6. 11, ⟨ *1,1* ⟩ 19, ⟨ *2,1* ⟩, ⟨ *2,2* ⟩ The order of entries is significant. Since the order of the entries is the same as the order of data record.
7. 1.8 2.0 3.2 3.4 3.8 The order of entries is not significant.
8. 1.8,⟨ *1,1* ⟩ 2.0,⟨ *1,2* ⟩ 3.2,⟨ *2,1* ⟩ 3.4,⟨ *1,3* ⟩ 3.8,⟨ *2,2* ⟩
   The order of entries is not significant.
9. Same as above
10. It can not be used to build a sparse index on *gpa*. Because Alternative (1) for data entries always leads to a dense index.
11. Neither alternative (2) nor (3) can be used to build a sparse index on *gpa*. Because sparse index has to be clustered, and the key values *gpa* are not in order.

For the dense index, the order of entries is not significant. Because there is one data entry in the index per record in the data file. However, for the sparse index, the order of entries is significent. There is only one entry for each page of records in the data file. So, the order of data entries in the index must corresponds to the order of records in the data file.

Exercise 8.5 Explain the difference between Hash indexes and B+-tree indexes. In particular, discuss how equality and range searches work, using an example.

Answer 8.5 Not yet available.

Exercise 8.6 Fill in the I/O costs in Figure 8.2.

File Type	*Scan*	*Equality Search*	*Range Search*	*Insert*	*Delete*
Heap file					
Sorted file					
Clustered file					
Unclustered tree index					
Unclustered hash index					

Figure 8.2 I/O Cost Comparison

Answer 8.6 Not yet available.

If no record qualifies, in a heap file, we have to search the entire file. So the cost is $B(D + RC)$. In a sorted file, even if no record qualifies, we have to do equality search to verify that no qualifying record exists. So the cost is the same as equality search, $Dlog_2B + Clog_2R$. In a hashed file, if no record qualifies, assuming no overflow page, we compute the hash value to find the bucket that would contain such a record (cost is H), bring that page in (cost is D), and search the entire page to verify that the record is not there (cost is RC). So the total cost is $H + D + RC$.

In all three file organizations, if the condition is not on the search key we have to search the entire file. There is an additional cost of C for each record that is deleted, and an additional D for each page containing such a record.

Exercise 8.9 What main conclusions can you draw from the discussion of the five basic file organizations discussed in Section **??**? Which of the five organizations would you choose for a file where the most frequent operations are as follows?

1. Search for records based on a range of field values.
2. Perform inserts and scans, where the order of records does not matter.
3. Search for a record based on a particular field value.

Answer 8.9 The main conclusion about the five file organizations is that all five have their own advantages and disadvantages. No one file organization is uniformly superior in all situations. The choice of appropriate structures for a given data set can have a significant impact upon performance. An unordered file is best if only full file scans are desired. A hash indexed file is best if the most common operation is an equality selection. A sorted file is best if range selections are desired and the data is static; a clustered B+ tree is best if range selections are important and the data is dynamic. An unclustered B+ tree index is useful for selections over small ranges, especially if we need to cluster on another search key to support some common query.

1. Using these fields as the search key, we would choose a sorted file organization or a clustered B+ tree depending on whether the data is static or not.
2. Heap file would be the best fit in this situation.
3. Using this particular field as the search key, choosing a hash indexed file would be the best.

Exercise 8.10 Consider the following relation:

Emp(*eid:* `integer`, *sal:* `integer`, *age:* `real`, *did:* `integer`)

Exercise 9.6 Consider again the disk specifications from Exercise 9.5, and suppose that a block size of 1024 bytes is chosen. Suppose that a file containing 100,000 records of 100 bytes each is to be stored on such a disk and that no record is allowed to span two blocks.

1. How many records fit onto a block?
2. How many blocks are required to store the entire file? If the file is arranged sequentially on the disk, how many surfaces are needed?
3. How many records of 100 bytes each can be stored using this disk?
4. If pages are stored sequentially on disk, with page 1 on block 1 of track 1, what page is stored on block 1 of track 1 on the next disk surface? How would your answer change if the disk were capable of reading and writing from all heads in parallel?
5. What time is required to read a file containing 100,000 records of 100 bytes each sequentially? Again, how would your answer change if the disk were capable of reading/writing from all heads in parallel (and the data was arranged optimally)?
6. What is the time required to read a file containing 100,000 records of 100 bytes each in a random order? To read a record, the block containing the record has to be fetched from disk. Assume that each block request incurs the average seek time and rotational delay.

Answer 9.6 1. $1024/100 = 10$. We can have at most 10 records in a block.

2. There are 100,000 records all together, and each block holds 10 records. Thus, we need 10,000 blocks to store the file. One track has 25 blocks, one cylinder has 250 blocks. we need 10,000 blocks to store this file. So we will use more than one cylinders, that is, need 10 surfaces to store this file.
3. The capacity of the disk is 500,000K, which has 500,000 blocks. Each block has 10 records. Therefore, the disk can store no more than 5,000,000 records.
4. There are 25K bytes, or we can say, 25 blocks in each track. It is block 26 on block 1 of track 1 on the next disk surface.

   If the disk were capable of reading/writing from all heads in parallel, we can put the first 10 pages on the block 1 of track 1 of all 10 surfaces. Therefore, it is block 2 on block 1 of track 1 on the next disk surface.
5. A file containing 100,000 records of 100 bytes needs 40 cylinders or 400 tracks in this disk. The transfer time of one track of data is 0.011 seconds. Then it takes $400 \times 0.011 = 4.4 seconds$ to transfer 400 tracks.

   This access seeks the track 40 times. The seek time is $40 \times 0.01 = 0.4 seconds$. Therefore, total access time is $4.4 + 0.4 = 4.8 seconds$.

If the disk were capable of reading/writing from all heads in parallel, the disk can read 10 tracks at a time. The transfer time is 10 times less, which is 0.44 seconds. Thus total access time is $0.44 + 0.4 = 0.84 seconds$

6. For any block of data, $averageaccesstime = seektime + rotationaldelay + transfertime$.

$$seektime = 10msec$$

$$rotationaldelay = 6msec$$

$$transfertime = \frac{1K}{2,250K/sec} = 0.44msec$$

The average access time for a block of data would be 16.44 msec. For a file containing 100,000 records of 100 bytes, the total access time would be 164.4 seconds.

Exercise 9.7 Explain what the buffer manager must do to process a read request for a page. What happens if the requested page is in the pool but not pinned?

Answer 9.7 When a page is requested the buffer manager does the following:

1. The buffer pool is checked to see if it contains the requested page. If the page is in the pool, skip to step 2. If the page is not in the pool, it is brought in as follows:
   (a) A frame is chosen for replacement, using the replacement policy.
   (b) If the frame chosen for replacement is dirty, it is *flushed* (the page it contains is written out to disk).
   (c) The requested page is read into the frame chosen for replacement.
2. The requested page is *pinned* (the *pin_count* of the chosen frame is incremented) and its address is returned to the requester.

Note that if the page is not pinned, it could be removed from buffer pool even if it is actually needed in main memory. *Pinning* a page prevents it from being removed from the pool.

Exercise 9.8 When does a buffer manager write a page to disk?

Answer 9.8 If a page in the buffer pool is chosen to be replaced and this page is dirty, the buffer manager must write the page to the disk. This is also called flushing the page to the disk.

Sometimes the buffer manager can also force a page to disk for recovery-related purposes (intuitively, to ensure that the log records corresponding to a modified page are written to disk before the modified page itself is written to disk).

2. If there are many queries running concurrently, the request of a page from different queries can be interleaved. In the worst case, it cause the cache miss on every page request, even with disk pre-fetching.
3. If we have pre-fetching offered by DBMS buffer manager, the buffer manager can predict the reference pattern more accurately. In particular, a certain number of buffer frames can be allocated *per* active scan for pre-fetching purposes, and interleaved requests would not compete for the same frames.
4. *Segmented caches* can work in a similar fashion to allocating buffer frames for each active scan (as in the above answer). This helps to solve some of the concurrency problem, but will not be useful at all if more files are being accessed than the number of segments. In this case, the DBMS buffer manager should still prefer to do pre-fetching on its own to handle a larger number of files, and to predict more complicated access patterns.

Exercise 9.16 Describe two possible record formats. What are the trade-offs between them?

Answer 9.16 Two possible record formats are: *fixed length records* and *variable length records*. (For details, see the text.)

Fixed length record format is easy to implement. Since the record size is fixed, records can be stored contiguously. Record address can be obtained very quickly.

Variable length record format is much more flexible.

Exercise 9.17 Describe two possible page formats. What are the trade-offs between them?

Answer 9.17 Two possible page formats are: *consecutive slots* and *slot directory*

The consecutive slots organization is mostly used for fixed length record formats. It handles the deletion by using bitmaps or linked lists.

The slot directory organization maintains a directory of slots for each page, with a $\langle$*record offset*, *record length*$\rangle$ pair per slot.

The slot directory is an indirect way to get the offset of an entry. Because of this indirection, deletion is easy. It is accomplished by setting the length field to 0. And records can easily be moved around on the page without changing their external identifier.

Exercise 9.18 Consider the page format for variable-length records that uses a slot directory.

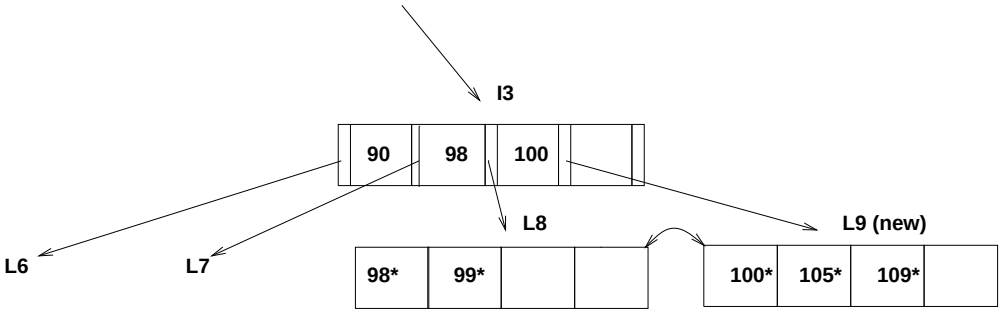

Figure 10.11

2. Show the B+ tree that would result from inserting a record with search key 109 into the tree.
3. Show the B+ tree that would result from deleting the record with search key 81 from the original tree.
4. Name a search key value such that inserting it into the (original) tree would cause an increase in the height of the tree.
5. Note that subtrees A, B, and C are not fully specified. Nonetheless, what can you infer about the contents and the shape of these trees?
6. How would your answers to the preceding questions change if this were an ISAM index?
7. Suppose that this is an ISAM index. What is the minimum number of insertions needed to create a chain of three overflow pages?

Answer 10.2 The answer to each question is given below.

1. I1, I2, and everything in the range [L2..L8].
2. See Figure 10.11. Notice that node L8 is split into two nodes.
3. Assuming that there is redistribution from the right sibling, the solution can be seen in Figure 10.12.
4. There are many search keys X such that inserting X would increase the height of the tree. Any search key in the range [65..79] would suffice. A key in this range would go in L5 if there were room for it, but since L5 is full already and since it can't redistribute any data entries over to L4 (L4 is full also), it must split; this in turn causes I2 to split, which causes I1 to split, and assuming I1 is the root node, a new root is created and the tree becomes taller.

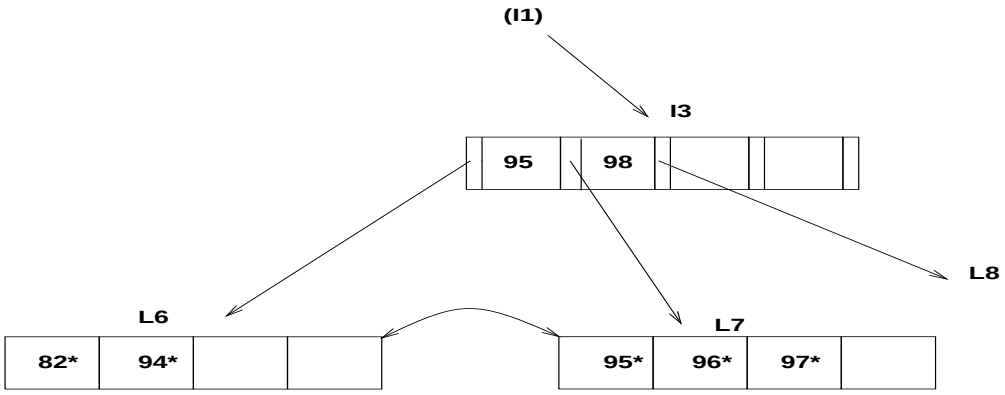

Figure 10.12

5. We can infer several things about subtrees A, B, and C. First of all, they each must have height one, since their "sibling" trees (those rooted at I2 and I3) have height one. Also, we know the ranges of these trees (assuming duplicates fit on the same leaf): subtree A holds search keys less than 10, B contains keys ≥ 10 and < 20, and C has keys ≥ 20 and < 30. In addition, each intermediate node has at least 2 key values and 3 pointers.

6. The answers for the questions above would change as follows if we were dealing with ISAM trees instead of B+ trees.

   (a) This is only a search, so the answer is the same. (The tree structure is not modified.)

   (b) Because we can never split a node in ISAM, we must create an overflow page to hold inserted key 109.

   (c) Search key 81 would simply be erased from L6; no redistribution would occur (ISAM has no minimum occupation requirements).

   (d) Being a *static* tree structure, an ISAM tree will never change height in normal operation, so there are no search keys which when inserted will increase the tree's height. (If we inserted an X in [65..79] we would have to create an overflow page for L5.)

   (e) We can infer several things about subtrees A, B, and C. First of all, they each must have height one, since their "sibling" trees (those rooted at I2 and I3) have height one. Here we suppose that we create a balanced ISAM tree. Also, we know the ranges of these trees (assuming duplicates fit on the same leaf): subtree A holds search keys less than 10, B contains keys ≥ 10 and < 20, and C has keys ≥ 20 and < 30. In addition, each of A, B, and C contains five leaf nodes (which may be of arbitrary fullness), and these nodes are the first 15 consecutive pages prior to L1.

3. The argument in part 2 does not assume anything about the data entries to be inserted; it is valid if duplicates can be inserted as well. Therefore the solution does not change.

Exercise 10.6 Answer Exercise 10.5 assuming that the tree is an ISAM tree! (Some of the examples asked for may not exist—if so, explain briefly.)

Answer 10.6 The answer to each question is given below.

1. The answer to each part is given below
   (a) Since ISAM trees use overflow buckets, any series of five inserts and deletes will result in the same tree.
   (b) If the leaves are not sorted, there is no sequence of inserts and deletes that will change the overall structure of an ISAM index. This is because inserts will create overflow buckets, and these overflow buckets will be removed when the elements are deleted, giving the original tree.
2. The height of the tree does never change since an ISAM index is static. If a leaf page becomes full, an overflow page is allocated; if a leaf page becomes empty, it remains empty.
3. See part 2.

Exercise 10.7 Suppose that you have a sorted file and want to construct a dense primary B+ tree index on this file.

1. One way to accomplish this task is to scan the file, record by record, inserting each one using the B+ tree insertion procedure. What performance and storage utilization problems are there with this approach?
2. Explain how the bulk-loading algorithm described in the text improves upon this scheme.

Answer 10.7 1. This approach is likely to be quite expensive, since each entry requires us to start from the root and go down to the appropriate leaf page. Even though the index level pages are likely to stay in the buffer pool between successive requests, the overhead is still considerable. Also, according to the insertion algorithm, each time a node splits, the data entries are redistributed evenly to both nodes. This leads to a fixed page utilization of 50%

2. The bulk loading algorithm has good performance and space utilization compared with the repeated inserts approach. Since the B+ tree is grown from the bottom up, the bulk loading algorithm allows the administrator to pre-set the amount each index and data page should be filled. This allows good performance for future inserts, and supports some desired space utilization.

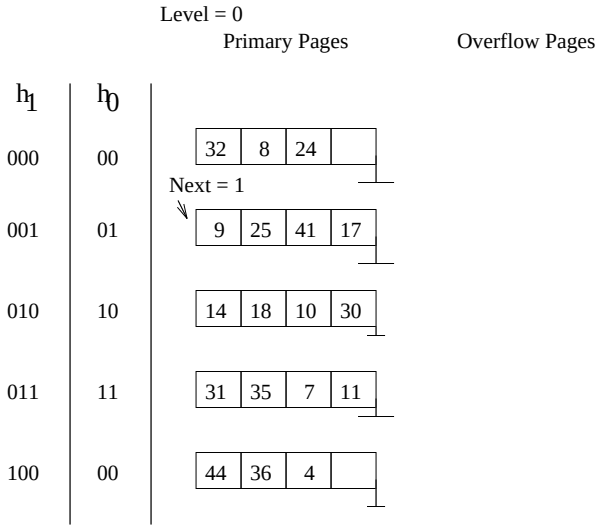

Figure 11.7 Index from Figure 11.6 after insertion of an entry with hash value 4

2. If the last item that was inserted had a hashcode $h_0(keyvalue) = 00$ then it caused a split, otherwise, any value could have been inserted.

3. The last data entry which caused a split satisfies the condition

$$h_0(keyvalue) = 00$$

   as there are no overflow pages for any of the other buckets.

4. See Fig 11.7

5. See Fig 11.8

6. See Fig 11.9

7. The following constitutes the minimum list of entries to cause two overflow pages in the index :

$$63, 127, 255, 511, 1023$$

   The first insertion causes a split and causes an update of $Next$ to 2. The insertion of 1023 causes a subsequent split and $Next$ is updated to 3 which points to this bucket.

   This overflow chain will not be redistributed until three more insertions (a total of 8 entries) are made. In principle if we choose data entries with key values of the form $2^k + 3$ with sufficiently large k, we can take the maximum number of entries that can be inserted to reduce the length of the overflow chain to be greater than

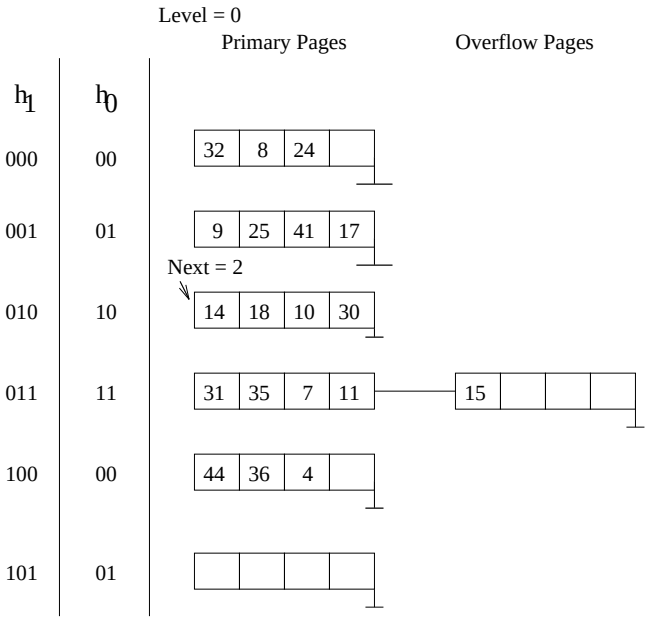

Figure 11.8 Index from Figure 11.6 after insertion of an entry with hash value 15

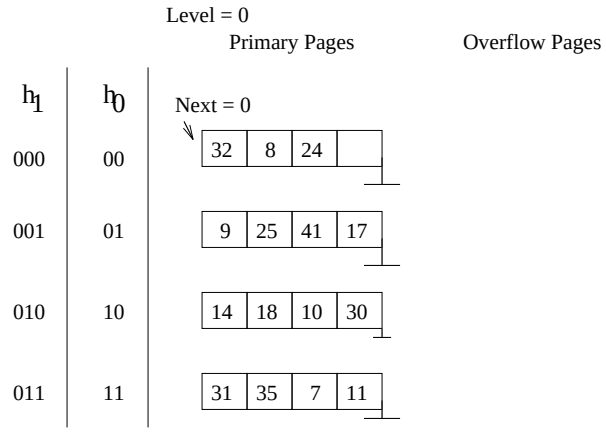

Figure 11.9 Index from Figure 11.6 after deletion of entries with hash values 36 and 44

any arbitrary number. This is so because the initial index has 31(binary 11111), 35(binary 10011),7(binary 111) and 11(binary 1011). So by an appropriate choice of data entries as mentioned above we can make a split of this bucket cause just two values (7 and 31) to be redistributed to the new bucket. By choosing a sufficiently large k we can delay the reduction of the length of the overflow chain till any number of splits of this bucket.

Exercise 11.3 Answer the following questions about Extendible 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.

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:

(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.