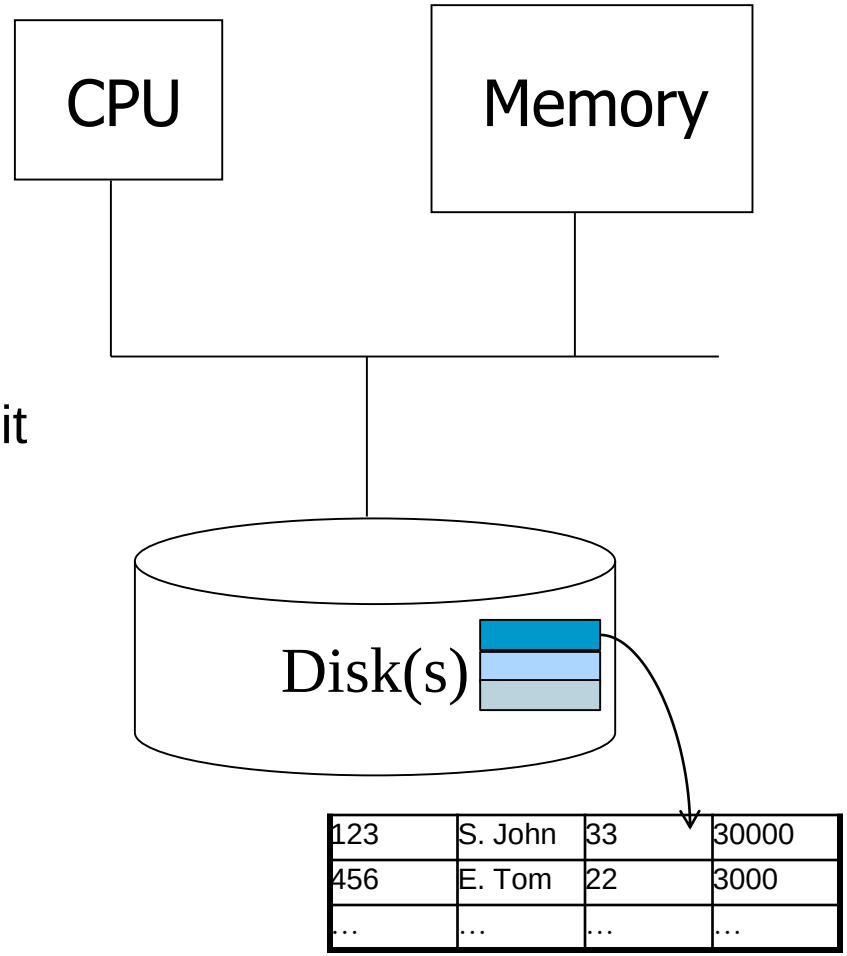
# 高级数据结构

# Advanced Data Structure

## 3. Access Methods for Multidimensional Data Retrieval 1

# Review - Primary key access methods

- Data stored on disk(s)
  - Transfer unit: block (page)
  - I/O complexity: in number of blocks accessed
- Problem
  - Minimize the number of transfer unit
  - Handle dynamic database
- Indexing
  - Loading factor
  - External memory based
  - Dynamic database
  - Easy implementation/maintenance



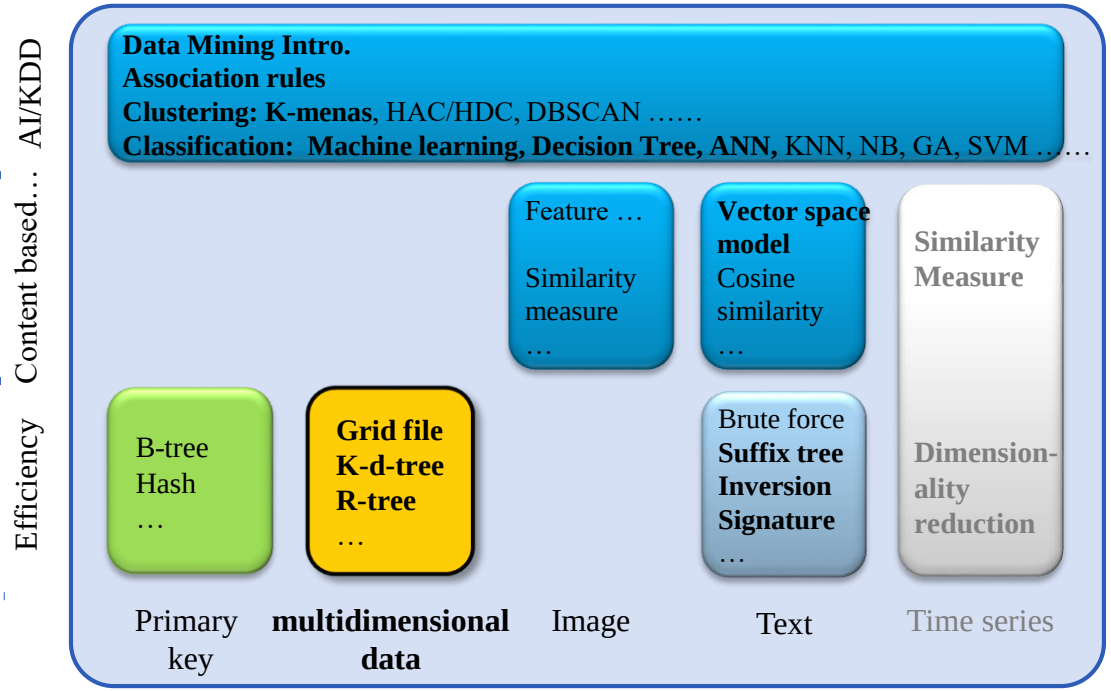
# Review - B-tree family vs. Hashing

- *Hash-based* indices are best for exact match queries. Faster than *B+-tree*!
  - *Hash-based* indices typically require 1-2 I/Os per query
  - *B*+-tree requires 4-5 I/Os (logarithmic)
- *B family tree* support answering
  - *range queries*
  - *nearest neighbor queries*
  - *ordered sequential scanning*...

*Hash-based* indices don't support.

# Agenda

- Problem: **Multidimensional Data Retrieval**
- Method:
  - **Grid file**
  - **K-d-tree**
  - **R-tree**
  - ……



# Agenda

- Problem: **Multidimensional Data Retrieval**
  - Point access
  - Spatial access
    - Key idea: Progressive refinement
- Method:
  - Partition space: Grid file, K-d-tree
  - Partition dataset: R-tree

  .....
- Dimensionality Curse

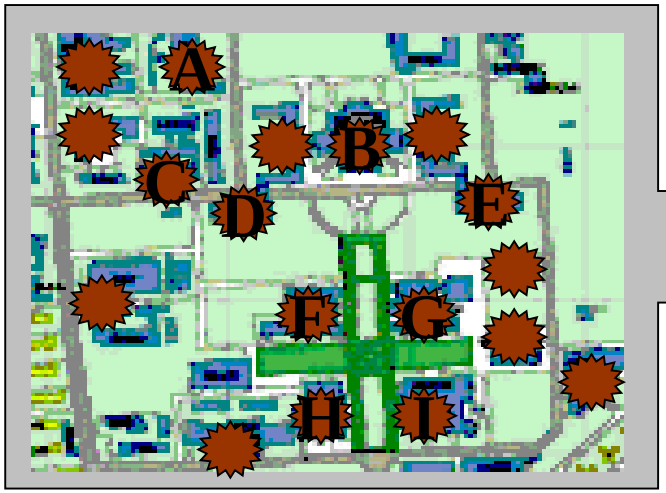
# Agenda

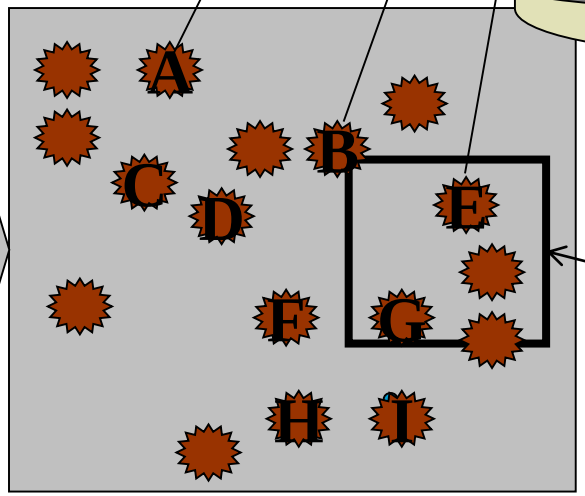
- Problem: **Multidimensional Data Retrieval**
- Method:
  - **Grid file**
  - **K-d-tree**
  - R-tree
  - ……

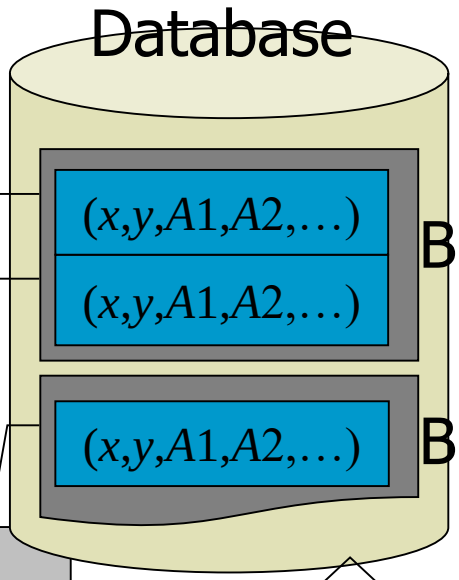
# Motivating problem 1 - Spatial data accessing

- Given a point (region) set and a query, find the points (regions) that satisfy the query

*Each building → a point/polygon/…*

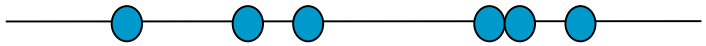


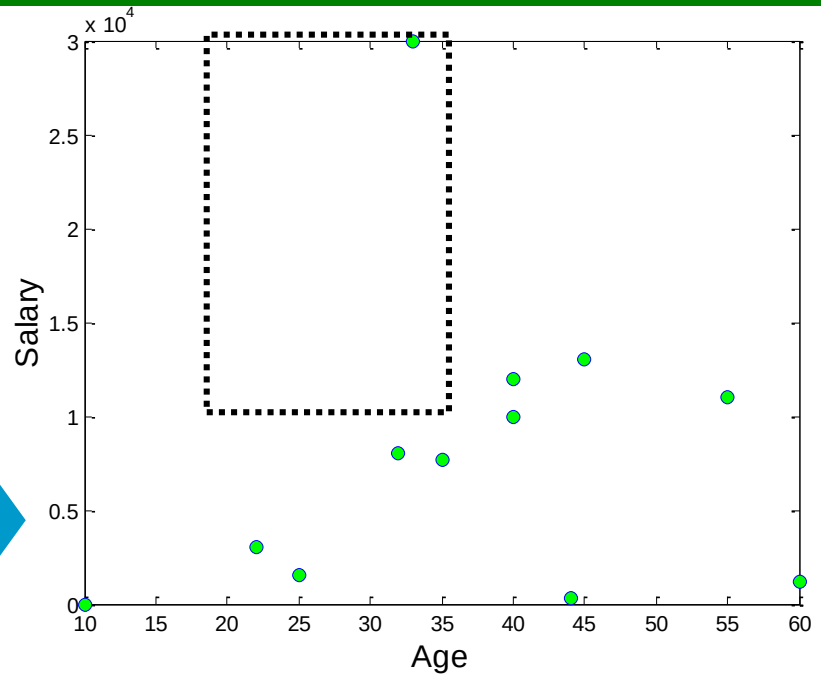




Query
e.g. (x1,y1,x2,y2)

# Motivating problem 2 - Multi-key based accessing



Age>18 and Age<35

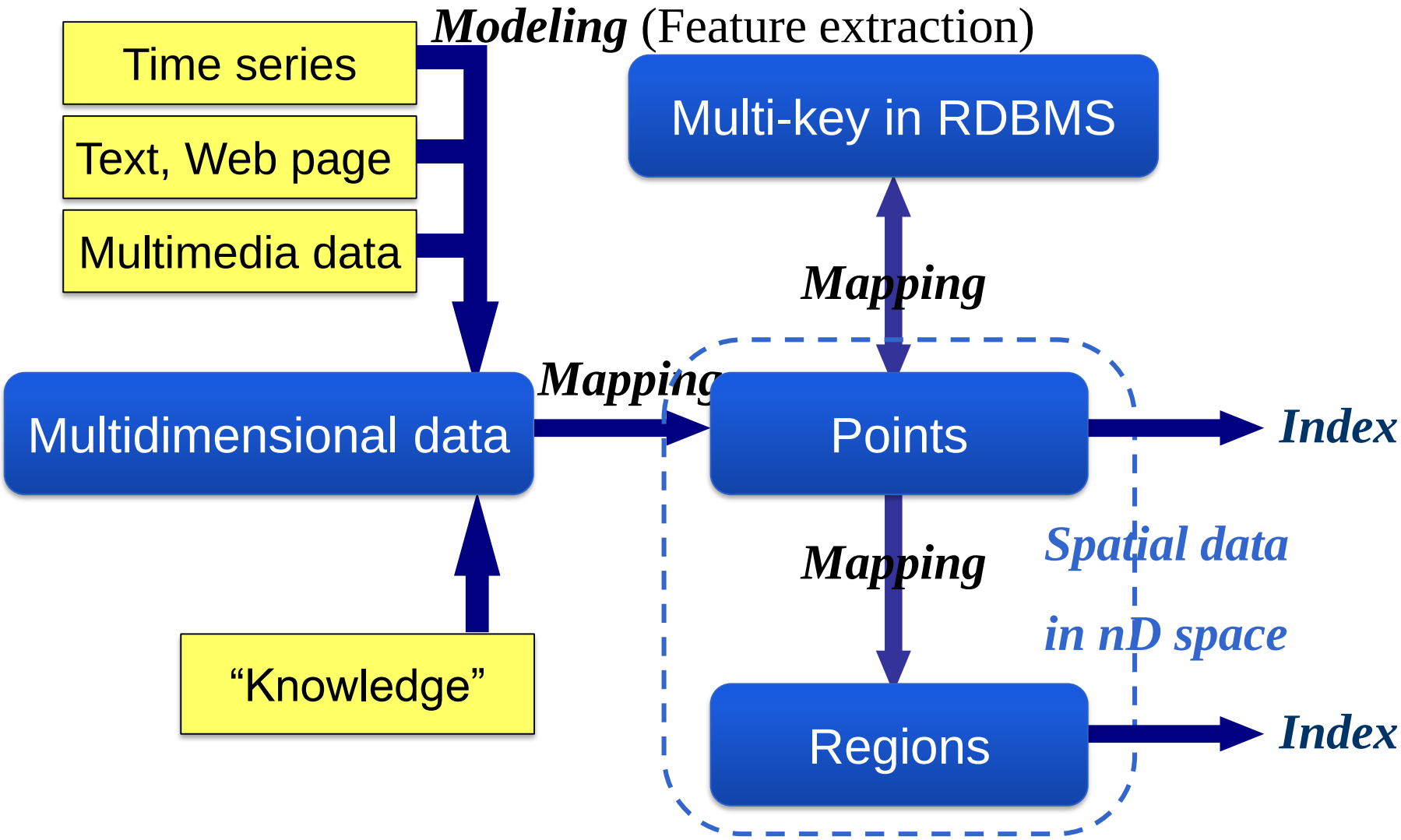



Map to 2D points (age,salary)

Age>18 and Age<35 and Salary>10000

| ID | Name | Age | Salary |
|---|---|---|---|
| 123 | S. John | 33 | 30000 |
| 456 | J. Tom | 22 | 3000 |
| … | … | … | … |

# Related topics – Searching multimedia data by content



# Queries

Given

- data set $P=\{P_1,P_2,\ldots\}$, $P_n=(x_n,y_n)$, or $P_n=(x_n,y_n,z_n)$ ,……
- Distance measurement: D($A$,$B$)

- ***Exact match***
  - Query point: Q=(x,y)
  - M(Q)={B∈P|B=Q}
- ***Nearest neighbor queries***
  - Query point: Q=(x,y)
  - NN(Q)={B∈P|∀A∈P→D(B,Q)≤D(A,Q)}
  - ***K-Nearest-neighbor queries***: find the k-nearest points to Q
- ***Range queries*** (circle region)
  - Query point: Q=(x,y), threshold $\varepsilon$
  - $RQ_C(Q)=\{B\in P|D(B, Q)\leq\varepsilon\}$

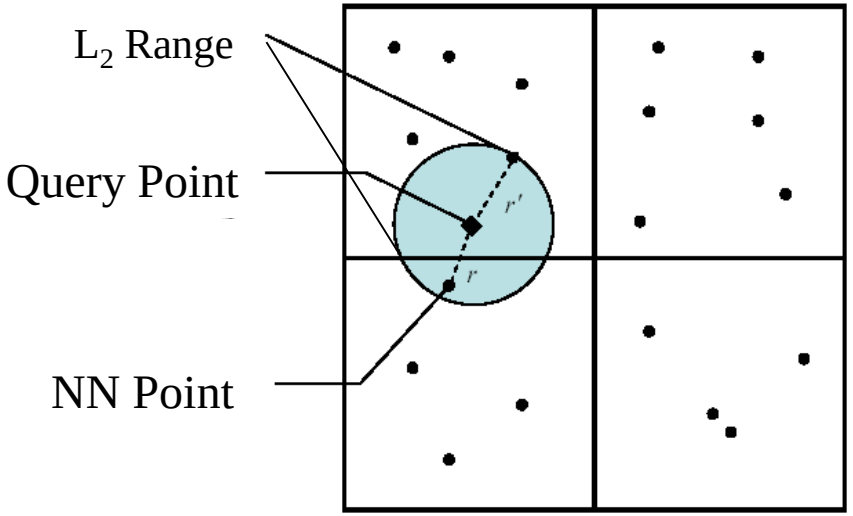
***Range queries*** (rectangle region, ***window query***)

- Query region: $R$=(x1,y1,x2,y2)
- $RQ_R(Q)=\{B\in P|B\in R\}$

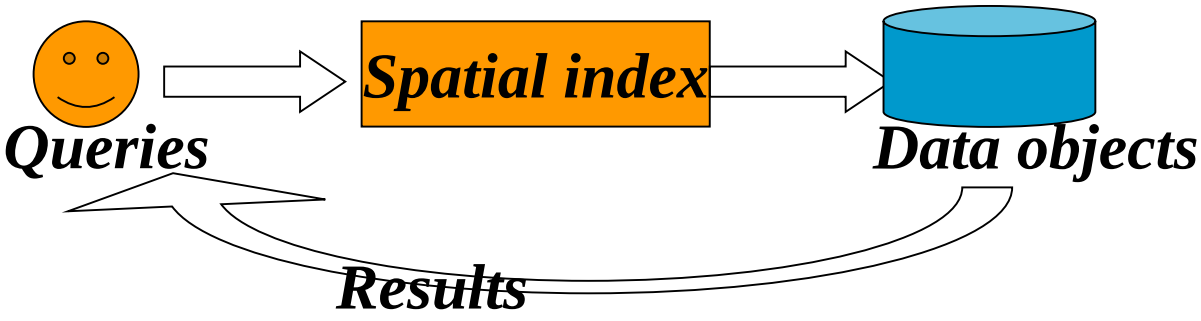
# Distance measurement

$$L_p = \left( \sum_{i=1}^{n} |A_i - B_i|^p \right)^{1/p}$$

- $L_2$: Euclidean distance
- $L_1$: Manhattan, city-block distance
- $L_\infty(A,B) = \max_i |A_i - B_i|$



# Methodologies




- Point Access Methods (PAMs): index point data
- Spatial Access Methods (SAMs): index both points and regions

# Methodologies

- Idea: divide and conquer
  - Partition the search space into subspaces, how?
- Organize/partition the **embedding space**
  - →Grid-like structure for indexing spatial data?
- Organize/partition the **specific set of data**
  - →A tree-like structure for indexing spatial data?

# Methods

- Inverted files
- Grid files
- K-d-trees
- Space filling curves
- R-trees

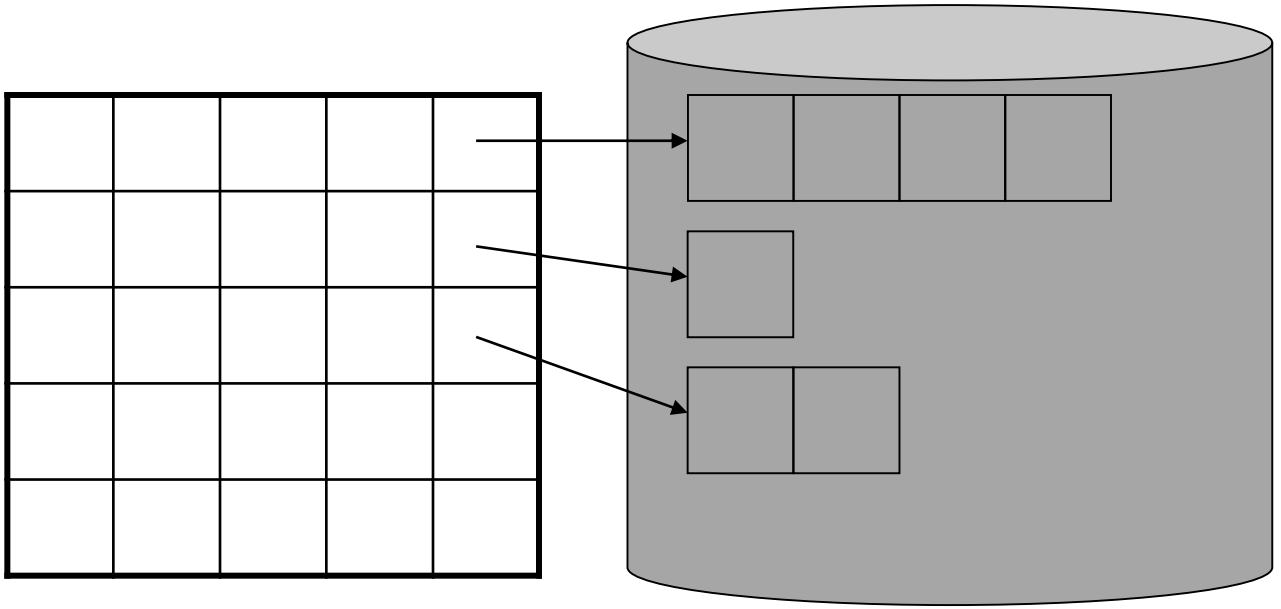
# Agenda

- Problem: **Multidimensional Data Retrieval**
- Method:
  - **Grid file:** Generalization of extendible hashing in multiple dimensions
  - **K-d-tree**
  - R-tree
  - ……

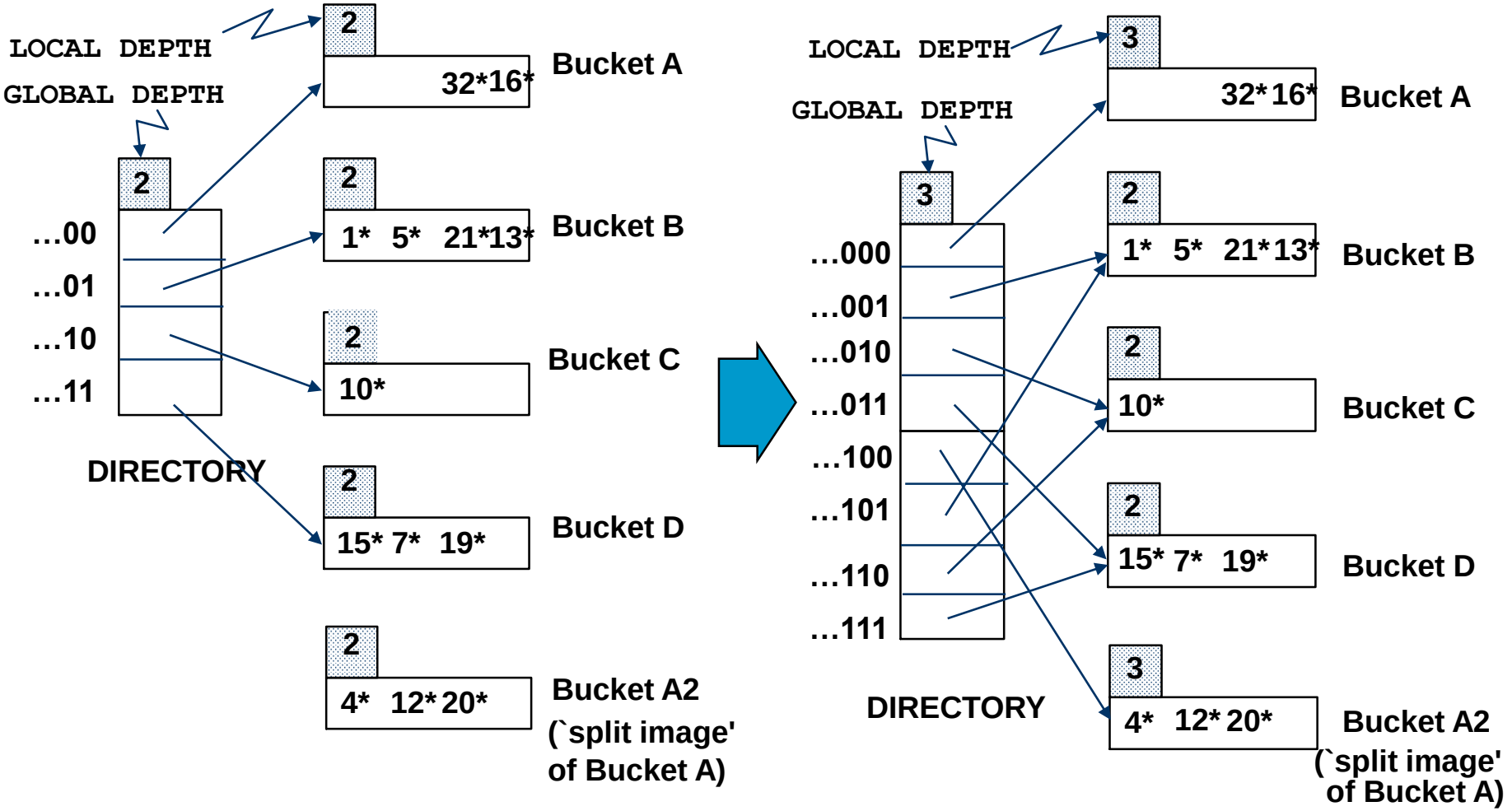
# Grid - Naïve version

- Idea: Use a grid to partition the space into cells
- All cell with fixed size
- Non-uniform distribution → Uncertain searching depth

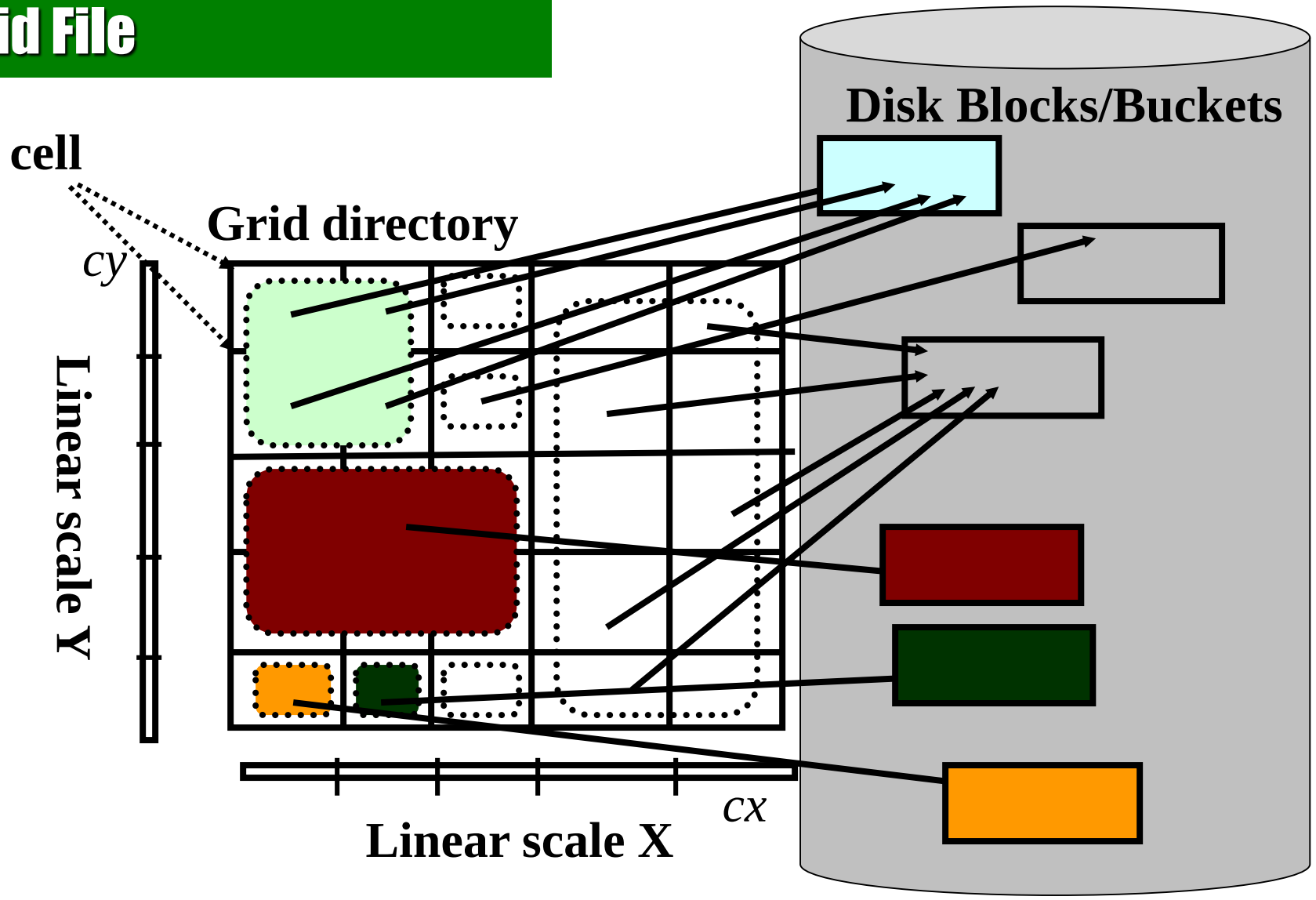
  Ideally, we want each cell associated with one page



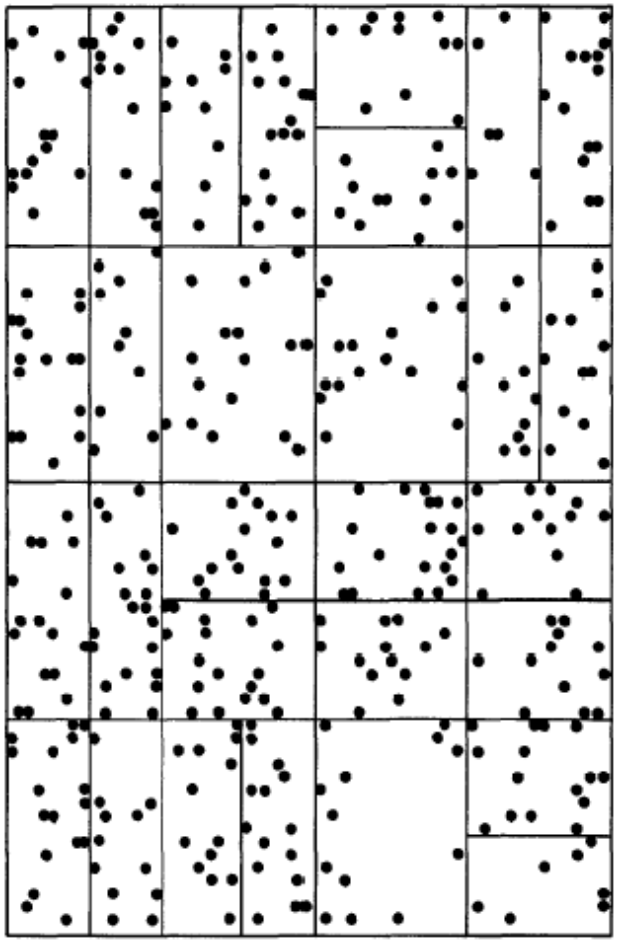
# Extendible hashing

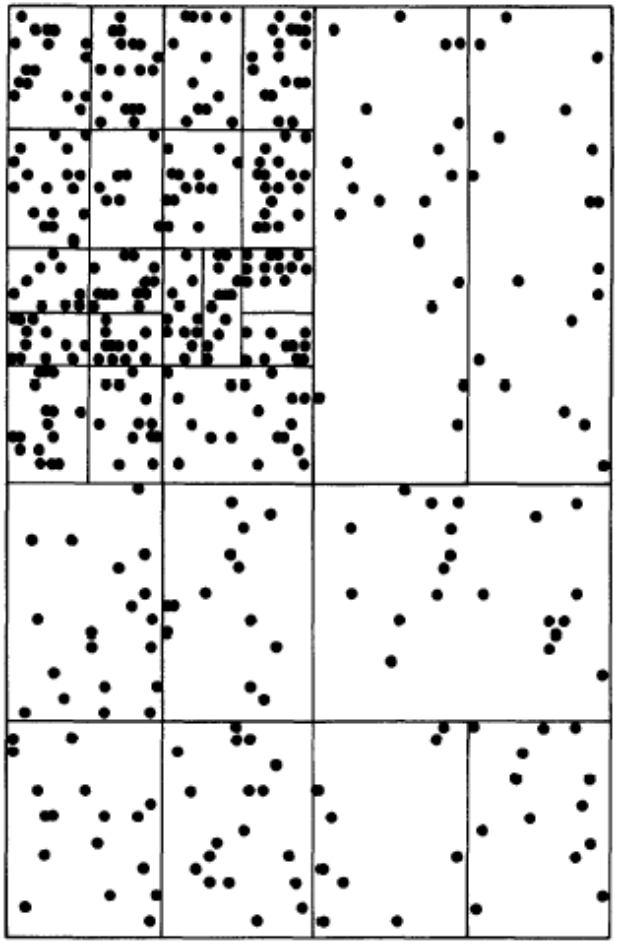


# Grid File


Disk Blocks/Buckets
cell
Grid directory
cy
Linear scale Y
cx
Linear scale X

# Grid File - Example





# Grid file structure

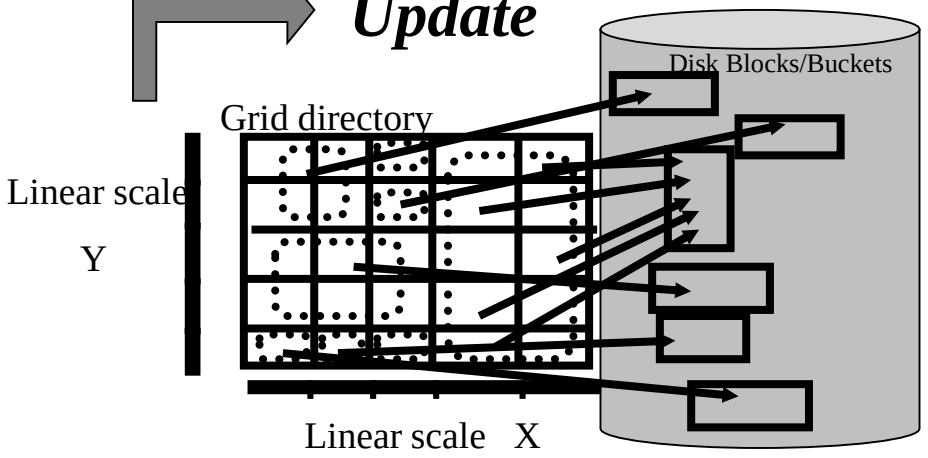
- Data stored in *disk blocks* (*buckets*)
- Dynamic structure using a grid directory
  - ***Grid array* (*directory*)**: a *2* dimensional array with pointers to buckets
    - $G(0... n_{x-1}, 0 ... n_{y-1})$
    - stored in *disk blocks or main memory*
  - ***Linear scales***: Two 1-dimensional arrays that used to access the grid array
    - $X(0 ... n_{x-1})$, $Y(0 ... n_{y-1})$
    - E.g. X=(1600,1900,1950,1980,1990,2010)
    - stored in *main memory*

# Operations on grid file

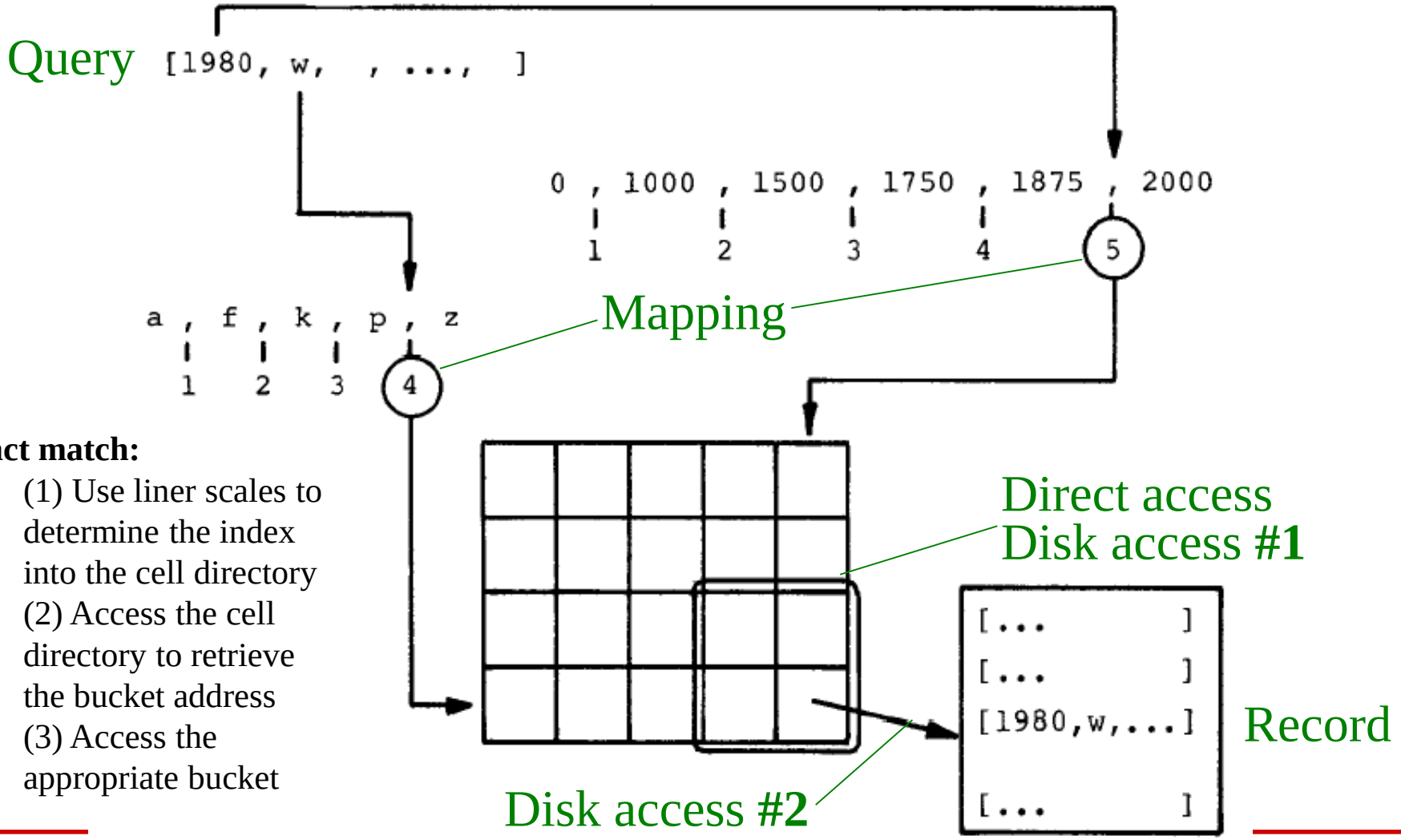
- Mapping
  - (*x*,*y*)→(*cx*,*cy*)
- Direct access
  - G(*cx*,*cy*)=*pointer to bucket*
- Adjust directory
  - Merge
  - Split

***Query processing***

***Update***



# Operations – Mapping & direct access (exact match)

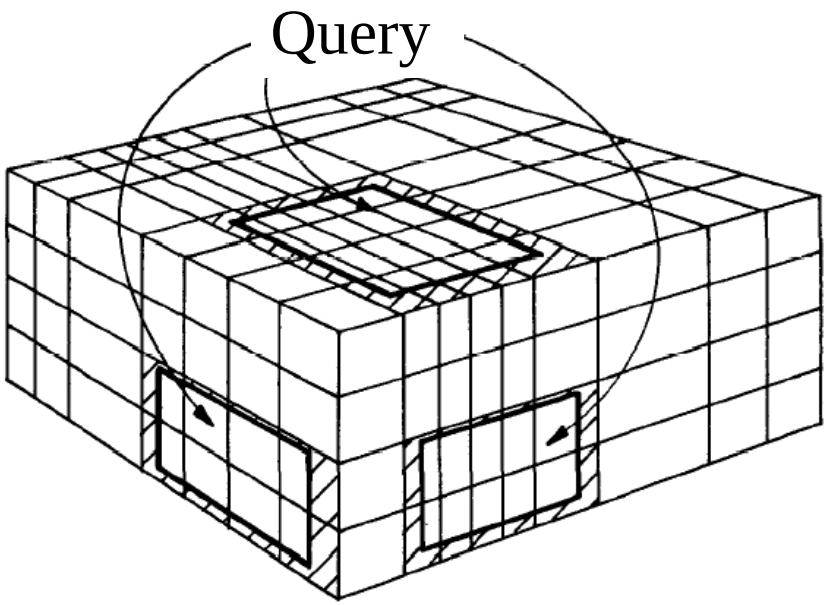



**Exact match:**

(1) Use liner scales to determine the index into the cell directory
(2) Access the cell directory to retrieve the bucket address
(3) Access the appropriate bucket

# Grid file search - Range queries

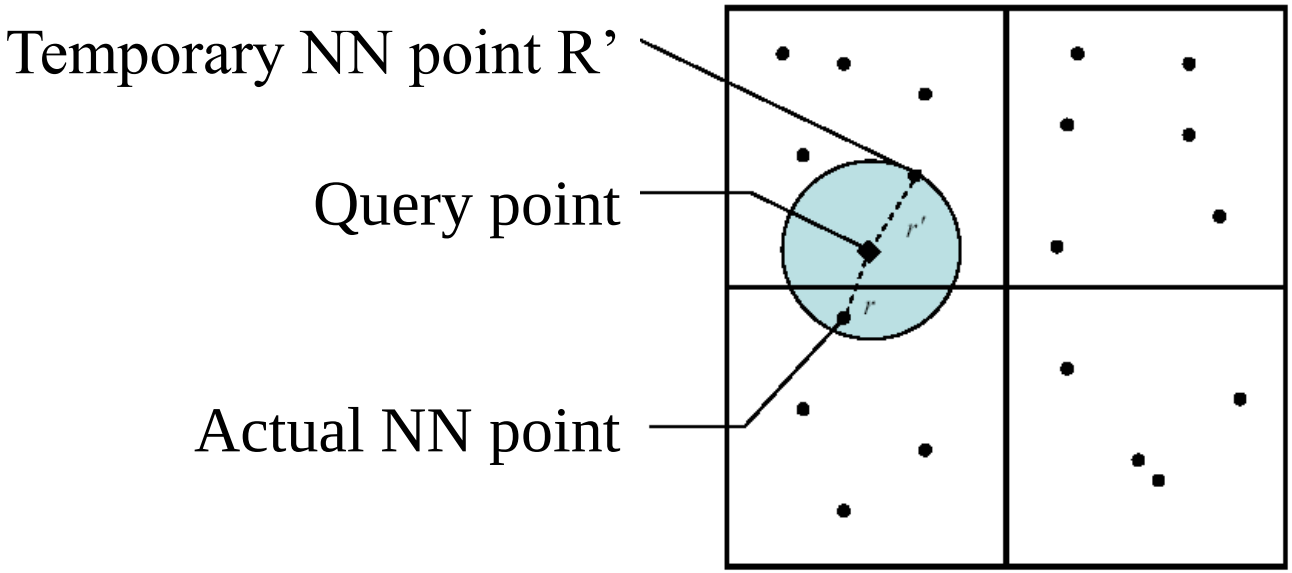
- Range queries:
  - (1) Use linear scales to determine the index into the cell directory
  - (2) Access the cell directory to retrieve the bucket addresses to visit
  - (3) Access the buckets
  - (4) Verification



# Grid file search – Nearest neighbor query - I

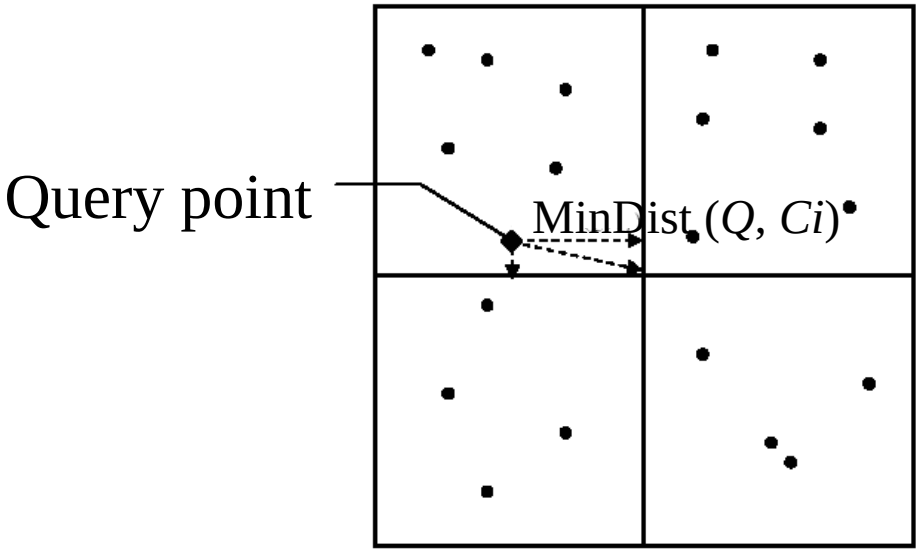
- Identify the bucket to which the query point Q belongs.
- By examining the corresponding records, generate:

  (1) a *temporary NN point R'* and (2) a *temporary NN distance r'*=D(*Q*,*R'*).



# Grid file search – Nearest neighbor query - II

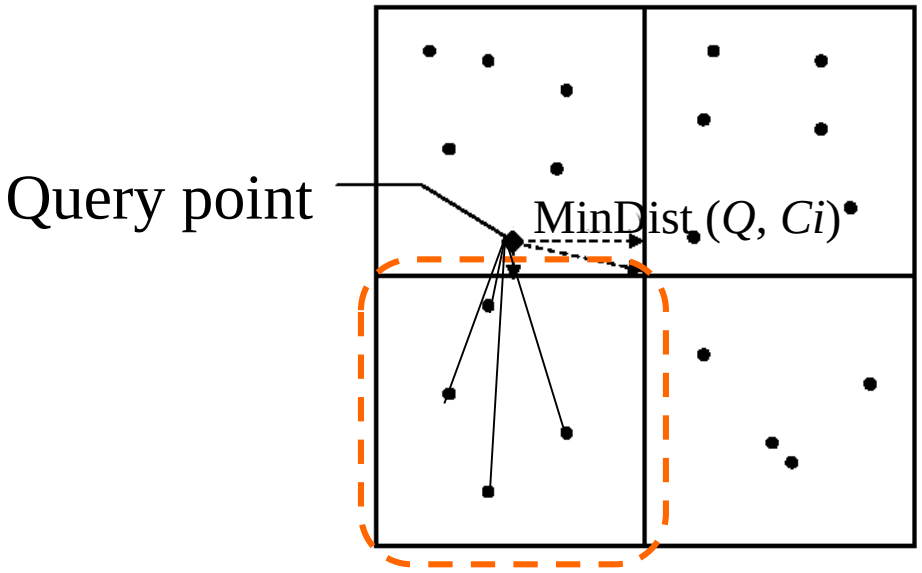
- Select candidate
  - Compute the minimum distances MinDist ($Q$, $C_i$) from the query point $Q$ to each cell $C_i$ intersecting with the *temporary NN sphere*
  - For each cell $C_i$, if MinDist($Q$, $C_i$)≥$r$', it is discarded (why?)
  - Otherwise, the cell is added to a *priority candidate list*, in which the cell with smaller *MinDist* has a higher priority



# Grid file search – Nearest neighbor query - III

- The disk block of each candidate cell is accessed to update both *temporary NN point R'* and NN distance *r'*.
- The search stops whenever

  *MinDist* of the next candidate ≥ the current *NN* distance *r'*.

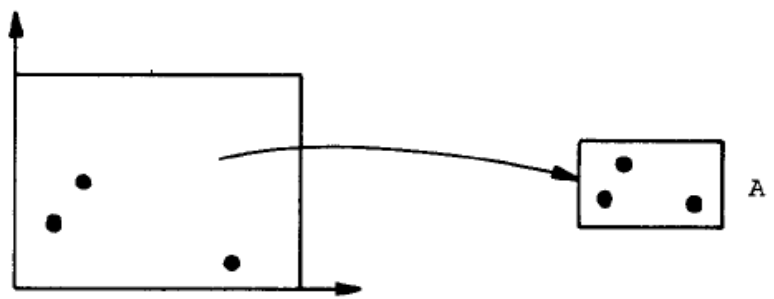


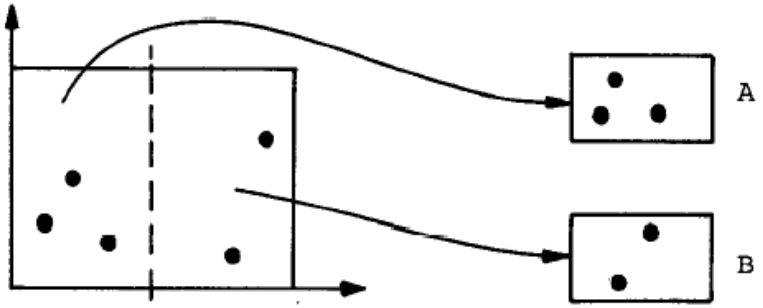
# Grid file insertion

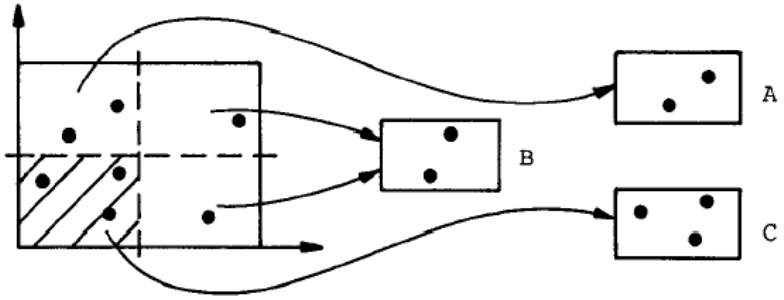
- Determine the bucket into which insertion will occur.
- If space in bucket, insert.
- Else, split bucket.
  - If it cause the directory split, do so and adjust linear scales.

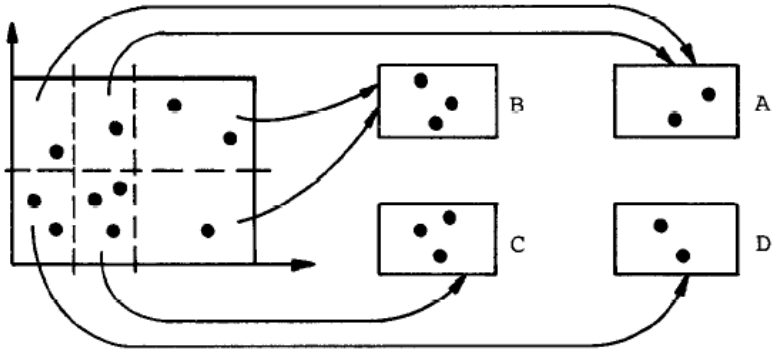
- Directory need split?
- Which dimension to split?
- Insertion of new entries potentially requires a complete reorganization of the cell directory

# Example – 2D

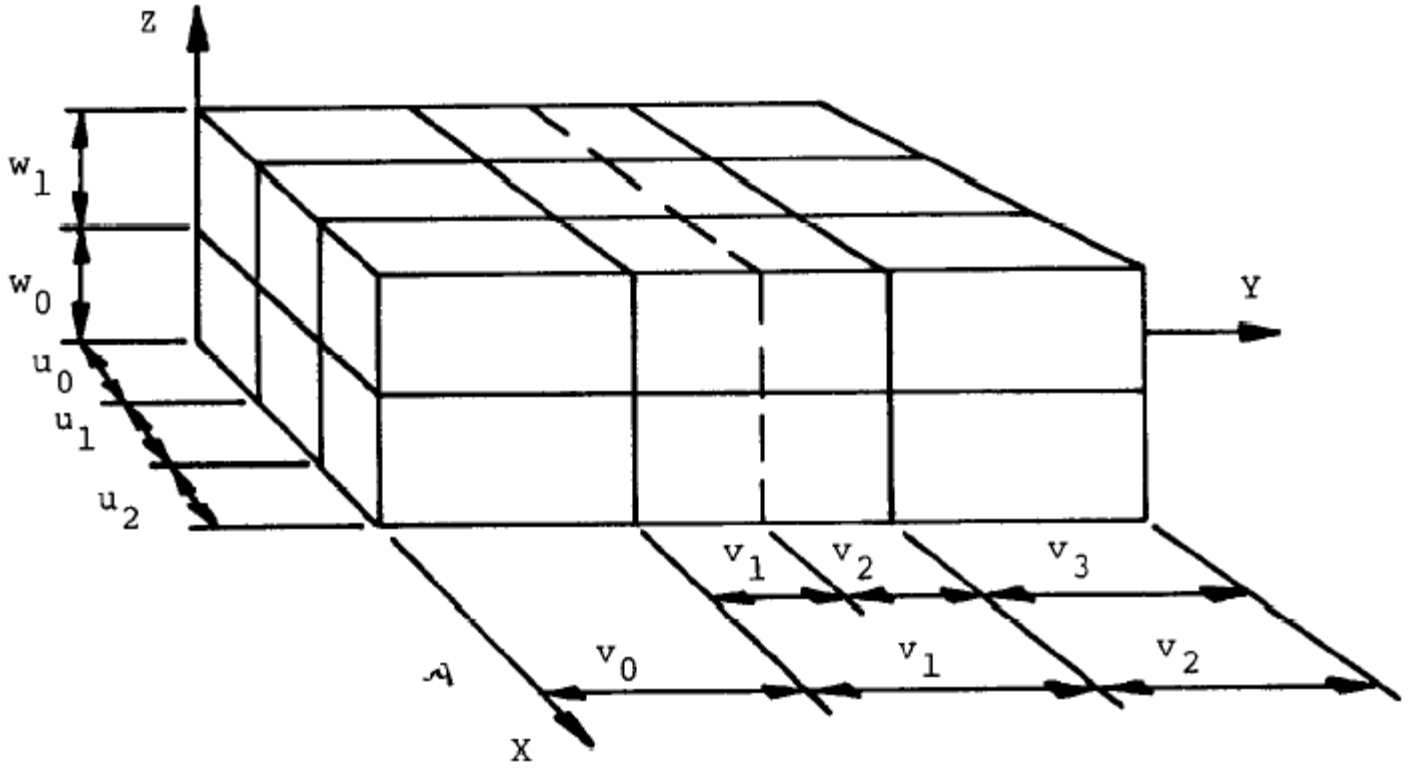


# Example – 3D



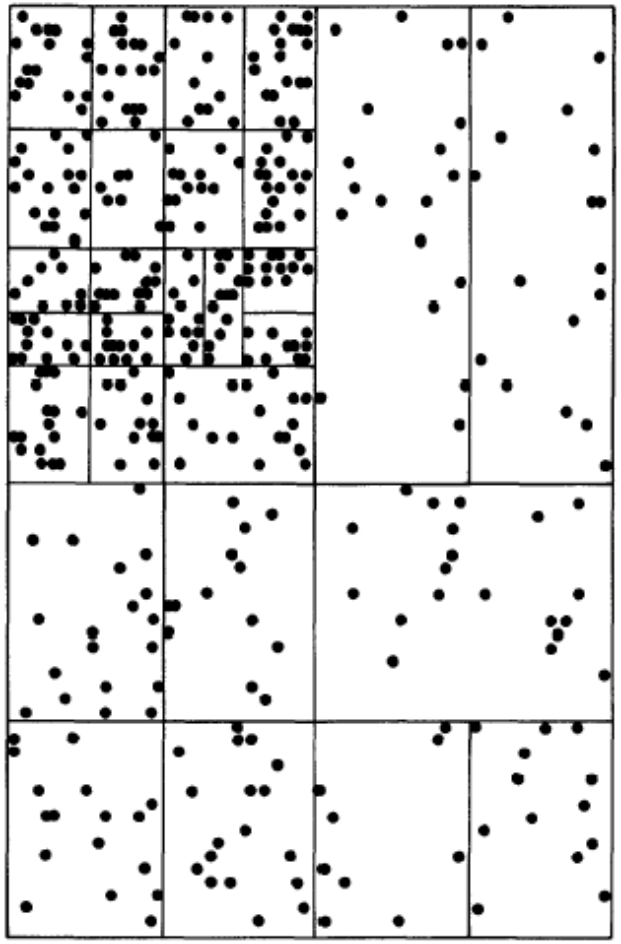
# Grid File deletion

- Determine the bucket from which deletion will occur.
- Delete the record from the bucket
- If *space utilization* of the bucket is less than a threshold
  - Merge the bucket with adjacent one

# Remarks

- A **space partitioning** strategy
- **Select** dividers (**not fixed**) along each dimension.
- Adapt to **non-uniform distributions**
- **Two disk access** (exact match)

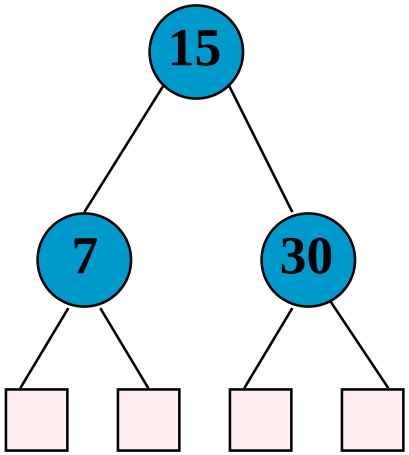
- What if the attribute values are correlated (e.g., age and salary)
- **Cell directory potentially exponential in the number of dimensions**
- **Split and merge may cause a rewrite of the entire directory**

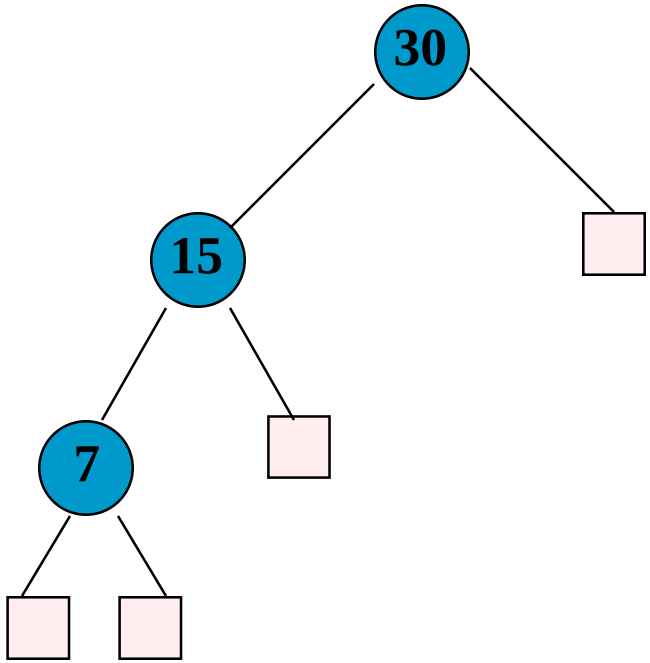


# Agenda

- Problem: **Multidimensional Data Retrieval**
- Method:
  - **Grid file**
  - **K-d-tree:** Generalization of binary search tree in multiple dimensions
  - R-tree
  - .....

# Binary search tree




***15, 7, 30,……***

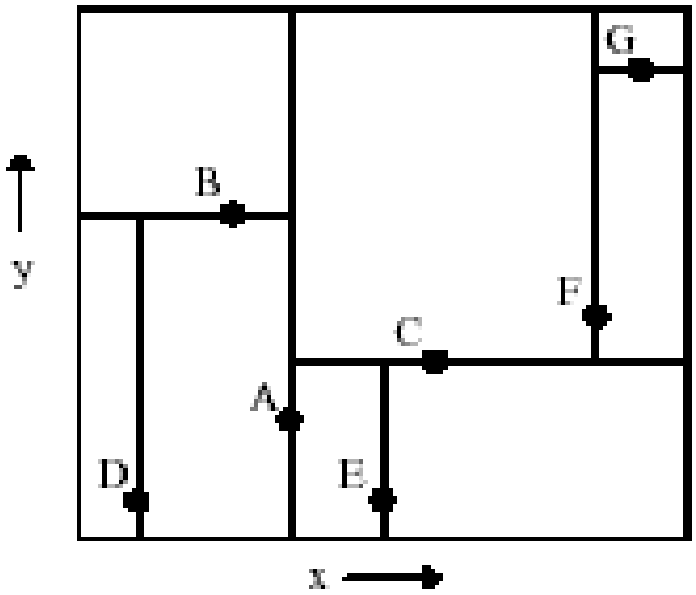
K-d tree: Extension of Binary Search Tree

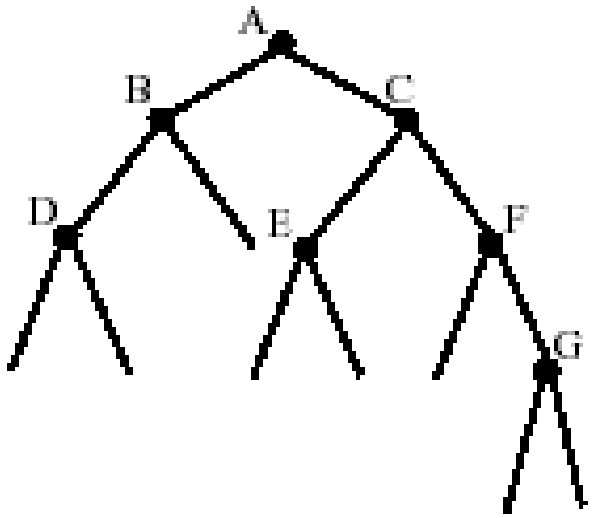
# K-d tree structure

- At each Internal node, the k-d tree divides the k-dimensional space into two parts by a (k-1)-dimensional *divider*

  height 0 cuts on x, height 1 cuts on *y*,

  height 2 cuts on *x*, height 3 cuts on *y*, …

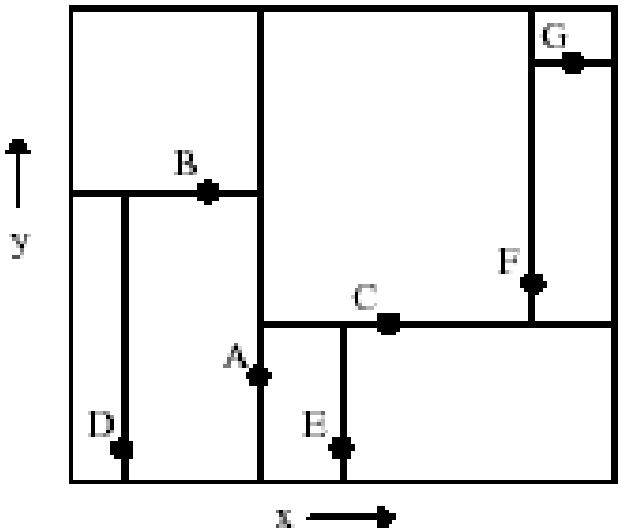


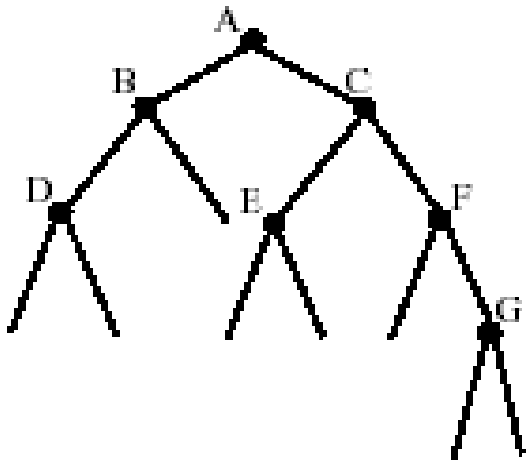




***x* coordinate first**

# K-d tree structure

- Internal nodes *N*: *k*-dimensional data
  - *N* is a decision point on a specific dimension
  - Splits the 1+(level(*N*) mod *k*) dimension into two regions
  - 2d-node {x,y: real, info:…, left, right: ^2d-node}
- Leaf nodes: failure

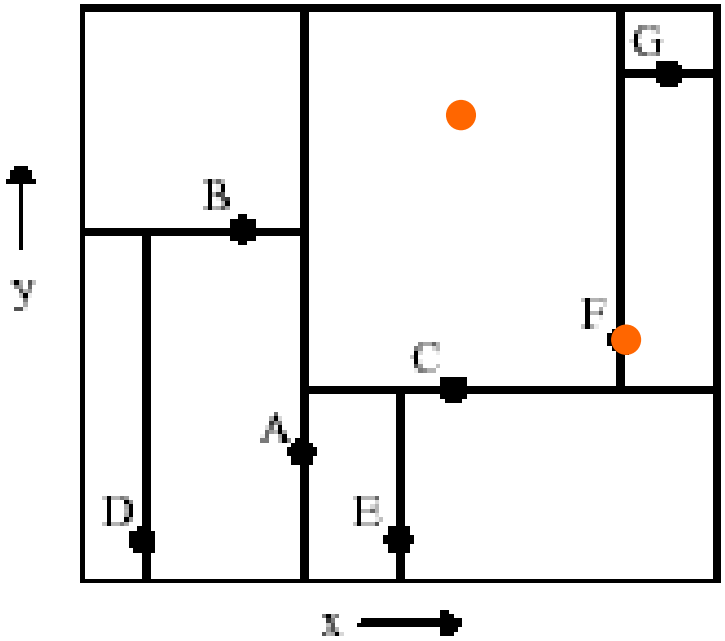


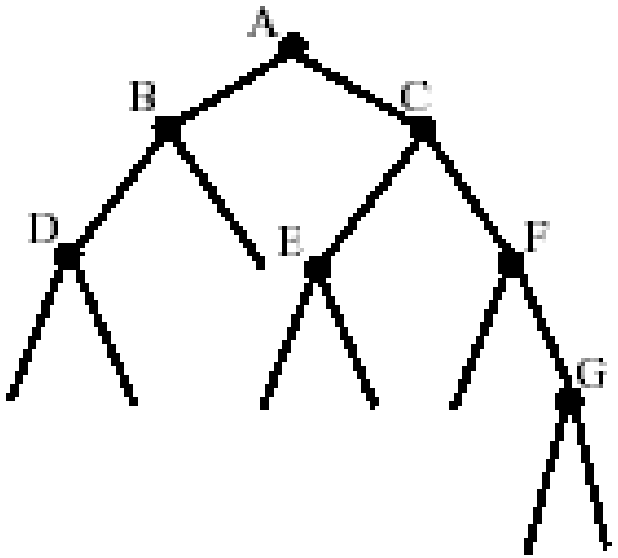




- Data stored in main-memory

# K-d tree search - Exact match

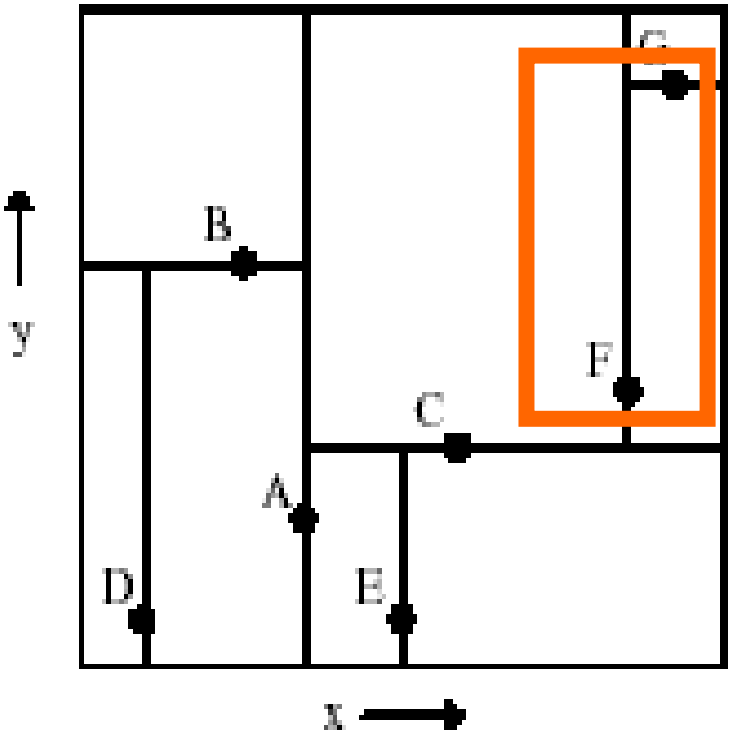
- Straightforward!

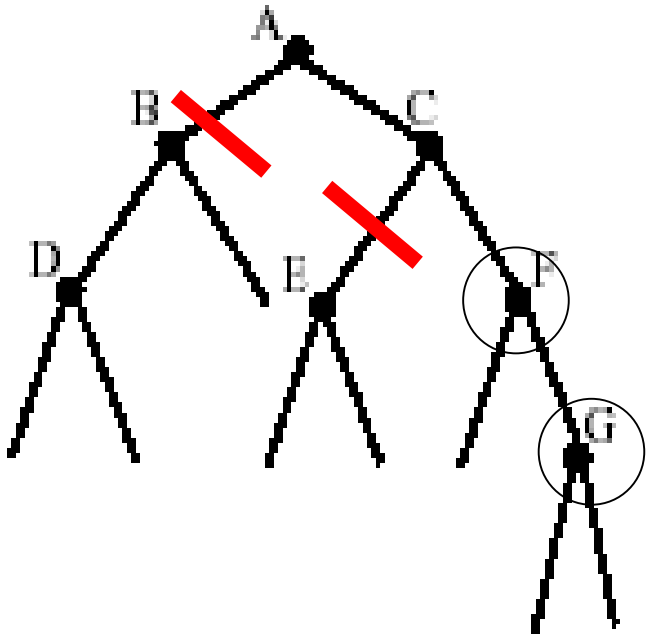


# K-d tree search – range queries

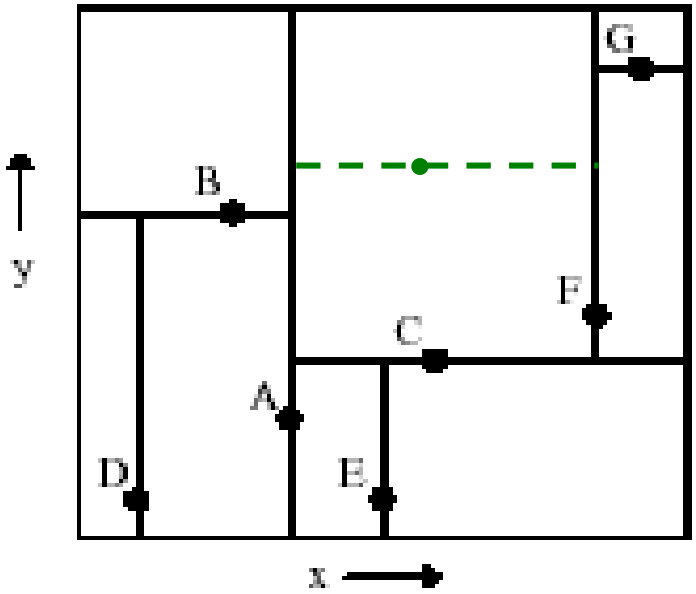
- Straightforward!

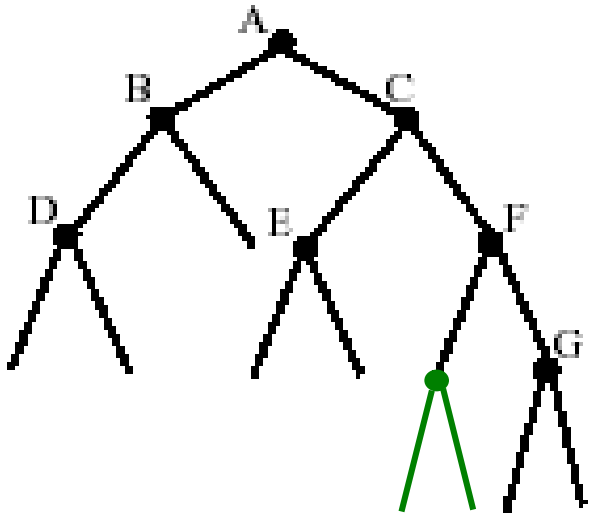


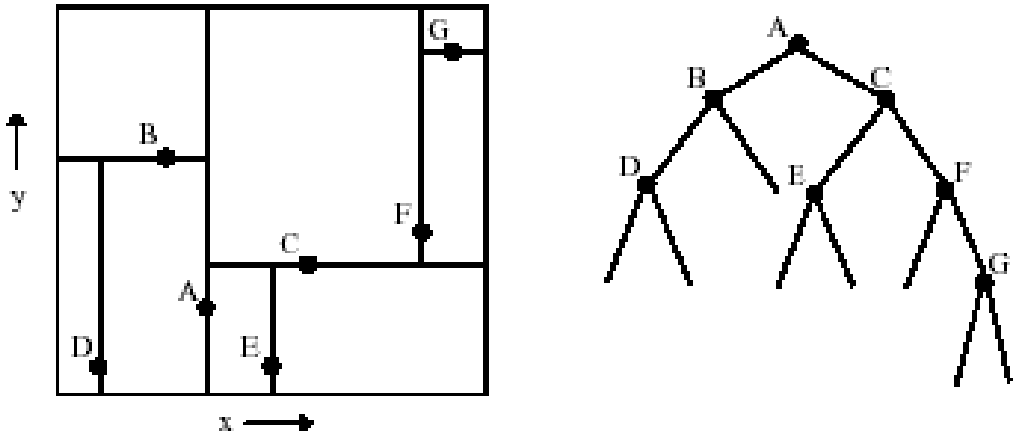
# K-d tree insert

- Straightforward!



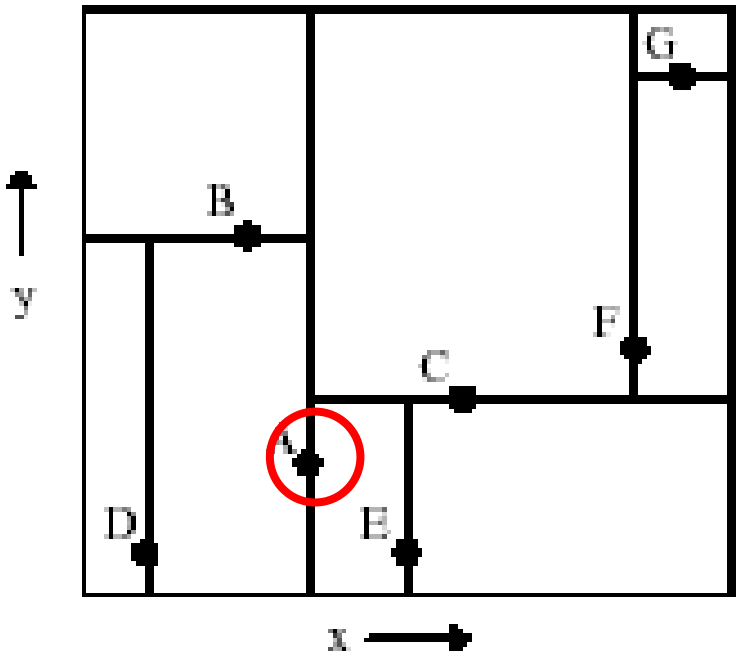
# K-d tree deletion

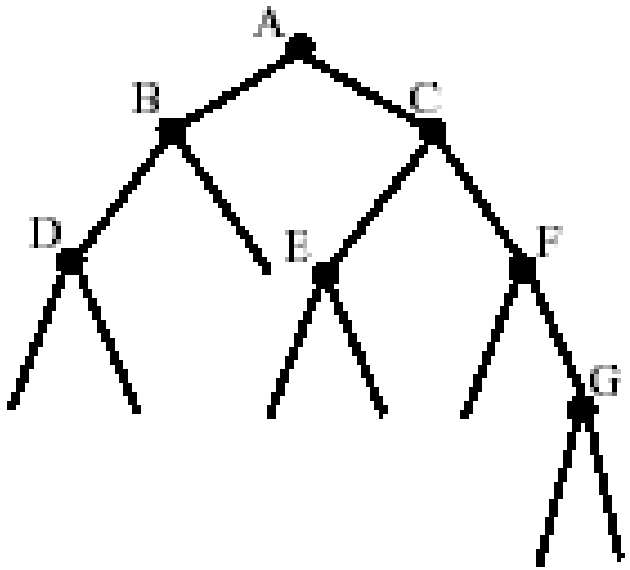



- Deletion may cause re-organization of the tree under the deleted node

- Deletion of a node n1:
  - Locate the node n1 on the tree
  - Find a *candidate* replacement node n2
  - Replace the contents of n1 with n2
  - Recursively delete n2
- Candidate node:
  - If n1 is an x node, then a possible candidate is
    - a node with the smallest x value in the right subtree, or
    - a node with the largest x value in the left subtree
  - Similarly for y nodes

# K-d tree deletion

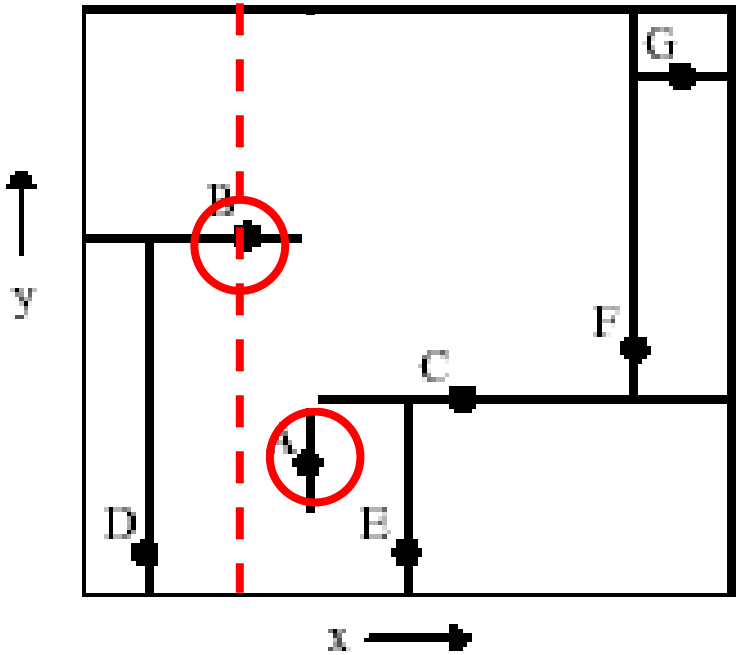


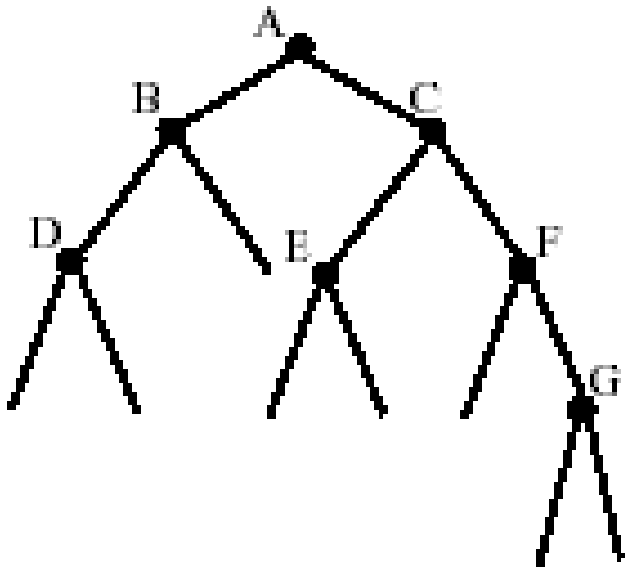




**Deletion:**

**<u>Locate the node n1 on the tree</u>**
**Find a candidate replacement node n2**
**Replace the contents of n1 with n2**
**Recursively delete n2**

# K-d tree deletion




**Deletion:**

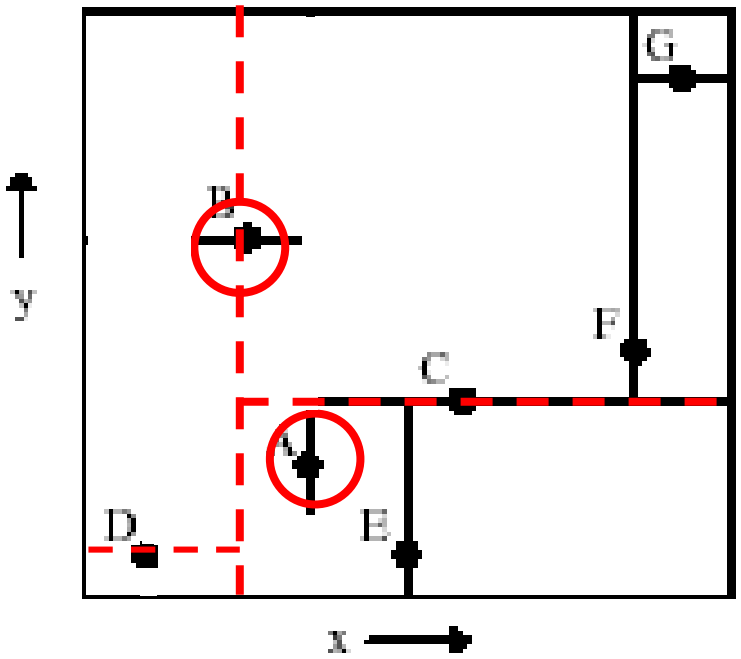
**Locate the node n1 on the tree**

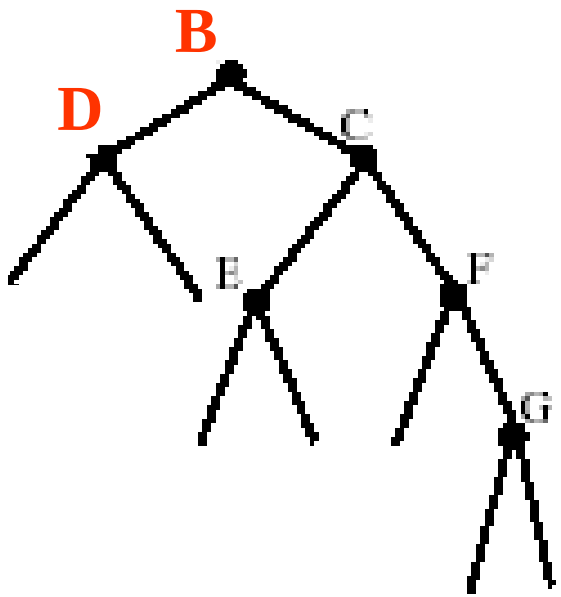
**Find a candidate replacement node n2**

**Replace the contents of n1 with n2**

**Recursively delete n2**

# K-d tree deletion



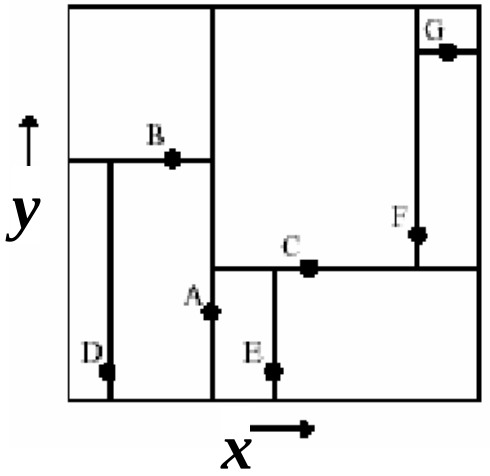



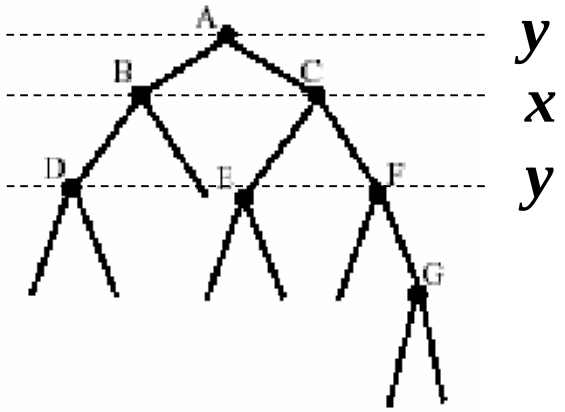

**Deletion:**

**Locate the node n1 on the tree**
**Find a candidate replacement node n2**
**Replace the contents of n1 with n2**
**Recursively delete n2**

# Remarks – K-d tree

- Extension of Binary Search Tree
- Height of the tree: O(log n)
  Search time for exact match: O(log n)
- K-d tree
  - Depends heavily on the insertion order of the points (best/worst case?)
  - The divider does not divide the space at the best possible positions, resulting in an unbalanced tree
  - Uncertain searching depth
- The tree scales linearly with the number of dimensions
- Main memory based
- Elegant ideas that
  - have been widely used in various fields
  - appear in many text books
- Many extensions
  - Adaptive k-d tree
  - K-d-B tree

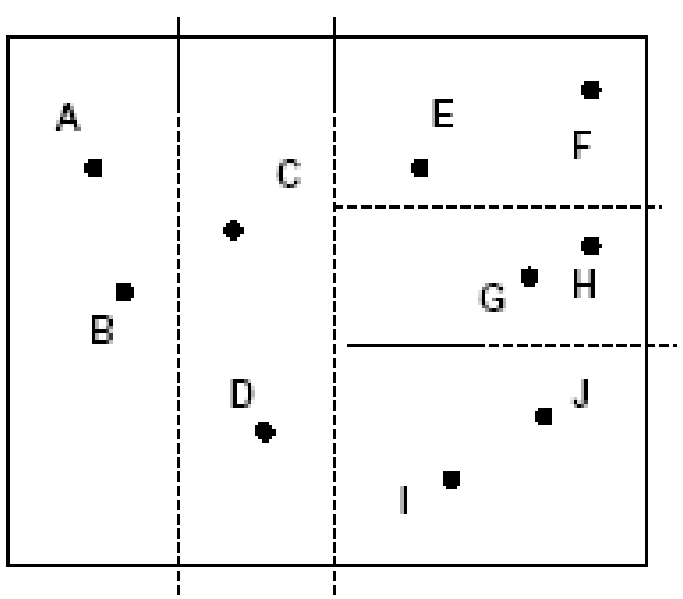


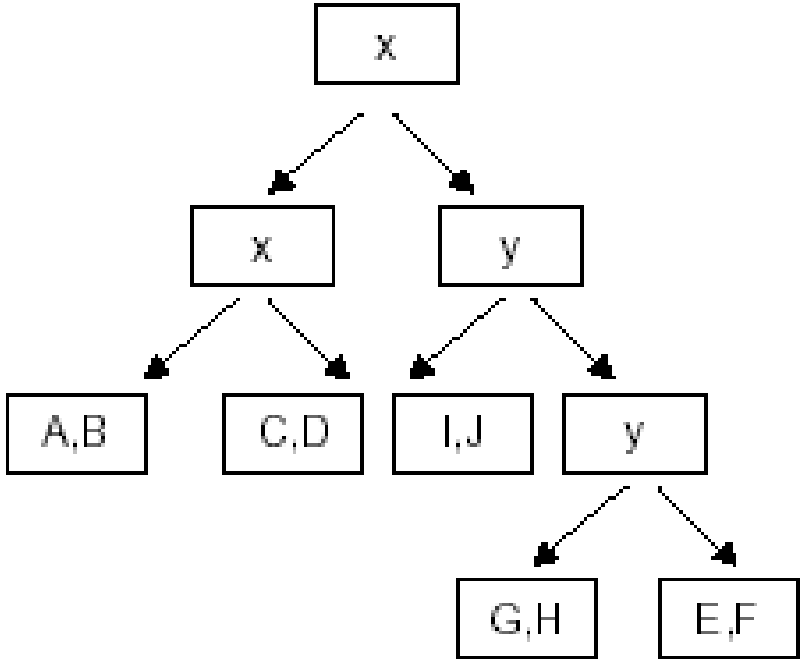



# Adaptive k-d tree

- An improved version of k-d tree
  - **All points are stored in the leaves**
  - Internal nodes contain the dimension (e.g. x or y), and the coordinate of the respective split
  - The divider is chosen as the **median value** between the points
  - The partitioning dimension is chosen **adaptively** (not alternately)
- It works well when the data is known a-priori, and there are rare updates in the tree

# Adaptive k-d tree

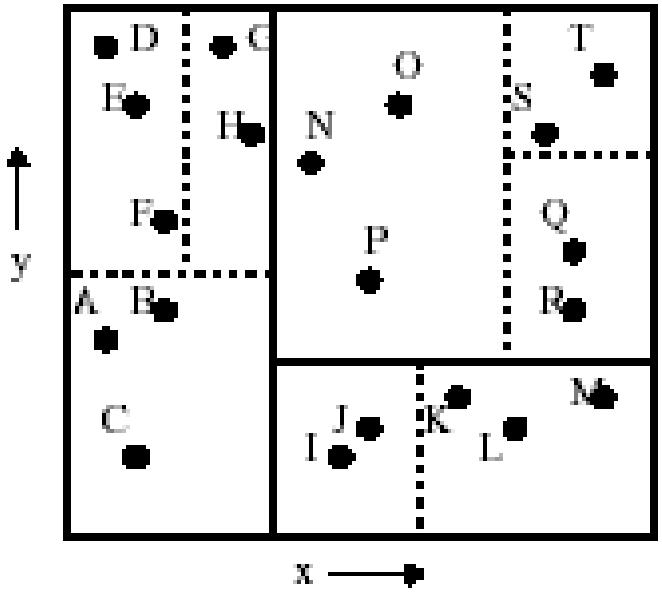


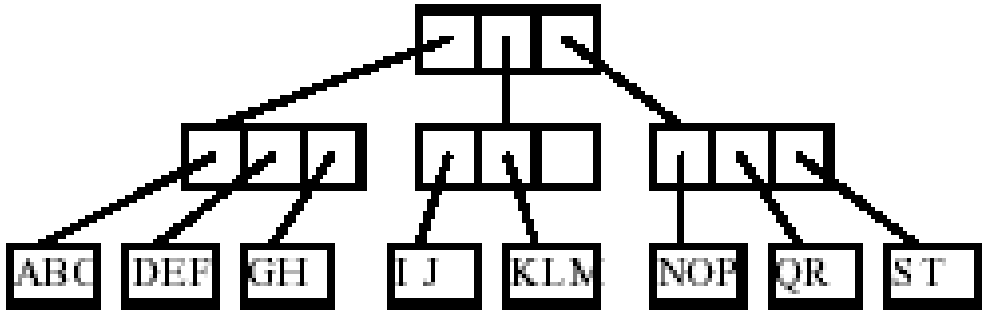



# k-d-B tree

- The k-d-B tree combines properties of both the adaptive k-d tree and the B-tree
  - More than one divider divide a tree node (depending on the tree's storage utilization)
  - All nodes of the tree correspond to disk blocks
  - Like the B-tree, the k-d-B tree is perfectly balanced
  - Insertion becomes quite complex and expensive
  - Storage utilization guarantee?

# k-d-B tree



# Conclusions

- Problem: Accessing method for multidimensional data
  - Data: spatial points/ regions, RDBMS, ….
  - Several query forms
- Grid file and k-d-tree
  - Architecture, update, access …
  - Grid file and k-d-tree are used in many stand-along systems

# Homework

- Read 4.1 (optional), 4.2.1, 4.2.2
  and additional reading material

- Given a 2D data set by yourself, construct the Grid file and k-d tree

# Thanks

## Feedback welcome