

11

HASH-BASED INDEXING

Exercise 11.1 Consider the Extendible Hashing index shown in Figure 11.1. Answer the following questions about this index:

1. What can you say about the last entry that was inserted into the index?
2. What can you say about the last entry that was inserted into the index if you know that there have been no deletions from this index so far?
3. Suppose you are told that there have been no deletions from this index so far. What can you say about the last entry whose insertion into the index caused a split?
4. Show the index after inserting an entry with hash value 68.

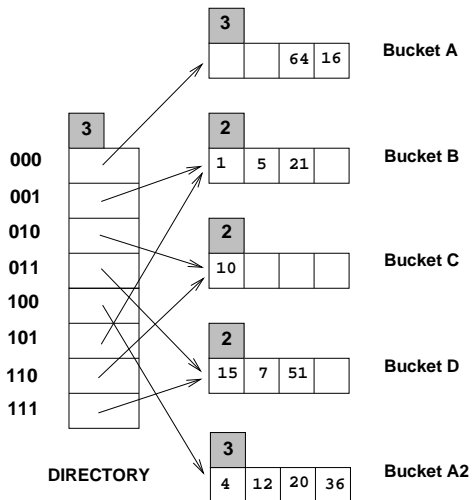


Figure 11.1 Figure for Exercise 11.1

5. Show the index after inserting entries with hash values 17 and 69 into the original tree.
6. Show the index after deleting the entry with hash value 21 into the original tree. (Assume that the full deletion algorithm is used.)
7. Show the index after deleting the entry with hash value 10 into the original tree. Is a merge triggered by this deletion? If not, explain why. (Assume that the full deletion algorithm is used.)

Answer 11.1 The answer to each question is given below.

1. It could be any one of the data entries in the index. We can always find a sequence of insertions and deletions with a particular key value, among the key values shown in the index as the last insertion. For example, consider the data entry 16 and the following sequence:
 1 5 21 10 15 7 51 4 12 36 64 8 24 56 16 56_D 24_D 8_D
 The last insertion is the data entry 16 and it also causes a split. But the sequence of deletions following this insertion cause a merge leading to the index structure shown in Fig 11.1.
2. The last insertion could not have caused a split because the total number of data entries in the buckets A and A_2 is 6. If the last entry caused a split the total would have been 5.
3. The last insertion which caused a split cannot be in bucket C. Buckets B and C or C and D could have made a possible bucket-split image combination but the total number of data entries in these combinations is 4 and the absence of deletions demands a sum of at least 5 data entries for such combinations. Buckets B and D can form a possible bucket-split image combination because they have a total of 6 data entries between themselves. So do A and A_2 . But for the B and D to be split images the starting global depth should have been 1. If the starting global depth is 2, then the last insertion causing a split would be in A or A_2 .
4. See Fig 11.2.
5. See Fig 11.3.
6. See Fig 11.4.
7. The deletion of the data entry 10 which is the only data entry in bucket C doesn't trigger a merge because bucket C is a primary page and it is left as a place holder. Right now, directory element 010 and its split image 110 already point to the same bucket C. We can't do a further merge.
 See Fig 11.5.

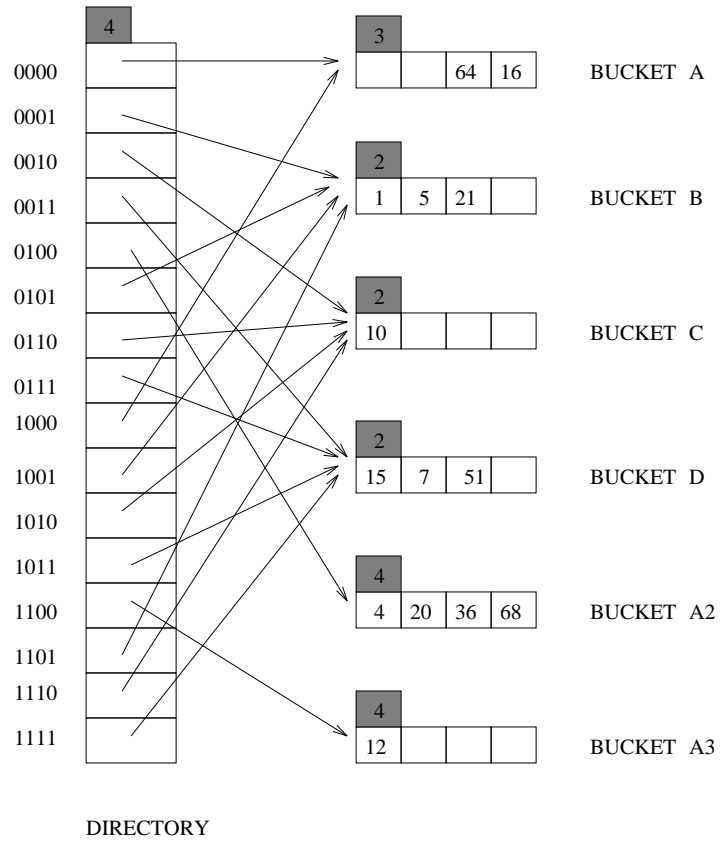


Figure 11.2

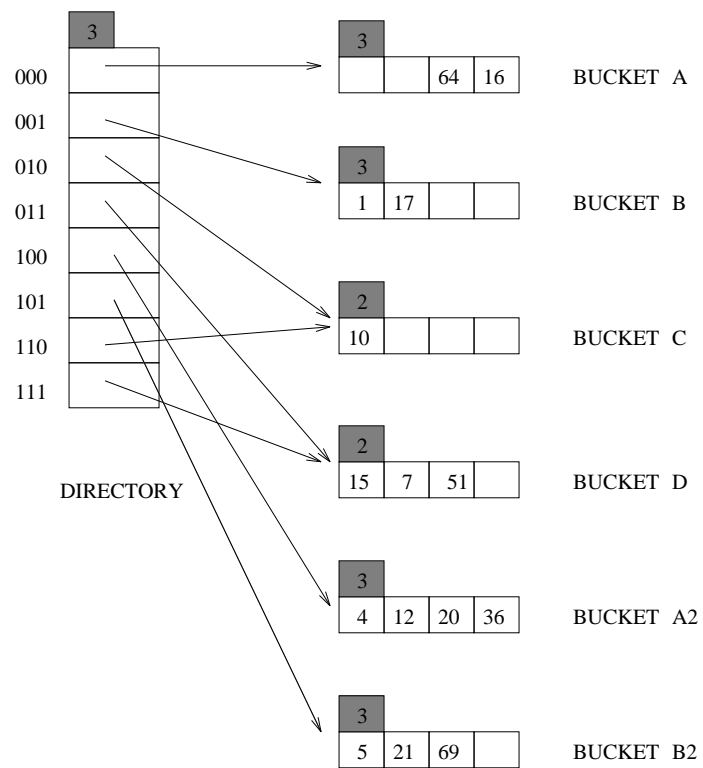


Figure 11.3

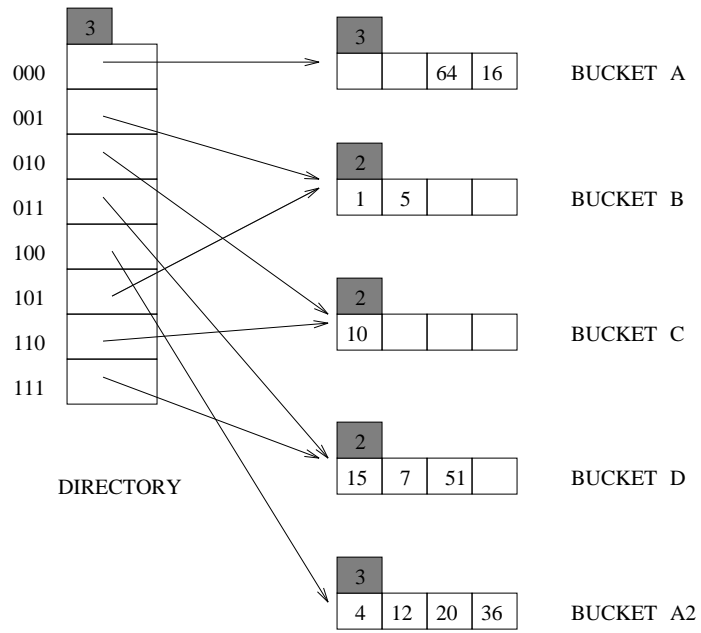


Figure 11.4

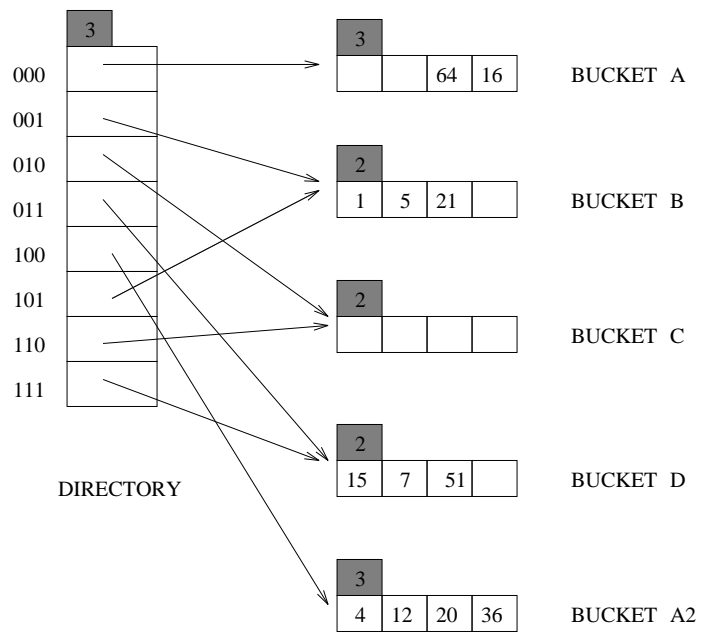


Figure 11.5

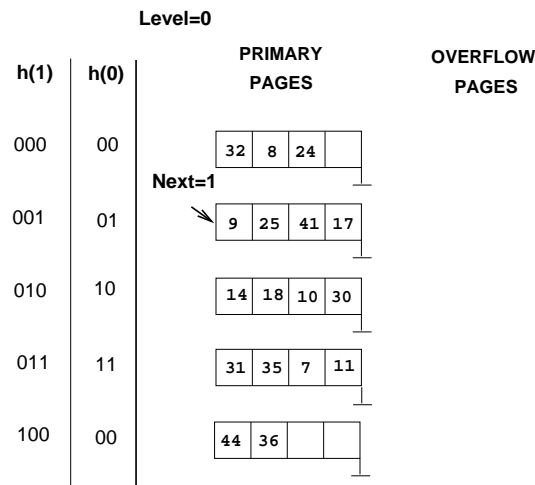


Figure 11.6 Figure for Exercise 11.2

Exercise 11.2 Consider the Linear Hashing index shown in Figure 11.6. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

1. What can you say about the last entry that was inserted into the index?
2. What can you say about the last entry that was inserted into the index if you know that there have been no deletions from this index so far?
3. Suppose you know that there have been no deletions from this index so far. What can you say about the last entry whose insertion into the index caused a split?
4. Show the index after inserting an entry with hash value 4.
5. Show the index after inserting an entry with hash value 15 into the original tree.
6. Show the index after deleting the entries with hash values 36 and 44 into the original tree. (Assume that the full deletion algorithm is used.)
7. Find a list of entries whose insertion into the original index would lead to a bucket with two overflow pages. Use as few entries as possible to accomplish this. What is the maximum number of entries that can be inserted into this bucket before a split occurs that reduces the length of this overflow chain?

Answer 11.2 Answer omitted.

Exercise 11.3 Answer the following questions about Extensible Hashing:

1. Explain why local depth and global depth are needed.
2. After an insertion that causes the directory size to double, how many buckets have exactly one directory entry pointing to them? If an entry is then deleted from one of these buckets, what happens to the directory size? Explain your answers briefly.
3. Does Extendible Hashing guarantee at most one disk access to retrieve a record with a given key value?
4. If the hash function distributes data entries over the space of bucket numbers in a very skewed (non-uniform) way, what can you say about the size of the directory? What can you say about the space utilization in data pages (i.e., non-directory pages)?
5. Does doubling the directory require us to examine all buckets with local depth equal to global depth?
6. Why is handling duplicate key values in Extendible Hashing harder than in ISAM?

Answer 11.3 The answer to each question is given below.

1. Extendible hashing allows the size of the directory to increase and decrease depending on the number and variety of inserts and deletes. Once the directory size changes, the hash function applied to the search key value should also change. So there should be some information in the index as to which hash function is to be applied. This information is provided by the *global depth*.

An increase in the directory size doesn't cause the creation of new buckets for each new directory entry. All the new directory entries except one share buckets with the old directory entries. Whenever a bucket which is being shared by two or more directory entries is to be split the directory size need not be doubled. This means for each bucket we need to know whether it is being shared by two or more directory entries. This information is provided by the *local depth* of the bucket. The same information can be obtained by a scan of the directory, but this is costlier.

2. Exactly two directory entries have only one directory entry pointing to them after a doubling of the directory size. This is because when the directory is doubled, one of the buckets must have split causing a directory entry to point to each of these two new buckets.

If an entry is then deleted from one of these buckets, a merge may occur, but this depends on the deletion algorithm. If we try to merge two buckets only when a bucket becomes empty, then it is not necessary that the directory size decrease after the deletion that was considered in the question. However, if we try to merge

two buckets whenever it is possible to do so then the directory size decreases after the deletion.

3. No "minimum disk access" guarantee is provided by extendible hashing. If the directory is not already in memory it needs to be fetched from the disk which may require more than one disk access depending upon the size of the directory. Then the required bucket has to be brought into the memory. Also, if alternatives 2 and 3 are followed for storing the data entries in the index then another disk access is possibly required for fetching the actual data record.
4. Consider the index in Fig 11.1. Let us consider a list of data entries with search key values of the form 2^i where $i > k$. By an appropriate choice of k , we can get all these elements mapped into the *Bucket A*. This creates 2^k elements in the directory which point to just $k + 3$ different buckets. Also, we note there are k buckets (data pages), but just one bucket is used. So the utilization of data pages = $1/k$.
5. No. Since we are using extendible hashing, only the local depth of the bucket being split needs be examined.
6. Extendible hashing is not supposed to have overflow pages (overflow pages are supposed to be dealt with using redistribution and splitting). When there are many duplicate entries in the index, overflow pages may be created that can never be redistributed (they will always map to the same bucket). Whenever a "split" occurs on a bucket containing only duplicate entries, an empty bucket will be created since all of the duplicates remain in the same bucket. The overflow chains will never be split, which makes inserts and searches more costly.

Exercise 11.4 Answer the following questions about Linear Hashing:

1. How does Linear Hashing provide an average-case search cost of only slightly more than one disk I/O, given that overflow buckets are part of its data structure?
2. Does Linear Hashing guarantee at most one disk access to retrieve a record with a given key value?
3. If a Linear Hashing index using Alternative (1) for data entries contains N records, with P records per page and an average storage utilization of 80 percent, what is the worst-case cost for an equality search? Under what conditions would this cost be the actual search cost?
4. If the hash function distributes data entries over the space of bucket numbers in a very skewed (non-uniform) way, what can you say about the space utilization in data pages?

Answer 11.4 Answer omitted.

Exercise 11.5 Give an example of when you would use each element (A or B) for each of the following ‘A versus B’ pairs:

1. A hashed index using Alternative (1) versus heap file organization.
2. Extendible Hashing versus Linear Hashing.
3. Static Hashing versus Linear Hashing.
4. Static Hashing versus ISAM.
5. Linear Hashing versus B+ trees.

Answer 11.5 The answer to each question is given below.

1. **Example 1:** Consider a situation in which most of the queries are equality queries based on the search key field. It pays to build a hashed index on this field in which case we can get the required record in one or two disk accesses. A heap file organisation may require a full scan of the file to access a particular record.

Example 2: Consider a file on which only sequential scans are done. It may fare better if it is organised as a heap file. A hashed index built on it may require more disk accesses because the occupancy of the pages may not be 100%.

2. **Example 1:** Consider a set of data entries with search keys which lead to a skewed distribution of hash key values. In this case, extendible hashing causes splits of buckets at the necessary bucket whereas linear hashing goes about splitting buckets in a round-robin fashion which is useless. Here extendible hashing has a better occupancy and shorter overflow chains than linear hashing. So equality search is cheaper for extendible hashing.

Example 2: Consider a very large file which requires a directory spanning several pages. In this case extendible hashing requires $d + 1$ disk accesses for equality selections where d is the number of directory pages. Linear hashing is cheaper.

3. **Example 1:** Consider a situation in which the number of records in the file is constant. Let all the search key values be of the form $2^n + k$ for various values of n and a few values of k . The traditional hash functions used in *linear hashing* like taking the last d bits of the search key lead to a skewed distribution of the hash key values. This leads to long overflow chains. A static hashing index can use the hash function defined as

$$h(2^n + k) = n$$

A family of hash functions can't be built based on this hash function as k takes only a few values. In this case static hashing is better.

Example 2: Consider a situation in which the number of records in the file varies a lot and the hash key values have a uniform distribution. Here linear hashing is clearly better than static hashing which might lead to long overflow chains thus considerably increasing the cost of equality search.

4. **Example 1:** Consider a situation in which the number of records in the file is constant and only equality selections are performed. Static hashing requires one or two disk accesses to get to the data entry. ISAM may require more than one depending on the height of the ISAM tree.

Example 2: Consider a situation in which the search key values of data entries can be used to build a clustered index and most of the queries are range queries on this field. Then ISAM definitely wins over static hashing.

5. **Example 1:** Again consider a situation in which only equality selections are performed on the index. Linear hashing is better than B+ tree in this case.

Example 2: When an index which is clustered and most of the queries are range searches, B+ indexes are better.

Exercise 11.6 Give examples of the following:

1. A Linear Hashing index and an Extendible Hashing index with the same data entries, such that the Linear Hashing index has more pages.
2. A Linear Hashing index and an Extendible Hashing index with the same data entries, such that the Extendible Hashing index has more pages.

Answer 11.6 Answer omitted.

Exercise 11.7 Consider a relation $R(a, b, c, d)$ containing 1 million records, where each page of the relation holds 10 records. R is organized as a heap file with unclustered indexes, and the records in R are randomly ordered. Assume that attribute a is a candidate key for R , with values lying in the range 0 to 999,999. For each of the following queries, name the approach that would most likely require the fewest I/Os for processing the query. The approaches to consider follow:

- Scanning through the whole heap file for R .
- Using a B+ tree index on attribute $R.a$.
- Using a hash index on attribute $R.a$.

The queries are:

1. Find all R tuples.
2. Find all R tuples such that $a < 50$.
3. Find all R tuples such that $a = 50$.
4. Find all R tuples such that $a > 50$ and $a < 100$.

Answer 11.7 Let h be the height of the B+ tree (usually 2 or 3) and M be the number of data entries per page ($M > 10$). Let us assume that after accessing the data entry it takes one more disk access to get the actual record. Let c be the occupancy factor in hash indexing.

Consider the table shown below (disk accesses):

Problem	Heap File	B+ Tree	Hash Index
1. All tuples	10^5	$h + \frac{10^6}{M} + 10^6$	$\frac{10^6}{cM} + 10^6$
2. $a < 50$	10^5	$h + \frac{50}{M} + 50$	100
3. $a = 50$	10^5	$h + 1$	2
4. $a > 50$ and $a < 100$	10^5	$h + \frac{50}{M} + 49$	98

1. From the first row of the table, we see that heap file organization is the best (has the fewest disk accesses).
2. From the second row of the table, with typical values for h and M , the B+ Tree has the fewest disk accesses.
3. From the third row of the table, hash indexing is the best.
4. From the fourth row of the table, again we see that B+ Tree is the best.

Exercise 11.8 How would your answers to Exercise 11.7 change if a is not a candidate key for R? How would they change if we assume that records in R are sorted on a ?

Answer 11.8 Answer omitted.

Exercise 11.9 Consider the snapshot of the Linear Hashing index shown in Figure 11.12. Assume that a bucket split occurs whenever an overflow page is created.

1. What is the *maximum* number of data entries that can be inserted (given the best possible distribution of keys) before you have to split a bucket? Explain very briefly.
2. Show the file after inserting a *single* record whose insertion causes a bucket split.
3. (a) What is the *minimum* number of record insertions that will cause a split of all four buckets? Explain very briefly.
 (b) What is the value of *Next* after making these insertions?
 (c) What can you say about the number of pages in the fourth bucket shown after this series of record insertions?

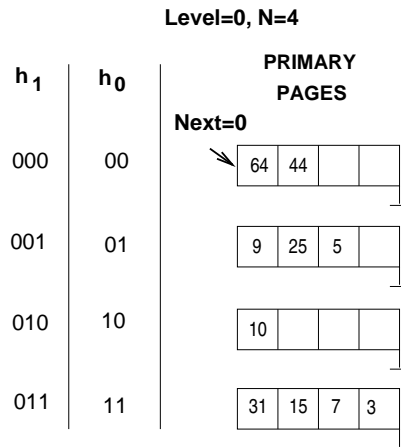


Figure 11.7 Figure for Exercise 11.9

Answer 11.9 The answer to each question is given below.

1. The maximum number of entries that can be inserted without causing a split is 6 because there is space for a total of 6 records in all the pages. A split is caused whenever an entry is inserted into a full page.
2. See Fig 11.13
3. (a) Consider the list of insertions 63, 41, 73, 137 followed by 4 more entries which go into the same bucket, say 18, 34, 66, 130 which go into the 3rd bucket. The insertion of 63 causes the first bucket to be split. Insertion of 41, 73 causes the second bucket split leaving a full second bucket. Inserting 137 into it causes third bucket-split. At this point at least 4 more entries are required to split the fourth bucket. A minimum of 8 entries are required to cause the 4 splits.
 - (b) Since all four buckets would have been split, that particular round comes to an end and the next round begins. So $Next = 0$ again.
 - (c) There can be either one data page or two data pages in the fourth bucket after these insertions. If the 4 more elements inserted into the 2^{nd} bucket after 3^{rd} bucket-splitting, then 4^{th} bucket has 1 data page. If the new 4 more elements inserted into the 4^{th} bucket after 3^{rd} bucket-splitting and all of them have 011 as its last three bits, then 4^{th} bucket has 2 data pages. Otherwise, if not all have 011 as its last three bits, then the 4^{th} bucket has 1 data page.

Exercise 11.10 Consider the data entries in the Linear Hashing index for Exercise 11.9.

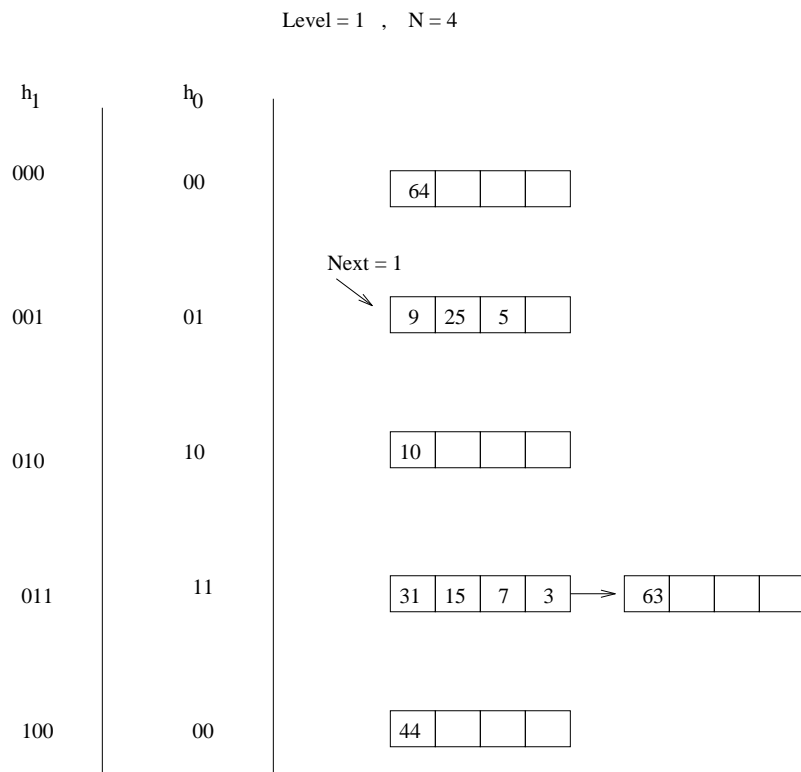


Figure 11.8

1. Show an Extendible Hashing index with the same data entries.
2. Answer the questions in Exercise 11.9 with respect to this index.

Answer 11.10 Answer omitted.

Exercise 11.11 In answering the following questions, assume that the full deletion algorithm is used. Assume that merging is done when a bucket becomes empty.

1. Give an example of Extendible Hashing where deleting an entry reduces global depth.
2. Give an example of Linear Hashing in which deleting an entry decrements *Next* but leaves *Level* unchanged. Show the file before and after the deletion.
3. Give an example of Linear Hashing in which deleting an entry decrements *Level*. Show the file before and after the deletion.
4. Give an example of Extendible Hashing and a list of entries e_1, e_2, e_3 such that inserting the entries in order leads to three splits and deleting them in the reverse order yields the original index. If such an example does not exist, explain.
5. Give an example of a Linear Hashing index and a list of entries e_1, e_2, e_3 such that inserting the entries in order leads to three splits and deleting them in the reverse order yields the original index. If such an example does not exist, explain.

Answer 11.11 The answers are as follows.

1. See Fig 11.16
2. See Fig 11.17
3. See Fig 11.18
4. Let us take the transition shown in Fig 11.19. Here we insert the data entries 4, 5 and 7. Each one of these insertions causes a split with the initial split also causing a directory split. But none of these insertions redistribute the already existing data entries into the new buckets. So when we delete these data entries in the reverse order (actually the order doesn't matter) and follow the full deletion algorithm we get back the original index.
5. This example is shown in Fig 11.20. Here the idea is similar to that used in the previous answer except that the bucket being split is the one into which the insertion being made. So bucket 2 has to be split and not bucket 3. Also the order of deletions should be exactly reversed because in the deletion algorithm *Next* is decremented only if the last bucket becomes empty.

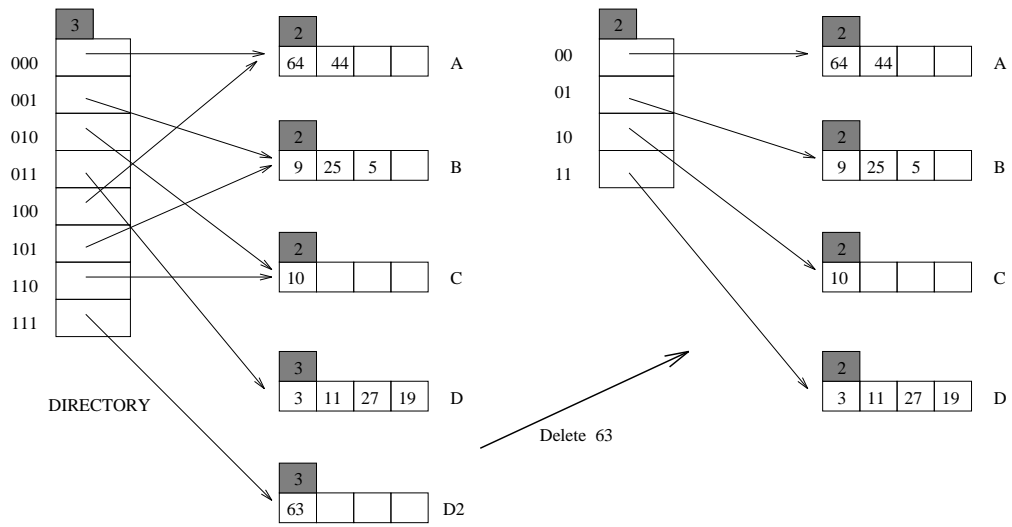


Figure 11.9

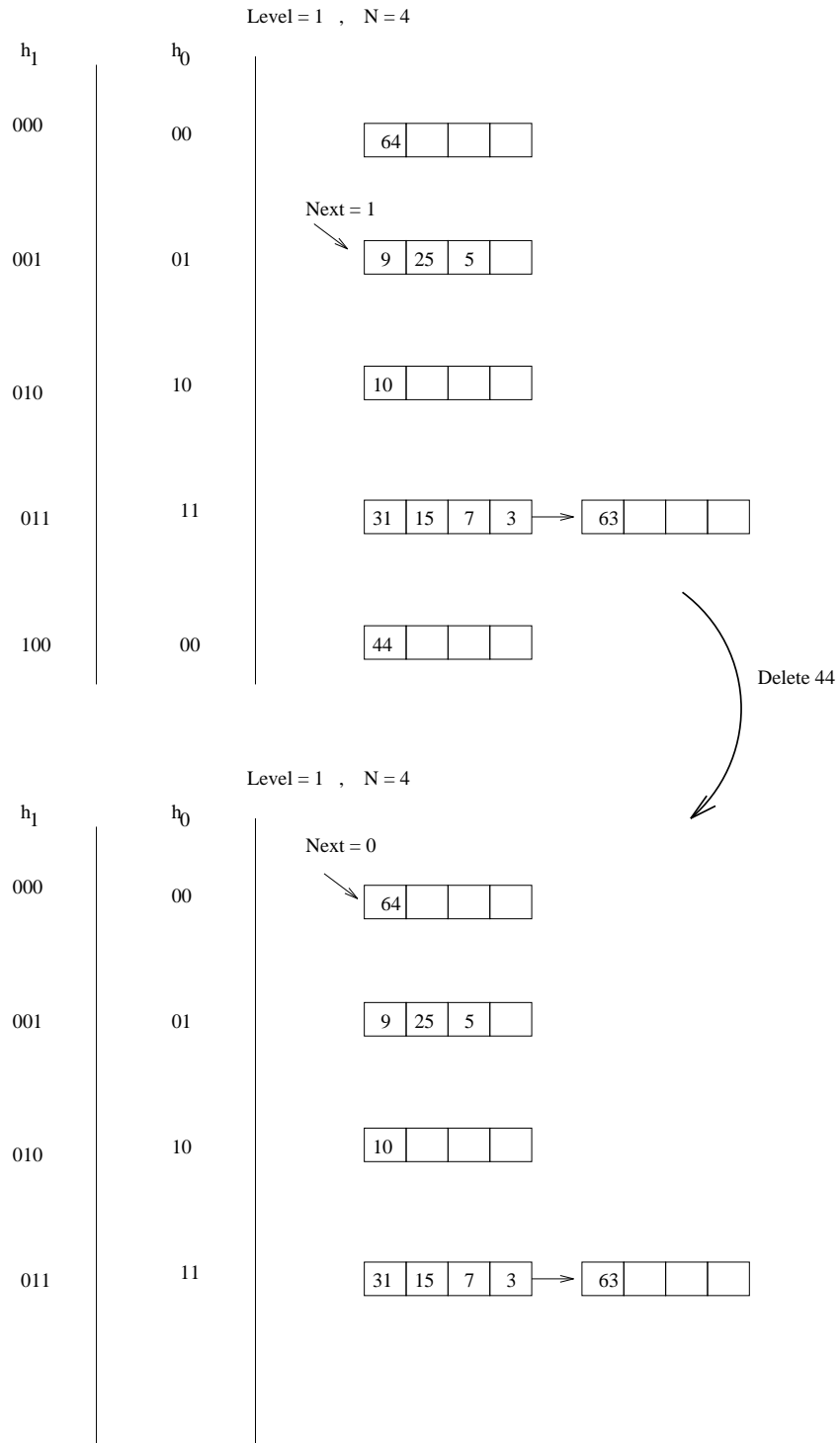


Figure 11.10

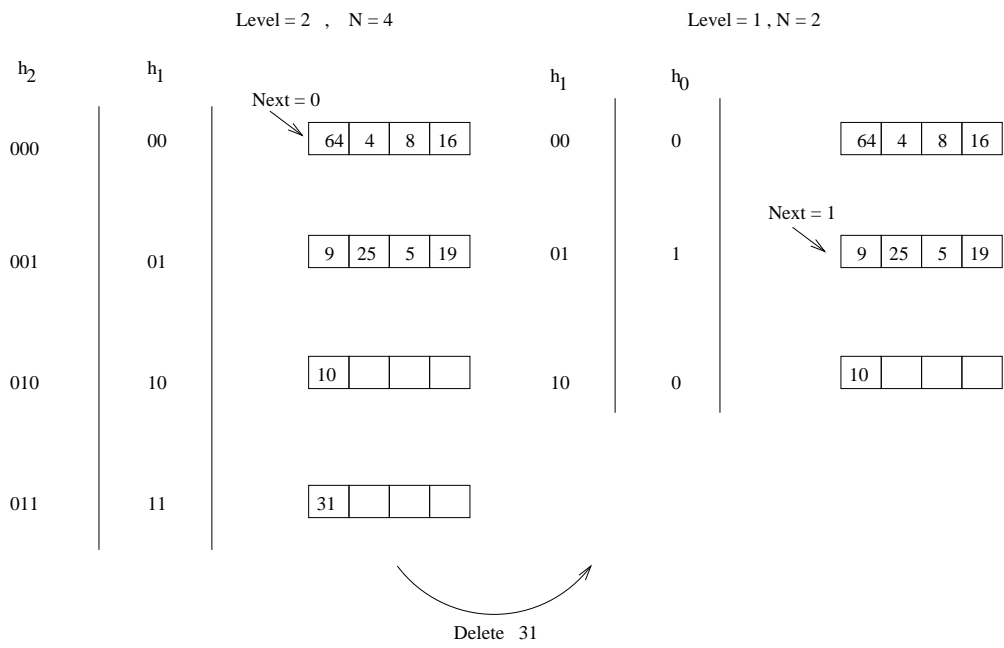


Figure 11.11

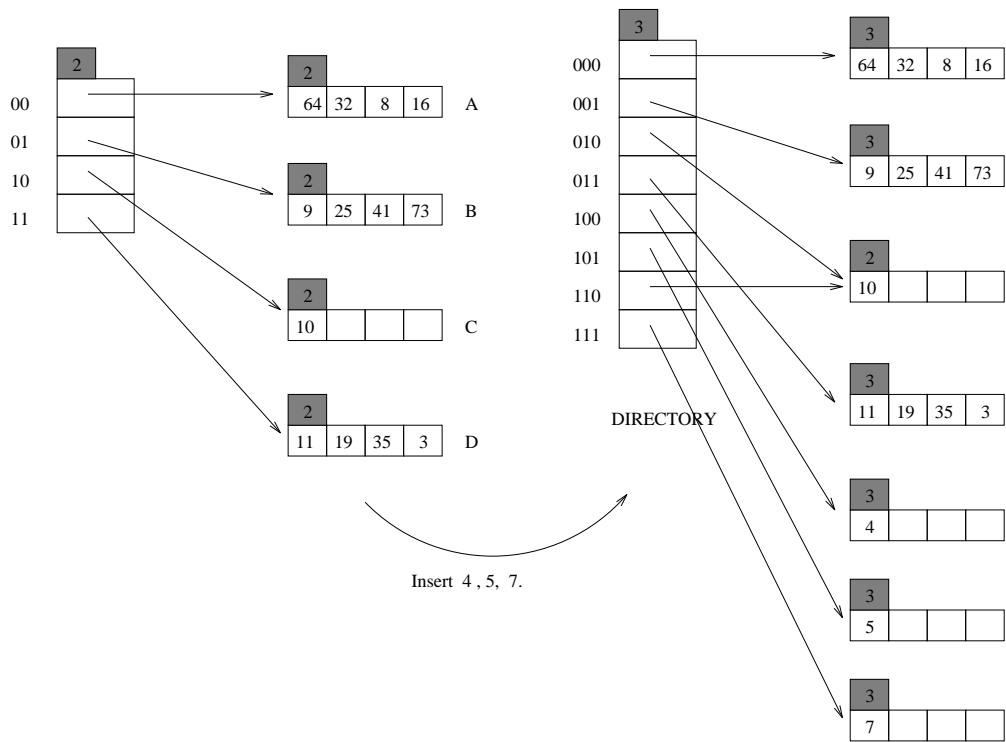


Figure 11.12

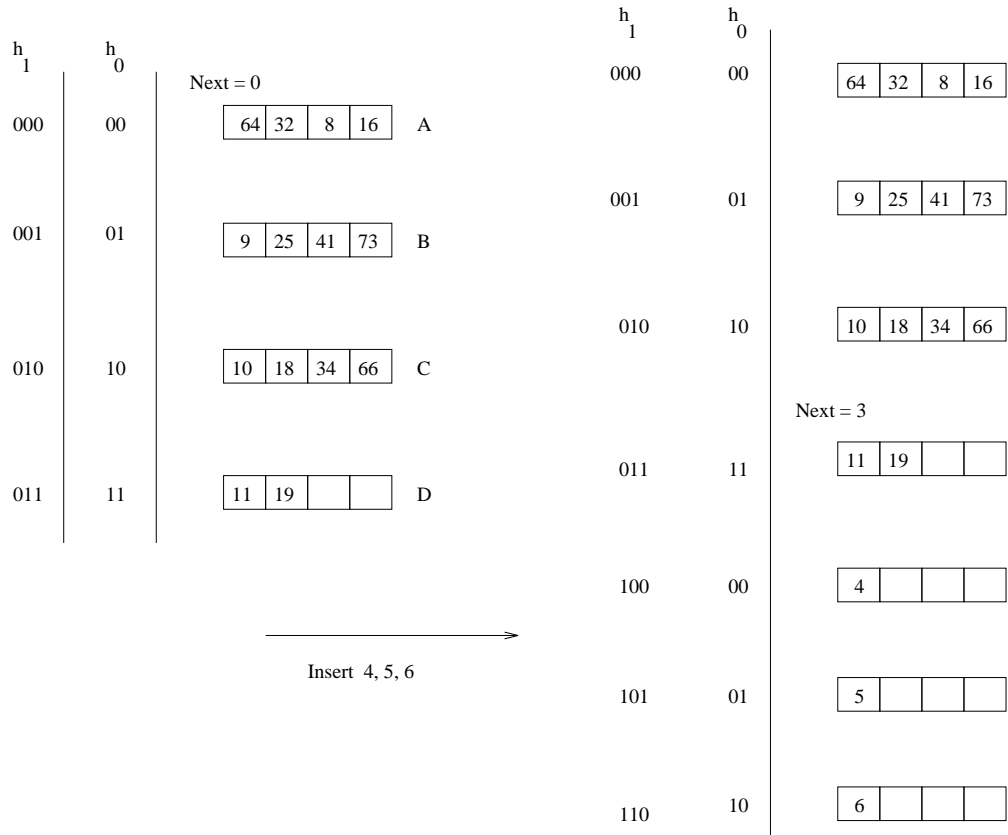


Figure 11.13