whose result is shown in Figure 13.10(b). The second is a zigzag rotation, whose result is shown in Figure 13.10(c). The final step is a single rotation resulting in the tree of Figure 13.10(d). Notice that the splaying process has made the tree shallower.

13.3 Spatial Data Structures

All of the search trees discussed so far — BSTs, AVL trees, splay trees, 2-3 trees, B-trees, and tries — are designed for searching on a one-dimensional key. A typical example is an integer key, whose one-dimensional range can be visualized as a number line. These various tree structures can be viewed as dividing this one-dimensional numberline into pieces.

Some databases require support for multiple keys, that is, records can be searched based on any one of several keys. Typically, each such key has its own one-dimensional index, and any given search query searches one of these independent indices as appropriate.

A multidimensional search key presents a rather different concept. Imagine that we have a database of city records, where each city has a name and an xy-coordinate. A BST or splay tree provides good performance for searches on city name, which is a one-dimensional key. Separate BSTs could be used to index the x- and y-coordinates. This would allow us to insert and delete cities, and locate them by name or by one coordinate. However, search on one of the two coordinates is not a natural way to view search in a two-dimensional space. Another option is to combine the xy-coordinates into a single key, say by concatenating the two coordinates, and index cities by the resulting key in a BST. That would allow search by coordinate, but would not allow for efficient two-dimensional **range queries** such as searching for all cities within a given distance of a specified point. The problem is that the BST only works well for one-dimensional keys, while a coordinate is a two-dimensional key where neither dimension is more important than the other.

Multidimensional range queries are the defining feature of a **spatial application**. Because a coordinate gives a position in space, it is called a **spatial attribute**. To implement spatial applications efficiently requires the use of **spatial data structures**. Spatial data structures store data objects organized by position and are an important class of data structures used in geographic information systems, computer graphics, robotics, and many other fields.

This section presents two spatial data structures for storing point data in two or more dimensions. They are the **k-d tree** and the **PR quadtree**. The k-d tree is a

natural extension of the BST to multiple dimensions. It is a binary tree whose splitting decisions alternate among the key dimensions. Like the BST, the k-d tree uses object space decomposition. The PR quadtree uses key space decomposition and so is a form of trie. It is a binary tree only for one-dimensional keys (in which case it is a trie with a binary alphabet). For d dimensions it has 2^d branches. Thus, in two dimensions, the PR quadtree has four branches (hence the name "quadtree"), splitting space into four equal-sized quadrants at each branch. Section 13.3.3 briefly mentions two other variations on these data structures, the **bintree** and the **point quadtree**. These four structures cover all four combinations of object versus key space decomposition on the one hand, and multi-level binary versus 2^d-way branching on the other. Section 13.3.4 briefly discusses spatial data structures for storing other types of spatial data.

13.3.1 The K-D Tree

The k-d tree is a modification to the BST that allows for efficient processing of multidimensional keys. The k-d tree differs from the BST in that each level of the k-d tree makes branching decisions based on a particular search key associated with that level, called the **discriminator**. We define the discriminator at level i to be $i \bmod k$ for k dimensions. For example, assume that we store data organized by xy-coordinates. In this case, k is 2 (there are two coordinates), with the x-coordinate field arbitrarily designated key 0, and the y-coordinate field designated key 1. At each level, the discriminator alternates between x and y. Thus, a node *N* at level 0 (the root) would have in its left subtree only nodes whose x values are less than N_x (because x is search key 0, and $0 \bmod 2 = 0$). The right subtree would contain nodes whose x values are greater than N_x. A node *M* at level 1 would have in its left subtree only nodes whose y values are less than M_y. There is no restriction on the relative values of M_x and the x values of *M*'s descendants, because branching decisions made at *M* are based solely on the y coordinate. Figure 13.11 shows an example of how a collection of two-dimensional points would be stored in a k-d tree.

In Figure 13.11 the region containing the points is (arbitrarily) restricted to a 128×128 square, and each internal node splits the search space. Each split is shown by a line, vertical for nodes with x discriminators and horizontal for nodes with y discriminators. The root node splits the space into two parts; its children further subdivide the space into smaller parts. The children's split lines do not cross the root's split line. Thus, each node in the k-d tree helps to decompose the space into rectangles that show the extent of where nodes can fall in the various subtrees.

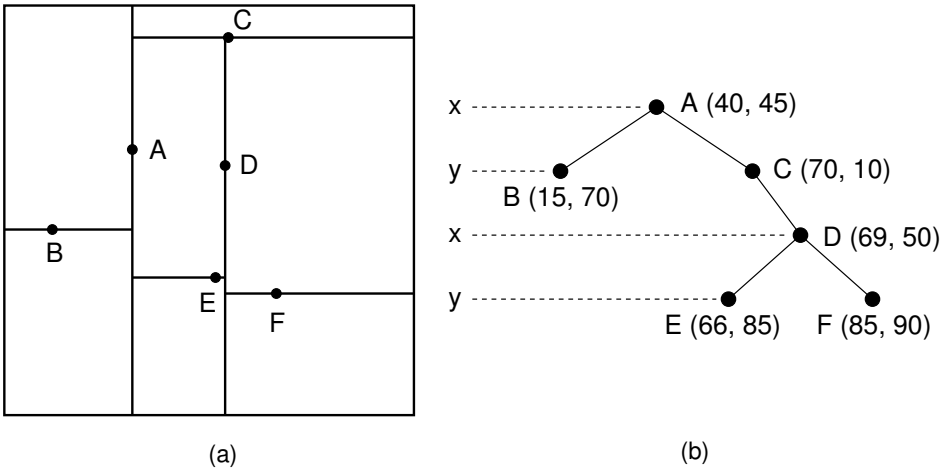

Figure 13.11 Example of a k-d tree. (a) The k-d tree decomposition for a 128×128-unit region containing seven data points. (b) The k-d tree for the region of (a).

Searching a k-d tree for the record with a specified xy-coordinate is like searching a BST, except that each level of the k-d tree is associated with a particular discriminator.

Example 13.5 Consider searching the k-d tree for a record located at $P = (69,\ 50)$. First compare P with the point stored at the root (record A in Figure 13.11). If P matches the location of A, then the search is successful. In this example the positions do not match (A's location (40, 45) is not the same as (69, 50)), so the search must continue. The x value of A is compared with that of P to determine in which direction to branch. Because A_x's value of 40 is less than P's x value of 69, we branch to the right subtree (all cities with x value greater than or equal to 40 are in the right subtree). A_y does not affect the decision on which way to branch at this level. At the second level, P does not match record C's position, so another branch must be taken. However, at this level we branch based on the relative y values of point P and record C (because $1 \bmod 2 = 1$, which corresponds to the y-coordinate). Because C_y's value of 10 is less than P_y's value of 50, we branch to the right. At this point, P is compared against the position of D. A match is made and the search is successful.

As with a BST, if the search process reaches a **`NULL`** pointer, then the search point is not contained in the tree. Here is an implementation for k-d tree search,

equivalent to the **findhelp** function of the BST class. Note that **KD** class private member **D** stores the key's dimension.

```
// Find the record with the given coordinates
bool findhelp(BinNode<Elem>* root, int* coord,
              Elem& e, int discrim) const {
  // Member "coord" of a node is an integer array storing
  // the node's coordinates.
  if (root == NULL) return false;     // Empty tree
  int* currcoord = (root->val())->coord();
  if (EqualCoord(currcoord, coord)) { // Found it
    e = root->val();
    return true;
  }
  if (currcoord[discrim] < coord[discrim])
    return findhelp(root->left(),coord,e,(discrim+1)%D);
  else
    return findhelp(root->right(),coord,e,(discrim+1)%D);
}
```

Inserting a new node into the k-d tree is similar to BST insertion. The k-d tree search procedure is followed until a **NULL** pointer is found, indicating the proper place to insert the new node.

Example 13.6 Inserting a record at location (10, 50) in the k-d tree of Figure 13.11 first requires a search to the node containing record *B*. At this point, the new record is inserted into *B*'s left subtree.

Deleting a node from a k-d tree is similar to deleting from a BST, but slightly harder. As with deleting from a BST, the first step is to find the node (call it *N*) to be deleted. It is then necessary to find a descendant of *N* which can be used to replace *N* in the tree. If *N* has no children, then *N* is replaced with a **NULL** pointer. Note that if *N* has one child that in turn has children, we cannot simply assign *N*'s parent to point to *N*'s child as would be done in the BST. To do so would change the level of all nodes in the subtree, and thus the discriminator used for a search would also change. The result is that the subtree would no longer be a k-d tree because a node's children might now violate the BST property for that discriminator.

Similar to BST deletion, the record stored in *N* should be replaced either by the record in *N*'s right subtree with the least value of *N*'s discriminator, or by the record in *N*'s left subtree with the greatest value for this discriminator. Assume that *N* was at an odd level and therefore y is the discriminator. *N* could then be replaced by the record in its right subtree with the least y value (call it $Y_{\min}$). The problem is that $Y_{\min}$ is not necessarily the leftmost node, as it would be in the BST. A modified

```
// Return a pointer to the node with the least value in root
// for the selected descriminator
BinNode<Elem>* findmin(BinNode<Elem>* root,
                       int discrim, int currdis) const {
  // discrim: discriminator key used for minimum search;
  // currdis: current level (mod D);
  if (root == NULL) return NULL;
  BinNode<Elem> *minnode = findmin(root->left(), discrim,
                                   (currdis+1)%D);
  if (discrim != currdis) { // If not at descrim's level,
                            // we must search both subtrees
    BinNode<Elem> *rightmin =
        findmin(root->right(), discrim, (currdis+1)%D);
    // Check if right side has smaller key value
    minnode = min(minnode, rightmin, discrim);
  } // Now, minnode has the smallest value in children
  return min(minnode, root, discrim);
}
```

Figure 13.12 The k-d tree **findmin** method. On levels using the minimum value's discriminator, branching is to the left. On other levels, both children's subtrees must be visited. Helper function **min** takes two nodes and a discriminator as input, and returns the node with the smaller value in that discriminator.

search procedure to find the least y value in the left subtree must be used to find it instead. The implementation for **findmin** is shown in Figure 13.12. A recursive call to the delete routine will then remove $Y_{\min}$ from the tree. Finally, $Y_{\min}$'s record is substituted for the record in node N.

Note that we can replace the node to be deleted with the least-valued node from the right subtree only if the right subtree exists. If it does not, then a suitable replacement must be found in the left subtree. Unfortunately, it is not satisfactory to replace N's record with the record having the greatest value for the discriminator in the left subtree, because this new value might be duplicated. If so, then we would have equal values for the discriminator in N's left subtree, which violates the ordering rules for the k-d tree. Fortunately, there is a simple solution to the problem. We first move the left subtree of node N to become the right subtree (i.e., we simply swap the values of N's left and right child pointers). At this point, we proceed with the normal deletion process, replacing the record of N to be deleted with the record containing the *least* value of the discriminator from what is now N's right subtree.

Assume that we want to print out a list of all records that are within a certain distance d of a given point P. We will use Euclidean distance, that is, point P is

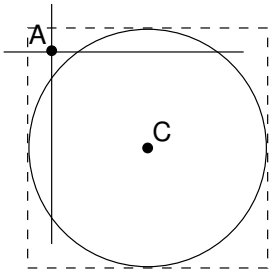

Figure 13.13 Function **InCircle** must check the Euclidean distance between a record and the query point. It is possible for a record *A* to have x- and y-coordinates each within the query distance of the query point *C*, yet have *A* itself lie outside the query circle.

defined to be within distance d of point *N* if[1]

$$\sqrt{(P_x - N_x)^2 + (P_y - N_y)^2} \leq d.$$

If the search process reaches a node whose key value for the discriminator is more than d above the corresponding value in the search key, then it is not possible that any record in the right subtree can be within distance d of the search key because all key values in that dimension are always too great. Similarly, if the current node's key value in the discriminator is d less than that for the search key value, then no record in the left subtree can be within the radius. In such cases, the subtree in question need not be searched, potentially saving much time. In the average case, the number of nodes that must be visited during a range query is linear on the number of data records that fall within the query circle.

Example 13.7 Find all cities in the k-d tree of Figure 13.14 within 25 units of the point (25, 65). The search begins with the root node, which contains record *A*. Because (40, 45) is exactly 25 units from the search point, it will be reported. The search procedure then determines which branches of the tree to take. The search circle extends to both the left and the right of *A*'s (vertical) dividing line, so both branches of the tree must be searched. The left subtree is processed first. Here, record *B* is checked and found to fall within the search circle. Because the node storing *B* has no children, processing of the left subtree is complete. Processing of *A*'s right subtree now begins. The coordinates of record *C* are checked and found not to fall within

[1] A more efficient computation is $(P_x - N_x)^2 + (P_y - N_y)^2 \leq d^2$. This avoids performing a square root function.

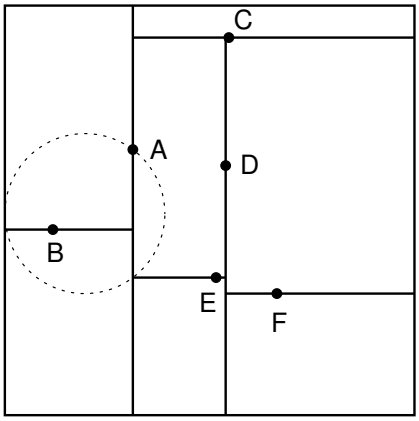

Figure 13.14 Searching in the k-d treeof Figure 13.11. (a) The k-d tree decomposition for a 128×128-unit region containing seven data points. (b) The k-d tree for the region of (a).

the circle. Thus, it should not be reported. However, it is possible that cities within *C*'s subtrees could fall within the search circle even if *C* does not. As *C* is at level 1, the discriminator at this level is the y-coordinate. Because $65 - 25 > 10$, no record in *C*'s left subtree (i.e., records above *C*) could possibly be in the search circle. Thus, *C*'s left subtree (if it had one) need not be searched. However, cities in *C*'s right subtree could fall within the circle. Thus, search proceeds to the node containing record *D*. Again, *D* is outside the search circle. Because $25 + 25 < 69$, no record in *D*'s right subtree could be within the search circle. Thus, only *D*'s left subtree need be searched. This leads to comparing record *E*'s coordinates against the search circle. Record *E* falls outside the search circle, and processing is complete. So we see that we only search subtrees whose rectangles fall within the search circle.

Figure 13.15 shows an implementation for the region search method. When a node is visited, function **`InCircle`** is used to check the Euclidean distance between the node's record and the query point. It is not enough to simply check that the differences between the x- and y-coordinates are each less than the query distances because the the record could still be outside the search circle, as illustrated by Figure 13.13.

```
// Print all points within distance "rad" of "coord"
void regionhelp(BinNode<Elem>* root, int* coord,
                int rad, int discrim) const {
  if (root == NULL) return;       // Empty tree
  // Check if record at root is in circle
  if (InCircle((root->val())->coord(), coord, rad))
    cout << root->val() << endl;  // Do what is appropriate
  int* currcoord = (root->val())->coord();
  if (currcoord[discrim] > (coord[discrim] - rad))
    regionhelp(root->left(), coord, rad, (discrim+1)%D);
  if (currcoord[discrim] < (coord[discrim] + rad))
    regionhelp(root->right(), coord, rad, (discrim+1)%D);
}
```

Figure 13.15 The k-d tree region search method.

13.3.2 The PR quadtree

In the Point-Region Quadtree (hereafter referred to as the PR quadtree) each node either has exactly four children or is a leaf. That is, the PR quadtree is a full four-way branching (4-ary) tree in shape. The PR quadtree represents a collection of data points in two dimensions by decomposing the region containing the data points into four equal quadrants, subquadrants, and so on, until no leaf node contains more than a single point. In other words, if a region contains zero or one data points, then it is represented by a PR quadtree consisting of a single leaf node. If the region contains more than a single data point, then the region is split into four equal quadrants. The corresponding PR quadtree then contains an internal node and four subtrees, each subtree representing a single quadrant of the region, which might in turn be split into subquadrants. Each internal node of a PR quadtree represents a single split of the two-dimensional region. The four quadrants of the region (or equivalently, the corresponding subtrees) are designated (in order) NW, NE, SW, and SE. Each quadrant containing more than a single point would in turn be recursively divided into subquadrants until each leaf of the corresponding PR quadtree contains at most one point.

For example, consider the region of Figure 13.16(a) and the corresponding PR quadtree in Figure 13.16(b). The decomposition process demands a fixed key range. In this example, the region is assumed to be of size 128×128. Note that the internal nodes of the PR quadtree are used solely to indicate decomposition of the region; internal nodes do not store data records. Because the decomposition lines are predetermined (i.e, key-space decomposition is used), the PR quadtree is a trie.

Search for a record matching point Q in the PR quadtree is straightforward. Beginning at the root, we continuously branch to the quadrant that contains Q until

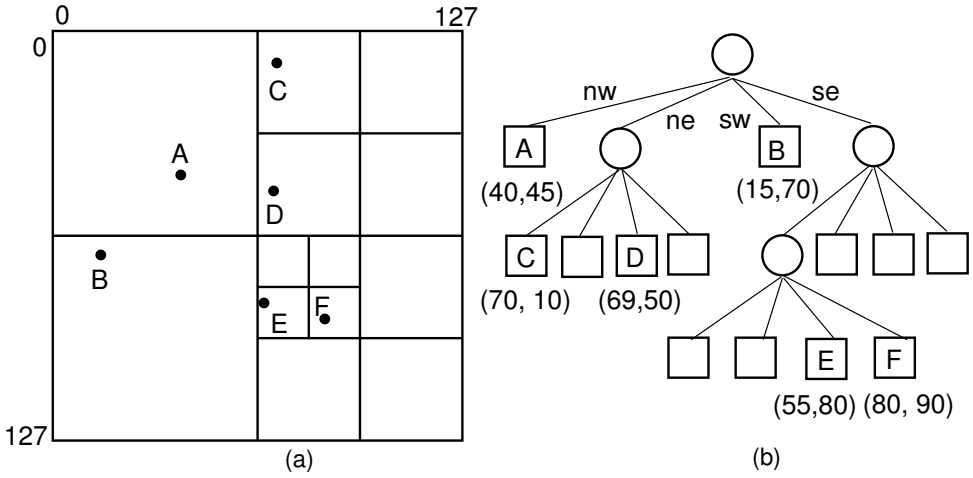

Figure 13.16 Example of a PR quadtree. (a) A map of data points. We define the region to be square with origin at the upper-left-hand corner and sides of length 128. (b) The PR quadtree for the points in (a). (a) also shows the block decomposition imposed by the PR quadtree for this region.

our search reaches a leaf node. If the root is a leaf, then just check to see if the node's data record matches point Q. If the root is an internal node, proceed to the child that contains the search coordinate. For example, the NW quadrant of Figure 13.16 contains points whose x and y values each fall in the range 0 to 63. The NE quadrant contains points whose x value falls in the range 64 to 127, and whose y value falls in the range 0 to 63. If the root's child is a leaf node, then that child is checked to see if Q has been found. If the child is another internal node, the search process continues through the tree until a leaf node is found. If this leaf node stores a record whose position matches Q then the query is successful; otherwise Q is not in the tree.

Inserting record P into the PR quadtree is performed by first locating the leaf node that contains the location of P. If this leaf node is empty, then P is stored at this leaf. If the leaf already contains P (or a record with P's coordinates), then a duplicate record should be reported. If the leaf node already contains another record, then the node must be repeatedly decomposed until the existing record and P fall into different leaf nodes. Figure 13.17 shows an example of such an insertion.

Deleting a record P is performed by first locating the node N of the PR quadtree that contains P. Node N is then changed to be empty. The next step is to look at N's three siblings. N and its siblings must be merged together to form a single node N' if only one point is contained among them. This merging process continues until some level is reached at which at least two points are contained in the subtrees rep-

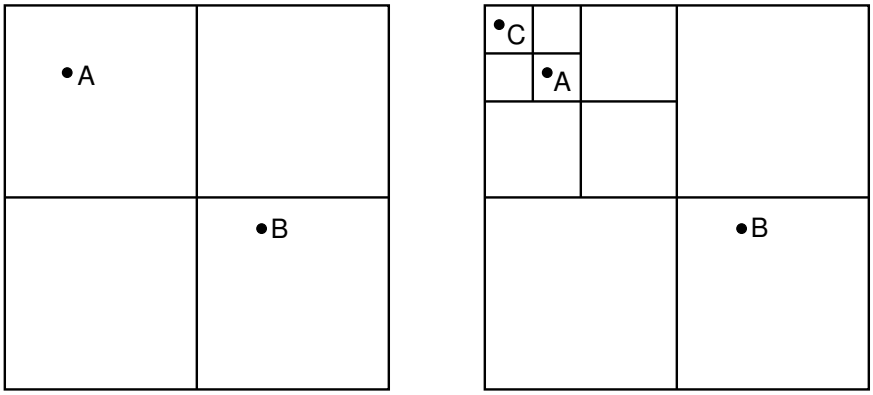

Figure 13.17 PR quadtree insertion example. (a) The initial PR quadtree containing two data points. (b) The result of inserting point *C*. The block containing *A* must be decomposed into four subblocks. Points *A* and *C* would still be in the same block if only one subdivision takes place, so a second decomposition is required to separate them.

resented by node N' and its siblings. For example, if point *C* is to be deleted from the PR quadtree representing Figure 13.17(b), the resulting node must be merged with its siblings, and that larger node again merged with its siblings to restore the PR quadtree to the decomposition of Figure 13.17(a).

Region search is easily performed with the PR quadtree. To locate all points within radius r of query point Q, begin at the root. If the root is an empty leaf node, then no data points are found. If the root is a leaf containing a data record, then the location of the data point is examined to determine if it falls within the circle. If the root is an internal node, then the process is performed recursively, but *only* on those subtrees containing some part of the search circle.

Let us now consider how structure of the PR quadtree affects the design of its node representation. The PR quadtree is actually a trie (as defined in Section 13.1). Decomposition takes place at the mid-points for internal nodes, regardless of where the data points actually fall. The placement of the data points does determine *whether* a decomposition for a node takes place, but not *where* the decomposition for the node takes place. Internal nodes of the PR quadtree are quite different from leaf nodes, in that internal nodes have children (leaf nodes do not) and leaf nodes have data fields (internal nodes do not). Thus, it is likely to be beneficial to represent internal nodes differently from leaf nodes. Finally, there is the fact that approximately half of the leaf nodes will contain no data field.

Another issue to consider is: How does a routine traversing the PR quadtree get the coordinates for the square represented by the current PR quadtree node? One possibility is to store with each node its spatial description (such as upper-left corner and width). However, this will take a lot of space — perhaps as much as the space needed for the data records, depending on what information is being stored.

Another possibility is to pass in the coordinates when the recursive call is made. For example, consider the search process. Initially, the search visits the root node of the tree, which has origin at (0, 0), and whose width is the full size of the space being covered. When the appropriate child is visited, it is a simple matter for the search routine to determine the origin for the child, and the width of the square is simply half that of the parent. Not only does passing in the size and position information for a node save considerable space, but avoiding storing such information in the nodes we enables a good design choice for empty leaf nodes, as discussed next.

How should we represent empty leaf nodes? On average, half of the leaf nodes in a PR quadtree are empty (i.e., do not store a data point). One implementation option is to use a **`NULL`** pointer in internal nodes to represent empty nodes. This will solve the problem of excessive space requirements. There is an unfortunate side effect that using a **`NULL`** pointer requires the PR quadtree processing methods to understand this convention. In other words, you are breaking encapsulation on the node representation because the tree now must know things about how the nodes are implemented. This is not too horrible for this particular application, because the node class can be considered private to the tree class, in which case the node implementation is completely invisible to the outside world. However, it is undesirable if there is another reasonable alternative.

Fortunately, there is a good alternative. It is called the Flyweight design pattern. In the PR quadtree, a flyweight is a single empty leaf node that is reused in all places where an empty leaf node is needed. You simply have *all* of the internal nodes with empty leaf children point to the same node object. This node object is created once at the beginning of the program, and is never removed. The node class recognizes from the pointer value that the flyweight is being accessed, and acts accordingly.

Note that when using the Flyweight design pattern, you *cannot* store coordinates for the node in the node. This is an example of the concept of intrinsic versus extrinsic state. Intrinsic state for an object is state information stored in the object. If you stored the coordinates for a node in the node object, those coordinates would be intrinsic state. Extrinsic state is state information about an object stored elsewhere in the environment, such as in global variables or passed to the method. If your recursive calls that process the tree pass in the coordinates for the current

node, then the coordinates will be extrinsic state. A flyweight can have in its intrinsic state *only* information that is accurate for *all* instances of the flyweight. Clearly coordinates do not qualify, because each empty leaf node has its own location. So, if you want to use a flyweight, you must pass in coordinates.

Another design choice is: Who controls the work, the node class or the tree class? For example, on an insert operation, you could have the tree class control the flow down the tree, looking at (querying) the nodes to see their type and reacting accordingly. This is the approach used by the BST implementation in Section 5.4. An alternate approach is to have the node class do the work. That is, you have an insert method for the nodes. If the node is internal, it passes the city record to the appropriate child (recursively). If the node is a flyweight, it replaces itself with a new leaf node. If the node is a full node, it replaces itself with a subtree. This is an example of the Composite design pattern, discussed in Section 5.3.1.

13.3.3 Other Point Data Structures

The differences between the k-d tree and the PR quadtree illustrate many of the design choices encountered when creating spatial data structures. The k-d tree provides an object space decomposition of the region, while the PR quadtree provides a key space decomposition (thus, it is a trie). The k-d tree stores records at all nodes, while the PR quadtree stores records only at the leaf nodes. Finally, the two trees have different structures. The k-d tree is a binary tree, while the PR quadtree is a full tree with 2^d branches (in the two-dimensional case, $2^2 = 4$). Consider the extension of this concept to three dimensions. A k-d tree for three dimensions would alternate the discriminator through the x, y, and z dimensions. The three-dimensional equivalent of the PR quadtree would be a tree with 2^3 or eight branches. Such a tree is called an **octree**.

We can also devise a binary trie based on a key space decomposition in each dimension, or a quadtree that uses the two-dimensional equivalent to an object space decomposition. The **bintree** is a binary trie that uses keyspace decomposition and alternates discriminators at each level in a manner similar to the k-d tree. The bintree for the points of Figure 13.11 is shown in Figure 13.18. Alternatively, we can use a four-way decomposition of space centered on the data points. The tree resulting from such a decomposition is called a **point quadtree**. The point quadtree for the data points of Figure 13.11 is shown in Figure 13.19.

13.3.4 Other Spatial Data Structures

This section has barely scratched the surface of the field of spatial data structures. By now dozens of distinct spatial data structures have been invented, many with

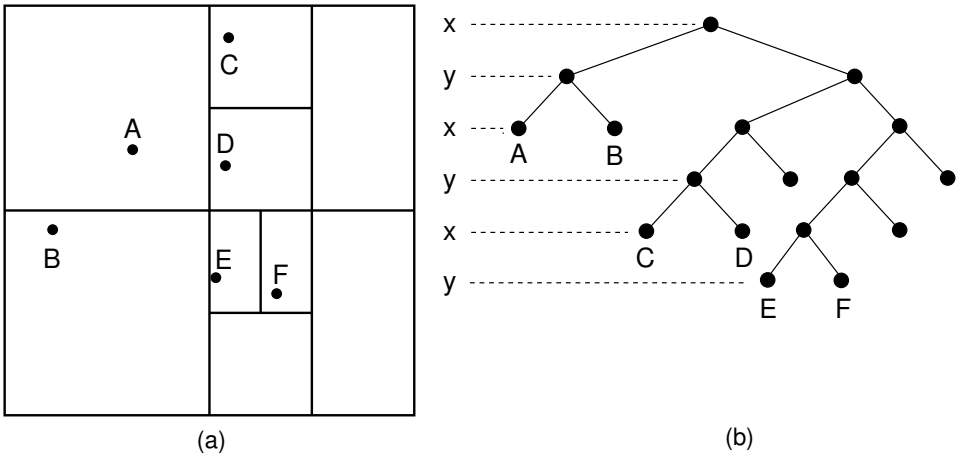

Figure 13.18 An example of the bintree, a binary tree using key space decomposition and discriminators rotating among the dimensions. Compare this with the k-d tree of Figure 13.11 and the PR quadtree of Figure 13.16.

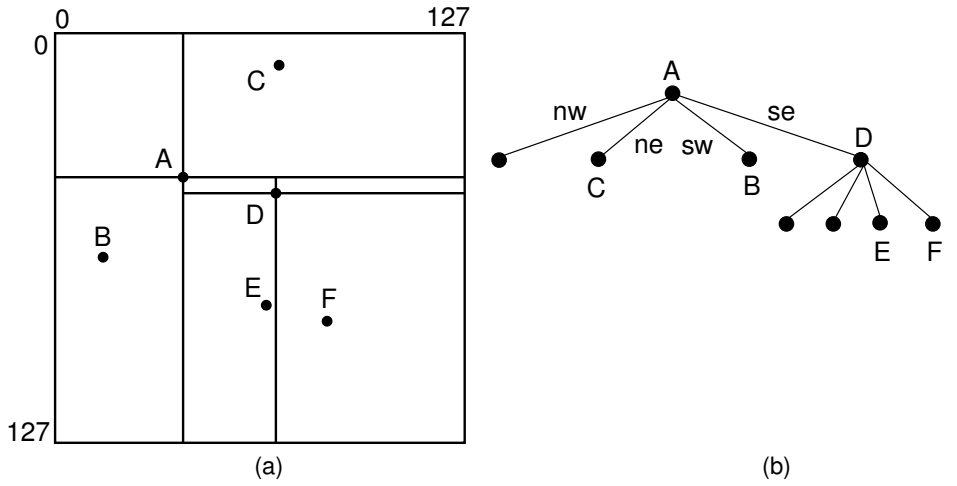

Figure 13.19 An example of the point quadtree, a 4-ary tree using object space decomposition. Compare this with the PR quadtree of Figure 13.11.

variations and alternate implementations. Spatial data structures exist for storing many forms of spatial data other than points. The most important distinctions between are the tree structure (binary or not, regular decompositions or not) and the decomposition rule used to decide when the data contained within a region is so complex that the region must be subdivided.

Perhaps the best known spatial data structure is the "region quadtree" for storing images where the pixel values tend to be blocky, such as a map of the countries of the world. The region quadtree uses a four-way regular decomposition scheme similar to the PR quadtree. The decompostion rule is simply to divide any node containing pixels of more than one color or value.

Spatial data structures can also be used to store line object, rectangle object, or objects of arbitrary shape (such as polygons in two dimensions or polyhedra in three dimensions). A simple, yet effective, data structure for storing rectangles or arbitrary polygonal shapes can be derived from the PR quadtree. Pick a threshold value c, and subdivide any region into four quadrants if it contains more than c objects. A special case must be dealt with when more than c object intersect.

Some of the most interesting developments in spatial data structures have to do with adapting them for disk-based applications. However, all such disk-based implementations boil down to storing the spatial data structure within some variant on either B-trees or hashing.

13.4 Further Reading

PATRICIA tries and other trie implementations are discussed in *Information Retrieval: Data Structures & Algorithms*, Frakes and Baeza-Yates, eds. [FBY92].

See Knuth [Knu97] for a discussion of the AVL tree. For further reading on splay trees, see "Self-adjusting Binary Search" by Sleator and Tarjan [ST85].

The world of spatial data structures is rich and rapidly evolving. For a good introduction, see the two books by Hanan Samet, *Applications of Spatial Data Structures* and *Design and Analysis of Spatial Data Structures* [Sam90a, Sam90b]. The best reference for more information on the PR quadtree is also [Sam90b]. The k-d tree was invented by John Louis Bentley. For further information on the k-d tree, in addition to [Sam90b], see [Ben75]. For information on using a quadtree to store arbitrary polygonal objects, see [SH92].

For a discussion on the relative space requirements for two-way versus multiway branching, see "A Generalized Comparison of Quadtree and Bintree Storage Requirements" by Shaffer, Juvvadi, and Heath [SJH93].