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.

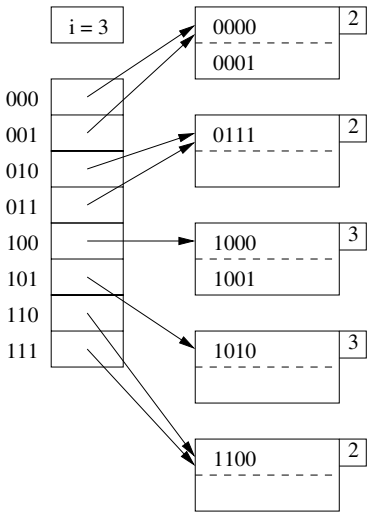

Figure 25: The hash table now uses three bits of the hash function

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 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. □

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 $a_1 a_2 \cdots a_i$; that is, $a_1 a_2 \cdots a_i$ are the last i bits of $h(K)$. Then let $a_1 a_2 \cdots a_i$ 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 24: Figure 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 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. 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

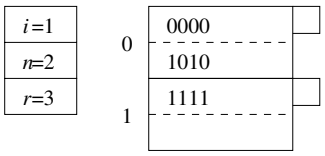

Figure 26: A linear hash table

(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. □

3.8 Insertion Into Linear Hash Tables

When we insert a new record, we determine its bucket by the algorithm outlined in Section 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 25: We shall continue with Example 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. 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. 27.

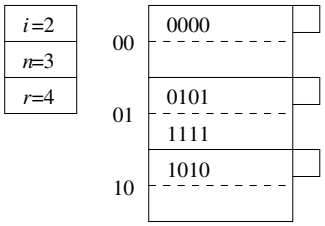

Figure 27: Adding a third bucket

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. 28.

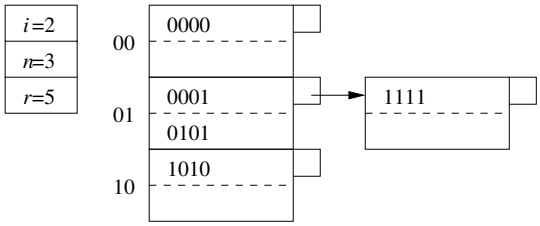

Figure 28: Overflow blocks are used if necessary

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. 29.

Notice that the next time we insert a record into Fig. 29, we shall exceed

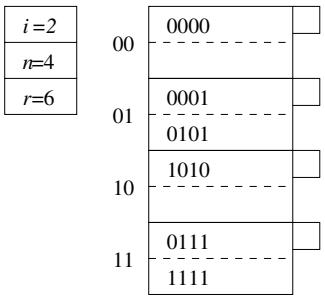

Figure 29: Adding a fourth bucket

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.

3.9 Exercises for Section 3

Exercise 3.1: Show what happens to the buckets in Fig. 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 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 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.