Section 2.4: Geometric Applications of BSTs

- 1d range search
- Line segment intersection
- $kd$ trees

https://algs4.cs.princeton.edu
Overview

This lecture. Intersections among geometric objects.

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

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

This lecture. Only the tip of the iceberg.
GEOMETRIC APPLICATIONS OF BSTs

- 1d range search
- line segment intersection
- kd trees
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 |
| search G to K | H I |
| count G to K | 2 |
Suppose that the keys are stored in a sorted array of length $n$. What is the worst-case running time for range search as a function of both $n$ and $R$?

A. $\Theta(\log R)$  
B. $\Theta(\log n)$  
C. $\Theta(\log n + R)$  
D. $\Theta(n + R)$
1d range search: elementary implementations

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

**Ordered array.** Slow insert; fast 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>

$n = \text{number of keys}$

$R = \text{number of keys that match}$
1d range search. Find all keys between $k_1$ and $k_2$.

- 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.** Take $\Theta(\log n + R)$ time in the worst case.

**Pf.** Nodes examined = search path to $k_1$ + search path to $k_2$ + matches.
1d range search: summary of performance

**Unordered list.** Slow insert; slow range search.  
**Ordered array.** Slow insert; fast range search.  
**BST.** Fast insert; fast range search/count.

### 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>$\log n + R$</td>
</tr>
<tr>
<td>balanced BST</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n + R$</td>
</tr>
</tbody>
</table>

*use rank() function (see precept)*  
$n =$ number of keys  
$R =$ number of keys that match
Geometric Applications of BSTs

- 1d range search
- line segment intersection
- kd trees
Orthogonal line segment intersection

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

**Quadratic algorithm.** Check all pairs of line segments for 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 = line segment (or rectangle) intersection.
Orthogonal line segment intersection: sweep-line algorithm

Non-degeneracy 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]
• \text{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.

non-degeneracy assumption: all \( x\)- and \( y\)-coordinates are distinct
Orthogonal line segment intersection: sweep-line analysis

**Proposition.** The sweep-line algorithm takes $\Theta(n \log n + R)$ time in the worst case to find all $R$ intersections among $n$ orthogonal line segments.

**Pf.**

- Sort $x$-coordinates. [ $n \log n$ ]
- Insert $y$-coordinates into BST. [ $n \log n$ ]
- Delete $y$-coordinates from BST. [ $n \log n$ ]
- Range searches in BST. [ $n \log n + R$ ]

**Bottom line.** Sweep line reduces 2d orthogonal line segment intersection search to 1d range search.
The sweep-line algorithm is a key technique in computational geometry.

Geometric intersection.
- General line segment intersection.
- Axis-aligned rectangle intersection.
- ...

More problems.
- Andrew’s algorithm for convex hull.
- Fortune’s algorithm Voronoi diagram.
- Scanline algorithm for rendering computer graphics.
- ...
Geometric Applications of BSTs

- 1d range search
- line segment intersection
- kd trees
2-d orthogonal range search

Extension of ordered symbol-table to 2d keys.

- Insert 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 **$h-v$ rectangle**

rectangle is axis-aligned
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.

Grid
Quadtree
2d tree
BSP tree
2d tree construction

Recursively partition plane into two halfplanes.
Where would point K be inserted in the 2d tree below?

A. Left child of G.
B. Left child of J.
C. Right child of J.
D. Right child of I.
2d tree implementation

**Data structure.** BST, but alternate using $x$- and $y$-coordinates as key.
- Each node corresponds to a rectangle containing point in node.
- Can store rectangle explicitly in node.
2d tree demo: range search

**Goal.** Find all points in a query rectangle.
- Check if query rectangle contains point in node.
- Recursively search left/bottom and right/top subtrees.
- Optimization: prune subtree if it can’t contain a point in rectangle.
**Goal.** Find all points in a query rectangle.

- Check if query rectangle contains point in node.
- Recursively search left/bottom and right/top subtrees.
- Optimization: prune subtree if it can’t contain a point in rectangle.
Suppose we explore the right/top subtree before the left/bottom subtree in range search. What effect would it have on typical inputs?

A. Returns wrong answer.
B. Explores more nodes.
C. Both A and B.
D. Neither A nor B.
Range search in a 2d tree analysis

Typical case. $\Theta(R + \log n)$.

Worst case (assuming tree is balanced). $\Theta(R + \sqrt{n})$. 
Goal. Find closest point to query point.

2d tree demo: nearest neighbor
2d tree demo: nearest neighbor

- Check distance from point in node to query point.
- Recursively search left/bottom and right/top subtrees.
- Optimization 1: prune subtree if it can’t contain a closer point.
- Optimization 2: explore subtree toward the query point first.

nearest neighbor = E
Suppose we always explore the left/bottom subtree before the right/top subtree in nearest-neighbor search. What effect will it have on typical inputs?

A. Returns wrong answer.
B. Explores more nodes.
C. Both A and B.
D. Neither A nor B.
Which of the following is the worst case for nearest-neighbor search?

A.

B.

C.

D.
Nearest neighbor search in a 2d tree analysis

Typical case. $\Theta(\log n)$.

Worst case (even if tree is balanced). $\Theta(n)$.
2d tree: implementation

Q. How to implement a 2d-tree?
A. Explicit node data type for binary tree.

```java
private class KdTreeST<Value>
{
    private Node root;

    private class Node
    {
        private Point2D p;
        private Value val;

        private Node left, right;
        private Node parent;

        private RectHV rect;

        ...
    }
}
```
Flocking birds

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

https://www.youtube.com/watch?v=XH-groCeKbE
Flocking boids [Craig Reynolds, 1986]

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

- **Flock centering:** point towards the center of mass of $k$ nearest boids.
- **Direction matching:** update velocity towards average of $k$ nearest boids.
- **Collision avoidance:** among $k$ nearest boids, point away from close ones.
**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 $n^2$. 

https://www.youtube.com/watch?v=ua7YIN4eL_w
Appel’s algorithm for n-body simulation

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

- Treat the 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 $n \log n \Rightarrow$ enables new research.
## 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><img src="example1.png" alt="1d range search example" /></td>
<td><em>binary search tree</em></td>
</tr>
<tr>
<td>2d orthogonal line segment</td>
<td><img src="example2.png" alt="2d orthogonal line segment example" /></td>
<td><em>sweep line reduces problem to 1d range search</em></td>
</tr>
<tr>
<td>2d range search</td>
<td><img src="example3.png" alt="2d range search example" /></td>
<td><em>2d tree</em></td>
</tr>
<tr>
<td>kd range search</td>
<td><img src="example4.png" alt="kd range search example" /></td>
<td><em>kd tree</em></td>
</tr>
</tbody>
</table>