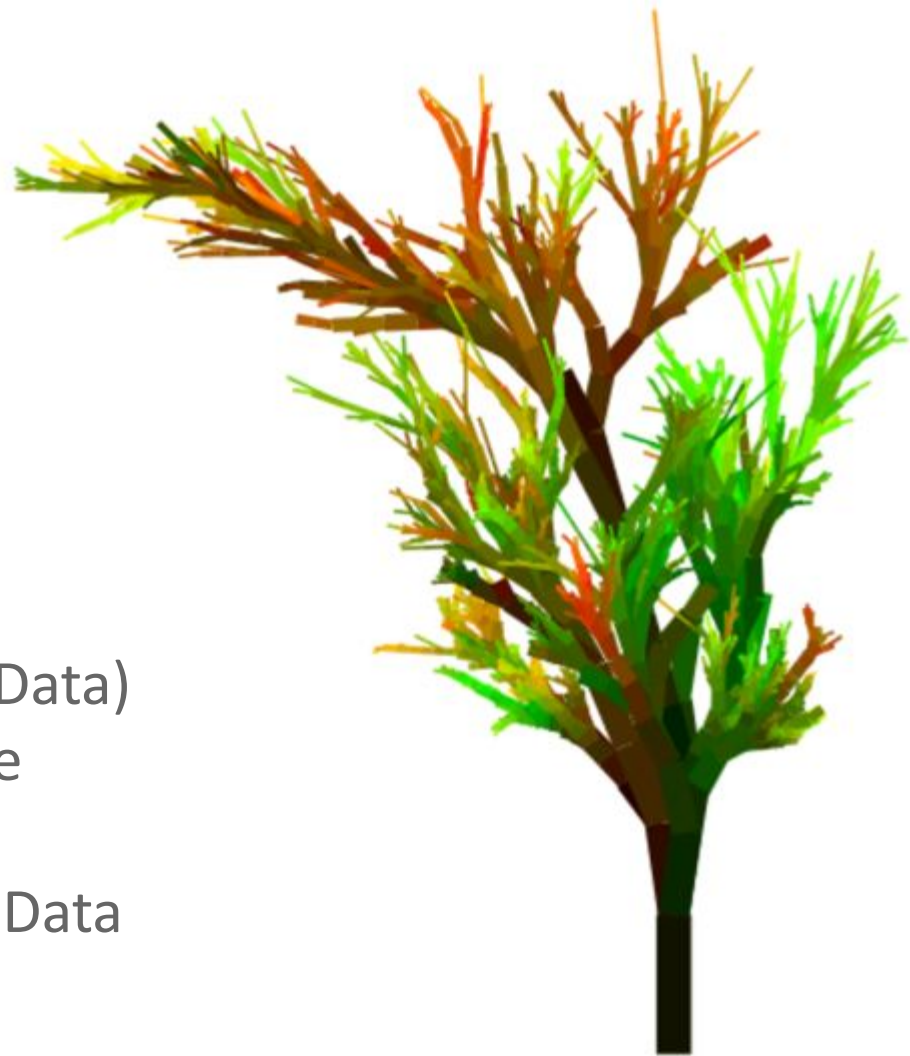


# CS61B

## Lecture 19: Multi-Dimensional Data

- Range-Finding and Nearest (1D Data)
- Multi-Dimensional Data Example
- QuadTrees for 2D Data
- KdTrees for Higher Dimensional Data
- Uniform Partitioning



# Range-Finding and Nearest

# Search Trees

---

So far we've seen three different efficient implementations of a Map (or Set):

- Binary Search Tree.
- 2-3 Tree / 2-3-4 Tree / B-Tree.
- Red Black Tree.

These “search tree” data structures support very fast insert, remove, and delete operations for arbitrary amounts of data.

- Requires that data can be compared to each other with some total order.
- We used the “Comparable” interface as our comparison engine.

# Expanding the Power of our Set

---

There are other operations we might want to include in a Set.

For example, we might add a `select` operation:

- `select(int i)`: Returns the *i*th smallest item in the set.
  - `select(0)`: 1
  - `select(3)`: 6

`{1, 4, 5, 6, 9, 11, 14, 17, 20}`

# Expanding the Power of our Set

---

There are other operations we might want to include in a Set.

For example, we might add a rank operation:

- $\text{rank}(T, x)$ : Returns the “rank” of  $x$  in the set (opposite of select).
  - $\text{rank}(1)$ : 0
  - $\text{rank}(6)$ : 3

$\{1, 4, 5, 6, 9, 11, 14, 17, 20\}$

# Expanding the Power of our Set

---

There are other operations we might want to include in a Set.

For example, we might add a subSet operation:

- `subSet(T from, T to)`: Returns all items between from and to.
  - `subSet(4, 9)`: Returns {4, 5, 6, 9}.
  - `subSet(3, 12)`: Returns {4, 5, 6, 9, 11}.
  - `subSet(12, 13)`: Returns {}.

{1, 4, 5, 6, 9, 11, 14, 17, 20}

# Expanding the Power of our Set

---

There are other operations we might want to include in a Set.

If we have a notion of distance between items, we might add a nearest operation:

- `nearest(T x)`: Returns the value closest to `x`.
  - `nearest(6)`: Returns 6.
  - `nearest(8)`: Returns 9.
  - `nearest(10)`: Returns 9 or 11.

`{1, 4, 5, 6, 9, 11, 14, 17, 20}`

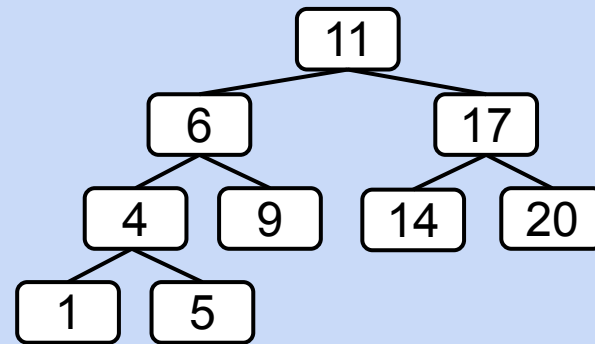
# Implementing Fancier Set Operations with a BST

It turns out that a BST can efficiently support the `select`, `rank`, `subSet`, and `nearest` operations.

`nearest` returns the item that is closest. Examples:

- `nearest(6)`: Returns 6.
- `nearest(8)`: Returns 9.
- `nearest(10)`: Returns 9 or 11.

Challenge: How would you find `nearest(N)`?





# Implementing Fancier Set Operations with a BST

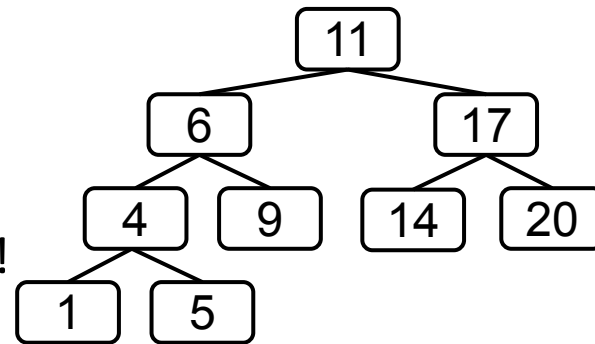
It turns out that a BST can efficiently support the `select`, `rank`, `subSet`, and `nearest` operations.

`nearest` returns the item that is closest. Examples:

- `nearest(6)`: Returns 6.
- `nearest(8)`: Returns 9.
- `nearest(10)`: Returns 9 or 11.

Challenge: How would you find `nearest(N)`?

- Just search for `N` and record closest item seen!



# Implementing Fancier Set Operations with a BST

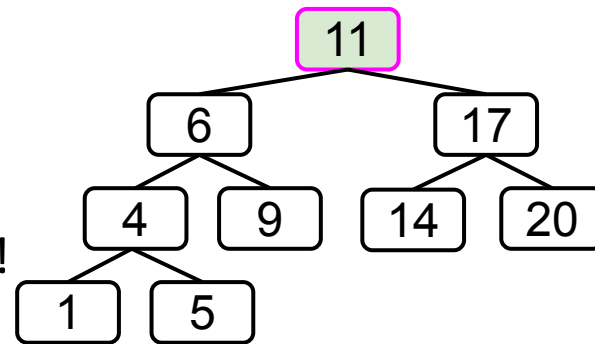
It turns out that a BST can efficiently support the `select`, `rank`, `subSet`, and `nearest` operations.

`nearest` returns the item that is closest. Examples:

- `nearest(6)`: Returns 6.
- `nearest(8)`: Returns 9.
- `nearest(10)`: Returns 9 or 11.

Challenge: How would you find `nearest(N)`?

- Just search for `N` and record closest item seen!
- Example: `nearest(5.4)`: 11 is best seen so far.



# Implementing Fancier Set Operations with a BST

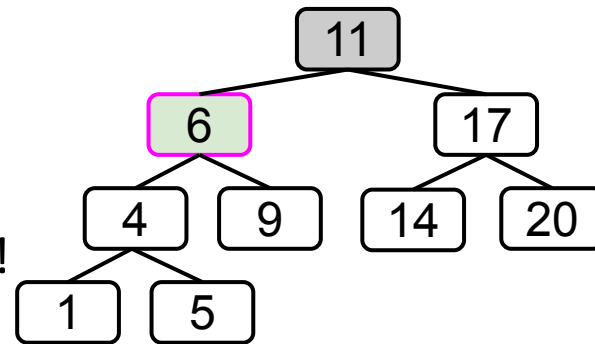
It turns out that a BST can efficiently support the `select`, `rank`, `subSet`, and `nearest` operations.

`nearest` returns the item that is closest. Examples:

- `nearest(6)`: Returns 6.
- `nearest(8)`: Returns 9.
- `nearest(10)`: Returns 9 or 11.

Challenge: How would you find `nearest(N)`?

- Just search for `N` and record closest item seen!
- Example: `nearest(5.4)`: 6 is best seen so far.



# Implementing Fancier Set Operations with a BST

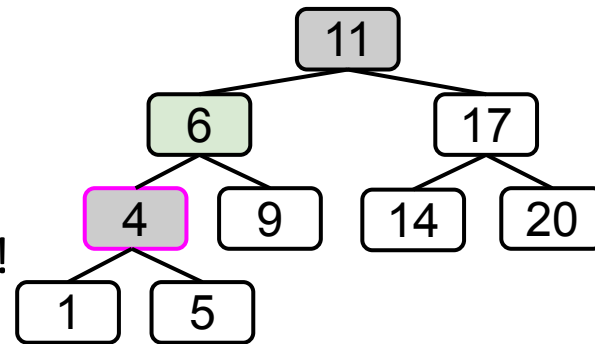
It turns out that a BST can efficiently support the `select`, `rank`, `subSet`, and `nearest` operations.

`nearest` returns the item that is closest. Examples:

- `nearest(6)`: Returns 6.
- `nearest(8)`: Returns 9.
- `nearest(10)`: Returns 9 or 11.

Challenge: How would you find `nearest(N)`?

- Just search for `N` and record closest item seen!
- Example: `nearest(5.4)`: 6 is still best so far.



# Implementing Fancier Set Operations with a BST

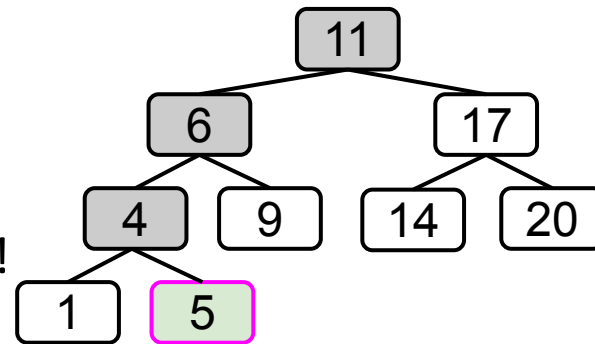
It turns out that a BST can efficiently support the `select`, `rank`, `subSet`, and `nearest` operations.

`nearest` returns the item that is closest. Examples:

- `nearest(6)`: Returns 6.
- `nearest(8)`: Returns 9.
- `nearest(10)`: Returns 9 or 11.

Challenge: How would you find `nearest(N)`?

- Just search for `N` and record closest item seen!
- Example: **`nearest(5.4)`: 5 is best.**



Exploiting BST structure took less time than looking at all values.

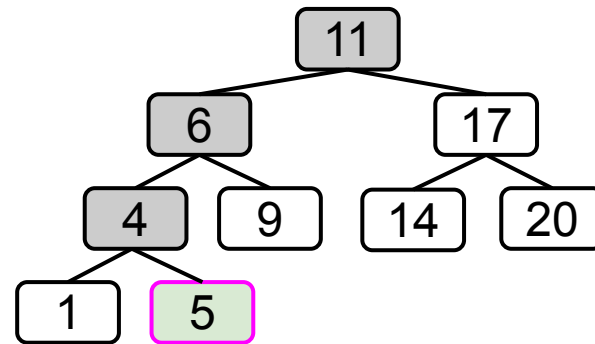
# Implementing Fancier Set Operations with a BST

---

It turns out that a BST can efficiently support the `select`, `rank`, `subSet`, and `nearest` operations.

Similar ideas can be used for `select`, `rank`, `subSet`, and many more.

- Won't discuss in lecture.



# Sets and Maps on 2D Data

---

So far we've only discussed "one dimensional data". That is, all data could be compared under some total order. Examples:

- $1 < 3 < 6 < 9 < 15$
- "cat" < "doghouse" < "if" < "zebra" [using alphabetical order]
- "if" < "cat" < "zebra" < "doghouse" [using string length]

But not all data can be compared along a single dimension.

- We'll see that search trees require some design tweaks to function efficiently on multi-dimensional data.

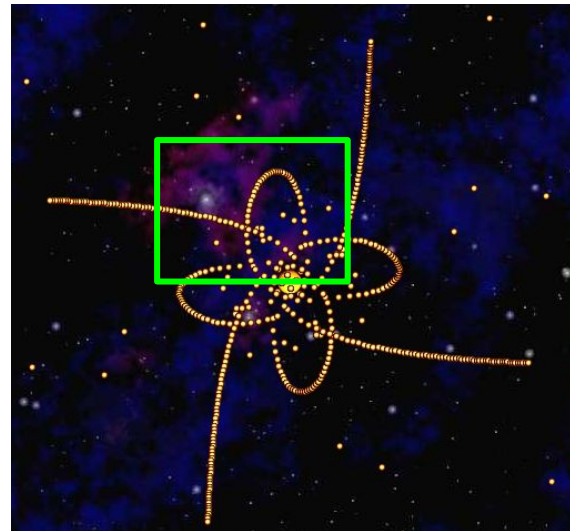
# Multi-dimensional Data



# Motivation: 2D Range Finding and Nearest Neighbors

---

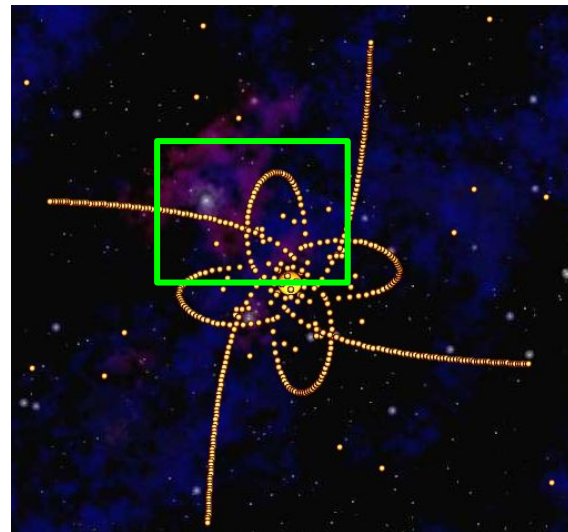
Suppose we want to perform operations on a **set** of Body objects in 2D space?



# Motivation: 2D Range Finding and Nearest Neighbors

Suppose we want to perform operations on a **set** of Body objects in 2D space?


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



Note: 2D range search is a generalization of the “subSet” method from earlier.

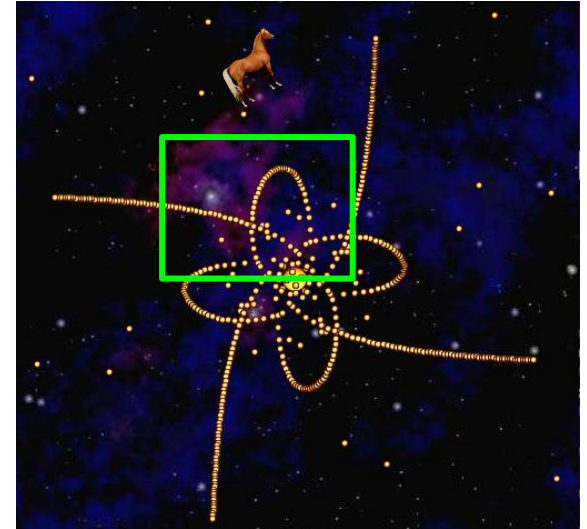
# Motivation: 2D Range Finding and Nearest Neighbors

Suppose we want to perform operations on a **set** of Body objects in 2D space?

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

Ideally, we'd like to store our data in a format (like a BST) that allows more efficient approaches than just iterating over all objects.

Let's see what goes wrong if we try to build a BST of 2 dimensional data.



Note: 2D range search is a generalization of the “subSet” method from earlier.

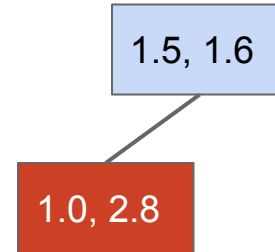
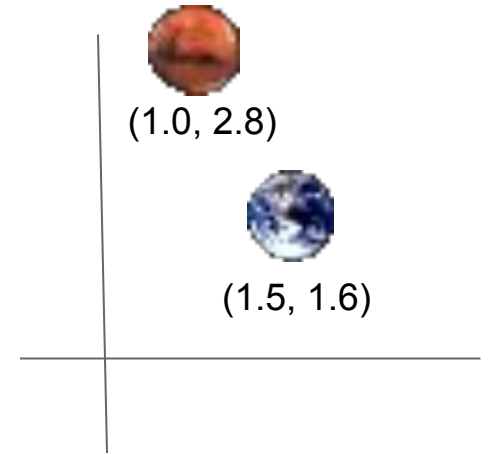
# Building Trees of Two Dimensional Data

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

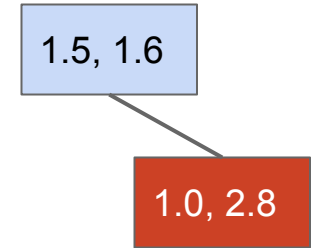
- `earth.xPos = 1.5`, `earth.yPos = 1.6`
- `mars.xPos = 1.0`, `mars.yPos = 2.8`

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

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



X-Based Tree



Y-Based Tree

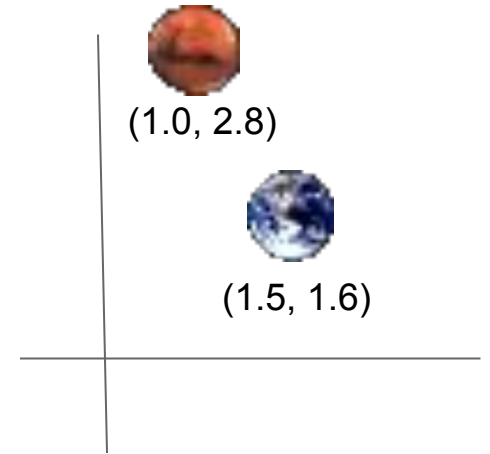
# Building Trees of Two Dimensional Data

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

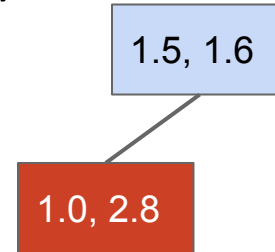
For a BST, we need some notion of “less than”:

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

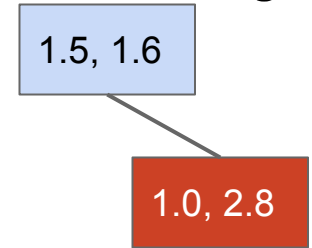


Could just pick one, but you're losing some of your information about ordering.

- Will make nearest/2D range finding slow.



X-Based Tree



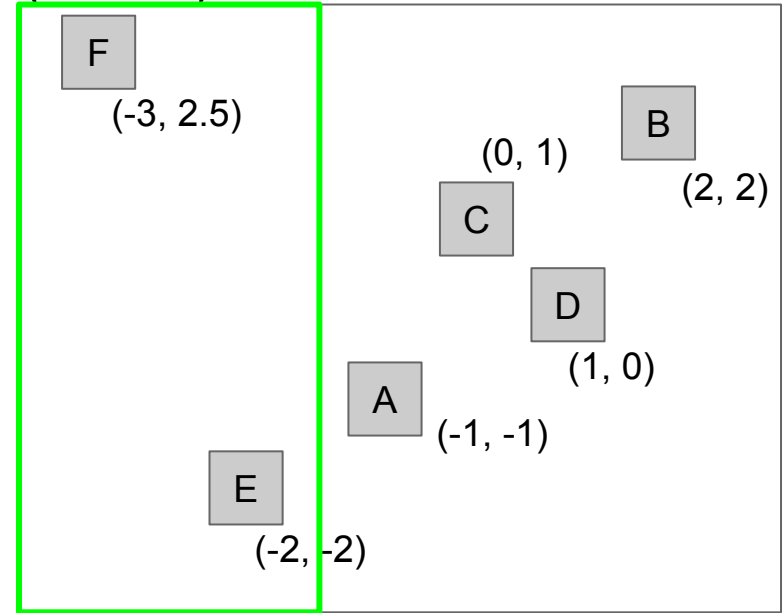
Y-Based Tree

## Example of a 2D Range-Finding Query

Suppose we want to know “What are all the points with x-coordinate less than -1.5?”

- Equivalent to “What’s in the green rectangle?”

As humans, we can easily visually identify them (E and F).

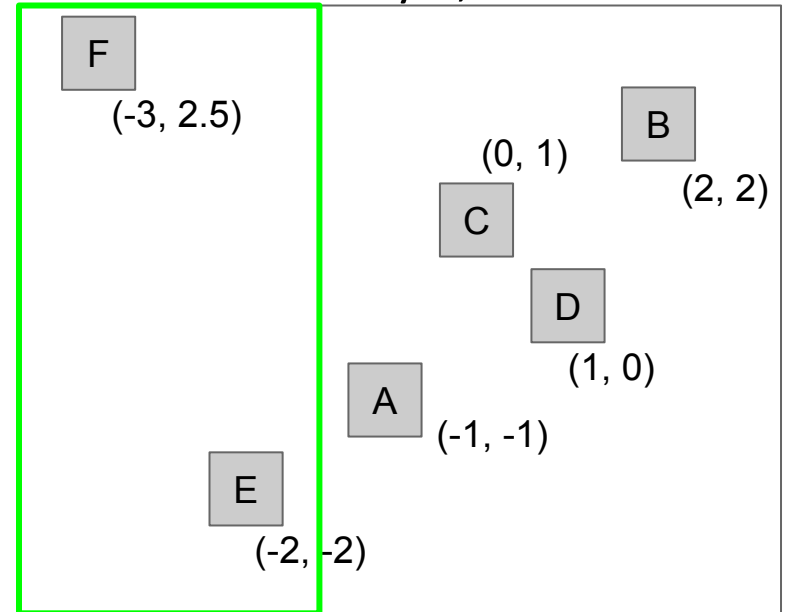


## Example of a 2D Range-Finding Query

If we have the points stored in a list, then we'd have to iterate over the entire list.

- Slow! Requires looking at every single point.

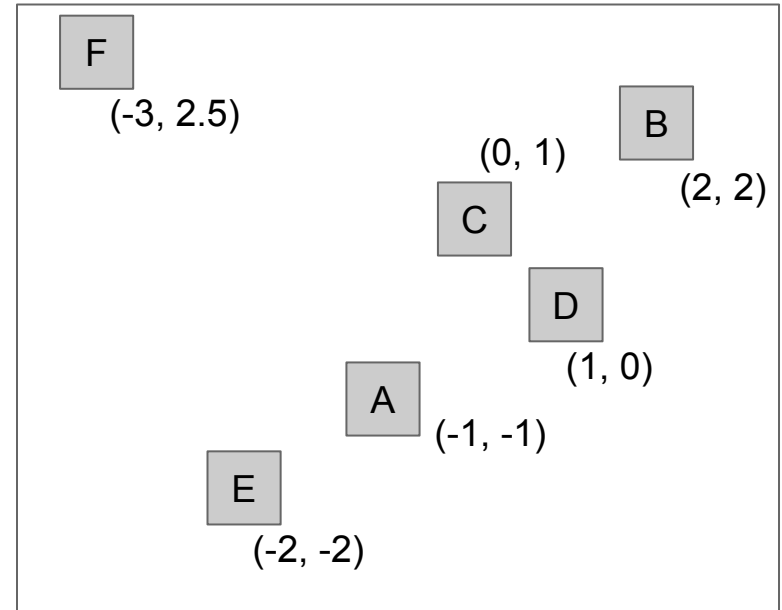
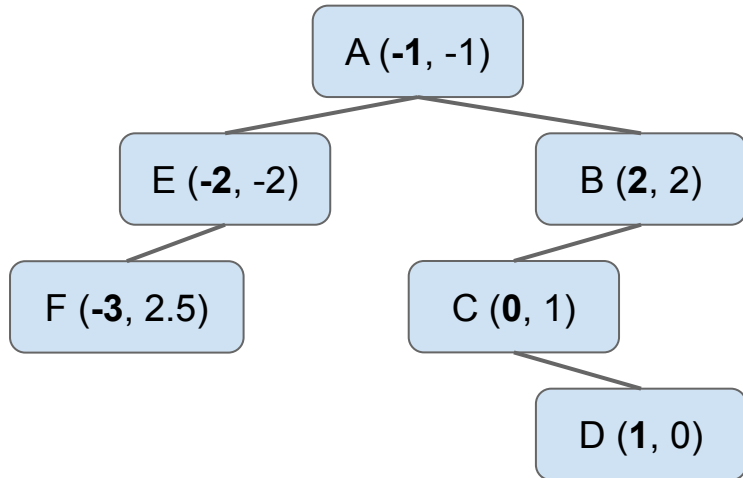
Let's see what happens if we try to store them in a BST ordered by X, then Y coordinate.



# 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)`

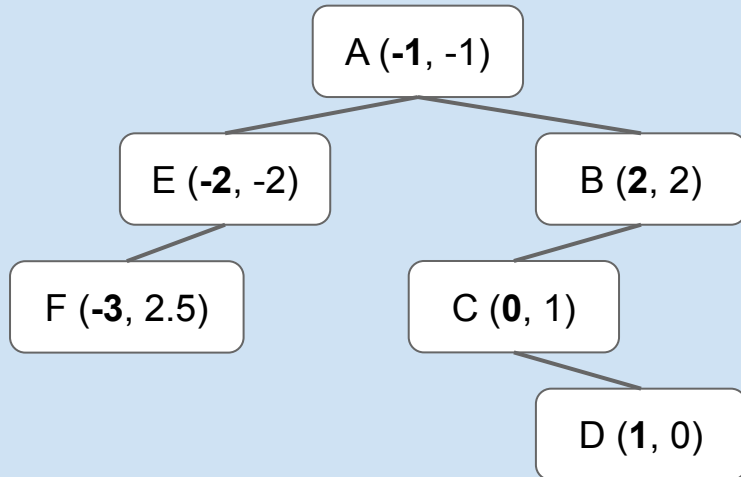




# Building Trees of Two Dimensional Data

Challenge: Given the BST below, identify all the points with x-coordinate less than -1.5.

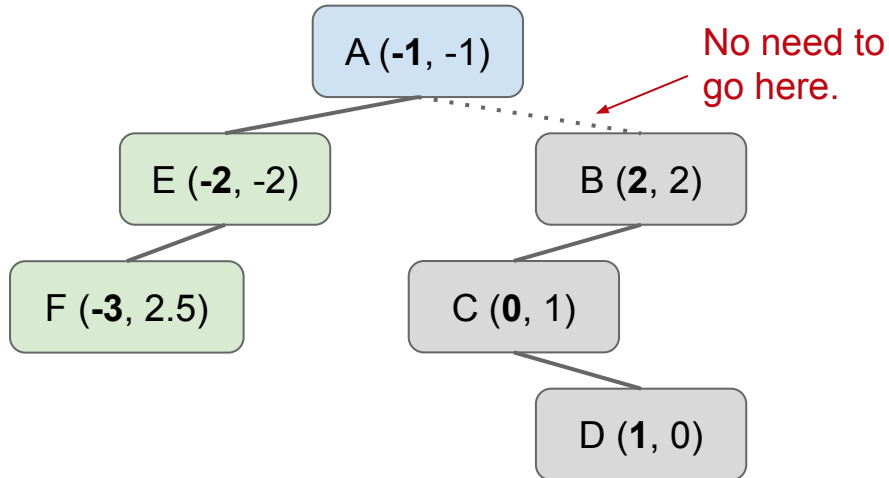
How can you avoid looking at every node?



# Building Trees of Two Dimensional Data

Challenge: Given the BST below, identify all the points with x-coordinate less than -1.5.

- We can simply work our way down the tree, ignoring pursuing any options that are impossible.
- For example, no need to go down the right child of A because no point over there could possible have x coordinate  $< -1.5$ .

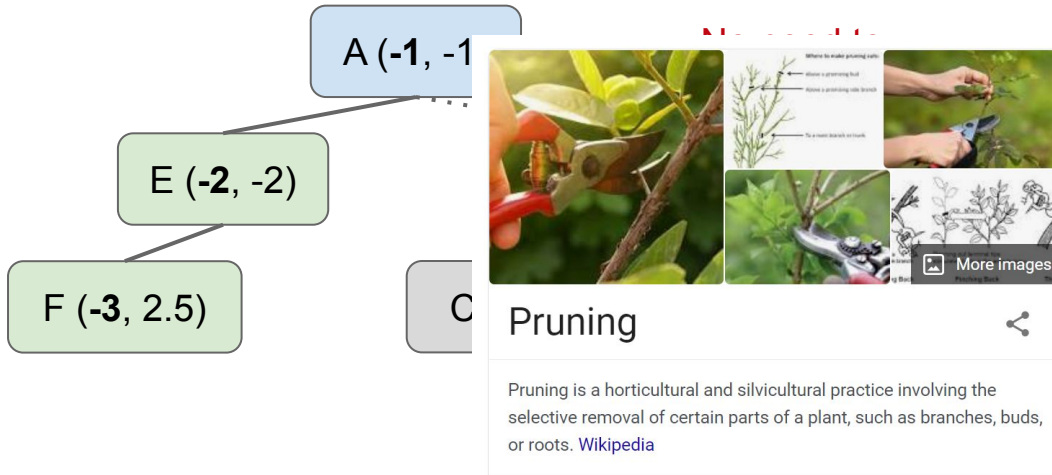


This process of cutting off a tree search early is called “Pruning”.

# Building Trees of Two Dimensional Data

Challenge: Given the BST below, identify all the points with x-coordinate less than -1.5.

- We can simply work our way down the tree, ignoring pursuing any options that are impossible.
- For example, no need to go down the right child of A because no point over there could possible have x coordinate  $< -1.5$ .

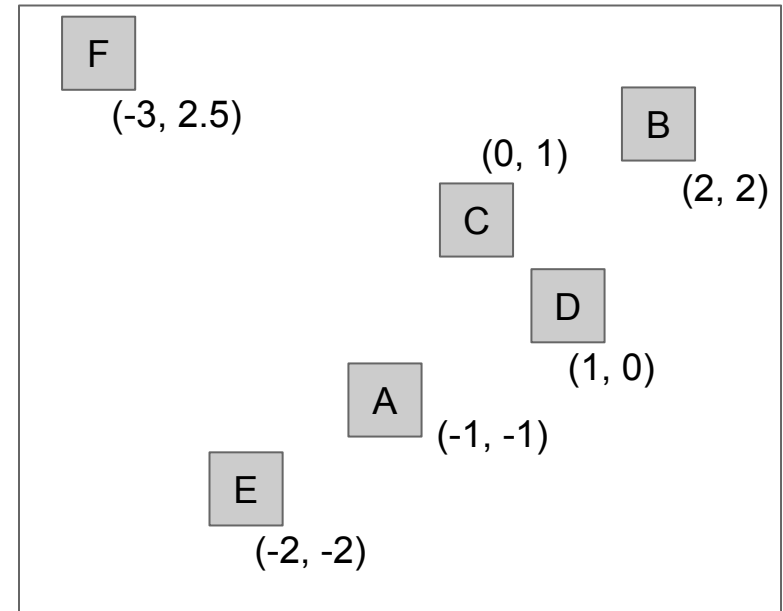
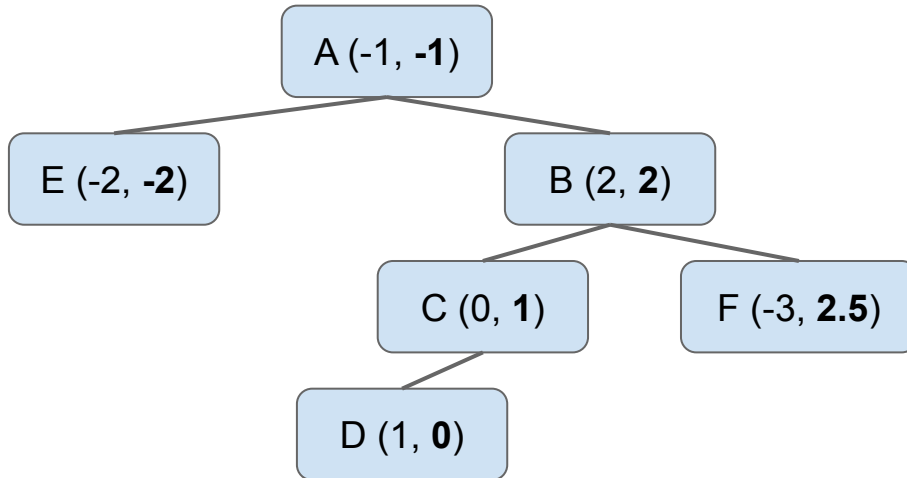


This process of cutting off a tree search early is called “Pruning”.

# Building Trees of Two Dimensional Data

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

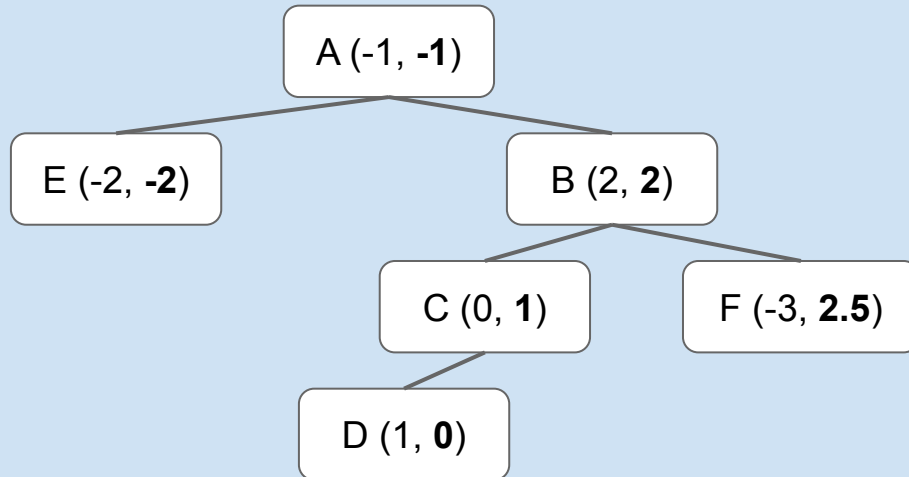
- `put((-1, -1), A)`
- `put(( 2, 2), B)`
- `put(( 0, 1), C)`
- `put(( 1, 0), D)`
- `put((-2, -2), E)`
- `put((-3, 2.5), F)`



# Building Trees of Two Dimensional Data

Challenge: Given the BST below, identify all the points with x-coordinate less than -1.5.

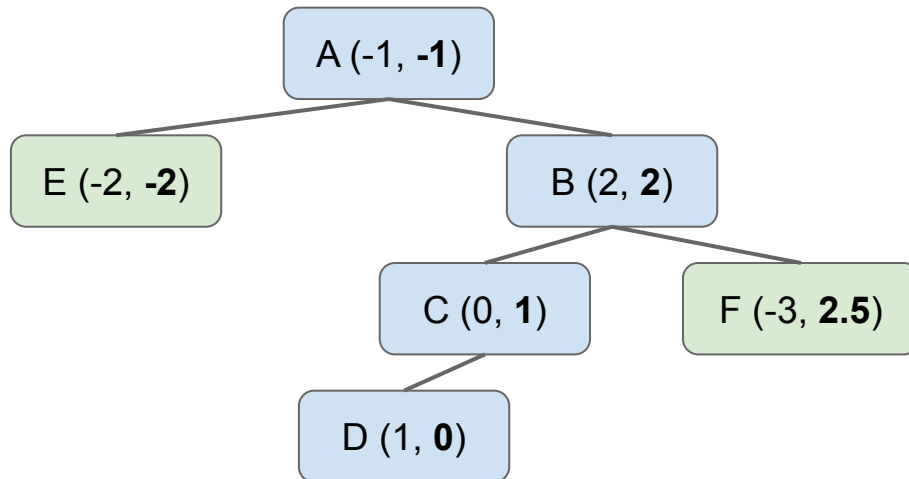
How can you avoid looking at every node?



# Building Trees of Two Dimensional Data

Challenge: Given the BST below, identify all the points with x-coordinate less than -1.5.

In this case, pruning was impossible!

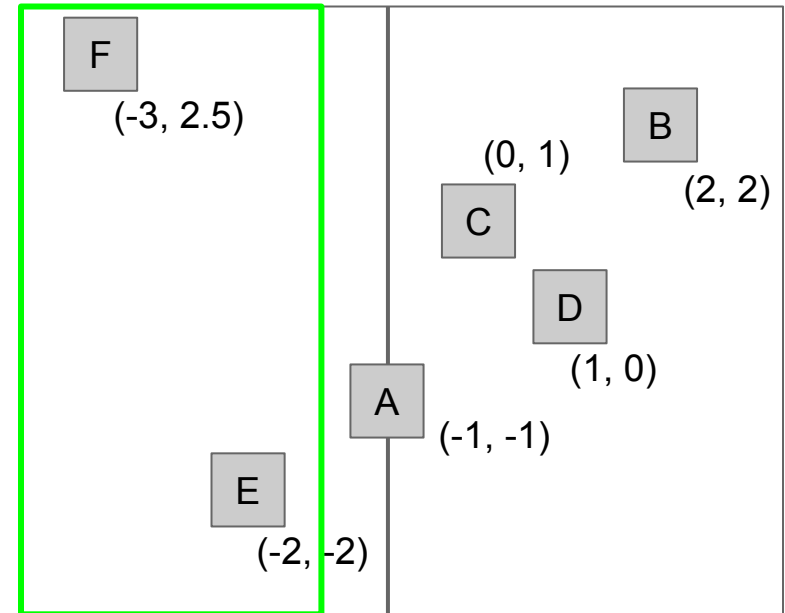
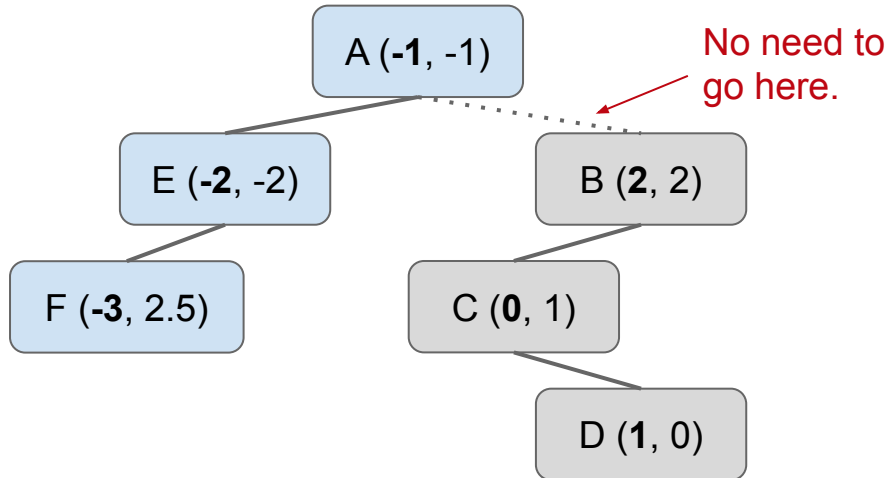


# Spatial Partitioning / Rectangle Intersection Interpretation

In the X-oriented tree, the “A” node partitioned the universe into a left and right side.

- Left: All points with  $X < -1$ .
- Right: All points with  $X > -1$ .

Right side doesn't intersect query rectangle.

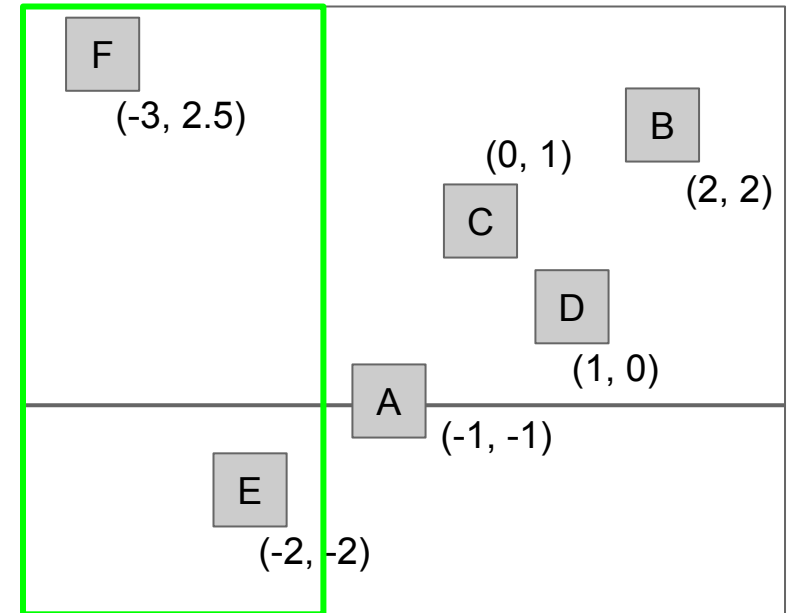
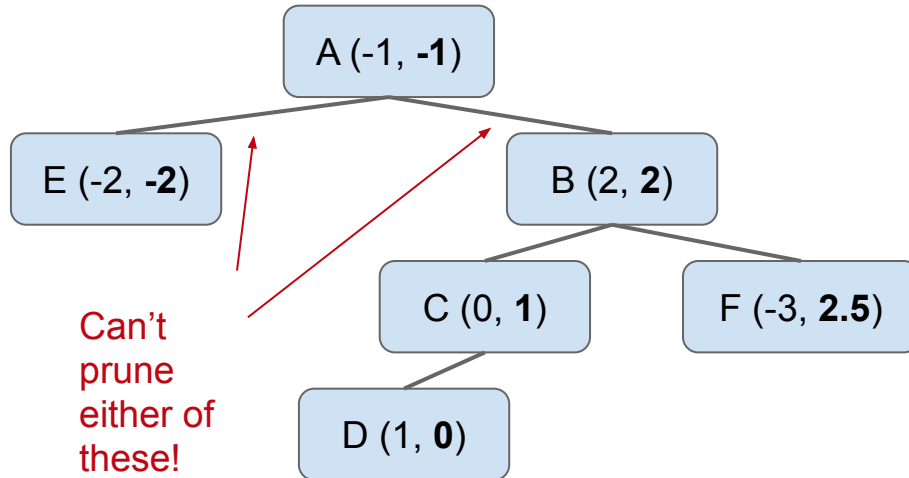


# Spatial Partitioning / Rectangle Intersection Interpretation

In the Y-oriented tree, the “A” node partitioned the universe into a top and bottom side.

- Bottom (left): All points with  $Y < -1$ .
- Top (right): All points with  $Y > -1$ .

Both sides intersect query rectangle.



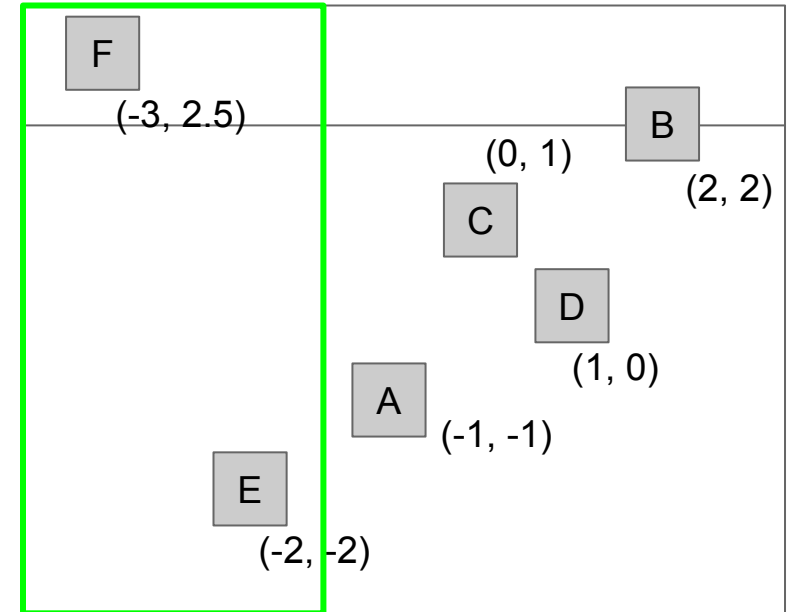
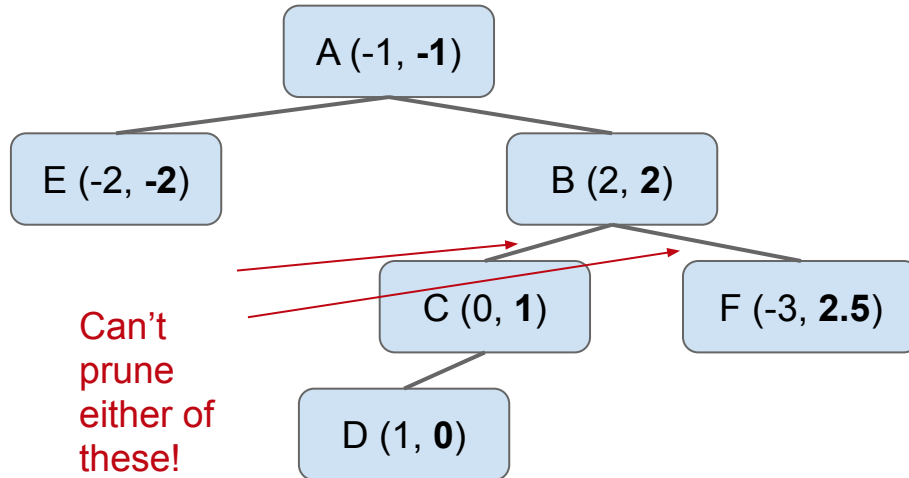


# Spatial Partitioning / Rectangle Intersection Interpretation

In the Y-oriented tree, the “A” node partitioned the universe into a top and bottom side.

- Bottom (left): All points with  $Y < -1$ .
- Top (right): All points with  $Y > -1$ .

Both sides intersect query rectangle.



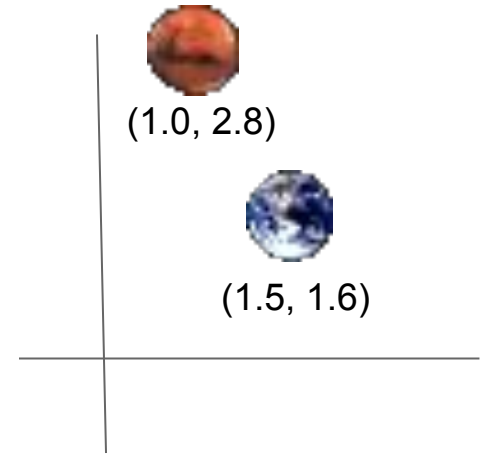
# Building Trees of Two Dimensional Data

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.

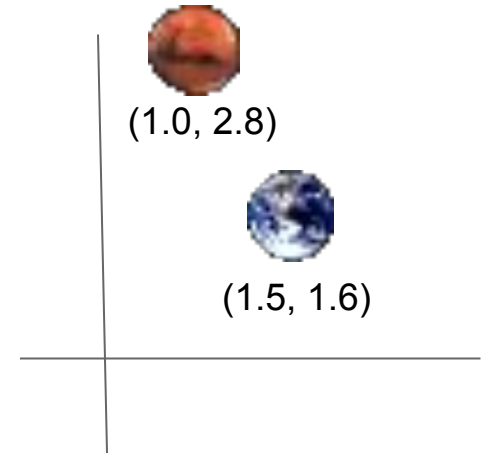
Coming up next, we'll see two efficient approaches to storing 2D data in a tree.

# QuadTrees

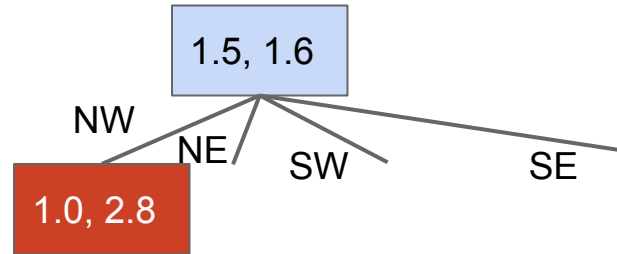
# The QuadTree

A QuadTree is the simplest solution conceptually.

- Every Node has **four** children:
  - Top left, a.k.a. northwest.
  - Top right, a.k.a. northeast.
  - Bottom left, a.k.a. southwest.
  - Bottom right, a.k.a. southeast.



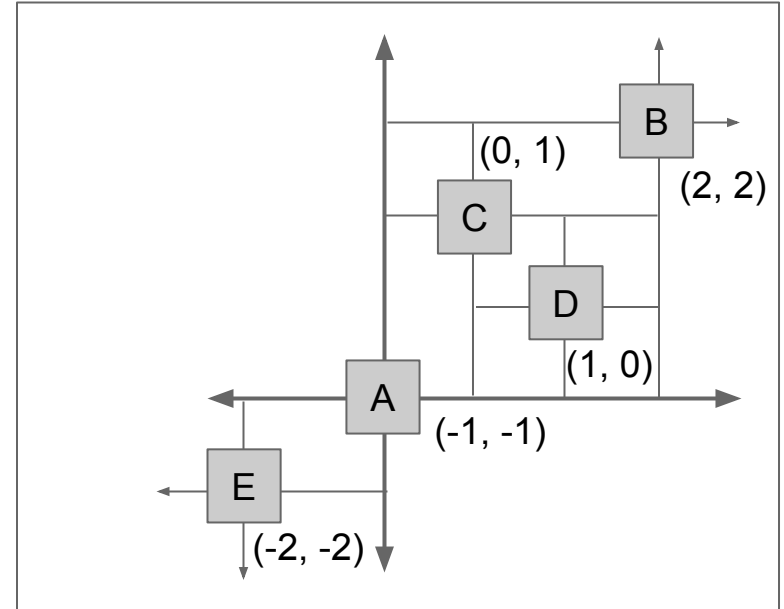
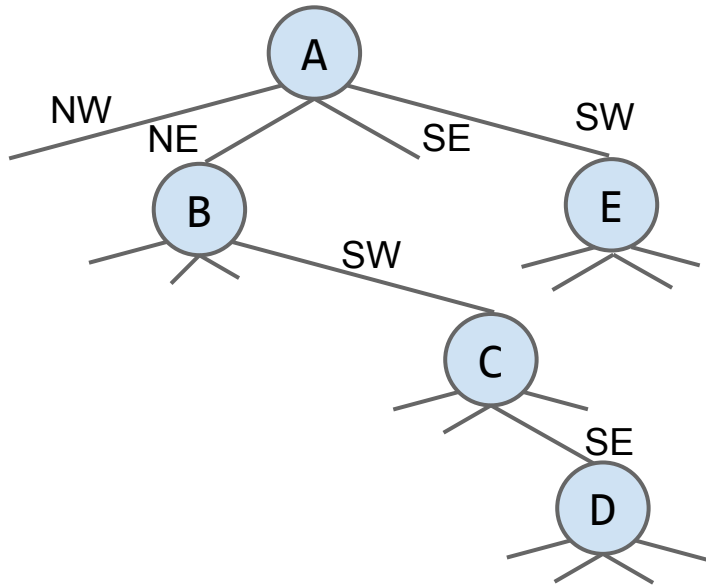
So if we insert Earth, then Mars, we have the unique tree below:



# QuadTree Insertion Demo

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

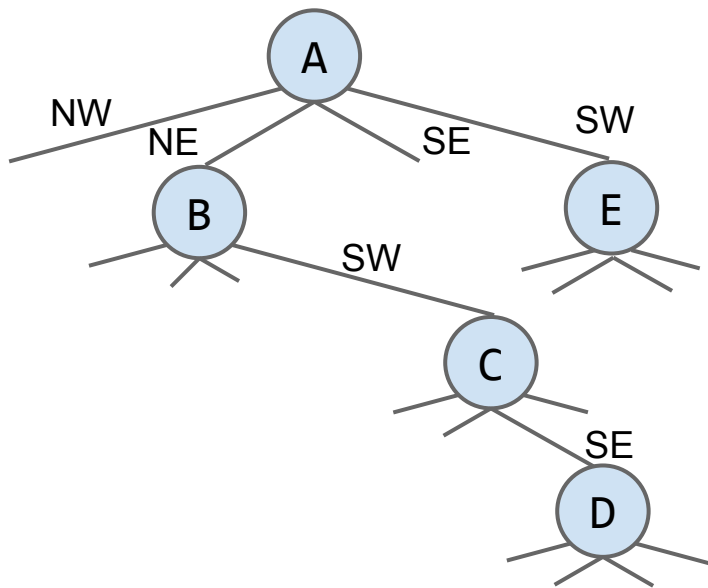
- Insertion Demo: [Link](#)



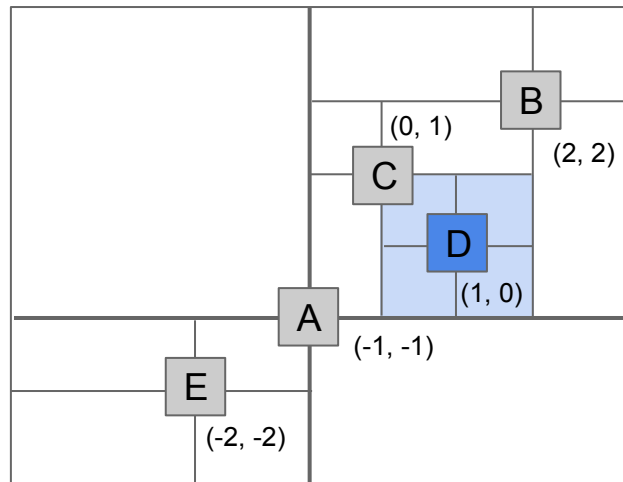
# QuadTrees

Quadtrees are a form of “spatial partitioning”.

- Quadtrees: Each node “owns” 4 subspaces.
  - Space is more finely divided in regions where there are more points.
  - Results in better runtime in many circumstances.



D “owns”  
the 4 blue  
subspaces.

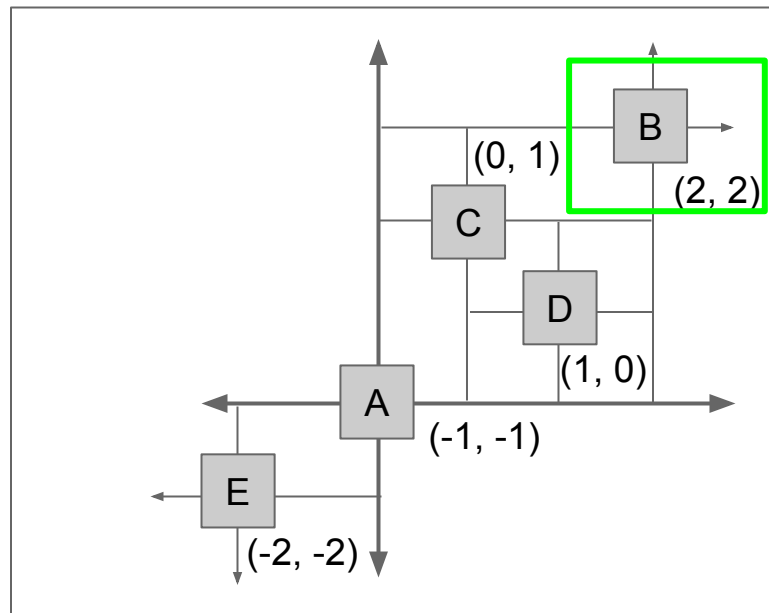
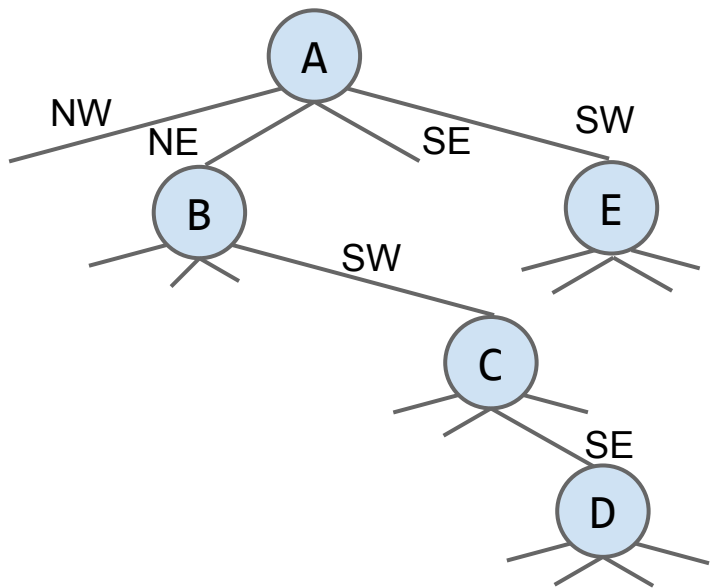


16 subspaces of varying sizes.

# QuadTree Range Search Demo

Quadtrees allow us to prune when performing a rectangle search.

- Simple idea: Prune (i.e. don't explore) subspaces that don't intersect the query rectangle.
- Range Search Demo: [Link](#)

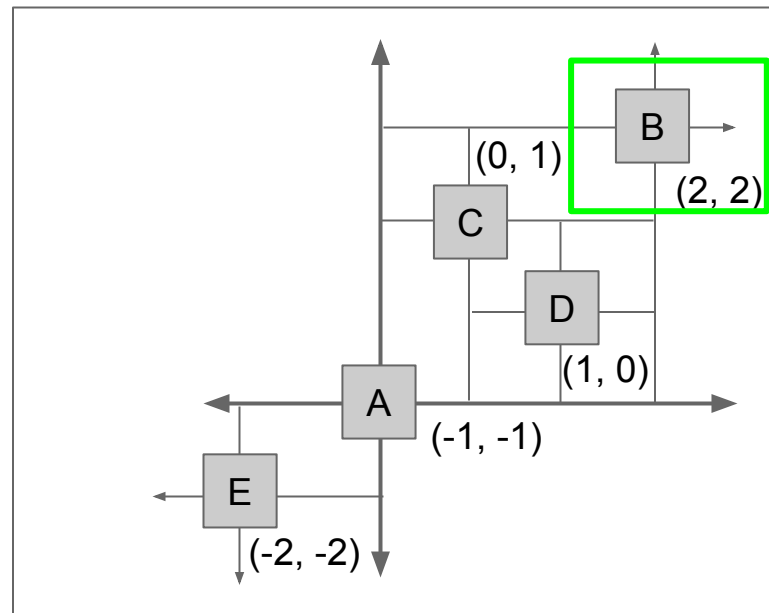
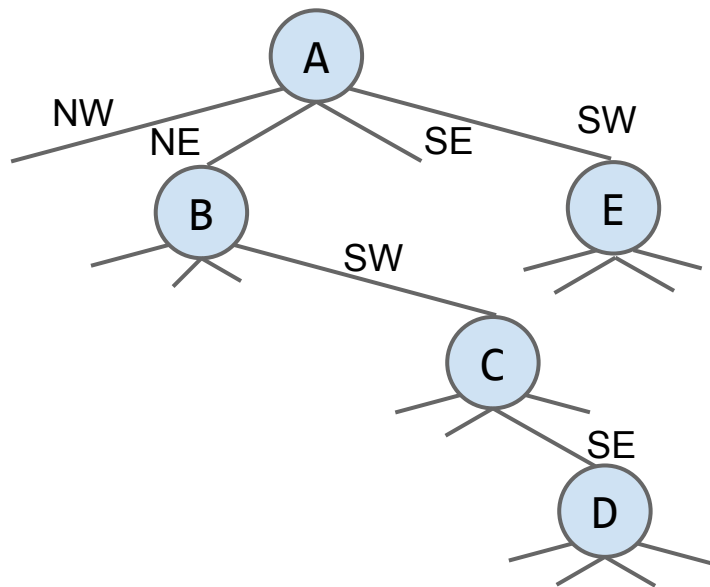


Only item that in the rectangle is B.

# QuadTree Range Search Demo

Quadtrees allow us to prune when performing a rectangle search.

- Simple idea: Prune (i.e. don't explore) subspaces that don't intersect the query rectangle.
- Range Search Demo: [Link](#)



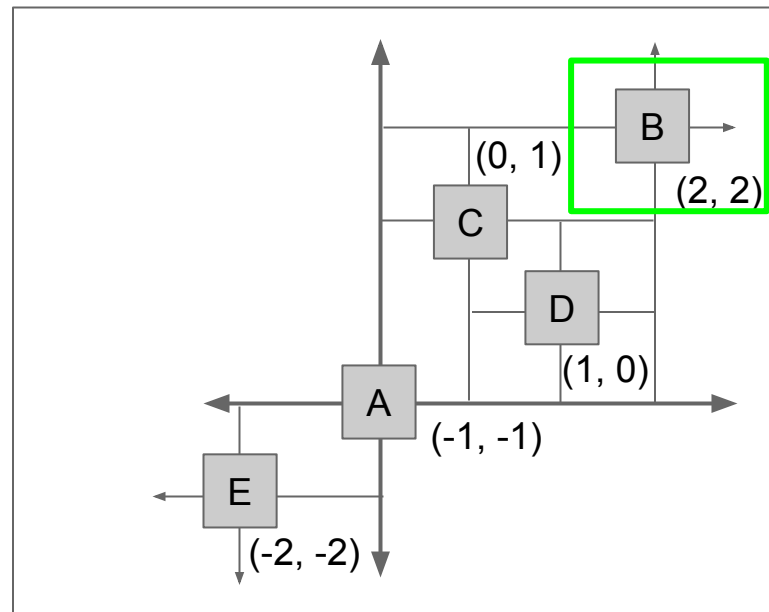
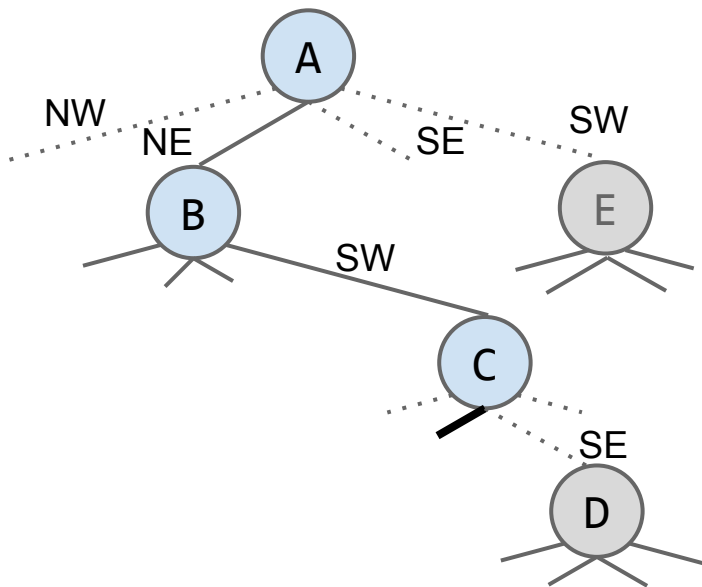
Only item that in the rectangle is B.



# QuadTree Range Search Demo

Quadtrees allow us to prune when performing a rectangle search.

- For the example below, we were able to prune all of the dotted links.



results = [B]

# Higher Dimensional Data

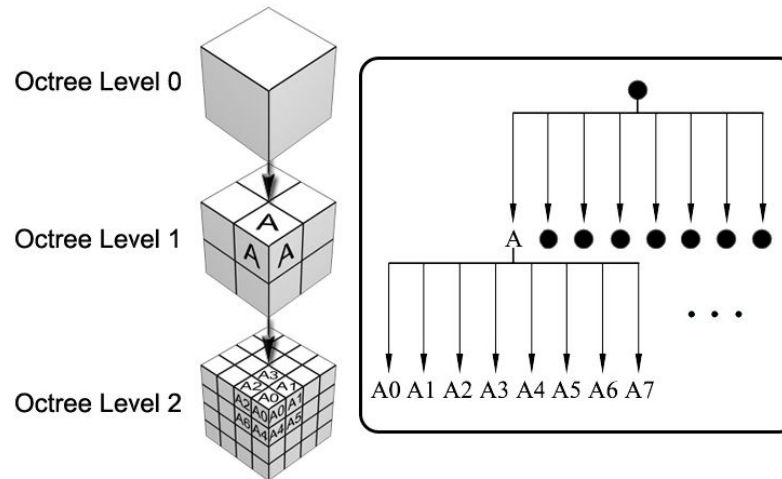
# 3D Data

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.

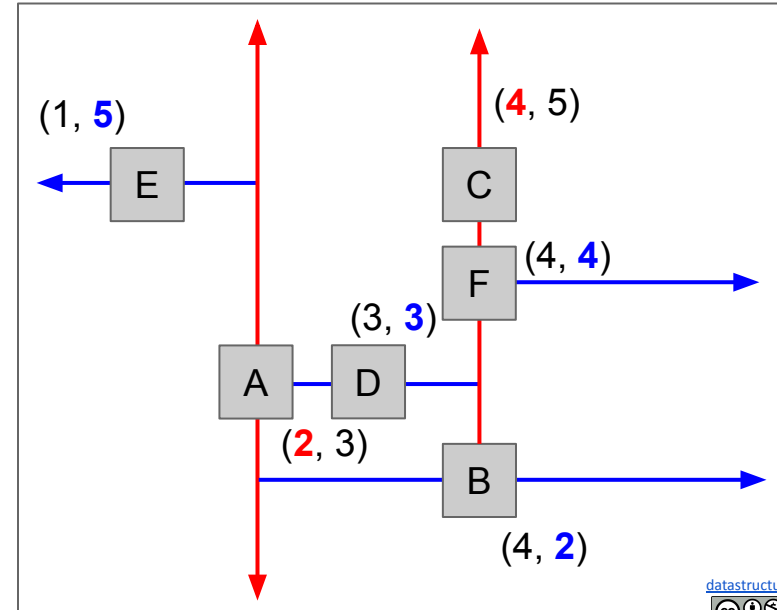
# K-d Trees

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:

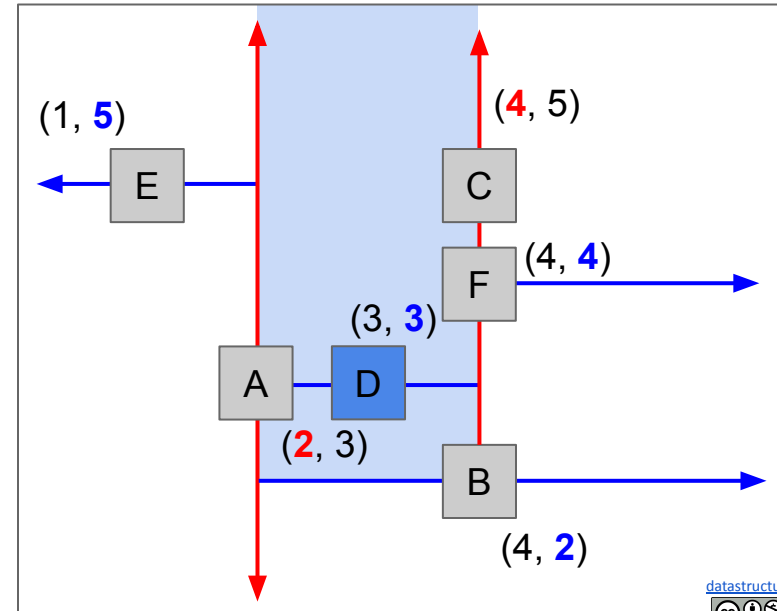
- Basic idea, root node partitions entire space into left and right (by x).
- All depth 1 nodes partition subspace into up and down (by y).
- All depth 2 nodes partition subspace into left and right (by x).

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.



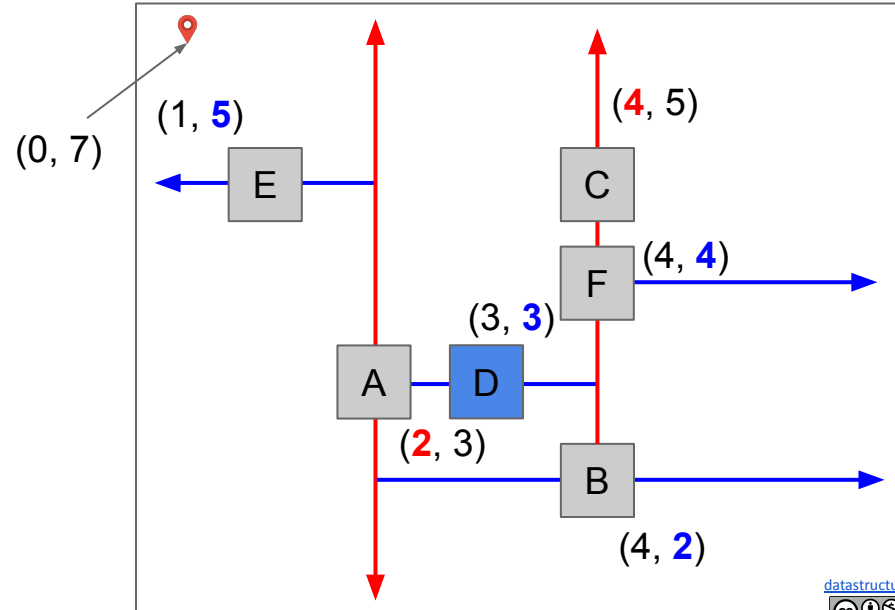
# K-d Trees and Nearest Neighbor

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

- Optimization: Do not explore subspaces that can't possibly 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  $n.\text{distance}(\text{goal}) < \text{best}.\text{distance}(\text{goal})$ ,  $\text{best} = n$
- If  $\text{goal} < n$  (according to n's comparator):
  - $\text{goodSide} = n.\text{"left"Child}$
  - $\text{badSide} = n.\text{"right"Child}$
  - else:
    - $\text{goodSide} = n.\text{"right"Child}$
    - $\text{badSide} = n.\text{"left"Child}$
- $\text{best} = \text{nearest}(\text{goodSide}, \text{goal}, \text{best})$
- If bad side could still have something useful
  - $\text{best} = \text{nearest}(\text{badSide}, \text{goal}, \text{best})$
- 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  $n.\text{distance}(\text{goal}) < \text{best}.\text{distance}(\text{goal})$ ,  $\text{best} = n$
- $\text{best} = \text{nearest}(n.\text{leftChild}, \text{goal}, \text{best})$
- $\text{best} = \text{nearest}(n.\text{rightChild}, \text{goal}, \text{best})$
- 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 efficient version on the previous slide.

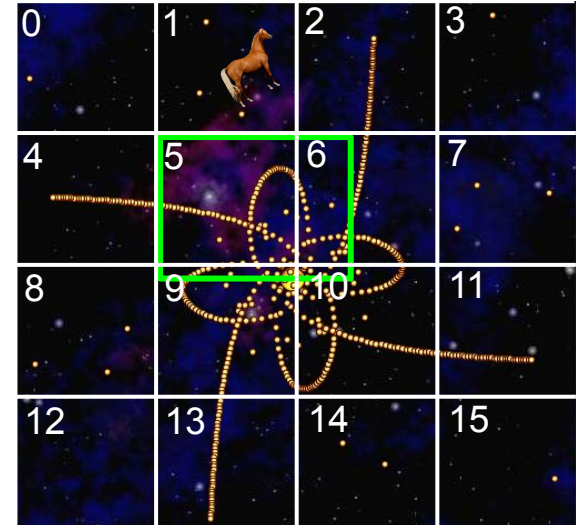
# Uniform Partitioning

# Uniform Partitioning

To end today, let's discuss a totally different approach called "Uniform partitioning". Unlike other approaches, isn't based on a tree at all.

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

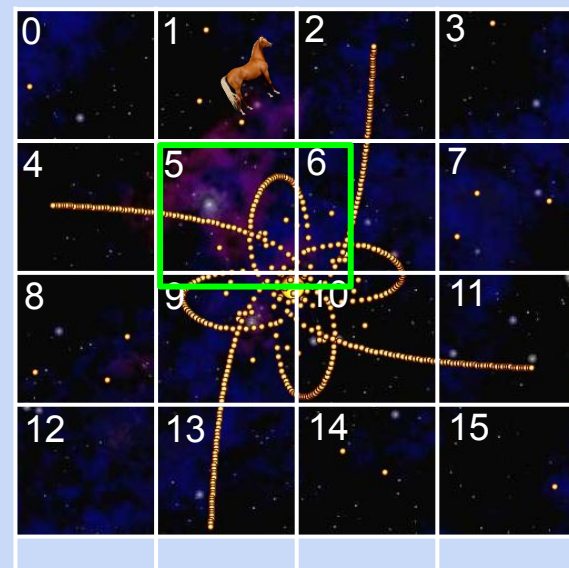
- Example on right for 4x4 grid of such buckets.



# Uniform Partitioning

Questions:

- Which buckets do we need to iterate over to find all points in the green range?

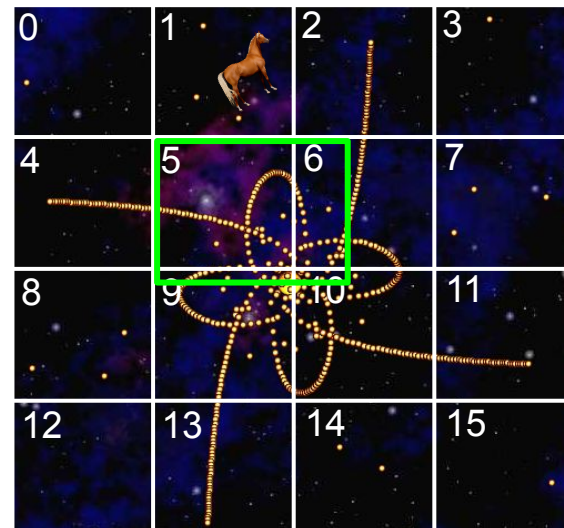


# Uniform Partitioning: Range Finding

Questions:

- Which buckets do we need to iterate over to find all points in the green range?
  - Only boxes 5, 6, 9, and 10.

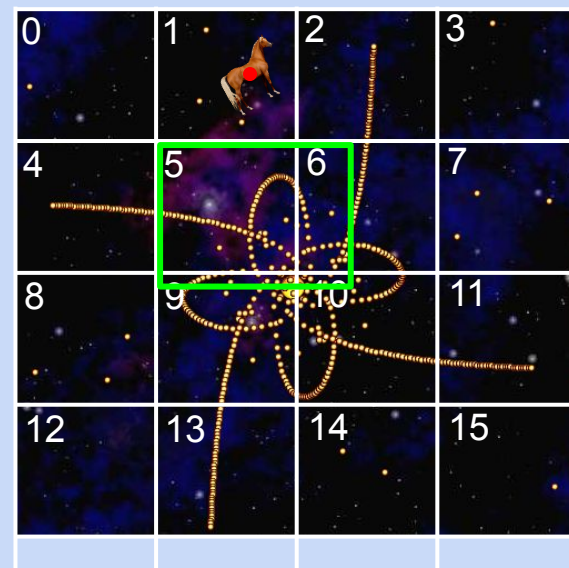
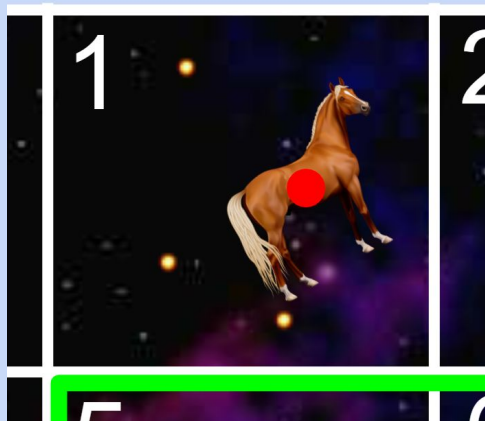
Note: Algorithm is still  $\Theta(N)$ , but it's faster than iterating over all points.



# Uniform Partitioning

Question:

- Which buckets do we need to look at to solve nearest(🐎)? Assume we're looking at distance to the red dot in the middle.

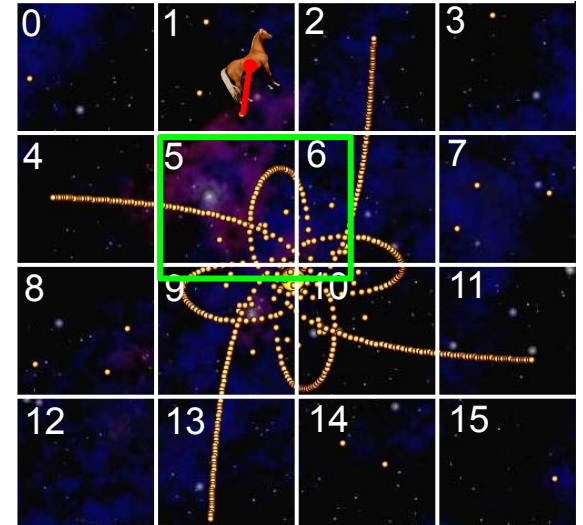
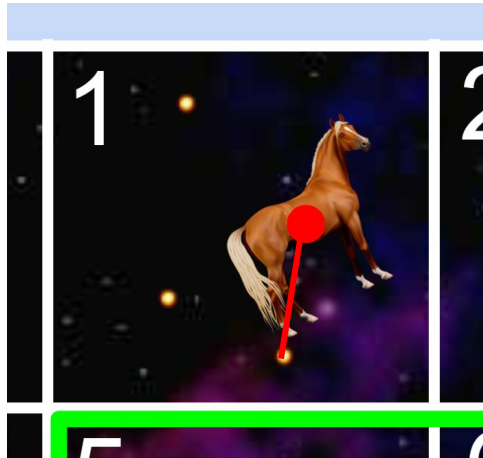


# Uniform Partitioning: Nearest

Question:

- Which buckets do we need to look at to solve nearest(🐎)?

After looking at bucket 1, we can prune 0, 4, 5, and 6 because there are no possible points in these boxes that could be closer than the closest in 1.



# Uniform Partitioning: Nearest

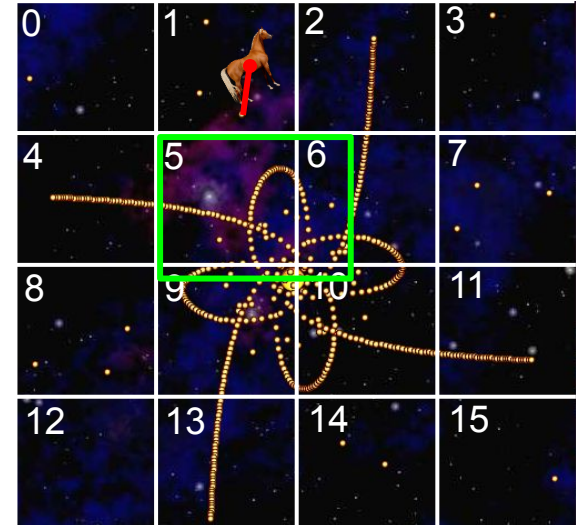
Question:

- Which buckets do we need to look at to solve nearest(🐎)?

After looking at bucket 1, we can prune 0, 4, 5, and 6 because there are no possible points in these boxes that could be closer than the closest in 1.

Overall runtime is still  $\Theta(N)$ .

- ... but maybe up to 16 times faster than iterating over all points.

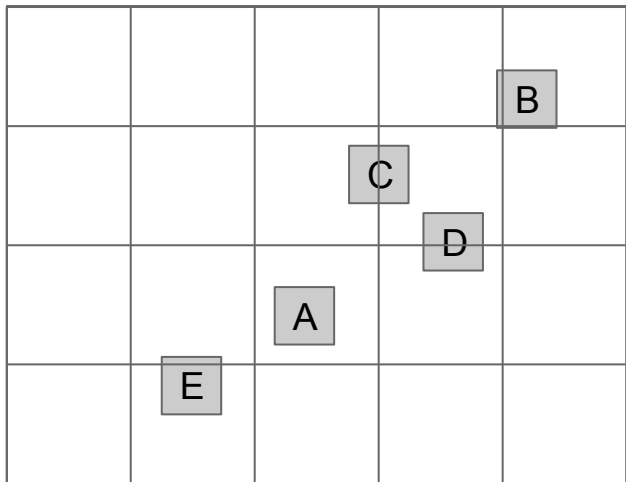




# Uniform vs. Hierarchical Partitioning

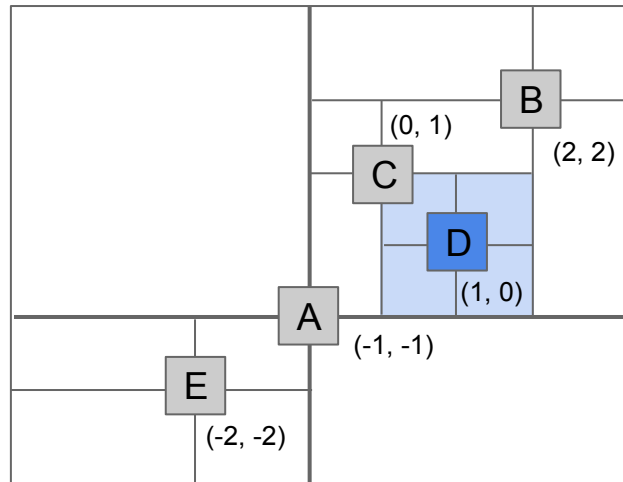
All of our approaches today boil down to spatial partitioning.

- Uniform partitioning (perfect grid of rectangles).
- Quadtrees and KdTrees: Hierarchical partitioning.
  - Each node “owns” 4 and 2 subspaces, respectively.
  - Space is more finely divided into subspaces where there are more points.



25 subspaces of uniform sizes.

D “owns”  
the 4 blue  
subspaces.



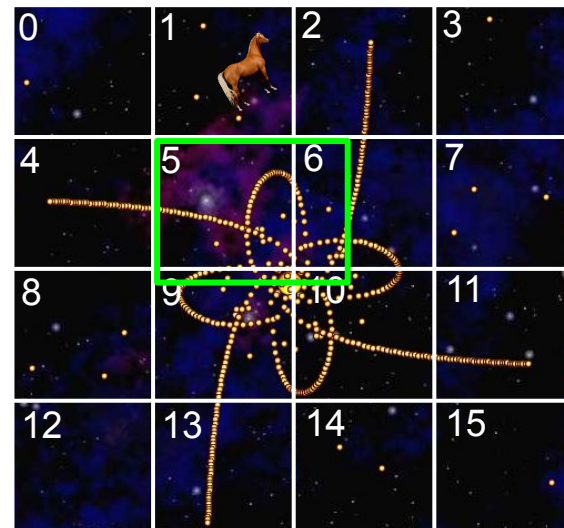
16 subspaces of varying sizes.

# Uniform Partitioning vs. Quadrees and Kd-Trees

Uniform partitioning is easier to implement than a Quadtree or Kd-Tree.

- May be good enough for many applications.

In our next lecture we'll see a very clever 1D version of this idea called a "hash table".



# Summary and Applications

# Multi-Dimensional Data Summary

---

Set operations can be more complex than just contains, e.g.:

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

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

- **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.
- **Uniform Partitioning:** Partition space into uniform chunks.

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)