1. The address of the prime data track to which the entry refers.
2. The highest key of a record in the prime data track.
3. The highest key of a logical record in that data track, including records in the overflow areas (i.e., it is the highest key of an overflow record, if there were one or more, associated with that track).
4. The address of a record with the lowest key in the overflow area associated with that track (the address of the first record in the overflow chain).

Items 1 and 2 make up the normal track index entry and items 3 and 4 make up the overflow track index entry. If there were no overflow from a given track, items 2 and 3 would contain the same key value and item 4 would be set to a null value. If more than one record were required to be stored in an overflow area, these records will be chained so they can be reached from the first track overflow record. The structure of these track index entries for the cylinder is shown in Figure 3.13.

The address of the prime track entry in the normal track index does not change, nor does the highest key value of the logical block. The highest key value entry in the prime data track and the address of the first overflow record changes when a new record inserted in the prime data track causes an existing record to be bumped into the overflow area. (This is illustrated in Figure 3.14b). The last digit in the pointer refers to the record number on the track.

Figure 3.13 Typical cylinder organization.

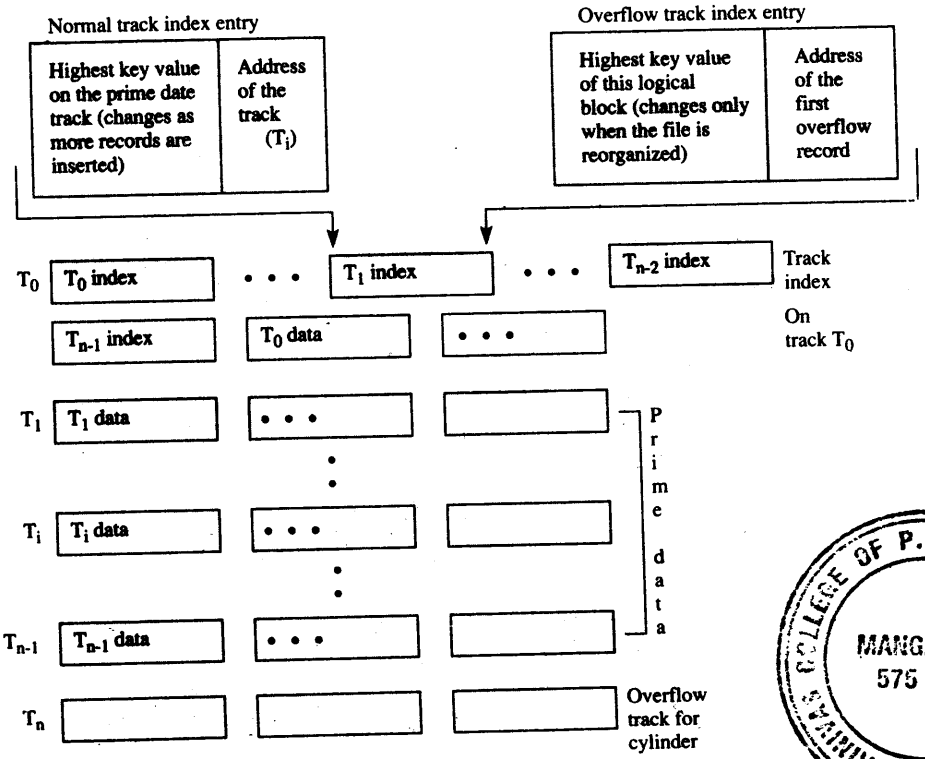
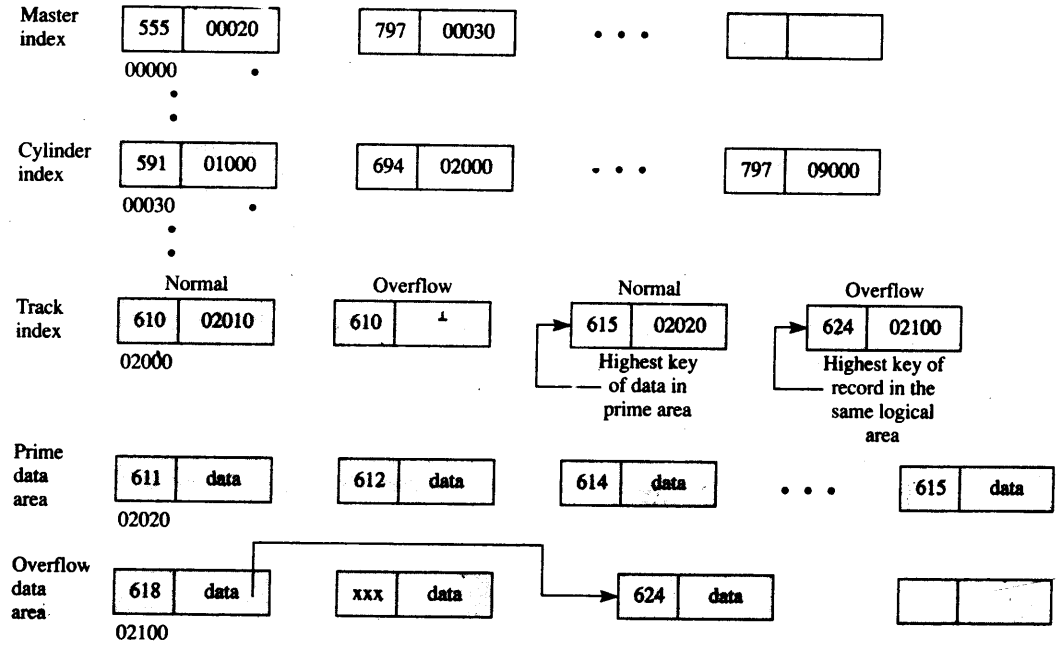
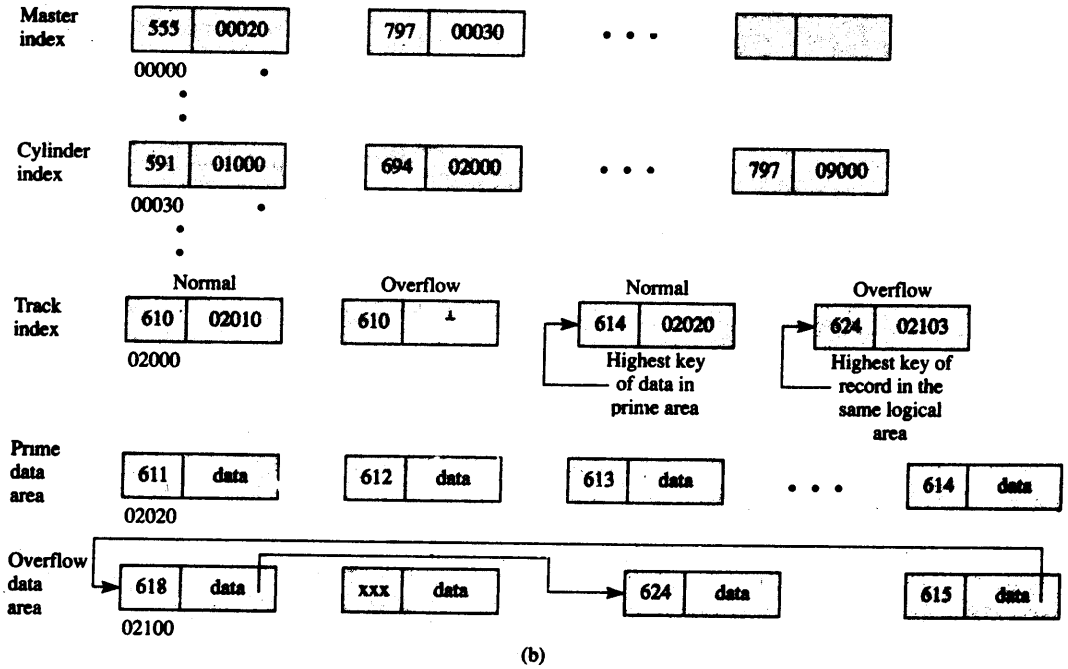


Figure 3.14 Structure of an index-sequential file.



(a)



(b)

The track index entries are used by the file system to determine the track address of a given logical record. The cylinder's overflow area is used to store records that are forced off the prime data track when new records are inserted. The records in the overflow area are unblocked and stored in the order of their insertions or placement rather than, in key sequence. The logical sequence of records is maintained by prefixing a sequence link to each logical record. The access to records in an overflow area is via these links and therefore inefficient.

A file with records in overflow areas and with deleted records needs reorganization. (It has to be recreated.) Deleted records are not physically deleted, but marked as having been deleted. The space and the contents are physically undisturbed. Such marked records are retrieved by the file manager, and it is up to the application program to ascertain their status. Normally, on subsequent insertions, a marked record is not forced off the prime area to the overflow area. The only exception is when a record having the highest key value in a cylinder is marked as deleted. When such a record is forced off the prime data track due to subsequent insertions, it is written in the overflow area. Additional independent overflow areas are used when a cylinder overflow area becomes full.

The structure of an index-sequential file, including the index, prime data, and overflow areas is shown in Figure 3.14. The address is given as the cylinder address followed by the track address, both being two digits in this example. The final digit represents a record number. The index area of the file contains a cylinder index (shown in the figure as being stored in record 0 of cylinder 00, track 03) and may contain a master index (shown on cylinder 00, track 00). It does not contain the track index (which is stored on the cylinders themselves). Each cylinder in the prime data area has an entry in the cylinder index. The entry contains the address of the track index in that cylinder and the highest key stored on that cylinder. The cylinder index is used by the file system to determine the cylinder on which a record might or should be and the address of the track index for the cylinder.

An index-sequential file may be updated in sequential or random mode. In sequential mode, the insertion of new records in their proper sequence (update type U_1) requires the creation of a new file, so it is performed only if a very large number of new records are being added. Under certain file managers new records may be added to the end of the file in sequential mode only if there is enough space in the prime data areas, not the overflow areas. In random mode all types of updates can be performed on an existing file.

Retrieval from an index-sequential file may be sequential or random. In sequential mode, it may be possible to specify both a start and an end point. This is very useful for processing grouped data. The records, including those in the overflow area, are available in their logical sequence. All pointers between the overflow records in a sequence are handled automatically by the FM to retain the logical sequence. In random processing mode any arbitrary record may be accessed. **Skip-sequential processing,** wherein the records not needed for processing are skipped over, is also made very easy and efficient. For low hit rates, whole tracks and cylinders may be skipped. In a sequential file in which keys are stored separately from data, it is possible to skip records but every key must be read.

Number of Disk Accesses

Let us now consider the number of disk accesses required when searching for a given record. We again assume uniform distribution of records within blocks, tracks, and

cylinders. Let there be L levels of indexing and the size of the index at a level, for instance j, be I_j blocks. Assume that each block is on a different track and access to a block consequently requires one disk access. Then, at each level, as we have assumed uniform distribution, we expect on average that half the number of index blocks will be accessed (in a sequential search). Therefore, the average total number of index blocks accessed is:

$$\sum_{j=1}^{L} \left\lceil \frac{I_j}{2} \right\rceil$$

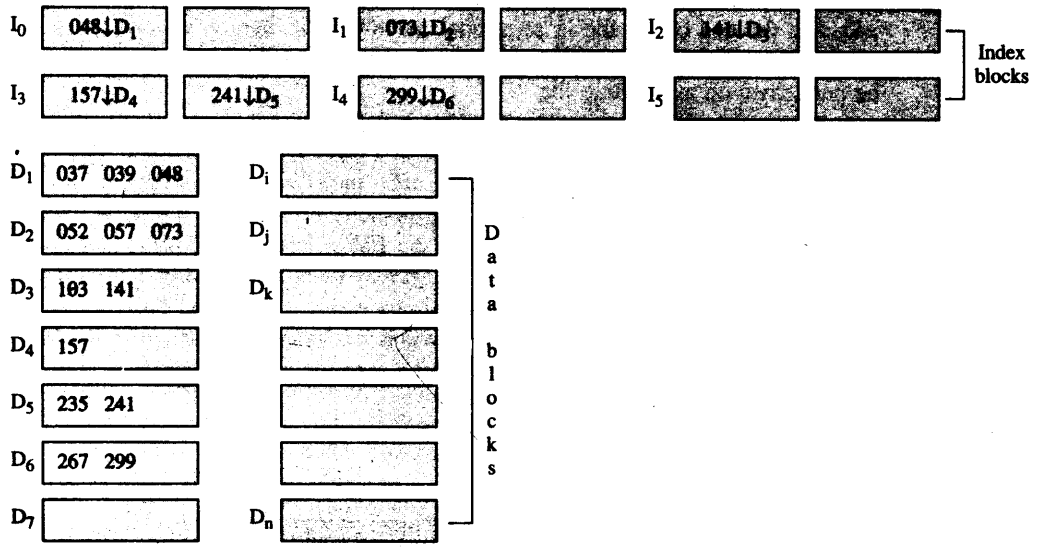
In addition, we need to access the block on which the actual record resides. If the record is in a prime area, only one block has to be accessed; otherwise number O ($\geq$0) overflow blocks are also accessed. As such, the total number of blocks accessed on average is:

$$\sum_{j=1}^{L} \left\lceil \frac{I_j}{2} \right\rceil + \begin{bmatrix} 1 \text{ if data on prime area} \\ 0 \text{ if data not on prime area} \end{bmatrix}$$

3.4.5 VSAM

The major disadvantage of the index-sequential organization is that as the file grows, performance deteriorates rapidly because of overflows and consequently there arises the need for periodic reorganization. Reorganization is an expensive process and the file becomes unavailable during reorganization. The **virtual storage access method (VSAM)** is IBM's advanced version of the index-sequential organization that avoids these disadvantages. The system is immune to the characteristics of the storage me-

Figure 3.15 Index and data blocks of a VSAM control interval.

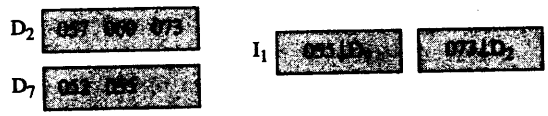


dium, which could be considered as a pool of blocks. The VSAM files are made up of two components: the index and data. However, unlike index-sequential organization, overflows are handled in a different manner. The VSAM index and data are assigned to distinct blocks of virtual storage called a **control interval.** To allow for growth, each time a data block overflows it is divided into two blocks and appropriate changes are made to the indexes to reflect this division.

Figure 3.15 shows the structure of a control interval of a VSAM file. The index block and the data blocks are included in a control interval. We can consider the control interval to serve the same purpose that the track does in the index-sequential organization. Higher level indices also exist in VSAM; however, these are not shown in Figure 3.15. The control interval contains a number of empty index and data blocks, which are used when a data block overflows. The index entry I_1 indicates that the highest key value of a record in data block D_2 is 73; the pointer to data block D_2 is indicated by $\downarrow D_2$. The method of handling overflow is illustrated in Example 3.7.

Example 3.7

Suppose the records to be added have the key values of 55 and 60. These records will logically be added into data block D_2. However, since D_2 has a block size of 4, only one record can be added without an overflow. The solution used in VSAM is to split the logical block D_2 into two blocks, let us say D_2 and D_7. The records are inserted in the correct logical sequence. Furthermore, the index entry I_1 is divided into two index entries as shown below:



■

In VSAM, a number of control intervals are grouped together into a **control area.** An index exists for each control area. A control interval can be viewed as a track and a control area as a cylinder of the index-sequential organization.

Each control interval also contains control information that can be used in conjunction with routines provided in VSAM to allow retrieval of records, using either the key value or the relative position of a record. The relative position can either be the relative position in bytes from the start of the file or, in the case of fixed-length records, the relative number of the record.

3.5 Direct File

In the index-sequential file organization considered in the previous sections, the mapping from the search-key value to the storage location is via index entries. In direct

middle region. The start and end digits are concatenated and the concatenated string of digits is added to the middle region digits. This new number, mod s, where s is the upper limit of the hash function, gives the bucket address:

123456	123456789012	654321

For the above key (converted to integer value if required) the end folding gives the two values to be added as: 123456654321 and 123456789012.

3. Square all or part of the key and take a part from the result. The whole or some defined part of the key is squared and a number of digits are selected from the square as being part of the hash result. A variation is the multiplicative scheme where one part of the key is multiplied by the remaining part and a number of digits are selected from the result.
4. Division. As stated in the beginning of this section, the key can be divided by a number, usually a prime, and the remainder is taken as the bucket address. A simple check with, for instance, a divisor of 100 tells us that the last two digits of any key will remain unchanged. In applications where keys may be in some multiples, this would produce a poor result. Therefore, division by a prime number is recommended. For many applications, division by odd numbers that have no divisors less than about 19 gives satisfactory results.

We can conclude from the above discussion that a number of possible methods for generating a hash function exist. In general it has been found that hash functions using division or multiplication perform quite well under most conditions.

Let us now consider the retrieve, insert, and delete operations using hashing to locate our records. Let K be the set of keys and A be the set of bucket addresses so that the hashing function h is a function from K to A. The hash value h(k) is the address of the bucket that contains the <key, address> pair for the record with key k. Here we assume the size of the bucket is chosen such that overflow would not occur. A special dummy record is always assumed to be the last record in each bucket and it is used in the search to indicate a failure. The bucket with address h(k) is examined for the <k, address> pair. If there is no match, the record with key k does not exist. If the operation was either a simple retrieval or a deletion, this results in a notfound message (or error condition). For insertions, the <k, address> pair is inserted in this bucket. The record is, of course, inserted in the file at the location given by the address. If the <k, address> pair exists, then for an insertion this would be an attempt to insert a duplicate record (which may or may not be permitted in the application). In the case of a deletion, we would delete the actual record as well as the bucket entry. Algorithm 3.3 specifies the sequence of steps.

As mentioned earlier, we require that the hash function uniformly distribute the keys in the buckets. This seems to be a reasonable approach until we examine certain details more closely. Although we may know the range of key values, do we also know their distribution characteristics? Note that not all key values are likely to occur. Different distributions require different hash functions to satisfy the uniformity requirement. The hash value is also required to lie within the range of addresses for the buckets, i.e., this range is prespecified. These considerations preclude any changes to the hash function once it has been implemented. Over a volatile file we can choose our range of addresses, A, to be large, but we waste valuable space. If

Algorithm 3.3

Algorithm for Search, Delete, Insert Using Hashing

```
{To find, delete, or insert a record with key labeled SEARCHKEY; the
operation types (OPTYPE) are FIND, DELETE or INSERT. If a record has to
be inserted, we assume that the address of the block (INSERTADDR) in
which the record will be inserted is specified by the file manager.}
i := h (SEARCHKEY)
{The hash function h will convert nonnumeric keys too. The hash value is
numeric and lies in the bucket address range.}
read bucket with address i into memory.
{Bucket entries are <key, address> pairs. The last key in each bucket is a
dummy and cannot be exceeded. It is used for detecting the last entry.}
get first <key,address> pair
while key ≠ SEARCHKEY and key ≠ DUMMY do
                get next <key,address> pair
found := key = SEARCHKEY {found is Boolean}
case (OPTYPE) of
   'INSERT' : if found then error ('Record Already Exists')
              else
               begin {insert record}
                insert record in data block at INSERTADDR
                insert <SEARCHKEY, INSERTADDR> pair in bucket
                end{insert}
      'FIND' : if not found then error ('Record Does Not Exist')
               else
                begin
                 a := address
                 get data from block a
                 search for record within block
                end{find}
   'DELETE' : if not found then error ('Record Does Not Exist')
              else
               begin
                delete
                delete <SEARCHKEY,address> entry from bucket
                put new block w/i
                delete the appropriate record from block
               end {delete}
end{case}
```

A is too small, the buckets will be large, containing a larger proportion of key values, and the performance will degrade. File reorganization is an expensive proposition. What we want is to be able to modify the hash function as and when required. There are a number of techniques to do this, referred to as **dynamic hashing.** We look at a simple technique called extendable hashing.

3.5.1 Extendable Hashing

Extendable hashing handles file growth and shrinkage by splitting or coalescing buckets, i.e., the number of buckets or the bucket address range changes with the file. Since the hash function, once implemented, can only generate values in some predefined range, the extendable hashing scheme requires that the hash function generates values over a very large range. Instead of using these values as addresses to buckets, some variable number of bits from these values are used as a key for entries in a bucket address table (Figure 3.18). In other words, another level of indirection is introduced. The entries in the bucket address table (BAT) are <*length* (of key), *key*, *bucket address*> triplets.

Let the hash function generate an a bit long value, $b_1b_2 \ldots b_a$. A number of high order bits are used as a pseudosearch key into the bucket address table. The number of bits to be used for each match is determined from the entries in the BAT table. Each *key* in the BAT table is of different length and the length is specified by the corresponding entry in the *length* field. For a given entry in the BAT table, if the value of the length field is $p (p \leq a)$, the p high order bit sequence $b_1b_2 \ldots b_p$ of the hash function generated value becomes the search key and is matched against the key entry in the BAT table. A match gives the *bucket address* where the required search key can be found.

Insertion

When a record is inserted, we follow the same procedure as in the simple hashing scheme. The only difference is when a bucket is full. We refer to it as the *original* bucket (with bucket address given by *original_address*). A new bucket is created; let us call it the *new* bucket (with bucket address given by *new_address*). Let us assume that the key was p bits long. Now since we have two buckets where there was one before, the *length* value has to be increased by one. Thus, the old key $b_1b_2 \ldots b_p$ is replaced by the new keys $b_1b_2 \ldots b_pb_{p+1}$ with the bit b_{p+1} being either 0 or 1. The *key* for the old bucket becomes $b_1b_2 \ldots b_p0$ and for the new bucket $b_1b_2 \ldots b_p1$. We divide the entries from the *original* bucket into the *original* and *new* buckets. In this manner, all keys with their high order bits equal to $b_1b_2 \ldots b_p0$ are placed in the *original* bucket and all keys with their high order bits equal to $b_1b_2 \ldots b_p1$ are placed in the *new* bucket. We modify the BAT entry <p, $b_1b_2 \ldots b_p$, *original_address*> for *original* bucket to become <$p + 1$, $b_1b_2 \ldots b_p0$, *original_*

Figure 3.18 Using extendable hashing.