K-D Trees and Quad Trees

CSE 326
Data Structures
Lecture 9

Reading

- Chapter 12.6

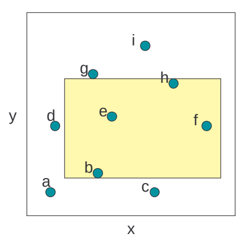
Geometric Data Structures

- Organization of points, lines, planes, … to support faster processing
- Applications
  - Astrophysical simulation – evolution of galaxies
  - Graphics – computing object intersections
  - Data compression
    - Points are representatives of 2x2 blocks in an image
    - Nearest neighbor search

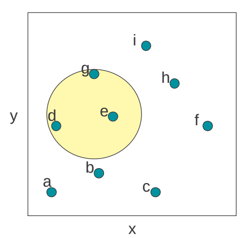
k-d Trees

- Jon Bentley, 1975, while an undergraduate
- Tree used to store spatial data.
  - Nearest neighbor search.
  - Range queries.
  - Fast look-up
- k-d tree are guaranteed $\log_2 n$ depth where n is the number of points in the set.
  - Traditionally, k-d trees store points in d-dimensional space which are equivalent to vectors in d-dimensional space.

Range Queries

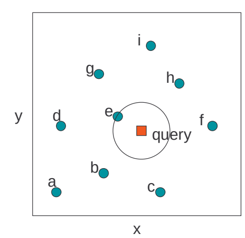


Rectangular query



Circular query

Nearest Neighbor Search

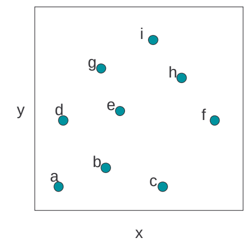


Nearest neighbor is e.

k-d Tree Construction

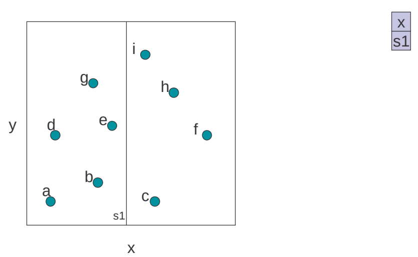
- If there is just one point, form a leaf with that point.
- Otherwise, divide the points in half by a line perpendicular to one of the axes.
- Recursively construct k-d trees for the two sets of points.
- Division strategies
  - divide points perpendicular to the axis with widest spread.
  - divide in a round-robin fashion (book does it this way)

k-d Tree Construction (1)

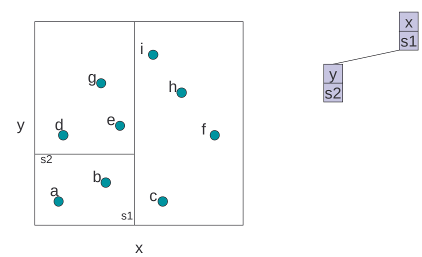


divide perpendicular to the widest spread.

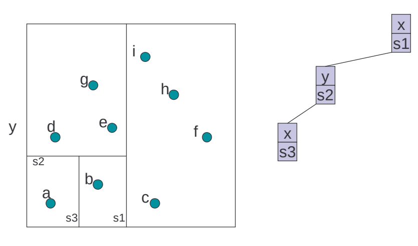
k-d Tree Construction (2)



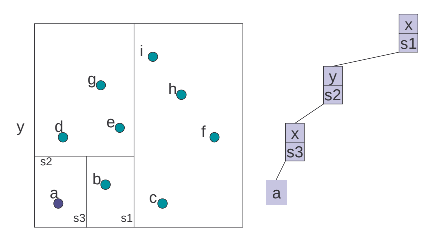
k-d Tree Construction (3)



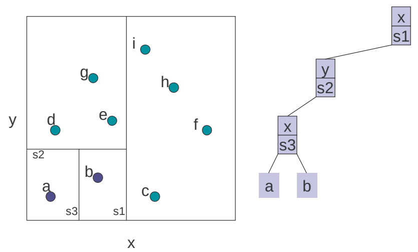
k-d Tree Construction (4)



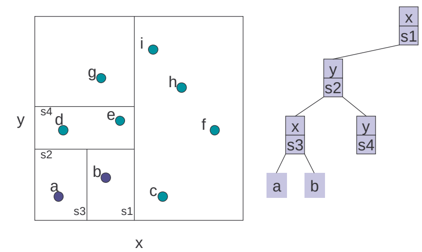
k-d Tree Construction (5)



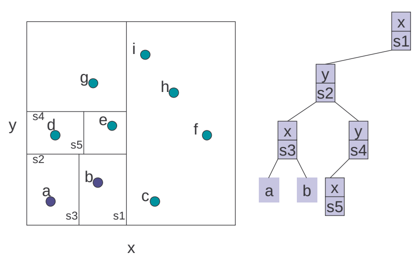
k-d Tree Construction (6)



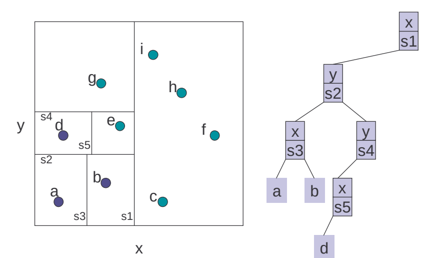
k-d Tree Construction (7)



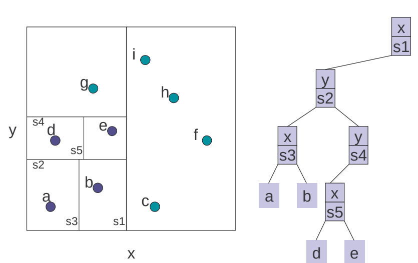
k-d Tree Construction (8)



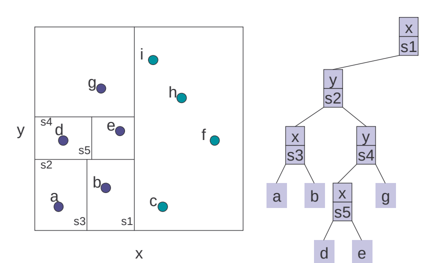
k-d Tree Construction (9)



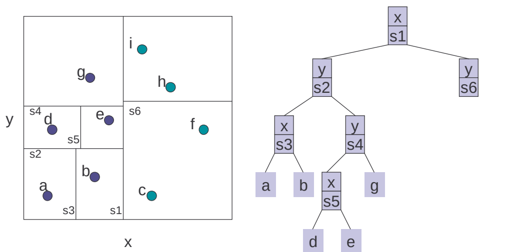
k-d Tree Construction (10)



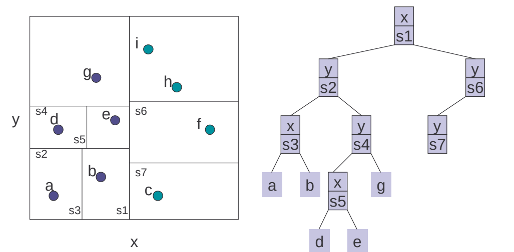
k-d Tree Construction (11)



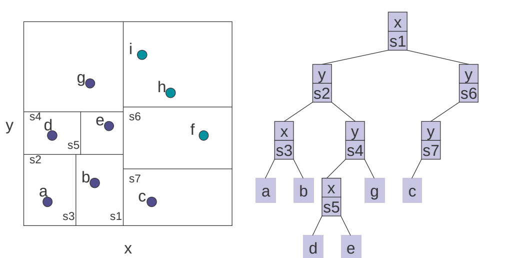
k-d Tree Construction (12)



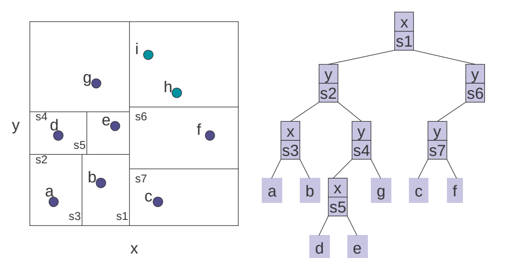
k-d Tree Construction (13)



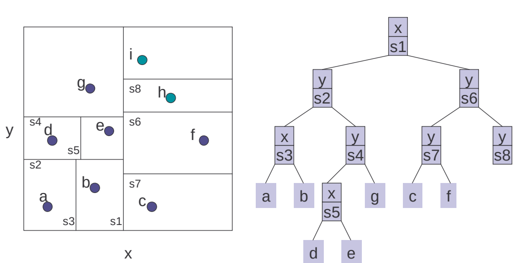
k-d Tree Construction (14)



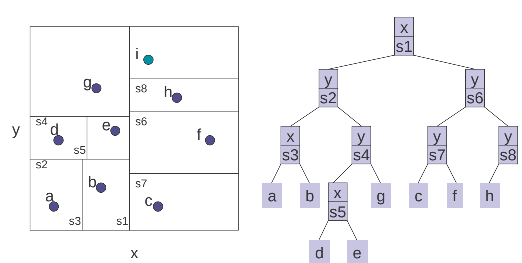
k-d Tree Construction (15)



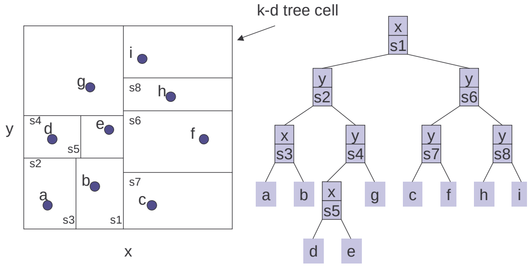
k-d Tree Construction (16)



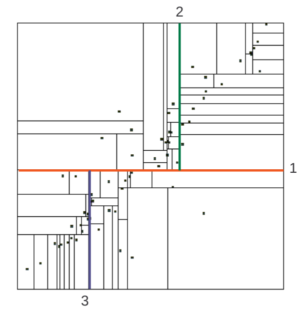
k-d Tree Construction (17)



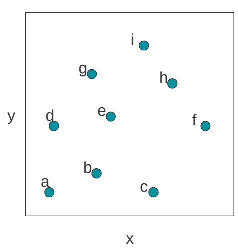
k-d Tree Construction (18)



2-d Tree Decomposition



k-d Tree Splitting

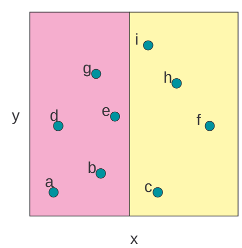


sorted points in each dimension

	1	2	3	4	5	6	7	8	9
x	a	d	g	b	e	i	c	h	f
y	a	c	b	d	f	e	h	g	i

- max spread is the max of $f_x - a_x$ and $i_y - a_y$.
- In the selected dimension the middle point in the list splits the data.
- To build the sorted lists for the other dimensions scan the sorted list adding each point to one of two sorted lists.

k-d Tree Splitting



sorted points in each dimension

	1	2	3	4	5	6	7	8	9
x	a	d	g	b	e	i	c	h	f
y	a	c	b	d	f	e	h	g	i

indicator for each set

a	b	c	d	e	f	g	h	i
0	0	1	0	0	1	0	1	1

scan sorted points in y dimension and add to correct set

y	a	b	d	e	g	c	f	h	i

k-d Tree Construction Complexity

- First sort the points in each dimension.
  - O(dn log n) time and dn storage.
  - These are stored in A[1..d,1..n]
- Finding the widest spread and equally divide into two subsets can be done in O(dn) time.
- We have the recurrence
  - $T(n,d) \leq 2T(n/2,d) + O(dn)$
- Constructing the k-d tree can be done in O(dn log n) and dn storage

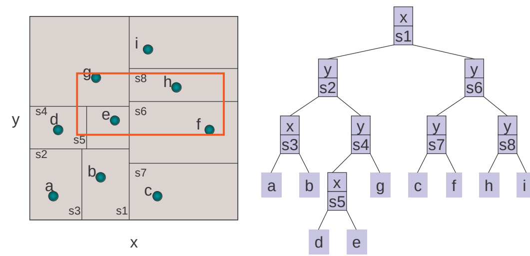
Node Structure for k-d Trees

- A node has 5 fields
  - axis (splitting axis)
  - value (splitting value)
  - left (left subtree)
  - right (right subtree)
  - point (holds a point if left and right children are null)

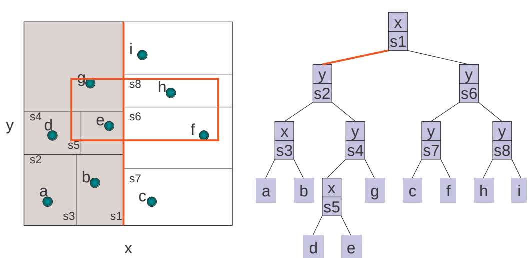
Rectangular Range Query

- Recursively search every cell that intersects the rectangle.

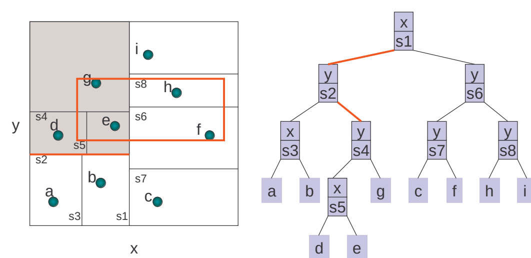
Rectangular Range Query (1)



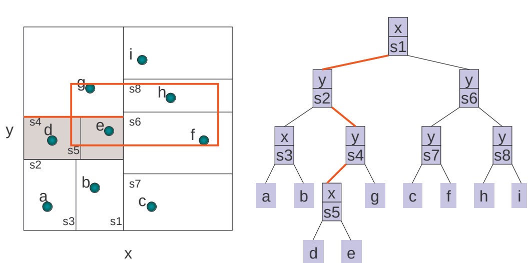
Rectangular Range Query (2)



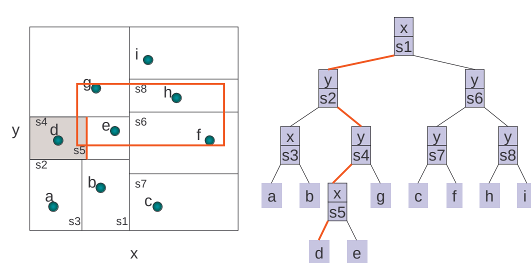
Rectangular Range Query (3)



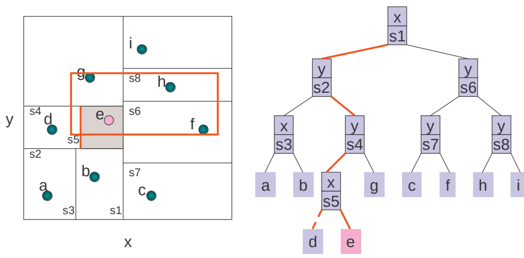
Rectangular Range Query (4)



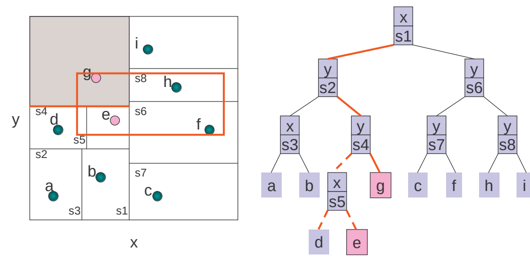
Rectangular Range Query (5)



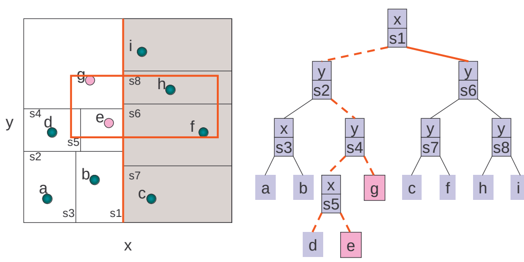
Rectangular Range Query (6)



Rectangular Range Query (7)



Rectangular Range Query (8)

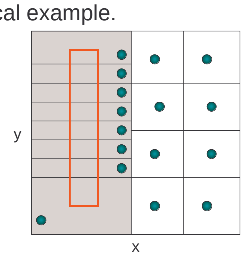


Rectangular Range Query

```
print_range(xlow, xhigh, ylow, yhigh :integer, root: node pointer) {
   Case {
    root = null: return;
    root.left = null:
        if xlow ≤ root.point.x and root.point.x ≤ xhigh
        and ylow ≤ root.point.y and root.point.y ≤ yhigh
          then print(root);
    else
        if(root.axis = “x” and xlow ≤ root.value ) or
        (root.axis = “y” and ylow ≤ root.value ) then
           print_range(xlow, xhigh, ylow, yhigh, root.left);
        if (root.axis = “x” and xlow > root.value ) or
        (root.axis = “y” and ylow > root.value ) then
           print_range(xlow, xhigh, ylow, yhigh, root.right);
}}
```


Analysis of Rectangular Range Query

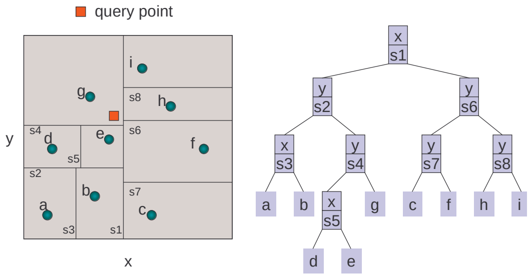
- Worst case time is O(n) as seen by the pathological example.



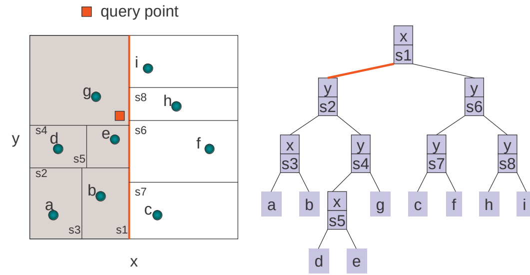
k-d Tree Nearest Neighbor Search

- Search recursively to find the point in the same cell as the query.
- On the return search each subtree where a closer point than the one you already know about might be found.

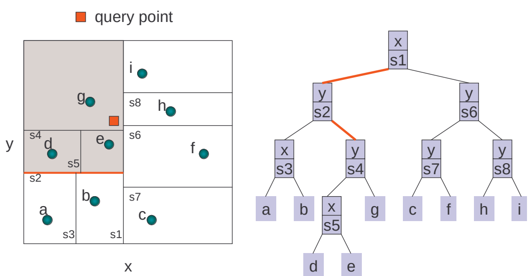
k-d Tree NNS (1)



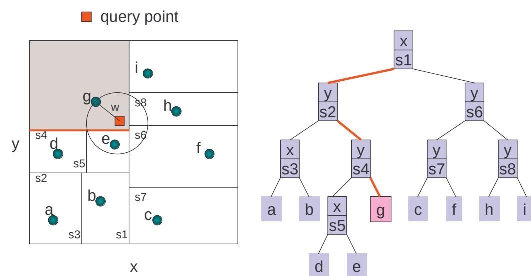
k-d Tree NNS (2)



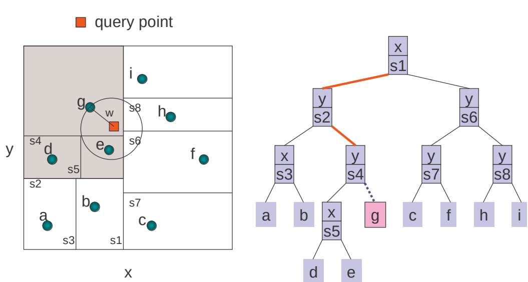
k-d Tree NNS (3)



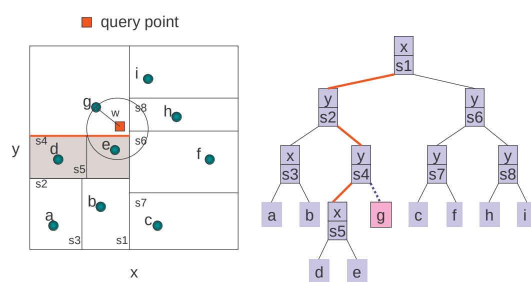
k-d Tree NNS (4)



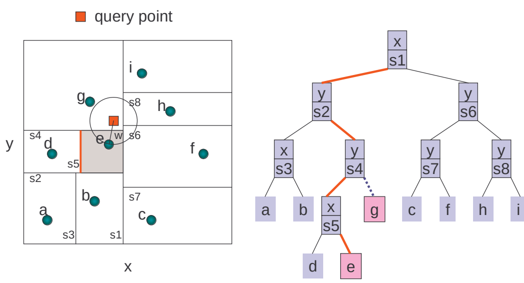
k-d Tree NNS (5)



k-d Tree NNS (6)

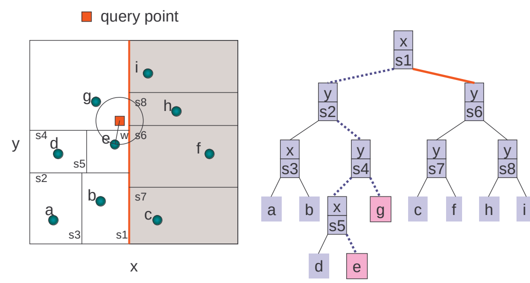


k-d Tree NNS (7)



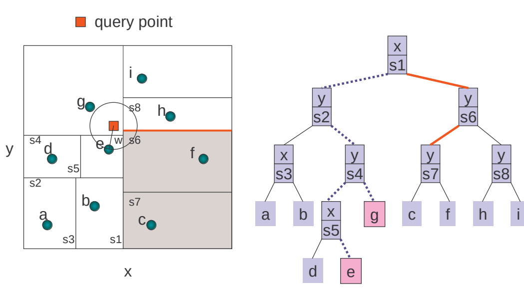
K-D Trees and Quad Trees - Lecture 9 49

k-d Tree NNS (10)



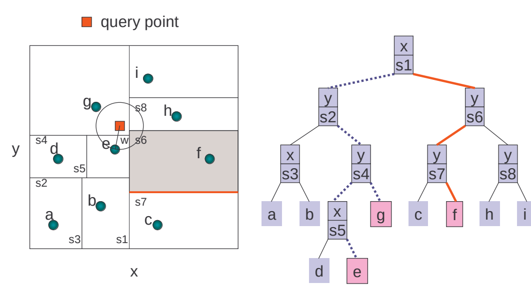
K-D Trees and Quad Trees - Lecture 9 50

k-d Tree NNS (11)



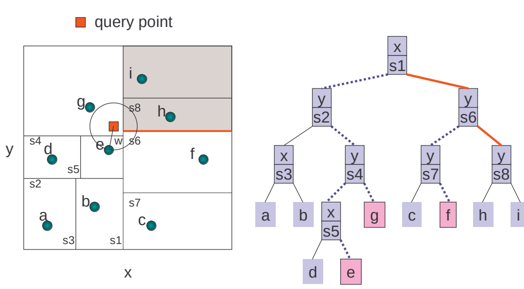
K-D Trees and Quad Trees - Lecture 9 51

k-d Tree NNS (12)



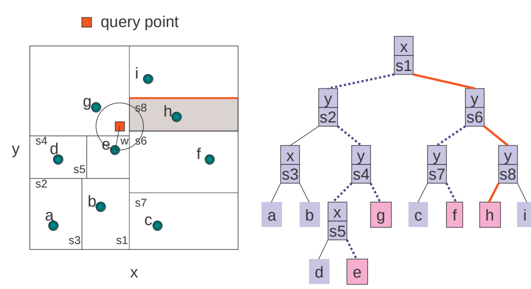
K-D Trees and Quad Trees - Lecture 9 52

k-d Tree NNS (13)



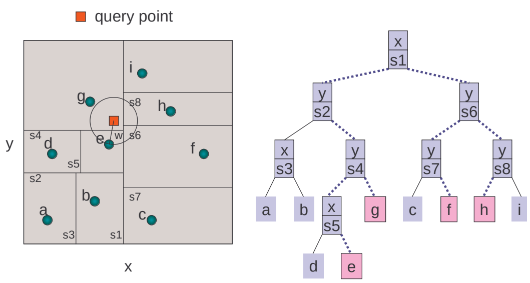
K-D Trees and Quad Trees - Lecture 9 53

k-d Tree NNS (14)



K-D Trees and Quad Trees - Lecture 9 54

k-d Tree NNS (15)



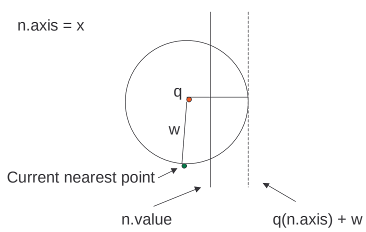
Main is NNS(q,root,null,infinity)

Nearest Neighbor Search

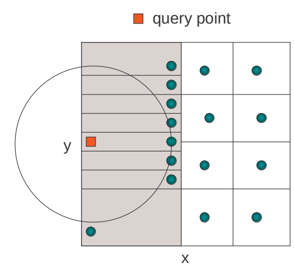
```
NNS(q: point, n: node, p: point, w: distance) : point {
if n.left = null then {leaf case}
   if distance(q,n.point) < w then return n.point else return p;
else
   if w = infinity then
     if q(n.axis) ≤ n.value then
        p := NNS(q,n.left,p,w);
        w := distance(p,q);
        if q(n.axis) + w > n.value then p := NNS(q, n.right, p, w);
     else
        p := NNS(q,n.right,p,w);
        w := distance(p,q);
        if q(n.axis) - w ≤ n.value then p := NNS(q, n.left, p, w);
   else //w is finite//
      if q(n.axis) - w ≤ n.value then
        p := NNS(q, n.left, p, w);
        w := distance(p,q);
      if q(n.axis) + w > n.value then p := NNS(q, n.right, p, w);
   return p
}
```


The Conditional

`q(n.axis) + w > n.value`



Worst-Case for Nearest Neighbor Search



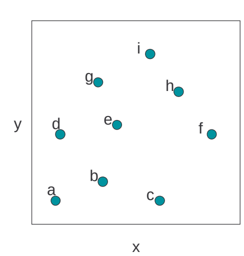
- Half of the points visited for a query
- Worst case O(n)
- But: on average (and in practice) nearest neighbor queries are O(log N)

Notes on k-d NNS

- Has been shown to run in O(log n) average time per search in a reasonable model. (Assume d a constant)
- Storage for the k-d tree is O(n).
- Preprocessing time is O(n log n) assuming d is a constant.

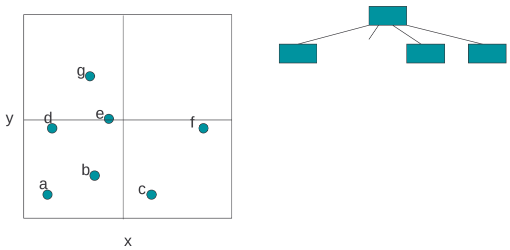
Quad Trees

- Space Partitioning



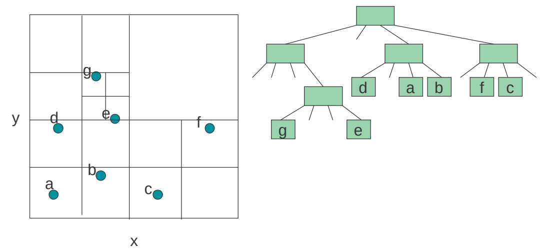
Quad Trees

- Space Partitioning

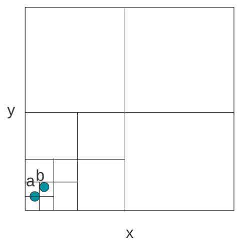


Quad Trees

- Space Partitioning



A Bad Case



Notes on Quad Trees

- Number of nodes is $O(n(1+ \log(\Delta/n)))$ where n is the number of points and Δ is the ratio of the width (or height) of the key space and the smallest distance between two points
- Height of the tree is $O(\log n + \log \Delta)$

K-D vs Quad

- k-D Trees
  - Density balanced trees
  - Height of the tree is O(log n) with batch insertion
  - Good choice for high dimension
  - Supports insert, find, nearest neighbor, range queries
- Quad Trees
  - Space partitioning tree
  - May not be balanced
  - Not a good choice for high dimension
  - Supports insert, delete, find, nearest neighbor, range queries

Geometric Data Structures

- Geometric data structures are common.
- The k-d tree is one of the simplest.
  - Nearest neighbor search
  - Range queries
- Other data structures used for
  - 3-d graphics models
  - Physical simulations