8

File Processing and External Sorting

Earlier chapters presented basic data structures and algorithms that operate on data stored in main memory. Some applications require that large amounts of information be stored and processed — so much information that it cannot all fit into main memory. In that case, the information must reside on disk and be brought into main memory selectively for processing.

You probably already realize that main memory access is much faster than access to data stored on disk or other storage devices. The relative difference in access times is so great that efficient disk-based programs require a different approach to algorithm design than most programmers are used to. As a result, many programmers do a poor job when it comes to file processing applications.

This chapter presents the fundamental issues relating to the design of algorithms and data structures for disk-based applications.[1] We begin with a description of the significant differences between primary memory and secondary storage. Section 8.2 discusses the physical aspects of disk drives. Section 8.3 presents basic methods for managing buffer pools. Section 8.4 discusses the Java model for random access to data stored on disk. Section 8.5 discusses the basic principles for sorting collections of records too large to fit in main memory.

[1] Computer technology changes rapidly. I provide examples of disk drive specifications and other hardware performance numbers that are reasonably up to date as of the time when the book was written. When you read it, the numbers might seem out of date. However, the basic principles do not change. The approximate ratios for time, space, and cost between memory and disk have remained surprisingly steady for over 20 years.

Medium	1996	1997	2000	2004	2006	2008
RAM	$45.00	7.00	1.500	0.3500	0.1500	0.0339
Disk	0.25	0.10	0.010	0.0010	0.0005	0.0001
Flash drive	–	–	–	0.1000	0.0900	0.0029
Floppy	0.50	0.36	0.250	0.2500	–	–
Tape	0.03	0.01	0.001	0.0003	–	–

Figure 8.1 Price comparison table for some writeable electronic data storage media in common use. Prices are in US Dollars/MB.

8.1 Primary versus Secondary Storage

Computer storage devices are typically classified into **primary** or **main** memory and **secondary** or **peripheral** storage. Primary memory usually refers to **Random Access Memory** (RAM), while secondary storage refers to devices such as hard disk drives, removeable "flash" drives, floppy disks, CDs, DVDs, and tape drives. Primary memory also includes registers, cache, and video memories, but we will ignore them for this discussion because their existence does not affect the principal differences between primary and secondary memory.

Along with a faster CPU, every new model of computer seems to come with more main memory. As memory size continues to increase, is it possible that relatively slow disk storage will be unnecessary? Probably not, because the desire to store and process larger files grows at least as fast as main memory size. Prices for both main memory and peripheral storage devices have dropped dramatically in recent years, as demonstrated by Figure 8.1. However, the cost for disk drive storage per megabyte is about two orders of magnitude less than RAM and has been for many years.

There is now a wide range of removable media available for transfering data or storing data offline in relative safety. These include floppy disks (now largely obsolete), writable CDs and DVDs, "flash" drives, and magnetic tape. Optical storage such as CDs and DVDs costs roughly half the price of hard disk drive space per megabyte, and have become practical for use as backup storage within the past few years. Tape used to be much cheaper than other media, and was the preferred means of backup. "Flash" drives cost the most per megabyte, but due to their storage capacity and flexibility, have now replaced floppy disks as the primary storage device for transferring data between computer when direct network transfer is not available.

Secondary storage devices have at least two other advantages over RAM memory. Perhaps most importantly, disk and tape files are **persistent**, meaning that

they are not erased from disk and tape when the power is turned off. In contrast, RAM used for main memory is usually **volatile** — all information is lost with the power. A second advantage is that floppy disks, CD-ROMs, and "flash" drives can easily be transferred between computers. This provides a convenient way to take information from one computer to another.

In exchange for reduced storage costs, persistence, and portability, secondary storage devices pay a penalty in terms of increased access time. While not all accesses to disk take the same amount of time (more on this later), the typical time required to access a byte of storage from a disk drive in 2007 is around 9 ms (i.e., 9 *thousandths* of a second). This might not seem slow, but compared to the time required to access a byte from main memory, this is fantastically slow. Typical access time from standard personal computer RAM in 2007 is about 5-10 nanoseconds (i.e., 5-10 *billionths* of a second). Thus, the time to access a byte of data from a disk drive is about six orders of magnitude greater than that required to access a byte from main memory. While disk drive and RAM access times are both decreasing, they have done so at roughly the same rate. The relative speeds have remained the same for over twenty-five years, in that the difference in access time between RAM and a disk drive has remained in the range between a factor of 100,000 and 1,000,000.

To gain some intuition for the significance of this speed difference, consider the time that it might take for you to look up the entry for disk drives in the index of this book, and then turn to the appropriate page. Call this your "primary memory" access time. If it takes you about 20 seconds to perform this access, then an access taking 500,000 times longer would require months.

It is interesting to note that while processing speeds have increased dramatically, and hardware prices have dropped dramatically, disk and memory access times have improved by less than an order of magnitude over the past ten years. However, the situation is really much better than that modest speedup would suggest. During the same time period, the size of both disk and main memory has increased by about three orders of magnitude. Thus, the access times have actually decreased in the face of a massive increase in the density of these storage devices.

Due to the relatively slow access time for data on disk as compared to main memory, great care is required to create efficient applications that process disk-based information. The million-to-one ratio of disk access time versus main memory access time makes the following rule of paramount importance when designing disk-based applications:

Minimize the number of disk accesses!

There are generally two approaches to minimizing disk accesses. The first is to arrange information so that if you do access data from secondary memory, you will get what you need in as few accesses as possible, and preferably on the first access. **File structure** is the term used for a data structure that organizes data stored in secondary memory. File structures should be organized so as to minimize the required number of disk accesses. The other way to minimize disk accesses is to arrange information so that each disk access retrieves additional data that can be used to minimize the need for future accesses, that is, to guess accurately what information will be needed later and retrieve it from disk now, if this can be done cheaply. As you shall see, there is little or no difference in the time required to read several hundred contiguous bytes from disk as compared to reading one byte, so this strategy is indeed practical.

One way to minimize disk accesses is to compress the information stored on disk. Section 3.9 discusses the space/time tradeoff in which space requirements can be reduced if you are willing to sacrifice time. However, the disk-based space/time tradeoff principle stated that the smaller you can make your disk storage requirements, the faster your program will run. This is because the time to read information from disk is enormous compared to computation time, so almost any amount of additional computation to unpack the data is going to be less than the disk read time saved by reducing the storage requirements. This is precisely what happens when files are compressed. CPU time is required to uncompress information, but this time is likely to be much less than the time saved by reducing the number of bytes read from disk. Current file compression programs are not designed to allow random access to parts of a compressed file, so the disk-based space/time tradeoff principle cannot easily be taken advantage of in normal processing using commercial disk compression utilities. However, in the future disk drive controllers might automatically compress and decompress files stored on disk, thus taking advantage of the disk-based space/time tradeoff principle to save both space and time. Many cartridge tape drives (which must process data sequentially) automatically compress and decompress information during I/O.

8.2 Disk Drives

A Java programmer views a random access file stored on disk as a contiguous series of bytes, with those bytes possibly combining to form data records. This is called the **logical** file. The **physical** file actually stored on disk is usually not a contiguous series of bytes. It could well be in pieces spread all over the disk. The **file manager**, a part of the operating system, is responsible for taking requests for data from a logical file and mapping those requests to the physical location

of the data on disk. Likewise, when writing to a particular logical byte position with respect to the beginning of the file, this position must be converted by the file manager into the corresponding physical location on the disk. To gain some appreciation for the the approximate time costs for these operations, you need to understand the physical structure and basic workings of a disk drive.

Disk drives are often referred to as **direct access** storage devices. This means that it takes roughly equal time to access any record in the file. This is in contrast to **sequential access** storage devices such as tape drives, which require the tape reader to process data from the beginning of the tape until the desired position has been reached. As you will see, the disk drive is only approximately direct access: At any given time, some records are more quickly accessible than others.

8.2.1 Disk Drive Architecture

A hard disk drive is composed of one or more round **platters**, stacked one on top of another and attached to a central **spindle**. Platters spin continuously at a constant rate. Each usable surface of each platter is assigned a **read/write head** or **I/O head** through which data are read or written, somewhat like the arrangement of a phonograph player's arm "reading" sound from a phonograph record. Unlike a phonograph needle, the disk read/write head does not actually touch the surface of a hard disk. Instead, it remains slightly above the surface, and any contact during normal operation would damage the disk. This distance is very small, much smaller than the height of a dust particle. It can be likened to a 5000-kilometer airplane trip across the United States, with the plane flying at a height of one meter!

A hard disk drive typically has several platters and several read/write heads, as shown in Figure 8.2(a). Each head is attached to an **arm**, which connects to the **boom**. The boom moves all of the heads in or out together. When the heads are in some position over the platters, there are data on each platter directly accessible to each head. The data on a single platter that are accessible to any one position of the head for that platter are collectively called a **track**, that is, all data on a platter that are a fixed distance from the spindle, as shown in Figure 8.2(b). The collection of all tracks that are a fixed distance from the spindle is called a **cylinder**. Thus, a cylinder is all of the data that can be read when the arms are in a particular position.

Each track is subdivided into **sectors**. Between each sector there are **intersector gaps** in which no data are stored. These gaps allow the read head to recognize the end of a sector. Note that each sector contains the same amount of data. Because the outer tracks have greater length, they contain fewer bits per inch than do the inner tracks. Thus, about half of the potential storage space is wasted, because only the innermost tracks are stored at the highest possible data density. This ar-

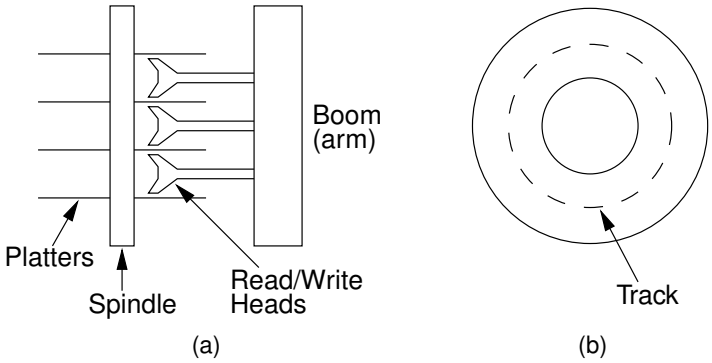

Figure 8.2 (a) A typical disk drive arranged as a stack of platters. (b) One track on a disk drive platter.

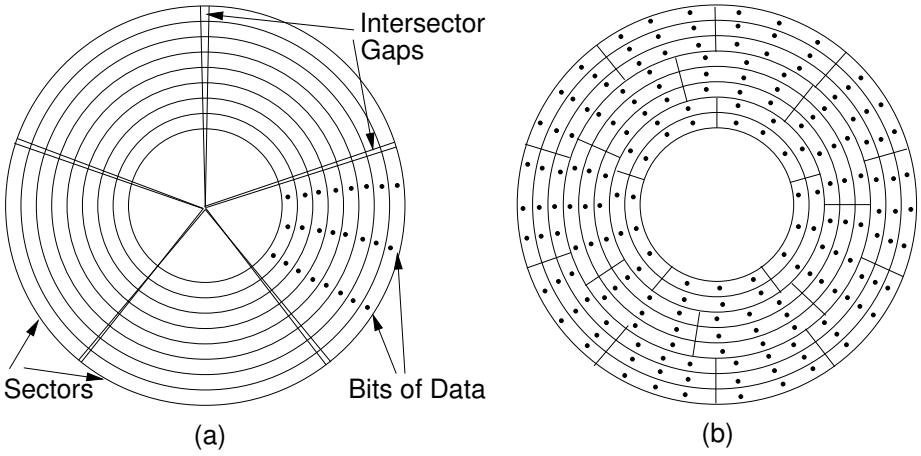

Figure 8.3 The organization of a disk platter. Dots indicate density of information. (a) Nominal arrangement of tracks showing decreasing data density when moving outward from the center of the disk. (b) A "zoned" arrangement with the sector size and density periodically reset in tracks further away from the center.

rangement is illustrated by Figure 8.3a. Disk drives today actually group tracks into "zones" such that the tracks in the innermost zone adjust their data density going out to maintain the same radial data density, then the tracks of the next zone reset the data density to make better use of their storage ability, and so on. This arrangement is shown in Figure 8.3b.

In contrast to the physical layout of a hard disk, a CD-ROM consists of a single spiral track. Bits of information along the track are equally spaced, so the information density is the same at both the outer and inner portions of the track. To keep

the information flow at a constant rate along the spiral, the drive must speed up the rate of disk spin as the I/O head moves toward the center of the disk. This makes for a more complicated and slower mechanism.

Three separate steps take place when reading a particular byte or series of bytes of data from a hard disk. First, the I/O head moves so that it is positioned over the track containing the data. This movement is called a **seek**. Second, the sector containing the data rotates to come under the head. When in use the disk is always spinning. At the time of this writing, typical disk spin rates are 7200 rotations per minute (rpm). The time spent waiting for the desired sector to come under the I/O head is called **rotational delay** or **rotational latency**. The third step is the actual transfer (i.e., reading or writing) of data. It takes relatively little time to read information once the first byte is positioned under the I/O head, simply the amount of time required for it all to move under the head. In fact, disk drives are designed not to read one byte of data, but rather to read an entire sector of data at each request. Thus, a sector is the minimum amount of data that can be read or written at one time.

In general, it is desirable to keep all sectors for a file together on as few tracks as possible. This desire stems from two assumptions:

1. Seek time is slow (it is typically the most expensive part of an I/O operation), and
2. If one sector of the file is read, the next sector will probably soon be read.

Assumption (2) is called **locality of reference**, a concept that comes up frequently in computer applications.

Contiguous sectors are often grouped to form a **cluster**. A cluster is the smallest unit of allocation for a file, so all files are a multiple of the cluster size. The cluster size is determined by the operating system. The file manager keeps track of which clusters make up each file.

In Microsoft Windows systems, there is a designated portion of the disk called the **File Allocation Table**, which stores information about which sectors belong to which file. In contrast, UNIX does not use clusters. The smallest unit of file allocation and the smallest unit that can be read/written is a sector, which in UNIX terminology is called a **block**. UNIX maintains information about file organization in certain disk blocks called **i-nodes**.

A group of physically contiguous clusters from the same file is called an **extent**. Ideally, all clusters making up a file will be contiguous on the disk (i.e., the file will consist of one extent), so as to minimize seek time required to access different portions of the file. If the disk is nearly full when a file is created, there might not be an extent available that is large enough to hold the new file. Furthermore, if a file

grows, there might not be free space physically adjacent. Thus, a file might consist of several extents widely spaced on the disk. The fuller the disk, and the more that files on the disk change, the worse this file fragmentation (and the resulting seek time) becomes. File fragmentation leads to a noticeable degradation in performance as additional seeks are required to access data.

Another type of problem arises when the file's logical record size does not match the sector size. If the sector size is not a multiple of the record size (or vice versa), records will not fit evenly within a sector. For example, a sector might be 2048 bytes long, and a logical record 100 bytes. This leaves room to store 20 records with 48 bytes left over. Either the extra space is wasted, or else records are allowed to cross sector boundaries. If a record crosses a sector boundary, two disk accesses might be required to read it. If the space is left empty instead, such wasted space is called **internal fragmentation**.

A second example of internal fragmentation occurs at cluster boundaries. Files whose size is not an even multiple of the cluster size must waste some space at the end of the last cluster. The worst case will occur when file size modulo cluster size is one (for example, a file of 4097 bytes and a cluster of 4096 bytes). Thus, cluster size is a tradeoff between large files processed sequentially (where a large cluster size is desirable to minimize seeks) and small files (where small clusters are desirable to minimize wasted storage).

Every disk drive organization requires that some disk space be used to organize the sectors, clusters, and so forth. The layout of sectors within a track is illustrated by Figure 8.4. Typical information that must be stored on the disk itself includes the File Allocation Table, **sector headers** that contain address marks and information about the condition (whether usable or not) for each sector, and gaps between sectors. The sector header also contains error detection codes to help verify that the data have not been corrupted. This is why most disk drives have a "nominal" size that is greater than the actual amount of user data that can be stored on the drive. The difference is the amount of space required to organize the information on the disk. Additional space will be lost due to fragmentation.

8.2.2 Disk Access Costs

The primary cost when accessing information on disk is normally the seek time. This assumes of course that a seek is necessary. When reading a file in sequential order (if the sectors comprising the file are contiguous on disk), little seeking is necessary. However, when accessing a random disk sector, seek time becomes the dominant cost for the data access. While the actual seek time is highly variable, depending on the distance between the track where the I/O head currently is and

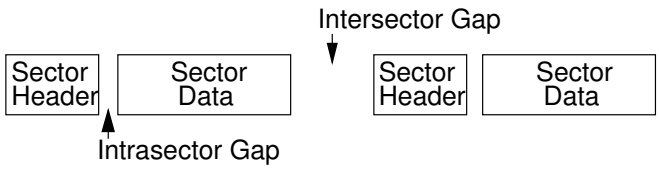

Figure 8.4 An illustration of sector gaps within a track. Each sector begins with a sector header containing the sector address and an error detection code for the contents of that sector. The sector header is followed by a small intrasector gap, followed in turn by the sector data. Each sector is separated from the next sector by a larger intersector gap.

the track where the head is moving to, we will consider only two numbers. One is the track-to-track cost, or the minimum time necessary to move from a track to an adjacent track. This is appropriate when you want to analyze access times for files that are well placed on the disk. The second number is the average seek time for a random access. These two numbers are often provided by disk manufacturers. A typical example is the 120GB Western Digital WD Caviar SE serial ATA drive. The manufacturer's specifications indicate that the track-to-track time is 2.0 ms and the average seek time is 9.0 ms.

For many years, typical rotation speed for disk drives was 3600 rpm, or one rotation every 16.7 ms. Most disk drives today have a rotation speed of 7200 rpm, or 8.3 ms per rotation. When reading a sector at random, you can expect that the disk will need to rotate halfway around to bring the desired sector under the I/O head, or 4.2 ms for a 7200-rpm disk drive.

Once under the I/O head, a sector of data can be transferred as fast as that sector rotates under the head. If an entire track is to be read, then it will require one rotation (8.3 ms at 7200 rpm) to move the full track under the head. If only part of the track is to be read, then proportionately less time will be required. For example, if there are 16K sectors on the track and one sector is to be read, this will require a trivial amount of time (1/16K of a rotation).

Example 8.1 Assume that an older disk drive has a total (nominal) capacity of 16.8GB spread among 10 platters, yielding 1.68GB/platter. Each platter contains 13,085 tracks and each track contains (after formating) 256 sectors of 512 bytes/sector. Track-to-track seek time is 2.2 ms and average seek time for random access is 9.5 ms. Assume the operating system maintains a cluster size of 8 sectors per cluster (4KB), yielding 32 clusters per track. The disk rotation rate is 5400 rpm (11.1 ms per rotation). Based on this information we can estimate the cost for various file processing operations.

How much time is required to read the track? On average, it will require half a rotation to bring the first sector of the track under the I/O head, and then one complete rotation to read the track.

How long will it take to read a file of 1MB divided into 2048 sector-sized (512 byte) records? This file will be stored in 256 clusters, because each cluster holds 8 sectors. The answer to the question depends in large measure on how the file is stored on the disk, that is, whether it is all together or broken into multiple extents. We will calculate both cases to see how much difference this makes.

If the file is stored so as to fill all of the sectors of eight adjacent tracks, then the cost to read the first sector will be the time to seek to the first track (assuming this requires a random seek), then a wait for the initial rotational delay, and then the time to read. This requires

$$9.5 + 11.1 \times 1.5 = 26.2 \text{ ms}.$$

At this point, because we assume that the next seven tracks require only a track-to-track seek because they are adjacent, each requires

$$2.2 + 11.1 \times 1.5 = 18.9 \text{ ms}.$$

The total time required is therefore

$$26.2\text{ms} + 7 \times 18.9\text{ms} = 158.5\text{ms}.$$

If the file's clusters are spread randomly across the disk, then we must perform a seek for each cluster, followed by the time for rotational delay. Once the first sector of the cluster comes under the I/O head, very little time is needed to read the cluster because only 8/256 of the track needs to rotate under the head, for a total time of about 5.9 ms for latency and read time. Thus, the total time required is about

$$256(9.5 + 5.9) \approx 3942\text{ms}$$

or close to 4 seconds. This is much longer than the time required when the file is all together on disk!

This example illustrates why it is important to keep disk files from becoming fragmented, and why so-called "disk defragmenters" can speed up file processing time. File fragmentation happens most commonly when the disk is nearly full and the file manager must search for free space whenever a file is created or changed.

8.3 Buffers and Buffer Pools

Given the specifications of the disk drive from Example 8.1, we find that it takes about $9.5+11.1\times1.5 = 26.2$ ms to read one track of data on average. It takes about $9.5+11.1/2+(1/256)\times11.1 = 15.1$ ms on average to read a single sector of data. This is a good savings (slightly over half the time), but less than 1% of the data on the track are read. If we want to read only a single byte, it would save us effectively no time over that required to read an entire sector. For this reason, nearly all disk drives automatically read or write an entire sector's worth of information whenever the disk is accessed, even when only one byte of information is requested.

Once a sector is read, its information is stored in main memory. This is known as **buffering** or **caching** the information. If the next disk request is to that same sector, then it is not necessary to read from disk again because the information is already stored in main memory. Buffering is an example of one method for minimizing disk accesses mentioned at the beginning of the chapter: Bring off additional information from disk to satisfy future requests. If information from files were accessed at random, then the chance that two consecutive disk requests are to the same sector would be low. However, in practice most disk requests are close to the location (in the logical file at least) of the previous request. This means that the probability of the next request "hitting the cache" is much higher than chance would indicate.

This principle explains one reason why average access times for new disk drives are lower than in the past. Not only is the hardware faster, but information is also now stored using better algorithms and larger caches that minimize the number of times information needs to be fetched from disk. This same concept is also used to store parts of programs in faster memory within the CPU, using the CPU cache that is prevalent in modern microprocessors.

Sector-level buffering is normally provided by the operating system and is often built directly into the disk drive controller hardware. Most operating systems maintain at least two buffers, one for input and one for output. Consider what would happen if there were only one buffer during a byte-by-byte copy operation. The sector containing the first byte would be read into the I/O buffer. The output operation would need to destroy the contents of the single I/O buffer to write this byte. Then the buffer would need to be filled again from disk for the second byte, only to be destroyed during output. The simple solution to this problem is to keep one buffer for input, and a second for output.

Most disk drive controllers operate independently from the CPU once an I/O request is received. This is useful because the CPU can typically execute millions of instructions during the time required for a single I/O operation. A technique that

takes maximum advantage of this microparallelism is **double buffering**. Imagine that a file is being processed sequentially. While the first sector is being read, the CPU cannot process that information and so must wait or find something else to do in the meantime. Once the first sector is read, the CPU can start processing while the disk drive (in parallel) begins reading the second sector. If the time required for the CPU to process a sector is approximately the same as the time required by the disk controller to read a sector, it might be possible to keep the CPU continuously fed with data from the file. The same concept can also be applied to output, writing one sector to disk while the CPU is writing to a second output buffer in memory. Thus, in computers that support double buffering, it pays to have at least two input buffers and two output buffers available.

Caching information in memory is such a good idea that it is usually extended to multiple buffers. The operating system or an application program might store many buffers of information taken from some **backing storage** such as a disk file. This process of using buffers as an intermediary between a user and a disk file is called **buffering** the file. The information stored in a buffer is often called a **page**, and the collection of buffers is called a **buffer pool**. The goal of the buffer pool is to increase the amount of information stored in memory in hopes of increasing the likelihood that new information requests can be satisfied from the buffer pool rather than requiring new information to be read from disk.

As long as there is an unused buffer available in the buffer pool, new information can be read in from disk on demand. When an application continues to read new information from disk, eventually all of the buffers in the buffer pool will become full. Once this happens, some decision must be made about what information in the buffer pool will be sacrificed to make room for newly requested information.

When replacing information contained in the buffer pool, the goal is to select a buffer that has "unnecessary" information, that is, that information least likely to be requested again. Because the buffer pool cannot know for certain what the pattern of future requests will look like, a decision based on some **heuristic**, or best guess, must be used. There are several approaches to making this decision.

One heuristic is "first-in, first-out" (FIFO). This scheme simply orders the buffers in a queue. The buffer at the front of the queue is used next to store new information and then placed at the end of the queue. In this way, the buffer to be replaced is the one that has held its information the longest, in hopes that this information is no longer needed. This is a reasonable assumption when processing moves along the file at some steady pace in roughly sequential order. However, many programs work with certain key pieces of information over and over again, and the importance of information has little to do with how long ago the informa-

tion was first accessed. Typically it is more important to know how many times the information has been accessed, or how recently the information was last accessed.

Another approach is called "least frequently used" (LFU). LFU tracks the number of accesses to each buffer in the buffer pool. When a buffer must be reused, the buffer that has been accessed the fewest number of times is considered to contain the "least important" information, and so it is used next. LFU, while it seems intuitively reasonable, has many drawbacks. First, it is necessary to store and update access counts for each buffer. Second, what was referenced many times in the past might now be irrelevant. Thus, some time mechanism where counts "expire" is often desirable. This also avoids the problem of buffers that slowly build up big counts because they get used just often enough to avoid being replaced. An alternative is to maintain counts for all sectors ever read, not just the sectors currently in the buffer pool.

The third approach is called "least recently used" (LRU). LRU simply keeps the buffers in a list. Whenever information in a buffer is accessed, this buffer is brought to the front of the list. When new information must be read, the buffer at the back of the list (the one least recently used) is taken and its "old" information is either discarded or written to disk, as appropriate. This is an easily implemented approximation to LFU and is often the method of choice for managing buffer pools unless special knowledge about information access patterns for an application suggests a special-purpose buffer management scheme.

The main purpose of a buffer pool is to minimize disk I/O. When the contents of a block are modified, we could write the updated information to disk immediately. But what if the block is changed again? If we write the block's contents after every change, that might be a lot of disk write operations that can be avoided. It is more efficient to wait until either the file is to be closed, or the buffer containing that block is flushed from the buffer pool.

When a buffer's contents are to be replaced in the buffer pool, we only want to write the contents to disk if it is necessary. That would be necessary only if the contents have changed since the block was read in originally from the file. The way to insure that the block is written when necessary, but only when necessary, is to maintain a boolean variable with the buffer (often referred to as the **dirty bit**) that is turned on when the buffer's contents are modified by the client. At the time when the block is flushed from the buffer pool, it is written to disk if and only if the dirty bit has been turned on.

Modern operating systems support **virtual memory**. Virtual memory is a technique that allows the programmer to pretend that there is more of the faster main memory (such as RAM) than actually exists. This is done by means of a buffer pool

reading blocks stored on slower, secondary memory (such as on the disk drive). The disk stores the complete contents of the virtual memory. Blocks are read into main memory as demanded by memory accesses. Naturally, programs using virtual memory techniques are slower than programs whose data are stored completely in main memory. The advantage is reduced programmer effort because a good virtual memory system provides the appearance of larger main memory without modifying the program.

Example 8.2 Consider a virtual memory whose size is ten sectors, and which has a buffer pool of five buffers associated with it. We will use a LRU replacement scheme. The following series of memory requests occurs.

$$9017668135171$$

After the first five requests, the buffer pool will store the sectors in the order 6, 7, 1, 0, 9. Because Sector 6 is already at the front, the next request can be answered without reading new data from disk or reordering the buffers. The request to Sector 8 requires emptying the contents of the least recently used buffer, which contains Sector 9. The request to Sector 1 brings the buffer holding Sector 1's contents back to the front. Processing the remaining requests results in the buffer pool as shown in Figure 8.5.

Example 8.3 Figure 8.5 illustrates a buffer pool of five blocks mediating a virtual memory of ten blocks. At any given moment, up to five sectors of information can be in main memory. Assume that Sectors 1, 7, 5, 3, and 8 are curently in the buffer pool, stored in this order, and that we use the LRU buffer replacement strategy. If a request for Sector 9 is then received, then one sector currently in the buffer pool must be replaced. Because the buffer containing Sector 8 is the least recently used buffer, its contents will be copied back to disk at Sector 8. The contents of Sector 9 are then copied into this buffer, and it is moved to the front of the buffer pool (leaving the buffer containing Sector 3 as the new least-recently used buffer). If the next memory request were to Sector 5, no data would need to be read from disk. Instead, the buffer containing Sector 5 would be moved to the front of the buffer pool.

When implementing buffer pools, there are two basic approaches that can be taken regarding the transfer of information between the user of the buffer pool and

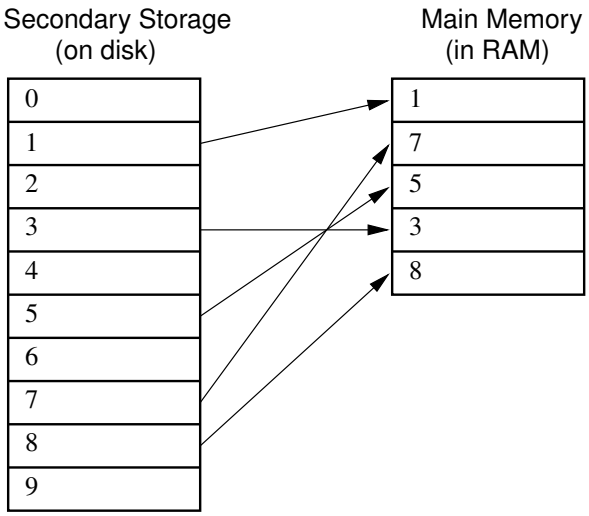

Figure 8.5 An illustration of virtual memory. The complete collection of information resides in the slowe, secondary storage (on disk). Those sectors recently accessed are held in the fast main memory (in RAM). In this example, copies of Sectors 1, 7, 5, 3, and 8 from secondary storage are currently stored in the main memory. If a memory access to Sector 9 is received, one of the sectors currently in main memory must be replaced.

the buffer pool class itself. The first approach is to pass "messages" between the two. This approach is illustrated by the following abstract class:

```
/** ADT for buffer pools using the message-passing style */
public interface BufferPoolADT {
  /** Copy "sz" bytes from "space" to position "pos" in the
      buffered storage */
  public void insert(byte[] space, int sz, int pos);

  /** Copy "sz" bytes from position "pos" of the buffered
      storage to "space". */
  public void getbytes(byte[] space, int sz, int pos);
}
```

This simple class provides an interface with two member functions, **`insert`** and **`getbytes`**. The information is passed between the buffer pool user and the buffer pool through the **`space`** parameter. This is storage space, provided by the bufferpool client and at least **`sz`** bytes long, which the buffer pool can take information from (the **`insert`** function) or put information into (the **`getbytes`** function). Parameter **`pos`** indicates where the information will be placed in the

buffer pool's logical storage space. Physically, it will actually be copied to the appropriate byte position in some buffer in the buffer pool.

Example 8.4 Assume each sector of the disk file (and thus each block in the buffer pool) stores 1024 bytes. Assume that the buffer pool is in the state shown in Figure 8.5. If the next request is to copy 40 bytes beginning at position 6000 of the file, these bytes should be placed into Sector 5 (whose bytes go from position 5120 to position 6143). Because Sector 5 is currently in the buffer pool, we simply copy the 40 bytes contained in **space** to byte positions 880-919. The buffer containing Sector 5 is then moved to the buffer pool ahead of the buffer containing Sector 1.

The alternative approach is to have the buffer pool provide to the user a direct pointer to a buffer that contains the necessary information. Such an interface might look as follows:

```
/** ADT for buffer pools using the buffer-passing style */
public interface BufferPoolADT {
  /** Return pointer to the requested block */
  public byte[] getblock(int block);

  /** Set the dirty bit for the buffer holding "block" */
  public void dirtyblock(int block);

  // Tell the size of a buffer
  public int blocksize();
};
```

In this approach, the user of the buffer pool is made aware that the storage space is divided into blocks of a given size, where each block is the size of a buffer. The user requests specific blocks from the buffer pool, with a pointer to the buffer holding the requested block being returned to the user. The user might then read from or write to this space. If the user writes to the space, the buffer pool must be informed of this fact. The reason is that, when a given block is to be removed from the buffer pool, the contents of that block must be written to the backing storage if it has been modified. If the block has not been modified, then it is unnecessary to write it out.

Example 8.5 We wish to write 40 bytes beginning at logical position 6000 in the file. Assume that the buffer pool is in the state shown in Figure 8.5. Using the second ADT, the client would need to know that blocks (buffers) are of size 1024, and therefore would request access to Sector 5.

A pointer to the buffer containing Sector 5 would be retuned by the call to **`getblock`**. The client would then copy 40 bytes to positions 880-919 of the buffer, and call **`dirtyblock`** to warn the buffer pool that the contents of this block have been modified.

A further problem with the second ADT is the risk of **stale pointers**. When the buffer pool user is given a pointer to some buffer space at time **T1**, that pointer does indeed refer to the desired data at that time. As further requests are made to the buffer pool, it is possible that the data in any given buffer will be removed and replaced with new data. If the buffer pool user at a later time **T2** then refers to the data refered to by the pointer given at time **T1**, it is possible that the data are no longer valid because the buffer contents have been replaced in the meantime. Thus the pointer into the buffer pool's memory has become "stale." To guarantee that a pointer is not stale, it should not be used if intervening requests to the buffer pool have taken place.

We can solve this problem by introducing the concept of a user (or possibly multiple users) gaining access to a buffer, and then releasing the buffer when done. We will add method **`acquireBuffer`** and **`releaseBuffer`** for this purpose. Method **`acquireBuffer`** takes a block ID as input and returns a pointer to the buffer that will be used to store this block. The buffer pool will keep a count of the number of requests currently active for this block. Method **`releaseBuffer`** will reduce the count of active users for the associated block. Buffers associated with active blocks will not be eligible for flushing from the buffer pool. This will lead to a problem if the client neglects to release active blocks when they are no longer needed. There would also be a problem if there were more total active blocks than buffers in the buffer pool. However, the buffer pool should always be initialized to include more buffers than will ever be active at one time.

An additional problem with both ADTs presented so far comes when the user intends to completely overwrite the contents of a block, and does not need to read in the old contents already on disk. However, the buffer pool cannot in general know whether the user wishes to use the old contents or not. This is especially true with the message-passing approach where a given message might overwrite only part of the block. In this case, the block will be read into memory even when not needed, and then its contents will be overwritten.

This inefficiency can be avoided (at least in the buffer-passing version) by separating the assignment of blocks to buffers from actually reading in data for the block. In particular, the following revised buffer-passing ADT does not actually read data in the **`acquireBuffer`** method. Users who wish to see the old con-

tents must then issue a **readBlock** request to read the data from disk into the buffer, and then a **getDataPointer** request to gain direct access to the buffer's data contents.

```
/** Improved ADT for buffer pools using the buffer-passing
    style. Most user functionality is in the buffer class,
    not the buffer pool itself. */

/** A single buffer in the buffer pool */
public interface BufferADT {
  /** Read the associated block from disk (if necessary) and
      return a pointer to the data */
  public byte[] readBlock();

  /** Return a pointer to the buffer's data array
      (without reading from disk) */
  public byte[] getDataPointer();

  /** Flag the buffer's contents as having changed, so that
      flushing the block will write it back to disk */
  public void markDirty();

  /** Release the block's access to this buffer. Further
      accesses to this buffer are illegal. */
  public void releaseBuffer();
}

/** The bufferpool */
public interface BufferPoolADT {

  /** Relate a block to a buffer, returning a pointer to a
      buffer object */
  Buffer acquireBuffer(int block);
}
```

Clearly, the buffer-passing approach places more obligations on the user of the buffer pool. These obligations include knowing the size of a block, not corrupting the buffer pool's storage space, and informing the buffer pool both when a block has been modified and when it is no longer needed. So many obligations make this approach prone to error. An advantage is that there is no need to do an extra copy step when getting information from the user to the buffer. If the size of the records stored is small, this is not an important consideration. If the size of the records is large (especially if the record size and the buffer size are the same, as typically is the case when implementing B-trees, see Section 10.5), then this efficiency issue might become important. Note however that the in-memory copy time will always be far less than the time required to write the contents of a buffer to disk. For applications

where disk I/O is the bottleneck for the program, even the time to copy lots of information between the buffer pool user and the buffer might be inconsequential. Another advantage to buffer passing is the reduction in unnecessary read operations for data that will be overwritten anyway.

You should note that the implementations for class **`BufferPool`** above are not generic. Instead, the **`space`** parameter and the buffer pointer are declared to be **`byte[]`**. When a class is generic, that means that the record type is arbitrary, but that the class knows what the record type is. In contrast, using a **`byte[]`** pointer for the space means that not only is the record type arbitrary, but also the buffer pool does not even know what the user's record type is. In fact, a given buffer pool might have many users who store many types of records.

In a buffer pool, the user decides where a given record will be stored but has no control over the precise mechanism by which data are transfered to the backing storage. This is in contrast to the memory manager described in Section 12.3 in which the user passes a record to the manager and has no control at all over where the record is stored.

8.4 The Programmer's View of Files

The Java programmer's logical view of a random access file is a single stream of bytes. Interaction with a file can be viewed as a communications channel for issuing one of three instructions: read bytes from the current position in the file, write bytes to the current position in the file, and move the current position within the file. You do not normally see how the bytes are stored in sectors, clusters, and so forth. The mapping from logical to physical addresses is done by the file system, and sector-level buffering is done automatically by the disk controller.

When processing records in a disk file, the order of access can have a great effect on I/O time. A **random access** procedure processes records in an order independent of their logical order within the file. **Sequential access** processes records in order of their logical appearance within the file. Sequential processing requires less seek time if the physical layout of the disk file matches its logical layout, as would be expected if the file were created on a disk with a high percentage of free space.

Following are the primary Java functions for accessing information from random access disk files when using class **`RandomAccessFile`**.

- **`RandomAccessFile(String name, String mode)`**: Class constructor, opens a disk file for processing.

- **`read(byte[] b)`**: Read some bytes from the current position in the file. The current position moves forward as the bytes are read.
- **`write(byte[] b)`**: Write some bytes at the current position in the file (overwriting the bytes already at that position). The current position moves forward as the bytes are written.
- **`seek(long pos)`**: Move the current position in the file. This allows bytes at other places within the file to be read or written.
- **`close()`**: Close a file at the end of processing.

8.5 External Sorting

We now consider the problem of sorting collections of records too large to fit in main memory. Because the records must reside in peripheral or external memory, such sorting methods are called **external sorts**. This is in contrast to the internal sorts discussed in Chapter 7 which assume that the records to be sorted are stored in main memory. Sorting large collections of records is central to many applications, such as processing payrolls and other large business databases. As a consequence, many external sorting algorithms have been devised. Years ago, sorting algorithm designers sought to optimize the use of specific hardware configurations, such as multiple tape or disk drives. Most computing today is done on personal computers and low-end workstations with relatively powerful CPUs, but only one or at most two disk drives. The techniques presented here are geared toward optimized processing on a single disk drive. This approach allows us to cover the most important issues in external sorting while skipping many less important machine-dependent details. Readers who have a need to implement efficient external sorting algorithms that take advantage of more sophisticated hardware configurations should consult the references in Section 8.6.

When a collection of records is too large to fit in main memory, the only practical way to sort it is to read some records from disk, do some rearranging, then write them back to disk. This process is repeated until the file is sorted, with each record read perhaps many times. Given the high cost of disk I/O, it should come as no surprise that the primary goal of an external sorting algorithm is to minimize the amount of information that must be read from or written to disk. A certain amount of additional CPU processing can profitably be traded for reduced disk access.

Before discussing external sorting techniques, consider again the basic model for accessing information from disk. The file to be sorted is viewed by the programmer as a sequential series of fixed-size **blocks**. Assume (for simplicity) that each

block contains the same number of fixed-size data records. Depending on the application, a record might be only a few bytes — composed of little or nothing more than the key — or might be hundreds of bytes with a relatively small key field. Records are assumed not to cross block boundaries. These assumptions can be relaxed for special-purpose sorting applications, but ignoring such complications makes the principles clearer.

As explained in Section 8.2, a sector is the basic unit of I/O. In other words, all disk reads and writes are for one or more complete sectors. Sector sizes are typically a power of two, in the range 512 to 16K bytes, depending on the operating system and the size and speed of the disk drive. The block size used for external sorting algorithms should be equal to or a multiple of the sector size.

Under this model, a sorting algorithm reads a block of data into a buffer in main memory, performs some processing on it, and at some future time writes it back to disk. From Section 8.1 we see that reading or writing a block from disk takes on the order of one million times longer than a memory access. Based on this fact, we can reasonably expect that the records contained in a single block can be sorted by an internal sorting algorithm such as Quicksort in less time than is required to read or write the block.

Under good conditions, reading from a file in sequential order is more efficient than reading blocks in random order. Given the significant impact of seek time on disk access, it might seem obvious that sequential processing is faster. However, it is important to understand precisely under what circumstances sequential file processing is actually faster than random access, because it affects our approach to designing an external sorting algorithm.

Efficient sequential access relies on seek time being kept to a minimum. The first requirement is that the blocks making up a file are in fact stored on disk in sequential order and close together, preferably filling a small number of contiguous tracks. At the very least, the number of extents making up the file should be small. Users typically do not have much control over the layout of their file on disk, but writing a file all at once in sequential order to a disk drive with a high percentage of free space increases the likelihood of such an arrangement.

The second requirement is that the disk drive's I/O head remain positioned over the file throughout sequential processing. This will not happen if there is competition of any kind for the I/O head. For example, on a multi-user timeshared computer the sorting process might compete for the I/O head with the processes of other users. Even when the sorting process has sole control of the I/O head, it is still likely that sequential processing will not be efficient. Imagine the situation where all processing is done on a single disk drive, with the typical arrangement

of a single bank of read/write heads that move together over a stack of platters. If the sorting process involves reading from an input file, alternated with writing to an output file, then the I/O head will continuously seek between the input file and the output file. Similarly, if two input files are being processed simultaneously (such as during a merge process), then the I/O head will continuously seek between these two files.

The moral is that, with a single disk drive, there often is no such thing as efficient sequential processing of a data file. Thus, a sorting algorithm might be more efficient if it performs a smaller number of non-sequential disk operations rather than a larger number of logically sequential disk operations that require a large number of seeks in practice.

As mentioned previously, the record size might be quite large compared to the size of the key. For example, payroll entries for a large business might each store hundreds of bytes of information including the name, ID, address, and job title for each employee. The sort key might be the ID number, requiring only a few bytes. The simplest sorting algorithm might be to process such records as a whole, reading the entire record whenever it is processed. However, this will greatly increase the amount of I/O required, because only a relatively few records will fit into a single disk block. Another alternative is to do a **key sort**. Under this method, the keys are all read and stored together in an **index file**, where each key is stored along with a pointer indicating the position of the corresponding record in the original data file. The key and pointer combination should be substantially smaller than the size of the original record; thus, the index file will be much smaller than the complete data file. The index file will then be sorted, requiring much less I/O because the index records are smaller than the complete records.

Once the index file is sorted, it is possible to reorder the records in the original database file. This is typically not done for two reasons. First, reading the records in sorted order from the record file requires a random access for each record. This can take a substantial amount of time and is only of value if the complete collection of records needs to be viewed or processed in sorted order (as opposed to a search for selected records). Second, database systems typically allow searches to be done on multiple keys. For example, today's processing might be done in order of ID numbers. Tomorrow, the boss might want information sorted by salary. Thus, there might be no single "sorted" order for the full record. Instead, multiple index files are often maintained, one for each sort key. These ideas are explored further in Chapter 10.

8.5.1 Simple Approaches to External Sorting

If your operating system supports virtual memory, the simplest "external" sort is to read the entire file into virtual memory and run an internal sorting method such as Quicksort. This approach allows the virtual memory manager to use its normal buffer pool mechanism to control disk accesses. Unfortunately, this might not always be a viable option. One potential drawback is that the size of virtual memory is usually limited to something much smaller than the disk space available. Thus, your input file might not fit into virtual memory. Limited virtual memory can be overcome by adapting an internal sorting method to make use of your own buffer pool.

A more general problem with adapting an internal sorting algorithm to external sorting is that it is not likely to be as efficient as designing a new algorithm with the specific goal of minimizing disk fetches. Consider the simple adaptation of Quicksort to use a buffer pool. Quicksort begins by processing the entire array of records, with the first partition step moving indices inward from the two ends. This can be implemented efficiently using a buffer pool. However, the next step is to process each of the subarrays, followed by processing of sub-subarrays, and so on. As the subarrays get smaller, processing quickly approaches random access to the disk drive. Even with maximum use of the buffer pool, Quicksort still must read and write each record $\log n$ times on average. We can do much better. Finally, even if the virtual memory manager can give good performance using a standard quicksort, wthi will come at the cost of using a lot of the system's working memory, which will mean that the system cannot use this space for other work. Better methods can save time while also using less memory.

Our approach to external sorting is derived from the Mergesort algorithm. The simplest form of external Mergesort performs a series of sequential passes over the records, merging larger and larger sublists on each pass. The first pass merges sublists of size 1 into sublists of size 2; the second pass merges the sublists of size 2 into sublists of size 4; and so on. A sorted sublist is called a **run**. Thus, each pass is merging pairs of runs to form longer runs. Each pass copies the contents of the file to another file. Here is a sketch of the algorithm, as illustrated by Figure 8.6.

1. Split the original file into two equal-sized **run files**.
2. Read one block from each run file into input buffers.
3. Take the first record from each input buffer, and write a run of length two to an output buffer in sorted order.
4. Take the next record from each input buffer, and write a run of length two to a second output buffer in sorted order.

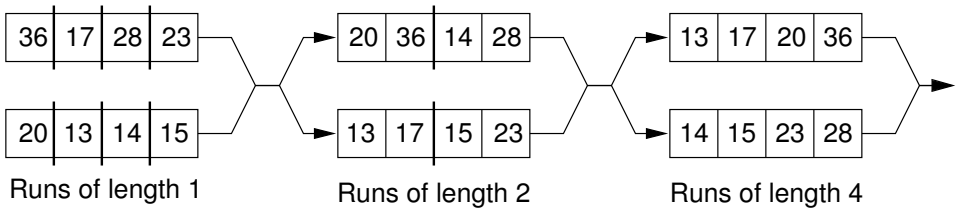

Figure 8.6 A simple external Mergesort algorithm. Input records are divided equally between two input files. The first runs from each input file are merged and placed into the first output file. The second runs from each input file are merged and placed in the second output file. Merging alternates between the two output files until the input files are empty. The roles of input and output files are then reversed, allowing the runlength to be doubled with each pass.

5. Repeat until finished, alternating output between the two output run buffers. Whenever the end of an input block is reached, read the next block from the appropriate input file. When an output buffer is full, write it to the appropriate output file.
6. Repeat steps 2 through 5, using the original output files as input files. On the second pass, the first two records of each input run file are already in sorted order. Thus, these two runs may be merged and output as a single run of four elements.
7. Each pass through the run files provides larger and larger runs until only one run remains.

Example 8.6 Using the input of Figure 8.6, we first create runs of length one split between two input files. We then process these two input files sequentially, making runs of length two. The first run has the values 20 and 36, which are output to the first output file. The next run has 13 and 17, which is ouput to the second file. The run 14, 28 is sent to the first file, then run 15, 23 is sent to the second file, and so on. Once this pass has completed, the roles of the input files and output files are reversed. The next pass will merge runs of length two into runs of length four. Runs 20, 36 and 13, 17 are merged to send 13, 17, 20, 36 to the first output file. Then runs 14, 28 and 15, 23 are merged to send run 14, 15, 23, 28 to the second output file. In the final pass, these runs are merged to form the final run 13, 14, 15, 17, 20, 23, 28, 36.

This algorithm can easily take advantage of the double buffering techniques described in Section 8.3. Note that the various passes read the input run files se-

quentially and write the output run files sequentially. For sequential processing and double buffering to be effective, however, it is necessary that there be a separate I/O head available for each file. This typically means that each of the input and output files must be on separate disk drives, requiring a total of four disk drives for maximum efficiency.

The external Mergesort algorithm just described requires that $\log n$ passes be made to sort a file of n records. Thus, each record must be read from disk and written to disk $\log n$ times. The number of passes can be significantly reduced by observing that it is not necessary to use Mergesort on small runs. A simple modification is to read in a block of data, sort it in memory (perhaps using Quicksort), and then output it as a single sorted run.

Example 8.7 Assume that we have blocks of size 4KB, and records are eight bytes with four bytes of data and a 4-byte key. Thus, each block contains 512 records. Standard Mergesort would require nine passes to generate runs of 512 records, whereas processing each block as a unit can be done in one pass with an internal sort. These runs can then be merged by Mergesort. Standard Mergesort requires eighteen passes to process 256K records. Using an internal sort to create initial runs of 512 records reduces this to one initial pass to create the runs and nine merge passes to put them all together, approximately half as many passes.

We can extend this concept to improve performance even further. Available main memory is usually much more than one block in size. If we process larger initial runs, then the number of passes required by Mergesort is further reduced. For example, most modern computers can provide tens or even hundreds of megabytes of RAM to the sorting program. If all of this memory (excepting a small amount for buffers and local variables) is devoted to building initial runs as large as possible, then quite large files can be processed in few passes. The next section presents a technique for producing large runs, typically twice as large as could fit directly into main memory.

Another way to reduce the number of passes required is to increase the number of runs that are merged together during each pass. While the standard Mergesort algorithm merges two runs at a time, there is no reason why merging needs to be limited in this way. Section 8.5.3 discusses the technique of multiway merging.

Over the years, many variants on external sorting have been presented, but all are based on the following two steps:

1. Break the file into large initial runs.

2. Merge the runs together to form a single sorted file.

8.5.2 Replacement Selection

This section treats the problem of creating initial runs as large as possible from a disk file, assuming a fixed amount of RAM is available for processing. As mentioned previously, a simple approach is to allocate as much RAM as possible to a large array, fill this array from disk, and sort the array using Quicksort. Thus, if the size of memory available for the array is M records, then the input file can be broken into initial runs of length M. A better approach is to use an algorithm called **replacement selection** that, on average, creates runs of $2M$ records in length. Replacement selection is actually a slight variation on the Heapsort algorithm. The fact that Heapsort is slower than Quicksort is irrelevant in this context because I/O time will dominate the total running time of any reasonable external sorting algorithm. Building longer initial runs will reduce the total I/O time required.

Replacement selection views RAM as consisting of an array of size M in addition to an input buffer and an output buffer. (Additional I/O buffers might be desirable if the operating system supports double buffering, because replacement selection does sequential processing on both its input and its output.) Imagine that the input and output files are streams of records. Replacement selection takes the next record in sequential order from the input stream when needed, and outputs runs one record at a time to the output stream. Buffering is used so that disk I/O is performed one block at a time. A block of records is initially read and held in the input buffer. Replacement selection removes records from the input buffer one at a time until the buffer is empty. At this point the next block of records is read in. Output to a buffer is similar: Once the buffer fills up it is written to disk as a unit. This process is illustrated by Figure 8.7.

Replacement selection works as follows. Assume that the main processing is done in an array of size M records.

1. Fill the array from disk. Set LAST $= M - 1$.
2. Build a min-heap. (Recall that a min-heap is defined such that the record at each node has a key value *less* than the key values of its children.)
3. Repeat until the array is empty:
   (a) Send the record with the minimum key value (the root) to the output buffer.
   (b) Let *R* be the next record in the input buffer. If *R*'s key value is greater than the key value just output ...
      i. Then place *R* at the root.

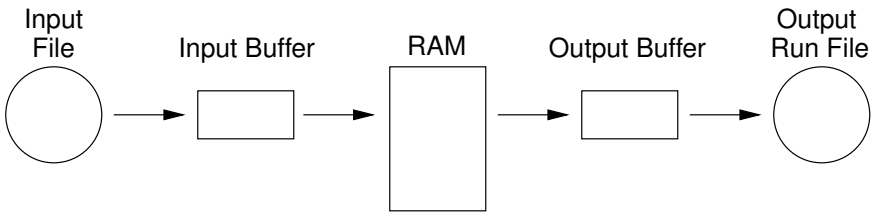

Figure 8.7 Overview of replacement selection. Input records are processed sequentially. Initially RAM is filled with M records. As records are processed, they are written to an output buffer. When this buffer becomes full, it is written to disk. Meanwhile, as replacement selection needs records, it reads them from the input buffer. Whenever this buffer becomes empty, the next block of records is read from disk.

ii. Else replace the root with the record in array position LAST, and place *R* at position LAST. Set LAST = LAST − 1.

(c) Sift down the root to reorder the heap.

When the test at step 3(b) is successful, a new record is added to the heap, eventually to be output as part of the run. As long as records coming from the input file have key values greater than the last key value output to the run, they can be safely added to the heap. Records with smaller key values cannot be output as part of the current run because they would not be in sorted order. Such values must be stored somewhere for future processing as part of another run. However, because the heap will shrink by one element in this case, there is now a free space where the last element of the heap used to be! Thus, replacement selection will slowly shrink the heap and at the same time use the discarded heap space to store records for the next run. Once the first run is complete (i.e., the heap becomes empty), the array will be filled with records ready to be processed for the second run. Figure 8.8 illustrates part of a run being created by replacement selection.

It should be clear that the minimum length of a run will be M records if the size of the heap is M, because at least those records originally in the heap will be part of the run. Under good conditions (e.g., if the input is sorted), then an arbitrarily long run is possible. In fact, the entire file could be processed as one run. If conditions are bad (e.g., if the input is reverse sorted), then runs of only size M result.

What is the expected length of a run generated by replacement selection? It can be deduced from an analogy called the **snowplow argument**. Imagine that a snowplow is going around a circular track during a heavy, but steady, snowstorm. After the plow has been around at least once, snow on the track must be as follows. Immediately behind the plow, the track is empty because it was just plowed. The greatest level of snow on the track is immediately in front of the plow, because

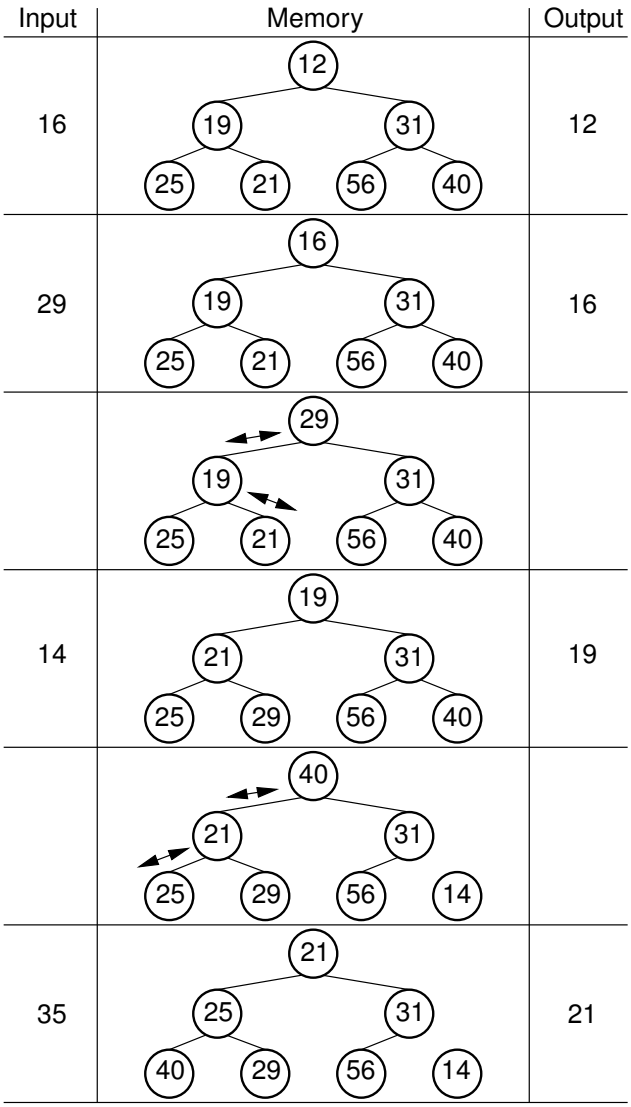

Figure 8.8 Replacement selection example. After building the heap, root value 12 is output and incoming value 16 replaces it. Value 16 is output next, replaced with incoming value 29. The heap is reordered, with 19 rising to the root. Value 19 is output next. Incoming value 14 is too small for this run and is placed at end of the array, moving value 40 to the root. Reordering the heap results in 21 rising to the root, which is output next.

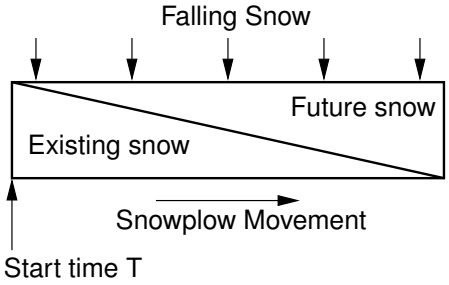

Figure 8.9 The snowplow analogy showing the action during one revolution of the snowplow. A circular track is laid out straight for purposes of illustration, and is shown in cross section. At any time T, the most snow is directly in front of the snowplow. As the plow moves around the track, the same amount of snow is always in front of the plow. As the plow moves forward, less of this is snow that was in the track at time T; more is snow that has fallen since.

this is the place least recently plowed. At any instant, there is a certain amount of snow S on the track. Snow is constantly falling throughout the track at a steady rate, with some snow falling "in front" of the plow and some "behind" the plow. (On a circular track, everything is actually "in front" of the plow, but Figure 8.9 illustrates the idea.) During the next revolution of the plow, all snow S on the track is removed, plus half of what falls. Because everything is assumed to be in steady state, after one revolution S snow is still on the track, so $2S$ snow must fall during a revolution, and $2S$ snow is removed during a revolution (leaving S snow behind).

At the beginning of replacement selection, nearly all values coming from the input file are greater (i.e., "in front of the plow") than the latest key value output for this run, because the run's initial key values should be small. As the run progresses, the latest key value output becomes greater and so new key values coming from the input file are more likely to be too small (i.e., "after the plow"); such records go to the bottom of the array. The total length of the run is expected to be twice the size of the array. Of course, this assumes that incoming key values are evenly distributed within the key range (in terms of the snowplow analogy, we assume that snow falls evenly throughout the track). Sorted and reverse sorted inputs do not meet this expectation and so change the length of the run.

8.5.3 Multiway Merging

The second stage of a typical external sorting algorithm merges the runs created by the first stage. Assume that we have R runs to merge. If a simple two-way merge is used, then R runs (regardless of their sizes) will require $\log R$ passes through the file. While R should be much less than the total number of records (because

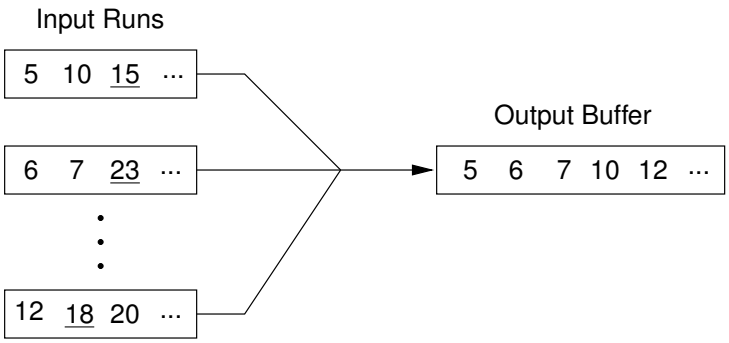

Figure 8.10 Illustration of multiway merge. The first value in each input run is examined and the smallest sent to the output. This value is removed from the input and the process repeated. In this example, values 5, 6, and 12 are compared first. Value 5 is removed from the first run and sent to the output. Values 10, 6, and 12 will be compared next. After the first five values have been output, the "current" value of each block is the one underlined.

the initial runs should each contain many records), we would like to reduce still further the number of passes required to merge the runs together. Note that two-way merging does not make good use of available memory. Because merging is a sequential process on the two runs, only one block of records per run need be in memory at a time. Keeping more than one block of a run in memory at any time will not reduce the disk I/O required by the merge process. Thus, most of the space just used by the heap for replacement selection (typically many blocks in length) is not being used by the merge process.

We can make better use of this space and at the same time greatly reduce the number of passes needed to merge the runs if we merge several runs at a time. Multiway merging is similar to two-way merging. If we have B runs to merge, with a block from each run available in memory, then the B-way merge algorithm simply looks at B values (the frontmost value for each input run) and selects the smallest one to output. This value is removed from its run, and the process is repeated. When the current block for any run is exhausted, the next block from that run is read from disk. Figure 8.10 illustrates a multiway merge.

Conceptually, multiway merge assumes that each run is stored in a separate file. However, this is not necessary in practice. We only need to know the position of each run within a single file, and use **`seek`** to move to the appropriate block whenever we need new data from a particular run. Naturally, this approach destroys the ability to do sequential processing on the input file. However, if all runs were stored on a single disk drive, then processing would not be truely sequential anyway because the I/O head would be alternating between the runs. Thus, multiway merging

replaces several (potentially) sequential passes with a single random access pass. If the processing would not be sequential anyway (such as when all processing is on a single disk drive), no time is lost by doing so.

Multiway merging can greatly reduce the number of passes required. If there is room in memory to store one block for each run, then all runs can be merged in a single pass. Thus, replacement selection can build initial runs in one pass, and multiway merging can merge all runs in one pass, yielding a total cost of two passes. However, for truly large files, there might be too many runs for each to get a block in memory. If there is room to allocate B blocks for a B-way merge, and the number of runs R is greater than B, then it will be necessary to do multiple merge passes. In other words, the first B runs are merged, then the next B, and so on. These super-runs are then merged by subsequent passes, B super-runs at a time.

How big a file can be merged in one pass? Assuming B blocks were allocated to the heap for replacement selection (resulting in runs of average length $2B$ blocks), followed by a B-way merge, we can process on average a file of size $2B^2$ blocks in a single multiway merge. $2B^{k+1}$ blocks on average can be processed in k B-way merges. To gain some appreciation for how quickly this grows, assume that we have available 0.5MB of working memory, and that a block is 4KB, yielding 128 blocks in working memory. The average run size is 1MB (twice the working memory size). In one pass, 128 runs can be merged. Thus, a file of size 128MB can, on average, be processed in two passes (one to build the runs, one to do the merge) with only 0.5MB of working memory. A larger block size would reduce the size of the file that can be processed in one merge pass for a fixed-size working memory; a smaller block size or larger working memory would increase the file size that can be processed in one merge pass. With 0.5MB of working memory and 4KB blocks, a file of size 16 gigabytes could be processed in two merge passes, which is big enough for most applications. Thus, this is a very effective algorithm for single disk drive external sorting.

Figure 8.11 shows a comparison of the running time to sort various-sized files for the following implementations: (1) standard Mergesort with two input runs and two output runs, (2) two-way Mergesort with large initial runs (limited by the size of available memory), and (3) R-way Mergesort performed after generating large initial runs. In each case, the file was composed of a series of four-byte records (a two-byte key and a two-byte data value), or 256K records per megabyte of file size. We can see from this table that using even a modest memory size (two blocks) to create initial runs results in a tremendous savings in time. Doing 4-way merges of the runs provides another considerable speedup, however largescale multi-way

File Size	Sort 1	Sort 2 Memory size (in blocks)				Sort 3 Memory size (in blocks)		
		2	4	16	256	2	4	16
1	0.61	0.27	0.24	0.19	0.10	0.21	0.15	0.13
	4,864	2,048	1,792	1,280	256	2,048	1,024	512
4	2.56	1.30	1.19	0.96	0.61	1.15	0.68	0.66*
	21,504	10,240	9,216	7,168	3,072	10,240	5,120	2,048
16	11.28	6.12	5.63	4.78	3.36	5.42	3.19	3.10
	94,208	49,152	45,056	36,864	20,480	49,152	24,516	12,288
256	220.39	132.47	123.68	110.01	86.66	115.73	69.31	68.71
	1,769K	1,048K	983K	852K	589K	1,049K	524K	262K

Figure 8.11 A comparison of three external sorts on a collection of small records for files of various sizes. Each entry in the table shows time in seconds and total number of blocks read and written by the program. File sizes are in Megabytes. For the third sorting algorithm, on file size of 4MB, the time and blocks shown in the last column are for a 32-way merge. 32 is used instead of 16 because 32 is a root of the number of blocks in the file (while 16 is not), thus allowing the same number of runs to be merged at every pass.

merges for R beyond about 4 or 8 runs does not help much because a lot of time is spent determining which is the next smallest element among the R runs.

We see from this experiment that building large initial runs reduces the running time to slightly more than one third that of standard Mergesort, depending on file and memory sizes. Using a multiway merge further cuts the time nearly in half.

In summary, a good external sorting algorithm will seek to do the following:

- Make the initial runs as long as possible.
- At all stages, overlap input, processing, and output as much as possible.
- Use as much working memory as possible. Applying more memory usually speeds processing. In fact, more memory will have a greater effect than a faster disk. A faster CPU is unlikely to yield much improvement in running time for external sorting, because disk I/O speed is the limiting factor.
- If possible, use additional disk drives for more overlapping of processing with I/O, and to allow for sequential file processing.

8.6 Further Reading

A good general text on file processing is Folk and Zoellig's *File Structures: A Conceptual Toolkit* [FZ98]. A somewhat more advanced discussion on key issues in file processing is Betty Salzberg's *File Structures: An Analytical Approach* [Sal88].

A great discussion on external sorting methods can be found in Salzberg's book. The presentation in this chapter is similar in spirit to Salzberg's.

For details on disk drive modeling and measurement, see the article by Ruemmler and Wilkes, "An Introduction to Disk Drive Modeling" [RW94]. See Andrew S. Tanenbaum's *Structured Computer Organization* [Tan06] for an introduction to computer hardware and organization. An excellent, detailed description of memory and hard disk drives can be found online at "The PC Guide," by Charles M. Kozierok [Koz05] (`www.pcguide.com`). The PC Guide also gives detailed descriptions of the Microsoft Windows and UNIX (Linux) file systems.

See "Outperforming LRU with an Adaptive Replacement Cache Algorithm" by Megiddo and Modha for an example of a more sophisticated algorithm than LRU for managing buffer pools.

The snowplow argument comes from Donald E. Knuth's *Sorting and Searching* [Knu98], which also contains a wide variety of external sorting algorithms.

8.7 Exercises

8.1 Computer memory and storage prices change rapidly. Find out what the current prices are for the media listed in Figure 8.1. Does your information change any of the basic conclusions regarding disk processing?

8.2 Assume a disk drive from the late 1990s is configured as follows. The total storage is approximately 675MB divided among 15 surfaces. Each surface has 612 tracks; there are 144 sectors/track, 512 bytes/sector, and 8 sectors/cluster. The disk turns at 3600 rpm. The track-to-track seek time is 20 ms, and the average seek time is 80 ms. Now assume that there is a 360KB file on the disk. On average, how long does it take to read all of the data in the file? Assume that the first track of the file is randomly placed on the disk, that the entire file lies on adjacent tracks, and that the file completely fills each track on which it is found. A seek must be performed each time the I/O head moves to a new track. Show your calculations.

8.3 Using the specifications for the disk drive given in Exercise 8.2, calculate the expected time to read one entire track, one sector, and one byte. Show your calculations.

8.4 Using the disk drive specifications given in Exercise 8.2, calculate the time required to read a 10MB file assuming

(a) The file is stored on a series of contiguous tracks, as few tracks as possible.

(b) The file is spread randomly across the disk in 4KB clusters.

Show your calculations.

8.5 Assume that a disk drive is configured as follows. The total storage is approximately 1033MB divided among 15 surfaces. Each surface has 2100 tracks, there are 64 sectors/track, 512 bytes/sector, and 8 sectors/cluster. The disk turns at 7200 rpm. The track-to-track seek time is 3 ms, and the average seek time is 20 ms. Now assume that there is a 512KB file on the disk. On average, how long does it take to read all of the data on the file? Assume that the first track of the file is randomly placed on the disk, that the entire file lies on contiguous tracks, and that the file completely fills each track on which it is found. Show your calculations.

8.6 Using the specifications for the disk drive given in Exercise 8.5, calculate the expected time to read one entire track, one sector, and one byte. Show your calculations.

8.7 Using the disk drive specifications given in Exercise 8.5, calculate the time required to read a 10MB file assuming

(a) The file is stored on a series of contiguous tracks, as few tracks as possible.

(b) The file is spread randomly across the disk in 4KB clusters.

Show your calculations.

8.8 A typical disk drive from 2004 has the following specifications.[2] The total storage is approximately 120GB on 6 platter surfaces or 20GB/platter. Each platter has 16K tracks with 2560 sectors/track (a sector holds 512 bytes) and 16 sectors/cluster. The disk turns at 7200 rpm. The track-to-track seek time is 2.0 ms, and the average seek time is 10.0 ms. Now assume that there is a 6MB file on the disk. On average, how long does it take to read all of the data on the file? Assume that the first track of the file is randomly placed on the disk, that the entire file lies on contiguous tracks, and that the file completely fills each track on which it is found. Show your calculations.

8.9 Using the specifications for the disk drive given in Exercise 8.8, calculate the expected time to read one entire track, one sector, and one byte. Show your calculations.

8.10 Using the disk drive specifications given in Exercise 8.8, calculate the time required to read a 10MB file assuming

[2]To make the exercise doable, this specification is completely ficticious with respect to the track and sector layout. While sectors do have 512 bytes, and while the number of platters and amount of data per track is plausible, the reality is that all modern drives use a zoned organization to keep the data density from inside to outside of the disk reasonably high. The rest of the numbers are typical for a drive from 2004.

(a) The file is stored on a series of contiguous tracks, as few tracks as possible.

(b) The file is spread randomly across the disk in 8KB clusters.

Show your calculations.

8.11 At the end of 2004, the fastest disk drive I could find specifications for was the Maxtor Atlas. This drive had a nominal capacity of 73.4GB using 4 platters (8 surfaces) or 9.175GB/surface. Assume there are 16,384 tracks with an average of 1170 sectors/track and 512 bytes/sector.[3] The disk turns at 15,000 rpm. The track-to-track seek time is 0.4 ms and the average seek time is 3.6 ms. How long will it take on average to read a 6MB file, assuming that the first track of the file is randomly placed on the disk, that the entire file lies on contiguous tracks, and that the file completely fills each track on which it is found. Show your calculations.

8.12 Using the specifications for the disk drive given in Exercise 8.11, calculate the expected time to read one entire track, one sector, and one byte. Show your calculations.

8.13 Using the disk drive specifications given in Exercise 8.11, calculate the time required to read a 10MB file assuming

(a) The file is stored on a series of contiguous tracks, as few tracks as possible.

(b) The file is spread randomly across the disk in 8KB clusters.

Show your calculations.

8.14 Prove that two tracks selected at random from a disk are separated on average by one third the number of tracks on the disk.

8.15 Assume that a file contains one million records sorted by key value. A query to the file returns a single record containing the requested key value. Files are stored on disk in sectors each containing 100 records. Assume that the average time to read a sector selected at random is 10.0 ms. In contrast, it takes only 2.0 ms to read the sector adjacent to the current position of the I/O head. The "batch" algorithm for processing queries is to first sort the queries by order of appearance in the file, and then read the entire file sequentially, processing all queries in sequential order as the file is read. This algorithm implies that the queries must all be available before processing begins. The "interactive" algorithm is to process each query in order of its arrival, searching for the requested sector each time (unless by chance two queries in a row are to the same sector). Carefully define under what conditions the batch method is more efficient than the interactive method.

[3] Again, this track layout does does not account for the zoned arrangement on modern disk drives.

8.16 Assume that a virtual memory is managed using a buffer pool. The buffer pool contains five buffers and each buffer stores one block of data. Memory accesses are by block ID. Assume the following series of memory accesses takes place:

5 2 5 12 3 6 5 9 3 2 4 1 5 9 8 15 3 7 2 5 9 10 4 6 8 5

For each of the following buffer pool replacement strategies, show the contents of the buffer pool at the end of the series, and indicate how many times a block was found in the buffer pool (instead of being read into memory). Assume that the buffer pool is initially empty.

(a) First-in, first out.
(b) Least frequently used (with counts kept only for blocks currently in memory, counts for a page are lost when that page is removed, and the oldest item with the smallest count is removed when there is a tie).
(c) Least frequently used (with counts kept for all blocks, and the oldest item with the smallest count is removed when there is a tie).
(d) Least recently used.
(e) Most recently used (replace the block that was most recently accessed).

8.17 Suppose that a record is 32 bytes, a block is 1024 bytes (thus, there are 32 records per block), and that working memory is 1MB (there is also additional space available for I/O buffers, program variables, etc.). What is the *expected* size for the largest file that can be merged using replacement selection followed by a *single* pass of multiway merge? Explain how you got your answer.

8.18 Assume that working memory size is 256KB broken into blocks of 8192 bytes (there is also additional space available for I/O buffers, program variables, etc.). What is the *expected* size for the largest file that can be merged using replacement selection followed by *two* passes of multiway merge? Explain how you got your answer.

8.19 Prove or disprove the following proposition: Given space in memory for a heap of M records, replacement selection will completely sort a file if no record in the file is preceded by M or more keys of greater value.

8.20 Imagine a database containing ten million records, with each record being 100 bytes long. Provide an estimate of the time it would take (in seconds) to sort the database on a typical workstation.

8.21 Assume that a company has a computer configuration satisfactory for processing their monthly payroll. Further assume that the bottleneck in payroll

processing is a sorting operation on all of the employee records, and that an external sorting algorithm is used. The company's payroll program is so good that it plans to hire out its services to do payroll processing for other companies. The president has an offer from a second company with 100 times as many employees. She realizes that her computer is not up to the job of sorting 100 times as many records in an acceptable amount of time. Describe what impact each of the following modifications to the computing system is likely to have in terms of reducing the time required to process the larger payroll database.

(a) A factor of two speedup to the CPU.
(b) A factor of two speedup to disk I/O time.
(c) A factor of two speedup to main memory access time.
(d) A factor of two increase to main memory size.

8.22 How can the external sorting algorithm described in this chapter be extended to handle variable-length records?

8.8 Projects

8.1 For a database application, assume it takes 10 ms to read a block from disk, 1 ms to search for a record in a block stored in memory, and that there is room in memory for a buffer pool of 5 blocks. Requests come in for records, with the request specifying which block contains the record. If a block is accessed, there is a 10% probability for each of the next ten requests that the request will be to the same block. What will be the expected performance improvement for each of the following modifications to the system?

(a) Get a CPU that is twice as fast.
(b) Get a disk drive that is twice as fast.
(c) Get enough memory to double the buffer pool size.

Write a simulation to analyze this problem.

8.2 Pictures are typically stored as an array, row by row, on disk. Consider the case where the picture has 16 colors. Thus, each pixel can be represented using 4 bits. If you allow 8 bits per pixel, no processing is required to unpack the pixels (because a pixel corresponds to a byte, the lowest level of addressing on most machines). If you pack two pixels per byte, space is saved but the pixels must be unpacked. Which takes more time to read from disk and access every pixel of the image: 8 bits per pixel, or 4 bits per pixel with 2 pixels per byte? Program both and compare the times.

8.3 Implement a disk-based buffer pool class based on the LRU buffer pool replacement strategy. Disk blocks are numbered consecutively from the beginning of the file with the first block numbered as 0. Assume that blocks are 4096 bytes in size, with the first 4 bytes used to store the block ID corresponding to that buffer. Use the first **`BufferPool`** abstract class given in Section 8.3 as the basis for your implementation.

8.4 Implement an external sort based on replacement selection and multiway merging as described in this chapter. Test your program both on files with small records and on files with large records. For what size record do you find that key sorting would be worthwhile?

8.5 Implement a quicksort for large files on disk by replacing all array access in the normal quicksort application with access to a virtual array implemented using a buffer pool. That is, whenever a record in the array would be read or written by quicksort, use a call to a buffer pool function instead. Compare the running time of this implementation with implementations for external sorting based on mergesort as described in this chapter.

8.6 Section 8.5.1 suggests that an easy modification to the basic 2-way mergesort is to read in a large chunk of data into main memory, sort it with quicksort, and write it out for initial runs. Then, a standard 2-way merge is used in a series of passes to merge the runs together. However, this makes use of only two blocks of working memory at a time. Each block read is essentially random access, because the various files are read in an unknown order, even though each of the input and output files is processed sequentially on each pass. A possible improvement would be, on the merge passes, to divide working memory into four equal sections. One section is allocated to each of the two input files and two output files. All reads during merge passes would be in full sections, rather than single blocks. While the total number of blocks read and written would be the same as a regular 2-way mergsort, it is possible that this would speed processing because a series of blocks that are logically adjacent in the various input and output files would be read/written each time. Implement this variation, and compare its running time against a standard series of 2-way merge passes that read/write only a single block at a time. Before beginning implementation, write down your hypothesis on how the running time will be affected by this change. After implementing, did you find that this change has any meaningful effect on performance?