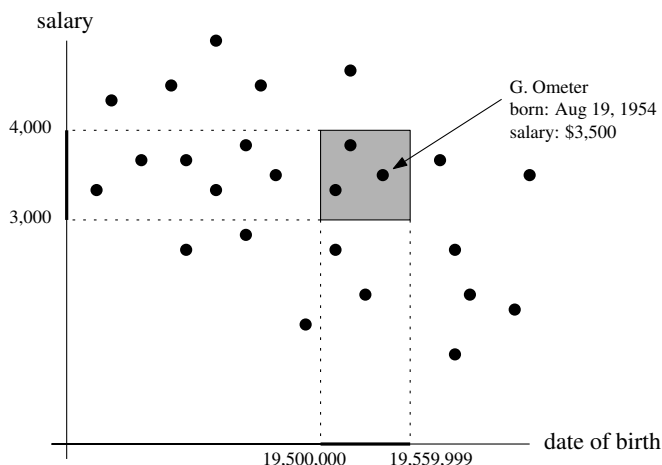# 5 Orthogonal Range Searching

Querying a Database

At first sight it seems that databases have little to do with geometry. Nevertheless, many types of questions—from now on called *queries*—about data in a database can be interpreted geometrically. To this end we transform records in a database into points in a multi-dimensional space, and we transform the queries about the records into queries on this set of points. Let's demonstrate this with an example.

G. Ometer
born: Aug 19, 1954
salary: $3,500

salary

4,000

3,000

19,500,000   19,559,999   date of birth

Figure 5.1
Interpreting a database query geometrically

Consider a database for personnel administration. In such a database the name, address, date of birth, salary, and so on, of each employee are stored. A typical query one may want to perform is to report all employees born between 1950 and 1955 who earn between $3,000 and $4,000 a month. To formulate this as a geometric problem we represent each employee by a point in the plane. The first coordinate of the point is the date of birth, represented by the integer $10,000 \times year + 100 \times month + day$, and the second coordinate is the monthly salary. With the point we also store the other information we have about the employee, such as name and address. The database query asking for all employees born between 1950 and 1955 who earn between $3,000 and
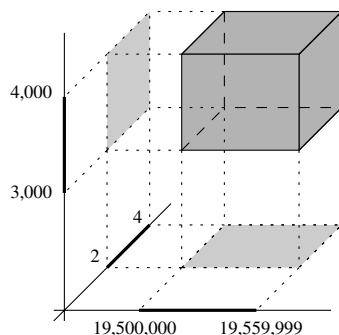
$4,000$ transforms into the following geometric query: report all points whose first coordinate lies between 19,500,000 and 19,559,999, and whose second coordinate lies between 3,000 and 4,000. In other words, we want to report all the points inside an axis-parallel query rectangle—see Figure 5.1.

What if we also have information about the number of children of each employee, and we would like to be able to ask queries like "report all employees born between 1950 and 1955 who earn between $3,000 and $4,000 a month and have between two and four children"? In this case we represent each employee by a point in 3-dimensional space: the first coordinate represents the date of birth, the second coordinate the salary, and the third coordinate the number of children. To answer the query we now have to report all points inside the axis-parallel box $[19,500,000 : 19,559,999] \times [3,000 : 4,000] \times [2 : 4]$. In general, if we are interested in answering queries on d fields of the records in our database, we transform the records to points in d-dimensional space. A query asking to report all records whose fields lie between specified values then transforms to a query asking for all points inside a d-dimensional axis-parallel box. Such a query is called a *rectangular range query*, or an *orthogonal range query*, in computational geometry. In this chapter we shall study data structures for such queries.

5.1 1-Dimensional Range Searching

Before we try to tackle the 2- or higher-dimensional rectangular range searching problem, let's have a look at the 1-dimensional version. The data we are given is a set of points in 1-dimensional space—in other words, a set of real numbers. A query asks for the points inside a 1-dimensional query rectangle—in other words, an interval $[x : x']$.

Let $P := \{p_1, p_2, \ldots, p_n\}$ be the given set of points on the real line. We can solve the 1-dimensional range searching problem efficiently using a well-known data structure: a balanced binary search tree $\mathcal{T}$. (A solution that uses an array is also possible. This solution does not generalize to higher dimensions, however, nor does it allow for efficient updates on P.) The leaves of $\mathcal{T}$ store the points of P and the internal nodes of $\mathcal{T}$ store splitting values to guide the search. We denote the splitting value stored at a node v by x_v. We assume that the left subtree of a node v contains all the points smaller than or equal to x_v, and that the right subtree contains all the points strictly greater than x_v.

To report the points in a query range $[x : x']$ we proceed as follows. We search with x and x' in $\mathcal{T}$. Let μ and μ' be the two leaves where the searches end, respectively. Then the points in the interval $[x : x']$ are the ones stored in the leaves in between μ and μ' plus, possibly, the point stored at μ and the point stored at μ'. When we search with the interval $[18 : 77]$ in the tree of Figure 5.2, for instance, we have to report all the points stored in the dark grey leaves, plus the point stored in the leaf μ. How can we find the leaves in between μ and μ'? As Figure 5.2 already suggests, they are the leaves of certain subtrees in between the search paths to μ and μ'. (In Figure 5.2, these subtrees are dark
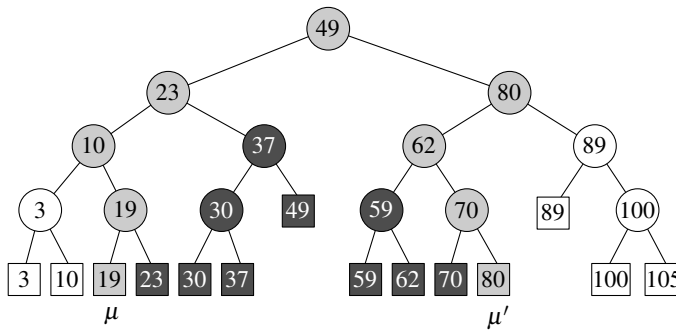
Figure 5.2
A 1-dimensional range query in a binary
search tree

grey, whereas the nodes on the search paths are light grey.) More precisely, the
subtrees that we select are rooted at nodes v in between the two search paths
whose parents are on the search path. To find these nodes we first search for
the node v_{split} where the paths to x and x' split. This is done with the following
subroutine. Let $lc(v)$ and $rc(v)$ denote the left and right child, respectively, of a
node v.

FINDSPLITNODE($\mathcal{T}, x, x'$)
Input. A tree $\mathcal{T}$ and two values x and x' with $x \leqslant x'$.
Output. The node v where the paths to x and x' split, or the leaf where both
 paths end.
1. $v \leftarrow root(\mathcal{T})$
2.   **while** v is not a leaf **and** ($x' \leqslant x_v$ **or** $x > x_v$)
3.   **do if** $x' \leqslant x_v$
4.   **then** $v \leftarrow lc(v)$
5.   **else** $v \leftarrow rc(v)$
6.   **return** v

Starting from v_{split} we then follow the search path of x. At each node where the
path goes left, we report all the leaves in the right subtree, because this subtree
is in between the two search paths. Similarly, we follow the path of x' and we
report the leaves in the left subtree of nodes where the path goes right. Finally,
we have to check the points stored at the leaves where the paths end; they may
or may not lie in the range $[x : x']$.

   Next we describe the query algorithm in more detail. It uses a subroutine
REPORTSUBTREE, which traverses the subtree rooted at a given node and
reports the points stored at its leaves. Since the number of internal nodes of any
binary tree is less than its number of leaves, this subroutine takes an amount of
time that is linear in the number of reported points.

Algorithm 1DRANGEQUERY($\mathcal{T}, [x : x']$)
Input. A binary search tree $\mathcal{T}$ and a range $[x : x']$.
Output. All points stored in $\mathcal{T}$ that lie in the range.
1. $v_{\text{split}} \leftarrow$FINDSPLITNODE($\mathcal{T}, x, x'$)
2.   **if** v_{split} is a leaf
3.   **then** Check if the point stored at v_{split} must be reported.

the selected subtrees
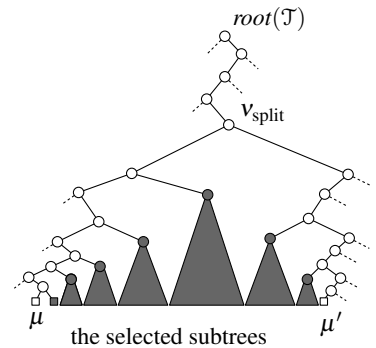
4.   **else** (∗ Follow the path to x and report the points in subtrees right of the path. ∗)

5. $v \leftarrow lc(v_{\text{split}})$

6.   **while** v is not a leaf

7.   **do if** $x \leqslant x_v$

8.   **then** REPORTSUBTREE($rc(v)$)

9. $v \leftarrow lc(v)$

10.   **else** $v \leftarrow rc(v)$

11.   Check if the point stored at the leaf v must be reported.

12.   Similarly, follow the path to x', report the points in subtrees left of the path, and check if the point stored at the leaf where the path ends must be reported.

We first prove the correctness of the algorithm.

Lemma 5.1 *Algorithm* 1DRANGEQUERY *reports exactly those points that lie in the query range.*

Proof. We first show that any reported point p lies in the query range. If p is stored at the leaf where the path to x or to x' ends, then p is tested explicitly for inclusion in the query range. Otherwise, p is reported in a call to REPORTSUB-TREE. Assume this call was made when we followed the path to x. Let v be the node on the path such that p was reported in the call REPORTSUBTREE($rc(v)$). Since v and, hence, $rc(v)$ lie in the left subtree of v_{split}, we have $p \leqslant x_{v_{\text{split}}}$. Because the search path of x' goes right at v_{split} this means that $p < x'$. On the other hand, the search path of x goes left at v and p is in the right subtree of v, so $x < p$. It follows that $p \in [x : x']$. The proof that p lies in the range when it is reported while following the path to x' is symmetrical.

It remains to prove that any point p in the range is reported. Let μ be the leaf where p is stored, and let v be the lowest ancestor of μ that is visited by the query algorithm. We claim that $v = \mu$, which implies that p is reported. Assume for a contradiction that $v \neq \mu$. Observe that v cannot be a node visited in a call to REPORTSUBTREE, because all descendants of such a node are visited. Hence, v is either on the search path to x, or on the search path to x', or both. Because all three cases are similar, we only consider the third case. Assume first that μ is in the left subtree of v. Then the search path of x goes right at v (otherwise v would not be the lowest visited ancestor). But this implies that $p < x$. Similarly, if μ is in the right subtree of v, then the path of x' goes left at v, and $p > x'$. In both cases, the assumption that p lies in the range is contradicted.   ☐

We now turn our attention to the performance of the data structure. Because it is a balanced binary search tree, it uses $O(n)$ storage and it can be built in $O(n \log n)$ time. What about the query time? In the worst case all the points could be in the query range. In this case the query time will be $\Theta(n)$, which seems bad. Indeed, we do not need any data structure to achieve $\Theta(n)$ query time; simply checking all the points against the query range leads to the same result. On the other hand, a query time of $\Theta(n)$ cannot be avoided when we have to report all the points. Therefore we shall give a more refined analysis

of the query time. The refined analysis takes not only n, the number of points in the set P, into account, but also k, the number of reported points. In other words, we will show that the query algorithm is *output-sensitive*, a concept we already encountered in Chapter 2.

Recall that the time spent in a call to REPORTSUBTREE is linear in the number of reported points. Hence, the total time spent in all such calls is $O(k)$. The remaining nodes that are visited are nodes on the search path of x or x'. Because $\mathcal{T}$ is balanced, these paths have length $O(\log n)$. The time we spend at each node is $O(1)$, so the total time spent in these nodes is $O(\log n)$, which gives a query time of $O(\log n + k)$.

The following theorem summarizes the results for 1-dimensional range searching:

Theorem 5.2 *Let P be a set of n points in 1-dimensional space. The set P can be stored in a balanced binary search tree, which uses $O(n)$ storage and has $O(n \log n)$ construction time, such that the points in a query range can be reported in time $O(k + \log n)$, where k is the number of reported points.*

5.2 Kd-Trees

Now let's go to the 2-dimensional rectangular range searching problem. Let P be a set of n points in the plane. In the remainder of this section we assume that no two points in P have the same x-coordinate, and no two points have the same y-coordinate. This restriction is not very realistic, especially not if the points represent employees and the coordinates are things like salary or number of children. Fortunately, the restriction can be overcome with a nice trick that we describe in Section 5.5.

A 2-dimensional rectangular range query on P asks for the points from P lying inside a query rectangle $[x : x'] \times [y : y']$. A point $p := (p_x, p_y)$ lies inside this rectangle if and only if

$$p_x \in [x : x'] \qquad \text{and} \qquad p_y \in [y : y'].$$

We could say that a 2-dimensional rectangular range query is composed of two 1-dimensional sub-queries, one on the x-coordinate of the points and one on the y-coordinate.

In the previous section we saw a data structure for 1-dimensional range queries. How can we generalize this structure—which was just a binary search tree—to 2-dimensional range queries? Let's consider the following recursive definition of the binary search tree: the set of (1-dimensional) points is split into two subsets of roughly equal size; one subset contains the points smaller than or equal to the splitting value, the other subset contains the points larger than the splitting value. The splitting value is stored at the root, and the two subsets are stored recursively in the two subtrees.

In the 2-dimensional case each point has two values that are important: its x- and its y-coordinate. Therefore we first split on x-coordinate, next on
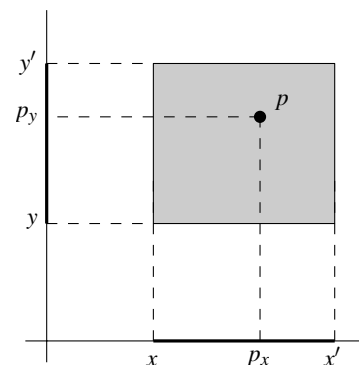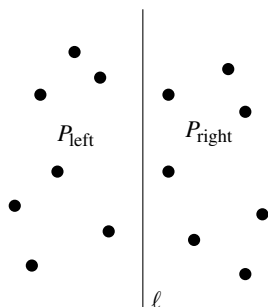
y-coordinate, then again on x-coordinate, and so on. More precisely, the process is as follows. At the root we split the set P with a vertical line ℓ into two subsets of roughly equal size. The splitting line is stored at the root. P_{left}, the subset of points to the left or on the splitting line, is stored in the left subtree, and P_{right}, the subset to the right of it, is stored in the right subtree. At the left child of the root we split P_{left} into two subsets with a horizontal line; the points below or on it are stored in the left subtree of the left child, and the points above it are stored in the right subtree. The left child itself stores the splitting line. Similarly, the set P_{right} is split with a horizontal line into two subsets, which are stored in the left and right subtree of the right child. At the grandchildren of the root, we split again with a vertical line. In general, we split with a vertical line at nodes whose depth is even, and we split with a horizontal line at nodes whose depth is odd. Figure 5.3 illustrates how the splitting is done and what the corresponding binary tree looks like. A tree like this is called a *kd-tree*. Originally, the name
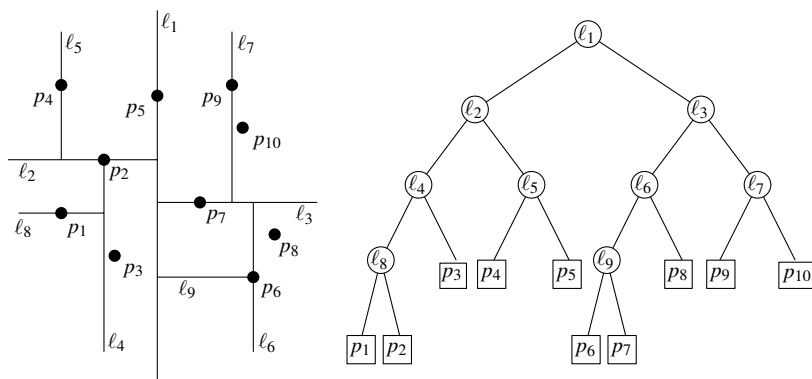

Figure 5.3
A kd-tree: on the left the way the plane
is subdivided and on the right the
corresponding binary tree

stood for k-dimensional tree; the tree we described above would be a 2d-tree. Nowadays, the original meaning is lost, and what used to be called a 2d-tree is now called a 2-dimensional kd-tree.

We can construct a kd-tree with the recursive procedure described below. This procedure has two parameters: a set of points and an integer. The first parameter is the set for which we want to build the kd-tree; initially this is the set P. The second parameter is depth of recursion or, in other words, the depth of the root of the subtree that the recursive call constructs. The depth parameter is zero at the first call. The depth is important because, as explained above, it determines whether we must split with a vertical or a horizontal line. The procedure returns the root of the kd-tree.

Algorithm BUILDKDTREE($P, depth$)
Input. A set of points P and the current depth *depth*.
Output. The root of a kd-tree storing P.
1.   **if** P contains only one point
2.   **then return** a leaf storing this point
3.   **else if** *depth* is even
4.   **then** Split P into two subsets with a vertical line ℓ through the median x-coordinate of the points in P. Let P_1 be the set of

points to the left of ℓ or on ℓ, and let P_2 be the set of points to the right of ℓ.

5.   **else** Split P into two subsets with a horizontal line ℓ through the median y-coordinate of the points in P. Let P_1 be the set of points below ℓ or on ℓ, and let P_2 be the set of points above ℓ.

6. $v_{\text{left}} \leftarrow$ BUILDKDTREE($P_1, depth + 1$)
7. $v_{\text{right}} \leftarrow$ BUILDKDTREE($P_2, depth + 1$)
8.   Create a node v storing ℓ, make v_{left} the left child of v, and make v_{right} the right child of v.
9.   **return** v

The algorithm uses the convention that the point on the splitting line—the one determining the median x- or y-coordinate—belongs to the subset to the left of, or below, the splitting line. For this to work correctly, the median of a set of n numbers should be defined as the $\lceil n/2 \rceil$-th smallest number. This means that the median of two values is the smaller one, which ensures that the algorithm terminates.

Before we come to the query algorithm, let's analyze the construction time of a 2-dimensional kd-tree. The most expensive step that is performed at every recursive call is finding the splitting line. This requires determining the median x-coordinate or the median y-coordinate, depending on whether the depth is even or odd. Median finding can be done in linear time. Linear time median finding algorithms, however, are rather complicated. A better approach is to presort the set of points both on x- and on y-coordinate. The parameter set P is now passed to the procedure in the form of two sorted lists, one on x-coordinate and one on y-coordinate. Given the two sorted lists, it is easy to find the median x-coordinate (when the depth is even) or the median y-coordinate (when the depth is odd) in linear time. It is also easy to construct the sorted lists for the two recursive calls in linear time from the given lists. Hence, the building time $T(n)$ satisfies the recurrence

$$T(n) = \begin{cases} O(1), & \text{if } n = 1, \\ O(n) + 2T(\lceil n/2 \rceil), & \text{if } n > 1, \end{cases}$$

which solves to $O(n \log n)$. This bound subsumes the time we spend for presorting the points on x- and y-coordinate.

To bound the amount of storage we note that each leaf in the kd-tree stores a distinct point of P. Hence, there are n leaves. Because a kd-tree is a binary tree, and every leaf and internal node uses $O(1)$ storage, this implies that the total amount of storage is $O(n)$. This leads to the following lemma.

Lemma 5.3 *A kd-tree for a set of n points uses $O(n)$ storage and can be constructed in $O(n \log n)$ time.*

We now turn to the query algorithm. The splitting line stored at the root partitions the plane into two half-planes. The points in the left half-plane are stored in the left subtree, and the points in the right half-plane are stored in the

right subtree. In a sense, the left child of the root corresponds to the left half-plane and the right child corresponds to the right half-plane. (The convention used in BUILDKDTREE that the point on the splitting line belongs to the left subset implies that the left half-plane is closed to the right and the right half-plane is open to the left.) The other nodes in a kd-tree correspond to a region of the plane as well. The left child of the left child of the root, for instance, corresponds to the region bounded to the right by the splitting line stored at the root and bounded from above by the line stored at the left child of the root. In general, the region corresponding to a node v is a rectangle, which can be unbounded on one or more sides. It is bounded by splitting lines stored at ancestors of v—see Figure 5.4. We denote the region corresponding to a node
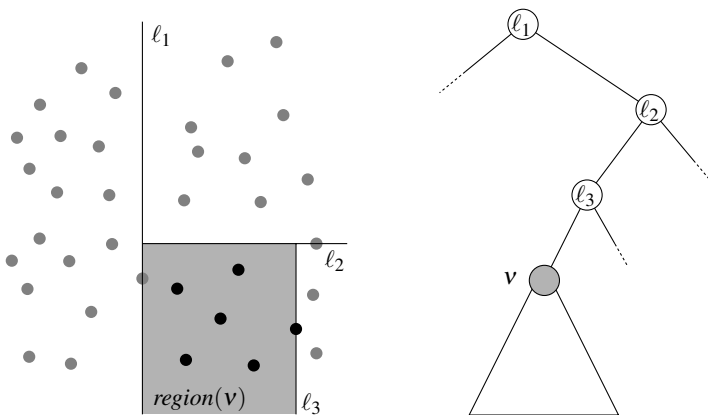

Figure 5.4
Correspondence between nodes in a
kd-tree and regions in the plane

v by *region(v)*. The region of the root of a kd-tree is simply the whole plane. Observe that a point is stored in the subtree rooted at a node v if and only if it lies in *region(v)*. For instance, the subtree of the node v in Figure 5.4 stores the points indicated as black dots. Therefore we have to search the subtree rooted at v only if the query rectangle intersects *region(v)*. This observation leads to the following query algorithm: we traverse the kd-tree, but visit only nodes whose region is intersected by the query rectangle. When a region is fully contained in the query rectangle, we can report all the points stored in its subtree. When the traversal reaches a leaf, we have to check whether the point stored at the leaf is contained in the query region and, if so, report it. Figure 5.5 illustrates the query algorithm. (Note that the kd-tree of Figure 5.5 could not have been constructed by Algorithm BUILDKDTREE; the median wasn't always chosen as the split value.) The grey nodes are visited when we query with the grey rectangle. The node marked with a star corresponds to a region that is completely contained in the query rectangle; in the figure this rectangular region is shown darker. Hence, the dark grey subtree rooted at this node is traversed and all points stored in it are reported. The other leaves that are visited correspond to regions that are only partially inside the query rectangle. Hence, the points stored in them must be tested for inclusion in the query range; this results in points p_6 and p_{11} being reported, and points p_3, p_{12}, and p_{13} not being reported. The query algorithm is described by the following recursive
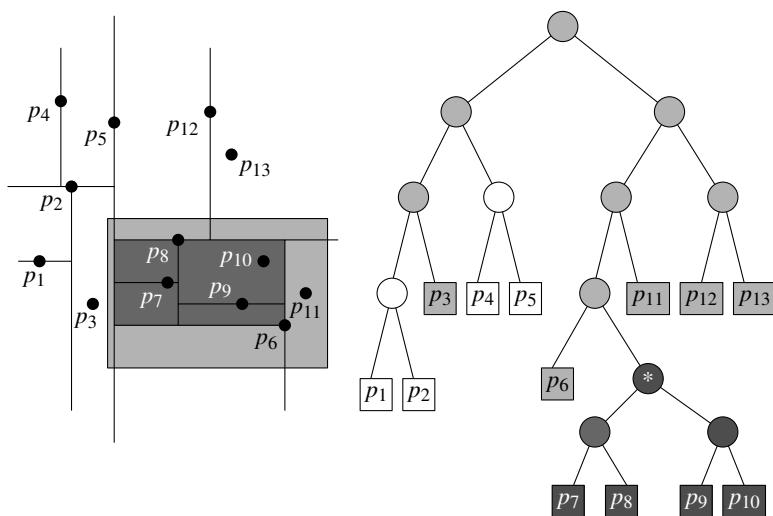
Figure 5.5
A query on a kd-tree

procedure, which takes as arguments the root of a kd-tree and the query range R. It uses a subroutine REPORTSUBTREE(v), which traverses the subtree rooted at a node v and reports all the points stored at its leaves. Recall that $lc(v)$ and $rc(v)$ denote the left and right child of a node v, respectively.

Algorithm SEARCHKDTREE(v, R)
*Input.* The root of (a subtree of) a kd-tree, and a range R.
Output. All points at leaves below v that lie in the range.
1.   **if** v is a leaf
2.   **then** Report the point stored at v if it lies in R.
3.   **else if** *region*($lc(v)$) is fully contained in R
4.   **then** REPORTSUBTREE($lc(v)$)
5.   **else if** *region*($lc(v)$) intersects R
6.   **then** SEARCHKDTREE($lc(v), R$)
7.   **if** *region*($rc(v)$) is fully contained in R
8.   **then** REPORTSUBTREE($rc(v)$)
9.   **else if** *region*($rc(v)$) intersects R
10.   **then** SEARCHKDTREE($rc(v), R$)

The main test the query algorithm performs is whether the query range R intersects the region corresponding to some node v. To be able to do this test we can compute *region*(v) for all nodes v during the preprocessing phase and store it, but this is not necessary: one can maintain the current region through the recursive calls using the lines stored in the internal nodes. For instance, the region corresponding to the left child of a node v at even depth can be computed from *region*(v) as follows:

$$region(lc(v)) = region(v) \cap \ell(v)^{\text{left}},$$

where $\ell(v)$ is the splitting line stored at v, and $\ell(v)^{\text{left}}$ is the half-plane to the left of and including $\ell(v)$.
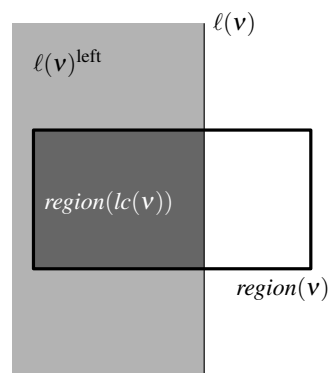
Observe that the query algorithm above never assumes that the query range R is a rectangle. Indeed, it works for any other query range as well.

We now analyze the time a query with a rectangular range takes.

Lemma 5.4 *A query with an axis-parallel rectangle in a kd-tree storing n points can be performed in $O(\sqrt{n}+k)$ time, where k is the number of reported points.*

Proof. First of all, note that the time to traverse a subtree and report the points stored in its leaves is linear in the number of reported points. Hence, the total time required for traversing subtrees in steps 4 and 8 is $O(k)$, where k is the total number of reported points. It remains to bound the number of nodes visited by the query algorithm that are not in one of the traversed subtrees. (These are the light grey nodes in Figure 5.5.) For each such node v, the query range properly intersects $region(v)$, that is, $region(v)$ is intersected by, but not fully contained in the range. In other words, the boundary of the query range intersects $region(v)$. To analyze the number of such nodes, we shall bound the number of regions intersected by any vertical line. This will give us an upper bound on the number of regions intersected by the left and right edge of the query rectangle. The number of regions intersected by the bottom and top edges of the query range can be bounded in the same way.

Let ℓ be a vertical line, and let $\mathcal{T}$ be a kd-tree. Let $\ell(root(\mathcal{T}))$ be the splitting line stored at the root of the kd-tree. The line ℓ intersects either the region to the left of $\ell(root(\mathcal{T}))$ or the region to the right of $\ell(root(\mathcal{T}))$, but not both. This observation seems to imply that $Q(n)$, the number of intersected regions in a kd-tree storing a set of n points, satisfies the recurrence $Q(n) = 1 + Q(n/2)$. But this is not true, because the splitting lines are horizontal at the children of the root. This means that if the line ℓ intersects for instance $region(lc(root(\mathcal{T}))$, then it will always intersect the regions corresponding to both children of $lc(root(\mathcal{T}))$. Hence, the recursive situation we get is not the same as the original situation, and the recurrence above is incorrect. To overcome this problem we have to make sure that the recursive situation is exactly the same as the original situation: the root of the subtree must contain a vertical splitting line. This leads us to redefine $Q(n)$ as the number of intersected regions in a kd-tree storing n points whose root contains a vertical splitting line. To write a recurrence for $Q(n)$ we now have to go down two steps in the tree. Each of the four nodes at depth two in the tree corresponds to a region containing $n/4$ points. (To be precise, a region can contain at most $\lceil \lceil n/2 \rceil /2 \rceil = \lceil n/4 \rceil$ points, but asymptotically this does not influence the outcome of the recurrence below.) Two of the four nodes correspond to intersected regions, so we have to count the number of intersected regions in these subtrees recursively. Moreover, ℓ intersects the region of the root and of one of its children. Hence, $Q(n)$ satisfies the recurrence
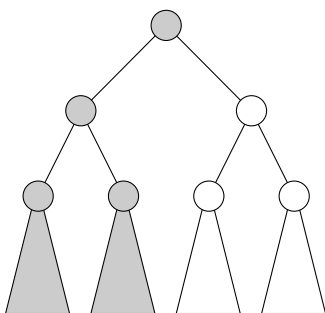
$$Q(n) = \begin{cases} O(1), & \text{if } n = 1, \\ 2 + 2Q(n/4), & \text{if } n > 1. \end{cases}$$

This recurrence solves to $Q(n) = O(\sqrt{n})$. In other words, any vertical line intersects $O(\sqrt{n})$ regions in a kd-tree. In a similar way one can prove that the

total number of regions intersected by a horizontal line is $O(\sqrt{n})$. The total number of regions intersected by the boundary of a rectangular query range is bounded by $O(\sqrt{n})$ as well. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad$ ☐

The analysis of the query time that we gave above is rather pessimistic: we bounded the number of regions intersecting an edge of the query rectangle by the number of regions intersecting the line through it. In many practical situations the range will be small. As a result, the edges are short and will intersect much fewer regions. For example, when we search with a range $[x : x] \times [y : y]$—this query effectively asks whether the point (x, y) is in the set—the query time is bounded by $O(\log n)$.

The following theorem summarizes the performance of kd-trees.

Theorem 5.5 *A kd-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. A rectangular range query on the kd-tree takes $O(\sqrt{n} + k)$ time, where k is the number of reported points.*

Kd-trees can also be used for point sets in 3- or higher-dimensional space. The construction algorithm is very similar to the planar case: At the root, we split the set of points into two subsets of roughly the same size by a hyperplane perpendicular to the x_1-axis. In other words, at the root the point set is partitioned based on the first coordinate of the points. At the children of the root the partition is based on the second coordinate, at nodes at depth two on the third coordinate, and so on, until at depth $d - 1$ we partition on the last coordinate. At depth d we start all over again, partitioning on first coordinate. The recursion stops when there is only one point left, which is then stored at a leaf. Because a d-dimensional kd-tree for a set of n points is a binary tree with n leaves, it uses $O(n)$ storage. The construction time is $O(n \log n)$. (As usual, we assume d to be a constant.)

Nodes in a d-dimensional kd-tree correspond to regions, as in the plane. The query algorithm visits those nodes whose regions are properly intersected by the query range, and traverses subtrees (to report the points stored in the leaves) that are rooted at nodes whose region is fully contained in the query range. It can be shown that the query time is bounded by $O(n^{1-1/d} + k)$.

5.3 Range Trees

Kd-trees, which were described in the previous section, have $O(\sqrt{n} + k)$ query time. So when the number of reported points is small, the query time is relatively high. In this section we shall describe another data structure for rectangular range queries, the *range tree*, which has a better query time, namely $O(\log^2 n + k)$. The price we have to pay for this improvement is an increase in storage from $O(n)$ for kd-trees to $O(n \log n)$ for range trees.

As we observed before, a 2-dimensional range query is essentially composed of two 1-dimensional sub-queries, one on the x-coordinate of the points and one
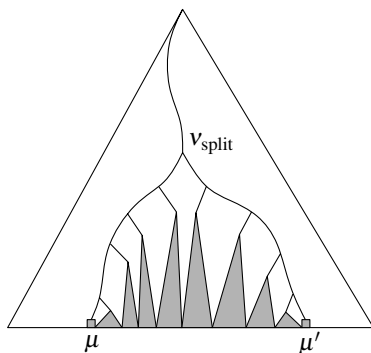
on the y-coordinate. This gave us the idea to split the given point set alternately on x- and y-coordinate, leading to the kd-tree. To obtain the range tree, we shall use this observation in a different way.

Let P be a set of n points in the plane that we want to preprocess for rectangular range queries. Let $[x : x'] \times [y : y']$ be the query range. We first concentrate on finding the points whose x-coordinate lies in $[x : x']$, the x-interval of the query rectangle, and worry about the y-coordinate later. If we only care about the x-coordinate then the query is a 1-dimensional range query. In Section 5.1 we have seen how to answer such a query: with a binary search tree on the x-coordinate of the points. The query algorithm was roughly as follows. We search with x and x' in the tree until we get to a node v_{split} where the search paths split. From the left child of v_{split} we continue the search with x, and at every node v where the search path of x goes left, we report all points in the right subtree of v. Similarly, we continue the search with x' at the right child of v_{split}, and at every node v where the search path of x' goes right we report all points in the left subtree of v. Finally, we check the leaves μ and μ' where the two paths end to see if they contain a point in the range. In effect, we select a collection of $O(\log n)$ subtrees that together contain exactly the points whose x-coordinate lies in the x-interval of the query rectangle.

Let's call the subset of points stored in the leaves of the subtree rooted at a node v the *canonical subset* of v. The canonical subset of the root of the tree, for instance, is the whole set P. The canonical subset of a leaf is simply the point stored at that leaf. We denote the canonical subset of node v by $P(v)$. We have just seen that the subset of points whose x-coordinate lies in a query range can be expressed as the disjoint union of $O(\log n)$ canonical subsets; these are the sets $P(v)$ of the nodes v that are the roots of the selected subtrees. We are not interested in all the points in such a canonical subset $P(v)$, but only want to report the ones whose y-coordinate lies in the interval $[y : y']$. This is another 1-dimensional query, which we can solve, provided we have a binary search tree on the y-coordinate of the points in $P(v)$ available. This leads to the following data structure for rectangular range queries on a set P of n points in the plane.

- The main tree is a balanced binary search tree $\mathcal{T}$ built on the x-coordinate of the points in P.

- For any internal or leaf node v in $\mathcal{T}$, the canonical subset $P(v)$ is stored in a balanced binary search tree $\mathcal{T}_{\text{assoc}}(v)$ on the y-coordinate of the points. The node v stores a pointer to the root of $\mathcal{T}_{\text{assoc}}(v)$, which is called the *associated structure* of v.

This data structure is called a range tree. Figure 5.6 shows the structure of a range tree. Data structures where nodes have pointers to associated structures are often called *multi-level data structures*. The main tree $\mathcal{T}$ is then called the *first-level tree*, and the associated structures are *second-level trees*. Multi-level data structures play an important role in computational geometry; more examples can be found in Chapters 10 and 16.

A range tree can be constructed with the following recursive algorithm, which receives as input the set $P := \{p_1, ..., p_n\}$ of points sorted on x-coordinate and

binary search tree on
x-coordinates

$\mathcal{T}$

v

$\mathcal{T}_{assoc}(v)$

binary search tree
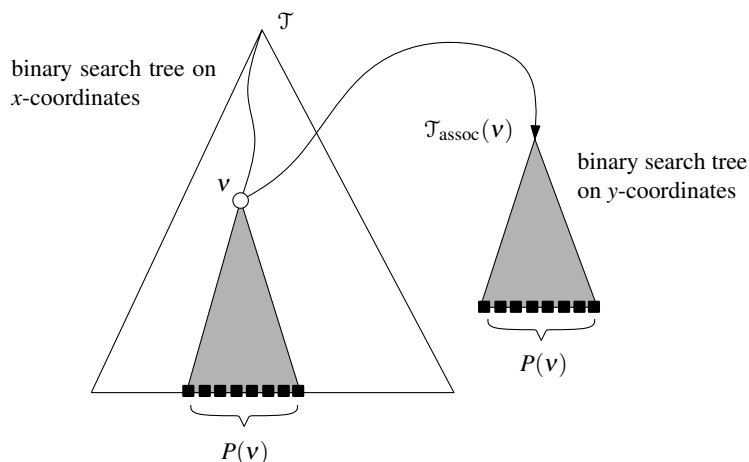on *y*-coordinates

$P(v)$

$P(v)$

Figure 5.6
A 2-dimensional range tree

returns the root of a 2-dimensional range tree $\mathcal{T}$ of P. As in the previous section, we assume that no two points have the same *x*- or *y*-coordinate. We shall get rid of this assumption in Section 5.5.

Algorithm BUILD2DRANGETREE(P)
*Input.* A set P of points in the plane.
Output. The root of a 2-dimensional range tree.
1. Construct the associated structure: Build a binary search tree $\mathcal{T}_{assoc}$ on the set P_y of *y*-coordinates of the points in P. Store at the leaves of $\mathcal{T}_{assoc}$ not just the *y*-coordinate of the points in P_y, but the points themselves.
2. **if** P contains only one point
3.   **then** Create a leaf v storing this point, and make $\mathcal{T}_{assoc}$ the associated structure of v.
4.   **else** Split P into two subsets; one subset P_{left} contains the points with *x*-coordinate less than or equal to x_{mid}, the median *x*-coordinate, and the other subset P_{right} contains the points with *x*-coordinate larger than x_{mid}.
5. $v_{left} \leftarrow$ BUILD2DRANGETREE(P_{left})
6. $v_{right} \leftarrow$ BUILD2DRANGETREE(P_{right})
7.   Create a node v storing x_{mid}, make v_{left} the left child of v, make v_{right} the right child of v, and make $\mathcal{T}_{assoc}$ the associated structure of v.
8. **return** v

Note that in the leaves of the associated structures we do not just store the *y*-coordinate of the points but the points themselves. This is important because, when searching the associated structures, we need to report the points and not just the *y*-coordinates.

Lemma 5.6 *A range tree on a set of n points in the plane requires $O(n \log n)$ storage.*

Proof. A point p in P is stored only in the associated structure of nodes on the path in $\mathcal{T}$ towards the leaf containing p. Hence, for all nodes at a given depth of $\mathcal{T}$,
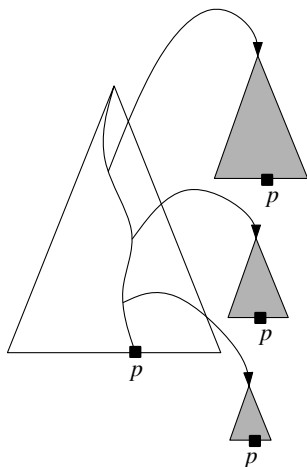
the point p is stored in exactly one associated structure. Because 1-dimensional range trees use linear storage it follows that the associated structures of all nodes at any depth of $\mathcal{T}$ together use $O(n)$ storage. The depth of $\mathcal{T}$ is $O(\log n)$. Hence, the total amount of storage required is bounded by $O(n \log n)$. ⬚

Algorithm BUILD2DRANGETREE as it is described will not result in the optimal construction time of $O(n \log n)$. To obtain this we have to be a bit careful. Constructing a binary search tree on an unsorted set of n keys takes $O(n \log n)$ time. This means that constructing the associated structure in line 1 would take $O(n \log n)$ time. But we can do better if the points in P_y are presorted on y-coordinate; then the binary search tree can be constructed bottom-up in linear time. During the construction algorithm we therefore maintain the set of points in two lists, one sorted on x-coordinate and one sorted on y-coordinate. This way the time we spend at a node in the main tree $\mathcal{T}$ is linear in the size of its canonical subset. This implies that the total construction time is the same as the amount of storage, namely $O(n \log n)$. Since the presorting takes $O(n \log n)$ time as well, the total construction time is again $O(n \log n)$.

The query algorithm first selects $O(\log n)$ canonical subsets that together contain the points whose x-coordinate lie in the range $[x : x']$. This can be done with the 1-dimensional query algorithm. Of those subsets, we then report the points whose y-coordinate lie in the range $[y : y']$. For this we also use the 1-dimensional query algorithm; this time it is applied to the associated structures that store the selected canonical subsets. Thus the query algorithm is virtually the same as the 1-dimensional query algorithm 1DRANGEQUERY; the only difference is that calls to REPORTSUBTREE are replaced by calls to 1DRANGEQUERY.

Algorithm 2DRANGEQUERY$(\mathcal{T}, [x : x'] \times [y : y'])$
Input. A 2-dimensional range tree $\mathcal{T}$ and a range $[x : x'] \times [y : y']$.
Output. All points in $\mathcal{T}$ that lie in the range.
1. $v_{\text{split}} \leftarrow$ FINDSPLITNODE$(\mathcal{T}, x, x')$
2. **if** v_{split} is a leaf
3. **then** Check if the point stored at v_{split} must be reported.
4. **else** ($*$ Follow the path to x and call 1DRANGEQUERY on the subtrees right of the path. $*$)
5. $v \leftarrow lc(v_{\text{split}})$
6. **while** v is not a leaf
7. **do if** $x \leqslant x_v$
8. **then** 1DRANGEQUERY$(\mathcal{T}_{\text{assoc}}(rc(v)), [y : y'])$
9. $v \leftarrow lc(v)$
10. **else** $v \leftarrow rc(v)$
11. Check if the point stored at v must be reported.
12. Similarly, follow the path from $rc(v_{\text{split}})$ to x', call 1DRANGE-QUERY with the range $[y : y']$ on the associated structures of subtrees left of the path, and check if the point stored at the leaf where the path ends must be reported.

Lemma 5.7 *A query with an axis-parallel rectangle in a range tree storing n points takes $O(\log^2 n + k)$ time, where k is the number of reported points.*

Proof. At each node v in the main tree $\mathcal{T}$ we spend constant time to decide where the search path continues, and we possibly call 1DRANGEQUERY. Theorem 5.2 states that the time we spend in this recursive call is $O(\log n + k_v)$, where k_v is the number of points reported in this call. Hence, the total time we spend is

$$\sum_v O(\log n + k_v),$$

where the summation is over all nodes in the main tree $\mathcal{T}$ that are visited. Notice that the sum $\sum_v k_v$ equals k, the total number of reported points. Furthermore, the search paths of x and x' in the main tree $\mathcal{T}$ have length $O(\log n)$. Hence, $\sum_v O(\log n) = O(\log^2 n)$. The lemma follows. $\quad\square$

The following theorem summarizes the performance of 2-dimensional range trees.

Theorem 5.8 *Let P be a set of n points in the plane. A range tree for P uses $O(n \log n)$ storage and can be constructed in $O(n \log n)$ time. By querying this range tree one can report the points in P that lie in a rectangular query range in $O(\log^2 n + k)$ time, where k is the number of reported points.*

The query time stated in Theorem 5.8 can be improved to $O(\log n + k)$ by a technique called *fractional cascading*. This is described in Section 5.6.

5.4 Higher-Dimensional Range Trees

It is fairly straightforward to generalize 2-dimensional range trees to higher-dimensional range trees. We only describe the global approach.

Let P be a set of points in d-dimensional space. We construct a balanced binary search tree on the first coordinate of the points. The canonical subset $P(v)$ of a node v in this first-level tree, the main tree, consists of the points stored in the leaves of the subtree rooted at v. For each node v we construct an associated structure $\mathcal{T}_{assoc}(v)$; the second-level tree $\mathcal{T}_{assoc}(v)$ is a $(d-1)$-dimensional range tree for the points in $P(v)$, restricted to their last $d-1$ coordinates. This $(d-1)$-dimensional range tree is constructed recursively in the same way: it is a balanced binary search tree on the second coordinate of the points, in which each node has a pointer to a $(d-2)$-dimensional range tree of the points in its subtree, restricted to the last $(d-2)$ coordinates. The recursion stops when we are left with points restricted to their last coordinate; these are stored in a 1-dimensional range tree—a balanced binary search tree.

The query algorithm is also very similar to the 2-dimensional case. We use the first-level tree to locate $O(\log n)$ nodes whose canonical subsets together contain all the points whose first coordinates are in the correct range. These canonical subsets are queried further by performing a range query on the corresponding second-level structures. In each second-level structure we select
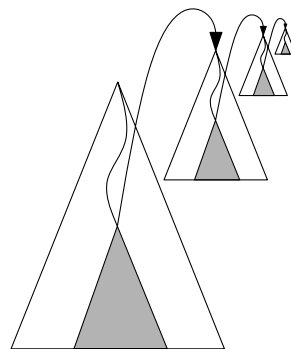
$O(\log n)$ canonical subsets. This means there are $O(\log^2 n)$ canonical subsets in the second-level structures in total. Together, they contain all points whose first and second coordinate lie in the correct ranges. The third-level structures storing these canonical subsets are then queried with 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 $\widehat{R}$. $\qquad \qquad \square$

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

number of points stored in the data structure, and k is the number of reported points.) In this section we describe a technique, called *fractional cascading*, to reduce the query time to $O(\log n + k)$.

Let's briefly recall how a range tree works. A range tree for a set P of points in the plane is a two-level data structure. The main tree is a binary search tree on the x-coordinate of the points. Each node ν in the main tree has an associated structure $\mathcal{T}_{\mathrm{assoc}}(\nu)$, which is a binary search tree on the y-coordinate of the points in $P(\nu)$, the canonical subset of ν. A query with a rectangular range $[x : x'] \times [y : y']$ is performed as follows: First, a collection of $O(\log n)$ nodes in the main tree is identified whose canonical subsets together contain the points with x-coordinate in the range $[x : x']$. Second, the associated structures of these nodes are queried with the range $[y : y']$. Querying an associated structure $\mathcal{T}_{\mathrm{assoc}}(\nu)$ is a 1-dimensional range query, so it takes $O(\log n + k_\nu)$ time, where k_ν is the number of reported points. Hence, the total query time is $O(\log^2 n + k)$.

If we could perform the searches in the associated structures in $O(1 + k_\nu)$ time, then the total query time would reduce to $O(\log n + k)$. But how can we do this? In general, it is not possible to answer a 1-dimensional range query in $O(1 + k)$ time, with k the number of answers. What saves us is that we have to do *many* 1-dimensional searches with the *same range*, and that we can use the result of one search to speed up other searches.

We first illustrate the idea of fractional cascading with a simple example. Let S_1 and S_2 be two sets of objects, where each object has a key that is a real number. These sets are stored in sorted order in arrays A_1 and A_2. Suppose we want to report all objects in S_1 and in S_2 whose keys lie in a query interval $[y : y']$. We can do this as follows: we do a binary search with y in A_1 to find the smallest key larger than or equal to y. From there we walk through the array to the right, reporting the objects we pass, until a key larger than y' is encountered. The objects from S_2 can be reported in a similar fashion. If the total number of reported objects is k, then the query time will be $O(k)$ plus the time for two binary searches, one in A_1 and one in A_2. If, however, the keys of the objects in S_2 are a subset of the keys of the objects in S_1, then we can avoid the second binary search as follows. We add pointers from the entries in A_1 to the entries in A_2: if $A_1[i]$ stores an object with key y_i, then we store a pointer to the entry in A_2 with the smallest key larger than or equal to y_i. If there is no such key then the pointer from $A_1[i]$ is **nil**. Figure 5.7 illustrates this. (Only the keys are shown in this figure, not the corresponding objects.)

How can we use this structure to report the objects in S_1 and S_2 whose keys are in a query interval $[y : y']$? Reporting the objects in S_1 is still done as before: do a binary search with y in A_1, and walk through A_1 to the right until a key larger than y' is encountered. To report the points from S_2 we proceed as follows. Let the search for y in A_1 end at $A[i]$. Hence, the key of $A[i]$ is the smallest one in S_1 that is larger than or equal to y. Since the keys from S_2 form a subset of the keys from S_1, this means that the pointer from $A[i]$ must point to the smallest key from S_2 larger than or equal to y. Hence, we can follow this pointer, and from there start to walk to the right through A_2. This way the binary search in
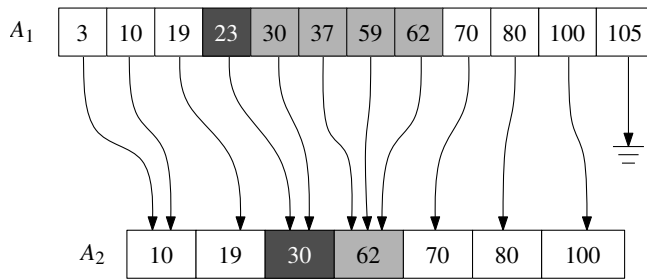
A_1 | 3 | 10 | 19 | 23 | 30 | 37 | 59 | 62 | 70 | 80 | 100 | 105

A_2 | 10 | 19 | 30 | 62 | 70 | 80 | 100

Figure 5.7
Speeding up the search by adding pointers

A_2 is avoided, and reporting the objects from S_2 takes only $O(1+k)$ time, with k the number of reported answers.

Figure 5.7 shows an example of a query. We query with the range $[20:65]$. First we use binary search in A_1 to find 23, the smallest key larger than or equal to 20. From there we walk to the right until we encounter a key larger than 65. The objects that we pass have their keys in the range, so they are reported. Then we follow the pointer from 23 into A_2. We get to the key 30, which is the smallest one larger than or equal to 20 in A_2. From there we also walk to the right until we reach a key larger than 65, and report the objects from S_2 whose keys are in the range.

Now let's return to range trees. The crucial observation here is that the canonical subsets $P(lc(v))$ and $P(rc(v))$ both are subsets of $P(v)$. As a result we can use the same idea to speed up the query time. The details are slightly more complicated, because we now have two subsets of $P(v)$ to which we need fast access rather than only one. Let $\mathcal{T}$ be a range tree on a set P of n points in the plane. Each canonical subset $P(v)$ is stored in an associated structure. But instead of using a binary search tree as associated structure, as we did in Section 5.3, we now store it in an array $A(v)$. The array is sorted on the y-coordinate of the points. Furthermore, each entry in an array $A(v)$ stores two pointers: a pointer into $A(lc(v))$ and a pointer into $A(rc(v))$. More precisely, we add the following pointers. Suppose that $A(v)[i]$ stores a point p. Then we store a pointer from $A(v)[i]$ to the entry of $A(lc(v))$ such that the y-coordinate of the point p' stored there is the smallest one larger than or equal to p_y. As noted above, $P(lc(v))$ is a subset of $P(v)$. Hence, if p has the smallest y-coordinate larger than or equal to some value y of any point in $P(v)$, then p' has the smallest y-coordinate larger than or equal to y of any point in $P(lc(v))$. The pointer into $A(rc(v))$ is defined in the same way: it points to the entry such that the y-coordinate of the point stored there is the smallest one that is larger than or equal to p_y.

This modified version of the range tree is called a *layered range tree*; Figures 5.8 and 5.9 show an example. (The positions in which the arrays are drawn corresponds to the positions of the nodes in the tree they are associated with: the topmost array is associated with the root, the left array below it is associated with the left child of the root, and so on. Not all pointers are shown in the figure.)

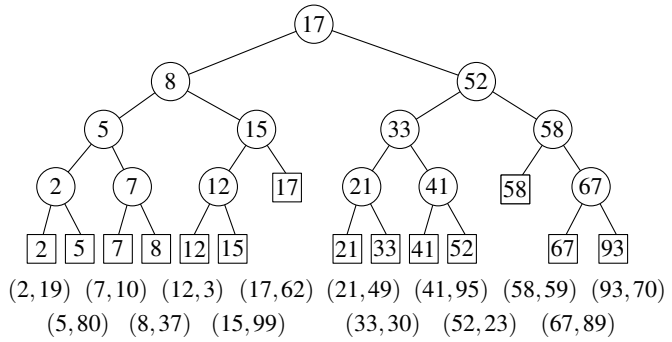Let's see how to answer a query with a range $[x:x'] \times [y:y']$ in a layered

Figure 5.8
The main tree of a layered range tree:
the leaves show only the *x*-coordinates;
the points stored are given below

(2,19) (7,10) (12,3) (17,62) (21,49) (41,95) (58,59) (93,70)
(5,80) (8,37) (15,99)   (33,30) (52,23) (67,89)

Figure 5.9
The arrays associated with the nodes in
the main tree, with the *y*-coordinate of
the points of the canonical subsets in
sorted order (not all pointers are shown)
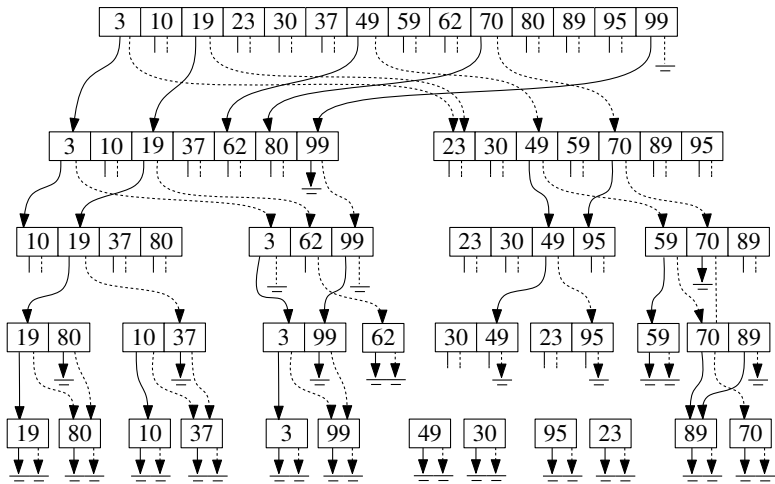
range tree. As before we search with x and x' in the main tree $\mathcal{T}$ to determine $O(\log n)$ nodes whose canonical subsets together contain the points with x-coordinate in the range $[x : x']$. These nodes are found as follows. Let v_{split} be the node where the two search paths split. The nodes that we are looking for are the ones below v_{split} that are the right child of a node on the search path to x where the path goes left, or the left child of a node on the search path to x' where the path goes right. At v_{split} we find the entry in $A(v_{\text{split}})$ whose y-coordinate is the smallest one larger than or equal to y. This can be done in $O(\log n)$ time by binary search. While we search further with x and x' in the main tree, we keep track of the entry in the associated arrays whose y-coordinate is the smallest one larger than or equal to y. They can be maintained in constant time by following the pointers stored in the arrays. Now let v be one of the $O(\log n)$ nodes we selected. We must report the points stored in $A(v)$ whose y-coordinate is in the range $[y : y']$. For this it suffices to be able to find the point with smallest y-coordinate larger than or equal to y; from there we can just walk through the array, reporting points as long as their y-coordinate is less than or equal to y'. This point can be found in constant time, because $parent(v)$ is on the search path, and we kept track of the points with smallest y-coordinate larger than or equal to y in the arrays on the search path. Hence, we can report the points of $A(v)$ whose y-coordinate is in the range $[y : y']$ in $O(1 + k_v)$ time, where k_v is the number of reported answers at node v. The total query time now becomes $O(\log n + k)$.

Fractional cascading also improves the query time of higher-dimensional range trees by a logarithmic factor. Recall that a d-dimensional range query was solved by first selecting the points whose d-th coordinate is in the correct range in $O(\log n)$ canonical subsets, and then solving a $(d-1)$-dimensional query on these subsets. The $(d-1)$-dimensional query is solved recursively in the same way. This continues until we arrive at a 2-dimensional query, which can be solved as described above. This leads to the following theorem.

Theorem 5.11 *Let P be a set of n points in d-dimensional space, with $d \geqslant 2$. A layered 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. With this range tree one can report the points in P that lie in a rectangular query range in $O(\log^{d-1} n + k)$ time, where k is the number of reported points.*

5.7 Notes and Comments

In the 1970s—the early days of computational geometry—orthogonal range searching was one of the most important problems in the field, and many people worked on it. This resulted in a large number of results, of which we discuss a few below.

One of the first data structures for orthogonal range searching was the quadtree, which is discussed in Chapter 14 in the context of meshing. Unfortunately, the worst-case behavior of quadtrees is quite bad. Kd-trees, de-

scribed first by Bentley [44] in 1975, are an improvement over quadtrees. Samet's books [333, 334] discuss quadtrees, kd-trees, and their applications in great detail. A few years later, the range tree was discovered independently by several people [46, 251, 261, 387]. The improvement in query time to $O(\log n + k)$ by fractional cascading was described by Lueker [261] and Willard [386]. Fractional cascading applies in fact not only to range trees, but in many situations where one does many searches with the same key. Chazelle and Guibas [105, 106] discuss this technique in its full generality. Fractional cascading can also be used in a dynamic setting [275]. The most efficient data structure for 2-dimensional range queries is a modified version of the layered range tree, described by Chazelle [87]; he succeeded in improving the storage to $O(n \log n / \log \log n)$ while keeping the query time $O(\log n + k)$. Chazelle [90, 91] also proved that this is optimal. If the query range is unbounded to one side, for instance when it is of the form $[x : x'] \times [y : +\infty]$, then one can achieve $O(\log n)$ query time with only linear space, using a priority search tree—see Chapter 10. In higher dimensions the best result for orthogonal range searching is also due to Chazelle [90]: he gave a structure for d-dimensional queries with $O(n(\log n / \log \log n)^{d-1})$ storage and polylogarithmic query time. Again, this result is optimal. Trade-offs between storage and query time are also possible [338, 391].

The lower-bound result is only valid under certain models of computation. This allows for improved results in specific cases. In particular, Overmars [300] describes more efficient data structures for range searching when the points lie on a $U \times U$ grid, yielding query time bounds of $O(\log \log U + k)$ or $O(\sqrt{U} + k)$, depending on the preprocessing time allowed. The results use data structures described earlier by Willard [389, 390]. When compared with the general case, better time bounds can be obtained for many problems in computational geometry once the coordinates of the objects are restricted to lie on grid points. Examples are the nearest neighbor searching problem [224, 225], point location [287], and line segment intersection [226].

In databases, range queries are considered the most general of three basic types of multi-dimensional queries. The two other types are *exact match queries* and *partial match queries*. Exact match queries are simply queries of the type: Is the object (point) with attribute values (coordinates) such and such present in the database? The obvious data structure for exact match queries is the balanced binary tree that uses, for instance, a lexicographical order on the coordinates. With this structure exact match queries can be answered in $O(\log n)$ time. If the dimension—the number of attributes—increases, it can be useful to express the efficiency of queries not only in terms of n, but also in the dimension d. If a binary tree is used for exact match queries, the query time is $O(d \log n)$ because it takes $O(d)$ time to compare two points. This can easily be reduced to $O(d + \log n)$ time, which is optimal. A partial match query specifies only a value for a subset of the coordinates and asks for all points with the specified coordinate values. In the plane, for instance, a partial match query specifies only an x-coordinate, or only a y-coordinate. Interpreted geometrically, a partial match query in the plane asks for the points on a horizontal line, or on

a vertical line. With a d-dimensional kd-tree, a partial match query specifying s coordinates (with $s < d$) can be answered in $O(n^{1-s/d} + k)$ time, where k is the number of reported points [44].

In many applications the data that we are given are not a set of points, but a set of certain objects such as polygons. If we want to report the objects that are completely contained in a query range $[x : x'] \times [y : y']$, then it is possible to transform the query to a query on point data in higher dimensions—see Exercise 5.13. Often one also wants to find the objects that are only partially in the range. This specific problem is called the windowing problem and is discussed in Chapter 10.

Other variants of the range searching problem are obtained by allowing other types of query ranges, such as circles or triangles. Many of these variants can be solved using so-called partition trees, which are discussed in Chapter 16.

5.8 Exercises

5.1 In the proof of the query time of the kd-tree we found the following recurrence:
$$Q(n) = \begin{cases} O(1), & \text{if } n = 1, \\ 2 + 2Q(n/4), & \text{if } n > 1. \end{cases}$$

Prove that this recurrence solves to $Q(n) = O(\sqrt{n})$. Also show that $\Omega(\sqrt{n})$ is a lower bound for querying in a kd-tree by defining a set of n points and a query rectangle appropriately.

5.2 Describe algorithms to insert and delete points from a kd-tree. In your algorithm you do not need to take care of rebalancing the structure.

5.3 In Section 5.2 it was indicated that kd-trees can also be used to store sets of points in higher-dimensional space. Let P be a set of n points in d-dimensional space. In parts a. and b. you may consider d to be a constant.

a. Describe an algorithm to construct a d-dimensional kd-tree for the points in P. Prove that the tree uses linear storage and that your algorithm takes $O(n \log n)$ time.
b. Describe the query algorithm for performing a d-dimensional range query. Prove that the query time is bounded by $O(n^{1-1/d} + k)$.
c. Show that the dependence on d in the amount of storage is linear, that is, show that the amount of storage is $O(dn)$ if we do not consider d to be a constant. Give the dependence on d of the construction time and the query time as well.

5.4 Kd-trees can be used for partial match queries. A 2-dimensional partial match query specifies a value for one of the coordinates and asks for all points that have that value for the specified coordinate. In higher

dimensions we specify values for a subset of the coordinates. Here we allow multiple points to have equal values for coordinates.

a. Show that 2-dimensional kd-trees can answer partial match queries in $O(\sqrt{n}+k)$ time, where k is the number of reported answers.

b. Explain how to use a 2-dimensional range tree to answer partial match queries. What is the resulting query time?

c. Describe a data structure that uses linear storage and solves 2-dimensional partial match queries in $O(\log n + k)$ time.

d. Show that with a d-dimensional kd-tree we can solve a d-dimensional partial match query in $O(n^{1-s/d}+k)$ time, where s (with $s < d$) is the number of specified coordinates.

e. Describe a data structure that uses linear storage and that can answer d-dimensional partial match queries in $O(\log n + k)$ time. *Hint:* Use a structure whose dependence on d in the amount of storage is exponential (more precisely, a structure that uses $O(d2^d n)$ storage).

5.5 Algorithm SEARCHKDTREE can also be used when querying with other ranges than rectangles. For example, a query is answered correctly if the range is a triangle.

a. Show that the query time for range queries with triangles is linear in the worst case, even if no answers are reported at all. *Hint:* Choose all points to be stored in the kd-tree on the line $y = x$.

b. Suppose that a data structure is needed that can answer triangular range queries, but only for triangles whose edges are horizontal, vertical, or have slope $+1$ or -1. Develop a linear size data structure that answers such range queries in $O(n^{3/4}+k)$ time, where k is the number of points reported. *Hint:* Choose 4 coordinate axes in the plane and use a 4-dimensional kd-tree.

c. Improve the query time to $O(n^{2/3}+k)$. *Hint:* Solve Exercise 5.4 first.

5.6 Describe algorithms to insert and delete points from a range tree. You don't have to take care of rebalancing the structure.

5.7 In the proof of Lemma 5.7 we made a rather rough estimate of the query time in the associated structures by stating that this was bounded by $O(\log n)$. In reality the query time is dependent on the actual number of points in the associated structure. Let n_v denote the number of points in the canonical subset $P(v)$. Then the total time spent is

$$\sum_v \Theta(\log n_v + k_v),$$

where the summation is over all nodes in the main tree $\mathcal{T}$ that are visited. Show that this bound is still $\Theta(\log^2 n + k)$. (That is, our more careful analysis only improves the constant in the bound, not the order of magnitude.)

5.8 Theorem 5.8 showed that a range tree on a set of n points in the plane requires $O(n \log n)$ storage. One could bring down the storage require-

ments by storing associated structures only with a subset of the nodes in the main tree.

a. Suppose that only the nodes with depth 0, 2, 4, 6, … have an associated structure. Show how the query algorithm can be adapted to answer queries correctly.
b. Analyze the storage requirements and query time of such a data structure.
c. Suppose that only the nodes with depth 0, $\lfloor \frac{1}{j} \log n \rfloor$, $\lfloor \frac{2}{j} \log n \rfloor$, … have an associated structure, where $j \geqslant 2$ is a constant. Analyze the storage requirements and query time of this data structure. Express the bounds in n and j.

5.9 One can use the data structures described in this chapter to determine whether a particular point (a, b) is in a given set by performing a range query with range $[a : a] \times [b : b]$.

a. Prove that performing such a range query on a kd-tree takes time $O(\log n)$.
b. What is the time bound for such a query on a range tree? Prove your answer.

5.10 In some applications one is interested only in the number of points that lie in a range rather than in reporting all of them. Such queries are often referred to as *range counting queries*. In this case one would like to avoid having an additive term of $O(k)$ in the query time.

a. Describe how a 1-dimensional range tree can be adapted such that a range counting query can be performed in $O(\log n)$ time. Prove the query time bound.
b. Using the solution to the 1-dimensional problem, describe how d-dimensional range counting queries can be answered in $O(\log^d n)$ time. Prove the query time.
c.* Describe how fractional cascading can be used to improve the running time with a factor of $O(\log n)$ for 2- and higher-dimensional range counting queries.

5.11 Let S_1 be a set of n disjoint horizontal line segments and let S_2 be a set of m disjoint vertical line segments. Give a plane-sweep algorithm that counts in $O((n+m)\log(n+m))$ time how many intersections there are in $S_1 \cup S_2$.

5.12 In Section 5.5 it was shown that one can treat sets of points in the plane that contain equal coordinate values using composite numbers. Extend this notion to points in d-dimensional space. To this end you should define the composite number of d numbers and define an appropriate order on them. Next, show how to transform the point $p := (p_1, \ldots, p_d)$ and the range $R := [r_1 : r'_1] \times \cdots \times [r_d : r'_d]$ using this order; the transformation should be such that $p \in R$ if and only if the transformed point lies in the transformed range.

5.13 In many application one wants to do range searching among objects other than points.

a. Let S be a set of n axis-parallel rectangles in the plane. We want to be able to report all rectangles in S that are completely contained in a query rectangle $[x : x'] \times [y : y']$. Describe a data structure for this problem that uses $O(n \log^3 n)$ storage and has $O(\log^4 n + k)$ query time, where k is the number of reported answers. *Hint:* Transform the problem to an orthogonal range searching problem in some higher-dimensional space.

b. Let P consist of a set of n polygons in the plane. Again describe a data structure that uses $O(n \log^3 n)$ storage and has $O(\log^4 n + k)$ query time to report all polygons completely contained in the query rectangle, where k is the number of reported answers.

c.* Improve the query time of your solutions (both for a. and b.) to $O(\log^3 n + k)$.

5.14* Prove the $O(n \log^{d-1} n)$ bounds on the storage and construction time in Theorem 5.11.