# File Organization - Syllabus Section 7

**Summary**

| File Structure Name | Storage Details | Access Method (searching) |
|---|---|---|
| **Sequential (serial)** file | **Ordered** (sorted) or **unordered** records | Sequential Access |
| **Parially Indexed** file | **Ordered** | Sequential access to index, followed by direct access to the first record in the group, then sequential search to find the desired record |
| **Fully Indexed** file | **Unordered** | Sequential access to the index, followed by direct access to the correct record |
| **Direct Access** file | Either ordered or unordered | A calculation (hash) provides the location of a record, followed by direct access to the record |

---

# Zonker's Fabulous File Fable (File Organization Examples)

Zonker loves computers. He knows everything about PCs and the Internet. He decides to "leverage" his knowledge and earn some money. He starts a dot-com business selling secret information about celebrities (movie stars, singers, etc).

He figures he can collect the information free from on-line newspapers and fan web-sites, and whenever he needs more info he'll just make it up, because nobody will notice the difference anyway (similar to some news services). He'll also provide a chat-room, so all the fans can trade more secret info with each other, and he can steal that information and sell it, too.

The success of the business will depend more on selling advertising than actually selling the info, although he will charge a small fee for the customers. He'll use e-mail to remind customers every week that they should check in and read the latest gossip. He reckons he can spend a year building up the business, sell out for a million dollars, and have an easy life.

---

This is a story about the development of a customer database. In all the examples, the file stores the **Name** of the user, the **E-mail address,** and a **password.** The procedures are written as basic examples – they do not function perfectly, and contain very little error-trapping code.

*** Warning - the Java code in the examples has **not been tested.** It is preliminary "pseudocode". ***
*** It may contain minor syntax errors, but hopefully contains no major conceptual errors. ***

## Stage 1 : Serial (unsorted) Text File

To get started, Zonker gives free subscriptions to all his friends (about 20 people). He asks them for their E-mail addresses and lets them choose a password. "How should I **store** the customer data?" he wonders. Then he remembers the programmers' secret weapon – ** **NOTEPAD** ** to the rescue! He fires up Notepad, types in his data, and saves it in a **text file** named **CUSTOMERS.TXT,** like this:

```
Allen Allstar AAllstar@hotmail.com AAAAAA

 Tommy Lee Tunes TomTune27@aol.com a1b2c3xxx

 Carla Charles TooCool@yahoo.com sekret

  ....

 Bobby Baker BB@hotmail.com BeeBee
```

It doesn't take long to type in the data - he just types it out of his head, so it isn't in order (unsorted).

Zonker needs a program to **automate** the **log-in** process on his web-site. He wants to use a Java program to **verify** the user-names and passwords when the users log in. He starts writing his Java program, but realizes that it's going to be pretty difficult to **parse** the strings in the text file (take them apart into the separate pieces: name, e-mail, password), because all the names and e-mail addresses are all different lengths. He can't rely on the blank spaces as **delimiters**, because some people have 2 names while others have 3. He considers the following **algorithm:**

```
Find the @ sign in the string

Search backward for the first blank space before the @

Take all the stuff before the blank as the full name

Search forward from the @ sign for a blank space

Take all the stuff after the blank as the password
```

Another possibility is to use /slashes/ as **delimiters,** like this:

```
Tommy Lee Tunes/TomTune27@aol.com/a1b2c3xxx
```

But then he realizes that **only the computer** is ever going to read the file, so he will make it convenient by writing **one field** on each line of the file, like this:

```
Allen Allstar
AAllstar@hotmail.com
AAAAAA
Tommy Lee Tunes
TomTune27@aol.com
a1b2c3xxx
Carla Charles
TooCool@yahoo.com
sekret
 ....
 ....
 ....
Bob Baker
BB@hotmail.com
BeeBee
```

Now each **record** is written on three lines as three separate **fields,** and the **sequential search (linear**

**search)** algorithm is easy to program. This **method** searches for a user by name, and returns the password if the user is found - otherwise it returns and empty String.

```
public String findPassword(String target)
{
   String name = "";
   String email = "";
   String password = "";
   String answer = "";

   BufferedReader customers =
        new BufferedReader(new FileReader("customers"));

   while (customers.ready())
   {
      name = customers.readLine();
      email = customers.readLine();
      password = customers.readLine();

      if (name == target)
      {  answer = password; }

   }

   customers.close();

   return answer;

}
```

## Stage 2 : Sequential (sorted) Text File

After a couple weeks, news has spread around the school about Zonker's celebrity gossip site. Lots more people have signed up, and he has over a hundred names in the file. The new customers paid 5 Euros each for lifetime memberships - the original customers got in for free. One of the new members is angry because she paid her membership fee but can't log-in. Zonker has debugged the program and is convinced there is nothing wrong. He prints out all the data to check through it, thinking maybe he left out a line (field) in one of the entries - this would mess up the program, which always reads 3 fields for each record. There are no lines missing, but he notices the same name is in the file twice, in two different places. Apparently the user has been typing in one password, but the program keeps finding the other password and rejecting her. Zonker realizes that this problem would have been easier to see if the names were **sorted**. Then the duplicate name would be easy to notice.

How to sort the file? He could program a bubble-sort, but the separate fields make that messy. He tries to do it by cutting and pasting in Notepad. It only takes about 20 minutes to do. He realizes that he can easily **insert** new records in the right place in the file, so it will always be sorted. He decides to keep the file in sorted order all the time, by using Notepad to insert new records in the correct order..

Now that the file is **sorted**, he can speed up the FIND procedure by having it "stop early" when it does not find a name. It only requires one change in the code:

```
while (customers.ready() && (target.compareTo(name)>=0) )
```

As soon as the TARGET is "smaller" than the NAME in the file (target **precedes** name), the program can stop because the rest of the names will be even bigger (alphabetically). This works because the file is sorted. On the average, the FIND routine now reads half the file instead of the entire file. Unfortunately this improvement has little effect on the access time for his site, because there aren't very many names in the file anyway. The time required by FIND is now 5 milliseconds instead of 10 milliseconds - not a noticeable difference.

Now that the file is sorted, it is also easy to write a procedure to **automatically** detect **duplicate** names:

```
public void showDuplicates()
{
   String name = "";
   String email = "";
   String password = "";

   BufferedReader customers =
        new BufferedReader(new FileReader("customers"));

   String previous = "";

   while (customers.ready())
   {
      name = customers.readLine();
      email = customers.readLine();
      password = customers.readLine();

      if (name == previous)
      {  output(name); }

   }
   customers.close();
}
```

The customer database continues to grow exponentially - doubling every week. After it gets to 2000 names, Zonker is finding it time-consuming to put the new names into the file in the right place - he keeps searching up and down to find the right place. He has a clever idea how to speed things up. He puts a notebook next to the keyboard, searches once through the file, and writes down the approximate position where the B's start, where C starts, etc, and ends up with a list like this:

**PARTIAL INDEX**

| *comment* | **Letter** | **Position (pages)** |
|---|---|---|
| The numbers tell how many times Zonker pressed the Page-Down key to get to the letter. | A | 1 |
| | B | 5 |
| | C | 7 |
| | D | 12 |
| | ... | ... |
| | T | 57 |
| | ... | ... |
| | Z | 79 |

**DATA FILE**

```
Allen Allstar
AAllstar@hotmail.com
AAAAAA
....
Bob Baker
BB@hotmail.com
BeeBee
....
Carla Charles
TooCool@yahoo.com
sekret
....
....
....

Tommy Lee Tunes
TomTune27@aol.com
a1b2c3xxx
....

Zztop
....
```

This seemed like a good idea at the time, and did speed up the data-entry, but every couple days Zonker must scribble out the numbers in the **index** and write them new. He realizes he can write a simple procedure to automatically print out the index. His procedure will count the number of **records** until each letter comes. He can convert this to an approximate number of "pages" by dividing by 10, as there are 10 records on the screen at one time.

```
public void showDuplicates()
{
   String name = "";
   String email = "";
   String password = "";

   BufferedReader customers =
        new BufferedReader(new FileReader("customers"));

   char previous = ' ';
   int count = 0;
   while (customers.ready())
   {
      count = count + 1;

      name = customers.readLine();
      email = customers.readLine();
      password = customers.readLine();

      char firstLetter = name.charAt(0);
      if (firstLetter != previous)
      {  output(firstLetter + ":" + (count / 10) ); }
      previous = firstLetter;
   }
   customers.close();
}
```

## Stage 4 : Merge (sorted + sorted ==> sorted)

It doesn't take very many weeks until Zonker starts **hating Notepad!** In fact, just after 2500 records, Notepad **won't open the file** any more - it is **over 64 KB**, and Notepad can't open such big files. Windows suggests using **WordPad** to open the file. Zonker doesn't know what WordPad is, but he opens the file, types in some names, and saves the result. The next day, he gets a few phone calls from users saying they **can't log-in**. Oh, wow, they are really mad (especially his girl-friend), because Zonker had promised a super-duper-extra-surprise tidbit about a big-shot movie star. Zonker tries to log-in and finds out he can't log-in either. He opens the data-file in WordPad and it looks fine. But when he runs the FIND program in **debug** mode, he sees the program is getting a lot of **garbage** from the file instead of names and passwords. After a 2 hour wait on the My-Cross-Oft help line, Zonker figures out that WordPad has saved his data-file in **MS-Word .DOC format,** so the text-file **isn't a text-file** any more. But don't worry about the 2500 names - he has a backup. The backup is only a week old, so he only loses 1000 names (ouch!). After a week pounding on the keyboard he's back to where he started, and decides to do the right thing and write a **proper data-processing program** to take care of his data.

Can't the computer automatically put the new records in the right place in the list, without Zonker messing around in the file? Unfortunately, there is **no way** for a program to **insert** new data in the middle of a **text-file.** But it could do the following (pseudo-code description):

```
Input a new name, e-mail, and password

Search for the name which should be just before the new name.

During the search, write all the data into a new file

Write the new data record into the new file

Continue reading the old file and copy the rest of the data to the new file

Erase (kill) the old file

Rename the new file to have the same name as the old file
```

This is called "**merging**" the new data into the file. This is pretty straightforward. But Zonker decides to take this one step further. He can make a **sorted list** of new data. It should be possible to **merge** this new file into the old file in **one single pass**, like **shuffling** two piles of cards together. Here is the idea:

```
At each point in the algorithm:

   the first elements in the two lists are compared

   the smaller of the two is removed from it's list
        and written into the new file

   this continues until one list is empty

   then the rest of the other list is copied into the new file
```

Zonker writes the **merge** procedure and everything is better, at least for a few weeks.

---

## Stage 5a : Partially Indexed (text-file - oops?)

As the number of users continues to double each week, Zonker is starting to get worried about the size of the data-file - not huge problem (yet!) The access time is **proportional** to the size of the file - **O(n)**. So if the access time for 100 records is 5 ms, then 1000 records is 50 ms, 10 000 is 500 ms, etc. At 20 000 records, the access time is 1 sec. But this means at 200 000 it will be 10 seconds, and that starts to be worrisome. He's not so much worried about the customers who are logging in - but the slow look-ups might keep his server so busy that it doesn't keep up with the customer's other (Web) requests.

Zonker remembers using his **manual** version of an index, writing page numbers for each letter. He decides this might also speed up his program, so he creates an **indexed-sequential access** routine. It works like this (pseudo-code):

```
Get the first letter of the TARGET name

Find this letter in the INDEX

Read the corresponding position (record number)

Go directly to that position in the file

Start reading sequentially until the name is found,
   or until the next letter of the alphabet is reached
```

That sounds good - should be **26 times faster** right off the bat, as the sequential search must only look through 1 section of the file (a single starting letter). If the file gets even bigger, just make the index bigger, by recording the positions of Aa, Ab, Ac, Ad, ... This cuts the search area by a **factor of 26** again.

Zonker starts writing the program, then realizes that it is **not possible** to **jump** to a specific **position** in a **text-file**. Text-files are strictly **sequential access** - you must read the 1st record, then the 2nd, etc in order.

..... So .....

## Stage 5b : Converting to Direct-Access (random access – that's better)

To implement a partially indexed file (indexed-sequential access), Zonker needs **two data-structures**:

1. **Index** - contains letters (A,B,C...) and the **starting position** of each letter section
   This can be a text-file, as it is small and can be searched sequentially.
   Or it could be stored in arrays.

2. **Data -** the actual customer data must be stored in a **direct-access** (**random access)** file, so it is possible to jump to any location in the file, e.g. the **S** section when searching for Smith.

The **search algorithm** is implemented as stated above in 5a.

Now it's really **the end** of using Notepad and Wordpad. Direct-access can only function with **fixed-length records**. Every **field** has a fixed length. For example:

```
class customer
{
   String name = "";          // 30 characters
   String email = "";         // 40 characters
   String password = "";      // 20 characters
}
```

The **reading** and **writing** methods must be careful to ensure that every record is **exactly the same size** as every other record. If UTF Strings are used, they need 2 extra bytes each, so the total size of the record would be 30 + 2 + 40 + 2 + 20 + 2 = 96 bytes. Zonker was never very good at math, so he decides to "round this off" to 100 bytes. That means each record will occupy 100 bytes, and the records will start at positions like 0 , 100 , 200 , 300 , ...

Each NAME must be **exactly** 30 characters long (+2), so short names will be **padded with blanks** to fill up the field. This would be a big pain to do by hand, typing all those extra spaces. Even worse, counting all those extra spaces is unreliable. If you get it wrong, the data ends up looking like this:

```
Albert Einstein             AE                  //The NAME field was too short
instein@Hotmailisdeadfinally.co.uk              // so the email address is missing
   passwordwithspaces                           // the first two letters AE
```

The NAME field above is too small, so the first two letters of the EMAIL field get included as part of the name, so both the EMAIL address and the name are actually wrong. **Manual editing** of fixed-length records is definitely a **bad idea!**

Why are fixed-length records required anyway? The only way to "jump" into the middle of a file is to **calculate** a position, in bytes, then **seek** to that position. To find record #500, calculate 100*500 = 50000, then seek to byte #50000. This doesn't work in normal text files, where each line of data could be any length, so the starting positions of records is unpredictable.

Here is a **method** to **force** a string to have the correct length. It may not be super efficient, but that could be improved later if necessary.

```
public String forceLength(String data,int size)
{
   while (data.length() < size) { data = data + " "; }
   if (data.length() > size) { data = data.substring(0,size); }
   return data;
}
```

Zonker doesn't want to fix the original text file by hand, so he writes a **conversion utility** to read the text file and create a new, random-access data file:

```
public void convertToRandom()
{
   BufferedReader customers =
     new BufferedReader(new FileReader("customers"));

   RandomAccessFile ranCustomers = new RandomAccessFile("ranCustomers");

   Customer cust = new Customer();       // see Customer class above

   int record = 0;

   while(customers.ready())
   {
      cust.name = customers.readLine();
      cust.email = customers.readLine();
      cust.password = customers.readLine();

      ranCustomers.seek(record*100);

      ranCustomers.writeUTF(forceLength(cust.name,30));
      ranCustomers.writeUTF(forceLength(cust.email,40));
      ranCustomers.writeUTF(forceLength(cust.password,20));

      record = record + 1;
   }

   customers.close();
   ranCustomers.close();
}
```

Now Zonker needs to create the **INDEX**, with **POINTERS** into the **random-access** data file. This is pretty similar to his original PRINTINDEX algorithm, but this time he wants the results stored in an array. He plans to run this procedure once each day, then keep the arrays in **main memory** for fast access. The INDEX and POINTER arrays need to be **global**, so they can be used by other procedures later.

```
char[] index = new char[100];
long[] pointers = new long[100];

public void makeIndex()
{
   char targetLetter;
   int pos = 0;

   for (targetLetter = 'A'; targetLetter <= 'Z'; targetLetter++)
   {
      RandomAccessFile ranFile = new RandomAccessFile("ranCustomers");
      long maxRecord = ranFile.length() / 100;
      int rec = -1;
      char firstLetter = ' ';
      while (firstLetter < targetLetter && rec < maxRecord)
      {
         rec = rec + 1;
         ranFile.seek(100*rec);
         String name = ranFile.readUTF();
         firstLetter = name.charAt(0);
      }
      ranFile.close();

      index[pos] = targetLetter;
      pointers[pos] = rec;
      pos = pos + 1;
   }
}
```

Zonker is an amazing programmer – 8 hours and 3 pizzas later, he's finished! **.... and Finally .....**

## Stage 5c : Partially Indexed Access (yeah!!)

Zonker takes a break to think about how to write the **access** routine – the one which searches for a NAME using the algorithm from 5a (above). This sounds tricky. While he is lying there under the tanning lamp thinking about this problem, he realizes he still needs to keep the records **sorted**, or at least **grouped** in their first-letter groups. He will need a procedure to **insert** new records into the middle of the file. Rather than rewriting his **merge** procedure, he decides to contract out this job to an eager IB Computer Science student, who works cheap – wait, no, even better, he can get the IB student to do it as part of their Internal Assessment project, for free! Solved that problem!

Now he can get on to the important part – making the log-in procedure to look up names and passwords. With the random-access-data-file and index structures created, Zonker is ready to finish the access routine. The first part is easy – finding the first letter of the name in the INDEX array:

```
public long getPointer(String target)
{
   for (int p = 0; p < 26; p = p + 1)
   {
      if (index[p] == target.charAt(0))
      {
         return p;
      }
   }
   return -1;
}
```

Now the program needs to **jump** to the position indicated by **getPointer**. In Java the command is:

```
ranFile.seek(pointers(getPointer(targetName)));
```

Here is the rest of the search algorithm. It is pretty similar to the original **findPassword** method above, but it uses **getPointer** to decide where to start searching:

```
public String getPassword(String targetName)
{
   String password = "";
   long pos = getPosition(targetName);

   Customer cust = new Customer();
   RandomAccessFile ranFile = new RandomAccessFile("ranCustomers");

   while (pos < ranFile.length() )
   {
      ranFile.seek(pos);
      cust.name = ranFile.readUTF();
      cust.email = ranFile.readUTF();
      cust.password = ranFile.readUTF();

      if ( targetName.equals(cust.name) )
      {
         password = cust.password;
      }
      pos = pos + 100;
   }

   ranFile.close();
   return password;
}
```

The log-in routine now looks like this:

```
public void login()
{
   String username = "";
   String password = "";
   int count = 0;
   boolean okay = false;

   while (count < 3 && !okay)
   {
      username = input("Type your user name");
      password = input("Type your password");
      if (password.equals( getPassword(username) ) )
      {  startWebSite(); }
      else
      {  count = count+1; }
      if (count >= 3)
      {
         recordHackingAttempt();
         shutdown();
      }
   }
}
```

## Stage 6a : Binary Search (good and bad)

As the size of the **ranCustomers** file increases further (now a half million records, thanks to a mention in a CNN news story), the search times are getting longer and longer, even with the partially-indexed retrieval. Zonker bought a faster server, but that only made things twice as fast (1 second wait instead of 2). He thinks about rewriting the **index** routines to make 2-letter indexes Aa, Ab,Ac, etc. This should cut the time by a factor of 26. But that sounds messy, and he heard about a **binary search** method which is supposed to be super fast – **O( log n ).** He downloaded a binary search routine from the web and rewrote it to work with his data. The routine is **recursive** – it calls itself. It assumes the file has already been opened. The basic idea (in pseudocode) is as follows:

```
At the beginning, start = 0 and end = maximum record in the file
Calculate the middle of the file
Get the name from the middle position
If the target is the same as the middle pos name
    then quit and return the position (that means found)
Otherwise
   if target comes after the middle, then
      set start = middle + 1
      leave end unchanged
      go back and calculate the middle again and continue
   if target comes before the middle, then
      set end = middle - 1
      leave start unchanged
      go back and calculate the middle again and continue
If end and middle come together (equal),
   then it is hopeless to continue, so report "not found" (pos = -1)
```

The binary search only works as long as the data file is sorted. That's fine if he is searching for a NAME. But sometimes the users forget their NAME because they are actually using a phony name (Keyboard King or something like that). When this happens, the user can log-in using their EMAIL address and password. Or they can request the NAME and PASSWORD be sent to their EMAIL address. Either way, they need to type their EMAIL address and the computer must search for the EMAIL address in the data file.

The EMAIL search **must** be **sequential**, because the EMAIL addresses are **not sorted** – only the NAMES are sorted. Zonker cannot use a binary search for EMAIL addresses. This is very slow. Fortunately, it doesn't happen very often.

To solve the EMAIL search problem, ZONKER decides to keep **two copies** of the data-file – one is sorted by NAME, the other sorted by EMAIL. Then both NAME and EMAIL searches can use the binary search method. But this means that two copies of the data-file must be maintained, and with a half-million new users each week, this is a major problem. The only rational approach seems to be keeping one file, and then **resorting** it each day to produce the second copy.

## Stage 6c : Re-Sorting (ouch!)

How fast is the sorting process? Zonker wrote a **bubble-sort** routine and tested it on 1000 records. It took 0.1 seconds to sort the NAMES, and another 0.1 seconds to resort it in EMAIL order. Not bad, he thinks. Unfortunately, the efficiency of a bubble sort is **O(n^2).** This means that doubling the length of the list will multiply the time by 4, with the following disastrous behaviour:

| Size of list n | Sorting time in seconds |
|---|---|
| 1000 | 0.1 |
| 2000 | 0.4 |
| 4000 | 1.6 |
| 8000 | 6.4 |
| 16000 | 25.6 |
| 32000 | 102.4 |
| 64000 | 409.6 |
| 128000 | 1638.4 |
| 256000 | 6553.6 |
| 512000 | **7.2 hours !!!** |

It doesn't look good for using a bubble sort to sort the data-file - it will take almost an entire working day!! At first Zonker thought he would only need to run the sorting routine once a day, in the morning, for a couple minutes. But it looks like that won't work.

A **quick sort** is faster - efficiency **O(n log n),** where log means log base 2. So it behaves like this:

| Size of list n | n log n | Sorting time (theoretical) n log n / 1024 * 0.1 sec |
|---|---|---|
| 2^10 | 1024*10 = 10240 | 0.1 |
| 2^11 | 2048*11 = 22528 | 0.22 |
| 2^12 | 49152 | 0.48 |
| 2^13 | 106496 | 1.04 |
| 2^14 | 491520 | 2.24 |
| 2^15 | 1048576 | 4.8 |
| 2^16 | 2228224 | 10.24 |
| 2^17 | 4718592 | 21.76 |
| 2^18 | 4718592 | 46.08 |
| 2^19 = 512000 | 9961472 | 97.28 sec (1.6 min) |

This is more like Zonker wanted – he can resort the file each day, in just 1.6 minutes. But try as he may, he doesn't get the quick-sort to work. Just then, Zonker gets an offer of 100,000 Euros to sell his business. He decides to sell out, take the money, and go to college. Now someone else can worry about taking care of this huge data-file. He wasn't really that interested in programming anyway.

## Stage 7 : Fully-Indexed (multiple indexes, ISAM)

The new owners of Zonker's web-site and data-base is a company named MarketSoft. They are more ambitious than Zonker, and they have lots of competent programmers working for them.

MarketSoft makes money by collecting e-mail lists and personal data, then selling these lists to companies who send **spam** to millions of e-mail accounts every day. The spam operation only works effectively if the advertising is **targeted** – e-mails are not sent out at random, but sent to addresses where the user is likely to be interested in the product. The first job for MarketSoft is to collect a lot of personal data from each of Zonker's customers.

MarketSoft sends e-mails to all of Zonker's subscribers to tell them the company has been sold. Customers will be permitted to keep their memberships (for free) if they fill out a long questionnaire with lots of personal information – age, hobbies, profession, etc. Otherwise, they will need to pay a higher membership fee. The customers read the e-mail, and most figure they might as well send in their personal data if they can keep their accounts open at no extra cost. So MarketSoft gets lots and lots of new data, including the country where each user lives - the new data must be recorded in a new data-file with a much larger record structure.

Now there are many different kinds of searches that need to be performed, but the file can only be kept sorted according to one field (normally the name). If all the searches are going to be sequential, things could get messy and slow as the data-base continues to grow.

The solution is to maintain **multiple indexes.** There is one single data-file, plus a separate index file for each field which needs to be searched. This only works if each index is a **full-index** – it contains an entry for each record in the file, even if their are duplicate entries. It might look something like this:Data

**File**

| Position | Data (Name/Email/Country) ** |
|---|---|
| 1 | Al Anders / aa@fis.edu / DE |
| 2 | Barbie Brooks/king@jj.com/US |
| 3 | Carl Cook / dude@ab.co.uk/ UK |
| 4 | Debbie Duke/ duke@ibm.com/ US |
| 5 | Ed Elliott / wow@blah.com / JA |
| 6 | Fern Fool / fool@ok.com / IR |
| 7 | Greg Gun / gg@iq.de / DE |
| 8 | Heidi Hahn / cow@news.com / CH |

**Email Index**

| Data | Pos |
|---|---|
| aa@fis.edu | 1 |
| cow@news.com | 8 |
| dude@ab.co.uk | 3 |
| duke@ibm.com | 4 |
| fool@ok.com | 6 |
| gg@iq.de | 7 |
| king@jj.com | 2 |
| wow@blah.com | 5 |

**Country Index**

| Data | Pos |
|---|---|
| CH | 8 |
| DE | 1 |
| DE | 7 |
| IR | 6 |
| JA | 5 |
| UK | 3 |
| US | 2 |
| US | 4 |

** this is still a random access data-file, **
** but the data records have been abbreviated **
** with /slashes/ for simplicity **

Since the data file is normally quite large, with very large records (several kilobytes each) the "overhead" of having many extra index files is not a significant disadvantage.

It is not actually necessary to keep the file sorted by NAME – instead, a NAME INDEX file can be created, and the INDEX file is kept sorted. It is **still possible** to use a **binary search**, because the search will occur in the INDEX file, not in the data file. Once the NAME is found, the rest of the data can be retrieved by following the pointer to the correct position in the data-file. Further, a binary search can be performed on **any** index file.

Now maintaining the data-file is much simpler, because new records are simply added at the end of the file. Well, it isn't quite that simple. When a new record is added, a new entry must also be added to **every index file**, and these entries must be **inserted.** So it isn't totally simple, but it's basically a very efficient data-structure concept.

The basic concept for **accessing (**searching for something) is:

```
Get the TARGET string
Choose the appropriate INDEX
Perform a binary search in the INDEX to find the TARGET,
   returning the POSITION of the desired record
If found, open the DATA-FILE, jump to record POSITION,
   and retrieve the desired record
```

The very popular **ISAM** technology (Indexed-Sequential-Access-Method) is based on this concept. It is a bit more complicated, and was developed thoroughly and marketed by IBM many in the 1980s. At a serious company like MarketSoft, the programmers probably start with a basic, complete, general **DBMS (Data Base Management System)** which implements ISAM automatically – examples are MS Access, Oracle, and various SQL systems (including MySQL). They start with a complete system, and modify it to do exactly what they need. They **don't** actually sit around writing code for ISAM operations – this has already been done enough times, and the code has been **optimized** during many years of use and revision.

---

So what happened to Zonker? He went off to college, but couldn't give up programming – he was addicted. He turned into a real hacker – the good kind, who stays awake all night trying to write very clever programs, not bothering anyone and certainly not breaking into other people's servers. He wasted too much time hacking and missed too many classes, and eventually dropped out of college. He got a job at MarketSoft where he got paid for hacking (programming, that is). His best hack was a new database access method. Poor old Zonker never could spell very well. He wanted to name his method "Hacking", but he typed that into a word-processor and the spell-check changed it to "Hashing", and now we are stuck with that name. So, what's **hashing?** Just when you thought you were finished with these notes, there's still **ONE MORE PAGE!!!**

# ####### Hashing #######

***- Hash Code -***

Use a **key-field** in the data to **calculate** a **hash-code -** the **position** in the array where the data will be stored.

***- MOD Wrap around -***

The hash-code is often a very large number, so this would require an enormous list. By calculating **Hashcode MOD ListSize**, a position can be chosen in a smaller list. For this to work well, the **ListSize** should be a **prime** number - e.g. 101 instead of 100.

***- Collision -***

Several different records might calculate the same hash-code. But they cannot all sit in the same place in the list. This is commonly resolved by searching for the **next free position.** Collisions are reduced by using a **sparse array** - many empty spaces, e.g. 50%.

***- Unique Keys -***

**Collisions** can be reduced by using a **unique key field**, where all key-field values only occur once - for one data item. Unique keys can be generated by combining two fields - for example **Name + Birthdate.** However, there will still be collisions if **mod** is used.

***- Access -*** To retrieve a value:

- Calculate the hash-code
- Look in that position - if it's the right data, finished
- If it is **not** the right data, then do a **linear search** starting at the next position, until the item is found.

***= Speed = O(constant)***

**Hashing** is used in **large** data files to speed up access. If you are lucky, the data is retrieved directly from the hash-code calculation, in **1 step**. More common is one hash calculation followed by a few steps in a linear search. This is something like O(5) (5 search steps) – so the search time is **constant**, and **does not increase** as the list grows.

***Data Item***

**"Smith, Joe" , 17 , "Germany" , "15.08.1982"**

Key Field = First 4 letters of Name = "Smit"
→ use 4 ASCII bytes as 32-bit integer
= [83][109][105][116]

***Hash Calculation***

**"S" "m" "i" "t"**
**83 * 256^3 + 109*256^2 + 105*256 + 116**
**→ 1399679348 = Hash Code**
**1399679348 mod 101 → 37 = Position**

***Position***
"Smith, Joe" lands in position [37]. If another "Smith" gets recorded, it goes into position [38]

---

Using this algorithm, and a list of length 101, (mod 101) we get few collisions in 20 names:

```
Adam   = 1097097581 --> 29
Bobo   = 1114595951 --> 48
Carl   = 1130459756 --> 86
Dave   = 1147237989 --> 98
Eddy   = 1164207225 --> 21
Frank  = 1181901166 --> 75
Gina   = 1198091873 --> 78
Hellen = 1214606444 --> 38
Irene  = 1232233838 --> 3
Jackie = 1247896427 --> 17
Karen  = 1264677477 --> 18
Lara   = 1281454689 --> 19
Mildred= 1298754660 --> 3  ==> 4
Norman = 1315926637 --> 61
Owen   = 1333224814 --> 69
Petra  = 1348826226 --> 11
Quincy = 1366649198 --> 18 ==> 20
Ruth   = 1383429224 --> 5
Susie  = 1400206185 --> 58
Tony   = 1416588921 --> 89
```

**Mildred** collides with **Irene** and gets moved to position 4. **Quincy** collides with **Karen**, but position 19 is already occupied by **Lara**, so **Quincy** lands in position **20.**

== **ADT** and **FILES** for **MASTERY MARKS** ==

Zonker's fable describes several different **data file storage structures.** A **RandomAccessFile** containing records is only one of these possibilities. But it is a good way to for an IB student to score **mastery marks.**
An HL student must demonstrate mastery of 10 items chosen from 15:

**- 5 SL techniques (1 mark)**
**- Direct Access Files (direct access) : inserting, deleting, searching (3 marks)**
**- Linked-lists / trees : inserting, deleting, searching, handling errors (4 marks)**
**- Object-Oriented Programming : Polymorphism, Encapsulation, Inheritance (3 marks)**
**- Others (difficult) : merging two lists, recursion, parse a text file, hierarchical data (4 marks)**

A typical HL dossier contains a RandomAccessFile (3 marks) and an ADT (perhaps a Linked-List for 3-4 marks), use 5 SL techniques (1 mark) and score 3 marks in OO programming. This is just enough.

Without RandomAccessFiles and ADTs, it is impossible to score all 10 marks. Each missing mark reduces the final grade by 10%. So it is **RECOMMENDED** that the dossier implements a RandomAccessFile and an ADT.

Construction of an **indexed-data-file** or **hashed file** have the potential of demonstrating many of these skills in a single package. This is simpler for both the student and the examiner. For example, the **index** can be constructed and stored in memory as a linked-list or tree, while the data is **permanently stored** in a random-access file.

**Careful!** The syllabus requires that records are **directly** added to the data-file, and that means **inserting** them. So although an **index** should be the primary access method, it would not be sufficient to **only** add records at the end of the file. Program an **insert** function which inserts a record into a specific position in the file, **in addition** to the normal adding at the end. Program a **sorting** routine which reorganizes the file into a specific sorted order – say by name – this "sort" of thing is called a **primary key** sort.

It may seem silly or wasteful to have **two different** access methods on the same data-file, until you think of it in slightly different terms. Normally software has an **end-user(s)**, but **also** there are **support technicians** for maintenance. The support staff probably uses **low-level** tools to fix technical problems in the software – e.g. physically re-ordering, adding, or removing records by direct access. The end-user has different needs, and wants to use **high-level** tools to accomplish their work – e.g. printing lists of various subsets of the data in various orders. The end user does **not** want to worry about what is going on inside the program.

This all makes sense if there are **two sets** of interfaces/commands/tools available:

**End-user tools AND Maintenance tools**

Although the **end-**user is your main target, there are usually also **technical support** people involved. The redundancy (duplication) of having two different access methods in the two different tool sets makes your software more **robust** (reliable).

**Merging** is one of the mastery items. Don't ignore this one – it is fairly easy to program and scores 1 point just like the more difficult deleting functions. Merging means:
(1) Start with **two** sorted lists
(2) Merge them together one record at a time into a third list
(3) The third list must also be sorted

**Caution!** Whatever you put into a class module, you will only receive mastery credit if that function is actually **used** for a real purpose in the finished program. Thus, it is not sufficient to simply create a merge function – it must also be "used for some non-trivial purpose" in the program. It must be used correctly and efficiently, but this does **not** require that it is the most efficient method possible. Thus, you might be merging just a few records into a big file, and this could be done more efficiently by inserting, but that's okay.

**Before** starting your program, **design data-structures, algorithms** and **classes** to ensure mastery factors work!

## Data Structures (B1)

A **data-structure** is a **collection** of data, not a single **primitive type**. So a single number is **not** a data-structure - it is simply data. A **String** can be a data-structure if it contains several pieces of data, for example: "Name/email/phone". An **array** is a data-structure - it contains a **list** of data. **Text files** and **RandomAccessFiles** are data structures. Other examples are **stacks**, **queues**, and **linked-lists**. If a data-structure contains other data-structures, it is called **hierarchical composite** data-structure - for example, an array of records or a record containing an array.

Data-structures consist of **data** (the data items) and **structure** (organization). Before starting a program, you should clearly specify the needed data-structures. This requires some thought and decisions.

For the IB IA dossier (project), you are required to **describe** and **illustrate** your data-structures. This must clearly explain the **organization** as well as the **type(s) of data.** The descriptions should use **standard Computer Science vocabulary**. Thus, you should say "an array of Strings" rather than "a list of names". The **illustration** requires diagrams - an array must be shown using boxes, a linked list with nodes and pointers, etc. Both the description and the illustration must contain **sample data**. The sample data should cover a **range** of values - e.g. small numbers and large numbers, long Strings and empty Strings, etc. The sample data should be realistic - e.g. ages between 1 and 100 rather than random numbers.

If there are several similar structures, such as an array of names of basketball players and an array of names of soccer players, it is not necessary to draw pictures of both - you can say something like "the soccer list is just like the basketball list, but with the names of different students."

The maximum 4 marks are awarded if:

| |
|---|
| The student has discussed and **clearly illustrated all** of the data structures/types to be used to solve the problem, and provided **sample data** for **all** of them. |

## Algorithms (B2)

Algorithms refer to **methods** that will accomplish specific tasks. Typical examples include:

- **Search** for a name in an array
- **Sort** a RandomAccessFile
- **Count** the number of words in a sentence
- **Parse** a String to find key-words
- **Reverse** the order of the nodes in a linked-list

Notice that each example above is **connected** to a **data-structure.** For your project, you must describe all the algorithms that your program will implement. You are not required to describe very simple, common algorithms like inputting and capitalizing a String. For the most part, you can limit your descriptions to the algorithms that operate on data-structures. For standard algorithms, a simple statement of the name of the algorithm is sufficient - e.g. "use a Bubble-Sort to put the array in alphabetical order". For non-standard algorithms, you must provide a description of the process. For example, Zonker needed to convert the text-file to a RandomAccessFile. The Java code above is more detailed than required. This could be written in pseudo-code as follows:

```
Open the text-file
Create a RandomAccessFile
Repeat
   Read name,email,password from the text file
   Fix the lengths of the Strings:
        name = 30 chars, email = 40 chars, password = 20 chars
   Seek the next record position in the RandomAccessFile
   WriteUTF strings for each of name, email, password
until the end of the text-file
```

The maximum 4 marks for algorithms are awarded if:

| |
|---|
| The algorithms discussed are sufficiently **logical**, **detailed**, and **well documented** to be used to create the solution in Java. |

That doesn't mean the algorithms are **totally correct** - you can leave out many small details, especially extensive error-checking and user-interface details. It is really the **important automation** algorithms that must be described. If you are going to implement a fully-indexed data-file, like Zonker, you need to describe the algorithms that save data and retrieve data, as well as the algorithms for creating all the indexes.

As a rule-of-thumb, algorithms must be described if they:
- involve loops
- affect data-structures
- involve complex logic (lots of if.. commands)
- implement automated processes

The simplest way to describe the algorithms is through pseudo-code. There are no "rules" about how to write "correct" pseudo-code. It is basically "sloppy Java" - Java written without worrying about syntax. But that doesn't mean you can invent things that don't exist. So you cannot just say "I will implement the Java command that validates, sorts, and saves the array." That simply doesn't exist.

A warning about Java commands - some of the commands in **java.util.*** are considered "cheating". For example, java.util contains a LinkedList class, but students will not receive ADT mastery marks for using this class. You must **create your own** linked-list ADT from scratch. There is no expectation that you write a "complete" linked-list class as good as the one in **java.util.** The requirement of making a "complete" ADT means that it contains all the algorithms necessary for your project, not all the algorithms that a truly **universal** linked-list would have.

## Modular Organization = Classes (B3)

The term **module** is generic and applies to Object Oriented languages like Java as well as older high-level structured languages like Fortran. But we are using Java, and in Java a module is usually a **class.**

You are expected to organize your project into classes that **encapsulate** data-structures with their corresponding algorithms. If you create an **ADT**, you should definitely do this as a single **class** (which may contain or extend sub-classes). It should have a thoroughly specified **API (**application programming interface) consisting of the **constructors** and other **methods** that can be used by an application programmer.

This is an **organizational task**. It requires the programmer to decide which modules (classes) will be **required** or **permitted** to execute algorithms that modify the contents of data-structures. For example, in Zonker's application the **log-in** program can **read** the password file to check whether a password is correct. But it should **not** be able to **change** a password or **delete** a user. That is the **responsibility** of some other module - probably in the system-maintenance or administrative functions.

A sensible approach for this task is:
1. Make a **list** of data-structures
2. Make a **list** of algorithms for each data-structure
3. **Collect** data-structures with algorithms into **logical** classes, using sensible **encapsulation**
4. Identify **responsibilities** and **access**

## Suggested Process for Stage B

1. Read and think about **user stories** and **goals** from Stage A

2. Identify **NOUNS** and create corresponding **data-items** and organize these into **data-structures**
   Think about **data-types, records** and **files.**
   ***Example:***
   nouns : user name password email
   Data-structures: **user-record** = name/email/password (3 Strings)
   **user-file** = file contains **user-records**

3. Make a fairly **complete list** of all the data and **data-structures** for your program
   Check that **every noun** (data item) has a place in this list.

4. Again read and think about **user stories** and **goals** from Stage A

5. Identify **VERBS** and create **names** for **methods** and **algorithms** required for your program.
   Think about **standard algorithms** like **sorting** and **searching.**
   When you record an algorithm, always **identify the data-structure** that will be involved.
   At this stage the description of the algorithm is not very detailed - just write your initial ideas.
   ***Example:***
   verbs: log-in look-up validate deny-access
   Algorithms: **login**
   input user name and password
   check whether these match those in the **customer file**
   then either start the program or reject the log-in attempt

6. Distribute **algorithms** and **data-structures** into **modules (boxes) ,** putting things together that belong together.
   Make up **names** for the boxes - these are **class** names.

7. Look over your modules, and try to **extract** ideas for **classes** that will work effectively -
   here you are trying to **minimize work** by **maximizing reuse** and improve **reliability** by enforcing **encapsulation** to **improve cohesion** and prevent **side-effects** due to **inappropriate access.**

8. **Organize** your **classes**, **data-structures** and **algorithms** in a **hierarchical outline** (example below).**Discuss** this **first draft** with the teacher, then make changes to the **outline** as the teacher suggested.

9. Return to your list of **data-structures** and fill in the details:
   - **Draw a picture** of each data-structure including **one example of sample data** for **every data item.**
   - For **lists** (files and arrays), provide at least **4 pieces of sample data**
   - Where possible, write **rules** or **ranges** for **valid data items**
   - **Describe** the **organizational structure** of lists and files. Use **standard vocabulary** when possible - for example, "sequential file" or "random-access-file"
   - Outline **structural integrity rules** such as "file is always sorted" or "array must end with 'xxx' "

10. Return to your list of **algorithms**, and fill in the **details** in the algorithm **pseudocode**, e.g:

```
login
    input user name
    input password
    search in customers file for the user name
        if not found, quit
    if  password == filePassword
       start program
     else
       try again - after 3 tries, reject the user
```

11. Create a **diagram(s)** showing **module interactions** - that is, classes that access each other. This is a diagram similar to what **BlueJ** produces for a project. This represents the same thing as **collaboration** in CRC cards. If possible, this should not be a tangled mess of arrows. If there are too many arrows (making a messy picture) you may need to reorganize your classes into a more **hierarchical** system.

    You may prefer to stick to an **outline** showing the hierarchical structure, but this only makes sense if there are very few connections (e.g. arrows) between classes that are separated in the hierarchy. Your outline from #8 is probably adequate, but leave out the details of the algorithms - just provide meaningful names. The outline is easier, but the picture with arrows looks nice. DON'T draw arrows in an outline. If you need to connect classes, just use the names.

12. Once you have finished the **classes outline, data-structure diagrams**, and **algorithm pseudo-code,** print them and give the to the teacher to be graded. **Due Date: Monday 28 Nov**

**Sample Docs Stage B – First Draft**

**== Data Structures ========**

**-- Document Server --**

- **Documents** = a set of documents (.html, .rtf) stored in a folder on a server
  It may be useful to have a LIST of the NAMES of these documents stored in a file somewhere (?)

- **Vocabulary List Files** = for each reading document, a matching vocab list containing Vocab records
  - Vocab Record = word, definition

- **Master Dictionary File** = complete merged list of all the vocabulary files, to be used for backup reference if the user clicks on a word that is NOT in the current vocab list

**-- Student Data Files --**

- **Accounts File** = file of Account records for each student
  - Account Record = student-name, password

- **Assignments File** = one big file containing Assignment records for all students
  - Assignment Record = student-name, document-name, accessed flag (on after document accessed)

- **Reading Sessions File** = one big file containing ReadingSession records for all students
  - ReadingSession Record = doc-name , student-name, date, starting-time , ending-time, list of vocabulary clicks

**-- Internal Data-Structures --**

- **Words Clicked List** = an array containing the words that were clicked by the student - could be an array if they click the same word twice, it is recorded again

- **Current Vocab** = a Vocab List (ADT), to be loaded from the file when a document loads.
  When a student clicks a word, it is looked-up here. This should be sorted, so an efficient look-up can be performed.

- **Dictionary** = a Vocab List (ADT), loaded from the Dictionary file.
  If student clicks a word that is not found in the Current Vocab Array, then the program will search in the Dictionary array before going to a web-site.

**-- ADT's --**

- **Vocab List** = a list of words and definitions, providing fast look-ups (e.g. binary search or hashing