HIERARCHICAL REPRESENTATIONS OF POINT DATA

Hanan Samet
Computer Science Department and
Institute for Advanced Computer Studies and
Center for Automation Research
University of Maryland

College Park, MD 20742 USA
e-mail: hjs@umiacs.umd.edu

PRELIMINARIES

- File ≡ collection of records (N)
- Each record contains several attributes or keys (k)

Queries:

1. Point query
2. Range query (includes partial match)
3. Boolean query ≡ combine 1 and 2 with AND, OR, NOT

Search methods

1. Organize data to be stored
   - boundaries of regions in the search space are determined by the data
   - e.g., binary search tree
2. Organize the embedding space from which the data is drawn
   - region boundaries in the search space are fixed
   - e.g., address computation methods such as digital searching
3. Hybrid
   - use 1 for some attributes and 2 for others

Extreme solution:

- Bitmap representation where one bit is reserved for every possible record in the multidimensional point space whether or not it is present
- Problems:
  1. large number of attributes
  2. continuous data (non-discrete)

SIMPLE NON-HIERARCHICAL DATA STRUCTURES

1. Sequential list

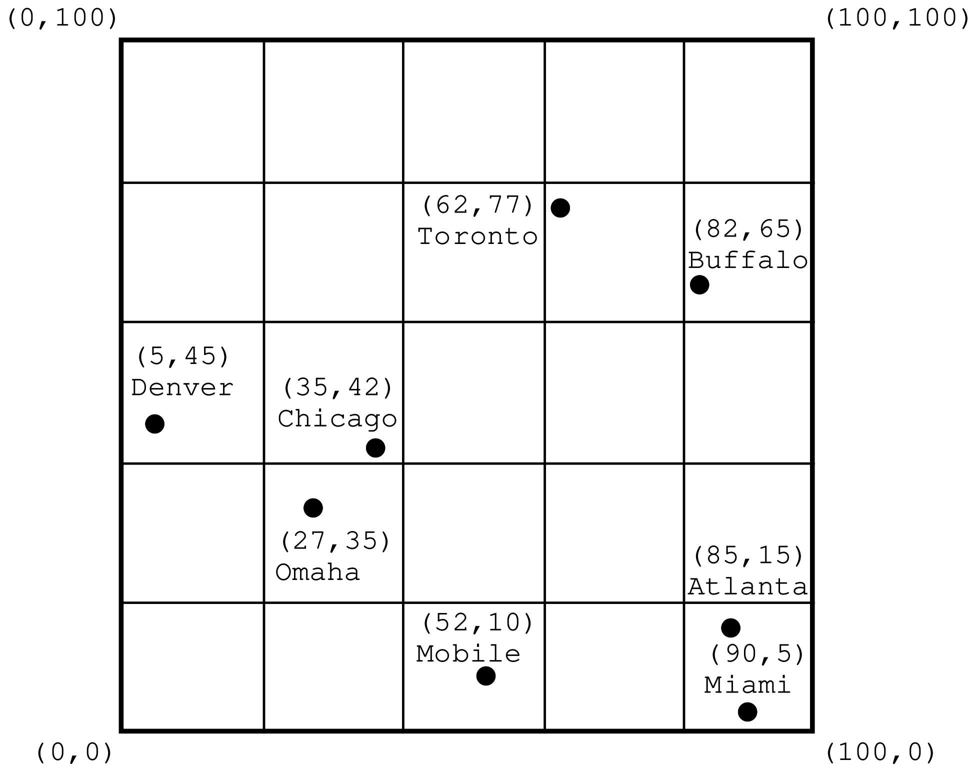
NAME	X	Y
Chicago	35	42
Mobile	52	10
Toronto	62	77
Buffalo	82	65
Denver	5	45
Omaha	27	35
Atlanta	85	15
Miami	90	5

2. Inverted List

X	Y
Denver	Miami
Omaha	Mobile
Chicago	Atlanta
Mobile	Omaha
Toronto	Chicago
Buffalo	Denver
Atlanta	Buffalo
Miami	Toronto

- 2 sorted lists
- Data is pointers
- Enables pruning the search with respect to one key

GRID METHOD



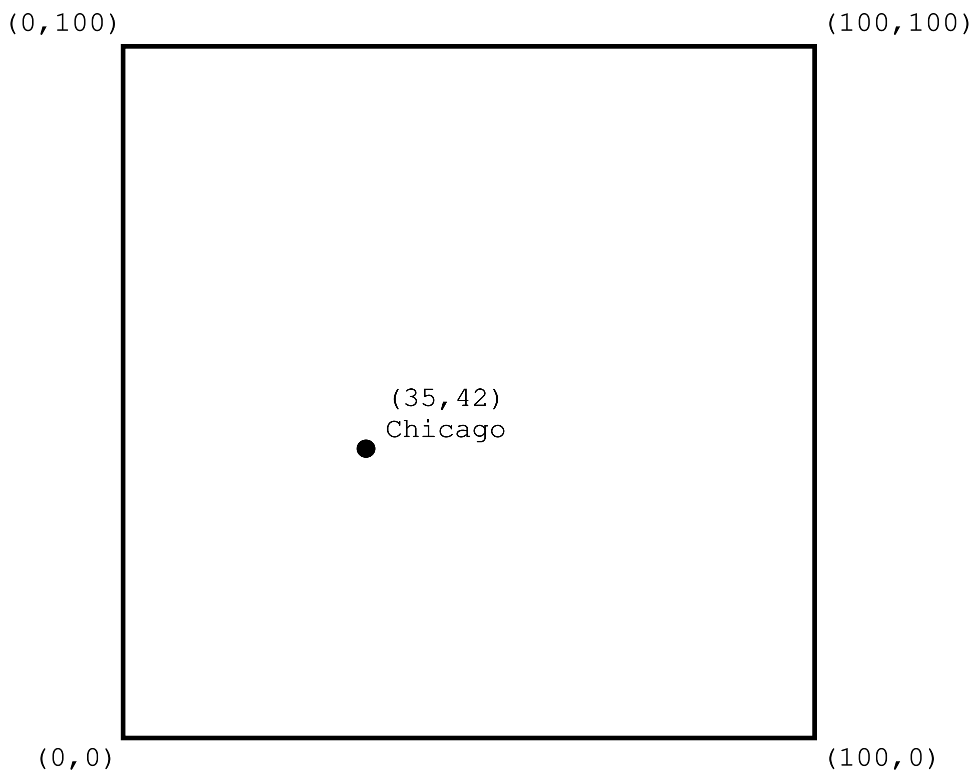
- Divide space into squares of width equal to the search region
- Each cell contains a list of all points within it
- Assume L_∞ distance metric (i.e., chessboard)
- Assume C = uniform distribution of points per cell
- Average search time for k-dimensional space is $O(F \cdot 2^k)$

  F = number of records found = C since query region has the width of a cell

  2^k = number of cells examined

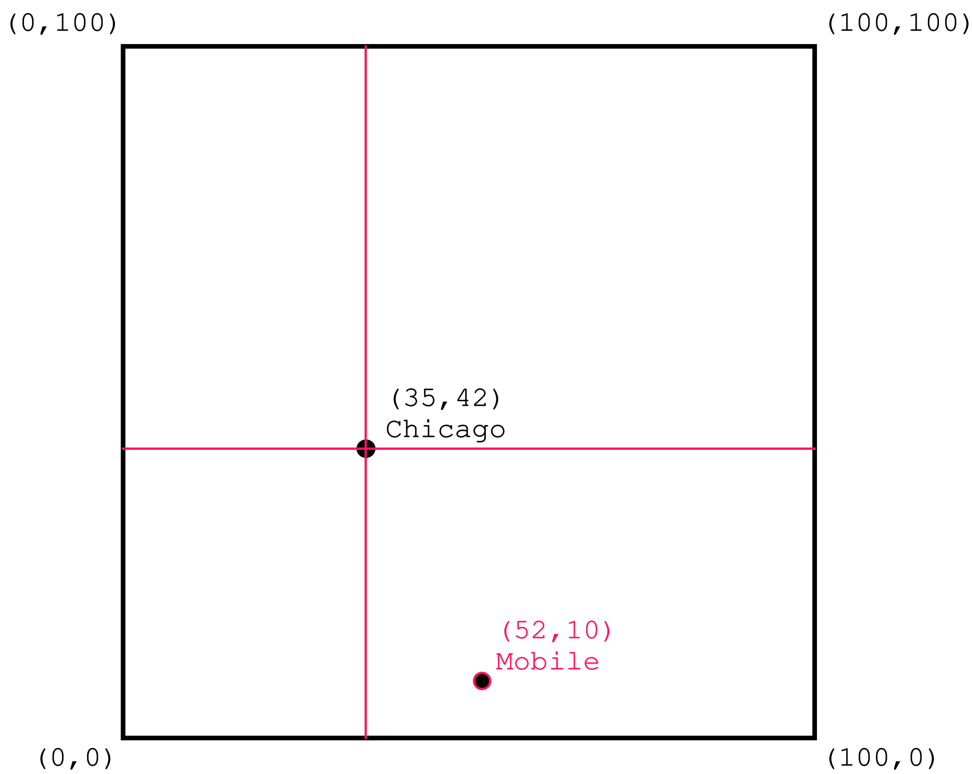
POINT QUADTREE (Finkel/Bentley)

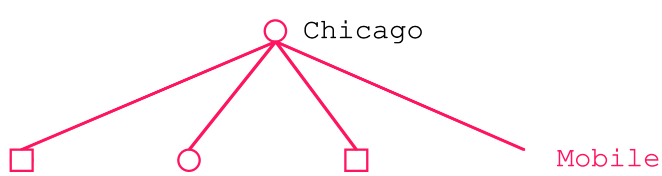
- Marriage between a uniform grid and a binary search tree



POINT QUADTREE (Finkel/Bentley)

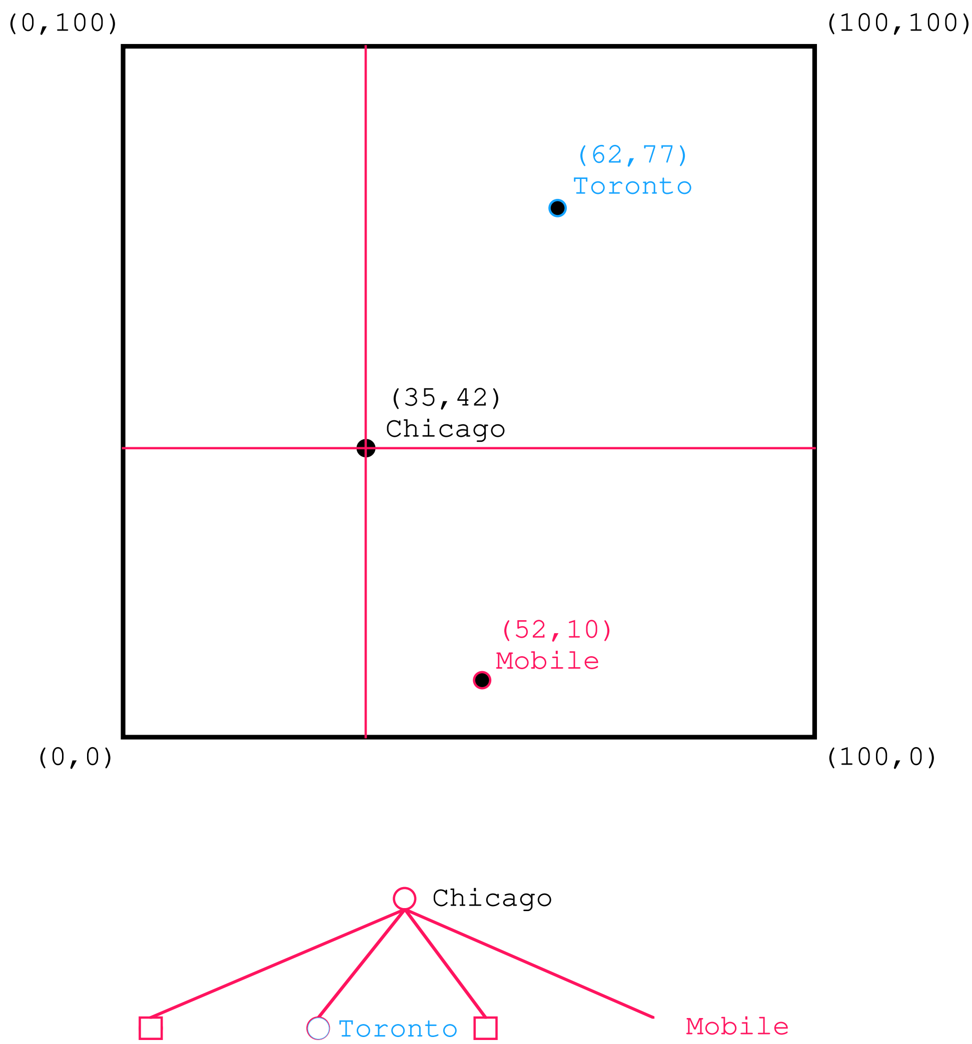
- Marriage between a uniform grid and a binary search tree





POINT QUADTREE (Finkel/Bentley)

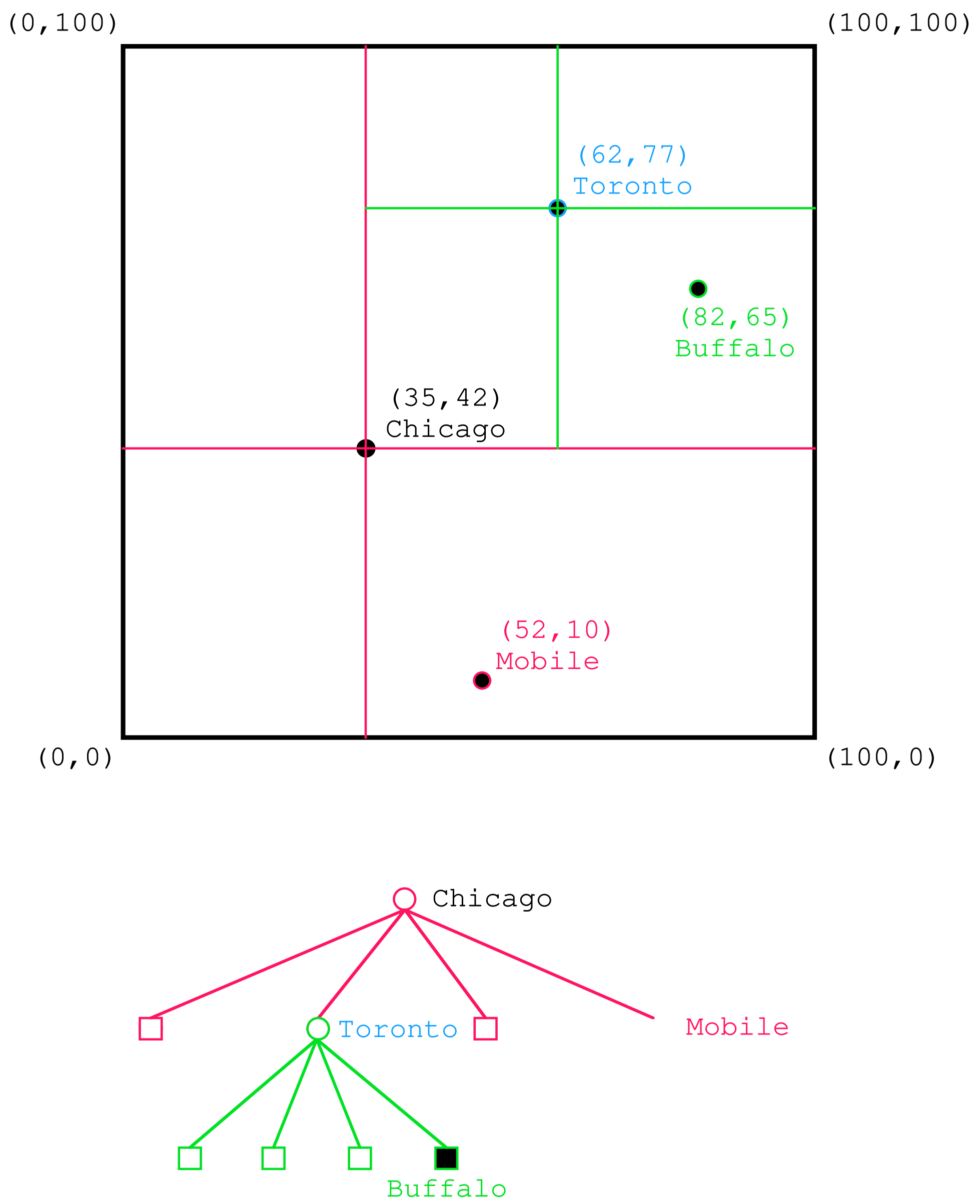
- Marriage between a uniform grid and a binary search tree



POINT QUADTREE (Finkel/Bentley)

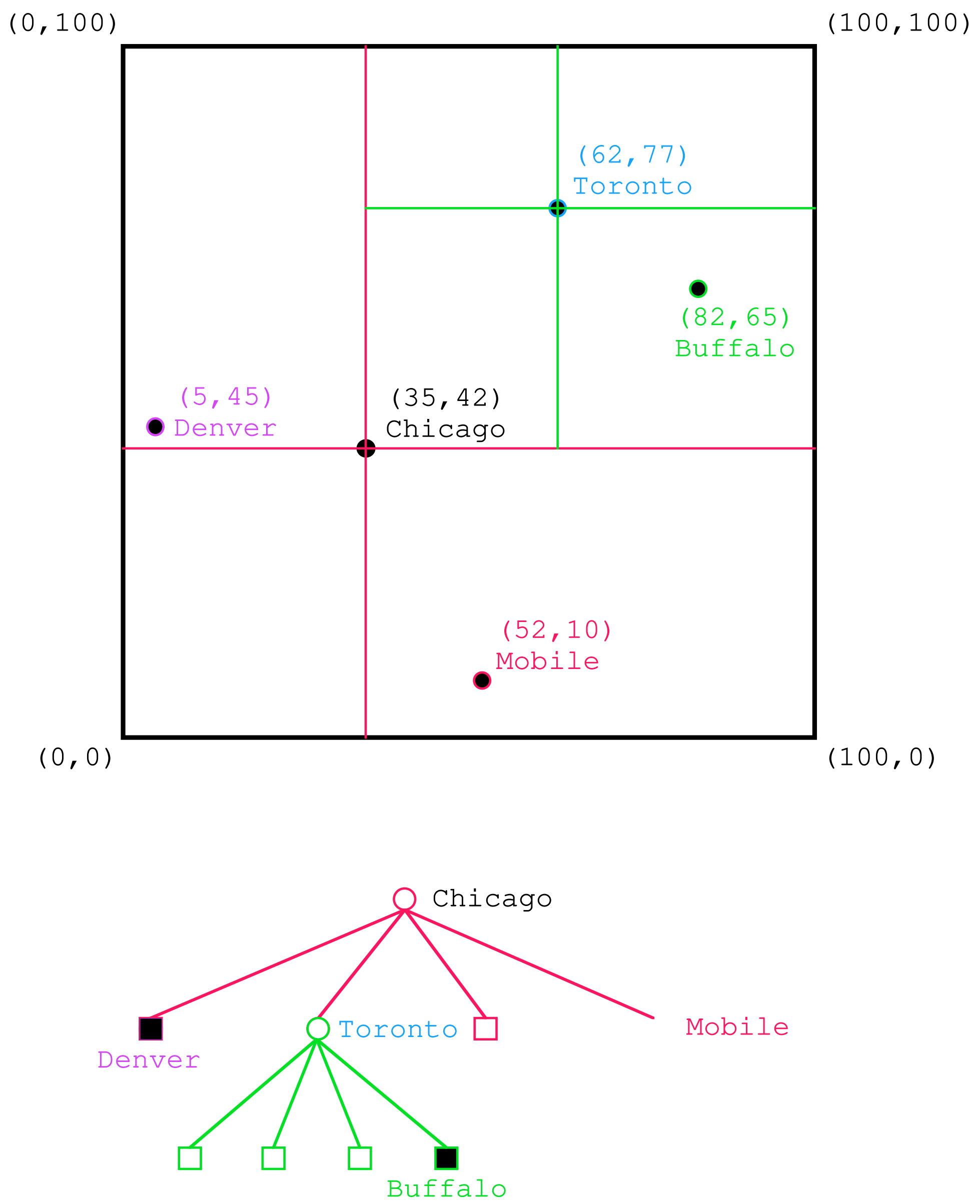
4 3 2 1 hp4
g z r b

- Marriage between a uniform grid and a binary search tree



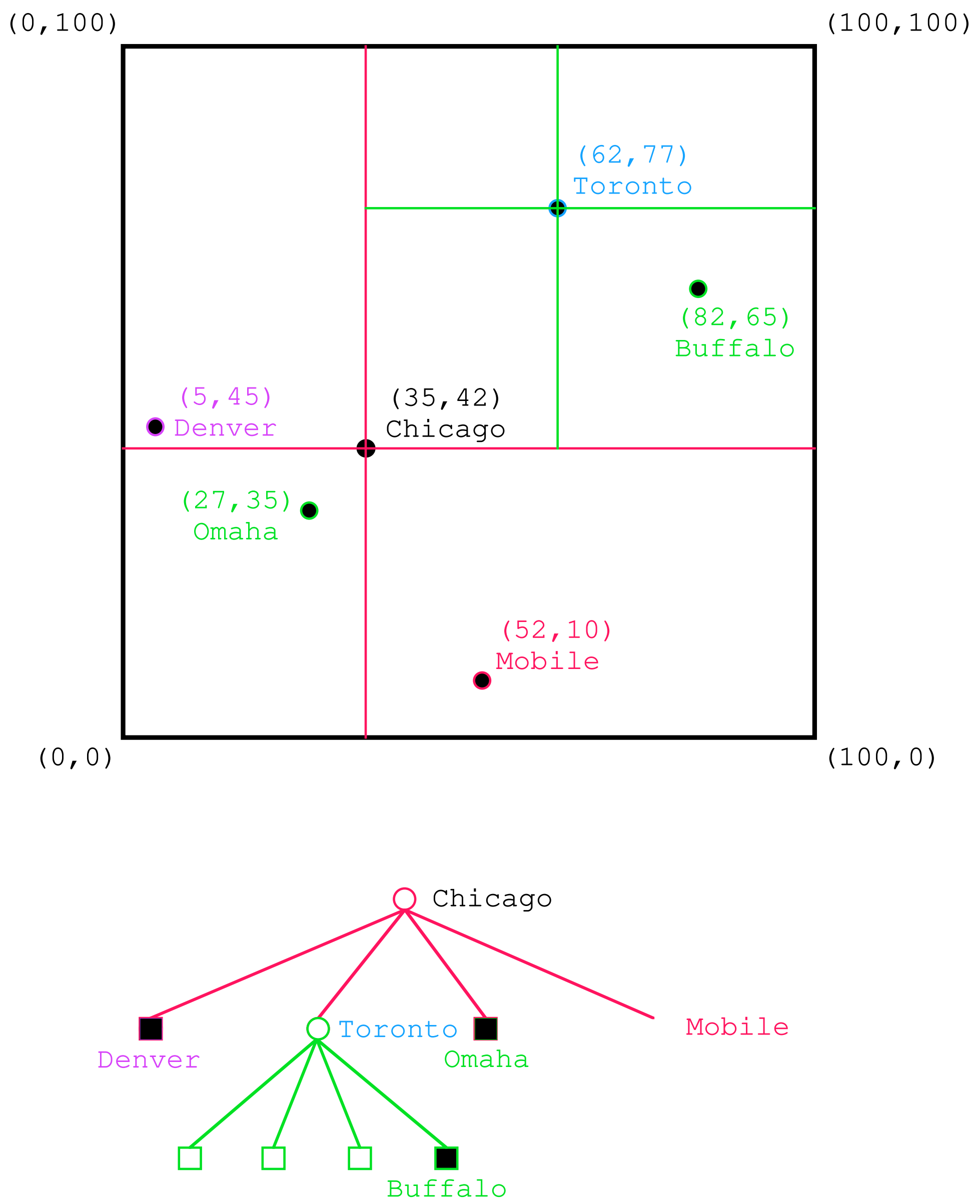
POINT QUADTREE (Finkel/Bentley)

- Marriage between a uniform grid and a binary search tree



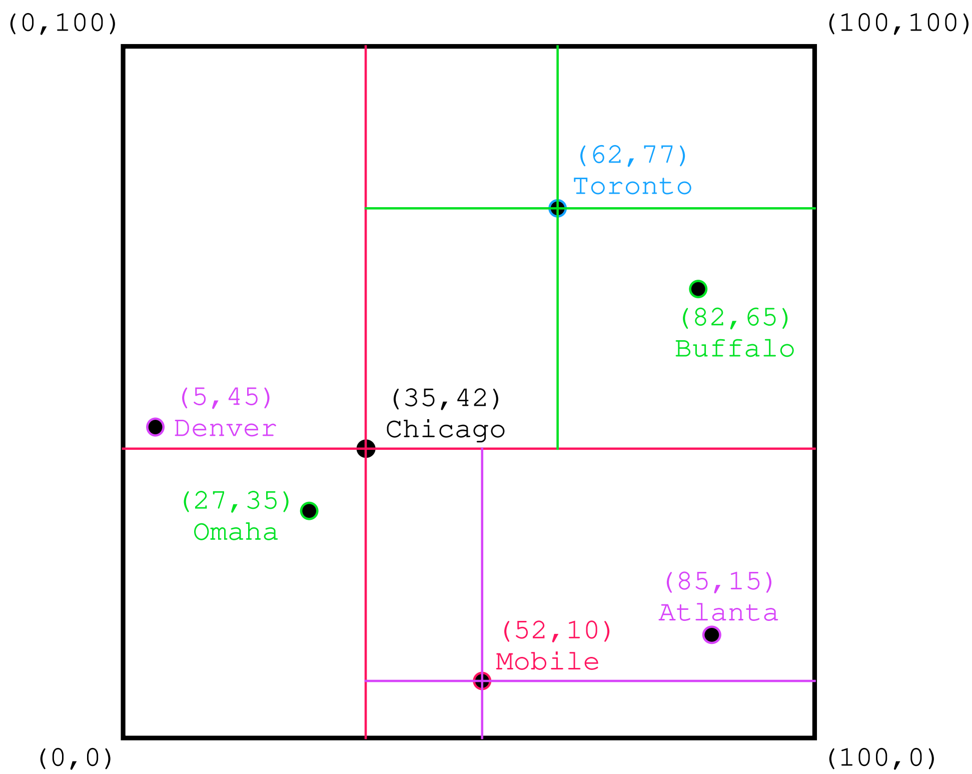
POINT QUADTREE (Finkel/Bentley)

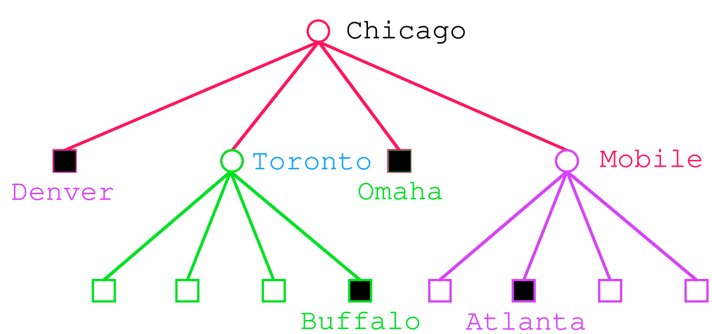
- Marriage between a uniform grid and a binary search tree



POINT QUADTREE (Finkel/Bentley)

- Marriage between a uniform grid and a binary search tree

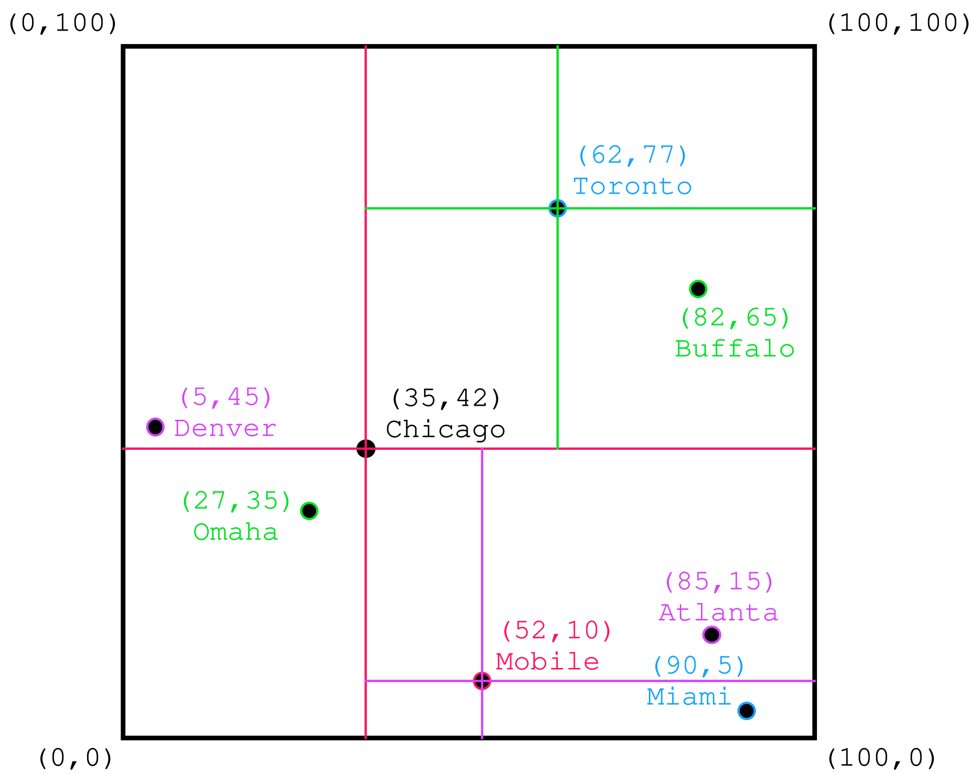


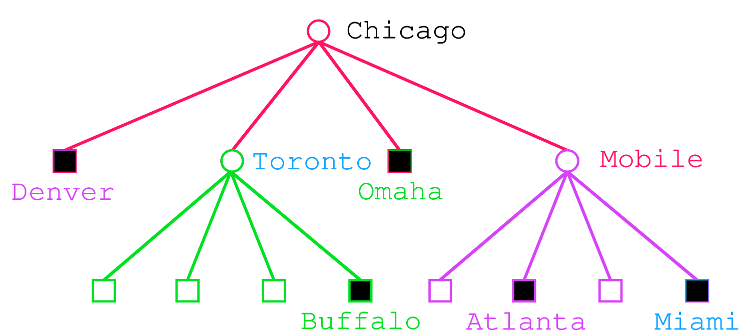


8 7 6 5 4 3 2 1
z v g v g z r b
hp4

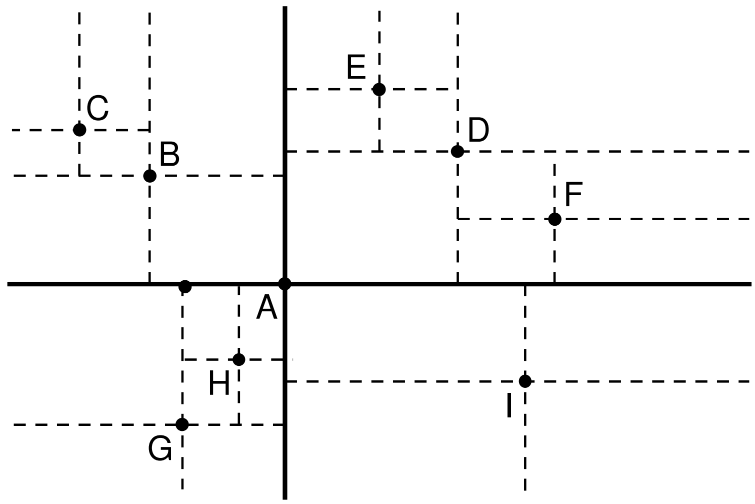
POINT QUADTREE (Finkel/Bentley)

- Marriage between a uniform grid and a binary search tree



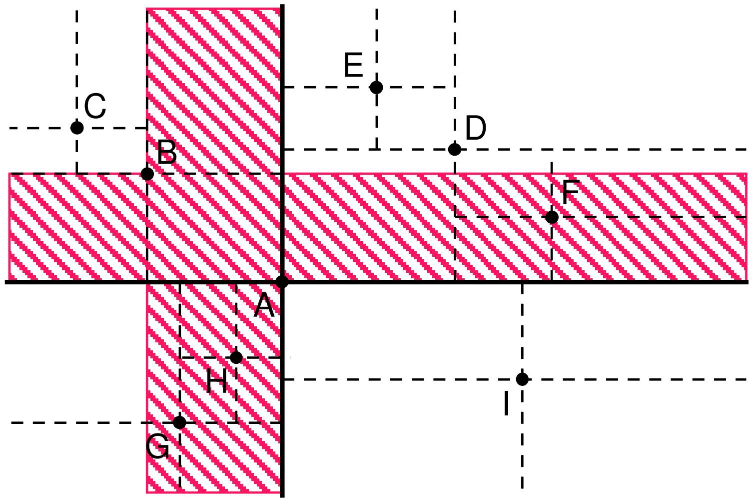


PROBLEM OF DELETION IN POINT QUADTREES



- Delete node A
- Conventional algorithm takes one son as the new root and reinserts the remaining subtrees

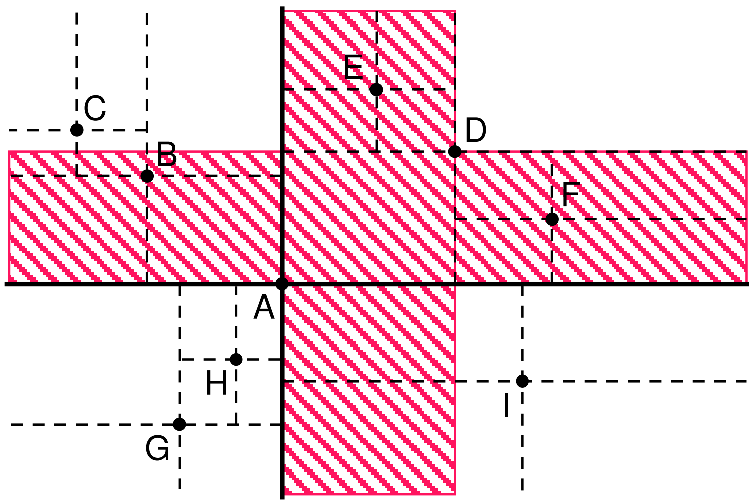
PROBLEM OF DELETION IN POINT QUADTREES



- Delete node A
- Conventional algorithm takes one son as the new root and reinserts the remaining subtrees

  1. B is the new root

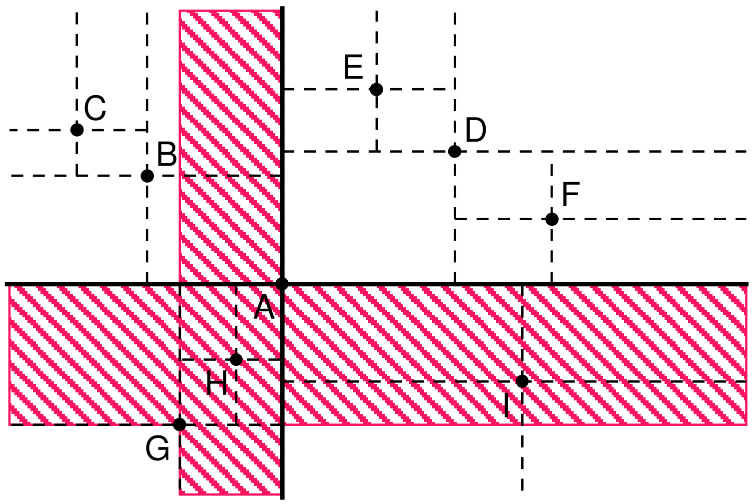
PROBLEM OF DELETION IN POINT QUADTREES



- Delete node A
- Conventional algorithm takes one son as the new root and reinserts the remaining subtrees

2. D is the new root

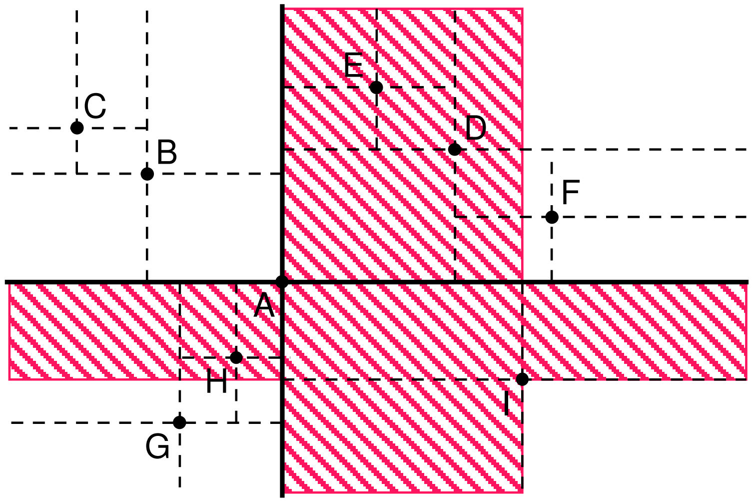
PROBLEM OF DELETION IN POINT QUADTREES



- Delete node A
- Conventional algorithm takes one son as the new root and reinserts the remaining subtrees

3. G is the new root

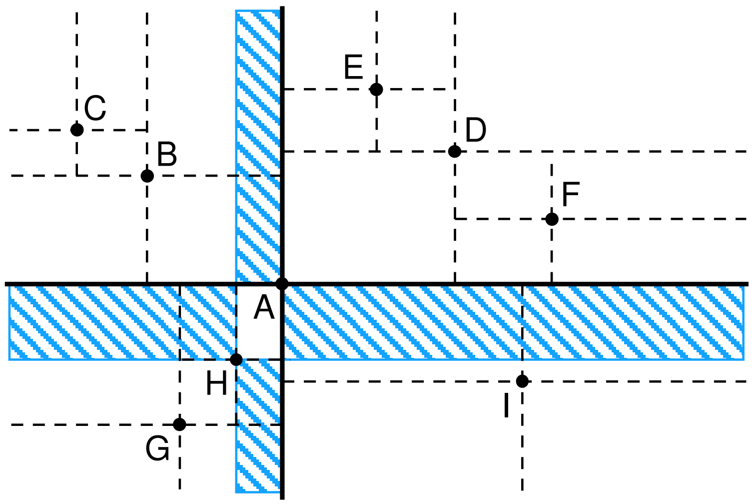
PROBLEM OF DELETION IN POINT QUADTREES



- Delete node A
- Conventional algorithm takes one son as the new root and reinserts the remaining subtrees

4. I is the new root

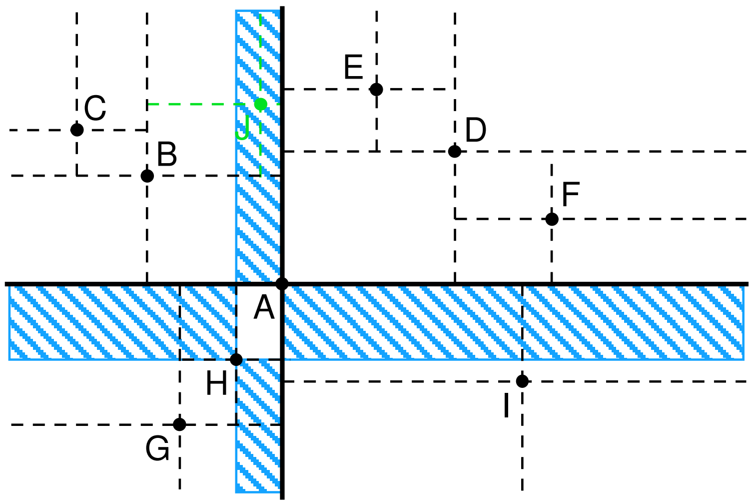
PROBLEM OF DELETION IN POINT QUADTREES



- Delete node A
- Conventional algorithm takes one son as the new root and reinserts the remaining subtrees

- Optimal solution is to use H as the new root since the shaded region is empty

PROBLEM OF DELETION IN POINT QUADTREES

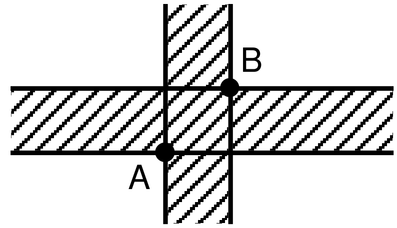


- Delete node A
- Conventional algorithm takes one son as the new root and reinserts the remaining subtrees

- Optimal solution is to use H as the new root since the shaded region is empty
- Problems:
  1. must search for H
  2. a node such as H may possibly not exist as is the case if node J is present

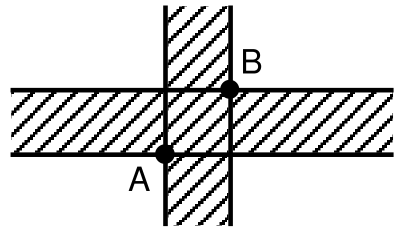
MECHANICS OF DELETION IN POINT QUADTREES

- Ideally, want to replace deleted node (A) with a node (B) that leaves an empty shaded region

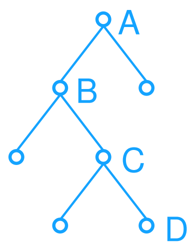


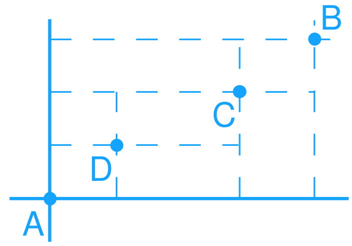
MECHANICS OF DELETION IN POINT QUADTREES

- Ideally, want to replace deleted node (A) with a node (B) that leaves an empty shaded region



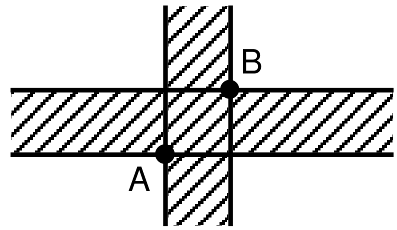
- Involves search and instead settle on a set of candidate nodes obtained using a method analogous to searching a binary search tree



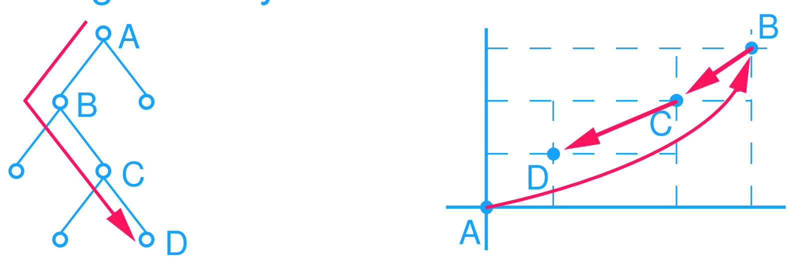


MECHANICS OF DELETION IN POINT QUADTREES

- Ideally, want to replace deleted node (A) with a node (B) that leaves an empty shaded region



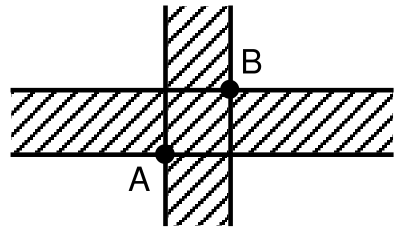
- Involves search and instead settle on a set of candidate nodes obtained using a method analogous to searching a binary search tree



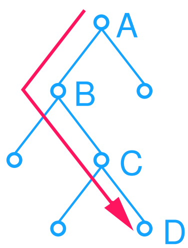
D is the closest node

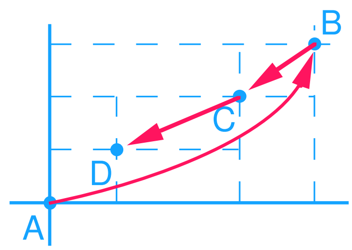
MECHANICS OF DELETION IN POINT QUADTREES

- Ideally, want to replace deleted node (A) with a node (B) that leaves an empty shaded region



- Involves search and instead settle on a set of candidate nodes obtained using a method analogous to searching a binary search tree



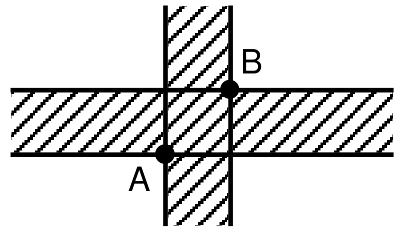


D is the closest node

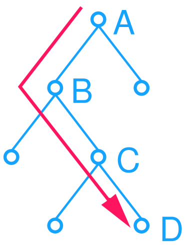
- Set of candidates = "closest" node in each quadrant
  1. choose the candidate node that is closer to each of its bordering axes than any other candidate node which is on the same side of those axes

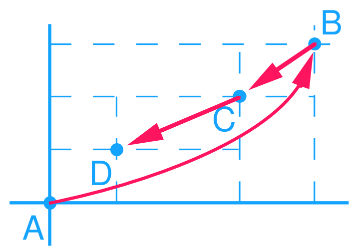
MECHANICS OF DELETION IN POINT QUADTREES

- Ideally, want to replace deleted node (A) with a node (B) that leaves an empty shaded region



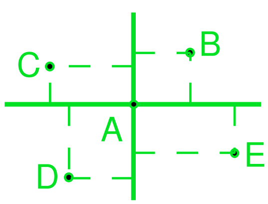
- Involves search and instead settle on a set of candidate nodes obtained using a method analogous to searching a binary search tree





D is the closest node

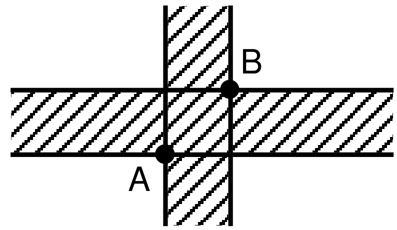
- Set of candidates = "closest" node in each quadrant
  1. choose the candidate node that is closer to each of its bordering axes than any other candidate node which is on the same side of those axes



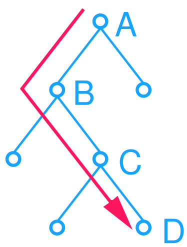
condition does not always hold

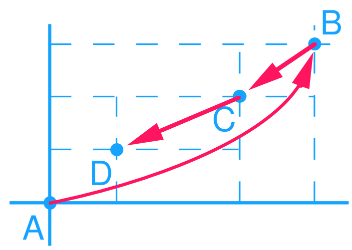
MECHANICS OF DELETION IN POINT QUADTREES

- Ideally, want to replace deleted node (A) with a node (B) that leaves an empty shaded region



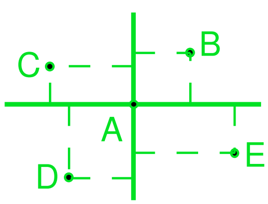
- Involves search and instead settle on a set of candidate nodes obtained using a method analogous to searching a binary search tree



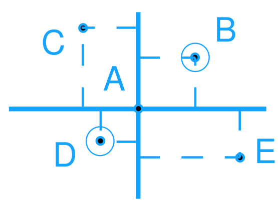


D is the closest node

- Set of candidates = "closest" node in each quadrant
  1. choose the candidate node that is closer to each of its bordering axes than any other candidate node which is on the same side of those axes



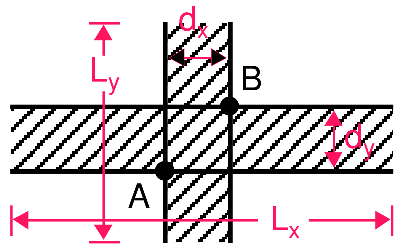
condition does not always hold



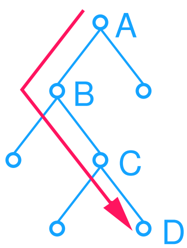
more than one node satisfies it

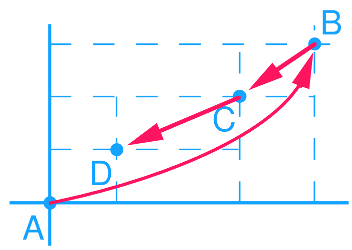
MECHANICS OF DELETION IN POINT QUADTREES

- Ideally, want to replace deleted node (A) with a node (B) that leaves an empty shaded region



- Involves search and instead settle on a set of candidate nodes obtained using a method analogous to searching a binary search tree

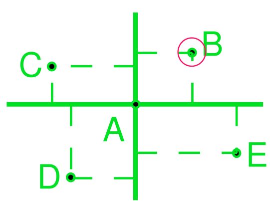


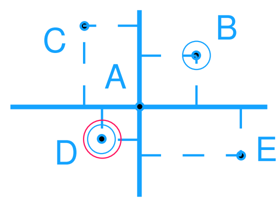


D is the closest node

- Set of candidates = "closest" node in each quadrant

1. choose the candidate node that is closer to each of its bordering axes than any other candidate node which is on the same side of those axes



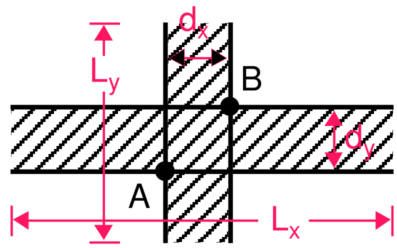


condition does not always hold | more than one node satisfies it

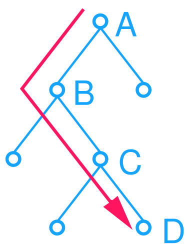
2. Use the L1 metric to break ties or deadlocks
   - L1 metric is the sum of the displacements from the bordering *x* and *y* axes
   - rationale: area of shaded region is $L_x \cdot d_y + L_y \cdot d_x - d_x \cdot d_y$ which can be approximated by $2d_x \cdot (L_x + L_y)$ assuming $d_x \approxeq d_y$ and d_x being very small

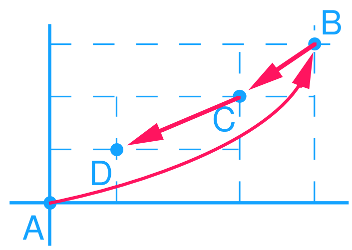
MECHANICS OF DELETION IN POINT QUADTREES

- Ideally, want to replace deleted node (A) with a node (B) that leaves an empty shaded region



- Involves search and instead settle on a set of candidate nodes obtained using a method analogous to searching a binary search tree

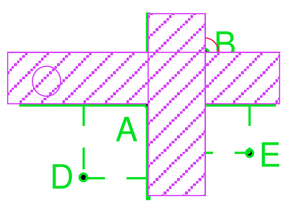




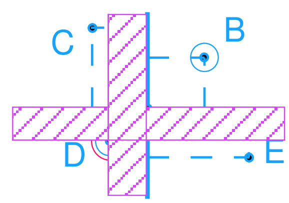
D is the closest node

- Set of candidates = "closest" node in each quadrant

1. choose the candidate node that is closer to each of its bordering axes than any other candidate node which is on the same side of those axes



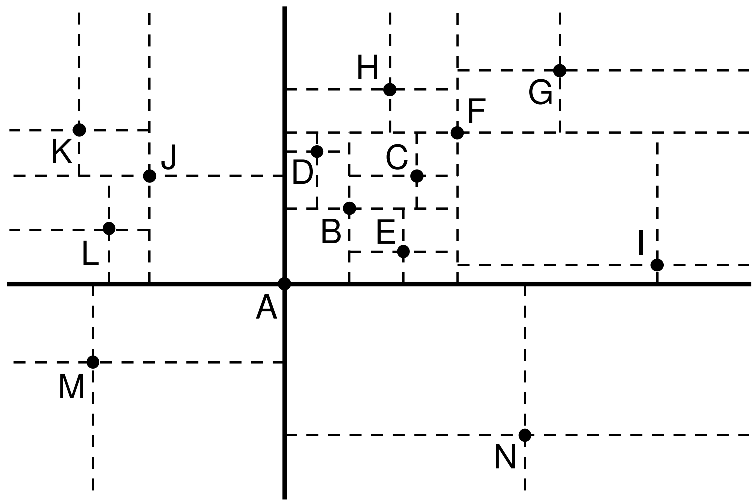
condition does not always hold



more than one node satisfies it

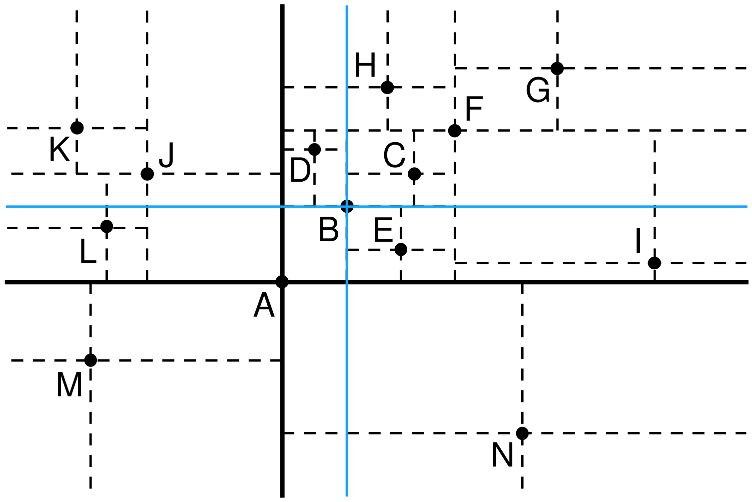
2. Use the L1 metric to break ties or deadlocks
   - L1 metric is the sum of the displacements from the bordering *x* and *y* axes
   - rationale: area of shaded region is $L_x \cdot d_y + L_y \cdot d_x - d_x \cdot d_y$ which can be approximated by $2d_x \cdot (L_x + L_y)$ assuming $d_x \cong d_y$ and d_x being very small
   - at most one of the remaining candidates will be in the shaded region

ALGORITHM FOR DELETION IN POINT QUADTREES



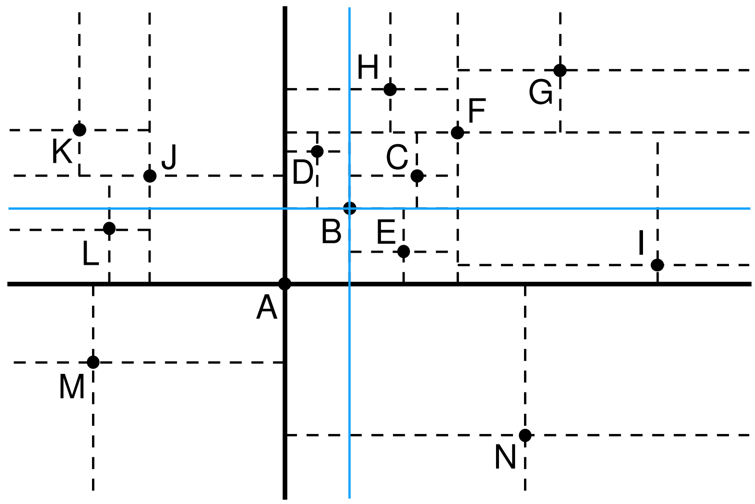
- Select a node satisfying the "closest" criteria to the deleted node A to serve as the new root (B in the NE quadrant

ALGORITHM FOR DELETION IN POINT QUADTREES



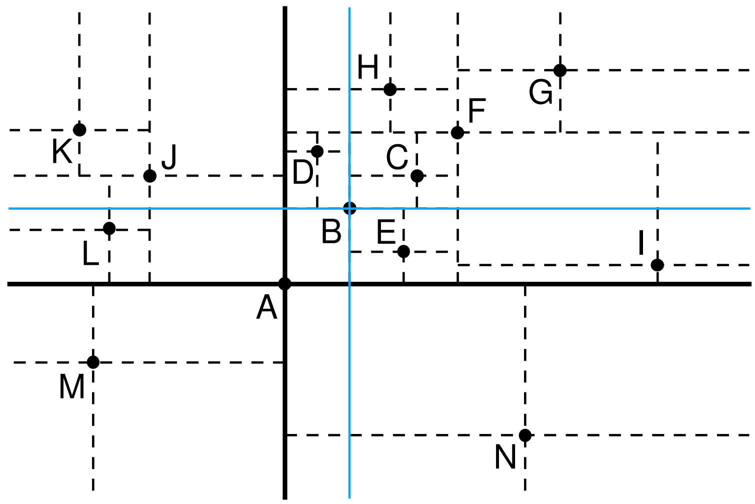
- Select a node satisfying the "closest" criteria to the deleted node A to serve as the new root (B in the NE quadrant

ALGORITHM FOR DELETION IN POINT QUADTREES



- Select a node satisfying the "closest" criteria to the deleted node A to serve as the new root (B in the NE quadrant
  1. no reinsertion in opposite quadrant (SW)

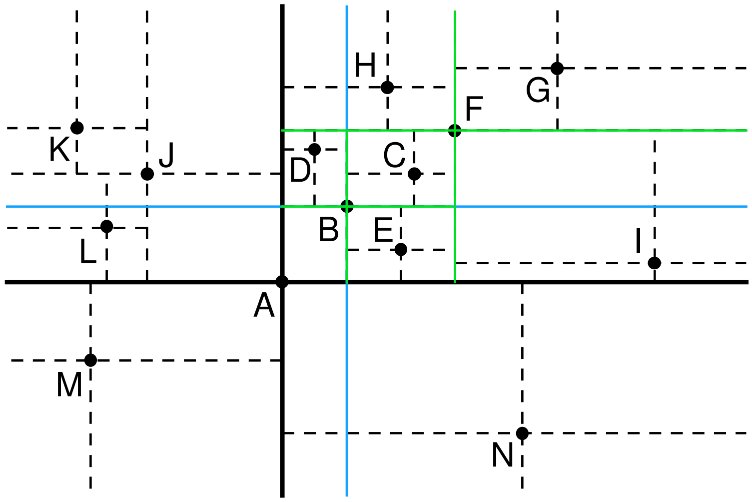
ALGORITHM FOR DELETION IN POINT QUADTREES



- Select a node satisfying the "closest" criteria to the deleted node A to serve as the new root (B in the NE quadrant
  1. no reinsertion in opposite quadrant (SW)
  2. ADJ: apply to adjacent quadrants (NW,SE):
     if root remains in the quadrant (J) then
     - no reinsertion in 2 subquadrants (NW,NE)
     - apply ADJ to remaining 2 subquadrants (SW,SE)

     else reinsert entire quadrant

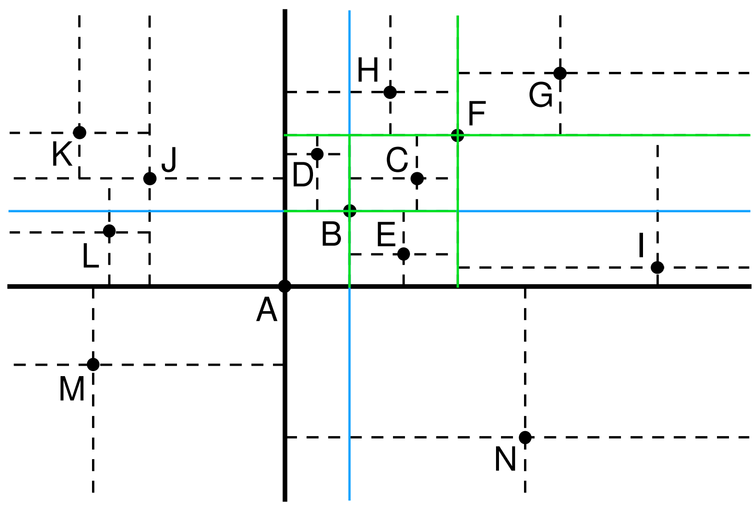
ALGORITHM FOR DELETION IN POINT QUADTREES



- Select a node satisfying the "closest" criteria to the deleted node A to serve as the new root (B in the NE quadrant
  1. no reinsertion in opposite quadrant (SW)
  2. ADJ: apply to adjacent quadrants (NW,SE):
     if root remains in the quadrant (J) then
     - no reinsertion in 2 subquadrants (NW,NE)
     - apply ADJ to remaining 2 subquadrants (SW,SE)

     else reinsert entire quadrant
  3. NEWROOT: apply to quadrant containing replacement node (NE)
     - same subquadrant (NE): no reinsertion
     - adjacent subquadrants (NW,SE): ADJ
     - opposite subquadrant (SW): NEWROOT

ALGORITHM FOR DELETION IN POINT QUADTREES

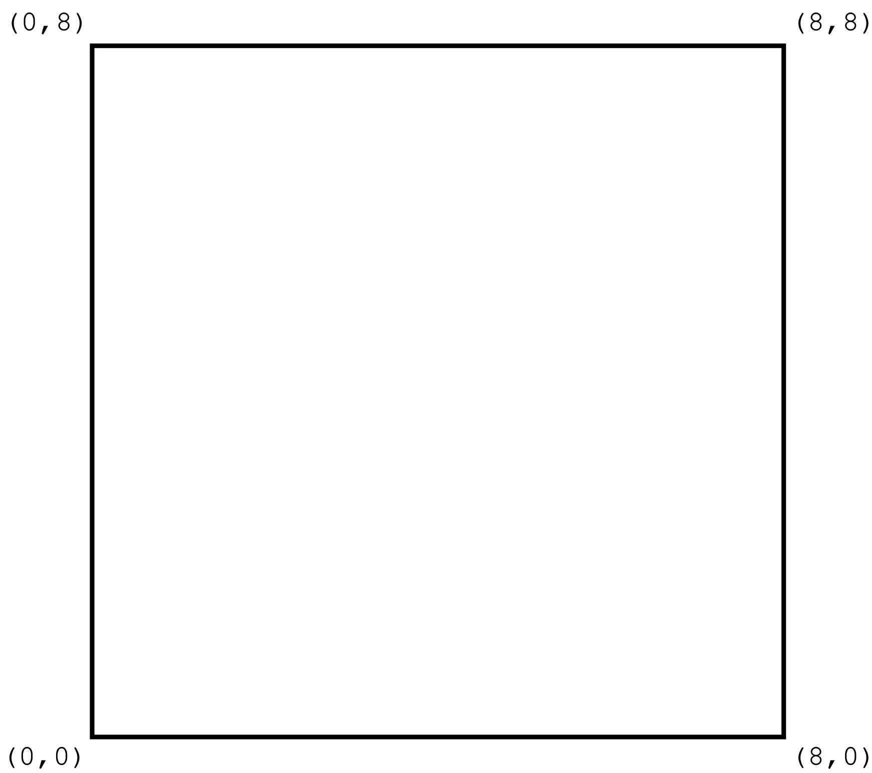


- Select a node satisfying the "closest" criteria to the deleted node A to serve as the new root (B in the NE quadrant
  1. no reinsertion in opposite quadrant (SW)
  2. ADJ: apply to adjacent quadrants (NW,SE):
     if root remains in the quadrant (J) then
     - no reinsertion in 2 subquadrants (NW,NE)
     - apply ADJ to remaining 2 subquadrants (SW,SE)

     else reinsert entire quadrant
  3. NEWROOT: apply to quadrant containing replacement node (NE)
     - same subquadrant (NE): no reinsertion
     - adjacent subquadrants (NW,SE): ADJ
     - opposite subquadrant (SW): NEWROOT
- Comparison with conventional reinsertion algorithm
  1. 5/6 reduction in number of nodes requiring reinsertion (2/3 if pick a candidate at random)
  2. number of comparisons during reinsertion was observed to be ~$\log_4 n$ vs. a much larger factor
  3. (average total path length)/(optimal total path length) was observed to be constant vs. an increase

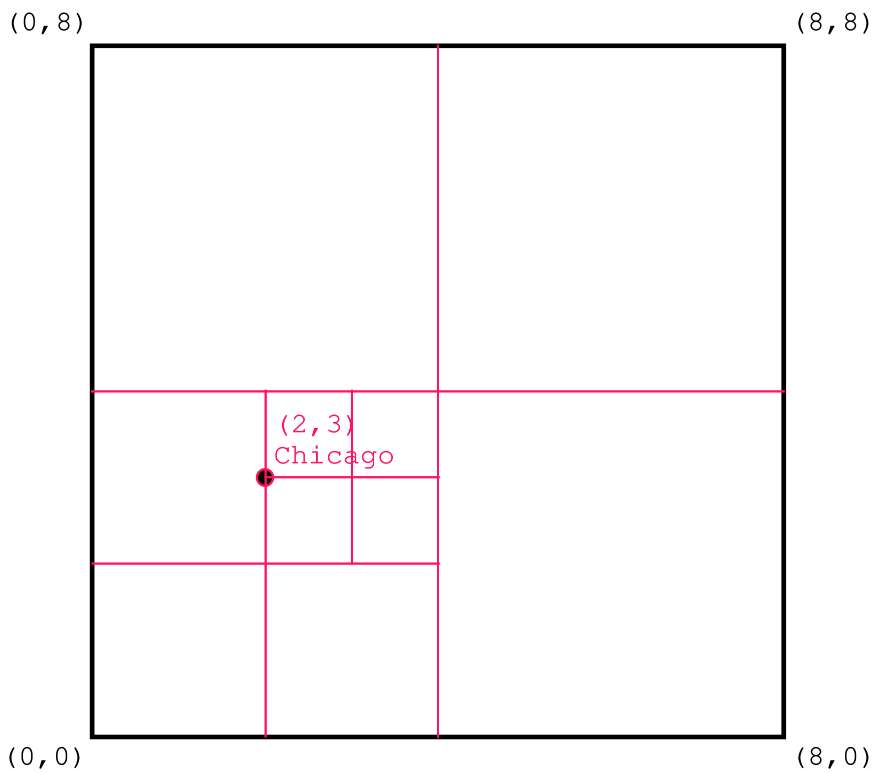
MX QUADTREE (Hunter)

- Points are like BLACK pixels in a region quadtree
- Useful for raster to vector conversion
- Empty cells are merged to form larger empty cells
- Only good for discrete data
- Good for sparse matrix applications
- Assume that the point is associated with the lower left corner of each cell
- Ex: assume an 8 x 8 array
  divide coordinate values by 12.5



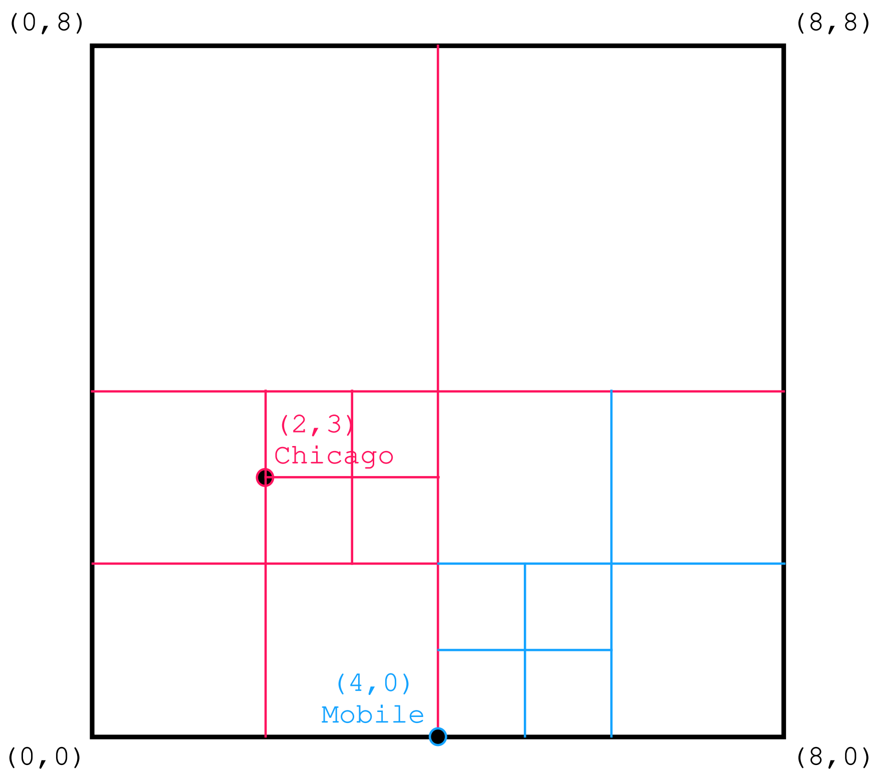
MX QUADTREE (Hunter)

- Points are like BLACK pixels in a region quadtree
- Useful for raster to vector conversion
- Empty cells are merged to form larger empty cells
- Only good for discrete data
- Good for sparse matrix applications
- Assume that the point is associated with the lower left corner of each cell
- Ex: assume an 8 x 8 array
  divide coordinate values by 12.5



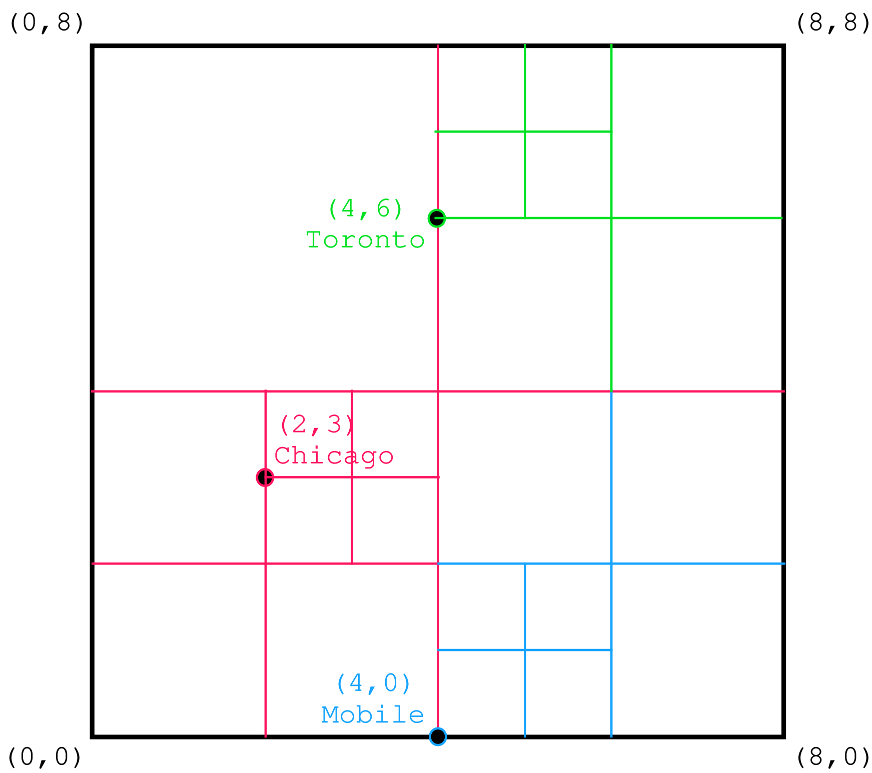
MX QUADTREE (Hunter)

- Points are like BLACK pixels in a region quadtree
- Useful for raster to vector conversion
- Empty cells are merged to form larger empty cells
- Only good for discrete data
- Good for sparse matrix applications
- Assume that the point is associated with the lower left corner of each cell
- Ex: assume an 8 x 8 array
  divide coordinate values by 12.5



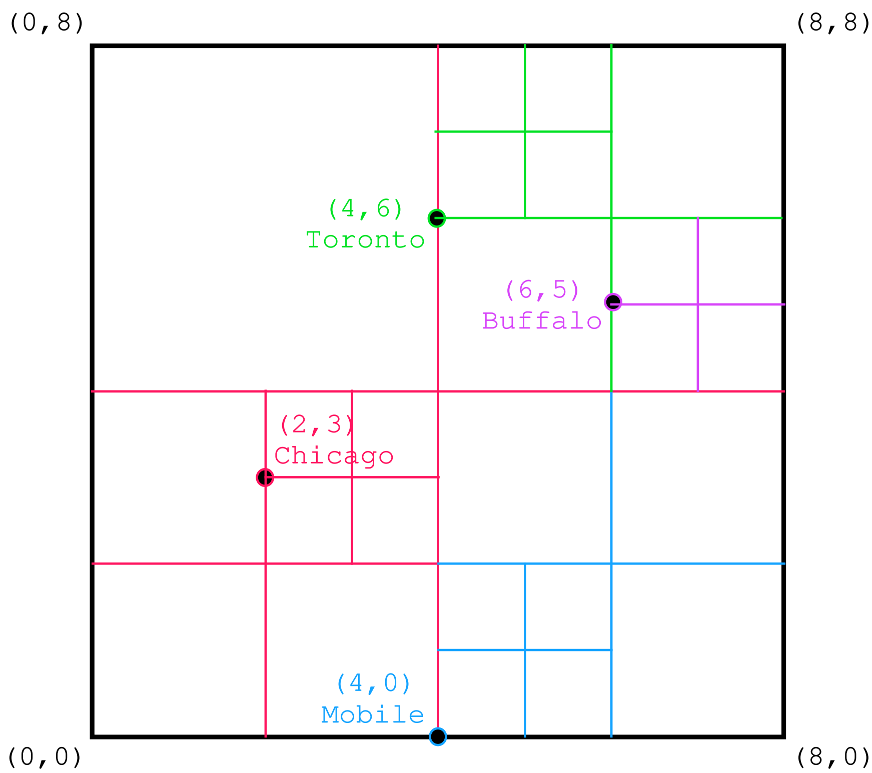
MX QUADTREE (Hunter)

- Points are like BLACK pixels in a region quadtree
- Useful for raster to vector conversion
- Empty cells are merged to form larger empty cells
- Only good for discrete data
- Good for sparse matrix applications
- Assume that the point is associated with the lower left corner of each cell
- Ex: assume an 8 x 8 array
  divide coordinate values by 12.5



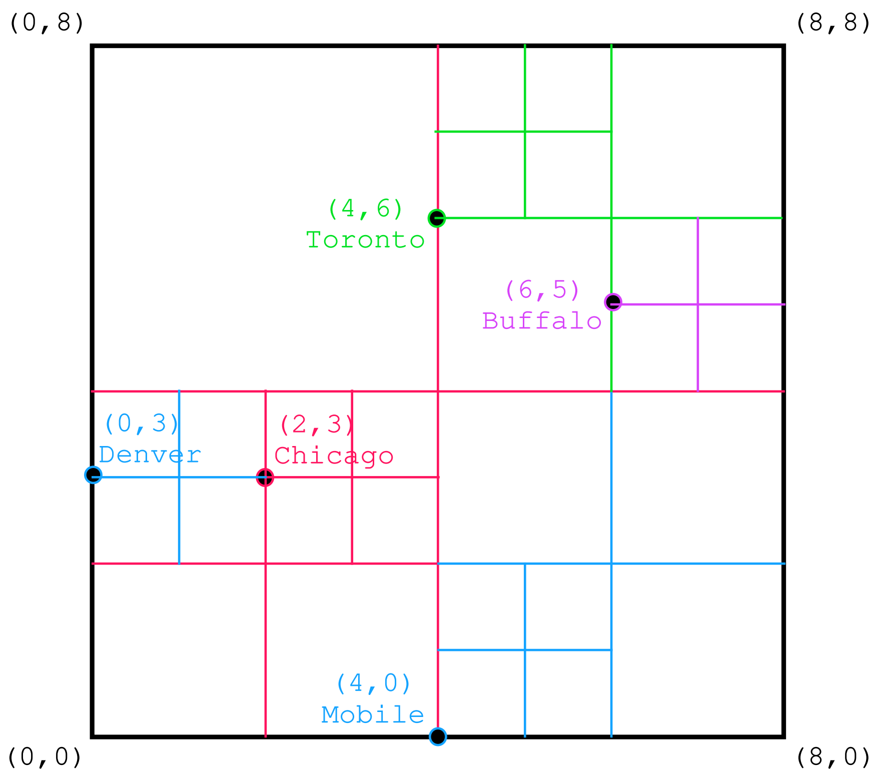
MX QUADTREE (Hunter)

- Points are like BLACK pixels in a region quadtree
- Useful for raster to vector conversion
- Empty cells are merged to form larger empty cells
- Only good for discrete data
- Good for sparse matrix applications
- Assume that the point is associated with the lower left corner of each cell
- Ex: assume an 8 x 8 array
  divide coordinate values by 12.5



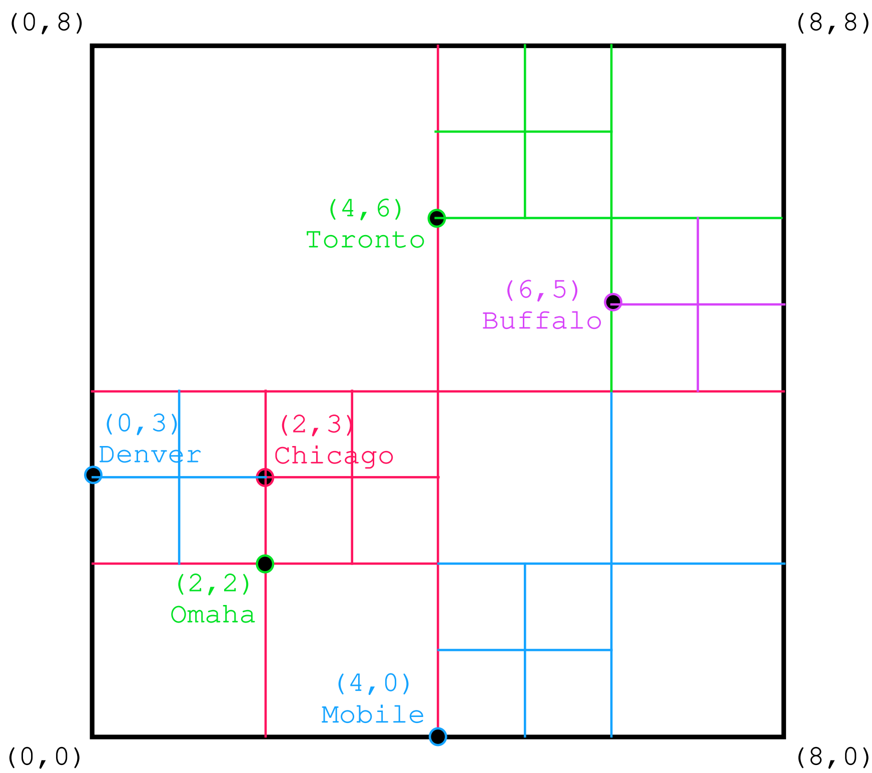
MX QUADTREE (Hunter)

- Points are like BLACK pixels in a region quadtree
- Useful for raster to vector conversion
- Empty cells are merged to form larger empty cells
- Only good for discrete data
- Good for sparse matrix applications
- Assume that the point is associated with the lower left corner of each cell
- Ex: assume an 8 x 8 array
  divide coordinate values by 12.5



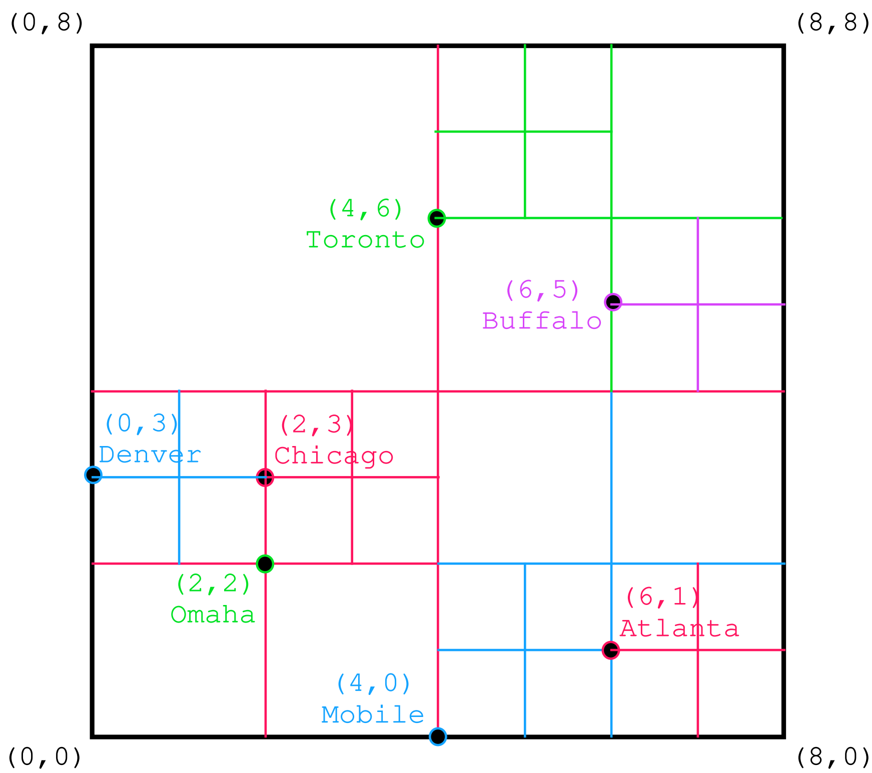
MX QUADTREE (Hunter)

- Points are like BLACK pixels in a region quadtree
- Useful for raster to vector conversion
- Empty cells are merged to form larger empty cells
- Only good for discrete data
- Good for sparse matrix applications
- Assume that the point is associated with the lower left corner of each cell
- Ex: assume an 8 x 8 array
  divide coordinate values by 12.5



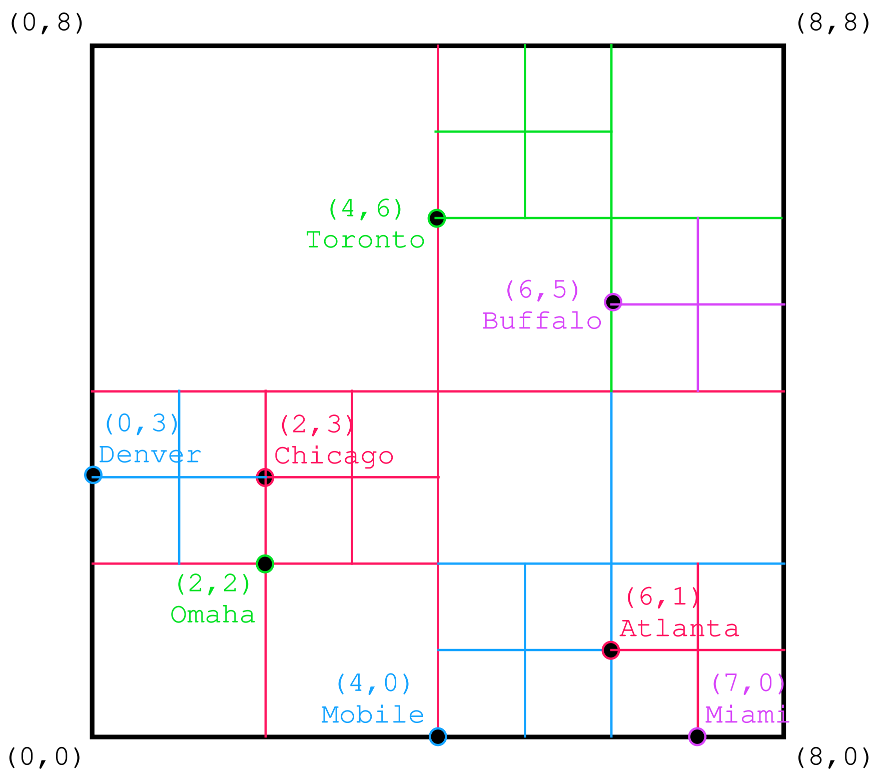
MX QUADTREE (Hunter)

- Points are like BLACK pixels in a region quadtree
- Useful for raster to vector conversion
- Empty cells are merged to form larger empty cells
- Only good for discrete data
- Good for sparse matrix applications
- Assume that the point is associated with the lower left corner of each cell
- Ex: assume an 8 x 8 array
  divide coordinate values by 12.5



MX QUADTREE (Hunter)

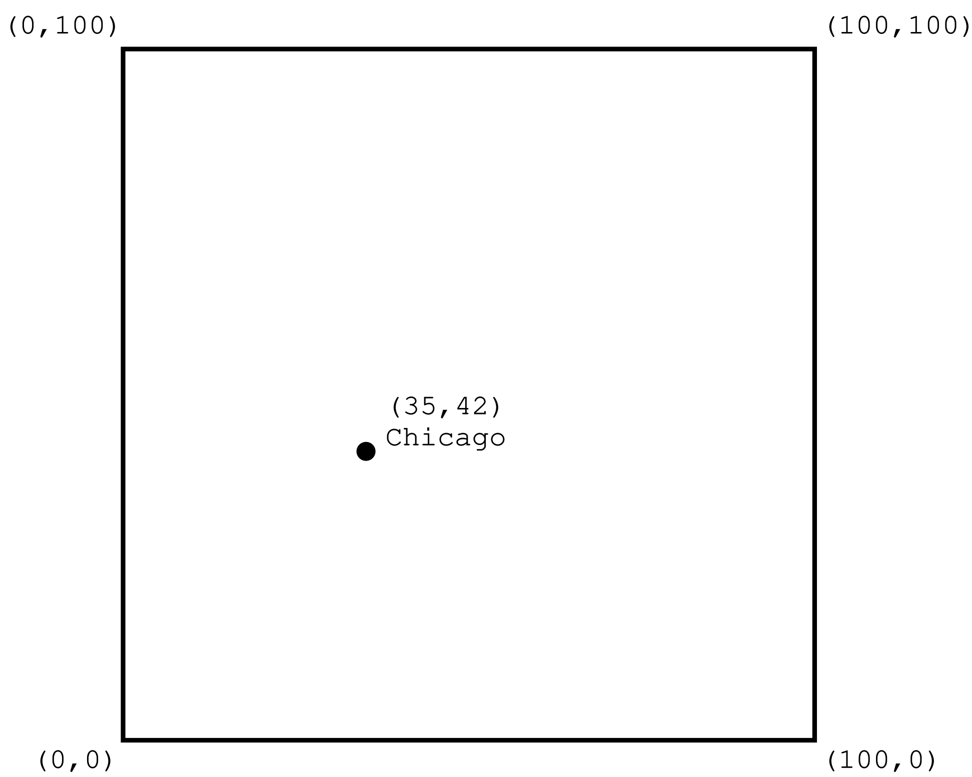
- Points are like BLACK pixels in a region quadtree
- Useful for raster to vector conversion
- Empty cells are merged to form larger empty cells
- Only good for discrete data
- Good for sparse matrix applications
- Assume that the point is associated with the lower left corner of each cell
- Ex: assume an 8 x 8 array
  divide coordinate values by 12.5



PR QUADTREE (Orenstein)

1. Regular decomposition point representation
2. Decomposition occurs whenever a block contains more than one point
3. Useful when the domain of data points is not discrete but finite
4. Maximum level of decomposition depends on the minimum separation between two points
   - if two points are very close, then decomposition can be very deep
   - can be overcome by viewing blocks as buckets with capacity *c* and only decomposing the block when it contains more than *c* points

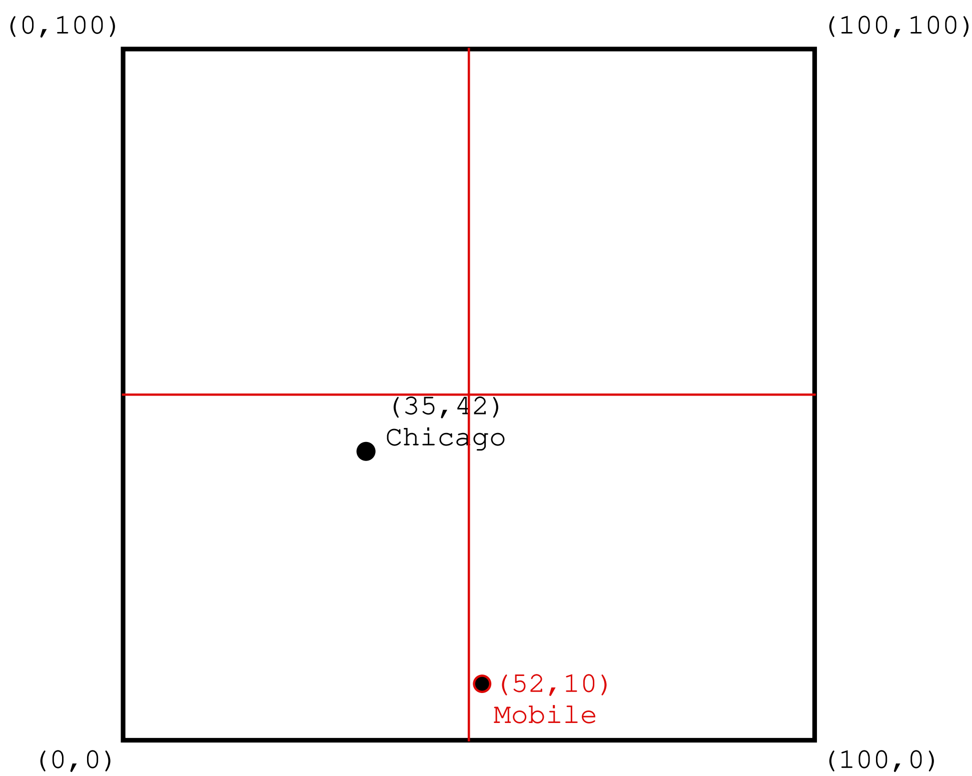
   Ex: $c = 1$



PR QUADTREE (Orenstein)

1. Regular decomposition point representation
2. Decomposition occurs whenever a block contains more than one point
3. Useful when the domain of data points is not discrete but finite
4. Maximum level of decomposition depends on the minimum separation between two points
   - if two points are very close, then decomposition can be very deep
   - can be overcome by viewing blocks as buckets with capacity *c* and only decomposing the block when it contains more than *c* points

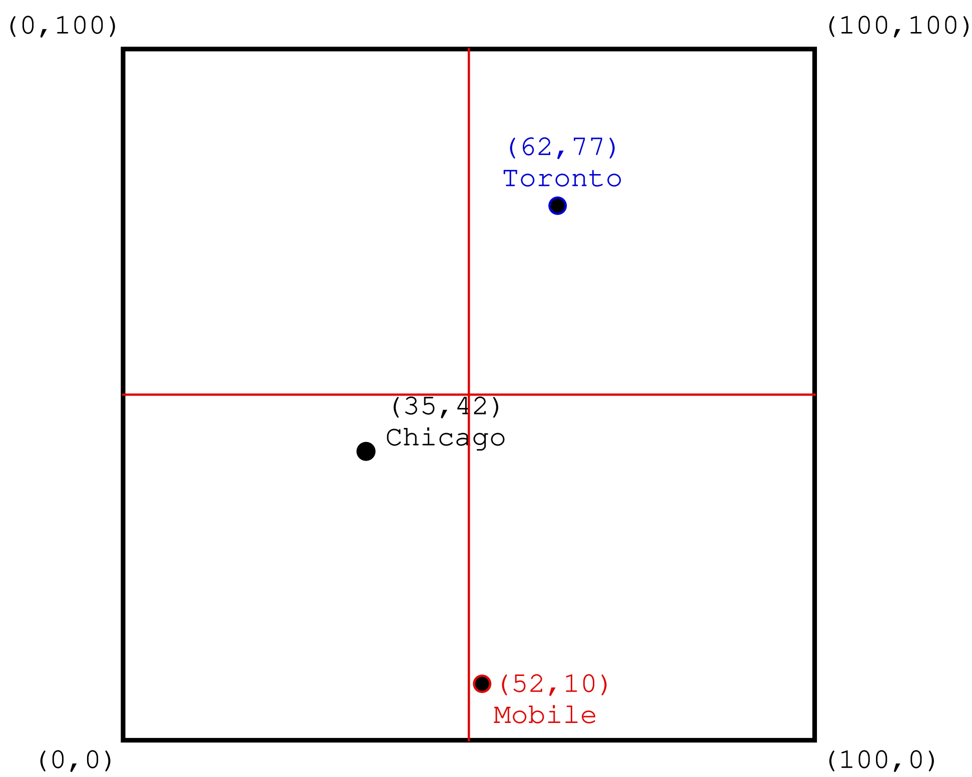
   Ex: $c = 1$



PR QUADTREE (Orenstein)

1. Regular decomposition point representation
2. Decomposition occurs whenever a block contains more than one point
3. Useful when the domain of data points is not discrete but finite
4. Maximum level of decomposition depends on the minimum separation between two points
   - if two points are very close, then decomposition can be very deep
   - can be overcome by viewing blocks as buckets with capacity *c* and only decomposing the block when it contains more than *c* points

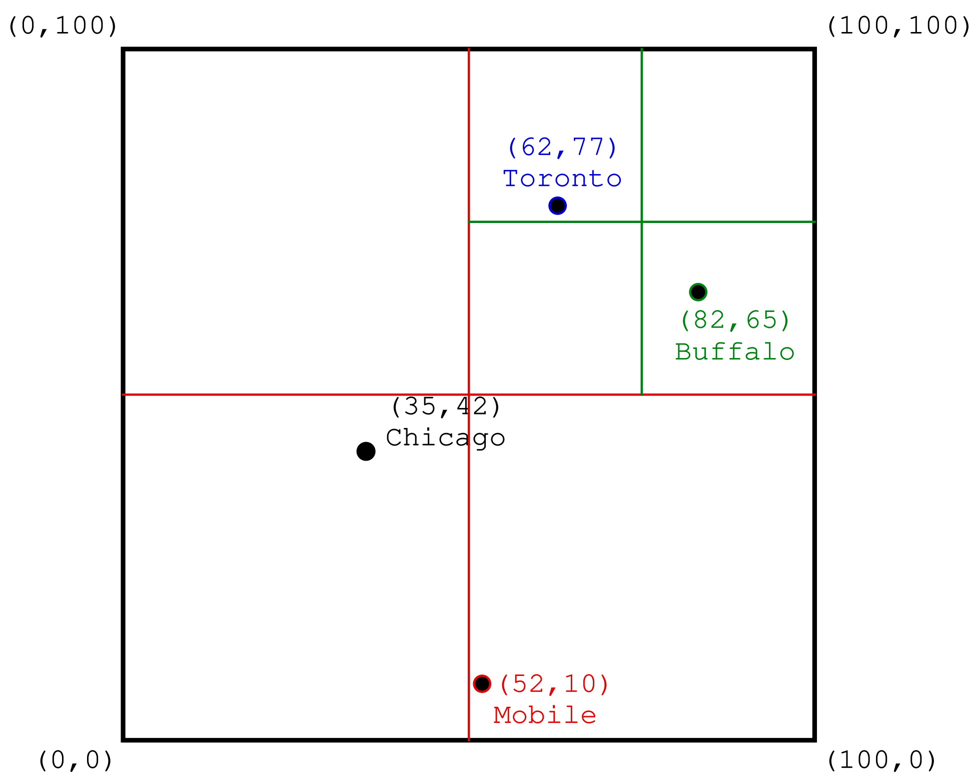
   Ex: $c = 1$



PR QUADTREE (Orenstein)

1. Regular decomposition point representation
2. Decomposition occurs whenever a block contains more than one point
3. Useful when the domain of data points is not discrete but finite
4. Maximum level of decomposition depends on the minimum separation between two points
   - if two points are very close, then decomposition can be very deep
   - can be overcome by viewing blocks as buckets with capacity *c* and only decomposing the block when it contains more than *c* points

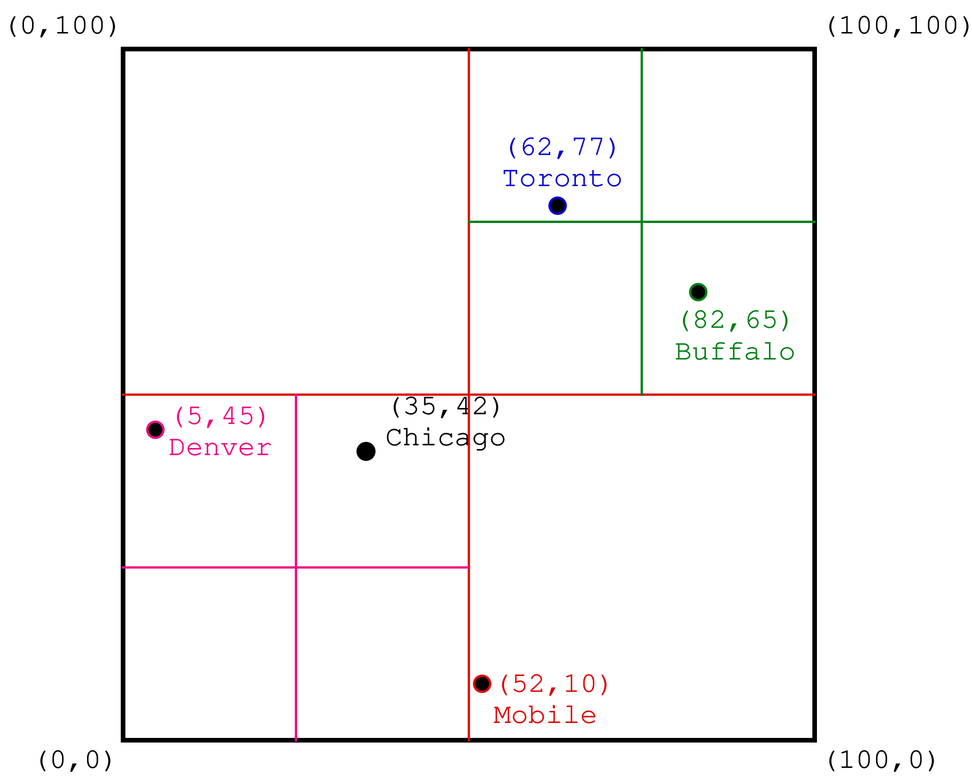
   Ex: $c = 1$



PR QUADTREE (Orenstein)

1. Regular decomposition point representation
2. Decomposition occurs whenever a block contains more than one point
3. Useful when the domain of data points is not discrete but finite
4. Maximum level of decomposition depends on the minimum separation between two points
   - if two points are very close, then decomposition can be very deep
   - can be overcome by viewing blocks as buckets with capacity *c* and only decomposing the block when it contains more than *c* points

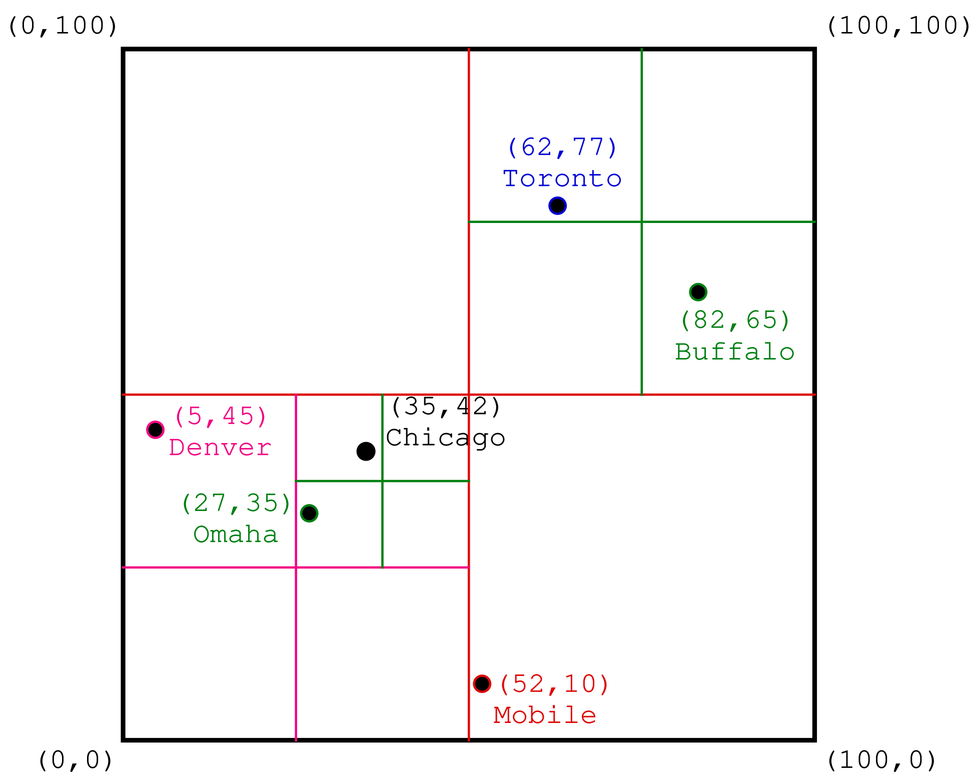
   Ex: $c = 1$



PR QUADTREE (Orenstein)

1. Regular decomposition point representation
2. Decomposition occurs whenever a block contains more than one point
3. Useful when the domain of data points is not discrete but finite
4. Maximum level of decomposition depends on the minimum separation between two points
   - if two points are very close, then decomposition can be very deep
   - can be overcome by viewing blocks as buckets with capacity c and only decomposing the block when it contains more than c points

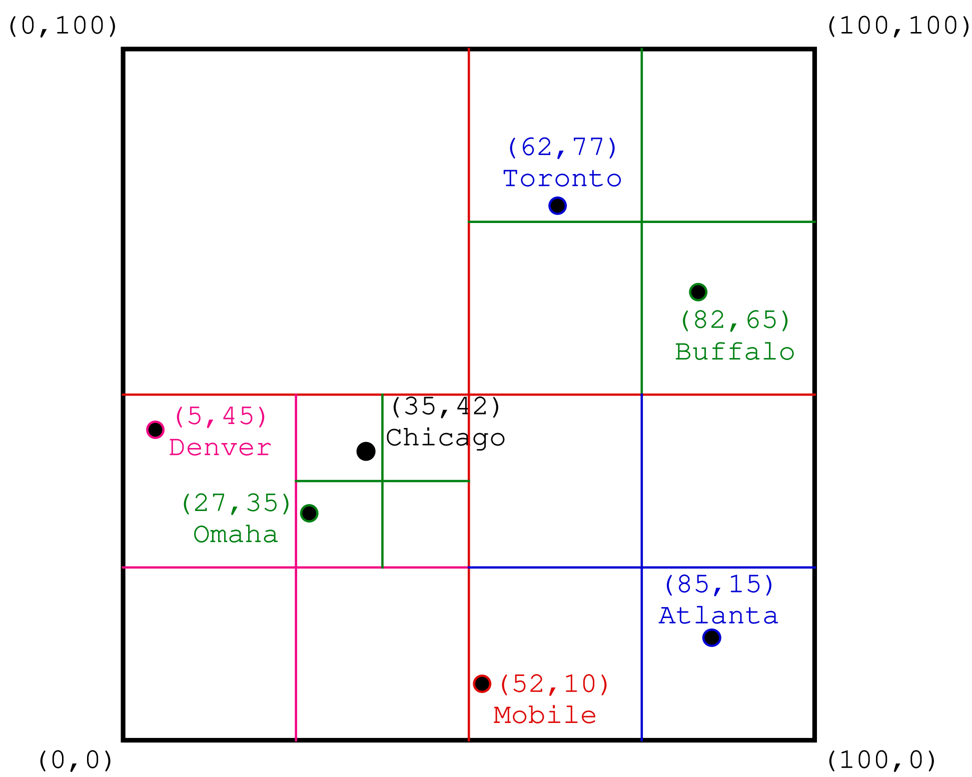
   Ex: $c = 1$



PR QUADTREE (Orenstein)

1. Regular decomposition point representation
2. Decomposition occurs whenever a block contains more than one point
3. Useful when the domain of data points is not discrete but finite
4. Maximum level of decomposition depends on the minimum separation between two points
   - if two points are very close, then decomposition can be very deep
   - can be overcome by viewing blocks as buckets with capacity *c* and only decomposing the block when it contains more than *c* points

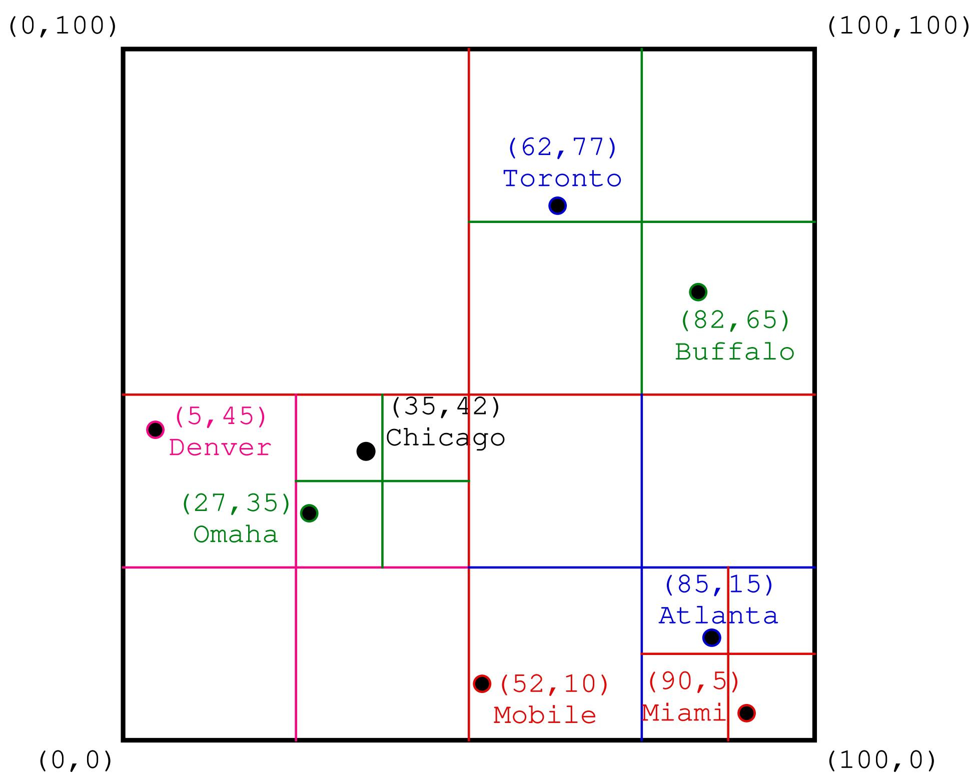
   Ex: $c = 1$



PR QUADTREE (Orenstein)

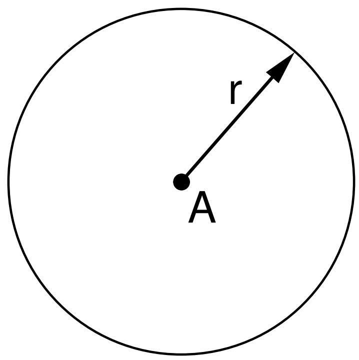
1. Regular decomposition point representation
2. Decomposition occurs whenever a block contains more than one point
3. Useful when the domain of data points is not discrete but finite
4. Maximum level of decomposition depends on the minimum separation between two points
   - if two points are very close, then decomposition can be very deep
   - can be overcome by viewing blocks as buckets with capacity *c* and only decomposing the block when it contains more than *c* points

Ex: $c = 1$



REGION SEARCH

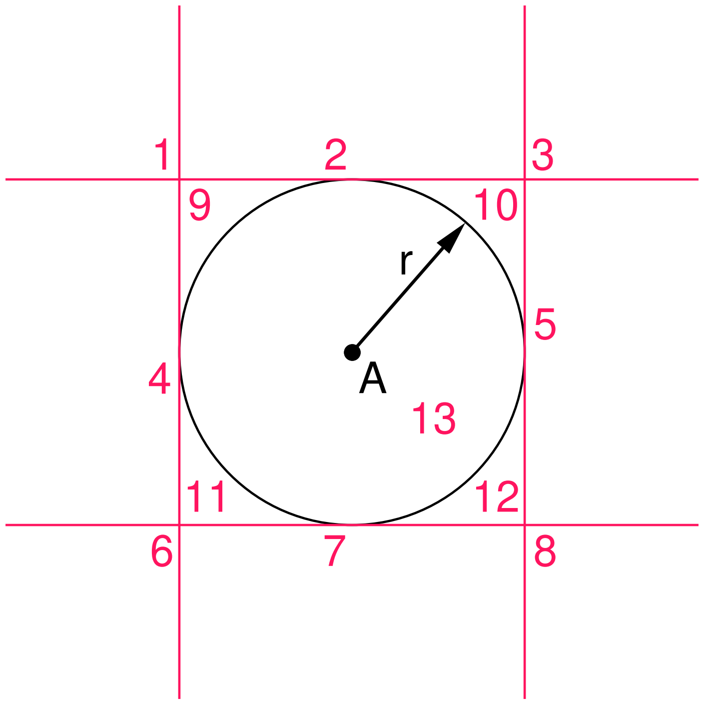
- Ex: Find all points within radius *r* of point A



- Use of quadtree results in pruning the search space

REGION SEARCH

- Ex: Find all points within radius *r* of point A

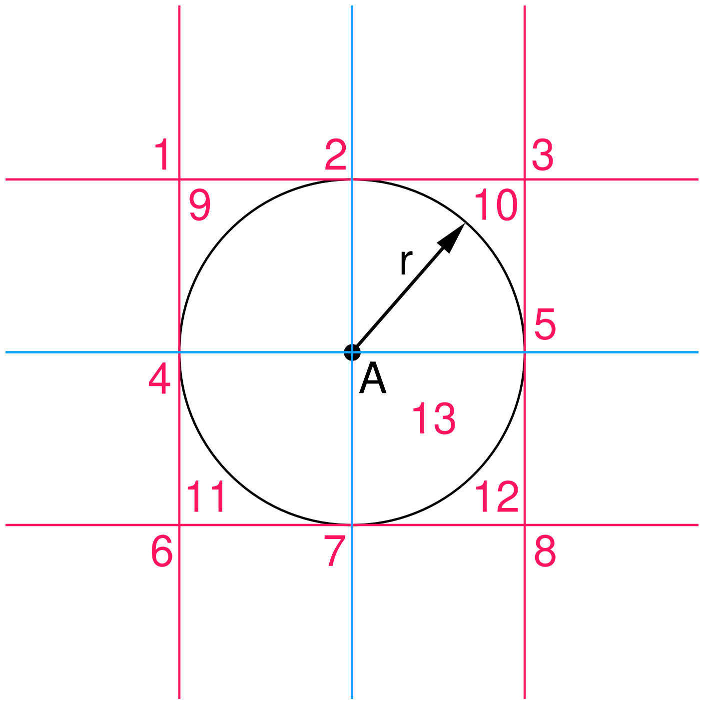


- Use of quadtree results in pruning the search space
- If a quadrant subdivision point *p* lies in a region *l*, then search the quadrants of *p* specified by *l*

1. SE
2. SE, SW
3. SW
4. SE, NE
5. SW, NW
6. NE
7. NE, NW
8. NW
9. All but NW
10. All but NE
11. All but SW
12. All but SE
13. All

REGION SEARCH

- Ex: Find all points within radius *r* of point A

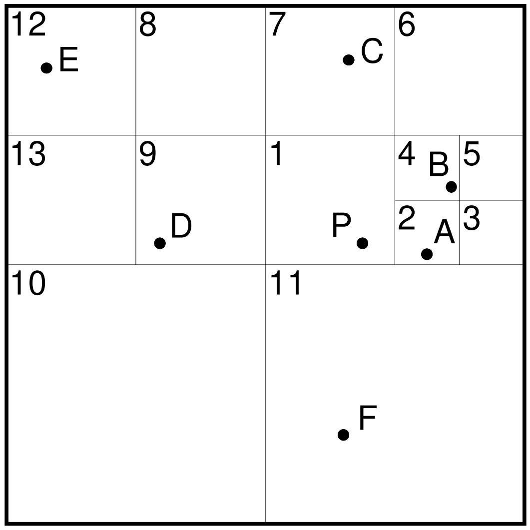


- Use of quadtree results in pruning the search space
- If a quadrant subdivision point *p* lies in a region *l*, then search the quadrants of *p* specified by *l*

1.	SE	6.	NE	11.	All but SW
2.	SE, SW	7.	NE, NW	12.	All but SE
3.	SW	8.	NW	13.	All
4.	SE, NE	9.	All but NW		
5.	SW, NW	10.	All but NE		

FINDING THE NEAREST OBJECT

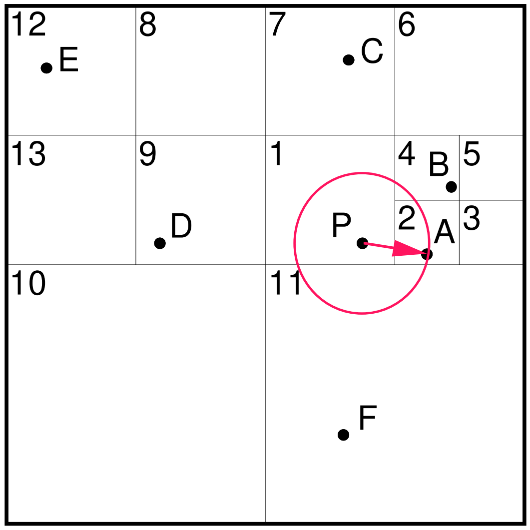
- Ex: find the nearest object to P



- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

FINDING THE NEAREST OBJECT

- Ex: find the nearest object to P

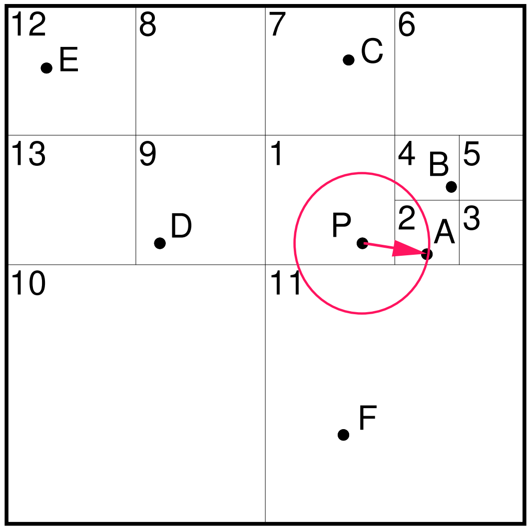


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

1. start at block 2 and compute distance to P from A

FINDING THE NEAREST OBJECT

- Ex: find the nearest object to P

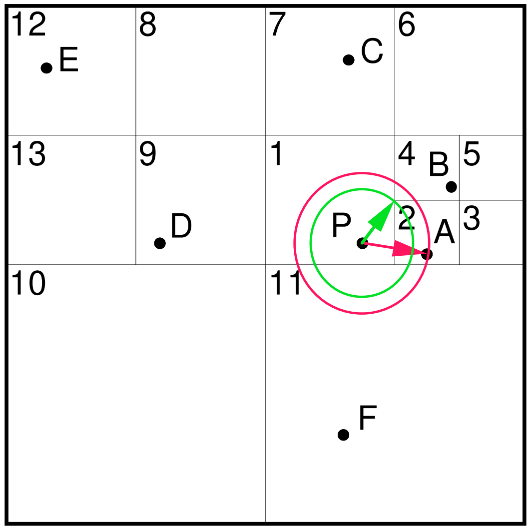


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

 1. start at block 2 and compute distance to P from A
 2. ignore block 3 whether or not it is empty as A is closer to P than any point in 3

FINDING THE NEAREST OBJECT

- Ex: find the nearest object to P

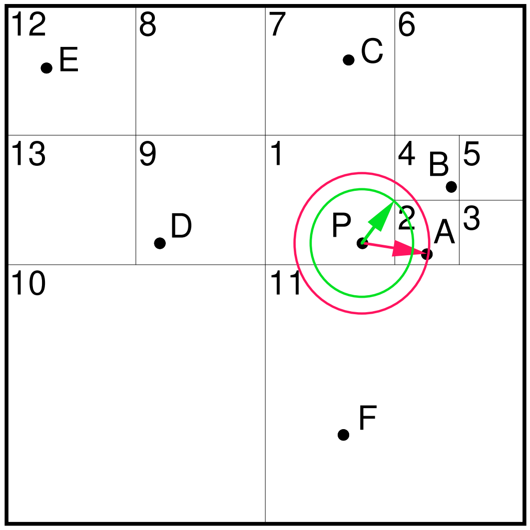


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

  1. start at block 2 and compute distance to P from A
  2. ignore block 3 whether or not it is empty as A is closer to P than any point in 3
  3. examine block 4 as distance to SW corner is shorter than the distance from P to A; however, reject B as it is further from P than A

FINDING THE NEAREST OBJECT

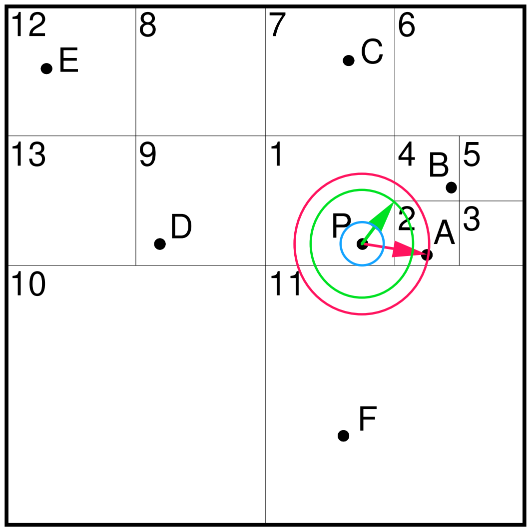
- Ex: find the nearest object to P



- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:
  1. start at block 2 and compute distance to P from A
  2. ignore block 3 whether or not it is empty as A is closer to P than any point in 3
  3. examine block 4 as distance to SW corner is shorter than the distance from P to A; however, reject B as it is further from P than A
  4. ignore blocks 6, 7, 8, 9, and 10 as the minimum distance to them from P is greater than the distance from P to A

FINDING THE NEAREST OBJECT

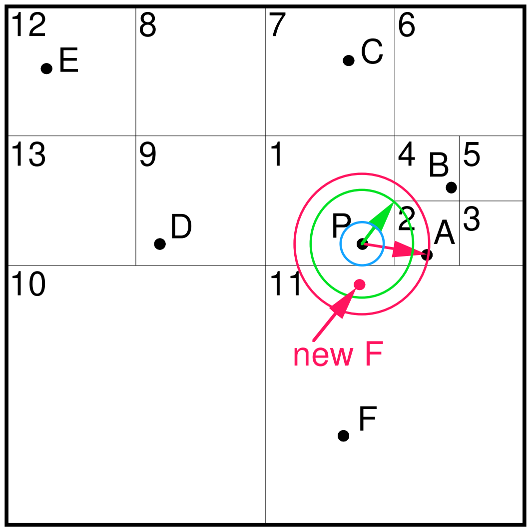
- Ex: find the nearest object to P



- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:
  1. start at block 2 and compute distance to P from A
  2. ignore block 3 whether or not it is empty as A is closer to P than any point in 3
  3. examine block 4 as distance to SW corner is shorter than the distance from P to A; however, reject B as it is further from P than A
  4. ignore blocks 6, 7, 8, 9, and 10 as the minimum distance to them from P is greater than the distance from P to A
  5. examine block 11 as the distance from P to the southern border of 1 is shorter than the distance from P to A; however, reject F as it is further from P than A

FINDING THE NEAREST OBJECT

- Ex: find the nearest object to P



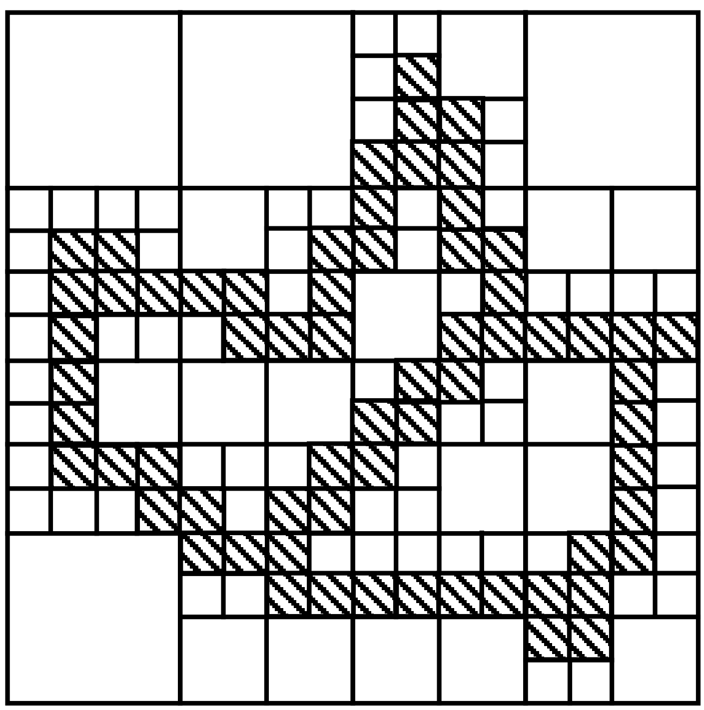
- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:
  1. start at block 2 and compute distance to P from A
  2. ignore block 3 whether or not it is empty as A is closer to P than any point in 3
  3. examine block 4 as distance to SW corner is shorter than the distance from P to A; however, reject B as it is further from P than A
  4. ignore blocks 6, 7, 8, 9, and 10 as the minimum distance to them from P is greater than the distance from P to A
  5. examine block 11 as the distance from P to the southern border of 1 is shorter than the distance from P to A; however, reject F as it is further from P than A
- If F was moved, a better order would have started with block 11, the southern neighbor of 1, as it is closest

COMPARISON OF POINT, MX, AND PR QUADTREES

FEATURE	MX	PR	POINT
Regular decomposition	Yes	Yes	No
Type of nodes	Data Stored in leaf nodes and non-leaf nodes are for control	Data stored in leaf nodes and non-leaf nodes are for control	Data stored in leaf nodes and non-leaf nodes
Shape of tree depends on order of inserting nodes	No	No	Yes
Deletion	Simple but may have to collapse WHITE nodes	Simple but may have to collapse WHITE nodes	Complex
Size of space represented	Finite	Finite	Unbounded
Type of data represented	Discrete	Continuous	Continuous
Shape of space represented	Square	Rectangle	Unbounded
Stores coordinates	No	Yes	Yes
Depth of tree (d); assume M points	All nodes are at the same depth, n for a 2^n by 2^n region	For square region with side length L and minimum separation S between two points, $\lceil \log_4(M-1) \rceil \leq d$ and $d \leq \lceil \log_2((L/S)2^{.5}) \rceil$	$\lceil \log_4(3M) \rceil \leq d$ and $d \leq M-1$

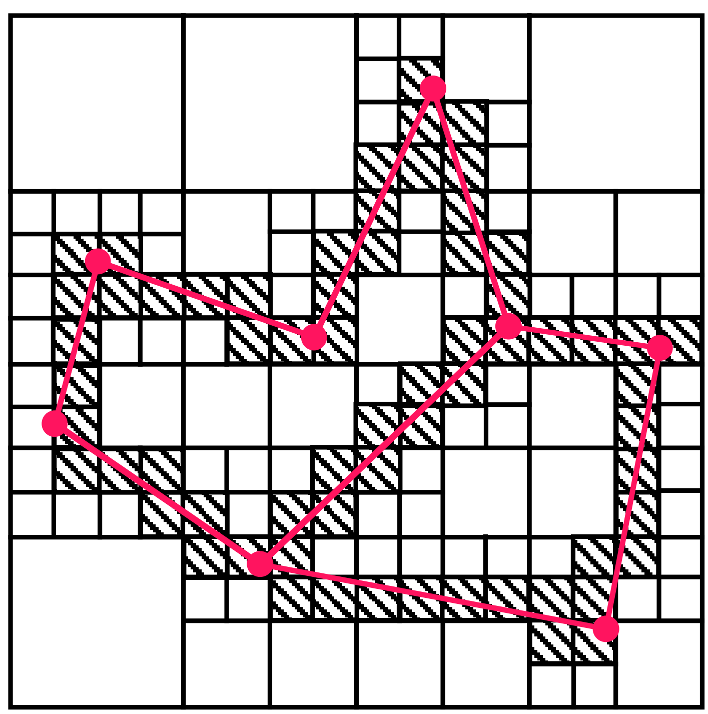
APPLICATION OF THE MX QUADTREE (Hunter)

- Represent the boundary as a sequence of BLACK pixels in a region quadtree
- Useful for a simple digitized polygon (i.e., non-intersecting edges)
- Three types of nodes
  1. interior - treat like WHITE nodes
  2. exterior - treat like WHITE nodes
  3. boundary - the edge of the polygon passes through them and treated like BLACK nodes
- Disadvantages
  1. a thickness is associated with the line segments
  2. no more than 4 lines can meet at a point



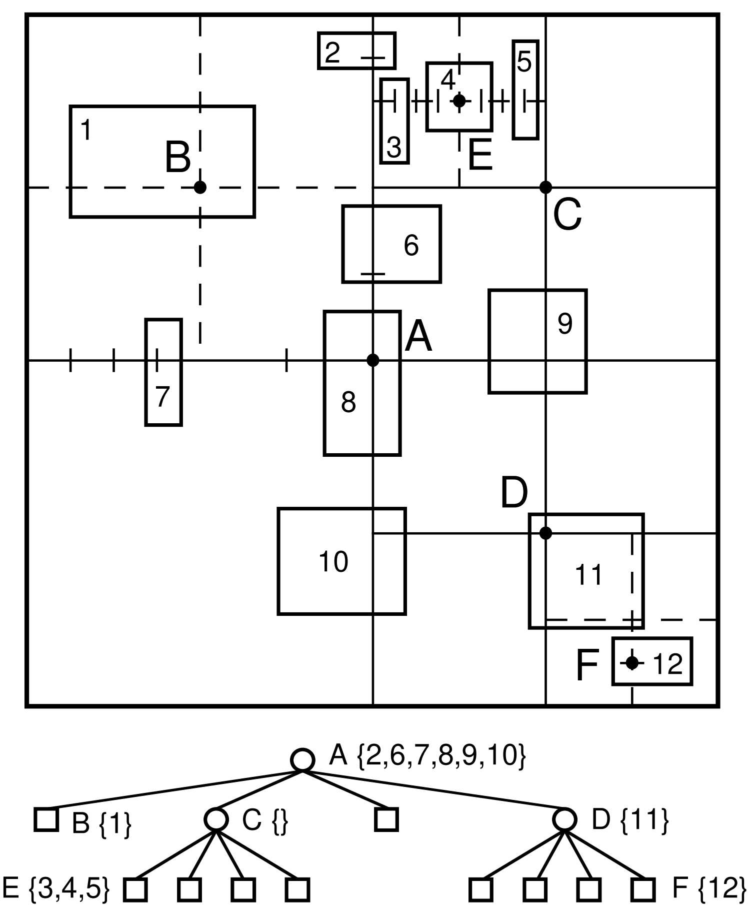
APPLICATION OF THE MX QUADTREE (Hunter)

- Represent the boundary as a sequence of BLACK pixels in a region quadtree
- Useful for a simple digitized polygon (i.e., non-intersecting edges)
- Three types of nodes
  1. interior - treat like WHITE nodes
  2. exterior - treat like WHITE nodes
  3. boundary - the edge of the polygon passes through them and treated like BLACK nodes
- Disadvantages
  1. a thickness is associated with the line segments
  2. no more than 4 lines can meet at a point



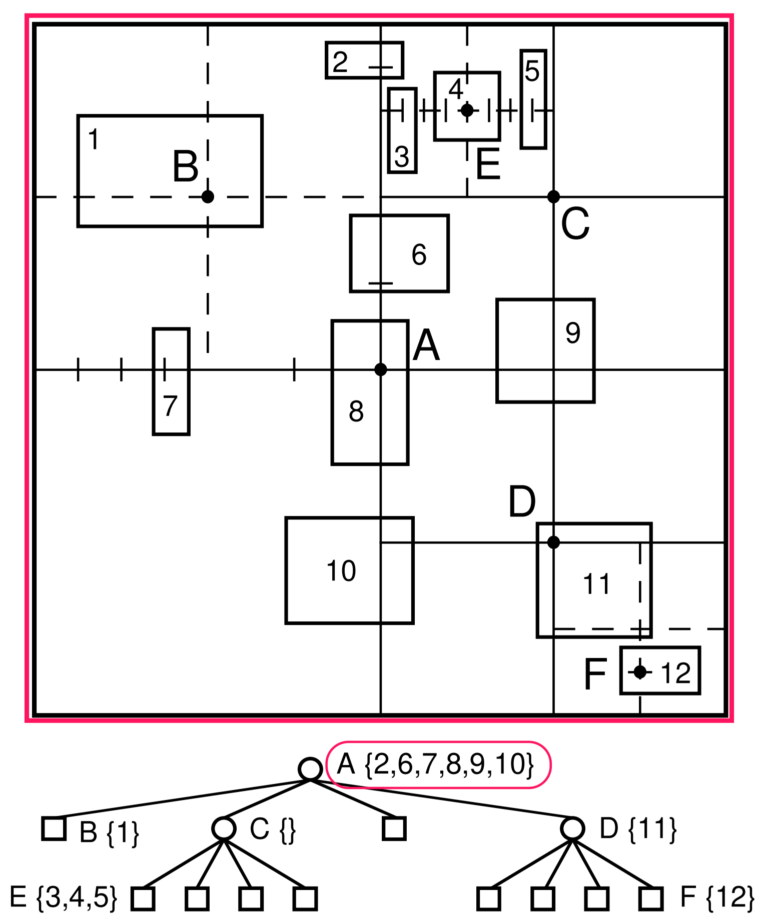
MX-CIF QUADTREE (Kedem)

1. Collections of small rectangles for VLSI applications
2. Each rectangle is associated with its minimum enclosing quadtree block
3. Like hashing: quadtree blocks serve as hash buckets



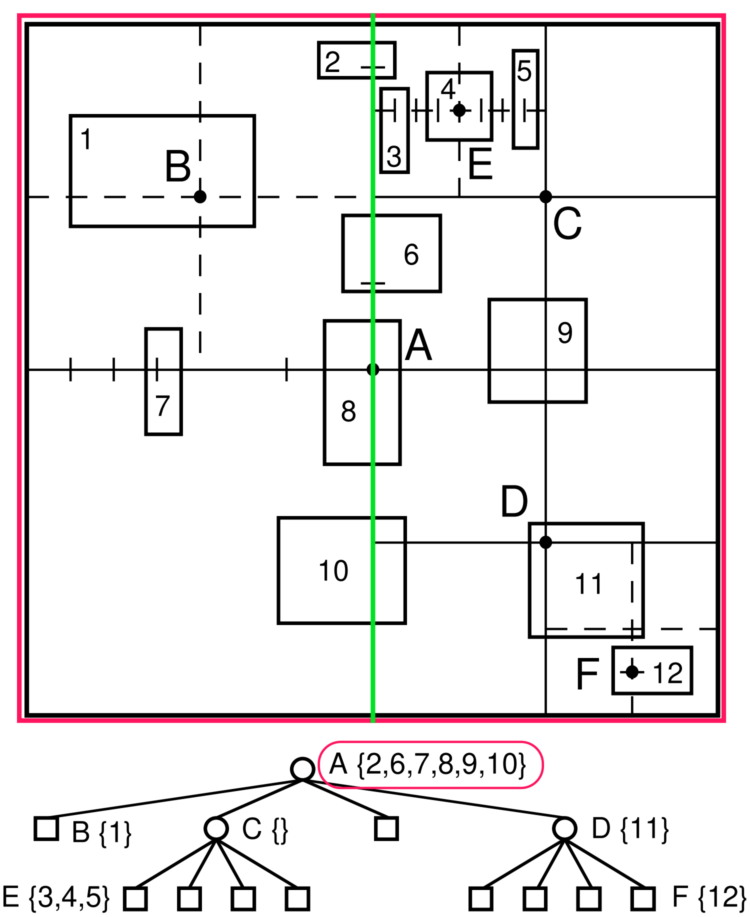
MX-CIF QUADTREE (Kedem)

1. Collections of small rectangles for VLSI applications
2. Each rectangle is associated with its minimum enclosing quadtree block
3. Like hashing: quadtree blocks serve as hash buckets
4. Collision = more than one rectangle in a block
   - resolve by using two one-dimensional MX-CIF trees to store the rectangle intersecting the lines passing through each subdivision point

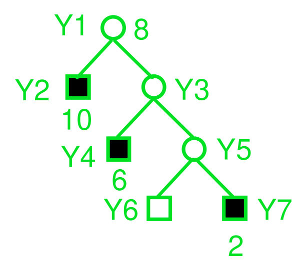


MX-CIF QUADTREE (Kedem)

1. Collections of small rectangles for VLSI applications
2. Each rectangle is associated with its minimum enclosing quadtree block
3. Like hashing: quadtree blocks serve as hash buckets
4. Collision = more than one rectangle in a block
   - resolve by using two one-dimensional MX-CIF trees to store the rectangle intersecting the lines passing through each subdivision point
   - one for y-axis

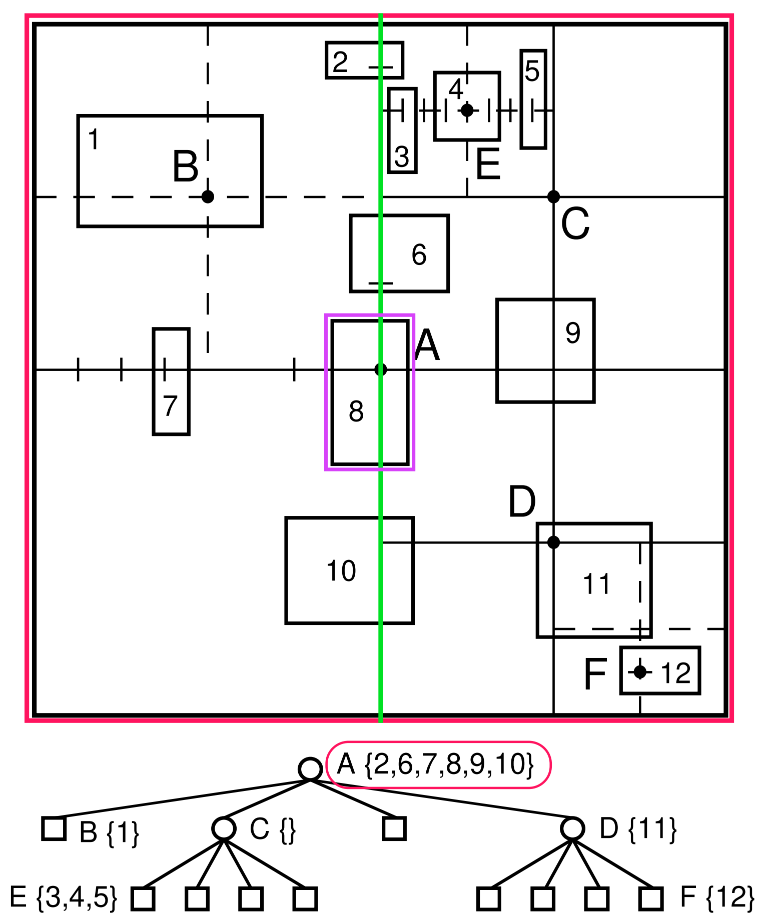


Binary tree for *y*-axis through A

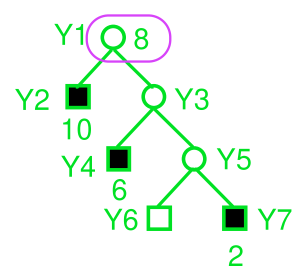


MX-CIF QUADTREE (Kedem)

1. Collections of small rectangles for VLSI applications
2. Each rectangle is associated with its minimum enclosing quadtree block
3. Like hashing: quadtree blocks serve as hash buckets
4. Collision = more than one rectangle in a block
   - resolve by using two one-dimensional MX-CIF trees to store the rectangle intersecting the lines passing through each subdivision point
   - one for y-axis
   - if a rectangle intersects both *x* and *y* axes, then associate it with the *y* axis

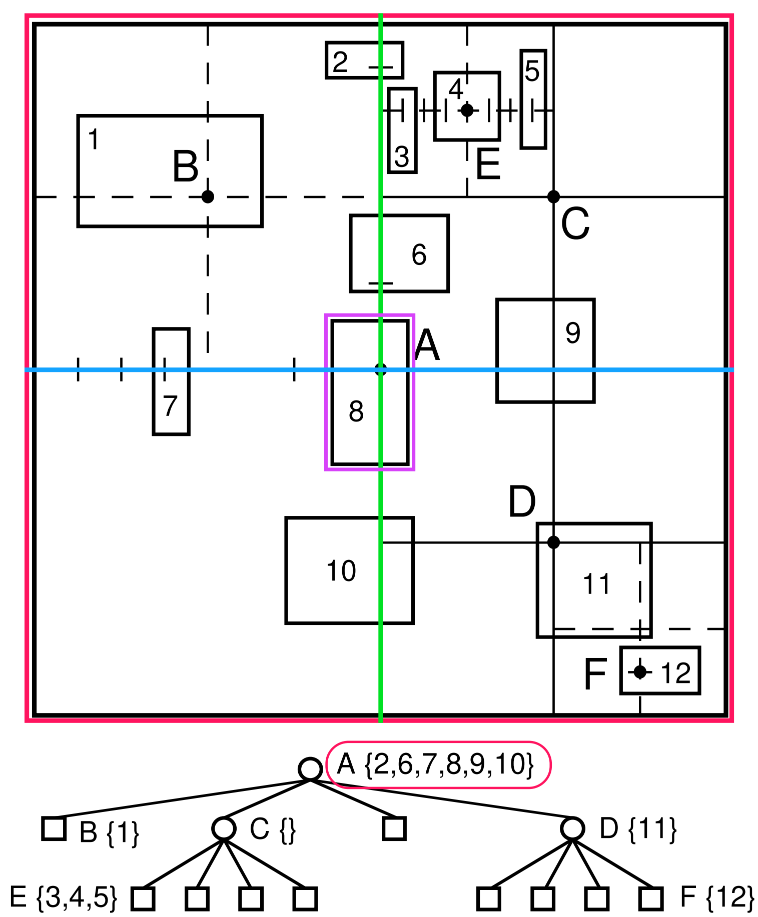


Binary tree for *y*-axis through A

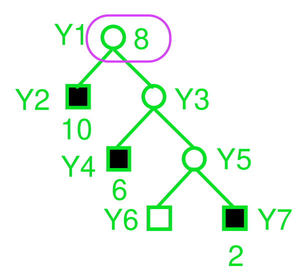


MX-CIF QUADTREE (Kedem)

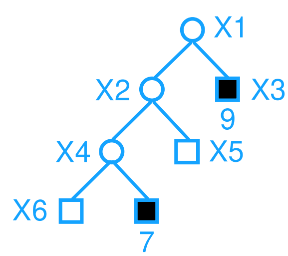
1. Collections of small rectangles for VLSI applications
2. Each rectangle is associated with its minimum enclosing quadtree block
3. Like hashing: quadtree blocks serve as hash buckets
4. Collision = more than one rectangle in a block
   - resolve by using two one-dimensional MX-CIF trees to store the rectangle intersecting the lines passing through each subdivision point
   - one for y-axis
   - if a rectangle intersects both *x* and *y* axes, then associate it with the *y* axis
   - one for x-axis



Binary tree for *y*-axis through A

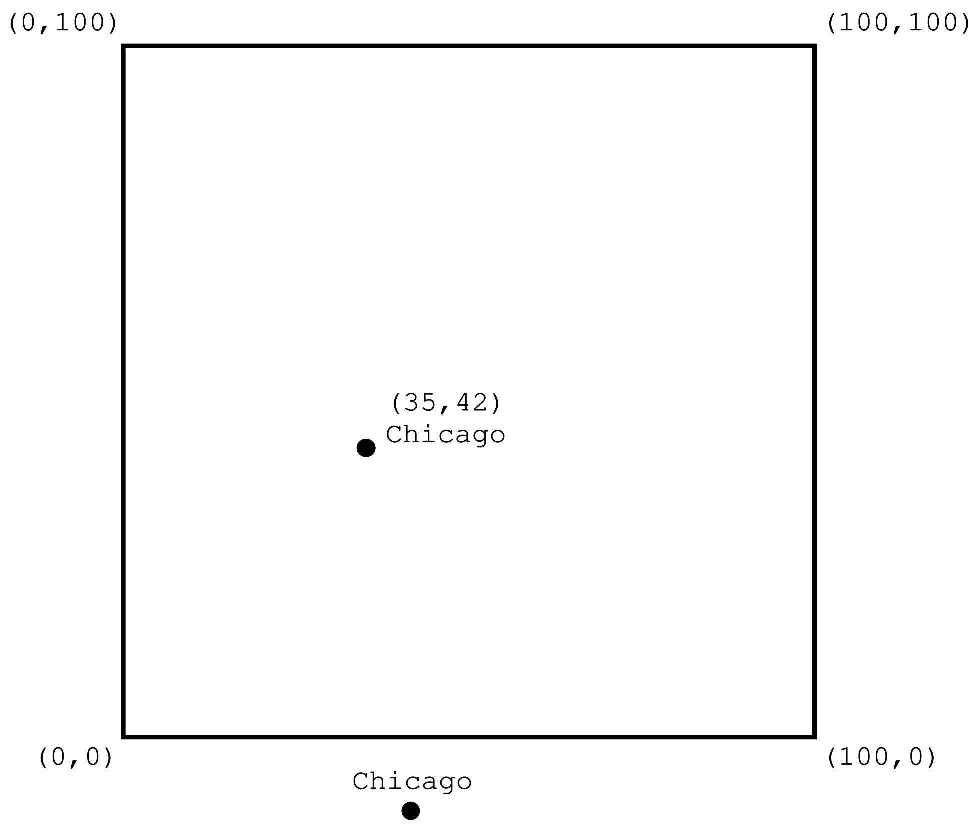


Binary tree for *x*-axis through A



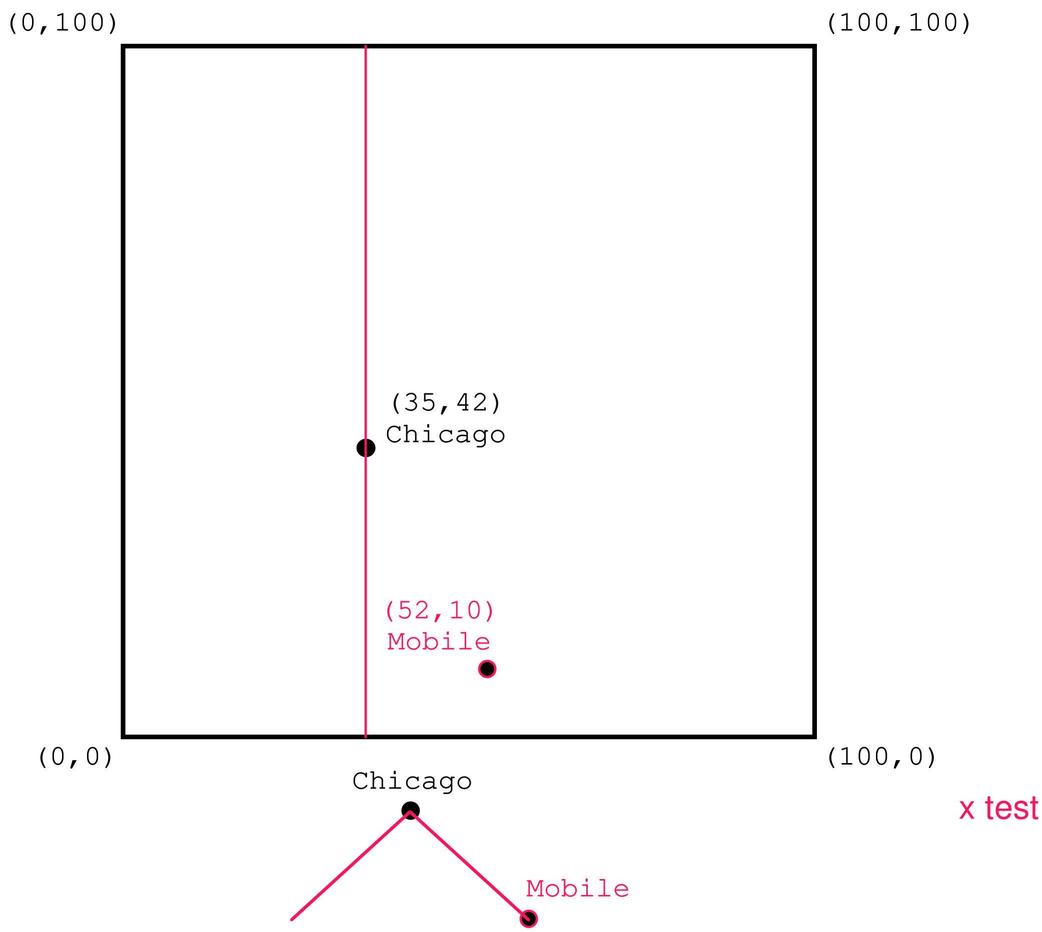
K-D TREE (Bentley)

- Test one attribute at a time instead of all simultaneously as in the point quadtree
- Usually cycle through all the attributes
- Shape of the tree depends on the order in which the data is encountered



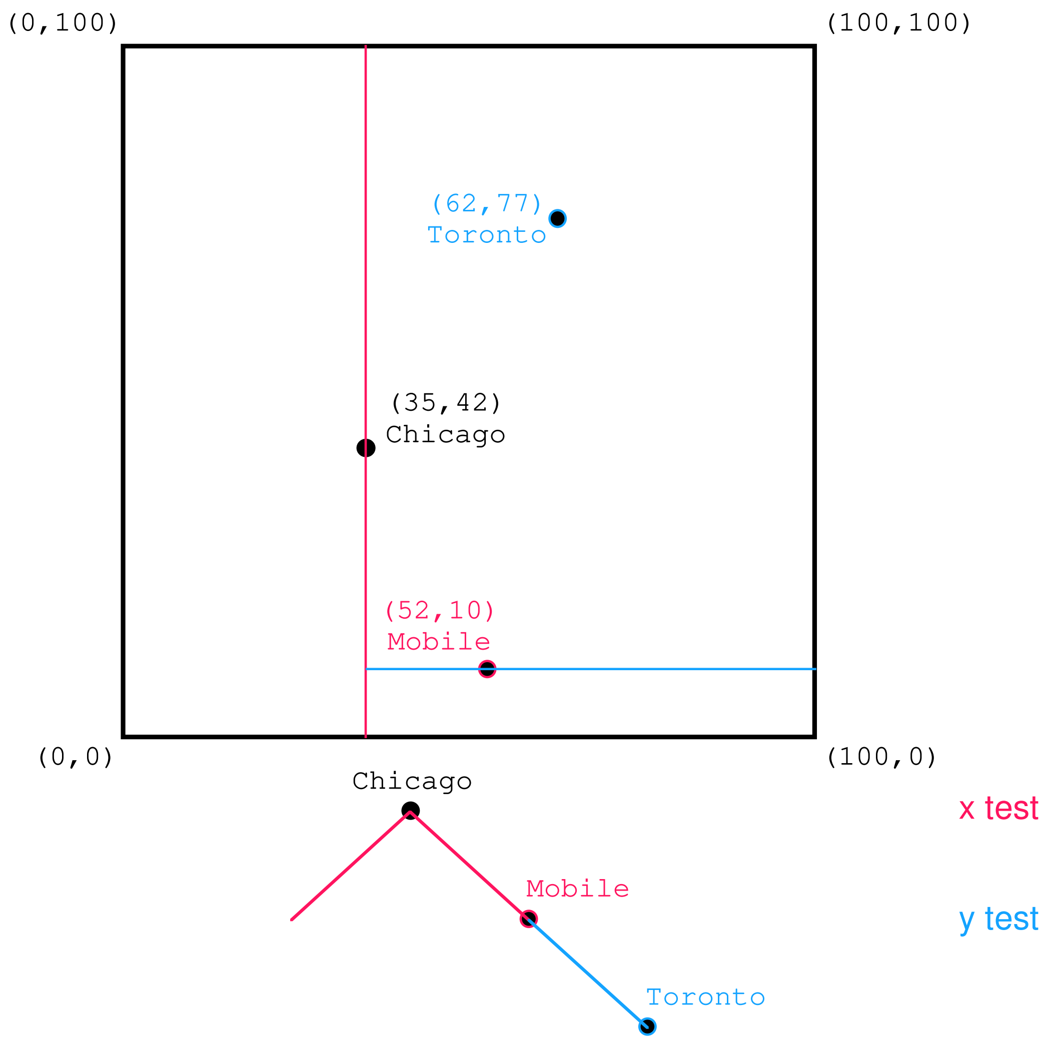
K-D TREE (Bentley)

- Test one attribute at a time instead of all simultaneously as in the point quadtree
- Usually cycle through all the attributes
- Shape of the tree depends on the order in which the data is encountered



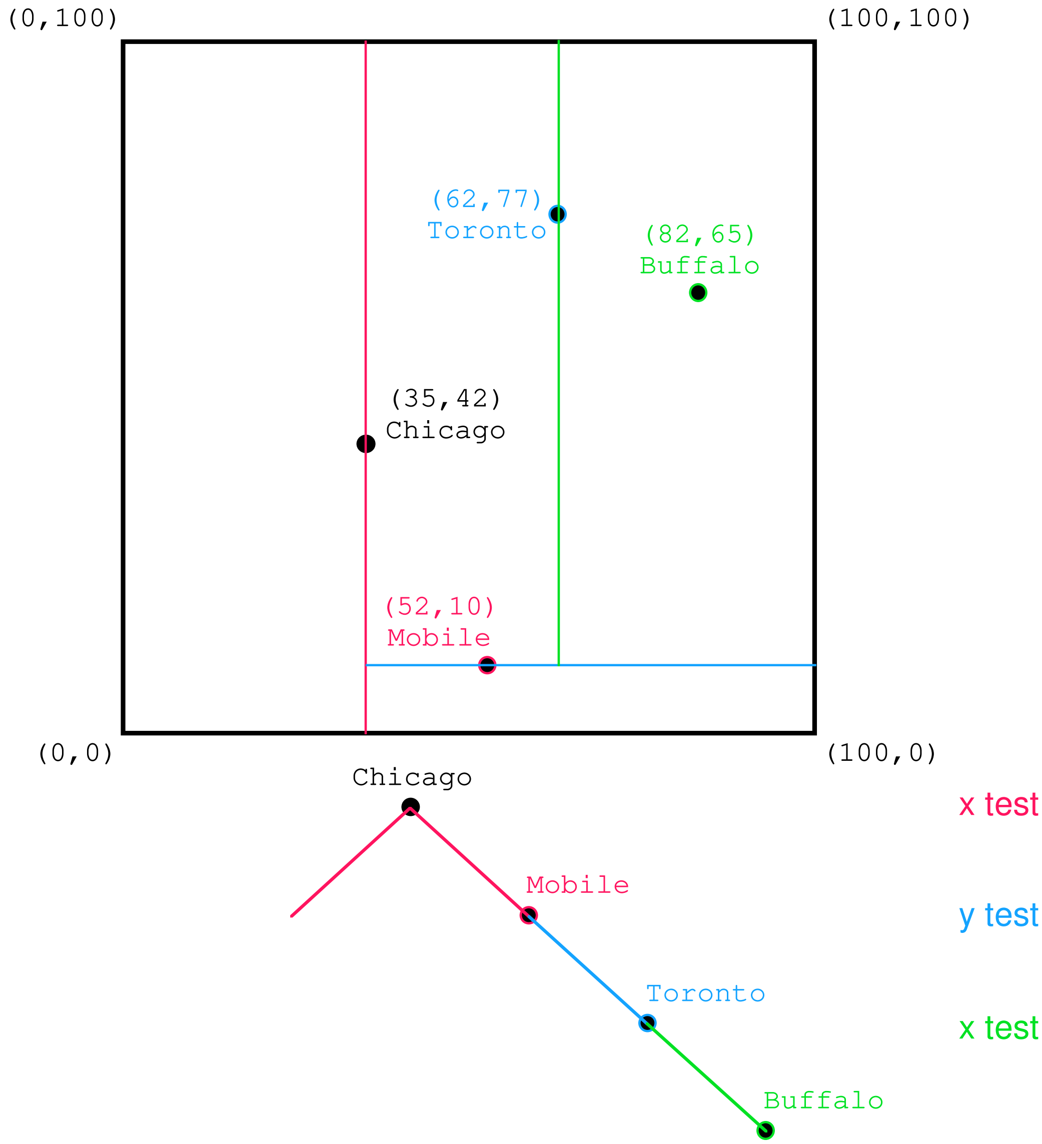
K-D TREE (Bentley)

- Test one attribute at a time instead of all simultaneously as in the point quadtree
- Usually cycle through all the attributes
- Shape of the tree depends on the order in which the data is encountered



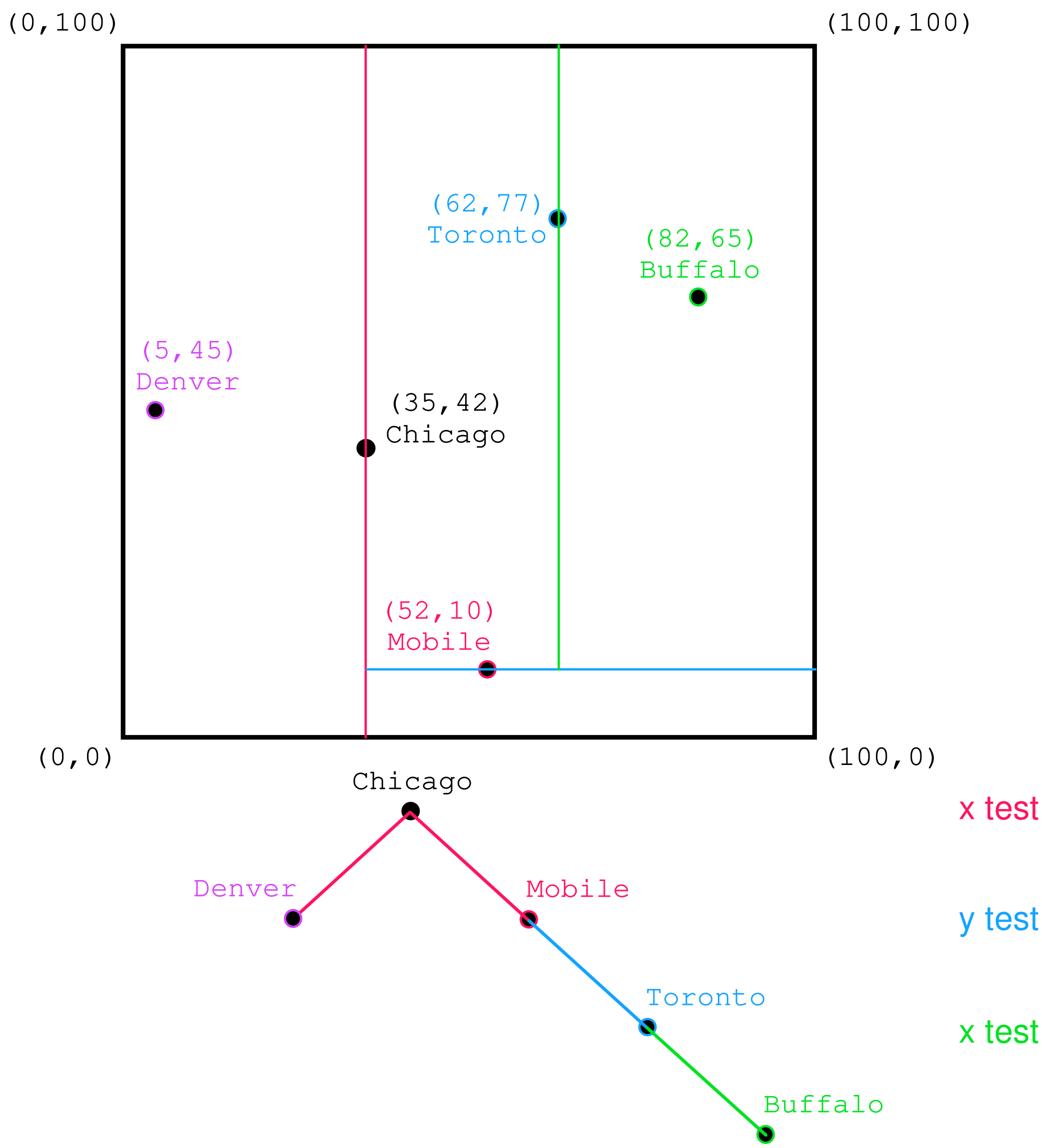
K-D TREE (Bentley)

- Test one attribute at a time instead of all simultaneously as in the point quadtree
- Usually cycle through all the attributes
- Shape of the tree depends on the order in which the data is encountered



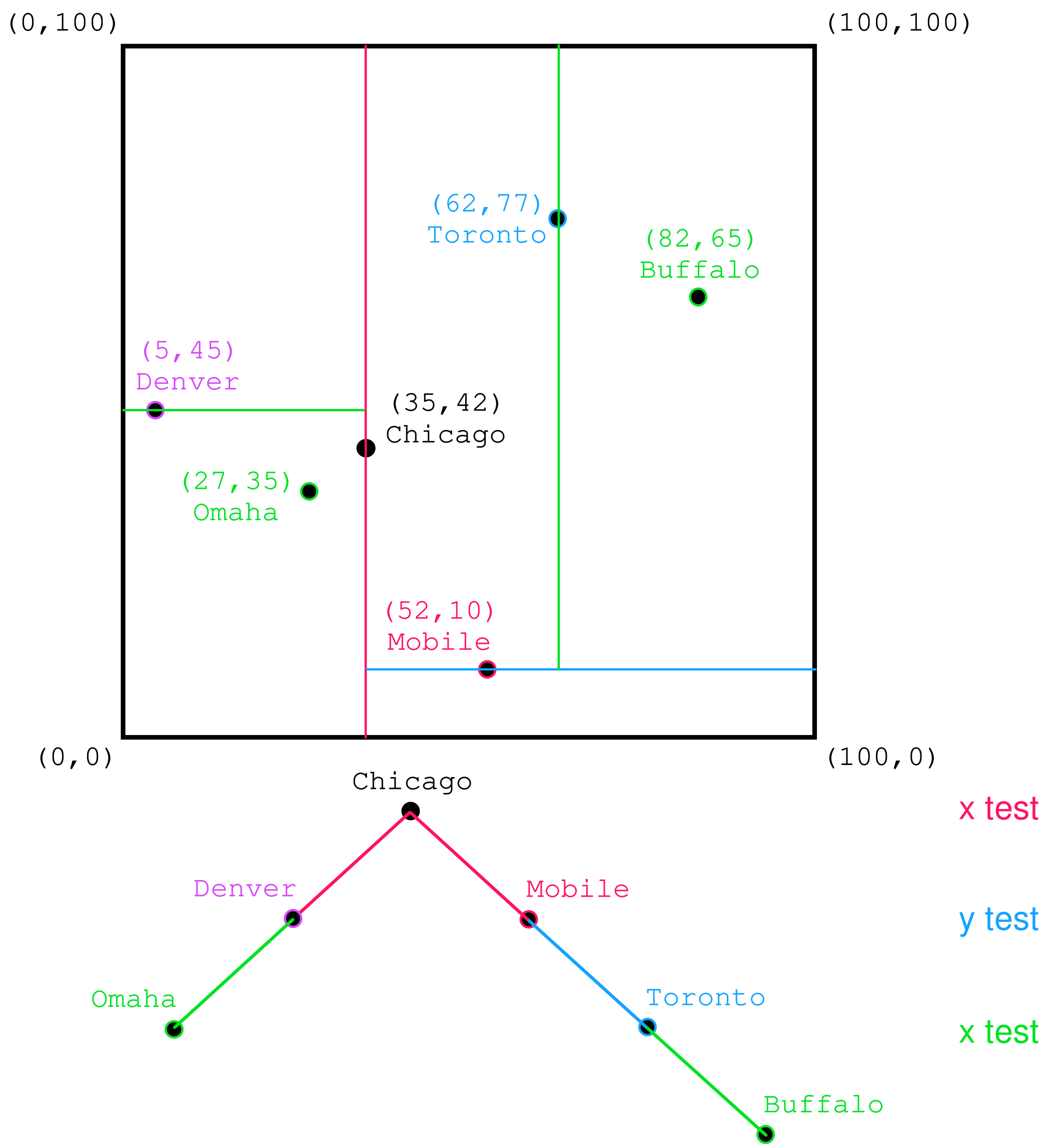
K-D TREE (Bentley)

- Test one attribute at a time instead of all simultaneously as in the point quadtree
- Usually cycle through all the attributes
- Shape of the tree depends on the order in which the data is encountered



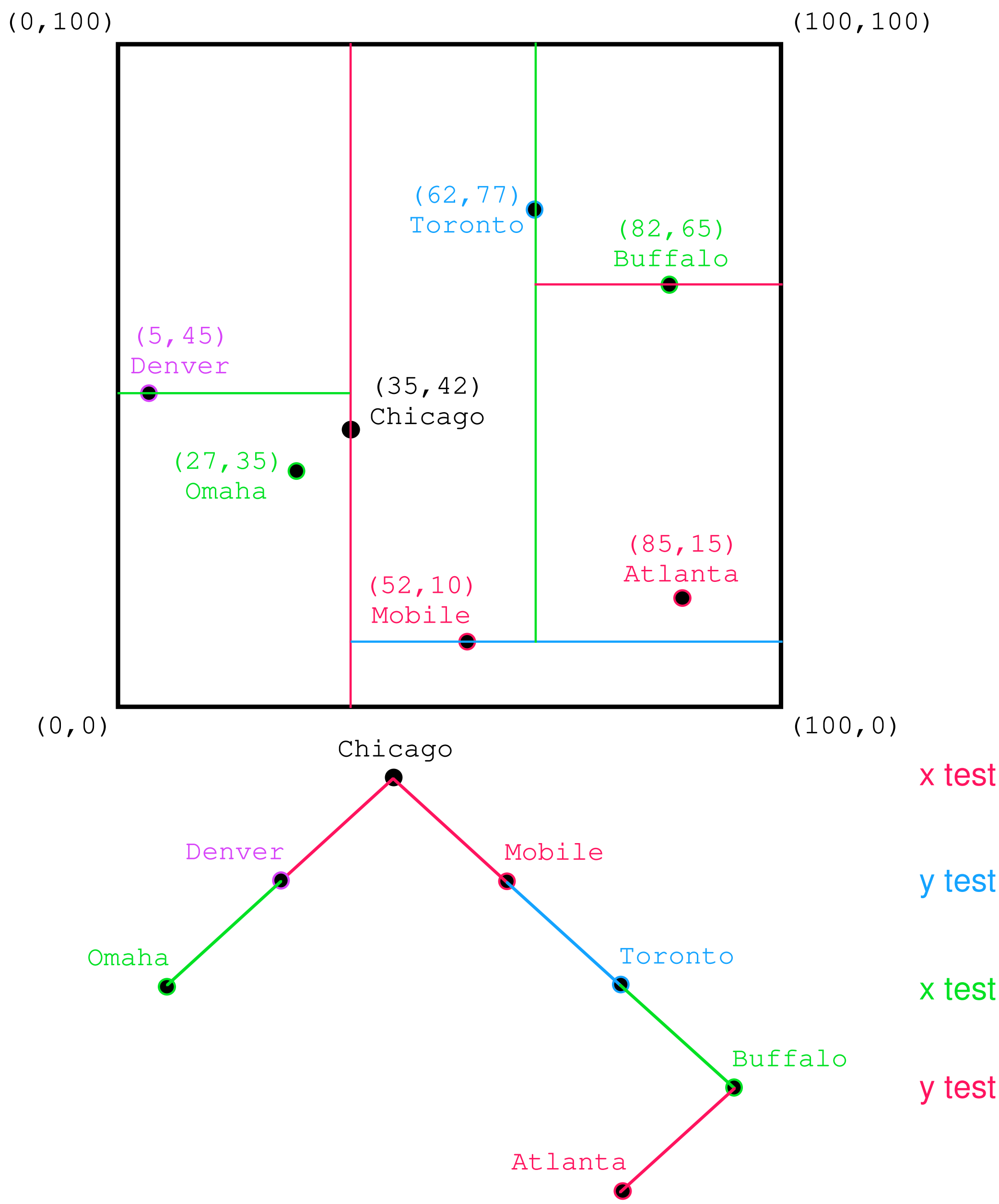
K-D TREE (Bentley)

- Test one attribute at a time instead of all simultaneously as in the point quadtree
- Usually cycle through all the attributes
- Shape of the tree depends on the order in which the data is encountered



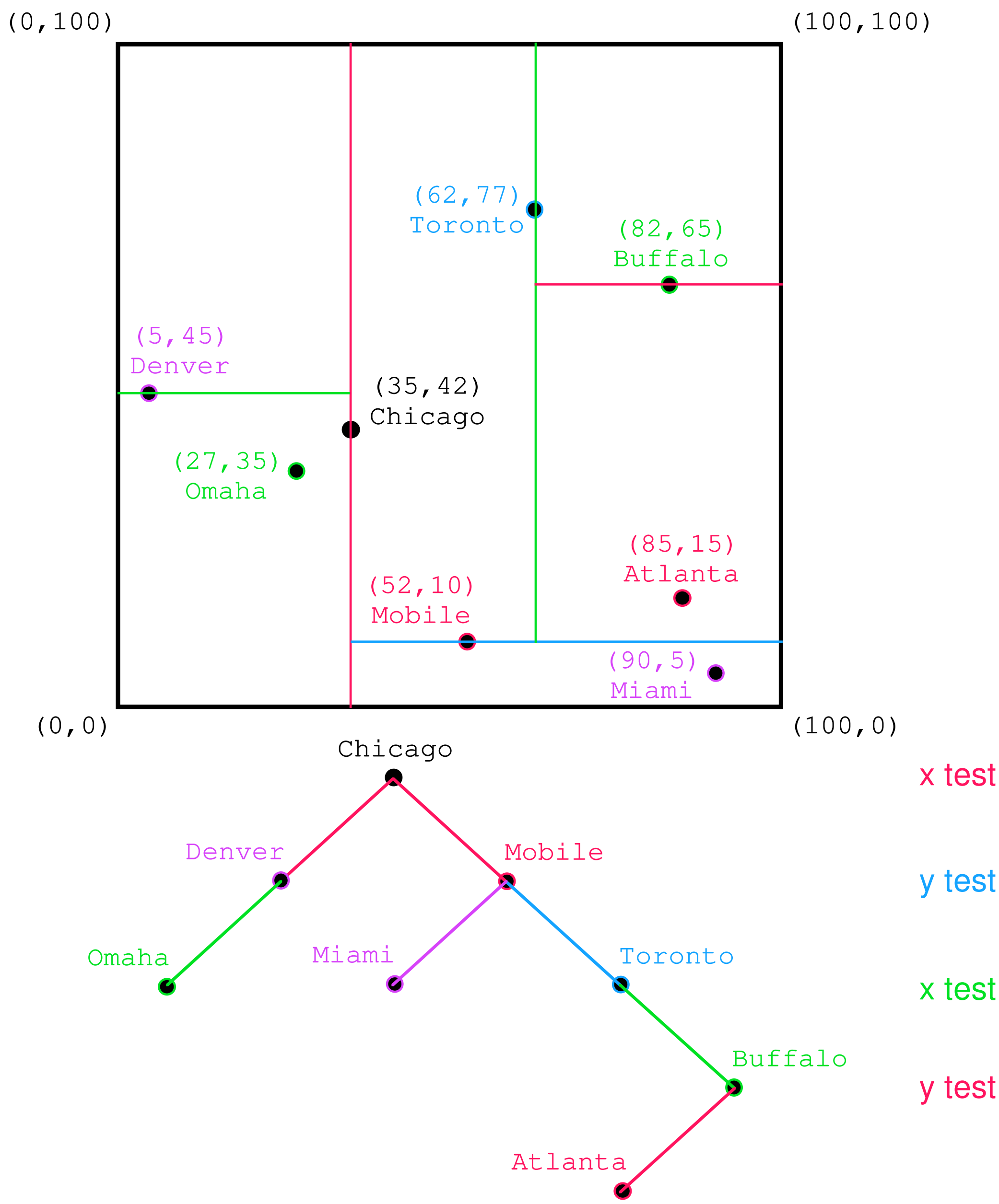
K-D TREE (Bentley)

- Test one attribute at a time instead of all simultaneously as in the point quadtree
- Usually cycle through all the attributes
- Shape of the tree depends on the order in which the data is encountered



K-D TREE (Bentley)

- Test one attribute at a time instead of all simultaneously as in the point quadtree
- Usually cycle through all the attributes
- Shape of the tree depends on the order in which the data is encountered

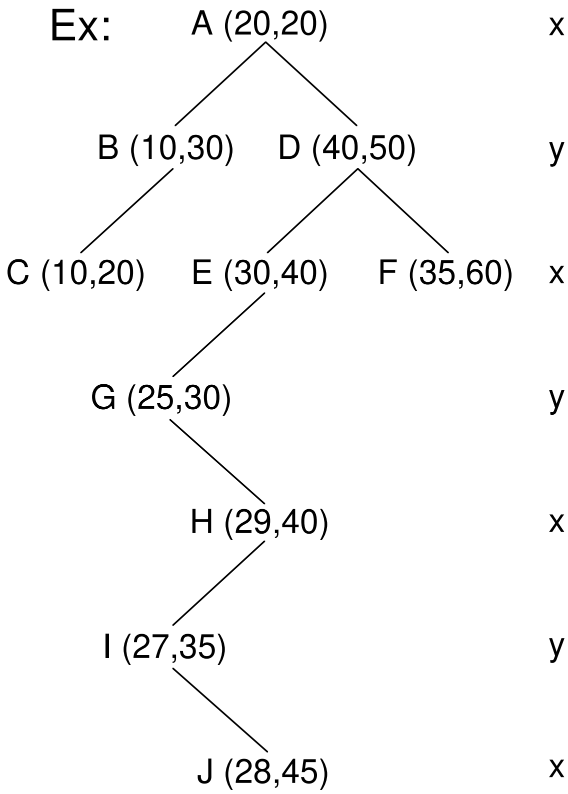


K-D TREE DELETION

- Similar to deletion in binary search trees
- Assume branch to HISON if test value is ≥ root value
- Assume root discriminates on the *x* coordinate and subsequent alternation with the *y* coordinate value
- Algorithm:
  1. replace root by node in HISON with the minimum *x* coordinate value
  2. repeat the process for the position of the replacement node using the *x* or *y* coordinate values depending on whether the replacement node is an *x* or a *y* discriminator

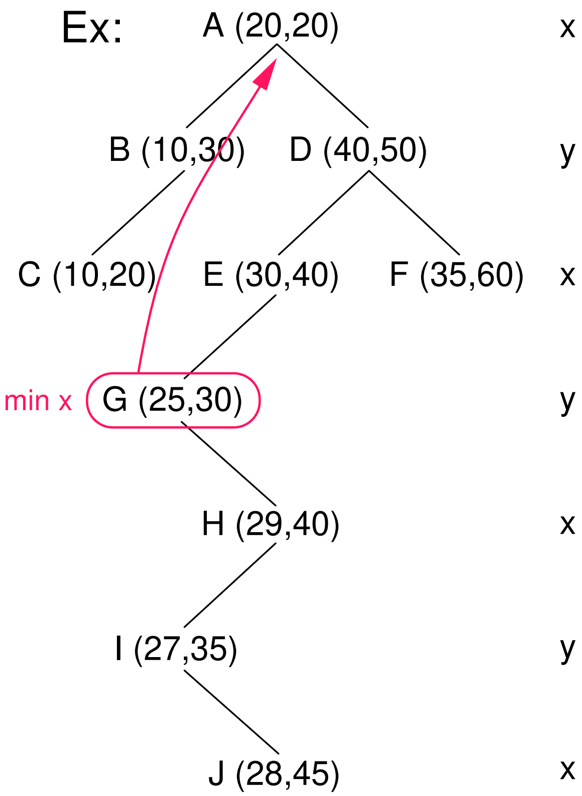
EXAMPLE OF K-D TREE DELETION

Ex:



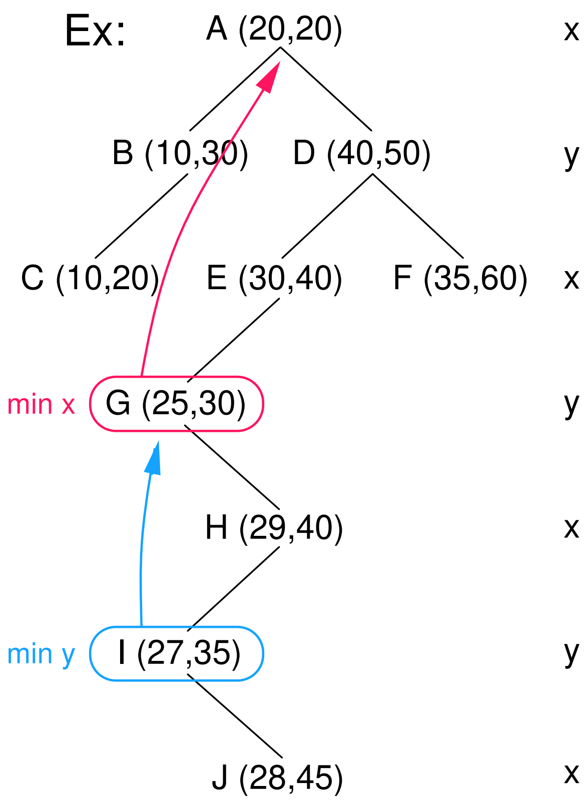
EXAMPLE OF K-D TREE DELETION

Ex:



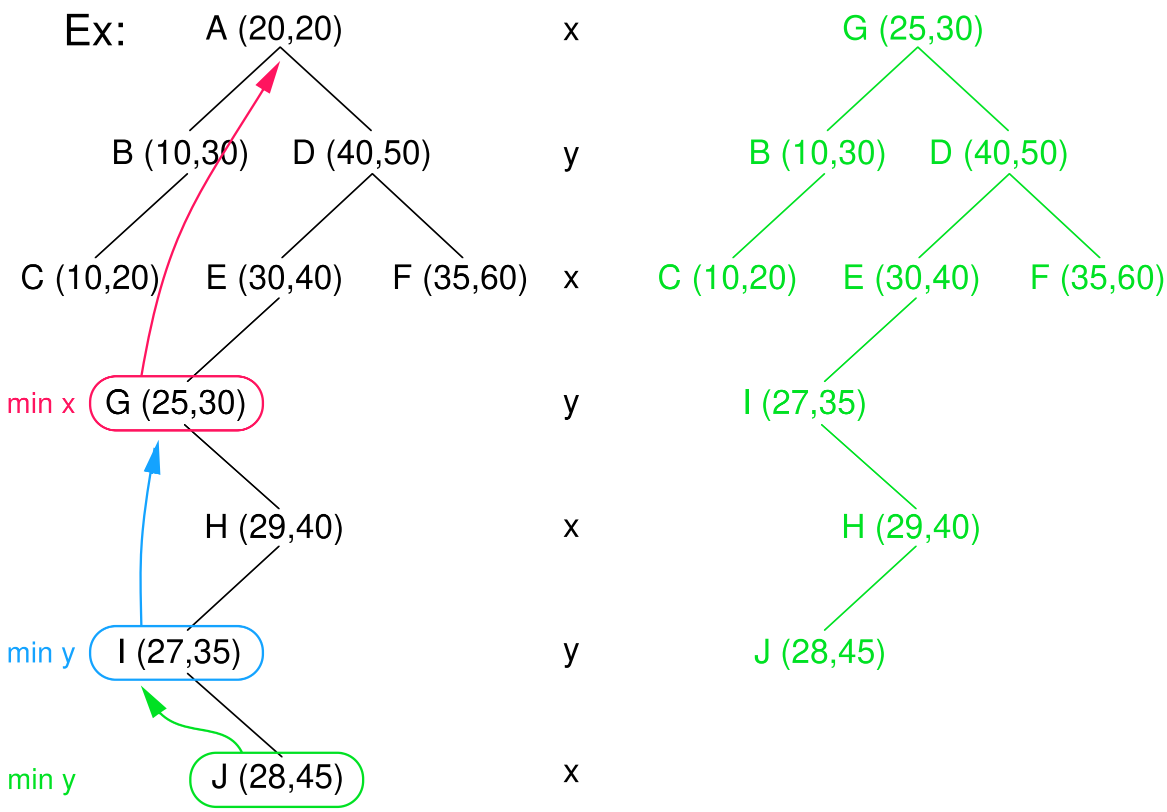
EXAMPLE OF K-D TREE DELETION

Ex:



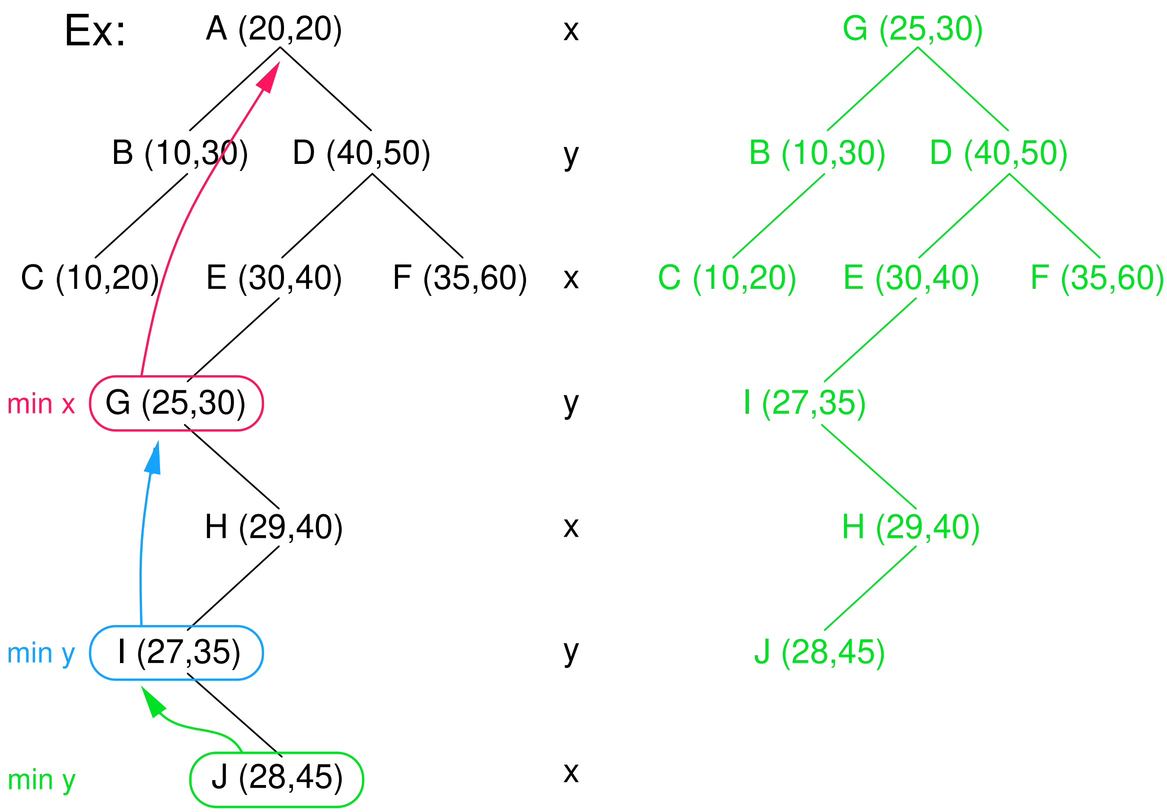
EXAMPLE OF K-D TREE DELETION

Ex:



EXAMPLE OF K-D TREE DELETION

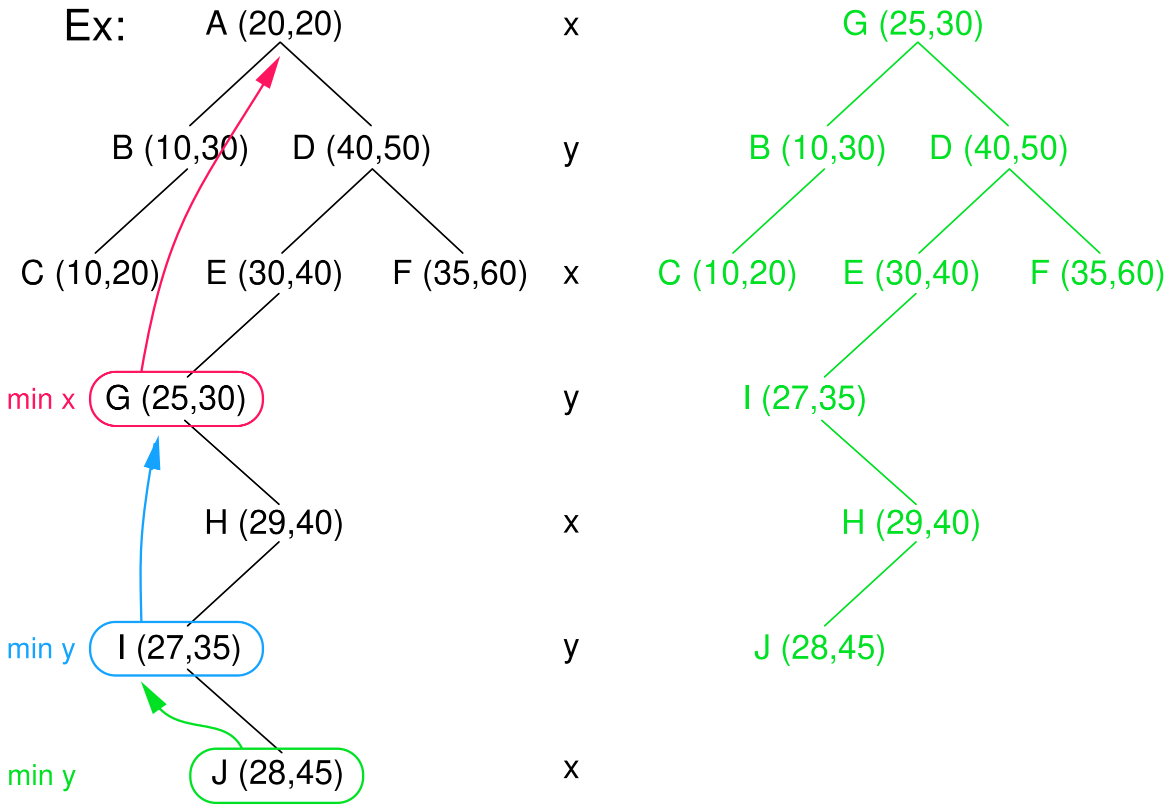
Ex:



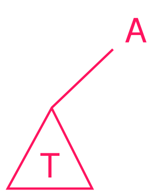
- Analogy with binary search trees implies that we can replace the root by the node in LOSON with the max *x* coordinate value. However, if there is more than one node with the same max value, then after replacement there would be a node in LOSON which is not strictly less than the new root (e.g., nodes B and C)

EXAMPLE OF K-D TREE DELETION

Ex:

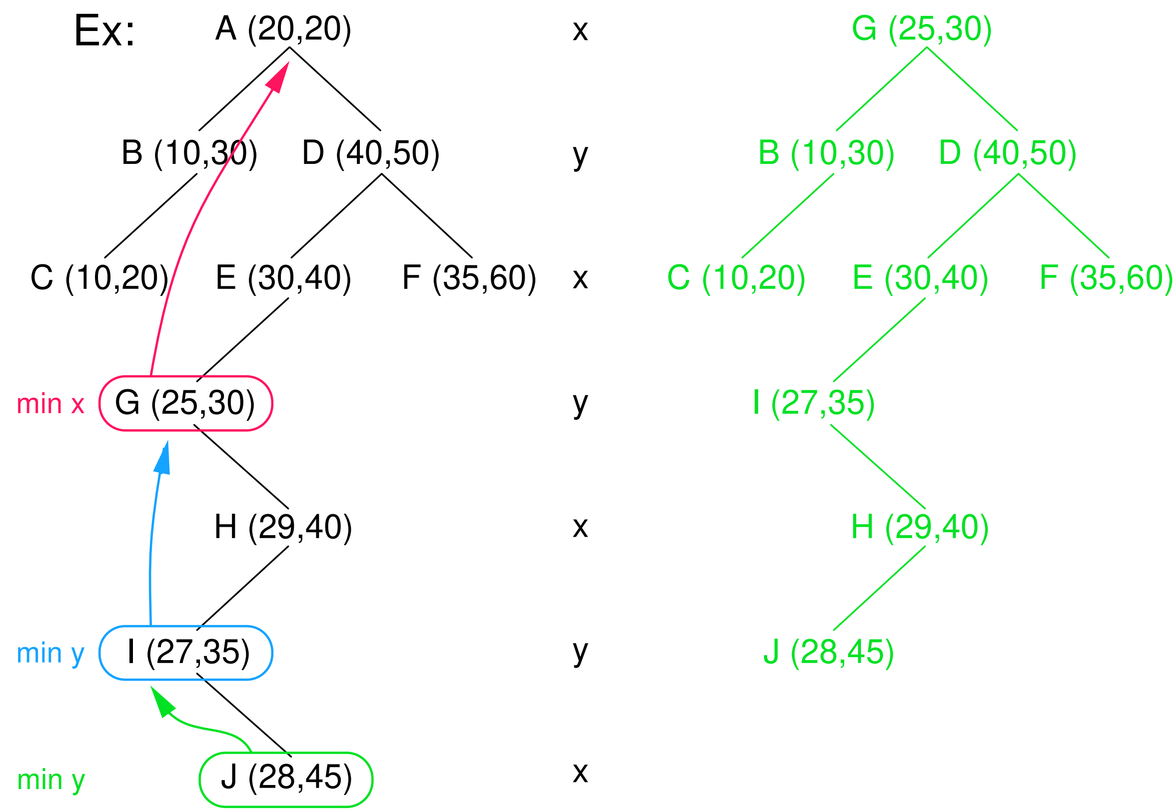


- Analogy with binary search trees implies that we can replace the root by the node in LOSON with the max *x* coordinate value. However, if there is more than one node with the same max value, then after replacement there would be a node in LOSON which is not strictly less than the new root (e.g., nodes B and C)
- What if the HISON of the root is empty?

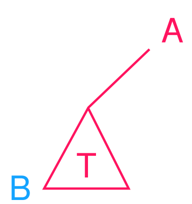


EXAMPLE OF K-D TREE DELETION

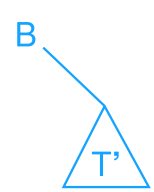
Ex:



- Analogy with binary search trees implies that we can replace the root by the node in LOSON with the max *x* coordinate value. However, if there is more than one node with the same max value, then after replacement there would be a node in LOSON which is not strictly less than the new root (e.g., nodes B and C)
- What if the HISON of the root is empty?
  1. replace root node A with the node B in LOSON having the minimum *x* coordinate value and set the HISON pointer of A to be the old LOSON pointer of A
  2. repeat process for the position of the replacement node

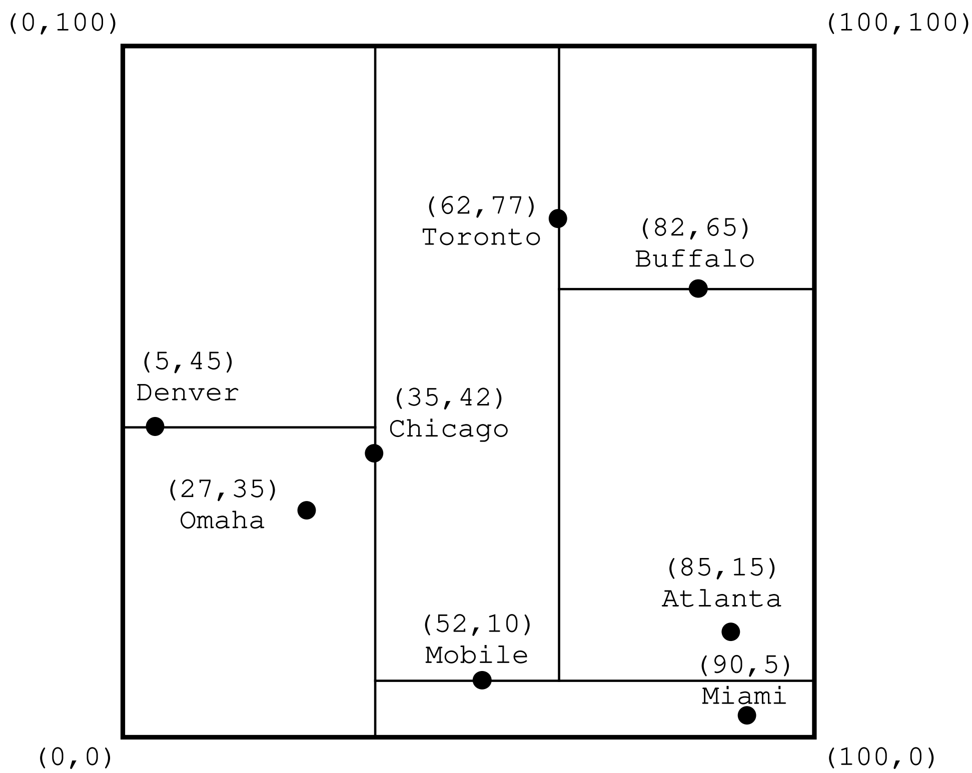


B is the node with the minimum *x* coordinate value and replaces A



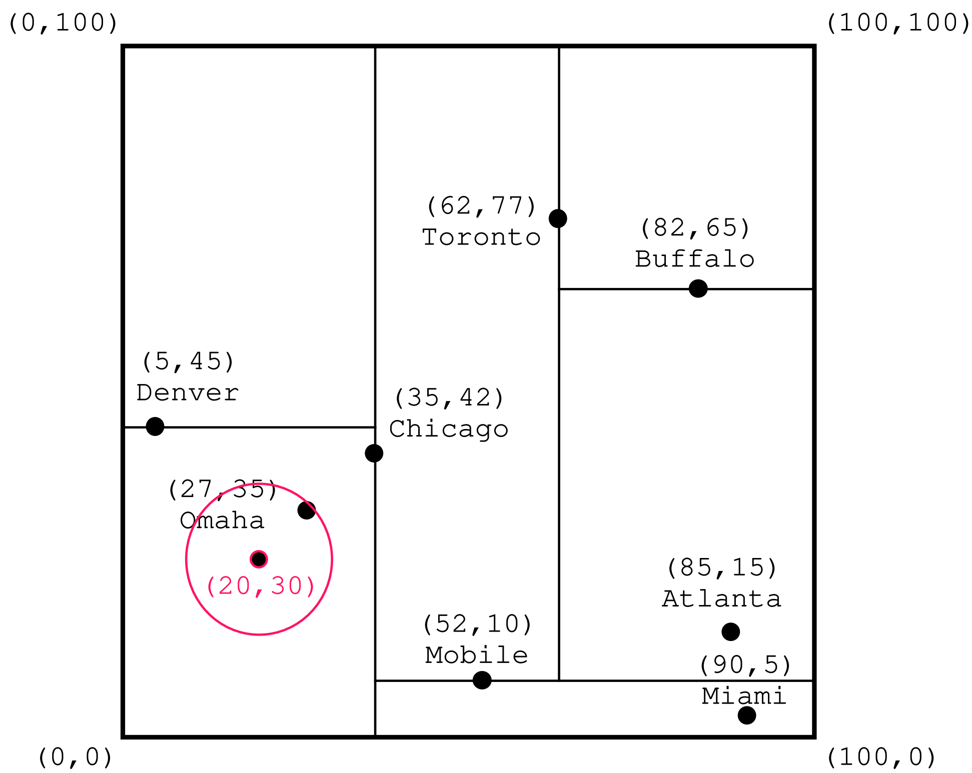
K-D TREE SEARCH

- Search space can be pruned by testing if the search region is completely contained in one of the partitions of the node as now only one subtree must be examined



K-D TREE SEARCH

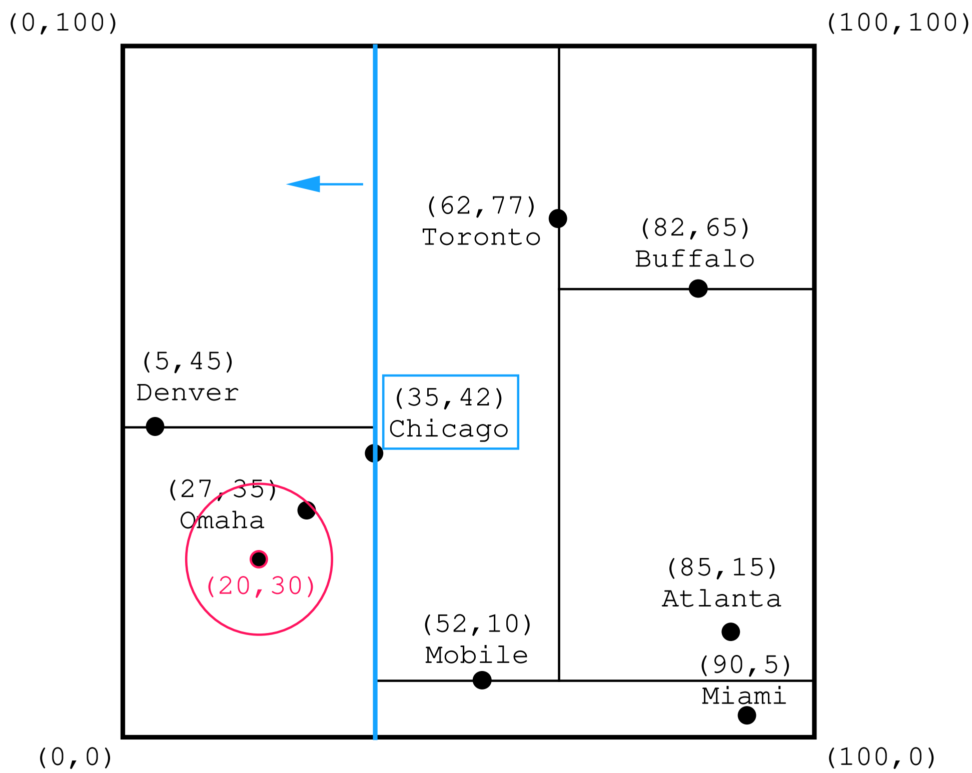
- Search space can be pruned by testing if the search region is completely contained in one of the partitions of the node as now only one subtree must be examined



- Ex: find all points within 10 of (20,30)

K-D TREE SEARCH

- Search space can be pruned by testing if the search region is completely contained in one of the partitions of the node as now only one subtree must be examined

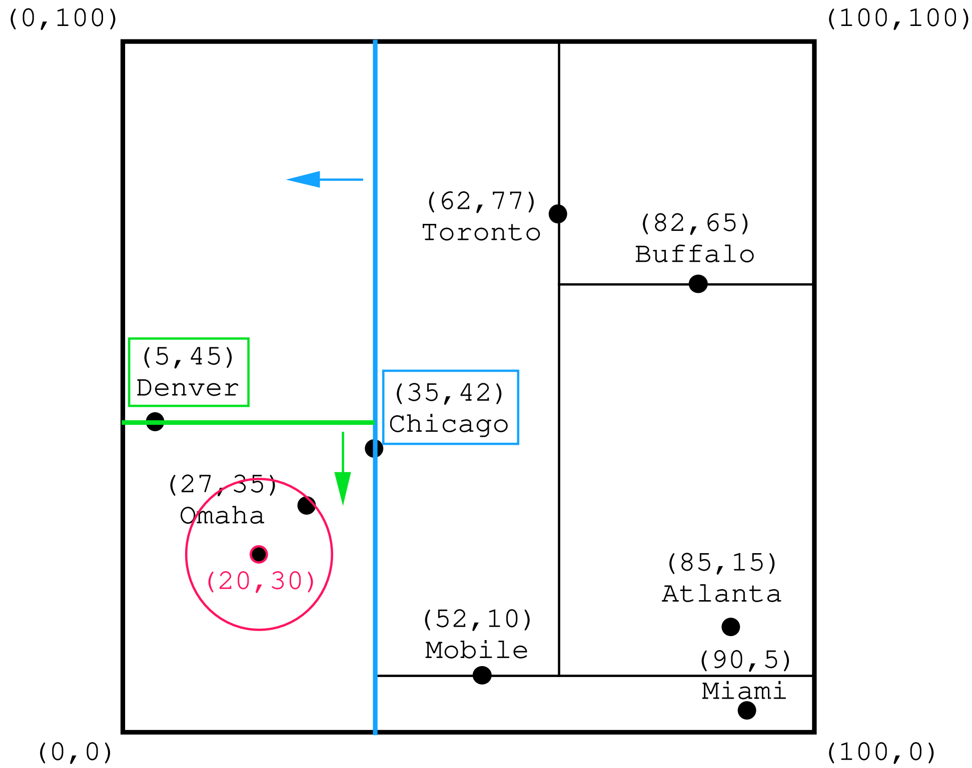


- Ex: find all points within 10 of (20,30)

  1. search the region entirely to the left of `Chicago` (x=35)

K-D TREE SEARCH

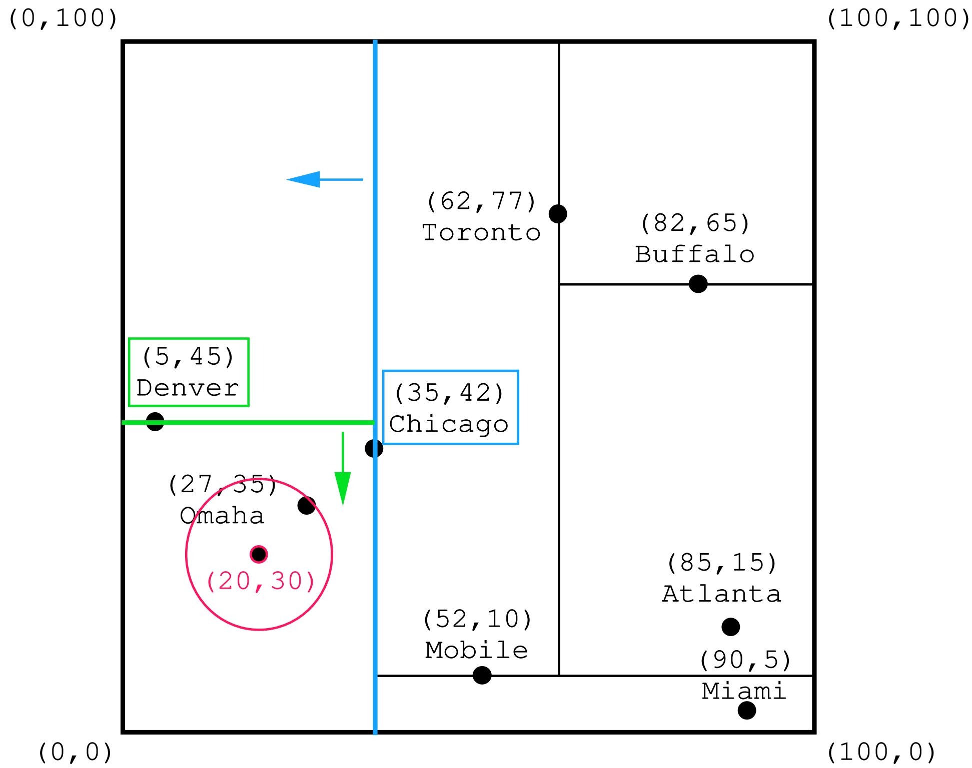
- Search space can be pruned by testing if the search region is completely contained in one of the partitions of the node as now only one subtree must be examined



- Ex: find all points within 10 of (20,30)
  1. search the region entirely to the left of `Chicago` (x=35)
  2. search the region entirely below `Denver` (y=45)

K-D TREE SEARCH

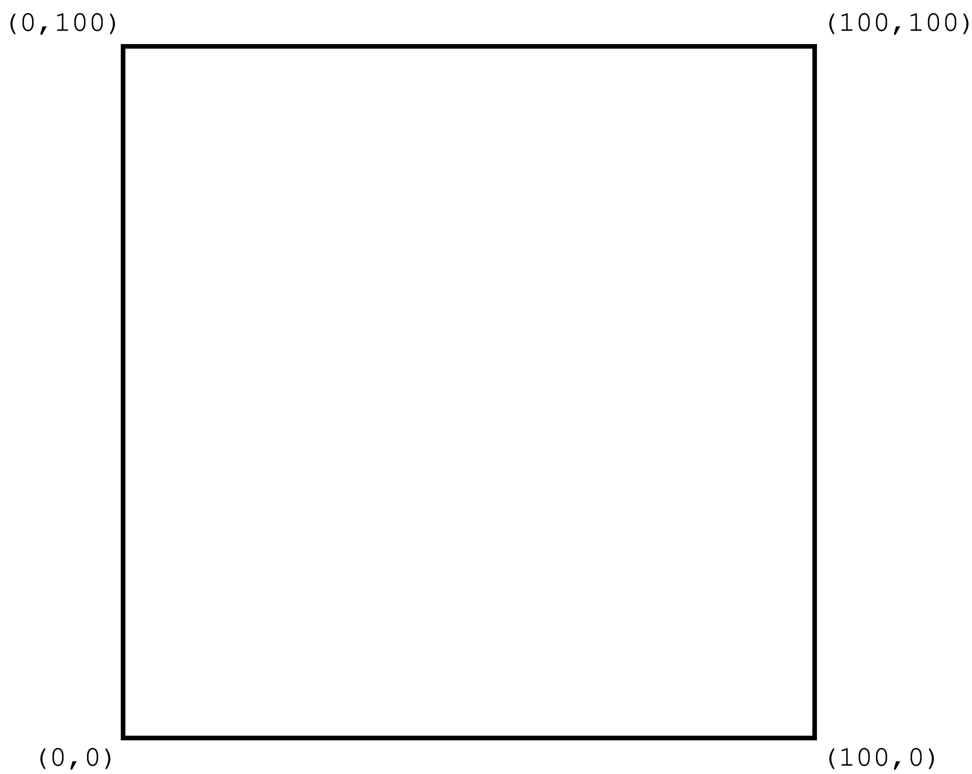
- Search space can be pruned by testing if the search region is completely contained in one of the partitions of the node as now only one subtree must be examined



- Ex: find all points within 10 of (20,30)
  1. search the region entirely to the left of `Chicago` (x=35)
  2. search the region entirely below `Denver` (y=45)
  3. search yields `Omaha`

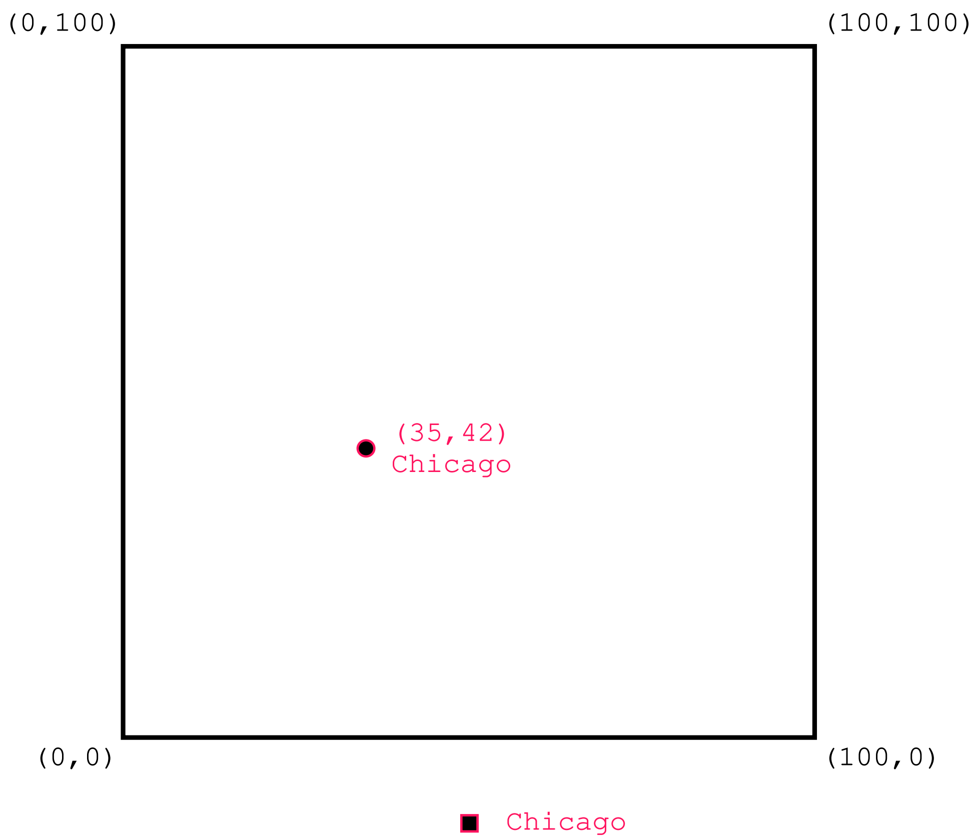
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



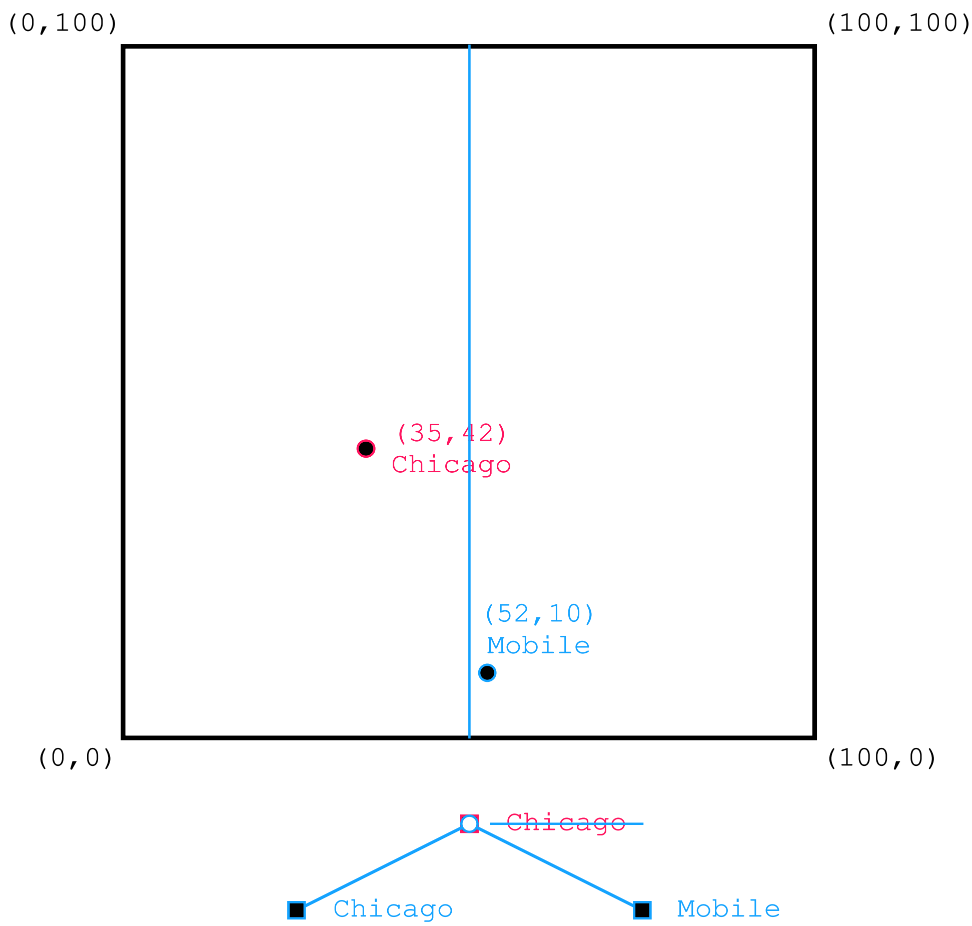
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



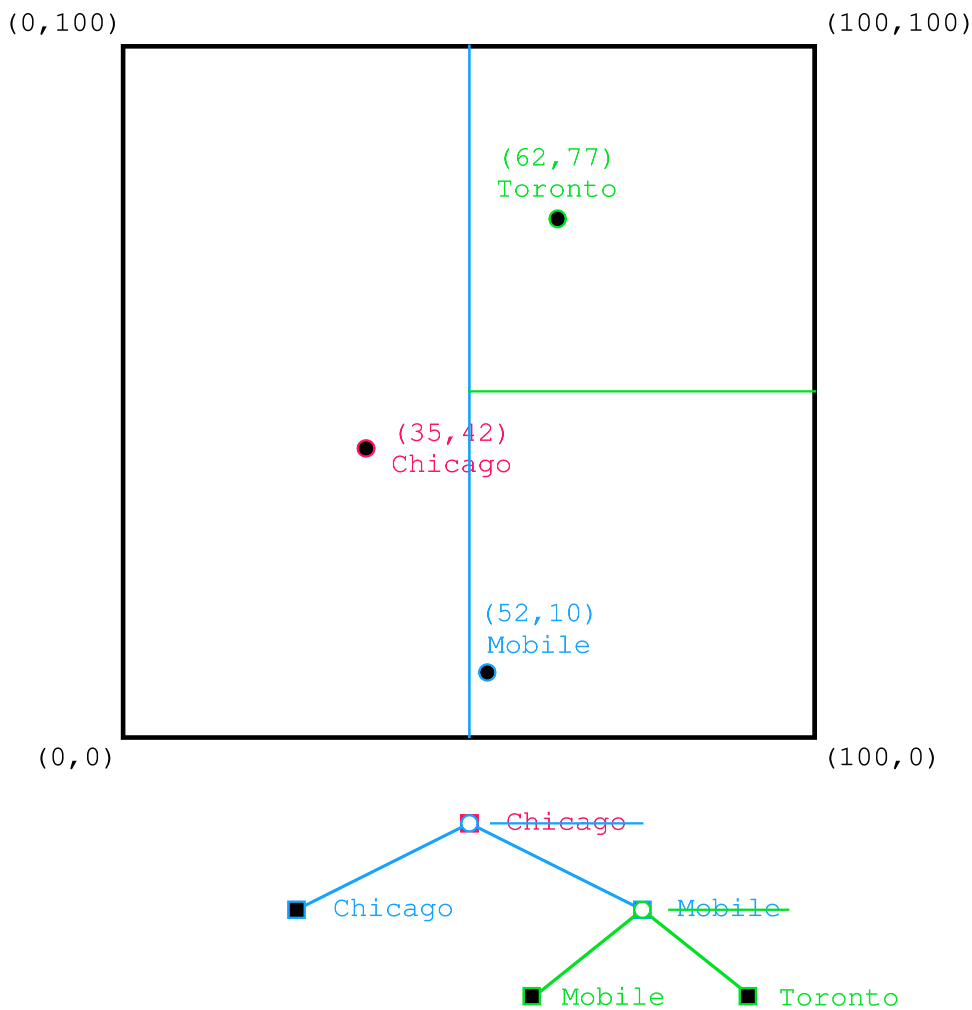
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



PR K-D TREE (Knowlton)

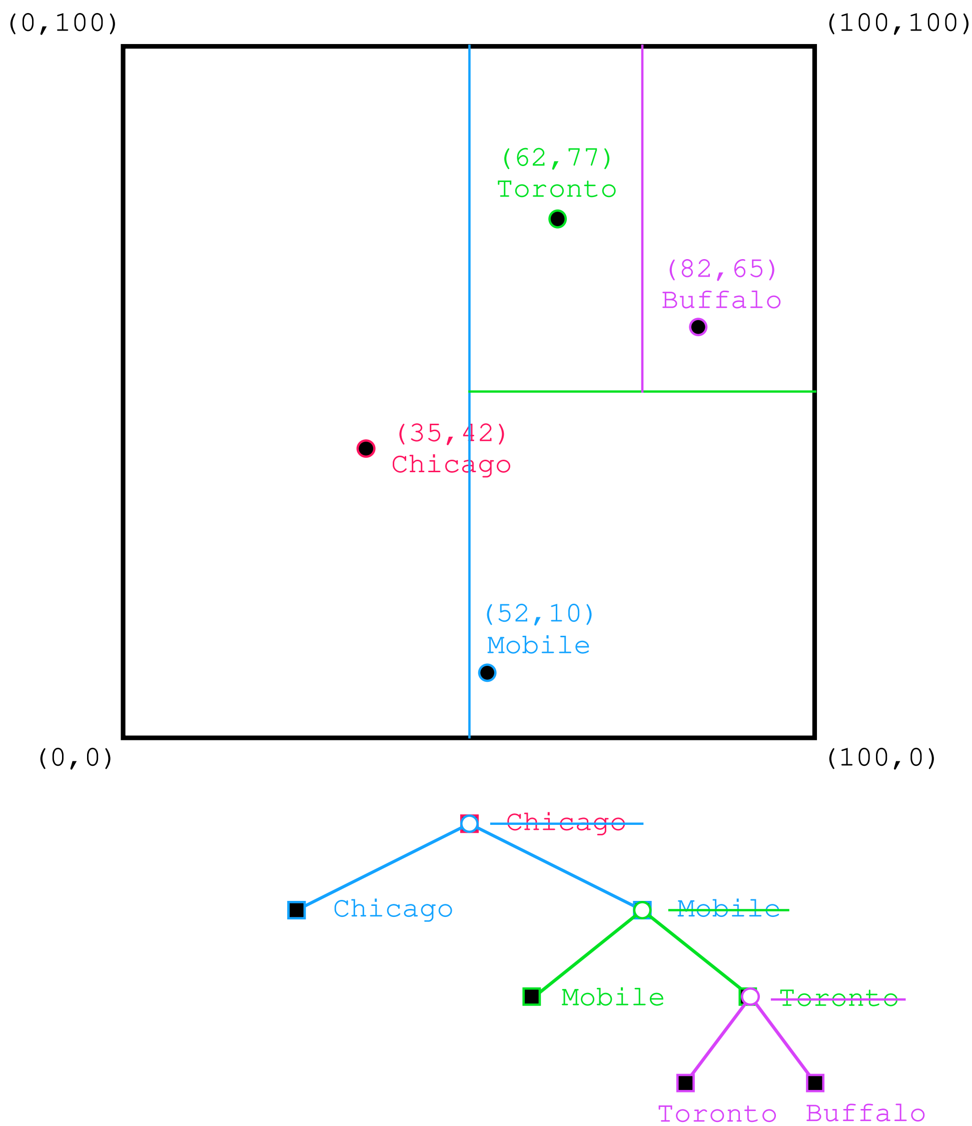
- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



5 4 3 2 1 hp19
v g z r b

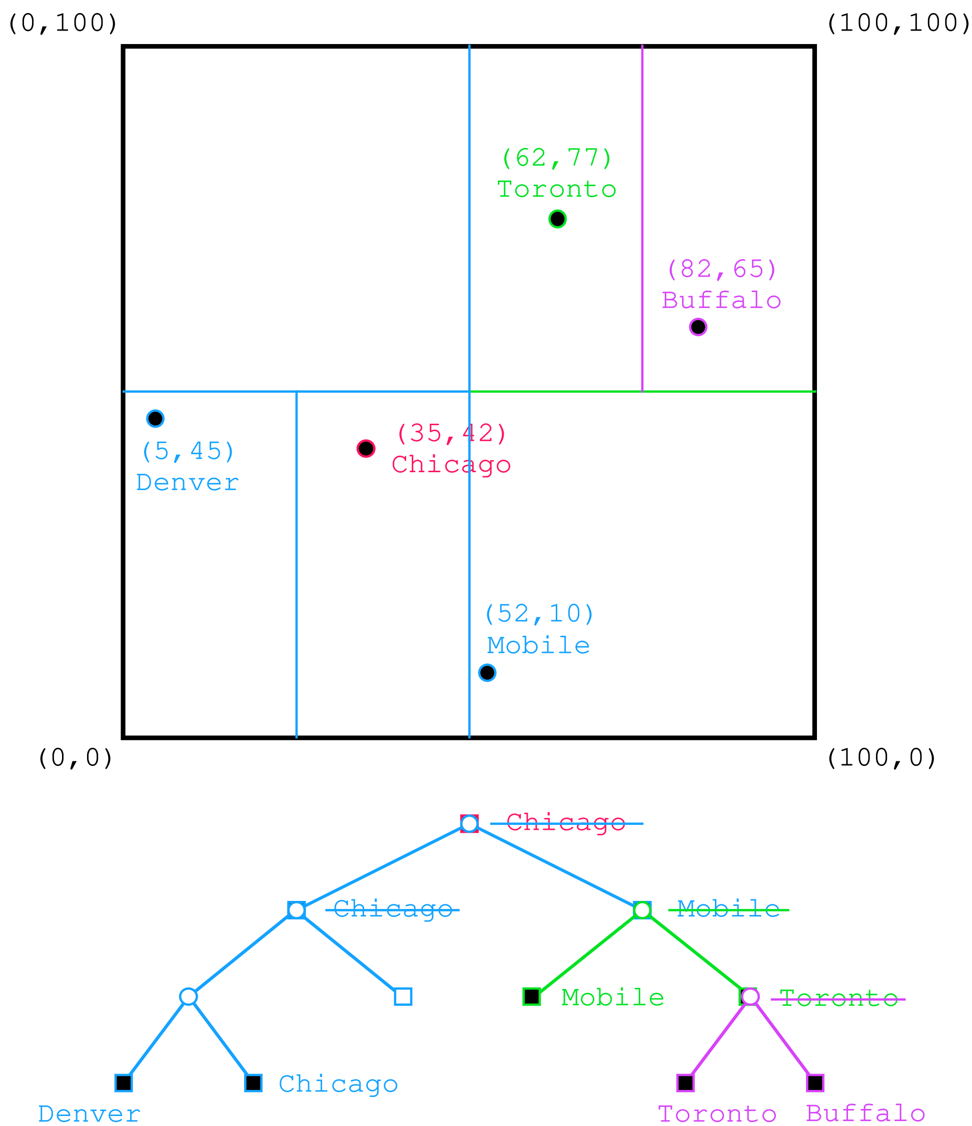
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



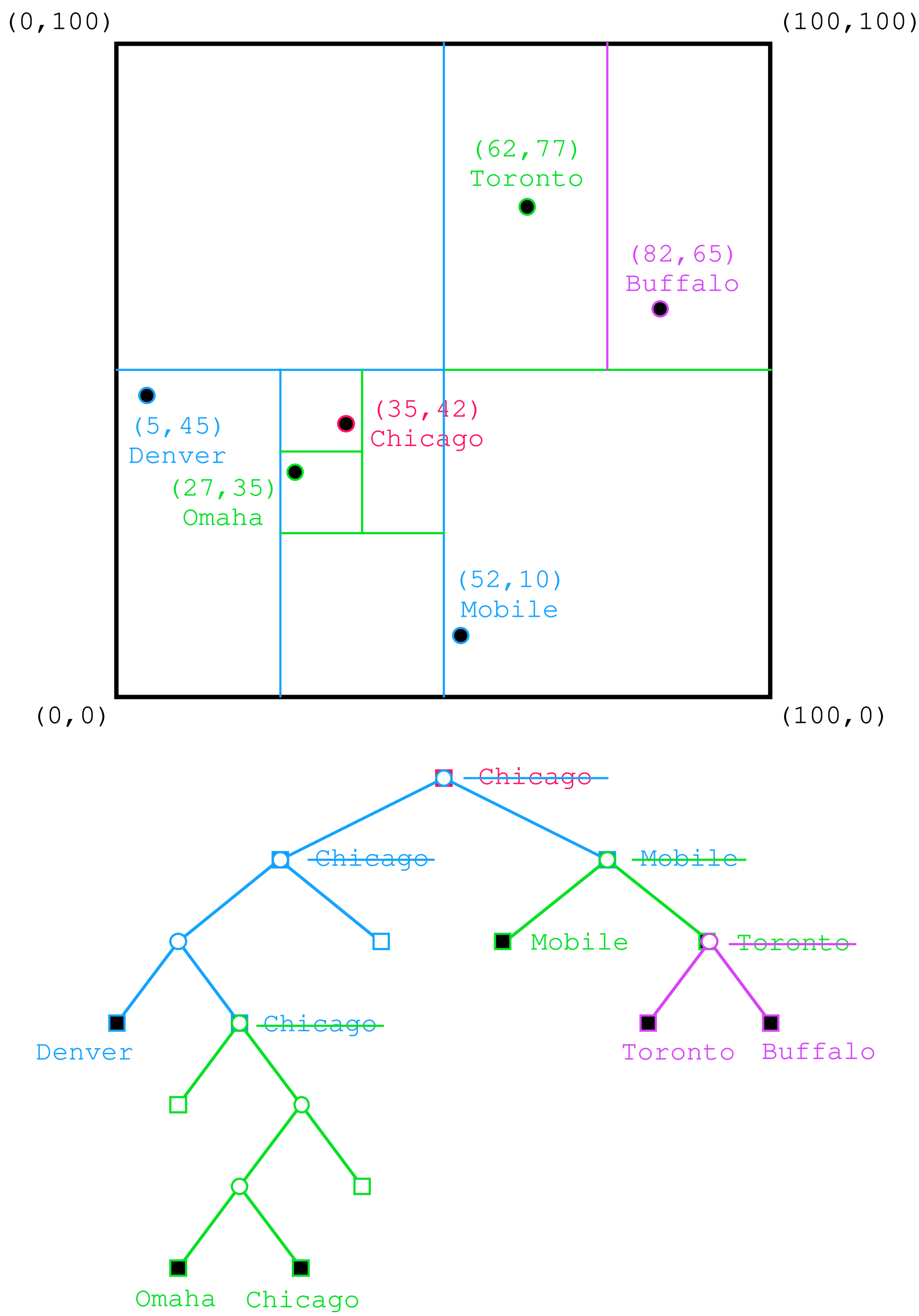
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1

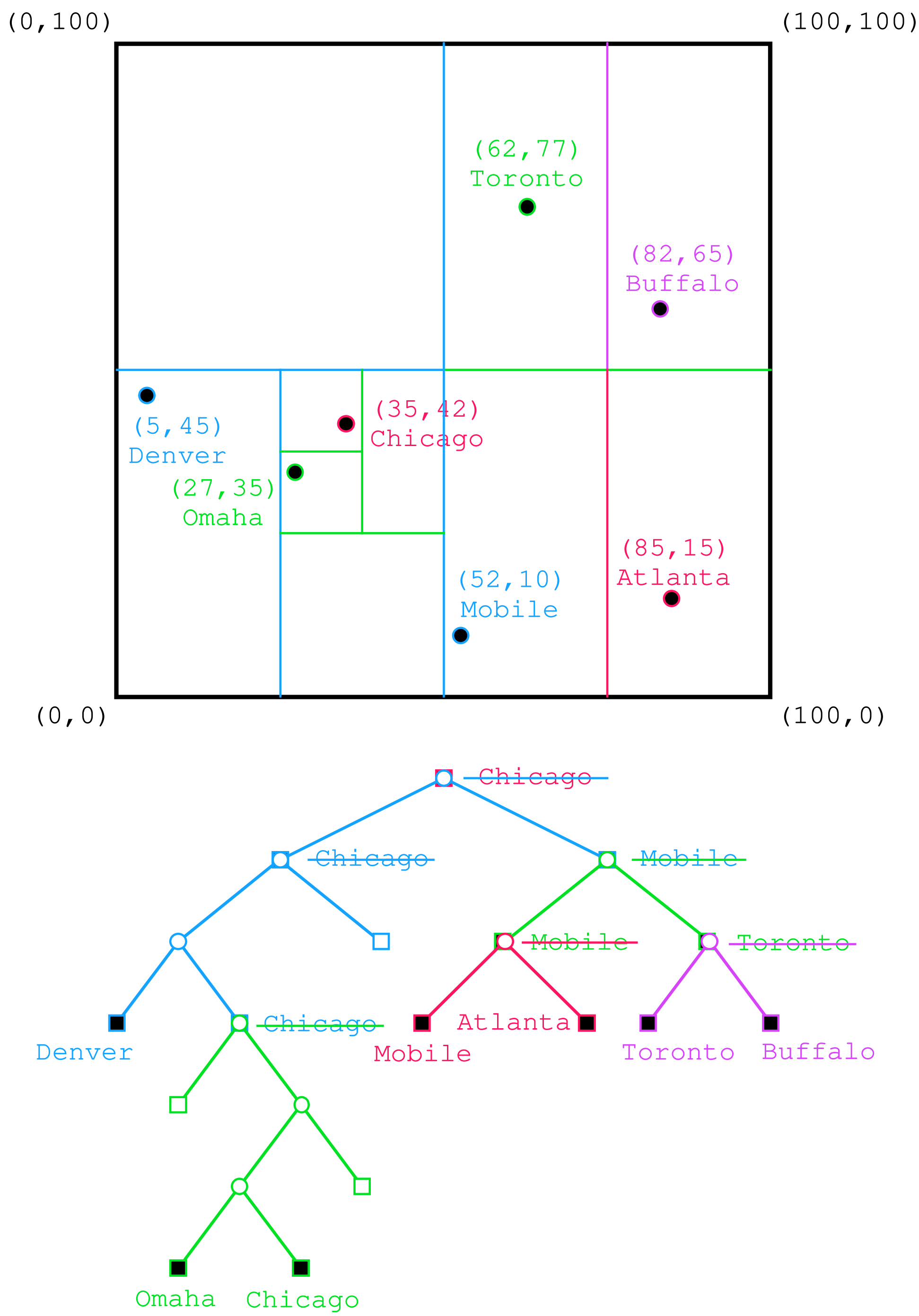


8 7 6 5 4 3 2 1
r g z v g z r b

hp19

PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1

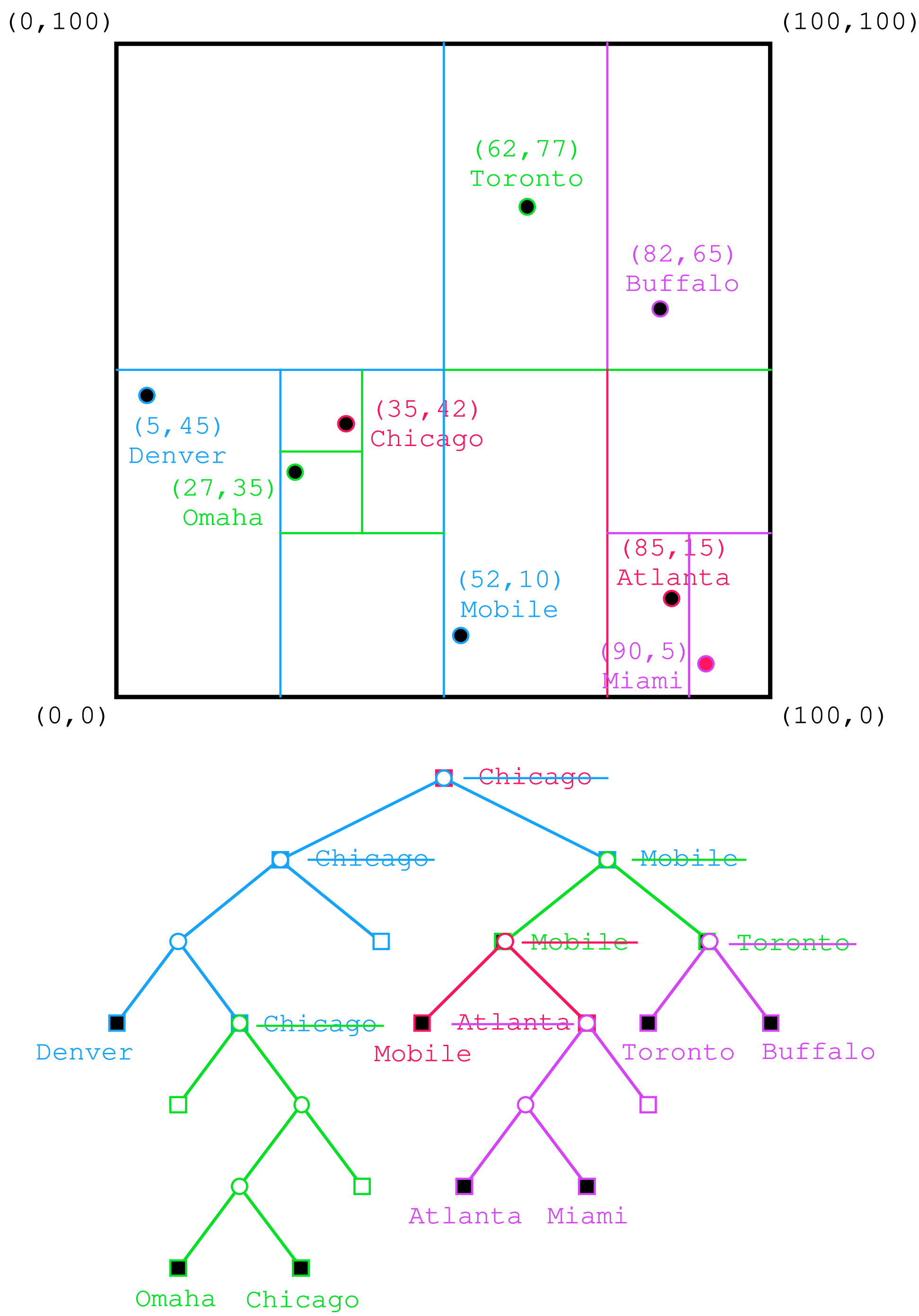


9 8 7 6 5 4 3 2 1
v r g z v g z r b

hp19

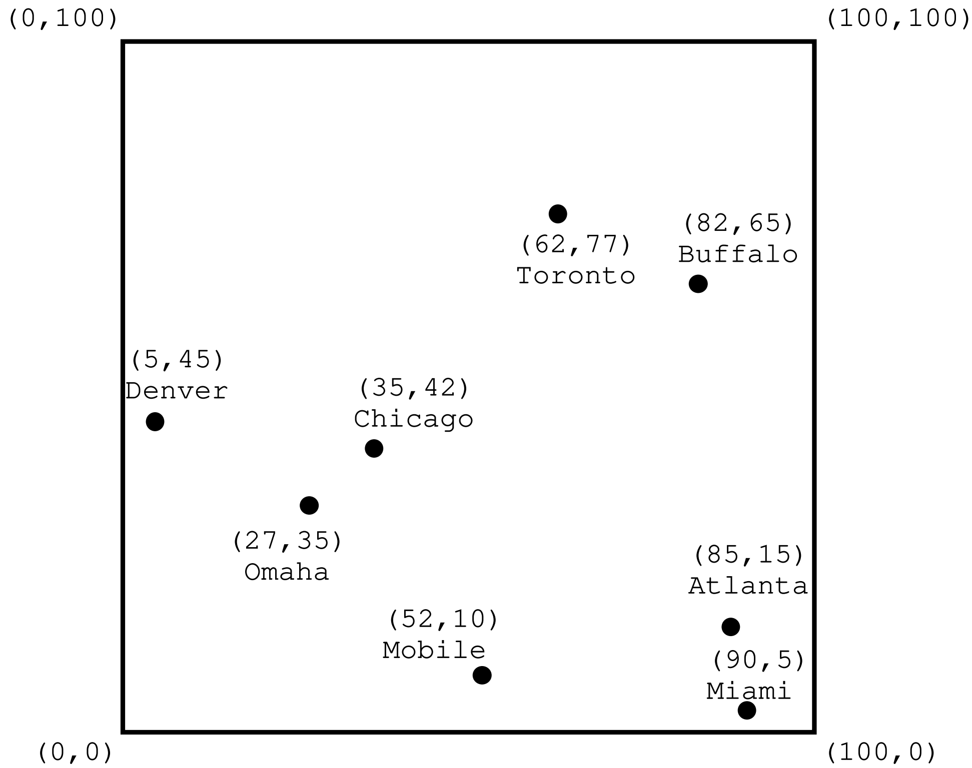
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



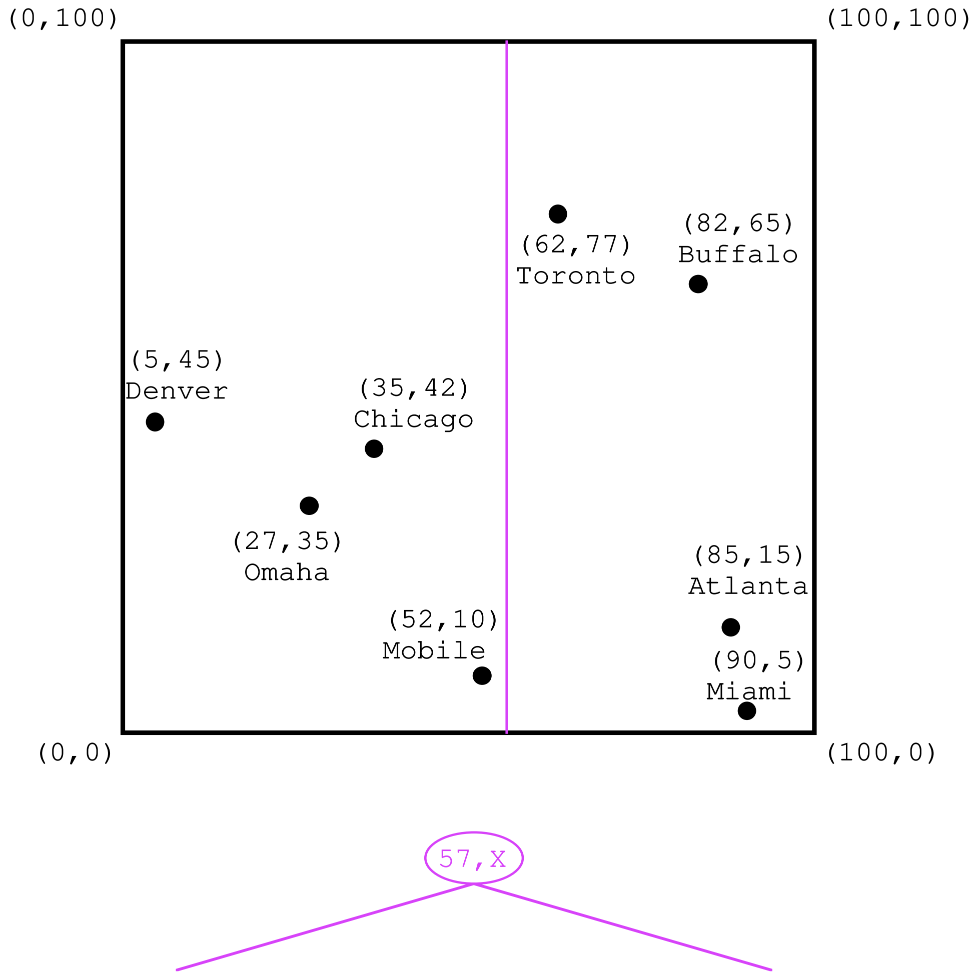
ADAPTIVE K-D TREE

- Data is only stored in terminal nodes
- An interior node contains the median of the set as the discriminator
- The discriminator key is the one for which the spread of the values of the key is a maximum



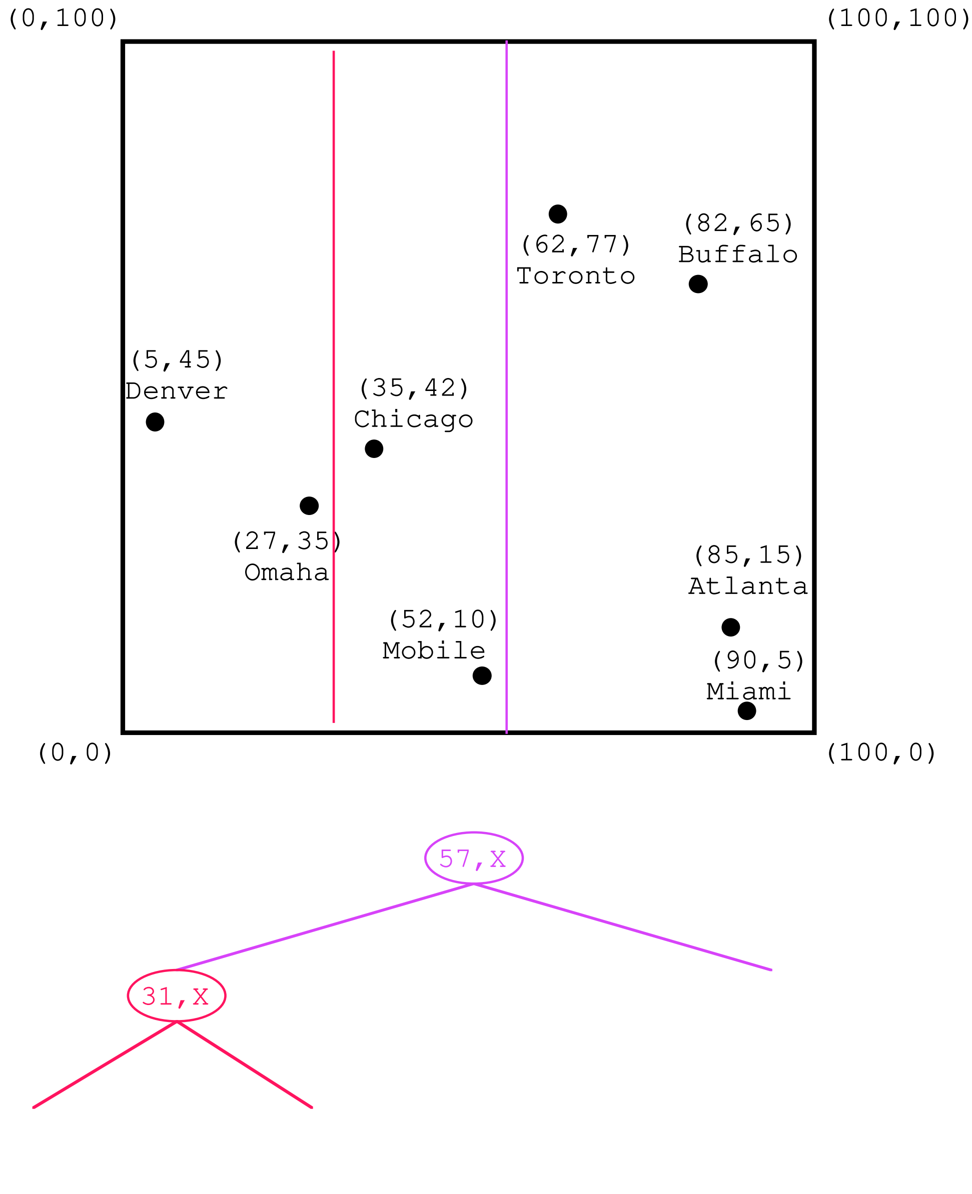
ADAPTIVE K-D TREE

- Data is only stored in terminal nodes
- An interior node contains the median of the set as the discriminator
- The discriminator key is the one for which the spread of the values of the key is a maximum



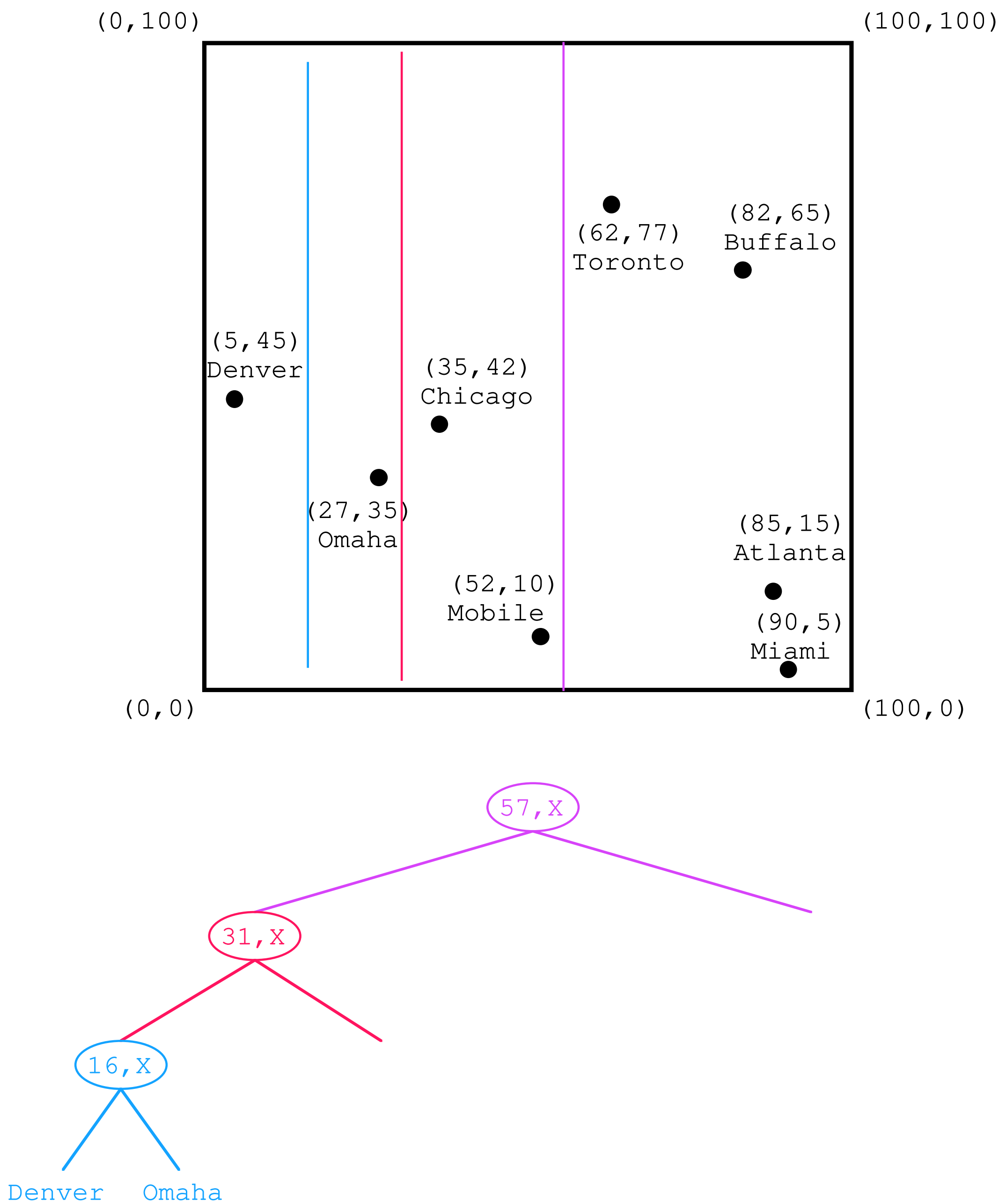
ADAPTIVE K-D TREE

- Data is only stored in terminal nodes
- An interior node contains the median of the set as the discriminator
- The discriminator key is the one for which the spread of the values of the key is a maximum



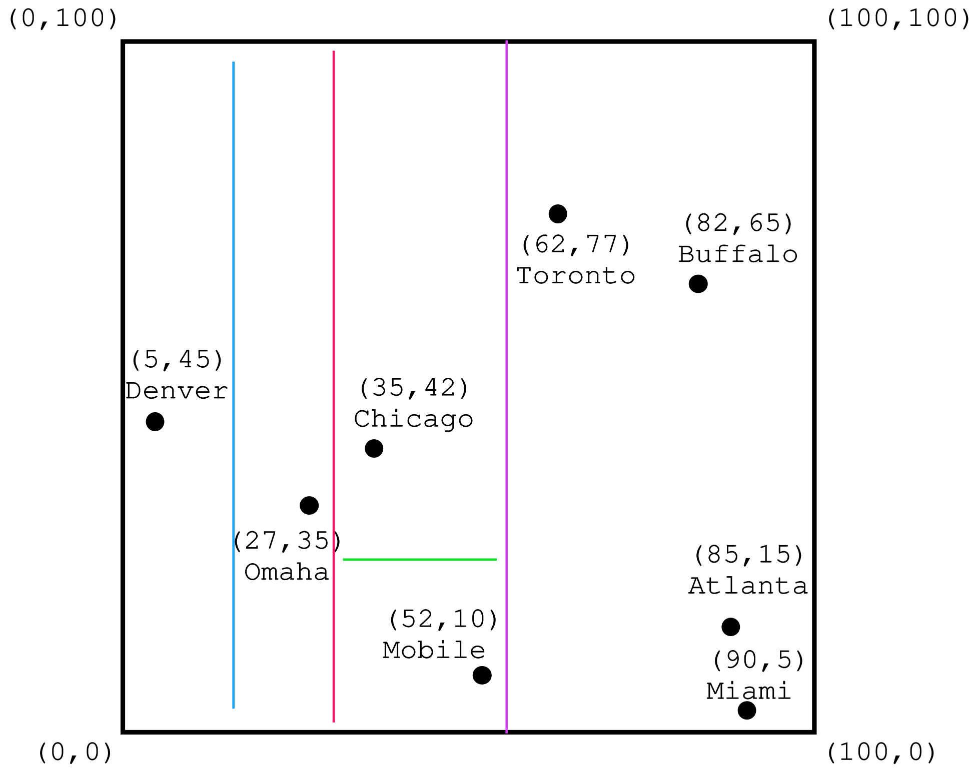
ADAPTIVE K-D TREE

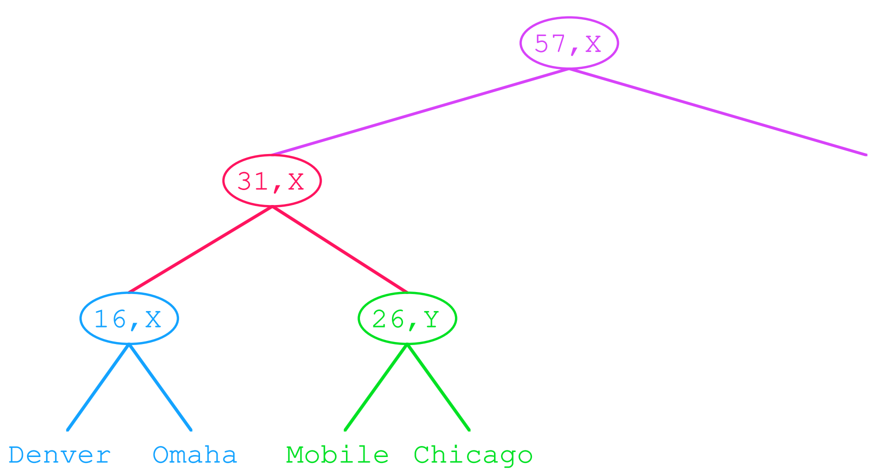
- Data is only stored in terminal nodes
- An interior node contains the median of the set as the discriminator
- The discriminator key is the one for which the spread of the values of the key is a maximum



ADAPTIVE K-D TREE

- Data is only stored in terminal nodes
- An interior node contains the median of the set as the discriminator
- The discriminator key is the one for which the spread of the values of the key is a maximum

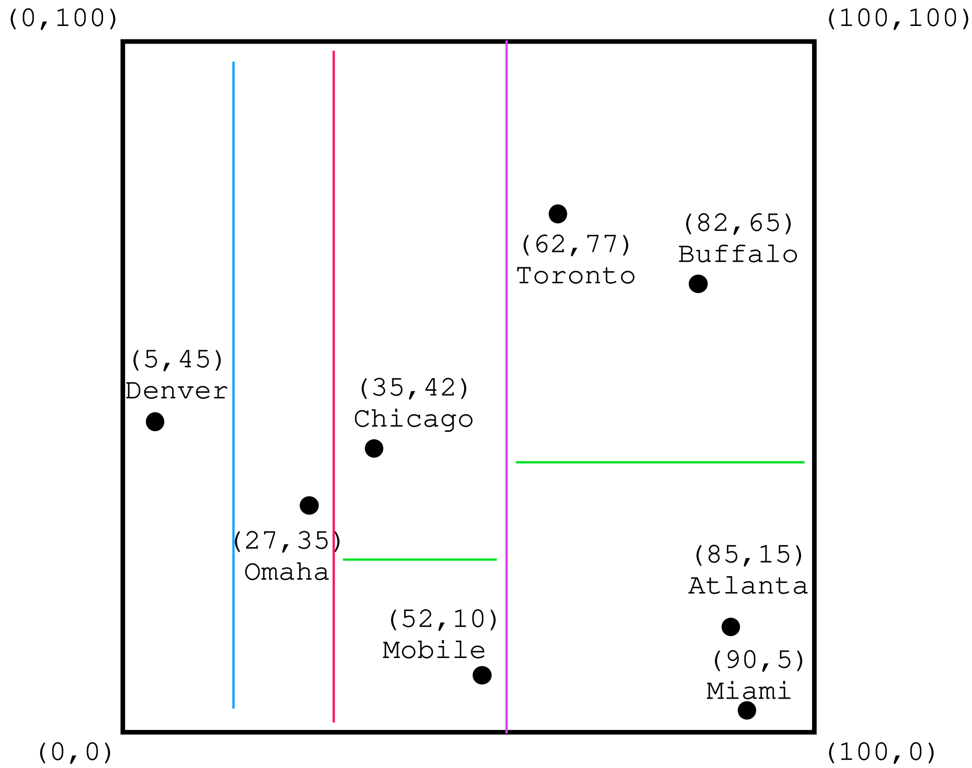


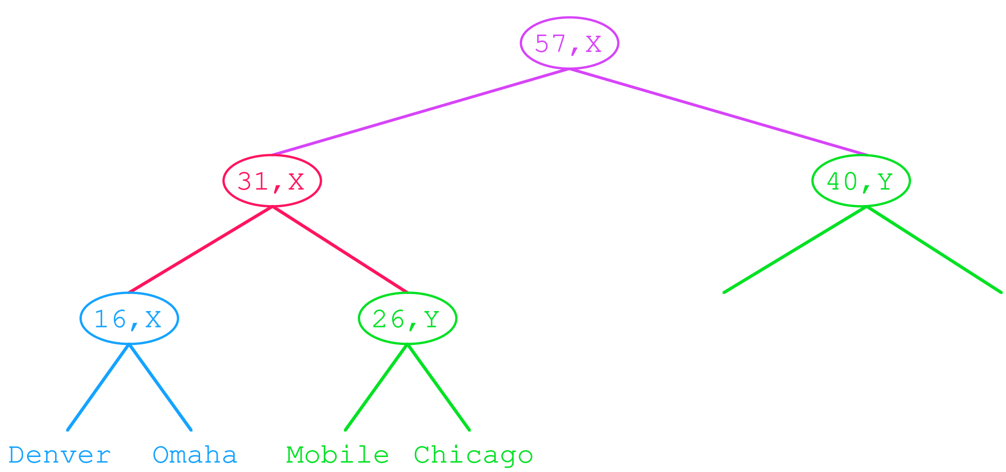


6 5 4 3 2 1 hp20
g g z r v b

ADAPTIVE K-D TREE

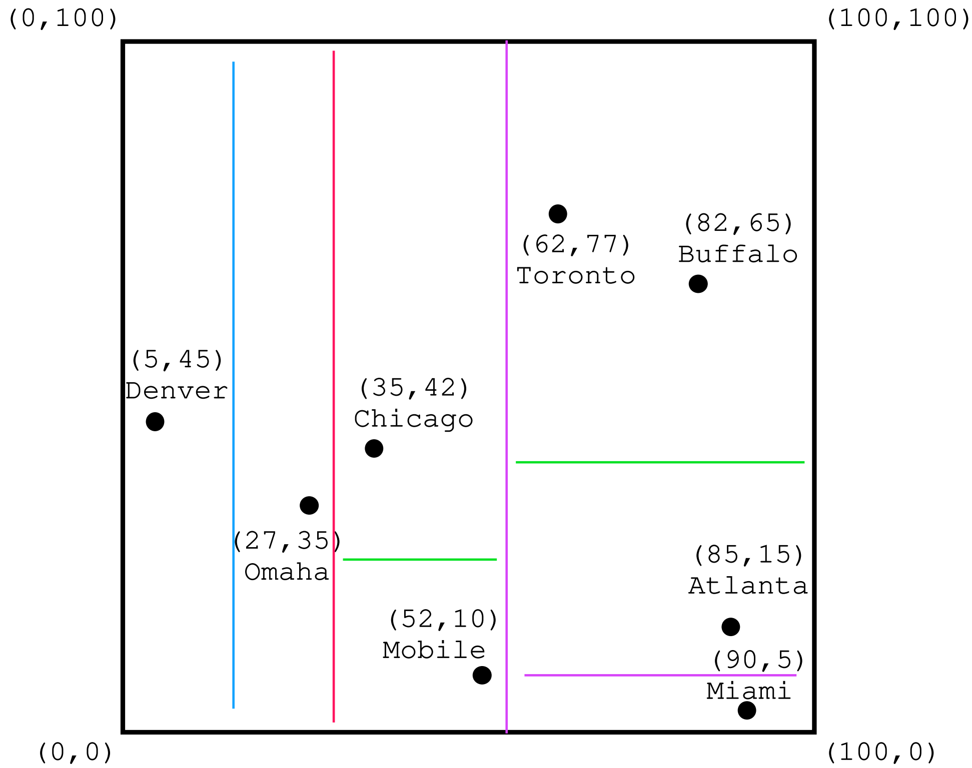
- Data is only stored in terminal nodes
- An interior node contains the median of the set as the discriminator
- The discriminator key is the one for which the spread of the values of the key is a maximum

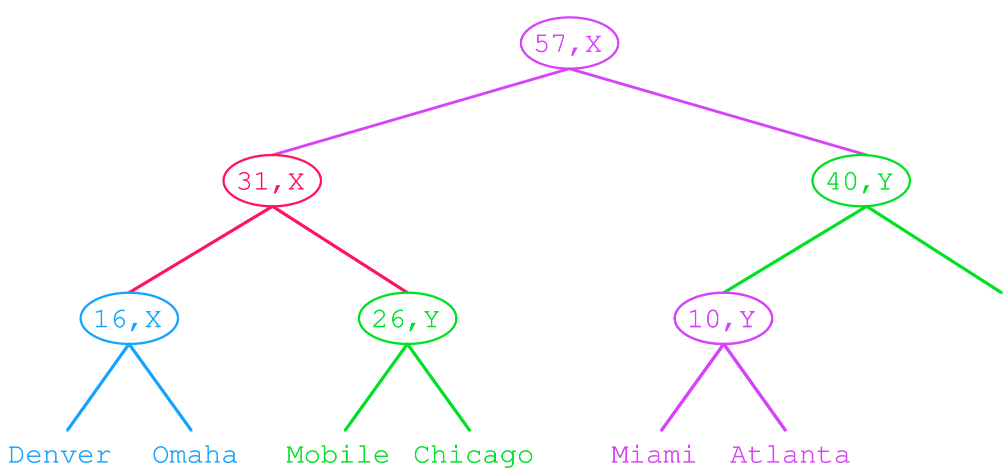




ADAPTIVE K-D TREE

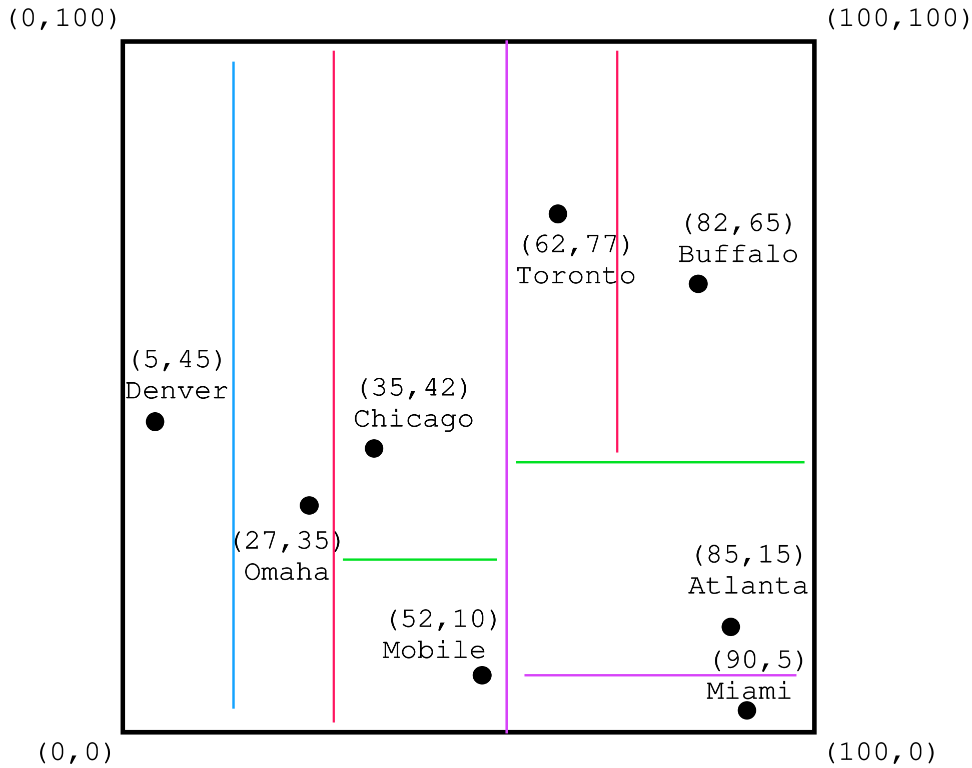
- Data is only stored in terminal nodes
- An interior node contains the median of the set as the discriminator
- The discriminator key is the one for which the spread of the values of the key is a maximum

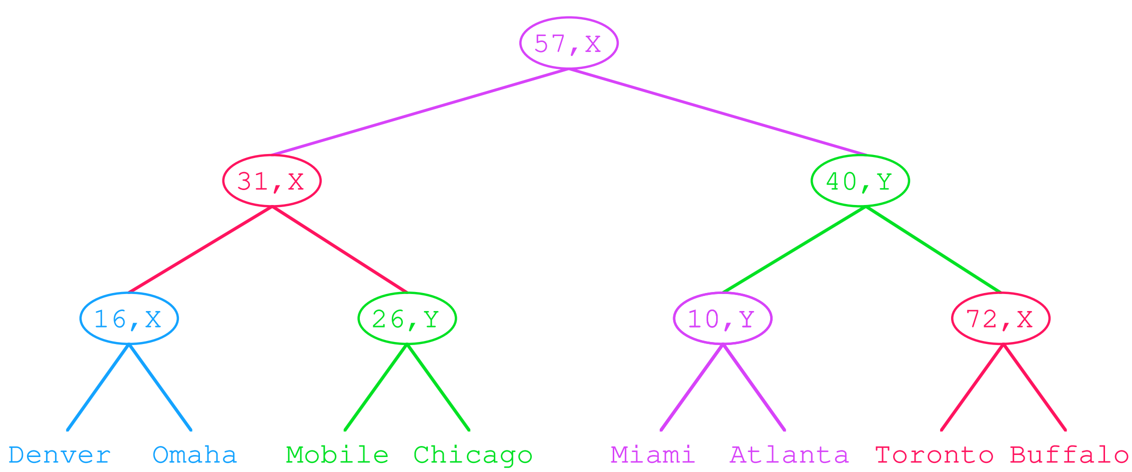




ADAPTIVE K-D TREE

- Data is only stored in terminal nodes
- An interior node contains the median of the set as the discriminator
- The discriminator key is the one for which the spread of the values of the key is a maximum



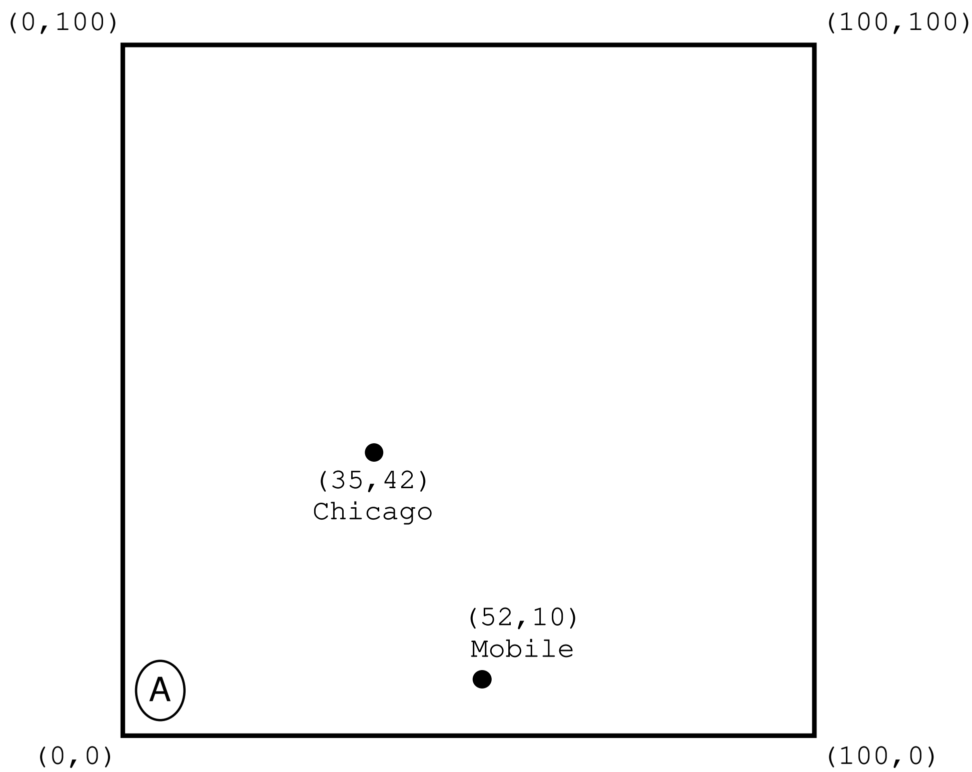


GRID FILE (Nievergelt,Hinterberger,Sevcik)

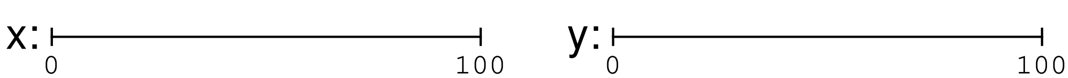
- Two level grid for storing points
- Uses a grid directory (a 2d array of grid blocks) on disk that contains the address of the bucket (i.e., page) that contains the data associated with the grid block
- Linear scales (a pair of 1d arrays) in core that access the grid block in the grid directory (on disk) that is associated with a particular point thereby enabling the decomposition of the space to be arbitrary
- Guarantees access to any record with two disk operations — one for each level of the grid
  1. access the grid block
  2. access the bucket
- Each bucket has finite capacity
- Partition upon overflow
  1. bucket partition — overflowing bucket is associated with more than one grid block
  2. grid partition — overflowing bucket is associated with just one grid block
- Splitting policies
  1. split at midpoint and cycle through attributes
  2. adaptive
     - increases granularity of frequently queried attributes
     - favors some attributes over others

GRID FILE EXAMPLE

- Assume bucket size = 2



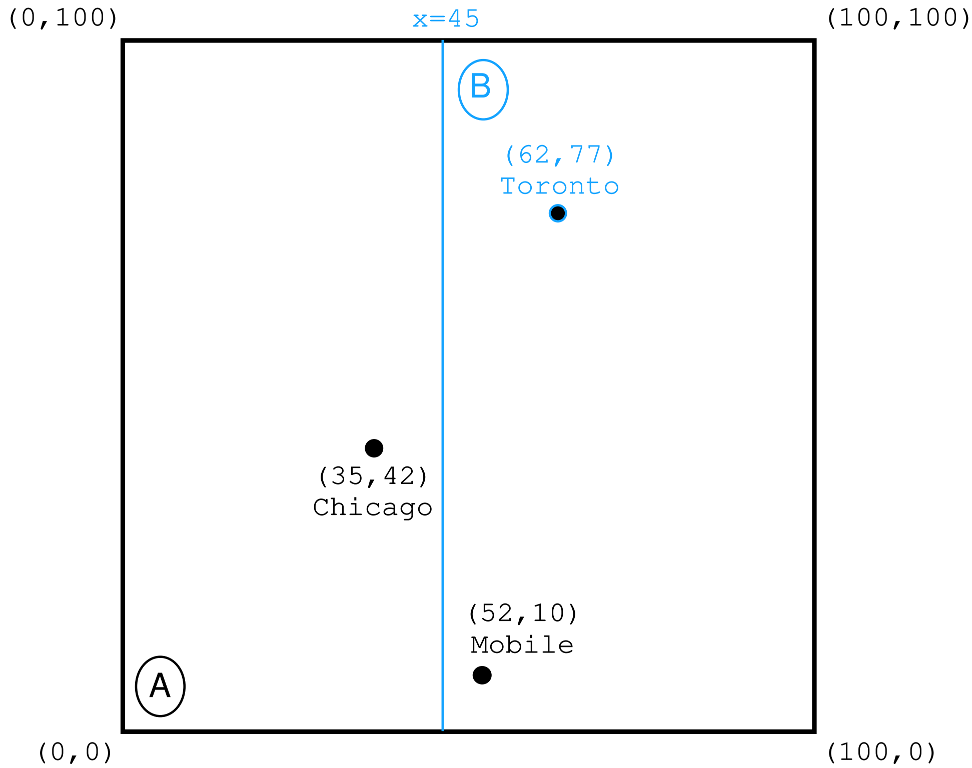
Linear scales



1. Initially `Chicago` and `Mobile` in bucket `A`

GRID FILE EXAMPLE

- Assume bucket size = 2

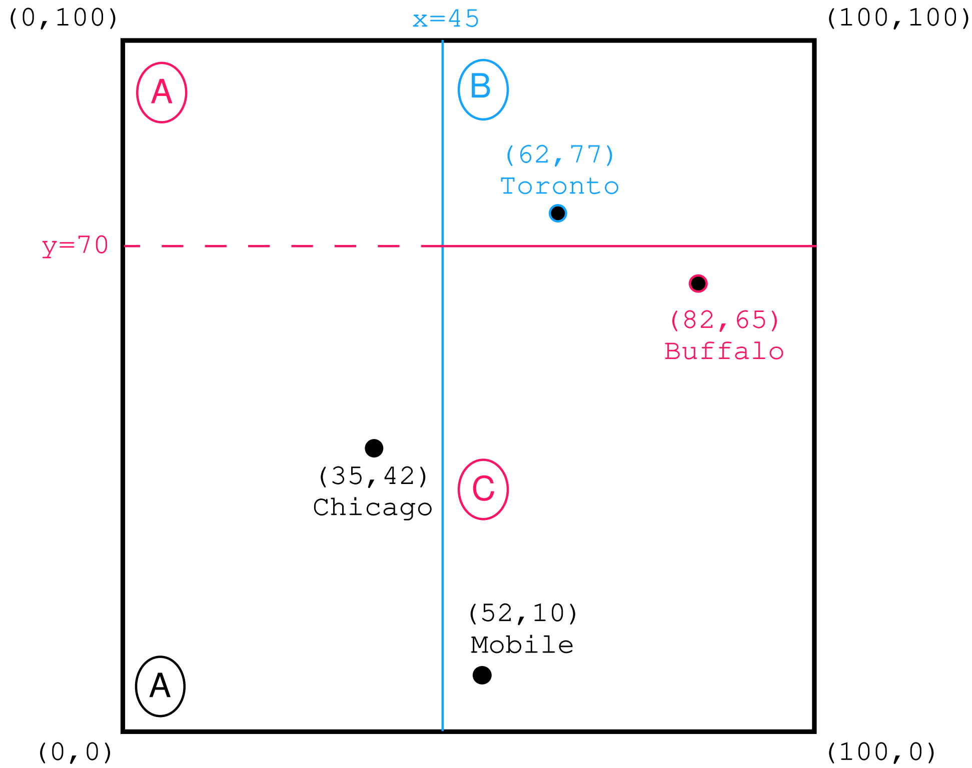


Linear scales

1. Initially Chicago and Mobile in bucket A
2. Insert Toronto causing a grid partition yielding bucket B

GRID FILE EXAMPLE

- Assume bucket size = 2

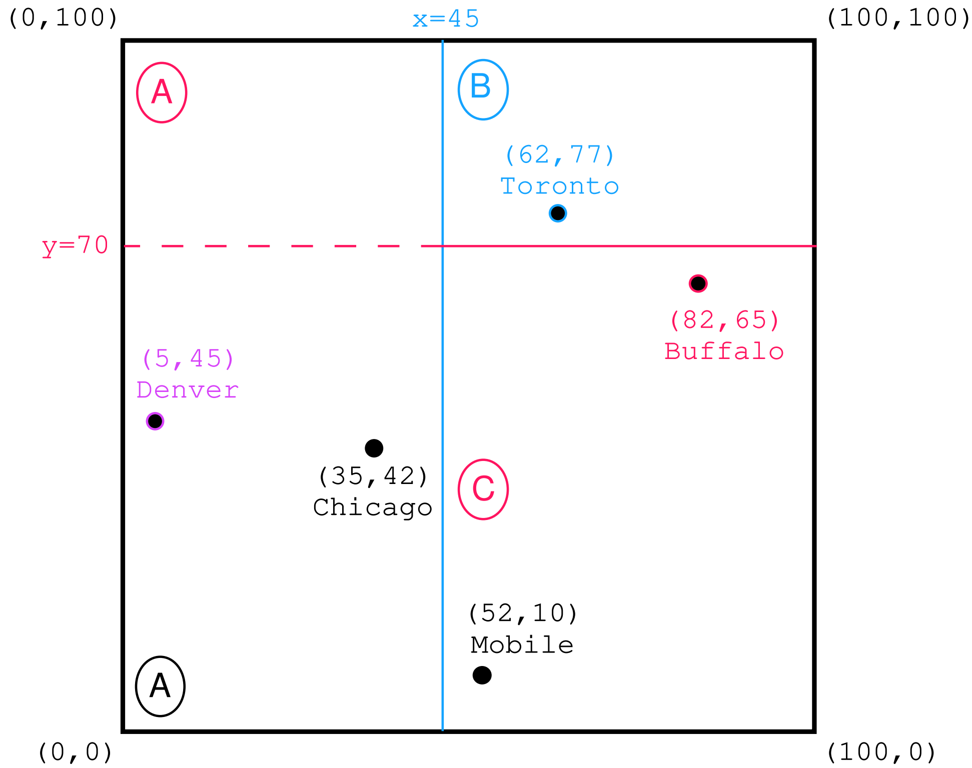


Linear scales

1. Initially `Chicago` and `Mobile` in bucket A
2. Insert `Toronto` causing a grid partition yielding bucket B
3. Insert `Buffalo` causing a grid partition yielding bucket C

GRID FILE EXAMPLE

- Assume bucket size = 2



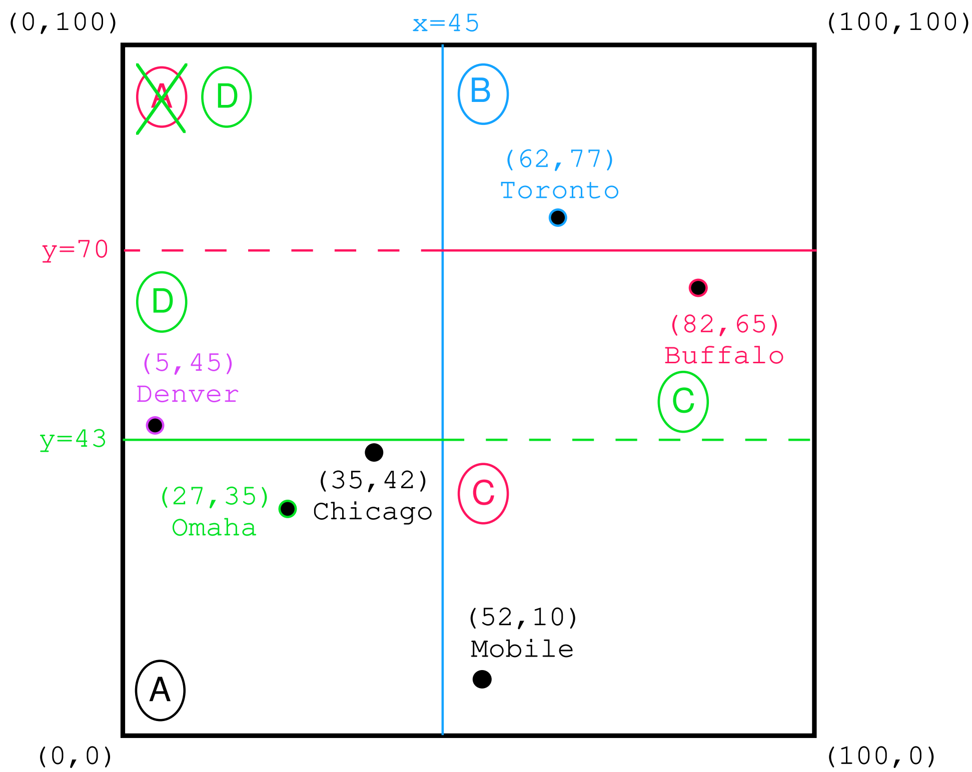
Linear scales

1. Initially `Chicago` and `Mobile` in bucket A
2. Insert `Toronto` causing a grid partition yielding bucket B
3. Insert `Buffalo` causing a grid partition yielding bucket C
4. Insert `Denver` causing no change

5 4 3 2 1 hp22
g v r z b

GRID FILE EXAMPLE

- Assume bucket size = 2

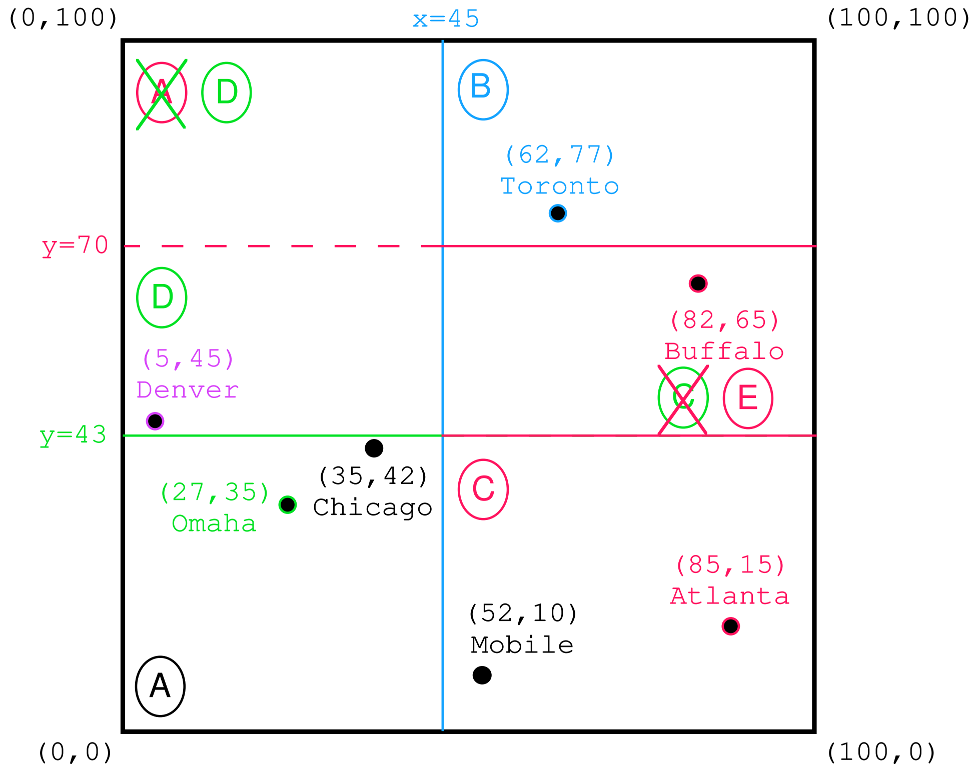


Linear scales

1. Initially `Chicago` and `Mobile` in bucket A
2. Insert `Toronto` causing a grid partition yielding bucket B
3. Insert `Buffalo` causing a grid partition yielding bucket C
4. Insert `Denver` causing no change
5. Insert `Omaha` causing a grid partition yielding bucket D

GRID FILE EXAMPLE

- Assume bucket size = 2

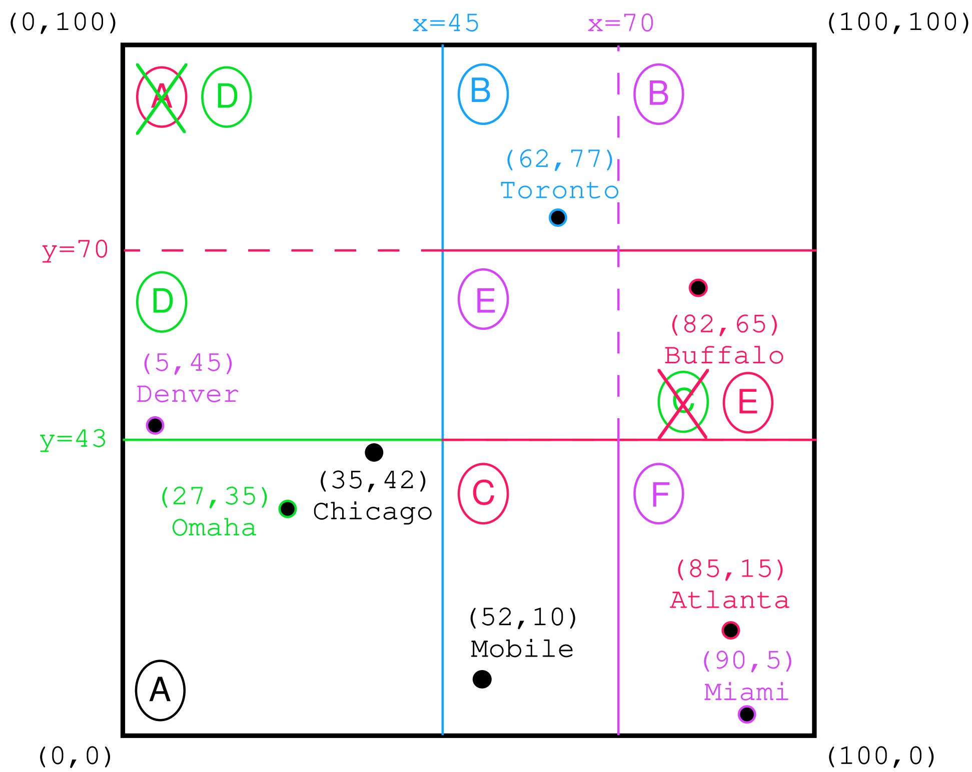


Linear scales

1. Initially `Chicago` and `Mobile` in bucket A
2. Insert `Toronto` causing a grid partition yielding bucket B
3. Insert `Buffalo` causing a grid partition yielding bucket C
4. Insert `Denver` causing no change
5. Insert `Omaha` causing a grid partition yielding bucket D
6. Insert `Atlanta` causing a bucket partition yielding bucket E

GRID FILE EXAMPLE

- Assume bucket size = 2

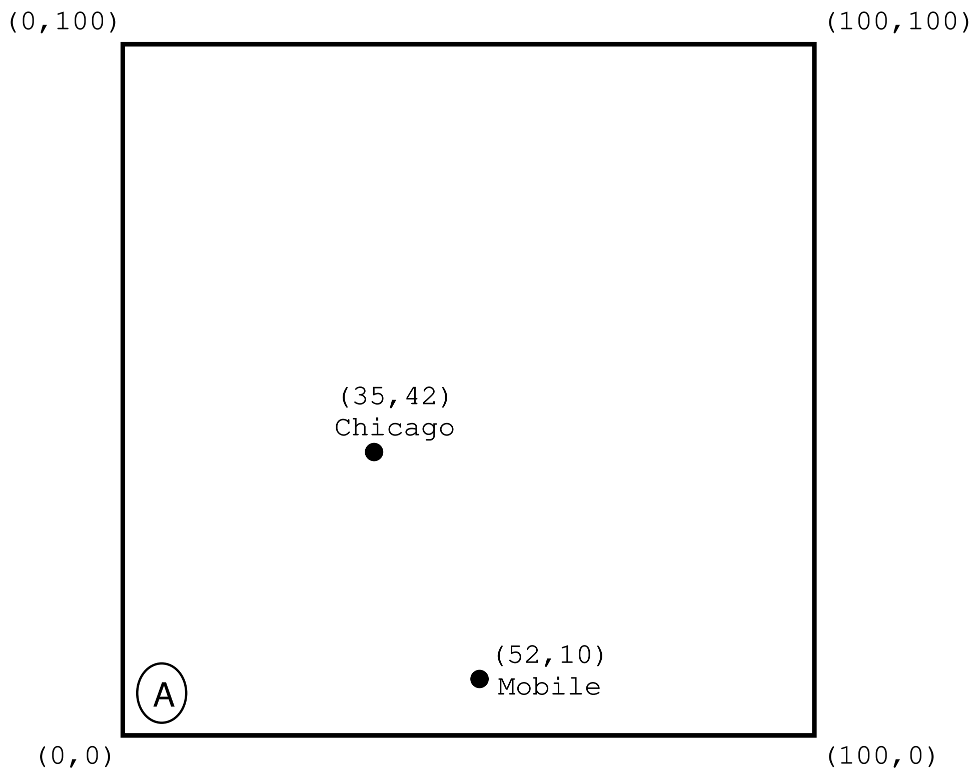


Linear scales

1. Initially Chicago and Mobile in bucket A
2. Insert Toronto causing a grid partition yielding bucket B
3. Insert Buffalo causing a grid partition yielding bucket C
4. Insert Denver causing no change
5. Insert Omaha causing a grid partition yielding bucket D
6. Insert Atlanta causing a bucket partition yielding bucket E
7. Insert Miami causing a grid partition yielding bucket F

EXCELL (Tamminen)

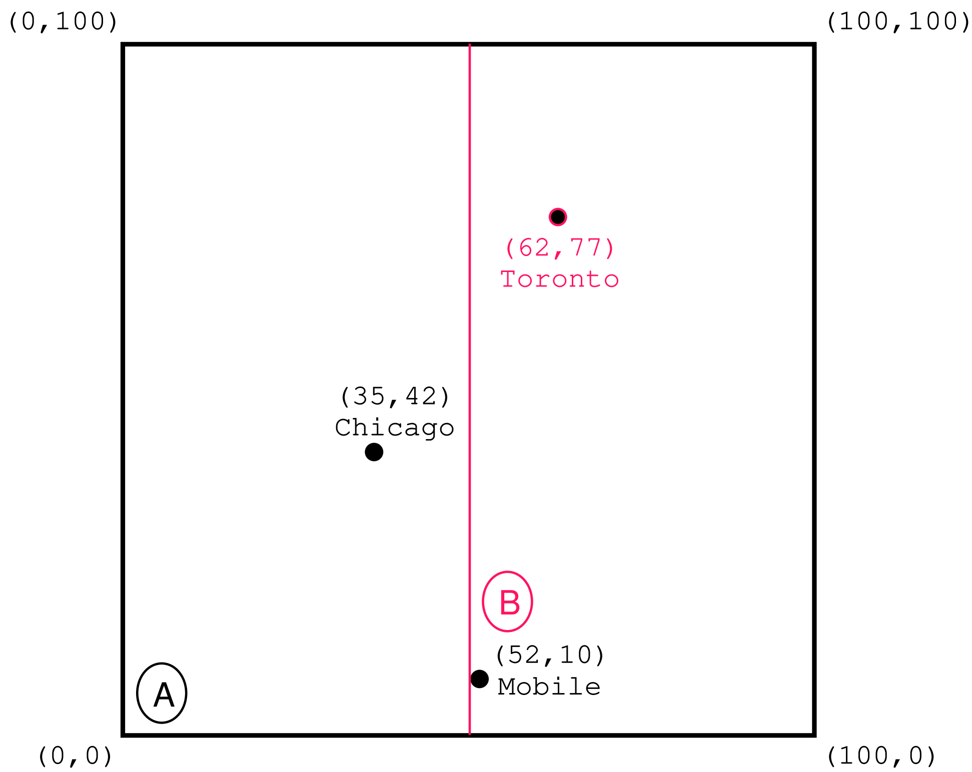
- Uses regular decomposition
- Like grid file, guarantees access to any record with two disk operations
- Differentiated from grid file by absence of linear scales which enable decomposition of space to be arbitrary
- Grid partition results in doubling the size of grid directory
- Ex: bucket size = 2



1. Initially, `Chicago` and `Mobile` in bucket `A`

EXCELL (Tamminen)

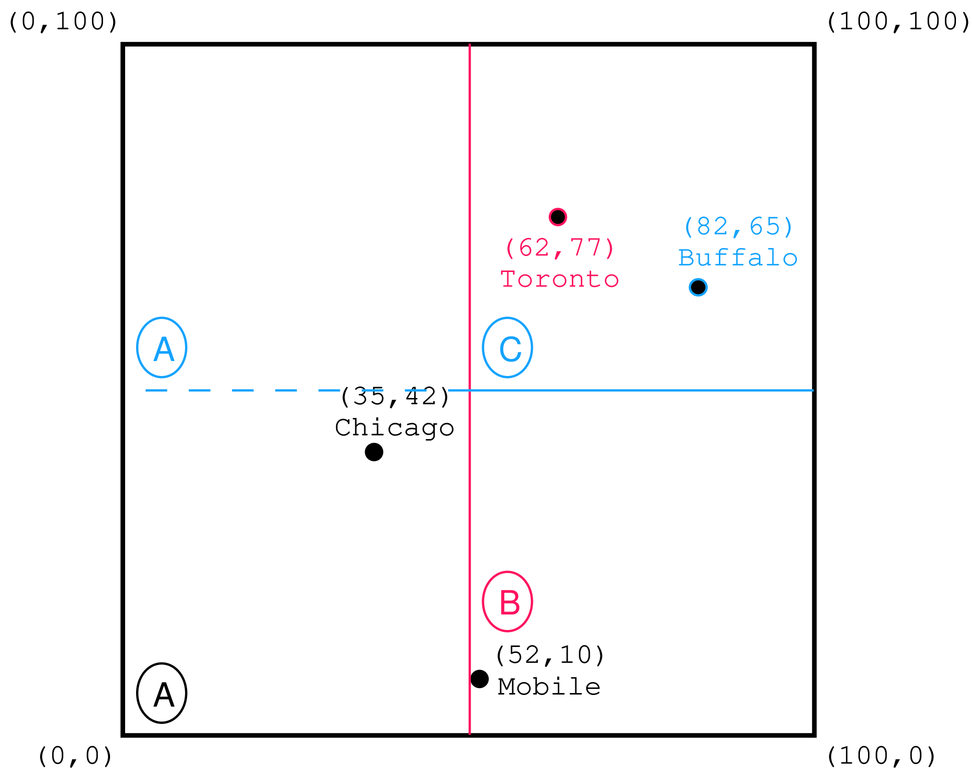
- Uses regular decomposition
- Like grid file, guarantees access to any record with two disk operations
- Differentiated from grid file by absence of linear scales which enable decomposition of space to be arbitrary
- Grid partition results in doubling the size of grid directory
- Ex: bucket size = 2



1. Initially, `Chicago` and `Mobile` in bucket A
2. Insert `Toronto` causing a grid partition yielding bucket B

EXCELL (Tamminen)

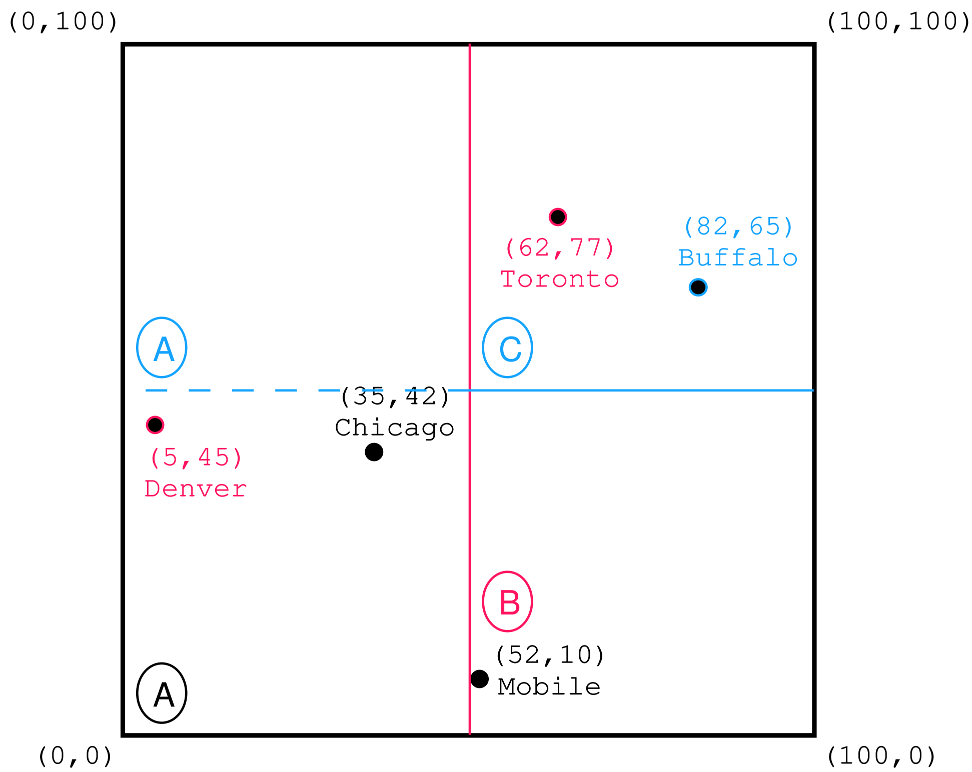
- Uses regular decomposition
- Like grid file, guarantees access to any record with two disk operations
- Differentiated from grid file by absence of linear scales which enable decomposition of space to be arbitrary
- Grid partition results in doubling the size of grid directory
- Ex: bucket size = 2



1. Initially, `Chicago` and `Mobile` in bucket A
2. Insert `Toronto` causing a grid partition yielding bucket B
3. Insert `Buffalo` causing a grid partition yielding bucket C

EXCELL (Tamminen)

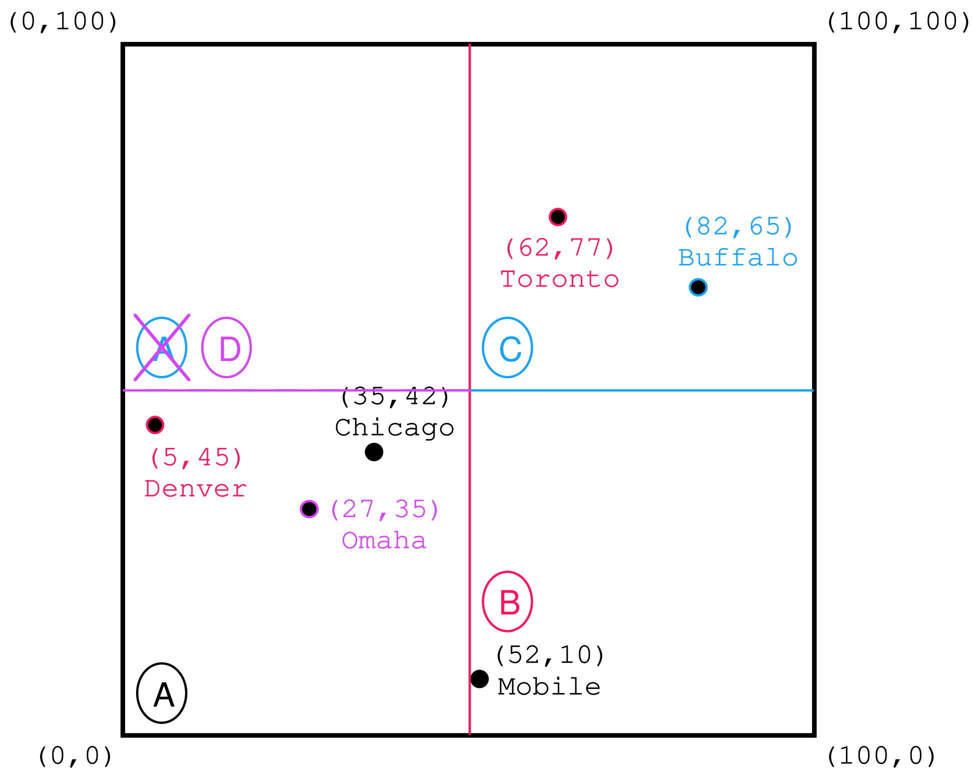
- Uses regular decomposition
- Like grid file, guarantees access to any record with two disk operations
- Differentiated from grid file by absence of linear scales which enable decomposition of space to be arbitrary
- Grid partition results in doubling the size of grid directory
- Ex: bucket size = 2



1. Initially, `Chicago` and `Mobile` in bucket A
2. Insert `Toronto` causing a grid partition yielding bucket B
3. Insert `Buffalo` causing a grid partition yielding bucket C
4. Insert `Denver` causing no change

EXCELL (Tamminen)

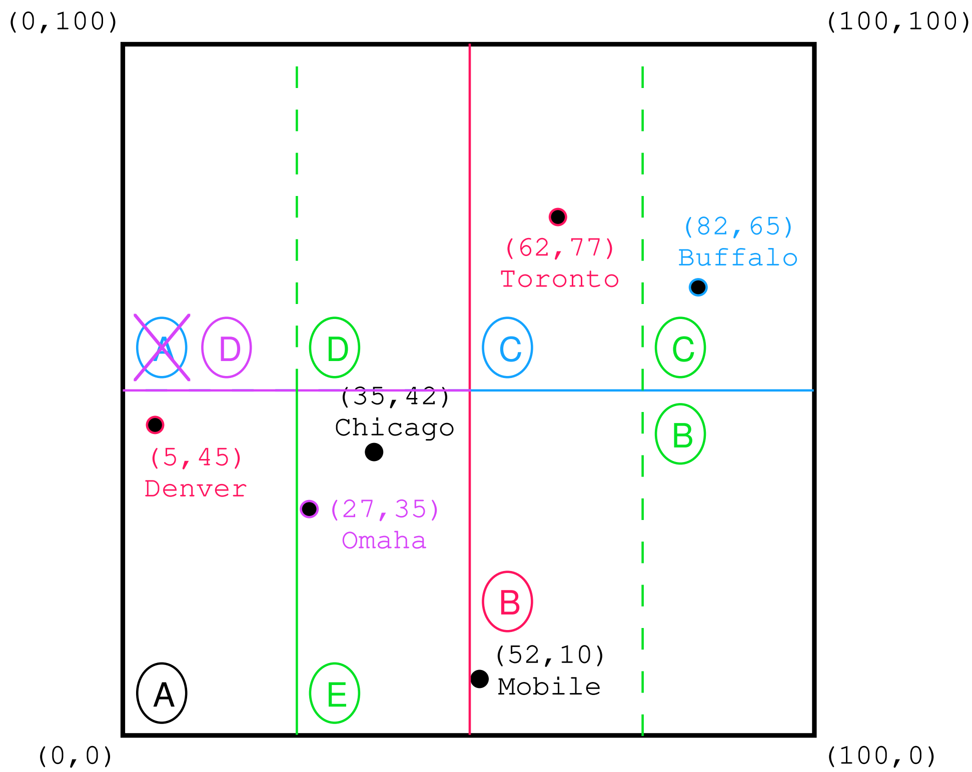
- Uses regular decomposition
- Like grid file, guarantees access to any record with two disk operations
- Differentiated from grid file by absence of linear scales which enable decomposition of space to be arbitrary
- Grid partition results in doubling the size of grid directory
- Ex: bucket size = 2



1. Initially, `Chicago` and `Mobile` in bucket A
2. Insert `Toronto` causing a grid partition yielding bucket B
3. Insert `Buffalo` causing a grid partition yielding bucket C
4. Insert `Denver` causing no change
5. Insert `Omaha`; bucket A overflows; split A yielding bucket D

EXCELL (Tamminen)

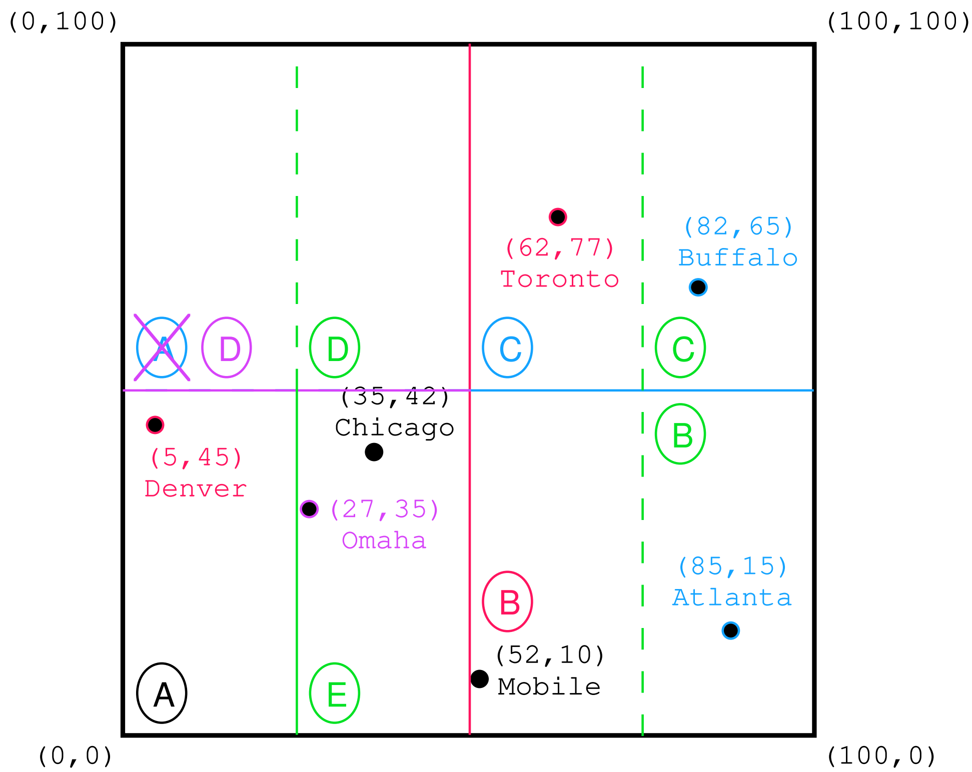
- Uses regular decomposition
- Like grid file, guarantees access to any record with two disk operations
- Differentiated from grid file by absence of linear scales which enable decomposition of space to be arbitrary
- Grid partition results in doubling the size of grid directory
- Ex: bucket size = 2



1. Initially, Chicago and Mobile in bucket A
2. Insert Toronto causing a grid partition yielding bucket B
3. Insert Buffalo causing a grid partition yielding bucket C
4. Insert Denver causing no change
5. Insert Omaha; bucket A overflows; split A yielding bucket D
6. Bucket A is still too full, so perform a grid partition

EXCELL (Tamminen)

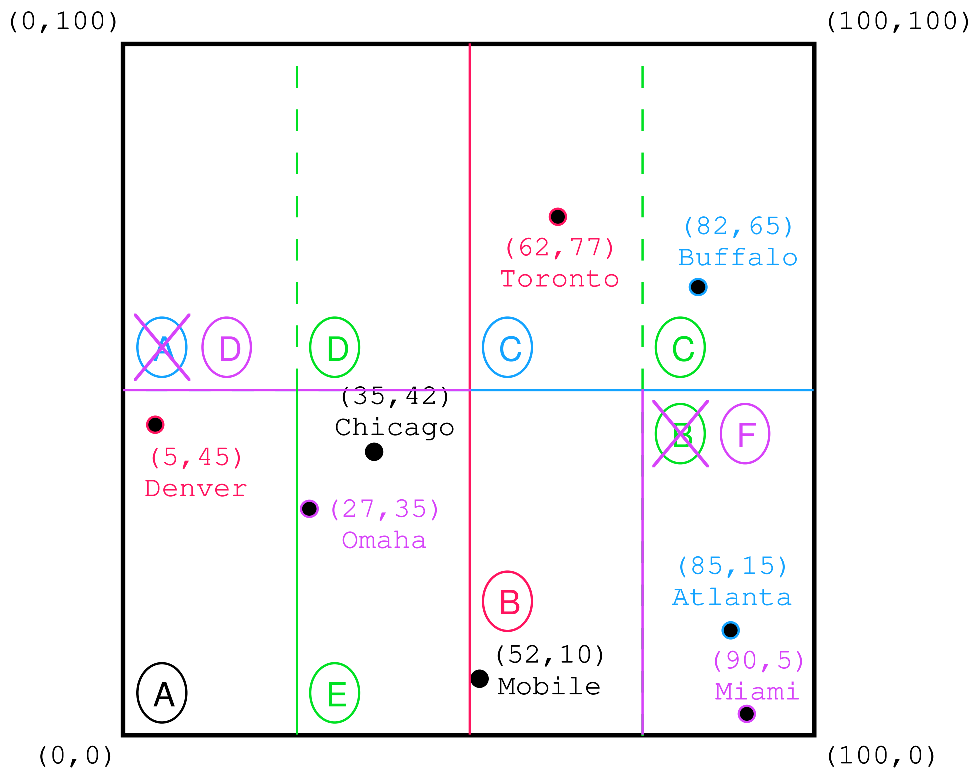
- Uses regular decomposition
- Like grid file, guarantees access to any record with two disk operations
- Differentiated from grid file by absence of linear scales which enable decomposition of space to be arbitrary
- Grid partition results in doubling the size of grid directory
- Ex: bucket size = 2



1. Initially, `Chicago` and `Mobile` in bucket A
2. Insert `Toronto` causing a grid partition yielding bucket B
3. Insert `Buffalo` causing a grid partition yielding bucket C
4. Insert `Denver` causing no change
5. Insert `Omaha`; bucket A overflows; split A yielding bucket D
6. Bucket A is still too full, so perform a grid partition
7. Insert `Atlanta` causing no change

EXCELL (Tamminen)

- Uses regular decomposition
- Like grid file, guarantees access to any record with two disk operations
- Differentiated from grid file by absence of linear scales which enable decomposition of space to be arbitrary
- Grid partition results in doubling the size of grid directory
- Ex: bucket size = 2



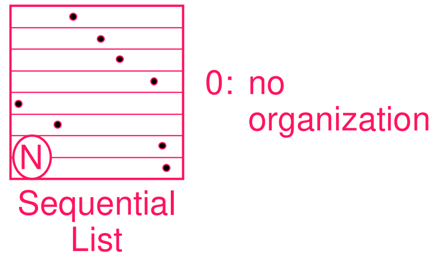
1. Initially, Chicago and Mobile in bucket A
2. Insert Toronto causing a grid partition yielding bucket B
3. Insert Buffalo causing a grid partition yielding bucket C
4. Insert Denver causing no change
5. Insert Omaha; bucket A overflows; split A yielding bucket D
6. Bucket A is still too full, so perform a grid partition
7. Insert Atlanta causing no change
8. Insert Miami causing a bucket partition yielding bucket F

SUMMARY

- Data structures can be grouped:
  - N = No data organization
  - D = Organize data to be stored (e.g., binary search tree)
  - E = Organize the embedding space from which the data is drawn (e.g., digital searching)
  - H = Hybrid (combines at least two of N, D, and E)

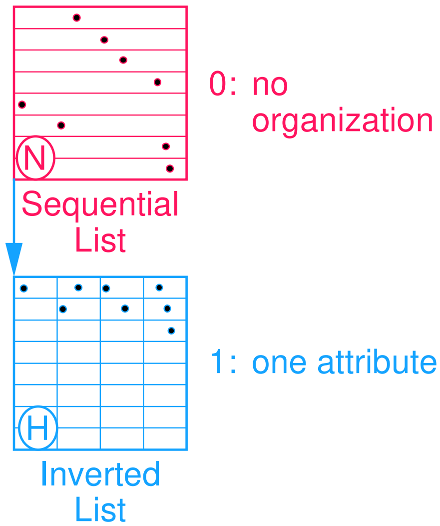
SUMMARY

- Data structures can be grouped:
  N = No data organization
  D = Organize data to be stored (e.g., binary search tree)
  E = Organize the embedding space from which the data is drawn (e.g., digital searching)
  H = Hybrid (combines at least two of N, D, and E)



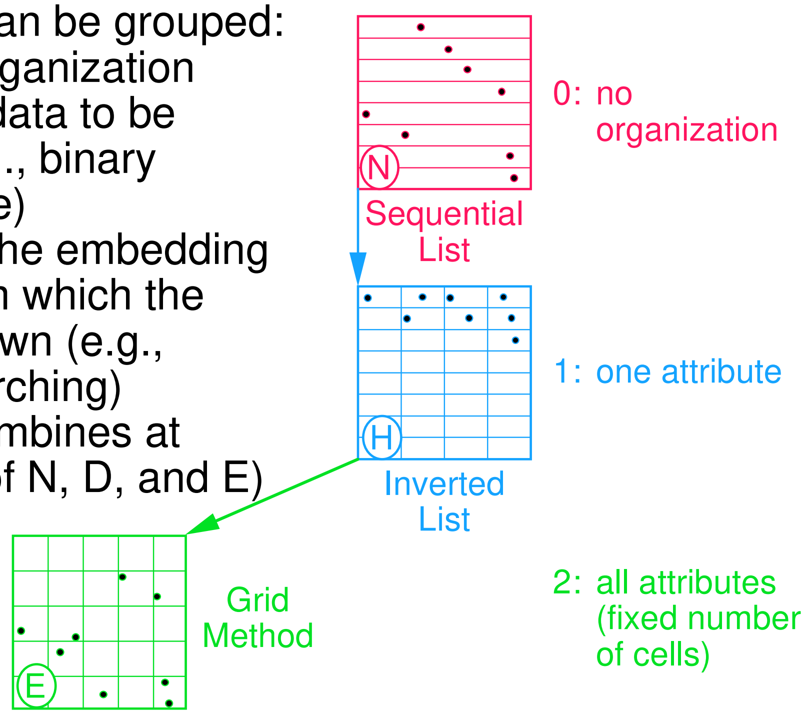
SUMMARY

- Data structures can be grouped:
  - N = No data organization
  - D = Organize data to be stored (e.g., binary search tree)
  - E = Organize the embedding space from which the data is drawn (e.g., digital searching)
  - H = Hybrid (combines at least two of N, D, and E)



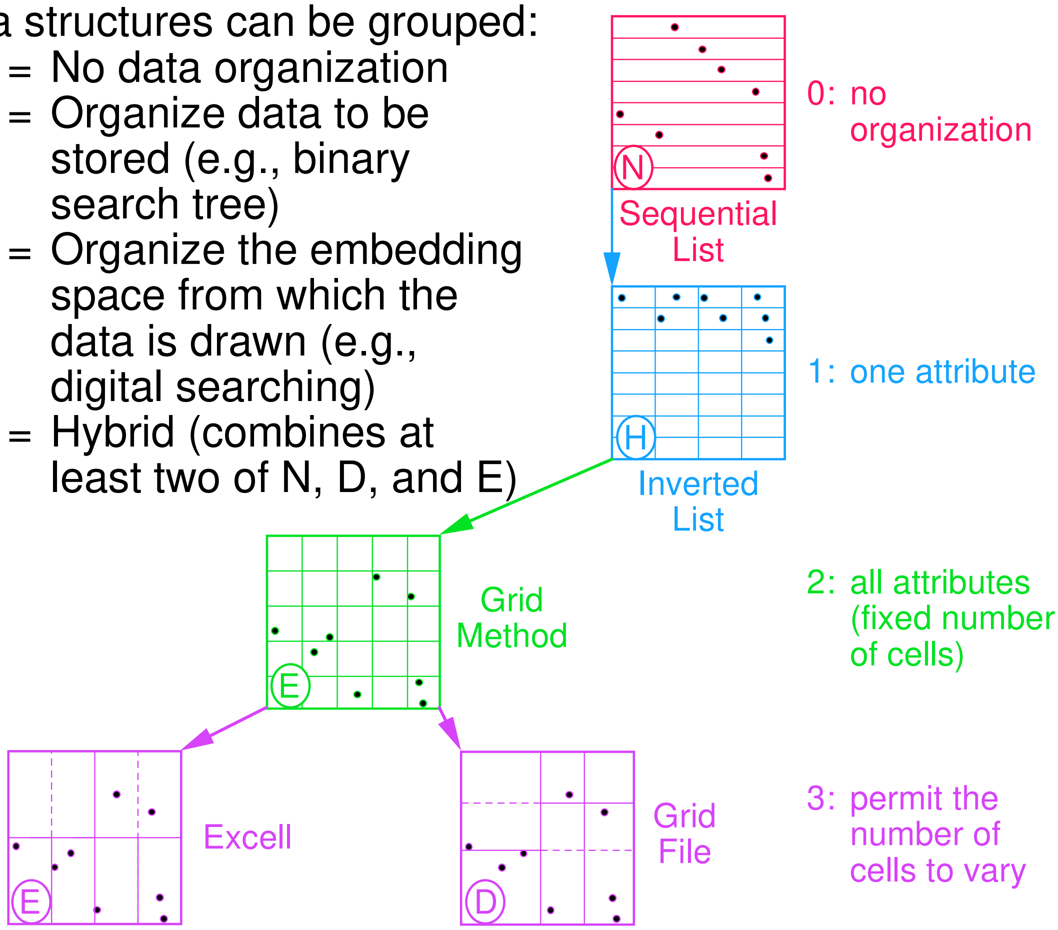
SUMMARY

- Data structures can be grouped:
  - N = No data organization
  - D = Organize data to be stored (e.g., binary search tree)
  - E = Organize the embedding space from which the data is drawn (e.g., digital searching)
  - H = Hybrid (combines at least two of N, D, and E)



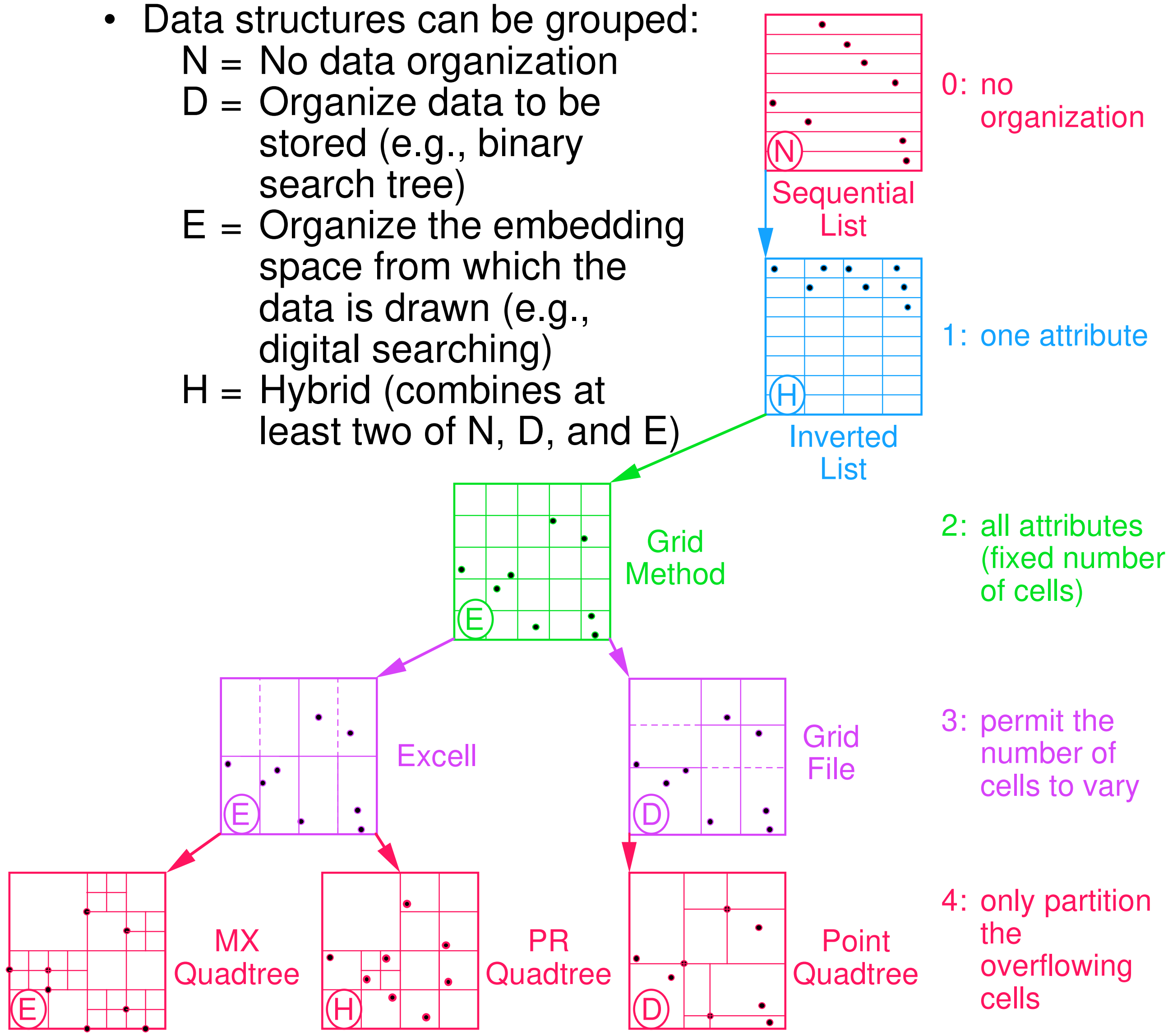
SUMMARY

- Data structures can be grouped:
  N = No data organization
  D = Organize data to be stored (e.g., binary search tree)
  E = Organize the embedding space from which the data is drawn (e.g., digital searching)
  H = Hybrid (combines at least two of N, D, and E)



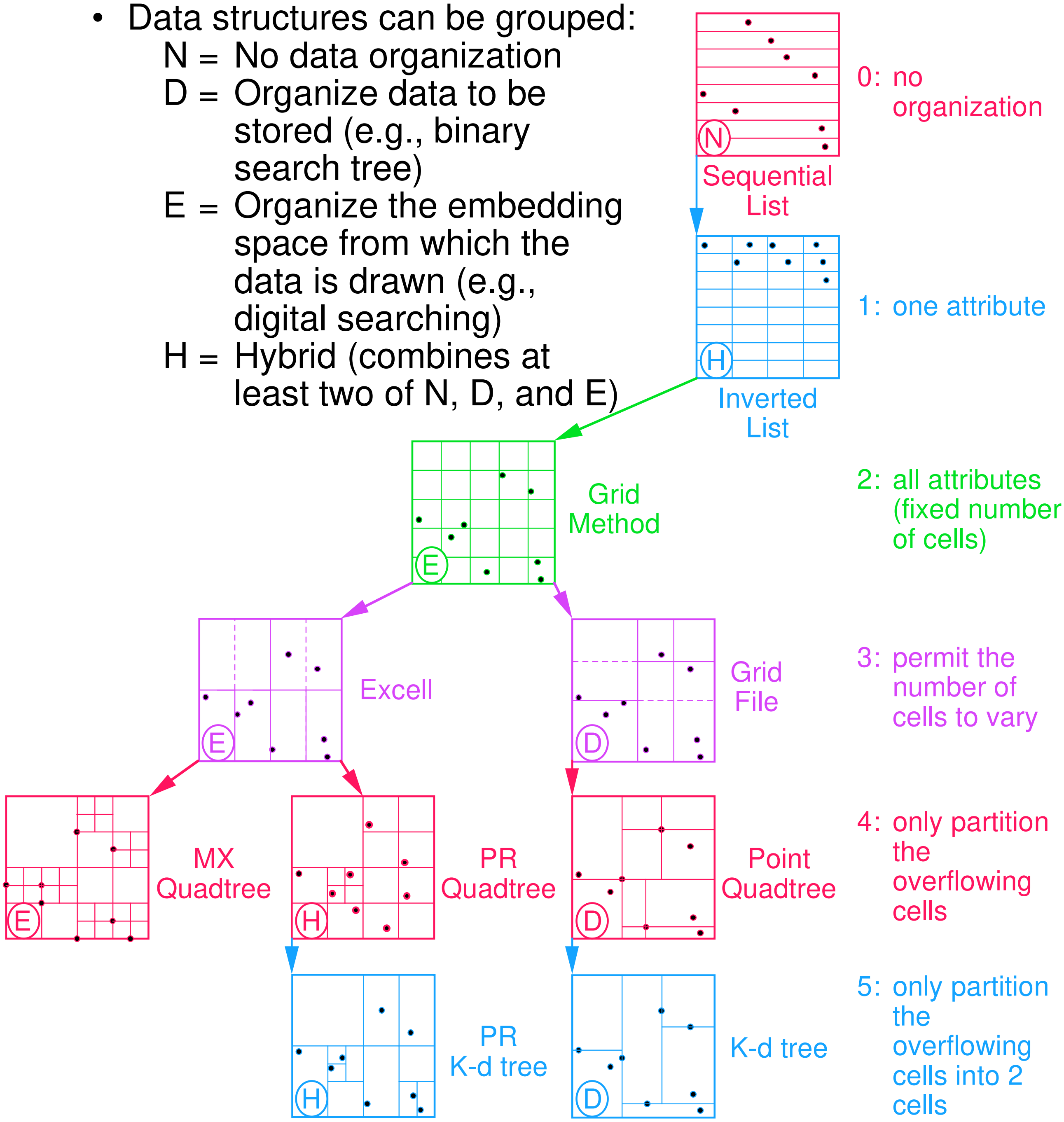
SUMMARY

- Data structures can be grouped:
  - N = No data organization
  - D = Organize data to be stored (e.g., binary search tree)
  - E = Organize the embedding space from which the data is drawn (e.g., digital searching)
  - H = Hybrid (combines at least two of N, D, and E)



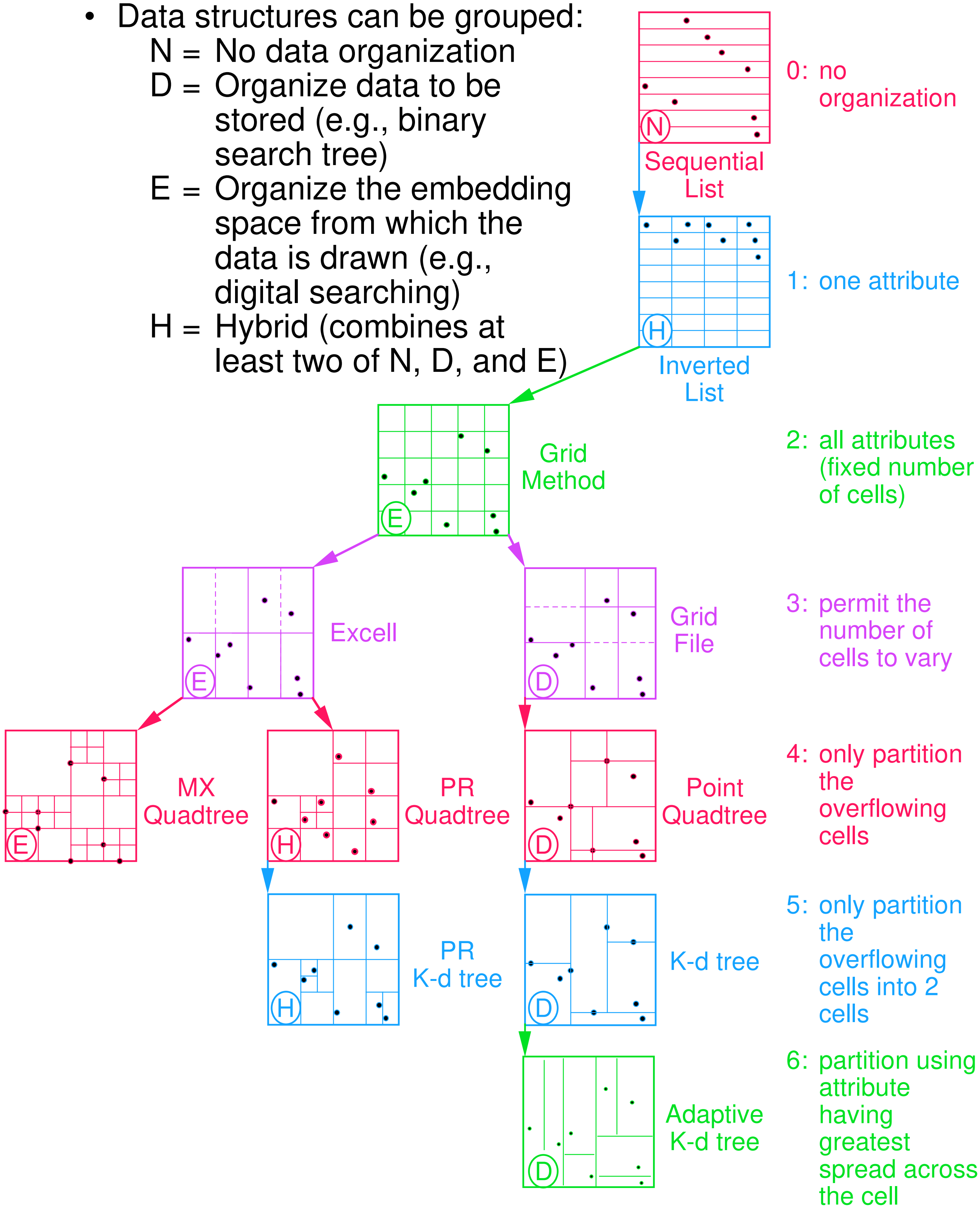
SUMMARY

- Data structures can be grouped:
  - N = No data organization
  - D = Organize data to be stored (e.g., binary search tree)
  - E = Organize the embedding space from which the data is drawn (e.g., digital searching)
  - H = Hybrid (combines at least two of N, D, and E)



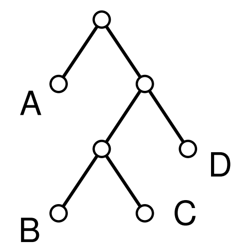
SUMMARY

- Data structures can be grouped:
  - N = No data organization
  - D = Organize data to be stored (e.g., binary search tree)
  - E = Organize the embedding space from which the data is drawn (e.g., digital searching)
  - H = Hybrid (combines at least two of N, D, and E)



DRAWBACKS OF MOST HASHING METHODS

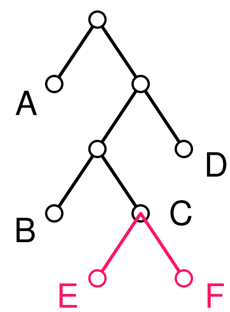
- Require rehashing all the data when the hash table becomes too full
- Goal: only move a few records
- Solutions:
  1. Knott
     - use a trie in the form of a binary tree



  2. extendible hashing
     (Fagin, Nievergelt, Pippenger, Strong)
     - like a trie except that all buckets are at same level
     - buckets are accessed by use of a directory
     - directory elements are NOT the same as buckets
  3. Linear hashing (Litwin)
     - provides for linear growth in the number of buckets (i.e., the hash table grows at a rate of one bucket at a time)
     - does not make use of a directory

DRAWBACKS OF MOST HASHING METHODS

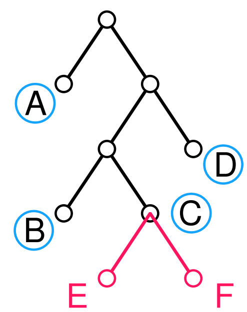
- Require rehashing all the data when the hash table becomes too full
- Goal: only move a few records
- Solutions:



1. Knott
   - use a trie in the form of a binary tree
   - bucket overflow is solved by splitting
   - drawback: accessing a bucket at level *m* requires *m* operations
2. extendible hashing
   (Fagin, Nievergelt, Pippenger, Strong)
   - like a trie except that all buckets are at same level
   - buckets are accessed by use of a directory
   - directory elements are NOT the same as buckets
3. Linear hashing (Litwin)
   - provides for linear growth in the number of buckets (i.e., the hash table grows at a rate of one bucket at a time)
   - does not make use of a directory

DRAWBACKS OF MOST HASHING METHODS

- Require rehashing all the data when the hash table becomes too full
- Goal: only move a few records
- Solutions:
  1. Knott
     - use a trie in the form of a binary tree
     - bucket overflow is solved by splitting
     - drawback: accessing a bucket at level *m* requires *m* operations

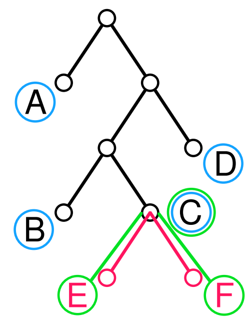


  2. extendible hashing (Fagin, Nievergelt, Pippenger, Strong)
     - like a trie except that all buckets are at same level
     - buckets are accessed by use of a directory
     - directory elements are NOT the same as buckets
     - implemented as a directory of pointers to buckets
     - accessing a bucket requires 1 operation

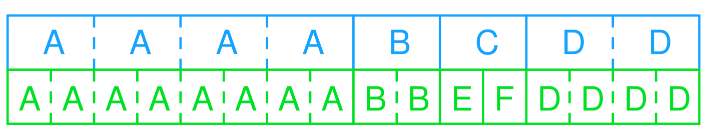
  3. Linear hashing (Litwin)
     - provides for linear growth in the number of buckets (i.e., the hash table grows at a rate of one bucket at a time)
     - does not make use of a directory

DRAWBACKS OF MOST HASHING METHODS

- Require rehashing all the data when the hash table becomes too full
- Goal: only move a few records
- Solutions:



1. Knott
   - use a trie in the form of a binary tree
   - bucket overflow is solved by splitting
   - drawback: accessing a bucket at level *m* requires *m* operations
2. extendible hashing (Fagin, Nievergelt, Pippenger, Strong)
   - like a trie except that all buckets are at same level
   - buckets are accessed by use of a directory
   - directory elements are NOT the same as buckets
   - implemented as a directory of pointers to buckets
   - accessing a bucket requires 1 operation
   - bucket overflow may cause doubling the directory
   - e.g., EXCELL



3. Linear hashing (Litwin)
   - provides for linear growth in the number of buckets (i.e., the hash table grows at a rate of one bucket at a time)
   - does not make use of a directory

MECHANICS OF LINEAR HASHING

- Assume a file with m buckets
- Use two hashing functions $h_i(k) = f(k) \bmod 2^{i+1}$ for $i = n$ and $i = n+1$
  1. compute $h_{n+1}(k) = x$ and use the result if $x < m$
  2. otherwise use $h_n(k)$
- Such a file is said to be at level $n,n+1$
- There exist primary and overflow buckets
- When a record hashes to a full primary bucket, then it is inserted into an overflow bucket corresponding to the primary bucket
- τ : storage utilization factor

  τ = number of records in file divided by the total of available slots in primary and overflow buckets
- When $\tau >$ a given load α, then one of the buckets is split
- When bucket i is split, it and its overflow bucket's records are rehashed using h_{n+1} and distributed into buckets i and $i + 2^n$ as is appropriate

LINEAR HASHING INSERTION ALGORITHM

- Let *s* denote the identity of the next bucket to be split and cycles from 0 to 2^n-1
- Insertion algorithm
  1. compute bucket address *i* for record *r*
  2. insert *r* in bucket *i*
  3. if $\tau > \alpha$, then split bucket *i* creating bucket $i + 2^n$ and reinsert in buckets *i* and $i + 2^n$
  4. if $s = 2^n$, then all buckets have been split
     - increment *n*
     - reset *s* to 0
  5. if buckets *i* or *i+1* overflow, then allocate an overflow bucket
  6. if rehashing causes some overflow buckets to be reclaimed, repeat steps 3-5
- Notes
  1. a bucket split need not necessarily occur when a record hashes to a full bucket, nor does the bucket being split need to be full
  2. key principle is that eventually every bucket will be split and ideally all overflow buckets will be emptied and reclaimed
  3. if the storage utilization gets too low, then buckets should be reclaimed

BIT INTERLEAVING HASHING FUNCTIONS

- Bit interleaving takes one bit from the binary representation of the *x* coordinate value and one bit from the binary representation of the *y* coordinate value and alternates them
- Use city coordinate values and divide by 12.5 so that each coordinate value requires just three binary digits
- Example with *y* being more significant than *x*

City	x	y	f(z)=z div 12.5 x	f(z)=z div 12.5 y	Bit Interleaved Value
Chicago	35	42	2	3	14
Mobile	52	10	4	0	16
Toronto	62	77	4	6	56
Buffalo	82	65	6	5	54
Denver	5	45	0	3	10
Omaha	27	35	2	2	12
Atlanta	85	15	6	1	22
Miami	90	5	7	0	21

BIT INTERLEAVING HASHING FUNCTIONS

- Bit interleaving takes one bit from the binary representation of the *x* coordinate value and one bit from the binary representation of the *y* coordinate value and alternates them
- Use city coordinate values and divide by 12.5 so that each coordinate value requires just three binary digits
- Example with *y* being more significant than *x*

City	x	y	f(z)=z div 12.5 x	f(z)=z div 12.5 y	Bit Interleaved Value
Chicago	35	42	2	3	14
Mobile	52	10	4	0	16
Toronto	62	77	4	6	56
Buffalo	82	65	6	5	54
Denver	5	45	0	3	10
Omaha	27	35	2	2	12
Atlanta	85	15	6	1	22
Miami	90	5	7	0	21

| 1 | 1 | 0 | x |

- Ex: `Atlanta` (6,1)

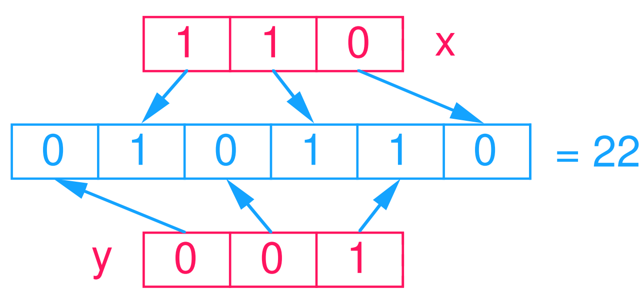
| y | 0 | 0 | 1 |

BIT INTERLEAVING HASHING FUNCTIONS

- Bit interleaving takes one bit from the binary representation of the *x* coordinate value and one bit from the binary representation of the *y* coordinate value and alternates them
- Use city coordinate values and divide by 12.5 so that each coordinate value requires just three binary digits
- Example with *y* being more significant than *x*

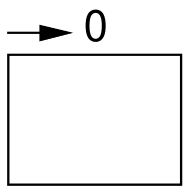
City	x	y	f(z)=z div 12.5 x	f(z)=z div 12.5 y	Bit Interleaved Value
Chicago	35	42	2	3	14
Mobile	52	10	4	0	16
Toronto	62	77	4	6	56
Buffalo	82	65	6	5	54
Denver	5	45	0	3	10
Omaha	27	35	2	2	12
Atlanta	85	15	6	1	22
Miami	90	5	7	0	21

- Ex: `Atlanta` (6,1)



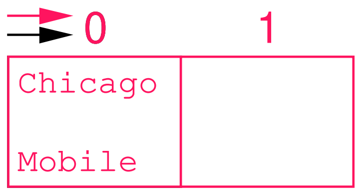
EXAMPLE OF LINEAR HASHING

- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



EXAMPLE OF LINEAR HASHING

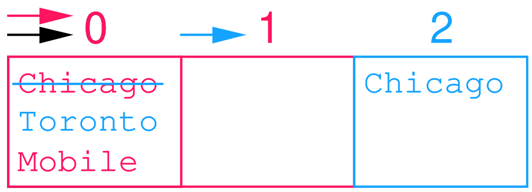
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (14) and `Mobile` (16):
  τ=1 and split bucket 0 creating bucket 1

EXAMPLE OF LINEAR HASHING

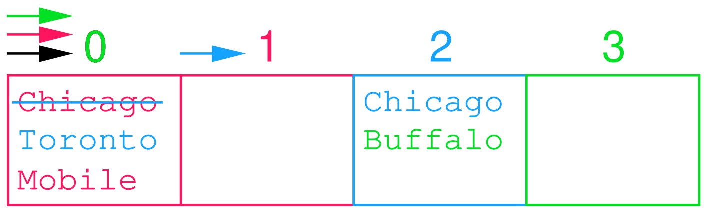
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (14) and `Mobile` (16):
  τ=1 and split bucket 0 creating bucket 1
- Insert `Toronto` (56) into bucket 0:
  τ=0.75 and split bucket 0 creating bucket 2; move `Chicago` to bucket 2

EXAMPLE OF LINEAR HASHING

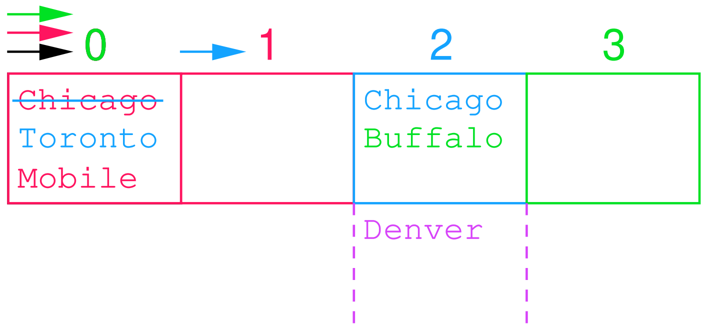
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (14) and `Mobile` (16):
  τ=1 and split bucket 0 creating bucket 1
- Insert `Toronto` (56) into bucket 0:
  τ=0.75 and split bucket 0 creating bucket 2; move `Chicago` to bucket 2
- Insert `Buffalo` (54) into bucket 2:
  τ=0.67 and split bucket 1 creating bucket 3

EXAMPLE OF LINEAR HASHING

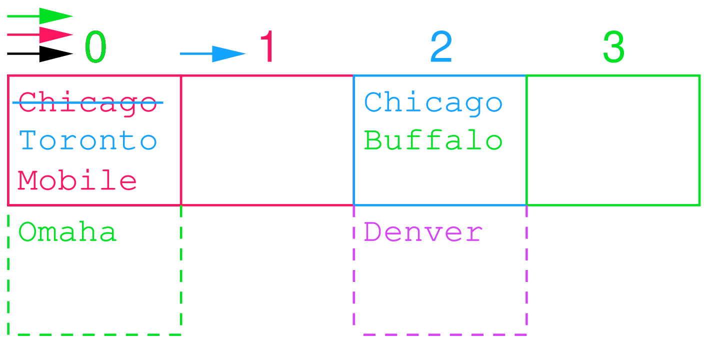
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (14) and `Mobile` (16):
  τ=1 and split bucket 0 creating bucket 1
- Insert `Toronto` (56) into bucket 0:
  τ=0.75 and split bucket 0 creating bucket 2; move `Chicago` to bucket 2
- Insert `Buffalo` (54) into bucket 2:
  τ=0.67 and split bucket 1 creating bucket 3
- Insert `Denver` (10) into bucket 2 which causes it to overflow

EXAMPLE OF LINEAR HASHING

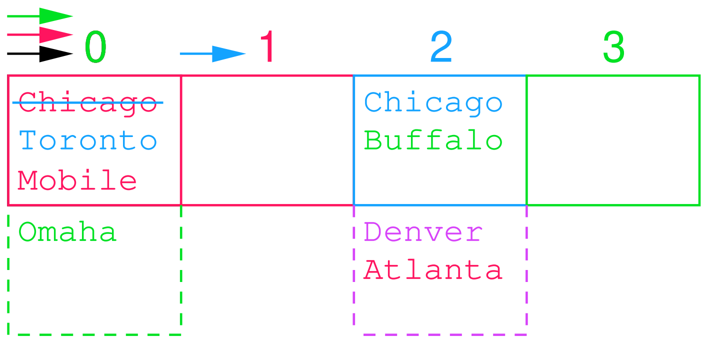
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (14) and `Mobile` (16):
  τ=1 and split bucket 0 creating bucket 1
- Insert `Toronto` (56) into bucket 0:
  τ=0.75 and split bucket 0 creating bucket 2; move `Chicago` to bucket 2
- Insert `Buffalo` (54) into bucket 2:
  τ=0.67 and split bucket 1 creating bucket 3
- Insert `Denver` (10) into bucket 2 which causes it to overflow
- Insert `Omaha` (12) into bucket 0 which causes it to overflow

EXAMPLE OF LINEAR HASHING

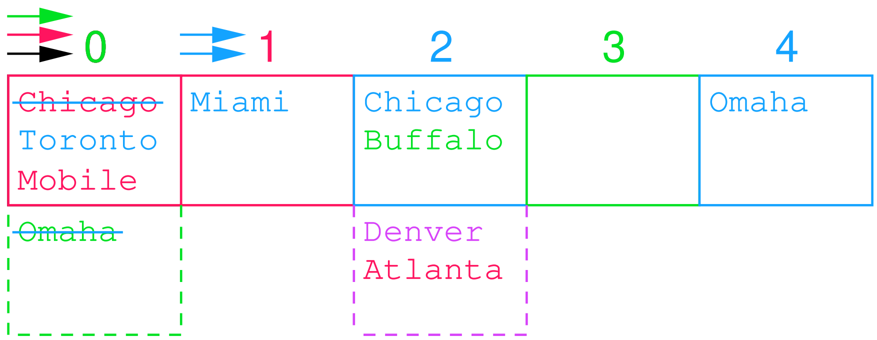
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (14) and `Mobile` (16):
  τ=1 and split bucket 0 creating bucket 1
- Insert `Toronto` (56) into bucket 0:
  τ=0.75 and split bucket 0 creating bucket 2; move `Chicago` to bucket 2
- Insert `Buffalo` (54) into bucket 2:
  τ=0.67 and split bucket 1 creating bucket 3
- Insert `Denver` (10) into bucket 2 which causes it to overflow
- Insert `Omaha` (12) into bucket 0 which causes it to overflow
- Insert `Atlanta` (22) into bucket 2's overflow area

EXAMPLE OF LINEAR HASHING

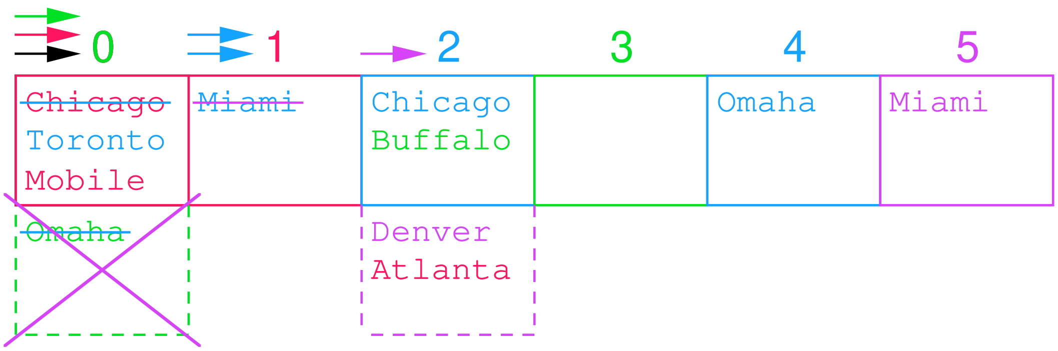
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (14) and `Mobile` (16):
  τ=1 and split bucket 0 creating bucket 1
- Insert `Toronto` (56) into bucket 0:
  τ=0.75 and split bucket 0 creating bucket 2; move `Chicago` to bucket 2
- Insert `Buffalo` (54) into bucket 2:
  τ=0.67 and split bucket 1 creating bucket 3
- Insert `Denver` (10) into bucket 2 which causes it to overflow
- Insert `Omaha` (12) into bucket 0 which causes it to overflow
- Insert `Atlanta` (22) into bucket 2's overflow area
- Insert `Miami` (21) into bucket 1:
  τ=0.67 and split bucket 0 creating bucket 4; move `Omaha` to bucket 4

EXAMPLE OF LINEAR HASHING

- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (14) and `Mobile` (16):
  τ=1 and split bucket 0 creating bucket 1
- Insert `Toronto` (56) into bucket 0:
  τ=0.75 and split bucket 0 creating bucket 2; move `Chicago` to bucket 2
- Insert `Buffalo` (54) into bucket 2:
  τ=0.67 and split bucket 1 creating bucket 3
- Insert `Denver` (10) into bucket 2 which causes it to overflow
- Insert `Omaha` (12) into bucket 0 which causes it to overflow
- Insert `Atlanta` (22) into bucket 2's overflow area
- Insert `Miami` (21) into bucket 1:
  τ=0.67 and split bucket 0 creating bucket 4; move `Omaha` to bucket 4
- Reclaim the overflow area of bucket 0:
  τ=0.67 again and split bucket 1 creating bucket 5; move `Miami` to bucket 5

ORDER PRESERVING LINEAR HASHING (OPLH)

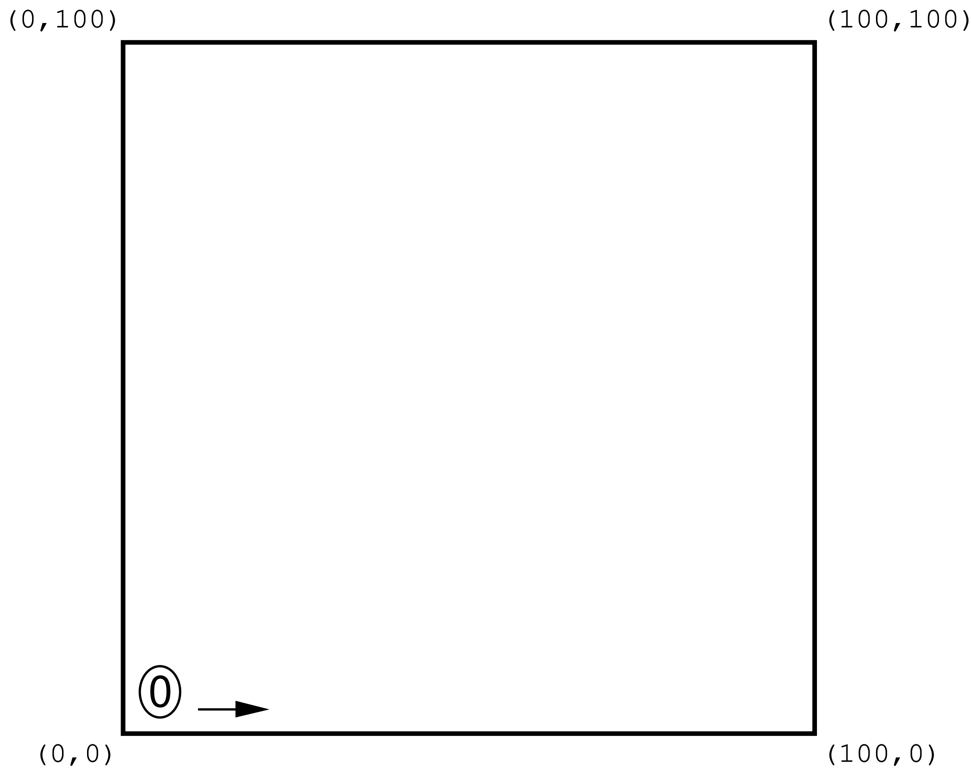
- Problem: the hash function $h_n(k) = k \bmod 2^n$ implies that all records in a given bucket agree in the least n significant bits
  1. OK for random access
  2. unacceptable for sequential access as each record will be in a different bucket
- Solution: use the hash function $h_n(k) = (reverse(k)) \bmod 2^n$
  1. tests the n most significant bits
  2. all records in a bucket are within a given range

City	x	y	f(z)=z mod 12.5 x	f(z)=z mod 12.5 y	Bit Interleaved Value x most sig	Bit Interleaved Value y most sig
Chicago	35	42	2	3	44	28
Mobile	52	10	4	0	1	2
Toronto	62	77	4	6	11	7
Buffalo	82	65	6	5	39	27
Denver	5	45	0	3	40	20
Omaha	27	35	2	2	12	12
Atlanta	85	15	6	1	37	26
Miami	90	5	7	0	21	42

- Shortcomings
  1. records may not be scattered too well
     - overflow is much more common than with traditional hashing methods
     - random access is slower since several overflow buckets may have to be examined
  2. creates a large number of sparsely filled buckets
     - sequential access may be slower as may have to examine many empty buckets

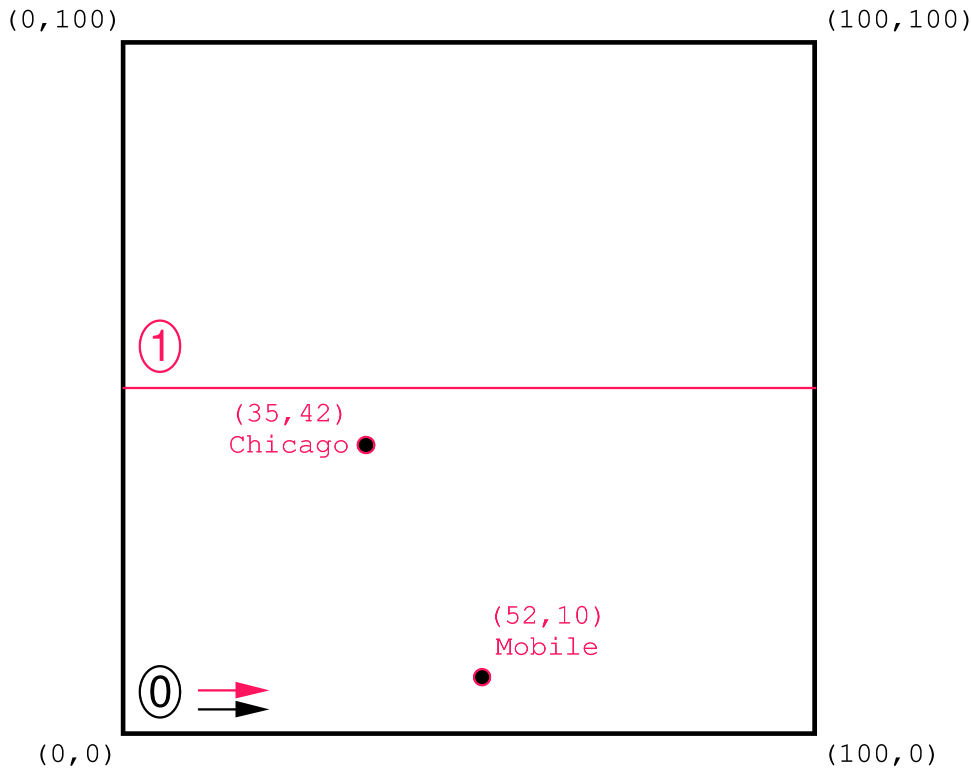
EXAMPLE OF ORDER PRESERVING LINEAR HASHING

- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



EXAMPLE OF ORDER PRESERVING LINEAR HASHING

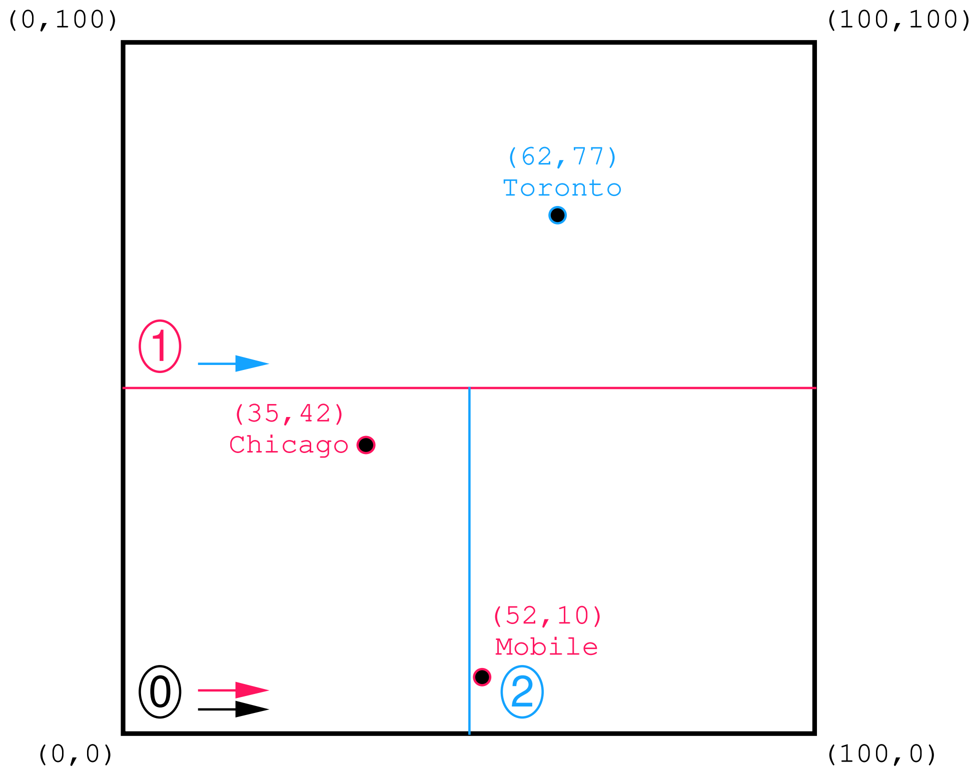
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (28) and `Mobile` (2): $\tau=1$ and split bucket 0 creating bucket 1

EXAMPLE OF ORDER PRESERVING LINEAR HASHING

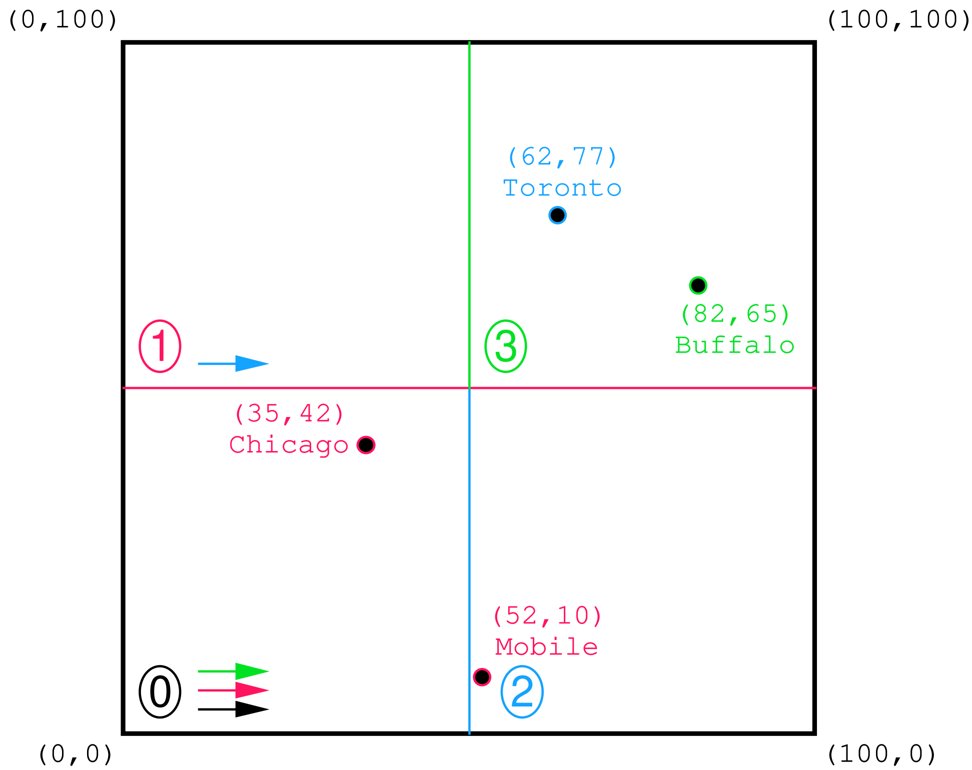
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (28) and `Mobile` (2): $\tau=1$ and split bucket 0 creating bucket 1
- Insert `Toronto` (7) into bucket 1: $\tau=0.75$ and split bucket 0 creating bucket 2; move `Mobile` to bucket 2

EXAMPLE OF ORDER PRESERVING LINEAR HASHING

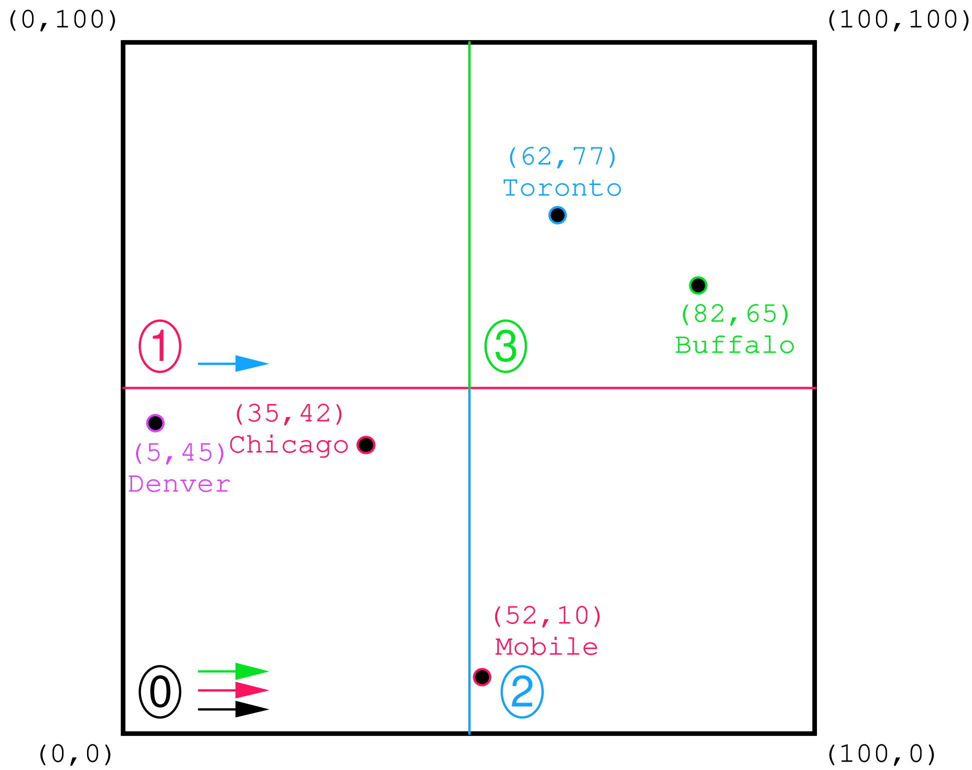
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (28) and `Mobile` (2): $\tau=1$ and split bucket 0 creating bucket 1
- Insert `Toronto` (7) into bucket 1: $\tau=0.75$ and split bucket 0 creating bucket 2; move `Mobile` to bucket 2
- Insert `Buffalo` (27) into 1: $\tau=0.67$, split bucket 1 creating bucket 3; move `Toronto` and `Buffalo` to bucket 3

EXAMPLE OF ORDER PRESERVING LINEAR HASHING

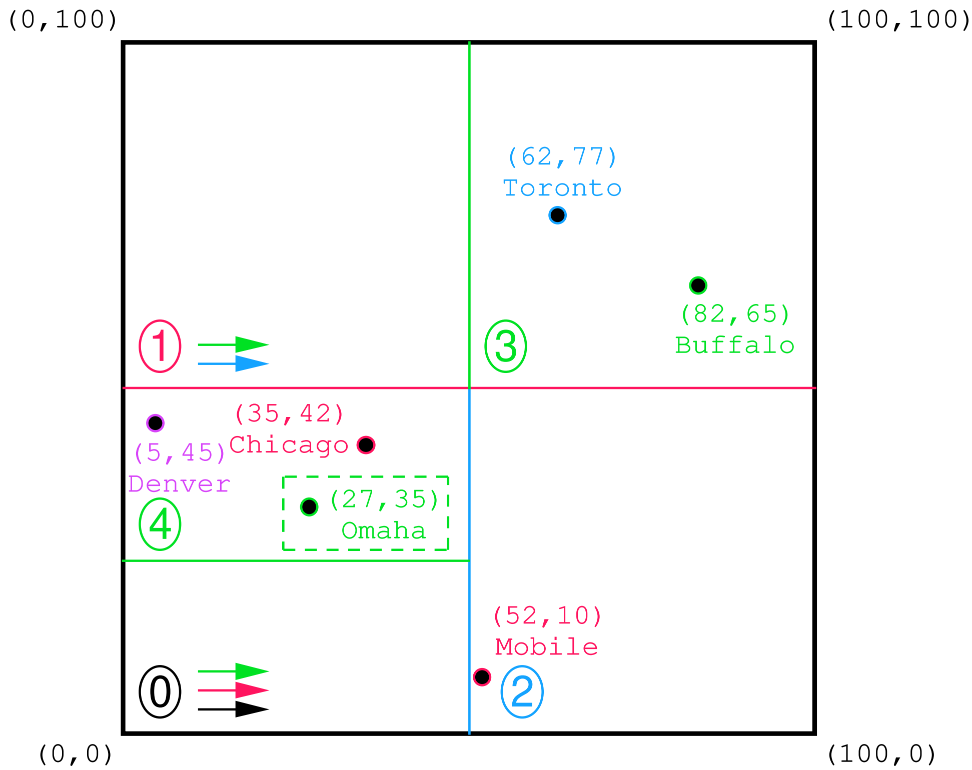
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (28) and `Mobile` (2): $\tau=1$ and split bucket 0 creating bucket 1
- Insert `Toronto` (7) into bucket 1: $\tau=0.75$ and split bucket 0 creating bucket 2; move `Mobile` to bucket 2
- Insert `Buffalo` (27) into 1: $\tau=0.67$, split bucket 1 creating bucket 3; move `Toronto` and `Buffalo` to bucket 3
- Insert `Denver` (20) into bucket 0

EXAMPLE OF ORDER PRESERVING LINEAR HASHING

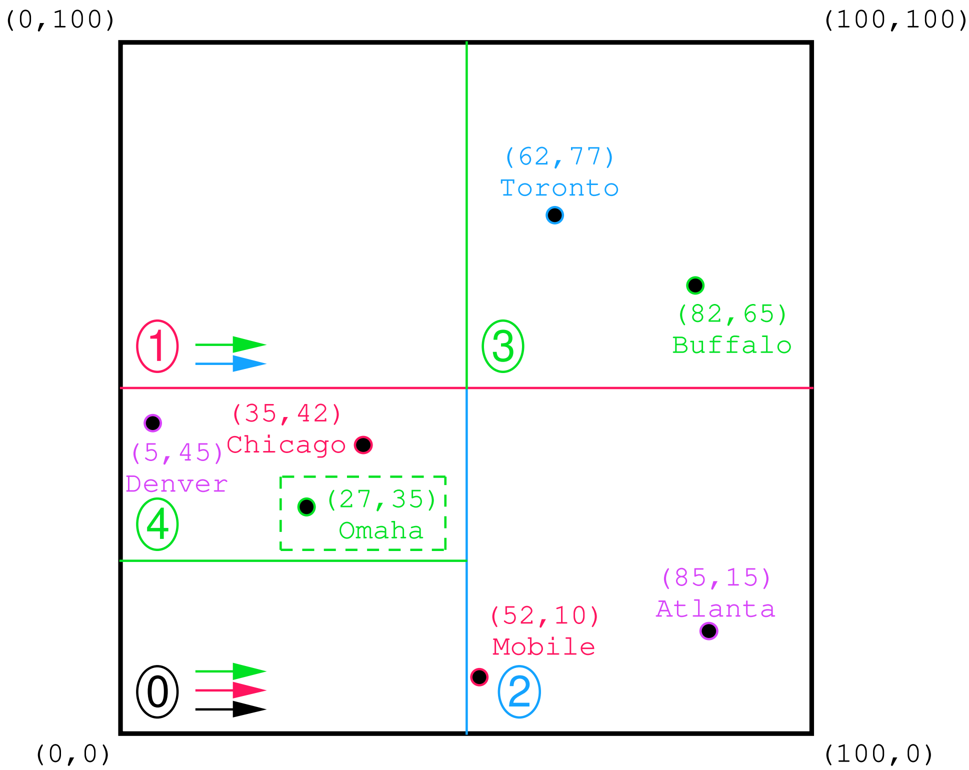
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert Chicago (28) and Mobile (2): $\tau=1$ and split bucket 0 creating bucket 1
- Insert Toronto (7) into bucket 1: $\tau=0.75$ and split bucket 0 creating bucket 2; move Mobile to bucket 2
- Insert Buffalo (27) into 1: $\tau=0.67$, split bucket 1 creating bucket 3; move Toronto and Buffalo to bucket 3
- Insert Denver (20) into bucket 0
- Insert Omaha (12) into 0: $\tau=0.75$ and split bucket 0 creating bucket 4; move Denver and Chicago to bucket 4

EXAMPLE OF ORDER PRESERVING LINEAR HASHING

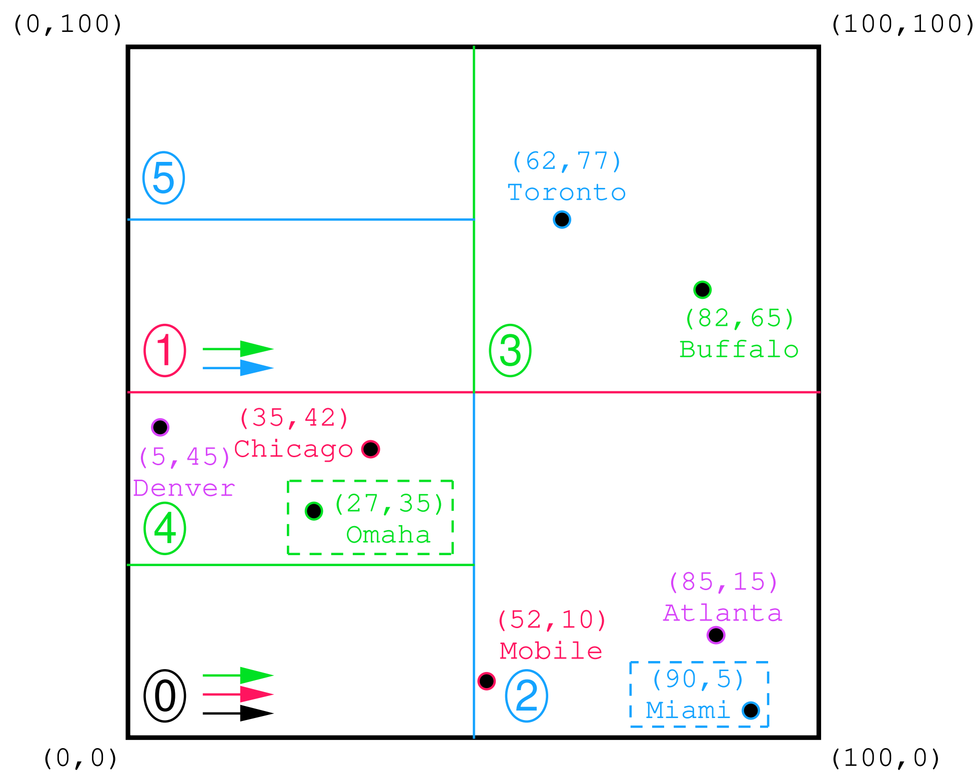
- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (28) and `Mobile` (2): $\tau=1$ and split bucket 0 creating bucket 1
- Insert `Toronto` (7) into bucket 1: $\tau=0.75$ and split bucket 0 creating bucket 2; move `Mobile` to bucket 2
- Insert `Buffalo` (27) into 1: $\tau=0.67$, split bucket 1 creating bucket 3; move `Toronto` and `Buffalo` to bucket 3
- Insert `Denver` (20) into bucket 0
- Insert `Omaha` (12) into 0: $\tau=0.75$ and split bucket 0 creating bucket 4; move `Denver` and `Chicago` to bucket 4
- Insert `Atlanta` (26) into bucket 2

EXAMPLE OF ORDER PRESERVING LINEAR HASHING

- Assume primary and overflow bucket capacity is 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 0 exists



- Insert `Chicago` (28) and `Mobile` (2): $\tau=1$ and split bucket 0 creating bucket 1
- Insert `Toronto` (7) into bucket 1: $\tau=0.75$ and split bucket 0 creating bucket 2; move `Mobile` to bucket 2
- Insert `Buffalo` (27) into 1: $\tau=0.67$, split bucket 1 creating bucket 3; move `Toronto` and `Buffalo` to bucket 3
- Insert `Denver` (20) into bucket 0
- Insert `Omaha` (12) into 0: $\tau=0.75$ and split bucket 0 creating bucket 4; move `Denver` and `Chicago` to bucket 4
- Insert `Atlanta` (26) into bucket 2
- Insert `Miami` (42) into bucket 2: $\tau=0.67$ and split bucket 1 creating bucket 5; move `Miami` to bucket 2's overflow area

COMPARISON OF OPLH WITH EXCELL

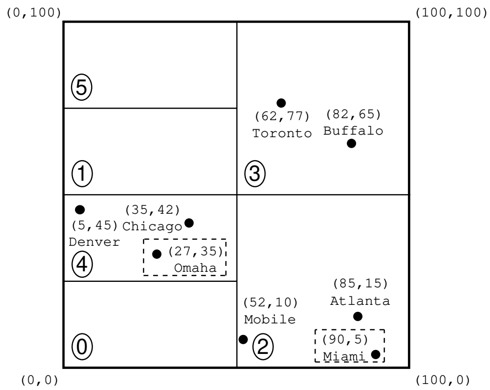
OPLH	EXCELL
1. Implicit directory • one directory element per primary bucket • all buckets stored at one of two levels • overflow buckets	1. Explicit directory • set of primary buckets
2. Reverse bit interleaving	2. Reverse bit interleaving
3. Bucket overflow • allocate at most two additional buckets (one for the bucket that has been split and one for the overflowing bucket)	3. Bucket overflow • triggers bucket split or directory doubling
4. Retrieval of a record requires examining primary and overflow buckets	4. Retrieve any record with two disk accesses

- Summary: order preserving linear hashing (OPLH) yields a more gradual growth in the size of the directory at the expense of the loss of the guarantee of retrieval of any record with just two disk accesses

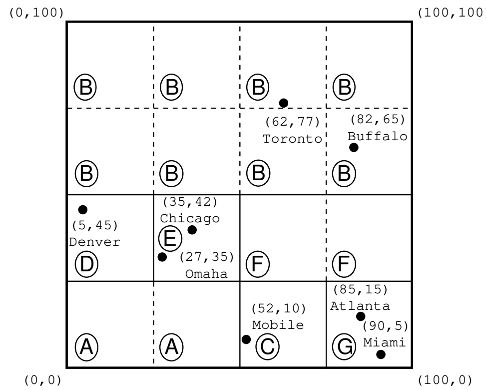
EXAMPLES OF COMPARISON OF OPLH WITH EXCELL

1. Reversed bit interleaving with the *y* coordinate value as the most significant (i.e., *y* is split first)

OPLH:



EXCELL:

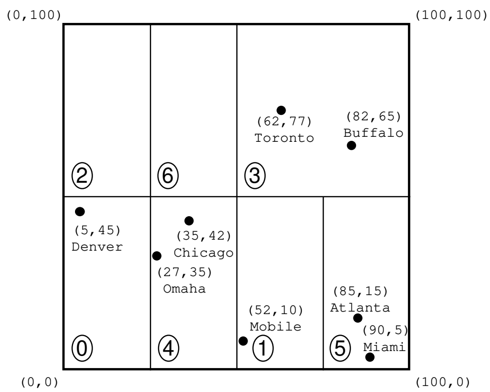


- 6 primary buckets
- 2 overflow buckets

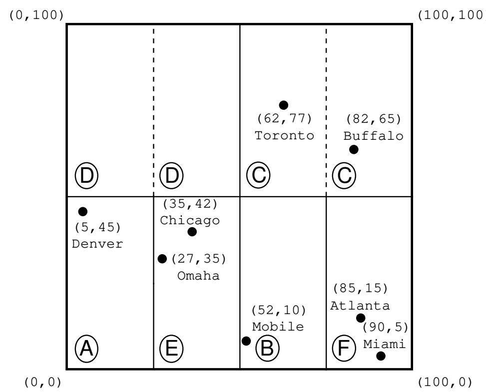
- 7 buckets
- 16 directory elements

2. Reversed bit interleaving with the *x* coordinate value as the most significant (i.e., *x* is split first)

OPLH:



EXCELL:



- 7 primary buckets
- no overflow buckets

- 6 buckets
- 8 directory elements

SPIRAL HASHING (Martin)

- Drawbacks of linear hashing:
  1. order in which buckets are split is unrelated to the probability of the occurence of overflow
  2. all buckets that are candidates for a split have the same probability of overflowing
- Central idea is the existence of an ever-changing (and growing) address space of active bucket addresses
  1. records are distributed in the active buckets in an uneven manner
  2. split the bucket with the highest probability of overflowing
- When a bucket s is split
  1. create d new buckets
  2. rehash the contents of s into the d new buckets
  3. bucket s is no longer used
- If $[s,t]$ are the active buckets, then $[s+1, t+d]$ are the active buckets after the split
- Ex: assume that initially there are $d - 1$ active buckets starting at address 1

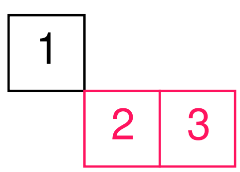
  $d = 2$: $\boxed{1}$

- After s bucket splits, there are $(s+1)\cdot(d+1)$ active buckets starting at address $s+1$ (prove by induction)

SPIRAL HASHING (Martin)

- Drawbacks of linear hashing:
  1. order in which buckets are split is unrelated to the probability of the occurence of overflow
  2. all buckets that are candidates for a split have the same probability of overflowing
- Central idea is the existence of an ever-changing (and growing) address space of active bucket addresses
  1. records are distributed in the active buckets in an uneven manner
  2. split the bucket with the highest probability of overflowing
- When a bucket s is split
  1. create d new buckets
  2. rehash the contents of s into the d new buckets
  3. bucket s is no longer used
- If $[s,t]$ are the active buckets, then $[s+1, t+d]$ are the active buckets after the split
- Ex: assume that initially there are d - 1 active buckets starting at address 1

  $d = 2$:

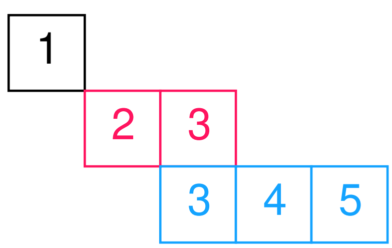


- After s bucket splits, there are $(s+1)\bullet(d+1)$ active buckets starting at address $s+1$ (prove by induction)

SPIRAL HASHING (Martin)

- Drawbacks of linear hashing:
  1. order in which buckets are split is unrelated to the probability of the occurence of overflow
  2. all buckets that are candidates for a split have the same probability of overflowing
- Central idea is the existence of an ever-changing (and growing) address space of active bucket addresses
  1. records are distributed in the active buckets in an uneven manner
  2. split the bucket with the highest probability of overflowing
- When a bucket s is split
  1. create d new buckets
  2. rehash the contents of s into the d new buckets
  3. bucket s is no longer used
- If $[s,t]$ are the active buckets, then $[s+1, t+d]$ are the active buckets after the split
- Ex: assume that initially there are d - 1 active buckets starting at address 1

$d = 2$:

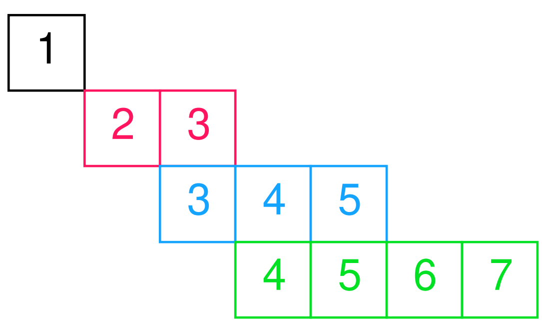


- After s bucket splits, there are $(s+1)\bullet(d+1)$ active buckets starting at address $s + 1$ (prove by induction)

SPIRAL HASHING (Martin)

- Drawbacks of linear hashing:
  1. order in which buckets are split is unrelated to the probability of the occurence of overflow
  2. all buckets that are candidates for a split have the same probability of overflowing
- Central idea is the existence of an ever-changing (and growing) address space of active bucket addresses
  1. records are distributed in the active buckets in an uneven manner
  2. split the bucket with the highest probability of overflowing
- When a bucket s is split
  1. create d new buckets
  2. rehash the contents of s into the d new buckets
  3. bucket s is no longer used
- If $[s,t]$ are the active buckets, then $[s+1, t+d]$ are the active buckets after the split
- Ex: assume that initially there are d - 1 active buckets starting at address 1

$d = 2$:



- After s bucket splits, there are $(s+1)\bullet(d+1)$ active buckets starting at address $s+1$ (prove by induction)

SPIRAL HASHING FUNCTION

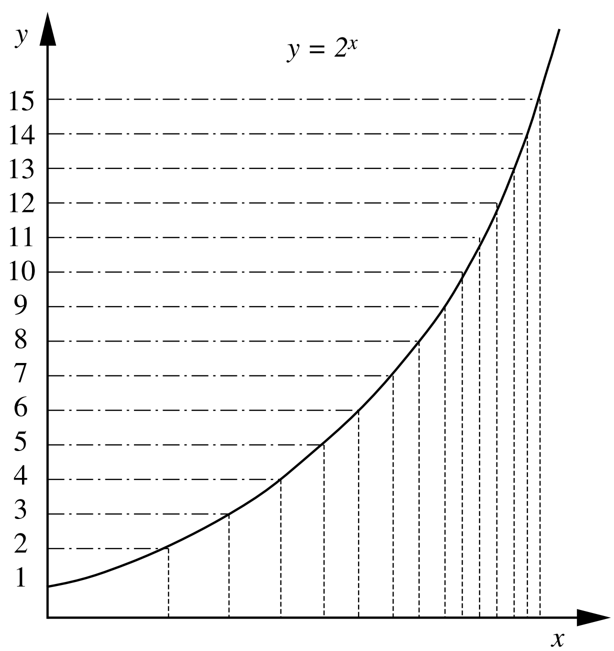
- Assume that initially there are $d-1$ active buckets starting at address 1
- Key idea is the behavior of the function $y = d^x$
  1. $d^{x+1} - d^x = d^x \bullet (d-1)$
  2. bucket s has just been split
  3. let $d^x = s+1$ = address of first active bucket
  4. implies $d^{x+1} - d^x = (s+1) \bullet (d-1)$ = number of active buckets
  5. last active bucket is at address $d^{x+1}-1$
  6. $[d^x, d^{x+1})$ is range of active buckets
- Use two hashing functions
  1. $h(k)$ maps key k uniformly into [0,1) which is the range of the difference in exponent values of addresses in $y = d^{x+1} - d^x$
  2. $y(k)$ maps $h(k)$ into an address in $[s+1,(s+1)+(s+1)\bullet(d-1))$
     - $y(k) = \lfloor d^{x(k)} \rfloor$ with $x(k)$ in range $[\log_d(s+1),\log_d(s+1)+1)$
     - before split, active buckets lay in range $[\log_d s,\log_d s+1)$
     - want to make sure that all key values previously in bucket s – i.e., $x(k)$ in $[\log_d s,\log_d(s+1))$ are rehashed into one of the new buckets with an $x(k)$ value in $[\log_d s+1,\log_d(s+1)+1)$
     - leave other key values in $[\log_d(s+1),\log_d s+1)$ unchanged
     - difficult to choose $x(k)$
       a. $x(k) = \log_d(s+1) + h(k)$
          - drawback: must rehash all keys when a bucket is split

       b. $x(k) = \lceil \log_d(s+1) - h(k) \rceil + h(k)$
          - guarantees that if k is hashed into bucket b $(\geq s+1)$ then it continues to hash there until bucket b is split
          - implies that $x(k)$ is a number in the range $[\log_d(s+1),\log_d(s+1)+1)$ whose fractional part is $h(k)$

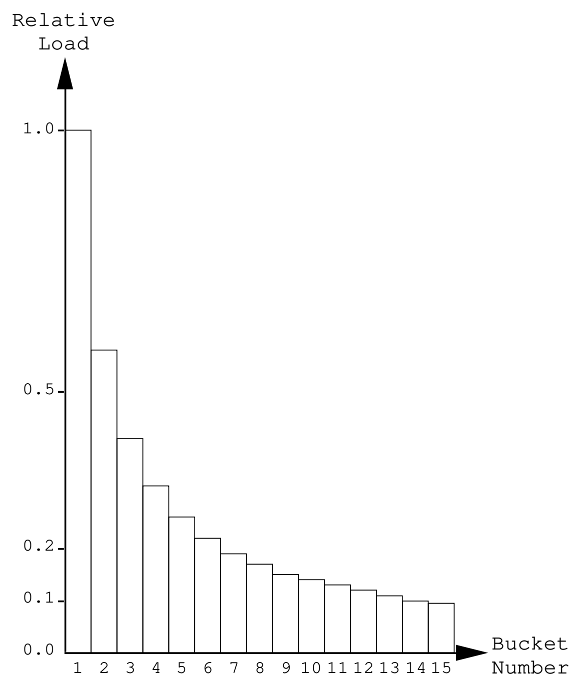
BEHAVIOR OF THE SPIRAL HASHING FUNCTION

- Ex: $d = 2$

Bucket address	Hash interval	Relative load
1	[0.0000,1.0000)	1.0000
2	[0.0000,0.5849)	0.5849
3	[0.5849,1.0000)	0.4151
4	[0.0000,0.3219)	0.3219
5	[0.3219,0.5849)	0.2630
6	[0.5849,0.8073)	0.2224
7	[0.8073,1.0000)	0.1927
8	[0.0000,0.1699)	0.1699
9	[0.1699,0.3219)	0.1520
10	[0.3219,0.4594)	0.1375
11	[0.4594,0.5849)	0.1255
12	[0.5849,0.7004)	0.1155
13	[0.7004,0.8073)	0.1069
14	[0.8073,0.9068)	0.0995
15	[0.9068,1.0000)	0.0932



The function $y = 2^x$

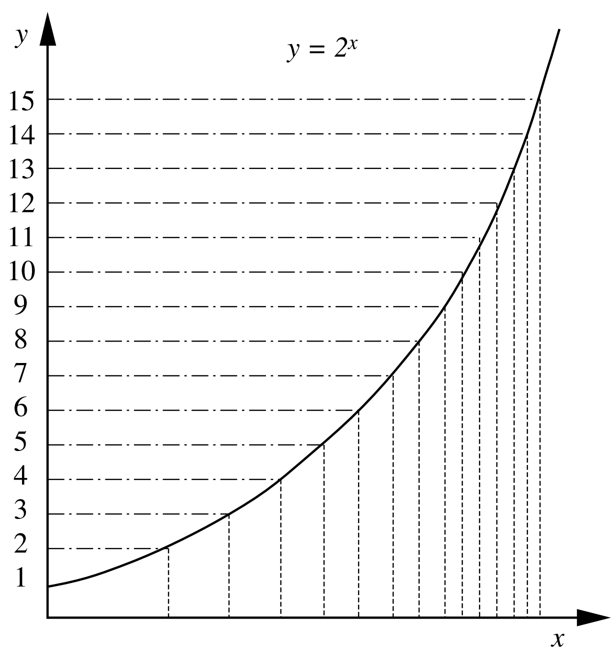


Relative load of buckets 1-15

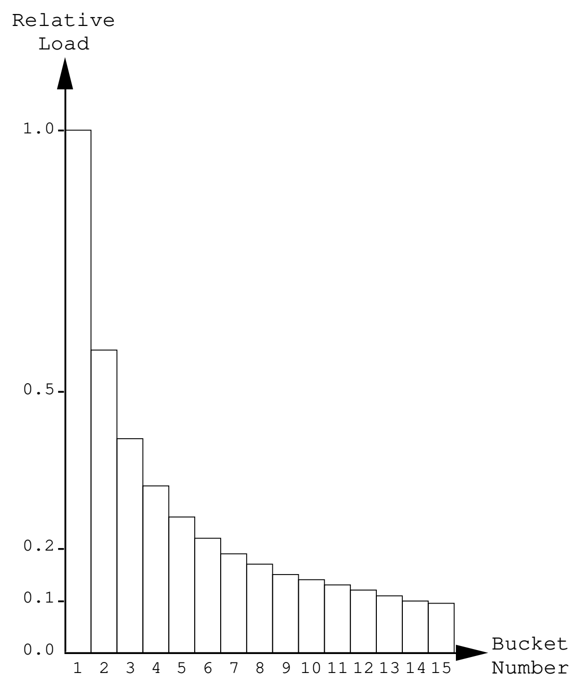
BEHAVIOR OF THE SPIRAL HASHING FUNCTION

- Ex: $d = 2$

Bucket address	Hash interval	Relative load
1	[0.0000,1.0000)	1.0000
2	[0.0000,0.5849)	0.5849
3	[0.5849,1.0000)	0.4151
4	[0.0000,0.3219)	0.3219
5	[0.3219,0.5849)	0.2630
6	[0.5849,0.8073)	0.2224
7	[0.8073,1.0000)	0.1927
8	[0.0000,0.1699)	0.1699
9	[0.1699,0.3219)	0.1520
10	[0.3219,0.4594)	0.1375
11	[0.4594,0.5849)	0.1255
12	[0.5849,0.7004)	0.1155
13	[0.7004,0.8073)	0.1069
14	[0.8073,0.9068)	0.0995
15	[0.9068,1.0000)	0.0932



The function $y = 2^x$



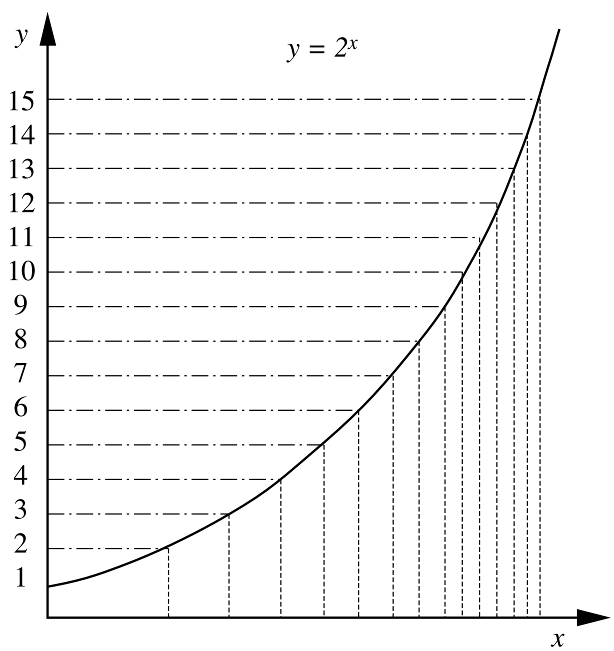
Relative load of buckets 1-15

- Split bucket 3

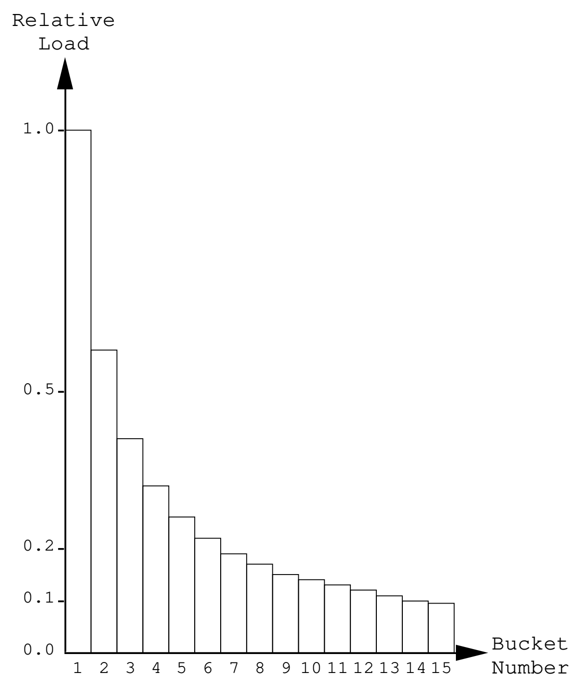
BEHAVIOR OF THE SPIRAL HASHING FUNCTION

- Ex: $d = 2$

Bucket address	Hash interval	Relative load
1	[0.0000,1.0000)	1.0000
2	[0.0000,0.5849)	0.5849
3	[0.5849,1.0000)	0.4151
4	[0.0000,0.3219)	0.3219
5	[0.3219,0.5849)	0.2630
6	[0.5849,0.8073)	0.2224
7	[0.8073,1.0000)	0.1927
8	[0.0000,0.1699)	0.1699
9	[0.1699,0.3219)	0.1520
10	[0.3219,0.4594)	0.1375
11	[0.4594,0.5849)	0.1255
12	[0.5849,0.7004)	0.1155
13	[0.7004,0.8073)	0.1069
14	[0.8073,0.9068)	0.0995
15	[0.9068,1.0000)	0.0932



The function $y = 2^x$



Relative load of buckets 1-15

- Split bucket 3
- Yields buckets 6 and 7

TABLES FOR EXAMPLE OF SPIRAL HASHING

- Use bit interleaving to form a value *k* - i.e., take one bit from the binary representation of the *x* coordinate value and one bit from the binary representation of the *y* coordinate value and alternate them
- Use city coordinate values and divide by 12.5 so that each coordinate value requires just three binary digits
- Use *h(k) = k/64* which has same effect as reverse bit interleaving and behavior is analogous to OPLH
- Example with *y* being more significant than *x*

City	x	y	f(z)=z div 12.5 x	y	k/64	
Chicago	35	42	2	3	14	.21875
Mobile	52	10	4	0	16	.25
Toronto	62	77	4	6	56	.875
Buffalo	82	65	6	5	54	.84375
Denver	5	45	0	3	10	.15625
Omaha	27	35	2	2	12	.1875
Atlanta	85	15	6	1	22	.34375
Miami	90	5	7	0	21	.328125

TABLES FOR EXAMPLE OF SPIRAL HASHING

- Use bit interleaving to form a value *k* - i.e., take one bit from the binary representation of the *x* coordinate value and one bit from the binary representation of the *y* coordinate value and alternate them
- Use city coordinate values and divide by 12.5 so that each coordinate value requires just three binary digits
- Use *h(k) = k/64* which has same effect as reverse bit interleaving and behavior is analogous to OPLH
- Example with *y* being more significant than *x*

City	x	y	f(z)=z div 12.5 x	y	k/64	
Chicago	35	42	2	3	14	.21875
Mobile	52	10	4	0	16	.25
Toronto	62	77	4	6	56	.875
Buffalo	82	65	6	5	54	.84375
Denver	5	45	0	3	10	.15625
Omaha	27	35	2	2	12	.1875
Atlanta	85	15	6	1	22	.34375
Miami	90	5	7	0	21	.328125

1	1	0	x

- Ex: `Atlanta` (6,1)

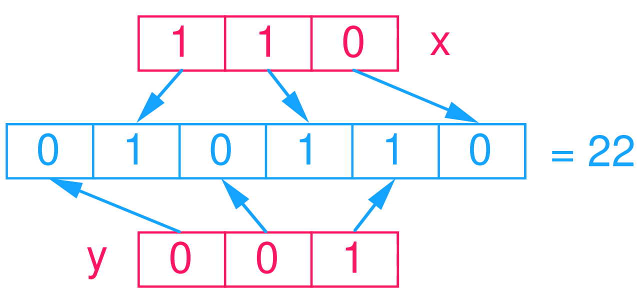
y	0	0	1

TABLES FOR EXAMPLE OF SPIRAL HASHING

- Use bit interleaving to form a value *k* - i.e., take one bit from the binary representation of the *x* coordinate value and one bit from the binary representation of the *y* coordinate value and alternate them
- Use city coordinate values and divide by 12.5 so that each coordinate value requires just three binary digits
- Use *h(k) = k/64* which has same effect as reverse bit interleaving and behavior is analogous to OPLH
- Example with *y* being more significant than *x*

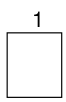
City	x	y	f(z)=z div 12.5			
			x	y	k/64	
Chicago	35	42	2	3	14	.21875
Mobile	52	10	4	0	16	.25
Toronto	62	77	4	6	56	.875
Buffalo	82	65	6	5	54	.84375
Denver	5	45	0	3	10	.15625
Omaha	27	35	2	2	12	.1875
Atlanta	85	15	6	1	22	.34375
Miami	90	5	7	0	21	.328125

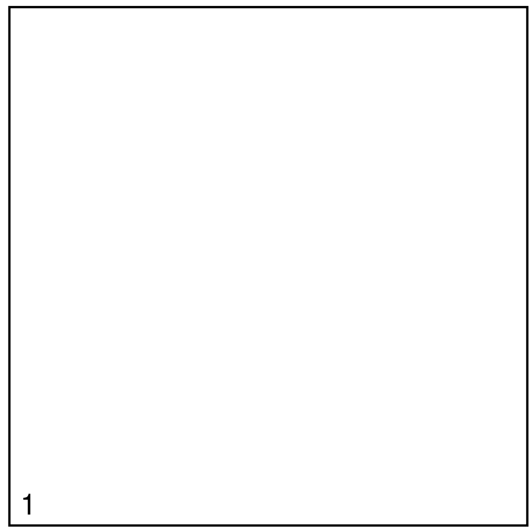
- Ex: `Atlanta` (6,1)



MECHANICS OF EXAMPLE OF SPIRAL HASHING

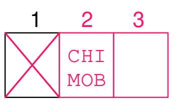
- Assume $d = 2$ and primary and overflow bucket capacity of 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 1 exists

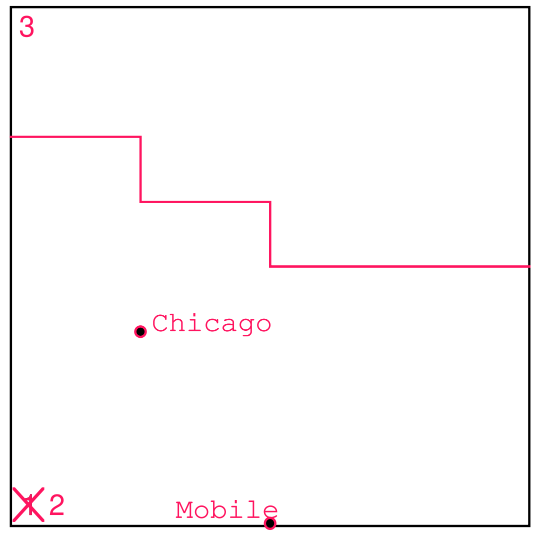




MECHANICS OF EXAMPLE OF SPIRAL HASHING

- Assume $d = 2$ and primary and overflow bucket capacity of 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 1 exists

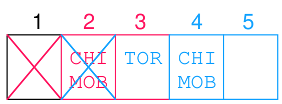


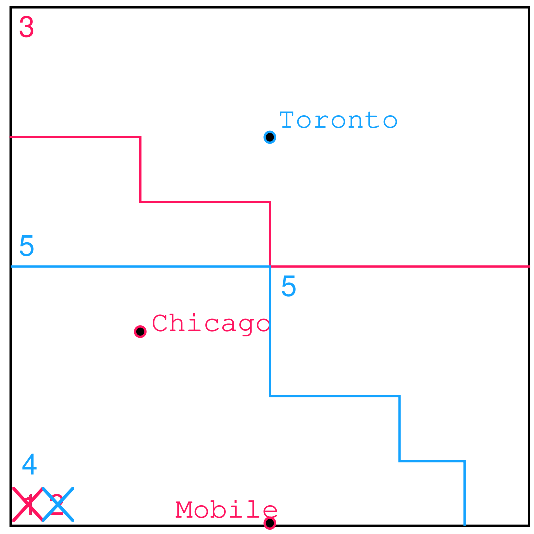


- Insert `Chicago` (.22) and `Mobile` (.25): τ=1 and split bucket 1 creating buckets 2 and 3; `Chicago` and `Mobile` are moved to bucket 2

MECHANICS OF EXAMPLE OF SPIRAL HASHING

- Assume $d = 2$ and primary and overflow bucket capacity of 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 1 exists

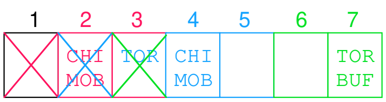


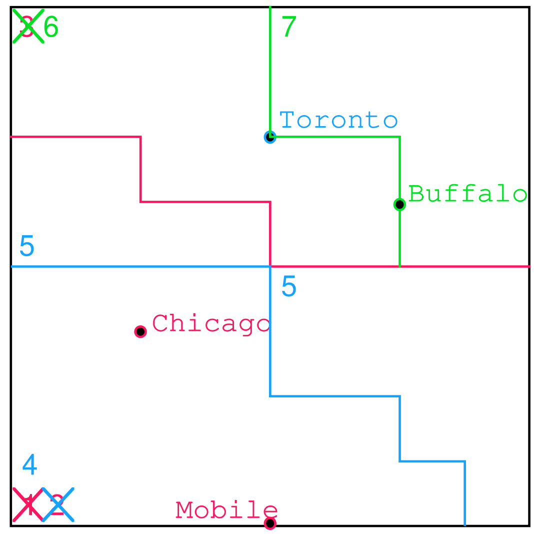


- Insert `Chicago` (.22) and `Mobile` (.25): τ=1 and split bucket 1 creating buckets 2 and 3; `Chicago` and `Mobile` are moved to bucket 2
- Insert `Toronto` (.87) into bucket 3: τ=0.75 and split bucket 2 creating buckets 3 and 4; `Chicago` and `Mobile` are moved to bucket 4

MECHANICS OF EXAMPLE OF SPIRAL HASHING

- Assume *d = 2* and primary and overflow bucket capacity of 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 1 exists

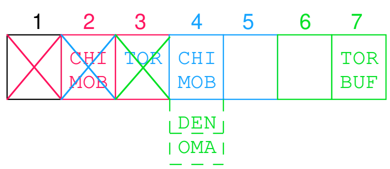


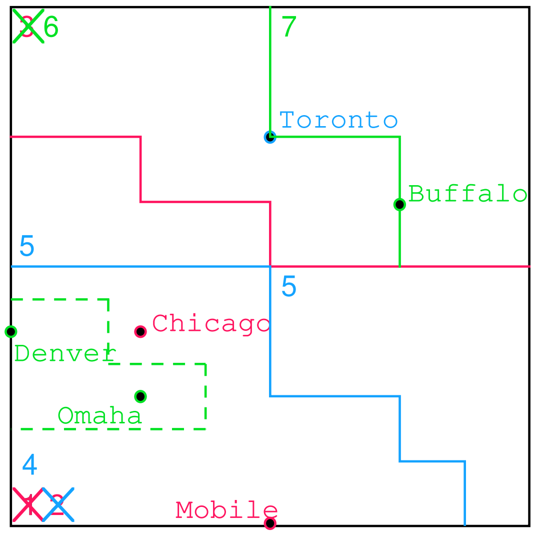


- Insert `Chicago` (.22) and `Mobile` (.25): τ=1 and split bucket 1 creating buckets 2 and 3; `Chicago` and `Mobile` are moved to bucket 2
- Insert `Toronto` (.87) into bucket 3: τ=0.75 and split bucket 2 creating buckets 3 and 4; `Chicago` and `Mobile` are moved to bucket 4
- Insert `Buffalo` (.84) into bucket 3: τ=0.67 and split bucket 3 creating buckets 6 and 7; `Toronto` and `Buffalo` are moved to bucket 7

MECHANICS OF EXAMPLE OF SPIRAL HASHING

- Assume *d* = *2* and primary and overflow bucket capacity of 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 1 exists

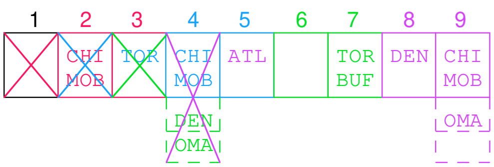


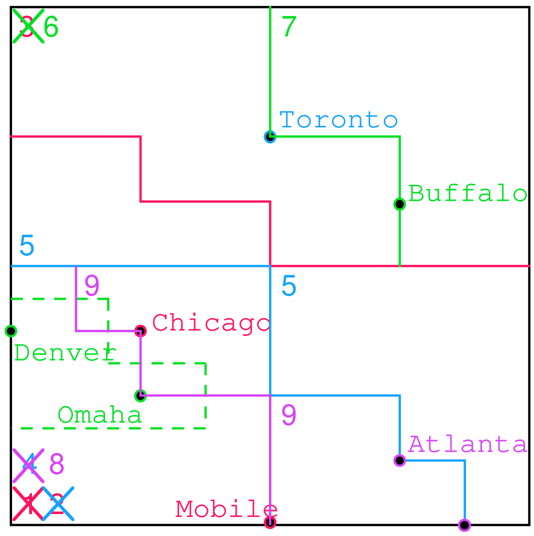


- Insert `Chicago` (.22) and `Mobile` (.25): τ=1 and split bucket 1 creating buckets 2 and 3; `Chicago` and `Mobile` are moved to bucket 2
- Insert `Toronto` (.87) into bucket 3: τ=0.75 and split bucket 2 creating buckets 3 and 4; `Chicago` and `Mobile` are moved to bucket 4
- Insert `Buffalo` (.84) into bucket 3: τ=0.67 and split bucket 3 creating buckets 6 and 7; `Toronto` and `Buffalo` are moved to bucket 7
- Insert `Denver` (.16) and `Omaha` (.19) into bucket 4's overflow area

MECHANICS OF EXAMPLE OF SPIRAL HASHING

- Assume $d = 2$ and primary and overflow bucket capacity of 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 1 exists

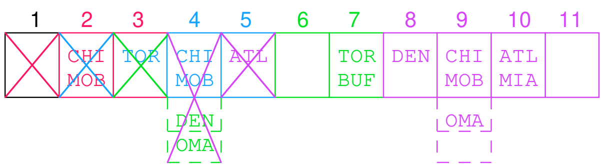


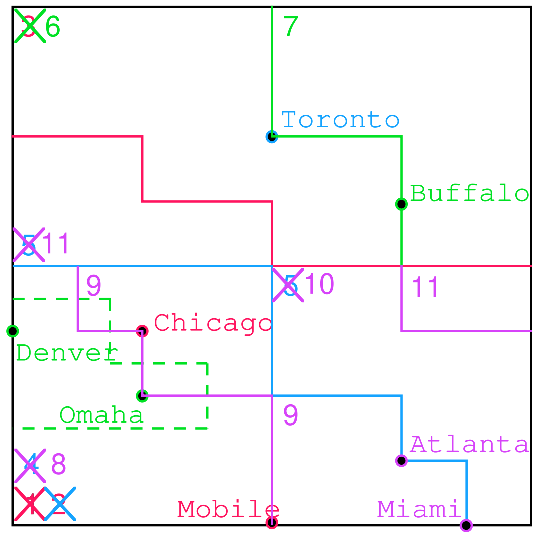


- Insert `Chicago` (.22) and `Mobile` (.25): τ=1 and split bucket 1 creating buckets 2 and 3; `Chicago` and `Mobile` are moved to bucket 2
- Insert `Toronto` (.87) into bucket 3: τ=0.75 and split bucket 2 creating buckets 3 and 4; `Chicago` and `Mobile` are moved to bucket 4
- Insert `Buffalo` (.84) into bucket 3: τ=0.67 and split bucket 3 creating buckets 6 and 7; `Toronto` and `Buffalo` are moved to bucket 7
- Insert `Denver` (.16) and `Omaha` (.19) into bucket 4's overflow area
- Insert `Atlanta` (.34) into bucket 5: τ=0.7 and split bucket 4 creating buckets 8 and 9; `Denver` is moved to bucket 8, while `Chicago`, `Mobile`, and `Omaha` are moved to bucket 9 which causes it to overflow

MECHANICS OF EXAMPLE OF SPIRAL HASHING

- Assume *d* = *2* and primary and overflow bucket capacity of 2
- A bucket is split whenever $\tau \geq \alpha = 0.66$
- Initially, only bucket 1 exists





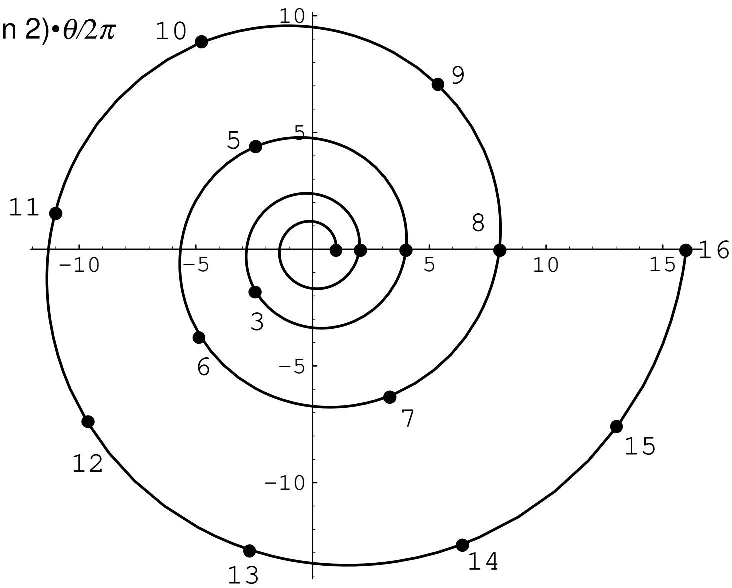
- Insert `Chicago` (.22) and `Mobile` (.25): τ=1 and split bucket 1 creating buckets 2 and 3; `Chicago` and `Mobile` are moved to bucket 2
- Insert `Toronto` (.87) into bucket 3: τ=0.75 and split bucket 2 creating buckets 3 and 4; `Chicago` and `Mobile` are moved to bucket 4
- Insert `Buffalo` (.84) into bucket 3: τ=0.67 and split bucket 3 creating buckets 6 and 7; `Toronto` and `Buffalo` are moved to bucket 7
- Insert `Denver` (.16) and `Omaha` (.19) into bucket 4's overflow area
- Insert `Atlanta` (.34) into bucket 5: τ=0.7 and split bucket 4 creating buckets 8 and 9; `Denver` is moved to bucket 8, while `Chicago`, `Mobile`, and `Omaha` are moved to bucket 9 which causes it to overflow
- Insert `Miami` (.33) into bucket 5: τ=0.67 and split bucket 5 creating buckets 10 and 11; Move `Atlanta` and `Miami` to bucket 10

COMPARISON OF SPIRAL AND LINEAR HASHING

- Main advantage of spiral hashing over linear hashing is that the bucket being split is the one most likely to overflow
- Disadvantages of spiral hashing:
  1. the buckets that have been split are not reused
     - overcome by using a mapping between logical and physical addresses
  2. expensive to calculate function $y = d^x$
     - overcome by use of an approximation

WHY SPIRAL?

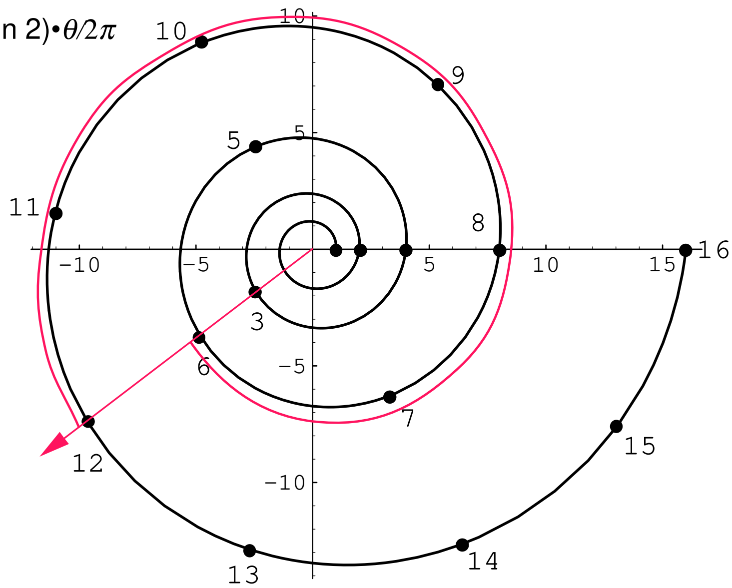
- Can rewrite $y = \lfloor d^x \rfloor$ as $\lfloor \rho = e^{j\theta} \rfloor$ using polar coordinates which yields the equation of a spiral
- Ex: $\rho = e^{(\ln 2) \cdot \theta / 2\pi}$



- Polar coordinates mean that the active buckets are always within one complete arc of the spiral – i.e., $\theta = 2\pi$
- Mechanics of a bucket split
  1. let first active bucket be at $a = \lfloor e^{j \cdot b} \rfloor$ (i.e., $\theta = b$)
  2. last active bucket is at $c = \lceil e^{j \cdot (b+2\pi)} \rceil - 1$
  3. bucket split means that the contents of the active bucket at $\rho = a$ (i.e., $\theta = b$) are distributed into buckets $c + 1$ through g where $g = \lfloor e^{j \cdot (b+2\pi+\phi)} \rfloor$ and ϕ is the solution of $a + 1 = e^{j \cdot (b+\phi)}$ – i.e., $\phi = (\ln (a + 1))/j - b$
  4. buckets $a + 1$ through g are now the active buckets
- Ex: $d = 2$

WHY SPIRAL?

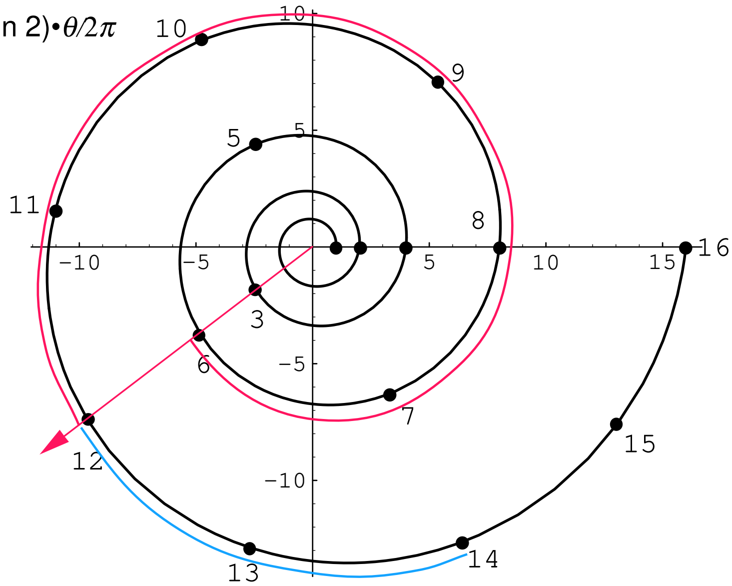
- Can rewrite $y = \lfloor d^x \rfloor$ as $\lfloor \rho = e^{j\theta} \rfloor$ using polar coordinates which yields the equation of a spiral
- Ex: $\rho = e^{(\ln 2) \cdot \theta/2\pi}$



- Polar coordinates mean that the active buckets are always within one complete arc of the spiral – i.e., $\theta = 2\pi$
- Mechanics of a bucket split
  1. let first active bucket be at $a = \lfloor e^{j \cdot b} \rfloor$ (i.e., $\theta = b$)
  2. last active bucket is at $c = \lceil e^{j \cdot (b+2\pi)} \rceil - 1$
  3. bucket split means that the contents of the active bucket at $\rho = a$ (i.e., $\theta = b$) are distributed into buckets $c + 1$ through g where $g = \lfloor e^{j \cdot (b+2\pi+\phi)} \rfloor$ and ϕ is the solution of $a + 1 = e^{j \cdot (b+\phi)}$ – i.e., $\phi = (\ln (a + 1))/j - b$
  4. buckets $a + 1$ through g are now the active buckets
- Ex: $d = 2$
  1. active buckets 6 through 11

WHY SPIRAL?

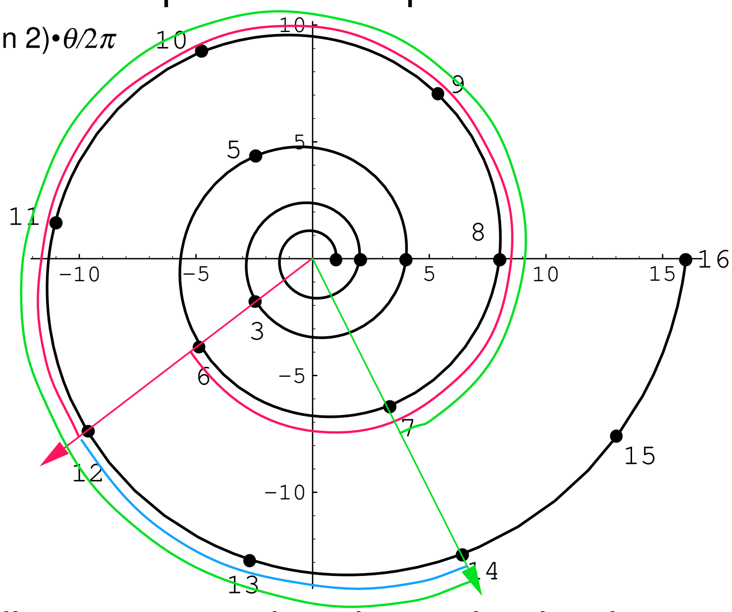
- Can rewrite $y = \lfloor d^x \rfloor$ as $\lfloor \rho = e^{j\theta} \rfloor$ using polar coordinates which yields the equation of a spiral
- Ex: $\rho = e^{(\ln 2) \cdot \theta / 2\pi}$



- Polar coordinates mean that the active buckets are always within one complete arc of the spiral – i.e., $\theta = 2\pi$
- Mechanics of a bucket split
  1. let first active bucket be at $a = \lfloor e^{j \cdot b} \rfloor$ (i.e., $\theta = b$)
  2. last active bucket is at $c = \lceil e^{j \cdot (b+2\pi)} \rceil - 1$
  3. bucket split means that the contents of the active bucket at $\rho = a$ (i.e., $\theta = b$) are distributed into buckets $c + 1$ through g where $g = \lfloor e^{j \cdot (b+2\pi+\phi)} \rfloor$ and ϕ is the solution of $a + 1 = e^{j \cdot (b+\phi)}$ – i.e., $\phi = (\ln\,(a+1))/j - b$
  4. buckets $a + 1$ through g are now the active buckets
- Ex: $d = 2$
  1. active buckets 6 through 11
  2. split bucket 6 to yield buckets 12 and 13

WHY SPIRAL?

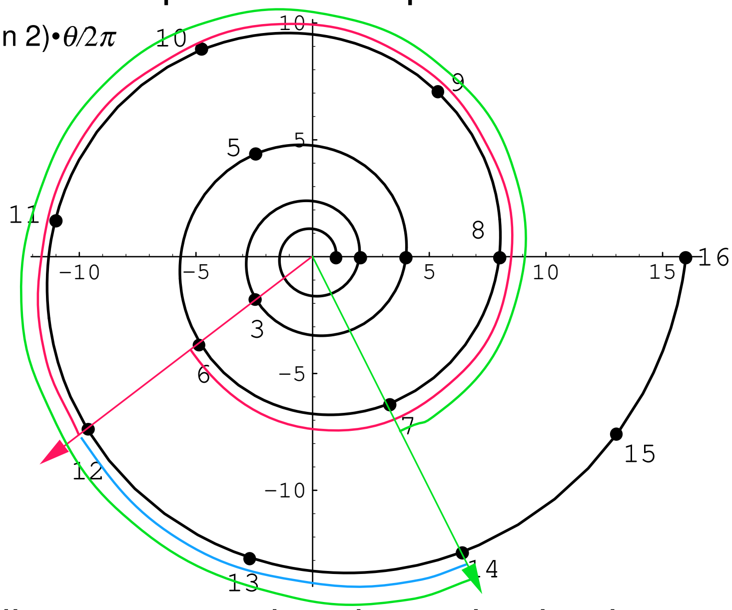
- Can rewrite $y = \lfloor d^x \rfloor$ as $\lfloor \rho = e^{j\theta} \rfloor$ using polar coordinates which yields the equation of a spiral
- Ex: $\rho = e^{(\ln 2) \cdot \theta / 2\pi}$



- Polar coordinates mean that the active buckets are always within one complete arc of the spiral – i.e., $\theta = 2\pi$
- Mechanics of a bucket split
  1. let first active bucket be at $a = \lfloor e^{j \cdot b} \rfloor$ (i.e., $\theta = b$)
  2. last active bucket is at $c = \lceil e^{j \cdot (b+2\pi)} \rceil - 1$
  3. bucket split means that the contents of the active bucket at $\rho = a$ (i.e., $\theta = b$) are distributed into buckets $c + 1$ through g where $g = \lfloor e^{j \cdot (b+2\pi+\phi)} \rfloor$ and ϕ is the solution of $a + 1 = e^{j \cdot (b+\phi)}$ – i.e., $\phi = (\ln(a+1))/j - b$
  4. buckets $a + 1$ through g are now the active buckets
- Ex: $d = 2$
  1. active buckets 6 through 11
  2. split bucket 6 to yield buckets 12 and 13
  3. active buckets 7 through 13

WHY SPIRAL?

- Can rewrite $y = \lfloor d^x \rfloor$ as $\lfloor \rho = e^{j\theta} \rfloor$ using polar coordinates which yields the equation of a spiral
- Ex: $\rho = e^{(\ln 2)\bullet\theta/2\pi}$



- Polar coordinates mean that the active buckets are always within one complete arc of the spiral – i.e., $\theta = 2\pi$
- Mechanics of a bucket split
  1. let first active bucket be at $a = \lfloor e^{j\bullet b} \rfloor$ (i.e., $\theta = b$)
  2. last active bucket is at $c = \lceil e^{j\bullet(b+2\pi)} \rceil - 1$
  3. bucket split means that the contents of the active bucket at $\rho = a$ (i.e., $\theta = b$) are distributed into buckets $c + 1$ through g where $g = \lfloor e^{j\bullet(b+2\pi+\phi)} \rfloor$ and ϕ is the solution of $a + 1 = e^{j\bullet(b+\phi)}$ – i.e., $\phi = (\ln(a+1))/j - b$
  4. buckets $a + 1$ through g are now the active buckets
- Ex: $d = 2$
  1. active buckets 6 through 11
  2. split bucket 6 to yield buckets 12 and 13
  3. active buckets 7 through 13
- Observations
  1. insead of $h(k)$ uniformly mapping key k into [0,1), use $h_\theta(k)$ to uniformly map k into $[0,2\pi)$
  2. length of arc has constant value between successive integer values of ρ