1982

SLOP - An interactive library database system

Michael P. Shepanski
University of Wollongong

SLOP – AN INTERACTIVE LIBRARY DATABASE SYSTEM

Michael P. Shepanski

Department of Computing Science
University of Wollongong

Preprint No. 82-12

May 26, 1982
Reprinted 9.4.84

P.O. Box 1144, WOLLONGONG N.S.W. 2500, AUSTRALIA
tel (042)-270-859
telex AA29022

THE UNIVERSITY OF WOLLONGONG

DEPARTMENT OF COMPUTING SCIENCE

DEPARTMENTAL NOTES AND PREPRINTS

SLOP – AN INTERACTIVE LIBRARY DATABASE SYSTEM

Michael P. SHEPANSKI

Department of Computing Science
University of Wollongong

Abstract

SLOP is a modular tree-based database system for library management, featuring fast arbitrary data retrieval. Its module structure is briefly discussed together with a superficial review of the support modules, followed by an explanation of the data structures used. Finally the internal control structure of the central module is examined. A knowledge of user-level features of SLOP is assumed.

Keywords: Databases, Inventory Systems, Trees, Optimal Binary Search Trees, Modular Programming.

Preprint No 82-12 May 26, 1982

P.O. Box 1144, WOLLONGONG, N.S.W. AUSTRALIA
telephone (042)-282-981
telex AA29022

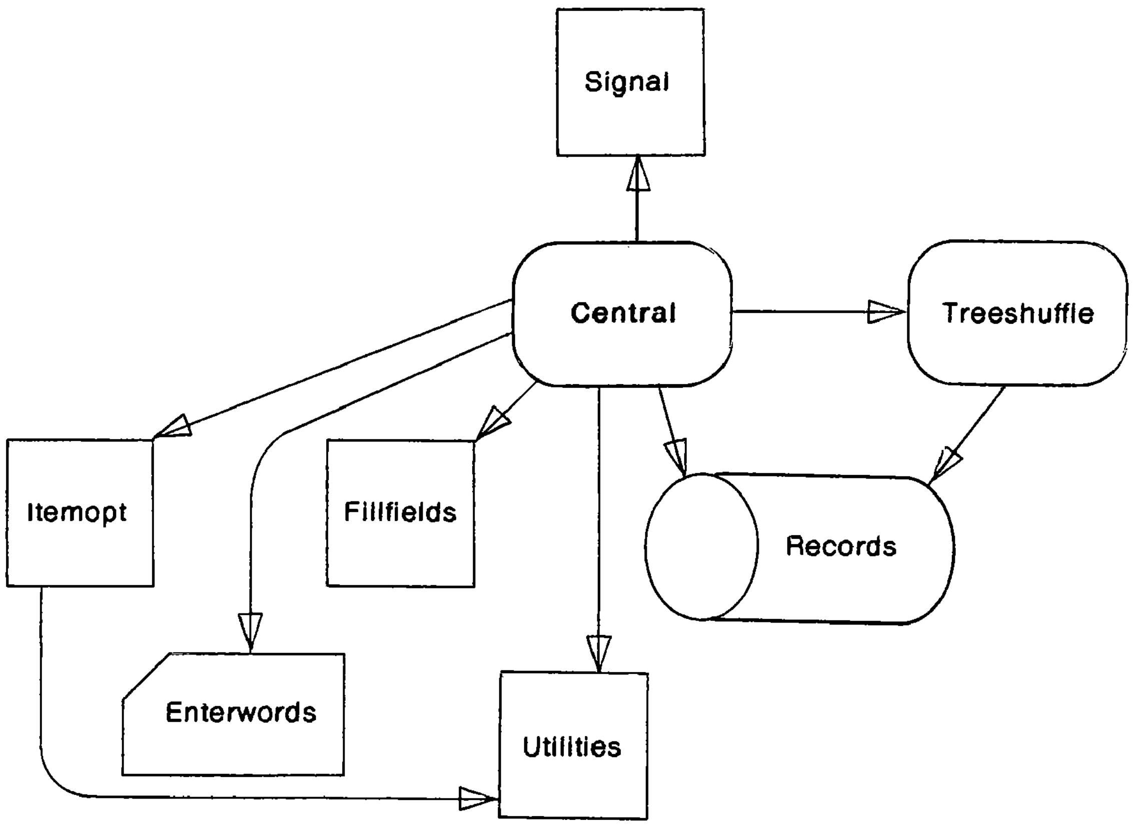
SLOP – An Interactive Library Database System.

Michael Shepanski

Computing Science Dept.,
University of Wollongong
Northfields Ave,
Wollongong
N.S.W. 2500
Australia.

1. SUPPORT MODULES

SLOP is implemented as a series of modules. The calling hierachy of the module structure is as follows :



The majority of processing is done in the **Central** module, which contains some 24 functions and most of the code. Another module, **Treeshuffle**, controls all the database maintenance which is done in the background when the user has quit execution. However, all of the keyboard operations, file access and signal handling are done elsewhere.

The lower-level routines (**Utilities**, **Records**, **Signal**, **Enterwords**, **Itemopt** and **Fillfields**) will be superficially examined first, since they provide the abstractions which are needed for *Central* and *Treeshuffle*.

1.1. Utilities

This module provides a set of low-level convenience functions for service to other routines. These include the comparing of two strings (as used for keying comparisons), conversion from lowercase to uppercase, removing unwanted spaces in a string, determining whether a character is a vowel, and searching tables.

NOTE: there is a type of table commonly used throughout SLOP, called ` **struct types []** '. This has the form of a list of character-strings, with a corresponding *char* for each one. It is useful when the program is to converse with the user in strings, but use single bytes for internal processing. This is done for space-saving (e.g. the Class field is stored on the database as a single character), or for use in the *switch* construct (e.g. Choosing the appropriate action after a command has been read), or for other reasons. The two functions

search (table, string)
unsearch (table, character)

do the lookups in *table* to return the character corresponding to *string* [search], or the string corresponding to *character* [unsearch].

1.2. Records

This module controls all access to the database file. It is an interface between the record-oriented view required by the rest of the SLOP system, and the UNIX* primitives (*creat, open, lseek, read, write, fstat*). It presents the rest of the program with a header node, followed by a sequence of records numbered 1, 2, 3, etc.

It is initialized by the call

dbinit ()

which tries to open the database file according to the global filename string **dbfile**. If access is denied, or if the file is locked, a message will be printed to the standard error output and the program will terminate execution. However, if the global variable **override** is *SIGNALLED* (a constant equal to 255), the locking mechanism is overlooked, and the database is opened if at all possible.

Access to the file header is provided through the functions:

getnulrec (structptr)
putnulrec (structptr)

These functions retrieve (or store) a structure of type **DBhead** in location *structptr* in core.

Similarly, access to the data nodes is provided through the functions:

getdbrec (recordno, structptr)
pudbrec (recordno, structptr)

These operate similarly with structures of type **DBrecord**, where *recordno* is the ordinal number of the record, and *structptr* is a pointer to the required structure. Note that these return *recordno* when successful, and the constant NULL when not.

The number of records in the database may also be found, with the function

*UNIX is a Trademark of Bell Laboratories.

filesize ()

which returns the total number of records (including the header).

1.3. Signal

This implementation-dependent module ensures that all likely interrupting signals are handled properly. The required actions for these signals are as follows:

SIGHUP	Quit as fast as possible (while keeping database intact).
SIGINT	Branch to appropriate part of Central module to prompt, read and execute the next command.
SIGQUIT	Cause a core dump immediately, then quit as fast as possible (while keeping database intact).
SIGTERM	Quit as fast as possible (while keeping database intact).
SIGPIPE	Reassign the standard output to a data sink (i.e. `/dev/null') and the standard error output to the terminal.

Signals due to program errors (e.g. arithmetic exceptions, memory segmentation violations etc.) are not caught or ignored.

Because SIGINT causes a branch into *Central* it is necessary, at that point, to have the system function

setjmp (environment)

where *environment* is a global datum (used by *Signal*) of type *jmp_buf**

This signal-catching is initialized by the call

siginit ()

which sets up the initial trapping functions.

Of course, it is necessary for some processing to proceed unhindered. This is done with the call:

disablesigs ()

From this call, until further notice, there will be no interruptions to program execution. SIGQUIT will still make a core dump immediately and SIGPIPE will still readjust the standard outgoing streams, but these effects will be transparent as far as the central procedure is concerned.

To re-enable the catching of signals, the call

enablesigs ()

is used. This checks whether any signals have occurred during the period of disablement and, if so, takes the appropriate action. This means either jumping to the `set_jmp' call or quickly tidying up the database and quitting execution.

Since all execution *should* begin and end in the central module, the actual work of clearing up the database and exiting is done by a function called **dienow** in *Central* (see below). This is not to be confused with the work of *Treeshuffle* which does a complete overhaul of the database.

*See *setjmp* (3), UNIX Programmer's Manual.

1.4. Enterwords

The *Enterwords* module interfaces to and controls the keyboard, analogously to the way *Records* interfaces to the database file. All keyboard data entry is done through *Enterwords* which uses the UNIX standard I/O library function *gets* and buffers the output to provide a word-oriented data entry facility.

The most important function supplied is

readitm (stringptr)

which fills the string beginning at *stringptr* with one input item from the keyboard. An input item is defined as EITHER a string of non-whitespace input characters, wherein an underscore ('_') is interpreted as a space by *readitm*, OR any sequence of items between matching quotes (i.e. either between two apostrophes or between two double-quotes). In any case, an item can never include the newline character, and any amount of blank lines will be ignored until *readitm* finds an item to supply.

The auxiliary function

readln (\Istringptr\I)

fills the string with the whole input line, regardless of how much has already been read. This is useful when no interpretation at all is required, e.g. shell-escape commands.

If it is necessary to ensure that a new line of input is being read, the function

newln ()

may be used. It throws away any text which has been buffered from the last input line and guarantees that the any text provided by the *Enterwords* module will be from a new one.

The other auxiliary function

eoln ()

returns a true value if there is no more text available on the current line, and false (0) if there is some.

Note that end-of-file termination (cntrl-D) is caught in the *Enterwords* module and, since this requires the same handling as the 'quit' command, a call (non-returning) is made to the *Central* function **quit** (), when this happens.

1.5. Itemopt

Most of the commands available to the SLOP user are of the form:

command [[by] fieldname <possibly something else>]

Central is only ever interested in the numeric code of the field, not *fieldname*. This numeric code is a number in the range

0 <= fieldno < NKEYFIELDS

where NKEYFIELDS is a constant (currently 7). The *Itemopt* module provides an interface between this *Central* view, and the text-oriented view supplied by *Enterwords*.

Once the central module has decided which command it is processing, the function

itemopt ()

will examine the user's input, and return a single-byte field number corresponding to *fieldname* ("title" default).

1.6. Fillfields

This module provides all key strings for *Central.* It contains a cluster of functions which either initialize a field or prompt the user (where necessary) and use *Enterwords* to solicit a string, which is then checked and/or modified as required. The reason *Fillfields* must be a module by itself is that its contents may often need to be modified for slight changes in application, and this can be done without reference to other routines.

The following functions:

get_title (pointer)
get_class (pointer)
get_author (pointer)
get_source (pointer)
get_borrower (pointer)
get_ISBN (pointer)
get_libref (pointer)
get_status (pointer)
get_pubdate (pointer)
get_bdate (pointer)
get_borrowed (pointer)
get_location (pointer)
get_keys (pointer)
get_comments (pointer)

prompt the user as necessary and solicit a string to fill the appropriate field starting at *pointer* and continuing to an appropriate maximum length. There are also the two initialization functions

init_status (pointer)
init_lcr (structpointer)

which are used to set initial values when a new record is being added [*init_status* queries the user and sets the string at *pointer* to the single-byte code for either 'PRESENT' or 'ONORDER'; *init_lcr* sets the left, centre and right tree-pointers (see FILE STRUCTURE below) to their required initial values].

The data-entry functions involve some modification of the input data (usually extraneous spaces are deleted) and testing (e.g. for control characters, maximum length, etc.). In the case of *get_class, get_status* and *init_status* the string is converted into its corresponding character for internal use. In any case, diagnostics and prompts will be produced as necessary, and the *Fillfields* functions will always produce a valid field.

NOTE: *get_pubdate, get_borrowed, get_location, get_comments* and *get_keys* are not currently available.

2. DATABASE STRUCTURES AND ALGORITHMS

The database used by SLOP is a system of ordered binary trees of linked lists of records. This structured is explained (as it evolved), by a process of *stepwise enhancement;* A simplified version of the binary tree will be discussed first. Then the linked lists will be introduced. The relationship between the various trees will be explained and then there will be an examination of tree-rebalancing algorithms and the freelist.

2.1. The Binary Tree

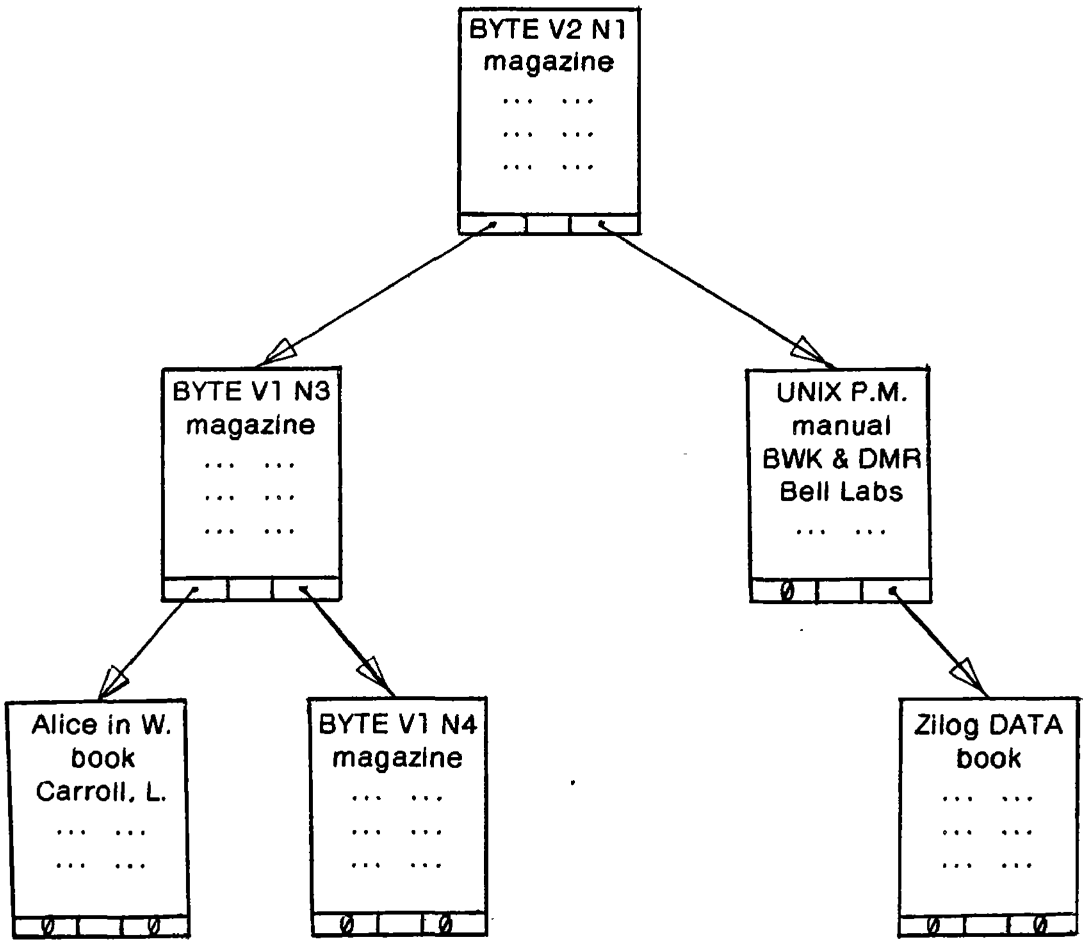
In this section, we will consider a simplified model of the databasc in which only the 'title' field is used as search key.

The database is a sequence of records, one after another. The *Records* module allows us to access these as the basic units of file access.

First, there is a header node, which contains all information pertinent to the database as a whole, but not relating to any specific record. For example, this contains the locking flag which prevents multiple simultaneous access to the database. It is the first record on the file.

Following this, there is a sequence of data records numbered 1, 2, 3, etc. Each of these contains all of the data fields for one item in the library, and may be accessed directly (through the *Records* module) once this **record number** is known.

In our hypothetical model, an ordered binary tree is used to provide fast access when the title is given. It may look like the following:



Thus, in addition to the data fields, each record contains two 'pointers'. These are, of course, the record numbers of the roots of the appropriate sub-trees. The left subtree contains records which have titles that are LESS, and the right subtree contains records which have titles that are GREATER. [The *Utilities* module provides a routine (**strcomp**) to do this comparison of fields.]

The record number of the root of the tree is stored in the header node. If there is no data in the tree, then this record number is NULL (0). In fact, at any leaves of the tree, the pointer to a nonexistent subtree is always NULL. Note that trying to read a data record numbered NULL is impossible.

Thus, the algorithm to find a record in this tree is simple. Once the required title is known, we simply start at the root and iteratively do the following:

Try to read the current node. If this is unsuccessful, the required record does not exist.

If the target title is less than the title of the current node, Move downwards along the left pointer.

If the target title is greater than the title of the current node, Move downwards along the right pointer

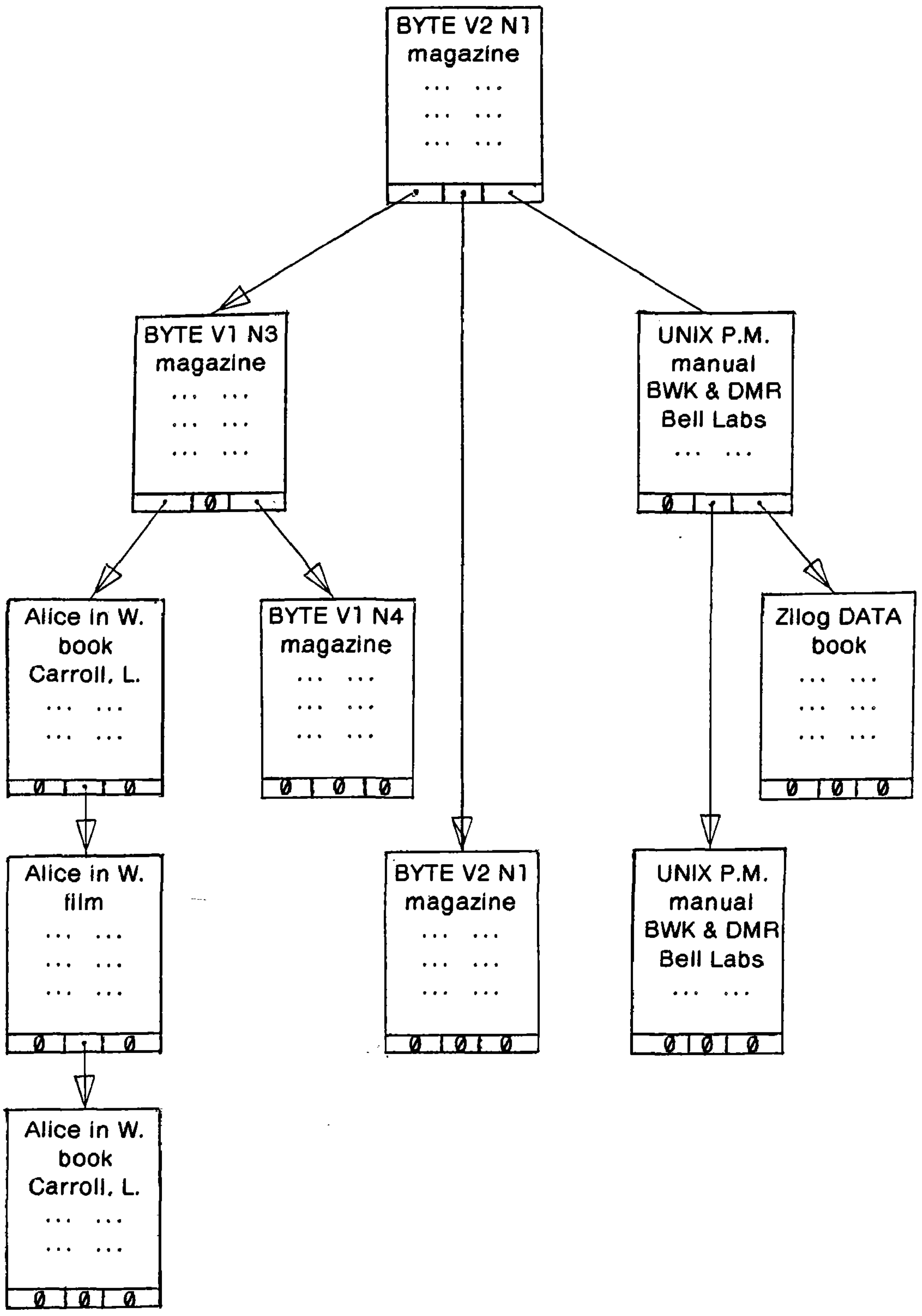
Otherwise (i.e. the target title is equal to the title of the current node) we have found it.

[This is done in the **binsearch** function.]

2.2. Linked Lists

The above model demonstrates how the tree mechanism can be used to find ONE record with a given title. But what if there is more than one ?

When there is a number of items with the same title, they are stored as a linked list and the first item on the list is a node on the binary tree. Another pointer is needed to point vertically down the linked list. So each record stores three record numbers, left, centre and right. Thus the database may look like this:



In the parlance of SLOP identifiers, this whole complex is called a *ternary* tree, since each record has three branches (although not all are used) and to distinguish it from the binary tree, which is only that part consisting of the headers to the linked lists.

Many of the SLOP commands operate on all the records which match a particular title. (Consistent with our present simplified model, we will not yet consider searching by other fields.) To do this, all we must do is a binary search (as described above) to find the first node of the linked list and, if it exists, iteratively follow the chain to its end, doing the required operation to each node on the way.

To provide an in-order list of all records (i.e. the `list' command) all that is needed is to recursively list the left subtree, list all the way down the current linked list and then list the right subtree.

Adding a new record, however, becomes slightly more complicated by this arrangement. Unlike a simple binary tree, where new records can always be hung from the leaves, care must be exercised with these chains because they may grow very long. Under no circumstances do we wish to traverse the length of the chain to add a single record.

We do as follows: First, we ascertain the record number of the place where it is to go (see `Freelist' below). Then, the data fields for the record are solicited and all its tree-pointers are set to NULL. The binary tree is descended until either a match is found or a leaf is reached. If a leaf is reached, then the new node is linked on the end, like in a simple binary tree. If a matching record is found, then the new node will be joined into its linked list in the position immediately after the first node. In any case, after the new node has been linked to the ternary tree, it is then written out to the file in its nominated position.

Removing a record is done simply by flagging its *status* field to the constant *FREE*. This inhibits any printing of the contents of that record.

2.3. Multiple Trees

We have, so far, worked with a simplified model wherein only the `title' field is used for locating records. However, SLOP finds records equally well regardless of which field is used as a key. This is because *there is one tree for almost every field.* The trees (and hence the corresponding fields) are numbered in the range:

0 <= field number < NKEYFIELDS

(where NKEYFIELDS is a constant, currently 7).

There is, nevertheless, only one copy of each record, so no data is duplicated. Each record really contains three *arrays* of pointers (left, centre and right), for each field, with three pointers for each tree.

All that has already been said about the title tree still applies when the other trees are ignored. Similarly, the same could be said about the author tree, or the class tree, or any of the others. Note that a new node must be added to all trees, but a search or a traversal need only reference one.

Note also that there is no tree for the `status' field. Since status changes every time an item is received, borrowed, returned or removed, having a status tree would mean moving a node from one place to another on the tree (The same problem arises with the `borrower' tree). This is a planned extension. The proposed algorithm is to flag the old node `FREE' and then add the updated node to each tree in the same way that new nodes are added.

2.4. Tree Rebalancing Algorithms

The tree manipulation algorithms described above are designed to give the best possible *immediate* response to the user. That is to say, hanging a new record directly from the leaves of the trees and simply flagging removed records are very quick operations in themselves. However, if the input is (by chance or otherwise) sorted to a large extent, then this node-addition algorithm will soon lead to a painfully slow search tree. Similarly if `removed' records are left in the trees.

The solution to these problems, which does not incur any penalty in immediate response time, is to rebalance the trees *totally* at a time when the user is not waiting.

Thus, when the user has quit execution a background process examines a count of the number of changes and/or removes which have been done since the trees were last rebalanced and, if it is significant relative to the size of the database, the following maintenance is done:

It is necessary for each *binary* tree (i.e. disregarding the lengths of vertical chains) to be perfectly balanced so that the head of any given chain can be located as fast as possible. So each tree (taken one at a time) is traversed (in-order) to construct an in-core list of record numbers in order of the current field. This list is then used to rebuild a perfectly balanced tree.

It is also desirable to disconnect all free nodes from the tree. Thus the traversal algorithm (function **buildlist** in the *Treeshuffle* module) which is done on each tree is as follows:

- Attempt to load the root node. If this is impossible, then we are past the leaf of the tree, so return.
- Recursively traverse the left sub-tree (i.e. the tree whose root has a record number given by the current left pointer).
- Move down the current chain (if necessary) until reaching either a non-free node or the end of the chain.
- If a non-free node was reached :
  - This will be the head of the new chain. Put its record number on the in-core list.
  - Move down the chain to the bottom, ignoring the free nodes and linking the others together.
- Recursively traverse the right sub-tree.

Now all the chains are optimal and intact, and we must use this list of record-numbers to generate a new, perfectly balanced binary tree using the following algorithm (function **buildtree** in the *Treeshuffle* module) on the whole list:

- We are given the two addresses between which the list has been built. If there is no space between them, then there is no tree to build, so return NULL.
- Ascertain the middle address of the list.
- Recursively build a tree from the sub-list between the start and middle addresses, and ascertain the record-number of its root.
- Recursively build a tree from the sub-list between the middle and finish addresses, and ascertain the record-number of its root.
- Load the record whose record-number is specified at the middle address. This is the root of the tree. Set its left and right pointers to the record-numbers (just found) of the roots of the sub-trees. Re-store the root record.
- Return its record-number.

Once this has been done, all the non-free nodes have been re-structured into an optimal binary tree of optimal linked lists. This whole rebalancing process is repeated for each tree. When it is finished, there are no free nodes in any trees at all. They have been completely isolated.

2.5. The Freelist

When a record is first flagged FREE, it is still necessary because it is a node in all of the search trees. However, once the tree rebalancing algorithms we have just seen have been completed, the free record has no purpose whatsoever. Its space on the file is being wasted. The UNIX file system does not permit these records to be thrown away, since no file can ever decrease in size. What we must do, then, is to *re-use* them when we want to add more recods.

In order to do this, the unlinked records are joined together in a linked list. Each node in this **freelist** points to the one physically before, starting from the last one on the file and continuing to the first one, which has a null pointer. Any of the integer fields (i.e. those which hold a record number) in the record could be used for this link. The arbitrary choice is that the title-left pointer is used. The record number of the start of the free list is stored in the header node. If there are no records in the free list, then this record number is NULL.

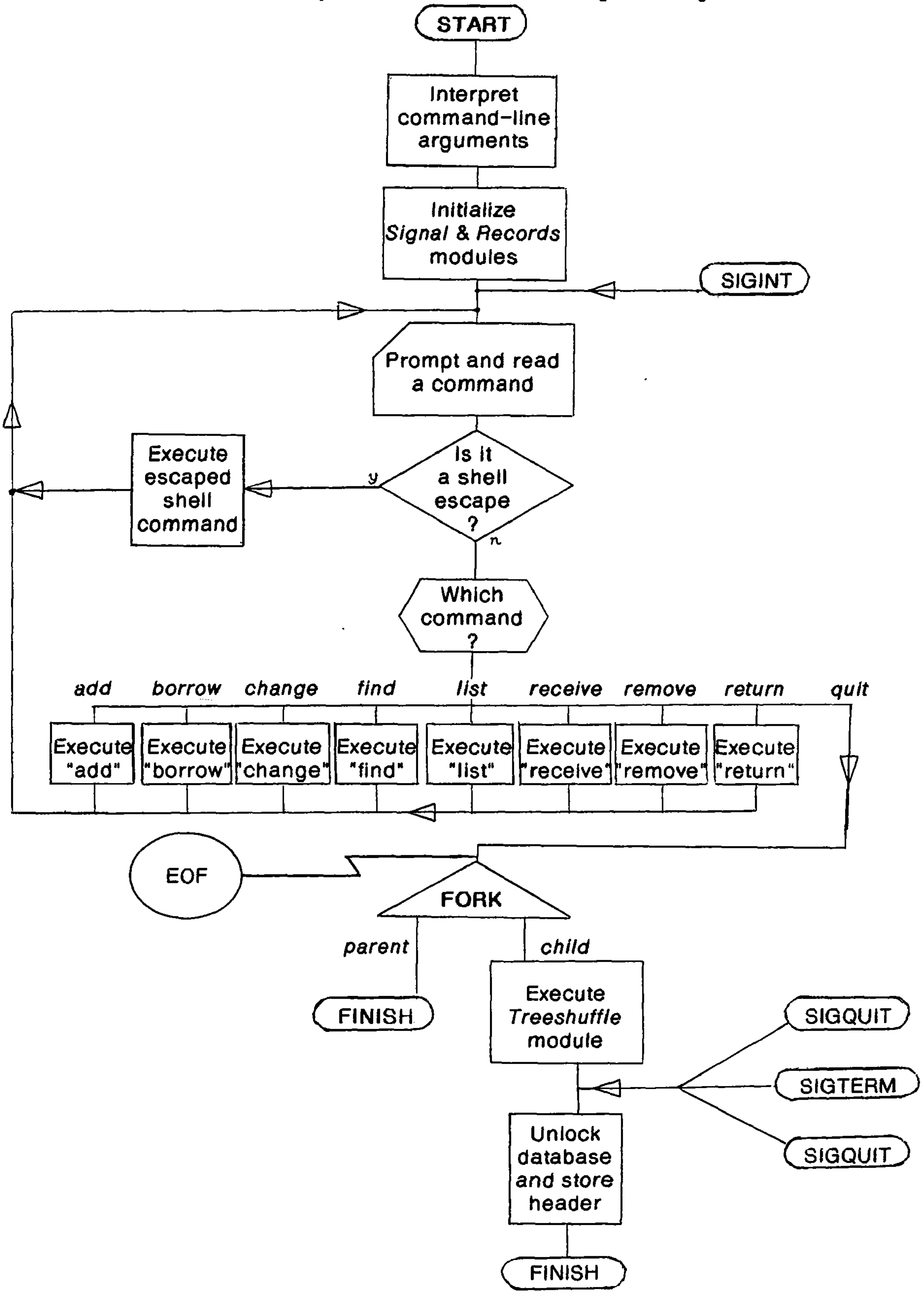
The freelist is constructed by the **flist_update** function in the *treeshuffle* module, but only after all the trees have been rebalanced and, even then, only if a sufficiently large number of records have been removed since the last time the freelist was rebuilt. The algorithm for building it is relatively straightforward:

```
Set the integer datum lastfree to NULL (0).
For each record on the database file (sequential search):
    Load the record.
    If it flagged FREE:
        Set its title-left link to lastfree.
        Set lastfree to its record-number.
        Re-store the record.
Set the freelist-pointer in the header node to lastfree.
```

From then on, when a record is to be added, the freelist-pointer on the header node is consulted. If there is a freelist, the second one along becomes the new first node of the freelist, and the new data record is put in the physical place of the free node just unlinked. Of course, if there is no freelist, the new addition is appended to the end of the database file.

3. CENTRAL CONTROL STRUCTURE

In this section, the internal control structure of the *Central* module will be examined. The overall structure may be seen from the following flow diagram:



The **main** function is responsible for the initialization processing and for sustaining the repetition of the command-processing operations. [Note that a SIGINT cannot cause the initialization to be bypassed, because its junction on the flowchart is only set up (by the *set_jmp* call (see *Signal* in section 1 above)) when that stage is reached by regular means.]

3.1. The PROCESS_CMD Function

The **process_cmd** function is called repeatedly to prompt and read a user command, and call the appropriate routine to execute it. The shell escape has such a radically different input format, that it is treated as a special case. If the command is a shell escape, *process_cmd* itself does all the necessary work to invoke a temporary shell with the standard C library function *system*. Otherwise, the command-name is looked up in a table (struct types **commands** []) and the character representation is used in a *switch* construct to call the appropriate command-handling function, which does the rest of the work (including reading arguments) for itself. Each of these functions is named directly according to the command it executes, except for the **return_** function which executes the 'return' command. [The underscore is to avoid confusion with the C 'return' instruction.]

3.2. The BORROW, FIND, RECEIVE, REMOVE and RETURN_ Command Processors

The commands:

borrow
find
receive
remove
return

are all very similar in many ways. Each involves finding all records with a given key, and performing some operation upon them.

The **findwork** function is used by each command-processor in this family to ascertain which key is being used and what particular value of that key is being sought, and then to search the appropriate binary tree for the head of the required chain. The actual tree search is done in the **binsearch** function (see *DATABASE STRUCTURES AND ALGORITHMS* above). *Findwork* places the appropriate field number in the global variable **fieldno** (which is used for this purpose throughout SLOP), and returns the record number of the node that was found (if the search was unsuccessful, it prints an appropriate message and returns NULL).

If the call to *findwork* is successful, the command-processor works its way iteratively down the chain, loading each node into the global record named **record** (which is used generally as a core copy of whatever node is being dealt with), and doing whatever operation is required.

3.3. The LIST Command Processor

The 'list' command requires that a field number is ascertained and the appropriate tree is traversed in in in-order fashion.

The **list** command-processor simply uses the *Itemopt* module to choose a tree, and then calls the recursive function **listwork** to do all the traversal, according to the algorithm given in section 2 above. Note that free nodes are treated just like the others, except in the output function, **printrec**, where they are ignored.

3.4. The CHANGE Command Processor

The `change' command is used to interactively update one or more fields of a record, and then to re-store it in such a way that the integrity of the database is not disturbed.

This command-processor has not yet been implemented (it is currently an empty function), mainly because of the lack of a suitable facility for interactively changing particular fields in a record.

3.5. The ADD Command Processor

This is an implementation of the record-addition algorithm described in 2.2, 2.3 and 2.5 above.

Firstly, it consults the freelist to determine where the next record is to go. [Note that the freelist is not actually changed yet, because an interrupt during data entry could prevent the new record from being stored.] Then the function **solicit_record** is called, which communicates with the *Fillfields* module to load all the fields of the current in-core record. The database is now about to be manipulated, so signals are disabled. The function **linknewnode** is then invoked for each tree to link the node (whose record number has just been determined) logically into the database. It is then stored physically and the freelist is updated.

3.6. Finishing Execution

When either a `quit' command is given, or an end-of-file is encountered from the standard input, the function **quit** is invoked. Its purpose is to provide a normal exit from the program, using *Treeshuffle* to optimise the database as necessary.

Its functions must proceed unhindered, so a *disablesigs* call is issued straight away. Then a *fork* system call is used to create a child process, while the parent quits immediately, leaving the database locked but returning control to the user.

This child process then calls the *Treeshuffle* module which tests the number of changes and removes which have been done and, if there have been sufficient, performs the optimising operations described in 2.4 and 2.5 above. When this is finished a non-returning call is made to the function **dienow**.

This function's job is to do the minimum possible maintenance on the database so that it is still usable, and then quit execution as fast as possible. It can also be called directly from the *Signal* module, in the event of SIGHUP, SIGTERM or SIGQUIT. All it does is flag the database as unlocked, write the core copy of the header node out to the file, and exit.

4. ACKNOWLEDGEMENTS

Many thanks go to Ross Nealon, who had the initial conception of the database being a binary tree, wrote the first hundred-or-so lines of code which sketched out the control structure that is still used, and patiently answered many questions.