Abstract

A file handling system which allows the user to define a database according to his own needs is described. The operations on the database include the ability to append new entries, modify them as well as delete unwanted ones and list the entries in a sorted order. In order to meet this end the binary tree file structure was used and appropriate algorithms were designed and implemented. Included is a full description of the data structures and algorithms used. The whole project was tested and its performance evaluated.

Name : Diomidis D. Spinellis
School or College : Athens College (GCE School)
Centre Number : 92060
Candidate Number : _____
Title of Project : Database Management System

1 Summary of purpose and scope of the project

1.1 Need

In a society where people are living in organised communities one is often overwhelmed by the mass of data one encounters when one deals with various aspects of our lives. The human memory which was once a reliable way to organise any activity is incapable to deal with most data connected situations in our modern world.

Various filing systems have been used over the years to deal with this problem. Most of them relied on the printed word and the ability to store many pages in a small space. Typical examples are the telephone directories and the library catalogues. These filing systems have usually arranged their entries in some predefined order (usually alphabetic) and so, one can easily locate the entry one is looking for. Some of them, like a book index or a library catalogue, are usually just look up tables for larger databases.

These systems have various inefficiencies. Namely they consume much space, are difficult to use and many of them are difficult to update (Encyclopedias and telephone directories are reprinted at periodic intervals).

With the advent of the microcomputer and the availability of cheap mass storage media the computerising of these databases has become feasible.

1.2 Purpose

The purpose of this project is to demonstrate this capability by designing a system that would allow the user to define and use a database. The following features seem to be useful : Field naming, record indexing, insertion, deletion, display and editing of entries as well as a sorted listing of them.

Furthermore this project should demonstrate the use of the binary tree file indexing method which should provide speed and efficiency.

2 First some terms ...

It would seem appropriate at this point to discuss some terms that will widely be used during the next pages.

File : is a set of structured data and in this text it will always be associated with its presence in the backing store (disk).

Record : will be used to identify one entry in the file.

Field : each record is subdivided in fields. Each field which consists of one or more bytes contains different types of information.

Tree : is a way to structure data. Every data item contains pointers to other data items and here each data item has only one pointer associated with it. Thus the whole structure resembles the form of a tree.

Binary tree : binary trees are a special kind of trees. On them each item points to zero one or two other items.

Node or leaf or branch : is the name that is given in this project to the pointer which points to the next items in the tree.

Root : every item in the tree with the exception of the first one is pointed by another item .

3 Discussion on the specifications

The specifications seem reasonable and of the kind that can be implemented with the existing software, hardware and expertise. It seems that an analysis of the goals, and more important, the shortcomings of the system, is something that will be needed because of the complexity of the task. A project of this size when made to fit around the 600 line guide will most probably have many shortcomings.

However I believe that it will be useful on its own account as a pilot project for a complete database design. As for the aspect of reliability I can only quote one of Gilb's laws of reliability "Investment in reliability will increase until it exceeds the probable cost of errors or until someone insists on getting some useful work done."

4 Available hardware tools

4.1 CPU

The CPU to be used (Apple IIe 6502 based) is a typical von Newmann machine based on a microprocessor. It has the capability to work on 8 bit quantities (bytes) on various manners. It is connected to a modifiable random access memory and to a random access read only memory. Only one user can use the machine at a time so no provisions must be taken in order to implement resource sharing such as record locking, semaphoring and other multi-user or multi-tasking procedures. Almost inherent on the design of the machine and mainly based on the input/output devices is the collating sequence of the character set. The character set used is the ASCII character set and its order is used as a collating sequence. This sequence is used in order to compare to strings of bytes.

4.2 Main storage

The main storage is composed of 128k of random access modifiable memory. The access time for this memory is of the order of hundreds of nanoseconds. This storage will be used to hold the operating system, the program as well as the structure of the data-file and other variables. Two buffers are also allocated by the program . These buffers are used as a temporary storage for the index file contents during their manipulation. As the operating system provides the file buffering the area of the main memory which is used for file buffering is allocated by the operating system.

4.3 Backing storage

Two backing storage devices are available : A non removable Winchester disk drive and one 5 1/4 floppy disk drive. The hard disk drive has a capacity of 20M bytes and the floppy disk drive a capacity of 200K bytes .On both of them data is allocated in the form of clusters which are formed by the division of them in tracks, sectors and in the case of the hard disk drive in cylinders.

The slowest operations on this type of media are the head movement which results from the stepping from one track to the next and the motor startup time (only for the floppy disk drive). So it will be of great advantage to the program a) if data is allocated in clusters on the same track and b) if the operating system buffers the read operation and stores e.g. one whole track after every read.

4.4 Input

The main input device for this microcomputer is the keyboard. It seems appropriate for this kind of project. The keyboard is directly connected to an I/O port of the CPU. All functions required to scan the keyboard matrix and translate the result to an ASCII code are handled by the built in ROM routines.

A serial port is also available. Without programme modification it could be used to connect a bar code reader. Usually these devices come with software and/or hardware which "traps" the requests for a character read and supply their own output when needed. Thus no provisions are made for this kind of input device, although it can be very effectively used in various database usage environments, such as libraries, stores etc..

4.5 Output

The output devices are a Video Display Unit and a dot matrix impact printer .Both of them can display the full ASCII character set. The VDU supports advanced functions like direct

cursor addressing and screen clearing. Although the printer supports dot addressable graphics, they will not be used.

5 Available software tools

5.1 Input Output system

The I/O system is located in ROM. It handles all requests for the I/O devices including the printer Centronics protocol, the screen dual port memory access arbitration and the backing storage control in a primitive way. Functions like get a character or put a character or read/write a sector are available.

5.2 Operating system

The operating system is able to handle more complex tasks. The operating system is the UCSD P-system originally developed at the University of California San Diego and now distributed by SofTech Microsystems. It is composed by a kernel written in native code and a set of utilities and more advance functions written in a high level language, namely Pascal.

A main advantage of this operating system is the portability it guarantees for all products that convey its standard because the language compilers it provides are not producing native code but an intermediate pseudo code called P-code. The P-code is implemented as an interpreted code in all machines and thus for the operating system to become available on a specific machine only a new P-code interpreter is needed. A disadvantage of the system is its slowness because of the interpreted code. Compared to other operating systems like UNIX MS-DOS or VMS it is rather primitive.

Nevertheless it is able to handle I/O redirection and random access files which are used in this project. All I/O requests are transferred to the operating system via the Pascal compiler calls.

5.3 Language

The language used is the UCSD Pascal . UCSD Pascal is the Pascal variant used in the UCSD P-system. Pascal is suitable for this project because it supports data structuring and strong type checking. Strong type checking was especially useful in the early stages of the development where much of the checking for program consistency was done by the compiler. Pascal has rightly been described as a wilfully worn straightjacket. I believe that with the structured programming habits it imposes on the programmer self-documenting, clear, easy to understand and maintain programs are created.

Two of the procedures used are recursive and because Pascal supports recursive procedure calling their logic is easily understandable. Records are used extensively and this makes the relationship between various data items more clear and easy to understand. Local variables eliminate the danger of a subroutine to inadvertently change a global variable and passing by reference allows subroutines the change the contents of variables. Sets have been used to a lesser extent mainly for input validation. While developing this project I thought of using variant records as a way to represent the varying users needs. However because variant records do not allow someone to define e.g. an array whose length would be known during the runtime, this idea was abandoned.

All the structuring constructs were used i.e. IF THEN, IF THEN ELSE, WHILE DO, FOR DO, REPEAT UNTIL, CASE etc.. Each one of them proved its usefulness in many situations. The GOTO statement was not used at all. A small inconvenience was created by the absence of a statement like the return() statement in the C programming language which allows the programme flow to exit from any point of a function. Thus many functions are nested in IF THEN ELSE blocks only because an error was found at the beginning of the function.

6 Data structures employed

6.1 Memory

The following structure types are used :

The index file as well as the memory buffers are composed of IndexFileRex. It should be noted that all pointers that point to nothing have the reserved value of 0. This is composed as follows :

```
IndexFileRec=Record
  LeftBranch  : Integer; Pointer to the left node.
  RightBranch : Integer; Pointer to the rigth node.
  Key         : KeyType; Key on which the file is indexed.
  DataPoint   : Integer; Pointer to the database file.
End;
```

The information of each field is stored in a record which contains the name of the field, its length and the validation type. The validation type can be any of the following :

```
'A'  ['A'..'Z','a'..'z']  Alpha.
'N'  ['0'..'9','+','-','#','.',',']  Numeric.
'D'  ['0'..'9'] Digit.
'Y'  ['Y','N','y','n','T','t','F','f'] Yes or No True or false.
'E'  [' '..'~'] Everything. All the ASCII set.
```

The record named FieldInfo is defined as follows :

```
FieldInfoType = Record
  Name        : string[];
```

```
    Length      : Integer;
    Validate    : Char;
  end;
```

The information about the whole file structure is stored in a record of FileStrucType which holds the number of fields, the record length and the field on which the database is indexed. Moreover it holds an array of length as the maximum number of fields of field specific information as described above.

```
  FileStrucType = Record
    FieldNum,IndexField,RecordLength : Integer;
    FieldInfo : array[1..MaxFieldNumber] of FieldInfoType;
  end;
```

6.2 Disk

6.2.1 Structure file

The structure file is a text file. Because of this it can easily be displayed and altered, something which can be an aid during the programme debuging. The file contains the following data :

```
No of fields (integer)
Field 1 Name (String)
Field 1 length (Integer)
Field 1 type (Character) (Validation type as explained in ValidRead)
Field 2 Name
Field 2 le.....................................
.........................................
.........................................
Field n type
Field on which the Database is indexed (integer)
```

6.2.2 Index file

The index file is a random access file which consists of entries of type IndexFileRecType. These are stored one after the other. The first two records are not holding any user information. They are more fully described in the initialization section.

6.2.3 Data file

The data file consists of characters in order to save space. The record length is variable and so a special procedure reads all fields in a procrustean manner i.e. chopping the long ones and padding the short ones before they are written to the data file. This results in huge space savings as one can easily

imagine when one thinks of a telephone directory with a record length of 200 characters.

7 Description of the algorithm

7.1 Search

In order to locate an entry in the index file the following algorithm was used :

```
While there are other entries and the entry is not found
    Read an entry
    Compare the two keys
    greater : new position is right pointer
    less : new position is left pointer
    equal : entry found
    If the new position is 0 there are no other entries
end
If the new position is 0 then
    the entry was not found
else
    the entry was found.
```

During the whole search a global variable named root points to the previous file position. This serves two purposes : a) If the entry is found the root position is known and can be used by a procedure, such as delete, to eliminate the data links, and b) If the entry is not found, the root obviously reflects the nearest point where the key should be and can thus be used by procedures such as the insert procedure.

7.2 Insert

The insert procedure is relatively simple because it uses much information of the search procedure. First of all I check and verify that the entry is not already present. After that I allocate space on the file for the new entry. The space can be allocated from two sources. Either from an already deleted entry that is unlinked from the deleted entries linked list - or if such an entry does not exist - the new entry is directly appended to the end of the file. After that, and based on the global variable Root which after the search call indicates the entry where the new entry should be hanged the only thing needed is to find out which of the two nodes of the Root entry must point to the new entry. This is easily found by comparing the two keys. Subsequently the Root entry is modified to point to the new entry.

7.3 Delete

7.3.1 General

The delete procedure was the one that I found the most difficult in this project. Two were the main difficulties that I encountered. First of all the problem of removing an entry from the binary tree without ruining the tree structure and secondly the problem of re-usage of the space. The first problem was divided in three different situations which I will shortly explain. The second problem was solved with the use of a linked list. When an entry has been deleted the pointers have no more use. So I started A linked list using as pointers the left pointers of all deleted entries starting with the left pointer of the dummy record #0. This linked list is updated for the sake of simplicity from its start i.e. from the entry #0 both for insertion and for deletion. This is not the most elegant solution because now the linked list works as a LIFO storage (Last in first out). This means that the last entry deleted will be the first entry to be reused and this makes it impossible to add an undelete feature to the programme.

In order to add an item to the linked list entry #0 is simply made to point to this item and the left pointer of this item holds the old contents of the left pointer of entry #0. As usual the reserved value of 0 means that this is the last entry.

7.3.2 No nodes

Now I will consider the different aspects of the delete operation. One entry can have by definition zero, one or two nodes. The easiest case is that where an entry has zero nodes. In that case in order to remove the entry from the tree structure the pointer of its root which points to it is simply assigned the value of 0.

7.3.3 One node

A more difficult problem occurs when the entry point to one and only one entry. As one can easily understand this segment of the tree resembles a linked list so the only way to remove an entry from its middle is to make its predecessor point to its successor. This is what the delete procedure does in this case. In order to minimise the decision making paths one small trick is used : The successor pointer is found by adding the left and the right pointers of the entry as one of them is guaranteed to be zero.

7.3.4 Two nodes

The most difficult situation occurs when an entry is surrounded by others i.e. it has two successors. Obviously the linked list algorithm can not be applied here. Instead another method is applied. For the sake of simplicity this method is composed as a hybrid between the deletion of one node and the insertion of another. The routine works as follows : I assume that this entry has only on successor. For that purpose I save the other successor pointer in a local variable and then I zero the pointer. I may now call the delete procedure recursively which will apply the one-node strategy and thus return to this point after having removed the entry from the tree. What now remains is one successor which is completely out of the tree structure. I now proceed to insert this successor into the tree structure in the normal manner. Now the tree structure is left intact.

7.4 List

The list procedure was implemented because of its nature in a totaly recursive way. Assume that eone wants to list a specific entry. If the left node of the entry is non zero one will have to list that entry first. After that the current entry must be displayed and then the right node entry must be listed (if it exists). So I am now ready to discuss the elements of this recursive strategy approach .

a) Reduction : The problem list(entry) is reduced to the problem list(entry left node), display entry, list(entry right node).

b) Termination : The recursive process terminates when one of the nodes is zero something which is true for the whole tree frontier.

A more formal proof of this procedure is beyond the scope of this project.

7.5 Edit

In order to minimise the code complexity the editing is composed of a deletion of the entry to be edited and after that a new insertion. Because of the way the deleted space is used no space waste is done. This method has two disadvantages : a) the order of the tree is disturbed and a series of edit operations may yield to an unbalanced tree b) It is time consuming.

However this method has a serious advantage over any other method. It allows the user to alter the key name if he so wishes.

Before the insert procedure is called the global variable Edit is set to the value True and the old contents of the entry are read into the global variable OldDataPack . The read data procedure behaves in a different way under these circumstances. Namely before each entry it asks the user if he wishes to change

it or not. Only in the case the user wishes to modify the contents the routine asks him to re-enter them. This saves some amount of repetitive typing.

7.6 Initialize

7.6.1 General

When a new file is created the three data files must be initialized. This is performed immediately after the creation in order to avoid unnecessary program complexity during all operations. The data structures in the files are designed in such a way so that after initialization the file will always appear the same to all procedures. Having to take into account during the search or the delete procedure whether the file is empty or not would double the procedures complexity.

7.6.2 Structure file

The structure file is filled with validated entries which show the number of fields per record and the name, length and type of each field. It is made sure that the total length of all fields does not exceed 200 characters and that no type that does not exist is entered. The field onto which the whole data file is indexed is also recorded.

7.6.3 Index file

The index file is initialized by being filled with two dummy records. Dummy record #0 is a place holder which ensures that no pointer will ever take the reserved value of 0. This value has the special meaning that there exists no other entry for the specific chain. One other use of the record #0 is to mark the beginning of the deleted items linked list. As the file is empty and no deleted items exist it takes the value of zero. Thus when a deleted item is added to the chain it will take the value of zero and thus mark the end of the linked list. In this way the linked list add remove algorithm is simplified.

The second record that is initialized is record #1. This record is made to contain a blank entry which can not be entered by the user. Both nodes are made to point to zero. This second record exists so that I will always know the beginning (root) of the tree. If that was to be a user entered record I would have to provide a way to distinguish between an empty file and a non empty file during insert and I would have to hold the start of the tree in a separate variable because this entry could also be deleted. It is obvious that programme complexity is minimised by the use of this strategy. The greater search time is independent of the search length and we can thus ignore it.

7.6.4 Database file

The database file is just pointed by the index record and it needs no other initialization other than the ReWrite procedure that will remove its old contents (if any).

7.7 Analogies between algorithm and human behaviour

After having described all the algorithms used in this project it would be interesting to observe if any analogies between this system and other non-machine assisted systems exist. One system that suits our purpose is that of the library catalogue. When one wants to find an entry in a card based system one opens the card drawer and looks at the front card. Most probably the book one is searching for is not there. After that one may take two actions. Either he will guess the cards approximate position by taking into account the letter on which the card index started and the letter he is searching for (which is an approximation to the hashing algorithm - not implemented in this project) or, most likely,he will look at the card in the middle of the drawer. He will then proceed either to the cards in front of the middle or those on the back of the middle each time halving his distance from the entry he is searching for.

This is an approximation to the binary tree searching algorithm that I have implemented in the project. Having found the entry he is searching one will now look at the number of the shelf where the book can be found. The analogy with looking at the index to the database file from the indexfile is too obvious!

8 Data validation and error analysis

8.1 Input validation

During all user entries the programme validates the entry so that it is of a legal value. This is done by calling a special procedure which only allows validated entry. Furthermore the numerical entries are checked to be within specific bounds.

8.2 Internal validation

The programme is not designed in a vary robust way towards internal errors. For example a node that would point to its self would make the programme crash. For this reason this programme is not recommended for heavy or critical use. Moreover the programme lacks the ability to correct errors in the structure of the file.

The only measure that has been taken to avoid such errors is the stringent testing of the programme and the validation of the user inputs.

8.3 User validation

The ultimate user of this programme will be someone who has little or no experience with automatic data processing (e.g. a librarian). For this reason the programme enables the person who designs the database to specify the kind of fields he wishes to use and more important their type.

In this way the ultimate user of the programme will not be allowed to enter letters in a numeric field or a number in a Yes/No field.

8.4 Programme limits

In order to cope with the need to declare the array length before the compilation certain limits had to be imposed. The use of pointers and dynamic memory allocation would make the whole programme too complex without making it much better. (It would only marginally effect its capabilities). The following limits were decided. As all the limits are coded as constants into the programme it will be relatively easy to change them if one so wishes.

Maximum number of fields	: 20	fields
Maximum field name length	: 30	characters
Maximum record length	: 200	bytes
Maximum Key length	: 30	characters

9 Programme

9.1 Procedures and functions

```
function ValidRead( ValidType : Char ; StringLength : Integer ) : MaxString ;
```
Read a string with input length and type validation

```
procedure Prompt(Name : MaxString);
```
Will initialize the screen for the operation named in name i.e. Clear the screen and write the name on the top.

```
function Upper(C : Char) :;
```
Make C uppercase if required

```
function Compare(A,B : KeyType) : CompareResult;
```
Compare to entries of the index file according to ASCII collating sequence and return Greater Less or Equal

```
procedure IndexRead(Where : Integer ; var What : IndexFileRec);
```
Read an entry from position Where in the index file into What

```
procedure IndexWrite(Where : Integer ; What : IndexFileRec);
```
Write an entry to position Where in the index file from What

```
procedure InitVars;
```

```
Initialize global variables

procedure OpenFiles;
Open all database files

Procedure CloseFiles;
Close all Database files

Procedure PrepareNewFile;
Initializes all Database files by creating new ones. The user
specifies their characteristics.

Procedure UseFile;
Prepares the program to use an old data file by reading the
structure file contents

procedure DataRead(ItemNum : Integer ; var DataPack : RecordString);
Read data from position ItemNum in the database file into DataPack

Procedure DisplayItem(ItemNum : Integer);
Display the contents of the item located in position ItemNum in DBF file

Function KeyName : KeyType;
Returns the name of the field onto which the database is indexed

Procedure NotFoundError(Name : KeyType);
Reports an error if entry with key name Name was no found

function ReadKeyName : KeyType ;
Returns the name of the record the user wants to act upon

Procedure ReadItem(ItemNum : Integer ; Key : KeyType);
Reads all the contents of a record. If Edit is true allows default responses
according to the variable OldDataPack. ItemNum points to the database file

Procedure List(Node:integer);
Goes through the tree structure in the collating sequence manner

Function Search(Key:KeyType):integer;
Return the Position of Key in file, 0 If not existing.
Also set the global Variable Root to the Root of the record or where the
nonexisting record should be hanged.

Function Insert:Boolean;
Inserts a record into the file structure. Return False if it exists.

Function Delete(Key:KeyType):Boolean;
Removes the record Key from the tree structure and returns True if it exists.
The record is also appended to the deleted records linked list.

procedure OptionsScreen;
Prompt the user the available options he can perform
```

9.2 Programme body

The programme body is a big repeat-until loop. The options available to the user are displayed and the user presses the first letter of the option he wishes to use. When no file has been selected only three options are available. The user may either create a new file or use an existing file or quit to the system. After an existing file has been chosen the user may specify all other options such as add a new entry or view all entries.

10 A sample session

10.1 Some technical information

In the following pages a sample session is displayed. The screens that are printed are real screens from the programme. They were generated in the following manner: The programme was compiled on an IBM-PC with a Turbo-Pascal compiler. After that the programme Sidekick was loaded as a memory resident programme. This programme offers the capabilities of a Note-Pad, a Calculator, a Calendar and an ASCII table at any instant of the computers operation. When the database programme started running and the first screen appeared I entered Sidekicks Note-Pad feature. This Note-Pad is like a word processor that opens a window on the screen one is operating. It has the capability to import things from the background screen (which was the database system screen). So I imported the screen and saved it on the disk. In this way 15 screens were created. The interesting thing was that a look at the directory of the disk showed me that not only had I captured samples of the most important screens of the programme but furthermore I had a specific timetable of the whole operation as the MS-DOS marks the dates and times of all files. In the following screens I have inserted some comments. These are written in **bold** characters. No other editing was performed.

10.2 Thirty minutes with the programme

Directory of files produced

Name	Size	Time	Comments
START	375	9:30a	;Starting screen
CREATE	1215	9:39a	;Create a new file screen
USE	105	9:40a	;Use an existing file screen
SCREEN2	494	9:42a	;Starting screen with file in use
ADD1	251	9:43a	;Add new entry screen
ADD2	113	9:46a	;Add new entry screen
ADD3	247	9:49a	;Add new entry screen
LIST	750	9:51a	;Alphabetical listing
DISP1	231	9:52a	;Display existing entry screen (with error)
DISP2	268	9:53a	;Display existing entry screen (no error)
REMOVE	118	9:54a	;Remove entry screen

```
LIST2          574 9:57a      ;Alphabetical listing screen
EDIT1          331 9:59a      ;Editing screen
EDIT2          371 10:00a     ;Editing screen
QUIT           106 10:00a     ;Quit screen
```

Starting screen

```
GCE  A Level Computer Science Project         (C) 1985,86 Diomidis D. Spinellis

                            Data Base Management System
                            ---------------------------



Available Options
-----------------

C(reate a new file
U(se an existing data file
Q(uit from the programme



Select operation by typing the options first letter
```

The starting screen (Only three options are available)

Create a new file screen

The create a new file screen. Some erroneous entries were made to demonstrate the input validation. It should be noted that some errors could not be made to appear on the screen as the programme would just refuse to accept illegitimate characters.

```
Create a New File
-----------------

File name :programs
Number of fields (1-20) :0    <-Error in entry (too small)
Number of fields (1-20) :30   <-Error in entry (too large)
Number of fields (1-20) :7
Enter field 1 name :Company
Enter field length (up to 200 ) :25
Field type :A(lphabetic N(umeric D(igit Y(es/No E(verything :E
Enter field 2 name :Title
Enter field length (up to 175 ) :176
Enter field length (up to 175 ) :20
Field type :A(lphabetic N(umeric D(igit Y(es/No E(verything :E
Enter field 3 name :Usage
Enter field length (up to 155 ) :30
Field type :A(lphabetic N(umeric D(igit Y(es/No E(verything :E
Enter field 4 name :Number of disks
Enter field length (up to 125 ) :3
Field type :A(lphabetic N(umeric D(igit Y(es/No E(verything :D
Enter field 5 name :Rating
Enter field length (up to 122 ) :4
Field type :A(lphabetic N(umeric D(igit Y(es/No E(verything :N
Enter field 6 name :Backup
Enter field length (up to 118 ) :1
Field type :A(lphabetic N(umeric D(igit Y(es/No E(verything :Y
Enter field 7 name :Comments
Enter field length (up to 117 ) :45
Field type :A(lphabetic N(umeric D(igit Y(es/No E(verything :E
Index on which field (1-7) :8     <-Error (non-existing field)
Index on which field (1-7) :0     <-Error (too small)
Index on which field (1-7) :2

Press any key to continue


Use an existing file screen
```

No checking is done for the existence of the file in order to make the programme more portable.

```
Use an Existing Data File
-------------------------

File name :programs

Press any key to continue
```

Starting screen with file in use

Now new options have been made available. Note that other keys than those with which an option starts are ignored.

```
GCE  A Level Computer Science Project        (C) 1985,86 Diomidis D. Spinellis

                          Data Base Management System
                          ---------------------------



Available Options
-----------------

C(reate a new file
U(se an existing data file
A(dd a new entry
D(isplay an existing entry
L(ist Alphabeticaly
R(emove an existing entry
E(dit an existing entry
Q(uit from the programme



Select operation by typing the options first letter
```

Add new entry screen

A new entry is added. A check is made that the entry does not already exist.

```
Add a New Entry
---------------

Enter Title:Framework II
Company:Ashton Tate
Usage:Integrated Software
Number of disks:7
Rating:9.6
Backup:Y
Comments:An easy to use, powerfull integrated package
Record Inserted

Press any key to continue
```

Add new entry screen (with error)

Here the entry already existed.

Add a New Entry

Enter Title:Framework II
Record already exists

Press any key to continue

Add new entry screen

Add a New Entry

Enter Title:Sidekick
Company:Borland International
Usage:Memory resident add on
Number of disks:1
Rating:9.2
Backup:Y
Comments:Very usefull, a real time saver
Record Inserted

Press any key to continue

Alphabetical listing

A sorted listing of the file.

List Alphabeticaly

Company : Marketed by IBM
Title : Easy Writer
Usage : Word processing
Number of disks : 2
Rating : 6
Backup : N
Comments : A primitive word processor

Company : Ashton Tate
Title : Framework II
Usage : Integrated Software
Number of disks : 7
Rating : 9.6
Backup : Y
Comments : An easy to use, powerfull integrated package

Company : Lotus research
Title : Lotus 123 Release 2

```
Usage : Spreadsheet
Number of disks : 5
Rating : 9.3
Backup : Y
Comments : Powerfull, fast but complex

Company : Borland International
Title : Sidekick
Usage : Memory resident add on
Number of disks : 1
Rating : 9.2
Backup : Y
Comments : Very usefull, a real time saver

Press any key to continue
```

Display existing entry screen (with error)

Here I request to see an entry of a programme which does not exist. The name of the field together with the specific request are used to report the error.

```
Display an Existing Entry
-------------------------

Enter Title:Word Star
An entry with the Title Word Star could not be located.
Either you have misspeled it or it was never entered in the file.

Press any key to continue
```

Display existing entry screen (no error)

This is a normal request to display an entry.

```
Display an Existing Entry
-------------------------

Enter Title:Easy Writer

Company : Marketed by IBM
Title : Easy Writer
Usage : Word processing
Number of disks : 2
Rating : 6
Backup : N
Comments : A primitive word processor

Press any key to continue
```

Remove entry screen

Always when the message "Enter Title" appears a check is made if that entry exists. If the entry does not exist the result is the same as that demonstrated in the display entry (with error).

```
Remove an Existing Entry
------------------------

Enter Title:Sidekick
Name Deleted

Press any key to continue
```

Alphabetical listing screen

This is a listing after the entry Sidekick was removed from the list. The complex internal operations are totally invisible to the user.

```
List Alphabeticaly
------------------


Company : Marketed by IBM
Title : Easy Writer
Usage : Word processing
Number of disks : 2
Rating : 6
Backup : N
Comments : A primitive word processor

Company : Ashton Tate
Title : Framework II
Usage : Integrated Software
Number of disks : 7
Rating : 9.6
Backup : Y
Comments : An easy to use, powerfull integrated package

Company : Lotus research
Title : Lotus 123 Release 2
Usage : Spreadsheet
Number of disks : 5
Rating : 9.3
Backup : Y
Comments : Powerfull, fast but complex

Press any key to continue
```

Editing screen

When editing an entry the user is asked if he wishes to change a specific field. In the case of an affirmative answer the message disappears and the name of the field appear. Then the user acts as if he was entering the field for the first time.

```
Edit an Existing Enty
---------------------

Enter Title:Framework II
Old contents of entry :

Company : Ashton Tate
Title : Framework II
Usage : Integrated Software
Number of disks : 7
Rating : 9.6
Backup : Y
Comments : An easy to use, powerfull integrated package

Enter Title:Frameork II
Enter new Company(Y/N) ?
```

Editing screen

If the user does not wish does change a field he replies with "N". then the message again disappears and is replaced with the old contents of that field.

```
Edit an Existing Enty
---------------------

Enter Title:Framework II
Old contents of entry :

Company : Ashton Tate
Title : Framework II
Usage : Integrated Software
Number of disks : 7
Rating : 9.6
Backup : Y
Comments : An easy to use, powerfull integrated package

Enter Title:Frameork II
Company:Ashton Tate
Usage:Integrated Software
Number of disks:
```

Quit screen

This is what appears on the screen when the programme ends.

```
End of the Database Management System
You are reminded of the necessity of frequent backups of your data
```

10.3 Some testing

Some notes on the testing

Because of the complexity of the algorithms used an extensive testing of the programme was required. In order to make the file structure visible so that I could check the way a particular procedure worked a special non-documented, non-user-oriented procedure was created. This was the eXamine file procedure which displayed the contents of the index file (by far the most complex file of the system) in a manner that I could understand. Almost all the errors in the programme logic were located with the aid of this procedure. Some screens with the use of this procedure follow. Again the notes that I have added to those screens are written in **bold characters.**

Examine 1

These are the contents of the file after the sample session. The Sidekick entry is a deleted entry and is thus pointed by the left node of Rec #0. No other deleted entries exist and so the Sidekicks left node is 0. The tree in use starts from Rec #4 which is pointed by the right node of Rec #1. This entry is Lotus 123. Easy Writer has a smaller value than Lotus so it is pointed by the left node of Lotus. On the other hand because "F" follows "E" Framework is pointed by the right node of Easy Writer. All other nodes are 0 signalling that no other entries exist. The right column is composed of pointers to the data file.

REC	Key	LEFT	RIGHT	DataPoint
0		3	0	0
1		0	4	0
2	Framework II	0	0	0
3	Sidekick	0	0	128
4	Lotus 123 Release 2	5	0	256
5	Easy Writer	0	2	384

Examine 2

Here more entries have been added. Note that the deleted records linked list has been emptied (left node of Rec #0=0) and that the Norton Utilities have taken the place of the entry of the deleted Sidekick. The reader is encouraged **to examine the tree structure by plotting a tree diagram.**

REC	Key	LEFT	RIGHT	DataPoint
0		0	0	0
1		0	4	0
2	Frameork II	0	15	0
3	Norton Utilities	7	6	128
4	Lotus 123 Release 2	5	3	256
5	Easy Writer	8	2	384
6	Sargon III	16	9	512
7	Microsoft Word	12	0	640
8	Cross Talk	10	0	768
9	Symphony	0	11	896
10	AutoCAD	0	0	1024
11	TopView	0	13	1152
12	Microsoft C	18	0	1280
13	Turbo Pascal	0	14	1408
14	smARTWORK	17	0	1536
15	Hitchikers guide...	0	0	1664
16	Professional Editor	0	0	1792
17	micro Prolog	0	0	1920
18	Macro Assembler 4.0	0	0	2048

Examine 3

In the following two screens the programme Sargon III is deleted and this shows the creation of the "deleted" linked list.

REC	Key	LEFT	RIGHT	DataPoint
0		6	0	0
1		0	4	0
2	Frameork II	0	15	0
3	Norton Utilities	7	9	128
4	Lotus 123 Release 2	5	3	256
5	Easy Writer	8	2	384
6	Sargon III	0	9	512
7	Microsoft Word	12	0	640
8	Cross Talk	10	0	768
9	Symphony	16	11	896
10	AutoCAD	0	0	1024
11	TopView	0	13	1152
12	Microsoft C	18	0	1280
13	Turbo Pascal	0	14	1408
14	smARTWORK	17	0	1536

15	Hitchikers guide...	0	0	1664
16	Professional Editor	0	0	1792
17	micro Prolog	0	0	1920
18	Macro Assembler 4.0	0	0	2048

Examine 4

Here Set FX-Plus has taken the place of the deleted record #6.

REC	Key	LEFT	RIGHT	DataPoint
0		0	0	0
1		0	4	0
2	Frameork II	0	15	0
3	Norton Utilities	7	9	128
4	Lotus 123 Release 2	5	3	256
5	Easy Writer	8	2	384
6	Set FX-Plus	0	0	512
7	Microsoft Word	12	0	640
8	Cross Talk	10	0	768
9	Symphony	16	11	896
10	AutoCAD	0	0	1024
11	TopView	0	13	1152
12	Microsoft C	18	0	1280
13	Turbo Pascal	0	14	1408
14	smARTWORK	17	0	1536
15	Hitchikers guide...	0	0	1664
16	Professional Editor	0	6	1792
17	micro Prolog	0	0	1920
18	Macro Assembler 4.0	0	0	2048

11 Evaluation of computer results

11.1 Speed

The programme behaved in a fast manner without delays after an operation was requested. With larger databases small amounts of delay were observed. Although the importance of the tree balance can not be underestimated it is generally believed that the data structure used was suitable for the specific task. No evaluation was done on non disk base operations as these take a minimum amount of time. The biggest time burden was the disk access time which was pressed down to a log2(n) time increase factor for n records (assuming a perfectly balanced tree).

11.2 Space

The space overhead of the database files was divided into two parts. 1) the structure file which has a fixed length during all the life of the database and should normally not concern us. 2) the index file which grows together with the database file. In the index file 6 bytes are used for every record stored. In that figure the length of the key field should be added. For databases with a very large amount of very small records this organization is unsuitable. However with databases with a relatively large record length the index file takes up only a small amount of the total disk space occupied by the data.

11.3 Algorithm

The algorithms used can not be correctly evaluated without taking into account the size of the database to be used. For a small database (up to 40 records) the procedures used are clearly a waste of time, space, reliability and programming effort. However for larger databases (which should usually be the case as small ones do not justify the cost of being computerised in the first place) the algorithms justify the effort of designing them. Very large databases (such as a computerised police record) are also unsuitable for this programme. This is furthermore true as no provisions have been taken for record locking as would be needed in a multi-tasking environment. Generally these algorithms are very suitable for medium size databases.

12 Evaluation of user interface

12.1 Screen design

The screens are usually self explaining. The questions are asked in a precise non technical manner. However for the sake of simplicity many things that seem trivial to someone who has some experience with computers are not explained on the screen (such as the need to press the Return button after an input). Some training and documentation is necessary.

12.2 Input Validation

The input is validated in all places. Wrong inputs are not accepted at all from the keyboard (e.g. letters in the place of numbers) or when a numerical entry is entered that is wrong the user is prompted to reenter it. No error messages are usually displayed. All options that are not available for a specific combination of inputs prompt this so the possibility for an internal system error is minimised. The only kind of errors which have not been taken care of are the input-output errors such as a

faulty disk drive or even a non-selected printer. These errors should be dealt by the operating system.

12.3 Security

No security measures have been provided such as passwords or coding procedures. These measures make the user over confident of the security and can usually very easily be short circuited by an experienced computer user. It is better for the user to guard the data in the old fashioned way on which he has a full control he can easily understand the security that is provided. E.g. one can lock the diskettes in a drawer or even in a save.

The backup of the data is left to the operating system but the user is prompted to its necessity after each programme session.

12.4 Documentation

Three levels of documentation would be needed for this programme.

1) Technical documentation aimed to someone who should have to maintain the programme. This should include a listing of the programme with remarks and notes on the procedures and data structures used. Flow charts and pseudo-code tables should be provided. This report has some of this information.

2) User documentation which should be used by someone who would set up the database at a specific site. It should include some technical information as it is assumed that this type of user has some experience and can solve some elementary problems that could arise. (He could be a member of the EDP department of a company).

3) End user documentation. This is provided for the person which will ultimately use the programme and should be very detailed including annotated keyboard diagrams and sample screens. This should also include a tutorial manual. In an ideal situation this documentation should make the documentation (2) obsolete.

13 Evaluation of the project

13.1 General

This project gave me the opportunity to solve an information processing problem of a some complexity and to understand the interaction of the various aspects of the problem. The testing of the programme which consumed more than 60% of the time I spend on this project was a valuable exercise. Even more interesting was the writing of the report which was a summation of all the experience I had gained during the creation of the programme.

13.2 Further enhancements

Many are the enhancements that could be introduced into a project of this kind. One major problem with binary trees is that of the tree balancing. In a balanced tree all nodes on the tree frontier differ at most by one level. The shape of the tree is determined by the way new entries arrive and by the other operations that are performed on it, e.g. deletions. In a worst case situation where all entries arrive in a sorted manner the tree search will behave exactly like a linear search. Various methods exist for balancing a tree. Most of the require reordering the tree structure by a series of operations performed at various levels. I feel that such a procedure is beyond the scope of this project.

The input interface could have been expanded to allow input from other programmes (importing). This would allow the database to accept data that would be outputed from the redirection of the output of another programme. Provisions would be needed for the parsing of the record to fields and for the inclusion of various delimiters that are implemented in other systems like the basic comma and quotes delimiters.

A more effective way to cope with the deletion of entries would be to implement a file crunch procedure. In that way the length of the file would always reflect the actual contents of it and not the contents of the deleted entries. However the shortening of a file is an issue particularly unpopular in all operating systems. Other operating systems do not implement it at all and require the file to be copied in another and the old one deleted whereas others need a special procedure to compact the disk space after the alternation of a file length.

Another improvement of the programme would be to use a more modern tree structure like B+trees or AVL trees. These structures overcome many deficiencies of the binary tree file structure but impose other problems such as more complex algorithms or wasted disk space. I believe that for this case the binary tree file structure is a fair compromise.

14 Bibliography

acm computing surveys Volume 6 Number 3 September 1974
J. Nievergelt Binary Search Trees and File Organization
Brian W. Kernighan, Dennis M. Ritchie
The C Programming Language, Prentice Hall Software Series
Geoff Vincent, Jim Gill
Software Development Handbook, Texas Instruments
UCSD p-System Introduction
Regents of the University of California and SofTech Microsystems
UCSD p-System Operating System Reference Manual
Regents of the University of California and SofTech Microsystems
UCSD p-System Configuration
Regents of the University of California and SofTech Microsystems
personal computing with UCSD p-System
Regents of the University of California and SofTech Microsystems
The Scope for Automatic Data Processing in the British Library
Department of education and science
London Her Majesty’s Stationary Office 1972

15 Appendix A (Programme)

```
PROGRAM dbase(INPUT,OUTPUT);

{

G.C.E. A Level Computing Science (105) Paper 3

Athens College (GCE School) Athens Greece

Centre Number    : 92060

Title of Project : Data Base Management System
                   --------------------------

Programmer       : Diomidis D. Spinellis
```

Purpose and scope of the project : This system allows the user to define and use a custom designed database. The following features are available : Field naming, record indexing, insertion, deletion, editing, display, search and sorted listing of records. This project demonstrates the use of the binary tree file indexing method used for speed and efficiency.

Database file structure :
The DataBase consists of the following files :

1. Structure File (.STR)
   It is file including the following data :
   No of fields (integer)
   Field 1 Name (String)
   Field 1 length (Integer)
   Field 1 type (Character) (Validation type as explained in ValidRead)
   Field 2 Name
   Field 2 le.....
   .....
   .....
   Field n type
   Field on which the Database is indexed (integer)
2. Index File (.NDX)
   It consists of entries of type IndexFileRec which point to the .DBF file
   The first two records are not normal records.
   Record 0 points to a linked list (by the left node) of the deleted items.
   Record 1 is a dummy record which serves as the root of the tree. It can not be deleted and thus the search always begins from it.
3. Database file (.DBF)
   It contains all the records in a packed form.

Limits :
Max number of fields : 20 fields
Max field name length : 30 Characters
Max record length : 200 bytes
Max Key length : 30 Characters

```
}
{-------------------------------------------------------------------------}
const
  KeyLen=20;
  MaxFieldNumber=20;
  MaxFieldNameLength=30;
  MaxRecordLength=200;
  MaxFileNameLength=80;
  MaxStringLength=255;

{-------------------------------------------------------------------------}
type
  KeyType=String[KeyLen];
  FileNameType = string[MaxFileNameLength];
  FieldNameType = string[MaxFieldNameLength];
  CompareResult = (Greater,Less,Equal);
  MaxString = string[255];

  IndexFileRec=Record
    LeftBranch  : Integer;
    RightBranch : Integer;
    Key         : KeyType;
    DataPoint   : Integer;
  End;

  FieldInfoType = Record
    Name        : string[MaxFieldNameLength];
    Length      : Integer;
    Validate    : Char;
  end;

  FileStrucType = Record
    FieldNum,IndexField,RecordLength : Integer;
    FieldInfo : array[1..MaxFieldNumber] of FieldInfoType;
  end;

  RecordString = string[MaxRecordLength];


{-------------------------------------------------------------------------}
Var
  IndexFile     : FILE OF IndexFileRec;
  DatabaseFile  : File of Char;
  StructureFile : Text;
  Buffer_A,Buffer_b,Buffer_C  : IndexFileRec;
  FileStruc     : FileStrucType;
  Root          : integer; {Root is the local Variable changed by Search}
  CODE,I        : integer;
```

```
  Name          : KeyType;
  C,CC          : Char;
  PaddString,OldDataPack : RecordString;
  Dummy         : Boolean; {Used for function calls that return boolean type}
  Edit          : Boolean;{If set the ReadItem proc will allow default entries}
  FileInUse     : Boolean;

{-------------------------------------------------------------------------}

function ValidRead( ValidType : Char ; StringLength : Integer ) : MaxString ;
{Read a string with input length and type validation}
Const
  Return = 13;
  BackSpace = 8 ;
  Bell = 7 ;
Var
  c : Char;
  i : Integer;
  Result : MaxString;

  Function Valid(C : Char):Boolean;
  Var
    Result : Boolean;
  begin
    Result:=False;
    case ValidType of
      'A' : If c in ['A'..'Z','a'..'z'] then Result:=True; {Alpha}
      'N' : If c in ['0'..'9','+','-','#','.',','] then Result:=True; {Numeric}
      'D' : If c in ['0'..'9'] then Result:=True; {Digit}
      'Y' : If c in ['Y','N','y','n','T','t','F','f'] then Result:=True;{YesNo}
      'E' : If c in [' '..'~'] then Result:=True; {All the printable ASCII set}
    end;
    Valid:=Result;
  end;

begin
  i:=0;
  Result:='';
  repeat
    Read(Kbd,C);
    If C=Chr(BackSpace) then
      if i>0 then
      begin
        Result:=Copy(Result,1,Length(Result)-1);
        Write(Chr(BackSpace),' ',Chr(BackSpace));
        i:=i-1;
      end
      else
        Write(Chr(Bell))
    else
      If Valid(c) and (i<StringLength) then
      begin
        Result:=Concat(Result,c);
        Write(c);
```

```
        i:=i+1;
      end
      else
        if not (c=Chr(Return)) then
          Write(chr(Bell));
  Until C=Chr(Return) ;
  ValidRead:=Result;
  Writeln('');
end;

{-------------------------------------------------------------------------------}
procedure Prompt(Name : MaxString);
{Will initialize the screen for the operation named in name}
begin
  ClrScr;
  WriteLn(Name);
  WriteLn(Copy('-----------------------------------------',1,Length(Name)));
  WriteLn('');
end;

{-------------------------------------------------------------------------------}
function Upper(C : Char) : Char;
{Make C uppercase if required}
var
  cc : char;
begin
  if (C>='a') and (C<='z') then
    cc:=chr(Ord(c)-Ord('a')+Ord('A'))
  else
    cc:=c;
  Upper:=cc;
end;

{-------------------------------------------------------------------------------}
function Compare(A,B : KeyType) : CompareResult;
{Compare to entries of the index file acording to ASCII colating sequence and
return Greater Less or Equal}
begin
  if A>B then
    Compare:=Greater
  else if A<B then
    Compare:=Less
  else
    Compare:=Equal;
end;

{-------------------------------------------------------------------------------}
procedure IndexRead(Where : Integer ; var What : IndexFileRec);
{Read an entry from position Where in the index file into What}
begin
  Seek(IndexFile,Where);
  Read(IndexFile,What);
end;
```

```
{-------------------------------------------------------------------------}
procedure IndexWrite(Where : Integer ; What : IndexFileRec);
{Write an entry to position Where in the index file from What}
begin
  Seek(IndexFile,Where);
  Write(IndexFile,What);
end;

{-------------------------------------------------------------------------}
procedure InitVars;
{Initialize global variables}
var
  i : Integer;
begin
  PaddString:='';
  for i:=1 to MaxRecordLength do
    PaddString:=ConCat(PaddString,' ');
  Edit:=False; {Only the edit function sets edit to true}
  FileInUse:=False;
end;

{-------------------------------------------------------------------------}
procedure OpenFiles;
{Open all database files}
Var
  FileName,StructureName,DatabaseName,IndexName : FileNameType;

Begin
  Write('File name :');
  FileName:=ValidRead('E',MaxFileNameLength);
  StructureName:=ConCat(FileName,'.STR');
  DatabaseName:=ConCat(FileName,'.DBF');
  IndexName:=ConCat(FileName,'.NDX');
  Assign(StructureFile,StructureName);
  Assign(DatabaseFile,DatabaseName);
  Assign(IndexFile,IndexName);
end;

{-------------------------------------------------------------------------}
Procedure CloseFiles;
{Close all Database files}
begin
  Close(IndexFile);
  Close(DatabaseFile);
  Close(StructureFile);
  FileInUse:=False;
end;

{-------------------------------------------------------------------------}
Procedure PrepareNewFile;
{Initializes all Database files}

Procedure PrepareStructureFile;
```

```
Var
  IndexField,I,FieldNumber,FieldLength,RecordLength : Integer;
  FieldName : FieldNameType;
  FieldType : Char;
  NumberInStringForm : String[20];
  EvalResult : Integer;

begin
  Repeat
    Write('Number of fields (1-',MaxFieldNumber,') :');
    NumberInStringForm:=ValidRead('D',5);
    Val(NumberInStringForm,FieldNumber,EvalResult);
  until (FieldNumber<=MaxFieldNumber) and (FieldNumber>0) ;
  WriteLn(StructureFile,FieldNumber);
  RecordLength:=0;
  For i:=1 to FieldNumber Do
  begin
    Write('Enter field ',i,' name :');
    FieldName:=ValidRead('E',MaxFieldNameLength);
    repeat
      Write('Enter field length (up to ',MaxRecordLength-RecordLength,' ) :');
      NumberInStringForm:=ValidRead('D',5);
      Val(NumberInStringForm,FieldLength,EvalResult);
    until RecordLength+FieldLength<MaxRecordLength;
    Write('Field type :A(lphabetic N(umeric D(igit Y(es/No E(verything :');
    repeat
      Read(Kbd,FieldType);
      FieldType:=Upper(FieldType);
    until FieldType in ['A','N','D','Y','E'];
    Writeln(FieldType);
    RecordLength:=RecordLength+FieldLength;
    WriteLn(StructureFile,FieldName);
    WriteLn(StructureFile,FieldLength);
    WriteLn(StructureFile,FieldType);
  end;
  repeat
    Write('Index on which field (1-',FieldNumber,') :');
    NumberInStringForm:=ValidRead('D',5);
    Val(NumberInStringForm,IndexField,EvalResult);
  until (IndexField>0) and (IndexField<=FieldNumber);
  Writeln(StructureFile,IndexField);
end;

Procedure PrepareIndexFile;
Var
  I:integer;
Begin
  Buffer_A.LeftBranch:=0;
  Buffer_A.RightBranch:=0;
  For I:=1 TO KeyLen Do
    Buffer_A.Key[i]:=chr(0);
    Buffer_A.DataPoint:=0;
  Write(IndexFile,Buffer_A);
  IndexWrite(1,Buffer_A);
```

```
End;


begin
  OpenFiles;
  Rewrite(StructureFile);
  Rewrite(DatabaseFile);
  Rewrite(IndexFile);
  PrepareStructureFile;
  PrepareIndexFile;
  CloseFiles;
end;

{-------------------------------------------------------------------------------}
Procedure UseFile;
{Prepares the program to use an old data file}
begin
  OpenFiles;
  Reset(StructureFile);
  Reset(DatabaseFile);
  Reset(IndexFile);
  ReadLn(StructureFile,FileStruc.FieldNum);
  FileStruc.RecordLength:=0;
  For i:=1 to FileStruc.FieldNum Do
  begin
    Readln(StructureFile,FileStruc.FieldInfo[i].Name);
    Readln(StructureFile,FileStruc.FieldInfo[i].Length);
    Readln(StructureFile,FileStruc.FieldInfo[i].Validate);
    FileStruc.RecordLength:=FileStruc.RecordLength       +
                            FileStruc.FieldInfo[i].Length
  end;
  Readln(StructureFile,FileStruc.IndexField);
  FileInUse:=True;
end;

{-------------------------------------------------------------------------------}
procedure DataRead(ItemNum : Integer ; var DataPack : RecordString);
{Read data from position ItemNum in the database file into DataPack}
var
  i : integer;
  c : Char;
begin
  seek(DatabaseFile,ItemNum);
  DataPack:='';
  For i:=1 to FileStruc.RecordLength do
  begin
    Read(DatabaseFile,C);
    DataPack:=Concat(DataPack,C);
  end;
end;

{-------------------------------------------------------------------------------}
Procedure DisplayItem(ItemNum : Integer);
{Display the contents of the item located in postion ItemNum in DBF file}
```

```
Var
  DataPack : RecordString ;
  i,DataPackPos : integer;
begin
  WriteLn('');
  DataRead(ItemNum,DataPack);
  DataPackPos:=1;
  For i:=1 to FileStruc.FieldNum do
  begin
    Write(FileStruc.FieldInfo[i].Name,' : ');
    WriteLn(Copy(DataPack,DataPackPos,FileStruc.FieldInfo[i].Length));
    DataPackPos:=DataPackPos+FileStruc.FieldInfo[i].Length;
  end;
end;

{-------------------------------------------------------------------------}
Function KeyName : KeyType;
{Returns the name of the field onto which the database is indexed}
begin
  KeyName:=FileStruc.FieldInfo[FileStruc.IndexField].Name;
end;

{-------------------------------------------------------------------------}
Procedure NotFoundError(Name : KeyType);
{Reports an error if entry with key name Name was no found}
begin
  WriteLn('An entry with the ',Keyname,' ',Name,' could not be located.');
  WriteLn('Either you have misspeled it or it was never entered in the file.');
end;

{-------------------------------------------------------------------------}
function ReadKeyName : KeyType ;
{Returns the name of the record the user wants to act uppon}
Var
  Key : KeyType;
begin
  Write('Enter ',KeyName,':');
  Key:=ValidRead('E',KeyLen);
  ReadKeyName:=Key;
end;

{-------------------------------------------------------------------------}
Procedure ReadItem(ItemNum : Integer ; Key : KeyType);
{Reads all the contents of a record. If Edit is true allows default responses
acording to the variable OldDataPack. ItemNum points to the database file}
Var
  DataPack,DataRead : RecordString;
  i,DataPackPos : integer;
  c : Char;

function Padd(What : RecordString) : RecordString;
  begin
    Padd:=Copy(Concat(What,PaddString),1,MaxRecordLength);
  end;
```

```
function Min(a,b : Integer ) : Integer;
begin
  if a>b then Min:=b else Min:=a;
end;

begin
  seek(DatabaseFile,ItemNum);
  DataPack:='';
  For i:=1 to FileStruc.FieldNum do
  begin
    If i=FileStruc.IndexField Then
      DataRead:=Key
    else
      If Edit then
      begin
        Write('Enter new ',FileStruc.FieldInfo[i].Name,'(Y/N) ?');
        Repeat
          Read(Kbd,C);
        until c in ['Y','y','N','n'] ;
        DelLine;
        GotoXY(1,WhereY);
        If (c='N') or (c='n') then
        begin
          DataRead:=Copy(OldDataPack,Length(DataPack)+1,
                         FileStruc.FieldInfo[i].Length);
          Write(FileStruc.FieldInfo[i].Name,':');
          WriteLn(DataRead);
        end {No Change}
        else
        begin
          Write(FileStruc.FieldInfo[i].Name,':');
          DataRead:=ValidRead(FileStruc.FieldInfo[i].Validate,
                              FileStruc.FieldInfo[i].Length);
        end; {Change}
      end {If edit}
      else {No edit}
      begin
        Write(FileStruc.FieldInfo[i].Name,':');
        DataRead:=ValidRead(FileStruc.FieldInfo[i].Validate,
                            FileStruc.FieldInfo[i].Length);
      end; {No edit}
    DataPack:=ConCat(DataPack,Copy(Padd(DataRead),1,
                     FileStruc.FieldInfo[i].Length));
  end; {for i}
  For i:=1 to FileStruc.RecordLength do
    Write(DatabaseFile,DataPack[i]);
end;

{-------------------------------------------------------------------------}
Procedure List(Node:integer);
{Goes throught the tree structure in the collating sequence manner}
Var
  Buffer : IndexFileRec;
```

```
Begin
  IndexRead(Node,Buffer);
  If Buffer.LeftBranch<>0 Then
    List(Buffer.LeftBranch);
  DisplayItem(Buffer.DataPoint);
  If Buffer.RightBranch<>0 Then
    List(Buffer.RightBranch);
End;

{-------------------------------------------------------------------------}
Function Search(Key:KeyType):integer;
{Return the Position of Key in file, 0 If not existing.
Also set the global Variable Root to the Root of the record or where the
nonexisting record should be hanged.}

Var
  Pos:integer;
  Found,StillOthers:Boolean;

Begin
  Found:=False;
  StillOthers:=True;
  Pos:=1;
  Root:=0;
  While StillOthers and not Found Do
  Begin
    IndexRead(Pos,Buffer_A);
    case Compare(Buffer_A.Key,Key) of
      Equal :
      Begin
        Found:=True;
        Search:=Pos;
      End;

      Greater :
      Begin
        Root:=Pos;
        If Buffer_A.LeftBranch<>0 Then
          Pos:=Buffer_A.LeftBranch
        Else
        Begin
          StillOthers:=False;
          Search:=0;
        End;
      End;

      Less :
      Begin
        Root:=Pos;
        If Buffer_A.RightBranch<>0 Then
          Pos:=Buffer_A.RightBranch
        Else
        Begin
          StillOthers:=False;
```

```
          Search:=0;
        End; {If}
      End; {Less}
    End; {Case}
  End; {While}
End; {Function}

{-------------------------------------------------------------------------}
Function Insert:Boolean;
{Inserts a record into the file structure}
Var
  InsertPosition,IndexPos,DataPosition : integer;
  Rec : IndexFileRec;
  Key : KeyType;

Procedure FindInsertPosition ;
{Sets InsertPosition to the position to insert a record and updates the
deleted linked list if needed .Buffer_B is destroyed . Also the DataPosition
for the database file is set }
  Var
    NewListPointer : Integer;
  Begin
    IndexRead(0,Buffer_B);
    if Buffer_B.LeftBranch=0 Then
    begin
      InsertPosition:=FileSize(IndexFile);
      DataPosition:=FileSize(DatabaseFile)
    end
    Else
    Begin
      InsertPosition:=Buffer_B.LeftBranch;
      IndexRead(InsertPosition,Buffer_B);
      DataPosition:=Buffer_B.DataPoint;
      NewListPointer:=Buffer_B.LeftBranch;
      IndexRead(0,Buffer_B);
      Buffer_B.LeftBranch:=NewListPointer;
      IndexWrite(0,Buffer_B);
    end;
  end;


Begin
  Key:=ReadKeyName;
  IndexPos:=Search(Key);
  If IndexPos=0 Then
  Begin
    FindInsertPosition;
    ReadItem(DataPosition,Key);
    IndexRead(Root,Buffer_A);
    Rec.LeftBranch:=0;
    Rec.RightBranch:=0;
    Rec.DataPoint:=DataPosition;
    Rec.Key:=Key;
    case Compare(Buffer_a.Key,Rec.Key) of
```

```
      Greater :
        Buffer_A.LeftBranch:=InsertPosition;
      Less :
        Buffer_A.RightBranch:=InsertPosition;
    end;
    IndexWrite(Root,Buffer_A);
    IndexWrite(InsertPosition,Rec);
    Insert:=True;
    End
  Else
    Insert:=False;
End;

{-------------------------------------------------------------------------}
Function Delete(Key:KeyType):Boolean;
{Removes the record Key from the tree structure and returns True if it exists.
The record is also appended to the deleted records linked list}
Var
  DelPos,HangPos,Pos,Left :integer;
  Dummy :Boolean;

Procedure UpdateDeletedList;
  Begin
    IndexRead(0,Buffer_B);
    IndexRead(DelPos,Buffer_A);
    {Check to see if already done (due to recursivness it is called twice)}
    If not (Buffer_B.LeftBranch=DelPos) then
    begin
      {Make the deleted record point to the old deleted record}
      Buffer_A.LeftBranch:=Buffer_B.LeftBranch;
      {Make the start of the linked list point to the deleted record}
      Buffer_B.LeftBranch:=DelPos;
      IndexWrite(DelPos,Buffer_A);
      IndexWrite(0,Buffer_B);
    end;
  end;

Begin
  DelPos:=Search(Key);
  If not (DelPos=0) Then
  Begin
    IndexRead(DelPos,Buffer_a);
    If (Buffer_A.LeftBranch=0) OR (Buffer_A.RightBranch=0) Then
    Begin
      IndexRead(Root,Buffer_B);
      HangPos:=Buffer_A.LeftBranch+Buffer_A.RightBranch; {1 or 2 of them are 0}
      Case Compare(Buffer_B.Key,Key) of
        Greater : Buffer_B.LeftBranch:=HangPos;
        Less    : Buffer_B.RightBranch:=HangPos;
      end;
      IndexWrite(Root,Buffer_B);
      Delete:=True;
    End
    Else
```

```
    Begin
      { save the left pointer and make it zero call Delete recursively
        hang the node pointed by the left pointer where apropriate  }
      Left:=Buffer_A.LeftBranch;
      Buffer_A.LeftBranch:=0;
      IndexWrite(DelPos,Buffer_A);
      Dummy:=Delete(Buffer_A.Key);
      IndexRead(Left,Buffer_A);
      Pos:=Search(Buffer_A.Key);
      {Pos should be zero because this node is now disconnected}
      IndexRead(Left,Buffer_A);
      IndexRead(Root,Buffer_B); {Where this node should be hanged}
      Case Compare(Buffer_B.Key,Buffer_A.Key) of
        Greater : Buffer_B.LeftBranch:=Left;
        Less    : Buffer_B.RightBranch:=Left;
      end;
      IndexWrite(Root,Buffer_B);
      Delete:=True;
    End;
    UpdateDeletedList;
  End
  Else
    Delete:=False;
End;

{-------------------------------------------------------------------------}
procedure OptionsScreen;
{Prompt the user the available options he can perform}
begin
  ClrScr;
  WriteLn('GCE  A Level Computer Science Project        (C) 1985,86 Diomidis D. Spin
  WriteLn('');
  WriteLn('                              Data Base Management System');
  WriteLn('                              ---------------------------');
  WriteLn('');
  WriteLn('');
  WriteLn('');
  WriteLn('Available Options');
  WriteLn('-----------------');
  WriteLn('');
  WriteLn('C(reate a new file');
  WriteLn('U(se an existing data file');
  If FileInUse then
  begin
    WriteLn('A(dd a new entry');
    WriteLn('D(isplay an existing entry');
    WriteLn('L(ist Alphabeticaly');
    WriteLn('R(emove an existing entry');
    WriteLn('E(dit an existing entry');
  end;
  WriteLn('Q(uit from the programme');
  WriteLn('');
  WriteLn('');
  WriteLn('');
```

```
  Write('Select operation by typing the options first letter ');
end;

{=============================================================================}
Begin
  InitVars;
  repeat
    OptionsScreen;
    repeat
      read(kbd,c);
      c:=Upper(c);
    until ( (c in ['C','U','A','D','L','R','E','Q']) and FileInUse) or
          (c in ['C','U','Q']);
    WriteLn('');
    Case c of

      'Q' : {Quit}
      begin
        Prompt('Quit');
      end;

      'C' : {Create new file}
      Begin
        Prompt('Create a New File');
        If FileInUse then CloseFiles;
        PrepareNewFile;
      End;

      'A' : {Add new entry}
      Begin
        Prompt('Add a New Entry');
        If Insert Then
          WriteLn('Record Inserted')
        Else
          WriteLn('Record already exists');
      End;

      'D' : {Display}
      Begin
        Prompt('Display an Existing Entry');
        Name:=ReadKeyName;
        I:=Search(Name);
        If (I<>0) Then
        Begin
          IndexRead(I,Buffer_A);
          DisplayItem(Buffer_A.DataPoint);
        End
        Else
          NotFoundError(Name);
      End;

      'L' : {list}
      Begin
        Prompt('List Alphabeticaly');
```

```
      IndexRead(1,Buffer_A);
      If not (Buffer_A.RightBranch=0) then List(Buffer_A.RightBranch);
    End;

    'R' :       {Remove}
    Begin
      Prompt('Remove an Existing Entry');
      Name:=ReadKeyName;
      If Delete(Name) Then WriteLn('Name Deleted') Else NotFoundError(Name);
    End;

    'E' : {Edit}
    Begin
      Prompt('Edit an Existing Enty');
      Name:=ReadKeyName;
      I:=Search(Name);
      If (I<>0) Then
      Begin
        IndexRead(I,Buffer_A);
        DataRead(Buffer_A.DataPoint,OldDataPack);
        WriteLn('Old contents of entry :');
        DisplayItem(Buffer_a.DataPoint);
        WriteLn('');
        Dummy:=Delete(Name);
        Edit:=True; {Allow default responses based on OldDataPack}
        Dummy:=Insert;
        Edit:=False; {Disable default responses for all other uses}
      End
      Else
        NotFoundError(Name);
    End;

    'U' : {Use an old data file}
    begin
      Prompt('Use an Existing Data File');
      If FileInUse then CloseFiles;
      UseFile;
    end;

{
Non documented feature is commented out for the official release. Was
extensively used as an examine file utility during programme development
and debuging.
      'X' : eXamine
      Begin
        RESET(IndexFile);
        I:=0;
        WriteLn('  REC          Key          LEFT RIGHT             DataPoint');
        WriteLn('');
        While NOT(EOF(IndexFile)) Do
        Begin
          Read(IndexFile,Buffer_A);
          WriteLn(I:5,Buffer_A.Key:KeyLen,Buffer_A.LeftBranch:5,
          Buffer_A.RightBranch:5,'     ',Buffer_A.DataPoint:5);
```

```
          I:=I+1;
        End
      End;
}

    End;
    WriteLn('');
    Write('Press any key to continue ') ;
    Read(Kbd,CC) ;
    WriteLn('');
  until C='Q' ;
  If FileInUse then CloseFiles;
  ClrScr;
  WriteLn('End of the Database Management System');
  WriteLn('You are reminded of the necessity of frequent backups of your data');
  WriteLn('');
End.
```

16 Appendix B (Technical details)

This report was written using an IBM PC personal computer and the Framework integrated software package. An outline was first constructed and then all the part of it were filled. When I thought of a new aspect that should be covered I just opened a new frame on the corresponding outline entry. The spelling checker together with a computing dictionary were used to verify the spelling. The view page pagination always kept me informed of the amount I had written.

All the frames were used with the word justify feature on and with a paragraph indent of 6 characters.

17 Appendix C (Trademarks)

MS DOS is a registered trademark of Microsoft Corp.
IBM is a registered trademark of International Business Machines Corp.
UNIX is a trademark of AT&T Bell Laboratories
Apple and **Apple II** are registered trademarks of Apple Computer, Inc.
VMS is a trademark of Digital Equipment Corp.
Framework II and **Fred** are trademarks of Ashton-Tate
UCSD, UCSD Pascal and **UCSD p-System** are all trademarks of the Regents of the University of California.
Turbo Pascal and **Sidekick** are trademarks of Borland Inc.

Other names of products which could be trademarks may have been used in the text. They have been used for reference purposes only.