The main problem with Static Hashing is that the number of buckets is fixed. If a file shrinks greatly, a lot of space is wasted; more importantly, if a file grows a lot, long overflow chains develop, resulting in poor performance. One alternative is to periodically 'rehash' the file to restore the ideal situation (no overflow chains, about 80 percent occupancy). However, rehashing takes time and the index cannot be used while rehashing is in progress. Another alternative is to use **dynamic hashing** techniques such as Extendible and Linear Hashing, which deal with inserts and deletes gracefully. We consider these techniques in the rest of this chapter.

10.1.1 Notation and Conventions

In the rest of this chapter, we use the following conventions. The first step in searching for, inserting, or deleting a data entry $k*$ (with search key k) is always to apply a hash function h to the search field, and we will denote this operation as $h(k)$. The value $h(k)$ identifies a bucket. We will often denote the data entry $k*$ by using the hash value, as $h(k)*$. Note that two different keys can have the same hash value.

10.2 EXTENDIBLE HASHING *

To understand Extendible Hashing, let us begin by considering a Static Hashing file. If we have to insert a new data entry into a full bucket, we need to add an overflow page. If we don't want to add overflow pages, one solution is to reorganize the file at this point by doubling the number of buckets and redistributing the entries across the new set of buckets. This solution suffers from one major defect—the entire file has to be read, and twice as many pages have to be written, to achieve the reorganization. This problem, however, can be overcome by a simple idea: use a **directory** of pointers to buckets, and double the size of the number of buckets by doubling just the directory and splitting *only* the bucket that overflowed.

To understand the idea, consider the sample file shown in Figure 10.2. The directory consists of an array of size 4, with each element being a pointer to a bucket. (The *global depth* and *local depth* fields will be discussed shortly; ignore them for now.) To locate a data entry, we apply a hash function to the search field and take the last two bits of its binary representation to get a number between 0 and 3. The pointer in this array position gives us the desired bucket; we assume that each bucket can hold four data entries. Thus, to locate a data entry with hash value 5 (binary 101), we look at directory element 01 and follow the pointer to the data page (bucket B in the figure).

To insert a data entry, we search to find the appropriate bucket. For example, to insert a data entry with hash value 13 (denoted as 13*), we would examine directory element 01 and go to the page containing data entries 1*, 5*, and 21*. Since the page has space for an additional data entry, we are done after we insert the entry (Figure 10.3).

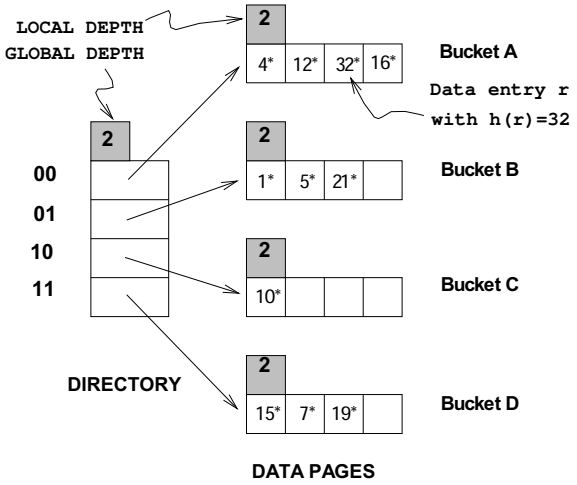

Figure 10.2 Example of an Extendible Hashed File

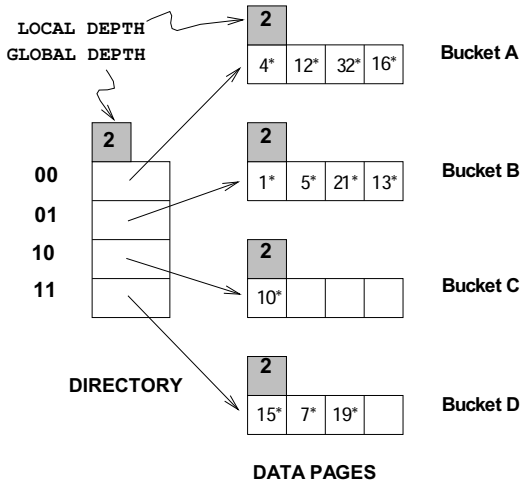

Figure 10.3 After Inserting Entry r with $h(r)$=13

Next, let us consider insertion of a data entry into a full bucket. The essence of the Extendible Hashing idea lies in how we deal with this case. Consider the insertion of data entry 20* (binary 10100). Looking at directory element 00, we are led to bucket A, which is already full. We must first **split** the bucket by allocating a new bucket[1] and redistributing the contents (including the new entry to be inserted) across the old bucket and its 'split image.' To redistribute entries across the old bucket and its split image, we consider the last *three* bits of $h(r)$; the last two bits are 00, indicating a data entry that belongs to one of these two buckets, and the third bit discriminates between these buckets. The redistribution of entries is illustrated in Figure 10.4.

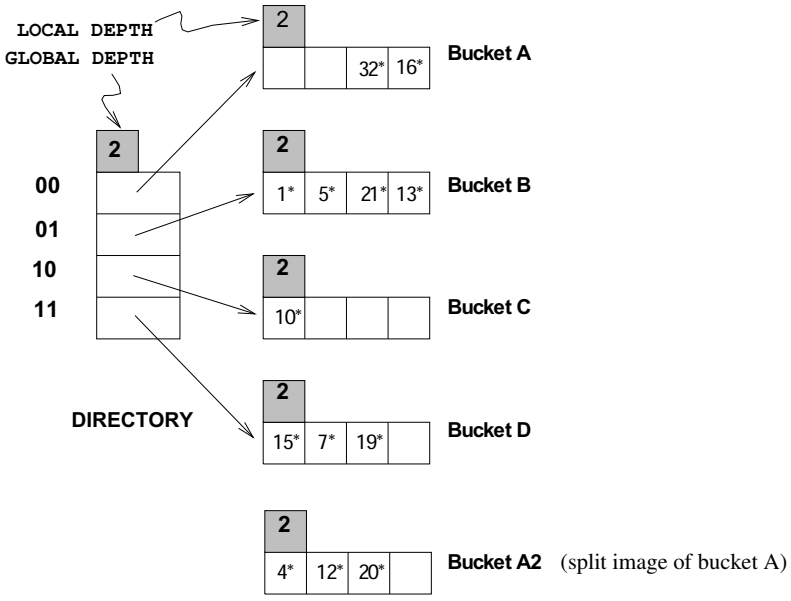

Figure 10.4 While Inserting Entry r with $h(r)$=20

Notice a problem that we must now resolve—we need three bits to discriminate between two of our data pages (A and A2), but the directory has only enough slots to store all two-bit patterns. The solution is to *double the directory*. Elements that differ only in the third bit from the end are said to 'correspond': *corresponding elements* of the directory point to the same bucket with the exception of the elements corresponding to the split bucket. In our example, bucket 0 was split; so, new directory element 000 points to one of the split versions and new element 100 points to the other. The sample file after completing all steps in the insertion of 20* is shown in Figure 10.5.

Thus, doubling the file requires allocating a new bucket page, writing both this page and the old bucket page that is being split, and doubling the directory array. The

[1]Since there are no overflow pages in Extendible Hashing, a bucket can be thought of as a single page.

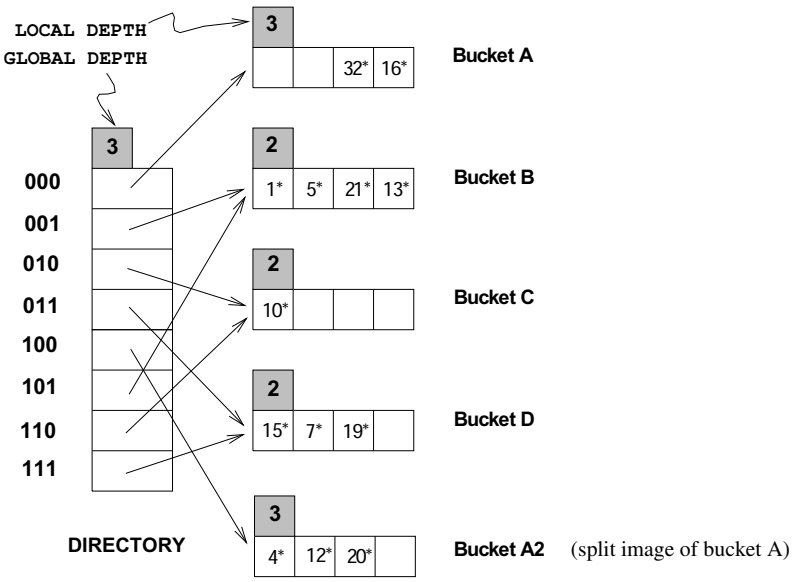

Figure 10.5 After Inserting Entry r with $h(r)$=20

directory is likely to be much smaller than the file itself because each element is just a page-id, and can be doubled by simply copying it over (and adjusting the elements for the split buckets). The cost of doubling is now quite acceptable.

We observe that the basic technique used in Extendible Hashing is to treat the result of applying a hash function h as a binary number and to interpret the last d bits, where d depends on the size of the directory, as an offset into the directory. In our example d is originally 2 because we only have four buckets; after the split, d becomes 3 because we now have eight buckets. A corollary is that when distributing entries across a bucket and its split image, we should do so on the basis of the dth bit. (Note how entries are redistributed in our example; see Figure 10.5.) The number d is called the **global depth** of the hashed file and is kept as part of the header of the file. It is used every time we need to locate a data entry.

An important point that arises is whether splitting a bucket necessitates a directory doubling. Consider our example, as shown in Figure 10.5. If we now insert 9*, it belongs in bucket B; this bucket is already full. We can deal with this situation by splitting the bucket and using directory elements 001 and 101 to point to the bucket and its split image, as shown in Figure 10.6.

Thus, a bucket split does not necessarily require a directory doubling. However, if either bucket A or A2 grows full and an insert then forces a bucket split, we are forced to double the directory again.

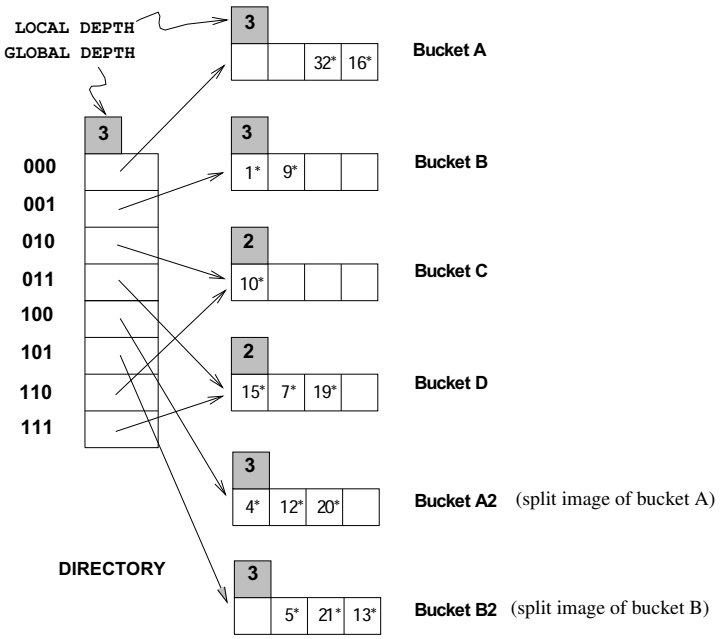

Figure 10.6 After Inserting Entry r with $h(r)$=9

In order to differentiate between these cases, and determine whether a directory doubling is needed, we maintain a **local depth** for each bucket. If a bucket whose local depth is equal to the global depth is split, the directory must be doubled. Going back to the example, when we inserted 9* into the index shown in Figure 10.5, it belonged to bucket B with local depth 2, whereas the global depth was 3. Even though the bucket was split, the directory did not have to be doubled. Buckets A and A2, on the other hand, have local depth equal to the global depth and, if they grow full and are split, the directory must then be doubled.

Initially, all local depths are equal to the global depth (which is the number of bits needed to express the total number of buckets). We increment the global depth by 1 each time the directory doubles, of course. Also, whenever a bucket is split (whether or not the split leads to a directory doubling), we increment by 1 the local depth of the split bucket and assign this same (incremented) local depth to its (newly created) split image. Intuitively, if a bucket has local depth l, the hash values of data entries in it agree upon the last l bits; further, no data entry in any other bucket of the file has a hash value with the same last l bits. A total of 2^{d-l} directory elements point to a bucket with local depth l; if $d = l$, exactly one directory element is pointing to the bucket, and splitting such a bucket requires directory doubling.

A final point to note is that we can also use the first d bits (the *most significant* bits) instead of the last d (*least significant* bits), but in practice the *last* d bits are used. The reason is that a directory can then be doubled simply by copying it.

In summary, a data entry can be located by computing its hash value, taking the last d bits, and looking in the bucket pointed to by this directory element. For inserts, the data entry is placed in the bucket to which it belongs and the bucket is split if necessary to make space. A bucket split leads to an increase in the local depth, and if the local depth becomes greater than the global depth as a result, to a directory doubling (and an increase in the global depth) as well.

For deletes, the data entry is located and removed. If the delete leaves the bucket empty, it can be merged with its split image, although this step is often omitted in practice. Merging buckets decreases the local depth. If each directory element points to the same bucket as its split image (i.e., 0 and 2^{d-1} point to the same bucket, namely A; 1 and $2^{d-1} + 1$ point to the same bucket, namely B, which may or may not be identical to A; etc.), we can halve the directory and reduce the global depth, although this step is not necessary for correctness.

The insertion examples can be worked out backwards as examples of deletion. (Start with the structure shown after an insertion and delete the inserted element. In each case the original structure should be the result.)

If the directory fits in memory, an equality selection can be answered in a single disk access, as for Static Hashing (in the absence of overflow pages), but otherwise, two disk I/Os are needed. As a typical example, a 100 MB file with 100 bytes per data entry and a page size of 4 KB contains 1,000,000 data entries and only about 25,000 elements in the directory. (Each page/bucket contains roughly 40 data entries, and we have one directory element per bucket.) Thus, although equality selections can be twice as slow as for Static Hashing files, chances are high that the directory will fit in memory and performance is the same as for Static Hashing files.

On the other hand, the directory grows in spurts and can become large for *skewed data distributions* (where our assumption that data pages contain roughly equal numbers of data entries is not valid). In the context of hashed files, a **skewed data distribution** is one in which the distribution of *hash values of search field values* (rather than the distribution of search field values themselves) is skewed (very 'bursty' or nonuniform). Even if the distribution of search values is skewed, the choice of a good hashing function typically yields a fairly uniform distribution of hash values; skew is therefore not a problem in practice.

Further, **collisions**, or data entries with the same hash value, cause a problem and must be handled specially: when more data entries than will fit on a page have the same hash value, we need overflow pages.