Spatial Data and Queries

- Spatial Data
- Spatial Relationships
- Spatial Queries
- Issues in Query Processing
- The R-tree
- Spatial Query Processing
  - Spatial Selections
  - Nearest Neighbor Queries
  - Spatial Joins

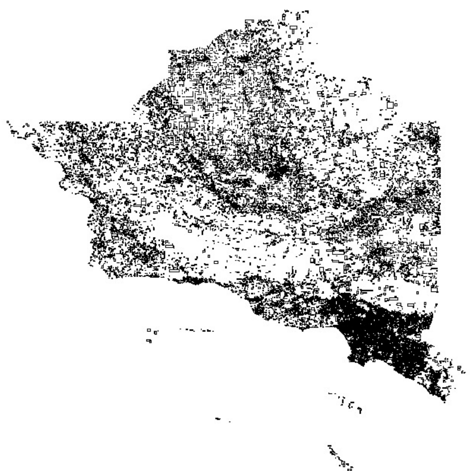
Spatial Data and Queries

Spatial Data

- Spatial objects are a special case of multidimensional objects
  - Just two dimensions
  - Both dimensions are interval-scaled
  - Natural mapping of dimensions to real maps
- Spatial objects are characterized by
  - location
  - geometry (only for non-points)
- Multidimensional objects are points

Spatial Data Management

- Spatial Database Systems manage large collections of 2D/3D objects
- A spatial object (at least) one spatial attribute that describes its location and/or geometry
- A spatial dataset is an organized collection of spatial objects of the same entity (e.g. rivers, cities, road segments)



Road segments from an area in CA

ID	Name	Type	Polyline
1	Boulevard	avenue	(10023,1094), (9034,1567), (9020,1610)
2	Leeds	highway	(4240,5910), (4129,6012), (3813,6129), (3602,6129)
…	…	…	…

A spatial dataset (relation)

Spatial Data

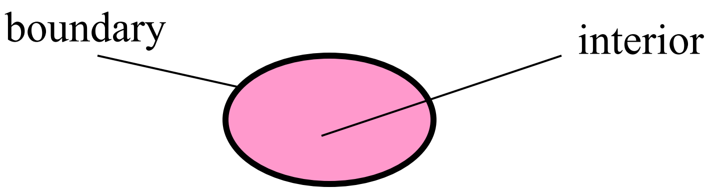
- Spatial data can be found in many applications
  - Geographic Information Systems
  - Segmented images (e.g., objects in X-rays)
  - Components of CAD constructs or VLSI circuits
  - Stars on the sky
  - …
- Spatial database systems are used by
  - Users of mobile devices (find the nearest restaurant)
  - Geographers, astrologers, life scientists, army commanders, etc.

Spatial Relationships

- A spatial relationship or spatial predicate associates two objects according to their relative location and extent in space
  - Example: “My house is close to Central Park”
- Sometimes also called “spatial relation”.
- Spatial relationships are classified to
  - topological relationships (for objects with geometries)
  - distance relationships (mostly for point objects)
  - directional relationships (mostly for point objects)

Topological Relationships

- Each object is characterized by the space it occupies in the universe.
  - a (finite or infinite) set of elementary points (pixels)
- Each object has a boundary and an interior
  - boundary: the set of pixels the object occupies, that are adjacent to at least one pixel not occupied by the object
  - interior: the set of pixels occupied by the object, which are not part of its boundary



- Note: in some representation models, some objects may not have interior
  - e.g., points, line segments, etc.

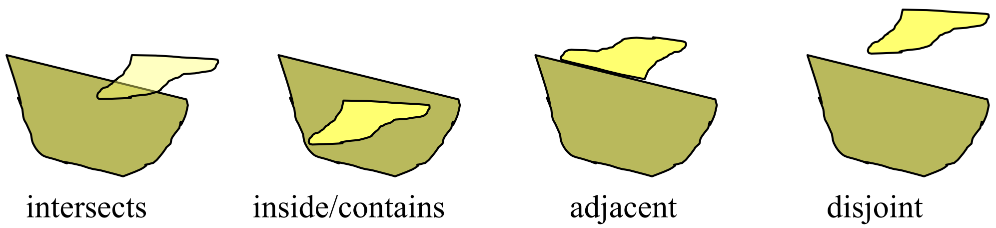
Topological Relationships

- A topological relationship between two objects is defined by a set of (set-based) relationships between their boundaries and interiors
  - E.g., o_1 is inside o_2 if interior(o_1) $\subset$ interior(o_2)
- intersects (or overlap) means any of equals, inside, contains, adjacent
- intersects $\Leftrightarrow$ $\neg$disjoint

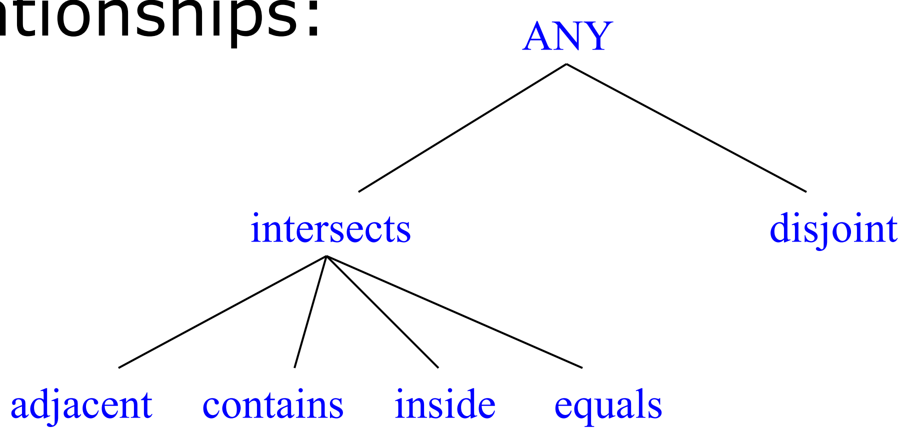
Topological relationship	equivalent boundary/interior relationships
$disjoint(o_1, o_2)$	$(interior(o_1) \cap interior(o_2) = \emptyset) \wedge (boundary(o_1) \cap boundary(o_2) = \emptyset)$
$intersects(o_1, o_2)$ (or $overlaps(o_1, o_2)$)	$(interior(o_1) \cap interior(o_2) \neq \emptyset) \vee (boundary(o_1) \cap boundary(o_2) \neq \emptyset)$
$equals(o_1, o_2)$	$(interior(o_1) = interior(o_2)) \wedge (boundary(o_1) = boundary(o_2))$
$inside(o_1, o_2)$	$interior(o_1) \subset interior(o_2)$
$contains(o_1, o_2)$	$interior(o_2) \subset interior(o_1)$
$adjacent(o_1, o_2)$	$(interior(o_1) \cap interior(o_2) = \emptyset) \wedge (boundary(o_1) \cap boundary(o_2) \neq \emptyset)$

Topological Relationships

- Examples of topological relationships

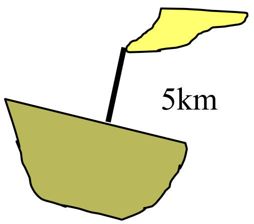


- In fact, there is a hierarchy of topological relationships:



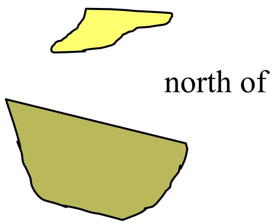
Distance Relationships

- Distance relationships associate two objects based on their geometric distance (typically, the minimum Euclidean distance)
- Distance is usually abstracted (i.e., discretized) into the human mind.
  - Example (distances in a city)
    - 0-100m: near
    - 100m-1km: reachable
    - 1km-10km: far
    - >10km: very far
- Distances are typically measured and then mapped to some abstract distance class (e.g., near, far)



Directional Relationships

- Directional relationships associate two objects based on their relative orientation according to a global reference system
- Example: My house is north of the river
  - relationship can also be a number:
    - e.g. house 96 degrees relative to river
- Examples of directional relationships:
  - north, south, east, west, northeast, etc.
  - left, right, above, below, front, behind, etc.
- Topological, distance, and directional relationships can be combined :
  - My house is disjoint with the park, 100 meters north of it



Spatial Queries

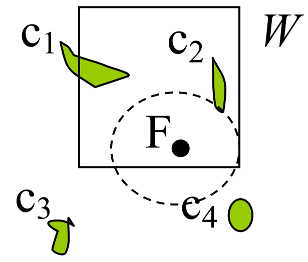
- Applied on one (or more) spatial datasets
- Retrieve objects (or combinations of objects) satisfying some spatial relationships
  - between them or
  - with a reference query object
- Examples:
  - Nearest neighbor query: What is the nearest city to my current location?
  - Spatial join: Find all pairs of hotels and restaurants within 100m distance from each other

Spatial Queries

- Range query (spatial selection, window query)

  e.g. find all cities that *intersect* window *W*

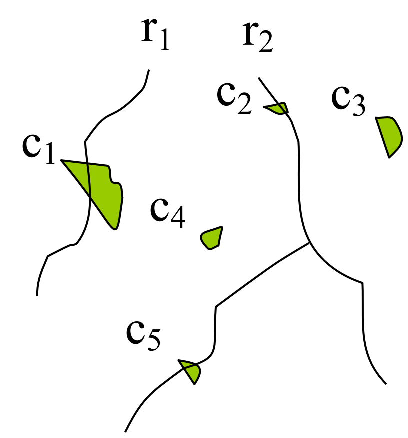
  Answer set: $\{c_1, c_2\}$



- Nearest neighbor query

  e.g. find the city closest to the forest F

  Answer: c_2



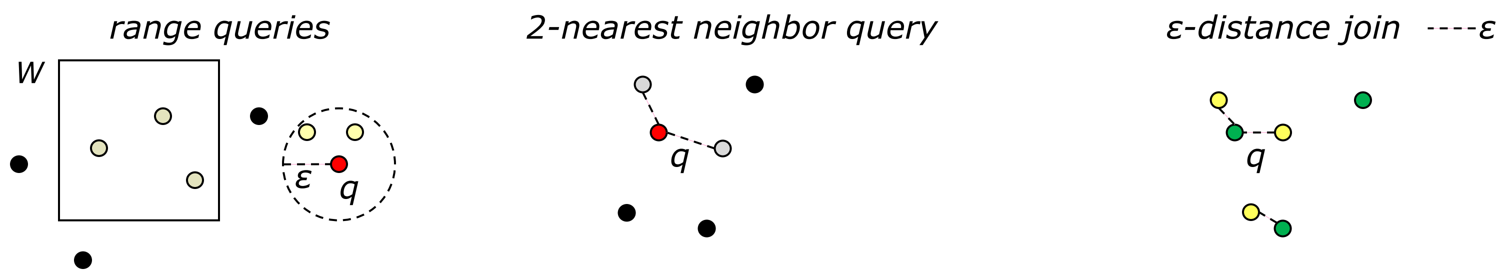
- Spatial join

  e.g. find all pairs of cities and rivers that intersect

  Answer set: $\{(r_1,c_1), (r_2,c_2), (r_2,c_5)\}$

Spatial Queries for points

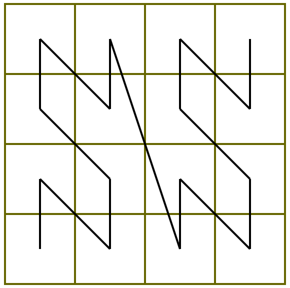
- For point data, the typical queries are
  - Range queries where the range can be:
    - A rectangular window
    - A circular range around a reference point
  - Nearest neighbor queries
    - Find k-nearest points
  - Spatial distance joins
    - Find pairs that are near each other
    - Find the closest pairs



Spatial Data Management

Issues in Spatial Data Management

- Data dimensionality
  - There is no total ordering of objects in the multidimensional space that preserves proximity
  - Example: space-filling curves
    - Two nearby points could be far in curve order
    - Neighboring values in curve could be far in space

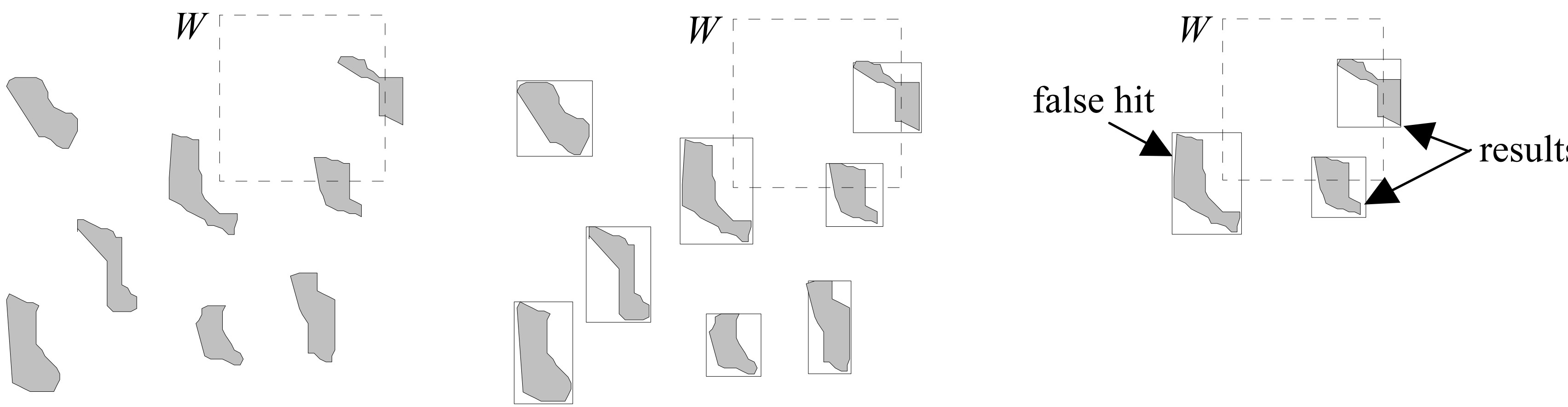


z-curve

- Geometries of non-point objects
  - adds to the complexity of partitioning the objects for indexing purposes
- Hence, traditional indexing and search techniques for 1D data are not applicable

Two-step Spatial Query Processing

- Complex geometries of spatial objects are approximated by their minimum bounding rectangles (MBR)
- A spatial query is then processed in two steps:
  - Filter step: The MBR is tested against the query predicate
  - Refinement step: The exact geometry of objects that pass the filter step is tested for qualification

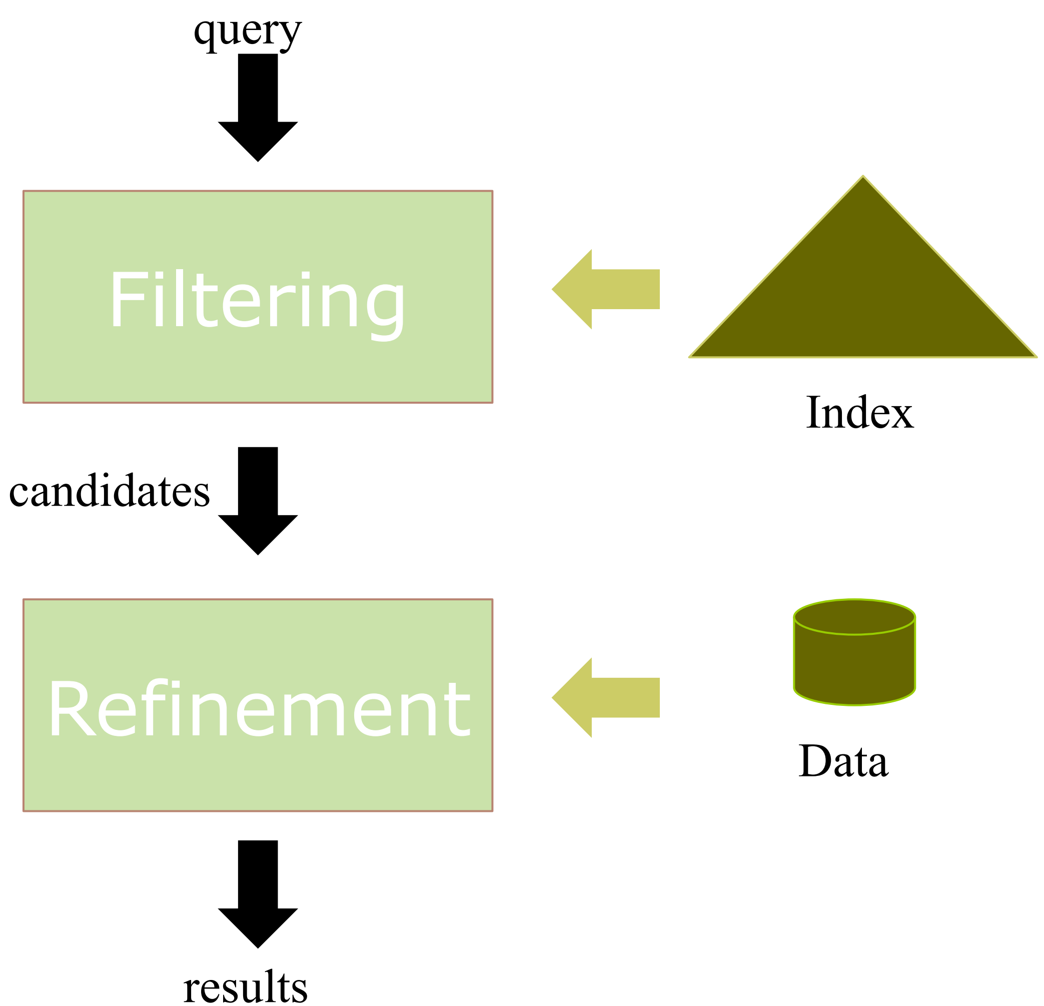


(a) objects and a query (b) object MBRs (c) candidates and results

The Filter-and-Refine Paradigm

- Reduce the search space
- Cheap filter techniques

- Remove false positives
- Verify results



Indexing techniques for points

- Spatial point objects in applications:
  - GPS locations
  - Points of Interest on maps
  - Scientific applications (e.g., astronomy)
- Indexes for points
  - space-filling curves
  - grid
  - k-d-tree
  - quadtree
- All of them can be generalized for multi-dimensional points

Z-order curve (Morton curve)

- Reading:
  - https://en.wikipedia.org/wiki/Z-order_curve
- Maps multi-dimensional points to 1D values
  - Space is discretized, coordinates are converted to integers
  - Applies bit-interleaving to the binary representations of coordinates
- A multi-dimensional range query is equivalent to finding the points in a set of 1D ranges on the Z-order curve
- See also: Hilbert space-filling curve

Z-order curve (Morton curve)

	x: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
y: 0 000	000000	000001	000100	000101	010000	010001	010100	010101
1 001	000010	000011	000110	000111	010010	010011	010110	010111
2 010	001000	001001	001100	001101	011000	011001	011100	011101
3 011	001010	001011	001110	001111	011010	011011	011110	011111
4 100	100000	100001	100100	100101	110000	110001	110100	110101
5 101	100010	100011	100110	100111	110010	110011	110110	110111
6 110	101000	101001	101100	101101	111000	111001	111100	111101
7 111	101010	101011	101110	101111	111010	111011	111110	111111

Example: (x,y) = (3,6)

x = 0b011   y = 0b110

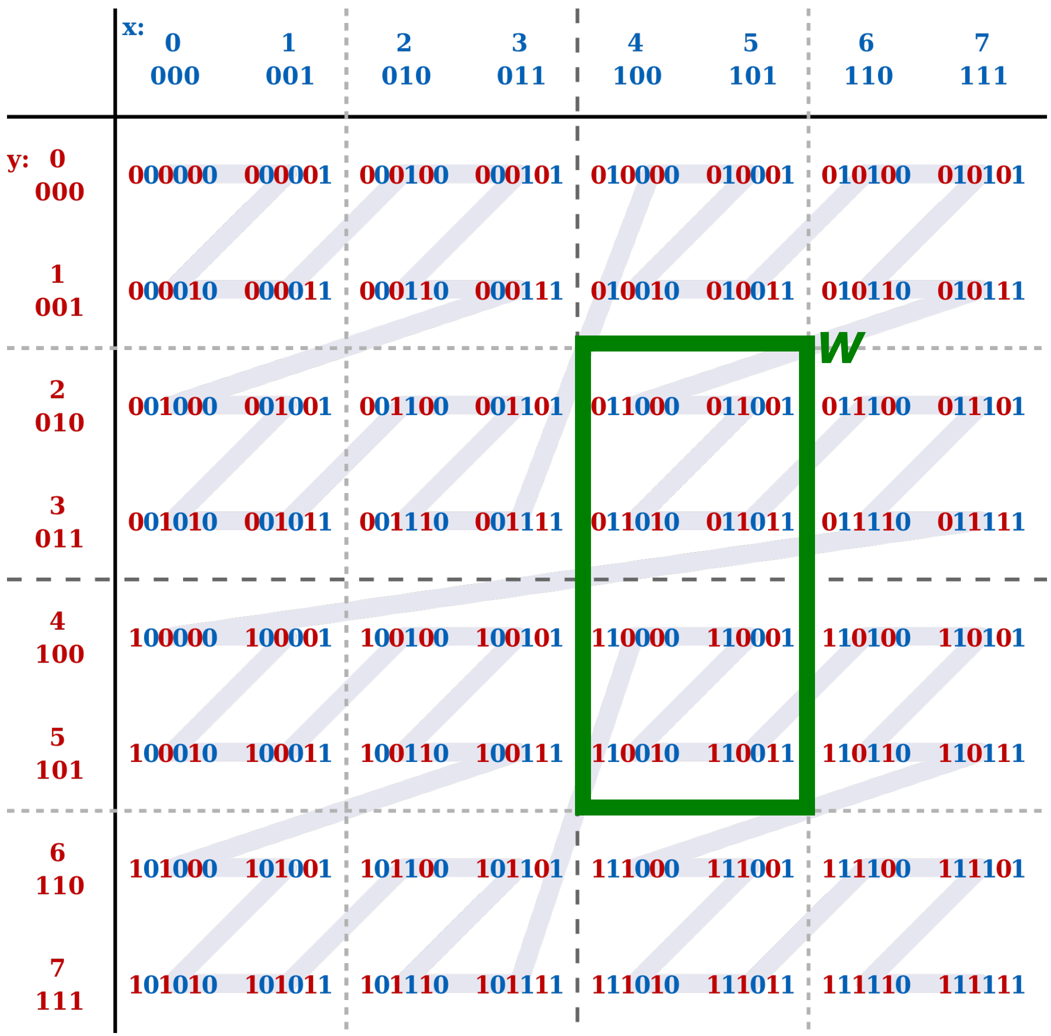
z = 0b101101 = 45

Example: z = 14

z = 0b001110

x = 0b010   y = 0b011

Z-order curve (Morton curve)



Range query

x in [4,5]
y in [2,5]

Can be converted to 1D ranges:

[0b011000, 0b011011]

[0b110000, 0b111011]

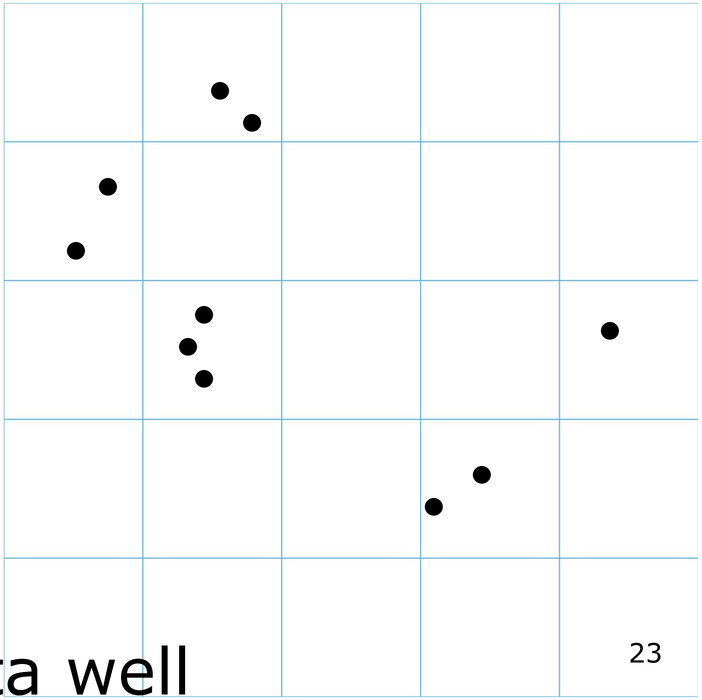
Evaluate query ranges if z-order values of points are indexed by a B$^+$-tree

Alternative approach using a BST:

Tropf, H.; Herzog, H. (1981), "Multidimensional Range Search in Dynamically Balanced Trees" Angewandte Informatik, 2: 71–77

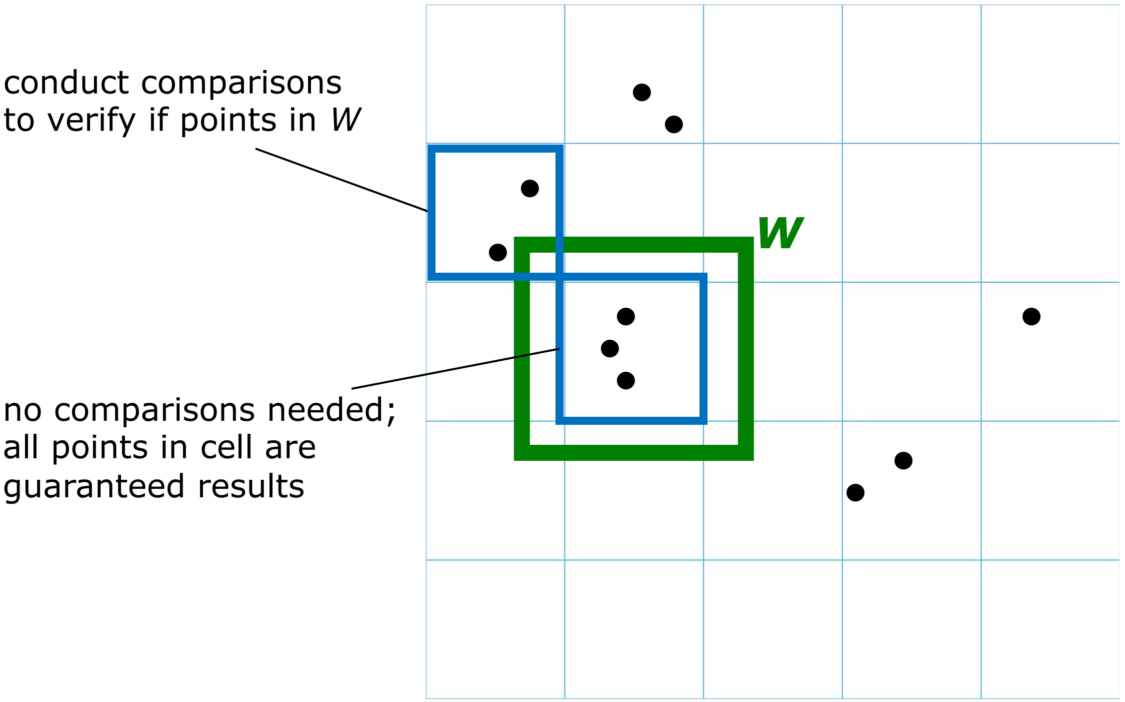
Grid Indexing

- Simple and suitable for highly dynamic data
- Space is divided by a uniform grid to disjoint cells
- Each grid cell defines a space partition
- Each point is assigned to one partition
  - Cell that contains the point is found in O(1) using algebraic operations
- Supports fast insert/delete and search in main memory
  - Not as good for disk
- Range queries, NN queries and joins also fast
- Does not support skewed data well



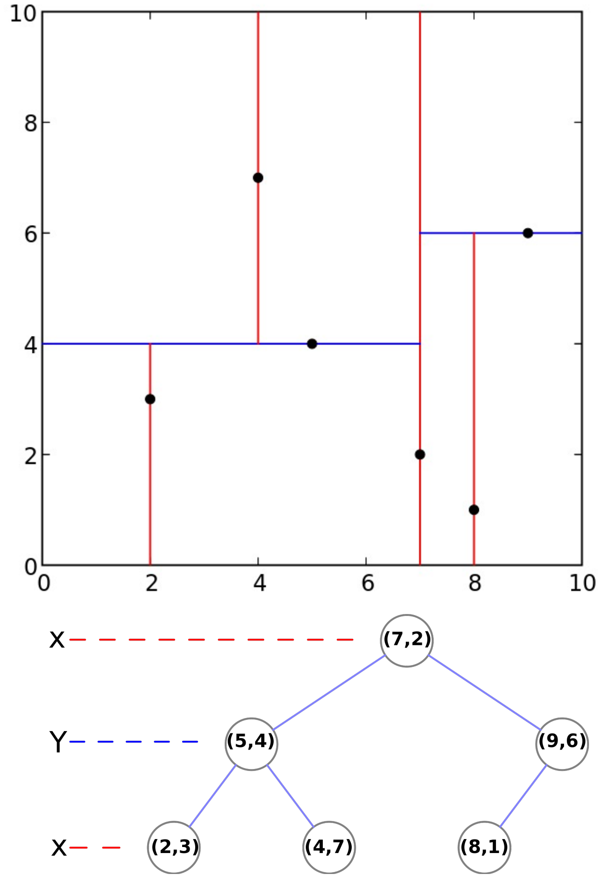
Grid Indexing – range queries

- Find cells that intersect *W* in O(1) using range extent at each dimension at each dimension
- Empty cells are disregarded
- Cells totally covered by *W* need no comparisons



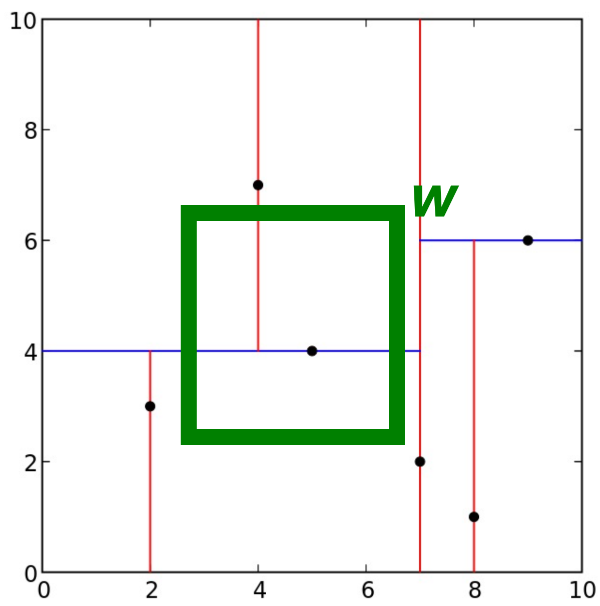
k-d tree

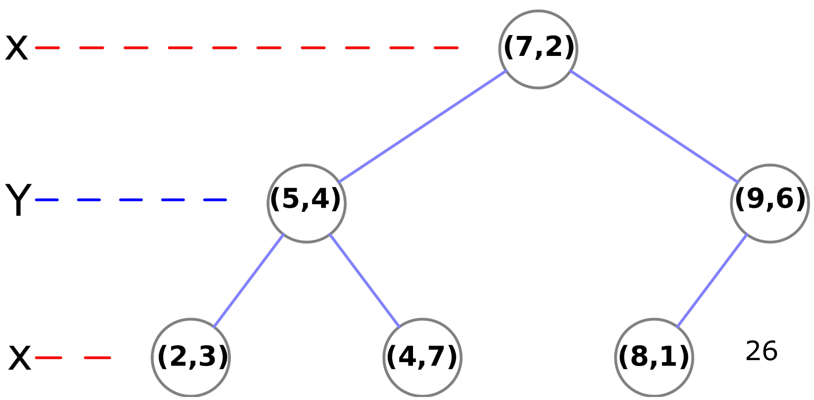
- Reading: https://en.wikipedia.org/wiki/K-d_tree
- A binary tree for points in a k-dimensional space
- Root uses one point (median) to divide the space into two subspaces using one dimension
  - sort a sample to find median
- Each node uses a point to divide its subspace in two parts
- Dimensions alternate per level



k-d tree – range queries

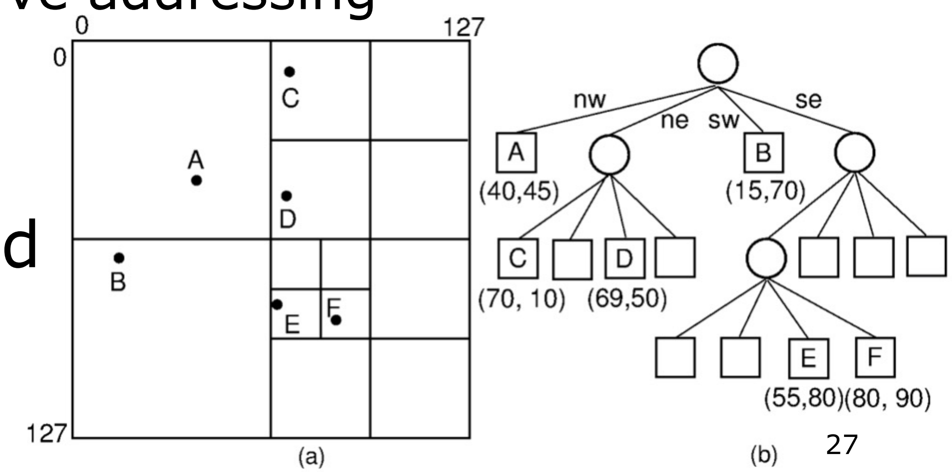
- Case 1: W entirely before node-point in splitting dimension
  - recursive call for left son
- Case 2: W entirely after node-point in splitting dimension
  - recursive call for right son
- Case 3: W covers node-point in splitting dimension
  - report node if in *W*; recursive call for left and right son





Point-region (PR) quadtree

- Reading: https://en.wikipedia.org/wiki/Quadtree
- Each non-leaf divides its area to four quadrants
- Each leaf has points (up to a max capacity)
  - Leaves that exceed the max capacity are split
- Various tree representations
  - Traditional: use pointers to link nodes
  - Implicit: use z-order curve addressing
- Quadtree may not be balanced
- Quadtrees are also used for approximating regions or images

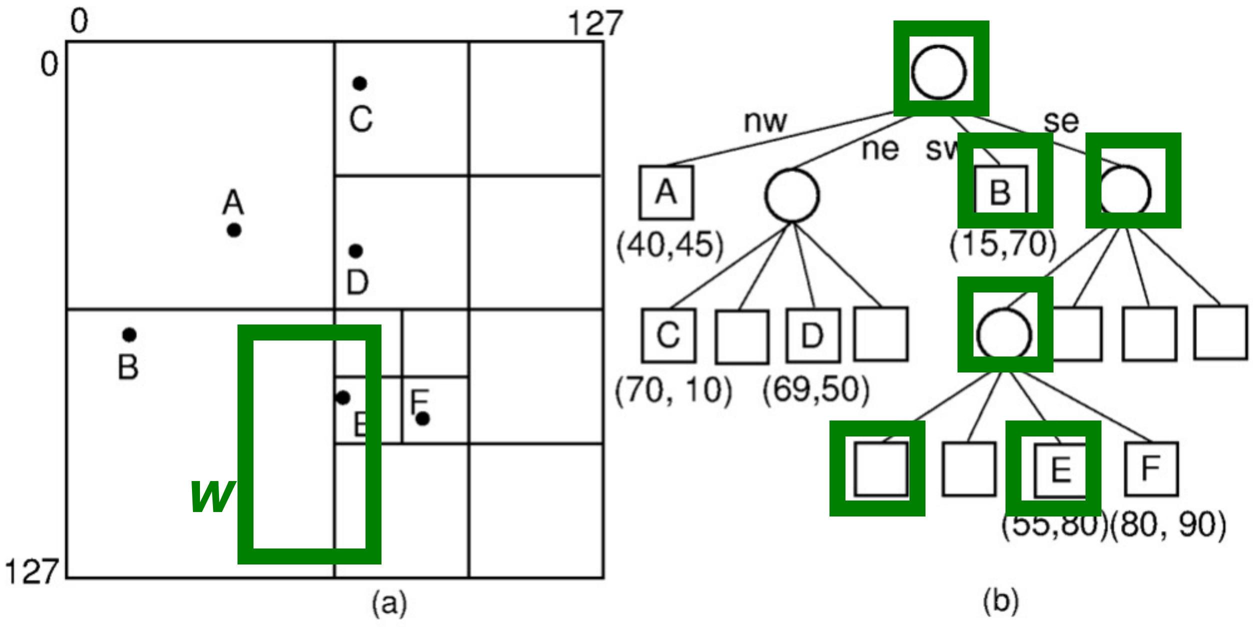


PR-quadtree, max capacity = 1

https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/PRquadtree.html

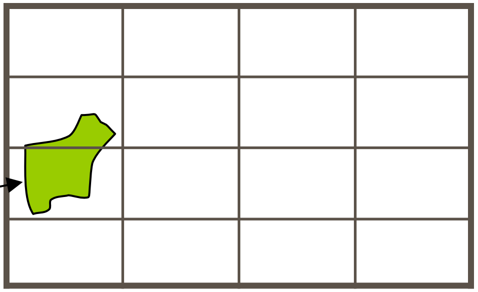
Point-region (PR) quadtree - queries

- Visit nodes that overlap with query range recursively
- Can use bit-interleaving and prefixes of query bounds to guide search at non-leaf nodes
  - That is, prefixes of z-order codes of query range



Spatial Access Methods

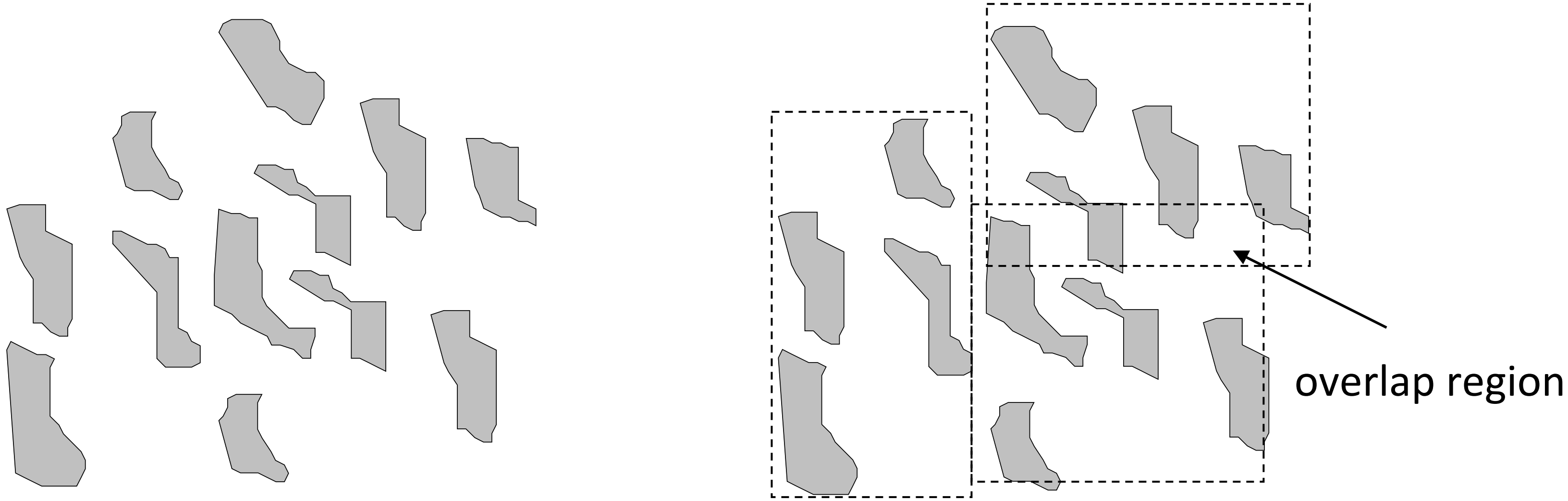
- Point access methods are not effective for extended objects
  - They divide the space into disjoint partitions
  - Objects may need to be clipped into several parts which leads to data redundancy and affects performance negatively



object clipping in a grid

Spatial Access Methods

- Object clipping can be avoided if we allow the regions of object groups to overlap

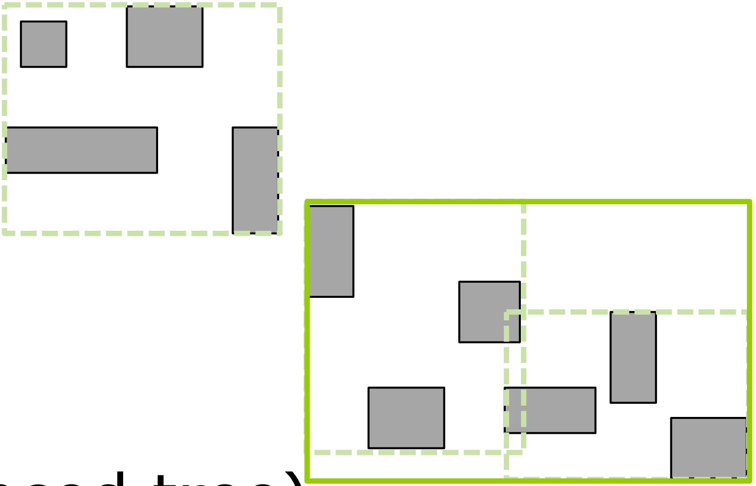


The R-tree

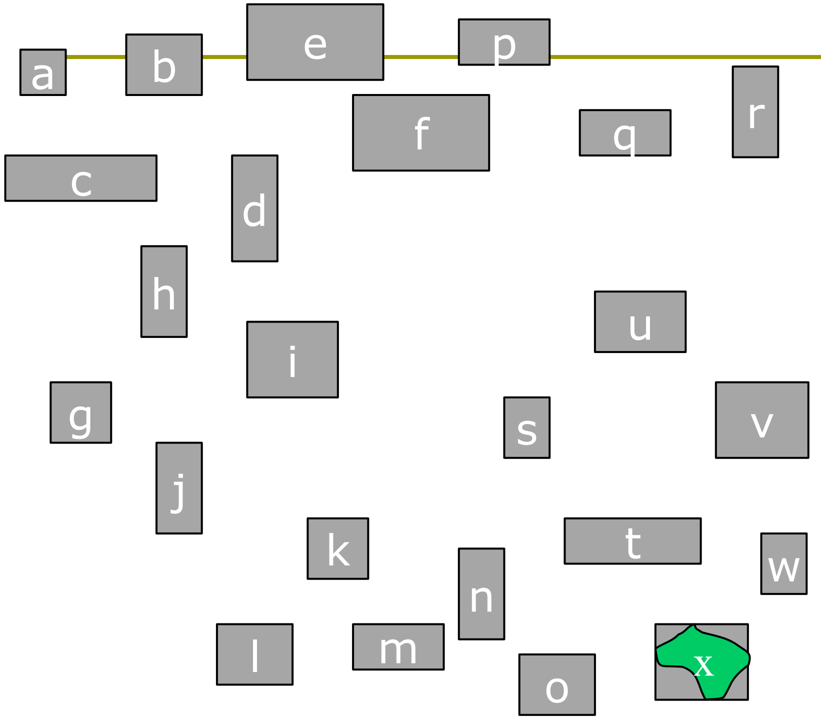
- Groups object MBRs to disk (or memory) blocks hierarchically
- Each group of objects (a block) is a leaf of the tree
- The MBRs of the leaf nodes are grouped to form nodes at the next level
- Grouping is recursively applied at each level until a single group (the root) is formed

The R-tree

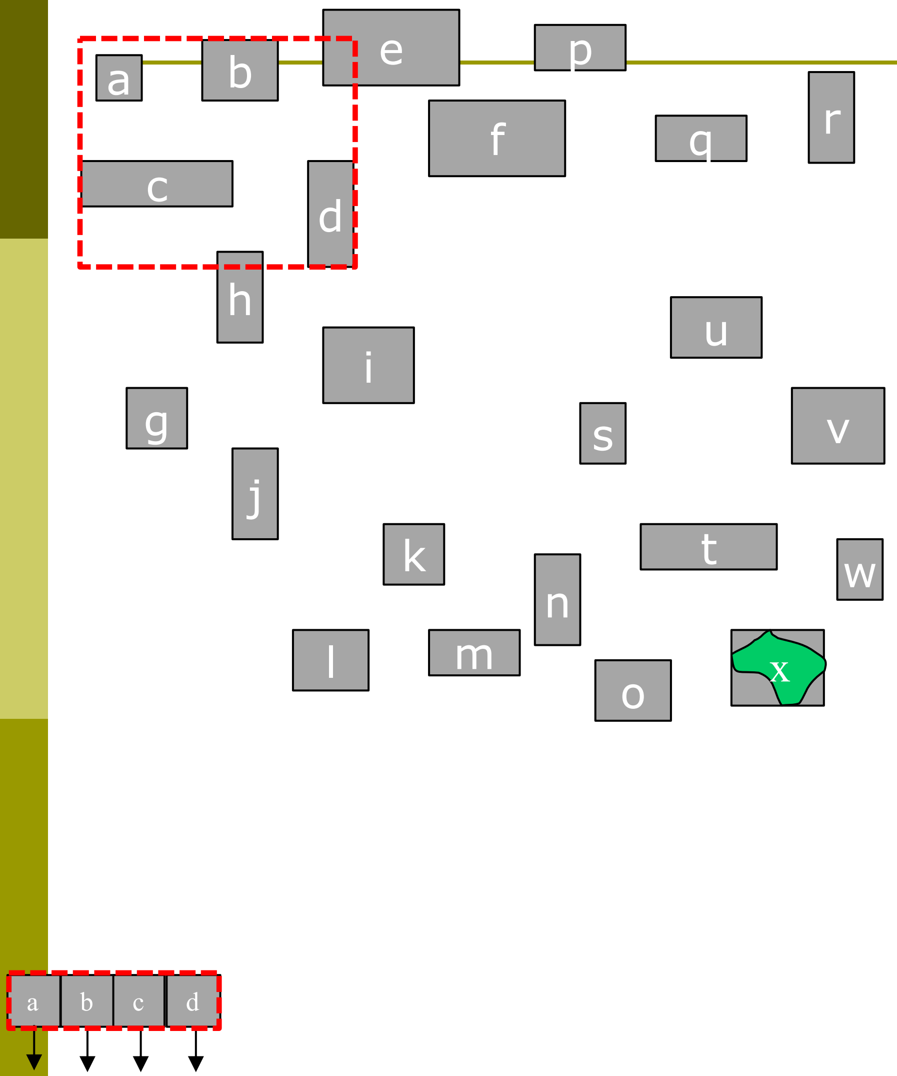
- Leaf node entries: <MBR, object-id>
- Non-leaf node entries: <MBR, ptr>
- The MBR of a non-leaf node entry is the MBR of all entries in the node pointed by it
- Parameters (except root):
  - M (max no of entries per node)
  - m (min no of entries per node)
  - m <= M/2
  - usually m=0.4M
- Root has at least two children
- All leaves in same level (balanced tree)
- 1 node → 1 disk block



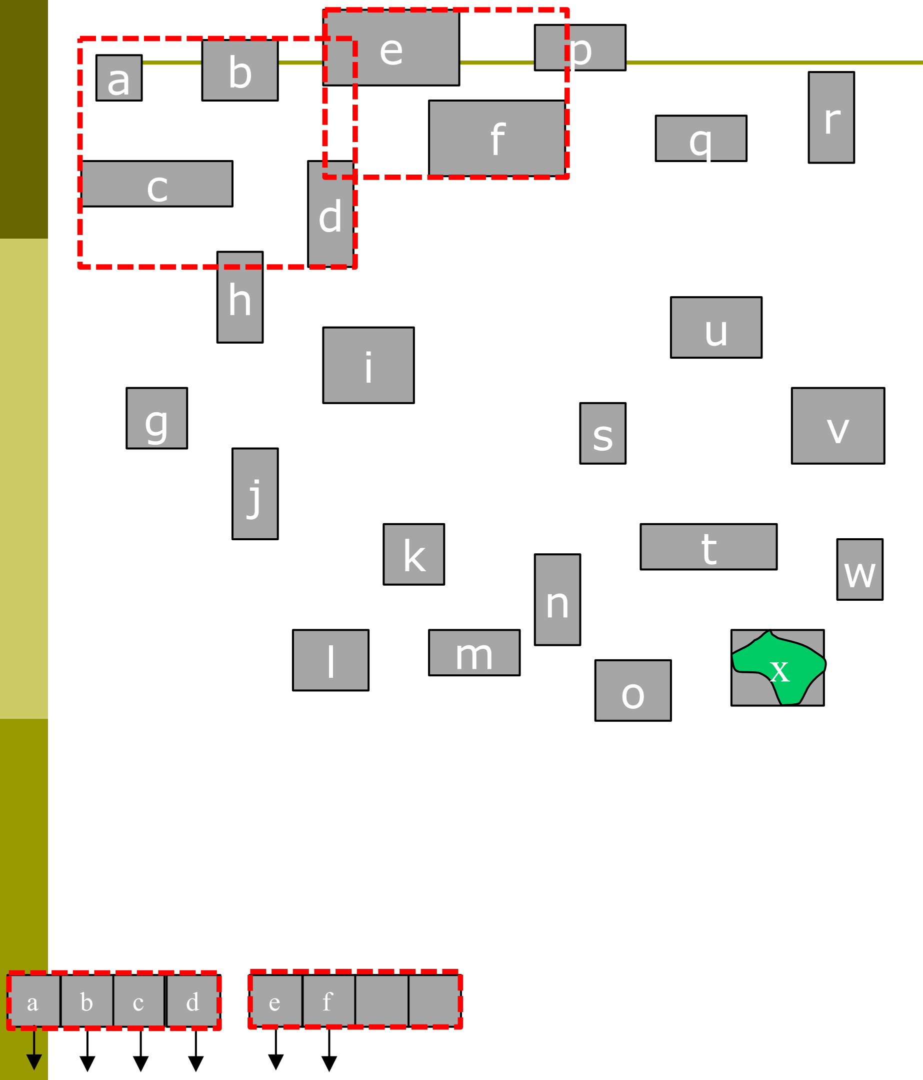
The R-tree - example



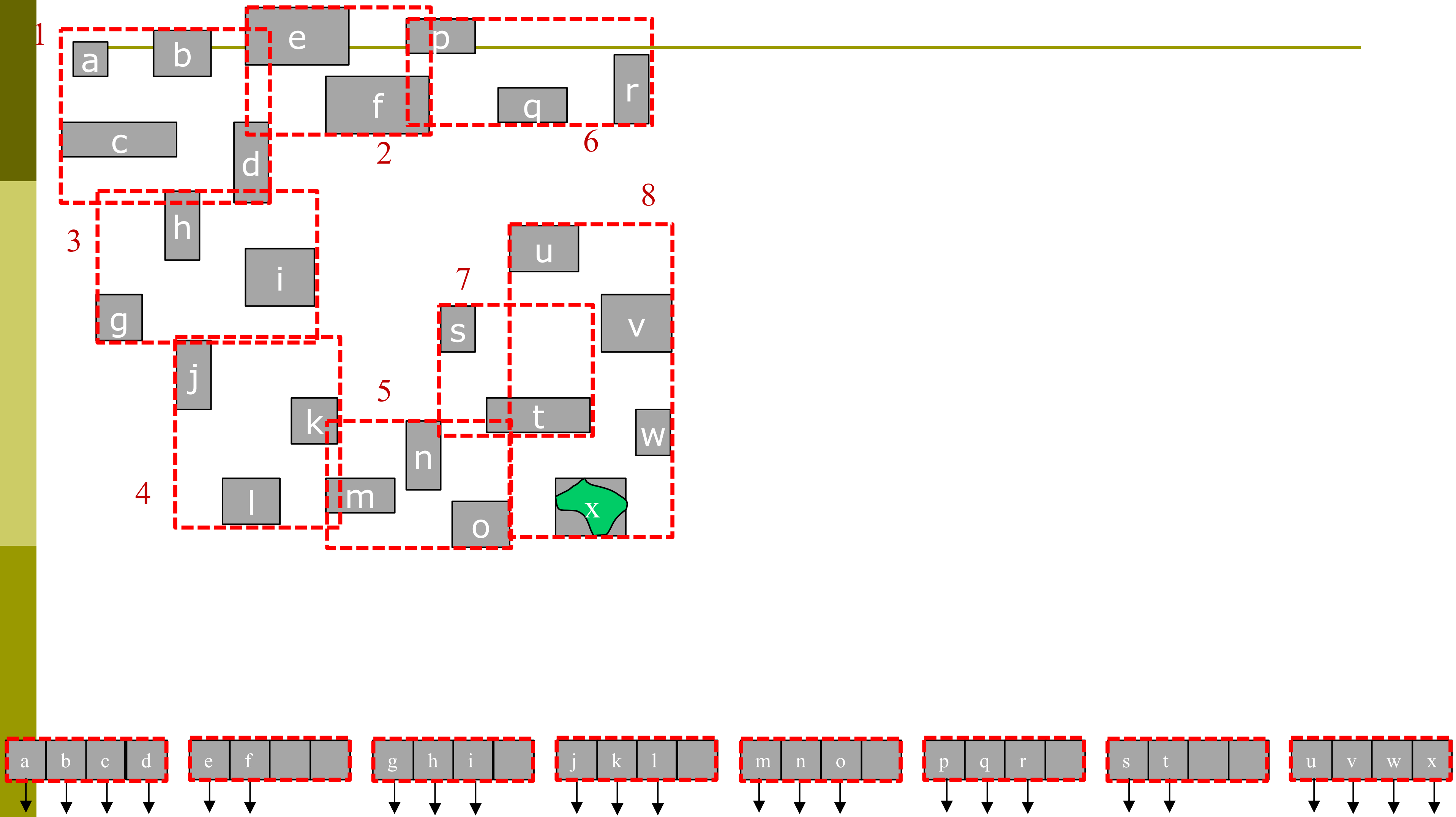
The R-tree - example



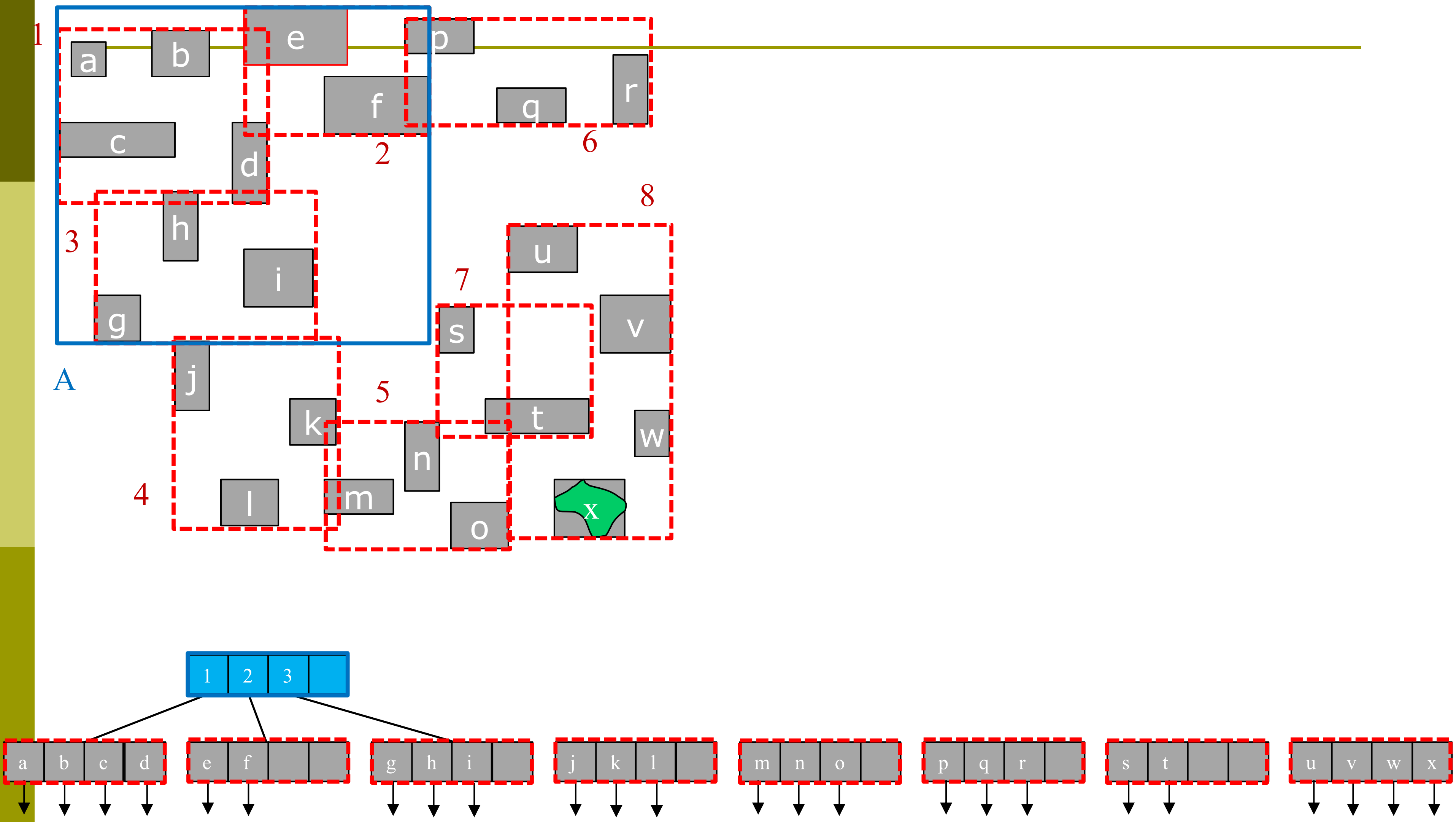
The R-tree - example



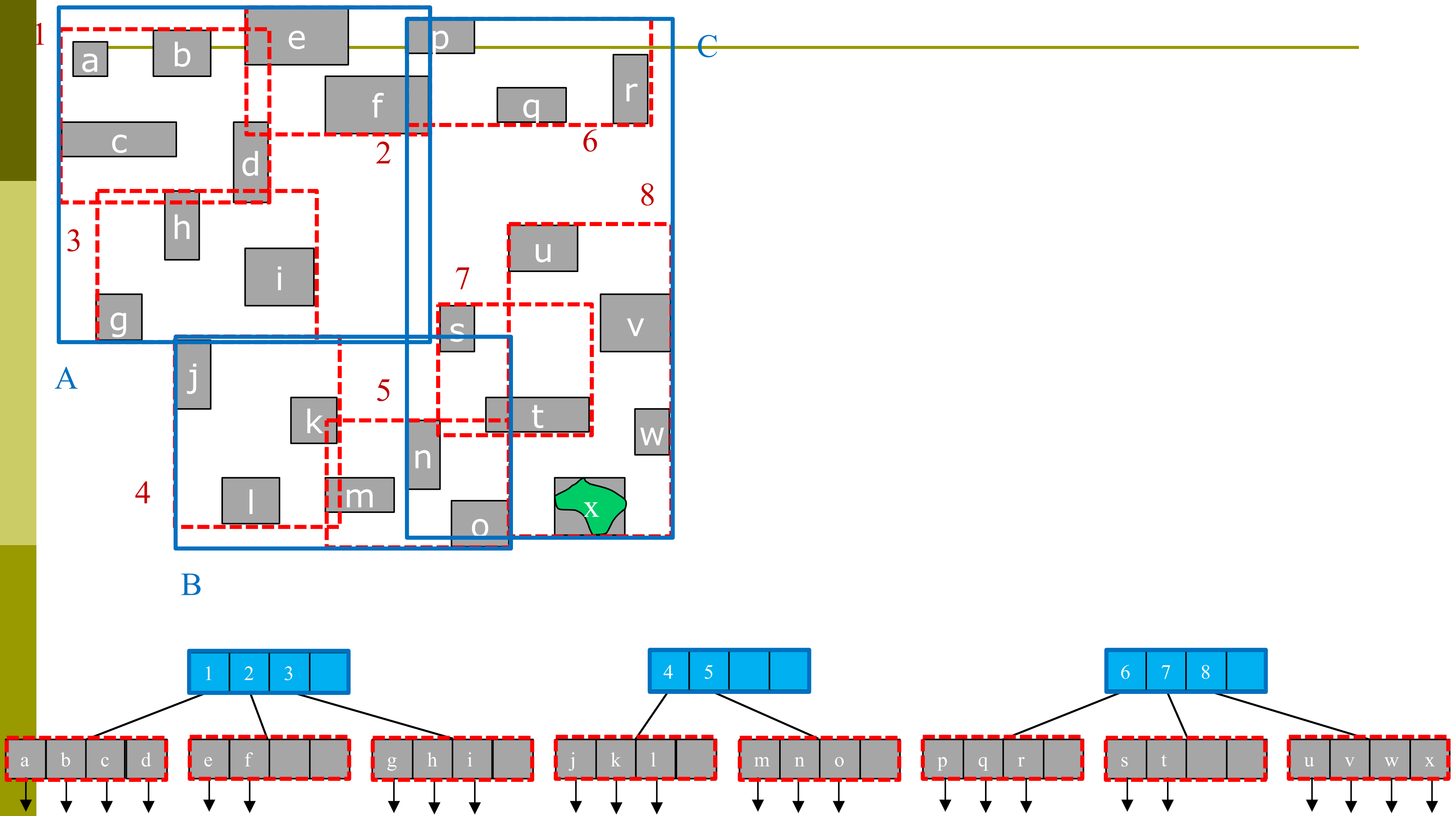
The R-tree - example



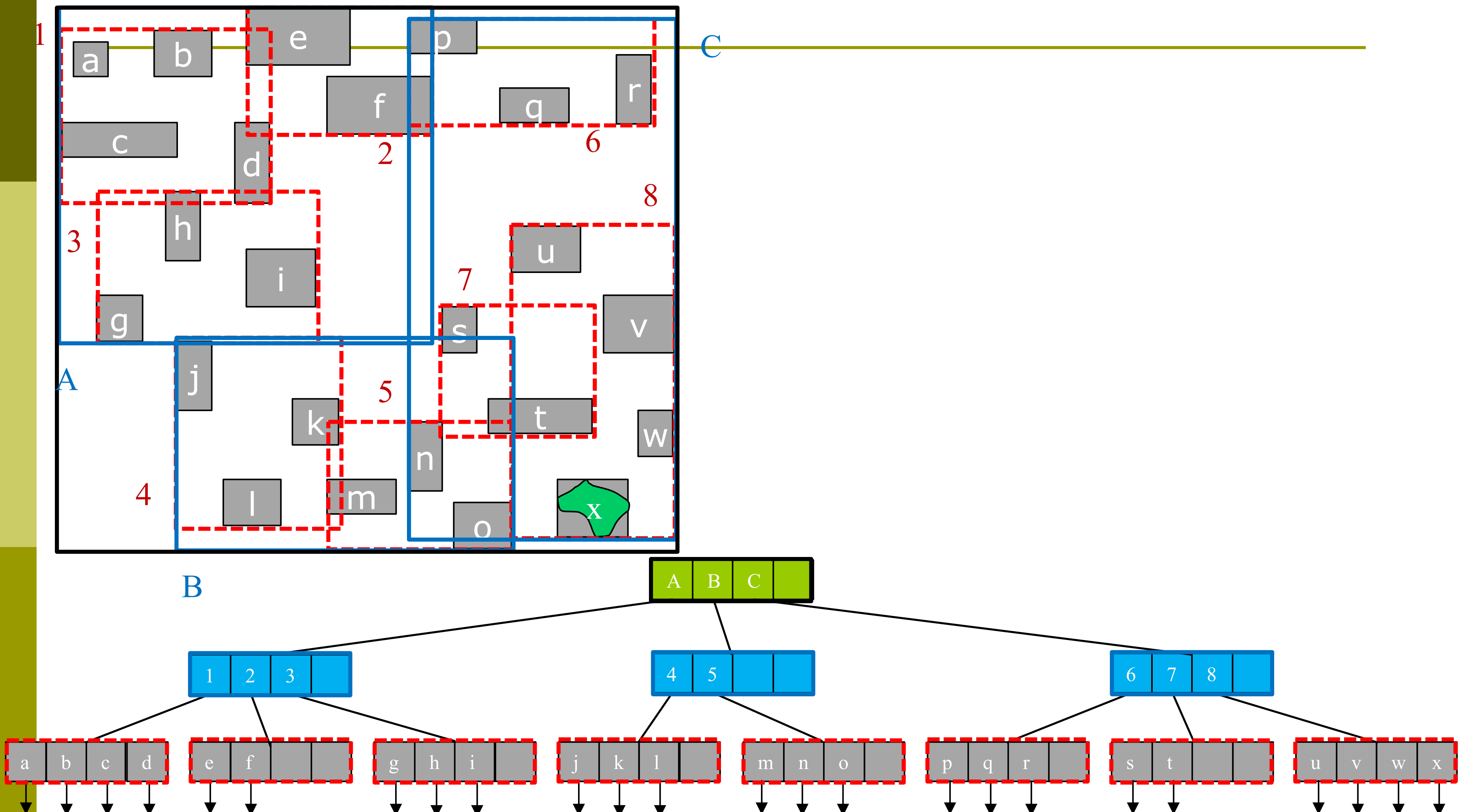
The R-tree - example



The R-tree - example



The R-tree - example



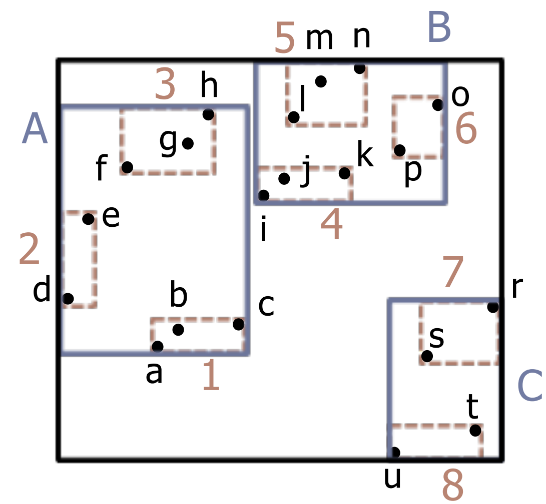
Spatial Query Evaluation

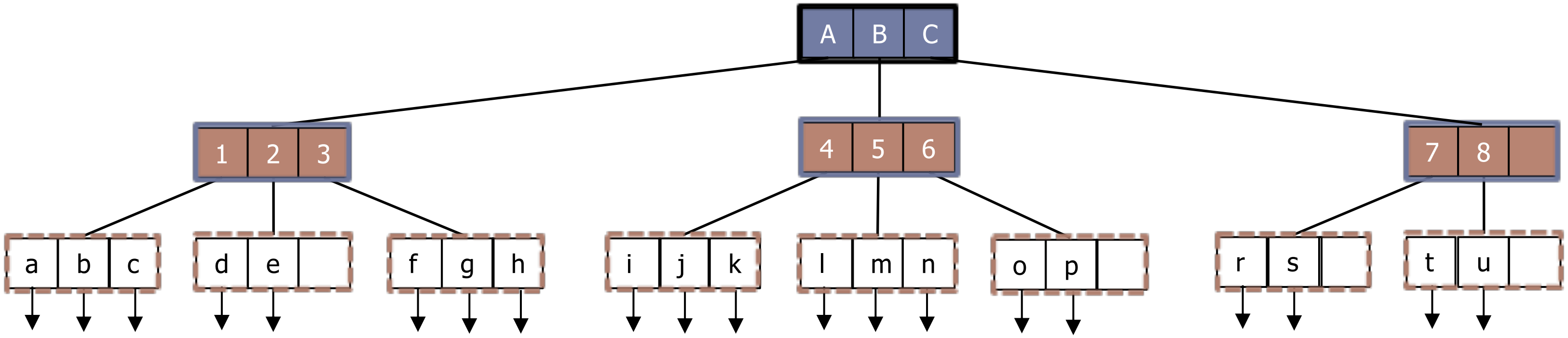
Range searching using an R-tree

- Range_query(query *W*, R-tree node *n*):
  - If n is not a leaf node
    - For each index entry *e* in *n* such that e.MBR intersects *W*
      - visit node *n*′ pointed by *e.ptr*
      - Range_query(*W*, *n*′)
  - If n is a leaf
    - For each index entry *e* in *n* such that e.MBR intersects *W*
      - visit object o pointed by *e.object-id*
      - test range query against exact geometry of o; if o intersects W, report o
- May follow multiple paths during search
- Different search predicates are used for different relationships with *W*.
  - What if we want to find all objects inside W?

Spatial range query

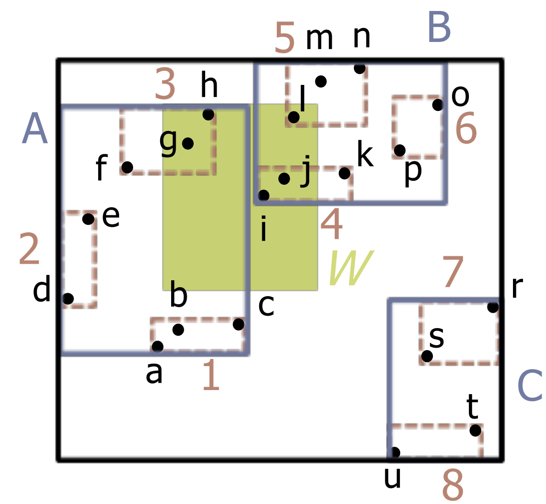
- Traverse tree
  - Top-down
  - Visit nodes whose MBRs qualify predicate

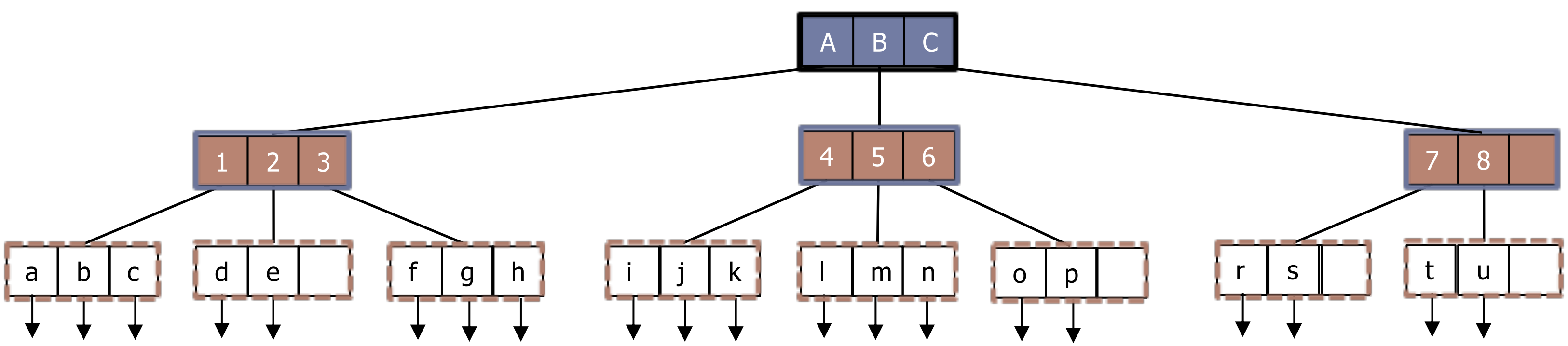




Spatial range query

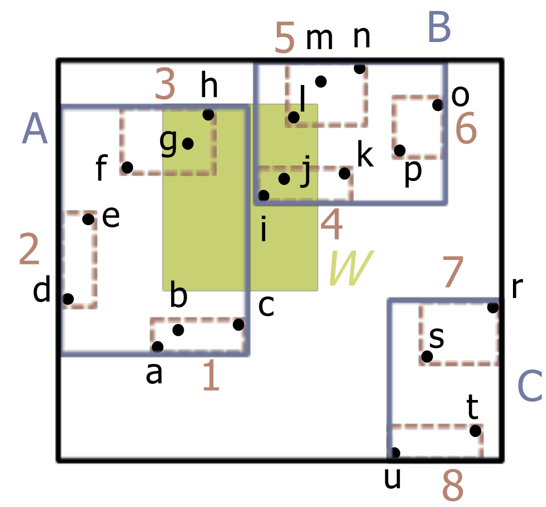
- Traverse tree
  - Top-down
  - Visit nodes whose MBRs qualify predicate
- E.g., find points inside *W*

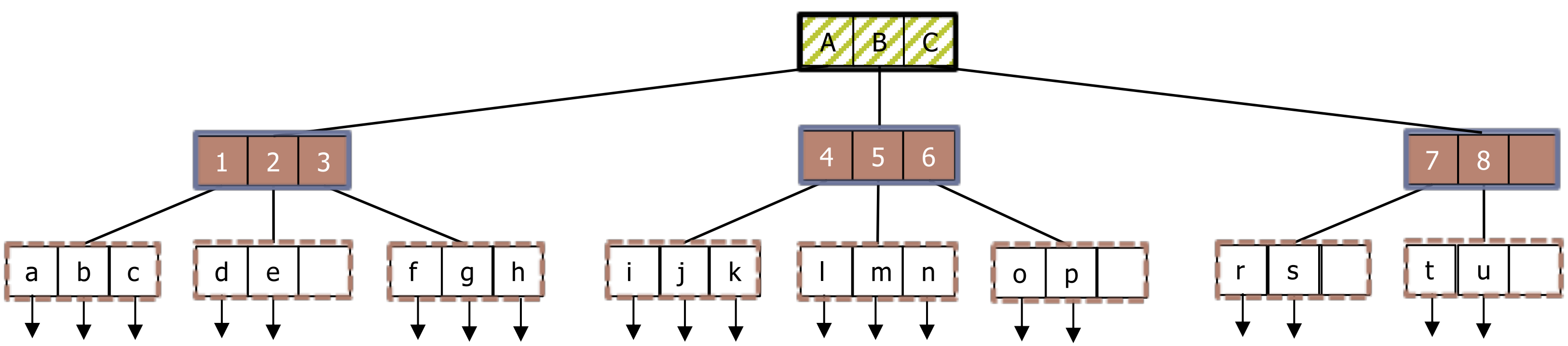




Spatial range query

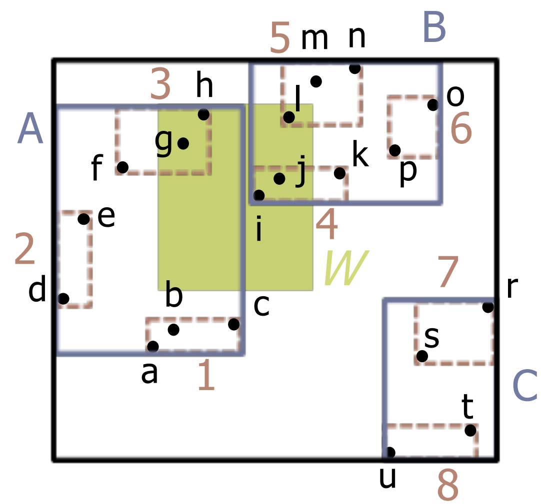
- Traverse tree
  - Top-down
  - Visit nodes whose MBRs qualify predicate
- E.g., find points inside *W*

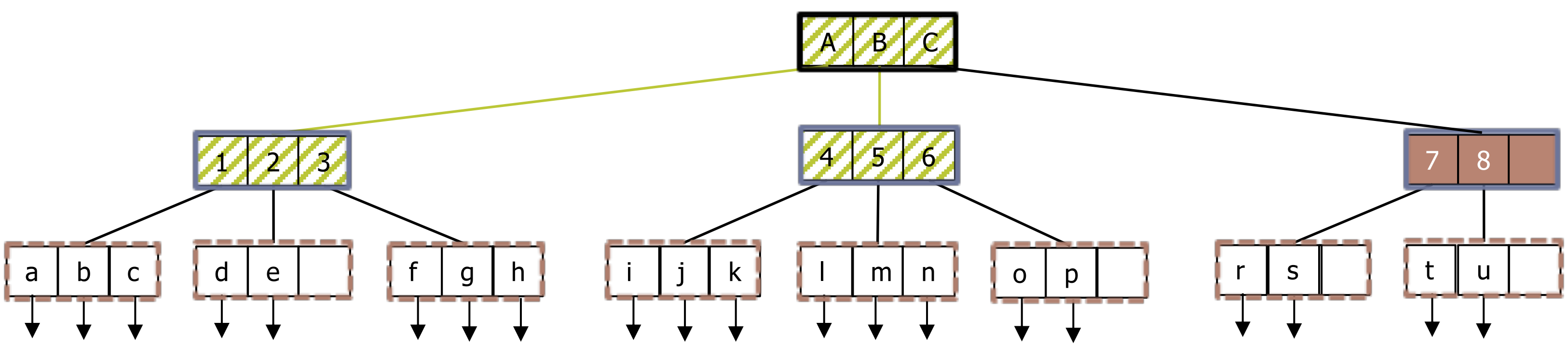




Spatial range query

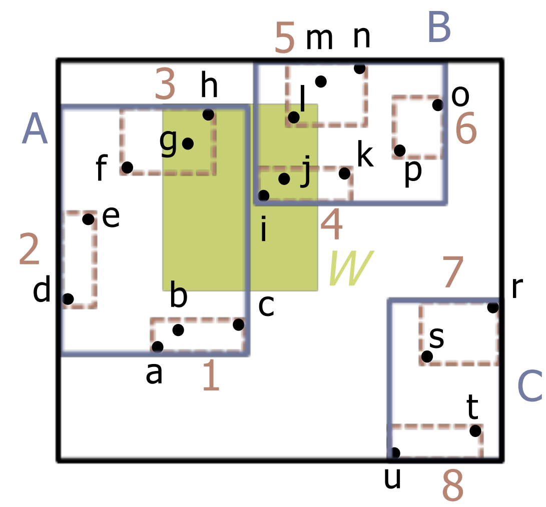
- Traverse tree
  - Top-down
  - Visit nodes whose MBRs qualify predicate
- E.g., find points inside *W*

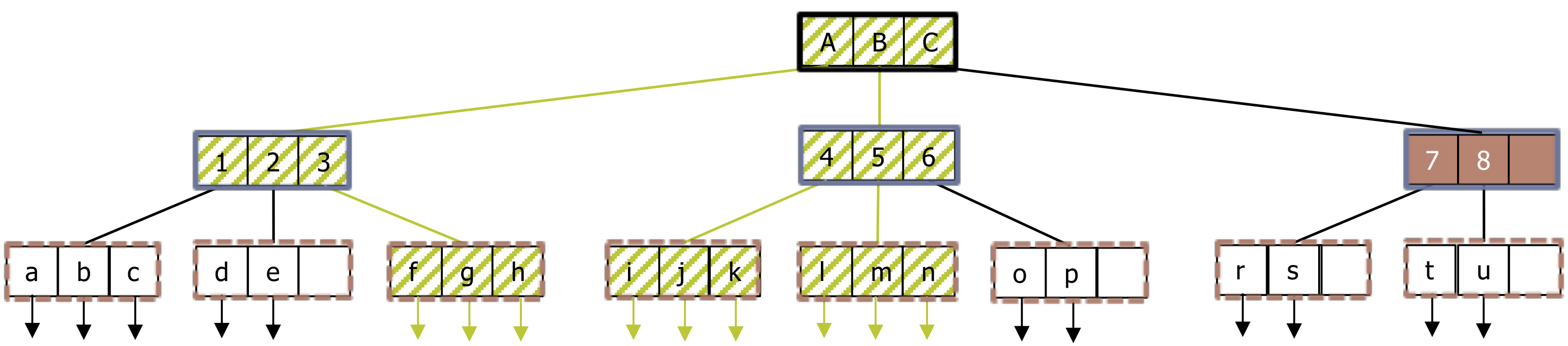




Spatial range query

- Traverse tree
  - Top-down
  - Visit nodes whose MBRs qualify predicate
- E.g., find points inside *W*





R-tree Construction

Construction of the R-tree

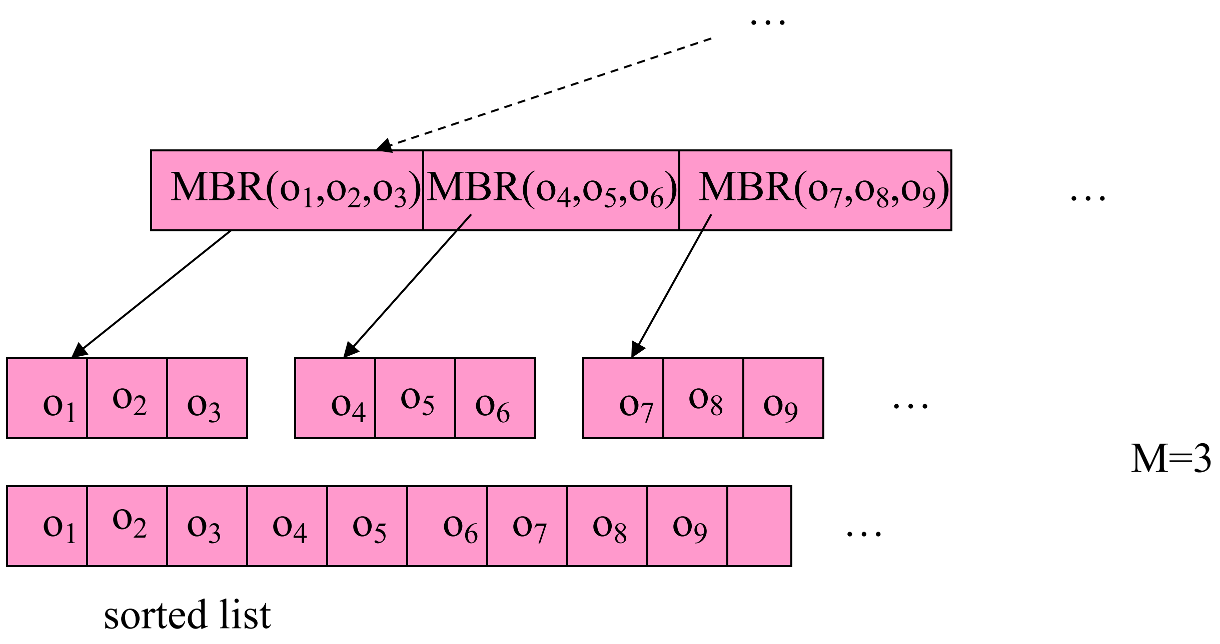
- Dynamically constructed/maintained
- Insertions/deletions interleave with search operations
- Insertion similar to B^+-tree
  - However special optimization algorithms have to be designed for
    - choosing the path where a new MBR is inserted
    - splitting overflown nodes
- Underflows in deletions are handled by
  - deleting the underflown leaf node
  - re-inserting the remaining entries

Bulk-loading R-trees

- Given a static set S of rectangles, build an R-tree that indexes S.
- Method 1: iteratively insert rectangles into an initially empty tree
  - tree reorganization is slow
  - tree nodes are not as full as possible: more space occupied for the tree
- Method 2: bulk-load the rectangles into the tree using some fast (sort or hash-based) process
  - R-tree is built fast
  - good space utilization

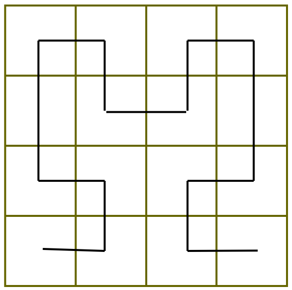
Bulk-loading R-trees

- Method 1: Sort using only one axis
  - sort rectangles using the x-coordinate of their center
  - pack M consecutive rectangles in leaf nodes
  - build tree bottom-up



Bulk-loading R-trees

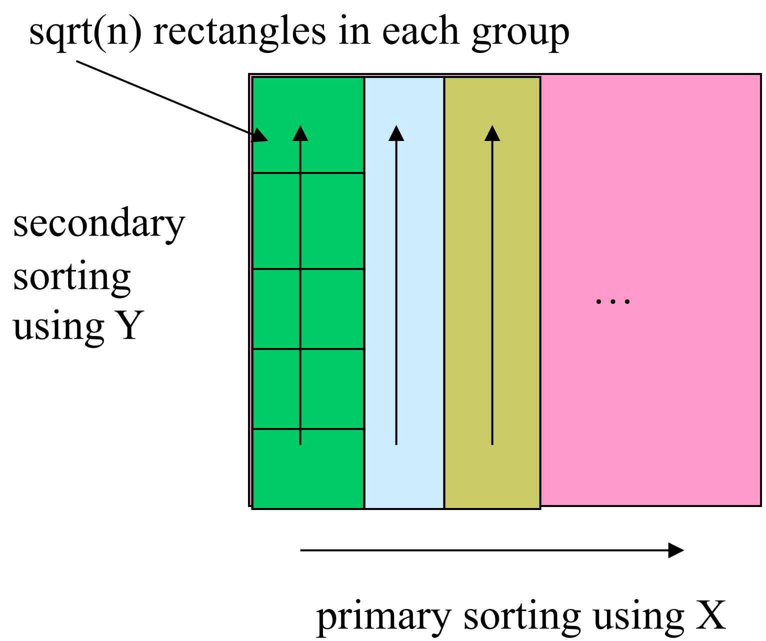
- Method 1 results in leaf nodes that are have long stripes as MBRs
- Method 2: use a space-filling curve to order the rectangles
  - much better structure, but still the nodes have large overlap



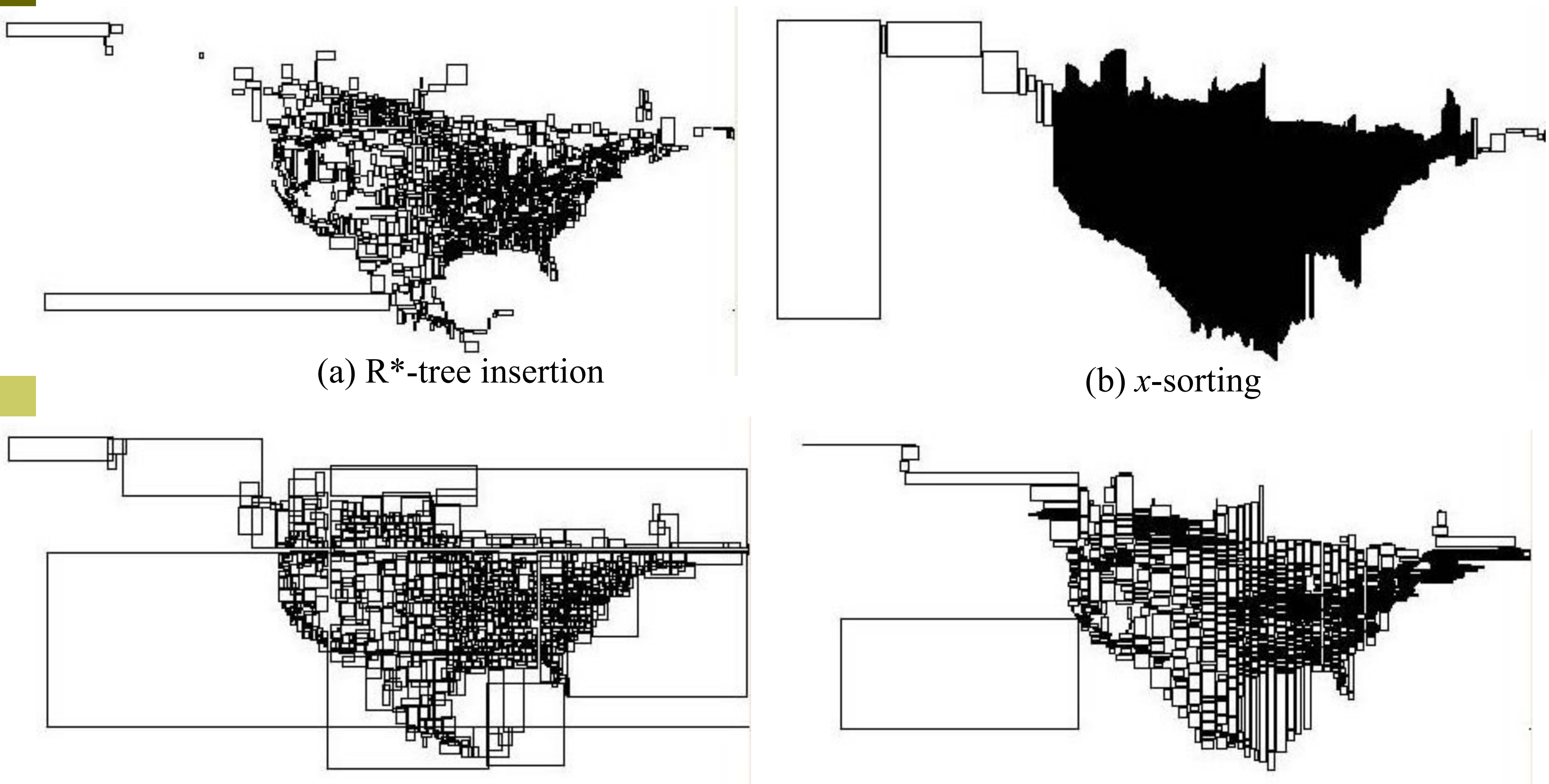
Hilbert-curve

Bulk-loading R-trees

- Method 3: Sort using one axis first and then groups of sqrt(n) rectangles using the other axis
- Usually the best structure compared to bulk-loading methods



R-tree leaf nodes by different construction methods



(a) R*-tree insertion

(b) *x*-sorting

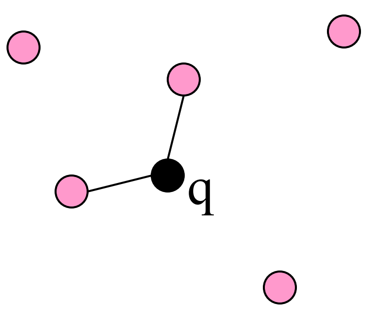
(c) Hilbert sorting

(d) Sort-tile recursive

Nearest Neighbor Queries

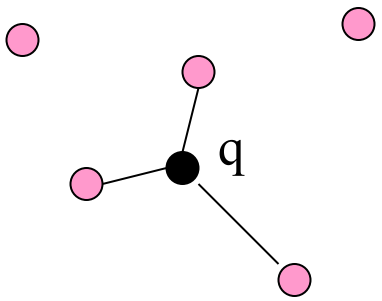
Nearest neighbor search

- Basic problem:
  - Given a spatial relation R and a query object q, find the nearest neighbor of q in R
  - Formally:
    - $NN(q,R) = o \in R: dist(q,o) \leq dist(q,o'), \forall o' \in R$
- Note:
  - We can have more than one NN (with equal minimum distance)
  - Break ties arbitrarily



Nearest neighbor search

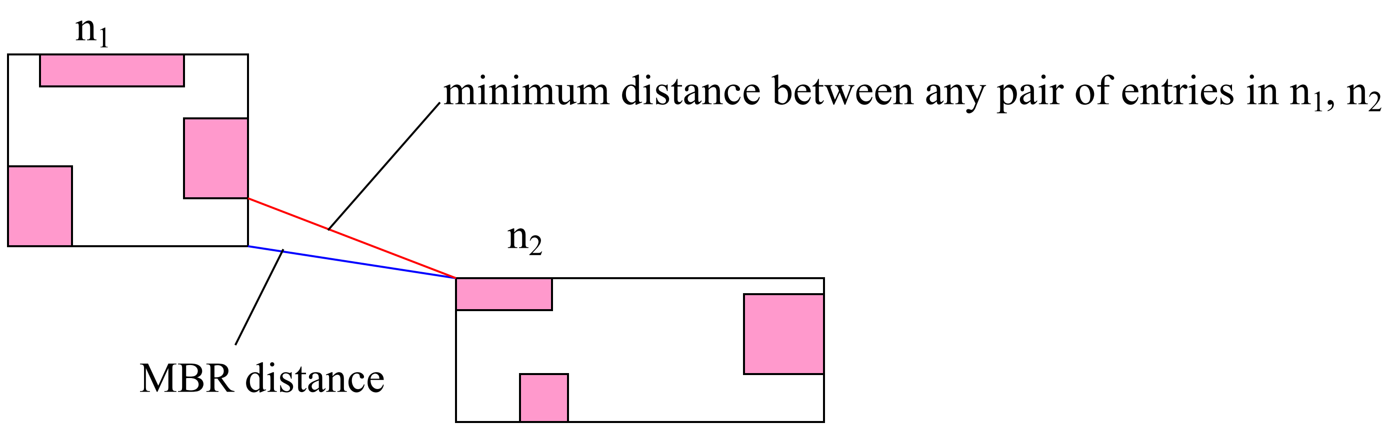
- Generalized problem:
  - Given a spatial relation R, a query object q, and a number k <|R|, find the k-nearest neighbors of q in R
  - Formally:
    - $NN(q,k,R) = S \subset R : |S| = k, dist(q,o) \leq dist(q,o'), \forall o \in S \; \forall o' \in R-S$
- Note:
  - We can have more than one k-NN sets (with multiple possible equidistant furthest points in them)
- Simplification
  - NN (and k-NN) operations return any NN (and k-NN sets)
  - We usually focus on point-sets



3-NN(q,R)

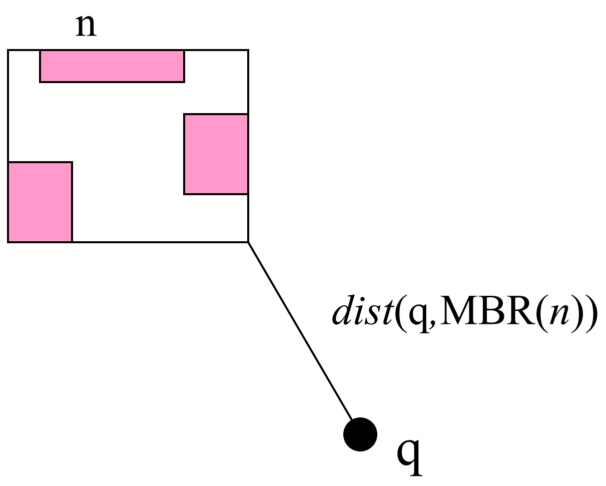
Distance measures and MBRs

- Distances between R-tree node MBRs lower-bound the distances between the entries in them
  - $dist(\mathrm{MBR}(n_i),\mathrm{MBR}(n_j)) \leq dist(e_i.\mathrm{MBR}, e_j.\mathrm{MBR})$, $\forall\ e_i \in n_i, e_j \in n_j$



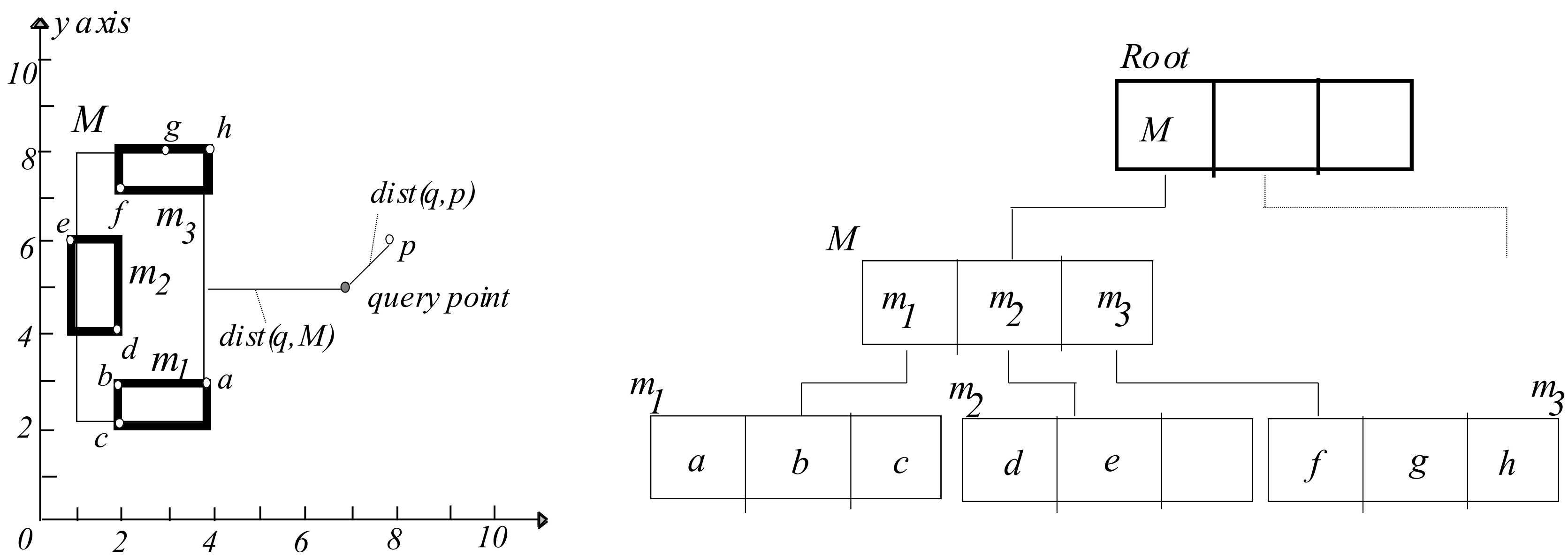
Distance measures and MBRs

- The distance between a query object q and an R-tree node MBR lower-bounds the distances between q an the objects indexed under this node
  - *dist*(q,MBR(*n*)) ≤ *dist*(q,o) ∀ *o* indexed under *n*



Using MBR distances to guide/prune search in an R-tree

- Problem: find the NN of q
- Do we need to look for it in node M if we know dist(q,p)?



Depth-first NN search using an R-tree

- Start from the root and visit the first entry.
- Continue recursively, until a leaf node n_l is visited.
- Find the NN of q in n_l.
- Continue visiting other nodes after backtracking as long there are nodes closer to q than the current NN.
  - Do not visit a node, if the nearest distance between q and the node's MBR exceeds the current dist(q,o_{NN})

Depth-first NN search using an R-tree

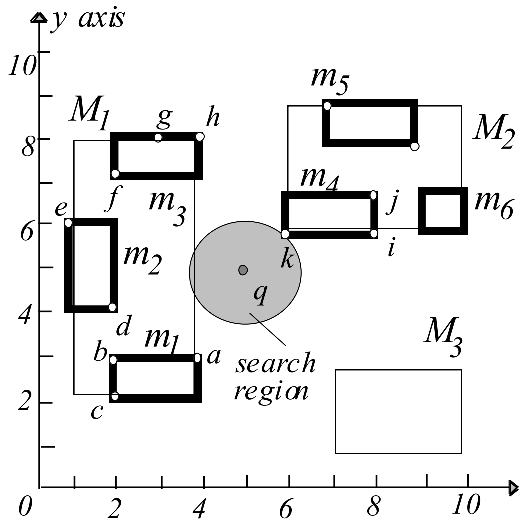
- Recursive function (for data points only)

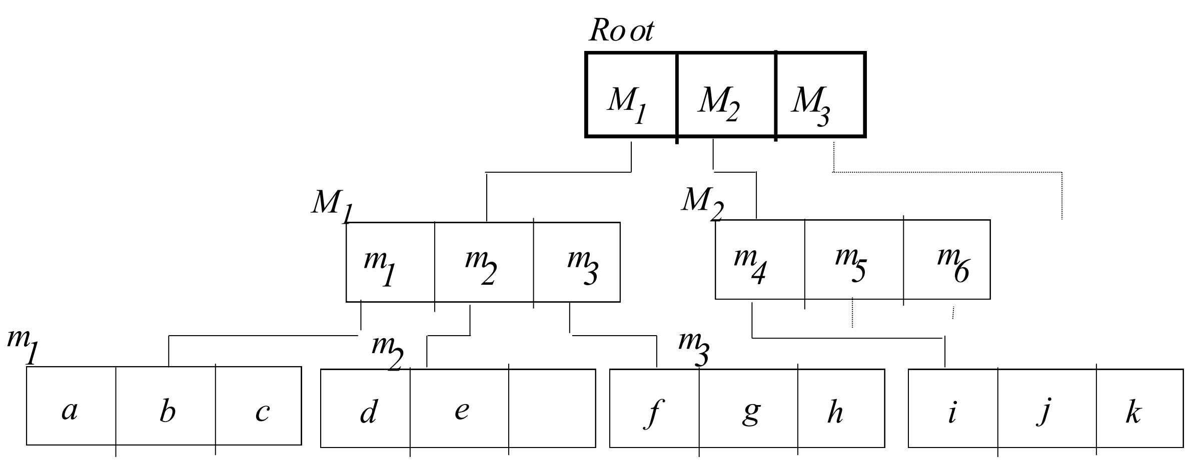
Initially: n is tree's root

Initially:
- o_{NN}=null
- $dist(q,o_{NN}) = \infty$

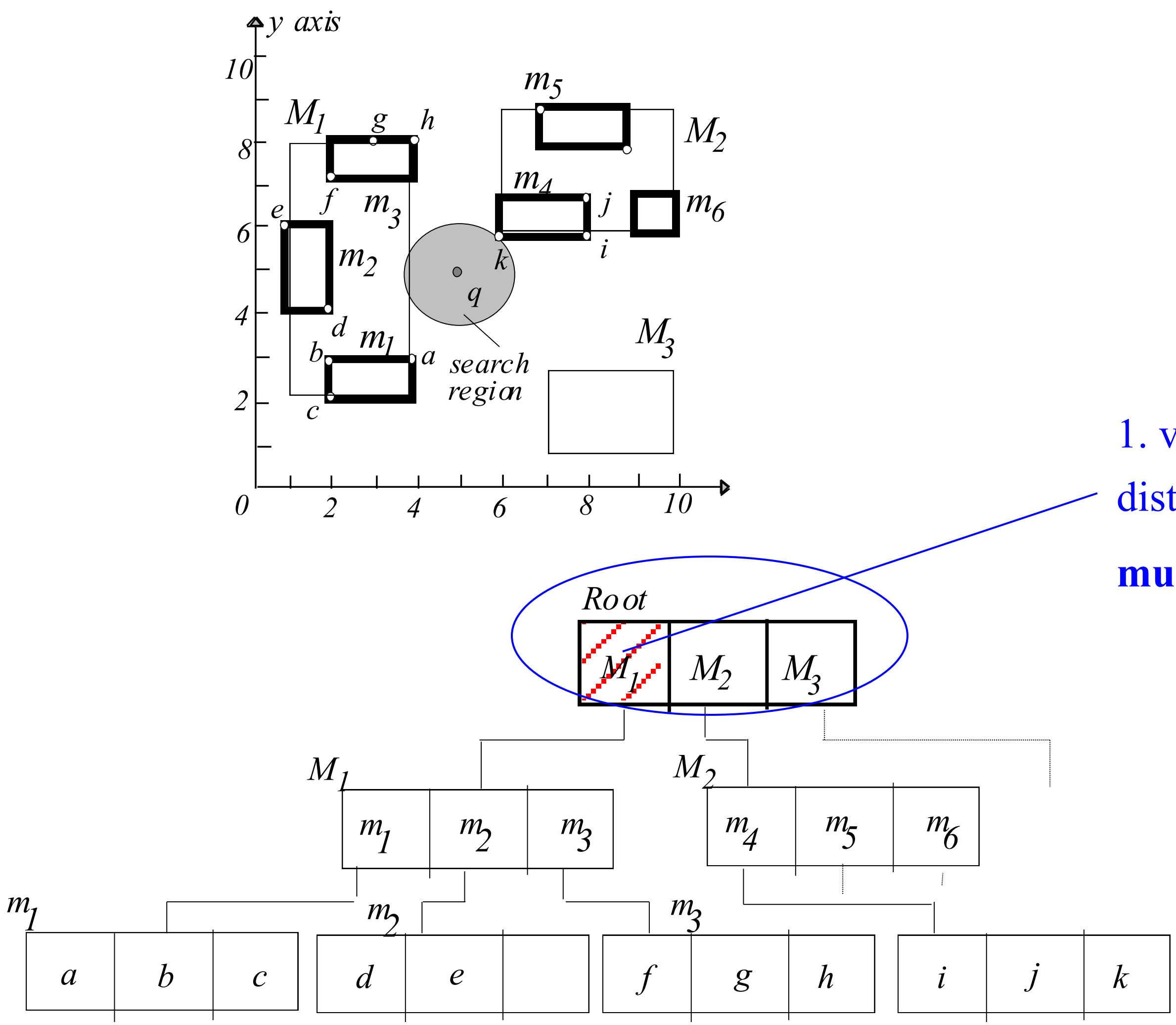
```
DFNN(query point q, node n, point o_NN)
  if n is a leaf node then
     for each entry e in n do
         if dist(q,e)<dist(q,o_NN) then
              oNN = e  // found closest point than current NN
  else
     for each entry e in n do
         if dist(q,e.MBR)<dist(q,o_NN) then
             DFNN(q, e.ptr, o_NN) // recursive call for node pointed by e
```

Depth-first NN search using an R-tree





Depth-first NN search using an R-tree



o_{NN} = NULL

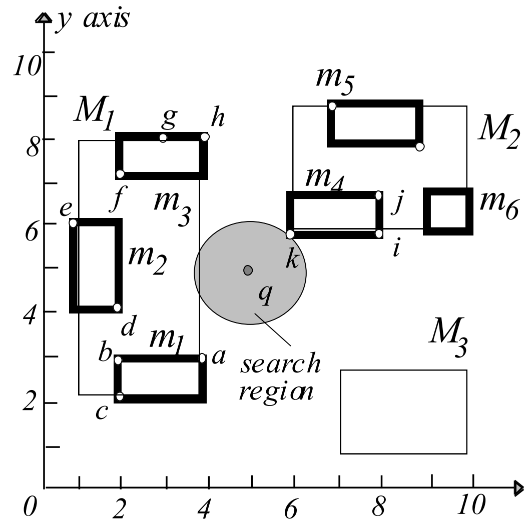
$dist(q,o_{NN}) = \infty$

1. visit root

$dist(q,M_1) < dist(q,oNN)$

must visit node M_1

Depth-first NN search using an R-tree



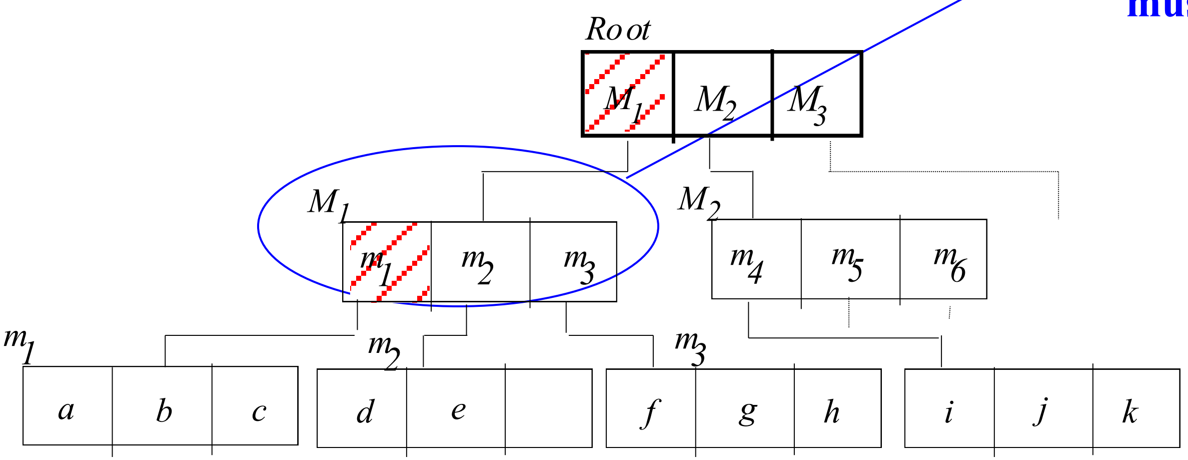
o_{NN} = NULL

$dist(q,o_{NN})= \infty$

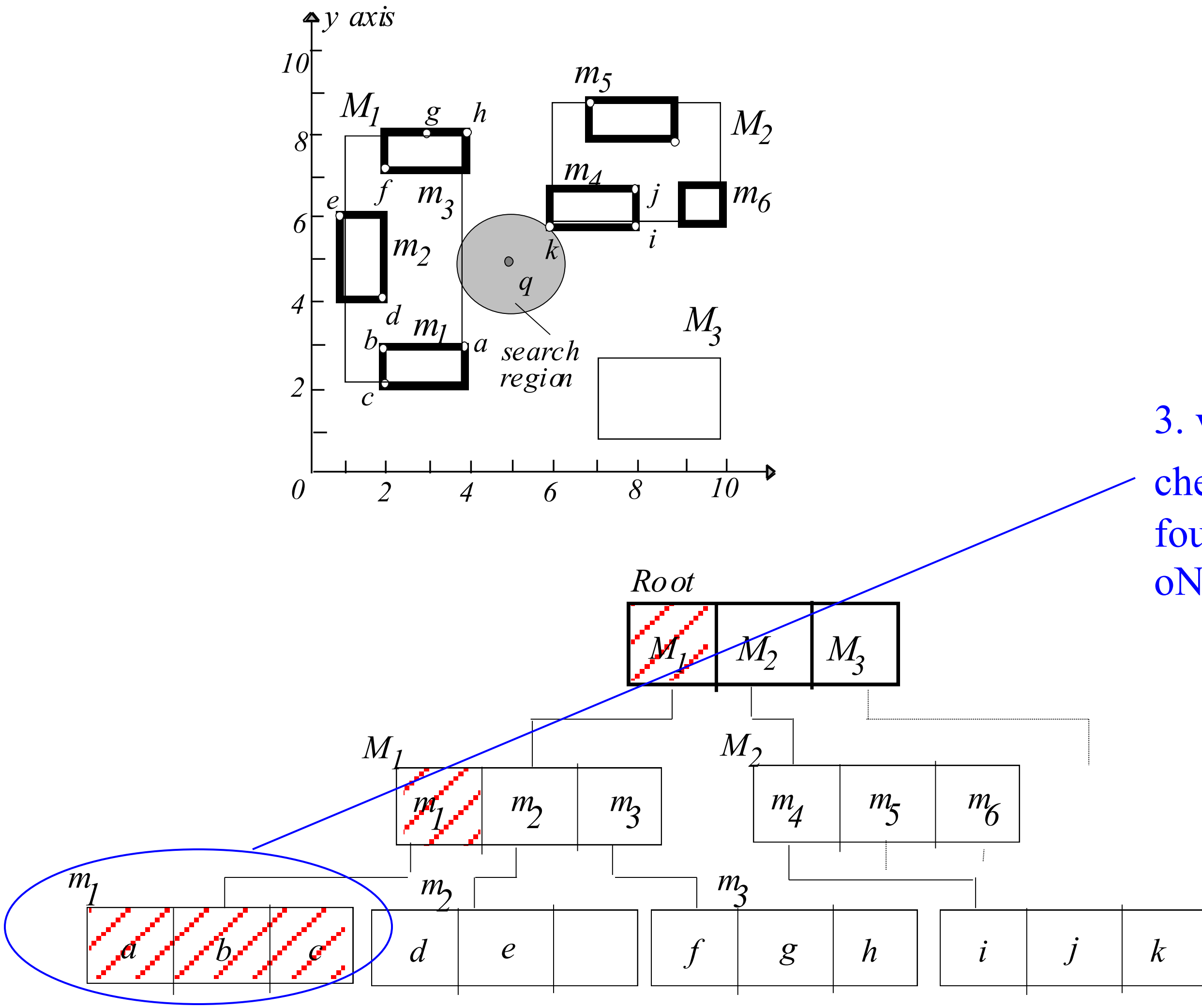
2. visit M_1

$dist(q,m_1)<dist(q,oNN)$

must visit node m_1



Depth-first NN search using an R-tree



o_{NN} = NULL

$dist(q,o_{NN}) = \infty$

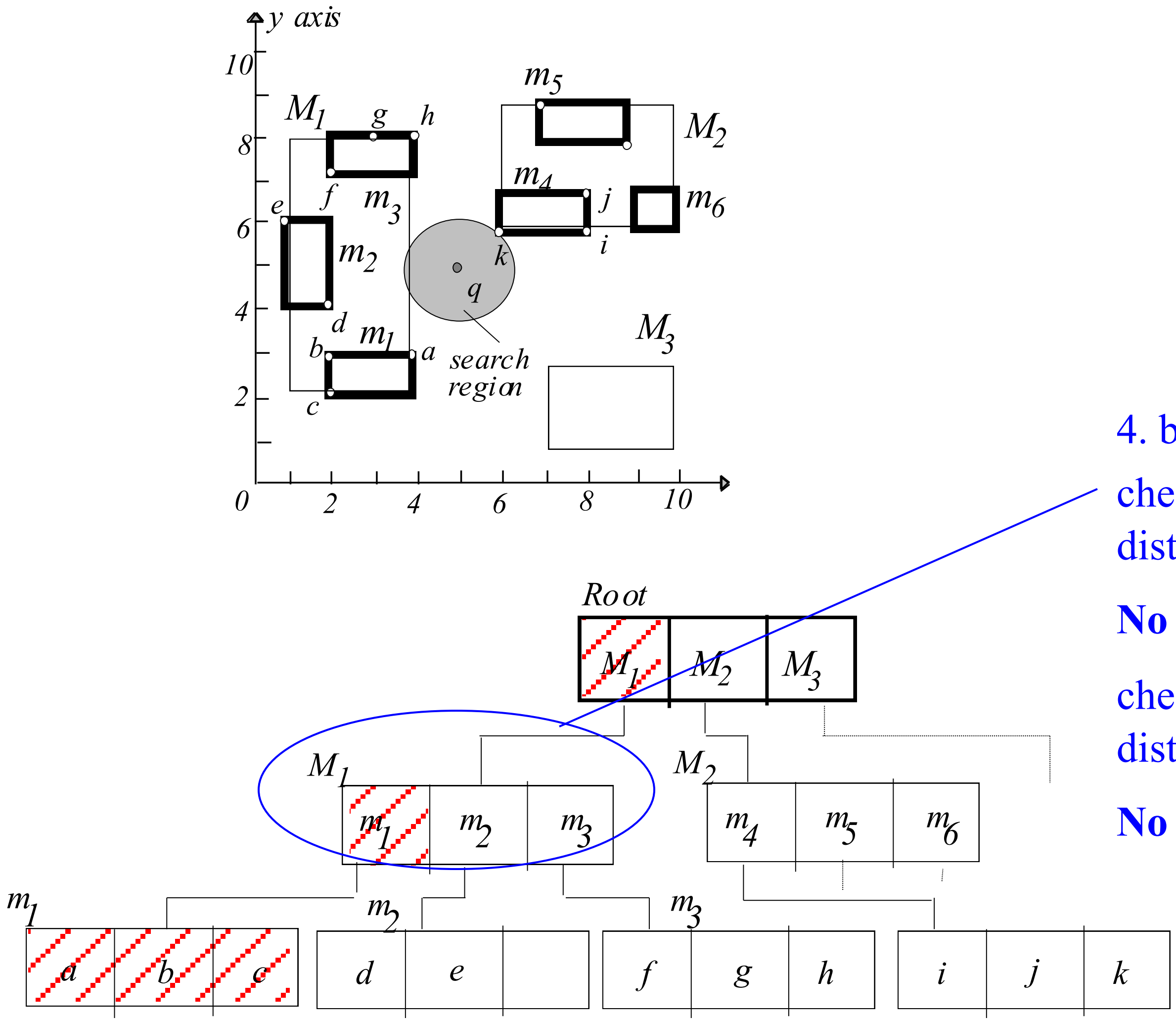
3. visit m_1

check a,b,c

found new NN:

oNN = a, $dist(q,o_{NN}) = \sqrt{5}$

Depth-first NN search using an R-tree



$o_{NN} = a$

$dist(q,o_{NN}) = \sqrt{5}$

4. backtrack to M_1

check m_2

$dist(q,m_2) = 3 \geq \sqrt{5}$:

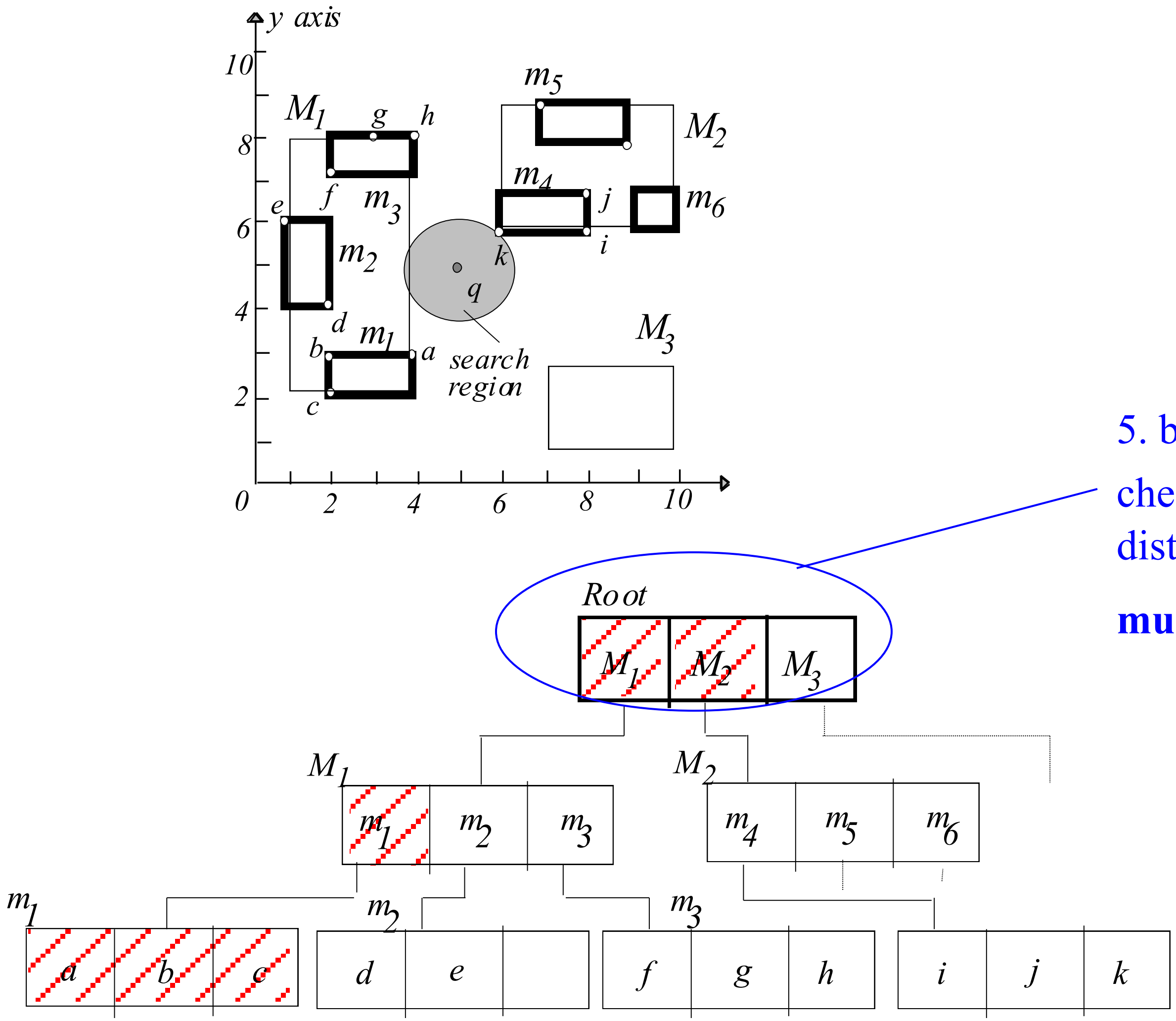
No need to visit node m_2

check m_3

$dist(q,m_3) = \sqrt{5} \geq \sqrt{5}$:

No need to visit node m_3

Depth-first NN search using an R-tree



$o_{NN} = a$

$dist(q,o_{NN}) = \sqrt{5}$

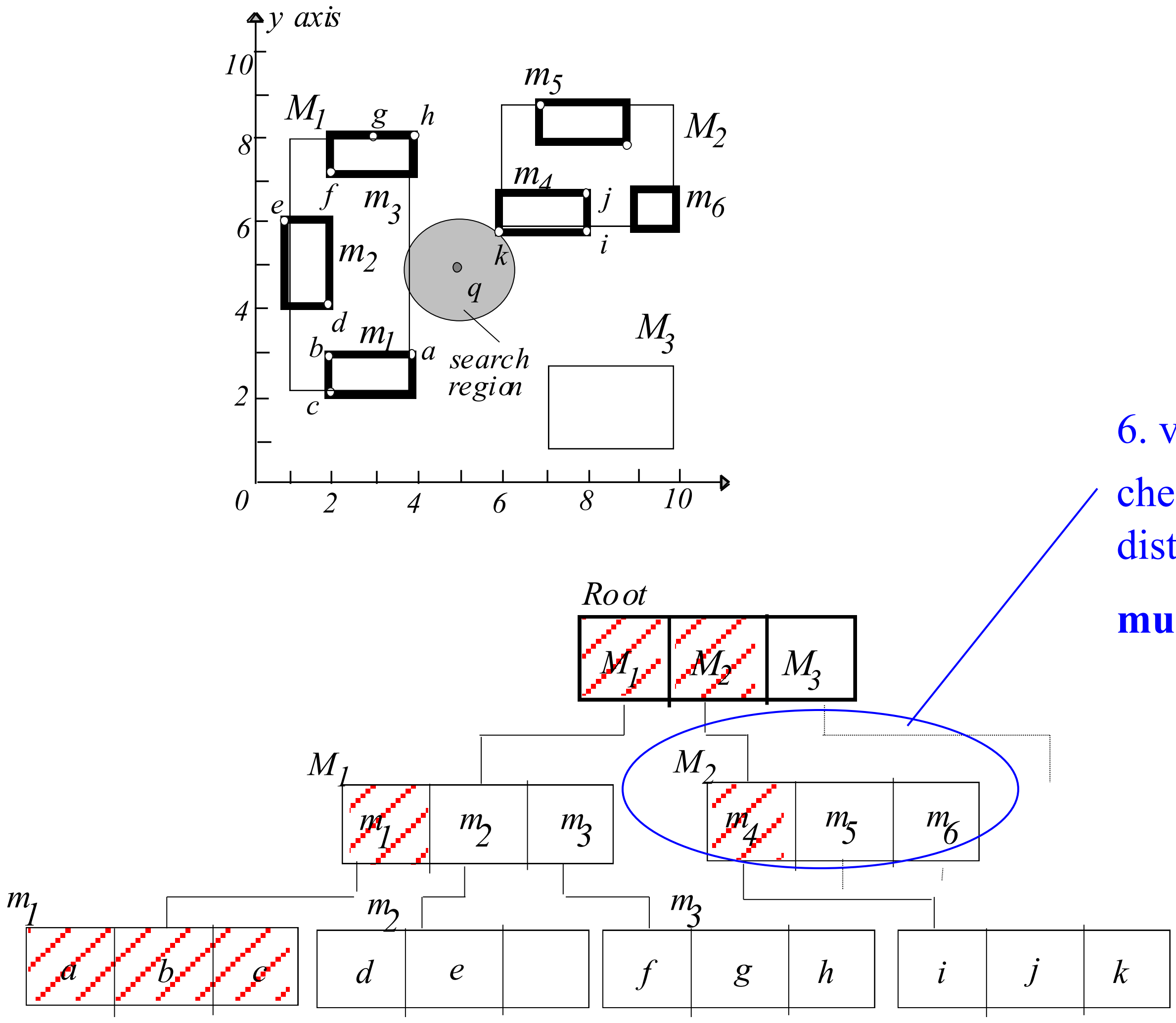
5. backtrack to root

check M_2

$dist(q,M_2) = \sqrt{2} < \sqrt{5}$:

must visit node M_2

Depth-first NN search using an R-tree



$o_{NN} = a$

$dist(q,o_{NN}) = \sqrt{5}$

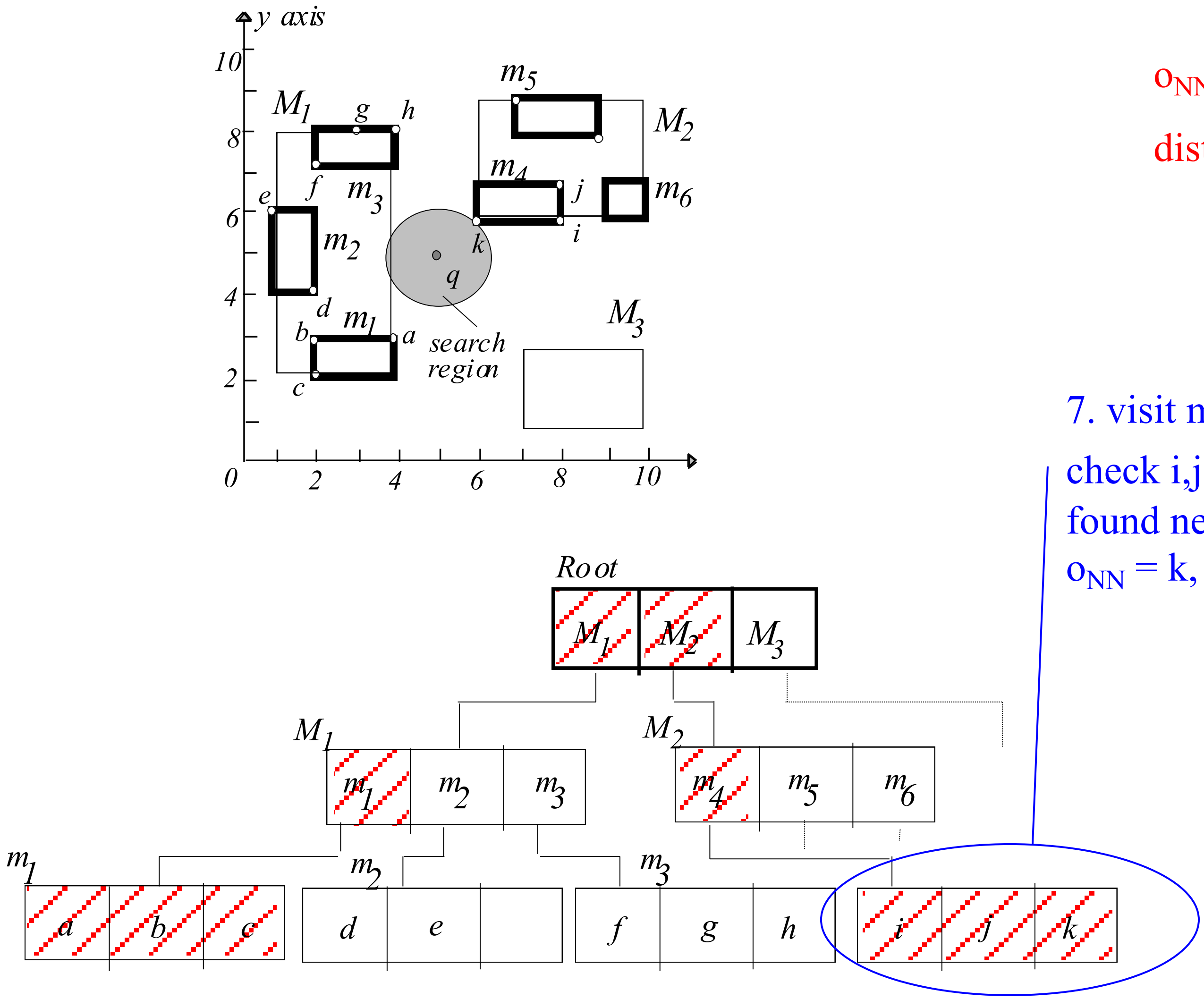
6. visit M_2

check m_4

$dist(q,m_4) = \sqrt{2} < \sqrt{5}$:

must visit node m_4

Depth-first NN search using an R-tree



$o_{NN} = a$

$dist(q, o_{NN}) = \sqrt{5}$

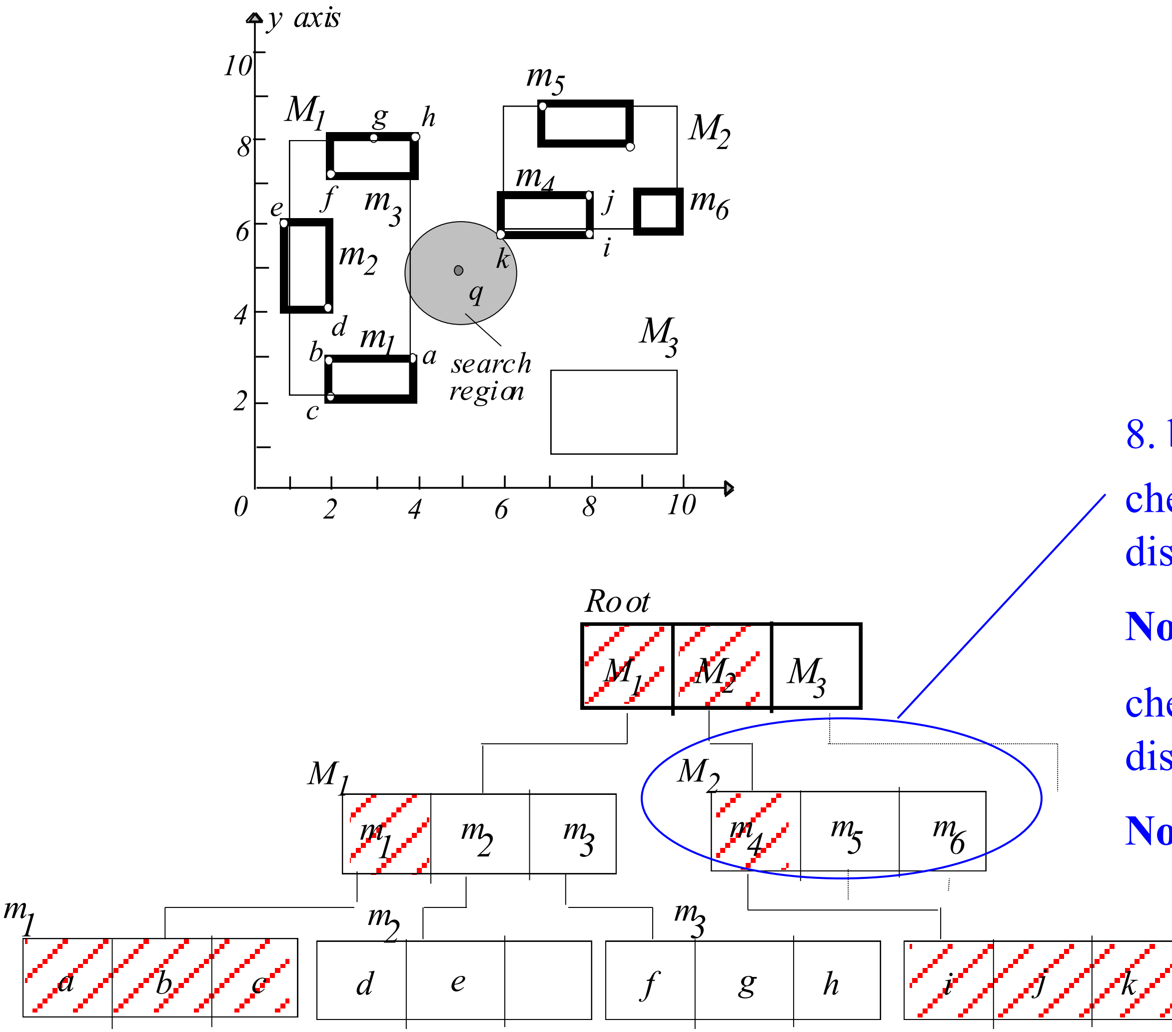
7. visit m_4

check i,j,k

found new NN:

$o_{NN} = k$, $dist(q, o_{NN}) = \sqrt{2}$

Depth-first NN search using an R-tree



$o_{NN} = k$

$dist(q, o_{NN}) = \sqrt{2}$

8. backtrack to M_2

check m_5

$dist(q,m_5) \geq \sqrt{2}$:

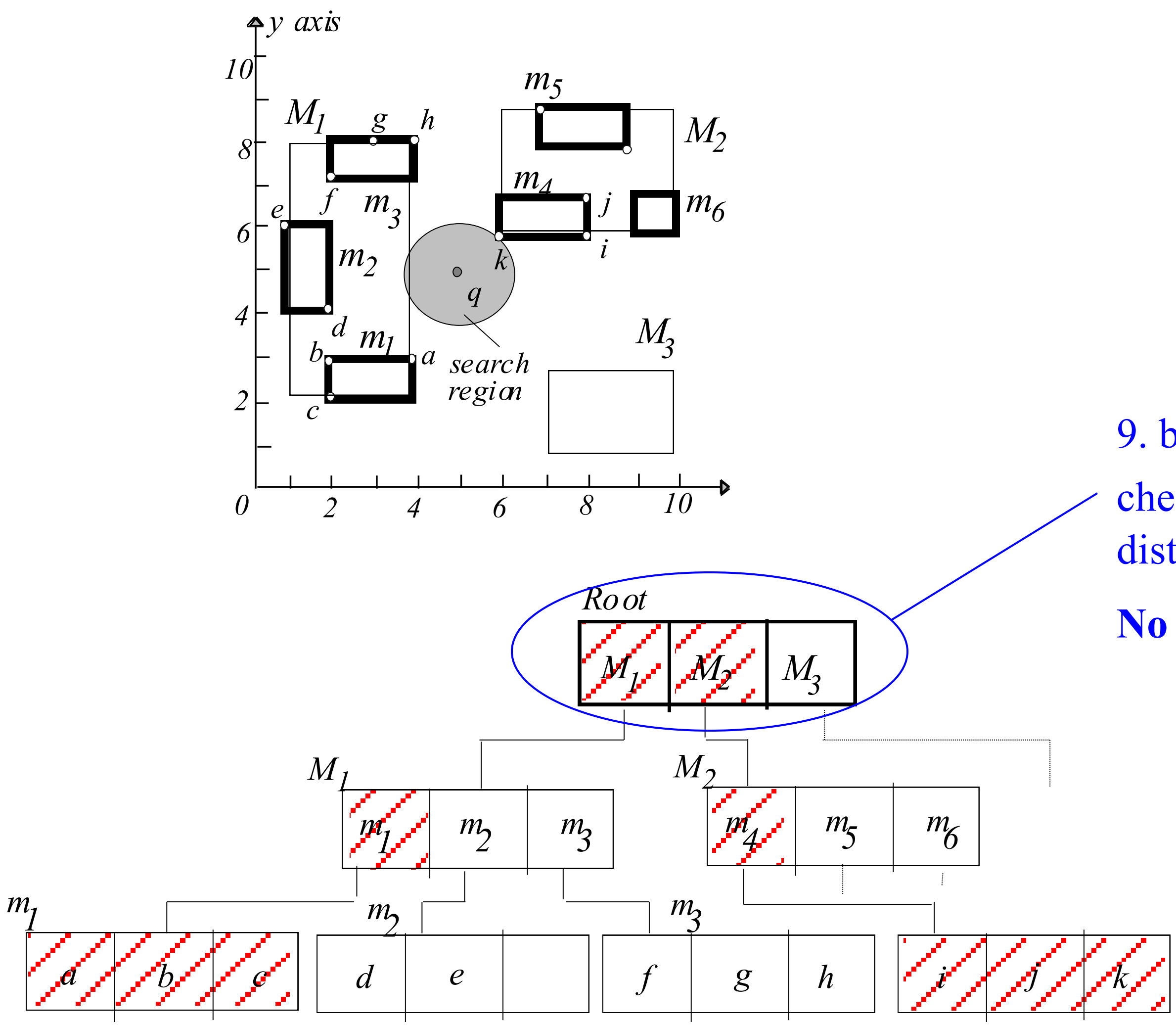
No need to visit node m_5

check m_6

$dist(q,m_6) \geq \sqrt{2}$:

No need to visit node m_6

Depth-first NN search using an R-tree



$o_{NN} = k$

$dist(q,o_{NN}) = \sqrt{2}$

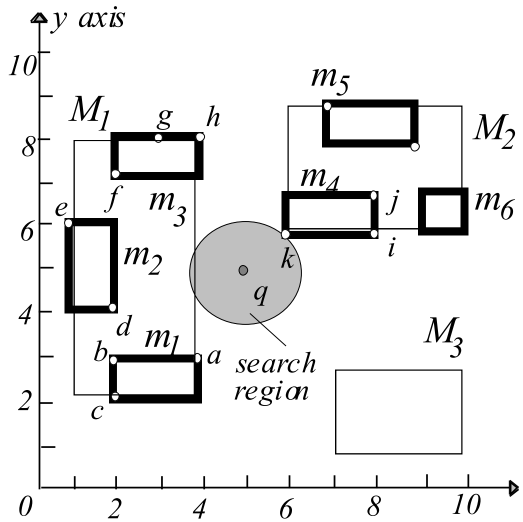
9. backtrack to root

check M_3

$dist(q,M_3) \geq \sqrt{2}$:

No need to visit node M_3

Depth-first NN search using an R-tree



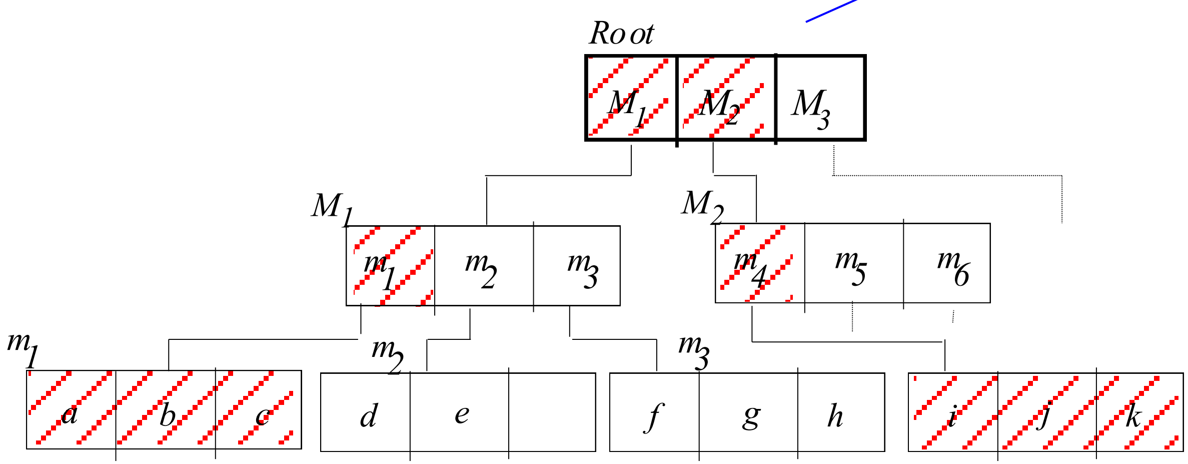
$o_{NN} = k$

$dist(q, o_{NN}) = \sqrt{2}$

10. backtrack from root

Algorithm terminates

oNN =k with dist(q,oNN)= √2 found



Notes on Depth-first NN search

- Large space can be pruned by avoiding visiting R-tree nodes and their sub-trees
- Can order the entries of a node in increasing distance from q to maximize potential for a good NN found fast
- Can be easily adapted for k-NN search (how?)
- Requires at most one tree path to be currently in memory – good for small memories / caching
  - Characteristic of all depth-first search algorithms
  - Recall that the range search algorithm is also DF
- However, does not visit the least possible number of nodes
- Also, not incremental – more on this later…

Best-first NN search

- A more efficient algorithm
  - Performs fewer comparisons
  - Visits the smallest possible number of R-tree nodes, for a given query q
- Uses a priority queue to organize seen entries and prioritize the next node to be visited
- Can be used for k-NN search and incremental NN search

Best-first NN search

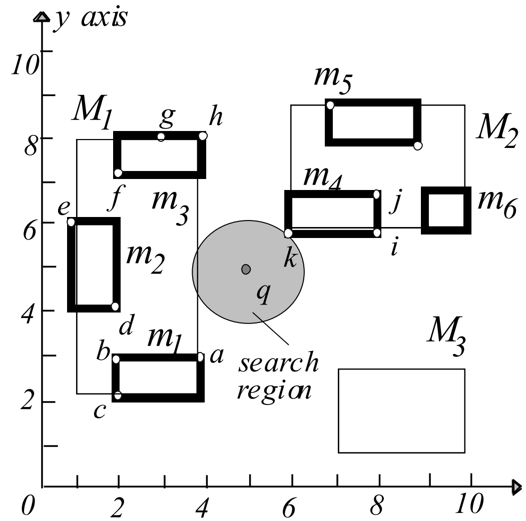
- Observation about DF-search:
  - The closest entry to q in the current node is “opened” and control is passed to the node pointed by it
  - However the entries in that node may not be the closest entries to q from those seen so far
- Idea of BF search:
  - Put all entries in a priority queue and always “open” the closest one, independently of the node that contains it
  - Thus the best (i.e., closest) entry is always visited first

Best-first NN search using an R-tree

- For data points only

```
BFNN(query point q, R-tree R)
  add all entries of R.root into a min-heap Q      Heap's key: dist(q,e.MBR)
  while Q is not empty
     e = Q.top; remove e from Q
     if e is a leaf entry // point
          return e // first removed data point is guaranteed to be the NN
     n = node of R pointed by e
     for each entry e in n do
         Q.enheap(e)
```

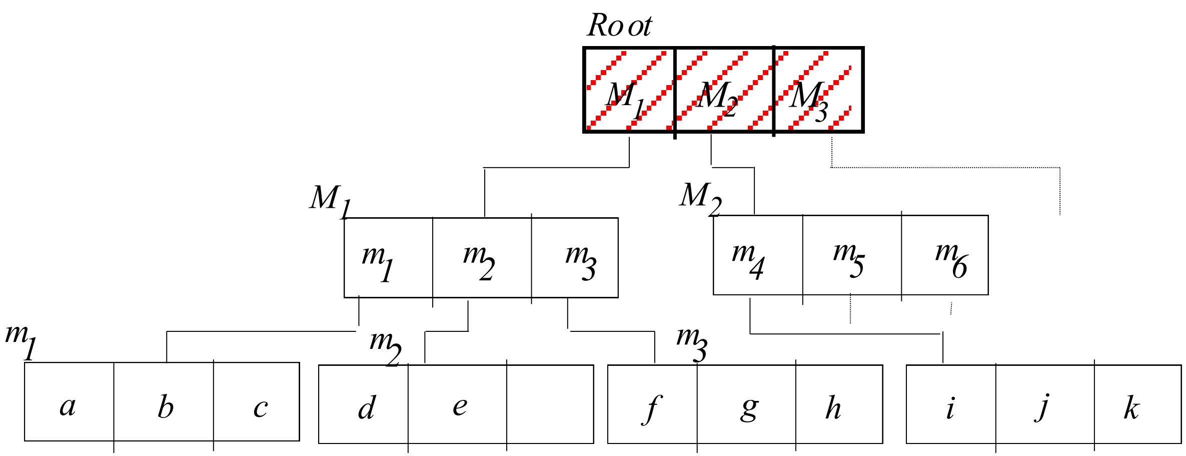
Best-first NN search



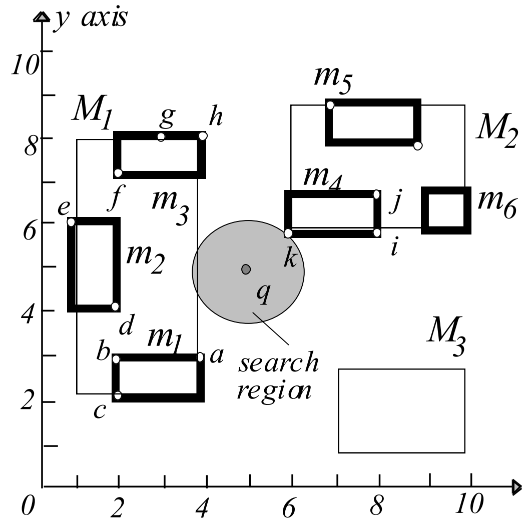
Step 1: put all entries of root on heap Q

$Q = M_1(1), M_2(\sqrt{2}), M_3(\sqrt{8})$

distance from q

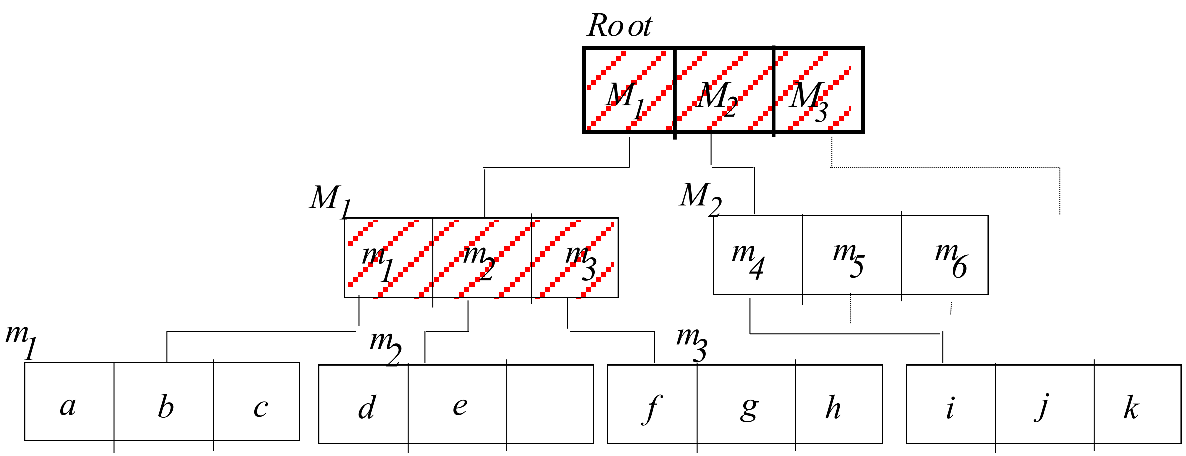


Best-first NN search

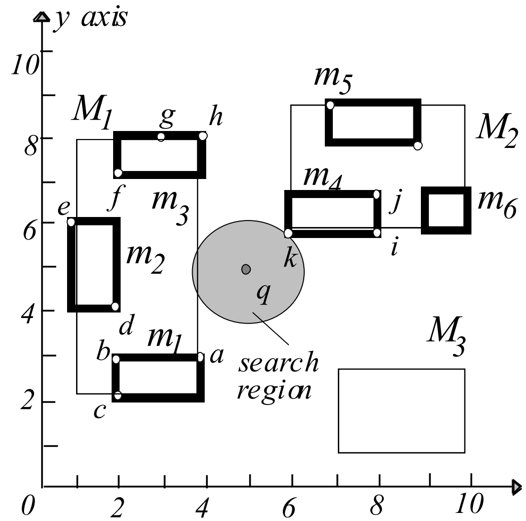


Step 2: get closest entry (top element of Q): $M_1(1)$. Visit node M_1. Put all entries of visited node on heap Q

$Q = M_2(\sqrt{2}), m_1(\sqrt{5}), m_3(\sqrt{5}), M_3(\sqrt{8}), m_2(3)$



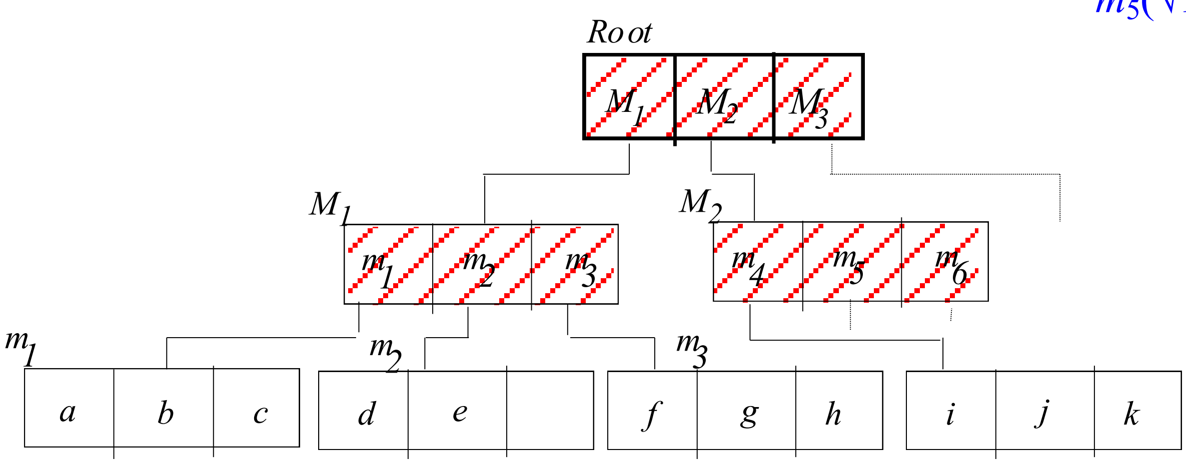
Best-first NN search



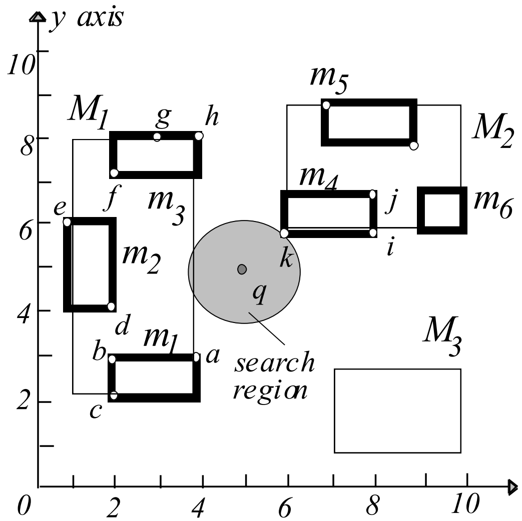
Step 3: get closest entry (top element of Q):

$M_2(\surd 2)$. Visit node M_2. Put all entries of visited node on heap Q

$Q = m_4(\surd 2), m_1(\surd 5), m_3(\surd 5), M_3(\surd 8), m_2(3), m_5(\surd 13), m_6(\surd 17)$

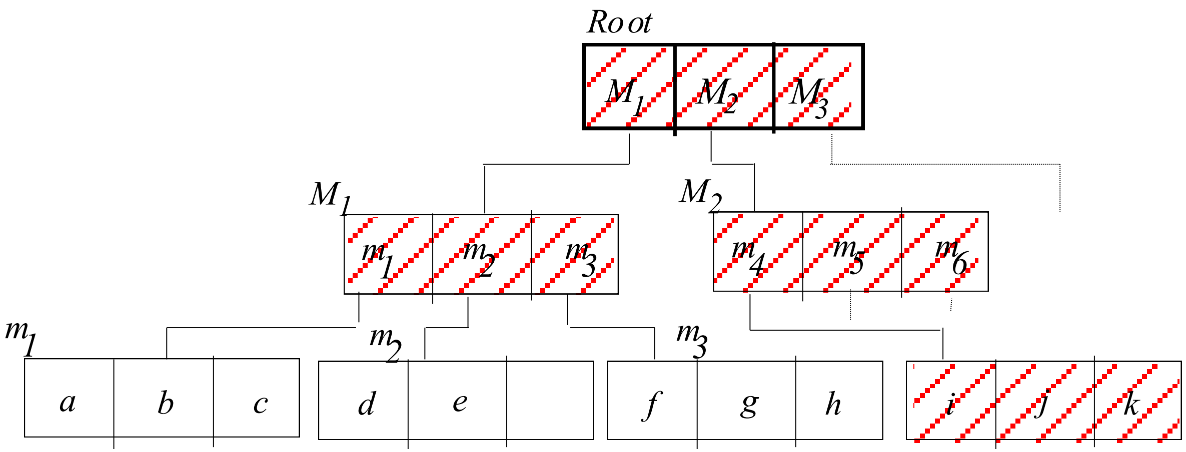


Best-first NN search

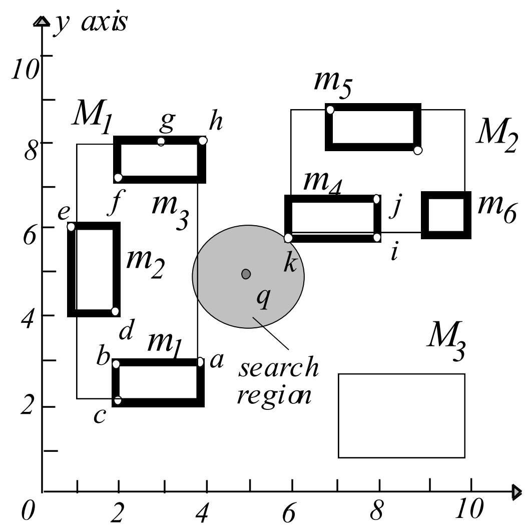


Step 4: get closest entry (top element of Q): $m_4(\sqrt{2})$. Visit node m_4. Put all entries of visited node on heap Q

$Q = k(\sqrt{2}), m_1(\sqrt{5}), m_3(\sqrt{5}), M_3(\sqrt{8}), m_2(3), i(\sqrt{10}), j(\sqrt{13}), m_5(\sqrt{13}), m_6(\sqrt{17})$



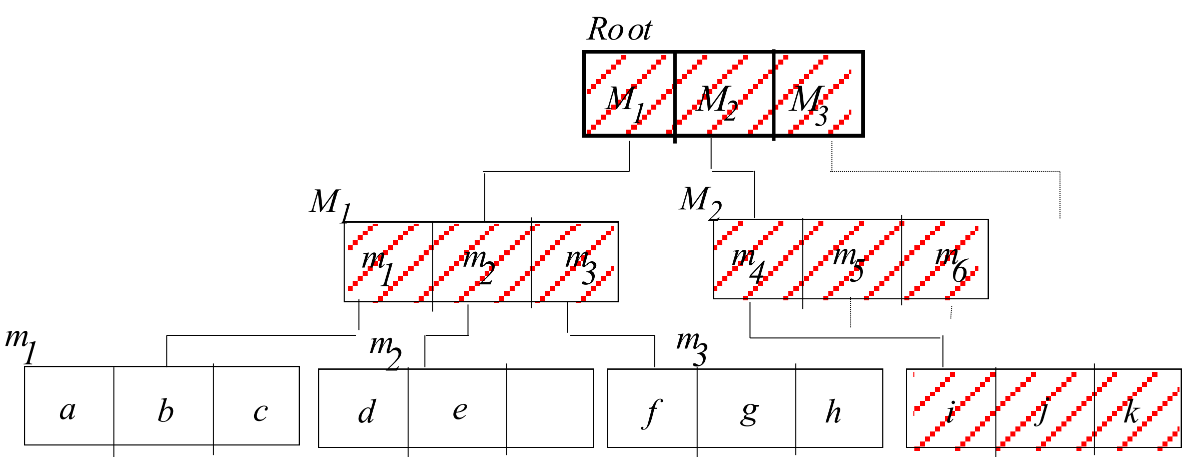
Best-first NN search



Step 5: get closest entry (top element of Q):

$k(\sqrt{2})$. Since k is a data point, search stops and k is returned as the NN of q

$Q = m_1(\sqrt{5}), m_3(\sqrt{5}), M_3(\sqrt{8}), m_2(3),$
$i(\sqrt{10}), j(\sqrt{13}), m_5(\sqrt{13}), m_6(\sqrt{17})$



Notes on Best-first NN search

- In the previous example, we have visited fewer nodes compared to DF-NN algorithm
  - Only nodes whose MBR intersects the disk centered at q with radius the real NN distance are visited (see if you can you prove this)
- The algorithm can be used for incremental NN search
  - After having found the NN, we can continue, until the next data point is de-heaped (the next NN) and so on without having to start the algorithm from the beginning
- The algorithm can be used for k-NN search
  - Continue the algorithm until k data points are de-heaped
- Drawback of best-first NN algorithm:
  - The heap can grow very large
  - In the worst case, all R-tree entries are en-heaped before the NN is found

Why incremental NN search?

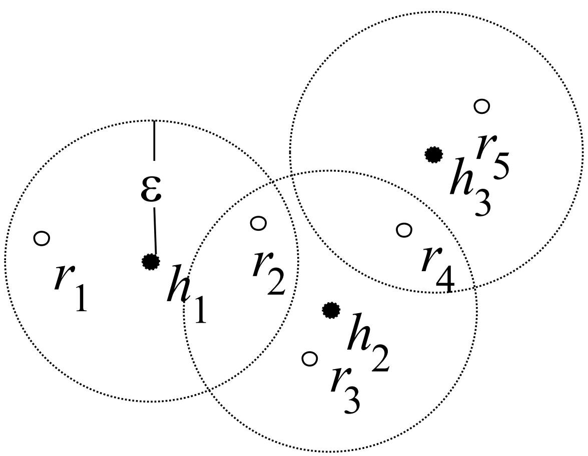
- Example 1: find the nearest large city (>10,000 residents) to my current position
  - Solution 1:
    - find all large cities
    - apply NN search on the result
    - could be slow if many such cities
    - also R-tree may not be available for large cities only
  - Solution 2:
    - incrementally find NN and check if the large city requirement is satisfied; if not get the next NN
- Example 2: find the nearest hotel; see if you like it; if not get the next one; see if you like it; …
- Also: similarity search in multimedia databases

Spatial Intersection Joins

- Input:
  - two spatial relations R, S (e.g., R=cities, S=rivers)
  - a spatial relationship θ (e.g., θ=intersects)
- Output:
  - $\{(r,s): r \in R,\ s \in S,\ r\ \theta\ s \text{ is true}\}$
  - Example: find all pairs of cities and rivers that intersect

Spatial Joins based on distances

- Distance join: Find pairs of hotels, restaurants close to each other (with distance smaller than 100m)
- Closest pairs: Find the closest pair of hotels, restaurants
- All-NN join: For each hotel find the nearest restaurant
- Iceberg distance join: Find hotels close to at least 10 restaurants



Spatial Join Algorithms

- Take advantage of existing indexes as much as possible
  - R-tree join
- Inspired from relational join algorithms
  - Spatial hash join
- Details are out of scope

Summary

- Spatial Data are ubiquitous
- Two main types of spatial data
  - Points
  - Extended objects
- Queries based on spatial relationships
  - topological, distance, directional
- Main query types
  - range selection, nearest neighbor search, spatial joins
- Indexes for points and/or extended objects
  - R-tree is the dominant index
- Spatial query algorithms for range and NN queries
  - Also applicable for multidimensional points