Point location

**Point location problem:** Preprocess a planar subdivision such that for any query point $q$, the face of the subdivision containing $q$ can be given quickly (name of the face)

- From GPS coordinates, find the region on a map where you are located
- Subroutine for many other geometric problems (Chapter 13: motion planning, or shortest path computation)
**Planar subdivision:** Partition of the plane by a set of non-crossing line segments into vertices, edges, and faces

*non-crossing:* disjoint, or at most a shared endpoint
Data structuring question, so interest in query time, storage requirements, and preprocessing time

To store: set of \( n \) non-crossing line segments and the subdivision they induce
First solution

Idea: Draw vertical lines through all vertices, then do something for every vertical strip that appears
First solution
In one strip

Inside a single strip, there is a well-defined bottom-to-top order on the line segments.

Use this for a balanced binary search tree that is valid if the query point is in this strip (knowing between which edges we are is knowing in which face we are).
Solution with strips

search tree on $x$-coordinate
Solution with strips

To answer a query with $q = (q_x, q_y)$, search in the main tree with $q_x$ to find a leaf, then follow the pointer to search in the tree that is correct for the strip that contains $q_x$.

**Question:** What are the storage requirements and what is the query time of this structure?
Solution with strips

\[ \frac{n}{4} \text{ strips} \]
Solution with strips
The vertical strips idea gave a *refinement* of the original subdivision, but the number of faces went up from linear in $n$ to quadratic in $n$.

Is there a different refinement whose size remains linear, but in which we can still do point location queries easily?
Vertical decomposition

Suppose we draw vertical extensions from every vertex up and down, *but only until the next line segment*

- Assume the input line segments are not vertical
- Assume every vertex has a distinct $x$-coordinate
- Assume we have a bounding box $R$ that encloses all line segments that define the subdivision

This is called the *vertical decomposition* or *trapezoidal decomposition*
Vertical decomposition
The vertical decomposition has triangular and trapezoidal faces
Vertical decomposition faces

Every face has a vertical left and/or right side that is a vertical extension, and is bounded from above and below by some line segment of the input.

The left and right sides are defined by some endpoint of a line segment.

\[ \text{leftp}(\Delta) \quad \Delta \quad \text{rightp}(\Delta) \]

\[ \text{top}(\Delta) \quad \text{bottom}(\Delta) \]
Vertical decomposition faces

Every face is defined by no more than four line segments.

For any face, we ignore vertical extensions that end on top($\Delta$) and bottom($\Delta$).
Two trapezoids (including triangles) are *neighbors* if they share a vertical side.

Each trapezoid has 1, 2, 3, or 4 neighbors.
Neighbors of faces

A trapezoid could have many neighbors if vertices had the same $x$-coordinate
We could use a DCEL to represent a vertical decomposition, but we use a more direct & convenient structure

- Every face \( \Delta \) is an object; it has fields for \( \text{top}(\Delta) \), \( \text{bottom}(\Delta) \), \( \text{leftp}(\Delta) \), and \( \text{rightp}(\Delta) \) (two line segments and two vertices)
- Every face has fields to access its up to four neighbors
- Every line segment is an object and has fields for its endpoints (vertices) and the name of the face in the original subdivision directly above it
- Each vertex stores its coordinates
Representation

\[
\begin{array}{c}
\text{R}
\end{array}
\]
Any trapezoid $\Delta$ can find out the name of the face it is part of via $\text{bottom}(\Delta)$ and the stored name of the face.
A vertical decomposition of $n$ non-crossing line segments inside a bounding box $R$, seen as a proper planar subdivision, has at most $6n + 4$ vertices and at most $3n + 1$ trapezoids.
The input to planar point location is a planar subdivision, for example in DCEL format

First, store with each edge the name of the face above it (our structure will find the edge below any query point)

Then extract the edges to define a set $S$ of non-crossing line segments; ignore the DCEL otherwise
Point location solution

We will use *randomized incremental construction* to build, for a set $S$ of non-crossing line segments,

- a vertical decomposition $T$ of $S$ and $R$
- a search structure $D$ whose leaves correspond to the trapezoids of $T$

The simple idea: Start with $R$, then add the line segments in random order and maintain $T$ and $D$
Point location solution

Let $s_1, \ldots, s_n$ be the $n$ line segments in random order

Let $T_i$ be the vertical decomposition of $R$ and $s_1, \ldots, s_i$, and let $D_i$ be the search structure obtained by inserting $s_1, \ldots, s_i$ in this order.
Let $s_1, \ldots, s_n$ be the $n$ line segments in random order

Let $T_i$ be the vertical decomposition of $R$ and $s_1, \ldots, s_i$, and let $D_i$ be the search structure obtained by inserting $s_1, \ldots, s_i$ in this order.
The search structure $D$ has $x$-nodes, which store an endpoint, and $y$-nodes, which store a line segment $s_j$.

For any query point $t$, we only test at an $x$-node: Is $t$ left or right of the vertical line through the stored point?

For any query point $t$, we only test at an $y$-node: Is $t$ below or above the stored line segment?

We will guarantee that the question at a $y$-node is only asked if the query point $t$ is between the vertical lines through $p_j$ and $q_j$, if line segment $s_j = p_jq_j$ is stored.
Point location solution

\[ \Delta_1 \]
\[ s_1 \]
\[ p_1 \]
\[ q_1 \]
\[ \Delta_2 \]
\[ \Delta_3 \]
\[ \Delta_4 \]

\[ T_1 \]
\[ D_1 \]
\[ p_1 \]
\[ q_1 \]
\[ s_1 \]

\[ R \]
\[ \Delta_1 \]
\[ \Delta_2 \]
\[ \Delta_3 \]
\[ \Delta_4 \]
Point location solution

\[ \Delta_1 \quad \Delta_2 \quad \Delta_3 \quad \Delta_4 \]

\[ s_1 \quad p_1 \quad q_1 \]

\[ s_2 \quad p_2 \quad q_2 \]

\[ T_1 \quad D_1 \quad p_1 \quad q_1 \quad s_1 \]

\[ R \quad \Delta_1 \quad \Delta_2 \quad \Delta_3 \quad \Delta_4 \]
Point location solution

The search structure
Updating the vertical decomposition
Updating the search structure

Computational Geometry
Lecture 9: Planar point location
Point location solution

The search structure
Updating the vertical decomposition
Updating the search structure

Introduction
Vertical decomposition
Analysis

Computational Geometry
Lecture 9: Planar point location
Point location solution

- **Introduction**
- Vertical decomposition
- Analysis
- The search structure
- Updating the vertical decomposition
- Updating the search structure

**Computational Geometry**

**Lecture 9: Planar point location**
Point location solution

We want only one leaf in $D$ to correspond to each trapezoid; this means we get a search graph instead of a search tree.

It is a directed acyclic graph, or DAG, hence the name $D$. 
Point location solution

The search structure

Updating the vertical decomposition

Updating the search structure

Computational Geometry

Lecture 9: Planar point location
A point location query is done by following a path in the search structure $D$ to a leaf, then following its pointer to a trapezoid of $T$, then accessing $\text{bottom}(\ldots)$ of this trapezoid, and reporting the name of the face stored with it.
Point location query

\[ \Delta_1, \Delta_3, \Delta_8, \Delta_9, \Delta_5, \Delta_6, \Delta_7 \]

\[ p_1, q_1, p_2, q_2 \]

\[ s_1, s_2 \]

\[ T_2, D_2 \]
Suppose we have $D_{i-1}$ and $T_{i-1}$, how do we add $s_i$?

Because $D_{i-1}$ is a valid point location structure for $s_1, \ldots, s_{i-1}$, we can use it to find the trapezoid of $T_{i-1}$ that contains $p_i$, the left endpoint of $s_i$.

Then we use $T_{i-1}$ to find all other trapezoids that intersect $s_i$. 
Find intersected trapezoids

\[ \Delta_0 \]

\[ p_i \]

\[ s_i \]

\[ q_i \]
Find intersected trapezoids

\[ \Delta_0 \]

\[ p_i \]

\[ s_i \]

\[ q_i \]
Find intersected trapezoids

\[ \Delta_0 \quad \Delta_1 \quad \Delta_2 \quad \Delta_3 \quad \Delta_4 \quad \Delta_5 \quad \Delta_6 \quad \Delta_7 \]

\( p_i \quad s_i \quad q_i \)
Find intersected trapezoids
Find intersected trapezoids

After locating the trapezoid that contains $p_i$, we can determine all $k$ trapezoids that intersect $s_i$ in $O(k)$ time by traversing $T_{i-1}$. 
Updating the vertical decomposition
Updating the vertical decomposition

\[ \Delta_0, \Delta_1, \Delta_2, \Delta_3, \Delta_4, \Delta_5, \Delta_6, \Delta_7 \]
Updating the vertical decomposition
We can update the vertical decomposition in $O(k)$ time as well.
The search structure has \( k \) leaves that are no longer valid as leaves; they become internal nodes.

We find these using the pointers from \( T_{i-1} \) to \( D_{i-1} \).

From the update of the vertical decomposition \( T_{i-1} \) into \( T_i \) we know what new leaves we must make for \( D_i \).

All new nodes besides the leaves are \( x \)-nodes with \( p_i \) and \( q_i \) and \( y \)-nodes with \( s_i \).
Updating the search structure
Updating the search structure

leaves for the new trapezoids in $T_i$
Updating the search structure

$D_{i-1}$

$T_i$

leaves for the new trapezoids in $T_i$
Updating the search structure

$T_i$  

$D_{i-1}$  

Leaves for the new trapezoids in $T_i$
Observations

For a single update step, adding $s_i$ and updating $T_{i-1}$ and $D_{i-1}$, we observe:

- If $s_i$ intersects $k_i$ trapezoids of $T_{i-1}$, then we will create $O(k_i)$ new trapezoids in $T_i$.
- We find the $k_i$ trapezoids in time linear in the search path of $p_i$ in $D_{i-1}$, plus $O(k_i)$ time.
- We update by replacing $k_i$ leaves by $O(k_i)$ new internal nodes and $O(k_i)$ new leaves.
- The maximum depth increase is three nodes.
Questions

Question: In what case is the depth increase three nodes?

Question: We noticed that the directed acyclic graph $D$ is binary in its out-degree, what is the maximum in-degree?
A common but special update

If $p_i$ was already an existing vertex, we search in $D_{i-1}$ with a point a fraction to the right of $p_i$ on $s_i$.
Randomized incremental construction, where does it matter?

- The vertical decomposition $T_i$ is independent of the insertion order among $s_1, \ldots, s_i$
- The search structure $D_i$ can be different for many orders of $s_1, \ldots, s_i$
- The number of nodes in $D_i$ depends on the order
- The depth of search paths in $D_i$ depends on the order
Randomized incremental construction
Randomized incremental construction

$p_1

q_1

s_1

p_2

q_2

s_2

p_3

q_3

s_3

p_4

q_4

s_4

p_5

q_5

s_5

s_1s_2 s_3 s_4 s_5
Randomized incremental construction
Storage of the structure

The vertical decomposition structure $T$ always uses linear storage.

The search structure $D$ can use anything between linear and quadratic storage.

We analyse the expected number of new nodes when adding $s_i$, using backwards analysis (of course).
Backwards analysis in this case: Suppose we added $s_i$ and have computed $T_i$ and $D_i$. All line segments (existing in $T_i$) had the same probability of having been the last one added.
For each of the $i$ line segments, we can see how many trapezoids would have been created if it were the last one added.
For each of the $i$ line segments, we can see how many trapezoids would have been created if it were the last one added.
For each of the $i$ line segments, we can see how many trapezoids would have been created if it were the last one added.
Storage of the structure

The number of created trapezoids is linear in the number of deleted trapezoids (leaves of $D_{i-1}$), or intersected trapezoids by $s_i$ in $T_{i-1}$; this is linear in $k_i$.

We will analyze

$$K_i = \sum_{j=1}^{i} \text{[no. of trapezoids created if } s_j \text{ were last]}$$
Consider $K_i$ from the "trapezoid perspective": For any trapezoid $\Delta$, there are at most four line segments whose insertion would have created it (top($\Delta$), bottom($\Delta$), leftp($\Delta$), and rightp($\Delta$)).
Consider $K_i$ from the “trapezoid perspective”: For any trapezoid $\Delta$, there are at most four line segments whose insertion would have created it (top($\Delta$), bottom($\Delta$), leftp($\Delta$), and rightp($\Delta$)).
Consider $K_i$ from the “trapezoid perspective”: For any trapezoid $\Delta$, there are at most four line segments whose insertion would have created it (top($\Delta$), bottom($\Delta$), leftp($\Delta$), and rightp($\Delta$)).
We know: There are at most $3i + 1$ trapezoids in a vertical decomposition of $i$ line segments in $R$

Hence,

$$K_i = \sum_{\Delta \in T_i} \text{[no. of segments that would create } \Delta]\]$$

$$\leq \sum_{\Delta \in T_i} 4 = 12i + 4$$
Since $K_i$ is defined as a sum over $i$ line segments, the average number of trapezoids in $T_i$ created by $s_i$ is at most $(12i + 4)/i \leq 13$.

Since the expected number of new nodes is at most 13 in every step, the expected size of the structure after adding $n$ line segments is $O(n)$.
Query time of the structure

Fix any point $q$ in the plane as a query point, we will analyze the probability that inserting $s_i$ makes the search path to $q$ longer.
**Backwards analysis:** Take the situation after $s_i$ has been added, and ask the question: How many of the $i$ line segments made the search path to $q$ longer?
**Backwards analysis:** Take the situation after $s_i$ has been added, and ask the question: How many of the $i$ line segments made the search path to $q$ longer?

The search path to $q$ only became longer if $q$ is in a trapezoid that was *just created* by the latest insertion!

At most four line segments define the trapezoid that contains $q$, so the probability is $4/i$.
We analyze

\[ \sum_{i=1}^{n} \text{[search path became longer due to } i\text{-th addition]} \]

\[ \leq \sum_{i=1}^{n} \frac{4}{i} = 4 \cdot \sum_{i=1}^{n} \frac{1}{i} \leq 4(1 + \log_e n) \]

So the expected query time is \( O(\log n) \)
**Theorem:** Given a planar subdivision defined by a set of $n$ non-crossing line segments in the plane, we can preprocess it for planar point location queries in $O(n \log n)$ expected time, the structure uses $O(n)$ expected storage, and the expected query time is $O(\log n)$.