**Overview**

This lecture. Intersections among geometric objects.

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

Efficient solutions. Binary search trees (and extensions).

---

**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.
1d range search: elementary implementations

**Unordered list.** Fast 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 |  |
|---|---|---|---|
| data structure | insert | range count | range search |
| unordered list | 1 | \( N \) | \( N \) |
| ordered array | \( N \) | \( \log N \) | \( R + \log N \) |
| goal | \( \log N \) | \( \log N \) | \( R + \log N \) |

\( N = \text{number of keys} \)  
\( R = \text{number of keys that match} \)

1d range count: BST implementation

**1d range count.** How many keys between \( l_0 \) and \( h_i \) ?

```java
public int size(Key l0, Key hi) {
    if (contains(hi)) return rank(hi) - rank(lo) + 1;
    else return rank(hi) - rank(lo);
}
```

**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 \( l_0 \) 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 proportional to \( R + \log N \).  
**Pf.** Nodes examined = search path to \( l_0 \) + search path to \( h_i \) + 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.

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.

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 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). \( \rightarrow N \log N \)
- Insert \( y \)-coordinates into BST. \( \rightarrow N \log N \)
- Delete \( y \)-coordinates from BST. \( \rightarrow N \log N \)
- Range searches in BST. \( \rightarrow N \log N + R \)

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

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.

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

**Geometric interpretation.**
- Keys are point in the plane.
- Find/count points in a given \( h \times v \) 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.

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.
**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.

**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).

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.

- 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).** $N$.

Flocking birds

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

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=ua7YlN4eL_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.

Impact. Running time per step is $N \log N \Rightarrow$ enables new research.
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 intervals intersect \((9, 16)\)?
A. \((7, 10)\) and \((15, 18)\).

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.

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

1d interval search API

```java
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 \((lo, hi)\)
    }
}
```

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)
```

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;
```

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: 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 \(lo \leq \text{max}\).
- Then for any interval \((a, b)\) in right subtree of \(x\),
  \(hi < c \leq a \Rightarrow\) no intersection in right.

Interval search tree: analysis

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

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

<table>
<thead>
<tr>
<th>operation</th>
<th>brute</th>
<th>interval search tree</th>
<th>best in theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert interval</td>
<td>(1)</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>
</tr>
<tr>
<td>delete interval</td>
<td>(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>(\log N)</td>
<td>(\log N)</td>
</tr>
<tr>
<td>find all intervals that intersects ((lo, hi))</td>
<td>(N)</td>
<td>(R \log N)</td>
<td>(R + \log N)</td>
</tr>
</tbody>
</table>

Order of growth of running time for \(N\) intervals

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.

**Geometric Applications of BSTs**

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

![example of orthogonal rectangle intersection]

Algorithms and Moore’s law

“Moore’s law.” Processing power doubles every 18 months.
- $197x$: check $N$ rectangles.
- $197(x+1.5)$: 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. (!)

**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.

![example of orthogonal rectangle intersection using sweep-line algorithm]

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).
- Insert $y$-intervals into ST.
- Delete $y$-intervals from ST.
- Interval searches for $y$-intervals.

**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><strong>1d range search</strong></td>
<td></td>
<td><strong>BST</strong></td>
</tr>
<tr>
<td><strong>2d orthogonal line segment intersection</strong></td>
<td><img src="example1.png" alt="Example" /></td>
<td>sweep line reduces to 1d range search</td>
</tr>
<tr>
<td><strong>kd range search</strong></td>
<td><img src="example2.png" alt="Example" /></td>
<td><strong>kd tree</strong></td>
</tr>
<tr>
<td><strong>1d interval search</strong></td>
<td><img src="example3.png" alt="Example" /></td>
<td><strong>interval search tree</strong></td>
</tr>
<tr>
<td><strong>2d orthogonal rectangle intersection</strong></td>
<td><img src="example4.png" alt="Example" /></td>
<td>sweep line reduces to 1d interval search</td>
</tr>
</tbody>
</table>