Exercise 10.8 Assume that you have just built a dense B+ tree index using Alternative (2) on a heap file containing 20,000 records. The key field for this B+ tree index is a 40-byte string, and it is a candidate key. Pointers (i.e., record ids and page ids) are (at most) 10-byte values. The size of one disk page is 1000 bytes. The index was built in a bottom-up fashion using the bulk-loading algorithm, and the nodes at each level were filled up as much as possible.

1. How many levels does the resulting tree have?
2. For each level of the tree, how many nodes are at that level?
3. How many levels would the resulting tree have if key compression is used and it reduces the average size of each key in an entry to 10 bytes?
4. How many levels would the resulting tree have without key compression but with all pages 70 percent full?

Answer 10.8 The answer to each question is given below.

1. Since the index is a primary dense index, there are as many data entries in the B+ tree as records in the heap file. An index page consists of at most $2d$ keys and $2d+1$ pointers. So we have to maximize d under the condition that $2d \cdot 40 + (2d+1) \cdot 10 \leq 1000$. The solution is $d = 9$, which means that we can have 18 keys and 19 pointers on an index page. A record on a leaf page consists of the key field and a pointer. Its size is 40+10=50 bytes. Therefore a leaf page has space for (1000/50)=20 data entries. The resulting tree has $\lceil \log_{19}(20000/20) + 1 \rceil = 4$ levels.
2. Since the nodes at each level are filled as much as possible, there are $\lceil 20000/20 \rceil =$ 1000 leaf nodes (on level 4). (A full index node has $2d+1 = 19$ children.) Therefore there are $\lceil 1000/19 \rceil = 53$ index pages on level 3, $\lceil 53/19 \rceil = 3$ index pages on level 2, and there is one index page on level 1 (the root of the tree).
3. Here the solution is similar to part 1, except the key is of size 10 instead of size 40. An index page consists of at most $2d$ keys and $2d + 1$ pointers. So we have to maximize d under the condition that $2d \cdot 10 + (2d + 1) \cdot 10 \leq 1000$. The solution is $d = 24$, which means that we can have 48 keys and 49 pointers on an index page. A record on a leaf page consists of the key field and a pointer. Its size is 10+10=20 bytes. Therefore a leaf page has space for (1000/20)=50 data entries. The resulting tree has $\lceil \log_{49}(20000/50) + 1 \rceil = 3$ levels.
4. Since each page should be filled only 70 percent, this means that the usable size of a page is $1000 \cdot 0.70 = 700$ bytes. Now the calculation is the same as in part 1 but using pages of size 700 instead of size 1000. An index page consists of at most $2d$ keys and $2d + 1$ pointers. So we have to maximize d under the condition that $2d \cdot 40 + (2d+1) \cdot 10 \leq 700$. The solution is $d = 6$, which means that we can have 12 keys and 13 pointers on an index page. A record on a leaf page consists of the key

field and a pointer. Its size is 40+10=50 bytes. Therefore a leaf page has space for (700/50)=14 data entries. The resulting tree has $\lceil \log_{13}(20000/14) + 1 \rceil = 4$ levels.

Exercise 10.9 The algorithms for insertion and deletion into a B+ tree are presented as recursive algorithms. In the code for *insert*, for instance, a call is made at the parent of a node N to insert into (the subtree rooted at) node N, and when this call returns, the current node is the parent of N. Thus, we do not maintain any 'parent pointers' in nodes of B+ tree. Such pointers are not part of the B+ tree structure for a good reason, as this exercise demonstrates. An alternative approach that uses parent pointers—again, remember that such pointers are *not* part of the standard B+ tree structure!—in each node appears to be simpler:

> Search to the appropriate leaf using the search algorithm; then insert the entry and split if necessary, with splits propagated to parents if necessary (using the parent pointers to find the parents).

Consider this (unsatisfactory) alternative approach:

1. Suppose that an internal node N is split into nodes N and $N2$. What can you say about the parent pointers in the children of the original node N?
2. Suggest two ways of dealing with the inconsistent parent pointers in the children of node N.
3. For each of these suggestions, identify a potential (major) disadvantage.
4. What conclusions can you draw from this exercise?

Answer 10.9 The answer to each question is given below.

1. The parent pointers in either d or $d+1$ of the children of the original node N are not valid any more: they still point to N, but they should point to $N2$.
2. One solution is to adjust all parent pointers in the children of the original node N which became children of $N2$. Another solution is to leave the pointers during the insert operation and to adjust them later when the page is actually needed and read into memory anyway.
3. The first solution requires at least $d+1$ additional page reads (and sometime later, page writes) on an insert, which would result in a remarkable slowdown. In the second solution mentioned above, a child M, which has a parent pointer to be adjusted, is updated if an operation is performed which actually reads M into

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 1. Let us take the data entries

$$8, 16, 24, 32, 40, 48, 56, 64, 128, 7, 15, 31, 63, 127, 1, 10, 4$$

and the indexes shown in Fig 11.10 and Fig 11.11. Extendible hashing uses 9 pages including the directory page(assuming it spans just one page) and linear hashing uses 10 pages.

2. Consider the list of data entries

$$0, 4, 1, 5, 2, 6, 3, 7$$

and the usual hash functions for both and a page capacity of 4 records per page. Extendible hashing takes 4 data pages and also a directory page whereas linear hashing takes just 4 pages.

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:

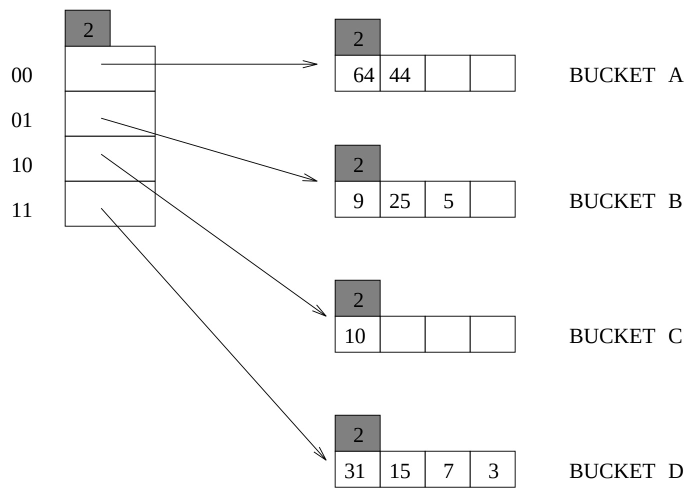

Figure 11.14

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-spliting 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.

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 An Extendible Hashing index with the same entries as Exercise 11.9 can be seen in Fig 11.14.

1. Six entries, for the same reason as in question 11.9.
2. See Fig 11.15

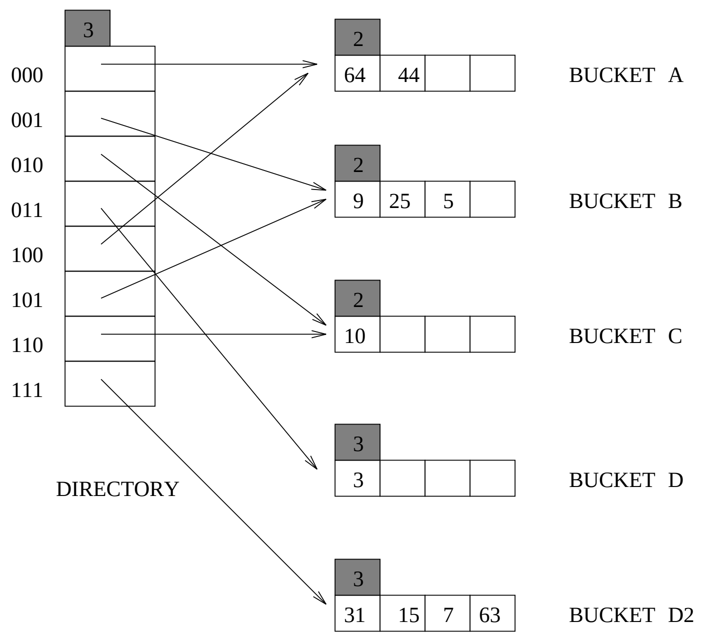

Figure 11.15

3. (a) 10. A bucket is split in extendible hashing only if it is full and a new entry is to be inserted into it.

   (b) The next pointer is not applicable in Extendible Hashing.

   (c) 1 page. Extendible hashing is not supposed to have overflow pages.

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.

(b) Create 256 'input' buffers of 1 page each, create an 'output' buffer of 64 pages, and do 256-way merges.

(c) Create 16 'input' buffers of 16 pages each, create an 'output' buffer of 64 pages, and do 16-way merges.

(d) Create eight 'input' buffers of 32 pages each, create an 'output' buffer of 64 pages, and do eight-way merges.

(e) Create four 'input' buffers of 64 pages each, create an 'output' buffer of 64 pages, and do four-way merges.

Answer 13.4 In Pass 0, 31250 sorted runs of 320 pages each are created. For each run, we read and write 320 pages sequentially. The I/O cost per run is $2*(10+5+1*320) = 670$ms. Thus, the I/O cost for Pass 0 is $31250*670 = 20937500$ms. For each of the cases discussed below, this cost must be added to the cost of the subsequent merging passes to get the total cost. Also, the calculations below are slightly simplified by neglecting the effect of a final read/written block that is slightly smaller than the earlier blocks.

1. For 319-way merges, only 2 more passes are needed. The first pass will produce

$$\lceil 31250/319 \rceil = 98$$

   sorted runs; these can then be merged in the next pass. Every page is read and written individually, at a cost of 16ms per read or write, in each of these two passes. The cost of these merging passes is therefore $2*(2*16)*10000000 = 640000000$ms. (The formula can be read as 'number of passes times cost of read and write per page times number of pages in file'.)

2. With 256-way merges, only two additional merging passes are needed. Every page in the file is read and written in each pass, but the effect of blocking is different on reads and writes. For reading, each page is read individually at a cost of 16ms. Thus, the cost of reads (over both passes) is $2*16*10000000 = 320000000$ms. For writing, pages are written out in blocks of 64 pages. The I/O cost per block is $10+5+1*64 = 79$ms. The number of blocks written out per pass is $10000000/64 = 156250$, and the cost per pass is $156250*79 = 12343750$ms. The cost of writes over both merging passes is therefore $2*12343750 = 24687500$ms. The total cost of reads and writes for the two merging passes is $320000000 + 24687500 = 344687500$ms.

3. With 16-way merges, 4 additional merging passes are needed. For reading, pages are read in blocks of 16 pages, at a cost per block of $10+5+1*16 = 31$ms. In each pass, $10000000/16 = 625000$ blocks are read. The cost of reading over the 4 merging passes is therefore $4*625000*31 = 77500000$ms. For writing, pages are written in 64 page blocks, and the cost per pass is 12343750ms as before. The cost of writes over 4 merging passes is $4*12343750 = 49375000$ms, and the total cost of the merging passes is $77500000+49375000 = 126875000$ms.

4. With 8-way merges, 5 merging passes are needed. For reading, pages are read in blocks of 32 pages, at a cost per block of $10 + 5 + 1 * 32 = 47$ms. In each pass, $10000000/32 = 312500$ blocks are read. The cost of reading over the 5 merging passes is therefore $5 * 312500 * 47 = 73437500$ms. For writing, pages are written in 64 page blocks, and the cost per pass is 12343750ms as before. The cost of writes over 5 merging passes is $5 * 12343750 = 61718750$ms, and the total cost of the merging passes is $73437500 + 61718750 = 135156250$ms.

5. With 4-way merges, 8 merging passes are needed. For reading, pages are read in blocks of 64 pages, at a cost per block of $10 + 5 + 1 * 64 = 79$ms. In each pass, $10000000/64 = 156250$ blocks are read. The cost of reading over the 8 merging passes is therefore $8 * 156250 * 79 = 98750000$ms. For writing, pages are written in 64 page blocks, and the cost per pass is 12343750ms as before. The cost of writes over 8 merging passes is $8 * 12343750 = 98750000$ms, and the total cost of the merging passes is $98750000 + 98750000 = 197500000$ms.

There are several lessons to be drawn from this (rather tedious) exercise. First, the cost of the merging phase varies from a low of 126875000ms to a high of 640000000ms. Second, the highest cost is associated with the option of maximizing fanout, choosing a buffer size of 1 page! Thus, the effect of blocked I/O is significant. However, as the block size is increased, the number of passes increases slowly, and there is a trade-off to be considered: it does not pay to increase block size indefinitely. Finally, while this example uses a different block size for reads and writes, for the sake of illustration, in practice a single block size is used for both reads and writes.

Exercise 13.5 Consider the refinement to the external sort algorithm that produces runs of length $2B$ on average, where B is the number of buffer pages. This refinement was described in Section 11.2.1 under the assumption that all records are the same size. Explain why this assumption is required and extend the idea to cover the case of variable-length records.

Answer 13.5 The assumption that all records are of the same size is used when the algorithm moves the smallest entry with a key value large than k to the output buffer and replaces it with a value from the input buffer. This "replacement" will only work if the records of the same size.

If the entries are of variable size, then we must also keep track of the size of each entry, and replace the moved entry with a new entry that fits in the available memory location. Dynamic programming algorithms have been adapted to decide an optimal replacement strategy in these cases.