Lecture 11: Spatial Partitioning, Part 2

CS 6017 – Data Analytics and Visualization

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SUMMER 2023

Lecture 12 – Topics

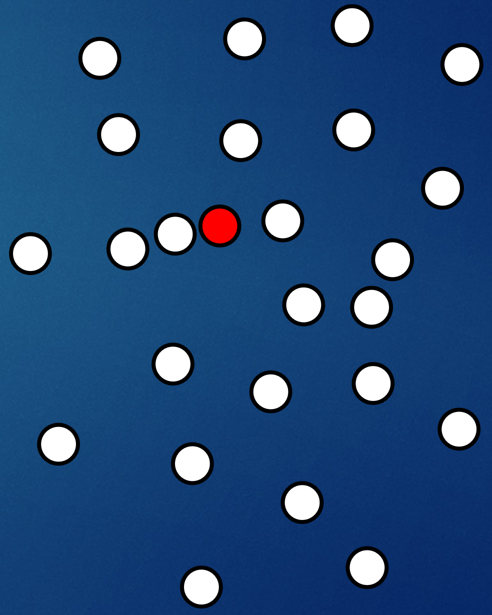
- Spatial Partitioning
  - Uniform Grid / Bucketing
  - Quad Tree
  - KD Tree (brief intro)

Miscellaneous

- Questions?
- Remember, HW3 is due next Tuesday…

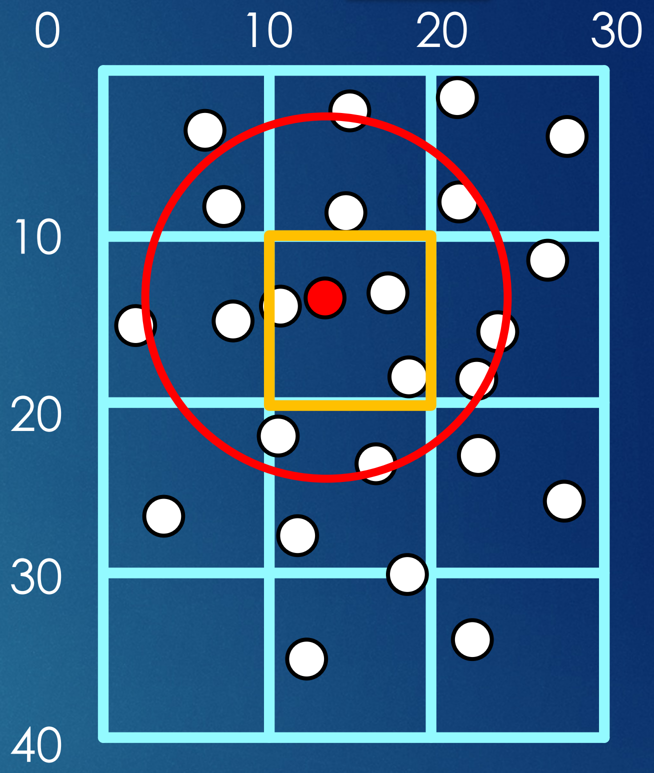
Making KNN Fast

- Naïve: Look at every point
- Today we look at better approaches…



Approach One: Bucketing

- Break space into fixed size "buckets"
- Within each bucket, we just store an array (list) of the points that are contained inside that bucket
- Which training points would we look at to find a value for the red point?
  - `KNN( red_pt, 3 )`
  - Which bucket(s) are they in?
    - [1, 1]
  - How many buckets do we look at?
    - [0, 0], [0, 1], [0, 2]
    - [1, 0], [1, 1], [1, 2]
    - [2, 0], [2, 1], [2, 2]
    - As many as it takes (moving outward) until we find K points.
  - Note: Do we only consider the points in the red circle?
    - No



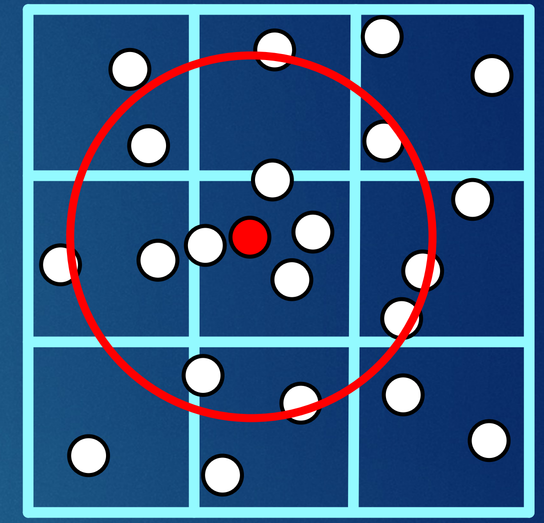
- How do we calculate these bucket indices? (Red point is: 13, 14)
  - Can be calculated in O(1)
  - How did we put data values together in histograms?
- This is a fixed size (or uniform) grid

Approach One: Bucketing (cont.)

- To do a range query, we find out which buckets (x - radius and x + radius, y – radius, y + radius) are in, then loop through all the buckets "between" them
- In a single bucket, we just check the distance from `pt` of each point and keep it if it's less than the radius
- We can quickly figure out which bucket a point is in with:

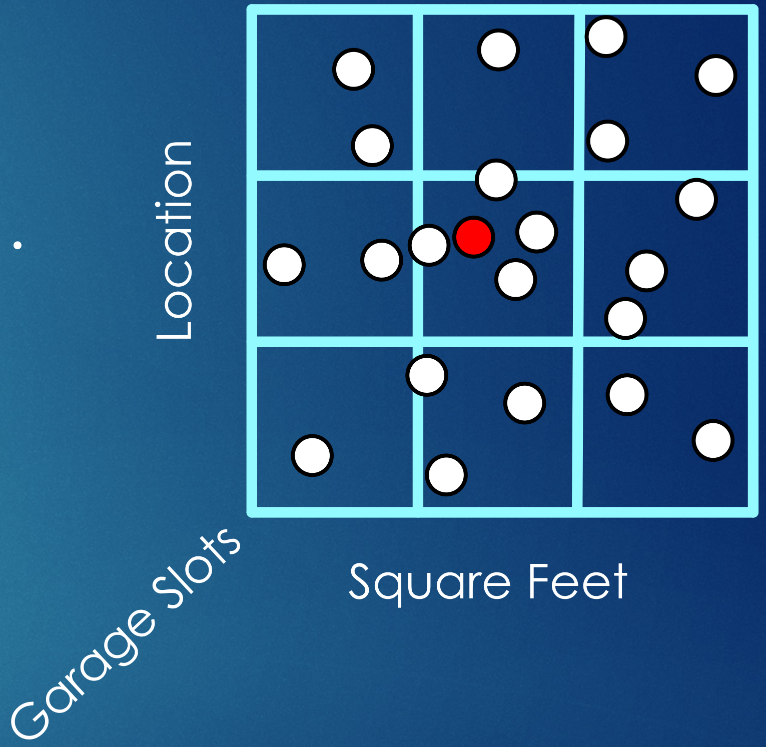
```
# for each dimension
bucketIndex = (position - bucketCorner) / bucketSize
```

- If we create N buckets along each dimension, we will have a total of $N^{dimensions}$ buckets.
    - Note: this can be bad for high dimension data!
- To do a KNN query, we do a range query and increase the radius if we don't find enough points
- Note: *Training* is just creating this data structure and storing the appropriate points in the appropriate bucket.



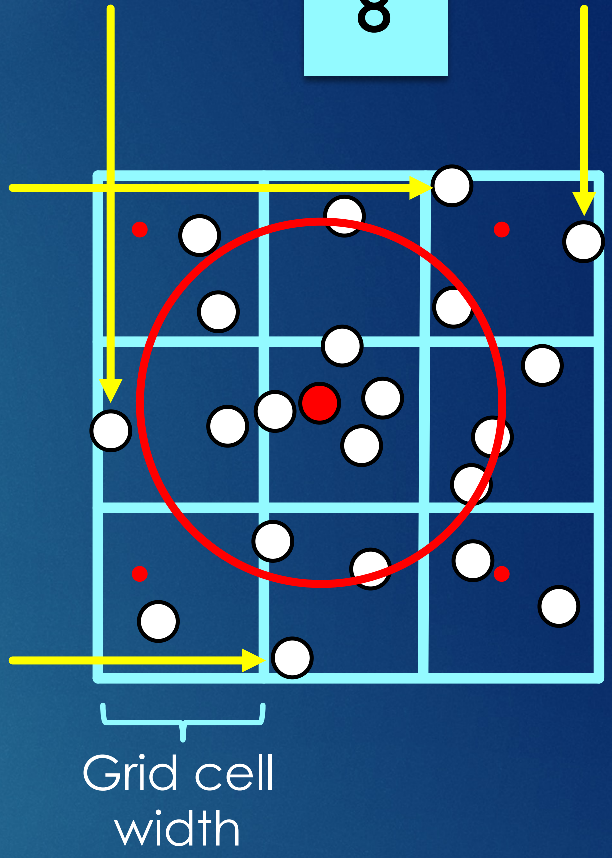
Approach One: Bucketing (cont.)

- What is a *dimension*? Note: I'm currently displaying 2 dimensions…
- Let's think about this in terms of house sell price… What might the axes (ie, the dimension) be for each axis? x? y?…
  - X axis: # of square feet
  - Y axis: Location
  - Z axis: # of Garage Spaces
- More axes… and I've run out of letters?!?
  - X_3: # of Bathrooms
  - X_4: Lot Size
  - X_5: Age of roof
  - …



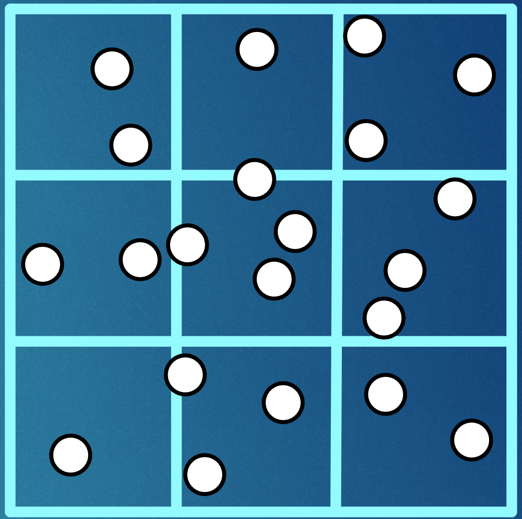
Approach One: Bucketing (math)

- How to determine the x and y values of upper left and lower right points on the grid itself?
  - min(pts.x), max(pts.x)
  - min(pts.y), max(pts.y)
- Bucket index math?
  - col = (p.x – grid_min.x) / cell_width
  - row = (p.y – grid_min.y) / cell_height
- Which buckets are "near" the red point?
  - Determine the "square" around point of interest
  - p.x +/- radius, p.y +/- radius



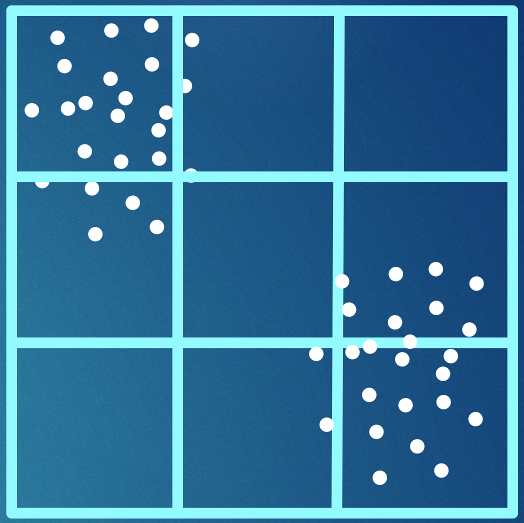
Uniform Grid / Bucket Approach

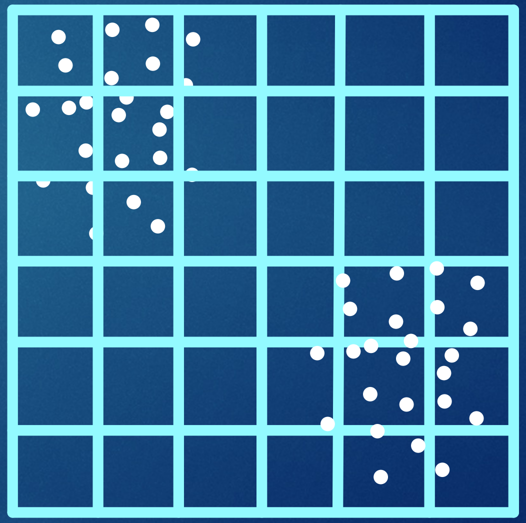
- Issues
  - 1 (giant) bucket vs 1000s of small buckets?
- Trickiness
  - Choosing the # of buckets…
    - S splits (3 in this example)
    - S^{dim} buckets…
    - As *dim* gets big?
      - "Curse of Dimensionality"
  - Storing the array of buckets…
    - Could "flatten" the array
    - Need a way to covert "row 1, column 2" into an index in a 1D array
      - Note: most languages support 1 and 2 dimensional arrays… but we may have many dimensions…
    - Could use a hash function for this. If a lot of buckets are empty, this works very well! (Spatial Hashing – Row,Col as key to hash table)
  - In 2D you could say `index = row*numColumns + column`.
    - Can produce a similar formula for any number of dimensions



Uniform Grid Issues

- What are some issues that a uniform grid of buckets might have?
  - Non-uniform data…
  - Perhaps make more buckets?
  - Empty buckets cause overhead
  - Still issues, so…
- Uniform grid works well if data points are relatively evenly distributed…
  - Note: Non-uniform (stretched) grid is a possibility – though finding buckets can be more difficult.



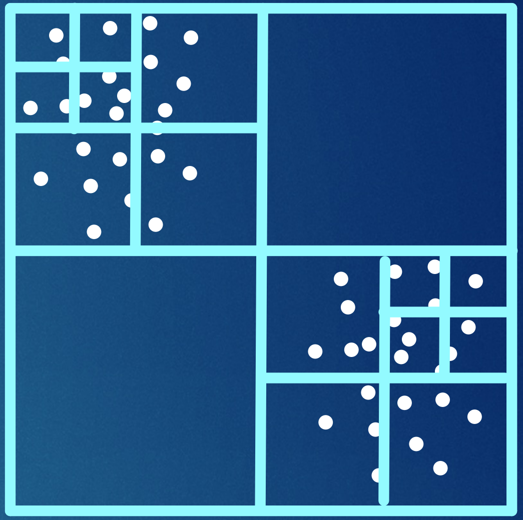


Spatial Partitioning

- These approaches we are discussing are in a family of algorithms for spatial partitioning
  - Breaking up "space" into regions
- Uniform grid works well on uniformly distributed data…
  - For Non-uniform data –> Want non-uniform buckets
- Need a "data-aware" structure
  - Will look at two different *tree-like* data structures which are designed to split up the space where there are more points.
  - Building these data structures is more complicated, but they are better adapted for a lot of real data sets.

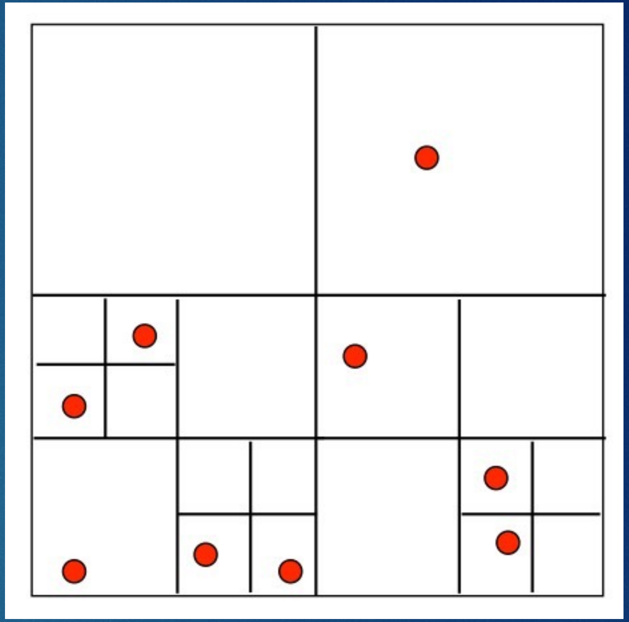
Approach Two: Quad Tree

- Create a grid
  - *Adaptively* add more cells based on…
    - distribution of data
  - Fix number of splits to 1…
    - Thus always dealing with a 2x2 grid.
  - How do we reduce the number of points in a grid cell?
    - Split cells with too many points again
    - Repeat recursively…



Quad Tree

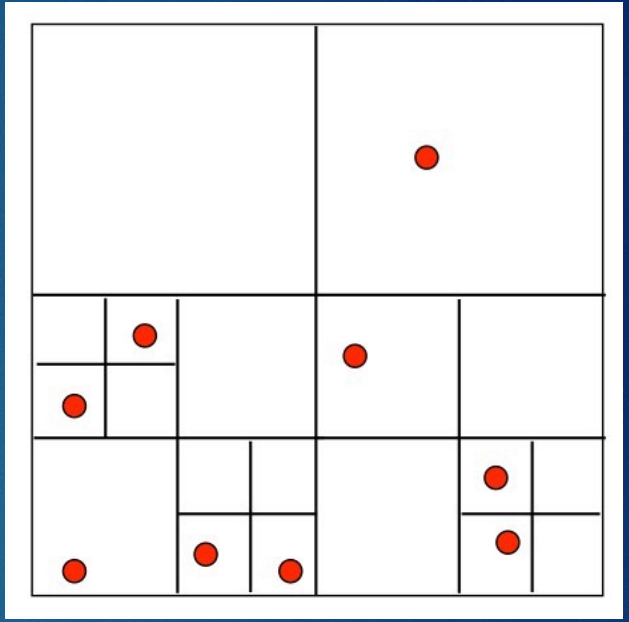
- A node is either a "bucket" (array of points)
  - Or an internal node with 4 children (each of the children covers 25% of parent's area)
- We pick a "threshold" number of points to decide whether to split a node or make it a bucket (leaf)
- Each node stores the region it covers (an "axis aligned bounding box" (AABB)) and either a list of points, or pointers to its 4 child nodes
  - Children are sometimes named NW, NE, SW, SE
- Quad Tree because each sub-cell is broken into 4 children… What is the 3D version of this called?
  - Oct Tree
  - Note: # of children per node is $2^{dimensions}$.
  - Thus usually these are only used up to 3 dimensions.



Example where each node is only allowed to store one point (before being sub-divided.

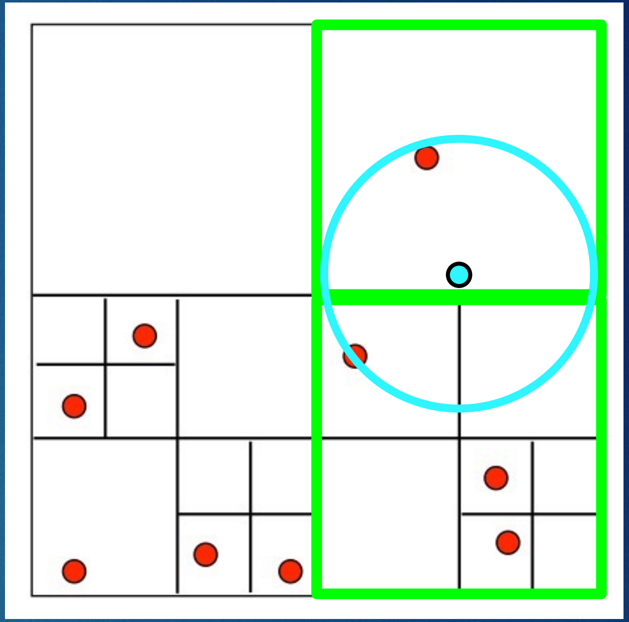
Quad Tree Construction

- Input – "Point Soup"
  - Points are not organized in any way
- Since a Quad Tree is a Tree:
  - `root = new Node( points, aabb )`
    - aabb specifies the area the node is responsible for
- Recursive approach…
- So what is the base case?
  - # of points is less than X, then this is a leaf node (and we are done).
    - Note: X is a *knob* that is used to *tune* the data structure
- Otherwise, create four children…
  - How do we decide which points to send to each child?
    - Find midpoints: (aabb.max + min) / 2
    - Loop through the points and place them in the NW, NE, SW, SE list of points
      - "Array Partitioning"
    - `NW = new Node( NW_list, nw_aabb )`



Quad Tree Querying

- Function - `QT::rangeQuery( pt, radius )`
- Recursive algorithm, so:
  - `points = Node::rangeQuery( pt, radius )`
- `points = root.rangeQuery( my_pt, r )`
- Only need to "think about" handing this for a single node, and let recursion take care of the rest.
- Two choices:
  - Leaf node:
    - Look through the array of points this leaf contains.
      - Is point within `r` of `my_pt`?
  - otherwise?
    - determine which child/children `my_pt+radius` is/are in, and call (for example):
    - `ne_points = NE.rangeQuery( my_pt, radius )`
    - Note: Depending on radius, we may have to call the `rangeQuery` on multiple children. (Bounding box checks with AABB)



Quad Tree KNN Query

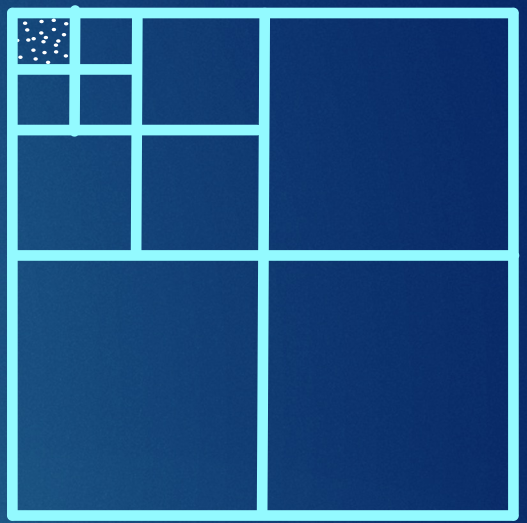
- `Node::knnQuery( pt, K, result )`
  - Note: `result` keeps track of the list of the nearest K points
- Similar to range query, but if statements are more complicated…

```
if leaf
    for each point in bucket…
        if len( result ) < K
            add point
        else if distance( point, pt ) < distance( pt, worst in result )
            replace worst with point
else // internal node
    for each child
        if necessary // <– return length < K or closestPointInAABB( pt ) is closer than worst in list
            recurse
```

- Can think of this as having a "search radius"
  - The search radius starts at infinity, but shrinks as we add points to the `result` list.

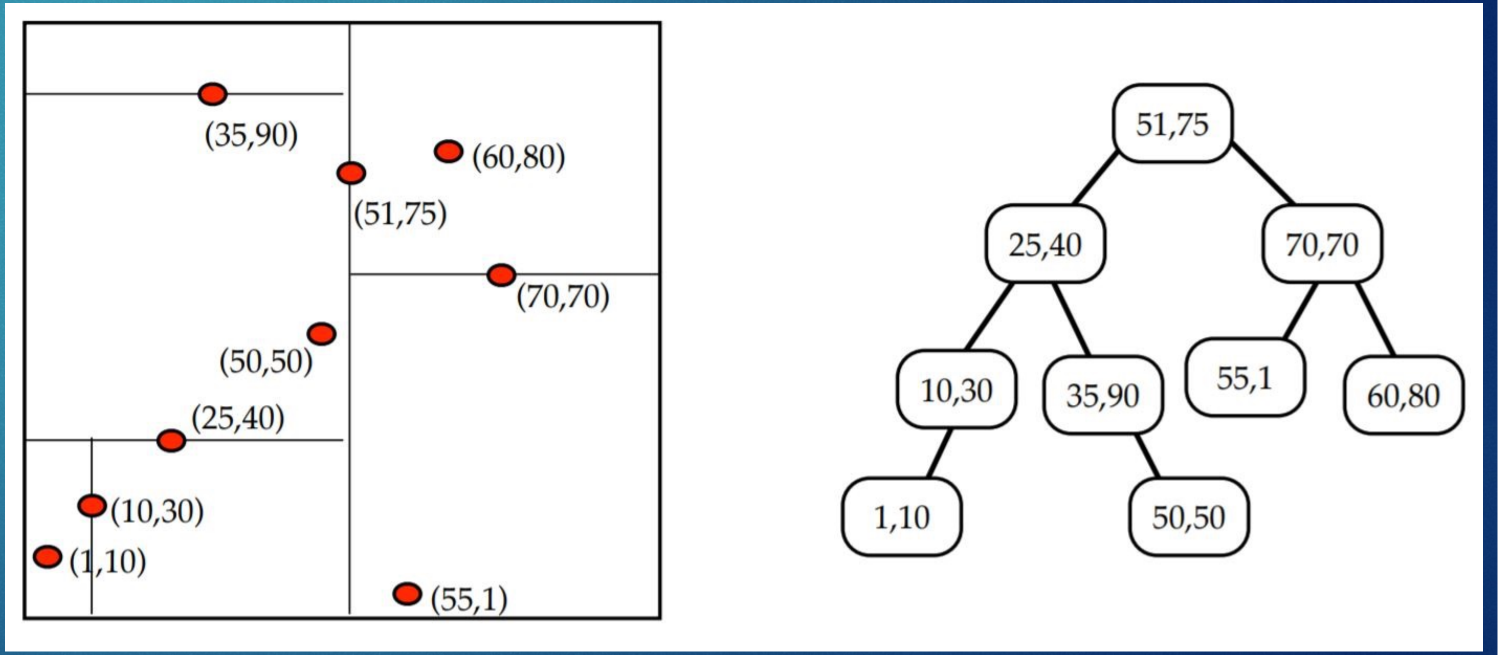
Quad Tree Query Complexity

- Asymptotic Complexity (Big O)?
- Height of tree? (Number of levels down?)
  - $\sim\log_4 N$
- Query time: $\sim\log_4 N \lozenge K \lozenge r$
- If tree is not balanced… And depending on K and radius the number of cells we are looking at can change (think the "width" of the search)…
- How could we split better?
  - See next slide…



Approach Three: KD Tree

- Generalization of a BST to many "keys"
- Each node stores the median of it's subtree according to one of the dimensions (x, y, z, etc)
- Each time we go down one level, we move to the next dimension (if I split by x, my children split by y)
- Each node stores a point and the dimension it splits by
- More tomorrow



~ *Fin* ~