CHAPTER 9

Multilevel Indexing and B-Trees

CHAPTER OBJECTIVES

- Place the development of B-trees in the historical context of the problems they were designed to solve.
- Look briefly at other tree structures that might be used on secondary storage, such as paged AVL trees.
- Introduce multirecord and multilevel indexes and evaluate the speed of the search operation.
- Provide an understanding of the important properties possessed by B-trees and show how these properties are especially well suited to secondary storage applications.
- Present the object-oriented design of B-trees
  - Define class BTreeNode, the in-memory representation of the nodes of B-trees.
  - Define class BTree, the full representation of B-trees including all operations.
- Explain the implementation of the fundamental operations on B-trees.
- Introduce the notion of page buffering and virtual B-trees.
- Describe variations of the fundamental B-tree algorithms, such as those used to build B* trees and B-trees with variable-length records.

CHAPTER OUTLINE

9.1 Introduction: The Invention of the B-Tree
9.2 Statement of the Problem
9.3 Indexing with Binary Search Trees
9.3.1 AVL Trees
9.3.2 Paged Binary Trees
9.3.3 Problems with Paged Trees
9.4 Multilevel Indexing: A Better Approach to Tree Indexes
9.5 B-Trees: Working up from the Bottom
9.6 Example of Creating a B-Tree
9.7 An Object-Oriented Representation of B-Trees
9.7.1 Class BTreeNode: Representing B-Tree Nodes in Memory
9.7.2 Class BTree: Supporting Files of B-Tree Nodes
9.8 B-Tree Methods Search, Insert, and Others
9.8.1 Searching
9.8.2 Insertion
9.8.3 Create, Open, and Close
9.8.4 Testing the B-Tree
9.9 B-Tree Nomenclature
9.10 Formal Definition of B-Tree Properties
9.11 Worst-Case Search Depth
9.12 Deletion, Merging, and Redistribution
9.12.1 Redistribution
9.13 Redistribution During Insertion: A Way to Improve Storage Utilization
9.14 B* Trees
9.15 Buffering of Pages: Virtual B-Trees
9.15.1 LRU Replacement
9.15.2 Replacement Based on Page Height
9.15.3 Importance of Virtual B-Trees
9.16 Variable-Length Records and Keys

9.1 Introduction: The Invention of the B-Tree

Computer science is a young discipline. As evidence of this youth, consider that at the start of 1970, after astronauts had twice traveled to the moon, B-trees did not yet exist. Today, twenty-seven years later, it is hard to think of a major, general-purpose file system that is not built around a B-tree design.

Douglas Comer, in his excellent survey article, "The Ubiquitous B-Tree" (1979), recounts the competition among computer manufacturers and independent research groups in the late 1960s. The goal was the discovery of a general method for storing and retrieving data in large file systems that would provide rapid access to the data with minimal overhead cost. Among the competitors were R. Bayer and E. McCreight, who were working for Boeing Corporation. In 1972 they published an article, "Organization and Maintenance of Large Ordered Indexes," which announced B-trees to the world. By 1979, when Comer published his survey article, B-trees had already become so widely used that Comer was able to state that "the B-tree is, *de facto*, the standard organization for indexes in a database system."

We have reprinted the first few paragraphs of the 1972 Bayer and McCreight article[1] because it so concisely describes the facets of the problem that B-trees were designed to solve: how to access and efficiently maintain an index that is too large to hold in memory. You will remember that this is the same problem that is left unresolved in Chapter 7, on simple index structures. It will be clear as you read Bayer and McCreight's introduction that their work goes straight to the heart of the issues we raised in the indexing chapter.

> In this paper we consider the problem of organizing and maintaining an index for a dynamically changing random access file. By an *index* we mean a collection of index elements which are pairs (x, a) of fixed size physically adjacent data items, namely a key x and some associated information a. The key x identifies a unique element in the index, the associated information is typically a pointer to a record or a collection of records in a random access file. For this paper the associated information is of no further interest.
>
> We assume that the index itself is so voluminous that only rather small parts of it can be kept in main store at one time. Thus the bulk of the index must be kept on some backup store. The class of backup stores considered are *pseudo random access devices* which have rather long access or wait time—as opposed to a true random access device like core store—and a rather high data rate once the transmission of physically sequential data has been initiated. Typical pseudo random access devices are: fixed and moving head disks, drums, and data cells.
>
> Since the data file itself changes, it must be possible not only to search the index and to retrieve elements, but also to delete and to insert

1. From Acta-Informatica, 1:173–189, ©1972, Springer Verlag, New York. Reprinted with permission.

> keys—more accurately index elements—economically. The index organization described in this paper allows retrieval, insertion, and deletion of keys in time proportional to $\log_k I$ or better, where I is the size of the index, and k is a device dependent natural number which describes the page size such that the performance of the maintenance and retrieval scheme becomes near optimal.

Bayer and McCreight's statement that they have developed a scheme with retrieval time proportional to $\log_k I$, where k is related to the page size, is very significant. As we will see, the use of a B-tree with a page size of sixty-four to index an file with 1 million records results in being able to find the key for any record in no more than three seeks to the disk. A binary search on the same file can require as many as twenty seeks. Moreover, we are talking about getting this kind of performance from a system that requires only minimal overhead as keys are inserted and deleted.

Before looking in detail at Bayer and McCreight's solution, let's first return to a more careful look at the problem, picking up where we left off in Chapter 7. We will also look at some of the data and file structures that were routinely used to attack the problem before the invention of B-trees. Given this background, it will be easier to appreciate the contribution made by Bayer and McCreight's work.

One last matter before we begin: why the name *B-tree?* Comer (1979) provides this footnote:

> The origin of "B-tree" has never been explained by [Bayer and McCreight]. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees.

2 Statement of the Problem

The fundamental problem with keeping an index on secondary storage is, of course, that accessing secondary storage is slow. This can be broken down into two more specific problems:

- *Searching the index must be faster than binary searching.* Searching for a key on a disk often involves seeking to different disk tracks. Since seeks are expensive, a search that has to look in more than three or four locations before finding the key often requires more time than is desirable. If we are using a binary search, four seeks is enough only to differentiate among fifteen items. An average of about 9.5 seeks is

required to find a key in an index of one thousand items using a binary search. We need to find a way to home in on a key using fewer seeks.

- *Insertion and deletion must be as fast as search.* As we saw in Chapter 7, if inserting a key into an index involves moving a large number of the other keys in the index, index maintenance is very nearly impractical on secondary storage for indexes consisting of only a few hundred keys, much less thousands of keys. We need to find a way to make insertions and deletions that have only local effects in the index rather than requiring massive reorganization.

These were the two critical problems that confronted Bayer and McCreight in 1970. They serve as guideposts for steering our discussion of the use of tree structures and multilevel indexes for secondary storage retrieval.

9.3 Indexing with Binary Search Trees

Let's begin by addressing the second of these two problems: looking at the cost of keeping a list in sorted order so we can perform binary searches. Given the sorted list in Fig. 9.1, we can express a binary search of this list as a *binary search tree,* as shown in Fig. 9.2.

Using elementary data structure techniques, it is a simple matter to create nodes that contain right and left link fields so the binary search tree can be constructed as a linked structure. Figure 9.3 illustrates a linked representation of the first two levels of the binary search tree shown in Fig. 9.2. In each node, the left and right links point to the left and right *children* of the node.

What is wrong with binary search trees? We have already said that binary search is not fast enough for disk resident indexing. Hence, a binary search tree cannot solve our problem as stated earlier. However, this is not the only problem with binary search trees. Chief among these is the lack of an effective strategy of balancing the tree. That is, making sure that the height of the leaves of the tree is uniform: no leaf is much farther from the root than any other leaf. Historically, a number of attempts were made to solve these problems, and we will look at two of them: AVL trees and paged binary trees.

X CL DE FB FT HN JD KF NR PA RF SD TK WS YJ

igure 9.1 Sorted list of keys.

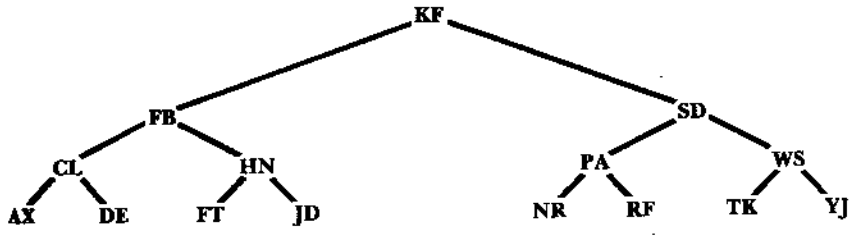


Figure 9.2 Binary search tree representation of the list of keys.

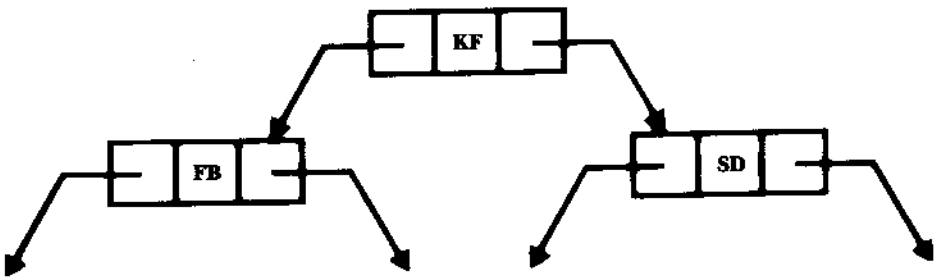


Figure 9.3 Linked representation of part of a binary search tree.

However, to focus on the costs and not the advantages is to miss the important new capability that this tree structure gives us: we no longer have to sort the file to perform a binary search. Note that the records in the file illustrated in Fig. 9.4 appear in random rather than sorted order. The sequence of the records in the file has no necessary relation to the structure of the tree; all the information about the logical structure is carried in the link fields. The very positive consequence that follows from this is that if we add a new key to the file, such as LV, we need only link it to the appropriate leaf node to create a tree that provides search performance that is as good as we would get with a binary search on a sorted list. The tree with LV added is illustrated in Fig. 9.5 (page 376).

Search performance on this tree is still good because the tree is in a *balanced* state. By balanced we mean that the height of the shortest path to a leaf does not differ from the height of the longest path by more than one level. For the tree in Fig. 9.5, this difference of one is as close as we can get to *complete balance*, in which all the paths from root to leaf are exactly the same length.

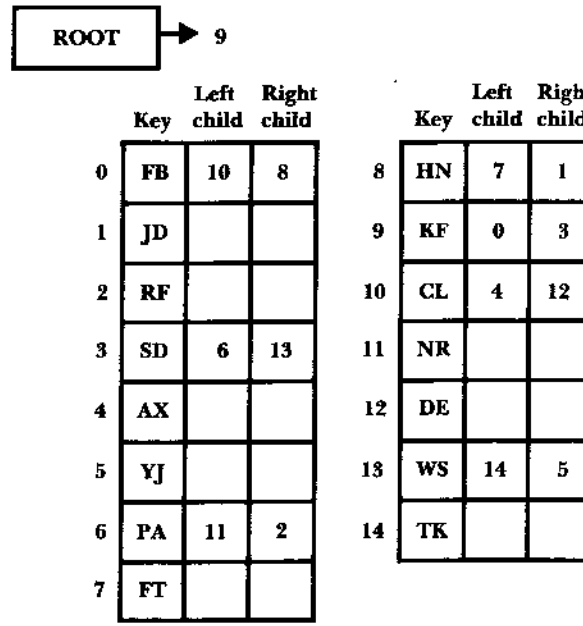


Figure 9.4 Record contents for a linked representation of the binary tree in Figure 9.2.

Consider what happens if we go on to enter the following eight keys to the tree in the sequence in which they appear:

```
NP MB TM LA UF ND TS NK
```

Just searching down through the tree and adding each key at its correct position in the search tree results in the tree shown in Fig. 9.6.

The tree is now out of balance. This is a typical result for trees that are built by placing keys into the tree as they occur without rearrangement. The resulting disparity between the length of various search paths is undesirable in any binary search tree, but it is especially troublesome if the nodes of the tree are being kept on secondary storage. There are now keys that require seven, eight, or nine seeks for retrieval. A binary search on a sorted list of these twenty-four keys requires only five seeks in the worst case. Although the use of a tree lets us avoid sorting, we are paying for this convenience in terms of extra seeks at retrieval time. For trees with hundreds of keys, in which an out-of-balance search path might extend to thirty, forty, or more seeks, this price is too high.

If each node is treated as a fixed-length record in which the link fields contain relative record numbers (RRNs) pointing to other nodes, then it is possible to place such a tree structure on secondary storage. Figure 9.4

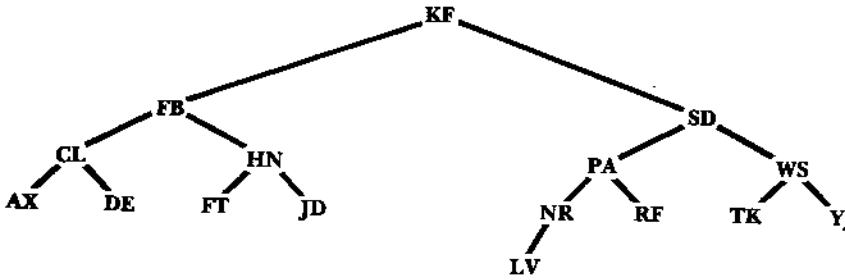


Figure 9.5 Binary search tree with LV added.

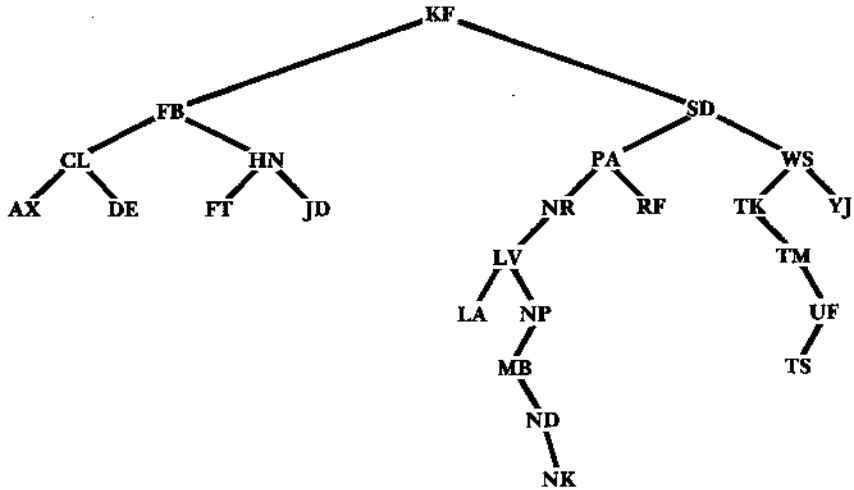


Figure 9.6 Binary search tree showing the effect of added keys.

illustrates the contents of the fifteen records that would be required to form the binary tree depicted in Fig. 9.2.

Note that more than half of the link fields in the file are empty because they are leaf nodes with no children. In practice, leaf nodes need to contain some special character, such as -1, to indicate that the search through the tree has reached the leaf level and that there are no more nodes on the search path. We leave the fields blank in this figure to make them more noticeable, illustrating the potentially substantial cost in terms of space utilization incurred by this kind of linked representation of a tree.

9.3.1 AVL Trees

Earlier we said that there is no *necessary* relationship between the order in which keys are entered and the structure of the tree. We stress the word *necessary* because it is clear that order of entry is, in fact, important in determining the structure of the sample tree illustrated in Fig. 9.6. The reason for this sensitivity to the order of entry is that, so far, we have just been linking the newest nodes at the leaf levels of the tree. This approach can result in some very undesirable tree organizations. Suppose, for example, that our keys consist of the letters A–G and that we receive these keys in alphabetical order. Linking the nodes as we receive them produces a degenerate tree that is, in fact, nothing more than a linked list, as illustrated in Fig. 9.7.

The solution to this problem is somehow to reorganize the nodes of the tree as we receive new keys, maintaining a near optimal tree structure. One elegant method for handling such reorganization results in a class of trees known as *AVL trees*, in honor of the pair of Russian mathematicians, G. M. Adel'son-Vel'skii and E. M. Landis, who first defined them. An AVL tree is a *height-balanced* tree. This means that there is a limit placed on the amount of difference allowed between the heights of any two subtrees sharing a common root. In an AVL tree the maximum allowable difference is one. An AVL tree is therefore called a *height-balanced 1-tree* or *HB(1) tree*. It is a member of a more general class of height-balanced trees known as *HB(k)* trees, which are permitted to be *k* levels out of balance.

The trees illustrated in Fig. 9.8 have the AVL, or HB(1) property. Note that no two subtrees of any root differ by more than one level. The trees in Fig. 9.9 are *not* AVL trees. In each of these trees, the root of the subtree that is not in balance is marked with an *X*.

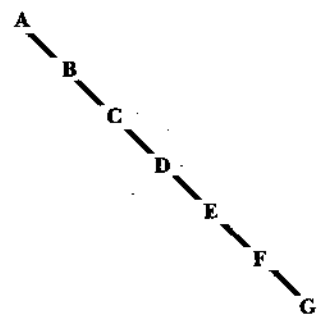


Figure 9.7
A degenerate tree.

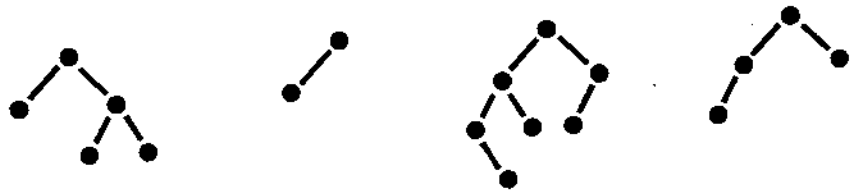


Figure 9.8 AVL trees.

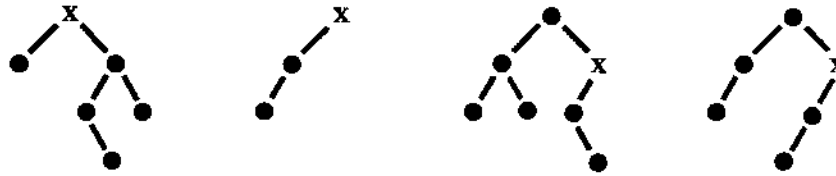


Figure 9.9 Trees that are not AVL trees.

The two features that make AVL trees important are

- By setting a maximum allowable difference in the height of any two subtrees, AVL trees guarantee a minimum level of performance in searching; and
- Maintaining a tree in AVL form as new nodes are inserted involves the use of one of a set of four possible rotations. Each of the rotations is confined to a single, local area of the tree. The most complex of the rotations requires only five pointer reassignments.

AVL trees are an important class of data structure. The operations used to build and maintain AVL trees are described in Knuth (1998), Standish (1980), and elsewhere. AVL trees are not themselves directly applicable to most file structure problems because, like all strictly *binary* trees, they have too many levels—they are too *deep*. However, in the context of our general discussion of the problem of accessing and maintaining indexes that are too large to fit in memory, AVL trees are interesting because they suggest that it is possible to define procedures that maintain height balance.

The fact that an AVL tree is height-balanced guarantees that search performance approximates that of a *completely balanced* tree. For example, the completely balanced form of a tree made up from the input keys

B C G E F D A

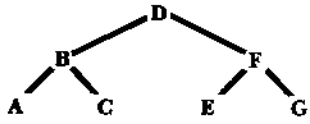


Figure 9.10
A completely balanced search tree.

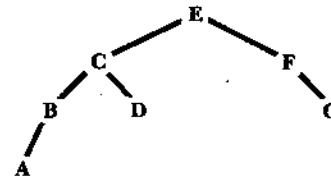


Figure 9.11 A search tree constructed using AVL procedures.

is illustrated in Fig. 9.10, and the AVL tree resulting from the same input keys, arriving in the same sequence, is illustrated in Fig. 9.11.

For a completely balanced tree, the worst-case search to find a key, given N possible keys, looks at

$$\log_2 (N+1)$$

levels of the tree. For an AVL tree, the worst-case search could look at

$$1.44 \log_2 (N+2)$$

levels. So, given 1 000 000 keys, a completely balanced tree requires seeking to 20 levels for some of the keys, but never to 21 levels. If the tree is an AVL tree, the maximum number of levels increases to only 29. This is a very interesting result, given that the AVL procedures guarantee that a single reorganization requires no more than five pointer reassignments. Empirical studies by VanDoren and Gray (1974), among others, have shown that such local reorganizations are required for approximately every other insertion into the tree and for approximately every fourth deletion. So height balancing using AVL methods guarantees that we will obtain a reasonable approximation of optimal binary tree performance at a cost that is acceptable in most applications using primary, random-access memory.

When we are using secondary storage, a procedure that requires more than five or six seeks to find a key is less than desirable; twenty or twenty-eight seeks is unacceptable. Returning to the two problems that we identified earlier in this chapter,

- Binary searching requires too many seeks, and
- Keeping an index in sorted order is expensive,

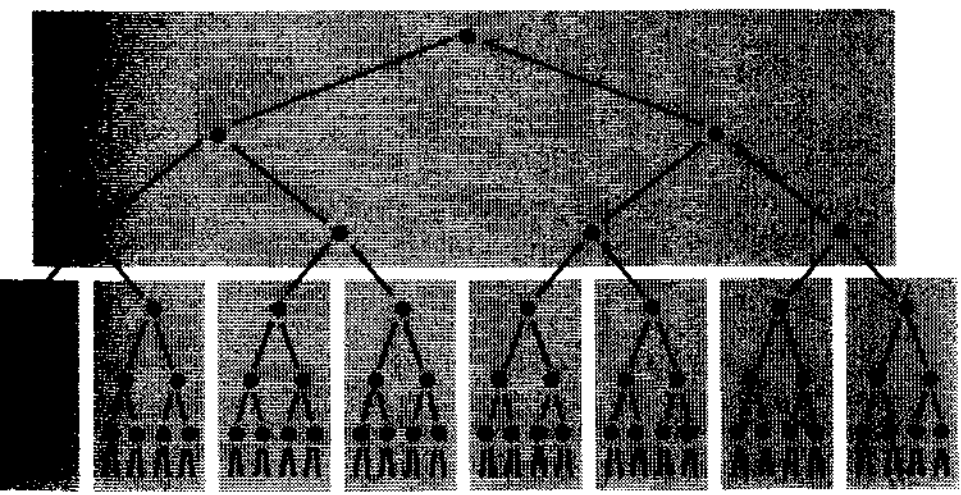
we can see that height-balanced trees provide an acceptable solution to the second problem. Now we need to turn our attention to the first problem.

9.3.2 Paged Binary Trees

Disk utilization of a binary search tree is extremely inefficient. That is, when we read a node of a binary search tree, there are only three useful pieces of information—the key value and the addresses of the left and right subtrees. Each disk read produces a minimum of a single page—at least 512 bytes. Reading a binary node wastes most of the data read from the disk. Since this disk read is the critical factor in the cost of searching, we cannot afford to waste the reads. It is imperative that we choose an index record that uses all of the space read from the disk.

The paged binary tree attempts to address the problem by locating multiple binary nodes on the same disk page. In a paged system, you do not incur the cost of a disk seek just to get a few bytes. Instead, once you have taken the time to seek to an area of the disk, you read in an entire page from the file. This page might consist of a great many individual records. If the next bit of information you need from the disk is in the page that was just read in, you have saved the cost of a disk access.

Paging, then, is a potential solution to the inefficient disk utilization of binary search trees. By dividing a binary tree into pages and then storing each page in a block of contiguous locations on disk, we should be able to reduce the number of seeks associated with any search. Figure 9.12 illustrates such a paged tree. In this tree we are able to locate any one



ure 9.12 Paged binary tree.

of the 63 nodes in the tree with no more than two disk accesses. Note that every page holds 7 nodes and can branch to eight new pages. If we extend the tree to one additional level of paging, we add sixty-four new pages; we can then find any one of 511 nodes in only three seeks. Adding yet another level of paging lets us find any one of 4095 nodes in only four seeks. A binary search of a list of 4095 items can take as many as twelve seeks.

Clearly, breaking the tree into pages has the potential to result in faster searching on secondary storage, providing us with much faster retrieval than any other form of keyed access that we have considered up to this point. Moreover, our use of a page size of seven in Fig. 9.12 is dictated more by the constraints of the printed page than by anything having to do with secondary storage devices. A more typical example of a page size might be 8 kilobytes, capable of holding 511 key/reference field pairs. Given this page size and assuming that each page contains a completely balanced full tree and that the pages are organized as a completely balanced full tree, it is then possible to find any one of 134 217 727 keys with only three seeks. That is the kind of performance we are looking for. Note that, while the number of seeks required for a worst-case search of a completely full, balanced binary tree is

$$\log_2 (N + 1)$$

where N is the number of keys in the tree, the number of seeks required for the *paged* versions of a completely full, balanced tree is

$$\log_{k+1} (N + 1)$$

where N is, once again, the number of keys. The new variable, k, is the number of keys held in a single page. The second formula is actually a generalization of the first, since the number of keys in a page of a purely binary tree is 1. It is the logarithmic effect of the page size that makes the impact of paging so dramatic:

$$\log_2 (134\,217\,727 + 1) = 27 \text{ seeks}$$
$$\log_{511+1} (134\,217\,727 + 1) = 3 \text{ seeks}$$

The use of large pages does not come free. Every access to a page requires the transmission of a large amount of data, most of which is not used. This extra transmission time is well worth the cost, however, because it saves so many seeks, which are far more time-consuming than the extra transmissions. A much more serious problem, which we look at next, has to do with keeping the paged tree organized.

9.3.3 Problems with Paged Trees

The major problem with paged trees is still inefficient disk usage. In the example in Fig. 9.12, there are seven tree nodes per page. Of the fourteen reference fields in a single page, six of them are reference nodes within the page. That is, we are using fourteen reference fields to distinguish between eight subtrees. We could represent the same information with seven key fields and eight subtree references. A significant amount of the space in the node is still being wasted.

Is there any advantage to storing a binary search tree within the page? It's true that in doing so we can perform binary search. However, if the keys are stored in an array, we can still do our binary search. The only problem here is that insertion requires a linear number of operations. We have to remember, however, that the factor that determines the cost of search is the number of disk accesses. We can do almost anything in memory in the time it takes to read a page. The bottom line is that there is no compelling reason to produce a tree inside the page.

The second problem, if we decide to implement a paged tree, is how to build it. If we have the entire set of keys in hand before the tree is built, the solution to the problem is relatively straightforward: we can sort the list of keys and build the tree from this sorted list. Most important, if we plan to start building the tree from the root, we know that the middle key in the sorted list of keys should be the *root key* within the *root page* of the tree. In short, we know where to begin and are assured that this beginning point will divide the set of keys in a balanced manner.

Unfortunately, the problem is much more complicated if we are receiving keys in random order and inserting them as soon as we receive them. Assume that we must build a paged tree as we receive the following sequence of single-letter keys:

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

We will build a paged binary tree that contains a maximum of three keys per page. As we insert the keys, we rotate them within a page as necessary to keep each page as balanced as possible. The resulting tree is illustrated in Fig. 9.13. Evaluated in terms of the depth of the tree (measured in pages), this tree does not turn out too badly. (Consider, for example, what happens if the keys arrive in alphabetical order.)

Even though this tree is not dramatically misshapen, it clearly illustrates the difficulties inherent in building a paged binary tree from the top down. When you start from the root, the initial keys must, of necessity, go into the root. In this example at least two of these keys, C and D, are not

keys that we want there. They are adjacent in sequence and tend toward the beginning of the total set of keys. Consequently, they force the tree out of balance.

Once the wrong keys are placed in the root of the tree (or in the root of any subtree farther down the tree), what can you do about it? Unfortunately, there is no easy answer to this. We cannot simply rotate entire pages of the tree in the same way that we would rotate individual keys in an unpaged tree. If we rotate the tree so the initial root page moves down to the left, moving the C and D keys into a better position, then the S key is out of place. So we must break up the pages. This opens up a whole world of possibilities and difficulties. Breaking up the pages implies rearranging them to create new pages that are both internally balanced and well arranged relative to other pages. Try creating a page rearrangement algorithm for the simple, three-keys-per-page tree from Fig. 9.13. You will find it very difficult to create an algorithm that has only local effects, rearranging just a few pages. The tendency is for rearrangements and adjustments to spread out through a large part of the tree. This situation grows even more complex with larger page sizes.

So, although we have determined that collecting keys into pages is a very good idea from the standpoint of reducing seeks to the disk, we have

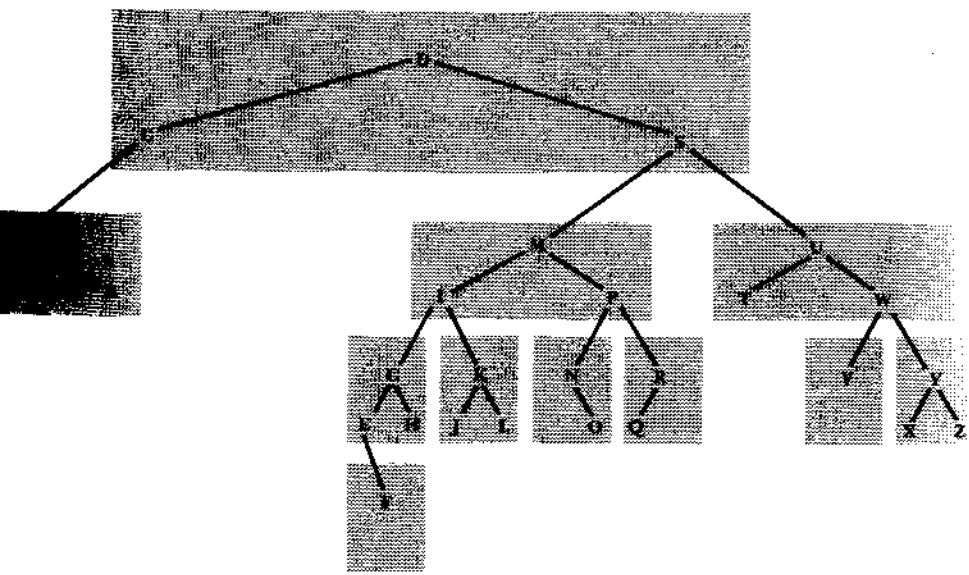


Figure 9.13 Paged tree constructed from keys arriving in random input sequence.

not yet found a way to collect the right keys. We are still confronting at least two unresolved questions:

- How do we ensure that the keys in the root page turn out to be good *separator* keys, dividing up the set of other keys more or less evenly?
- How do we avoid grouping keys, such as C, D, and S in our example, that should not share a page?

There is, in addition, a third question that we have not yet had to confront because of the small page size of our sample tree:

- How can we guarantee that each of the pages contains at least some minimum number of keys? If we are working with a larger page size, such as 8191 keys per page, we want to avoid situations in which a large number of pages each contains only a few dozen keys.

Bayer and McCreight's 1972 B-tree article provides a solution directed precisely at these questions.

A number of the elegant, powerful ideas used in computer science have grown out of looking at a problem from a different viewpoint. B-trees are an example of this viewpoint-shift phenomenon.

The key insight required to make the leap from the kinds of trees we have been considering to a new solution, B-trees, is that we can choose to *build trees upward from the bottom instead of downward from the top.* So far, we have assumed the necessity of starting construction from the root as a given. Then, as we found that we had the wrong keys in the root, we tried to find ways to repair the problem with rearrangement algorithms. Bayer and McCreight recognized that the decision to work down from the root was, of itself, the problem. Rather than finding ways to undo a bad situation, they decided to avoid the difficulty altogether. With B-trees, you allow the root to *emerge,* rather than set it up and then find ways to change it.

4 Multilevel Indexing: A Better Approach to Tree Indexes

The previous section attempted to develop an ideal strategy for indexing large files based on building search trees, but serious flaws were uncovered. In this section we take a different approach. Instead of basing our strategy on binary tree searches, we start with the single record indexing strategy of

Chapter 7. We extend this to multirecord indexes and then multilevel indexes. Ultimately, this approach, too, is flawed, but it is the source of the primary efficiency of searching and leads us directly to B-trees.

In Chapter 7, we noted that a single record index puts a limit on the number of keys allowed and that large files need multirecord indexes. A multirecord index consists of a sequence of simple index records. The keys in one record in the list are all smaller than the keys of the next record. A binary search is possible on a file that consists of an ordered sequence of index records, but we already know that binary search is too expensive.

To illustrate the benefits of an indexed approach, we use the large example file of Chapter 8, an 80-megabyte file of 8 000 000 records, 100 bytes each, with 10-byte keys. An index of this file has 8 000 000 key-reference pairs divided among a sequence of index records. Let's suppose that we can put 100 key-reference pairs in a single index record. Hence there are 80 000 records in the index. In order to build the index, we need to read the original file, extract the key from each record, and sort the keys. The strategies outlined in Chapter 8 can be used for this sorting. The 100 largest keys are inserted into an index record, and that record is written to the index file. The next largest 100 keys go into the next record of the file, and so on. This continues until we have 80 000 index records in the index file. Although we have reduced the number of records to be searched by a factor of 100, we still must find a way to speed up the search of this 80 000-record file.

Can we build an index of the index file, and how big will it be? Since the index records form a sorted list of keys, we can choose one of the keys (for example, the largest) in each index record as the key of that whole record. These *second-level* keys can be used to build a second-level index with 80 000 keys, or 800 index records. In searching the second-level index for a key k, we choose the smallest second-level key that is greater than or equal to k. If k is in the first-level index, it must be in the block referenced by that second-level key.

Continuing to a third level, we need just 8 index records to index the largest keys in the 800 second-level records. Finally, the fourth level consists of a single index record with only 8 keys. These four levels together form an index tree with a fan-out of 100 and can be stored in a single index file. Each node of the tree is an index record with 100 children. Each of the children of a node is itself an index node, except at the leaves. The children of the leaf nodes are data records.

A single index file containing the full four-level index of 8 000 000 records requires 80 809 index records, each with 100 key-reference

pairs. The lowest level index is an index to the data file, and its reference fields are record addresses in the data file. The other indexes use their reference fields for index record addresses, that is, addresses within the index file.

The costs associated with this multilevel index file are the space overhead of maintaining the extra levels, the search time, and the time to insert and delete elements. The space overhead is 809 more records than the 80 000 minimum for an index of the data file. This is just 1 percent. Certainly this is not a burden.

The search time is simple to calculate—it's three disk reads! An analysis of search time always has multiple parts: the minimum search time, the maximum search time, and the average search time for keys that are in the index and for keys that are not in the index. For this multilevel index, all of these cases require searching four index records. That is, each level of the index must be searched. For a key that is in the index, we need to search all the way to the bottom level to get the data record address. For a key not in the index, we need to search all the way to the bottom to determine that it is missing. The average, minimum, and maximum number of index blocks to search are all four, that is, the number of levels in the index. Since there is only one block at the top level, we can keep that block in memory. Hence, a maximum of three disk accesses are required for any key search. It might require fewer disk reads if any of the other index records are already in memory.

Look how far we've come: an arbitrary record in an 80-megabyte file can be read with just four disk accesses—three to search the index and one to read the data record. The total space overhead, including the primary index, is well below 10 percent of the data file size. This tree is not full, since the root node has only eight children and can accommodate one hundred. This four-level tree will accommodate twelve times this many data records, or a total of 100 million records in a file of 10 gigabytes. Any one of these records can be found with only three disk accesses. This is what we need to produce efficient indexed access!

The final factor in the cost of multilevel indexes is the hardest one. How can we insert keys into the index? Recall that the first-level index is an ordered sequence of records. Does this imply that the index file must be sorted? The search strategy relies on indexes and record addresses, not on record placement in the file. As with the simple indexes of Chapter 7, this indexed search supports entry-sequenced records. As long as the location of the highest level index record is known, the other records can be anywhere in the file.

Having an entry-sequenced index file does not eliminate the possibility of linear insertion time. For instance, suppose a new key is added that will be the smallest key in the index. This key must be inserted into the first record of the first-level index. Since that record is already full with one hundred elements, its largest key must be inserted into the second record, and so on. Every record in the first-level index must be changed. This requires 80 000 reads and writes. This is truly a fatal flaw in simple multilevel indexing.

9.5 B-Trees: Working up from the Bottom

B-trees are multilevel indexes that solve the problem of linear cost of insertion and deletion. This is what makes B-trees so good, and why they are now the standard way to represent indexes. The solution is twofold. First, don't require that the index records be full. Second, don't shift the overflow keys to the next record; instead split an overfull record into two records, each half full. Deletion takes a similar strategy of merging two records into a single record when necessary.

Each node of a B-tree is an index record. Each of these records has the same maximum number of key-reference pairs, called the *order* of the B-tree. The records also have a minimum number of key-reference pairs, typically half of the order. A B-tree of order one hundred has a minimum of fifty keys and a maximum of one hundred keys per record. The only exception is the single root node, which can have a minimum of two keys.

An attempt to insert a new key into an index record that is not full is cheap. Simply update the index record. If the new key is the new largest key in the index record, it is the new higher-level key of that record, and the next higher level of the index must be updated. The cost is bounded by the height of the tree.

When insertion into an index record causes it to be overfull, it is split into two records, each with half of the keys. Since a new index node has been created at this level, the largest key in this new node must be inserted into the next higher level node. We call this the *promotion* of the key. This promotion may cause an overflow at that level. This in turn causes that node to be split, and a key promoted to the next level. This continues as far as necessary. If the index record at the highest level overflows, it must be split. This causes another level to be added to the multilevel index. In this way, a B-tree grows up from the leaves. Again the cost of insertion is bounded by the height of the tree.

The rest of the secrets of B-trees are just working out the details. How to split nodes, how to promote keys, how to increase the height of the tree, and how to delete keys.

6 Example of Creating a B-Tree

Let's see how a B-tree grows given the key sequence that produces the paged binary tree illustrated in Fig. 9.13. The sequence is

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

We use an order four B-tree (maximum of four key-reference pairs per node). Using such a small node size has the advantage of causing pages to split more frequently, providing us with more examples of splitting. We omit explicit indication of the reference fields so we can fit a larger tree on the printed page.

Figure 9.14 illustrates the growth of the tree up to the point where it is about to split the root for the second time. The tree starts with a single empty record. In Fig. 9.14(a), the first four keys are inserted into that record. When the fifth key, A, is added in Fig. 9.14(b), the original node is split and the tree grows by one level as a new root is created. The keys in the root are the largest key in the left leaf, D, and the largest key in the right leaf, T.

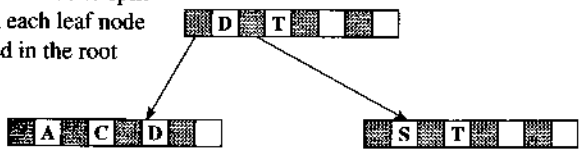
The keys M, P, and I all belong in the rightmost leaf node, since they are larger than the largest key in the right node. However, inserting I makes the rightmost leaf node overfull, and it must be split, as shown in Fig. 9.14(c). The largest key in the new node, P, is inserted into the root. This process continues in Figs. 9.14(d) and (e), where B, W, N, G, and U are inserted.

In the tree of Fig. 9.14(e), the next key in the list, R, should be put into the rightmost leaf node, since it is greater than the largest key in the previous node, P, and less than or equal to the largest key in that node, W. However, the rightmost leaf node is full, and so is the root. Splitting that leaf node will overfill the root node. At this point a new root must be created, and the height of the tree increased to three.

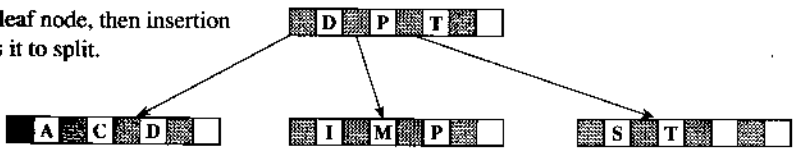
Figure 9.15 shows the tree as it grows to height three. The figure also shows how the tree continues to grow as the remaining keys in the sequence are added. Figure 9.15(b) stops after Z is added. The next key in the sequence, F, requires splitting the second-leaf node, as shown in Fig. 9.15(c). Although the leaf level of the tree is not shown in a single line, it is still a single level. Insertions of X and V causes the rightmost leaf to be

) Insertions of C, S, D, T
no the initial node.

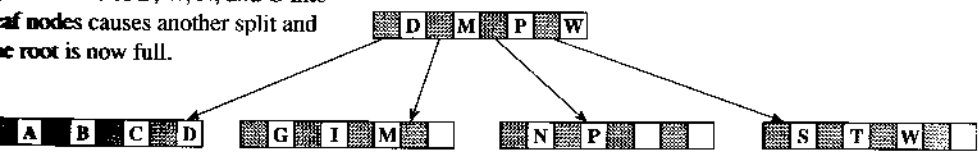
) Insertion of A causes node to split
nd the largest key in each leaf node
D and T) to be placed in the root
ode.



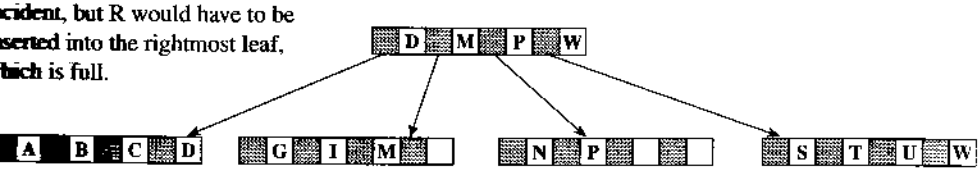
) M and P are inserted into the
ghtmost leaf node, then insertion
f I causes it to split.



) Insertions of B, W, N, and G into
af nodes causes another split and
e root is now full.

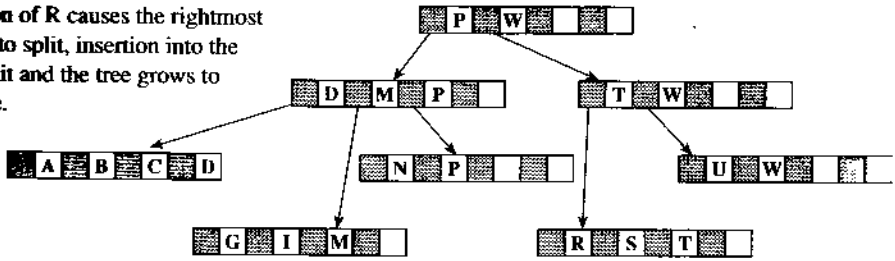


) Insertion of U proceeds without
cident, but R would have to be
serted into the rightmost leaf,
hich is full.

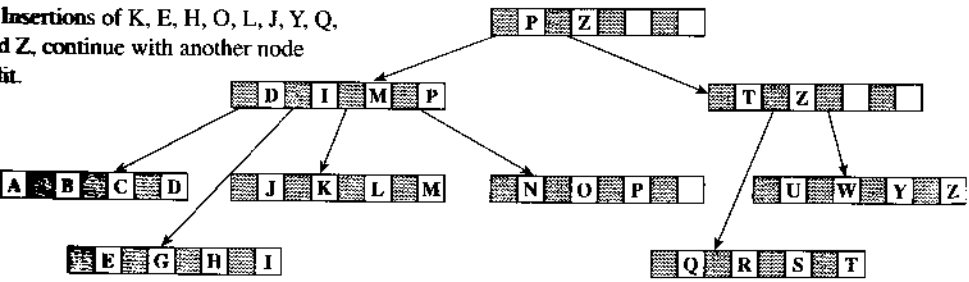


igure 9.14 Growth of a B-tree, part 1. The tree grows to a point at which the root needs
be split the second time.

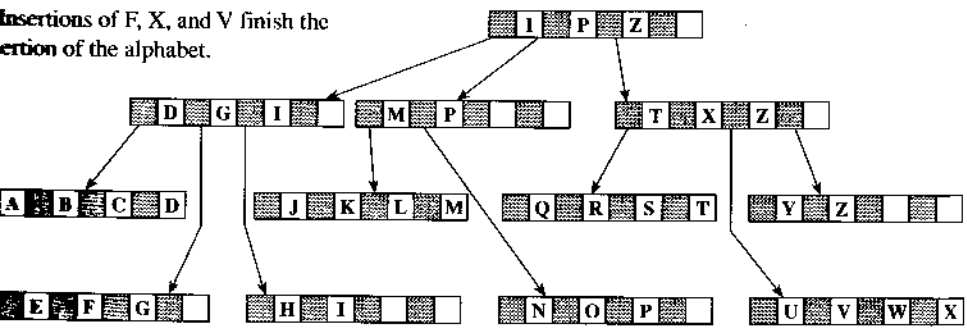
Insertion of R causes the rightmost
f node to split, insertion into the
ot to split and the tree grows to
vel three.



Insertions of K, E, H, O, L, J, Y, Q,
d Z, continue with another node
it.



Insertions of F, X, and V finish the
ertion of the alphabet.



ure 9.15 Growth of a B-tree, part 2. The root splits to level three; remaining keys are
erted.

overfull and split. The rightmost leaf of the middle level is also overfull and is split. All twenty-six letters are inserted into a tree of height three and order four.

Note that the number of nodes affected by any insertion is never more than two nodes per level (one changed and a new one created by a split), so the insertion cost is strictly linear in the height of the tree.

9.7 An Object-Oriented Representation of B-Trees

9.7.1 Class BTreeNode: Representing B-Tree Nodes in Memory

As we have seen, a B-tree is an index file associated with a data file. Most of the operations on B-trees, including insertion and deletion, are applied to the B-tree nodes in memory. The B-tree file simply stores the nodes when they are not in memory. Hence, we need a class to represent the memory resident B-tree nodes. Class `BTreeNode`, given in Fig. 9.16 and in file `btnode.h` of Appendix I, is a template class based on the `SimpleIndex` template class that was described in Section 7.4.3. Note that a `BTreeNode` object has methods to insert and remove a key and to split and merge nodes. There are also protected members that store the file address of the node and the minimum and maximum number of keys. You may notice that there is no search method defined in the class. The search method of the base class `SimpleIndex` works perfectly well.

It is important to note that not every data member of a `BTreeNode` has to be stored when the object is not in memory. The difference between the memory and the disk representations of `BTreeNode` objects is managed by the pack and unpack operations.

Class `BTreeNode` is designed to support some optimizations of the in-memory operations. For example, the number of keys is actually one more than the order of the tree, as shown in the constructor. The call to the `SimpleIndex` constructor creates an index record with `maxKeys+1` elements:

```
template <class keyType>
BTreeNode<keyType>::BTreeNode(int maxKeys, int unique)
   :SimpleIndex<keyType>(maxKeys+1, unique)
{  Init();}
```

```
Late <class keyType>
s BTreeNode: public SimpleIndex <keyType>
his is the in-memory version of the BTreeNode
lic:
reeNode(int maxKeys, int unique = 1);
t Insert (const keyType key, int recAddr);
t Remove (const keyType key, int recAddr = -1);
t LargestKey (); // returns value of Largest key
t Split (BTreeNode<keyType>*newNode);//move into newNode
t Pack (IOBuffer& buffer) const;
t Unpack (IOBuffer& buffer);
ected:
t MaxBKeys; // maximum number of keys in a node
t Init ();
iend class BTree<keyType>;
```

e 9.16 The main members and methods of class BTreeNode: template class for node in memory.

For this class, the order of the B-tree node (member `MaxBKeys`) is one less than the value of `MaxKeys`, which is a member of the base class `SimpleIndex`. Making the index record larger allows the `Insert` method to create an overfull node. The caller of `BTreeNode::Insert` needs to respond to the overflow in an appropriate fashion. Similarly, the `Remove` method can create an underfull node.

Method `Insert` simply calls `SimpleIndex::Insert` and then checks for overflow. The value returned is 1 for success, 0 for failure, and −1 for overflow:

```
ate <class keyType>
reeNode<keyType>::Insert (const keyType key, int recAddr)

t result = SimpleIndex<keyType>::Insert (key, recAddr);
(!result) return 0; // insert failed
(NumKeys > MaxBKeys) return -1; // node overflow
turn 1;
```

9.7.2 Class BTree: Supporting Files of B-Tree Nodes

We now look at class `BTree` which uses in-memory `BTreeNode` objects, adds the file access portion, and enforces the consistent size of the nodes. Figure 9.17 and file `btree.h` of Appendix I contain the definition of class `BTree`. Here are methods to create, open, and close a B-tree and to search, insert, and remove key-reference pairs. In the protected area of the class, we find methods to transfer nodes from disk to memory (`Fetch`) and back to disk (`Store`). There are members that hold the root node in memory and represent the height of the tree and the file of index records. Member `Nodes` is used to keep a collection of tree nodes in memory and reduce disk accesses, as will be explained later.

```
template <class keyType>
class BTree
public:
   BTree(int order, int keySize=sizeof(keyType), int unique=1);
   int Open (char * name, int mode);
   int Create (char * name, int mode);
   int Close ();
   int Insert (const keyType key, const int recAddr);
   int Remove (const keyType key, const int recAddr = -1);
   int Search (const keyType key, const int recAddr = -1);
rotected:
   typedef BTreeNode<keyType> BTNode; // necessary shorthand
   BTNode * FindLeaf (const keyType key);
      // load a branch into memory down to the leaf with key
   BTNode * Fetch(const int recaddr);//load node into memory
   int Store (BTNode *);// store node into file
   BTNode Root;
   int Height; // height of tree
   int Order; // order of tree
   BTNode ** Nodes; // storage for a branch
   // Nodes[1] is level 1, etc. (see FindLeaf)
   // Nodes[Height-1] is leaf
   RecordFile<BTNode> BTreeFile;
;
```

igure 9.17 Main members and methods of class BTree: whole B-tree implementation—cluding methods Create, Open, Search, Insert, and Remove.

B-Tree Methods Search, Insert, and Others

Now that we have seen the principles of B-tree operations and we have the class definitions and the single node operations, we are ready to consider the details of the B-tree methods.

9.8.1 Searching

The first B-tree method we examine is a tree-searching procedure. Searching is a good place to begin because it is relatively simple, yet it still illustrates the characteristic aspects of most B-tree algorithms:

- They are iterative, and
- They work in two stages, operating alternatively on entire pages (class `BTree`) and then *within* pages (class `BTreeNode`).

The searching procedure is iterative, loading a page into memory and then searching through the page, looking for the key at successively lower levels of the tree until it reaches the leaf level. Figure 9.18 contains the code for method `Search` and the protected method `FindLeaf` that does almost all of the work. Let's work through the methods by hand, searching for the key L in the tree illustrated in Fig. 9.15(a). For an object `btree` of type `BTree<char>` and an integer `recAddr`, the following code finds that there is no data file record with key L:

```
recAddr = btree.Search ('L');
```

Method `Search` calls method `FindLeaf`, which searches down a branch of the tree, beginning at the root, which is referenced by the pointer value `Nodes[0]`. In the first iteration, with `level` = 1, the line

```
recAddr = Nodes[level-1]->Search(key,-1,0);
```

is an inexact search and finds that L is less than P, the first key in the record. Hence, `recAddr` is set to the first reference in the root node, which is the index file address of the first node in the second level of the tree of Fig. 9.15(a). The line

```
Nodes[level]=Fetch(recAddr);
```

reads that second-level node into a new `BTreeNode` object and makes `Nodes[1]` point to this new object. The second iteration, with `level` = 2, searches for L in this node. Since L is less than M, the second key in the

```
=plate <class keyType>
t BTree<keyType>::Search (const keyType key, const int recAddr)

  BTreeNode<keyType> * leafNode;
  leafNode = FindLeaf (key);
  return leafNode -> Search (key, recAddr);


=plate <class keyType>
reeNode<keyType> * BTree<keyType>::FindLeaf (const keyType key)
 load a branch into memory down to the leaf with key

  int recAddr, level;
  for (level = 1; level < Height; level++)
  {
     recAddr = Nodes[level-1]->Search(key,-1,0);//inexact search
     Nodes[level]=Fetch(recAddr);
  }
  return Nodes[level-1];
```

gure 9.18 Method BTree::Search and BTree::FindLeaf.

record, the second reference is selected, and the second node in the leaf level of the tree is loaded into `Nodes[2]`. After the `for` loop increments `level`, the iteration stops, and `FindLeaf` returns the address of this leaf node. At the end of this method, the array `Nodes` contains pointers to the complete branch of the tree.

After `FindLeaf` returns, method `Search` uses an exact search of the leaf node to find that there is no data record that has key L. The value returned is -1.

Now let's use method `Search` to look for G, which *is* in the tree of Fig. 9.15(a). It follows the same downward path that it did for L, but this time the exact search in method `Search` finds G in position 1 of the second-leaf node. It returns the first reference field in the node, which is the data file address of the record with key G.

9.8.2 Insertion

There are two important observations we can make about the insertion, splitting, and promotion process:

- It begins with a search that proceeds all the way down to the leaf level, and
- After finding the insertion location at the leaf level, the work of insertion, overflow detection, and splitting proceeds upward from the bottom.

Consequently, we can conceive of our iterative procedure as having three phases:

1. Search to the leaf level, using method `FindLeaf`, before the iteration;
2. Insertion, overflow detection, and splitting on the upward path;
3. Creation of a new root node, if the current root was split.

Let's use the example of inserting R and its data record address (called `recAddr`) into the tree of Fig. 9.14(e) so we can watch the insertion procedure work through these phases. The result of this insertion is shown in Fig. 9.15(a). Method `Insert` is the most complicated of the methods included in file `btree.tc` in Appendix I. We will look at some of its code here.

The first operation in method `Insert` is to search to the root for key R using `FindLeaf`:

```
thisNode = FindLeaf (key);
```

As described above, `FindLeaf` loads a complete branch into memory. In this case, `Nodes[0]` is the root node, and `Nodes[1]` is the rightmost leaf node (containing S, T, U, and W).

The next step is to insert *R* into the leaf node

```
result = thisNode -> Insert (key, recAddr);
```

The result here is that an overflow is detected. The object `thisNode` now has five keys. The node must be split into two nodes, using the following code:

```
newNode = NewNode();
thisNode -> Split (newNode);
Store(thisNode); Store(newNode);
```

Now the two nodes, one with keys R, S, and T, and one with U and W, have been stored back in the file. We are done with the leaf level and are ready to move up the tree.

The next step is to update the parent node. Since the largest key in `thisNode` has changed, method `UpdateKey` is used to record the change (`largestKey` has been set to the previous largest key in `thisNode`):

```
arentNode->UpdateKey(largestKey, thisNode->LargestKey());
```

Hence the value W in the root is changed to T. Then the largest value in the new node is inserted into the root of the tree:

```
arentNode->Insert(newNode->LargestKey(), newNode->RecAddr);
```

The value W is inserted into the root. This is often called *promoting* the key W. This causes the root to overflow with five keys. Again, the node is split, resulting in a node with keys D, M, and P, and one with T and W.

There is no higher level of the tree, so a new root node is created, and the keys P and W are inserted into it. This is accomplished by the following code:

```
nt newAddr = BTreeFile.Append(Root);//put previous root into file
  insert 2 keys in new root node
oot.Keys[0]=thisNode->LargestKey();
oot.RecAddrs[0]=newAddr;
oot.Keys[1]=newNode->LargestKey();
oot.RecAddrs[1]=newNode->RecAddr;
oot.NumKeys=2;
eight++;
```

It begins by appending the old root node into the B-tree file. The very first index record in the file is always the root node, so the old root node, which is no longer the root, must be put somewhere else. Then the insertions are performed. Finally the height of the tree is increased by one.

`Insert` uses a number of support functions. The most obvious one is method `BTreeNode::Split` which distributes the keys between the original page and the new page. Figure 9.19 contains an implementation of this method. Some additional error checking is included in the full implementation in Appendix I. Method `Split` simply removes some of the keys and references from the overfull node and puts them into the new node.

The full implementation of `BTree::Insert` in Appendix I includes code to handle the special case of the insertion of a new largest key in the tree. This is the only case where an insertion adds a new largest key to a node. This can be verified by looking at method `FindLeaf`, which is used to determine the leaf node to be used in insertion. `FindLeaf` always chooses a node whose largest key is greater than or equal to the search key. Hence, the only case where `FindLeaf` returns a leaf node in which the search key is greater than the largest key is where that leaf node is the rightmost node in the tree and the search key is greater than any key in the tree. In this case, the insertion of the new key

```
plate <class keyType>
BTreeNode<keyType>::Split (BTreeNode<keyType> * newNode)

   find the first Key to be moved into the new node
int midpt = (NumKeys+1)/2;
int numNewKeys = NumKeys - midpt;
  move the keys and recaddrs from this to newNode
for (int i = midpt; i< NumKeys; i++)
{
   newNode->Keys[i-midpt] = Keys[i];
   newNode->RecAddrs[i-midpt] = RecAddrs[i];
}
   set number of keys in the two Nodes
newNode->NumKeys = numNewKeys;
NumKeys = midpt;
return 1;
```

re 9.19 Method `Split` of class `BTreeNode`.

requires changing the largest key in the rightmost node in every level of the index. The code to handle this special case is included in `BTree::Insert`.

9.8.3 Create, Open, and Close

We need methods to create, open, and close B-tree files. Our object-oriented design and the use of objects from previous classes have made these methods quite simple, as you can see in file `btree.tc` of Appendix I. Method `Create` has to write the empty root node into the file `BTreeFile` so that its first record is reserved for that root node. Method `Open` has to open `BTreeFile` and load the root node into memory from the first record in the file. Method `Close` simply stores the root node into `BTreeFile` and closes it.

9.8.4 Testing the B-Tree

The file `tstbtree.cpp` in Appendix I has the full code of a program to test creation and insertion of a B-tree. Figure 9.20 contains most of the code. As you can see, this program uses a single character key (`class`

```
const char * keys="CSDTAMPIBWNGURKEHOLJYQZFXV";
const int BTreeSize = 4;
main (int argc, char * argv)
{
   int result, i;
   BTree <char> bt (BTreeSize);
   result = bt.Create ("testbt.dat",ios::in|ios::out);
   for (i = 0; i<26; i++)
   {
      cout<<"Inserting "<<keys[i]<<endl;
      result = bt.Insert(keys[i],i);
      bt.Print(cout); // print after each insert
   }
   return 1;
}
```

Figure 9.20 Program tstbtree.cpp.

`BTree<char>`) and inserts the alphabet in the same order as in Fig. 9.14 and 9.15. The tree that is created is identical in form to those pictured in the figures.

9.9 B-Tree Nomenclature

Before moving on to discuss B-tree performance and variations on the basic B-tree algorithms, we need to formalize our B-tree terminology. Providing careful definitions of terms such as *order* and *leaf* enables us to state precisely the properties that must be present for a data structure to qualify as a B-tree.

This definition of B-tree properties, in turn, informs our discussion of matters such as the procedure for deleting keys from a B-tree.

Unfortunately, the literature on B-trees is not uniform in its use of terms. Reading that literature and keeping up with new developments therefore require some flexibility and some background: the reader needs to be aware of the different uses of some of the fundamental terms.

For example, Bayer and McCreight (1972), Comer (1979), and a few others refer to the *order* of a B-tree as the *minimum* number of *keys* that can be in a page of a tree. So, our initial sample B-tree (Fig. 9.14), which

can hold a *maximum* of four keys per page, has an *order* of two, using Bayer and McCreight's terminology. The problem with this definition of order is that it becomes clumsy when you try to account for pages that hold an *odd*, maximum number of keys. For example, consider the following question: Within the Bayer and McCreight framework, is the page of an order three B-tree full when it contains six keys or when it contains seven keys?

Knuth (1998) and others have addressed the odd/even confusion by defining the *order* of a B-tree to be the *maximum* number of *descendants* that a page can have. This is the definition of *order* that we use in this text. Note that this definition differs from Bayer and McCreight's in two ways: it references a *maximum*, not a *minimum*, and it counts *descendants* rather than *keys*.

When you split the page of a B-tree, the descendants are divided as evenly as possible between the new page and the old page. Consequently, every page except the root and the leaves has at *least* $m/2$ descendants. Expressed in terms of a ceiling function, we can say that the minimum number of descendants is $\lceil m/2 \rceil$.

Another term that is used differently by different authors is *leaf*. Bayer and McCreight refer to the lowest level of keys in a B-tree as the leaf level. This is consistent with the nomenclature we have used in this text. Other authors, including Knuth, consider the leaves of a B-tree to be one level *below* the lowest level of keys. In other words, they consider the leaves to be the actual data records that might be pointed to by the lowest level of keys in the tree. We do *not* use this definition; instead we stick with the notion of leaf as the lowest level of B-tree nodes.

Finally, many authors call our definition of B-tree a *B^+ tree*. The term *B-tree* is often used for a version of the B-tree that has data record references in all of the nodes, instead of only in the leaf nodes. A major difference is that our version has the full index in the leaf nodes and uses the interior nodes as higher level indexes. This results in a duplication of keys, since each key in an interior node is duplicated at each lower level. The other version eliminates this duplication of key values, and instead includes data record references in interior nodes. While it seems that this will save space and reduce search times, in fact it often does neither. The major deficiency of this version is that the size of the interior nodes is much larger for the same order B-tree. Another way to look at the difference is that for the same amount of space in the interior nodes, by eliminating the data references, we could significantly increase the order of the tree, resulting in shallower trees. Of course, the shallower the tree, the shorter the search.

In this book, we use the term *B+ tree* to refer to a somewhat more complex situation in which the data file is not entry sequenced but is organized into a linked list of sorted blocks of records. The data file is organized in much the same way as the leaf nodes of a B-tree. The great advantage of the B+ tree organization is that both indexed access and sequential access are optimized. This technique is explained in detail in the next chapter.

You may have recognized that the largest key in each interior B-tree node is not needed in the searching. That is, in method `FindLeaf`, whenever the search key is bigger than any key in the node, the search proceeds to the rightmost child. It is possible and common to implement B-trees with one less key than reference in each interior node. However, the insertion method is made more complicated by this optimization, so it has been omitted in the B-tree classes and is included as a programming exercise.

10 Formal Definition of B-Tree Properties

Given these definitions of order and leaf, we can formulate a precise statement of the properties of a B-tree of order *m:*

- Every page has a maximum of m descendants.
- Every page, except for the root and the leaves, has at least $\lceil m/2 \rceil$ descendants.
- The root has at least two descendants (unless it is a leaf).
- All the leaves appear on the same level.
- The leaf level forms a complete, ordered index of the associated data file.

11 Worst-Case Search Depth

It is important to have a quantitative understanding of the relationship between the page size of a B-tree, the number of keys to be stored in the tree, and the number of levels that the tree can extend. For example, you might know that you need to store 1 000 000 keys and that, given the nature of your storage hardware and the size of your keys, it is reasonable

to consider using a B-tree of order 512 (maximum of 511 keys per page). Given these two facts, you need to be able to answer the question: In the worst case, what will be the maximum number of disk accesses required to locate a key in the tree? This is the same as asking how deep the tree will be.

We can answer this question by noting that every key appears in the leaf level. Hence, we need to calculate the maximum height of a tree with 1 000 000 keys in the leaves.

Next we need to observe that we can use the formal definition of B-tree properties to calculate the *minimum* number of descendants that can extend from any level of a B-tree of some given order. This is of interest because we are interested in the *worst-case* depth of the tree. The worst case occurs when every page of the tree has only the minimum number of descendants. In such a case the keys are spread over a *maximal height* for the tree and a *minimal breadth.*

For a B-tree of order m, the minimum number of descendants from the root page is 2, so the second level of the tree contains only 2 pages. Each of these pages, in turn, has at least $\lceil m/2 \rceil$ descendants. The third level, then, contains

$$2 \times \lceil m/2 \rceil$$

pages. Since each of these pages, once again, has a minimum of $\lceil m/2 \rceil$ descendants, the general pattern of the relation between depth and the minimum number of descendants takes the following form:

Level	Minimum number of descendants
1 (root)	2
2	$2 \times \lceil m/2 \rceil$
3	$2 \times \lceil m/2 \rceil \times \lceil m/2 \rceil$ or $2 \times \lceil m/2 \rceil^2$
4	$2 \times \lceil m/2 \rceil^3$
...	...
d	$2 \times \lceil m/2 \rceil^{d-1}$

So, in general, for any level d of a B-tree, the *minimum* number of descendants extending from that level is

$$2 \times \lceil m/2 \rceil^{d-1}$$

For a tree with N keys in its leaves, we can express the relationship between keys and the minimum height d as

$$N \geq 2 \times \lceil m/2 \rceil^{d-1}$$

Solving for d, we arrive at the following expression:

$$d \leq 1 + \log_{\lceil m/2 \rceil}(N/2).$$

This expression gives us an *upper bound* for the depth of a B-tree with N keys. Let's find the upper bound for the hypothetical tree that we describe at the start of this section: a tree of order 512 that contains 1 000 000 keys. Substituting these specific numbers into the expression, we find that

$$d \leq 1 + \log_{256} 500\,000$$

or

$$d \leq 3.37$$

So we can say that given 1 000 000 keys, a B-tree of order 512 has a depth of no more than three levels.

9.12 Deletion, Merging, and Redistribution

Indexing 1 000 000 keys in no more than three levels of a tree is precisely the kind of performance we are looking for. As we have just seen, this performance is predicated on the B-tree properties we described earlier. In particular, the ability to guarantee that B-trees are broad and shallow rather than narrow and deep is coupled with the rules that state the following:

- Every page except for the root and the leaves has at least $\lceil m/2 \rceil$ descendants.
- A page contains at least $\lceil m/2 \rceil$ keys and no more than m keys.

We have already seen that the process of page splitting guarantees that these properties are maintained when new keys are inserted into the tree. We need to develop some kind of equally reliable guarantee that these properties are maintained when keys are *deleted* from the tree.

Working through some simple deletion situations by hand helps us demonstrate that the deletion of a key can result in several different situations. We start with the B-tree of Fig. 9.15(c) that contains all the letters of the alphabet. Consider what happens when we try to delete some of its keys.

The simplest situation is illustrated in the result of deleting key C in Fig. 9.21(a). Deleting the key from the first leaf node does not cause an

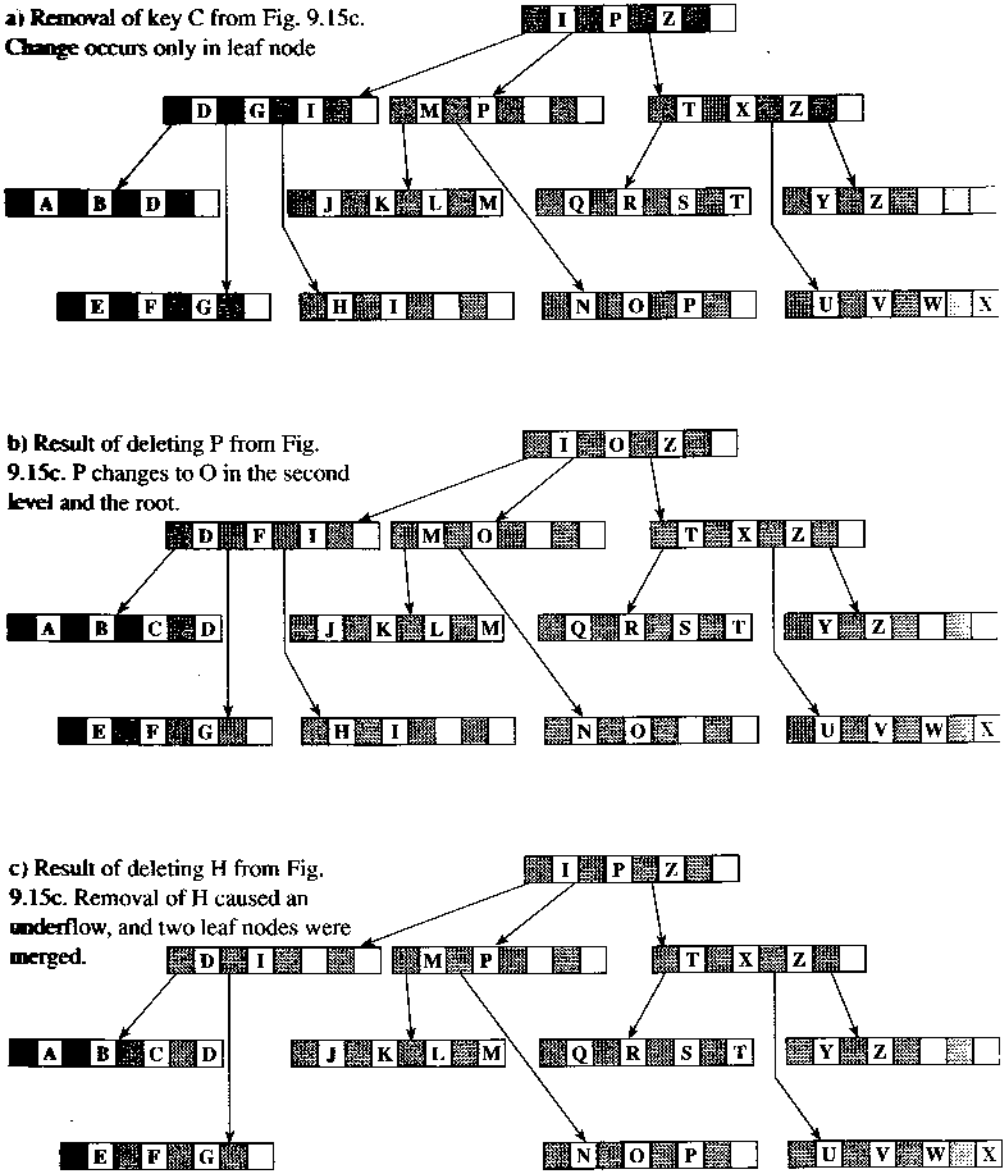


Figure 9.21 Three situations that can occur during deletions.

underflow in the node and does not change its largest value. Consequently, deletion involves nothing more than removing the key from the node.

Deleting the P in Fig. 9.21(b) is more complicated. Removal of P from the second leaf node does not cause underflow, but it does change the largest key in the node. Hence, the second-level node must be modified to reflect this change. The key to the second leaf node becomes O, and the second-level node must be modified so that it contains O instead of P. Since P was the largest key in the second node in the second level, the root node must also have key P replaced by O.

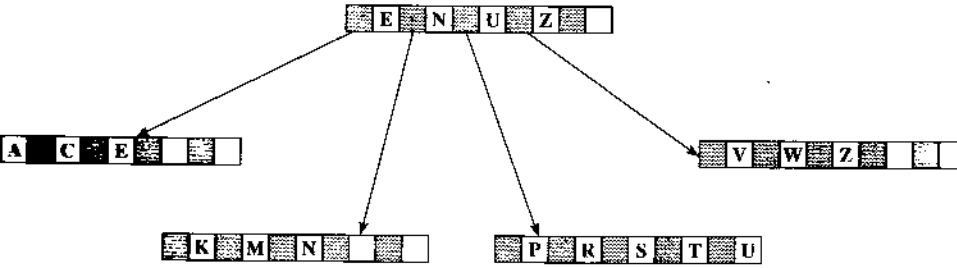
Deleting the H in Fig 9.21(c) causes an underflow in the third leaf node. After H is deleted, the last remaining key in the node, I, is inserted into the neighbor node, and the third leaf node is deleted. Since the second leaf node has only three keys, there is room for the key I in that node. This illustrates a more general *merge* operation. After the merge, the second-level node is modified to reflect the current status of the leaf nodes.

Merging and other modifications can propagate to the root of the B-tree. If the root ends up with only one key and one child, it can be eliminated. Its sole child node becomes the new root of the tree and the tree gets shorter by one level.

The rules for deleting a key *k* from a node *n* in a B-tree are as follows:

1. If *n* has more than the minimum number of keys and the *k* is not the largest in *n*, simply delete *k* from *n*.
2. If n has more than the minimum number of keys and the *k* is the largest in *n*, delete *k* and modify the higher level indexes to reflect the new largest key in *n*.
3. If *n* has exactly the minimum number of keys and one of the siblings of *n* has few enough keys, merge *n* with its sibling and delete a key from the parent node.
4. If *n* has exactly the minimum number of keys and one of the siblings of *n* has extra keys, redistribute by moving some keys from a sibling to *n*, and modify the higher level indexes to reflect the new largest keys in the affected nodes.

Rules 3 and 4 include references to "few enough keys" to allow merging and "extra keys" to allow redistribution. These are not exclusive rules, and the implementation of delete is allowed to choose which rule to use when they are both applicable. Look at the example of an order five tree in Fig. 9.22, and consider deleting keys C, M, and W. Since three is the minimum number of keys, deleting any of these keys requires some adjustment of the leaf nodes. In the case of deleting *C*, the only sibling node has three



gure 9.22 Example of order five B-tree. Consider delete of keys C, M, and W.

keys. After deleting C, there are five keys in the two sibling nodes, so a merge is allowed. No redistribution is possible because the sibling node has the minimum number of keys. In the case of deleting W, the only sibling has five keys, so one or two of the keys can be moved to the underfull node. No merge is possible here, since there are seven keys remaining in the two sibling nodes—too many for a single node. In the case of deleting M, there are two options: merge with the left sibling or redistribute keys in the right sibling.

9.12.1 Redistribution

Unlike merge, which is a kind of reverse split, redistribution is a new idea. Our insertion algorithm does not require operations analogous to redistribution.

Redistribution differs from both splitting and merging in that it never causes the collection of nodes in the tree to change. It is guaranteed to have strictly local effects. Note that the term *sibling* implies that the pages have the same parent page. If there are two nodes at the leaf level that are logically adjacent but do not have the same parent—for example, HI and JKLM in the tree of Fig. 9.22(a)—these nodes are not siblings. Redistribution algorithms are generally written so they do not consider moving keys between nodes that are not siblings, even when they are logically adjacent. Can you see the reasoning behind this restriction?

Another difference between redistribution on the one hand and merging and splitting on the other is that there is no necessary, fixed prescription for how the keys should be rearranged. A single deletion in a properly formed B-tree cannot cause an underflow of more than one key. Therefore, redistribution can restore the B-tree properties by moving only one key from a sibling into the page that has underflowed, even if the

distribution of the keys between the pages is very uneven. Suppose, for example, that we are managing a B-tree of order 101. The minimum number of keys that can be in a page is 50; the maximum is 100. Suppose we have one page that contains the minimum and a sibling that contains the maximum. If a key is deleted from the page containing 50 keys, an underflow condition occurs. We can correct the condition through redistribution by moving one key, 50 keys, or any number of keys between 1 and 50. The usual strategy is to divide the keys as evenly as possible between the pages. In this instance that means moving 25 keys.

.13 Redistribution During Insertion: A Way to Improve Storage Utilization

As you may recall, B-tree insertion does not require an operation analogous to redistribution; splitting is able to account for all instances of overflow. This does not mean, however, that it is not *desirable* to use redistribution during insertion as an option, particularly since a set of B-tree maintenance algorithms must already include a redistribution procedure to support deletion. Given that a redistribution procedure is already present, what advantage might we gain by using it as an alternative to node splitting?

Redistribution during insertion is a way of avoiding, or at least postponing, the creation of new pages. Rather than splitting a full page and creating two approximately half-full pages, redistribution lets us place some of the overflowing keys into another page. The use of redistribution in place of splitting should therefore tend to make a B-tree more efficient in its utilization of space.

It is possible to quantify this efficiency of space usage by viewing the amount of space used to store information as a percentage of the total amount of space required to hold the B-tree. After a node splits, each of the two resulting pages is about half full. So, in the worst case, space utilization in a B-tree using two-way splitting is around 50 percent. Of course, the actual degree of space utilization is better than this worst-case figure. Yao (1978) has shown that, for large trees of relatively large order, space utilization approaches a theoretical average of about 69 percent if insertion is handled through two-way splitting.

The idea of using redistribution as an alternative to splitting when possible, splitting a page only when both of its siblings are full, is

introduced in Bayer and McCreight's original paper (1972). The paper includes some experimental results that show that two-way splitting results in a space utilization of 67 percent for a tree of order 121 after five thousand random insertions. When the experiment was repeated, using redistribution when possible, space utilization increased to over 86 percent. Subsequent empirical testing by students at Oklahoma State University using B-trees of order 49 and 303 also resulted in space utilization exceeding 85 percent when redistribution was used. These findings and others suggest that any serious application of B-trees to even moderately large files should implement insertion procedures that handle overflow through redistribution when possible.

.14 B* Trees

In his review and amplification of work on B-trees in 1973, Knuth (1998) extends the notion of redistribution during insertion to include new rules for splitting. He calls the resulting variation on the fundamental B-tree form a *B** tree.

Consider a system in which we are postponing splitting through redistribution, as outlined in the preceding section. If we are considering any page other than the root, we know that when it is finally time to split, the page has at least one sibling that is also full. This opens up the possibility of a two-to-three split rather than the usual one-to-two or two-way split.

The important aspect of this two-to-three split is that it results in pages that are each about two-thirds full rather than just half full. This makes it possible to define a new kind of B-tree, called a B* tree, which has the following properties:

1. Every page has a maximum of m descendants.
2. Every page except for the root has at least $\lceil (2m - 1)/3 \rceil$ descendants.
3. The root has at least two descendants (unless it is a leaf).
4. All the leaves appear on the same level.

The critical changes between this set of properties and the set we define for a conventional B-tree are in rule 2: a B* tree has pages that contain a minimum $\lceil (2m - 1)/3 \rceil$ keys. This new property, of course, affects procedures for deletion and redistribution.

To implement B* tree procedures, one must also deal with the question of splitting the root, which, by definition, never has a sibling. If there

is no sibling, no two-to-three split is possible. Knuth suggests allowing the root to grow to a size larger than the other pages so, when it does split, it can produce two pages that are each about two-thirds full. This has the advantage of ensuring that all pages below the root level adhere to B* tree characteristics. However, it has the disadvantage of requiring that the procedures be able to handle a page that is larger than all the others. Another solution is to handle the splitting of the root as a conventional one-to-two split. This second solution avoids any special page-handling logic. On the other hand, it complicates deletion, redistribution, and other procedures that must be sensitive to the minimum number of keys allowed in a page. Such procedures would have to be able to recognize that pages descending from the root might legally be only half full.

.15 Buffering of Pages: Virtual B-Trees

We have seen that the B-tree can be a very efficient, flexible storage structure that maintains its balanced properties after repeated deletions and insertions and that provides access to any key with just a few disk accesses. However, focusing on just the structural aspects, as we have so far, can cause us inadvertently to overlook ways of using this structure to full advantage. For example, the fact that a B-tree has a depth of three levels does not at all mean that we need to do three disk accesses to retrieve keys from pages at the leaf level. We can do much better than that.

Obtaining better performance from B-trees involves looking in a precise way at our original problem. We needed to find a way to make efficient use of indexes that are too large to be held *entirely* in memory. Up to this point we have approached this problem in an all-or-nothing way: an index has been held entirely in memory, organized as a list or binary tree, or accessed entirely on secondary store, using a B-tree structure. But, stating that we cannot hold *all* of an index in memory does not imply that we cannot hold *some* of it there. In fact, our implementation of class `BTree` is already keeping the root in memory at all times and keeping a full branch in memory during insertion and deletion.

For example, assume that we have an index containing 1 megabyte of records and that we cannot reasonably use more than 256 kilobytes of memory for index storage at any given time. Given a page size of 4 kilobytes, holding around 64 keys per page, our B-tree can be contained in three levels. We can reach any one of our keys in no more than two disk

accesses. That is certainly acceptable, but why should we settle for this kind of performance? Why not try to find a way to bring the average number of disk accesses per search down to one disk access or less?

If we're thinking of the problem strictly in terms of physical storage structures, retrieval averaging one disk access or less sounds impossible. But remember, our objective was to find a way to manage our megabyte of index within 256 kilobytes of memory, not within the 4 kilobytes required to hold a single page of our tree.

The simple, keep-the-root strategy we have been using suggests an important, more general approach: rather than just holding the root page in memory, we can create a *page buffer* to hold some number of B-tree pages, perhaps five, ten, or more. As we read pages in from the disk in response to user requests, we fill up the buffer. Then, when a page is requested, we access it from memory if we can, thereby avoiding a disk access. If the page is not in memory, then we read it into the buffer from secondary storage, replacing one of the pages that was previously there. A B-tree that uses a memory buffer in this way is sometimes referred to as a *virtual B-tree.*

For our implementation, we can use the `Nodes` member and the `Fetch` and `Store` methods to manage this page buffer. `Fetch` and `Store` can keep track of which nodes are in memory and avoid the disk read or write whenever possible. This modification is included as an exercise.

9.15.1 LRU Replacement

Clearly, such a buffering scheme works only if we are more likely to request a page that is in the buffer than one that is not. The process of accessing the disk to bring in a page that is *not* already in the buffer is called a *page fault.* There are two causes of page faults:

1. We have never used the page.
2. It was once in the buffer but has since been replaced with a new page.

The first cause of page faults is unavoidable: if we have not yet read in and used a page, there is no way it can already be in the buffer. But the second cause is one we can try to minimize through buffer management. The critical management decision arises when we need to read a new page into a buffer that is already full: which page do we decide to replace?

One common approach is to replace the page that was least recently used; this is called *LRU* replacement. Note that this is different from

replacing the page that was *read into* the buffer least recently. Instead, the LRU method keeps track of the *requests* for pages. The page to be replaced is the one that has gone the longest time without a request for use.

Some research by Webster (1980) shows the effect of increasing the number of pages that can be held in the buffer area under an LRU replacement strategy. Table 9.1 summarizes a small but representative portion of Webster's results. It lists the average number of disk accesses per search given different numbers of page buffers. These results are obtained using a simple LRU replacement strategy without accounting for page height. Keeping less than 15 percent of the tree in memory (20 pages out of the total 140) reduces the average number of accesses per search to less than one.

Note that the decision to use LRU replacement is based on the assumption that we are more likely to need a page that we have used recently than we are to need a page that we have never used or one that we used some time ago. If this assumption is not valid, then there is absolutely no reason to retain preferentially pages that were used recently. The term for this kind of assumption is *temporal locality*. We are assuming that there is a kind of *clustering* of the use of certain pages over time. The hierarchical nature of a B-tree makes this kind of assumption reasonable.

For example, during redistribution after overflow or underflow, we access a page and then access its sibling. Because B-trees are hierarchical, accessing a set of sibling pages involves repeated access to the parent page in rapid succession. This is an instance of temporal locality; it is easy to see how it is related to the tree's hierarchy.

9.15.2 Replacement Based on Page Height

There is another, more direct way to use the hierarchical nature of the B-tree to guide decisions about page replacement in the buffers. Our simple, keep-the-root strategy exemplifies this alternative: always retain the pages that occur at the highest levels of the tree. Given a larger amount of buffer

Table 9.1 Effect of using more buffers with a simple LRU replacement strategy.

Buffer Count	1	5	10	20
Average Accesses per Search	3.00	1.71	1.42	0.97
Number of keys = 2400				
Total pages = 140				
Tree height = 3 levels				

space, it might be possible to retain not only the root, but also all of the pages at the second level of a tree.

Let's explore this notion by returning to a previous example in which we have access to 256 kilobytes of memory and a 1-megabyte index. Since our page size is 4 kilobytes, we could build a buffer area that holds 64 pages within the memory area. Assume that our 1 megabyte worth of index requires around 1.2 megabytes of storage on disk (storage utilization = 83 percent). Given the 4-kilobyte page size, this 1.2 megabytes requires slightly more than 300 pages. We assume that, on the average, each of our pages has around 30 descendants. It follows that our three-level tree has, of course, a single page at the root level, followed by 9 or 10 pages at the second level, with all the remaining pages at the leaf level. Using a page replacement strategy that always retains the higher-level pages, it is clear that our 64-page buffer eventually contains the root page and all the pages at the second level. The approximately 50 remaining buffer slots are used to hold leaf-level pages. Decisions about which of these pages to replace can be handled through an LRU strategy. It is easy to see how, given a sizable buffer, it is possible to bring the average number of disk accesses per search down to a number that is less than one.

Webster's research (1980) also investigates the effect of taking page height into account, giving preference to pages that are higher in the tree when it comes time to decide which pages to keep in the buffers. Augmenting the LRU strategy with a weighting factor that accounts for page height reduces the average number of accesses, given a 10-page buffer, from 1.42 accesses per search down to 1.12 accesses per search.

9.15.3 Importance of Virtual B-Trees

It is difficult to overemphasize the importance of including a page buffering scheme with any implementation of a B-tree index structure. Because the B-tree structure is so interesting and powerful, it is easy to fall into the trap of thinking that the B-tree organization is itself a sufficient solution to the problem of accessing large indexes that must be maintained on secondary storage. As we have emphasized, to fall into that trap is to lose sight of the original problem: to find a way to *reduce* the amount of memory required to handle large indexes. We did not, however, need to reduce the amount of memory to the amount required for a single index page. It is usually possible to find enough memory to hold a number of pages. Doing so can dramatically increase system performance.

9.16 Variable-Length Records and Keys

In many applications the information associated with a key varies in length. Secondary indexes that reference inverted lists are an excellent example of this. One way to handle this variability is to place the associated information in a separate, variable-length record file; the B-tree would contain a reference to the information in this other file. Another approach is to allow a variable number of keys and records in a B-tree page.

Up to this point we have regarded B-trees as being of some order m. Each page has a fixed maximum and minimum number of keys that it can legally hold. The notion of a variable-length record and, therefore, a variable number of keys per page is a significant departure from the point of view we have developed so far. A B-tree with a variable number of keys per page clearly has no single, fixed order.

The variability in length can also extend to the keys as well as to entire records. For example, in a file in which people's names are the keys, we might choose to use only as much space as required for a name rather than allocate a fixed-size field for each key. As we saw in earlier chapters, implementing a structure with variable-length fields can allow us to put many more names in a given amount of space since it eliminates internal fragmentation. If we can put more keys in a page, then we have a larger number of descendants from a page and very probably a tree with fewer levels.

Accommodating this variability in length means using a different kind of page structure. We look at page structures appropriate for use with variable-length keys in detail in the next chapter. We also need a different criterion for deciding when a page is full and when it is in an underflow condition. Rather than use a maximum and minimum number of keys per page, we need to use a maximum and minimum number of bytes.

Once the fundamental mechanisms for handling variable-length keys or records are in place, interesting new possibilities emerge. For example, we might consider the notion of biasing the splitting and redistribution methods so that the shortest variable-length keys are promoted upward in preference to longer keys. The idea is that we want to have pages with the largest numbers of descendants up high in the tree, rather than at the leaf level. Branching out as broadly as possible as high as possible in the tree tends to reduce the overall height of the tree. McCreight (1977) explores this notion in the article, "Pagination of B* Trees with Variable-Length Records."

The principal point we want to make with these examples of variations on B-tree structures is that this chapter introduces only the most basic forms of this very useful, flexible file structure. Implementations of B-trees do not slavishly follow the textbook form of B-trees. Instead, they use many of the other organizational techniques we study in this book, such as variable-length record structures in combination with the fundamental B-tree organization to make new, special-purpose file structures uniquely suited to the problems at hand.

SUMMARY

We begin this chapter by picking up the problem we left unsolved at the end of Chapter 7: simple, linear indexes work well if they are held in memory, but they are expensive to maintain and search if they are so big that they must be held on secondary storage. The expense of using secondary storage is most evident in two areas:

- Sorting of the index; and
- Searching, since even binary searching requires more than two or three disk accesses.

We first address the question of structuring an index so it can be kept in order without sorting. We use tree structures to do this, discovering that we need a *balanced* tree to ensure that the tree does not become overly deep after repeated random insertions. We see that AVL trees provide a way of balancing a binary tree with only a small amount of overhead.

Next we turn to the problem of reducing the number of disk accesses required to search a tree. The solution to this problem involves dividing the tree into pages so a substantial portion of the tree can be retrieved with a single disk access. Paged indexes let us search through very large numbers of keys with only a few disk accesses.

Unfortunately, we find that it is difficult to combine the idea of *paging* of tree structures with the *balancing* of these trees by AVL methods. The most obvious evidence of this difficulty is associated with the problem of selecting the members of the root page of a tree or subtree when the tree is built in the conventional top-down manner. This sets the stage for introducing Bayer and McCreight's work on B-trees, which solves the paging and balancing dilemma by starting from the leaf level, promoting keys upward as the tree grows.

Our discussion of B-trees begins by emphasizing the multilevel index approach. We include a full implementation of insertion and searching and examples of searching, insertion, overflow detection, and splitting to show how B-trees grow while maintaining balance in a paged structure. Next we formalize our description of B-trees. This formal definition permits us to develop a formula for estimating worst-case B-tree depth. The formal description also motivates our work on developing deletion procedures that maintain the B-tree properties when keys are removed from a tree.

Once the fundamental structure and procedures for B-trees are in place, we begin refining and improving on these ideas. The first set of improvements involves increasing the storage utilization within B-trees. Of course, increasing storage utilization can also result in a decrease in the height of the tree and therefore in improvements in performance. We sometimes find that by redistributing keys during insertion rather than splitting pages, we can improve storage utilization in B-trees so it averages around 85 percent. Carrying our search for increased storage efficiency even further, we find that we can combine redistribution during insertion with a different kind of splitting to ensure that the pages are about two-thirds full rather than only half full after the split. Trees using this combination of redistribution and two-to-three splitting are called B^* trees.

Next we turn to the matter of buffering pages, creating a *virtual B-tree*. We note that the use of memory is not an all-or-nothing choice: indexes that are too large to fit into memory do not have to be accessed *entirely* from secondary storage. If we hold pages that are likely to be reused in memory, then we can save the expense of reading these pages in from the disk again. We develop two methods of guessing which pages are to be reused. One method uses the height of the page in the tree to decide which pages to keep. Keeping the root has the highest priority, the root's descendants have the next priority, and so on. The second method for selecting pages to keep in memory is based on recentness of use: we always replace the least recently used (LRU) page and retain the pages used most recently. We see that it is possible to combine these methods and that doing so can result in the ability to find keys while using an average of less than one disk access per search.

We close the chapter with a brief look at the use of variable-length records within the pages of a B-tree, noting that significant savings in space and consequent reduction in the height of the tree can result from the use of variable-length records. The modification of the basic textbook B-tree definition to include the use of variable-length records is just one

example of the many variations on B-trees that are used in real-world implementations.

KEY TERMS

AVL tree. A height-balanced (HB(1)) binary tree in which insertions and deletions can be performed with minimal accesses to local nodes. AVL trees are interesting because they keep branches from getting overly long after many random insertions.

B-tree of order *m*. A multilevel index tree with these properties:

0 Every node has a maximum of m descendants.
1 Every node except the root has at least $\lceil m/2 \rceil$ descendants.
2 The root has at least two descendants (unless it is a leaf).
3 All of the leaves appear on the same level.

B-trees are built upward from the leaf level, so creation of new pages always starts at the leaf level.

The power of B-trees lies in the facts that they are balanced (no overly long branches); they are shallow (requiring few seeks); they accommodate random deletions and insertions at a relatively low cost while remaining in balance; and they guarantee at least 50 percent storage utilization.

B* tree. A special B-tree in which each node is at least two-thirds full. B* trees generally provide better storage utilization than B-trees.

Height-balanced tree. A tree structure with a special property: for each node there is a limit to the amount of difference that is allowed among the heights of any of the node's subtrees. An *HB(k)* tree allows subtrees to be k levels out of balance. (See *AVL tree.*)

Leaf of a B-tree. A page at the lowest level in a B-tree. All leaves in a B-tree occur at the same level.

Merging. When a B-tree node underflows (becomes less than 50 percent full), it sometimes becomes necessary to combine the node with an adjacent node, thus decreasing the total number of nodes in the tree. Since merging involves a change in the number of nodes in the tree, its effects can require reorganization at many levels of the tree.

Order of a B-tree. The maximum number of descendants that a node in the B-tree can have.

Paged index. An index that is divided into blocks, or pages, each of which can hold many keys. The use of paged indexes allows us to search through very large numbers of keys with only a few disk accesses.

Redistribution. When a B-tree node underflows (becomes less than 50 percent full), it may be possible to move keys into the node from an adjacent node with the same parent. This helps ensure that the 50 percent-full property is maintained. When keys are redistributed, it becomes necessary to alter the contents of the parent as well. Redistribution, as opposed to *merging*, does not involve creation or deletion of nodes—its effects are entirely local. Often redistribution can also be used as an alternative to splitting.

Splitting. Creation of two nodes out of one when the original node becomes overfull. Splitting results in the need to promote a key to a higher-level node to provide an index separating the two new nodes.

Virtual B-tree. A B-tree index in which several pages are kept in memory in anticipation of the possibility that one or more of them will be needed by a later access. Many different strategies can be applied to replacing pages in memory when virtual B-trees are used, including the least-recently-used strategy and height-weighted strategies.

FURTHER READINGS

Currently available textbooks on file and data structures contain surprisingly brief discussions on B-trees. These discussions do not, in general, add substantially to the information presented in this chapter and the following chapter. Consequently, readers interested in more information about B-trees must turn to the articles that have appeared in journals over the past 15 years.

The article that introduced B-trees to the world is Bayer and McCreight's "Organization and Maintenance of Large Ordered Indexes" (1972). It describes the theoretical properties of B-trees and includes empirical results concerning, among other things, the effect of using redistribution in addition to splitting during insertion. Readers should be aware that the notation and terminology used in this article differ from those used in this text in a number of important respects.

Comer's (1979) survey article, "The Ubiquitous B-tree," provides an excellent overview of some important variations on the basic B-tree form. Knuth's (1998) discussion of B-trees, although brief, is an important

resource in part because many of the variant forms such as B* trees were first collected together in Knuth's discussion. McCreight (1977) looks specifically at operations on trees that use variable-length records and that are therefore of variable order. Although this article speaks specifically about B* trees, the consideration of variable-length records can be applied to many other B-tree forms. In "Time and Space Optimality on B-trees," Rosenberg and Snyder (1981) analyze the effects of initializing B-trees with the minimum number of nodes. In "Analysis of Design Alternatives for Virtual Memory Indexes," Murayama and Smith (1977) look at three factors that affect the cost of retrieval: choice of search strategy, whether pages in the index are structured, and whether keys are compressed. Gray and Reuter (1993) provide an analysis of issues in B-tree implementation. Zoellick (1986) discusses the use of B-tree—like structures on optical discs.

Since B-trees in various forms have become a standard file organization for databases, a good deal of interesting material on applications of B-trees can be found in the database literature. Held and Stonebraker (1978), Snyder (1978), Kroenke (1998), and Elmasri and Navathe (1994) discuss the use of B-trees in database systems generally. Ullman (1986) covers the problem of dealing with applications in which several programs have access to the same database concurrently and identifies literature concerned with concurrent access to B-tree.

Uses of B-trees for secondary key access are covered in many of the previously cited references. There is also a growing literature on multidimensional dynamic indexes, including variants of the B-tree, *k-d* B-tree and R trees. *K-d* B-trees are described in papers by Ouskel and Scheuermann (1981) and Robinson (1981). R trees support multidimensional queries, so-called *range queries*, and were first described in Guttman (1984) and further extended in Sellis et al (1987), Beckmann et al (1990), and Kamel and Floutsos (1992). Shaffer (1997) and Standish (1995) include extensive coverage of a variety of tree structures. Other approaches to secondary indexing include the use of *tries* and *grid files*. Tries are covered in many texts on files and data structures, including Knuth (1998) and Loomis (1989). Grid files are covered thoroughly in Nievergelt et al. (1984).

An interesting early paper on the use of dynamic tree structures for processing files is "The Use of Tree Structures for Processing Files," by Sussenguth (1963). Wagner (1973) and Keehn and Lacy (1974) examine the index design considerations that led to the development of VSAM. VSAM uses an index structure very similar to a B-tree but appears to have

been developed independently of Bayer and McCreight's work. Readers interested in learning more about AVL trees should read Knuth (1998), who takes a more rigorous, mathematical look at AVL tree operations and properties.

EXERCISES

1. Balanced binary trees can be effective index structures for memory-based indexing, but they have several drawbacks when they become so large that part or all of them must be kept on secondary storage. The following questions should help bring these drawbacks into focus and thus reinforce the need for an alternative structure such as the B-tree.
   a. There are two major problems with using binary search to search a simple sorted index on secondary storage: the number of disk accesses is larger than we would like, and the time it takes to keep the index sorted is substantial. Which of the problems does a binary search tree alleviate?
   b. Why is it important to keep search trees balanced?
   c. In what way is an AVL tree better than a simple binary search tree?
   d. Suppose you have a file with 1 000 000 keys stored on disk in a completely full, balanced binary search tree. If the tree is not paged, what is the maximum number of accesses required to find a key? If the tree is paged in the manner illustrated in Fig. 9.12, but with each page able to hold 15 keys and to branch to 16 new pages, what is the maximum number of accesses required to find a key? If the page size is increased to hold 511 keys with branches to 512 nodes, how does the maximum number of accesses change?
   e. Consider the problem of balancing the three-key-per-page tree in Fig. 9.13 by rearranging the pages. Why is it difficult to create a tree-balancing algorithm that has only local effects? When the page size increases to a more likely size (such as 512 keys), why does it become difficult to guarantee that each of the pages contains at least some minimum number of keys?
   f. Explain the following statement: B-trees are built upward from the bottom, whereas binary trees are built downward from the top.
   g. Although B-trees are generally considered superior to binary search trees for external searching, binary trees are still commonly used for internal searching. Why is this so?

2. Show the B-trees of order four that result from loading the following sets of keys in order:
   a. C G J X
   b. C G J X N S U O A E B H I
   c. C G J X N S U O A E B H I F
   d. C G J X N S U O A E B H I F K L Q R T V U W Z
3. Given a B-tree of order 256,
   a. What is the maximum number of descendants from a page?
   b. What is the minimum number of descendants from a page (excluding the root and leaves)?
   c. What is the minimum number of descendants from the root?
   d. What is the maximum depth of the tree if it contains 100 000 keys?
4. Using a method similar to that used to derive the formula for worst-case depth, derive a formula for best-case, or minimum, depth for an order m B-tree with N keys. What is the minimum depth of the tree described in the preceding question?
5. Suppose you have a B-tree index for an unsorted file containing N data records, where each key has stored with it the RRN of the corresponding record. The depth of the B-tree is d. What are the maximum and minimum numbers of disk accesses required to
   a. Retrieve a record?
   b. Add a record?
   c. Delete a record?
   d. Retrieve all records from the file in sorted order?

   Assume that page buffering is *not* used. In each case, indicate how you arrived at your answer.
6. Show the trees that result after each of the keys N, P, Q, and Y is deleted from the B-tree of Figure 9.15(c).
7. A common belief about B-trees is that a B-tree cannot grow deeper unless it is 100 percent full. Discuss this.
8. Suppose you want to delete a key from a node in a B-tree. You look at the right sibling and find that redistribution does not work; merging would be necessary. You look to the left and see that redistribution is an option here. Do you choose to merge or redistribute?

9. What is the difference between a B* tree and a B-tree? What improvement does a B* tree offer over a B-tree, and what complications does it introduce? How does the minimum depth of an order m B* tree compare with that of an order m B-tree?
10. What is a virtual B-tree? How can it be possible to average fewer than one access per key when retrieving keys from a three-level virtual B-tree? Write a description for an LRU replacement scheme for a ten-page buffer used in implementing a virtual B-tree.
11. Discuss the trade-offs between storing the information indexed by the keys in a B-tree with the key and storing the information in a separate file.
12. We noted that, given variable-length keys, it is possible to optimize a tree by building in a bias toward promoting shorter keys. With fixed-order trees we promote the middle key. In a variable-order, variable-length key tree, what is the meaning of "middle key"? What are the trade-offs associated with building in a bias toward shorter keys in this selection of a key for promotion? Outline an implementation for this selection and promotion process.

PROGRAMMING EXERCISES

13. Implement the `Delete` method of class `BTree`.
14. Modify classes `BTreeNode` and `BTree` to have one more reference than key in each interior mode.
15. Write an interactive program that allows a user to find, insert, and delete keys from a B-tree.
16. Write a B-tree program that uses keys that are strings rather than single characters.
17. Write a program that builds a B-tree index for a data file in which records contain more information than just a key. Use the `Person`, `Recording`, `Ledger`, or `Transaction` files from previous chapters.
18. Implement B* trees by modifying class `BTree`.

PROGRAMMING PROJECT

This is the seventh part of the programming project. We add B-tree indexes to the data files created by the third part of the project in Chapter 4.

19. Use class `BTree` to create a B-tree index of a student record file with the student identifier as key. Write a driver program to create a B-tree file from an existing student record file.

20. Use class `BTree` to create a B-tree index of a course registration record file with the student identifier as key. Write a driver program to create a B-tree file from an existing course registration record file.

21. Write a program that opens a B-tree indexed student file and a B-tree indexed 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 10.