Geometric Applications of BSTs

1) 1D range search

Geometric Objects

- extension of the ordered symbol table
  - insert key-value pair
  - search for key k
  - delete key k
  - range search: find all keys between k_1 and k_2
  - range count: no. of keys between k_1 and k_2
  - Application: database queries
- Geometric interpretation
  - keys are a point on line
  - find / count points in a given 1d interval

Implementations

O(1) O(N)

- unordered array - fast insert, slow range search
- ordered array - slow insert, binary search for k_1 and k_2 to do range search

- size = rank(high) − rank(low) + 1 if high

range count

range search

- recursively find all keys in left subtree (if any could fall in range)
- check key in current node
- recursively find all keys in right subtree

II) Line segment intersection

- given N horizontal and vertical line segments, find all intersections

* quadratic algorithm => check all pairs of line segments for intersection

* all (x-) and (y-) coordinates are distinct

Sweep-line algorithm

- Sweep vertical line from left to right
  - x-coordinates define events
  - h-segment (left endpoint): insert y-coordinate into BST
  - horizontal segment (right endpoint): remove y-coordinate from BST
  - vertical segment: range search for interval of y-endpoints

$O(N \log N + R)$ to find all R intersections

III) kd-Trees

- extension of ordered symbol-table to 2d keys
  - insert a 2D key / delete
  - search for a 2D key
  - range search: find all keys that lie in a 2D range
  - range count: no. of keys that lie in a 2D range

- keys are points in a plane
- find/count points in a given h-v rectangle

* divide space into M by M grid of squares
  - add points to a square

M x M => M too large => waste of space

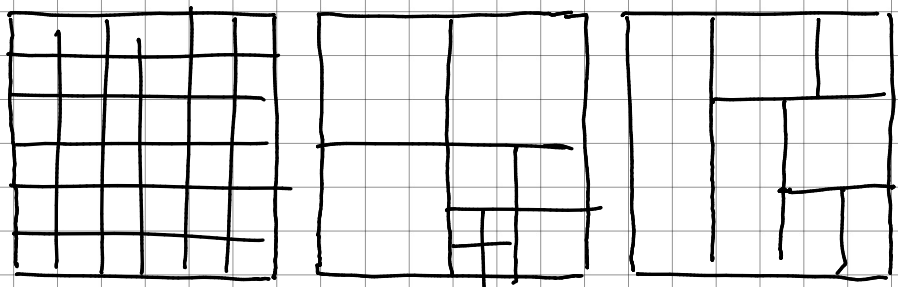
$O(M^2 + N)$ - space

=> M too small => too many points per square => waste of time

$O(1 + N/M^2)$ - time

$\Rightarrow \boxed{M = \sqrt{N}} \quad \Rightarrow \boxed{\text{problem} = \text{clustering}}$

* Use a tree to represent a recursive sub-division of a plane



grid

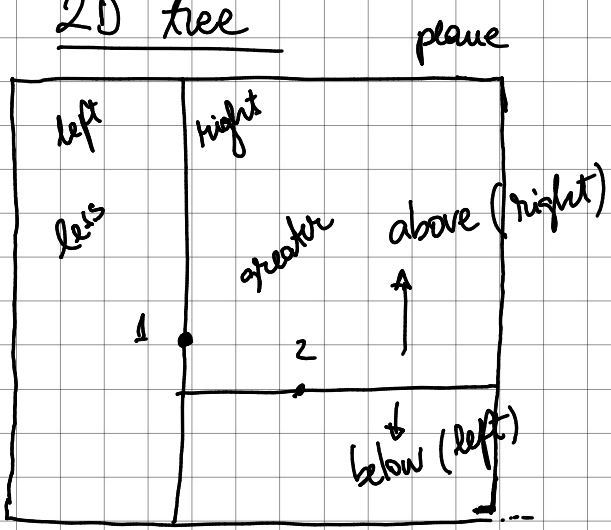
Quad-tree

rec. divide space into 4 quadrants

2d tree

rec. div. space into 2 half planes

2D tree



tree level – start with vertical ⇒ switch at every level

* Data structure ⇒ BST but alternate using x- and y- coordinates as keys

- Range search
  - recursively search
    - left-bottom (if any could fall in rectangle)
    - right-top

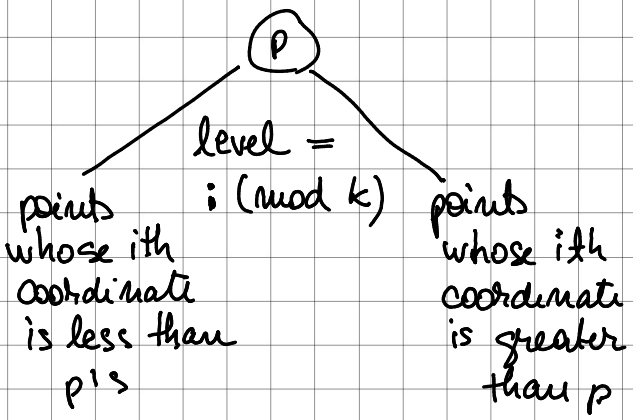
– typical $R + \log N$, R = no. of points in range

worst $R + \sqrt{N}$

- find closest point to query point
  - compute distance from query to current point (save as min)

k-d tree

- recursively partition k-dimensional space into 2 halfspaces
- implementation: BST, but cycle through dimensions ala 2D trees



IV) Interval Search Trees

1-D interval search = data structure to hold a set of (overlapping) intervals

- insert an interval (low, high)
- search
- delete

o interval intersection query: given an interval (low, high) find all intervals in data structure that intersects (low, high)

class IntervalSearchTree

put(key low, key high, value)
get(key low, key high)
delete(key low, key high)
intersects(key low, key high)

* no two intervals have the same left endpoint
* use left endpoint as BST key
* store max endpoint in subtree rooted at node

Search: 1) if interval in node intersects query interval return it

- else if left subtree is null, go right
- else if max endpoint in left subtree is less than low ⇒ go right
- else go left

⇒ use Red-Black BST for performance

V) Rectangle Intersection

- find all intersections among a set of orthogonal rectangles
- all coordinates are distinct
  - sweep vertical line from left to right
  - x-coordinates of left and right endpoints define events
  - maintain set of rectangles that intersect the sweep line in an interval search tree (using y intervals of rectangle)
  - left endpoint: interval search for y interval of rectangle

insert y - interval

- right endpoint - remove y interval