### 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.
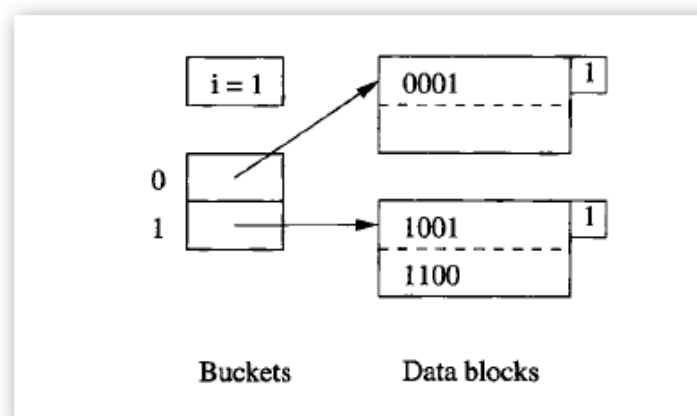


**Figure 14.23**: An extensible hash table

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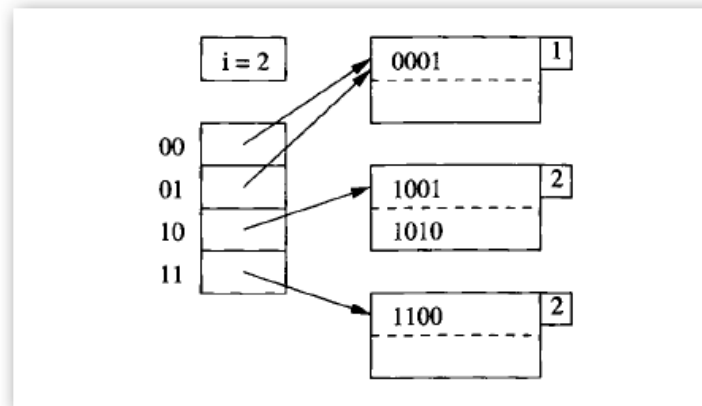
**Figure 14.24**: Now, two bits of the hash function are used

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.
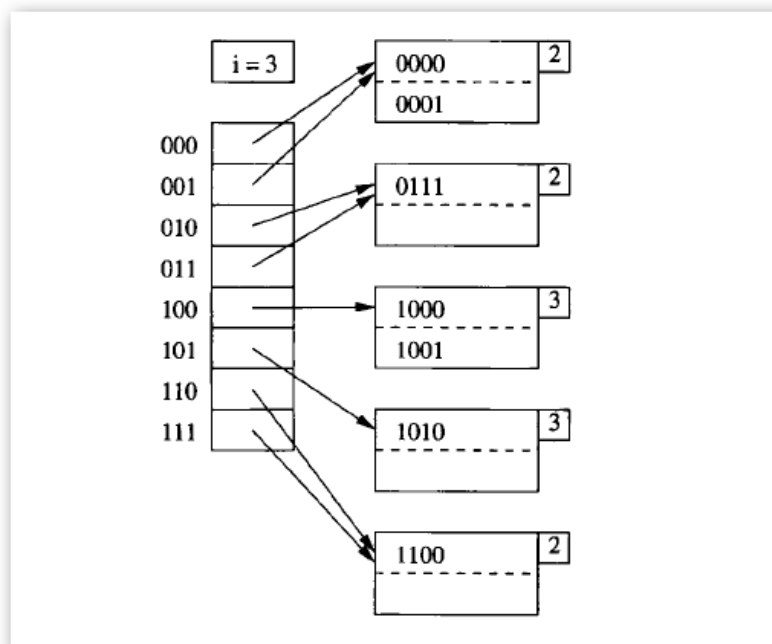


**Figure 14.25**: The hash table now uses 3 bits of the hash function

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 log2 n, 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 $a_1a_2 \ldots a_i$; that is $a_1a_2 \ldots a_i$ are the last i bits of h(K ). Then let $a_1a_2 \ldots a_i$ 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 $n < m < 2^i$, 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.

**E x am p le 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
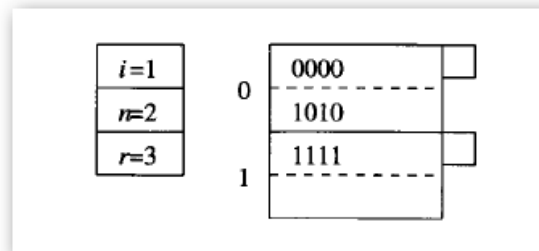


**Figure 14.26**: A linear hash table

(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 < 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 > 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 $1a_2 \ldots a_i$, then we split the bucket numbered $0a_2 \ldots a_i$, 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 $a_2 \ldots a_i$, 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.
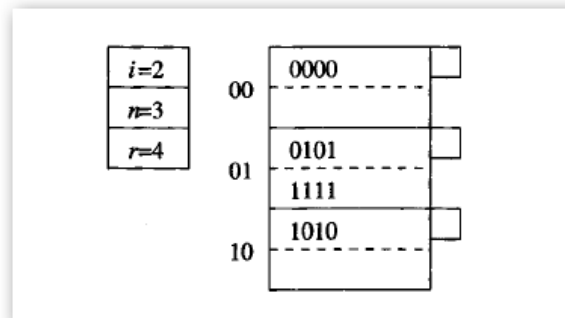


**Figure 14.27**: Adding a third bucket

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.
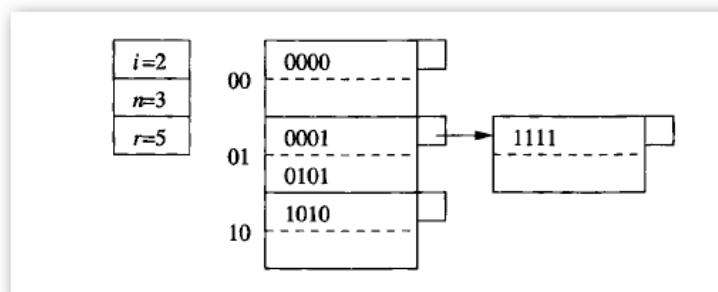


**Figure 14.28**: Overflow blocks are used if necessary

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 the 1.7 ratio of records to buckets. Then, we shall raise n to 5 and i becomes 3.
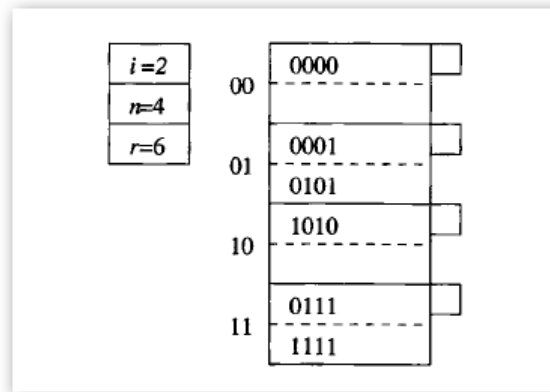
**Figure 14.29**: Adding a fourth bucket