AUTOMATIC CREATION OF INDICES

If a relation is declared to have a primary key, most database implementations automatically create an index on the primary key. Whenever a tuple is inserted into the relation, the index can be used to check that the primary key constraint is not violated (that is, there are no duplicates on the primary key value). Without the index on the primary key, whenever a tuple is inserted, the entire relation would have to be read to ensure that the primary-key constraint is satisfied.

Because secondary-key order and physical-key order differ, if we attempt to scan the file sequentially in secondary-key order, the reading of each record is likely to require the reading of a new block from disk, which is very slow.

The procedure described earlier for deletion and insertion can also be applied to secondary indices; the actions taken are those described for dense indices storing a pointer to every record in the file. If a file has multiple indices, whenever the file is modified, *every* index must be updated.

Secondary indices improve the performance of queries that use keys other than the search key of the clustering index. However, they impose a significant overhead on modification of the database. The designer of a database decides which secondary indices are desirable on the basis of an estimate of the relative frequency of queries and modifications.

11.2.5 Indices on Multiple Keys

Although the examples we have seen so far have had a single attribute in a search key, in general a search key can have more than one attribute. A search key containing more than one attribute is referred to as a **composite search key**. The structure of the index is the same as that of any other index, the only difference being that the search key is not a single attribute, but rather is a list of attributes. The search key can be represented as a tuple of values, of the form $(a_1, \ldots, a_n)$, where the indexed attributes are $A_1, \ldots, A_n$. The ordering of search-key values is the *lexicographic ordering*. For example, for the case of two attribute search keys, $(a_1, a_2) < (b_1, b_2)$ if either $a_1 < b_1$ or $a_1 = b_1$ and $a_2 < b_2$. Lexicographic ordering is basically the same as alphabetic ordering of words.

As an example, consider an index on the *takes* relation, on the composite search key (*course_id*, *semester*, *year*). Such an index would be useful to find all students who have registered for a particular course in a particular semester/year. An ordered index on a composite key can also be used to answer several other kinds of queries efficiently, as we shall see later in Section 11.5.2.

11.3 B+-Tree Index Files

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans

through the data. Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable.

The **B$^+$-tree** index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B$^+$-tree index takes the form of a **balanced tree** in which every path from the root of the tree to a leaf of the tree is of the same length. Each nonleaf node in the tree has between $\lceil n/2 \rceil$ and n children, where n is fixed for a particular tree.

We shall see that the B$^+$-tree structure imposes performance overhead on insertion and deletion, and adds space overhead. The overhead is acceptable even for frequently modified files, since the cost of file reorganization is avoided. Furthermore, since nodes may be as much as half empty (if they have the minimum number of children), there is some wasted space. This space overhead, too, is acceptable given the performance benefits of the B$^+$-tree structure.

11.3.1 Structure of a B$^+$-Tree

A B$^+$-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. Figure 11.7 shows a typical node of a B$^+$-tree. It contains up to $n - 1$ search-key values $K_1, K_2, \ldots, K_{n-1}$, and n pointers $P_1, P_2, \ldots, P_n$. The search-key values within a node are kept in sorted order; thus, if $i < j$, then $K_i < K_j$.

We consider first the structure of the **leaf nodes**. For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i. Pointer P_n has a special purpose that we shall discuss shortly.

Figure 11.8 shows one leaf node of a B$^+$-tree for the *instructor* file, in which we have chosen n to be 4, and the search key is *name*.

Now that we have seen the structure of a leaf node, let us consider how search-key values are assigned to particular nodes. Each leaf can hold up to $n - 1$ values. We allow leaf nodes to contain as few as $\lceil (n - 1)/2 \rceil$ values. With $n = 4$ in our example B$^+$-tree, each leaf must contain at least 2 values, and at most 3 values.

The ranges of values in each leaf do not overlap, except if there are duplicate search-key values, in which case a value may be present in more than one leaf. Specifically, if L_i and L_j are leaf nodes and $i < j$, then every search-key value in L_i is less than or equal to every search-key value in L_j. If the B$^+$-tree index is used as a dense index (as is usually the case) every search-key value must appear in some leaf node.

Now we can explain the use of the pointer P_n. Since there is a linear order on the leaves based on the search-key values that they contain, we use P_n to chain

Figure 11.7 Typical node of a B$^+$-tree.

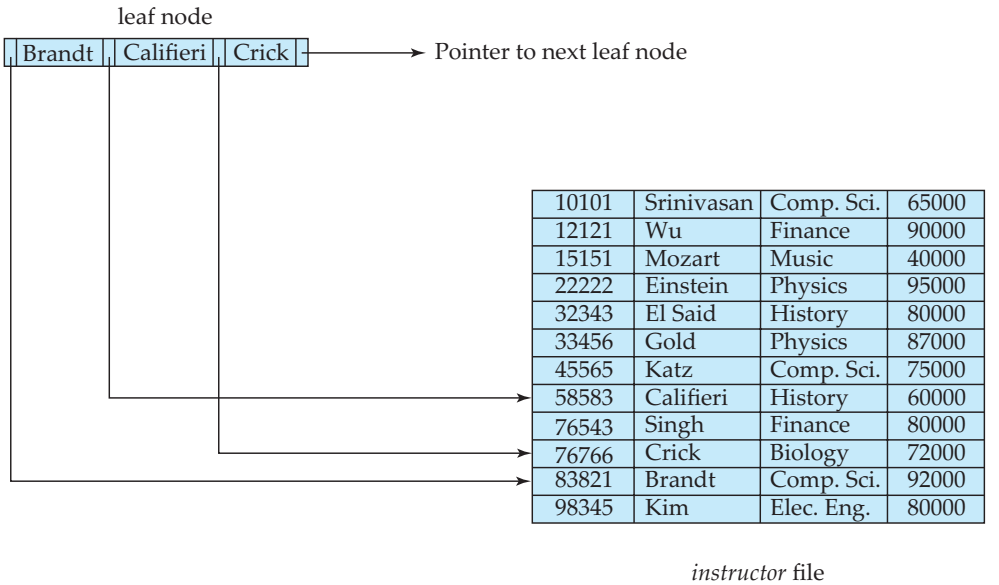

Figure 11.8 A leaf node for *instructor* B+-tree index ($n = 4$).

together the leaf nodes in search-key order. This ordering allows for efficient sequential processing of the file.

The **nonleaf nodes** of the B+-tree form a multilevel (sparse) index on the leaf nodes. The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A nonleaf node may hold up to n pointers, and *must* hold at least $\lceil n/2 \rceil$ pointers. The number of pointers in a node is called the *fanout* of the node. Nonleaf nodes are also referred to as **internal nodes**.

Let us consider a node containing m pointers ($m \leq n$). For $i = 2, 3, \ldots, m-1$, pointer P_i points to the subtree that contains search-key values less than K_i and greater than or equal to K_{i-1}. Pointer P_m points to the part of the subtree that contains those key values greater than or equal to K_{m-1}, and pointer P_1 points to the part of the subtree that contains those search-key values less than K_1.

Unlike other nonleaf nodes, the root node can hold fewer than $\lceil n/2 \rceil$ pointers; however, it must hold at least two pointers, unless the tree consists of only one node. It is always possible to construct a B+-tree, for any n, that satisfies the preceding requirements.

Figure 11.9 shows a complete B+-tree for the *instructor* file (with $n = 4$). We have shown instructor names abbreviated to 3 characters in order to depict the tree clearly; in reality, the tree nodes would contain the full names. We have also omitted null pointers for simplicity; any pointer field in the figure that does not have an arrow is understood to have a null value.

Figure 11.10 shows another B+-tree for the *instructor* file, this time with $n = 6$. As before, we have abbreviated instructor names only for clarity of presentation.

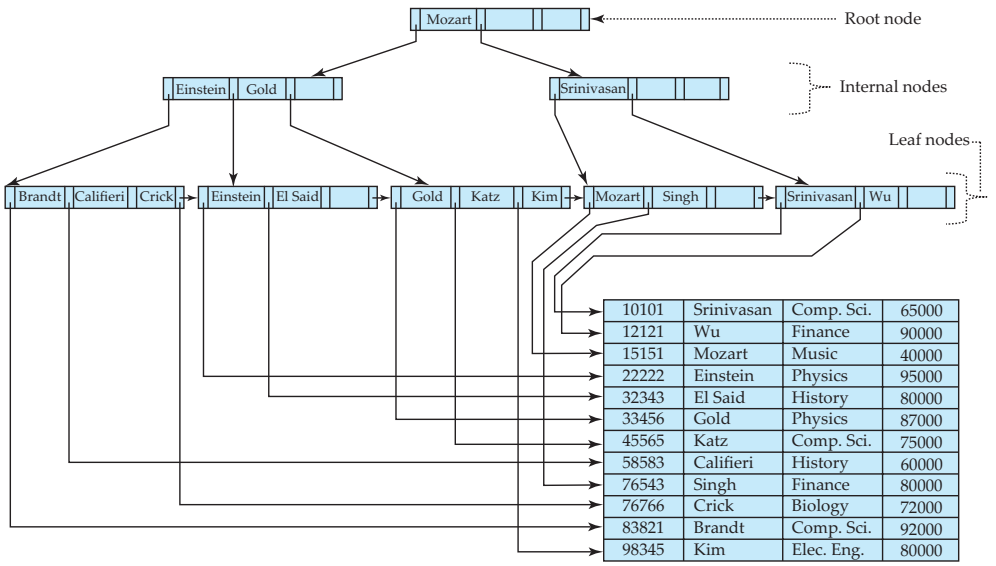

Figure 11.9 B$^+$-tree for *instructor* file ($n = 4$).

Observe that the height of this tree is less than that of the previous tree, which had $n = 4$.

These examples of B$^+$-trees are all balanced. That is, the length of every path from the root to a leaf node is the same. This property is a requirement for a B$^+$-tree. Indeed, the "B" in B$^+$-tree stands for "balanced." It is the balance property of B$^+$-trees that ensures good performance for lookup, insertion, and deletion.

11.3.2 Queries on B$^+$-Trees

Let us consider how we process queries on a B$^+$-tree. Suppose that we wish to find records with a search-key value of V. Figure 11.11 presents pseudocode for a function `find()` to carry out this task.

Intuitively, the function starts at the root of the tree, and traverses the tree down until it reaches a leaf node that would contain the specified value if it exists in the tree. Specifically, starting with the root as the current node, the function repeats the following steps until a leaf node is reached. First, the current node is examined, looking for the smallest i such that search-key value K_i is greater

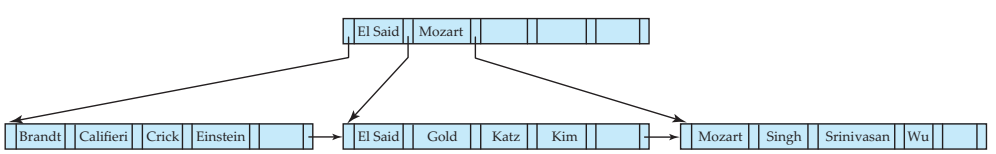

Figure 11.10 B$^+$-tree for *instructor* file with $n = 6$.

```
function find(value V)
/* Returns leaf node C and index i such that C.P_i points to first record
* with search key value V */
    Set C = root node
    while (C is not a leaf node) begin
        Let i = smallest number such that V ≤ C.K_i
        if there is no such number i then begin
            Let P_m = last non-null pointer in the node
            Set C = C.P_m
        end
        else if (V = C.K_i)
            then Set C = C.P_{i+1}
        else C = C.P_i /* V < C.K_i */
    end
    /* C is a leaf node */
    Let i be the least value such that K_i = V
    if there is such a value i
        then return (C, i)
        else return null ; /* No record with key value V exists*/


procedure printAll(value V)
/* prints all records with search key value V */
    Set done = false;
    Set (L, i) = find(V);
    if ((L, i) is null) return
    repeat
        repeat
            Print record pointed to by L.P_i
            Set i = i + 1
        until (i > number of keys in L or L.K_i > V)
        if (i > number of keys in L)
            then L = L.P_n
            else Set done = true;
    until (done or L is null)
```

Figure 11.11 Querying a B+-tree.

than or equal to V. Suppose such a value is found; then, if K_i is equal to V, the current node is set to the node pointed to by P_{i+1}, otherwise $K_i > V$, and the current node is set to the node pointed to by P_i. If no such value K_i is found, then clearly $V > K_{m-1}$, where P_m is the last nonnull pointer in the node. In this case the current node is set to that pointed to by P_m. The above procedure is repeated, traversing down the tree until a leaf node is reached.

At the leaf node, if there is a search-key value equal to V, let K_i be the first such value; pointer P_i directs us to a record with search-key value K_i. The function

then returns the leaf node L and the index i. If no search-key with value V is found in the leaf node, no record with key value V exists in the relation, and function `find` returns null, to indicate failure.

If there is at most one record with a search key value V (for example, if the index is on a primary key) the procedure that calls the `find` function simply uses the pointer $L.P_i$ to retrieve the record and is done. However, in case there may be more than one matching record, the remaining records also need to be fetched.

Procedure `printAll` shown in Figure 11.11 shows how to fetch all records with a specified search key V. The procedure first steps through the remaining keys in the node L, to find other records with search-key value V. If node L contains at least one search-key value greater than V, then there are no more records matching V. Otherwise, the next leaf, pointed to by P_n may contain further entries for V. The node pointed to by P_n must then be searched to find further records with search-key value V. If the highest search-key value in the node pointed to by P_n is also V, further leaves may have to be traversed, in order to find all matching records. The **repeat** loop in `printAll` carries out the task of traversing leaf nodes until all matching records have been found.

A real implementation would provide a version of `find` supporting an iterator interface similar to that provided by the JDBC `ResultSet`, which we saw in Section 5.1.1. Such an iterator interface would provide a method `next`(), which can be called repeatedly to fetch successive records with the specified search-key. The `next`() method would step through the entries at the leaf level, in a manner similar to `printAll`, but each call takes only one step, and records where it left off, so that successive calls `next` step through successive records. We omit details for simplicity, and leave the pseudocode for the iterator interface as an exercise for the interested reader.

B^+-trees can also be used to find all records with search key values in a specified range (L, U). For example, with a B^+-tree on attribute *salary* of *instructor*, we can find all *instructor* records with salary in a specified range such as $(50000, 100000)$ (in other words, all salaries between 50000 and 100000). Such queries are called **range queries**. To execute such queries, we can create a procedure `printRange`(L, U), whose body is the same as `printAll` except for these differences: `printRange` calls `find`(L), instead of `find`(V), and then steps through records as in procedure `printAll`, but with the stopping condition being that $L.K_i > U$, instead of $L.K_i > V$.

In processing a query, we traverse a path in the tree from the root to some leaf node. If there are N records in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(N) \rceil$.

In practice, only a few nodes need to be accessed. Typically, a node is made to be the same size as a disk block, which is typically 4 kilobytes. With a search-key size of 12 bytes, and a disk-pointer size of 8 bytes, n is around 200. Even with a more conservative estimate of 32 bytes for the search-key size, n is around 100. With $n = 100$, if we have 1 million search-key values in the file, a lookup requires only $\lceil \log_{50}(1{,}000{,}000) \rceil = 4$ nodes to be accessed. Thus, at most four blocks need to be read from disk for the lookup. The root node of the tree is usually heavily accessed and is likely to be in the buffer, so typically only three or fewer blocks need to be read from disk.

An important difference between B+-tree structures and in-memory tree structures, such as binary trees, is the size of a node, and as a result, the height of the tree. In a binary tree, each node is small, and has at most two pointers. In a B+-tree, each node is large—typically a disk block—and a node can have a large number of pointers. Thus, B+-trees tend to be fat and short, unlike thin and tall binary trees. In a balanced binary tree, the path for a lookup can be of length $\lceil \log_2(N) \rceil$, where N is the number of records in the file being indexed. With $N = 1{,}000{,}000$ as in the previous example, a balanced binary tree requires around 20 node accesses. If each node were on a different disk block, 20 block reads would be required to process a lookup, in contrast to the four block reads for the B+-tree. The difference is significant, since each block read could require a disk arm seek, and a block read together with the disk arm seek takes about 10 milliseconds on a typical disk.

11.3.3 Updates on B+-Trees

When a record is inserted into, or deleted from a relation, indices on the relation must be updated correspondingly. Recall that updates to a record can be modeled as a deletion of the old record followed by insertion of the updated record. Hence we only consider the case of insertion and deletion.

Insertion and deletion are more complicated than lookup, since it may be necessary to **split** a node that becomes too large as the result of an insertion, or to **coalesce** nodes (that is, combine nodes) if a node becomes too small (fewer than $\lceil n/2 \rceil$ pointers). Furthermore, when a node is split or a pair of nodes is combined, we must ensure that balance is preserved. To introduce the idea behind insertion and deletion in a B+-tree, we shall assume temporarily that nodes never become too large or too small. Under this assumption, insertion and deletion are performed as defined next.

- **Insertion**. Using the same technique as for lookup from the `find()` function (Figure 11.11), we first find the leaf node in which the search-key value would appear. We then insert an entry (that is, a search-key value and record pointer pair) in the leaf node, positioning it such that the search keys are still in order.
- **Deletion**. Using the same technique as for lookup, we find the leaf node containing the entry to be deleted, by performing a lookup on the search-key value of the deleted record; if there are multiple entries with the same search-key value, we search across all entries with the same search-key value until we find the entry that points to the record being deleted. We then remove the entry from the leaf node. All entries in the leaf node that are to the right of the deleted entry are shifted left by one position, so that there are no gaps in the entries after the entry is deleted.

We now consider the general case of insertion and deletion, dealing with node splitting and node coalescing.

.

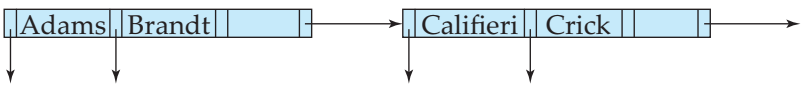

Figure 11.12 Split of leaf node on insertion of "Adams"

11.3.3.1 Insertion

We now consider an example of insertion in which a node must be split. Assume that a record is inserted on the *instructor* relation, with the *name* value being Adams. We then need to insert an entry for "Adams" into the B$^+$-tree of Figure 11.9. Using the algorithm for lookup, we find that "Adams" should appear in the leaf node containing "Brandt", "Califieri", and "Crick." There is no room in this leaf to insert the search-key value "Adams." Therefore, the node is *split* into two nodes. Figure 11.12 shows the two leaf nodes that result from the split of the leaf node on inserting "Adams". The search-key values "Adams" and "Brandt" are in one leaf, and "Califieri" and "Crick" are in the other. In general, we take the n search-key values (the $n - 1$ values in the leaf node plus the value being inserted), and put the first $\lceil n/2 \rceil$ in the existing node and the remaining values in a newly created node.

Having split a leaf node, we must insert the new leaf node into the B$^+$-tree structure. In our example, the new node has "Califieri" as its smallest search-key value. We need to insert an entry with this search-key value, and a pointer to the new node, into the parent of the leaf node that was split. The B$^+$-tree of Figure 11.13 shows the result of the insertion. It was possible to perform this insertion with no further node split, because there was room in the parent node for the new entry. If there were no room, the parent would have had to be split, requiring an entry to be added to its parent. In the worst case, all nodes along the path to the root must be split. If the root itself is split, the entire tree becomes deeper.

Splitting of a nonleaf node is a little different from splitting of a leaf node. Figure 11.14 shows the result of inserting a record with search key "Lamport" into the tree shown in Figure 11.13. The leaf node in which "Lamport" is to be inserted already has entries "Gold", "Katz", and "Kim", and as a result the leaf node has to be split. The new right-hand-side node resulting from the split contains the search-key values "Kim" and "Lamport". An entry (Kim, $n1$) must then be added

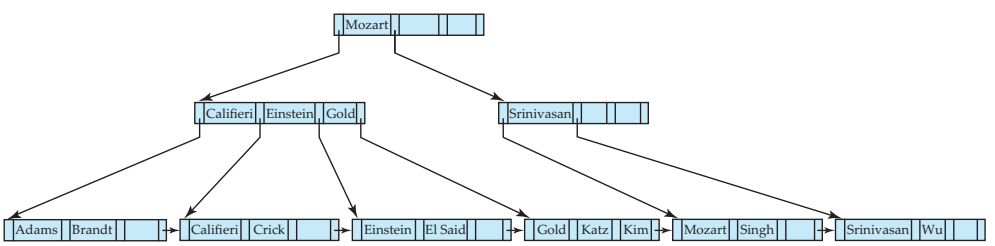

Figure 11.13 Insertion of "Adams" into the B$^+$-tree of Figure 11.9.

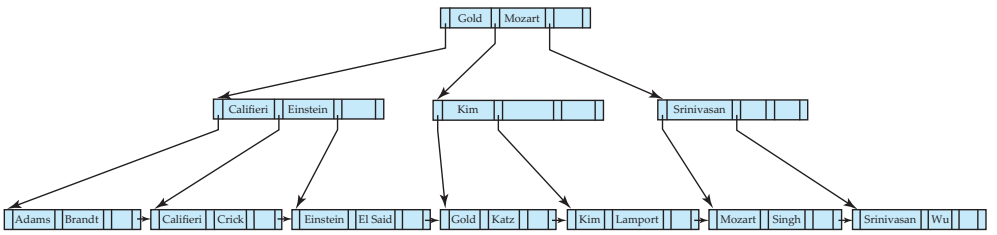

Figure 11.14 Insertion of "Lamport" into the B+-tree of Figure 11.13.

to the parent node, where $n1$ is a pointer to the new node, However, there is no space in the parent node to add a new entry, and the parent node has to be split. To do so, the parent node is conceptually expanded temporarily, the entry added, and the overfull node is then immediately split.

When an overfull nonleaf node is split, the child pointers are divided among the original and the newly created nodes; in our example, the original node is left with the first three pointers, and the newly created node to the right gets the remaining two pointers. The search key values are, however, handled a little differently. The search key values that lie between the pointers moved to the right node (in our example, the value "Kim") are moved along with the pointers, while those that lie between the pointers that stay on the left (in our example, "Califieri" and "Einstein") remain undisturbed.

However, the search key value that lies between the pointers that stay on the left, and the pointers that move to the right node is treated differently. In our example, the search key value "Gold" lies between the three pointers that went to the left node, and the two pointers that went to the right node. The value "Gold" is not added to either of the split nodes. Instead, an entry (Gold, $n2$) is added to the parent node, where $n2$ is a pointer to the newly created node that resulted from the split. In this case, the parent node is the root, and it has enough space for the new entry.

The general technique for insertion into a B+-tree is to determine the leaf node l into which insertion must occur. If a split results, insert the new node into the parent of node l. If this insertion causes a split, proceed recursively up the tree until either an insertion does not cause a split or a new root is created.

Figure 11.15 outlines the insertion algorithm in pseudocode. The procedure `insert` inserts a key-value pointer pair into the index, using two subsidiary procedures `insert_in_leaf` and `insert_in_parent`. In the pseudocode, L, N, P and T denote pointers to nodes, with L being used to denote a leaf node. $L.K_i$ and $L.P_i$ denote the ith value and the ith pointer in node L, respectively; $T.K_i$ and $T.P_i$ are used similarly. The pseudocode also makes use of the function `parent`(N) to find the parent of a node N. We can compute a list of nodes in the path from the root to the leaf while initially finding the leaf node, and can use it later to find the parent of any node in the path efficiently.

The procedure `insert_in_parent` takes as parameters N, K', N', where node N was split into N and N', with K' being the least value in N'. The procedure

```
procedure insert(value K, pointer P)
    if (tree is empty) create an empty leaf node L, which is also the root
    else Find the leaf node L that should contain key value K
    if (L has less than n − 1 key values)
        then insert_in_leaf (L, K, P)
        else begin /* L has n − 1 key values already, split it */
            Create node L′
            Copy L.P1 . . . L.Kn−1 to a block of memory T that can
                 hold n (pointer, key-value) pairs
            insert_in_leaf (T, K, P)
            Set L′.Pn = L.Pn; Set L.Pn = L′
            Erase L.P1 through L.Kn−1 from L
            Copy T.P1 through T.K⌈n/2⌉ from T into L starting at L.P1
            Copy T.P⌈n/2⌉+1 through T.Kn from T into L′ starting at L′.P1
            Let K′ be the smallest key-value in L′
            insert_in_parent(L, K′, L′)
        end

procedure insert_in_leaf (node L, value K, pointer P)
    if (K < L.K1)
        then insert P, K into L just before L.P1
        else begin
            Let Ki be the highest value in L that is less than K
            Insert P, K into L just after T.Ki
        end

procedure insert_in_parent(node N, value K′, node N′)
    if (N is the root of the tree)
        then begin
            Create a new node R containing N, K′, N′   /* N and N′ are pointers */
            Make R the root of the tree
            return
        end
    Let P = parent (N)
    if (P has less than n pointers)
        then insert (K′, N′) in P just after N
        else begin /* Split P */
            Copy P to a block of memory T that can hold P and (K′, N′)
            Insert (K′, N′) into T just after N
            Erase all entries from P; Create node P′
            Copy T.P1 . . . T.P⌈n/2⌉ into P
            Let K″ = T.K⌈n/2⌉
            Copy T.P⌈n/2⌉+1 . . . T.Pn+1 into P′
            insert_in_parent(P, K″, P′)
        end
```

Figure 11.15 Insertion of entry in a B$^+$-tree.

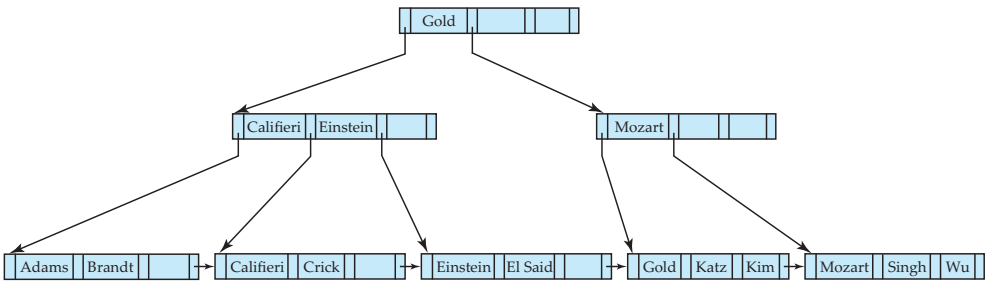

Figure 11.16 Deletion of "Srinivasan" from the B+-tree of Figure 11.13.

modifies the parent of N to record the split. The procedures `insert_into_index` and `insert_in_parent` use a temporary area of memory T to store the contents of a node being split. The procedures can be modified to copy data from the node being split directly to the newly created node, reducing the time required for copying data. However, the use of the temporary space T simplifies the procedures.

11.3.3.2 Deletion

We now consider deletions that cause tree nodes to contain too few pointers. First, let us delete "Srinivasan" from the B+-tree of Figure 11.13. The resulting B+-tree appears in Figure 11.16. We now consider how the deletion is performed. We first locate the entry for "Srinivasan" by using our lookup algorithm. When we delete the entry for "Srinivasan" from its leaf node, the node is left with only one entry, "Wu". Since, in our example, $n = 4$ and $1 < \lceil (n-1)/2 \rceil$, we must either merge the node with a sibling node, or redistribute the entries between the nodes, to ensure that each node is at least half-full. In our example, the underfull node with the entry for "Wu" can be merged with its left sibling node. We merge the nodes by moving the entries from both the nodes into the left sibling, and deleting the now empty right sibling. Once the node is deleted, we must also delete the entry in the parent node that pointed to the just deleted node.

In our example, the entry to be deleted is (Srinivasan, $n3$), where $n3$ is a pointer to the leaf containing "Srinivasan". (In this case the entry to be deleted in the nonleaf node happens to be the same value as that deleted from the leaf; that would not be the case for most deletions.) After deleting the above entry, the parent node, which had a search key value "Srinivasan" and two pointers, now has one pointer (the leftmost pointer in the node) and no search-key values. Since $1 < \lceil n/2 \rceil$ for $n = 4$, the parent node is underfull. (For larger n, a node that becomes underfull would still have some values as well as pointers.)

In this case, we look at a sibling node; in our example, the only sibling is the nonleaf node containing the search keys "Califieri", "Einstein", and "Gold". If possible, we try to coalesce the node with its sibling. In this case, coalescing is not possible, since the node and its sibling together have five pointers, against a maximum of four. The solution in this case is to **redistribute** the pointers between

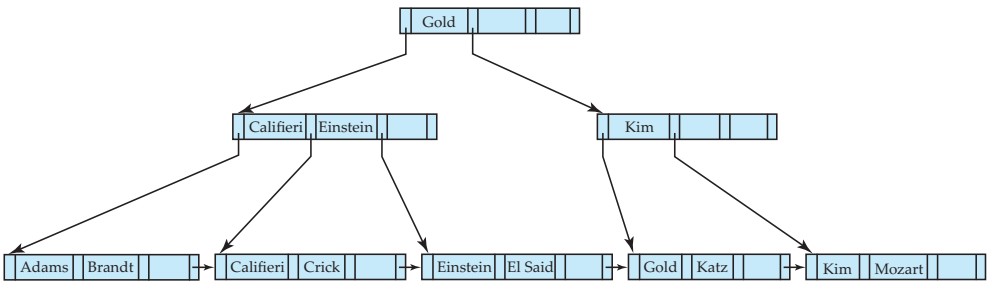

Figure 11.17 Deletion of "Singh" and "Wu" from the B$^+$-tree of Figure 11.16.

the node and its sibling, such that each has at least $\lceil n/2 \rceil = 2$ child pointers. To do so, we move the rightmost pointer from the left sibling (the one pointing to the leaf node containing "Mozart") to the underfull right sibling. However, the underfull right sibling would now have two pointers, namely its leftmost pointer, and the newly moved pointer, with no value separating them. In fact, the value separating them is not present in either of the nodes, but is present in the parent node, between the pointers from the parent to the node and its sibling. In our example, the value "Mozart" separates the two pointers, and is present in the right sibling after the redistribution. Redistribution of the pointers also means that the value "Mozart" in the parent no longer correctly separates search-key values in the two siblings. In fact, the value that now correctly separates search-key values in the two sibling nodes is the value "Gold", which was in the left sibling before redistribution.

As a result, as can be seen in the B$^+$-tree in Figure 11.16, after redistribution of pointers between siblings, the value "Gold" has moved up into the parent, while the value that was there earlier, "Mozart", has moved down into the right sibling.

We next delete the search-key values "Singh" and "Wu" from the B$^+$-tree of Figure 11.16. The result is shown in Figure 11.17. The deletion of the first of these values does not make the leaf node underfull, but the deletion of the second value does. It is not possible to merge the underfull node with its sibling, so a redistribution of values is carried out, moving the search-key value "Kim" into the node containing "Mozart", resulting in the tree shown in Figure 11.17. The value separating the two siblings has been updated in the parent, from "Mozart" to "Kim".

Now we delete "Gold" from the above tree; the result is shown in Figure 11.18. This results in an underfull leaf, which can now be merged with its sibling. The resultant deletion of an entry from the parent node (the nonleaf node containing "Kim") makes the parent underfull (it is left with just one pointer). This time around, the parent node can be merged with its sibling. This merge results in the search-key value "Gold" moving down from the parent into the merged node. As a result of this merge, an entry is deleted from its parent, which happens to be the root of the tree. And as a result of that deletion, the root is left with only one child pointer and no search-key value, violating the condition that the root

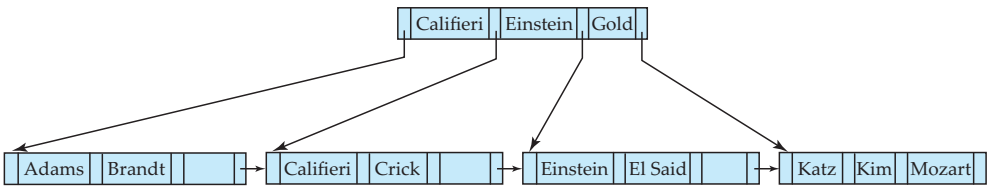

Figure 11.18 Deletion of "Gold" from the B+-tree of Figure 11.17.

have at least two children. As a result, the root node is deleted and its sole child becomes the root, and the depth of the B+-tree has been decreased by 1.

It is worth noting that, as a result of deletion, a key value that is present in a nonleaf node of the B+-tree may not be present at any leaf of the tree. For example, in Figure 11.18, the value "Gold" has been deleted from the leaf level, but is still present in a nonleaf node.

In general, to delete a value in a B+-tree, we perform a lookup on the value and delete it. If the node is too small, we delete it from its parent. This deletion results in recursive application of the deletion algorithm until the root is reached, a parent remains adequately full after deletion, or redistribution is applied.

Figure 11.19 outlines the pseudocode for deletion from a B+-tree. The procedure `swap_variables`(N, N') merely swaps the values of the (pointer) variables N and N'; this swap has no effect on the tree itself. The pseudocode uses the condition "too few pointers/values." For nonleaf nodes, this criterion means less than $\lceil n/2 \rceil$ pointers; for leaf nodes, it means less than $\lceil (n-1)/2 \rceil$ values. The pseudocode redistributes entries by borrowing a single entry from an adjacent node. We can also redistribute entries by repartitioning entries equally between the two nodes. The pseudocode refers to deleting an entry (K, P) from a node. In the case of leaf nodes, the pointer to an entry actually precedes the key value, so the pointer P precedes the key value K. For nonleaf nodes, P follows the key value K.

11.3.4 Nonunique Search Keys

If a relation can have more than one record containing the same search key value (that is, two or more records can have the same values for the indexed attributes), the search key is said to be a **nonunique search key**.

One problem with nonunique search keys is in the efficiency of record deletion. Suppose a particular search-key value occurs a large number of times, and one of the records with that search key is to be deleted. The deletion may have to search through a number of entries, potentially across multiple leaf nodes, to find the entry corresponding to the particular record being deleted.

A simple solution to this problem, used by most database systems, is to make search keys unique by creating a composite search key containing the original search key and another attribute, which together are unique across all records. The extra attribute can be a record-id, which is a pointer to the record, or any other attribute whose value is unique among all records with the same search-

```
procedure delete(value K, pointer P)
    find the leaf node L that contains (K, P)
    delete_entry(L, K, P)

procedure delete_entry(node N, value K, pointer P)
    delete (K, P) from N
    if (N is the root and N has only one remaining child)
    then make the child of N the new root of the tree and delete N
    else if (N has too few values/pointers) then begin
        Let N' be the previous or next child of parent(N)
        Let K' be the value between pointers N and N' in parent(N)
        if (entries in N and N' can fit in a single node)
            then begin /* Coalesce nodes */
                if (N is a predecessor of N') then swap_variables(N, N')
                if (N is not a leaf)
                    then append K' and all pointers and values in N to N'
                    else append all (K_i, P_i) pairs in N to N'; set N'.P_n = N.P_n
                delete_entry(parent(N), K', N); delete node N
            end
        else begin /* Redistribution: borrow an entry from N' */
            if (N' is a predecessor of N) then begin
                if (N is a nonleaf node) then begin
                    let m be such that N'.P_m is the last pointer in N'
                    remove (N'.K_{m-1}, N'.P_m) from N'
                    insert (N'.P_m, K') as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_{m-1}
                end
                else begin
                    let m be such that (N'.P_m, N'.K_m) is the last pointer/value
                        pair in N'
                    remove (N'.P_m, N'.K_m) from N'
                    insert (N'.P_m, N'.K_m) as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_m
                end
            end
            else ... symmetric to the then case ...
        end
    end
```

Figure 11.19 Deletion of entry from a B$^+$-tree.

key value. The extra attribute is called a **uniquifier** attribute. When a record is to be deleted, the composite search-key value is computed from the record, and then used to look up the index. Since the value is unique, the corresponding leaf-

level entry can be found with a single traversal from root to leaf, with no further accesses at the leaf level. As a result, record deletion can be done efficiently.

A search with the original search-key attribute simply ignores the value of the uniquifier attribute when comparing search-key values.

With nonunique search keys, our B+-tree structure stores each key value as many times as there are records containing that value. An alternative is to store each key value only once in the tree, and to keep a bucket (or list) of record pointers with a search-key value, to handle nonunique search keys. This approach is more space efficient since it stores the key value only once; however, it creates several complications when B+-trees are implemented. If the buckets are kept in the leaf node, extra code is needed to deal with variable-size buckets, and to deal with buckets that grow larger than the size of the leaf node. If the buckets are stored in separate blocks, an extra I/O operation may be required to fetch records. In addition to these problems, the bucket approach also has the problem of inefficiency for record deletion if a search-key value occurs a large number of times.

11.3.5 Complexity of B+-Tree Updates

Although insertion and deletion operations on B+-trees are complicated, they require relatively few I/O operations, which is an important benefit since I/O operations are expensive. It can be shown that the number of I/O operations needed in the worst case for an insertion is proportional to $\log_{\lceil n/2 \rceil}(N)$, where n is the maximum number of pointers in a node, and N is the number of records in the file being indexed.

The worst-case complexity of the deletion procedure is also proportional to $\log_{\lceil n/2 \rceil}(N)$, provided there are no duplicate values for the search key. If there are duplicate values, deletion may have to search across multiple records with the same search-key value to find the correct entry to be deleted, which can be inefficient. However, making the search key unique by adding a uniquifier attribute, as described in Section 11.3.4, ensures the worst-case complexity of deletion is the same even if the original search key is nonunique.

In other words, the cost of insertion and deletion operations in terms of I/O operations is proportional to the height of the B+-tree, and is therefore low. It is the speed of operation on B+-trees that makes them a frequently used index structure in database implementations.

In practice, operations on B+-trees result in fewer I/O operations than the worst-case bounds. With fanout of 100, and assuming accesses to leaf nodes are uniformly distributed, the parent of a leaf node is 100 times more likely to get accessed than the leaf node. Conversely, with the same fanout, the total number of nonleaf nodes in a B+-tree would be just a little more than 1/100th of the number of leaf nodes. As a result, with memory sizes of several gigabytes being common today, for B+-trees that are used frequently, even if the relation is very large it is quite likely that most of the nonleaf nodes are already in the database buffer when they are accessed. Thus, typically only one or two I/O operations are required to perform a lookup. For updates, the probability of a node split

occurring is correspondingly very small. Depending on the ordering of inserts, with a fanout of 100, only between 1 in 100 to 1 in 50 insertions will result in a node split, requiring more than one block to be written. As a result, on an average an insert will require just a little more than one I/O operation to write updated blocks.

Although B^+-trees only guarantee that nodes will be at least half full, if entries are inserted in random order, nodes can be expected to be more than two-thirds full on average. If entries are inserted in sorted order, on the other hand, nodes will be only half full. (We leave it as an exercise to the reader to figure out why nodes would be only half full in the latter case.)

11.4 B^+-Tree Extensions

In this section, we discuss several extensions and variations of the B^+-tree index structure.

11.4.1 B^+-Tree File Organization

As mentioned in Section 11.3, the main drawback of index-sequential file organization is the degradation of performance as the file grows: With growth, an increasing percentage of index entries and actual records become out of order, and are stored in overflow blocks. We solve the degradation of index lookups by using B^+-tree indices on the file. We solve the degradation problem for storing the actual records by using the leaf level of the B^+-tree to organize the blocks containing the actual records. We use the B^+-tree structure not only as an index, but also as an organizer for records in a file. In a **B^+-tree file organization**, the leaf nodes of the tree store records, instead of storing pointers to records. Figure 11.20 shows an example of a B^+-tree file organization. Since records are usually larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node. However, the leaf nodes are still required to be at least half full.

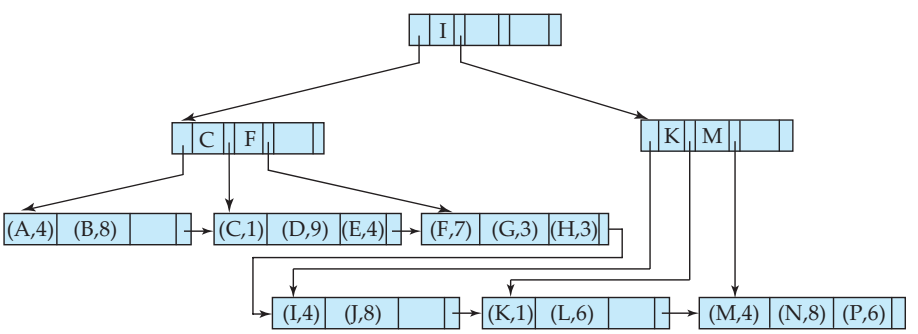

Figure 11.20 B^+-tree file organization.

Insertion and deletion of records from a B+-tree file organization are handled in the same way as insertion and deletion of entries in a B+-tree index. When a record with a given key value v is inserted, the system locates the block that should contain the record by searching the B+-tree for the largest key in the tree that is $\leq v$. If the block located has enough free space for the record, the system stores the record in the block. Otherwise, as in B+-tree insertion, the system splits the block in two, and redistributes the records in it (in the B+-tree–key order) to create space for the new record. The split propagates up the B+-tree in the normal fashion. When we delete a record, the system first removes it from the block containing it. If a block B becomes less than half full as a result, the records in B are redistributed with the records in an adjacent block B'. Assuming fixed-sized records, each block will hold at least one-half as many records as the maximum that it can hold. The system updates the nonleaf nodes of the B+-tree in the usual fashion.

When we use a B+-tree for file organization, space utilization is particularly important, since the space occupied by the records is likely to be much more than the space occupied by keys and pointers. We can improve the utilization of space in a B+-tree by involving more sibling nodes in redistribution during splits and merges. The technique is applicable to both leaf nodes and nonleaf nodes, and works as follows:

During insertion, if a node is full the system attempts to redistribute some of its entries to one of the adjacent nodes, to make space for a new entry. If this attempt fails because the adjacent nodes are themselves full, the system splits the node, and splits the entries evenly among one of the adjacent nodes and the two nodes that it obtained by splitting the original node. Since the three nodes together contain one more record than can fit in two nodes, each node will be about two-thirds full. More precisely, each node will have at least $\lfloor 2n/3 \rfloor$ entries, where n is the maximum number of entries that the node can hold. ($\lfloor x \rfloor$ denotes the greatest integer that is less than or equal to x; that is, we drop the fractional part, if any.)

During deletion of a record, if the occupancy of a node falls below $\lfloor 2n/3 \rfloor$, the system attempts to borrow an entry from one of the sibling nodes. If both sibling nodes have $\lfloor 2n/3 \rfloor$ records, instead of borrowing an entry, the system redistributes the entries in the node and in the two siblings evenly between two of the nodes, and deletes the third node. We can use this approach because the total number of entries is $3\lfloor 2n/3 \rfloor - 1$, which is less than $2n$. With three adjacent nodes used for redistribution, each node can be guaranteed to have $\lfloor 3n/4 \rfloor$ entries. In general, if m nodes ($m - 1$ siblings) are involved in redistribution, each node can be guaranteed to contain at least $\lfloor (m - 1)n/m \rfloor$ entries. However, the cost of update becomes higher as more sibling nodes are involved in the redistribution.

Note that in a B+-tree index or file organization, leaf nodes that are adjacent to each other in the tree may be located at different places on disk. When a file organization is newly created on a set of records, it is possible to allocate blocks that are mostly contiguous on disk to leaf nodes that are contiguous in the tree. Thus a sequential scan of leaf nodes would correspond to a mostly sequential scan on disk. As insertions and deletions occur on the tree, sequentiality is increasingly

lost, and sequential access has to wait for disk seeks increasingly often. An index rebuild may be required to restore sequentiality.

B^+-tree file organizations can also be used to store large objects, such as SQL clobs and blobs, which may be larger than a disk block, and as large as multiple gigabytes. Such large objects can be stored by splitting them into sequences of smaller records that are organized in a B^+-tree file organization. The records can be sequentially numbered, or numbered by the byte offset of the record within the large object, and the record number can be used as the search key.

11.4.2 Secondary Indices and Record Relocation

Some file organizations, such as the B^+-tree file organization, may change the location of records even when the records have not been updated. As an example, when a leaf node is split in a B^+-tree file organization, a number of records are moved to a new node. In such cases, all secondary indices that store pointers to the relocated records would have to be updated, even though the values in the records may not have changed. Each leaf node may contain a fairly large number of records, and each of them may be in different locations on each secondary index. Thus a leaf-node split may require tens or even hundreds of I/O operations to update all affected secondary indices, making it a very expensive operation.

A widely used solution for this problem is as follows: In secondary indices, in place of pointers to the indexed records, we store the values of the primary-index search-key attributes. For example, suppose we have a primary index on the attribute *ID* of relation *instructor*; then a secondary index on *dept_name* would store with each department name a list of instructor's *ID* values of the corresponding records, instead of storing pointers to the records.

Relocation of records because of leaf-node splits then does not require any update on any such secondary index. However, locating a record using the secondary index now requires two steps: First we use the secondary index to find the primary-index search-key values, and then we use the primary index to find the corresponding records.

The above approach thus greatly reduces the cost of index update due to file reorganization, although it increases the cost of accessing data using a secondary index.

11.4.3 Indexing Strings

Creating B^+-tree indices on string-valued attributes raises two problems. The first problem is that strings can be of variable length. The second problem is that strings can be long, leading to a low fanout and a correspondingly increased tree height.

With variable-length search keys, different nodes can have different fanouts even if they are full. A node must then be split if it is full, that is, there is no space to add a new entry, regardless of how many search entries it has. Similarly, nodes can be merged or entries redistributed depending on what fraction of the space in the nodes is used, instead of being based on the maximum number of entries that the node can hold.

The fanout of nodes can be increased by using a technique called **prefix compression**. With prefix compression, we do not store the entire search key value at nonleaf nodes. We only store a prefix of each search key value that is sufficient to distinguish between the key values in the subtrees that it separates. For example, if we had an index on names, the key value at a nonleaf node could be a prefix of a name; it may suffice to store "Silb" at a nonleaf node, instead of the full "Silberschatz" if the closest values in the two subtrees that it separates are, say, "Silas" and "Silver" respectively.

11.4.4 Bulk Loading of B+-Tree Indices

As we saw earlier, insertion of a record in a B+-tree requires a number of I/O operations that in the worst case is proportional to the height of the tree, which is usually fairly small (typically five or less, even for large relations).

Now consider the case where a B+-tree is being built on a large relation. Suppose the relation is significantly larger than main memory, and we are constructing a nonclustering index on the relation such that the index is also larger than main memory. In this case, as we scan the relation and add entries to the B+-tree, it is quite likely that each leaf node accessed is not in the database buffer when it is accessed, since there is no particular ordering of the entries. With such randomly ordered accesses to blocks, each time an entry is added to the leaf, a disk seek will be required to fetch the block containing the leaf node. The block will probably be evicted from the disk buffer before another entry is added to the block, leading to another disk seek to write the block back to disk. Thus a random read and a random write operation may be required for each entry inserted.

For example, if the relation has 100 million records, and each I/O operation takes about 10 milliseconds, it would take at least 1 million seconds to build the index, counting only the cost of reading leaf nodes, not even counting the cost of writing the updated nodes back to disk. This is clearly a very large amount of time; in contrast, if each record occupies 100 bytes, and the disk subsystem can transfer data at 50 megabytes per second, it would take just 200 seconds to read the entire relation.

Insertion of a large number of entries at a time into an index is referred to as **bulk loading** of the index. An efficient way to perform bulk loading of an index is as follows. First, create a temporary file containing index entries for the relation, then sort the file on the search key of the index being constructed, and finally scan the sorted file and insert the entries into the index. There are efficient algorithms for sorting large relations, which are described later in Section 12.4, which can sort even a large file with an I/O cost comparable to that of reading the file a few times, assuming a reasonable amount of main memory is available.

There is a significant benefit to sorting the entries before inserting them into the B+-tree. When the entries are inserted in sorted order, all entries that go to a particular leaf node will appear consecutively, and the leaf needs to be written out only once; nodes will never have to be read from disk during bulk load, if the B+-tree was empty to start with. Each leaf node will thus incur only one I/O operation even though many entries may be inserted into the node. If each leaf

contains 100 entries, the leaf level will contain 1 million nodes, resulting in only 1 million I/O operations for creating the leaf level. Even these I/O operations can be expected to be sequential, if successive leaf nodes are allocated on successive disk blocks, and few disk seeks would be required. With current disks, 1 millisecond per block is a reasonable estimate for mostly sequential I/O operations, in contrast to 10 milliseconds per block for random I/O operations.

We shall study the cost of sorting a large relation later, in Section 12.4, but as a rough estimate, the index which would have taken a million seconds to build otherwise, can be constructed in well under 1000 seconds by sorting the entries before inserting them into the B^+-tree, in contrast to more than 1,000,000 seconds for inserting in random order.

If the B^+-tree is initially empty, it can be constructed faster by building it bottom-up, from the leaf level, instead of using the usual insert procedure. In **bottom-up B^+-tree construction**, after sorting the entries as we just described, we break up the sorted entries into blocks, keeping as many entries in a block as can fit in the block; the resulting blocks form the leaf level of the B^+-tree. The minimum value in each block, along with the pointer to the block, is used to create entries in the next level of the B^+-tree, pointing to the leaf blocks. Each further level of the tree is similarly constructed using the minimum values associated with each node one level below, until the root is created. We leave details as an exercise for the reader.

Most database systems implement efficient techniques based on sorting of entries, and bottom-up construction, when creating an index on a relation, although they use the normal insertion procedure when tuples are added one at a time to a relation with an existing index. Some database systems recommend that if a very large number of tuples are added at once to an already existing relation, indices on the relation (other than any index on the primary key) should be dropped, and then re-created after the tuples are inserted, to take advantage of efficient bulk-loading techniques.

11.4.5 B-Tree Index Files

B-tree indices are similar to B^+-tree indices. The primary distinction between the two approaches is that a B-tree eliminates the redundant storage of search-key values. In the B^+-tree of Figure 11.13, the search keys "Califieri", "Einstein", "Gold", "Mozart", and "Srinivasan" appear in nonleaf nodes, in addition to appearing in the leaf nodes. Every search-key value appears in some leaf node; several are repeated in nonleaf nodes.

A B-tree allows search-key values to appear only once (if they are unique), unlike a B^+-tree, where a value may appear in a nonleaf node, in addition to appearing in a leaf node. Figure 11.21 shows a B-tree that represents the same search keys as the B^+-tree of Figure 11.13. Since search keys are not repeated in the B-tree, we may be able to store the index in fewer tree nodes than in the corresponding B^+-tree index. However, since search keys that appear in nonleaf nodes appear nowhere else in the B-tree, we are forced to include an additional

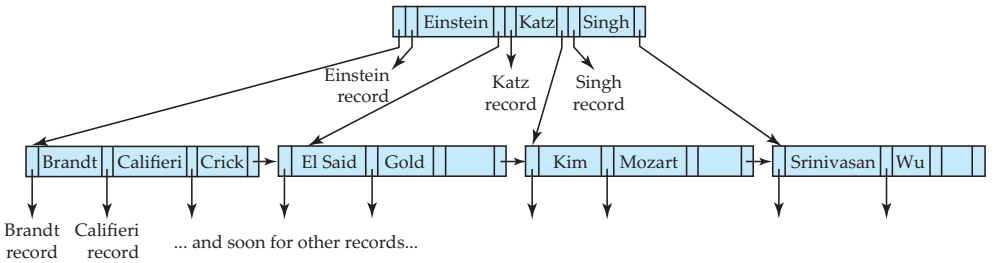

Figure 11.21 B-tree equivalent of B+-tree in Figure 11.13.

pointer field for each search key in a nonleaf node. These additional pointers point to either file records or buckets for the associated search key.

It is worth noting that many database system manuals, articles in industry literature, and industry professionals use the term B-tree to refer to the data structure that we call the B+-tree. In fact, it would be fair to say that in current usage, the term B-tree is assumed to be synonymous with B+-tree. However, in this book we use the terms B-tree and B+-tree as they were originally defined, to avoid confusion between the two data structures.

A generalized B-tree leaf node appears in Figure 11.22a; a nonleaf node appears in Figure 11.22b. Leaf nodes are the same as in B+-trees. In nonleaf nodes, the pointers P_i are the tree pointers that we used also for B+-trees, while the pointers B_i are bucket or file-record pointers. In the generalized B-tree in the figure, there are $n-1$ keys in the leaf node, but there are $m-1$ keys in the nonleaf node. This discrepancy occurs because nonleaf nodes must include pointers B_i, thus reducing the number of search keys that can be held in these nodes. Clearly, $m < n$, but the exact relationship between m and n depends on the relative size of search keys and pointers.

The number of nodes accessed in a lookup in a B-tree depends on where the search key is located. A lookup on a B+-tree requires traversal of a path from the root of the tree to some leaf node. In contrast, it is sometimes possible to find the desired value in a B-tree before reaching a leaf node. However, roughly n times as many keys are stored in the leaf level of a B-tree as in the nonleaf levels, and, since n is typically large, the benefit of finding certain values early is relatively

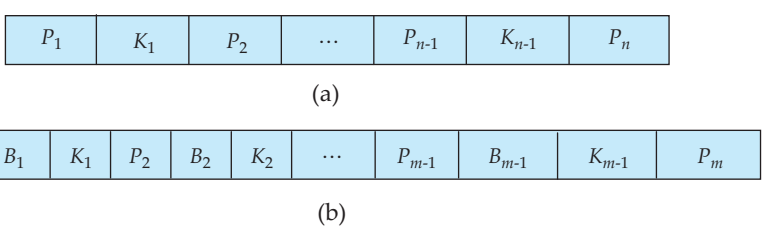

Figure 11.22 Typical nodes of a B-tree. (a) Leaf node. (b) Nonleaf node.

small. Moreover, the fact that fewer search keys appear in a nonleaf B-tree node, compared to B^+-trees, implies that a B-tree has a smaller fanout and therefore may have depth greater than that of the corresponding B^+-tree. Thus, lookup in a B-tree is faster for some search keys but slower for others, although, in general, lookup time is still proportional to the logarithm of the number of search keys.

Deletion in a B-tree is more complicated. In a B^+-tree, the deleted entry always appears in a leaf. In a B-tree, the deleted entry may appear in a nonleaf node. The proper value must be selected as a replacement from the subtree of the node containing the deleted entry. Specifically, if search key K_i is deleted, the smallest search key appearing in the subtree of pointer P_{i+1} must be moved to the field formerly occupied by K_i. Further actions need to be taken if the leaf node now has too few entries. In contrast, insertion in a B-tree is only slightly more complicated than is insertion in a B^+-tree.

The space advantages of B-trees are marginal for large indices, and usually do not outweigh the disadvantages that we have noted. Thus, pretty much all database-system implementations use the B^+-tree data structure, even if (as we discussed earlier) they refer to the data structure as a B-tree.

11.4.6 Flash Memory

In our description of indexing so far, we have assumed that data are resident on magnetic disks. Although this assumption continues to be true for the most part, flash memory capacities have grown significantly, and the cost of flash memory per gigabyte has dropped equally significantly, making flash memory storage a serious contender for replacing magnetic-disk storage for many applications. A natural question is, how would this change affect the index structure.

Flash-memory storage is structured as blocks, and the B^+-tree index structure can be used for flash-memory storage. The benefit of the much faster access speeds is clear for index lookups. Instead of requiring an average of 10 milliseconds to seek to and read a block, a random block can be read in about a microsecond from flash-memory. Thus lookups run significantly faster than with disk-based data. The optimum B^+-tree node size for flash-memory is typically smaller than that with disk.

The only real drawback with flash memory is that it does not permit in-place updates to data at the physical level, although it appears to do so logically. Every update turns into a copy+write of an entire flash-memory block, requiring the old copy of the block to be erased subsequently; a block erase takes about 1 millisecond. There is ongoing research aimed at developing index structures that can reduce the number of block erases. Meanwhile, standard B^+-tree indices can continue to be used even on flash-memory storage, with acceptable update performance, and significantly improved lookup performance compared to disk storage.

11.5 Multiple-Key Access

Until now, we have assumed implicitly that only one index on one attribute is used to process a query on a relation. However, for certain types of queries, it is