University of Wisconsin
Computer Sciences Department
1210 West Dayton Street
Madison, Wisconsin 53706

SAC-1A FOR PAGING: AN ADAPTATION OF SAC-1 FOR ALPHANUMERIC STRING LIST PROCESSING AND FOR A PAGING MACHINE.

by

William Fabens

Technical Report #83

March 2, 1970

SAC-1A FOR PAGING: AN ADAPTATION OF SAC-1 FOR ALPHANUMERIC STRING LIST PROCESSING AND FOR A PAGING MACHINE

by

William Fabens

Introduction

The collection of Algol routines I am about to describe represents a version of the SAC-1 List Processing systems with the following three additions.

(1) It is programmed in Algol instead of FORTRAN*,

(2) it adds conceptually two new types of lists; the character string and the compact alphanumeric word list; and

(3) the space used for representation of lists is paged, and algorithm for allocating space within this data area is implemented within the framework of the SAC-1 system.

The Base System: SAC-1

The SAC-1 List Processing system currently forms the foundation of the SAC-1 algebraic symbol manipulation system, which is being developed by G. E. Collins [1]. The List processing part is similar to the more well known SLIP system developed by Weizenbaum [2] in that it is an embedded list processing system which uses reference counts. SAC-1, however, is a descendant of REFCO I [3],

*The string processing routines but not the others are also programmed in FORTRAN (most of them in A. S. A. Fortran); system being described has been used on the Univac 1108 in FORTRAN, and on the Burroughs B5500 in Algol.

one of the first reference count systems. In SAC-1, a list is any set of cells addressed in the data area, the members of which are determined by following successor links. Each cell contains 4 fields: the successor field, the type field, the count field and the element field. The type field can have two values, 1 and 0. One indicates that the element field contains a pointer to a sublist, zero indicates that the element field contains an atomic element. The count field indicates how many pointers address the particular cell. A cell is put on available space list if and only if its count field becomes zero. Table 1 shows all of the SAC-1 list processing routines as they are used in SAC-1a. The table demonstrates them all translated into Algol. I/O routines are not shown.

Table 1: SAC-1 List Processing Routines

This chart shows examples of calls of all list processing routines, the variables used indicate the contents of variables or expressions that are used or expected with each routine. For their meaning see the chart at the bottom. Notes: 1 defines 1st arg., changes 2nd; 2 defines all arguments.

Informational routines: none of these alter list storage space		Storage modifying routines which do not increase or decrease amount of storage	
e: = first (c)	yield value of element field of cell c.	alter (e,c)	change element field of cell c.
ℓ: = tail (c)	yield value of successor field of cell c.	ssucc (ℓ,c)	change successor field of cell c.
t: = type (c)	yield value of type field of cell c.	stype (t,c)	change type field of cell c.
n: = count (c)	yield value of ref. count field of cell c.	scount (n,c)	change ref. count field of cell c.
m: = length (ℓ)	counts number of first-order cells on list ℓ.	ℓ1:=inv(ℓ2)	makes each cell in ℓ2 point to preceding cell, returns this <u>in</u>verted list as ℓ1
adv(e,c)[1]	delivers element field of cell c as e, and successor field as c.	ℓ1:=conc(ℓ2,ℓ3)	makes last cell of ℓ2 point to ℓ3, returns this <u>conc</u>atenated list as ℓ1
		ℓ1: = borrow (ℓ2)	if ℓ2 is non-zero, increase increases its ref. count by 1. Returns value of ℓ2 to ℓ1.

Storage increasing routines: routines which generally use some of available space.		Storage decreasing routines: routines which generally return some of available space.	
ℓ1: = pfa(a, ℓ2) ℓ1: = pfℓ(ℓ3, ℓ2)	creates new cell with type $\begin{smallmatrix}0\\1\end{smallmatrix}$, element $\begin{smallmatrix}a\\ \ell 3\end{smallmatrix}$, count 1 and successor ℓ2, calls this cell ℓ1. Thus $\begin{smallmatrix}a\\ \ell 3\end{smallmatrix}$ is prefixed list ℓ2:	decap(e,c)[1]	yields element field of c as e, and the successor field as c. Then returns this cell to available space. Ref. count of the cell must have been 1.
		erase(ℓ)	If ℓ is not zero, reduces its ref. count by 1. If that makes the ref. count 0, return the cell to available space, and use erase on the successor cell, if cell had type 1, also erase the sub-list.
ℓ1: = cinv (ℓ2)	creates (length(ℓ2)) new cells in reverse using pfa or pfℓ, whose element fields are the same as those of ℓ2. If their type is 1, borrow them.	erla(ℓ)	ℓ is a list where no cell has type 1, erase algorithm (simplified) is applied.

stack2(e1,e2)
stack3(e1,e2,e3)
	pushes (e3 then) e2 then e1 onto list called stack.	unstk2(e1,e2)[2]
		unstk3(e1,e2,e3)[2]
			pops e1 then e2 (then e3) off of list called stack.

c = any non-zero address	a = any computer word (atom)	e = any a or ℓ (atom or list
n = any non-negative integer	ℓ = any c, or 0 (list)	
m= any non-negative integer	t = 0 or 1	

SAC-1a: Additions for Alphanumeric-String List Processing

The list processing system just described provides an efficient base for handling general lists. SAC-1a builds on it a basis for alphanumeric string processing. SAC-1a adds conceptually two new types of lists to SAC-1: packed and unpacked a-lists ('a' for alphanumeric). A-lists are handled using new SAC-1a routines programmed in SAC-1. These routines handle the basic operations of I/O, conversion, comparison and pattern-match replacement. The resulting set of routines is the SAC-1a system.

Packed and Unpacked a-lists

I will refer to both lists and strings in this discussion of the new routines. In doing so, I use 'string' to refer to the contents of a 'list,' the terms are complementary, not contrasting. An <u>unpacked a-list</u> is a first-order SAC-1 list where each cell contains one character. This method of storing strings is efficient only when it is used for character-by-character pattern matching. The <u>packed a-list</u> is the usual form of string storage. Each a-word, composed of non-blank, non-parenthesis alphanumeric characters, is stored 5 or 6* symbols per cell with special symbols to indicate continuation. This format can be used for word-by-word pattern matching. Blanks never appear in packed lists since they are what were used to delimit words; parentheses never appear either - they are represented by sublisting.

*5 or 6: depends on whether one is using 36 or 48 bits for SAC-1 atoms.

Input/Output

Each type of list has its associated reading and writing routines. Aread causes as many cards (or records) to be read as it takes to come to matching parentheses. Words are delimited by blanks and are packed 6 (or 5)* letters per SAC-1 cell in packed a-list form. Parentheses causes sub-a-lists to be inserted in the a-list which Aread creates. Awrite causes a packed a-list to be written in parenthesized notation (such that it could be read in as an equivalent list if one used Aread on Awrite's output). Cread causes the reading in of an unpacked (character) list. Reading continues until a special end-of-string symbol is found; each character encountered before the end-of-string is placed in a single SAC-1 cell. Cwrite causes character lists to be written (in a compatible format - followed by the end-of-string symbol). The two reading routines can fail due to lack of input (end-of-file). If so, they return -1 instead of a list.

Conversion

Lists can be packed and unpacked using the two conversion routines: S1toS6 (or S1toS5)* and S6toS1 (or S5toS1)*. S6toS1 converts packed a-lists to unpacked a-lists. It inserts parentheses to surround sublists, and blanks to separate words. It is parallel to Awrite in operation. S1toS6 converts from unpacked to packed except that parentheses do not cause sublists to be created since (1) the parentheses may not match, and (2) there are no outside parentheses

---

*on 36-bit machines

to delimit shows the end of the list. Otherwise it is like Aread. This absence of conceptual symmetry due to parenthesization may be reprogrammed and designed out in later versions.

Comparison.

Two comparison routines have been programmed. The first, streql (for 'string equal'), can be used to compare two unpacked strings or two packed strings. It simply scans down both strings, testing for equality. If each atomic cell agrees with each corresponding cell, it returns true, otherwise false. The scan stops when the scan encounters either the end of a string or the beginning of a sublist. With packed lists, which may contain sublists, the comparison is thus restricted to the left-most cells - this way one can compare a simple packed string with the the beginning of a complicated a-list.

The other routine is called treeql (for 'tree-equal'). This routine scans the complete list structure of each of its two arguments. If the two structures are equivalent and the corresponding atoms within these are equal, the value is true, otherwise false.

These routines are of type boolean and neither one of them alters its arguments.

Pattern-Match Replacement

There are two boolean routines, replst (for 'replace string') and repltr (for 'replace tree') of three list arguments which look through the first list for an instance of match with the second and if it is found, replace that matched part by the third argument. They differ from each other in where they look and what kind of equality they are looking for. These routines are similar to the basic statement types of COMIT [4] and SNOBOL [5]. Using replst with the other SAC-1a routines, one can achieve a rough equivalence of the SNOBOL pattern match except for the matching of variable-length unknown strings and all that that entails.

Replst (A,B,C) searches along string A, trying to find a sequence of words in it which match the sequence B. If it finds them, it replaces that sequence with a copy of string C, and returns true. Otherwise nothing is modified and the value is false. A and B (and usually C) ought to be all packed or all unpacked.

Repltr (A,B,C) tries to find a match of B in the treeql sense, first with A, then each of A's sublists (if any) going down then across, left-most first. If it finds a match, C is borrowed and put in the place of the match. In that case the value will be true, otherwise nothing changes and the value is false.

Either of these routines, if successful, will modify the list A. If their matches include the first cell of A, the address A itself will be changed.

SAC-1a Adaptation to a paging machine.

Paging Problems

The SAC-1a system is implemented on a computer, the B5500, which handles arrays using disk pages. A particular page in general does not reside in memory unless it has recently been referenced, but list processing usually makes references to words located randomly in a large, multi-page array. Without some arrangement to control the randomness of such references, time consuming page fluttering could easily result.

List Entropy

Looking at the problem a little more closely, the reason that systems can start to flutter pages is mainly attributable to the handling of available space. The list of available cells is initialized linearly throughout memory. When a new cell is needed, it is taken off the top of this list. In the beginning this causes no problem. The problem begins either when cells become cross-referenced, or more commonly when cells are returned to that list of available cells. If cells which had occurred in the middle of the list are returned to one end of it, available space can double back across page boundaries. As more cells are returned, the available space list can cross more and more page boundaries. Then when new cells are needed their locations can become more and more haphazard simply because of the increasing randomness of available space, this is the problem of list entropy.

The solution to list entropy is simple: maintain a list of available space on each page. SAC-1a does this in its three cell-returning procedures.

Page Links

Let us call any two pages in which a cell of one references a cell of the other, 'linked pages.' Inevitably with any list processing system, there will be linked pages. One can only hope to keep this number to a minimum. With most list processing applications, programmers can create page links by putting references on one list to a sublist on another page. This type of natural page links cannot be prevented.* The artificial kind, caused by overflowing page space when building a list, can.

The method this program uses to overcome artificial page links along with the rationale for it, follows: The only time for prevention of artificial page links comes when a new cell is being prefixed to a list (since remapping is not allowable). Thus the method becomes one of deciding which page should be used for the new cell in the prefixing routines pfa and pfl.

Same Page. One would like to choose to take a cell from the same page as the cell being prefixed to. But complications can develop: (1) no such page address is given (e.g. when prefixing to a null list), or (2) the indicated page has no more cells. Last-used page. In the first case, one can choose, say, the last page which was referenced (since this is the most likely page to actually be in storage), but even after that, one may run into problem 2.

*except through a remapping process - in SAC-1 unlike Slip, any cell is referencible not only through programmed variables but from any number of other cells. So with SAC-1 remapping is not feasible.

List of Available Pages

This system, on encountering case 2, chooses its page from one of two sources: a list of available pages and if none exists, creates and initializes a new page. The list of available pages is maintained through the erasure routines. A page becomes available if some cells have been erased and that makes the page empty.

Successor Page

Now, if after case 2 has arisen because page a was desired but full and page b was used for the rest of the list, it would be a good idea to remember that b follows a so if case 2 would arise with respect to page a again one could try b again. To implement this each page contains a pointer to its successor page (if any). Before trying a new page the successor page is tried. If that page is found full it is removed as successor to page a, and a new (available) page is used and called a's successor. Otherwise it is used and the number of page links stays the same.

Page Cushions

If cases 1 occurs but the latest page is almost full, it makes no sense to begin a fresh list on it. If this happens, a new (available) page is taken. The other way around, if a page becomes nearly empty it can be uneconomical to tie it up as the successor page of some other page. Thus when erasing causes a page to become nearly empty it is split off from the predecessor and put on the available page list. What 'nearly full' and 'nearly empty' mean is programmable by means of two variables 'PAGEMEPTY' and 'PAGEFULL.' Tables 2-8 show the effects of various setting of PAGEFULL and PAGEEMPTY on a list application which is not atypical. Figure 1 shows an example of the arrangement of pages in the running of the system. The size and number of pages can also be determined by the programmer.

Summary and Discussion

This paper deals with two essentially unrelated topics: string manipulation in a list processing system, and storage maintenance for un-remappable list storage.

As with Collins' SAC-1 system the string manipulation routines given here are a minimal rather than an exhaustive set of routines upon which one can build his own type of system. One can build other routines around this set to achieve some of the advantages other string manipulation languages like SNOBOL offer, without carrying the overhead of unwanted generality.

The storage maintenance part seems to run well. Actual experience running the system shows it to be viable in a virtual paging system, and it seems to be closely competitive with non-list processing programs written for similar problems.

Appendix:

SAC-1a I/O for the B5500 - extensions of SAC-1 I/O.

The B5500 offers essentially two features of I/O which are not always found or necessary on non-interactive machines: teletype I/O and run-time assignment of disk files. To take advantages of these, the I/O routines were augmented and a new routine was added.

Each SAC-1 I/O routine is given a unit number (which represents printer, card reader, etc.) as one of its arguments. Since the B5500 refers to all its I/O units as files (with names, not numbers), and since some of these files have different attributes (e.g. not backspacable, random addressibility, etc.). much more information about a file must be given and used than just a Unit Number.

To supply this information with as little inconvenience to the programmer as possible, an new routine was added. It is called DECLARE and does all file assigning, releasing and status and presence checking. If its arguments are (1a) a number saying what type of file is being declared, and (1b) the name (if any) to be given to the file or that the file should have, DECLARE returns either a Unit Word (coded with bits telling the I/O procedures how to handle I/O on that unit) or a negative number indicating that the file has bad priority doesn't exist or is doubly defined. If its argument is (2) the negative of a Unit Word, DECLARE either releases, disconnects or purges the file. It is the 'Unit Word' that corresponds to SAC-1's 'Unit Number', so once a file's Unit Word has been defined, one uses it just as one would in SAC-1.

Variables exist in the system whereby a user can find out where his record pointer is on each randomly accessible file. The user can have his read or write command use a particular record by setting a bit and an address field in the Unit Word, otherwise serial operation is assumed. To rewind a file the user calls <u>read1</u> with a negative unit number.

All I/O, not just list processing I/O can be done using DECLARED Unit Words: there are procedures, which use the basic SAC-1a routines, which are given lists and formats.

<u>Teletype I/O</u>

Teletype files need not be declared - such declaration is done automatically when reading or writing. There are two ways of doing teletype I/O: multiple user round-robin input and selected input (they converge when only one teletype is connected). For multiple teletypes, the Unit Word equals 0 and need not be explicitly DECLARED. If more that one teletype is attached, reading is done from the next one inputting in the round-robin circle. A global variable keeps track of the number of the referenced teletype. Writing on Unit 0 causes output on the last-read teletype. (If 0 teletypes are connected, I/O goes to file "printer" and comes from file "cards.") Individual teletypes are addressed using their teletype number as the Unit Word.

References

1. Collins, G. E., "The SAC-1 List Processing System," Computer Sciences Department and Computing Center, University of Wisconsin, Madison, Wisconsin, (July 11, 1967).

2. Weizenbaum, J., "Symmetric List Processor," Comm. ACM, Vol. 6, No. 9. (September 1963), pages 524-544.

3. Collins, G. E., "A Method for Overlapping and Erasure of Lists." Comm. ACM, Vol. 3, No. 12. (December 1960).

4. Sammet, Jean E., "COMIT" in Programming Languages: History and Fundamentals Prentice-Hall, Inc. Englewood Cliffs, N. J. (1969), pages 416-436.

5. Farber, D. J., Griswold, P. E., and Polonsky, I. P., "SNOBOL, A String Manipulation Language," J. ACM., Vol. 11, No. 1. (January 1964) pages 21-30.

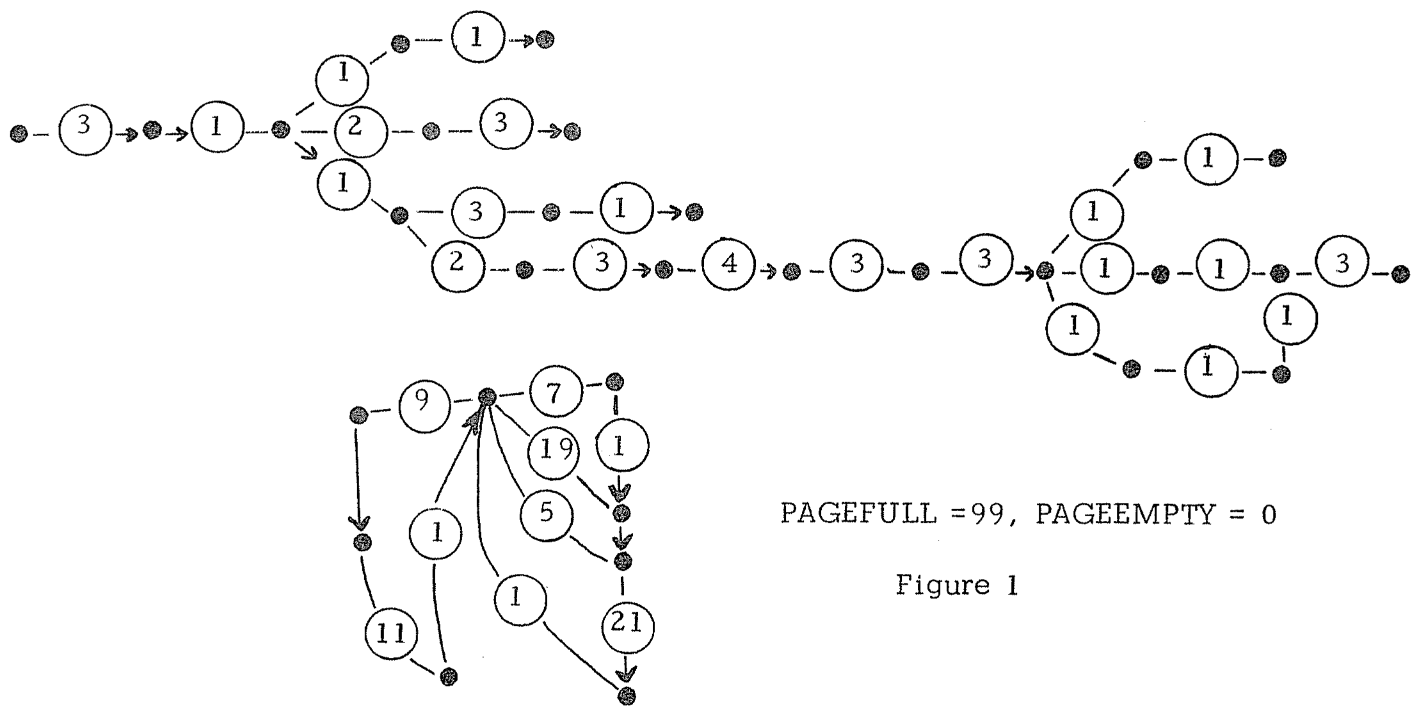
Tables 2-8. Various settings of PAGEFULL and PAGEEMPTY were tested by doing a certain amount of processing, stopping and recording various statistics on the state of the system, then restarting and doing the same exact processing with different settings. The statistics are described below. In each case the other variable, page size, was kept at 99 cells per page. N Number of cells in use was always 2236.

Table	Statistic	Explanation
2	Links	number of pairs of pages, one of which refers to the other
3	Pages	number of pages with cells in use on them
4	Cells per Page &б	average number of cells in use per page, and root-mean-square of cells per page minus the average
5	Full	number of pages with no more available space on them
6	Partial	number of pages with some cells in use, some not
7	Density of links	the percent the number of page links is to the number of possible page links ($= \frac{n(n-1)}{2}$, n = number of pages).
8	Empty	number of pages created at one time but not now containing data

Figures 1 and 2:

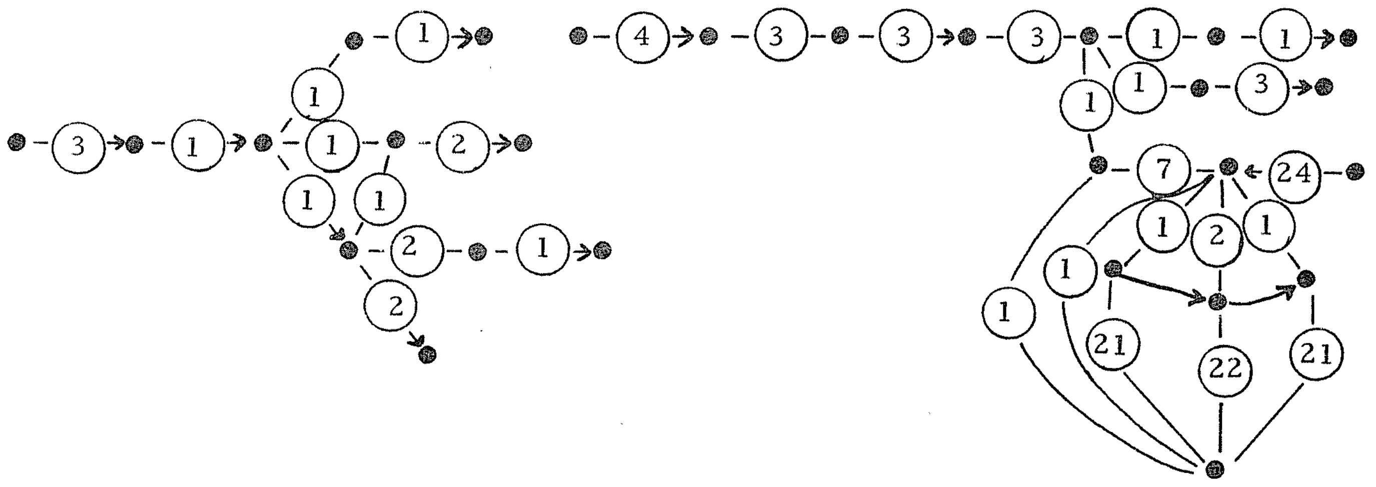
These pictures show the arrangement of pages under two of the 66 settings where statistics were collected in tables 2-8.

The numbers between dots are the numbers of page-linking references. Arrows show the current successor pages of each page. The dots are pages.



PAGEFULL =99, PAGEEMPTY = 0

Figure 1



PAGEFULL = 95, PAGEEMPTY = 50

Figure 2

R/6

PAGE EMPTY =

PAGEFULL =	0	10	20	30	40	50	60	70	80	90	99
99	26	27	27	27	27	26	27	27	28	31	33
90	29	28	28	28	29	28	28	28	29	29	29
80	31	30	31	30	30	30	29	30	30	31	31
70	29	29	29	29	29	29	29	32	32	32	32
60	34	34	34	34	34	33	33	33	33	34	34
50	36	38	38	35	35	34	35	35	36	36	36

LINKS

Table 2

PAGE EMPTY =

PAGEFULL =	0	10	20	30	40	50	60	70	80	90	99
99	28	27	27	27	27	26	26	26	26	27	28
90	29	28	28	28	28	27	27	27	27	27	27
80	30	29	29	29	28	28	28	28	28	28	28
70	27	27	27	27	27	27	27	29	29	29	29
60	33	31	31	31	31	29	29	29	29	29	29
50	37	35	35	34	34	32	32	32	32	32	32

PAGES

Table 3

PAGE EMPTY =

PAGEFULL =	0		10		20		30		40		50		60		70		80		90		99	
99	80	28	83	24	83	24	83	25	83	25	86	22	86	22	86	22	86	23	83	26	80	30
90	77	31	80	27	80	27	80	27	80	28	83	26	83	26	83	26	83	27	83	27	83	27
80	74	31	77	28	77	28	77	28	80	28	80	28	80	28	80	28	80	29	80	29	80	29
70	83	19	83	19	83	19	83	19	83	19	83	19	83	19	77	28	77	28	77	28	77	28
60	68	33	72	29	72	29	72	29	72	29	77	23	77	23	77	23	77	23	77	23	77	23
50	60	31	64	29	64	29	66	29	66	29	70	24	70	24	70	24	70	24	70	24	70	24

Cells per page & 6

Table 4

PAGE EMPTY =

PAGEFULL =	0	10	20	30	40	50	60	70	80	90	99
99	15	15	15	15	15	15	15	15	16	16	18
90	13	13	13	13	13	13	13	13	14	14	14
80	12	12	12	12	12	12	12	12	13	13	13
70	9	9	9	9	9	9	9	12	12	12	12
60	11	11	11	11	11	9	9	9	9	9	9
50	11	11	11	11	11	8	8	8	8	8	8

FULL

Table 5

PAGE EMPTY =

PAGEFULL	0	10	20	30	40	50	60	70	80	90	99
99	13	12	12	12	12	11	11	11	10	11	10
90	16	15	15	15	15	14	14	14	13	13	13
80	18	17	17	17	16	16	16	16	15	15	15
70	18	18	18	18	18	18	18	17	17	17	17
60	22	20	20	20	20	20	20	20	20	20	20
50	26	24	24	23	23	24	24	24	24	24	24

PARTIAL

Table 6

PAGEEMPTY =

PAGEFULL	0	10	20	30	40	50	60	70	80	90	99
99	6.9	7.7	7.7	7.7	7.7	8.0	8.3	8.3	8.6	8.8	8.7
90	7.1	7.4	7.4	7.4	7.7	8.0	8.0	8.0	8.3	8.3	8.3
80	7.1	7.4	7.6	7.4	7.9	7.9	7.7	7.9	7.9	8.2	8.2
70	8.3	8.3	8.3	8.3	8.3	8.3	8.3	7.9	7.9	7.9	7.9
60	6.4	7.3	7.3	7.3	7.3	8.1	8.1	8.1	8.1	8.4	8.4
50	5.4	6.4	6.4	6.2	6.2	6.9	7.1	7.1	7.3	7.3	7.3

DENSITY OF LINKS

Table 7

PAGE EMPTY =

PAGEFULL	0	10	20	30	40	50	60	70	80	90	99
99	3	3	3	3	2	2	2	2	2	2	0
90	1	1	1	0	0	0	0	0	0	0	0
80	0	0	0	0	0	0	0	0	0	0	0
70	0	0	0	0	0	0	0	0	0	0	0
60	0	0	0	0	0	0	0	0	0	0	0
50	0	0	0	0	0	0	0	0	0	0	0

EMPTY

Table 8