not part of the database, and therefore do not have their own logical addresses in the map table. Assuming that physical addresses use the minimum possible number of bytes for physical addresses (as calculated in Exercise 3.3.1), and logical addresses likewise use the minimum possible number of bytes for logical addresses, how many blocks of 4096 bytes does the map table for the disk occupy?

*! **Exercise 3.3.7:** Suppose that we have 4096-byte blocks in which we store records of 100 bytes. The block header consists of an offset table, as in Fig. 3.8, using 2-byte pointers to records within the block. On an average day, two records per block are inserted, and one record is deleted. A deleted record must have its pointer replaced by a "tombstone," because there may be dangling pointers to it. For specificity, assume the deletion on any day always occurs before the insertions. If the block is initially empty, after how many days will there be no room to insert any more records?

! **Exercise 3.3.8:** Repeat Exercise 3.3.7 on the assumption that each day there is one deletion and 1.1 insertions on the average.

Exercise 3.3.9: Repeat Exercise 3.3.7 on the assumption that instead of deleting records, they are moved to another block and must be given an 8-byte forwarding address in their offset-table entry. Assume either:

! a) All offset-table entries are given the maximum number of bytes needed in an entry.

!! b) Offset-table entries are allowed to vary in length in such a way that all entries can be found and interpreted properly.

* **Exercise 3.3.10:** Suppose that if we swizzle all pointers automatically, we can perform the swizzling in half the time it would take to swizzle each one separately. If the probability that a pointer in main memory will be followed at least once is p, for what values of p is it more efficient to swizzle automatically than on demand?

! **Exercise 3.3.11:** Generalize Exercise 3.3.10 to include the possibility that we never swizzle pointers. Suppose that the important actions take the following

If we need to replace records by tombstones, it would be wise to have at the very beginning of the record header a bit that serves as a tombstone; i.e., it is 0 if the record is *not* deleted, while 1 means that the record has been deleted. Then, only this bit must remain where the record used to begin, and subsequent bytes can be reused for another record, as suggested by Fig. 3.19.[4] When we follow a pointer to the deleted record, the first thing we see is the "tombstone" bit telling us that the record was deleted. We then know not to look at the following bytes.

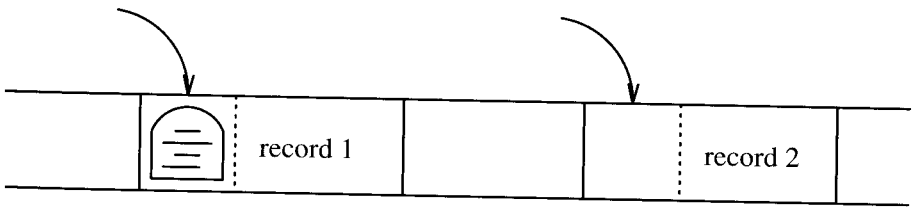

Figure 3.19: Record 1 can be replaced, but the tombstone remains; record 2 has no tombstone and can be seen by following a pointer to it

3.5.3 Update

When a fixed-length record is updated, there is no effect on the storage system, because we know it can occupy exactly the same space it did before the update. However, when a variable-length record is updated, we have all the problems associated with both insertion and deletion, except that it is never necessary to create a tombstone for the old version of the record.

If the updated record is longer than the old version, then we may need to create more space on its block. This process may involve sliding records or even the creation of an overflow block. If variable-length portions of the record are stored on another block, as in Fig. 3.14, then we may need to move elements around that block or create a new block for storing variable-length

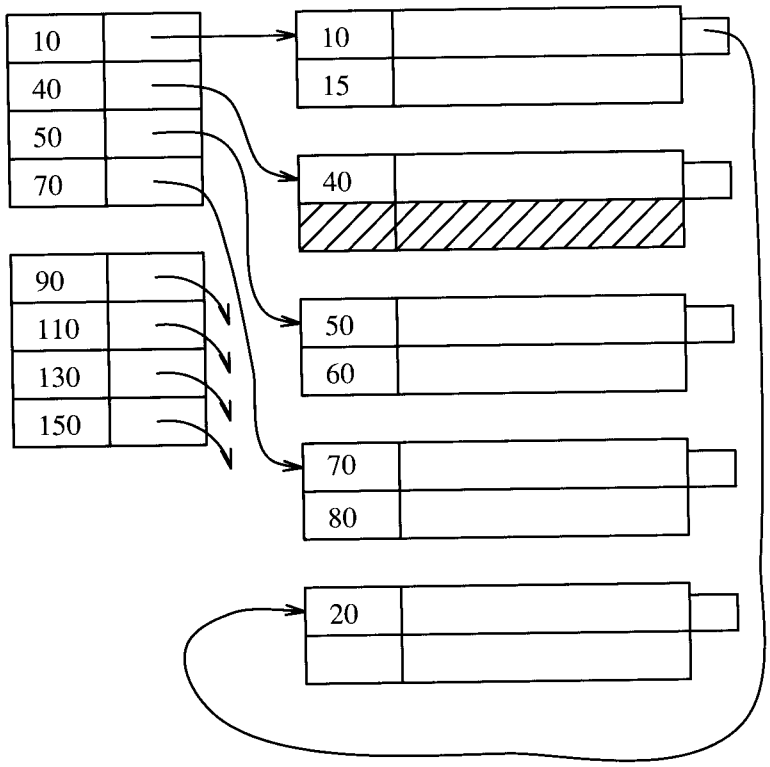

Figure 4.14: Insertion into a file with a sparse index, using overflow blocks

of data block 1, not a block of the sequential file on its own. □

4.1.7 Exercises for Section 4.1

* **Exercise 4.1.1:** Suppose blocks hold either three records, or ten key-pointer pairs. As a function of n, the number of records, how many blocks do we need to hold a data file and:

  a) A dense index?

  b) A sparse index?

Exercise 4.1.2: Repeat Exercise 4.1.1 if blocks can hold up to 30 records or 200 key-pointer pairs, but neither data- nor index-blocks are allowed to be more than 80% full.

! **Exercise 4.1.3:** Repeat Exercise 4.1.1 if we use as many levels of index as is appropriate, until the final level of index has only one block.

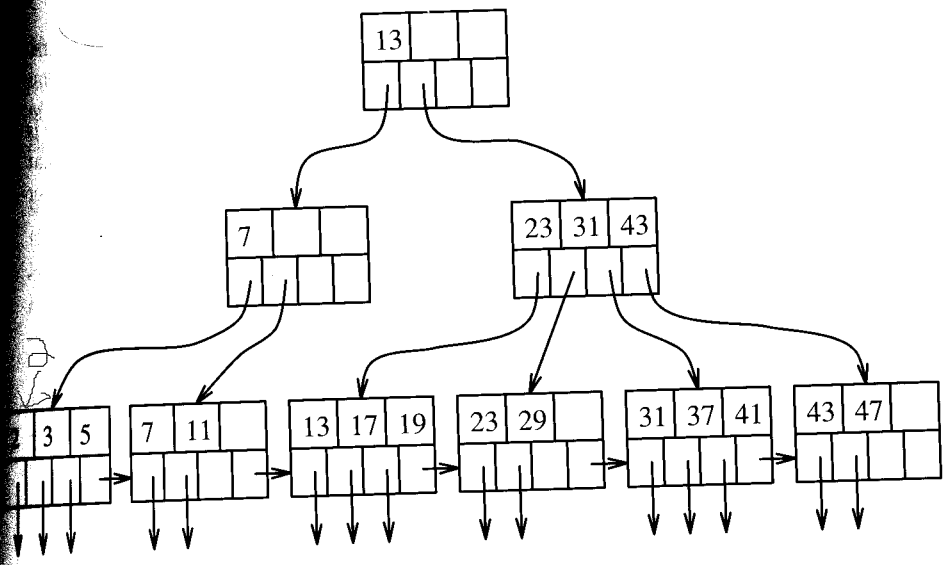

Figure 4.23: A B+ tree

via the first pointer from those reachable via the second. That is, keys up to 12 could be found in the first subtree of the root, and keys 13 and up are in the second subtree.

If we look at the first child of the root, with key 7, we again find two pointers, one to keys less than 7 and the other to keys 7 and above. Note that the second pointer in this node gets us only to keys 7 and 11, not to *all* keys ≥ 7, such as 13 (although we could reach the larger keys by following the next-block pointers in the leaves).

Finally, the second child of the root has all four pointer slots in use. The first gets us to some of the keys less than 23, namely 13, 17, and 19. The second pointer gets us to all keys K such that $23 \leq K < 31$; the third pointer lets us reach all keys K such that $31 \leq K < 43$, and the fourth pointer gets us to some of the keys ≥ 43 (in this case, to all of them). □

4.3.2 Applications of B-trees

The B-tree is a powerful tool for building indexes. The sequence of pointers to records at the leaves can play the role of any of the pointer sequences coming out of an index file that we learned about in Sections 4.1 or 4.2. Here are some examples:

1. The search key of the B-tree is the primary key for the data file, and the index is dense. That is, there is one key-pointer pair in a leaf for every record of the data file. The data file may or may not be sorted by primary key.

Exercise 4.3.4: What are the minimum numbers of keys and pointers in B-tree (*i*) interior nodes and (*ii*) leaves, when:

* a) $n = 10$; i.e., a block holds 10 keys and 11 pointers.

b) $n = 11$; i.e., a block holds 11 keys and 12 pointers.

Exercise 4.3.5: Execute the following operations on Fig. 4.23. Describe the changes for operations that modify the tree.

a) Lookup the record with key 41.

b) Lookup the record with key 40.

c) Lookup all records in the range 20 to 30.

d) Lookup all records with keys less than 30.

e) Lookup all records with keys greater than 30.

f) Insert a record with key 1.

g) Insert records with keys 14 through 16.

h) Delete the record with key 23.

i) Delete all the records with keys 23 and higher.

! **Exercise 4.3.6:** We mentioned that the leaf of Fig. 4.21 and the interior node of Fig. 4.22 could never appear in the same B-tree. Explain why.

Exercise 4.3.7: When duplicate keys are allowed in a B-tree, there are some necessary modifications to the algorithms for lookup, insertion, and deletion that we described in this section. Give the changes for:

* a) Lookup.

b) Insertion.

c) Deletion.

! **Exercise 4.3.8:** In Example 4.26 we suggested that it would be possible to borrow keys from a nonsibling to the right (or left) if we used a more complicated algorithm for maintaining keys at interior nodes. Describe a suitable algorithm that rebalances by borrowing from adjacent nodes at a level, regardless of whether they are siblings of the node that has too many or too few key-pointer pairs.

Exercise 4.3.9: If we use the 3-key, 4-pointer nodes of our examples in this section, how many different B-trees are there when the data file has:

key (which we may call the *hash key*) as an argument and computes from it an integer in the range 0 to $B-1$, where B is the number of *buckets*. A *bucket array*, which is an array indexed from 0 to $B-1$, holds the headers of B linked lists, one for each bucket of the array. If a record has search key K, then we store the record by linking it to the bucket list for the bucket numbered $h(K)$, where h is the hash function.

4.4.1 Secondary-Storage Hash Tables

A hash table that holds a very large number of records, so many that they must be kept mainly in secondary storage, differs from the main-memory version in small but important ways. First, the bucket array consists of blocks, rather than pointers to the headers of lists. Records that are hashed by the hash function h to a certain bucket are put in the block for that bucket. If a bucket *overflows*, meaning that it cannot hold all the records that belong in that bucket, then a chain of *overflow blocks* can be added to the bucket to hold more records.

We shall assume that the location of the first block for any bucket i can be found given i. For example, there might be a main-memory array of pointers to blocks, indexed by the bucket number. Another possibility is to put the first block for each bucket in fixed, consecutive disk locations, so we can compute the location of bucket i from the integer i.

Example 4.28: Figure 4.30 shows a hash table. To keep our illustrations manageable, we assume that a block can hold only two records, and that $B = 4$; i.e., the hash function h returns values from 0 to 3. We show certain records populating the hash table. Keys are letters a through f in Fig. 4.30. We assume that $h(d) = 0$, $h(c) = h(e) = 1$, $h(b) = 2$, and $h(a) = h(f) = 3$. Thus, the six records are distributed into blocks as shown. □

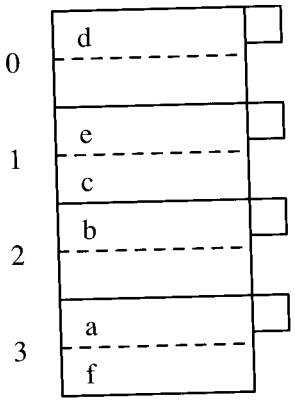

Figure 4.30: A hash table

Note that we show each block in Fig. 4.30 with a "nub" at the right end.

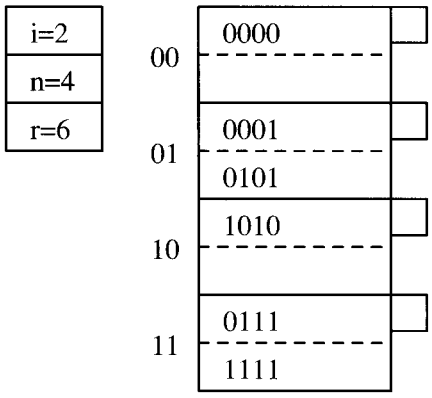

Figure 4.39: Adding a fourth bucket

must look in the bucket whose number is 11. Since that bucket number as a binary integer is $m = 3$, and $m \geq n$, the bucket 11 does not exist. We redirect to bucket 01 by changing the leading 1 to 0. However, bucket 01 has no record whose key has hash value 1011, and therefore surely our desired record is not in the hash table. □

4.4.9 Exercises for Section 4.4

Exercise 4.4.1: Show what happens to the buckets in Fig. 4.30 if the following insertions and deletions occur:

i. Records g through j are inserted into buckets 0 through 3, respectively.

ii. Records a and b are deleted.

iii. Records k through n are inserted into buckets 0 through 3, respectively.

iv. Records c and d are deleted.

Exercise 4.4.2: We did not discuss how deletions can be carried out in a linear or extensible hash table. The mechanics of locating the record(s) to be deleted should be obvious. What method would you suggest for executing the deletion? In particular, what are the advantages and disadvantages of restructuring the table if its smaller size after deletion allows for compression of certain blocks?

! **Exercise 4.4.3:** The material of this section assumes that search keys are unique. However, only small modifications are needed to allow the techniques to work for search keys with duplicates. Describe the necessary changes to insertion, deletion, and lookup algorithms, and suggest the major problems that arise when there are duplicates in:

* a) A simple hash table.