Before we go on to more formal material, let us examine the attribute *Birth_Date* in greater detail. We are all well aware that every date is not a legal date, eg, January 33, 1959. Furthermore, a future date could not be a valid value for this file. If we want to restrict the values that can be assigned to some attribute for a given record, we must define the set of legal values for that attribute. We refer to this set of legal values as the domain of an attribute.

Until now, we have been informally discussing certain concepts related to data organization on secondary storage. We shall now set these concepts in more formal terms.

A file F is a collection or "bag" of records, that is, $F = \{r_1, r_2, \ldots, r_n\}$, where the r_i's are used to represent the records in a file F containing n records. File F, in general, is not a set of records because duplicate records may be permitted. A "bag" permits duplicate occurrences although it may be difficult to visualize many situations where this would occur.

As discussed earlier, an attribute is used to capture some characteristic or property of an entity. A record r_i is a set of <attribute (or field), value> pairs defined on the set of attributes $A = \{A_{i1}, \ldots, A_{im}\}$ over the set of corresponding domains $D = \{D_{i1}, \ldots, D_{im}\}$. It is not necessary that D_{ij} and D_{ik}, $j \neq k$, be distinct domains, as different attributes can be defined on the same domain.

The record r_i can be represented as the set $r_i = \{(A_{i1}, v_{i1}), \ldots, (A_{im}, v_{im})\}$, where each $v_{ij} \in D_{ij}$, for $j = 1, \ldots, m$. If every record of a file contains <attribute,value> pairs for the same set of attributes, the file is said to contain **homogeneous records.** If the attribute-value pairs are similarly ordered in all the records of the file, i.e., for all r_i, $(A_{11} = A_{21} = \ldots = A_{n1})$, $(A_{12} = A_{22} = \ldots A_{n2})$, $\ldots$, $(A_{1m} = A_{2m} = \ldots = A_{nm})$, then the fact that the attribute order is known can be used to achieve efficiency in record representation. It is in fact usual to represent a record using positional notation, i.e., $r_i = (v_{i1}, \ldots, v_{im})$, where the attributes are discerned from the position of the associated value. This is how we represented a record in Figure 2.15. The order of the attributes has no semantic importance. For a data record using positional notation to make sense, the mapping between position and attribute names must be known. Although the attribute name may not be specifically incorporated in the record, we can logically associate the appropriate attribute name with the values stored.

Example 3.2

In the above example, the attribute order in each record of the file is given as *Birth_Date*, followed by *Last_Name* and then *First_Name*. On an access to a record we are presented with a sequence of bytes that we map, logically, to our three attributes. The first k bytes represent the *Birth_Date*, the succeeding k′ bytes the *Last_Name*, and the remaining k″ bytes the *First_Name*. Given the sequence of bytes, their decoding mechanism, and the values k, k′, and k″, we can interpret the sequence of bytes that constitutes the record. ■

A file that contains nonhomogeneous records needs to store the attribute names (or identification codes) within the records.

3.1.3 Formal Specification of Storage of a File

All storage organizations are ultimately constituted from bytes. Let us call the set of all possible bytes BYTES. We can define an attribute value, a record, and a file in terms of a sequence of bytes. The length of a sequence, s, is written as #s. An informal treatment of sequences is given in Appendix 3.1 at the end of the text.

An attribute value or simply an attribute is some sequence of bytes[1]:

ATTRIBUTE : : = sequence of BYTES

The values for different attributes and also the different values for some attributes would not all be encoded using equal-length sequences. We have to specify the length of the sequence. For attributes that can accept variable-length sequences as values, we specify the minimum (#min) and maximum (#max) sequence lengths. Fixed-length values have #min = #max. Thus we have:

ATTRIBUTE : : = sequence of BYTES of length (#min . . #max)

A record is defined in terms of some bag of attribute values. Physically a record is defined as:

RECORD : : = sequence of BYTES

However, logically we think of a record as:

RECORD : : = sequence of ATTRIBUTES

Records are stored on the physical medium in blocks. For simplicity we assume fixed-length blocks. Then we have:

FL_BLOCK : : = array [1 . . BLOCK_SIZE] of BYTES
FL_BLOCK : : = sequence of RECORD

The first definition pertains to physical blocks and the second to logical blocks. The first definition allows a logical record to span over physical blocks.

Similarly, we define the file:

FILE : : = sequence of FL_BLOCK

(Note that from the definition of FL_BLOCK, we may physically consider a file to be just a sequence of BYTES, or logically as a sequence of RECORD.)

The emphasis on "sequence of BYTES" is deliberate, for this precisely represents the fact that all data is stored in the form of bytes (or bits). This is important, too, for if we have a sequence and we wish to map it into a given logical structure, we should know (1) the beginning point of the sequence and (2) a definition of the logical structure into sequences of bytes. This has implications for searching without transferring data between the secondary medium and main memory. A processing element associated with the read/write head of a storage device can decide that it has located some desired sequence only if it knows the starting point. These are encoded or physically embedded in actual storage devices.

---

[1] Note that some attributes are Boolean and need less than one byte; however, many implementations use an entire byte to tore a single Boolean valued attribute.

A number of initialization operations must be performed by the file manager before the initial access is made to a file. This is usually done by issuing an open file (or in Pascal, a reset/rewrite command). This initiates some internal housekeeping by the file manager. The creation of a number of buffers of appropriate size and initializing pointers (one each for blocks and records within blocks) would be necessary. We shall name these pointers BLOCK_PTR and RECORD_PTR, respectively.

Assume that we have issued an open command for some file; then we can assume that BLOCK_PTR = 1 and RECORD_PTR = 1. The number of blocks in the file is given by #file_block and the number of records in the block BLOCK_PTR is given by #record(BLOCK_PTR). Algorithm 3.1 for get_record follows. Here we ensure that we do not attempt to access a record past the last record of a block or access a block past the last block in the file. Provided that the pointers are correctly set, the next record is made available. If we had already accessed the last record in the block, the block pointer is incremented and the record pointer within the block is reset to 1. (Note that this algorithm is much more simple than what happens in reality. First, it is implicit that somehow the data from the secondary storage is already available. In practice, the blocks would have to be read off the secondary storage. Second, the sizes (# . . .) are made available from some system record

Algorithm 3.1

Algorithm to Get a Record

Input: ...ized values for BLOCK_PTR, RECORD_PTR, #file_block(number of ... to the file) and #record(BLOCK_PTR) (number of records in the ...

Output: ... end_of_file error condition

...LOCK_PTR) *then begin*
...PTR,RECORD_PTR)
...ORD_PTR + 1

FIRST: $F \rightarrow r_1$ NEXT: $(F,r_i) \rightarrow r_{i+1}$ if $i \neq n$
LAST: $F \rightarrow r_n$ NEXT: $(F,r_n) \rightarrow$ ERROR (end_of_file)
PREVIOUS: $(F,r_i) \rightarrow r_{i-1}$ if $i > 1$
PREVIOUS: $(F,r_1) \rightarrow$ ERROR (beginning_of_file)

We sometimes do not require access to every attribute of a record, simply to a subset. Similarly, we may access only those records that satisfy some given condition. In general, we can specify access or retrieve operations on a file as:

<target_list | qualification>

where the target_list is the list of attributes for which the values of records satisfying the specified qualification clause are to be retrieved. The qualification clause is a Boolean expression, a sequence of terms connected with Boolean operators as defined below:

<qualification> ::= <term> [<Boolean_operator> <qualification>]
<term> ::= [<negation>] <attribute> <relational operator> <constant>
<relational operator> ::= '='|'≠'|'>'|'≥'|'<'|'≤'
<Boolean operator> ::= AND|OR
<negation> ::= NOT

In principle, we can retrieve records from a file based on the value of any attribute. However, it is common to retrieve records based on some subset of the attributes, designated as key attributes. The file is organized so that retrievals based on these key attributes will be efficient. Remember that certain attributes, **primary keys,** may be used to uniquely identify records in a file, while other attributes, secondary or nonprimary keys, can identify a set of records.

Example 3.3

In Figure 3.6, assume that the names are unique, i.e., the name can be used to identify a record. As such, name would be the primary key for the GPA file.

In the Birth_Date file we recorded information about the birth dates of persons we know. We can assume that the combination of *First_Name* and *Last_Name* uniquely identifies a record, i.e., that every person we know has a unique name. Suppose that a person's *First_Name* and *Birth_Date* also uniquely identify a record. In other words, some persons have a common birth date and a few of the persons have the same first name, but no two persons with the same first name have the same birth date (at least for this example). Thus we can assume that either one of the two combinations <*First_Name, Last_Name*> or <*First_Name, Birth_Date*> can be used as a primary key.

Let us choose the <*First_Name, Last_Name*> combination as our primary key. We can choose any or all of the attributes *First_Name, Last_Name,* and *Birth_Date* as secondary keys. For simplicity, we choose *Birth_Date*. Now since we allow the possibility of more than one person with the same birth date, we expect to retrieve zero, one, or several records when we use *Birth_Date* to access this file. Accessing the file using the <*First_Name, Last_Name*> combination would retrieve at most one record. ■

The data contained within the file may have to be changed. The changes could be the addition (or insertion) of new records, the removal (or deletion) of an existing one, or the changing (or modifying) of some of the contents of an existing one. The insertion, deletion, and modification operations are collectively known as **update operations.** Update operations can also be expressed in terms of *target_list* and *qualification_list*. The *target_list* permits assignment statements in the form of attribute := expression. Insert operations have an empty qualification clause.

An update is a mapping from one (old) version of a file to another (new) version of it, i.e., $F \rightarrow F'$. Assume that #F_record represents the number of records in the file F. An update may include any of the following four possible procedures:

U_1. Insert records in their proper logical sequence. Let $F = \{r_1, \ldots, r_{k-1}, r_{k+1}, \ldots\}$ and #F_record = n, then

$$\text{INSERT: } (F, r_k) \rightarrow F'$$

where $F' = \{r_1, \ldots, r_{k-1}, r_k, r_{k+1}, \ldots\}$ and #F'_record = n+1.

The operation is accomplished logically by copying records $r_1, \ldots, r_{k-1}$ into file F', then storing record r_k and copying the remaining records, $r_k, r_{k+1}, \ldots$ into F'.

U_2. Delete one or more existing records from the file. Let $F = \{r_1, \ldots, r_{k-1}, r_k, r_{k+1}, \ldots\}$ and #F_record = n, then

$$\text{DELETE : } (F, r_k) \rightarrow F'$$

where $F' = \{r_1, \ldots, r_{k-1}, r_{k+1}, \ldots\}$ and #F'_record = n − 1.

The operation is accomplished logically by copying records $r_1 \ldots, r_{k-1}$ into file F', ignoring record r_k, and then copying the remaining records, $r_k, r_{k+1}, \ldots$ into F'.

U_3 Modify the data values in some existing record. This is akin to deleting record r_i and inserting r_i', where r_i' is the modified record.

$$\text{MODIFY: } (F, r_i) \rightarrow F'$$

Let $F = \{r_1, \ldots, r_i, \ldots, r_{k-1}, r_k, r_{k+1}, \ldots\}$ and
$F' = \{r_1, \ldots, r_i', \ldots, r_{k-1}, r_k, r_{k+1}, \ldots\}$

The operation is accomplished logically by copying records $r_1, \ldots, r_{i-1}$ into the file F', modifying record r_i to r_i' and copying it to file F', and then copying the remaining records, $r_{i+1}, \ldots$ into F'. Note that the relative positions of the records remain unchanged.

U_4. Modify the data values in existing records (it is common to assume that the record length remains the same).

$$\text{MODIFY: } (F, A_i, v_i, v_i') \rightarrow F'$$

This modifies data values in *all* records that have value v_i for the attribute A_i. The operation is accomplished logically as follows: Copy a record r_j, such that the value of attribute $A_i \neq v_i$, into the file F'; or modify a record r_k, such that the value of attribute $A_i = v_i$, to r_k' where the value of attribute A_i is modified to v_i' and copy the modified record to file F'.

Remember that update operations may also cause exceptions, such as when we try to delete or modify a nonexistent record. In most applications insertion of a

typically used to maintain records chronologically; one such application is to record transactions.

3.3 Sequential Files

In a sequential file, records are maintained in the logical sequence of their primary key values. The processing of a sequential file is conceptually simple but inefficient for random access. However, if access to the file is strictly sequential, a sequential file is suitable. A sequential file could be stored on a sequential storage device such as a magnetic tape.

Search for a given record in a sequential file requires, on average, access to half the records in the file. Consider a system where the file is stored on a direct access device such as a disk. Suppose the key value is separated from the rest of the record and a pointer is used to indicate the location of the record. In such a system, the device may scan over the key values at rotation speeds and only read in the desired record. A binary[2] or logarithmic search technique may also be used to search for a record. In this method, the cylinder on which the required record is stored is located by a series of decreasing head movements. The search, having been localized to a cylinder, may require the reading of half the tracks, on average, in the case where keys are embedded in the physical records, or require only a scan over the tracks in the case where keys are also stored separately.

Updating usually requires the creation of a new file. To maintain file sequence, records are copied to the point where amendment is required. The changes are then made and copied into the new file. Following this, the remaining records in the original file are copied to the new file. This method of updating a sequential file creates an automatic backup copy. It permits updates of the type U_1 through U_4.

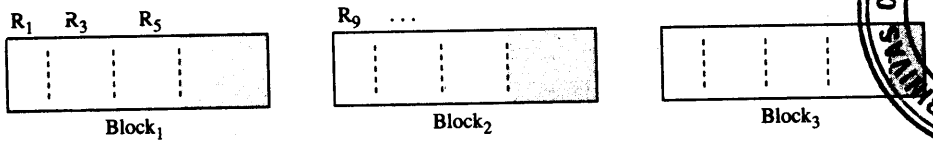
Addition can be handled in a manner similar to updating. Adding a record necessitates the shifting of all records from the appropriate point to the end of file to create space for the new record. Inversely, deletion of a record requires a compression of the file space, achieved by the shifting of records. Changes to an existing record may also require shifting if the record size expands or shrinks.

The basic advantage offered by a sequential file is the ease of access to the next record, the simplicity of organization, and the absence of auxiliary data structures. However, replies to simple queries are time consuming for large files. Updates, as seen above, usually require the creation of a new file. A single update is an expensive proposition if a new file must be created. To reduce the cost per update, all such requests are batched, sorted in the order of the sequential file, and then used to update the sequential file in a single pass. Such a file, containing the updates to be made to a sequential file, is sometimes referred to as a **transaction file.**

In the batched mode of updating, a transaction file of update records is made and then sorted in the sequence of the sequential file. The update process requires the examination of each individual record in the original sequential file (the **old master file**). Records requiring no changes are copied directly to a new file (the **new**

[2]Factors such as seek and latency time rule out the use of binary search in favor of some form of indexing scheme for disk-based files.

Figure 3.8 A file with empty spaces for record insertions (the figure shows some fixed-length records and unused space).



master file); records requiring one or more changes are written into the new master file only after all necessary changes have been made. Insertions of new records are made in the proper sequence: They are written into the new master file at the appropriate place. Records to be deleted are not copied to the new master file. A big advantage of this method of update is the creation of an automatic backup copy. The new master file can always be recreated by processing the old master file and the transaction file.

A possible method of reducing the creation of a new file at each update run is to create the original file with "holes" (space left for the addition of new records, as shown in Figure 3.8). As such, if a block could hold k records, then at initial creation it is made to contain only L * k records, where $0 < L \leq 1$ is known as the loading factor. Additional space may also be earmarked for records that may "overflow" their blocks, e.g., If the record r_i logically belongs to block B_i but the physical block B_i does not contain the requisite free space. This additional free space is known as the overflow area. A similar technique is employed in index-sequential files.

3.4 Index-Sequential Files

The retrieval of a record from a sequential file, on average, requires access to half the records in the file, making such enquiries not only inefficient but very time consuming for large files. To improve the query response time of a sequential file, a type of indexing technique can be added.

An index is a set of <key, address> pairs. Indexing associates a set of objects to a set of orderable quantities, which are usually smaller in number or their properties provide a mechanism for faster search. The purpose of indexing is to expedite the search process. Indexes created from a sequential (or sorted) set of primary keys are referred to as index sequential. Although the indices and the data blocks are held together physically, we distinguish between them logically. We shall use the term **index file** to describe the indexes and **data file** to refer to the data records. The index is usually small enough to be read into the processor memory.

A sequential (or sorted on primary keys) file that is indexed is called an index-sequential file. The index provides for random access to records, while the sequential nature of the file provides easy access to the subsequent records as well as sequential processing. An additional feature of this file system is the overflow area. This feature provides additional space for record addition without necessitating the creation of a new file.

If the group corresponding to the first index key greater than K_r is, let us say, G_s, then the logical position of record corresponding to the key K_r is in group G_s. This is because K_r is greater than the largest key in group G_{s-1} but smaller than or equal to the largest key in group G_s. The key K_r is then compared to the keys in the group G_s to find a match.

This search procedure based on a set of ordered indexes, (the largest keys of different groups of sorted keys) is called an **index-sequential search.** It is shown in Algorithm 3.2.

In Algorithm 3.2 we assume that the index is available in memory and the entries are INDEXKEY and ADDRESS. This first entry in the index gives us the first sequential index key value and the location address of the associated block. We compare the given search key value with that of successive index key value entries until we get to the desired entry. This would be a block suitable for holding a record with the search key value. LOCATION returns the address of the block to which the record corresponding to SEARCHKEY belongs (logically).

In some systems, instead of the largest keys of the different groups being maintained in the sequential index, the smallest keys are kept. This requires that the key K_r be compared with the group keys until the group with the key $G_i > K_r$ is located. Then K_r may be contained in the preceding group, G_{i-1}.

Number of Comparisons

Assume that the groups are of the same size, i.e., $s_1 = s_2 = \ldots = s_m = s$. Then the number of records $n = m*s$ where there are m groups. In every group, more than one key value may exist; therefore, when searching for a record with a given key value, we have to check this key value against those of the records in the group. The number of comparisons associated with index-sequential search for different keys is presented in Figure 3.11. Part a of the figure shows the index and the number of

Algorithm 3.2

Index-Sequential Search Algorithm

Input: index table, and SEARCHKEY, the key of record to be retrieved

Output: The address of block for the record with key equal to SEARCHKEY

{Assumption: The last index entry has a key value that cannot be exceeded}

get first index entry, <INDEXKEY, ADDRESS>
while INDEXKEY < SEARCHKEY *do*
get next index entry
LOCATION := ADDRESS

{the record associated with the SEARCHKEY logically belongs in the block with the address LOCATION, and this address can be used to retrieve, insert, delete, or modify the record.}

Figure 3.11 Index key comparisons.

Index	*Address of Block*	*Number of Comparisons*
K_s	A_1	1
K_{2s}	A_2	2
.	.	.
.	.	.
K_n	A_m	m

(a)

Block at Address	*Record Key*	*Number of Comparisons*	
A_1	K_1	1	(2)
	K_2	2	(3)
		.	.
	K_s	s	(s+1)
A_2	K_{s+1}	1	(3)
	K_{s+2}	2	(4)
	.	.	.
	.	.	.
A_m	.	.	.
	K_n	s	(s+m)

(b)

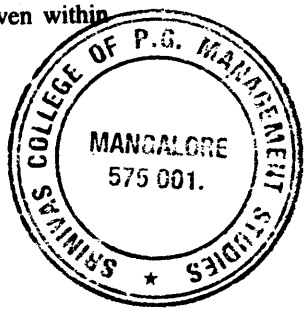
comparisons required to sequentially search for a key in the index. Part b indicates the block structure and the number of comparisons required for the sequential search for a record with a given key. The total number of comparisons made in searching for a key is the sum of the comparisons for the index and the block, given within parentheses in Figure 3.11b.

For key K_i the total number of comparisons is given by:

$$\lceil i/s \rceil + (i - 1) \bmod s + 1$$

and the average number of comparisons is given by

$$1 + (m + s)/2$$



Example 3.5

Assume a file of 10,000 records distributed over 100 blocks, i.e., every block has 100 records. Also assume that every record is equally likely to be accessed. In trying to locate a particular record, we first examine the index, which is assumed to be within a single block. To locate the block containing the required record, we have to examine each index entry. The number of comparisons required are:

Search for	*# Comparisons*
First entry	1
Second entry	2
Third entry	3
.	
.	
.	
99th entry	99
100th entry	100

The total number of comparisons made is 100 * (101)/2 = 5050 and the average number of comparisons per access is 50.5. By similar reasoning, we know that the average number of comparisons required for the actual record from the data block (it also contains 100 entries) is also 50.5. Therefore, the average number of comparisons required is 101. This value agrees with the value calculated using the expression 1 + (m + s)/2, since in our example the block size, s, is 100 and the number of blocks, m, is 100. ■

It is normal to organize a file with several logical records per track (we can even consider a lower division of a track into a number of sectors and assign several logical records per sector). If the records are held in key sequence, it is sufficient to index only the highest record key within each track (or sector). The index entries, then, consist of <track number, highest key in track> pairs. A record (with a given search key) is located by reading the index into main memory and comparing its key with the index entries to locate the track. The record is then searched for within the track.

At the beginning of this chapter we abstracted a disk as a two dimensional array with tracks and blocks (sectors). Instead of labeling all the tracks uniquely, we can group them in sets. One such grouping is formed around cylinders. Let c be the number of cylinders on which n records are organized in an indexed sequential organization. Each cylinder contains m tracks for storing records and each track contains s records. Let us also assume that n = cms. Assuming that access to all records is equally likely, the average number of comparisons is given by (c + m + s + 3)/2.

Example 3.6

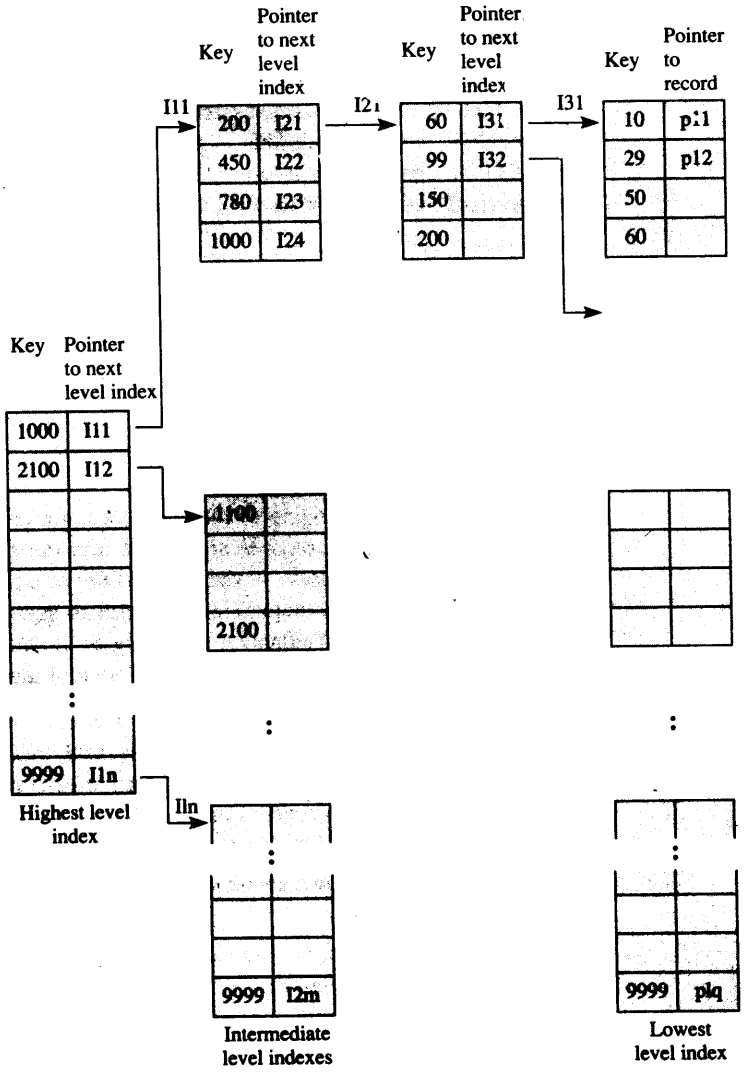
Assume that a file occupies 100 cylinders of 20 tracks each. Each track holds 20 records. Then the average number of comparisons to locate a given record is (100 + 20 + 20 + 3)/2 = 71.5. ■

The above expressions are for average number of comparisons. They do not indicate the number of disk accesses made in the retrieval of a record. Expressions for disk accesses are given in Section 3.4.4.

3.4.3 Multilevel Indexing Schemes: Basic Technique

In a full indexing scheme, the address of every record is maintained in the index. For a small file, this index would be small and can be processed very efficiently in main memory. For a large file, the index's size would pose problems. It is possible to create a hierarchy of indexes with the lowest level index pointing to the records, while the higher level indexes point to the indexes below them (Figure 3.12). The higher level indices are small and can be moved to main memory, allowing the search to be localized to one of the larger lower level indices.

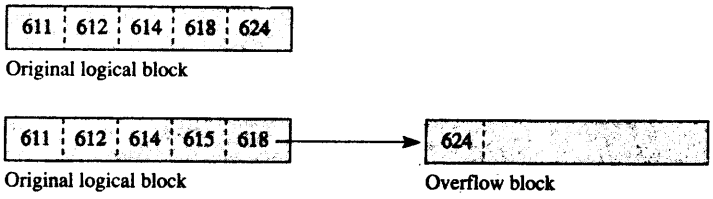
Figure 3.12 Hierarchy of indexes.



The lowest level index consists of the <key, address> pair for each record in the file; this is costly in terms of space. Updates of records require changes to the index file as well as the data file. Insertion of a record requires that its <key, address> pair be inserted in the index at the correct point, while deletion of a record requires that the <key, address> pair be removed from the index. Therefore, maintenance of the index is also expensive. In the simplest case, updates of variable length records require that changes be made to the address field of the record entry. In a variation of this scheme, the address value in the lowest level index entry points to a block of records and the key value represents the highest key value of records in this block. Another variation of this scheme is described in the next section.

3.4.4 Structure of Index Sequential Files

An index-sequential file consists of the data plus one or more levels of indexes. When inserting a record, we have to maintain the sequence of records and this may necessitate shifting subsequent records. For a large file this is a costly and inefficient process. Instead, the records that overflow their logical area are shifted into a designated overflow area and a pointer is provided in the logical area or associated index entry points to the overflow location. This is illustrated below. Record 615 is inserted in the original logical block causing a record to be moved to an overflow block.



Multiple records belonging to the same logical area may be chained to maintain logical sequencing. When records are forced into the overflow areas as a result of insertion, the insertion process is simplified, but the search time is increased. Deletion of records from index-sequential files creates logical gaps; the records are not physically removed but only flagged as having been deleted. If there were a number of deletions, we may have a great amount of unused space.

An index-sequential file is therefore made up of the following components:

1. A primary data storage area. In certain systems this area may have unused spaces embedded within it to permit addition of records. It may also include records that have been marked as having been deleted.
2. Overflow area(s). This permits the addition of records to the files. A number of schemes exist for the incorporation of records in these areas into the expected logical sequence.
3. A hierarchy of indices. In a random enquiry or update, the physical location of the desired record is obtained by accessing these indices.

The primary data area contains the records written by the users' programs. The records are written in data blocks in ascending key sequence. These data blocks are in turn stored in ascending sequence in the primary data area. The data blocks are sequenced by the highest key of the logical records contained in them.

When using a disk device to store the index-sequential files, the data is stored on the cylinders, each of which is made up of a number of tracks. Some of these tracks are reserved for a prime data area and others are used for an overflow area associated with the prime data area on the cylinder.

A **track index** is written and maintained by the file system. Each cylinder of the index-sequential file has its own track index. The track index contains an entry for each prime data track in the cylinder as well as an entry to indicate if any records have overflowed from the track. Each prime track may be considered as a logical block.

Each track index entry is made up of the following items: