26 Range Searching

Given a set of points in the plane, it is natural to ask which of those points fall within some specified area. "List all cities within 50 miles of Princeton" is a question of this type which could reasonably be asked if a set of points corresponding to the cities of the U.S. were available. When the geometric shape is restricted to be a rectangle, the issue readily extends to non-geometric problems. For example, "list all those people between 21 and 25 with incomes between $60. 000 and $ 1 00. 000" asks which "points" from a file of data on people's names, ages, and incomes fall within a certain rectangle in the age-income plane.

Extension to more than two dimensions is immediate. If we want to list all stars within 50 light-years of the sun, we have a three-dimensional problem, and if we want the rich young people of the paragraph above to be tall and female as well, we have a four-dimensional problem. In fact, the dimension of such problems can get very high.

In general, we assume that we have a set of *records* with certain *attributes* that take on values from some ordered set. (This is sometimes called a *database*, though more specific and complete definitions have been developed for this important term.) Finding all records in a database that satisfy specified range restrictions on a specified set of attributes is called *range searching*, *and* is a difficult and important problem in practical applications. In this chapter, we'll concentrate on the two dimensional geometric problem in which records are points and attributes are their coordinates, and then discuss appropriate generalizations.

The methods we'll look at are direct generalizations of methods we have seen for searching on single keys (in one dimension). We presume that many queries will be made on the same set of points, so the problem splits into two parts: we need a *preprocessing* algorithm, which builds the given points into a structure supporting efficient range searching, and a *range-searching* algorithm, which uses the structure to return points falling within any given (multidimensional) range. This separation makes different methods difficult to compare, since the total cost depends not only on the distribution of the points involved but also on the number and nature of the queries.

The range-searching problem in one dimension is to return all points falling within a specified interval. This can be done by sorting the points for preprocessing and then doing a binary search on the endpoints of the interval to return all the points that fall in between. Another solution is to build a binary search tree and then do a simple recursive traversal of the tree, returning points within the interval and ignoring parts of the tree outside the interval. The program required is a simple recursive tree traversal (see Chapter 4). If the left endpoint of the interval falls to the left of the point at the root, we (recursively) search the left subtree, and similarly for the right, checking each node we encounter to see whether its point falls within the interval:

```
type interval = record xl , x2: integer end;
procedure treerange(t: link; int: interval);
  var txl. tx2: boolean;
  begin
  if t < >z then
      begin
      txl := t|.key >= int.xl;
      tx2 := t|.key < = int.x2;
      if txl then treerange (t|.l, int);
      if txl and tx2
          then t|.key is in the range
      if tx2 then treerange (t|.r.  int);
      end
  end;
```

(This program could be made slightly more efficient by maintaining the interval *int* as a global variable rather than passing its unchanged values through the recursive calls.) Figure 26.1 shows the points found when this program is run on a sample tree. Note that the points returned do not need to be connected in the tree.

Property 26.1 *One-dimensional range searching can be done with O(N logN) steps for preprocessing and O(R+IogN) for range searching, where R is the number of points actually falling in the range.*

This follows directly from elementary properties of the search structures (see Chapters 14 and 15). A balanced tree could be used, if desired.

Our goal in this chapter will be to achieve these same running times for multidimensional range searching. The parameter R can be quite significant: given the facility to make range queries, a user could easily formulate queries that could require all or nearly all of the points. This type of query could reasonably be

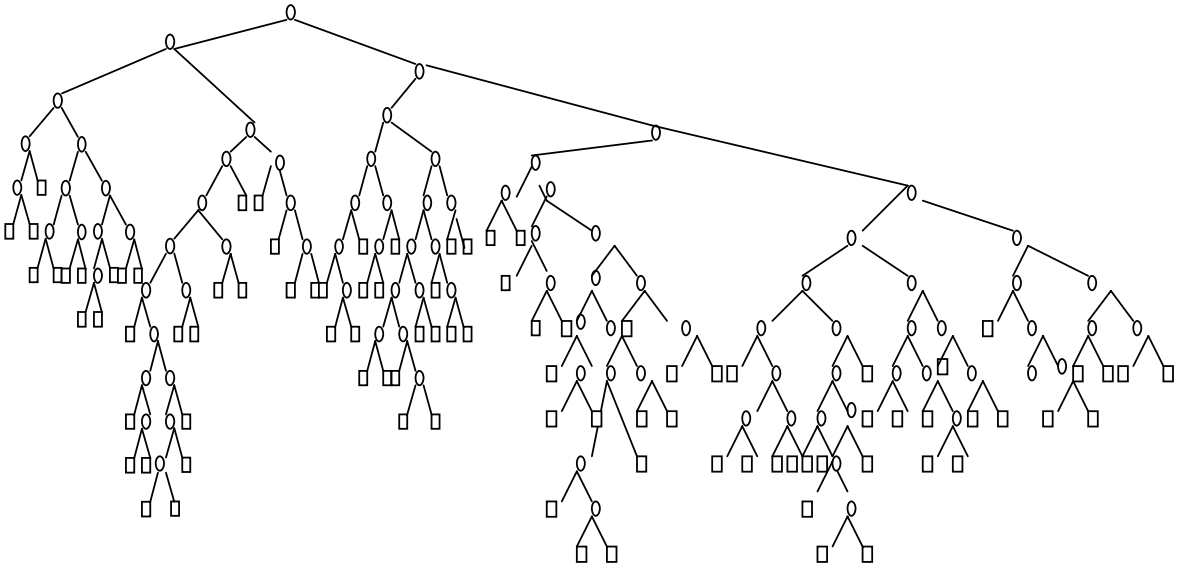


Figure 26.1 Range searching (one-dimensional) with a binary search tree.

expected to occur in many applications, but sophisticated algorithms are not necessary if all queries are of this type. The algorithms we consider are designed to be efficient for queries that are not expected to return a large number of points.

Elementary Methods

In two dimensions, our "range" is an area in the plane. For simplicity, we'll consider the problem of finding all points whose x coordinates fall within a given x-interval and whose y coordinates fall within a given y-interval: that is, we seek all points falling within a given rectangle. Thus, we'll assume a type *rectangle* which is a record of four integers, the horizontal and vertical interval endpoints. Our basic operation is to test whether a point falls within a given rectangle, so we'll assume a function *insiderect(p: point; rect: rectangle)* which checks this in the obvious way, returning *true if* p falls within rect. Our goal is to find all the points that fall within a given rectangle, using as few calls to *insiderect* as possible.

The simplest way to solve this problem is *sequential search:* scan through all the points, testing each to see if it falls within the specified range (by calling *insiderect* for each point). This method is in fact used in many database applications because it is easily improved by "batching" the range queries, testing for many different ones in the same scan through the points. In a very large database, where the data is on an external device and the time to read it is by far the dominating cost factor, this can be a very reasonable method: collect as many queries as will fit in internal memory and search for them all in one pass through the large external data file. If this type of batching is inconvenient or the database is somewhat smaller, however, there are much better methods available.

A simple first improvement to sequential search is direct application of a known one-dimensional method along one or more of the dimensions to be searched. Figure 26.2 shows an example of a search rectangle for our sample sets of points.

One way to proceed is to find the points whose x coordinates fall within the x range specified by the rectangle, then check the y coordinates of those points to determine whether or not they fall within the rectangle. Thus, points that cannot

coordinates

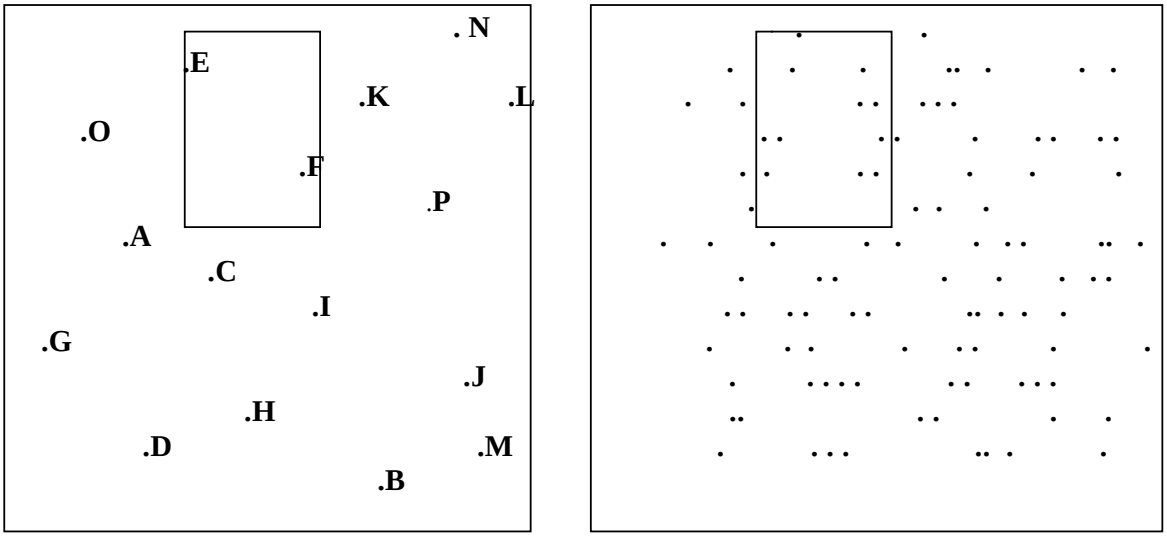


Figure 26.2 Two-dimensional range searching

be within the rectangle because their x coordinates are out of range are never examined. This technique is called *projection;* obviously we could also project on *y.* For our example, we would check E C H F and I for an x projection, as described above, and we would check O E F K P N and L for a y projection.

If the points are uniformly distributed in a rectangular region, then it's trivial to calculate the average number of points checked. The fraction of points we would expect to find in a given rectangle is simply the ratio of the area of that rectangle to the area of the full region; the fraction of points we would expect to check for an x projection is the ratio of the width of the rectangle to the width of the region, and similarly for ay projection. For our example, using a 4-by-6 rectangle in a 16-by16 region means that we would expect to find 3/32 of the points in the rectangle, 1/4 of them in an x projection, and 3/8 of them in a y projection. Obviously, under such circumstances, it's best to project onto the axis corresponding to the narrower of the two rectangle dimensions. On the other hand, it's easy to construct situations in which the projection technique could fail miserably: for example, if the point set forms an "L" shape and the search is for a range that encloses only the point at the comer of the "L," then projection on either axis eliminates only half the points.

At first glance, it seems that the projection technique could be improved somehow to "intersect" the points that fall within the x range and the points that fall within the y range. Attempts to do this without examining in the worst case either all the points in the x range or all the points in the y range serve mainly to make one appreciate the more sophisticated methods we are about to study.

Grid Method

A simple but effective technique for maintaining proximity relationships among points in the plane is to construct an artifical grid that divides the are to be searched into small squares and keep short lists of points falling into each square. (This technique is used in archeology, for example.) Then, when points lying within a given rectangle area sought, only the lists corresponding to squares that intersect the rectangle need to be searched. In our example, only E, C, F, and K are examined, as shown in Figure 26.3.

The main decision to be made in implementing this method is to determine the size of the grid: if it is too coarse, each grid square will contain too many points, and if it is too fine, there will be too many grid squares to search (most of which will be empty). One way to strike a balance between these two extremes is to choose the grid size so that the number of grid squares is a constant fraction of the total number of points. Then the number of points in each square is expected to be about equal to some small constant. For our small sample point set, using a 4 by 4 grid for a sixteen-point set means that each grid square is expected to contain one point.

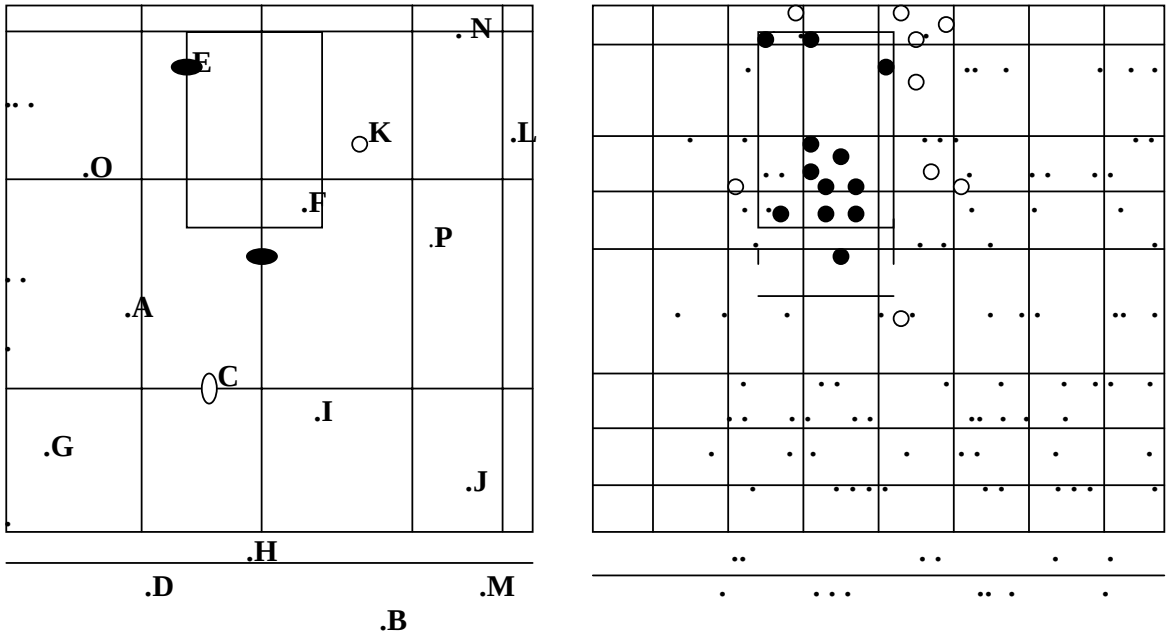


Figure 26.3 Grid method for range searching

Below is a straightforward implementation of a program to buil the grid structure containing the points in an array p[0..N] of points of the type described at the beginning of Chapter 24. The variable *size* is used to control how big the grid squares are and thus determine the resolution of the grid. Assume for simplicy that the coordinates of all the points fall between 0 and some maximum value *max*. Then *size* is taken to be the width of a grid square and there are max/size by max/size grid squares. To find which grid square a point belongs to, we divide its coordinates by *size*, as in the following implementation:

```
const maxG=20;
type link= |node;
           node=record p: point; next: link end;
var grid: array [O.. maxG,  0.. maxG ] of link;
        size: integer;
        z: link;
procedure preprocess;
    procedure insert (p: point);
        var t: link;
        begin
        new (t); t|.p:=p;
        t|.next:=grid[p.x div size.p.y div size];
       grid[p.x div size.p.y div size]:=t;
    end;
begin
new (z);
size:=l; while size*size<max*max/N do size:=size+size;
for i:=0 to maxG do
      for j:=0 to maxG do grid [i,j ]:=z;
for i: =0 to N do insert (p [i I);
end;
```

This program uses our standard linked-list representations, with dummy tail node z. Again, the variable max is assumed to be global, perhaps set to the maximum coordinate value encountered at the time the points are input.

As mentioned above, how to set the variable *size* depends on the number of points, the amount of memory available, and the range of coordinate values. Roughly, to get M points per grid square, *size* should be chosen to be the nearest integer to max divided by $\sqrt{N/M}$. This leads to about N/M grid squares. These estimates aren't accurate for small values of the parameters, but they are useful for most situations, and similar estimates can easily be formulated for specialized applications. The value need not be computed exactly-the implementation above makes *size* a power of two, which should make multiplication and division by *size* much more efficient in most programming environments.

The above implementation uses M = 1, a commonly used choice. If space is at a premium, a large value may be appropriate, but a smaller value is not likely to be useful except in specialized situations.

Now, most of the work for range searching is handled simply by indexing into thegrid array, as follows:

```
procedure gridrange (rect: rectangle);
        var t: link;
            i,j: integer;
begin
  for i:=(rect.xl div size) to (rect.x2 div size) do
    for j:=(rect.yl div size) to (rect.y2 div size) do
          begin
           t:=grid[i,j];
           while t< >z do
                begin
                if insiderect(t|.p, rect)
                    then point t|.p is within the range
                            t:=t|.next
               end
            end
  end;
```

The running time of this program is proportional to the number of grid squares touched. Since we were careful to arrange things so that each grid square contains a constant number of points on the average, the number of grid squares touched is also proportional, on the average, to the number of points examined.

Property 26.2 *The grid method for range searching is linear in the number of points in the range, on the average, and linear in the total number of points in the worst case.*

If the number of points in the search rectangle is R, then the number of grid squares examined is proportional to R. The number of grid squares examined that do not fall completely inside the search rectangle is certainly less than a small constant times R, so the total running time (on the average) is linear in R. For large R, the number of points examined that don't fall in the search rectangle gets quite small: all such points fall in a grid square that intersects the edge of the search rectangle, and the number of such squares is proportional to @R for large R. Note that this argument falls apart if the grid squares are too small (too many empty grid squares inside the search rectangle) or too large (too many points in grid squares on the perimeter of the search rectangle) or if the search rectangle is thinner than the grid squares (it could intersect many grid squares but have few points inside it). o

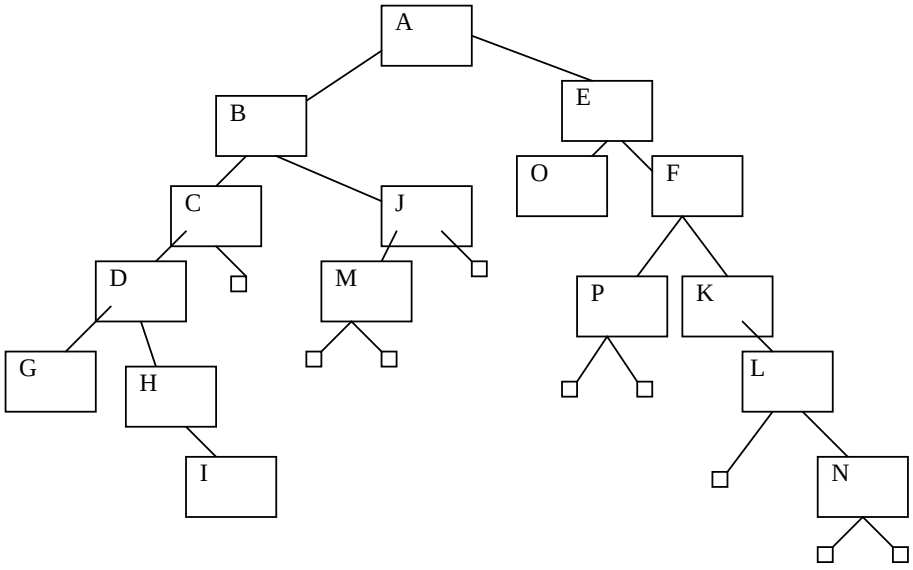


Figure 26.4 A two-dimensional (2D) tree

The grid method works well if the points are well distributed over the assumed range, but badly if they are clustered together. (For example, all the points could fall in one grid box, which would mean that all the grid machinery gained nothing.) The method we examine next makes this worst case very unlikely by subdividing the space in a nonuniform way, adapting to the point set at hand.

Two-Dimensional Trees

Two-dimensional (2D) trees are dynamic, adaptable data structures that are very similar to binary trees but divide up a geometric space in a manner convenient for use in range searching and other problems. The idea is to build binary search trees with points in the nodes, using the y and x coordinates of the points as keys in a strictly alternating sequence.

The same algorithm is used to insert points into 2D trees as in normal binary search trees, but at the root we use the y coordinate (if the point to be inserted has a smaller y coordinate than the point at the root, go left; otherwise go right), then at the next level we use the x coordinate, then at the next level the y coordinate, etc., alternating until an external node is encountered. Figure 26.4 shows the 2D tree corresponding to our small sample set of points.

The significance of this technique is that it corresponds to dividing up the plane in a simple way: all the points below the point at the root go in the left subtree, all those above in the right subtree, then all the points above the point at the root and to the left of the point in the right subtree go in the left subtree of the right subtree of the root, etc.

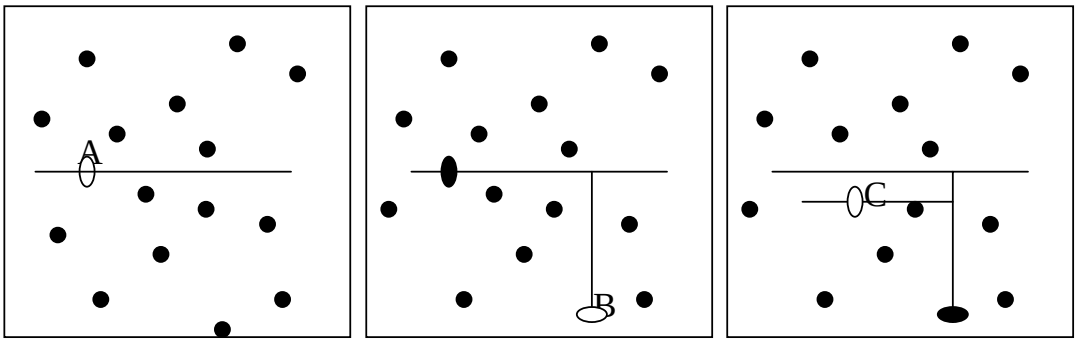


Figure 26.5 Subdividing the plane with a 2D tree; initial steps.

Figures 26.5 and 26.6 show how the plane is subdivided corresponding to the construction of the tree in Figure 26.4. First a horizontal line is drawn at the y-coordinate of A, the first node inserted. Then, since B is below A, it goes to the left of A in the tree, and the halfplane below A is divided with a vertical line at the x-coordinate of B (second diagram in Figure 26.5 ,). Then, since C is below A, we go left at the root, and since it is to the left of **B** we go left at B, and divide the portion of the plane below A and to the left of B with a horizontal line at the *y* coordinate of C (third diagram in Figure 26.5). The insertion of D is similar, then E goes to the right of A since it is above it (first diagram of Figure 26.6), etc.

Every external node of the tree corresponds to some rectangle in the plane. Each region corresponds to an external node in the tree; each point lies on a horizontal or vertical line segment that defines the division made in the tree at that point.

The code to construct of 2D trees is a straightforward modification of standard binary tree search to switch between x and y coordinates at each level.

```
type    link= |node;
            node=record p: point; l, r: link end;
var     t, head, z: link;
procedure treeinsert(p: point; t: link);
  var f: link;
        d, td: boolean;
```

```
begin
d := true;
repeat
      if d then td := p.x < t|.p.x
                  else td := p.y<t|.p.y;
                   f := t;
      if td then t := t|.l else t := t|.r;
       d := not d;
   until t = z;
   new(t); t|.p := p; t|.l := z; t|.r := z;
  if td then f|.l := t else f|.r := t;
  end;
```

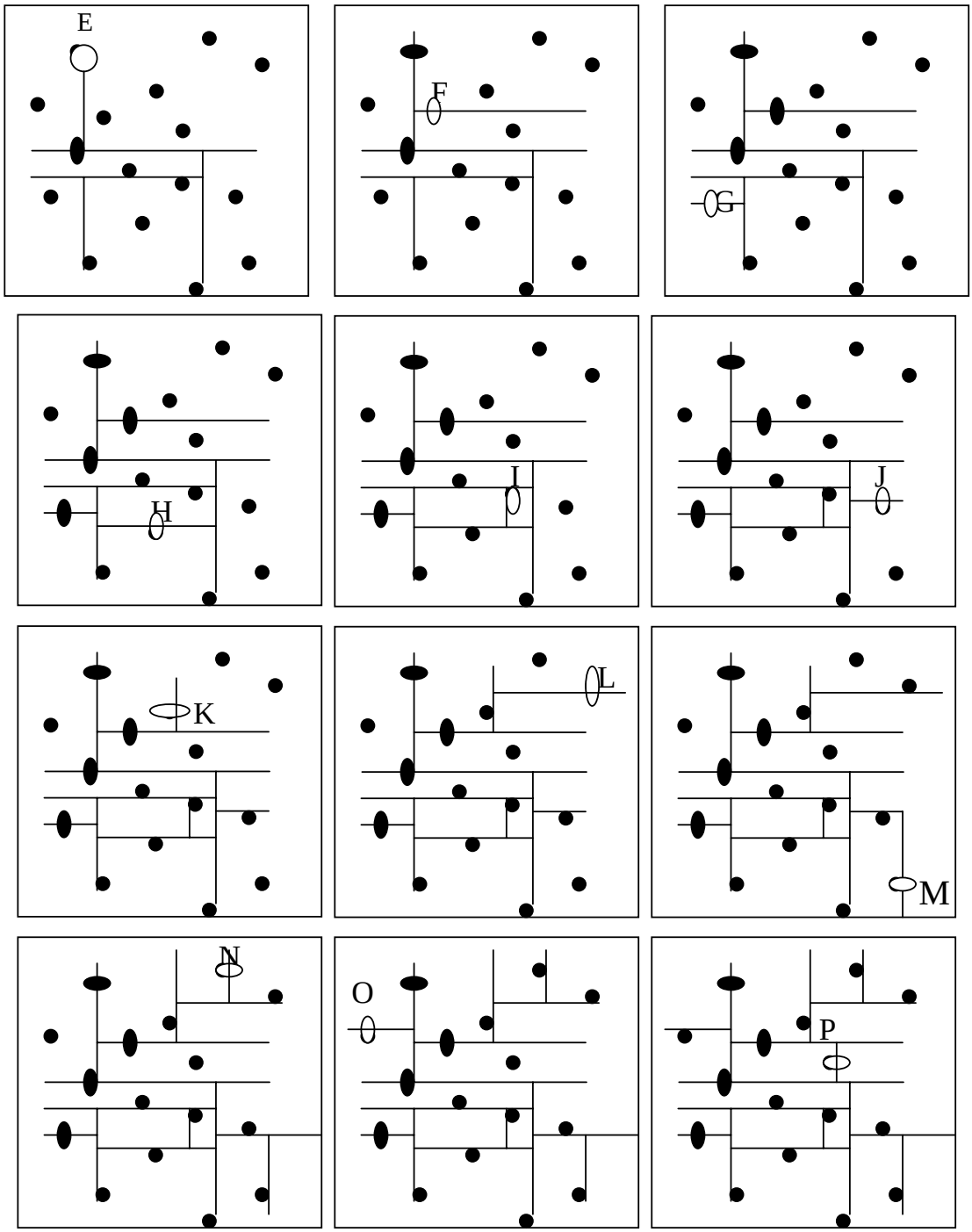


Figure 26.6 Subdividing the plane with a 2D tree; continuation.

As usual, we use a header node head with an artificial point (0, 0) which is "less than" all the other points so that the tree hangs off the right link of *head,* and an artificial node z is used to represent all the external nodes. The call *treeinsert (p,head)* inserts a new node containing p into the tree. A boolean variable d is toggled on the way down the tree to effect the alternating tests on x and y coordinates. Otherwise the procedure is identical to the standard procedure from Chapter 14.

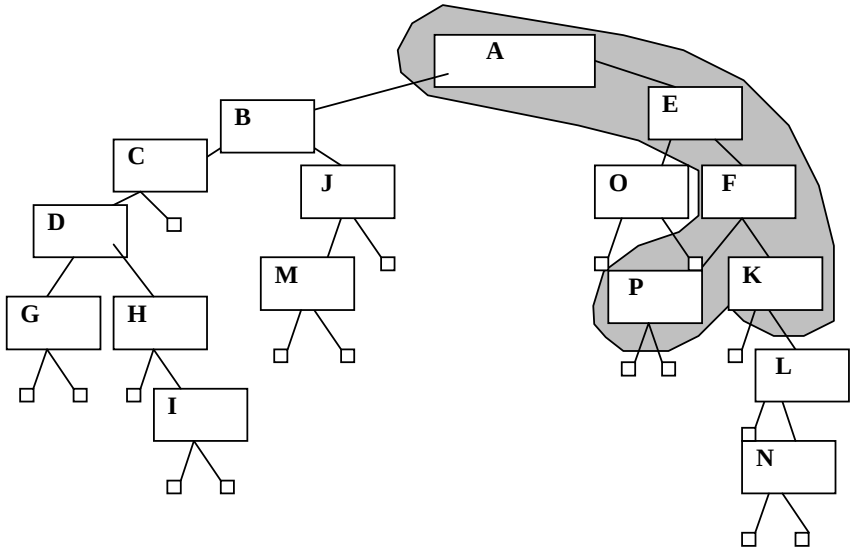


Figure 26.7 Range searching with a 2D tree

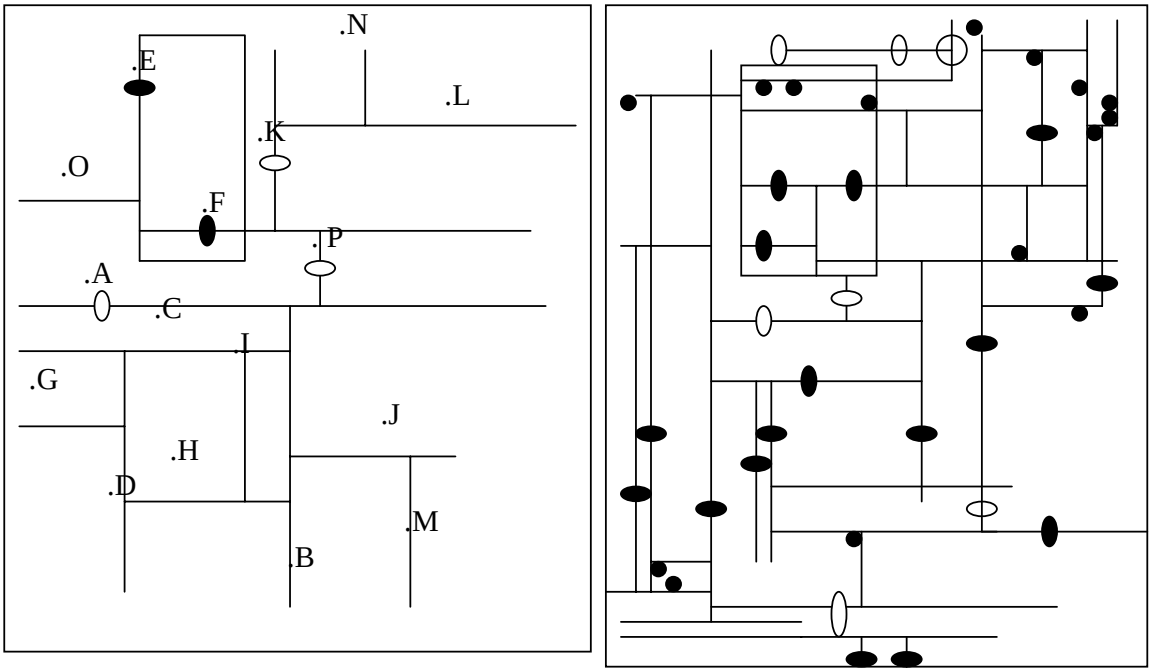
Property 26.3 *Constr (,tion of a 2D tree.from N random points requires 2N In N comparisons, on the average.*

Indeed, for randomly distributed points, 2D trees have the same performance characteristics as binary search trees. Both coordinates act as random "keys". 0

To do range searching using 2D trees, we first build the 2D tree from the points in the preprocessing phase:

```
--------------------------------------------------------------------------
procedure preprocess;
function initialize: link;
        var t: link;
       begin
      p[0].x := 0; p[0].y := 0; p[O].info := 0;
      new(t);  t|.p := p[0];  t|.r := z;
      initialize := t
      end;
begin
new(z); head := initialize;
for i: =1 to N do treeinsert (p[i], head);
end;
```

Then, for range searching, we test the point at each node against the range along the dimension used to divide the plane of that node. For our example, we begin by going right at the root and right at node E, since our search rectangle is entirely above A and to the right of E. Then, at node F, we must go down both subtrees, since F falls in the x range defined by the rectangle (note carefully that this is *not* the same as saying that F falls within the rectangle). Then the left subtrees of P and K are checked, corresponding to checking the areas of the plane that overlap the search rectangle. (See Figures 26.7 and 26.8.)



This process is easily implemented with a straightforward generalization of the 1D range procedure examined at the beginning of this chapter:

```
procedure range(t: link; rect: rectangle; d: boolean);
    var tl,  t2, txl, tx2, tyl, ty2: boolean;
    begin
    if t< >z then
            begin
            txl := rect.xl<t|.p.x;  tx2 := t|.p.x<=rect.x2;
            tyl := rect.yl<t|.p.y; ty2 := t|.p.y<=rect.y2;
             if d then begin tl := txl; t2 := tx2 end
    else begin tl := tyl; t2 := ty2 end;
            if tl then range (t|.l, rect , not d);
            if insiderect(t|.p, rect)
                    then point t|.p is in the range
            if t2 then range (t|.r , rect , not d);
             end
    end;
```

This procedure goes down both subtrees only when the dividing line cuts the rectangle, which should happen infrequently for relatively small rectangles. Figure 26.8 shows the planar subdivisions and the points examined for our two examples.

Property26.4 Range searching with a 2D tree seems to use about R+LogN steps *to find R points in reasonable ranges in a region containing N points.*

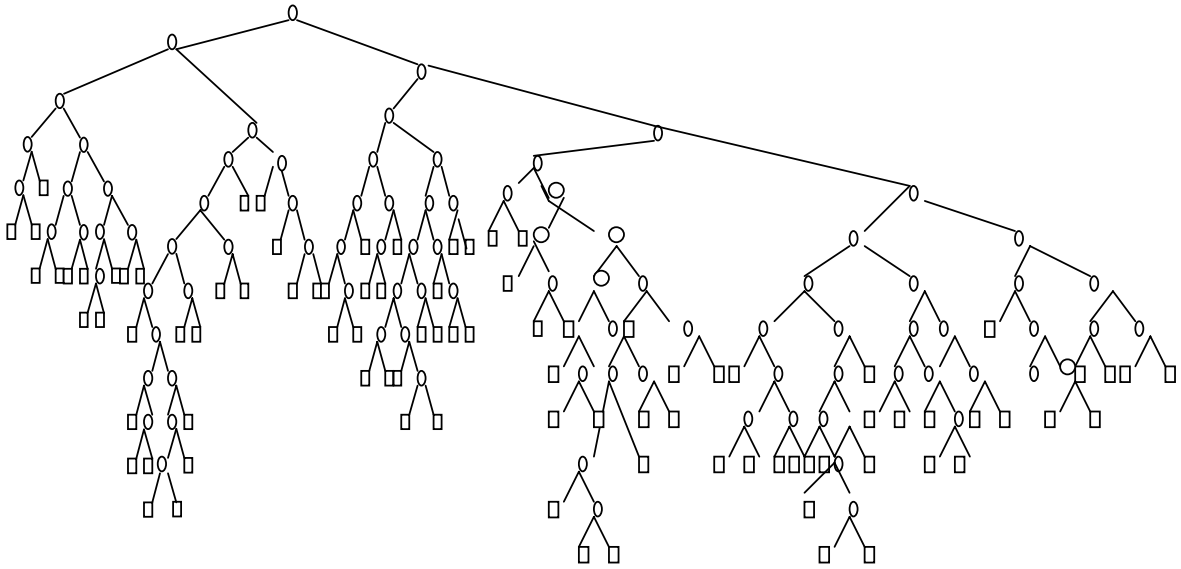


Figure 26.9 Range searching with a large 2D tree.

This method has yet to be analyzed, and the property stated is a conjecture based purely on empirical evidence. Of course, the performance (and the analysis) is as always very dependent on the type of range used. But the method is very competitive with the grid method, and somewhat less dependent on "randomness" in the point set. Figure 26.9 shows the 2D tree for our large example.

Multidimensional Range Searching

Both the grid method and 2D trees generalize directly to more than two dimensions: simple, straightforward extensions to the above algorithms immediately yield range-searching methods that work for more than two dimensions. However, the nature of multidimensional space calls for some caution and suggests that the performance characteristics of the algorithms might be difficult to predict for a particular application.

To implement the grid method for k-dimensional searching, we simply make *grid* a k-dimensional array and use one index per dimension. The main problem is to pick a reasonable value for *size*. This problem becomes quite obvious when large k is considered: what type of grid should we use for 10-dimensional search? The problem is that even if we use only three divisions per dimension, we need 310 grid squares, most of which will be empty, for reasonable values of *N*.

The generalization from 2D to *k* D trees is also straightforward: simply cycle through the dimensions (as we did for two dimensions by alternating between x and y) while going down the tree. As before, in a random situation, the resulting trees have the same characteristics as binary search trees. Also as before, there is a natural correspondence between the trees and a simple geometric process. In three dimensions, branching at each node corresponds to cutting the three-dimensional region of interest with a plane; in general we cut the k-dimensional region of interest with a *(k* -])-dimensional hyperplane.

If k is very large, there is likely to be a significant amount of imbalance in the kD trees, again because practical point sets can't be large enough to exhibit randomness over a large number of dimensions. Typically, all points in a subtree will have the same value across several dimensions, which leads to several oneway branches in the trees. One way to alleviate this problem is, rather than simply cycling through the dimensions, always to use the dimension that divides up the point set in the best way. This technique can also be applied to 2D trees. It requires that extra information (which dimension should be discriminated upon) be stored in each node, but it does relieve imbalance, especially in high-dimensional trees.

In summary, though it is easy to see how to generalize our programs for range searching to handle multidimensional problems, such a step should not be taken lightly for a large application. Large databases with many attributes per record can be very complicated objects indeed, and a good understanding of the characteristics of a database is often necessary in order to develop an efficient range-searching method for a particular application. This is a quite important problem which is still being actively studied.