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. □
14.3. HASH TABLES

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.
CHAPTER 14. INDEX STRUCTURES

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](image)

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.

![Diagram of hash table](image)

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.