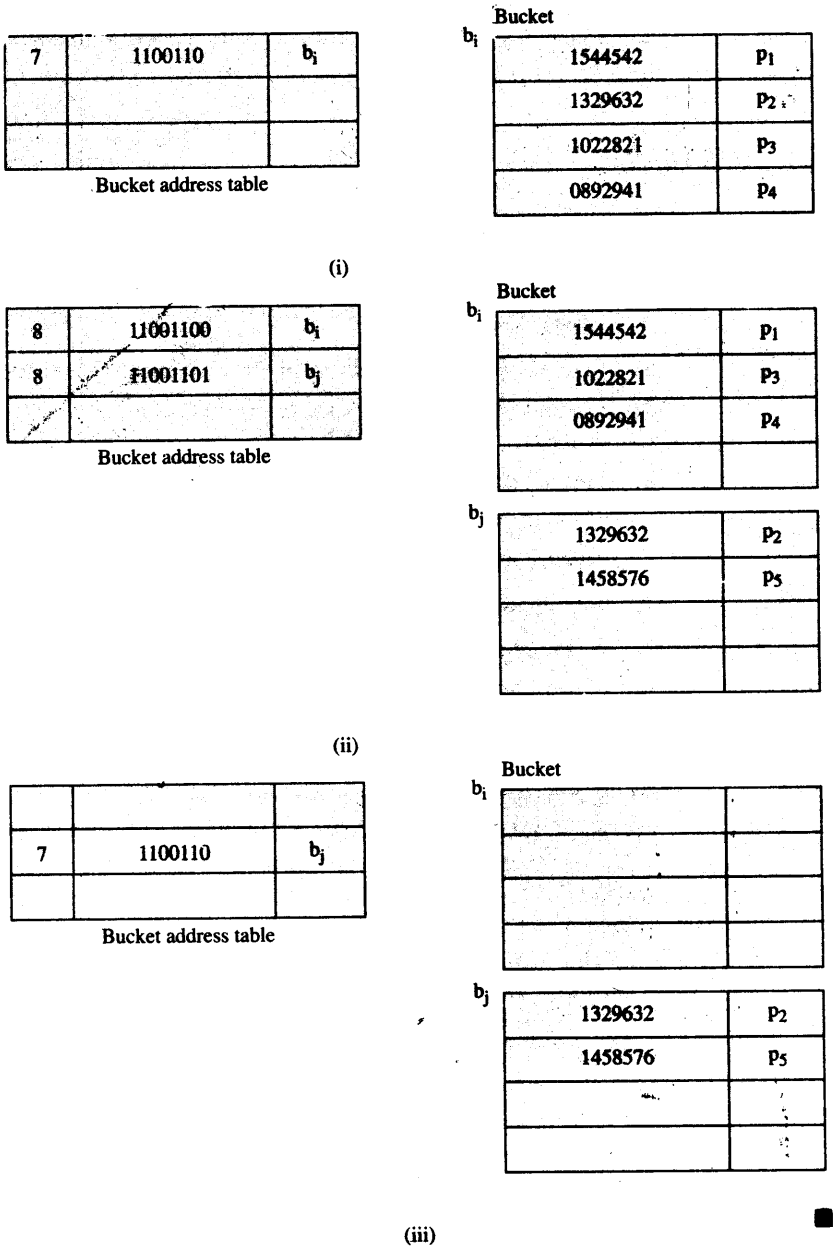
Figure 3

Example of extendable hashing.



3.6 Secondary Key Retrieval

In the previous sections we have considered the retrieval and update of data based on the primary key. In the following sections we consider file organizations that facilitate secondary key retrieval. Secondary key retrieval is characterized by the multiplicity of records satisfying a given key value. As such, there is no longer a

one-to-one correspondence between key values and records. File organizations for secondary key retrieval are used in conjunction with methods for primary key retrieval.

Query and Update Types

Queries are in general formulated to retrieve records based on one or multiple key values. In the latter case, the retrieval expression contains key values punctuated with Boolean operators.

Query Types:

1. Find all employees working in the computer science department.
2. Find all employees working in the computer science department who are analysts.
3. Find all students who are taking the files and database course, but not the artificial intelligence course.

Update types:

1. Add records in proper sequence.
2. Delete records satisfying some condition.
3. Modify attribute values of records, satisfying some condition.

The above queries and updates can be simply but inefficiently handled by scanning every record in the file. A number of file organizations permit faster and more efficient retrieval. The choice between them, just as in the case of primary key retrieval, is solely dependent on the application. Faster access to the records is provided by the use of indexes and/or the linking together in lists or some other suitable structure of logically related records. It is usual to relate records based on <attribute, value> pairs.

The secondary key structures support access to all records that satisfy some <attribute, value> pair. Logically, as shown in Figure 3.19a, the secondary key access file is made up of a set of records containing (attribute, value, record_list). Here record_list is a list of records that contain the <attribute, value> pair. For example, in the following secondary key access file entry, the records $R_{ij_1}, \ldots, R_{ij_n}$ contain the value v_{ij} for the attribute A_i:

$$\{<A_i, v_{ij}>,(R_{ij_1}, \ldots, R_{ij_n})\}$$

The R_{ij_k}'s are used to represent the associated records and may be either the primary key values, some unique system assigned identifiers, or unique physical addresses.

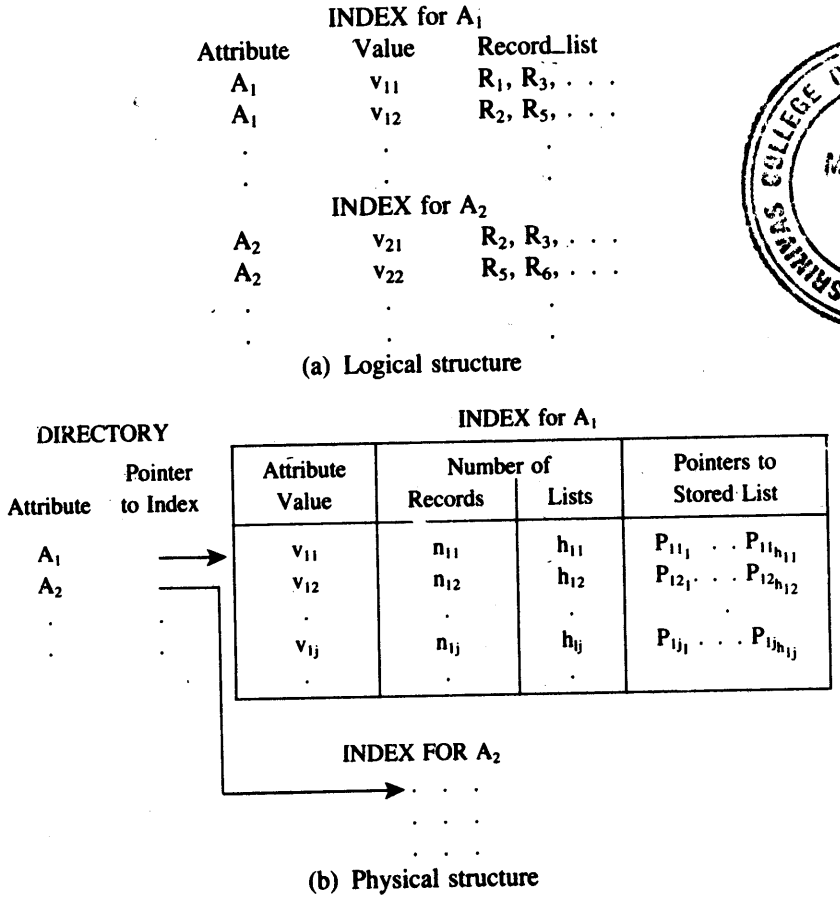
In general, the record_list $(R_{ij_1}, \ldots, R_{ij_n})$ may be maintained as a number of separate stored lists, for instance, h_{ij}, such that we have

$$<A_i, v_{ij}, n_{ij}, h_{ij}> (P_{ij_1}, \ldots, P_{ij_{h_{ij}}})$$

where n_{ij} is the number of records with value v_{ij} for the attribute A_i (i.e., n_{ij} is the number of records in the record_list $R_{ij_1}, \ldots, R_{ij_n}$) and P_{ij_k} is the pointer to the kth stored list, for all $k = 1, \ldots, h_{ij}$. The average length of each stored list is n_{ij}/h_{ij}.

Physically, as shown in Figure 3.19b, the names of the attributes may be separated from the values and record_list and kept in a directory. Each entry in the

Figure 3.19 Structure of the directory and index.



(a) Logical structure

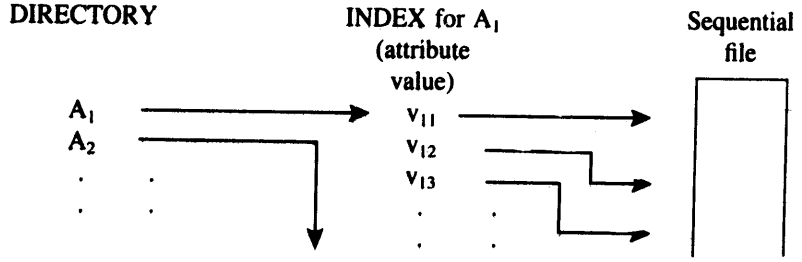
(b) Physical structure

directory is associated with a given attribute and points to a structure containing the set of associated (value, record_list) pairs. For the moment, we can think of the structure containing the (value, record_list) pairs as a sequential file, referring to it as the value-access file or as the attribute index. There are two common methods of organizing the value-access file: the inverted index method and the multilist. We discuss these organizations in the following sections.

3.6.1 Inverted Index Files

The **inverted index file** (or simply the **inverted file**) contains the list of all records satisfying the particular <attribute, value> pair in the index, wherein h_{ij} (the number of separate stored lists) is equal to n_{ij} (the number of records with the given attribute value) and each P_{jk_m} points to a list of records of length one (P_{jk_m} is in effect R_{ij}, a pointer to the record instead of to a record list). In other words, a pointer for every record with the given value v_{ij} for the attribute A_i is kept in the index. This pointer

Figure 3.20 A simple implementation of an inverted index.



A simple implementation of an inverted list to maintain the record_list for each value for a given attribute as a sequential file is shown in Figure 3.20. The index contains a <value, pointer> pair, where the pointer points to the starting position of the associated record_list in the sequential file.

3.6.2 Multilist Files

In a **multilist file** there is only one stored list for every <attribute, value> pair. Therefore, the index of a multilist file contains only the single address P_{ij} for the <attribute,value> pair $<A_i, v_{ij}>$; $h_{ij} = 1$. There is only one stored list of length n_{ij}. The records in the stored list are linked together in the form of a list. Thus, the record list of a multilist file is implemented as a list of records. One exists for every <attribute, value> pair (as the name suggests), with each stored record containing a pointer indicating the succeeding member of every list to which it belongs. A pointer to the first member of every list is maintained in the index. The length of each list can also be maintained in the index (this is illustrated in Figure 3.22a).

Figure 3.21 gives, in pseudo-Pascal, the definition of a record, all of whose attributes participate in multilists. The pointer field associated with each attribute can

Figure 3.21 Pseudo-Pascal definition of a stored record in a multilist file.

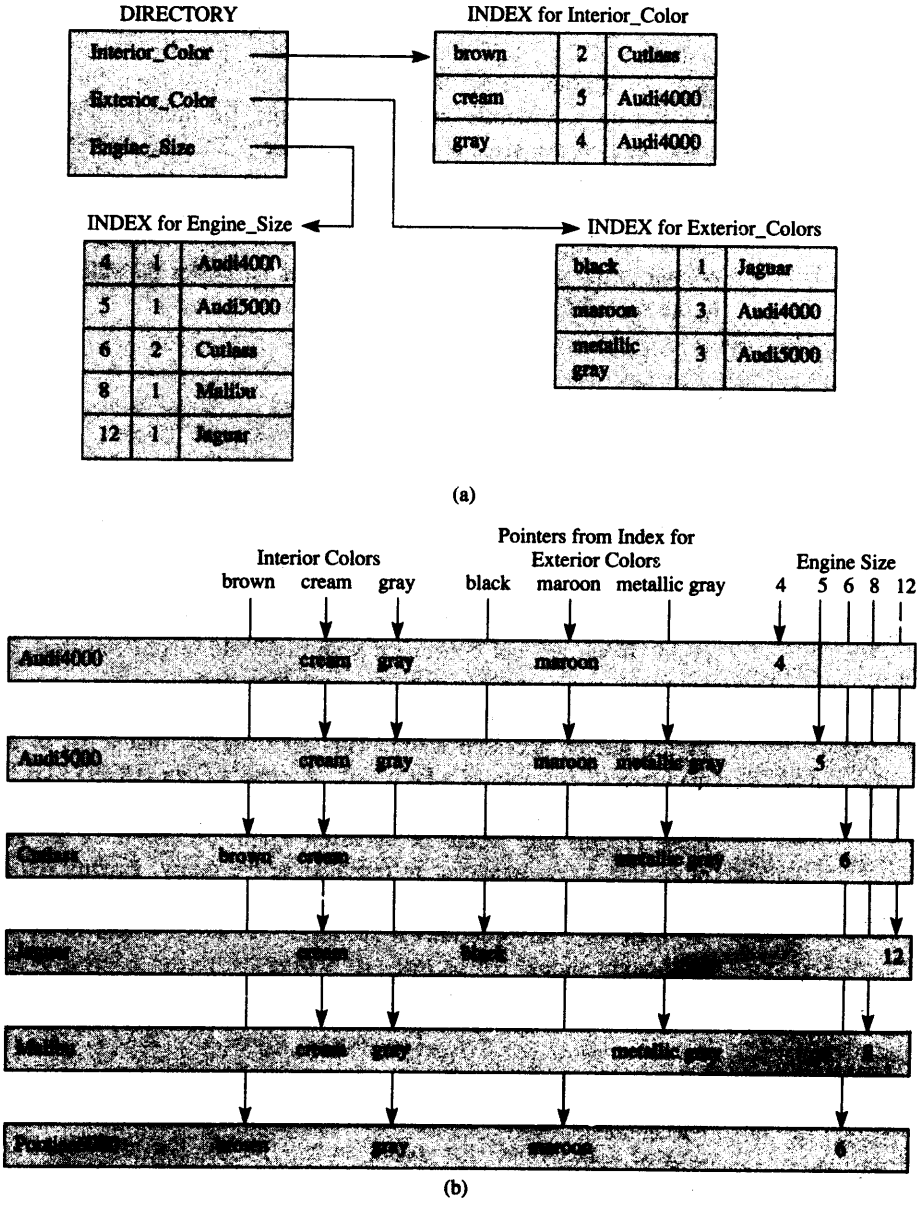
```
attribute_rec_type_i = record
    value : attribute_type_i;
    next : pointer {pointer to next record}
    end;
    .
    .

stored_record = record
    attribute_1 : attribute_rec_type_1;

    attribute_j : array[1..m] of attribute_rec_type_i;

     attribute_n : attribute_rec_type_n;
     end;
```

Figure 3.22 Multilist file.



store the pointer to the next record with the same value. If an attribute has multiple values (e.g., the same model car in the automobile dealership example comes in many interior and exterior colors and engine sizes), the attribute may be stored as an array of size m, as indicated for the *attribute_j* in Figure 3.21.

A simple method of creating multilist files is to insert new records at the front of the list. Searching for a specific record with a given value for an attribute requires

and discover that we already retrieved the record for Audi4000. We do not retrieve that record and find from the entry for Audi4000 that the next record in the list for *Interior_Color* = gray is Audi5000. Before actually retrieving this record we consult the DONTAG list again and discover that the record for Audi5000 has been processed and the next record in the list for *Interior_Color* = gray is Malibu. However, since there is an entry for Malibu in the DONTAG list, it was already retrieved. From this entry for Malibu in the DONTAG list we find the next record in the list for *Interior_Color* = gray to be Pontiac6000. There being no entry for Pontiac6000 in the DONTAG list, we retrieve and process it. Since there are no more records in the list for *Interior_Color* = gray, we have accessed all records. In this way we ensure that each record satisfying more than one term in the disjunct will be retrieved only once. ■

Maintenance of Multilist Files

The deletion of records entails the removal of the record from the various lists. In some implementations of the multilist where the record is not physically removed but only flagged to indicate its deletion, no change is involved. While the record is still physically part of the lists, it is not so logically. If a record is both deleted and physically removed, all the lists of which the record forms a part have to be altered as well. In any case, the length of each of the lists in which the record was involved is decremented.

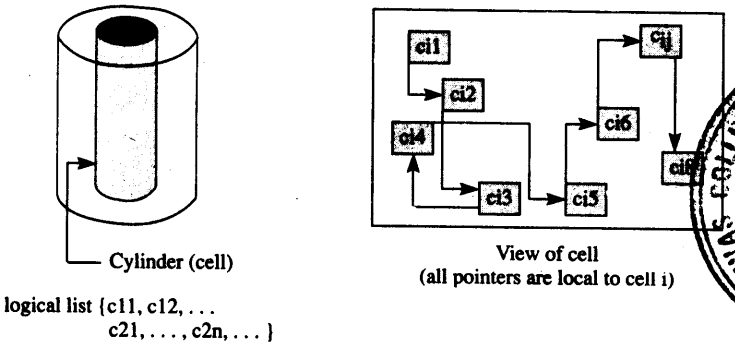
A record must first be located before a change can be made to its data values. If the value to be changed belongs to a secondary key field, we would have to alter the relevant list. This entails that the list be traversed with the old value, the record removed from the list, the value changed, and the record added to the list for the new value. If data values in a number of fields are changed, this may require the traversal and update of many lists. The process is simpler if records are double-chained with pointers to both successor and predecessor records.

The performance of a multilist file is satisfactory when the individual lists are short. Regarding conjunctive queries, if the length of the lists are included in the index, the shortest list is used for record retrieval. However, the number of records actually satisfying all terms of the query may be a very small fraction of those retrieved. The use of the DONTAG list avoids reaccessing the same records in the case of disjunctive queries. When the lists become lengthy, it is desirable to break each list up into a number of sublists as in the case of the cellular lists discussed in the next section.

3.6.3 Cellular Lists

Lists in a multilist file can become lengthy. The fact that the stored records may be distributed among many physical (disk) storage units, or within the same storage unit in some manageable cluster of cylinders (the cluster may be a single cylinder), or some other manageable storage area, could be used to advantage by partitioning

Figure 3.23 Cellular list.



the lists along these boundaries (or cells). Thus, in a **cellular list** organization the lists are limited to be within a physical area of storage, referred to as a **cell.** Figure 3.23 is an example of a cellular list. The lists are limited to a single cylinder of a movable-head-disk-type storage device. The number of stored lists, h_{ij}, for a given <attribute, value> pair $<A_i, v_{ij}>$, may be more than 1, $1 \leq h_{ij} \leq n_{ij}$.

The number of stored lists still does not approach the inverted file case, except where there is only a single record in every cell. However, there are more stored lists than in the multilist case. The processing complexity lies between the inverted and multilist cases. Such an organization is particularly useful if the cell size is chosen so that the lists may be traversed in internal memory. In the case of paged systems, this may equal the page size. In multiprocessor systems, different processors may traverse lists within different cells in parallel to improve response times.

Let us reconsider the index structure of Figure 3.19 to explain the three file structures examined so far. In an inverted index the number of groups chosen is equal to the number of records, i.e., $h_{ij} = n_{ij}$. Each group is of length one and each pointer points to a single record. In a multilist file, $h_{ij} = 1$ and only one list of length n_{ij} exists for value v_{ij} of attribute A_i. With a cellular multilist, there are h_{ij} lists for value v_{ij} of attribute A_i, each list being limited to a convenient size to maximize the response time. The size of the list may be determined by the characteristics of the physical storage device. In the case of a disk-type device, the list may be limited to a single cylinder.

3.6.4 Ring Files

The last records of the lists in a multilist file points to a null record. In **ring files** the last record entry in each list points back to the index entry. Therefore, from any point within the list a forward traversal of the links would bring us to the index entry. The index entries contain the value for the attribute, making it unnecessary to store the attribute-value in the physical records. This makes for a smaller record. Figure 3.24 shows a number of rings for the car dealership data, shown in Figure C of example 3.10.

In DBMSs using the network data model, a set is implemented as a ring by linking the member record occurrences in a ring that starts at the owner record oc-

cannot, because they are the leaf nodes.) The pointers T_{Lj}, $1 \leq j \leq n$ (note, not n + 1), in the leaf nodes point to storage areas containing either records having a key value k_{lj}, or pointers to records, each of which has a key value k_{lj}. The number of key values in each leaf node is at least $[(m - 1)/2]$ and at most $m - 1$.

Note that unlike the index-sequential file, the B^+-tree need not be a clustering index. That is, records may or may not be arranged in storage according to their key values.

The pointer $T_{L(n+1)}$ is used to chain the leaf nodes in a sequential order. This allows for sequential processing of the underlying file of records.

The following conditions are satisfied by the nodes of a B^+-tree (and also by the nodes of the older B-tree scheme):

1. The height of the tree is ≥ 1.
2. The root has at least two children.
3. All nodes other than the root node and the leaf nodes have at least $\lceil m/2 \rceil$ children, where m is the order of the tree.
4. All leaf nodes are at the same level.

Example 3.14

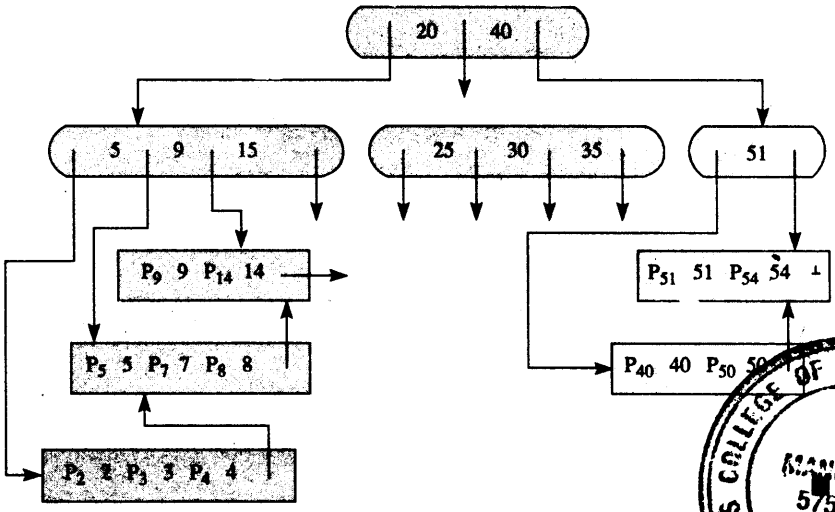
Assume that we are given a file containing the following records:

Book#	*Subject Area*
2	Files
3	Database
4	Artificial intelligence
5	Files
7	Discrete structures
8	Software engineering
9	Programming methodology
.	.
.	.
.	.
40	Operating systems
50	Graphics
51	Database
52	Data structures

A B^+-tree of order 4 on Book# is shown in Figure E.

■

Figure E A B^+-tree (showing only some of the leaf nodes). Each P_i is a pointer to the storage area containing records (or pointers) for the key Book# = i; $\perp$ represents a null pointer.



3.7.3 Operations

The nonleaf nodes of the B^+-tree act as a traversal map with the leat nodes containing the actual records or the key values with pointers to the storage location containing the records. Therefore, all operations require access to the leaf nodes.

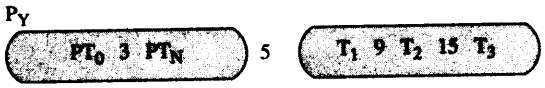
Searcn

The search algorithm for the B^+-tree is given in below. The number of nodes accessed is equal to the height of the tree. Once the required leaf node is reached, we can retrieve the pointer for the storage location containing the records; knowing the storage location, we can retrieve the required record(s).

Insertion and Deletion

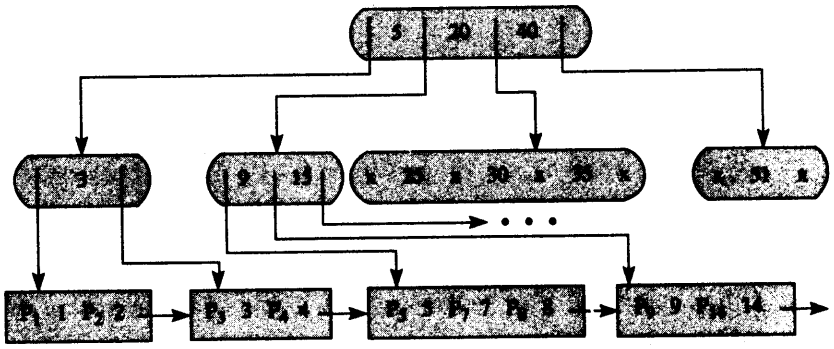
The insertion and deletion of records with a given key first requires a search of the tree. Below, we discuss the insertion (or deletion) of record keys from the trees. We assume that the records themselves would be inserted in (or deleted from) the pertinent storage locations. Insertion and deletion that violates the conditions on the number of keys in a node requires the redistribution of keys among a node, its sibling, and their parent.

The insertion causes a split of this node into the following two nodes with the key value 5, along with a pointer passed to the parent of the node:



Let the address of the new node be P_Y. Then the pair <5, P_Y> is passed to the parent node (in this case the root) for insertion. The relevant portion of the resultant B^+-tree is shown in Figure F.

Figure F The B^+-tree of Example 3.14 after insertion of the key for Book# 1.



■

Deletion

When a key is deleted, the leaf node may end up with less than $\lceil (m-1)/2 \rceil$ keys. This situation may also be handled by moving a key to the node from one of its left or right sibling nodes, and redistributing the keys in the parent node. However, if the siblings have no keys that could be spared, such redistribution is not possible. In this case, the node is merged with a sibling along with the deletion of a key from the parent node. The loss of the key from the parent node may in turn cause further redistribution or merging at this higher level of the tree.

The leaf node containing the key to be deleted is found and the key entry in the node deleted. If the resultant node (let us refer to it as TD) is empty or has fewer than $\lceil (m-1)/2 \rceil$ keys,

1. The data from the sibling node could be redistributed, i.e., the sibling has more than the minimum number of keys and one of these keys is enough to bring the number of keys in node TD to be equal to $\lceil (m-1)/2 \rceil$.
2. Or, the node TD is merged with the sibling to become a single node. This is possible if the sibling has only the minimum number of keys. The merger of the two nodes would still make the number of keys in the new node less than the maximum.

In the former case the key entry in the parent node will be changed to reflect the redistribution, and in the latter case the associated entry in the parent node would also be deleted.

Example 3.16

Let us delete the entry for Book# 5 from the tree shown in Example 3.14. The resultant tree is shown in part i of Figure G. Note that the key value 5 is maintained in the internal node.

Figure G (i) The B^+-tree that results after the deletion of key 5 from the tree of Example 3.14. (ii) The B^+-tree after the deletion of key 7.

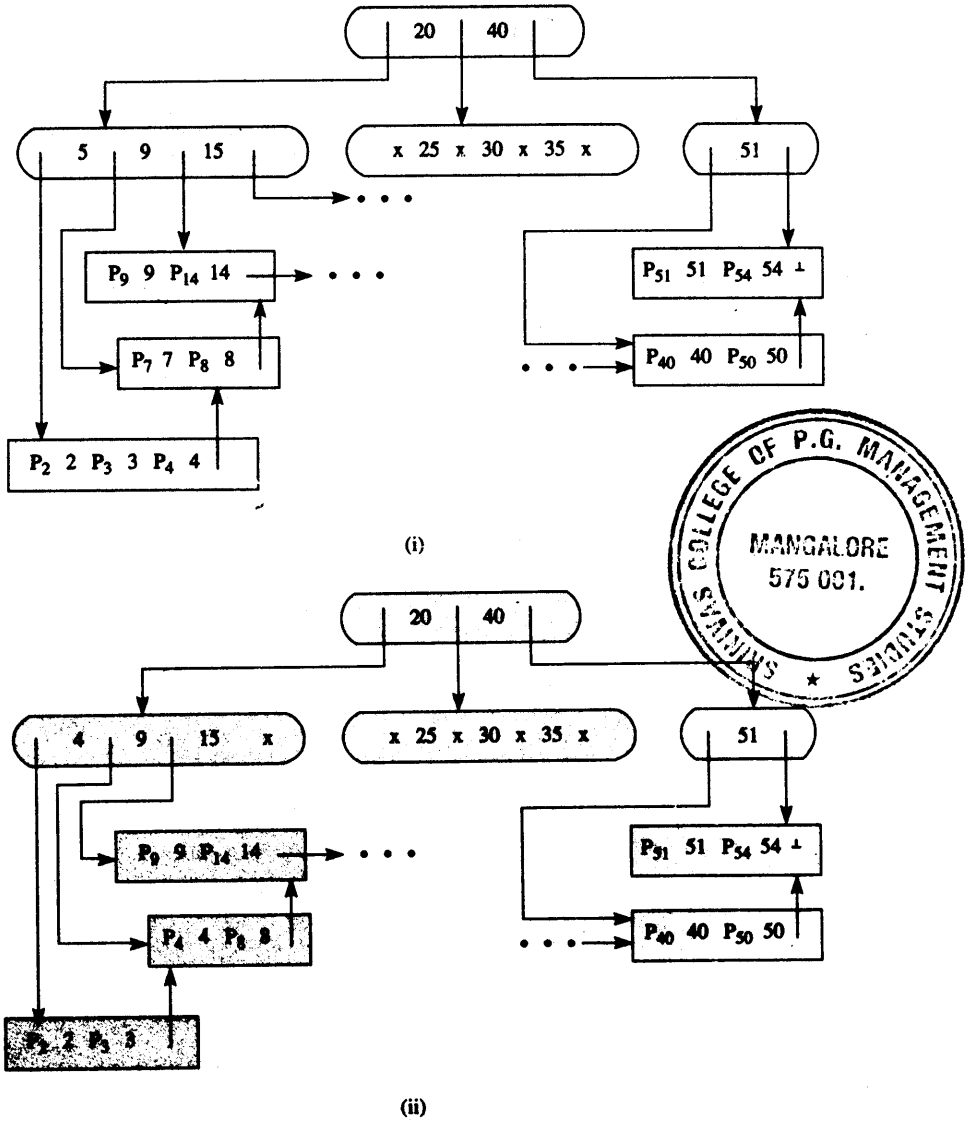
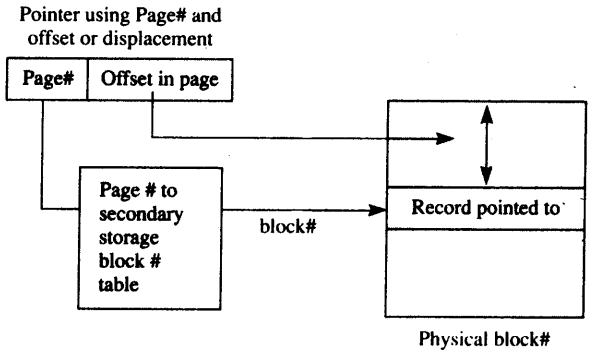


Figure 3.27 Deriving address for clustered storage.



would necessitate changes to their values. Then should all pointers be implemented as logical addresses (i.e., by some key of the record)? This requires that there exist a mapping scheme from the key to the physical address. If this mapping is provided by an index, it entails additional accesses for each logical pointer access. Similarly, this applies for the hashing of the key values, except in unlikely hash functions that produce no collisions.

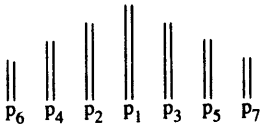
It is possible to use addresses based on page or bucket numbers and displacement within page where each page or bucket contains a set of blocks, i.e., a page contains a large number of records. The physical location of each of these file pages can be stored in a small table; this table can be brought into main memory when the file is in use. The displacement is used as a modifier, and the logical to physical address mapping can be done as shown in Figure 3.27 without additional secondary storage accesses. When the file is moved around on the disks, the only requirement is that the cluster of records in the page are moved together so that their displacements are not altered.

3.9 Record Placement

We began this chapter by stating that the time needed to access data on secondary storage could be optimized by minimizing the component of response time that we called the access time. In the sections above, we considered how access is facilitated by employing certain file organizations. The primary consideration in all organizations is access to the next or some particular logical record. Our main concern has been with access methods. We stated that the response time could also be optimized by suitable **record placement.**

A suitable placement strategy necessitates the knowledge or estimation of access frequencies or probabilities. We want the records to be placed in such a manner that the average head movement is minimal. It has been proven that the cost is minimal when the most frequent (or likely) records are grouped together in blocks and the blocks arranged such that the block access probabilities form an organ pipe arrangement. This type of arrangement results when we sequence block placement in non-

Figure 3.28 Organ pipe arrangement.



increasing access probability order. We first place the block with the highest access probability at some point and the other blocks in nonincreasing access probability order, alternately to the left or to the right of the already-placed blocks. Let us consider, for instance, n blocks and let the access probability of the *i*th block be p_i, where $p_1 \geq p_2 \geq \ldots \geq p_n$. The resultant optimal placement of blocks is shown in Figure 3.28. The optimal record placement strategy is applicable, even to the file organizations considered earlier in this chapter.

3.10 Concluding Remarks

In this chapter we looked at some common file organizations. They occur quite often in systems and applications work. As we have seen, no one organization can efficiently support all applications and types of access. It may be necessary to design a file that supports different organizations for different key fields, depending on the application requirements. However, it is not wise to design elaborate organizations for rare types of access. In file design, particular emphasis is placed on usage and factors of growth. We should also be aware of the space/time tradeoff in file design. Speeding up some accesses is always accompanied by increased storage demands. The simplest serial file has minimal wastage of storage space or overheads. However, as we have seen, access and updates are expensive. The other file organizations improve performance of certain operations, but require additional storage space.

In the index-sequential file the records are ordered with respect to the primary key. In this way it is possible to allow random and sequential access to any record. An index-sequential scheme, however, becomes inefficient if there are a large number of insertions and consequent overflows, and it requires periodic maintenance. For a file that is growing rapidly, index-sequential organization may be inappropriate. B^+-tree indexing, with its built-in maintenance, allows growth without the penalty of performance degradation. Both types of indexing allow random search followed by sequential search. However, the records in the case of the B^+-tree file may not be clustered and therefore it is possible that a disk access may be required to retrieve each record. Range queries, wherein records have a range of key values, can be handled by these file organizations.

With direct access supported by hashing, random access to any record is obtained in a fixed time but if the records are not clustered on the key used for hashing, sequential or range queries can only be handled as a series of independent requests. The hashing function maps a key value into a bucket address. With a good hashing function sequential keys need not be mapped to the same or consecutive buckets. However, having obtained the first bucket address, we have no way of knowing which bucket will contain the next key.

homogeneous records
primary block
overflow block
update operations
sequential file
serial file
index-sequential file
direct file
nonkeyed sequential file
transaction file
old master file
new master file
index file
data file
implicit index
explicit index
limit indexing
block
bucket
sequential index key
index-sequential search
track index
skip-sequential processing
virtual storage access method (VSAM)
control interval
control area
hashing
collision
dynamic hashing
extendable hashing
inverted index file
inverted file
multilist file
cellular list
cell
ring file
leaf node
m-order tree
B^+-tree
overflow
redistribution
B-tree
failure nodes
record placement

Exercises

3.1 Access methods are measured by access and storage efficiencies. Define each term and its major objectives. Which is the most important consideration in a batch environment? In an online environment? Give reasons.

3.2 Discuss the differences between the following file organizations:

(a) serial
(b) index-sequential
(c) hashed
(d) inverted

Compare their storage and access efficiencies. To what type of application is each of the organizations suited?

3.3 We are given a file of 1 million records, each record being 200 bytes long, of which 10 bytes are for the key field. A physical block is 1000 bytes long and block addresses are 5 bytes long.

(a) Using a hashed file organization with 1000 buckets, calculate the bucket size in blocks. Assume all blocks contain the average number of records. What is the average number of accesses needed to search for a record that exists in the file?
(b) Using an index-sequential file with one level of indexing and assuming that all file blocks are as full as possible (with no overflow), how many blocks are needed for the index? If we employ a binary search on the index, how many accesses are required on average to find a record?
(c) If we use a B^+-tree and assume that all blocks are as full as possible, how many index blocks are needed? What is the height of the tree?
(d) Repeat part (c) if all blocks are half full.

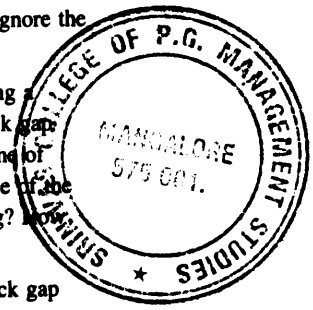
3.4 We are given a file of 10 million records, each record being 100 bytes long, of which 5 bytes are for the key field. A physical block is 10000 bytes long and block addresses are 5 bytes long.

(a) Using a hashed file organization with 10,000 buckets, calculate the bucket size in blocks; assume all buckets are half full. What is the average number of accesses needed to search for a record that exists in the file?

(b) Using an index-sequential file with two levels of indexing and assuming that all data blocks are half full, how many blocks are needed for the index? If we employ a binary search on the index, how many accesses are required on average to find a record?

(c) If we use a B^+-tree of order 500, how many index blocks are needed? What is the height of the tree? How many disk accesses are required to find a record?

3.5 A file of 1,000,000 fixed-length records, each 200 bytes long, is stored on a magnetic tape. The tape handler characteristics are a 100KB/sec transfer rate and a start/stop time of 25 msec. Compare the time required to read all the records if the block size is chosen as (a) 5000 bytes, (b) 50,000 bytes and the tape has to be stopped after reading a block. Ignore the time used for processing after a block is read.



3.6 Records of 250 bytes are stored in blocks with a blocking factor of 20. A drive using a 3600-foot tape having a recording density of 6400 bpi (bytes per inch), an interblock gap size of 0.5 inch, a read/write speed of 200 kilobytes per second, and a start-stop time of 0.010 seconds is being used. How many records can the tape hold? What percentage of the tape is wasted? How long will it take to read the file from the tape without stopping? How much time is spent in reading the file if only one block is read at a time?

3.7 Given a record length of 32 bytes, a recording density of 1600 bpi, and an interblock gap size of 0.6 inch, calculate the blocking factor to have 80% of a 1600-foot tape holding data.

3.8 A file of 100,000 fixed-length records, each 100 bytes long, is stored on a magnetic tape. The tape handler characteristics are a 40KB/sec transfer rate and a start/stop time of 20 msec. The file is recorded at 1600 bpi and the interblock gap is 1/2 inch. Find the length of the tape required and compare the times required to read all the records if the block size is chosen as (a) 100 bytes, and (b) 10,000 bytes.

3.9 Consider a hash function h(k) = k mod 17 for a direct access file using extendable hashing. Assume that the bucket capacity is four records. Show the structure of the file including the bucket address table after the insertion of the following records: 87,13, 53, 82, 48, 921, 872, 284, 36, 128, 172.

3.10 In a multilist organization, give efficient algorithms to process the following queries:

(a) get all records with $Key_1 = x$ and $Key_2 = y$

(b) get all records with $Key_1 = x$ or $Key_2 = y$

If a ring organization is used instead, what complications are introduced into the processing of the above queries?

3.11 The following file contains student records. The Rec# is the address used to retrieve the record using a direct access function on the primary key (Id).

(a) Generate a directory for a multilist that has indexes for Dept, Advisor, and Status. Fill in the appropriate record number values in the Ptr field provided within the file.

(b) Using this multilist directory and the data file, indicate how you will answer the query to retrieve all records for students who are in the COMP department, or who have SMITH F. as an advisor, or whose status is F2, without accessing redundant records.

(c) Using the above data and assuming that there are three records per cell, generate a directory for a cellular multilist file with entries for Dept, Advisor, and Status.

Rec#	Name	Id	Dept	Ptr	Advisor	Ptr	Status	Ptr
1	MICROSLAW Kalik	3634592	COMP		SMITH F.		F2	
2	PASSASLO Joseph	3894336	PHYS		JONES A.		F3	
3	PRONOVOST Pierre	6888954	ELEC		WAGNER B.		I1	
4	LOANNIDES Lambi	3518445	CHEM		ACIAN R.		F3	
5	MACIOCIA Charles	7564019	ENGL		BROST A.		P2	
6	CHO BYUNG Chu	2566984	CHEM		JONES A.		F2	
7	CANNON Joe	7868286	PHYS		JONES A.		F3	
8	BERGERON Daniel	2736849	COMP		JONES A.		I2	
9	ABOND Daniel	7382943	ELEC		WEGNER B.		I3	
10	HAMMERBELL Abraham	6792839	COMP		SMITH F.		P2	
11	LANGEVIN Joseph	2768736	ENGL		NEWELL J.		P3	
12	PELLERIN George	6689184	COMP		WEGNER B.		F2	
13	ROBERT Louis	3707939	COMP		MARTIN R.		P1	
14	SHARPE George	9877546	CHEM		SMITH F.		I2	
15	PETIT Guy	2742619	ELEC		SMITH F.		I3	

3.12 What are the advantages and disadvantages of the index-sequential file?

3.13 Consider a cylinder of an index-sequential file as shown below. Only the key values are shown. The following changes are made to this cylinder:

add ID, **add** FW, **add** KP, **delete** FV, **add** FU, **delete** IQ, **add** JK, **add** IS, add IT, add JR

Here **add** indicates that a record is to be inserted into the file and **delete** indicates that the record is to be deleted from the file. Only the key values are given. The changes occur in the order specified. ⊥ indicates null pointers.

	HA	Block1	Block2	Block3	Block4	Block5	Block6
	2900	Tr.Index	FP	FR	FT	FV	FZ
P	2901	GB	GE	GH	GK	GM	GR
r	2902	GV	GY	HB	HC	HF	HI
i A	2903	HL	HO	HQ	HT	HX	IA
m r	2904	IC	IG	IJ	IM	IQ	IY
e e	2905	IZ	JB	JF	JJ	JN	JQ
a	2906	KA	KD	KG	KL	KO	KS
	2907	KT	KV	KY	KZ	LB	LF
Overflow	2908	⊥	⊥	⊥	⊥	⊥	⊥
Area	2909	⊥	⊥	⊥	⊥	⊥	⊥

Show the initial and final values of the track index. Also show the contents of the cylinder after all of the above changes have been made.