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 14.2.6: 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 14.2.7:** In Example 14.17 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 14.2.8:** 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 the following numbers of records: (a) 6 (b) 10 !! (c) 15.

! **Exercise 14.2.9:** Suppose we have B-tree nodes with room for three keys and four pointers, as in the examples of this section. Suppose also that when we split a leaf, we divide the pointers 2 and 2, while when we split an interior node, the first 3 pointers go with the first (left) node, and the last 2 pointers go with the second (right) node. We start with a leaf containing pointers to records with keys 1, 2, and 3. We then add in order, records with keys 4, 5, 6, and so on. At the insertion of what key will the B-tree first reach four levels?

14.3 Hash Tables

There are a number of data structures involving a hash table that are useful as indexes. We assume the reader has seen the hash table used as a main-memory data structure. In such a structure there is a *hash function* h that takes a search key (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 $h(K)$.

14.3.1 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 has too many records, 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.

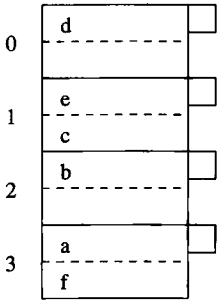

Figure 14.20: A hash table

Example 14.19: Figure 14.20 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. 14.20. 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. □

Note that we show each block in Fig. 14.20 with a "nub" at the right end. This nub represents additional information in the block's header. We shall use it to chain overflow blocks together, and starting in Section 14.3.5, we shall use it to keep other critical information about the block.

14.3.2 Insertion Into a Hash Table

When a new record with search key K must be inserted, we compute $h(K)$. If the bucket numbered $h(K)$ has space, then we insert the record into the block for this bucket, or into one of the overflow blocks on its chain if there is no room

Choice of Hash Function

The hash function should "hash" the key so the resulting integer is a seemingly random function of the key. Thus, buckets will tend to have equal numbers of records, which improves the average time to access a record, as we shall discuss in Section 14.3.4. Also, the hash function should be easy to compute, since we shall compute it many times.

A common choice of hash function when keys are integers is to compute the remainder of K/B, where K is the key value and B is the number of buckets. Often, B is chosen to be a prime, although there are reasons to make B a power of 2, as we discuss starting in Section 14.3.5. For character-string search keys, we may treat each character as an integer, sum these integers, and take the remainder when the sum is divided by B.

in the first block. If none of the blocks of the chain for bucket $h(K)$ has room, we add a new overflow block to the chain and store the new record there.

Example 14.20: Suppose we add to the hash table of Fig. 14.20 a record with key g, and $h(g) = 1$. Then we must add the new record to the bucket numbered 1. However, the block for that bucket already has two records. Thus, we add a new block and chain it to the original block for bucket 1. The record with key g goes in that block, as shown in Fig. 14.21. □

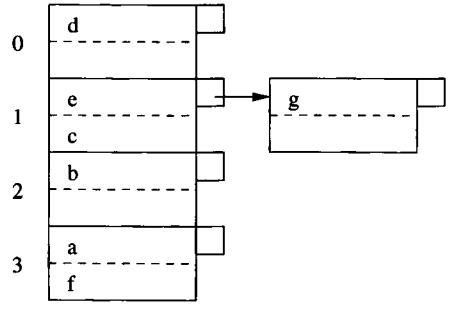

Figure 14.21: Adding an additional block to a hash-table bucket

14.3.3 Hash-Table Deletion

Deletion of the record (or records) with search key K follows the same pattern as insertion. We go to the bucket numbered $h(K)$ and search for records with that search key. Any that we find are deleted. If we are able to move records

around among blocks, then after deletion we may optionally consolidate the blocks of a bucket into one fewer block.[6]

Example 14.21: Figure 14.22 shows the result of deleting the record with key c from the hash table of Fig. 14.21. Recall $h(c) = 1$, so we go to the bucket numbered 1 (i.e., the second bucket) and search all its blocks to find a record (or records if the search key were not the primary key) with key c. We find it in the first block of the chain for bucket 1. Since there is now room to move the record with key g from the second block of the chain to the first, we can do so and remove the second block.

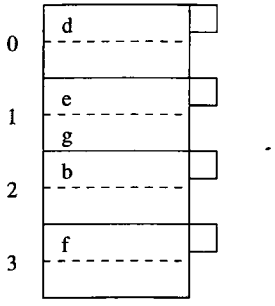

Figure 14.22: Result of deletions from a hash table

We also show the deletion of the record with key a. For this key, we found our way to bucket 3, deleted it, and "consolidated" the remaining record at the beginning of the block. □

14.3.4 Efficiency of Hash Table Indexes

Ideally, there are enough buckets that most of them fit on one block. If so, then the typical lookup takes only one disk I/O, and insertion or deletion from the file takes only two disk I/O's. That number is significantly better than straightforward sparse or dense indexes, or B-tree indexes (although hash tables do not support range queries as B-trees do; see Section 14.2.4).

However, if the file grows, then we shall eventually reach a situation where there are many blocks in the chain for a typical bucket. If so, then we need to search long lists of blocks, taking at least one disk I/O per block. Thus, there is a good reason to try to keep the number of blocks per bucket low.

The hash tables we have examined so far are called *static hash tables*, because B, the number of buckets, never changes. However, there are several kinds of *dynamic hash tables*, where B is allowed to vary so it approximates the number

[6] A risk of consolidating blocks of a chain whenever possible is that an oscillation, where we alternately insert and delete records from a bucket, will cause a block to be created or destroyed at each step.

of records divided by the number of records that can fit on a block; i.e., there is about one block per bucket. We shall discuss two such methods:

1. Extensible hashing in Section 14.3.5, and

2. Linear hashing in Section 14.3.7.

The first grows B by doubling it whenever it is deemed too small, and the second grows B by 1 each time statistics of the file suggest some growth is needed.

14.3.5 Extensible Hash Tables

Our first approach to dynamic hashing is called *extensible hash tables*. The major additions to the simpler static hash table structure are:

1. There is a level of indirection for the buckets. That is, an array of pointers to blocks represents the buckets, instead of the array holding the data blocks themselves.

2. The array of pointers can grow. Its length is always a power of 2, so in a growing step the number of buckets doubles.

3. However, there does not have to be a data block for each bucket; certain buckets can share a block if the total number of records in those buckets can fit in the block.

4. The hash function h computes for each key a sequence of k bits for some large k, say 32. However, the bucket numbers will at all times use some smaller number of bits, say i bits, from the beginning or end of this sequence. The bucket array will have 2^i entries when i is the number of bits used.

Example 14.22: Figure 14.23 shows a small extensible hash table. We suppose, for simplicity of the example, that $k = 4$; i.e., the hash function produces a sequence of only four bits. At the moment, only one of these bits is used, as indicated by $i = 1$ in the box above the bucket array. The bucket array therefore has only two entries, one for 0 and one for 1.

The bucket array entries point to two blocks. The first holds all the current records whose search keys hash to a bit sequence that begins with 0, and the second holds all those whose search keys hash to a sequence beginning with 1. For convenience, we show the keys of records as if they were the entire bit sequence to which the hash function converts them. Thus, the first block holds a record whose key hashes to 0001, and the second holds records whose keys hash to 1001 and 1100. □

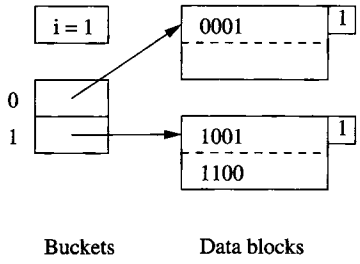

Figure 14.23: An extensible hash table

We should notice the number 1 appearing in the "nub" of each of the blocks in Fig. 14.23. This number, which would actually appear in the block header, indicates how many bits of the hash function's sequence is used to determine membership of records in this block. In the situation of Example 14.22, there is only one bit considered for all blocks and records, but as we shall see, the number of bits considered for various blocks can differ as the hash table grows. That is, the bucket array size is determined by the maximum number of bits we are now using, but some blocks may use fewer.

14.3.6 Insertion Into Extensible Hash Tables

Insertion into an extensible hash table begins like insertion into a static hash table. To insert a record with search key K, we compute $h(K)$, take the first i bits of this bit sequence, and go to the entry of the bucket array indexed by these i bits. Note that we can determine i because it is kept as part of the data structure.

We follow the pointer in this entry of the bucket array and arrive at a block B. If there is room to put the new record in block B, we do so and we are done. If there is no room, then there are two possibilities, depending on the number j, which indicates how many bits of the hash value are used to determine membership in block B (recall the value of j is found in the "nub" of each block in figures).

1. If $j < i$, then nothing needs to be done to the bucket array. We:
   (a) Split block B into two.
   (b) Distribute records in B to the two blocks, based on the value of their $(j+1)$st bit — records whose key has 0 in that bit stay in B and those with 1 there go to the new block.
   (c) Put $j+1$ in each block's "nub" (header) to indicate the number of bits used to determine membership.
   (d) Adjust the pointers in the bucket array so entries that formerly pointed to B now point either to B or the new block, depending on their $(j+1)$st bit.

Note that splitting block B may not solve the problem, since by chance all the records of B may go into one of the two blocks into which it was split. If so, we need to repeat the process on the overfull block, using the next higher value of j and the block that is still overfull.

2. If $j = i$, then we must first increment i by 1. We double the length of the bucket array, so it now has 2^{i+1} entries. Suppose w is a sequence of i bits indexing one of the entries in the previous bucket array. In the new bucket array, the entries indexed by both $w0$ and $w1$ (i.e., the two numbers derived from w by extending it with 0 or 1) each point to the same block that the w entry used to point to. That is, the two new entries share the block, and the block itself does not change. Membership in the block is still determined by whatever number of bits was previously used. Finally, we proceed to split block B as in case 1. Since i is now greater than j, that case applies.

Example 14.23: Suppose we insert into the table of Fig. 14.23 a record whose key hashes to the sequence 1010. Since the first bit is 1, this record belongs in the second block. However, that block is already full, so it needs to be split. We find that $j = i = 1$ in this case, so we first need to double the bucket array, as shown in Fig. 14.24. We have also set $i = 2$ in this figure.

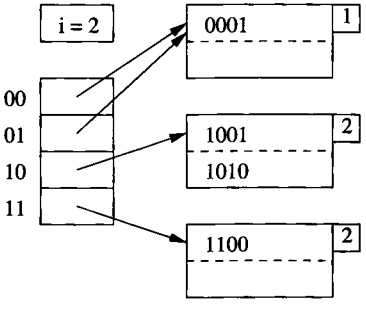

Figure 14.24: Now, two bits of the hash function are used

Notice that the two entries beginning with 0 each point to the block for records whose hashed keys begin with 0, and that block still has the integer 1 in its "nub" to indicate that only the first bit determines membership in the block. However, the block for records beginning with 1 needs to be split, so we partition its records into those beginning 10 and those beginning 11. A 2 in each of these blocks indicates that two bits are used to determine membership. Fortunately, the split is successful; since each of the two new blocks gets at least one record, we do not have to split recursively.

Now suppose we insert records whose keys hash to 0000 and 0111. These both go in the first block of Fig. 14.24, which then overflows. Since only one bit is used to determine membership in this block, while $i = 2$, we do not have to

adjust the bucket array. We simply split the block, with 0000 and 0001 staying, and 0111 going to the new block. The entry for 01 in the bucket array is made to point to the new block. Again, we have been fortunate that the records did not all go in one of the new blocks, so we have no need to split recursively.

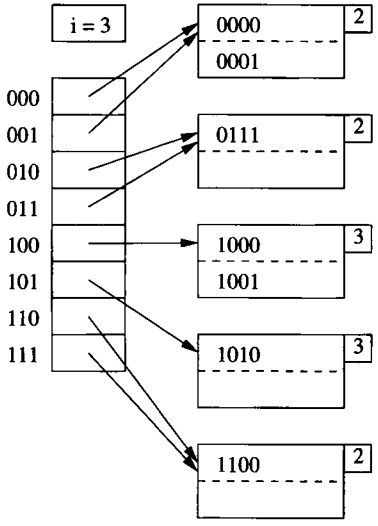

Figure 14.25: The hash table now uses three bits of the hash function

Now suppose a record whose key hashes to 1000 is inserted. The block for 10 overflows. Since it already uses two bits to determine membership, it is time to split the bucket array again and set $i = 3$. Figure 14.25 shows the data structure at this point. Notice that the block for 10 has been split into blocks for 100 and 101, while the other blocks continue to use only two bits to determine membership. □

14.3.7 Linear Hash Tables

Extensible hash tables have some important advantages. Most significant is the fact that when looking for a record, we never need to search more than one data block. We also have to examine an entry of the bucket array, but if the bucket array is small enough to be kept in main memory, then there is no disk I/O needed to access the bucket array. However, extensible hash tables also suffer from some defects:

1. When the bucket array needs to be doubled in size, there is a substantial amount of work to be done (when i is large). This work interrupts access to the data file, or makes certain insertions appear to take a long time.

2. When the bucket array is doubled in size, it may no longer fit in main memory, or may crowd out other data that we would like to hold in main memory. As a result, a system that was performing well might suddenly start using many more disk I/O's per operation.

3. If the number of records per block is small, then there is likely to be one block that needs to be split well in advance of the logical time to do so. For instance, if there are two records per block as in our running example, there might be one sequence of 20 bits that begins the keys of three records, even though the total number of records is much less than 2^{20}. In that case, we would have to use $i = 20$ and a million-bucket array, even though the number of blocks holding records was much smaller than a million.

Another strategy, called *linear hashing*, grows the number of buckets more slowly. The principal new elements we find in linear hashing are:

- The number of buckets n is always chosen so the average number of records per bucket is a fixed fraction, say 80%, of the number of records that fill one block.

- Since blocks cannot always be split, overflow blocks are permitted, although the average number of overflow blocks per bucket will be much less than 1.

- The number of bits used to number the entries of the bucket array is $\lceil \log_2 n \rceil$, where n is the current number of buckets. These bits are always taken from the *right* (low-order) end of the bit sequence that is produced by the hash function.

- Suppose i bits of the hash function are being used to number array entries, and a record with key K is intended for bucket $a_1a_2\cdots a_i$; that is, $a_1a_2\cdots a_i$ are the last i bits of $h(K)$. Then let $a_1a_2\cdots a_i$ be m, treated as an i-bit binary integer. If $m < n$, then the bucket numbered m exists, and we place the record in that bucket. If $n \leq m < 2^i$, then the bucket m does not yet exist, so we place the record in bucket $m - 2^{i-1}$, that is, the bucket we would get if we changed a_1 (which must be 1) to 0.

Example 14.24: Figure 14.26 shows a linear hash table with $n = 2$. We currently are using only one bit of the hash value to determine the buckets of records. Following the pattern established in Example 14.22, we assume the hash function h produces 4 bits, and we represent records by the value produced by h when applied to the search key of the record.

We see in Fig. 14.26 the two buckets, each consisting of one block. The buckets are numbered 0 and 1. All records whose hash value ends in 0 go in the first bucket, and those whose hash value ends in 1 go in the second.

Also part of the structure are the parameters i (the number of bits of the hash function that currently are used), n (the current number of buckets), and r

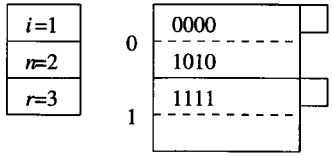

Figure 14.26: A linear hash table

(the current number of records in the hash table). The ratio r/n will be limited so that the typical bucket will need about one disk block. We shall adopt the policy of choosing n, the number of buckets, so that there are no more than $1.7n$ records in the file; i.e., $r \leq 1.7n$. That is, since blocks hold two records, the average occupancy of a bucket does not exceed 85% of the capacity of a block. □

14.3.8 Insertion Into Linear Hash Tables

When we insert a new record, we determine its bucket by the algorithm outlined in Section 14.3.7. We compute $h(K)$, where K is the key of the record, and we use the i bits at the end of bit sequence $h(K)$ as the bucket number, m. If $m < n$, we put the record in bucket m, and if $m \geq n$, we put the record in bucket $m - 2^{i-1}$. If there is no room in the designated bucket, then we create an overflow block, add it to the chain for that bucket, and put the record there.

Each time we insert, we compare the current number of records r with the threshold ratio of r/n, and if the ratio is too high, we add the next bucket to the table. Note that the bucket we add bears no relationship to the bucket into which the insertion occurs! If the binary representation of the number of the bucket we add is $1a_2 \cdots a_i$, then we split the bucket numbered $0a_2 \cdots a_i$, putting records into one or the other bucket, depending on their last i bits. Note that all these records will have hash values that end in $a_2 \cdots a_i$, and only the ith bit from the right end will vary.

The last important detail is what happens when n exceeds 2^i. Then, i is incremented by 1. Technically, all the bucket numbers get an additional 0 in front of their bit sequences, but there is no need to make any physical change, since these bit sequences, interpreted as integers, remain the same.

Example 14.25: We shall continue with Example 14.24 and consider what happens when a record whose key hashes to 0101 is inserted. Since this bit sequence ends in 1, the record goes into the second bucket of Fig. 14.26. There is room for the record, so no overflow block is created.

However, since there are now 4 records in 2 buckets, we exceed the ratio 1.7, and we must therefore raise n to 3. Since $\lceil \log_2 3 \rceil = 2$, we should begin to think of buckets 0 and 1 as 00 and 01, but no change to the data structure is necessary. We add to the table the next bucket, which would have number 10. Then, we split the bucket 00, that bucket whose number differs from the added

bucket only in the first bit. When we do the split, the record whose key hashes to 0000 stays in 00, since it ends with 00, while the record whose key hashes to 1010 goes to 10 because it ends that way. The resulting hash table is shown in Fig. 14.27.

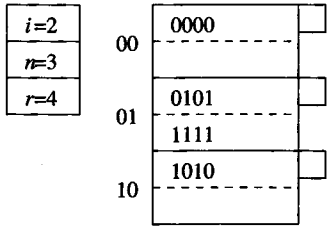

Figure 14.27: Adding a third bucket

Next, let us suppose we add a record whose search key hashes to 0001. The last two bits are 01, so we put it in this bucket, which currently exists. Unfortunately, the bucket's block is full, so we add an overflow block. The three records are distributed among the two blocks of the bucket; we chose to keep them in numerical order of their hashed keys, but order is not important. Since the ratio of records to buckets for the table as a whole is 5/3, and this ratio is less than 1.7, we do not create a new bucket. The result is seen in Fig. 14.28.

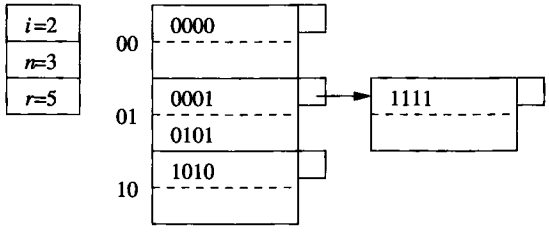

Figure 14.28: Overflow blocks are used if necessary

Finally, consider the insertion of a record whose search key hashes to 0111. The last two bits are 11, but bucket 11 does not yet exist. We therefore redirect this record to bucket 01, whose number differs by having a 0 in the first bit. The new record fits in the overflow block of this bucket.

However, the ratio of the number of records to buckets has exceeded 1.7, so we must create a new bucket, numbered 11. Coincidentally, this bucket is the one we wanted for the new record. We split the four records in bucket 01, with 0001 and 0101 remaining, and 0111 and 1111 going to the new bucket. Since bucket 01 now has only two records, we can delete the overflow block. The hash table is now as shown in Fig. 14.29.

Notice that the next time we insert a record into Fig. 14.29, we shall exceed

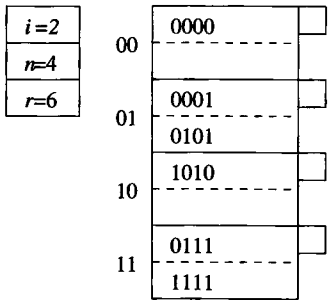

Figure 14.29: Adding a fourth bucket

the 1.7 ratio of records to buckets. Then, we shall raise n to 5 and i becomes 3. □

Lookup in a linear hash table follows the procedure we described for selecting the bucket in which an inserted record belongs. If the record we wish to look up is not in that bucket, it cannot be anywhere.

14.3.9 Exercises for Section 14.3

Exercise 14.3.1: Show what happens to the buckets in Fig. 14.20 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 14.3.2: 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 14.3.3:** 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 each of the following kinds of hash tables: (a) simple (b) linear (c) extensible.

! **Exercise 14.3.4:** Some hash functions do not work as well as theoretically possible. Suppose that we use the hash function on integer keys i defined by $h(i) = i^2 \bmod B$, where B is the number of buckets.

a) What is wrong with this hash function if $B = 10$?

b) How good is this hash function if $B = 16$?

c) Are there values of B for which this hash function is useful?

Exercise 14.3.5: In an extensible hash table with n records per block, what is the probability that an overflowing block will have to be handled recursively; i.e., all members of the block will go into the same one of the two blocks created in the split?

Exercise 14.3.6: Suppose keys are hashed to four-bit sequences, as in our examples of extensible and linear hashing in this section. However, also suppose that blocks can hold three records, rather than the two-record blocks of our examples. If we start with a hash table with two empty blocks (corresponding to 0 and 1), show the organization after we insert records with hashed keys:

a) $0000, 0001, \ldots, 1111$, and the method of hashing is extensible hashing.

b) $0000, 0001, \ldots, 1111$, and the method of hashing is linear hashing with a capacity threshold of 100%.

c) $1111, 1110, \ldots, 0000$, and the method of hashing is extensible hashing.

d) $1111, 1110, \ldots, 0000$, and the method of hashing is linear hashing with a capacity threshold of 75%.

Exercise 14.3.7: Suppose we use a linear or extensible hashing scheme, but there are pointers to records from outside. These pointers prevent us from moving records between blocks, as is sometimes required by these hashing methods. Suggest several ways that we could modify the structure to allow pointers from outside.

!! **Exercise 14.3.8:** A linear-hashing scheme with blocks that hold k records uses a threshold constant c, such that the current number of buckets n and the current number of records r are related by $r = ckn$. For instance, in Example 14.24 we used $k = 2$ and $c = 0.85$, so there were 1.7 records per bucket; i.e., $r = 1.7n$.

a) Suppose for convenience that each key occurs exactly its expected number of times.[7] As a function of c, k, and n, how many blocks, including overflow blocks, are needed for the structure?

[7]This assumption does not mean all buckets have the same number of records, because some buckets represent twice as many keys as others.

b) Keys will not generally distribute equally, but rather the number of records with a given key (or suffix of a key) will be *Poisson distributed.* That is, if λ is the expected number of records with a given key suffix, then the actual number of such records will be i with probability $e^{-\lambda}\lambda^i/i!$. Under this assumption, calculate the expected number of blocks used, as a function of c, k, and n.

! **Exercise 14.3.9:** Suppose we have a file of 1,000,000 records that we want to hash into a table with 1000 buckets. 100 records will fit in a block, and we wish to keep blocks as full as possible, but not allow two buckets to share a block. What are the minimum and maximum number of blocks that we could need to store this hash table?

14.4 Multidimensional Indexes

All the index structures discussed so far are *one dimensional*; that is, they assume a single search key, and they retrieve records that match a given search-key value. Although the search key may involve several attributes, the one-dimensional nature of indexes such as B-trees comes from the fact that values must be provided for all attributes of the search key, or the index is useless. So far in this chapter, we took advantage of a one-dimensional search-key space in several ways:

- Indexes on sequential files and B-trees both take advantage of having a single linear order for the keys.
- Hash tables require that the search key be completely known for any lookup. If a key consists of several fields, and even one is unknown, we cannot apply the hash function, but must instead search all the buckets.

In the balance of this chapter, we shall look at index structures that are suitable for multidimensional data. In these structures, any nonempty subset of the fields that form the dimensions can be given values, and some speedup will result.

14.4.1 Applications of Multidimensional Indexes

There are a number of applications that require us to view data as existing in a 2-dimensional space, or sometimes in higher dimensions. Some of these applications can be supported by conventional DBMS's, but there are also some specialized systems designed for multidimensional applications. One way in which these specialized systems distinguish themselves is by using data structures that support certain kinds of queries that are not common in SQL applications.

One important application of multidimensional indexes involves geographic data. A *geographic information system* stores objects in a (typically) two-dimensional space. The objects may be points or shapes. Often, these databases