



<http://algs4.cs.princeton.edu>

## GEOMETRIC APPLICATIONS OF BSTs

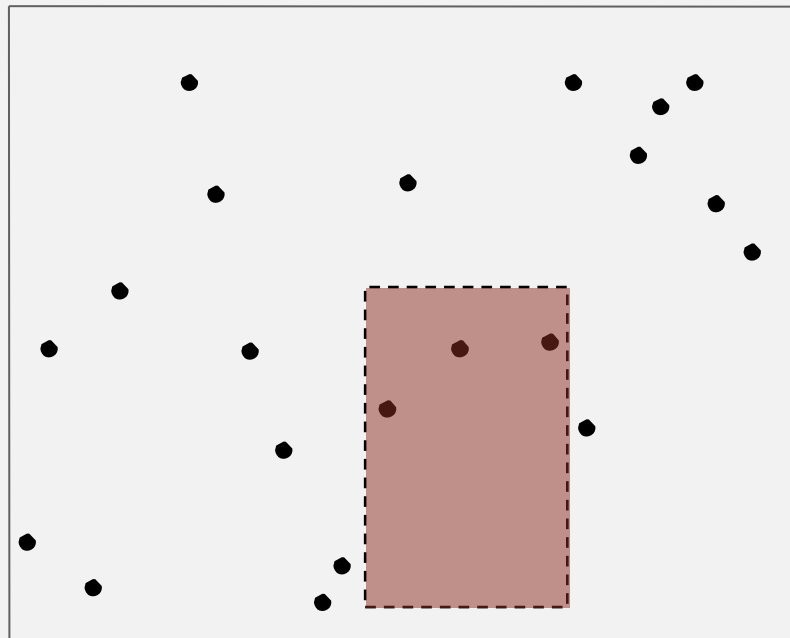
---

- *1d range search*
- *line segment intersection*
- *kd trees*
- *interval search trees*
- *rectangle intersection*
- *S3*

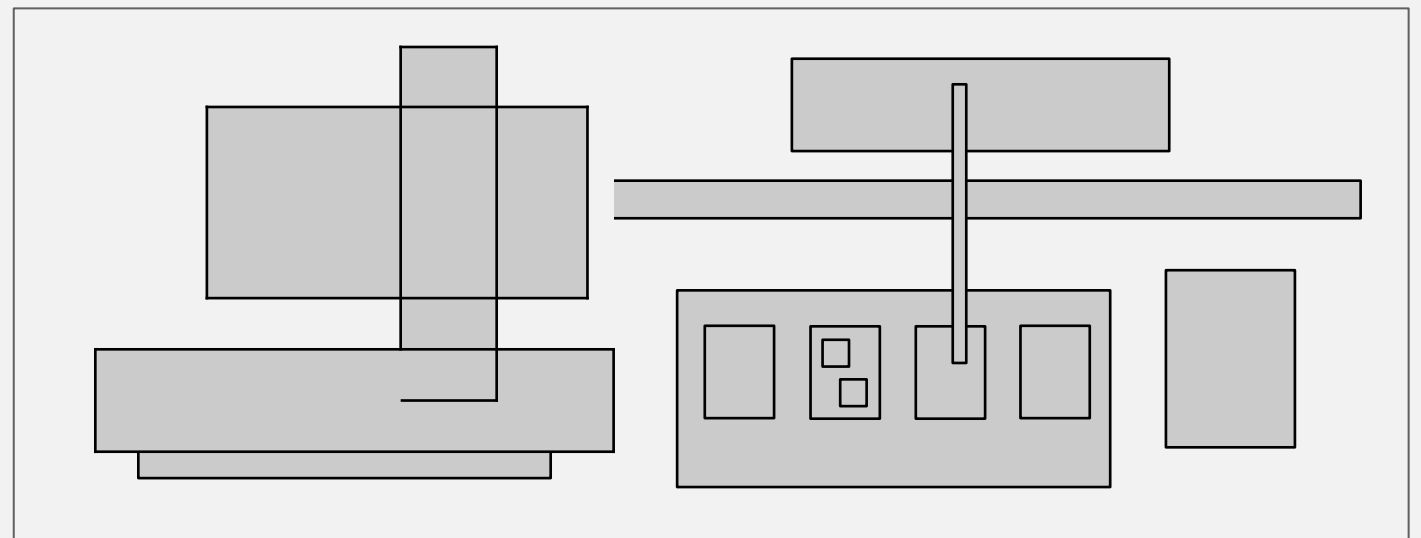
# Overview

---

**This lecture.** Intersections among **geometric objects**.



2d orthogonal range search



orthogonal rectangle intersection

**Applications.** CAD, games, movies, virtual reality, databases, GIS, ....

**Efficient solutions.** **Binary search trees** (and extensions).



<http://algs4.cs.princeton.edu>

## GEOMETRIC APPLICATIONS OF BSTs

---

- *1d range search*
- *line segment intersection*
- *kd trees*
- *interval search trees*
- *rectangle intersection*

# 1d range search

---

## Extension of ordered symbol table.

- Insert key-value pair.
- Search for key  $k$ .
- Delete key  $k$ .
- **Range search:** find all keys between  $k_1$  and  $k_2$ .
- **Range count:** number of keys between  $k_1$  and  $k_2$ .

Svæðisleit

Svæðistalning

## Application. Database queries.

## Geometric interpretation.

- Keys are point on a **line**.
- Find/count points in a given **1d interval**.



```
insert B B
insert D B D
insert A A B D
insert I A B D I
insert H A B D H I
insert F A B D F H I
insert P A B D F H I P
count G to K 2
search G to K H I
```

## Quiz 1

---

Suppose that the keys are stored in a sorted array. What is the order of growth of the running time to perform **range count** as a function of  $N$  and  $R$ ?



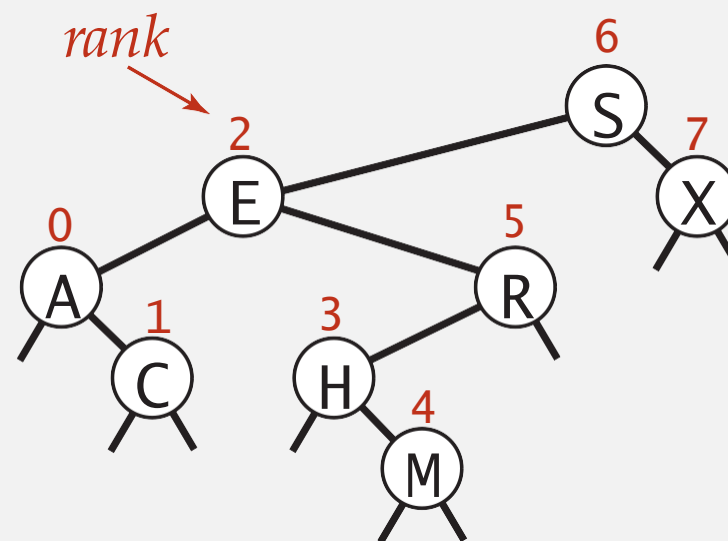
$N$  = number of keys

$R$  = number of matching keys

- A.  $\log R$
- B.  $\log N$
- C.  $R + \log N$
- D.  $R + N$
- E. *I don't know.*

# 1d range count: BST implementation

1d range count. How many keys between  $lo$  and  $hi$  ?



5 keys between E and S

$\text{rank}(E) = 2$

$\text{rank}(S) = 6$

```
public int size(Key lo, Key hi)
{
    if (contains(hi)) return rank(hi) - rank(lo) + 1;
    else               return rank(hi) - rank(lo);
}
```

← number of keys < hi

**Proposition.** Running time proportional to  $\log N$ . ← assuming BST is balanced

**Pf.** Nodes examined = search path to  $lo$  + search path to  $hi$ .

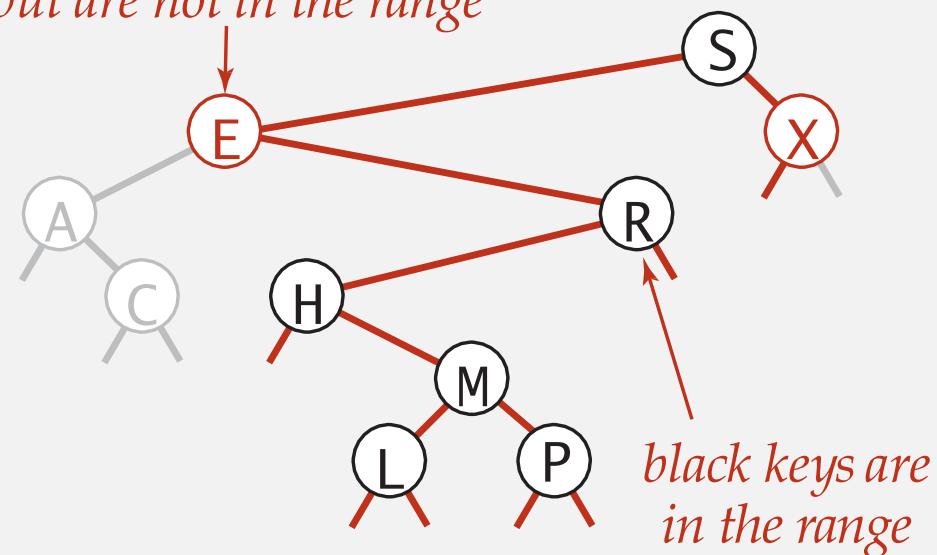
# 1d range search: BST implementation

**1d range search.** Find all keys between  $l_o$  and  $h_i$ .

- Recursively find all keys in left subtree (if any could fall in range).
- Check key in current node.
- Recursively find all keys in right subtree (if any could fall in range).

searching in the range [F..T]

*red keys are used in compares  
but are not in the range*



**Proposition.** Running time proportional to  $R + \log N$ .

**Pf.** Nodes examined = search path to  $l_o$  + search path to  $h_i$  + matches.





<http://algs4.cs.princeton.edu>

## GEOMETRIC APPLICATIONS OF BSTs

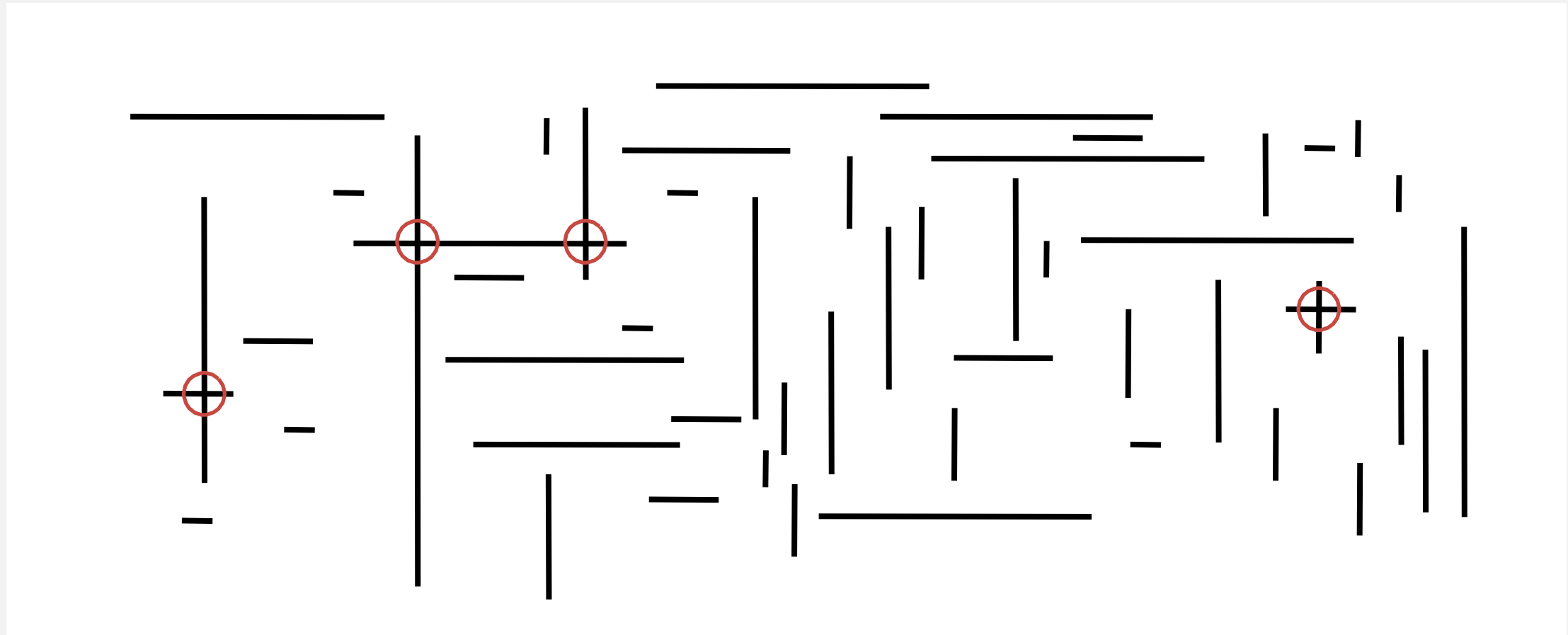
---

- *1d range search*
- *line segment intersection*
- *kd trees*
- *interval search trees*
- *rectangle intersection*



# Orthogonal line segment intersection

Given  $N$  horizontal and vertical line segments, find all intersections.



**Quadratic algorithm.** Check all pairs of line segments for intersection.

Einfeldni

**Nondegeneracy assumption.** All  $x$ - and  $y$ -coordinates are distinct.

# Microprocessors and geometry

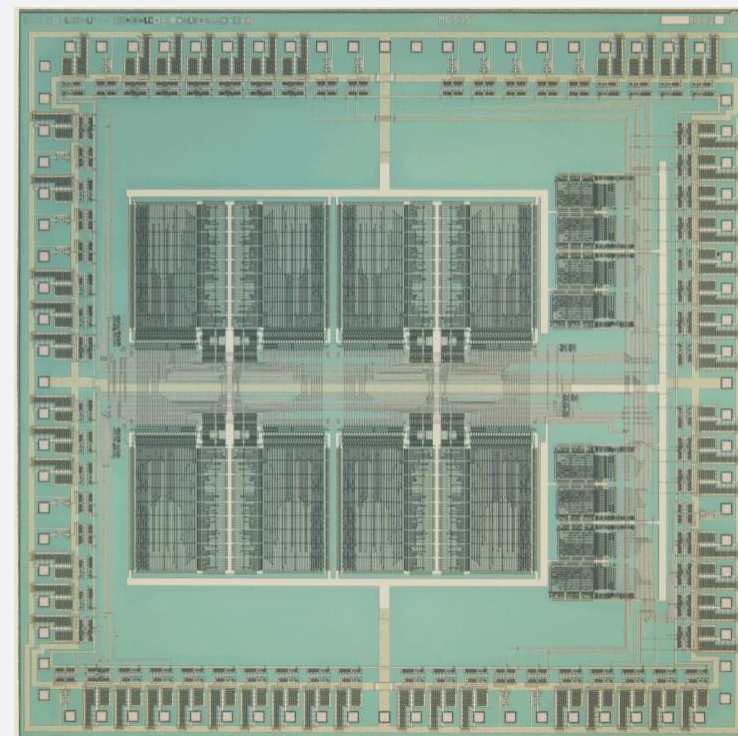
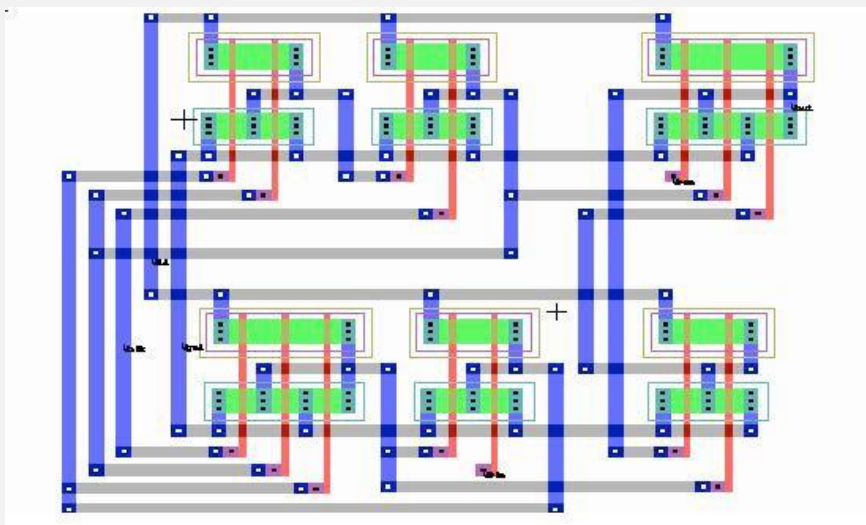
---

**Early 1970s.** Microprocessor design became a **geometric** problem.

- Very Large Scale Integration (VLSI).
- Computer-Aided Design (CAD).

**Design-rule checking.**

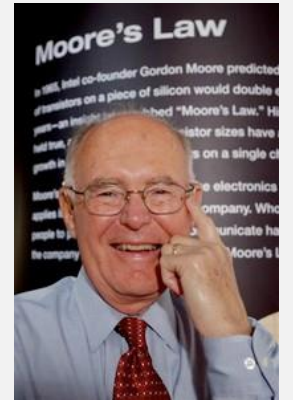
- Certain wires cannot intersect.
- Certain spacing needed between different types of wires.
- Debugging = orthogonal rectangle intersection search.



# Algorithms and Moore's law

---

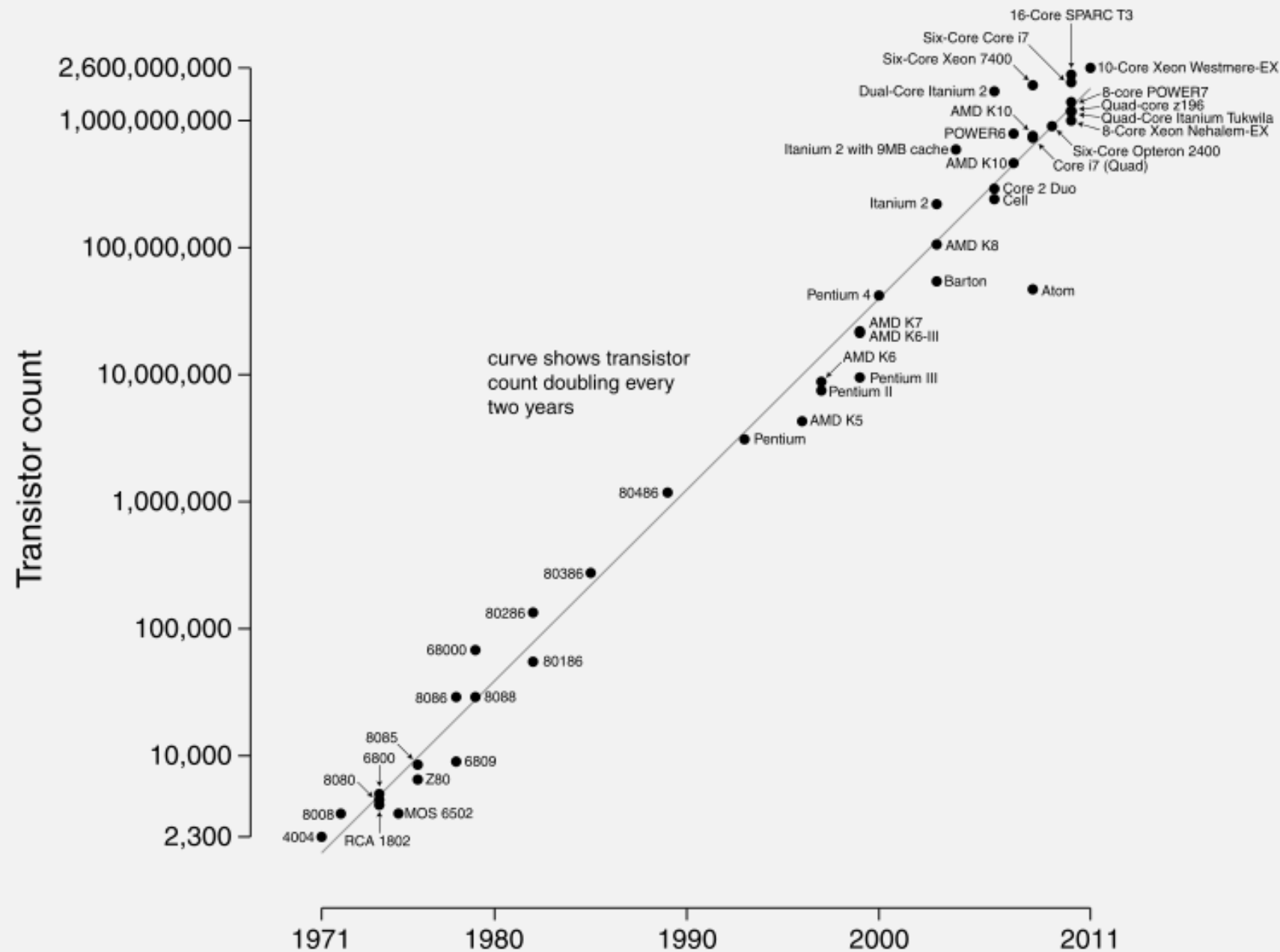
Moore's law. ??? doubles every X years.



**Gordon Moore**

# Algorithms and Moore's law

Moore's law. Transistor count doubles every 2 years.

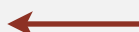



Gordon Moore

[http://commons.wikimedia.org/wiki/File%3ATransistor\\_Count\\_and\\_Moore's\\_Law\\_-\\_2011.svg](http://commons.wikimedia.org/wiki/File%3ATransistor_Count_and_Moore's_Law_-_2011.svg)

# Algorithms and Moore's law

## Sustaining Moore's law.

- Problem size doubles every 2 years.  problem size = transistor count
- Processing power doubles every 2 years.  get to use faster computer
- How much \$ do I need to get the job done with a quadratic algorithm?

$$T_N = a N^2 \quad \text{running time today}$$

$$T_{2N} = (a/2) (2N)^2 \quad \text{running time in 2 years}$$

$$= 2 T_N$$

$$= 2 a N^2$$

running time	1970	1972	1974	2000
$N$	$\$X$	$\$X$	$\$X$	$\$X$
$N \log N$	$\$X$	$\$X$	$\$X$	$\$X$
$N^2$	$\$X$	$\$2X$	$\$4X$	$\$2^{15} X$

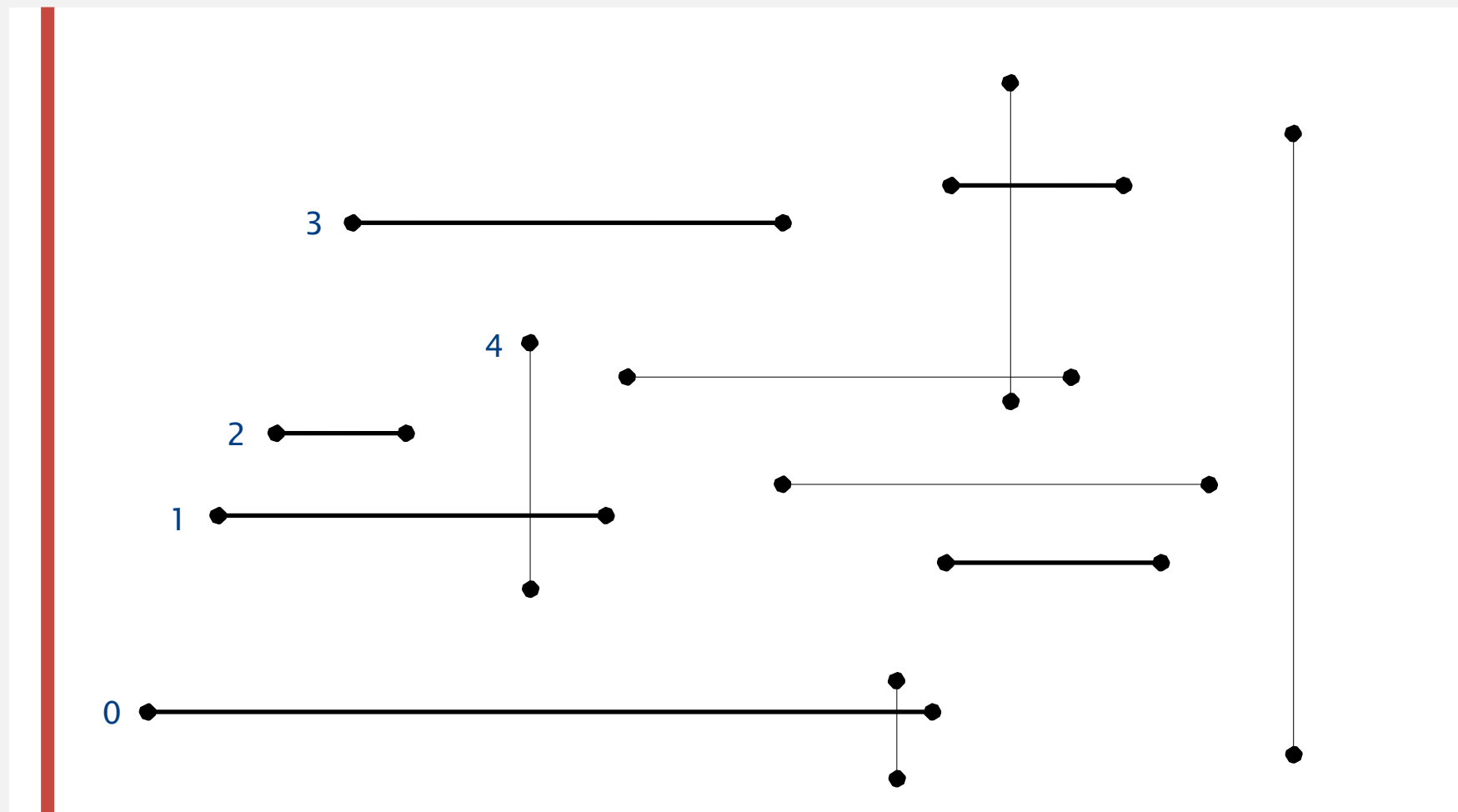
**Bottom line.** Linearithmic algorithm is **necessary** to sustain Moore's Law.

# Orthogonal line segment intersection: sweep-line algorithm

Sweep vertical line from left to right.

Draglína, slæðilína

- $x$ -coordinates define events.
- $h$ -segment (left endpoint): insert  $y$ -coordinate into BST.



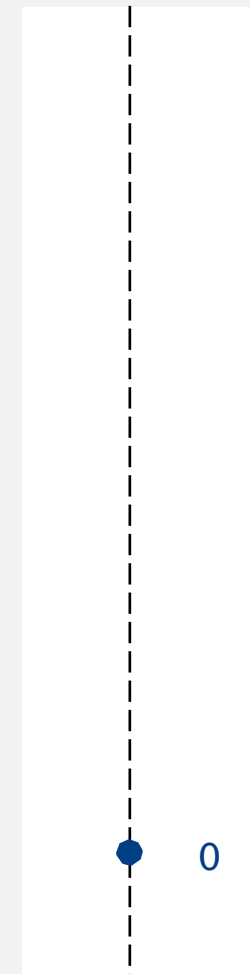
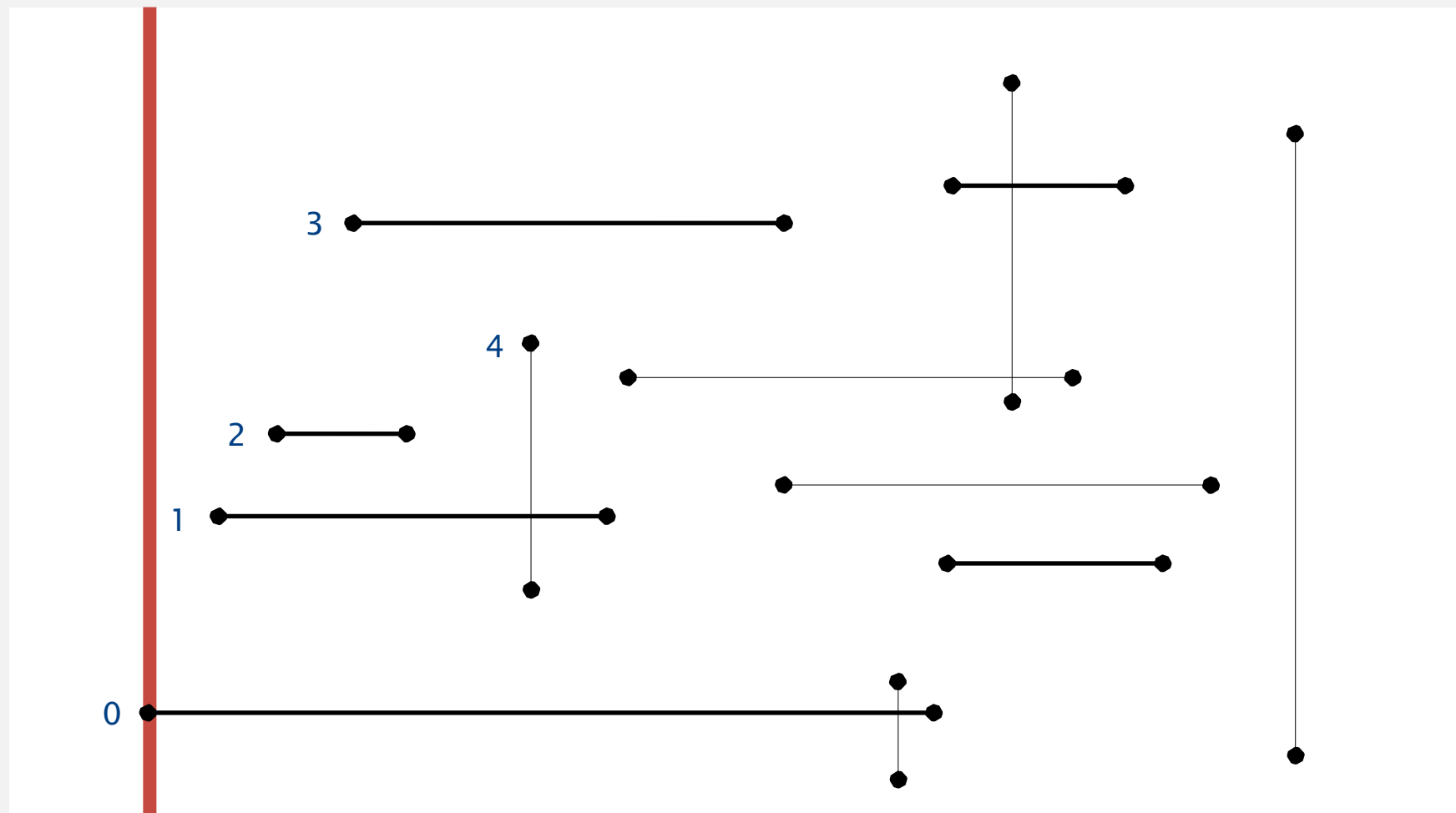
y - c o o r d i n a t e s

# Orthogonal line segment intersection: sweep-line algorithm

Sweep vertical line from left to right.

Draglína, slæðilína

- $x$ -coordinates define events.
- $h$ -segment (left endpoint): insert  $y$ -coordinate into BST.



y - coordinates

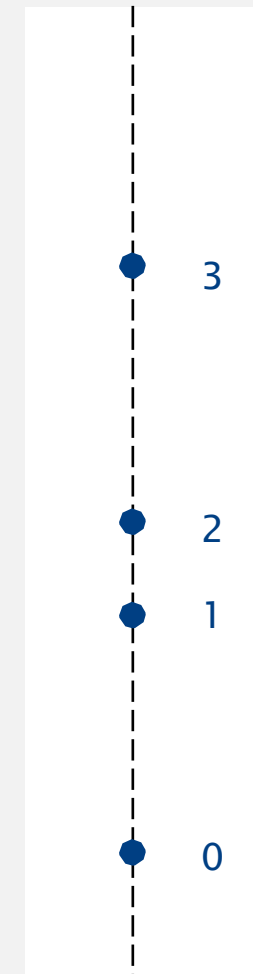
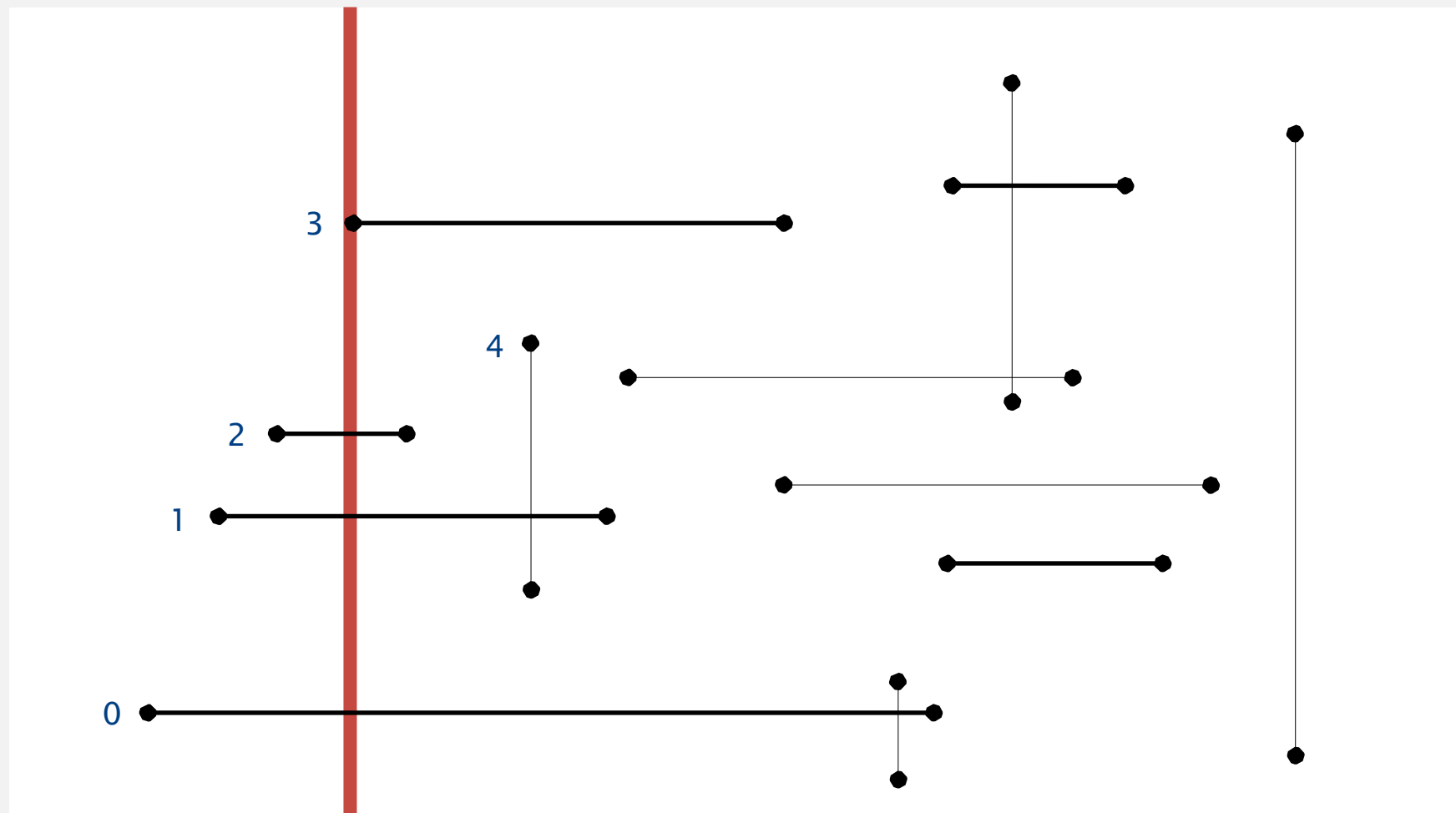


# Orthogonal line segment intersection: sweep-line algorithm

Sweep vertical line from left to right.

Draglína, slæðilína

- $x$ -coordinates define events.
- $h$ -segment (left endpoint): insert  $y$ -coordinate into BST.

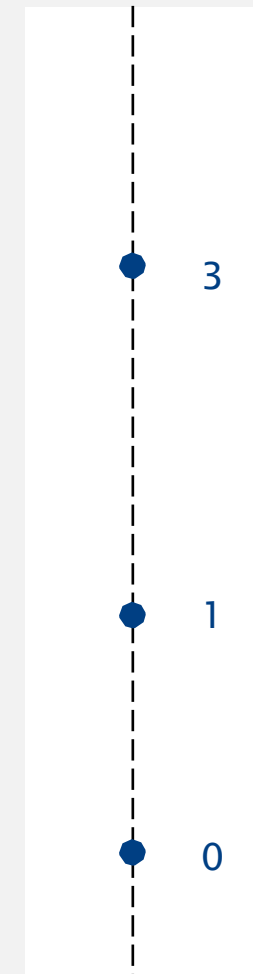
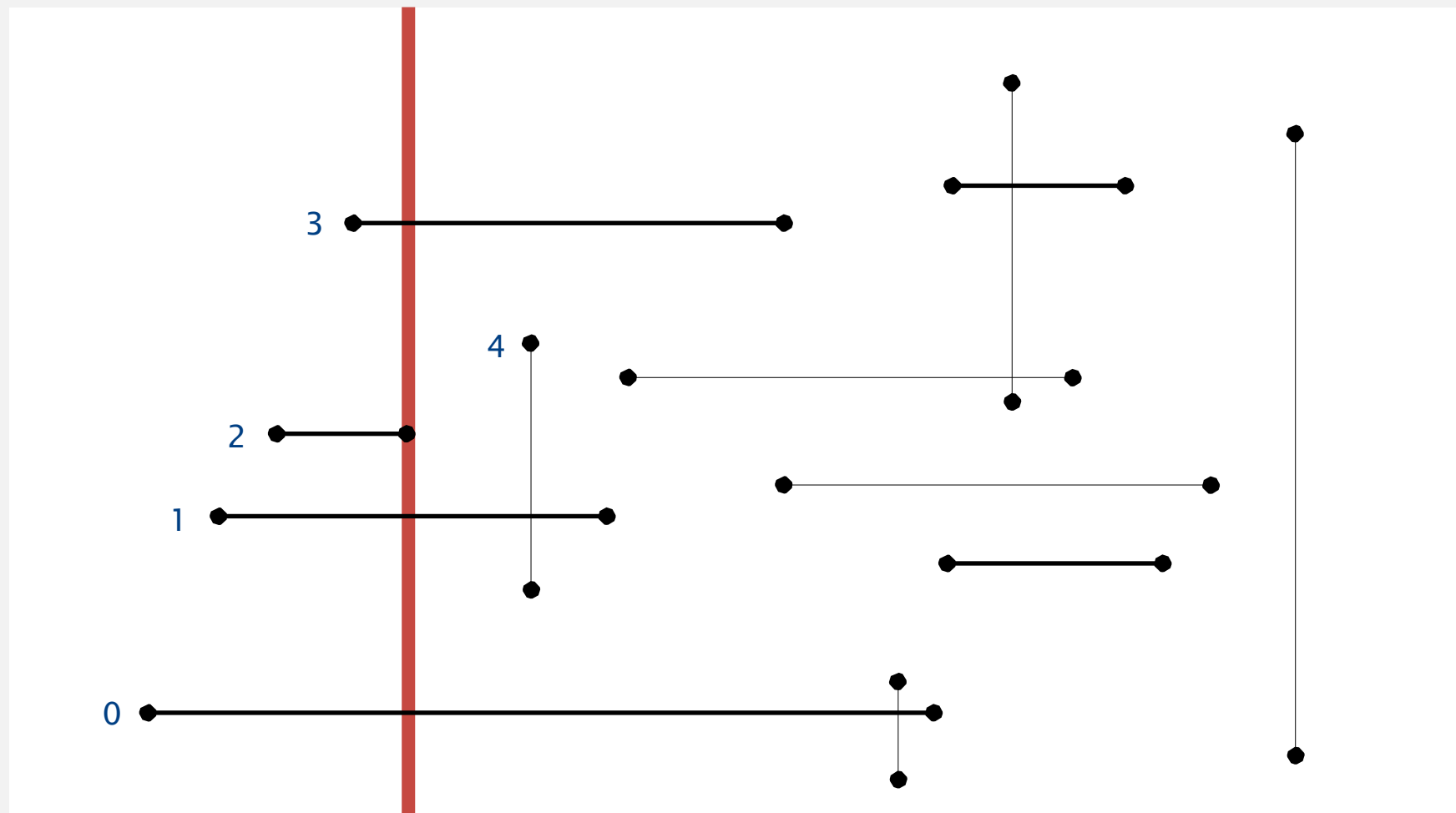


y - coordinates

# Orthogonal line segment intersection: sweep-line algorithm

Sweep vertical line from left to right.

- $x$ -coordinates define events.
- $h$ -segment (left endpoint): insert  $y$ -coordinate into BST.
- $h$ -segment (right endpoint): remove  $y$ -coordinate from BST.

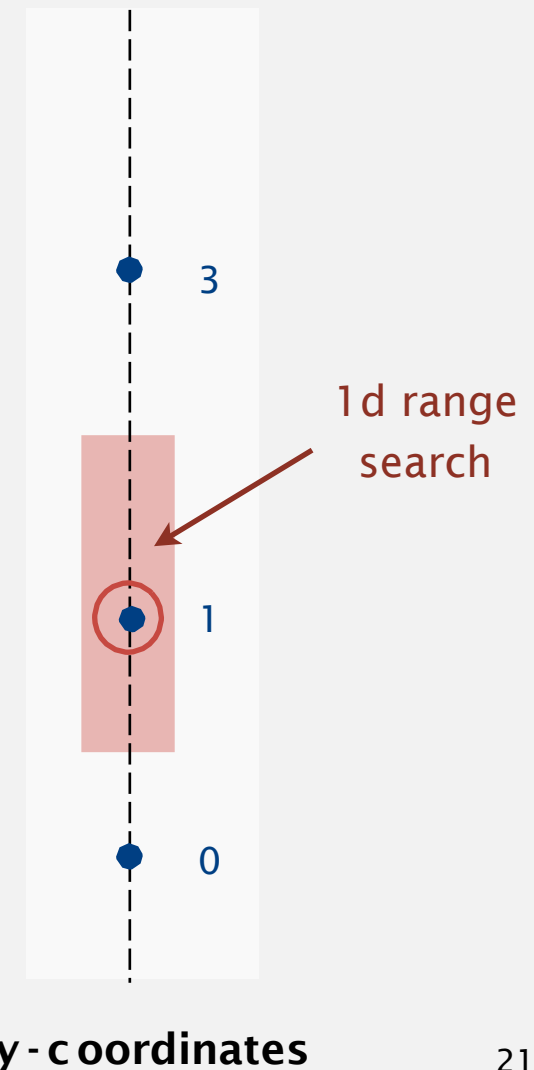
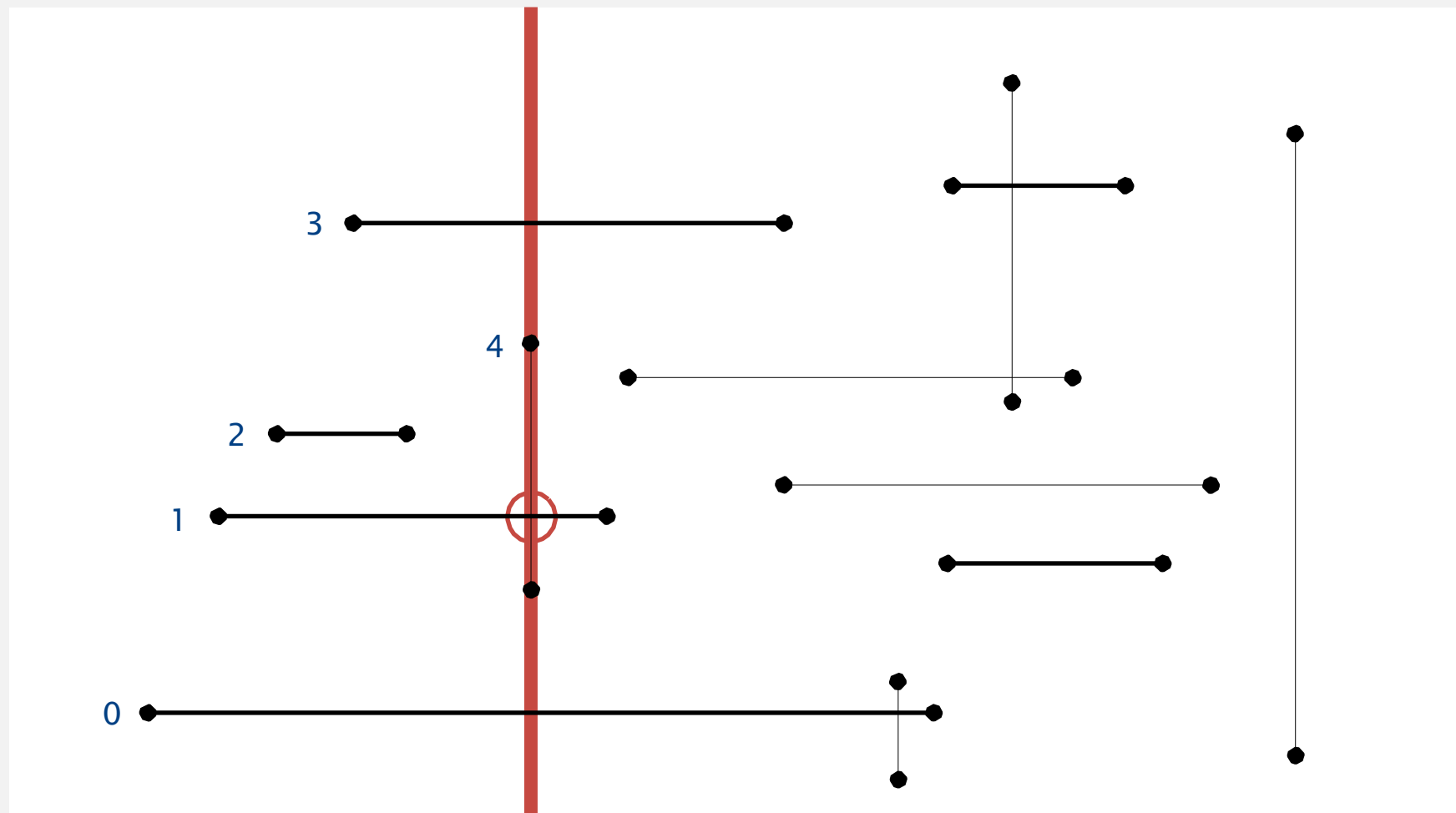


y - c coordinates

# Orthogonal line segment intersection: sweep-line algorithm

Sweep vertical line from left to right.

- $x$ -coordinates define events.
- $h$ -segment (left endpoint): insert  $y$ -coordinate into BST.
- $h$ -segment (right endpoint): remove  $y$ -coordinate from BST.
- $v$ -segment: range search for interval of  $y$ -endpoints.



# Orthogonal line segment intersection: sweep-line algorithm

---

**Proposition.** The sweep-line algorithm takes time proportional to  $N \log N + R$  to find all  $R$  intersections among  $N$  orthogonal line segments.

**Pf.**

- Put  $x$ -coordinates on a PQ (or sort).  $\leftarrow N \log N$
- Insert  $y$ -coordinates into BST.  $\leftarrow N \log N$
- Delete  $y$ -coordinates from BST.  $\leftarrow N \log N$
- Range searches in BST.  $\leftarrow N \log N + R$

**Bottom line.** Sweep line reduces 2d orthogonal line segment intersection search to 1d range search.



<http://algs4.cs.princeton.edu>

## GEOMETRIC APPLICATIONS OF BSTs

---

- *1d range search*
- *line segment intersection*
- *kd trees*
- *interval search trees*
- *rectangle intersection*

# 2d orthogonal range search

---

## Extension of ordered symbol-table to 2d keys.

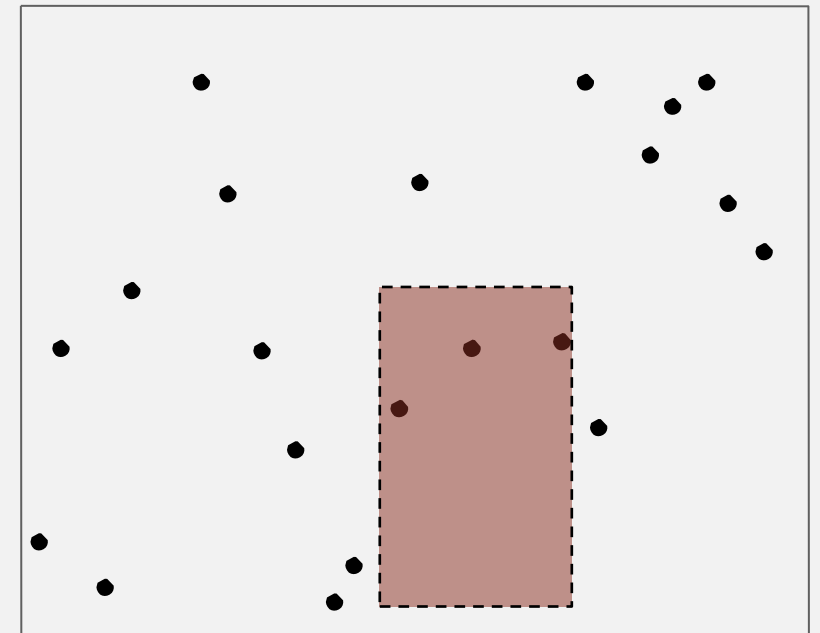
- Insert a 2d key.
- Search for a 2d key.
- Delete a 2d key.
- **Range search:** find all keys that lie in a 2d range.
- **Range count:** number of keys that lie in a 2d range.

**Applications.** Networking, circuit design, databases, ...

## Geometric interpretation.

- Keys are point in the **plane**.
- Find/count points in a given  **$h-v$  rectangle**

↑  
rectangle is axis-aligned



# 2d orthogonal range search: grid implementation

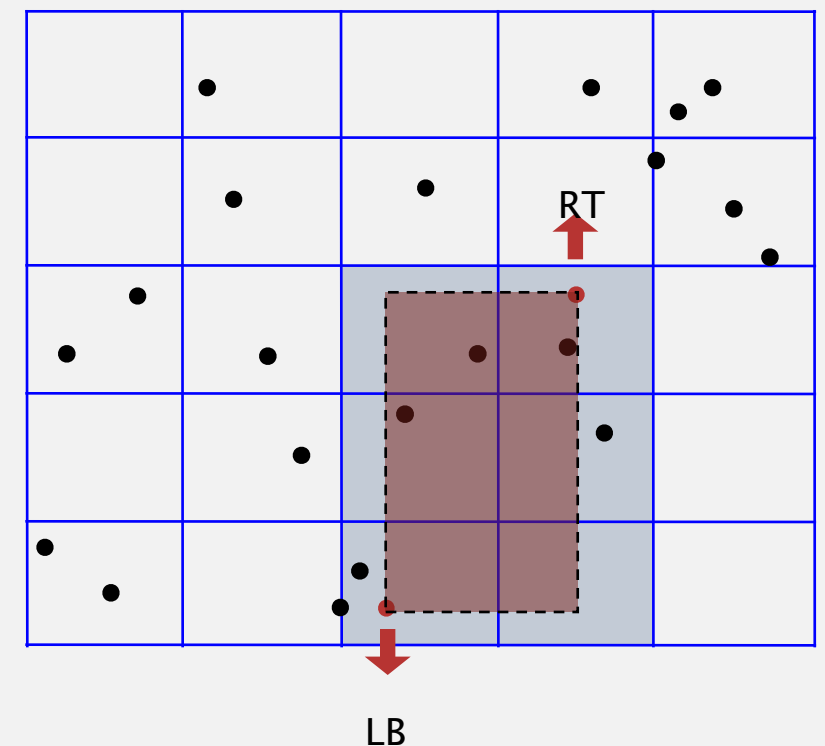
---

## Grid implementation.

- Divide space into  $M$ -by- $M$  grid of squares.
- Create list of points contained in each square.
- Use 2d array to directly index relevant square.
- Insert: add  $(x, y)$  to list for corresponding square.
- Range search: examine only squares that intersect 2d range query.

## Analysis

- Space: ??
- Time: ??





# 2d orthogonal range search: grid implementation analysis

---

## Space-time tradeoff.

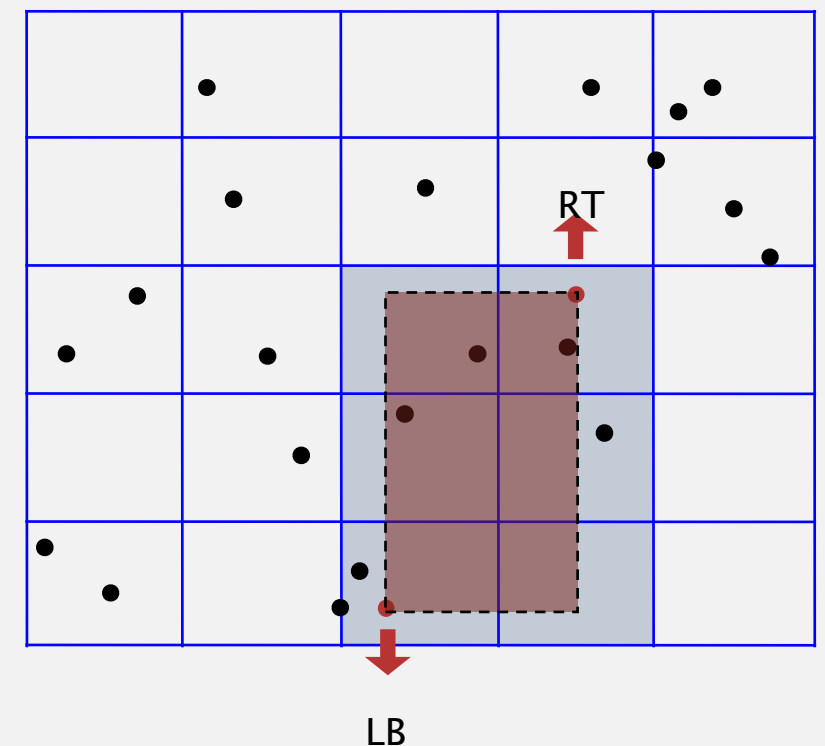
- Space:  $M^2 + N$ .
- Time:  $1 + N / M^2$  per square examined, on average.

## Choose grid square size to tune performance.

- Too small: wastes space.
- Too large: too many points per square.
- Rule of thumb:  $\sqrt{N}$ -by- $\sqrt{N}$  grid.

## Running time. [if points are evenly distributed]

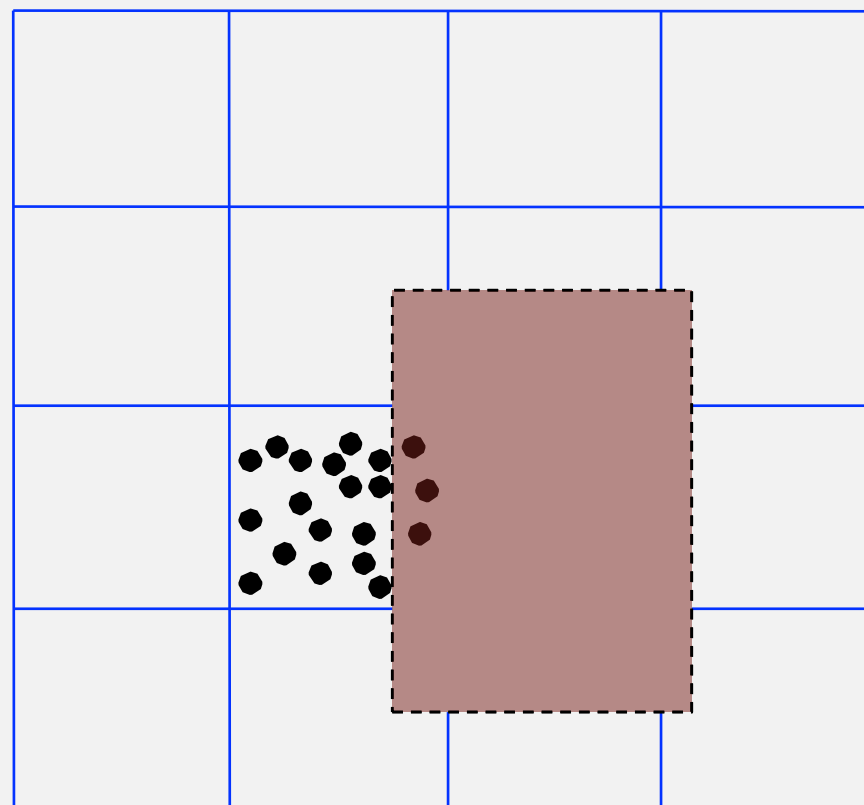
- Initialize data structure:  $N$ .
  - Insert point: 1.
  - Range search: 1 per point in range.
- choose  $M \sim \sqrt{N}$



**Grid implementation.** Fast, simple solution for evenly-distributed points.

**Problem.** **Clustering** a well-known phenomenon in geometric data.

- Lists are too long, even though average length is short.
- Need data structure that adapts gracefully to data.



# Clustering

---

**Grid implementation.** Fast, simple solution for evenly-distributed points.

**Problem.** **Clustering** a well-known phenomenon in geometric data.

**Ex.** USA map data.



13,000 points, 1000 grid squares



↑  
half the squares are empty

↑  
half the points are  
in 10% of the squares

# Space-partitioning trees

---

Use a **tree** to represent a recursive subdivision of 2d space.

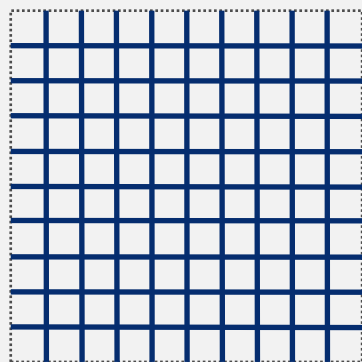
**Grid.** Divide space uniformly into squares.

Quadtree. Recursively divide space into four quadrants.

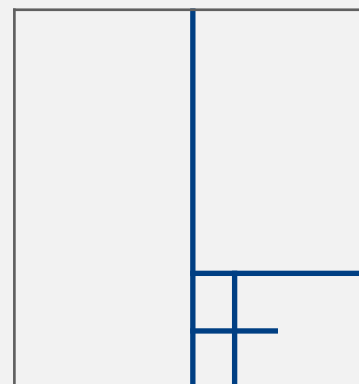
**2d tree.** Recursively divide space into two halfplanes.

BSP tree. Recursively divide space into two regions

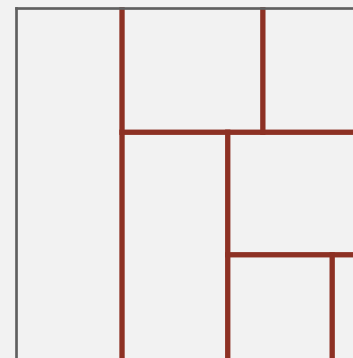
háflslétta



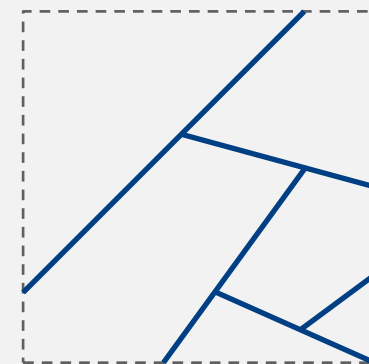
**Grid**



**Quadtree**



**2d tree**

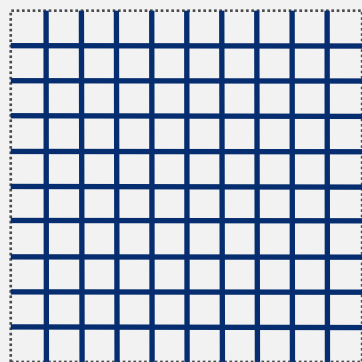


**BSP tree**

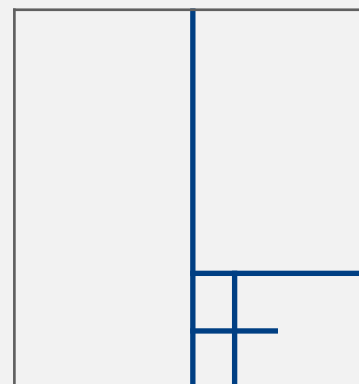
# Space-partitioning trees: applications

## Applications.

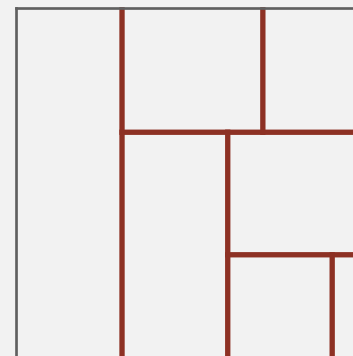
- Ray tracing.
- 2d range search.
- Flight simulators.
- N-body simulation.
- Collision detection.
- Astronomical databases.
- Nearest neighbor search.
- Adaptive mesh generation.
- Accelerate rendering in Doom.
- Hidden surface removal and shadow casting.



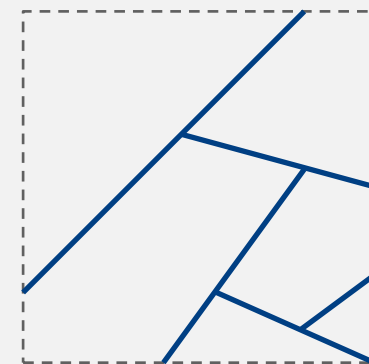
Grid



Quadtree



2d tree



BSP tree



# Curse of dimensionality

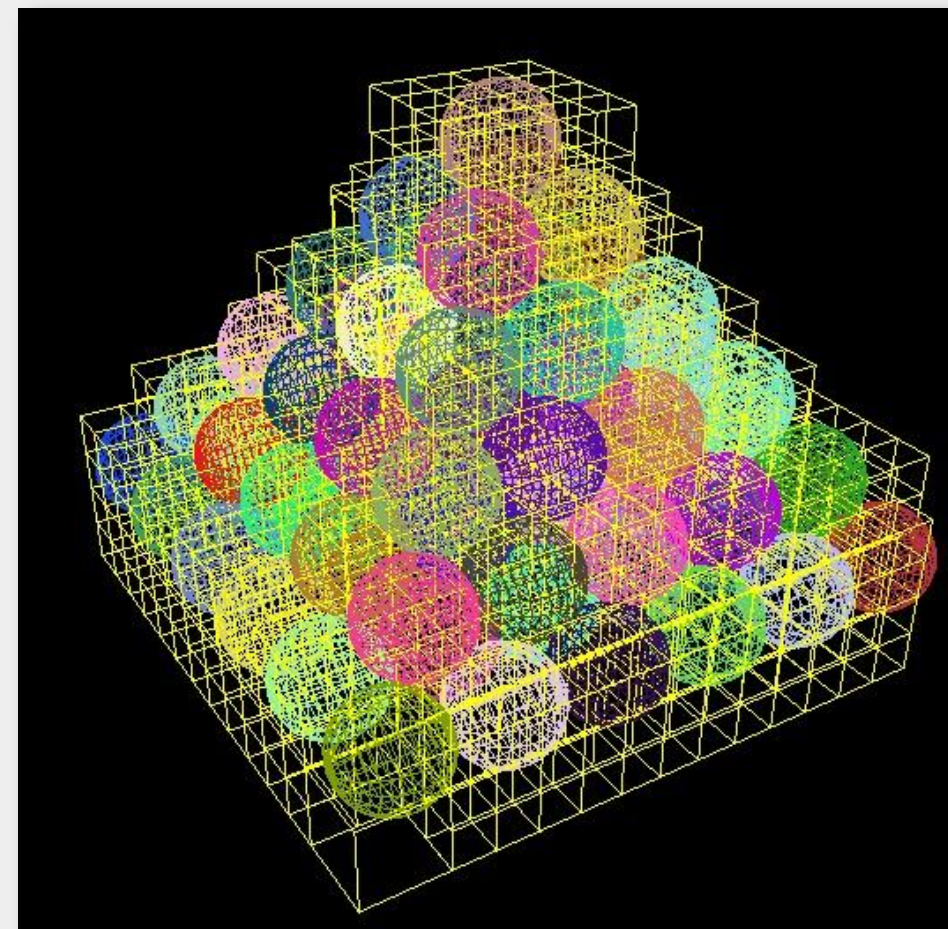
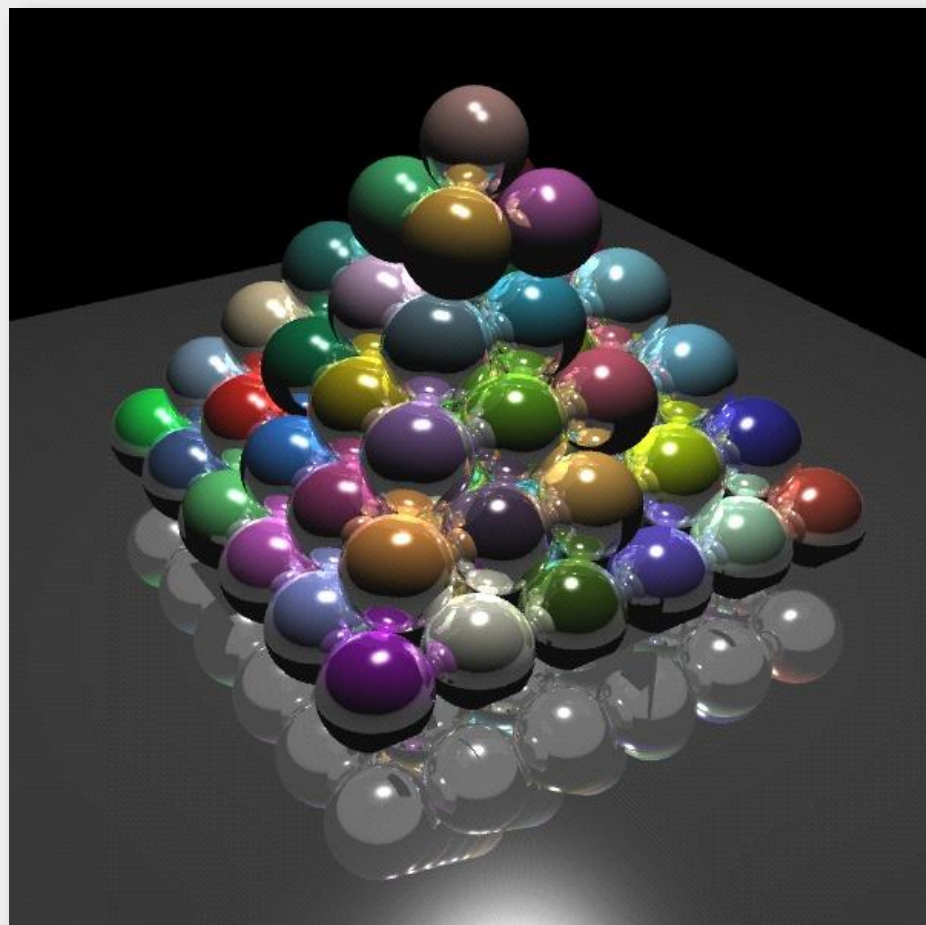
---

**k-d range search.** Orthogonal range search in  $k$ -dimensions.

**Main application.** Multi-dimensional databases.

**3d space.** Octrees: subdivide 3d space into 8 octants.

**100d space.** Centrees: subdivide 100d space into  $2^{100}$  centrants???

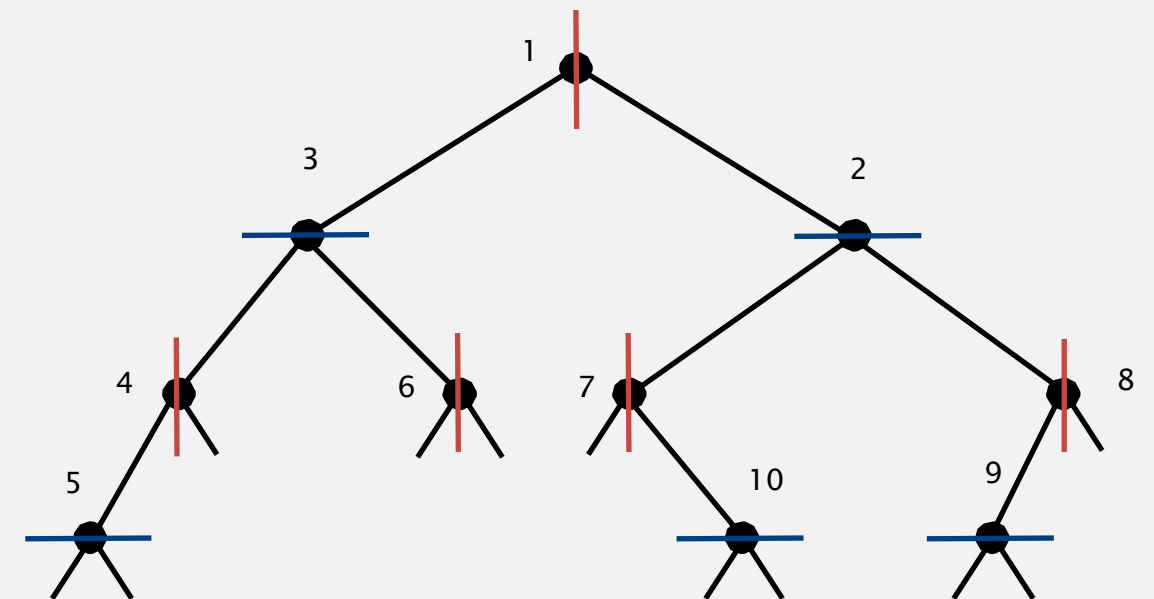
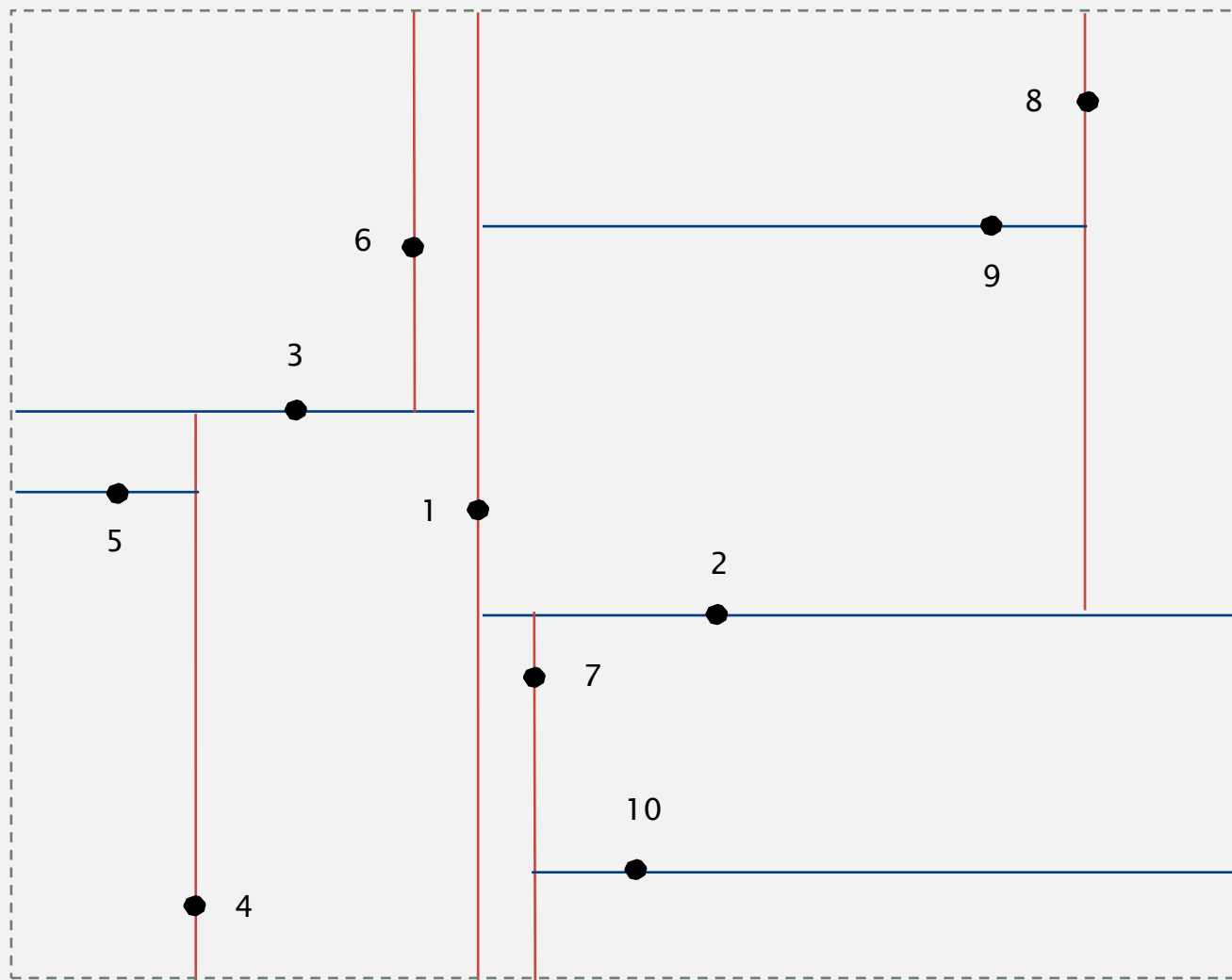


Raytracing with octrees

<http://graphics.cs.ucdavis.edu/~gregorsk/graphics/275.html>

# 2d tree construction

Recursively partition plane into two halfplanes.

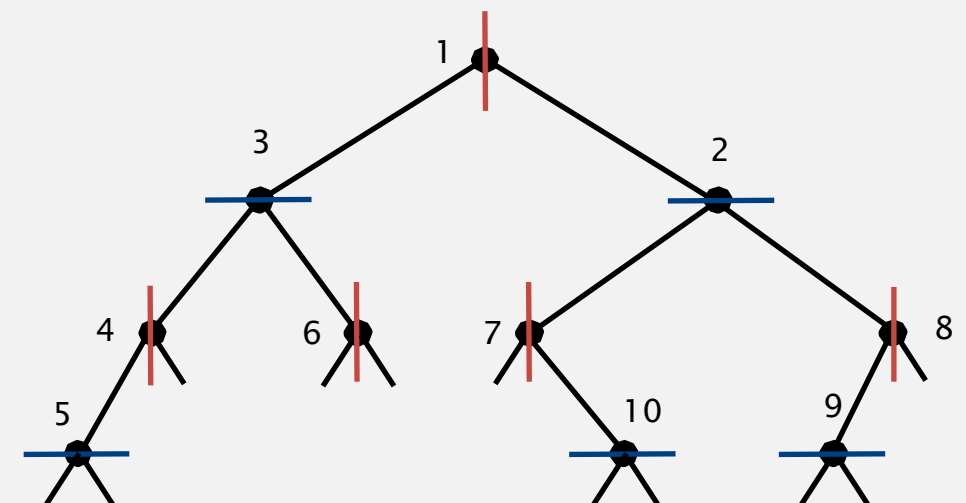
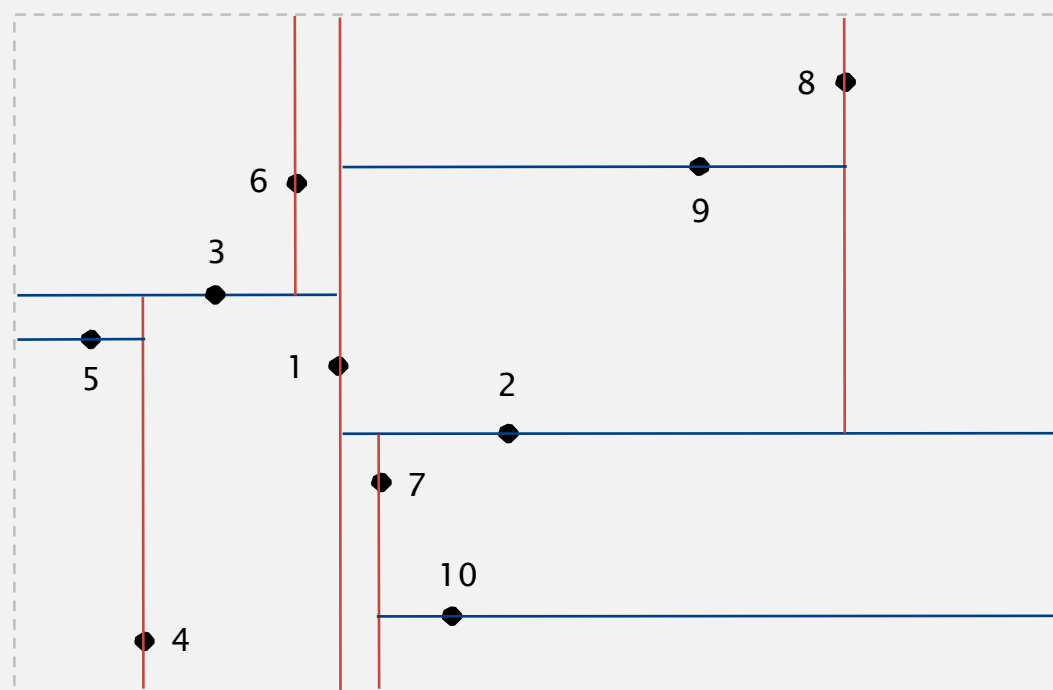
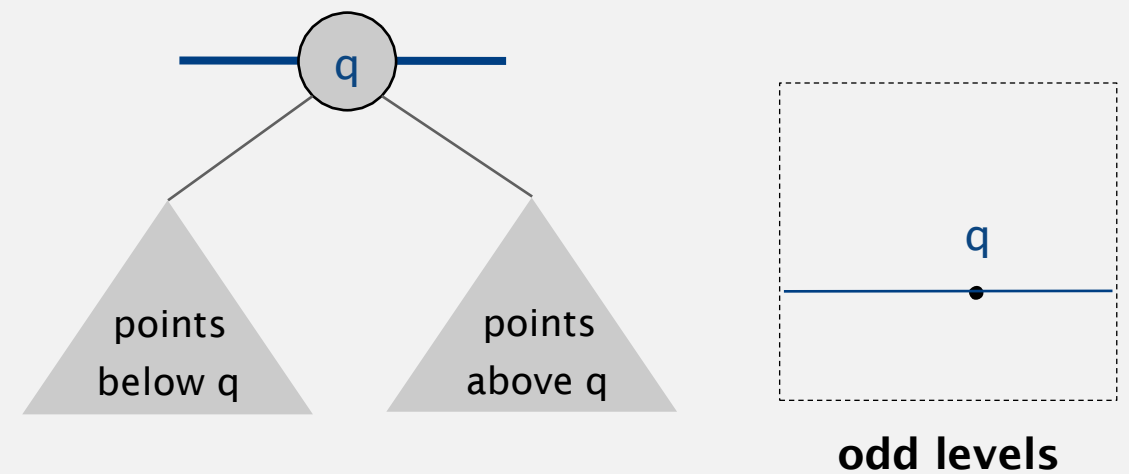
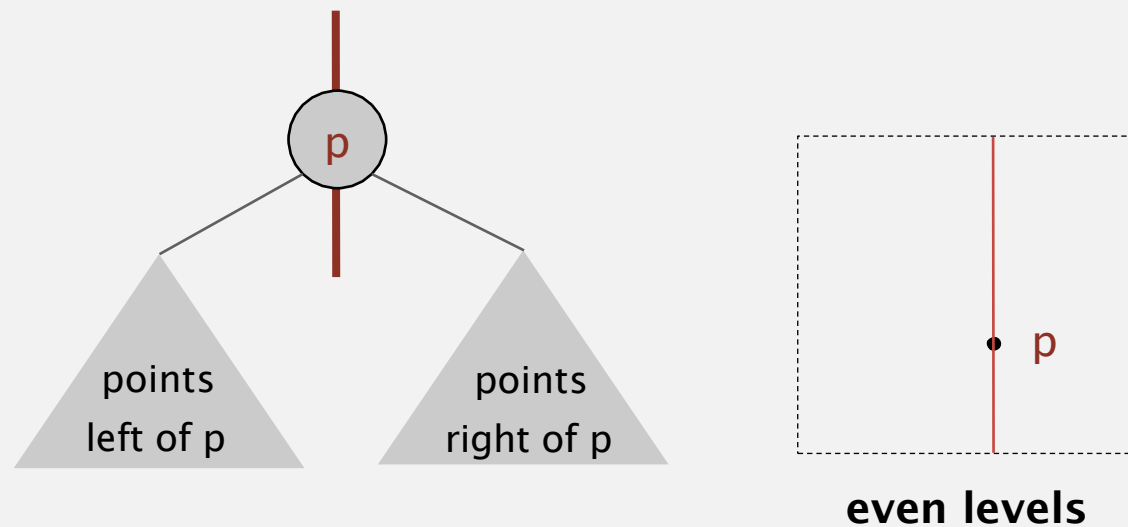




# 2d tree implementation

**Data structure.** BST, but alternate using  $x$ - and  $y$ -coordinates as key.

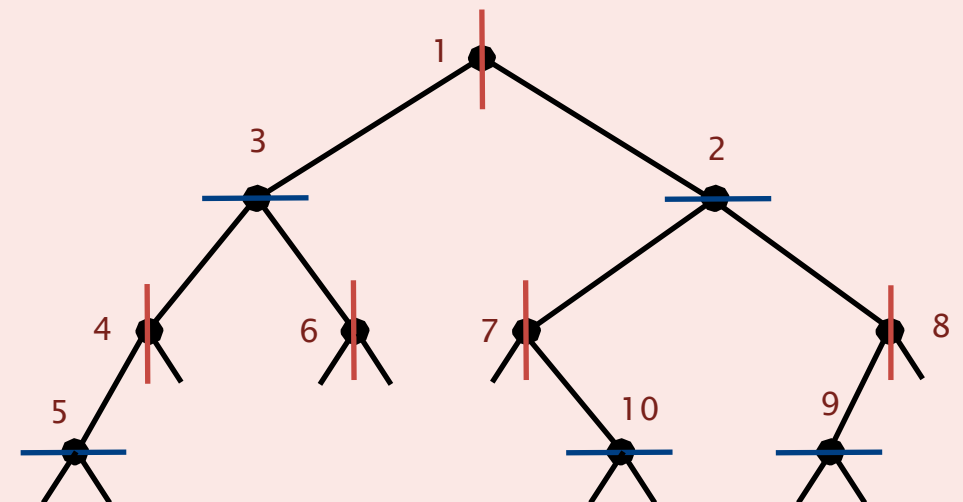
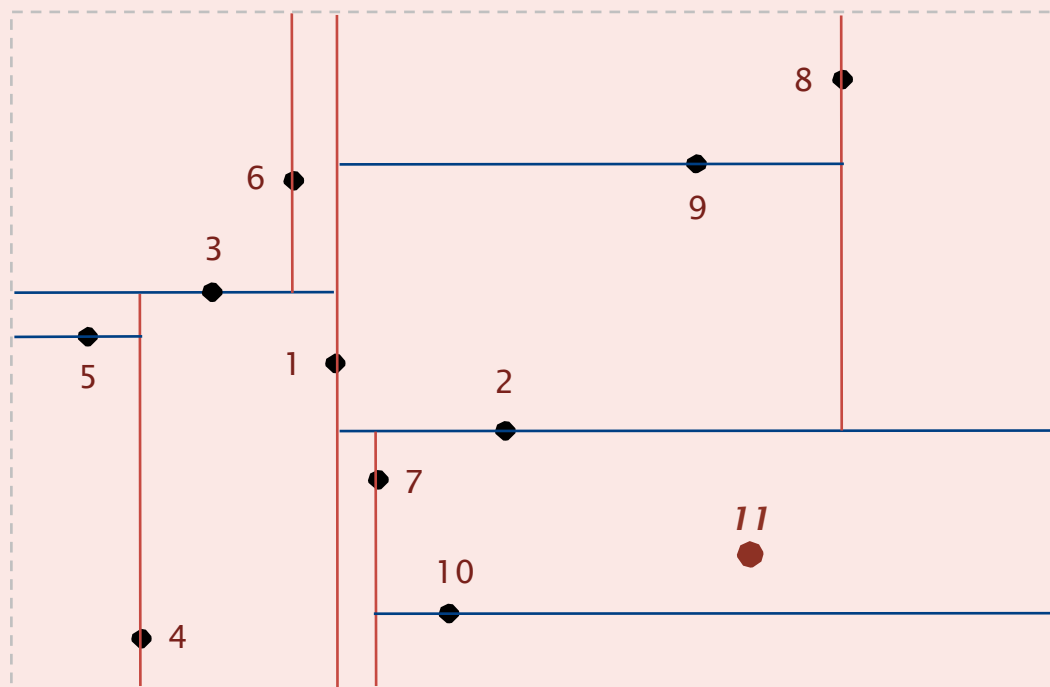
- Search gives rectangle containing point.
- Insert further subdivides the plane.



## Quiz 2

Where would point 11 be inserted in the kd-tree below?

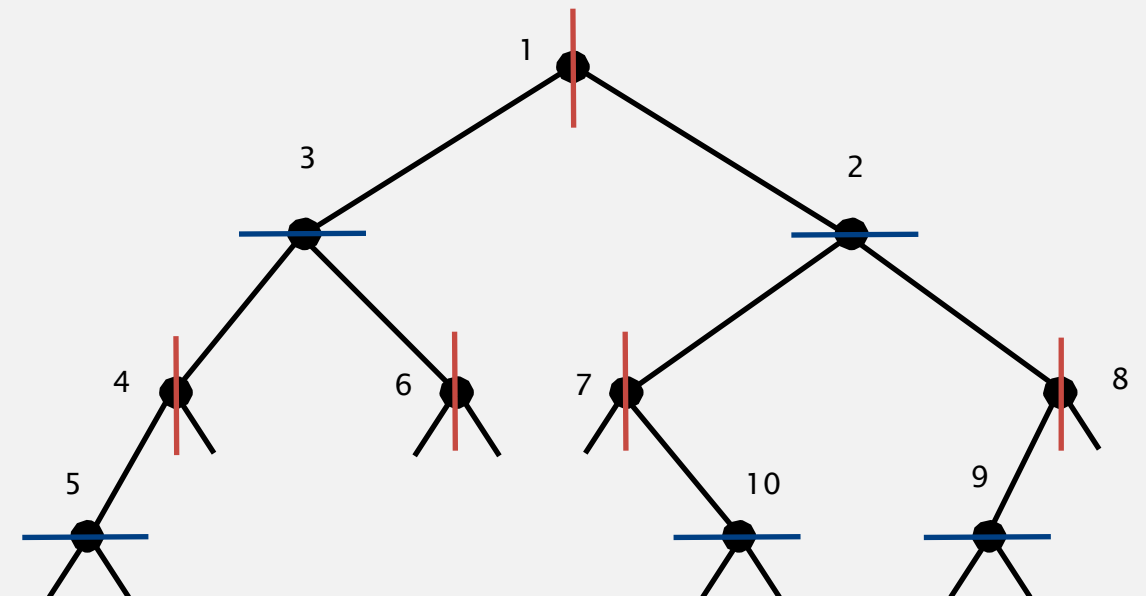
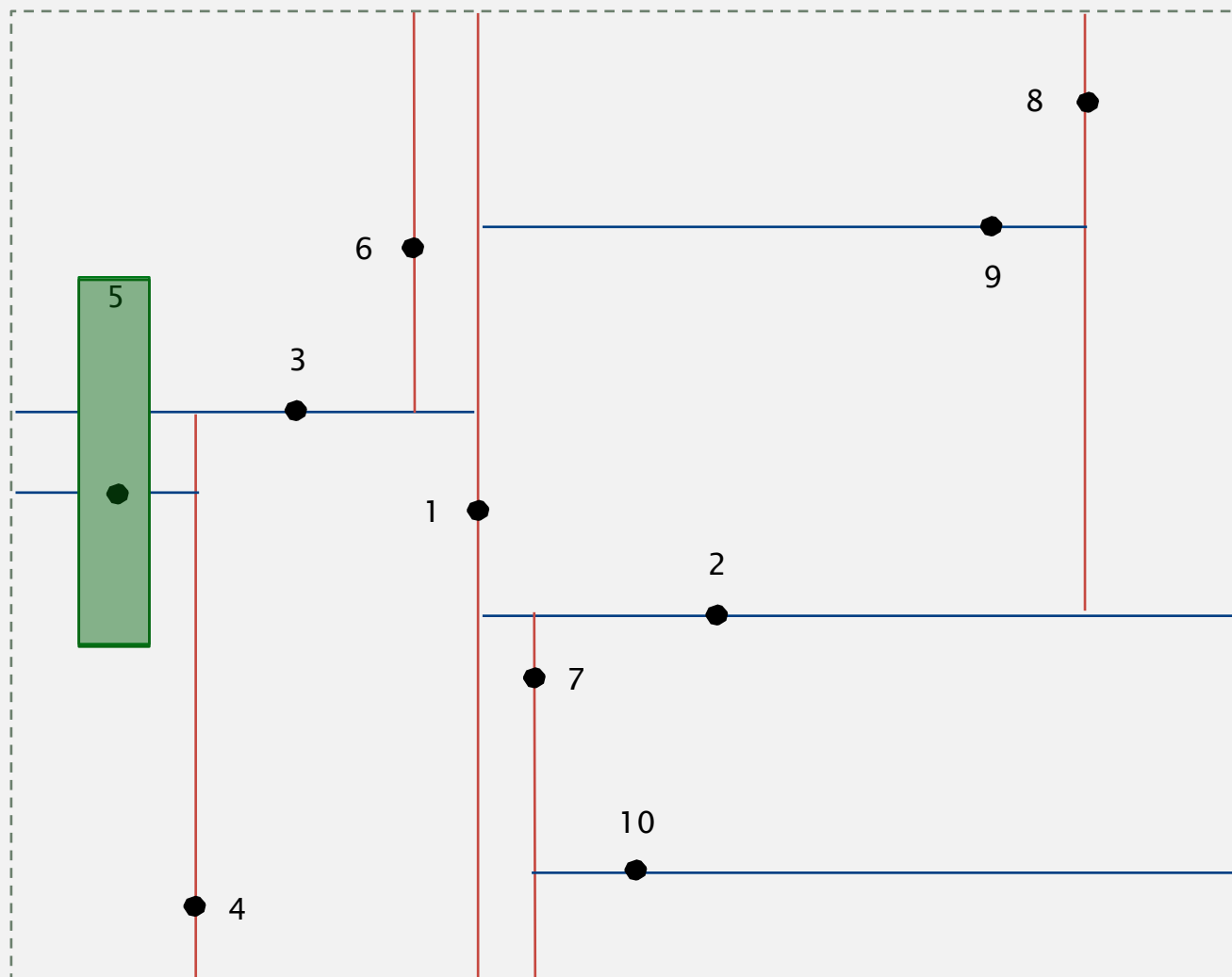
- A. Right child of 6.
- B. Left child of 7.
- C. Left child of 10.
- D. Right child of 10.
- E. *I don't know.*



## 2d tree demo: range search

**Goal.** Find all points in a query axis-aligned rectangle.

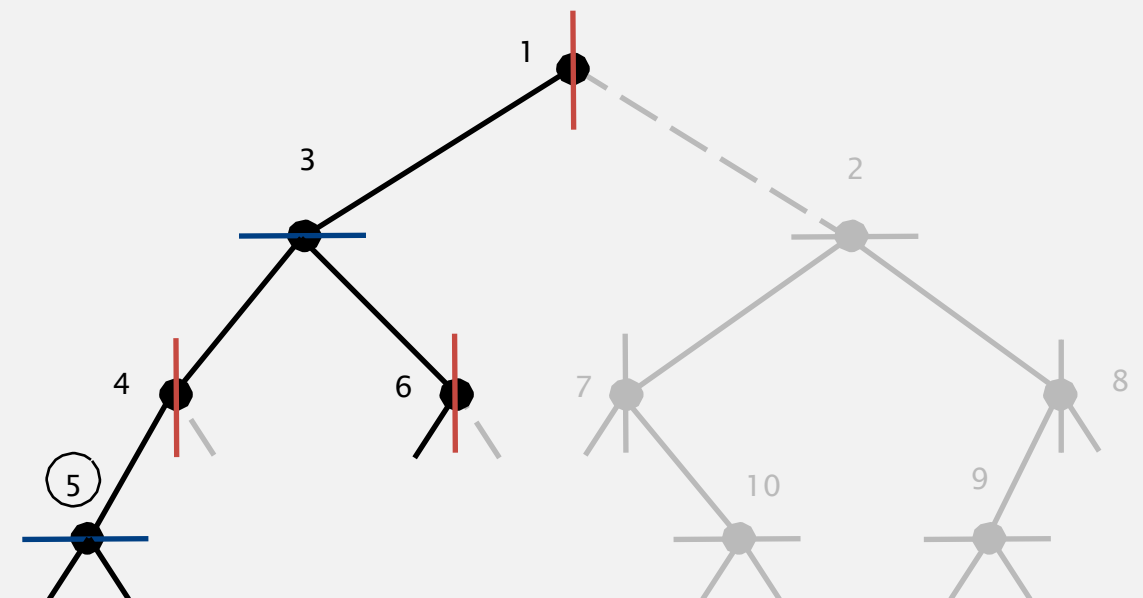
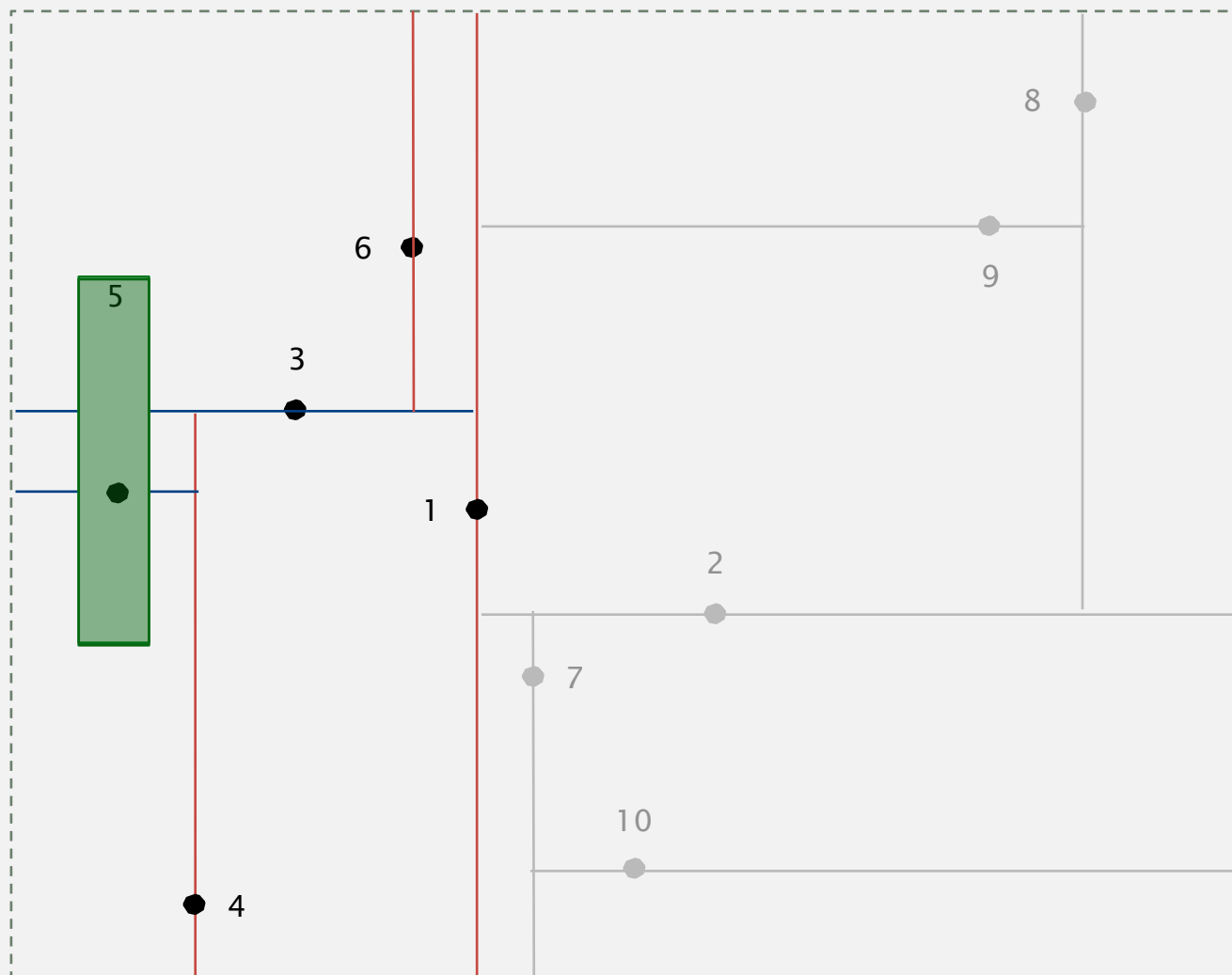
- Check if point in node lies in given rectangle.
- Recursively search left/bottom (if any could fall in rectangle).
- Recursively search right/top (if any could fall in rectangle).



## 2d tree demo: range search

**Goal.** Find all points in a query axis-aligned rectangle.

- Check if point in node lies in given rectangle.
- Recursively search left/bottom (if any could fall in rectangle).
- Recursively search right/top (if any could fall in rectangle).

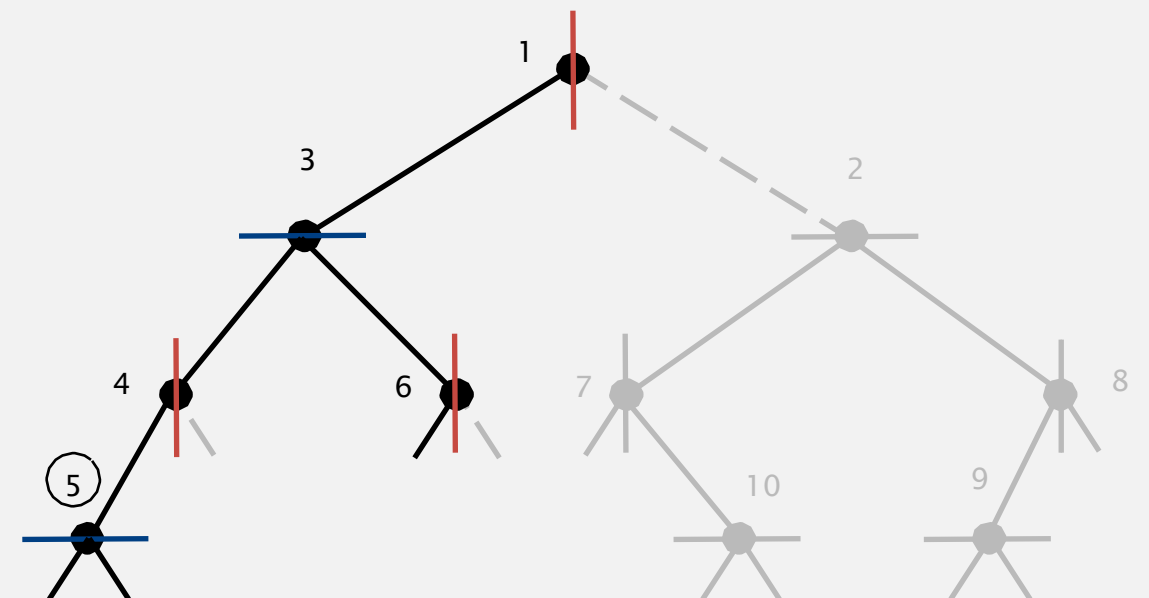
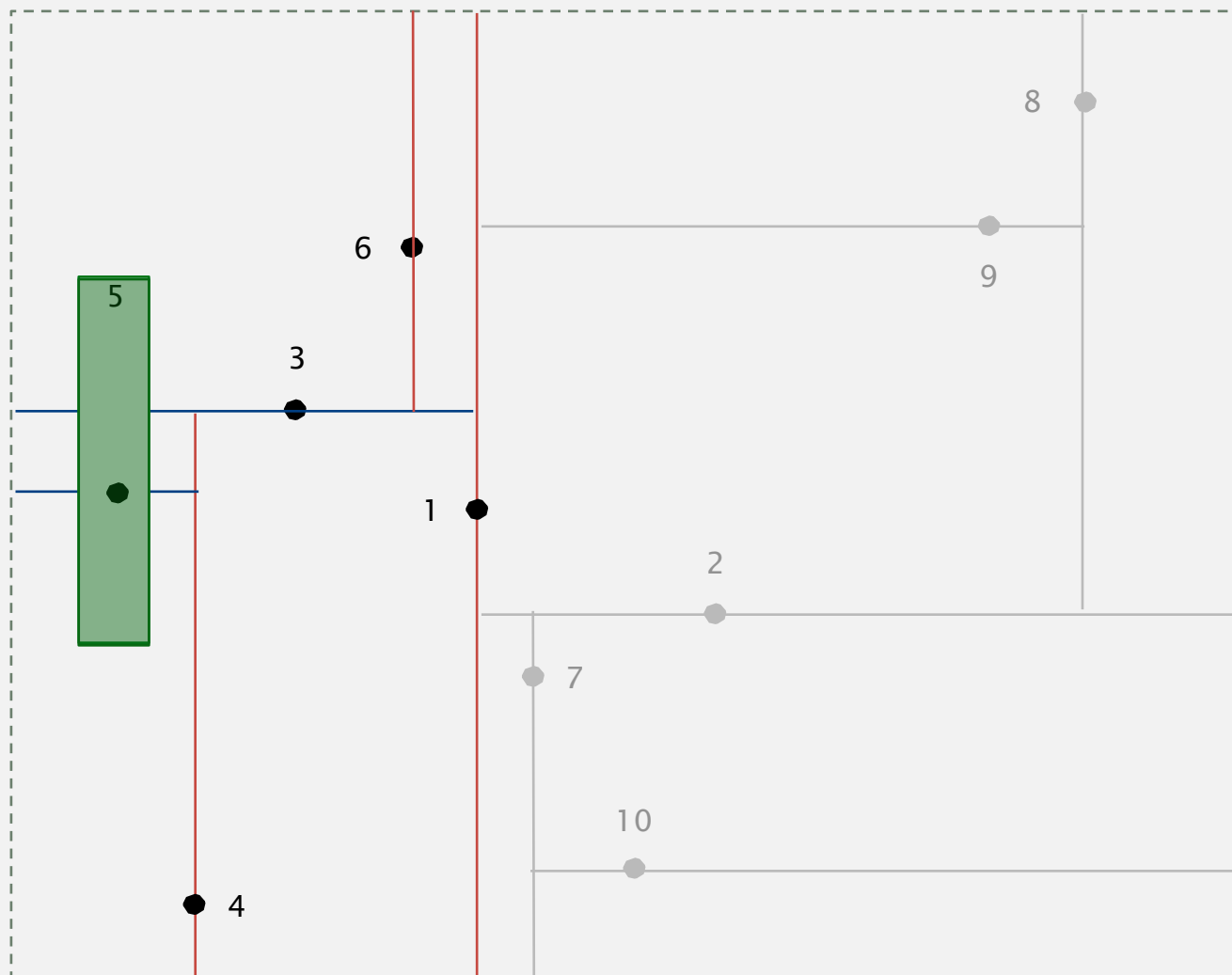


done

# Range search in a 2d tree: analysis

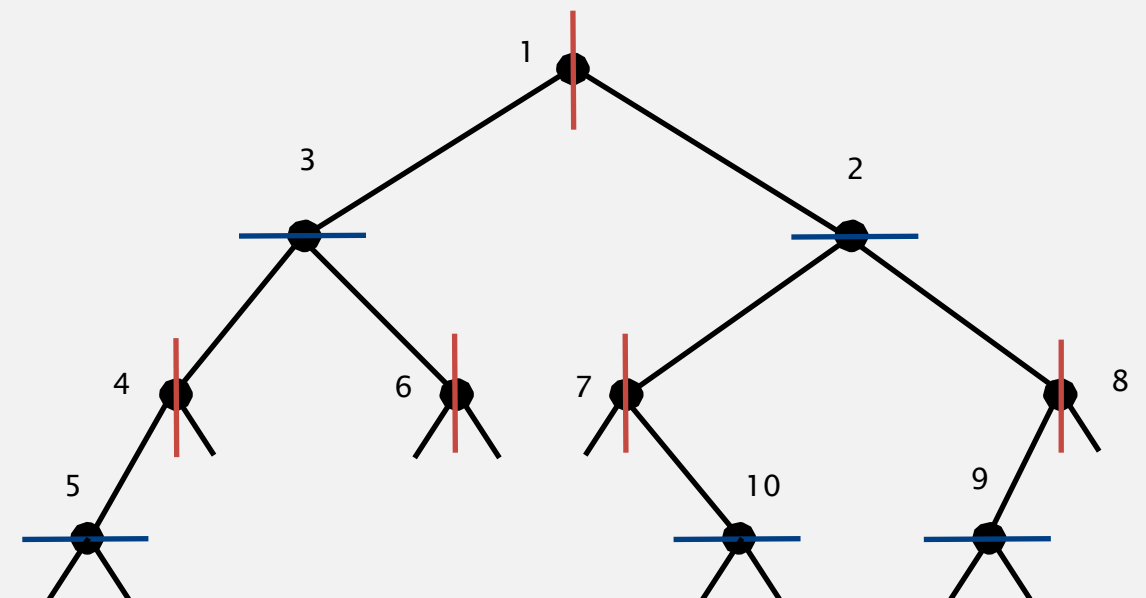
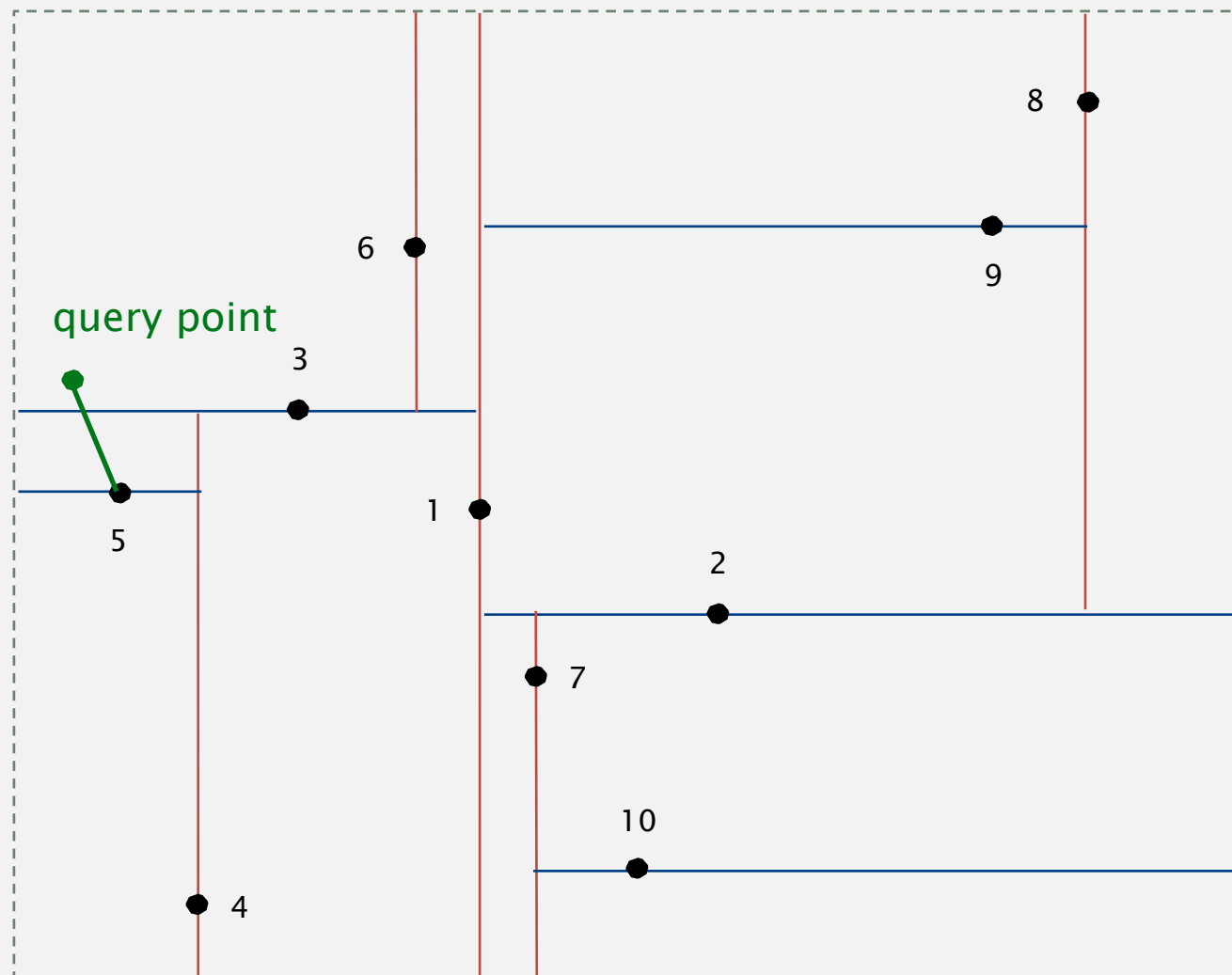
Typical case.  $R + \log N$ .

Worst case (assuming tree is balanced).  $R + \sqrt{N}$ .



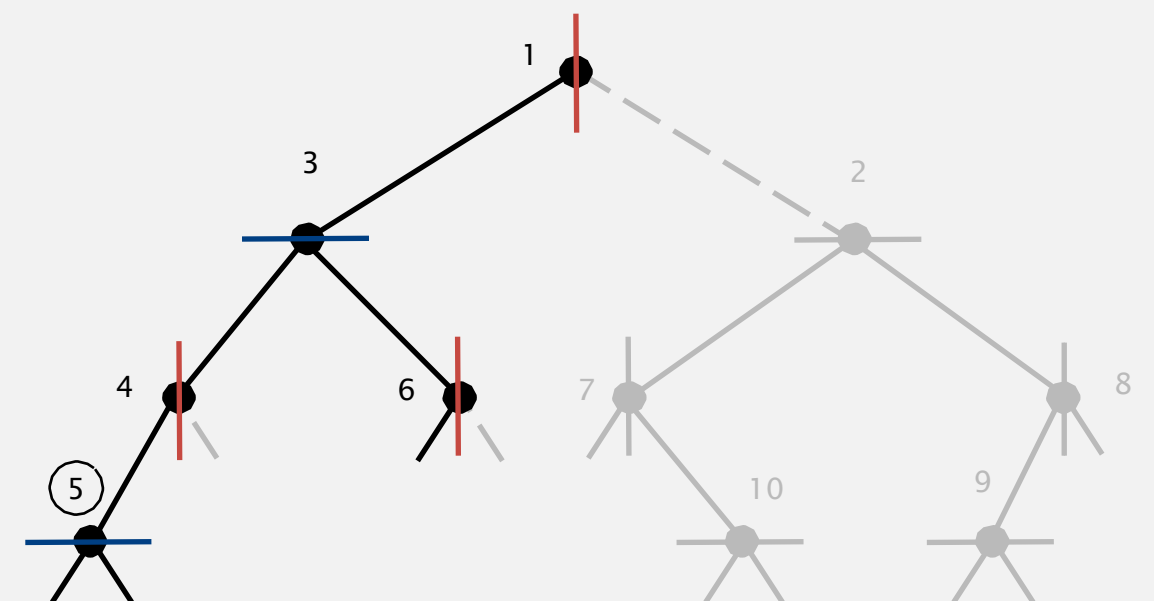
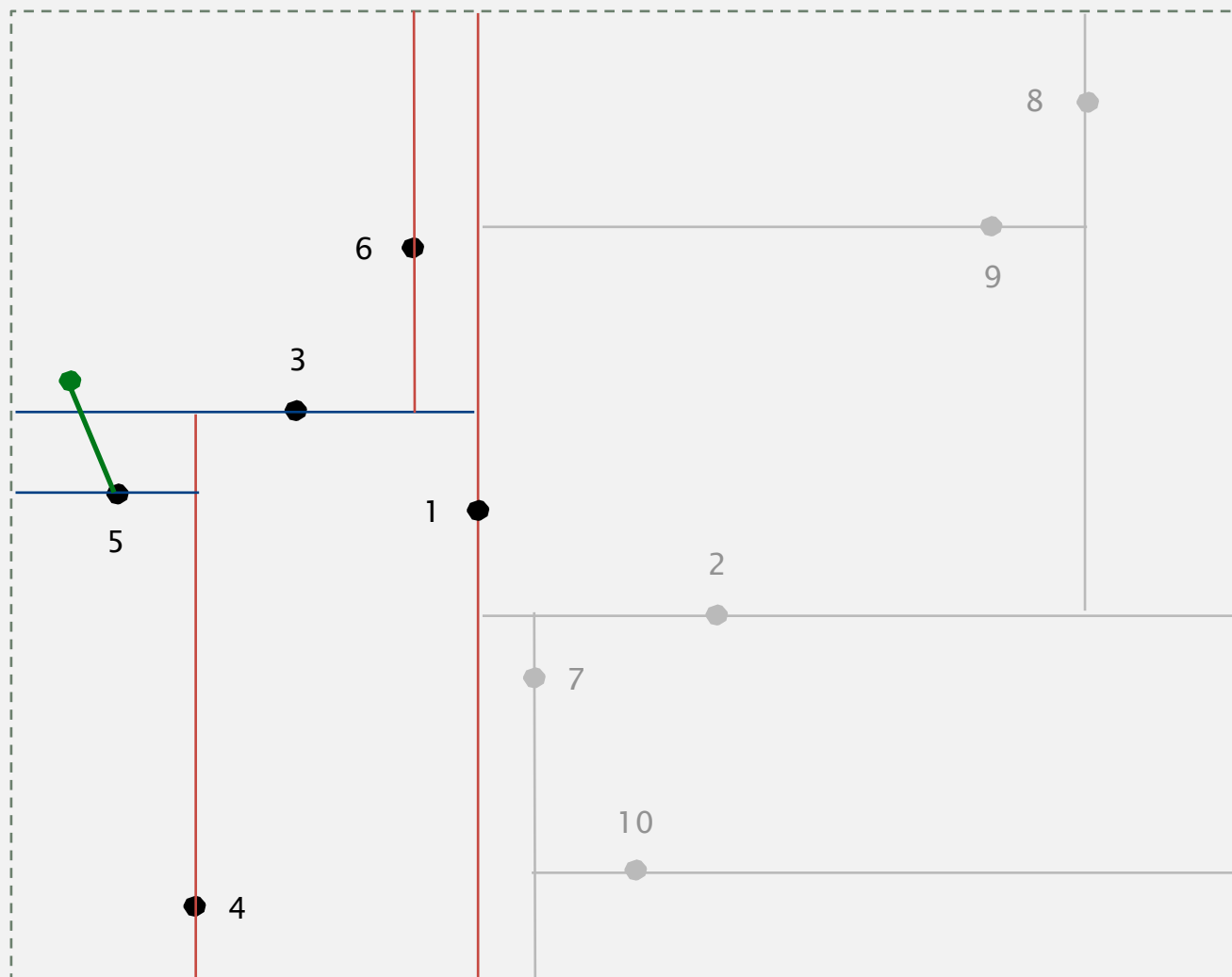
## 2d tree demo: nearest neighbor

**Goal.** Find closest point to query point.



## 2d tree demo: nearest neighbor

- Check distance from point in node to query point.
- Recursively search left/bottom (if it could contain a closer point).
- Recursively search right/top (if it could contain a closer point).
- Organize method so that it begins by searching for query point.



**nearest neighbor = 5**



## Quiz 3

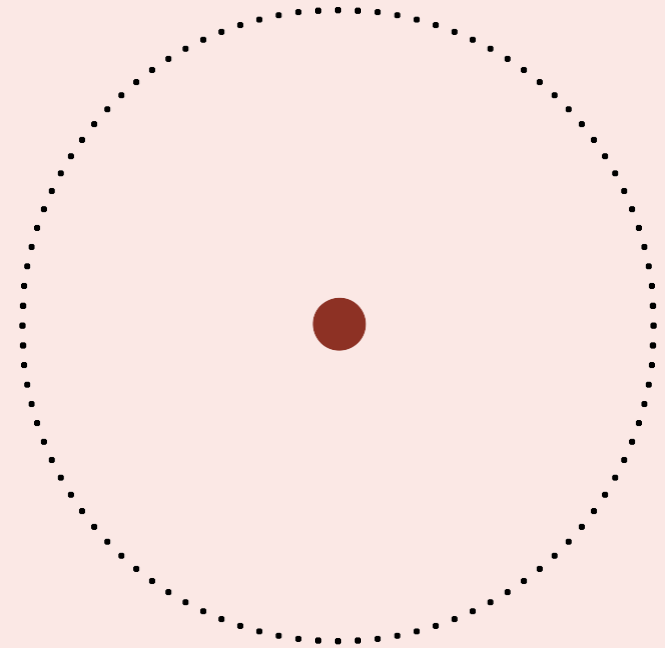
---

Which of the following is the worst case for nearest neighbor search?

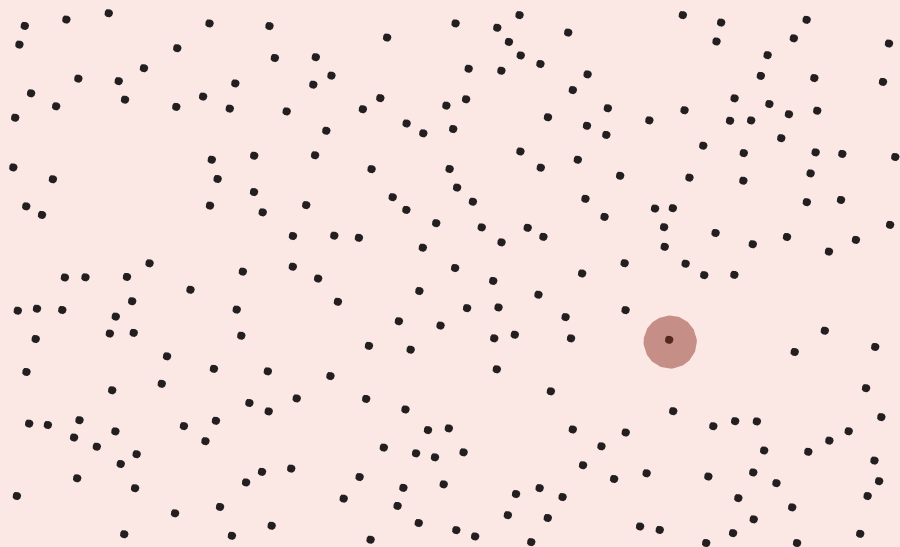
A.



C.



B.

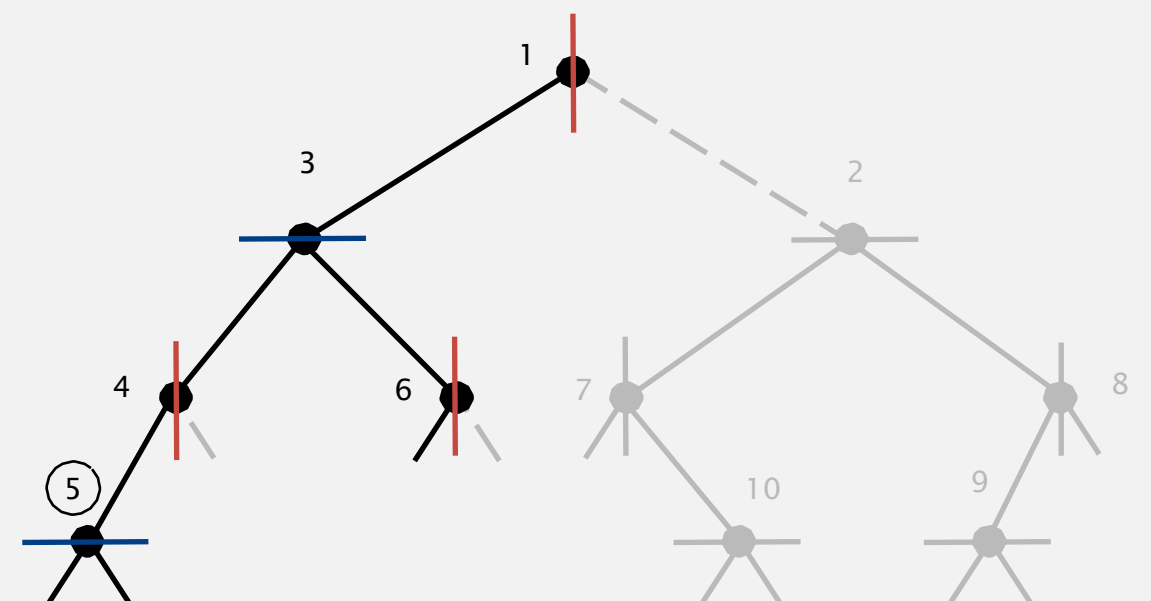
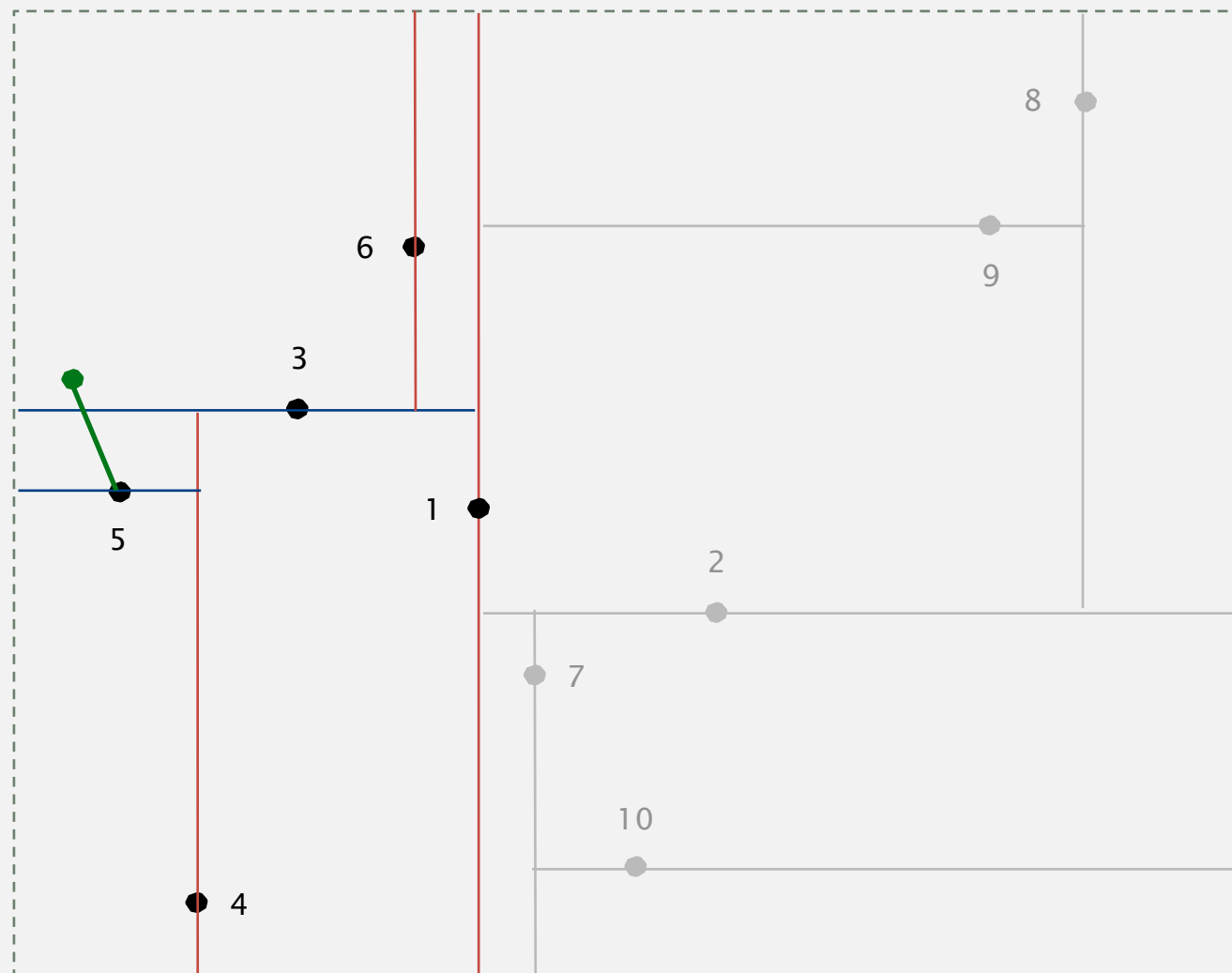


D. *I don't know.*

# Nearest neighbor search in a 2d tree: Analysis

Typical case.  $\log N$ .

Worst case (even if tree is balanced).  $N$ .



nearest neighbor = 5

# Flocking birds

---

Q. What "natural algorithm" do starlings, migrating geese, starlings, cranes, bait balls of fish, and flashing fireflies use to flock?



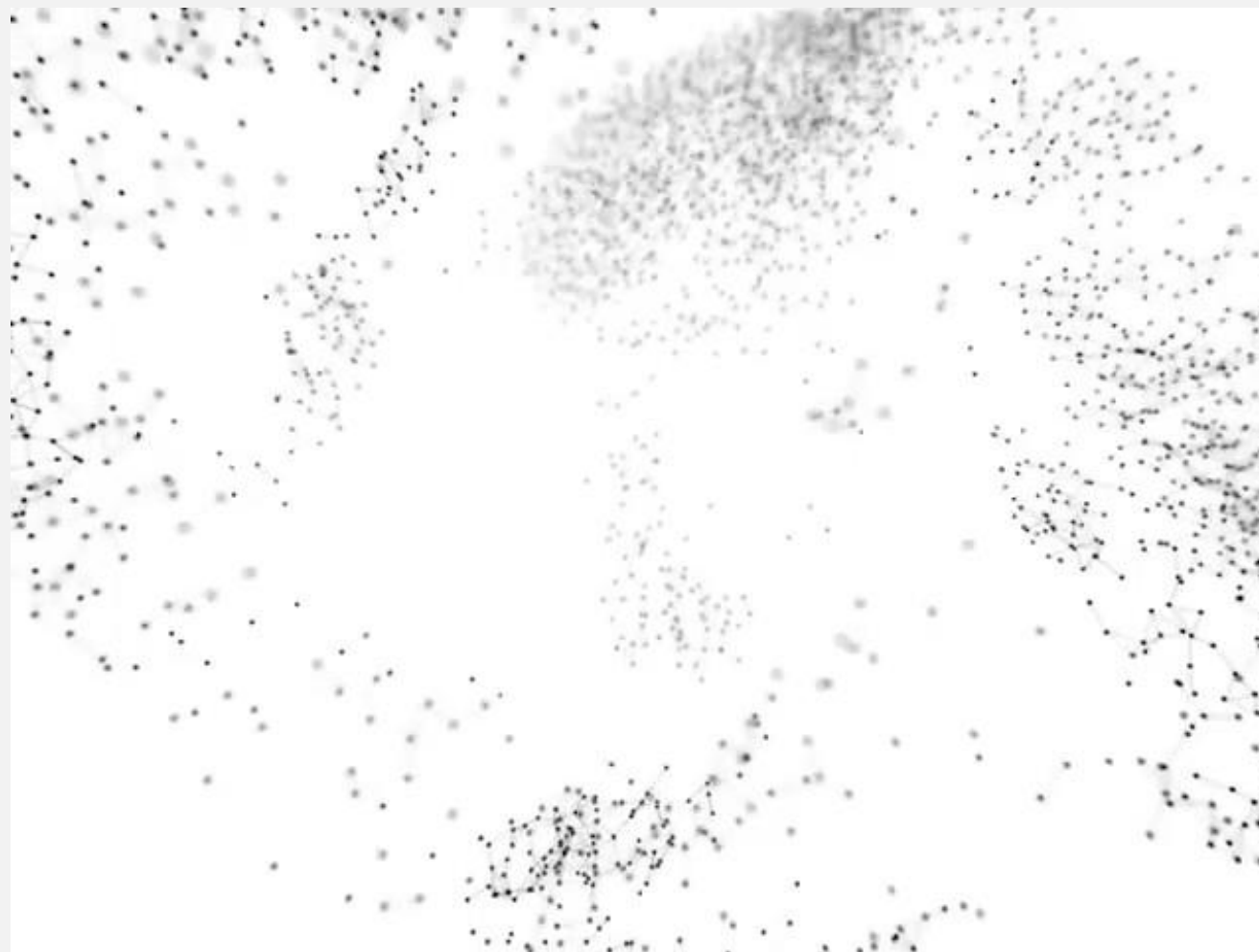
<http://www.youtube.com/watch?v=XH-groCeKbE>

# Flocking boids [Craig Reynolds, 1986]

---

**Boids.** Three simple rules lead to complex emergent flocking behavior:

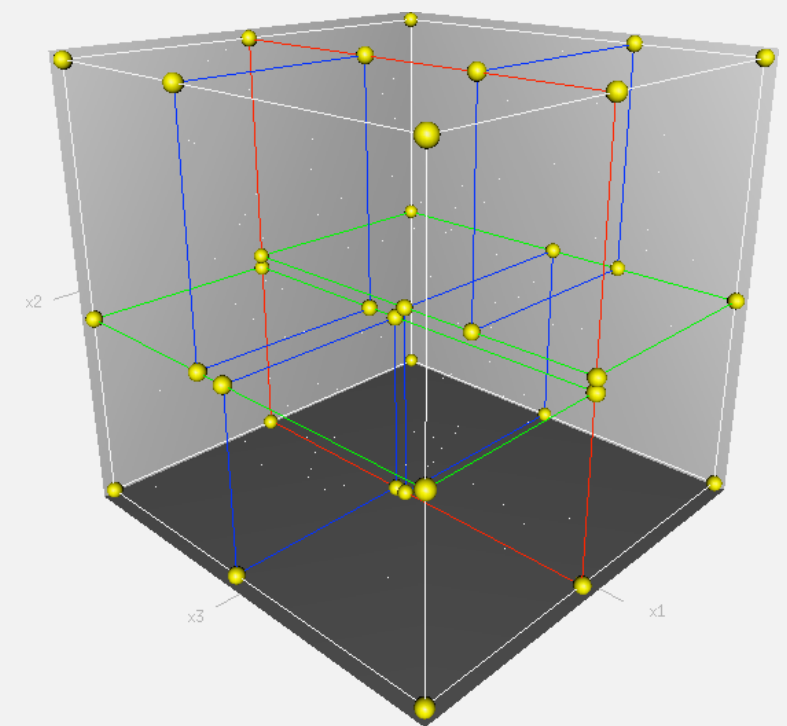
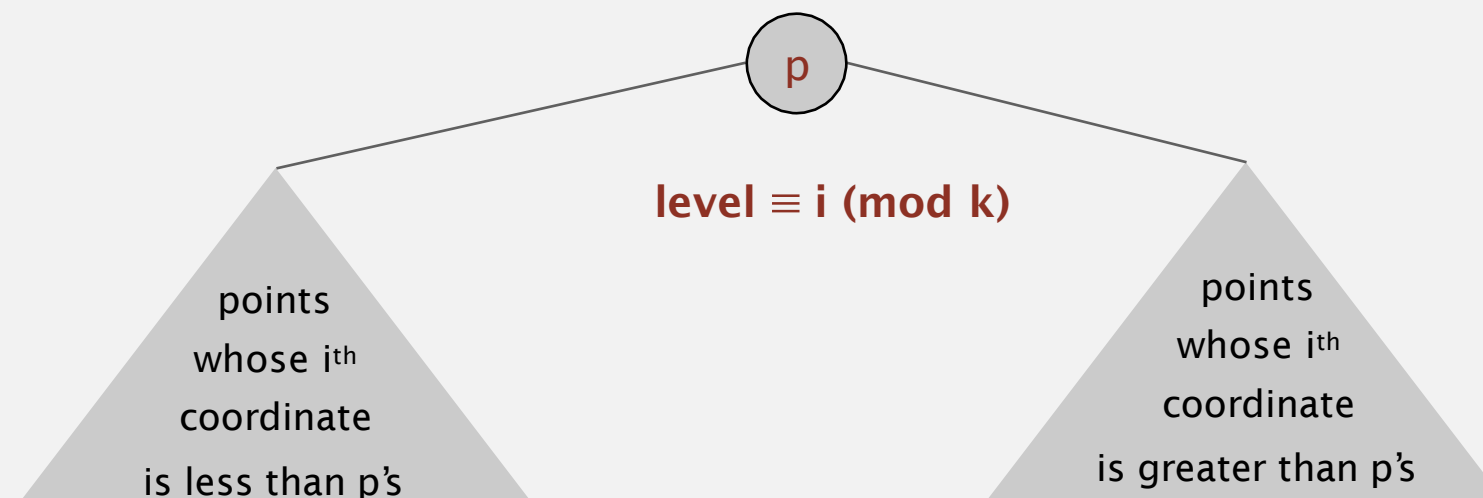
- Collision avoidance: point away from **k nearest** boids.
- Flock centering: point towards the center of mass of **k nearest** boids.
- Velocity matching: update velocity to the average of **k nearest** boids.



# Kd tree

**Kd tree.** Recursively partition  $k$ -dimensional space into 2 halfspaces.

**Implementation.** BST, but cycle through dimensions ala 2d trees.



Efficient, simple data structure for processing  $k$ -dimensional data.

- Widely used.
- Adapts well to high-dimensional and clustered data.
- Discovered by an undergrad in an algorithms class!



Jon Bentley



<http://algs4.cs.princeton.edu>

## GEOMETRIC APPLICATIONS OF BSTs

---

- *1d range search*
- *line segment intersection*
- *kd trees*
- *interval search trees*
- *rectangle intersection*

# 1d interval search

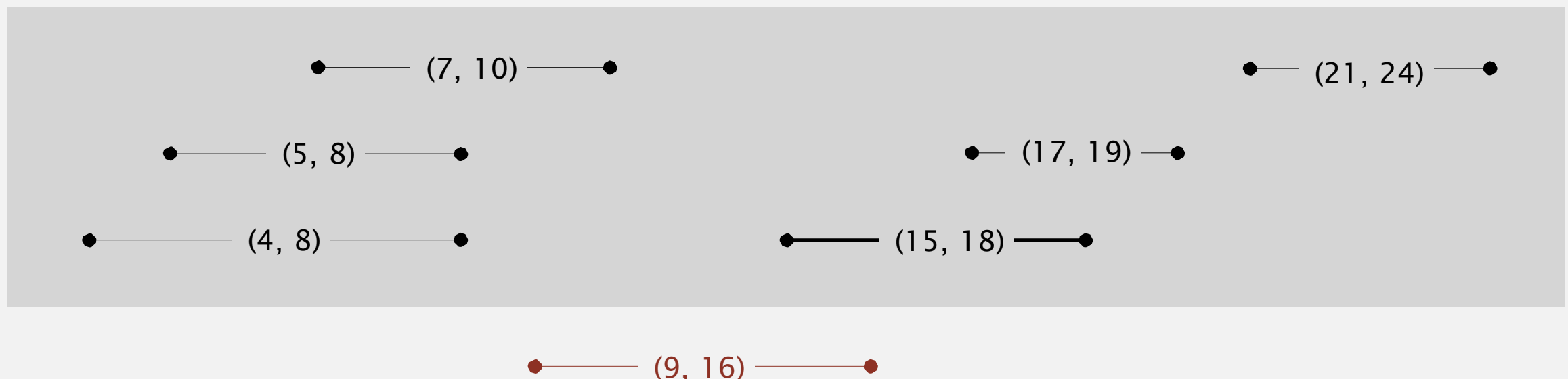
---

**1d interval search.** Data structure to hold set of (overlapping) intervals.

- Insert an interval ( $lo$ ,  $hi$ ).
- Search for an interval ( $lo$ ,  $hi$ ).
- Delete an interval ( $lo$ ,  $hi$ ).
- **Interval intersection query:** given an interval ( $lo$ ,  $hi$ ), find all intervals (or one interval) in data structure that intersects ( $lo$ ,  $hi$ ).

**Q.** Which interval(s) intersect (9, 16) ?

**A.** (7, 10) and (15, 18).





# 1d interval search API

---

```
public class IntervalST<Key extends Comparable<Key>, Value>
```

```
    IntervalST()
```

*create interval search tree*

```
    void put(Key lo, Key hi, Value val)
```

*put interval-value pair into ST*

```
    Value get(Key lo, Key hi)
```

*value paired with given interval*

```
    void delete(Key lo, Key hi)
```

*delete the given interval*

```
    Iterable<Value> intersects(Key lo, Key hi)
```

*all intervals that intersect (lo, hi)*

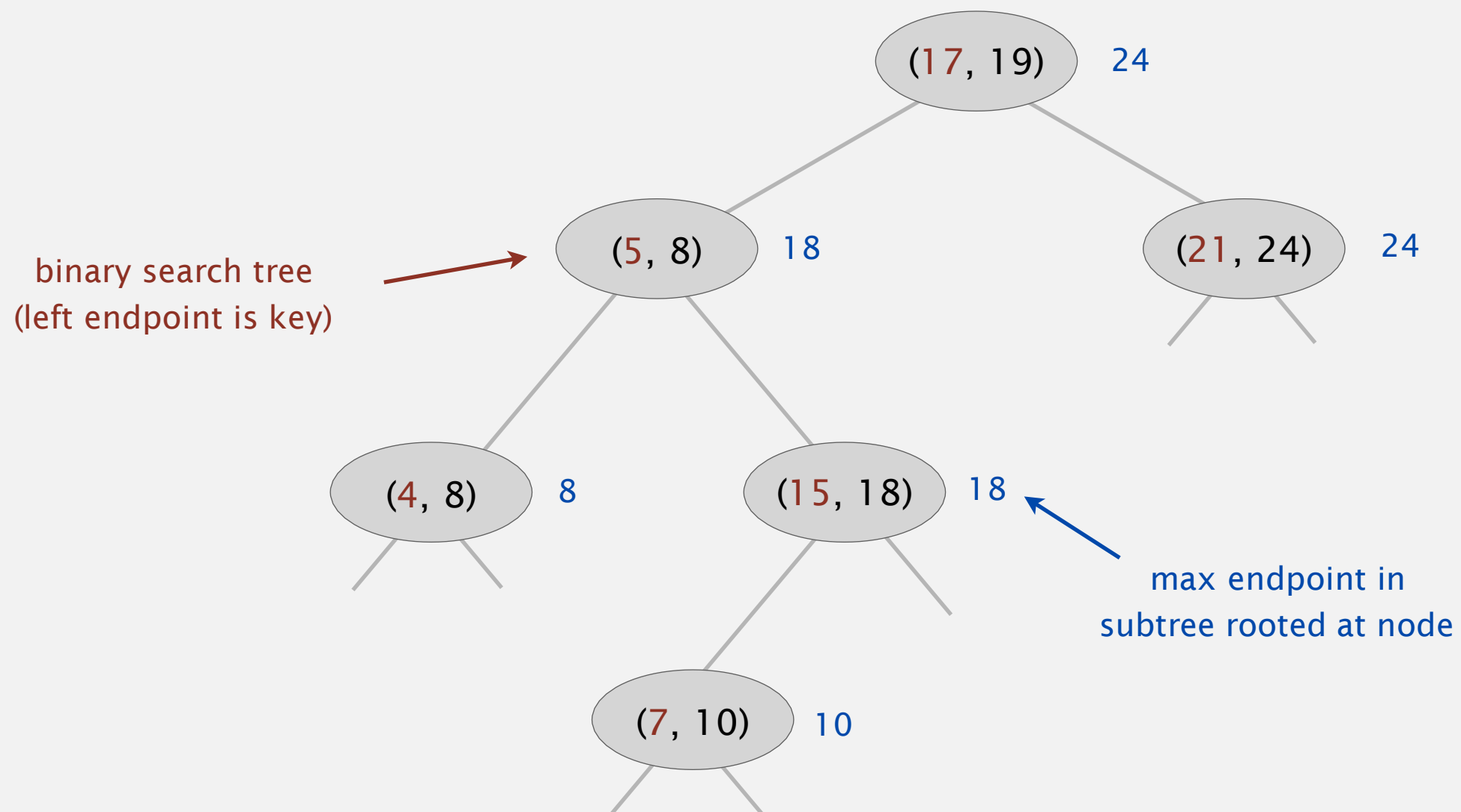
**Nondegeneracy assumption.** No two intervals have the same left endpoint.

# Interval search trees

---

Create BST, where each node stores an interval  $(lo, hi)$ .

- Use left endpoint as BST **key**.
- Store **max endpoint** in subtree rooted at node.



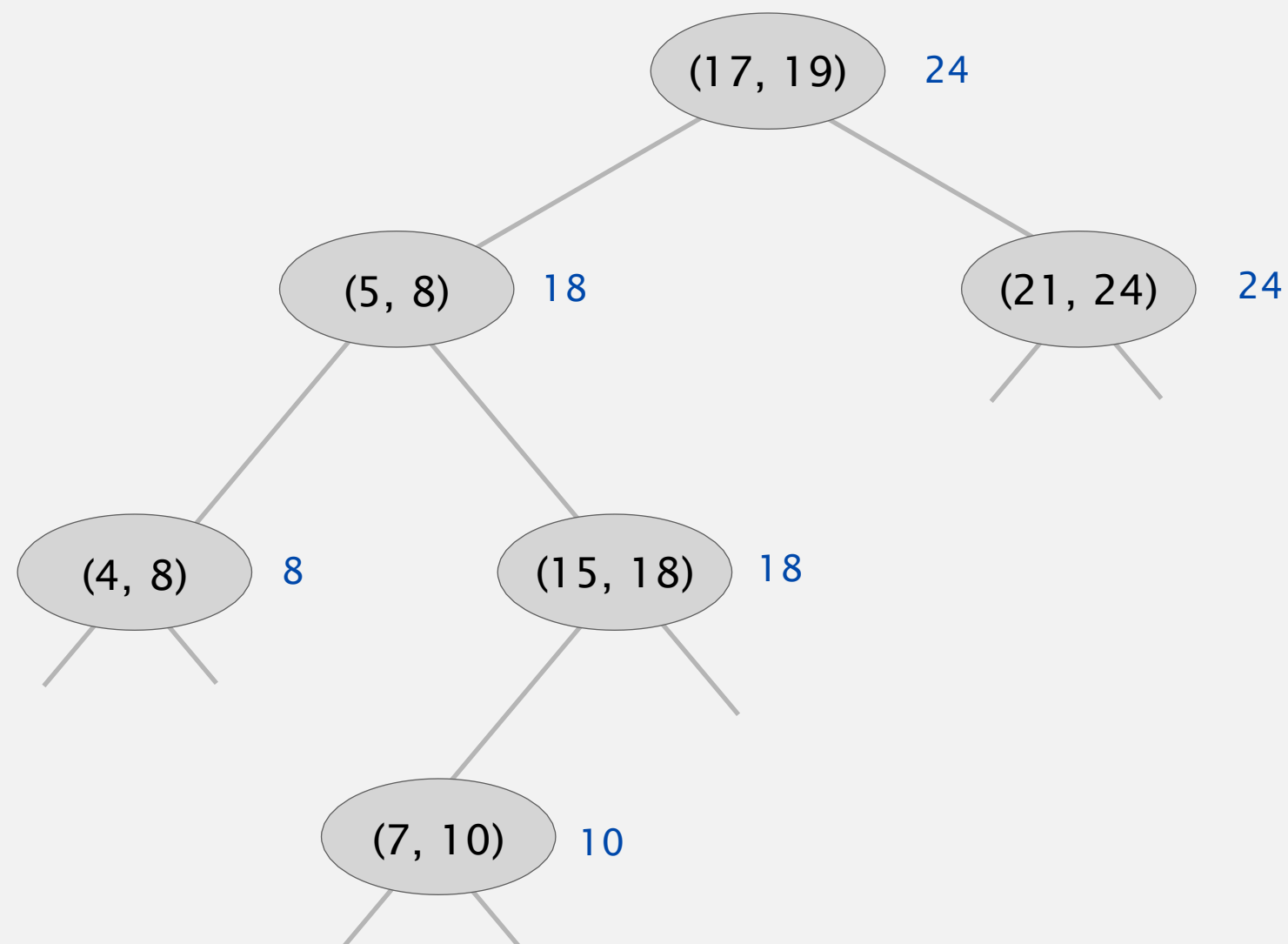
# Interval search tree demo: insertion

To insert an interval  $(lo, hi)$ :

- Insert into BST, using  $lo$  as the key.
- Update max in each node on search path.



insert interval (16, 22)



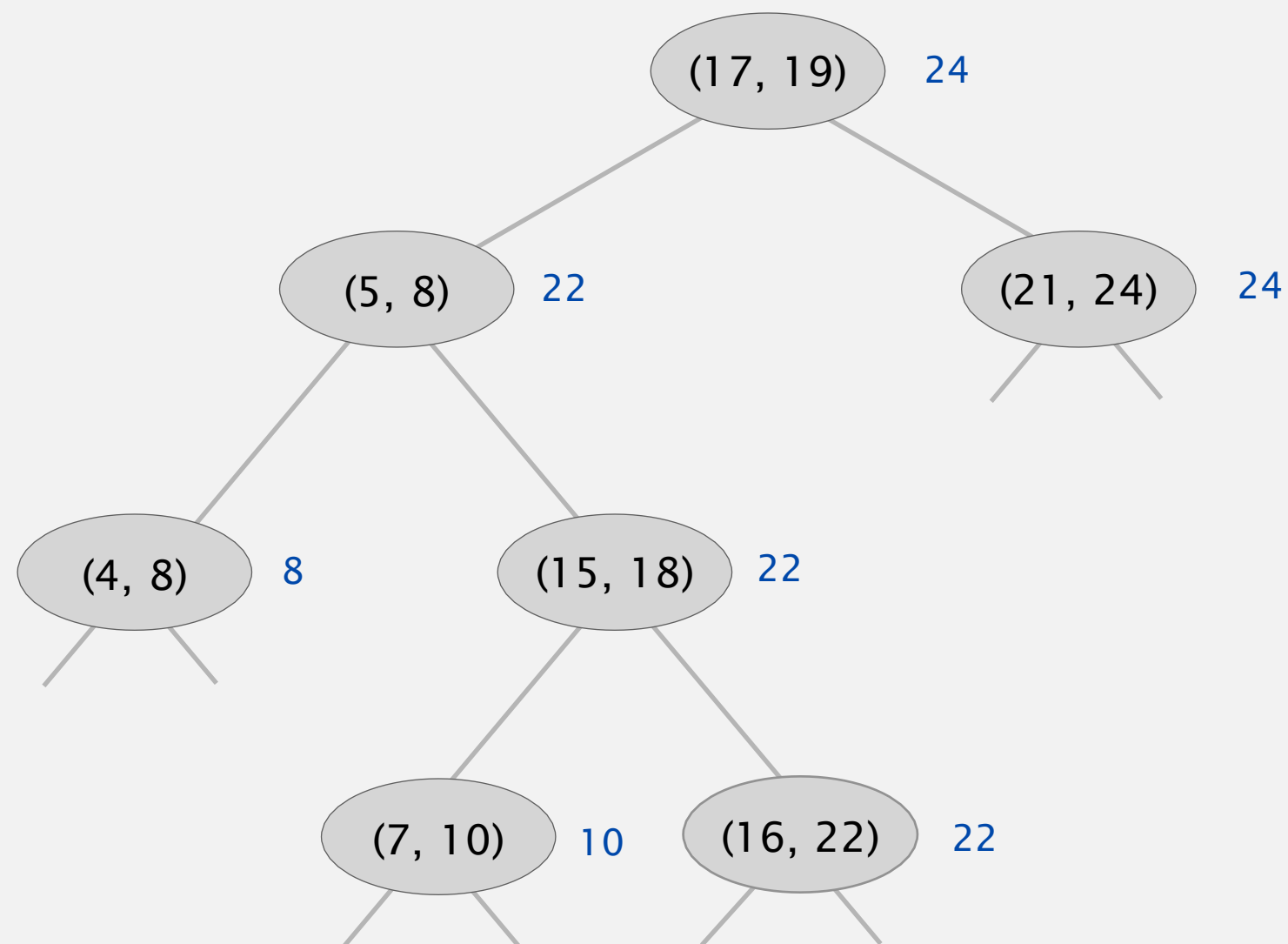
# Interval search tree demo: insertion

To insert an interval  $(lo, hi)$ :

- Insert into BST, using  $lo$  as the key.
- Update max in each node on search path.



insert interval (16, 22)

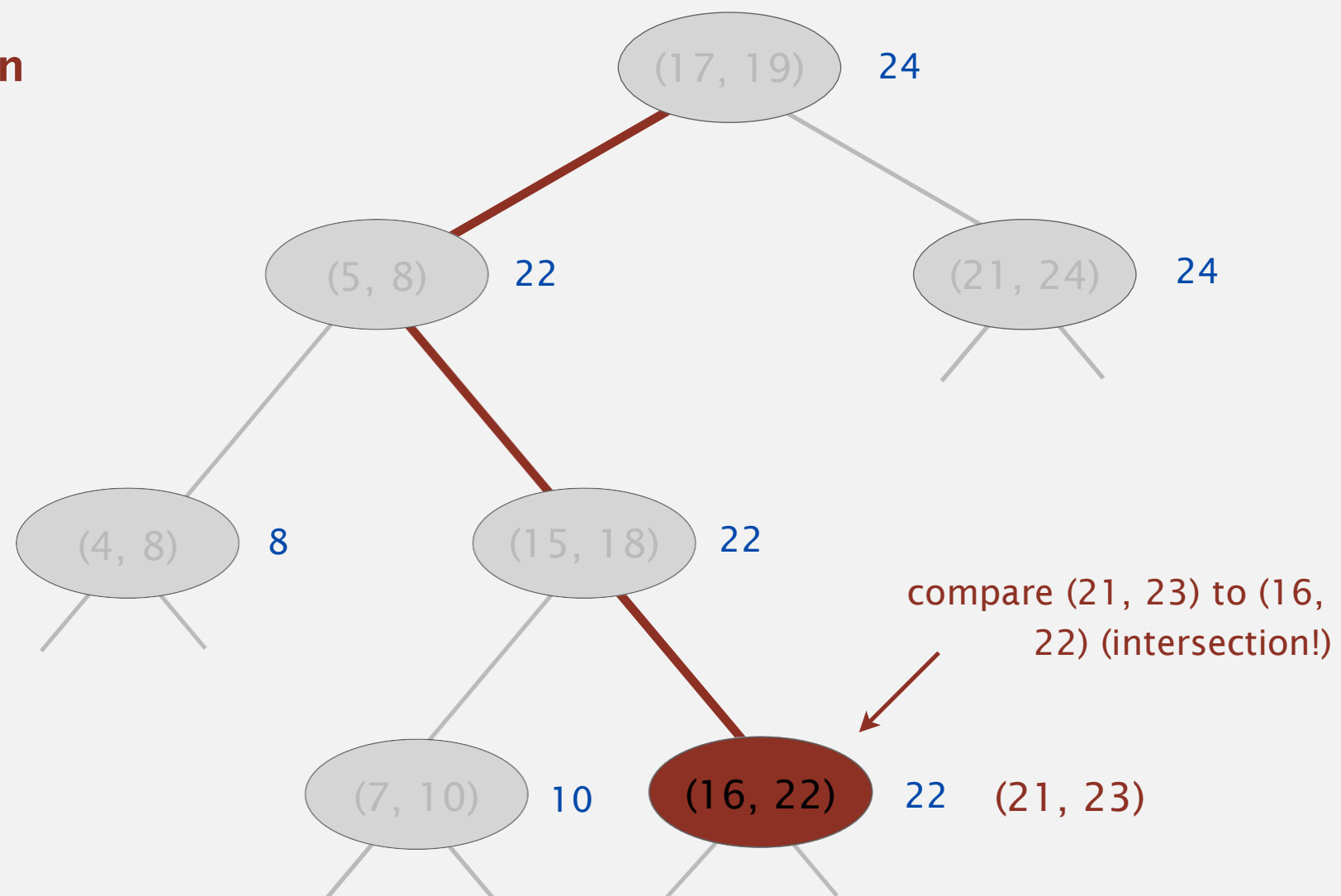


# Interval search tree demo: intersection

To search for any one interval that intersects query interval  $(lo, hi)$ :

- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than  $lo$ , go right.
- Else go left.

**interval intersection**  
**search for (21, 23)**



## Search for an intersecting interval: implementation

---

To search for any one interval that intersects query interval  $(lo, hi)$ :

- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than  $lo$ , go right.
- Else go left.

```
Node x = root;
while (x != null)
{
    if      (x.interval.intersects(lo, hi)) return x.interval;
    else if (x.left == null)                x = x.right;
    else if (x.left.max < lo)                x = x.right;
    else                                     x = x.left;
}
return null;
```

## Search for an intersecting interval: analysis

---

To search for any one interval that intersects query interval  $(lo, hi)$ :

- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than  $lo$ , go right.
- Else go left.

**Case 1.** If search goes **right**, then no intersection in left.

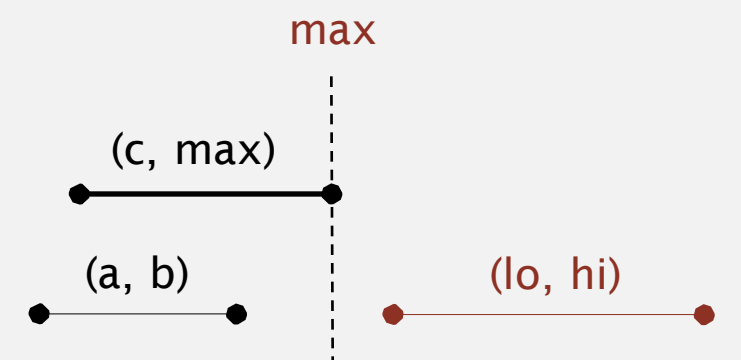
**Pf.** Suppose search goes right and left subtree is non empty.

- Since went right, we have  $max < lo$ .
- For any interval  $(a, b)$  in left subtree of  $x$ ,  
we have  $b \leq max < lo$ .

definition of max

reason for going right

- Thus,  $(a, b)$  will not intersect  $(lo, hi)$ .



left subtree of x

right subtree of x



## Search for an intersecting interval: analysis

To search for any one interval that intersects query interval  $(lo, hi)$ :

- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than  $lo$ , go right.
- Else go left.

**Case 2.** If search goes **left**, then there is either an intersection in left subtree or no intersections in either.

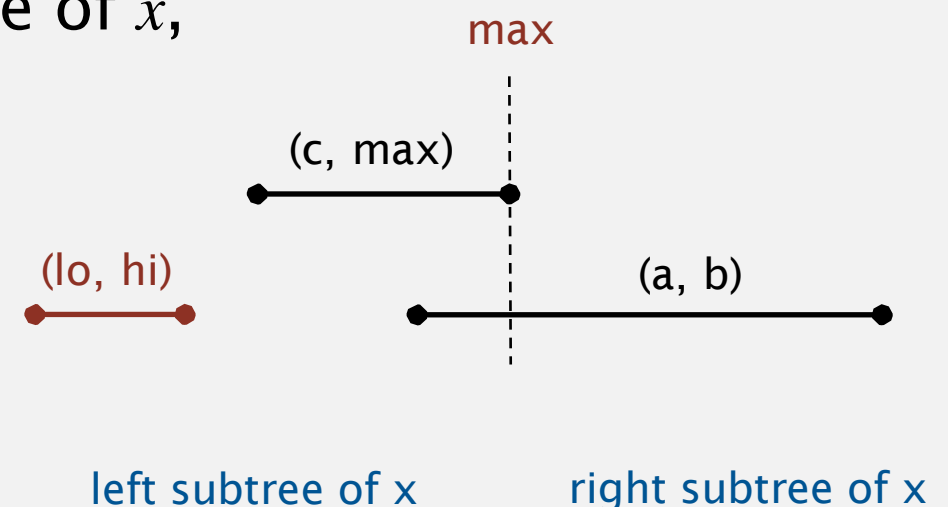
**Pf.** Suppose no intersection in left.

- Since went left, we have  $lo \leq max$ .
- Then for any interval  $(a, b)$  in right subtree of  $x$ ,

$hi \leq c \leq a \Rightarrow$  no intersection in right.

no intersections  
in left subtree

intervals sorted  
by left endpoint



# Interval search tree: analysis

---

**Implementation.** Use a **red-black BST** to guarantee performance.

easy to maintain auxiliary information  
(log N extra work per operation)

operation	brute	BST	interval search tree	best in theory
insert interval	$N$	$\log N$	$\log N$	$\log N$
find interval	$N$	$\log N$	$\log N$	$\log N$
delete interval	$N$	$\log N$	$\log N$	$\log N$
find <b>any one</b> interval that intersects (lo, hi)	$N$	$N$	$\log N$	$\log N$
find <b>all</b> intervals that intersects (lo, hi)	$N$	$N$	$R \log N$	$R + \log N$

order of growth of running time for data structure with N intervals



<http://algs4.cs.princeton.edu>

## GEOMETRIC APPLICATIONS OF BSTs

---

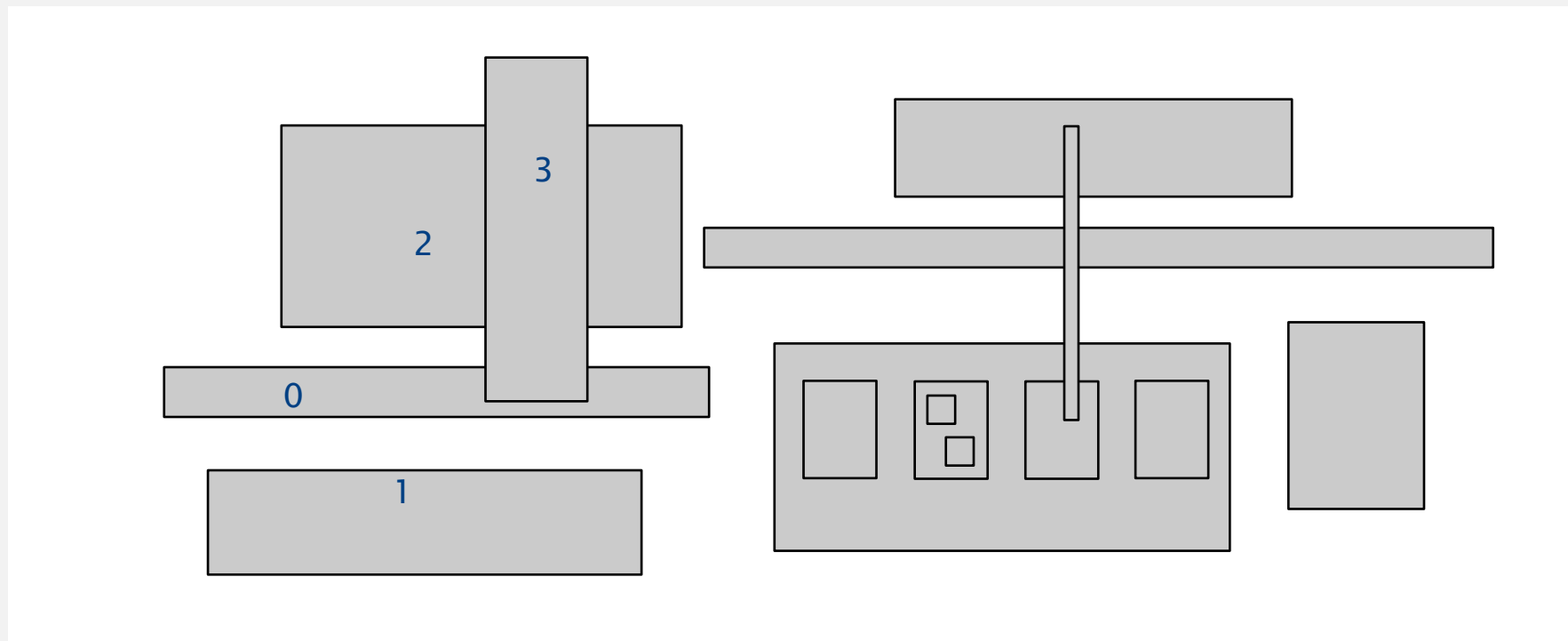
- *1d range search*
- *line segment intersection*
- *kd trees*
- *interval search trees*
- ***rectangle intersection***

# Orthogonal rectangle intersection:

---

**Goal.** Find all intersections among a set of  $N$  orthogonal rectangles.

**Quadratic algorithm.** Check all pairs of rectangles for intersection.

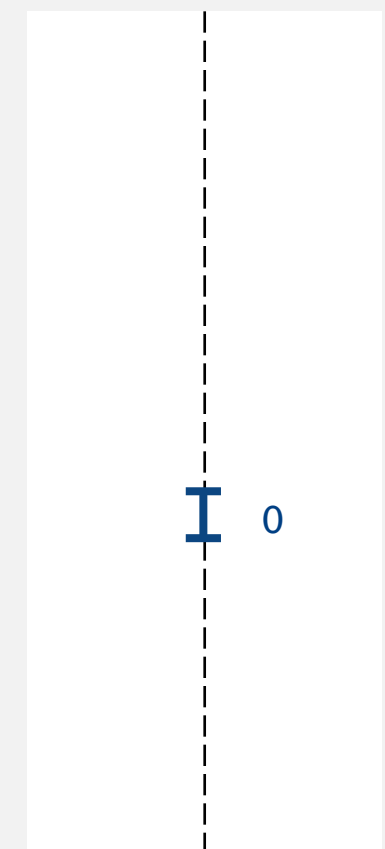
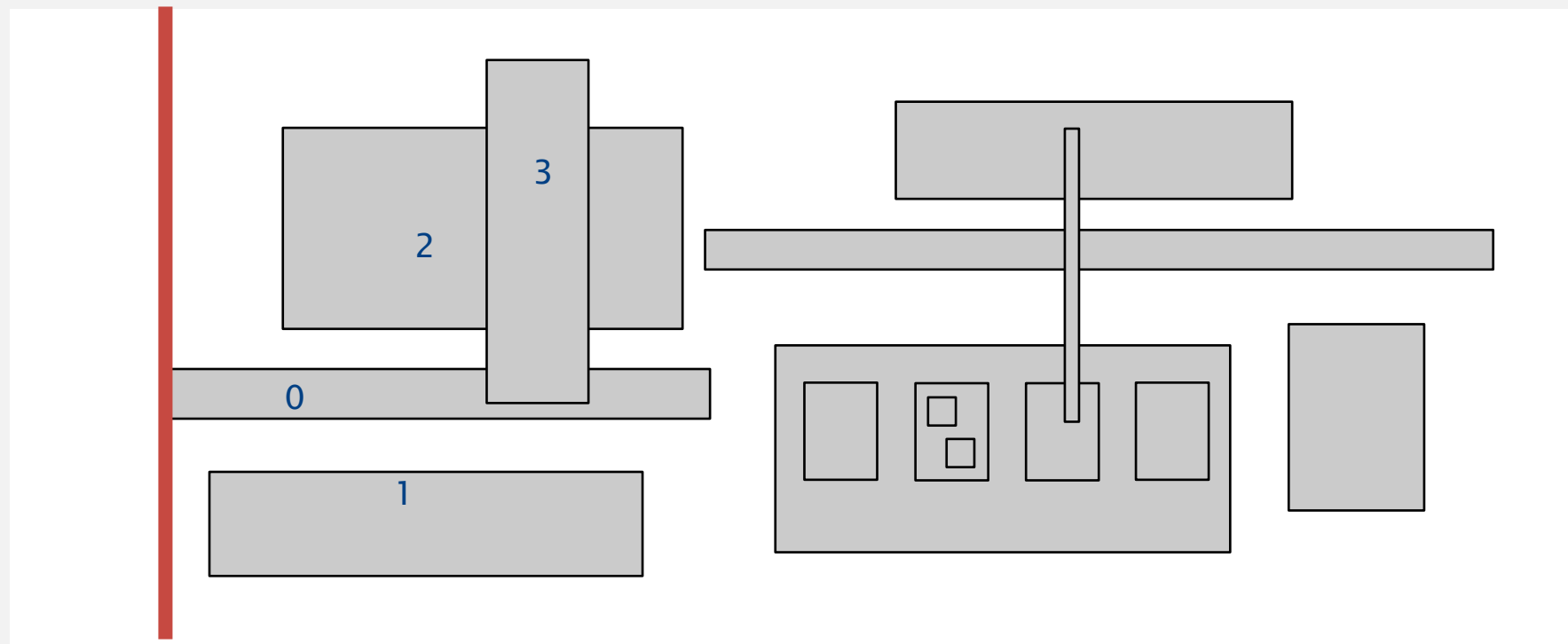


**Non-degeneracy assumption.** All  $x$ - and  $y$ -coordinates are distinct.

# Orthogonal rectangle intersection: sweep-line analysis

Sweep vertical line from left to right.

- $x$ -coordinates of left and right endpoints define events.
- Maintain set of rectangles that intersect the sweep line in an interval search tree (using  $y$ -intervals of rectangle).
- Left endpoint: interval search for  $y$ -interval of rectangle; insert  $y$ -interval.
- Right endpoint: remove  $y$ -interval.

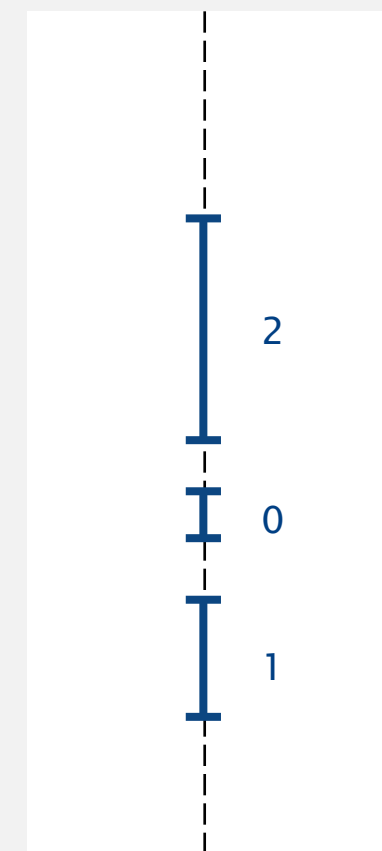
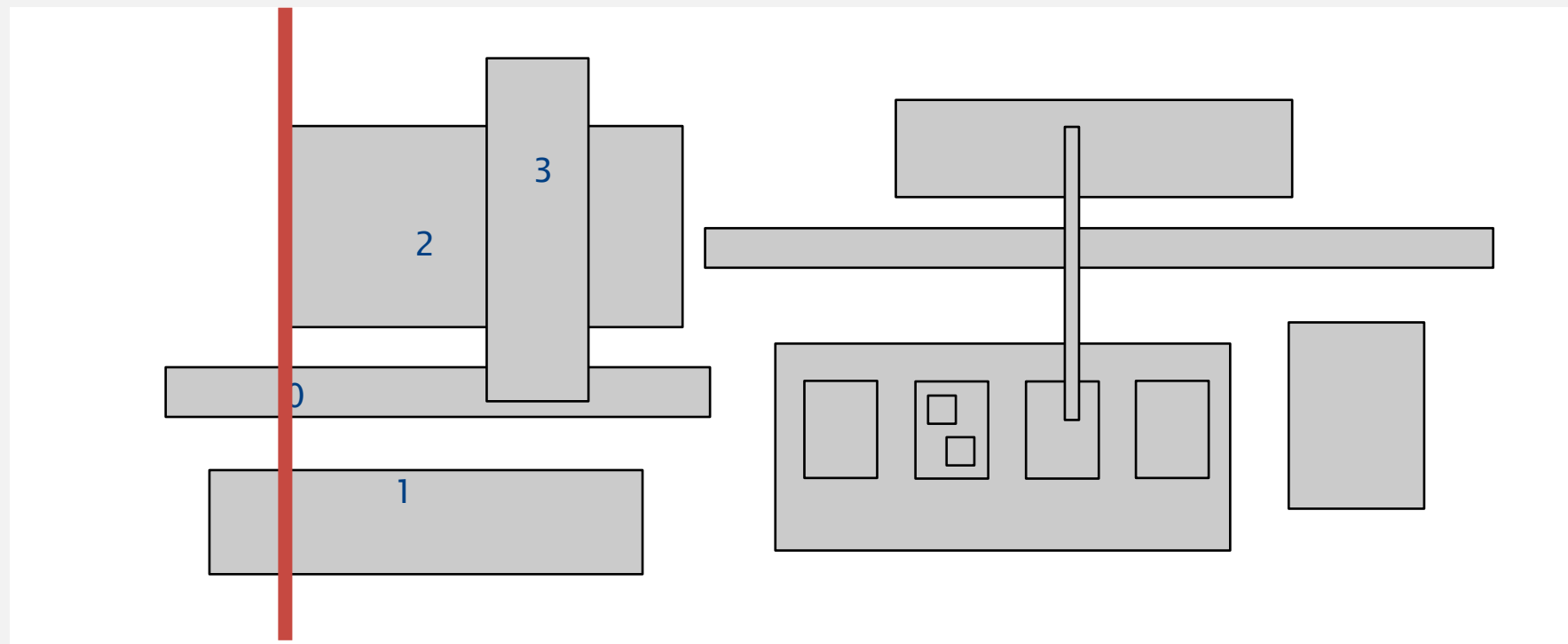


y - c o o r d i n a t e s

# Orthogonal rectangle intersection: sweep-line analysis

Sweep vertical line from left to right.

- $x$ -coordinates of left and right endpoints define events.
- Maintain set of rectangles that intersect the sweep line in an interval search tree (using  $y$ -intervals of rectangle).
- Left endpoint: interval search for  $y$ -interval of rectangle; insert  $y$ -interval.
- Right endpoint: remove  $y$ -interval.

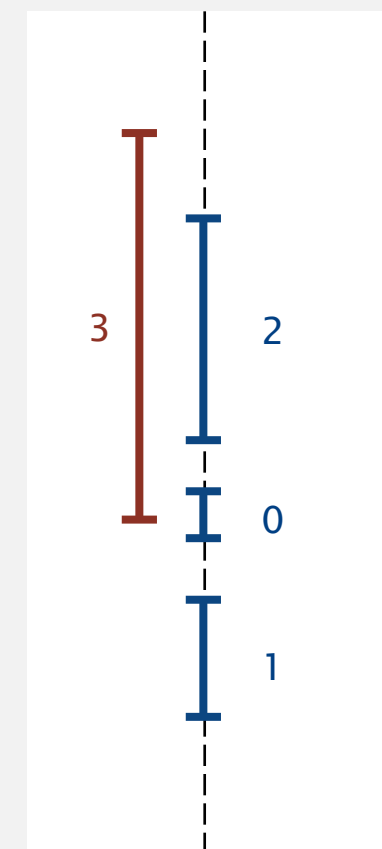
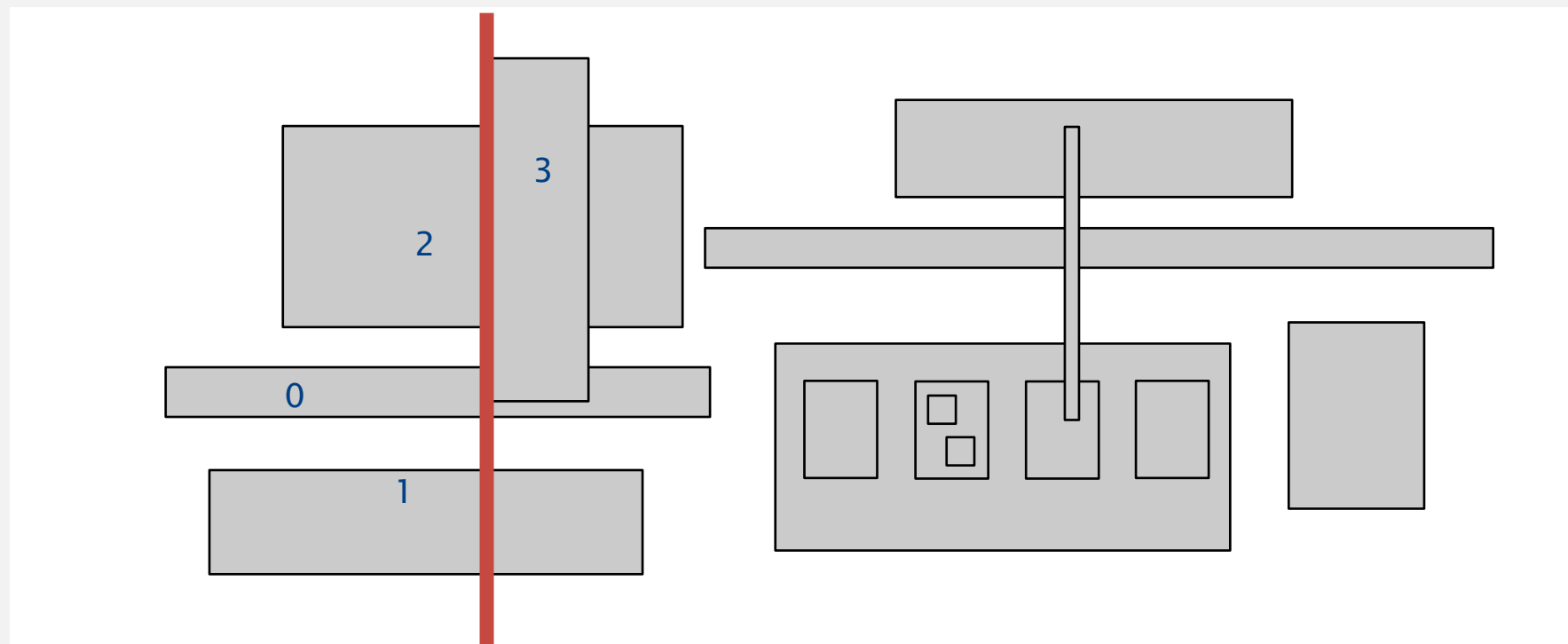


y - c coordinates

# Orthogonal rectangle intersection: sweep-line analysis

Sweep vertical line from left to right.

- $x$ -coordinates of left and right endpoints define events.
- Maintain set of rectangles that intersect the sweep line in an interval search tree (using  $y$ -intervals of rectangle).
- Left endpoint: interval search for  $y$ -interval of rectangle; insert  $y$ -interval.
- Right endpoint: remove  $y$ -interval.



y - c coordinates

# Orthogonal rectangle intersection: sweep-line analysis

---

**Proposition.** Sweep line algorithm takes time proportional to  $N \log N + R \log N$  to find  $R$  intersections among a set of  $N$  rectangles.


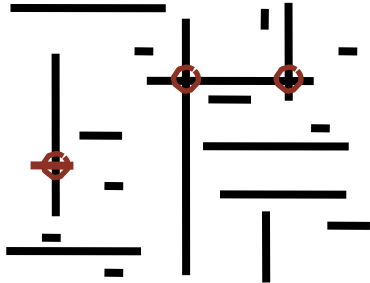
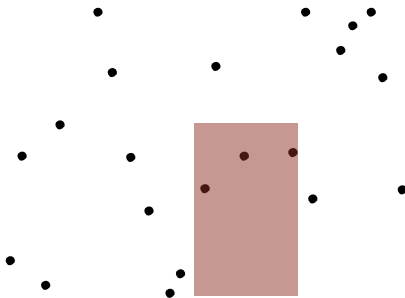

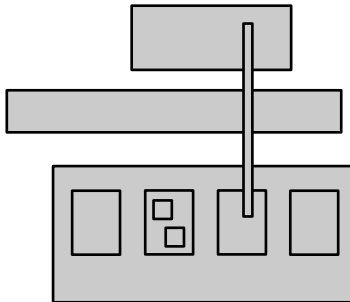
**Pf.**

- Put  $x$ -coordinates on a PQ (or sort).  $\longleftarrow N \log N$
- Insert  $y$ -intervals into ST.  $\longleftarrow N \log N$
- Delete  $y$ -intervals from ST.  $\longleftarrow N \log N$
- Interval searches for  $y$ -intervals.  $\longleftarrow N \log N + R \log N$

**Bottom line.** Sweep line reduces 2d orthogonal rectangle intersection search to 1d interval search.



# Geometric applications of BSTs

problem	example	solution
1d range search		<i>BST</i>
2d orthogonal line segment intersection		<i>sweep line reduces problem to 1d range search</i>
2d range search kd range search		<i>2d tree</i> <i>kd tree</i>
1d interval search		<i>interval search tree</i>
2d orthogonal rectangle intersection		<i>sweep line reduces problem to 1d interval search</i>



<http://algs4.cs.princeton.edu>

## GEOMETRIC APPLICATIONS OF BSTs

---

- *1d range search*
- *line segment intersection*
- *kd trees*
- *interval search trees*
- *rectangle intersection*

▸ **Assignment S3**

# Programming Assignment S3: kd-Trees

---

See how 2d-trees actually work by

- implementing them
- comparing them with non-geometric implementations

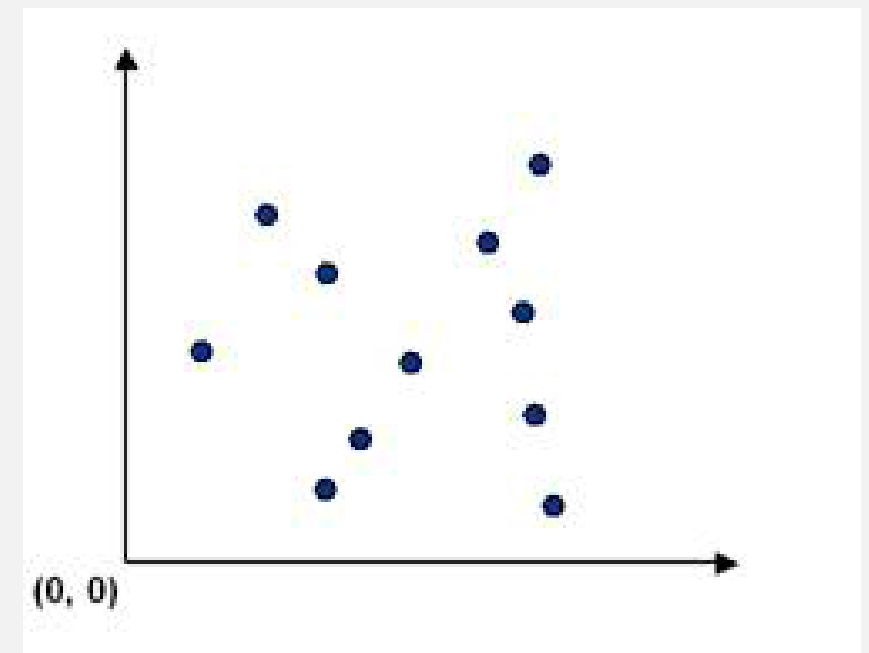
Manipulate points in the 2d-plane.

Perform search, insert, nearest-neighbor search and range search

`PointSET.java`: Implement using balanced binary search trees (red-black)

`KdTree.java`: Implement using 2d-trees

`Point2d.java`, `RectHV.java`: Supplied



# Programming Assignment S3: kd-Trees

---

See how 2d-trees actually work by

- implementing them
- comparing them with non-geometric implementations

Manipulate points in the 2d-plane.

Perform search, insert, nearest-neighbor search and range search

Visualizers:

`KdTreeVisualizer.java`:

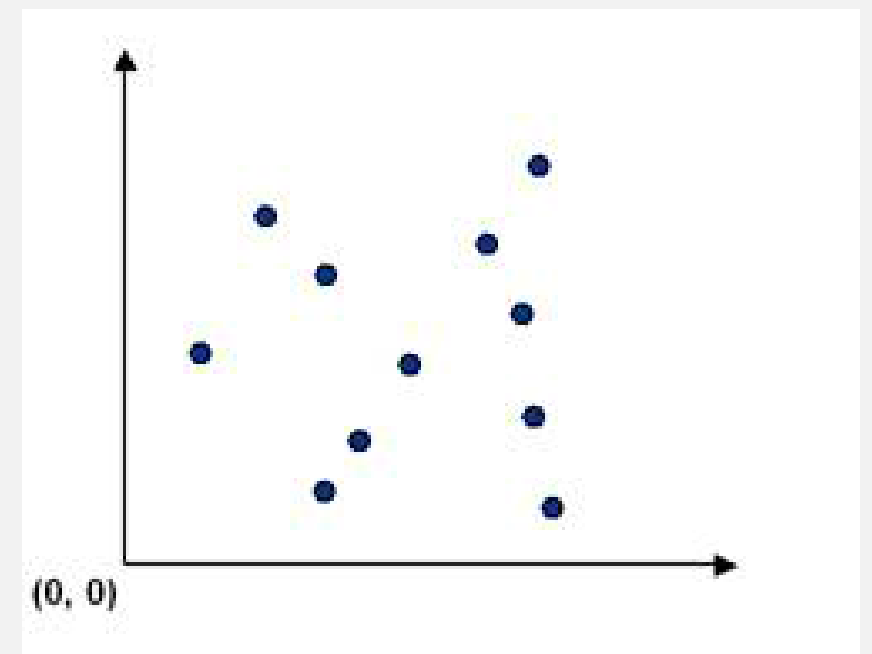
Interactive creation of pointset (and tree)

`RangeSearchVisualizer.java`:

Test functionality of the range search

`NearestNeighborVisualizer.java`:

Test functionality of nn-search



# Announcements

---

## NCPC/Háskólakeppnin í forritun

- Saturday 7 October
- Practice contests at [kthtraining.kattis.com](https://kthtraining.kattis.com):
  - Saturday September 23, 11:00-16:00 CEST
  - Sunday, October 1, 11:00-16:00 CEST