10

TREE-STRUCTURED INDEXING

Exercise 10.1 Consider the B+ tree index of order $d = 2$ shown in Figure 10.1.

1. Show the tree that would result from inserting a data entry with key 9 into this tree.
2. Show the B+ tree that would result from inserting a data entry with key 3 into the original tree. How many page reads and page writes does the insertion require?
3. Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the left sibling is checked for possible redistribution.
4. Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the right sibling is checked for possible redistribution.
5. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 46 and then deleting the data entry with key 52.
6. Show the B+ tree that would result from deleting the data entry with key 91 from the original tree.

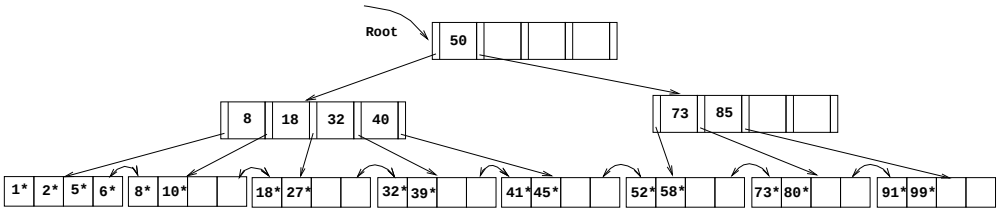

Figure 10.1 Tree for Exercise 10.1

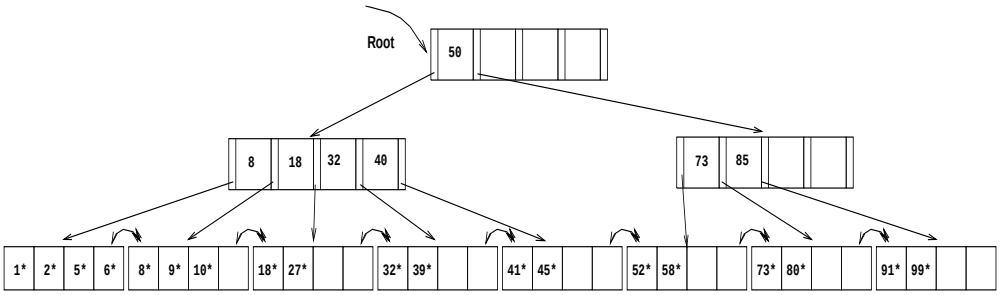

Figure 10.2

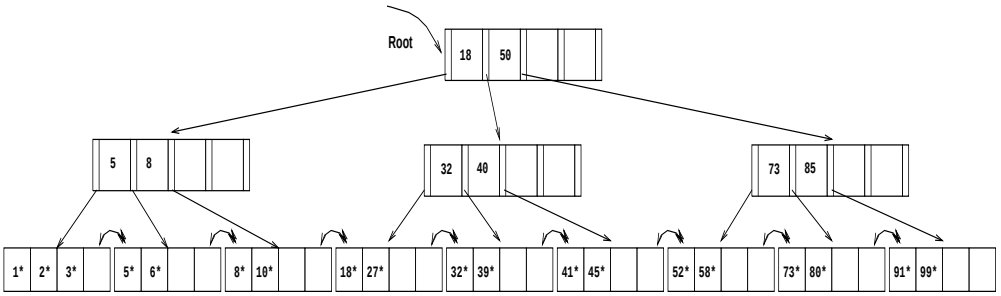

Figure 10.3

7. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 59, and then deleting the data entry with key 91.

8. Show the B+ tree that would result from successively deleting the data entries with keys 32, 39, 41, 45, and 73 from the original tree.

Answer 10.1 1. The data entry with key 9 is inserted on the second leaf page. The resulting tree is shown in figure 10.2.

2. The data entry with key 3 goes on the first leaf page F. Since F can accommodate at most four data entries ($d = 2$), F splits. The lowest data entry of the new leaf is given up to the ancestor which also splits. The result can be seen in figure 10.3. The insertion will require 5 page writes, 4 page reads and allocation of 2 new pages.

3. The data entry with key 8 is deleted, resulting in a leaf page N with less than two data entries. The left sibling L is checked for redistribution. Since L has more than two data entries, the remaining keys are redistributed between L and N, resulting in the tree in figure 10.4.

4. As is part 3, the data entry with key 8 is deleted from the leaf page N. N's right sibling R is checked for redistribution, but R has the minimum number of

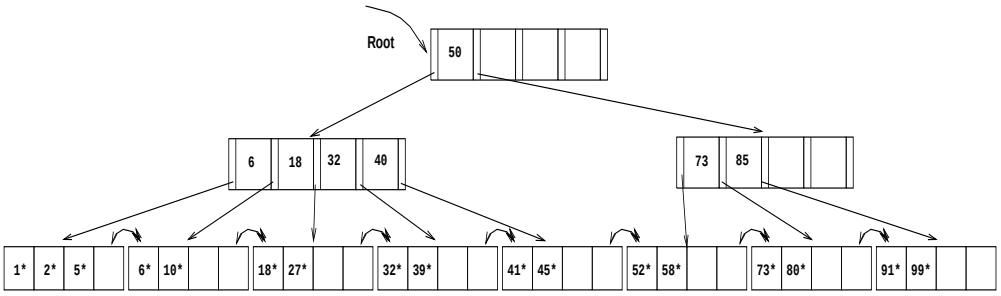

Figure 10.4

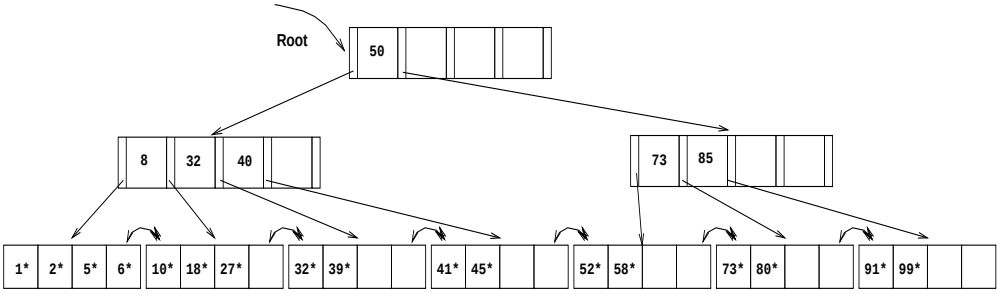

Figure 10.5

keys. Therefore the two siblings merge. The key in the ancestor which distinguished between the newly merged leaves is deleted. The resulting tree is shown in figure 10.5.

5. The data entry with key 46 can be inserted without any structural changes in the tree. But the removal of the data entry with key 52 causes its leaf page L to merge with a sibling (we chose the right sibling). This results in the removal of a key in the ancestor A of L and thereby lowering the number of keys on A below the minimum number of keys. Since the left sibling B of A has more than the minimum number of keys, redistribution between A and B takes place. The final tree is depicted in figure 10.6.

6. Deleting the data entry with key 91 causes a scenario similar to part 5. The result can be seen in figure 10.7.

7. The data entry with key 59 can be inserted without any structural changes in the tree. No sibling of the leaf page with the data entry with key 91 is affected by the insert. Therefore deleting the data entry with key 91 changes the tree in a way very similar to part 6. The result is depicted in figure 10.8.

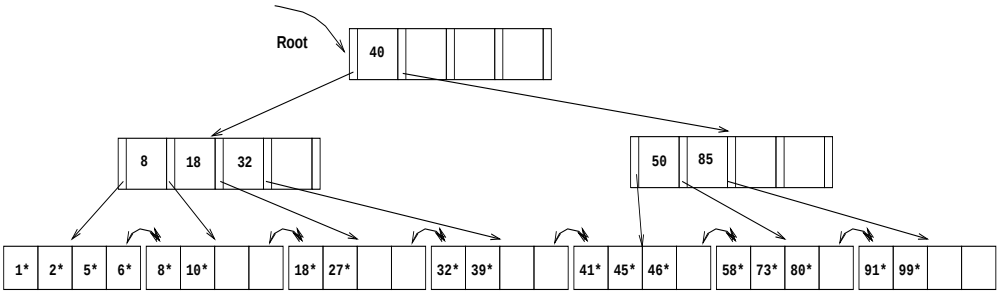

Figure 10.6

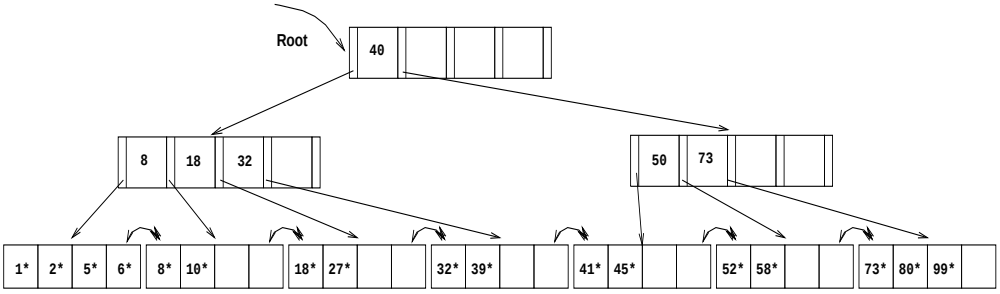

Figure 10.7

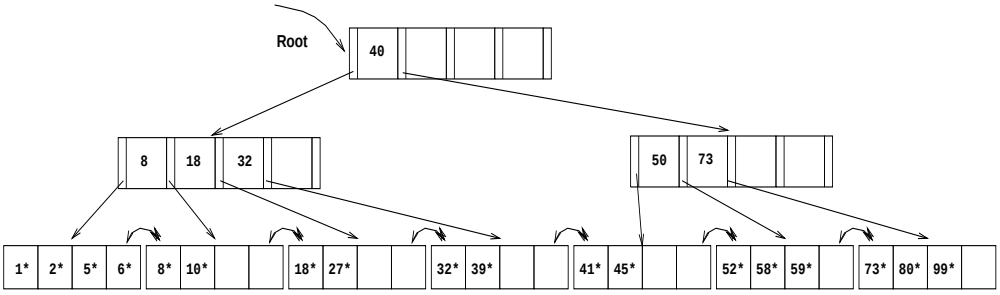

Figure 10.8

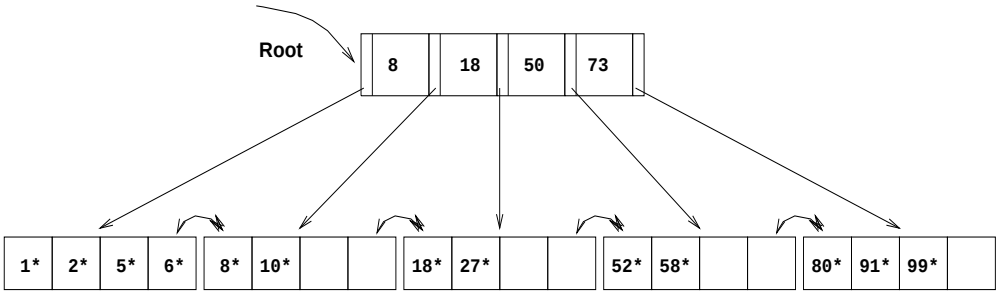

Figure 10.9

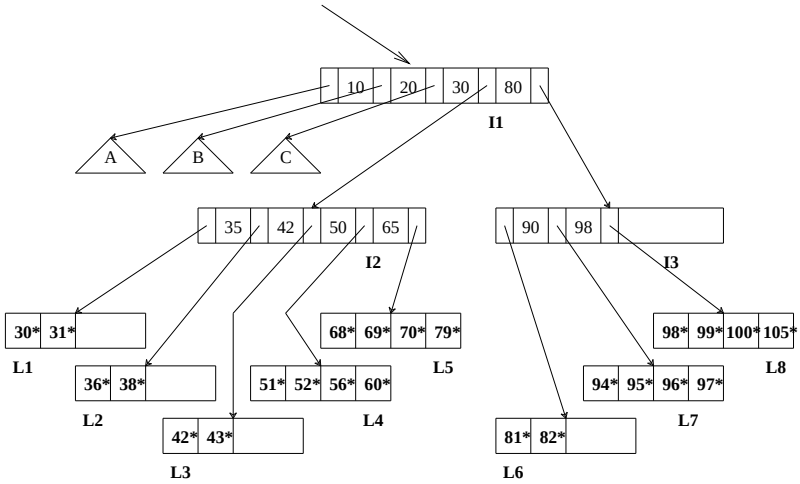

Figure 10.10 Tree for Exercise 10.2

8. Considering checking the right sibling for possible merging first, the successive deletion of the data entries with keys 32, 39, 41, 45 and 73 results in the tree shown in figure 10.9.

Exercise 10.2 Consider the B+ tree index shown in Figure 10.10, which uses Alternative (1) for data entries. Each intermediate node can hold up to five pointers and four key values. Each leaf can hold up to four records, and leaf nodes are doubly linked as usual, although these links are not shown in the figure. Answer the following questions.

1. Name all the tree nodes that must be fetched to answer the following query: "Get all records with search key greater than 38."

2. Show the B+ tree that would result from inserting a record with search key 109 into the tree.
3. Show the B+ tree that would result from deleting the record with search key 81 from the original tree.
4. Name a search key value such that inserting it into the (original) tree would cause an increase in the height of the tree.
5. Note that subtrees A, B, and C are not fully specified. Nonetheless, what can you infer about the contents and the shape of these trees?
6. How would your answers to the preceding questions change if this were an ISAM index?
7. Suppose that this is an ISAM index. What is the minimum number of insertions needed to create a chain of three overflow pages?

Answer 10.2 Answer omitted.

Exercise 10.3 Answer the following questions:

1. What is the minimum space utilization for a B+ tree index?
2. What is the minimum space utilization for an ISAM index?
3. If your database system supported both a static and a dynamic tree index (say, ISAM and B+ trees), would you ever consider using the *static* index in preference to the *dynamic* index?

Answer 10.3 The answer to each question is given below.

1. By the definition of a B+ tree, each index page, except for the root, has at least d and at most $2d$ key entries. Therefore—with the exception of the root—the minimum space utilization guaranteed by a B+ tree index is 50 percent.
2. The minimum space utilization by an ISAM index depends on the design of the index and the data distribution over the lifetime of ISAM index. Since an ISAM index is static, empty spaces in index pages are never filled (in contrast to a B+ tree index, which is a dynamic index). Therefore the space utilization of ISAM index pages is usually close to 100 percent by design. However, there is no guarantee for leave pages' utilization.
3. A static index without overflow pages is faster than a dynamic index on inserts and deletes, since index pages are only read and never written. If the set of keys that will be inserted into the tree is known in advance, then it is possible to build

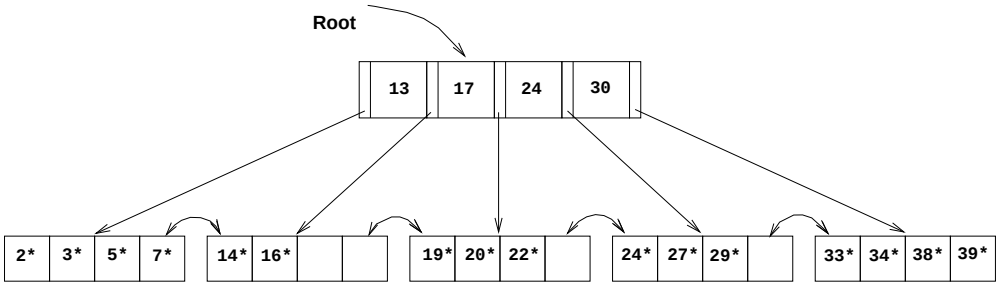

Figure 10.11 Tree for Exercise 10.5

a static index which reserves enough space for all possible future inserts. Also if the system goes periodically off line, static indices can be rebuilt and scaled to the current occupancy of the index. Infrequent or scheduled updates are flags for when to consider a static index structure.

Exercise 10.4 Suppose that a page can contain at most four data values and that all data values are integers. Using only B+ trees of order 2, give examples of each of the following:

1. A B+ tree whose height changes from 2 to 3 when the value 25 is inserted. Show your structure before and after the insertion.
2. A B+ tree in which the deletion of the value 25 leads to a redistribution. Show your structure before and after the deletion.
3. A B+ tree in which the deletion of the value 25 causes a merge of two nodes but without altering the height of the tree.
4. An ISAM structure with four buckets, none of which has an overflow page. Further, every bucket has space for exactly one more entry. Show your structure before and after inserting two additional values, chosen so that an overflow page is created.

Answer 10.4 Answer omitted.

Exercise 10.5 Consider the B+ tree shown in Figure 10.21.

1. Identify a list of five data entries such that:
   (a) Inserting the entries in the order shown and then deleting them in the opposite order (e.g., insert a, insert b, delete b, delete a) results in the original tree.

(b) Inserting the entries in the order shown and then deleting them in the opposite order (e.g., insert a, insert b, delete b, delete a) results in a different tree.

2. What is the minimum number of insertions of data entries with distinct keys that will cause the height of the (original) tree to change from its current value (of 1) to 3?

3. Would the minimum number of insertions that will cause the original tree to increase to height 3 change if you were allowed to insert duplicates (multiple data entries with the same key), assuming that overflow pages are not used for handling duplicates?

Answer 10.5 The answer to each question is given below.

1. The answer to each part is given below.

   (a) One example is the set of five data entries with keys 17, 18, 13, 15, and 25. Inserting 17 and 18 will cause the tree to split and gain a level. Inserting 13, 15, and 25 does change the tree structure any further, so deleting them in reverse order causes no structure change. When 18 is deleted, redistribution will be possible from an adjacent node since one node will contain only the value 17, and its right neighbor will contain 19, 20, and 22. Finally, when 17 is deleted, no redistribution will be possible so the tree will loose a level and will return to the original tree.

   (b) Inserting and deleting the set 13, 15, 18, 25, and 4 will cause a change in the tree structure. When 4 is inserted, the right most leave will split causing the tree to gain a level. When it is deleted, the tree will not shrink in size. Since inserts 13, 15, 18, and 25 did not affect the right most node, their deletion will not change the altered structure either.

2. Let us call the current tree depicted in Figure 10.21 T. T has 16 data entries. The smallest tree S of height 3 which is created exclusively through inserts has $(1*2*3*3)*2+1 = 37$ data entries in its leaf pages. S has 18 leaf pages with two data entries each and one leaf page with three data entries. T has already four leaf pages which have more than two data entries; they can be filled and made to split, but after each spilt, one of the two pages will still has three data entries remaining. Therefore the smallest tree of height 3 which can possibly be created from T only through inserts has $(1*2*3*3)*2+4 = 40$ data entries. Therefore the minimum number of entries that will cause the height of T to change to 3 is $40 - 16$=24.

3. The argument in part 2 does not assume anything about the data entries to be inserted; it is valid if duplicates can be inserted as well. Therefore the solution does not change.

Exercise 10.6 Answer Exercise 10.5 assuming that the tree is an ISAM tree! (Some of the examples asked for may not exist—if so, explain briefly.)

Answer 10.6 Answer omitted.

Exercise 10.7 Suppose that you have a sorted file and want to construct a dense primary B+ tree index on this file.

1. One way to accomplish this task is to scan the file, record by record, inserting each one using the B+ tree insertion procedure. What performance and storage utilization problems are there with this approach?
2. Explain how the bulk-loading algorithm described in the text improves upon this scheme.

Answer 10.7 1. This approach is likely to be quite expensive, since each entry requires us to start from the root and go down to the appropriate leaf page. Even though the index level pages are likely to stay in the buffer pool between successive requests, the overhead is still considerable. Also, according to the insertion algorithm, each time a node splits, the data entries are redistributed evenly to both nodes. This leads to a fixed page utilization of 50%

2. The bulk loading algorithm has good performance and space utilization compared with the repeated inserts approach. Since the B+ tree is grown from the bottom up, the bulk loading algorithm allows the administrator to pre-set the amount each index and data page should be filled. This allows good performance for future inserts, and supports some desired space utilization.

Exercise 10.8 Assume that you have just built a dense B+ tree index using Alternative (2) on a heap file containing 20,000 records. The key field for this B+ tree index is a 40-byte string, and it is a candidate key. Pointers (i.e., record ids and page ids) are (at most) 10-byte values. The size of one disk page is 1000 bytes. The index was built in a bottom-up fashion using the bulk-loading algorithm, and the nodes at each level were filled up as much as possible.

1. How many levels does the resulting tree have?
2. For each level of the tree, how many nodes are at that level?
3. How many levels would the resulting tree have if key compression is used and it reduces the average size of each key in an entry to 10 bytes?
4. How many levels would the resulting tree have without key compression but with all pages 70 percent full?

Answer 10.8 Answer omitted.

Exercise 10.9 The algorithms for insertion and deletion into a B+ tree are presented as recursive algorithms. In the code for *insert*, for instance, a call is made at the parent of a node N to insert into (the subtree rooted at) node N, and when this call returns, the current node is the parent of N. Thus, we do not maintain any 'parent pointers' in nodes of B+ tree. Such pointers are not part of the B+ tree structure for a good reason, as this exercise demonstrates. An alternative approach that uses parent pointers—again, remember that such pointers are *not* part of the standard B+ tree structure!—in each node appears to be simpler:

> Search to the appropriate leaf using the search algorithm; then insert the entry and split if necessary, with splits propagated to parents if necessary (using the parent pointers to find the parents).

Consider this (unsatisfactory) alternative approach:

1. Suppose that an internal node N is split into nodes N and $N2$. What can you say about the parent pointers in the children of the original node N?
2. Suggest two ways of dealing with the inconsistent parent pointers in the children of node N.
3. For each of these suggestions, identify a potential (major) disadvantage.
4. What conclusions can you draw from this exercise?

Answer 10.9 The answer to each question is given below.

1. The parent pointers in either d or $d+1$ of the children of the original node N are not valid any more: they still point to N, but they should point to $N2$.
2. One solution is to adjust all parent pointers in the children of the original node N which became children of $N2$. Another solution is to leave the pointers during the insert operation and to adjust them later when the page is actually needed and read into memory anyway.
3. The first solution requires at least $d+1$ additional page reads (and sometime later, page writes) on an insert, which would result in a remarkable slowdown. In the second solution mentioned above, a child M, which has a parent pointer to be adjusted, is updated if an operation is performed which actually reads M into memory (maybe on a down path from the root to a leaf page). But this solution modifies M and therefore requires sometime later a write of M, which might not have been necessary if there were no parent pointers.

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	3.8
53666	Jones	jones@cs	18	3.4
53901	Jones	jones@toy	18	3.4
53902	Jones	jones@physics	18	3.4
53903	Jones	jones@english	18	3.4
53904	Jones	jones@genetics	18	3.4
53905	Jones	jones@astro	18	3.4
53906	Jones	jones@chem	18	3.4
53902	Jones	jones@sanitation	18	3.8
53688	Smith	smith@ee	19	3.2
53650	Smith	smith@math	19	3.8
54001	Smith	smith@ee	19	3.5
54005	Smith	smith@cs	19	3.8
54009	Smith	smith@astro	19	2.2

Figure 10.12 An Instance of the Students Relation

4. In conclusion, to add parent pointers to the B+ tree data structure is not a good modification. Parent pointers cause unnecessary page updates and so lead to a decrease in performance.

Exercise 10.10 Consider the instance of the Students relation shown in Figure 10.22. Show a B+ tree of order 2 in each of these cases below, assuming that duplicates are handled using overflow pages. Clearly indicate what the data entries are (i.e., do not use the $k*$ convention).

1. A B+ tree index on *age* using Alternative (1) for data entries.
2. A dense B+ tree index on *gpa* using Alternative (2) for data entries. For this question, assume that these tuples are stored in a sorted file in the order shown in Figure 10.22: The first tuple is in page 1, slot 1; the second tuple is in page 1, slot 2; and so on. Each page can store up to three data records. You can use $\langle$*page-id, slot*$\rangle$ to identify a tuple.

Answer 10.10 Answer omitted.

Exercise 10.11 Suppose that duplicates are handled using the approach without overflow pages discussed in Section **??**. Describe an algorithm to search for the left-most occurrence of a data entry with search key value K.

Answer 10.11 The key to understanding this problem is to observe that when a leaf splits due to inserted duplicates, then of the two resulting leaves, it may happen that the left leaf contains other search key values less than the duplicated search key value. Furthermore, it could happen that the least element on the right leaf could be the duplicated value. (This scenario could arise, for example, when the majority of data entries on the original leaf were for search keys of the duplicated value.) The parent index node (assuming the tree is of at least height 2) will have an entry for the duplicated value with a pointer to the rightmost leaf.

If this leaf continues to be filled with entries having the same duplicated key value, it could split again causing another entry with the same key value to be inserted in the parent node. Thus, the same key value could appear many times in the index nodes as well. While searching for entries with a given key value, the search should proceed by using the left-most of the entries on an index page such that the key value is less than or equal to the given key value. Moreover, on reaching the leaf level, it is possible that there are entries with the given key value (call it k) on the page to the *left* of the current leaf page, unless some entry with a smaller key value is present on this leaf page. Thus, we must scan to the left using the neighbor pointers at the leaf level until we find an entry with a key value *less than* k (or come to the beginning of the leaf pages). Then, we must scan forward along the leaf level until we find an entry with a key value *greater than* k.

Exercise 10.12 Answer Exercise 10.10 assuming that duplicates are handled without using overflow pages, using the alternative approach suggested in Section 9.7.

Answer 10.12 Answer omitted.