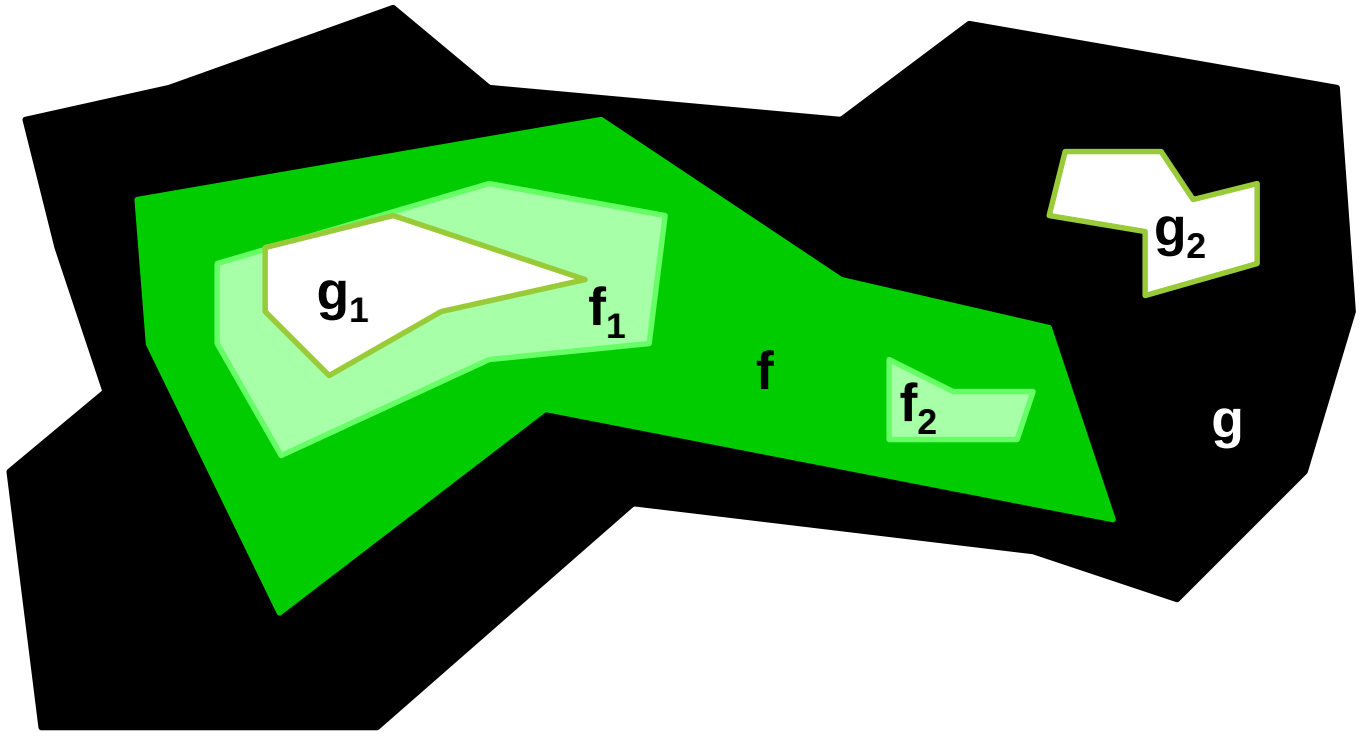
Definition:

- **R-area** *f* is couple (c, H) such that c is an R-cycle, H = $\{h_1, \dots, h_m\}$ is a set of R-cycles and it holds:
  - $\forall i \in \{1, \dots, m\}$: h_i is edge nested in c;
  - $\forall i, j \in \{1, \dots, m\}, i \neq j$: h_i and h_j are edge disjointed;
  - and it is not possible to create any other cycle from segments describing area *f*.
- Note: the last condition stands for unambiguous representation of area.

Reality Mapping - Regions

Definition:

- Let *f*=(f_0,F) and *g*=(g_0,G) be two R-areas. We say that *f* is area-nested in *g* if and only if:
  - f_0 is area-nested in g_0 and
  - $\forall g \in G$:
    - g is area-disjointed with f_0 or
    - $\exists f \in F$: g is area-nested in f



Reality Mapping - Regions

Definition:

- A value ***F*** of the type *regions* is a set of edge-disjointed R-areas.

Definition:

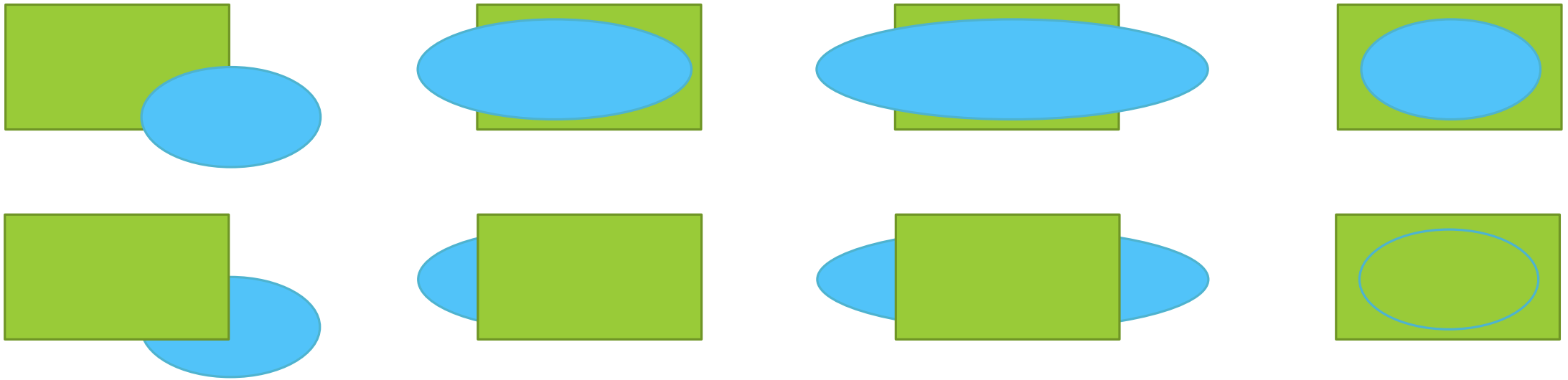
- Let ***F***, ***G*** be two values of the type regions, then ***F*** is area-nested in ***G*** if and only if:
- $\forall f \in \boldsymbol{F}: \exists\, g \in \boldsymbol{G}$: *f* is area-nested in *g*

Mutual Relation Among Spatial Elements

Some example can be seen for R-cycles above.

Nevertheless, is it sufficient? NO!

Quite many situations…



Mutual Relation Among Spatial Elements

Can we detect it somehow formally? Using some small set of computations?

- YES!

Depends on level of complexity, holes, level of intersection, other features of spatial elements…

Solution:

- For a given model, define mutually exclusive basic relations
  (*e.g. touch, inside, over, overleap, disjoint*)
- Operators that handle borders
- Combine them to handle all possibilities
  (*levels of interaction in 0D-2D:*
  256 combinations -> 52 meaningful -> covered by 5 relations and 3 border operators)

Operations Over Spatial Data

Numerical constants/characteristics (computed during storage, thus operations):

- Length, size, volume, area, diameter, perimeter, etc.

Spatial constants/characteristics (computed during storage, create new spatial value):

- Middle, center of gravity, etc.

Metrics (number as a result, cannot be computed in advance):

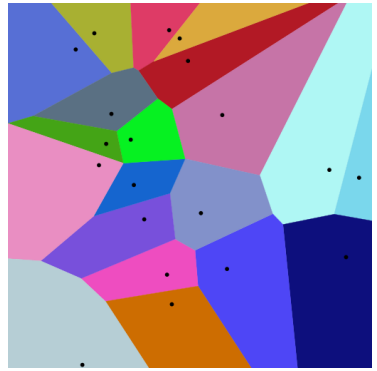
- Distance, diameter for set of elements, etc.

Predicates:

- Equal, inside, intersects, neighbor of, etc.

New objects are created/selected:

- Intersection, difference, voronoi, convex hull, etc.



Operations Over Spatial Data

Complexity of operations:

- Variable
- Some operations for spatial elements are worth to be computed during storage

Presence of operations:

- Variable ☺
- Complex operations usually missing

Possible problems:

- Operations over sets of data (coming as a result from other operations)
- Complex operations (time/space consuming)
- Operations are not closed over data types
  (e.g. result of an operation cannot be stored as a single spatial element)

Spatial Data in Relational Algebra

Not only algebra, but calculus, SQL, etc. too

- Operations in non-spatial data involve spatial operation (maps – fusion)
- Spatial join, select, projection (rare, in combination with other kinds of data)
- Operations involving viewport
  - E.g. show also rivers in the visible area
  - Coloring of the result
  - User interaction – e.g. compute center of gravity for <pick/click>

Storage

Usual storage of relational data:

- Rows of a relational table in a database are stored one behind the other in the file on the physical storage hardware

Why?

- Fixed size of each row, simple to traverse, index, etc.

Storage

What about spatial data?

- Points – fine
- Rectangles – fine
- Line – fine
- Poly-line, polygon, area, volume, etc. – problem!
- Voronoi, R-areas, Regions, etc. – BIG problem!

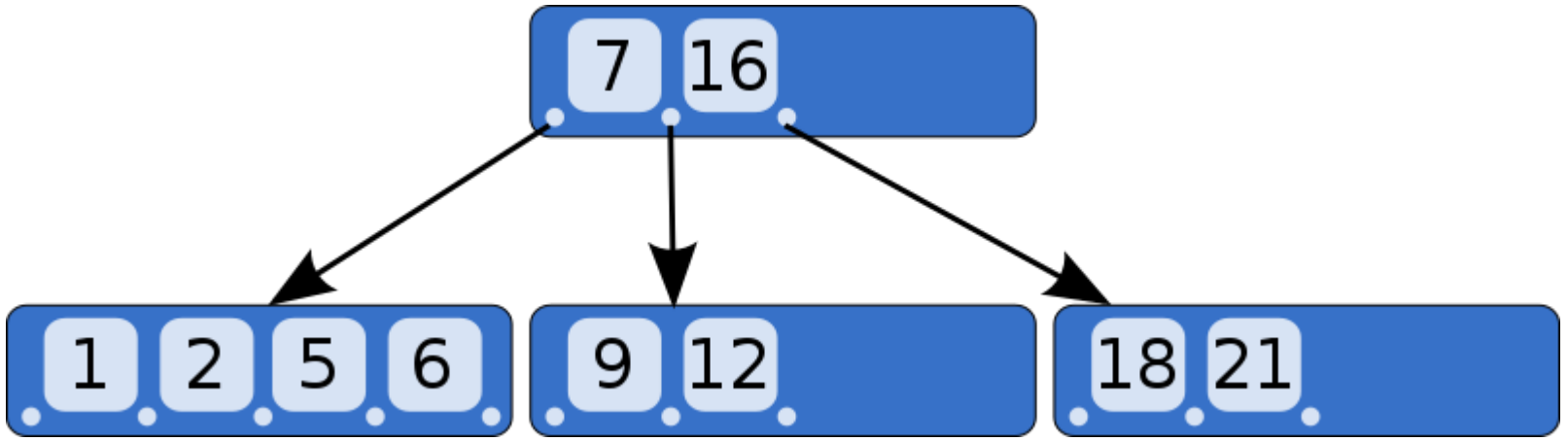
What is so problematic in storing such elements?

- Large, huge
- Variable size
- Structure of other spatial elements

Indexing

Traditional methods

- Hashing – linear, adaptive
- B-tree



Indexing – Hashing

Linear:

- Range [A,B) is divided to intervals
- (B-A)/2k, or (B-A)/2k+1, k≥0
- Match page size
- Pointer t separates already filled intervals from non-filled ones
- During insertion the interval may be split (just once), or overflow may appear
- As soon as t reaches B, it is necessary to re-divide whole file

Adaptive:

- The same as linear one, just intervals are called cells
- Overflow area is handled by directory
  - It contains index/reference of every cell
- If partition overflows all cells are split to two
  - (in the beginning cell=partition)
- After split, more cells for one partition
  - Data regions

Indexing – 1D Data

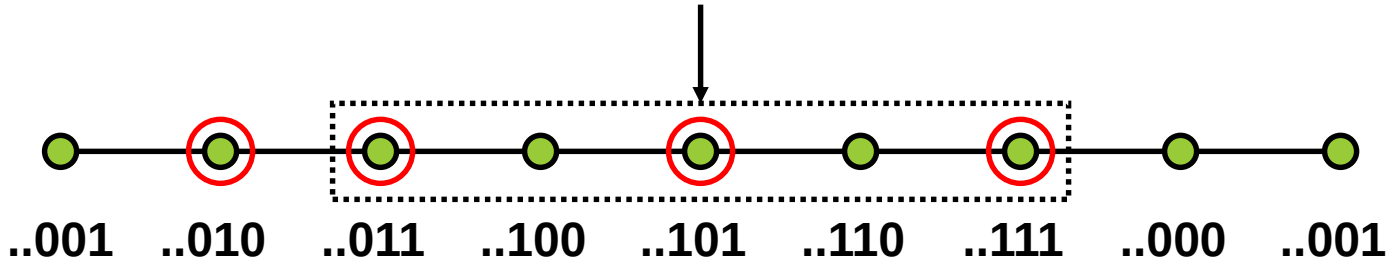
1D data are simple to map to integers

Usually, they are not distributed equally

- For equal distribution, hashing is a better approach
- Otherwise, B-tree should be used

It is easy to define

- Predecessor/successor
- Neighborhood
- Various algorithms for traversal, search, indexing

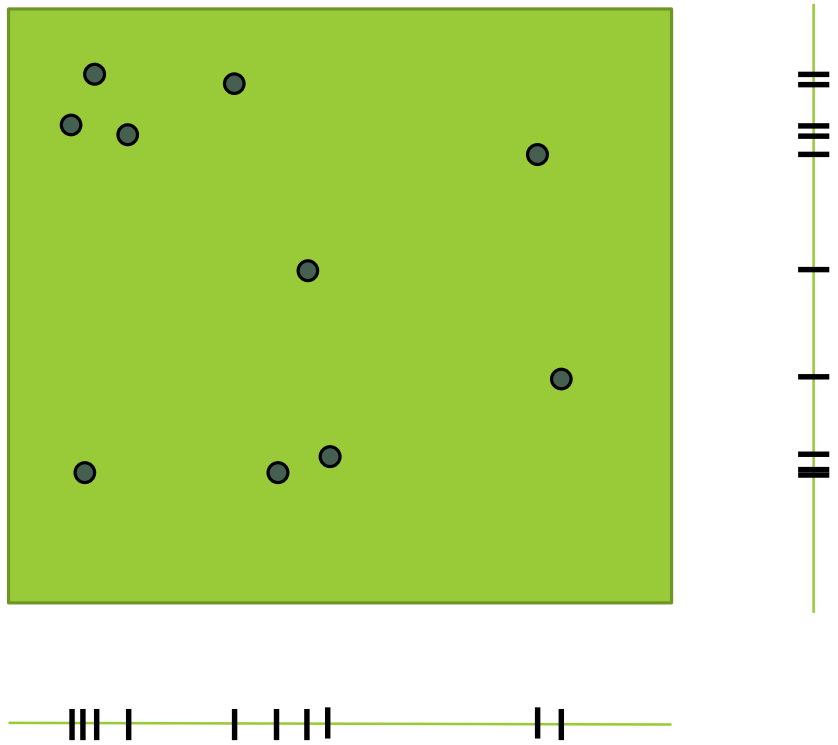


Indexing – 2D Data

Let’s take into account just points, the simplest spatial element

Problem!

- Missing ordering over 2D, 3D, … data such that keeps neighborhood
- Mapping to 1D
  - Possible!
  - Useless ☹



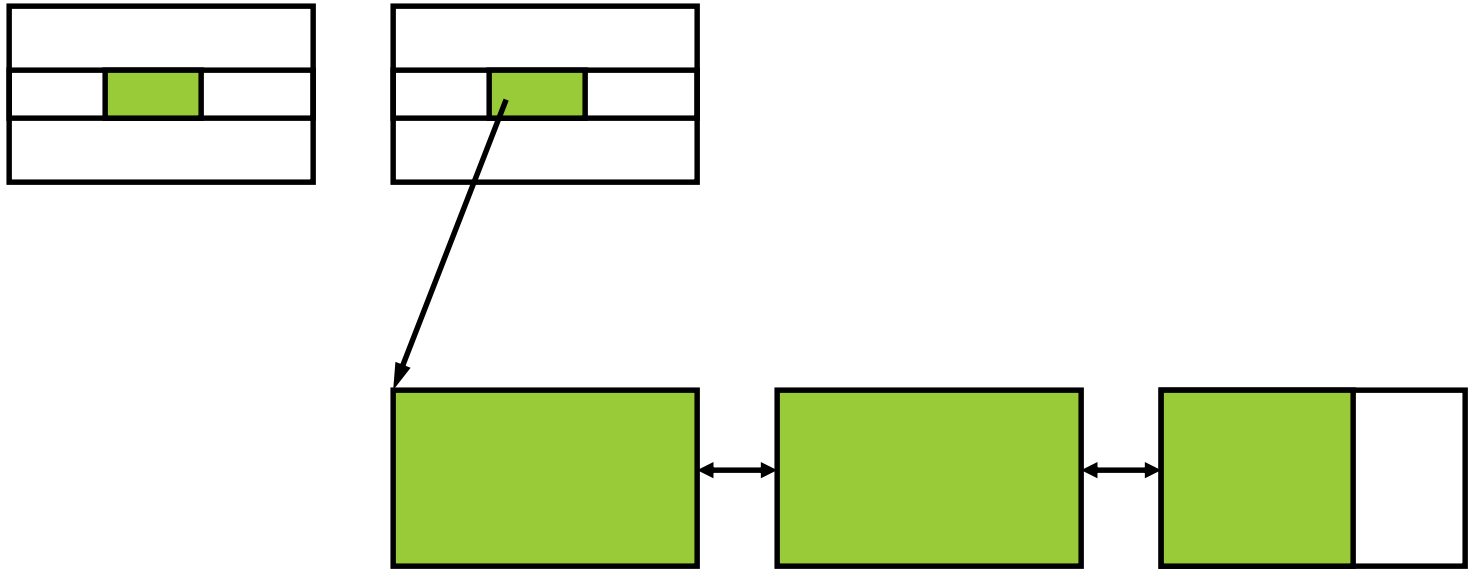
How To Store Spatial Data

Large Data

Except points, lines – possibly large data (polylines, polygons, areas, …)

Cannot store within regular data

- Skipping during search
- Storage on media may lead to long times to skip (read and skip, no continuous read)



Size Changes within Single Type

E.g. polygons representing borders of EU countries

We store fixed size information and variable information is stored elsewhere, just pointer to that part is stored (fixed size)

Data stored within others should

- be small
- contain necessary information about the object
- be useful

Approximations and Other Information

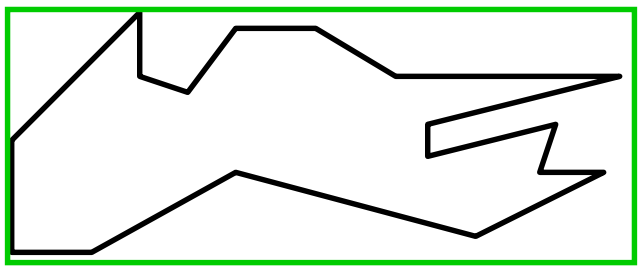
General way of processing

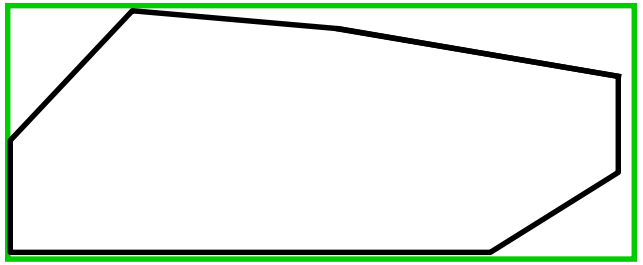
- Data ↔ Internal representation ↔ User interface (textual, graphical)

Spatial data types

- Internal representation ↔ Approximation

Pre-calculated values to be exploited (fixed values, e.g. area, length, volume, etc.)





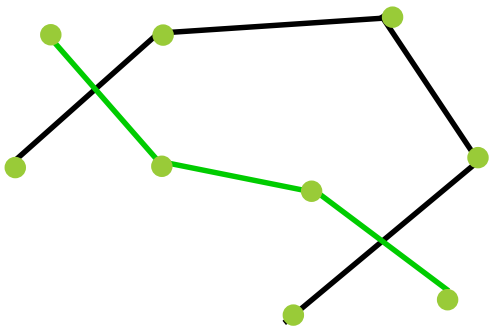
Approximations First

Operations over spatial data types

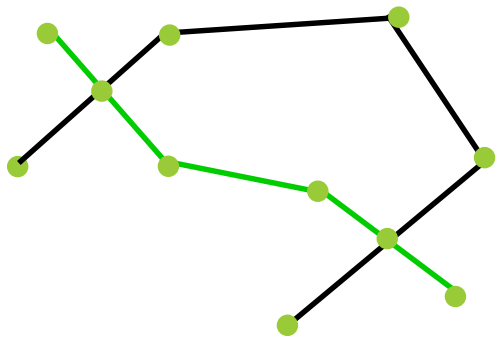
- Approximations first
- Exploitation of pre-calculated constant function results
- Algorithms from numeric geometry
- Algorithm influenced by data storage

Example

- Intersection points are never calculated during query processing, when prepared in advance, stored in both objects
- Intersection: O(n log n + k)



O(n + k)



Storage

Small fixed size information

- E.g. number of vertices, some pre-calculated values often queried, MBB
- Reference to variable data

Variable data

- Fixed part: additional pre-calculated values
- Variable, optional part: possible additional approximation, intersections, etc.
- Data part: all points/values

Indexing in Spatial Databases

Topics

General notions and approaches

Point indexing

- Trees
- Hashing / grid files

Area indexing

- Trees

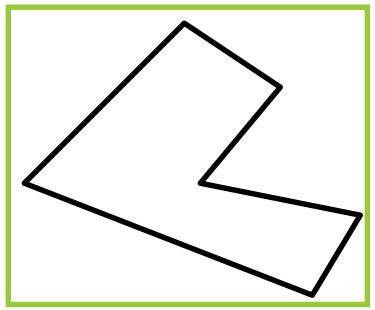
General Notions and Approaches

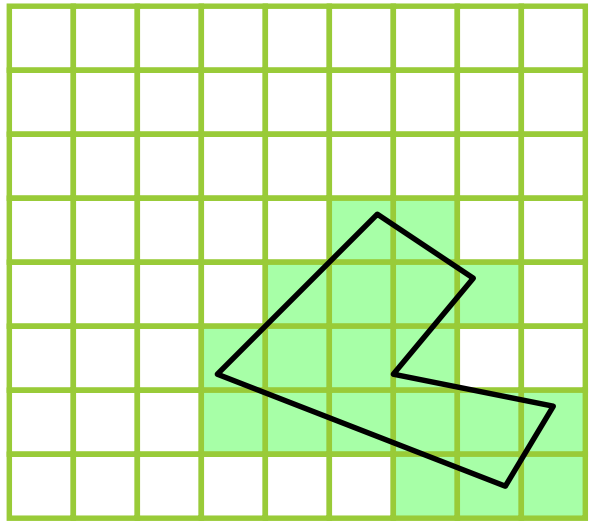
Support for

- Spatial selection, spatial join, even for other operations
- Queries: sizes, closest neighbor(s), distance, intersection, containment

Usage of suitable approximations!

- Continuous approximation
- Discrete/grid approximation

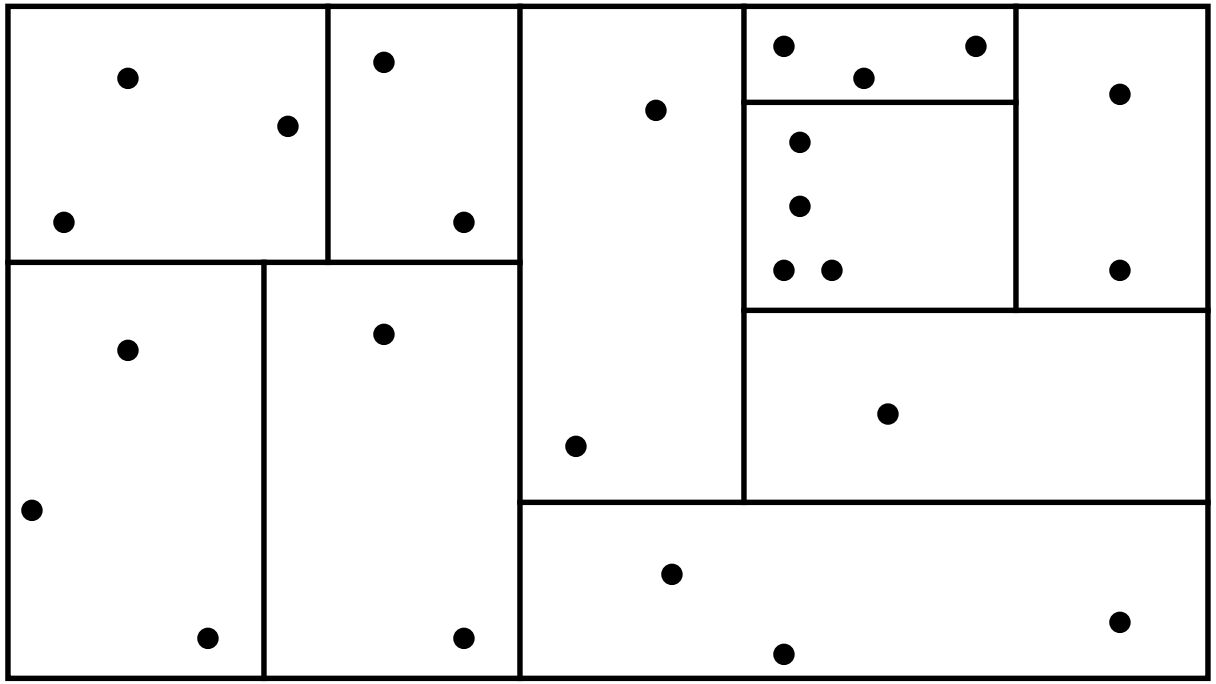




General Notions and Approaches

Space decomposition – space partitions/chunks of a fixed capacity

- To keep neighbors
- To handle irregular data distribution in space
- To make a search fast



Basic Tree Indexing Structures for Points

k-D Tree and its variants

- Adaptive k-D Tree
- Bin-Tree
- BSP Tree
- Quad Tree

Basic idea:

- Decompose space until threshold is reached
- Decomposition store in a tree structure
- If the tree is closed to balanced one then search is in log n time, insertion may be also

Problems:

- Almost all algorithms have problems when deleting data
- Tree may not be balanced

k-D Tree

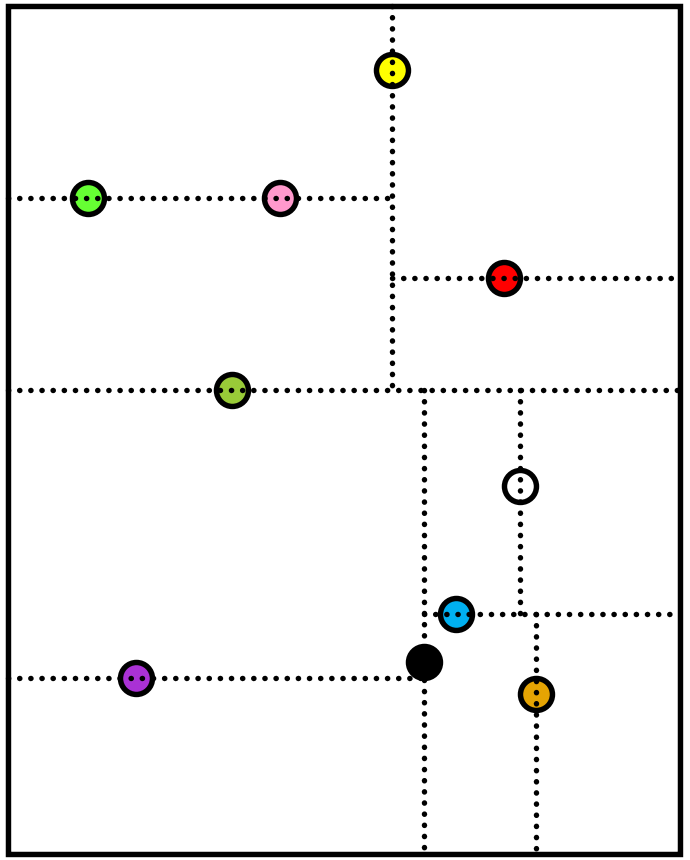
Indexed space is split
by hyperplanes
on the topmost level
(hyperplanes parallel
with axes – in 2D lines)

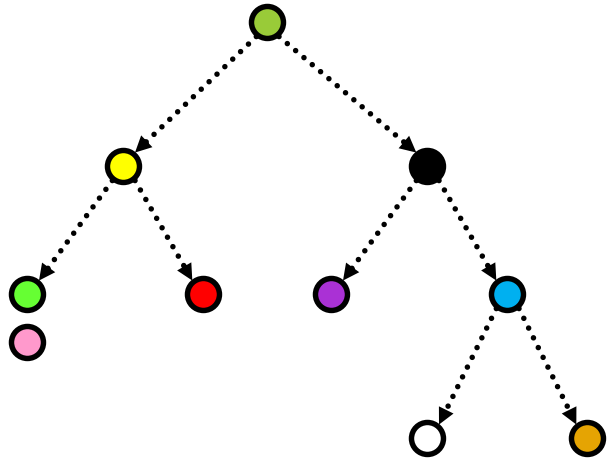
Each hyperplane must contain
at least one point

Insertion, querying – OK

Deletion – problem

Just points





k-D Tree - Creation

Optimal variant – static data, simplest implementation via recursion

Input: array/list of points
Output: tree structure or empty tree (if input is empty)

1. Set the actual dimension (e.g. axis X)
2. If the input is empty then return empty tree
3. Sort points in the actual dimension
4. Select middle point(s) and store them in a tree node
5. Move to the next dimension
   a) Go to (2) for all the points < the stored middle point as input and resulted tree add as a left branch of the node from (3)
   b) Go to (2) for all the points > the stored middle point as input and resulted tree add as a right branch of the node from (3)
6. Return the tree node

k-D Tree – Search

Simplest implementation via cycle/loop

Input: k-D Tree structure – tree (root node), point to be searched for
Output: True – point is in the structure, False – no such a point is in the structure

1. Set the actual dimension (e.g. axis X)
2. If the tree is empty then return False (no point in the tree)
3. If point(s) stored in the root of the tree have the same actual dimension as the input point
   then compare stored point(s) and respective parameter on all dimensions,
   if some match then return True else False
4. If point(s) stored in the root of the tree have value of the actual dimension greater than the same actual dimension of the input point
   then set the next dimension, set the root as the left subtree of the root, go to (2)
5. Set the next dimension, set the root as the right subtree of the root, go to (2)

k-D Tree – Insert – 1st part

Simplest implementation is via recursion

Input: k-D Tree structure (root node), point to be inserted
Output: True – point newly added/False – point already in the structure +
modified tree structure (even new one!)

1. Set the actual dimension (e.g. axis X)
2. If the tree is empty
   then create new node, store point in it, subtrees set as empty ones, return True + the node
3. If point(s) stored in the root of the tree have the same actual dimension as the input point
   then compare stored point(s) and respective parameter on all dimensions,
   if some match then return False + the root else add point to root, return True + the root
4. If point(s) stored in the root of the tree have value of the actual dimension greater than the same actual dimension of the input point then select left subtree else select right subtree
5. Set the next dimension …

k-D Tree – Insert – 2nd part

6. Case of recursive call from (2) on selected subtree and input point (dimension changed):
   - True + tree: change appropriate branch (assign new value), return True + root
   - False + tree: return False + root

Version with no recursion must handle empty tree extra, plus add “history” parameter on previous parent node in the tree, otherwise the same.

k-D Tree – Delete

HOMEWORK!

Adaptive k-D Tree

Key change:

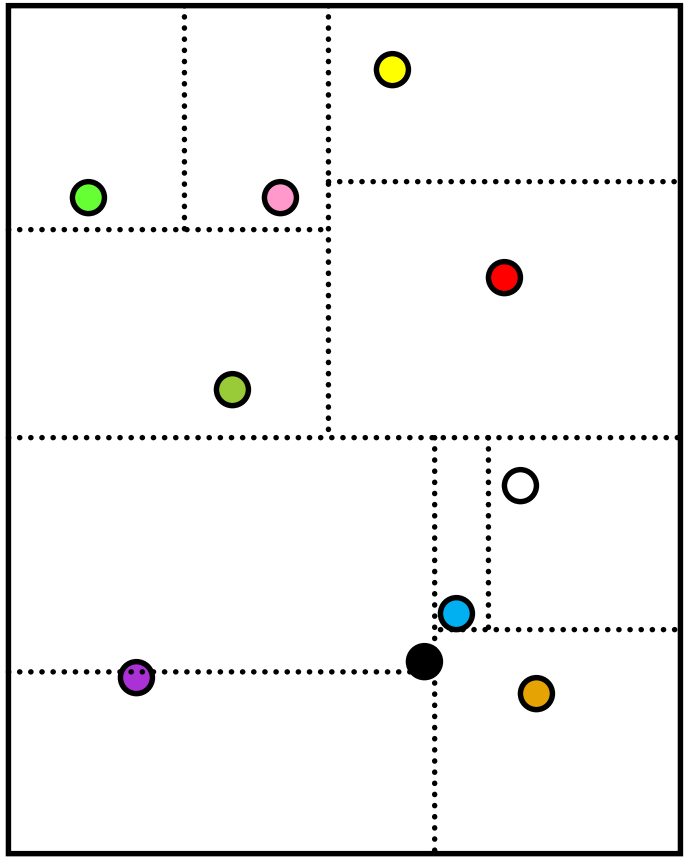
- Eliminates disadvantage given by order of data insertion
- Data are scattered in the tree (not stored in the inner nodes)

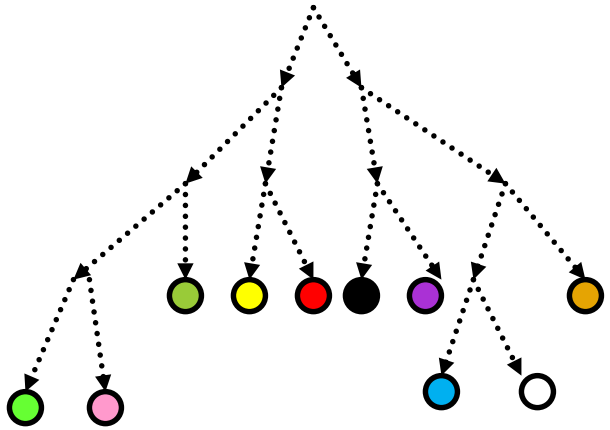
Hyperplanes split the space such a way, so that the same number of points is on each side equal

Even if hyperplanes are parallel with axes they do not carry points – data are moved to leaves (just a single point in the resulting subspace ?)

Suitable for static data

Adaptive k-D Tree – Demonstration





Adaptive k-D Tree – Creation I

Optimal variant – static data, simplest implementation via recursion

Input: array/list of points
Output: tree structure or empty tree (if input is empty)

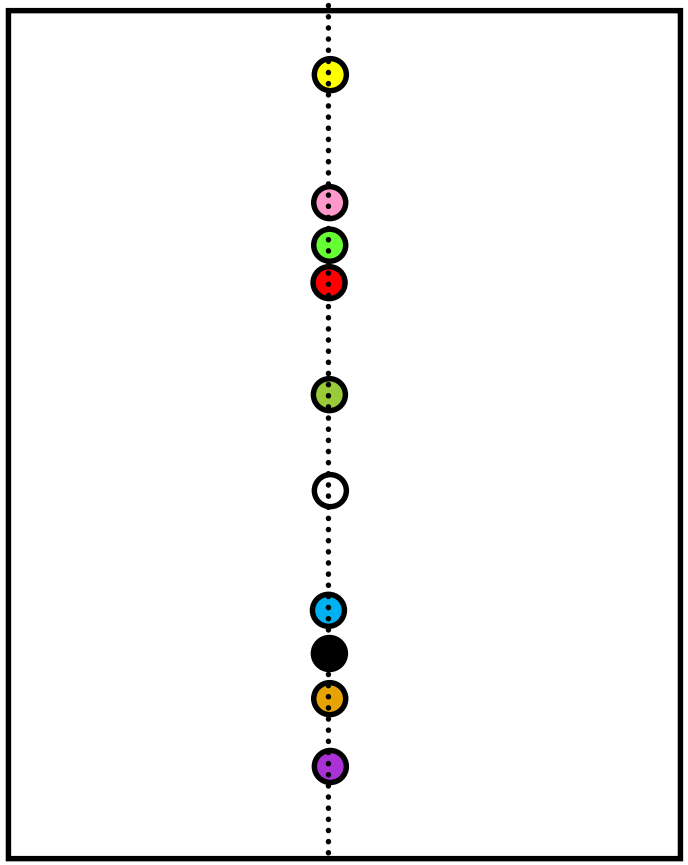
1. Set the actual dimension (e.g. axis X)
2. If the input is empty then return empty tree
3. If the input contains **one (1)** point create a leaf node, store it in it, return it as a result
4. Walk through points and set the minimum and maximum value in the first dimension
5. Set mid-value between minimum and maximum, store in the new node
6. Move to the next dimension
   a) Go to (3) for all the points ≤ the mid-value in the given dimension and resulted tree add as a **left** branch of the node from (5)
   b) Go to (3) for all the points > the mid-value in the given dimension and resulted tree add as a **right** branch of the node from (5)
7. Return the tree node

Adaptive k-D Tree – Creation Problem

Starting with X axis

No split line can be used

Starting with Y axis just postpones the problem



Adaptive k-D Tree – Creation II – 1st part

Optimal variant – static data, simplest implementation via recursion

Input: array/list of points
Output: tree structure or empty tree (if input is empty)

1. Set the actual dimension (e.g. axis X)
2. If the input is empty then return empty tree
3. If the input contains **one (1)** point create a leaf node, store it in it, return it as a result
4. Walk through points and set the minimum and maximum value in the first dimension
5. If the maximum and minimum are the same
   then create a leaf node, store all points in it, return it as a result

…

Adaptive k-D Tree – Creation II – 2nd part

…

6. Set mid-value between minimum and maximum, store in the new node
7. Move to the next dimension
   a) Go to (3) for all the points ≤ the mid-value in the given dimension and resulted tree add as a **left** branch of the node from (6)
   b) Go to (3) for all the points > the mid-value in the given dimension and resulted tree add as a **right** branch of the node from (6)
8. Return the tree node

Adaptive k-D Tree – Search

Simplest implementation via cycle/loop

Input: Adapt. k-D Tree structure – tree (root node), point to be searched for
Output: True – point is in the structure, False – no such a point is in the structure

1. Set the actual dimension (e.g. axis X)
2. If the tree is empty then return False (no point in the tree)
3. If the tree node is a leaf then compare stored point(s) and respective parameter on all dimensions,
   if values match (at least on one point) then return True else False
4. If the value stored in the node is greater than or equal to value of actual dimension in the respective parameter then set the next dimension, set the node as the left subtree of the root, go to (3)
5. Set the next dimension, set the node as the right subtree of the root, go to (3)

Adaptive k-D Tree – Insert – 1st part

Simplest implementation is via recursion

Input: Adapt. k-D Tree structure (root node), point to be inserted
Output: True – point newly added/False – point already in the structure +
modified tree structure (even new one!)

1. Set the actual dimension (e.g. axis X)
2. If the tree is empty
   then create new leaf node, store point in it, return True + the node
3. If the root node is leaf then compare stored point(s) and respective parameter on all dimensions,
   if values match (at least on one point) then return False + the root
   else if values of the first dimension of both stored point(s) and respective parameter equals
   then add parameter point to the list of stored points, return False + root
   else set mid-value on the actual dimension from stored point(s) and parameter
   create **new node**, its value set to mid-value
   actual root (leaf) store to appropriate subtree of the **new node**
   create **new leaf node**, store the parameter point into it
   new leaf node add to appropriate subtree of the **new node**, return True + **new node**

Adaptive k-D Tree – Insert – 2nd part

4. If the value stored in the root of the tree have value of the first dimension greater than or equal to the same first dimension of the input point then select left subtree else select right subtree
5. Set the next dimension
6. Case of recursive call from (3) on selected subtree and input point (dimension changed):
   - True + tree: change appropriate branch (assign new value), return True + root
   - False + tree: return False + root

Version with no recursion must handle empty tree extra, plus add “history” parameter on previous parent node in the tree, otherwise the same or very much similar.

Homework

Deletion

Optimal split to make tree balanced

Basic Grid Indexing Structures for Points

Adaptive hashing
- **Grid File**
- EXCELL
- Two-Level Grid File
- BANG File
- Twin Grid File

Linear hashing too (MOHLPE, Quantile hashing, Z-hashing, …)

Hybrid algorithms
- Buddy Tree – algorithm applicable elsewhere, e.g. Grid File

Basic idea:
- Cover space with hash-able grid
- Grid map to storage
- Index-sequential search

Problems:
- Point clusters

Grid File

The space is covered by an n-dimensional grid

Not necessarily regular

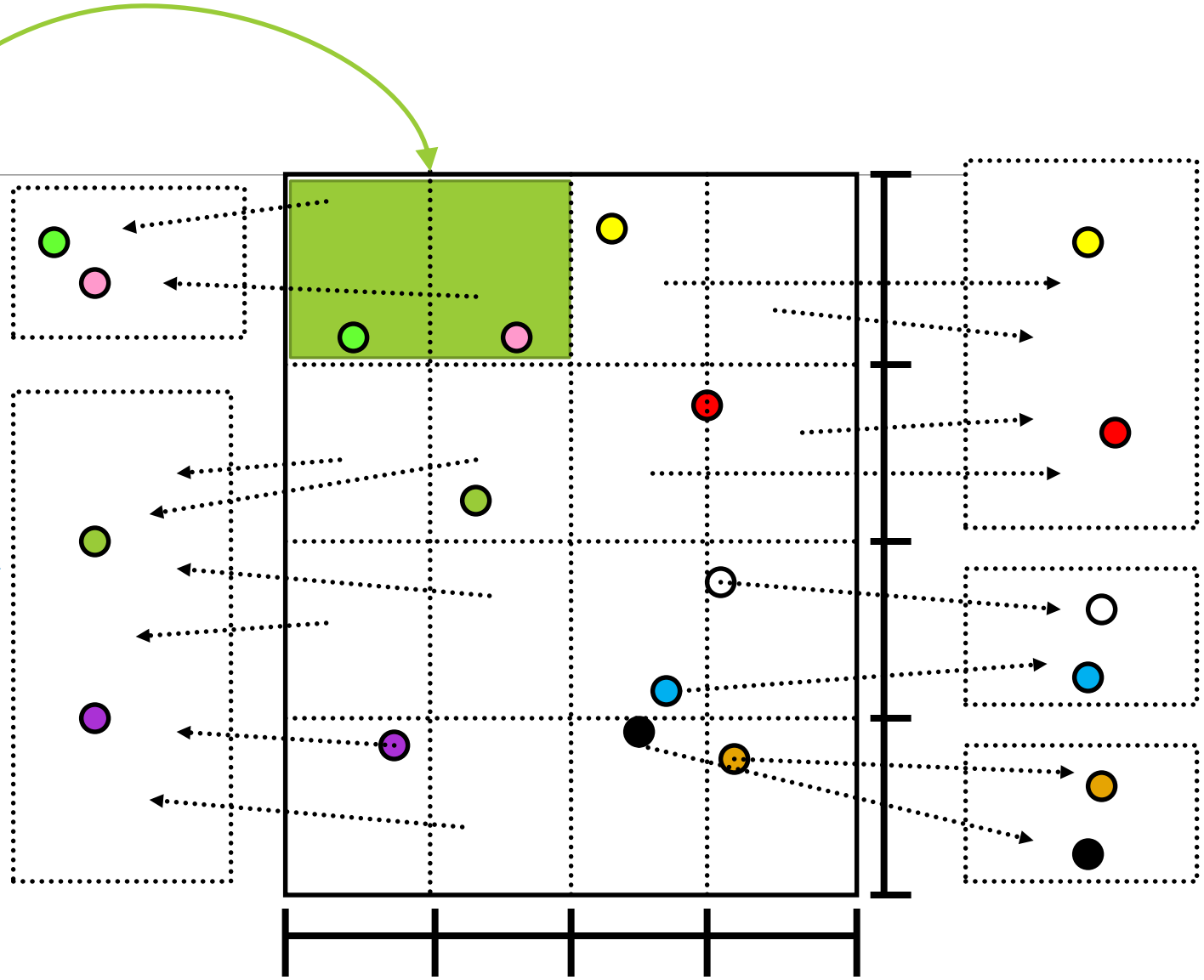
Resulting cells are varying

Directory assigns every cell (even more) to particular **bucket (data unit)**

Directory is large – on a disk

Grid on disk (at most two reads from a disk)

Space utilization about 69% until slowdown



Grid File – Creation

Optimal variant – grid size fine, simplest implementation for 2D

Input: array/list of points, grid sizes M, N, space size <Xmin,Xmax>, <Ymin,Ymax>
Output: Grid file structure

1. Create grid from M, N, <Xmin,Xmax>, <Ymin,Ymax>
   (array, tree, list, …)
2. Let B = new bucket (data unit)
3. Traverse grid in a suitable way
   a) As long as some threshold is not reached target grid cells to the B, add their points to the B
   b) Let B = new bucket (data unit)
   c) Continue traversal until all grid cells are traversed
4. Return grid, buckets

Grid File – Search

Simplest implementation via cycle/loop

Input: Grid File structure – grid & buckets, point to be searched for
Output: True – point is in the structure, False – no such a point is in the structure

1. Determine grid cell according to X and Y coordinates and grid structure
2. Load bucket via pointer in the cell (from step 1)
3. Go through list of points in the bucket
   if the point is found then return True else False

Grid File – Insert

Simplest implementation, no split strategy defined

Input: Grid File structure – grid & buckets, point to be inserted for
Output: True – point inserted, structure modified, False – point there, Fail – cannot split

1. Determine grid cell according to X and Y coordinates and grid structure
2. Load bucket via pointer in the cell (from step 1)
3. Go through list of points in the bucket
   if the point is found then return False
4. If the bucket still contains empty space, insert point there, return True
5. If the cells can be split (more pointers, points scattered OR no split so far) then
   a) Create new bucket
   b) Split cell pointers, split points
   c) Insert point to a proper bucket
   d) return True
6. Fail

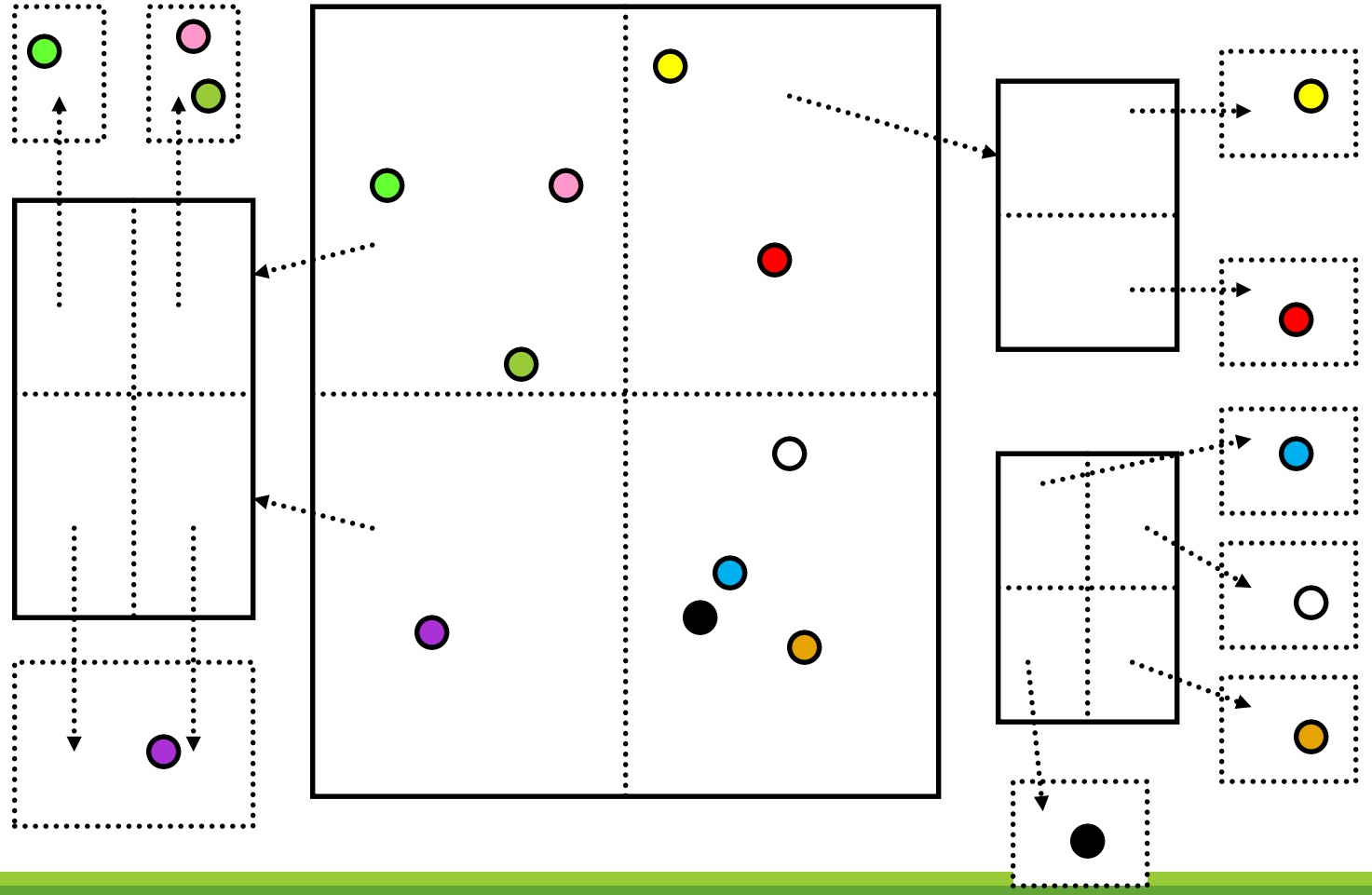
Grid File – Deletion

Homework – simple version first, think about merging buckets/cells…

Two-Level Grid File

- Grid on the second level is used for directory management
  - Root directory
  - Sub-directory
- Changes often local
  - Nevertheless, issues of overflow/underflow not resolved satisfactory

Two-Level Grid File Demonstration



Homework

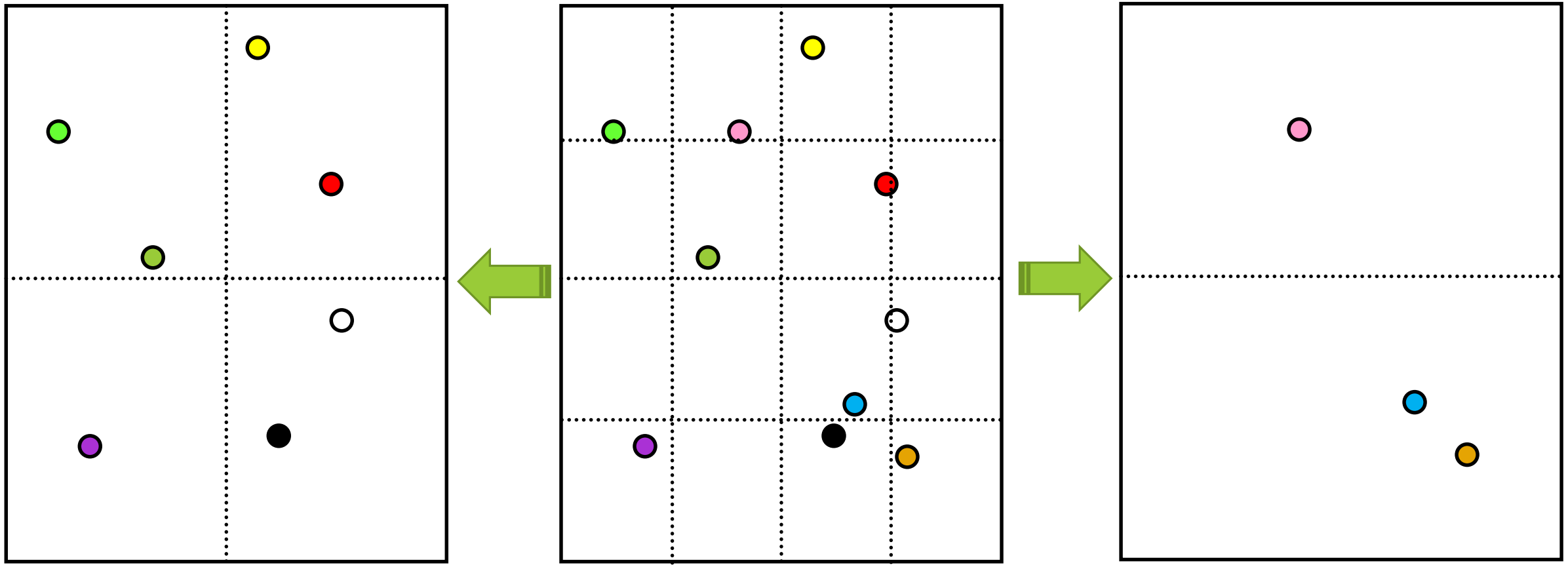
Modify algorithms for Grid File to make them work as Two-Level Grid File – for insertion/deletion do split/merge just on the sub-directory level.

Twin Grid File

Doubled Grid File structure

- Space exploitation improved
- Relation not hierarchical
- Data split equally
- Space use up to 90% without “slowdown”
- One of Files is “primary”, the other is “overflow”, they may swap

Twin Grid File Demonstration



Twin Grid File – Search

Input and output the same as for Grid File, just instead of one structure, two are passed.

Algorithm:

Run the search in parallel in both structures, if any returns True, return True, return False otherwise.

Homework

Define algorithms for creation, deletion and insertion for Twin Grid File – exploit Grid File ones.

Multi-Dimensional Objects Indexing

2D – area indexing

Why area?

Where it can help?

- Which operations? Which not?

Can we store points too?

- What is affected?
- How is it affected?
- Is it an advantage?

Key Approaches

Transformation – mapping of objects from one space to another (no space traveling, no magic, no de-re-composition ☺)

Overlapping – delimitation of a space covered by indexing structure is not precise, such structures may overlap

Clipping – object duplication in several indexing sub-structures due to exact delimitation of the indexed (sub)space

Space filling curves

Transformation

Objects in n-D space are points in the k*n-D space

- E.g. rectangle in 2D is a point in 4D
- 2 points (X1,Y1) , (X2,Y2) define a single point (X1,Y1,X2,Y2)

Optimism $\rightarrow$ disillusion

- Some queries cannot be defined
- Mapping is difficult (impossible) for some objects especially as results
- Result interpretation issues

Overlapping

Many representatives

Key one – nestor

- R-Tree

Derived – lots of

- R*-Tree

Idea:

- Indexing structures may overlap

R-Tree

Antonin Guttman

University of Carolina, Berkeley, 1984

R-TREES: A DYNAMIC INDEX STRUCTURE FOR SPATIAL SEARCHING

http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf

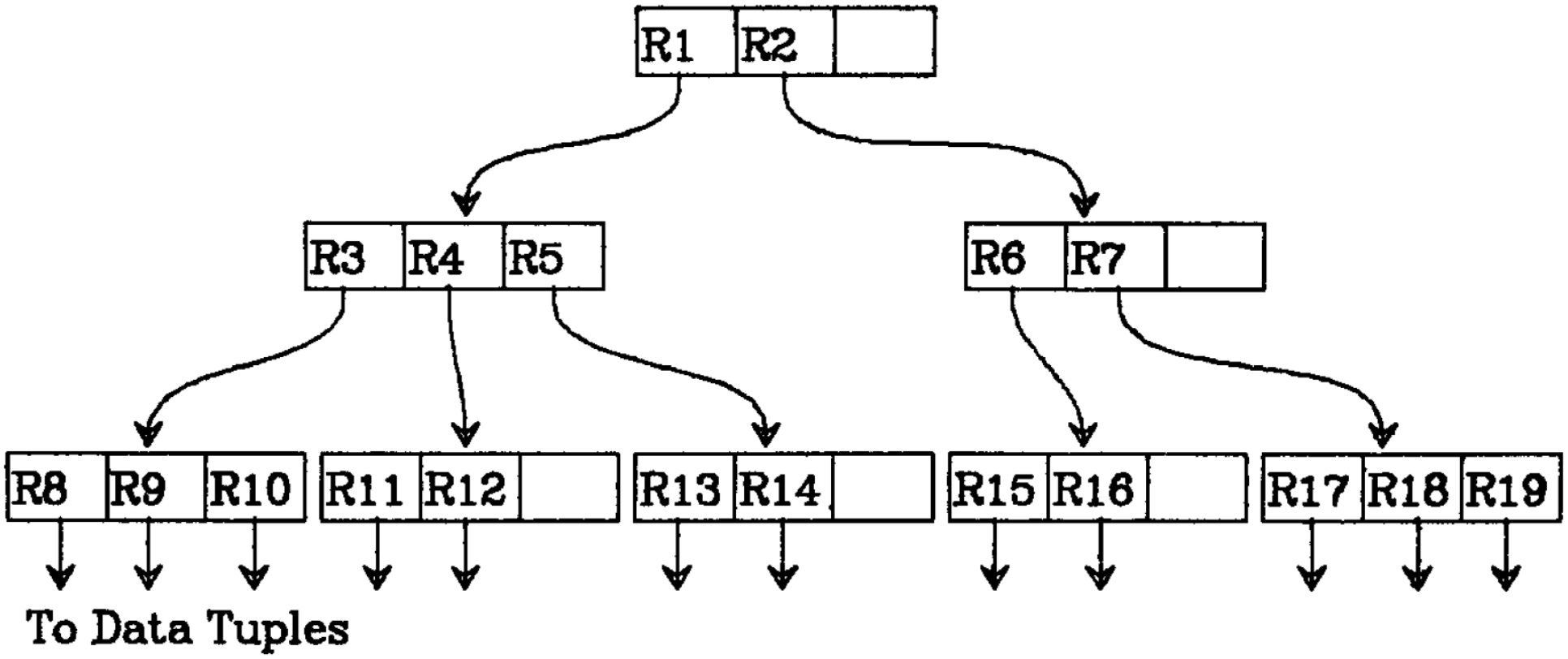
B-Tree inspiration – balanced, fixed number of descendants

Differences: splitting algorithm

Features: several search paths

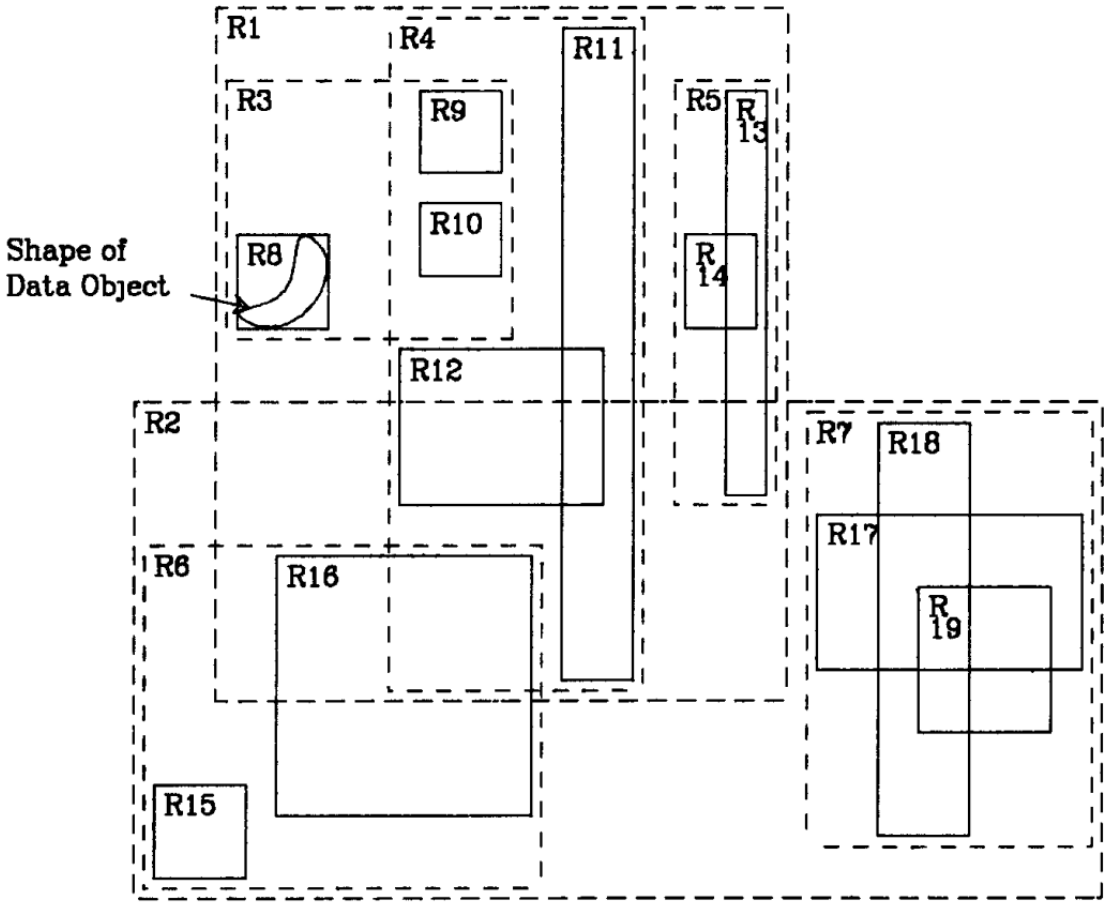
R-Tree Demonstration

From Guttman, 1984



R-Tree Demonstration

From Guttman, 1984



R-Tree – Search

Input: root-node T of the R-Tree, search rectangle S
Output: list of all index records overlapping with S

1. If T is not a leaf, check each entry E to determine whether EI (rectangle part of the index entry) overlaps S. For all overlapping entries, invoke **Search** on the tree whose root node is pointed to by Ep (tuple-identifier or child pointer).
2. If T is a leaf, check all entries E to determine whether EI overlaps S. If so then E is a qualifying record.

Guttman, 84

Homework: detect that two rectangles overlap – ordered 2D coordinates, ll & ur corners, it holds that ll-x/y < ur-x/y

R-Tree – Insert

Input: R-Tree structure, entry E to be inserted
Output: modified R-Tree structure

1. Invoke **ChooseLeaf** to select a leaf node L in which to place E.
2. If L has room for another entry, install E. Otherwise, invoke **SplitNode** to obtain L and LL containing E and all the old entries of L.
3. Invoke **AdjustTree** on L, also passing LL, if a split was performed.
4. If node split propagation caused the root to split, create a new root whose children are the two resulting nodes.

Guttman, 84

Minimum m <= M/2, where M is the maximal number of entries in a node.

R-Tree – Insert: ChooseLeaf

Input: R-Tree structure, entry E to be inserted
Output: Leaf node, where to place E

1. Set N to be the root node of the R-Tree structure.
2. If N is a leaf then return N.
3. If N is not a leaf then let F be an entry in N whose rectangle FI needs least enlargement to include EI. Resolve ties by choosing the entry with the rectangle of smallest area.
4. Set N to be the child pointed to by Fp and repeat from point (2).

R-Tree – Insert: AdjustTree

Ascend from a leaf node L to the root, adjusting covering rectangles and propagating node splits as necessary.

Input: R-Tree structure, leaf node L, if the L was split then also LL
Output: modified R-Tree structure, or two nodes, if the root node was hit

1. Set N = L. If L was split previously, set NN to be resulting second node.
2. If N is the root then stop.
3. Let P be the parent node of N, and let E_N be the N's entry in P. Adjust E_NI so that it tightly encloses all entry rectangles in N.
4. If N has a partner NN resulting from an earlier split, create a new entry E_{NN} with E_{NN}p pointing to NN and E_{NN}I enclosing all rectangles in NN. Add E_{NN} to P if there is room. Otherwise, invoke **SplitNode** to produce P and PP containing E_{NN} and all P's old entries.
5. Set N = P and set NN = PP if a split occurred, repeat from 2.

R-Tree – Deletion

Input: R-Tree structure, entry E to be removed
Output: modified R-Tree structure

1. Invoke **FindLeaf** to locate the leaf node L containing E. Stop if the record was not found!
2. Remove E from L.
3. Invoke **CondenseTree**, passing L.
4. If the root node has only one child after the tree has been adjusted then make the child the new root.

Guttman, 84

R-Tree – Deletion: FindLeaf

Input: R-Tree structure, with root node T, index entry E
Output: Appropriate leaf or fail (not found)

- If T is not a leaf, check each entry F in T to determine if FI overlaps EI. For each such entry invoke **FindLeaf** on the tree whose root is pointed to be Fp until E is found or all entries have been checked.
- If T is a leaf, check each entry to see if it matches E. I E is found return T.

R-Tree – Deletion: CondenseTree

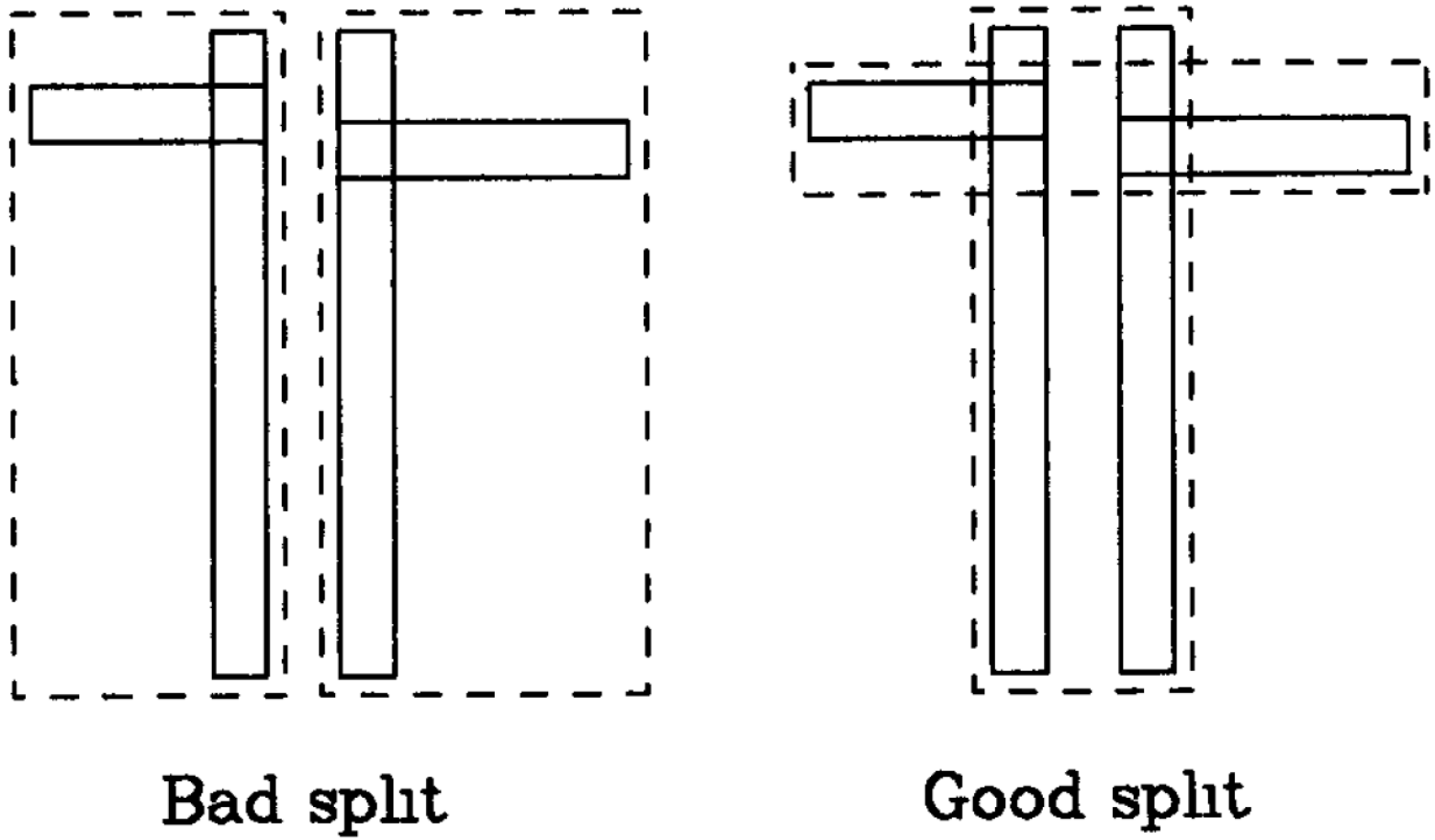
Input: leaf node L (where entry was deleted) of R-Tree structure
Output: modified R-Tree structure

1. Set N=L. Set Q, the set of eliminated nodes, to be empty.
2. If N is the root then go to point (6). Otherwise, let P be the parent of N, and let E_N be N's entry in P.
3. If N has fewer than *m* entries then delete E_N from P and add N to set Q.
4. If N has not been eliminated then adjust E_NI to tightly contain all entries in N.
5. Set N=P and repeat from point (2).
6. Re-insert all entries of nodes in set Q. Entries from eliminated leaf nodes are re-inserted in tree leaves as described in algorithm **Insert**, but entries from higher level nodes must be placed higher in the tree, so that leaves of their dependent sub-trees will be on the same level as leaves of the main tree.

R-Tree – Node Splitting

Not a trivial task
Many approaches
Key feature

Guttman, 84



R-Tree – Node Splitting

Task: M+1 nodes to be split

Best way:

- Find all possible combinations
- Detect the best splitting
- 2^{M-1} combinations (approximately)
- Ideal M for 1kiB page is 50, for 4kiB page is 100-200 (based on architecture)
  - 4 numbers (rectangle) of 4 bytes, pointer of 4 bytes = 20
  - 4 numbers (rectangle) of 8 bytes, pointer of 8 bytes = 40
- So number of combinations is extremely large => not possible to implement this way

R-Tree – Quadratic Split [Guttman, 84]

Input: set of M+1 entries
Output: two disjoint groups of the entries

1. Apply algorithm **PickSeeds** to choose two entries to be the first elements of the groups. Assign each to a group.
2. If all entries have been assigned then stop. If one group has so few entries that all the rest must be assigned to it in order for it to have minimum number *m*, assign them and stop.
3. Invoke algorithm **PickNext** to choose the next entry to assign. Add it to the group whose covering rectangle 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, then to either. Repeat from point (2).

R-Tree – Quadratic Split: PickSeeds

Input: set of entries where seeds are to be found
Output: pair of entries composing seeds

1. For each pair of entries E_1 and E_2, compose a rectangle J including E_1I and E_2I. Calculate d = area(J) – area(E_1I) – area(E_2I).
2. Choose the pair with the largest d.

Homework: how to compose J?

R-Tree – Quadratic Split: PickNext

Input: Two groups of entries, set of entries not being in any of the two group
Output: Entry to be added in a next step to a group

1. For each entry E not yet in a group, calculate d_1 = the area increase required in the covering rectangle of Group 1 to include EI. Calculate d_2 similarly for Group 2.
2. Chose an entry with the maximum difference between d_1 and d_2.

R-Tree – Linear Split [Guttman, 84]

The algorithm is the same as for Quadratic Split, just instead of PickSeeds the LinearPickSeeds is used and similar replacement for PickNext is done too with LinearPickNext.

R-Tree – Linear Split: LinearPickSeeds

Input: set of entries where seeds are to be found
Output: pair of entries composing seeds

1. Along each dimension, find the entry whose rectangle has the highest low side, and the one with lowest high side. Record the separation.
2. Normalize the separations by dividing by the width of the entire set along the corresponding dimension.
3. Chose the pair with the greatest normalized separation along any dimension.

Homework: implement it so that the algorithm has linear time complexity worst case.

R-Tree – Linear Split: LinearPickNext

Input: Two groups of entries, set of entries not being in any of the two group
Output: Entry to be added in a next step to a group

1. Chose any entry.

R*-Tree

Aim: Improve locality over R-Tree

- Less overlaps, better coverage (tighter)

Beckman et al. 1990

Squarish rectangles tend to have better features.

Leaf nodes: a record whose entry covering rectangle if enlarged has the least overlap with other covering rectangles; a tie is resolved by choosing the entry whose rectangle needs the least area enlargement.

Inner nodes: an entry whose covering rectangle needs the least area enlargement is chosen to include new record; a tie is resolved by choosing the entry with the smallest resultant area.

Clipping

R^+-Tree

- Derived from R-Tree, of course ☺

Data units (MBB, rectangles) do not overlap.

They are enlarged during insertion, but as they cannot overlap a deadlock may appear.

R^+-Tree

Timo Sellis, Nick Roussopoulos, Christos Faloutsos

University of Maryland, 1987

THE R^+-TREE: A DYNAMIC INDEX FOR MULTI-DIMENSIONAL OBJECTS

http://www.vldb.org/conf/1987/P507.PDF

R-tree and K-D-B-tree compromise

Summary on Indexing

One operation missing – which one? How to implement?

Can points be indexed by R-Tree or any of its descendant?

- If yes, how the algorithms must be changed?

Where else, except spatial DBMS, the algorithms can be used?