CHAPTER 10

Indexed Sequential File Access and Prefix B+ Trees

CHAPTER OBJECTIVES

- Introduce *indexed sequential* files.
- Describe operations on a *sequence set* of blocks that maintains records in order by key.
- Show how an *index set* can be built on top of the sequence set to produce an indexed sequential file structure.
- Introduce the use of a B-tree to maintain the index set, thereby introducing *B+ trees* and *simple prefix B+ trees.*
- Illustrate how the B-tree index set in a simple prefix B+ tree can be of variable order, holding a variable number of separators.
- Compare the strengths and weaknesses of B+ trees, simple prefix B+ trees, and B-trees.

CHAPTER OUTLINE

Indexed Sequential Access

Indexed sequential file structures provide a choice between two alternative views of a file:

- *Indexed:* the file can be seen as a set of records that is *indexed* by key; or
- *Sequential:* the file can be accessed sequentially (physically contiguous records—no seeking), returning records in order by key.

The idea of having a single organizational method that provides both of these views is a new one. Up to this point we have had to choose between them. As a somewhat extreme, though instructive, example of the potential divergence of these two choices, consider the file structure of Chapter 9 that consists of a file of entry-sequenced records indexed by a separate B-tree. This structure provides excellent *indexed* access to any individual record by key, even as records are added and deleted. Now let's suppose that we also want to use this file as part of a cosequential merge. In cosequential processing we want to retrieve all the records in order by key. Since the records in this file system are *entry sequenced,* the only way to retrieve them in order by key without sorting is through the index. For a file of N records, following the N pointers from the index into the entry-

sequenced set requires N essentially random seeks into the record file. This is a *much* less efficient process than the sequential reading of physically adjacent records—so much so that it is unacceptable for any situation in which cosequential processing is a frequent occurrence.

On the other hand, our discussions of indexing show us that a file consisting of a set of records sorted by key, though ideal for cosequential processing, is an unacceptable structure when we want to access, insert, and delete records by key in random order.

What if an application involves both interactive random access and cosequential batch processing? There are many examples of such dual-mode applications. Student record systems at universities, for example, require keyed access to individual records while also requiring a large amount of batch processing, as when grades are posted or when fees are paid during registration. Similarly, credit card systems require both batch processing of charge slips and interactive checks of account status. Indexed sequential access methods were developed in response to these kinds of needs.

2 Maintaining a Sequence Set

We set aside, for the moment, the indexed part of indexed sequential access, focusing on the problem of keeping a set of records in physical order by key as records are added and deleted. We refer to this ordered set of records as a *sequence set*. We will assume that once we have a good way of maintaining a sequence set, we will find some way to index it as well. You will notice a strong parallel between these sequence set methods and the methods presented in Chapter 9 for creating and maintaining B-trees.

10.2.1 The Use of Blocks

We can immediately rule out sorting and resorting the entire sequence set as records are added and deleted, since we know that sorting an entire file is an expensive process. We need instead to find a way to *localize* the changes. One of the best ways to restrict the effects of an insertion or deletion to just a part of the sequence set involves a tool we first encountered in Chapters 3 and 4: we can collect the records into *blocks*.

When we block records, the block becomes the basic unit of input and output. We read and write entire blocks at once. Consequently, the size of the buffers we use in a program is such that they can hold an entire block.

After reading in a block, all the records in a block are in memory, where we can work on them or rearrange them much more rapidly.

An example illustrates how the use of blocks can help us keep a sequence set in order. Suppose we have records that are keyed on last name and collected together so there are four records in a block. We also include *link fields* in each block that point to the preceding block and the following block. We need these fields because, as you will see, consecutive blocks are not necessarily physically adjacent.

As with B-trees, the insertion of new records into a block can cause the block to *overflow*. The overflow condition can be handled by a block-splitting process that is analogous to, but not the same as, the block-splitting process used in a B-tree. For example, Fig. 10.1(a) shows what our blocked sequence set looks like before any insertions or deletions take place. We show only the forward links. In Fig. 10.1(b) we have inserted a new record with the key CARTER. This insertion causes block 2 to split. The second half of what was originally block 2 is found in block 4 after the split. Note that this block-splitting process operates differently from the splitting we encountered in B-trees. In a B-tree a split results in the *promotion* of a key. Here things are simpler: we just divide the records between two blocks and rearrange the links so we can still move through the file in order by key, block after block.

Deletion of records can cause a block to be less than half full and therefore to *underflow*. Once again, this problem and its solutions are analogous to what we encounter when working with B-trees. Underflow in a B-tree can lead to either of two solutions:

- If a neighboring node is also half full, we can *merge* the two nodes, freeing one up for reuse.
- If the neighboring nodes are more than half full, we can *redistribute* records between the nodes to make the distribution more nearly even.

Underflow within a block of our sequence set can be handled through the same kinds of processes. As with insertion, the process for the sequence set is simpler than the process for B-trees since the sequence set is *not* a tree and there are, therefore, no keys and records in a parent node. In Fig. 10.1(c) we show the effects of deleting the record for DAVIS. Block 4 underflows and is then merged with its successor in *logical* sequence, which is block 3. The merging process frees up block 3 for reuse. We do not show an example in which underflow leads to redistribution rather than merging, because it is easy to see how the redistribution process works. Records are simply moved between logically adjacent blocks.

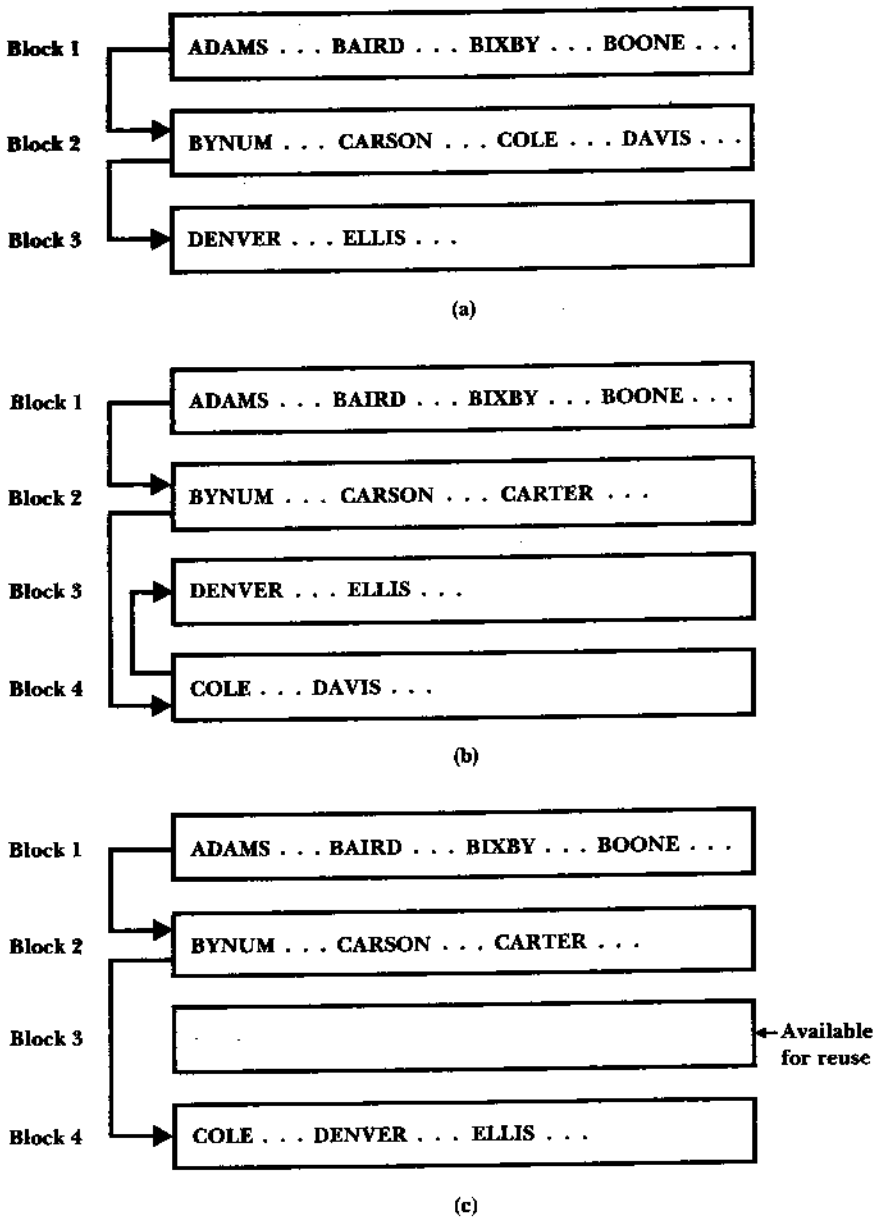


Figure 10.1 Block splitting and merging due to insertions and deletions in the sequence set. (a) Initial blocked sequence set. (b) Sequence set after insertion of CARTER record—block 2 splits, and the contents are divided between blocks 2 and 4. (c) Sequence set after deletion of DAVIS record—block 4 is less than half full, so it is concatenated with block 3.

Given the separation of records into blocks, along with these fundamental block-splitting, merging, and redistribution operations, we can keep a sequence set in order by key without ever having to sort the entire set of records. As always, nothing comes free; consequently, there are costs associated with this avoidance of sorting:

- Once insertions are made, our file takes up more space than an unblocked file of sorted records because of internal fragmentation within a block. However, we can apply the same kinds of strategies used to increase space utilization in a B-tree (for example, the use of redistribution in place of splitting during insertion, two-to-three splitting, and so on). Once again, the implementation of any of these strategies must account for the fact that the sequence set *is not a tree* and therefore there is no promotion of keys.
- The order of the records is not necessarily *physically* sequential throughout the file. The maximum guaranteed extent of physical sequentiality is within a block.

This last point leads us to the important question of selecting a block size.

10.2.2 Choice of Block Size

As we work with our sequence set, a block is the basic unit for our I/O operations. When we read data from the disk, we never read less than a block; when we write data, we always write at least one block. A block is also, as we have said, the maximum *guaranteed* extent of physical sequentiality. It follows that we should think in terms of *large* blocks, with each block holding many records. So the question of block size becomes one of identifying the *limits* on block size: why not make the block size so big we can fit the entire file in a single block?

One answer to this is the same as the reason we cannot always use a memory sort on a file: we usually do not have enough memory available. So our first consideration regarding an upper bound for block size is as follows:

Consideration 1: The block size should be such that we can hold several blocks in memory at once. For example, in performing a block split or merging, we want to be able to hold at least two blocks in memory at a time. If we are implementing two-to-three splitting to conserve disk space, we need to hold at least three blocks in memory at a time.

Although we are presently focusing on the ability to access our sequence set *sequentially*, we eventually want to consider the problem of randomly accessing a single record from our sequence set. We have to read in an entire block to get at any one record within that block. We can therefore state a second consideration:

Consideration 2: Reading in or writing out a block should not take very long. Even if we had an unlimited amount of memory, we would want to place an upper limit on the block size so we would not end up reading in the entire file just to get at a single record.

This second consideration is more than a little imprecise. How long is very long? We can refine this consideration by factoring in some of our knowledge of the performance characteristics of disk drives:

Consideration 2 (redefined): The block size should be such that we can access a block without having to bear the cost of a disk seek within the block read or block write operation.

This is not a *mandatory* limitation, but it is a sensible one: we are interested in a block because it contains records that are physically adjacent, so let's not extend blocks beyond the point at which we can guarantee such adjacency. And where is that?

When we discussed sector formatted disks back in Chapter 3, we introduced the term *cluster*. A cluster is the minimum number of sectors allocated at a time. If a cluster consists of eight sectors, then a file containing only 1 byte still uses up eight sectors on the disk. The reason for clustering is that it guarantees a minimum amount of physical sequentiality. As we move from cluster to cluster in reading a file, we may incur a disk seek, but within a cluster the data can be accessed without seeking.

One reasonable suggestion for deciding on a block size, then, is to make each block equal to the size of a cluster. Often the cluster size on a disk system has already been determined by the system administrator. But what if you are configuring a disk system for a particular application and can therefore choose your own cluster size? You need to consider the issues relating to cluster size raised in Chapter 3, along with the constraints imposed by the amount of memory available and the number of blocks you want to hold in memory at once. As is so often the case, the final decision will probably be a compromise between a number of divergent considerations. The important thing is that the compromise be a truly *informed* decision, based on knowledge of how I/O devices and file structures work rather than just a guess.

If you are working with a disk system that is not sector oriented but allows you to choose the block size for a particular file, a good starting point is to think of a block as an entire track of the disk. You may want to revise this downward, to half a track, for instance, depending on memory constraints, record size, and other factors.

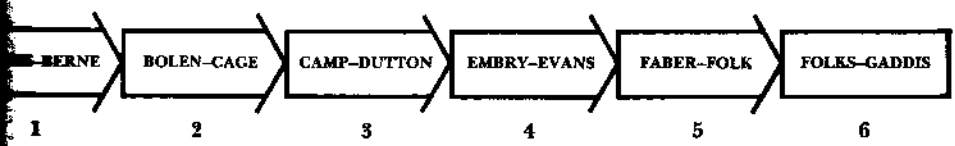
3 Adding a Simple Index to the Sequence Set

We have created a mechanism for maintaining a set of records so we can access them sequentially in order by key. It is based on the idea of grouping the records into blocks then maintaining the blocks, as records are added and deleted, through splitting, merging, and redistribution. Now let's see whether we can find an efficient way to locate some specific block containing a particular record, given the record's key.

We can view each of our blocks as containing a *range* of records, as illustrated in Fig. 10.2. This is an outside view of the blocks (we have not actually read any blocks and so do not know *exactly* what they contain), but it is sufficiently informative to allow us to choose which block *might* have the record we are seeking. We can see, for example, that if we are looking for a record with the key BURNS, we want to retrieve and inspect the second block.

It is easy to see how we could construct a simple, single-level index for these blocks. We might choose, for example, to build an index of fixed-length records that contain the key for the last record in each block, as shown in Fig. 10.3. Note that we are using the largest key in the block as the key of the whole block. In Chapter 9, we used the smallest key in a B-tree node as the key of the whole block, again because it is a little simpler. Yet another programming exercise is included in Chapter 9 to make the revisions required to use largest keys.

The combination of this kind of index with the sequence set of blocks provides complete indexed sequential access. If we need to retrieve a



e 10.2 Sequence of blocks showing the range of keys in each block.

Key	Block number
BERNE	1
CAGE	2
DUTTON	3
EVANS	4
FOLK	5
GADDIS	6

Figure 10.3 Simple index for the sequence set illustrated in Fig. 10.2.

specific record, we consult the index and then retrieve the correct block; if we need sequential access we start at the first block and read through the linked list of blocks until we have read them all. As simple as this approach is, it is a very workable one as long as the entire index can be held in memory. The requirement that the index be held in memory is important for two reasons:

- Since this is a simple index of the kind we discussed in Chapter 7, we find specific records by means of a binary search of the index. Binary searching works well if the searching takes place in memory, but, as we saw in the previous chapter on B-trees, it requires too many seeks if the file is on a secondary storage device.
- As the blocks in the sequence set are changed through splitting, merging, and redistribution, the index has to be updated. Updating a simple, fixed-length record index of this kind works well if the index is relatively small and contained in memory. If, however, the updating requires seeking to individual index records on disk, the process can become very expensive. Once again, this is a point we discussed more completely in earlier chapters.

What do we do, then, if the file contains so many blocks that the block index does not conveniently fit into memory? In the preceding chapter we found that we could divide the index structure into *pages,* much like the *blocks* we are discussing here, handling several pages, or blocks, of the index in memory at a time. More specifically, we found that B-trees are an excellent file structure for handling indexes that are too large to fit entirely in memory. This suggests that we might organize the index to our sequence set as a B-tree.

The use of a B-tree index for our sequence set of blocks is a very powerful notion. The resulting hybrid structure is known as a *B^+ tree,* which is appropriate since it is a B-tree index *plus* a sequence set that holds

the records. Before we can fully develop the notion of a B+ tree, we need to think more carefully about what it is we need to keep in the index.

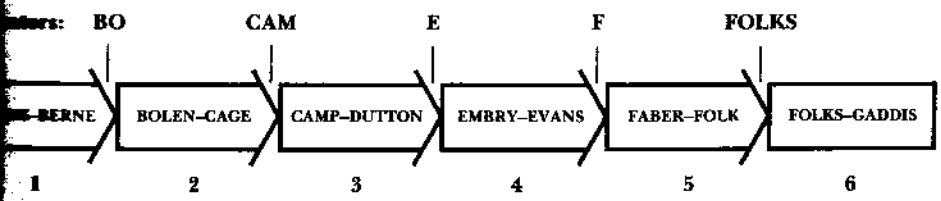
The Content of the Index: Separators Instead of Keys

The purpose of the index we are building is to assist us when we are searching for a record with a specific key. The index must guide us to the block in the sequence set that contains the record, if it exists in the sequence set at all. The index serves as a kind of road map for the sequence set. We are interested in the content of the index only insofar as it can assist us in getting to the correct block in the sequence set; the index set does not itself contain answers, only information about where to go to get answers.

Given this view of the index set as a road map, we can take the very important step of recognizing that *we do not need to have keys in the index set.* Our real need is for *separators.* Figure 10.4 shows one possible set of separators for the sequence set in Fig. 10.2.

Note that there are many potential separators capable of distinguishing between two blocks. For example, all of the strings shown between blocks 3 and 4 in Fig. 10.5 are capable of guiding us in our choice between the blocks as we search for a particular key. If a string comparison between the key and any of these separators shows that the key precedes the separator, we look for the key in block 3. If the key follows the separator, we look in block 4.

If we are willing to treat the separators as variable-length entities within our index structure (we talk about how to do this later), we can save space by placing the *shortest separator* in the index structure. Consequently, we use *E* as the separator to guide our choice between blocks 3 and 4. Note that there is not always a unique shortest separator. For exam-



10.4 Separators between blocks in the sequence set.

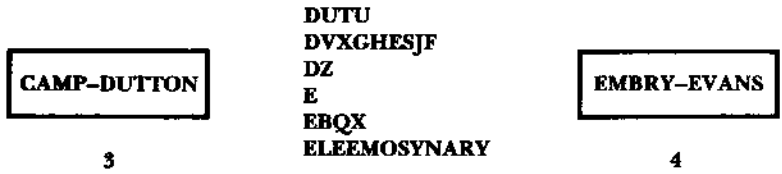


Figure 10.5 A list of potential separators.

ple, BK, BN, and BO are separators that are all the same length and are equally effective as separators between blocks 1 and 2 in Fig. 10.4. We choose BO and all of the other separators contained in Fig. 10.4 by using the logic embodied in the C++ function shown in Fig. 10.6.

Note that these functions can produce a separator that is the same as the second key. This situation is illustrated in Fig. 10.4 by the separator between blocks 5 and 6, which is the same as the first key contained in block 6. It follows that, as we use the separators as a road map to the sequence set, we must decide to retrieve the block to the right of the separator or the one to the left of the separator according to the following rule:

Relation of search key and separator	*Decision*
Key < separator	Go left
Key = separator	Go right
Key > separator	Go right

```
id FindSeparator (char * key1, char * key2, char * sep)
/ key1, key2, and sep point to the beginning of char arrays
while (1) // loop until break
{
   *sep = *key2; sep ++; //move the current character into sep
   if (*key2 != *key1) break; // stop when a difference is found
   if (*key2 == 0) break; // stop at end of key2
   key1 ++; key2 ++; // move to the next character of keys
}
*sep = 0; // null terminate the separator string
```

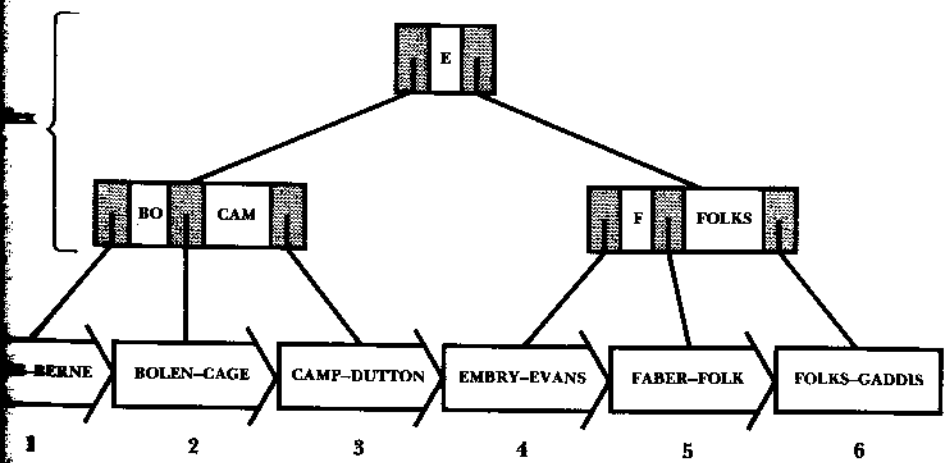
re 10.6 C++ function to find a shortest separator.

5 The Simple Prefix B+ Tree

Figure 10.7 shows how we can form the separators identified in Fig. 10.4 into a B-tree index of the sequence set blocks. The B-tree index is called the *index set*. Taken together with the sequence set, it forms a file structure called a *simple prefix B+ tree*. The modifier *simple prefix* indicates that the index set contains shortest separators, or *prefixes* of the keys rather than copies of the actual keys. Our separators are simple because they are, simply, prefixes. They are just the initial letters within the keys. More complicated (not simple) methods of creating separators from key prefixes remove unnecessary characters from the front of the separator as well as from the rear. (See Bayer and Unterauer, 1977, for a more complete discussion of prefix B+ trees.)[1]

As was noted previously, the implementation of B-trees in Chapter 9 has the same number of keys and references in all nodes, even though for interior nodes, the last key is not needed. We drop the extra key in the following examples and discussion. If we had as many separators as we

1. The literature on B+ trees and simple prefix B+ trees is remarkably inconsistent in the nomenclature used for these structures. B+ trees are sometimes called B* trees; simple prefix B+ trees are sometimes called simple prefix B-trees. Comer's important article in *Computing Surveys* in 1979 has reduced some of the confusion by providing a consistent, standard nomenclature which we use here.



10.7 A B-tree index set for the sequence set, forming a simple prefix B+ tree.

have children (references), the last separator would be larger than the largest key in the subtree. In essence, it separates keys in the subtree from those that are larger than the largest key in the subtree. This last separator is truly not needed in a prefix tree.

Note that the index set is a B-tree, and a node containing N separators branches to $N + 1$ children. If we are searching for the record with the key EMBRY, we start at the root of the index set, comparing EMBRY with the separator E. Since EMBRY comes after E, we branch to the right, retrieving the node containing the separators F and FOLKS. Since EMBRY comes before even the first of these separators, we follow the branch that is to the left of the F separator, which leads us to block 4, the correct block in the sequence set.

10.6 Simple Prefix B+ Tree Maintenance

10.6.1 Changes Localized to Single Blocks in the Sequence Set

Let's suppose that we want to delete the records for EMBRY and FOLKS and that neither of these deletions results in any merging or redistribution within the sequence set. Since there is no merging or redistribution, the effect of these deletions on the *sequence set* is limited to changes *within* blocks 4 and 6. The record that was formerly the second record in block 4 (let's say that its key is ERVIN) is now the first record. Similarly, the former second record in block 6 (we assume it has a key of FROST) now starts that block. These changes can be seen in Fig. 10.8.

The more interesting question is what effect, if any, these deletions have on the *index set*. The answer is that since the number of sequence set blocks is unchanged and since no records are moved between blocks, the index set can also remain unchanged. This is easy to see in the case of the EMBRY deletion: E is still a perfectly good separator for sequence set blocks 3 and 4, so there is no reason to change it in the index set. The case of the FOLKS deletion is a little more confusing because the string FOLKS appears both as a key in the deleted record and as a separator within the index set. To avoid confusion, remember to distinguish clearly between these two uses of the string FOLKS: FOLKS can continue to serve as a separator between blocks 5 and 6 even though the FOLKS record is deleted. (One could argue that although we do not *need* to replace the FOLKS

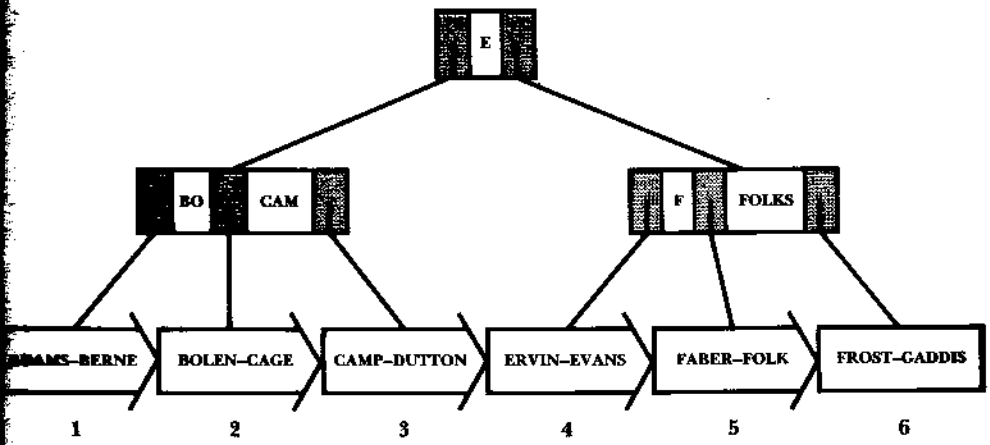


Figure 10.8 The deletion of the EMBRY and FOLKS records from the sequence set leaves the index set unchanged.

separator, we should do so anyway because it is now possible to construct a *shorter* separator. However, the cost of making such a change in the index set usually outweighs the benefits associated with saving a few bytes of space.)

The effect of inserting into the sequence set new records that do not cause block splitting is much the same as the effect of these deletions that do not result in merging: the index set remains unchanged. Suppose, for example, that we insert a record for EATON. Following the path indicated by the separators in the index set, we find that we will insert the new record into block 4 of the sequence set. We assume, for the moment, that there is room for the record in the block. The new record becomes the first record in block 4, but no change in the index set is necessary. This is not surprising: we decided to insert the record into block 4 on the basis of the existing information in the index set. It follows that the existing information in the index set is sufficient to allow us to find the record again.

10.6.2 Changes Involving Multiple Blocks in the Sequence Set

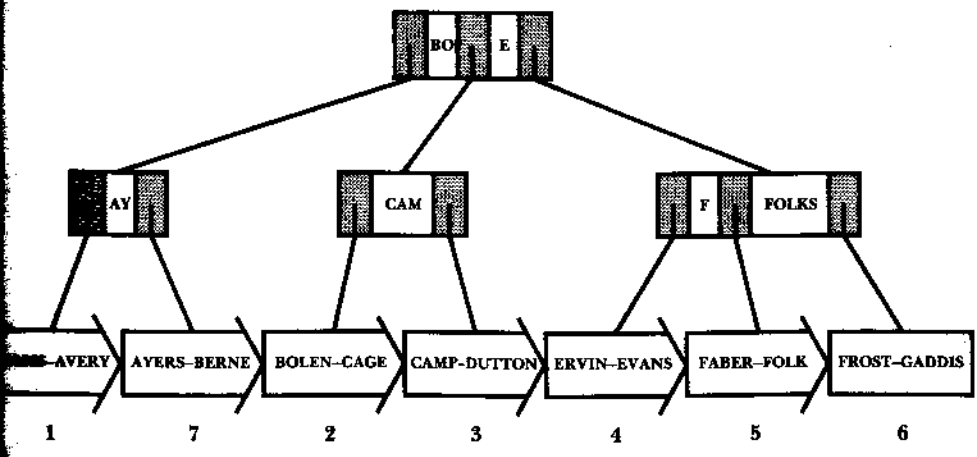
What happens when the addition and deletion of records to and from the sequence set *does* change the number of blocks in the sequence set? Clearly, if we have more blocks, we need additional separators in the index set, and if we have fewer blocks, we need fewer separators. Changing the

number of separators certainly has an effect on the index set, where the separators are stored.

Since the index set for a simple prefix B+ tree is just a normal B-tree, the changes to the index set are handled according to the familiar rules for B-tree insertion and deletion.[2] In the following examples, we assume that the index set is a B-tree of order three, which means that the maximum number of separators we can store in a node is two. We use this small node size for the index set to illustrate node splitting and merging while using only a few separators. As you will see later, implementations of simple prefix B+ trees place a much larger number of separators in a node of the index set.

We begin with an insertion into the sequence set shown in Fig. 10.8. Specifically, let's assume that there is an insertion into the first block and that this insertion causes the block to split. A new block (block 7) is brought in to hold the second half of what was originally the first block. This new block is linked into the correct position in the sequence set, following block 1 and preceding block 2 (these are the *physical* block numbers). These changes to the sequence set are illustrated in Fig. 10.9.

2. As you study the material here, you may find it helpful to refer back to Chapter 9, where we discuss B-tree operations in much more detail.



ure 10.9 An insertion into block 1 causes a split and the consequent addition of **ck 7.** The addition of a block in the sequence set requires a new separator in the index **. Insertion of the AY separator into the node containing BO and CAM causes a node t in the index set B-tree and consequent promotion of BO to the root.

Note that the separator that formerly distinguished between blocks 1 and 2, the string BO, is now the separator for blocks 7 and 2. We need a new separator, with a value of AY, to distinguish between blocks 1 and 7. As we start to place this separator into the index set, we find that the node into which we want to insert it, containing BO and CAM, is already full. Consequently, insertion of the new separator causes a split and promotion, according to the usual rules for B-trees. The promoted separator, BO, is placed in the root of the index set.

Now let's suppose we delete a record from block 2 of the sequence set and that this causes an underflow condition and consequent merging of blocks 2 and 3. Once the merging is complete, block 3 is no longer needed in the sequence set, and the separator that once distinguished between blocks 2 and 3 must be removed from the index set. Removing this separator, CAM, causes an underflow in an index set node. Consequently, there is another merging, this time in the index set, that results in the demotion of the BO separator from the root, bringing it back down into a node with the AY separator. Once these changes are complete, the simple prefix B+ tree has the structure illustrated in Fig. 10.10.

Although in these examples a block split in the sequence set results in a node split in the index set and a merging in the sequence set results in a

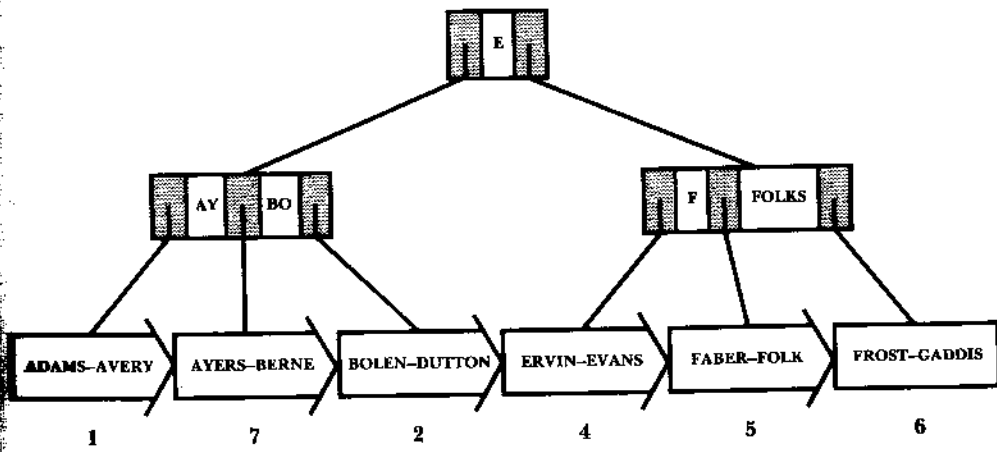


Figure 10.10 A deletion from block 2 causes underflow and the consequent merging of blocks 2 and 3. After the merging, block 3 is no longer needed and can be placed on an avail list. Consequently, the separator CAM is no longer needed. Removing CAM from its node in the index set forces a merging of index set nodes, bringing BO back down from the root.

merging in the index set, there is not always this correspondence of action. Insertions and deletions in the index set are handled as standard B-tree operations; whether there is splitting or a simple insertion, merging or a simple deletion, depends entirely on how full the index set node is.

Writing procedures to handle these kinds of operations is a straightforward task if you remember that the changes take place *from the bottom up.* Record insertion and deletion *always* take place in the sequence set, since that is where the records are. If splitting, merging, or redistribution is necessary, perform the operation just as you would *if there were no index set at all.* Then, after the record operations in the sequence set are complete, make changes as necessary in the index set:

- If blocks are split in the sequence set, a new separator must be inserted into the index set;
- If blocks are merged in the sequence set, a separator must be removed from the index set; and
- If records are redistributed between blocks in the sequence set, the value of a separator in the index set must be changed.

Index set operations are performed according to the rules for B-trees. This means that node splitting and merging *propagate* up through the higher levels of the index set. We see this in our examples as the BO separator moves in and out of the root. Note that the operations on the sequence set do not involve this kind of propagation. That is because the sequence set is a linear, linked list, whereas the index set is a tree. It is easy to lose sight of this distinction and to think of an insertion or deletion in terms of a *single* operation on the *entire* simple prefix B^+ tree. This is a good way to become confused. Remember: insertions and deletions happen in the *sequence set* because that is where the records are. Changes to the index set are secondary; they are a byproduct of the fundamental operations on the sequence set.

9.7 Index Set Block Size

Up to this point we have ignored the important issues of size and structure of the index set nodes. Our examples have used extremely small index set nodes and have treated them as fixed-order B-tree nodes, even though the separators are variable in length. We need to develop more realistic, useful ideas about the size and structure of index set nodes.

The physical size of a node for the index set is usually the same as the physical size of a block in the sequence set. When this is the case, we speak of index set *blocks*, rather than *nodes*, just as we speak of sequence set blocks. There are a number of reasons for using a common block size for the index and sequence sets:

- The block size for the sequence set is usually chosen because there is a good fit among this block size, the characteristics of the disk drive, and the amount of memory available. The choice of an index set block size is governed by consideration of the same factors; therefore, the block size that is best for the sequence set is usually best for the index set.
- A common block size makes it easier to implement a buffering scheme to create a *virtual* simple prefix B+ tree, similar to the virtual B-trees discussed in the preceding chapter.
- The index set blocks and sequence set blocks are often mingled within the same file to avoid seeking between two separate files while accessing the simple prefix B+ tree. Use of one file for both kinds of blocks is simpler if the block sizes are the same.

9.8 Internal Structure of Index Set Blocks: A Variable-Order B-Tree

Given a large, fixed-size block for the index set, how do we store the separators within it? In the examples considered so far, the block structure is such that it can contain only a fixed number of separators. The entire motivation behind the use of *shortest* separators is the possibility of packing more of them into a node. This motivation disappears completely if the index set uses a fixed-order B-tree in which there is a fixed number of separators per node.

We want each index set block to hold a variable number of variable-length separators. How should we go about searching through these separators? Since the blocks are probably large, any single block can hold a large number of separators. Once we read a block into memory for use, we want to be able to do a binary rather than sequential search on its list of separators. We therefore need to structure the block so it can support a binary search, despite the fact that the separators are of variable length.

In Chapter 7, which covers indexing, we see that the use of a separate index can provide a means of performing binary searches lists of variable-

Figure 10.11 Variable-length separators and corresponding index.

length entities. If the index consists of fixed-length references, we can use binary searching on the index, retrieving the variable-length records or fields through indirection. For example, suppose we are going to place the following set of separators into an index block:

```
As, Ba, Bro, C, Ch, Cra, Dele, Edi, Err, Fa, Fle
```

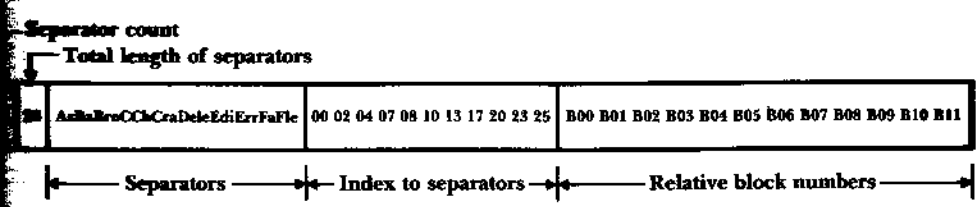
(We are using lowercase letters rather than all uppercase letters so you can find the separators more easily when we merge them.) We could merge these separators and build an index for them, as shown in Fig. 10.11.

If we are using this block of the index set as a road map to help us find the record in the sequence set for "Beck," we perform a binary search on the index to the separators, retrieving first the middle separator, "Cra," which starts in position 10. Note that we can find the length of this separator by looking at the starting position of the separator that follows. Our binary search eventually tells us that "Beck" falls between the separators "Ba" and "Bro." Then what do we do?

The purpose of the index set road map is to guide us downward through the levels of the simple prefix B+ tree, leading us to the sequence set block we want to retrieve. Consequently, the index set block needs some way to store references to its children, to the blocks descending from it in the next lower level of the tree. We assume that the references are made in terms of a relative block number (RBN), which is analogous to a relative record number except that it references a fixed-length block rather than a record. If there are N separators within a block, the block has $N + 1$ children and therefore needs space to store $N + 1$ RBNs in addition to the separators and the index to the separators.

There are many ways to combine the list of separators, the index to separators, and the list of RBNs into a single index set block. One possible approach is illustrated in Fig. 10.12. In addition to the vector of separators, the index to these separators, and the list of associated block numbers, this block structure includes:

- *Separator count:* we need this to help us find the middle element in the index to the separators so we can begin our binary search.



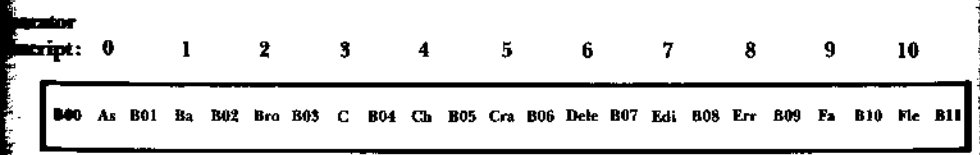
ure 10.12 Structure of an index set block.

- *Total length of separators:* the list of merged separators varies in length from block to block. Since the index to the separators begins at the end of this variable-length list, we need to know how long the list is so we can find the beginning of our index.

Let's suppose, once again, that we are looking for a record with the key "Beck" and that the search has brought us to the index set block pictured in Fig. 10.12. The total length of the separators and the separator count allow us to find the beginning, the end, and consequently the middle of the index to the separators. As in the preceding example, we perform a binary search of the separators through this index, finally concluding that the key "Beck" falls between the separators "Ba" and "Bro." *Conceptually,* the relation between the keys and the RBNs is illustrated in Fig. 10.13.

As Fig. 10.13 makes clear, discovering that the key falls between "Ba" and "Bro" allows us to decide that the *next* block we need to retrieve has the RBN stored in the B02 position of the RBN vector. This next block could be another index set block and thus another block of the road map, or it could be the sequence set block that we are looking for. In either case, the quantity and arrangement of information in the current index set block is sufficient to let us conduct our binary search *within* the index block and proceed to the next block in the simple prefix B+ tree.

There are many alternate ways to arrange the fundamental components of this index block. (For example, would it be easier to build the block if the vector of keys were placed at the end of the block? How would you handle the fact that the block consists of both *character* and *integer*



ure 10.13 Conceptual relationship of separators and relative block numbers.

entities with no constant, fixed dividing point between them?) For our purposes here, the specific implementation details for this particular index block structure are not nearly as important as the block's *conceptual structure*. This kind of index block structure illustrates two important points.

The first point is that a block is not just an arbitrary chunk cut out of a homogeneous file; it can be more than just a set of records. A block can have a sophisticated internal structure all its own, including its own internal index, a collection of variable-length records, separate sets of fixed-length records, and so forth. This idea of building more sophisticated data structures inside of each block becomes increasingly attractive as the block size increases. With very large blocks it becomes imperative that we have an efficient way of processing all of the data within a block once it has been read into memory. This point applies not only to simple prefix B+ trees but to any file structure using a large block size.

The second point is that a node within the B-tree index set of our simple prefix B+ tree is of variable order, since each index set block contains a variable number of separators. This variability has interesting implications:

- The number of separators in a block is directly limited by block size rather than by some predetermined *order* (as in an *order m* B-tree). The index set will have the maximum *order*, and therefore the minimum depth, that is possible given the degree of compression used to form the separators.
- Since the tree is of *variable order*, operations such as determining when a block is full, or half full, are no longer a simple matter of comparing a separator count against some fixed maximum or minimum. Decisions about when to split, merge, or redistribute become more complicated.

The exercises at the end of this chapter provide opportunities for exploring variable-order trees more thoroughly.

9.9 Loading a Simple Prefix B+ Tree

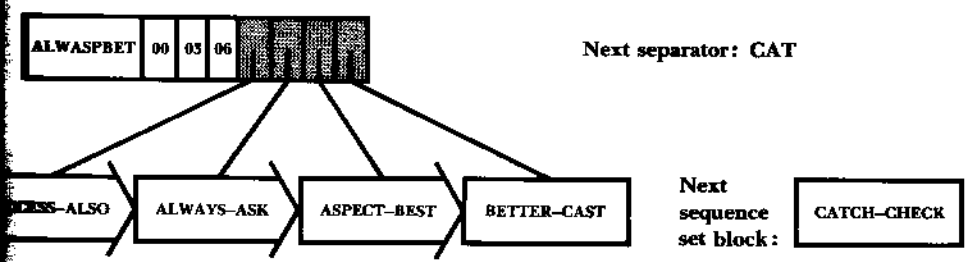
In the previous description of the simple prefix B+ tree, we focus first on building a sequence set and subsequently present the index set as something that is added or built on top of the sequence set. It is not only possible to *conceive* of simple prefix B+ trees this way, as a sequence set with an added index, but one can also *build* them this way.

One way of building a simple prefix B+ tree, of course, is through a series of successive insertions. We would use the procedures outlined in section 10.6, where we discuss the maintenance of simple prefix B+ trees, to split or redistribute blocks in the sequence set and in the index set as we added blocks to the sequence set. The difficulty with this approach is that splitting and redistribution are relatively expensive. They involve searching down through the tree for each insertion then reorganizing the tree as necessary on the way back up. These operations are fine for tree *maintenance* as the tree is updated, but when we are loading the tree we do not have to contend with a *random-order* insertion and therefore do not need procedures that are so powerful, flexible, and expensive. Instead, we can begin by sorting the records that are to be loaded. Then we can guarantee that the next record we encounter is the next record we need to load.

Working from a sorted file, we can place the records into sequence set blocks, one by one, starting a new block when the one we are working with fills up. As we make the transition between two sequence set blocks, we can determine the shortest separator for the blocks. We can collect these separators into an index set block that we build and hold in memory until it is full.

To develop an example of how this works, let's assume that we have sets of records associated with terms that are being compiled for a book index. The records might consist of a list of the occurrences of each term. In Fig. 10.14 we show four sequence set blocks that have been written out to the disk and one index set block that has been built in memory from the shortest separators derived from the sequence set block keys. As you can see, the next sequence set block consists of a set of terms ranging from CATCH through CHECK, and therefore the next separator is CAT. Let's suppose that the index set block is now full. We write it out to disk. Now what do we do with the separator CAT?

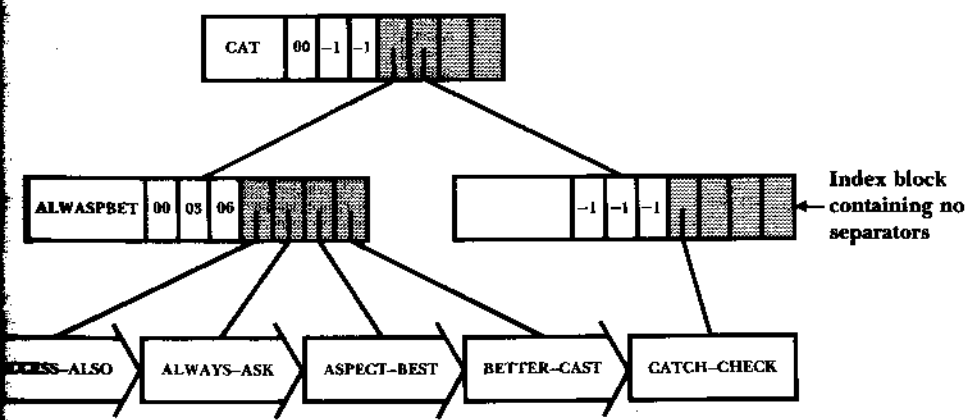
Clearly, we need to start a new index block. However, we cannot place CAT into another index block at the same level as the one containing the



re 10.14 Formation of the first index set block as the sequence set is loaded.

separators ALW, ASP, and BET because we cannot have two blocks at the same level without having a parent block. Instead, we promote the CAT separator to a higher-level block. However, the higher-level block cannot point directly to the sequence set; it must point to the lower-level index blocks. This means that we will now be building *two* levels of the index set in memory as we build the sequence set. Figure 10.15 illustrates this working-on-two-levels phenomenon: the addition of the CAT separator requires us to start a new, root-level index block as well as a lower-level index block. (Actually, we are working on *three* levels at once since we are also constructing the sequence set blocks in memory.) Figure 10.16 shows what the index looks like after even more sequence set blocks are added. As you can see, the lower-level index block that contained no separators when we added CAT to the root has now filled up. To establish that the tree works, do a search for the term CATCH. Then search for the two terms CASUAL and CATALOG. How can you tell that these terms are not in the sequence set?

It is instructive to ask what would happen if the last record were CHECK, so the construction of the sequence sets and index sets would stop with the configuration shown in Fig. 10.15. The resulting simple prefix B+ tree would contain an index set node that holds no separators. This is not an isolated possibility. If we use this sequential loading method to build the tree, there will be many points during the loading process at which there is an empty or nearly empty index set node. If the index set grows to more than two levels, this empty node problem can occur at even higher levels of the tree, creating a potentially severe out-of-



ure 10.15 Simultaneous building of two index set levels as the sequence set **tinues** to grow.

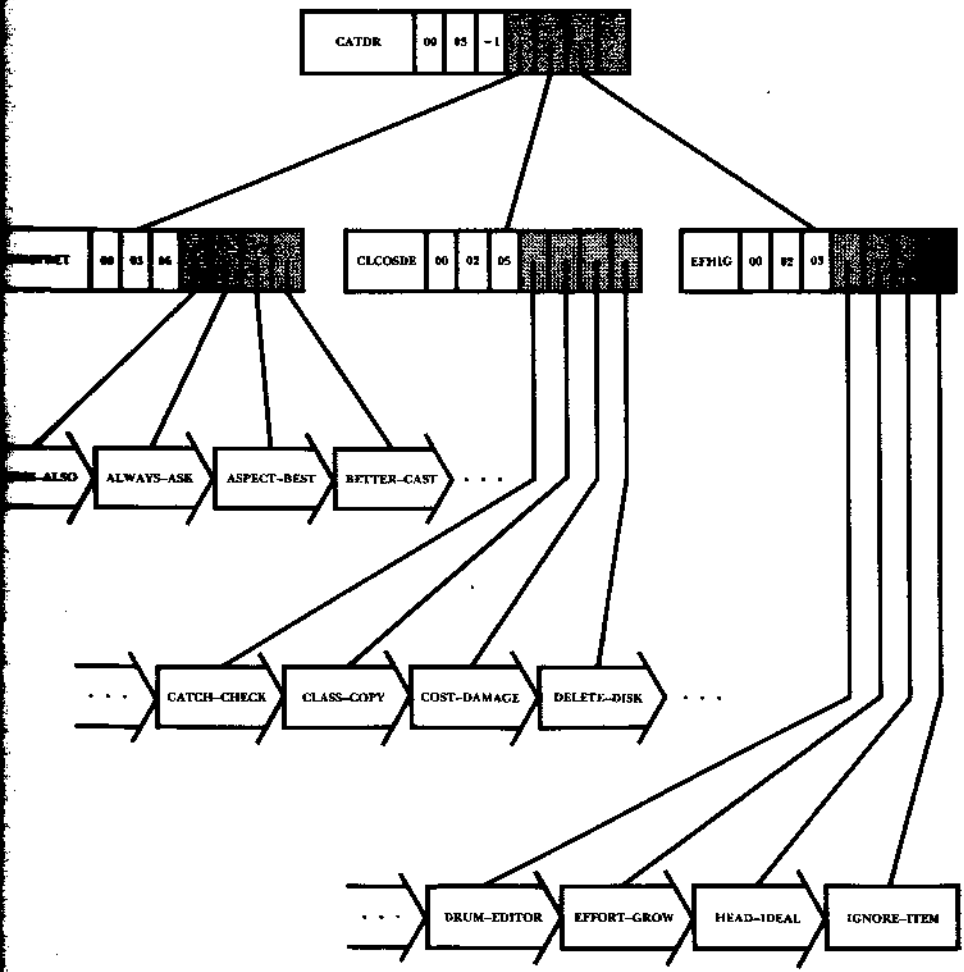


Figure 10.16 Continued growth of index set built up from the sequence set.

balance problem. Clearly, these empty node and nearly empty node conditions violate the B-tree rules that apply to the index set. However, once a tree is loaded and goes into regular use, the fact that a node is violating B-tree conditions can be used to guarantee that the node will be corrected through the action of normal B-tree maintenance operations. It is easy to write the procedures for insertion and deletion so a redistribution procedure is invoked when an underfull node is encountered.

The advantages of loading a simple prefix B+ tree in this way, as a sequential operation following a sort of the records, almost always

outweigh the disadvantages associated with the possibility of creating blocks that contain too few records or too few separators. The principal advantage is that the loading process goes more quickly because

- The output can be written sequentially;
- We make only one pass over the data, rather than the many passes associated with random order insertions; and
- No blocks need to be reorganized as we proceed.

There are two additional advantages to using a separate loading process such as the one we have described. These advantages are related to performance after the tree is loaded rather than performance during loading:

- Random insertion produces blocks that are, on the average, between 67 percent and 80 percent full. In the preceding chapter, when we were discussing B-trees, we increased this number by using such mechanisms as redistribution during insertion rather than just block splitting. But, still, we never had the option of filling the blocks completely so we had 100 percent utilization. The sequential loading process changes this. If we want, we can load the tree so it starts out with 100 percent utilization. This is an attractive option if we do not expect to add very many records to the tree. On the other hand, if we anticipate many insertions, sequential loading allows us to select any other degree of utilization that we want. Sequential loading gives us much more control over the amount and placement of empty space in the newly loaded tree.
- In the loading example presented in Fig. 10.15, we write out the first four sequence set blocks and then write out the index set block containing the separators for these sequence set blocks. If we use the same file for both sequence set and index set blocks, this process guarantees that an index set block starts out in *physical proximity* to the sequence set blocks that are its descendants. In other words, our sequential loading process is creating a degree of *spatial locality* within our file. This locality can minimize seeking as we search down through the tree.

10.10 B+ Trees

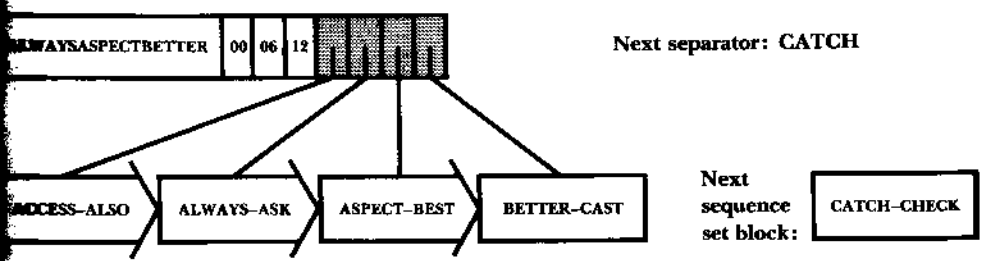
Our discussions up to this point have focused primarily on simple prefix B+ trees. These structures are actually a variant of an approach to file organization known simply as a *B+ tree*. The difference between a simple prefix

B+ tree and a plain B+ tree is that the latter structure does not involve the use of prefixes as separators. Instead, the separators in the index set are simply copies of the actual keys. Contrast the index set block shown in Fig. 10.17, which illustrates the initial loading steps for a B+ tree, with the index block that is illustrated in Fig. 10.14, where we are building a simple prefix B+ tree.

The operations performed on B+ trees are essentially the same as those discussed for simple prefix B+ trees. Both B+ trees and simple prefix B+ trees consist of a set of records arranged in key order in a sequence set, coupled with an index set that provides rapid access to the block containing any particular key/record combination. The only difference is that in the simple prefix B+ tree we build an index set of shortest separators formed from key prefixes.

One of the reasons behind our decision to focus first on simple prefix B+ trees, rather than on the more general notion of a B+ tree, is that we want to distinguish between the role of the *separators* in the index set and *keys* in the sequence set. It is much more difficult to make this distinction when the separators are exact copies of the keys. By beginning with simple prefix B+ trees, we have the pedagogical advantage of working with separators that are clearly different from the keys in the sequence set.

But another reason for starting with simple prefix B+ trees is that they are quite often a more desirable alternative than the plain B+ tree. We want the index set to be as shallow as possible, which implies that we want to place as many separators into an index set block as we can. Why use anything longer than the simple prefix in the index set? In general, the answer to this question is that we do not, in fact, want to use anything longer than a simple prefix as a separator; consequently, simple prefix B+ trees are often a good solution. There are, however, at least two factors that might give favor to using a B+ tree that uses full copies of keys as separators.



ure 10.17 Formation of the first index set block in a B+ tree without the use of ortest separators.

- The reason for using shortest separators is to pack more of them into an index set block. As we have already said, this implies, ineluctably, the use of variable-length fields within the index set blocks. For some applications the cost of the extra overhead required to maintain and use this variable-length structure outweighs the benefits of shorter separators. In these cases one might choose to build a straightforward B+ tree using fixed-length copies of the keys from the sequence set as separators.
- Some key sets do not show much compression when the simple prefix method is used to produce separators. For example, suppose the keys consist of large, consecutive alphanumeric sequences such as 34C18K756, 34C18K757, 34C18K758, and so on. In this case, to enjoy appreciable compression, we need to use compression techniques that remove redundancy from the *front* of the key. Bayer and Unterauer (1977) describe such compression methods. Unfortunately, they are more expensive and complicated than simple prefix compression. If we calculate that tree height remains acceptable with the use of full copies of the keys as separators, we might elect to use the no-compression option.

9.11 B-Trees, B+ Trees, and Simple Prefix B+ Trees in Perspective

In this chapter and the preceding chapter we have looked at a number of tools used in building file structures. These tools—B-trees, B+ trees, and simple prefix B+ trees—have similar-sounding names and a number of common features. We need a way to differentiate these tools so we can reliably choose the most appropriate one for a given file structure job.

Before addressing this problem of differentiation, however, we should point out that these are not the only tools in the toolbox. Because B-trees, B+ trees, and their relatives are such powerful, flexible file structures, it is easy to fall into the trap of regarding them as the answer to all problems. This is a serious mistake. Simple index structures of the kind discussed in Chapter 7, which are maintained wholly in memory, are a much simpler, neater solution when they suffice for the job at hand. As we saw at the beginning of this chapter, simple memory indexes are not limited to direct access situations. This kind of index can be coupled with a sequence set of blocks to provide effective indexed sequential access as well. It is only when

the index grows so large that we cannot economically hold it in memory that we need to turn to paged index structures such as B-trees and B^+ trees.

In the chapter that follows we encounter yet another tool, known as *hashing*. Like simple memory-based indexes, hashing is an important alternative to B-trees, B^+ trees, and so on. In many situations, hashing can provide faster access to a very large number of records than the use of a member of the B-tree family can.

So, B-trees, B^+ trees, and simple prefix B^+ trees are not a panacea. However, they do have broad applicability, particularly for situations that require the ability to access a large file sequentially, in order by key, and through an index. All three of these tools share the following characteristics:

- They are all paged index structures, which means that they bring entire blocks of information into memory at once. As a consequence, it is possible to choose between a great many alternatives (for example, the keys for hundreds of thousands of records) with just a few seeks out to disk storage. The shape of these trees tends to be broad and shallow.
- All three approaches maintain height-balanced trees. The trees do not grow in an uneven way, which would result in some potentially long searches for certain keys.
- In all cases the trees grow from the bottom up. Balance is maintained through block splitting, merging, and redistribution.
- With all three structures it is possible to obtain greater storage efficiency through the use of two-to-three splitting and of redistribution in place of block splitting when possible. These techniques are described in Chapter 9.
- All three approaches can be implemented as virtual tree structures in which the most recently used blocks are held in memory. The advantages of virtual trees were described in Chapter 9.
- Any of these approaches can be adapted for use with variable-length records using structures inside a block similar to those outlined in this chapter.

For all of this similarity, there are some important differences. These differences are brought into focus through a review of the strengths and unique characteristics of each of these file structures.

B-Trees as Multilevel Indexes

The B-trees of Chapter 9 are multilevel indexes to data files that are entry-sequenced. This is the simplest type of B-tree to implement and is a very

efficient representation for most cases. The strengths of this approach are the simplicity of implementation, the inherent efficiency of indexing, and a maximization of the breadth of the B-tree. The major weakness of this strategy is the lack of organization of the data file, resulting in an excessive amount of seeking for sequential access.

B-Trees with Associated Information

This type of B-tree has not been discussed in any detail but was mentioned briefly in Section 9.9. These B-trees contain information that is grouped as a set of *pairs*. One member of each pair is the *key;* the other member is the *associated information.* These pairs are distributed over *all* the nodes of the B-tree. Consequently, we might find the information we are seeking at any level of the B-tree. This differs from the B-trees of Chapter 9 and B+ trees, which require all searches to proceed all the way down to the lowest, sequence set level of the tree. Because this type of B-tree contains the actual keys and associated information and there is therefore no need for additional storage to hold separators, a B-tree can take up less space than a B+ tree.

Given a large enough block size and an implementation that treats the tree as a virtual B-tree, it is possible to use a B-tree for ordered sequential access as well as for indexed access. The ordered sequential access is obtained through an in-order traversal of the tree. The implementation as a virtual tree is necessary so this traversal does not involve seeking as it returns to the next highest level of the tree. This use of a B-tree for indexed sequential access works only when the record information is stored within the B-tree. If the B-tree merely contains pointers to records that are in entry sequence off in some other file, then indexed sequential access is not workable because of all the seeking required to retrieve the record information.

B-trees are most attractive when the key comprises a large part of each record stored in the tree. When the key is only a small part of the record, it is possible to build a broader, shallower tree using the methods of Chapter 9.

B+ Trees

The primary difference between the B+ tree and the B-tree is that in the B+ tree all the key and record information is contained in a linked set of blocks known as the *sequence set.* The key and record information is *not* in the upper-level, treelike portion of the B+ tree. Indexed access to this

sequence set is provided through a conceptually (though not necessarily physically) separate structure called the *index set*. In a B+ tree the index set consists of copies of the keys that represent the boundaries between sequence set blocks. These copies of keys are called *separators* because they separate a sequence set block from its predecessor.

There are three significant advantages that the B+ tree structure provides over the B-tree:

- The sequence set can be processed in a truly linear, sequential way, providing efficient access to records in order by key; and
- The index is built with a single key or separator per block of data records instead of one key per data record. The size of the lowest-level index is reduced by the blocking factor of the data file. Since there are fewer keys, the index is smaller and hence shallower.

In practice, the latter of these two advantages is often the more important one. The impact of the first advantage is lessened by the fact that it is often possible to obtain acceptable performance during an in-order traversal of a B-tree through the page buffering mechanism of a virtual B-tree.

Simple Prefix B+ Trees

We just indicated that the primary advantage of using a B+ tree instead of a B-tree is that a B+ tree sometimes allows us to build a shallower tree because we have fewer keys in the index. The simple prefix B+ tree builds on this advantage by making the separators in the index set *smaller* than the keys in the sequence set, rather than just using copies of these keys. If the separators are smaller, we can fit more of them into a block to obtain a higher branching factor out of the block. In a sense, the simple prefix B+ tree takes one of the strongest features of the B+ tree one step farther.

The price we have to pay to obtain this separator compression and consequent increase in branching factor is that we must use an index set block structure that supports variable-length fields. The question of whether this price is worth the gain is one that has to be considered on a case-by-case basis.

SUMMARY

We begin this chapter by presenting a new problem. In previous chapters we provided either indexed access or sequential access in order by key, without finding an efficient way to provide both of these kinds of access.

This chapter explores one class of solutions to this problem, a class based on the use of a blocked sequence set and an associated index set.

The sequence set holds all of the file's data records in order by key. Since all insertion or deletion operations on the file begin with modifications to the sequence set, we start our study of indexed sequential file structures with an examination of a method for managing sequence set changes. The fundamental tools used to insert and delete records while still keeping everything in order within the sequence set are ones that we encountered in Chapter 9: block splitting, block merging, and redistribution of records between blocks. The critical difference between the use made of these tools for B-trees and the use made here is that there is no promotion of keys during block splitting in a sequence set. A sequence set is just a linked list of blocks, not a tree; therefore there is no place to promote anything to.

In this chapter, we also discuss the question of how large to make sequence set blocks. There is no precise answer we can give to this question since conditions vary between applications and environments. In general a block should be large, but not so large that we cannot hold several blocks in memory or read in a block without incurring the cost of a seek. In practice, blocks are often the size of a cluster (on sector-formatted disks) or the size of a single disk track.

Once we are able to build and maintain a sequence set, we turn to the matter of building an index for the blocks in the sequence set. If the index is small enough to fit in memory, one very satisfactory solution is to use a simple index that might contain, for example, the key for the last record in every block of the sequence set.

If the index set turns out to be too large to fit in memory, we recommend the use of the same strategy we developed in the preceding chapter when a simple index outgrows the available memory space: we turn the index into a B-tree. This combination of a sequence set with a B-tree index set is our first encounter with the structure known as a *B^+ tree.*

Before looking at B^+ trees as complete entities, we take a closer look at the makeup of the index set. The index set does not hold any information that we would ever seek for its own sake. Instead, an index set is used only as a road map to guide searches into the sequence set. The index set consists of *separators* that allow us to choose between sequence set blocks. There are many possible separators for any two sequence set blocks, so we might as well choose the *shortest separator.* The scheme we use to find this shortest separator consists of finding the common prefix of the two keys on either side of a block boundary in the sequence set and then going one

letter beyond this common prefix to define a true separator. A B+ tree with an index set made up of separators formed in this way is called a *simple prefix B+ tree.*

We study the mechanism used to maintain the index set as insertions and deletions are made in the sequence set of a B+ tree. The principal observation we make about all of these operations is that the primary action is within the *sequence set,* since that is where the records are. Changes to the index set are secondary; they are a byproduct of the fundamental operations on the sequence set. We add a new separator to the index set only if we form a new block in the sequence set; we delete a separator from the index set only if we remove a block from the sequence set through merging. Block overflow and underflow in the index set differ from the operations on the sequence set in that the index set is potentially a *multilevel* structure and is therefore handled as a B-tree.

The size of blocks in the index set is usually the same as the size chosen for the sequence set. To create blocks containing variable numbers of variable-length separators while at the same time supporting binary searching, we develop an internal structure for the block that consists of block header fields (for the separator count and total separator length), the variable-length separators, an index to these separators, and a vector of relative block numbers (RBNs) for the blocks descending from the index set block. This illustrates an important general principle about large blocks within file structures: they are more than just a slice out of a homogeneous set of records; blocks often have a sophisticated internal structure of their own, apart from the larger structure of the file.

We turn next to the problem of loading a B+ tree. We find that if we start with a set of records sorted by key, we can use a single-pass, sequential process to place these records into the sequence set. As we move from block to block in building the sequence set, we can extract separators and build the blocks of the index set. Compared with a series of successive insertions that work down from the top of the tree, this sequential loading process is much more efficient. Sequential loading also lets us choose the percentage of space utilized, right up to a goal of 100 percent.

The chapter closes with a comparison of B-trees, B+ trees, and simple prefix B+ trees. These are the primary advantages that B+ trees offer over B-trees:

- They support true indexed sequential access; and
- The index set contains fewer elements (one per data block instead of one per data record) and hence can be smaller and shallower.

We suggest that the second of these advantages is often the more important one, since treating a B-tree as a virtual tree provides acceptable indexed sequential access in many circumstances. The simple prefix B^+ tree takes this second advantage and carries it further, compressing the separators and potentially producing an even shallower tree. The price for this extra compression in a simple prefix B^+ tree is that we must deal with variable-length fields and a variable-order tree.

KEY TERMS

B^+ tree. A B^+ tree consists of a *sequence set* of records that are ordered sequentially by key, along with an *index set* that provides indexed access to the records. All of the records are stored in the sequence set. Insertions and deletions of records are handled by splitting, concatenating, and redistributing blocks in the sequence set. The index set, which is used only as a finding aid to the blocks in the sequence set, is managed as a B-tree.

Index set. The index set consists of *separators* that provide information about the boundaries between the blocks in the sequence set of a B^+ tree. The index set can locate the block in the sequence set that contains the record corresponding to a certain key.

Indexed sequential access. Indexed sequential access is not a single-access method but rather a term used to describe situations in which a user wants both sequential access to records, ordered by key, and indexed access to those same records. B^+ trees are just one method for providing indexed sequential access.

Separator. Separators are derived from the keys of the records on either side of a block boundary in the sequence set. If a given key is in one of the two blocks on either side of a separator, the separator reliably tells the user which of the two blocks holds the key.

Sequence set. The sequence set is the base level of an indexed sequential file structure, such as B^+ tree. It contains all of the records in the file. When read in logical order, block after block, the sequence set lists all of the records in order by key.

Shortest separator. Many possible separators can be used to distinguish between any two blocks in the sequence set. The class of shortest separators consists of those separators that take the least space, given a particular compression strategy. We looked carefully at a compression

strategy that consists of removing as many letters as possible from the rear of the separators, forming the shortest simple prefix that can still serve as a separator.

Simple prefix B+ tree. A B+ tree in which the index set is made up of shortest separators that are simple prefixes, as described in the definition for shortest separator.

Variable order. A B-tree is of variable order when the number of direct descendants from any given node of the tree is variable. This occurs when the B-tree nodes contain a variable number of keys or separators. This form is most often used when there is variability in the lengths of the keys or separators. Simple prefix B+ trees always make use of a variable-order B-tree as an index set so it is possible to take advantage of the compression of separators and place more of them in a block.

FURTHER READINGS

The initial suggestion for the B+ tree structure appears to have come from Knuth (1998), although he did not name or develop the approach. Most of the literature that discusses B+ trees in detail (as opposed to describing specific implementations is in the form of articles rather than textbooks. Comer (1979) provides what is perhaps the best brief overview of B+ trees. Bayer and Unterauer (1977) offer a definitive article describing techniques for compressing separators. The article includes consideration of simple prefix B+ trees as well as a more general approach called a *prefix B+ tree.* McCreight (1977) describes an algorithm for taking advantage of the variation in the lengths of separators in the index set of a B+ tree. McCreight's algorithm attempts to ensure that short separators, rather than longer ones, are promoted as blocks split. The intent is to shape the tree so blocks higher up in the tree have a greater number of immediate descendents, thereby creating a shallower tree.

Rosenberg and Snyder (1981) study the effects of initializing a compact B-tree on later insertions and deletions. B-trees are compared with more rigid indexed sequential file organizations (such as ISAM) in Batory (1981).

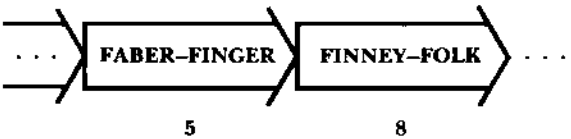
There are many commercial products that use methods related to the B+ tree operations described in this chapter, but detailed descriptions of their underlying file structures are scarce. An exception to this is IBM's Virtual Storage Access Method (VSAM), one of the most widely used

commercial products providing indexed sequential access. Wagner (1973) and Keehn and Lacy (1974) provide interesting insights into the early thinking behind VSAM. They also include considerations of key maintenance, key compression, secondary indexes, and indexes to multiple data sets. Good descriptions of VSAM can be found in several sources and from a variety of perspectives: in Comer (1979) (VSAM as an example of a B^+ tree), and Loomis (1989) (with examples from COBOL).

EXERCISES

1. Describe file structures that permit each of the following types of access: (a) sequential access only; (b) direct access only; (c) indexed sequential access.
2. A B^+ tree structure is generally superior to a B-tree for indexed sequential access. Since B^+ trees incorporate B-trees, why not use a B^+ tree whenever a hierarchical indexed structure is called for?
3. Consider the sequence set shown in Fig. 10.1(b). Show the sequence set after the keys *DOVER* and *EARNEST* are added; then show the sequence set after the key *DAVIS* is deleted. Did you use concatenation or redistribution for handling the underflow?
4. What considerations affect your choice of a block size for constructing a sequence set? If you know something about expected patterns of access (primarily sequential versus primarily random versus an even division between the two), how might this affect your choice of block size? On a sector-oriented drive, how might sector size and cluster size affect your choice of a block size?
5. It is possible to construct an indexed sequential file without using a tree-structured index. A simple index like the one developed in Chapter 7 could be used. Under what conditions might one consider using such an index? Under what conditions might it be reasonable to use a binary tree (such as an AVL tree) rather than a B-tree for the index?
6. The index set of a B^+ tree is just a B-tree, but unlike the B-trees discussed in Chapter 9, the separators do not have to be keys. Why the difference?
7. How does block splitting in the sequence set of a simple prefix B^+ tree differ from block splitting in the index set?

8. If the key *BOLEN* in the simple prefix B+ tree in Fig. 10.7 is deleted from the sequence set node, how is the separator BO in the parent node affected?
9. Consider the simple prefix B+ tree shown in Fig. 10.7. Suppose a key added to block 5 results in a split of block 5 and the consequent addition of block 8, so blocks 5 and 8 appear as follows:



   a. What does the tree look like after the insertion?
   b. Suppose that, subsequent to the insertion, a deletion causes underflow and the consequent concatenation of blocks 4 and 5. What does the tree look like after the deletion?
   c. Describe a case in which a deletion results in redistribution rather than concatenation, and show the effect it has on the tree.

10. Why is it often a good idea to use the same block size for the index set and the sequence set in a simple prefix B+ tree? Why should the index set nodes and the sequence set nodes usually be kept in the same file?
11. Show a conceptual view of an index set block, similar to the one illustrated in Fig. 10.11, that is loaded with the separators

    Ab Arch Astron B Bea

    Also show a more detailed view of the index block, as illustrated in Fig. 10.12.
12. If the initial set of records is sorted by key, the process of loading a B+ tree can be handled by using a single-pass sequential process instead of randomly inserting new records into the tree. What are the advantages of this approach?
13. Show how the simple prefix B+ tree in Fig. 10.16 changes after the addition of the node

    ITEMIZE–JAR

    Assume that the index set node containing the separators EF, H, and IG does not have room for the new separator but that there is room in the root.

14. Use the data stored in the simple prefix B+ tree in Fig. 10.16 to construct a B+ tree. Assume that the index set of the B+ tree is of order four. Compare the resulting B+ tree with the simple prefix B+ tree.

15. The use of variable-length separators and/or key compression changes some of the rules about how we define and use a B-tree and how we measure B-tree performance.
    a. How does it affect our definition of the order of a B-tree?
    b. Suggest criteria for deciding when splitting, concatenation, and redistribution should be performed.
    c. What difficulties arise in estimating simple prefix B+ tree height, maximum number of accesses, and space?

16. Make a table comparing B-trees, B+ trees, and simple prefix B+ trees in terms of the criteria listed below. Assume that the B-tree nodes do not contain data records, only keys and corresponding RRNs of data records. In some cases you will be able to give specific answers based on a tree's height or the number of keys in the tree. In other cases, the answers will depend on unknown factors, such as patterns of access or average separator length.
    a. The number of accesses required to retrieve a record from a tree of height h (average, best case, and worst case).
    b. The number of accesses required to insert a record (best and worst cases).
    c. The number of accesses required to delete a record (best and worst cases).
    d. The number of accesses required to process a file of n keys sequentially, assuming that each node can hold a maximum of k keys and a minimum of $k/2$ keys (best and worst cases).
    e. The number of accesses required to process a file of n keys sequentially, assuming that there are $h + 1$ node-sized buffers available.

17. Some commercially available indexed sequential file organizations are based on block interval splitting approaches very similar to those used with B+ trees. IBM's Virtual Storage Access Method (VSAM) offers the user several file access modes, one of which is called *key-sequenced* access and results in a file being organized much like a B+ tree. Look up a description of VSAM and report on how its key-sequenced organization relates to a B+ tree, as well as how it offers the user file-handling capabilities well beyond those of a straightforward B+ tree implementation. (See the Further Readings section of this chapter for articles and books on VSAM.)

18. Although B+ trees provide the basis for most indexed, sequential access methods now in use, this was not always the case. A method called ISAM (see Further Readings for this chapter) was once very common, especially on large computers. ISAM uses a rigid tree-structured index consisting of at least two and at most three levels. Indexes at these levels are tailored to the specific disk drive being used. Data records are organized by track, so the lowest level of an ISAM index is called the *track index*. Since the track index points to the track on which a data record can be found, there is one track index for each cylinder. When the addition of data records causes a track to overflow, the track is not split. Instead, the extra records are put into a separate overflow area and chained together in logical order. Hence, every entry in a track index may contain a pointer to the overflow area, in addition to its pointer to the home track.

The essential difference between the ISAM organization and B+ tree—like organizations—is in the way overflow records are handled. In the case of ISAM, overflow records are simply added to a chain of overflow records—the index structure is not altered. In the B+ tree case, overflow records are not tolerated. When overflow occurs, a block is split, and the index structure is altered to accommodate the extra data block.

Can you think of any advantages of using the more rigid index structure of ISAM, with separate overflow areas to handle overflow records? Why do you think B+ tree—like approaches—are replacing those that use overflow chains to hold overflow records? Consider the two approaches in terms of both sequential and direct access, as well as the addition and deletion of records.

PROGRAMMING EXERCISES

19. Design and implement a class `SequenceSet` in the style of class `BTree`. Your class should include methods `Add`, `Search`, and `Delete`.

Write a program that accepts a sorted file of strings as input. Your program should use this insert to build the strings into a `SequenceSet` with the following characteristics:

- The strings are stored in 15-byte records,
- A sequence set block is 128 bytes long, and
- Sequence set blocks are doubly linked.

20. Modify class `BTree` to support variable-sized keys, with the maximum number of keys per node determined by the actual size of the keys rather than by some fixed maximum.
21. Design and implement class `BplusTree`, which puts together the classes `SequenceSet` and `BTree`. B-tree characteristics should be maintained in the index set; the sequence set should, as before, be maintained so blocks are always at least half full. Consider the following suggestions:
    - Do not compress the keys as you form the separators for the index set.
    - Keep `BTree` nodes in the same file as the sequence set blocks. The header block should contain a reference to the root of the `BTree` as well as a reference to the beginning of the sequence set.
22. Write a test program that acts on the entire B+ tree that you created in the preceding exercise. Search, add, and delete capabilities should be tested, as they are in the earlier update program.

PROGRAMMING PROJECT

This is the eighth part of the programming project. We create a B+ tree of student records and of course registration records. This project depends on the successful completion of exercise 21.

23. Use class `BPlusTree` to create a B-tree index of a student record file with student identifier as key. Write a driver program to create a B-tree file from an existing student record file.
24. Use class `BTree` to create a B-tree index of a course registration record file with student identifier as key. Write a driver program to create a B-tree file from an existing course registration record file.
25. Write a program that opens a B+ tree student file and a B+ tree course registration file and retrieves information on demand. Prompt a user for a student identifier and print all objects that match it.

The next part of the programming project is in Chapter 12.