1-1-1982

The Implementation of a Network Oriented Database Management System Designed to Run Under the Unix Operating System

Mary Dvonch

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Rochester Institute of Technology
College of Applied Science and Technology

THE IMPLEMENTATION OF A NETWORK ORIENTED
DATABASE MANAGEMENT SYSTEM
DESIGNED TO RUN UNDER THE UNIX* OPERATING SYSTEM

A thesis submitted in partial fulfillment of the
Master of Science in Computer Science Degree Program
by
MARY ANN DVONCH

Approved by: Peter H. Lutz
Peter Lutz - Committee Chairman

William Stratton
William Stratton Committee Member

Roy D Czernikowski
Roy Czernikowski Committee Member

Michael J. Lutz
Michael Lutz - Special Technical Advisor

January 1982

*UNIX is a Trademark of Bell Laboratories
Title of Thesis:

THE IMPLEMENTATION OF A NETWORK ORIENTATED DATABASE MANAGEMENT
SYSTEM DESIGNED TO RUN UNDER THE UNIX* OPERATING SYSTEM

I, Mary Ann Dvonch, prefer to be contacted each time a request for reproduction
is made. I can be reached at the following address:

Computer Science Department
Rochester Institute of Technology
One Lomb Memorial Drive
Rochester, New York 14623

*UNIX is a Trademark of Bell Laboratories
I would like to thank my thesis advisor, Dr. Peter Lutz, for the endless hours he spent directing my efforts on this project. Without his help and guidance, this project could not have been completed. I would also like to thank Mr. William Stratton for the ideas for this project and for the use of his B-tree programs. I wish to thank Dr. Roy Czernikowski for the time he has spent as a member of my committee. I would like to thank Mr. Michael Lutz for the time he spent as special technical advisor. Last, but certainly not least, I would like to thank my husband, Jerry, and my children for their help and support during the evolution of this project.
ABSTRACT:

THE IMPLEMENTATION OF A NETWORK ORIENTED DATABASE MANAGEMENT SYSTEM

DESIGNED TO RUN UNDER THE UNIX* OPERATING SYSTEM

We have designed a network approach, low level access system. Our conceptual model is specifically designed to run under the UNIX operating system. Our model has been tooled in the UNIX tradition. We have also supplied a manipulation language for use on the database. Our system does its own input and output buffering.

*UNIX is a Trademark of Bell Laboratories
# TABLE OF CONTENTS

## 1. INTRODUCTION

1.1. DBTG NETWORK MODEL OVERVIEW 1-2
1.2. PROPOSED SYSTEM 1-4
1.3. ASSUMPTIONS AND DESIGN PRIORITIES 1-4

## 2. A NETWORK APPROACH DATA MODEL

2.1. MEMBERSHIP CLASS 2-1
2.2. MODEL ENTITIES 2-2
   2.2.1. RECORD TYPES 2-2
   2.2.2. SET TYPES 2-4
2.3. DATA STRUCTURES EMPLOYED 2-5
   2.3.1. B-TREE INDEXES 2-6
   2.3.2. TABLE STORAGE IN BINARY TREES 2-6
   2.3.3. EXPRESSING SET RELATIONSHIPS 2-8
2.4. INPUT/OUTPUT BUFFERING 2-9
2.5. ERROR HANDLING 2-10

## 3. PROTOTYPE

3.1. RECORD TYPE DESCRIPTIONS AND CONTENTS 3-1
3.2. SET TYPE DESCRIPTIONS AND CONTENTS 3-3
## TABLE OF CONTENTS

4. **AVAILABLE PRIMITIVES**

4.1. DATABASE DEFINITION DIRECTIVES 4-2
   
   4.1.1. RECORD TYPE ADD 4-2
   
   4.1.2. SET TYPE ADD 4-2

4.2. DATABASE MANIPULATION PRIMITIVES 4-3

   4.2.1. DATABASE EXPANSION 4-3
      
      4.2.1.1. ADD RECORD 4-3
      
      4.2.1.2. ADD OWNER 4-4
      
      4.2.1.3. ADD MEMBER 4-4

   4.2.2. DATABASE CONTRACTION 4-4
      
      4.2.2.1. DELETE RECORD 4-5
      
      4.2.2.2. DELETE MEMBER 4-5
      
      4.2.2.3. DELETE OWNER 4-6

   4.2.3. DATABASE NAVIGATION 4-7
      
      4.2.3.1. FIND RECORD 4-7
      
      4.2.3.2. FIND OWNER 4-8
      
      4.2.3.3. FIND FIRST 4-8
      
      4.2.3.4. FIND NEXT 4-9

   4.2.4. DATABASE REORGANIZATION 4-10
      
      4.2.4.1. CHANGE OWNER 4-10
      
      4.2.4.2. CHANGE ALL 4-11

   4.2.5. SAY GOOD-BYE 4-11
# TABLE OF CONTENTS

5. CONCLUSIONS AND FUTURE EXTENSIONS
   5.1. IMPLEMENT A SUB-SHEMA WITH A DATA SUBLANGUAGE 5-1
   5.2. TAKING OUT THE TRASH 5-2
   5.3. DELETION OF RECORD AND SET TYPES 5-4
   5.4. SPIFFY OPTIONS 5-5

BIBLIOGRAPHY

APPENDIX A: LISTING OF DATABASE MANAGEMENT PROGRAMS
1. INTRODUCTION

Out of the ever growing use of computers to store and manipulate data has grown the need to impose some structure on the data, facilitating the use of the data. To answer to those needs, the concepts of database systems and database management have been developed. A database is a collection of stored operational data used by an application system [Date 77]. A database management system provides the user or users with control of this operational data. The heart of any database system is the data model (or conceptual model). Although there are three well known approaches, relational, hierarchical, and network, we have decided to direct our attention to the network approach for our data model.

In a network approach, an entity is represented by a "record" and the association between entities is represented with a "link". The links are used to group records into sets. Each set type may have any number of occurrences. Each set occurrence has one owner record and any number of member records. However, a specific record type may not be an owner and a member in the same set. A network database system can implement many to many mappings more efficiently than a relational system, since a relational system would require more space to represent the mappings. Also, the network model tends to be more symmetric than the hierarchical model. Although a network model may offer more generality than a hierarchical model and may be more efficiently implemented than a relational model, there is little doubt that a network approach is more difficult to conceptualize [Date 77]. A network model requires an understanding of both record types and links, and their interrelationship. The implementation of many-to-many relationships is not straightforward, often requiring dummy record types. With practice one gets used to this technique. While the network approach to databases may be
conceptually more difficult to understand, we have made every effort to ensure simplicity and clarity of usage.

1.1. DBTG NETWORK MODEL OVERVIEW

A network data model is backed by the Conference on Data Systems Language (CODASYL) Data Base Task Group (DBTG). The reports of DBTG in 1971 and 1976 have influenced the national and international standards for databases. We have been guided in the design of our database system by the DBTG reports. The schema basically consists of the implementation of (1) schema declarations (DDL) and (2) a set of primitives for manipulation (DML). The following paragraphs compare our system with the DBTG proposal.

While the DBTG proposal included dividing the total storage space into a number of "areas", our system has only one area in which all stored data resides. We feel this single area is reasonable in our model since we do not envision the use of the present model as a large scale database.

The DBTG proposal allows the user to specify a membership class by means of an INSERTION/RETENTION entry. The INSERTION entry determines the storage class for a record type while the RETENTION parameter determines a removal class [Date 77]. The DBTG proposal allows AUTOMATIC or MANUAL as a storage class and FIXED, MANDATORY, or OPTIONAL for a removal class, with any combination of storage/removal class for any set type. That is, a given record type may have a different membership class in different set types. MANUAL storage requires that the user insert a member into a set occurrence with a specific "insert member" command. With AUTOMATIC storage, the database management system automatically inserts the member into the appropriate set.
FIXED retention dictates that once a record has been entered as a member in a set occurrence, that member can never have any existence in the database not as a member in that set occurrence. MANDATORY retention is like fixed except that a member is allowed to be moved to a new owner within a set type. OPTIONAL retention allows a member to be removed from a set occurrence without completely removing the member from the database. Given the time limitations of this project, we could not begin to support all of these membership classes. Instead, we chose a specific membership class (MANUAL/MANDATORY - See Chapter 2, Section 1) and implemented it across all record and set types. We have made every effort to tool the model in such a way as to allow future extensions to provide a richer set of membership classes.

DBTG allows several forms of set selection. The set selection clause may be thought of as defining an "access strategy" for the set [Date 77]. Our system places the responsibility for selecting the correct set occurrence on the user. He must supply the name of a set he expects to use. However, all of our routines operate through user established currency pointers, so it would be a relatively easy modification to allow the user the option of supplying the set name or using the current of run set type as a default.

The DBTG data model allows for both ACTUAL and VIRTUAL sources for data items. A VIRTUAL data item is one that is not physically stored in a record, but rather could be produced by applying some function to existing fields. Alternatively, the source of a data item could be declared as ACTUAL, which indicates that the data item actually exists in the database. The data item could be a field or a group of fields. Our network model does not deal in fields, other than to produce keys and to make sure that the number of fields in a record matches the
number of fields in its type definition. It would seem reasonable that the software
that provides a data sublanguage to the user might be able to provide some virtual
capabilities.

1.2. PROPOSED SYSTEM

Figure 1-1 depicts the proposed system. The user/database system interface
includes software to provide a subschema definition language and a stand alone user
query language. It is hoped that this software will be provided as a future exten-
sion of this system. See Section 5-1 for further information and clarification. The
Database Management System presently implemented provides definition directives,
which allow the creation of record and set types, and manipulation commands or
primitives, which allow the manipulation of the data stored in the database. The
definition directives are accepted by the DBMS and the appropriate record or set
types are defined. The record and set definitions are maintained in a binary tree
in memory while the Database Management System is in use. When the DBMS is
not being used, the definitions are stored on a file. See Section 2.4.2 for further
information. The manipulation primitives work through an input/output buffering
system maintained by the DBMS. See Section 2.4 for further information. The
manipulation primitives are discussed in detail in Section 4.

1.3. ASSUMPTIONS AND DESIGN PRIORITIES

Our network database model assumes that any record stored in the database
has a unique key. Under our present system, "unique key" is defined as a combina-
tion of one to ten fields to produce a record identification which is unique
throughout the system. Of course the limit of ten fields is merely a parameter
and easily changed. It would also be a minor change to require the key to be
DATA BASE MANAGEMENT
FILE

DATA BASE INFORMATION FILES:
1. B-TREE FILES
2. RECORD FILES
3. SET LINK FILES

FIGURE 1-1
PROPOSED SYSTEM
unique only within a record type and not throughout the entire system. Null keys are not allowed, nor is any field that is part of a key allowed to be null. We assume that only one user will be using the database at a time, so as to avoid the simultaneous update problem.

In general, the goal of efficient operation of a database system is bought at the cost of loss of flexibility of design, and flexibility of design causes some loss of efficiency of operation [Inmon 81]. In the trade-offs between performance versus flexibility, it is easy to opt for performance because the utility of performance is very apparent and immediately recognizable. On the other hand, the utility gained by building elasticity into a data structure doesn’t blossom until a later time. Data that is flexible survives best in the face of environmental change. It is only when data that is inflexible undergoes change that the wisdom of building flexible data structures becomes apparent. Along with data flexibility, or perhaps as an extension of it, we have added the goal of ease of extension. Our conception of this system is not that it is a total database package that will serve the needs of all users. Rather, this system is meant to serve as the kernel of a much larger database system. Thus, our efforts were centered on producing a system that would allow maximum flexibility and ease of extension. Obviously, these were paid for at times at the cost of loss of efficiency in either space or time.

A second and very real priority was the optimization of application development time. We attempted to design and implement a well defined system within the given time frame. Given unlimited time and resources, a larger, more powerful system could have been developed. However, it is hoped that the flexibility built into our system will allow it to meet changing user needs and expectations.
Since a network model is at best intricate, we emphasized ease of modification when designing our data structures and the create, delete, and find functions. One of our major concerns was minimizing pointer chasing. C was most helpful in dealing with the pointer problems, and we feel that the pointer design and implementation is one of the strengths of our system. We also attempted to keep the data structures as straightforward as possible in order to ease understanding. Although network models, by their very nature, are more difficult to understand than relational models, we hope that by emphasizing ease of manipulation in the project, we have also added ease of understanding.

Since optimization of execution time and minimizing data storage were not our main concerns, they have undoubtedly suffered. We did supply our own input/output buffering system in an attempt to lower execution time. We stored pointers to data instead of copies of data whenever it was feasible, in order to minimize data storage. However, if it came to a choice between flexibility and optimized efficiency, we invariably chose flexibility.

Our implementation of a network model has been specifically designed to run under the UNIX operating system. In keeping with the UNIX overall approach, we have designed tools which can be used to define record and set types (DDL), and to create, manipulate and delete record and set occurrences from a database (DML). For our purpose, a "tool" is a specialized function that solves a general problem and is easy to use.

Our tools are written in the C Programming Language. C is well supported under the UNIX system. The UNIX operating system itself is written in C, as is most of the UNIX applications software. C is independent of any particular machine architecture and every effort has been made to make this system
"portable". C's recursive ability has been used throughout our network system, allowing small pieces of code to perform significant tasks.

Our database system handles its own buffering for input and output. The reads and writes through the UNIX operating system are accomplished in page size blocks. Stonebraker [81] compared and contrasted the way operating systems, especially UNIX, handled buffer pool management for database systems. He concluded, "The strategy used by most DBMS's is to maintain a separate cache in user space. This buffer pool is managed by a DBMS specific algorithm." Therefore, we feel that handling our own input/output buffering is an efficiency enhancement to our system.
2. A NETWORK APPROACH DATA MODEL

We have provided a network approach, low level definition and manipulation system for a DBTG like network database, that operates under the UNIX operating system. A manipulation language has been provided to communicate with the storage and retrieval routines. However, it should be understood that the manipulation language is not written for the "user" of the database. It is assumed that another layer of software will take user commands and communicate them to this system. This software will provide a data sublanguage for the user, will parse the user commands, and then use the manipulation language to satisfy the user requests. The following is a list of the definition and manipulation primitives provided by this database management system. The primitives are explained in detail in Chapter 4.

Primitives to create record and set types:
  record type add
  set type add

Primitives to manipulate record and set occurrences:
  add record
  add member
  delete record
  delete member
  delete owner
  find record
  find owner
  find first
  find next
  change owner
  change all
  end update

2.1. MEMBERSHIP CLASS

The membership class for our system is MANUAL/MANDATORY. The MANUAL storage class implies that storing a record, R, in the database, does not
automatically insert R into a set occurrence in which it participates. To insert R into a set occurrence, an explicit "add member" command must be given. Note, however, that if R's record type participates as an owner in any set type, S, R is automatically an owner occurrence in S. The MANDATORY removal class dictates that once an occurrence of a member, say A, has been entered into a set type, say S, as a member in a set occurrence, A can never have any existence in the database not as a member of some occurrence in S [Date 77]. Specifically, A can be moved to a new owner in S with a "change owner" or "change all" command, but may not be removed from S without being removed from every set occurrence in which it participates. Note that the removal of an owner in a set occurrence then causes the recursive removal of all its member records. This removal in the hands of the untutored could create havoc in the database. MANDATORY removal class protects the integrity of the data, in that once a record is removed all traces of it are deleted.

2.2. MODEL ENTITIES

This model uses two different entity types to represent the information contained in the database. Record types are used to represent stored user data and set types are used to store relationships between record types.

2.2.1. RECORD TYPES

Record types represent occurrences of some entity that the user expects to store and manipulate. Before a user can store a record of a specific type in the database, he must define that record type for the database management system. A record definition consists of a record type name, a field delimiter used for that record type, a specification of the number of fields the record type will contain, a
specification of the number of fields that will make up the key for the record type, and the location of the key fields in the record type. The system creates the record type by inserting the record type name with ".rf" appended to it into the binary tree containing record type definitions. The user is allowed to use variable length records within the same record type.

Once the user has defined a record type, the system will store the information about that type for future use. The information is stored in a table (implemented as a binary tree), which is used by the Database Management System. The next time the system is used, it will recall all previous record type definitions. When the user then issues an "add record" command for a particular record type, the database system fills an input/output buffer with the desired record, checks the field count on the record for an error, and builds the record's key. The key is also error checked for duplicity. The record is then passed to the input/output buffering system where it is stored at the end of a file whose name is the record type name with the suffix ".rf". As the record is passed to the buffering system, the location of the record within the file is determined. The location of a record is the block offset from the beginning of the file and then an offset to the position within the block. Once the address of the record within the file is known, the address information and the record key is stored in the record type B-tree. If the record type participates as an owner in any set type, the new record is added as an owner occurrence each such set.

In an effort to keep these records "clean", we have stored them in regular flat files in normal UNIX tradition. "Clean" means that no database pointers, id numbers, or other database management information is stored in the user record files. This allows the stored database files to be used with any other UNIX tools.
For example, a record file could be printed directly. This separation of record files from database information is one of the strengths of our system, for it is in keeping with the UNIX idea of software tools. However, the user should remember that deleted records still remain on the record files until garbage collection is done.

2.2.2. SET TYPES

Set type occurrences are represented by links stored in set type information records. The set type information allows the user to walk around the links that associate a particular owner with members in his set. Such set type information is maintained separately from record type information.

Before a user can store any set occurrences for a set type, he must define the set type. This is done with a "set add" command, followed by the set type name, the owner record type name, and the member record type name. Once the set type is defined, the database management system stores the set type information in a table along with the currency information for that set type. The set type definition must be given before any records of the set owner record type have been added to the database. The system creates a set type by inserting the set type name, with a ".sf" appended to it, into the set type table (implemented as a binary tree). Once a set type has been defined in the database, it may not be removed. The next time the database is used, the database management system will recall all previous set type definitions.

After the set type has been defined, owner occurrences are automatically created whenever a record of the owner record type is added to the database. When an owner record occurrence is added to the database, a link-record for the
owner is built (See Section 4.3). This link-record is then given to the i/o buffer system to be added to a file named by the set type name with ".sf" appended to it. As the record is passed to the i/o buffer system, information is gathered on the location of the pointer record within the set type file. This location information and the owner key are stored in the set type B-tree. The link-records are stored as regular UNIX files and, therefore, can be used as arguments for other UNIX tools.

If the user wishes to add a member to a set occurrence, he must give an explicit "add member" command. The user specifies the set type and owner key of the appropriate set occurrence. Inserting the new member into a set occurrence is accomplished by building a link-record for the new member. These link-records make up a chain that connect the set occurrences, and can be traversed to obtain all the members of a set occurrence. (For exact information on link-record insertion, see Section 4.3.) The new member occurrence is added as the immediate successor to the owner. Order of retrieval is therefore LIFO (Figure 2-1).

As with the owner's link-record, the member's link-record are stored through the buffering system, on a file named by the set type name with ".sf" appended to it. Once the link-record is stored, the file offset information along with the new member key is stored in the set type B-tree.

2.3. DATA STRUCTURES EMPLOYED

We have used three major data structures: B-trees, binary trees, and doubly linked lists. Obviously, other data structures were used in the coding of the system, however, they have little impact on the design of the overall system.
2.3.1. B-TREE INDEXES

We used the B-tree index [Wirth 76] to store our key and offset information for both the record types and the set types. The B-tree is, de facto, the standard organization for indexes in a database system [Comer 79].

2.3.2. TABLE STORAGE IN BINARY TREES

We used binary trees to store the database management record type and set type tables. Separate trees were maintained for the record type table and the set type table. Each tree contains one node for each record/set type. The nodes are maintained so that at any node, the left subtree contains only record/set types whose name precedes the node’s record/set type name with a string comparison; the right subtree contains only types that are greater. Each record type node contains:

- the record type name,
- the field delimiter for this record type,
- the number of fields to expect in the record,
- the number of key fields, i.e., the number of fields used to build the key,
- an array containing the position of the key fields,
- a structure of currency pointers,
pointers to the left and right record type binary subtree.

Each set type node contains:

- the set type name,
- a pointer to the record table entry for the owner record type,
- a pointer to the record table entry for the member record type,
- the file descriptor and root for the B-tree index for this set type,
- current of set type key
- a structure of currency pointers,
- pointers to the left and right set type binary subtree.

"Currency pointer" structures are used primarily by the input/output buffering routines to place records in the buffers and ultimately into files. The structures contain:

- logical file and block offsets,
- physical file and block offsets,
- final file and block offsets,
- a switch that indicates if a record has just been added to the file,
- a pointer to the file buffer for this record/set type.

The logical offsets indicate the beginning of the record that was most recently referenced for the record/set type. This supplies the current of run for this record/set type. The physical offsets indicate the end of the record that was most recently referenced for the record/set type. The final offsets indicate the end of the file for the record/set type.

When a new record or set type is defined, a node is created for it in the appropriate binary tree. At the conclusion of the database update, the pertinent information from the binary type trees is stored in a file named dbm_file. (This name is a parameter, and easily changed.) The first field on the dbm_file is the number of record types defined within the database, followed by the number of set types defined. After these come the record type information and then the set type
information. When the database management system is run, the system builds the new binary type trees from the information stored on the dbm_file. At this time the record type B-tree is opened and its file descriptor and root are stored in global storage. Again, the dbm_file is a normal UNIX file that can be examined by and used with other UNIX tools. If no record types or set types have been defined, the system will print a message informing the user of the fact.

2.3.3. EXPRESSING SET RELATIONSHIPS

Our implementation uses doubly linked lists to represent set occurrences. The front pointer of any owner or member in a set occurrence contains the key of the next owner or member in the set occurrence. The back pointer of any member contains the key of the prior member or the owner. The owner's back pointer is always null. These pointers are placed in a link-record, with the forward pointer first and backward pointer second. The link-record is then stored on the set type file, and the owner/member key and the file offset of the link-record are stored in the set type B-tree. As an example, suppose owner A1 owns members B1 and 2B. Then A1's forward pointer would be 2B and backward pointer would be null. B1's fp would be A1 and bp would be 2B. 2B's fp would be B1 and bp would be A1. In order to follow the link to the next owner or member link, the key is found in the set type B-tree, and the offset for the next link-record is obtained. When a member is deleted from a set, the previous and following member's links are reset to exclude the member, and the deleted member's key is removed from the set type B-tree. The member is also removed from the record B-tree and is then removed from any other set where the member participates as an owner or member. When an owner is deleted from the set, the owner's key is removed from the set B-tree, then removed from the record B-tree, and finally removed from any
other set where the owner participates as an owner or member. When an owner is deleted from a set, each of his members is also deleted.

2.4. INPUT/OUTPUT BUFFERING

As a time efficiency measure, we have implemented buffered input and output under the control of the database management system. Many DBMSs, including INGRES and SYSTEM R, have chosen to put a DBMS managed buffer pool in user space to reduce overhead [Stonebraker 81]. Either at the beginning of the update session for predefined types, or at the time a new record type or set type is defined, a page-size block of memory is allocated for its i/o buffering. A pointer to the buffer block is stored in the set/record type table for use by the i/o system. All reads and writes are accomplished in page-size blocks. When the i/o system is asked by the calling routines to store a record, the i/o system expects to find the record in a sending buffer. The i/o routines will then add the record to the end of the appropriate file and return the location of the record to the calling routine. This is accomplished in the buffer area for the record/set type and does not always require actual writes to the file. When the i/o system is requested by the data model to retrieve a record, the i/o system will try to fill the order from data already resident in main store in the buffer area for that record/set type. This will be possible frequently, especially when link-records are sought. When the i/o system locates the appropriate record, it is placed in a receiving buffer supplied by the calling routine. The i/o system updates the currency pointers for record/set types as it does its storage and retrieval functions.
2.5. ERROR HANDLING

All error handling is done through two routines. One error routine accepts a single string as the error message, prints the message, and returns an error indicator. The other error routine accepts an error message and a key value. Both the message and the key value is printed and an error indicator is returned. It would be very easy to change these routines so that no error message was printed, but rather a specialized code would be returned.
3. PROTOTYPE

We built a simple prototype database to be manipulated with our database management system. The prototype consists of four record types and three set types with certain relationships expressed through set occurrences.

3.1. RECORD TYPE DESCRIPTIONS AND CONTENTS

Record Type Name: faculty
Field Delimiter: *
Number of Fields: 5
Number of Key Fields: 1
Position of Key Fields: 2
Contents:

Peter*A1*10*A186*25
Bill*A2*10*2132*57
Roy*3A*10*A285*72
Jack*4A*10*1116*13

Record Type Name: student
Field Delimiter: :
Number of Fields: 4
Number of Key Fields: 1
Position of Key Fields: 3
Contents:

Mary:CAST:B1:Comp Scie
Leslie:CAST:B2:Comp Scie
John:SP:3B:PPPD
Tom:CAST:4B:Syst Soft
Mary:SP:5B:PPPD
Record Type Name: housing
Field Delimiter: *
Number of Fields: 3
Number of Key Fields: 1
Position of Key Fields: 1

Content:

405*Billings*25
216*Watson*1105

Record Type Name: courses
Field Delimiter: *
Number of Fields: 8
Number of Key Fields: 4
Position of Key Fields: 5 1 3 4

Contents:

B1*0601*81*1*875*1*D*r
B2*0601*81*2*875*1*A*nr
B1*0601*81*1*720*2*B*nr
3B*0532*81*2*875*1*A*nr
B2*0532*81*2*850*1*B*r
B1*0601*81*2*875*1*B*r
5B*0601*80*2*875*1*D*r
5B*0601*81*3*875*1*B*r
3.2. SET TYPE DESCRIPTIONS AND CONTENTS

Set Type Name: fs
Owner Record Type: faculty
Member Record Type: student
Relationship: the faculty member is the advisor of the student

Contents:

A1 owns
  B1
  3B
A2 owns
  B2
  4B
3A owns
  5B
4A has no members

Set Type Name: sc
Owner Record Type: student
Member Record Type: courses
Relationship: the student has taken the course

Contents:

B1 owns
  875B1811
  720B1812
  875B1812
B2 owns
  875B2812
  850B2812
3B owns
  8753B812
4B has no members
5B owns
  8755B802
  8755B813

Set Type Name: hs
Owner Record Type: housing
Member Record Type: student
Relationship: the student is a resident of the housing complex

Contents:

405 owns
   Bl
5B
216 owns
   3B

See Figures 3-1 and 3-2 for a graphic representation of these relationships.
FIGURE 3-2
PROTOTYPE SET OCCURRENCES
4. AVAILABLE PRIMITIVES

We have made every attempt to "tool" our primitives in the UNIX tradition. We have tried to develop tools that are easy to use, work well within the overall UNIX environment, and efficiently accomplish the task for which they are designed. The manipulation language was designed to be terse, as most UNIX commands are. However, in most cases the user is allowed to be more verbose. As an example, the "ra" or record add command, is actually parsed by the leading "r". Anything else can follow within the word; thus "r", "ra", "recadd", and "recordadd" would all be accepted as a record add command.

All record type names and set type names consist of the first ten characters of the word supplied. The number ten exists as a parameter in the system and is easily changed. (UNIX allows eleven characters.) As an example, if the name RochesterInstitute was input as the name of a record type, the record type would be known to the system as RochesterI.

The maximum number of key fields for a record type in the system is set at ten, with the maximum number of characters in a key set at twenty. Again, these numbers are parameters and easily changed. If more than the maximum number of characters are entered for a key value, an error message is printed and the command is aborted. As an example, if A12345678901234567890 was entered as a key value in a command, an error message would be printed.

Some of the commands are allowed to include optional input or output file names. The file name, which can be any UNIX pathname, may be up to forty characters long; extra characters are ignored. Note: all examples in the following tool descriptions will refer to the prototype described in Chapter 3.
4.1. DATABASE DEFINITION DIRECTIVES

The database definition primitives deal with defining the record types and set types.

4.1.1. RECORD TYPE ADD

The record type add command describes a record type for the database management system. The DBMS then stores the information as a node in the binary record type tree. The call to this tool has the form:

\[ ra \text{name} \text{field delimiter} \# \text{of fields} \# \text{of key fields} \text{position of kf} \]

A record type add command for courses would look like:

\[ ra \text{courses} * 8 4 5 1 3 4 \]

4.1.2. SET TYPE ADD

The set type add tool allows the user to define a set type to the database management system. The DBMS then stores the information as a node in the binary set type tree.

The call to the set type add tool has the form:

\[ sa \text{name} \text{owner record type name} \text{member record type name} \]

The set type fs would have been defined by the command:

\[ sa \text{fs faculty student} \]
4.2. DATABASE MANIPULATION PRIMITIVES

The database manipulation primitives allow the user to add items to the database, remove items from the database, find items within the database, and reorganize members within set types.

4.2.1. DATABASE EXPANSION

When an owner or a member is added to a set, a link-record is built and integrated into the appropriate set. Link records consist of back and forward pointers and are discussed in detail in Chapter 2, Section 4.3. Once the link-record is built and added to the file for the set type, the key for the owner/member along with the file offset of the link-record is stored in the set type B-tree.

4.2.1.1. ADD RECORD

The add record tool allows the user to store a record for a particular record type in the database. If that record type is the owner in any set, another tool, "add owner", is called and the new record key is automatically added as an owner in the set. The call to the add record tool has the form:

    ar <record type name> [<input file name>]

If the input file name is supplied, all the records on the file are added to the database. Otherwise, records are taken from the standard input until an EOF is read. An add record command for a faculty record would look like:

    ar faculty
    Peter*A1*10*A186*25
    EOF
4.2.1.2. ADD OWNER

If a newly added record is an owner in a set type, the system will automatically generate an add owner call. The add owner tool call has the form:

ao <set type> <owner key>

For the above add record to faculty, a generated add owner tool call would look like:

ao fs A1

4.2.1.3. ADD MEMBER

The add member tool adds a member key to a set occurrence. The system will check the member record to make sure it conforms to set and record definitions. The call to the add member tool has the form:

am <member key> <set type name> <owner key>

A command to add student B1 to faculty A1 would be:

am B1 fs A1

4.2.2. DATABASE CONTRACTION

The database management system’s deletion tools are a recursive delight. Since deleting a record implies possible deletion of set occurrences and vice versa, recursion is required. Each deletion tool was designed to do its own particular job and then call a friend to do the rest.
4.2.2.1. DELETE RECORD

When a record is deleted from the database, the record key and file offset are removed from the record type B-tree. Since the offsets for the actual record can no longer be referenced, the record is logically deleted from the database. However, the actual physical record will still remain on the record type file. This record then becomes the province of the garbage collection system which is discussed in detail in Chapter 5. The call to the delete record tool has the form:

\[ \text{dr } \langle \text{record type}\rangle \langle \text{key}\rangle \]

A delete record command to delete student Bl's record would consist of:

\[ \text{dr } \text{student } \text{Bl} \]

Once the record is logically deleted, delete record recursively searches the set type binary tree looking for sets in which this record type participates as an owner or member. If a set is found, delete owner or delete member is called to remove the discarded record key from the set. For instance, the "\text{dr student Bl}" command above would cause Bl to be removed from the fs set and the hs set as a member and the sc set as an owner. The removal from the sc set has further ramifications which are covered in detain under "delete owner".

4.2.2.2. DELETE MEMBER

The delete member tool causes a member's key and link-record offset to be removed from the set type B-tree. The pointers in the link-records of prior and post set occurrence members are reset to exclude the deleted member. The actual link-record is left on the set type file, requiring garbage collection at a later time. However, since this link-record can no longer be referenced, it is effectively
deleted from the database. The call to the delete member tool has the form:

\[ \text{dm } \langle \text{set type} \rangle \langle \text{key} \rangle \]

A delete member command for student B1 in the fs set would consist of:

\[ \text{dm } \text{fs } \text{B1} \]

First, B1's key and link-record file offset would be deleted from the fs set type, then B1's record would be removed from the student record file, then B1 would be removed as a member from the hs set, and finally, B1 would be removed as an owner from the sc set, triggering the deletion of three course records.

**4.2.2.3. DELETE OWNER**

The delete owner tool is probably the most powerful tool in the database management system. Somewhat like the A-bomb, the delete owner command can cause fallout to the farthest reaches of the database. When the delete owner command is given, the owner's key and link-record file offset are removed from the set type B-tree. Next, "delete member" is repeatedly called until all of this owner's members are deleted. Finally, "delete record" is called to delete the owner's record from the record type file. The delete owner command has the form:

\[ \text{do } \langle \text{set type} \rangle \langle \text{owner key} \rangle \]

The command to delete owner A1 in set fs would consist of:

\[ \text{do } \text{fs } \text{A1} \]

This small, unimpressive looking command would cause A1 to be removed from the fs set. Next, B1 would be removed from the fs set and the hs set and courses
875B1811, 720B1812, and 875B1812 would be removed from the sc set and the courses record file, and B1 would be removed from the student record file. Then, 3B would be removed from the fs and hs set files, courses record 8753B812 would be removed from the sc set and the courses record file, and 3B would be removed from the student record file. Finally, A1 would be removed from the faculty record file. Obviously, when it comes to the deletion tools, looks are very deceiving, and caution is the key word.

4.2.3. DATABASE NAVIGATION

The find tools allow the user to navigate the database set occurrences. With the proper sequence of find commands, the user is able to extract information from the database. If an output file name is specified with the find command, the requested record will be output to the named file. Otherwise, the record is sent to standard output.

4.2.3.1. FIND RECORD

The find record tool will find a record given a specific record type and key. The current of run unit is set to the key of the located record. The find record command has the form:

```
fr <record type name> <key> [output file name]
```

The find record command:

```
fr housing 405
```

would cause the following record to be output:

```
405*Billings*25
```
4.2.3.2. FIND OWNER

The find owner tool finds the owner record for a member key in a particular set. The current of run unit and the current of set type is set to the owner key.

```
fo <set type> <member key> [<output file name>]
```

The find owner command for student B2 in the fs set would consist of:

```
fo fs B2
```

and would yield the record:

```
Bill*A2*10*2132*57
```

The current of run unit and the current of fs member would be: B2.

4.2.3.3. FIND FIRST

The find first tool finds the first member of an owner in a specific set. Both the current of run unit and the current of set type, are set to the key of the first member. The record of the first member is output. The find first command has the form:

```
ff <set type> <owner key> [<output file name>]
```

The command to find the first member of Bl in sc would consist of:

```
ff sc B1
```

and would yield the record:

```
Bl*0601*81*875*1*d*r
```
and would leave the current of run unit and current of set type set at 875B1811.

4.2.3.4. FIND NEXT

The find next tool finds the next member in a specified set. The next member is defined to be the next member in the set occurrence of the current of set key. The find next command resets the current of run unit and the current of set to the next member's key. If the next item in the set occurrence is the owner, a message is returned. Otherwise, the record of the next member is returned. The find next command has the form:

\[ \text{fn } <\text{set type}> \ [\text{<output file name>}] \]

Assuming that the above "ff fs A2" command was given, followed by:

\[ \text{fn sc} \]

The following record would be returned:

\[ \text{Bl*0601*81*1*720*2*B*nr} \]

If the following lists of commands were given:

\[ \text{ff fs A1} \]
\[ \text{ff sc 3B} \]
\[ \text{fn fs} \]
\[ \text{fn sc} \]

the output would be:

Mary:CAST:B1:Comp Scie
3B*0532*81*2*875*1*A*nr
John:SP:3B:PPPD
No more members
It should be noted that with these basic find commands, the software package interacting with the user, will be able to create different find macros, and therefore, offer the user a more extensive set of find commands. Also, since the record type files are regular UNIX files, the user can simply cat or print the record type file, if he wants a list of all the records in a particular file.

4.2.4. DATABASE REORGANIZATION

The database management system will allow the user to reorganize the set occurrences within a given set. A member can be moved from one owner to another owner within a given set.

4.2.4.1. CHANGE OWNER

The change owner tool moves a member from one set occurrence to another set occurrence. This is accomplished by resetting pointers in the link-records so that the member is linked to a new owner. The change owner command has the form:

\[
\text{co <new owner key> <set type> <member key>}
\]

The command to change student B1 in the set fs to a new advisor 4A would be:

\[
\text{co 4A fs B1}
\]

After this command, A1 would have only one advisee, 3B, and 4A would now be the advisor of B1.
4.2.4.2. CHANGE ALL

The change all tool moves all the members of an owner to a new owner. This is accomplished by repeated calls to change owner until all the members are moved. The change all command has the form:

ca <new owner key> <set type> <old owner key>

The command to move all of owner 405's members in hs to owner 216 would look like:

ca 216 hs 405

After this command was executed, 216 will have B1, 3B, and 5B as members, and 405 will have no members.

4.2.5. SAY GOOD-BYE

The final tool supplied by the database management system is a tool that allows the user to gracefully exit the updating session. The exit tool has the form:

q

After the exit command, the database management system stores record and set type definitions on a file, closes all record type, set type and B-tree files, and closes down the system.
5. CONCLUSIONS AND FUTURE EXTENSIONS

With our data model, we have tried to provide a solid foundation on which a flexible, user oriented sub-schema can sit. We have attempted to tailor our database management tools to accomplish specific tasks, yet allot them the power to do that task well. Our database record files are "clean" and fit well into the overall UNIX file system. However, as with all things this side of Paradise, certain extensions and enhancements would increase the effectiveness of our database management system. Our hope is that all or part of the following extensions will be implemented at a future time.

5.1. IMPLEMENT A SUB-Schema WITH A DATA SUBLANGUAGE

We need to implement a subschema definition language and a stand alone user query language. This sublanguage should offer a greatly simplified view of the database to the user.

Within the user query language, certain enhancements could be made to the database management system. "Find" commands defined at this level could take some of the navigational tediousness away from the user. For example, the query language might allow the user to just say "find next" without presetting the current of set with a "find first". By keeping track of previous commands, the software could change the "find next" to a "find first". Also, a find macro might be nice. When a user wants to walk around a set occurrence, instead of repeated calls to "find next", the macro could expand a command, say "find all", to a "find first" followed by as many calls to "find next"as are needed. Perhaps, the software might even provide some safety alarms for the delete commands, to save the user from himself. Although it is not immediately apparent how this could be done, it
is a matter that should be pursued. Otherwise, the insatiable appetite of the delete commands will be a constant threat to the unwary user.

The sub-schema would have to implement some sort of field manipulation for the user. Our database management system deals in records. An add command, adds a whole record. A find command, finds a whole record. There will be times when the user wants to see a specific field of a record or will want to change a field. Obviously, we can’t allow the user to change key fields, but he should be allowed to change non key fields. This option should be supplied to the user by the sub-schema.

5.2. TAKING OUT THE TRASH

When a delete of a set owner/member or the delete of a record is enacted under the database management system, the set occurrence or record is logically deleted from the database. That is, any reference to the deleted item will not be satisfied. However, the actual physical file record is not deleted. Over time, the database management system will cease to operate efficiently. Storage will be wasted on inaccessible and useless data. Time will be wasted since the buffering system will be required to do extra reads and writes to carry the defunct information. Obviously, we will want to create some vehicle for reclaiming this storage. There are many ways this garbage collection could be implemented.

The most elegant way to do garbage collection within the database management system would be to garbage collect while the user was inputing commands or perusing recently returned records. Some type of timing element would have to be built, so that every so often the garbage collection routine would check to see if the user had input a new command. This garbage collector would clean a certain
section of a file by moving the good records as far as possible toward the front of
the file. The B-tree offset would then have to be reset. The real key to this
approach would be the timing. If timed improperly, the garbage collector could
clean to the end of file, be interrupted by the user who adds a new record to the
end of the file, then start up again and blow away the recently added record.
Granted, this approach would be a challenging design problem. However, it would
buy you free garbage collection during user "think" time.

Another approach to the garbage collection problem is a B-tree walk and
move procedure. A tool could be built that would visit each node in the B-tree
index for a file. With the file offset gathered from the B-tree, the record could
be moved to a new file and the offsets readjusted to fit the record's new place-
ment. Once the new "clean" file is built, the old file space is returned to the sys-
tem. This tool would have to be run without interruption and could tend to be
expensive time wise if the database files got very large. However, it has the
advantage of providing a means to the user to be absolutely certain that only valid
records appear on a record file at a given time. Once the user ran this tool, he
could "cat" the record file and have a copy of all the records of a certain record
type.

Yet another approach to the refuse problem for record files would be the
inspect and slide function. Armed with the record type definition, the garbage col-
lection machine could move down the record file, inspecting each record as it
moved. The key for the record would be recreated using the record type defini-
tion. Then the B-tree would be searched for that key. If the key was present in
the B-tree, the record would slide as far forward on the file as possible, and then
the offsets in the B-tree would be reset to reflect the new location. If the key
was not in the B-tree, the record would be overlaid when a valid record from farther down the file was slid over it. This method works well for the record type files; however, it will not work for the set type files, since the link-records carry no indication of their owners. At the same time, the link-records will generally take up less space than the record type records, and it is doubtful that a user would ever want to "cat" the link-records. Therefore, garbage collecting just the record files may be enough in some instances.

There are undoubtedly many more ways to collect wasted file space. The designer of the garbage collection tool will have to weigh the merits of each possibility, and implement the most reasonable approach. However, we know that garbage collection can be effectively done on the database files.

5.3. DELETION OF RECORD AND SET TYPES

It would be nice if the database management system allowed the user to delete a record or set type once it had been defined. Deletion of a record or set type would automatically carry with it deletion of any existing records or sets of that type. This deletion could be accomplished by just removing the name of the record/set type from the binary tree node where it is stored. The record/set type would then be logically deleted. With minor changes, the end of update routine could be designed to ignore any record or set type with a null name and not store it on the dbm_file. The next time the database management system was started, the deleted record or set type would not be in the binary tree. This approach gets complicated if the user is allowed to delete a record or set type anytime, because then the database management system has to contend with searching binary trees with null names scattered about. However, if the user is only allowed to delete a record or set type at the end of an updating session, the implementation
becomes trivial.

5.4. SPIFFY OPTIONS

There are several options that would be convenient for the user and relatively easy to implement. It might be nice for the user to be able to get a copy of all deleted records. This could be accomplished by using a find to get the record for the user and then doing the delete. It could be implemented as a macro at either the schema or sub-schema level. A tool to allow the user to question the DBMS about a record type or set type definition would be useful. Perhaps the user would like to know what the field delimiter is for a particular record type. Along the same line, the user might want to change the field delimiter for a particular record type. In terms of implementation, these options are time consuming rather than difficult.

A very desirable tool would be a listing tool. The tool would allow the user to walk around a set type and list all the owners with their members. A variation of this tool, could list the record type files, with the fields listed in columns and the key fields indicated. A listing of all the set type and/or record type definitions would be useful.

Undoubtedly, pages and pages could be written on possible extensions the would be helpful to the user. Many users will have special requirements that make some options more important than others. However, as in all things, it comes down to time and resources. We have made every effort to make our data model flexible enough to handle future options and extensible enough to make the implementation easy. It is our fondest hope that this embryo will grow and thrive, and that it will eventually mature into a contributing adult in the field of database
management systems.
BIBLIOGRAPHY


Comer, Douglas, "The Ubiquitous B-Tree", Computing Surveys, Vol. 11, No. 2, June 1979

Date, C. J., An Introduction To Database Systems, Addison-Wesley Publishing Company, 1977


Joy, William, "An Introduction To The C Shell"


Kernighan and Ritchie, "UNIX Programming - Second Edition"

Lehman and Yao, "Efficient Locking For Concurrent Operations On B-trees", 1979


Thompson, K., "UNIX implementation", The Bell System Technical Journal, Vol. 57, No. 6, Part 2


APPENDIX A: LISTING OF DATABASE MANAGEMENT PROGRAMS

FILES LISTED

main.c
definitions
p_c_dec
parse_dbup
rec_t
set_t
init_concl
utility
io
dbup_c_dec
dbup
/* This file contains the main driver for the database management system. */

#include <stdio.h>
#include "definitions"
#include "parse_dbup"
#include "rec_t"
#include "set_t"
#include "init_concl"
#include "utility"
#include "lop"
#include "dbup"

main() {
    if (init() == OK)
    {
        dbupdate();
        store_dbmf();
    }
}
struct ci /*currency information for record or set types*/
{
    int fd;
    long log_fo;
    long phy_fo;
    long final_fo;
    int log_bo;
    int phy_bo;
    int final_bo;
    int add_on;
    char file_buf[BLOCKSIZE];
};

struct rec_type /*structure for information for a record type*/
{
    char name[MAX_TNAME + 1];
    char fdlim; /*field delimiter for this record type*/
    int nosfield;
    int noskey;
    int keyf[MAX_KEY_FIELDS];
    struct ci rci; /*record currency information*/
    struct rec_type *left;
    struct rec_type *right;
};

struct set_type /*current of set type*/
{
    char name[MAX_TNAME + 1];
    struct rec_type *owner;
    struct rec_type *member;
    char cos [MAX_KEY + 1];
    int sbfd;
    long sbtroot;
    struct ci sci; /*set currency information*/
    struct set_type *sleft;
    struct set_type *sright;
};

struct ptrs
{
<table>
<thead>
<tr>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>definitions 101</td>
<td>1</td>
<td>long btp; /<em>struct used for communication with btree routines</em>/</td>
</tr>
<tr>
<td>definitions 102</td>
<td>1</td>
<td>int bts;</td>
</tr>
<tr>
<td>definitions 103</td>
<td>1</td>
<td>long btd;</td>
</tr>
<tr>
<td>definitions 104</td>
<td>1</td>
<td>int btb;</td>
</tr>
<tr>
<td>definitions 105</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>definitions 106</td>
<td>0</td>
<td><em>/Global Declarations</em>/</td>
</tr>
<tr>
<td>definitions 107</td>
<td>0</td>
<td>struct rec_type *rt_root = NULL; /<em>root for record type</em>/</td>
</tr>
<tr>
<td>definitions 108</td>
<td>0</td>
<td>struct rec_type *cor_rt = NULL; /<em>current of run record type</em>/</td>
</tr>
<tr>
<td>definitions 109</td>
<td>0</td>
<td>struct set_type *st_root = NULL; /<em>root for set type</em>/</td>
</tr>
<tr>
<td>definitions 110</td>
<td>0</td>
<td>struct set_type *cor_st = NULL; /<em>current of run set type</em>/</td>
</tr>
<tr>
<td>definitions 111</td>
<td>0</td>
<td>char cor_unit[MAX_KEY+1]; /<em>current of run unit key</em>/</td>
</tr>
<tr>
<td>definitions 112</td>
<td>0</td>
<td>int cnt_rt=0; /<em>number of record types in db</em>/</td>
</tr>
<tr>
<td>definitions 113</td>
<td>0</td>
<td>int cnt_st=0; /<em>number of set types in db</em>/</td>
</tr>
<tr>
<td>definitions 114</td>
<td>0</td>
<td>long rbtroot; /<em>root for record btree</em>/</td>
</tr>
<tr>
<td>definitions 115</td>
<td>0</td>
<td>int rbtdf; /<em>file descriptor for record btree</em>/</td>
</tr>
</tbody>
</table>
/*This file contains common declarations for the parse routines.*/

<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_c_dec</td>
<td>1</td>
<td>0</td>
<td>char key[MAX_KEY + 1];</td>
</tr>
<tr>
<td>p_c_dec</td>
<td>2</td>
<td>0</td>
<td>char sname[MAX_TNAME + 1];</td>
</tr>
<tr>
<td>p_c_dec</td>
<td>3</td>
<td>0</td>
<td>char rname[MAX_TNAME + 1];</td>
</tr>
<tr>
<td>p_c_dec</td>
<td>4</td>
<td>0</td>
<td>char fname[MAX_TNAME + 1];</td>
</tr>
<tr>
<td>p_c_dec</td>
<td>5</td>
<td>0</td>
<td>extern struct set_type *cor_st;</td>
</tr>
<tr>
<td>p_c_dec</td>
<td>6</td>
<td>0</td>
<td>extern struct rec_type *cor_rt;</td>
</tr>
<tr>
<td>p_c_dec</td>
<td>7</td>
<td>0</td>
<td>extern char cor_unit[MAX_KEY + 1];</td>
</tr>
<tr>
<td>p_c_dec</td>
<td>8</td>
<td>0</td>
<td>struct rec_type *prname(), *pr;</td>
</tr>
<tr>
<td>p_c_dec</td>
<td>9</td>
<td>0</td>
<td>struct set_type *psname(), *ps;</td>
</tr>
<tr>
<td>p_c_dec</td>
<td>10</td>
<td>0</td>
<td>int spec;</td>
</tr>
</tbody>
</table>
file
parse_dub
  line
  level
  source
parse_dub
  1  0  /*Dbupdate parses user commands and calls appropriate subroutines*/
parse_dub
  2  0  dbupdate()
parse_dub
  3  0  { int more=TRUE;
parse_dub
  4  0  char cmd[MAX_CMD + 21];
parse_dub
  5  0  int i,cond,status;
parse_dub
  6  0  char c;
parse_dub
  7  1  #ifdef DEBUG
parse_dub
  8  1  printf("dbupdate\n");
parse_dub
  9  1  #endif
parse_dub
  10  1  cond=OK;
parse_dub
  11  1  while (more)
parse_dub
  12  1  {
parse_dub
  13  1  printf("%c", PROMPT);
parse_dub
  14  1  if (scanf(FMT2, cmd) != NULL)
parse_dub
  15  1  switch (*cmd)
parse_dub
  16  1  { case 'q': /*end of update*/
parse_dub
  17  1  break;
parse_dub
  18  1  case 'r': /*record type add*/
parse_dub
  19  1  status = prcc_add();
parse_dub
  20  1  break;
parse_dub
  21  1  case 'a': /*set type add*/
parse_dub
  22  1  status = psec_add();
parse_dub
  23  1  break;
parse_dub
  24  1  case 'o': /*owner add*/
parse_dub
  25  1  status = pownadd();
parse_dub
  26  1  break;
parse_dub
  27  1  case 'm': /*member add*/
parse_dub
  28  1  status = membadd();
parse_dub
  29  1  break;
parse_dub
  30  1  case 'r': /*record add*/
parse_dub
  31  1  status = precadd();
parse_dub
  32  1  break;
parse_dub
  33  1  default:
parse_dub
  34  1  cond = ERROR;
parse_dub
  35  1  break;
parse_dub
  36  1  } break;
case 'c':
    status = pchange(*cmd+1); /* parse change command*/
    break;
  case 'd':
    switch (*cmd+1)
    {
      case 'm': /* delete member*/
          status = pdel(*cmd+1);
          break;
      case 'o': /* delete owner*/
          status = pdel(*cmd+1);
          break;
      case 'r': /* delete record*/
          status = precdel();
          break;
      default:
          cond = ERROR;
          break;
    }
    break;
  case 'f':
    switch (*cmd+1)
    {
      case 'r': /* find record*/
          status = precfind();
          break;
      default:
          status = pfind(*cmd+1);
          break;
    }
    break;
  default:
      cond = ERROR;
      break;
}
else
  cond = ERROR;
if (cond == ERROR)
{
    printf("illegal command\n");
    cond = OK;
    status = LNP; /* full line not parsed*/
}
if (status != LP)
   while ((c = getc(stdin)) != END_OF_LINE)
     ; /* scan the rest of the line*/
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse_dbup</td>
<td>101</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>parse_dbup</td>
<td>102</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>parse_dbup</td>
<td>103</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>parse_dbup</td>
<td>104</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```c
file

int nosfld, noskeyf;
int keyf[MAX_KEY_FIELDS];
struct rec_type *prname();
char rntname[MAX_TNAME + 1];
char fdln;
int i;
int spec;

#ifdef DEBUG
printf("e prect_add\n");
#endif

if (prname(rntname, &spec) == NULL && spec != PRESENT)
    return(error("usage: add new-rec-type-name #fields #key-fields list-key-fields"));
else if (jumpp() != PRESENT || scanf(" %c , %fdln) == NULL)
    return(error("missing field delimiter");)
else
    /*read and validate key field information*/
    scanf(" %d %d", nosfld, noskeyf);
    if (noskeyf > MAX_KEY_FIELDS || noskeyf < 1 || noskeyf > nosfld)
        return(error("illegal number of key fields");)
    else if (nosfld < 1)
        return(error("illegal number of fields");)
    else
        for (i=0; i<MAX_KEY_FIELD; i++)
            *(keyf+i) = 0; /*zero key fld array*/
        for (i=0; i<noskeyf; i++)
            scanf(" %d", keyf+i);
        if (*(keyf+i) < 1 || *(keyf+i) > nosfld)
            return(error("illegal key field specification");)
    return(rect_add(rntname, fdln, nosfld, noskeyf, keyf)); /*add the record ty;*/
```
/*Psett_add parses the command line for a set type add command, gathering 
information. If the information is complete, sett_add is called.*/

psett_add()
{
    char stname[MAX_TNAME + 1];
    char owner[MAX_TNAME + 1], member[MAX_TNAME + 1];
    struct set_type *psname();
    struct rec_type *prname();
    int spec;

    #ifdef DEBUG
    printf("e psett_add\n");
    #endif

    if(psname(stname, &spec) == NULL & & spec != PRESENT)
        return(error("usage: sa new-set-name owner-name member-name"));
    else
    {
        /*scan owner and member names*/
        if (prname(owner, &spec) == NULL & & spec != PRESENT ||
            prname(member, &spec) == NULL & & spec != PRESENT)
            return(error("missing owner or member name"));
        else
                return(sett_add(stname, owner, member)); /*add new set type*/
    
}
parse_db up
Mon Sep 27 14:43:37 1982

file
line level source

parse_db up 191 0 /*Pownadd parses the command line for an owner add command, sets the current
parse_db up 192 0   of run pointers, and calls ownadd*/
parse_db up 193 0
parse_db up 194 0
parse_db up 195 0
parse_db up 196 0 {pownadd()}
parse_db up 197 1 /*include common parse declarations*/
parse_db up 198 1
parse_db up 199 1 ifdef DEBUG
parse_db up 200 1 printf("e pownadd\nn");
parse_db up 201 1 #endif
parse_db up 202 1
parse_db up 203 1
parse_db up 204 1 if ((ps = pnsname(sname,&spec)) == NULL &
parse_db up 205 1 spec 1= PRESENT) /*scan the set type name*/
parse_db up 206 1 return(error( missing set name"));
parse_db up 207 1 else if (ps == NULL)
parse_db up 208 1 return(error("set type not defined"));
else if (pkey(key) == ERROR) /*scan occurrence key*/
parse_db up 209 1 return(ERROR);
parse_db up 210 1 else
parse_db up 211 1 {cor_st = ps; /*set the cor pointers*/
parse_db up 212 1 strcpy(cor_unit,key);
parse_db up 213 2 return(ownadd()); /*add an owner set occurrence*/
parse_db up 214 2
parse_db up 215 2
parse_db up 216 2}
parse_db up 217 1
parse_db up 218 0
parse_db up 219 0
parse_db up 220 0
parse_db up 221 0
parse_db up 222 0
parse_dbup  Mon Sep 27 14:43:37 1982  Page 7

file line level source
parse_dbup 224 0 /*pmebadd parses the command line for an add member occurrence command, sets
parse_dbup 225 0 the current of run pointers, and calls member add.*/
parse_dbup 226 0
parse_dbup 227 0
parse_dbup 228 0 pmebadd()
parse_dbup 229 0 {
parse_dbup 230 1 #include "p_c_dec"
parse_dbup 231 1 /*common parse declarations*/
parse_dbup 232 1
parse_dbup 233 1 #ifdef DEBUG
parse_dbup 234 1 printf("e pmebadd\n");
parse_dbup 235 1 #endif
parse_dbup 236 1
parse_dbup 237 1
parse_dbup 238 1 if (pkey(mkey) == ERROR) /*parse member key*/
parse_dbup 239 1 return(ERROR);
parse_dbup 240 1 /*parse record type name*/
parse_dbup 241 1 else if (((ps = psname(sname,&spec)) == NULL && spec != PRESENT)
parse_dbup 242 1 return(error("missing record type"));
parse_dbup 243 1 else if (ps == NULL)
parse_dbup 244 1 return(error("set type not defined"));
parse_dbup 245 1 else if (pkey(key) == ERROR) /*parse owner key*/
parse_dbup 246 1 return(ERROR);
parse_dbup 247 1 else
parse_dbup 248 1 {
parse_dbup 249 2 cor_st = ps; /*set currency pointers*/
parse_dbup 250 2 strcpy(cor_unit,key);
parse_dbup 251 2
parse_dbup 252 2
parse_dbup 253 1
}
parse_dbup 254 0
parse_dbup 255 0
parse_dbup 256 0
parse_dbup 257 0
parse_dbup 258 0
parse_dbup Mon Sep 27 14:43:37 1962 Page 8

file
line level source

parse_dbup 260 0 /*Precadd parses the command line for a record occurrence add, sets the
parse_dbup 261 0 currency pointers, and calls recadd.*/
parse_dbup 262 0
parse_dbup 263 0
parse_dbup 264 0 precadd()
parse_dbup 265 0 {
parse_dbup 266 1 /*FILE *ifp;*/
parse_dbup 267 1 #include "p_c_dec.c"
parse_dbup 268 1 /*common parse definitions*/
parse_dbup 269 1
parse_dbup 270 1
parse_dbup 271 1 #ifdef DEBUG
parse_dbup 272 1 printf( "e precadd\n" );
parse_dbup 273 1 #endif
parse_dbup 274 1
parse_dbup 275 1 /*parse record type name*/
parse_dbup 276 1 if ((pr = prname(rname,&spec)) == NULL & & spec != PRESENT)
parse_dbup 277 1 return(error("missing record type"));
parse_dbup 278 1 else if (pr == NULL)
parse_dbup 279 1 return(error("record type not defined"));
parse_dbup 280 1 else if (pfname (fname, READ, &ifp) == ERROR) /*parse file name*/
parse_dbup 281 1 return(Err);
parse_dbup 282 1 else
parse_dbup 283 1 {
parse_dbup 284 2 /*set currency pointers*/
parse_dbup 285 2 cor rt = pr;
parse_dbup 286 2 while ((c = getc(stdin)) != END_OF_LINE)
parse_dbup 287 2 recadd(ifp); /*add record occurrence*/
parse_dbup 288 2 return(LP); /*have read past end of line*/
parse_dbup 289 2
parse_dbup 290 1 }
parse_dbup 291 0
parse_dbup 292 0
parse_dbup 293 0
parse_dbup 294 0
parse_dbup 295 0
/*precde parses the command line for a record occurrence delete, sets the
   currency pointers, and calls record occurrence delete.*/

precde()
{
#include "p_c_dec"

#ifdef DEBUG
    printf("e precde\n");
#else

    if ((pr = prname(rnameA&spec)) == NULL && spec != PRESENT)
        return(error("missing record type"));

    else if (pr == NULL)
        return(error("record type not defined"));

    else if (pkey (key) == ERROR)
        return(ERROR);

    else
    {
        cor_rt = pr; /*set currency pointers*/
        strcpy(cor_unit,key); /*delete record occurrence*/
        return (recdel());
    }

#endif
}
parse_dbup  Mon Sep 27 14:43:37 1982  Page 10

file line level source

parse_dbup  330  0  /*Pdel parses a command line for an owner or member occurrence delete, sets
parse_dbup  331  0  the currency pointers, and calls owner deleting or member deleting depending
parse_dbup  332  0  on the cmd parameter.*/
parse_dbup  333  0
parse_dbup  334  0
parse_dbup  335  0  pdel(cmd)
parse_dbup  336  0  char cmd;
parse_dbup  337  0  {
parse_dbup  338  1  #include "p_c_dec"  /*common parse declarations*/
parse_dbup  339  1
parse_dbup  340  1  ifdef DEBUG
parse_dbup  341  1  printf("e pdel\n");
parse_dbup  342  1  endif
parse_dbup  343  1
parse_dbup  344  1
parse_dbup  345  1  if ((ps = psname(sname,&spec)) == NULL & spec ! PRESENT)
parse_dbup  346  1  return(error("missing record type");
parse_dbup  347  1  else if (ps == NULL)
parse_dbup  348  1  return(error("set name undefined");
parse_dbup  349  1  else if (pkey(key) == ERROR)
parse_dbup  350  1  return(ERROR);
parse_dbup  351  1  else
parse_dbup  352  1  {
parse_dbup  353  2  cor_st = ps;  /*set currency pointers*/
parse_dbup  354  2  strcpy (cor_unit, key);
parse_dbup  355  2  if (cmd == "o")  /*call owner occurrence delete*/
parse_dbup  356  2  return (owndel());
parse_dbup  357  2  else
parse_dbup  358  2  return (membdel(NEWL,NEW));
parse_dbup  359  2  }
parse_dbup  360  1  }
parse_dbup  361  0
parse_dbup  362  0
parse_dbup  363  0
parse_dbup  364  0
parse_dbup  365  0
file

line   level   source

parse_dbuf  367     0  /*Preceed parses the command line for a find record occurrence command, sets
                currency pointers, and call record find*/
parse_dbuf  368     0
parse_dbuf  369     0
parse_dbuf  370     0
parse_dbuf  371     0  precfind()
parse_dbuf  372     0  
        {  
        
        #include "pcdec"
        FILE *ofp;
        parse_dbuf  375     1
        parse_dbuf  376     1
        ifdef DEBUG
        parse_dbuf  377     1  printf ("e precfind\n");
        endif
        parse_dbuf  378     1
        parse_dbuf  379     1
        parse_dbuf  380     1
        parse_dbuf  381     1  /*parse record type name*/
        parse_dbuf  382     1
        parse_dbuf  383     1
        parse_dbuf  384     1
        parse_dbuf  385     1  if ((pr = prname(rname,&spec)) == NULL & &spec != PRESENT)
                return(error("missing record type"));
        else if (pr == NULL)
                return(error("record type not defined"));
        parse_dbuf  386     1
        parse_dbuf  387     1  /*parse key and optional file name*/
        parse_dbuf  388     1
        parse_dbuf  389     1  else
        parse_dbuf  390     1  {
                cor Rt = pr;  /*set currency pointers*/
                strcpy (cor unit,key);
        parse_dbuf  391     2
        parse_dbuf  392     2
        parse_dbuf  393     2
        parse_dbuf  394     2
        parse_dbuf  395     1
        parse_dbuf  396     0
        parse_dbuf  397     0
        parse_dbuf  398     0
        parse_dbuf  399     0
        parse_dbuf  400     0
/*Pfind parses a command line for a find owner, a find first member, or a find next command, sets currency pointers, and appropriate find routines.*/

pfind (cmd)
char cmd;
{
  #include "p_cdec"
  FILE *ofp;        /*output file descriptor*/
  
  #ifndef DEBUG
    printf ("e pfind\n");
  #endif
  
  /*parse set name, key, and optional output file name*/
  if ((ps = psname(sname, &spec)) == NULL || spec != PRESENT)
    return(error("missing set type"));
  else if (ps == NULL)
    return(error("set type undefined"));
  else if (cmd == 'n')
    
    /*handle find next command*/
    
    cor_st = ps;
    if (pfname(fname, APPEND, &ofp) == ERROR)
      return (ERROR);
    else
      return (nextfind (ofp));
  else if (pkey(key) == ERROR || pfname(fname, APPEND, &ofp) == ERROR)
    return(ERROF);
  else
    {cor_st = ps;    /*set currency pointers*/
     strcpy (cor_unit, key);
     switch (cmd)
     {
       case 'f':    /*find first member*/
         return (mebfind(ofp));
       case 'o':
         return (ownfind(ofp));
       default:
         return(error("illegal command"));
     }
    }
  
  return (0);
}
file

line level source

```c
parse_dbup 453 0 /*Pchange parses the command line for a change all or change owner command,
   sets the currency pointers, and calls the appropriate change routine.*/
parse_dbup 454 0 
pchange (cmd)
parse_dbup 455 0 char cmd;
parse_dbup 456 0 
parse_dbup 457 0 { /*new owner/member key*/
parse_dbup 458 0 
parse_dbup 459 1 #include "p_c_dec"
parse_dbup 460 1 char newokey[MAX_KEY + 1];
parse_dbup 461 1 
parse_dbup 462 1 #ifdef DDEBUG
parse_dbup 463 1 printf ("e pchange\n");
parse_dbup 464 1 #endif
parse_dbup 465 1 
parse_dbup 466 1 if (pkey(newokey) == ERROR) /*parse new owner/member key*/
parse_dbup 467 1 return(ERROR);
parse_dbup 468 1 
parse_dbup 469 1 else if ((ps = psname(sname,&spec)) == NULL & spec != PRESENT)
parse_dbup 470 1 return(error("missing set type name"));
parse_dbup 471 1 else if (ps == NULL)
parse_dbup 472 1 return(error("set type undefined"));
parse_dbup 473 1 else if (pkey(key) == ERROR) /*parse old owner/member key*/
parse_dbup 474 1 return(ERROR);
parse_dbup 475 1 
parse_dbup 476 1 else
parse_dbup 477 2 
parse_dbup 478 2 
parse_dbup 479 2 cor_st = ps; /*set currency pointers*/
parse_dbup 480 2 strcpy(cor_unit,key); /*set at old owner/member*/
parse_dbup 481 2 switch(cmd)
parse_dbup 482 2 { /*call appropriate change routine*/
parse_dbup 483 3 case 'a': /*change all members to new owner*/
parse_dbup 484 3 return (cngall(newokey));
parse_dbup 485 3 case 'o': /*change member to new owner*/
parse_dbup 486 3 return (cngown(newokey));
parse_dbup 487 3 default:
parse_dbup 488 3 return(error("illegal command"));
parse_dbup 489 3 
parse_dbup 490 2 
parse_dbup 491 1 }
parse_dbup 492 0 
parse_dbup 493 0 parse_dbup 494 0 
parse_dbup 495 0 parse_dbup 496 0 /*parse utility routines*/
parse_dbup 497 0 
parse_dbup 498 0 parse_dbup 499 0
```
/*Prtme scans standard input for a string, truncates the string to MAX_NAME
and places it in string passed in as a parameter. Prtme searches the
record type tree for the record name indicated by the string.
prtme returns: NULL if the string is empty
NULL if the string is not a name in the record type tree
pointer to record type if rname is in the record type tree*/

struct rec_type *prtme(rname, spec)
char *rname;
int *spec;

extern struct rec_type *rt_root;
struct rec_type *rt_search();

#ifdef DEBUG
printf("e prtme \n\n");
#endif

if(*spec = jumpup()) == PRESENT)
{  printf(FMT1, rname);
}
else   return(NULL);
parse_dbup Mon Sep 27 14:43:37 1982  Page 15

file line level source

parse_dbup 534 0 /*Psname scans standard input for a string, truncates the string to MAX_TNAME,
parse_dbup 535 0 and places the string into the parameter passed into the routine. Psname
parse_dbup 536 0 searches the set type tree for the set name indicated by the string.
parse_dbup 537 0 psname returns: NULL if the string is empty
parse_dbup 538 0 NULL if the string is not a name in the set tree
parse_dbup 539 0 pointer to set type if name is in set tree*/
parse_dbup 540 0
parse_dbup 541 0
parse_dbup 542 0 struct set_type *psname (stname,spec)
parse_dbup 543 0 char *stname;
parse_dbup 544 0 int *spec;
parse_dbup 545 0 {
parse_dbup 546 1    extern struct set_type *st_root;
parse_dbup 547 1    struct set_type *st_search();
parse_dbup 548 1
parse_dbup 549 1    #ifdef DEBUG
parse_dbup 550 1    printf("entering set type parse
"));
parse_dbup 551 1    #endif
parse_dbup 552 1
parse_dbup 553 1    if (%spec = jumpup()) == PRESENT)
parse_dbup 554 1    {
parse_dbup 555 2        scanf(PMT1, stname); /*scan name*/
parse_dbup 556 2        return(st_search(st_root, stname));
parse_dbup 557 2    }
parse_dbup 558 1 else
parse_dbup 559 1    return(NULL);
parse_dbup 560 1
parse_dbup 561 0
parse_dbup 562 0
parse_dbup 563 0
parse_dbup 564 0
parse_dbup 565 0
parse_dbup 567 0 /*Pfname scans a file name. If the file name is null, fp is set to stdin
parse_dbup 568 0 or stdout depending on the mode. Otherwise, the file is opened.*/
parse_dbup 569 0
parse_dbup 572 0 pfname (fname, mode, fp)
parse_dbup 571 0 char *mode;
parse_dbup 572 0 char *fname;
parse_dbup 573 0 FILE **fp;
parse_dbup 574 0 {
parse_dbup 575 1 FILE *fopen();
parse_dbup 576 1
parse_dbup 577 1 #ifdef DEBUG
parse_dbup 578 1 printf("entering parse file name\n");
parse_dbup 579 1 #endif
parse_dbup 580 1
parse_dbup 581 1 if (jumpup (!= PRESENT))
parse_dbup 582 1 if (strcmp (mode, READ) == MATCH)
parse_dbup 583 1 *fp = stdin;
parse_dbup 584 1 else *fp = stdout;
parse_dbup 585 1 else
parse_dbup 586 1 {
parse_dbup 587 2 scanf (FMT1, fname);
parse_dbup 588 2 if ((*fp = fopen (fname, mode)) == NULL)
parse_dbup 589 2 return (error ("unable to open file"));
parse_dbup 590 2 }
parse_dbup 591 2
parse_dbup 592 1 return (OK);
/*pkey parses an input key. If key entered is too long an error message is
   returned. Otherwise, OK is returned.*/

char *key;
{
    char inkey[MAX_KEY + 6];

    #ifdef DEBUG
    printf ("entering pkey\n");
    #endif

    if (jumpup() == PRESENT)
    {
        scanf(FMT2, inkey);
        if (strlen(inkey) > MAX_KEY)
            return(error("input key too long"));
        else
        {
            strcpy(key, inkey);
            return (OK);
        }
    }
    else
        return(error("missing key"));
/*Jumpup scans blanks and tabs, and returns Present(1) if the next character is not a new line and NOTPRESENT if the next character is a new line.*/

```c
jumpup()
{
    char c;

    #ifdef DEBUG
    printf("e jumpup\n");
    #endif

    while (((c = getchar(stdin)) == BLANK) || (c == TAB))
        /*skip blanks and tabs*/
    ungetc(c, stdin);

    if (c == '\n')
        return (NOTPRESENT);
    else
        return (PRESENT);
}
```
file

rec_t 1 0 /*This file contains the routines to build the record type tree, insert a
rec_t 2 0 record type into the tree, and search the tree for a particular record type.*/
rec_t 3 0
rec_t 4 0 /*MAD 10-81*/
rec_t 5 0
rec_t 6 0 /*Brec_tree reads the data base management file and builds a tree contain-
rec_t 7 0 ing the information on the different record types in the data base*/
rec_t 8 0
rec_t 9 0 brec_tree (fp,number)
rec_t 10 0 FILE *fp;
rec_t 11 0 int number;
rec_t 12 0{
rec_t 13 1 int i;
rec_t 14 1
rec_t 15 1 #ifdef DEBUG
rec_t 16 1 printf("entering brec_tree\n");
rec_t 17 1 #endif
rec_t 18 1
rec_t 19 1
rec_t 20 1 for (i=1; i<= number; i++)
rec_t 21 1 if (arec_tt (fp) == ERROR)
rec_t 22 1 return(OK);
rec_t 23 1 if (arec_tt (fp) == ERROR)
rec_t 24 0 return(OK);
rec_t 25 0
rec_t 26 0
rec_t 27 0
rec_t Mon Sep 27 14:43:58 1982 Page 2

file line level source
rec_t 29 0 /*Rec_tt will scan the dbm file and build a record type and place it in
rec_t 30 0 the record type b-tree. Returns -1 if error occurs.*/
rec_t 31 0
rec_t 32 0 arec_tt (fp)
rec_t 33 0 FILE*fp; /*dbm file pointer*/
rec_t 34 0 {
  int i,cond;
  extern struct rec_type *rt_root;
  rec_t 37 1 struct rec_type rt;
  rec_t 38 1 struct rec_type *rt_insrt();
  rec_t 39 1 /*read information for one record type*/
  rec_t 40 1 fscanf(fp,"%s %c %d %d", r.name, &r.fdlim, &r.nosfield, &r.noskey);
  rec_t 41 1 for (i=0; i<r.noskey; i++)
    rec_t 42 1 fscanf(fp,"%d", r.keyf+i);
  rec_t 43 1 if ((r.rci.fd = openfile(r.name, RT)) == ERROR)
    rec_t 44 1 return(ERROR);
  rec_t 45 1 else
    rec_t 46 1 {
      fscanf(fp,"%d %d
",&r.rci.final_fs,&r.rci.final_be);
      rec_t 48 2 /*add node to tree*/
      rt_root = rt_insrt(rt_root, &r, &cond);
      rec_t 50 2 if (cond == MATCH)
        rec_t 52 2 return(error("unable to build record type tree"));
      rec_t 53 2 else
        rec_t 54 2 return(OK);
      rec_t 55 2 }
    rec_t 56 1 }
  rec_t 57 0
  rec_t 58 0
  rec_t 59 0
  rec_t 60 0
  rec_t 61 0
```c
/*rt_insert inserts a record type into the record type b-tree. If already present, condition is set to error. A pointer to the root is returned if record type is inserted.*/

struct rec_type *rt_insert(rec_t, r, cond)
struct rec_type *rec;
int *cond;

{ #ifdef DEBUG
    printf("rt_insert\n");
    #endif

    if(rec == NULL)
    {
        rec = RTALLOC;
        strcpy(rec->name, r->name);
        rec->fdlim = r->fdlim;
        rec->nosfield = r->nosfield;
        rec->oskeyf = r->oskeyf;
        rec->arrpy(r->keyf, r->keyf, MAX_KEY_FIELDS);
        rec->rci.fi = r->rci.fd;
        rec->rci.logfo = 2L;
        rec->rci.phyfo = 0L;
        rec->rci.finalfo = r->rci.finalfo;
        rec->rci.logbo = 0;
        rec->rci.phybo = 0;
        rec->rci.finalbo = r->rci.finalbo;
        rec->rci.add_on = FALSE;
        rec->left = prec->right = NULL;
        *cond = NO_MATCH;
    }

    else if (*cond = strcmp(r->name, prec->name)) == MATCH
        error("record type already defined");
    else if (*cond < 0)
        /*go to left subtree*/
        prec->left = rt_insert(prec->left, r, cond);
    else
        /*go to right subtree*/
        prec->right = rt_insert(prec->right, r, cond);

    return(prec);
```
/*Rect_add will add a record type to the data base*/

rect_add (rtname, rfdlim, nosfld, noskeyf, keyf)
char *rtname;
char rfdlim;
int nosfld, noskeyf;
int *keyf;
{
    struct rec_type r;
    int i, cond;
    extern struct rec_type *rt_root;
    extern cnt_rt;
    extern long rbtroot;
    extern int rbtfd;
    long initbt();
    extern int verbose = NONE;

    struct rec_type *rt_insert();
    struct rec_type *rt_search();

    #ifdef DEBUG
    printf("e rect_add\n");
    #endif

    if (cnt_rt == NONE)
    {
        /*create btree for record keys and offsets*/
        if((rtfd = creat(RBT, PMODE)) == ERROR)
            return(error("unable to create dbm.rec.bt file"));
        buildfl(rbtfd, SIZE, verbose);
        close(rbtfd);
        if((rbtd = open (RBT, RW)) == ERROR)
            return(error("unable to open dbm.rec.bt file");
        rbtroot = initbt(rbtfd);
    }
    if (rt_search(rt_root,"rtname") != NULL)
    return(error("record type already defined");
    if ((r.rcl.fd = create_file(rtname, RT)) == ERROR)
        return(ERROR);
    else
    {
        strcpy(r.name, rtname);
        r.fldlim = rfdlim;
        r.nosfield = nosfld;
        r.noskeyf = noskeyf;
        arccpy(r.keyf, keyf, MAX_KEY_FIELDS);
        r.rcl.final_fo = 0;
        r.rcl.final_bo = 0;
        rt_root = rt_insert(rt_root, &r, &cond);
        if(cond == MATCH)
            return(ERROR);
        else
            return(ERROR);
rec_t  Mon Sep 27 14:43:50 1982  Page 5

<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec_t</td>
<td>159</td>
<td>2</td>
<td>{}</td>
</tr>
<tr>
<td>rec_t</td>
<td>162</td>
<td>3</td>
<td>cnt_r += 1;</td>
</tr>
<tr>
<td>rec_t</td>
<td>161</td>
<td>3</td>
<td>return(OK);</td>
</tr>
<tr>
<td>rec_t</td>
<td>162</td>
<td>3</td>
<td>}</td>
</tr>
<tr>
<td>rec_t</td>
<td>163</td>
<td>2</td>
<td>}</td>
</tr>
<tr>
<td>rec_t</td>
<td>164</td>
<td>1</td>
<td>}</td>
</tr>
<tr>
<td>rec_t</td>
<td>165</td>
<td>0</td>
<td>}</td>
</tr>
<tr>
<td>rec_t</td>
<td>166</td>
<td>0</td>
<td>}</td>
</tr>
<tr>
<td>rec_t</td>
<td>167</td>
<td>0</td>
<td>}</td>
</tr>
<tr>
<td>rec_t</td>
<td>168</td>
<td>0</td>
<td>}</td>
</tr>
<tr>
<td>rec_t</td>
<td>169</td>
<td>0</td>
<td>}</td>
</tr>
</tbody>
</table>
/*rt_search searches the record_type b-tree for a specific occurrence of a
record type name. If present, a pointer to the set type is returned.
Otherwise, null is returned*/

struct rec_type *rt_search(prec, rname)
struct rec_type *prec;
char *rname;
{
    int cond;

    ifdef DEBUG
    printf("e rt_search\n");
    endif

    if (prec == NULL)
        return(NULL);
    else if ((cond = strcmp(rname, prec->name)) == MATCH)
        return(prec);    /*name found - return pointer*/
    else if (cond < 0)    /*look in left sub tree*/
        return(rt_search(prec->left, rname));
    else    /*look in right sub tree*/
        return(rt_search(prec->right, rname));
}

rec_t 171 0
rec_t 172 0
rec_t 173 0
rec_t 174 0
rec_t 175 0
struct rec_type *rt_search(prec, rname)
rec_t 176 0
struct rec_type *prec;
rec_t 177 0
char *rname;
rec_t 178 0
    int cond;
rec_t 179 1
rec_t 180 1
rec_t 181 1
ifdef DEBUG
printf("e rt_search\n");
endif
rec_t 183 1
rec_t 184 1
rec_t 185 1
    if (prec == NULL)
        return(NULL);
    else if ((cond = strcmp(rname, prec->name)) == MATCH)
        return(prec);    /*name found - return pointer*/
    else if (cond < 0)    /*look in left sub tree*/
        return(rt_search(prec->left, rname));
    else    /*look in right sub tree*/
        return(rt_search(prec->right, rname));
}

/*Rt_write saves information on the record types by writing it to the DEMF.*/

rt_write(fp, prec, cnt)
  FILE *fp;
  struct rec_type *prec;
  int *cnt;
  
  int i;

#ifndef DEBUG
  printf("e rt_write\n");
#endif

if (prec == NULL)
  return;
else
  
  if (prec->rc1.add_on) /*flush buffer*/
    {write_buf(prec->rc1.fd, prec->rc1.final_fo, prec->rc1.file_buf,
      prec->rc1.final_bo);
     printf(fp, "%s %c %d %d", prec->name, prec->fdlim, prec->nosfield, prec->noskeyf);
     for (i=0; i<prec->noskeyf; i++)
       printf(fp, "%d ", (prec->keyf+i));
     printf(fp, "%d %d\n", prec->rc1.final_fo, prec->rc1.final_bo);
     *cnt += 1;
    }
  rt_write(fp, prec->left, cnt);
  rt_write(fp, prec->right, cnt);
  return;
}
file

line level source

set_t 1 0 /*This file contains the routines to build the set type tree, insert a set
set_t 2 0 type into the tree, and search the tree for a particular set type.
set_t 3 0 A routine to write the information in the set type tree out to the DBM file
set_t 4 0 is included. */
set_t 5 0 /*MAD 10-81*/
set_t 6 0 /*bset_tree reads the data base management file and builds a tree
set_t 7 0 containing the information on the different set types in the data base*/
set_t 8 0 bset_tree (fp, number)
set_t 9 0 int *fp;
set_t 10 0 int number;
set_t 11 0 {
set_t 12 1 int i;
set_t 13 0 #ifdef DEBUG
set_t 14 0 printf( "e bset_tree\n\n" );
set_t 15 0 #endif
set_t 16 0 for ( i = 1; i <= number; i++ )
set_t 17 0 if ( aset_tt (fp) == ERROR )
set_t 18 0 return(OK);
set_t 19 0 }
set_t 20 0
file line level source
set_t 32 0 /*Aset tt will scan the dbm file and build a set type and place it in the
set_t 33 0 set type b-tree. Returns -1 if error occurs*/
set_t 34 0
set_t 35 0 aset tt (fp) /*dbm file pointer*/
set_t 36 0 FILE* fp;
set_t 37 0 {
  int 1, cond;
  set_t 39 1 char nowner[MAX_TNAME + 1];
  set_t 40 1 char nmember[MAX_TNAME + 1];
  set_t 41 1 extern struct set_type *st_root;
  set_t 42 1 extern struct rec_type *rt_root;
  set_t 43 1 struct set_type s;
  set_t 44 1 struct set_type *st_insert();
  set_t 45 1 struct rec_type *rt_search();
  set_t 46 1
  set_t 47 1 #ifdef DEBUG
  set_t 48 1 printf("e aset tt\n");
  set_t 49 1 #endif
  set_t 50 1
  set_t 51 1 /\read information for one set type*/
  set_t 52 1 fscanf(fp,"%s %s %d %d\n", s.sname, nowner, nmember, &s.scl.final_fo,
              &s.scl.final_bo);
  set_t 53 1 if((s.owner = rt_search(rt_root, nowner)) == NULL ||
              (s.member = rt_search(rt_root, nmember)) == NULL)
    return(error("unable to build set type - owner/member not a record type"));
  set_t 54 1 if ((s.scl.fd = openfile(s.sname, ST)) == ERROR ||
              (s.sbtfd = openfile(s.sname, BT)) == ERROR)
    return(ERROR);
  set_t 55 1 else
    set_t 56 2 { st_root = st_insert(st_root, &s, &cond); 
      set_t 57 2 if("cond == MATCH"
        set_t 58 3 return(error("unable to build set type tree"));
          set_t 59 4 else
            return(OK);
          set_t 60 5 }
  set_t 61 1 }
  set_t 62 2
  set_t 63 2
  set_t 64 2
  set_t 65 2
  set_t 66 2
  set_t 67 2
  set_t 68 1 }
set_t 69 0
set_t 70 0
set_t 71 0
set_t 72 0
set_t 73 0
/*St_insert inserts a new set type into the set type b-tree. If already present, condition is set to match(0). A pointer to the root is returned*/

struct set_type *st_insert(pset, s, cond)
struct set_type *pset;
struct set_type *s;
int *cond;

long initbt();
if (pset == NULL)
{
    /*build and add a new set type*/
    pset = malloc;
    strcpy(pset->snorm, s->name);
    pset->owner = s->owner;
    pset->member = s->member;
    pset->cos = NULL;
    pset->sci.fd = s->sci.fd;
    pset->sbtf = s->sbtf;
    pset->sbsroot = initbt(s->sbts);
    pset->sci.log_fo = 0;
    pset->sci.phy_fo = 0;
    pset->sci.final_fo = 0;
    pset->sci.log_bo = 0;
    pset->sci.phy_bo = 0;
    pset->sci.final_bo = 0;
    pset->sleft = NULL;
    pset->tright = NULL;
    *cond = NO_MATCH;
}
else if (*((cond = strcmp(s->snorm, pset->snorm)) == MATCH)
    error("set type already defined");
else if (*cond < 0)  /*move left to place set*/
    pset->sleft = st_insert(pset->sleft, s, cond);
else  /*move right to place set*/
    pset->tright = st_insert(pset->tright, s, cond);
return(pset);
typedef struct rec_type

extern struct rec_type *rt_root;
extern struct set_type *st_root;
struct set_type *st_search();
extern cnt_st;
struct set_type s;
struct set_type *st_insert();
struct rec_type *rt_search();
int cond,fd;
c realidad =NULL;
char *concat();
char newn[MAX_TNAME + APN + 1];
int verbose=NONE;
#endif DEBUG
printf("set_add \n");
#endif
if ((s.owner = rt_search(rt_root, owner)) == NULL쨔
mber = rt_search(rt_root, member)) == NULL
) return(error(" unable to build set type - owner/member not a record type"));
if (st_search(st_root, stname) != NULL)
return(error(" set type already defined");
if ((s.st_fd = create_file(stname, ST)) == ERROR
return(ERROR);
concat(newn, stname, ST);
if ((s.st_fd = creat(newn, PMODE)) == ERROR
return(error(" cannot create the set btree");
builtl(s.st_fd, SIZE, verbose);
close(s.st_fd);
if ((s.st_fd = open(newn, RW)) == ERROR
else return(error(" cannot open the set btree");
strcpy(s.name, stname);
s.sci.final_fo = OL;
s.sci.final_bo = 0;
st_root = st_insert(st_root, &s, scond);
if (cond == MATCH
return(ERROR);
else

cnt_st +=1;
return(OK);
/*St_search searches the set type b-tree for a specific occurrence by
set type name. If present, a pointer to the set type is returned.
Otherwise, null is returned*/

struct set_type *st_search(pset, name)

struct set_type *pset;
char *name;

{ int cond;
#endif

printf("e st_search\n");
#endif

if (pset == NULL) /*set not in tree*/
    return(NULL);
else if ((cond = strcmp(name, pset->name)) == MATCH)
    return(pset); /*match found*/
else if (cond < 0) /*look left*/
    return(st_search(pset->left, name));
else /*look right*/
    return(st_search(pset->right, name));
/*Write_st saves information on the set types by writing it to the DBMP.*/

set_t 205 0
set_t 206 0
set_t 207 0
set_t 208 0
set_t 209 0
set_t 210 0
set_t 211 0
set_t 212 0
set_t 213 1
set_t 214 1
set_t 215 1
set_t 216 1
set_t 217 1
set_t 218 1
set_t 219 1
set_t 220 1
set_t 221 1
set_t 222 1
set_t 223 1
set_t 224 1
set_t 225 1
set_t 226 2
set_t 227 2
set_t 228 2
set_t 229 2
set_t 230 2
set_t 231 2
set_t 232 2
set_t 233 2
set_t 234 2
set_t 235 2
set_t 236 2
set_t 237 2
set_t 238 2
set_t 239 2
set_t 240 2
set_t 241 1

st_write(fp, pset, cnt)
FILE *fp;
struct set_type *pset;
int *cnt;
{
    int i;
    struct rec_type *p;
    char *owner;
    char *member;
    #ifdef DEBUG
    printf(" e write_st\n");
    #endif
    if (pset == NULL)
        return;
    else
        if (pset->sci.add_on) /*flush buffer*/
            write_buf(pset->sci.fd, pset->sci.final_fd,
                      pset->sci.file_buf, pset->sci.final_bl);
    p = pset->owner;
    close(pset->sbfd);
    owner = p->name;
    p = pset->member;
    member = p->name; /*get name of owner record type*/
    fprintf(fp, %s %s %s %d %d\n", pset->sname, owner, member,
            pset->sci.final_fd, pset->sci.final_bl);
    *cnt +=1;
    st_write(fp, pset->sleft, cnt);
    st_write(fp, pset->sright, cnt);
    return;
This file contains the routines to initialize the database management system and to store information on record and set types when use of system concludes.

/* Init opens the database management file and sets up the record type b-tree and the set type b-tree. */

init()
{
    FILE *fp;
    extern int cnt_rt;
    extern long rbroot;
    extern int rbtfd;
    extern int cnt_st;
    long initbt();

    #ifdef DEBUG
    printf("e init\n");
    #endif

    if ((fp = fopen(DBMF, "r")) == NULL)
    {
        printf("record and set types have not been defined\n");
        return(OK);
    }

    fscanf(fp, "%d,%d,%d", &cnt_rt, &cnt_st);
    if (brecre_tree (fp, cnt_rt) == ERROR)
        return(ERROP);
    if (bset_tree (fp, cnt_st) == ERROR)
        return(ERROP);
    fclose (fp);

    if ((rbtfd = open(RBT_RW)) == ERROR) /* open and set record btree*/
        return(error("unable to open dbm.rec.bt"));
    rbroot = initbt(rbtfd);
    return(OK);
}
file

file  line  level  source
init_concl  46  0  /*Store_dbmf stores the information about record types and set types on the
init_concl  47  0  database management file.*/
init_concl  48  0
init_concl  49  0
init_concl  50  0  store_dbmf()
init_concl  51  0  {
init_concl  52  1  FILE *fp;
init_concl  53  1  int cnt;
init_concl  54  1  extern int cnt_rt;
init_concl  55  1  extern int cnt_st;
init_concl  56  1  extern struct rec_type *rt_root;
init_concl  57  1  extern struct set_type *st_root;
init_concl  58  1  extern int rbtfd;
init_concl  59  1
init_concl  60  1  #ifdef DEBUG
init_concl  61  1  printf("e store_dbmf");
init_concl  62  1  #endif
init_concl  63  1
init_concl  64  1  fp = fopen(DBMF,"w");
init_concl  65  1  fprintf(fp,"%d %d\n",cnt_rt,cnt_st);
init_concl  66  1  cnt = 0;
init_concl  67  1  rt_write(fp,rt_root,&cnt);
init_concl  68  1  if (cnt != cnt_rt)
init_concl  69  1  fprintf(stderr,"error during write of record tree on DBMF\n");
init_concl  70  1  cnt = 0;
init_concl  71  1  st_write(fp,st_root,&cnt);
init_concl  72  1  if (cnt != cnt_st)
init_concl  73  1  fprintf(stderr,"error during write of set tree on DBMF\n");
init_concl  74  1  fclose(fp);
init_concl  75  1  close(rbtfd);
init_concl  76  1  return;
init_concl  77  1 }
<table>
<thead>
<tr>
<th>line</th>
<th>file</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>utility</td>
<td>0</td>
<td>/<em>Utility contains functions that serve as utilities for the main program segments.</em>/</td>
</tr>
<tr>
<td>2</td>
<td>utility</td>
<td>0</td>
<td>/<em>Error prints and error message and returns an error flag.</em>/</td>
</tr>
<tr>
<td>3</td>
<td>utility</td>
<td>0</td>
<td>/<em>MAD 10/31</em>/</td>
</tr>
<tr>
<td>4</td>
<td>utility</td>
<td>0</td>
<td>error(msg)</td>
</tr>
<tr>
<td>5</td>
<td>utility</td>
<td>0</td>
<td>char msg[];</td>
</tr>
<tr>
<td>6</td>
<td>utility</td>
<td>0</td>
<td>/<em>error message</em>/</td>
</tr>
<tr>
<td>7</td>
<td>utility</td>
<td>0</td>
<td>{</td>
</tr>
<tr>
<td>8</td>
<td>utility</td>
<td>0</td>
<td>printf(stderr,&quot;%s\n&quot;,msg);</td>
</tr>
<tr>
<td>9</td>
<td>utility</td>
<td>0</td>
<td>return(ERROR);</td>
</tr>
<tr>
<td>10</td>
<td>utility</td>
<td>0</td>
<td>}</td>
</tr>
<tr>
<td>11</td>
<td>utility</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>utility</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>utility</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>utility</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>utility</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>utility</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>utility</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
/*error prints an error message and the accompanying key and returns an error flag.*/

terorg(msg,key)

char *msg;

char *key;

fprintf(stderr,"%s Key = %s\n",msg,key);

return(ERROR);
/*Rec_str replaces the end of record mark with a null bit and therefore, in
essence changes a record to a string.*/

utility 43 0
char *buf;

utility 44 1
int i;

utility 45 1
for (i=0; i< MAX_RECORD; i++)

utility 46 1
  if(*(buf+i) == END_OF_RECORD)
    *(buf+i) = NULL;

utility 47 1
return;

utility 48 2
}

utility 49 2
return(error("record too long"));

utility 50 2

utility 51 1

utility 52 1

utility 53 0

utility 54 0

utility 55 0
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>utility</td>
<td>57</td>
<td>0</td>
<td>/<em>Str_rec replaces the null bit at end of string with an end of record mark and in essence changes a string to a record.</em>/</td>
</tr>
<tr>
<td>utility</td>
<td>58</td>
<td>0</td>
<td>/<em>MAD 10-81</em>/</td>
</tr>
<tr>
<td>utility</td>
<td>60</td>
<td>0</td>
<td>str_rec(buf)</td>
</tr>
<tr>
<td>utility</td>
<td>62</td>
<td>0</td>
<td>char *buf;</td>
</tr>
<tr>
<td>utility</td>
<td>64</td>
<td>0</td>
<td>{</td>
</tr>
<tr>
<td>utility</td>
<td>65</td>
<td>1</td>
<td>int i;</td>
</tr>
<tr>
<td>utility</td>
<td>66</td>
<td>1</td>
<td>for (i=0; i &lt; MAX_RECORD; i++)</td>
</tr>
<tr>
<td>utility</td>
<td>67</td>
<td>1</td>
<td>if (*buf+i) == NULL</td>
</tr>
<tr>
<td>utility</td>
<td>68</td>
<td>1</td>
<td>{</td>
</tr>
<tr>
<td>utility</td>
<td>69</td>
<td>2</td>
<td>*(buf+i) = END_OF_RECORD;</td>
</tr>
<tr>
<td>utility</td>
<td>70</td>
<td>2</td>
<td>return;</td>
</tr>
<tr>
<td>utility</td>
<td>71</td>
<td>2</td>
<td>}</td>
</tr>
<tr>
<td>utility</td>
<td>72</td>
<td>1</td>
<td>return(error(&quot;string too long&quot;));</td>
</tr>
<tr>
<td>utility</td>
<td>73</td>
<td>1</td>
<td>}</td>
</tr>
<tr>
<td>utility</td>
<td>74</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>75</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>76</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>77</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>78</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>line</td>
<td>level</td>
<td>source</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>0</td>
<td>/<em>Arccpy copies one array of integers to another array.</em>/</td>
</tr>
<tr>
<td>utility</td>
<td>81</td>
<td>0</td>
<td>/<em>MAD 10-81</em>/</td>
</tr>
<tr>
<td>utility</td>
<td>82</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>83</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>84</td>
<td>0</td>
<td>arccpy (top, fromp, size)</td>
</tr>
<tr>
<td>utility</td>
<td>85</td>
<td>0</td>
<td>int *top, *fromp; /<em>array copied to and from</em>/</td>
</tr>
<tr>
<td>utility</td>
<td>86</td>
<td>0</td>
<td>int size; /<em>size of array to be copied</em>/</td>
</tr>
<tr>
<td>utility</td>
<td>87</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>88</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>89</td>
<td>1</td>
<td>int i; for (i = 0; i &lt; size; i++) *top++ = *fromp++;</td>
</tr>
<tr>
<td>utility</td>
<td>90</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>91</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>92</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>93</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>94</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>95</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>96</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Create_file will create and then open a file. The file opened will have a name consisting of a name with a string appended to the end of it.*/

create_file (filen, append)

char *filen,*append;

int fd;
char catfname[MAX_TNAME + APN + 1];
strcpy (catfname,filen);
strcat(catfname,append);

if ((fd = creat(catfname,PMODE)) == ERROR)
    return(error("Unable to create file ");
else if ((fd = open(catfname,RW)) == ERROR)
    return(error("Unable to open file ");
else
    return(fd);
utility Mon Sep 27 14:44:19 1982

file

utility 123 0 /* Openfile opens a character file for read/write options. The filename is
utility 124 0 a combination of a name and a string appended to the end of the name.
utility 125 0 If no error occurs, a file descriptor is returned. */
utility 126 0
utility 127 0
utility 128 0
utility 129 0
utility 130 0 openfile (filen, append)
utility 131 0 char *filen, *append;
utility 132 1 {
    int fd;
    char catfname[MAX_TNAME + APN + 1];
    strcpy (catfname,filen);
    strcat (catfname,append);
    utility 135 1 if ((fd = open(catfname,RW)) == ERROR)
utility 136 1 return(error("unable to open file\n"));
utility 137 1 else return(fd);
utility 138 1
utility 139 1
utility 140 1
utility 141 0
utility 142 0
utility 143 0
utility 144 0
utility 145 0
utility Mon Sep 27 14:44:19 1982 Page 8

file line level source
utility 147 0 /* Concatn concatenates two strings. */
utility 148 0
utility 149 0 char *concatn(newstring, bstring, estring)
utility 150 0 char *newstring, *bstring, *estring;
utility 151 0 {
   strcpy(newstring, bstring);
utility 152 1 strcat(newstring, estring);
utility 153 1 return(newstring);
utility 154 1
utility 155 1 }
```c
/*Read_rec reads a record from the file buffer into the record input/output
 buffer.*/

/*MAD 10/81*/

read_rec(f_offset, b_offset, rec_io, pci)

long f_offset;                     /*file offset of record to be read*/
int b_offset;                       /*block offset*/
char *rec_io;                       /*pointer to record input/output buffer*/
struct cl *pci;                     /*structure containing information for rec/set type*/

phy_io, phy_bo points at start of current of record type.

phy_io, phy_bo points at end of current of record type.

final_io, final_bo points at next place on file to add a

record. */

{ int size;

  #ifdef DEBUG
  printf("entering read_rec\n");
  #endif

  if (f_offset < 0 || b_offset < 0 || pci->fd < 0)
    return(error("Illegal input to read_rec"));
  if (pci->phy_io == 0L && pci->log_io == 0L && pci->phy_bo == 0L &&
      pci->add_on == FALSE)
    /*prime buffer the first time through*/
    { pci->phy_io = pci->final_io;
      pci->phy_bo = pci->final_bo;
      read_buf(pci->fd, pci->phy_io, pci->phy_bo, pci->file_buf, pci->phy_bo);
    }
  if (f_offset != pci->phy_io && pci->add_on)
    /*write to file previously added record*/
    { if (pci->final_bo == write_buf(pci->fd, pci->final_io, pci->file_buf, pci->final_io,
                              pci->add_on == FALSE);
      else
        return(error("write incomplete"));
    }
  pci->log_io = f_offset;
  if (f_offset != pci->phy_io)
    { /*reset currency pointers*/
      pci->phy_io = f_offset;
      if (pci->phy_io != pci->final_io)
        size = BLOCKSIZE;
      else
        size = pci->final_bo;
      if (read_buf(pci->fd, pci->phy_io, pci->file_buf, size) == ERROR)
        return(error("unable to read input file properly"));
    }
  printf("read_rec\n");
}``
if (pci->add_on && b_offset >= pci->final_bo) (f_offset == pci->final_fo && b_offs
   return(error("unable to read input-record never written"));
pci->log_bo = pci->phy_bo = b_offset;
return(copy_in(rec_io,rec_io+MAX_RECORD,pci));


file

line

level

source

10

63 0 /*Copy_in copies a record, character by character, from the file buffer into
10
64 0 the record input/output buffer.*/
10
65 0 */^AD 10/81*/
10
66 0 copy_in(rec_io,max_io,pci)
10
67 0 char *rec_io; /* " to record input/output buffer*/
10
68 0 char *max_io; /*. " to 1st char beyond i/o buffer*/
10
69 0 struct ci *pci; /*currency pointers for rec/set type*/
10
70 0 {
10
71 0 int size; /*move record to i/o buffer*/
10
72 0 /*number of characters to read*/
10
73 0 #ifdef DEBUG
10
74 0 printf ("e copy_in\n");
10
75 0 #endif
10
76 0 while (pci->phy_bo < BLOCKSIZE &&
10
77 0 (*rec_io++ = *(pci->file_buf+(pci->phy_bo++))) != END_OF_RECORD &&
10
78 0 rec_io < max_io)
10
79 0 ;
10
80 0 if (rec_io >= max_io)
10
81 0 return(error("record to large for input buffer"));
10
82 0 else
10
83 0 else if (pci->phy_bo == BLOCKSIZE)
10
84 0 { /*need to read next block from file*/
10
85 0 pci->phy_bo += BLOCKSIZE;
10
86 0 pc1->phy_bo = 0;
10
87 0 if(pci->phy_bo != pci->final_bo)
10
88 0 size = BLOCKSIZE;
10
89 0 else
10
90 0 size = pci->final_bo;
10
91 0 if (read_buf(pci->fd,pci->phy_bo,pci->file_buf,size) != ERROR)
10
92 0 return(copy_in(rec_io,max_io,pci));
10
93 0 else
10
94 0 return(error("read into buffer incomplete"));
10
95 0 }
10
96 0 else
10
97 0 return(OK);
file line level source

10 109 0 /*Read_buf reads one block from a file into a buffer*/
10 110 0
10 111 0 /*MAD 10/81*/
10 112 0
10 113 0 read_buf (fd,offset,file_buf,size)
10 114 0 int fd; /*file descriptor for input file*/
10 115 0 long offset; /*offset of desired block of file*/
10 116 0 char *file_buf; /*file input buffer*/
10 117 0 int size; /*number of characters to read*/
10 118 0
10 119 0 {
10 120 1 #ifdef DEBUG
10 121 1 char *ij;
10 122 1 printf( "read=\n");
10 123 1 for (i=file_buf;i<file_buf+size;i++)
10 124 1 printf("%c",*i);
10 125 1 printf("\n");
10 126 1 #endif
10 127 1
10 128 1 lseek(fd,offset,START); /*move to offset in file*/
10 129 1 return(read(fd,file_buf,size));
10 130 1 }
10 131 0
10 132 0
10 133 0
10 134 0
10 135 0
/*Write_rec writes one record from the input/output buffer to the file buffer*/

/*MAD 10/81*/

write_rec(f_offset,b_offset,rec_io,pci)
long *f_offset; /*pointer to file offset of record to be written*/
int *b_offset; /*pointer to block offset*/
char *rec_io; /*pointer to record input/output buffer*/
struct ci *pci; /*currency information for rec/set type*/

/*log_from, log_to points at start of current of record type.
phy_from, phy_to points at end of current of record type.
final_from, final_to points at next place on file to add
a record.*/

{ int cond;
  #ifdef DEBUG
    printf("enter line write_rec\n");
  #endif

    if (*f_offset == NEWL)
    { /*add new record to file*/
      pc1->log_from = pc1->phy_from = *f_offset = pc1->final_from;
      pc1->log_to = pc1->phy_to = *b_offset = pc1->final_to;
      if (!pc1->add_on)
      { /*final record not presently in core*/
        if (pc1->phy_from != pc1->final_from)
          cond = read_buf(pc1->fd,pc1->phy_from,pc1->file_buf,BLOCKSIZE);
        else
          cond = read_buf(pc1->fd,pc1->phy_from,pc1->file_buf,pc1->final_to);
        if (cond != ERROR)
          if (cond == ERROR & pc1->final_from == START & pc1->final_to == START)
            pc1->add_on = TRUE;
          else
            return(error("can not read output file");
        }
      }
    }
    copy_out(rec_io,rec_io+MAX_RECORD,pci);
    if (pc1->add_on)
    { 
      pc1->final_from = pc1->phy_from;
      pc1->final_to = pc1->phy_to;
    }
  else
    {
      /*write over existing record*/
      if (pc1->add_on & & pc1->final_from != *f_offset) /*write out new record left
      if (write_buf(pc1->fd,pc1->final_from,pc1->file_buf,pc1->final_to) == pc1->final_to
        return(error("file write improperly done");
    }
```c
else
    pci->add_on = FALSE;
if (pci->phy_fo != *f_offset)
{
    if (*f_offset != pci->final_fo)
        cond = read_buf(pci->fd, f_offset, pci->file_buf, BLOCKSIZE);
    else
        cond = read_buf(pci->fd, f_offset, pci->buffer, pci->final_bo);
    if (cond == ERROR)
        return(error("file read error"));
}
pci->log_fo = pci->phy_fo = *f_offset;
pci->log_bo = pci->phy_bo = *b_offset;
copy_out(rec, rec_len+MAX_RECORD, pci);
if (pci->phy_fo != pci->final_fo) /*write updated record*/
    cond = write_buf(pci->fd, pci->phy_fo, pci->file_buf, BLOCKSIZE);
else
    cond = write_buf(pci->fd, pci->phy_fo, pci->buffer, pci->final_bo);
if (cond == BLOCKSIZE || cond == pci->final_bo)
    return (OK);
else
    return(error("write not completed"));
```
/*Copy_out copies a record from an input/output area and places it in the file
  buffer. If need be, the file buffer is written out and a new one is made
  available*/
/*MAD 10/81*/
copy_out (rec_io, max_io, pci)
  char *rec_io; /* record input/output buffer */
  char *max_io; /* 1st character past record I/O buffer*/
  struct ci *pci; /*currency information for record/set type*/
  { int cond;
    char *i;
  }
 ifdef DEBUG
  printf("entering copy_out\n");
 endif
 for (i = rec_io; i < max_io & *i != END_OF_RECORD; ++i)
  { /*error check record size*/
    if (i == max_io)
      return(error("output record to large"));
    while (pci->phy_bo < BLOCKSIZE &
        (*(pci->file_buf + (pci->phy_bo)++) = *rec_io++) != END_OF_RECORD)
        ;
    if (*((rec_io-1) = END_OF_RECORD & pci->phy_bo != BLOCKSIZE)
        { if (1 pci->add_on) /*write revised record*/
          { if (pci->phy_bo != pci->final_bo)
              if (BLOCKSIZE == write_buf(pci->fd,pci->phy_bo,pci->file_buf,BLOCKSIZE)
                  return(OK);
              else if (pci->final_bo == write_buf(pci->fd,pci->phy_bo,pci->file_buf,pci
                  return(OK);
            else
              return(error("write not completed"));
          }
        }
        else if (BLOCKSIZE == write_buf(pci->fd,pci->phy_bo,pci->file_buf,BLOCKSIZE))
          /*write buffer area to output*/
          { pci->phy_bo += BLOCKSIZE;
            pci->phy_bo = 0;
          if (pci->add_on)
            { pci->final_bo = pci->phy_bo;
            pci->final_bo = 0;
          } else
            { if(pci->phy_bo != pci->final_bo)
cond = read_buf(pci->fd,pci->phy_file,pci->file_buf,BLOCK_SIZE);
else
cond = read_buf(pci->fd,pci->phy_file,pci->file_buf,pci->final_size);
if (cond == ERROR)
    return(error("read incomplete"));
}
return(copy_out(rec_io,max_io,pci));
else
return(error("incomplete write to output file"));
/*Write_buf writes one block onto a file from a buffer*/

int fd;

/*offset of desired block on file*/

int size;

#define DEBUG
char *i;

for(i=file_buf; i<file_buf+size; i++)
    printf("%c", *i);

printf("\n");

#define ENDIF

lseek(fd, offset, START); /*move to correct position in file*/
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbup_c_dec</td>
<td>1</td>
<td>0</td>
<td>/* These variables are used in the dbup routines. */</td>
</tr>
<tr>
<td>dbup_c_dec</td>
<td>2</td>
<td>0</td>
<td>#include &quot;ext_dec&quot;</td>
</tr>
<tr>
<td>dbup_c_dec</td>
<td>3</td>
<td>0</td>
<td>char iobuf[MAX_RECORD + 2];</td>
</tr>
<tr>
<td>dbup_c_dec</td>
<td>4</td>
<td>0</td>
<td>char key[MAX_KEY + 1];</td>
</tr>
<tr>
<td>dbup_c_dec</td>
<td>5</td>
<td>0</td>
<td>char fp[MAX_KEY + 1];</td>
</tr>
<tr>
<td>dbup_c_dec</td>
<td>6</td>
<td>0</td>
<td>char bp[MAX_KEY + 1];</td>
</tr>
<tr>
<td>dbup_c_dec</td>
<td>7</td>
<td>0</td>
<td>char savekey[MAX_KEY + 1];</td>
</tr>
<tr>
<td>dbup_c_dec</td>
<td>8</td>
<td>0</td>
<td>char savenext[MAX_KEY + 1];</td>
</tr>
<tr>
<td>dbup_c_dec</td>
<td>9</td>
<td>0</td>
<td>char unit[MAX_KEY + 1];</td>
</tr>
<tr>
<td>dbup_c_dec</td>
<td>10</td>
<td>0</td>
<td>struct ptrs os, saveos, saveos2;</td>
</tr>
<tr>
<td>dbup_c_dec</td>
<td>11</td>
<td>0</td>
<td>struct rec_type *rt;</td>
</tr>
<tr>
<td>dbup_c_dec</td>
<td>12</td>
<td>0</td>
<td>struct set_type *st;</td>
</tr>
</tbody>
</table>
/* This file contains the following routines to update the database. */
ra - add a record occurrence to the database - recadd
ao - add an owner set occurrence - ownadd
am - add a member set occurrence - mebadd
dr - delete a record occurrence from the database - recdel
do - delete an owner set occurrence - owndel
dm - delete a member set occurrence - mebdel
fr - find a record - recfind
fo - find an owner record of a specific member in a set - ownfind
ff - find the first member of an owner in a set - mebfind
fn - find next owner or member in a set - nextfind
cn - change owner of specific member to new owner in same set - cnrown
cn - change all members of an owner to new owner in same set - cnall

Service routines called by the update routines
setoad
owntdel
mebdel
rsoc
savecor
putpts
getpts
valkr
valks
bldkey
nfields

NOTE: dbup_c_dec includes declarations that are common to the update routines
and includes declarations of all external variables. */
file  line  level  source

dbup  41   0  /*Record add will add records to a record file type as designated by the cor

dbup  42   0  record type. If a file is specified, all the records in the file

dbup  43   0  are added. Otherwise, records are added from the standard input until

dbup  44   0  EOF is read. The cor unit is set to the key of the last record added.*/

dbup  45   0

dbup  46   0

dbup  47   0  recadd (ifp)

dbup  48   0  FILE *ifp;

dbup  49   0  {

dbup  50   1  #include "dbup_c_dec"

dbup  51   1  char c;

dbup  52   1  int more;

dbup  53   1  char *i;

dbup  54   1  struct set_type *pst;

dbup  55   1  int n;

dbup  56   1

dbup  57   1  ifndef DEBUG

dbup  58   1  printf ("entering record add\n");

dbup  59   1  #endif

dbup  60   1

dbup  61   1  more = TRUE;

dbup  62   1  while (more)

dbup  63   1   {

dbup  64   2   if (((c = getc(ifp)) == EOF)

dbup  65   2     more = FALSE;

dbup  66   2   else

dbup  67   2     {

dbup  68   3     ungetc(c,ifp);

dbup  69   3     i = 1obuf;

dbup  70   3     while (((c = getc(ifp)) != END_OF_RECORD &&

dbup  71   3     i < 1obuf + MAX_RECORD)

dbup  72   3     *i++ = c;

dbup  73   3     if (i == 1obuf + MAX_RECORD)

dbup  74   3     return(errorex("input record too long");

dbup  75   3     *i++ = END_OF_RECORD;

dbup  76   3     *i = NULL;

dbup  77   3     /*validate input record*/

dbup  78   3     if ((n=nfields(1obuf,cor_rt->fdir)) != cor_rt->nsofield)

dbup  79   3     return(errorex("illegal # of fields for record type");

dbup  80   3     else if (bldkey(1obuf,key) == ERROR)

dbup  81   3     return(ERROR);

dbup  82   3     else if (item_search(rblk,rbroot,key,(&os)) == PRESENT)

dbup  83   3     fprintf(stderr,"%s duplicate key\n", key);

dbup  84   3     else

dbup  85   3     {

dbup  86   4     /*write rec to file and save offsets*/

dbup  87   4     #ifdef DEBUG

dbup  88   4     printf("key= %sn", key);

dbup  89   4     #endif

dbup  90   4     os.btd = NEWL;
file line level source

```
091 4 os.btb = NEW;
092 4 if (write_rec(&os.btd,&os.btb,obuf,&cor_rt->rc1)
093 4 == ERROR)
094 4     return(FERROR);
095 4 else if (item_insert(rbtfd,&rbtroot,key,&os) == NULL)
096 4     return(ERROR);
097 4 else
098 4 {
099 5     strcpy(cor_unit, key);
100 5     pst = st_root;
101 5     setoadd (pst); /*add key as owner in sets*/
102 5 }
103 4 }
104 3 }
105 2 }
106 1 return(OK);
107 1 }
108 0 }
109 0 }
110 0 }
111 0 }
112 0 }
```
/*Ownadd adds an owner set occurrence to a set type*/

int ownadd()
{
    ifdef DEBUG
        printf("e ownadd\n");
    endif

cor_rt = cor_st->owner;
    if (valkr(Sos, lobuf) != ERROR) /*validate key and record format*/
    {
        os.btd = NEWL; /*build new owner record occur*/
        os.btb = NEW;
        *fp = NULL;
        *bp = NULL;
        /*store the new owner record and offset*/
        if (putps(fp, bp, &os) == ERROR)
            item_insert(cor_st->sbtfd, &cor_st->sbtroot, cor_unit, &os)
            i = PRESENT
            return(OK);
        else
            return(OK);
    }
    else
        return(OK); /*invalid key or record format*/
/*Mebad add a member set occurrence to a set type*/

membad(mkey) char *mkey;
{
    #include "dbup_c_dec"

    ifdef DEBUG
    printf(" me bad\n");
    #endif

cor = cor - > member;
strcpy (savekey, cor - > unit); /*save owner key*/
strcpy (cor - > unit, mkey); /*set currency to member key*/
if (valkr(Sos, lobuf) == ERROR)
{
    strcpy(cor - > unit, savekey); /*invalid key or record format*/
    return(OK);
}
else if (item search(cor - > sbtfd, cor - > sbtroot, savekey, &s)
    != PRESENT) /*make sure owner is in set type*/
{
    strcpy(cor - > unit, savekey);
    return(kerror("owner not a member of settype", cor - > unit));
}
else if (getpts(fp, bp, &os) == ERROR) /*get owner pointers*/
    return(OCR);
else if (*bp != NULL)
    return(error("owner key is member in the set");
else
{
    /*set owner forward pointer to new member*/
strcpy(savennex, fp);
strcpy(fp, cor - > unit);
putpts(fp, bp, &os);
if (*savennex == NULL) /*build a new member record*/
    strcpy(fp, savekey);
else
    strcpy(fp, savennex);
strcpy (bp, savekey);

os btw = NEWL;
os btt = NEW;
putpts(fp, bp, &os); /*store new member occur rec*/
item_insert(cor - > sbtfd, cor - > sbtroot, cor - > unit, &os);
if (*savennex != NULL) /*fix backward pointer of next member*/
    {
        if (item_search(cor - > sbtfd, cor - > sbtroot, savennex, 
            Sos) != PRESENT ||
        getpts(fp, bp, &os) == ERROR ||
        strcmp(bp, savekey) != MATCH)
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbup</td>
<td>200</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>201</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>202</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>203</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>204</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>205</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>206</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>207</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>208</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>209</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>210</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>211</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
/*Recdel deletes a record from the database record file. Any set occurrences for this key are also deleted.*/

recdel()
{
#include "dbup_c_dec"

struct set_type *pst;

#ifndef DEBUG

printf("entering recdel\n");

#endif

if (item_delete(rbtfd, &rbtroot, cor_unit, &os) != PRESENT)
    return(NOTPRESENT);

pst = st_root; /*now delete all set occur*/
owndel(pst); /*delete owner occur*/

pst = st_root; /*delete member occur*/
mebtdel(pst);

return(PRESENT);
}
/*Owndef deletes an owner occurrence in a set. This causes the record in the
record file to be deleted and all members to be deleted. Also, any set that
contains this owner as a member is also visited and the owner-now-member
is deleted.*/

owndef()
{
    #include "dbup.c_def"
    int more;
    #ifndef DEBUG
    printf("e owndef\n");
    #endif
    cor_rt = cor_st->owner;
    if (item_delete(cor_st->sbefd, &cor_st->sbroot, cor_unit, &os) != PRESENT)
        return(NOTPRESENT);
    else
    {
        #ifdef DEBUG
        printf("after delete fo= %d, bo= %d\", os.btd, os.btb);
        #endif
        savecor(unit, &st, &rt); /*save currency pointers*/
        #ifdef DEBUG
        printf("after savecor- unit=%s, st= %d, rt= %d\", unit, st, rt);
        #endif
        printf("should be -coru= %s, st= %d, rt= %d\", cor_unit, cor_st, cor_rt);
        #endif
        more = TRUE;
        while (more)
        {
            getpts(fp, bp, &os); /*get key for next member*/
            #ifdef DEBUG
            printf("getpts- fp= %s, bp= %s\", fp, bp);
            #endif
            if (*fp == NULL)
                more = FALSE;
            else
            {
                strcpy(cor_unit, fp); /*set currency for next member*/
                memdei(os.btd, os.btb);
            }
        }
        rscor(unit, &st, &rt); /*reset currency*/
```c
#define DEBUG

define "%s, corst= %d, corrt= %d"

printf("after rscor- coru= %s, corst= %d, corrt= %d
", cor_unit, cor_st, cor_rt);

#endif

define (); /*delete the record from the record ttree*/

rscor(unit,&st,&rt);
return(OK);
```
```c
/* mebdel deletes a member from a set occurrence. If the prior member or owner
 is not in the tree for the set, then the file offset and block offset passed
 as parameters are checked. If the fos is equal to newl, the routine is
 in the middle of a recursive call and this member will be deleted later. If
 fos is not equal to newl, it then contains the file offset of the prior member
 and is used to get the prior member's pointers. */

mebdel(fos, bos)

#include "dbup_c_dec"

#define DEBUG

printf("mebdel\n")

#define

cor_rt = cor_st->member;
savcor(unit, &st, &rt);

if(item_delete(cor_st->sbtfd, &cor_st->sbtroot, cor_unit, &os) != PRESENT)
  return(NOTPRESENT);

saveos = os;

getpts(fp, bp, &os);

if (*bp == NULL || *fp == NULL)
  return(kerror("delete processing error", cor_unit));

/* get pointers for this member */

if(item_search(cor_st->sbtfd, cor_st->sbtroot, bp, &os) != PRESENT)
  if (fos == NEW)
    return(NOTPRESENT); /* will get this member later */
  else
    { os.btd = fos; os.btb = bos; }

#define DEBUG

printf("back from i_search fo= %ld, bo= %d\n", os.btd, os.btb);
#endif

strcpy(savekey, bp);
strcpy(savenext, fp);

#define DEBUG

printf("savekey= %s, savenext= %s\n", savekey, savenext);
#endif

getpts(fp, bp, &os); /* get prior key's record */

if (strcmp(savekey, savenext) == "MATCH")
```
```c
{ /*last member for this owner*/
    if (*bp != NULL)
        return(kerror("delete processing error", bp));
    else
    {
        *fp = NULL;
        putpts(fp, bp, &os); /*reset prior's pointers*/
    }
} else
{
    strcpy(fp, savnext); /*reset prior's pointers*/
    putpts(fp, bp, &os);  
    if (item_search(cor_st->sbf, cor_st->sbyroot, savnext, &os)  
      != PRESENT)  
        return(kerror("missing set links", savnext));
    retpts(fp, bp, &os); /*reset next member's pointers*/
    if (*bp != NULL)  
        strcpy(bp, savekey);
    putpts(fp, bp, &os);
}
rscon (unit, &st, &rt); /*now delete the record if present*/
recdel();
return(OK);  
```
/* Recfind outputs a record using the currency pointers to decide which key */

#include "dbup_c_dec"

char c, *cptr;

/* get the record from db storage */

ifdef DEBUG

printf("entering recfind\n");

endif

if (item_search(rbtfd, rbroot, cor_unit, &os) != PRESENT)
  return(error(rbtfd, rbroot, cor_unit, &os));

/* get record into io buffer */

if (read_rec(os, bts, os.btb, iobuf, &cor_rt->rci) == ERROR)

  return (ERROR);

c = 'p';

cptr = iobuf;

while (c != END_OF_RECORD)

  { c = *cptr++; 

   putc (c, ofp);

  }

return(OK);
/*ownfind finds the owner for a member key and outputs the record for that owner.*/

ownfind (ofp)

FILE *ofp;

#include "dbup_c_dec"

#ifdef DEBUG

/*printf("e ownfind\n");

#endif

cor_rt = cor_st->owner;  /*find set owner occur*/

if (item_search(cor_st->sbfld, cor_st->sbfroot, cor_unit, sos) != PRESENT)
    return(kerror("key absent from set type", cor_unit));

getpts(fp, bp, sos);

if (*bp == NULL)
    return(kerror("key is an owner", cor_unit));

while (*bp != NULL)
    { /*follow links to owner*/

        if (item_search(cor_st->sbfld, cor_st->sbfroot, fp, sos) != PRESENT)
            return(kerror("processing error-missing set link", cor_unit));

        strcpy(cor_unit, fp);

        getpts(fp, bp, sos);

    }

recfind(ofp);  /*output owner record*/

return(OK);  

}
/*Mebfind finds the 1st member record for an owner*/

mebfnd (ofp)
FILE *ofp;
{
#include "dbup_cdefs"

#define DEBUG

printf("e mebfnd \n");
#endif

cor_rt = cor_st->member;

if (item_search(cor_st->stbsd, cor_st->subroot, cor_unit,&os) != PRESENT)
    Return(kerror("key absent from set type",cor_unit));

getpts(fp,bp,&os);

if (*bp != NULL)
    return(kerror("key not an owner",cor_unit));

if (*fp == NULL)
    return(kerror("key has no members",cor_unit));

strcpy (cor_unit,fp); /*reset currency to member's record*/
strcpy (cor_st->cos,cor_unit);
recfnd (ofp); /*output record*/
return(OK);
file    line    level    source

dbup     488     0    /*Nextfind outputs the record of the next key in the set link*/
dbup     489     0    
dbup     490     0    nextfind (ofp)
dbup     491     0    FILE *ofp;
dbup     492     0    {    

dbup     493     1    #include "dbut_c_dec"
dbup     494     1    

dbup     495     1    ifndef DEBUG

dbup     496     1    printf("e nextfind\n");

dbup     497     1    endif

dbup     498     1    

dbup     499     1    

dbup     500     1    

dbup     501     1    if (!(*cor_st->cos) || cor_st->cos)

dbup     502     1    strcpy(cor_unit,cor_st->cos);

dbup     503     1    if(item_search(cor_st->sbfid,cor_st->sbfroot,cor_unit,&os) != PRESENT)

dbup     504     1    return(kerror("key absent from set type",cor_unit));

dbup     505     1    getpts(fp,bp,&os);    /*find next key in set*/

dbup     506     1    if (!(*bp)  

dbup     507     1    

dbup     508     2    

dbup     509     2    

dbup     510     2    

dbup     511     2    

dbup     512     2    

dbup     513     2    

dbup     514     2    

dbup     515     2    

dbup     516     3    

dbup     517     3    

dbup     518     3    

dbup     519     3    

dbup     520     3    

dbup     521     2    

dbup     522     1    

dbup     523     1    

dbup     524     1    

dbup     525     0    

dbup     526     0    

dbup     527     0    

dbup     528     0    

dbup     529     0
file  line level source

dbup  531  0  /*Cmsgown changes a member in a set occurrence to a new owner in the same set*/
dbup  532  0

dbup  534  0  cmsgown(newokey)
dbup  535  0  char *newokey;
dbup  536  0

dbup  537  1  #include "dbup_c_dec"
dbup  538  1

dbup  539  1  ifndef DEBUG

dbup  540  1  printf("e cmsgown\n");
dbup  541  1  #endif

dbup  542  1

dbup  543  1

dbup  544  1

dbup  545  1  /*validate new owner key and member key*/
dbup  546  1  if (valks(newokey,os,&saveos) == ERROR)
dbup  547  1    return(ERROR);
dbup  548  1  getpts(fp,bp,&os);  /*set member's links*/
dbup  549  1  if (*bp == NULL)
dbup  550  1    return(kerror("key is owner in this set",cor_unit));
dbup  551  1  if (item_search(cor_st->sbfd,cor_st->sbroot,bp,&saveos2) != PRESENT)
dbup  552  1    return(kerror("missing set links",fp));
dbup  553  1  if (strcmp(fp,bp) == MATCH)
dbup  554  1  {  /*final member in this occur.*/
dbup  555  2    getpts(fp,bp,&saveos2);

dbup  556  2    *fp = NULL;

dbup  557  2    putpts(fp,bp,&saveos2);

dbup  558  2

dbup  559  1  else

dbup  560  1  {

dbup  561  2    strcpy(savenext,fp);  /*save member's links*/

dbup  562  2    strcpy(savekey,bp);

dbup  563  2    getpts(fp,bp,&saveos2);  /*reset prior member's links*/

dbup  564  2    strcpy(fp,savenext);

dbup  565  2    putpts(fp,bp,&saveos2);

dbup  566  2    if(item_search(cor_st->sbfd,cor_st->sbroot,savenext,&saveos2) != PRESENT)

dbup  567  2    return(kerror("missing set link",savenext));

dbup  568  2    getpts(fp,bp,&saveos2);  /*reset following mem's links*/

dbup  569  2    if (*bp != NULL)

dbup  570  2      strcpy(bp,savekey);

dbup  571  2    putpts(fp,bp,&saveos2);

dbup  572  2

dbup  573  1  /*now add member to new owner*/

dbup  574  1    getpts(fp,bp,&saveos);  /*reset new owner's links*/

dbup  575  1    strcpy(savenext,fp);

dbup  576  1    strcpy(fp,cor_unit);

dbup  577  1    putpts(fp,bp,&saveos);

dbup  578  1

dbup  579  1  if (*savenext != NULL)

dbup  580  1  {

if(item_search(cor_st->sbf, cor_st->sbfroot, save->next, &saveos) != PRESENT) {
    return(kerror("missing links", save->next));
    getpts(fp, bp, &saveos);
    strcpy(savekey, bp);
    strcpy(bp, cor_unit);
    putpts(fp, bp, &saveos);
} else {
    strcpy(save->next, newkey); /* new owner has no members*/
    strcpy(fp, save->next); /* set member's new links*/
    strcpy(bp, newkey);
    putpts(fp, bp, &os);
    return(OK);
}
file  line  level  source

dbup  601  0  /*Cngall changes all the members in a set occurrence to a new owner within
    the same set.*/
dbup  602  0

dbup  603  0
dbup  604  0
dbup  605  0  cngall(newokey)
dbup  606  0  char *newokey;
dbup  607  0  {
    dbup  608  1  #include "dbup_c_def"
    dbup  609  1
    dbup  610  1  #ifdef DEBUG
    dbup  611  1  printf("e cngall\n");
    dbup  612  1  #endif
    dbup  613  1
    dbup  614  1
    dbup  615  1
    dbup  616  1  if (walks(newokey,&os,&saveos) == ERROR)
    dbup  617  1    return(ERROR);
    dbup  618  1    getpts(fp,bp,&os);  /*get member key*/
    dbup  619  1    while (*fp != NULL)
    dbup  620  1    {
        dbup  621  2    strcpy(cor_unit,fp);  /*move each member to newowner*/
        dbup  622  2        if (cngown(newokey) == ERROR)
        dbup  623  2            return(ERROR);
        dbup  624  2    getpts(fp,bp,&os);
        dbup  625  2    }
    dbup  626  1    strcpy(cor_unit,newokey);
    dbup  627  1
    dbup  628  0
    dbup  629  0
    dbup  630  0
    dbup  631  0
    dbup  632  0
/* Service routines */

/* Setoadd searches the set type binary tree and if a match of owner and
cor record type is found, adds the new record key as an owner to the set */

setoadd(pst)
struct set_type *pst;
{
    #include "ext_dec"
    #ifdef DEBUG
        printf("e setoadd\n");
    #endif
    if(pst == NULL)
        return;
    else
        {
            if (pst->owner == cor_rt)  
            {  
                cor_st = pst;  
                ownadd();
            }
            setoadd(pst->sleft);
            setoadd(pst->sright);
        }
}
/* Owntdel searches the set type binary tree and if a match of owner and
cor record type is found, deletes any owner record with cor unit as key. */

struct set_type *pst;

#include "ext_dec"

#include "debug"

owntdel(pst)

if (pst == NULL) return;

else {
    if (pst->owner == cor_rt) {
        cor_st = pst;
        owndel();
        owntdel(pst->sleft);
        owntdel(pst->sright);
    }
    }
/*Mebt del searches the set type tree and if a match of member and cor_rt
is found, deletes the owner record of cor_unit.*/

mebt del(pst)
struct set_type *pst;
{
    #include "ext_dec"
    #ifdef DEBUG
     printf("e mbd el
");
    #endif
    if (pst == NULL)
        return;
    else
    {
        if (pst->member == cor rt)
        {
            cor St = pst;
            mbd el(NEWL,NEW);
        }
        mbd el(pst->sleft);
        mbd el(pst->sright);
        return;
    
    
}
/* Rscor resets the currency pointers */

rscor(unit, st, rt)
char *unit;
struct rec_type **rt;
struct set_type **st;
{
    #include "ext_dec"
        
    #ifdef DEBUG
    printf("e rscor\n");
    
    strcpy(cor_unit, unit);
    cor_st = *st;
    cor_rt = *rt;
    return;

    }
/*Savecor saves the currency pointers*/
savecor(unit, st, rt)
char *unit;
struct set_type **st;
struct rec_type **rt;
{
    #include "ext_dec"
    ifndef DEBUG
    printf("e savecor\n");
    endif
    strcpy(unit, cor_unit);
    *st = cor_st;
    *rt = cor_rt;
    return;
}
/*Putpts builds a set type record containing the links for an owner or member in
a set occurrence. Once the record is built, it is output to the set file.*/

putpts(fp,bp,os)
char *fp,*bp;
struct ptrs *os;
{
  #include "ext_dec"

  char *obuf[MAX_RECORD+1];
  char *pt;
  char c;

  ifdef DEBUG
  printf("e putpts
");
  endif

  /*build pointer record*/
  pt = obuf;
  c = 'a';
  while (c != NULL)
    c = *pt++ = *fp++;
  while(pt < obuf + MAX_KEY +1)
    *pt++ = 'e';
  c = 'a';
  while (c != NULL)
    c = *pt++ = *bp++;
  while(pt < obuf + 2*(MAX_KEY + 1))
    *pt++ = 'a';
  *pt = END_OF_RECORD;
  if (write_rec(&os->btd,&os->btb,obuf,&cor_st->sci) == OK)
    return(OK);
  else
    return(ERROR);
}
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbup</td>
<td>E21</td>
<td>0</td>
<td>/*Getpts retrieves the forward and rear pointers after reading a set type</td>
</tr>
<tr>
<td>dbup</td>
<td>E22</td>
<td>0</td>
<td>record*/</td>
</tr>
<tr>
<td>dbup</td>
<td>E23</td>
<td>0</td>
<td>}</td>
</tr>
<tr>
<td>dbup</td>
<td>E24</td>
<td>0</td>
<td>Getpts (fp,bp,os)</td>
</tr>
<tr>
<td>dbup</td>
<td>E25</td>
<td>0</td>
<td>char *fp,*bp;</td>
</tr>
<tr>
<td>dbup</td>
<td>E26</td>
<td>0</td>
<td>struct ptrs *os;</td>
</tr>
<tr>
<td>dbup</td>
<td>E27</td>
<td>0</td>
<td>{</td>
</tr>
<tr>
<td>dbup</td>
<td>E28</td>
<td>1</td>
<td>#include &quot;ext.dec&quot;</td>
</tr>
<tr>
<td>dbup</td>
<td>E29</td>
<td>1</td>
<td>char Iobuf[MAX_RECORD + 1];</td>
</tr>
<tr>
<td>dbup</td>
<td>E30</td>
<td>1</td>
<td>char *pt;</td>
</tr>
<tr>
<td>dbup</td>
<td>E31</td>
<td>1</td>
<td>}</td>
</tr>
<tr>
<td>dbup</td>
<td>E32</td>
<td>1</td>
<td>#ifdef DEBUG</td>
</tr>
<tr>
<td>dbup</td>
<td>E33</td>
<td>1</td>
<td>printf(&quot;e getpts\n&quot;);</td>
</tr>
<tr>
<td>dbup</td>
<td>E34</td>
<td>1</td>
<td>#endif</td>
</tr>
<tr>
<td>dbup</td>
<td>E35</td>
<td>1</td>
<td>}</td>
</tr>
<tr>
<td>dbup</td>
<td>E36</td>
<td>1</td>
<td>if(read_rec(os-&gt;btd,os-&gt;btt,lobuf,&amp;cor_st-&gt;sci) == ERROR)</td>
</tr>
<tr>
<td>dbup</td>
<td>E37</td>
<td>1</td>
<td>return(ERROR);</td>
</tr>
<tr>
<td>dbup</td>
<td>E38</td>
<td>1</td>
<td>pt = lobuf;</td>
</tr>
<tr>
<td>dbup</td>
<td>E39</td>
<td>1</td>
<td>while((*fp++ = *pt++) != NULL) /<em>get forward pointer</em>/</td>
</tr>
<tr>
<td>dbup</td>
<td>E40</td>
<td>1</td>
<td>;</td>
</tr>
<tr>
<td>dbup</td>
<td>E41</td>
<td>1</td>
<td>pt = lobuf + MAX_KEY + 1;</td>
</tr>
<tr>
<td>dbup</td>
<td>E42</td>
<td>1</td>
<td>while((*bp++ = *pt++) != NULL) /<em>get backward pointer</em>/</td>
</tr>
<tr>
<td>dbup</td>
<td>E43</td>
<td>1</td>
<td>;</td>
</tr>
<tr>
<td>dbup</td>
<td>E44</td>
<td>1</td>
<td>return(OK);</td>
</tr>
<tr>
<td>dbup</td>
<td>E45</td>
<td>1</td>
<td>}</td>
</tr>
</tbody>
</table>
/* Valkr validates the key for an add of owner or member into set type and
   checks record against cor record type format. */

valkr (os, iobuf)
struct ptrs *os;
char *iobuf;
{
  #include "ext_dec"
  char key[MAX_KEY + 1];

  #ifdef DEBUG
  printf(" e valkr\n");
  #endif

  if (item_search(cor_st->sbtfd, cor_st->sbtroot, cor_unit, os) == PRESENT)
    return(kerror("key already present in set type", cor_unit));
  if (item_search(rbtfd, rbtroot, cor_unit, os) != PRESENT)
    return(kerror("record not present in database", cor_unit));
  if (read_rec(os->btd, os->btb, iobuf, &cor_rt->rc1) == ERROR)
    return(ERROR);
  if (bldkey (iobuf, key) == ERROR || strcmp(key, cor_unit) != MATCH)
    return(kerror("record doesn't fit set type", key));
  else
    return(OK);
/*Walks validates the keys input for change owner and change all commands*/

valks(newokey,os,saveos)
char *newokey;
struct ptrs *os,*saveos;
{
#include "ext_dec"

#define DEBUG

printf("e valks\n");
#endif

if (item_search(cor_st->sbtd,cor_st->sbroot,newokey,saveos) != PRESENT)
  return(kerror("new owner key absent from set",newokey));
if (item_search(cor_st->sbtd,cor_st->sbtrg,cor_unit,os) != PRESENT)
  return(kerror("key absent from set",cor_unit));
return(OK);
file

```c
#include "ext_dec"
char *pfb; /* pointer to fields of buffer as we move down*/
char *prk; /* pointer to fields of key as we move down*/
int 1, num;
int *nextkf; /* start of each key field*/
char *start;

nextkf = cor_rt->keyf;
prk = key;
for (1=0; 1<cor_rt->noskeyf; 1++, nextkf++)
{
    pfb = 1obuf;
    num = 0;
    while(num != *nextkf -1)
    {
        /* find right field*/
        if (*pfb == NULL)
            return(error("illegal key field"));
        else if(*pfb++ == cor_rt->fdlim)
            num += 1;
    }
    start = pfb;
    while (*pfb != NULL && *pfb != END_OF_RECORD &&
        *pfb != cor_rt->fdlim & prk < key + MAX_KEY)
        *prk++ = *pfb++;
    if (start == pfb)
        return(error("null key fields are not allowed"));
    if (prk == key + MAX_KEY)
        return(error("key exceeds max length"));
    *prk = NULL; /* end key string*/
    if (valchar(key) == ERROR) /* validate characters in key*/
        return(ERROR);
}

return(OK);
```
```c
valchar (key)

char *key;

/*validate characters in key are printable*/

/*ascii dependent*/

#endif DEBUG

printf ("e valchar\n");

Wendif

while (*key != NULL) {
    if (*key < ' ' || *key > '¨') {
        return(error("invalid character in key"));
    } else
        key++;

return (OK);
```
```c
#define DEBUG

void nfields (string, fdlm)
char *string;
char fdlm;

int num;

for (; *string != NULL; string++)
    if (*string == fdlm || *string == END_OF_RECORD)
        num += 1;

return (num);
```