Data Structures for Computer Graphics

Point Based Representations and Data Structures

Lectured by Vlastimil Havran

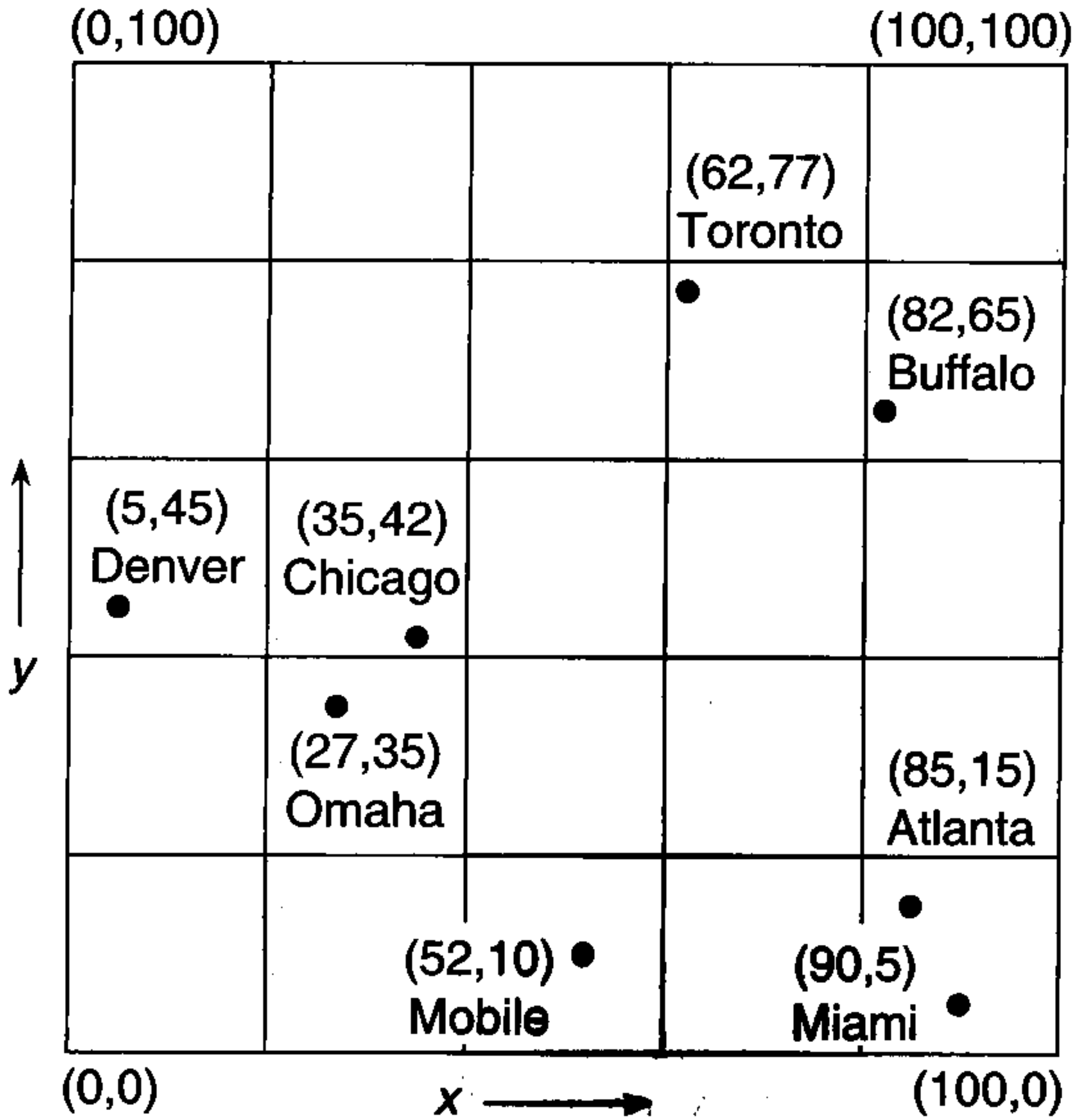
Point Based Data

- Data represented by points in multidimensional space (at least 2D)
- Many problems can be converted to point based representation, possibly in high dimensional space
- Data structures for points are usually spatial subdivisions:
  - Non-overlapping spatial cells
  - Each data point is only once in the data structures

Data Structures Overview

- Uniform grid
- Point quadtree
- Pseudo-quadtree
- TRIE based data structures
  - MX-tree
  - PR-quadtree
- Point Kd-tree
- Adaptive point Kd-tree
- BSP tree
- D-tree

Uniform Grid Representations (regular data structures)



Uniform Grid Representations

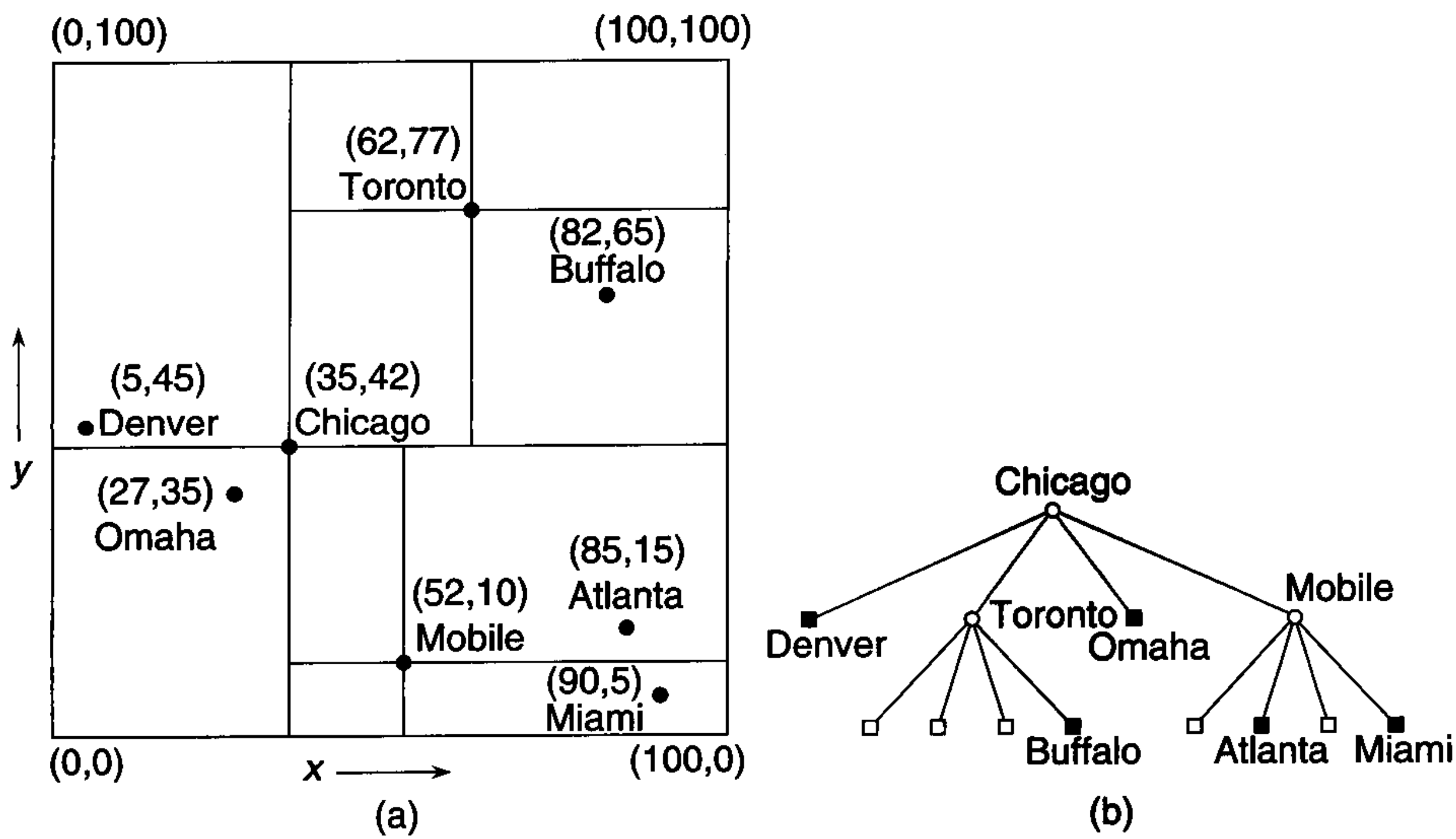
- Cell is directly addressed.
- In each cell index to the list of points
- Each point – two coordinates in 2D
- How to solve data collisions?

Note A: it can also be used for 3D/4D, but memory increase makes it difficult to use

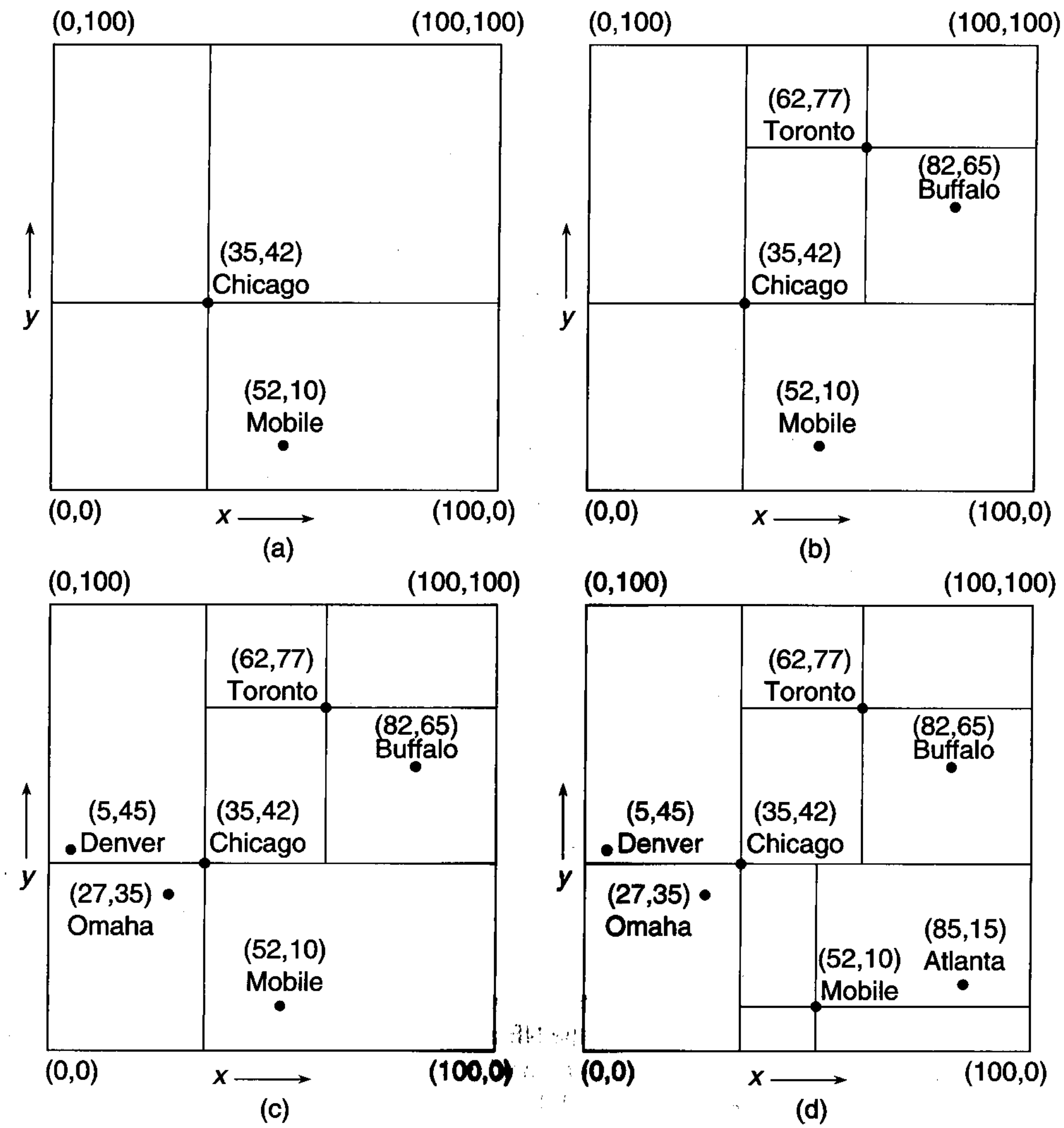
Note B: efficient for relatively uniform distribution of points, in particular in 2D

Point Quadtree in 2D

- Partitioning planes aligned with data points
- Leaves may also contain data points



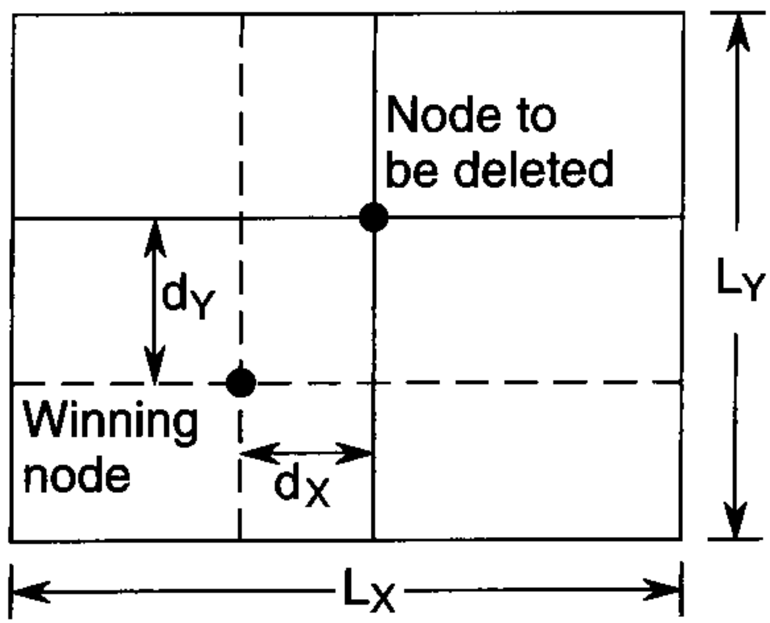
Point Quadtree Insertion



Insertion order:
1) Chicago
2) Mobile
3) Buffalo
4) Toronto
5) Omaha
6) Denver
7) Atlanta

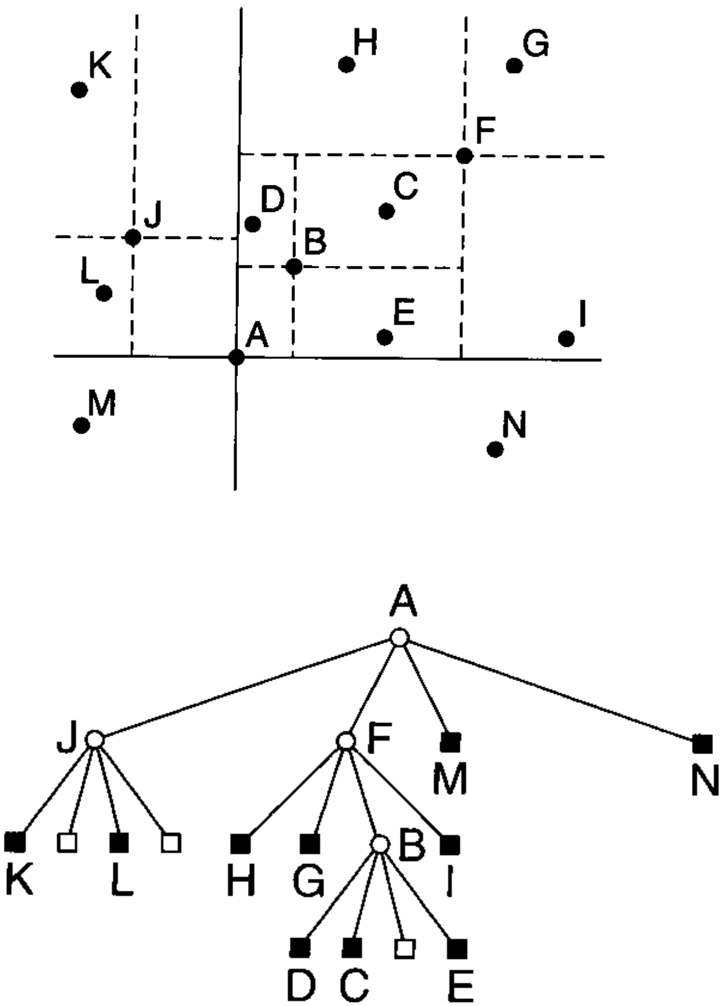
Point Quadtree Deletion

- Reconstructing the whole subtree rooted at node that contains the deleted point
- Candidate selection for new root in the area based on L1 (Manhattan) metric

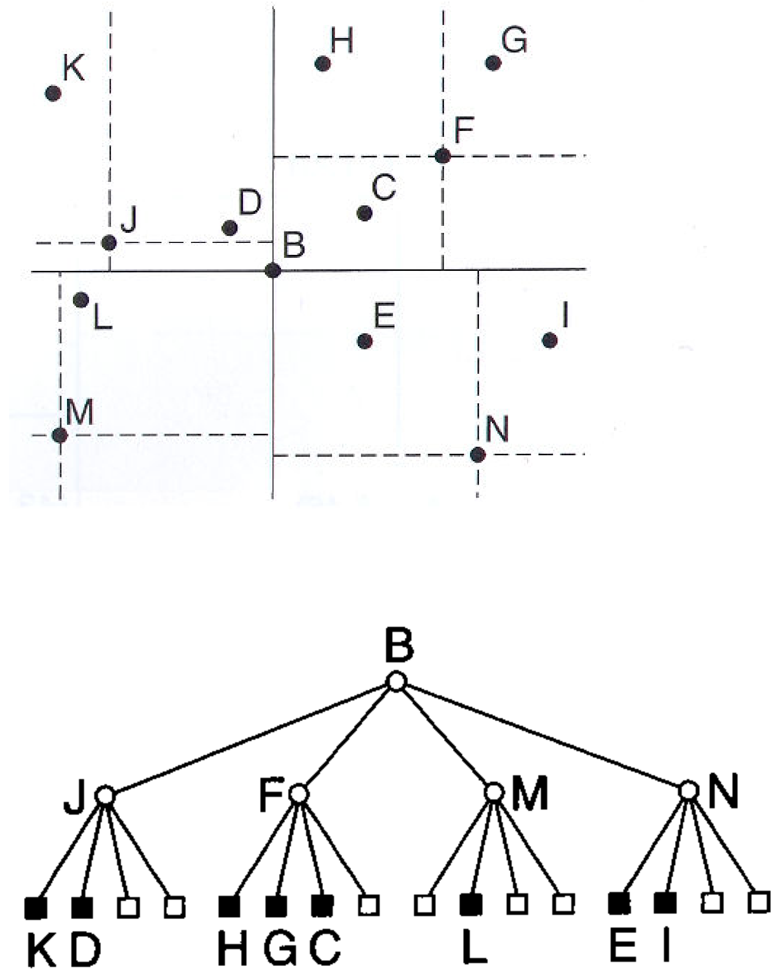


Point Quadtree Deletion (point A)

Before Deletion



After Deletion

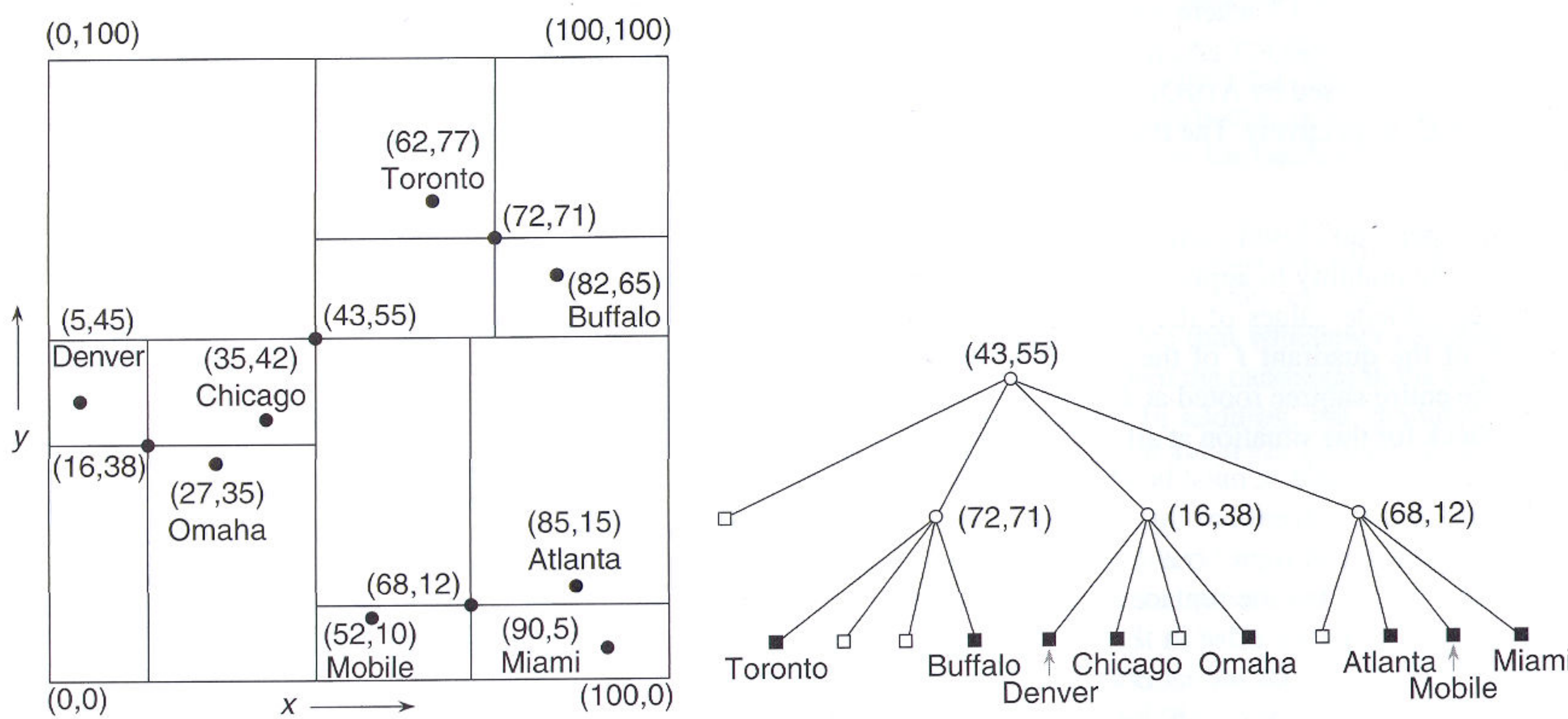


Point Quadtree Operations Overview

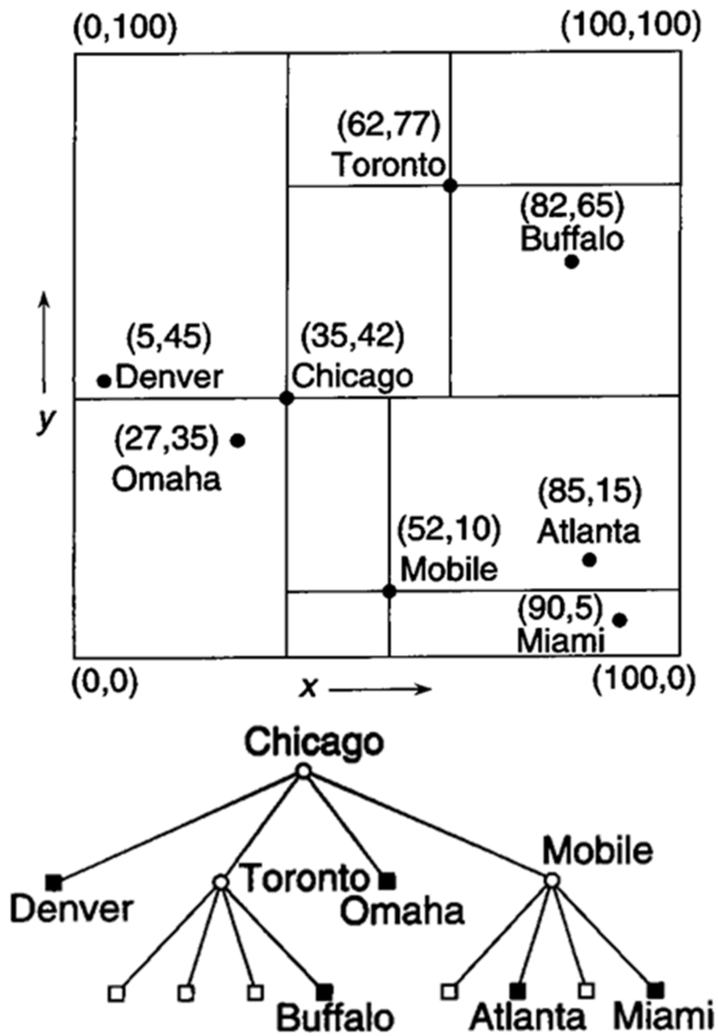
- Insertion – subdividing the cell containing more than one point
- Deletion of point A
  - Find a new candidate for splitting
  - Identify, which subtrees rooted in A are affected by deleting A
  - Identify all the points in the changed subtrees
  - Find a new candidate for splitting
  - Rebuild the tree

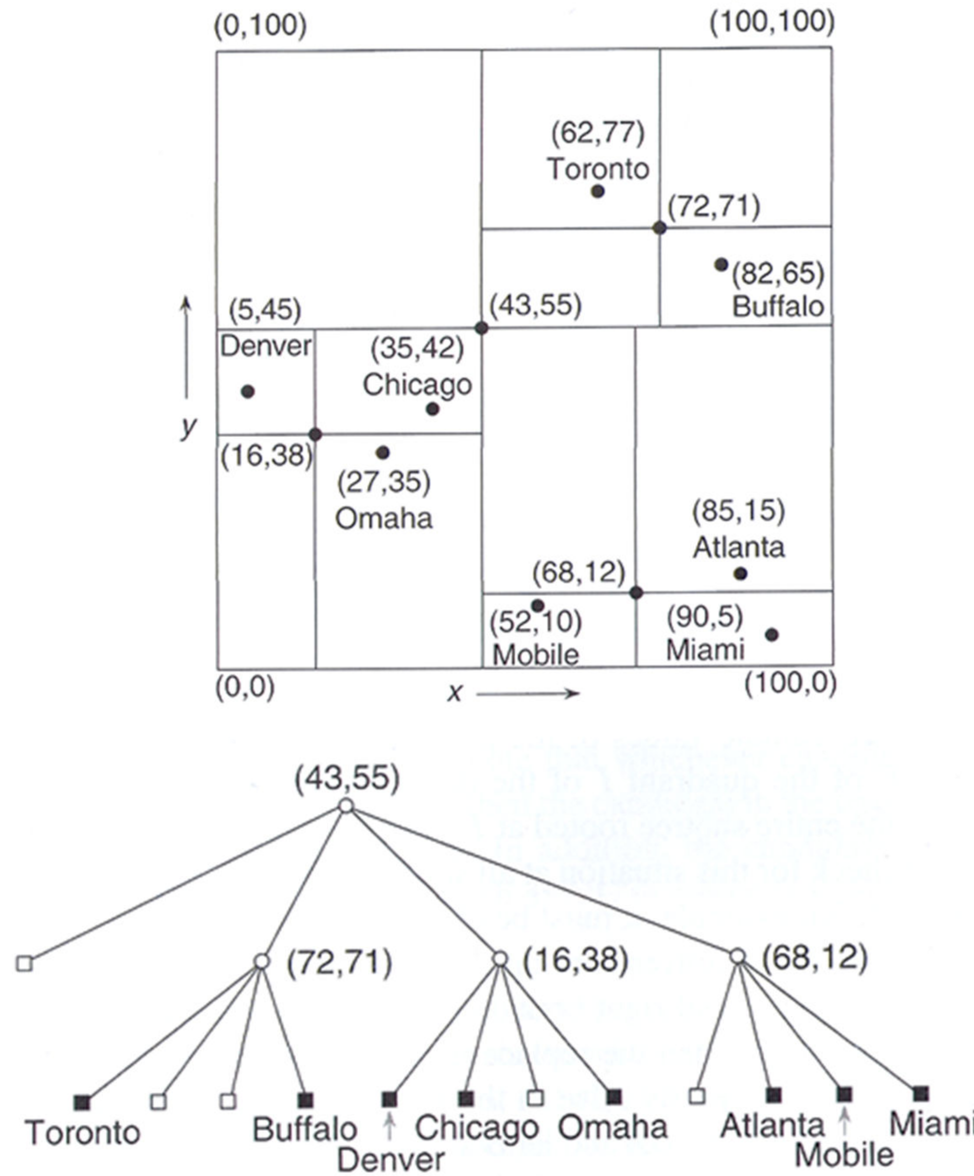
Pseudo-Quadtree

- Partitioning planes are not aligned with the data
- Data are only in leaves
- Insertion and in particular deletion is faster
- Higher space requirements.



Point Quadtrees *versus* Pseudo Quadtrees

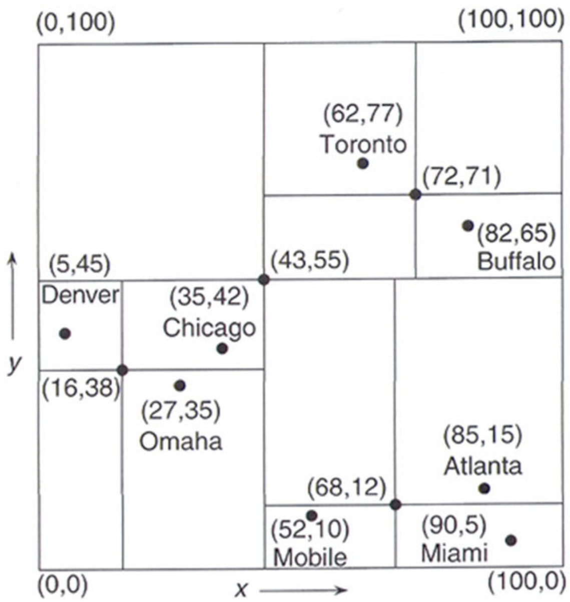


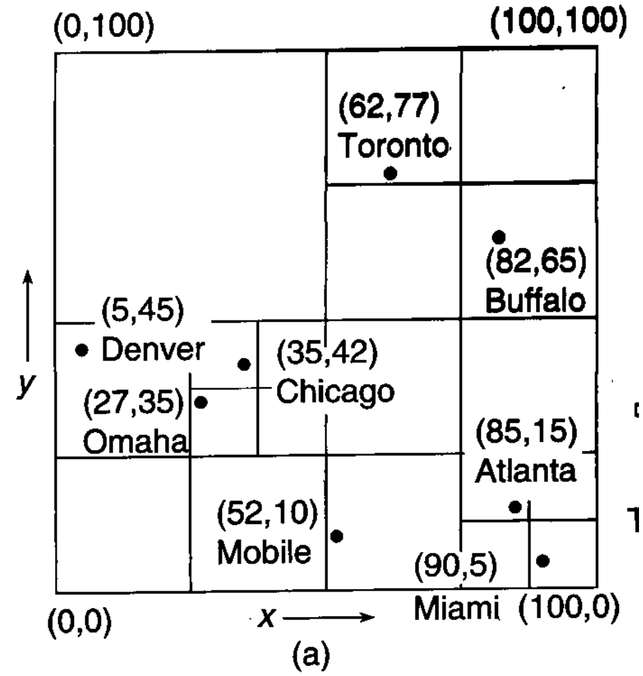


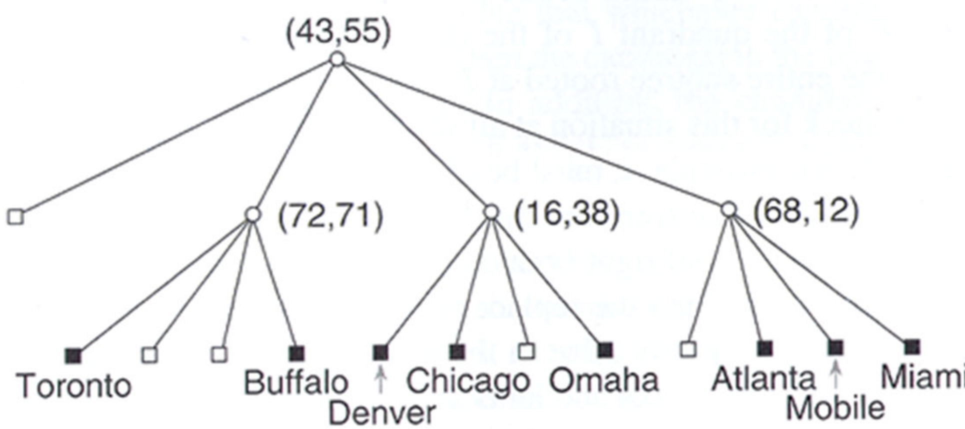
TRIE based Quadtrees

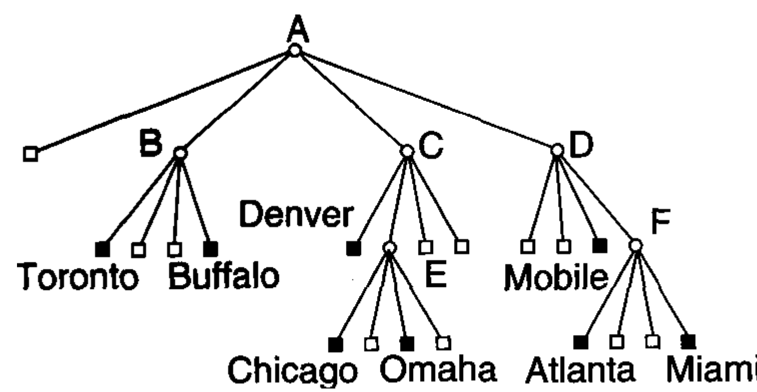
- TRIE
  - No data (points, positions) in interior nodes
  - Discretization in shape of the tree
  - Partitioning planes located exactly in the "middle" (spatial median subdivision)
- Either the data are also aligned with the discretization (MX-quadtree) or are specified exactly in leaves (PR-quadtree)
- Due to the discretization
  - Space complexity is lower
  - Dealing with non-uniform distribution is worse than for pseudo-quadtree

Pseudo Quadtrees *versus* PR-Quadtrees

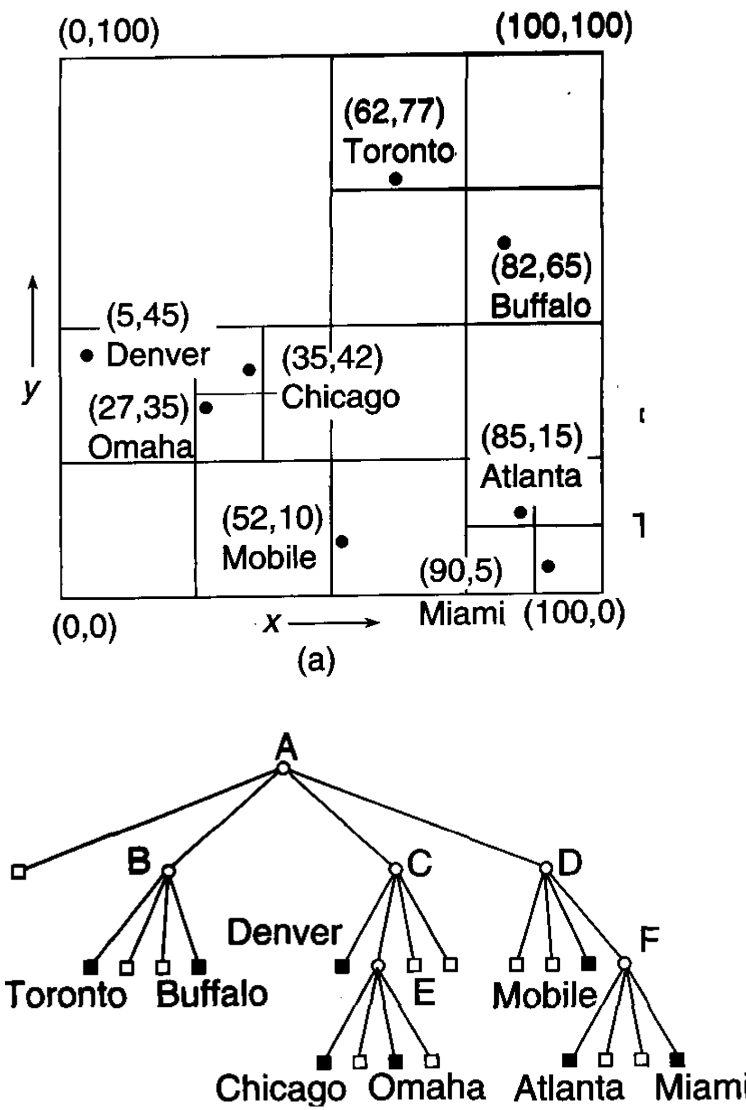


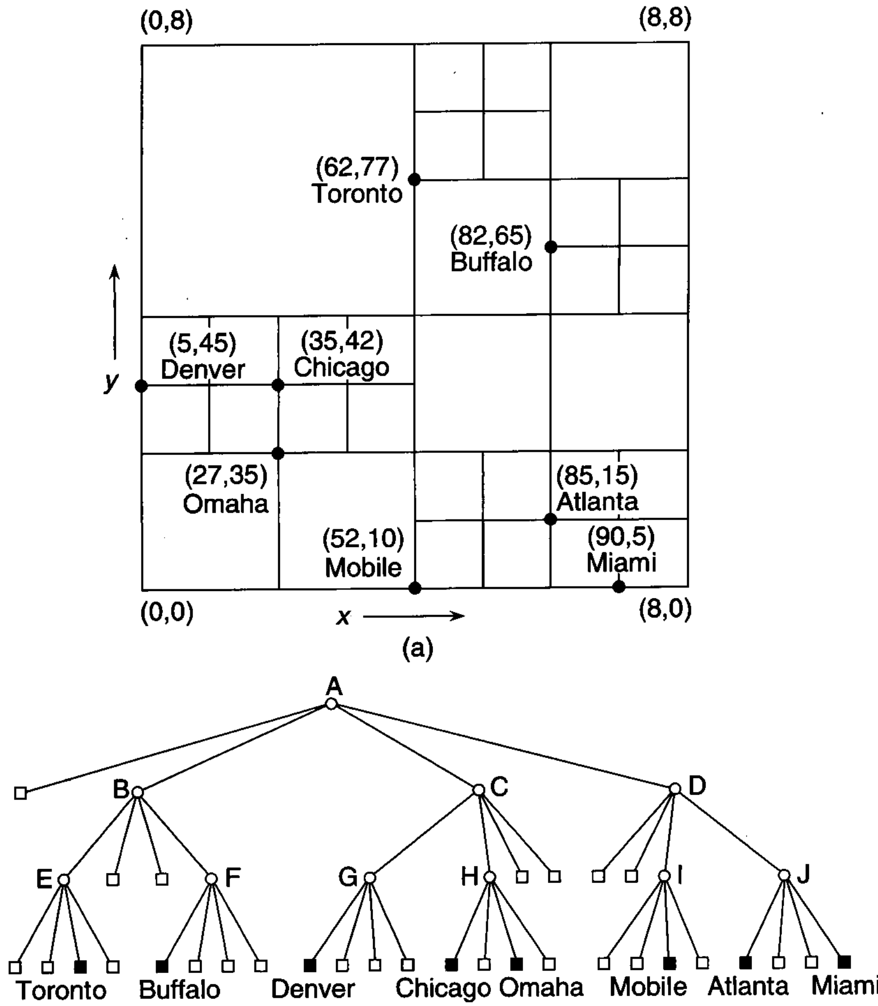






PR-quadtrees *versus* **MX-trees**





Octrees

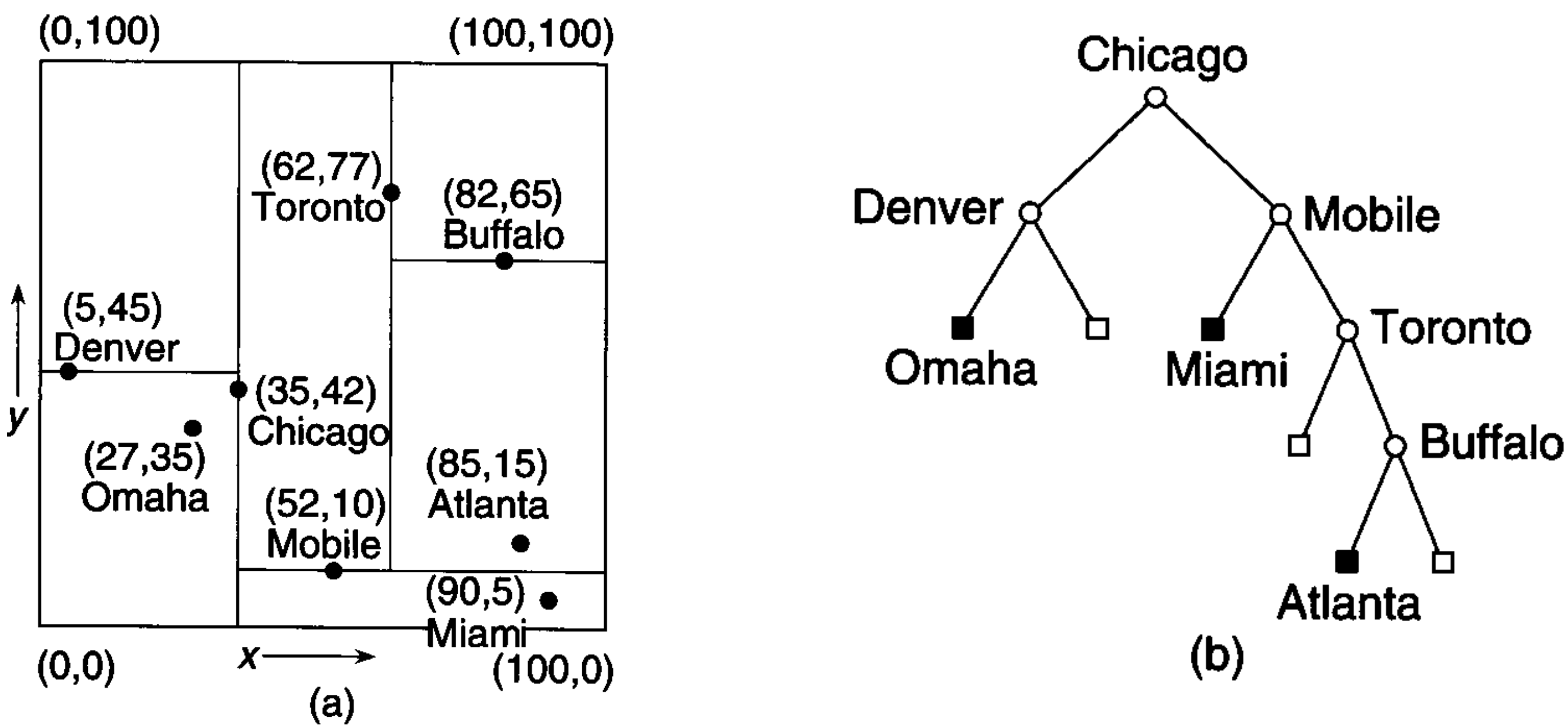
- Octree has the same principles as quadtree but the different dimension.
- Quadtree organizes 2D space (=plane, axis x and y), each interior node has 4 children
- Octree organizes 3D space (axis x, y, and z), each interior node has **8 children**

Kd-trees Introduction (year 1980)

- “K” in *Kd-trees* originally stands for the dimensionality of the tree
- The arity (also called the branching factor = number of child nodes) is always two irrespective of dimensionality of the data points
- Kd-trees correspond the most closely to 1D binary trees, but the interior nodes are interpreted differently (in multidimensional space)
- Kd-trees are very efficient and for many algorithms you can stay with kd-trees only

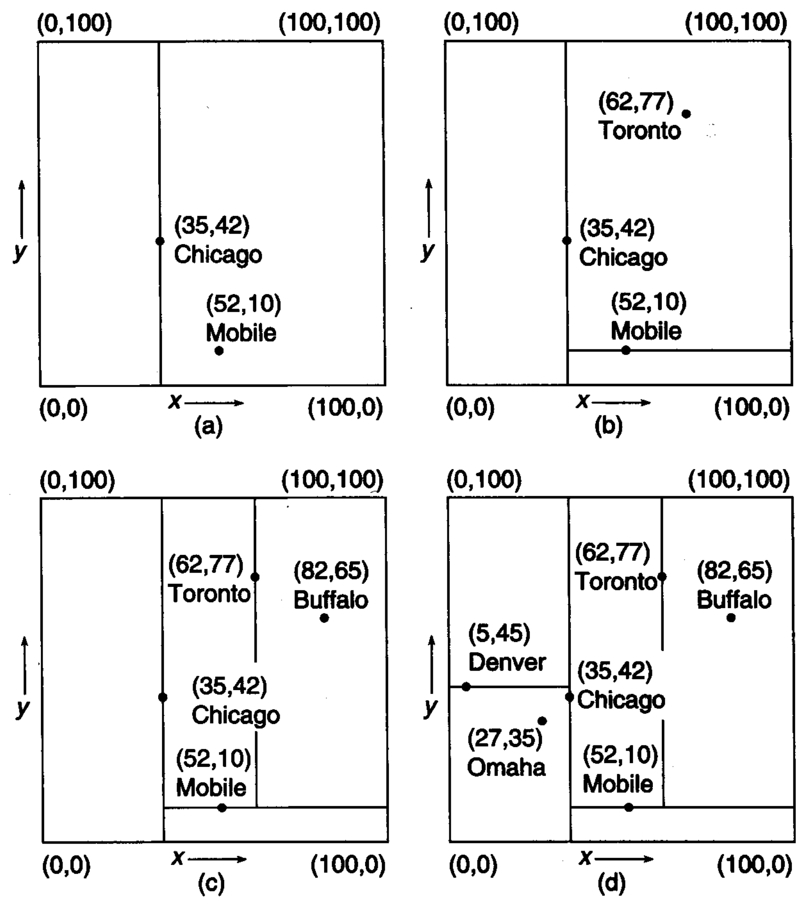
Point Kd-trees

- Organization of points similar to point quadtrees
- Partitioning planes are aligned with the data – the data are also in the interior nodes



Incrementally Building Up Kd-tree

- Assumption: taking the points in random order
- Time complexity is then O(N log N).



Insertion and Deletion in Point Kd-trees

(Analogy to *Point Quadtrees*)

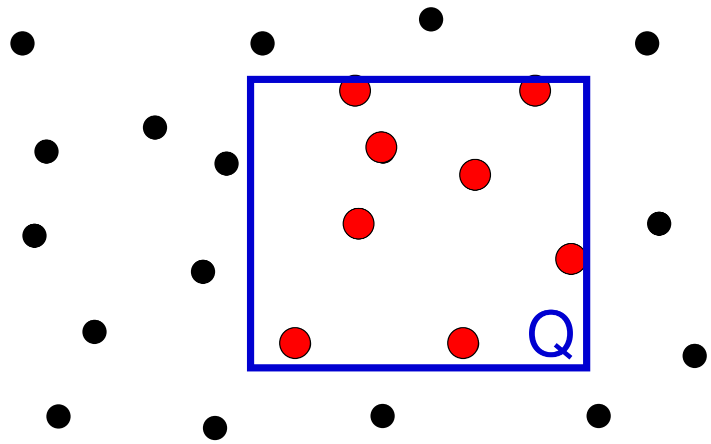
- Insertion
  - Simple
  - Locate a leaf and insert new partitioning plane
  - Assuming random order of points, cost is O(log N)

- Deletion
  - More difficult than insertion
  - Requires location an interior node N and reconstruction of the whole subtree rooted at N
  - Expected deletion cost is O(log N), as most of the data are near leaves

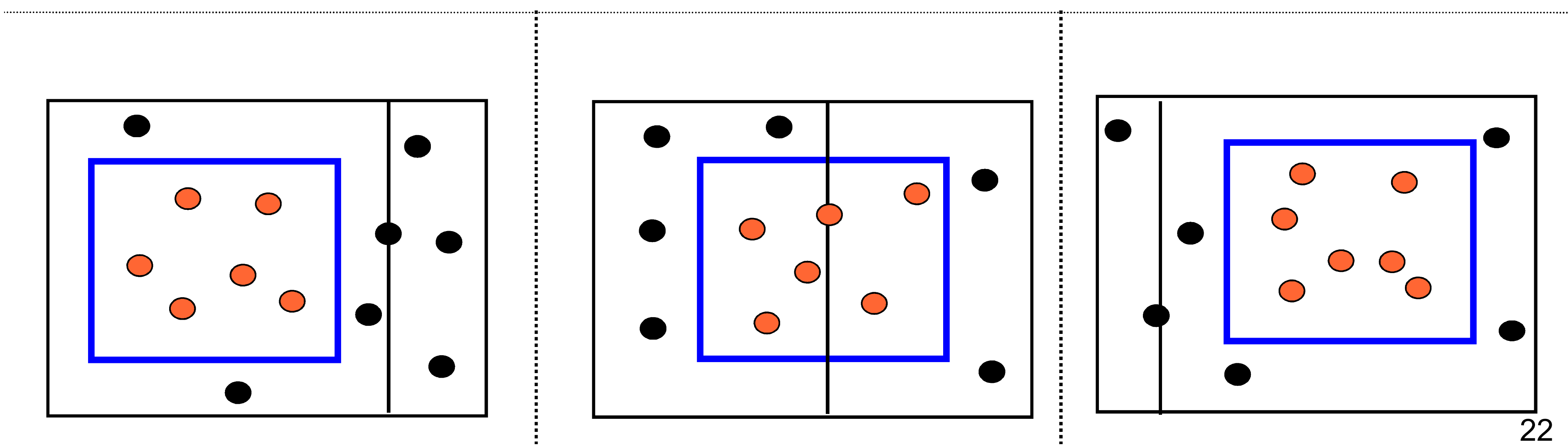
Example Application: Range Search

- A range is given by a rectangle (in 2D) / box in (3D)
- Find all the points in the query rectangle
- Assumption: kd-tree is built
- Practical use:
  - location of the cities around a visitor
  - Analogy in databases: find out all the employees with age from 30 to 50 and salary between 2,000 to 3,000 USD

Range Search with Point kd-trees



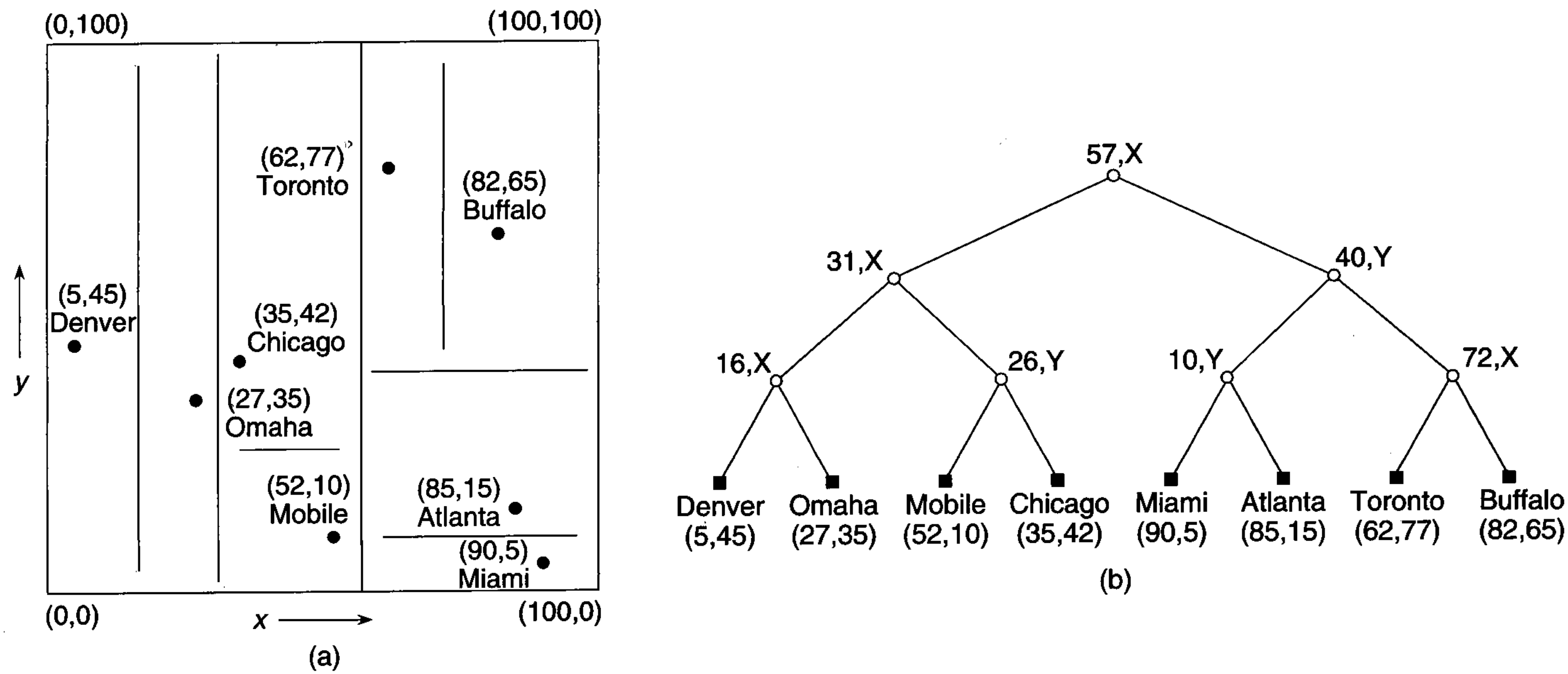
- Given a partitioning plane, there are three cases for traversal:
  - visit only left child
  - visit both children
  - visit only right child



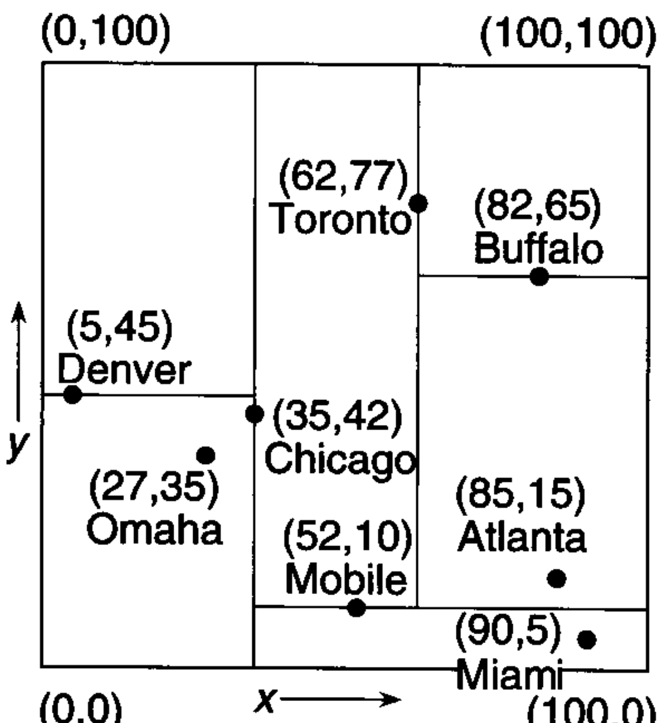
Adaptive Point Kd-trees

- It also organizes point data, similar to pseudo quadtrees
- The data are stored **only in leaves**
- The deletion is simpler
- The data storage is higher if partitioning is carried out until each leaf contains a single point
- Kd-trees enable to store more than one point in a leaf, which can be efficient for some queries (range search)

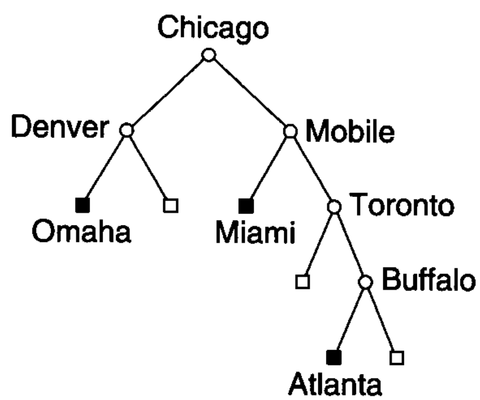
Adaptive Kd-tree Example



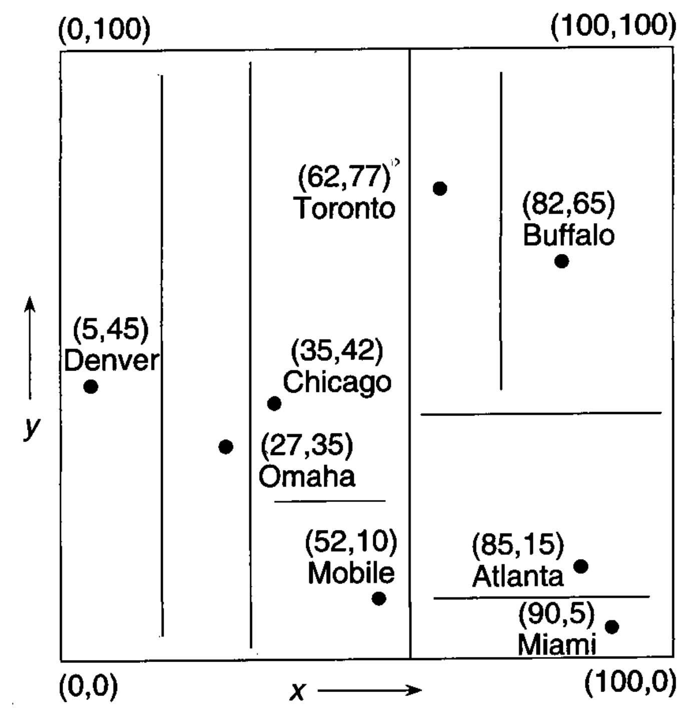
Non-adaptive *versus* Adaptive Point Kd-tree



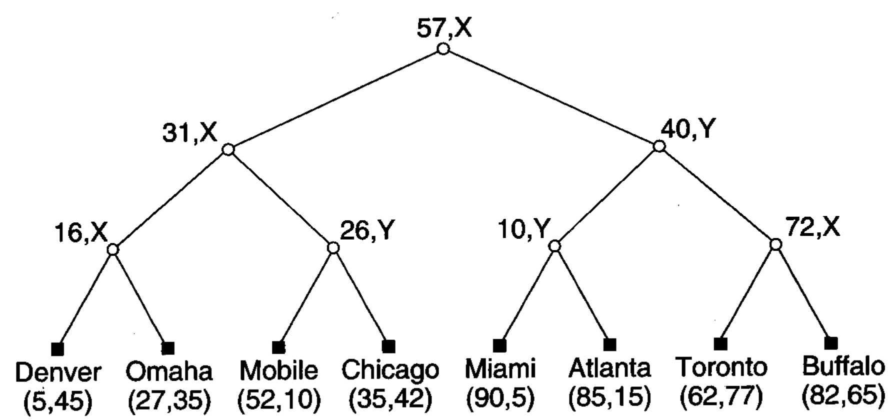
(a)



(b)



(a)



(b)

Choosing Partitioning Plane

- Geometrically in the middle – the children cells are of the same size
- By data median – half of the points on the left and half of the points on the right (if the number of points is even)
- By other means: by assumed query distribution (“the minimum-ambiguity kd-tree”), using cost function

Choosing Partitioning Axis

- We have to decide in which axis we partition a set of the data circumvented by a box in the current node
- Methods:
  - Round robin: x,y,z,x,y,….
  - Maximum extent of the box – select an axis in which box has maximum size
  - Combination: with 30% in round robin fashion based on the parent node and with 70% maximum extent of the box

Kd-tree Notes

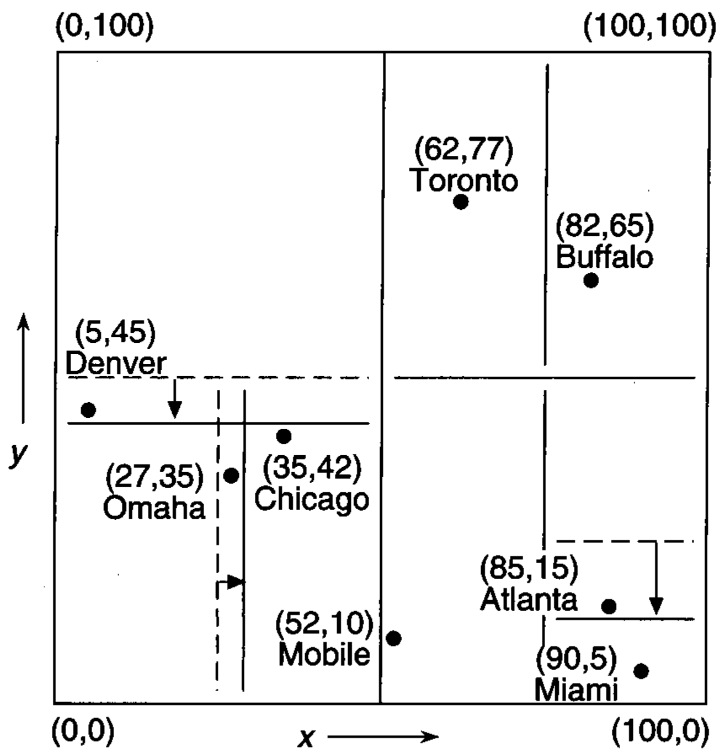
- There exist many variants: kd-tree, adaptive kd-tree, fair-split tree, VAMSplit kd-tree, minimum-ambiguity kd-tree, bucket generalized pseudo kd-tree, bucket adaptive kd-tree, PR kd-tree etc.
- The way of selection partitioning plane position and orientation is crucial for performance of the data structures
- The selection of the appropriate or the most efficient variant of kd-trees is application dependent

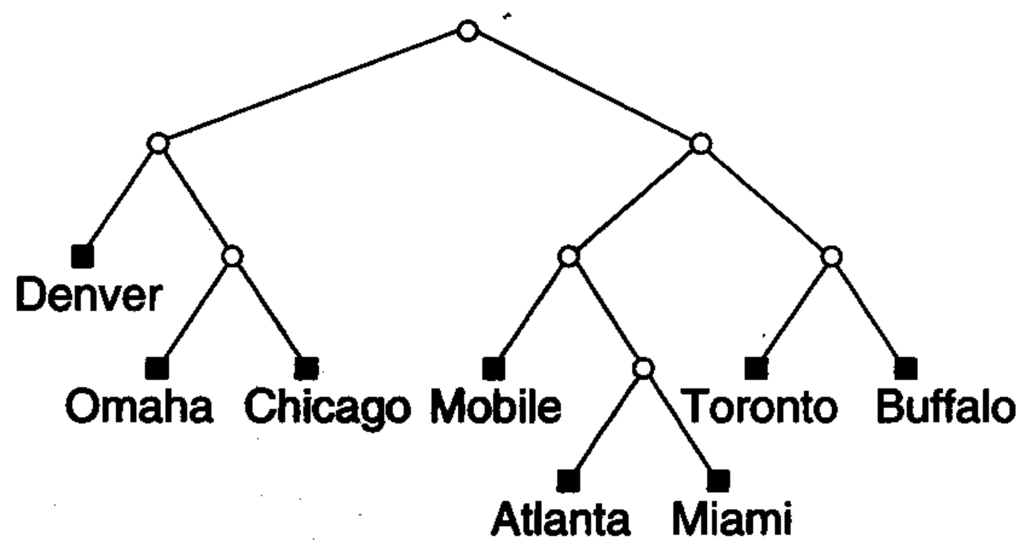
Example: Sliding-Midpoint kd-tree

- Efficient for range searching and kNN searching (in the lecture 6/7)

Sliding-Midpoint Kd-tree

- Partitioning plane in the middle:
  - Points on both side: nothing happens
  - Point on one side only: slide the partitioning plane to the nearest data point





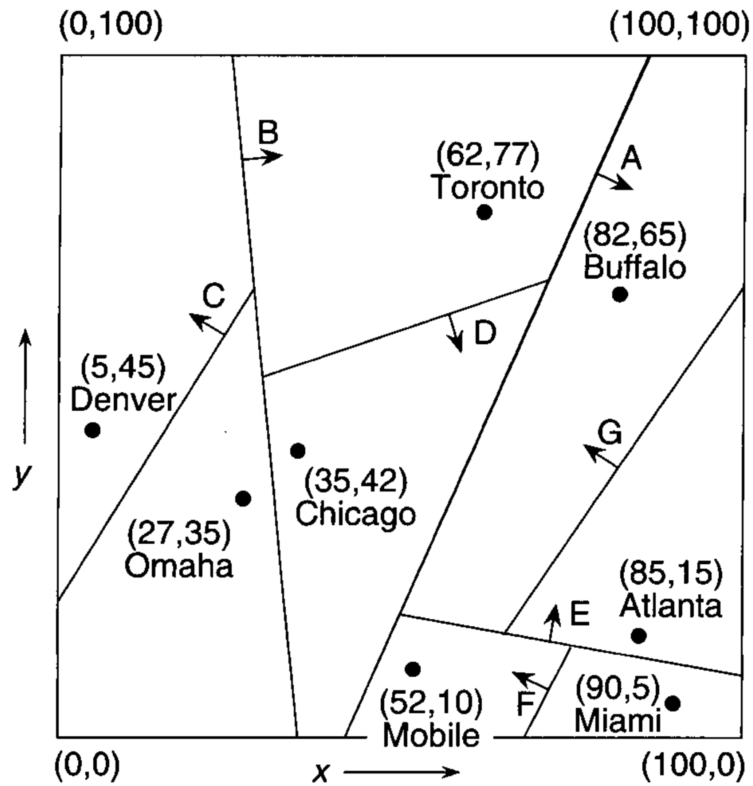
Examples of Other Data Structures

A content of the interior node can be anything !

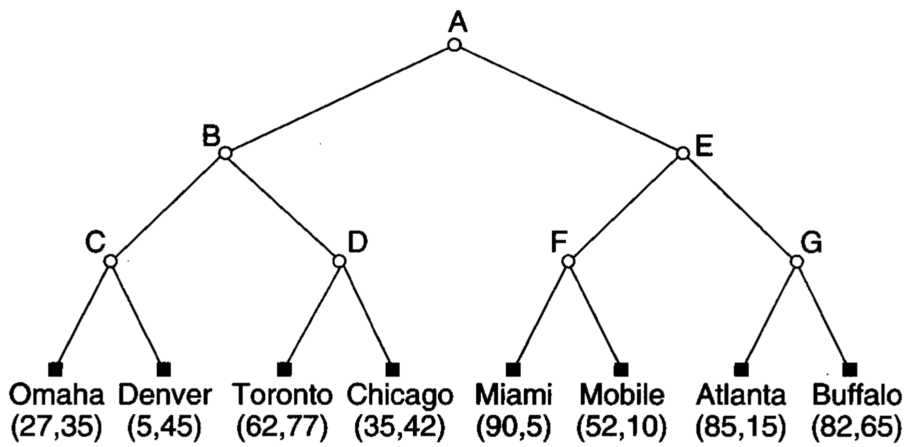
- Binary space partitioning tree (2D/3D) – general plane
- D-tree – general polyline in 2D

Binary Space Partitioning (BSP) trees

- General partitioning planes, not aligned with main coordinate system
- Note: in computer graphics often used for triangle-based scenes.



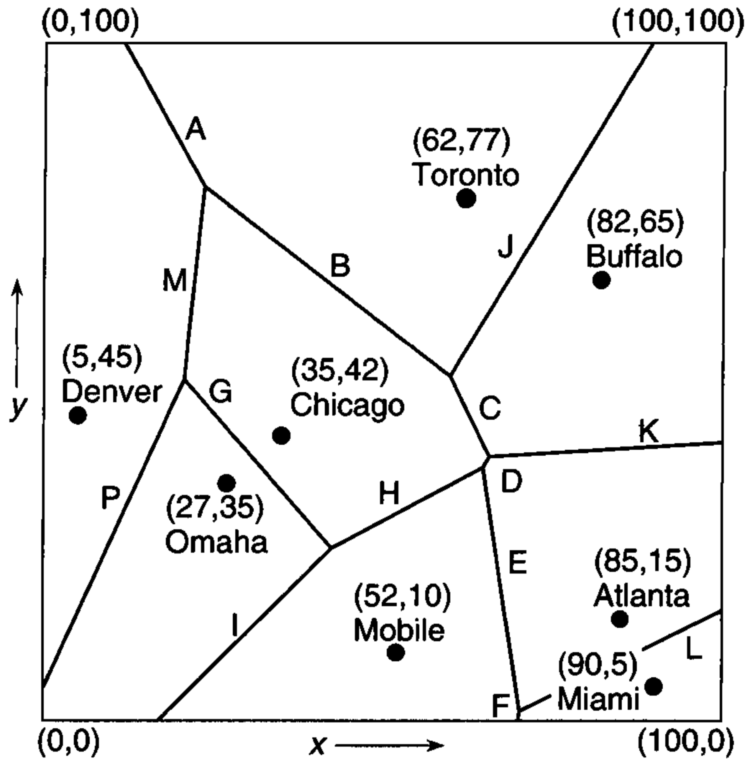
(a)

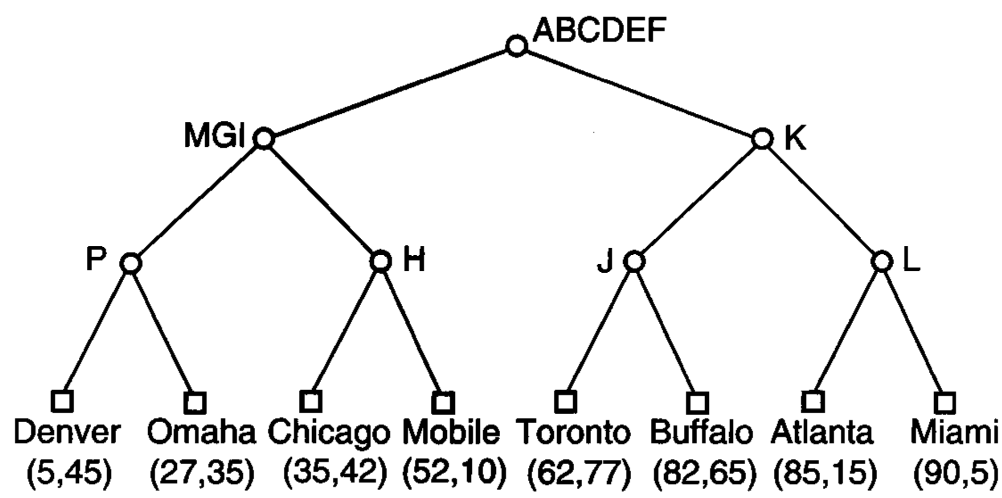


(b)

D-trees

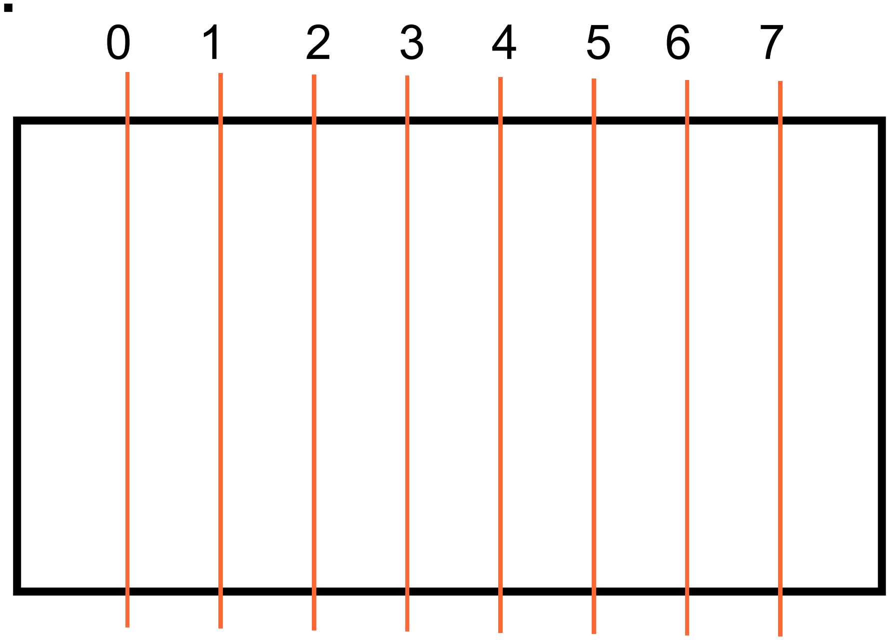
- For 2D: partitioning by polylines





Compression Methods for Point-Based Data Structures

- *Quantization of the partitioning plane* – do not store the partitioning plane by floating point representation, but say where it is in the box relatively using limited number of positions:

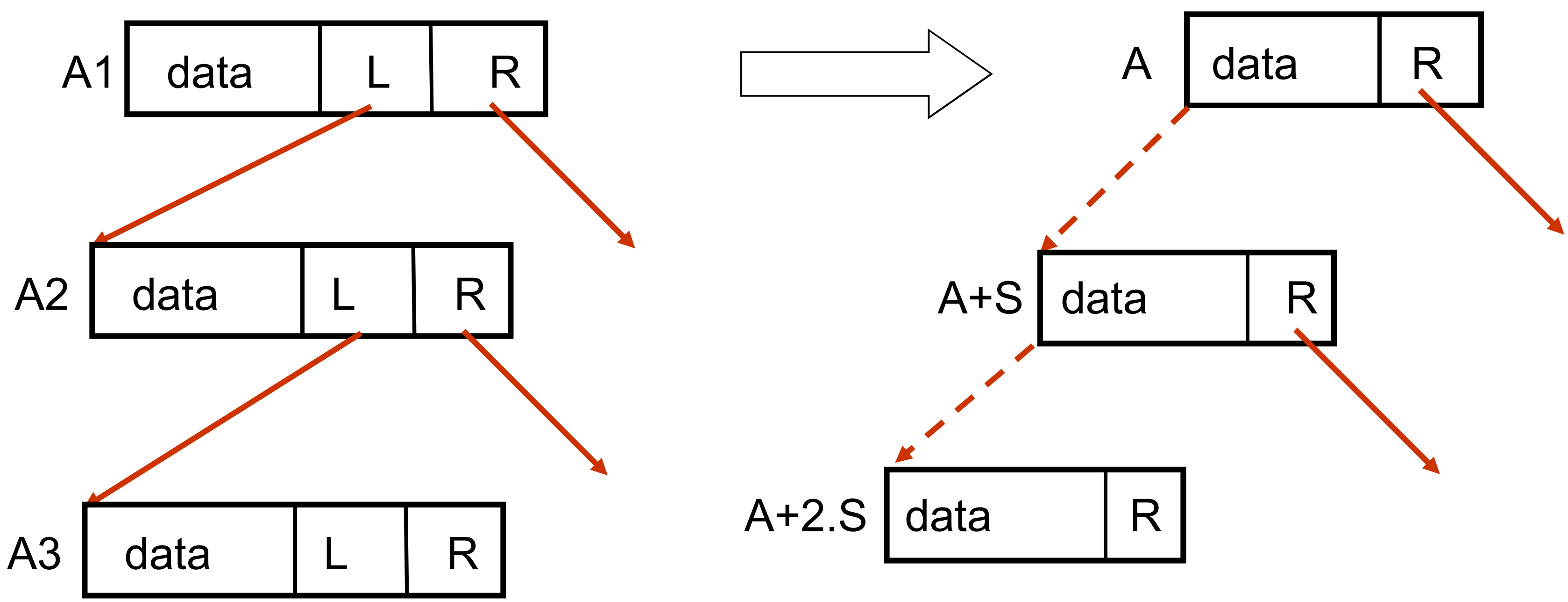


Compression Methods for Point-Based Data Structures

- Quantization of data positions (MX-trees)
- Round-robin fashion: if we use x,y,z,x,y… fashion in regular way, we may not need to store the axis orientation in the interior nodes.
- Implicit pointers: for depth-first-search (DFS) storage of interior nodes in the memory

Method of Implicit Pointers

- Saving one pointer in each interior node
- Address in memory: A1, A2, A3
- Size of the node is fixed: S bytes



Implicit pointers on 32-bit computers

- The addresses have to be aligned: A mod 8 = A
- 3 bits – type of the node (8 possibilities)
  - Leaf node
  - 7 types of interior nodes
- 29 bits as pointer to the right child
- Left child pointed explicitly by address A+8
- In 32 bits can be used to store the position of the splitting plane (for kd-trees) or several quantized positions for octrees etc.
- Interior nodes/leaves need then only 8 Bytes!
- A tree has to be constructed in DFS order

Practical Recommendations for Data Structures over Point Data

- First, use a kd-tree, with appropriate representation in the memory
  - Sliding-midpoint kd-tree
  - Implicit pointers
  - Data only in leaves – fast insertion and deletion
- If the performance or storage space of kd-trees is insufficient, try to use a different data structure suitable to the task
  - Uniform grids if data are known to be sufficiently uniform
  - Octrees and quadtrees, different versions
  - Special trees such as BSP-trees or D-trees if the data and searching algorithm fits to the task.

Thank you for your attention!