CS61B

Lecture 25: Multi-Dimensional Data

- Multi-Dimensional Data
- Uniform Partitioning
- Quadtrees
- K-d Trees

Multi-dimensional Data

Yet Another Type of Map/Set

We've seen two fairly general implementations of sets and maps:

- **Hash Table**: Requires that keys can be hashed.
- **Search Tree**: Requires that keys can be compared.

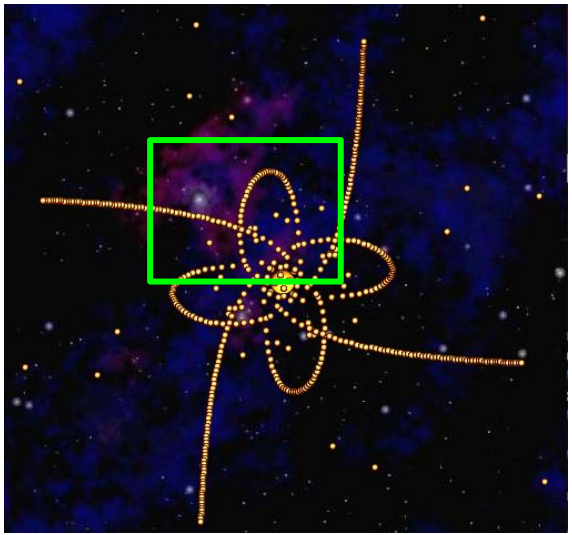
We've also seen two special case implementations:

- **Data Indexed Array**: Requires that keys are small.
- **Trie**: Requires that keys are strings.

Today, we'll discuss one very important special case: **Multi-dimensional keys**.

Motivation: 2D Range Finding and Nearest Neighbors

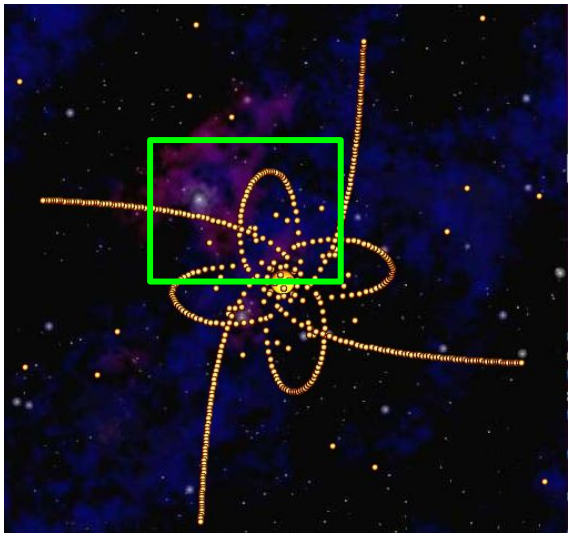
Suppose we want to perform operations on a **set** of Body objects in space?



Motivation: 2D Range Finding and Nearest Neighbors

Suppose we want to perform operations on a **set** of Body objects in space?

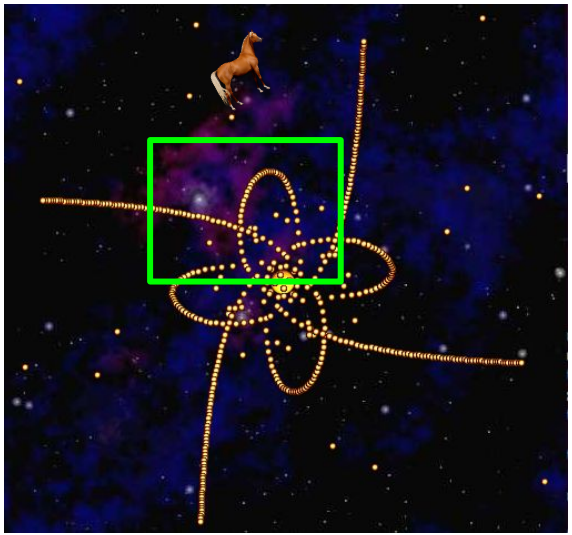
- 2D Range Searching: How many objects are in the highlighted rectangle?



Motivation: 2D Range Finding and Nearest Neighbors

Suppose we want to perform operations on a **set** of Body objects in space?

- 2D Range Searching: How many objects are in the highlighted rectangle?
- Nearest Neighbors: What is the closest object to the space horse?



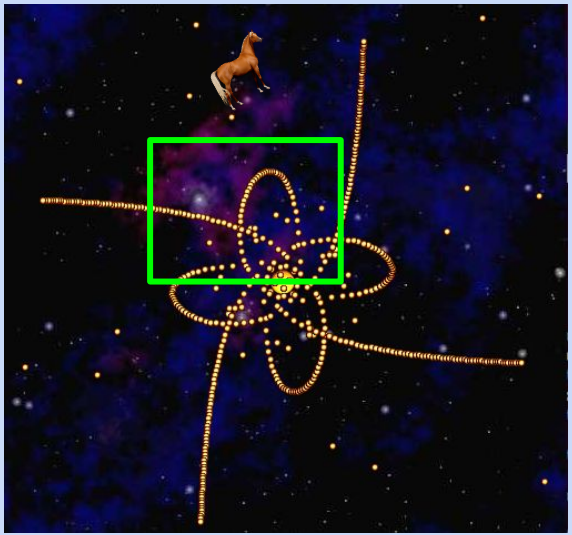
Motivation: 2D Range Finding and Nearest Neighbors

Suppose we want to perform operations on a **set** of Body objects in space?

- 2D Range Searching: How many objects are in the highlighted rectangle?
- Nearest Neighbors: What is the closest object to the space horse? 

Suppose the set is stored as a hash table, where the hash code of each Body is given below.

- How would nearest() be implemented?
- What would be the runtime of nearest?



```
hashCode pseudocode:
  return xPos.hashCode() * 31^4 + yPos.hashCode() * 31^3 +
         xVel.hashCode() * 31^2 + yVel.hashCode() * 31^1 +
         mass.hashCode()
```

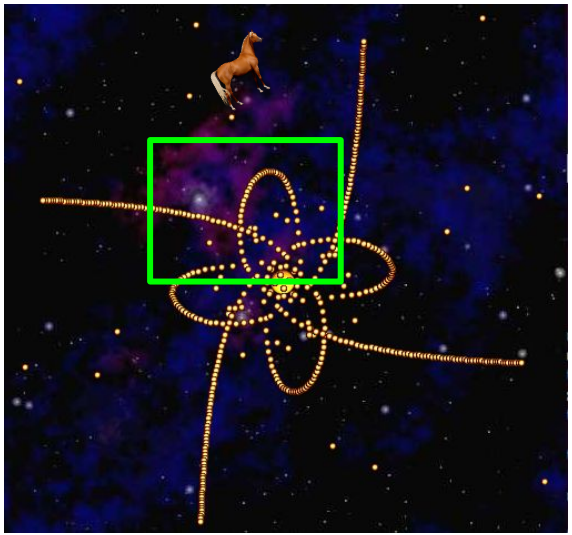
Motivation: 2D Range Finding and Nearest Neighbors

Suppose we want to perform operations on a **set** of Body objects in space?

- 2D Range Searching: How many objects are in the highlighted rectangle?
- Nearest Neighbors: What is the closest object to the space horse? 

Suppose the set is stored as a hash table, where the hash code of each Body is given below.

- How would nearest() be implemented?
  - The bucket that each object is effectively random. Have to iterate over all N items.
- What would be runtime of nearest?
  - Θ(N)



Uniform Partitioning

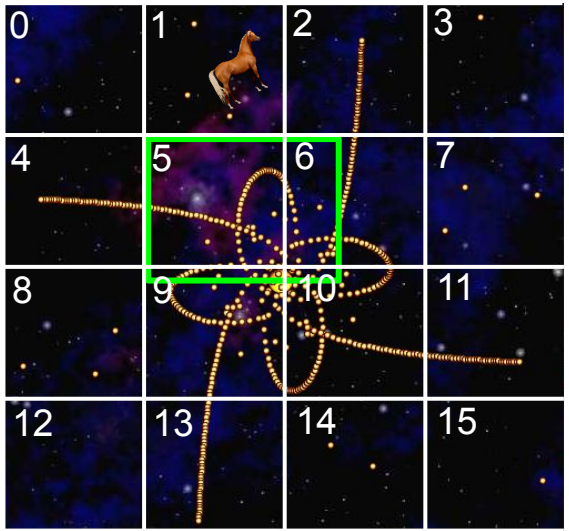
Uniform Partitioning

The problem with hash tables is that the bucket # of an item is effectively random.

- One fix is to ensure that the bucket #s depend only on position!

Simplest idea: Partition space into uniform rectangular buckets (also called “bins”).

- Example on right for 4x4 grid of such buckets.



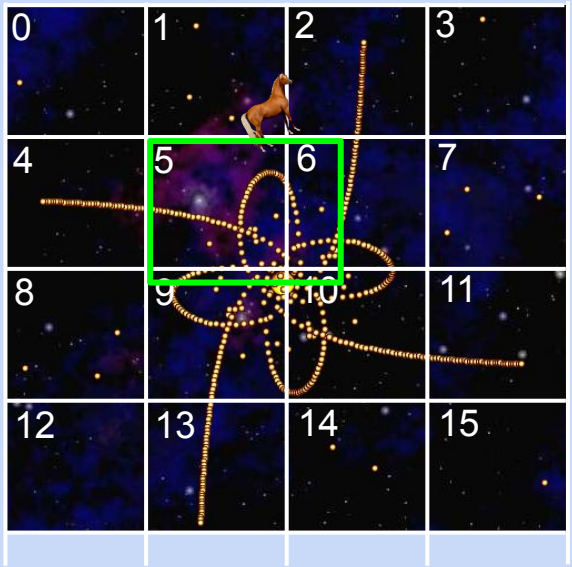
Uniform Partitioning

The problem with hash tables is that the bucket # of an item is effectively random.

- One fix is to ensure that the bucket #s depend only on position!

Questions:

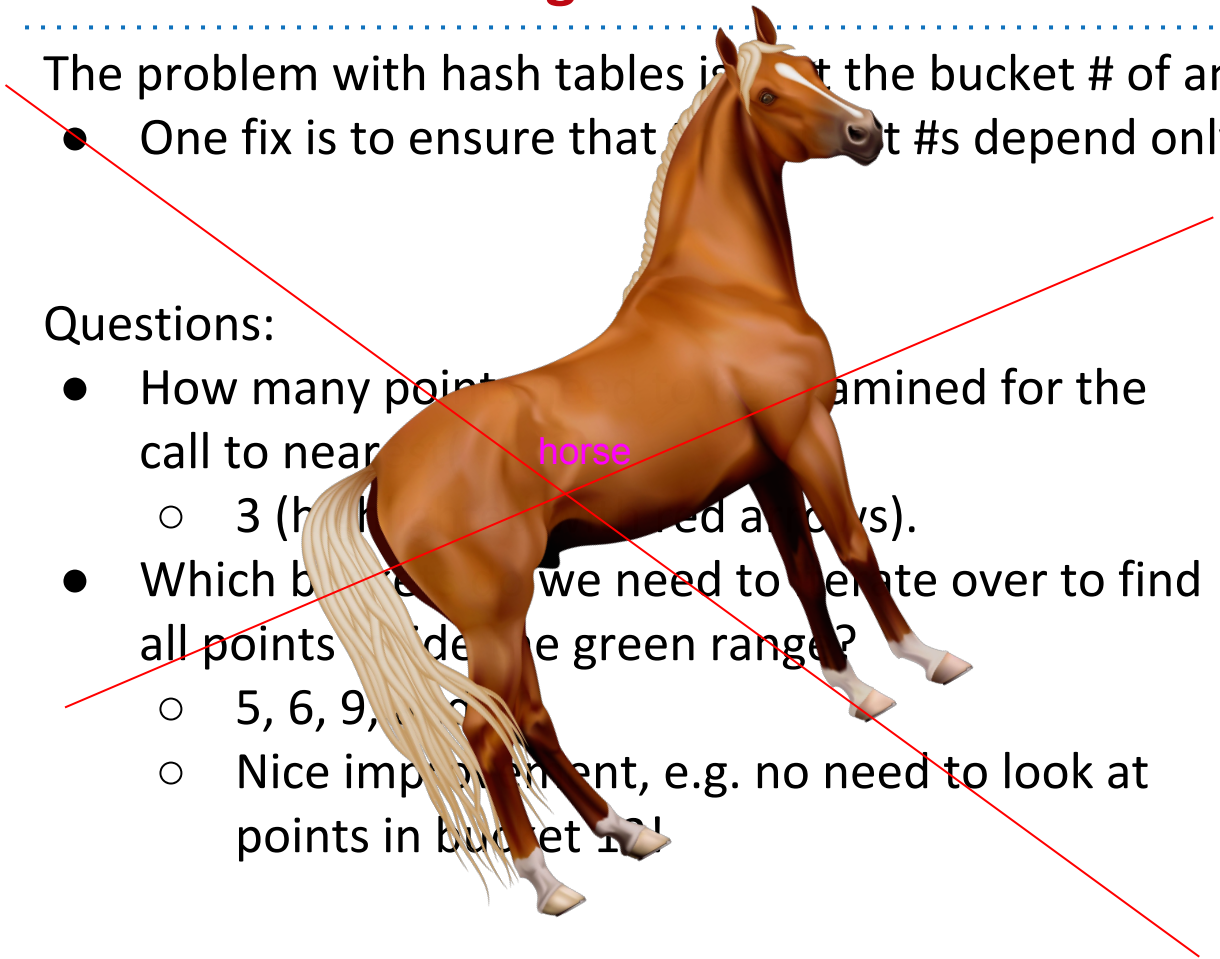
- How many boxes do we need to look at to solve nearest()? **1, 2, 5, 6, 0, 4**
- Which buckets do we need to iterate over to find all points in the green range?



Uniform Partitioning

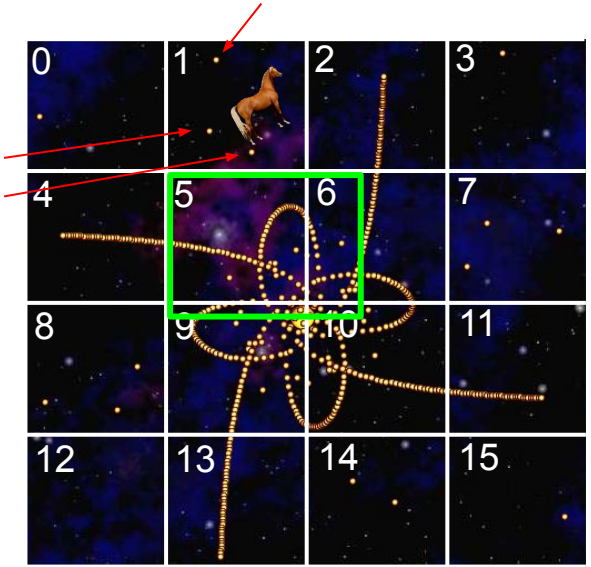
The problem with hash tables is that the bucket # of an item is effectively random.

- One fix is to ensure that bucket #s depend only on position!



Questions:

- How many points must be examined for the call to nearest(horse)?
  - 3 (highlighted arrows).
- Which buckets do we need to iterate over to find all points inside the green range?
  - 5, 6, 9, 10
  - Nice improvement, e.g. no need to look at points in bucket 12!



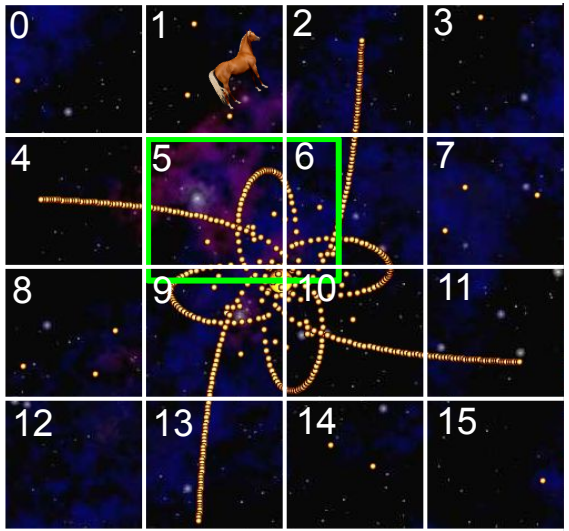
Uniform Partitioning

The problem with hash tables is that the bucket # of an item is effectively random.

- One fix is to ensure that the bucket #s depend only on position!

Simplest idea: Partition space into uniform rectangular buckets (also called “bins”).

- Example on right for 4x4 grid of such buckets.
- Typical implementation would have the container itself compute a bucket number. That is:
  - Don’t use the object’s hashCode().
  - Each object provides getX() and getY().
- This is sometimes called “spatial hashing”, though the term appears to be unofficial.



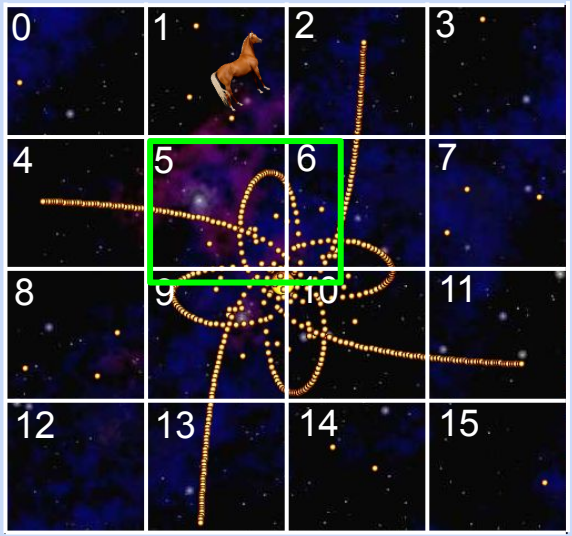
Uniform Partitioning

The problem with hash tables is that the bucket # of an item is effectively random.

- One fix is to ensure that the bucket #s depend only on position!

Question:

- What is the runtime for nearest assuming points are evenly spread out?



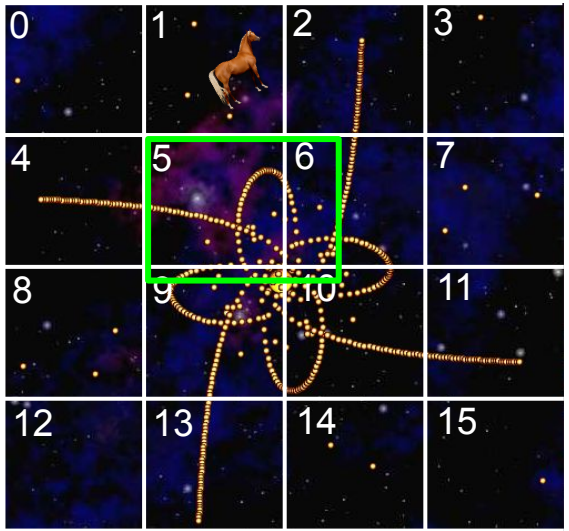
Uniform Partitioning

The problem with hash tables is that the bucket # of an item is effectively random.

- One fix is to ensure that the bucket #s depend only on position!

Question:

- What is the runtime for nearest assuming points are evenly spread out?
  - Still Θ(N).
  - On average, runtime will be 16 times faster than without the spatial partitioning, but N/16 is still Θ(N).
  - Note: Often works well in practice. But let's see a better way.



QuadTrees

Trees vs. Hash Tables

One key advantage of Search Trees over Hash Tables is that trees explicitly track the order of items.

Examples:

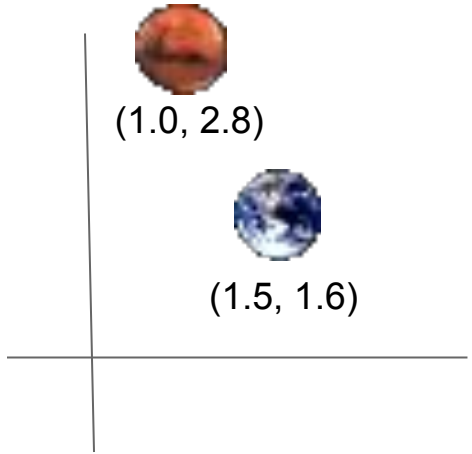
- Finding the minimum item in a BST is Θ(log N) time, but Θ(N) in a hash table.
- Can relatively easy modify BSTs to support operations like rank(5), which returns the 5th largest item in Θ(log N) time (see optional textbook).
    - Impossible to do efficiently in a hash table.

We'll exploit similar properties to improve the runtime of spatial operations like nearest.

Building Trees of Two Dimensional Data

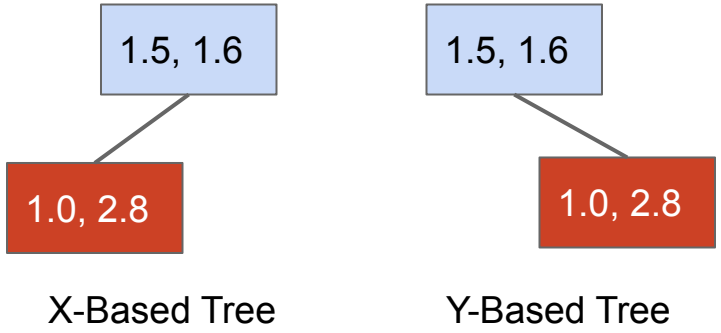
Consider trying to build a BST of Body objects in 2D space.

- earth.xPos = 1.5, earth.yPos = 1.6
- mars.xPos = 1.0, mars.yPos = 2.8



For a BST, we need some notion of “less than”:

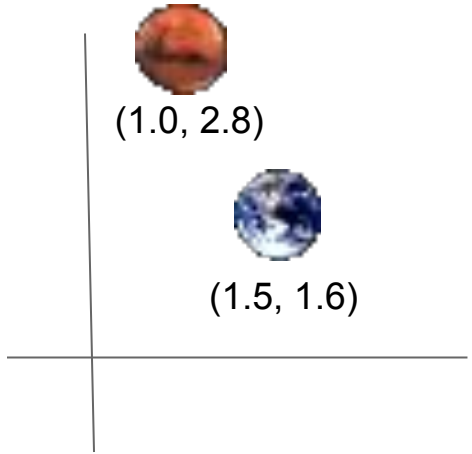
- In xPos, Mars < Earth (tree on left).
- In yPos, Mars > Earth (tree on right).



Building Trees of Two Dimensional Data

Consider trying to build a BST of Body objects in 2D space.

- earth.xPos = 1.5, earth.yPos = 1.6
- mars.xPos = 1.0, mars.yPos = 2.8

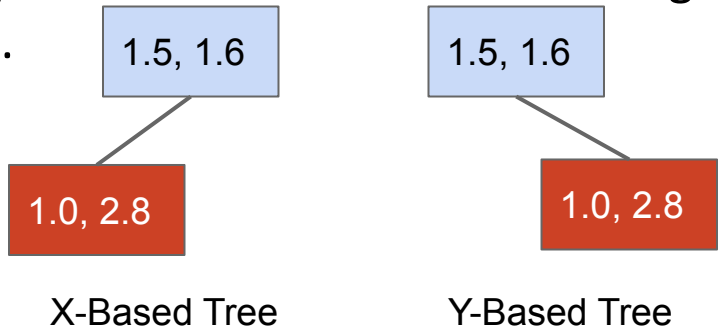


For a BST, we need some notion of “less than”:

- In xPos, Mars < Earth (tree on left).
- In yPos, Mars > Earth (tree on right).

Could just pick one, but you’re losing some of your information about ordering.

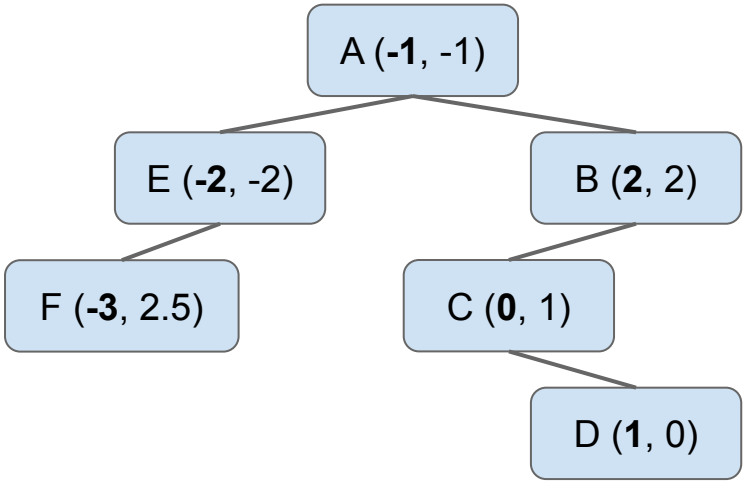
- Let’s get a quick sense of why this matters.

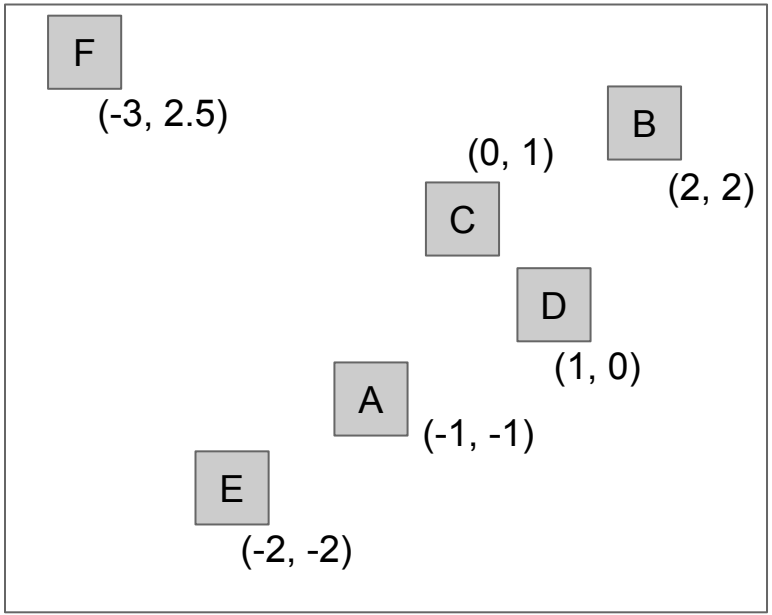


Building Trees of Two Dimensional Data

Example: Suppose we put points into a BST map ordered by x-coordinate.

- `put((-1, -1), A)`
- `put(( 2,  2), B)`
- `put(( 0,  1), C)`
- `put(( 1,  0), D)`
- `put((-2, -2), E)`
- `put((-3, 2.5), F)`

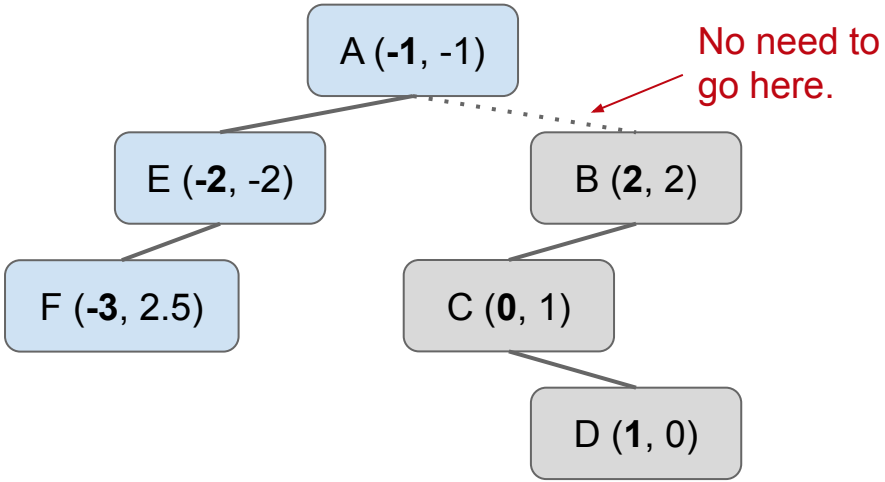


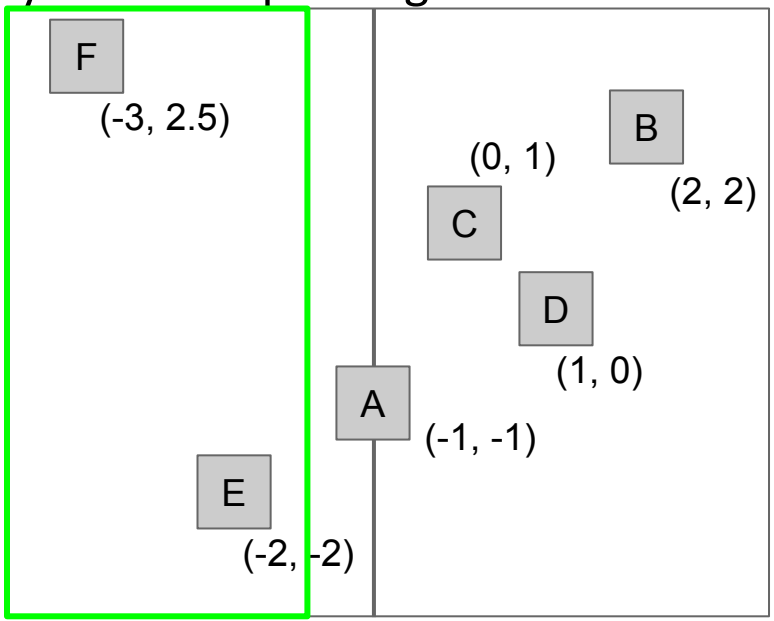


Building Trees of Two Dimensional Data

If we now ask “What are all the points with x-coordinate less than -1.5?”

- Since A partitions the space into points that have x < -1 and > -1, we know we don’t have to explore the right side.
- This process of cutting off a tree search early is called “pruning”.

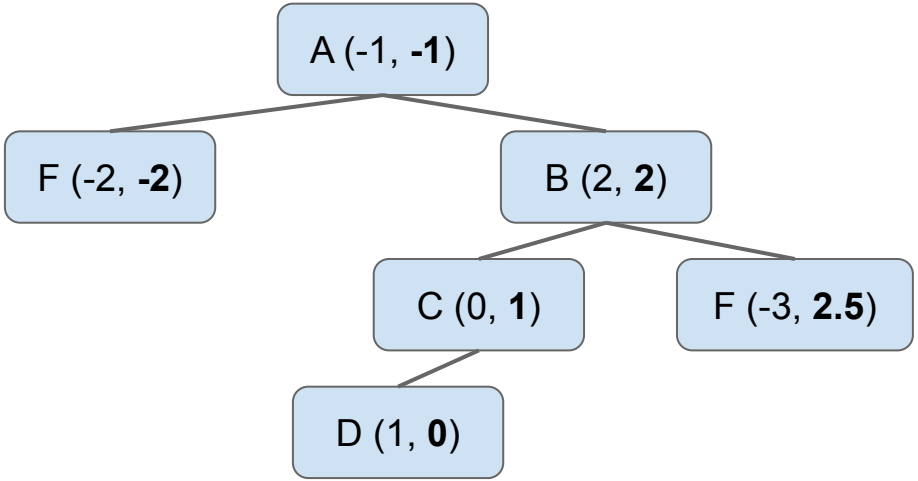


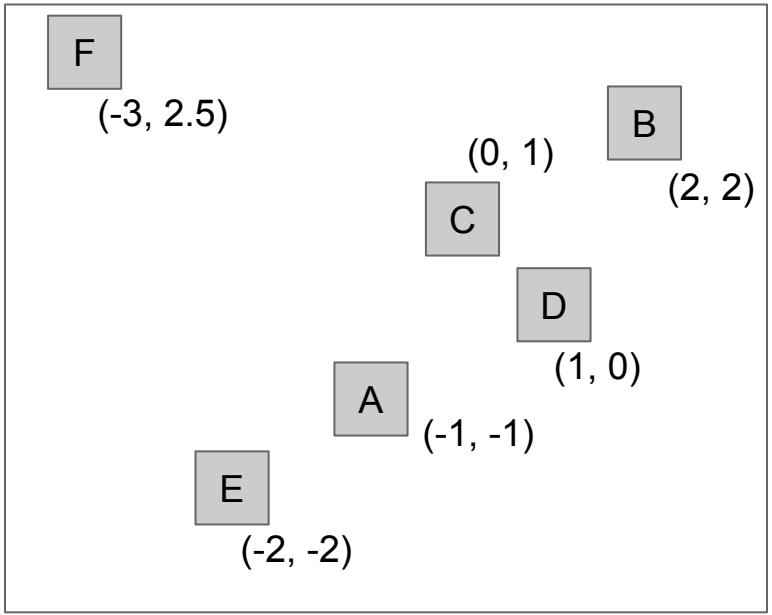


Building Trees of Two Dimensional Data

If we'd used y coordinate for comparisons, we'd get a slightly different BST.

- `put((-1, -1), A)`
- `put(( 2, 2), B)`
- `put(( 0, 1), C)`
- `put(( 1, 0), D)`
- `put((-2, -2), E)`
- `put((-3, 2.5), F)`

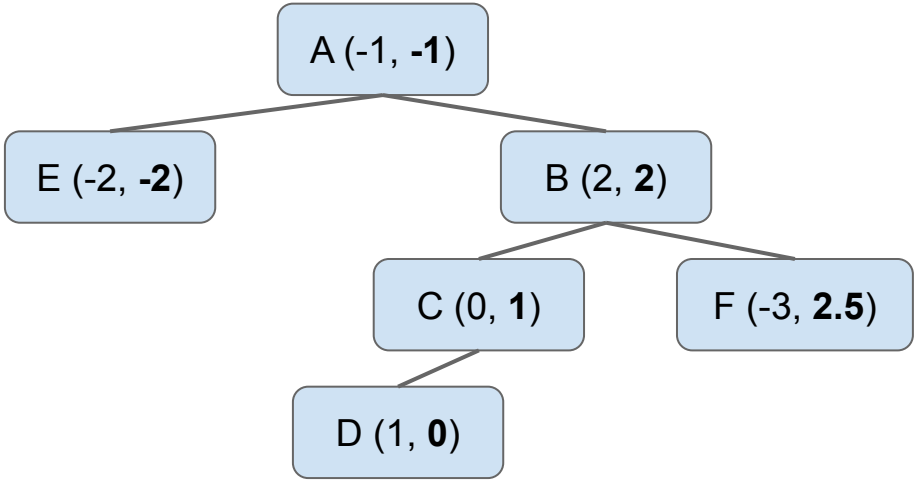


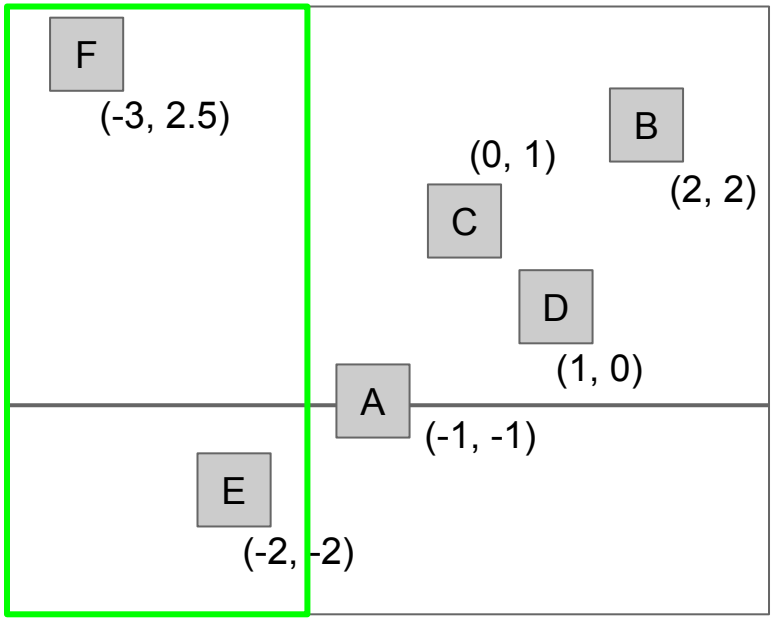


Building Trees of Two Dimensional Data

If we now ask “What are all the points with x-coordinate less than -1.5?”

- This time, can’t do any pruning!
  - Could be points on either side with x-coordinate less than -1.5.

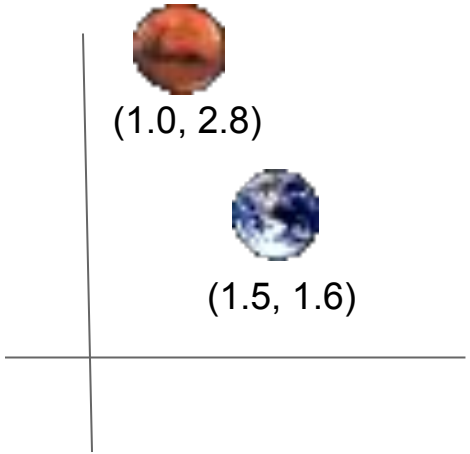




Building Trees of Two Dimensional Data

Consider trying to build a BST of Body objects in 2D space.

- earth.xPos = 1.5, earth.yPos = 1.6
- mars.xPos = 1.0, mars.yPos = 2.8



For a BST, we need some notion of “less than”:

- In xPos, Mars < Earth (tree on left).
- In yPos, Mars > Earth (tree on right).

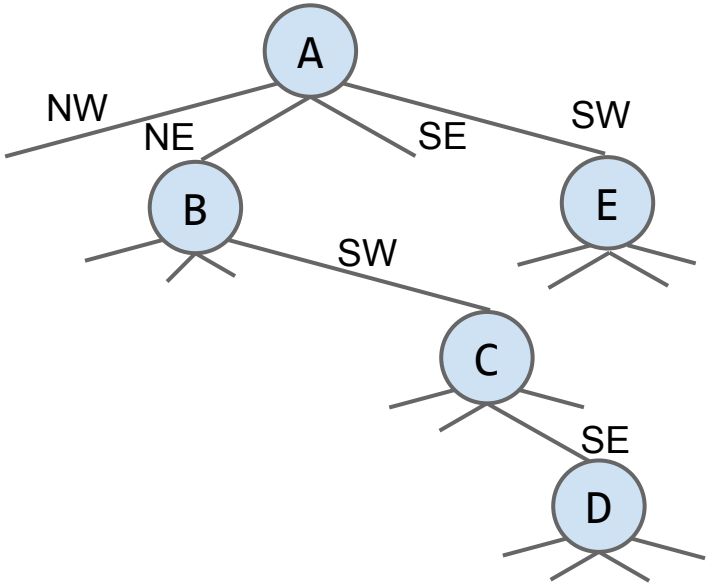
Could just pick one, but you’re losing some of your information about ordering.

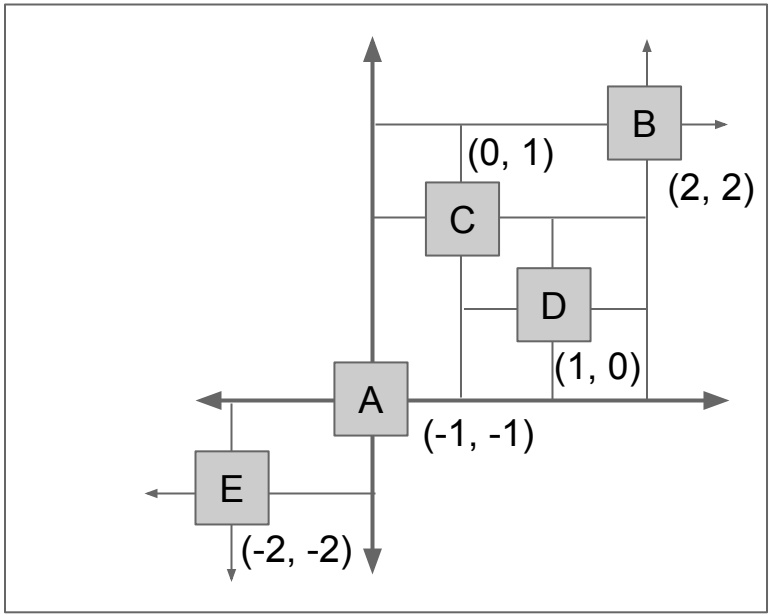
- Results in suboptimal pruning.
- There is a better way!
- Basic idea: Build a BST where every node has 4 neighbors, corresponding to northwest, northeast, southwest, and southeast.

Quadtree Insertion Demo

Below: Quadtree Representation of 5 objects in 2D space.

- Insertion Demo: Link

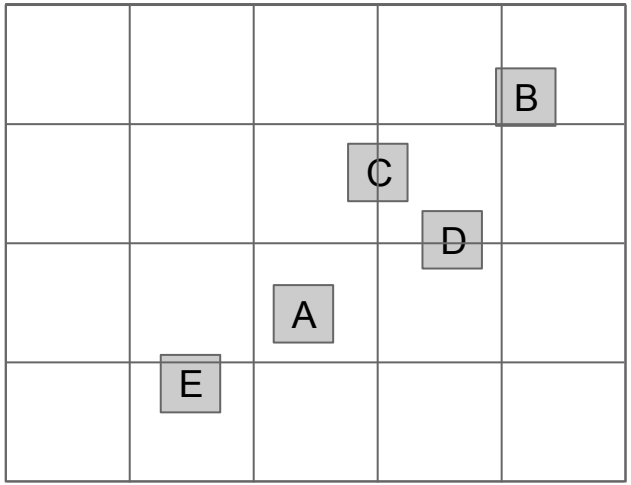




Quadtrees

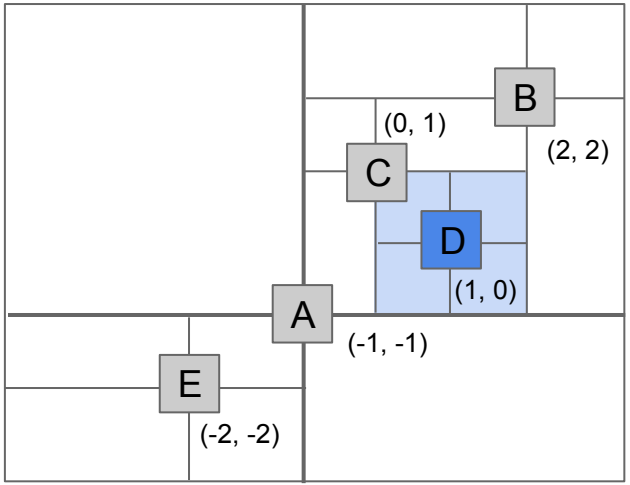
Quadtrees are also spatial partitioning in disguise.

- Earlier approach: Uniform partitioning (perfect grid of rectangles).
- Quadtrees: Hierarchical partitioning. Each node “owns” 4 subspaces.
  - Space is more finely divided in regions where there are more points.
  - Results in better runtime in many circumstances.



25 regions of uniform sizes.

D “owns” the 4 blue subspaces.

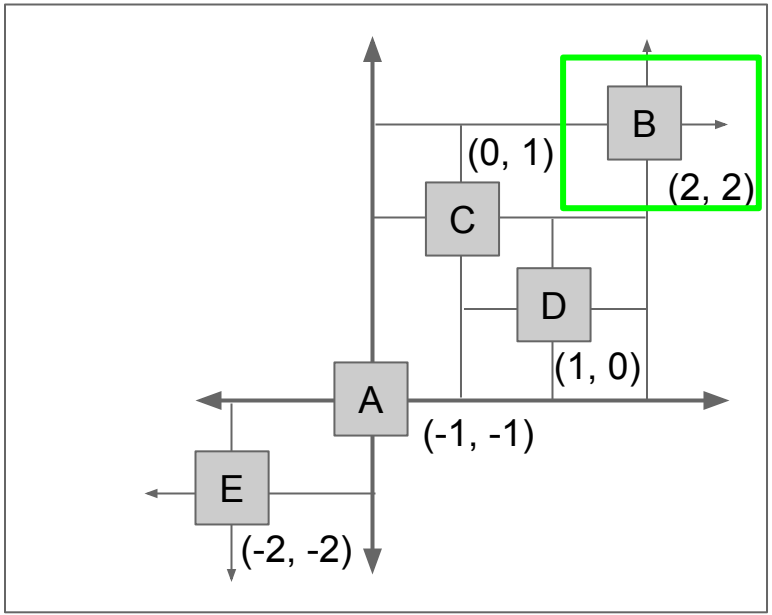


16 regions of varying sizes.

Quadtree Range Finding Demo

Quadtrees allow us to better prune when performing a rectangle search.

- Same optimization as before: Prune (i.e. don't explore) subspaces that don't intersect the query rectangle.
- Range Search Demo: Link



Only item that intersects box is B.

Higher Dimensional Data

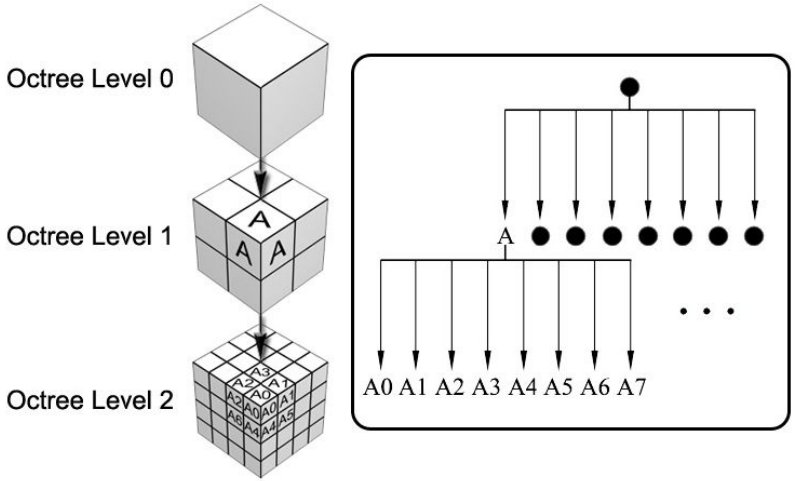
3D Data

Suppose we want to store objects in 3D space.

- Quadtrees only have four directions, but in 3D, there are 8.

One approach: Use an Oct-tree or Octree.

- Very widely used in practice.



Even Higher Dimensional Space

You may want to organize data on a large number of dimensions.

- Example: Want to find songs with the following features:
  - Length between 3 minutes and 6 minutes.
  - Between 1000 and 20,000 listens.
  - Between 120 to 150 BPM.
  - Were recorded after 2004.

In these cases, one somewhat common solution is a k-d tree.

- Fascinating data structure that handles arbitrary numbers of dimensions.
  - k-d means “k dimensional”.
- For the sake of simplicity in lecture, we’ll use 2D data, but the idea generalizes naturally.

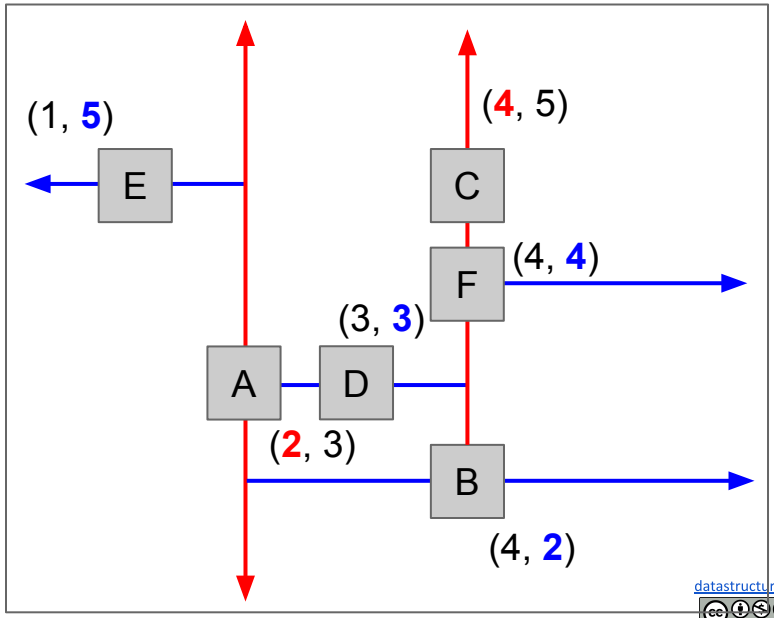
K-d Trees

k-d tree example for 2-d:

- Basic idea, root node partitions entire space into left and right (by x).
- All depth 1 nodes partition subspace into up and down (by y).
- All depth 2 nodes partition subspace into left and right (by x).

Let's see an example of how to build a k-d tree.

- K-d tree insertion demo.



K-d Trees

k-d tree example for 2-d:

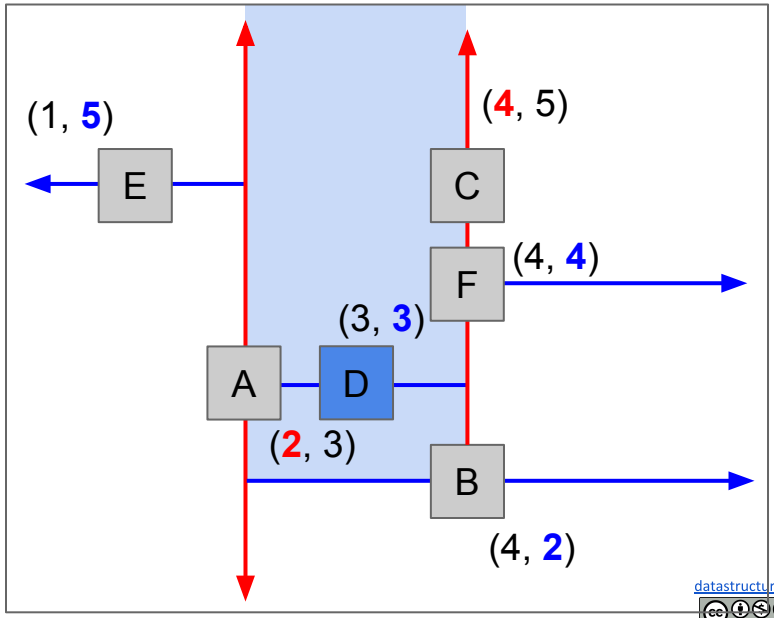
- Basic idea, root node partitions entire space into left and right (by x).
- All depth 1 nodes partition subspace into up and down (by y).
- All depth 2 nodes partition subspace into left and right (by x).

Let's see an example of how to build a k-d tree.

- K-d tree insertion demo.

Each point owns 2 subspaces.

- Similar to a quadtree.
- Example: D owns the two subspaces shown.
  - The top subspace is infinitely large.



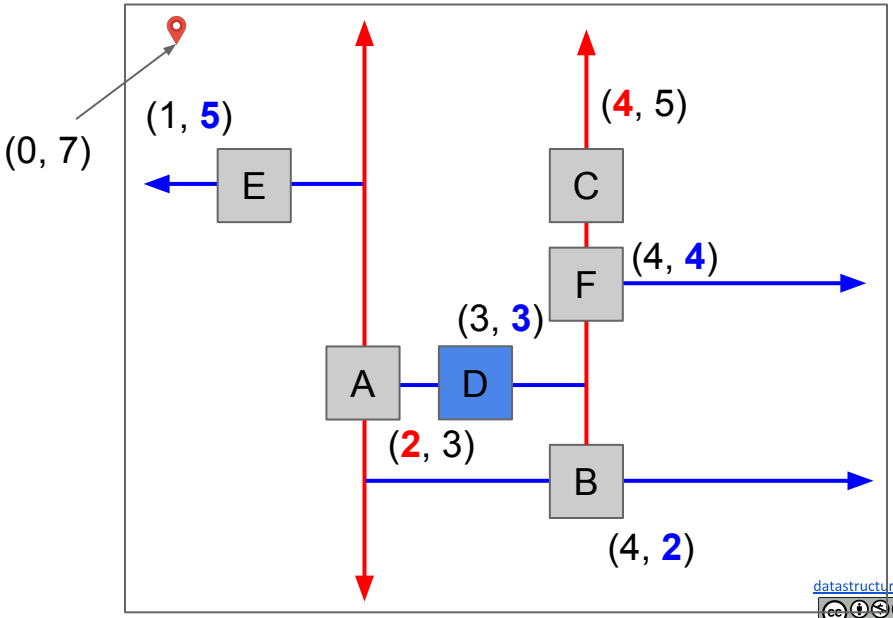
K-d Trees and Nearest Neighbor

Just like spatial partitioning and quadtrees, k-d trees support an efficient nearest method.

- Optimization: Do not explore subspaces that can't possible have a better answer than your current best.

Example: Find the nearest point to (0, 7).

- Answer is (1, 5).
- K-d tree nearest demo.



Nearest Pseudocode

nearest(Node n, Point goal, Node best):

- If n is null, return best
- If n.distance(goal) < best.distance(goal), best = n
- If goal < n (according to n's comparator):
    - goodSide = n."left"Child
    - badSide = n."right"Child
  - else:
    - goodSide = n."right"Child
    - badSide = n."left"Child
- best = nearest(goodSide, goal, best)
- If bad side could still have something useful
  - best = nearest(badSide, goal, best)
- return best

Nearest is a helper method that returns whichever is closer to goal out of the following two choices:
1. best
2. all items in the subtree starting at n

This is our pruning rule.

Inefficient Nearest Pseudocode

nearest(Node n, Point goal, Node best):

- If n is null, return best
- If n.distance(goal) < best.distance(goal), best = n
- best = nearest(n.leftChild, goal, best)
- best = nearest(n.rightChild, goal, best)
- return best

Consider implementing this inefficient version first.

- Easy to implement.
- Warning: Much slower than the NaivePointSet!
- Once you've verified this is working, you can implement the more efficent version on the previous slide.

Summary and Applications

Multi-Dimensional Data Summary

Multidimensional data has interesting operations:

- **Range Finding**: What are all the objects inside this (rectangular) subspace?
- **Nearest**: What is the closest object to a specific point?
  - Can be generalized to k-nearest.

The most common approach is **spatial partitioning**:

- **Uniform Partitioning**: Analogous to hashing.
- **Quadtree**: Generalized 2D BST where each node “owns” 4 subspaces.
- **K-d Tree**: Generalized k-d BST where each node “owns” 2 subspaces.
  - Dimension of ownership cycles with each level of depth in tree.

Spatial partitioning allows for **pruning** of the search space.

Application #1: Murmuration

You may not know this, but starlings murmurate:

- Real starlings: https://www.youtube.com/watch?v=eakKfY5aHmY
- Simulated starlings: https://www.youtube.com/watch?v=nbbd5uby0sY
- Efficient simulation means “birds” have to find their nearest neighbors efficiently.

Application #2: NBody Simulation

Your NBody simulation from Project 0 was a quadratic algorithm.

- Every object has to calculate the force from every other object.
- This is very silly if the force exerted is small, e.g. individual star in the Andromeda galaxy pulling on the sun.

One optimization is called Barnes-Hut. Basic idea:

- Represent universe as a quadtree (or octree) of objects.
- If a node is sufficiently far away from X, then treat that node and all of its children as a single object.
    - Example: Andromeda is very far away, so treat it as a single object for the purposes of force calculation on our sun.
- Visualization: https://www.youtube.com/watch?v=ouO6wmKqycc

Citations

Title figure: A thing I made (one of the first Java programs I wrote during my teaching career)

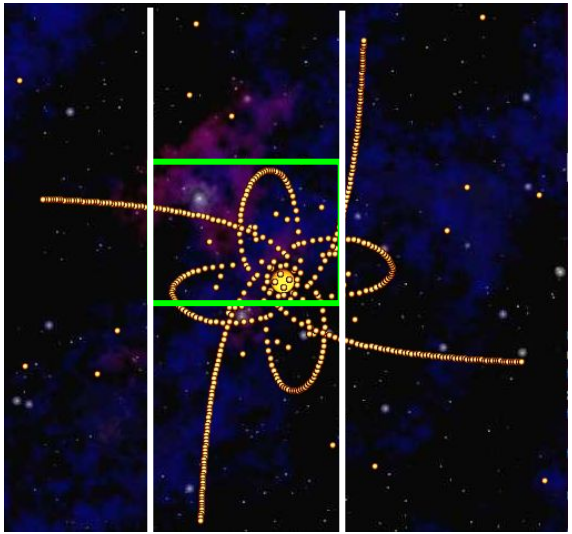
Pruning image: https://res.cloudinary.com/dc8hy36qb/image/upload/v1435213404/Fruit-Tree-Pruning-Methods_o7ieen_atkmmq.jpg

Jonathan Shewchuk: Nice intuitive use cases for various traversals.

Building Trees of Two Dimensional Data

Consider trying to build a BST of Body objects in 2D space.

- earth.xPos = 1.5, earth.yPos = 1.6
- mars.xPos = 1.0, mars.yPos = 2.8



For a BST, we need some notion of “less than”:

- In xPos, Mars < Earth (tree on left).
- In yPos, Mars > Earth (tree on right).

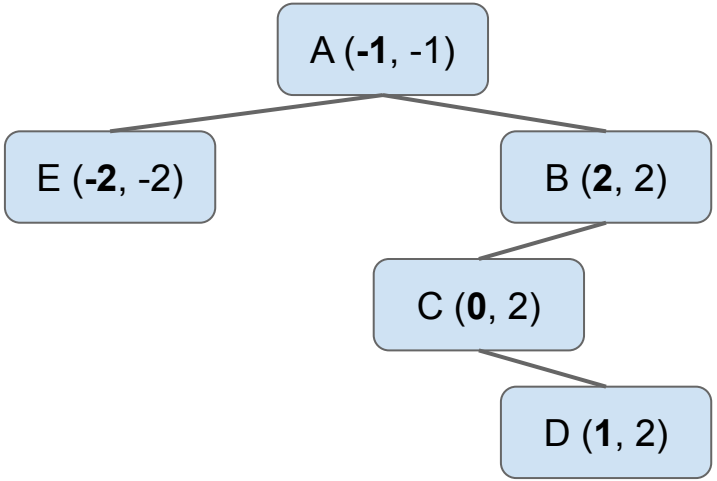
This approach does help speed up operations like 2D range search!

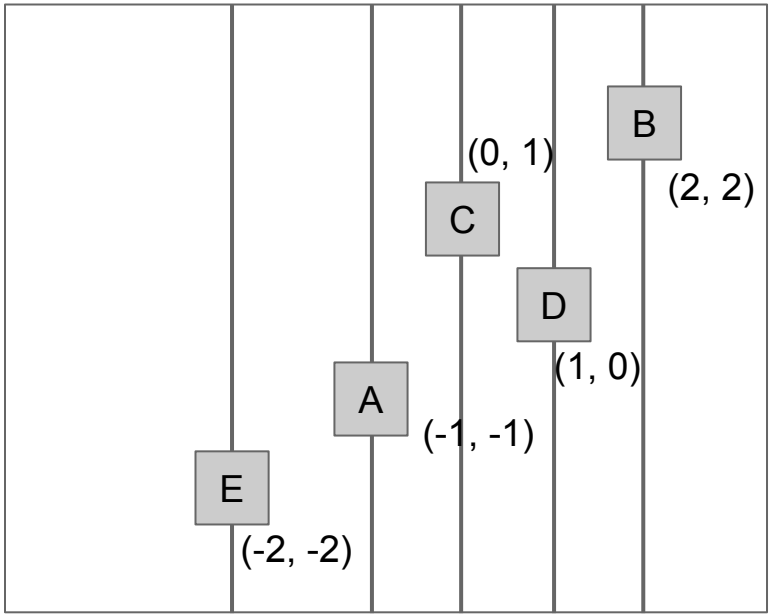
- Example, if sorted in xPos:
  - To find all points in the green range from before, effectively only need to look at points that lie between the vertical white lines.

Building Trees of Two Dimensional Data

Example: Suppose we make the following calls to a BST based map, where each Points are compared based on their x coordinate.

- `put((-1, -1), A)`
- `put(( 2,  2), B)`
- `put(( 0,  1), C)`
- `put(( 1,  0), D)`
- `put((-2, -2), E)`

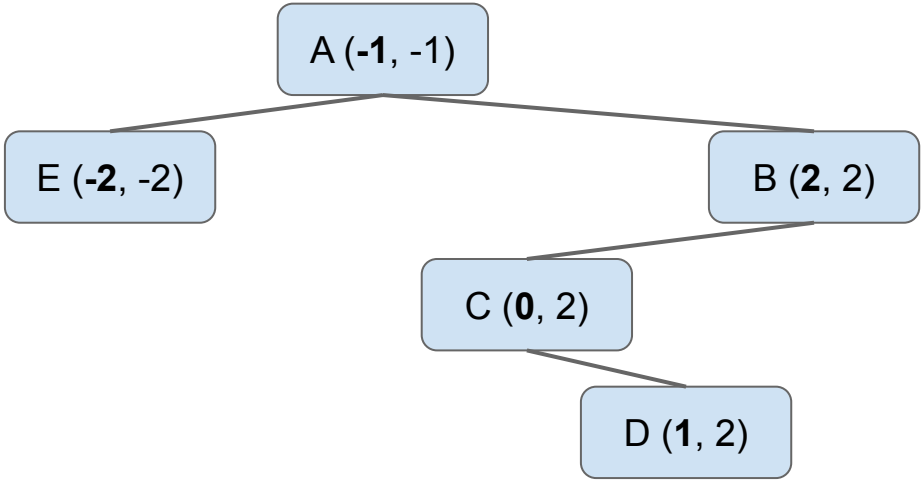


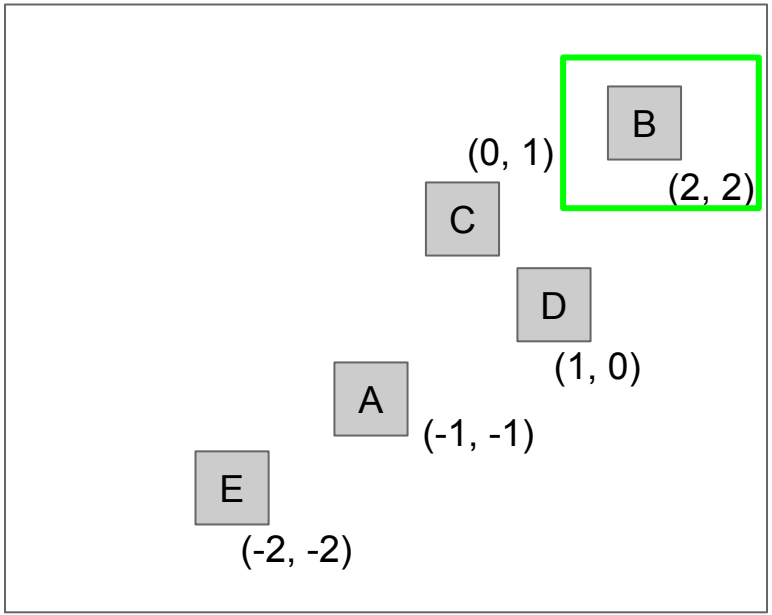


Building Trees of Two Dimensional Data

Example: Suppose we make the following calls to a BST based map, where each Points are compared based on their x coordinate.

If we now try to find objects in the rectangle





Building Trees of Two Dimensional Data

Example: Suppose we make the following calls to a BST based map, where each Points are compared based on their x coordinate.

- `put((-1, -1), A)`
- `put(( 2,  2), B)`
- `put(( 0,  1), C)`
- `put(( 1,  0), D)`
- `put((-2, -2), E)`