VII. DATA STRUCTURES: ORGANIZATION FOR NON-SEQUENTIAL ACCESS

A. DIRECTORIES
B. INVERTED LISTS
C. HASHING CONCEPTS
D. RELATIVE FILES
E. INDEXED FILES

The purpose of these types of data structures is to construct one or more additional data structures or to reformulate the primary list in a particular way so as to provide a quicker way to find a record.

All of this will result in a form or forms of data structures that – in one way or another – will "compartmentalize" the data. This "compartmentalization" will be based on one of several forms of data structures. Some forms result in auxiliary data structures; others will re-structure the primary list in a manner that, with specialized computations, provide improved access speed in the sense that fewer operations, e.g., looks/compares, are required.

The data will be logically or physically grouped into "compartments" or "neighborhoods". Thus, our initial steps to find a record that is directed toward identifying the neighborhood (or group) where the record is stored. Then, only that neighborhood needs to be linearly searched.

A. DIRECTORIES

1. *THE STRUCTURE OF A DIRECTORY*.

To begin, let's consider that our records in an ordered file are extremely large. Consequently, a sequential search of the primary list would take a relatively long time to read each record. (Make no mistake: There are such huge records out there.)

If, however, we had an additional structure – a list derived from the primary list – each entry of which contains only the record key value and the associated address of the complete record, then the search to find a match with a record key would be faster. We would need only to search the "directory" for a key match to find the address of the complete, but very large, record.

The first page of your handout (Overhead #1) is an example of this. There is one entry in the directory for each record in the primary list.

Let's now consider an ordered list of more normal sized records as shown on the next page (Overhead #2). You will note that the list has been divided into four, equal (more or less) sub-lists. We first wish to determine which sub-list we should look in when seeking a particular record (based on a key).

Suppose our search key is ONIONS. Let's compare it with the highest key of each sub-list.

IS ONIONS $\leq$ BROCCOLI? No!
IS ONIONS $\leq$ LETTUCE? No!

IS ONIONS ≤ PEAS? YES!!

Now we know that the record we are seeking is located in sub-list 3. Where do we start our linear search in the sub-list? We begin at the beginning, of course, at the host-vector address of 9. Looking at the next page (Overhead #3), you see that we can construct a DIRECTORY in which there is an entry for each sub-list, and each entry is composed of the highest key of the sub-list and the beginning address of that sub-list.

Is this process faster? Does it take fewer "looks" (compares)? Let us examine the process.

For a normal, sequential search of a primary list,

AVERAGE NO. OF LOOKS = (n + 1)/2 where n is the number of items.

In our example, n = 17. Hence, Ave # Looks = (17 + 1)/2 = 9

Now let's use the directory approach. First, what is the process?

1 – Perform a linear search of the directory to identify the appropriate sub-list.
2 – Perform a linear search of the selected sub-list

1. Ave # Looks to search the directory = (4+1)/2 = 2.5
2. Ave # Looks to search the sub-list = (4+1)/2 = 2.5

TOTAL NUMBER OF LOOKS: 5!!

(NOTE: We are talking about lists, so we can mean linked lists as well as dense lists.)

Now look at the next page (Overhead #4). We have an ordered, linked list as the primary list. Notice that we have two levels of directories: A directory to the next set of directories, with these last referencing the primary list. (NOTE: The * denotes the end of the sub-list while the numbers in parentheses are for the next logical item in the complete primary list.

Let's do an example. Let's say our search key is ERNIE.
Searching the outer directory,

IS CONNIE ≥ ERNIE? NO!
IS JOHN ≥ ERNIE? YES!!

So now we examine the indicated directory of the next group.

IS DIANE ≥ ERNIE? NO!
IS JAN ≥ ERNIE? YES!!

Now we are ready to search the sub-list starting at address 27.

Calculating the total Ave # of looks: 1st Directory: (3+1)/2 = 2
2nd Directory: (3+1)/2 = 2
Sub-List: (3+1)/2 = 2
TOTAL: 6

A linear search of the complete primary list yields
Ave # Looks = (27+1)/2 = 14.

2. *OPTIMAL SIZE OF THE SUB-LISTS AND DIRECTORIES.*

How do we pick the size of the sub-list? (This dictates the number of entries in the directory, since there must be an entry for each sub-list.) If we were to write the equation for the average number of looks based on the size of each sub-list and a primary list with n items, and then find the value for which the number of looks is a minimum (differentiating the equation with respect to the size if the sub-list), it can be shown that the optimum size of the sub list is

$\sqrt{N}$ (square root)

For two levels of directories

$\sqrt[3]{N}$ (cube root)

For three levels of directories

$\sqrt[4]{N}$ or $\sqrt{\sqrt{N}}$ (fourth root)

Or, in general,

$\sqrt[(d+1)]{N}$ where d= number of levels of directories.

What if the number of entries in the list doesn't yield whole number for the size of the sub-list? The example on the second page of your handout had 17 elements in the primary list, so one of the sub-lists had to contain 5 elements.

If we had 24 elements, four sub-lists would have 5 entries, and one sub-list would have 4 entries. We simple use judgement to keep the entries at a level that will minimize the number of searches. But we don't need to go any further on this issue.

3. *FORMAL DEFINITION OF THE STRUCTURE OF A DIRECTORY*

a. A directory is itself a list (usually ordered), a separate data structure, that has the following relationships:

- each element in the directory is composed of, at a minimum, a key field and a location field (i.e., address, pointer, link)

- The key field contains a key value from one of the records in the associated data structure. (This would be the primary list for a single level directory or for the directory closest to the primary list. For multiple level directories, it would be a key addressing the next directory moving closer to the primary list.)
- The location field (the address field) will contain one of the following three:
  - (i) The location of the record in the associated data structure with the same key value as that in the key field of the directory (i.e., a directory with an entry for each record in the primary list, as shown in our first example).
  - (ii) The location (address) of the beginning of the sub-list of the associated data structure segment (in this case, a sub-list of the primary data list) that contains the record with the same highest key field value in the sub-list. (For a one-level directory or the directory closest to the primary list).
  - (iii) The location of the beginning of the next level directory (closer to the primary list) that contains, in that next level directory, the same, highest key of that next level directory. (For multiple level directories.)

4. *DISADVANTAGES*

Clearly, it is faster to find a record using one or more directories than to do a linear search. Directories do, however, require additional storage as well as code to process the directory and sub-lists. Also, when adds and deletes take place with the primary list, the directory requires maintenance as well. (Linked list structures would be easier, but, with sufficient adds or sufficient deletes, the sub-list and directory may need to be rebuilt to maintain optimum performance.)

5. *COMPARISON OF SEARCH STRATEGIES* (See next page of handout and overhead.)

a. Linear, sequential search of n items in the list

$$\text{Ave \# of Looks} = (n+1)/2$$

b. Directories:

(i) Single Level Directory:

$$\begin{aligned}\text{Ave \# of Looks} &= (\sqrt{n}+1)/2 \quad \text{for directory} \\ &\quad + (\sqrt{n}+1)/2 \quad \text{for sub-list} \\ &= 2*(\sqrt{n}+1)/2 = \sqrt{n}+1\end{aligned}$$

(ii) Two Levels of Directories:

$$\begin{aligned}\text{Ave \# of Looks} &= (\sqrt[3]{n}+1)/2 \quad 1^{st}\text{directory} \\ &\quad + (\sqrt[3]{n}+1)/2 \quad \text{2nd directory} \\ &\quad + (\sqrt[3]{n}+1)/2 \quad \text{sub-list} \\ &= 3*(\sqrt[3]{n}+1)/2\end{aligned}$$

(iii) Three levels of Directories:

What would you think, based on the above?

$$\text{Ave \# of Looks} = 4*(\sqrt[4]{n} + 1)/2$$
$$= 2*(\sqrt[4]{n} + 1)$$

c. Binary Search

This is what the SEARCH ALL … verb does, and it requires a lot of calculation. The binary search process first examines the ordered list in its middle, then determines in which half of the list the desired record will reside. Then it looks in the middle of *that* half, and determines which half of that half the desired record will reside. And so on, successively dividing the segments by two (2).

The formula is $\text{Average \# of Looks} = \log_2 n$

That is, for $2^{\text{\# looks}} = n$. If n = 16, how many looks? (2 to what power equals 16?) What if n = 64

NOTE: Even though everyone is an expert in logarithms, it might be useful to note that we can use either common (to the base 10) logarithms or natural logarithms in the following, mathematical process.

If we look at $B^x = Y$, then

$\log (B^x) = \log y$

Or $x * \log B = \log y$

And $x = \log y / \log B$

B. INVERTED LISTS

An inverted list (or file) is an additional list in which one of the attributes (and not the unique record key) is the basis for the order of the list. Often, the attribute is a class. That is, the attribute value determines the order. Thus, for an inverted list, the role of the unique record key and the attribute are "inverted".

Looking at the next of your handout page (has Fig 5-15 printed on it), we see that the primary list is ordered by social security number.

The next page presents to lists, or indexes (since they don't contain the complete record). Both are ordered by last name. One directs the searcher to the record id (the ssn) in the primary list, and the other directs the searcher to the location.

In the next figure (denoted as Figure 5-18), we have an index ordered by work division, with the locations of those employees in that division listed after the division entry. NOTE THAT THE WORK DIVISIONS ARE ORDERED.

The next figure shows another way – a bit tortuous – with the address to the first entry for that division and with a link to another structure (a linked list at that!) which carries the addresses of the additional records of employees in that division.

Following, the figure shows a *fully inverted* index list, with the entries in the index in ordered sequence. And another representation. (Discuss briefly.)

The next figure is a ***partially inverted list***, based on major, in which links connecting elements in each major group has been added to the prime data list.

Why do we say, partially inverted and not fully inverted? With a fully inverted list, there is an entry – of one sort or another – for each record in the primary list. A partially inverted list does **not** have an entry for every primary list record.

The last figure we will consider is a modification of the previous one in that the **major** field is removed from the prime data list since the majors are specified in the partially inverted list. This could be risky if one wished to process the prime data list directly.

C. HASHING CONCEPTS

The inverted lists, the multi-linked lists, and the directories are examples of employing suitable auxiliary data structures to help us find a "record" faster. How? In the case using directories, we first search a subset list constructed from the ordered, primary list called a directory. The result of the search of the directory results in the selection of a directory item that contains the address of a particular sublist of the primary data list. That is, the directory puts us in the "neighborhood".

In the case of inverted lists, items that have a common attribute are "grouped" (that is, logically connected in one way or another). There is a not-to-distant similarity in an inverted list and a file ordered for control-break processing.

There is another way, however, to provide "neighborhood locatability". That method is called "***hashing***".
Instead of an auxiliary data structure as is the case with directories, the hashing approach makes use of a set of computations. Specifically, function is applied to a record key to provide an address, namely a

KEY-TO-ADDRESS transformation or mapping.

We select a "nice" function (to be determined later) that is applied to the record key to yield an address. That is ,

F(record-key) = record-address

How nice a simple this sounds! If we are given a search key, we need only calculate the address of the record we want. (The record may be on a disk or in a table in memory – the concept is the same.)

There is a problem: Any function that we know of results in a very large percentage of unused record locations, a condition which is not acceptable. But what we *are* able to do is identify functions that will yield a given set of addresses. (The number of different addresses is our choice.) Further, we won't have such a high percentage of empty, unused record locations, and, for a "nice" function, we won't be spending a terribly lot of time computing nor accessing. The set of hash values are called " buckets" because with each value, there are several records whose keys yield the same result when their respective keys are "hashed" (the function applied to them). Even though all the keys are unique, they *all* yield the same value. The records have no logical relations among them in terms of record key values or attributes. Any similar items are purely accidental. The only thing in common with records in the bucket (which can be considered a sub-list) is that they have a common hash value.

Let us now state formally the purposes and goals of hashing techniques (i.e., hashing functions).

a. (Ideal) To Provide a unique address for each record, a one-to-one mapping (no buckets with groups of records. This is not practicable for reasons mentioned above (too many empty spaces).
b. (Realistic) Provide a uniform distribution of records for each hash value. That is, we want an equal (or something acceptable) number of records for each bucket.
c. Be efficient (in the sense of computational speed) in calculation and accessing of records
d. Provide an acceptable level of data storage density (the percentage of empty, unused locations is acceptable).

What is a good function to realize the goal of b. above? (Remember: what we have is F(key) = hash-value.)

The simplest and most accepted general purpose algorithm is the *"division-remainder"* function. When used properly, it gives good, sometimes near, uniform distribution of the records among the buckets, AND it is simple!

It is based on the modulus function: n (mod b).

Examples: 18 (mod 7) = 4 That is, 18 divided by 7 has a remainder of 4.
20 (mod 5)= 0
5002 (mod 3) = 1

Since we cannot address anything (at least not in most user languages) with an address of 0, we construct our hash function as follows:

HASH VAL = key (mod b) + 1

NOTE: This does **not** represent an address, just the number of the bucket, and there are a total of **b** buckets.

Examples: For a key of 1234 and b = 23 (i.e., there are 23 buckets)

HASH VAL = 1234 (mod 23) + 1
= 15 + 1
= 16

Key = 2500
HASH VAL = 2550 (mod 23) + 1 = 21

Key 2300
HASH VAL = 0 + 1 = 1

How do we deal with all of this? How do we select the number of buckets we want? While this question goes beyond the scope of this course, we can look at a brief example. If we will have a maximum of 10,000 records, and we want a maximum of 110 records per bucket, then we must have 91 buckets. Why 91?

Well, 90*110 = 9,900, which is less than 10,000. So we must 91 buckets, since 91*110=10,100 records.

It is interesting to note that if we had 10,000 records per bucket, we would have only one, big bucket, which is essentially the original file. We wouldn't save anything in access time because of the long time to search.

At the other extreme, if we had only one record per bucket, we would have very fast access time, but, as noted above, an undesirable percentage of empty record locations.

Are there any criteria for the selection of the number of buckets? Criteria that assures us of a function which will yield a reasonably uniform distribution of the records among the buckets? Clearly, it depends only on the value of b. Yes! Some literature suggests that the number of buckets, b, should be the first prime number that equals or exceeds (in a whole number) the ratio of N/k, where N is the number of records, and k is the number of records per bucket.

Other literature says that selecting the first odd number that is equal or greater than N/k works almost as well.

So much for that.

SEARCHING: What does all this mean in terms of the steps to be taken to access a record?

In general, the steps are

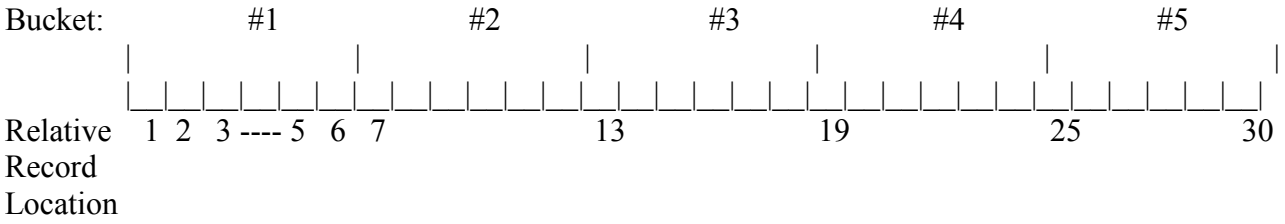
1. Calculate the hash value by applying the remainder operation the record key to determine in which bucket the record is located;
2. Go to that bucket and perform a linear search of the records in that bucket (remembering that the records in that bucket are not ordered).

An elementary look at this approach is to use a HASHTABLE. Its use is demonstrated on the last page of your handout. In this example, the records in a particular bucket (that is, the records that have the same hash value) are linked together. Each value in the HASHTABLE is really the address to a particular sublist.

TERMINOLOGY NOTE: When two or more records have keys that yield the same hash value, they are called "synonyms" a technical term borrowed from the condition when two words have the same meaning. The result of this is called a "collision", which is what we would have from trying to place a record in the same location. So we must take additional steps to place these two (or more) in the same bucket, which, as we know, may contain many records.

We will now examine the processes required to access a record in a file which is logically (that is to say, we have devised the logic) organized into buckets.

Suppose we have a file of 25 records with 5 buckets that are created to hold 5 records per bucket.



The sequence of accessing a particular record (or record location, depending on the purpose) is:

1. Calculate the beginning address of the bucket;
2. Perform a linear search of the record locations in that bucket until the desired condition is satisfied.

How do we do step 1? If we calculate for a particular key bucket #2, we want to start our linear search a record location 6. What kind of algorithm will yield the correct beginning address for each bucket?

Beginning Bucket Address = [key-value (mod 5)] * 6 + 1,

or, in general, Beginning Bucket Address = [key-value (mod b)] * k + 1

where b and k are defined as above.

The Step 2 process depends on the condition for which the linear search is satisfied. That is, it will depend on whether we are searching for a particular record with a particular key or searching for an "unused" record location, one which does not have valid data in it, as would be the case when loading the file.

D. RELATIVE FILES

Before we continue with the next two file structures, we should examine FILE PROCESSING in a very general sense.

Essentially, file processing consists of

- Reading a transaction record
- Processing the master file to accomplish the transaction (update, delete, insert)

This logic is pretty much true no matter what the structure of the master file. What are different from one file to the next are the logic of the search process and the logic of how the transactions are processed. These logical steps are clearly dependent on the structure of the file. The steps required for Assignment 4 is very much different than the steps, for example, in your first program to update a master, sequential file. Consequently, we must tailor our methods to the structure of the file.

An all-important issue when updating a master file (or any file, for that matter) is providing for a back up so that all is not lost if the file is corrupted. With the old, classic master file update of a sequential master file and an ordered, transaction file, a completely new master file was written. Then, if the new master file were corrupted, we could rebuild it from the old master file and the transaction file (implying, of course, that we kept them around in a safe place).

But things are different with most of the master files today. These files are nearly always updated in place, much as your modification of a document file changes the contents of the file. (The more current file structures support this.) Consider that we have a system that processes the transactions as they occur. What happens if disaster strikes? Unless we kept a back up or if your system made automatic backups, we were lost.

What kind of procedures should we have to back up our files, to recover in case of disaster? Basically, we should

- Periodically make a copy of the master file and store it in a secure location
- Copy each transaction to file (generally a tape file, we don't expect problems every day!)

Now we have the data to reconstruct the master file using the copy of the master file and the tape file of the transactions made since the last backup copy was made. And life can go on.

Now we can discuss relative files specifically.

You have already been presented with various aspects of relative files. It is appropriate to repeat them.

1. Overview: A relative file is characterized by the ability to provide direct access. The term "relative" means that the *address* of a record location is *relative* to the beginning of the file. Thus, if we wish to access the 14th record location, the address is 14, and we may access it by simply providing that address to the system. (This is what you are doing in Assignment 5.)

   We will employ a suitable *hash function* to yield an address. This *key-to-address* transform is an algorithm that operates on a *key value* as input and produces a *relative address* as output. The address must be within the range of the total space allocated to the file. Any two key values, K1 and K2, that, for a particular hash function, yield the same relative address, these keys are called *synonyms*. The phenomenon that K1 and K2 'hash' to the same location is referred to as a *collision*.

   How do we handle collisions? One of the simplest methods is the *linear probe*. This method is simply to add one (1) to the current address (the one that already has a record in it), and look at the next location. That is, if a location is full, look next door. This is the method you will use in Assignment 6.

   What do we do is the bucket is full of records and we have another record that should go in it? (NOTE: This event will not occur in Assignment 6!) One solution to this is to use an *overflow area* that may be used by all buckets. The records placed in this area for a particular bucket are linked together. That is, there is a linked-list for each bucket.

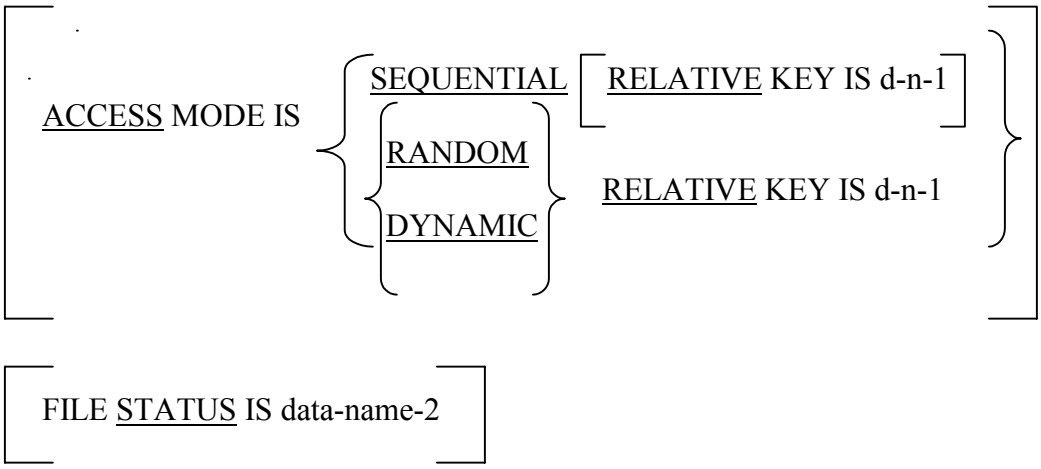
   In processing relative files with random access, there are three important steps to be considered *before* accessing the file.

   (a) Calculate the numerical address of the desired record location. (This step is not required if the desired address is already available.)
   (b) Move the numerical address to the working storage location that was designated in the RELATIVE KEY statement of the SELECT…ASSIGN… clause.
   (c) Execute the desired I/O command (READ, WRITE, REWRITE, etc.)

   NOTE: Steps (a) and (b) are not required if, because of previous operations, the desired value is already in the relative key.

2. COBOL STATEMENTS

ENVIRONMENT DIVISION.
SELECT internal-file-name
ASSIGN TO external-file-name
ORGANIZATION IS RELATIVE



Data-name-1 *must* be defined in WORKING-STORAGE SECTION as an unsigned, integer, numeric field.

Data-name-2 is defined in working storage as PIC XX.

The FILE STATUS code value is placed there by the system to indicate the status of the results of a particular file access command (including OPEN and CLOSE statements). If the execution of the command was successful, the FILE STATUS code will so indicate. If the command could not be executed, the FILE STATUS code will indicate why (more or less!). Please refer to your text or to Micro Focus Help for the values of the codes and what each of them mean.

I-O VERBS:

1. **WRITE** - File opened as OUTPUT, I-O, or EXTEND

   (a) SEQUENTIAL access mode:

   WRITE record-name [FROM identifier-1]

   NOTE: If the relative key clause is used, the system places the value of the relative record location into the relative key data-name.

(b) RANDOM access mode:

WRITE record-name [FROM identifier-1]

INVALID KEY imperative-statement-1

[NOT INVALID KEY imperative statement-1]

[END-WRITE]

2. **READ** - File opened as INPUT or I-O

(a) SEQUENTIAL access mode:

READ file-name [INTO identifier-1]

AT END imperative-statement

(b) RANDOM or DYNAMIC access mode (relative key value must be in the relative key data name):

READ record-name [INTO identifier-1]

INVALID KEY imperative-statement-1

[NOT INVALID KEY imperative statement-1]

[END-WRITE]

NOTE: This is for a random read ONLY!

(c) DYNAMIC access mode

(i) Random read is the same as (b) above

(ii) Sequential read

READ file-name NEXT RECORD [INTO identifier-1]

AT END imperative-statement

3. **REWRITE** - Only may be used for file opened as I-O

Prior to executing the REWRITE verb,
- a READ of any type must have been successfully executed
- the modified or new record has been prepared

REWRITE record-name [FROM identifier-1]

INVALID KEY imperative-statement-1

[NOT INVALID KEY imperative statement-1]

[END-WRITE]

4 **. DELETE –** file opened as I-O, any access mode. This command "logically removes" a record (sets a flag indicating that space at the record location is 'free').

For SEQUENTIAL access, a READ must be executed first.
For RANDOM or DYNAMIC access, the desired relative key value must be placed in the relative key data name.

After a delete has been executed, the only verb that can access that record space is a WRITE statement. The flag is a one-byte field that the system attaches to the record. It is not considered when defining the picture of the record. This 'flag' field is not visible to the program.

DELETE file-name RECORD

INVALID KEY imperative-statement-1

[NOT INVALID KEY imperative statement-1]

[END-DELETE]