SLAC-PUB-504
September 1968
(MISC)

DSM - A TEXT EDITOR WITH TIME REVERSAL CAPABILITY*

D. Ross, R. T. Braden,** R. E. Brody, J. O. Crowley,
D. B. Earl,† J. H. Halperin, R. M. Lonergan,†† and R. T. Stainton

SLAC Facility of the Stanford Computation Center
Stanford Linear Accelerator Center
Stanford University, Stanford, California

ABSTRACT

A conversational program for editing files of card images has the ability to "reverse time", in addition to the usual editing capabilities of inserting, deleting, and altering records of text. The program maintains a complete history of all editing actions, so it is possible to revive any previous editing state of a file.

New text is stored in chronological order, as it is introduced into the file. Once stored, the text never is altered or deleted. A set of pointers called a "file map" describes the correspondence between the physical placement of records of text, and their logical sequence within the file. Entries in the file map, reflecting editing changes to the file, also are stored in chronological order. Reversing time is accomplished by unlinking the most recent file map entries.

KEY WORDS AND PHRASES

editor, text editor, time reversal, conversational, file

CR CATEGORIES

3.73, 4.43

INTRODUCTION

DSM is a conversational program for editing files of text. In addition to the usual editing capabilities of inserting, deleting, and altering records of text, DSM

*Work supported by the U.S. Atomic Energy Commission.
**Currently at UCLA Campus Computing Network.
†Currently at Control Data Corp.
††IBM Corp.

has the ability to "reverse time". The program maintains a complete history of all editing actions, so it is possible to revive any previous editing state of a file.

The currently operating version of DSM is restricted to editing 72-column card images. Reviving a previous editing state is destructive of all states in the file which follow the revived state. These restrictions have been removed in a second version of DSM, which still is being coded. The second version will be described in this paper, since it is the more interesting of the two.

The files that DSM can edit are stored in OS/360 sequential data sets. DSM runs as a user job under OS/360 MFT or MVT, and is allocated data sets through standard Job Control Language DD statements. The data sets must reside on direct-access storage devices, such as disk or drum, while they are being edited.

Because DSM files maintain a history of the editing performed on their text, an internal bookkeeping system is required using a set of pointers called a "file map". A file consists of the text records and a file map. This "DSM internal format" is not compatible with other OS/360 processor programs, such as the FORTRAN compiler. DSM has commands for converting an "external format" (standard OS/360) data set into internal format for editing, and for converting from internal format to external format for compilation or other processing.

RECORD FORMATS

Text records in their external format may all be of any fixed length between 1 and 256 bytes, or they may be of variable length between 0 and 256 bytes. The user has the option of conserving storage space by compressing multiple blank characters in fixed length external format records, and storing the records in an internal format of variable length. Multiple blank characters in variable length external format records always are compressed.

DATA SET ORGANIZATION

Each data set may contain several files which can be edited independently. All text in the data set is stored in a single text area. The files are distinguished by their separate file maps. A file map describes the correspondence between the physical placement of records of text, and the logical sequence of the records within the file. Different files within a data set may share text in common, if their file maps point to the same text records.

All data in DSM internal format data sets are stored in blocks of fixed length. The blocks are numbered starting at 0. There are 3 types of blocks in a data set: bookkeeping blocks, file map blocks, and text blocks.

Block 0 is the first of the bookkeeping blocks. These blocks contain the names of all files in the data set, and the numbers of all file map blocks belonging to each file. The bookkeeping blocks also contain other information relevant to the entire

data set, including a description of the type of text (alphanumeric or numeric, record length, etc.), a pointer to the current end of the text, the number of the last block used, and the various language format parameters declared by the user.

Block 1 contains the file map for the first file in the data set. As editing is performed on the file, new entries are appended to the file map. When the file map becomes too large for a single block, it is continued on the next available block. Each file in the data set has its own blocks of file map.

Block 2 usually is the first block containing text. The text may belong to any or all of the files in the data set. As new text is introduced into any of the files, it is appended to the previously existing text. When text storage becomes too large for a single block, it is continued on the next available block.

Depending on record length, block length may not correspond to an integral number of records. Nevertheless, each text block contains an integral number of records. The beginning of the block coincides with the beginning of the first record in the block. Any space at the end of the block which is too small to hold the next record is wasted.

The remaining blocks in the data set contain extensions of the bookkeeping information or file maps or text, depending on the editing history of the files.

FILE ORGANIZATION

A file consists of file map and text. Access to the file map is made via information in the bookkeeping blocks. Access to the text is made via information in the file map.

As new text is entered into any file in the data set, it is appended to the existing text. Once stored, the text never is changed or deleted. The alteration of a string within a record is accomplished by appending a new record, containing the altered string.

New file map entries are created to reflect the consequences of the editing changes. The new entries are appended to existing entries, so their physical order of storage is the same as the chronological order in which the changes were made. Several sets of links maintain the logical order of the file map entries. Previous editing states of the file can be revived by unlinking the entries in the inverse order of their physical storage, which also is inverse chronological order.

An editing command which causes the file to be changed, also causes the counter of the current state number to be incremented by 1. DSM types the current state number to the user, prior to waiting for the next editing command. The command may cause several of the basic editing operations to be executed, such as insert, delete, copy, etc. DSM leaves a mark in the first file map entry of each state.

Later sections of this paper describe the 3 types of file map entries: directory entries, dummy entries, and state entries. All file map entries are of equal length, and are numbered 0, 1, 2, ... according to their physical placement within the file map.

RECORD NUMBERS, REGIONS, AND INTERVALS

A unique record number is associated with each text record in a file. Record numbers are stored in the file map, rather than with the text. A single physical text record may appear logically at several places in the file, with different record numbers.

A "region" of text is a set of records which are both physically consecutive in storage, and sequential in their record numbers. A region must be contained within a single block, in those data sets which have variable length internal format records. A region may span several consecutive blocks, in those data sets which have fixed length internal format records. Regions are defined such that any record can be located and accessed with at most one read operation, given the location of the first record in the region.

An "interval" is the set of records whose numbers lie between the lower and upper bounds of the interval, inclusive, regardless of where the records are stored.

Editing commands specify record intervals to be acted upon. The commands result in calls on the basic editing operations, such as: insert an interval of new text, delete an interval of existing text, or copy an interval of existing text so that it occupies a new interval. The basic editing operations process each region within the interval as a unit.

DIRECTORY ENTRIES

The file map contains one currently valid directory entry for each region of text in the file. After the first text is introduced into the file, the file map contains a single directory entry. As editing changes are made to the text, the initial correspondence between physical and logical ordering of the records ceases to be valid. The initial text becomes fragmented into several regions, and regions of new text are appended to the file. This causes new directory entries to be appended to the file map. Some of the existing entries may cease to be valid, because the regions they define have been deleted from the logical order of the file, or fragmented into smaller regions. All the entries which currently are valid, are linked together in ascending order of record numbers. Another set of links points from the new currently valid entries to the old formerly valid entries which the new entries displace. These links to displaced entries save the editing history of the file.

Directory entries contain the following fields:

(1) Forward link (2 bytes). This link contains the entry number of the next currently valid directory entry, in the order of ascending record numbers. Entry 0 of the file map does not correspond to a region, but points to the first and last currently valid directory entries.

(2) Backward link (2 bytes). This link contains the entry number of the preceding currently valid directory entry, in the order of ascending record numbers.

(3) Record number of the first record in the region (4 bytes).

(4) Count of the number of records in the region (2 bytes).

(5) Index to record number increments (1 byte). The increment is the difference between any two consecutive record numbers in the region. DSM recognizes only a few increments as valid: those which are 1, 2, or 5 * some power of 10. The valid increments are stored in a table which is part of the DSM program, but not part of any data set. This field contains an index into that table which identifies the increment used in this region.

(6) Text block number (2 bytes). This field contains the number of the block, relative to the start of the data set, which contains the first record in the region. The first record in the region may be anywhere within this block.

(7) Count of the number of intervening records (2 bytes). This is the number of records intervening between the start of block (6) and the first record in the region.

(8) Link to displaced directory entries (2 bytes). This link points from the logically first (lowest record numbers) of the new directory entries to the logically last of the old directory entries being displaced as a result of a single basic editing operation.

(9) Miscellaneous (1 byte). This field contains some redundancy for debugging purposes, and bits marking named records, control records, and the first directory entry in a new state.

The physical layout of the fields within a directory entry is not in the order they are described above, but rather in the order shown in Figs. 1, 2, 3, and 4.

EXAMPLE OF EDITING

Figs. 1, 2, 3, and 4 show an example of editing demonstrating the use of the file map. In this example, the records are of fixed length and of such a size that 6 records may be stored in each text block.

Initially the file is empty. The first text introduced into the file consists of 500 records. The text fills 83 blocks, and partly fills an 84th block. As described in "Data Set Organization", the first of the 84 blocks is block 2 in the data set. The records form a single region and the file map contains one directory entry, shown in Fig. 1.

The first editing change is the deletion of record 4, shown in Fig. 2. The text remains unchanged, but the file map is altered to indicate the deletion. Entry 1 in the file map is displaced by entries 2 and 3. Entry 3 corresponds to the region from record 5 to record 500. This region starts in block 2, with 4 records intervening between the start of block 2 and the start of the region. Observe that the forward link of the last of the displaced entries (entry 1) has been changed to point to the first of the displaced entries (also entry 1). This speeds the process of reviving former states.

The insertion of records 2.01, 2.02, and 2.03 is shown in Fig. 3. The new records are appended to the existing text, forming a new region. Records 1, 2, and 3 no longer form a single region, because the new inserted records interrupt their logical sequence.

The deletion of records 2.03 and 3, shown in Fig. 4, causes the displacement of 2 entries in the file map. The entry for records 2.01 to 2.03 is displaced by a new entry indicating the shorter extent of the region. The entry for record 3 is displaced, with no new entry replacing it.

DUMMY ENTRIES

Deleting an interval of text possibly may cause the displacement of previously valid directory entries, without creating any new directory entries. This occurs when the lower bound of the interval also is the lower bound of some existing region, and the upper bound of the interval also is the upper bound of some existing region.

In this event, a dummy entry is created solely to point to the displaced directory entries. The dummy entry occupies the same position among the forward and backward links that previously was occupied by the displaced entries. The only significant fields in a dummy entry are the forward link, backward link, link to displaced entries, and the bit in the "miscellaneous" field marking a new state.

STATE ENTRIES

The user may identify the current state of the file by its state number, which is typed out preceding each command, or by naming the state. Subsequently he may have that state revived, identifying it either by number or name.

State names and their corresponding numbers are stored in the file map in state entries. State entries are connected with backward links, on the assumption that users are most likely to be interested in the recently created states. Fig. 5 shows an example of state entries in the file map.

EXAMPLE OF EDITING, AND REVIVING BACKWARD AND THEN FORWARD IN TIME

Much of the complication of the file map can be eliminated for the illustration of this example. Fig. 6 shows a reduced diagram which represents both a directory

entry and its corresponding region of text. The letter A in the diagram represents the number of the directory entry in the file map. In the following discussion, "region A" means the text region corresponding to directory entry number A.

Fig. 7 shows the file after some initial editing, which has left the file at state number S. The forward and backward links indicate that the records in region B are numbered higher than the records in region A, the records in region C are numbered higher than the records in region B, etc.; but no assumption is made about the relative physical order of the directory entries numbered A, B, C, D, E.

In response to an editing command, an existing record interval which spans part of region B, all of region C, and part of region D, is replaced by new text. Fig. 8 shows the file map after the replacement. The new text is appended to all previously existing text, forming a single region at the end of text storage. The interval of new text coincides with this region, for which directory entry number J is created. Directory entry number I is created for the part of region B that was not replaced, and directory entry number K is created for the part of region D that was not replaced. Directory entry numbers I, J, K may be in any order among themselves, but as a set they must be contiguous, since they represent a single editing change. Numbers I, J, K must be greater than any of numbers A, B, C, D, E, since I, J, K are new directory entries which are appended to the file map. In order to make this example more concrete, it is assumed that J=I+1 and K=I+2.

Fig. 8 shows that the forward link of the last of the displaced entries (D) has been changed to point to the first of the displaced entries (B).

More editing could be performed subsequently, but at some later time the command is issued to revive a state numbered less than or equal to S. Reviving state S+1 leaves the file as shown in Fig. 8. Figs. 9, 10, and 11 show the process of reviving from state S+1 to state S. At each step in the process, the highest numbered directory entry is unlinked from its position in the file map, and replaced by any entries it has displaced. Entries K and J have not displaced any other entries, so they are unlinked without replacement. Entry I has displaced entries B, C and D. Entry B, the first of the displaced entries, has the same backward pointer as entry I. Entry B could have been found by following along the backward links from entry D, had its number not been put in the forward link of entry D. But following along the backward links could require unnecessary direct access I/O, if the file map occupies more than one block of storage.

Reviving forward again to state S+1 is just the reverse of the process shown in Figs. 9, 10, and 11. At each step, the lowest numbered directory entry is relinked into the file map. The backward and forward links of the new entry indicate which existing links must be broken. The link to displaced entries and the backward

link of the new entry indicate which existing entries become displaced. The steps in reviving forward are shown in Figs. 11, 10, 9, and 8.

Reviving forward is impossible once any editing changes are made. DSM has a command for duplicating a file map into a new file in the same data set. This allows the user to edit a file, and still keep a version that can be revived both backward and forward.

NAMED RECORDS

For certain types of text, it is natural to associate names with selected records of the text. For example, the statement number in a FORTRAN statement might be used to identify that record. DSM allows users to refer to named records by their names, as well as by their record numbers. Record names may be used in DSM commands in any way that record numbers may be used.

Routines appropriate to the text language being edited are called by DSM, to extract record names from the text itself. When new text is introduced into a file, these routines force DSM to start a new region at every named record. A bit in the "miscellaneous" field of the directory entry indicates that the first record of the region is a named record. Subsequently, when a record name is used, the routines scan all the named records until the desired record is located. In effect, record names invoke an automatic search of a selected subset of the records in the file.

CONTROL RECORDS

Certain editing applications can take advantage of DSM control statements embedded within the text. For example, when using DSM to compose business letters, a PAUSE control statement may be embedded within the text at the end of each page. DSM will pause in the display of the final "clean copy", allowing the typist time to insert a fresh sheet of paper. PAUSE also is useful when generating form letters, allowing the typist time to type in the recipient's name and address.

Like named records, records containing control statements always start a new region. A bit in the "miscellaneous" field of the directory entry indicates that the first record in the region contains control statements.

STORAGE CONSIDERATIONS

In a typical large file, say 5000 card images, the amount of space required for storage of the file map is insignificant. But if the file has been heavily edited, the data set may contain a large amount of obsolete text. Text storage space is not recovered when a previous editing state is revived, because the file might be revived forward again, or some other file might be sharing the text. The only way to recover space is to copy the currently valid text into a new file in another data set. The old data set then can be copied onto a permanent storage medium, such as tape, so the editing history will not be lost.

Our limited experience with the use of DSM has shown that once a file has been created and the irrelevant history cleaned out, about 1/3 to 1/2 of the subsequent changes will be incorrect, resulting in wasted text storage space. If the changes are minor, then the waste space will be insignificant. In one case, using DSM to modify its own source code from operation on an IBM 2741 typewriter terminal to operation on an IBM 2260 CRT display unit, the change required about 15% additional storage. Future modifications to enable operation on 2 other types of 2260 displays are expected to require about 10% more storage. We then will have 4 separate source programs, requiring only 25% storage more than any one of the programs.

Storage of only one record number per region, in the file map rather than with the text, makes a record length of 1 byte economical. Typical applications for 1-byte records are examining storage dumps, or inserting patches into programs without forcing recompilation.

EXECUTION CONSIDERATIONS

The only expensive operations in DSM are content searches of the text, and conversion from internal format to external format prior to compilation. Both these operations involve sequential processing of the records. An I/O scheduling routine tailored to sequential processing is used to minimize the number of direct access I/O calls. For other typical editing operations, statistical knowledge of the structure of the files is employed in I/O scheduling routines to minimize the expected number of direct access I/O calls. Many editing operations require no direct access I/O, since the last partly filled file map block and the last partly filled text block usually are retained in core memory buffers until they have been filled completely.

The only execution overhead directly attributable to maintaining the editing history is the possible I/O call necessary to change the forward link of the last displaced directory entry. During DSM format conversion there is some additional I/O required to rearrange the altered records of all previous editing runs, rather than just rearranging the altered records of the most recent editing run. This additional I/O properly should be charged to DSM's ability to keep record numbers constant from compilation to compilation, rather than to its ability to maintain an editing history.

The motivation for keeping record numbers constant from compilation to compilation is to enable print-off compilations after minor debugging changes. Any editor which allows random access changes to a file must have a final pass over all the currently valid text, rearranging it into sequential format for subsequent compilation. But keeping the record numbers constant prevents resequencing the file during the final pass. Therefore the increment to record numbers cannot be

uniform throughout the file. The space saving achieved by not storing the record numbers with the text leads to some pointer scheme such as the DSM file map, with its consequent additional I/O during format conversion.

MAINTAINING FILE INTEGRITY

Any scheme requiring the use of pointers is subject to complete failure if even one pointer is incorrect. In DSM, the pointers contained in the file map may require several blocks of storage. While one block of the file map is being updated to reflect an editing change, another block of the same file map still may contain pointers appropriate to the previous state of the file. If a system crash or hardware failure terminates DSM execution before all the relevant file map blocks have been updated, the file map will contain inconsistent pointers, rendering it useless.

During the editing of the file, DSM keeps two copies of the file map in secondary storage. One copy is in the permanent data set itself, and the other copy is in a temporary utility data set named "SYSUT1". One or more of the file map blocks from the copy in SYSUT1 are kept in core memory buffers, where they are updated to reflect each editing change. These blocks are written to SYSUT1 whenever their buffer space is needed for other blocks. Most of the time SYSUT1 contains an inconsistent file map, since some updated file map blocks still are in core memory.

At regular but comparitively infrequent intervals, all the file map blocks in core memory are written to SYSUT1, giving SYSUT1 a consistent file map. Then the file map in the permanent data set is updated from the contents of SYSUT1. At all times there is at least one consistent file map in secondary storage.

A "FAIL SAFE" message is issued each time the file map in the permanent data set is updated. System failure cannot cause the loss of editing done prior to the most recent "FAIL SAFE". Should the system crash during the updating of the permanent data set from SYSUT1, a later editing run may be used to restore the file map in the permanent data set.

THE EDITING LANGUAGE

In designing the editing language, we adopted the philosophy that typing a few extra characters in each editing command is an insignificant part of the total effort of editing a file. The editing commands were chosen for clarity and uniformity, rather than for brevity. A typical editing command might be:

TEXT AFTER 35

where 35 is the number of a record already existing in the file. This would be followed by typed-in records of new text. An extra carriage return for conversational editing, or a blank card for nonconversational editing, signifies the end of the text and makes DSM receptive to the next command.

Normally DSM chooses the starting record number and the increment to record numbers, using an algorithm which produces "pleasing" record numbers

based on an examination of the available interval into which the record numbers must be fit. The user can override this feature by specifying desired values of starting record number or increment.

The typical editing commands below give the flavor of the language:

```
DELETE 15 TO 17.2
COPY 7 TO 12 AT 20.1 STEP .005
SHOW 40 TO END
MOVE 43 OVER 50 TO 51
SEARCH BEGIN TO 30 FOR 'SIN(X)'
CHANGE * "SIN(X)"COS(Y-X)"
```

PROGRAM INSTALLATION

DSM operates either conversationally, using a 2741 typewriter terminal, or nonconversationally, obtaining commands and text from the SYSIN data stream. Conversational use of DSM requires 2741's with both the incoming and outgoing "break" features.

APPLICATIONS OF TIME REVERSAL

There are many application areas where it is important that files retain their history in a recoverable form. For example, files of medical records must be organized so as to allow convenient scanning of their entire history. Another example is the generation of computer programs, where small parts of the program may be experimentally modified. The ability to revive certain preselected editing states allows the modifications to be retracted if they prove undesirable, without requiring the storage of a second copy of the entire program.

A third example is the debugging of programs which modify files. For these programs, "snapshot and restart" methods (implied in [1]) are uneconomical, due to the cost of taking snapshots of the entire file being modified. Instead, these programs could be executed interpretively, with the interpreter recording the changes to all program variables. (An appropriate interpreter would have to be written for each source language.) When a program bug is detected, both the program variables and the files that were modified could be reversed in time until the program bug first appears. The time reversal would be under conversational control of the programmer. This method of debugging would obsolete most of the uses of break points, but could be coupled effectively with other conventional on-line debugging techniques [2].

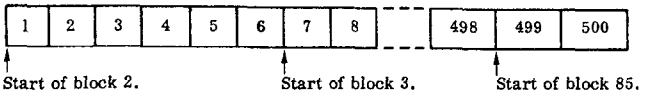
However, the most general application of time reversal is to a conversational text editor. Time reversal makes DSM tolerant of human errors. A mistyped digit in a record number, or an incorrect character transmitted over a communication line, could result in the accidental deletion of most of a file. In the absence of time reversal, these records would have to be typed in a second time. Other error-tolerant programs are designed to save computer effort when

errors are detected [3,4]. DSM is designed to save a significant amount of human effort when errors are detected.

REFERENCES

1. Van Horn, E. C. Three criteria for designing computing systems to facilitate debugging. Comm. ACM 11,5 (May 1968), pp. 360-365.
2. Evans, T. G., and Darley, D. L. On-line debugging techniques: a survey. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 37-50.
3. Irons, E. T. An error correcting parse algorithm. Comm. ACM 6,11 (Nov. 1963), pp. 669-673.
4. Wirth, N. A programming language for the 360 computers. Tech. Rpt. CS53, Computer Science Dept., Stanford University, Stanford, California (Dec. 1966).

TEXT:



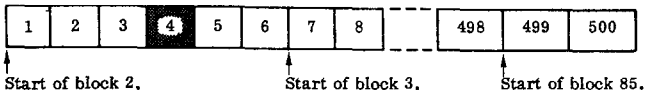
FILE MAP:

	Forward Link	Backward Link	First Record Number	Count of Records in Region	Count of Intervening Records	Block Number	Link to Displaced Entries	Increment Index	Misc. (in hex)
0:	1	1					0		00
1:	0	0	1	500	0	2	0	Index of 1.0	81

FIG. 1

Text and file map after the original 500 records have been introduced into the file.

TEXT:



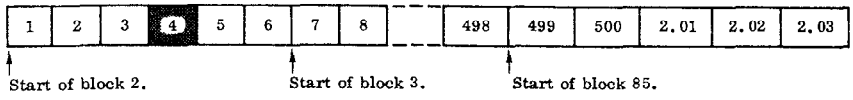
FILE MAP:

	Forward Link	Backward Link	First Record Number	Count of Records in Region	Count of Intervening Records	Block Number	Link to Displaced Entries	Increment Index	Misc. (in hex)
0:	2	3					0		00
1:	1	0	1	500	0	2	0	Index of 1.0	81
2:	3	0	1	3	0	2	1	Index of 1.0	81
3:	0	2	5	496	4	2	0	Index of 1.0	01

FIG. 2

Text and file map after deleting record 4. Record 4 has not been altered in storage.

TEXT:



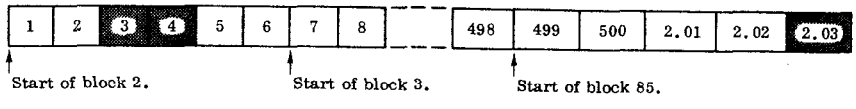
FILE MAP:

	Forward Link	Backward Link	First Record Number	Count of Records in Region	Count of Intervening Records	Block Number	Link to Displaced Entries	Increment Index	Misc. (in hex)
0:	4	3					0		00
1:	1	0	1	500	0	2	0	Index of 1.0	81
2:	2	0	1	3	0	2	1	Index of 1.0	81
3:	0	6	5	496	4	2	0	Index of 1.0	01
4:	5	0	1	2	0	2	2	Index of 1.0	81
5:	6	4	2.01	3	2	85	0	Index of .01	01
6:	3	5	3	1	2	2	0	Index of 1.0	01

FIG. 3

Text and file map after inserting records 2.01, 2.02 and 2.03.

TEXT:



FILE MAP:

	Forward Link	Backward Link	First Record Number	Count of Records in Region	Count of Intervening Records	Block Number	Link to Displaced Entries	Increment Index	Misc. (in hex)
0:	4	3					0		00
1:	1	0	1	500	0	2	0	Index of 1.0	81
2:	2	0	1	3	0	2	1	Index of 1.0	81
3:	0	7	5	496	4	2	0	Index of 1.0	01
4:	7	0	1	2	0	2	2	Index of 1.0	81
5:	6	4	2.01	3	2	85	0	Index of .01	01
6:	5	5	3	1	2	2	0	Index of 1.0	01
7:	3	4	2.01	2	2	85	6	Index of .01	81

FIG. 4

Text and file map after deleting records 2.03 and 3.

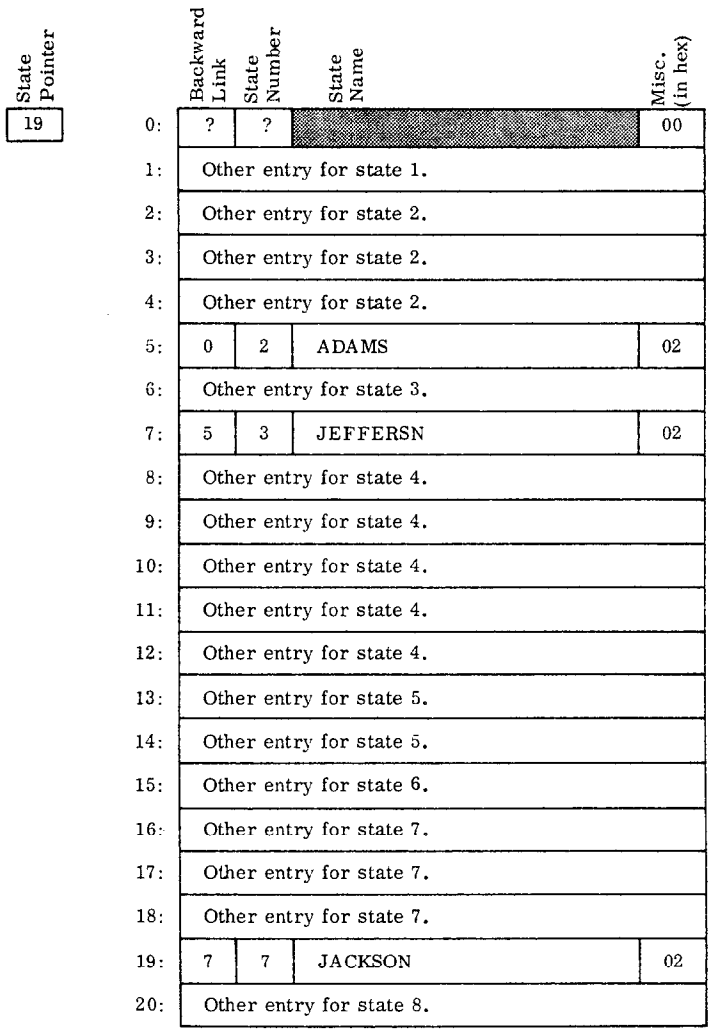
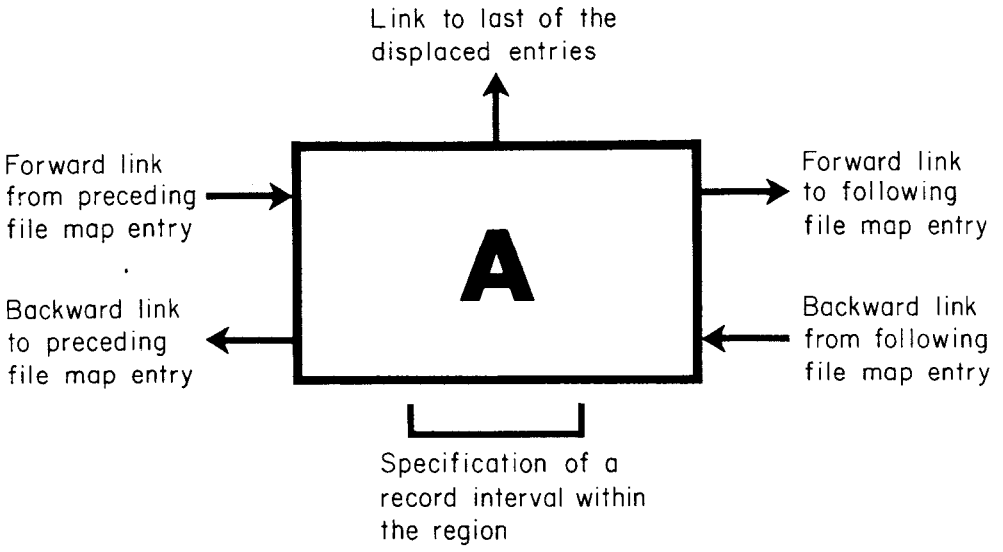


FIG. 5

File map containing state entries. JACKSON, JEEFERSN, and ADAMS are state names.



A = number of the directory entry, also used to identify the corresponding region.

1137AI

FIG. 6

Reduced diagram of both a directory entry and its corresponding region of text.

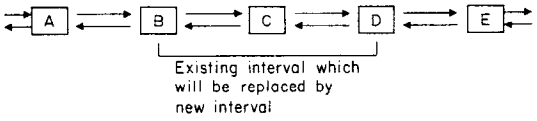


FIG. 7

Reduced diagram of the file after some initial editing.
State number = S.

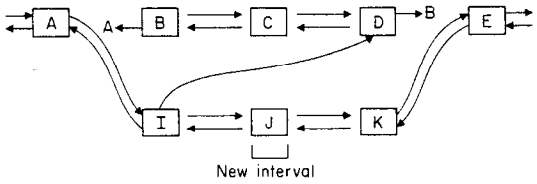


FIG. 8

Reduced diagram of the file after replacing the interval.
State number = S + 1.

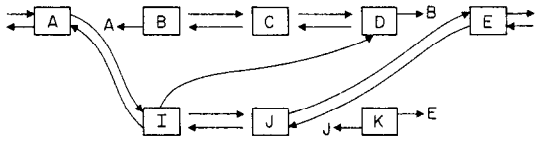


FIG. 9

Reduced diagram of the file after unlinking entry K.
State number S is being revived from state number S + 1.

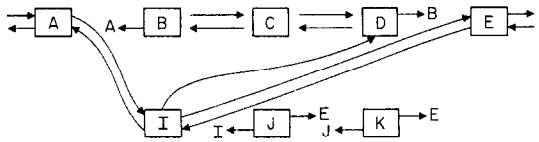


FIG. 10

Reduced diagram of the file after unlinking entry J.
State number S is being revived from state number S + 1.

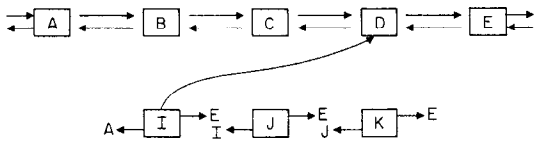


FIG. 11

Reduced diagram of the file after unlinking entry I.
State number = S.

1137B2