FILE ORGANISATION

In data-processing applications, the program (or programmer, if you prefer) views a file as a collection of *records.* A record is something like the data part of a C++ object, except that it is stored on some storage medium outside main store, typically on disc, and a whole record is written, or read, in a single operation. A record generally holds information about some real-world entity – the financial transactions of a single customer, the fee payments or grades of a single student, the diagnoses and prescriptions of a single patient, and so on. (In a relational database, a file often corresponds to a relation and a record corresponds to a tuple.)

These notes deal with methods of organising and accessing files of records. They describe the methods in general rather than in terms of any particular programming language. Often the file-handling facilities are provided at the system level and the application programmer can make use of them without knowing exactly how they are implemented. (In the jargon, the implementation is "transparent" to the programmer.) But sometimes the language or the system does not provide them and the programmer has to undertake the implementation.

C++ provides overloading of the input and output operators (>> and <<) so that it is possible to read or write an object with a single statement, and the functions `seekg` and `seekp` provide a simple form of random access. But beyond that, the file processing facilities provided by ISO C++ are very limited and not adequate for most data-processing applications.

For most of this course we will assume that a certain field of the record acts as a primary key, i.e. that each record has a sequence of letters and/or numbers which is unique to that record, like a car's registration number. This field, known loosely just as "the key", plays an important role in several methods of file organisation.

A data-processing application is likely to require some, or all, of the following facilities:

Inserting records into a file (which may or may not be initially empty)

Retrieving all the records from a file, one by one

Retrieving a record with a given key

Deleting a record with a given key

Changing a record (possibly in a way that alters its length)

Retrieving records one by one in some order

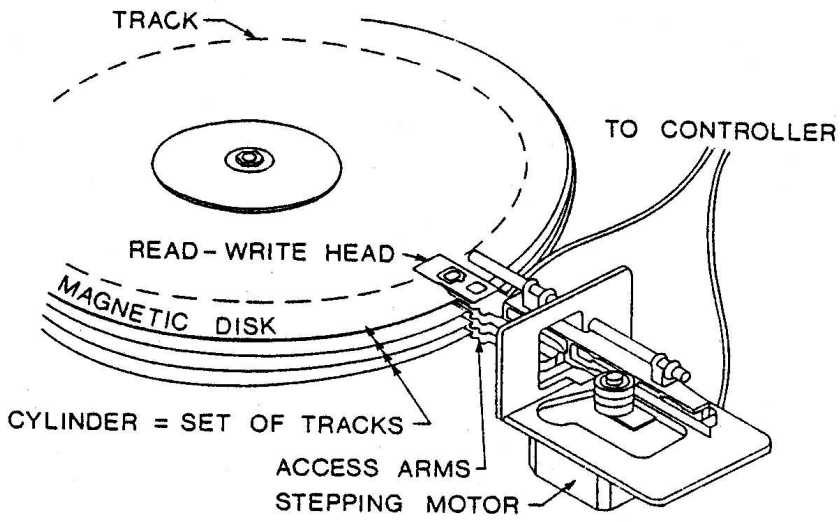
The Unix picture of a file as a sequence of bytes is unhelpful here; it appears to be problematic how you can put a new record into the file "in between" two others, or how you can extend the length of a record and put it back where it was. A better picture is a collection of items in a tub, in no particular order – you can put new ones in, take old ones out, rummage about to find one you want, make some changes to one and put it back in the tub, and so on.

The *physical* unit of data transfer between a file and main store is usually a *block.* These notes, however, adopt the programmer's point of view, whereby the units transferred from file to program are *records.* Often a block will contain several records; sometimes a record may span a block boundary. The packing of records into blocks may be of some importance in practice, but the topic is not covered in these notes.

A characterising feature of a file is whether all the records must be the same size (fixed-length records) or whether they may be of different sizes (variable-length records). In the latter case, the length of each record must be indicated in the file. For example, the record length could be stored with the record (such as a two-byte integer record header), or the end of each record could be marked with a special end-of-record marker, or a directory could be stored in each block with a pointer to the start of each record. The application programmer generally need not be concerned about the method used.

Files may be stored on a variety of media. Punched cards and magnetic tape were used in former times; floppy discs and CDs are used for portability. But the most widely used method is fixed disk ("hard disk" in the PC world). Since the physical features of a disk have some effect on file organisation, a short description of disks now follows.

An introduction to disk storage



A disk drive records data on vertically-stacked circular platters, usually on both the upper and lower surfaces of each platter. Exchangeable disk packs were once widely used - you could take one set of platters off the drive (as a single unit) and put another one on - but fixed disks are now more common, in which the whole unit is sealed in an airtight, dust-free container.

The platters revolve continuously at high speed (over 3000 revs per minute). Each surface has a number of concentric tracks on which data is recorded. The data is read from, or written to, a track by a read/write head positioned over (or under) the track – very close to the surface but not actually touching it. On some systems, each track is divided into a certain number of fixed-length blocks (or sectors); on others, the block length may be varied. On some systems, the heads write a constant amount of data per microsecond, so each track contains the same amount of data; on others, the heads write a constant amount of data per inch, so the outermost track contains more data than the innermost.

There is usually one read/write head for each surface, mounted on an access arm so that the head can be positioned over the track required. The read/write heads all move in and out together so that, at any given time, the tracks being accessed all correspond vertically, and such a set of vertically-corresponding tracks is known as a *cylinder*.

To locate a given point on the disk, the heads have to be positioned at the appropriate cylinder; this involves mechanical movement and is therefore (by computer standards) very slow, taking more than .01 of a second; this is known as the *seek time.* The heads then have to wait until the spinning of the platter brings the required point under the head; obviously this takes, on average, half a rotation and is known as the *rotational delay* or *latency.* The selection of the appropriate read/write head is done electronically and takes negligible time.

If the data is to be read or written sequentially, the obvious sequence is to access in turn each track (surface) of one cylinder, then each track of the next, and so on. Thus disk addressing, at the physical level,

takes the form:

CTB

where C is the cylinder number, T the track (surface) number within the cylinder, and B the block number within the track.

The following table gives information about some disk drives, the first a typical server disk such as we might use on the departmental SUN, then a typical PC hard disk, and finally a floppy:

	Seagate Cheetah ST39103LW	Fujitsu Desktop 18 MPD3084T	Floppy
Surfaces	6	4	2
Tracks per surface (cylinders)	9772	13033	80
Usable GBytes	9.1	8.4	.00144
Avge seek time (ms)	5.2	9.5	-
Avge latency (ms)	2.99	5.56	-
MBytes transmitted per sec	24.5	16.7	0.013

Some abbreviations:

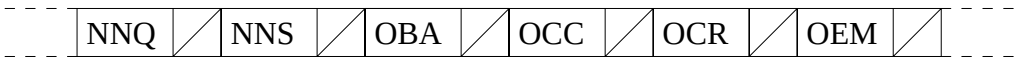
byte	storage for one character, which used to vary from one computer manufacturer to another but which is now universally 8 bits
KByte	kilobyte, 1000 (or 1024) bytes
MByte	megabyte, 1000 KB or a million bytes
GByte	gigabyte, 1000 MB or an (American) billion bytes
TByte	terabyte, 1000 GB or a thousand billion bytes
ms	millisecond (one thousandth of a second) – seek times and latencies are measured in ms
µs	microsecond (one millionth of a second)
ns	nanosecond (one thousandth of a microsecond) – CPU switching speeds are measured in ns
ps	picosecond (one thousandth of a nanosecond)

Sequential file organisation

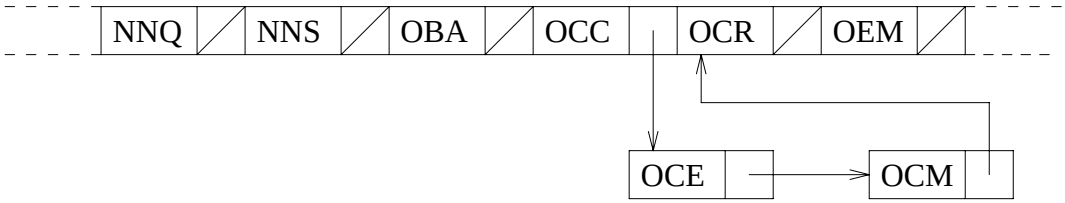
In the simplest sort of sequential file, the only ordering is simply the sequence of records - one comes after another. A sequential file may also be key-ordered, ie there is a unique identifying key (serial number, code number, reference number or whatever) in each record and the records are ordered by this key in the file. A sequential file may be implemented by the physical sequence of records on a disk. As well as being written or read, the records in such a file may be deleted (implemented by a "deleted" marker in the record) or changed, so long as the changes do not affect the record length. If the file is key-ordered, it may be possible to implement a (not very efficient) form of random access by performing a binary search on the file or by making an estimate of the position of the required record and then moving forwards or backwards (like looking up a name in a telephone directory).

A sequential file may also be implemented by a linked list of records on a disk. Each record consists of some data followed by the (CTB) address of the next record. With this method you can do a proper deletion, ie remove the record from the file rather than just marking it as "deleted"; you can write a new record anywhere in the file and you can change a record even if this changes its length. Sequential access, however, is likely to be unacceptably slow, and random access is not possible.

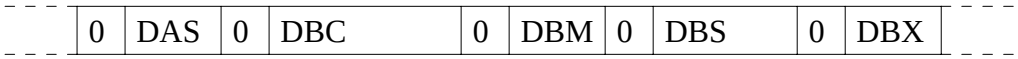
It is possible to have combinations of these two methods, i.e. a file which is written out initially in physical sequence but which allows the possibility of pointers to extra records stored in linked lists. One way to arrange this is for every record in the file to have a link field, all the link fields being initially set to NULL:



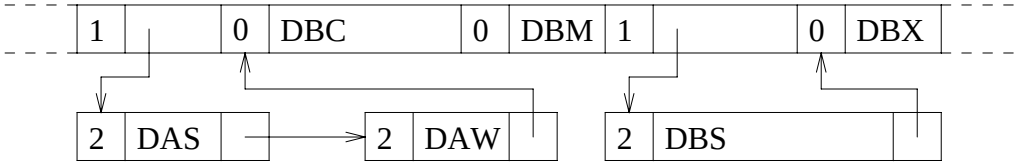
Insertion of records OCM and OCE:



Or each record can be preceded by a flag indicating whether what follows is a record (0) or a pointer (1) or both (2), and a new record is inserted by displacing a record (into a linked list) and replacing it by a pointer:



Insertion of DAW and rewrite of DBS increasing its length:



This method combines reasonably efficient sequential access with the ability to insert new records or make changes, but the sequential access will deteriorate as more such operations are performed, so the file will have to be periodically written out again in full to put all the records back into physical sequence.

Fully-indexed file organisation

In this system, no attempt is made to store the records in any particular order. You can picture them spread higgledy-piggledy around the available space. An *index* is held giving the address of every record in the file. To find a record, you first consult the index. This gives you the address of the record and you then retrieve the record. This system provides random access, easy insertion and deletion, and copes with variable-length records, but key-ordered sequential access is likely to be uncceptably slow.

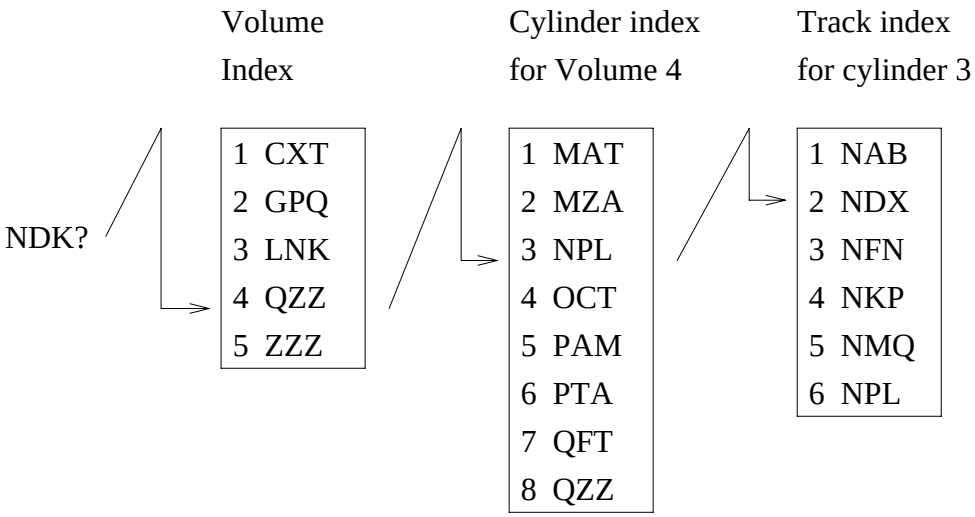
The index may resemble an array of addresses, where an algorithm converts a key to a subscript, eg the algorithm may convert the key value NDK to 5029, in which case the address of record NDK is element 5029 in the index. Alternatively, the index may hold all the keys with their addresses; to find record NDK, you first search the index for "NDK" to retrieve the address that is stored with it, and then you retrieve the record at that address.

The first allows faster retrieval of an address from the index, but may waste a lot of space with empty slots in the index. The second requires some searching of the index; if the index is large, it may be worthwhile to have an index to the index. The second is also less efficient when writing (inserting) a new record since a new entry has to be inserted at the appropriate place in the index, which may mean rearranging other parts of the index (depending on how the index itself is held). The second kind of index is likely to occupy less space, however, even though each entry in it is longer, because it needs to keep an entry only for each record actually in the file, not for each potential key value.

Indexed-sequential file organisation

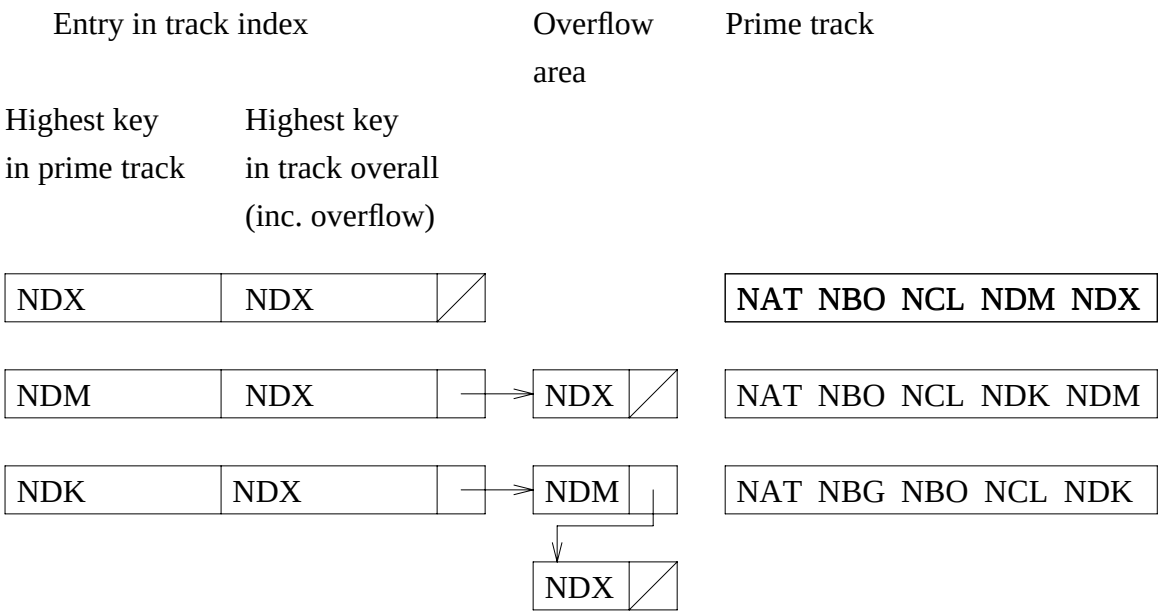
This used to be the normal implementation of a file structure supporting both random and sequential access but nowadays has been largely supplanted by B-trees (covered later in the course). It consists of a key-ordered sequential file (like the first method described above) together with an *index* to aid random access.

For example, on a disk, the first few blocks of each cylinder might be used to hold a track index - a table specifying the highest key on each track of the cylinder. On the first cylinder is stored a cylinder index - the highest key of each cylinder on the disk. If the file occupies more than one disk, a table on the first disk holds a volume index - the highest key on each of the disks. (A single disk in a multi-disk file is often called a "volume" by analogy with the volumes of a multi-volume dictionary or encyclopedia.) The volume index and perhaps the cylinder indexes are read into main store when the file is opened. Random access proceeds by searching each index in turn, starting at the top level. The following example illustrates how the search for record NDK would proceed. As you can see from the earlier table of disk statistics, the example has an unrealistically small number of cylinders.



The above scheme would be adequate for a static file - written once and thereafter only read. But a more complicated arrangement is needed to permit the insertion of new records anywhere in the file or changes that may increase a record's length.

Each cylinder is divided into a *prime* area and an *overflow* area. The file is initially written only in the prime areas. If, thereafter, a new record is to be inserted, its position is located, the new record is written there and the other records in the track are moved along. If there is not enough room in the track for all the records, the one(s) on the end are moved into an overflow area for this track, where they are arranged in a linked list. The following example shows the effect of inserting NDK and then NBG.



The overflow areas – gaps in the file at regular intervals – act rather like holes in a sponge; they enable the file to soak up insertions without much deterioration in performance. But eventually the overflow areas themselves fill up. If a track's overflow area fills up, any further records go into a cylinder overflow area, and if any of those fills up, further records go into a general overflow area. Performance deteriorates as the overflow areas fill up, particularly if a lot of the new records are in the same part of the file and the cylinder overflow areas come into use, so the entire file has to be written out again from time to time.

Directly-addressed file organisation

This is also known as "self-indexing" organisation. The distinctive feature of this method is that a space is reserved on the disk not just for each record that actually exists in the file but for every record that could possibly exist in the file. A simple algorithm converts a key to an address in such a way that each key value corresponds to a unique address on the disk, at which the record identified by that key (if such a record exists) is stored. For example, suppose there are 5 records per block, 4 blocks per track and 20 tracks per cylinder, and suppose we wish to store a record with key 3957. The following algorithm converts each key to a unique disk address in a straightforward way:

3957 div 5 = 791 remainder 2
791 div 4 = 197 remainder 3
197 div 20 = 9 remainder 17

So record 3957 goes in as record 2 (starting from record 0) in block 3 of track 17 of cylinder 9. Note that this algorithm cannot allocate the same address to two records.

In this method, a place is reserved on the disk for every possible key in the file. If every record had a four-digit key which could have any value from 0000 to 9999, you would have to reserve 10,000 places on the disk. This method is therefore practicable only when the number of *active* keys (ie occupied addresses) is a high proportion of *potential* key values (ie total addresses); otherwise, a lot of storage space is wasted. The algorithm will generally be chosen so that records are stored in key order, so key-ordered sequential access is easy. Random access is obviously easy to implement. This method cannot easily handle variable-length records except by making the record length equal to the length of the longest record in the file, though an exceptionally long record may be accommodated by splitting it and storing a pointer in the first part to where its other part is stored.

Random file organisation ("hashed")

The total storage space is divided into areas called buckets, each one large enough to hold a number of records; the buckets are numbered from zero upwards. An algorithm, known as a "hashing" or "key-transformation" algorithm, converts a record-key to a number, and the record is stored in the bucket with that number; for instance, if "NDK" were hashed to 27, record NDK would be stored in bucket 27. The algorithm will convert a number of different key values to the same bucket number. The key values that hash to the same bucket number form an *equivalence class.*

The algorithm needs to be reasonably efficient since it gets executed very often and each execution takes time. It should also ensure that all the records in the file at any given time are fairly evenly distributed across the buckets. A lot of work was done in the 50s and 60s on hashing algorithms, but a review paper – V.Y. Lum et al "Key-to-address transform techniques ..." *CACM (Communications of the Association of Computing Machinery),* April 1971, pp 228-239 – concluded that a good, general method was simply to divide the key by the number of buckets and to take the remainder as the bucket number; for example, key 4735 divided by 53 (= number of buckets) gives 89 remainder 18, so record 4735 goes into bucket 18.

The programmer must be careful not to choose a number of buckets that coincides with some sequence in the record keys; if there were 100 buckets, for example, the algorithm would simply take the last two digits of the key as the bucket number and this could well give an uneven distribution. A prime number is often chosen for this reason.

A bucket number will be mapped to a CTB address, a number of consecutive blocks being allocated as the bucket area. Within a bucket, records may be stored sequentially, perhaps key-ordered or even indexed to speed the searches.

To retrieve a record with a given key, you apply the hashing algorithm to the key to obtain the bucket number, then search that bucket for the record. Only one disk access is required for any record, so random access is good, which is what this system was designed for. If the records are required sequentially *in key order*, however, this method should certainly *not* be used since records that are adjacent in key order will probably be in different buckets.

If each bucket were large enough to hold all the records that might potentially belong to that bucket, hashing would have no advantage over direct addressing, other than the ability to handle variable-length records. The aim is to provide enough space to store all the records that are in the file at a given time without wasting space by being over-generous. Consequently, at any given time in the life of a file, some buckets will be only partly full while others may have overflowed. When a bucket is full and a further record is to be added to it, it may be placed in the next bucket or in some other bucket chosen by algorithm, or it may go into a linked list in an overflow area. If a record is not in the bucket it is supposed to be in, because of overflow, then more than one disk access will be required to find it. The more overflows, therefore, the slower the average retrieval will be.