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.

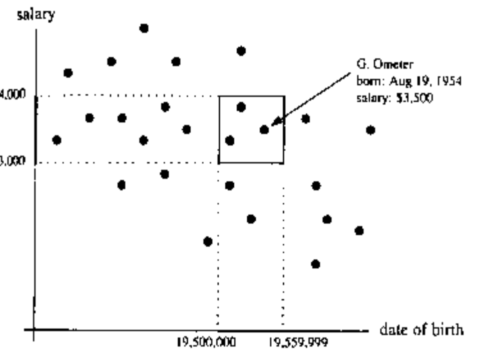
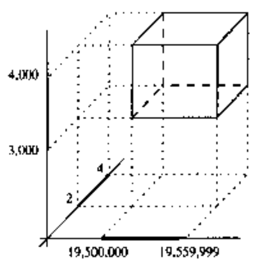


Figure 5.1
Interpreting a database query geometrically

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 ν by x_ν. We assume that the left subtree of a node ν contains all the points smaller than or equal to x_ν, and that the right subtree contains all the points strictly greater than x_ν.

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

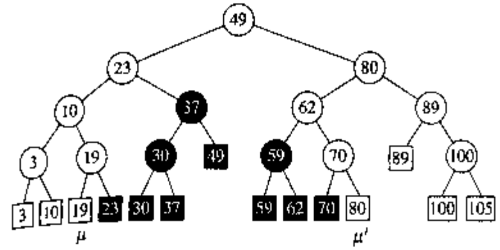
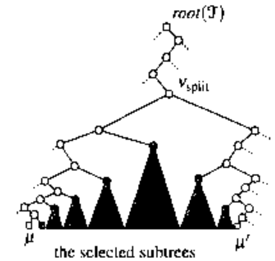


Figure 5.2
A 1-dimensional range query in a binary search tree

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

FINDSPLITNODE($\mathcal{T}, x, x'$)
Input. A tree $\mathcal{T}$ and two values x and x' with $x \leqslant x'$.
Output. The node ν where the paths to x and x' split, or the leaf where both paths end.

$\nu \leftarrow root(\mathcal{T})$
while ν is not a leaf **and** ($x' \leqslant x_\nu$ **or** $x > x_\nu$)
    do if $x' \leqslant x_\nu$
        then $\nu \leftarrow lc(\nu)$
        else $\nu \leftarrow rc(\nu)$
return ν



Starting from ν_{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.

$\nu_{split} \leftarrow$ FINDSPLITNODE($\mathcal{T}, x, x'$)
if ν_{split} is a leaf
    then Check if the point stored at ν_{split} must be reported.

```
else (* Follow the path to x and report the points in subtrees right of the
        path. *)
     ν ← lc(ν_split)
     while ν is not a leaf
         do if x ≤ x_ν
               then REPORTSUBTREE(rc(ν))
                    ν ← lc(ν)
               else ν ← rc(ν)
     Check if the point stored at the leaf ν must be reported.
     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 REPORTSUBTREE. Assume this call was made when we followed the path to x. Let ν be the node on the path such that p was reported in the call REPORTSUBTREE($rc(\nu)$). Since ν and, hence, $rc(\nu)$ lie in the left subtree of ν_{split}, we have $p \leqslant x_{\nu_{\text{split}}}$. Because the search path of x' goes right at ν_{split} this means that $p < x'$. On the other hand, the search path of x goes left at ν and p is in the right subtree of ν, 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 ν be the lowest ancestor of μ that is visited by the query algorithm. We claim that $\nu = \mu$, which implies that p is reported. Assume for a contradiction that $\nu \neq \mu$. Observe that ν cannot be a node visited in a call to REPORTSUBTREE, because all descendants of such a node are visited. Hence, ν 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 ν. Then the search path of x goes right at ν (otherwise ν would not be the lowest visited ancestor). But this implies that $p < x$. Similarly, if μ is in the right subtree of ν, then the path of x' goes left at ν, 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.

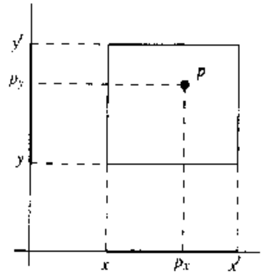
Recall that the time spent in a call to REPORTSUBTREE is linear in the number of reported points. Hence, the total time spent in all such calls is $O(k)$. The remaining nodes that are visited are nodes on the search path of x or x'. Because $\mathcal{T}$ is balanced, these paths have length $O(\log n)$.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

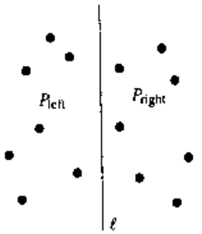
$$p_x \in [x : x'] \quad \text{and} \quad p_y \in [y : y'].$$

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 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.



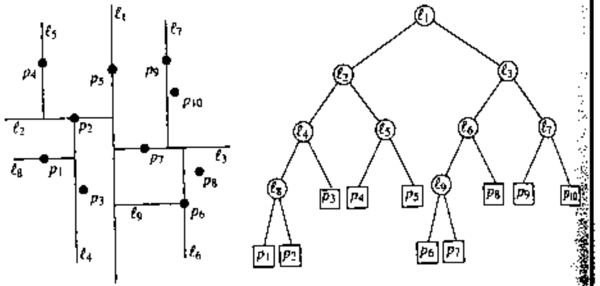


Figure 5.3
A kd-tree: on the left the way the plane is subdivided and on the right the corresponding binary tree

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.

if P contains only one point
  then return a leaf storing this point
  else if *depth* is even
    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.
    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.
  $v_{\text{left}} \leftarrow$ BUILDKDTREE(P_1, *depth* + 1)
  $v_{\text{right}} \leftarrow$ BUILDKDTREE(P_2, *depth* + 1)
  Create a node v storing ℓ, make v_{left} the left child of v, and make v_{right} the right child of v.
  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 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 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.

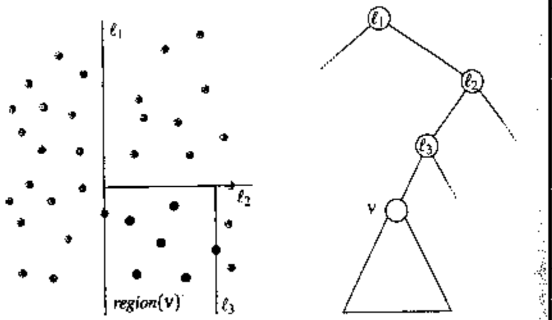


Figure 5.4
Correspondence between nodes in a kd-tree and regions in the plane

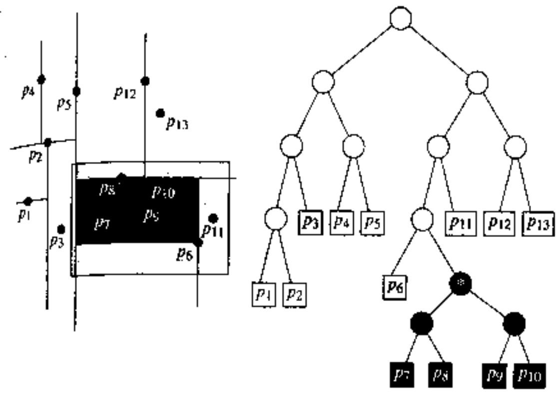
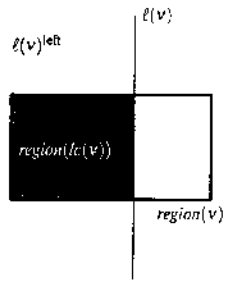


Figure 5.5
A query on a kd-tree

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.

```
If v is a leaf
  then Report the point stored at v if it lies in R.
  else if region(lc(v)) is fully contained in R
         then REPORTSUBTREE(lc(v))
         else if region(lc(v)) intersects R
                then SEARCHKDTREE(lc(v), R)
       if region(rc(v)) is fully contained in R
         then REPORTSUBTREE(rc(v))
         else if region(rc(v)) intersects R
                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 $\ell(v)$.

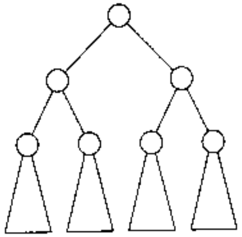
Observe that the query algorithm above never assumes that the query range R is a rectangle. Indeed, it works for any other query range as well.

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 ν, the query range properly intersects $region(\nu)$, that is, $region(\nu)$ is intersected by, but not fully contained in the range. In other words, the boundary of the query range intersects $region(\nu)$. 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. ⊟

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