# CMSC 754: Lecture 9
# Trapezoidal Maps and Planar Point Location

**Reading:** Chapter 6 of the 4M's.

**Point Location:** In *planar point location* we are given a polygonal subdivision of the plane, and the objective is to preprocess this subdivision into a data structure so that given a query point $q$, it is possible to efficiently determine which face of the subdivision contains $q$ (see Fig. 1(a)). For example, the subdivision might represent government subdivisions, such as countries, states, or counties, and we wish to identify the country, state, or county of a point given its GPS coordinates.

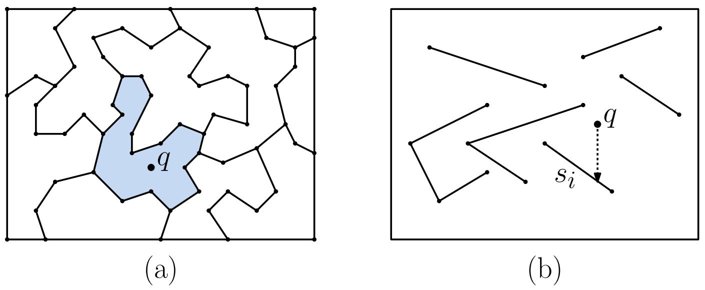

Fig. 1: (a) point location and (b) vertical ray-shooting queries.

It will be useful to generalize the above problem. Rather than assuming that the input is a subdivision of space into cells (what is commonly referred to as a *cell complex*), we will assume that the input is merely a set of $n$ line segments $S = \{s_1, \ldots, s_n\}$. The objective is to answer *vertical ray-shooting queries*, which means, given a query point $q$, what line segment $s_i$ (if any) lies immediately below the query point (see Fig. 1(b)). Observe that the ability to answer vertical ray-shooting queries implies that point-location queries can be answered. We simply label each segment with the identity of the subdivision cell that lies immediately above it.

We will make the usual general-position assumption that no two segment endpoints share the same $x$-coordinate (and hence there are no vertical lines), and that the query point does not lie on any segment nor directly above a segment endpoint.

For many years the best methods known for solving planar point location had an extra log factor, either in the space or in the query time. (That is, the space was $O(n \log n)$ or the query time was $O(\log^2 n)$. David Kirkpatrick achieved a breakthrough by presenting a time/space optimal algorithm. Kirkpatrick's algorithm has fairly high constant factors. Somewhat simpler and more practical optimal algorithms were discovered since then.

**Recap of Trapezoidal Maps:** Our point-location data structure will be based on the randomized trapezoidal map construction from the previous lecture. In that lecture we showed that a trapezoidal map of $O(n)$ space could be constructed in (randomized) $O(n \log n)$ expected time. In this lecture we show how to modify the construction so that, as a by product, we obtain a data structure for answering vertical ray-shooting queries. The preprocessing time for the data structure will also be $O(n \log n)$ in the expected case, the space required for the data structure will be $O(n)$, and the query time will be $O(\log n)$. The latter two bounds will hold unconditionally.

Let us recap some of the concepts from the previous lecture. Recall that the input as a set of segments in the plane $S = \{s_1, \ldots, s_n\}$ in the plane, which are assumed to have been randomly permuted. Let $S_i$ denotes the subset consisting of the first $i$ segments of $S$. Let $\mathcal{T} = \mathcal{T}(S)$ denote the trapezoidal map of $S$, which is the subdivision generated by shooting vertical rays both upwards and downwards from each line-segment endpoint until striking another segment (or hitting the bounding box of the input). Let $\mathcal{T}_i$ denote the trapezoidal map of $S_i$.

Recall from the previous lecture that each time we add a new line segment, it may result in the creation of the collection of new trapezoids, which are said to *depend* on this line segment. We showed that (under the assumption of the random insertion order) the expected number of new trapezoids that are created with each stage is $O(1)$. This fact will be used later in this lecture.

**Point Location Data Structure:** The point location data structure is based on a rooted directed acyclic graph (DAG). Each node will have either zero or two outgoing edges. Nodes with zero outgoing edges are called *leaves*. The leaves will be in 1–1 correspondence with the trapezoids of the map. The other nodes are called *internal nodes*, and they are used to guide the search to the leaves. This DAG can be viewed as a variant of a binary tree, where subtrees may be shared between different nodes. (This sharing is important for keeping the space to $O(n)$.)

There are two types of internal nodes, *$x$-nodes* and *$y$-nodes*. Each $x$-node contains the point $p$ (an endpoint of one of the segments), and its two children correspond to the points lying to the left and to the right of the vertical line passing through $p$ (see Fig. 2(a)). Each $y$-node contains a pointer to a line segment of the subdivision, and the left and right children correspond to whether the query point is above or below the line containing this segment, respectively (see Fig. 2(b)). (Don't be fooled by the name—$y$-node comparisons depend on both the $x$ and $y$ values of the query point.) Note that the search will reach a $y$-node only if we have already verified that the $x$-coordinate of the query point lies within the vertical slab that contains this segment.

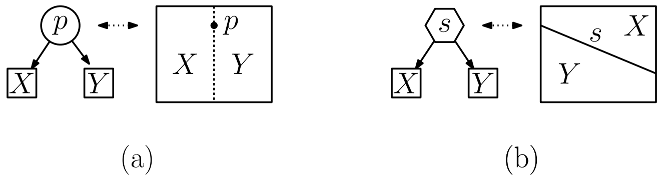

Fig. 2: (a) $x$-node and (b) $y$-node.

Our construction of the point location data structure mirrors the incremental construction of the trapezoidal map, as given in the previous lecture. In particular, if we freeze the construction just after the insertion of any segment, the current structure will be a point location structure for the current trapezoidal map.

In Fig. 3 below we show a simple example of what the data structure looks like for two line segments. For example, if the query point is in trapezoid $D$, we would first detect that it is to the right of enpoint $p_1$ (right child), then left of $q_1$ (left child), then below $s_1$ (right child), then right of $p_2$ (right child), then above $s_2$ (left child).

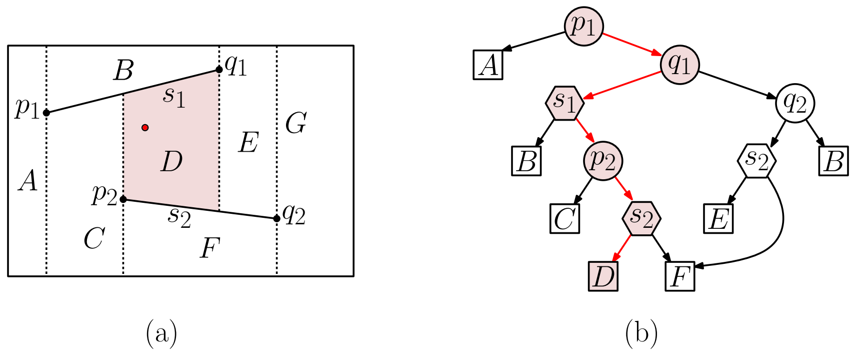

Fig. 3: Trapezoidal map point location data structure.

**Incremental Construction:** The question is how do we build this data structure incrementally? First observe that when a new line segment is added, we only need to adjust the portion of the tree that involves the trapezoids that have been deleted as a result of this new addition. Each trapezoid that is deleted will be replaced with a search structure that determines the newly created trapezoid that contains it.

Suppose that we add a line segment $s$. This results in the replacement of an existing set of trapezoids with a set of new trapezoids. As a consequence, we will replace the leaves associated with each such deleted trapezoid with a local search structure, which locates the new trapezoid that contains the query point. There are three cases that arise, depending on how many endpoints of the segment lie within the current trapezoid.

**Single (left or right) endpoint:** A single trapezoid $A$ is replaced by three trapezoids, denoted $X$, $Y$, and $Z$. Letting $p$ denote the endpoint, we create an $x$-node for $p$, and one child is a leaf node for the trapezoid $X$ that lies outside vertical projection of the segment. For the other child, we create a $y$-node whose children are the trapezoids $Y$ and $Z$ lying above and below the segment, respectively (see Fig. 4(a)).

**Two segment endpoints:** This happens when the segment lies entirely inside the trapezoid. In this case one trapezoid $A$ is replaced by four trapezoids, $U$, $X$, $Y$, and $Z$. Letting $p$ and $q$ denote the left and right endpoints of the segment, we create two $x$-nodes, one for $p$ and the other for $q$. We create a $y$-node for the line segment, and join everything together (see Fig. 4(b)).

**No segment endpoints:** This happens when the segment cuts completely through a trapezoid. A single trapezoid is replaced by two trapezoids, one above and one below the segment, denoted $Y$ and $Z$. We replace the leaf node for the original trapezoid with a $y$-node whose children are leaf nodes associated with $Y$ and $Z$ (see Fig. 4(c)).

It is important to notice that (through sharing) each trapezoid appears exactly once as a leaf in the resulting structure. How does this sharing occur? Whenever we add a segment,

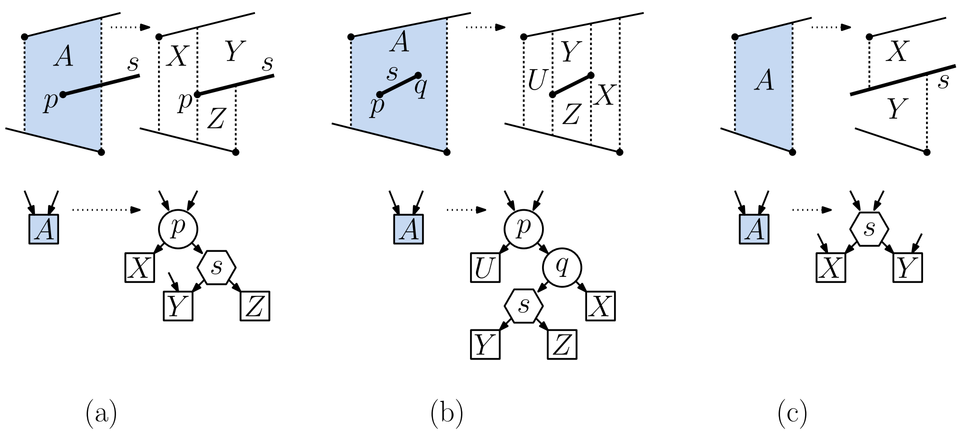

Fig. 4: Line segment insertion and updates to the point location structure. The single-endpoint case (left) and the two-endpoint case (right). The no-endpoint case is not shown.

the wall trimming that results can result in two distinct trapezoids being merged into one (see trapezoid $Y$ in Fig. 5(a) and $X$ and $Y$ in Fig. 5(b)). When this happens, the various paths leading into merged trapezoid are joined to a common node. An example showing the complete transformation to the data structure after adding a single segment is shown in Fig. 5 below.

**Analysis:** We claim that the size of the point location data structure is $O(n)$ and the query time is $O(\log n)$, both in the expected case. As usual, the expectation depends only on the order of insertion, not on the line segments or the location of the query point.

To prove the space bound of $O(n)$, observe that the number of new nodes added to the structure with each new segment is proportional to the number of newly created trapezoids. Last time we showed that with each new insertion, the expected number of trapezoids that were created was $O(1)$. Therefore, we add $O(1)$ new nodes with each insertion in the expected case, implying that the total size of the data structure is $O(n)$.

Analyzing the query time is a little subtler. In a normal probabilistic analysis of data structures we think of the data structure as being fixed, and then compute expectations over random queries. Here the approach will be to imagine that we have exactly one query point to handle. The query point can be chosen arbitrarily (imagine an adversary that tries to select the worst-possible query point) but this choice is made without knowledge of the random choices the algorithm makes. We will show that, given a fixed query point $q$, the expected search path length for $q$ is $O(\log n)$, where the expectation is over all segment insertion orders. (Note that this does not imply that the expected maximum depth of the tree is $O(\log n)$. We will discuss this issue later.)

Let $q$ denote the query point. Rather than consider the search path for $q$ in the final search structure, we will consider how $q$ moves incrementally through the structure with the addition of each new line segment. Let $\Delta_i$ denote the trapezoid of the map that $q$ lies in after the insertion of the first $i$ segments. Observe that if $\Delta_{i-1} = \Delta_i$, then insertion of the $i$th segment did not affect the trapezoid that $q$ was in, and therefore $q$ will stay where it is relative to the

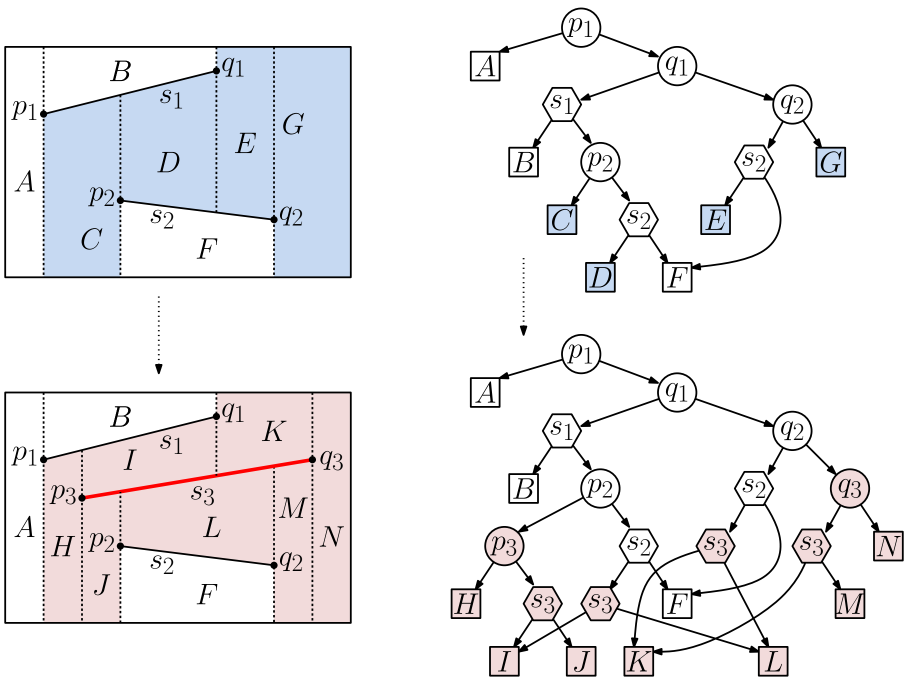

Fig. 5: Line segment insertion.

current search structure. (For example, if $q$ was in trapezoid $B$ prior to adding $s_3$ in Fig. 5 above, then the addition of $s_3$ does not incur any additional cost to locating $q$.)

However, if $\Delta_{i-1} \neq \Delta_i$, then the insertion of the $i$th segment caused $q$'s trapezoid to be replaced by a different one. As a result, $q$ must now perform some additional comparisons to locate itself with respect to the newly created trapezoids that overlap $\Delta_{i-1}$. Since there are a constant number of such trapezoids (at most four), there will be $O(1)$ work needed to locate $q$ with respect to these. In particular, $q$ may descend at most three levels in the search tree after the insertion. The worst case occurs in the two-endpoint case, where the query point falls into one of the trapezoids lying above or below the segment (see Fig. 4(b)).

Since a point can descend at most three levels with each change of its containing trapezoid, the expected length of the search path to $q$ is at most three times the number of times that $q$ changes its trapezoid as a result of each insertion. For $1 \leq i \leq n$, let $X_i(q)$ denote the random event that $q$ changes its trapezoid after the $i$th insertion, and let $\text{Prob}(X_i(q))$ denote the probability of this event. Letting $D(q)$ denote the average depth of $q$ in the final search tree, we have

$$D(q) \ \leq \ 3 \sum_{i=1}^{n} \text{Prob}(X_i(q)).$$

What saves us is the observation that, as $i$ becomes larger, the more trapezoids we have, and the smaller the probability that any random segment will affect a given trapezoid. In particular, we will show that $\text{Prob}(X_i(q)) \leq 4/i$. We do this through a backwards analysis. Consider the trapezoid $\Delta_i$ that contained $q$ *after* the $i$th insertion. Recall from the previous lecture that each trapezoid is dependent on at most four segments, which define the top and bottom edges, and the left and right sides of the trapezoid. Clearly, $\Delta_i$ would have changed as a result of insertion $i$ if any of these four segments had been inserted last. Since, by the random insertion order, each segment is equally likely to be the last segment to have been added, the probability that one of $\Delta_i$'s dependent segments was the last to be inserted is at most $4/i$. Therefore, $\text{Prob}(X_i(q)) \leq 4/i$.

From this, it follows that the expected path length for the query point $q$ is at most

$$D(q) \;\leq\; 3\sum_{i=1}^{n}\frac{4}{i} \;=\; 12\sum_{i=1}^{n}\frac{1}{i}.$$

Recall that $\sum_{i=1}^{n}\frac{1}{i}$ is the Harmonic series, and for large $n$, its value is very nearly $\ln n$. Thus we have

$$D(q) \;\leq\; 12\cdot\ln n \;=\; O(\log n).$$

**Guarantees on Search Time: (Optional)** One shortcoming with this analysis is that even though the search time is provably small in the expected case for a given query point, it might still be the case that once the data structure has been constructed there is a single very long path in the search structure, and the user repeatedly performs queries along this path. Hence, the analysis provides no guarantees on the running time of all queries.

It is far from trivial, but it can be shown that by repeated application of the randomized incremental construction, it is possible to achieve worst-case search time of $O(\log n)$, worst-case size of $O(n)$, and expected-case construction time is $O(n \log n)$.[1] The idea is to engineer the constants so that the probability of failure along any search path is extremely small (say $1/n^c$, for some constant $c \geq 1$). It follows that all the possible search paths will have the desired $O(\log n)$ depth with at least a constant probability. While we might be unlucky on any given execution of the algorithm, after a constant number of attempts, we expect one of them to succeed.

**Line Segment Intersection Revisited: (Optional)** Earlier this semester we presented a plane-sweep algorithm for computing line segment intersection. The algorithm had a running time of $O((n + I) \log n)$, where $I$ is the number of intersection points. It is interesting to note that the randomized approach we discussed today can be adapted to deal with intersecting segments as well. In particular, whenever a segment is added, observe that in addition to it stabbing vertical segments, it may generally cross over one of the existing segments. When this occurs, the algorithm must determine the trapezoid that is hit on the other side of the segment, and then continue the process of walking the segment. Note that the total size of the final decomposition is $O(n + I)$, which would suggest that the running time might be the same as the plane-sweep algorithm. It is remarkable, therefore, that the running time is actually better. Intuitively, the reason is that the $O(\log n)$ factor in the randomized algorithm comes from the point location queries, which are applied only to the left endpoint of each of the $n$ segments. With a bit of additional work, it can be shown that the adaptation of the randomized algorithm to general (intersecting) segments runs in $O(I + n \log n)$ time, thus removing the log factor from the $I$ term.

[1] M. Hemmer, M. Kleinbort, and D. Halperin. Optimal randomized incremental construction for guaranteed logarithmic planar point location. *Comput. Geom.*, 58:110–123, 2016.