9.1 Windowing Queries

Windowing queries answer important questions. Primarily, windowing queries answer information involving a given line segment and a windowing region. Given a set of \( n \) line segments \( S \) and a query window \( W := [x : x'] [y : y'] \), windowing queries answer which segments in \( S \) intersect \( W \).

The primary difference between windowing queries and range queries is the data. While range queries are associated with points, windowing queries are associated with line segments, polygons, curves, etc. Typically, windowing queries exist in 2- or 3-dimensions, whereas range queries deal with queries in higher-dimensional spaces.

In the examples below, we will visit three types of windowing queries, two of which consider the specific case regarding axis-aligned segments, and one which analyzes arbitrarily oriented line segments.

In every structure, all segments are assumed to be non-intersecting.

9.2 Interval Trees

Interval trees [9.1] present suitable windowing queries for axis-parallel line segments and an axis-parallel query window. Given the set \( S \) of \( n \) axis-parallel line segments and an axis-parallel query window \( W := [x : x'] \times [y : y'] \), the segments \( S \) intersecting \( W \) can be reported efficiently.

Intersecting segments in \( S \) are defined by having at least one endpoint inside \( W \). There different cases which satisfy this condition include: the segment can lie entirely inside \( W \), it can the boundary of \( W \) once, twice, or partially overlap the boundary of \( W \).

In the cases where one endpoint lies inside \( W \), we can use a 2-dimensional range query, which uses \( O(n \log n) \) storage and can be answered in \( O(\log^2 n + k) \) time. The use of fractional cascading can reduce query time to \( O(\log n + k) \).

In the other cases, where a segment in \( S \) crosses the boundary of \( W \) twice or lies on the edge of a boundary, another approach must be used. This is where the interval tree is needed.
Lines segments which cross the boundary of $W$ twice or which lie on the edge of a boundary can be queried against two of the four bounding edges: the vertical left edge of $W$ and the horizontal bottom edge of $W$. The remaining two edges do not need to be tested.

These tests can be simplified by representing each segment as an interval and each testing edge as a horizontal or vertical line rather than a line segment. In the remaining discussion, we will consider testing the horizontal segments $S_H \subseteq S$ against the left vertical edge of $W$: $\ell := (x = q_x)$; the bottom horizontal edge test is similar.

Therefore, a horizontal segment $s := (x, y)(x', y)$ which satisfies the edge test is a segment which is intersected by $\ell$. This condition occurs only if and only if $x \leq q_x \leq x'$. By using this representation, we have simplified the original problem into a 1-dimensional test: given a set of intervals, report the ones that contain the query point $q_x$.

To build the testing structure, we must build a balanced binary search tree according to the endpoints of the given segments in $S_H$. Each node in the tree corresponds to a vertical line, which intersects the median of the subspace of endpoints. Each leaf in the tree corresponds to an endpoint of an interval in $S_H$. Rather than splitting the set of intervals which spans the vertical partitioning line, these intervals are stored at the node itself.

A query for this structure involves testing against a given vertical line, represented as the point $q_x$. By tracing through the tree, it can easily be determined if $q_x$ lies to the left or right of a partitioning point $p$ [9.2][9.3]. Each node through the query path represents a list of candidate intervals which may possibly span $q_x$. The storage required at each node is represented by an
Figure 9.2: Sample query. $q_x$ represents a vertical partitioning line. $I_{\text{mid}}$ represents intervals which span the partitioning line, whereas $I_{\text{left}}$ and $I_{\text{right}}$ represent intervals which lie on either side.

associative structure. The choice of associative structure has a great impact both the storage and query complexity.

The nodes, then, must also be queried against the candidate point $q_x$. By sorting the intervals in each node by its endpoints' $x$-coordinates, we can easily iterate across these intervals, reporting each interval that satisfies the condition. If the candidate point $q_x$ lies to the left of the partitioning point $p$, then only left endpoints of the intervals need to be evaluated. Similarly, if $q_x$ lies to the right of $p$, only the right endpoints of the intervals need to be evaluated. Therefore, two sorted lists are necessary to efficiently represent the intervals. By iterating across the sorted list, we can easily determine all intervals which span $q_x$, and therefore, $k + 1$ tests must be made and $k$ intervals will be reported at each node.

Because each interval is a balanced binary tree, its depth is $O(\log n)$. The storage bound for the interval tree is $O(n)$, because each interval is only stored in one node and appears only once in the two sorted lists. The construction time for building the interval tree is $O(n \log n)$.

The query time for the interval tree is $O(\log n + k)$. This is due to the fact that $O(k + 1)$ time is spent per node to report intervals while traversing the tree, which has a depth of $O(\log n)$.

This analysis is not quite over, however, as we simplified our original problem from intersections against a vertical line segment to intersections against a vertical line. Therefore, our current method will include segments which lie outside the windowing range!

In order to rectify our initial algorithm one extra step must be taken. Instead of iterating
across intervals at a given node, we must perform a range query on the endpoints, to verify that their $y$-coordinate is within the given range. This additional structure comes with some penalty, however. The storage is increased by a factor of $O(\log n)$ increasing total storage of the structure to $O(n \log n)$. The query time is also increased, because we must spend $O(\log n + k)$ time at each node, thereby increasing total query time to $O(\log^2 n + k)$. Note that the query time will increase by a factor $O(\log^2 n)$ if fractional cascading is not used with the range query.

### 9.3 Priority Search Trees

As previously seen, we can represent a windowing query with an interval tree to obtain efficient space, construction, and query bounds. However, the use of a range tree as the associative structure for interval nodes can be improved upon. The priority search tree is an associative structure that has an improved storage complexity of $O(n)$ instead of the $O(n \log n)$ bound of the range tree.

The key advantage of a priority search tree is its representational structure as a heap. Normally heaps are associated with priority queries and also take advantage of efficient insertion and deletion properties. Here, the heap is structured to take advantage of analyzing the $y$-coordinate in an efficient manner.

Creation of the heap structure is obviously the most important aspect of this structure. It must be organized in such a way that we can easily analyze nodes according to spatial properties that will keep an efficient query time while exploiting the storage properties of the heap [9.4]. The creation is as follows:
The root node represents the smallest $x$-coordinate.

Partition the rest of the endpoints based on the median $y$-coordinate. This ensures a balanced partitioning.

Of each partition, its root node represents the smallest $x$-coordinate.

The recursive construction of a priority search tree can be done in $O(n \log n)$ time.

A query through a priority search tree involves iterating through the tree, while continually analyzing the nodes $y$-coordinate for inclusion in the given range $[q_y : q'_y]$. The valid range is represented by the paths created by tracing $q_y$ and $q'_y$ through the tree. The interior nodes traced by this path represents all endpoints (and therefore intervals) within the range.

Each visited node of the priority search tree requires $O(1)$ time and $O(\log n + k)$ total time is spent querying the tree, because at most $O(\log n)$ search paths are necessary to evaluate all interior nodes outlined by $q_y$ and $q'_y$. 
Along with the more efficient storage complexity of priority search trees, there is also an ease of use. While the use of fractional cascading with range trees gives the same query complexity, the algorithmic complexity is much greater. The simplicity of priority search trees makes them the preferred associative structure for interval trees.

Note that the entire storage complexity of the interval tree remains $O(n \log n)$ because despite the fact that the associative structure is of linear size, a range tree is still required to find line segments which intersect the window at only one point and whose endpoints which lie within the window. This structure improves upon the case where we must find segments which intersect the window in two places whose endpoint lie outside the window.

### 9.4 Segment Trees

Segment trees are window query structures which allow for querying non axis-parallel line segments. Once again, this method is only valid for non-intersecting line segments. The example presented finds vertical intersections; a similar method will find horizontal intersections.

In order to represent arbitrarily oriented line segments, we must first dissect them into elementary intervals. This is done by first sorting all endpoints by $x$-coordinate. There elementary intervals are then described as:

$(-\infty : p_1), [p_1 : p_2), [p_2 : p_3], \ldots, (p_{m-1} : p_m), [p_m : p_m), (p_m : \infty).$ \hfill (9.1)

The answer to a query is not necessarily the same at the interior of an elementary interval and at its endpoints, which is why there are closed intervals at the endpoints and open intervals otherwise.

Then, a binary search tree is created such that the leaves are the elementary intervals. An elementary interval at leaf $\mu$ is designated as $\text{Int}(\mu)$. If all of the intervals were stored at leaves, then it would be quite easy to query for a given point $q_x$ in $O(\log n + k)$ time. This is done by marching through to the leaf in $O(\log n)$ time on a point $q_x$ and reporting all intervals which span the elementary interval in $O(k + 1)$ time.

Unfortunately, storing all the intervals in leaves would be very redundant and result in very large storage complexity. Therefore, it would be smarter to represent the intervals in a method that would guarantee the least amount of storage possible. This property can be exploited by realizing that storing the intervals in the interior nodes can result in the least storage possible if the interior node represents an aggregation of leaves which the interval spans. Each node, then, represents the canonical subset of a given interval [9.5].

This representation results in storing any given interval a maximum of two times at any given depth. This is easily proven from the realization that a line segment that spans more than two nodes at a given depth can be propagated up to a shared parent node between two given nodes. Therefore, the segment tree has a storage complexity of $O(n \log n)$, because the depth of the tree is $O(\log n)$ with a potential of storing any given interval twice at every depth.

A query in the segment tree can be done in $O(\log n + k)$ time, because it takes $O(\log n)$ time to
Figure 9.5: Segment Tree: Shows elementary intervals and canonical representation.
traverse the tree and $O(k + 1)$ time per node to report each set of intervals in the node.

The segment tree can be constructed in $O(n \log n)$ time. The procedure for constructing the segment first involves sorting all endpoints by $x$-coordinate. The construction of the balanced binary tree on the elementary intervals can be done in a bottom-up fashion in linear time. Each segment is then inserted into the tree to determine its canonical subsets and placement in the nodes of the tree. Each insertion of a given interval starts at the root and traces through the tree. These insertions require at most 4 evaluations for canonical subsets per depth, which results in a total complexity of $O(\log n)$ per interval. Over all $n$ intervals, this results in a construction complexity of $O(n \log n)$.

Because of the storage complexity of $O(n \log n)$, care should be taken to ensure that a given set of line segments cannot be represented by an interval tree, which only has a storage complexity of $O(n)$.

Once again, this only takes into consideration line segments which intersect a given vertical line, not a query window. So, we must take the $y$-range into account. A node in the segment tree forms a slab, which represents the encompassing interval and all $y$-values $(-\infty : +\infty)$. A given line segment is a canonical subset of a node in the tree if it completely crosses or spans the slab but not the slab of the corresponding parent to the node.

This also gives an important ordering property in each slab. Because none of the line segments intersect each other in a given slab, we can sort the segments in a slab according to vertical order. When comparing a line segment $s$ with a given query segment $q := q_x[q_y : q'_y]$, the two intersect if and only if the lower end point of $q$ is below $s$ and the upper endpoint of $q$ is above $s$.

Since we can sort the line segments in a given slab according to vertical order, we find all segments intersecting with $q$ in $O(\log n + k)$ time if we use a binary search tree, resulting in a total query time of $O(\log^2 n + k)$. Also, since we are only using $O(n)$ storage to store the associated structure of a given node, the total storage remains $O(n \log n)$.

The construction time becomes $O(n \log^2 n)$ because of the added sorting requirement at each node. According to the text, these total construction time can remain $O(n \log n)$ if we maintain a partial vertical ordering on the segments while constructing the tree. Because of this ordering, construction of the associated structures can be done in linear time.

References