Random Access Filing

Two types of storage

Digital devices have two broad types of storage - volatile and non-volatile.

Volatile storage means the data is lost when power is removed. When the device is switched off, the information held in volatile storage is wiped. Memory (RAM) is volatile. In Java data structures such as ArrayLists are held in volatile memory. Physically, RAM is made of a very large number of electronic 'gates' which can be on or off, storing a 0 or 1. The gates stay in whatever state they are in until either they are 'written to' with a new bit, or power is lost.

Non-volatile storage keeps data even when power is removed. One form is a hard disc drive HDD. This consists of a spinning metal disc with a magnetic coating, and a head which can move across the disc. The head can magnetise a point on the disc in one direction (N-S) or the other (S-N), storing a 1 or a 0. The magnetisation state is kept when power is removed.

Another non-volatile storage medium is flash memory, used in USB flash drives, memory cards and solid-state drives SSDs. Flash memory is like RAM in that it consists of electronic gates, but these keep their state when power is removed.

Non-volatile storage is sometimes called file storage, because data is logically organised into files, grouped into folders (directories).

RAM is organised into 8 bit bytes, with each byte having a distinct address.

Usually file storage is much *larger*, but *slower*, than RAM. Program code is kept in executable files on a file system, but is loaded into RAM when it s being run.

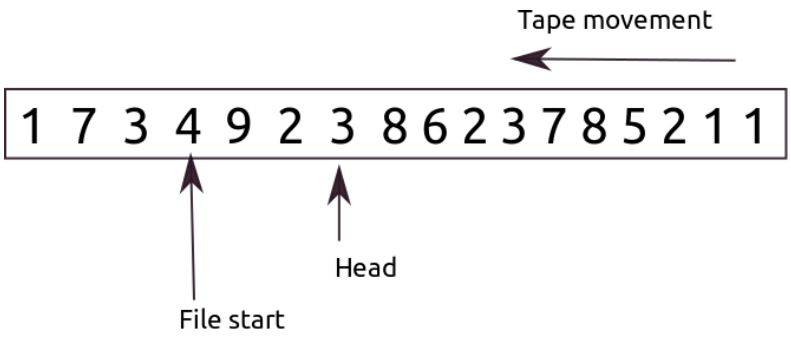
Random access

Random access means that all parts of the storage can be accessed at the same speed.

Suppose a processor needs to read the byte held at address 1000. Or to read the byte at address 5000. The speed of both is the same. Why? Because of the way RAM works. The address is fed in on an address *bus*, a read is signalled, and the data comes back on the data bus. Because of the electronics, all addresses are equally fast.

Serial access

In the early days of computing, files were held on magnetic tape, like this:



The data is held on a tape, which can be moved past a head, which can read or write data. The tape can be moved one way only (or completely re-wound).

In the diagram, only two things can happen here:

1. Read a value from the file. This will read 3, and the tape will be moved so that the 8 lies under the head.

2. Write a new value into the file. This new value would over-write the 3, which would be erased. The tape moves so that 8 lies under the head.

This is *serial access*. We access data in series. The only value we can get is the 'next one', the value that lies under the head.

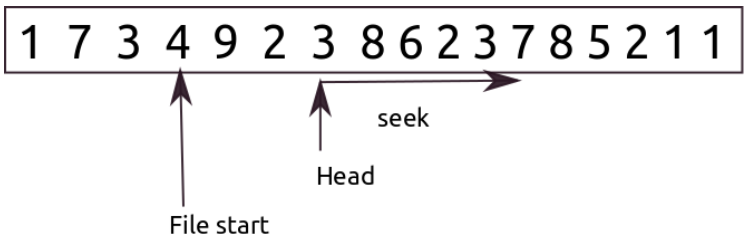
Tape storage is serial access, because this is how tapes work. RAM is random access - we can read or write any address at the same speed. With a serial access device we can only read or write the next byte.

Because tape was initially the only file storage available, algorithms were developed to process data (such as sort) serially. They are never very fast.

Random access on modern file storage

Most modern devices can use random access. This is true of flash memory, and more or less true of hdd.

Even so, we still often think of file storage as using serial access. If we use random access, we can make the head 'seek' a new position through the file, a given number of bytes from its current position:



So here, if we do a seek 5 bytes forward, then do a read, we will get a 7. This is random access.

Modern devices like a flash memory stick do not actually have read/write heads. But the operating system will track a 'current read/write position' through the file, so we can still think of it this way.

File formats and records

Some types of binary files are sequences of bytes in some format. As an example, the MS Windows BMP bitmapped image file contains in outline:

a file header. This starts with the bytes hex 42 4D (a file signature 'magic number' meaning its a bitmap) and contains the size of the file (to check validity)

a bitmap header. This includes fields such as image width and height in pixels

a colour table. Each pixel has a colour index number. The colour table says the red green and blue components of each colour index

the image data, as a pixel array, column by column and row by row

Each binary file format has its own specification, which enables software to read and 'parse' the file.

But many data files are made of a sequence of 'records'. A record is made of a set of fields, one of which will probably be a unique key field. A record corresponds to a row in a database table, or a node in a data structure in memory.

Serial access restricts us to reading all records, in order, from the start.

But random access means we can move to and read at any byte position through the file.

ByteBuffer

Reading or writing single bytes is very inefficient. It is much faster to read and write blocks of data. A buffer is a section of memory used to hold data before being written out, or after being read in.

Java does random access fling through a FileChannel instance, and this reads and writes data in a ByteBuffer. So we need to study ByteBuffers first.

A ByteBuffer is like a byte array - in fact it wraps a byte array. But it tracks a current position, where read and writes take place - so we can do random access or serial access with it, in memory. For example:

```
// make a buffer of 12 bytes
ByteBuffer byteBuffer = ByteBuffer.allocate(12);
// write an int (3) at the start of the buffer
// ints are 4 bytes long, so this will be byte positions 0 1 2 and 3
byteBuffer.putInt(3);
// write another, at 4 5 6 7
byteBuffer.putInt(4);
// and another. buffer is now full
byteBuffer.putInt(5);
// write at index 4
// iow over the 4
byteBuffer.putInt(4,6);
// back to start
byteBuffer.rewind();
// output them
while (byteBuffer.remaining()>0)
  System.out.println(byteBuffer.getInt()); // 3 6 5
```

The rewind method indicates this is still being thought of as a tape.

We are reading ints out of the buffer. This only works because we knew we wrote ints into it. The buffer is just a sequence of bytes.

Java's SeekableByteChannel

The ByteChannel interface is a channel that can read and write bytes.

A SeekableByteChannel is a sub-interface of this. It has a position which controls where the read and write operations take place. The position() method sets this.

A FileChannel is a class which implements SeekableByteChannel - so we can do random access filing with it.

As an example - the following code opens a file, then writes the string 0123456789 to it, at the start of the file. Then it moves to 5 bytes from the start, and writes it again:

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import static java.nio.file.StandardOpenOption.WRITE;
import java.nio.file.Path;
import java.nio.file.Paths;
import static java.nio.file.StandardOpenOption.CREATE;
import static java.nio.file.StandardOpenOption.READ;

public class Test {

  public static void main(String[] args) {
    // create a random access file and write to it
    try {
      // data to write
      String s = "0123456789\n";
      byte data[] = s.getBytes();// get string as byte array
       // use it as a byte buffer
      ByteBuffer outputBuffer = ByteBuffer.wrap(data);

      Path path = Paths.get("myfile.dat");
      // create a file system path from a string
      // open a FileChannel on this path
      // A FileChannel implements SeekableByteChannel
      FileChannel fc = FileChannel.open(path, READ, WRITE, CREATE);
      fc.position(0); // go to beginning
      while (outputBuffer.hasRemaining()) {
        fc.write(outputBuffer);
      }
      outputBuffer.rewind();
      fc.position(5); // go to 5 bytes from start
      while (outputBuffer.hasRemaining()) {
        fc.write(outputBuffer);
      }
      outputBuffer.rewind();
      fc.close();

    } catch (IOException ex) {
      ex.printStackTrace();
    }

  }

}
```

After this the file contains

012340123456789

Fixed length records

If the file is made of records with a fixed length in bytes, we can calculate where a given record is. If each record is b bytes long, then the nth record is simply n X b bytes from the file start.

We give a simple example.

We start with a class defining a record. Each record has an int key named ID, and one data field, which will contain 10 random characters. Each new record will have an ID which will be 1 or 2 more than the previous record (for a reason to be explained later):

```
class DataRecord {
// static members

  static int lastID = 0;
  static final int charLength = 10;
  static Random rng = new Random();
  static final int recordLength = 4 + charLength * 2;
  static final ByteBuffer buffer = ByteBuffer.allocate(recordLength);

  // data fields
  int ID;
  char[] data = new char[charLength];

  DataRecord() { // constructor
    // ID will be 1 or 2 more than last one
    ID = lastID + rng.nextInt(2) + 1;
    lastID = ID;
    // data is random a to z
    for (int index = 0; index < charLength; index++) {
      data[index] = (char) ('a' + (rng.nextInt(26)));
    }

  }

  // write record to filechannel, sequentially
  public void write(FileChannel fc) {
    ..

  } // end write

  // read the ID of the record from a given record number
  public static int fetchKey(int recNo, FileChannel fc) {

  ..
  }

  public String toString() {
    return ID + " " + new String(data);
  }

}
```

A record has 1 4 byte int, and 10 characters, each of which is 2 bytes long. So we find teh length of a record in bytes as:

```
  static final int recordLength = 4 + charLength * 2;
```

We only need one buffer, to read and write records, so the buffer is static (this would be more efficient if we read and wrote more than one record at a time, but is a simple demonstration).

The write method writes a record into an open FileChannel, at its current position, which is updated. We put the data into the buffer, then write to the file:

```
  // write record to filechannel, sequentially
  public void write(FileChannel fc) {
```

```
    // put data in buffer
    buffer.rewind();
    buffer.putInt(ID);
    for (int index = 0; index < charLength; index++) {
      buffer.putChar(data[index]);
    }

    try { // write buffer
      buffer.rewind();
      fc.write(buffer);
      System.out.println("Writing " + ID);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  } // end write
```

We can use this in driving code opening a file and writing 10 new records into it, as:

```
try {
    Path path = Paths.get("myfile.dat");

    FileChannel fc = FileChannel.open(path, READ, WRITE, CREATE);
    fc.position(0);
    for (int i = 0; i < 10; i++) {
      DataRecord data = new DataRecord();
      data.write(fc);
    }
    ..


    fc.close();
  } catch (IOException ex) {
    ex.printStackTrace();
  }
}
```

Code to read a given record is

```
// read the ID of the record from a given record number
public static int fetchKey(int recNo, FileChannel fc) {
  int bytePosition = recNo * recordLength;
  buffer.rewind();
  try {
    fc.read(buffer, bytePosition);
  } catch (IOException ex) {
    ex.printStackTrace();
  }

  int thisID = buffer.getInt(0);
  return thisID;

}
```

We can use this:

```
    for (int i = 0; i < 10; i++) {
      DataRecord data = new DataRecord();
      data.write(fc);
    }
    for (int i = 0; i < 10; i++) {
      System.out.println(DataRecord.fetchKey(i, fc));
    }
```

A sample run is..

```
Writing 1
Writing 2
```

```
Writing 4
Writing 6
Writing 8
Writing 9
Writing 11
Writing 12
Writing 14
Writing 16
1
2
4
6
8
9
11
12
14
16
```

Binary search

A binary search is an efficient way of searching a data list which is in sorted order. It works by looking at the middle item. If it is too small, we work (recursively) on the 'top half', else we use the 'bottom half'. This continues until we find it, or the range shrinks to nothing.

This is much faster than a linear search, where we simply start at the beginning and continue reading the next value until we find it, or hit the end. But, the list must be sorted.

We cannot do a binary search using serial access, because we cannot go directly to the middle element. But we can using random access.

We can do this with our fixed length record file (this is why we generate the keys in order, to avoid having to sort them).

```
public static String get(int targetKey, FileChannel fc, int start, int end)
 {

   if (start+1==end) return "Not  present";
   int middle=(start+end)/2;
   int lookAt=fetchKey(middle, fc);
   if (lookAt==targetKey) return "Got it at "+middle;
   if (lookAt>targetKey) return get(targetKey, fc, start, middle);
   else
     return get(targetKey, fc, middle, end);
 }
```

which we can use as

```
 try {
      Path path = Paths.get("myfile.dat");

      FileChannel fc = FileChannel.open(path, READ, WRITE, CREATE);
      fc.position(0);
      for (int i = 0; i < 10; i++) {
        DataRecord data = new DataRecord();
        data.write(fc);
      }
      for (int i = 0; i < 10; i++) {
        System.out.println(DataRecord.fetchKey(i, fc));
      }

      System.out.println(DataRecord.get(8, fc, 0, 9));

      fc.close();
    } catch (IOException ex) {
```

```
      ex.printStackTrace();
    }
```

Sample run:

```
Writing 1
Writing 3
Writing 5
Writing 7
Writing 8
Writing 9
Writing 10
Writing 12
Writing 13
Writing 14
1
3
5
7
8
9
10
12
13
14
Got it at 4
```

A linear search is O[n], while a binary search is O[log2 n].

If we have one million records, we can search two million with just one more step.

Multiple indexes

Suppose we want to search a file on two different fields? We cannot have the file sorted in two different orders at the same time.

But we can use an index file. This has two fields - a key value field and a pointer into the main file to the record with that key. The index is sorted on the key, so we can do a fast binary search on it.

And we can have two indexes, on different fields. Like this:

Main file

Record number	Field A	Field B
1	5	8
2	7	5
3	1	1
4	9	7
5	12	10
6	13	11
7	6	4
8	8	2

Index on A

Key	Pointer
1	3
5	1
6	7
7	2
8	8
9	4
12	5
13	6

Index on B

Key	Pointer
1	3
2	8
4	7
5	2
7	4
8	1
10	5
11	6

For example, suppose we want the row with B=7. We do a binary search on the B index, and find it with pointer 4. That means record number 4 in the main file is the row with B=7.

This can be used in database servers to provide fast access to a table on multiple keys.