

# GEOMETRIC APPLICATIONS OF BSTs

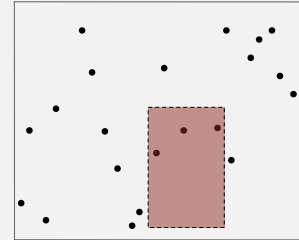


- ▶ 1d range search
- ▶ line segment intersection
- ▶ kd trees
- ▶ interval search trees
- ▶ rectangle intersection

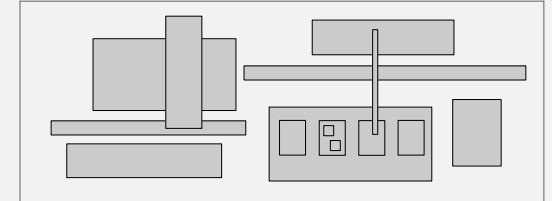
## Overview

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

Applications. CAD, games, movies, virtual reality, VLSI design, databases, ...



2d orthogonal range search



orthogonal rectangle intersection

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

- ▶ 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  $k_2$ .

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           |

## 1d range search: implementations

**Unordered array.** Fast insert, slow range search.

**Ordered array.** Slow insert, binary search for  $k_1$  and  $k_2$  to do range search.

| data structure  | insert | range count | range search |
|-----------------|--------|-------------|--------------|
| unordered array | 1      | N           | N            |
| ordered array   | N      | log N       | R + log N    |
| goal            | log N  | log N       | R + log N    |

order of growth of running time for 1d range search

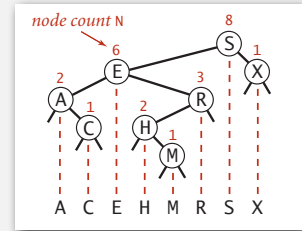
### Parameters.

- $N$  = number of keys.
- $R$  = number of keys that match. ← running time is output sensitive (number of matching keys can be N)

5

## 1d range count: BST implementation

**1d range count.** How many keys between  $l_0$  and  $h_i$  ?



```
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 is proportional to  $\log N$  (assuming BST is balanced).

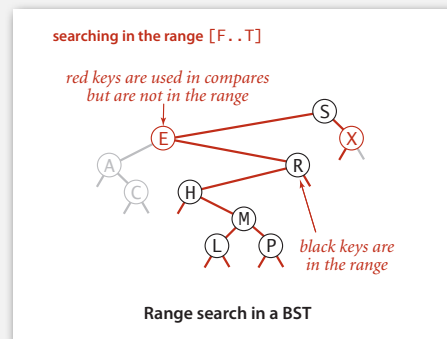
**Pf.** Nodes examined = search path to  $l_0$  + search path to  $h_i$ .

6

## 1d range search: BST implementation

**1d range search.** Find all keys between  $l_0$  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).



**Proposition.** Running time is proportional to  $R + \log N$  (assuming BST is balanced).

**Pf.** Nodes examined = search path to  $l_0$  + search path to  $h_i$  + matching keys.

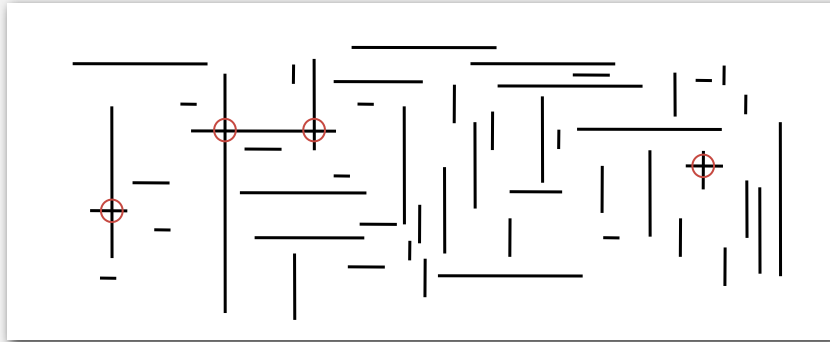
7

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

8

## Orthogonal line segment intersection search

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



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

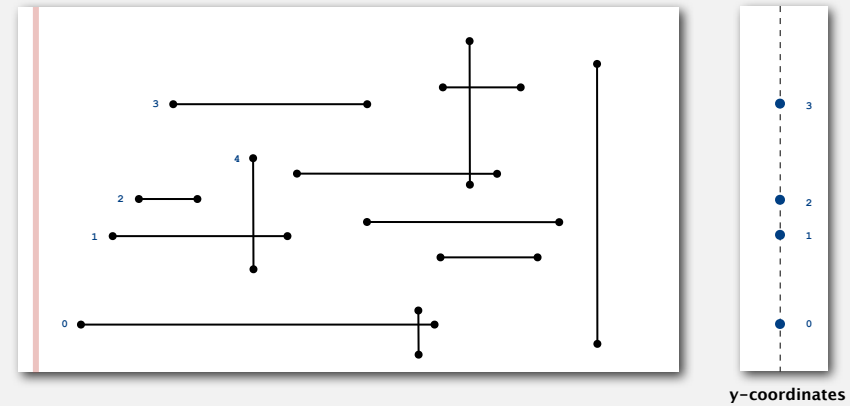
**Quadratic algorithm.** Check all pairs of line segments for intersection.

9

## Orthogonal line segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

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



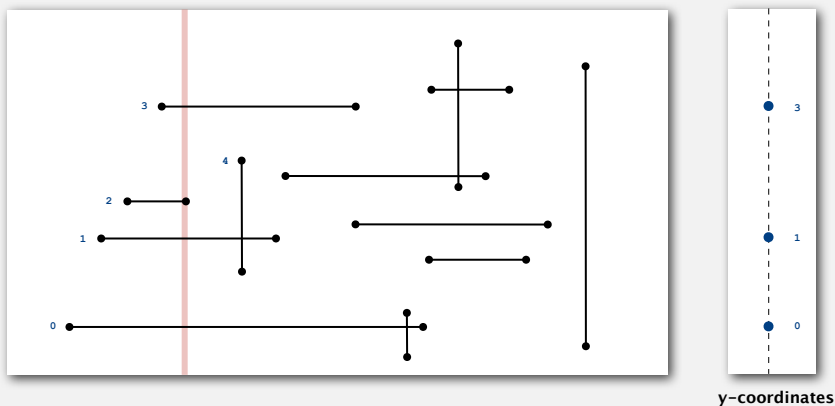
y-coordinates

10

## Orthogonal line segment intersection search: 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.



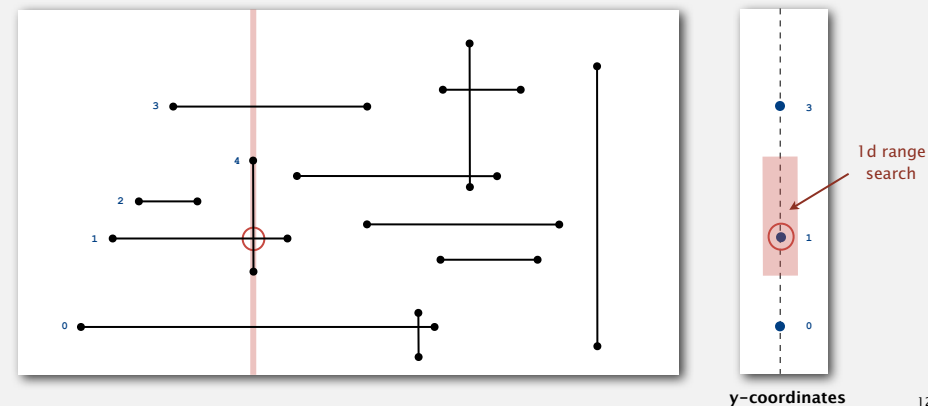
y-coordinates

11

## Orthogonal line segment intersection search: 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.



y-coordinates

12

## Orthogonal line segment intersection search: sweep-line algorithm analysis

**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.  $\leftarrow N \log N + R$

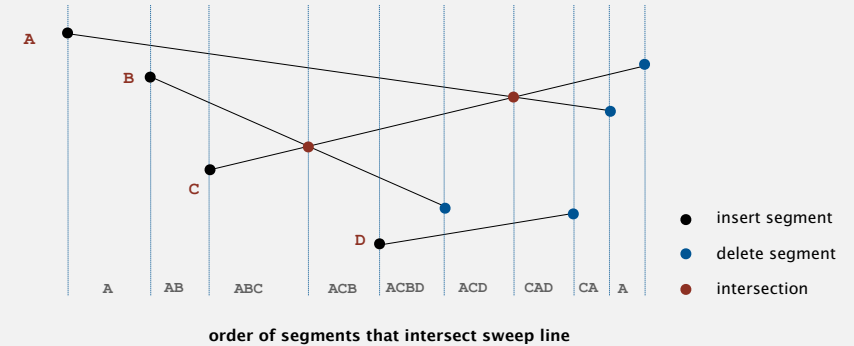
**Bottom line.** Sweep line reduces 2d orthogonal line segment intersection search to 1d range search.

13

## General line segment intersection search

**Sweep-line algorithm.**

- Maintain segments that intersect sweep line **ordered by  $y$ -coordinate**.
- Intersections can only occur between adjacent segments.
- Delete/add line segment  $\Rightarrow$  one/two new pairs of adjacent segments.
- Intersection  $\Rightarrow$  swap adjacent segments.



14

## General line segment intersection search: implementation

**Sweep-line algorithm.**

- Maintain PQ of important  $x$ -coordinates: endpoints and **intersections**.
- Maintain set of segments intersecting sweep line, in BST sorted by  $y$ -coordinates.

↑  
to support "next largest" and  
"next smallest" queries

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

**Implementation issues.**

- Degeneracy.
- Floating-point precision.
- Must use PQ, not presort (intersection events are unknown ahead of time).

15

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

16

## 2-d orthogonal range search

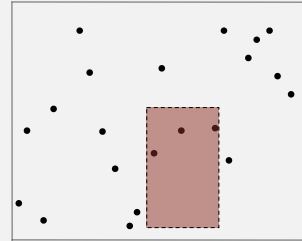
### Extension of ordered symbol-table to 2d keys.

- Insert a 2d key.
- Delete a 2d key.
- Search for 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.

### Geometric interpretation.

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

↑  
rectangle is axis-aligned



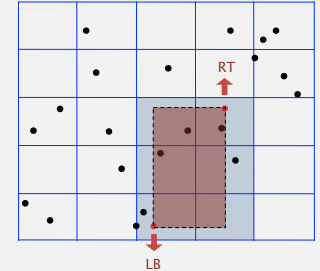
**Applications.** Networking, circuit design, databases.

17

## 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 those squares that intersect 2d range query.



18

## 2d orthogonal range search: grid implementation costs

### Space-time tradeoff.

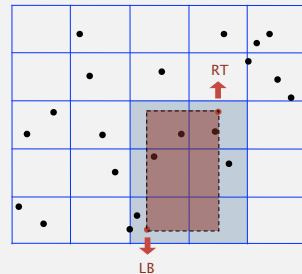
- Space:  $M^2 + N$ .
- Time:  $1 + N/M^2$  per square examined, on average.

### 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  $M \sim \sqrt{N}$



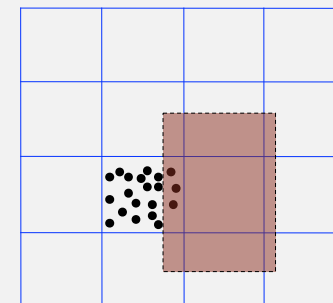
19

## Clustering

**Grid implementation.** Fast and 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 **gracefully** adapts to data.



20

## Clustering

**Grid implementation.** Fast and 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

21

## Space-partitioning trees

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

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

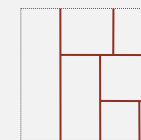
**2d tree.** Recursively divide space into two halfplanes.

**Quadtree.** Recursively divide space into four quadrants.

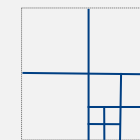
**BSP tree.** Recursively divide space into two regions.



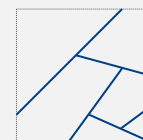
Grid



2d tree



Quadtree



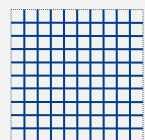
BSP tree

22

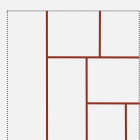
## Space-partitioning trees: applications

**Applications.**

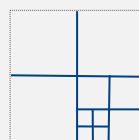
- Ray tracing.
- **2d range search.**
- Flight simulators.
- N-body simulation.
- Collision detection.
- Astronomical databases.
- **Nearest neighbor search.**
- Adaptive mesh generation.
- Accelerate rendering in Doom.
- Hidden surface removal and shadow casting.



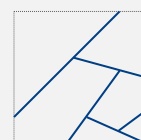
Grid



2d tree



Quadtree



BSP tree

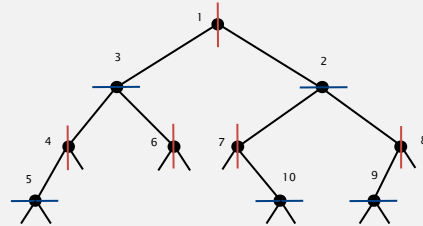
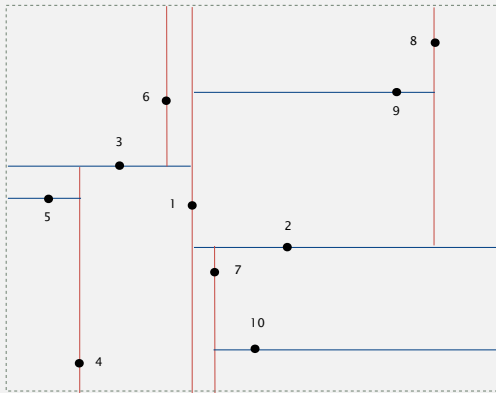
23

## 2d tree demo

24

## 2d tree

Recursively partition plane into two halfplanes.

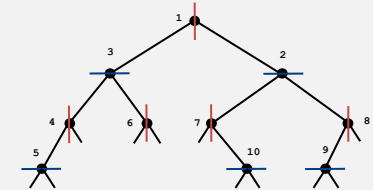
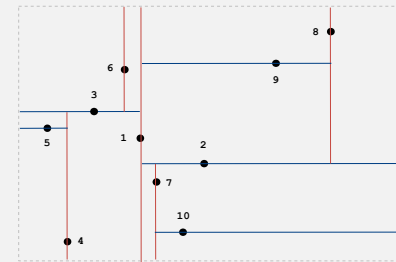
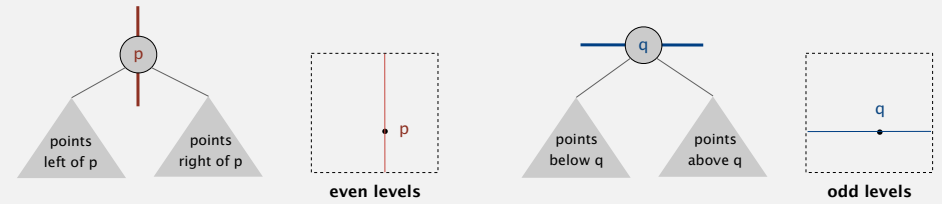


25

## 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.



26

## 2d tree demo: 2d orthogonal range search and nearest neighbor search

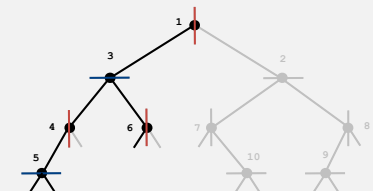
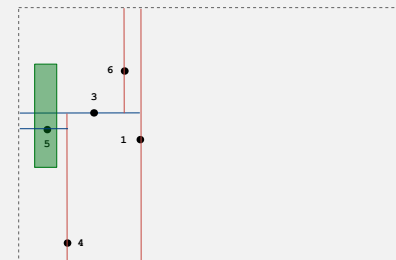
## 2d tree: 2d orthogonal range search

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

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

**Typical case.**  $R + \log N$ .

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



27

28

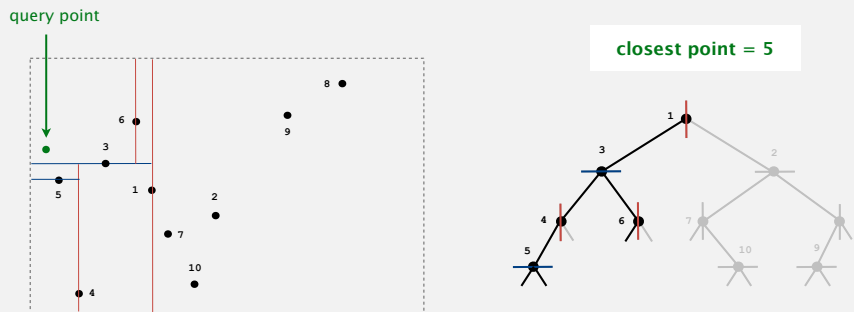
## 2d tree: nearest neighbor search

**Nearest neighbor search.** Given a query point, find the closest point.

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

**Typical case.**  $\log N$ .

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



29

## Point data type

```
public class Point2D implements Comparable<Point2D>
```

```

    Point2D(double x, double y)    create point (x, y)

    double x()                    x-coordinate
    double y()                    y-coordinate

    double distanceTo(Point2D p)  Euclidean distance between two points

    double distanceToSquared(Point2D p) square of Euclidean distance

    int compareTo(Point2D that)   natural order

    boolean equals(Object that)   does this point equal that?

```

why include this method?

30

## Axis-aligned rectangle data type

```
public class RectHV
```

```

    RectHV(...)                create [xmin, xmax] x [ymin, ymax]

    boolean contains(Point2D p) does this rectangle contain p?

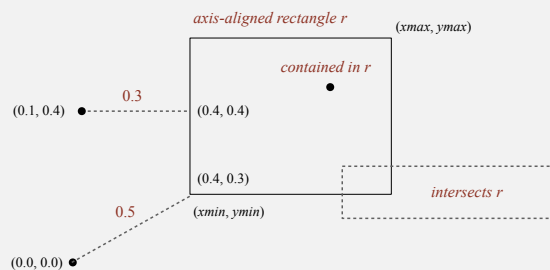
    boolean intersects(RectHV that) do two rectangles intersect?

    double distanceTo(Point2D p) Euclidean distance from p to rectangle

    boolean equals(Object that)  does this rectangle equal that?

```

how to implement?



31

## Flocking birds

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



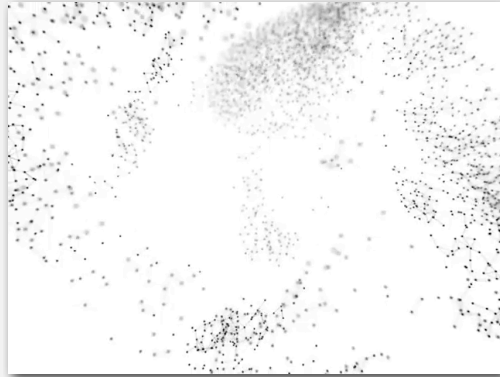
<http://www.youtube.com/watch?v=XH-groCeKbE>

32



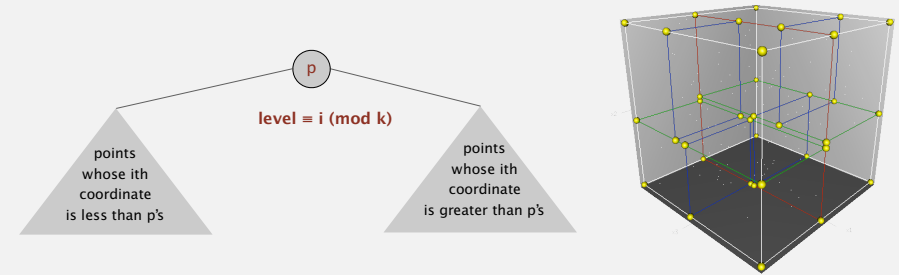
**Boids.** Three simple rules lead to complex emergent flocking behavior:

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

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



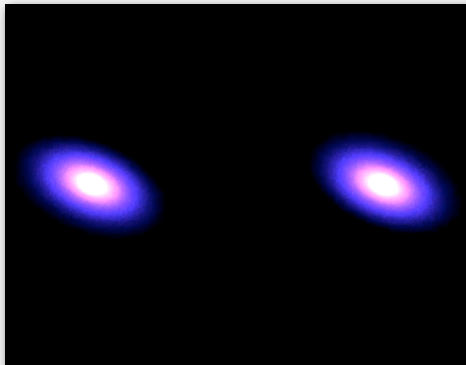
**Efficient, simple data structure for processing  $k$ -dimensional data.**

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



Jon Bentley

**Goal.** Simulate the motion of  $N$  particles, mutually affected by gravity.



[http://www.youtube.com/watch?v=ua7YIN4eL\\_w](http://www.youtube.com/watch?v=ua7YIN4eL_w)

**Brute force.** For each pair of particles, compute force.

$$F = \frac{G m_1 m_2}{r^2}$$

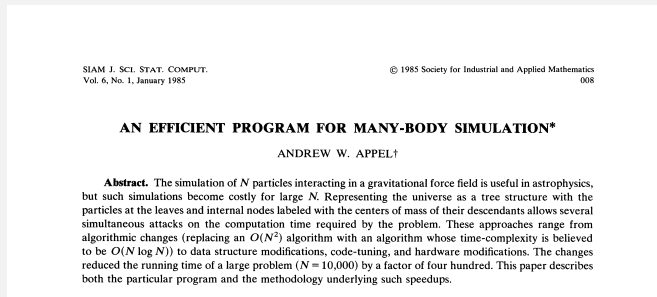
**Key idea.** Suppose particle is far, far away from cluster of particles.

- Treat cluster of particles as a single aggregate particle.
- Compute force between particle and **center of mass** of aggregate particle.



## Appel algorithm for N-body simulation

- Build 3d-tree with  $N$  particles as nodes.
- Store center-of-mass of subtree in each node.
- To compute total force acting on a particle, traverse tree, but stop as soon as distance from particle to subdivision is sufficiently large.



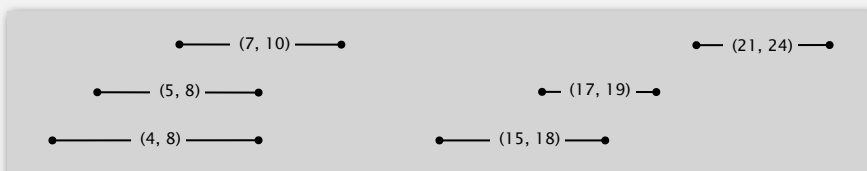
**Impact.** Running time per step is  $N \log N$  instead of  $N^2 \Rightarrow$  enables new research.

37

## 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 in data structure overlapping  $(lo, hi)$ .



39

## Interval search trees

```
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
                                                the given interval
```

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

40

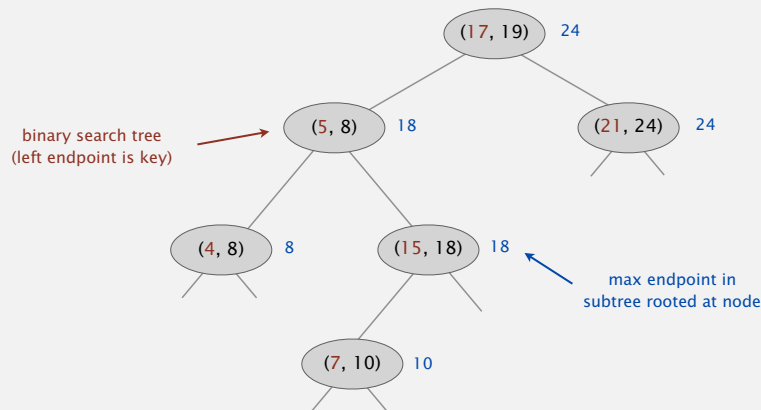
- › 1d range search
- › line segment intersection
- › kd trees
- › interval search trees
- › rectangle intersection

38

## 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.



41

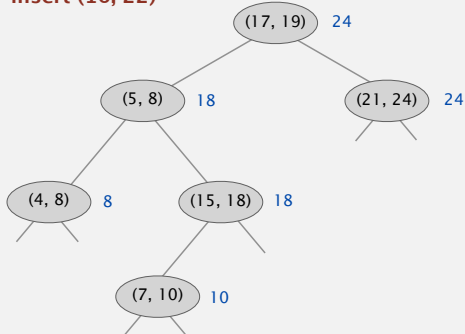
## Interval search tree demo

### Insert an interval

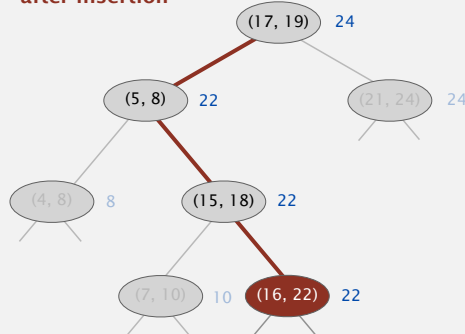
To insert an interval  $(lo, hi)$ :

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

insert (16, 22)



after insertion



43

### Search for an intersecting interval

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 empty, 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;
```

44

## Search for an intersecting interval

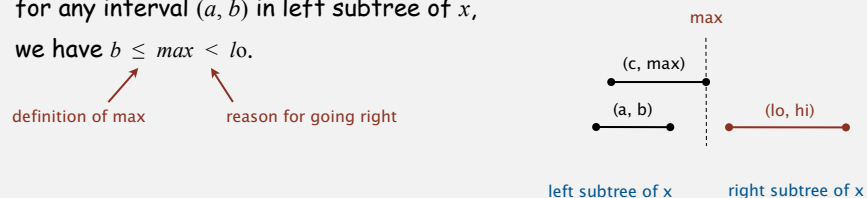
To search for any one interval that intersects query interval  $(lo, hi)$ :

- If interval in node intersects query interval, return it.
- Else if left subtree is empty, 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.**

- Left subtree is empty  $\Rightarrow$  trivial.
- Max endpoint  $max$  in left subtree is less than  $lo \Rightarrow$  for any interval  $(a, b)$  in left subtree of  $x$ , we have  $b \leq max < lo$ .



45

## Search for an intersecting interval

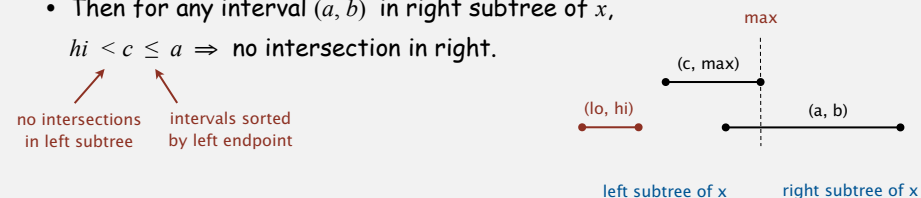
To search for any one interval that intersects query interval  $(lo, hi)$ :

- If interval in node intersects query interval, return it.
- Else if left subtree is empty, 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  $x$ ,  $hi < c \leq a \Rightarrow$  no intersection in right.



46

## Interval search tree: analysis

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

can maintain auxiliary information using  $\log N$  extra work per op

| operation   | brute | interval search tree | best in theory |
|---|-------|----------------------|----------------|
| insert interval   | 1     | $\log N$             | $\log N$       |
| find interval   | $N$   | $\log N$             | $\log N$       |
| delete interval   | $N$   | $\log N$             | $\log N$       |
| find <b>any one</b> interval that intersects $(lo, hi)$ | $N$   | $\log N$             | $\log N$       |
| find <b>all</b> intervals that intersects $(lo, hi)$    | $N$   | $R \log N$           | $R + \log N$   |

order of growth of running time for  $N$  intervals

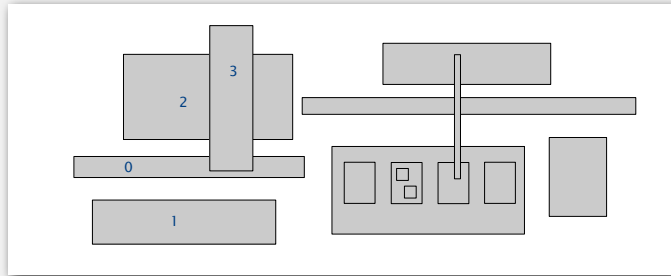
47

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

48

## Orthogonal rectangle intersection search

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



**Non-degeneracy assumption.** All  $x$ - and  $y$ -coordinates are distinct.  
**Quadratic algorithm.** Check all pairs of rectangles for intersection.

49

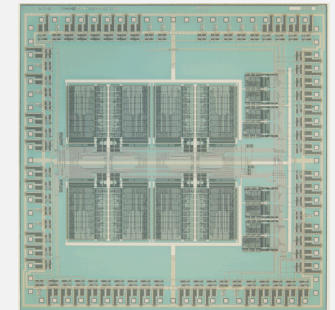
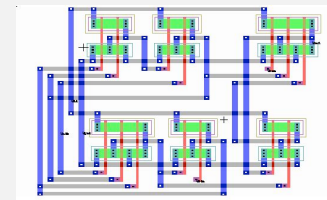
## Microprocessors and geometry

**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.

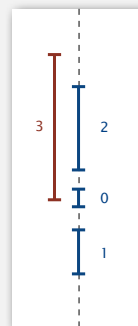
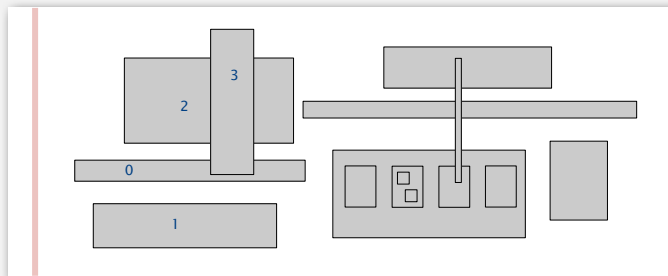


50

## Orthogonal rectangle intersection search: sweep-line algorithm

**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-coordinates

51

## Orthogonal rectangle intersection search: sweep-line algorithm analysis

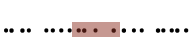
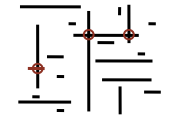
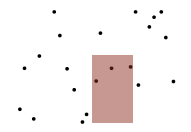
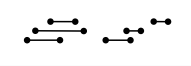
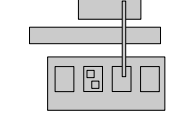
**Proposition.** Sweep line algorithm takes time proportional to  $N \log N + R \log N$  to find  $R$  intersections among a set of  $N$  rectangles.

**Pf.**

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

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

52

| problem  | example   | solution                                 |
|--|---|--|
| 1d range search                                |  | BST                                      |
| 2d orthogonal line segment intersection search |  | sweep line reduces to 1d range search    |
| kd range search                                |  | kd tree                                  |
| 1d interval search                             |  | interval search tree                     |
| 2d orthogonal rectangle intersection search    |  | sweep line reduces to 1d interval search |