RANKING IN SPATIAL DATABASES

GÍSLI R. HJALTASON
HANAN SAMET

COMPUTER SCIENCE DEPARTMENT AND
CENTER FOR AUTOMATION RESEARCH AND
INSTITUTE FOR ADVANCED COMPUTER STUDIES
UNIVERSITY OF MARYLAND

COLLEGE PARK, MARYLAND 20742-3411 USA

RANKING PROBLEM

- Ex: Find all the houses in the database in the order of their distance from location *p* or object *o*
- Ex: Find the nearest city of population greater than 30,000 to Portland, Maine
- If we use an index on the locational attributes corresponding to the location of the cities, then we want to obtain the cities in the order of their distance from Portland, Maine
- The process ceases once the condition on the non-locational population attribute is satisfied
- Query is also a partial ranking query
- We are interested in a solution where if the closest record does not satisfy our query, then we can get the next closest record by continuing from where we last left off rather that having to restart from the reference point of the index
- Used in browsing applications
  1. can restrict query region
  2. can vary nature of features retrieved as well as query object
  3. can serve as basis of incremental spatial join operations
- Our solution is illustrated by an example spatial index making use of a disjoint decomposition (e.g., PR quadtree) although implemented for other spatial indexes including R-trees

MULTIATTRIBUTE INDEXING

- Indexes are used in databases to facilitate retrieval of records with similar values
- For each particular attribute, an index provides an ordering in increasing (or decreasing) order of the attribute value
- How to index multiple attributes?
  1. primary and secondary attributes
     - sort by first attribute
     - use second attribute to break ties
     - OK as long as we just want the records ordered by the first attribute
     - if we want the records ordered by the second attribute, then index is useless as consecutive records are not necessarily ordered by the value of the second attribute
  2. build an additional index on the second attribute
     - problem: takes up additional space
     - separate indexes are acceptable as long as queries don't make use of a combination of attribute values
     - not all combinations are meaningful if dimensional units of the attribute values differ

       Ex: age (years) and weight (pounds)

       a. unlikely to determine nearest record to the one of John Jones in terms of age and weight as we don't have a commonly accepted notion of the year-pound unit
       b. however, Boolean combinations are useful
          - range (i.e., window) or partial range queries
          - e.g., find all people between 25 and 30 years who weigh between 50 and 60 kilograms

SPATIAL DATABASES

- Distinguished from conventional databases, in part, by fact that some of the attributes are locational in which case they have a common dimensional unit which is distance in space
  1. distance unit is the same whether we are in one, two, or three dimensions, etc.
  2. if combine attributes and seek to find nearest record of type *t* to Chicago, then unit is distance regardless of whether there are one, two, three, ... (or even more) locational attributes associated with *t*
  3. nearest is not meaningful for combinations of non-locational attributes
  4. not all attributes with type distance are locational (e.g., size corresponding to pant length, height, waist, etc.)
- Spatial databases are differentiated by the type of records that they store
  1. points (e.g., locations of features)
     - zero volumetric measure
     - discrete
  2. features (i.e., space occupied by the features)
     - nonzero volumetric measure (i.e., extent)
     - continuous
- Records in a conventional database are always discrete
  1. can be viewed as points in a higher dimensional space
  2. difference is that for spatial attributes, dimensional unit is always distance in space

ORDERING DATA

- Use a combination of the values of the locational attributes
- Facilitates storage of records
- Desirable for ordering to preserve proximity — i.e., records close to each other in the multidimensional space should also be close to each other in the ordering
- Hashing is a way of achieving ordering

  1. explicit order
     - mapping from higher dimensional space to one-dimensional space
     - e.g., bit interleaving (Morton order), Peano-Hilbert, Sierpinski, etc.
     - result is a space-filling curve
     - no order has property that ALL records that are close to each other in the multidimensional space of the locational attributes are also close to each other in the range of the mapping
  2. implicit order
     - bucketing methods
     - sort records on the basis of the space they occupy and group into cells or buckets of finite capacity

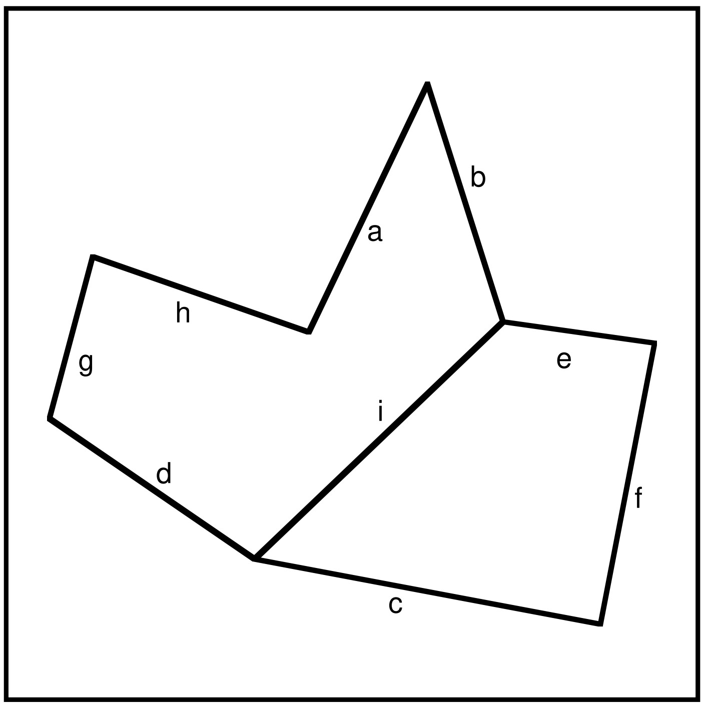
ORDERING ON THE BASIS OF SPATIAL OCCUPANCY

1. Minimum bounding rectangles
   - e.g., R-tree
   - good at distinguishing empty and non-empty space
   - drawbacks:
     a. non-disjoint decomposition of space
        - feature is associated with just one bounding rectangle while a point *p* can be in several bounding rectangles
        - just because we don't find a feature containing *p* in a bounding rectangle that overlaps *p* does not mean that features in other bounding rectangles do not contain *p*
        - may need to search entire space

     b. inability to correlate empty space in two maps
2. Disjoint cells
   - drawback: a feature may be reported more than once
     a. because it may have been decomposed into several pieces and thus be in several cells
     b. removing duplicates may require sorting
   - uniform grid
     a. all cells the same size
     b. drawback: possibility of many sparse cells
   - adaptive grid — quadtree variants
     a. regular decomposition
     b. all cells of width power of 2
   - partitions at arbitrary positions
     a. drawback: not a regular decomposition
     b. e.g., R^+-tree

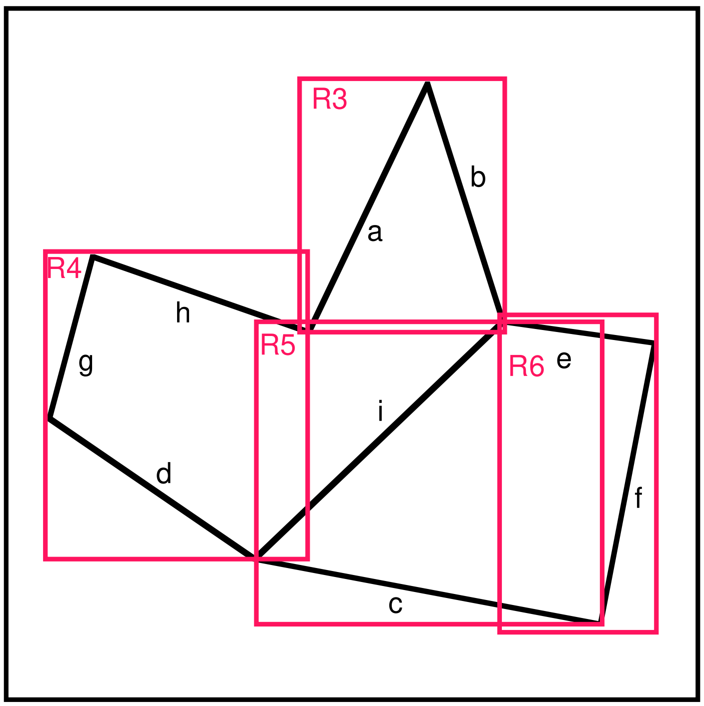
R-TREES

- Objects grouped into hierarchies, stored in another structure such as a B-tree
- Does not result in disjoint decomposition of space
- Object has single bounding rectangle, yet area that it spans may be included in several bounding rectangles
- Order (m,M) R-tree
  1. between $m \leqslant \lceil M/2 \rceil$ and M entries in each node except root
  2. at least 2 entries in root unless a leaf node



R-TREES

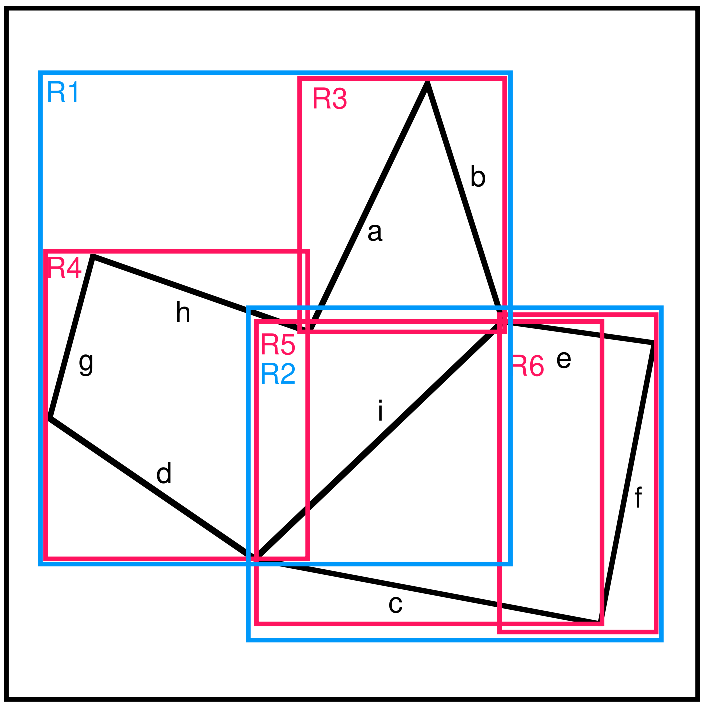
- Objects grouped into hierarchies, stored in another structure such as a B-tree
- Does not result in disjoint decomposition of space
- Object has single bounding rectangle, yet area that it spans may be included in several bounding rectangles
- Order (m,M) R-tree
  1. between $m \leqslant \lceil M/2 \rceil$ and M entries in each node except root
  2. at least 2 entries in root unless a leaf node

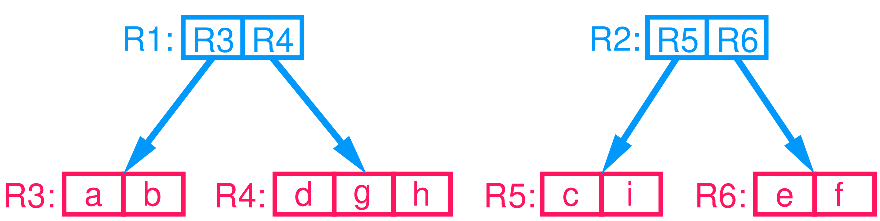


R3: | a | b | R4: | d | g | h | R5: | c | i | R6: | e | f |

R-TREES

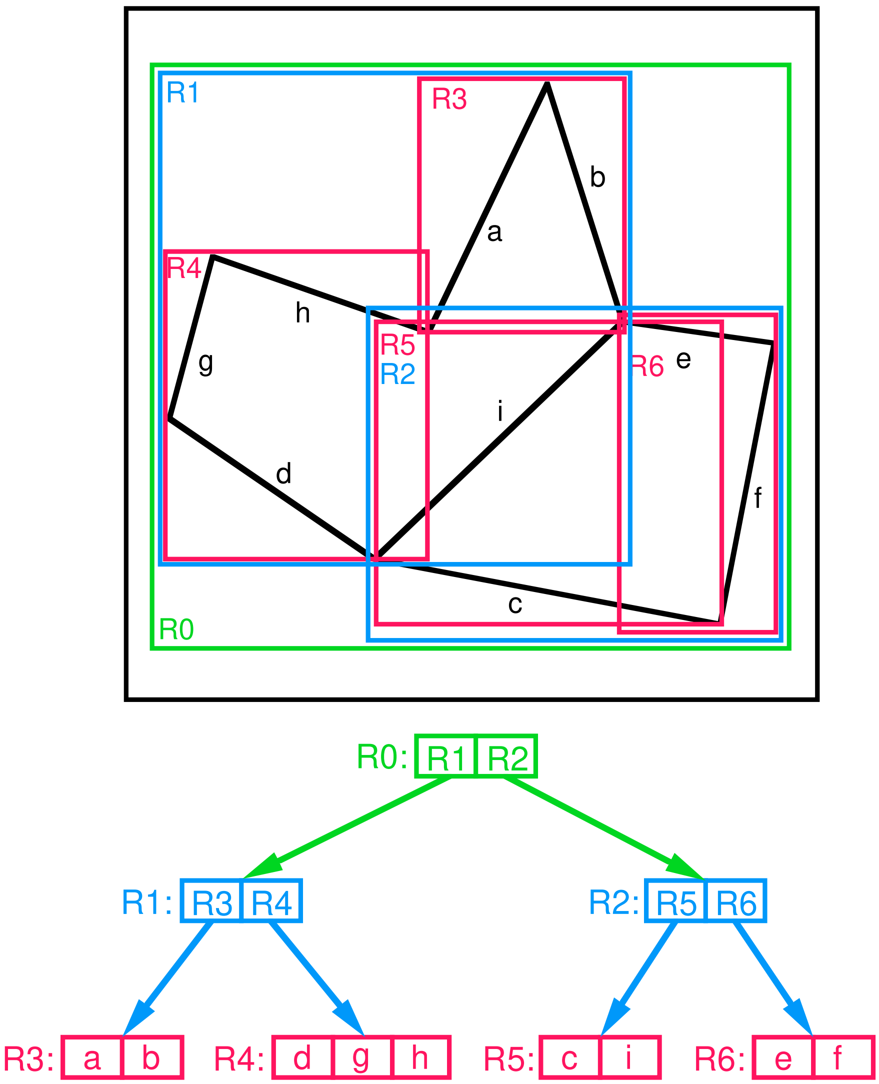
- Objects grouped into hierarchies, stored in another structure such as a B-tree
- Does not result in disjoint decomposition of space
- Object has single bounding rectangle, yet area that it spans may be included in several bounding rectangles
- Order (m,M) R-tree
  1. between $m \leqslant \lceil M/2 \rceil$ and M entries in each node except root
  2. at least 2 entries in root unless a leaf node





R-TREES

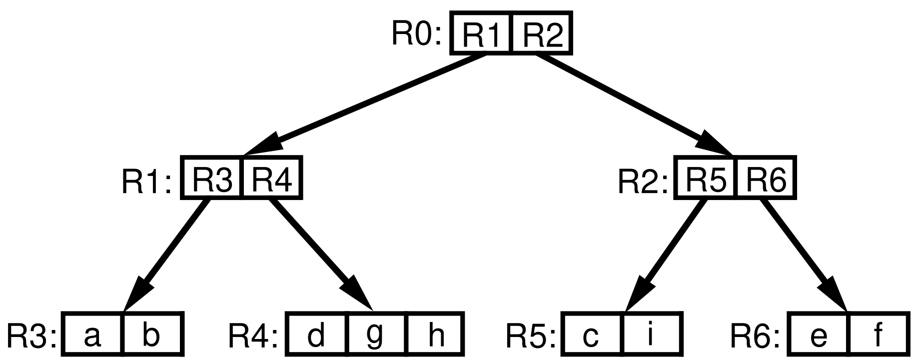
- Objects grouped into hierarchies, stored in another structure such as a B-tree
- Does not result in disjoint decomposition of space
- Object has single bounding rectangle, yet area that it spans may be included in several bounding rectangles
- Order (m,M) R-tree
  1. between $m \leqslant \lceil M/2 \rceil$ and M entries in each node except root
  2. at least 2 entries in root unless a leaf node

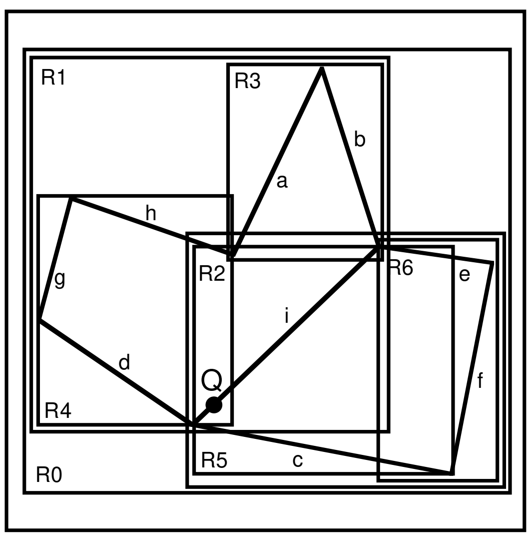


SEARCHING FOR A POINT OR LINE SEGMENT IN AN R-TREE

- May have to examine many nodes since a line segment can be contained in the covering rectangles of many nodes yet its record is contained in only one leaf node (e.g., i in R2, R3, R4, and R5)

Ex: Search for a line segment containing point Q

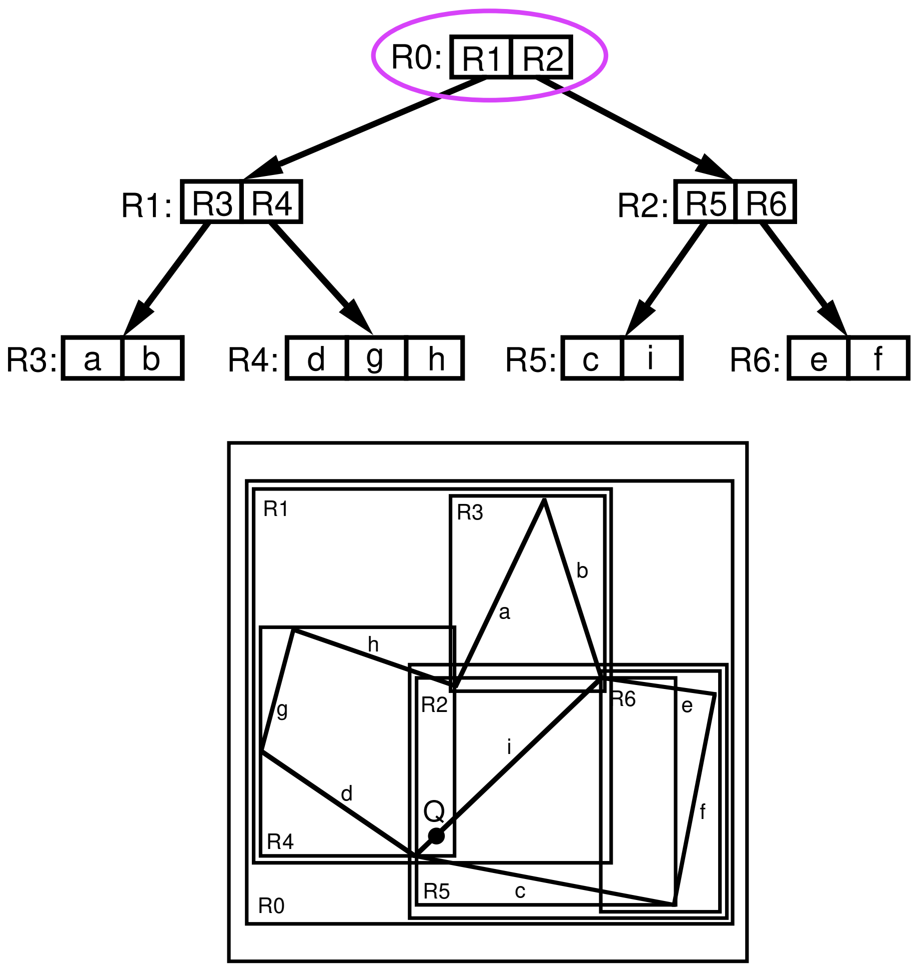




SEARCHING FOR A POINT OR LINE SEGMENT IN AN R-TREE

- May have to examine many nodes since a line segment can be contained in the covering rectangles of many nodes yet its record is contained in only one leaf node (e.g., i in R2, R3, R4, and R5)

Ex: Search for a line segment containing point Q

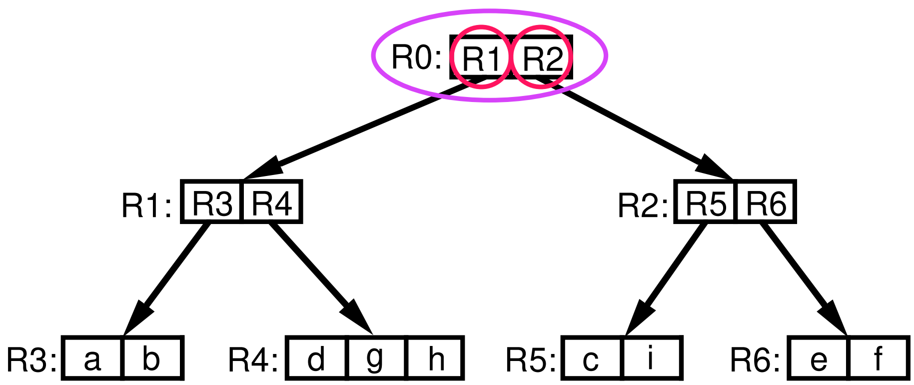


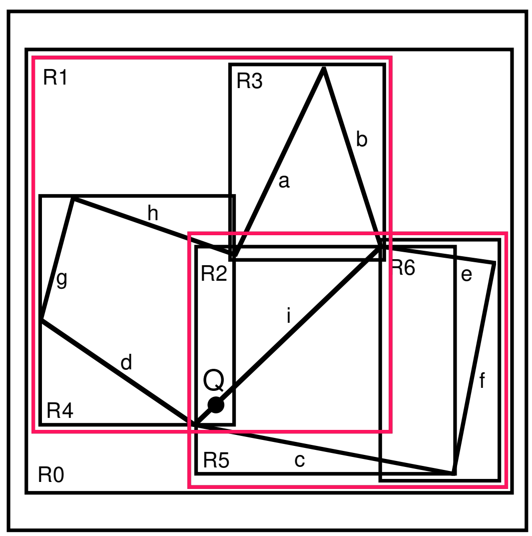
- Q is in R0

SEARCHING FOR A POINT OR LINE SEGMENT IN AN R-TREE

- May have to examine many nodes since a line segment can be contained in the covering rectangles of many nodes yet its record is contained in only one leaf node (e.g., i in R2, R3, R4, and R5)

Ex: Search for a line segment containing point Q



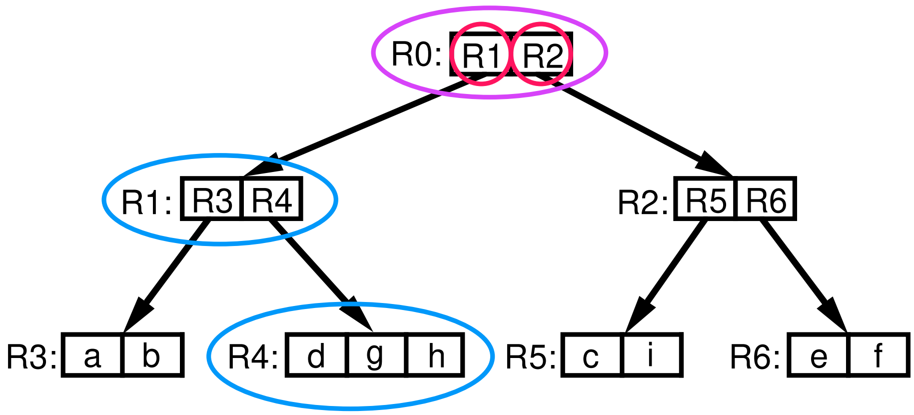


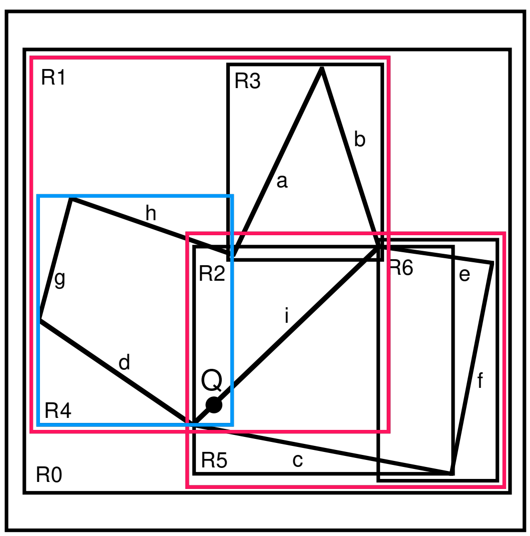
- Q is in R0
- Q can be in both R1 and R2

SEARCHING FOR A POINT OR LINE SEGMENT IN AN R-TREE

- May have to examine many nodes since a line segment can be contained in the covering rectangles of many nodes yet its record is contained in only one leaf node (e.g., i in R2, R3, R4, and R5)

Ex: Search for a line segment containing point Q



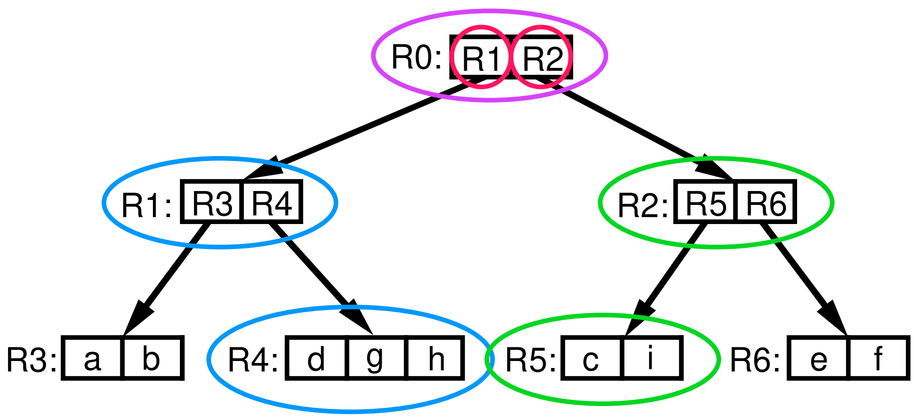


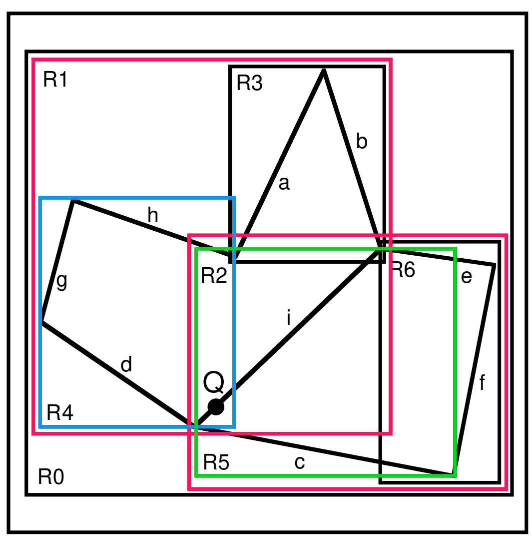
- Q is in R0
- Q can be in both R1 and R2
- Searching R1 first means that R4 is searched but this leads to failure even though Q is part of i which is in R4

SEARCHING FOR A POINT OR LINE SEGMENT IN AN R-TREE

- May have to examine many nodes since a line segment can be contained in the covering rectangles of many nodes yet its record is contained in only one leaf node (e.g., i in R2, R3, R4, and R5)

Ex: Search for a line segment containing point Q

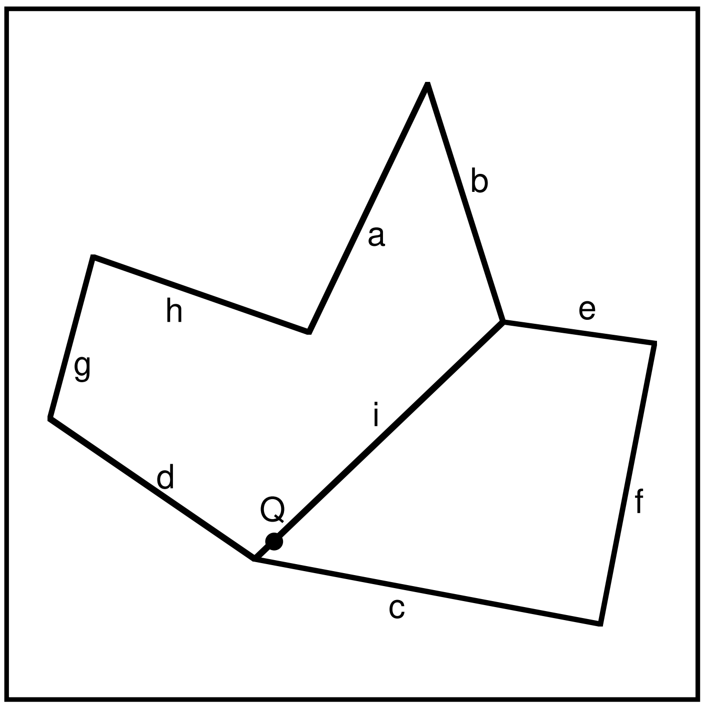




- Q is in R0
- Q can be in both R1 and R2
- Searching R1 first means that R4 is searched but this leads to failure even though Q is part of i which is in R4
- Searching R2 finds that Q can only be in R5

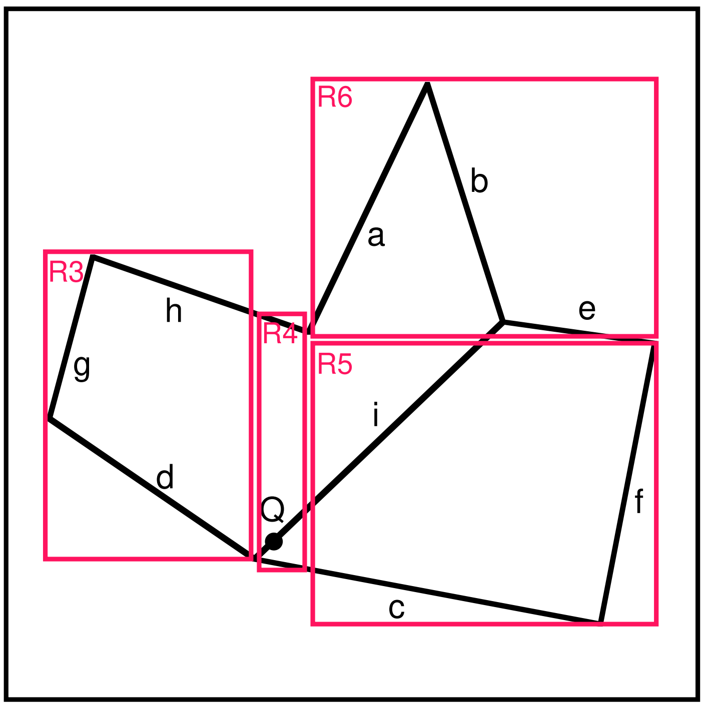
DISJOINT CELLS

- Objects decomposed into disjoint subobjects; each subobject in different cell
- Techniques differ in degree of regularity
- Drawback: in order to determine area covered by object, must retrieve all cells that it occupies
- R+-tree (also k-d-B-tree) and cell tree are examples of this technique



DISJOINT CELLS

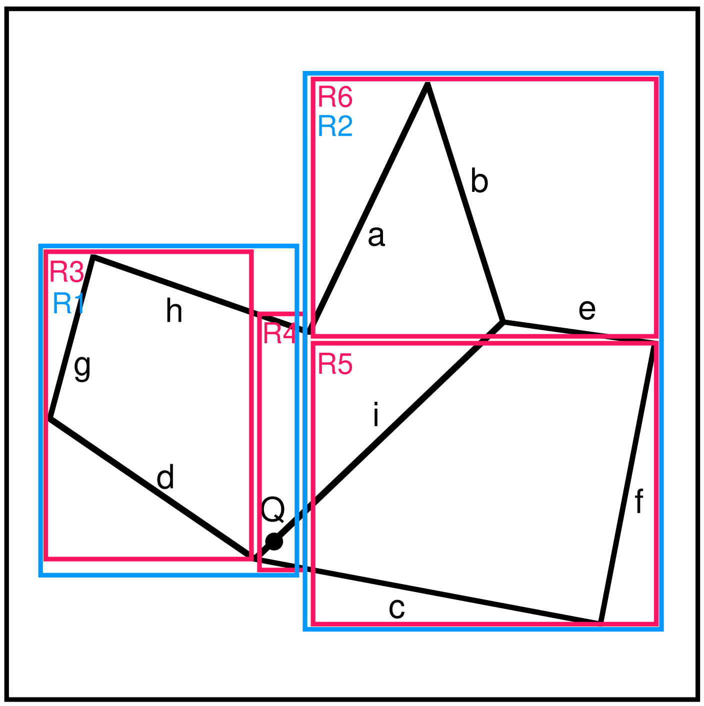
- Objects decomposed into disjoint subobjects; each subobject in different cell
- Techniques differ in degree of regularity
- Drawback: in order to determine area covered by object, must retrieve all cells that it occupies
- R+-tree (also k-d-B-tree) and cell tree are examples of this technique

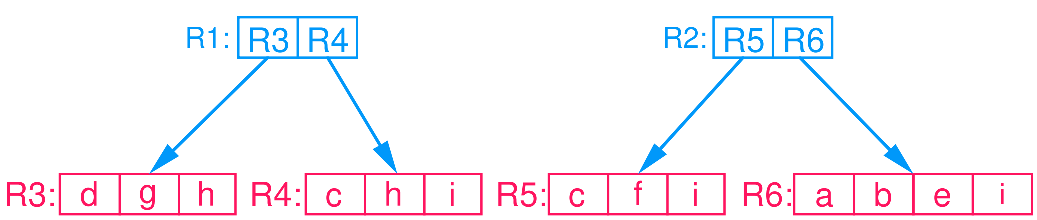


R3:	d	g	h		R4:	c	h	i		R5:	c	f	i		R6:	a	b	e	i

DISJOINT CELLS

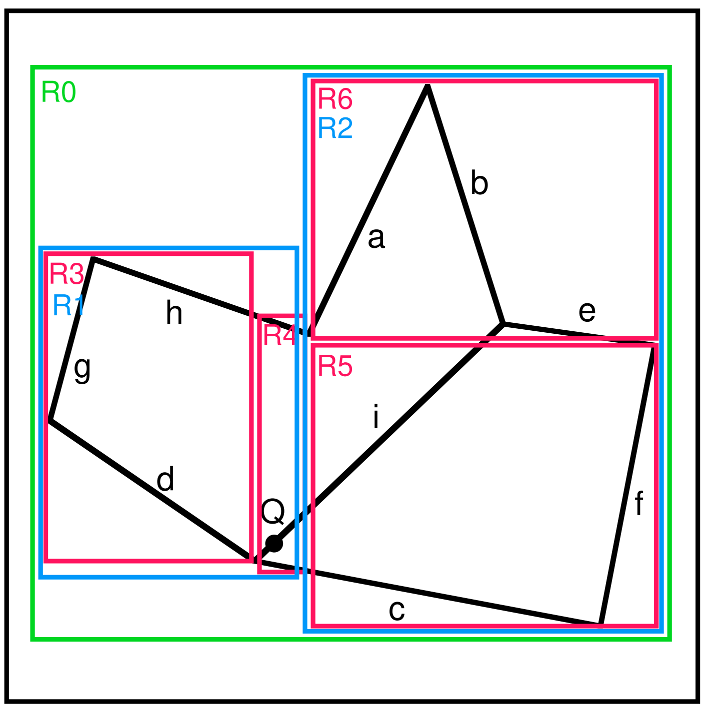
- Objects decomposed into disjoint subobjects; each subobject in different cell
- Techniques differ in degree of regularity
- Drawback: in order to determine area covered by object, must retrieve all cells that it occupies
- R+-tree (also k-d-B-tree) and cell tree are examples of this technique

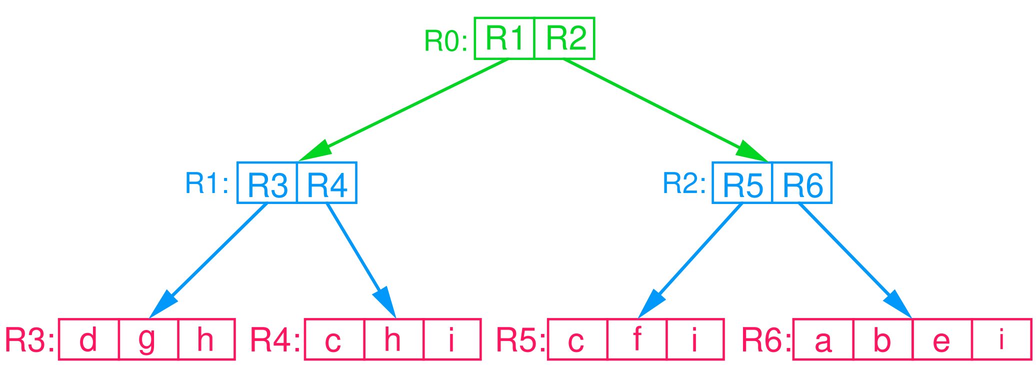




DISJOINT CELLS

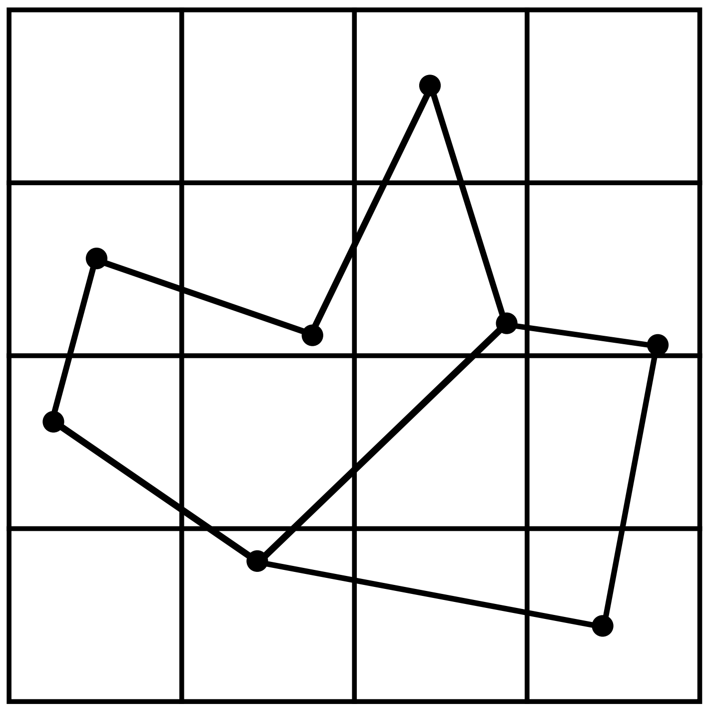
- Objects decomposed into disjoint subobjects; each subobject in different cell
- Techniques differ in degree of regularity
- Drawback: in order to determine area covered by object, must retrieve all cells that it occupies
- R+-tree (also k-d-B-tree) and cell tree are examples of this technique





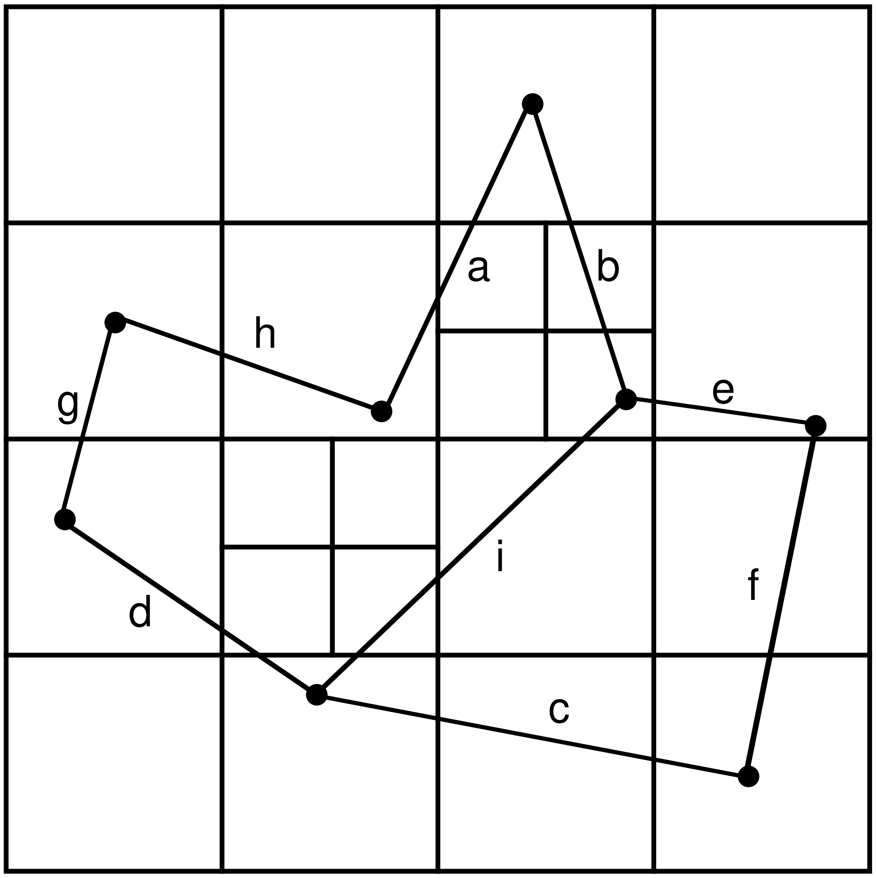
UNIFORM GRID

- Ideal for uniformly distributed data
- Supports set-theoretic operations
- Spatial data (e.g., line segment data) is rarely uniformly distributed



PM1 QUADTREE

- Vertex-based (one vertex per block)



DECOMPOSITION RULE:

Partitioning occurs when a block contains more than one segment unless all the segments are incident at the same vertex which is also in the same block

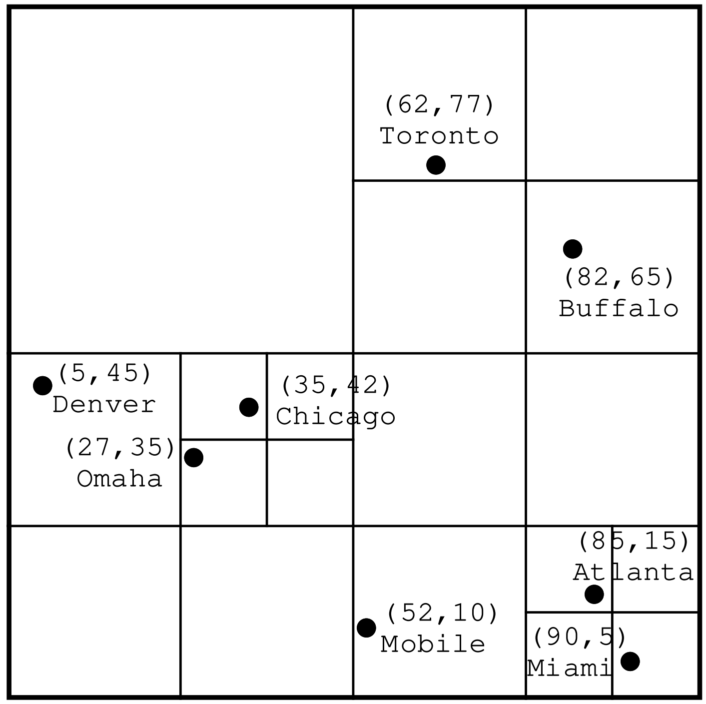
- Shape independent of order of insertion

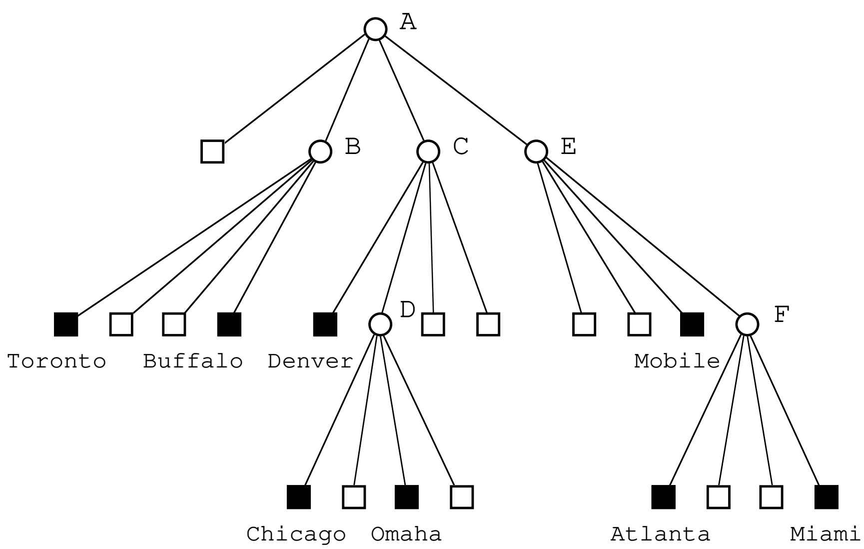
RANKING IN SPATIAL DATABASES

1. Goal: use ordering provided by the index to rank data based on its closeness to a given value *v* of attribute *a*
2. Ranking is a general problem
3. If ranking on the basis of the value of just one attribute, then the implicit and explicit indexes are equivalent and we can derive the ranking directly from the index
   - index is obtained by sorting the data with respect to a reference point (e.g., zero for data of type ratio)
   - e.g., rank individual in order of the closeness of their weight to that of John Smith which is 150 pounds
     a. look up 150 in the index and then proceed in the two directions along the index to get the nearest individuals by weight in constant time
     b. no need to rebuild index if want to also ask about Sam Jones whose weight is 200 pounds
4. If ranking involves more than one locational attribute all of whose values are of type distance, then more complex
   - if explicit index then not possible
     a. e.g., build index on basis of distance from point p_1 using a particular distance metric
     b. if want ranking on basis of distance from p_2, then need to resort as their distance from p_2 is not obtained by the addition or subtraction of some constant equal to the distance from p_1 to p_2 as is the case when there is just one attribute involved
   - use an implicit index based on sorting objects with respect to the space they occupy instead of each other or some common reference point (e.g., bucketing)

PR QUADTREE

- Decompose space until just one point per block





BOTTOM-UP SOLUTION

- Algorithm:
  1. locate block *b* containing query object *q*
  2. find nearest feature *o* by examining adjacent neighboring blocks of *b* in clockwise order
- Depending on nature of the distance metric (e.g., Euclidean) may have to visit block not immediately adjacent to *b*
- Obtain neighbors using neighbor-finding techniques which don’t restart search at root of the tree
- Especially fast if nearest feature is in a brother of *b*
- Worst case requires visiting all of the blocks around the node
- If need to find the next closest feature, then have to restart search from the beginning rather than from where last left off

TOP-DOWN SOLUTION

1. Easy to find leaf node(s) *b* containing query object *q*
2. Use recursion to keep track of blocks seen already
3. Problem is that *b* may be empty or not contain the nearest feature
   - need to unwind recursion to find nearest feature
4. Algorithm breaks down if want to get second nearest feature without restarting search from the start
   - when visit a leaf node *n*, we need to remember features already encountered which may still not yet be the closest ones

4. Solution is to replace the recursion stack by a priority queue to record:
   - blocks with unvisited descendants
   - features which have not yet been reported
5. Priority queue insertion rules:
   - feature *o* in *b* is inserted in the queue only if *o* has not yet been reported
     a. check if *o*'s distance from *q* is less than the distance from *b* to *q*
     b. if yes, then *o* must have been encountered in a block *c* which was closer to *q* than *b* and hence already been reported

ALGORITHM REQUIREMENTS

- Features can be of arbitrary type (points, rectangles, polygons, etc.)
- Must have a distance function between:
  1. query object type and the feature type stored in the index (*feature metric*)
  2. query object type and the container block type (*block metric*)
- Two distance functions must be consistent with each other
  1. means that for a feature f with distance $d(f,q)=s$, then there must exist a block b containing f such that $d(b,q) \leq s$
  2. always true if both distance functions are based on the same metric
     - e.g., Euclidean, Manhattan, Chessboard
  3. also implies that distance from a query object to the block that contains it is zero
- Works in any dimension
- Location of the query object q need not be in the space spanned by the dataset (e.g., outside a window whose features are to be ranked with respect to q)

ALGORITHM

```
IncNearest(QueryObject,SpatialIndex)
Queue ← NewPriorityQueue()
Block ← RootBlock(SpatialIndex)
Enqueue(Queue,Dist(Block,QueryObject),Block)
while not IsEmpty(Queue) do
  Element ← Dequeue(Queue)
  if Element is a spatial object then
    while Element = First(Queue) do DeleteFirst(Queue)
    Report Element
  else if Element is a leaf block then
    for each Object in leaf block Element do
      if Dist(Object,QueryObject) ≥
          Dist(Element,QueryObject) then
        Enqueue(Queue,Dist(Object,QueryObject),Object)
  else /* Element is a non-leaf container block */
    for each Child block of container block Element
              in SpatialIndex do
      Enqueue(Queue,Dist(Child,QueryObject),Child)
```

ALGORITHM

```
IncNearest(QueryObject,SpatialIndex)
1. Queue ← NewPriorityQueue()
2. Block ← RootBlock(SpatialIndex)
3. Enqueue(Queue,Dist(Block,QueryObject),Block)
4. while not IsEmpty(Queue) do
5.   Element ← Dequeue(Queue)
6.   if Element is a spatial object then
7.     while Element = First(Queue) do DeleteFirst(Queue)
8.     Report Element
9.   else if Element is a leaf block then
10.    for each Object in leaf block Element do
11.      if Dist(Object,QueryObject) ≥
12.         Dist(Element,QueryObject) then
13.        Enqueue(Queue,Dist(Object,QueryObject),Object)
14.  else /* Element is a non-leaf container block */
15.    for each Child block of container block Element
16.               in SpatialIndex do
17.      Enqueue(Queue,Dist(Child,QueryObject),Child)
```

1. Lines 1-3 initialize the priority queue

ALGORITHM

```
IncNearest(QueryObject,SpatialIndex)
1. Queue ← NewPriorityQueue()
2. Block ← RootBlock(SpatialIndex)
3. Enqueue(Queue,Dist(Block,QueryObject),Block)
4. while not IsEmpty(Queue) do
5.   Element ← Dequeue(Queue)
6.   if Element is a spatial object then
7.     while Element = First(Queue) do DeleteFirst(Queue)
8.     Report Element
9.   else if Element is a leaf block then
10.    for each Object in leaf block Element do
11.      if Dist(Object,QueryObject) ≥
12.         Dist(Element,QueryObject) then
13.        Enqueue(Queue,Dist(Object,QueryObject),Object)
14.  else /* Element is a non-leaf container block */
15.    for each Child block of container block Element
16.               in SpatialIndex do
17.      Enqueue(Queue,Dist(Child,QueryObject),Child)
```

1. Lines 1-3 initialize the priority queue
2. Line 8 reports a feature
   - coroutine-like behavior as control will resume here to get the next nearest

ALGORITHM

```
IncNearest(QueryObject,SpatialIndex)
1.  Queue ← NewPriorityQueue()
2.  Block ← RootBlock(SpatialIndex)
3.  Enqueue(Queue,Dist(Block,QueryObject),Block)
4.  while not IsEmpty(Queue) do
5.    Element ← Dequeue(Queue)
6.    if Element is a spatial object then
7.      while Element = First(Queue) do DeleteFirst(Queue)
8.      Report Element
9.    else if Element is a leaf block then
10.     for each Object in leaf block Element do
11.       if Dist(Object,QueryObject) ≥
12.          Dist(Element,QueryObject) then
13.         Enqueue(Queue,Dist(Object,QueryObject),Object)
14.   else /* Element is a non-leaf container block */
15.     for each Child block of container block Element
16.               in SpatialIndex do
17.       Enqueue(Queue,Dist(Child,QueryObject),Child)
```

1. Lines 1-3 initialize the priority queue
2. Line 8 reports a feature
   - coroutine-like behavior as control will resume here to get the next nearest
3. Line 11 ensures that features that have already been reported are not put on the queue again
   - to work properly, blocks must be retrieved from the queue before spatial features at the same distance
   - otherwise, a feature at the same distance as the block might be retrieved again after the block has been processed and we would not know that it had already been reported

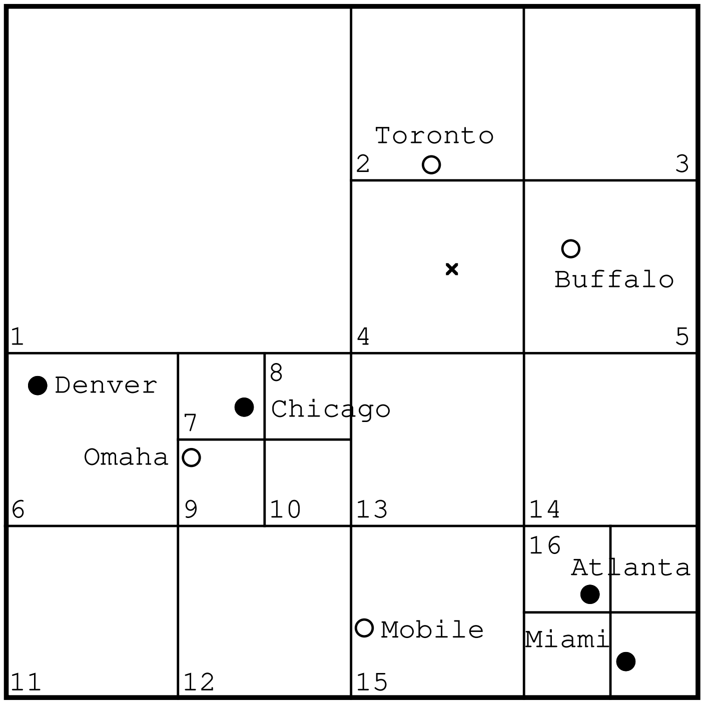
ALGORITHM

```
IncNearest(QueryObject,SpatialIndex)
1.  Queue ← NewPriorityQueue()
2.  Block ← RootBlock(SpatialIndex)
3.  Enqueue(Queue,Dist(Block,QueryObject),Block)
4.  while not IsEmpty(Queue) do
5.    Element ← Dequeue(Queue)
6.    if Element is a spatial object then
7.      while Element = First(Queue) do DeleteFirst(Queue)
8.      Report Element
9.    else if Element is a leaf block then
10.     for each Object in leaf block Element do
11.       if Dist(Object,QueryObject) ≥
12.          Dist(Element,QueryObject) then
13.         Enqueue(Queue,Dist(Object,QueryObject),Object)
14.   else /* Element is a non-leaf container block */
15.     for each Child block of container block Element
16.                 in SpatialIndex do
17.       Enqueue(Queue,Dist(Child,QueryObject),Child)
```

1. Lines 1-3 initialize the priority queue
2. Line 8 reports a feature
   - coroutine-like behavior as control will resume here to get the next nearest
3. Line 11 ensures that features that have already been reported are not put on the queue again
   - to work properly, blocks must be retrieved from the queue before spatial features at the same distance
   - otherwise, a feature at the same distance as the block might be retrieved again after the block has been processed and we would not know that it had already been reported
4. Line 7 eliminates duplicate instances of a feature from the queue
   - possible for disjoint feature decompositions (e.g., PMR quadtree, R+-tree)
   - duplicates are in front by virtue of having an ordering on the queue
   - more efficient than checking for membership in queue

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



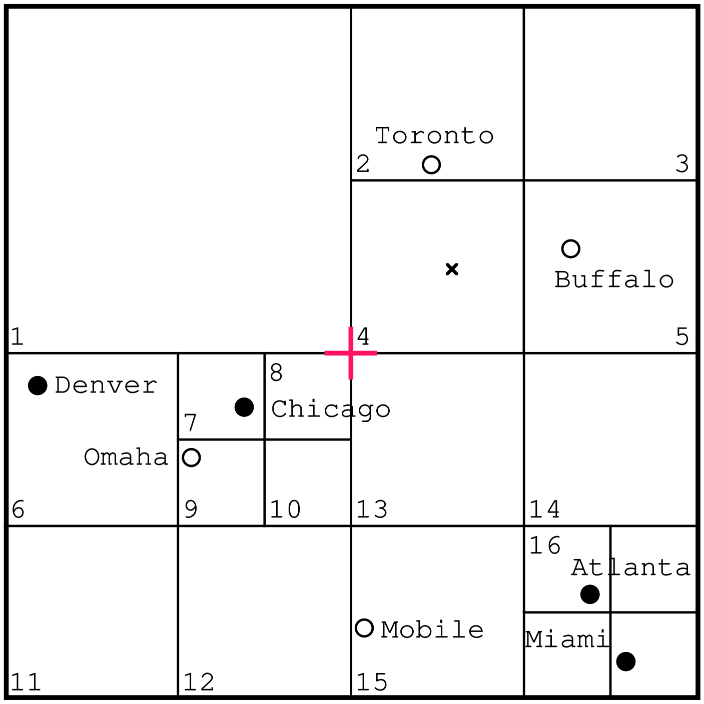
- Blocks labeled “depth/ NW-most descendant”
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn’t satisfy query

City	Pop (1000)	Pos
Atlanta	4,129	(85,15)
Buffalo	764	(82,65)
Chicago	6,532	(35,42)
Denver	1,381	(5,45)
Mobile	504	(52,10)
Omaha	416	(27,35)
Toronto	904	(62,77)
Miami	5,250	(90,5)

1. Initially, queue only contains root: [0/1]

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



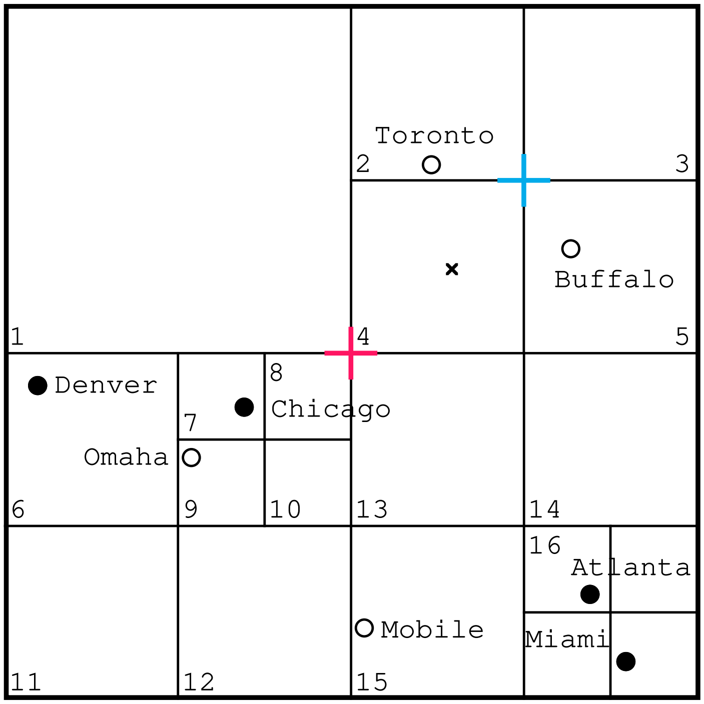
- Blocks labeled "depth/ NW-most descendant"
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn't satisfy query

City	Pop (1000)	Pos
Atlanta	4,129	(85,15)
Buffalo	764	(82,65)
Chicago	6,532	(35,42)
Denver	1,381	(5,45)
Mobile	504	(52,10)
Omaha	416	(27,35)
Toronto	904	(62,77)
Miami	5,250	(90,5)

1. Initially, queue only contains root: [0/1]
2. Dequeue root and enqueue 4 sons: [1/2 1/13 1/1 1/6]

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



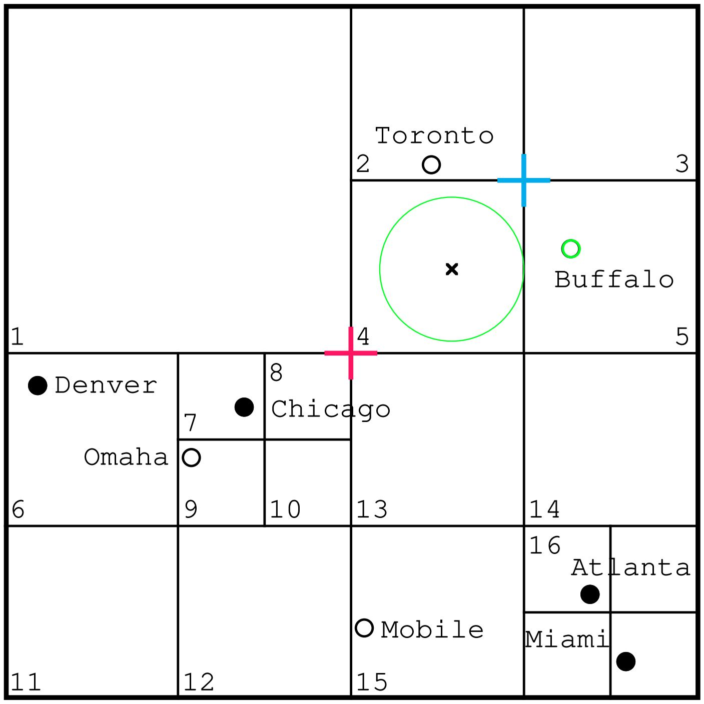
- Blocks labeled “depth/ NW-most descendant”
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn’t satisfy query

City	Pop (1000)	Pos
Atlanta	4,129	(85,15)
Buffalo	764	(82,65)
Chicago	6,532	(35,42)
Denver	1,381	(5,45)
Mobile	504	(52,10)
Omaha	416	(27,35)
Toronto	904	(62,77)
Miami	5,250	(90,5)

1. Initially, queue only contains root: [0/1]
2. Dequeue root and enqueue 4 sons: [1/2 1/13 1/1 1/6]
3. Dequeue 1/2 and enqueue 4 sons: [2/4 2/5 1/13 2/2 1/1 2/3 1/6]

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



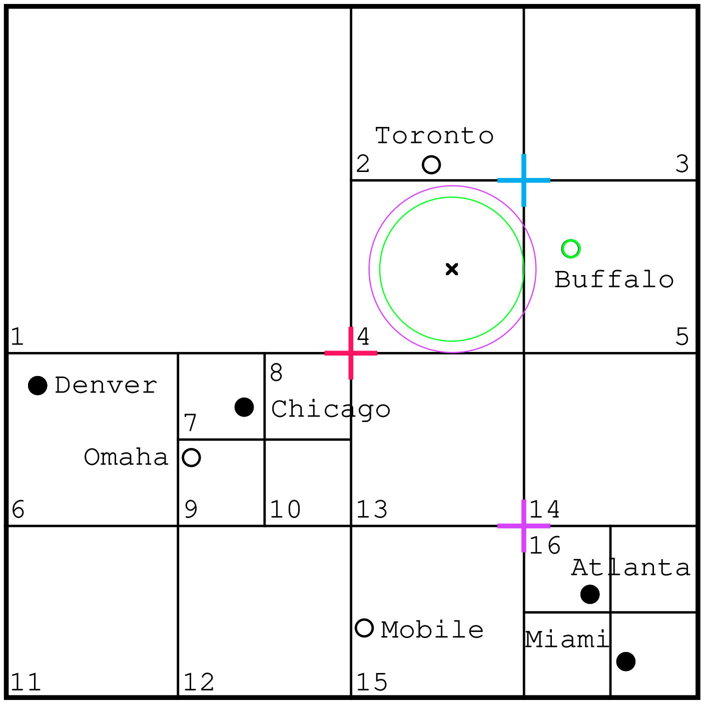
- Blocks labeled "depth/ NW-most descendant"
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn't satisfy query

City	Pop (1000)	Pos
Atlanta	4,129	(85,15)
Buffalo	764	(82,65)
Chicago	6,532	(35,42)
Denver	1,381	(5,45)
Mobile	504	(52,10)
Omaha	416	(27,35)
Toronto	904	(62,77)
Miami	5,250	(90,5)

1. Initially, queue only contains root: [0/1]
2. Dequeue root and enqueue 4 sons: [1/2 1/13 1/1 1/6]
3. Dequeue 1/2 and enqueue 4 sons: [2/4 2/5 1/13 2/2 1/1 2/3 1/6]
4. Dequeue 2/4 (empty); dequeue 2/5 containing Buffalo; enqueue Buffalo: [1/13 2/2 1/1 2/3 Buffalo 1/6]

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



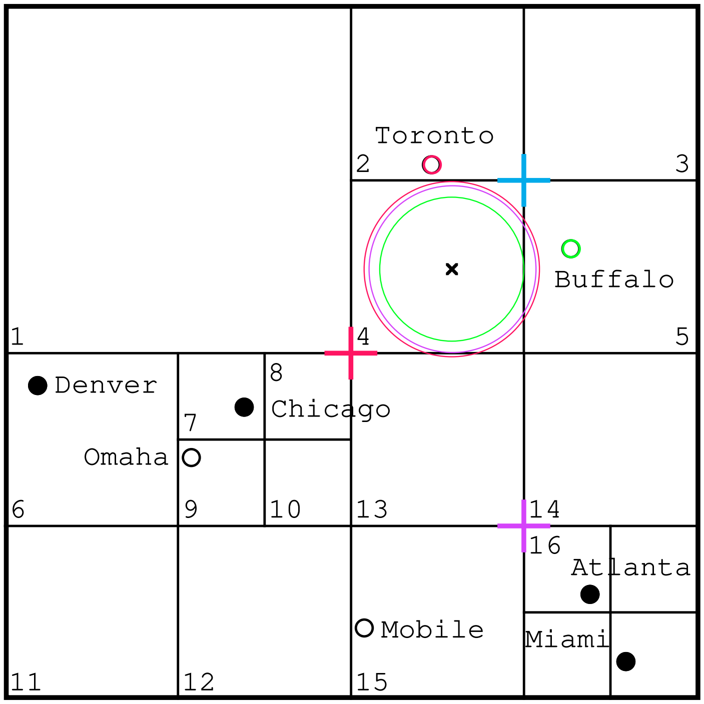
- Blocks labeled “depth/ NW-most descendant”
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn’t satisfy query

City	Pop (1000)	Pos
Atlanta	4,129	(85,15)
Buffalo	764	(82,65)
Chicago	6,532	(35,42)
Denver	1,381	(5,45)
Mobile	504	(52,10)
Omaha	416	(27,35)
Toronto	904	(62,77)
Miami	5,250	(90,5)

1. Initially, queue only contains root: [0/1]
2. Dequeue root and enqueue 4 sons: [1/2 1/13 1/1 1/6]
3. Dequeue 1/2 and enqueue 4 sons: [2/4 2/5 1/13 2/2 1/1 2/3 1/6]
4. Dequeue 2/4 (empty); dequeue 2/5 containing Buffalo; enqueue Buffalo: [1/13 2/2 1/1 2/3 Buffalo 1/6]
5. Dequeue 1/13 and enqueue 4 sons: [2/13 2/2 1/1 2/14 2/3 Buffalo 1/6 2/15 2/16]

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



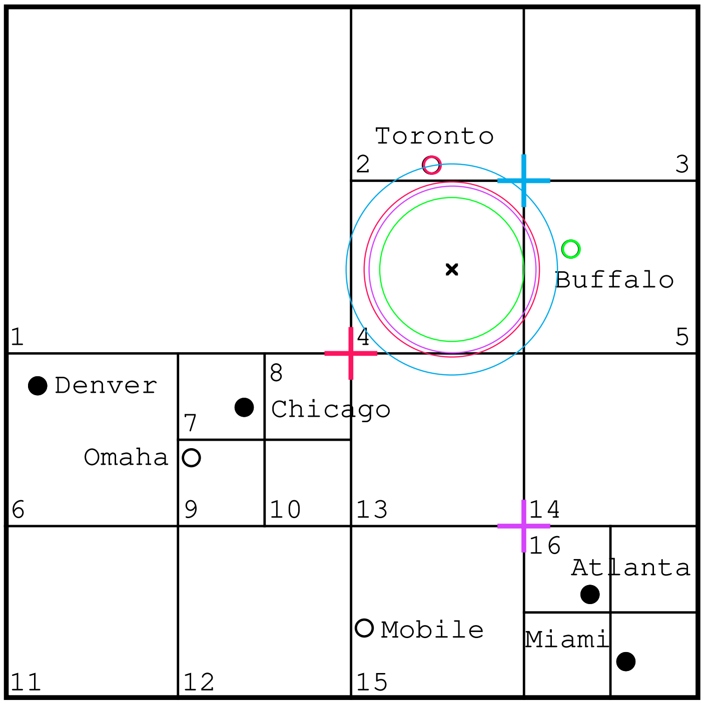
- Blocks labeled "depth/ NW-most descendant"
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn't satisfy query

City	Pop (1000)	Pos
Atlanta	4,129	(85,15)
Buffalo	764	(82,65)
Chicago	6,532	(35,42)
Denver	1,381	(5,45)
Mobile	504	(52,10)
Omaha	416	(27,35)
Toronto	904	(62,77)
Miami	5,250	(90,5)

1. Initially, queue only contains root: [0/1]
2. Dequeue root and enqueue 4 sons: [1/2 1/13 1/1 1/6]
3. Dequeue 1/2 and enqueue 4 sons: [2/4 2/5 1/13 2/2 1/1 2/3 1/6]
4. Dequeue 2/4 (empty); dequeue 2/5 containing Buffalo; enqueue Buffalo: [1/13 2/2 1/1 2/3 Buffalo 1/6]
5. Dequeue 1/13 and enqueue 4 sons: [2/13 2/2 1/1 2/14 2/3 Buffalo 1/6 2/15 2/16]
6. Dequeue 2/13 (empty); dequeue 2/2 containing Toronto; enqueue Toronto: [1/1 Toronto 2/14 2/3 Buffalo 1/6 2/15 2/16]

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



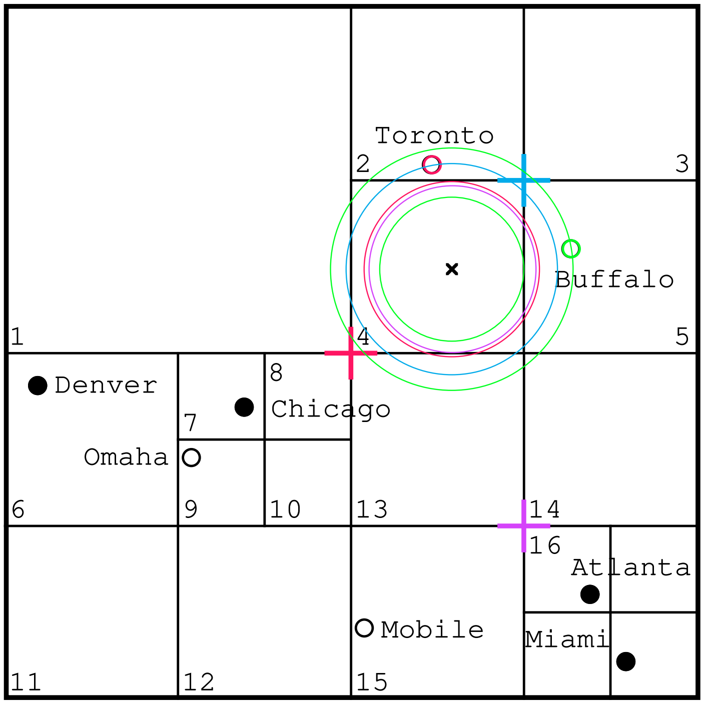
- Blocks labeled “depth/ NW-most descendant”
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn’t satisfy query

	City	Pop (1000)	Pos
	Atlanta	4,129	(85,15)
	Buffalo	764	(82,65)
	Chicago	6,532	(35,42)
	Denver	1,381	(5,45)
	Mobile	504	(52,10)
	Omaha	416	(27,35)
1	Toronto	904	(62,77)
	Miami	5,250	(90,5)

1. Initially, queue only contains root: [0/1]
2. Dequeue root and enqueue 4 sons: [1/2 1/13 1/1 1/6]
3. Dequeue 1/2 and enqueue 4 sons: [2/4 2/5 1/13 2/2 1/1 2/3 1/6]
4. Dequeue 2/4 (empty); dequeue 2/5 containing Buffalo; enqueue Buffalo: [1/13 2/2 1/1 2/3 Buffalo 1/6]
5. Dequeue 1/13 and enqueue 4 sons: [2/13 2/2 1/1 2/14 2/3 Buffalo 1/6 2/15 2/16]
6. Dequeue 2/13 (empty); dequeue 2/2 containing Toronto; enqueue Toronto: [1/1 Toronto 2/14 2/3 Buffalo 1/6 2/15 2/16]
7. Dequeue 1/1 which is empty; dequeue Toronto (population too small); dequeue 2/14 and 2/3 (empty): [Buffalo 1/6 2/15 2/16]

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



- Blocks labeled "depth/ NW-most descendant"
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn't satisfy query

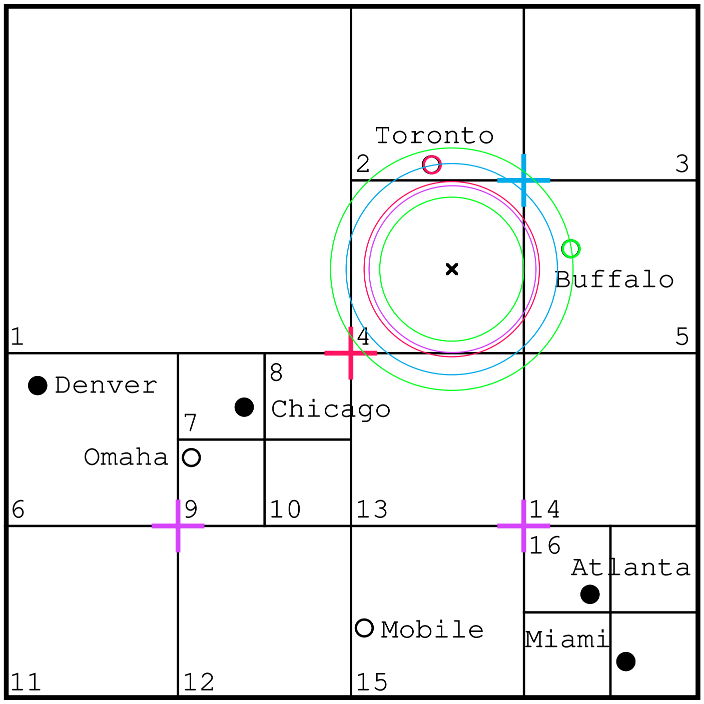
	City	Pop (1000)	Pos
	Atlanta	4,129	(85,15)
2	Buffalo	764	(82,65)
	Chicago	6,532	(35,42)
	Denver	1,381	(5,45)
	Mobile	504	(52,10)
	Omaha	416	(27,35)
1	Toronto	904	(62,77)
	Miami	5,250	(90,5)

1. Initially, queue only contains root: [0/1]
2. Dequeue root and enqueue 4 sons: [1/2 1/13 1/1 1/6]
3. Dequeue 1/2 and enqueue 4 sons: [2/4 2/5 1/13 2/2 1/1 2/3 1/6]
4. Dequeue 2/4 (empty); dequeue 2/5 containing Buffalo; enqueue Buffalo: [1/13 2/2 1/1 2/3 Buffalo 1/6]
5. Dequeue 1/13 and enqueue 4 sons: [2/13 2/2 1/1 2/14 2/3 Buffalo 1/6 2/15 2/16]
6. Dequeue 2/13 (empty); dequeue 2/2 containing Toronto; enqueue Toronto: [1/1 Toronto 2/14 2/3 Buffalo 1/6 2/15 2/16]
7. Dequeue 1/1 which is empty; dequeue Toronto (population too small); dequeue 2/14 and 2/3 (empty): [Buffalo 1/6 2/15 2/16]
8. Dequeue Buffalo (population too small): [1/6 2/15 2/16]

9 8 7 6 5 4 3 2 1
v g z r v g z r b

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



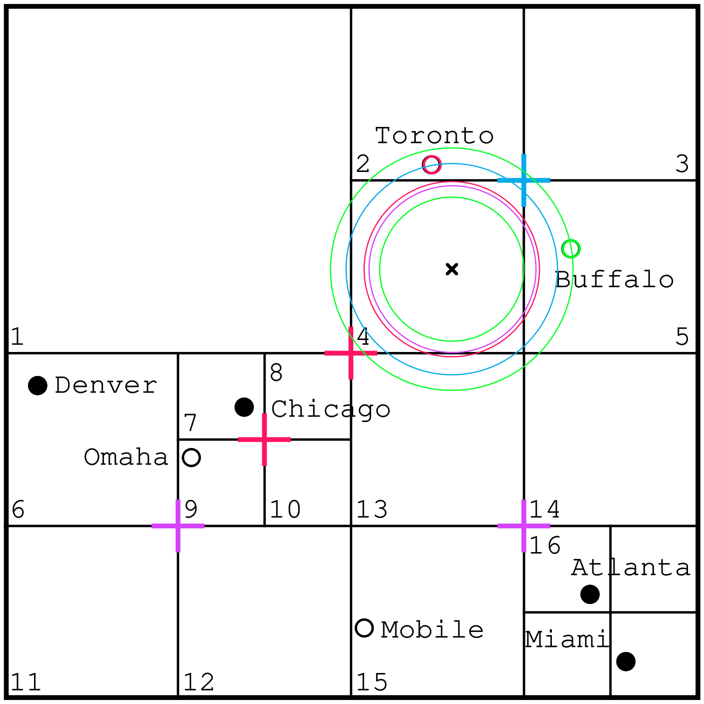
- Blocks labeled “depth/ NW-most descendant”
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn’t satisfy query

	City	Pop (1000)	Pos
	Atlanta	4,129	(85,15)
2	Buffalo	764	(82,65)
	Chicago	6,532	(35,42)
	Denver	1,381	(5,45)
	Mobile	504	(52,10)
	Omaha	416	(27,35)
1	Toronto	904	(62,77)
	Miami	5,250	(90,5)

1. Initially, queue only contains root: [0/1]
2. Dequeue root and enqueue 4 sons: [1/2 1/13 1/1 1/6]
3. Dequeue 1/2 and enqueue 4 sons: [2/4 2/5 1/13 2/2 1/1 2/3 1/6]
4. Dequeue 2/4 (empty); dequeue 2/5 containing Buffalo; enqueue Buffalo: [1/13 2/2 1/1 2/3 Buffalo 1/6]
5. Dequeue 1/13 and enqueue 4 sons: [2/13 2/2 1/1 2/14 2/3 Buffalo 1/6 2/15 2/16]
6. Dequeue 2/13 (empty); dequeue 2/2 containing Toronto; enqueue Toronto: [1/1 Toronto 2/14 2/3 Buffalo 1/6 2/15 2/16]
7. Dequeue 1/1 which is empty; dequeue Toronto (population too small); dequeue 2/14 and 2/3 (empty): [Buffalo 1/6 2/15 2/16]
8. Dequeue Buffalo (population too small): [1/6 2/15 2/16]
9. Dequeue 1/6 and enqueue 4 sons: [2/7 2/15 2/16 2/12 2/6 2/11]

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



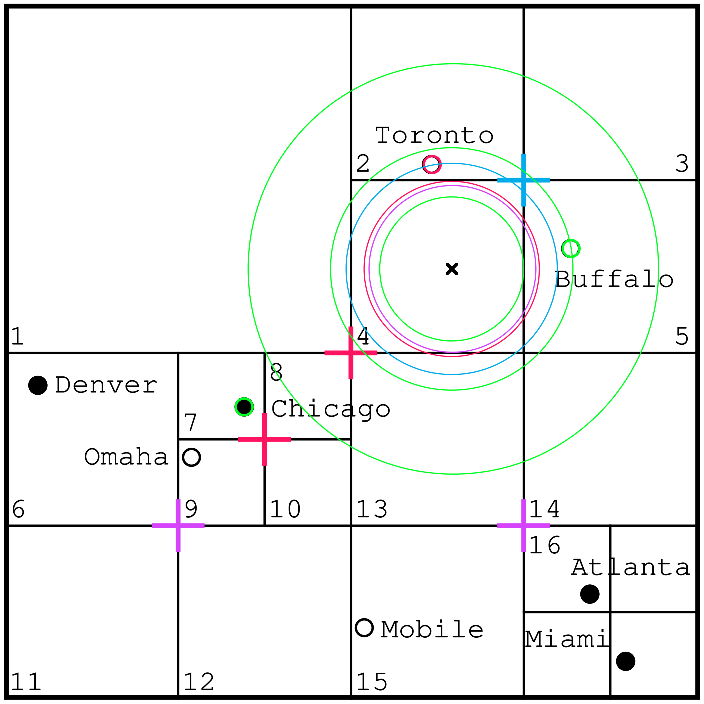
- Blocks labeled “depth/ NW-most descendant”
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn’t satisfy query

	City	Pop (1000)	Pos
	Atlanta	4,129	(85,15)
2	Buffalo	764	(82,65)
	Chicago	6,532	(35,42)
	Denver	1,381	(5,45)
	Mobile	504	(52,10)
	Omaha	416	(27,35)
1	Toronto	904	(62,77)
	Miami	5,250	(90,5)

1. Initially, queue only contains root: [0/1]
2. Dequeue root and enqueue 4 sons: [1/2 1/13 1/1 1/6]
3. Dequeue 1/2 and enqueue 4 sons: [2/4 2/5 1/13 2/2 1/1 2/3 1/6]
4. Dequeue 2/4 (empty); dequeue 2/5 containing Buffalo; enqueue Buffalo: [1/13 2/2 1/1 2/3 Buffalo 1/6]
5. Dequeue 1/13 and enqueue 4 sons: [2/13 2/2 1/1 2/14 2/3 Buffalo 1/6 2/15 2/16]
6. Dequeue 2/13 (empty); dequeue 2/2 containing Toronto; enqueue Toronto: [1/1 Toronto 2/14 2/3 Buffalo 1/6 2/15 2/16]
7. Dequeue 1/1 which is empty; dequeue Toronto (population too small); dequeue 2/14 and 2/3 (empty): [Buffalo 1/6 2/15 2/16]
8. Dequeue Buffalo (population too small): [1/6 2/15 2/16]
9. Dequeue 1/6 and enqueue 4 sons: [2/7 2/15 2/16 2/12 2/6 2/11]
10. Dequeue 2/7 and enqueue 4 sons: [3/8 3/10 3/7 3/9 2/15 2/16 2/12 2/6 2/11]

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



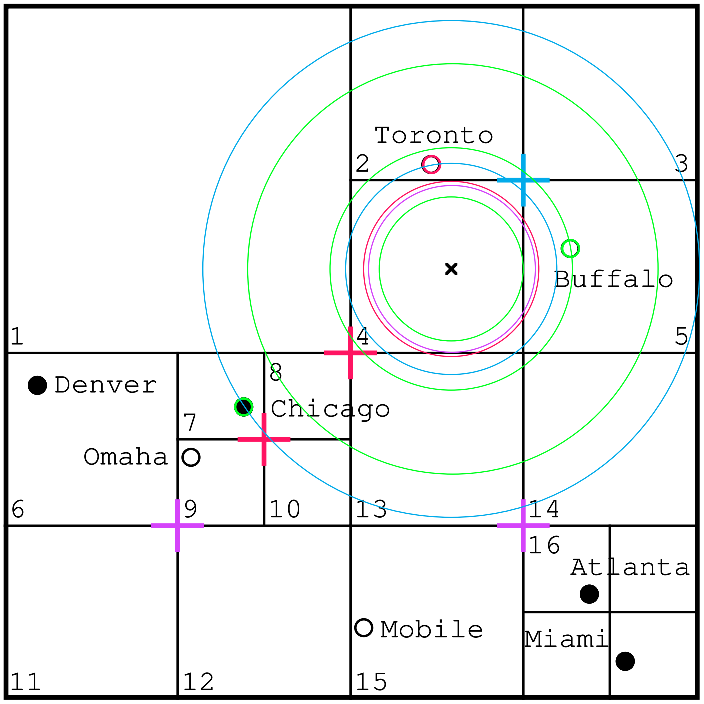
- Blocks labeled “depth/ NW-most descendant”
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn’t satisfy query

	City	Pop (1000)	Pos
	Atlanta	4,129	(85,15)
2	Buffalo	764	(82,65)
	Chicago	6,532	(35,42)
	Denver	1,381	(5,45)
	Mobile	504	(52,10)
	Omaha	416	(27,35)
1	Toronto	904	(62,77)
	Miami	5,250	(90,5)

1. Initially, queue only contains root: [0/1]
2. Dequeue root and enqueue 4 sons: [1/2 1/13 1/1 1/6]
3. Dequeue 1/2 and enqueue 4 sons: [2/4 2/5 1/13 2/2 1/1 2/3 1/6]
4. Dequeue 2/4 (empty); dequeue 2/5 containing Buffalo; enqueue Buffalo: [1/13 2/2 1/1 2/3 Buffalo 1/6]
5. Dequeue 1/13 and enqueue 4 sons: [2/13 2/2 1/1 2/14 2/3 Buffalo 1/6 2/15 2/16]
6. Dequeue 2/13 (empty); dequeue 2/2 containing Toronto; enqueue Toronto: [1/1 Toronto 2/14 2/3 Buffalo 1/6 2/15 2/16]
7. Dequeue 1/1 which is empty; dequeue Toronto (population too small); dequeue 2/14 and 2/3 (empty): [Buffalo 1/6 2/15 2/16]
8. Dequeue Buffalo (population too small): [1/6 2/15 2/16]
9. Dequeue 1/6 and enqueue 4 sons: [2/7 2/15 2/16 2/12 2/6 2/11]
10. Dequeue 2/7 and enqueue 4 sons: [3/8 3/10 3/7 3/9 2/15 2/16 2/12 2/6 2/11]
11. Dequeue 3/8 and 3/10 (empty); dequeue 3/7 containing Chicago; enqueue Chicago: [Chicago 3/9 2/15 2/16 2/12 2/6 2/11]

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



- Blocks labeled “depth/ NW-most descendant”
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn’t satisfy query

	City	Pop (1000)	Pos
	Atlanta	4,129	(85,15)
2	Buffalo	764	(82,65)
3	Chicago	6,532	(35,42)
	Denver	1,381	(5,45)
	Mobile	504	(52,10)
	Omaha	416	(27,35)
1	Toronto	904	(62,77)
	Miami	5,250	(90,5)

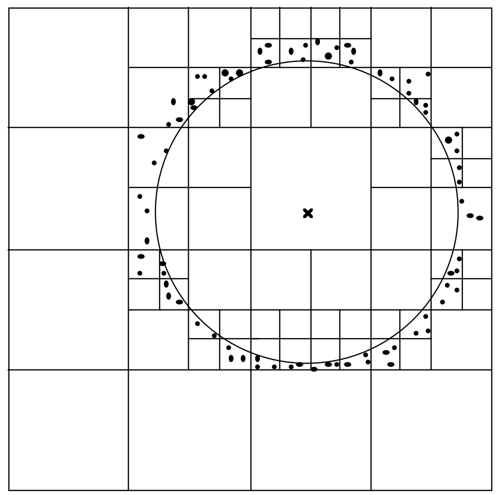
1. Initially, queue only contains root: [0/1]
2. Dequeue root and enqueue 4 sons: [1/2 1/13 1/1 1/6]
3. Dequeue 1/2 and enqueue 4 sons: [2/4 2/5 1/13 2/2 1/1 2/3 1/6]
4. Dequeue 2/4 (empty); dequeue 2/5 containing Buffalo; enqueue Buffalo: [1/13 2/2 1/1 2/3 Buffalo 1/6]
5. Dequeue 1/13 and enqueue 4 sons: [2/13 2/2 1/1 2/14 2/3 Buffalo 1/6 2/15 2/16]
6. Dequeue 2/13 (empty); dequeue 2/2 containing Toronto; enqueue Toronto: [1/1 Toronto 2/14 2/3 Buffalo 1/6 2/15 2/16]
7. Dequeue 1/1 which is empty; dequeue Toronto (population too small); dequeue 2/14 and 2/3 (empty): [Buffalo 1/6 2/15 2/16]
8. Dequeue Buffalo (population too small): [1/6 2/15 2/16]
9. Dequeue 1/6 and enqueue 4 sons: [2/7 2/15 2/16 2/12 2/6 2/11]
10. Dequeue 2/7 and enqueue 4 sons: [3/8 3/10 3/7 3/9 2/15 2/16 2/12 2/6 2/11]
11. Dequeue 3/8 and 3/10 (empty); dequeue 3/7 containing Chicago; enqueue Chicago: [Chicago 3/9 2/15 2/16 2/12 2/6 2/11]
12. Dequeue Chicago which satisfies the query

COMPARISON WITH OTHER WORK

- Not more efficient than similar methods of Arya/Mount and Hoel/Samet
- Works with arbitrary data structures including R-trees
- Arya/Mount make use of a k-d tree along with a priority queue but the results of their complexity analysis only hold for an approximate answer
- Unlike Arya/Mount we are not restricted to point features
- Compare with algorithm of Roussopoulos/Kelley/Vincent
  1. restricted to R-trees
  2. restricted to points as query objects
  3. *k* is fixed a priori
     - need to restart algorithm if want the *k*+1 nearest neighbors
     - maybe don't need all *k* neighbors
- Algorithm of Hoel/Samet is only good for finding nearest neighbor

ANALYSIS

1. Assume constant time to calculate distance metric
   - true for point query objects but not necessarily for more complex features such as polygons
2. Assume a PMR quadtree
   - for line data, $O(N)$ blocks for N lines
3. Assume spatial index exists and thus ignore cost of building it
4. Worst case queue size arises when:



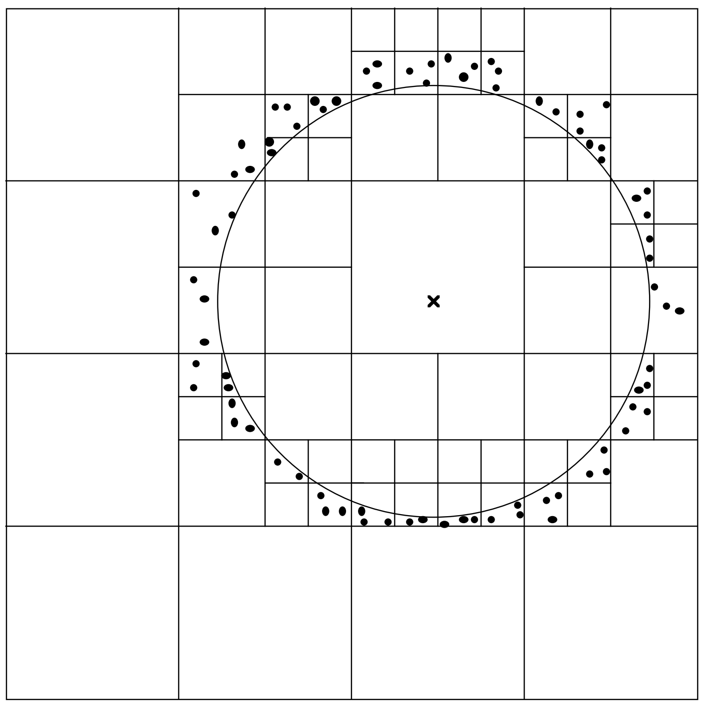
   - all features in queue are at a distance of at least d from q, and
   - all leaf blocks containing the features are at a distance less than d from q
   - implies all features are inserted into the queue before finding the nearest one — i.e., $O(N)$ space
   - if must rank all features, then $O(N \log M)$ execution time
     a. M is maximum queue size
     b. $O(N \log N)$ worst-case time which compares favorably with one-dimensional sorting
   - circular feature configuration is only bad if query object is at the center of the circle
   - highly unlikely for both the features to be in a circle and the query object to be at its center

ALTERNATIVE SOLUTIONS TO RANKING PROBLEM

1. Compute distance of all features from query object using conventional sorting techniques
   - $O(N \log N)$
   - our approach is better as don't have to retrieve all features at once
   - our approach is dynamic
   - we can usually do better than $O(N \log N)$
     a. we usually base the sort on the container blocks instead of the features
     b. important in a disk-based environment as usually we just need to look at the location and size of the container blocks rather than inspecting their contents which may require a disk access
2. Instead of enqueueing empty blocks, only insert nonempty blocks
   - costly as requires a disk access to inspect contents
   - our algorithm often avoids inspecting empty blocks as they get pruned by being too far away from the query object
   - however, if a total ranking, then it may be better to inspect blocks before enqueueing

CONCLUSION

1. For partial ranking, we visit a minimum number of container blocks in the sense that:
   - assume k^{th} nearest neighbor is at distance d_k from q
   - only examine contents of container blocks within a distance d_k from q
   - however, all of the container blocks could be within d_k of q so that the algorithm still takes the same amount of time as a total ranking which is $O(N \log N)$



2. Algorithm can be applied to other spatial indexes as long as they decompose the space into blocks that are organized using a containment hierarchy
3. Analysis was for a PMR quadtree and may differ for other spatial indexes