n-Dimensional Index Structures

Tecnologie delle Basi di Dati M

Multi-dimensional queries

- As we saw, B^+-tree is able to solve queries involving multiple attributes
- Which queries are solvable by exploiting a multi-attribute index?
- The query evaluation is efficient enough?

Types of n-dimensional queries

- $A_1 = v_1, A_2 = v_2, \ldots, A_n = v_n$ (point query)
- $l_1 \leq A_1 \leq h_1, l_2 \leq A_2 \leq h_2, \ldots, l_n \leq A_n \leq h_n$ (window query)
- $A_1 \approx v_1, A_2 \approx v_2, \ldots, A_n \approx v_n$ (nearest neighbor query)
- What if the data “value” is not a point?

Examples of use

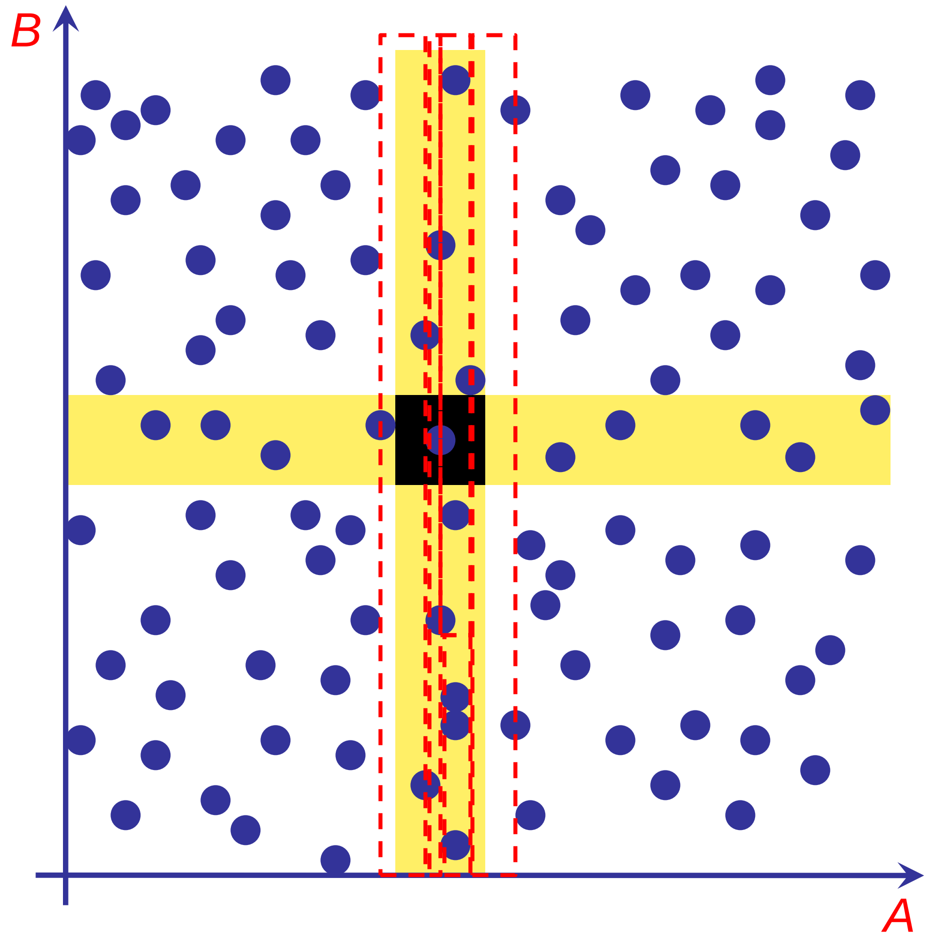
- Geographic/Spatial Information Systems
  - Coordinates of points
    - Places, cities
  - Objects with extension
    - Regions, streets, rivers
- Multimedia Databases
  - Content-Based Retrieval
    - Representing content by way of numerical characteristics (*features*)
    - Similarity of content is assessed by evaluating similarity of features
- ...

Using B^+-tree

- Suppose we have a window query on 2 attributes (A,B)
  - Every interval represents 10% of the total
  - We expect to retrieve 1% of data
- Possible solutions:
  - 1 bi-dimensional B^+-tree (A,B)
  - 2 mono-dimensional B^+-trees (A),(B)

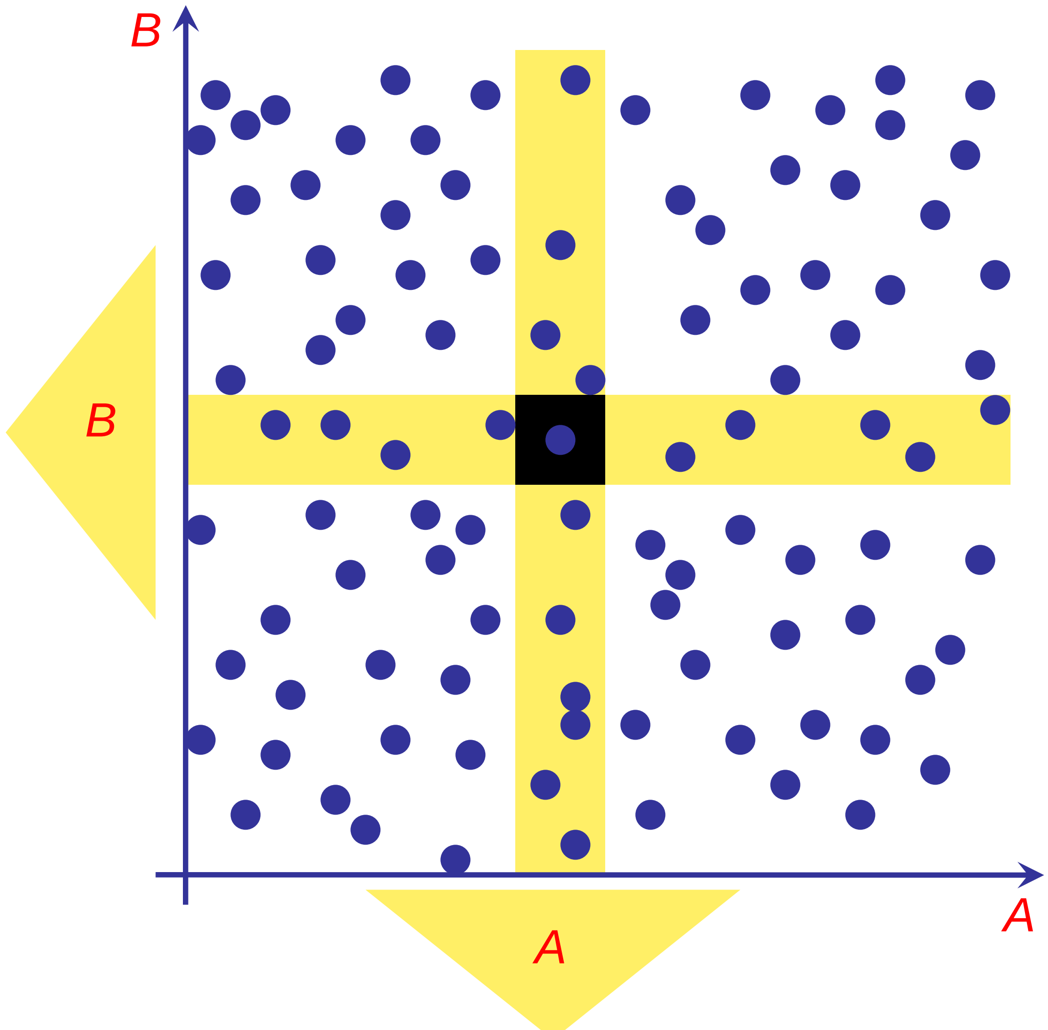
1 bi-dimensional B^+-tree (A,B)

- Leaf capacity = 3 records



2 mono-dimensional B+-trees

- In this case we access 20% of data



B^+-tree efficiency

- In both cases, too much wasted work
- The reason is that points which are close in space are stored in distant leaves
  - In the first case, by the “linearization” of attributes
  - In the other case, by ignoring the other attribute
- Multi-dimensional (spatial) indices try to maintain the spatial proximity of records

Spatial indexing

- Issue emerged in the ’70s due to the insurgence of 2/3-D problems
  - Cartography
  - Geographic Information Systems
  - VLSI
  - CAD
- Recovered in the ’90s to solve problems posed by new applications
  - Multimedia DBs
  - Data mining

Spatial indices: different approaches

- Derived by 1-D structures
  - k-d-B-tree, EXCELL, Grid file
- Mapping from n-D to 1-D
  - Z-order, Gray-order
- Ad-hoc structures
  - R-tree, R*-tree, X-tree, …
- In total: hundreds of data structures

Spatial indices: classification

- Type of objects
  - For points (records cannot have a spatial extension)
  - For regions
- Type of subdivision
  - On the space (splits are performed according to global considerations, à la linear hashing)
    - Good for uniform distributions, simple to implement
  - On the objects (splits are performed according to local considerations, à la B-tree)
    - Good for arbitrary distributions, hard to implement
- Type of organization
  - Tree-/hash-based

Spatial indices: general considerations

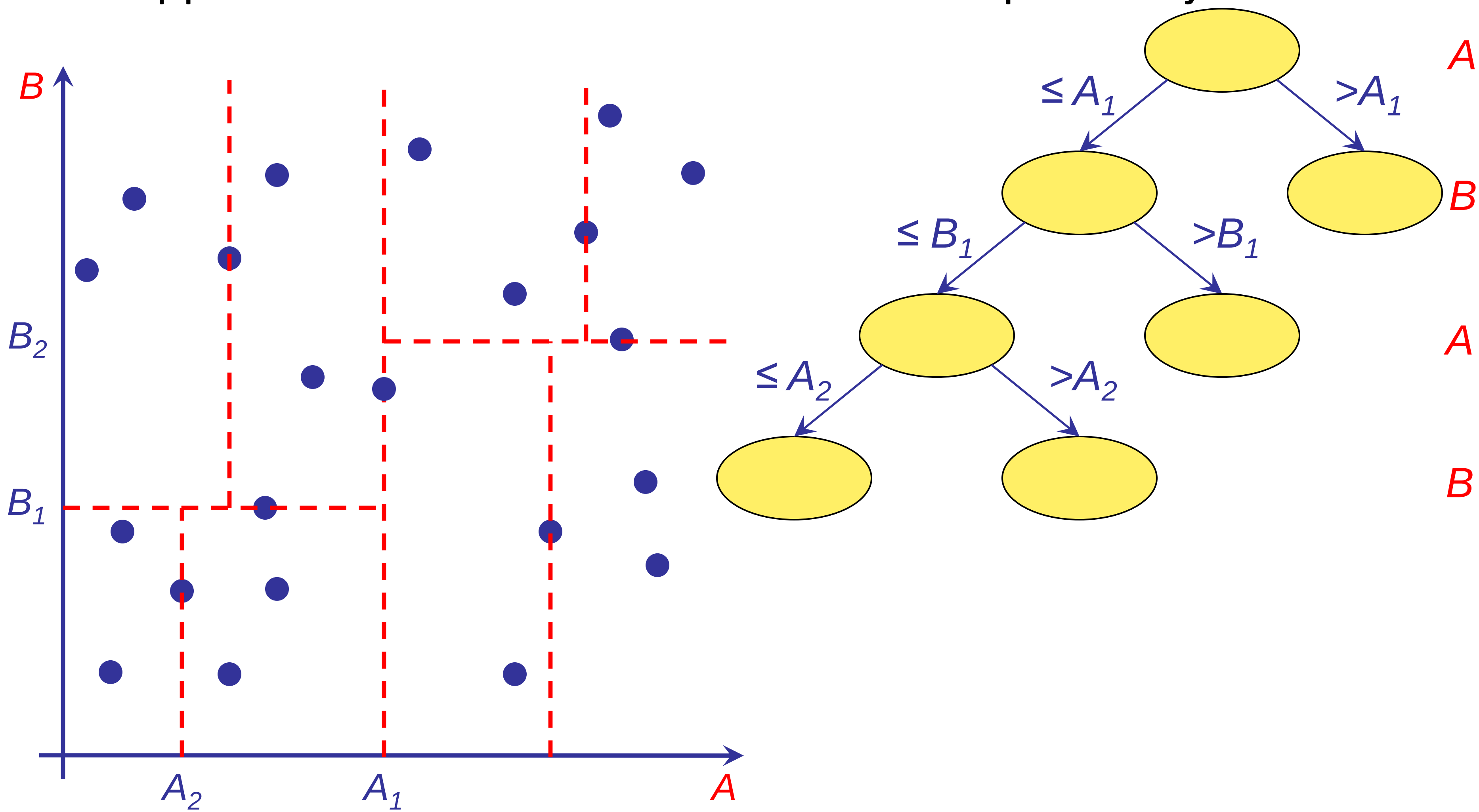
- Fundamental requirement (Local Order Preservation)
  - Group objects (points) in pages, guaranteeing that each page contains objects which are "close" in the n-D space
  - This prevents the use of hash functions, which are not order-preserving
  - The problem is not trivial, since in n-D a global order is not defined (does this sound familiar?)
    - In any case, some solutions define an order in n-D (à-la B+-tree)
- General approach
  - The space is organized in regions (or cells)
  - Each cell is mapped (not always 1-1) to a page

k-d-tree (Bentley, 1975)

- It is a main-memory structure
  - Non paged
  - Non balanced (any problem?)
- Binary search tree
  - Each level is (cyclic) tagged with one of the n coordinates
  - Every node contains a separator, given by the median value of the interval that is being splitted

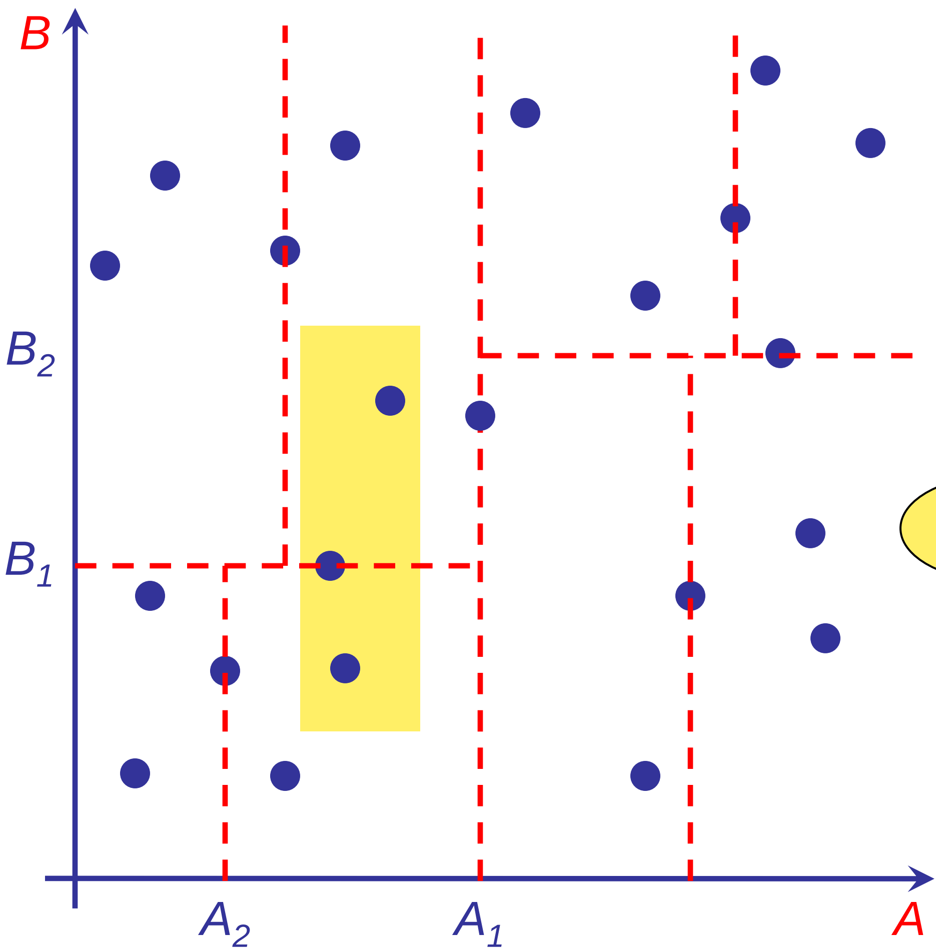
k-d-tree: example

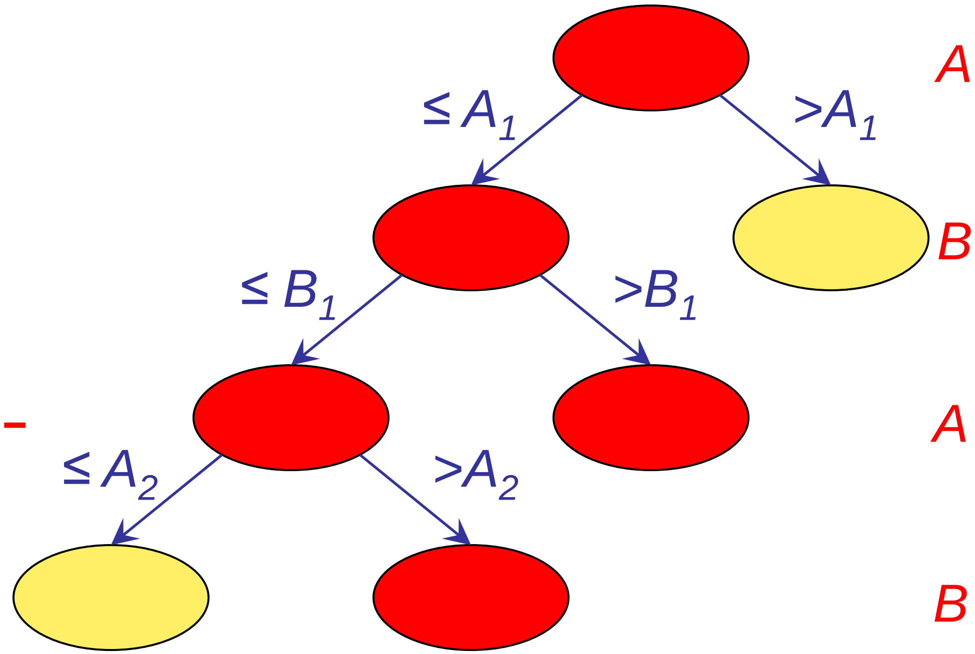
- Suppose that each leaf can accommodate up to 3 objects



k-d-tree: searching

- We visit all branches overlapped with the query





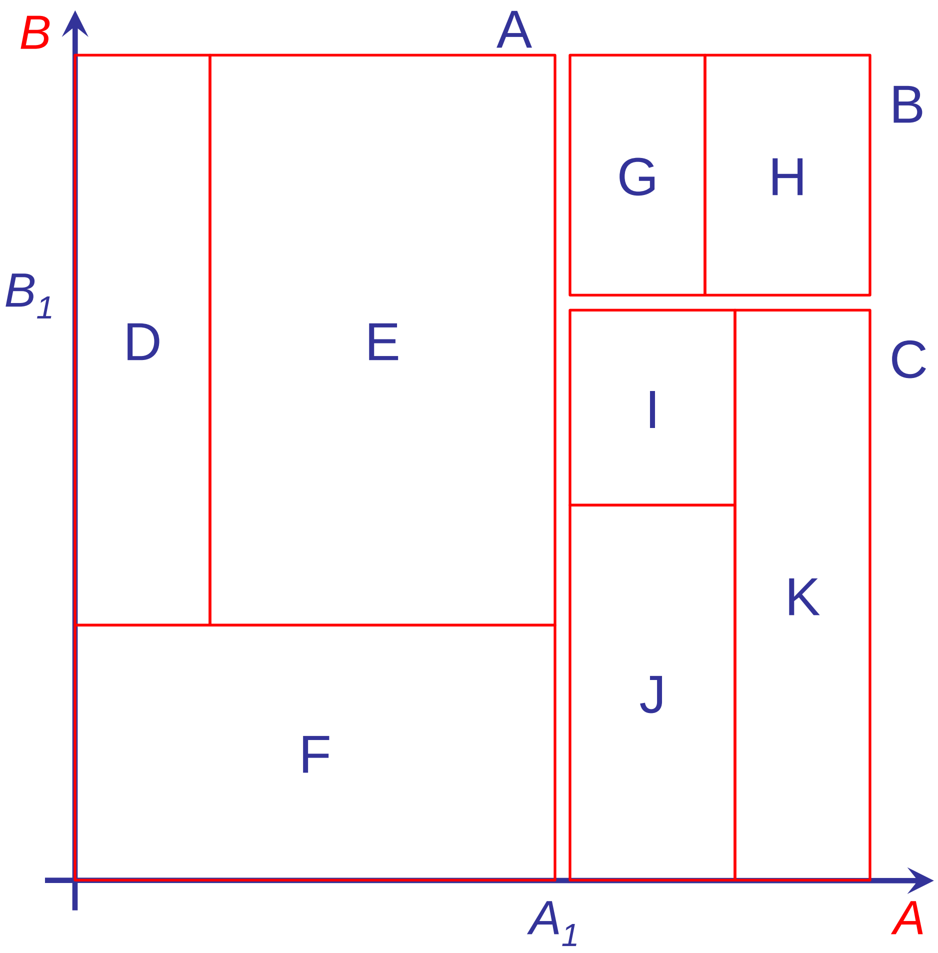
k-d-tree: considerations

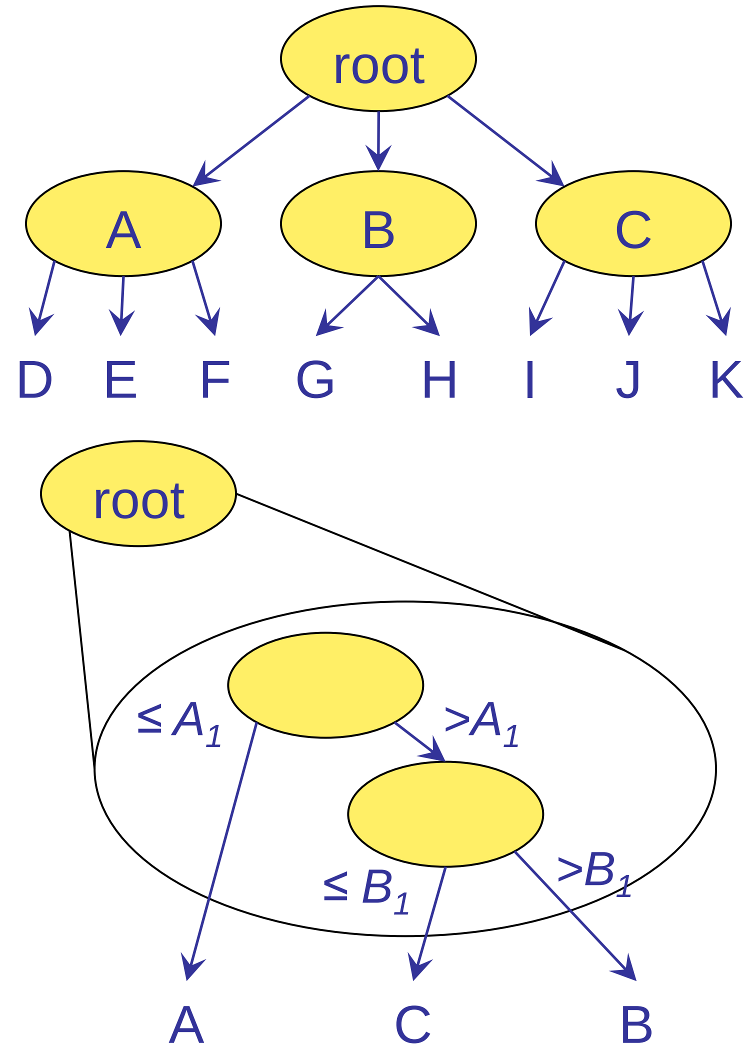
- During insertion, we search for the leaf where the new object should be inserted
  - If this is full: split (downward)
- The tree is not balanced
  - It should be periodically re-organized
- Deletions are extremely complicated
- Several variants which manage separators in different ways, e.g.:
  - BSP-tree uses arbitrary hyperplanes (non-parallel to axes)
  - VAMsplit kd-tree chooses the “best” split coordinate at each node, as the one with maximum variance

k-d-B-tree (Robinson, 1981)

- Paged version of k-d-tree
- The resulting structures resembles a B^+-tree
- Each node (page) corresponds to a (hyper-)rectangular region (box, brick) of the space, obtained as the union of children regions
- Internally, nodes are managed as k-d-trees
  - The “size” of the tree depends on the capacity of a page

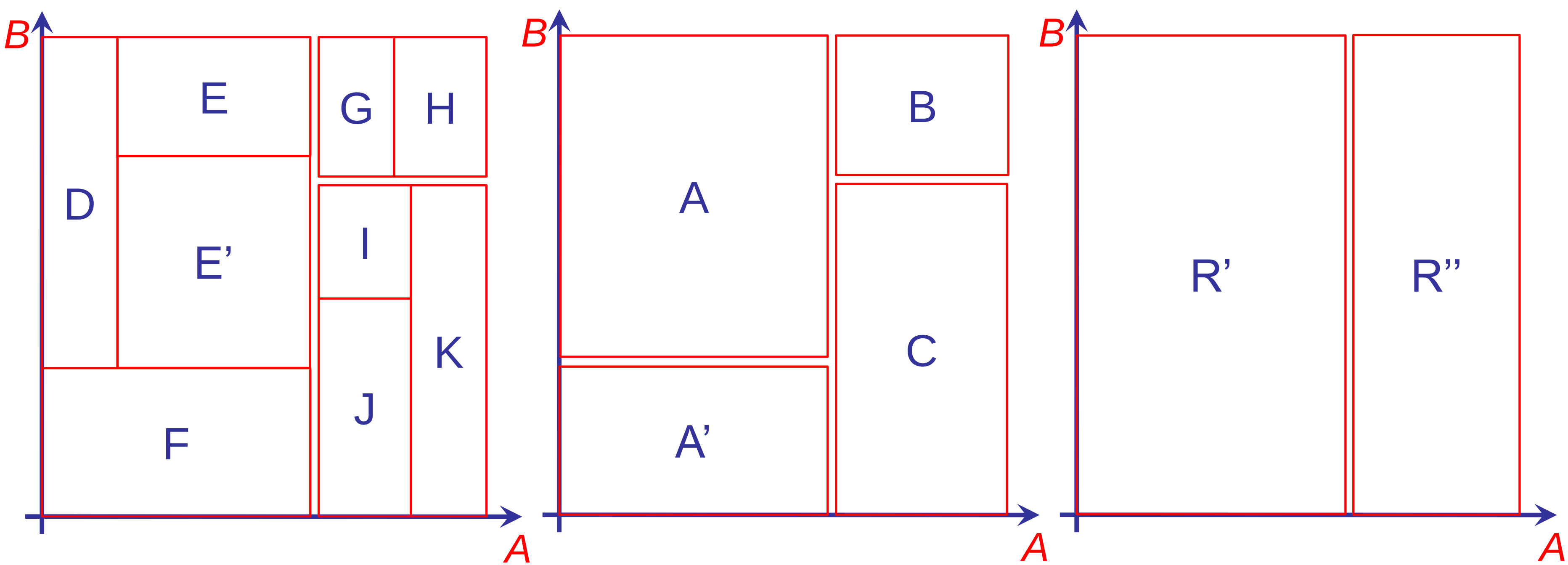
k-d-B-tree: example





k-d-B-tree: node overflow

- If an index node (region) overflows,
  the situation is much complex than in B-tree
- E.g.: split of data block E
  - We partition E, then A, and finally the root

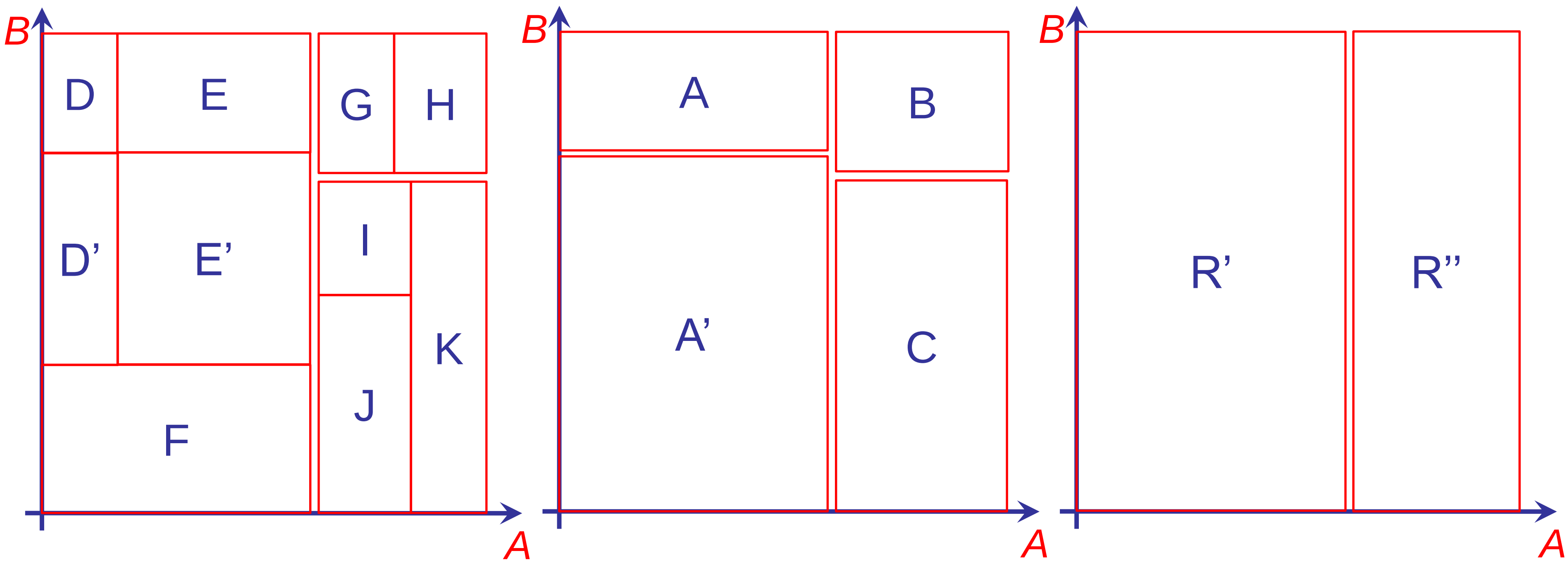


k-d-B-tree: split

- A balanced re-distribution is not always possible
- No lower bound on memory usage (~50-70%)
  - In the example, was partitioned into A and A' according to the first separator
- Robinson algorithmo
  - We consider an hyperplane splitting nodes in a balanced way
  - Splits are propagated downward (to descendant nodes)

k-d-B-tree: Robinson algorithm

- The A region is split into A’ and A’’
  - D is split into D and D’

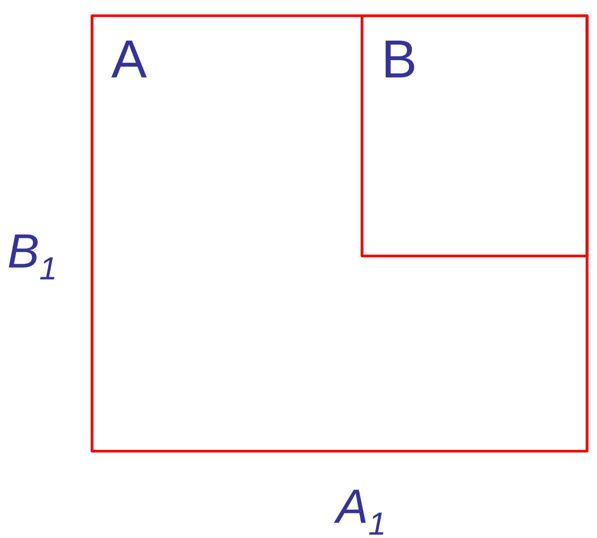


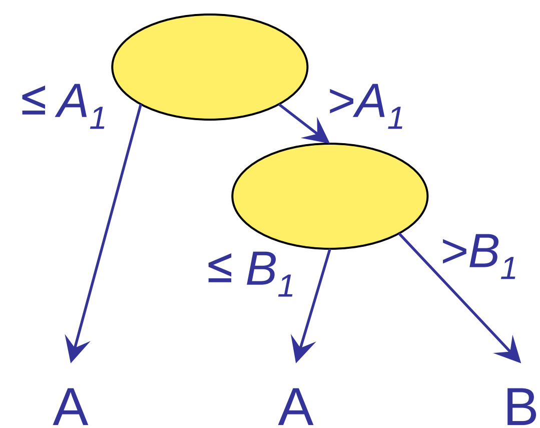
hB-tree (Lomet & Salzberg, 1990)

- Variant of k-d-B-tree
- Regions can contain “holes” (hB = “holey brick”)
- Positive effects:
  - Split of a data block: we can guarantee that, in the worst case, data are partitioned according to a 2:1 ratio
    (2/3 in one block and 1/3 in the other one)
  - Split of an index node: we obtain a balanced split
    (and thus a lower bound to the memory usage)
    without propagating splits to the descendant nodes

hB-tree: split of a data page

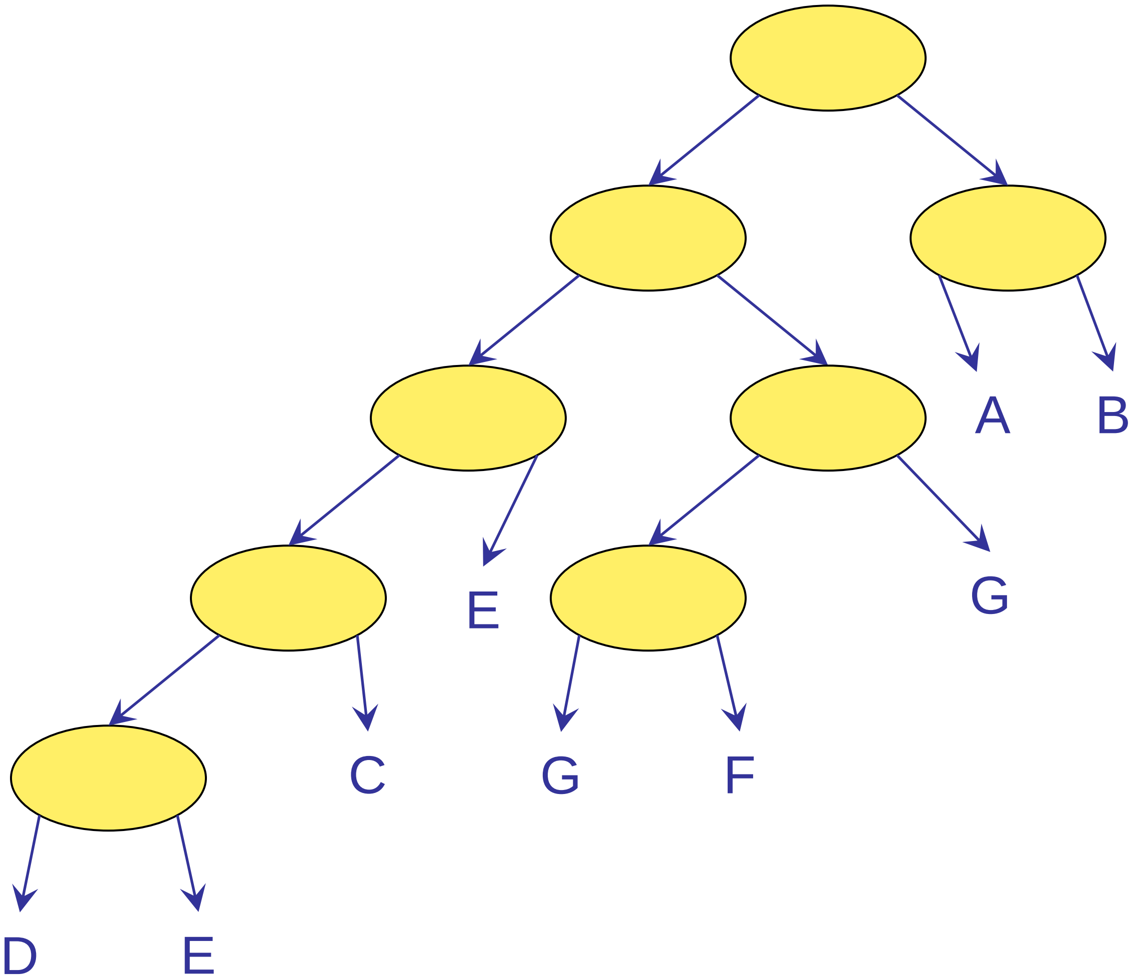
- As in k-d-B-tree, each node is internally organized as a k-d-tree
- The difference here is that a node can be “referenced” by multiple separations

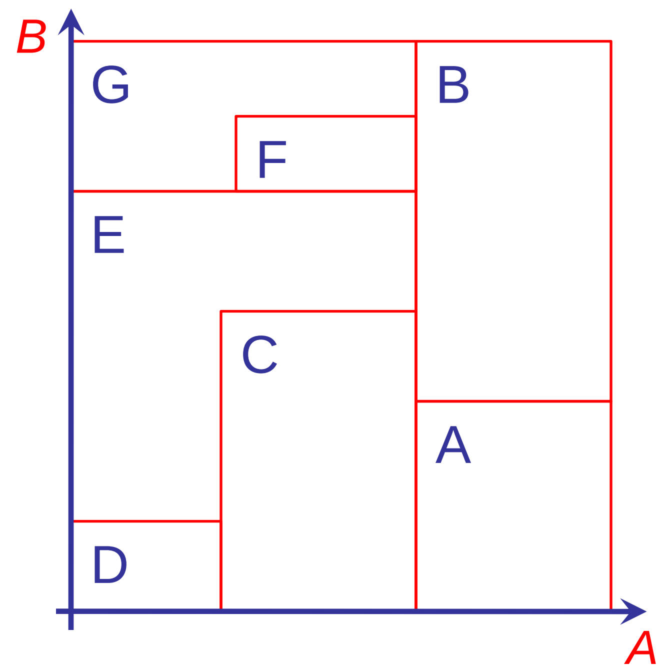




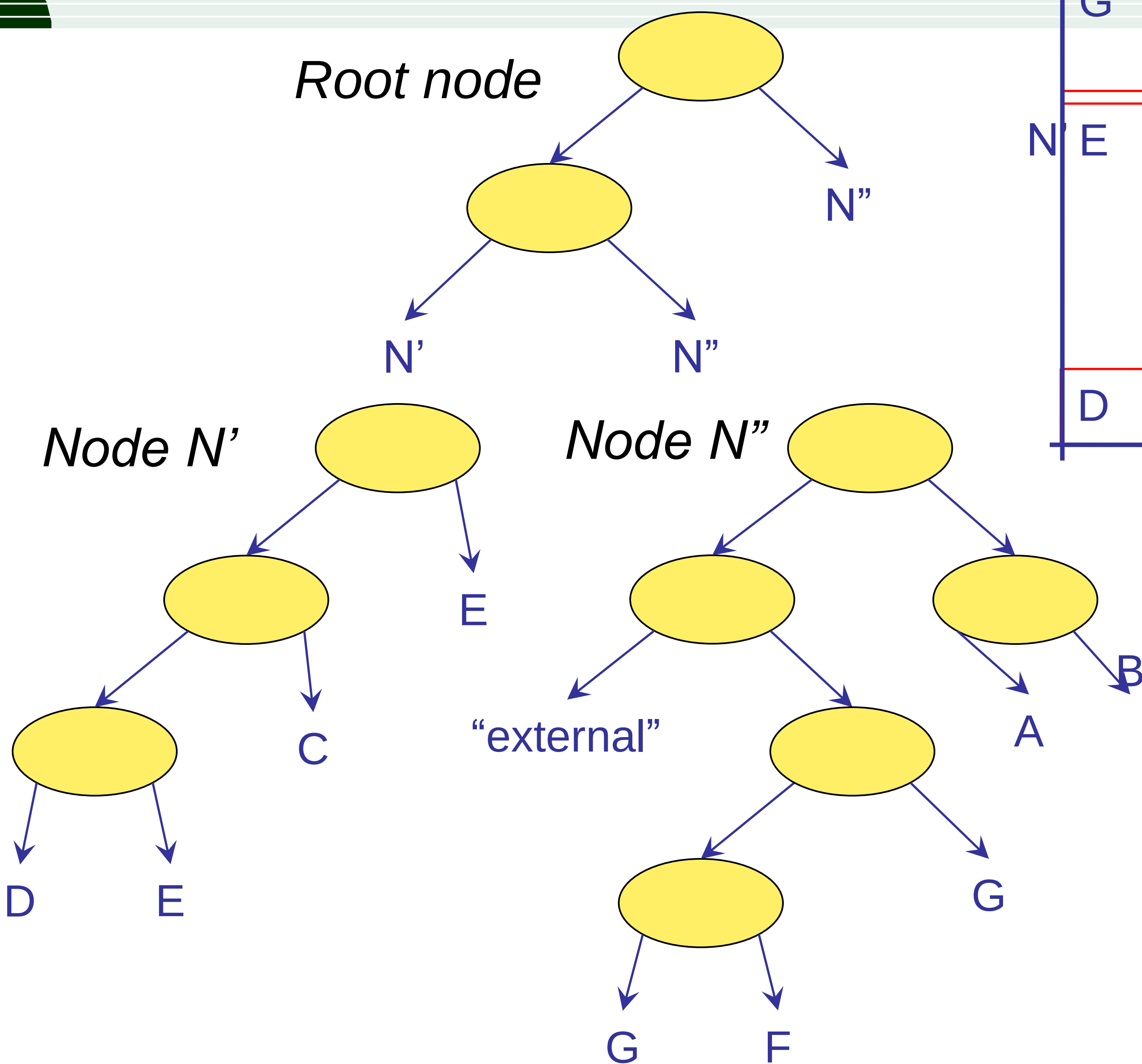
hB-tree: split example (i)

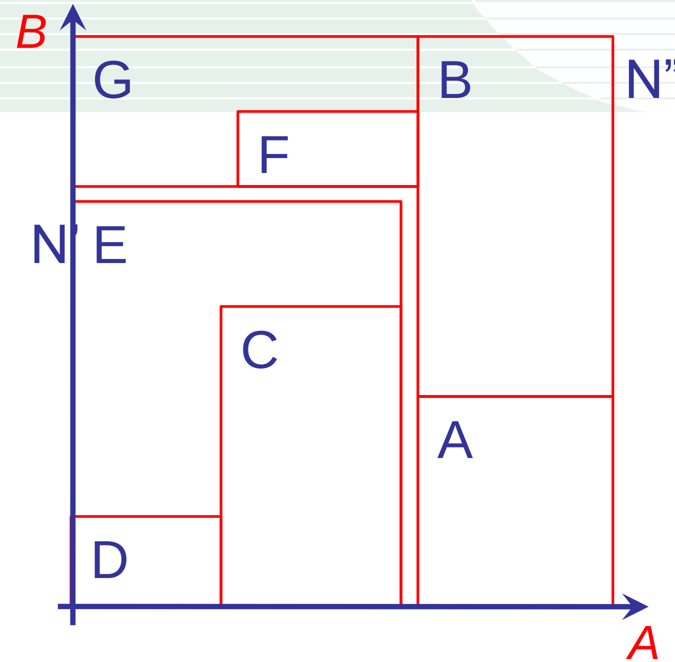
- Suppose that each page can contain up to 7 nodes
- The root overflows





hB-tree: split example (ii)



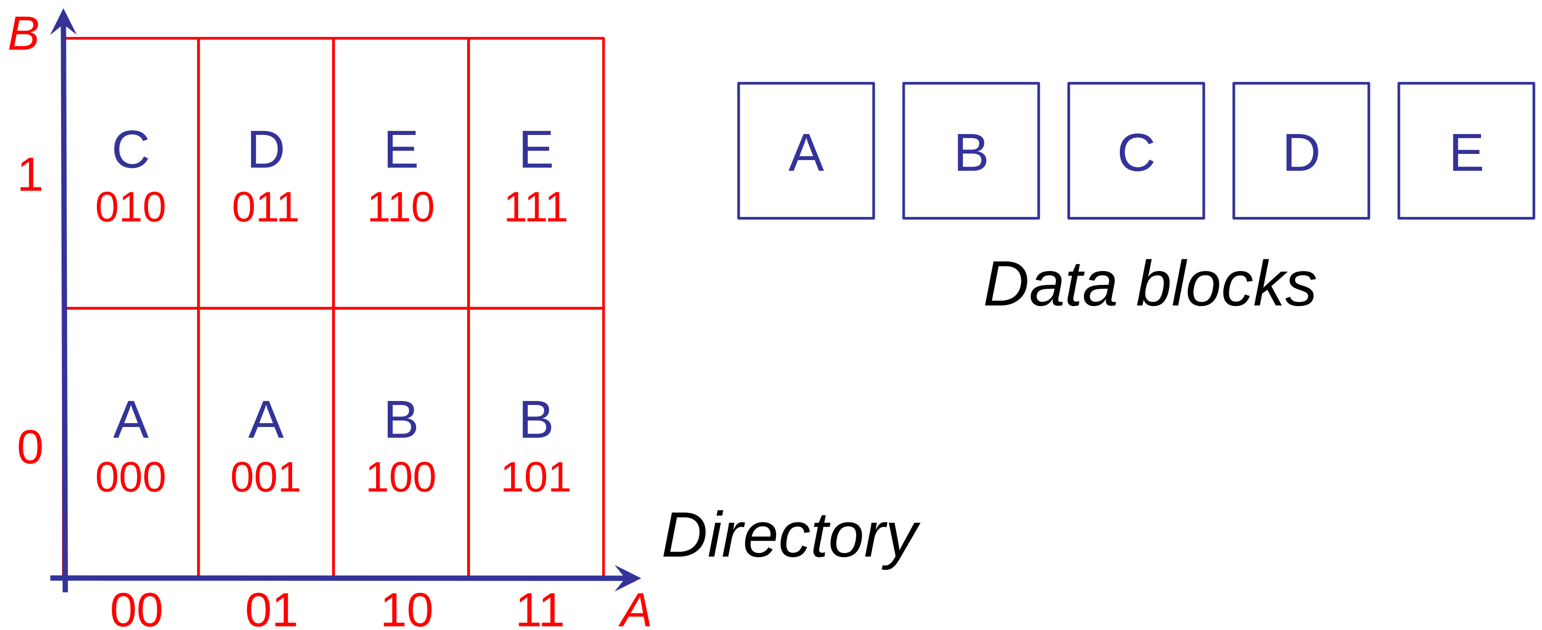


EXCELL (Tamminen, 1982)

- Uses a hash-based directory, regular grid in n dimensions
  - Each directory cell corresponds to a data page, but the converse is not necessarily true
  - The address of a cell is formed by interleaving coordinates bits
- Extends extendible hashing to multiple dimensions

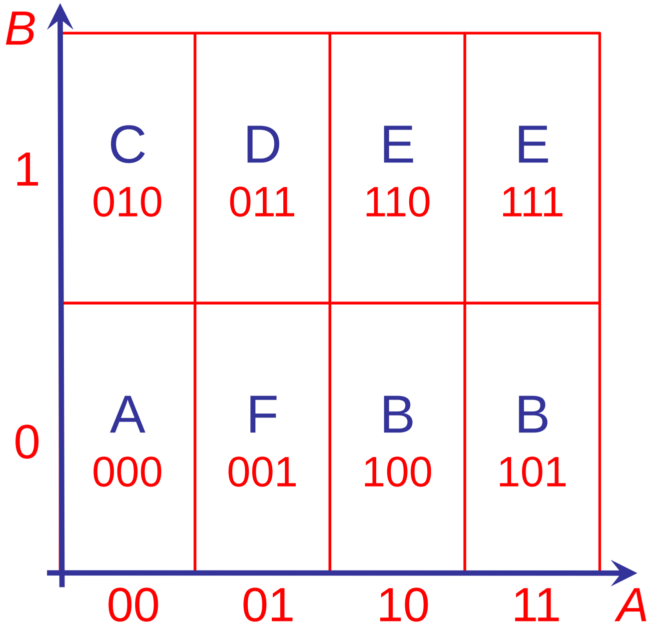
EXCELL: example

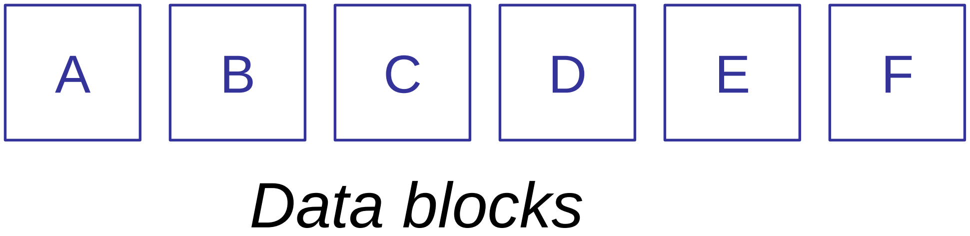
- When a data page overflows, it is split and, for the directory, we can have one of two cases
  - If the block was referenced by two (or more) cells, we only update pointers
  - Otherwise, the directory is doubled, by using an additional bit



EXCELL: split (i)

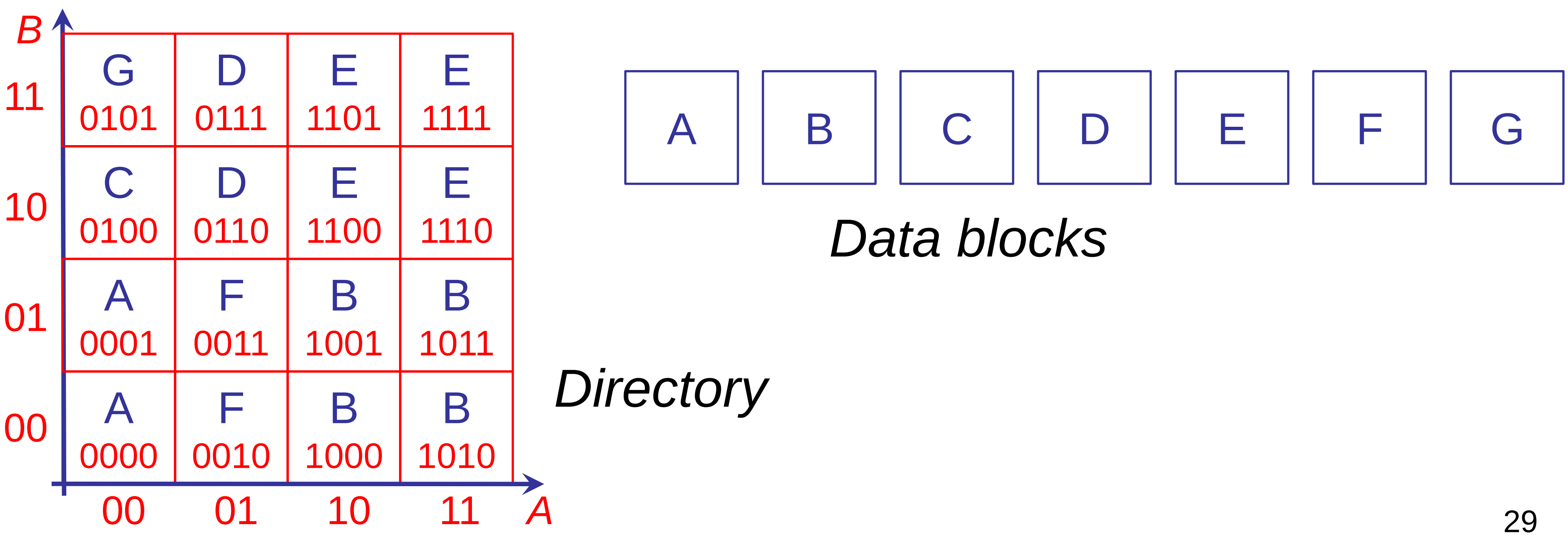
- First case: A overflows and is split into A and F
- It is sufficient to update the pointer in cell 001





EXCELL: split (ii)

- Second case: C overflows and is split into C and G
- We have to double the directory using an additional bit for coordinate B



EXCELL: considerations

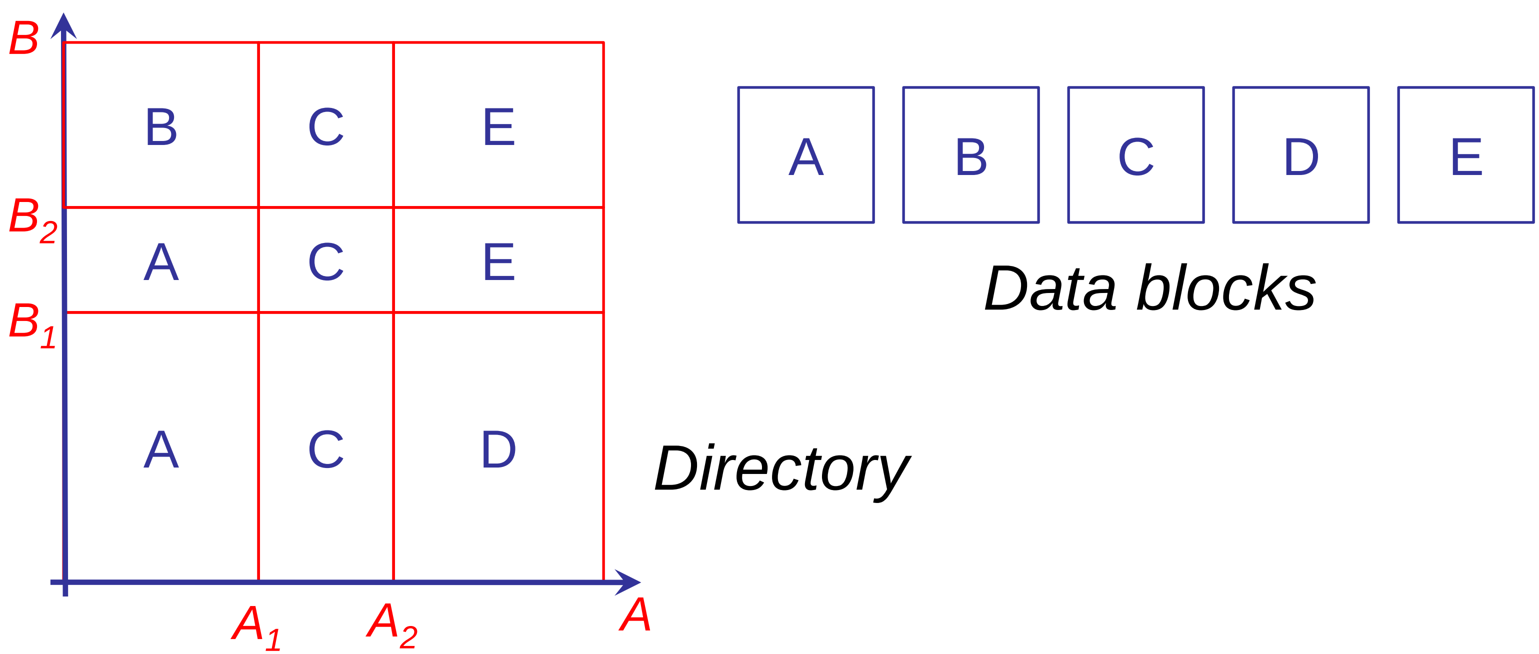
- The same arguments used for extendible hashing apply here
- Doubling the directory is sometimes not enough to solve the overflow of a bucket (why?)
- It works well for uniform distribution of data

Grid file (Nievergelt et al., 1984)

- Generalizes EXCELL, allowing to define arbitrarily sized intervals
  - To this aim, d scales are required, containing values used as separators for each dimension
- In case of intervals defined by way of a binary partitioning, scales are analogous to the directory of dynamic hashing

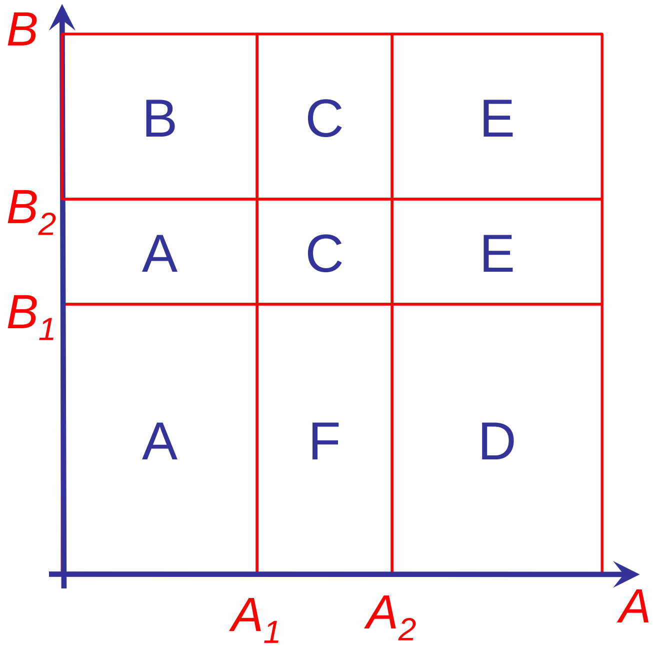
Grid file: example

- When a data page overflows, it is split and, for the directory, we can have one of two cases
  - If the block was referenced by two (or more) cells, we only update pointers
  - Otherwise, we add a separator to the directory

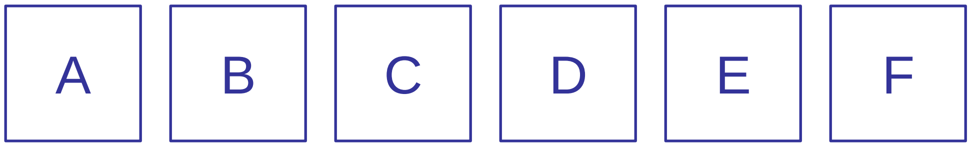


Grid file: split (i)

- First case: C overflows and is split into C and F
- It is sufficient to update the pointer of the cell



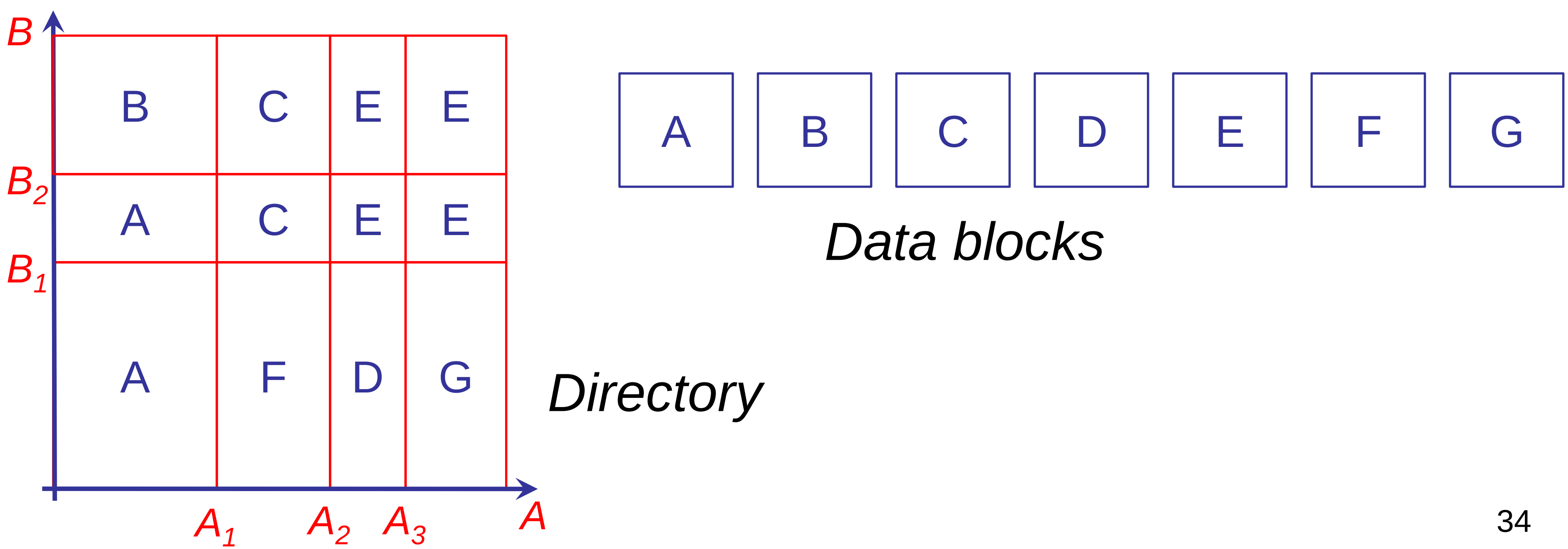
Directory



Data blocks

Grid file: split (ii)

- Second case: D overflows and is split into D and G
- We have to augment the directory using an additional separation, for example for coordinate A



Grid file: considerations

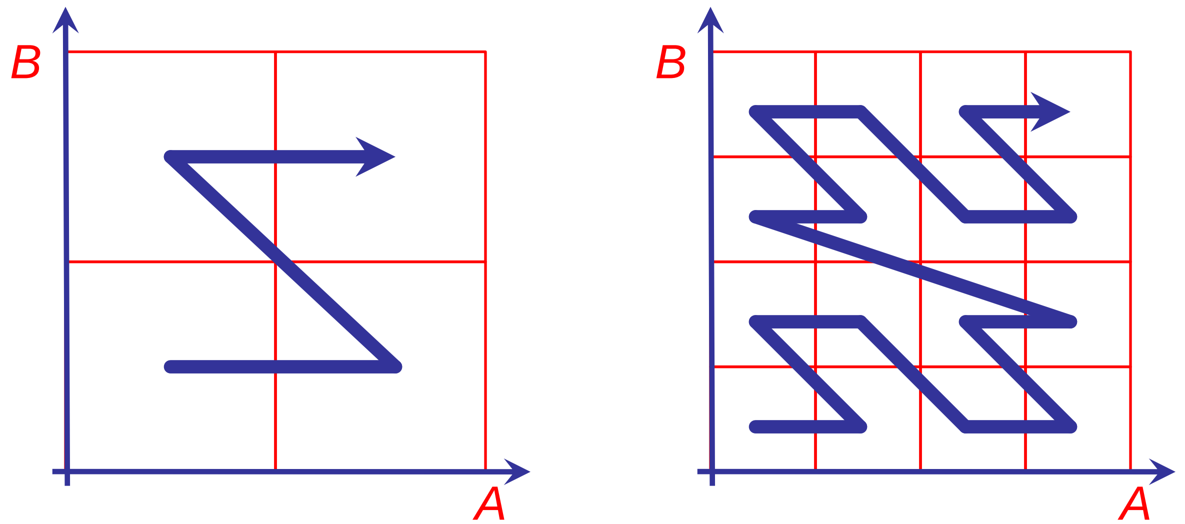
- In case of non-uniform distributions, storing N points could require a number of cells which grows like $O(N^d)$
- On the other hand, the regular structure of space partitioning greatly simplifies the resolution of window queries
- Main problem: directory management
  - Usually, scales are stored in main memory
  - In (quasi-)static cases, the directory can be stored on disk as a multi-dimensional array
  - In dynaimic cases, it is necessary to paginate the directory, leading to multi-level grid files

Mono-dimensional sorting

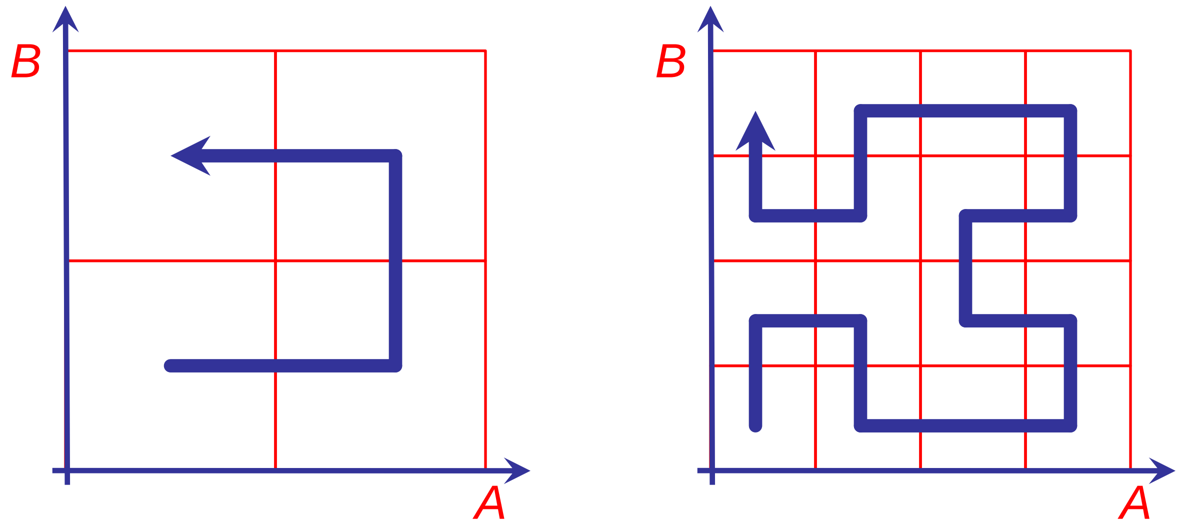
- We try to “linearize” the n-dimensional space so as to be able to exploit a mono-dimensional index, like the B^+-tree
- We obtain so-called “space-filling curves”
- Local Order Preservation requirement
  - Points which are “close” in the n-D space should also be close in the linearization

Examples of curves (i)

- Z-order

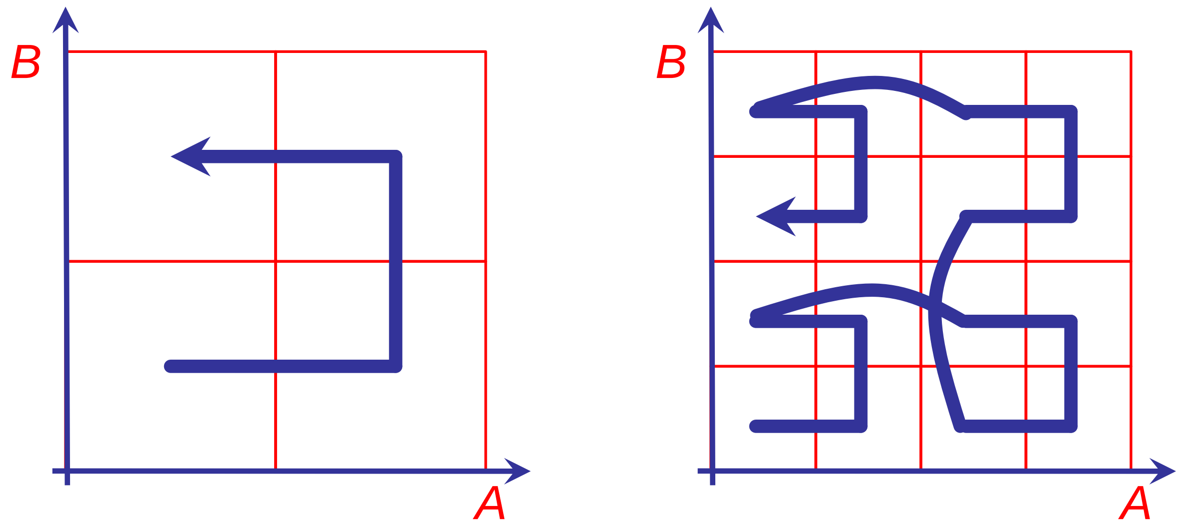


- Peano-Hilbert

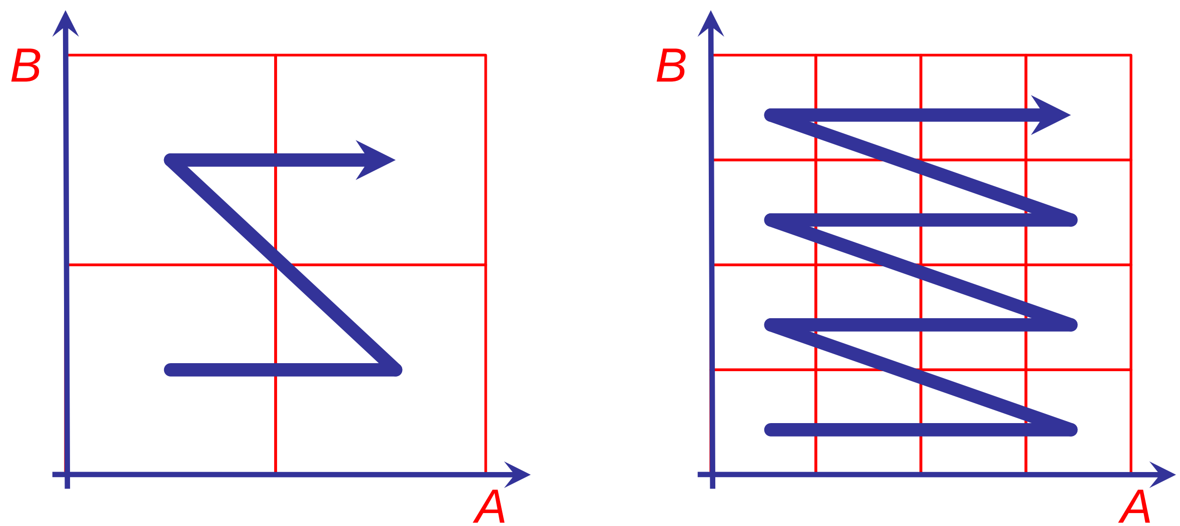


Examples of curves (ii)

- Gray-order



- Lexicographic order



Space-filling curves: considerations

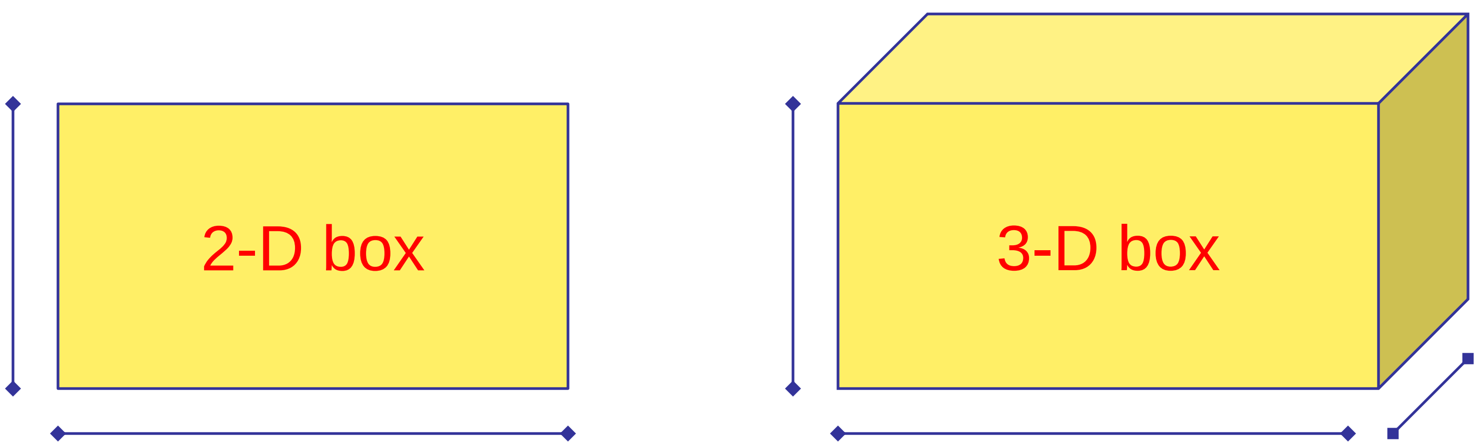
- As it is clear, no curve satisfies the local order preservation requirement
- Solving window queries is therefore plagued by the same problems seen for multi-attribute B^+-tree
  - Can we see analogies/equivalencies?
- Nearest neighbor search is further complicated…

R-tree (Guttman, 1984)

- Balanced and paginated tree-shaped structure, based on the hierarchical nesting of overlapping regions
- Each node corresponds to a rectangular region, defined as the MBB containing all children regions
- Storage utilization for each node varies from 100% to a minimum value (≤ 50%) which is a design parameter of R-tree
- Management mechanisms similar to those of B^+-tree, with the main difference that insertion of an object and possible splits can be managed according to different policies

R-tree: concept of MBB

- MBB = Minimum Bounding Box
  - The smallest rectangle, with sides parallel to coordinate axes, containing all children regions
  - It is defined as the product of n intervals

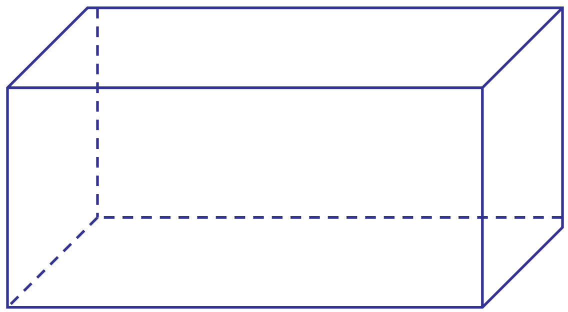


R-tree: definition of MBB (i)

- How many vertices has a n-dimensional (hyper-)rectangle? 2^n
- In order to define a (hyper-)rectangle we should specify the coordinates of all its vertices
- Moreover, the algorithm for computing the smallest (hyper-)rectangle containing a set of N points has a complexity
  - $O(N^2)$ in 2-dim
  - $O(N^3)$ in 3-dim
  - No algorithm is known for dim>3

R-tree: definition of MBB (ii)

- How many values are required for defining a box? 2n
  - It is sufficient to provide the coordinates of two any opposite vertices



- What is the complexity of the algorithm for computing the MBB for a set of N points? O(N)
  - It is sufficient to find the minimum and maximum value for each coordinate

R-tree: comparison with B^+-tree

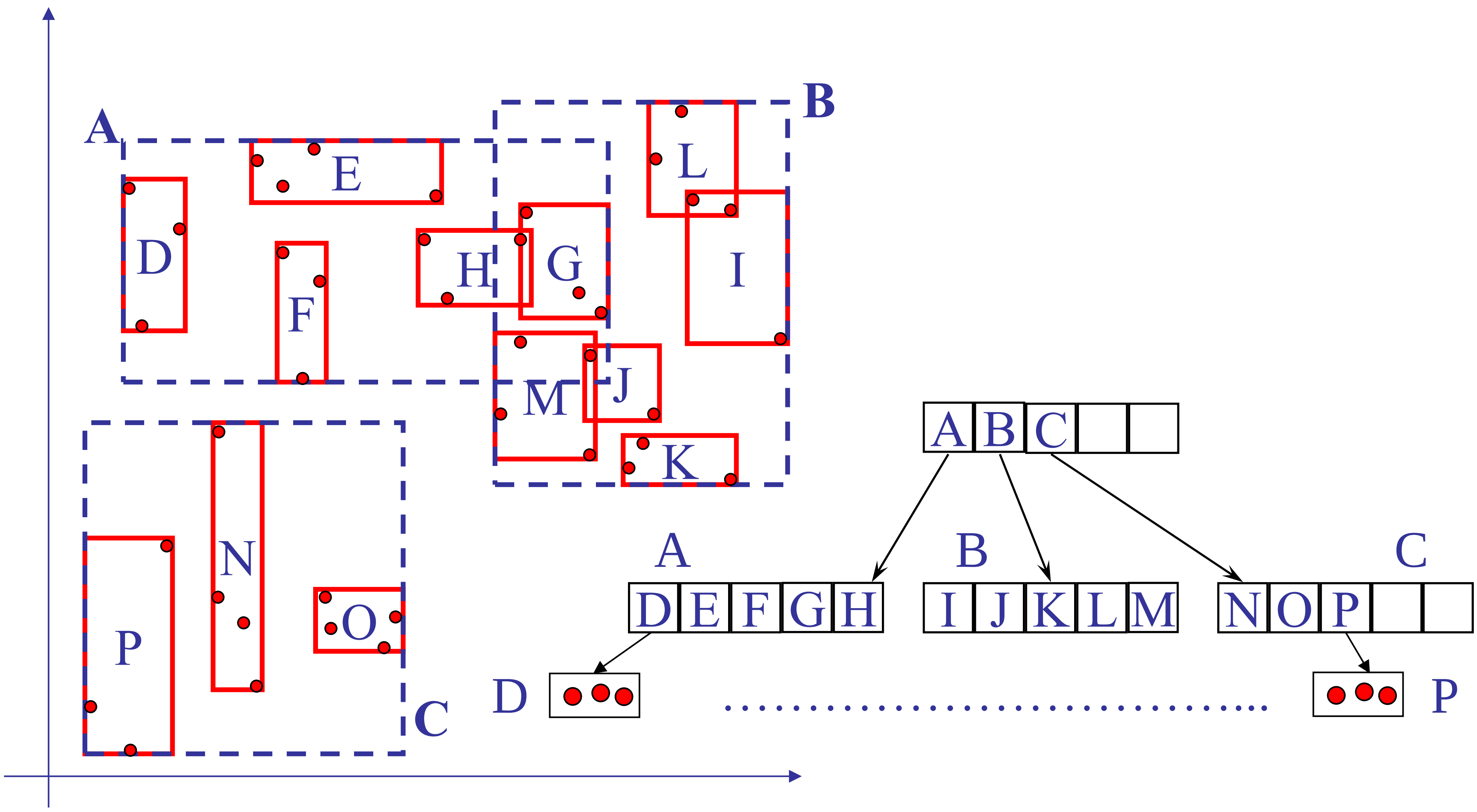
B^+-tree

- Balanced and paginated tree
- Data are stored in leaves
- Leaves are kept sorted
- Data are organized into 1-D intervals
  - Intervals do not overlap
- This principle is recursively applied towarts the root
- Point search follows a single path from root to a single leaf

R-tree

- Balanced and paginated tree
- Data are stored in leaves
- No data order exist
- Data are organized into n-D intervals (MBB)
  - Intervals do overlap (characteristic of n-D space)
- This principle is recursively applied towarts the root
- Point search could follow multiple paths from root to multiple leaves

R-tree: organization



R-tree: characteristics (i)

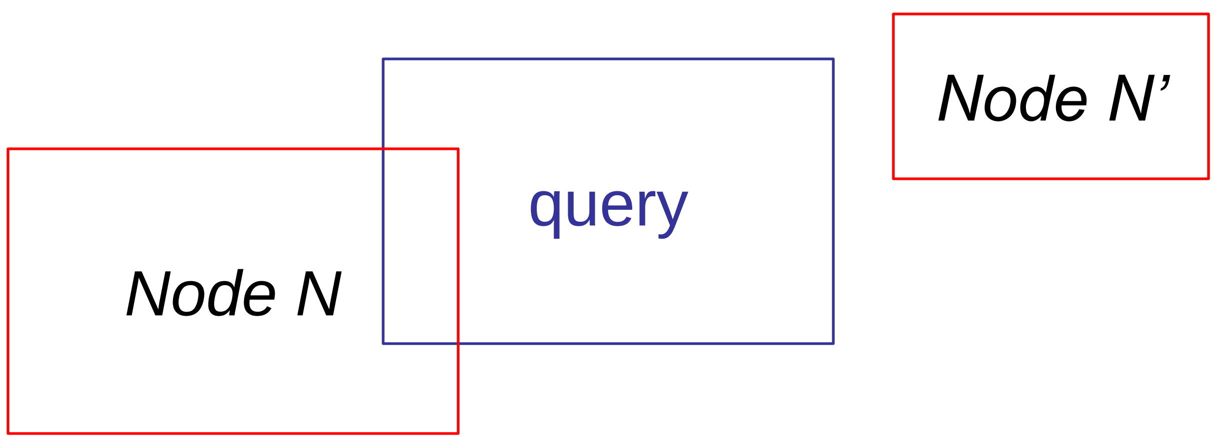
- Leaf nodes
  - Contain entries with the form (key, RID), where key stores the record coordinates
  - Actually, R-tree could also store n-dim objects with a spatial extensione, with key=MBB
- Internal nodes
  - Contain entries with the form (MBB, PID), where MBB stores the coordinates of the MBB containing children entries
- Overall, each node contains entries with the form (key, ptr), where key is a “spatial” value

R-tree: characteristics (ii)

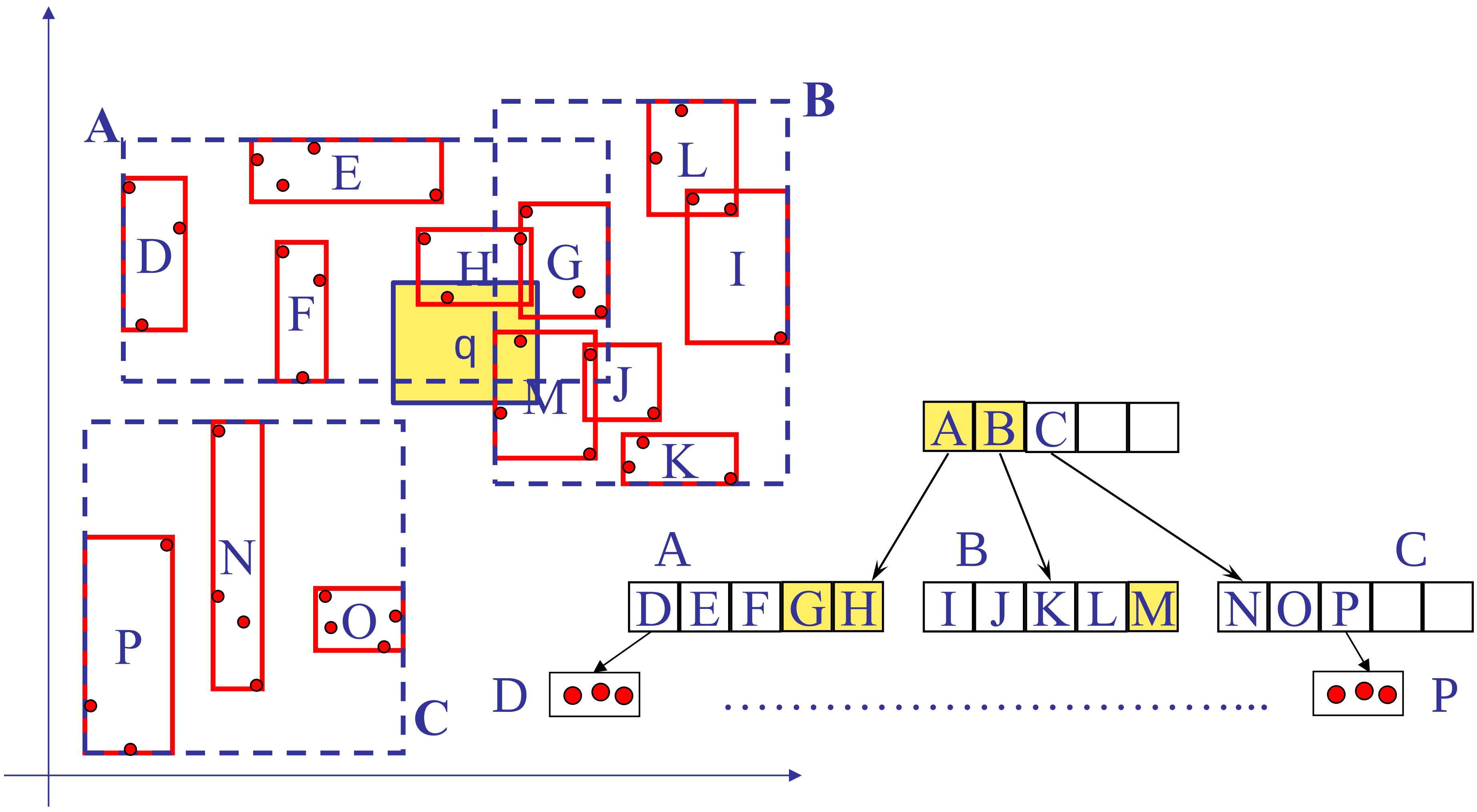
- Each node contains a number **m** of entries which can vary between **c** and **C**
  - $c \leq C/2$ is a storage utilization parameter
  - C depends on n and the page size
- As usual, the root can violate the minimum utilization constraint and contain only two entries

R-tree: search (window query)

- We have to retrieve all points included into a product of n intervals (that is, a box)
- Such points could only be found in nodes whose MBB overlaps with the query region
- E.g.: node N’ cannot contain records satisfying the query



R-tree: search example



R-tree: search algorithm

- Consistent(E,q)
  - **Input**: Entry E=(p,ptr) and search predicate q
  - **Output**: **if** p & q == false **then** false **else** true
- Both p and q are (hyper-)rectangles
- Consistent returns true if and only if p and q have non-null overlap
  - Consistent is oblivious to the “shape” of q
    - Could also be used for different queries (range, NN)
- It follows that the search can follow multiple paths within the tree

R-tree: construction algorithms

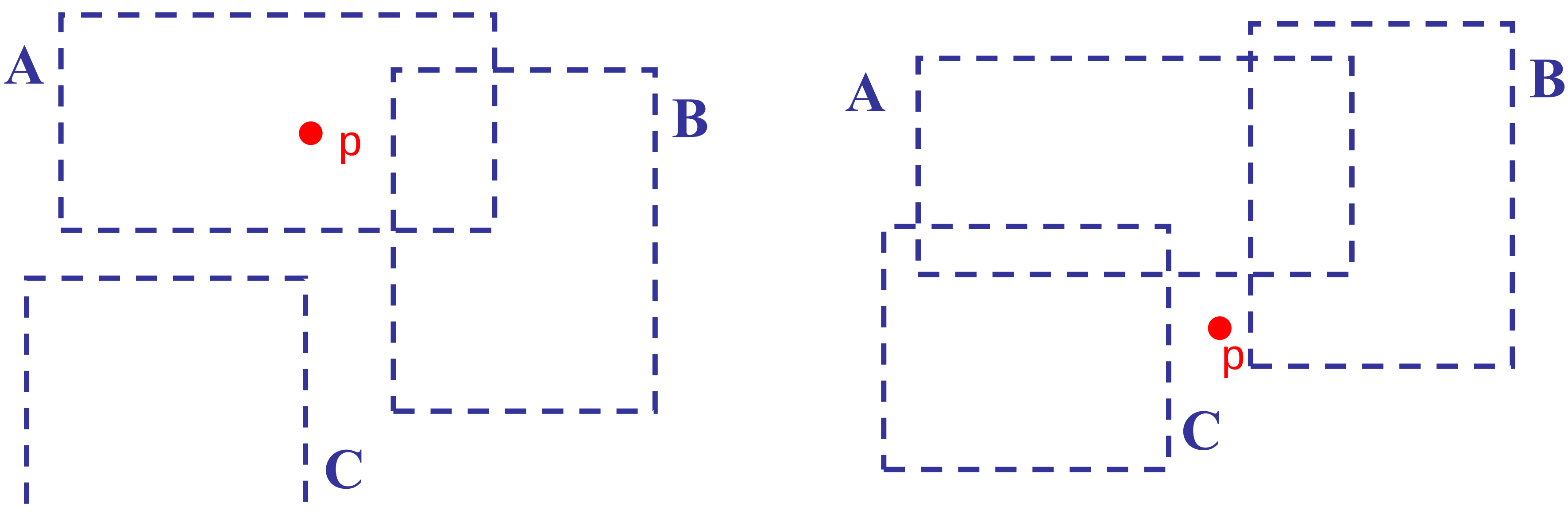
- We need to specify key methods Union, (Compress, Decompress,) Penalty, and PickSplit
- Different “variants” of R-tree exist, each differing from the others on how such choices are implemented
- We will see the implementation of the original R-tree and will discuss some variants
  - One of the most common is R*-tree (Beckmann et al., 1990)

R-tree: Union

- Union(P)
  - **Input**: Set of entries $P = \{(p_1,ptr_1),\ldots,(p_n,ptr_n)\}$
  - **Output**: A predicate r holding for all tuples reachable through one of the entries' pointers
- Both r and p_js are (hyper-)rectangles
- We return the MBB containing all p_js
- It is sufficient to compute the minimum and maximum value on each coordinate

R-tree: Penalty (i)

- Penalty(E_1,E_2)
  - **Input**: Entries $E_1 = (p_1, ptr_1)$ and $E_2 = (p_2, ptr_2)$
  - **Output**: A "penalty" value resulting from inserting E_2 into the sub-tree rooted at E_1
- What is the best way to insert a point?

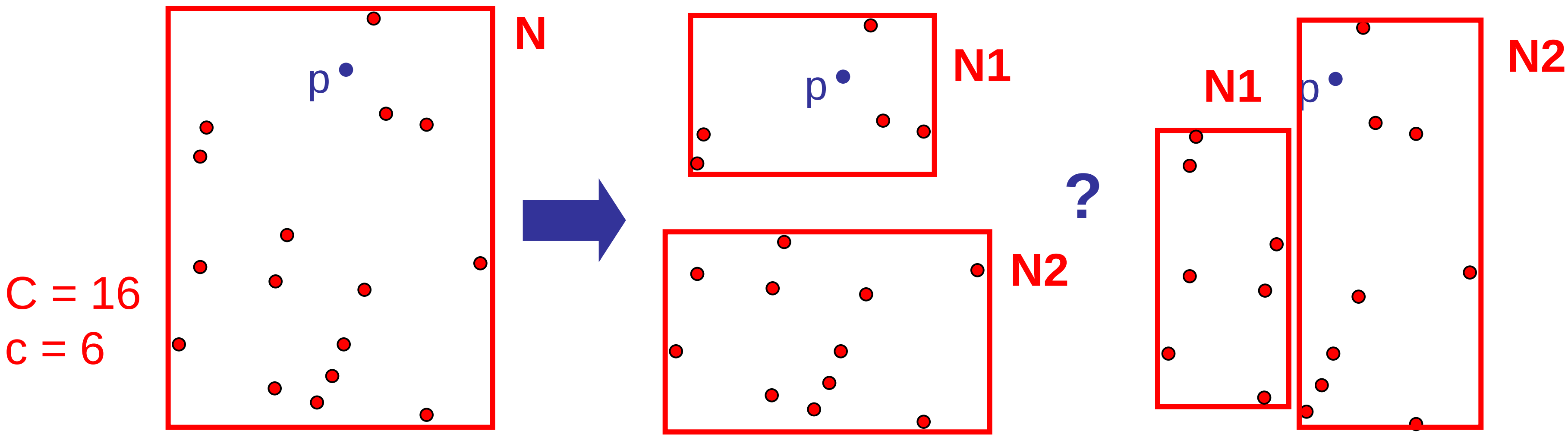


R-tree: Penalty (ii)

- If p is contained in E_1, the penalty is 0
- Otherwise, the penalty is given by the increment of volume (area) of the MBB
  - However, if we are in a leaf, R*-tree considers the increment of intersection with other entries
- Both criteria aim to obtain a tree with better performance:
  - Large volume: the chance of visiting the node during a query increases
  - Large overlap: the number of nodes visited during a query increases

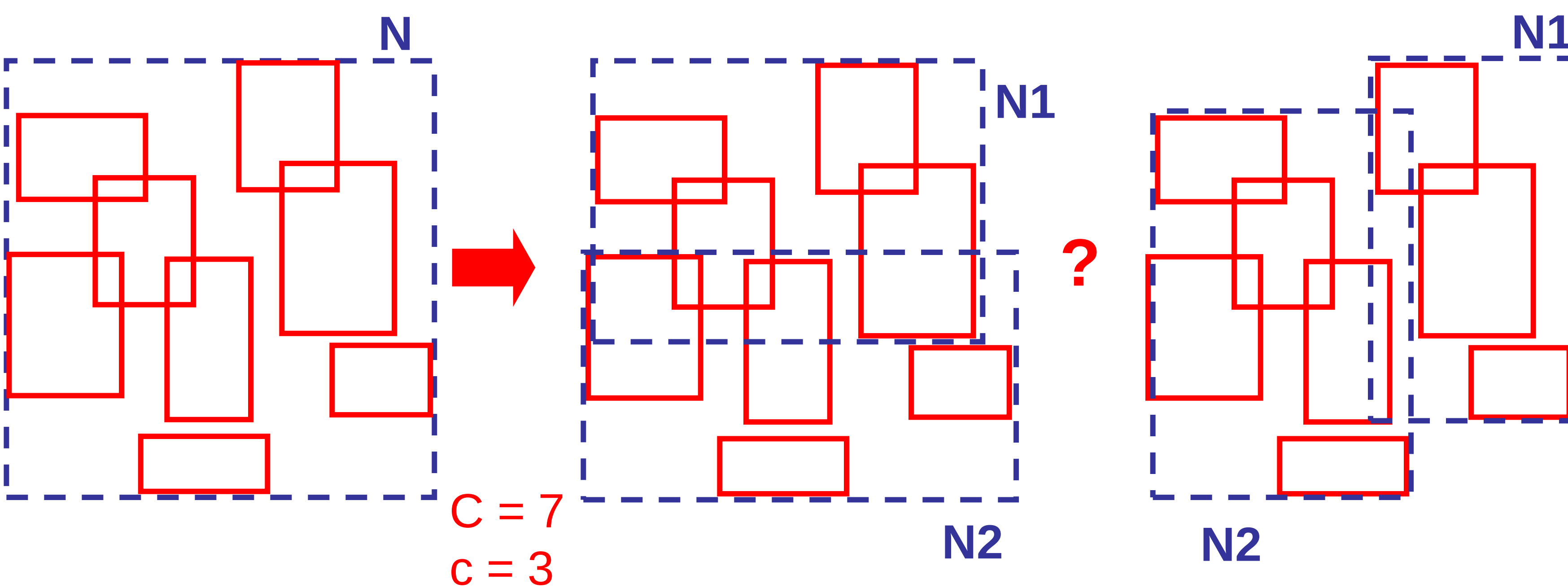
R-tree: Picksplit (i)

- PickSplit(P)
  - **Input**: Set of di C+1 entries
  - **Output**: two sets of entries, P_1 and P_2, with cardinality $\geq$ c



R-tree: Picksplit (ii)

- Search for a split minimizing the sum of volumes of the two nodes
  - Unfortunately, it is a NP-hard problem, thus we use heuristics
- Things gets worse in upper nodes
  - In particular, an overlap-free split is not guaranteed



R-tree: Picksplit (iii)

- The criterion adopted by R*-tree is more complicated and considers both nodes volume and perimeter and their overlap
- Moreover, R*-tree supports re-distribution in both overflow and underflow
  - All such choices are implemented through heuristics, since their efficiency is validated only experimentally
  - We obtain (slight) performance improvements for insertion, search, and storage utilization