CSE 326: Data Structures
Lecture #21
Multidimensional Search Trees

Henry Kautz
Winter Quarter 2002

1

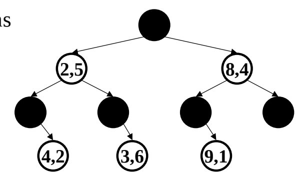
Today’s Outline

- Multidimensional search trees
- Range Queries
- *k*-D Trees
- Quad Trees

2

Multi-D Search ADT

- Dictionary operations
  - create
  - destroy
  - find
  - insert
  - delete
  - range queries



- Each item has *k* keys for a *k*-dimensional search tree
- Searches can be performed on one, some, or all the keys or on ranges of the keys

3

Applications of Multi-D Search

- Astronomy (simulation of galaxies) - 3 dimensions
- Protein folding in molecular biology - 3 dimensions
- Lossy data compression - 4 to 64 dimensions
- Image processing - 2 dimensions
- Graphics - 2 or 3 dimensions
- Animation - 3 to 4 dimensions
- Geographical databases - 2 or 3 dimensions
- Web searching - 200 or more dimensions

4

Range Query

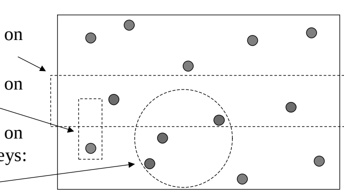
A *range query* is a search in a dictionary in which the exact key may not be entirely specified.

Range queries are the primary interface with multi-D data structures.

5

Range Query Examples: Two Dimensions

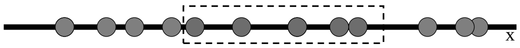
- Search for items based on *just one key*
- Search for items based on *ranges for all keys*
- Search for items based on a function of several keys: e.g., a *circular range query*



6

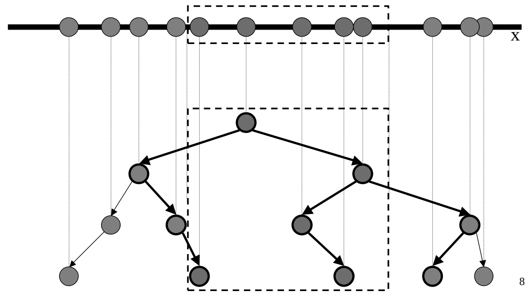
Range Querying in 1-D

Find everything in the rectangle…

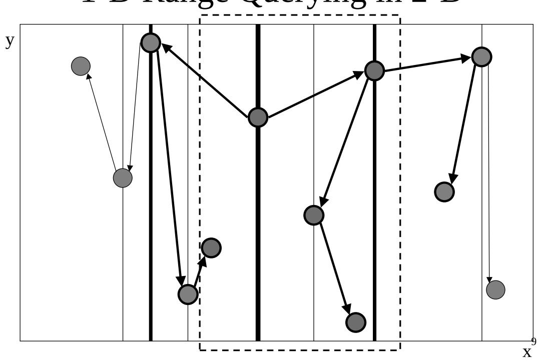


Range Querying in 1-D with a BST

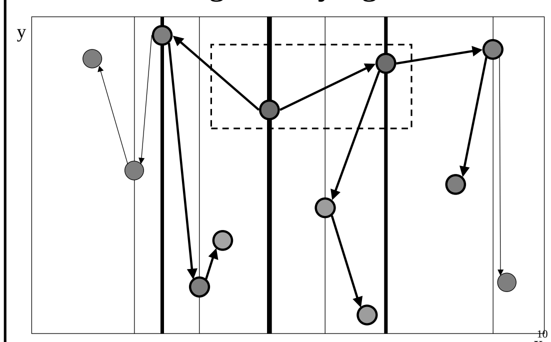
Find everything in the rectangle…



1-D Range Querying in 2-D



2-D Range Querying in 2-D



k-D Trees

- Split on the next dimension at each succeeding level
- If building in batch, choose the median along the current dimension at each level
  - guarantees logarithmic height and balanced tree
- In general, add as in a BST

k-D tree node

keys	value
dimension	
left	right

The dimension that this node splits on

Find in a *k*-D Tree

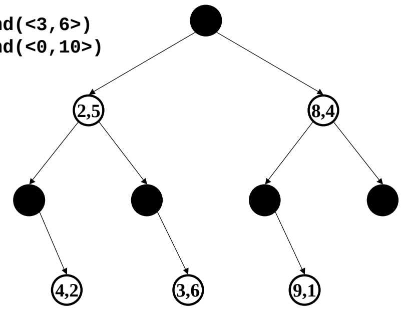
`find(<x`$_1$`,x`$_2$`, …, x`$_k$`>, root)` finds the node which has the given set of keys in it or returns **`null`** if there is no such node

```
Node *& find(const keyVector & keys,
             Node *& root) {
  int dim = root->dimension;
  if (root == NULL)
    return root;
  else if (root->keys == keys)
    return root;
  else if (keys[dim] < root->keys[dim])
    return find(keys, root->left);
  else
    return find(keys, root->right);
}
```

runtime:

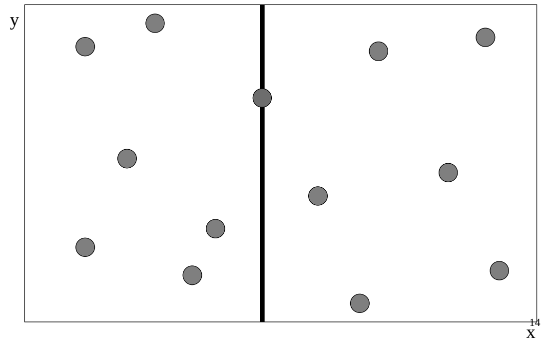
Find Example

```
find(<3,6>)
find(<0,10>)
```



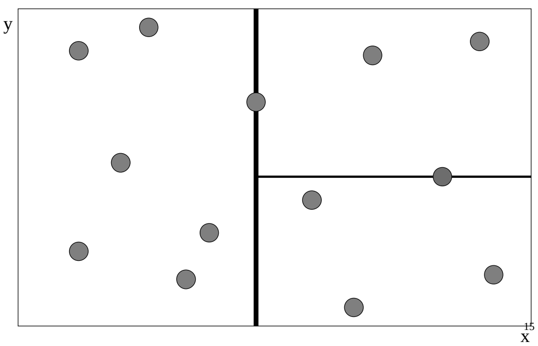
13

Building a 2-D Tree (1/4)



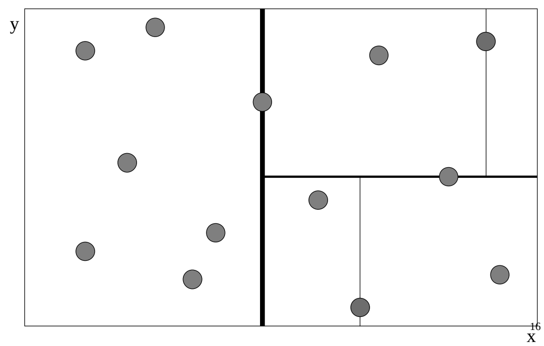
14

Building a 2-D Tree (2/4)



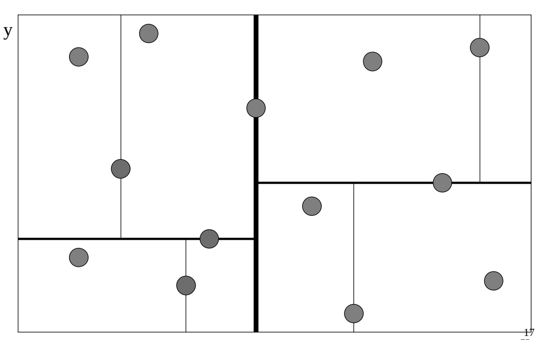
15

Building a 2-D Tree (3/4)



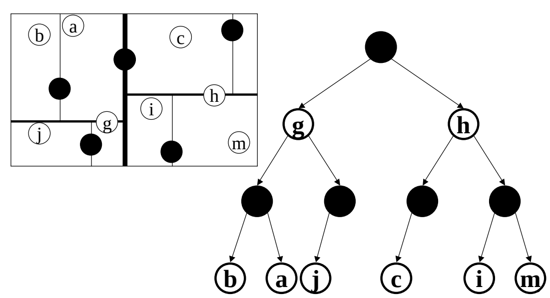
16

Building a 2-D Tree (4/4)

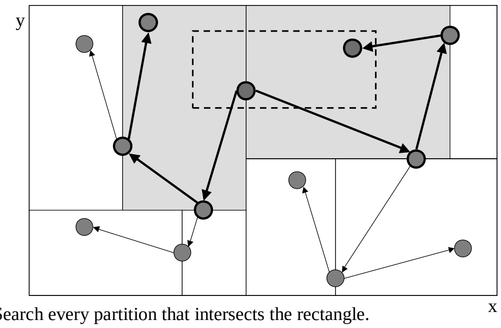


17

k-D Tree



2-D Range Querying in 2-D Trees



Search every partition that intersects the rectangle.
Check whether each node (including leaves) falls into the range.

19

Range Query in a 2-D Tree

```
print_range(int xlow, xhigh, ylow, yhigh, Node * root) {
   if (root == NULL) return;
   if ( xlow <= root.x && root.x <= xhigh &&
        ylow <= root.y && root.y <= yhigh ){
        print(root);
   if ((root.dim == “x” && xlow <= root.x ) ||
        (root.dim == “y” && ylow <= root.y ))
        print_range(root.left);
   if ((root.dim == “x” && root.x <= xhigh) ||
        (root.dim == “y” && root.y <= yhigh)
        print_range(root.right);
}
```

runtime: O(depth of tree)

20

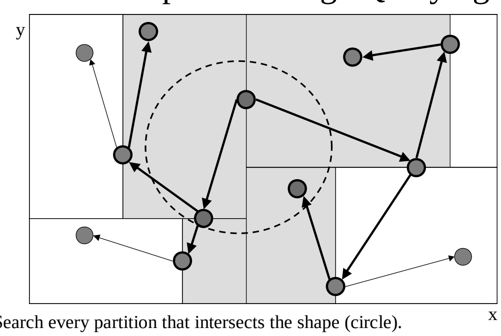
Range Query in a *k*-D Tree

```
print_range(int low[MAXD], high[MAXD], Node * root) {
   if (root == NULL) return;
   inrange = true;
   for (i=0; i<MAXD;i++){
        if ( root.coord[i] < low[i] ) inrange = false;
        if ( high[i] < root.coord[i] ) inrange = false; }
   if (inrange) print(root);
   if ((low[root.dim] <= root.coord[root.dim] )
        print_range(root.left);
   if (root.coord[root.dim] <= high[root.dim])
        print_range(root.right);
}
```

runtime: O(N)

21

Other Shapes for Range Querying

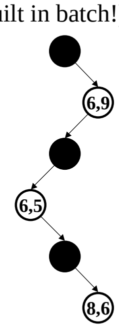


Search every partition that intersects the shape (circle).
Check whether each node (including leaves) falls into the shape.

22

k-D Trees Can Be Inefficient
(but not when built in batch!)

insert(<5,0>)
insert(<6,9>)
insert(<9,3>)
insert(<6,5>)
insert(<7,7>)
insert(<8,6>)

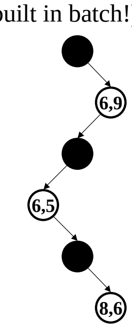


suck factor:

23

k-D Trees Can Be Inefficient
(but not when built in batch!)

insert(<5,0>)
insert(<6,9>)
insert(<9,3>)
insert(<6,5>)
insert(<7,7>)
insert(<8,6>)

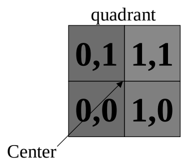


suck factor: O(n)

24

Quad Trees

- Split on *all* (two) dimensions at each level
- Split key space into equal size partitions (quadrants)
- Add a new node by adding to a leaf, and, if the leaf is already occupied, split until only one node per leaf



quad tree node

	keys	value
Center:	x	y
Quadrants:	0,0 1,0	0,1 1,1

25

Find in a Quad Tree

`find(<x, y>, root)` finds the node which has the given pair of keys in it or returns quadrant where the point should be if there is no such node

```
Node *& find(Key x, Key y, Node *& root) {
  if (root == NULL)
    return NULL;   // Empty tree
  if (root->isLeaf)
    return root;   // Key may not actually be here

  int quad = getQuadrant(x, y, root);
  return find(x, y, root->quadrants[quad]);
}
```

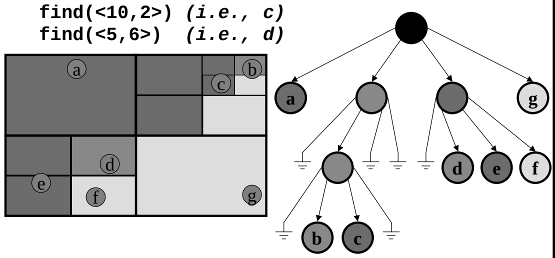
Compares against center; always makes the same choice on ties.

runtime: O(depth)

26

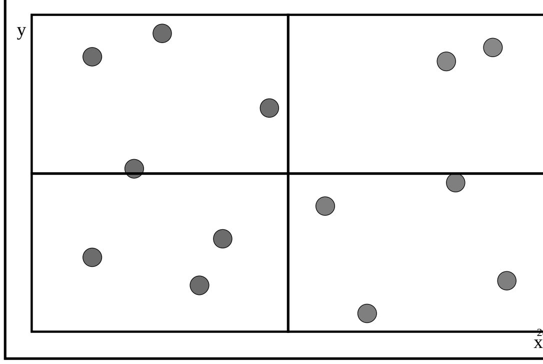
Find Example

`find(<10,2>)` ***(i.e., c)***
`find(<5,6>)` ***(i.e., d)***



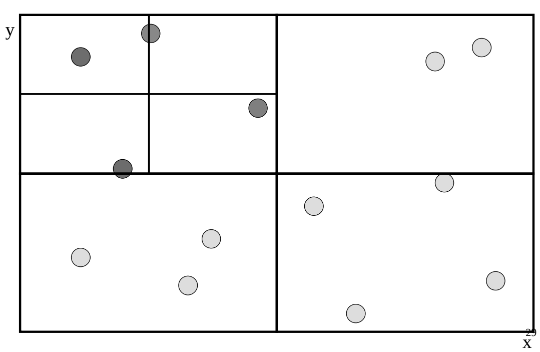
27

Building a Quad Tree (1/5)



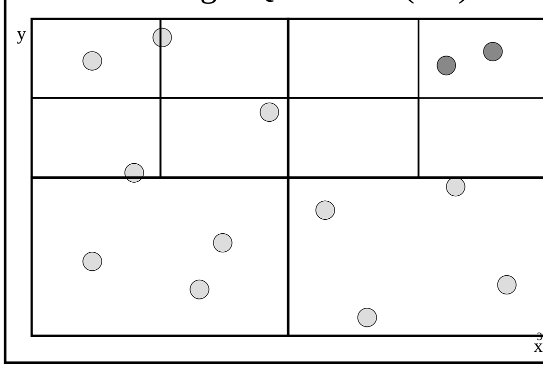
28

Building a Quad Tree (2/5)



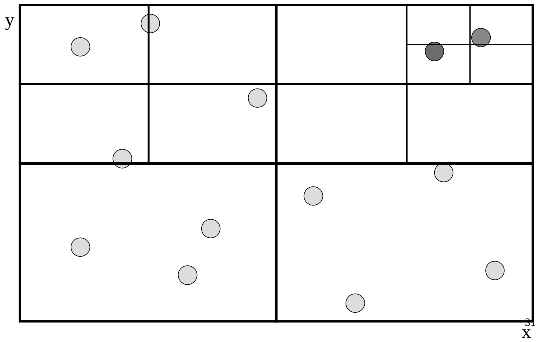
29

Building a Quad Tree (3/5)



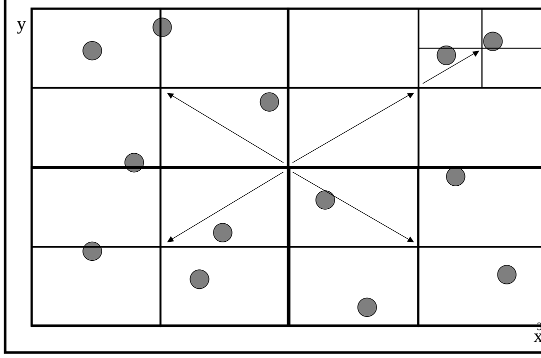
30

Building a Quad Tree (4/5)



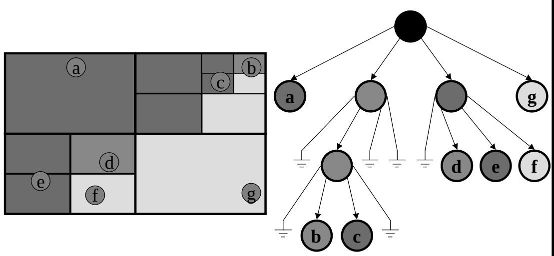
31

Building a Quad Tree (5/5)



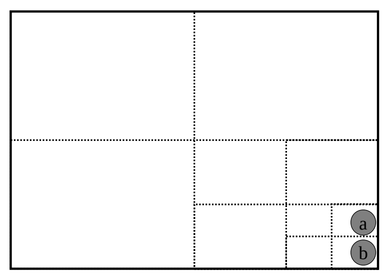
32

Quad Tree Example



33

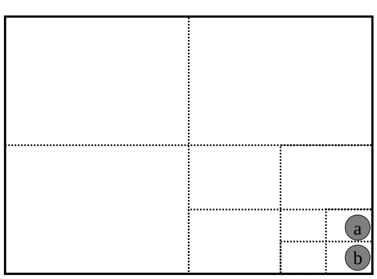
Quad Trees Can Suck



suck factor:

34

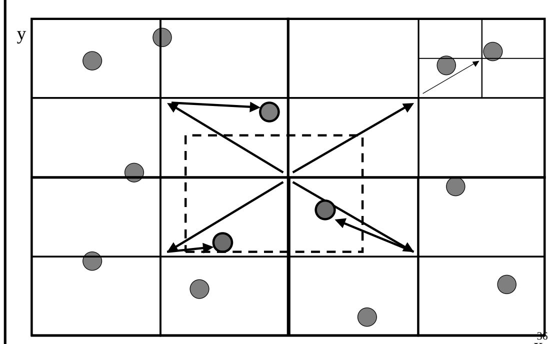
Quad Trees Can Suck



suck factor: O(log (1/minimum distance between nodes))

35

2-D Range Querying in Quad Trees



36

2-D Range Query in a Quad Tree

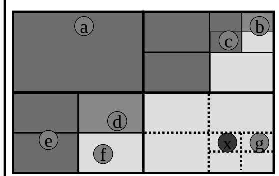
```
print_range(int xlow, xhigh, ylow, yhigh, Node * root){
   if (root == NULL) return;
   if ( xlow <= root.x && root.x <= xhigh &&
        ylow <= root.y && root.y <= yhigh ){
        print(root);

   if (xlow <= root.x && ylow <= root.y)
        print_range(root.lower_left);
   if (xlow <= root.x && root.y <= yhigh)
        print_range(root.upper_left);
   if (root.x <= x.high && ylow <= root.x)
        print_range(root.lower_right);
   if (root.x <= xhigh && root.y <= yhigh)
        print_range(root.upper_right);
}
```

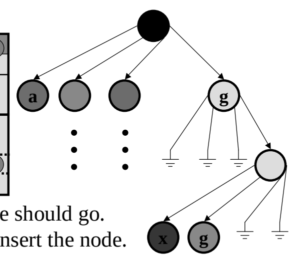
runtime: O(N)

37

Insert Example

`insert(<10,7>,x)`



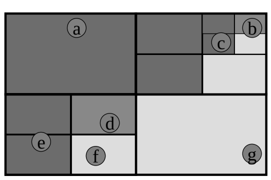


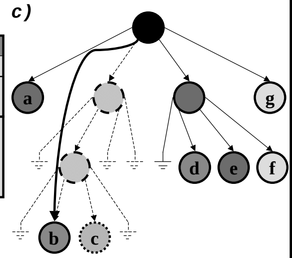
- Find the spot where the node should go.
- If the space is unoccupied, insert the node.
- If it is occupied, split until the existing node separates from the new one.

38

Delete Example

`delete(<10,2>)`*`(i.e., c)`*



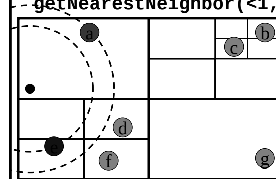


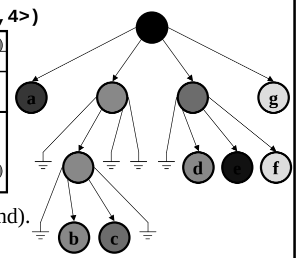
- Find and delete the node.
- If its parent has just one child, delete it.
- Propagate!

39

Nearest Neighbor Search

`getNearestNeighbor(<1,4>)`





- Find a nearby node (do a find).
- Do a circular range query.
- As you get results, tighten the circle.
- Continue until no closer node in query.

Works on *k*-D Trees, too!

40

Quad Trees vs. *k*-D Trees

- *k*-D Trees
  - Density balanced trees
  - Number of nodes is O(n) where *n* is the number of points
  - Height of the tree is O(log n) *with batch insertion*
  - Supports insert, find, nearest neighbor, range queries
- 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 $O(\log n + \log \Delta)$
  - Supports insert, delete, find, nearest neighbor, range queries

41

To Do / Coming Attractions

- Read (a little) about k-D trees in Weiss 12.6
- (More) Heaps ‘o fun
  - leftist heaps & binomial queues

42