CSE 326: Data Structures
Lecture #21
One Last Gasp

Bart Niswonger

Summer Quarter 2001

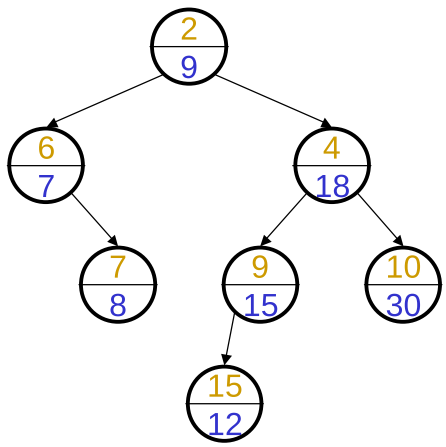
Today's Outline

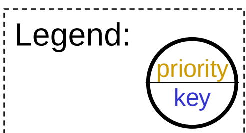
- Algorithm Design (from Friday)
  - Dynamic Programming
  - Randomized
  - Backtracking
- "Advanced" Data Structures

Treap Dictionary Data Structure

- Treaps have the binary search tree
  - binary tree property
  - search tree property
- Treaps also have the heap-order property!
  - randomly assigned priorities

heap in yellow; search tree in blue

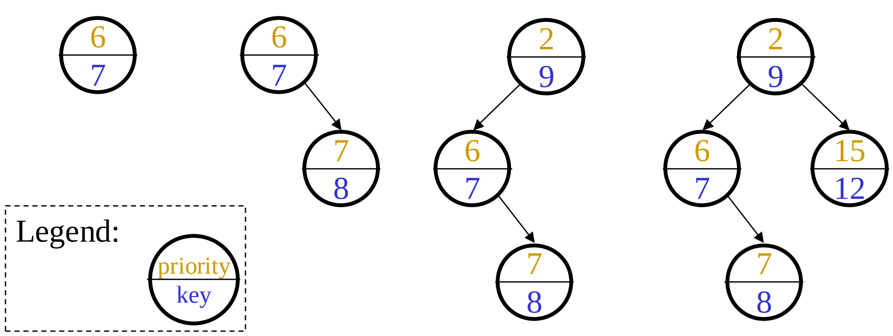




Tree + Heap… Why Bother?

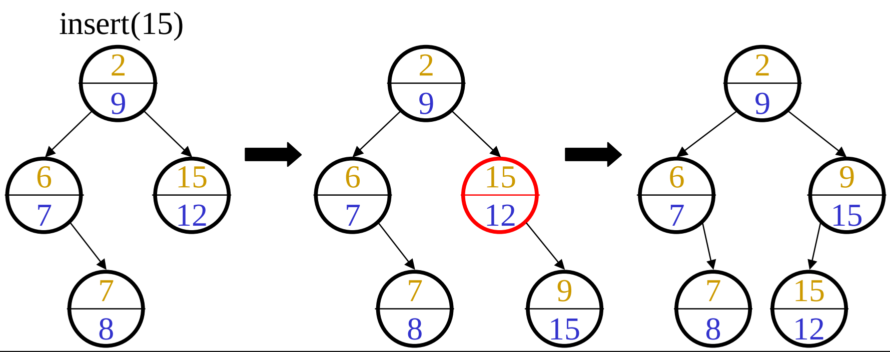
Insert data in sorted order into a treap; what shape tree comes out?

insert(7) insert(8) insert(9) insert(12)



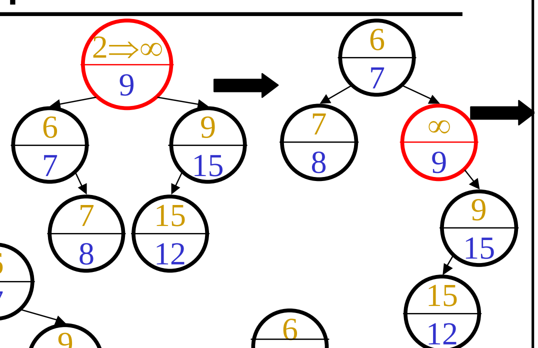
Treap Insert

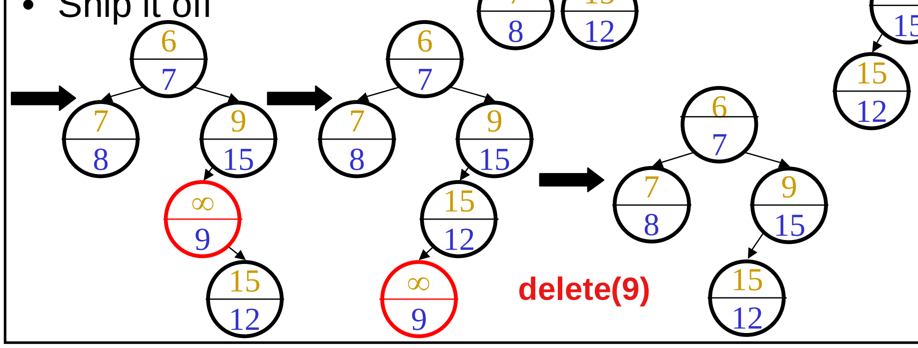
- Choose a random priority
- Insert as in normal BST
- Rotate up until heap order is restored



Treap Delete

- Find the key
- Increase its value to ∞
- Rotate it to the fringe
- Snip it off



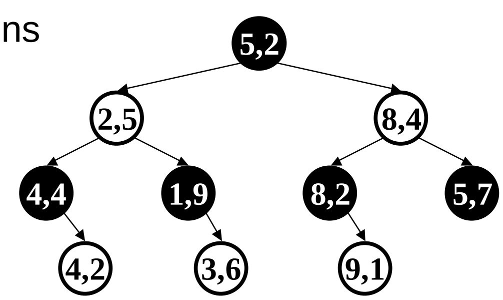


Treap Summary

- Implements Dictionary ADT
  - insert in expected O(log n) time
  - delete in expected O(log n) time
  - find in expected O(log n) time
- Memory use
  - O(1) per node
  - about the cost of AVL trees
- Complexity?

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

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

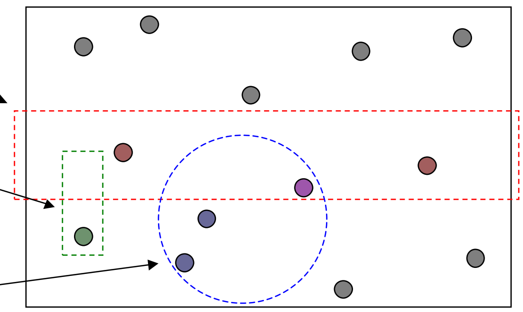
Range Query

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.

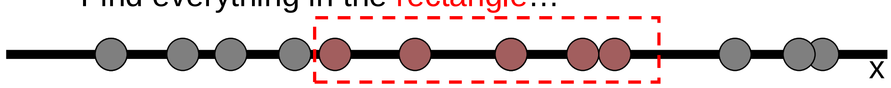
Range Query: Two Dimensions

- 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*



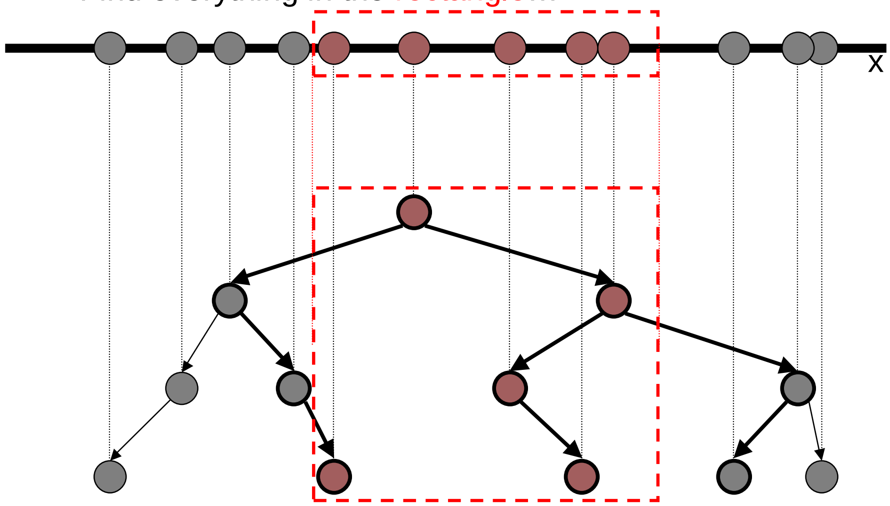
Range Querying in 1-D

Find everything in the rectangle…

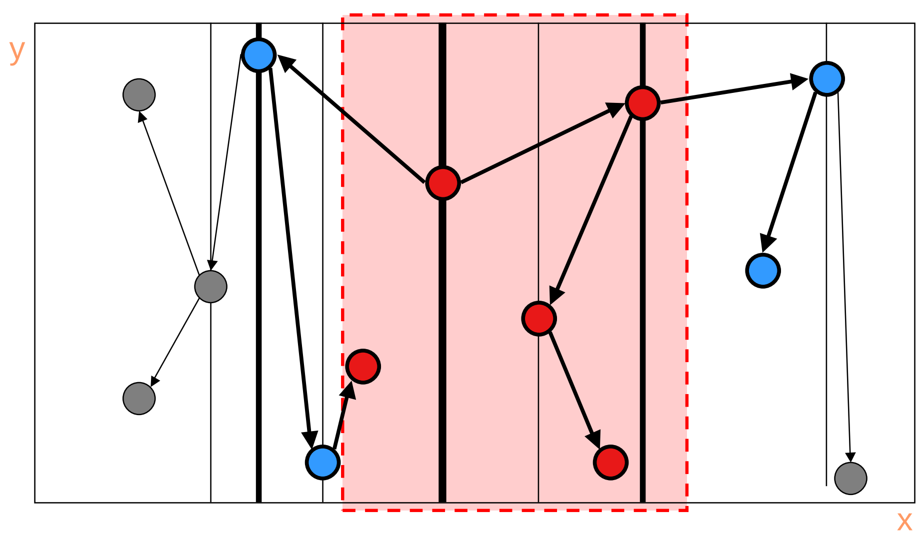


Range Querying in 1-D: BST

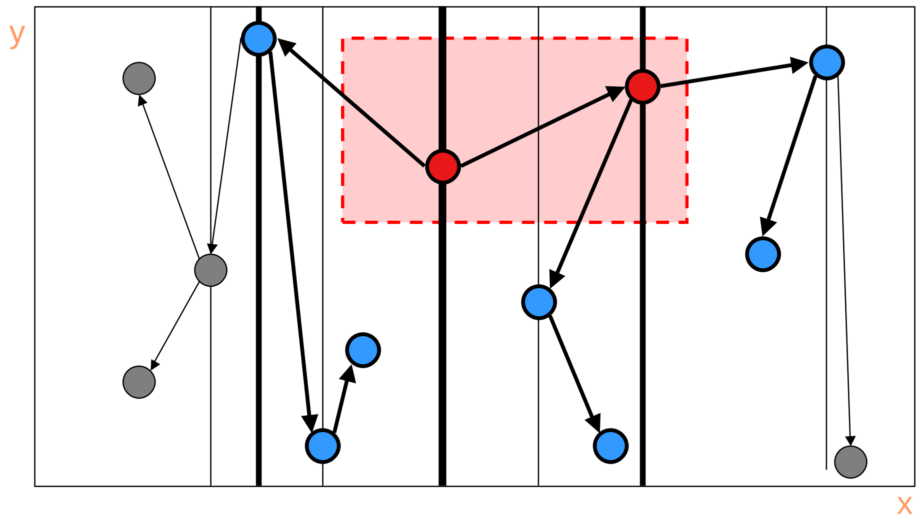
Find everything in the rectangle…



1-D Range Querying in 2-D

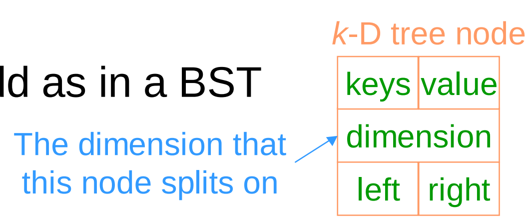


2-D Range Querying in 2-D

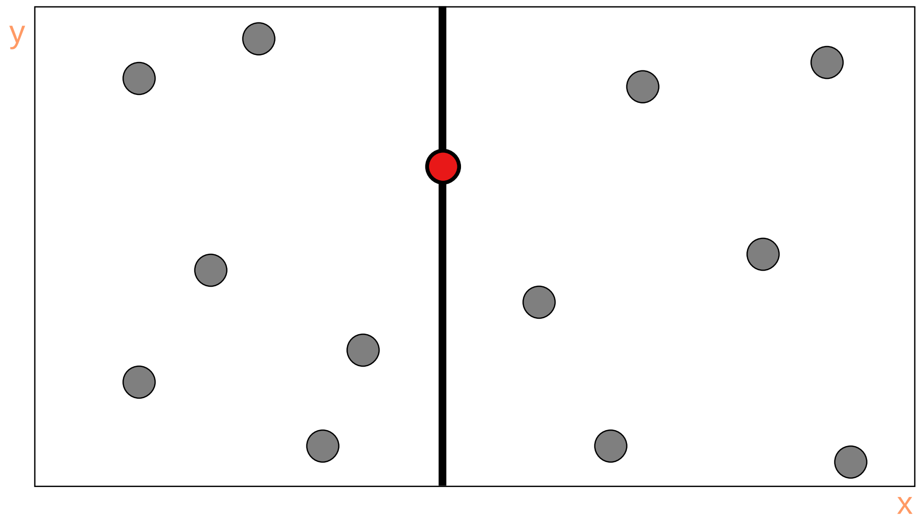


k-D Trees

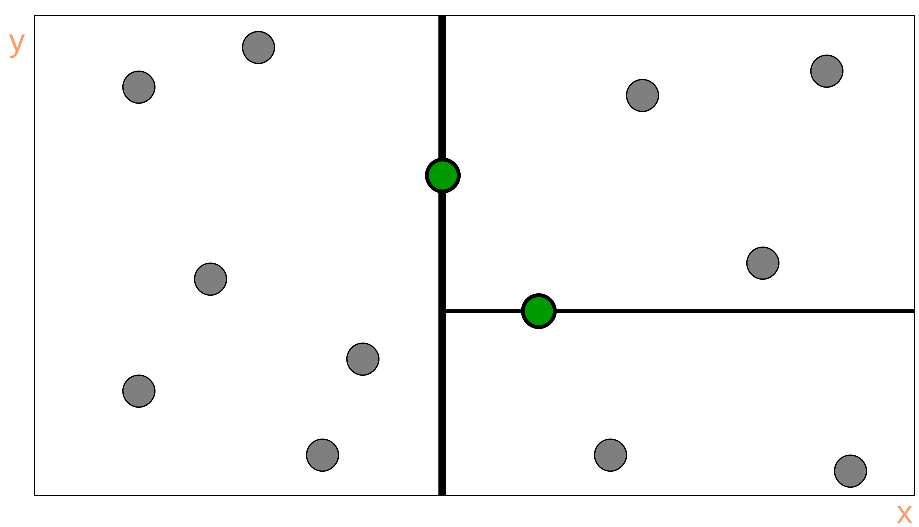
- Split on the next dimension at each succeeding level
- If building in batch, choose the median along the current dimension at each level
  - guarantees logarithmic height and balanced tree
- In general, add as in a BST



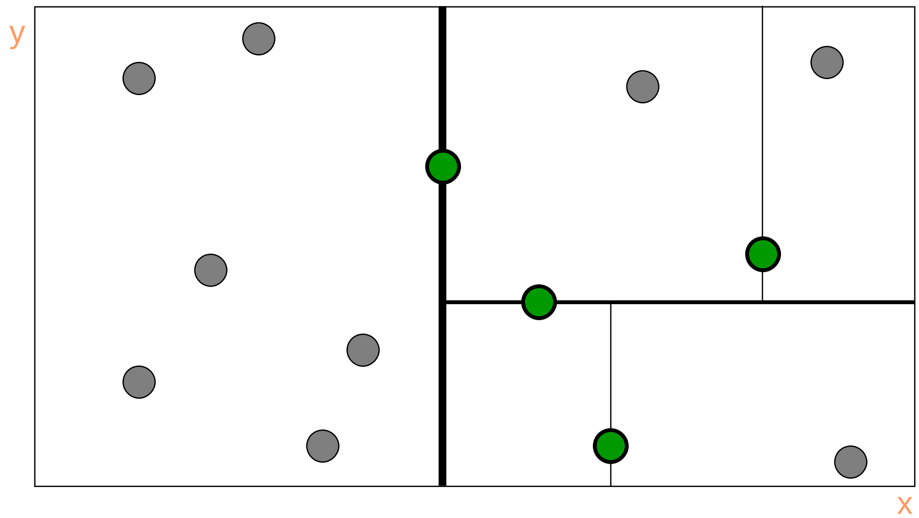
Building a 2-D Tree (1/4)



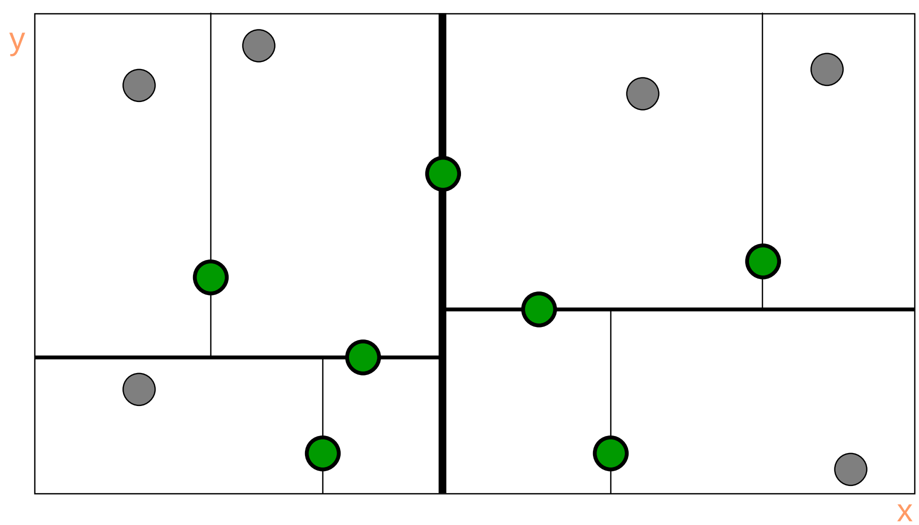
Building a 2-D Tree (2/4)



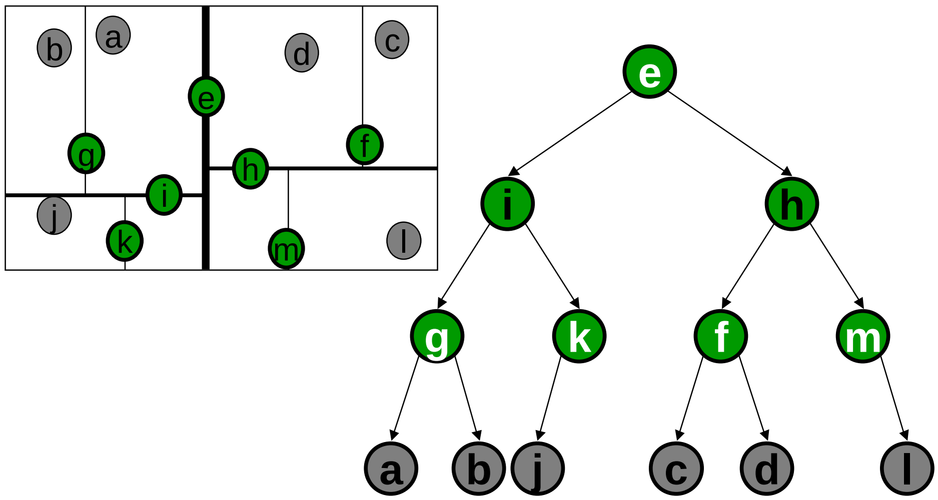
Building a 2-D Tree (3/4)



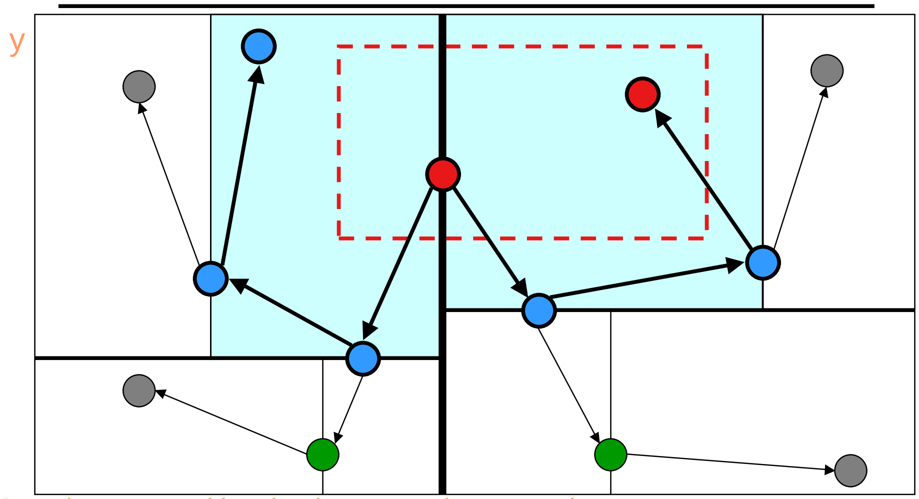
Building a 2-D Tree (4/4)



k-D Tree

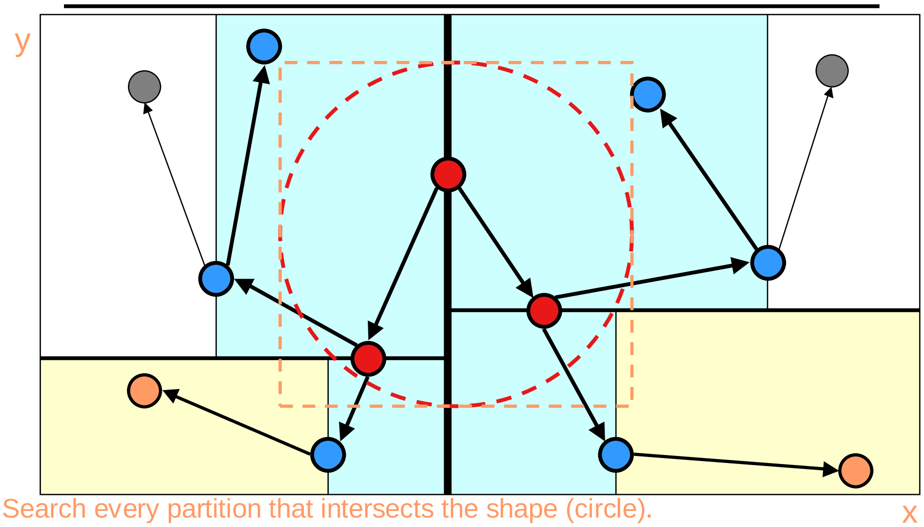


2-D Range Querying in 2-D Trees



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

Other Shapes for Range Querying



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

Find in a *k*-D Tree

```
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);
}
```

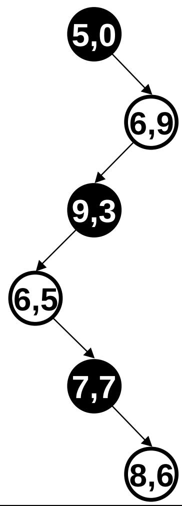
find(<x_1, x_2, …, x_k>, root) finds the node which has the given set of keys in it or returns `null` if there is no such node

runtime:

k-D Trees Can Suck

(but not when built in batch!)

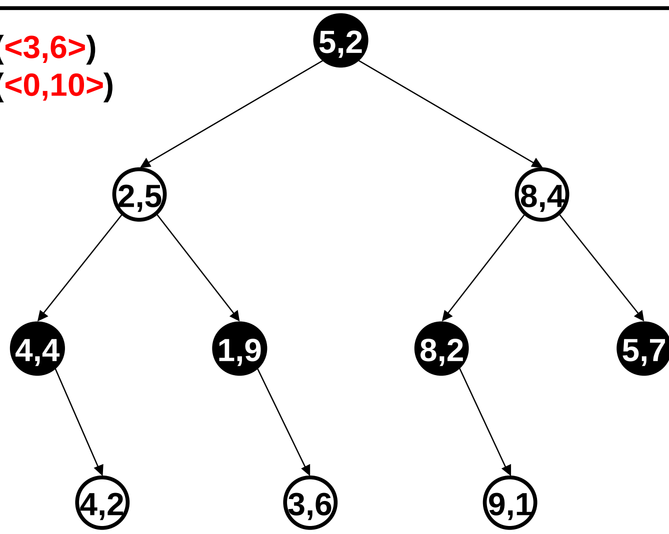
insert(<5,0>)
insert(<6,9>)
insert(<9,3>)
insert(<6,5>)
insert(<7,7>)
insert(<8,6>)



suck factor:

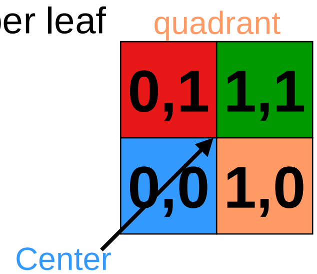
Find Example

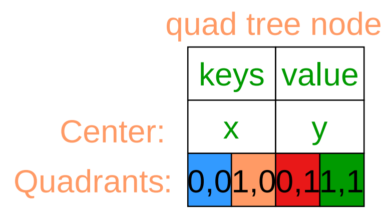
find(<3,6>)
find(<0,10>)



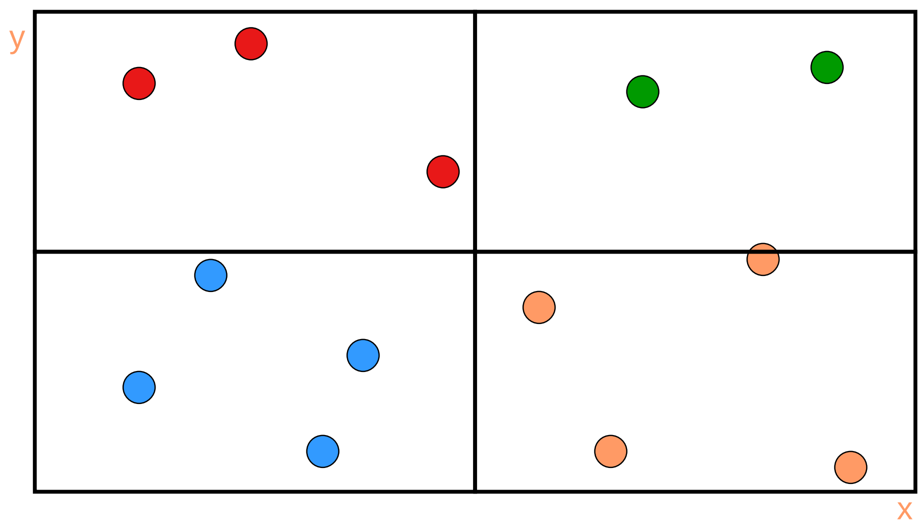
Quad Trees

- 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

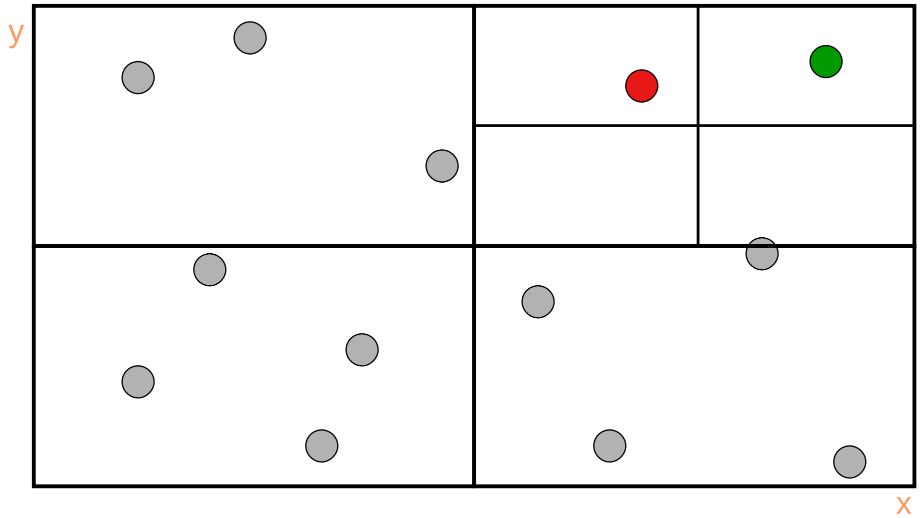




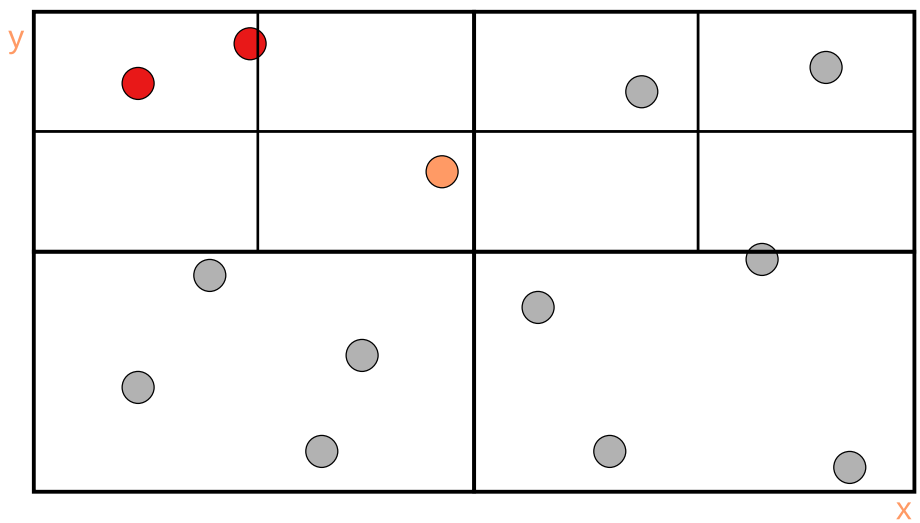
Building a Quad Tree (1/5)



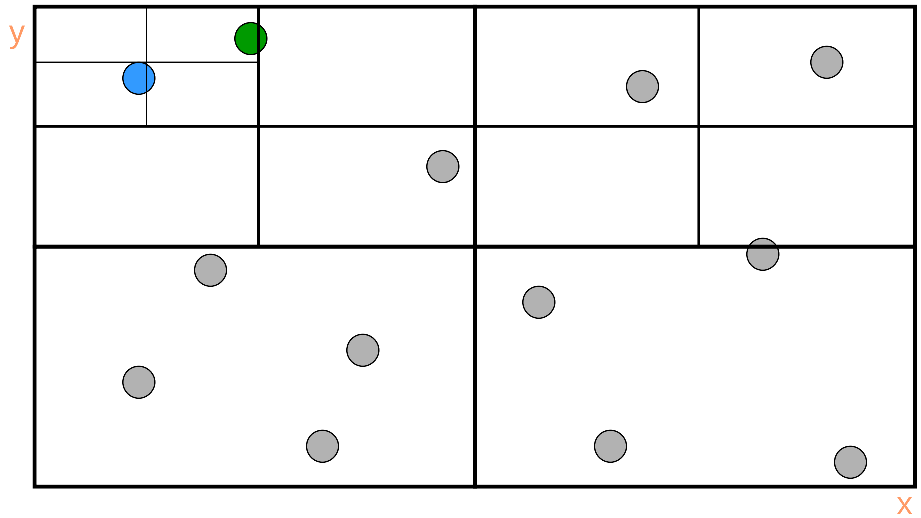
Building a Quad Tree (2/5)



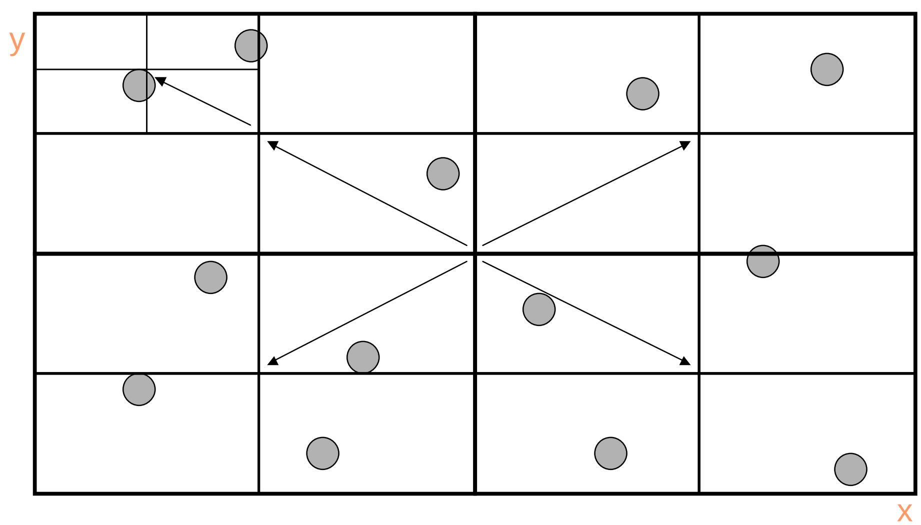
Building a Quad Tree (3/5)



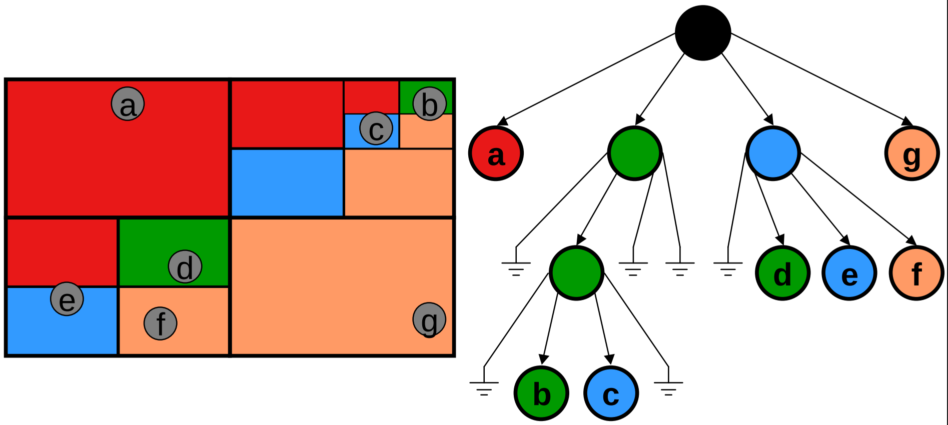
Building a Quad Tree (4/5)



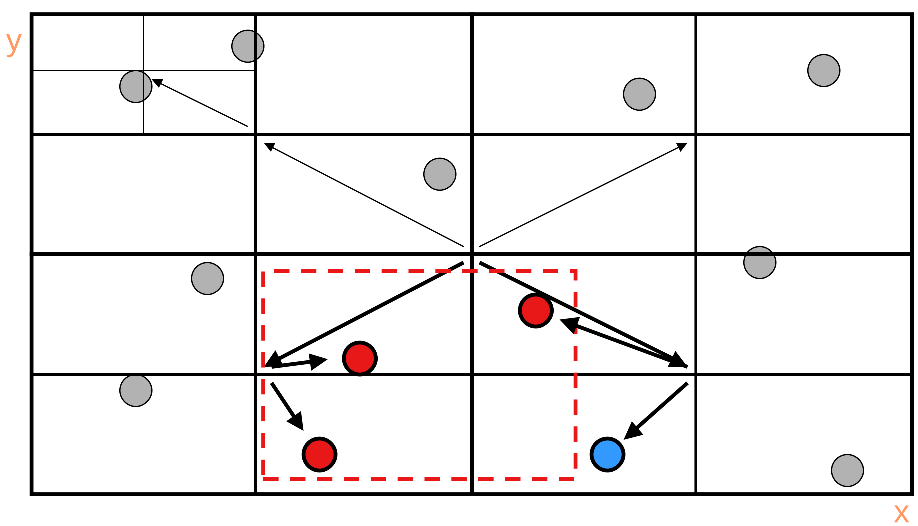
Building a Quad Tree (5/5)



Quad Tree Example



2-D Range Querying in Quad Trees



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

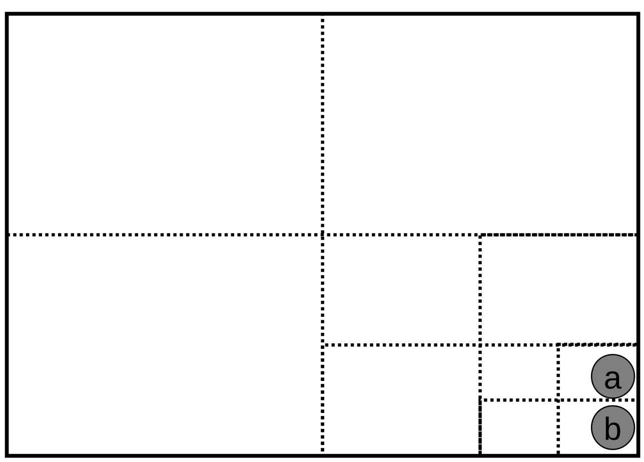
```
Node *& find(Key x, Key y, Node *& root) {
  if (root == NULL)
    return root;    // Empty tree
  if (root->isLeaf)
    return root;    // Key may not actually be here

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

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

runtime:

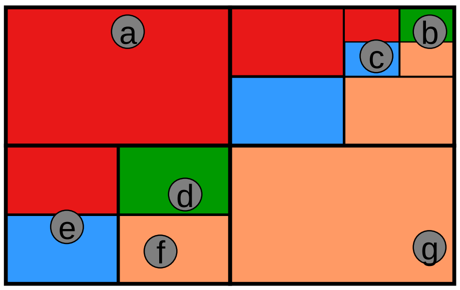
Quad Trees Can Suck

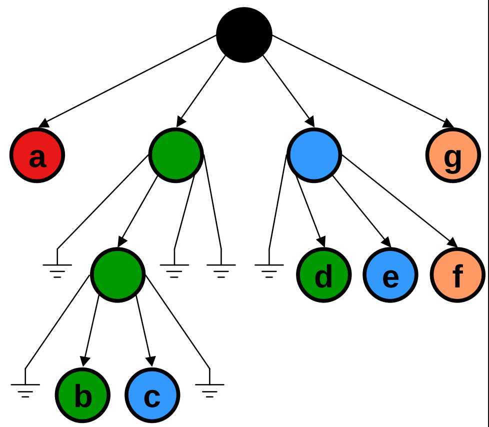


suck factor:

Find Example

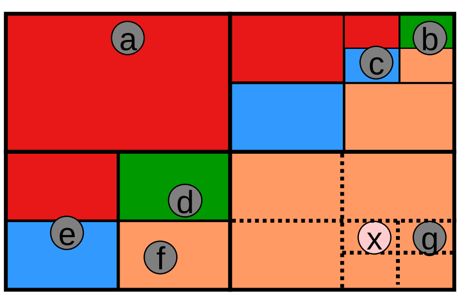
find(<10,2>) ***(i.e., c)***
find(<5,6>) ***(i.e., d)***

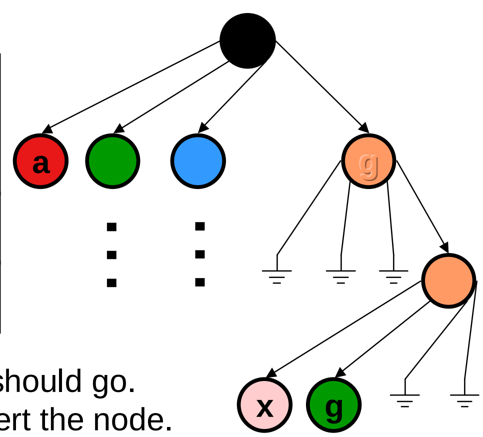




Insert Example

insert(<10,7>,x)

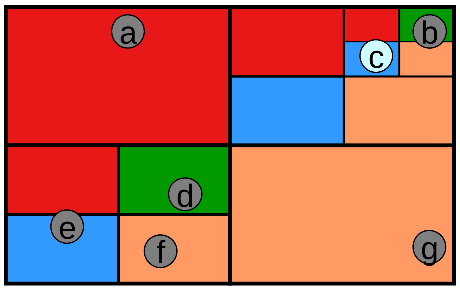


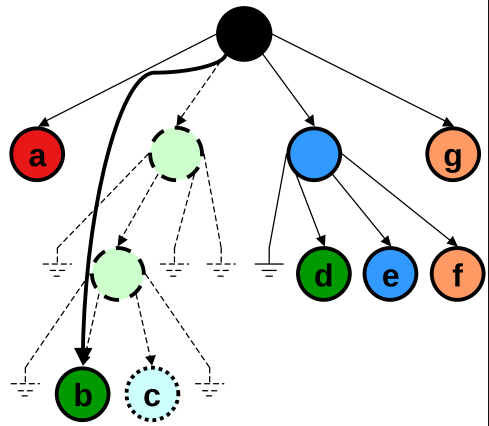


- Find the spot where the node should go.
- If the space is unoccupied, insert the node.
- If it is occupied, split until the existing node separates from the new one.

Delete Example

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

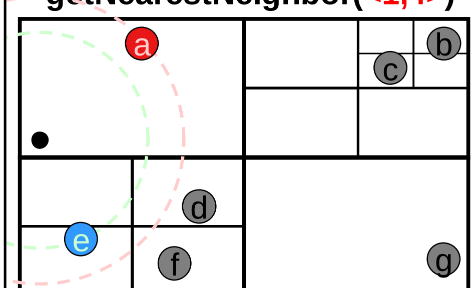


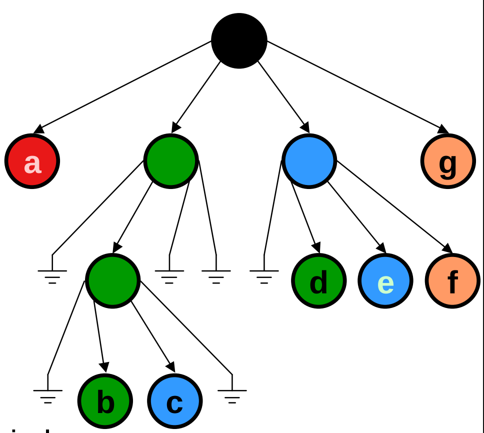


- Find and delete the node.
- If its parent has just one child, delete it.
- Propagate!

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!

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

To Do

- Project IV
  - Package up your executable and turn it in!
- Finish reading Chapter 12
- Study for the final!

Coming Up

- Course Discussion
- Final – Friday, this week!