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

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

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

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

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

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

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

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

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

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

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

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

5.6* Fractional Cascading

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

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 v in the main tree has an associated structure $\mathcal{T}_{\text{assoc}}(v)$, which is a binary search tree on the y-coordinate of the points in $P(v)$, the canonical subset of v. 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 $\mathcal{T}_{\text{assoc}}(v)$ is a 1-dimensional range query, so it takes $O(\log n + k_v)$ time, where k_v 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 $O(1 + k_v)$ 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

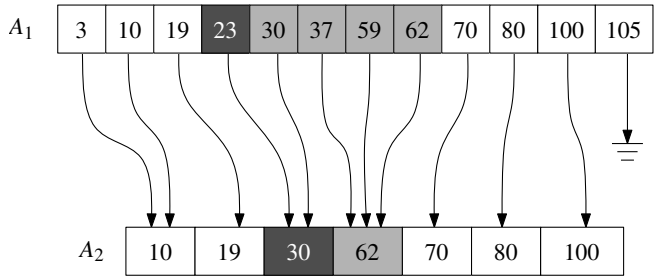

Figure 5.7
Speeding up the search by adding pointers

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

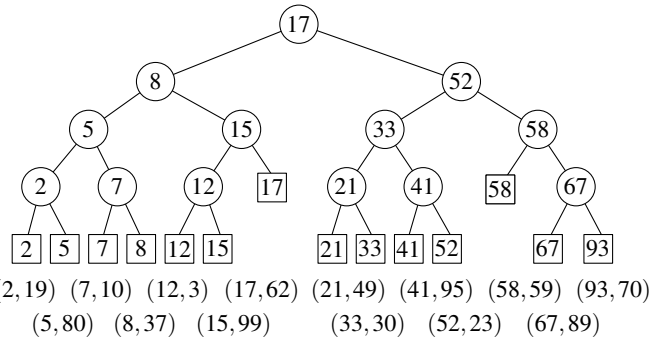

Figure 5.8
The main tree of a layered range tree: the leaves show only the x-coordinates; the points stored are given below

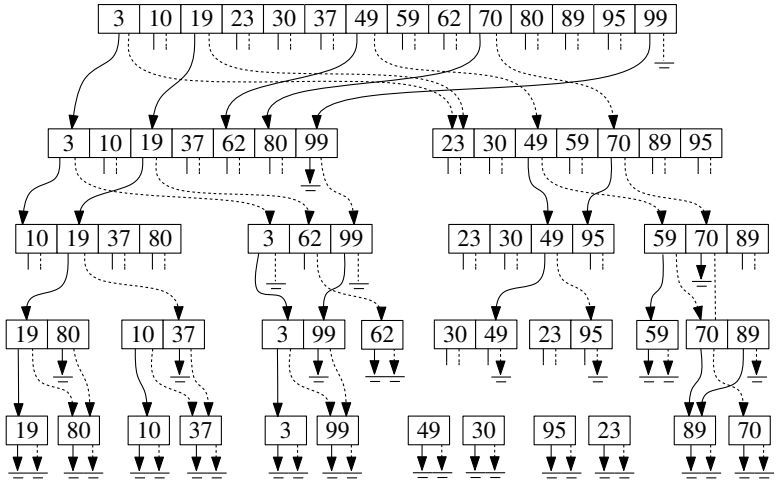

Figure 5.9
The arrays associated with the nodes in the main tree, with the y-coordinate of the points of the canonical subsets in sorted order (not all pointers are shown)

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 $\mathit{parent}(v)$ 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-