

Splaying node  $S$  involves a series of double rotations until  $S$  reaches either the root or the child of the root. Then, if necessary, a single rotation makes  $S$  the root. This process tends to rebalance the tree. In any case, it will make frequently accessed nodes stay near the top of the tree, resulting in reduced access cost. Proof that the splay tree does in fact meet the guarantee of  $O(m \log n)$  is beyond the scope of this book. For more information see the references in Section 13.4.

To illustrate how the splay tree operates, consider a search for value 89 in the splay tree of Figure 13.10(a). Searching for value 89 in the splay tree is identical to searching in a BST. However, once the value has been found, it is splayed to the root. Three rotations are required in this example. The first is a zigzig rotation, 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.

The splay tree's insert operation is the same as for the BST, except that once the node has been inserted it is splayed to the root. Likewise, deleting a node from the splay tree is identical to deleting it from the equivalent BST, except that the parent of the node deleted is then splayed to the root.

Ultimately, the only difference between the splay tree and the AVL tree is that the splay tree performs slightly different operations in certain situations. The question is: Does this minor change actually make any performance difference? The answer is that it does. The zigzig rotation provides enough improvement in the shape of the tree, as compared to AVL tree rotations, to make a difference in the overall cost over a long series of operations. A complicated analysis shows that, no matter what series of accesses (insert, delete, and find operations), the total cost for  $m$  operations will take  $O(m \log n)$  time for a tree of  $n$  nodes whenever  $m \geq n$ .

### 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. 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 index or hash table, and any given search query searches one or more 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

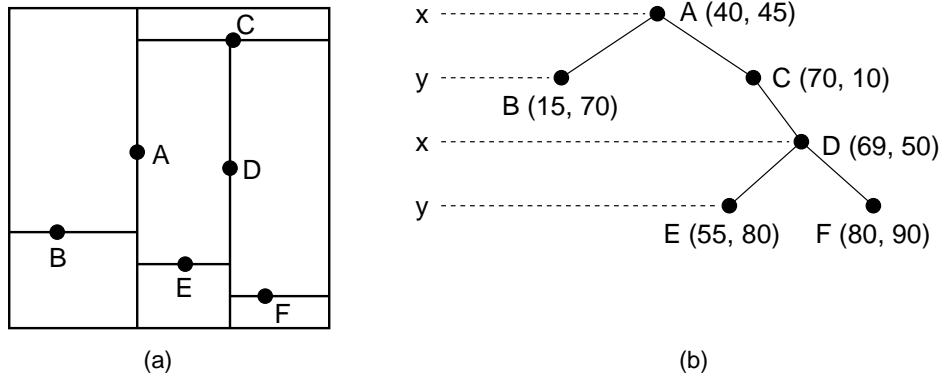
name. Separate BSTs could be used to index the  $x$ - and  $y$ -coordinates. This would allow you to insert and delete cities, and locate them by name or by one coordinate. However, search on one of the two coordinates is not a natural way to view search in a two-dimensional space. Another option is 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**. Since 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”), essentially splitting space into four equal-sized quadrants at each branch. Section 13.3.3 briefly mentions some other variations on these data structures.

### 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 for 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  $N_x$  (since  $x$  is search key 0, and  $0 \bmod 2 = 0$ ). The right subtree would contain nodes whose



**Figure 13.11** Example of a k-d tree. (a) The k-d tree decomposition for a  $100 \times 100$ -unit region containing seven data points. (b) The k-d tree for the region of (a).

$x$  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, since 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  $100 \times 100$  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 may fall in the various subtrees.

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. 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. Since  $A_x$ 's value of 40 is less than  $P$ 's 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  $C$  (since  $1 \bmod 2 = 1$ , which corresponds to the  $y$ -coordinate). Since  $C_y$ '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.

```
template <class Elem> bool KD<Elem>::
findhelp(BinNode<Elem>* subroot, int* coord,
         Elem& e, int discrim) const {
    if (subroot == NULL) return false; // Empty tree
    int* currcoord;
    currcoord = (subroot->val()).coord();
    if (EqualCoord(currcoord, coord)) { // Found it
        e = subroot->val();
        return true;
    }
    if (currcoord[discrim] < coord[discrim])
        return findhelp(subroot->left(), coord, e, (discrim+1)%D);
    else
        return findhelp(subroot->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. For example, 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 since 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

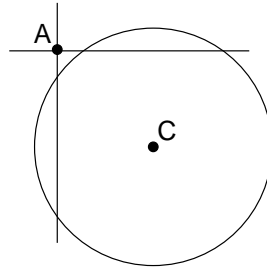
$Y_{\min}$  is not necessarily the leftmost node, as it would be in the BST. A modified search procedure to find the least  $y$  value in the left subtree must be used to find it instead. This find-minimum search can be implemented as follows:

```
template <class Elem> BinNode<Elem>* KD<Elem>::
findmin(BinNode<Elem>* subroot,
        int discrim, int currdis) const {
    // discrim: discriminator key used for minimum search;
    // currdis: current level (mod D);
    BinNode<Elem> *temp1, *temp2;
    int *coord, *tlcoord, *t2coord;
    if (subroot == NULL) return NULL;
    coord = (subroot->val()).coord();
    temp1 = findmin(subroot->left(), discrim, (currdis+1)%D);
    if (temp1 != NULL)
        tlcoord = (temp1->val()).coord();
    if (discrim != currdis) { // Min could be on either side
        temp2 = findmin(subroot->right(), discrim, (currdis+1)%D);
        if (temp2 != NULL)
            t2coord = (temp2->val()).coord();
        if ((temp1 == NULL) || ((temp2 != NULL) &&
            (t2coord[discrim] < tlcoord[discrim])))
            temp1 = temp2; // Right side has smaller key value
    } // Now, temp1 has the smallest value in children
    if ((temp1 == NULL) || (coord[discrim] < tlcoord[discrim]))
        return subroot;
    else return temp1;
}
```

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 13.12** 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

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

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 since 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 general, 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.

For example, assume that a search will be made to find all cities in the  $k$ - $d$  tree of Figure 13.11 within 25 units of the point  $(25, 65)$ . The search begins with the root node, which contains record  $A$ . Since  $(40, 45)$  is exactly 25 units from the search point, it should 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. Since 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 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. Since  $65 - 25 > 10$ , 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

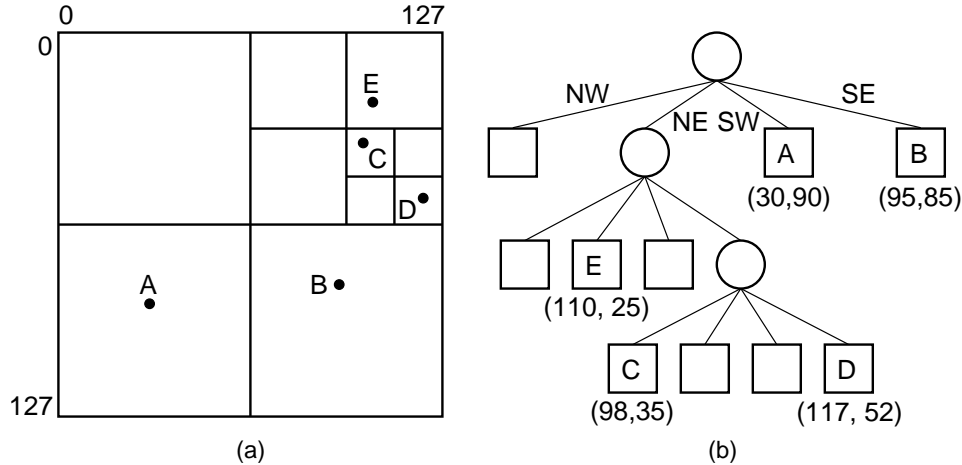
one) need not be searched. However, cities in  $C$ 's right subtree could fall within the circle. Thus, search proceeds to the node containing record  $D$ . Again,  $D$  is outside the search circle. Since  $25 + 25 < 69$ , no record in  $D$ 's right subtree could be within the search circle. Thus, only  $D$ '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. Here is an implementation for the region search function:

```
template <class Elem> void KD<Elem>::
regionhelp(BinNode<Elem>* subroot, int* coord,
           int rad, int discrim) const {
    // Check if record at subroot is in circle
    if (InCircle((subroot->val()).coord(), coord, D, rad))
        printout(subroot->val()); // Do what is appropriate
    int* currcoord = (subroot->val()).coord();
    if (currcoord[discrim] > (coord[discrim] - rad))
        regionhelp(subroot->left(), coord, rad, (discrim+1)%D);
    if (currcoord[discrim] < (coord[discrim] + rad))
        regionhelp(subroot->right(), coord, rad, (discrim+1)%D);
}
```

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 13.12.

### 13.3.2 The PR quadtree

The Point-Region Quadtree (hereafter referred to as the PR quadtree) is a tree structure where each node either has exactly four children or is a leaf, that is, it 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 may 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,



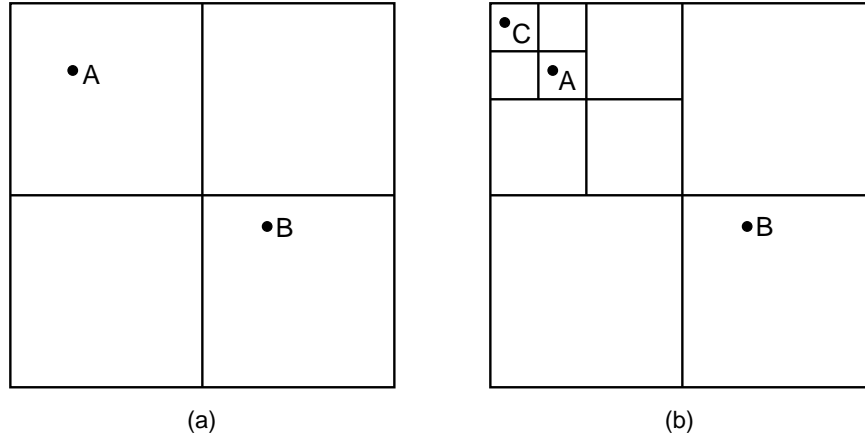
**Figure 13.13** Example of a PR quadtree. (a) A map of data points. We define the region to be square with origin at the upper-left-hand corner and sides of length 1024. (b) The PR quadtree for the points in (a). (a) also shows the block decomposition imposed by the PR quadtree for this region.

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 13.13(a) and the corresponding PR quadtree in Figure 13.13(b). The decomposition process demands a fixed key range. In this example, the region is assumed to be of size  $128 \times 128$ . 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.

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 it 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 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



**Figure 13.14** 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.

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 13.14 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 represented by node *N'* and its siblings. For example, if point *C* is to be deleted from the PR quadtree representing Figure 13.14(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 13.14(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 the structure of the PR quadtree, and the resulting impact this structure can have on design of the node representation and algorithm imple-

mentation. The PR quadtree is actually a trie, as described 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 distinct 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 – roughly 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 a search command. Initially, the search visits the root node of the tree, which has origin at  $(0, 0)$ , and whose width has 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 by avoiding storing such information in the nodes we will be free to implement a good design choice for empty leaf nodes, as discussed next.

How should we represent empty leaf nodes? In the average case, half of the leaf nodes in a PR quadtree are empty (i.e., do not store a data point). One implementation alternative 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, since 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, the flyweight is simply 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.

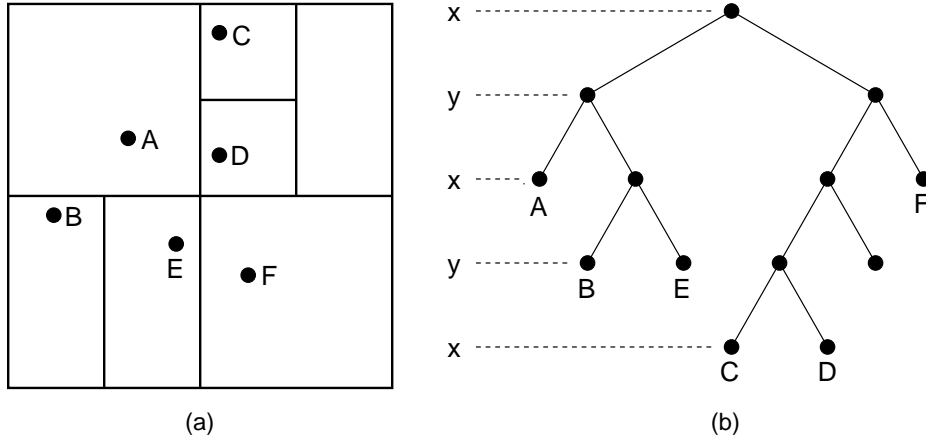
Note that with the flyweight design, you *cannot* store coordinates for the node in the node. This is an example of the concept of intrinsic versus extrinsic state. Intrinsic state for an object is state information stored in the object. If you stored the coordinates for a node in the node object, those 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* that information that is true for *all* instances of the flyweight. Clearly coordinates do not qualify, since 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 operations 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, also discussed in Section 5.3.1.

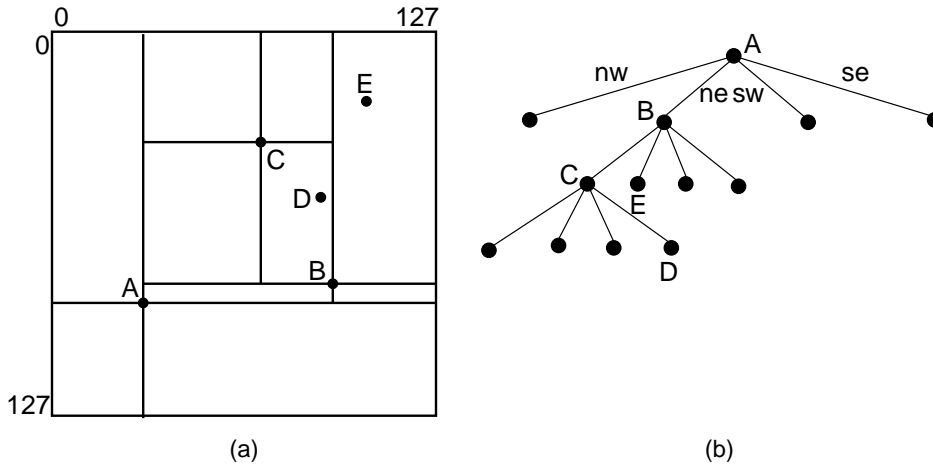
### 13.3.3 Other Spatial Data Structures

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 tree structures are distinct in their structure. The k-d tree is a binary tree, while the PR quadtree has  $2^d$  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**.

Following on these concepts, it is also possible to devise a binary trie based on a key space decomposition in each dimension, or a quadtree that uses the two-



**Figure 13.15** 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.



**Figure 13.16** An example of the point quadtree, a 4-ary tree using object space decomposition. Compare this with the PR quadtree of Figure 13.13.

dimensional equivalent to an object space decomposition. The **bintree** is a binary trie that 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 13.15. 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 13.13 is shown in Figure 13.16.

This section has only scratched the surface of the field of spatial data structures. By now dozens of distinct spatial data structures have been invented, many with variations and alternate implementations. Some of the most interesting developments have to do with adapting spatial data structures for disk-based applications. It is important to note that all such disk-based implementations boil down to variants 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 [Knu73] 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 a discussion on the relative space requirements for two-way versus multi-way branching, see “A Generalized Comparison of Quadtree and Bintree Storage Requirements” by Shaffer, Juvvadi, and Heath [SJH93].

## 13.5 Exercises

- 13.1 Show the binary trie (as illustrated by Figure 13.1) for the following collection of values: 42, 12, 100, 10, 50, 31, 7, 11, 99.
- 13.2 Show the PAT trie (as illustrated by Figure 13.3) for the following collection of values: 42, 12, 100, 10, 50, 31, 7, 11, 99.
- 13.3 Write the insertion routine for a trie.
- 13.4 Write the deletion routine for a trie.
- 13.5 Show the splay tree that results from searching for value 75 in the splay tree of Figure 13.10(d).
- 13.6 Show the splay tree that results from searching for value 18 in the splay tree of Figure 13.10(d).
- 13.7 Show the k-d tree for the points of Figure 13.13, inserted in alphabetical order.
- 13.8 Show the PR quadtree for the points of Figure 13.11, inserted in alphabetical order.