Geometric Search

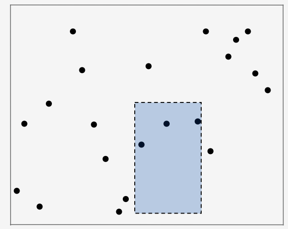
- range search
- space partitioning trees
- intersection search

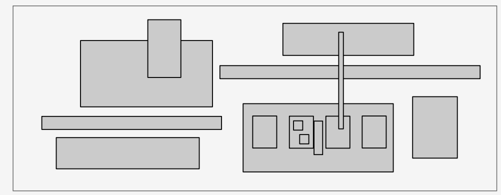
References:
Algorithms in C (2nd edition), Chapters 26-27
http://www.cs.princeton.edu/algs4/73range
http://www.cs.princeton.edu/algs4/74intersection

Overview

Types of data. Points, lines, circles, rectangles, planes, polygons, ...
This lecture. Intersection among N objects.

Example problems.
- 1d range searching.
- 2d range searching.
- Finding intersections among h-v line segments.
- Find intersections among axis-aligned rectangles.





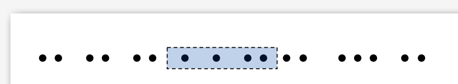
- range search
- space partitioning trees
- intersection search

1D range search

Extension of ordered symbol table.
- Insert key-value pair.
- Search for key k.
- Rank: how many keys less than k?
- Range count: how many keys between k_1 and k_2?
- Range search: find over all keys between k_1 and k_2.

Application. Database queries.

Geometric interpretation.
- Keys are point on a line.
- How many points in a given 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

Ordered array. Slow insert, binary search for `lo` and `hi` to find range.
Hash table. No reasonable algorithm (key order lost in hash).

data structure	insert	rank	range count	range search
ordered array	N	log N	log N	R + log N
hash table	1	N	N	N
BST	log N	log N	log N	R + log N

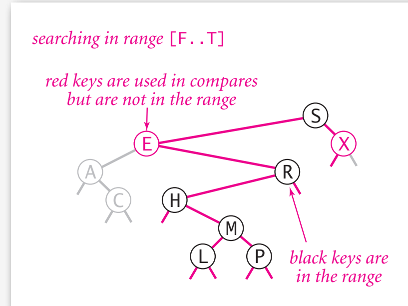
N = # keys
R = # keys that match

Goal. Modify standard BST to support efficient range queries.

BST: range search

Range search. Find all keys between `lo` and `hi`?

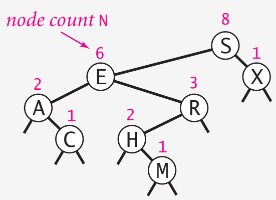
- Recursively find all keys in left subtree (if any could fall in range).
- Check key in current node.
- Recursively find all keys in right subtree (if any could fall in range).



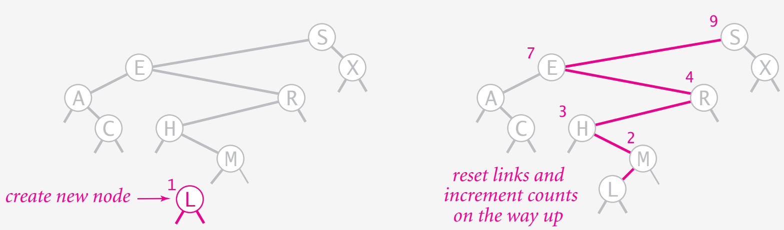
Worst-case running time. R + log N (assuming BST is balanced).

BST: maintaining node counts

BST. In each node x, maintain number of nodes in tree rooted at x.



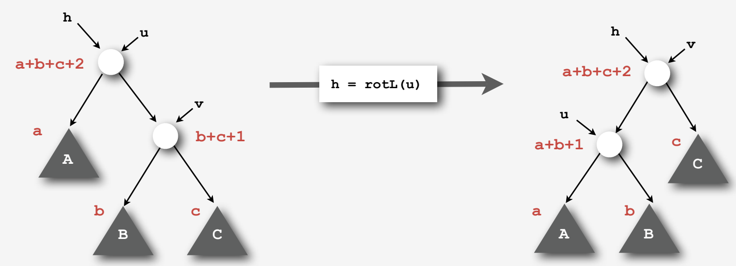
Updating node counts after insertion.



BST: maintaining node counts

BST. In each node x, maintain number of nodes in tree rooted at x.

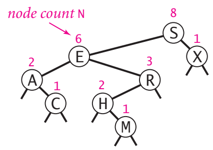
Updating node counts after rotation.



BST: range count

Rank. How many keys < k ?

```
public int rank(Key key)
{  return rank(key, root);  }

private int rank(Key key, Node x)
{
   if (x == null) return 0;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) return rank(key, x.left);
   else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
   else              return size(x.left);
}
```



Range count. How many keys between `lo` and `hi`?

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


2D orthogonal range search

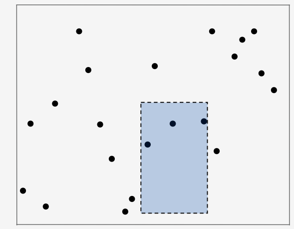
Extension of ordered symbol-table to 2D keys.

- Insert a 2D key.
- Search for a 2D key.
- Range count: how many keys lie in a 2D range?
- Range search: find all keys that lie in a 2D range?

Applications. Networking, circuit design, databases.

Geometric interpretation.

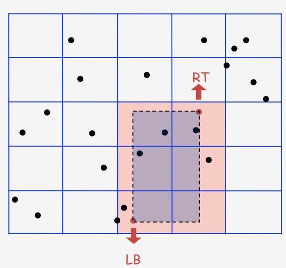
- Keys are point in the plane.
- How many points in a given h-v rectangle.



2D orthogonal range search: grid implementation

Grid implementation. [Sedgewick 3.18]

- 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: insert (x, y) into corresponding square.
- Range search: examine only those squares that intersect 2D range query.



2D orthogonal range search: grid implementation costs

Space-time tradeoff.

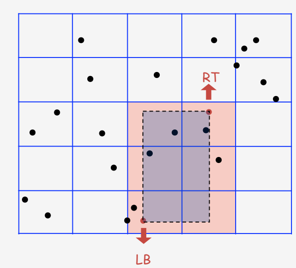
- Space: $M^2 + N$.
- Time: $1 + N / M^2$ per square examined, on average.

Choose grid square size to tune performance.

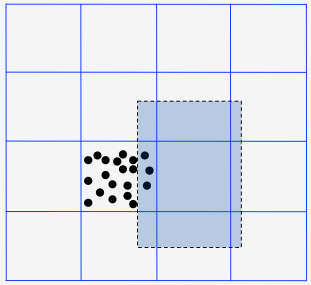
- Too small: wastes space.
- Too large: too many points per square.
- Rule of thumb: $\sqrt{N}$-by-$\sqrt{N}$ grid.

Running time. [if points are evenly distributed]

- Initialize: O(N).
- Insert: O(1). $M \approx \sqrt{N}$
- Range: O(1) per point in range.



Grid implementation. Fast, simple solution for well-distributed points.
Problem. Clustering a well-known phenomenon in geometric data.

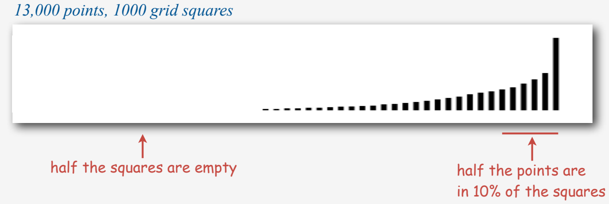


Lists are too long, even though average length is short.
Need data structure that gracefully adapts to data.

Grid implementation. Fast, simple solution for well-distributed points.
Problem. Clustering a well-known phenomenon in geometric data.

Ex. USA map data.

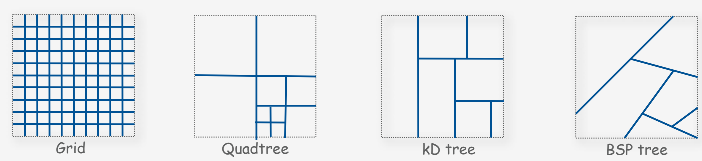
13,000 points, 1000 grid squares



- range search
- space partitioning trees
- intersection search

Use a tree to represent a recursive subdivision of k-dimensional space.

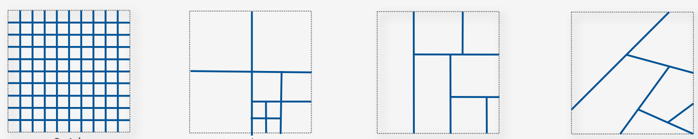
Quadtree. Recursively divide plane into four quadrants.
kD tree. Recursively divide k-dimensional space into two half-spaces.
BSP tree. Recursively divide space into two regions.



Space-partitioning trees: applications

Applications.

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

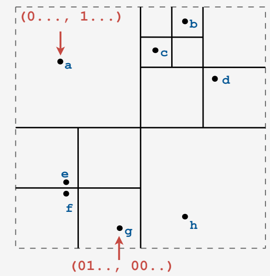


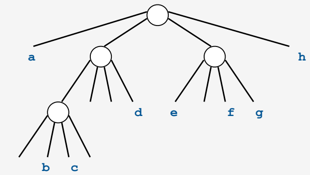
Grid   Quadtree   kD tree   BSP tree

Quadtree

Idea. Recursively divide plane into 4 quadrants.

Implementation. 4-way tree (actually a trie).






```
public class QuadTree
{
   private Quad quad;
   private Value val;
   private QuadTree NW, NE, SW, SE;
}
```

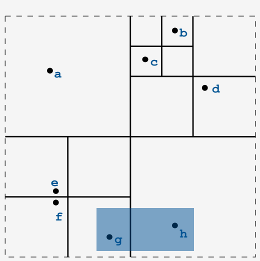
Benefit. Good performance in the presence of clustering.

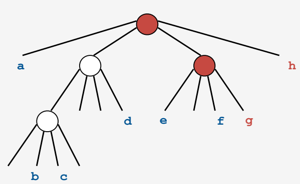
Drawback. Arbitrary depth!

Quadtree: 2D range search

Range search. Find all keys in a given 2D range.

- Recursively find all keys in NE quad (if any could fall in range).
- Recursively find all keys in NW quad (if any could fall in range).
- Recursively find all keys in SE quad (if any could fall in range).
- Recursively find all keys in SW quad (if any could fall in range).

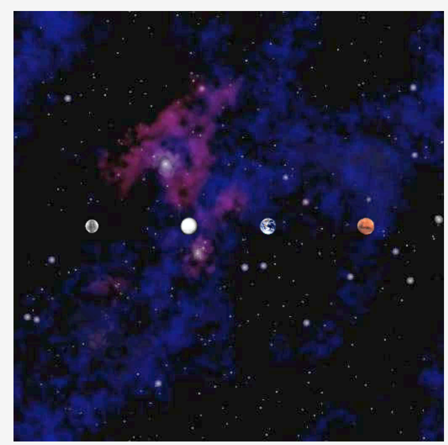




Typical running time. R + log N.

N-body simulation

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



Brute force. For each pair of particles, compute force. $F = \frac{G m_1 m_2}{r^2}$

Subquadratic N-body simulation

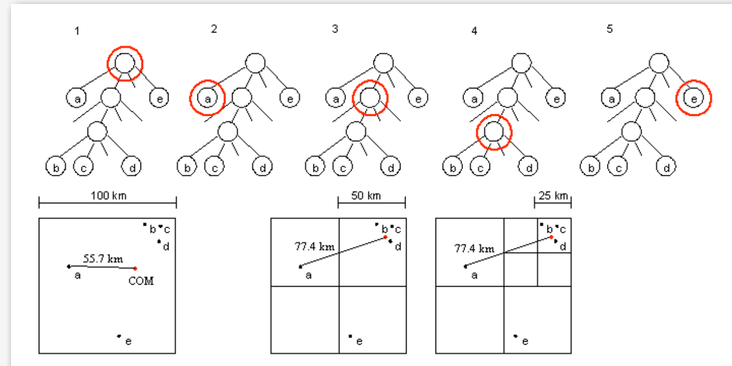
Key idea. Suppose particle is far, far away from cluster of particles.

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

Barnes-Hut algorithm

Algorithm.

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



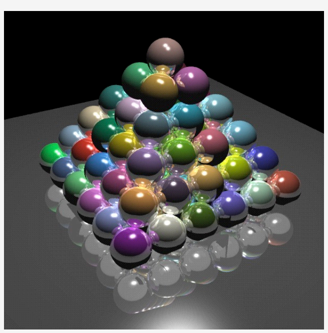
Curse of dimensionality

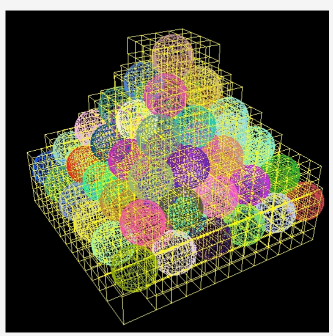
Range search / nearest neighbor in k dimensions?

Main application. Multi-dimensional databases.

3D space. Octrees: recursively divide 3D space into 8 octants.

100D space. Centrees: recursively divide into 2^{100} centrants???





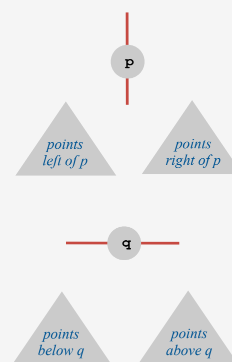
Raytracing with octrees
http://graphics.cs.ucdavis.edu/~gregorsk/graphics/275.html

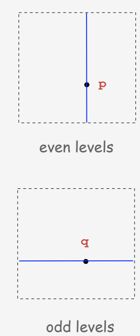
2D trees

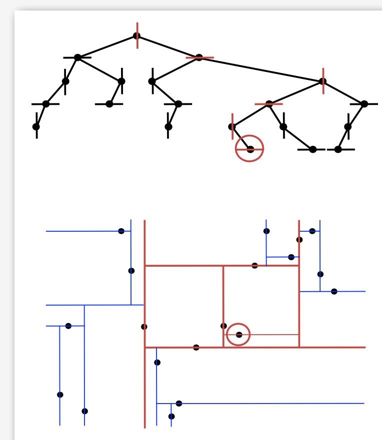
Recursively partition plane into two halfplanes.

Implementation. BST, but alternate using x- and y-coordinates as key.

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



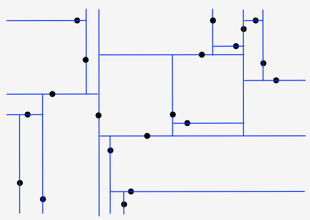


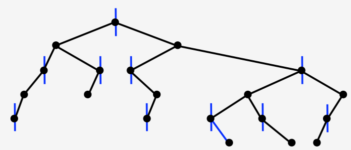


2D tree: 2D range search

Range search. Find all keys in a given 2D range.

- Check if point in node lies in given range.
- Recursively find all keys in left/top subdivision (if any could fall in range).
- Recursively find all keys in left/top subdivision (if any could fall in range).





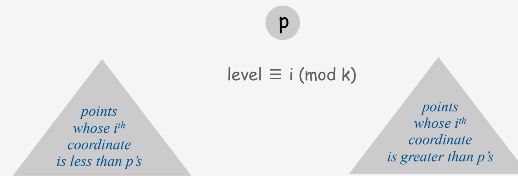
Worst case (assuming tree is balanced). R + √N.

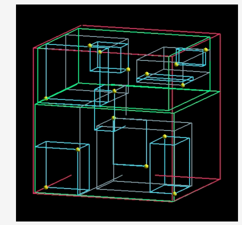
Typical case. R + log N

kD Trees

kD tree. Recursively partition k-dimensional space into 2 halfspaces.

Implementation. BST, but cycle through dimensions ala 2D trees.





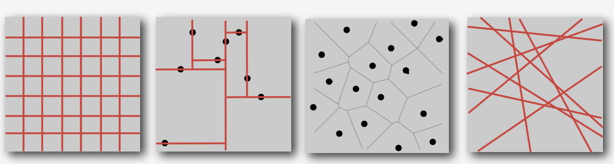
Efficient, simple data structure for processing k-dimensional data.

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

Summary

Basis of many geometric algorithms. Search in a planar subdivision.

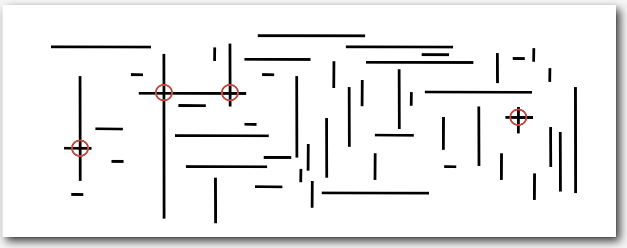
	grid	2D tree	Voronoi diagram	intersecting lines
basis	√N h-v lines	N points	N points	√N lines
representation	2D array of N lists	N-node BST	N-node multilist	~N-node BST
cells	~N squares	N rectangles	N polygons	~N triangles
search cost	1	log N	log N	log N
extend to kD?	too many cells	easy	cells too complicated	use (k-1)D hyperplane



Search for intersections

Problem. Find all intersecting pairs among set of N geometric objects.
Applications. CAD, games, movies, virtual reality.

Simple version. 2D, all objects are horizontal or vertical line segments.

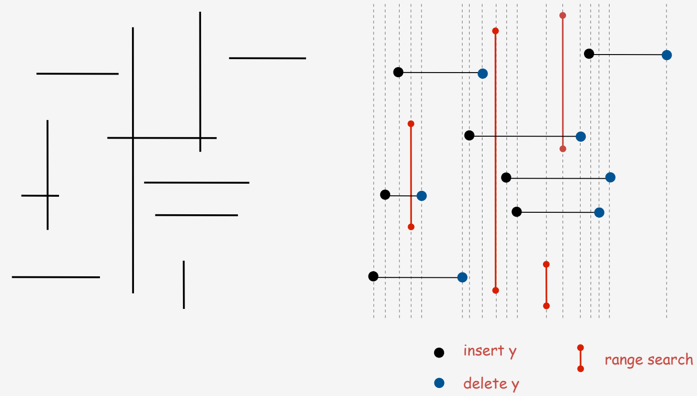


Brute force. Test all $\Theta(N^2)$ pairs of line segments for intersection.
Sweep line. Efficient solution extends to 3D and general objects.

Orthogonal segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

- x-coordinates define events.
- Left endpoint of h-segment: insert y coordinate into ST.
- Right endpoint of h-segment: remove y coordinate from ST.
- v-segment: range search for interval of y endpoints.



Orthogonal segment intersection search: sweep-line algorithm

Reduces 2D orthogonal segment intersection search to 1D range search!

Running time of sweep line algorithm.

- Put x-coordinates on a PQ (or sort). O(N log N)
- Insert y-coordinate into ST. O(N log N)
- Delete y-coordinate from ST. O(N log N)
- Range search. O(R + N log N)

N = # line segments
R = # intersections

Efficiency relies on judicious use of data structures.

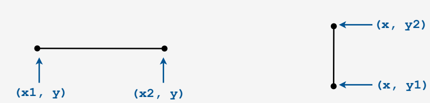
Immutable H-V segment ADT

```
public final class SegmentHV implements Comparable<SegmentHV>
{
   public final int x1, y1;
   public final int x2, y2;

   public SegmentHV(int x1, int y1, int x2, int y2)
   {  ...  }

   public boolean isHorizontal()
   {  ...  }
   public boolean isVertical()
   {  ...  }

   public int compareTo(SegmentHV b)
   {  ...  }
}
```

constructor

is segment horizontal?
is segment vertical?

compare by x-coordinate;
break ties by y-coordinate



horizontal segment *vertical segment*

Sweep-line event subclass

```
private class Event implements Comparable<Event>
{
   private int time;
   private SegmentHV segment;

   public Event(int time, SegmentHV segment)
   {
      this.time    = time;
      this.segment = segment;
   }

   public int compareTo(Event that)
   {  return this.time - that.time;  }
}
```


Sweep-line algorithm: initialize events

```
MinPQ<Event> pq = new MinPQ<Event>();

for (int i = 0; i < N; i++)
{
   if (segments[i].isVertical())
   {
      Event e = new Event(segments[i].x1, segments[i]);
      pq.insert(e);
   }

   else if (segments[i].isHorizontal())
   {
      Event e1 = new Event(segments[i].x1, segments[i]);
      Event e2 = new Event(segments[i].x2, segments[i]);
      pq.insert(e1);
      pq.insert(e2);
   }
}
```

initialize PQ

vertical segment

horizontal segment

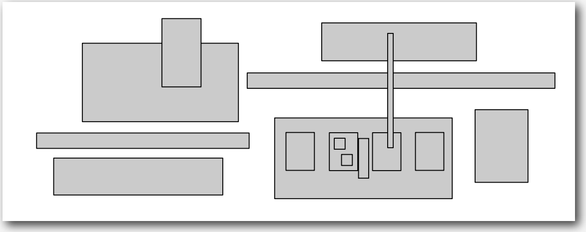
Sweep-line algorithm: simulate the sweep line

```
int INF = Integer.MAX_VALUE;

SET<SegmentHV> set = new SET<SegmentHV>();

while (!pq.isEmpty())
{
   Event event = pq.delMin();
   int sweep = event.time;
   SegmentHV segment = event.segment;

   if (segment.isVertical())
   {
      SegmentHV seg1, seg2;
      seg1 = new SegmentHV(-INF, segment.y1, -INF, segment.y1);
      seg2 = new SegmentHV(+INF, segment.y2, +INF, segment.y2);
      for (SegmentHV seg : set.range(seg1, seg2))
          StdOut.println(segment + " intersects " + seg);
   }

   else if (sweep == segment.x1) set.add(segment);
   else if (sweep == segment.x2) set.remove(segment);
}
```


- range search
- space partitioning trees
- intersection search
- **VLSI rules check**

Rectangle intersection search

Goal. Find all intersections among h-v rectangles.



Application. Design-rule checking in VLSI circuits.

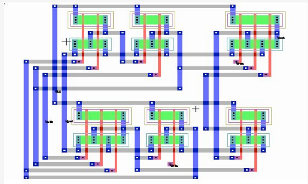
Microprocessors and geometry

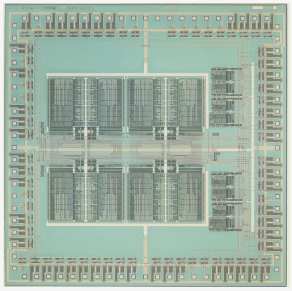
Early 1970s. microprocessor design became a geometric problem.

- 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 = rectangle intersection search.





Algorithms and Moore's law

"Moore's law." Processing power doubles every 18 months.

- 197x: need to check N rectangles.
- 197(x+1.5): need to check 2N rectangles on a 2x-faster computer.

Bootstrapping. We get to use the faster computer for bigger circuits.

But bootstrapping is not enough if using a quadratic algorithm:

- 197x: takes M days.
- 197(x+1.5): takes (4M)/2 = 2M days. (!)

quadratic algorithm

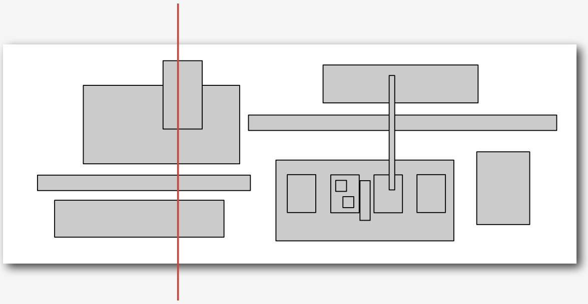
2x-faster computer

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

Rectangle intersection search

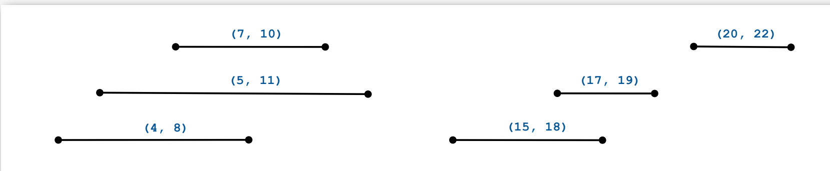
Move a vertical "sweep line" from left to right.

- Sweep line: sort rectangles by x-coordinate and process in this order, stopping on left and right endpoints.
- Maintain set of intervals intersecting sweep line.
- Key operation: given a new interval, does it intersect one in the set?



Interval search trees

`public class IntervalST<Value>`		
	`IntervalST()`	*create interval search tree*
`void`	`put(int lo, int hi, Value val)`	*put interval-value pair into ST*
`Value`	`get(int lo, int hi)`	*return value paired with given interval*
`boolean`	`remove(int lo, int hi)`	*remove the given interval*
`Iterable<Interval>`	`searchAll(int lo, int hi)`	*return all intervals that intersect the given interval*

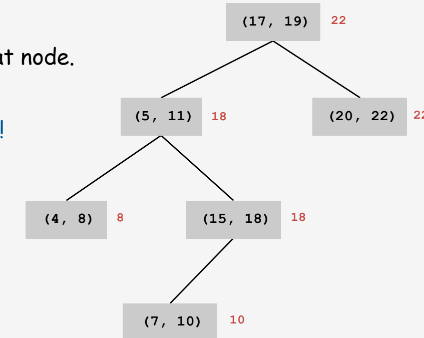


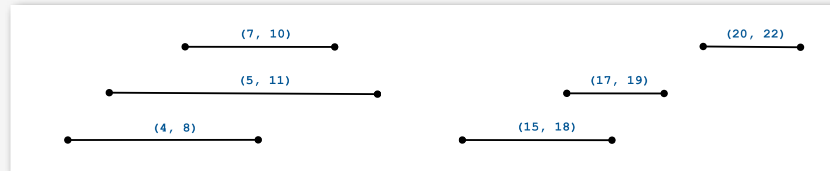
Interval search trees

Create BST, where each node stores an interval.

- Use `lo` endpoint as BST key.
- Store max endpoint in subtree rooted at node.

Suffices to implement all ops efficiently!!!

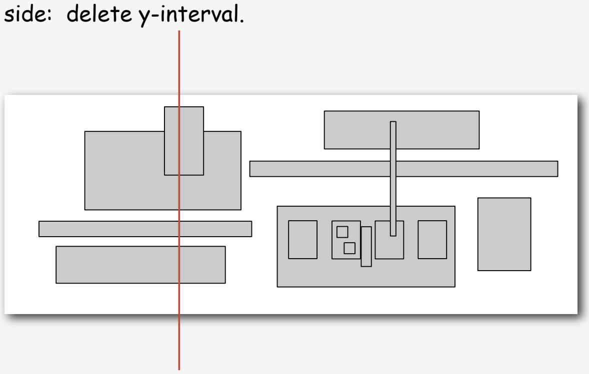




Rectangle intersection sweep-line algorithm: Review

Move a vertical "sweep line" from left to right.

- Sweep line: sort rectangles by x-coordinates and process in this order, stopping on left and right endpoints.
- Maintain set of rectangles that intersect the sweep line in an interval search tree (using y-interval of rectangle).
- Left side: interval search for y-interval of rectangle, insert y-interval.
- Right side: delete y-interval.



VLSI rules checking: sweep-line algorithm (summary)

Reduces 2D orthogonal rectangle intersection search to 1D interval search!

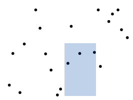
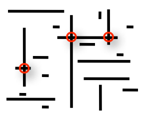
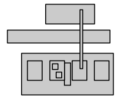
Running time of sweep line algorithm.

- Put x-coordinates on a PQ (or sort). $O(N \log N)$
- Insert y-interval into ST. $O(N \log N)$
- Delete y-interval from ST. $O(N \log N)$
- Interval search. $O(R + N \log N)$

N = # rectangles
R = # intersections

Efficiency relies on judicious use of data structures.

Geometric search summary: algorithms of the day

1D range search		BST
kD range search		kD tree
1D interval intersection search		interval tree
2D orthogonal line intersection search		sweep line reduces to 1D range search
2D orthogonal rectangle intersection search		sweep line reduces to 1D interval intersection search