Multidimensional Data and Modelling

Indexing of locational data

- Multidimensional data is a collection of points in a higher dimensional space which can represent locations and objects in space
- When data spans a continuous physical space (i.e., an infinite collection of locations), the issues become more interesting, maybe we are also interested in the space that they occupy
- For example, a line in two-dimensional space can be represented by the coordinate values of its endpoints and then stored as a point in a four-dimensional space, so we have constructed a transformation from a two-dimensional space to a four-dimensional space

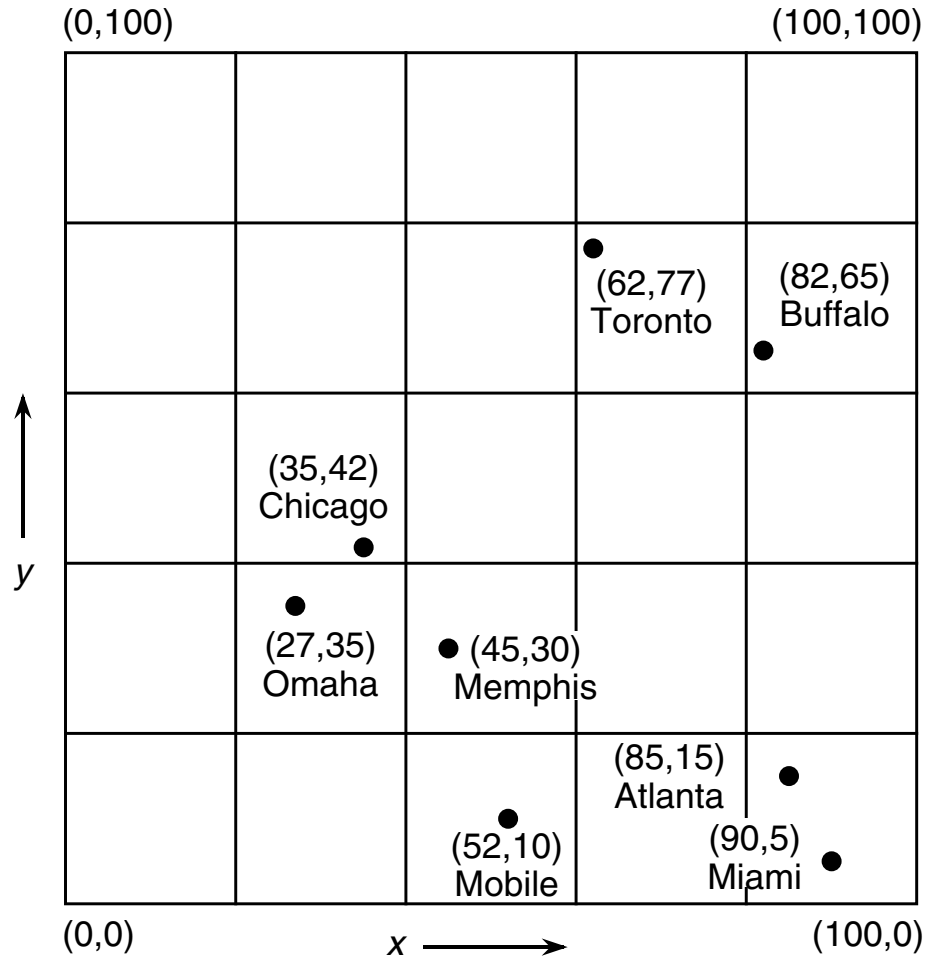
Indexing of locational data

- The transformation approach is fine if we are just interested in retrieving the data.
- It is appropriate for queries about the objects (e.g., determining all lines that pass through a given point or that share an endpoint, etc.) and the immediate space that they occupy.
- However, the drawback of the transformation approach is that it ignores the geometry inherent in the data and its relationship to the space in which it is embedded.

Indexing of locational data

- So, we explore a number of different representations of multidimensional data bearing the above issues in mind.
- The approach is primarily a descriptive one. Most of examples are of two-dimensional spatial data although the representations are applicable to higher dimensional spaces as well.
- The problem is that such analyses are difficult to perform for many of the data structures. This is true for data structures that are based on spatial occupancy like quadtree or R-tree or variants. Uniform grid:

Indexing of locational data



Indexing by subspaces (kd-tree)

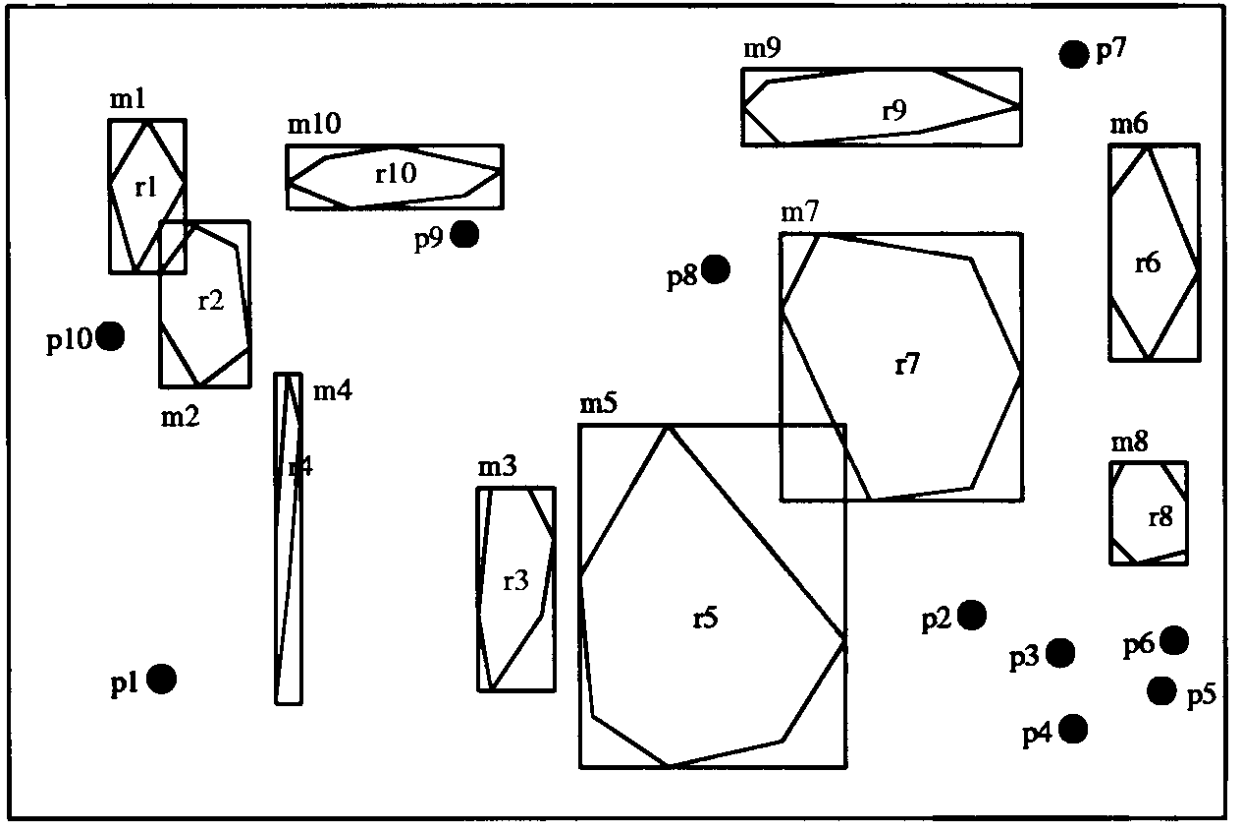
- **Input:** A set S of ***n*** points in k-dimensions.
- **Problem:** Construct a tree (hierarchical data structure) which partitions the space by half-planes such that each point is contained in it is own region.
- Typical algorithms construct k-d-trees by partitioning point sets. Each node in the tree is defined by a plane through one of the dimensions that partitions the set of points into left/right (or up/down) sets, each with half the points of the parent node.
- These children are again partitioned into equal halves, using planes through a different dimension.

Indexing by subspaces (kd-tree)

- Partitioning stops after *lg n* levels, with each point in its own leaf cell.
- Although many different flavors of kd-trees have been devised, their purpose is always to hierarchically decompose space into a relatively small number of cells such that no cell contains too many input objects.
- This provides a fast way to access any input object by position.
- Alternate kd-tree construction algorithms insert points incrementally and divide the appropriate cell, although such trees can become seriously unbalanced.

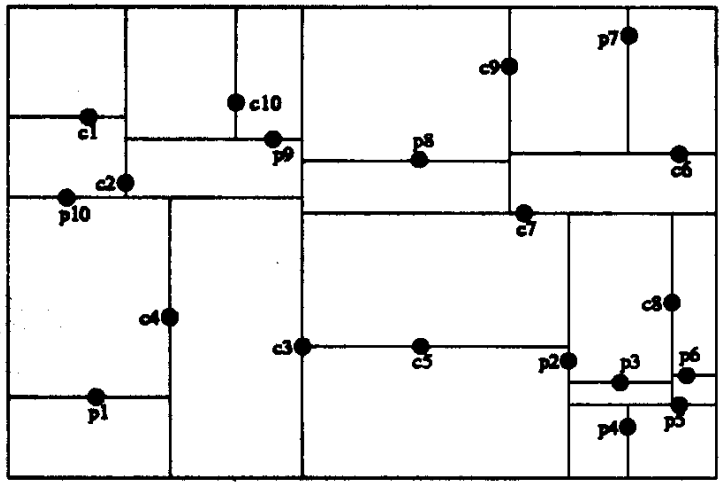
Indexing by subspaces (kd-tree)

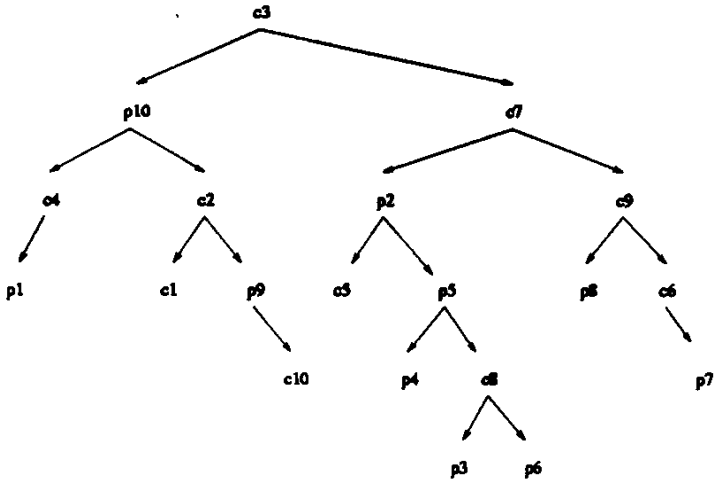
Running example:



Indexing by subspaces (kd-tree)

K – D – Tree

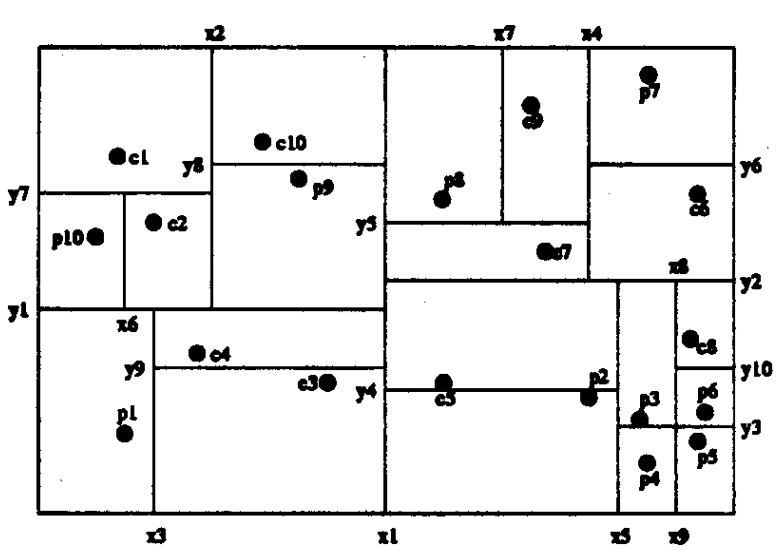


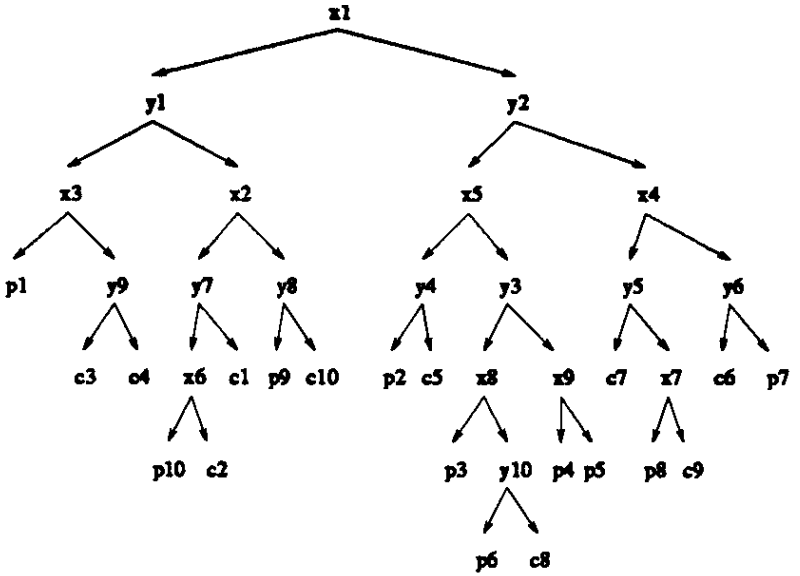


k-d-tree is a binary search tree that represents the recursive subdivision of the universe into subspaces by means of (d-1)-dimensional hyperplanes.

Indexing by subspaces (kd-tree)

Adaptive k-d-tree





Indexing by subspaces (kd-tree)

- Adaptive k-d-tree choose a split such that one finds about the same number of elements on both sides.
- While the splitting hyperplanes are still parallel to the axes, they do not have to contain a data point and their directions do not have to be strictly alternating anymore.
- As a result, the split points are not part of the input data; all data points are stored in the leaves.
- Interior nodes contain the dimension (e.g. x or y) and the coordinate of the corresponding split. Splitting is continued recursively until each subspace contains only a single point.

Indexing by subspaces (kd-tree)

- The k-d-tree and adaptive k-d-tree are not very dynamic structure; it is obviously difficult to keep the tree balanced in the presence of frequent insertions and deletions.
- The structure works best if all the data is known a priori and if updates are rare.

Indexing by subspaces (BSP-tree)

- Splitting the universe only along iso-oriented hyperplanes is a severe restriction.
- Allowing arbitrary orientations gives more flexibility to find a hyperplane that is well-suited for the split.
- The *binary space partitioning (BSP)* tree are binary trees that represent a recursive subdivision of the universe into subspaces by means of (d - 1)-dimensional hyperplanes.

Indexing by subspaces (BSP-tree)

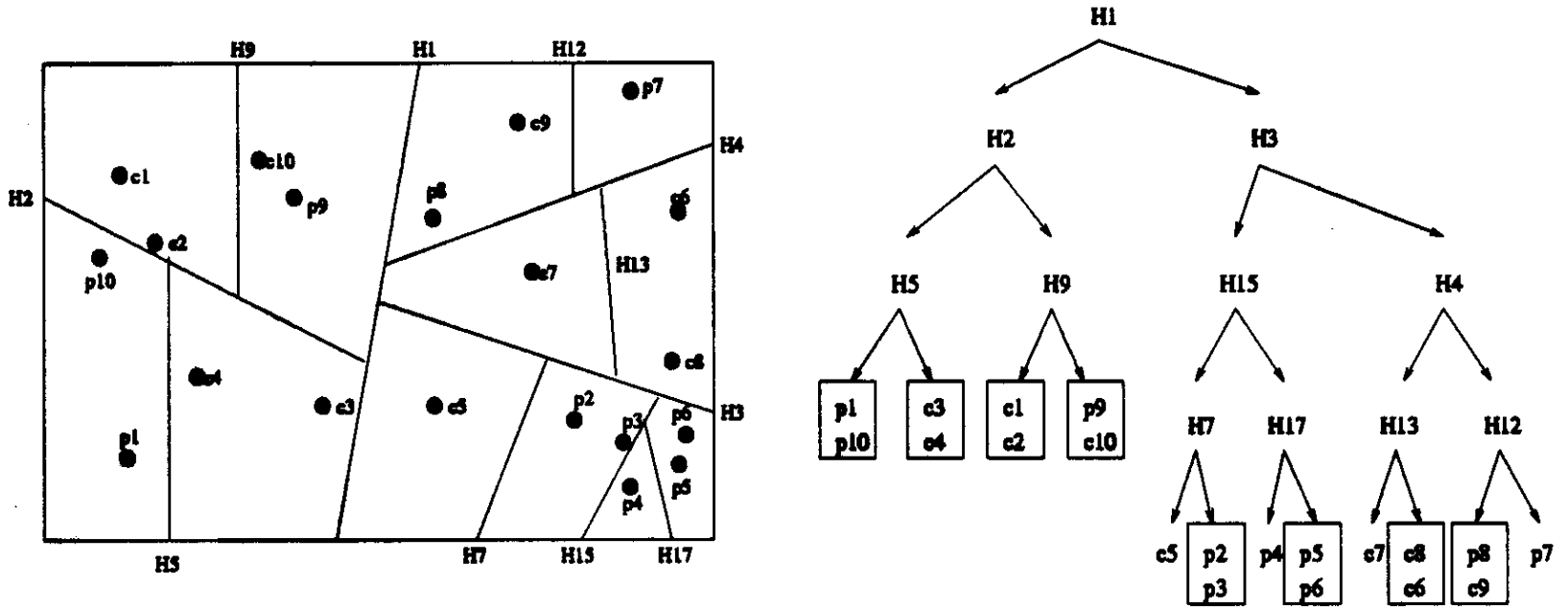
- Each subspace is subdivided independently of its history and of the other subspaces.
- The choice of the partitioning hyperplanes depends on the distribution of the data points in a given subspace.
- The decomposition usually continues until the number of points in each subspace is below a given threshold.
- Binary space partitioning was developed in the context of 3D computer graphics (rendering), constructive solid geometry, collision detection in robotics and 3-D video games, ray tracing and other computer applications that involve handling of complex spatial scenes.

Indexing by subspaces (BSP-tree)

- The resulting partition of the universe can be represented by a BSP tree, where each hyperplane corresponds to an interior node of the tree and each subspace corresponds to a leaf.
- Each leaf stores references to those data points that are contained in the corresponding subspace.
- Figure shows a BSP tree for the running example with no more than two data points per subspace.

Indexing by subspaces (BSP-tree)

BSP Tree



Indexing hierarchies (2-3-4-tree)

- Self balancing trees: expand on the idea of 2-3 trees by adding the 4-node
- Addition of this third item simplifies the insertion logic

FIGURE 11.48
2-, 3-, and 4-Nodes

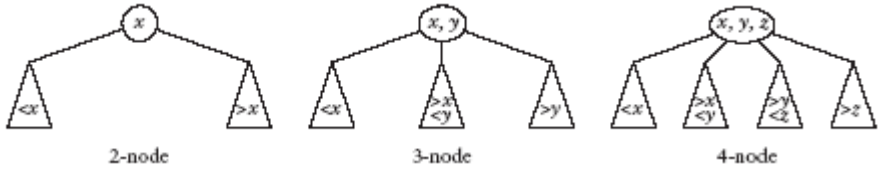


FIGURE 11.49
Example of a 2-3-4 Tree

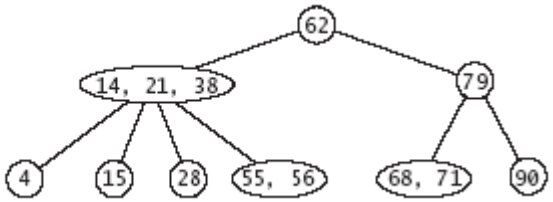


FIGURE 11.50
Result of Splitting a 4-Node

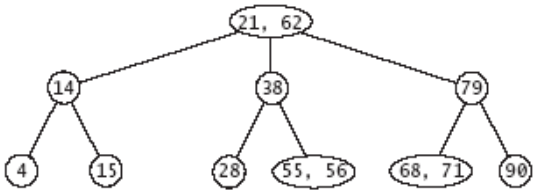


FIGURE 11.51
2-3-4 Tree After Inserting 25

Indexing hierarchies (B-tree)

- B-Tree: A Search Structure for Large Dynamic Indexes
- A B-tree extends the idea behind the 2-3 and 2-3-4 trees by allowing a maximum of data items in each node
- The order of a B-tree is defined as the maximum number of children for a node
- B-trees were developed to store indexes to databases on disk storage

Indexing hierarchies (B-tree)

- B-Tree: A Search Structure for Large Dynamic Indexes

FIGURE 11.57
Example of a B-Tree

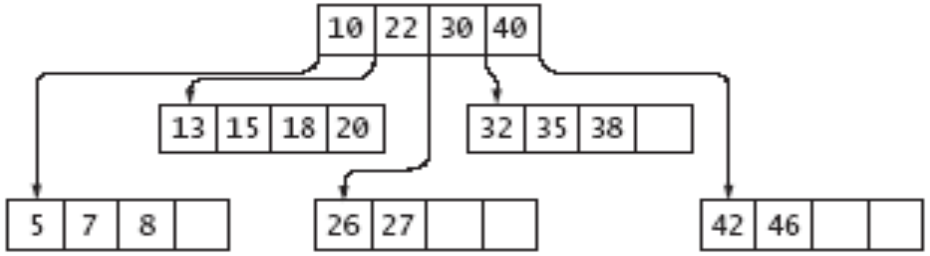
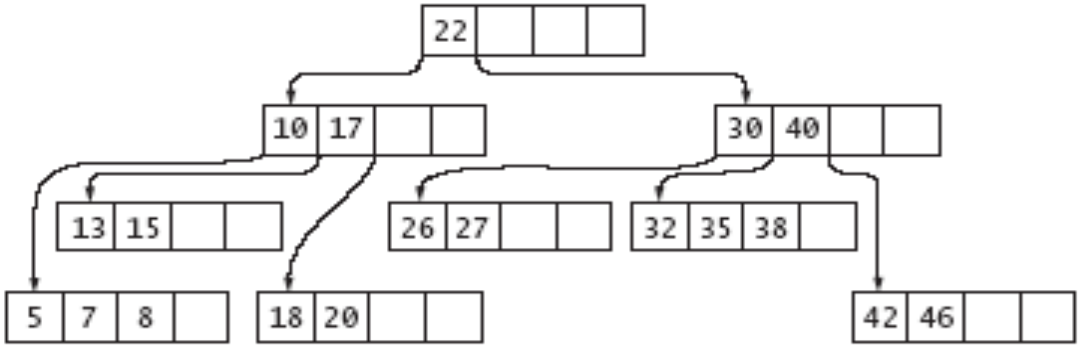


FIGURE 11.58
Inserting into a B-Tree



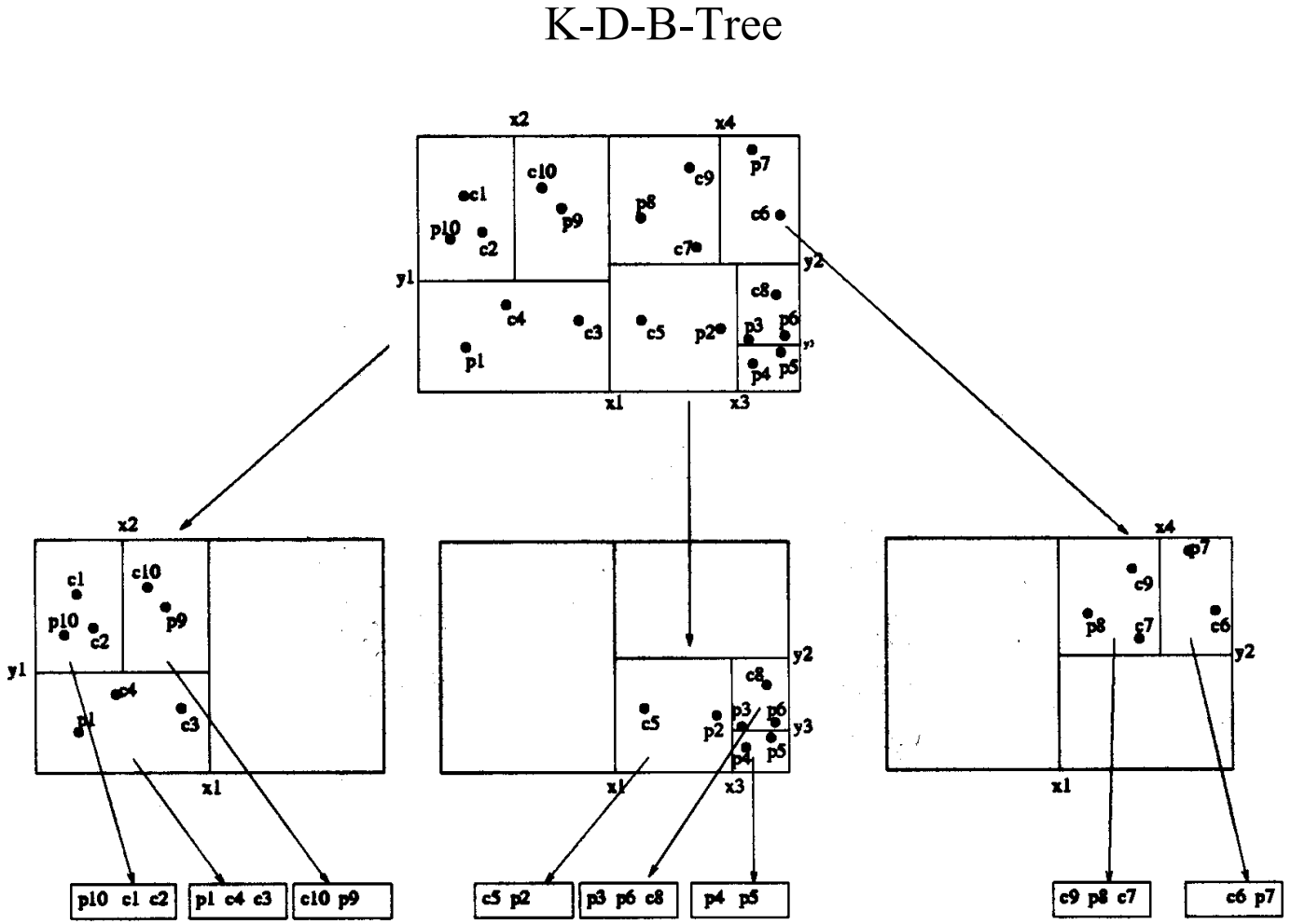
Indexing hierarchies (kdB-tree)

- K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes
- The problem of retrieving multikey records via range queries from a large, dynamic index.
- By *large* it is meant that most of the index must be stored on secondary memory.
- By *dynamic* it is meant that insertions and deletions are intermixed with queries, so that the index cannot be built beforehand.
- *K-D-B-tree* is presented as a solution to this problem.

Indexing hierarchies (kdB-tree)

- K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes
- K-D-B-trees combine properties of K-D-trees and B-trees.
- It is expected that the multidimensional search efficiency of balanced K-D-trees and the I/O efficiency of B-trees should both be approximated in the K-D-B-tree.

Indexing by subspaces (kdB-tree)



Indexing hierarchies (kdB-tree)

- The K-D-B-tree contains two types of pages:
- Region pages: A collection of *(region, child)* pairs containing a description of the bounding region along with a pointer to the child page corresponding to that region.
- Point pages: A collection of *(point, location)* pairs. In the case of databases, *location* may point to the index of the database record, while for points in *k*-dimensional space, it can be seen as the point's coordinates in that space.

Indexing hierarchies (kdB-tree)

- Page overflows occur when inserting an element into a K-D-B-tree results in the size of a node exceeding its optimal size.
- Since the purpose of the K-D-B-tree is to optimize external memory accesses like those from a hard-disk, a page is considered to have overflowed or be overfilled if the size of the node exceeds the external memory page size.

Indexing hierarchies (kdB-tree)

- Throughout insertion/deletion operations, the K-D-B-tree maintains a certain set of properties:
- The graph is a multi-way tree. Region pages always point to child pages, and can not be empty. Point pages are the leaf nodes of the tree.
- Like a B-tree, the path length to the leaves of the tree is the same for all queries.
- The regions that make up a region page are disjoint.
- If the root is a region page the union of its regions is the entire search space.

Indexing hierarchies (kdB-tree)

- When the *child* of a *(region, child)* pair in a region page is also a region page, the union of all the regions in the child is *region*.
- Conversely in the case above, if *child* is a point page, all points in *child* must be contained in *region*.
- **BKD tree:** it was proposed as a means to provide the fast queries and near 100% space utilization of a static K-D-B-tree. Instead of maintaining a single tree and re-balancing, a set of $\log_2 (N / M)$ K-D-B-trees are maintained and rebuilt at regular intervals. In this case, M is the size of the memory buffer in number of points.