As in the case of the point quadtree, the amount of work expended in building a k-d tree is equal to the total path length (TPL) of the tree as it reflects the cost of searching for all of the elements. Bentley [164] shows that the TPL of a k-d tree built by inserting N points in random order into an initially empty tree is $O(N \cdot \log_2 N)$ and thus the average cost of inserting a node is $O(\log_2 N)$. The extreme cases are worse since the shape of the k-d tree depends on the order in which the nodes are inserted into it, thereby affecting the TPL.

Exercises

Assume that each point in the k-d tree is implemented as a record of type *node* containing three plus d fields. The first two fields, LEFT and RIGHT contain pointers to the node's two children, corresponding to the directions 'LEFT' and 'RIGHT', respectively. If P is a pointer to a node, and I is a direction, then these fields are referenced as CHILD(P, I). At times, these two fields are also referred to as LOCHILD(P) and HICHILD(P), corresponding to the left and right children, respectively. We can determine the side of the tree in which a node, say P, lies relative to its father by use of the function CHILDTYPE(P), which has a value of I if CHILD(FATHER$(P), I) = P$. COORD is a one-dimensional array containing the values of the d coordinates of the data point. If P is a pointer to a node, and I is a coordinate name, then these fields are referenced as COORD(P, I). The DISC field indicates the name of the coordinate on whose value the node discriminates (i.e., tests). The DISC field is not always necessary as, for example, when the relative order in which the different axes (i.e., coordinates) are partitioned is constant. In this case, it is easy to keep track of the discriminator type of the node being visited as the tree is descended. The empty k-d tree is represented by NIL.

1. Give an algorithm KDCOMPARE to determine the child of a k-d tree rooted at r in which point p lies.
2. Give an algorithm KDINSERT to insert a point p in a k-d tree rooted at node r. Make use of procedure KDCOMPARE from Exercise 1. There is no need to make use of the CHILDTYPE function.
3. Modify procedures KDCOMPARE and KDINSERT to handle a k-d tree node implementation that makes use of a superkey in its discriminator field.
4. Prove that the TPL of a k-d tree of N nodes built by inserting the N points in a random order is $O(N \cdot \log_2 N)$.
5. Give the k-d tree analogs of the single and double rotation operators for use in balancing a binary search tree [1046, p. 461]. Make sure that you handle equal key values correctly.

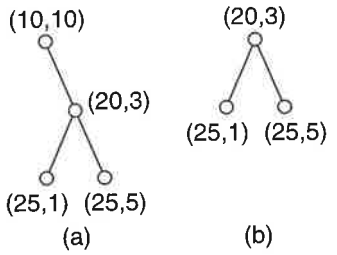

Figure 1.36
(a) Example of a two-dimensional k-d tree whose (b) right child is not a k-d tree.

1.5.1.2 Deletion

Deletion of nodes from k-d trees is considerably more complex than it is for binary search trees. Observe that, unlike the binary search tree, not every subtree of a k-d tree is itself a k-d tree. For example, although Figure 1.36(a) is a k-d tree, its right subtree (see Figure 1.36(b)) is not a k-d tree. This is because the root node in a two-dimensional k-d tree discriminates on the value of the x coordinate, while both children of the root node in Figure 1.36(b) have x coordinate values that are larger than that of the root node. Thus, we see that special care must be taken when deleting a node from a k-d tree.

In contrast, when deleting a node from a binary search tree, we simply move a node and a subtree. For example, deleting node 3 from Figure 1.37(a) results in replacing node 3 with node 4 and replacing the left subtree of 8 by the right subtree of 4 (see Figure 1.37(b)). However, this cannot be done, in general, in a k-d tree as the nodes with values 5 and 6 might not have the same relative relationship at their new depths.

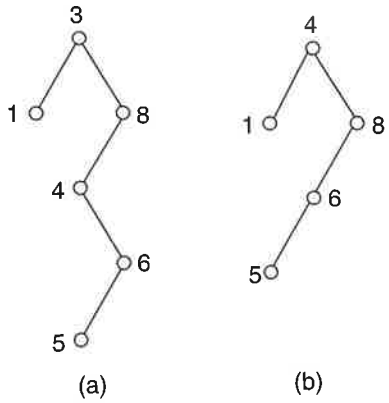

Figure 1.37
(a) Example of a binary search tree and (b) the result of deleting its root.

Deletion in a k-d tree can be achieved by the following recursive process. We use the k-d tree in Figure 1.38 to illustrate our discussion, and thus all references to nodes can be visualized by referring to the figure. Let us assume that we wish to delete the node (a,b) from the k-d tree. If both subtrees of (a,b) are empty, then replace (a,b) with the empty

tree. Otherwise, find a suitable replacement node in one of the subtrees of (a,b), say (c,d), and recursively delete (c,d) from the k-d tree. Once (c,d) has been deleted, replace (a,b) with (c,d).

At this point, it is appropriate to comment on what constitutes a "suitable" replacement node. Recall that an x-discriminator is a node that appears at an even depth, and hence partitions its space based on the value of its x coordinate. A y-discriminator is defined analogously for nodes at odd depths. Assume that (a,b) is an x-discriminator. We know that every node in (a,b)'s right subtree has an x coordinate with a value greater than or equal to a. The node that will replace (a,b) must bear the same relationship to the subtrees of (a,b). Using the analogy with binary search trees, it would seem that we would have a choice with respect to the replacement node. It must either be the node in the left subtree of (a,b) with the largest x coordinate value or the node in the right subtree of (a,b) with the smallest x coordinate value.

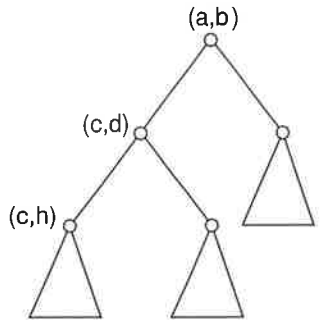

Figure 1.38
Example of a k-d tree illustrating why the replacement node should be chosen from the right subtree of the tree containing the deleted node.

Actually, we do not really have a choice as the following comments will make clear. If we use the node in the left subtree of (a,b) with the maximum x coordinate value, say (c,d), and if there exists another node in the same subtree with the same x coordinate value, say (c,h), then when (c,d) replaces node (a,b), there will be a node in the left subtree of (c,d) that does not belong there by virtue of having an x coordinate value of c. Thus, we see that, given our definition of a k-d tree, the replacement node must be chosen from the right subtree. Otherwise, a duplicate x coordinate value will disrupt the proper interaction between each node and its subtrees. Note that the replacement node need not be a leaf node.

The only remaining question is how to handle the case when the right subtree of (a,b) is empty. We use the k-d tree in Figure 1.39 to illustrate our discussion, and thus all references to nodes can be visualized by referring to the figure. This is resolved by the following recursive process. Find the node in the left subtree of (a,b) that has the smallest value for its x coordinate, say (c,d). Exchange the left and right subtrees of (a,b), replace the coordinate values of (a,b) with (c,d), and recursively apply the deletion procedure to node (c,d) from its prior position in the tree (i.e., in the previous left subtree of (a,b)).

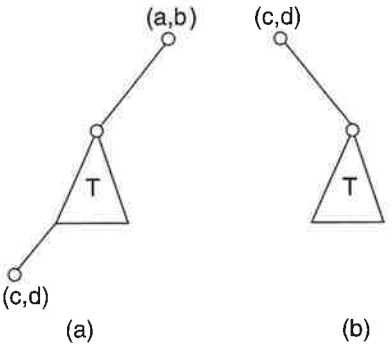

Figure 1.39
(a) Example k-d tree and (b) the result of deleting (a,b) from it.

With the aid of Figures 1.38 and 1.39, we have shown that the problem of deleting a node (a,b) from a k-d tree is reduced to that of finding the node with the smallest x coordinate value in a subtree of (a,b). Unfortunately, locating the node with the minimum x coordinate value is considerably more complex than the analogous problem for a binary search tree. In particular, although the node with the minimum x coordinate value must be in the left subtree of an x-discriminator, it could be in either subtree of a y-discriminator. Thus, search is involved, and care must be taken in coordinating this search so that when the deleted node is an x-discriminator at depth 0, only one of the two subtrees rooted at each odd depth is searched. This is done using procedure FINDDMINIMUM (see Exercise 1).

As can be seen from the discussion, deleting a node from a k-d tree can be costly. We can obtain an upper bound on the cost in the following manner. Clearly, the cost of deleting the root of a subtree is bounded from above by the number of nodes in the subtree. Letting TPL(T) denote the total path length of tree T, it can be shown that the sum of the subtree sizes of a tree is TPL(T) + N (see Exercise 5 in Section 1.4.1.2).

Bentley [164] proves that the TPL of a k-d tree built by inserting N points in a random order is $O(N \cdot \log_2 N)$, which means that the average cost of deleting a randomly selected node from a randomly built k-d tree has an upper bound of $O(\log_2 N)$. This relatively low value for the upper bound reflects the fact that most of the nodes in the k-d tree are leaf nodes. The cost of deleting root nodes is considerably higher. Clearly, it is bounded by N. Its cost is dominated by the cost of the process of finding a minimum element in a subtree, which is $O(N^{1-1/d})$ (see Exercise 6), since on every dth level (starting at the root), only one of the two subtrees needs to be searched.

As an example of the deletion process, consider the k-d tree in Figure 1.40(a, b). We wish to delete the root node A at (20,20). Assume that A is an x-discriminator. The node in the right subtree of A with a minimum value for its x coordinate is C at (25,50). Thus, C replaces A (see Figure 1.40(c)), and we recursively apply the deletion procedure to C's position in the tree. Since C's position was a y-discriminator, we seek the node with a minimum y coordinate value in the right subtree of C. However, C's right subtree was empty, which means that we must replace C with the node in its left subtree that has the minimum y coordinate value.

D at (35,25) is the node in the left subtree that satisfies this minimum value condition. It is moved up in the tree (see Figure 1.40(d)). Since D was an x-discriminator, we replace it by the node in its right subtree having a minimum x coordinate value. H at (45,35) satisfies this condition, and it is moved up in the tree (see Figure 1.40(e)). Again, H is an x-discriminator, and we replace it by I, the node in its right subtree with a minimum x coordinate value. Since I is a leaf node, our procedure terminates. Figure 1.40(f, g) shows the result of the deletion process.

In the above discussion, we had to make a special provision to account for our definition of a k-d tree node as a partition of space into two parts, one less than the key value tested by the node and one greater than or equal to the key value tested by the node. Defining a node in terms of a superkey alleviates this problem since we no longer always have to choose the replacing node from the right subtree. Instead, we now

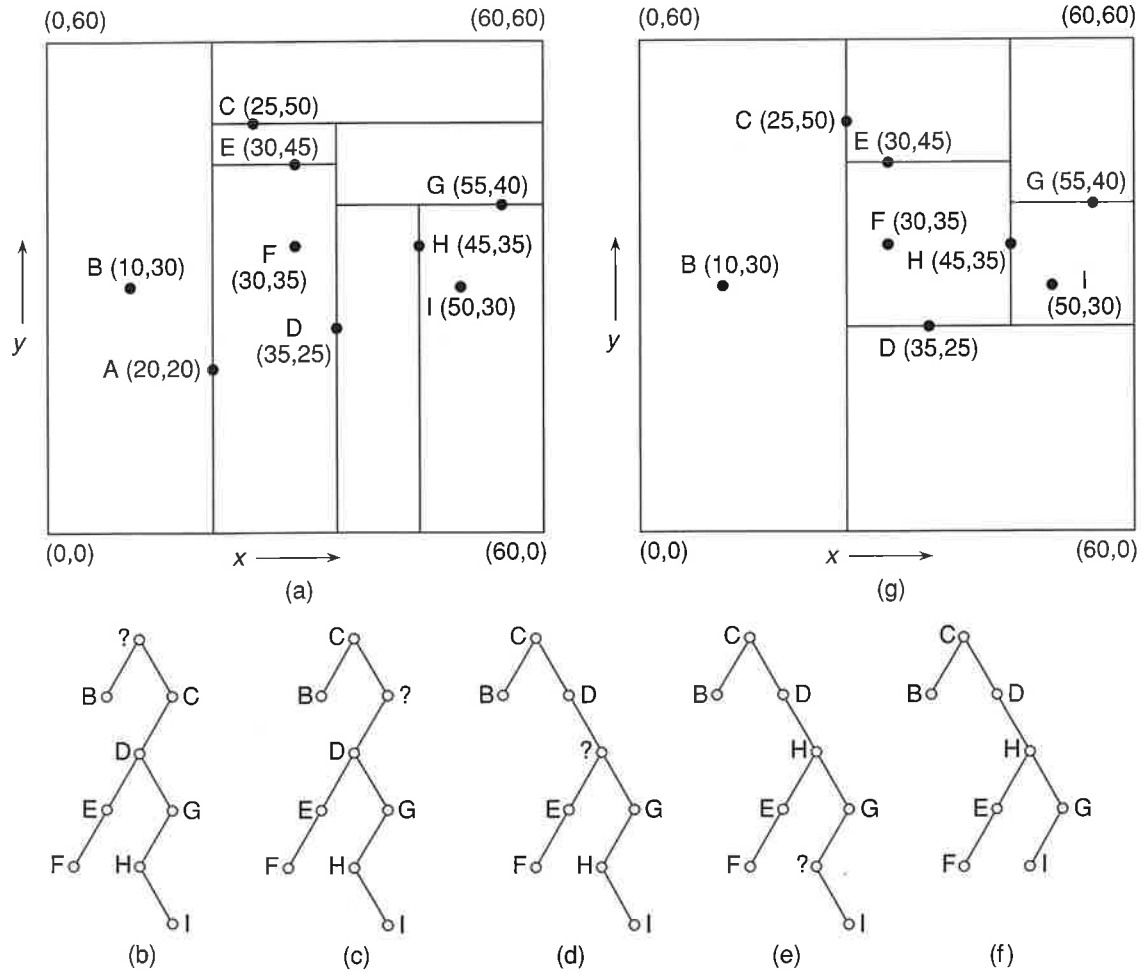

Figure 1.40
Example illustrating deletion in k-d trees where "?" indicates the node being deleted: (a) the original k-d tree, (b–f) successive steps in deleting node A, (g) the final k-d tree.

have a choice. The best algorithm is one that flip-flops between the left and right children, perhaps through the use of a random-number generator (see Exercise 5). Of course, node insertion and search are slightly more complex since it is possible that more than one key will have to be compared at each level in the tree (see Exercise 3 in Section 1.5.1.1 and Exercise 3 in Section 1.5.1.3).

Exercises

1. Given a node P in a k-d tree that discriminates on key D, write an algorithm, FINDDMINIMUM, to compute the D-minimum node in its left subtree. Repeat for the right subtree of P. Generalize your algorithm to work for either of the two subtrees.
2. Assuming the same k-d tree node implementation as in the exercises in Section 1.5.1.1, give an algorithm KDDELETE for deleting a point p from a k-d tree rooted at node r. The node does not have a FATHER field, although you may make use of the CHILDTYPE function. Make use of procedures KDCOMPARE and FINDDMINIMUM from Exercise 1 in Section 1.5.1.1 and Exercise 1 above, respectively.
3. Modify procedure KDDELETE in the solution to Exercise 2 to make use of a variant of FINDDMINIMUM that, in addition to returning the D-minimum node t, also returns t's CHILDTYPE values relative to its father f, as well as f.
4. Suppose that a k-d tree node is implemented as a record with a FATHER field containing a pointer to its father. Modify procedure KDDELETE in the solution to Exercise 2 to take advantage of this additional field.
5. Modify procedure KDDELETE in the solution to Exercise 2 to handle a k-d tree node implementation that makes use of a superkey in its discriminator field.
6. Prove that the cost of finding a D-minimum element in a k-d tree is $O(N^{1-1/d})$.

1.5.1.3 Search

Like quadtrees, k-d trees are useful in applications involving search. Again, we consider a typical query that seeks all nodes within a specified distance of a given point. The k-d tree data structure serves as a pruning device on the amount of search that is required; that is, many nodes need not be examined. To see how the pruning is achieved, suppose we are performing a region search of distance r around a node with coordinate values (a,b). In essence, we want to determine all nodes (x,y) whose Euclidean distance from (a,b) is less than or equal to r—that is, $r^2 \geq (a-x)^2 + (b-y)^2$.

Clearly, this is a circular region. The minimum x and y coordinate values of a node in this circle cannot be less than $a-r$ and $b-r$, respectively. Similarly, the maximum x and y coordinate values of a node in this circle cannot be greater than $a+r$ and $b+r$, respectively. Thus, if the search reaches a node with coordinate values (e,f), and KDCOMPARE$((a-r,b-r),(e,f)) =$ 'RIGHT', then there is no need to examine any nodes in the left subtree of (e,f). Similarly, the right subtree of (e,f) need not be searched when KDCOMPARE$((a+r,b+r),(e,f)) =$ 'LEFT'.

For example, suppose that we want to use the k-d tree in Figure 1.34 of the hypothetical database of Figure 1.1 to find all cities within three units of a point with coordinate values (88,6). In such a case, there is no need to search the left subtree of the root (i.e., Chicago with coordinate values (35,42)). Thus, we need examine only the right subtree of the tree rooted at Chicago. Similarly, there is no need to search the right subtree of the tree rooted at Mobile (i.e., coordinate values (52,10)). Continuing our search, we find that only Miami, at coordinate values (90,5), satisfies our request. Thus, we need examine only three nodes during our search.

Similar techniques are applied when the search region is rectangular, making the query meaningful for both locational and nonlocational data. In general, the search cost depends on the type of query. Given N points, Lee and Wong [1132] have shown that, in the worst case, the cost of a range search of a complete k-d tree is $O(d \cdot N^{1-1/d} + F)$,