南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

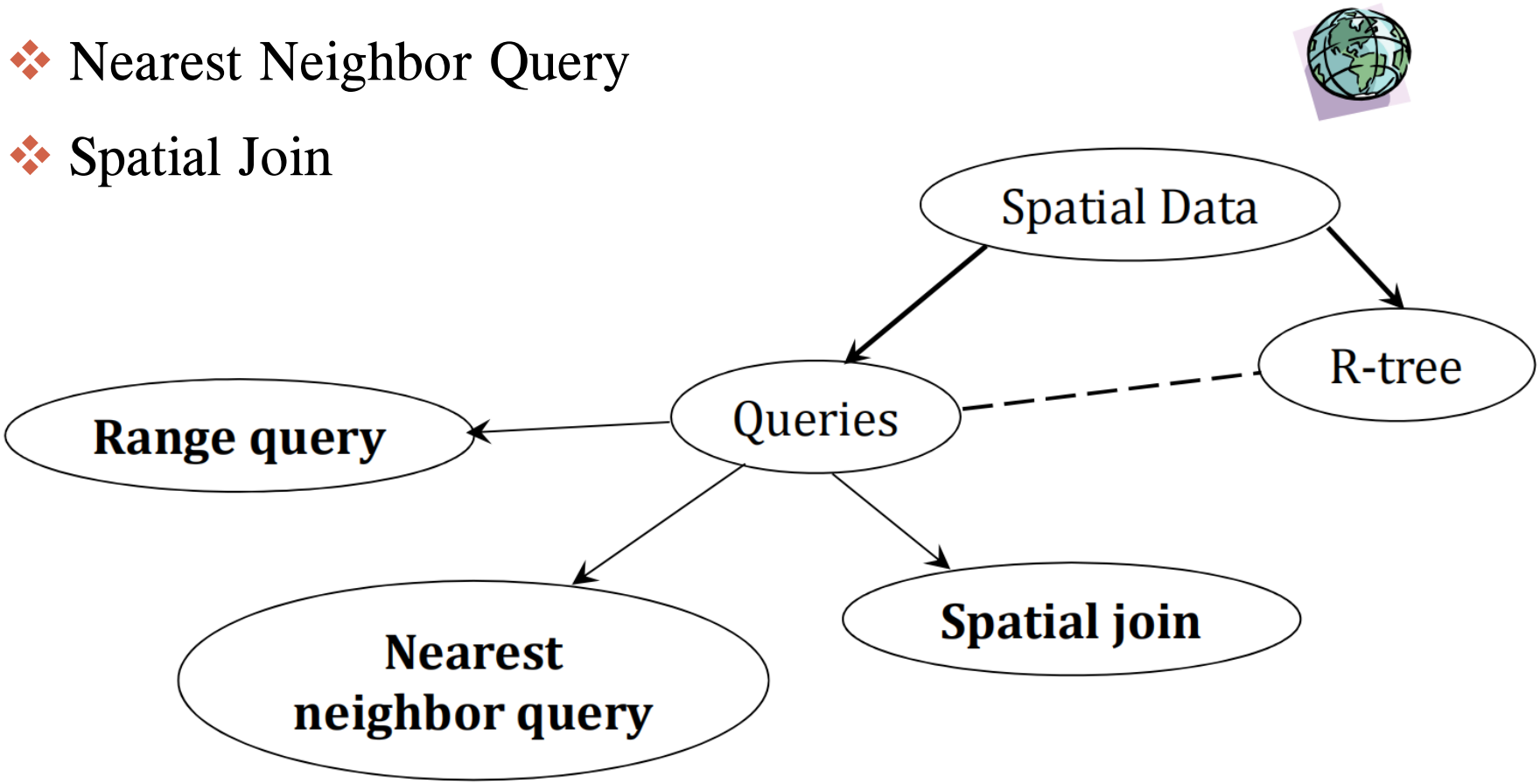
DBGroup

Spatial Databases II

南方科技大学
唐　博
tangb3@sustech.edu.cn

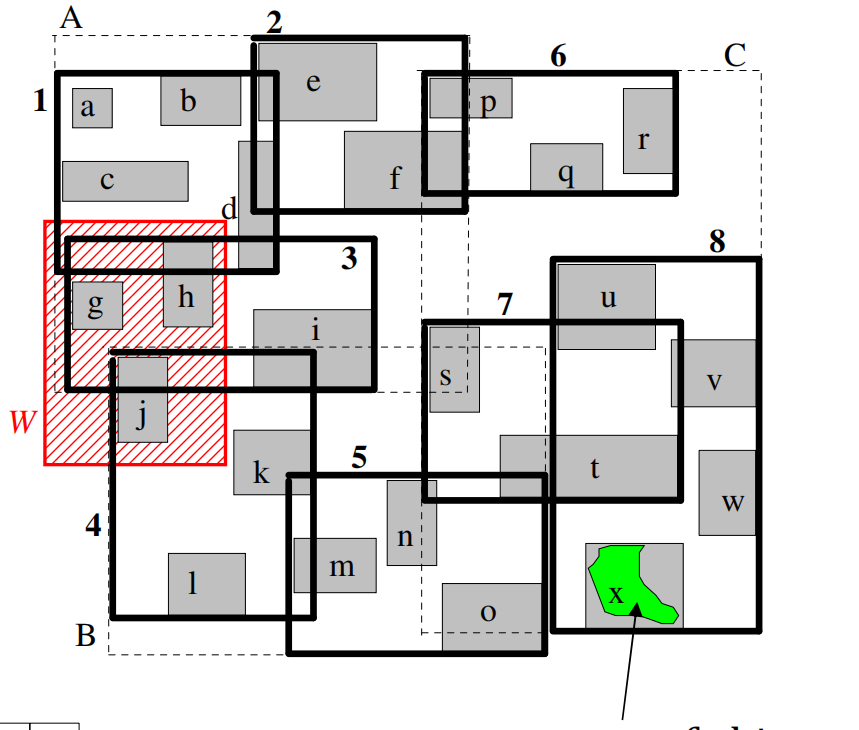
Spatial Query Processing

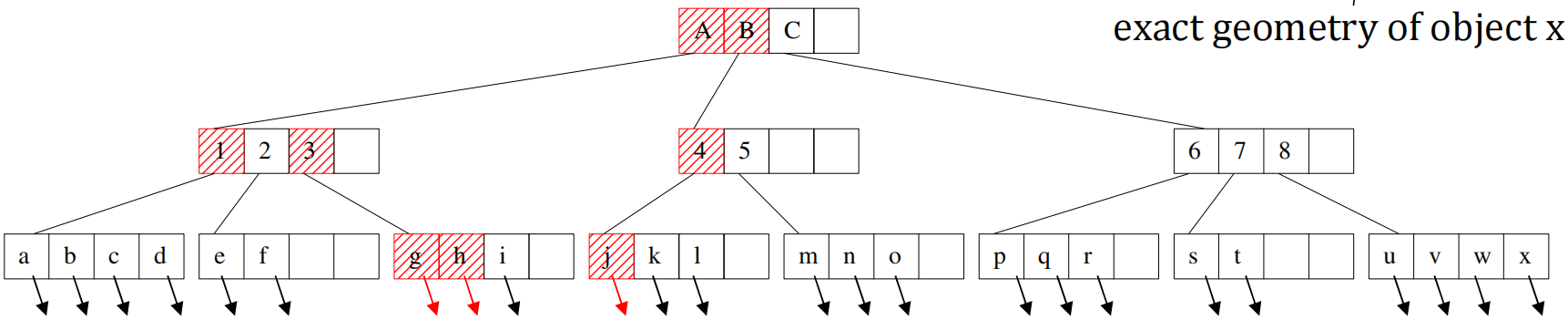
- Range Search (review)
- Nearest Neighbor Query
- Spatial Join



Range Search on R-tree

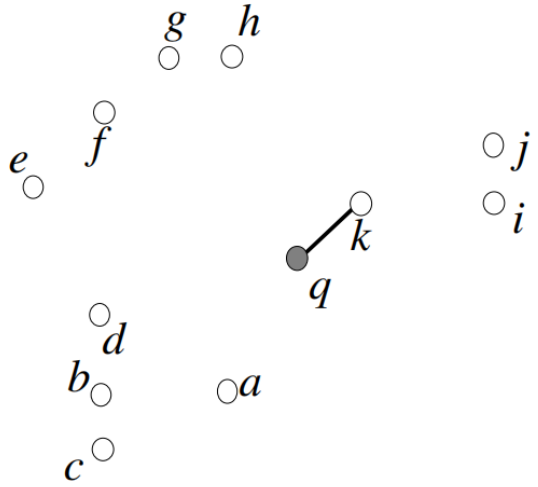
- W is a window query (range intersection query)





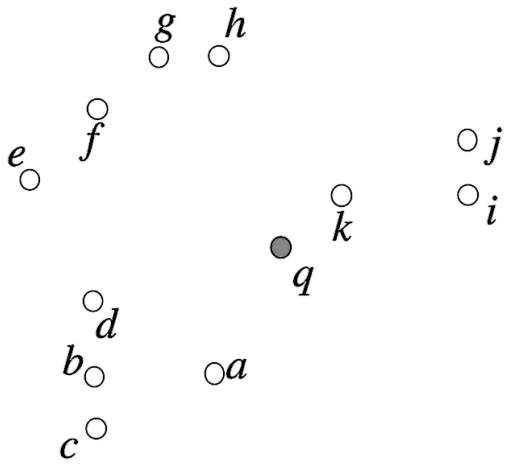
Nearest neighbor query

- Nearest neighbor query:
  - Given a spatial relation R and a query object q, find the nearest neighbor (NN) of q in R
  - Formal definition:
    - $\mathrm{NN}(q, R) = o \in R: \mathrm{dist}(q, o) \leq \mathrm{dist}(q, o'), \forall o' \in R$
- Simplification
  - We usually focus on point-sets
  - NN search return any NN, if there are ties (i.e., when multiple points have equal minimum distance)



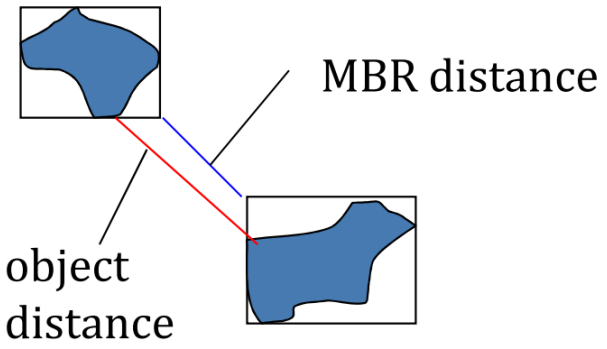
Linear scan method

- Linear scan method
  - Compare q with each object in R
  - Keep track of the object with minimum distance to q
  - Incurs high cost when the size of R is very large
- What is our goal?
  - Use R-tree to reduce the I/O cost of NN search



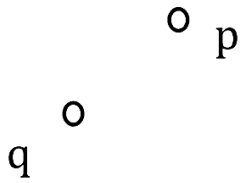
Distance measures and MBRs

- $dist(o_i, o_j)$
  - The minimum distance between the geometric extents of objects
- $dist(o_i.\mathrm{MBR}, o_j.\mathrm{MBR})$
  - The minimum distance between their MBRs
- **Property**: the distance between two MBRs lower-bounds the distances between the corresponding objects
  - $dist(o_i.\mathrm{MBR}, o_j.\mathrm{MBR}) \leq \mathrm{dist}(o_i, o_j)$



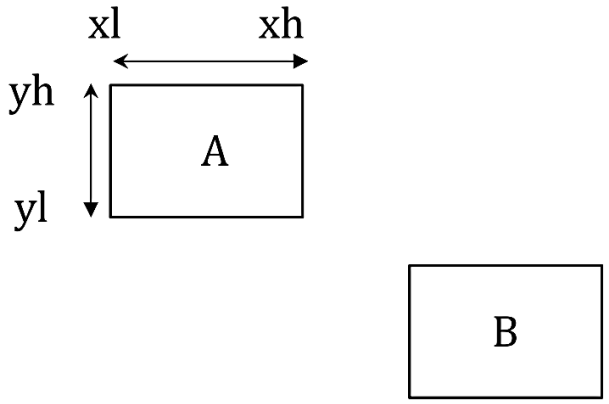
How to compute distances?

Distance between two points



$$dist(q,p) = \sqrt{(q.x - p.x)^2 + (q.y - p.y)^2}$$

Distance between two rectangles



$$dist_X(A,B) = \begin{cases} B.xl - A.xh & \text{if } B.xl > A.xh \\ A.xl - B.xh & \text{if } A.xl > B.xh \\ 0 & \text{otherwise} \end{cases}$$

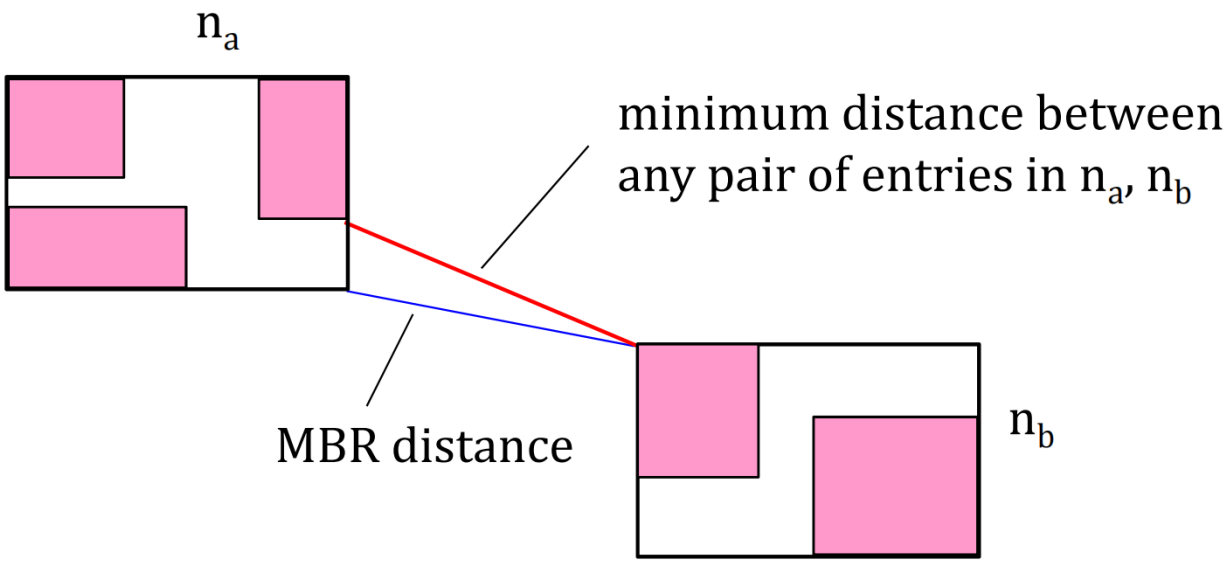
Similar definition is used for $dist_Y(A,B)$

$$dist(A,B) = \sqrt{dist^2_X(A,B) + dist^2_Y(A,B)}$$

Distance measures and MBRs

- Distances between R-tree node MBRs lower-bound the distances between the entries in them
  - $dist(n_a.\text{MBR}, n_b.\text{MBR}) \leq dist(e_i.\text{MBR}, e_j.\text{MBR})$

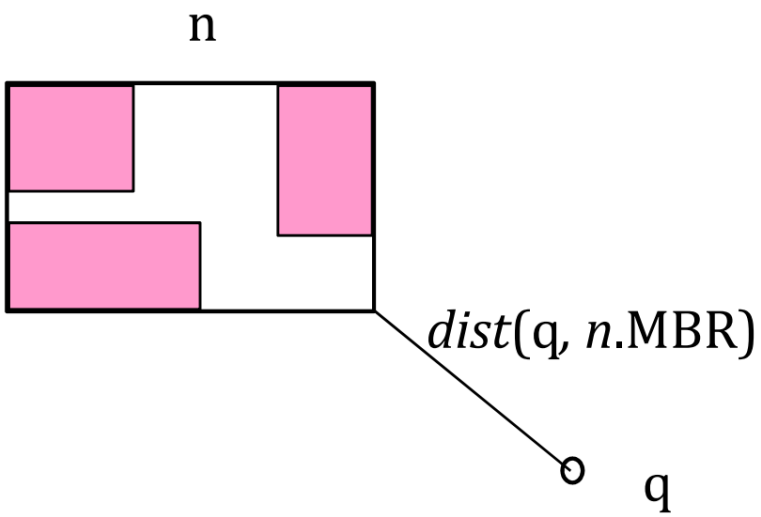
    for each entry $e_i \in n_a$, each entry $e_j \in n_b$



Distance measures and MBRs

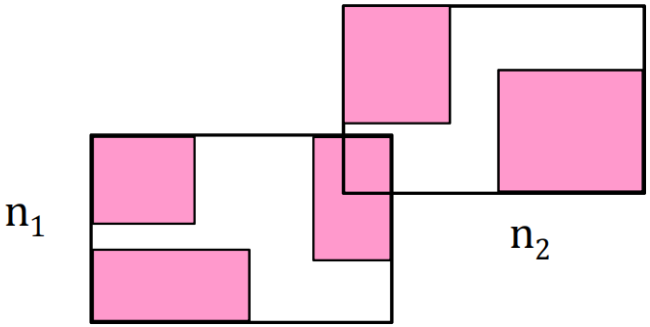
- The distance between a query object q and an R-tree node MBR lower-bounds the distances between q an the objects indexed under this node
  - $dist(\mathrm{q}, \mathrm{n.MBR}) \leq dist(\mathrm{q}, \mathrm{o})$

for each object o indexed under n



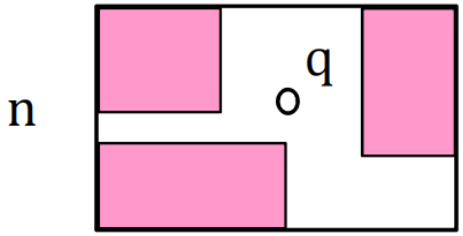
Questions

- Find the distance between n_1.MBR and n_2.MBR



$dist(n_1.\text{MBR}, n_2.\text{MBR}) = ?$

- Find the distance between q and n.MBR



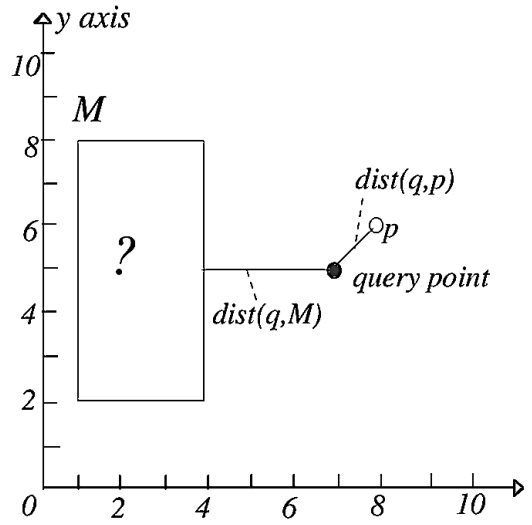
$dist(\text{q}, n.\text{MBR}) = ?$

- What can we conclude from these distances?

Discard "far-away" MBR

- Problem: find the NN of q in an R-tree
- Scenario:
  - Suppose that we have already seen a point **p**
  - Is it possible to find a point p' in the subtree of M such that p' is closer to q?
  - Prune the child node of M when
    - dist(q,p) ◦ dist(q,M)

Should we put ≤ or ≥ here?

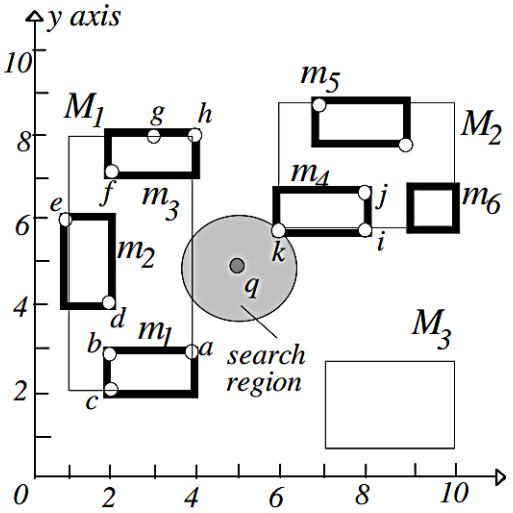


Depth-first NN search on R-tree

- Maintain an object o_{NN} as the nearest object found
  - Initialize it to a null object with $dist\ (q,o_{NN}) = \infty$

- Idea: modify the recursive range search algorithm to NN search algorithm
  - Start from the root and visit the node nearest to q
  - Continue recursively, until a leaf node n_L is visited
  - Find the NN of q in n_L
  - Continue visiting other nodes after backtracking, while there are nodes closer to q than the current NN

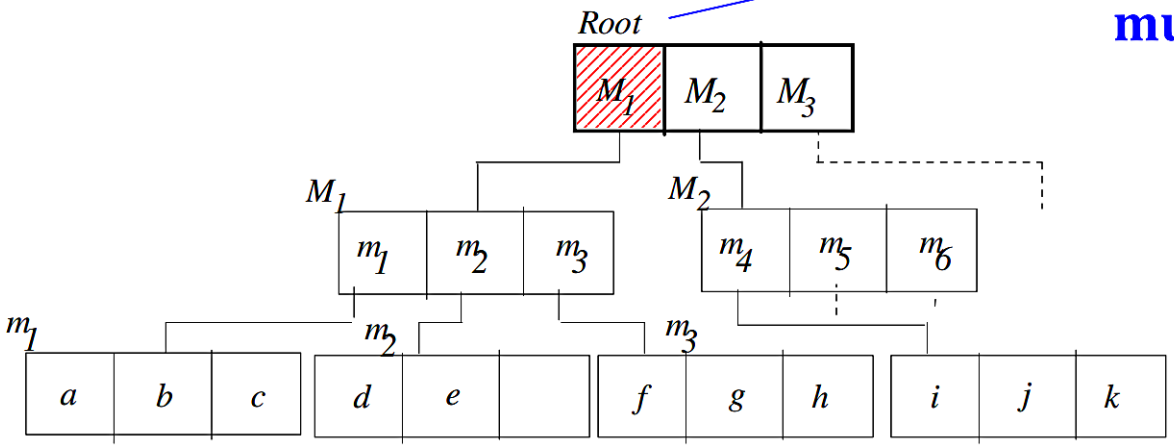
Depth-first NN search on R-tree



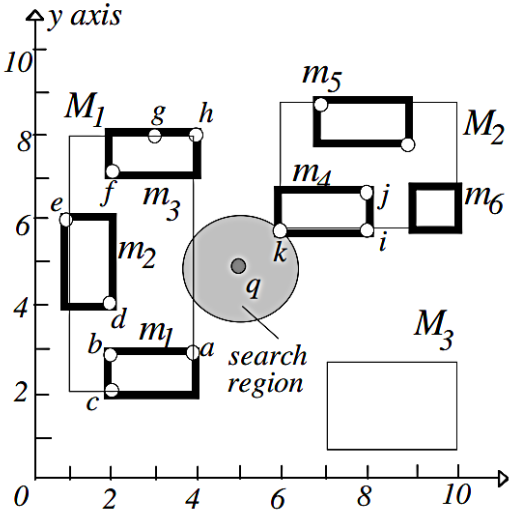
o_{NN} = NULL
$dist(q,o_{NN})= \infty$

1. visit root

$dist(q,M_1)<dist(q,o_{NN})$
must visit node M_1



Depth-first NN search on R-tree



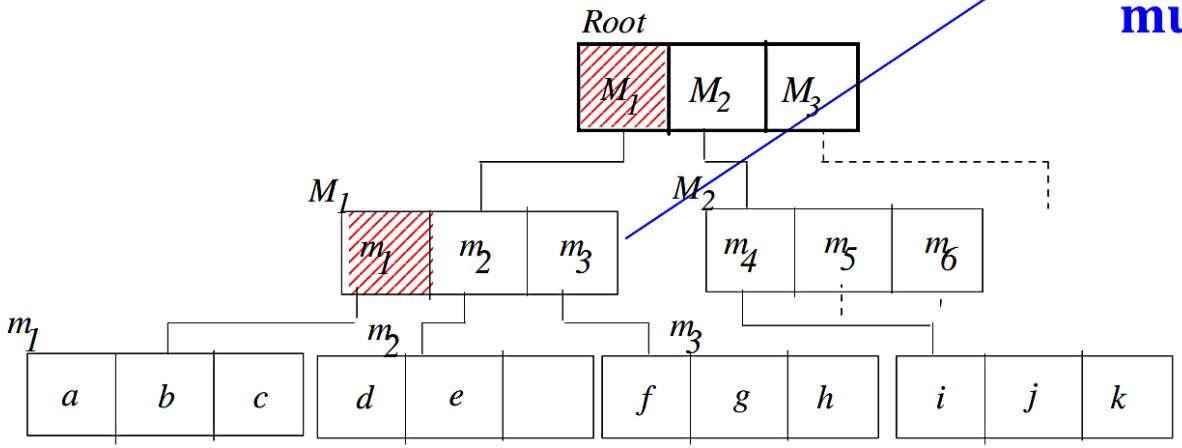
o_{NN} = NULL

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

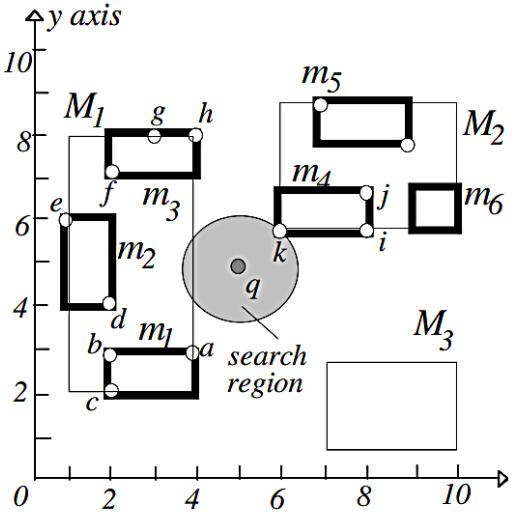
2. visit M_1

$dist(q,m_1)<dist(q,o_{NN})$

must visit node m_1



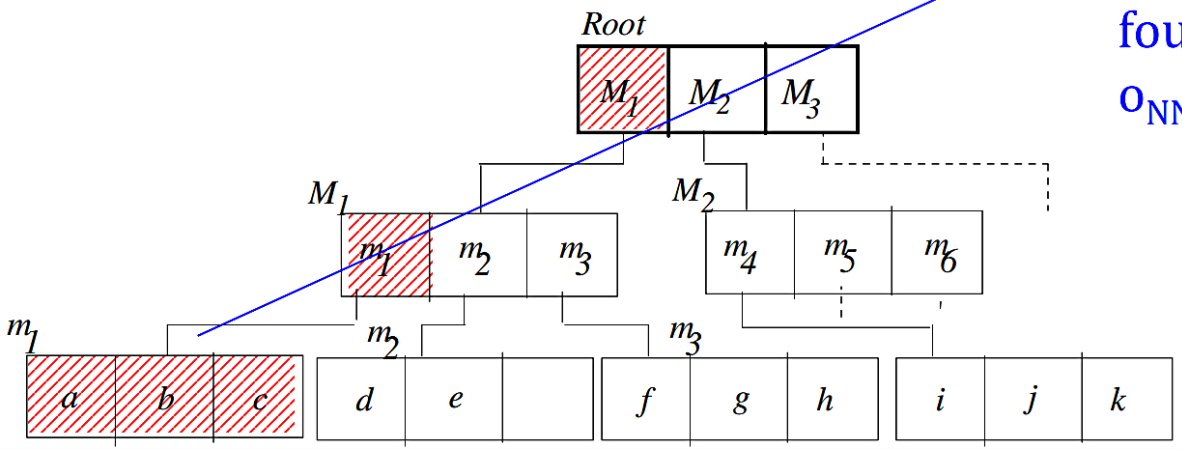
Depth-first NN search on R-tree



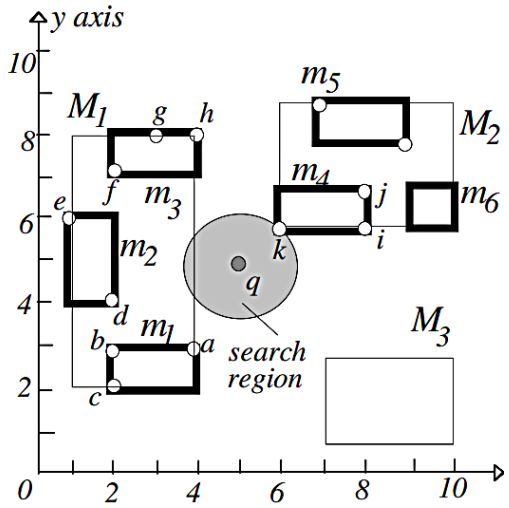
o_{NN} = NULL
$dist(q,o_{NN})= \infty$

3. visit m_1

check a,b,c
found new NN:
o_{NN} = a, $dist(q,o_{NN})$ = $\sqrt{5}$



Depth-first NN search on R-tree

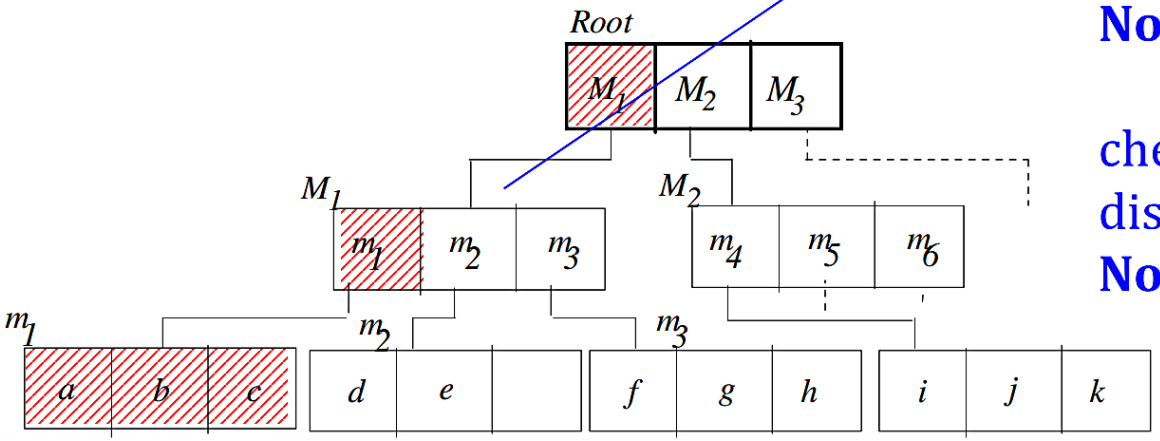


o_{NN} = a
$dist(q,o_{NN}) = \sqrt{5}$

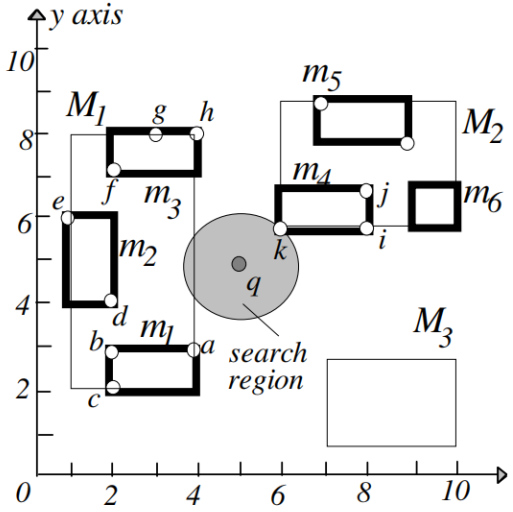
4. backtrack to M_1

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



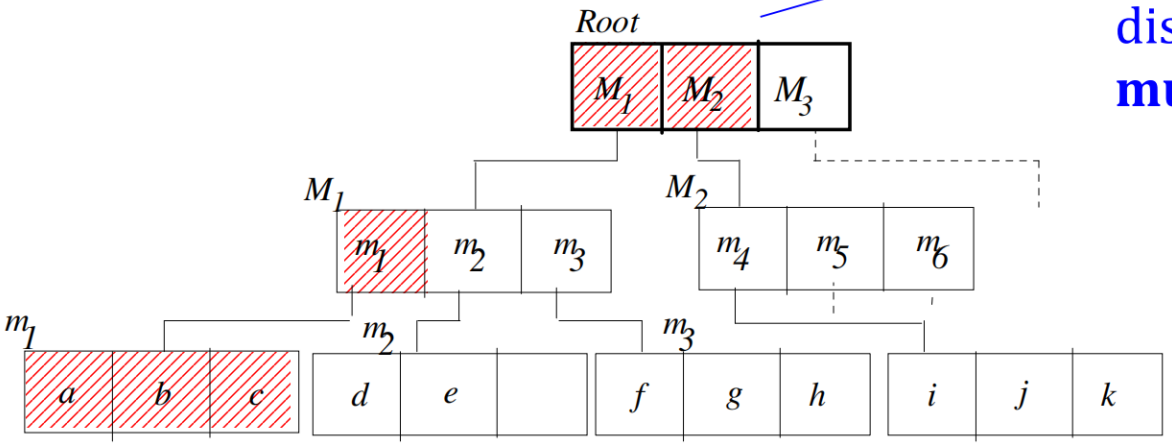
Depth-first NN search on R-tree



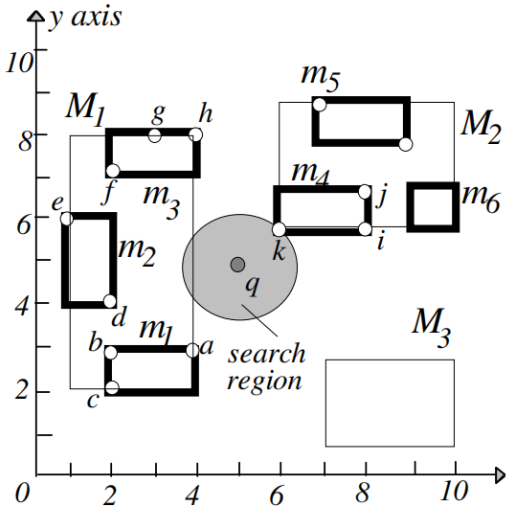
o_{NN} = a
$dist(q,o_{NN}) = \sqrt{5}$

5. backtrack to root

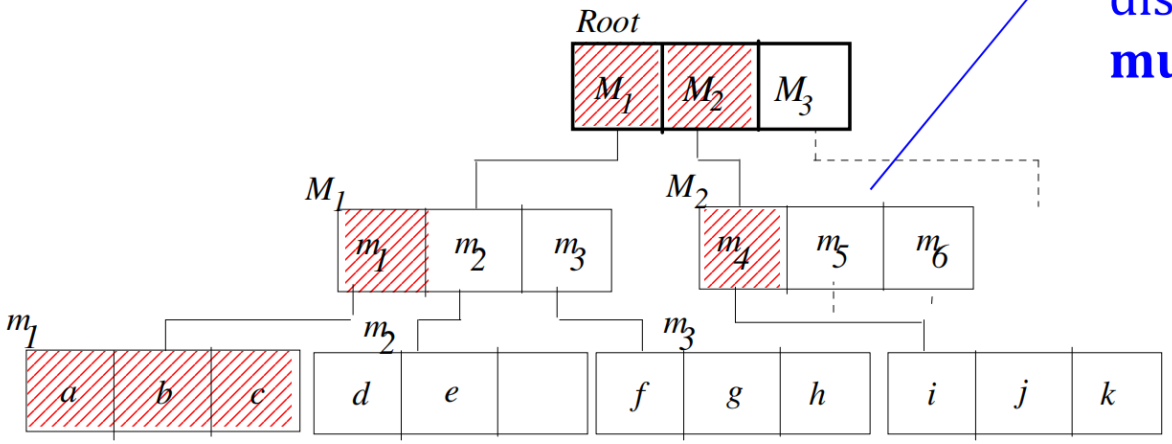
check M_2
$dist(q,M_2) = \sqrt{2} < \sqrt{5}$:
must visit node M_2



Depth-first NN search on R-tree



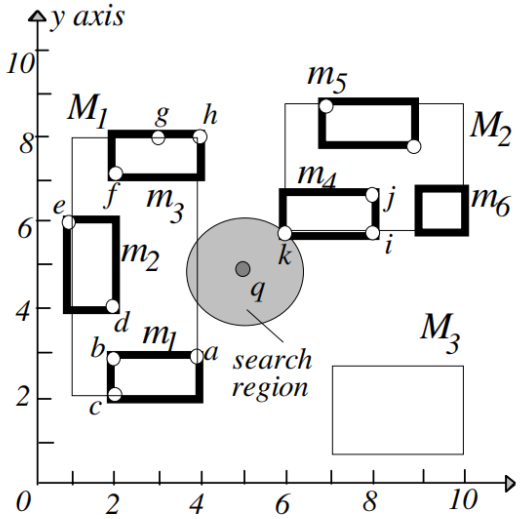
o_{NN} = a
$dist(q,o_{NN}) = \sqrt{5}$



6. visit M_2

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

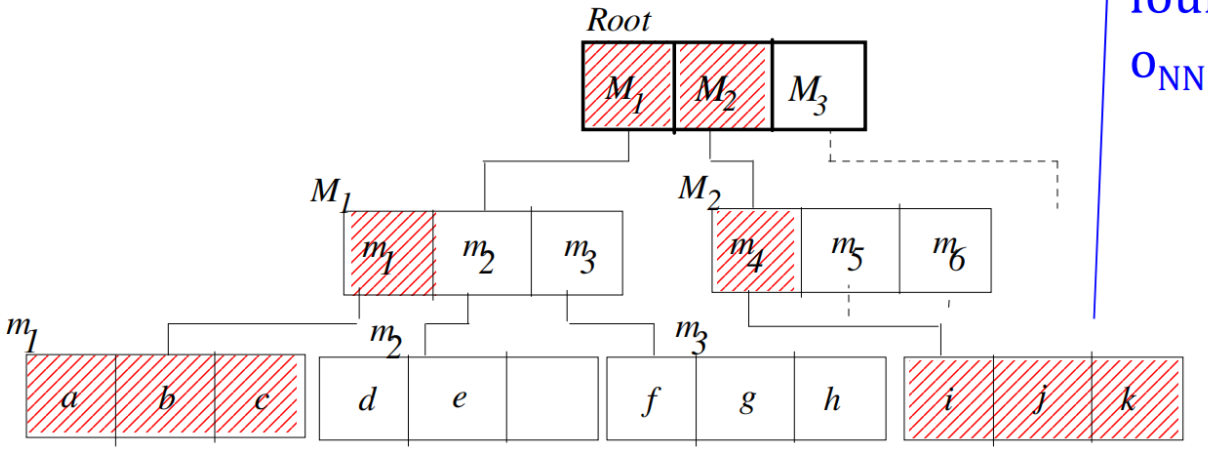
Depth-first NN search on R-tree



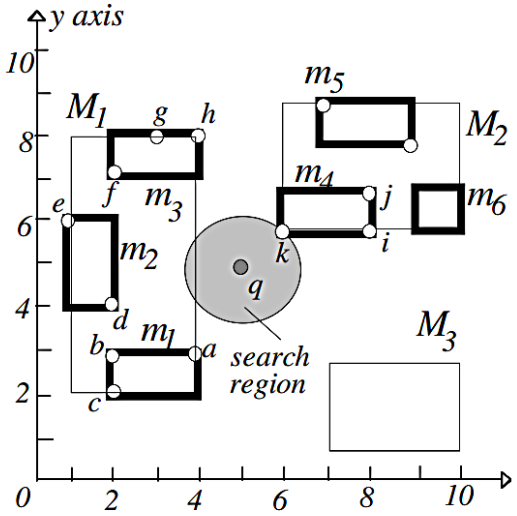
o_{NN} = a
$dist(q,o_{NN}) = \sqrt{5}$

7. visit m_4

check i,j,k
found new NN:
o_{NN} = k, $dist(q,o_{NN}) = \sqrt{2}$



Depth-first NN search on R-tree

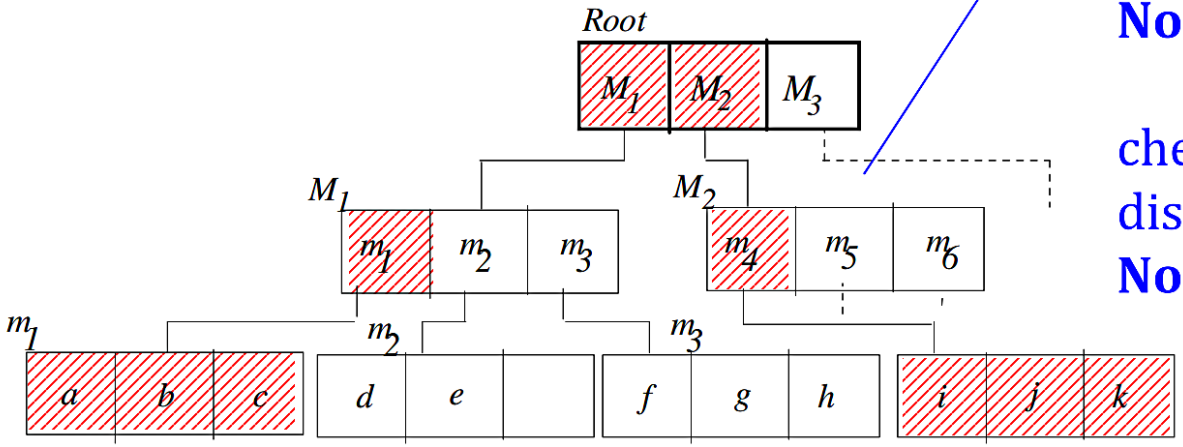


$o_{NN} = k$
$dist(q,o_{NN}) = \sqrt{2}$

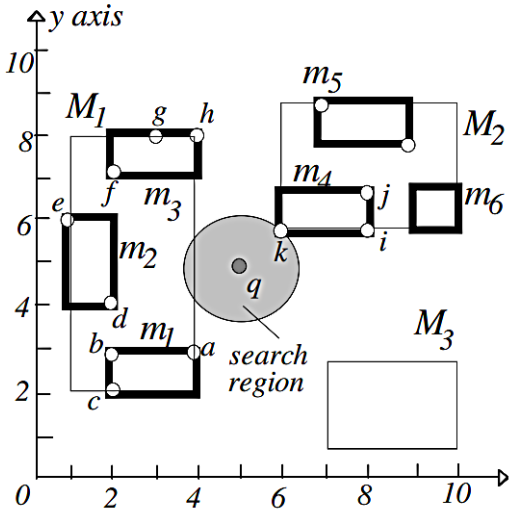
8. backtrack to M_2

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

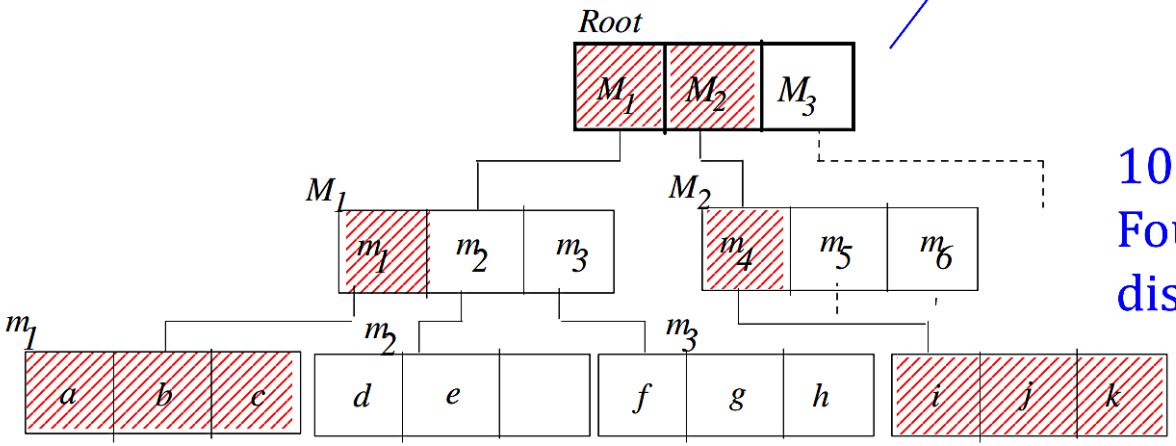


Depth-first NN search on R-tree



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



10. Algorithm terminates.
Found o_{NN} =k with
$dist(q,o_{NN}) = \sqrt{2}$

Depth-first NN search algorithm

Initially a null object with $dist(q, o_{NN}) = \infty$

```
function DF_NN_search(object q, Node n, object o_NN)
    if n is a leaf node then
        for each entry e ∈ n
            if dist(q, e.MBR) < dist(q, o_NN) then
                /* e.MBR is closer to q than its current NN*/
                o := object with address e.ptr;
                if dist(o, q) < dist(q, o_NN) then
                    /* found new NN */
                    o_NN := o;
    else /* n is not a leaf node */
        for each entry e ∈ n
            if dist(q, e.MBR) < dist(q, o_NN) then
                /* it is possible that n indexes the NN */
                n' := R–tree node with address e.ptr;
                DF_NN_search(q, n', o_NN);
```

Notes on depth-first NN search

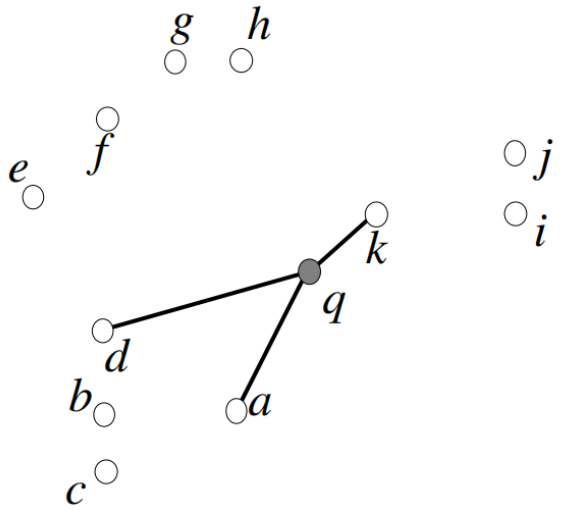
- An optimization: order entries of a node in increasing distance from q, so that a point close to q can be found fast
  - Sort entries in ascending order of dist(q, e.MBR)
  - Process the entries in the sorted order
    → tend to find the NN sooner

- Memory requirement: small, at most one tree path in memory
  - Feature of all depth-first algorithms (e.g., the range search algorithm)

- Node access cost: may not visit the least possible number of nodes

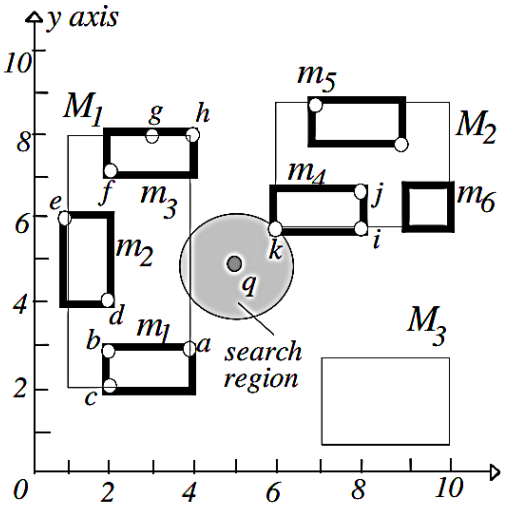
k-Nearest neighbor search

3NN(q,R)



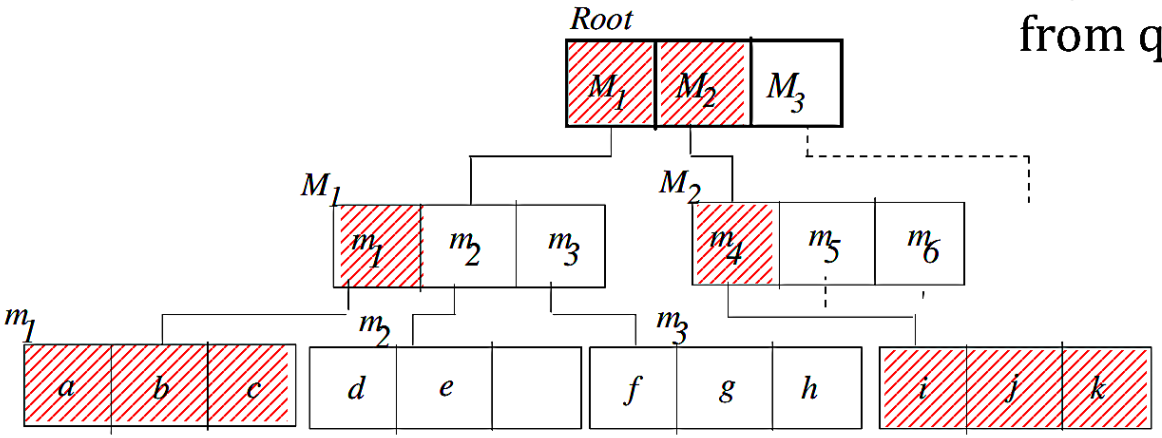
- Generalized problem: k-NN search
  - Given a spatial relation R, a query object q, and a number k <|R|, find the k-nearest neighbors of q in R
    - kNN(q, R) = S ⊂ R : |S| = k, dist (q,o) ≤ dist (q,o'), ∀ o ∈ S ∀ o' ∈ R-S
- We can extend the depth-first search method for k-NN search
  - Compare distance with the k-th NN found so far
  - Instead of keeping a single oNN, maintain k nearest objects

Visiting of unnecessary nodes



- It first checks the closest entry to q (and its subtree)
  - E.g., M_1, then m_1
- However entries in that node may not be the closest to q

  (among those entries seen so far)
  - E.g., m_1, m_2, m_3 are farther away from q, when compared to M_2



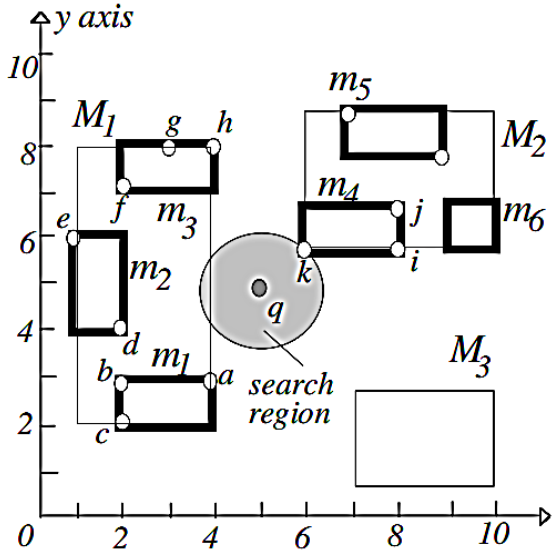
Best-first NN search

- Idea of **best-first** search:
  - Uses a **heap** to organize seen entries and prioritize the next node to be visited
  - Always "visit" the closest one (from the heap)
  - Low cost: optimal in the number of R-tree nodes visited for a given query q

Usage of data structure "heap"

- Heap Q
  - Organize seen entries and prioritize the next node to be visited
  - Entry format: tree entry, and distance from q
  - Q = $M_1(1)$, $M_2(\sqrt{2})$, $M_3(\sqrt{8})$
- Supported operations
  - Top(Q): get the top entry of Q
    - Result: we get the entry $M_1(1)$
  - Pop(Q): remove the top entry of Q
    - Result: Q = $M_2(\sqrt{2})$, $M_3(\sqrt{8})$
  - Push(Q,e): insert an entry into Q
    - Result: Q = $M_2(\sqrt{2})$, $M_4(\sqrt{5})$, $M_3(\sqrt{8})$

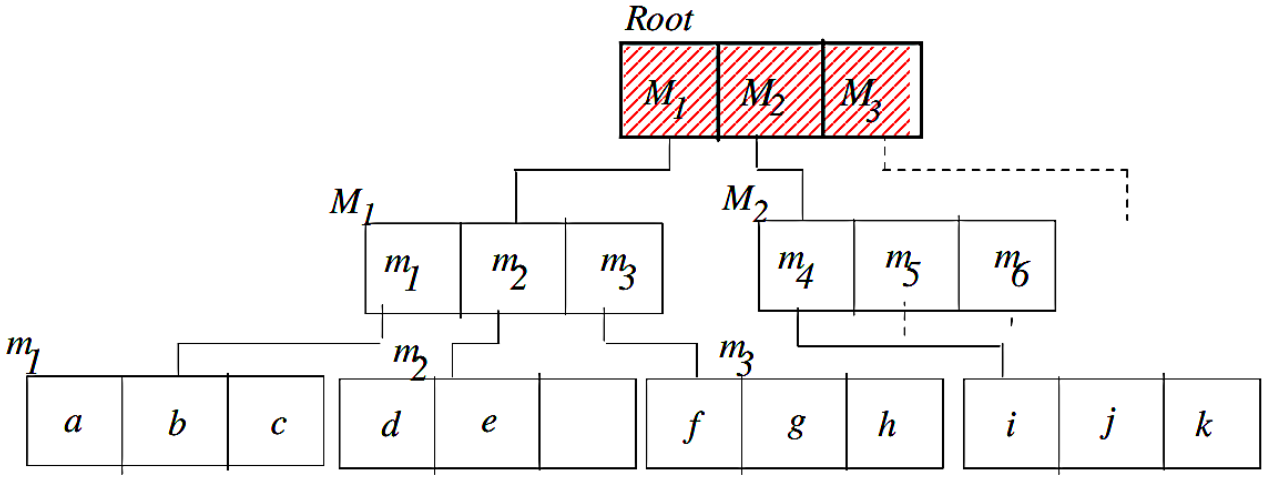
Best-first NN search



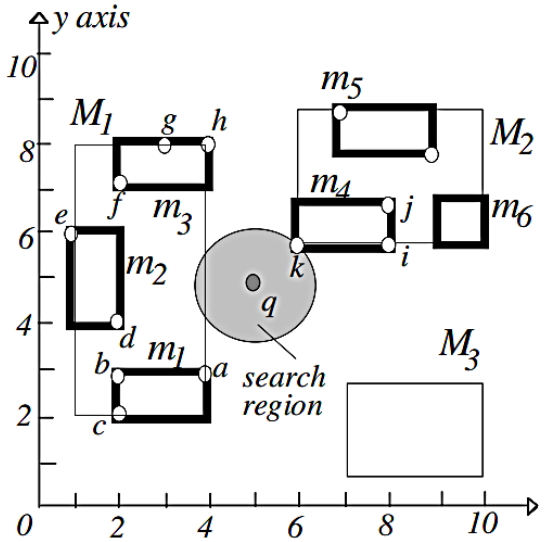
o_{NN} = NULL
$dist(q,o_{NN}) = \infty$

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

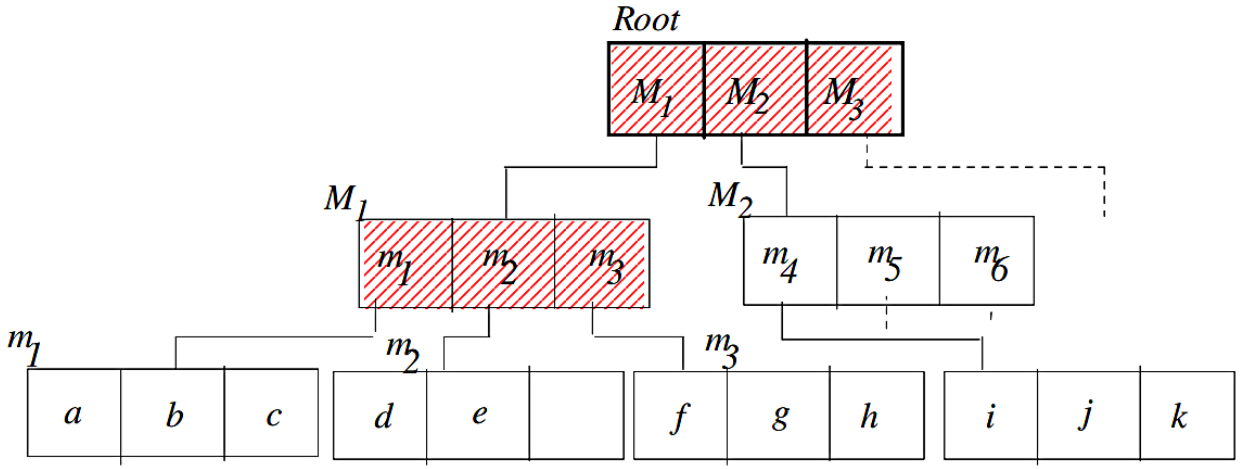


Best-first NN search

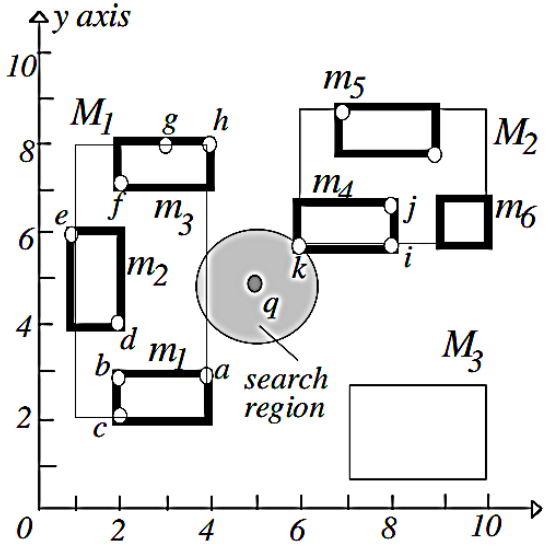


o_{NN} = NULL
$dist(q,o_{NN}) = \infty$

Step 2: get closest entry (top of Q): $M_1(1)$.
Visit node M_1.
Put all entries of visited node on heap Q
$Q = M_2(\sqrt{2}), m_1(\sqrt{5}), m_3(\sqrt{5}), M_3(\sqrt{8}), m_2(3)$

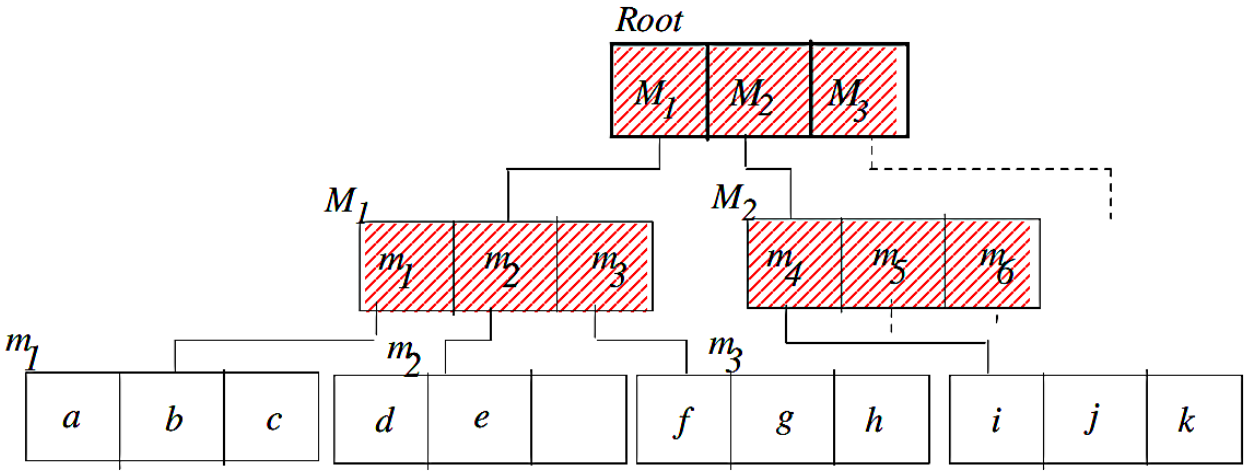


Best-first NN search

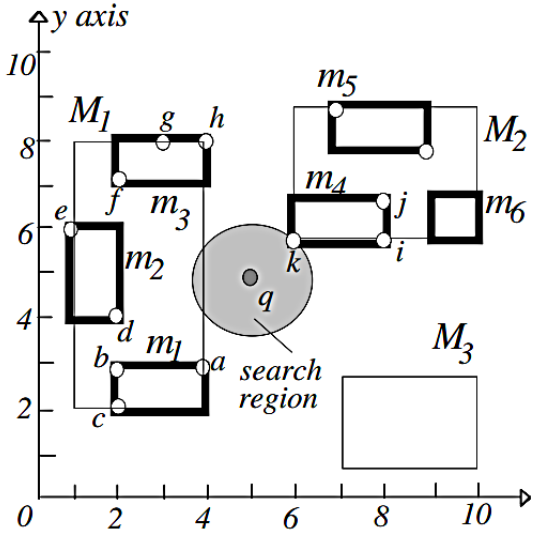


o_{NN} = NULL
$dist(q,o_{NN})= \infty$

Step 3: get closest entry (top of Q): $M_2(\sqrt{2})$.
Visit node M_2.
Put all entries of visited node on heap Q
$Q = m_4(\sqrt{2}), m_1(\sqrt{5}), m_3(\sqrt{5}), M_3(\sqrt{8}), m_2(3),$
$m_5(\sqrt{13}), m_6(\sqrt{17})$

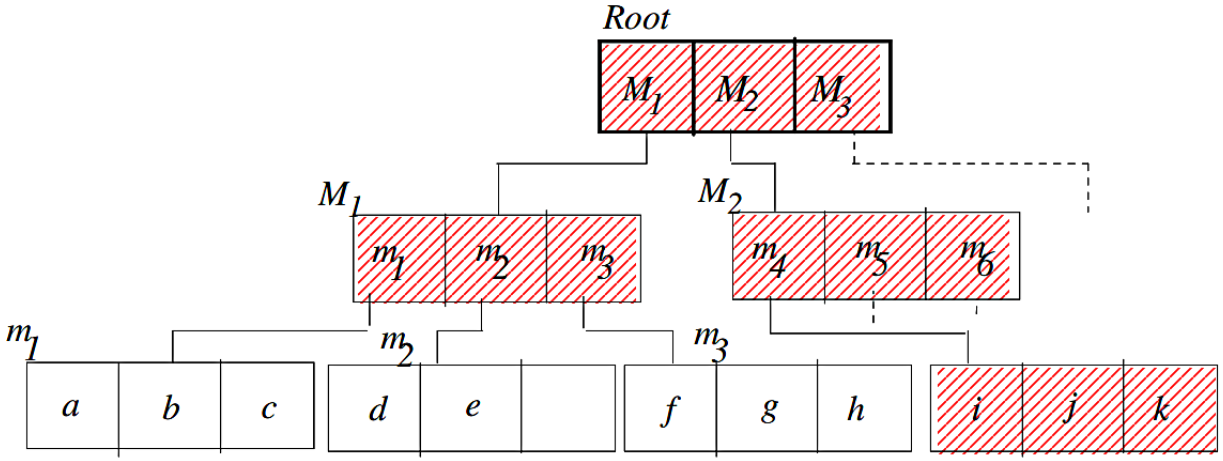


Best-first NN search

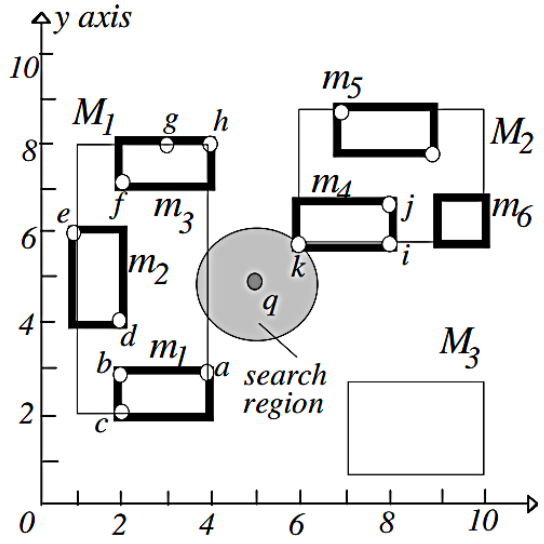


o_{NN} = NULL
$dist(q,o_{NN})= \infty$

Step 4: get closest entry (top of Q): $m_4(\sqrt{2})$.
Visit node m_4.
Since m_4 is a leaf node, we update NN if some object in m_4 is closer than the current NN:
o_{NN} = k, $dist(q,o_{NN})= \sqrt{2}$
$Q = m_1(\sqrt{5}), m_3(\sqrt{5}), M_3(\sqrt{8}), m_2(3),$
$m_5(\sqrt{13}), m_6(\sqrt{17})$

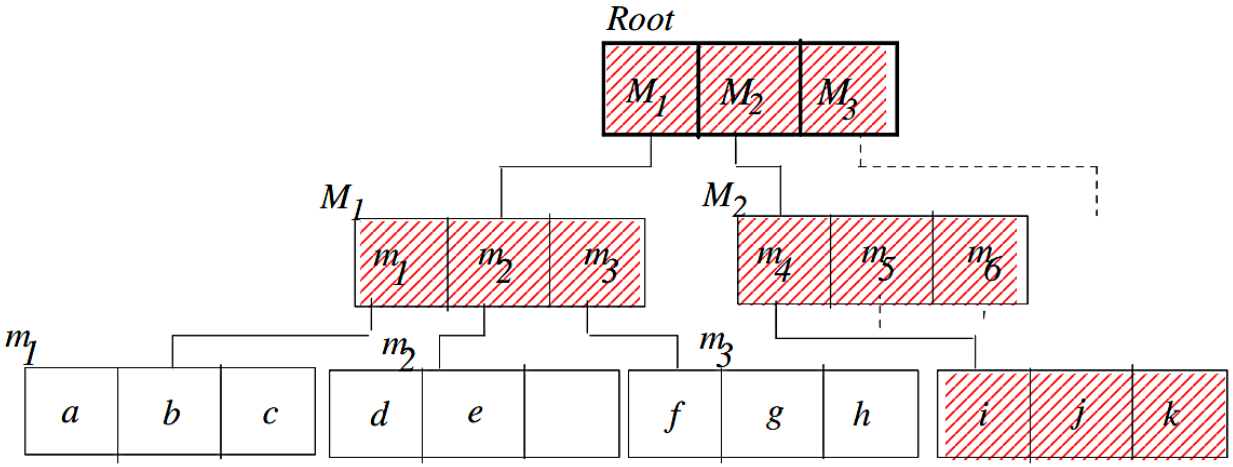


Best-first NN search



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

Step 5: get closest entry (top of Q): $m_1(\sqrt{5})$.
Since $\sqrt{5} \geq dist(q,o_{NN}) = \sqrt{2}$,
search stops and returns o_{NN} as the NN of q



Best-first NN search algorithm

```
function BF_NN_search(object q, R–tree R): object o_NN
    o_NN := NIL; dist(q, o_NN) := ∞
    initialize a priority queue Q;
    add all entries of R's root to Q;
    while not empty(Q) and dist(q, top(Q).MBR) < dist(q, o_NN)
        e := top(Q);
        remove e from Q;
        if e is a directory node entry then
            n := R–tree node with address e.ptr;
            for each entry e' ∈ n
                if dist(q, e'.MBR) < dist(q, o_NN) then
                    add e' on Q;
        else /* e is an entry of a leaf node */
            o := object with address e.ptr;
            if dist(o, q) < dist(q, o_NN) then
                /* found new NN */
                o_NN := o;
    return o_NN;
```

Q is a **min**-heap, the top entry has the smallest dist(q,e)

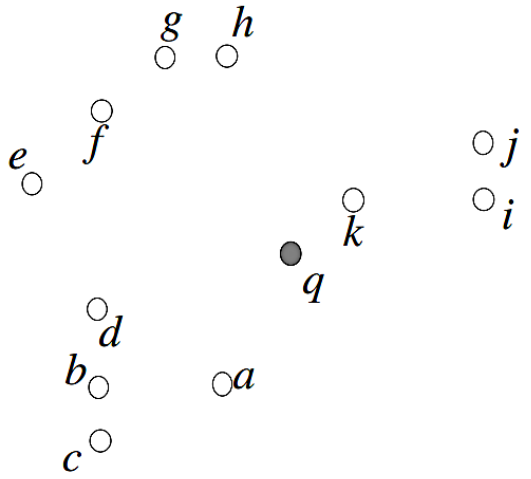
Notes on Best-first NN search

- In the previous example, fewer nodes are visited when compared to DF-NN algorithm
  - Only visit nodes whose MBR intersects the disk centered at q with radius the real NN distance
  - I.e., optimal number of node accesses
- Adaptable for kNN search
  - Maintain the best k objects found so far (like in DF-NN), or
  - Simply use the incremental search (in the next slide)
- However, the worst-case space requirement of the algorithm is high
  - The heap may grow large until the algorithm terminates
  - [Exercise] design a worst-case example to show this

Incremental NN search

- **Incremental search**: after having found the NN, find the next NN without starting search from the beginning
  - Why is it useful? The user may not know a "suitable" k in advance

- Example:
  - Get next NN: k
  - Get next NN: a
  - Get next NN: d
  - ……



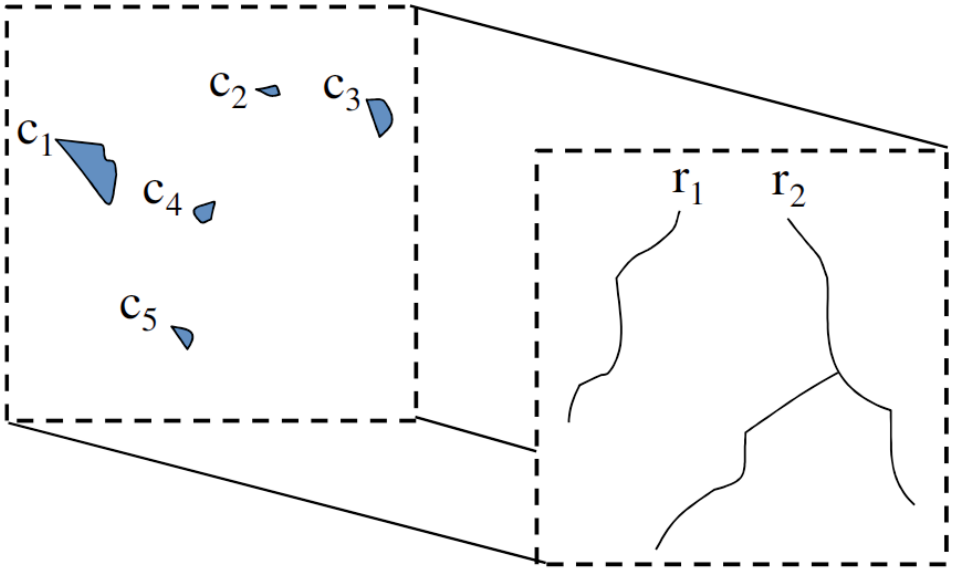
Incremental NN search

- Application 1: find the nearest large city (>10,000 residents) to my current location
  - incrementally find NN and check if the large city
  - requirement is satisfied; if not get the next NN
- Application 2: find the nearest hotel; see if you like it
  - if not get the next one; see if you like it
  - if not get the next one; see if you like it
  - ……
- Best-first NN search can be modified for incremental NN search
  - put objects on the heap
  - never discard objects/entries, wait until an object comes out

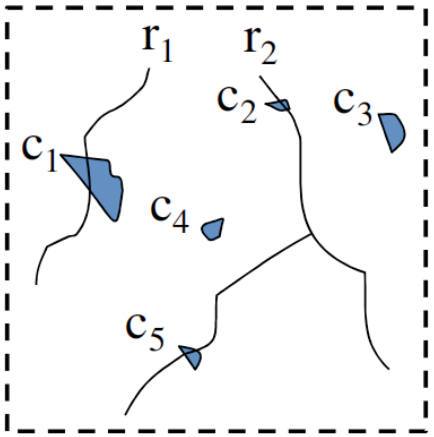
DF search vs. BF search

	Depth-first search	***Best-first search***
Number of node accesses	Number of node accesses not guaranteed to be the minimum	Optimal number of node accesses
Memory requirement	At most a single path of tree nodes	Unbounded memory
Incremental search	Not applicable	Applicable

Spatial Join

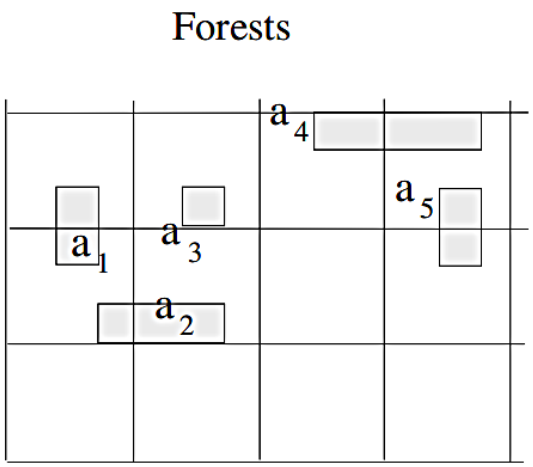


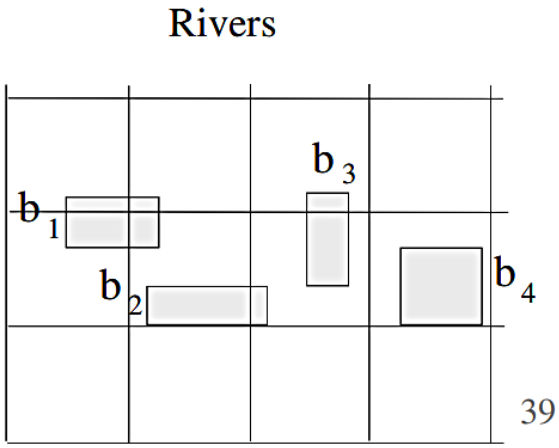
- Input:
  - two spatial relations R, S (e.g., R=cities, S=river
- Output:
  - {(r,s): r∈R, s∈S, r intersects s}
  - Example: find all pairs of cities and rivers that intersect
- We focus on spatial intersection join
  - We skip other types of spatial join here



Spatial Join

- Straightforward solution:
  - Nested loop join between the relations R and S
  - The cost is O(|R||S|); it is not scalable to large datasets
- Goal: suppose that both R and S are indexed by R-trees
  - How do we process the join efficiently?
  - How do we apply the filter/refinement process?



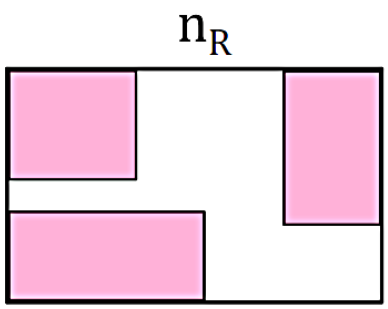


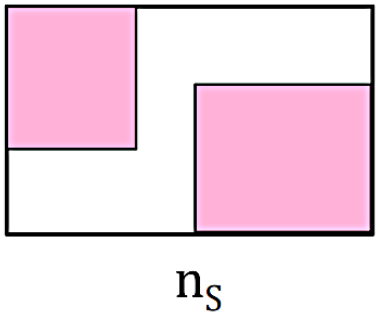
R-tree (Intersection) Join

Applies on two R-trees of spatial relations R and S

Observations:

- If a node $n_R \in R$ does not intersect $n_S \in S$, then no object $o_R \in R$ under n_R can intersect any object $o_S \in S$ under n_S
- Node MBRs at the high level of the trees restrict object combinations to be checked





- Utilize this property to traverse synchronously both trees following only entry pairs that intersect

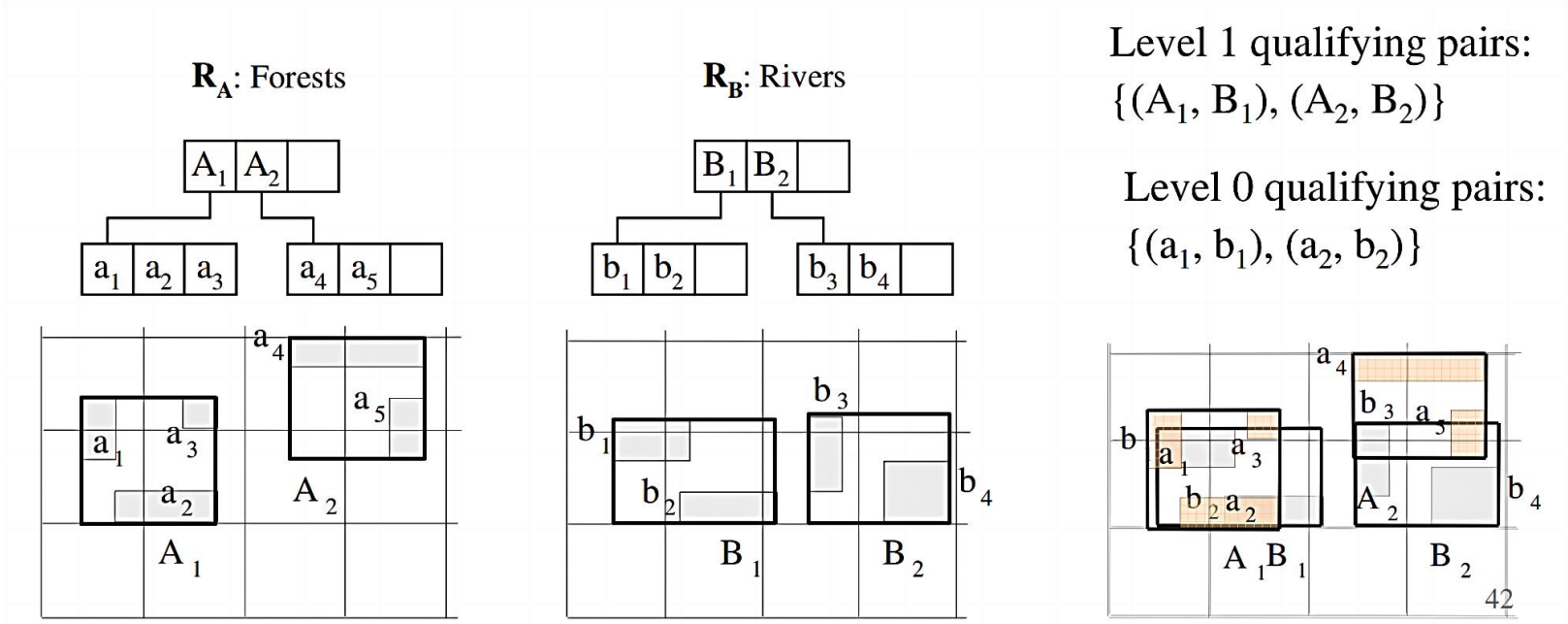
R-tree (Intersection) Join

- Initially called with parameters as the roots of the two trees
- This pseudo-code assumes that the trees have same height
  - easily extendable to trees of different heights

```
function RJ(Node n_R, Node n_S)
    for each e_i ∈ n_R
        for each e_j ∈ n_S, such that e_i.MBR ∩ e_j.MBR≠ ∅
            if n_R is a leaf node then /* n_S is also a leaf node */
                output (e_i.ptr,e_j.ptr); /* a pair of object-ids passing the filter step */
            else /* n_R, n_S are directory nodes */
                RJ(e_i.ptr,e_j.ptr); /* run recursively for the nodes pointed by intersecting entries*/
```

R-tree (Intersection) Join

- Example:
  - run for root(R_A), root(R_B)
  - for every intersecting pair there (e.g., A_1, B_1) run recursively for pointed nodes
  - intersecting pairs of leaf nodes are qualifying object MBR pairs



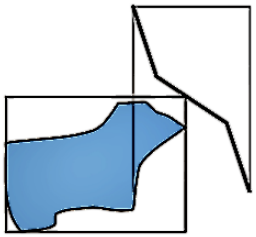
Filter-refinement steps

- A multi-step process is adopted for leaf pairs
  - MBR Filter step: find MBR pairs that intersect (done by R-tree join algorithm)
  - Geometric filter: compare some more detailed approximations to make conclusions (Why?)
  - Refinement: if the join predicate cannot be decided, then perform expensive refinement step
    - Apply computational geometry algorithms

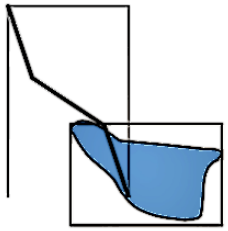
Very fast!

Fast!

Slow!



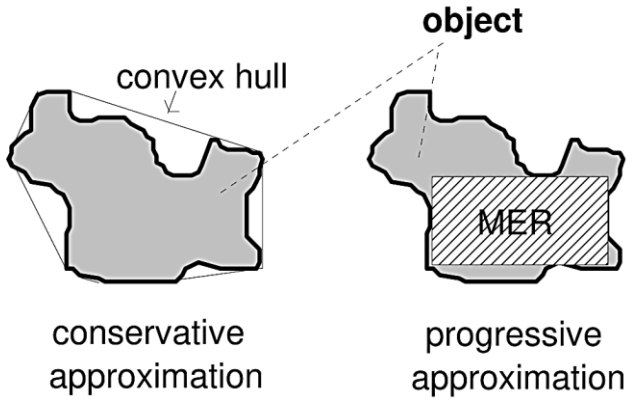
MBR filter step

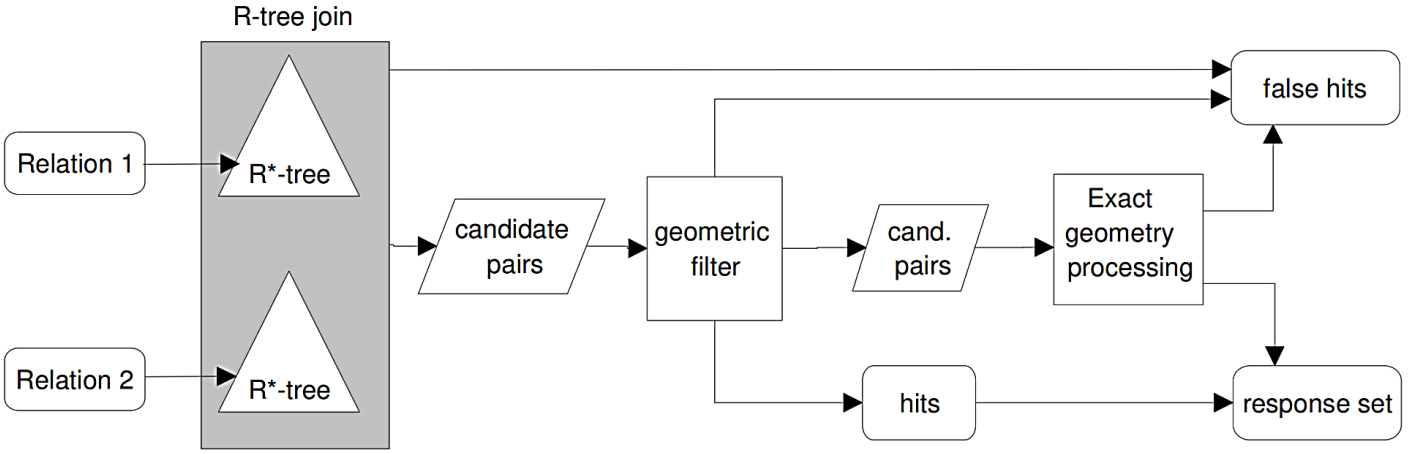


qualifying pair

Multi-step join processing

- Geometric filter (detail approximation)
  - Conservative approximation
    - E.g., convex hull
    - Detection of false hits
  - Progressive approximation
    - E.g., maximum enclosed rectangle
    - Detection of true hits
- Multi-step processing visualized:





Summary

- Nearest Neighbor Query
  - Distance lower-bounding property of MBR
  - Depth-first NN search
  - Best-first NN search
- Spatial Join
  - R-tree intersection join
  - Multistep filter-refinement process

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DBGroup

谢谢！

DBGroup @ SUSTech

Dr. Bo Tang (唐博)

tangb3@sustech.edu.cn