Mark de Berg · Otfried Cheong
Marc van Kreveld · Mark Overmars

Computational Geometry

Algorithms and Applications

Third Edition

Springer

$O(\log n)$ canonical subsets. This means there are $O(\log^2 n)$ canonical subsets in the second-level structures in total. Together, they contain all points whose first and second coordinate lie in the correct ranges. The third-level structures storing these canonical subsets are then queried with the range for the third coordinate, and so on, until we reach the 1-dimensional trees. In these trees we find the points whose last coordinate lies in the correct range and report them. This approach leads to the following result.

Theorem 5.9 *Let P be a set of n points in d-dimensional space, where $d \geqslant 2$. A range tree for P uses $O(n\log^{d-1} n)$ storage and it can be constructed in $O(n\log^{d-1} n)$ time. One can report the points in P that lie in a rectangular query range in $O(\log^d n + k)$ time, where k is the number of reported points.*

Proof. Let $T_d(n)$ denote the construction time for a range tree on a set of n points in d-dimensional space. By Theorem 5.8 we know that $T_2(n) = O(n\log n)$. The construction of a d-dimensional range tree consists of building a balanced binary search tree, which takes time $O(n\log n)$, and the construction of associated structures. At the nodes at any depth of the first-level tree, each point is stored in exactly one associated structure. The time required to build all associated structures of the nodes at some depth is $O(T_{d-1}(n))$, the time required to build the associated structure of the root. This follows because the building time is at least linear. Hence, the total construction time satisfies

$$T_d(n) = O(n\log n) + O(\log n)\cdot T_{d-1}(n).$$

Since $T_2(n) = O(n\log n)$, this recurrence solves to $O(n\log^{d-1} n)$. The bound on the amount of storage follows in the same way.

Let $Q_d(n)$ denote the time spent in querying a d-dimensional range tree on n points, not counting the time to report points. Querying the d-dimensional range tree involves searching in a first-level tree, which takes time $O(\log n)$, and querying a logarithmic number of $(d-1)$-dimensional range trees. Hence,

$$Q_d(n) = O(\log n) + O(\log n)\cdot Q_{d-1}(n),$$

where $Q_2(n) = O(\log^2 n)$. This recurrence easily solves to $Q_d(n) = O(\log^d n)$. We still have to add the time needed to report points, which is bounded by $O(k)$. The bound on the query time follows. □

As in the 2-dimensional case, the query time can be improved by a logarithmic factor—see Section 5.6.

5.5 General Sets of Points

Until now we imposed the restriction that no two points have equal x- or y-coordinate, which is highly unrealistic. Fortunately, this is easy to remedy. The crucial observation is that we never assumed the coordinate values to be real numbers. We only need that they come from a totally ordered universe, so that

we can compare any two coordinates and compute medians. Therefore we can use the trick described next.

We replace the coordinates, which are real numbers, by elements of the so-called *composite-number space*. The elements of this space are pairs of reals. The *composite number* of two reals a and b is denoted by $(a|b)$. We define a total order on the composite-number space by using a lexicographic order. So, for two composite numbers $(a|b)$ and $(a'|b')$, we have

$$(a|b) < (a'|b') \quad \Leftrightarrow \quad a < a' \text{ or } (a = a' \text{ and } b < b').$$

Now assume we are given a set P of n points in the plane. The points are distinct, but many points can have the same x- or y-coordinate. We replace each point $p := (p_x, p_y)$ by a new point $\hat{p} := ((p_x|p_y),(p_y|p_x))$ that has composite numbers as coordinate values. This way we obtain a new set $\widehat{P}$ of n points. The first coordinate of any two points in $\widehat{P}$ are distinct; the same holds true for the second coordinate. Using the order defined above one can now construct kd-trees and 2-dimensional range trees for $\widehat{P}$.

Now suppose we want to report the points of P that lie in a range $R := [x : x'] \times [y : y']$. To this end we must query the tree we have constructed for $\widehat{P}$. This means that we must also transform the query range to our new composite space. The transformed range $\widehat{R}$ is defined as follows:

$$\widehat{R} := [(x|-\infty) : (x'|+\infty)] \times [(y|-\infty) : (y'|+\infty)].$$

It remains to prove that our approach is correct, that is, that the points of $\widehat{P}$ that we report when we query with $\widehat{R}$ correspond exactly to the points of P that lie in R.

Lemma 5.10 *Let p be a point and R a rectangular range. Then*

$$p \in R \quad \Leftrightarrow \quad \hat{p} \in \widehat{R}.$$

Proof. Let $R := [x : x'] \times [y : y']$ and let $p := (p_x, p_y)$. By definition, p lies in R if and only if $x \leqslant p_x \leqslant x'$ and $y \leqslant p_y \leqslant y'$. This is easily seen to hold if and only if $(x|-\infty) \leqslant (p_x|p_y) \leqslant (x'|+\infty)$ and $(y|-\infty) \leqslant (p_y|p_x) \leqslant (y'|+\infty)$, that is, if and only if $\hat{p}$ lies in $\widehat{R}$. □

We can conclude that our approach is indeed correct: we will get the correct answer to a query. Observe that there is no need to actually store the transformed points: we can just store the original points, provided that we do comparisons between two x-coordinates or two y-coordinates in the composite space.

The approach of using composite numbers can also be used in higher dimensions.

5.6* Fractional Cascading

In Section 5.3 we described a data structure for rectangular range queries in the plane, the range tree, whose query time is $O(\log^2 n + k)$. (Here n is the total

10 More Geometric Data Structures

Windowing

In the future most cars will be equipped with a vehicle navigation system to help you determine your position and to guide you to your destination. Such a system stores a roadmap of, say, the whole of the U.S. It also keeps track of where you are, so that it can show the appropriate part of the roadmap at any time on a little computer screen; this will usually be a rectangular region around your present position. Sometimes the system will do even more for you. For example, it might warn you when a turn is coming up that you should take to get to your destination.

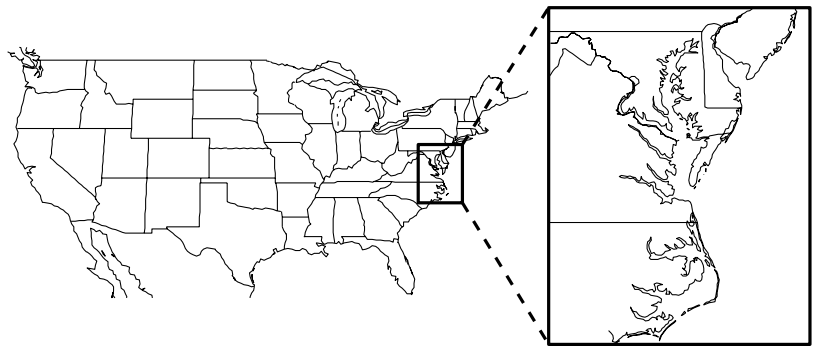
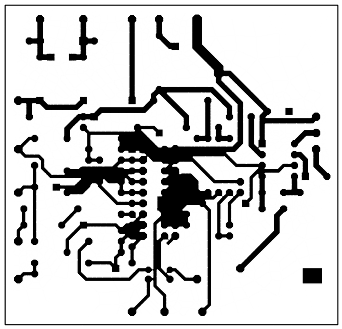


Figure 10.1
A windowing query in a map of the U.S.

To be of any use, the map should contain sufficient detail. A detailed map of the whole of Europe contains an enormous amount of data. Fortunately, only a small part of the map has to be displayed. Nevertheless, the system still has to find that part of the map: given a rectangular region, or a *window*, the system must determine the part of the map (roads, cities, and so on) that lie in the window, and display them. This is called a *windowing query*.

Checking every single feature of the map to see if it lies inside the window is not a workable method with the amount of data that we are dealing with. What we should do is to store the map in some kind of data structure that allows us to retrieve the part inside a window quickly.

Windowing queries are not only useful operations on geographic maps. They also play an important role in several applications in computer graphics and

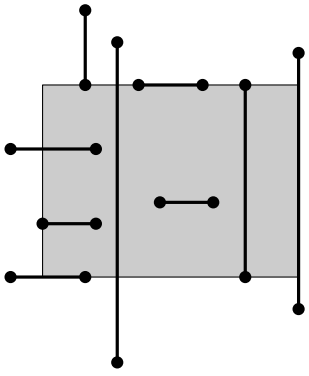


CAD/CAM. One example is flight simulation. A model of a landscape can consist of a huge number of triangles, but only a small part of the landscape will be within sight of the pilot. So we have to select the part of the landscape that lies within a given region. Here the region is 3-dimensional, and it is called the *viewing volume*. Another example comes from the design of printed circuit boards. Such a design typically consists of (a number of layers of) a planar drawing showing the location of traces and components. (See also Chapter 14.) In the design process one often wants to zoom in onto a certain portion of the board to inspect it more closely. Again, what we need is to determine the traces and components on the board that lie inside the window. In fact, windowing is required whenever one wants to inspect a small portion of a large, complex object.

Windowing queries are similar to the range queries studied in Chapter 5. The difference is the type of data that is dealt with: the data for range queries are points, whereas the data for windowing queries are typically line segments, polygons, curves, and so on. Also, for range queries we often deal with higher-dimensional search spaces, while for windowing queries the search space usually is 2- or 3-dimensional.

10.1 Interval Trees

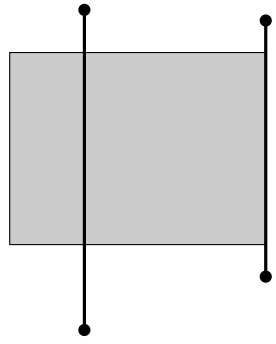
Let's start with the easiest of the examples that we gave above, namely windowing for a printed circuit board. The reason that this example is easier than the others is that the type of data is restricted: the objects on a printed circuit board normally have boundaries that consist of line segments with a limited number of possible orientations. Often they are parallel to one of the sides of the board or make 45 degree angles with the sides. Here we only consider the case were the segments are parallel to the sides. In other words, if we consider the x-axis to be aligned with the bottom side of the board, and the y-axis to be aligned with the left side of the board, then any segment is parallel to either the x-axis or the y-axis: the segments are *axis-parallel*, or *orthogonal*. We assume that the query window is an axis-parallel rectangle, that is, a rectangle whose edges are axis-parallel.



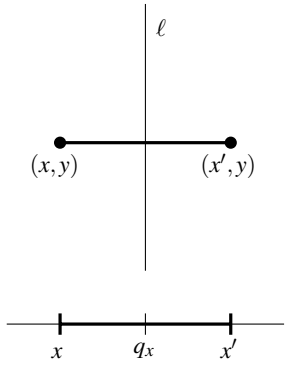
Let S be a set of n axis-parallel line segments. To solve windowing queries we need a data structure that stores S in such a way that the segments intersecting a query window $W := [x : x'] \times [y : y']$ can be reported efficiently. Let's first see in what ways a segment can intersect the window. There are a number of different cases: the segment can lie entirely inside W, it can intersect the boundary of W once, it can intersect the boundary twice, or it can (partially) overlap the boundary of W. In most cases the segment has at least one endpoint inside W. We can find such segments by performing a range query with W in the set of $2n$ endpoints of the segments in S. In Chapter 5 we have seen a data structure for this: the range tree. A 2-dimensional range tree uses $O(n \log n)$ storage, and a range query can be answered in $O(\log^2 n + k)$ time, where k is the number of

reported points. We have also shown that we can apply fractional cascading to reduce the query time to $O(\log n + k)$. There is one little problem. If we do a range query with W in the set of segment endpoints, we report the segments that have both endpoints inside W twice. This can be avoided by marking a segment when we report it for the first time, and only reporting segments that are not yet marked. Alternatively, when we find an endpoint of a segment to lie inside W, we can check whether the other endpoint lies inside W as well. If not we report the segment. If the other endpoint does lie inside W, we only report the segment when the current endpoint is the leftmost or bottom endpoint. This leads to the following lemma.

Lemma 10.1 *Let S be a set of n axis-parallel line segments in the plane. The segments that have at least one endpoint inside an axis-parallel query window W can be reported in $O(\log n + k)$ time with a data structure that uses $O(n \log n)$ storage and preprocessing time, where k is the number of reported segments.*



It remains to find the segments that do not have an endpoint inside the query window. Such segments either cross the boundary of W twice or contain one edge of the boundary. When the segment is vertical it will cross both horizontal edges of the boundary. When it is horizontal it will cross both vertical edges. Hence, we can find such segments by reporting all segments that intersect the left edge of the boundary and all segments that intersect the bottom edge of the boundary. (Note that there is no need to query with the other two edges of the boundary.) To be precise, we should only report those segments that do not have an endpoint inside W, because the others were already reported before. Let's consider the problem of finding the horizontal segments intersected by the left edge of W; to deal with the top edge we just have to reverse the roles of the x- and y-coordinates.



We have arrived at the following problem: preprocess a set S of horizontal line segments in the plane such that the segments intersecting a vertical query segment can be reported efficiently. To gain some insight into the problem we first look at a simpler version, namely where the query segment is a full line. Let $\ell := (x = q_x)$ denote the query line. A horizontal segment $s := \overline{(x,y)(x',y)}$ is intersected by ℓ if and only if $x \leqslant q_x \leqslant x'$. So only the x-coordinates of the segments play a role here. In other words, the problem becomes 1-dimensional: given a set of intervals on the real line, report the ones that contain the query point q_x.

Let $I := \{[x_1 : x'_1], [x_2 : x'_2], \ldots, [x_n : x'_n]\}$ be a set of closed intervals on the real line. To keep the connection with the 2-dimensional problem alive we image the real line to be horizontal, and we say "to the left (right) of" instead of "less (greater) than". Let x_{mid} be the median of the $2n$ interval endpoints, So at most half of the interval endpoints lies to the left of x_{mid} and at most half of the endpoints lies to the right of x_{mid}. If the query value q_x lies to the left of x_{mid} then the intervals that lie completely to the right of x_{mid} obviously do not contain q_x. We construct a binary tree based on this idea. The right subtree of the tree stores the set I_{right} of the intervals lying completely to the right of x_{mid}, and the left subtree stores the set I_{left} of intervals completely to the left of x_{mid}. These subtrees are constructed recursively in the same way. There is one problem that

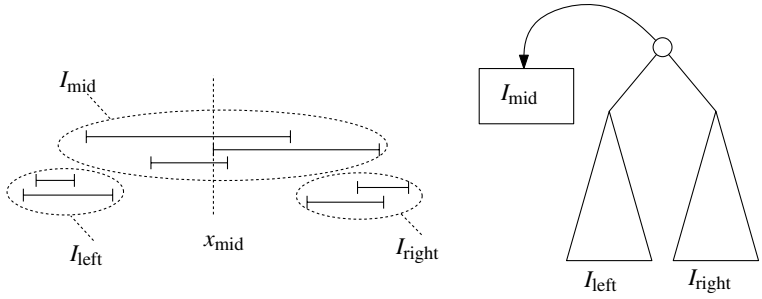
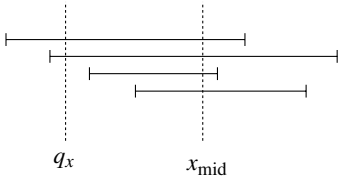


Figure 10.2
Classification of the segments with respect to x_{mid}

we still have to deal with: what to do with the intervals that contain x_{mid}? One possibility would be to store such intervals in both subtrees. For the children of the node, however, the same thing could happen again. Eventually one interval could be stored many times, and the amount of storage our data structure uses could become very large. To avoid the proliferation of intervals we deal with the problem differently: we store the set I_{mid} of intervals containing x_{mid} in a separate structure and associate that structure with the root of our tree. See Figure 10.2 for the situation. Note that in this figure (and others), although the intervals lie on a real line, we draw them on slightly different heights to distinguish them.

The associated structure should enable us to report the intervals in I_{mid} that contain q_x. So we ended up with the same problem that we started with: given a set I_{mid} of intervals, find those that contain q_x. But if we have bad luck I_{mid} could be the same as I. It seems we are back to exactly the same problem, but there is a difference. We know that all the intervals in I_{mid} contain x_{mid}, and this helps a great deal. Suppose, for example, that q_x lies to the left of x_{mid}. In that case we already know that the right endpoint of all intervals in I_{mid} lies right of q_x. So only the left endpoints of the intervals are important: q_x is contained in an interval $[x_j : x'_j] \in I_{\text{mid}}$ if and only if $x_j \leqslant q_x$. If we store the intervals in a list ordered on increasing left endpoint, then q_x can only be contained in an interval if q_x is also contained in all its predecessors in the sorted list. In other words, we can simply walk along the sorted list reporting intervals, until we come to an interval that does not contain q_x. At that point we can stop: none of the remaining intervals can contain q_x. Similarly, when q_x lies right of x_{mid} we can walk along a list that stores the intervals sorted on right endpoint. This list must be sorted on decreasing right endpoint, because it is traversed if the query point q_x lies to the right of x_{mid}. Finally, when $q_x = x_{\text{mid}}$ we can report all intervals in I_{mid}. (We do not need to treat this as a separate case. We can simply walk along one of the sorted lists.)



We now give a succinct description of the whole data structure that stores the intervals in I. The data structure is called an *interval tree*. Figure 10.3 shows an interval tree; the dotted vertical segments indicate the values x_{mid} for each node.

- If $I = \emptyset$ then the interval tree is a leaf.

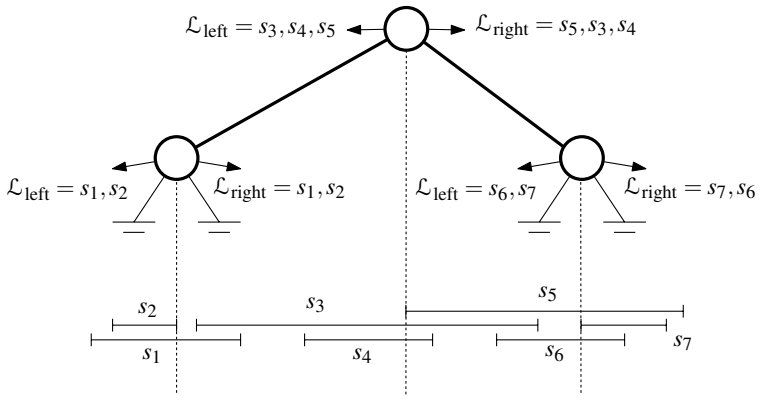


Figure 10.3
An interval tree

- Otherwise, let x_{mid} be the median of the endpoints of the intervals. Let

$$\begin{aligned} I_{\text{left}} &:= \{[x_j : x'_j] \in I : x'_j < x_{\text{mid}}\}, \\ I_{\text{mid}} &:= \{[x_j : x'_j] \in I : x_j \leqslant x_{\text{mid}} \leqslant x'_j\}, \\ I_{\text{right}} &:= \{[x_j : x'_j] \in I : x_{\text{mid}} < x_j\}. \end{aligned}$$

  The interval tree consists of a root node v storing x_{mid}. Furthermore,

  - the set I_{mid} is stored twice; once in a list $\mathcal{L}_{\text{left}}(v)$ that is sorted on the left endpoints of the intervals, and once in a list $\mathcal{L}_{\text{right}}(v)$ that is sorted on the right endpoints of the intervals,
  - the left subtree of v is an interval tree for the set I_{left},
  - the right subtree of v is an interval tree for the set I_{right}.

Lemma 10.2 *An interval tree on a set of n intervals uses $O(n)$ storage and has depth $O(\log n)$.*

Proof. The bound on the depth is trivial, so we prove the storage bound. Note that I_{left}, I_{mid}, and I_{right} are disjoint subsets. As a result, each interval is only stored in a set I_{mid} once and, hence, only appears once in the two sorted lists. This shows that the total amount of storage required for all associated lists is bounded by $O(n)$. The tree itself uses $O(n)$ storage as well. □

The following recursive algorithm for building an interval tree follows directly from its definition. (Recall that $lc(v)$ and $rc(v)$ denote the left and right child, respectively, of a node v.)

Algorithm CONSTRUCTINTERVALTREE(I)
Input. A set I of intervals on the real line.
Output. The root of an interval tree for I.
1. **if** $I = \emptyset$
2. **then return** an empty leaf
3. **else** Create a node v. Compute x_{mid}, the median of the set of interval endpoints, and store x_{mid} with v.

4. Compute I_{mid} and construct two sorted lists for I_{mid}: a list $\mathcal{L}_{\text{left}}(v)$ sorted on left endpoint and a list $\mathcal{L}_{\text{right}}(v)$ sorted on right endpoint. Store these two lists at v.
5. $lc(v) \leftarrow$ CONSTRUCTINTERVALTREE(I_{left})
6. $rc(v) \leftarrow$ CONSTRUCTINTERVALTREE(I_{right})
7. **return** v

Finding the median of a set of points takes linear time. Actually, it is better to compute the median by presorting the set of points, as in Chapter 5. It is easy to maintain these presorted sets through the recursive calls. Let $n_{\text{mid}} := \text{card}(I_{\text{mid}})$. Creating the lists $\mathcal{L}_{\text{left}}(v)$ and $\mathcal{L}_{\text{right}}(v)$ takes $O(n_{\text{mid}} \log n_{\text{mid}})$ time. Hence, the time we spend (not counting the time needed for the recursive calls) is $O(n + n_{\text{mid}} \log n_{\text{mid}})$. Using similar arguments as in the proof of Lemma 10.2 we can conclude that the algorithm runs in $O(n \log n)$ time.

Lemma 10.3 *An interval tree on a set of n intervals can be built in $O(n \log n)$ time.*

It remains to show how to use the interval tree to find the intervals containing a query point q_x. We already sketched how to do this, but now we can give the exact algorithm.

Algorithm QUERYINTERVALTREE(v, q_x)
Input. The root v of an interval tree and a query point q_x.
Output. All intervals that contain q_x.
1. **if** v is not a leaf
2. **then if** $q_x < x_{\text{mid}}(v)$
3. **then** Walk along the list $\mathcal{L}_{\text{left}}(v)$, starting at the interval with the leftmost endpoint, reporting all the intervals that contain q_x. Stop as soon as an interval does not contain q_x.
4. QUERYINTERVALTREE($lc(v), q_x$)
5. **else** Walk along the list $\mathcal{L}_{\text{right}}(v)$, starting at the interval with the rightmost endpoint, reporting all the intervals that contain q_x. Stop as soon as an interval does not contain q_x.
6. QUERYINTERVALTREE($rc(v), q_x$)

Analyzing the query time is not very difficult. At any node v that we visit we spend $O(1 + k_v)$ time, where k_v is the number of intervals that we report at v. The sum of the k_v's over the visited nodes is, of course, k. Furthermore, we visit at most one node at any depth of the tree. As noted above, the depth of the interval tree is $O(\log n)$. So the total query time is $O(\log n + k)$.

The following theorem summarizes the results about interval trees.

Theorem 10.4 *An interval tree for a set I of n intervals uses $O(n)$ storage and can be built in $O(n \log n)$ time. Using the interval 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.*

It's time to pause a moment and see where all this has brought us. The problem we originally wanted to solve is this: store a set S of axis-parallel segments in a data structure that allows us to find the segments intersecting a query window $W = [x : x'] \times [y : y']$. Finding the segments that have an endpoint inside W could be done using a data structure from Chapter 5, the range tree. The other segments intersecting W had to intersect the boundary of W twice. We planned to find these segments by querying with the left and top edges of W. So we needed a data structure that stores a set of horizontal segments such that the ones intersecting a vertical query segment can be reported efficiently, and a similar data structure storing the vertical segments that allows for intersection queries with horizontal segments. We started by developing a data structure that solves a slightly simpler problem, namely where the query object is a full line. This led to the interval tree. Now let's see how to extend the interval tree to the case where the query object is a vertical line segment.

Let $S_H \subseteq S$ be the subset of horizontal segments in S, and let q be the vertical query segment $q_x \times [q_y : q'_y]$. For a segment $s := [s_x : s'_x] \times s_y$ in S_H, we call $[s_x : s'_x]$ the *x-interval* of the segment. Suppose we have stored the segments in S_H in an interval tree $\mathcal{T}$ according to their x-intervals. Let's go through the query algorithm QUERYINTERVALTREE to see what happens when we query $\mathcal{T}$ with the vertical query segment q. Suppose q_x lies to the left of the x_{mid}-value stored at the root of the interval tree $\mathcal{T}$. It is still correct that we only search recursively in the left subtree: segments completely to the right of x_{mid} cannot be intersected by q so we can skip the right subtree. The way the set I_{mid} is treated, however, is not correct anymore. For a segment $s \in I_{\text{mid}}$ to be intersected by q, it is not sufficient that its left endpoint lies to the left of q; it is also required that its y-coordinate lies in the range $[q_y : q'_y]$. Figure 10.4 illustrates this. So

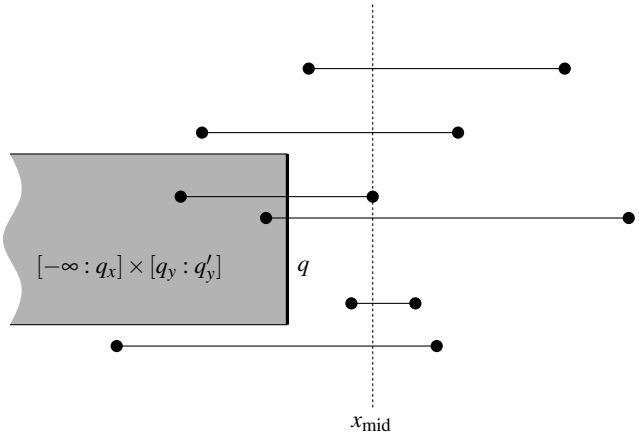
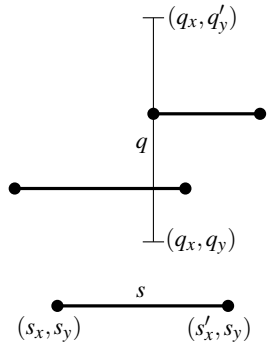


Figure 10.4
Segments intersected by q must have their left endpoint in the shaded region

storing the endpoints in an ordered list is not enough. We need a more elaborate associated structure: given a query range $(-\infty : q_x] \times [q_y : q'_y]$, we must be able to report all the segments whose left endpoint lies in that range. If q lies to the right of x_{mid} we want to report all segments whose right endpoint lies in the range $[q_x : +\infty) \times [q_y : q'_y]$, so for this case we need a second associated



structure. How should we implement the associated structure? Well, the query that we wish to perform is nothing more than a rectangular range query on a set of points. A 2-dimensional range tree, described in Chapter 5, will therefore do the trick. This way the associated structure uses $O(n_{\text{mid}} \log n_{\text{mid}})$ storage, where $n_{\text{mid}} := \text{card}(I_{\text{mid}})$, and its query time is $O(\log n_{\text{mid}} + k)$.

The data structure that stores the set S_H of horizontal segments is now as follows. The main structure is an interval tree $\mathcal{T}$ on the x-intervals of the segments. Instead of the sorted lists $\mathcal{L}_{\text{left}}(v)$ and $\mathcal{L}_{\text{right}}(v)$ we have two range trees: a range tree $\mathcal{T}_{\text{left}}(v)$ on the left endpoints of the segments in $I_{\text{mid}}(v)$, and a range tree $\mathcal{T}_{\text{right}}(v)$ on the right endpoints of the segments in $I_{\text{mid}}(v)$. Because the storage required for a range tree is a factor $\log n$ larger than for sorted lists, the total amount of storage for the data structure becomes $O(n \log n)$. The preprocessing time remains $O(n \log n)$.

The query algorithm is the same as QUERYINTERVALTREE, except that, instead of walking along the sorted list $\mathcal{L}_{\text{left}}(v)$, say, we perform a query in the range tree $\mathcal{T}_{\text{left}}(v)$. So at each of the $O(\log n)$ nodes v on the search path we spend $O(\log n + k_v)$ time, where k_v is the number of reported segments. The total query time therefore becomes $O(\log^2 n + k)$.

We have proved the following theorem.

Theorem 10.5 *Let S be a set of n horizontal 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.*

If we combine this with the result of Lemma 10.1 we get a solution to the windowing problem for axis-parallel segments.

Corollary 10.6 *Let S be a set of n axis-parallel segments in the plane. 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.*

10.2 Priority Search Trees

In the structure for windowing described in Section 10.1 we used a range tree for the associated structures. The range queries we perform on them have a special property: they are unbounded on one side. In this section we will describe a different data structure, the *priority search tree*, that uses this property to improve the bound on storage to $O(n)$. This data structure is also a lot simpler because it does not require fractional cascading. Using priority search trees instead of range trees in the data structure for windowing reduces the storage bound in Theorem 10.5 to $O(n)$. It does not improve the storage bound in

Corollary 10.6 because there we also need a range tree to report the endpoints that lie in the window.

Let $P := \{p_1, p_2, \ldots, p_n\}$ be a set of points in the plane. We want to design a structure for rectangular range queries of the form $(-\infty : q_x] \times [q_y : q'_y]$. To get some idea of how this special property can be used, let's look at the 1-dimensional case. A normal 1-dimensional range query would ask for the points lying in a range $[q'_x : q_x]$. To find these points efficiently we can store the set of points in a 1-dimensional range tree, as described in Chapter 5. If the range is unbounded to the left, we are asking for the points lying in $(-\infty : q_x]$. This can be solved by simply walking along an ordered list starting at the leftmost point, until we encounter a point that is not in the range. The query time is $O(1+k)$, instead of $O(\log n + k)$ which we needed in the general case.

How can we extend this strategy to 2-dimensional range queries that are unbounded to the left? Somehow we must integrate information about the y-coordinate in the structure without using associated structures, so that, among the points whose x-coordinate is in $(-\infty : q_x]$, we can easily select the ones whose y-coordinate is in $[q_y : q'_y]$. A simple linear list doesn't lend itself well for this. Therefore we take a different structure to work with: the *heap*.

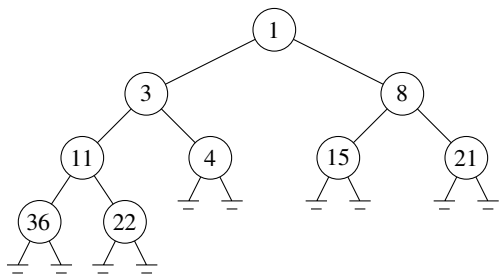
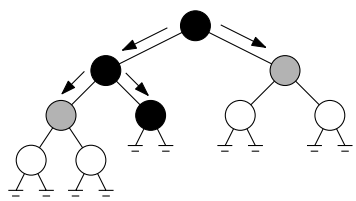


Figure 10.5
A heap for the set
$\{1,3,4,8,11,15,21,22,36\}$

A heap is normally used for priority queries that ask for the smallest (or largest) value in a set. But heaps can also be used to answer 1-dimensional range queries of the form $(-\infty : q_x]$. A heap has the same query time as a sorted list, namely $O(1+k)$. Normally the advantage of a heap over a sorted list is that points can be inserted and the maximum deleted more efficiently. For us the tree structure of a heap has another advantage: it makes it easier to integrate information about the y-coordinate, as we shall see shortly. A heap is a binary tree defined as follows. The root of the tree stores the point with minimum x-value. The remainder of the set is partitioned into two subsets of almost equal size, and these subsets are stored recursively in the same way. Figure 10.5 gives an example of a heap. We can do a query with $(-\infty : q_x]$ by walking down the tree. When we visit a node we check if the x-coordinate of the point stored at the node lies in $(-\infty : q_x]$. If it does, we report the point and continue the search in both subtrees; otherwise, we abort the search in this part of the tree. For example, when we search with $(-\infty : 5]$ in the tree of Figure 10.5, we report the points 1, 3, and 4. We also visit the nodes with 8 and 11 but abort the search there.



Heaps give us some freedom in how to partition the set into two subsets. If we also want to search on y-coordinate then the trick is to perform the partitioning not in an arbitrary way, as is the case for normal heaps, but according to y-coordinate. More precisely, we split the remainder of the set into two subsets of almost equal size such that the y-coordinate of any point in one subset is smaller than the y-coordinate of any point in the other subset. This is illustrated in Figure 10.6. The tree is drawn sideways to indicate that the partitioning

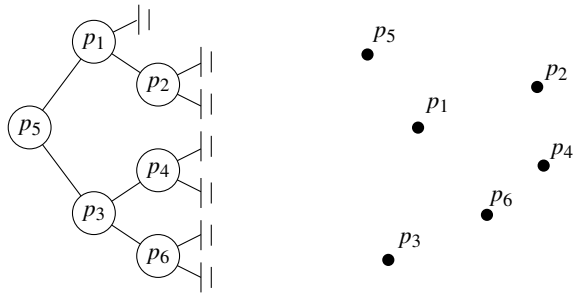


Figure 10.6
A set of points and the corresponding priority search tree

is on y-coordinate. In the example of Figure 10.6 the point p_5 has smallest x-coordinate and, hence, is stored in the root. The other points are partitioned in y-coordinate. The points p_3, p_4, and p_6 have smaller y-coordinate and are stored in the left subtree. Of these p_3 has smallest x-coordinate, so this point is placed in the root of the subtree, and so on.

A formal definition of a priority search tree for a set P of points is as follows. We assume that all the points have distinct coordinates. In Chapter 5 (more precisely, in Section 5.5) we saw that this involves no loss of generality; by using composite numbers we can simulate that all coordinates are distinct.

- If $P = \emptyset$ then the priority search tree is an empty leaf.
- Otherwise, let p_{min} be the point in the set P with the smallest x-coordinate. Let y_{mid} be the median of the y-coordinates of the remaining points. Let

$$\begin{aligned} P_{\text{below}} &:= \{p \in P \setminus \{p_{\text{min}}\} : p_y < y_{\text{mid}}\}, \\ P_{\text{above}} &:= \{p \in P \setminus \{p_{\text{min}}\} : p_y > y_{\text{mid}}\}. \end{aligned}$$

  The priority search tree consists of a root node v where the point $p(v) := p_{\text{min}}$ and the value $y(v) := y_{\text{mid}}$ are stored. Furthermore,
  - the left subtree of v is a priority search tree for the set P_{below},
  - the right subtree of v is a priority search tree for the set P_{above}.

It's straightforward to derive a recursive $O(n \log n)$ algorithm for building a priority search tree. Interestingly, priority search trees can even be built in linear time, if the points are already sorted on y-coordinate. The idea is to construct the tree bottom-up instead of top-down, in the same way heaps are normally constructed.

A query with a range $(-\infty : q_x] \times [q_y : q'_y]$ in a priority search tree is performed roughly as follows. We search with q_y and q'_y, as indicated in Figure 10.7. All

the shaded subtrees in the figure store only points whose y-coordinate lies in the correct range. So we can search those subtrees based on x-coordinate only. This is done with the following subroutine, which is basically the query algorithm for a heap.

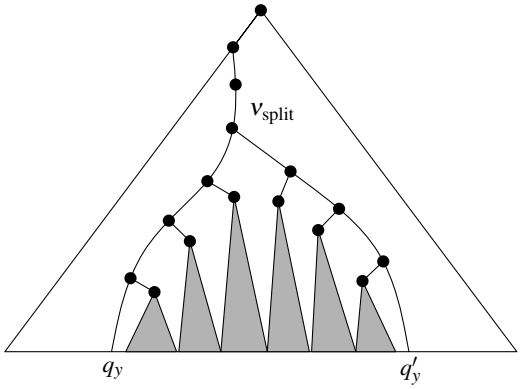


Figure 10.7
Querying a priority search tree

REPORTINSUBTREE(v, q_x)
Input. The root v of a subtree of a priority search tree and a value q_x.
Output. All points in the subtree with x-coordinate at most q_x.
1. **if** v is not a leaf and $(p(v))_x \leqslant q_x$
2. **then** Report $p(v)$.
3. REPORTINSUBTREE$(lc(v), q_x)$
4. REPORTINSUBTREE$(rc(v), q_x)$

Lemma 10.7 *REPORTINSUBTREE(v, q_x) reports in $O(1+k_v)$ time all points in the subtree rooted at v whose x-coordinate is at most q_x, where k_v is the number of reported points.*

Proof. Let $p(\mu)$ be a point with $(p(\mu))_x \leqslant q_x$ that is stored at a node μ in the subtree rooted at v. By definition of the data structure, the x-coordinates of the points stored at the path from μ to v form a decreasing sequence, so all these points must have x-coordinates at most q_x. Hence, the search is not aborted at any of these nodes, which implies that μ is reached and $p(\mu)$ is reported. We conclude that all points with x-coordinate at most q_x are reported. Obviously, those are the only reported points.

At any node μ that we visit we spend $O(1)$ time. When we visit a node μ with $\mu \neq v$ we must have reported a point at the parent of μ. We charge the time we spend at μ to this point. This way any reported point gets charged twice, which means that the time we spend at nodes μ with $\mu \neq v$ is $O(k_v)$. Adding the time we spend at v we get a total of $O(1+k_v)$ time. □

If we call REPORTINSUBTREE at each of the subtrees that we select (the shaded subtrees of Figure 10.7), do we find all the points that lie in the query range? The answer is no. The root of the tree, for example, stores the point with smallest x-coordinate. This may very well be a point in the query range. In fact, any

point stored at a node on the search path to q_y or q'_y may lie in the query range, so we should test them as well. This leads to the following query algorithm.

Algorithm QUERYPRIOSEARCHTREE($\mathcal{T}, (-\infty : q_x] \times [q_y : q'_y]$)
Input. A priority search tree and a range, unbounded to the left.
Output. All points lying in the range.

```
1.  Search with q_y and q'_y in T. Let v_split be the node where the two search
    paths split.
2.  for each node v on the search path of q_y or q'_y
3.      do if p(v) ∈ (−∞ : q_x] × [q_y : q'_y] then report p(v).
4.   for each node v on the path of q_y in the left subtree of v_split
5.      do if the search path goes left at v
6.            then REPORTINSUBTREE(rc(v), q_x)
7.   for each node v on the path of q'_y in the right subtree of v_split
8.      do if the search path goes right at v
9.            then REPORTINSUBTREE(lc(v), q_x)
```

Lemma 10.8 *Algorithm* QUERYPRIOSEARCHTREE *reports the points in a query range* $(-\infty : q_x] \times [q_y : q'_y]$ *in* $O(\log n + k)$ *time, where* k *is the number of reported points.*

Proof. First we prove that any point that is reported by the algorithm lies in the query range. For the points on the search paths to q_y and q'_y this is obvious: these points are tested explicitly for containment in the range. Consider a call REPORTINSUBTREE($rc(v), q_x$) in line 6. Let p be a point that is reported in this call. By Lemma 10.7 we have $p_x \leqslant q_x$. Furthermore, $p_y \leqslant q'_y$, because all the nodes visited in this call lie to the left of v_{split} and $q'_y > y(v_{\text{split}})$. Finally, $p_y \geqslant q_y$, because all the nodes visited in this call lie to the right of v and the search path to q_y went left at v. A similar argument applies for the points reported in line 9.

We have proved that all the reported points lie in the query range. Conversely, let $p(\mu)$ be a point in the range. Any point stored to the left of a node where the search path to q_y goes right must have a y-coordinate smaller than q_y. Similarly, any point stored to the right of a node where the search path to q'_y goes left must have a y-coordinate greater than q'_y. Hence, μ must either be on one of the search paths, or in one of the subtrees for which REPORTINSUBTREE is called. In both cases $p(\mu)$ will be reported.

It remains to analyze the time taken by the algorithm. This is linear in the number of nodes on the search paths to q_y and q'_y plus the time taken by all the executions of the procedure REPORTINSUBTREE. The depth of the tree and, hence, the number of nodes on the search paths, is $O(\log n)$. The time taken by all executions of REPORTINSUBTREE is $O(\log n + k)$ by Lemma 10.7. □

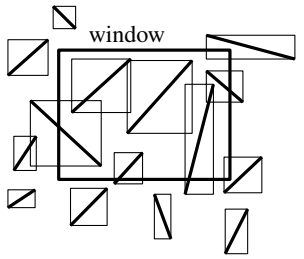
The performance of priority search trees is summarized in the following theorem.

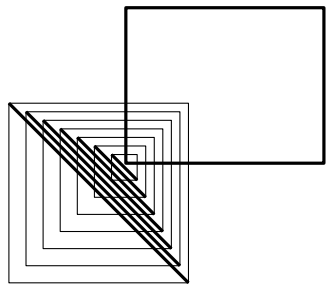
Theorem 10.9 *A priority search tree for a set* P *of* n *points in the plane uses* $O(n)$ *storage and can be built in* $O(n \log n)$ *time. Using the priority search tree we can report all points in a query range of the form* $(-\infty : q_x] \times [q_y : q'_y]$ *in* $O(\log n + k)$ *time, where* k *is the number of reported points.*

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.

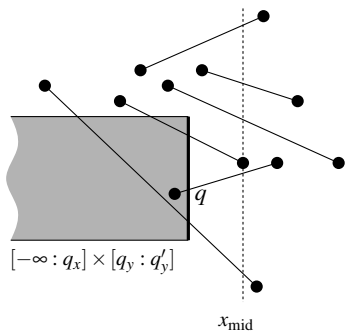
There is a trick that we can use to reduce the general problem to a problem on axis-parallel segments. We can replace each segment by its *bounding box*. Using the data structure for axis-parallel segments that we developed earlier, we can now find all the bounding boxes that intersect the query window 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.)

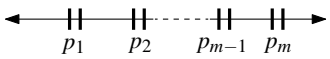
Let's first see if we can adapt the solution of the previous sections to the case of arbitrarily oriented segments. By searching with q_x in the interval tree we select a number of subsets $I_{\text{mid}}(v)$. For a selected node v with $x_{\text{mid}}(v) > q_x$, the right endpoint of any segment in $I_{\text{mid}}(v)$ 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.

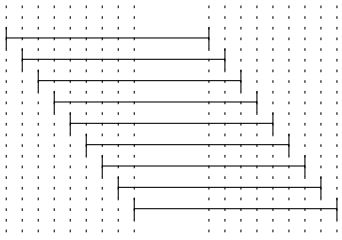
Let $I := \{[x_1 : x_1'], [x_2 : x_2'], \ldots, [x_n : x_n']\}$ be a set of n intervals on the real line. The data structure that we are looking for should be able to report the intervals containing a query point q_x. Our query has only one parameter, q_x, so the parameter space is the real line. Let $p_1, p_2, \ldots, p_m$ 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 v. So why not store the interval at node v (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.

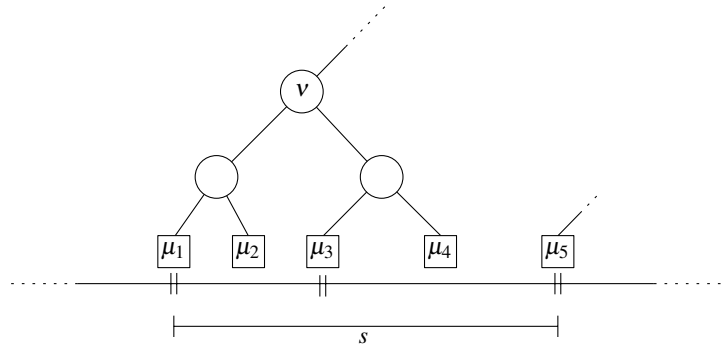


Figure 10.8
The segment s is stored at v instead of at μ_1, μ_2, μ_3, and μ_4

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.

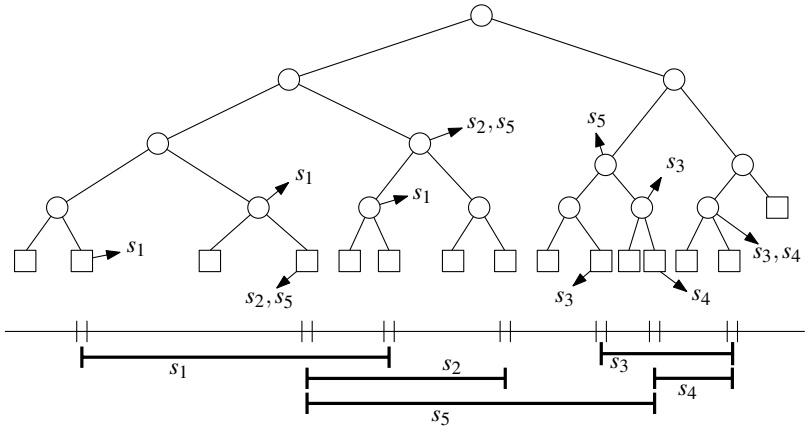


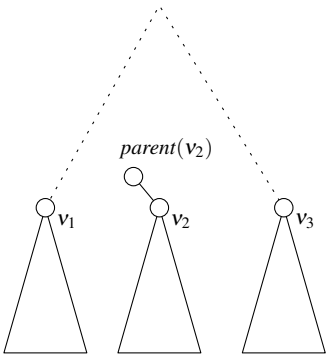
Figure 10.9
A segment tree: the arrows from the nodes point to their canonical subsets

- 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 $\text{Int}(v)$ corresponding to node v is the union of the elementary intervals $\text{Int}(\mu)$ of the leaves in the subtree rooted at v. (This implies that $\text{Int}(v)$ is the union of the intervals of its two children.)

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

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 $I(v)$ for at most two nodes at the same depth of the tree. To see why this is true, let v_1, v_2, v_3 be three nodes at the same depth, numbered from left to right. Suppose $[x:x']$ is stored at v_1 and v_3. This means that $[x:x']$ spans the whole interval from the left endpoint of $\text{Int}(v_1)$ to the right endpoint of $\text{Int}(v_3)$. Because v_2 lies between v_1 and v_3, $\text{Int}(parent(v_2))$ must be contained in $[x:x']$. Hence, $[x:x']$ will not be stored at v_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 $O(n\log n)$. □

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 $v = root(\mathcal{T})$.

Algorithm QUERYSEGMENTTREE(v, 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 $I(v)$.
2. **if** v is not a leaf
3. **then if** $q_x \in \text{Int}(lc(v))$
4. **then** QUERYSEGMENTTREE($lc(v), q_x$)
5. **else** QUERYSEGMENTTREE($rc(v), q_x$)

The query algorithm visits one node per level of the tree, so $O(\log n)$ nodes in total. At a node v we spend $O(1+k_v)$ time, where k_v 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 v of the tree the interval $\text{Int}(v)$ 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 $v = root(\mathcal{T})$.

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

```
5.      if Int(rc(v)) ∩ [x : x'] ≠ ∅
6.        then INSERTSEGMENTTREE(rc(v), [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 $I(v)$ in a simple structure like a linked list). When we visit a node v, we either store $[x : x']$ at v, or $\text{Int}(v)$ 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 v in $\mathcal{T}$ can now be considered to correspond to the vertical slab $\text{Int}(v) \times (-\infty : +\infty)$. A segment is in the canonical subset of v if it completely crosses the slab corresponding to v—we say that the segment *spans* the slab—but not the slab corresponding to the parent of v. We denote these subsets with $S(v)$. 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 $S(v)$ span the

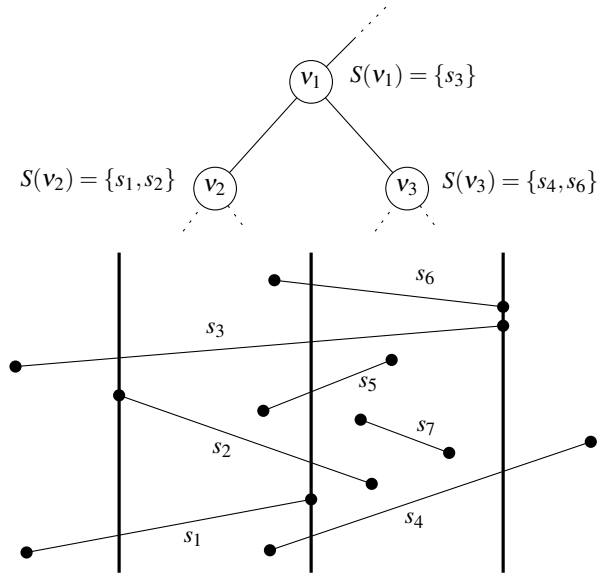
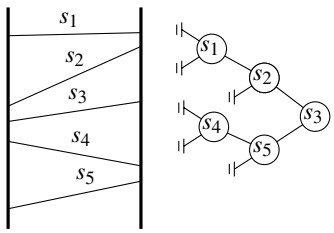


Figure 10.10
Canonical subsets contain segments that span the slab of a node, but not the slab of its parent

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.

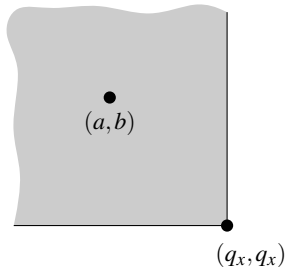
The query algorithm is quite simple: we search with q_x in the segment tree in the usual way, and at every node v on the search path we search with the upper and lower endpoint of q in $\mathcal{T}(v)$ to report the segments in $S(v)$ intersected by q. This basically is a 1-dimensional range query—see 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

normally needs to be stored in a dynamic structure. In a number of problems a dynamic version of the segment tree is required here.

The notion of *decomposable searching problems* [46, 48, 166, 254, 269, 276, 304, 306, 307, 308, 337] gave a great push toward the dynamization of a large class of data structures. Let S be a set of objects involved in a searching problem, and let $A \cup B$ be any partition of S. The searching problem is *decomposable* if the solution to the searching problem on S can be obtained in constant time from the solutions to the searching problems on A and on B. The stabbing query problem is decomposable because if we have reported the stabbed intervals in the two subsets of a partition, we have automatically reported all intervals in the whole set. Similarly, the stabbing counting query problem is decomposable because we can add the integer answers yielded by the subsets to give the answer for the whole set. Many other searching problems, like the range searching problem, are decomposable as well. Static data structures for decomposable searching problems can be turned into dynamic structures using general techniques. For an overview see the book by Overmars [299].

The stabbing query problem can be generalized to higher dimensions. Here we are given a collection of axis-parallel (hyper-)rectangles and ask for those rectangles that contain a given query point. To solve these higher-dimensional stabbing queries we can use multi-level segment trees. The structure uses $O(n \log^{d-1} n)$ storage and stabbing queries can be answered in $O(\log^d n)$ time. The use of fractional cascading—see Chapter 5—lowers the query time bound by a logarithmic factor. The use of the interval tree on the deepest level of associated structures lowers the storage bound with a logarithmic factor. A higher-dimensional version of the interval tree and priority search tree doesn't exist, that is, there is no clear extension of the structure that solves the analogous query problem in higher dimensions. But the structures can be used as associated structure of segment trees and range trees. This is useful, for instance, to solve the stabbing query problem in sets of axis-parallel rectangles, and for range searching with ranges of the form $[x : x'] \times [y : y'] \times [z : +\infty)$.

In geographic information systems, the most widely used geometric data structure is the *R-tree* [204]. It is a 2-dimensional extension of the well-known B-tree, suitable for storage on disk. On disk, memory is partitioned in blocks of some size B, and the idea is to have nodes of high degree that fit exactly into one block. An internal node gives rise to a multi-way split instead of just a binary split, allowing the tree to be much less deep. Queries that follow only one path down the tree require fewer blocks from disk, and are therefore answered more efficiently than with a binary tree. An R-tree can store any set of objects (points, line segments, polygons), and answer intersection queries with any query object. An R-tree uses linear storage, but the worst-case query time is also linear, making its performance nothing better than a simple linked list, in theory. In practice, however, R-trees perform well. A few theoretical results on R-trees are known. It was shown in [5] that any type of R-tree built on n points in d-dimensional space has to visit $\Omega((n/B)^{1-1/d} + k/B)$ blocks (where k is the number of answers). A variation of the R-tree, called the PR-tree [18], attains this bound if both the objects stored and the query objects are axis-parallel hyperrectangles. For an extensive overview of R-trees and related structures, see [335].