+

CSIT6000P Spatial and Multimedia Databases

2022 Spring

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Spatial Data Organization & Indexing

Prof Xiaofang Zhou

+ Learning Objectives

- What we will cover
  - Basic principles of managing multidimensional data
  - Representative data access methods for point and polygon data
  - Processing of some simple spatial operations using data access methods
- Goals
  - Understand major types of multidimensional data access methods, and their strengths and limits
  - Understand how point query, window query and join query are processed using various data access methods

+ Readings

- R. Güting, An Introduction to Spatial Database Systems, *The VLDB Journal*, 3:4, 1994
- V. Gaede and O Günther, Multidimensional Access Methods, *ACM Computing Surveys*, 30:2, 1998
- J. Orenstein and F. Manola, PROBE Spatial Data Modeling and Query Processing in an Image Database Application, *IEEE Transactions on Software Engineering*, 14:5,1988
- T. Brinkhoff, H.-P. Kriegel and B. Seeger, Efficient Processing of Spatial Joins Using R-Trees, SIGMOD 1993

+ Spatial Indexing

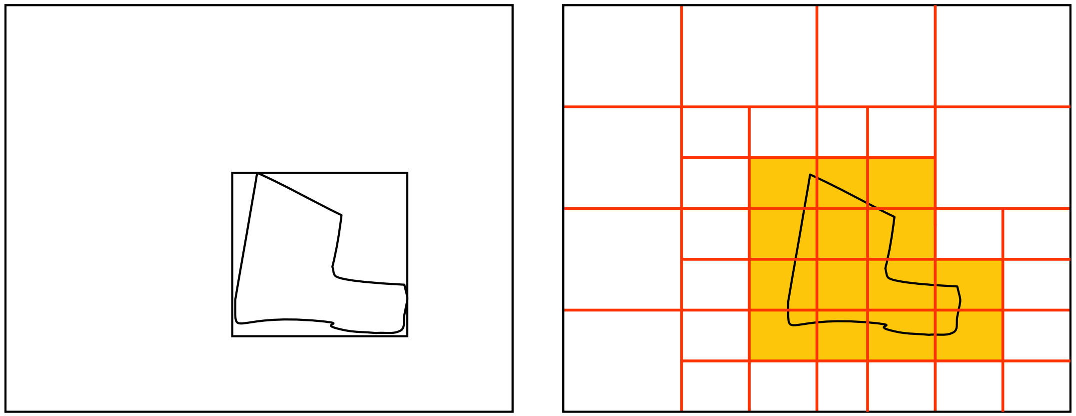
- Purpose:
  - Efficiency in processing spatial selection, join and other spatial operations
- Two strategies to organize space and objects
  - Dedicated external data structures
  - Map spatial objects into 1D space and use a standard index structure in RDBMS (e.g., B^+-tree)
- Basic ideas
  - Approximation and hierarchical data organisation

+ Object Approximation

- A fundamental idea of spatial indexing is the use of approximation

- Object approximation
  - Object centric
  - EG: use of MBRs (minimum bounding rectangles)

- Grid approximation
  - Space centric
  - EG: quadtree



+ What Do We Need to Know?

For each multidimensional access method:

- Motivation
  - Why is it proposed?
  - Good for what type of data?
    - points? polygons?
- Operations for creating and maintaining an index
  - Insert and delete data items
- Query operations using an index
  - Point query and window query
  - Spatial join queries and other queries

+ Background Knowledge

- Disks and files
- Basic indexing methods in relational DBMS
  - B^+-Tree
  - Hashing
  - Bitmap
- Query processing using indexes
  - What to achieve, and how?

+ Background: Disks and Files

- (S)DBMS stores information on hard disks
- This has major implications for DBMS design:
  - READ: transfer data from disk to main memory
  - WRITE: transfer data from RAM to disk
  - Both are high-cost operations (I/O), so must be planned carefully!

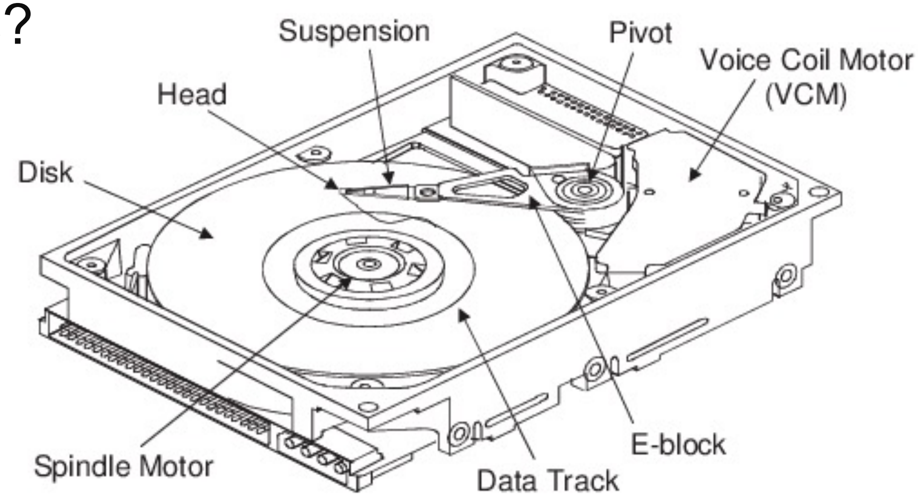
+ Magnetic Disks

- Secondary storage device of choice
- Supports random data access (vs. sequential access such as using tapes)
- Data is stored and retrieved in units called disk blocks or pages or buckets
- Unlike RAM, time to retrieve a page varies depending upon their locations on the disk

… technologies have evolved significantly, but these principles still hold

+ Accessing a Disk Block

- Time to access (read/write) a disk block:
  - seek time: moving arms to position disk head on the track
  - rotational delay: waiting for the block to rotate under the head
  - transfer time: moving data to/from the disk surface of the block
- Seek time and rotational delay dominate
- Key to lower I/O cost: reduce seek/rotation delays!
  - Hardware vs. software solutions?



+ Index Structure

- An index is an auxiliary file that makes it more efficient to search for a record in the data file
  - Think about the index at the end of a book
- An index is usually specified on one field of the file (although it could be specified on several fields)
  - It's called the search key, or simply key
- One form of an index is a file of entries ordered by key values: `<key, pointer >`
  - The pointer can point to a physical or logical address (of a page)

+ Costs and Benefits of Using Indexes

- An index always has two costs
  - Space to store it
  - The time to create and maintain it
- An index file typically occupies less disk blocks than the data file
  - Because its entries are much smaller
  - Therefore, index files are often kept in memory
- A binary search (for the sorted indexes) on the index yields a pointer to the file record
  - Then, an extra I/O access is needed to fetch the data

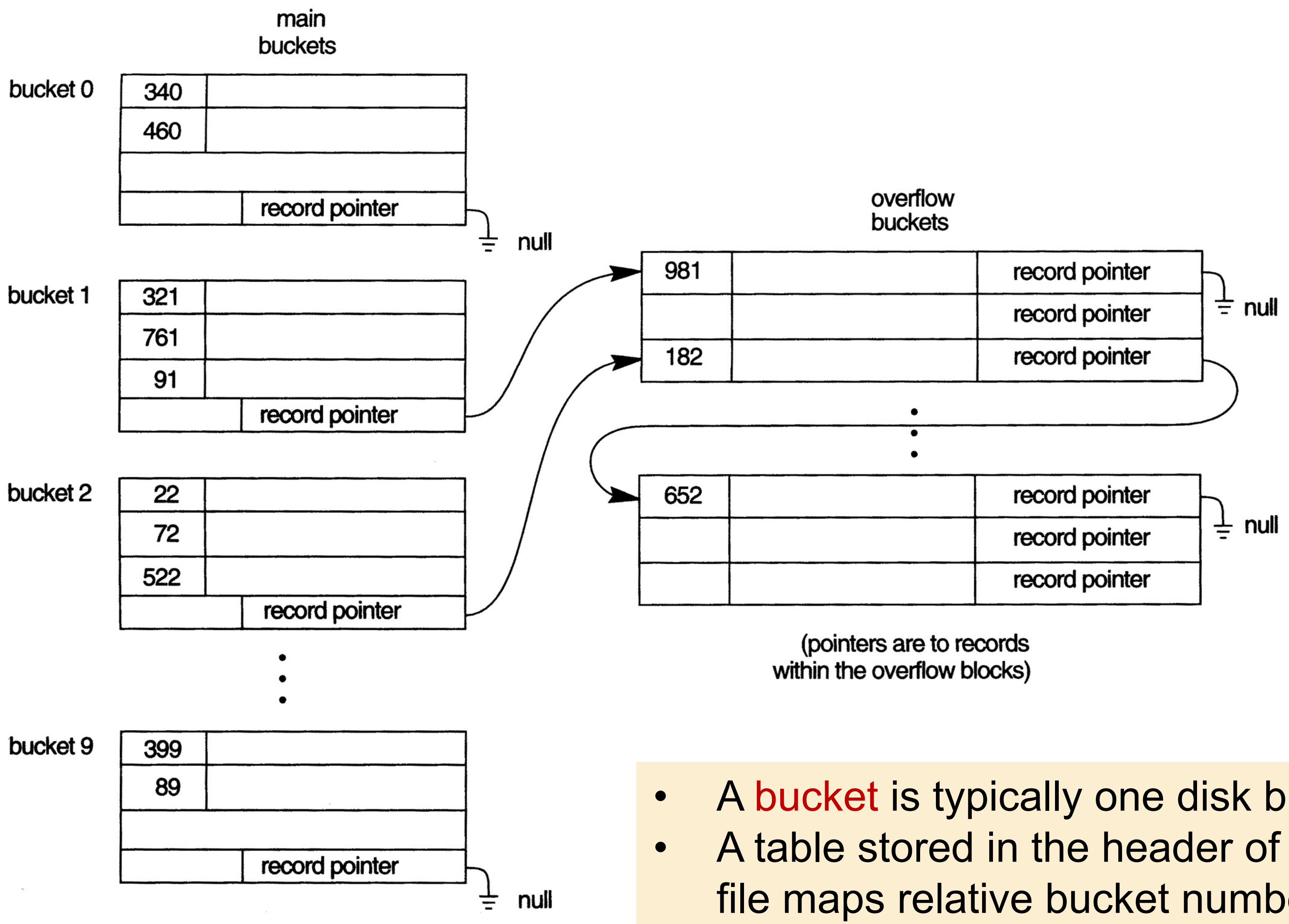
+ Data Access Methods

- One dimensional
  - Hashing and B^+-Trees
- Point data
  - Hashing: GRID and EXCELL
  - Hierarchical
    - Quadtree: point and region quadtrees
    - kd-Tree
    - Z-values and B+-tree
- Polygon data
  - Transformation: end point mapping and z-values
  - Overlapping: R-tree and R*-tree
  - Clipping: R+-tree

+ Static Hashing

- Hashing converts the key of a record to an address in which the record is stored
  - A hash function is used to map the key to the relative address of a bucket in which the record is stored
- If a file is allocated with *m* buckets, the hash function must convert a key *k* into the relative address of the block:
  - $h(k)$ in {0, ..., m-1}
  - For example, for integer key values, `h(k) = k mod m`
- Search cost?
  - Can be O(1)

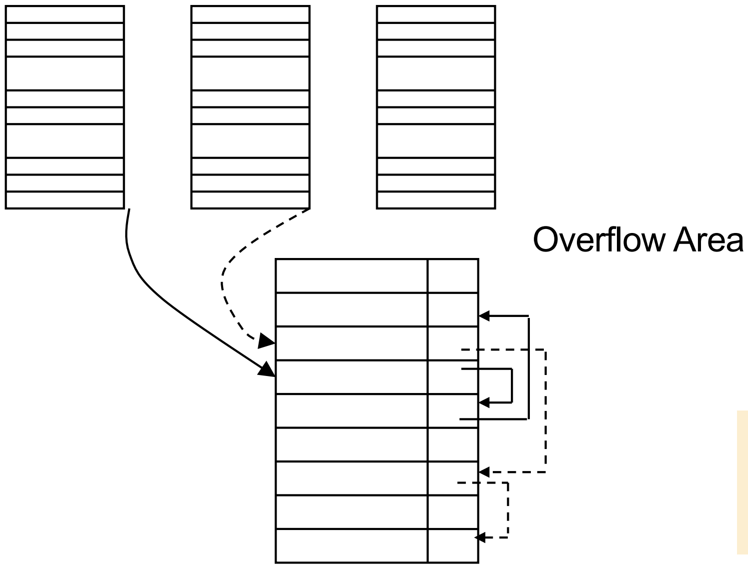
+ A Hashing Index Example



- A **bucket** is typically one disk block
- A table stored in the header of the hash file maps relative bucket numbers to actual disk block addresses

+ Collision

- Insertion of a new record may lead to collision
  - No space in $b = h(k)$
- Chaining: Use overflow buckets
  - Example: Common overflow area for all blocks in a file
    - Each block has a pointer to its first record in the overflow area
    - Records belonging to the same block are linked by pointers



- What problems may this cause?
- Why does this happen?

+ Hashing Functions

- A good hash functions must
  - be computed efficiently
  - minimize the number of collisions by spreading keys around the file as uniform as possible
- Example of hash functions
  - Truncation: take the first k bits of a key, for 2^k buckets
    - Taking the first 2 bits, there are $2^2 = 4$ buckets: 00, 01, 10, 11
  - Division: $h(k) = k \bmod m$, where *m* is the number of buckets

… for the truncation method, one can also takes the last k bits. Compare and contrast these two truncation methods

+ Pros and Cons of Hashing

- Excellent performance for searching on equality on the key used for hashing
  - Records are not ordered, thus any search other than on equality is very expensive
- Choose a good hash function is difficult
  - Must assume the data distribution (now and in the future)
  - Prediction of total number of buckets is difficult
    - Too few or too many are both bad
    - A practical approach is to estimate a “reasonable” size and periodically reorganize
- Recently strong renewed interest in hashing
  - Learn-to-hash, order-preserving hashing, and locality-sensitive hashing (LSH)

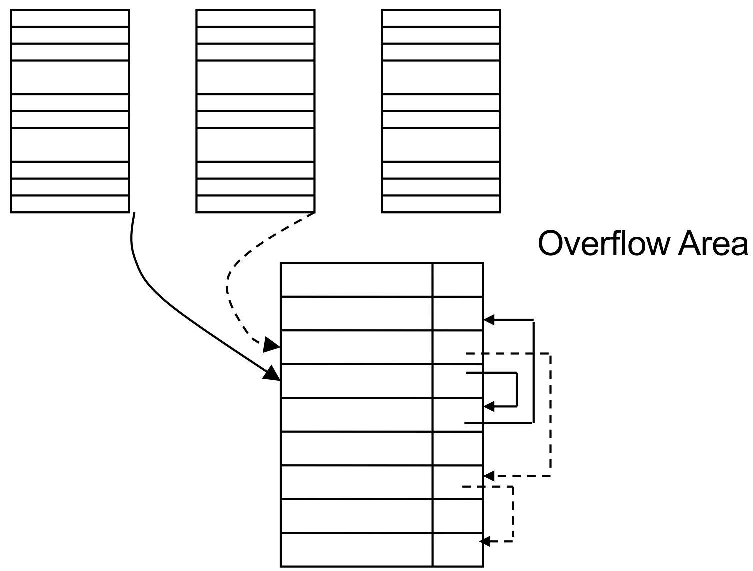
+ Linear Hashing

- The file size grows linearly, bucket by bucket
  - The file starts with *m* main buckets, numbered from 0 to *m*-1
- Initial hashing function $h_1(k) = k \bmod m$
- We keep the following info:
  - *n*: a pointer to the bucket that should be split next
  - Initially, *n* = 0

+ Insertion of a Record

If there is a collision, do:

- Push the record to an overflow bucket
- (Grow) A new bucket is appended at the end of the hash table
- (Split) Records in bucket number *n* (including those in the overflow space) are hashed again using
  - $h_2(k) = k \bmod (2m)$ (this will return either *n* or *m*+*n*, why?)
  - That is, these records will either remain in bucket *n* or in bucket *m*+*n*
- If $n = m$ (all original buckets have been split), we will set:
  - $n = 0$
  - $h_1(k) = k \bmod (2m)$
  - $h_2(k) = k \bmod (4m)$



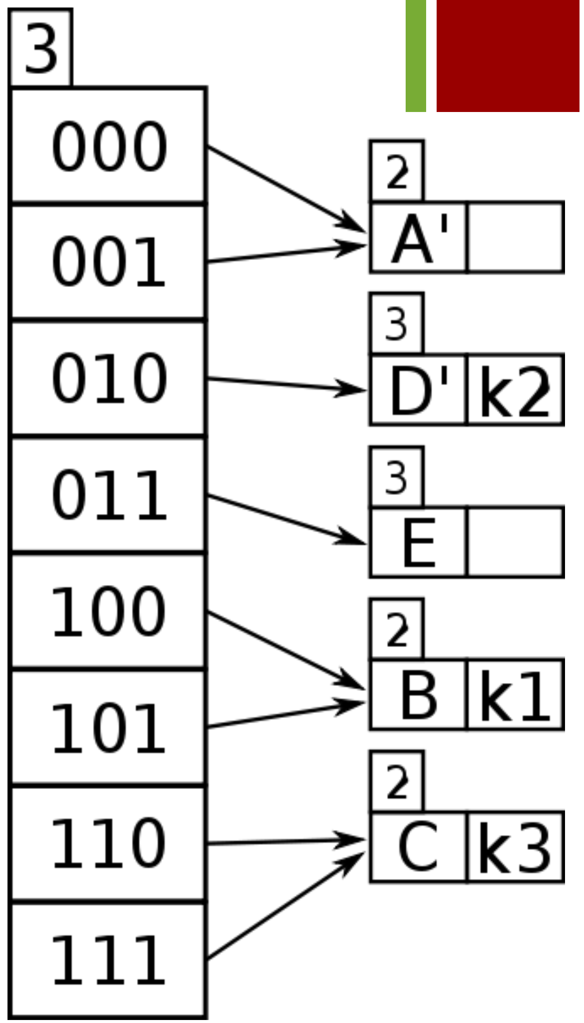
For all k values such that $h_1(k) = k \bmod m = n$, they are in the form of $k = i \times m + n$.
Now $h_2(k) = k \bmod (2m)$
When i is an odd number, $h_2(k) = m+n$
When i is an even number, $h_2(k) = n$

+ Search for a Record

- Search for a record with field value *k*
  - $l = h_1(k)$
  - if $l < n$ then $l = h_2(k)$ (i.e., bucket l has been split already)
  - search the bucket whose hash value is l
- Advantage
  - The file size grows linearly, bucket by bucket
- Disadvantage
  - No guarantee that a split relieves the overloaded bucket
    - A split is not at where the bucket currently overflows, but at the “next” bucket in a round-robin fashion
  - Still requires overflow buckets and chaining

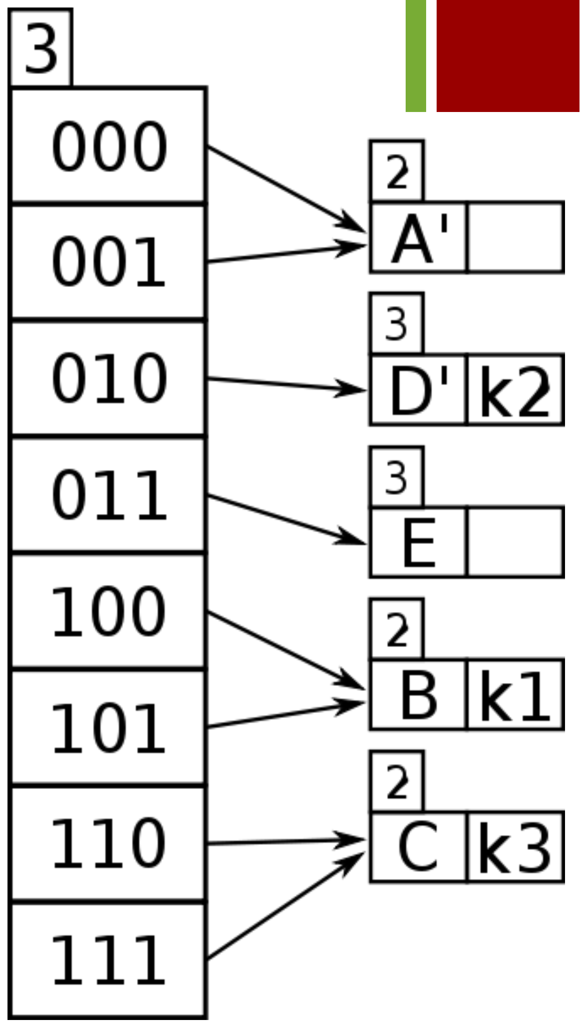
+ Extendible Hashing

- The file is structured into two levels:
  - directory and buckets
  - The directory has 2^d entries (d: global depth)
    - Each entry points to a bucket
    - Use some hashing function that generates a string of bits
    - The first *d* digits used as index into the directory
    - Several entries can point to the same bucket
  - local depth *d'* (*d'* <= *d*)
    - Only the first *d'* bits are used



+ Extendible Hashing Adjustment

- The directory size can be doubled or halved
  - Double: *d*=*d*+1, when a bucket with *d'*=*d* overflows
  - Half: *d*=*d*-1 when *d* > *d'* for all buckets
- Two-level search, highly efficient
  - No just-in-case space waste, no chain search



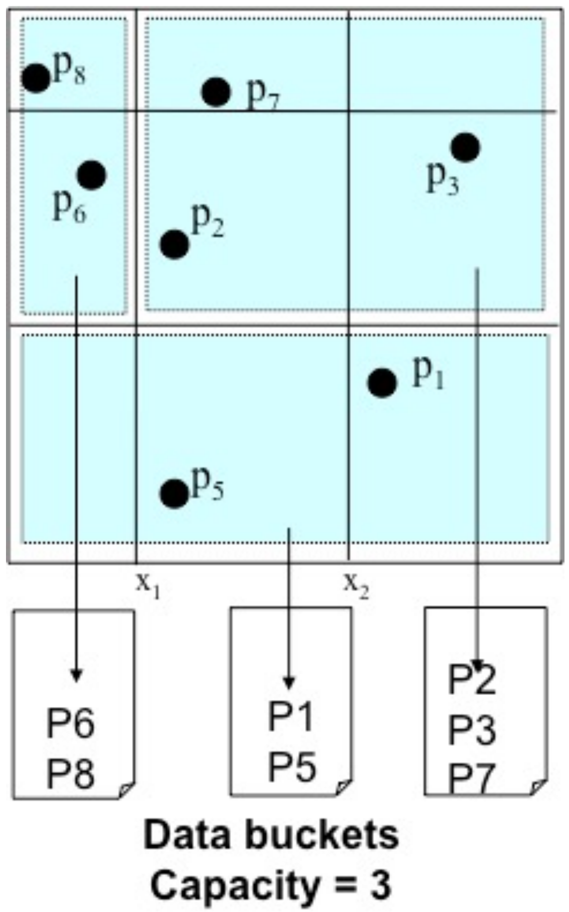
… practically all modern file systems use either extendible hashing (Ronald Fagin, 1979) or B-trees

+ Data Access Methods

- One dimensional
  - Hashing and B-Trees
- Point data
  - Hashing: GRID and EXCELL
  - Hierarchical
    - Quadtree: point and region quadtrees
    - kd-Tree
    - Z-values and B-tree
- Polygon data
  - Transformation: end point mapping and z-values
  - Overlapping: R-tree and R*-tree
  - Clipping: R+-tree

+ Grid File

- Basic idea
  - Superimpose a *k*-dimensional grid on the space
  - Cells can be of variable sizes
  - Cell-to-bucket mapping: many-to-1
    - *What about 1-to-many?*
  - The grid definition is kept in memory
  - The grid directory is kept on disk
- Motivation
  - A grid structure is simple, but a fixed-size grid is not suitable for non-uniformly distributed data



… what information to remember about the grid structure?

+ Grid File: Search

- To answer a point query:
  1. Use the grid definition to locate the cell
  2. Find the reference information (i.e., a pointer to the data bucket)
  3. Read the data bucket
  4. Search in the data block

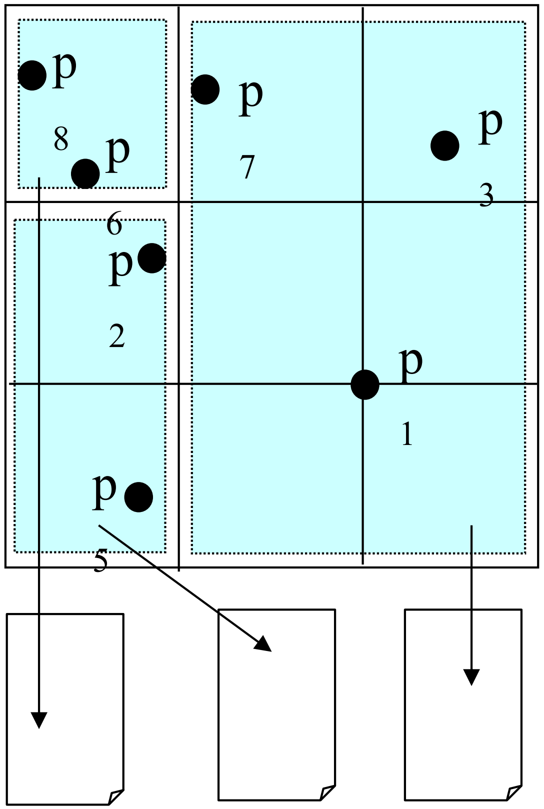
- To answer a range query:
  - Examine all the cells that overlap with the search region
  - Read the corresponding cells to locate the data buckets(s)
  - Read the data buckets, and search in these buckets

+ Grid File: Update

- To insert a point:
  - Search (using a point query) the matching cell and the corresponding data bucket
  - If there is sufficient space, insert into the data bucket
  - If the data bucket is full, add a vertical or horizontal line to split cells, and redistribute data accordingly
- To delete a point:
  - Can you do this?

+ EXCELL

- Motivation
  - Fixed grid is easier to manage and more efficient to use
- Basic idea
  - All cells are of the same size



Data buckets

… what information to remember about the grid structure?

+ EXCELL

- How to search for a point?
- How to search for all points in a given query window?
- How to insert a point?
- How to delete a point?

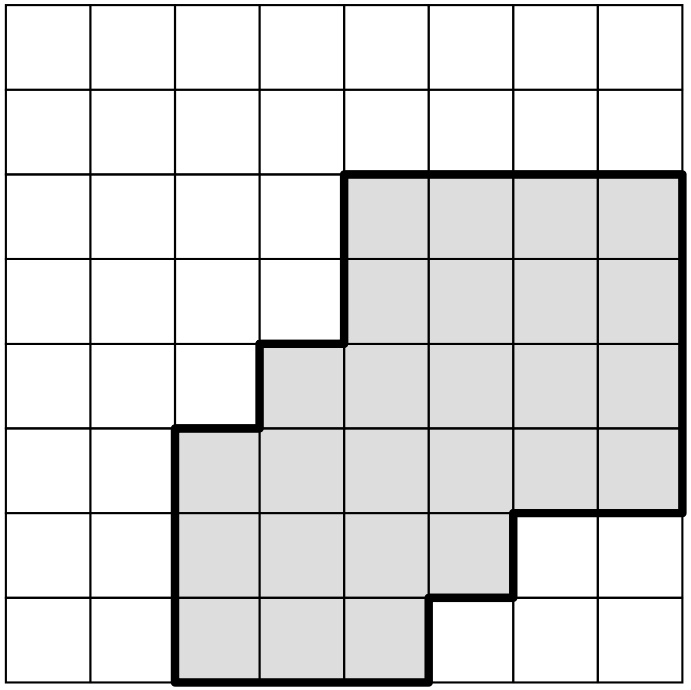
+ Data Access Methods

- One dimensional
  - Hashing and B-Trees
- Point data
  - Hashing: GRID and EXCELL
  - Hierarchical
    - Quadtree: point and region quadtrees
    - kd-Tree
    - Z-values and B-tree
- Polygon data
  - Transformation: end point mapping and z-values
  - Overlapping: R-tree and R*-tree
  - Clipping: R+-tree

+ Uniform Decomposition

Recursive decomposition of space

Resolution: max. level of decomposition, leading to $2^n \times 2^n$ cells



An object can be represented as a collection of shaded cells

To have 1 x 1 cm cells, what is required resolution?

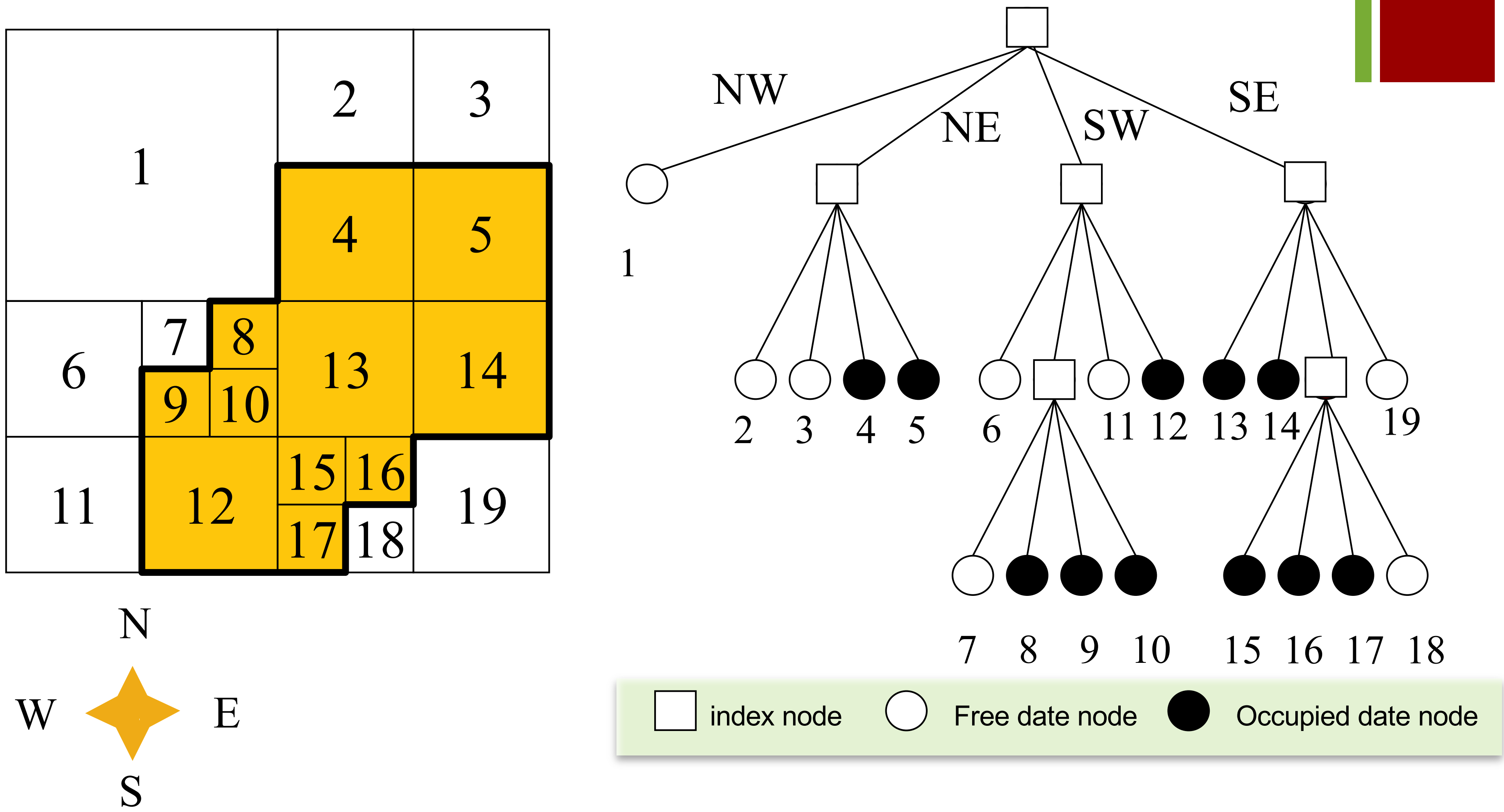
- an area of 5000 x 5000 km^2
- an area of 300 x 300 km^2

n=29 (25)

How many times do you need to fold a piece of paper to make it reach the moon?

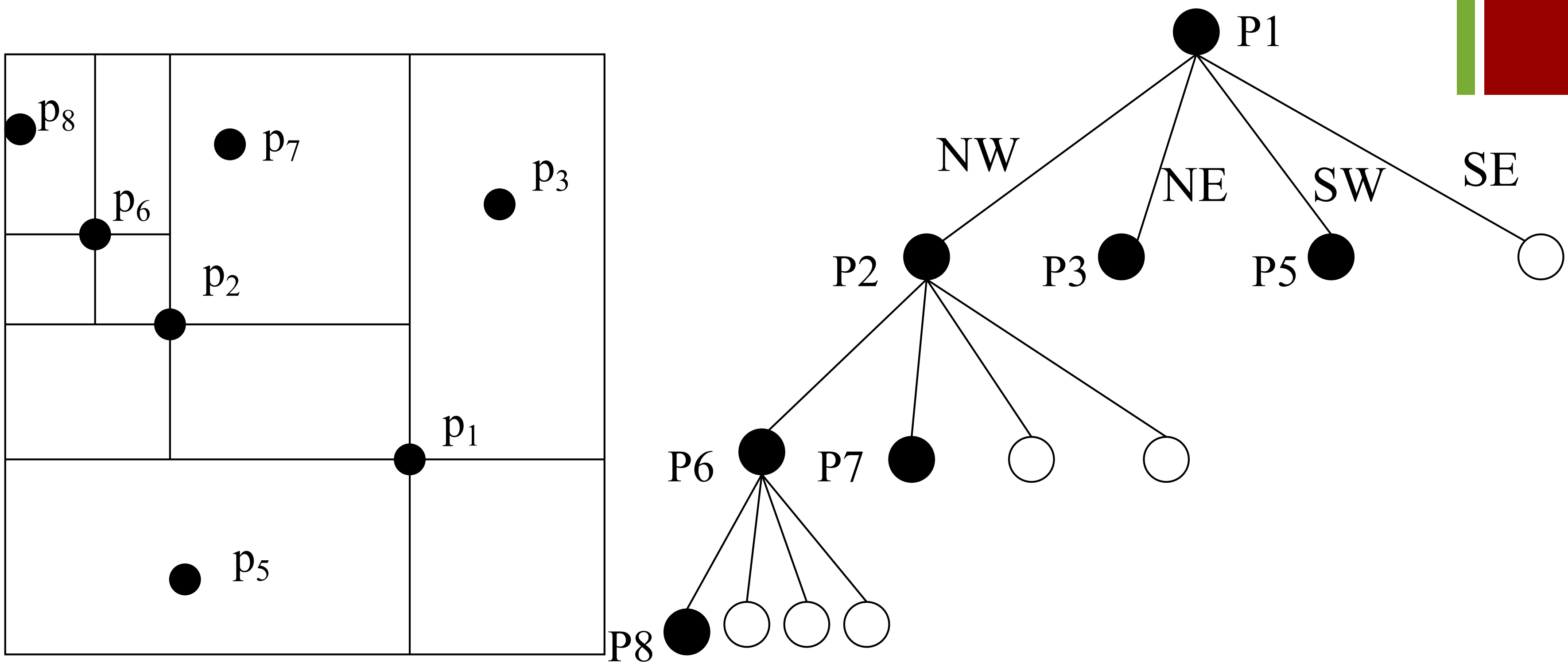
Thickness of paper sheet = 0.1mm
Distance = 384,403km

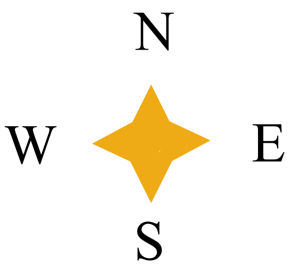
+ Quadtree – Basic Idea



…originally proposed for compact image representation that supports image union/intersection operations efficiently

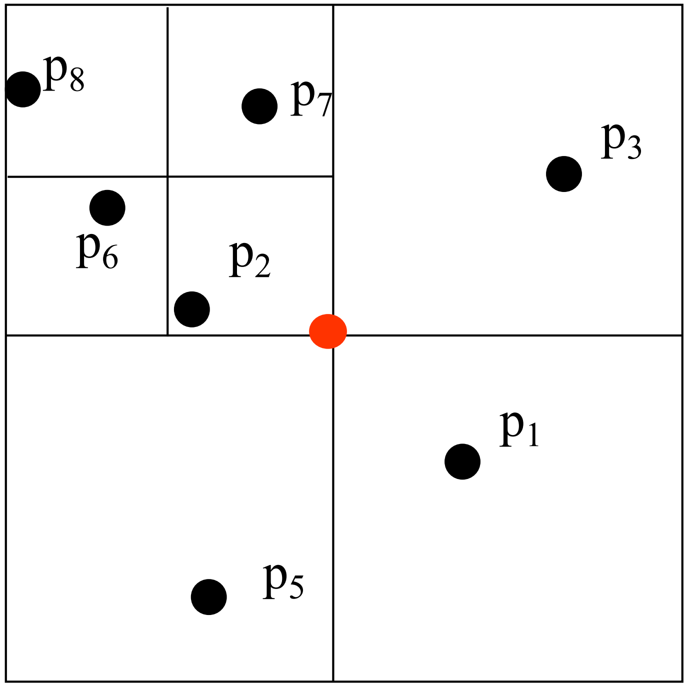
+ Point Quadtree Index

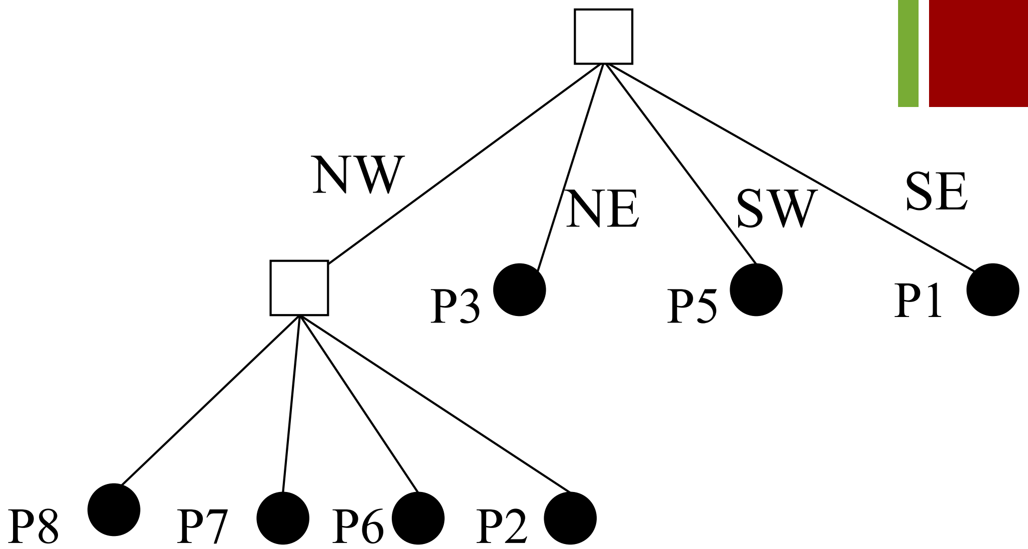




… the shape of the tree depends on the order to point insertion

+ Region Quadtree Index





… the shape of the tree <u>does not</u> depend on the order to point insertion

+ Some Questions

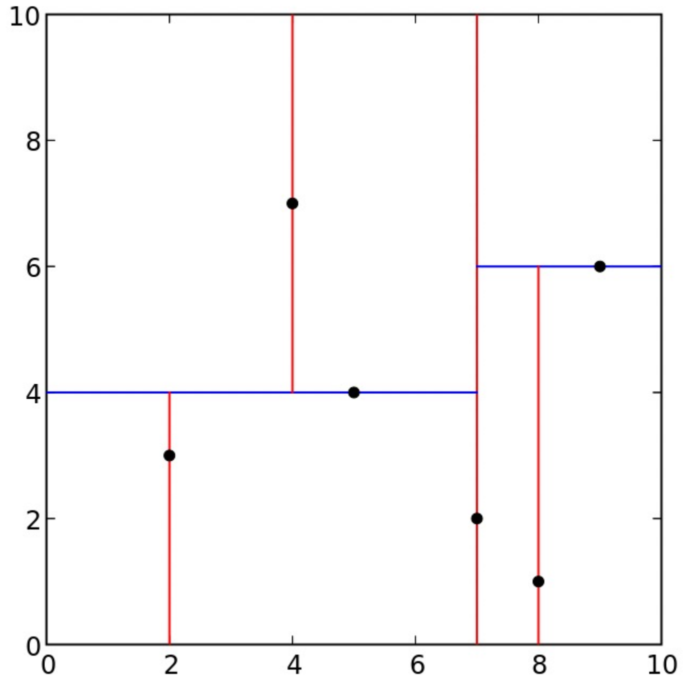
- Is the quadtree a balanced tree?
- When does deposition stop?
- What’s the main difference between the point quadtree and the region quadtree?
- Will the order of point insertion affect the shape of a quadtree?
- What information a node needs to record?
- How to perform point and region queries using these two types of quadtrees?

+ kd-Tree

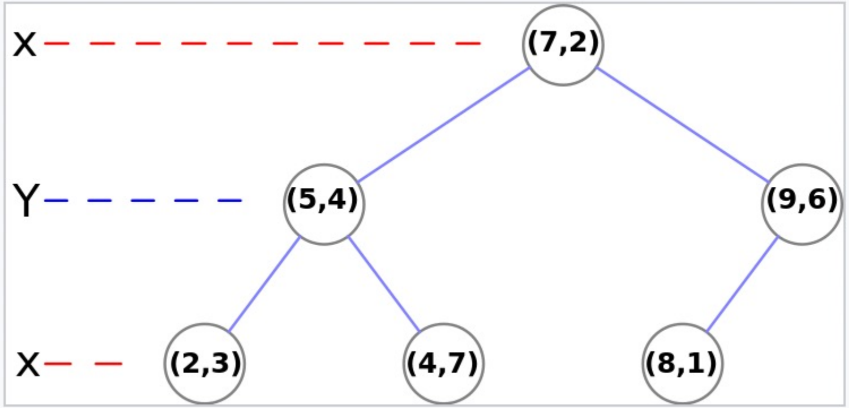
- Decomposition at data points (like the point quadtree)
- Motivation
  - For both point & region quadtrees: a node stores 2^k pointers and each time k comparisons are required to decide which subtree to go for a k-dimensional data
- Basic idea
  - Select one dimension, split according to this value and do the same recursively with the two new sub-partitions
  - Fan-out is a constant (=2) for arbitrary number of dimensions
  - Number of comparisons at each node is also a constant (=1)
- Problem
  - The resulting binary tree is high, and its *one-point-per-node structure* is not suitable for secondary storage

… very popular for in-memory data organization and commonly used in many application areas such robotics and computer vision

+ kd-Tree: Example



k-d tree decomposition for the point set `(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)`.



The resulting *k*-d tree.

The tree structure is dependent on the order of insertion (not robust for sorted data)

Many variations: non-alternative, data at leaves only, balanced kd-trees, representing regions etc.

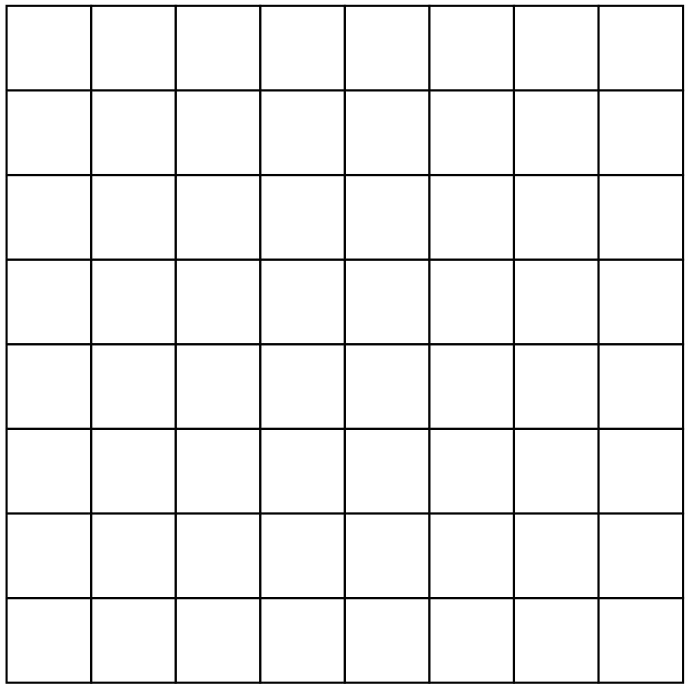
+ Some Questions

- How to use a kd-tree to support point query?
  - That is, to find if a given point exist in the database
  - Can you write the pseudo code for this?
- How to use a kd-tree to support range query?
  - That is, to find all points within a given rectangle?
  - Can you write the pseudo code for this?
- Later, we will discuss nearest neighbor queries, please come back to think about how you can support such a query using a kd-tree index
- What about insert or delete a point?
  - Typically used as an in-memory read-only data structure

+ Multidimensional Data

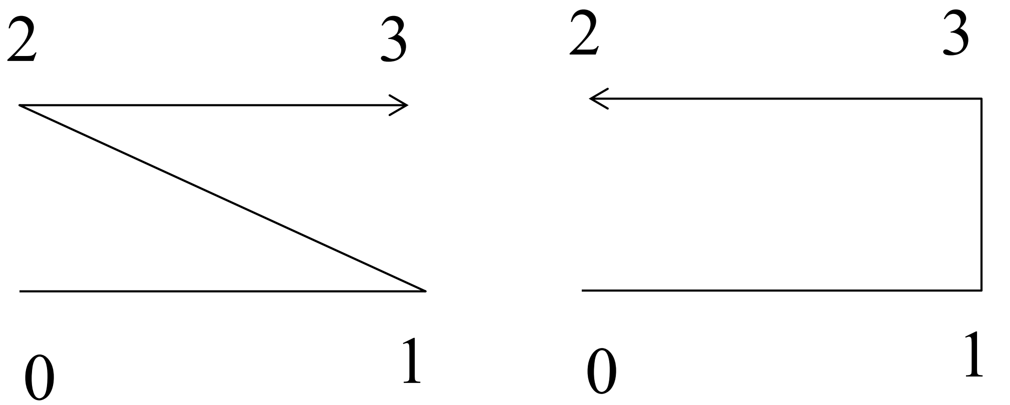
- There is no total order that preserves spatial proximity
- Solution:
  - Find heuristic solutions: use a total order mapping that can preserve the spatial proximity to some extent
- Idea:
  - If two objects are located close together in the original space (a k-dimensional), they should be close together in a target one-dimensional space (with high probability)
  - A balanced tree index structure for one-dimensional data is well known (B/B^+ tree)

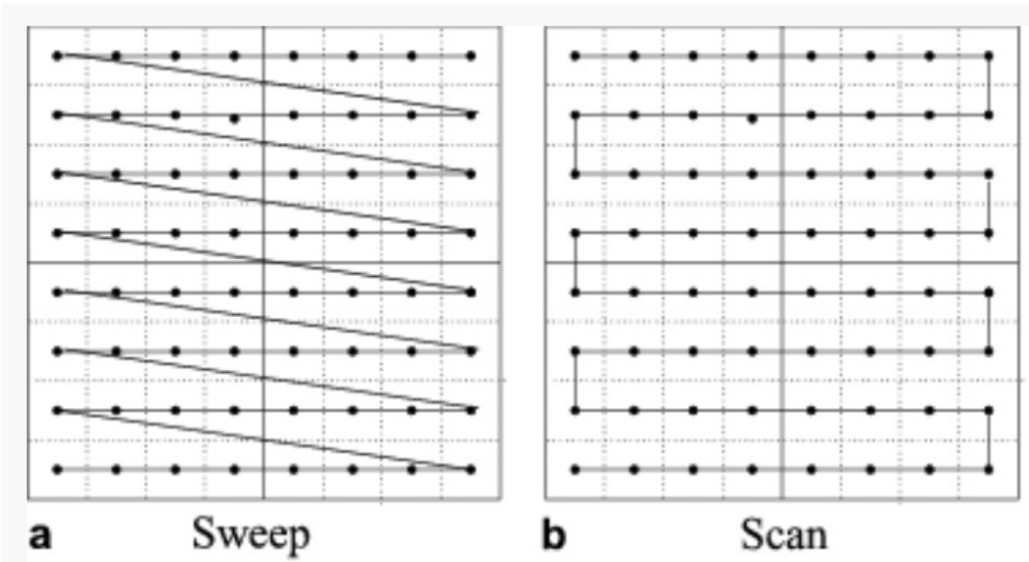
+ Space-Filling Curves



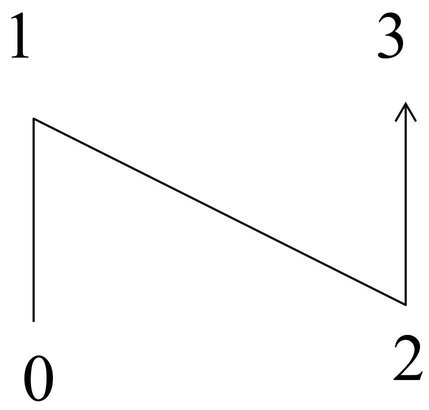
To find a space-filling curve in a high dimensional space to better preserve locality approximation

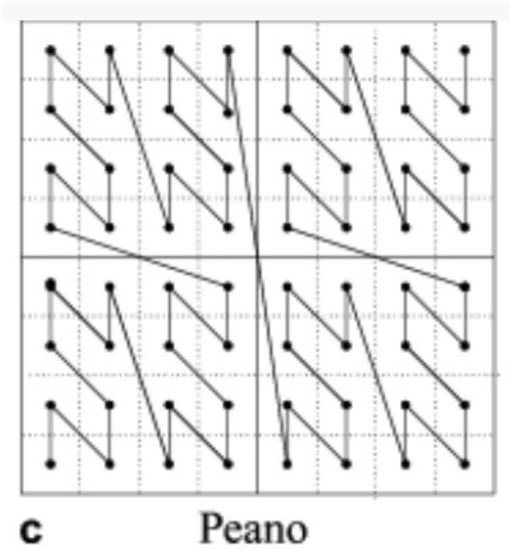
+ Row Order (or Column Order)





+ Z-Order (Peano Order)



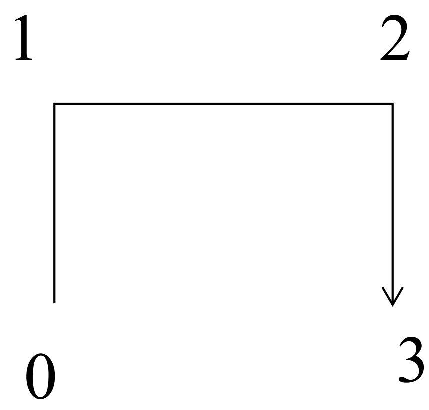


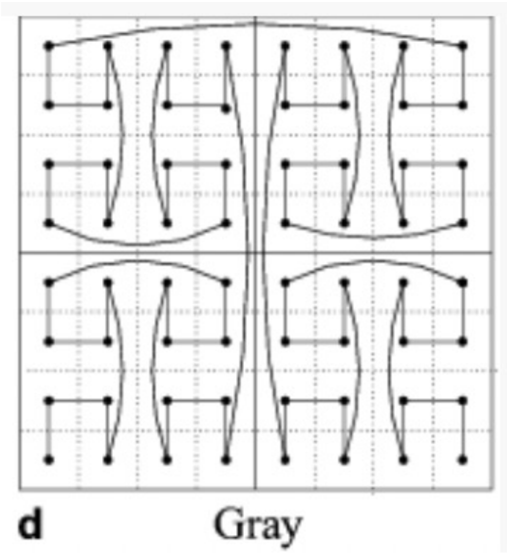
- Easy and elegant way to encode cells

- SIRO-DBMS (SDM) and Oracle use this order

Giuseppe Peano (/piˈaːnoʊ/) was an Italian mathematician (1858 – 1932)

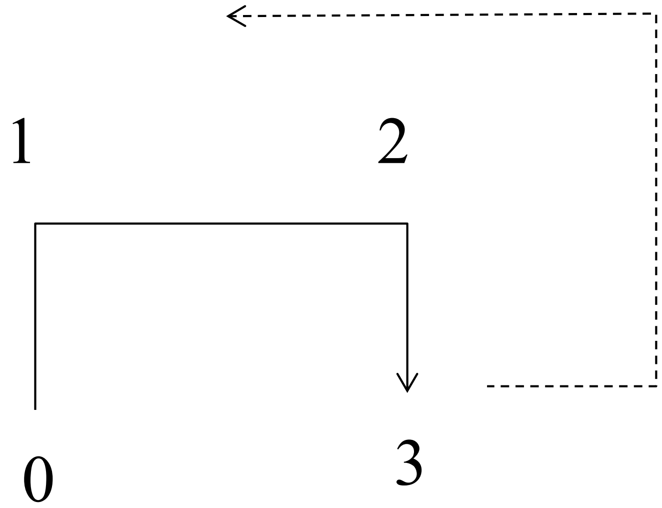
+ Gray Order

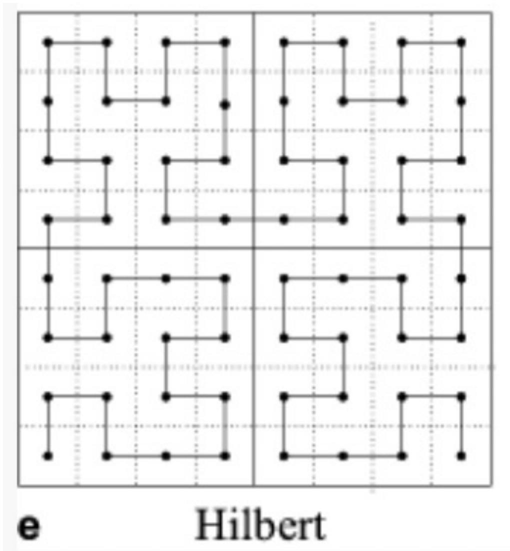




Frank Gray was a physicist and researcher at Bell Labs (1887 –1969)

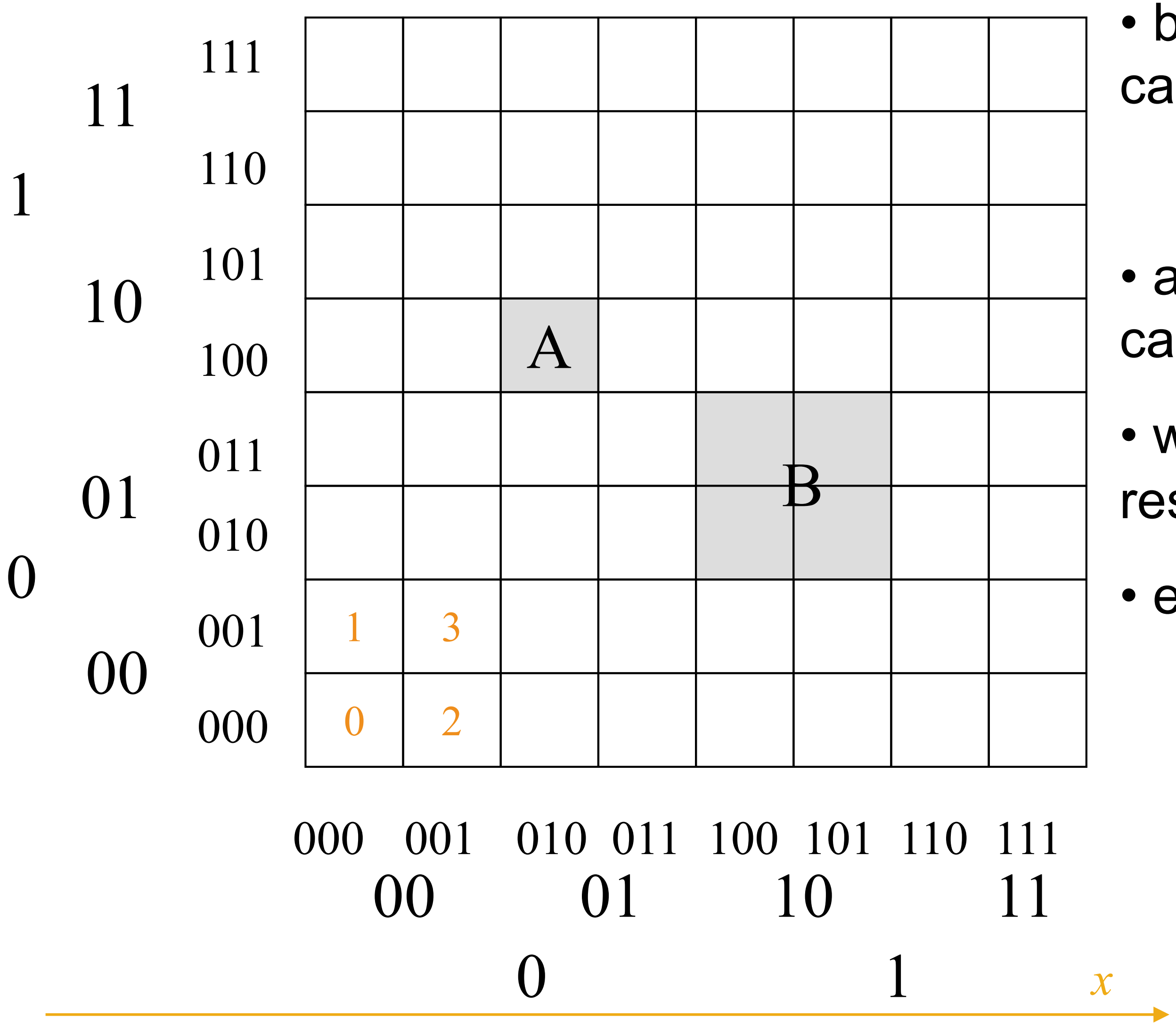
+ Hilbert Order





David Hilbert was a German mathematician (1862 –1943).

+ Bit-Interleaving



- bit interleaving for calculating z-values

  $x_0 y_0 x_1 y_1 \ldots$

- an alternative way for calculating z-values
- works fine with varying resolutions
- example for cell *A*

  $(120)_4 = (24)_{10}$

  A_x=010, A_y=100

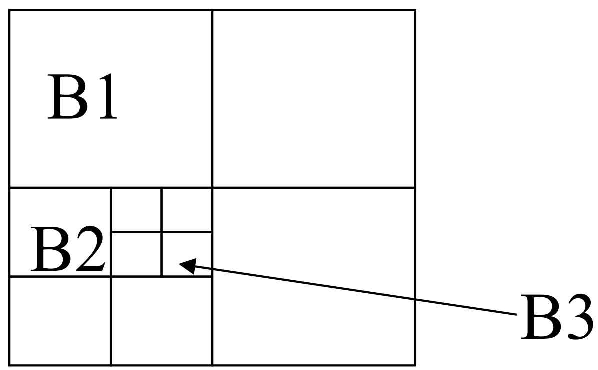
  $(011000)_2 = (24)_{10}$

+ Some Properties of Z-Values

- Variable length
  - Approximate at different levels
  - Appending ‘0’s at the end to unify z-value length
- Nesting Peano cells
  - $a = a_1 a_2 a_3 .. a_n$
  - $b = b_1 b_2 b_3 \dots b_n$
  - *a* is nested inside *b* if and only if
    - length(*a*) $\geq$ length(*b*), length(*a*) is the number of non-zero digits in z-value *a*
    - let k = length(a), $a_i = b_i$, $1 \leq i \leq k$

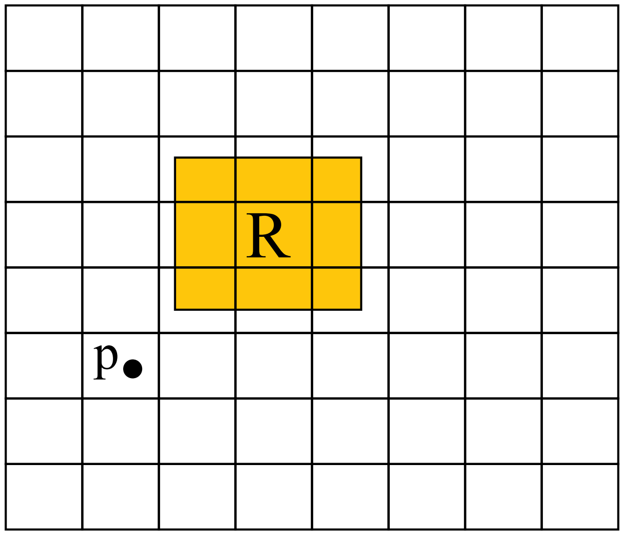
+ Using Z-Values and B-Tree (I)

- Motivation
  - Use standard B-tree to manage multidimensional data
- Basic Idea
  - a Peano cell corresponds to a bucket
  - Peano cells are of varying sizes
  - Z-values are managed by B-tree



+ Using Z-Values and B-Tree (II)

- Search
  - Point query: find the z-value for the unit Peano cell containing point *p*
  - Range query: find the min and max z-values for rectangle *R* *(or the z-values approximating R)*
- Insertion and deletion
- Compatibility of z-value indices
  - Origin and orientation
  - Spatial extent
  - Resolution

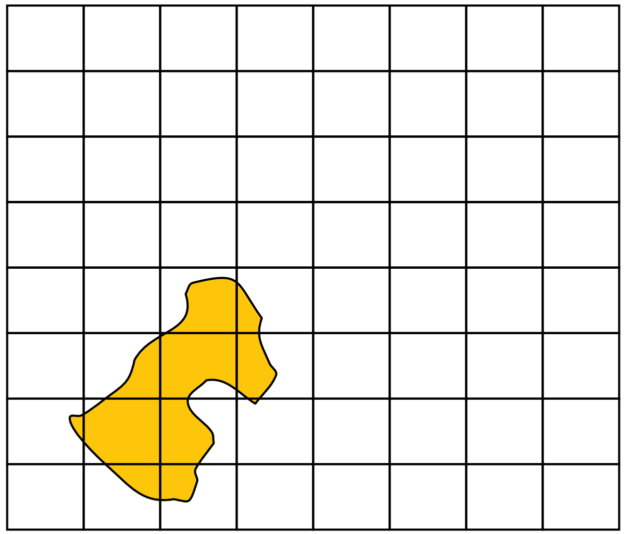


+ Data Access Methods

- One dimensional
  - Hashing and B-Trees
- Point data
  - Hashing: GRID and EXCELL
  - Hierarchical
    - Quadtree: point and region quadtrees
    - kd-Tree
    - Z-values and B-tree
- Polygon data
  - Transformation: end point mapping and z-values
  - Overlapping: R-tree and R*-tree
  - Clipping: R+-tree

+ Indexing Objects with Spatial Extent

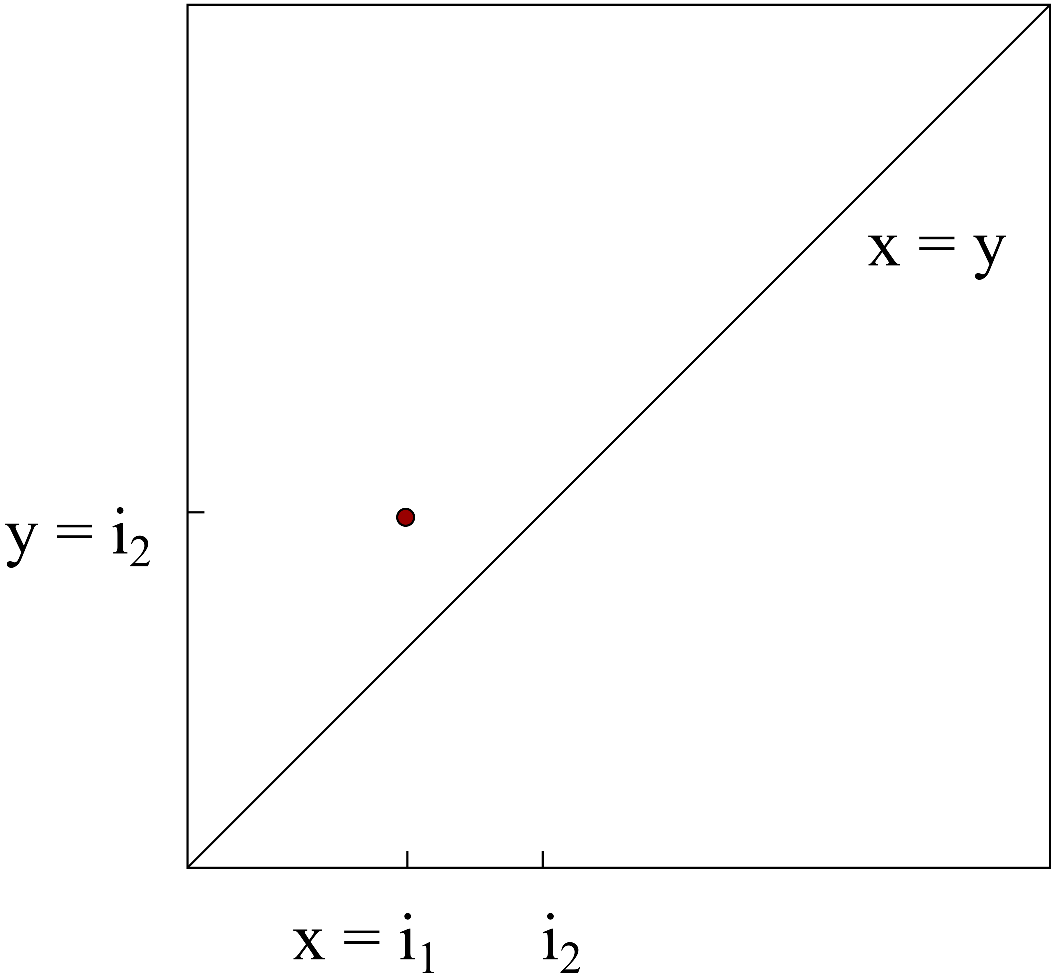
- Rectangles more difficult than points as they do not fall into a single cell of a bucket partition.
- Three strategies
  - Transformation
  - Overlapping regions
  - Clipping



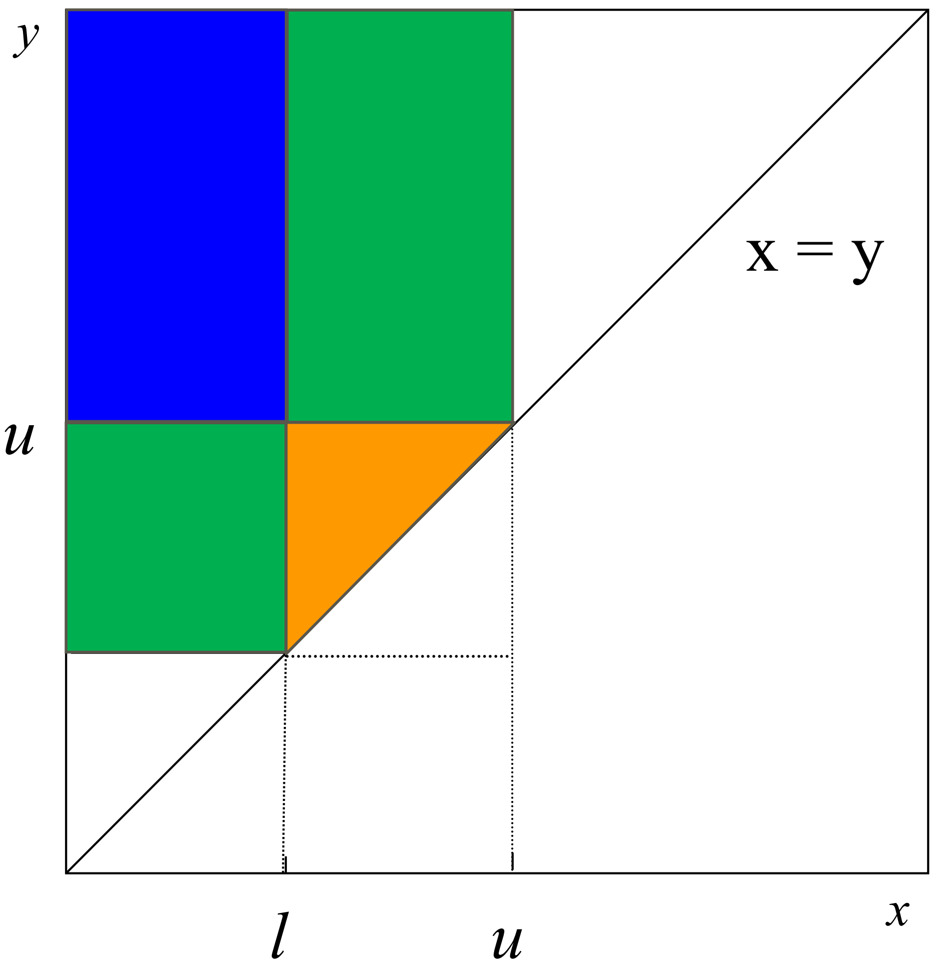
+ Transformation: High Dimensional Points

- Motivation
  - Points are easy to manage
- Basic Ideas
  - A rectangle in 2-D space can be mapped to a point in 4-D space
  - Using point access methods
- Two methods
  - Endpoint mapping, or midpoint mapping
    - $(x_{low}, y_{low}, x_{high}, y_{high}) \rightarrow (x_{low}, x_{high}, y_{low}, y_{high})$
    - $(x_{center}, y_{center}, x_{ext}, y_{ext}) \rightarrow (x_{center}, x_{ext}, y_{center}, y_{ext})$

+ Endpoint Mapping



+ Query Processing Using Endpoint Mapping

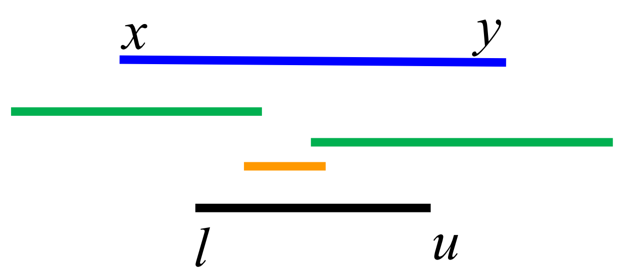


Given x-interval (*l*, *u*):

1) *intersection query*: find all x-intervals overlapping with (*l*, *u*).

2) *containment query*: find all x-intervals inside (*l*, *u*).

3) *enclosure query*: find all x-intervals enclosing (*l*, *u*)



+ Problems with Endpoint Mapping

- Points in the higher-D space are highly skewed
- Almost no relationship between the distances of two objects in the original space and the higher-D space
- A simple, intuitive query in the original space becomes complex and difficult to understand in the higher-D space
- Query processing in the higher-D space less efficient

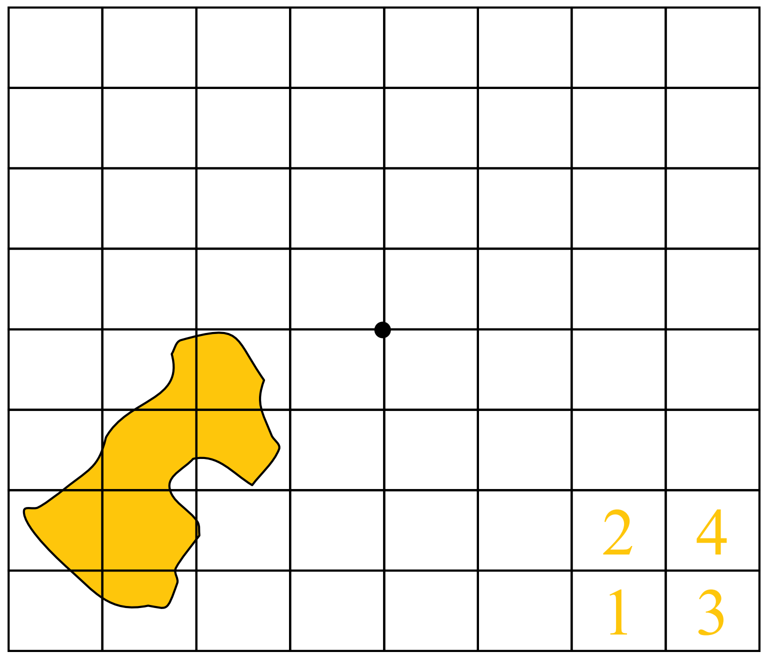
...conceptually very simple, but...

+ Transformation: Using Z-Ordering

- Motivation
  - Still using point access methods, but without drawbacks of the previous approach
- Basic idea
  - Instead of mapping a polygon into a point, decompose a polygon into a set of Peano cells and map each Peano cell into a number (i.e., z-value)

+ Transformation: Using Z-Ordering

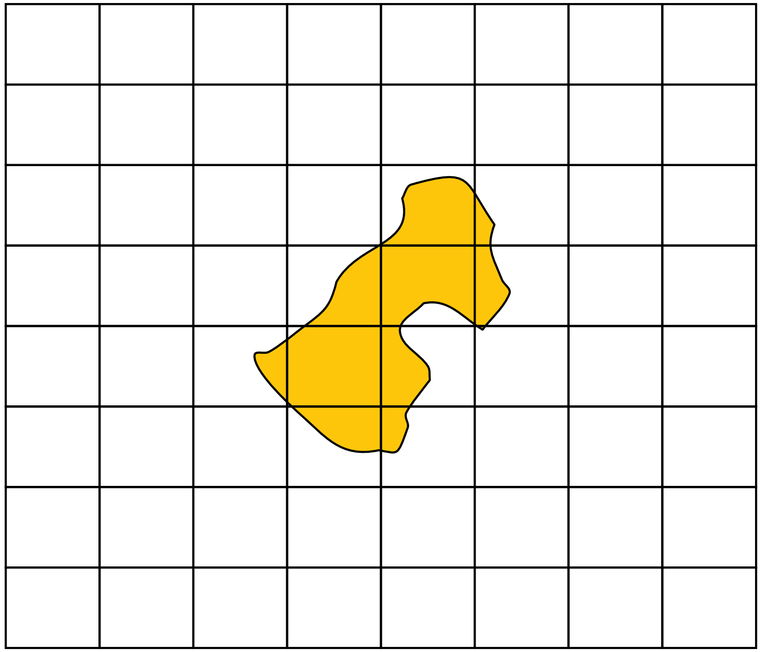
- Granularity
  - {11}, or {111, 112, 114}, or {111, 1121, 1123, 1124, 1141, 1142}
- When decomposition stops
  - Current cell either fully out or in the polygon
  - Reached the "resolution"



...the entire space is 1.

+ Redundancy in Z-Ordering

- Finer granularity
  - √ Improves approximation accuracy
  - √ Can reduce the number of "false hits"
  - × Too many index entries degrade query performance because of inflated index table
  - × May identify the same object multiple times in spatial query processing
- The 4-Key method



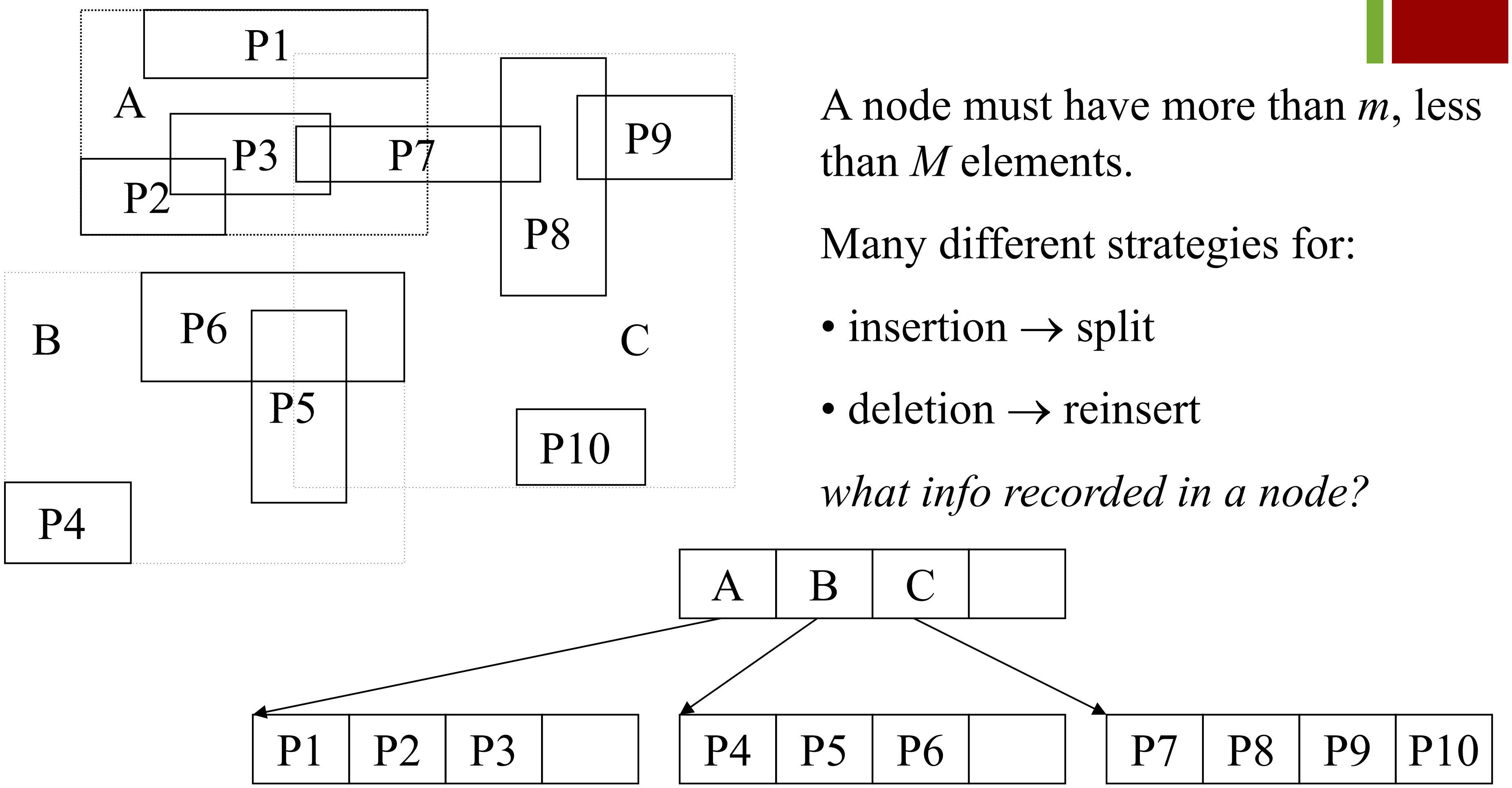
+ Query Processing Using Z-Values

- Key techniques
  - Identification of nesting Peano cells by their z-values
  - Filter-and-refine
- Point query
- Window query
- Within buffer
- Spatial join query

+ Overlapping Regions

- Motivation
  - Single index entry for a polygon
- Basic ideas
  - One object (or its key) in one bucket only
  - Cell boundary calculated according to polygons inside the cell
  - Allow overlapping cells: *inevitable*!
- Problems
  - Multiple cells need to be examined to search an object
  - Where to insert?

+ R-Tree and R*-Tree



A node must have more than *m*, less than *M* elements.

Many different strategies for:

- insertion → split
- deletion → reinsert

what info recorded in a node?

+ Query Processing Using R-Trees

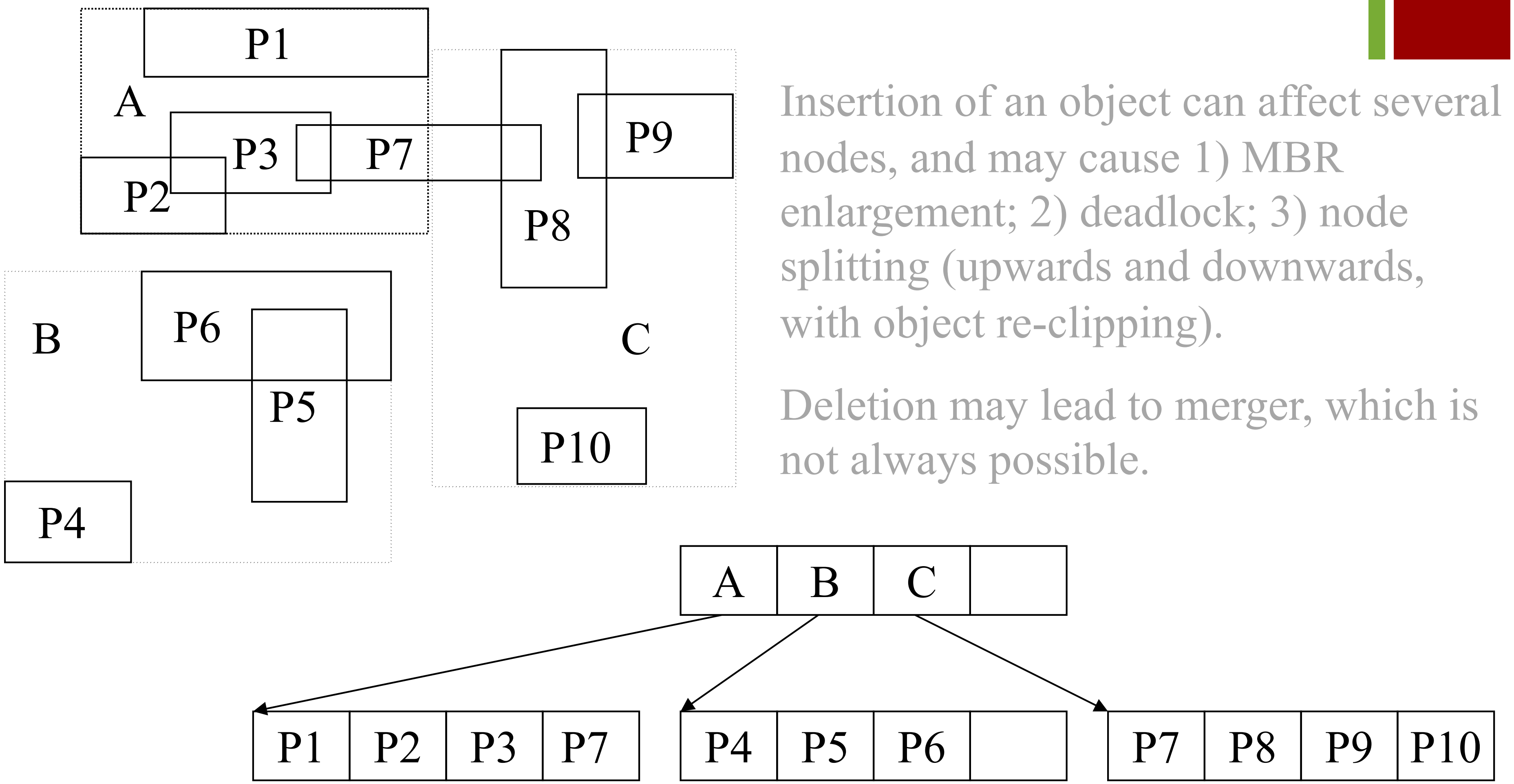
- A node records the MBR of all objects in the subtree rooted from the node
- Point query
- Window query
- Within buffer
- Spatial join query

…home work

+ Clipping

- Motivation
  - Single search path for a point query
- Basic ideas
  - Clipping polygon at cell boundaries
  - Allowing one polygon in multiple cells
- Problems
  - Multiple index entries for an object (increased search time, overflow more likely)
  - Enlargement deadlock
  - Cascading splitting

+ R⁺-Tree



Insertion of an object can affect several nodes, and may cause 1) MBR enlargement; 2) deadlock; 3) node splitting (upwards and downwards, with object re-clipping).

Deletion may lead to merger, which is not always possible.

+ Query Processing Using R$^+$-Trees

- Point query
- Window query
- Within buffer
- Spatial join query

…home work

+ Data Access Methods

- One dimensional
  - Hashing and B-Trees
- Point data
  - Hashing: GRID and EXCELL
  - Hierarchical
    - Quadtree: point and region quadtrees
    - kd-Tree
    - Z-values and B-tree
- Polygon data
  - Transformation: end point mapping and z-values
  - Overlapping: R-tree and R*-tree
  - Clipping: R+-tree

+ Indexing High Dimensional Data

- GIS applications in 2- or 3-D only
- Multimedia DB can have data with several hundred dimensions.
- While point/polygon access methods can be generalized for higher-D applications, they may be not efficient
- High-D indexing is a hard problem