13

Advanced Tree Structures

This chapter introduces several tree structures designed for use in specialized applications. The trie of Section 13.1 is commonly used to store and retrieve strings. It also serves to illustrate the concept of a key space decomposition. The AVL tree and splay tree of Section 13.2 are variants on the BST. They are examples of self-balancing search trees and have guaranteed good performance regardless of the insertion order for records. An introduction to several spatial data structures used to organize point data by xy-coordinates is presented in Section 13.3.

Descriptions of the fundamental operations are given for each data structure. One purpose for this chapter is to provide opportunities for class programming projects, so detailed implementations are left to the reader.

13.1 Tries

Recall that the shape of a BST is determined by the order in which its data records are inserted. One permutation of the records might yield a balanced tree while another might yield an unbalanced tree, with the extreme case becoming the shape of a linked list. The reason is that the value of the key stored in the root node splits the key range into two parts: those key values less than the root's key value, and those key values greater than the root's key value. Depending on the relationship between the root node's key value and the distribution of the key values for the other records in the the tree, the resulting BST might be balanced or unbalanced. Thus, the BST is an example of a data structure whose organization is based on an **object space decomposition**, so called because the decomposition of the key range is driven by the objects (i.e., the key values of the data records) stored in the tree.

The alternative to object space decomposition is to predefine the splitting position within the key range for each node in the tree. In other words, the root could be predefined to split the key range into two equal halves, regardless of the particular values or order of insertion for the data records. Those records with keys in the lower half of the key range will be stored in the left subtree, while those records

with keys in the upper half of the key range will be stored in the right subtree. While such a decomposition rule will not necessarily result in a balanced tree (the tree will be unbalanced if the records are not well distributed within the key range), at least the shape of the tree will not depend on the order of key insertion. Furthermore, the depth of the tree will be limited by the resolution of the key range; that is, the depth of the tree can never be greater than the number of bits required to store a key value. For example, if the keys are integers in the range 0 to 1023, then the resolution for the key is ten bits. Thus, two keys can be identical only until the tenth bit. In the worst case, two keys will follow the same path in the tree only until the tenth branch. As a result, the tree will never be more than ten levels deep. In contrast, a BST containing n records could be as much as n levels deep.

Splitting based on predetermined subdivisions of the key range is called **key space decomposition**. In computer graphics, the technique is known as **image space** decomposition, and this term is sometimes used to describe the process for data structures as well. A data structure based on key space decomposition is called a **trie**. Folklore has it that "trie" comes from "retrieval." Unfortunately, that would imply that the word is pronounced "tree," which would lead to confusion with regular use of the word "tree." "Trie" is actually pronounced as "try."

Like the B^+-tree, a trie stores data records only in leaf nodes. Internal nodes serve as placeholders to direct the search process. but since the split points are predetermined, internal nodes need not store "traffic-directing" key values. Figure 13.1 illustrates the trie concept. Upper and lower bounds must be imposed on the key values so that we can compute the middle of the key range. Because the largest value inserted in this example is 120, a range from 0 to 127 is assumed, as 128 is the smallest power of two greater than 120. The binary value of the key determines whether to select the left or right branch at any given point during the search. The most significant bit determines the branch direction at the root. Figure 13.1 shows a **binary trie**, so called because in this example the trie structure is based on the value of the key interpreted as a binary number, which results in a binary tree.

The Huffman coding tree of Section 5.6 is another example of a binary trie. All data values in the Huffman tree are at the leaves, and each branch splits the range of possible letter codes in half. The Huffman codes are actually reconstructed from the letter positions within the trie.

These are examples of binary tries, but tries can be built with any branching factor. Normally the branching factor is determined by the alphabet used. For binary numbers, the alphabet is $\{0, 1\}$ and a binary trie results. Other alphabets lead to other branching factors.

One application for tries is to store a dictionary of words. Such a trie will be referred to as an **alphabet trie**. For simplicity, our examples will ignore case in letters. We add a special character ($) to the 26 standard English letters. The $ character is used to represent the end of a string. Thus, the branching factor for

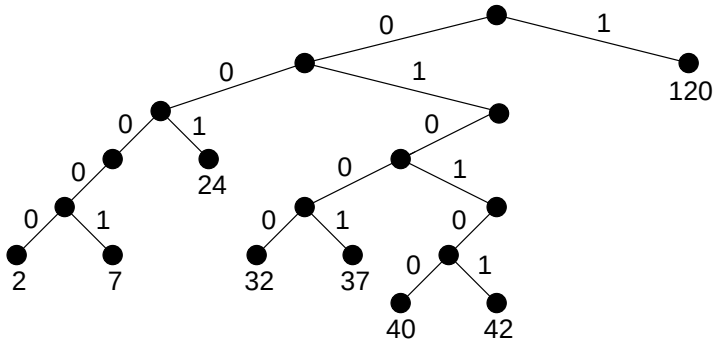

Figure 13.1 The binary trie for the collection of values 2, 7, 24, 31, 37, 40, 42, 120. All data values are stored in the leaf nodes. Edges are labeled with the value of the bit used to determine the branching direction of each node. The binary form of the key value determines the path to the record, assuming that each key is represented as a 7-bit value representing a number in the range 0 to 127.

each node is (up to) 27. Once constructed, the alphabet trie is used to determine if a given word is in the dictionary. Consider searching for a word in the alphabet trie of Figure 13.2. The first letter of the search word determines which branch to take from the root, the second letter determines which branch to take at the next level, and so on. Only the letters that lead to a word are shown as branches. In Figure 13.2(b) the leaf nodes of the trie store a copy of the actual words, while in Figure 13.2(a) the word is built up from the letters associated with each branch.

One way to implement a node of the alphabet trie is as an array of 27 pointers indexed by letter. Because most nodes have branches to only a small fraction of the possible letters in the alphabet, an alternate implementation is to use a linked list of pointers to the child nodes, as in Figure 6.9.

The depth of a leaf node in the alphabet trie of Figure 13.2(b) has little to do with the number of nodes in the trie, or even with the length of the corresponding string. Rather, a node's depth depends on the number of characters required to distinguish this node's word from any other. For example, if the words "anteater" and "antelope" are both stored in the trie, it is not until the fifth letter that the two words can be distinguished. Thus, these words must be stored at least as deep as level five. In general, the limiting factor on the depth of nodes in the alphabet trie is the length of the words stored.

Poor balance and clumping can result when certain prefixes are heavily used. For example, an alphabet trie storing the common words in the English language would have many words in the "th" branch of the tree, but none in the "zq" branch.

Any multiway branching trie can be replaced with a binary trie by replacing the original trie's alphabet with an equivalent binary code. Alternatively, we can use the techniques of Section 6.3.4 for converting a general tree to a binary tree without modifying the alphabet.

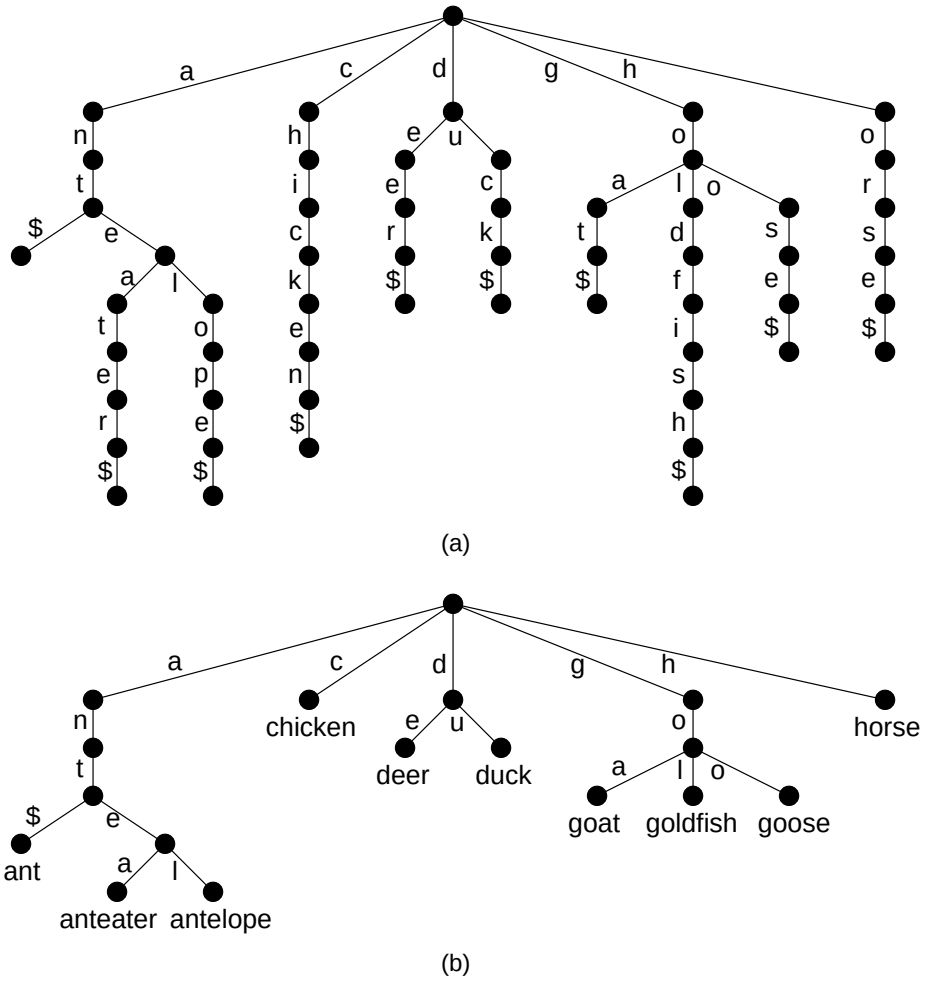

Figure 13.2 Two variations on the alphabet trie representation for a set of ten words. (a) Each node contains a set of links corresponding to single letters, and each letter in the set of words has a corresponding link. "$" is used to indicate the end of a word. Internal nodes direct the search and also spell out the word one letter per link. The word need not be stored explicitly. "$" is needed to recognize the existence of words that are prefixes to other words, such as 'ant' in this example. (b) Here the trie extends only far enough to discriminate between the words. Leaf nodes of the trie each store a complete word; internal nodes merely direct the search.

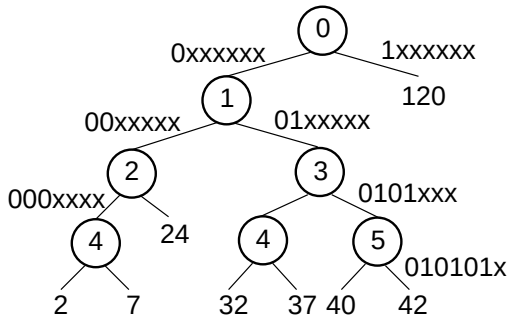

Figure 13.3 The PAT trie for the collection of values 2, 7, 24, 32, 37, 40, 42, 120. Contrast this with the binary trie of Figure 13.1. In the PAT trie, all data values are stored in the leaf nodes, while internal nodes store the bit position used to determine the branching decision, assuming that each key is represented as a 7-bit value representing a number in the range 0 to 127. Some of the branches in this PAT trie have been labeled to indicate the binary representation for all values in that subtree. For example, all values in the left subtree of the node labeled 0 must have value 0xxxxxx (where x means that bit can be either a 0 or a 1). All nodes in the right subtree of the node labeled 3 must have value 0101xxx. However, we can skip branching on bit 2 for this subtree because all values currently stored have a value of 0 for that bit.

The trie implementations illustrated by Figures 13.1 and 13.2 are potentially quite inefficient as certain key sets might lead to a large number of nodes with only a single child. A variant on trie implementation is known as PATRICIA, which stands for "Practical Algorithm To Retrieve Information Coded In Alphanumeric." In the case of a binary alphabet, a PATRICIA trie (referred to hereafter as a PAT trie) is a full binary tree that stores data records in the leaf nodes. Internal nodes store only the position within the key's bit pattern that is used to decide on the next branching point. In this way, internal nodes with single children (equivalently, bit positions within the key that do not distinguish any of the keys within the current subtree) are eliminated. A PAT trie corresponding to the values of Figure 13.1 is shown in Figure 13.3.

Example 13.1 When searching for the value 7 (0000111 in binary) in the PAT trie of Figure 13.3, the root node indicates that bit position 0 (the leftmost bit) is checked first. Because the 0th bit for value 7 is 0, take the left branch. At level 1, branch depending on the value of bit 1, which again is 0. At level 2, branch depending on the value of bit 2, which again is 0. At level 3, the index stored in the node is 4. This means that bit 4 of the key is checked next. (The value of bit 3 is irrelevant, because all values stored in that subtree have the same value at bit position 3.) Thus, the single branch that extends from the equivalent node in Figure 13.1 is just skipped. For key value 7, bit 4 has value 1, so the rightmost branch is taken. Because

this leads to a leaf node, the search key is compared against the key stored in that node. If they match, then the desired record has been found.

Note that during the search process, only a single bit of the search key is compared at each internal node. This is significant, because the search key could be quite large. Search in the PAT trie requires only a single full-key comparison, which takes place once a leaf node has been reached.

Example 13.2 Consider the situation where we need to store a library of DNA sequences. A DNA sequence is a series of letters, usually many thousands of characters long, with the string coming from an alphabet of only four letters that stand for the four amino acids making up a DNA strand. Similar DNA sequences might have long sections of their string that are identical. The PAT trie would avoid making multiple full key comparisons when searching for a specific sequence.

13.2 Balanced Trees

We have noted several times that the BST has a high risk of becoming unbalanced, resulting in excessively expensive search and update operations. One solution to this problem is to adopt another search tree structure such as the 2-3 tree or the binary trie. An alternative is to modify the BST access functions in some way to guarantee that the tree performs well. This is an appealing concept, and it works well for heaps, whose access functions maintain the heap in the shape of a complete binary tree. Unfortunately, requiring that the BST always be in the shape of a complete binary tree requires excessive modification to the tree during update, as discussed in Section 10.3.

If we are willing to weaken the balance requirements, we can come up with alternative update routines that perform well both in terms of cost for the update and in balance for the resulting tree structure. The AVL tree works in this way, using insertion and deletion routines altered from those of the BST to ensure that, for every node, the depths of the left and right subtrees differ by at most one. The AVL tree is described in Section 13.2.1.

A different approach to improving the performance of the BST is to not require that the tree always be balanced, but rather to expend some effort toward making the BST more balanced every time it is accessed. This is a little like the idea of path compression used by the UNION/FIND algorithm presented in Section 6.2. One example of such a compromise is called the **splay tree**. The splay tree is described in Section 13.2.2.

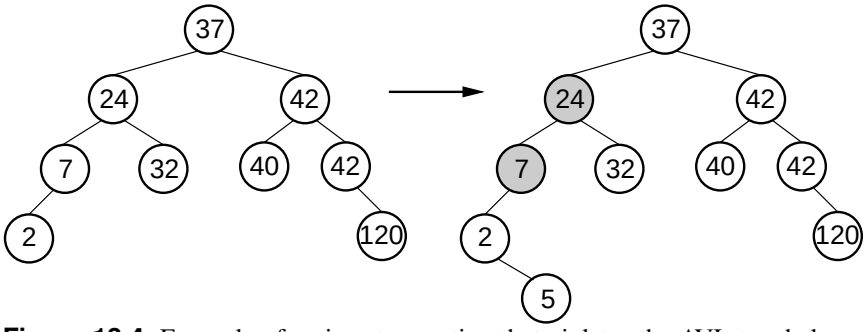

Figure 13.4 Example of an insert operation that violates the AVL tree balance property. Prior to the insert operation, all nodes of the tree are balanced (i.e., the depths of the left and right subtrees for every node differ by at most one). After inserting the node with value 5, the nodes with values 7 and 24 are no longer balanced.

13.2.1 The AVL Tree

The AVL tree (named for its inventors Adelson-Velskii and Landis) should be viewed as a BST with the following additional property: For every node, the heights of its left and right subtrees differ by at most 1. As long as the tree maintains this property, if the tree contains n nodes, then it has a depth of at most $\mathrm{O}(\log n)$. As a result, search for any node will cost $\mathrm{O}(\log n)$, and if the updates can be done in time proportional to the depth of the node inserted or deleted, then updates will also cost $\mathrm{O}(\log n)$, even in the worst case.

The key to making the AVL tree work is to alter the insert and delete routines so as to maintain the balance property. Of course, to be practical, we must be able to implement the revised update routines in $\Theta(\log n)$ time.

Consider what happens when we insert a node with key value 5, as shown in Figure 13.4. The tree on the left meets the AVL tree balance requirements. After the insertion, two nodes no longer meet the requirements. Because the original tree met the balance requirement, nodes in the new tree can only be unbalanced by a difference of at most 2 in the subtrees. For the bottommost unbalanced node, call it *S*, there are 4 cases:

1. The extra node is in the left child of the left child of *S*.
2. The extra node is in the right child of the left child of *S*.
3. The extra node is in the left child of the right child of *S*.
4. The extra node is in the right child of the right child of *S*.

Cases 1 and 4 are symmetrical, as are cases 2 and 3. Note also that the unbalanced nodes must be on the path from the root to the newly inserted node.

Our problem now is how to balance the tree in $\mathrm{O}(\log n)$ time. It turns out that we can do this using a series of local operations known as **rotations**. Cases 1 and

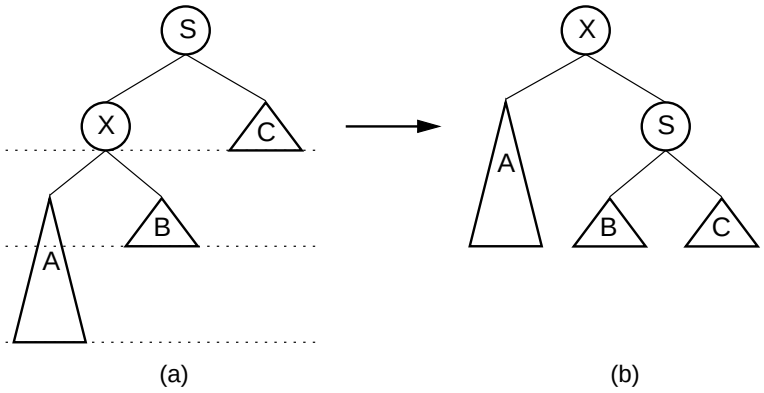

Figure 13.5 A single rotation in an AVL tree. This operation occurs when the excess node (in subtree *A*) is in the left child of the left child of the unbalanced node labeled *S*. By rearranging the nodes as shown, we preserve the BST property, as well as re-balance the tree to preserve the AVL tree balance property. The case where the excess node is in the right child of the right child of the unbalanced node is handled in the same way.

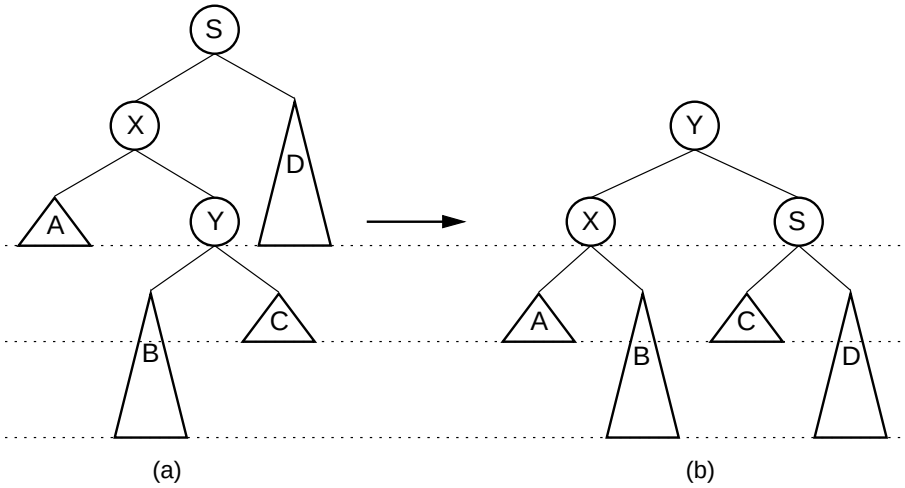

Figure 13.6 A double rotation in an AVL tree. This operation occurs when the excess node (in subtree *B*) is in the right child of the left child of the unbalanced node labeled *S*. By rearranging the nodes as shown, we preserve the BST property, as well as re-balance the tree to preserve the AVL tree balance property. The case where the excess node is in the left child of the right child of *S* is handled in the same way.

4 can be fixed using a **single rotation**, as shown in Figure 13.5. Cases 2 and 3 can be fixed using a **double rotation**, as shown in Figure 13.6.

The AVL tree insert algorithm begins with a normal BST insert. Then as the recursion unwinds up the tree, we perform the appropriate rotation on any node

that is found to be unbalanced. Deletion is similar; however, consideration for unbalanced nodes must begin at the level of the **`deletemin`** operation.

Example 13.3 In Figure 13.4 (b), the bottom-most unbalanced node has value 7. The excess node (with value 5) is in the right subtree of the left child of 7, so we have an example of Case 2. This requires a double rotation to fix. After the rotation, 5 becomes the left child of 24, 2 becomes the left child of 5, and 7 becomes the right child of 5.

13.2.2 The Splay Tree

Like the AVL tree, the splay tree is not actually a distinct data structure, but rather reimplements the BST insert, delete, and search methods to improve the performance of a BST. The goal of these revised methods is to provide guarantees on the time required by a series of operations, thereby avoiding the worst-case linear time behavior of standard BST operations. No single operation in the splay tree is guaranteed to be efficient. Instead, the splay tree access rules guarantee that a series of m operations will take O($m \log n$) time for a tree of n nodes whenever $m \geq n$. Thus, a single insert or search operation could take $\mathrm{O}(n)$ time. However, m such operations are guaranteed to require a total of $O(m \log n)$ time, for an average cost of $\mathrm{O}(\log n)$ per access operation. This is a desirable performance guarantee for any search-tree structure.

Unlike the AVL tree, the splay tree is not guaranteed to be height balanced. What is guaranteed is that the total cost of the entire series of accesses will be cheap. Ultimately, it is the cost of the series of operations that matters, not whether the tree is balanced. Maintaining balance is really done only for the sake of reaching this time efficiency goal.

The splay tree access functions operate in a manner reminiscent of the move-to-front rule for self-organizing lists from Section 9.2, and of the path compression technique for managing parent-pointer trees from Section 6.2. These access functions tend to make the tree more balanced, but an individual access will not necessarily result in a more balanced tree.

Whenever a node *S* is accessed (e.g., when *S* is inserted, deleted, or is the goal of a search), the splay tree performs a process called **splaying**. Splaying moves *S* to the root of the BST. When *S* is being deleted, splaying moves the parent of *S* to the root. As in the AVL tree, a splay of node *S* consists of a series of **rotations**. A rotation moves *S* higher in the tree by adjusting its position with respect to its parent and grandparent. A side effect of the rotations is a tendency to balance the tree. There are three types of rotation.

A **single rotation** is performed only if *S* is a child of the root node. The single rotation is illustrated by Figure 13.7. It basically switches *S* with its parent in a

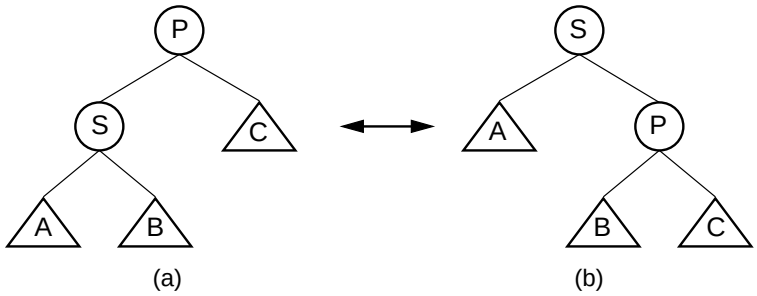

Figure 13.7 Splay tree single rotation. This rotation takes place only when the node being splayed is a child of the root. Here, node *S* is promoted to the root, rotating with node *P*. Because the value of *S* is less than the value of *P*, *P* must become *S*'s right child. The positions of subtrees *A*, *B*, and *C* are altered as appropriate to maintain the BST property, but the contents of these subtrees remains unchanged. (a) The original tree with *P* as the parent. (b) The tree after a rotation takes place. Performing a single rotation a second time will return the tree to its original shape. Equivalently, if (b) is the initial configuration of the tree (i.e., *S* is at the root and *P* is its right child), then (a) shows the result of a single rotation to splay *P* to the root.

way that retains the BST property. While Figure 13.7 is slightly different from Figure 13.5, in fact the splay tree single rotation is identical to the AVL tree single rotation.

Unlike the AVL tree, the splay tree requires two types of double rotation. Double rotations involve *S*, its parent (call it *P*), and *S*'s grandparent (call it *G*). The effect of a double rotation is to move *S* up two levels in the tree.

The first double rotation is called a **zigzag rotation**. It takes place when either of the following two conditions are met:

1. *S* is the left child of *P*, and *P* is the right child of *G*.
2. *S* is the right child of *P*, and *P* is the left child of *G*.

In other words, a zigzag rotation is used when *G*, *P*, and *S* form a zigzag. The zigzag rotation is illustrated by Figure 13.8.

The other double rotation is known as a **zigzig** rotation. A zigzig rotation takes place when either of the following two conditions are met:

1. *S* is the left child of *P*, which is in turn the left child of *G*.
2. *S* is the right child of *P*, which is in turn the right child of *G*.

Thus, a zigzig rotation takes place in those situations where a zigzag rotation is not appropriate. The zigzig rotation is illustrated by Figure 13.9. While Figure 13.9 appears somewhat different from Figure 13.6, in fact the zigzig rotation is identical to the AVL tree double rotation.

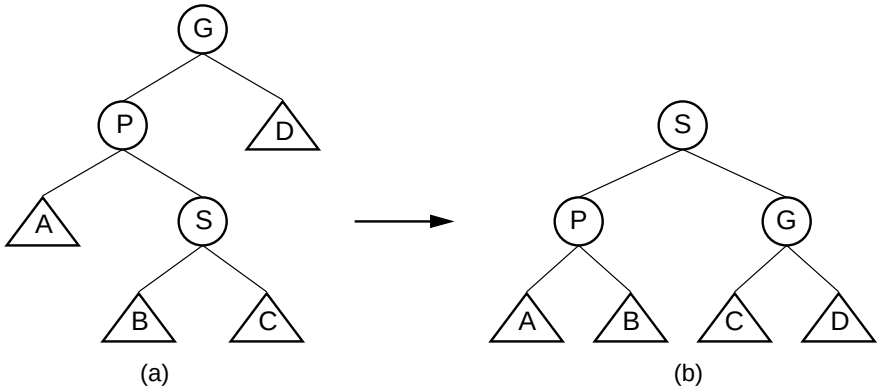

Figure 13.8 Splay tree zigzag rotation. (a) The original tree with *S*, *P*, and *G* in zigzag formation. (b) The tree after the rotation takes place. The positions of subtrees *A*, *B*, *C*, and *D* are altered as appropriate to maintain the BST property.

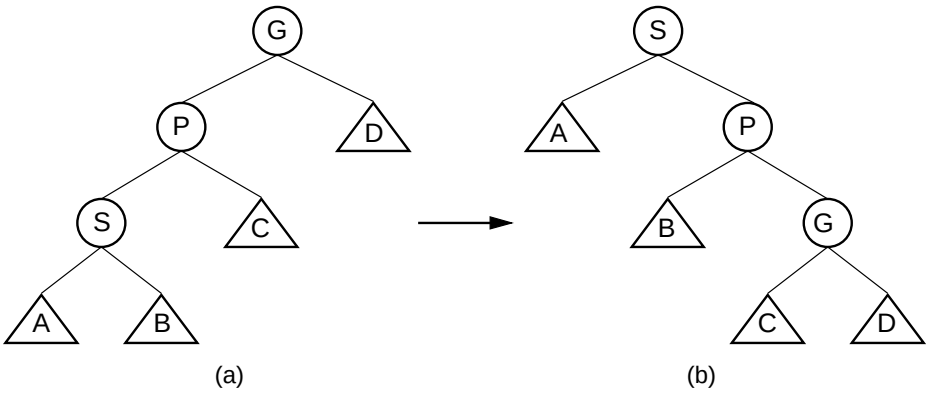

Figure 13.9 Splay tree zigzig rotation. (a) The original tree with *S*, *P*, and *G* in zigzig formation. (b) The tree after the rotation takes place. The positions of subtrees *A*, *B*, *C*, and *D* are altered as appropriate to maintain the BST property.

Note that zigzag rotations tend to make the tree more balanced, because they bring subtrees *B* and *C* up one level while moving subtree *D* down one level. The result is often a reduction of the tree's height by one. Zigzig promotions and single rotations do not typically reduce the height of the tree; they merely bring the newly accessed record toward the root.

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 re-balance the tree. Regardless of balance, splaying will make frequently accessed nodes stay near the top of the tree, resulting in reduced access cost. Proof that the splay tree meets the guarantee of $O(m \log n)$ is beyond the scope of this book. Such a proof can be found in the references in Section 13.4.

Example 13.4 Consider a search for value 89 in the splay tree of Figure 13.10(a). The splay tree's search operation 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.

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 number line into pieces.

Some databases require support for multiple keys. In other words, records can be searched for using any one of several key fields, such as name or ID number. 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.

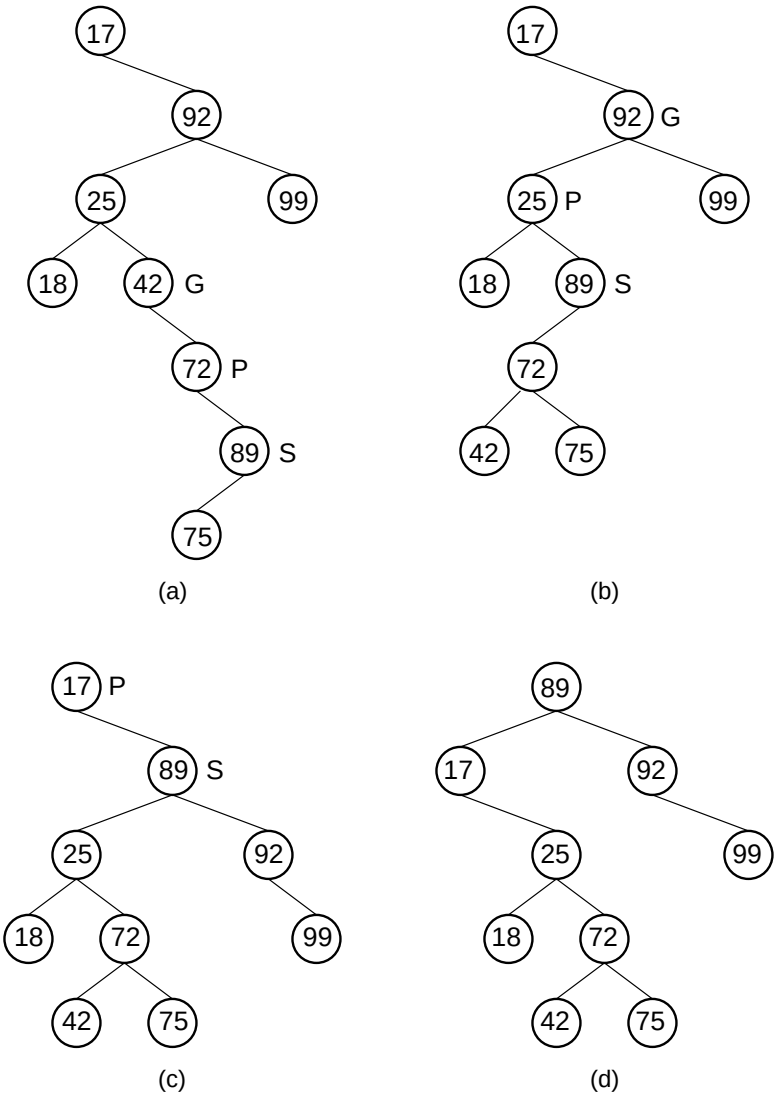

Figure 13.10 Example of splaying after performing a search in a splay tree. After finding the node with key value 89, that node is splayed to the root by performing three rotations. (a) The original splay tree. (b) The result of performing a zigzig rotation on the node with key value 89 in the tree of (a). (c) The result of performing a zigzag rotation on the node with key value 89 in the tree of (b). (d) The result of performing a single rotation on the node with key value 89 in the tree of (c). If the search had been for 91, the search would have been unsuccessful with the node storing key value 89 being that last one visited. In that case, the same splay operations would take place.

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**. In principle, the k-d tree could be used to unify key searching across any arbitrary set of keys such as name and zipcode. But in practice, it is nearly always used to support search on multidimensional coordinates, such as locations in 2D or 3D space. 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

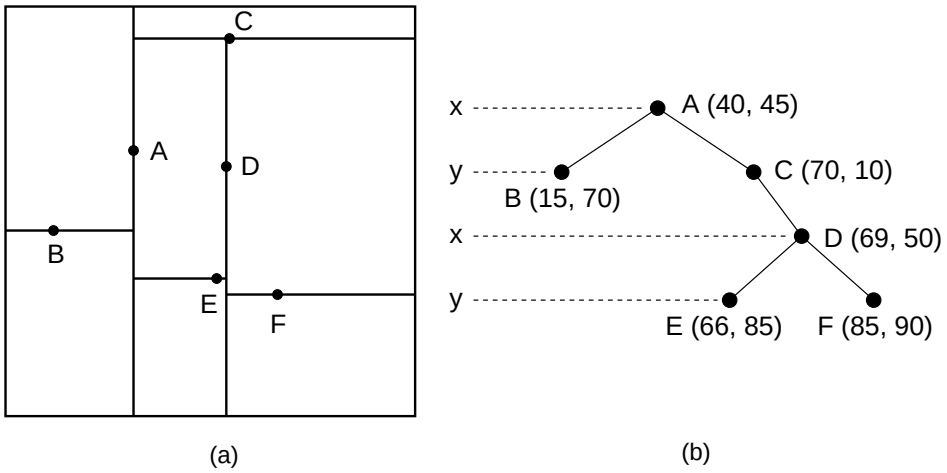

Figure 13.11 Example of a k-d tree. (a) The k-d tree decomposition for a 128×128-unit region containing seven data points. (b) The k-d tree for the region of (a).

space into rectangles that show the extent of where nodes can 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.

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.

If the search process reaches a **`null`** pointer, then that point is not contained in the tree. Here is a k-d tree search implementation, equivalent to the **`findhelp`** function of the BST class. **`KD`** class private member **`D`** stores the key's dimension.

```
private E findhelp(KDNode<E> rt, int[] key, int level) {
  if (rt == null) return null;
  E it = rt.element();
  int[] itkey = rt.key();
  if ((itkey[0] == key[0]) && (itkey[1] == key[1]))
    return rt.element();
  if (itkey[level] > key[level])
    return findhelp(rt.left(), key, (level+1)%D);
  else
    return findhelp(rt.right(), key, (level+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 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

```
private KDNode<E>
findmin(KDNode<E> rt, int descrim, int level) {
  KDNode<E> temp1, temp2;
  int[] key1 = null;
  int[] key2 = null;
  if (rt == null) return null;
  temp1 = findmin(rt.left(), descrim, (level+1)%D);
  if (temp1 != null) key1 = temp1.key();
  if (descrim != level) {
    temp2 = findmin(rt.right(), descrim, (level+1)%D);
    if (temp2 != null) key2 = temp2.key();
    if ((temp1 == null) || ((temp2 != null) &&
                  (key1[descrim] > key2[descrim])))
    temp1 = temp2;
    key1 = key2;
  } // Now, temp1 has the smaller value
  int[] rtkey = rt.key();
  if ((temp1 == null) || (key1[descrim] > rtkey[descrim]))
    return rt;
  else
    return temp1;
}
```

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.

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 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,

[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.

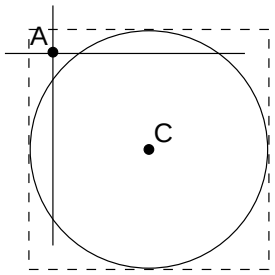

Figure 13.13 Function **`InCircle`** must check the Euclidean distance between a record and the query point. It is possible for a record *A* to have x- and y-coordinates each within the query distance of the query point *C*, yet have *A* itself lie outside the query circle.

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 We will now 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 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.

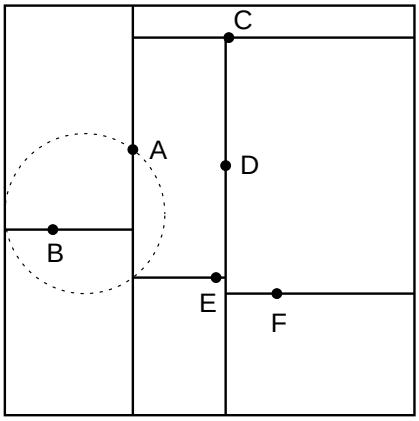

Figure 13.14 Searching in the k-d treeof Figure 13.11. (a) The k-d tree decomposition for a 128×128-unit region containing seven data points. (b) The k-d tree for the region of (a).

```
private void rshelp(KDNode<E> rt, int[] point,
                    int radius, int lev) {
  if (rt == null) return;
  int[] rtkey = rt.key();
  if (InCircle(point, radius, rtkey))
    System.out.println(rt.element());
  if (rtkey[lev] > (point[lev] - radius))
    rshelp(rt.left(), point, radius, (lev+1)%D);
  if (rtkey[lev] < (point[lev] + radius))
    rshelp(rt.right(), point, radius, (lev+1)%D);
}
```

Figure 13.15 The k-d tree region search method.

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.

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

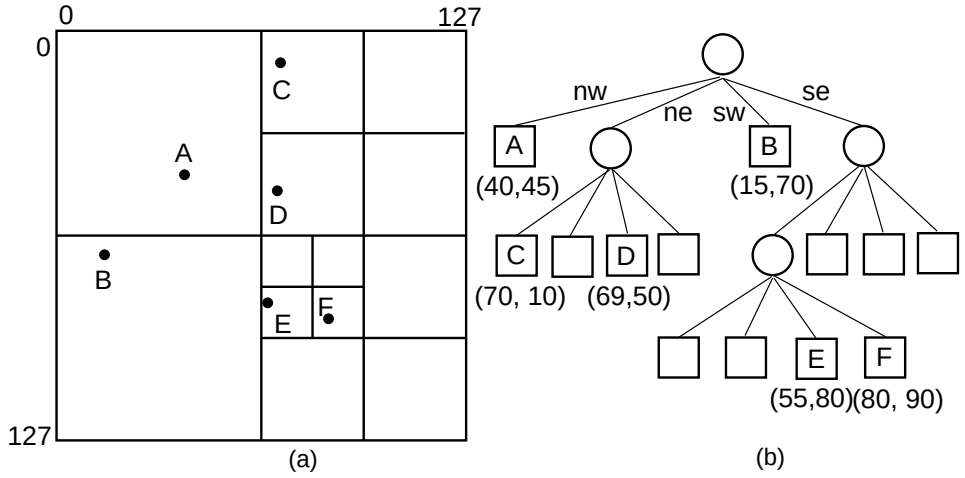

Figure 13.16 Example of a PR quadtree. (a) A map of data points. We define the region to be square with origin at the upper-left-hand corner and sides of length 128. (b) The PR quadtree for the points in (a). (a) also shows the block decomposition imposed by the PR quadtree for this region.

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 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

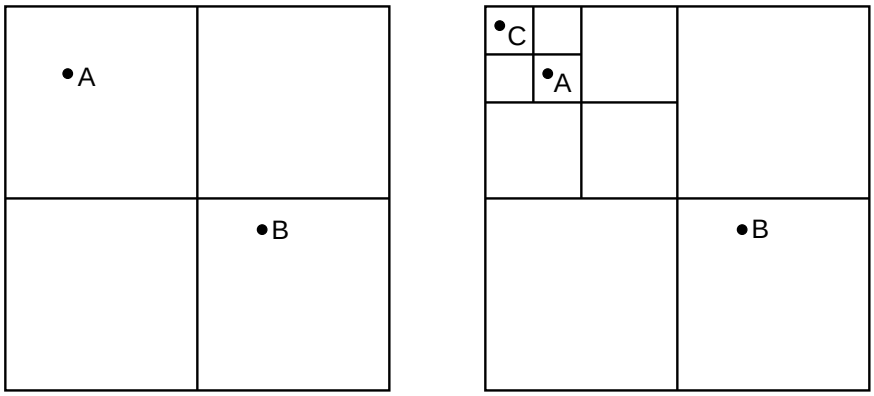

Figure 13.17 PR quadtree insertion example. (a) The initial PR quadtree containing two data points. (b) The result of inserting point *C*. The block containing *A* must be decomposed into four sub-blocks. Points *A* and *C* would still be in the same block if only one subdivision takes place, so a second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

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 represented 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 the 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 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. Use of the composite design would be difficult if `null` pointers are used to represent empty leaf nodes. It turns out that the PR quadtree insert and delete methods are easier to implement when using the composite design.

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 (and need not be full), 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

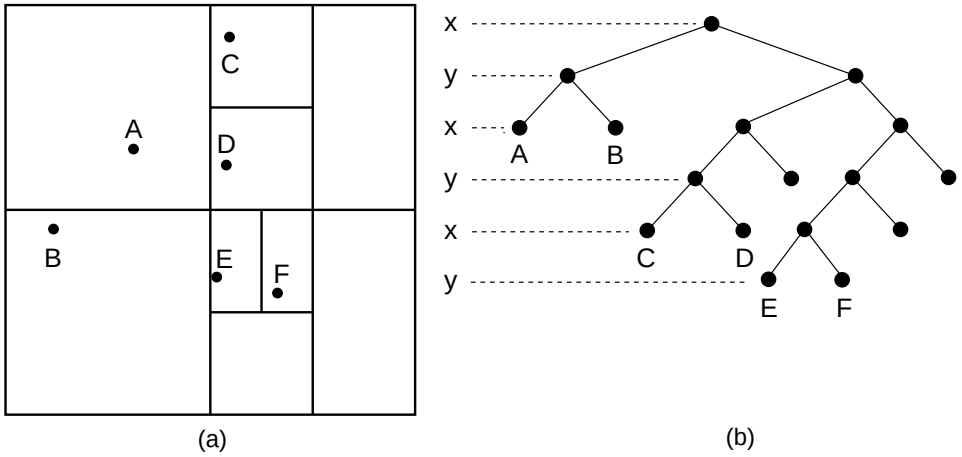

Figure 13.18 An example of the bintree, a binary tree using key space decomposition and discriminators rotating among the dimensions. Compare this with the k-d tree of Figure 13.11 and the PR quadtree of Figure 13.16.

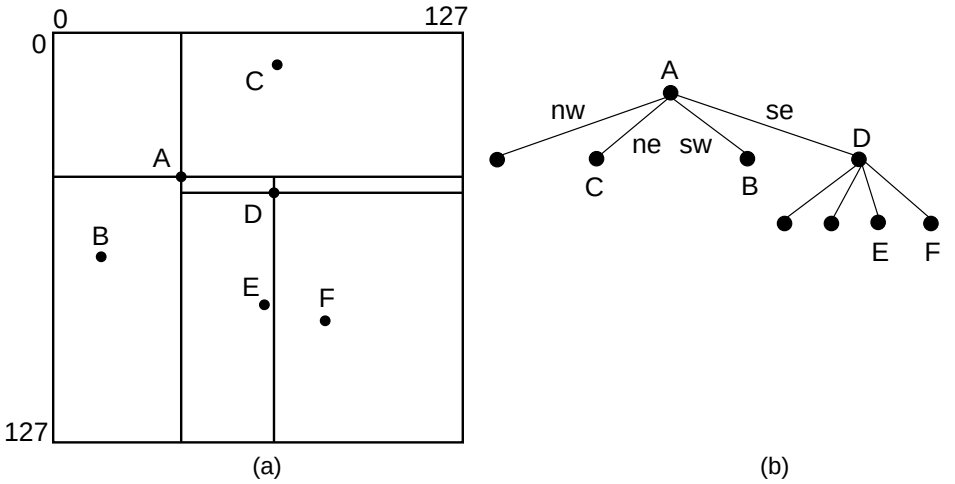

Figure 13.19 An example of the point quadtree, a 4-ary tree using object space decomposition. Compare this with the PR quadtree of Figure 13.11.

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. Dozens of distinct spatial data structures have been invented, many with 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.

One such 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 decomposition 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 *Foundations of Multidimensional and Metric Data Structures* by Hanan Samet [Sam06]. This is also the best reference for more information on the PR quadtree. The k-d tree was invented by John Louis Bentley. For further information on the k-d tree, in addition to [Sam06], 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 multi-way branching, see "A Generalized Comparison of Quadtree and Bintree Storage Requirements" by Shaffer, Juvvadi, and Heath [SJH93].

Closely related to spatial data structures are data structures for storing multi-dimensional data (which might not necessarily be spatial in nature). A popular

data structure for storing such data is the R-tree, which was originally proposed by Guttman [Gut84].

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 binary trie as shown in Figure 13.1.

13.4 Write the deletion routine for a binary trie as shown in Figure 13.1.

13.5 **(a)** Show the result (including appropriate rotations) of inserting the value 39 into the AVL tree on the left in Figure 13.4.

(b) Show the result (including appropriate rotations) of inserting the value 300 into the AVL tree on the left in Figure 13.4.

(c) Show the result (including appropriate rotations) of inserting the value 50 into the AVL tree on the left in Figure 13.4.

(d) Show the result (including appropriate rotations) of inserting the value 1 into the AVL tree on the left in Figure 13.4.

13.6 Show the splay tree that results from searching for value 75 in the splay tree of Figure 13.10(d).

13.7 Show the splay tree that results from searching for value 18 in the splay tree of Figure 13.10(d).

13.8 Some applications do not permit storing two records with duplicate key values. In such a case, an attempt to insert a duplicate-keyed record into a tree structure such as a splay tree should result in a failure on insert. What is the appropriate action to take in a splay tree implementation when the insert routine is called with a duplicate-keyed record?

13.9 Show the result of deleting point A from the k-d tree of Figure 13.11.

13.10 **(a)** Show the result of building a k-d tree from the following points (inserted in the order given). A (20, 20), B (10, 30), C (25, 50), D (35, 25), E (30, 45), F (30, 35), G (55, 40), H (45, 35), I (50, 30).

(b) Show the result of deleting point A from the tree you built in part (a).

13.11 **(a)** Show the result of deleting F from the PR quadtree of Figure 13.16.

(b) Show the result of deleting records E and F from the PR quadtree of Figure 13.16.

13.12 **(a)** Show the result of building a PR quadtree from the following points (inserted in the order given). Assume the tree is representing a space of 64 by 64 units. A (20, 20), B (10, 30), C (25, 50), D (35, 25), E (30, 45), F (30, 35), G (45, 25), H (45, 30), I (50, 30).

(b) Show the result of deleting point C from the tree you built in part (a).

(c) Show the result of deleting point F from the resulting tree in part (b).

13.13 On average, how many leaf nodes of a PR quadtree will typically be empty? Explain why.

13.14 When performing a region search on a PR quadtree, we need only search those subtrees of an internal node whose corresponding square falls within the query circle. This is most easily computed by comparing the x and y ranges of the query circle against the x and y ranges of the square corresponding to the subtree. However, as illustrated by Figure 13.13, the x and y ranges might overlap without the circle actually intersecting the square. Write a function that accurately determines if a circle and a square intersect.

13.15 **(a)** Show the result of building a bintree from the following points (inserted in the order given). Assume the tree is representing a space of 64 by 64 units. A (20, 20), B (10, 30), C (25, 50), D (35, 25), E (30, 45), F (30, 35), G (45, 25), H (45, 30), I (50, 30).

(b) Show the result of deleting point C from the tree you built in part (a).

(c) Show the result of deleting point F from the resulting tree in part (b).

13.16 Compare the trees constructed for Exercises 12 and 15 in terms of the number of internal nodes, full leaf nodes, empty leaf nodes, and total depths of the two trees.

13.17 Show the result of building a point quadtree from the following points (inserted in the order given). Assume the tree is representing a space of 64 by 64 units. A (20, 20), B (10, 30), C (25, 50), D (35, 25), E (30, 45), F (31, 35), G (45, 26), H (44, 30), I (50, 30).

13.6 Projects

13.1 Use the trie data structure to devise a program to sort variable-length strings. The program's running time should be proportional to the total number of letters in all of the strings. Note that some strings might be very long while most are short.

13.2 Define the set of **suffix strings** for a string *S* to be *S*, *S* without its first character, *S* without its first two characters, and so on. For example, the complete set of suffix strings for "HELLO" would be

$$\{\text{HELLO}, \text{ELLO}, \text{LLO}, \text{LO}, \text{O}\}.$$

A **suffix tree** is a PAT trie that contains all of the suffix strings for a given string, and associates each suffix with the complete string. The advantage of a suffix tree is that it allows a search for strings using "wildcards." For example, the search key "TH*" means to find all strings with "TH" as the first two characters. This can easily be done with a regular trie. Searching for "*TH" is not efficient in a regular trie, but it is efficient in a suffix tree.

Implement the suffix tree for a dictionary of words or phrases, with support for wildcard search.

13.3 Revise the BST class of Section 5.4 to use the AVL tree rotations. Your new implementation should not modify the original BST class ADT. Compare your AVL tree against an implementation of the standard BST over a wide variety of input data. Under what conditions does the splay tree actually save time?

13.4 Revise the BST class of Section 5.4 to use the splay tree rotations. Your new implementation should not modify the original BST class ADT. Compare your splay tree against an implementation of the standard BST over a wide variety of input data. Under what conditions does the splay tree actually save time?

13.5 Implement a city database using the k-d tree. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x- and y-coordinates. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. You should also support region queries, that is, a request to print all records within a given distance of a specified point.

13.6 Implement a city database using the PR quadtree. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x- and y-coordinates. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. You should also support region queries, that is, a request to print all records within a given distance of a specified point.

13.7 Implement and test the PR quadtree, using the composite design to implement the insert, search, and delete operations.

13.8 Implement a city database using the bintree. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x- and y-coordinates. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. You should also support region queries, that is, a request to print all records within a given distance of a specified point.

13.9 Implement a city database using the point quadtree. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x- and y-coordinates. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. You should also support region queries, that is, a request to print all records within a given distance of a specified point.

13.10 Use the PR quadtree to implement an efficient solution to Problem 6.5. That is, store the set of points in a PR quadtree. For each point, the PR quadtree is used to find those points within distance *D* that should be equivalenced. What is the asymptotic complexity of this solution?

13.11 Select any two of the point representations described in this chapter (i.e., the k-d tree, the PR quadtree, the bintree, and the point quadtree). Implement your two choices and compare them over a wide range of data sets. Describe which is easier to implement, which appears to be more space efficient, and which appears to be more time efficient.

13.12 Implement a representation for a collection of (two dimensional) rectangles using a quadtree based on regular decomposition. Assume that the space being represented is a square whose width and height are some power of two. Rectangles are assumed to have integer coordinates and integer width and height. Pick some value c, and use as a decomposition rule that a region is subdivided into four equal-sized regions whenever it contains more that c rectangles. A special case occurs if all of these rectangles intersect at some point within the current region (because decomposing such a node would never reach termination). In this situation, the node simply stores pointers to more than c rectangles. Try your representation on data sets of rectangles with varying values of c.