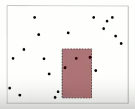
Kd-trees

Extension of ordered symbol table to 2d keys (points in 2d space):

* Insert a 2d key
* ~~Delete a 2d key~~
* Search for a 2d key
* Range search: find all keys that lie in a 2d range
* Range count: number of keys that lie in a 2d range

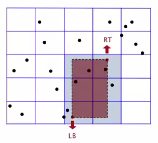
Geometric interpretation:

* Keys are a point in the plane
* Find/count points in a given h-v rectangle



* Applications include: networking, circuit design, databases

Grid implementation

1. Divide space into an m-by-m grid of squares  
   . 
2. Create a list of points contained in each square
3. User 2d array to directly index relevant square
4. Insert: add (x, y) to list for corresponding square
5. Range search: examine squares that intersect 2d range query

Trade off:

* Space: M2 + N
* Time: 1 + N / M2 per square examined on avg

Choose grid square size to tune performance:

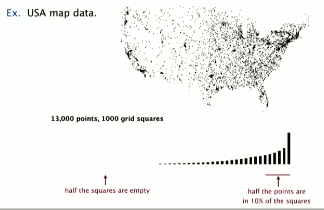
* Too small: wastes space
* Too large: too many points per square
* Rule of thumb: √N-by-√N grid

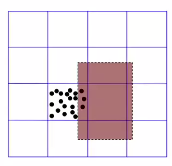
Running time:

* Initiaalize: N
* Insert: 1
* Range search: 1/ point in range

Grid implementation is good for evenly distributed points, BUT

CLUSTERING is a well-known phenomenon that presents challenges in geometric data





* Lists will be too long even though avg length is short
* Need data structure that gracefully adapts to the distribution of the data

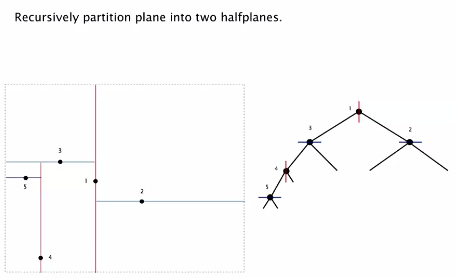
Space-partitioning trees

Use a tree to represent a recursive subdivision of the plane of 2d space

Applications:

* Flight simulators are only made possible through space partitioning trees
* This is also extremely important for scientific data processing

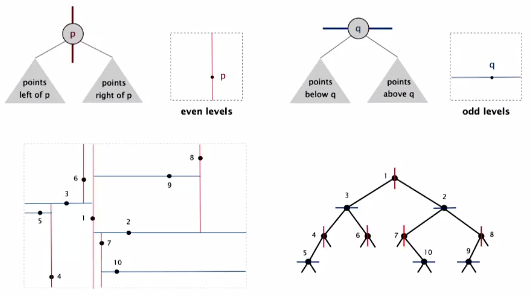
Visualization



Implementation further visualized:

Same as a BST, but we alternate using x and y coordinates as keys   
(even levels vertical, odd levels horizontal)

* Search gives rectangle containing point
* Insert further subdivides in the plane

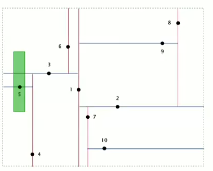


Solving range search in a 2d tree

Check if point in node lies in a given rectangle

Recursively search left/bottom (if any could fall in rectangle)

Recursively search right/top (if any could fall in rectangle)



If splitting line hits a rectangle, still must check both branches, but this is still a very efficient algorithm

**Run time analysis of operations:**

* Typical case: R + log N
* Worst case (assuming a balanced tree): R + √N

*E.g. points are arranged in a circle*

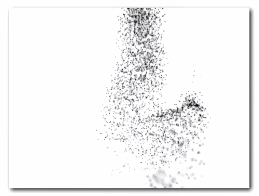
**Nearest neighbor search**

* Check distance from point in node to query point
* Recursively search left/bottom (if it could contain a closer point)
* Recursively search right/top (if it could contain a closer point)
* Organize method so it begins by searching for query point

**Running time**: typical running time is log N, but worst case is linear

Boids (to simulate flocking behavior of flying birds)

1. Collision avoidance-> point away from *k* nearest boids
2. Flock centering-> point towards the center of mass of *k* nearest boids
3. Velocity matching-> update velocity to the average of *k* nearest boids

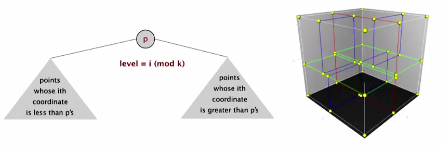


2d trees extremely effective in quickly processing huge amounts of geometric data

Simple modification can add additional dimensions, this is called a:

**Kd tree**

Recursively partition *k*-dimensional space into 2 halfspaces



Implementation: BST, but cycle through dimensions ala 2d trees

At level *I* we put on the left coordinates less than p and on the right coordinates are greater than p (cycle through the dimensions)

Implementation is simple (except for the comparison)

We get the same kind of partitioning for 3d data- so we can do boids in 3d

In databases with more dimensions, we can do boids with many more dimensions

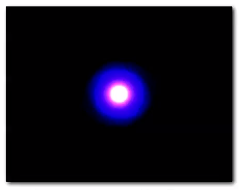
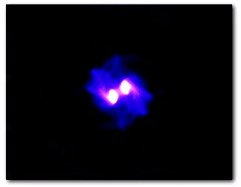
Efficient and simple data structure for processing *k*-dimensional data

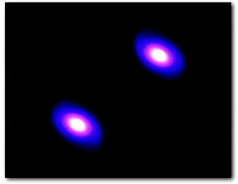
* Widely used
* Adapts well to high-dimensional data and clustered data
* Discovered by an undergrad in an algorithms class (Jon Bentley @ Stanford)

N-body simulation

Simulate the motion of *N* particles mutually effected by gravity

Brute force: For each pair of particles, compute force F = (G m1 m2) / r2





For each pair of particles… N2 QUADRATIC time

Quadratic calculation won’t work with quadratic

Appel algorithm is a more efficient algorithm (Linearithmic):

* Treat cluster of particles as a single aggregate particle
* Compute force between particle and center of mass of aggregate

How?

* Use 3d tree with *N* particles as nodes
* Store center-of-mass of subtree in each node
* To compute total force acting on a particle, traverse tree, but stop as soon as distance from particle to subdivision is sufficiently large