

Indexing

Motivation

Many queries reference only a small proportion of the records in a file. For example, a query like “Find all instructors in the Physics department” or “Find the total number of credits earned by the student with ID 22201” references only a fraction of the student records. It is inefficient for the system to read every tuple in the instructor relation to check if the dept_name value is “Physics”. Likewise, it is inefficient to read the entire student relation just to find the one tuple for the ID “32556,”. Ideally, the system should be able to locate these records directly. To allow these forms of access, we design additional structures that we associate with files.

Basic Concepts

An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by a word or a phrase) in the textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information for which we are looking. The words in the index are in sorted order, making it easy to find the word we want. Moreover, the index is much smaller than the book, further reducing the effort needed.

Database-system indices play the same role as book indices in libraries. For example, to retrieve a student record given an ID, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate student record.

Keeping a sorted list of students’ ID would not work well on very large databases with thousands of students, since the index would itself be very big; further, even though keeping the index sorted reduces the search time, finding a student can still be rather time-consuming. Instead, more sophisticated indexing techniques may be used. We shall discuss several of these techniques.

There are two basic kinds of indices:

- Ordered indices: Based on a sorted ordering of the values.
- Hash indices: Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a hash function.

We shall consider several techniques for both ordered indexing and hashing. No one technique is the best. Rather, each technique is best suited to particular database applications. Each technique must be evaluated on the basis of these factors:

- Access types
The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
- Access time
The time it takes to find a particular data item, or set of items, using the technique in question.
- Insertion time
The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
- Deletion time
The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
- Space overhead
The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

We often want to have more than one index for a file. For example, we may wish to search for a book by author, by subject, or by title.

An attribute or set of attributes used to look up records in a file is called a **search key**. Note that this definition of key differs from that used in primary key, candidate key, and superkey. This duplicate meaning for key is (unfortunately) well established in practice. Using our notion of a search key, we see that if there are several indices on a file, there are several search keys.

Ordered Indices

To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it.

The records in the indexed file may themselves be stored in some sorted order, just as books in a library are stored according to some attribute such as the Dewey decimal number. A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a clustering index is an index whose search key also defines the sequential order of the file. Clustering indices are also called primary indices; the term primary index may appear to denote an index on a primary key, but such indices can in fact be built on any search key. The search key of a clustering index is often the primary key, although that is not necessarily so. Indices whose search key specifies an order different from the sequential order of the file are called nonclustering indices, or secondary indices. The terms “clustered” and “nonclustered” are often used in place of “clustering” and “nonclustering.”

In Dense-Sparse Indices and index update, we assume that all files are ordered sequentially on some search key. Such files, with a clustering index on the search key, are called index-sequential files. In the example of Figure 11.1, the records are stored in sorted order of instructor ID, which is used as the search key.

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	




Figure 11.1 Sequential file for *instructor* records.

Dense and Sparse Indices

An index entry, or index record, consists of a search-key value and pointers to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.

There are two types of ordered indices that we can use:

- **Dense index:** In a dense index, an index entry appears for every search-key value in the file. In a dense clustering index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search-key value would be stored sequentially after the first record, since, because the index is a clustering one, records are sorted on the same search key. In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value.

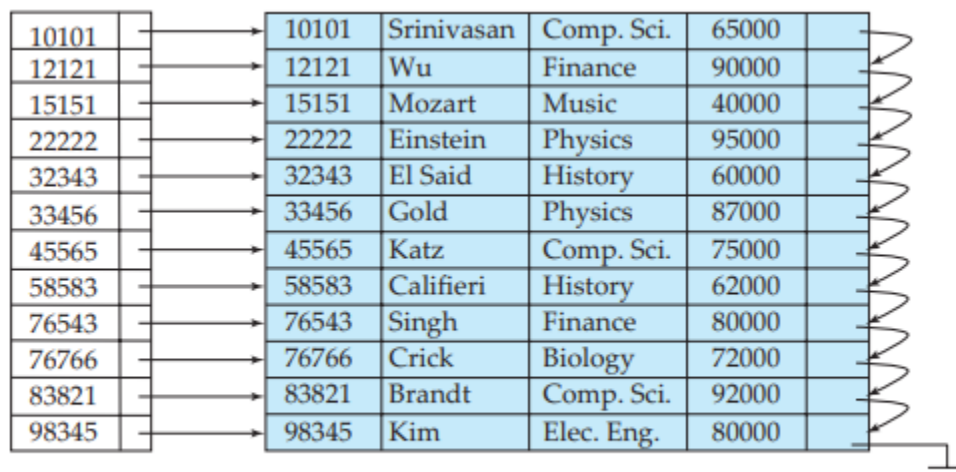


Figure 11.2 Dense index.

- **Sparse index:** In a sparse index, an index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key, that is, if the index is a clustering index. As is true in dense indices, each index entry contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.

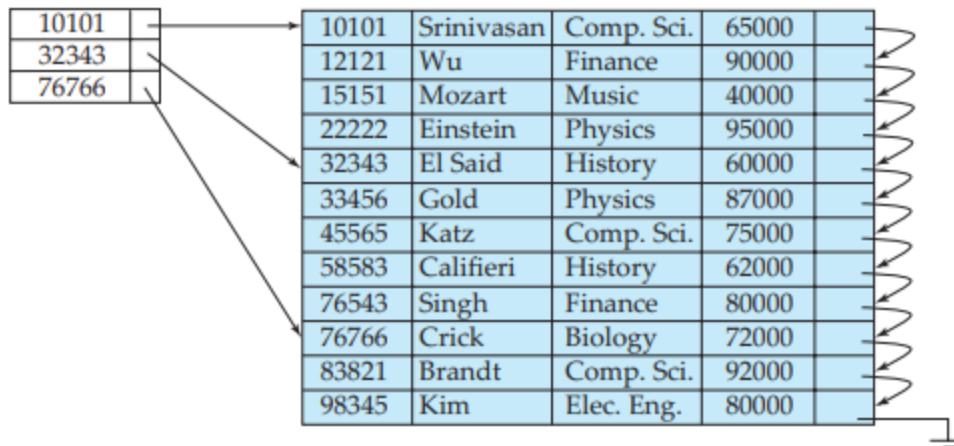


Figure 11.3 Sparse index.

Figures 11.2 and 11.3 show dense and sparse indices, respectively, for the instructor file. Suppose that we are looking up the record of instructor with ID “22222”. Using the dense index of Figure 11.2, we follow the pointer directly to the desired record. Since ID is a primary key, there exists only one such record and the search is complete. If we are using the sparse index (Figure 11.3), we do not find an index entry for “22222”. Since the last entry (in numerical order) before “22222” is “10101”, we follow that pointer. We then read the instructor file in sequential order until we find the desired record.

Consider a (printed) dictionary. The header of each page lists the first word alphabetically on that page. The words at the top of each page of the book index together form a sparse index on the contents of the dictionary pages.

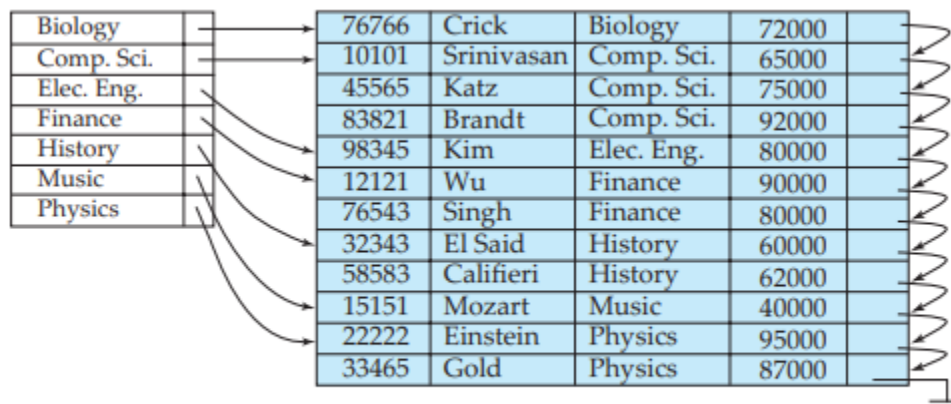


Figure 11.4 Dense index with search key *dept_name*.

As another example, suppose that the search-key value is not a primary key. Figure 11.4 shows a dense clustering index for the instructor file with the search key being *dept_name*. Observe that in this case the instructor file is sorted on the search key

dept_name, instead of ID, otherwise the index on dept_name would be a nonclustering index. Suppose that we are looking up records for the History department. Using the dense index of Figure 11.4, we follow the pointer directly to the first History record. We process this record, and follow the pointer in that record to locate the next record in search-key (dept_name) order. We continue processing records until we encounter a record for a department other than History.

As we have seen, it is generally faster to locate a record if we have a dense index rather than a sparse index. However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.

There is a trade-off that the system designer must make between access time and space overhead. Although the decision regarding this trade-off depends on the specific application, a good compromise is to have a sparse index with one index entry per block. The reason this design is a good trade-off is that the dominant cost in processing a database request is the time that it takes to bring a block from disk into main memory. Once we have brought in the block, the time to scan the entire block is negligible. Using this sparse index, we locate the block containing the record that we are seeking. Thus, unless the record is on an overflow block (see Section 10.6.1), we minimize block accesses while keeping the size of the index (and thus our space overhead) as small as possible.

For the preceding technique to be fully general, we must consider the case where records for one search-key value occupy several blocks. It is easy to modify our scheme to handle this situation.

Should we create sparse index on nonclustering index? No.

Multilevel Indices

Suppose we build a dense index on a relation with 1,000,000 tuples. Index entries are smaller than data records, so let us assume that 100 index entries fit on a 4 kilobyte block. Thus, our index occupies 10,000 blocks. If the relation instead had 100,000,000 tuples, the index would instead occupy 1,000,000 blocks, or 4 gigabytes of space. Such large indices are stored as sequential files on disk.

If an index is small enough to be kept entirely in main memory, the search time to find an entry is low. However, if the index is so large that not all of it can be kept in memory, index blocks must be fetched from disk when required. (Even if an index is smaller than the main memory of a computer, main memory is also required for a number of other tasks, so it may not be possible to keep the entire index in memory.) The search for an entry in the index then requires several disk-block reads.

Binary search can be used on the index file to locate an entry, but the search still has a large cost. If the index would occupy b blocks, binary search requires as many as $\text{ceiling}(\log_2(b))$ blocks to be read. ($\text{ceiling}(x)$ denotes the least integer that is greater than or equal to x ; that is, we round upward.) For a 10,000-block index, binary search requires 14 block reads. On a disk system where a block read takes on average 10 milliseconds, the index search will take 140 milliseconds. This may not seem much, but we would be able to carry out only seven index searches a second, whereas a more efficient search mechanism would let us carry out far more searches per second, as we shall see shortly. Note that, if overflow blocks have been used, binary search is not possible. In that case, a sequential search is typically used, and that requires b block reads, which will take even longer. Thus, the process of searching a large index may be costly.

To deal with this problem, we treat the index just as we would treat any other sequential file, and construct a sparse outer index on the original index, which we now call the inner index, as shown in Figure 11.5. Note that the index entries are always in sorted order, allowing the outer index to be sparse. To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less than or equal to the one that we desire. The pointer points to a block of the inner index. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking.

In our example, an inner index with 10,000 blocks would require 10,000 entries in the outer index, which would occupy just 100 blocks. If we assume that the outer index is

already in main memory, we would read only one index block for a search using a multilevel index, rather than the 14 blocks we read with binary search. As a result, we can perform 14 times as many index searches per second.

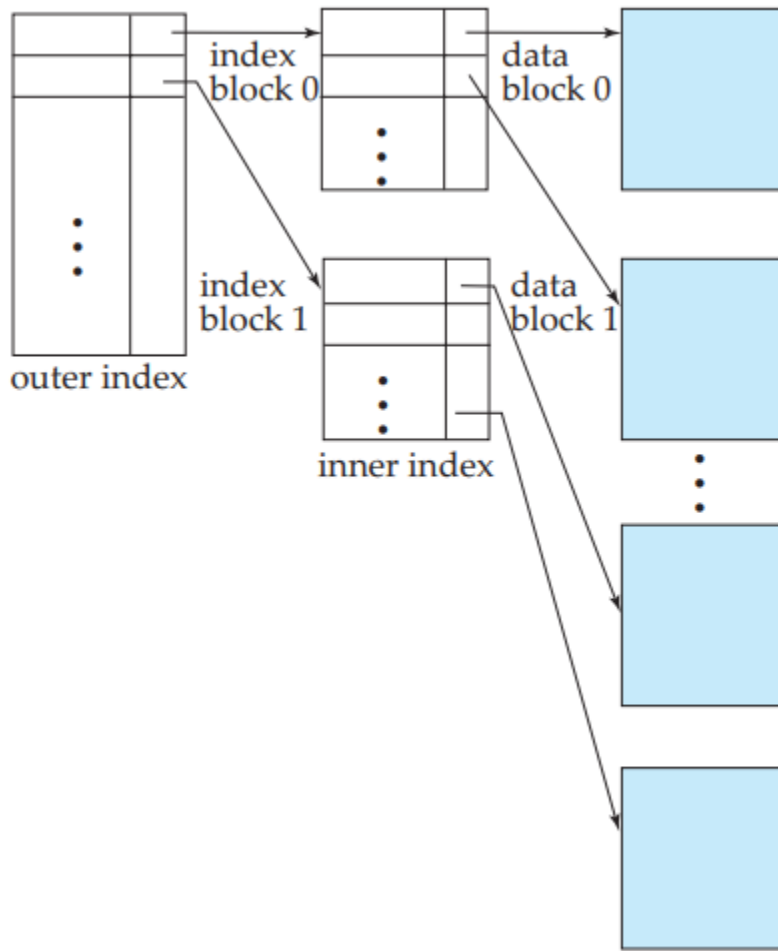


Figure 11.5 Two-level sparse index.

If our file is extremely large, even the outer index may grow too large to fit in main memory. With a 100,000,000 tuple relation, the inner index would occupy 1,000,000 blocks, and the outer index occupies 10,000 blocks, or 40 megabytes. Since there are many demands on main memory, it may not be possible to reserve that much main memory just for this particular outer index. In such a case, we can create yet another level of index. Indeed, we can repeat this process as many times as necessary. Indices with two or more levels are called **multilevel** indices. Searching for records with a multilevel index requires significantly fewer I/O operations than does searching for records by binary search.

Multilevel indices are closely related to tree structures, such as the binary trees used for in-memory indexing. We shall examine the relationship later

Index Update

Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file. Further, in case a record in the file is updated, any index whose search-key attribute is affected by the update must also be updated; for example, if the department of an instructor is changed, an index on the dept_name attribute of instructor must be updated correspondingly. Such a record update can be modeled as a deletion of the old record, followed by an insertion of the new value of the record, which results in an index deletion followed by an index insertion. As a result we only need to consider insertion and deletion on an index, and do not need to consider updates explicitly. We first describe algorithms for updating single-level indices.

Insertion

First, the system performs a lookup using the search-key value that appears in the record to be inserted. The actions the system takes next depend on whether the index is dense or sparse:

- Dense indices
 1. If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position.
 2. Otherwise the following actions are taken:
 - a. If the index entry stores pointers to all records with the same search-key value, the system adds a pointer to the new record in the index entry.
 - b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.
- Sparse indices

We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into the index. On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.

Deletion

To delete a record, the system first looks up the record to be deleted. The actions the system takes next depend on whether the index is dense or sparse:

- Dense indices
 1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index entry from the index.
 2. Otherwise the following actions are taken:
 - a. If the index entry stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index entry.
 - b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index entry to point to the next record.
- Sparse indices
 1. If the index does not contain an index entry with the search-key value of the deleted record, nothing needs to be done to the index.
 2. Otherwise the system takes the following actions:
 - a. If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
 - b. Otherwise, if the index entry for the search-key value points to the record being deleted, the system updates the index entry to point to the next record with the same search-key value.

Insertion and deletion algorithms for multilevel indices are a simple extension of the scheme just described. On deletion or insertion, the system updates the lowest-level index as described. As far as the second level is concerned, the lowest-level index is merely a file containing records—thus, if there is any change in the lowest-level index, the system updates the second-level index as described. The same technique applies to further levels of the index, if there are any.

Secondary Indices

Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. A clustering index may be sparse, storing only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file, as described earlier. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.

A secondary index on a candidate key looks just like a dense clustering index, except that the records pointed to by successive values in the index are not stored sequentially. In general, however, secondary indices may have a different structure from clustering indices. If the search key of a clustering index is not a candidate key, it suffices if the index points to the first record with a particular value for the search key, since the other records can be fetched by a sequential scan of the file.

In contrast, if the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value. The remaining records with the same search-key value could be anywhere in the file, since the records are ordered by the search key of the clustering index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.

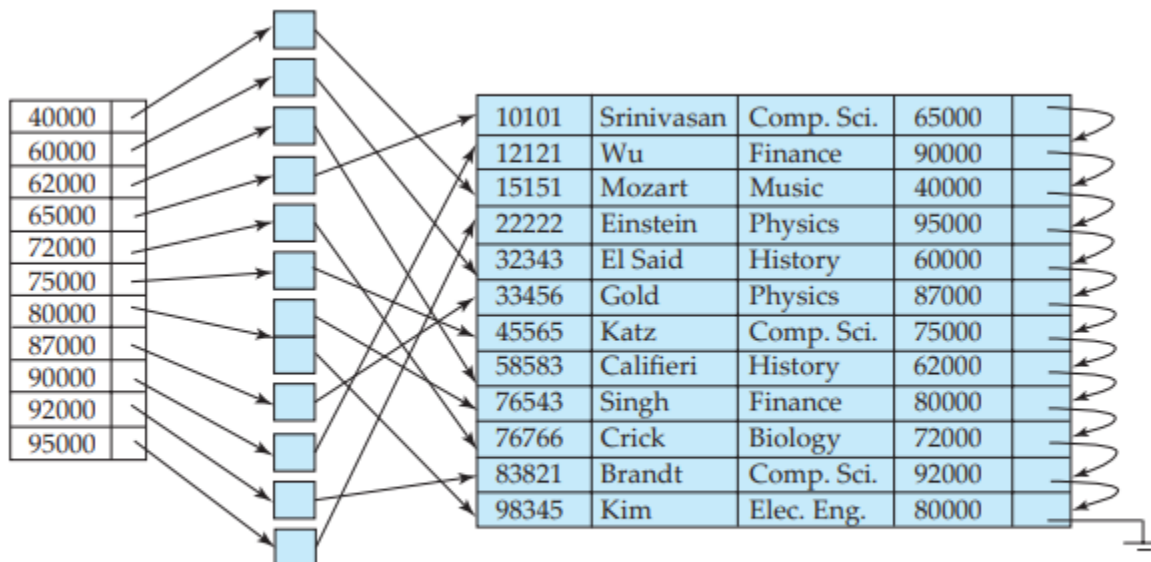


Figure 11.6 Secondary index on *instructor* file, on noncandidate key *salary*.

We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file. Figure 11.6 shows the structure of a secondary index that uses an extra level of indirection on the instructor file, on the search key salary.

A sequential scan in clustering index order is efficient because records in the file are stored physically in the same order as the index order. However, we cannot (except in rare special cases) store a file physically ordered by both the search key of the clustering index and the search key of a secondary index.

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.

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.

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, \dots, a_n) , where the indexed attributes are

A_1, \dots, 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

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 $\text{ceiling}(n/2)$ 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.

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.

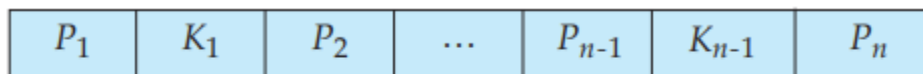


Figure 11.7 Typical node of a B⁺-tree.

It contains up to $n - 1$ search-key values K_1, K_2, \dots, K_{n-1} , and n pointers P_1, P_2, \dots, 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 $i = 1, 2, \dots, n-1$, 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.

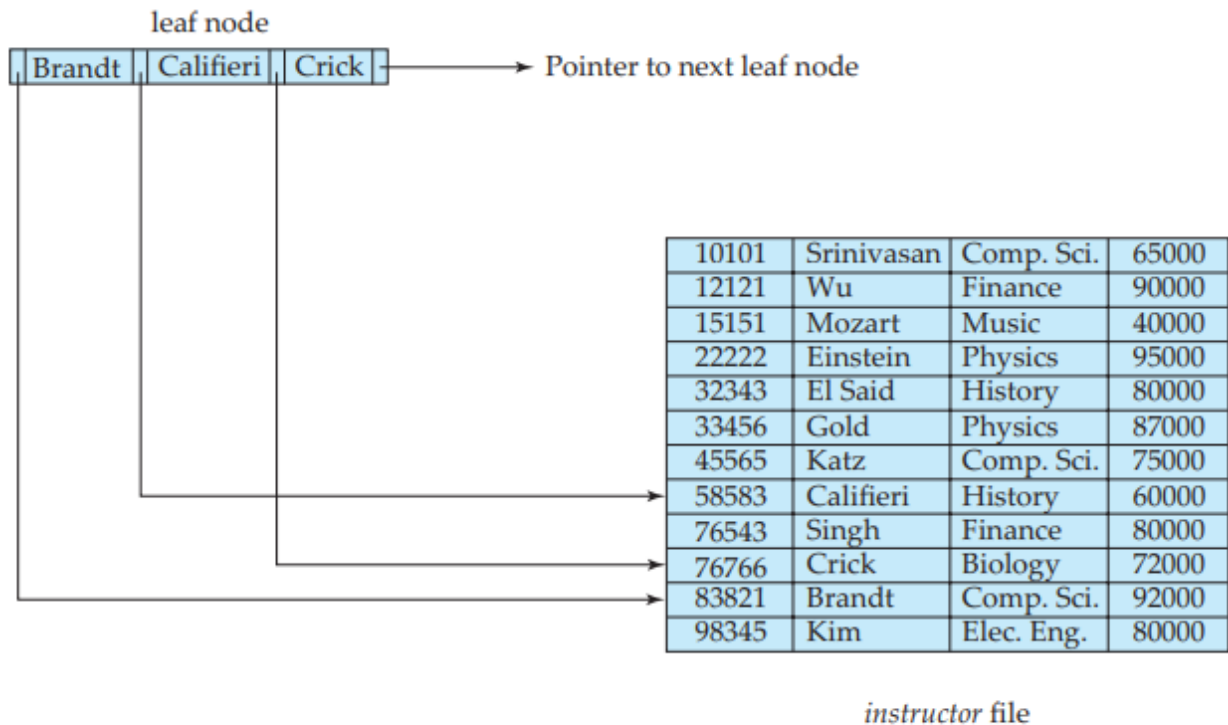


Figure 11.8 A leaf node for *instructor* B⁺-tree index ($n = 4$).

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 $\text{ceiling}((n - 1)/2)$ 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 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 $i = 2, 3, \dots, m - 1$, 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 , 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.

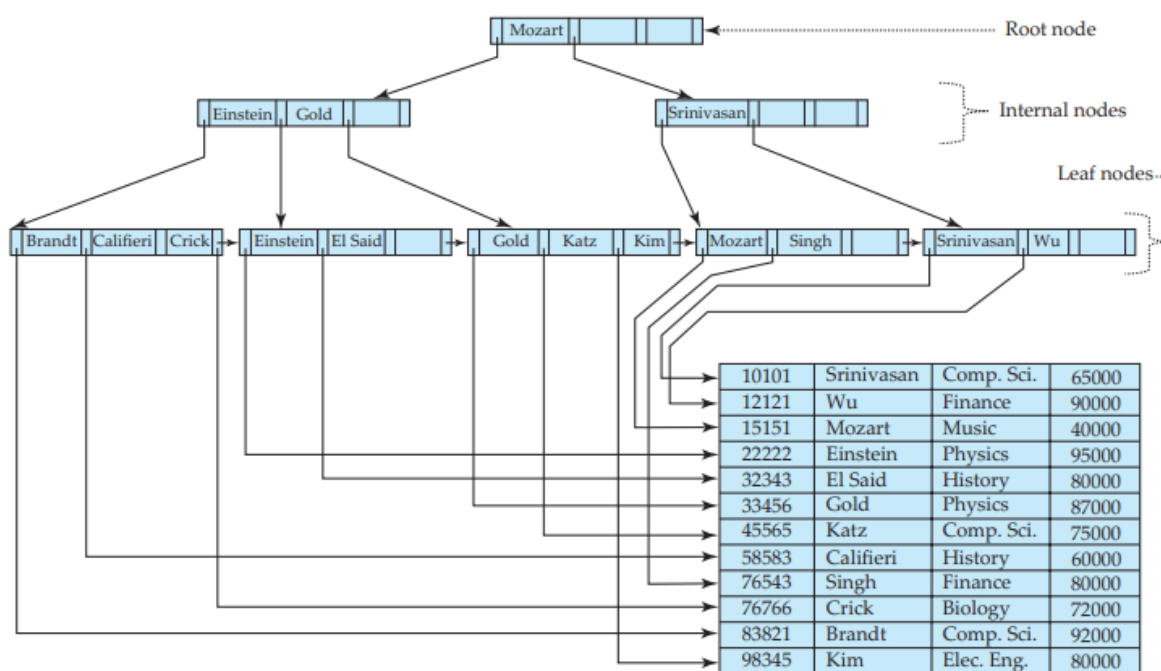


Figure 11.9 B⁺-tree for *instructor* file ($n = 4$).

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

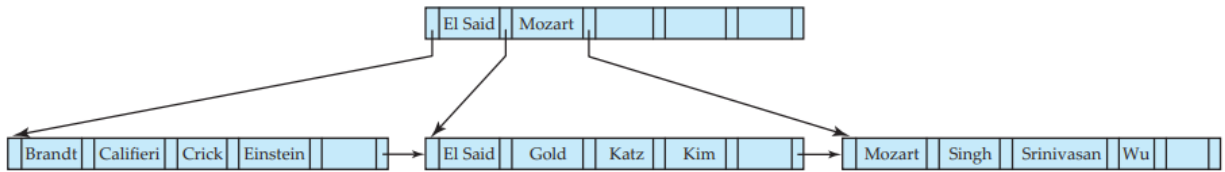


Figure 11.10 B⁺-tree for *instructor* file with $n = 6$.

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.

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 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.

```

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  $V \leq C.K_i$ 
        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 ( $i > \text{number of keys in } L$  or  $L.K_i > V$ )
        if ( $i > \text{number of keys in } L$ )
            then  $L = L.P_n$ 
            else Set done = true;
    until (done or L is null)

```

Figure 11.11 Querying a B⁺-tree.

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.

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 $\text{ceiling}(\log_{\text{ceiling}(n/2)}(N))$.

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 $\log_{50}(1,000,000) = 4$ 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 $\log_2(N)$, 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.

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.

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 $\text{ceiling}(n/2)$ in the existing node and the remaining values in a newly created node.

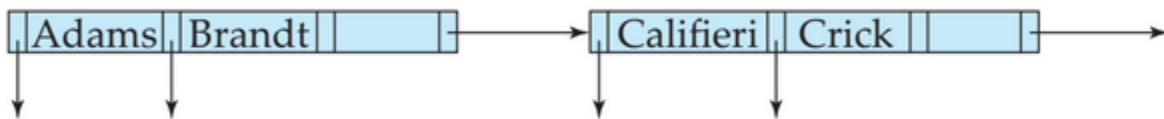


Figure 11.12 Split of leaf node on insertion of “Adams”

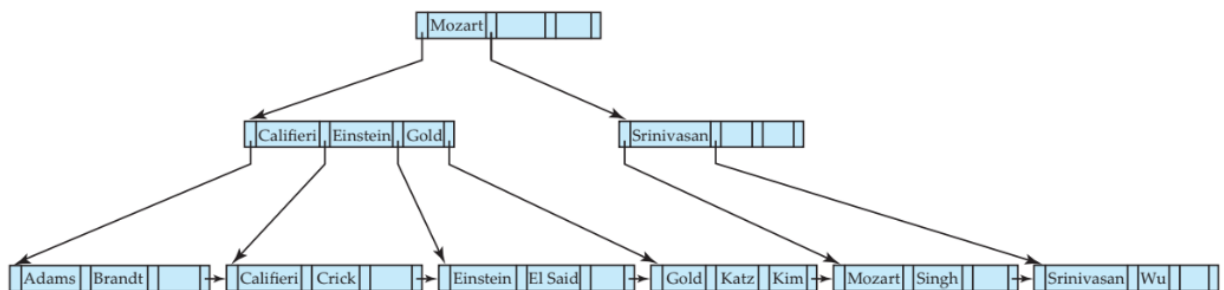


Figure 11.13 Insertion of “Adams” into the B⁺-tree of Figure 11.9.

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 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.

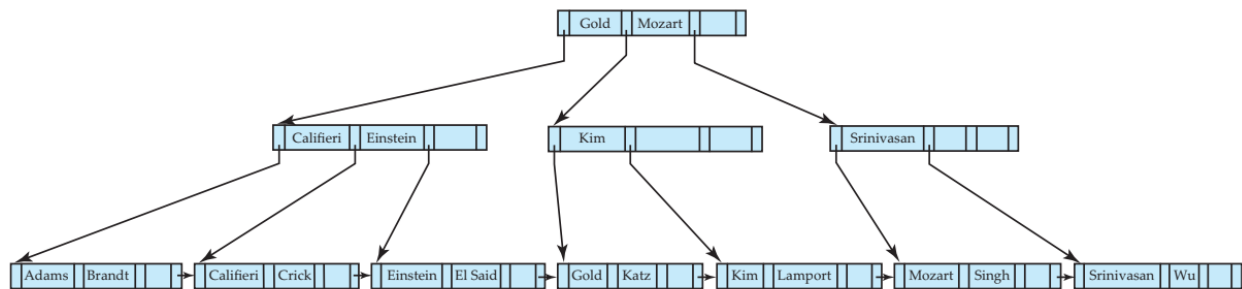


Figure 11.14 Insertion of “Lamport” into the B⁺-tree of Figure 11.13.

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.

```

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  $L.P_1 \dots L.K_{n-1}$  to a block of memory  $T$  that can
      hold  $n$  (pointer, key-value) pairs
    insert_in_leaf ( $T, K, P$ )
    Set  $L'.P_n = L.P_n$ ; Set  $L.P_n = L'$ 
    Erase  $L.P_1$  through  $L.K_{n-1}$  from  $L$ 
    Copy  $T.P_1$  through  $T.K_{\lceil n/2 \rceil}$  from  $T$  into  $L$  starting at  $L.P_1$ 
    Copy  $T.P_{\lceil n/2 \rceil + 1}$  through  $T.K_n$  from  $T$  into  $L'$  starting at  $L'.P_1$ 
    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 ( $K < L.K_1$ )
    then insert  $P, K$  into  $L$  just before  $L.P_1$ 
  else begin
    Let  $K_i$  be the highest value in  $L$  that is less than  $K$ 
    Insert  $P, K$  into  $L$  just after  $T.K_i$ 
  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  $P = \text{parent}(N)$ 
  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  $T.P_1 \dots T.P_{\lceil n/2 \rceil}$  into  $P$ 
    Let  $K'' = T.K_{\lceil n/2 \rceil}$ 
    Copy  $T.P_{\lceil n/2 \rceil + 1} \dots T.P_{n+1}$  into  $P'$ 
    insert_in_parent( $P, K'', P'$ )
  end

```

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

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 i th value and the i th 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 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.

Deletion

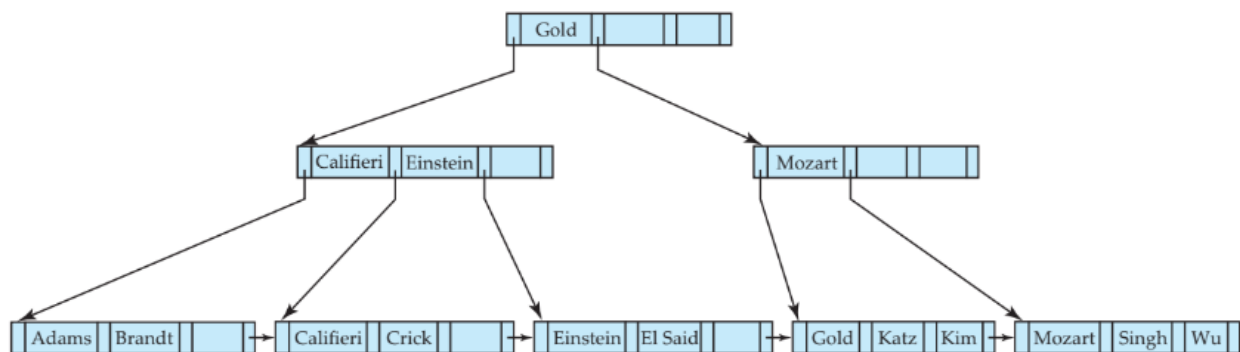


Figure 11.16 Deletion of “Srinivasan” from the B⁺-tree of Figure 11.13.

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 $1 < \text{ceiling}((n - 1)/2)$, 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, n_3), where n_3 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 $1 < \text{ceiling}(n/2)$ 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 the node and its sibling, such that each has at least $\text{ceiling}(n/2) = 2$ 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”.

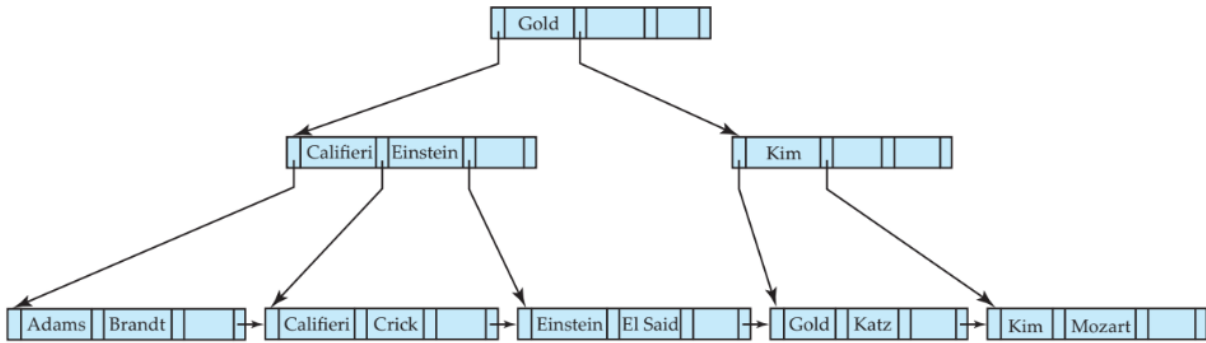


Figure 11.17 Deletion of “Singh” and “Wu” from the B⁺-tree of Figure 11.16.

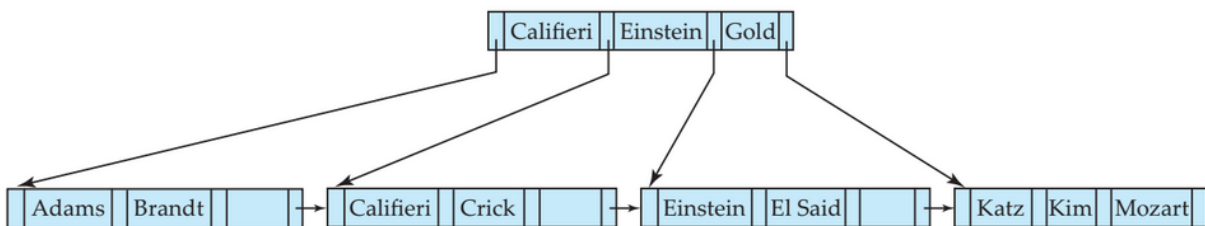


Figure 11.18 Deletion of “Gold” from the B⁺-tree of Figure 11.17.

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 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.

```

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 (Ki, Pi) pairs in N to N'; set N'.Pn = N.Pn
                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'.Pm is the last pointer in N'
                    remove (N'.Km-1, N'.Pm) from N'
                    insert (N'.Pm, K') as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.Km-1
                end
            else begin
                let m be such that (N'.Pm, N'.Km) is the last pointer/value
                    pair in N'
                remove (N'.Pm, N'.Km) from N'
                insert (N'.Pm, N'.Km) as the first pointer and value in N,
                    by shifting other pointers and values right
                replace K' in parent(N) by N'.Km
            end
        end
        else ... symmetric to the then case ...
    end
end

```

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

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 $\text{ceiling}(n/2)$ pointers; for leaf nodes, it means less than $\text{ceiling}((n - 1)/2)$ 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`.