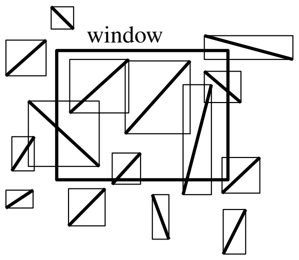
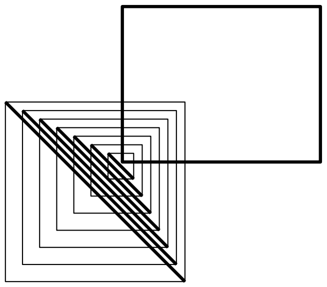
10.3 Segment Trees

So far we have studied the windowing problem for a set of axis-parallel line segments. We developed a nice data structure for this problem using interval trees with priority search trees as associated structure. The restriction to axis-parallel segments was inspired by the application to printed circuit board design. When we are doing windowing queries in roadmaps, however, we must drop this restriction: roadmaps contain line segments at arbitrary orientation.

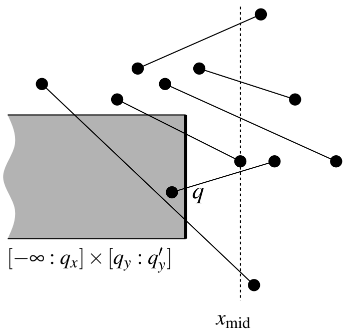


There is a trick that we can use to reduce the general problem to a problem on axis-parallel segments. We can replace each segment by its *bounding box*. Using the data structure for axis-parallel segments that we developed earlier, we can now find all the bounding boxes that intersect the query window W. We then check every segment whose bounding box intersects W to see if the segment itself also intersects W. In practice this technique usually works quite well: the majority of the segments whose bounding box intersects W will also intersect W themselves. In the worst case, however, the solution is quite bad: all bounding boxes may intersect W whereas none of the segments do. So if we want to guarantee a fast query time, we must look for another method.



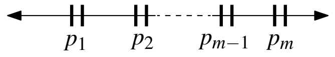
As before we make a distinction between segments that have an endpoint in the window and segments that intersect the window boundary. The first type of segments can be reported using a range tree. To find the answers of the second type we perform an intersection query with each of the four boundary edges of the window. (Of course care has to be taken that answers are reported only once.) We will only show how to perform queries with vertical boundary edges. For the horizontal edges a similar approach can be used. So we are given a set S of line segments with arbitrary orientations in the plane, and we want to find those segments in S that intersect a vertical query segment $q := q_x \times [q_y : q'_y]$. We will assume that the segments in S don't intersect each other, but we allow them to touch. (For intersecting segments the problem is a lot harder to solve and the time bounds are worse. Techniques like the ones described in Chapter 16 are required in this case.)

Let's first see if we can adapt the solution of the previous sections to the case of arbitrarily oriented segments. By searching with q_x in the interval tree we select a number of subsets $I_{\text{mid}}(\nu)$. For a selected node ν with $x_{\text{mid}}(\nu) > q_x$, the right endpoint of any segment in $I_{\text{mid}}(\nu)$ lies to the right of q. If the segment is horizontal, then it is intersected by the query segment if and only if its left endpoint lies in the range $(-\infty : q_x] \times [q_y : q'_y]$. If the segments have arbitrary orientation, however, things are not so simple: knowing that the right endpoint of a segment is to the right of q doesn't help us much. The interval tree is therefore not very useful in this case. Let's try to design a different data structure for the 1-dimensional problem, one that is more suited for dealing with arbitrarily oriented segments.



One of the paradigms for developing data structures is the *locus approach*. A query is described by a number of parameters; for the windowing problem, for example, there are four parameters, namely q_x, q'_x, q_y, and q'_y. For each choice of the parameters we get a certain answer. Often nearby choices give the same answer; if we move the window slightly it will often still intersect the same collection of segments. Let the *parameter space* be the space of all possible choices for the parameters. For the windowing problem this space is 4-dimensional. The locus approach suggests partitioning the parameter space into regions such that queries in the same region have the same answer. Hence, if we locate the region that contains the query then we know the answer to it. Such an approach only works well when the number of regions is small. For the windowing problem this is not true. There can be $\Theta(n^4)$ different regions. But we can use the locus approach to create an alternative for the interval tree.

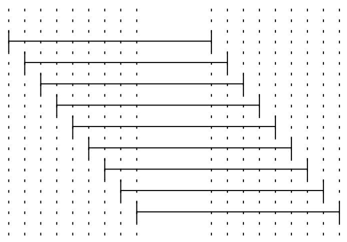
Let $I := \{[x_1 : x'_1], [x_2 : x'_2], \ldots, [x_n : x'_n]\}$ be a set of n intervals on the real line. The data structure that we are looking for should be able to report the intervals containing a query point q_x. Our query has only one parameter, q_x, so the parameter space is the real line. Let $p_1, p_2, \ldots, p_m$ be the list of distinct interval endpoints, sorted from left to right. The partitioning of the parameter space is simply the partitioning of the real line induced by the points p_i. We call the regions in this partitioning *elementary intervals*. Thus the elementary intervals are, from left to right,



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

The list of elementary intervals consists of open intervals between two consecutive endpoints p_i and p_{i+1}, alternated with closed intervals consisting of a single endpoint. The reason that we treat the points p_i themselves as intervals is, of course, that the answer to a query is not necessarily the same at the interior of an elementary interval and at its endpoints.

To find the intervals that contain a query point q_x, we must determine the elementary interval that contains q_x. To this end we build a binary search tree $\mathcal{T}$ whose leaves correspond to the elementary intervals. We denote the elementary interval corresponding to a leaf μ by $\mathrm{Int}(\mu)$.



If all the intervals in I containing $\mathrm{Int}(\mu)$ are stored at the leaf μ, then we can report the k intervals containing q_x in $O(\log n + k)$ time: first search in $O(\log n)$ time with q_x in $\mathcal{T}$, and then report all the intervals stored at μ in $O(1+k)$ time. So queries can be answered efficiently. But what about the storage requirement of the data structure? Intervals that span a lot of elementary intervals are stored at many leaves in the data structure. Hence, the amount of storage will be high if there are many pairs of overlapping intervals. If we have bad luck the amount of storage can even become quadratic. Let's see if we can do something to reduce the amount of storage. In Figure 10.8 you see an interval that spans five elementary intervals. Consider the elementary intervals corresponding to the leaves μ_1, μ_2, μ_3, and μ_4. When the search path to q_x ends in one of those leaves we must report the interval. The crucial observation is that a search path ends in μ_1, μ_2, μ_3, or μ_4 if and only if the path passes through the internal node ν. So why not store the interval at node ν (and at μ_5) instead of at the leaves μ_1, μ_2, μ_3, and μ_4 (and at μ_5)? In general, we store an interval at a number of nodes that together cover the interval, and we choose these nodes as high as possible.

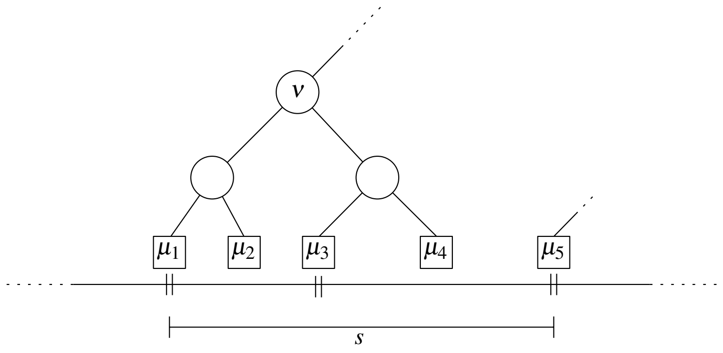


Figure 10.8
The segment s is stored at ν instead of at μ_1, μ_2, μ_3, and μ_4

The data structure based on this principle is called a *segment tree*. We now describe the segment tree for a set I of intervals more precisely. Figure 10.9 shows a segment tree for a set of five intervals.

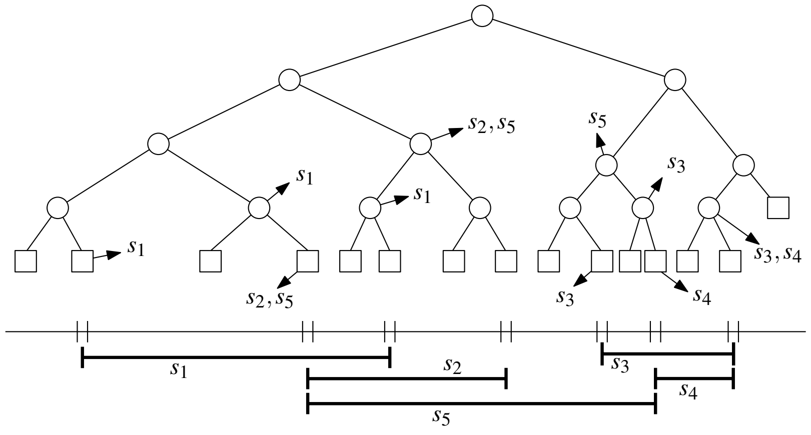


Figure 10.9
A segment tree: the arrows from the nodes point to their canonical subsets

- The skeleton of the segment tree is a balanced binary tree $\mathcal{T}$. The leaves of $\mathcal{T}$ correspond to the elementary intervals induced by the endpoints of the intervals in I in an ordered way: the leftmost leaf corresponds to the leftmost elementary interval, and so on. The elementary interval corresponding to leaf μ is denoted $\text{Int}(\mu)$.

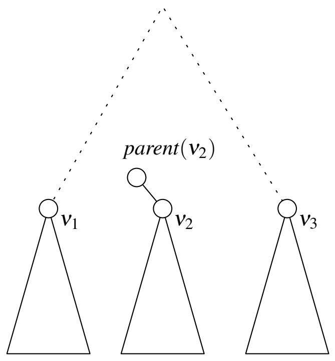
- The internal nodes of $\mathcal{T}$ correspond to intervals that are the union of elementary intervals: the interval $\text{Int}(\nu)$ corresponding to node ν is the union of the elementary intervals $\text{Int}(\mu)$ of the leaves in the subtree rooted at ν. (This implies that $\text{Int}(\nu)$ is the union of the intervals of its two children.)

- Each node or leaf ν in $\mathcal{T}$ stores the interval $\text{Int}(\nu)$ and a set $I(\nu) \subseteq I$ of intervals (for example, in a linked list). This *canonical subset* of node ν contains the intervals $[x : x'] \in I$ such that $\text{Int}(\nu) \subseteq [x : x']$ and $\text{Int}(\mathit{parent}(\nu)) \not\subseteq [x : x']$.

Let's see if our strategy of storing intervals as high as possible has helped to reduce the amount of storage.

Lemma 10.10 *A segment tree on a set of n intervals uses $O(n \log n)$ storage.*

Proof. Because $\mathcal{T}$ is a balanced binary search tree with at most $4n+1$ leaves, its height is $O(\log n)$. We claim that any interval $[x:x'] \in I$ is stored in the set $I(\nu)$ for at most two nodes at the same depth of the tree. To see why this is true, let ν_1, ν_2, ν_3 be three nodes at the same depth, numbered from left to right. Suppose $[x:x']$ is stored at ν_1 and ν_3. This means that $[x:x']$ spans the whole interval from the left endpoint of $\mathrm{Int}(\nu_1)$ to the right endpoint of $\mathrm{Int}(\nu_3)$. Because ν_2 lies between ν_1 and ν_3, $\mathrm{Int}(\mathit{parent}(\nu_2))$ must be contained in $[x:x']$. Hence, $[x:x']$ will not be stored at ν_2. It follows that any interval is stored at most twice at a given depth of the tree, so the total amount of storage is $O(n \log n)$. $\square$



So the strategy has helped: we have reduced the worst-case amount of storage from quadratic to $O(n \log n)$. But what about queries: can they still be answered easily? The answer is yes. The following simple algorithm describes how this is done. It is first called with $\nu = \mathit{root}(\mathcal{T})$.

```
Algorithm QuerySegmentTree($\nu, q_x$)
Input. The root of a (subtree of a) segment tree and a query point $q_x$.
Output. All intervals in the tree containing $q_x$.
1.  Report all the intervals in $I(\nu)$.
2.  if $\nu$ is not a leaf
3.     then if $q_x \in \mathrm{Int}(lc(\nu))$
4.             then QuerySegmentTree($lc(\nu), q_x$)
5.             else QuerySegmentTree($rc(\nu), q_x$)
```

The query algorithm visits one node per level of the tree, so $O(\log n)$ nodes in total. At a node ν we spend $O(1+k_\nu)$ time, where k_ν is the number of reported intervals. This leads to the following lemma.

Lemma 10.11 *Using a segment tree, the intervals containing a query point q_x can be reported in $O(\log n + k)$ time, where k is the number of reported intervals.*

To construct a segment tree we proceed as follows. First we sort the endpoints of the intervals in I in $O(n \log n)$ time. This gives us the elementary intervals. We then construct a balanced binary tree on the elementary intervals, and we determine for each node ν of the tree the interval $\mathrm{Int}(\nu)$ it represents. This can be done bottom-up in linear time. It remains to compute the canonical subsets for the nodes. To this end we insert the intervals one by one into the segment tree. An interval is inserted into $\mathcal{T}$ by calling the following procedure with $\nu = \mathit{root}(\mathcal{T})$.

```
Algorithm InsertSegmentTree($\nu, [x:x']$)
Input. The root of a (subtree of a) segment tree and an interval.
Output. The interval will be stored in the subtree.
1.  if $\mathrm{Int}(\nu) \subseteq [x:x']$
2.     then store $[x:x']$ at $\nu$
3.     else if $\mathrm{Int}(lc(\nu)) \cap [x:x'] \neq \emptyset$
4.             then InsertSegmentTree($lc(\nu), [x:x']$)
```

```
5.            if Int(rc(ν)) ∩ [x : x′] ≠ ∅
6.              then InsertSegmentTree(rc(ν), [x : x′])
```

How much time does it take to insert an interval $[x:x']$ into the segment tree? At every node that we visit we spend constant time (assuming we store $I(\nu)$ in a simple structure like a linked list). When we visit a node ν, we either store $[x:x']$ at ν, or $\text{Int}(\nu)$ contains an endpoint of $[x:x']$. We have already seen that an interval is stored at most twice at each level of $\mathcal{T}$. There is also at most one node at every level whose corresponding interval contains x and one node whose interval contains x'. So we visit at most 4 nodes per level. Hence, the time to insert a single interval is $O(\log n)$, and the total time to construct the segment tree is $O(n \log n)$.

The performance of segment trees is summarized in the following theorem.

Theorem 10.12 *A segment tree for a set I of n intervals uses $O(n\log n)$ storage and can be built in $O(n\log n)$ time. Using the segment tree we can report all intervals that contain a query point in $O(\log n + k)$ time, where k is the number of reported intervals.*

Recall that an interval tree uses only linear storage, and that it also allows us to report the intervals containing a query point in $O(\log n + k)$ time. So for this task an interval tree is to be preferred to a segment tree. When we want to answer more complicated queries, such as windowing in a set of line segments, then a segment tree is a more powerful structure to work with. The reason is that the set of intervals containing q_x is *exactly* the union of the canonical subsets that we select when we search in the segment tree. In an interval tree, on the other hand, we also select $O(\log n)$ nodes during a query, but not all intervals stored at those nodes contain the query point. We still have to walk along the sorted list to find the intersected intervals. So for segment trees, we have the possibility of storing the canonical subsets in an associated structure that allows for further querying.

Let's go back to the windowing problem. Let S be a set of arbitrarily oriented, disjoint segments in the plane. We want to report the segments intersecting a vertical query segment $q := q_x \times [q_y : q'_y]$. Let's see what happens when we build a segment tree $\mathcal{T}$ on the x-intervals of the segments in S. A node ν in $\mathcal{T}$ can now be considered to correspond to the vertical slab $\text{Int}(\nu) \times (-\infty : +\infty)$. A segment is in the canonical subset of ν if it completely crosses the slab corresponding to ν—we say that the segment *spans* the slab—but not the slab corresponding to the parent of ν. We denote these subsets with $S(\nu)$. See Figure 10.10 for an illustration.

When we search with q_x in $\mathcal{T}$ we find $O(\log n)$ canonical subsets—those of the nodes on the search path—that collectively contain all the segments whose x-interval contains q_x. A segment s in such a canonical subset is intersected by q if and only if the lower endpoint of q is below s and the upper endpoint of q is above s. How do we find the segments between the endpoints of q? Here we use the fact that the segments in the canonical subset $S(\nu)$ span the

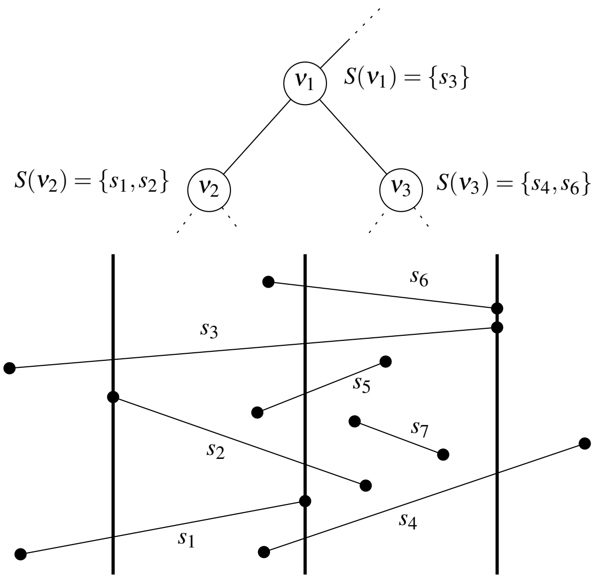
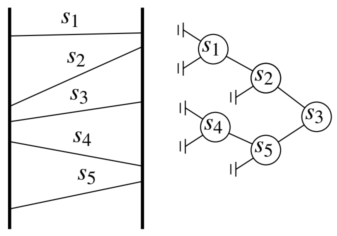


Figure 10.10
Canonical subsets contain segments that span the slab of a node, but not the slab of its parent

slab corresponding to v and that they do not intersect each other. This implies that the segments can be ordered vertically. Hence, we can store $S(v)$ in a search tree $\mathcal{T}(v)$ according to the vertical order. By searching in $\mathcal{T}(v)$ we can find the intersected segments in $O(\log n + k_v)$ time, where k_v is the number of intersected segments. The total data structure for the set S is thus as follows.



- The set S is stored in a segment tree $\mathcal{T}$ based on the x-intervals of the segments.
- The canonical subset of a node v in $\mathcal{T}$, which contains the segments spanning the slab corresponding to v but not the slab corresponding to the parent of v, is stored in a binary search tree $\mathcal{T}(v)$ based on the vertical order of the segments within the slab.

Because the associated structure of any node v uses storage linear in the size of $S(v)$, the total amount of storage remains $O(n \log n)$. The associated structures can be built in $O(n \log n)$ time, leading to a preprocessing time of $O(n \log^2 n)$. With a bit of extra work this can be improved to $O(n \log n)$. The idea is to maintain a (partial) vertical order on the segments while building the segment tree. With this order available the associated structures can be computed in linear time.

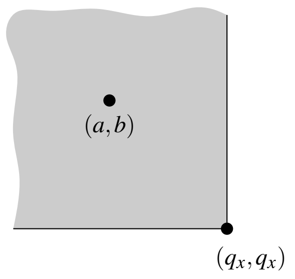
The query algorithm is quite simple: we search with q_x in the segment tree in the usual way, and at every node v on the search path we search with the upper and lower endpoint of q in $\mathcal{T}(v)$ to report the segments in $S(v)$ intersected by q. This basically is a 1-dimensional range query—see Section 5.1. The search in $\mathcal{T}(v)$ takes $O(\log n + k_v)$ time, where k_v is the number of reported segments at v. Hence, the total query time is $O(\log^2 n + k)$, and we obtain the following theorem.

Theorem 10.13 *Let S be a set of n disjoint segments in the plane. The segments intersecting a vertical query segment can be reported in $O(\log^2 n + k)$ time with a data structure that uses $O(n \log n)$ storage, where k is the number of reported segments. The structure can be built in $O(n \log n)$ time.*

Actually, it is only required that the segments have disjoint interiors. It is easily verified that the same approach can be used when the endpoints of segments are allowed to coincide with other endpoints or segments. This leads to the following result.

Corollary 10.14 *Let S be a set of n segments in the plane with disjoint interiors. The segments intersecting an axis-parallel rectangular query window can be reported in $O(\log^2 n + k)$ time with a data structure that uses $O(n \log n)$ storage, where k is the number of reported segments. The structure can be built in $O(n \log n)$ time.*

10.4 Notes and Comments



The query that asks for all intervals that contain a given point is often referred to as a *stabbing query*. The interval tree structure for stabbing queries is due to Edelsbrunner [157] and McCreight [270]. The priority search tree was designed by McCreight [271]. He observed that the priority search tree can be used for stabbing queries as well. The transformation is simple: map each interval $[a : b]$ to the point (a,b) in the plane. Performing a stabbing query with a value q_x can be done by doing a query with the range $(-\infty : q_x] \times [q_x : +\infty)$. Ranges of this type are a special case of the ones supported by priority search trees.

The segment tree was discovered by Bentley [45]. Used as a 1-dimensional data structure for stabbing queries it is less efficient than the interval tree since it requires $O(n \log n)$ storage. The importance of the segment tree is mainly that the sets of intervals stored with the nodes can be structured in any manner convenient for the problem at hand. Therefore, there are many extensions of segment trees that deal with 2- and higher-dimensional objects [103, 157, 163, 301, 375]. A second plus of the segment tree over the interval tree is that the segment tree can easily be adapted to stabbing *counting* queries: report the number of intervals containing the query point. Instead of storing the intervals in a list with the nodes, we store an integer representing their number. A query with a point is answered by adding the integers on one search path. Such a segment tree for stabbing counting queries uses only linear storage and queries require $O(\log n)$ time, so it is optimal.

There has been a lot of research in the past on the dynamization of interval trees and segment trees, that is, making it possible to insert and/or delete intervals. Priority search trees were initially described as a fully dynamic data structure, by replacing the binary tree with a red-black tree [199, 137] or other balanced binary search tree that requires only $O(1)$ rotations per update. Dynamization is important in situations where the input changes. Dynamic data structures are also important in many plane sweep algorithms where the status