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, databases, GIS, ....

Efficient solutions. Binary search trees (and extensions).
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 \( k_2 \).

Application. Database queries.

Geometric interpretation.
• Keys are point on a line.
• Find/count points in a given 1d interval.

<table>
<thead>
<tr>
<th>Insert</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>insert B</td>
<td>B</td>
</tr>
<tr>
<td>insert D</td>
<td>B D</td>
</tr>
<tr>
<td>insert A</td>
<td>A B D</td>
</tr>
<tr>
<td>insert I</td>
<td>A B D I</td>
</tr>
<tr>
<td>insert H</td>
<td>A B D H I</td>
</tr>
<tr>
<td>insert F</td>
<td>A B D F H I</td>
</tr>
<tr>
<td>insert P</td>
<td>A B D F H I P</td>
</tr>
<tr>
<td>search G to K</td>
<td>H I</td>
</tr>
<tr>
<td>count G to K</td>
<td>2</td>
</tr>
</tbody>
</table>

• • • • • • • • • • • • • • • • • • • •
1d range search: elementary implementations

**Unordered list.** Slow insert, slow range search.

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

### Order of growth of running time for 1d range search

<table>
<thead>
<tr>
<th>data structure</th>
<th>insert</th>
<th>range count</th>
<th>range search</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered list</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
</tr>
<tr>
<td>ordered array</td>
<td>( N )</td>
<td>( \log N )</td>
<td>( R + \log N )</td>
</tr>
<tr>
<td>goal</td>
<td>( \log N )</td>
<td>( \log N )</td>
<td>( R + \log N )</td>
</tr>
</tbody>
</table>

\( N = \) number of keys
\( R = \) number of keys that match
1d range count. How many keys between \( l_0 \) and \( h_i \) ?

**Proposition.** Running time proportional to \( \log N \).  

**Pf.** Nodes examined = search path to \( l_0 \) + search path to \( h_i \).
1d range search: BST implementation

1d 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).

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

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

**Nondegeneracy assumption.** All $x$- and $y$-coordinates are distinct.
Sweep vertical line from left to right.

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

![Diagram of orthogonal line segment intersection using sweep-line algorithm](image)
Orthogonal line segment intersection: sweep-line 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). $\longrightarrow N \log N$
- Insert $y$-coordinates into BST. $\longrightarrow N \log N$
- Delete $y$-coordinates from BST. $\longrightarrow N \log N$
- Range searches in BST. $\longrightarrow N \log N + R$

**Bottom line.** Sweep line reduces 2d orthogonal line segment intersection search to 1d range search.
Geometric Applications of BSTs

- 1d range search
- line segment intersection
- $kd$ trees
- interval search trees
- rectangle intersection
2-d 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*.
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.
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.
Clustering

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

Worst case (assuming tree is balanced). $R + \sqrt{N}$. 
**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 search in a 2d tree analysis

Typical case. \( \log N \).

Worst case (even if tree is balanced). \( N \).
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!
**N-body simulation**

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

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

**Running time.** Time per step is $N^2$.

http://www.youtube.com/watch?v=ua7YN4eL_w
Appel's algorithm for 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.
Appel's 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 $N \log N \Rightarrow \text{enables new research.}$
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public class IntervalST&lt;Key extends Comparable&lt;Key&gt;, Value&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>IntervalST()</code></td>
<td>create interval search tree</td>
</tr>
<tr>
<td><code>void put(Key lo, Key hi, Value val)</code></td>
<td>put interval-value pair into ST</td>
</tr>
<tr>
<td><code>Value get(Key lo, Key hi)</code></td>
<td>value paired with given interval</td>
</tr>
<tr>
<td><code>void delete(Key lo, Key hi)</code></td>
<td>delete the given interval</td>
</tr>
<tr>
<td><code>Iterable&lt;Value&gt; intersects(Key lo, Key hi)</code></td>
<td>all intervals that intersect (lo, hi)</td>
</tr>
</tbody>
</table>

**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 \((21, 23)\)**

\[(4, 8)\] 8
\[(5, 8)\] 22
\[(7, 10)\] 10
\[(15, 18)\] 22
\[(16, 22)\] 22
\[(21, 24)\] 24
\[(17, 19)\] 24

compare \((21, 23)\) to \((16, 22)\) (intersection!)
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 \(b \leq max < lo\).

- Thus, \((a, b)\) will not intersect \((lo, hi)\).
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.
**Interval search tree: analysis**

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

This structure is easy to maintain auxiliary information (log N extra work per operation).

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

**order of growth of running time for data structure with N intervals**
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.
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.
Moore’s law. Transistor count doubles every 2 years.
Sustaining Moore's law.

- Problem size doubles every 2 years.  
- Processing power doubles every 2 years.  
- How much $ do I need to get the job done with a quadratic algorithm?

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

\[
T_{2N} = \left(\frac{a}{2}\right) (2N)^2 = 2 T_N \quad \text{running time in 2 years}
\]

\[
= 2 a N^2
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
<td>($X)</td>
<td>($X)</td>
<td>($X)</td>
<td>($X)</td>
</tr>
<tr>
<td>(N \log N)</td>
<td>($X)</td>
<td>($X)</td>
<td>($X)</td>
<td>($X)</td>
</tr>
<tr>
<td>(N^2)</td>
<td>($X)</td>
<td>($2X)</td>
<td>($4X)</td>
<td>($2^{15} X)</td>
</tr>
</tbody>
</table>

Bottom line.  Linearithmic algorithm is necessary to sustain Moore's Law.
Orthogonal rectangle intersection: 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.
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). $\rightarrow N \log N$
- Insert $y$-intervals into ST. $\rightarrow N \log N$
- Delete $y$-intervals from ST. $\rightarrow N \log N$
- Interval searches for $y$-intervals. $\rightarrow N \log N + R \log N$

**Bottom line.** Sweep line reduces 2d orthogonal rectangle intersection search to 1d interval search.
Geometric applications of BSTs

<table>
<thead>
<tr>
<th>problem</th>
<th>example</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1d range search</td>
<td></td>
<td>BST</td>
</tr>
<tr>
<td>2d orthogonal line segment intersection</td>
<td></td>
<td>sweep line reduces problem to 1d range search</td>
</tr>
<tr>
<td>2d range search</td>
<td></td>
<td>2d tree</td>
</tr>
<tr>
<td>kd range search</td>
<td></td>
<td>kd tree</td>
</tr>
<tr>
<td>1d interval search</td>
<td></td>
<td>interval search tree</td>
</tr>
<tr>
<td>2d orthogonal rectangle intersection</td>
<td></td>
<td>sweep line reduces problem to 1d interval search</td>
</tr>
</tbody>
</table>