Big Data Summer School

Indexing

Outline

- Datasets
  - Trajectory
    - Car 1: (time, location, Busy/Free)
    - Car 2: (time, location, Busy/Free)
    - … …
  - Order
    - Passenger: (time, source, destination)
  - Road network
    - Graphs
      - Nodes
      - Edges (roads)

Outline

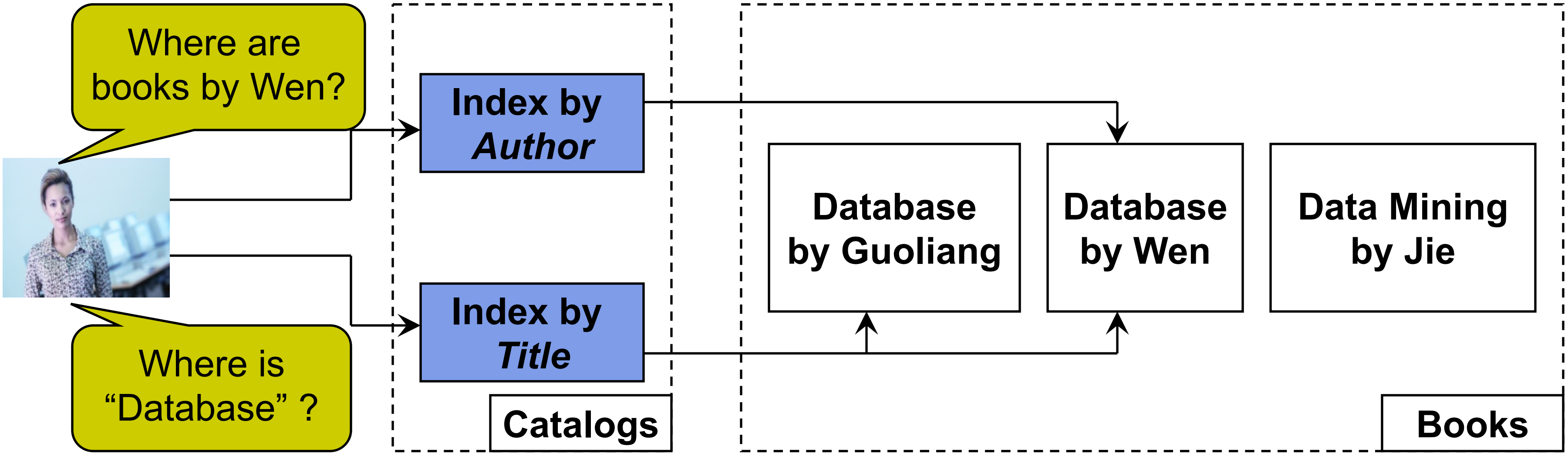
- Applications
  - Given an order, how to find k nearest cars?
    - Euclidean distance $\mathrm{d}(\mathbf{p},\mathbf{q}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}.$
    - Road distance (graph distance)
  - How to get the best route from source to destination?
    - Graph algorithms
  - Where to park after finishing an order?
    - Clustering
    - Approximate query processing

Outline

- Techniques to Handel Big Data
  - Indexing
    - B-tree, R-tree, KD-tree
  - Clustering
    - K means, DBScan
  - Graph
    - Dijkstra, G-tree,GraphChi
  - Distributed Computing
    - Spark SQL
  - Approximate Query Processing
    - Sampling

The Concept of Index

- Searching a Book…



- Index
  - A data structure that helps us find data quickly

- Note
  - Can be a separate structure (we may call it the index file) or in the records themselves.
  - Usually sorted on some attribute

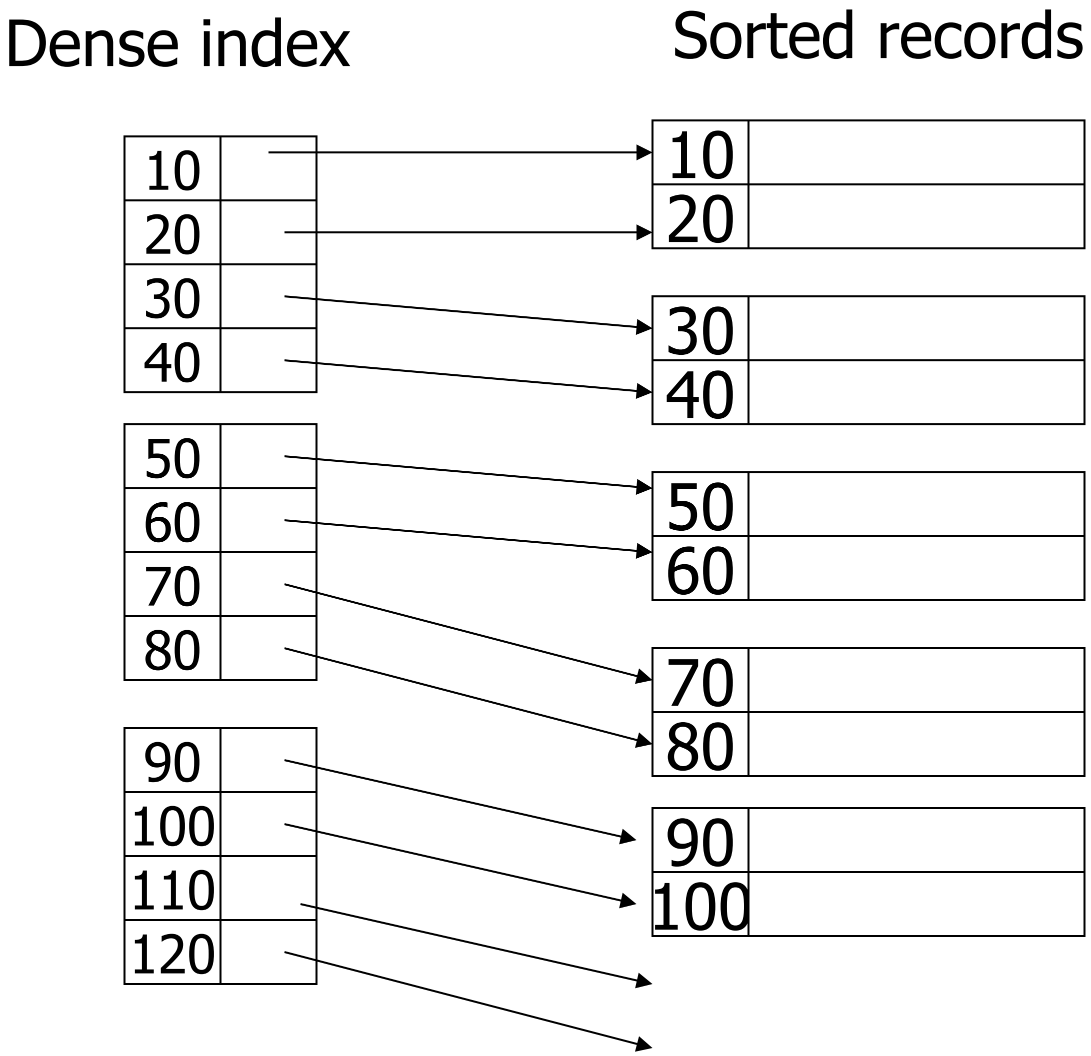
Query, Key and Search Key

- Queries
  - Exact match (point query)
    - Q1: Find me the book with the name “Database”
  - Range query
    - Q2: Find me the books published between year 2003-2005

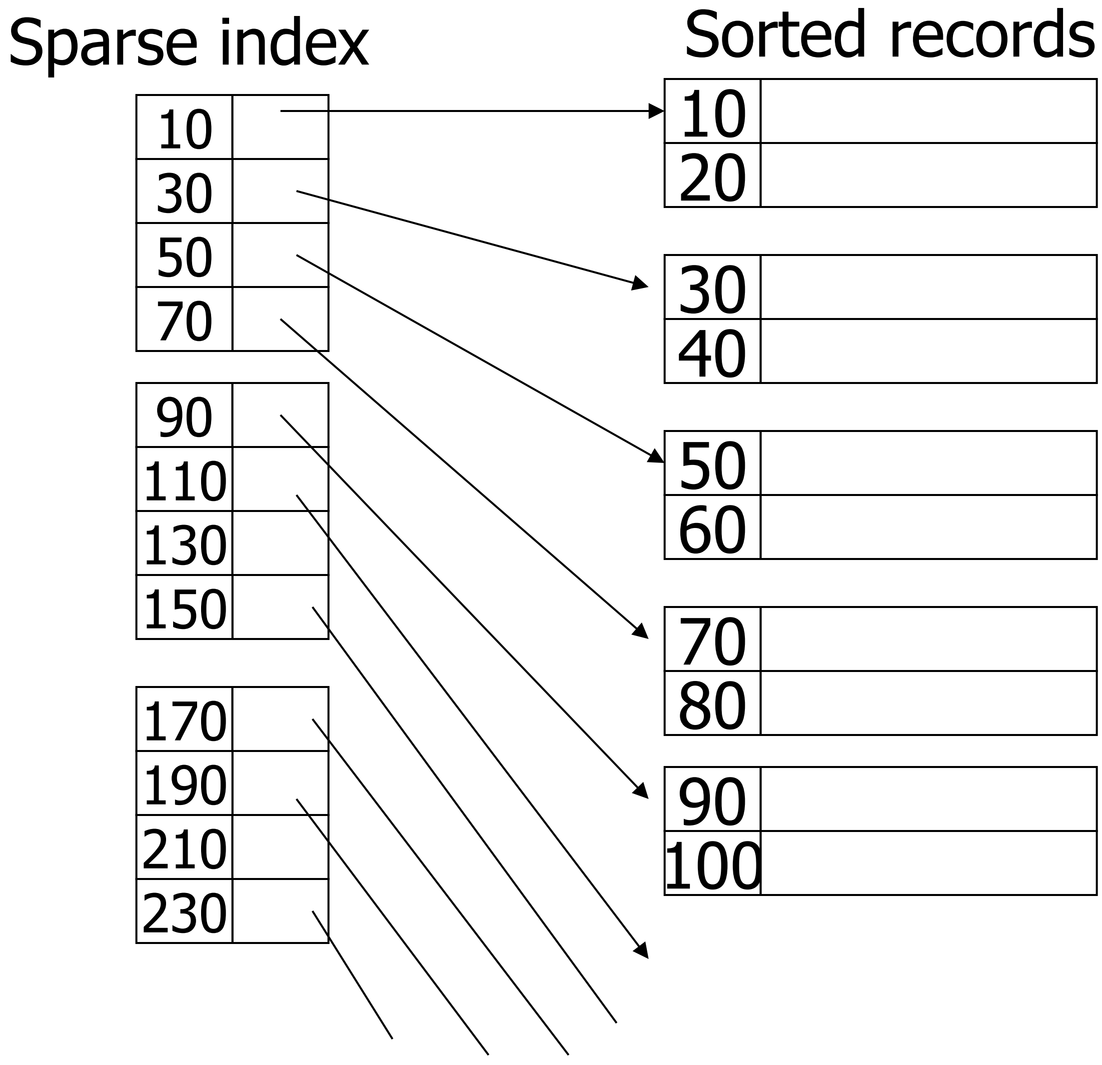
- Searching methods
  - Sequential scan — too expensive
  - Through index – if records are sorted on some attribute, we may do a binary search
    - If sorted on “book name”, then we can do binary search for Q1
    - If sorted on “year published”, then we can do binary search for Q2

- Key vs. Search key
  - Key: the indexed attribute
  - Search key: the attribute queried on

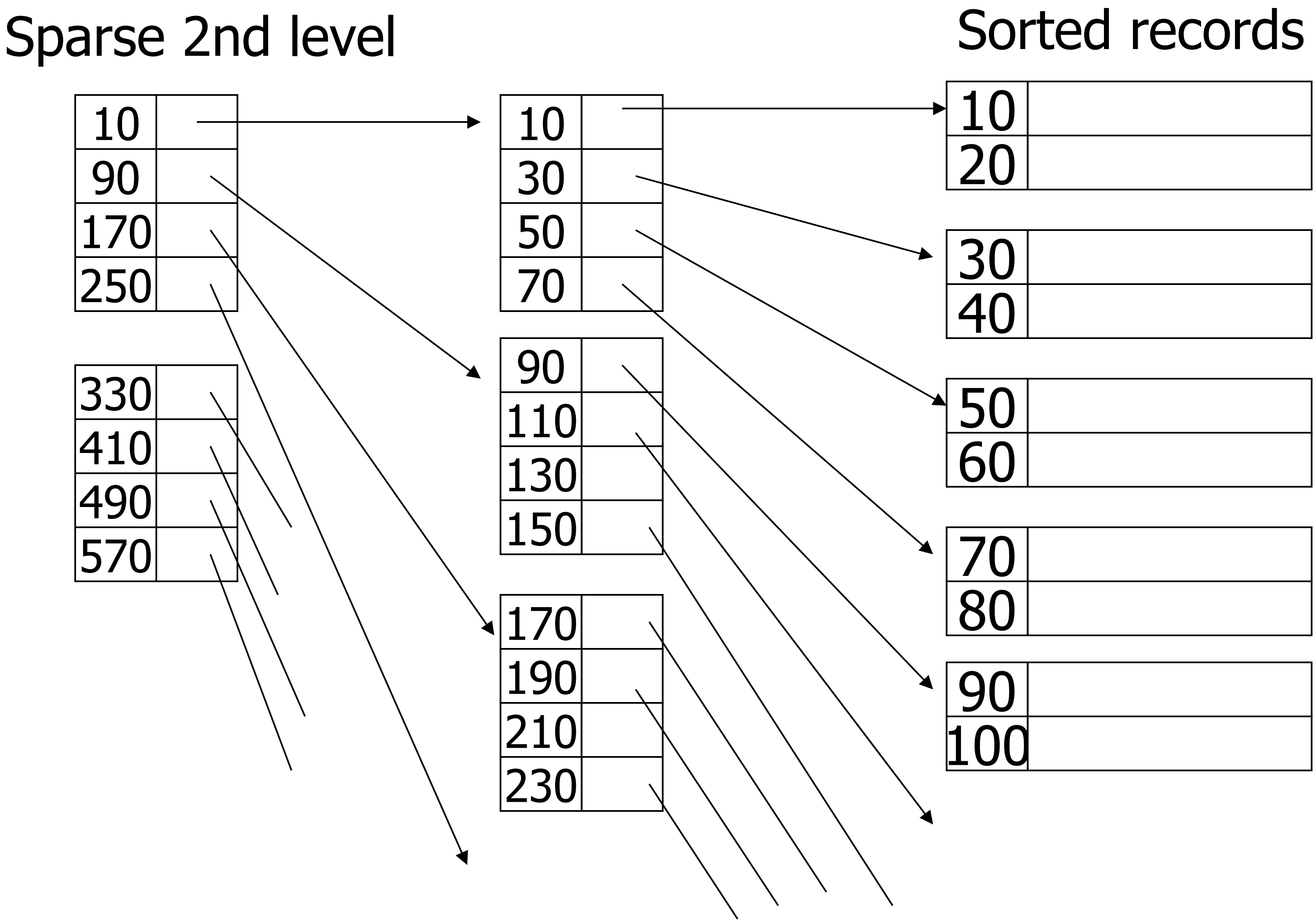
Simple Index File (Clustered, Dense)



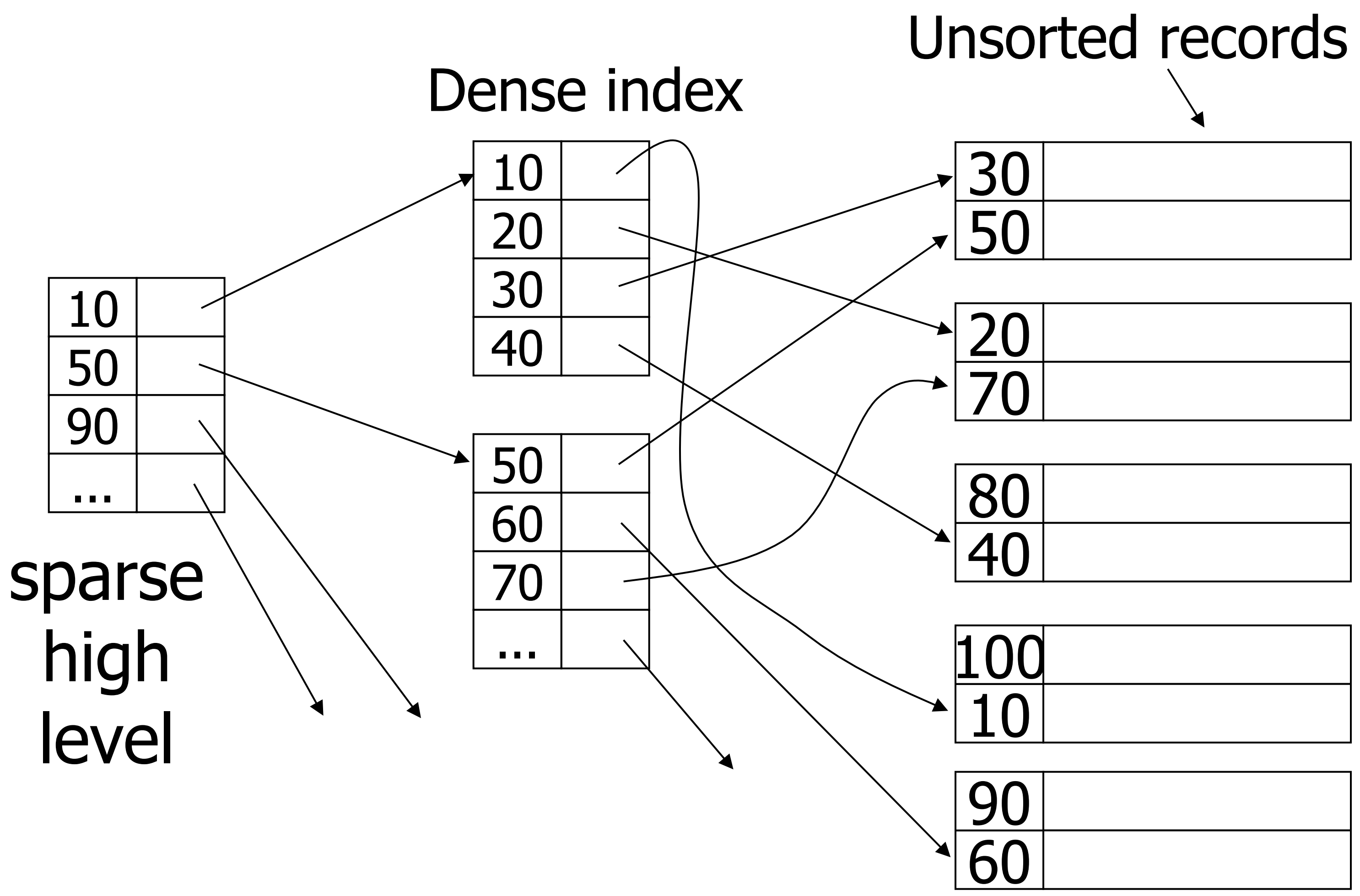
Simple Index File (Clustered, Sparse)



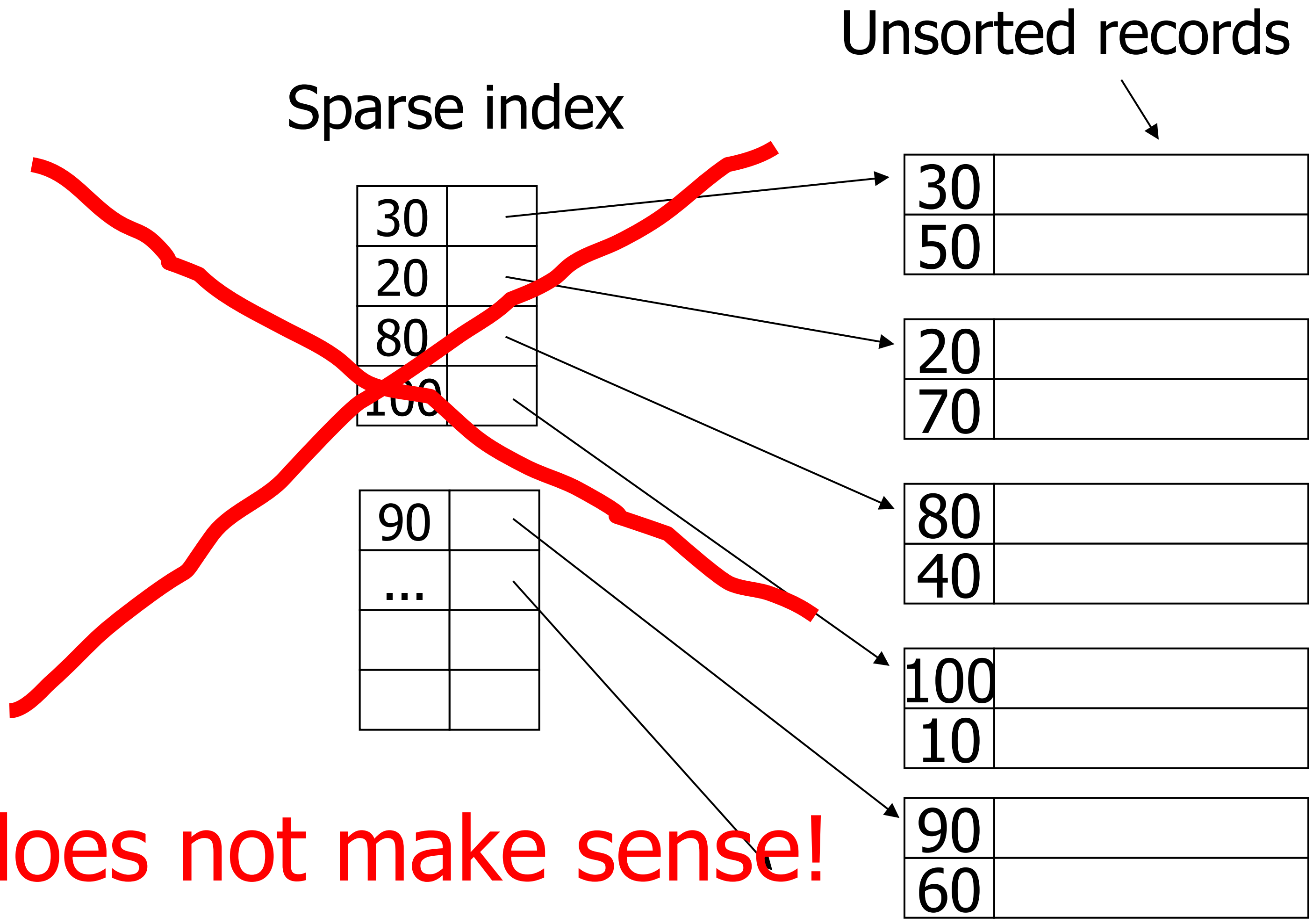
Simple Index File (Clustered, Multi-level)



Simple Index File (Unclustered, Dense)



Simple Index File (Unclustered, Sparse ?)

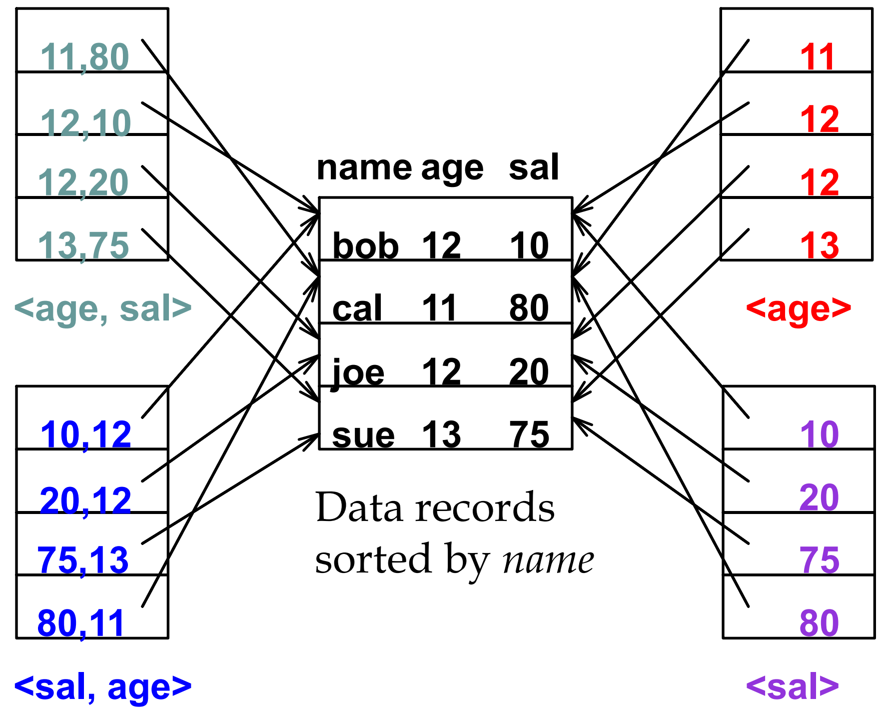


does not make sense!

Indexes on Composite Keys

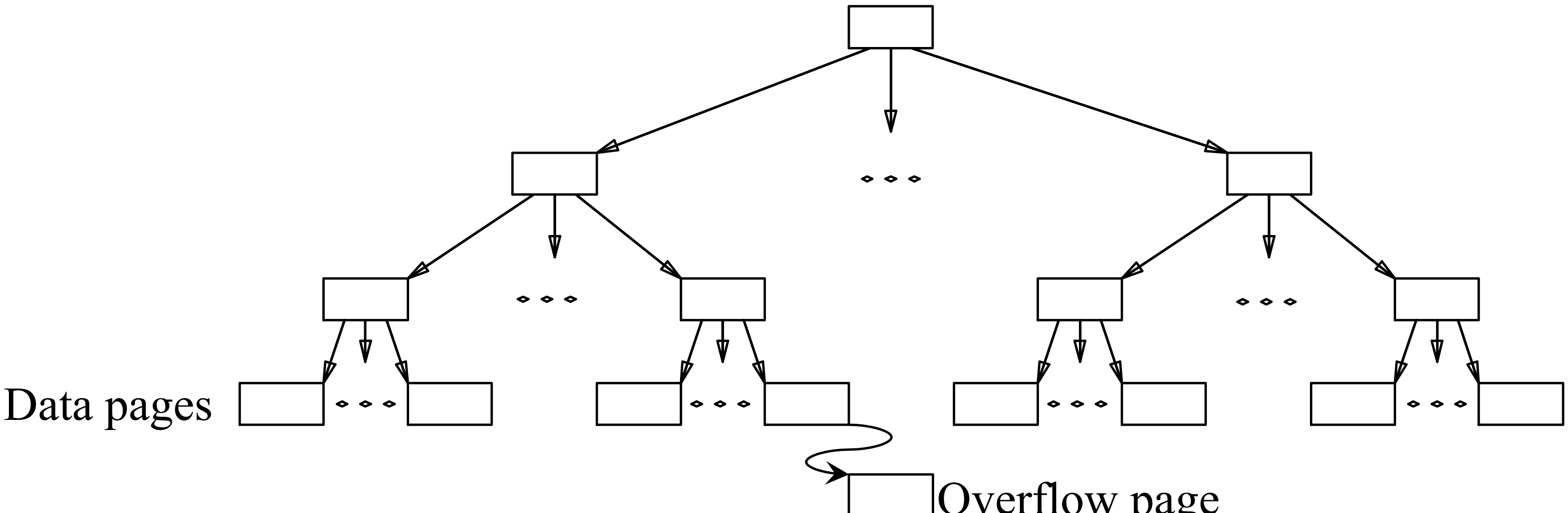
- Q3: age=20 & sal=10
- Index on two or more attributes: entries are sorted first on the first attribute, then on the second attribute, the third …
- Q4: age=20 & sal>10
- Q5: sal=10 & age>20
- Note
  - Different indexes are useful for different queries

Examples of composite key indexes using lexicographic order.



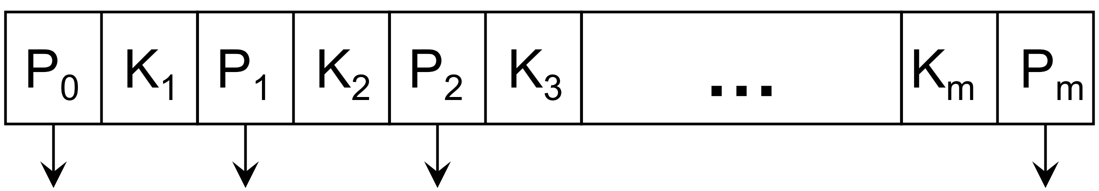
Indexed sequential access method (ISAM)

- Tree structured index
- Support queries
  - Point queries
  - Range queries
- Problems
  - Static: inefficient for insertions and deletions

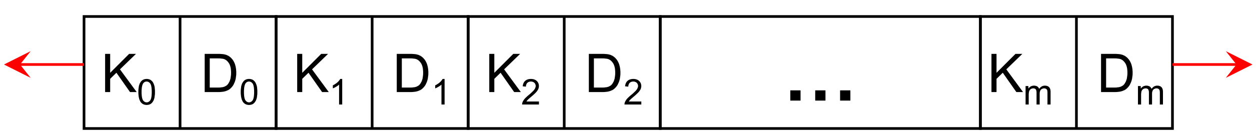


The B^+-Tree: A Dynamic Index Structure

- Grows and shrinks dynamically
- Minimum 50% occupancy (except for root).
  - Each node contains $d <= m <= 2d$ entries. The parameter d is called the ***order of the tree***.
- Height-balanced
  - Insert/delete at $log_f N$ cost (f = fanout, N = No. leaf pages)
- Pointers to sibling pages
  - Non-leaf pages (internal pages)

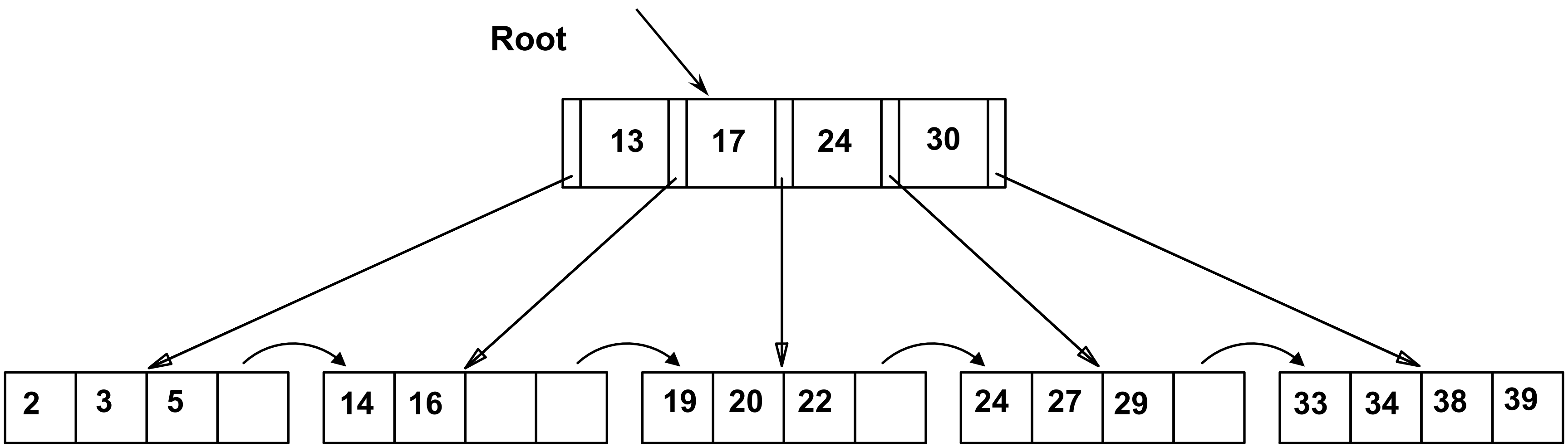


  - Leaf pages
    - If directory page, same as non-leaf pages; pointers point to data page addresses
    - If data page



Searching in a B^+-Tree

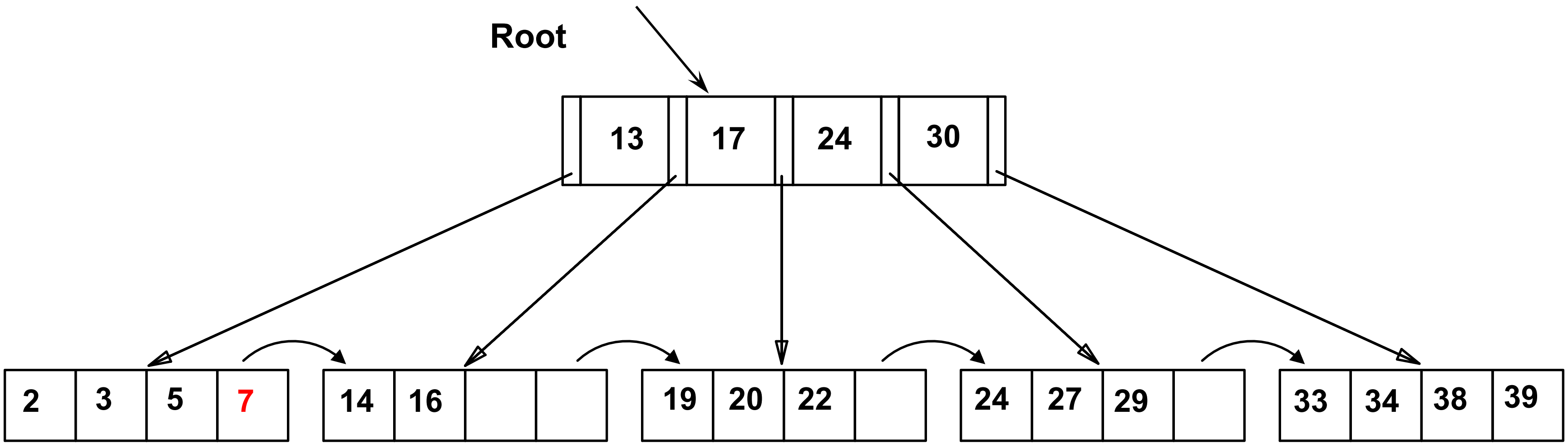
- Search begins at root, and key comparisons direct it to a leaf (as in ISAM)
- Search for 5, 15, all data entries >= 24 ...
- What about all entries <= 24 ?



Insertion in a B+-Tree

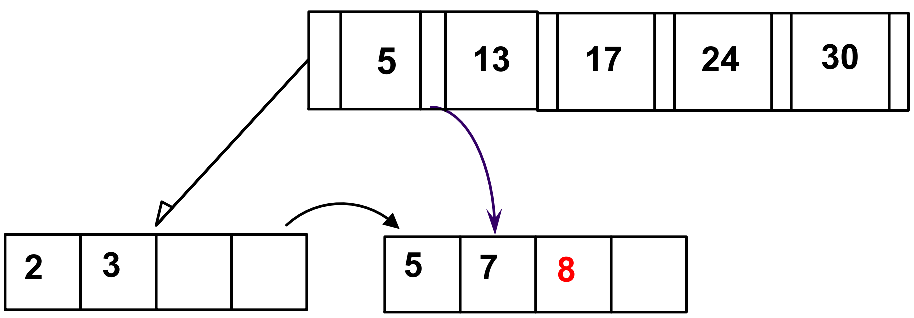
- Find correct leaf *L*
- Put data entry onto *L*
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, ***copy up*** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - *To split index node*, redistribute entries evenly, but ***push up*** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Inserting 7 & 8 into the B+-Tree



- Observe how minimum occupancy is guaranteed in both leaf and index pg splits

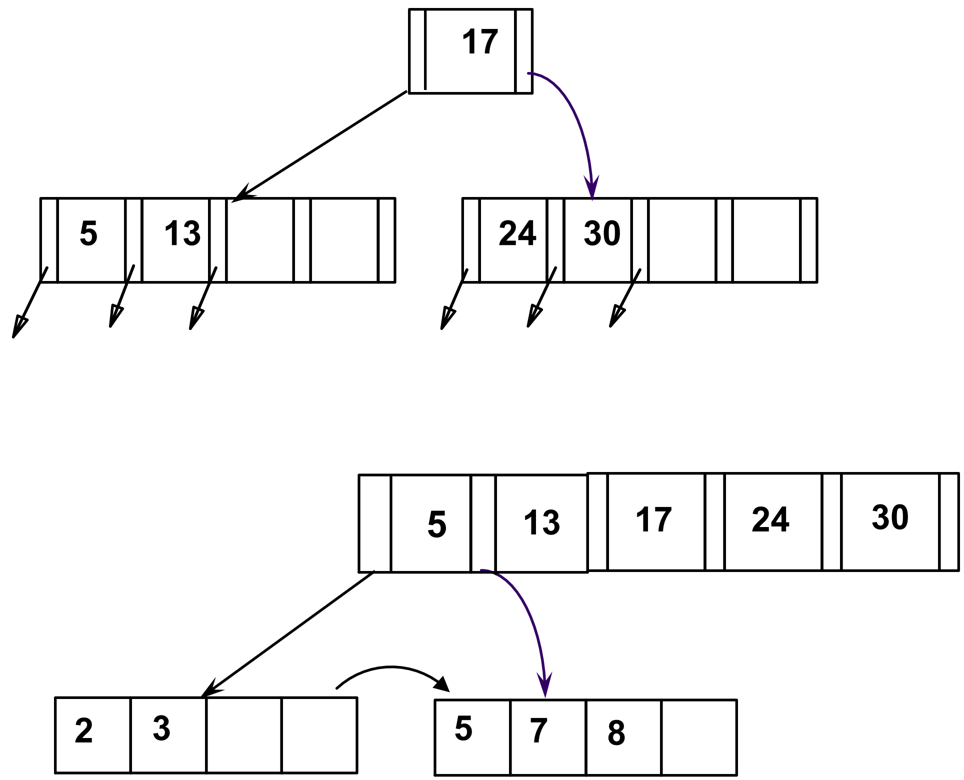
(Note that 5 is copied up and continues to appear in the leaf.)



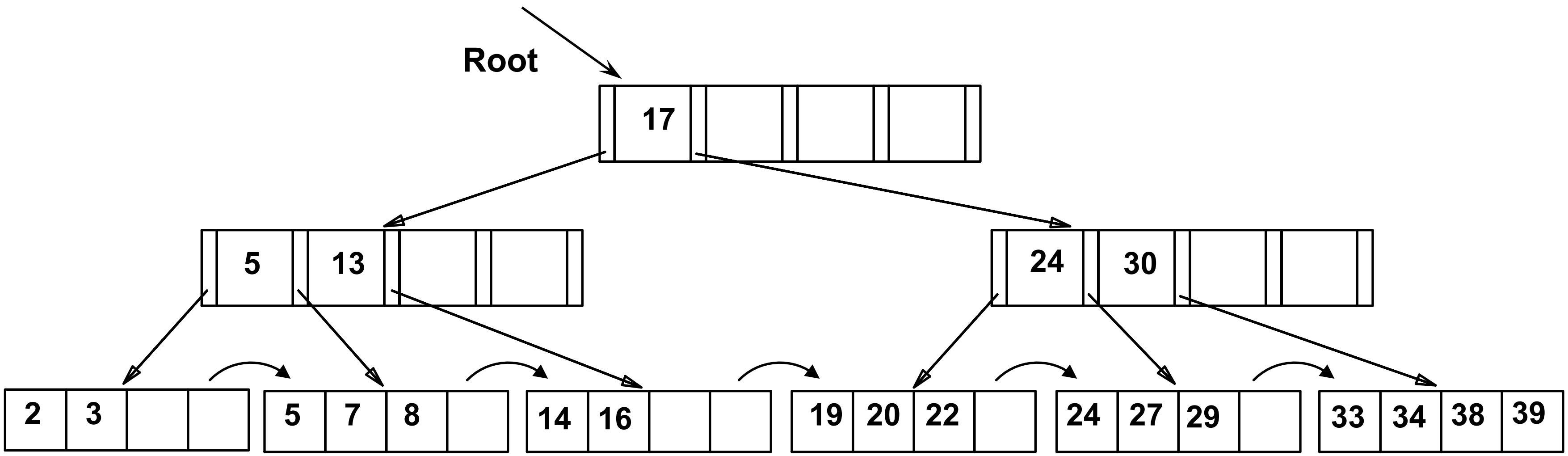
Inserting 8 into the B+-Tree (continued)

- Note the difference between ***copy up*** and ***push up***

(17 is pushed up and only appears once in the index. Contrast this with a leaf split.)



The B$^+$-Tree After Inserting 8

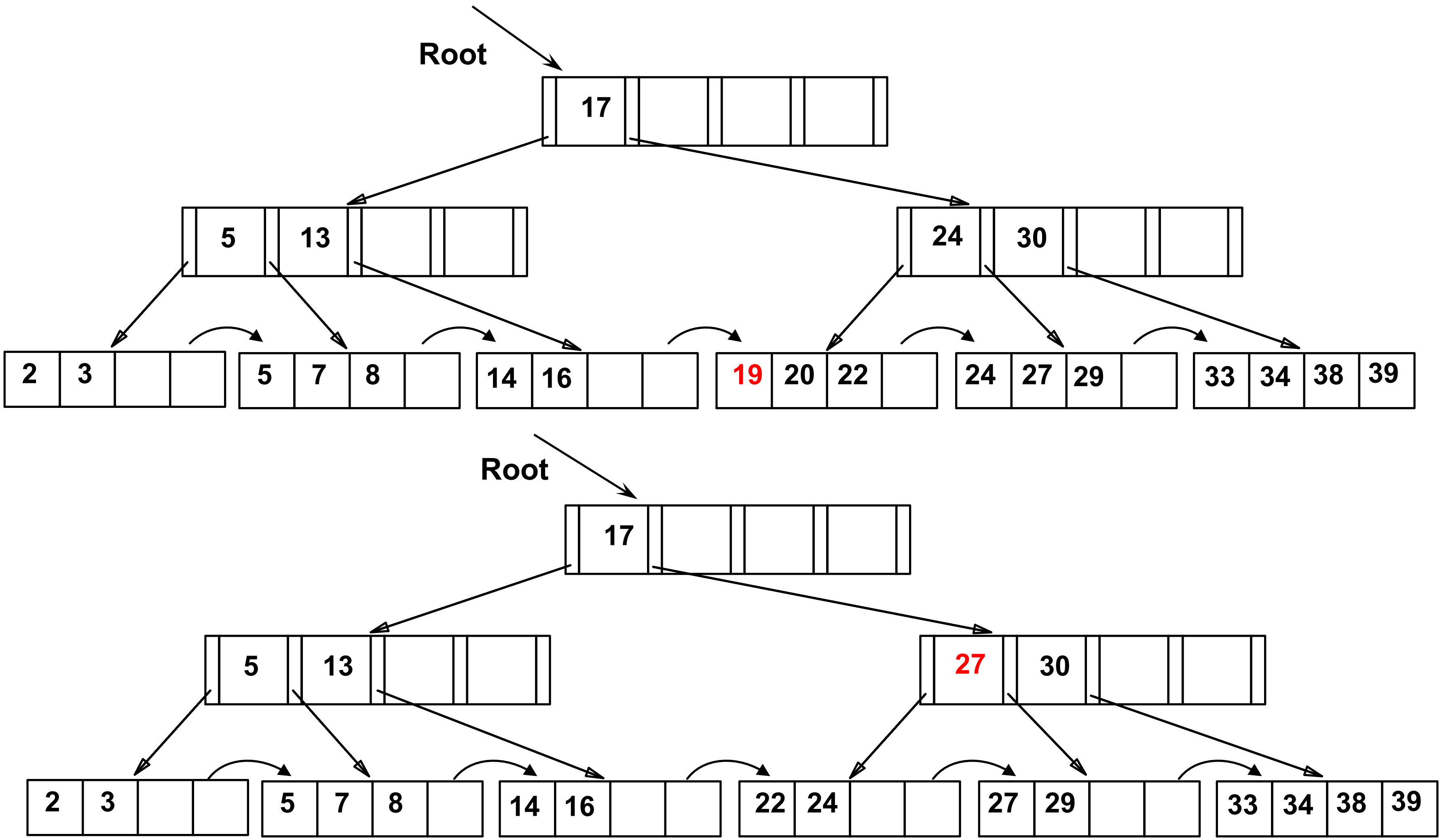


- Note that root was split, leading to increase in height
- We can avoid splitting by re-distributing entries. However, this is usually not done in practice. Why?

Deletion in a B^+-Tree

- Start at root, find leaf *L* where the entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to *redistribute*, borrowing from *sibling* *(adjacent node with same parent as L)*.
    - If redistribution fails, *merge* *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

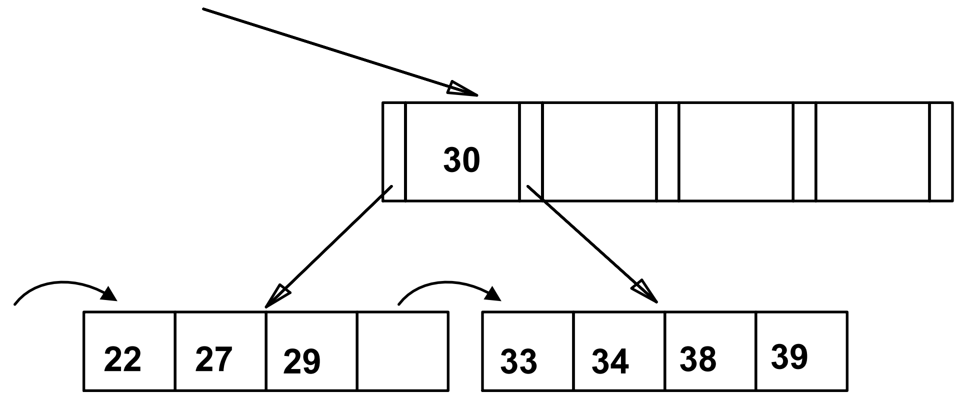
Deleting 19 & 20 from the B^+-Tree

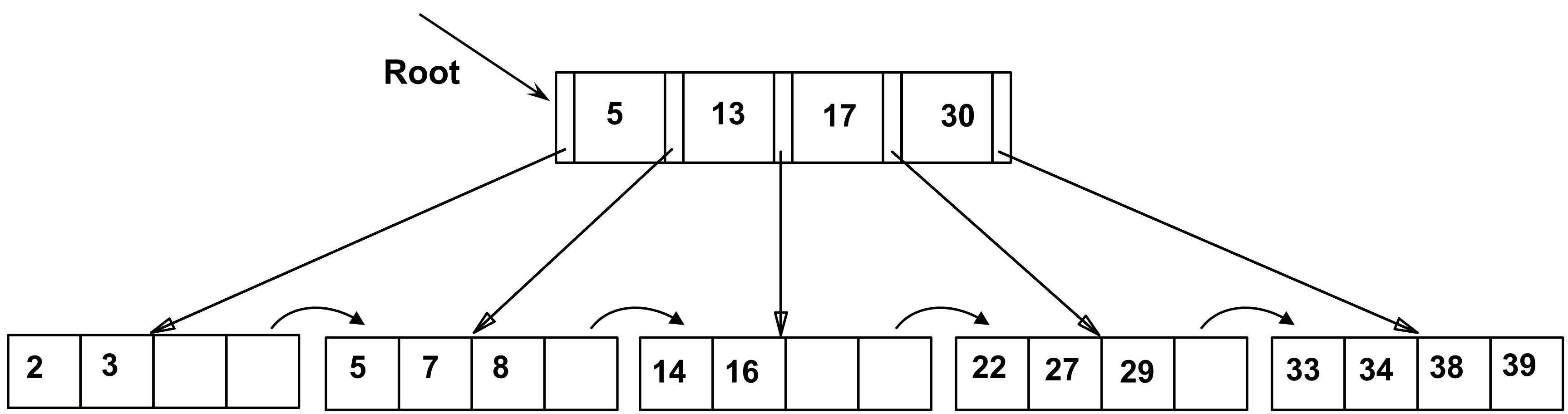


- Deleting 20 is done with re-distribution. Note how the middle key is ***copied up***

Deleting 24 from the B+-Tree

- Merge happens
- Observe *‘**toss**’* of index entry (on right), and *‘**pull down**’* of index entry (below).
- Note the decrease in height

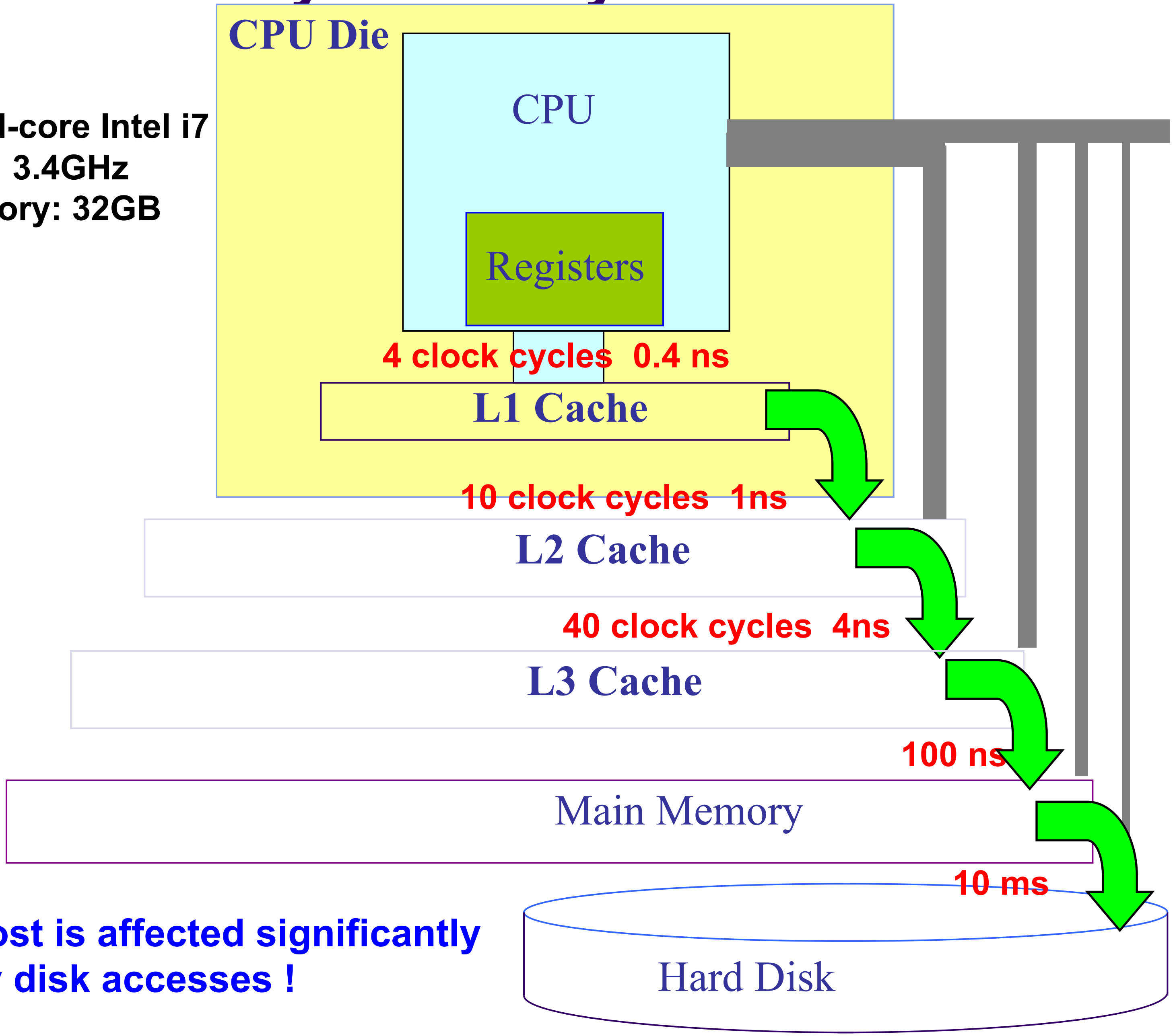




How to set b?

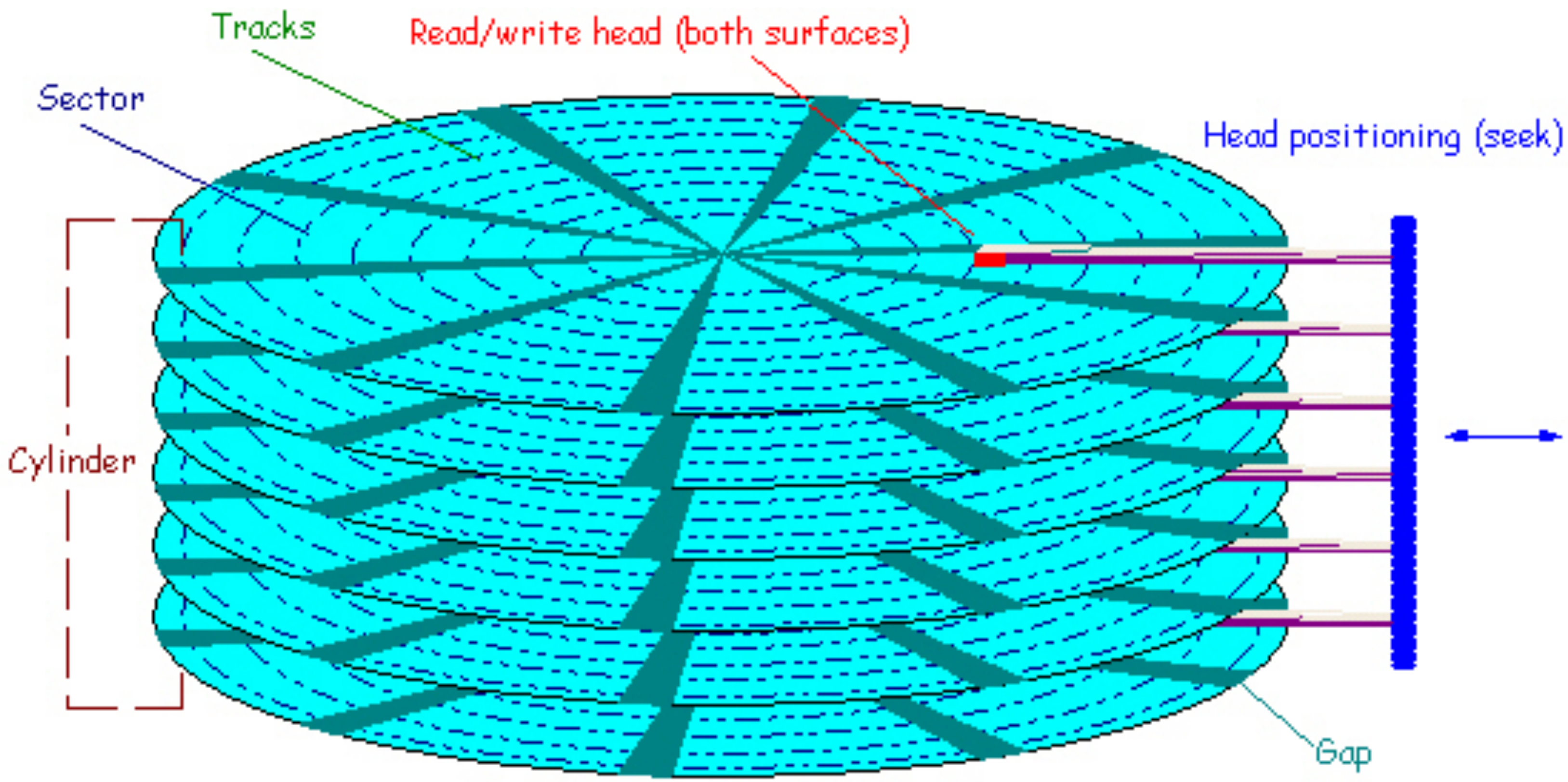
The Memory Hierarchy

iMac
Quad-core Intel i7
CPU: 3.4GHz
Memory: 32GB



Cost is affected significantly by disk accesses !

The Hard (Magnetic) disk



- The time for a ***disk block access***, or ***disk page access*** or ***disk I/O***
  access time = seek time + rotational delay + transfer time
- Segate Desktop HDD.15: 4TB
  - Seek time: 8.5 msec
  - Rotational delay: 5.1 msec
  - Transfer rate: 146MB/sec, that is, 0.027msec/4KB

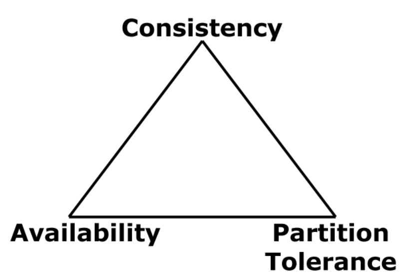
The Cost Model

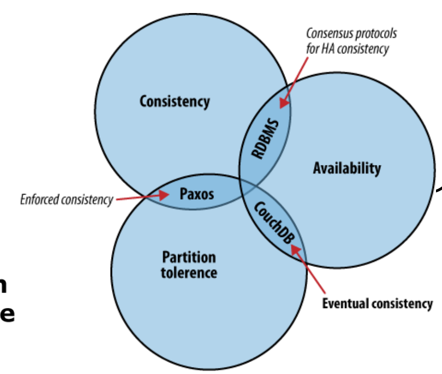
- Cost measure: *number of page accesses*
- Objective
  - A simple way to estimate the cost (in terms of execution time) of database operations
- Reason
  - Page access cost is usually the dominant cost of database operations
  - An accurate model is too complex for analyzing algorithms
- Note
  - This cost model is for disk based databases; ***NOT*** applicable to main memory databases
  - Blocked access: sequential access

Summary of the B^+-Tree

- A dynamic structure – height balanced – **robustness**
- **Scalability**
  - Typical order: 100. Typical fill-factor: 67%.
    - average fanout = 133
  - Typical capacities (root at Level 1, and has 133 entries)
    - Level 5: 133^4 = 312,900,700 records
    - Level 4: 133^3 = 2,352,637 records
  - Can often hold top levels in buffer pool:
    - Level 1 = 1 page = 8 Kbytes
    - Level 2 = 133 pages = 1 Mbyte
    - Level 3 = 17,689 pages = 133 Mbytes
- Efficient point and range queries – **performance**
- **Concurrency**
  - **CAP**

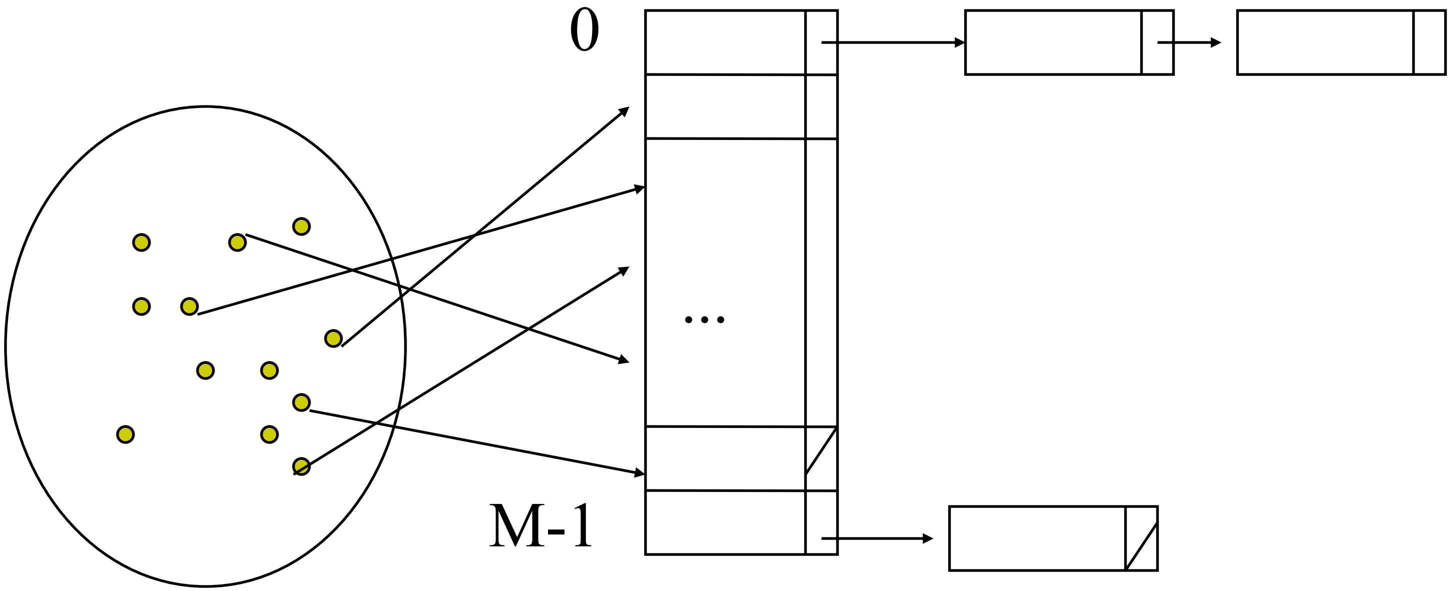
Essential properties of a DBMS





Other Basic Indexes: Hash Tables

- Static hashing: static table size, with overflow chains



- Extendible hashing: dynamic table size, with no overflows
- Efficient point query, but inefficient range query

Other Basic Indexes: Bitmap

- Bitmap with bit position to indicate the presence of a value

gender

M	F
1	0
1	0
0	1
1	0

cusid	name	gender	rating
112	Joe	M	3
115	Ram	M	5
119	Sue	F	5
120	Woo	M	4

rating

1	2	3	4	5
0	0	1	0	0
0	0	0	0	1
0	0	0	0	1
0	0	0	1	0

- Advantages
  - Efficient bit-wise operations
  - Efficient for aggregation queries: counting bits with 1′s
  - More compact than trees-- amenable to the use of compression techniques
- Limitations
  - Only good for domain of small cardinality

B^+-Tree For All and Forever?

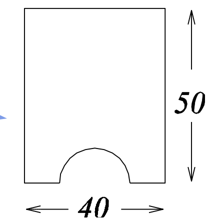
- Can the B+-tree being a single-dimensional index be used for emerging applications such as:
  - Spatial databases
  - High-dimensional databases
  - Temporal databases
  - Main memory databases
  - String databases
  - Genomic/sequence databases
  - ...

- Coming next … Indexing Multidimensional Data

Multidimensional Data

- Spatial data (low-dimensionality)
  - Geographic Information: Melbourne (37, 145)
  - Which city is at (30, 140)?

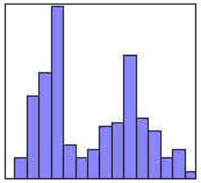
  - Computer Aided Design: width and height (40, 50)
  - Any part that has a width of 40 and height of 50?



- Records with multiple attributes (medium-dimensionality)
  - Employee (ID, age, score, salary, …)
  - Is there any employee whose
    age is under 25 and
    performance score is greater than 80 and
    salary is between 3000 and 5000

ID	Age	Score	Salary	…
…				

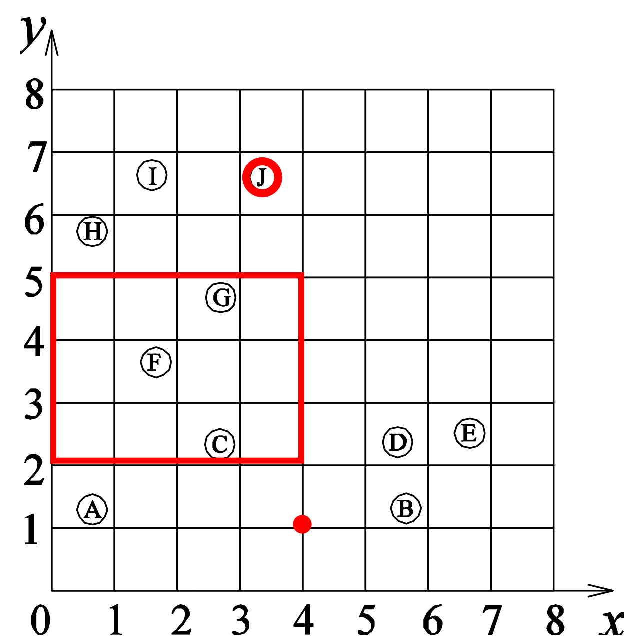
- Multimedia data (high-dimensionality)
  - Color histograms of images
  - Give me the most similar image to



  - Multimedia Features: color, shape, texture

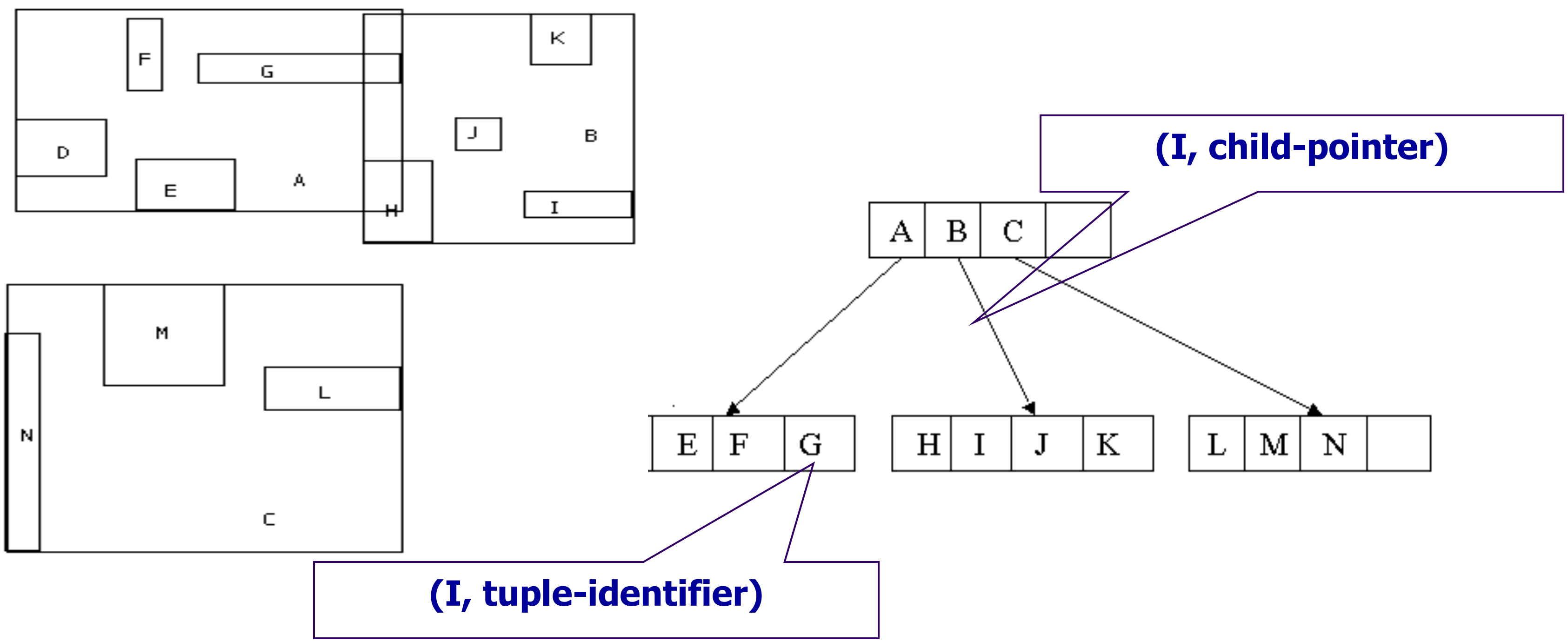
Multidimensional Queries

- Point query
  - Return the objects located at $Q(x_1, x_2, \ldots, x_d)$.
  - E.g. Q=(3.4, 6.6).
- Window query
  - Return all the objects enclosed or intersected by the hyper-rectangle $W\{[L_1, U_1], [L_2, U_2], \ldots, [L_d, U_d]\}$.
  - E.g. W={[0,2],[4,5]}
- K-Nearest Neighbor Query (KNN Query)
  - Return k objects whose distances to Q are no larger than any other object' distance to Q.
  - E.g. 3NN of Q=(4,1)

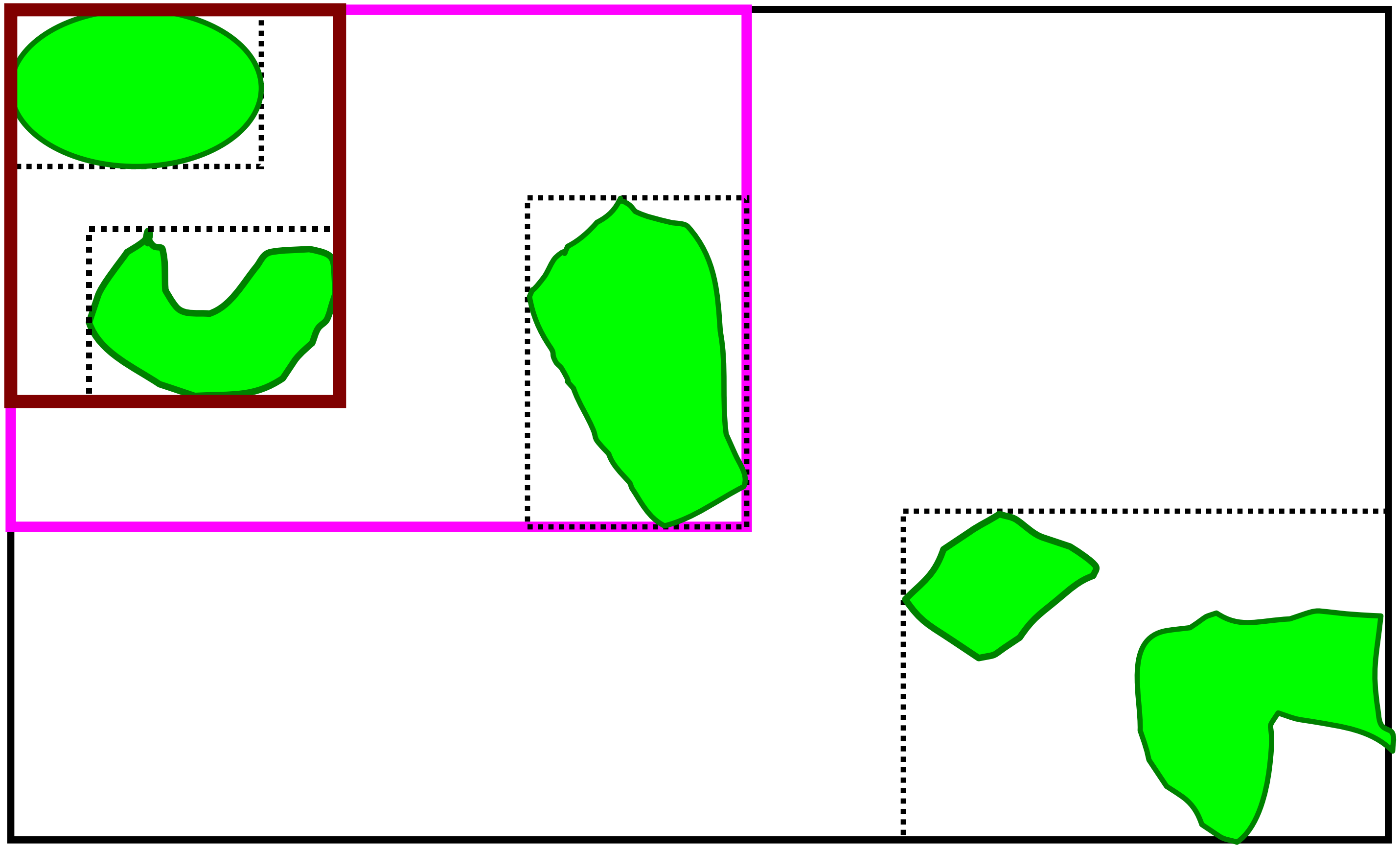


R-tree

- A balanced tree similar to a B+-tree;
- Each tuple has a unique identifier which can be used to retrieve it



Minimum Bounding Rectangle (MBR)

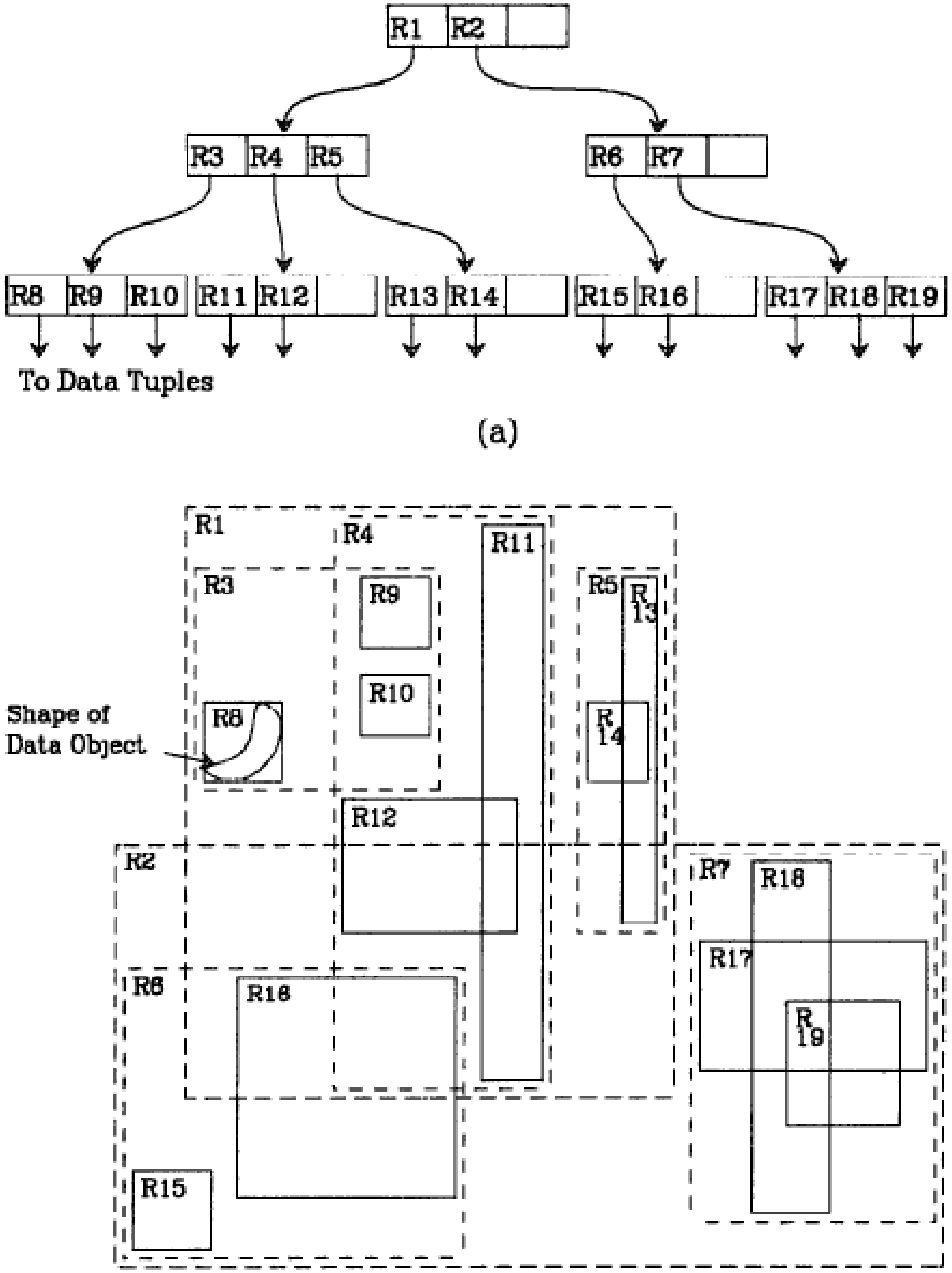


R-Trees: The Structure.

- Internal nodes : (MBR, child pointer)
  - MBR is minimum bounding rectangle
  - Pointer to all rectangles contained by the MBR.
- Leaf Nodes : (MBR , tuple-identifier)
  - MBR is minimum bounding rectangle
  - Tuple-identifier is a pointer to the data object.

Example

- 3 levels
- m=2, M=4



Properties of R-trees

An R-tree satisfies the following properties

- The root has at least two children unless it is a leaf
- Every non-leaf node has between m and M children unless it is the root
- Every leaf node contains between m and M entries unless it is the root
- All leaves appear on the same level

M: the maximum number of entries in a node

m: the minimum number of entries in a node

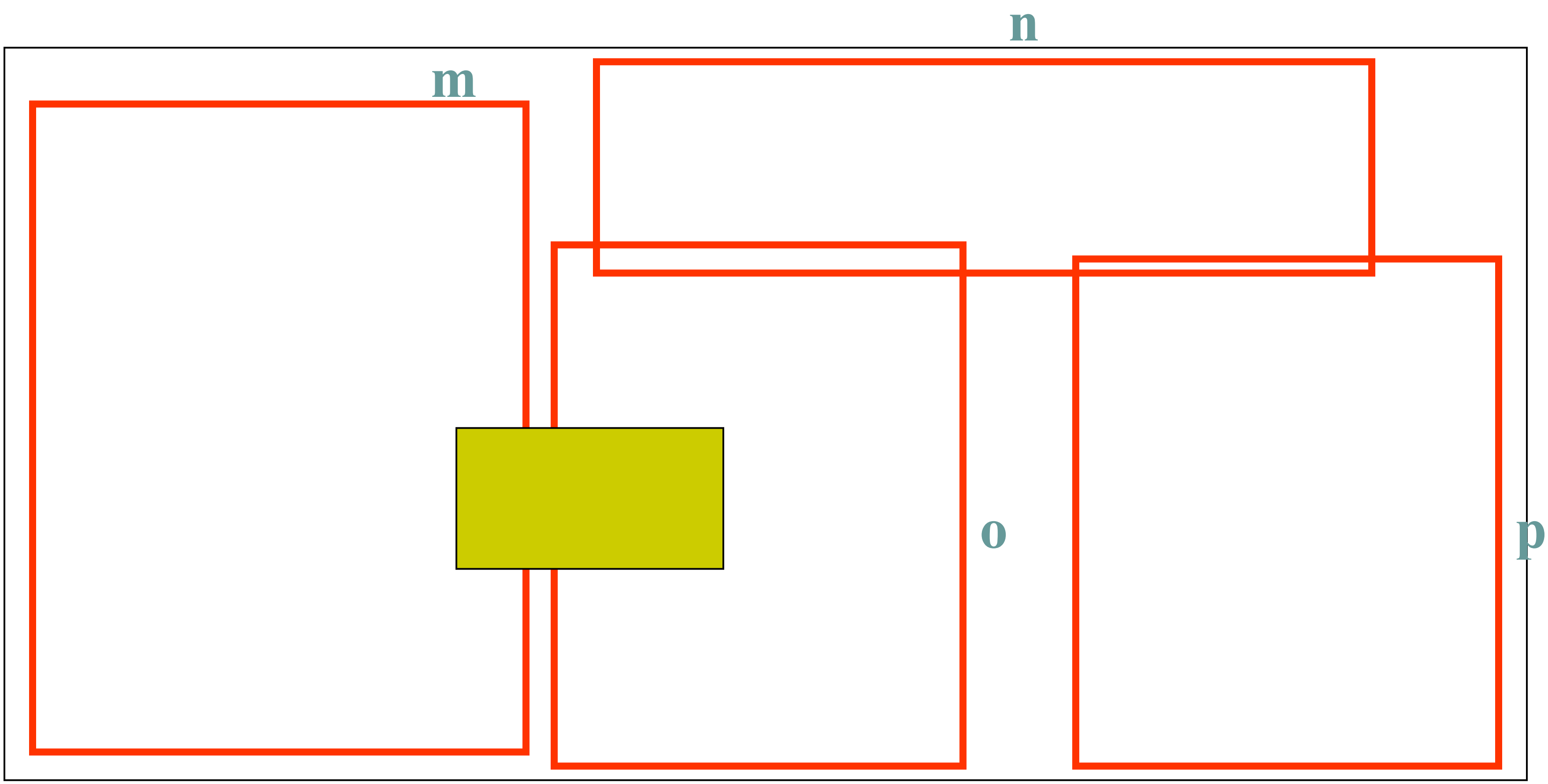
R-Trees: Operations

- Insertion
- Deletion
- Update (delete and re-insert)
- Queries/Searches
  - Give me all rectangles that are contained in the input rectangle.
  - Give me all rectangles intersecting this rectangle.
  - Give me k nearest neighbors to a given location

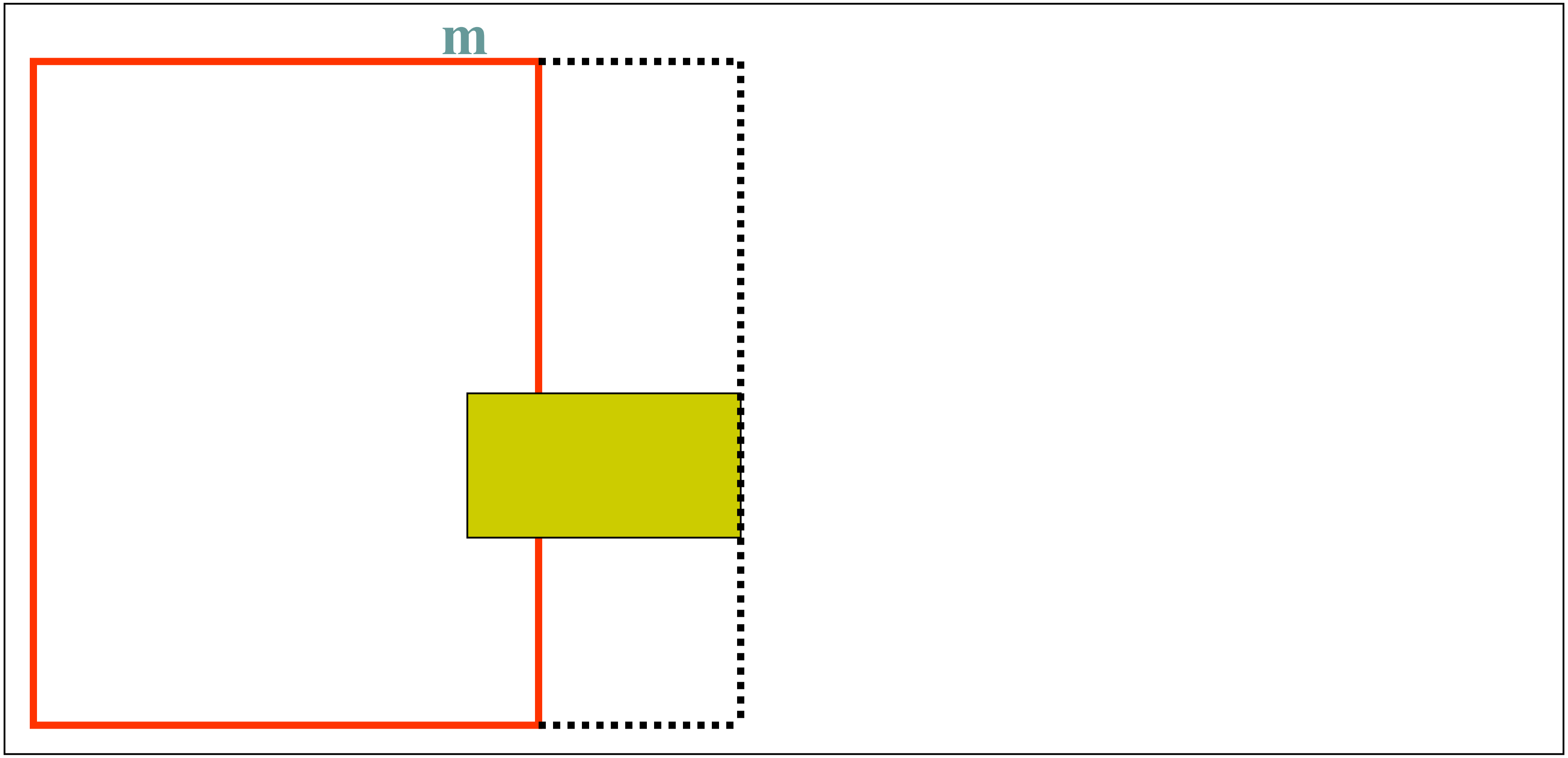
Insertion

- Similar to insertion into B+-tree but may insert into any leaf; leaf splits in case capacity exceeded.
  - Which leaf to insert into? (Choose Leaf)
    - Find the mbr whose rectangle needs least enlargement to include the record
  - How to split a node? (Node Split)
    - Similar to B-trees, new index records are added to the leaves
    - Nodes that overflow are split
    - Splits propagate up the tree

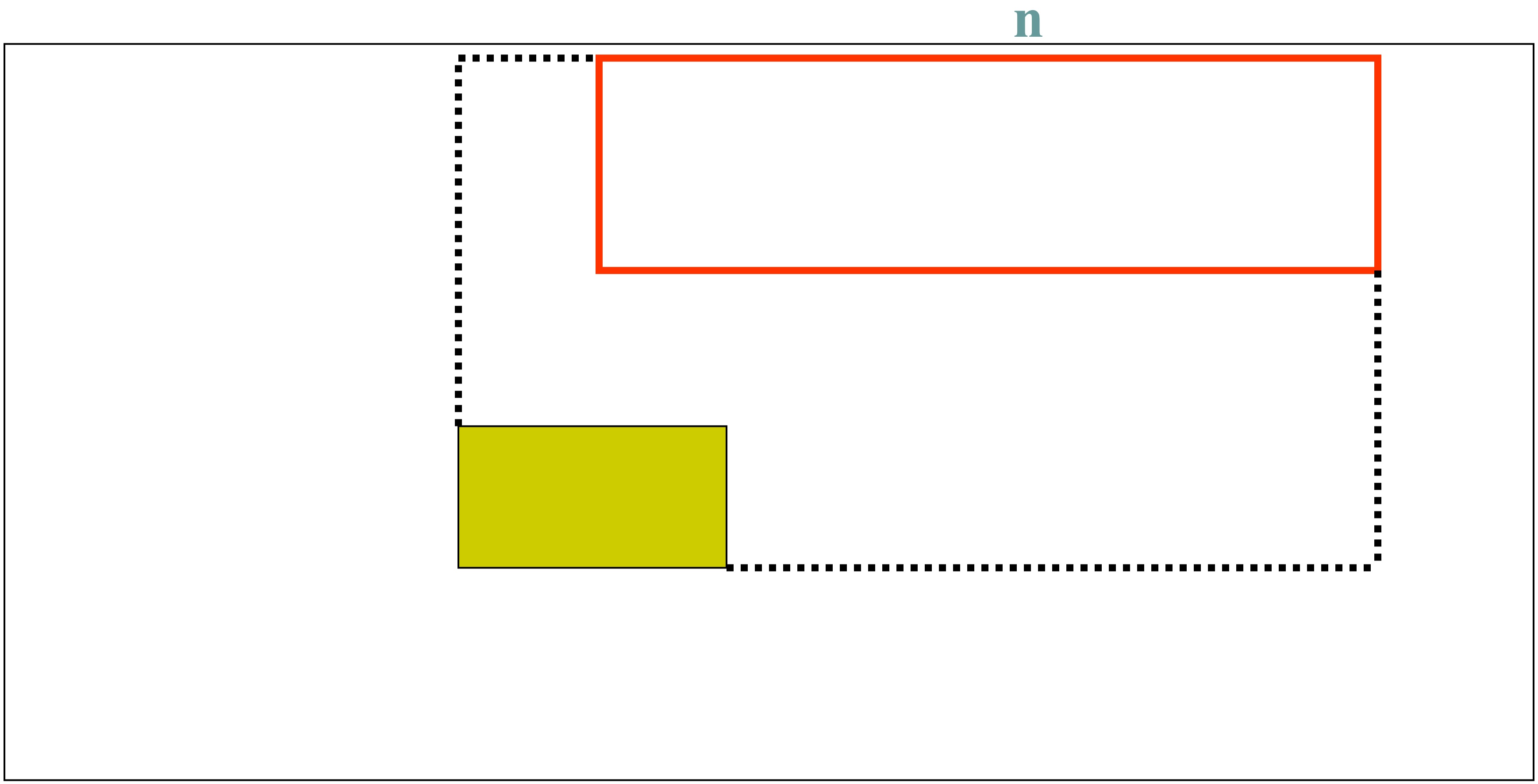
Insertion: Choose Leaf



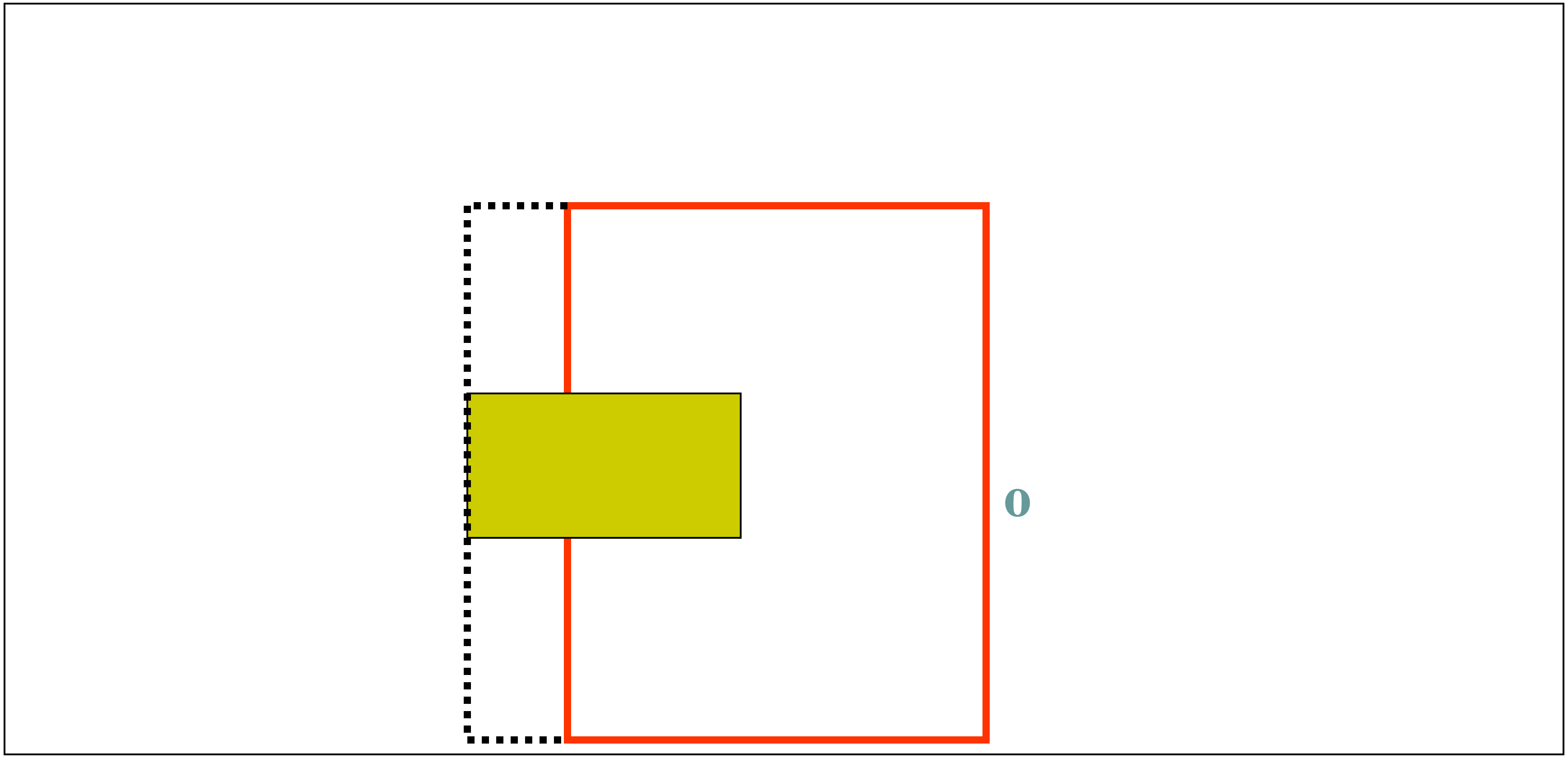
Insertion: Choose Leaf



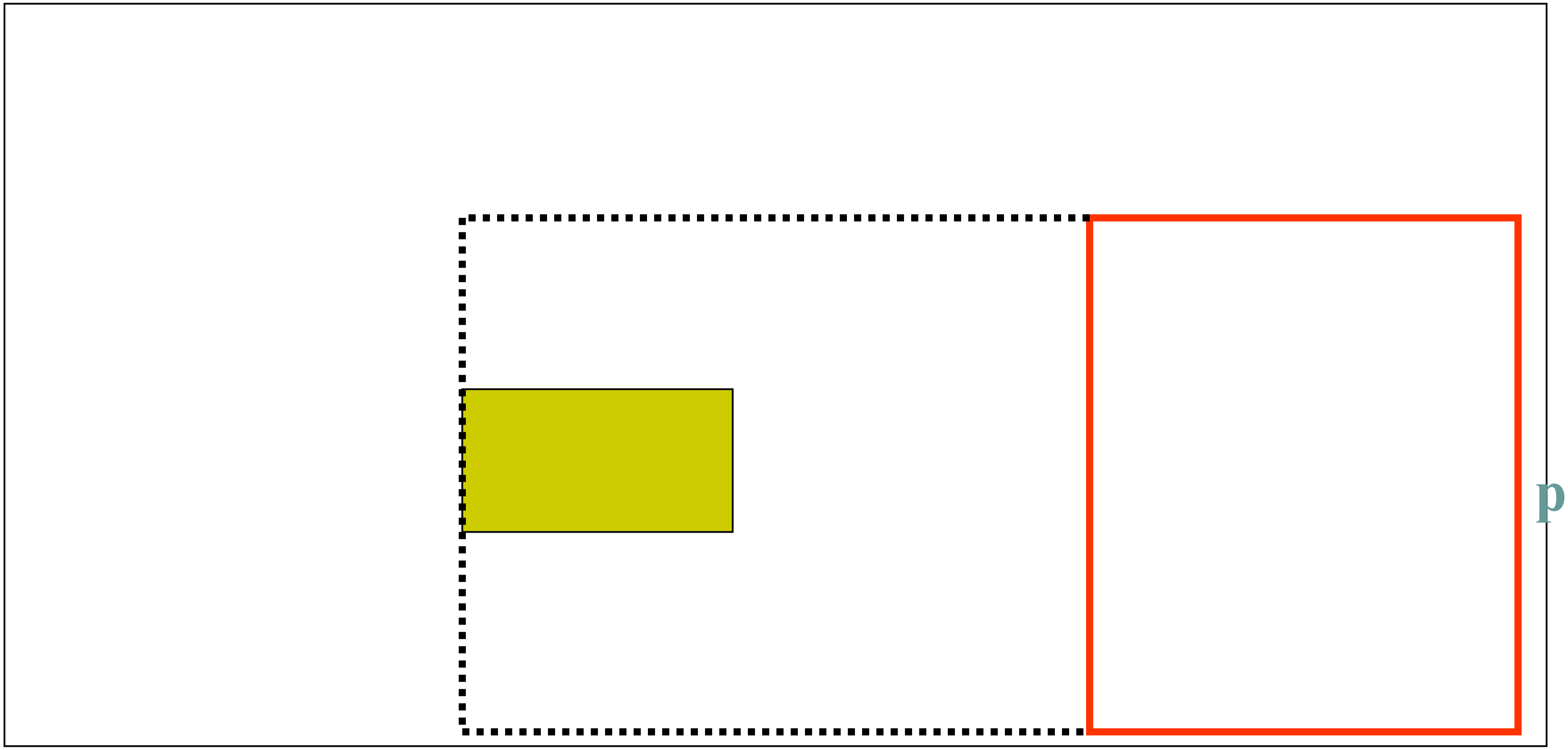
Insertion: Choose Leaf



Insertion: Choose Leaf

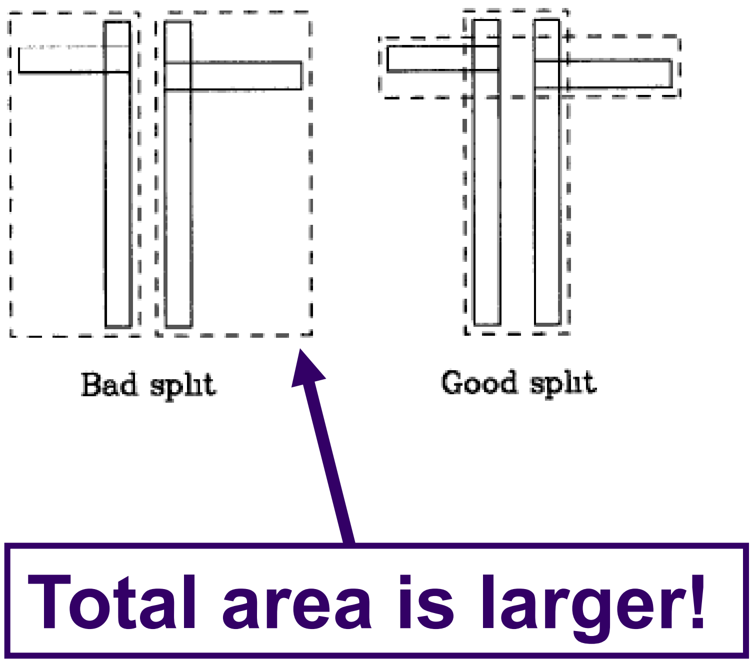


Insertion: Choose leaf



Node Splitting

- A full node contains M entries.
- Divide the collection of M+1 entries to 2 nodes.
- Objective: Make it as unlikely as possible for the resulting two new nodes to be examined on subsequent searches.
- Heuristic: The total area of two covering rectangles after a split should be minimized.



Node Splitting

- Divide M+1 objects to 2 groups
- Exhaustive algorithm
  - Generate all possible cases and choose the best case with minimum total area.
- Quadratic method
  - A heuristic to find a small-area split.
  - Cost is quadratic in M and linear in the number of dimensions.
- Linear method
  - A heuristic to find a small-area split.
  - Cost is linear in M and linear in the number of dimensions.

Quadratic method

- Initialize: Pick two of the M+1 entries to be the first elements of the two new groups, such that
  - O_i in Group 1
  - O_j in Group 2
  - $O_{i,j} – O_i – O_j$ is maximized
- Pick Next
  - $d_1 = O_{l,1} – O_l – G_1$
  - $d_2 = O_{l,2} – O_l – G_2$
  - Find O_l to maximize $|d_1 - d_2|$
  - Add O_l to Group 1 if $d_1 < d_2$; otherwise Group 2

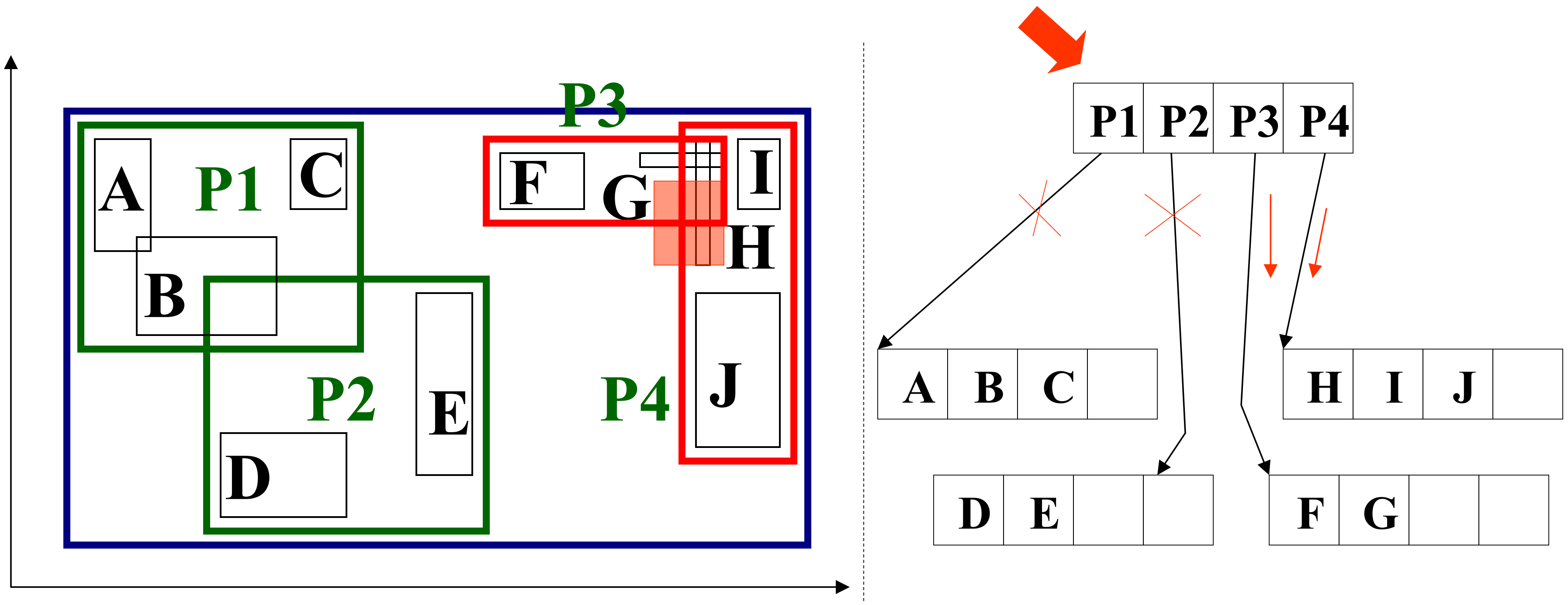
Linear method

- Initialize: Choose two objects that are furthest apart.
- Pick Next
  - simply chooses any of the remaining entries and insert it into the group to minimize the total area

Delete

- Straightforward.
- The only complication is under-flows:
- An under-full node can be merged with whichever sibling will have its area increased least.

R-tree: range query



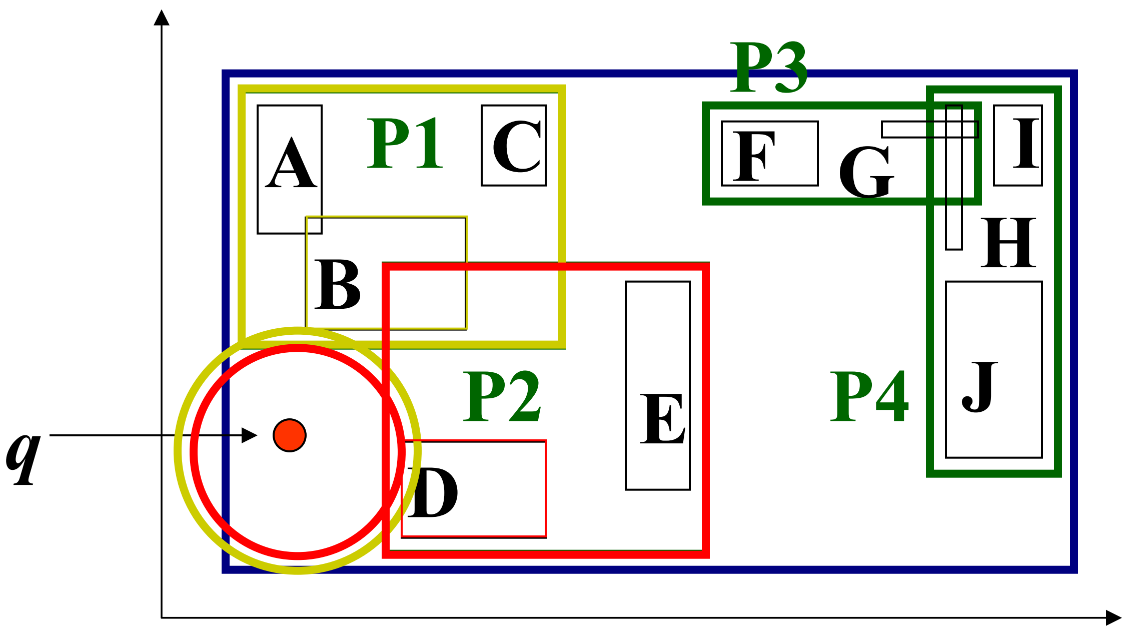
R-tree: range query – pseudo code

```
Range-search(RtreeNode R; query rectangle Q) {
   For each entry of R:
        If its MBR intersects the query rectangle Q
              Apply range-search on the child node of
              the entry;
              or print out, if this is a leaf entry;
}
```

MBR: $\{r_{x1}, r_{x2}, r_{y1}, r_{y2}\}$

R-tree: NN query

- Depth-first search
  - Find a near object
  - (circular) range query
  - Refine the query range (circle)

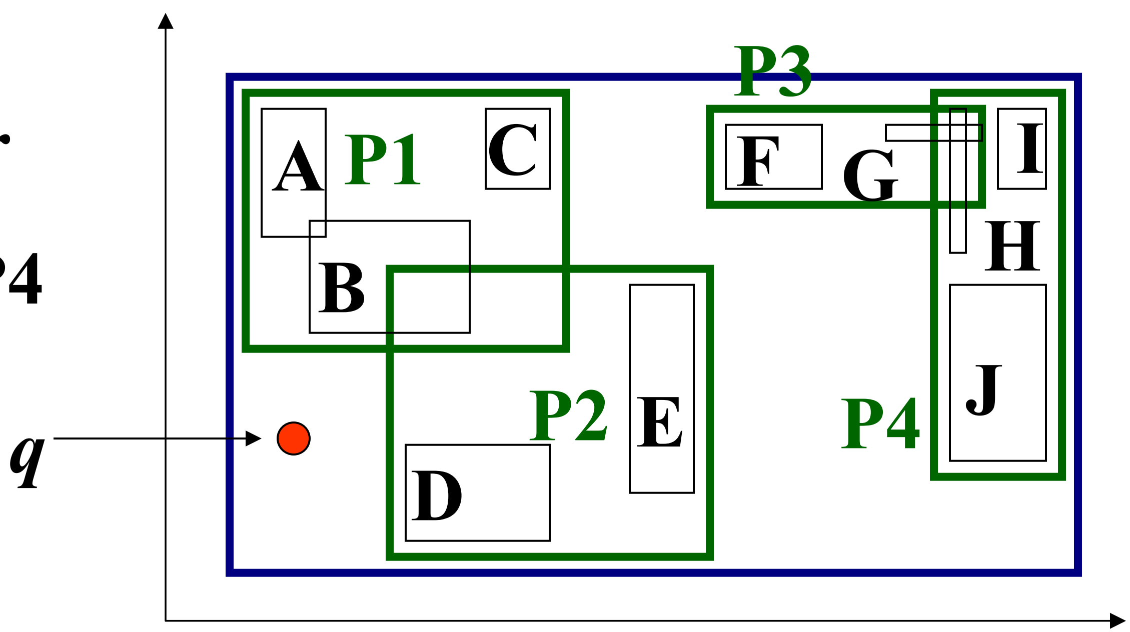


R-tree: NN query

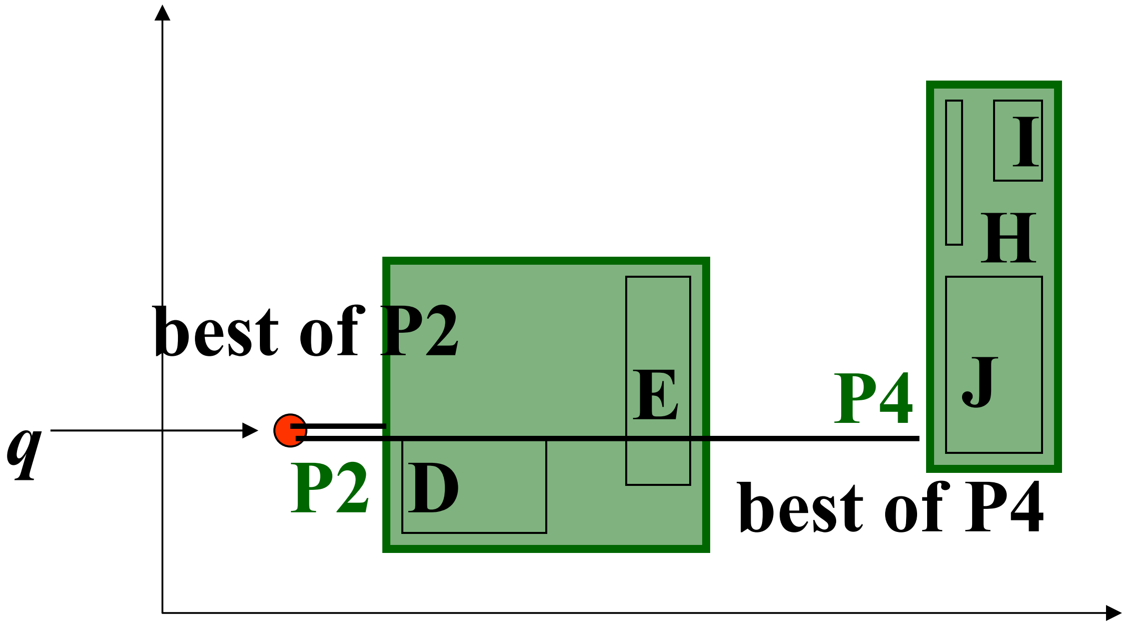
- Best-first search
  - priority queue, with promising MBRs, and their best and worst-case distance
- Main idea: Every face of any MBR contains at least one point of an actual spatial object

Consider

P2 and P4



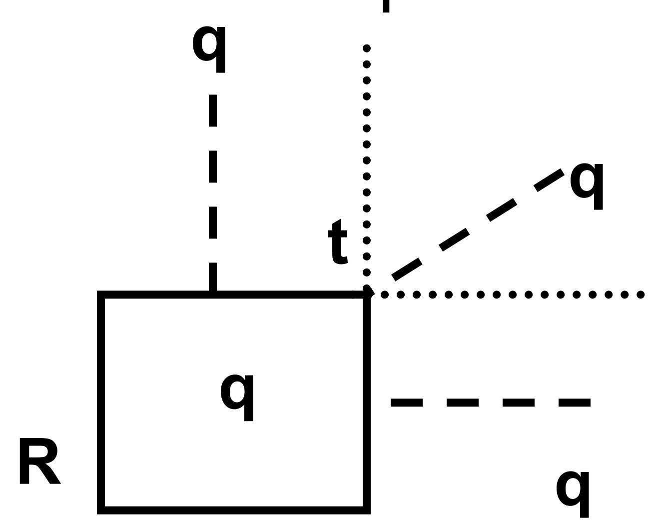
R-tree: NN query



P2 is more promising for 1-nn

MINDIST Function

- **MINDIST (q, R)** is the minimum distance between a point q and a rectangle R.
  - If the point is inside the rectangle, MINDIST = 0;
  - If the point is outside the rectangle, MINDIST is the minimal possible distance from the point to any object in or on the perimeter of the rectangle.



Search

- Dist(q, o): distance between query q and object o
- MinDist(q, R) function: minimal distance from q to R
  - q: location
  - R: MBR
- NN Search
  - If Dist(q, o_{best})<MinDist(q, R) , prune R
- KNN Search
  - k best objects

NN Search Algorithm

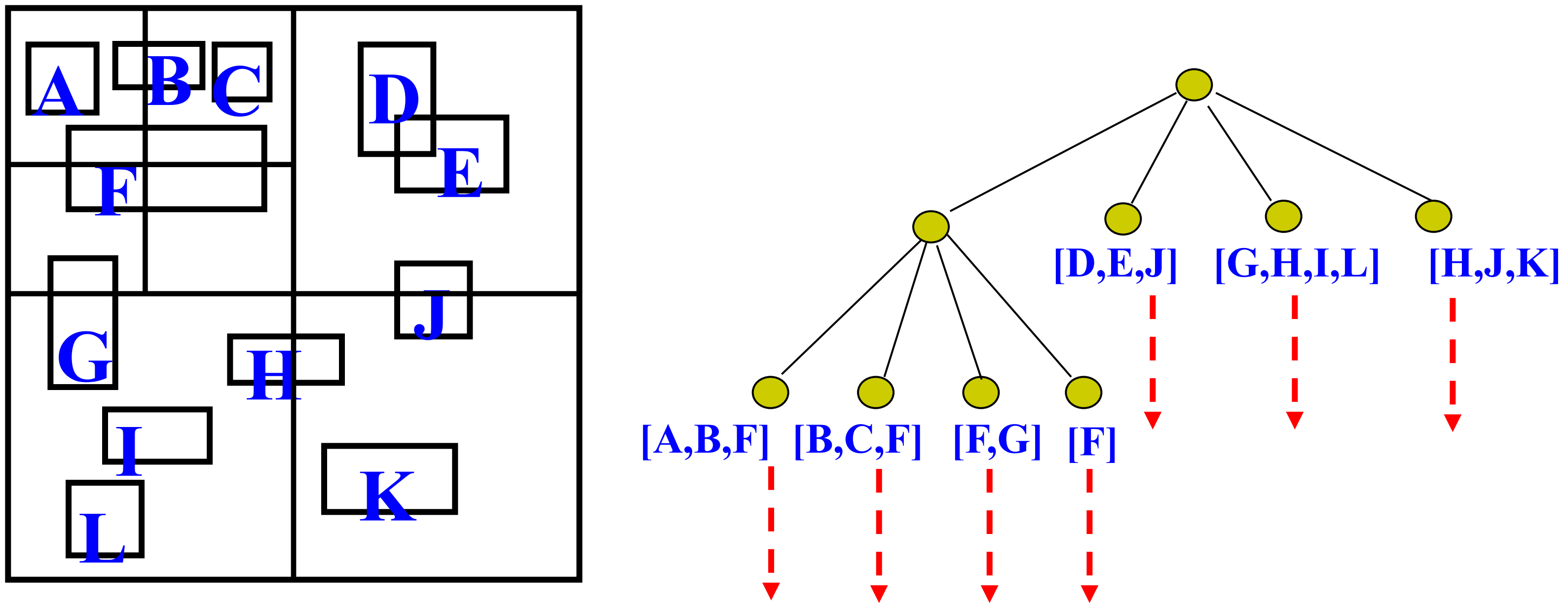
- Maintain a priority queue Q
- Insert the root into Q
- While Q is not empty
  - Dequeue the element **e** with minimal MinDist(q, **e**)
  - If **e** is an object
    - return **e**
  - Else
    - For each child C of **e**, Add C into Q

KNN Search Algorithm

- Maintain a priority queue Q and a result set A
- Insert <root, 0> into Q
- While Q is not empty and |A|<k
  - Dequeue the element **e** with minimal MinDist(q, **e**)
  - If **e** is an object
    - add **e** into A
  - Else if **e** is a non-leaf MBR,
    - For each child C of the MBR, Add <C, MinDist(q, C)> into Q
  - Else if e is a leaf MBR
    - For each object O in the MBR, Add <O, Dist(q, O)> into Q

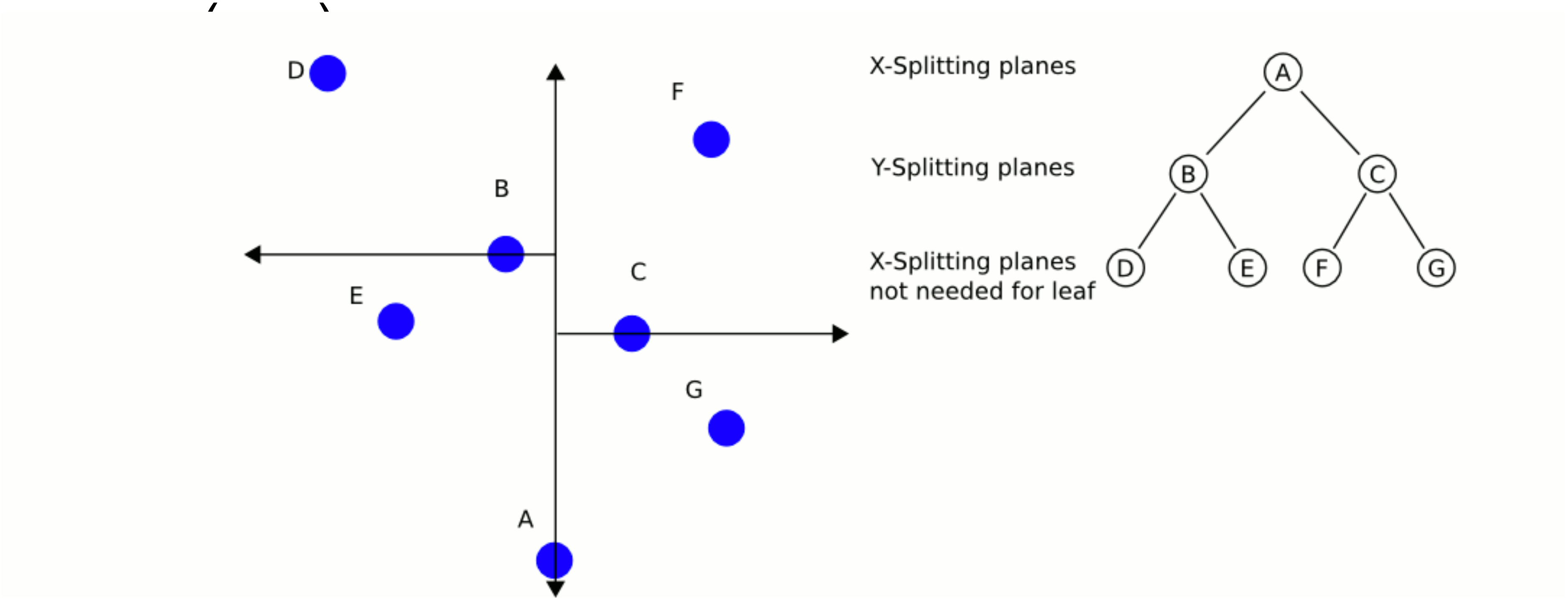
Quadtree

- Space-based hierarchical structures
- A recursive subdivision of the space into 4 quadrants
- The subdivision can be into either equal sized or unequal sized quadtrants
- Each leaf node has at most m objects



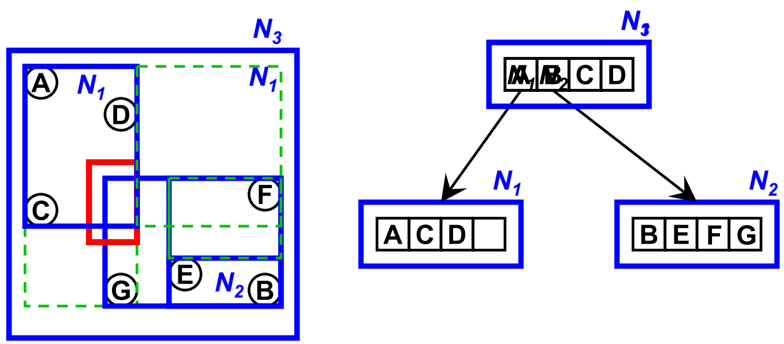
KD-tree

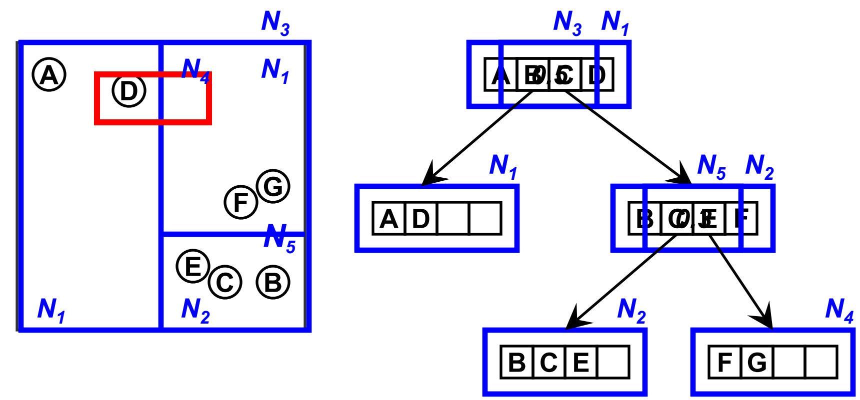
- **A recursive space partitioning tree**.
  - Partition along x and y axis in an alternating fashion.
  - Each internal node stores the splitting node along x

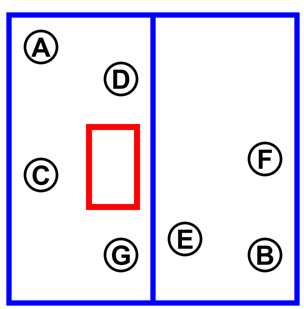


Hierarchical Tree Structures

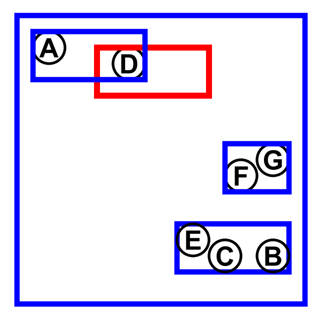
- R-tree
  - Minimum bounding rectangle (MBR)
  - Incomplete and overlapping partitioning
  - Disk-based; Balanced
- K-d-tree
  - Space division recursively
  - Complete and disjoint partitioning
  - In-memory; Unbalanced
  - There are algorithms to page and balance the tree, but with more complex manipulations







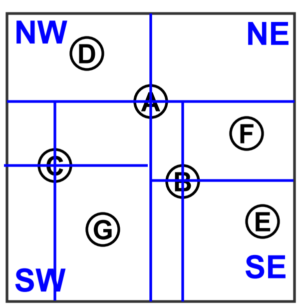
Problem: Overlap

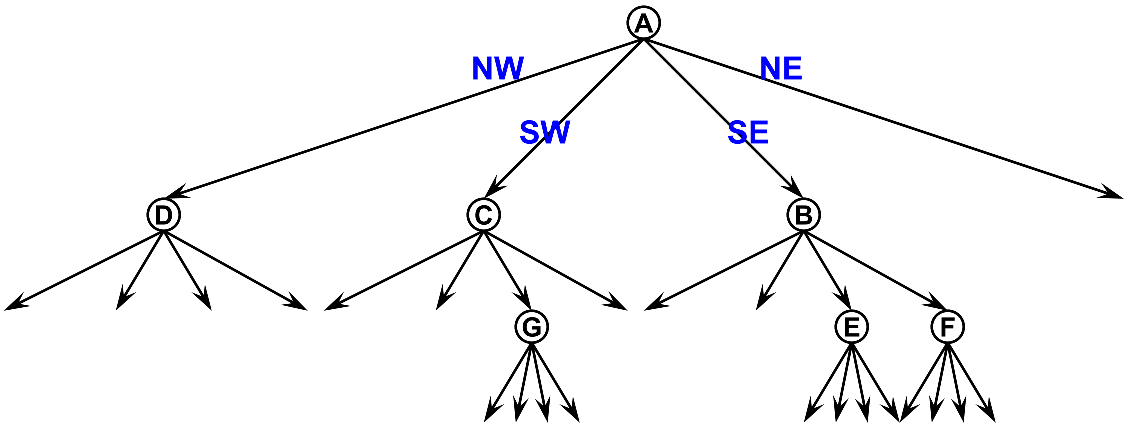


Problem: Empty space

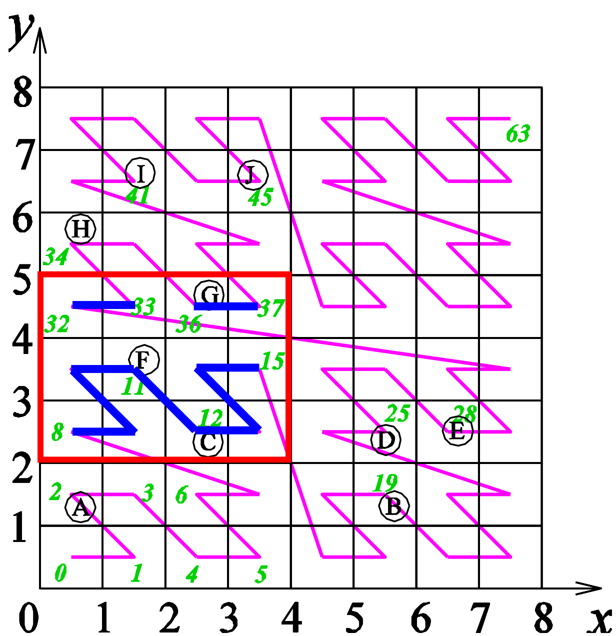
Hierarchical Tree Structures (continued)

- Quad-tree
  - Space divided into 4 rectangles recursively.
  - Complete and disjoint partitioning
  - In-memory; Unbalanced
  - There are algorithms to page and balance the tree, but with more complex manipulations
- The ***point quad-tree***





Mapping Based Multidimensional Indexing



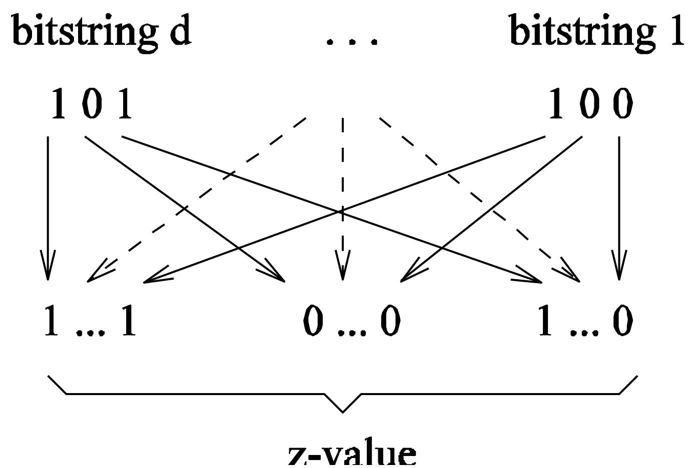
Sort

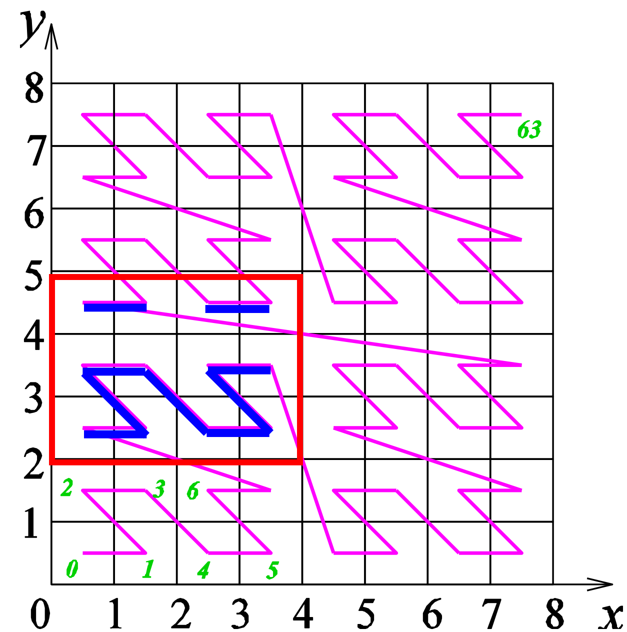
Name	x	y	Block	Height
A	0.7	1.2	2	100
B / F	5.7 / 1.8	1.2 / 3.8	19 / 11	120 / 50
C	2.7	2.3	12	80
B / D	5.8 / 5.6	1.2 / 2.4	19 / 25	50 / 90
D / E	6.6 / 5.6	2.4 / 2.5	25 / 28	90 / 40
E	6.7 / 1.6	2.5 / 3.8	28 / 11	40 / 120
G / H	2.8 / 0.6	4.8 / 5.7	36 / 34	50 / 100
G / H	2.8 / 0.6	4.8 / 5.7	36 / 34	50 / 100
I	1.6	6.7	41	60
J	3.4	6.6	45	40

- Story
  - The CBD: [0,2][4,5]
  - Blocks in the CBD are: [8,15], [32,33] and [36,37]
- General strategy: three steps
  - Data mapping and indexing
  - Query mapping and data retrieval
  - Filtering out false positive

The Z-curve and Other Space-Filling Curves

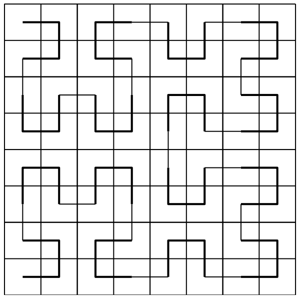
- The Z-curve
  - Z-value calculation: bit-interleaving

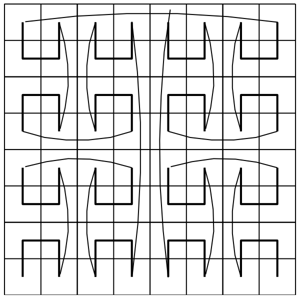


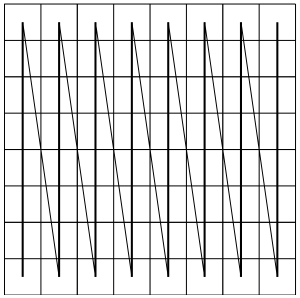


  - Support efficient window queries
  - Disadvantage
    - Jumps

- Other space-filling curves
  - Hilbert-curves
  - Gray-code
  - Column-wise scan







Summary of the Indexing Techniques

Index	Disk-based / In-memory	Balanced	Efficient query type	Dimensi onality	Comments
R-tree	Disk-based	Yes	Point, window, kNN	Low	Disadvantage is overlap
K-d-tree	In-memory	No	Point, window, kNN(?)	Low	Inefficient for skewed data
Quad-tree	In-memory	No	Point, window, kNN(?)	Low	Inefficient for skewed data
Z-curve + B^+-tree	Disk-based	Yes	Point, window	Low	Order of the Z-curve affects performance

Index Implementations in major DBMS

- SQL Server
  - B+-Tree data structure
  - Clustered indexes are sparse
  - Indexes maintained as updates/insertions/deletes are performed
- Oracle
  - B+-tree, hash, bitmap, spatial extender for R-Tree
  - Clustered index
  - Index organized table (unique/clustered)
  - Clusters used when creating tables
- DB2
  - B+-Tree data structure, spatial extender for R-tree
  - Clustered indexes are dense
  - Explicit command for index reorganization