Geometric Objects

- **Scalars**: 1-d poin
- **Point**: location in d-dimensional space. *d*-tuple of scalars. $P=(x_1,x_2,x_3...,x_d)$
  - arrays: double p[d];
  - structures: struct { double x, y, z; }
  - good compromise:

```
struct Point {
  const int DIM = 3;
  double coord[DIM];
};
```

- **Vectors**: direction and magnitude (length) in that direction.

Lines, Segments, Rays

- **Line**: infinite in both directions
  - $y = mx + b$ [slope m, intercept b]
  - $ax + by = c$
  - In higher dimensions, any two points define a line.
- **Ray**: infinite in one direction
- **Segment**: finite in both directions
- **Polygons**: cycle of joined line segments
  - *simple* if they don't corss
  - *convex* if any line segment connecting two points on its surface lies entirely within the shape.
  - *convex hull* of a set of points P: smallest convex set that contains P

What's a good representation for a polygon?

circularly linked list of points

Types of Queries

- Is the object in the set?
- What is the closest object to a given point?
- What objects does a query object intersect with?
- What is the first object hit by the given ray? [Ray shooting]
- What objects contain P?
- What objects are in a given range? [range queries]

Applications of Geometric / Spatial Data Structs.

- Computer graphics, games, movies
- computer vision, CAD, street maps (google maps / google Earth)
- Human-computer interface design (windowing systems)
- Virtual reality
- Visualization (graphing complex functions)

Why are geometric (spatial) data different?

No natural ordering...

- In 1-d:
  - we usually had a natural ordering on the keys (integers, alphabetical order, ...)
  - But how do you order a set of points?
- Take a step back:
  - In the 1-d case, how did we *use* this ordering?
  - Mostly, it gave us an implicit was to *partition* the data.
- So:
  - Instead of explicitly ordering and implicitly partitioning, we usually: explicitly partition.
  - Partitioning is very natural in geometric spaces.

Why are geometric (spatial) data different?

Static case also interesting...

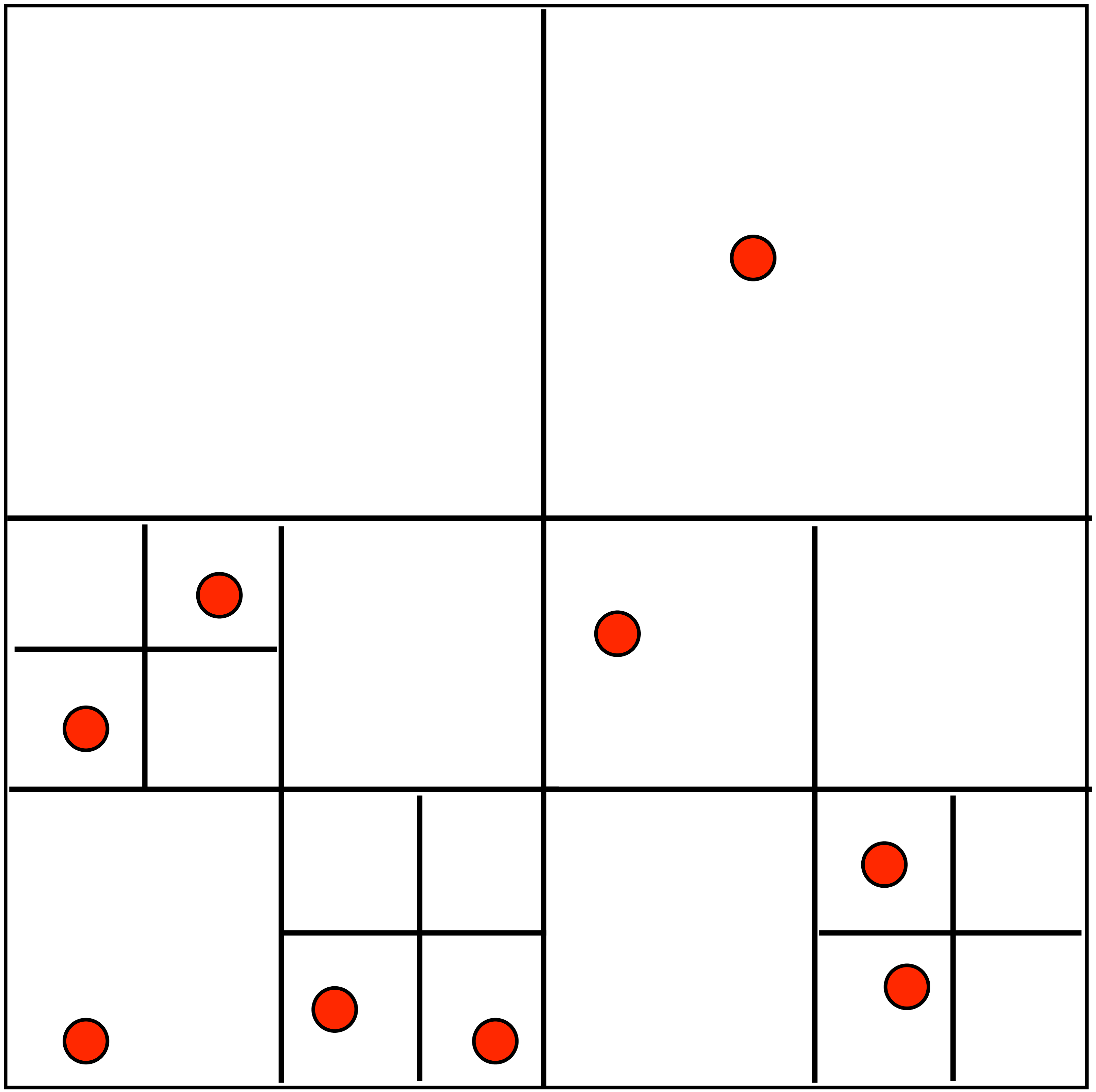
- In 1-d:
  - usually the static case (all data known at start) is not very interesting
  - can be solved by sorting the data
    (heaps => sorted lists, balanced trees => binary search)
- With geometric data,
  - it's sometimes hard to answer queries even if all data are known (what's the analog of binary search for a set of points?)
  - Therefore, emphasize updates less (though we'll still consider them)
  - Model: preprocess the data (may be "slow" like
    O(n log n)) and then have efficient answers to queries.

Point Data Sets – Today

- Data we want to store is a collection of *d*-dimensional points.
  - We'll focus on 2-d for now (hard to draw anything else)

- Simplest query: "Is point P in the collection?"

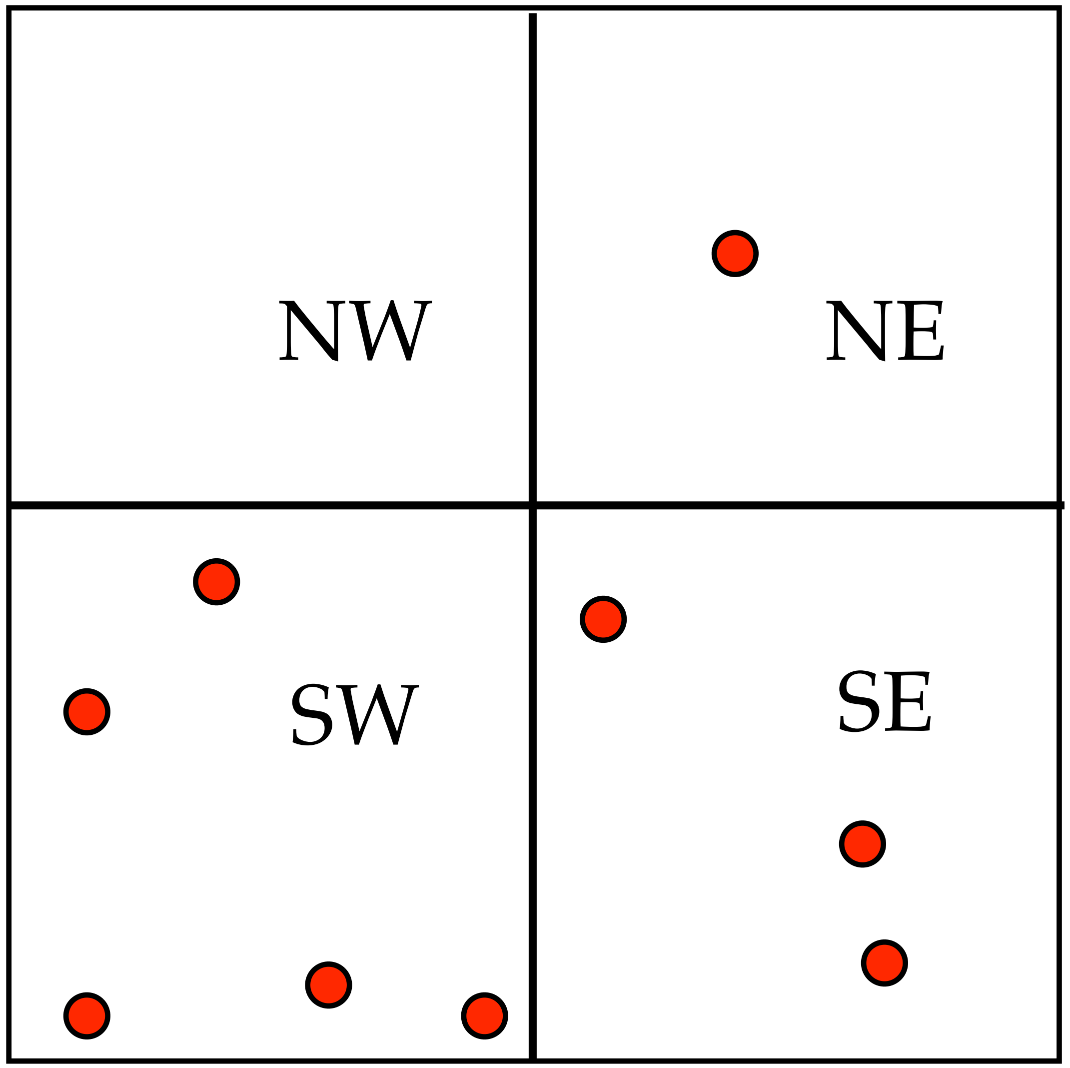
PR Quadtrees

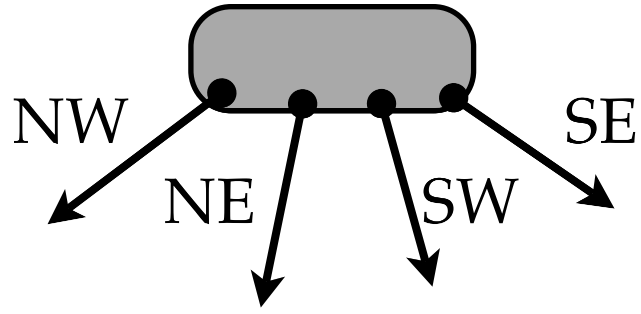


PR Quadtrees (Point-Region)

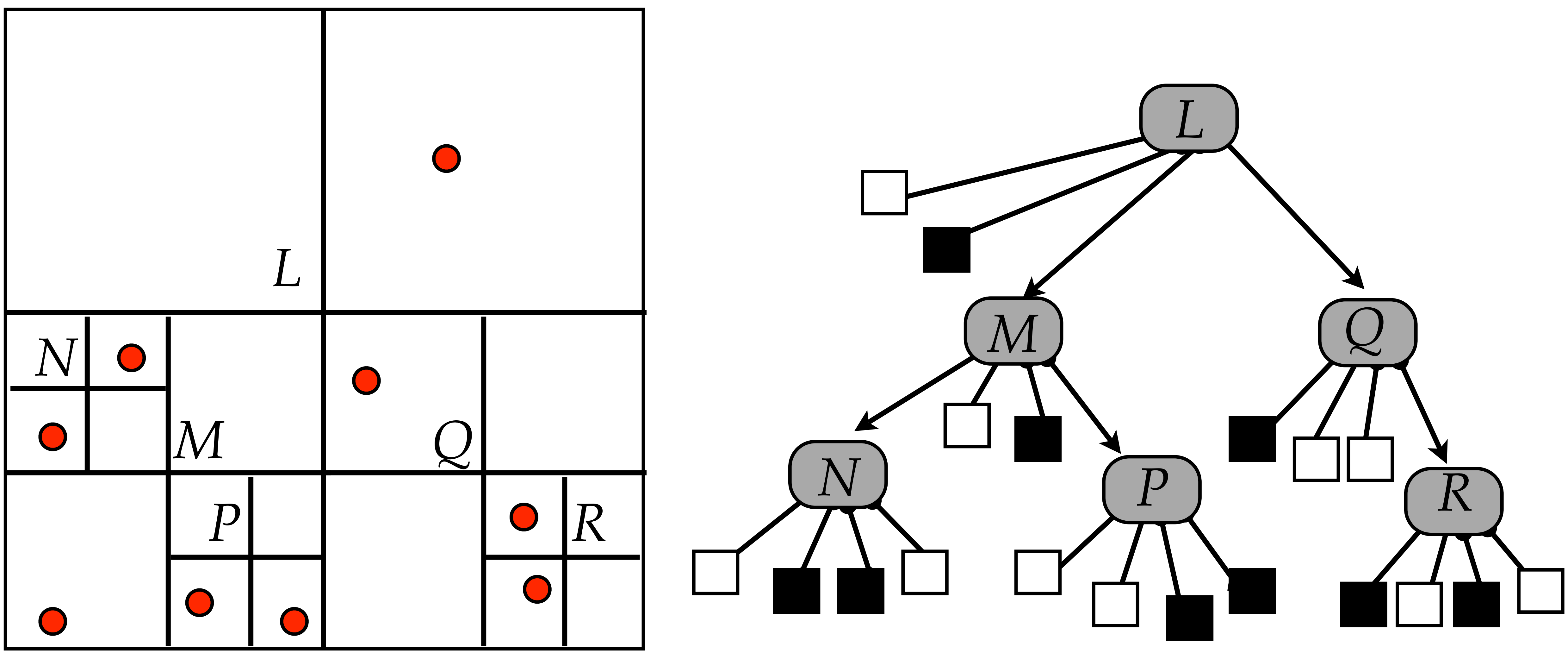
- Recursively subdivide cells into 4 equal-sized subcells until a cell has only one point in it.
- Each division results in a single node with 4 child pointers.
- When cell contains no points, add special "no-point" node.
- When cell contains 1 point, add node containing point + data associated with that point (perhaps a pointer out to a bigger data record).

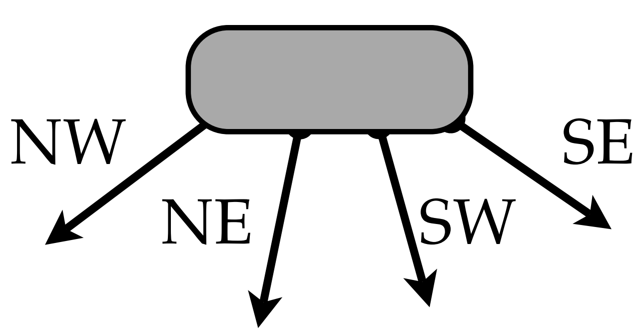
PR Quadtrees Internal Nodes



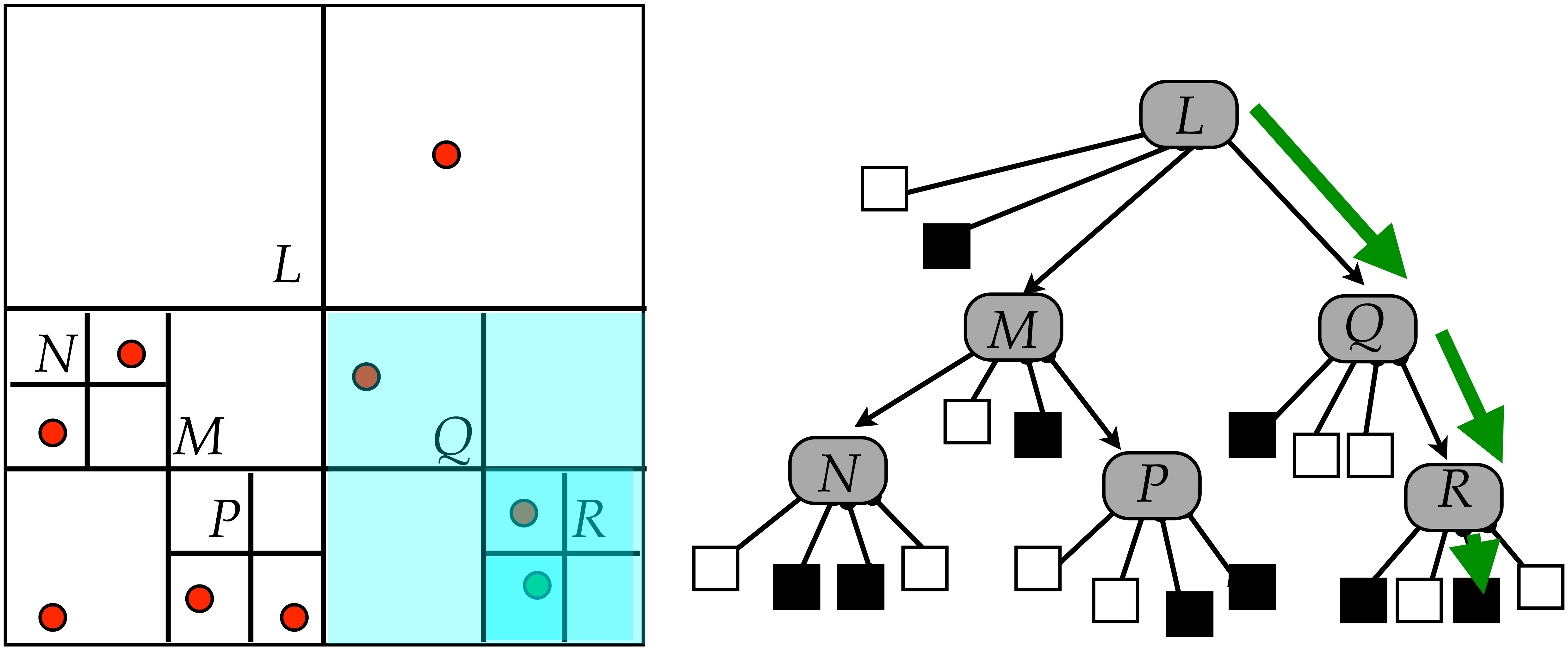


PR Quadtrees





Find in PR Quadtrees



Insert **in PR Quadtrees**

- insert(P):
  - find(P)
  - if cell where P would go is empty, then add P to it (change from □ to ■)
  - If cell where P would go has a point Q in it, repeatedly split until P is separated from Q. Then add P to correct (empty) cell.

- How many times might you have to split?

unbounded in *n*

Delete **in PR Quadtrees**

- delete(P):
  - find(P)
  - If cell that would contain P is empty, return **not found!**
  - Else, remove P (change ■ to □).
  - If at most 1 siblings of the cell has a point, merge siblings into a single cell. Repeat until at least two siblings contain a point.

- A cell “has a point” if it is ■ or .

Features of PR Quadtrees

- Locations of splits don't depend on exact point values (it is a partitioning of space, not of the set of keys)

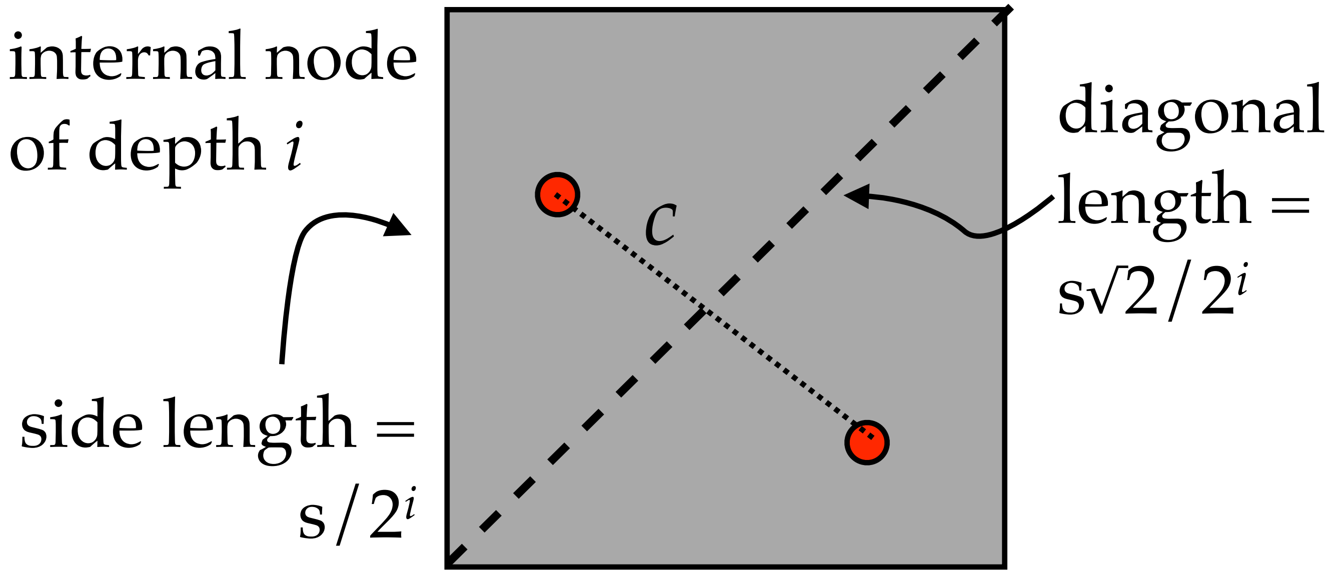
- Leaves should be treated differently that internal nodes because:
  - Empty leaf nodes are common,
  - Only leaves contain data

- Bounding boxes constructed on the fly and passed into the recursive calls.

- Extension: allow a constant $b > 1$ points in a cell (*bucket quadtrees*)

Height Lemma

- if
  - c is the smallest distance between any two points
  - s is the side length of the initial square containing all the points
- Then
  - the depth of a quadtree is $\leq \log(s/c) + 3/2$



Therefore, $s\sqrt{2}/2^i \geq c$

Hence,

$$i \leq \log s\sqrt{2}/c = \log(s/c) + 1/2$$

Height of tree is max depth of internal node + 1, so height $\leq \log(s/c) + 3/2$

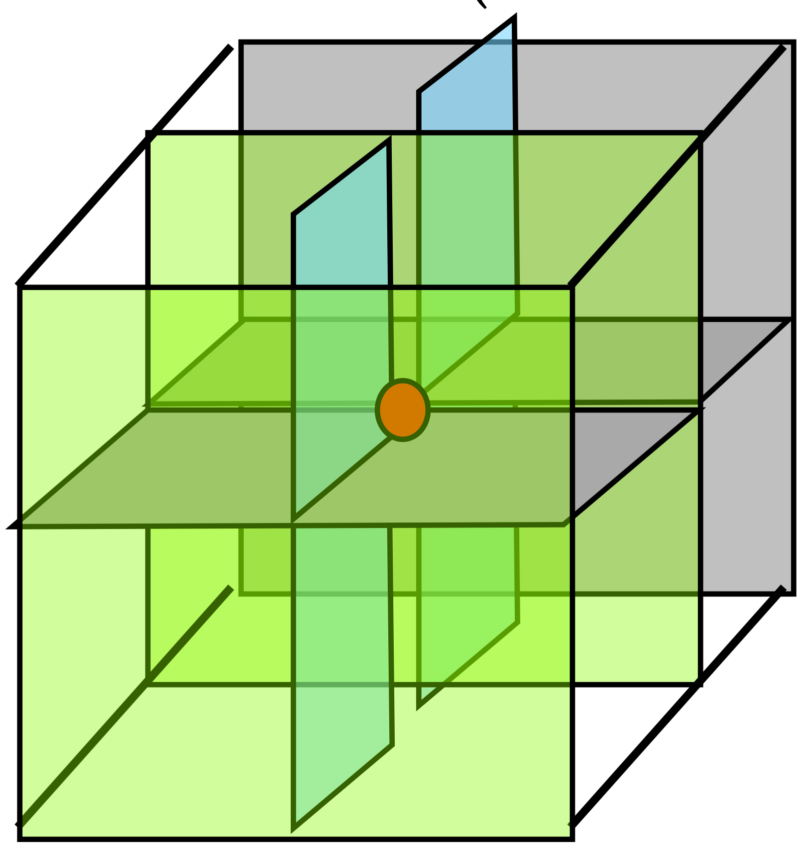
An Advantage of PR quadtrees

- Since partition locations don't depend on the data points, two *different* sets of data can be stored in two *separate* PR quadtrees
  - The partition locations will be "the same"
  - E.g. a quadrant Q_1 in T_1 is either the same as, a superset of, or a subset of any quadrant Q_2 in T_2
  - You cannot get partially overlapping quadrants
  - Recursive algorithms cleaner, e.g.

Issues with PR Quadtrees

- Can be inefficient:
  - two closely spaced points may require a lot of levels in the tree to split them
  - Have to divide up space finely enough so that they end up in different cells
- Generalizing to large dimensions uses a lot of space.
  - <u>*octtree*</u> = Quadtree in 3-D (each node has 8 pointers)

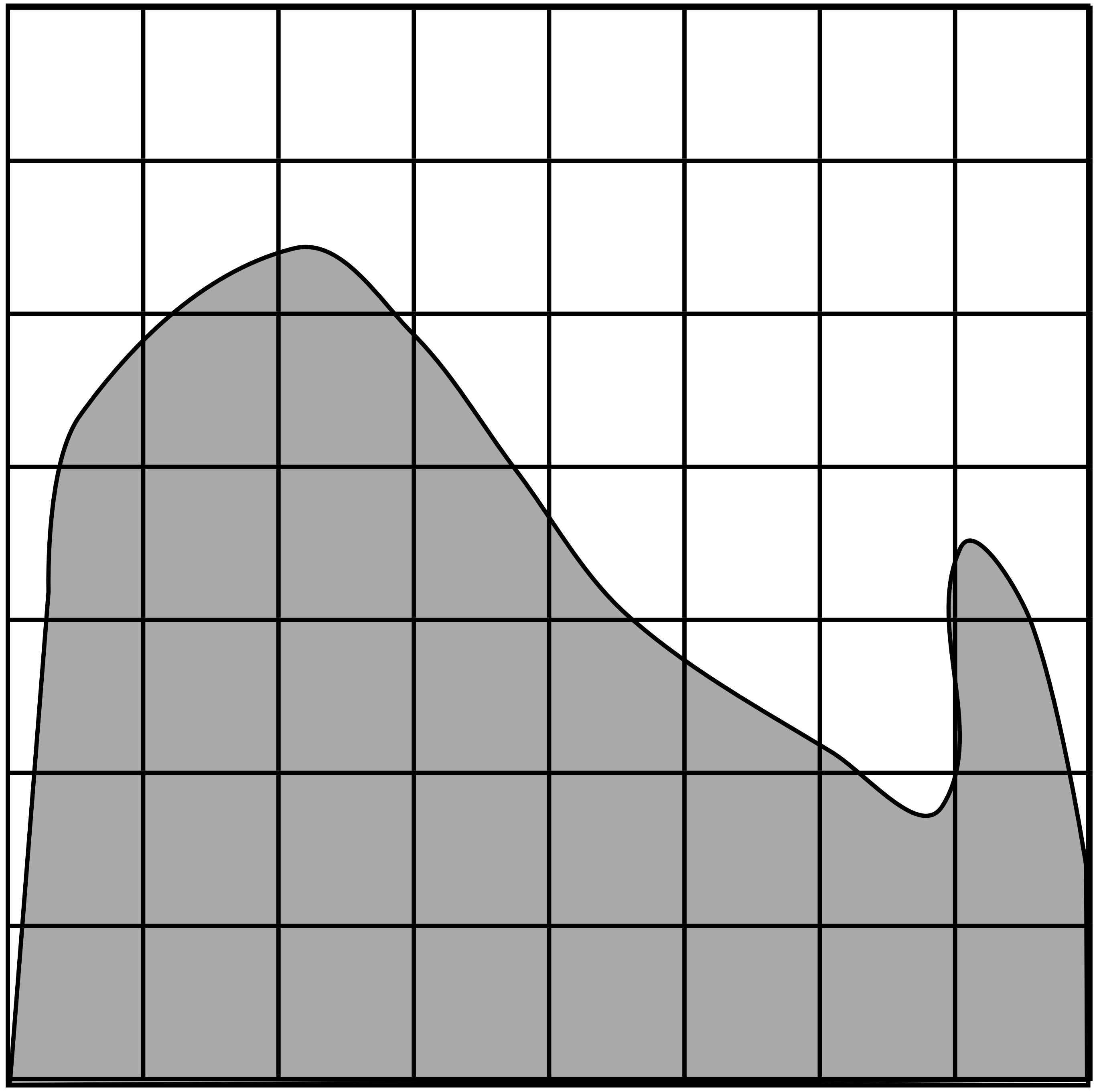
In d dimensions, each node has 2^d pointers!



d = 20 => nodes will ~ 1 million children

Split & Merge Decomposition

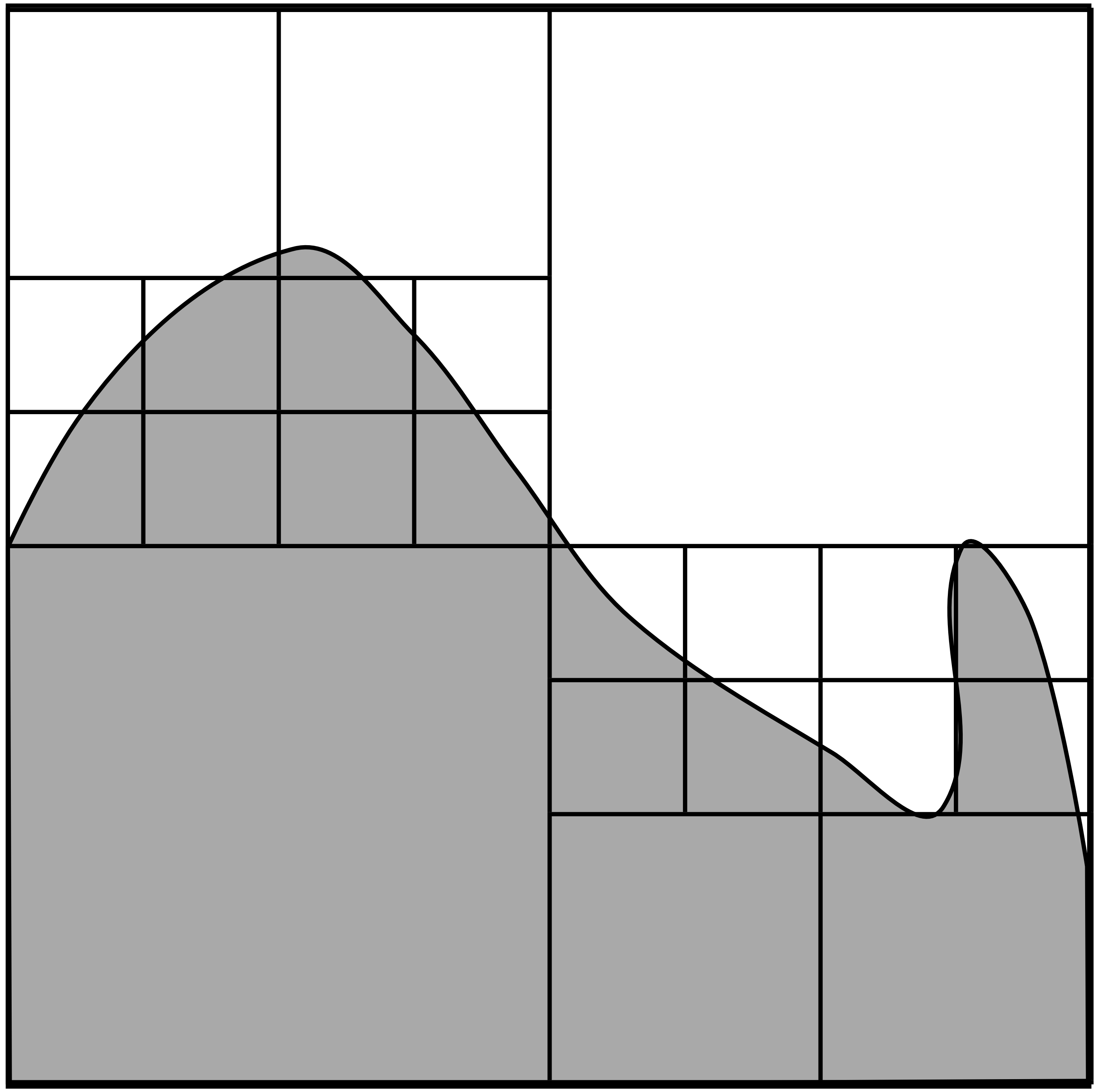
Subdivide into uniform blocks



Split & Merge Decomposition

Subdivide into uniform blocks

Merge similar brothers

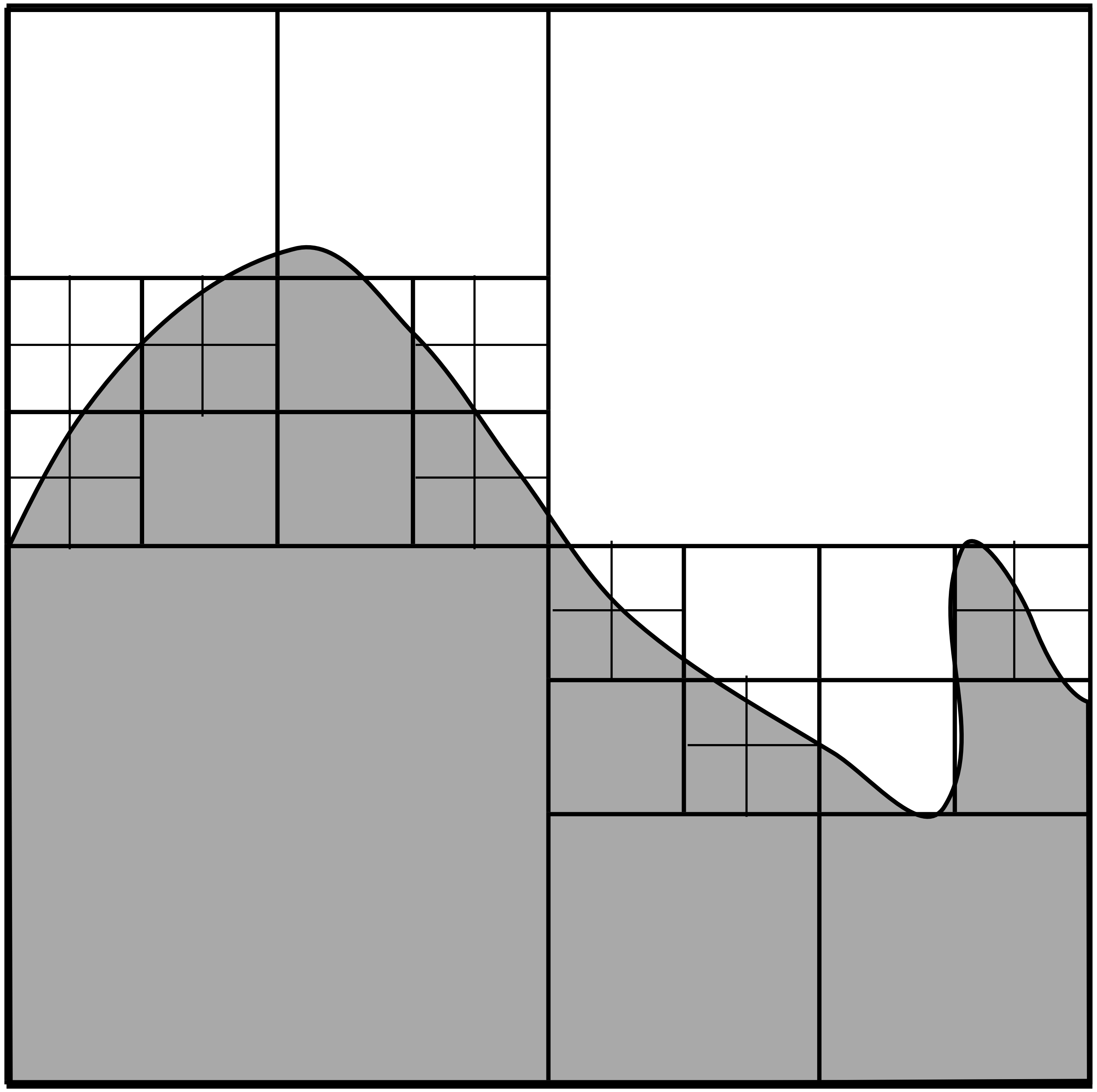


Split & Merge Decomposition

Subdivide into uniform blocks

Merge similar brothers

Subdivide non-homogenous cells



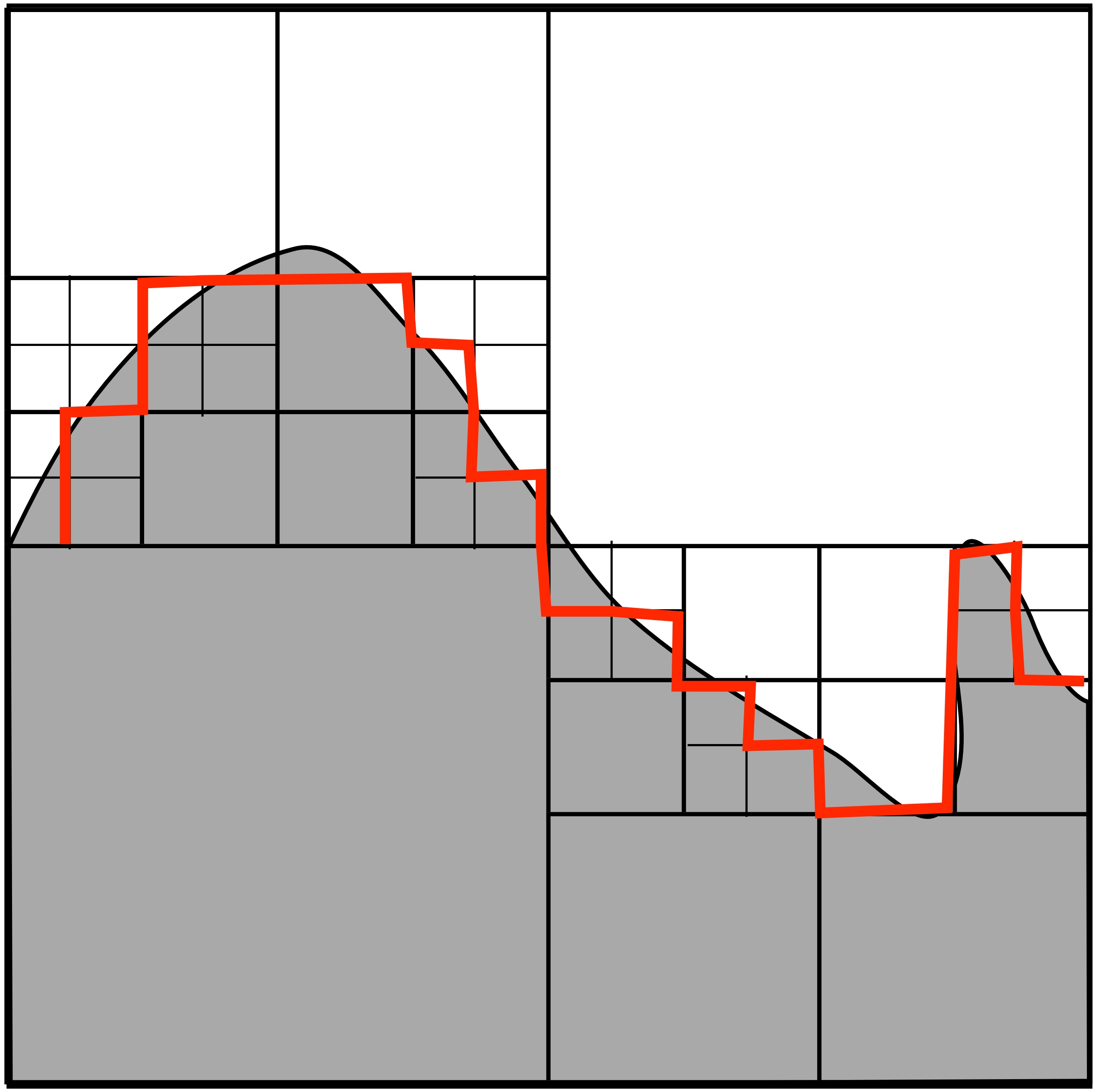
Split & Merge Decomposition

Subdivide into uniform blocks

Merge similar brothers

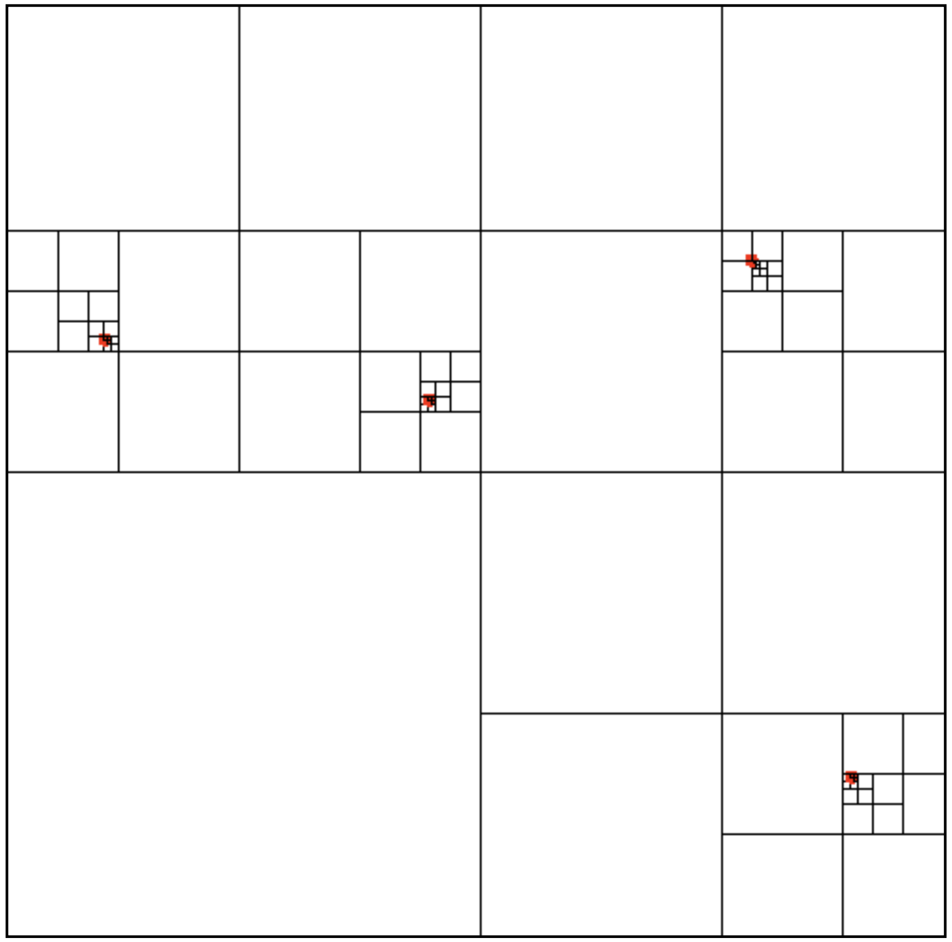
Subdivide non-homogenous cells

Group identical blocks to get regions



MX Quadtrees

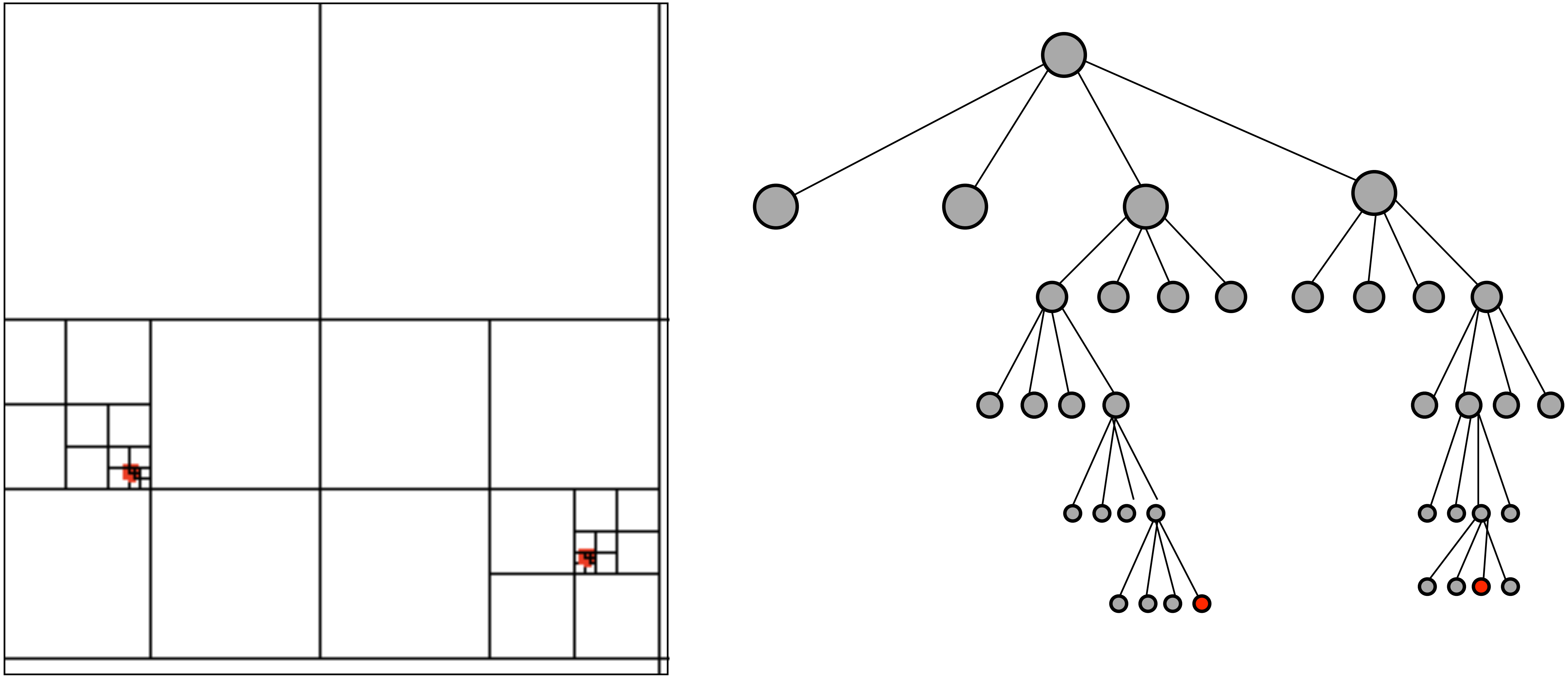
- Good for image data
  - smallest element is known, e.g. a pixel
  - Space is recursively subdivided until smallest unit is reached:
  - **Always subdivide to smallest unit:**



MX (MatriX) Quadtrees

Shape of final tree independent of insertion order

- Points are always at leaves
- All leaves *with points* are the same depth:



MX Quadtree Notes & Applications

- Shape of final tree independent of insertion order

- Can be used to represent a matrix (especially 0/1 matrix)
  - recursive decomposition of matrix (given by the MX tree) can be used for faster matrix transposition and multiplication

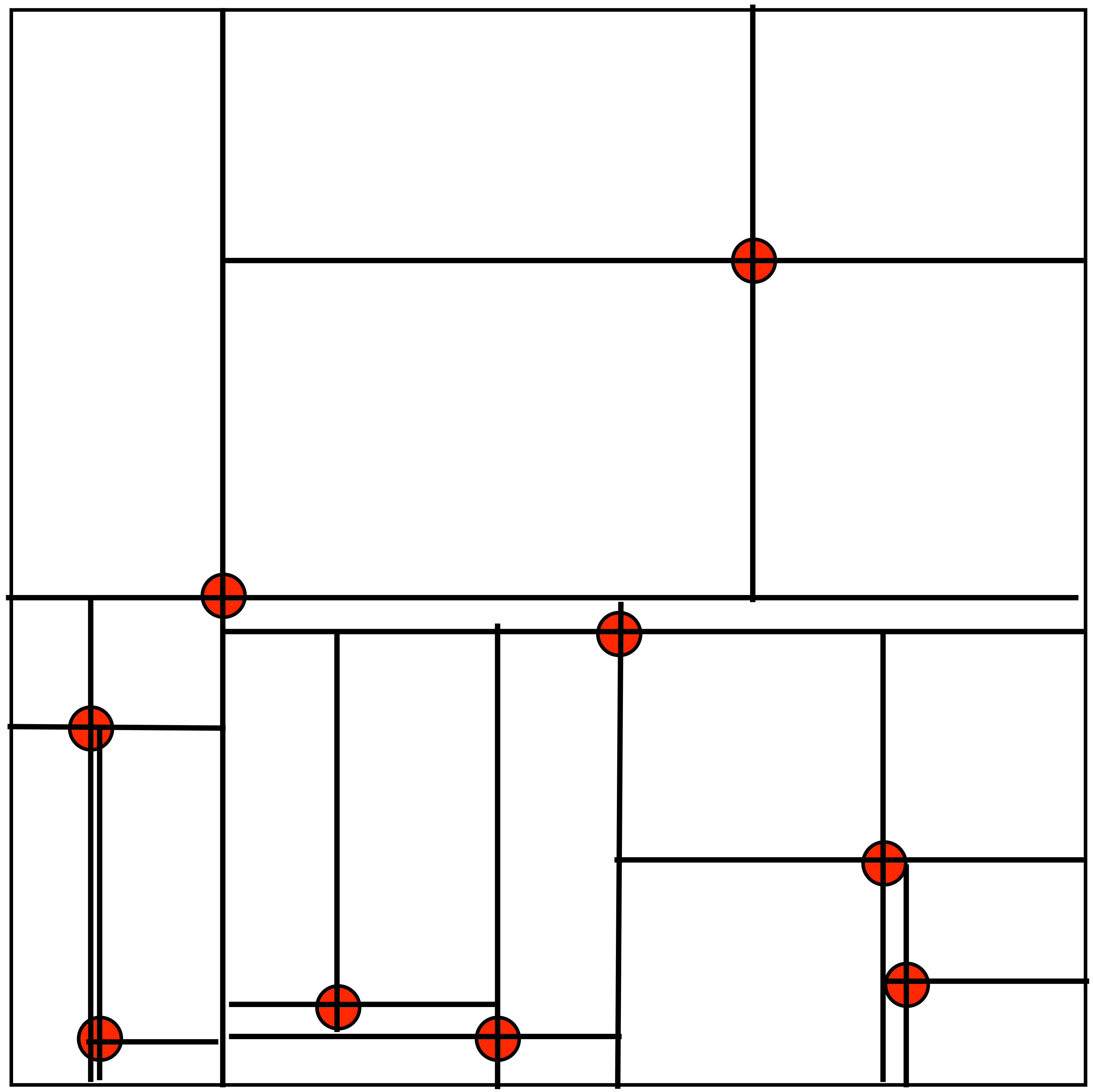
- Compression and transmission of images
  - Hierarchy => progressive transmission:
  - transmitting high levels of the tree gives you a rough image
  - lower levels gives you more detail

- Requires points come from a finite & discrete domain

Point Quadtrees

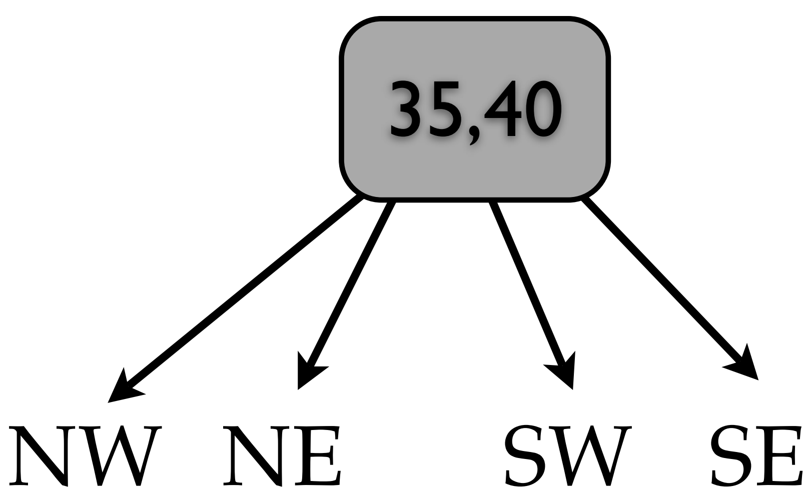
- Similar to PR Quadtrees, except we split on points in the data set, rather than evenly dividing space.
- Handling infinite space:
  - Special infinity value => allow rectangles to extend to infinity in some directions
  - Assume global bounding box

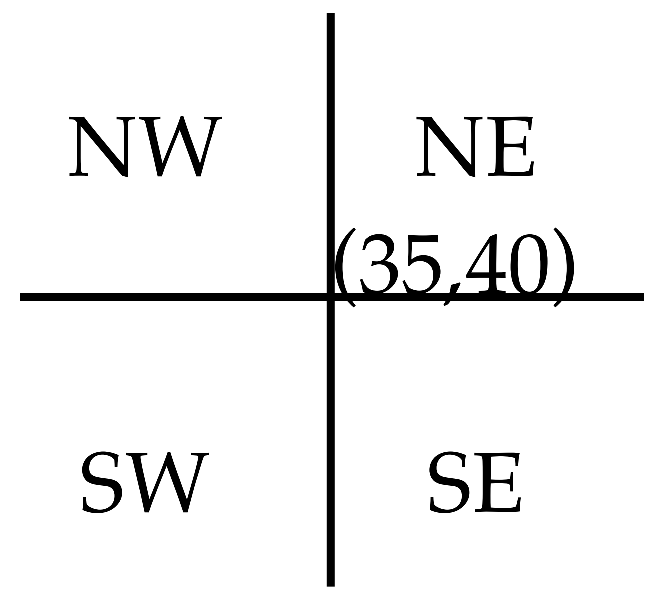
Point Quadtrees



Insertion into Point Quadtrees

- Insert(P):
  - Find the region that would contain the point P.
  - If P is encountered during the search, report **Duplicate!**
  - Add point where you fall off the tree.





Deletion from Point Quadtrees

- Reinsert all the points in the subtree rooted at the deleted node P.
- Can be expensive.
- There are some more clever ways to delete that work well under some assumptions about the data.

Some performance facts (random data):

- Cost of building a point quadtree empirically shown to be O(n log4 n) [Finkel,Bentley] with random insertions

- Expected height is O(log n).

- Expected cost of inserting the ith node into a d-dimensional quad tree is $(2/d)\ln i + O(1)$.

More balanced Point Quadtrees

- *Optimized Point Quadtree*: want no subtree rooted at node A to contain more than half the nodes (points) under A.
- Assume you know all the data at the start:
  x1 y1
  x2 y2
  x3 y3
  ...
- Sort the points lexicographically: primary key is x-coordinate, secondary key is y-coordinate.
- Make root = the median of this list (middle element)
  => half the elements will be to the left of the root, half to the right.
- Recursively apply to top and bottom halves of the list.

Pseudo Point Quadtrees

- Like PR quadtrees: splits don't occur at data points.
- Like Point Quadtrees: actual key values determine splits
- Determine a point that splits up the dataset in the most balanced way.
  - Overmars & van Leeuwen: for any N points, there is a partitioning point so that each quadrant contains ≤ ceil(N/(d+1)) points.

Comparison of Point-based & Trie-based Quadtrees

- "Trie-based" = MX and PR quadtrees
  - rely on regular space decomposion
  - data points associated only with leaf nodes
  - simple deletion
  - shape independent of insertion order
- Point-based quadtrees
  - data points in internal nodes
  - often have fewer nodes
  - harder deletion
  - shape depends on insertion order

Problems with Point Quadtrees

- May not be balanced…
  - But expected to be if points are randomly inserted.
- Size is bounded in *n*.
  - Partitioning *key space* rather than geometric space.
  - Because each node contains a point, you have at most *n* nodes.
- But may have lots of unused pointers if *d* is large!
- Solution is *kd-trees*.