Chapter 14-3, Lecture Notes

Processing Data in External Storage

14.3 External Tables

1. This section discusses techniques for organizing records in external storage so that you can efficiently perform ADT table operations such as retrieval, insertion, deletion, and traversal.
   1.1 Although this discussion will only scratch the surface of this topic, you do have a head start: Two of the most important external table implementations are variations of the 2-3 tree and hashing, which you studied in Chapter 12.
   1.2 Suppose that you have a direct access file of records that are to be table items.
      1.2.1 The file is partitioned into blocks, as described earlier in this chapter.
      1.2.2 One of the simplest table implementations stores the records in order by their search key, perhaps sorting the file by using the external mergesort algorithm developed in the previous section.
      1.2.3 Once it is sorted, you can easily traverse the file in sorted order by using the following algorithm:

```
traverseTable(in dataFile:File, in numberOfBlocks:integer,
              in recordsPerBlock:integer, in visit:FunctionType)
// Traverses the sorted file dataFile in sorted order,
// calling function visit() once for each item.

    // read each block of file dataFile into an internal
    // buffer buf
    for (blockNumber = 1 through numberOfBlocks)
    { buf.readBlock(dataFile, blockNumber)
      // visit each record in the block
      for (recordNumber = 1 through recordsPerBlock)
         Visit record buf.getRecord(recordNumber_l)
    } // end for
```

   1.3 To perform the tableRetrieve operation on the sorted file, you can use a binary search algorithm as follows:

```
tableRetrieve(in dataFile:File, in recordsPerBlock:integer,
              in first:integer, in last:integer,
              in searchKey:KeyItemType,
              out tableItem:TableItemType):boolean
// Searches blocks first through last of file dataFile,
// copies into tableItem the record whose search key equals
// searchKey, and returns true. The function returns false
// if no such item exists.
```

```
if (first > last or
        nothing is left to read from dataFile)
    return false
else
{ // read the middle block of file dataFile into
  // array buf
  mid = (first + last)12 buf.readBlock(dataFile, mid)

  if ( (searchKey>= (buf.getRecord(O)).getKey()) &&
       (searchKey <=
         (buf.getRecord(recordsperBlock_l)).getKey()) )

  {     // desired block is found
        Search buffer buf for record buf.getRecord(j)
                whose search key equals searchKey
        if (record is found)
        {  tableItem = buf.getRecord(j)
           return true
        }
        else
                return false
    } // end if

// else search appropriate half of the file
else if (searchKey < (buf.getRecord(O)).getKey())
        return tableRetrieve(dataFile, recordsPerBlock,
                first, mid-i, searchKey, tableItem)

else
        return tableRetrieve(dataFile, recordsPerBlock,
                                mid+i, last, searchKey, tableItem)
} // end if
```

1.4 The tableRetrieve algorithm recursively splits the file in half and reads the middle block into the internal object buf.

1.4.1 Splitting a file segment requires that you know the numbers of the first and last blocks of the segment.

1.4.2 You would pass these values as arguments, along with the file variable, to tableRetrieve.

1.5 Once you have read the middle block of the file segment into buf, you determine whether a record whose search key equals searchKey could be in this block.

1.5.1 You can make this determination by comparing searchKey to the smallest search key in buf-which is in buf. getRecord (0) and to the largest search key in buf-which is in buf. getRecord (recordsPerBlock-l).

1.5.2 If searchKey does not lie between the values of the smallest and largest search. keys in buf, you must recursively search one of the halves of the file (which hali to search depends on whether searchKey is less than or greater than the search keys in the block you just examined).
1.5.3 If, on the other hand, searcbKey does lie between the values of the smallest and largest search keys of the block in buf, you must search buf for the record.
1.5.4 Because the records within the block buf are sorted, you could use a binary search on the records within: this block.
1.5.5 However, the number of records in the block buf is typically small, and thus the time required to scan the block sequentially is insignificant compared to the time required to read the block from the file.
1.5.6 It is therefore common simply to scan the block sequentially.

1.6 This external implementation of the ADT table is not very different from the internal sorted array-based implementation.
1.6.1 As such, it has many of same advantages and disadvantages. Its main advantage is that because records are sorted sequentially, you can use a binary search to locate the block that contains a given search key.
1.6.2 The main disadvantage of the implementation is that, as is the case with an array-based implementation, the tableInsert and tableDelete operations must shift table items.
1.6.3 Shifting records in an external file is, in general, far more costly than shifting array items.
1.6.4 A file may contain an enormous number of large records, which are organized as several thousand blocks.
1.6.5 As a consequence, the shifting could require a prohibitively large number of block accesses.

1.7 Consider, for example, Figure 14-5. If you insert a new record into block k, you must shift the records not only in block k, but also in every block after it.
1.7.1 As a result, you must shift some records across block boundaries.
1.7.2 Thus, for each of these blocks, you must read the block into internal memory, shift its records by using an assignment such as

```
buf.setRecord(i+l, buf.getRecord(i))
```

1.7.3 and write the block to the file so that the file reflects the change.
1.7.4 This large number of block accesses makes the external sorted array-based implementation practical only for tables where insertions and deletions are rare. (See Exercise 1 at the end of this chapter.)

Indexing an External File

2 Two of the best external table implementations are variations of the internal hashing and search-tree schemes.

2.1 The biggest difference between the internal and external versions of these implementations is that in the external versions, it is often advantageous to organize an index to the data file rather than to organize the data file itself.
   2.1.1 An index to a data file is conceptually similar to other indexes with which you are familiar. For example, consider a library catalog.
   2.1.2 Rather than looking allover the library for a particular title, you can simply search the catalog.
   2.1.3 The catalog is typically organized alphabetically by title (or by author), so it is a simple matter to locate the appropriate entry.
   2.1.4 The entry for each book contains an indication (for example, a Library of Congress number) of where on the shelves you can find the book.

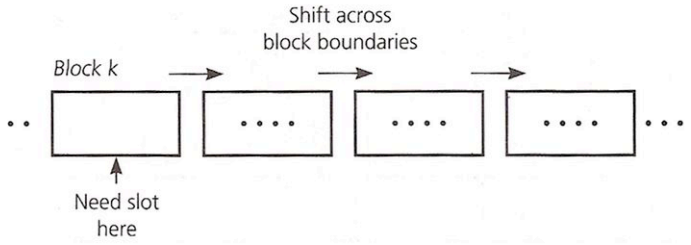

FIGURE 14-5
Shifting across block boundaries

2.2 Using a catalog to index the books in a library has at least three benefits:

- Because each catalog entry is much smaller than the book it represents, the entire catalog for a large library can fit into a small space. A patron can thus locate a particular book quickly.

- The library can organize the books on the shelves in any way, without regard to how easy it will be for a patron to scan the shelves for a particular book. To locate a particular book, the patron searches the catalog for the appropriate entry.

- The library can have different types of catalogs to facilitate different types of searches. For example, it can have one catalog organized by title and another organized by author.

2.3 Now consider how you can use an index to a data file to much the same advantage as the library catalog.
   2.3.1 As Figure 14-6 illustrates, you can leave the data file in a disorganized state and maintain an organized index to it.
   2.3.2 When you need to locate a particular record in the data file, you search the index for the corresponding entry, which will tell you where to find the desired record in the data file.

2.3.3 An index to the data file is simply a file, called the **index file**, that contains an **index record** for each record in the data file, just as a library catalog contains an entry for each book in the library.

2.3.3.1 An index record has two parts: a key, which contains the same value as the search key of its corresponding record in the data file, and a pointer, which shows the number of the block in the data file that contains this data record. (Despite its name, an index record's pointer contains an integer, not a C++ pointer.)

2.3.3.2 You thus can determine which block of the data file contains the record whose search key equals searchKey by searching the index file for the index record whose key equals searchKey.

2.3.4 Maintaining an index to a data file has benefits analogous to those provided by the library's catalog:

- In general, an index record will be much smaller than a data record. While the data record may contain many components, an index record contains only two: a key, which is also part of the data record, and a single integer pointer, which is the block number. Thus, just as a library catalog occupies only a small fraction of the space occupied by the books it indexes, an index file is only a fraction of the size of the data file. As you will see, the small size of the index file often allows you to manipulate it with fewer block accesses than you would need to manipulate the data file.

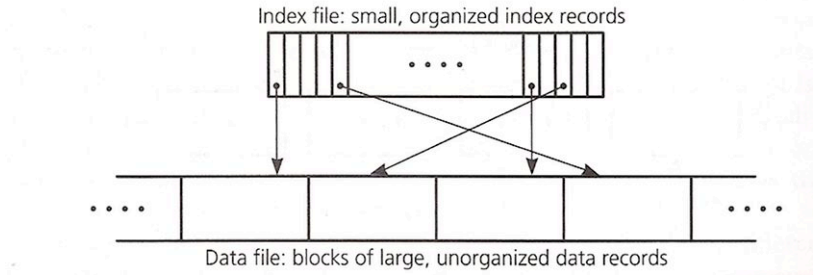

FIGURE 14-6
A data file with an index

- Because you do not need to maintain the data file in any particular order, you can insert new records in any convenient location, such as at the end of the file. As you will see, this flexibility eliminates the need to shift the data records during insertions and deletions .

- You can maintain several indexes simultaneously. Just as a library can have one catalog organized by title and another organized by author, you can have one index file that indexes the data file by one search key (for example, an index file that consists of <name, pointer> records), and a second index file that indexes the data file by another search key

(for example, an index file that consists of <socSec, pointer> records). Such **multiple indexing** is discussed briefly at the end of this chapter.

2.4 Although you do not organize the data file, you must organize the index file so that you can search and update it rapidly.

2.4.1 Before considering how to organize an index file by using either hashing or search-tree schemes, first consider a less complex organization that illustrates the concepts of indexing.

2.4.2 In particular, let the index file simply store the index records sequentially, sorted by their keys, as shown in Figure 14-7.

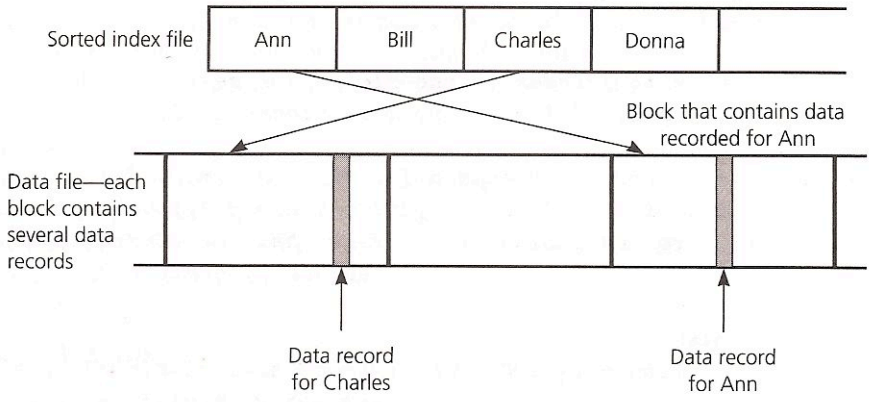

FIGURE 14-7
A data file with a sorted index file

2.4.3 To perform the tableRetrieve operation, for example, you can use a binary search on the index file as follows:

```
tableRetrieve(in tIndex:File, in tData:File,
              in searchKey:KeyItemType,
              out tableItem:TableItemType):boolean
// Retrieves into tableItem the record whose search key
// equals searchKey, where tIndex is the index file and
// tData is the data file. The operation returns true if
// the record exists, false otherwise.

    if (no blocks are left in tIndex to read)
        return false

    else
    { // read the middle block of index file into object buf
      mid = number of middle block of index file tIndex
      buf.readBlock(tIndex, mid)

        if ((searchKey >= (buf.getRecord(O)).getKey()) &&
            (searchKey <=
```

```
                (buf.getRecord(indexrecordsPerBlock-l)).getKey()))
        { // desired block of index file found
          Search buf for index file record
                buf.getRecord(j) whose key value equals searchKey

           if (index record buf.getRecord(j) is found)
           { blockNum = number of the data-file block to
                which buf.getRecord(j) points
             data.readBlock(tData, blockNum)
             Find data record data.getRecord(k) whose search
                key equals searchKey
              tableItem = data.getRecord(k)
              return true
        }

        else
                return false
}

else if (tIndex is one block in size)
        return false II no more blocks in file

// else search appropriate half of index file
else if (searchKey < (buf.getRecord(O)).getKey())
        return tableRetrieve(first half of tIndex, tData,
                                searchKey, tableItem)
else
        return tableRetrieve(second half of tIndex, tData,
                                searchKey, tableItem)

} // end if
```

2.5 Because the index records are far smaller than the data records, the index file contains far fewer blocks than the data file.

2.5.1 For example, if the index records are one-tenth the size of the data records and the data file contains 1,000 blocks, the index file will require only about 100 blocks.

2.5.2 As a result, the use of an index cuts the number of block accesses in tableRetrieve down from about log21000 '" 10 to about $1 + \log_2 100 \approx 8$. (The one additional block access is into the data file once you have located the appropriate index record.)

2.6 The reduction in block accesses is far more dramatic for the tableInsert and tableDelete operations.

2.6.1 In the implementation of an external table discussed earlier in this section, if you insert a record into or delete a record from the first block of data,

for example, you have to shift records in every block, requiring that you access all 1,000 blocks of the data file. (See Figure 14-5.)

2.6.2 However, when you perform an insertion or a deletion by using the index scheme, you have to shift only index records.

2.6.3 When you use an index file, you do not keep the data file in any particular order, so you can insert a new data record into any convenient location in the data file.

2.6.4 This flexibility means that you can simply insert a new data record at the end of the file or at a position left vacant by a previous deletion (as you will see).

2.6.5 As a result, you never need to shift records in the data file.

2.6.6 However, you do need to shift records in the index file to create an opening for a corresponding index entry in its proper sorted position.

2.6.7 Because the index file contains many fewer blocks than the data file (100 versus 1,000 in the previous example), the maximum number of block accesses required is greatly reduced.

2.6.8 A secondary benefit of shifting index records rather than data records is a reduction in the time requirement for a single shift.

2.6.9 Because the index records themselves are smaller, the time required for the statement buf. setRecord (i + 1, buf. getRecord (i)) is decreased.

2.7 Deletions under the index scheme reap similar benefits.

2.7.1 Once you have searched the index file and located the data record to be deleted, you can simply leave its location vacant in the data file, and thus you need not shift any data records.

2.7.2 You can keep track of the vacant locations in the data file (see Exercise 2), so that you can insert new data records into the vacancies, as was mentioned earlier.

2.7.3 The only shifting required is in the index file to fill the gap created when you remove the index record that corresponds to the deleted data record.

2.8 Even though this scheme is an improvement over maintaining a sorted data file, in many applications it is far from satisfactory.

2.8.1 The 100 block accesses that could be required to insert or delete an index record often would be prohibitive.

2.8.2 Far better implementations are possible when you use either hashing or search trees to organize the index file.

External Hashing

3 The external hashing scheme is quite similar to the internal scheme described in Chapter 12.

3.1 In the internal hashing scheme, each entry of the array table contains a pointer to the beginning of the list of items that hash into that location.

3.1.1 In the external hashing scheme, each entry of table still contains a pointer to the beginning of a list, but here each list consists of blocks of index records.

3.1.2 In other words, you hash an index file rather than the data file, as Figure 14-8 illustrates. (In many applications the array table is itself so large that you must keep it in external storage-for example, in the first K blocks of the index file.

3.1.3 To avoid this extra detail, you can assume here that the array table is an internal array.)

3.2 Associated with each entry table[i] is a linked list of blocks of the index file, as you can see in Figure 14-8.

3.2.1 Each block of table[i]'s linked list contains index records whose keys (and thus whose corresponding data records' search keys) hash into location i.

3.2.2 To form the linked lists, you must reserve space in each block for a block pointer – the integer block number of the next block in the chain – as Figure 14-9 illustrates.

3.2.3 That is, in this linked list the pointers are integers, not C++ pointers. A pointer value of ~ 1 is used as a NULL pointer.

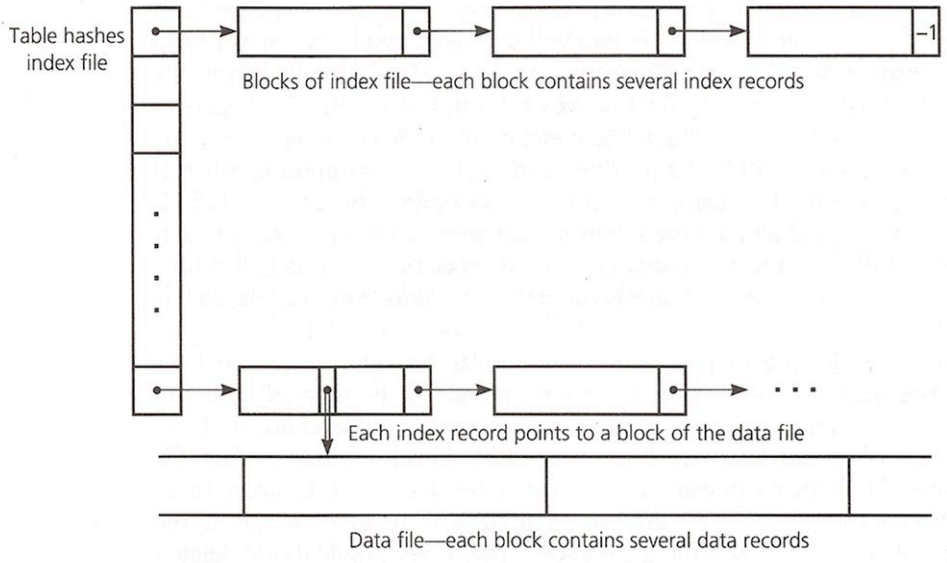

FIGURE 14-8
A hashed index file

3.3 Retrieval under external hashing of an index file. The tableRetrieve operation appears in pseudocode as follows:

```
tableRetrieve(in tIndex:File, in tData:File,
              in searchKey:KeyItemType,
              out tableItem:TableItemType):boolean
// Retrieves into tableItem the item whose search key
```

```
// equals searchKey, where tIndex is the index file, which
// is hashed, and tData is the data file. The function
// returns true if the record exists; false otherwise.

        // apply the hash function to the search key
        i = h(searchKey)

        // find the first block in the chain of index blocks
        // these blocks contain index records that hash into
        // location i
        p = table[i]

        // if p == -1, no values have hashed into location i
        if (p != -1)
                buf. readBlock (tIndex, p)

        // search for the block with the desired index record
        while (p != -1 and buf does not contain an index record
                        whose key value equals searchKey)
        { p = number of next block in chain
          // if p == -1, you are at the last block in the chain
          if (p != -1)
              buf.readBlock(tIndex, p)

        } // end while

        // retrieve the data item if present
        if (p != -1)
        { // buf.getRecord(j) is the index record whose
          // key value equals searchKey
          blockNum = number of the data-file block to
                          which buf.getRecord(j) points
        data.readBlock(tData, blockNum)
        Find data record data.getRecord(k) whose search key
                equals searchKey
        tableItem = data.getRecord(k)
        return true
  }
  else
        return false
```

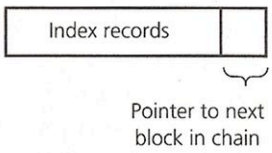

FIGURE 14-9

A single block with a pointer

3.4 **Insertion under external hashing of an index file**.

3.4.1 The external hashing versions of tableInsert and tableDelete are also similar to the internal hashing versions.

3.4.2 The major difference is that, in the external environment, you must insert or delete both a data record and the corresponding index record.

3.4.3 To insert a new data record whose search key is searchKey, you take the following steps:

1. **Insert the data record into the data file**. Because the data file is not ordered, the new record can go anywhere you want. If a previous deletion has left a free slot in the middle of the data file, you can insert it there. (See Exercise 2.)

If no slots are free, you insert the new data record at the end of the last block, or, if necessary, you append a new block to the end of the data file and store the record there. In either case, let p denote the number of the block that contains this new data record.

2. **Insert a corresponding index record into the index file**. You need to insert into the index file an index record that has key value searchKey and pointer value p. (Recall that p is the number of the block in the data file into which you inserted the new data record.) Because the index file is hashed, you first apply the hash function to searchKey) letting

i = h(searchKey)

You then insert the index record <searchKey, p> into the chain of blocks that the entry table [i] points to. You can insert this record into any block in the chain that contains a free slot, or, if necessary, you can allocate a new block and link it to the beginning of the chain.

3.4.4 **Deletion under external hashing of an index file**. To delete the data r whose search key is searchKey, you take the following steps:

1. **Search the index file for the corresponding index record**. You apply hash function to searchKey) letting

i = h(searchKey)

You then search the chain of index blocks pointed to by the table [iJ for an index record whose key value equals searchKey. do not find such a record, you can conclude that the data file does contain a record whose search key equals searchKey. However, if an index record <searchKey, p>, you delete it from the index noting the block number p, which indicates where in the data file you can find the data record to be deleted.

2. **Delete the data record from the data file**. You know that the data record is in block p of the data file. You simply access this block, search the block for the record, delete the record, and write the block back to the file.

3.4.5 Observe that for each of the operations tableRetrieve, tableInsert, and tableDelete the number of block accesses is very low.

3.4.5.1 You never have to access more than one block of the data file, and at worst you have to access all of the blocks along a single hash chain of the index file.

3.4.5.2 You can take measures to keep the length of each of the chains quite short (for example, one or two blocks long), just as you can with internal hashing.

3.4.5.3 You should make the size of the array table large enough so that the average length of a chain is near one block, and the hash function should scatter the keys evenly.

3.4.5.4 If necessary, you can even structure each chain as an external search tree-a B-tree-by using the techniques described in the next section.

3.4.6 The hashing implementation is the one to choose when you need to perform the operations tableRetrieve, tableInsert, and tableDelete on a large external table.

3.4.6.1 As is the case with internal hashing, however, this implementation is not practical for certain other operations, such as sorted traversal, retrieval of the smallest or largest item, and range queries that require ordered data.

3.4.6.2 When these types of operations are added to the basic table operations tableRetrieve, tableInsert, and tableDelete, you should use a search-tree implementation instead of hashing.

B-Trees

3.5 Another way to search an external table is to organize it as a balanced search tree. Just as you can apply external hashing to the index file, you can organize the index file, not the data file, as an external search tree.

3.5.1 The implementation developed here is a generalization of the 2-3 tree of Chapter 12.

3.5.2 You can organize the blocks of an external file into a tree structure by using block numbers for child pointers.

3.5.3 In Figure 14-10a, for example, the blocks are organized into a 2-3 tree. Each block of the file is a node in the tree and contains three child pointers, each of which is the integer block number of the child.

3.5.4 A child pointer value of -1 plays the role of a NULL pointer, and thus, for example, a leaf will contain three child pointers with the value -1.

3.5.5 If you organized the index file into a 2-3 tree, each node (block of the index file) would contain either one or two index records, each of the form <key, pointer>, and three child pointers.

3.5.6 The pointer portion of an index record has nothing to do with the tree structure of the index file; pointer indicates the block (in the data file) that contains the data record whose search key equals key. (See Figure l4-10b.)

3.5.7 To help avoid confusion, the pointers in the tree structure of the index file will be referred to as child pointers.

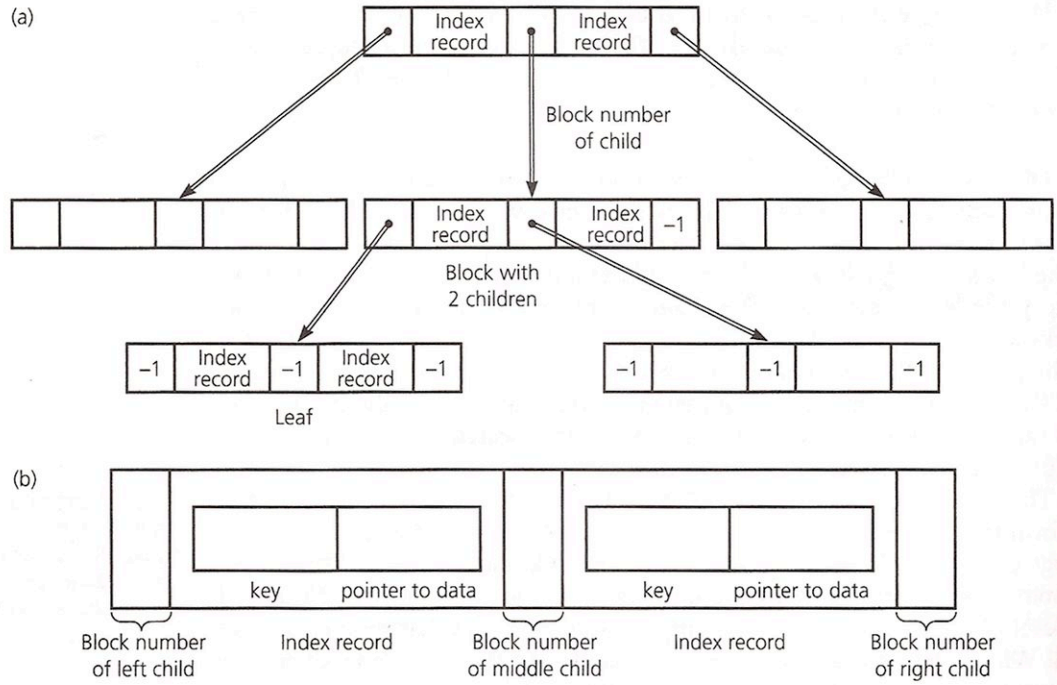

FIGURE 14-10

(a) Blocks organized into a 2-3 tree; (b) a single node of the 2-3 tree

3.5.8 You must organize the index records in the tree so that their keys obey the same search-tree ordering property as an internal 2-3 tree.

3.5.9 This organization allows you to retrieve the data record with a given value in its search key as follows:

```
tableRetrieve(in tIndex:File, in tData:File,
                        in rootNum:integer, in searchKey: KeyItem Type ,
                        out tableItem:TableItemType):boolean

// Retrieves into tableItem the record whose search key
// equals searchKey. tIndex is the index file, which is
// organized as a 2-3 tree. rootNum is the block number (of
// the index file) that contains the root of the tree. tData
// is the data file. The function returns true if the
// record exists; false otherwise.

        if (no blocks are left in tIndex to read)
                return false

        else
        { // read from index file into internal array buf the
          // block that contains the root of the 2-3 tree
          buf.readBlock (tIndex, rootNum)

          // search for the index record whose key value
          // equals searchKey

          if (searchKey is in the root)
          { blockNum = number of the data-file block that
                              index record specifies
            data.readBlock(tData, blockNum)
            Find data record data.getRecord(k) whose
                  search key equals searchKey
            tableItem = data.getRecord(k)
            return true
          }

          // else search the appropriate subtree
          else if (the root is a leaf)
                  return false

          else
          { child = block number of root of
                              appropriate subtree
            return tableRetrieve(tIndex, tData, child,
                                    searchKey, tableItem)

          } // end if
        } // end if
```

3.6 You also can perform insertions and deletions in a manner similar to the internal version, with the addition that you must insert records into and delete records from both the index file and the data file (as was the case in the external hashing scheme described earlier).
  3.6.1 In the course of insertions into and deletions from the index file, you must split and merge nodes of the tree just as you do for the internal version.
  3.6.2 You perform insertions into and deletions from the data file-which, recall, is not ordered in any way-exactly as described for the external hashing implementation.
  3.6.3 You thus can support the table operations fairly well by using an external version of the 2-3 tree.
  3.6.4 However, you can generalize the 2-3 tree to a structure that is even more suitable for an external environment.
  3.6.5 Recall the discussion in Chapter 12 about search trees whose nodes can have many children.
  3.6.6 Adding more children per node reduces the height of the search tree but increases the number of comparisons at each node during the search for a value.

3.7 In an external environment, however, the advantage of keeping a search tree short far outweighs the disadvantage of performing extra work at each node.
  3.7.1 As you traverse the search tree in an external environment, you must perform a block access for each node visited.
  3.7.2 Because the time required to access a block of an external file is, in general, far greater than the time required to process the data in that block once it has been read in, the overriding concern is to reduce the number of block accesses required.
  3.7.3 This fact implies that you should attempt to reduce the height of the tree, even at the expense of requiring more comparisons at each node.
  3.7.4 In an external search tree, you should thus allow each node to have as many children as possible, with only the block size as a limiting factor.
  3.7.5 How many children can a block of some fixed size accommodate?
  3.7.6 If a node is to have m children, clearly you must be able to fit m child pointers in the node.
  3.7.7 In addition to child pointers, however, the node must also contain index records.
3.8 Before you can answer the question of how many children a block can accommodate, you must first consider this related question: If a node N in a search tree has m children, how many key values-and thus how many index records-must it contain?

4 In a binary search tree, if the node N has two children, it must contain one key value, as Figure 14-11a indicates.
  4.1 You can think of the key value in node N as separating the key values in N's two subtrees-all of the key values in N's left subtree are less than N's key value, and all of the key values in N's right subtree are greater than N's key value.

4.1.1 When you are searching the tree for a given key value, the key value in N tells you which branch to take.
4.1.2 Similarly, if a node N in a 2-3 tree has three children, it must contain two key values. (See Figure 14-11b.)
4.1.3 These two values separate the key values in N's three subtrees-all of the key values in the left subtree are less than N's smaller key value, all of the key values in N's middle subtree lie between N's two key values, and all of the key values in N's right subtree are greater than N's larger key value.
4.1.4 As is the case with a binary search tree, this requirement allows a search algorithm to know which branch to take at any given node.

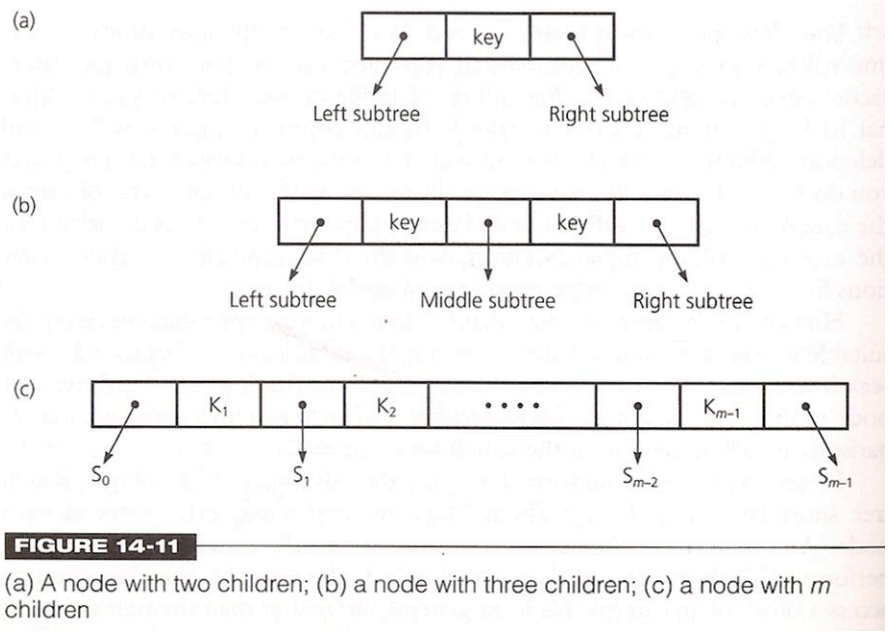

FIGURE 14-11

(a) A node with two children; (b) a node with three children; (c) a node with *m* children

4.2 In general, if a node N in a search tree is to have m children, it must contain m - 1 key values to separate the values in its subtrees correctly. (See Figure 14-11c.)
4.2.1 Suppose that you denote the subtrees of N as So, Sl, and so on to Sm-l and denote the key values in N as K_1, K_2, and so on to K_{m-1} (with $K_1 < K_2 < ... < K_{m-1}$)
4.2.2 The key values in N must separate the values in its subtrees as follows:

• All the values in subtree So must be less than the key value K1.

• For all i) 1 ~ i ~ m - 2, all the values in subtree Si must lie between the key values Ki and Ki+l'

• All the values in subtree Sm-l must be greater than the key value Km-1.

4.2.3 If every node in the tree obeys this property, you can search the tree by using a generalized version of a search tree's retrieval algorithm.
4.2.4 Thus, you can perform the tableRetrieve operation as follows:

```
tableRetrieve(in tIndex:File, in tData:File,
                  in rootNum:integer, in searchKey:KeyItem Type,
                  out tableItem:TableItemType):boolean
// Retrieves into tableItem the record whose search key
// equals searchKey. tIndex is the index file, which is
// organized as a search tree. rootNum is the block number
// (of the index file) that contains the root of the tree.
// tData is the data file. The function returns true if
// the record exists, false otherwise.

if (no blocks are left in tIndex to read)
        return false
else
{ // read from index file into internal array buf the
  // block that contains the root of the tree
  buf.readBlock(tIndex, rootNum)

  // search for the index record whose key value
  // equals searchKey
  if (searchKey is one of the ,Ki in the root)
  { blockNum = number of the data-file block that
                          index record specifies
    data.readBlock(tData, blockNum)
    Find data record data.getRecord(k) whose
       search key equals searchKey
    tableItem = data.getRecord(k)
    return true
  }
  // else search the appropriate subtree
  else if (the root is a leaf)
        return false

  else
  { Determine which subtree Si to search
    child = block number of the root of Si
    return tableRetrieve(tIndex, tData, child,
    searchKey, tableItem)
  } // end if
} // end if
```

4.3 Now return to the question of how many children the nodes of the search tree can have-that is, how big can m be?

4.3.1 If you wish to organize the index file into a search tree, the items that you store in each node will be records of the form <key, pointer>.

4.3.2 Thus, if each node in the tree (which, recall, is a block of the index file) is to have m children, it must be large enough to accommodate m child pointers and m - 1 records of the form <key, pointer>.

4.3.3 You should choose m to be the largest integer such that m child pointers (which, recall, are integers) and m - 1 <key, pointer> records can fit into a single block of the file.

4.3.4 Actually, the algorithms are somewhat simplified if you always choose an odd number for m.

4.3.5 That is, you should choose m to be the largest odd integer such that m child pointers and m - 1 index records can fit into a single block.

4.4 Ideally, then, you should structure the external search tree so that every internal node has m children, where m is chosen as just described, and all leaves are at the same level, as is the case with full trees and 2-3 trees.

4.4.1 For example, Figure 14-12 shows a full tree whose internal nodes each have five children.

4.4.2 Although this search tree has the minimum possible height, its balance is too difficult to maintain in the face of insertions and deletions.

4.4.3 As a consequence, you must make a compromise.

4.4.4 You can still insist that all the leaves of the search tree be at the same level-that is, that the tree be balanced-but you must allow each internal node to have between m and [mI2] + 1 children. (The [ ] notation means greatest integer in. Thus, [5/2] is 2, for example.)

4.4.5 This type of search tree is known as a B-tree of degree m and has the following characteristics:

• All leaves are at the same level.

• Each node contains between m - 1 and [m / 2] records, and each internal node has one more child than it has records. An exception to this rule is that the root of the tree can contain as few as one record and can have as few as two children. This exception is necessitated by the insertion and deletion algorithms described next.

4.4.6 A 2-3. tree is a B-tree of degree 3. Furthermore, the manner in which the B-tree insertion and deletion algorithms maintain the structure of the tree is a direct generalization of the 2-3 tree's strategy of splitting and merging nodes.

4.5 The B-tree insertion and deletion algorithms are illustrated next by means of an example.

4.5.1 Assume that the index file is organized into a B-tree of degree 5-that is, 5 is the maximum and 3 is the minimum number of children that an internal node (other than the root) in the tree can have. (Typically, a B-tree will be of a higher degree, but the diagrams would get out of hand!)

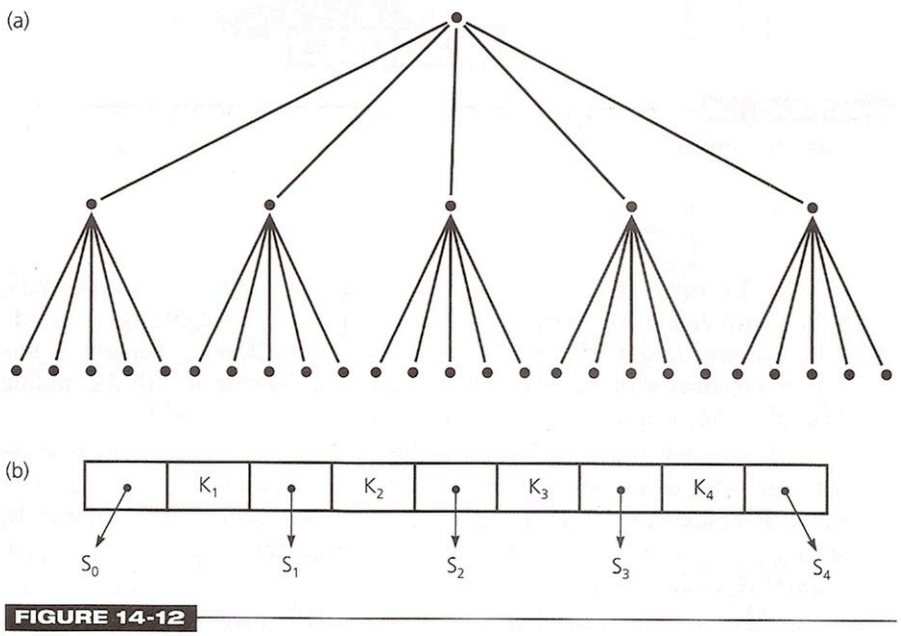

FIGURE 14-12

(a) A full tree whose internal nodes have five children; (b) the format of a single node

4.5.2 **Insertion into a B-tree.** To insert a data record with search key 55 into the tree shown in Figure 14-13, you take the following steps:

1. Insert the data record into the data file. First you find block p in the data file into which you can insert the new record. As was true with the external hashing implementation, block p is either any block with a vacant slot or a new block.

2. **Insert a corresponding index record into the index file**. You now must insert the index record <55, p> into the index file, which is a B-tree of degree 5. The first step is to locate the leaf of the tree in which this index record belongs by determining where the search for 55 would terminate.

Suppose that this is the leaf L shown in Figure 14-14a. Conceptually, you insert the new index record into L, causing it to contain five records (Figure 14-14b). Since a node can contain only four records, you must split L into L_1 and L_2. With an action analogous to the splitting of a node in a 2-3 tree, L_1 gets the two records with the smallest key values, L_2 gets the two records with the largest key values, and the record with the middle key value (56) is moved up to the parent P. (See Figure 14-14c.)

In this example, P now has six children and five records, so it must be split into PI and P2. The record with the middle key value (56) is moved up to P's parent, Q Then P's children must be distributed appropriately, as

happens with a 2-3 tree when an internal node is split. (See Figure 14-14d.)

At this point the insertion is complete, as P's parent Q now contains only three records and has only four children. In general, though, an insertion might cause splitting to propagate all the way up to the root (Figure 14-14e). If the root must be split, the new root will contain only one record and have only two children-the definition of a B-tree allows for this eventuality.

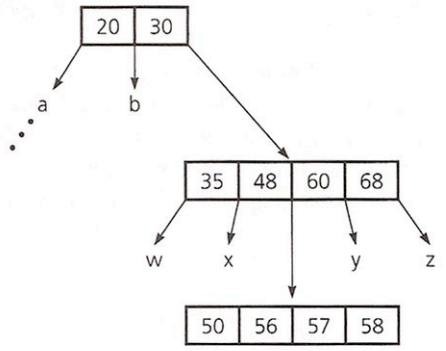

FIGURE 14-13
A B-tree of degree 5

4.5.3 **Deletion from a B-tree.** To delete a data record with a given search key from a B-tree, you take the following steps:

1. Locate the index record in the index file. You use the search algorithm to locate the index record with the desired key value. If this record is not already in a leaf, you swap it with its inorder successor. (See Exercise 8.) Suppose that the leaf L shown in Figure 14-15a contains the index record with the desired key value, 73. After noting the value p of the pointer in this index record (you will need p in Step 2 to delete the data record), you remove the index record from L (Figure 14-15b). Because L now contains only one value (recall that a node must contain at least two values), and since L's siblings cannot spare a value, you merge L with one of the siblings and bring down a record from the parent P (Figure 14-1Sc). Notice that this step is analogous to the merge step for a 2-3 tree. However, P now has only one value and two children, and since its siblings cannot spare a record and child, you must merge P with its sibling PI and bring a record down from P's parent, Q Because P is an internal node, its children must be adopted by Pl. (See Figure 14-1Sd.)

After this merge, P's parent Q is left with only two children and one record. In this case, however, Q's sibling Q_1 can spare a record and a child, so you redistribute children and records among Q_1, Q, and the parent S to complete the deletion. (See Figure 14-1Sc.) If a deletion ever propagates

all the way up to the root, leaving it with only one record and only two children, you are finished because the definition of a B-tree allows this situation. If a future deletion causes the root to have a single child and no records, you remove the root so that the tree's height decreases by 1, as Figure 14-15f illustrates. The deletion of the index record is complete, and you now must delete the data record.

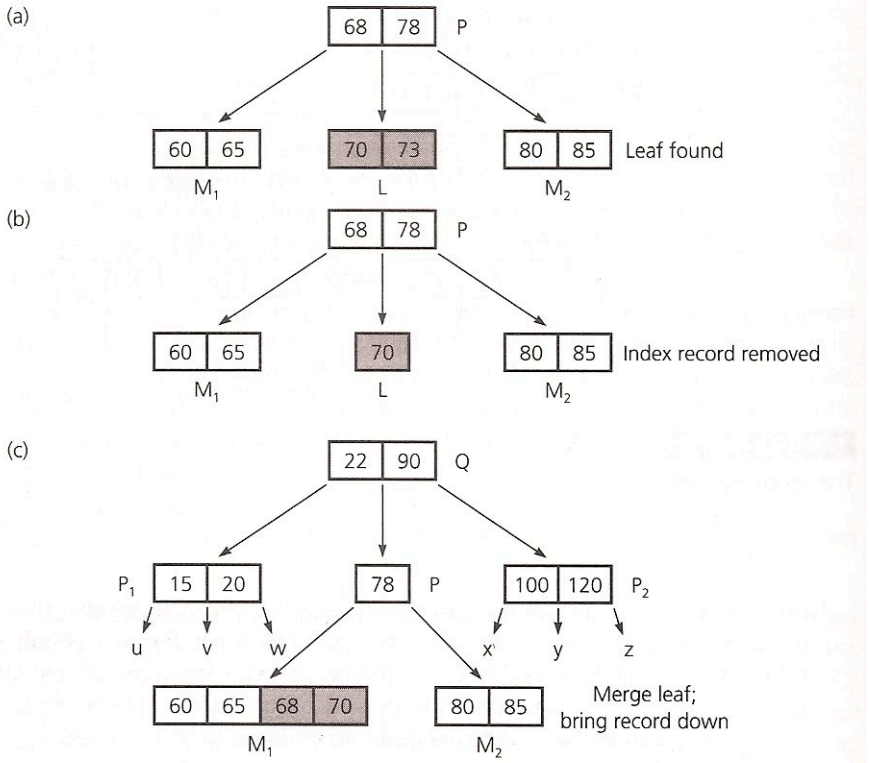

FIGURE 14-15

The steps for deleting 73

(continues)

2. Delete the data record from the data file. Prior to deleting the index record, you noted the value p of its pointer. Block p of the data file contains the data record to be deleted. Thus, you simply access block p, delete the data record, and write the block back to the file. The high-level pseudocode for the insertion and deletion algorithms parallels that of the 2-3 tree and is left as an exercise.

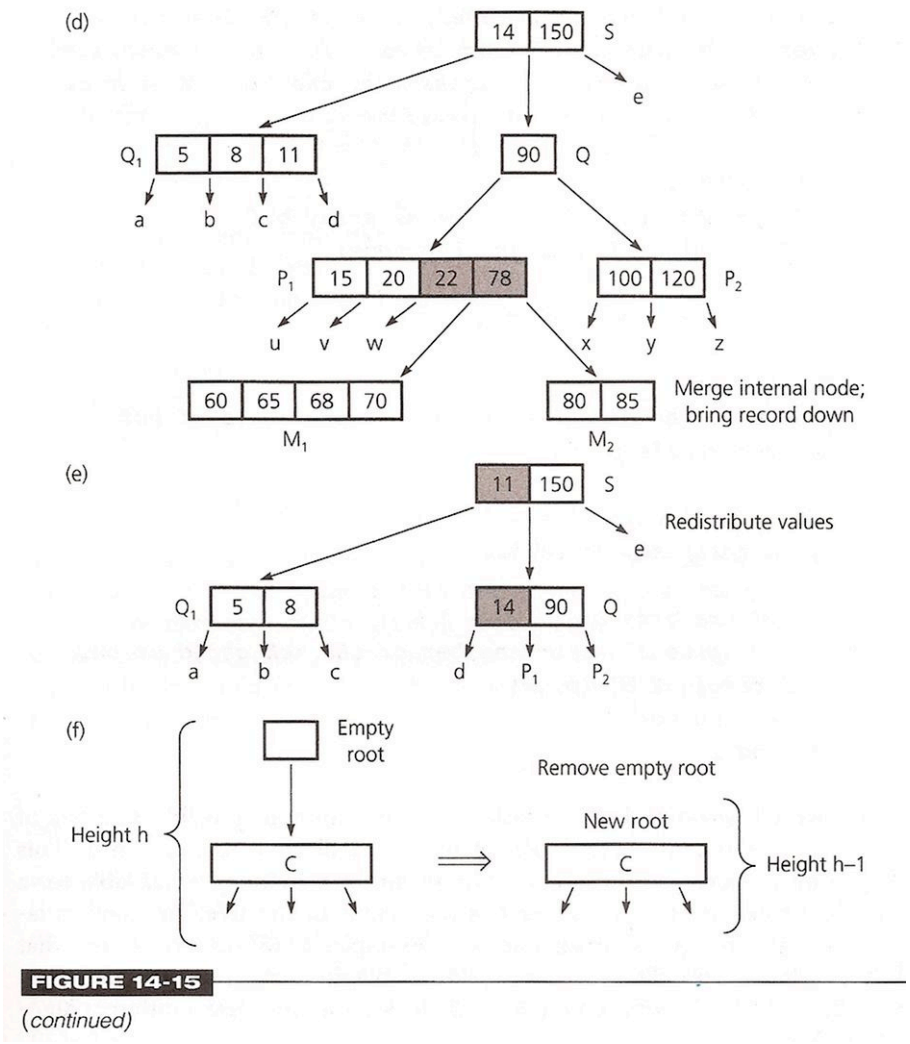

FIGURE 14-15

(continued)

Traversals

5 Now consider the operation traverseTable in sorted order, which is one of the operations that hashing does not support at all efficiently.

5.1 Often an application requires only that the traversal display the search keys of the records. If such is the case; the B-tree implementation can efficiently support the operation, because you do not have to access the data file.

5.2 You can visit the search keys in sorted order by using an inorder traversal of the B-tree, as follows:

```
traverseTable(in blockNum:integer, in m:integer)
// Traverses in sorted order an index file that is organized
// as a B-tree of degree m. blockNum is the block number of
// the root of the B-tree in the index file.
```

```
if (blockNum != -1)
{ // read the root into internal array buf
  buf.readBlock(indexFile, blockNum)

  // traverse the children

  // traverse SO
  Let p be the block number of the 0th child of buf
  traverseTable(p, m)

  for (i = 1 through m - 1)
  { Display key Ki of buf

    // traverse Si
    Let p be the block number of the ith child of buf
    traverseTable(p, m)
  } // end for
} // end if
```

5.3 This traversal accomplishes the task with the minimum possible number of block accesses because each block of the index file is read only once.

5.3.1 This algorithm, however, assumes that enough internal memory is available for a recursive stack of h blocks, where h is the height of the tree. In many situations this assumption is reasonable-for example, a 255-degree B-tree that indexes a file of 16 million data records has a height of no more than 3.

5.3.2 When internal memory cannot accommodate h blocks, you must use a different algorithm. (See Exercise 12.)

5.3.3 If the traversal must display the entire data record (and not just the search key), the B-tree implementation is less attractive.

5.3.4 In this case, as you traverse the B-tree, you must access the appropriate block of the data file.

5.3.5 The traversal becomes

```
traverseTable(in blockNum:integer, in m:integer)
// Traverses in sorted order a data file that is indexed
// with a B-tree of degree m. blockNum is the block number
// of the root of the B-tree.

if (blockNum != -1)
{ // read the root into internal array buf
  buf.readBlock(indexFile, blockNum)
  // traverse S0
  Let p be the block number of the 0th child of buf
  traverseTable(p, m)
  for (i = 1 through m - 1)
```

```
  { Let p_i be the pointer in the ith index
        record of buf
    data.readBlock(dataFile, p_i)
    Extract from data the data record whose search key
        equals Ki
    Display the data record

    // traverse Si
    Let p be block number of the ith child of buf
    traverseTable(p, m)
   } // end for
 } // end if
```

5.4 This traversal requires you to read a block of the data file before you display each data record; that is, the number of data-file block accesses is equal to the number of data records.
5.5 In general, such a large number of block accesses would not be acceptable.
5.6 If you must perform this type of traversal frequently, you probably would modify the B-tree scheme so that the data file itself was kept nearly sorted.

Multiple Indexing

6 Before concluding the discussion of external implementations, let's consider the multiple indexing of a data file.
6.1 Chapter 12 presented a problem in which you had to support multiple organizations for data stored in internal memory.
6.1.1 Such a problem is also common for data stored externally.
6.1.2 For example, suppose that a data file contains a collection of employee records on which you need to perform two types of retrievals:

```
retrieveN(in aName:NameType):ItemType
// Retrieves the item whose search key contains the
// name aName.

retrieveS(in ssn:SSNType):ItemType
// Retrieves the item whose search key contains the
// social security number ssn.
```

6.2 One solution to this problem is to maintain two independent index files to the data file.
6.2.1 For example, you could have one index file that contains index records of the form <name, pointer> and a second index file that contains index records of the form <socSec, pointer>.
6.2.2 These index files could both be hashed, could both be B-trees, or could be one of each, as Figure 14-16 indicates.

6.2.3 The choice would depend on the operations you wanted to perform with each search key. (Similarly, if an application required extremely fast retrievals on socSec and also required operations such as traverse in sorted socSec order and range queries on socSec, it might be reasonable to have two socSec index files-one hashed, the other a B-tree.)

6.3 Although you can perform each retrieval operation by using only one of the indexes (that is, use the name index for retrieveN and the socSec index for retrieveS), insertion and deletion operations must update both indexes.

6.4 For example, the delete-by-name operation deleteN (Jones) requires the following steps:

1. Search the name index file for Jones and delete the index record.

2. Delete the appropriate data record from the data file, noting the socSec value ssn of this record.

3. Search the socSec index file for ssn and delete this index record.

6.5 In general, the price paid for multiple indexing is more storage space and an additional overhead for updating each index whenever you modify the data file.

6.6 This chapter has presented at a very high level the basic principles of managing data in external storage.

6.6.1 The details of implementing the algorithms depend heavily on your specific computing system.

6.6.2 Particular situations often mandate either variations of the techniques described here or completely different approaches.

6.6.3 In future courses and work experience, you will undoubtedly learn much more about these techniques.

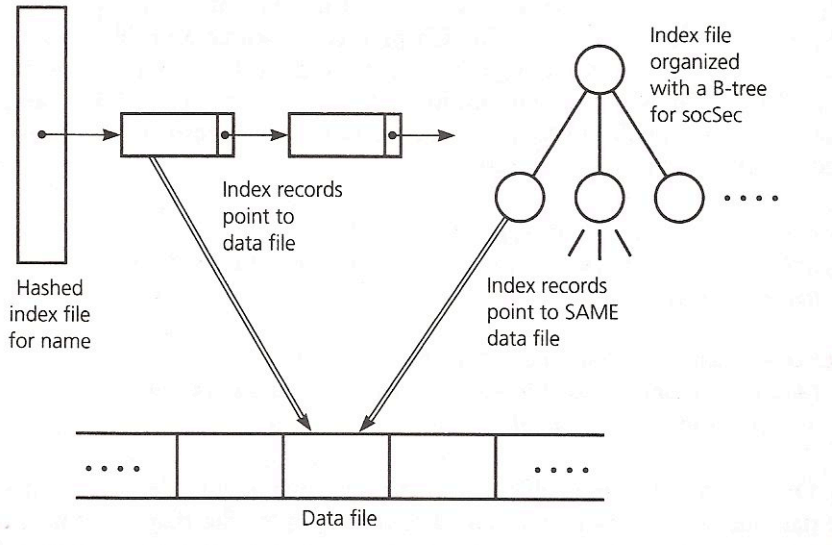

FIGURE 14-16
Multiple index files