16. A *B*-tree* of order m is a search tree that either is empty or satisfies the following properties:
    (a) The root node has at least two and at most $2\lfloor(2m-2)/3\rfloor + 1$ children.
    (b) The remaining internal nodes have at most m and at least $\lceil(2m-1)/3\rceil$ children each.
    (c) All external nodes are on the same level.
    For a B*-tree of order m that contains N elements, show that if $x = \lceil(2m-1)/3\rceil$, then
    (a) the height, h, of the B*-tree satisfies $h \leq 1 + \log_x\{(N+1)/2\}$
    (b) the number of nodes p in the B*-tree satisifies $p \leq 1 + (N-1)/(x-1)$
    What is the average number of splits per insert if a B*-tree is built up starting from an empty B*-tree?
17. Using the splitting technique of Exercise 15, write an algorithm to insert a new element, x, into a B*-tree of order m. How many disk accesses are made in the worst case and on the average? Assume that the B-tree was initially of depth l and that it
    is maintained on a disk. Each access retrieves or writes one node.
18. Write an algorithm to delete the element whose key is x from a B*-tree of order m. What is the maximum number of accesses needed to delete from a B*-tree of depth l? Make the same assumptions as in Exercise 17.

11.3 B^+-TREES

11.3.1 Definition

A B^+-tree is a close cousin of the B-tree. The essential differences are:

(1) In a B^+-tree we have two types of nodes—index and data. The index nodes of a B^+-tree correspond to the internal nodes of a B-tree while the data nodes correspond to external nodes. The index nodes store keys (not elements) and pointers and the data nodes store elements (together with their keys but no pointers).

(2) The data nodes are linked together, in left to right order, to form a doubly linked list.

Figure 11.11 gives an example B^+-tree of order 3. The data nodes are shaded while the index nodes are not. Notice that the index nodes form a 2-3 tree whose height is 2. The capacity of a data node need not be the same as that of an index node. In Figure

11.11 each data node can hold 3 elements while each index node can hold 2 keys.

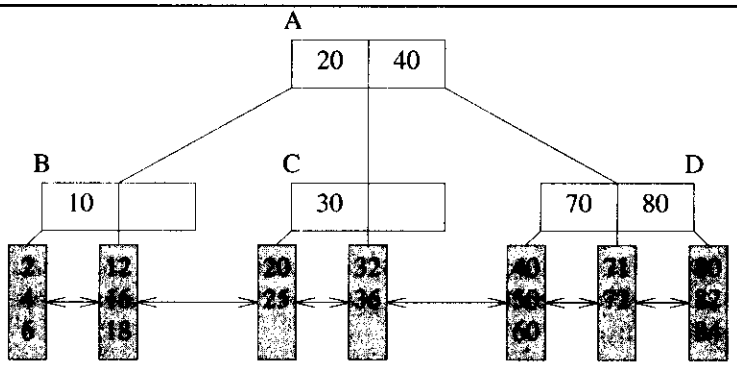


Figure 11.11: A B^+-tree of order 3

Definition: A *B^+-tree of order m* is a tree that either is empty or satisfies the following properties:

(1) All data nodes are at the same level and are leaves. Data nodes contain elements only.

(2) The index nodes define a B-tree of order m; each index node has keys but not elements.

(3) Let

$$n, A_0, (K_1,A_1), (K_2,A_2), \cdots, (K_n,A_n)$$

where the A_i, $0 \leq i \leq n < m$, are pointers to subtrees, and the K_i, $1 \leq i \leq n < m$, are keys be the format of some index node. Let $K_0 = -\infty$ and $K_{n+1} = \infty$. All elements in the subtree A_i have key less than K_{i+1} and greater than or equal to K_i, $0 \leq i \leq n$. □

11.3.2 Searching a B^+-Tree

B^+-trees support two types of searches— exact match and range. To search the tree of Figure 11.11 for the element whose key is 32, we begin at the root A, which is an index node. From the definition of a B^+-tree we know that all elements in the left subtree of A (i.e., the subtree whose root is B) have a key smaller than 20; those in the subtree with

root C have keys ≥ 20 and < 40; and those in the subtree with root D have keys ≥ 40. So, the search moves to the index node C. Since the search key is ≥ 30, the search moves from C to the data node that contains the elements with keys 32 and 36. This data node is searched and the desired element reported. Program 11.4 gives a high-level description of the algorithm to search a B+-tree.

```
/* search a B+-tree for an element with key x,
   return the element if found, return NULL otherwise */
if the tree is empty return NULL;
K0 = - MAXKEY;
for (*p = root; p is an index node; p = Ai)
{
    Let p have the format n, A0, (K1, A1), ···, (Kn, An);
    Kn+1 = MAXKEY;
    Determine i such that Ki ≤ x < Ki+1;
}
/* search the data node p */
Search p for an element E with key x;
if such an element is found return E
else return NULL;
```

Program 11.4: Searching a B+-tree

To search for all elements with keys in the range [16, 70], we proceed as in an exact match search for the start, 16, of the range. This gets us to the second data node in Figure 11.11. From here, we march down (rightward) the doubly linked list of data nodes until we reach a data node that has an element whose key exceeds the end, 70, of the search range (or until we reach the end of the list). In our example, 4 additional data nodes are examined. All examined data nodes other than the first and last contain at least one element that is in the search range.

11.3.3 Insertion into a B+-Tree

An important difference between inserting into a B-tree and inserting into a B+-tree is how we handle the splitting of a data node. When a data node becomes overfull, half the elements (those with the largest keys) are moved into a new node; the key of the smallest element so moved together with a pointer to the newly created data node are inserted into the parent index node (if any) using the insertion procedure for a B-tree. The splitting of an index node is identical to the splitting of an internal node of a B-tree.

Consider inserting an element with key 27 into the B+-tree of Figure 11.11. We

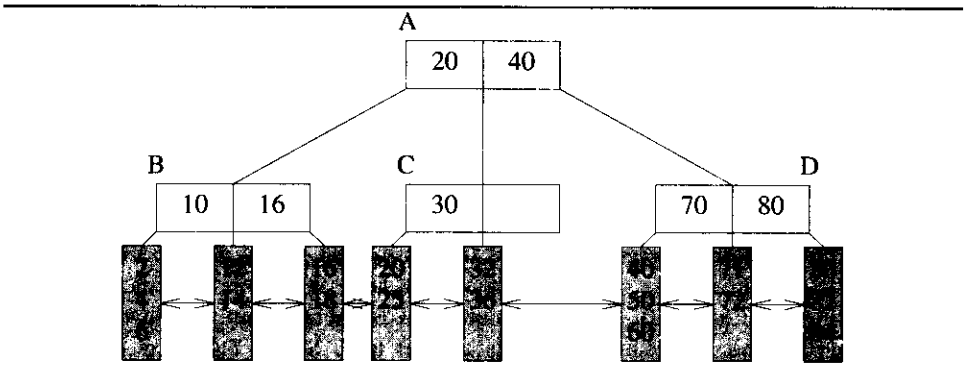
first search for this key. The search gets us to the data node that is the left child of C. Since this data node contains no element with key 27 and since this data node isn't full, we insert the new element as the third element in this data node. Next, consider the insertion of an element with key 14. The search for 14 gets us to the second data node, which is full. Symbolically inserting the new element into this full node results in an overfull node with the key sequence 12, 14, 16, 18. The overfull node is split into two by moving the largest half of the elements (those with keys 16 and 18) into a new data node, which is then inserted into the doubly linked list of data nodes. The smallest key, 16, in this new data node together with a pointer to the new data node are inserted in the parent index node B to get the configuration of Figure 11.12 (a).

Finally, consider inserting an element with key 86 into the B^+-tree of Figure 11.12 (a). The search for 86 gets us to the rightmost data node, which is full. Symbolically inserting the new element into this node results in the key sequence 80, 82, 84, 86. Splitting the overfull data node creates a new data node with the elements whose keys are 84 and 86. The new data node is inserted into the doubly linked list of data nodes. Then we insert the key 84 and a pointer to the new data node into the parent index node D, which becomes overfull. The overfull D is split using Eq. 11.5. The 84 along with two of the 4 subtrees of the overfull D are moved into a new index node E and the 80 together with a pointer to E inserted into the parent A of D. This causes A to become overfull. The overfull A is split using Eq. 11.5 and we get a new index node F that has the key 80 and 2 of the 4 subtrees of the overfull A. The key 40 together with pointers to A and F form the new root of the B^+-tree (Figure 11.12 (b)).

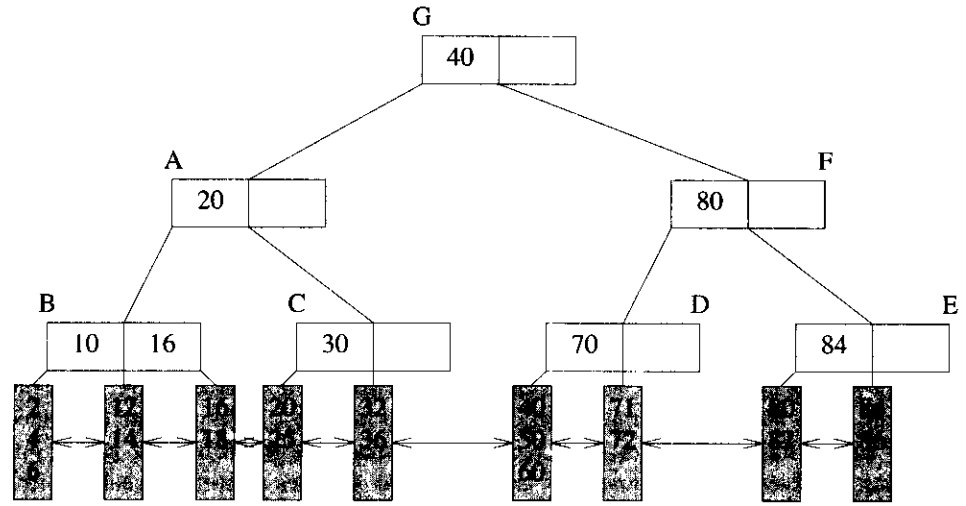
11.3.4 Deletion from a B^+-Tree

Since elements are stored only in the leaves of a B^+-tree, we need concern ourselves only with deletion from a leaf (recall that in the case of a B-tree we had to transform a deletion from a non-leaf into a deletion from a leaf; this case doesn't arise for B^+-trees). Since the index nodes of a B^+-tree form a B-tree, a non-root index node is deficient when it has fewer than $\lceil m/2 \rceil - 1$ keys and a root index node is deficient when it has no key. When is a data node deficient? The definition of a B^+-tree doesn't specify a minimum occupancy for a data node. However, we may get some guidance from our algorithm to insert an element. Following the split of an overfull data node, the original data node as well as the new one each have at least $\lceil c/2 \rceil$ elements, where c is the capacity of a data node. So, except when a data node is the root of the B^+-tree, its occupancy is at least $\lceil c/2 \rceil$. We shall say that a non-root data node is deficient iff it has fewer than $\lceil c/2 \rceil$ elements; a root data node is deficient iff it is empty.

We illustrate the deletion process by an example. Consider the B^+-tree of Figure 11.11. The capacity c of a data node is 3. So, a non-root data node is deficient iff it has fewer than 2 elements. To delete the element whose key is 40, we first search for the element to be deleted. This element is found in the data node that is the left child of the



(a) 14 inserted



(b) 86 inserted

Figure 11.12: Insertion into the B^+-tree of Figure 11.11

index node D. Following the deletion of the element with key 40, the occupancy of this data node becomes 2. So, the data node isn't deficient and we need merely write the modified data node to disk (assuming the B^+-tree is disk resident) and we are done. Notice that when the deletion of an element doesn't result in a deficient data node, no index node is changed.

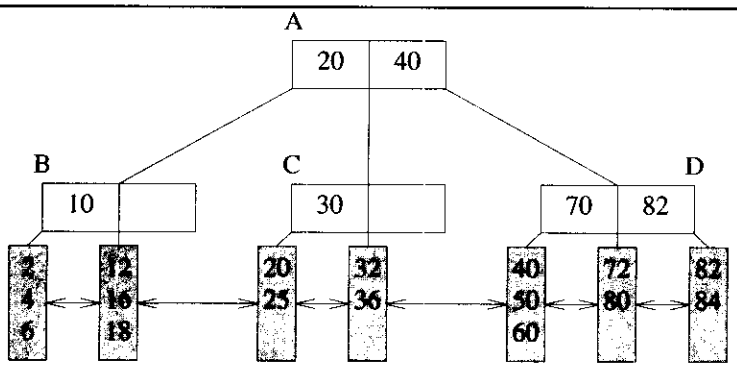
Next consider the deletion of the element whose key is 71 from the B^+-tree of

Figure 11.11. This element is found in the middle child of D. Following its deletion, the middle child of D becomes deficient. We check either its nearest right or nearest left sibling and determine whether the checked sibling has more than the required minimum number ($\lceil c/2 \rceil$) of elements. Suppose we check the nearest right sibling, which has the key sequence 80, 82, 84. Since this node has an excess element, we borrow the smallest and update the in-between key in the parent D from 80 to that of the smallest remaining element in the right sibling, 82. Figure 11.13 (a) shows the result. For a disk-resident B^+-tree, this deletion would require us to write out one altered index node (D) and two altered data nodes. For data nodes with larger capacity, when a data node becomes deficient, we may borrow several elements from a nearest sibling that has excess elements. For example, when $c=10$, a deficient data node will have 4 elements and its nearest sibling may have 10. We could borrow 3 elements from the nearest sibling thereby balancing the occupancy in both data nodes to 7. Such a balancing is expected to improve performance.

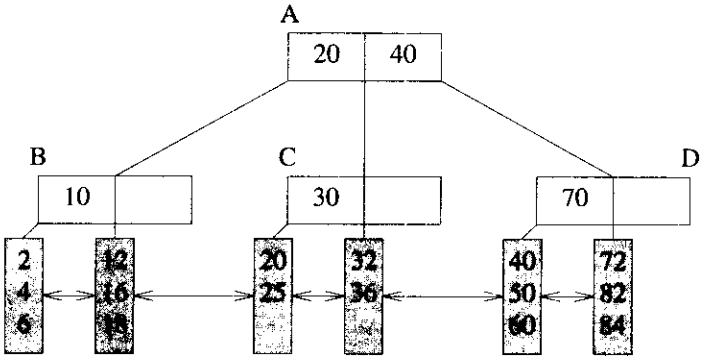
When we delete the element with key 80 from the B^+-tree of Figure 11.13 (a), the middle child of D becomes deficient. We check its nearest right sibling and discover that this sibling has only $\lceil c/2 \rceil$ elements. So, the 2 data nodes are combined into one and the in-between key 82 that is in the parent index node D deleted. Figure 11.13 (b) shows the resulting B^+-tree. Notice that combining two data nodes into one requires the deletion of a data node from the doubly linked list of data nodes. Note also that in the case of a disk resident B^+-tree, the just performed deletion requires us to write out one altered data node (the middle child of D) and one altered index node (D).

As another example for deletion, consider deleting the element with key 32 from the B^+-tree of Figure 11.12 (b). This element is in the middle child of C. Following the deletion, the middle child becomes deficient. Since its nearest sibling has only $\lceil c/2 \rceil$ elements, we cannot borrow from it. Instead, we combine the two data nodes deleting one from its doubly linked list and delete the in-between key (30) in the parent. Figure 11.14 (a) shows the result. As we can see, the index node C now has become deficient. When an index node becomes deficient, we examine a nearest sibling. If the examined nearest sibling has excess keys, we balance the occupancy of the two index nodes; this balancing involves moving some keys and associated subtrees as well as changing the in-between key in the parent. For our example, the in-between key 20 is moved from A to C, the rightmost key 16 of B is moved to A, and the right subtree of B moved to C. Figure 11.14 (b) shows the resulting B^+-tree.

As a final example, consider the deletion of the element with key 86 from the B^+-tree of Figure 11.12 (b). The middle child of E becomes deficient and is combined with its sibling; a data node is deleted from the doubly linked list of data nodes and the in-between key 84 in the parent also is deleted. This results in a deficient index node E and the configuration of Figure 11.15 (a). The deficient index node E combines with its sibling index node D and the in-between key 80 to get the configuration of Figure 11.15 (b). Finally, the deficient index node F combines with its sibling A and the in-between key 40 in its parent G. This causes the parent G, which is the root, to become deficient.



(a) 71 deleted from Figure 11.11



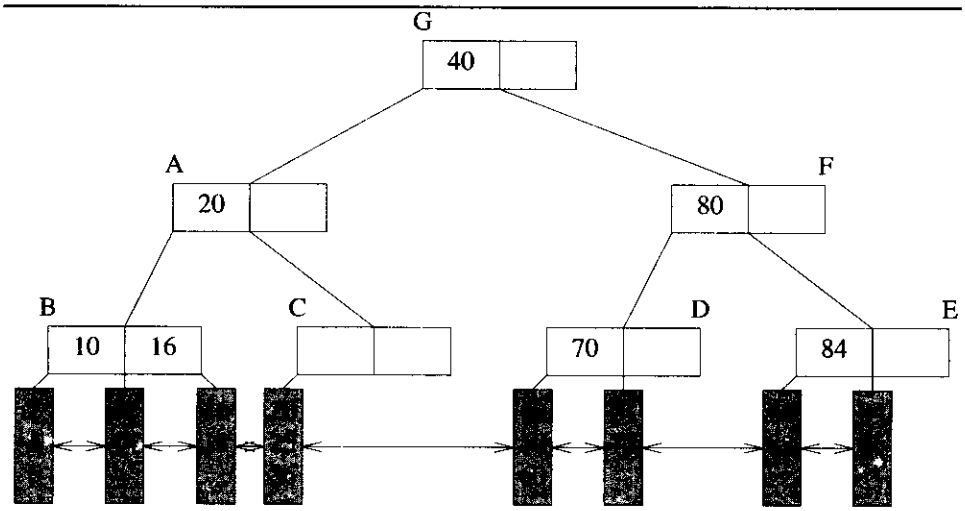
(b) 80 deleted from (a)

Figure 11.13: Deletion from a B^+-tree

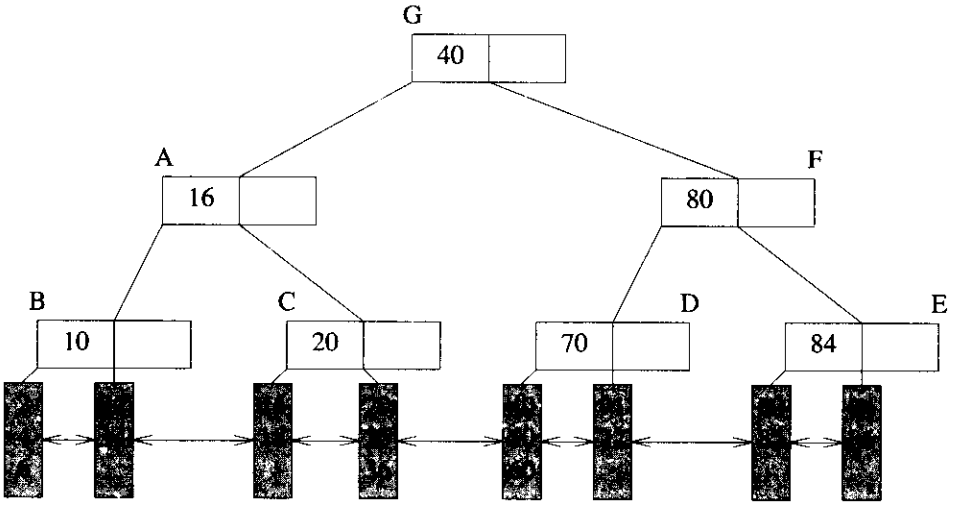
The deficient root is discarded and we get the B^+-tree of Figure 11.12 (a). In the case of a disk resident B^+-tree, the deletion of 86 would require us to write to disk one altered data node and 2 altered index nodes (A and D).

EXERCISES

1. Into the B^+-tree of Figure 11.11 insert elements with keys 5, 38, 45, 11 and 81 (in this order). Use the insertion method described in the text. Draw the B^+-tree following each insert.

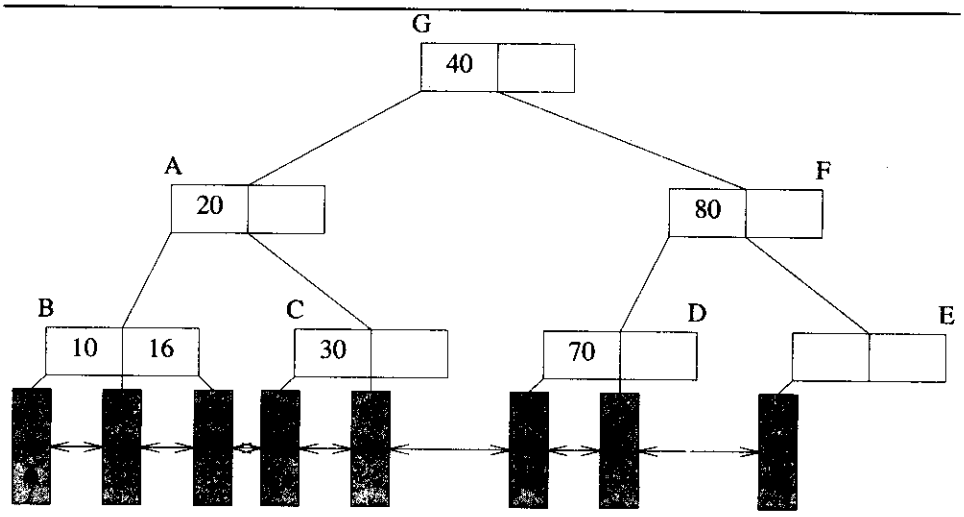


(a) C is deficient

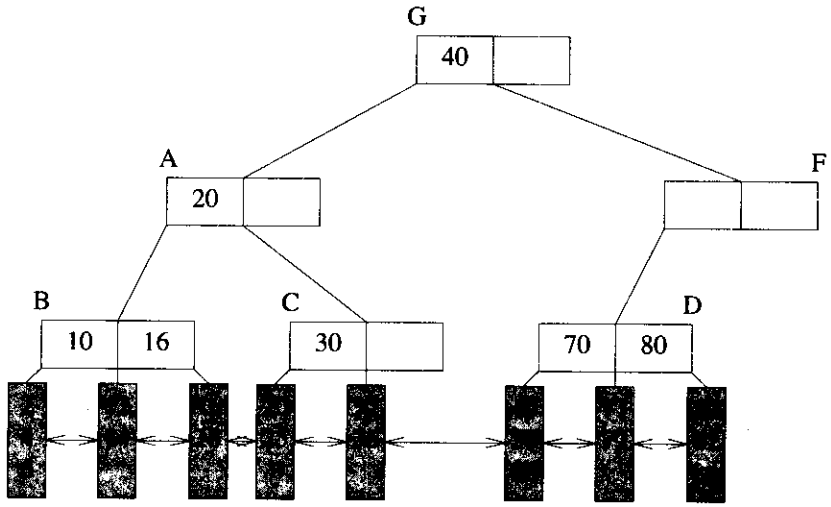


(b) After borrowing from B

Figure 11.14: Stages in deleting 32 from the B^+-tree of Figure 11.12 (b)



(a) E becomes deficient



(b) F becomes deficient

Figure 11.15: Stages in deleting 86 from the B^+-tree of Figure 11.12 (b)

2. Provide a high-level description (similar to Program 11.4) of the algorithm to insert into a B^+-tree.
3. Suppose that a B^+-tree whose height is h is disk resident. How many disk accesses are needed, in the worst case, to insert a new element? Assume that each node may be read/written with a single access and that we have sufficient memory to save the h nodes accessed in the search phase so that these nodes don't have to be re-read during the bottom-up node splitting phase.
4. From the B^+-tree of Figure 11.12 (b) delete the elements with keys 6, 71, 14, 18, 16 and 2 (in this order). Use the deletion method described in the text. Show the B^+-tree following each delete.
5. Provide a high-level description (similar to Program 11.4) of the algorithm to delete from a B^+-tree.
6. Suppose that a B^+-tree whose height is h is disk resident. How many disk accesses are needed, in the worst case, to delete an element? Assume that each node may be read/written with a single access and that we have sufficient memory to save the h nodes accessed in the search phase so that these nodes don't have to be re-read during the bottom-up borrow and combine phase.
7. Discuss the merits/demerits of replacing the doubly linked list of data nodes in a B^+-tree by a singly linked list.
8. Program B^+-tree functions for exact and range search as well as for insert and delete. Test all functions using your own test data.

11.4 REFERENCES AND SELECTED READINGS

B-trees were invented by Bayer and McCreight. For further reading on B-trees and their variants, see "Organization and maintenance of large ordered indices," by R. Bayer and E. McCreight, *Acta Informatica*, 1972; *The art of computer programming, Vol. 3, Sorting and Searching*, Second Edition, by D. Knuth, Addison Wesley, 1997; "The ubiquitous B-tree," by D. Comer, *ACM Computing Surveys*, 1979; and " B trees," by D. Zhang, in *Handbook of data structures and applications*, D. Mehta and S. Sahni editors, Chapman & Hall/CRC, 2005.