Exercise 8.3 Consider a relation stored as a randomly ordered file for which the only index is an unclustered index on a field called *sal*. If you want to retrieve all records with $sal > 20$, is using the index always the best alternative? Explain.

Answer 8.3 No. In this case, the index is unclustered, each qualifying data entry could contain an rid that points to a distinct data page, leading to as many data page I/Os as the number of data entries that match the range query. In this situation, using index is actually worse than file scan.

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 A Hash index is constructed by using a hashing function that quickly maps an search key value to a specific location in an array-like list of elements called buckets. The buckets are often constructed such that there are more bucket locations than there are possible search key values, and the hashing function is chosen so that it is not often that two search key values hash to the same bucket. A B+-tree index is constructed by sorting the data on the search key and maintaining a hierarchical search data structure that directs searches to the correct page of data entries.

Insertions and deletions in a hash based index are relatively simple. If two search values hash to the same bucket, called a collision, a linked list is formed connecting multiple records in a single bucket. In the case that too many of these collisions occur, the number of buckets is increased. Alternatively, maintaining a B+-tree's hierarchical search data structure is considered more costly since it must be updated whenever there

are insertions and deletions in the data set. In general, most insertions and deletions will not modify the data structure severely, but every once in awhile large portions of the tree may need to be rewritten when they become over-filled or under-filled with data entries.

Hash indexes are especially good at equality searches because they allow a record look up very quickly with an average cost of 1.2 I/Os. B+-tree indexes, on the other hand, have a cost of 3-4 I/Os per individual record lookup. Assume we have the employee relation with primary key *eid* and 10,000 records total. Looking up all the records individually would cost 12,000 I/Os for Hash indexes, but 30,000-40,000 I/Os for B+-tree indexes.

For range queries, hash indexes perform terribly since they could conceivably read as many pages as there are records since the data is not sorted in any clear grouping or set. On the other hand, B+-tree indexes have a cost of 3-4 I/Os plus the number of qualifying pages or tuples, for clustered or unclustered B+-trees respectively. Assume we have the employees example again with 10,000 records and 10 records per page. Also assume that there is an index on *sal* and query of *age* ¿ 20,000, such that there are 5,000 qualifying tuples. The hash index could cost as much as 100,000 I/Os since every page could be read for every record. It is not clear with a hash index how we even go about searching for every possible number greater than 20,000 since decimals could be used. An unclustered B+-tree index would have a cost of 5,004 I/Os, while a clustered B+-tree index would have a cost of 504 I/Os. It helps to have the index clustered whenever possible.

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 The answer to the question is given in Figure 8.3. We use B to denote the number of data pages total, R to denote the number of records per page, and D to denote the average time to read or write a page.

File Type	*Scan*	*Equality Search*	*Range Search*	*Insert*	*Delete*
Heap file	*BD*	*0.5BD*	*BD*	*2D*	*Search+* D
Sorted file	*BD*	$Dlog_2B$	$Dlog_2B$+# *matching pages*	*Search+* BD	*Search+* BD
Clustered file	*1.5BD*	$Dlog_F1.5B$	$Dlog_FB$+# *matching pages*	*Search+ D*	*Search+ D*
Unclustered tree index	*BD(R+ 0.15)*	*D(1+* $log_F0.15B$*)*	*D(*$log_F0.15B$+# *matching records)*	*D(3+* $log_F0.15B$*)*	*Search+ 2D*
Unclustered hash index	*BD(R+ 0.125)*	*2D*	*BD*	*4D*	*Search+ 2D*

Figure 8.3 I/O Cost Comparison

Exercise 8.7 If you were about to create an index on a relation, what considerations would guide your choice? Discuss:

1. The choice of primary index.
2. Clustered versus unclustered indexes.
3. Hash versus tree indexes.
4. The use of a sorted file rather than a tree-based index.
5. Choice of search key for the index. What is a composite search key, and what considerations are made in choosing composite search keys? What are index-only plans, and what is the influence of potential index-only evaluation plans on the choice of search key for an index?

Answer 8.7 The answer to each question is given below.

1. The choice of the primary key is made based on the semantics of the data. If we need to retrieve records based on the value of the primary key, as is likely, we should build an index using this as the search key. If we need to retrieve records based on the values of fields that do not constitute the primary key, we build (by definition) a secondary index using (the combination of) these fields as the search key.
2. A clustered index offers much better range query performance, but essentially the same equality search performance (modulo duplicates) as an unclustered index.

Further, a clustered index is typically more expensive to maintain than an unclustered index. Therefore, we should make an index be clustered only if range queries are important on its search key. At most one of the indexes on a relation can be clustered, and if range queries are anticipated on more than one combination of fields, we have to choose the combination that is most important and make that be the search key of the clustered index.

3. If it is likely that ranged queries are going to be performed often, then we should use a B+-tree on the index for the relation since hash indexes cannot perform range queries. If it is more likely that we are only going to perform equality queries, for example the case of social security numbers, than hash indexes are the best choice since they allow for the faster retrieval than B+-trees by 2-3 I/Os per request.

4. First of all, both sorted files and tree-based indexes offer fast searches. Insertions and deletions, though, are much faster for tree-based indexes than sorted files. On the other hand scans and range searches with many matches are much faster for sorted files than tree-based indexes. Therefore, if we have read-only data that is not going to be modified often, it is better to go with a sorted file, whereas if we have data that we intend to modify often, then we should go with a tree-based index.

5. A composite search key is a key that contains several fields. A composite search key can support a broader range as well as increase the possibility for an index-only plan, but are more costly to maintain and store. An index-only plan is query evaluation plan where we only need to access the indexes for the data records, and not the data records themselves, in order to answer the query. Obviously, index-only plans are much faster than regular plans since it does not require reading of the data records. If it is likely that we are going to performing certain operations repeatly that only require accessing one field, for example the average value of a field, it would be an advantage to create a search key on this field since we could then accomplish it with an index-only plan.

Exercise 8.8 Consider a delete specified using an equality condition. For each of the five file organizations, what is the cost if no record qualifies? What is the cost if the condition is not on a key?

Answer 8.8 If the search key is not a candidate key, there may be several qualifying records. In a heap file, this means we have to search the entire file to be sure that we've found all qualifying records; the cost is $B(D + RC)$. In a sorted file, we find the first record (cost is that of equality search; $Dlog_2B + Clog_2R$) and then retrieve and delete successive records until the key value changes. The cost of the deletions is C per deleted record, and D per page containing such a record. In a hashed file, we hash to find the appropriate bucket (cost H), then retrieve the page (cost D; let's assume

no overflow pages), then write the page back if we find a qualifying record and delete it (cost D).

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.

10

TREE-STRUCTURED INDEXING

Exercise 10.1 Consider the B+ tree index of order $d = 2$ shown in Figure 10.1.

1. Show the tree that would result from inserting a data entry with key 9 into this tree.
2. Show the B+ tree that would result from inserting a data entry with key 3 into the original tree. How many page reads and page writes does the insertion require?
3. Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the left sibling is checked for possible redistribution.
4. Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the right sibling is checked for possible redistribution.
5. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 46 and then deleting the data entry with key 52.
6. Show the B+ tree that would result from deleting the data entry with key 91 from the original tree.

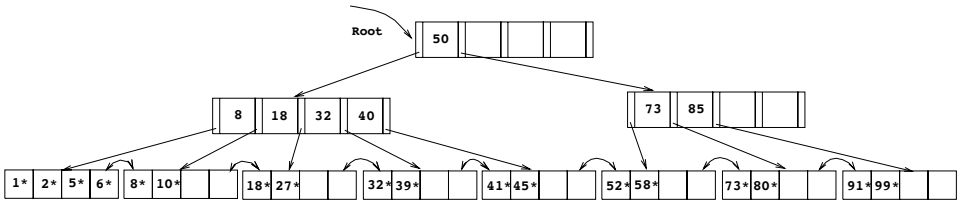

Figure 10.1 Tree for Exercise 10.1

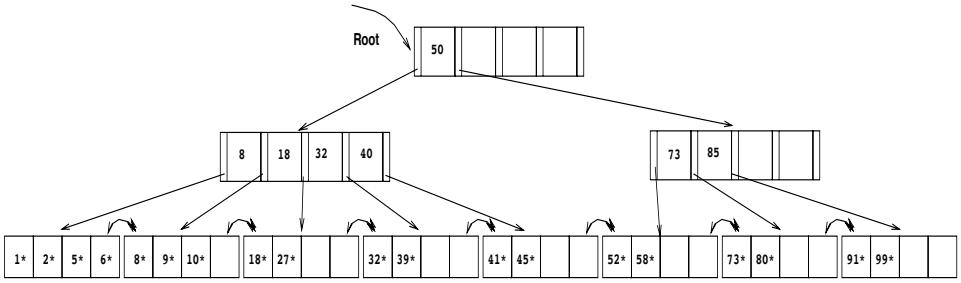

Figure 10.2

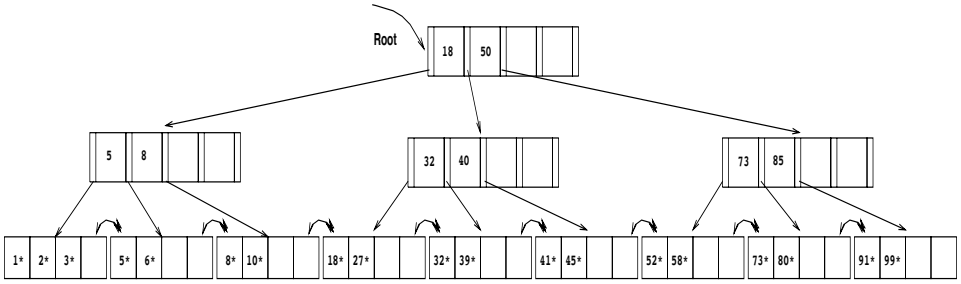

Figure 10.3

7. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 59, and then deleting the data entry with key 91.

8. Show the B+ tree that would result from successively deleting the data entries with keys 32, 39, 41, 45, and 73 from the original tree.

Answer 10.1 1. The data entry with key 9 is inserted on the second leaf page. The resulting tree is shown in figure 10.2.

2. The data entry with key 3 goes on the first leaf page F. Since F can accommodate at most four data entries ($d = 2$), F splits. The lowest data entry of the new leaf is given up to the ancestor which also splits. The result can be seen in figure 10.3. The insertion will require 5 page writes, 4 page reads and allocation of 2 new pages.

3. The data entry with key 8 is deleted, resulting in a leaf page N with less than two data entries. The left sibling L is checked for redistribution. Since L has more than two data entries, the remaining keys are redistributed between L and N, resulting in the tree in figure 10.4.

4. As is part 3, the data entry with key 8 is deleted from the leaf page N. N's right sibling R is checked for redistribution, but R has the minimum number of

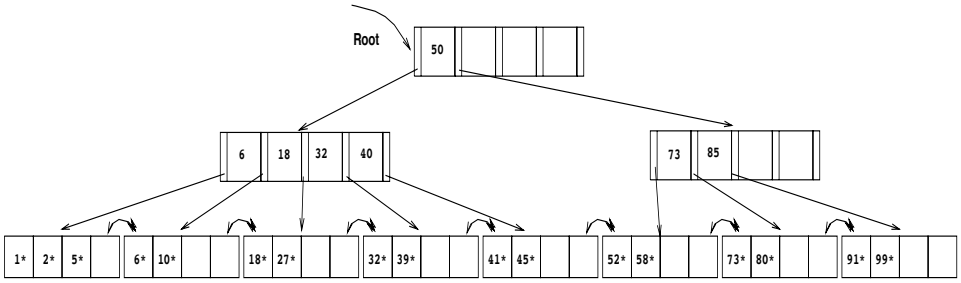

Figure 10.4

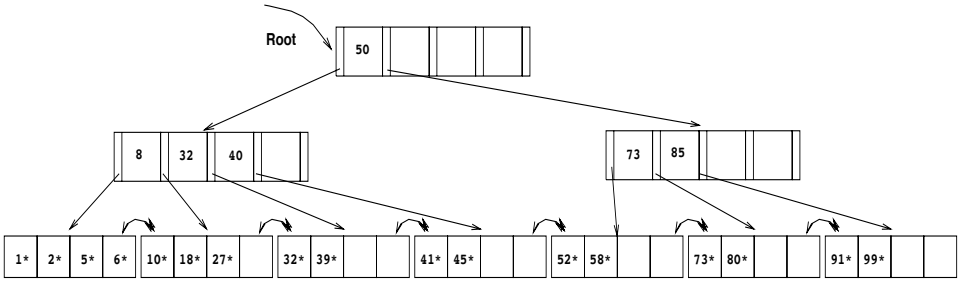

Figure 10.5

keys. Therefore the two siblings merge. The key in the ancestor which distinguished between the newly merged leaves is deleted. The resulting tree is shown in figure 10.5.

5. The data entry with key 46 can be inserted without any structural changes in the tree. But the removal of the data entry with key 52 causes its leaf page L to merge with a sibling (we chose the right sibling). This results in the removal of a key in the ancestor A of L and thereby lowering the number of keys on A below the minimum number of keys. Since the left sibling B of A has more than the minimum number of keys, redistribution between A and B takes place. The final tree is depicted in figure 10.6.

6. Deleting the data entry with key 91 causes a scenario similar to part 5. The result can be seen in figure 10.7.

7. The data entry with key 59 can be inserted without any structural changes in the tree. No sibling of the leaf page with the data entry with key 91 is affected by the insert. Therefore deleting the data entry with key 91 changes the tree in a way very similar to part 6. The result is depicted in figure 10.8.

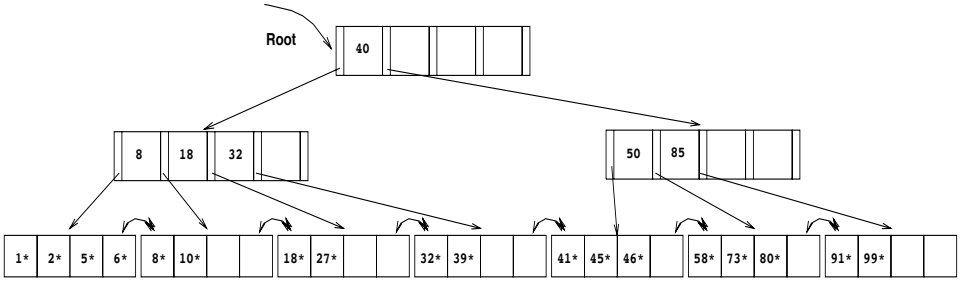

Figure 10.6

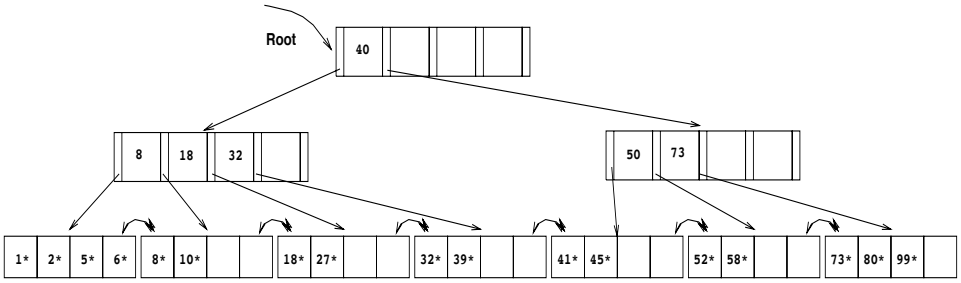

Figure 10.7

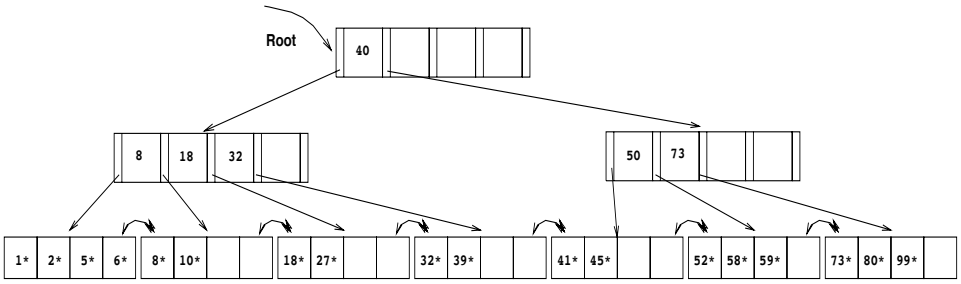

Figure 10.8

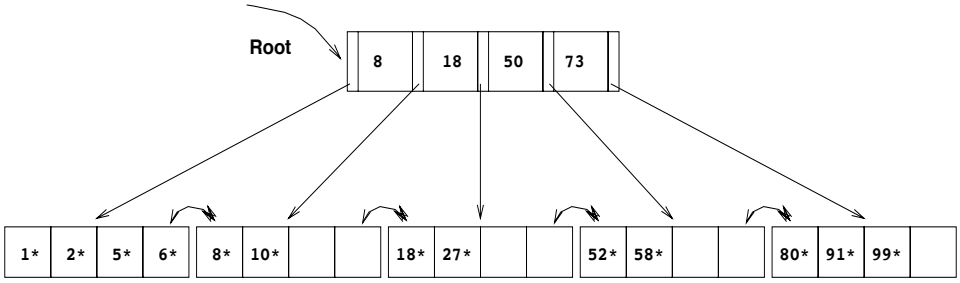

Figure 10.9

8. Considering checking the right sibling for possible merging first, the successive deletion of the data entries with keys 32, 39, 41, 45 and 73 results in the tree shown in figure 10.9.

Exercise 10.3 Answer the following questions:

1. What is the minimum space utilization for a B+ tree index?
2. What is the minimum space utilization for an ISAM index?
3. If your database system supported both a static and a dynamic tree index (say, ISAM and B+ trees), would you ever consider using the *static* index in preference to the *dynamic* index?

Answer 10.3 The answer to each question is given below.

1. By the definition of a B+ tree, each index page, except for the root, has at least d and at most $2d$ key entries. Therefore—with the exception of the root—the minimum space utilization guaranteed by a B+ tree index is 50 percent.
2. The minimum space utilization by an ISAM index depends on the design of the index and the data distribution over the lifetime of ISAM index. Since an ISAM index is static, empty spaces in index pages are never filled (in contrast to a B+ tree index, which is a dynamic index). Therefore the space utilization of ISAM index pages is usually close to 100 percent by design. However, there is no guarantee for leaf pages' utilization.
3. A static index without overflow pages is faster than a dynamic index on inserts and deletes, since index pages are only read and never written. If the set of keys that will be inserted into the tree is known in advance, then it is possible to build a static index which reserves enough space for all possible future inserts. Also if the system goes periodically off line, static indices can be rebuilt and scaled to the current occupancy of the index. Infrequent or scheduled updates are flags for when to consider a static index structure.

Exercise 10.5 Consider the B+ tree shown in Figure 10.21.

1. Identify a list of five data entries such that:

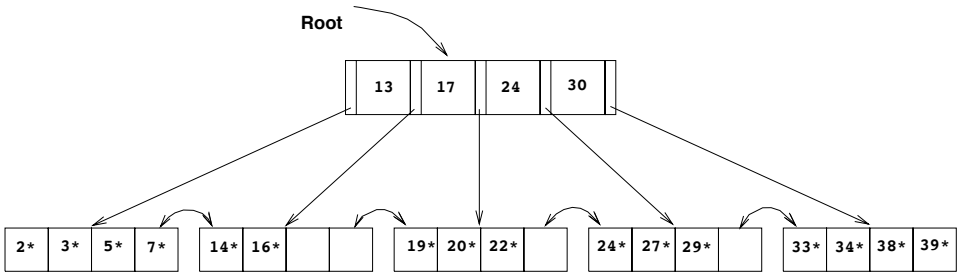

Figure 10.21 Tree for Exercise 10.5

(a) Inserting the entries in the order shown and then deleting them in the opposite order (e.g., insert a, insert b, delete b, delete a) results in the original tree.

(b) Inserting the entries in the order shown and then deleting them in the opposite order (e.g., insert a, insert b, delete b, delete a) results in a different tree.

2. What is the minimum number of insertions of data entries with distinct keys that will cause the height of the (original) tree to change from its current value (of 1) to 3?

3. Would the minimum number of insertions that will cause the original tree to increase to height 3 change if you were allowed to insert duplicates (multiple data entries with the same key), assuming that overflow pages are not used for handling duplicates?

Answer 10.5 The answer to each question is given below.

1. The answer to each part is given below.

   (a) One example is the set of five data entries with keys 17, 18, 13, 15, and 25. Inserting 17 and 18 will cause the tree to split and gain a level. Inserting 13, 15, and 25 does change the tree structure any further, so deleting them in reverse order causes no structure change. When 18 is deleted, redistribution will be possible from an adjacent node since one node will contain only the value 17, and its right neighbor will contain 19, 20, and 22. Finally, when 17 is deleted, no redistribution will be possible so the tree will loose a level and will return to the original tree.

   (b) Inserting and deleting the set 13, 15, 18, 25, and 4 will cause a change in the tree structure. When 4 is inserted, the right most leave will split causing the tree to gain a level. When it is deleted, the tree will not shrink in size. Since inserts 13, 15, 18, and 25 did not affect the right most node, their deletion will not change the altered structure either.

2. Let us call the current tree depicted in Figure 10.21 T. T has 16 data entries. The smallest tree S of height 3 which is created exclusively through inserts has $(1*2*3*3)*2+1 = 37$ data entries in its leaf pages. S has 18 leaf pages with two data entries each and one leaf page with three data entries. T has already four leaf pages which have more than two data entries; they can be filled and made to split, but after each spilt, one of the two pages will still has three data entries remaining. Therefore the smallest tree of height 3 which can possibly be created from T only through inserts has $(1*2*3*3)*2+4 = 40$ data entries. Therefore the minimum number of entries that will cause the height of T to change to 3 is $40 - 16$=24.

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.