PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 4/71

A COMPARATIVE STUDY OF SYMBOL TABLES

by

Marcelo Pardo Brown

Computer Science Department - Rio Datacenter

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

A COMPARATIVE STUDY OF SYMBOL TABLES

Marcelo Pardo Brown

Computer Science Department
PUC/RJ

Series Editor: A. L. Furtado

July/1971.

ABSTRACT

Since symbol table look-up appears so frequently in everyday programming, an appropriate choice of the techniques to be employed strongly reflects on the efficiency of such programs.

The purpose of the present work is to compare the best known symbol table construction and search techniques, with regard to processing time and core storage requirements.

An attempt is then made to establich some criteria that would indicate which technique should be used for a particular application.

Experiments were run on an IBM-7044, using from one to six characters (one machine word) for symbol names. Modified algorithms to work with variable length symbols are presented but no measurements are given for these.

1. SEARCH TECHNIQUES.

If a name is to be associated with a given value, one can consider this value as the result of a function applied to this name.

$$f(\text{name}) = \text{value}$$

To find this function, when there is no name-value relationship can become quite a cumbersome task since the search tecniques simulate this function with greater speed and efficiency.

As an alternative to the application of this function one can create two associeted vectors in which name_i - value_i are stored. These associated vectors are known as symbol tables.

The table contains only those name_i-value_i's for which the function is defined.

$$f(\text{name}_i) = \text{value}_i$$

there is no relationship such as

$$f(\text{name}_k) = \phi(\text{empty})$$

The search is done by looking-up the name_m. If there is no such name an exception condition exists. If there is such a name the answer will be its associated value_m.

SYM is defined as the name vector, an overall naming for symbols.

ATR is defined as the value vector attributed to the symbols. This vector might contain alphabetic symbols (as in a dictionary), alphanumeric symbols, values (characteristics of the associated symbols) or pointers to to structures with information about this symbol. The nature of the associated value is different for every table use.

We define field as a unit of information. The field length depends' on the stored information. Obviously the length for every field in a vector is constant and equal to its largest element. Due to this fact a word can be a fraction of the field, equal to the field or it might contain a number of fields of information.

When dealing with algorithms it is supposed that a missing symbol in the table implies that it is included in the table.

On the other hand if the symbol is in the table it will produce the associated value as output.

As a basis for comparison let us consider the most intuitive technique: linear search. It is used for i symbol fields ($1 \le i \le N$, N is the present number of symbols in the table); the search terminates if the symbol is found or if there is no such symbol. In this case it is inserted $N = N+1$ positions alread, if and only if $N<NMAX$; NMAX is the upper limit of vector SYM.

Algorithm

SYM	symbol vector
ATR	attribute vector
SYMSCH	searched symbol
ATRINC	included attribute
N	present table size
NMAX	upper limit table.

```
LS1. - I ← 0;
LS2. - I ← I+1;
LS3. - If (I>N)  ⇒ go to LS4.
       If (SYM(I) = SYMSCH) ⇒ 'answer' ← ATR(I), END.
       go to LS2;
LS4. - N ← N+1;
       If(N>NMAX) ⇒ 'overflow',END.
       SYM(N) ← SYMSCH;
       ATR(N) ← ATRINC, END;
```

The best known construction and search techniques for symbol table are described below.

1.1 - BINARY SEARCH

Area is defined as one or more consecutive fields of vector SYM.

Let us suppose that the elements in vector SYM are ordered. The search for a symbol in this vector is done in the existing symbol area. An attempt is made to match the searched symbol with the symbol located in one-half the area used;

if the searched symbol is greater than this one-half area, the searching area is reduced to its upper half, while if it were less than the former area it would be reduced to its lower half, if it is equal, the desired output is given.

This process is repeated in the manner described above until the rigth answer is output.

If the searched symbol is not found it is included in vector SYM by moving up the other symbols, so as to maintain the same order.

BS. Algorithm

SYM	symbol vector
ATR	attribute vector
SYMSCH	searched symbol
ATRINC	included attribute
N	present table size
NMAX	upper limit table

```
BS  1. - LUPP ← N+1 ; LLOW ← 1;
         If (N<1) ⇒ I ← 1  and go to BS8.
BS  2. - |Computation of the overage symbol of the area|
         I ← |(LUPP + LLOW)/2|;
BS  3. - If(SYM(I) > SYMSCH) ⇒ go to BS4.
         If(SYM(I) < SYMSCH) ⇒ go to BS5.
         'answer' ← ATR(I), END;
BS  4. - If (LUPP = I) ⇒ go to BS7.
         |Modification of upper limit|
         LUPP ← I  and go to BS2;
```

```
BS  5. - If (LLOW = I) ⇒ I ← I+1  and go to BS6.
         |modification of lower limit|
         LLOW ← I  and go to BS2;
BS  6. - IF(N = LLOW) ⇒ go to BS8.
BS7.   - |SYMSCH is not in the table|
         If(N+1 > NMAX) ⇒ 'overflow', END.
         ∀K((K = N,N-1,N-2,...,I) ⇒
              SYM(K+1) ← SYM(K);
              ATR(K+1) ← ATR(K););
BS8.   - N ← N+1; SYM(I) ← SYMSCH;
              ATR(I) ← ATRINC;
```

This algorithm can be modified in such ways as to use a displaceable pointer vector (see 1.5.1) for a fixed table optimizing construction time. On the other hand this algorithm optimizes search time which is its best feature.

1.2 - EXTENSION TO TERNARY SEARCH

It was thought that if the same principles used in the binary search technique were used in a n-nary search technique one could divide the search area in n sub areas, trying n-1 matches in the sub area in which the process would be repeated. If on one hand the subfields are reduced to 1/n por every iteration, on the other hand the search is increased by n-1 matching attempts. For n = 3 ternary search a number of experiments were made.

TS Algorithm

SYM	vector
ATR	attribute vector
SYMSCH	searched symbol
ATRINC	included attribute
N	present size
NMAX	upper limit table
I1	upper average limit
I2	lower average limit

```
TS1  LUPP ← N+1; LLOW ← 1;
     If(N<1) ⇒ I←1  and go to TS7.
TS2.-I1 ←|(LLOW + LUPP + LUPP)/3|;
     I2 ←|(LLOW + LLOW + LUPP)/3|;
TS3.-If(SYM(I1)<SYMSCH) ⇒ go to TS4.
     If(SYM(I1) = SYMSCH) ⇒ 'answer'← ATR(I1), END.
     |New upper limit|
     LUPP ← I1 and go to TS5;
TS4.-If(LLOW ≠ I1) ⇒LLOW←I1 and go to TS2.
     I ← I+1  and go to TS6;
TS5.-If(SYM(I2)< SYMSCH) ⇒ LLOW ← I2 and go to TS2.
     If(SYM(I2)= SYMSCH) ⇒ 'answer'  ATR(I2), END.
     LUPP ← I2 and go to TS2;
TS6.-|Morning up|
     If(N = LLOW) ⇒ go to TS7.
     If(N+1>NMAX) ⇒'overflow', END.
     ∀K((K = N,N-1,N-2,...,I)
     SYM(K+1) ← SYM(K);
     ATR(K+1) ← ATR(K););
TS7.-N ← N+1;
     SYM(I) ← SYMSCH;
     ATR(I) ← ATRINC, END;
```

1.3 - TABLE IN A BINARY TREE STRUCTURE

As its name indicates the structure of this table is that of a binary tree. Every node in the tree has a four field set.

SYMBOL	ATTRIBUTE	LEFT NODE	RIGHT NODE

In the following example only the name of the symbol as a node is indicated.

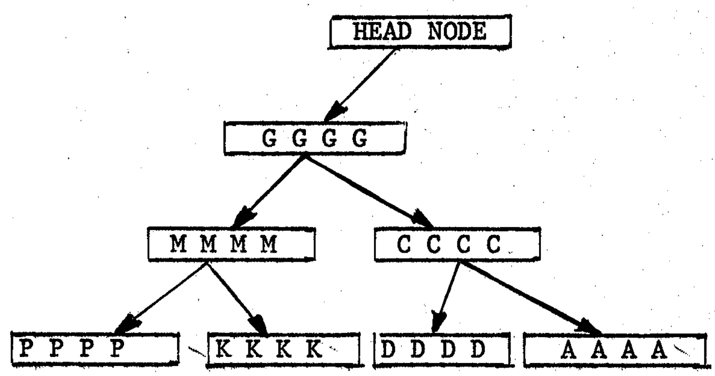


TABLE IN BINARY TREE STRUCTURE

FIGURE 1

The construction of that tree is done in the following way: The first node to be inserted is the root of the tree. The second one might be placed either at the right or the left of the root depending on whether it is greater or less than the root. The following symbols will be inserted in the same way for every node in the subtree. The searching is done analogously, but with a different action with respect to the associeted attribute.

It is a different approach to the binary search since the search areas are divided after the matching is attempted with the root of the selected subtree. (When dealing with a symmetric tree the search area is reduced to one half the preceding one).

This method has the advantages of avaiding displacement in the table in its construction phase; the searching time is less since there is no need to compute the search limits. The difference can be considered small.

Its other advantages are:

a) Depending on the order in which the symbols are input, a non symmetric tree can be generated. This results in an increase in matching attempts, with a heavier populated sub tree. This problem could be overcome if a symmetric tree could be created, in creasing, nevertheless, processing time and memory space.

b) When dealing with the four field node, used memory space is also incresed, since the node is two fields longer than the binary search node.

This sort of method is used in fixed tables (see 3.3)

Due to the problems described above it was not included in the compared tables.

BTS Algorithm

SYM	symbol field
ATR	attribute field
LEFT	attribute field
RIGHT	right pointer
HEAD	head node pointer

```
BTS1.- IP ← LEFT(HEAD);
BST2.- If(SYM(IP) = (SYMSCH) ⇒'answer' ← ATR(IP),END.
       If(SYM(IP) <  SYMSCH) ⇒ go to BST4.
BST3.- |Search in the right subtree of the node pointed
        by IP|
       IPP ← RIGHT(IP);
       If(IPP ≥ 1) ⇒ IP ← IPP  and go to BST2.
       |There is no right sub tree|.
       IX  ⇐ FREE;
       SYM(IX) ← SYMSCH;
       ATR(IX) ← ATRINC;
       RIGHT(IP) ← IX,END;
```

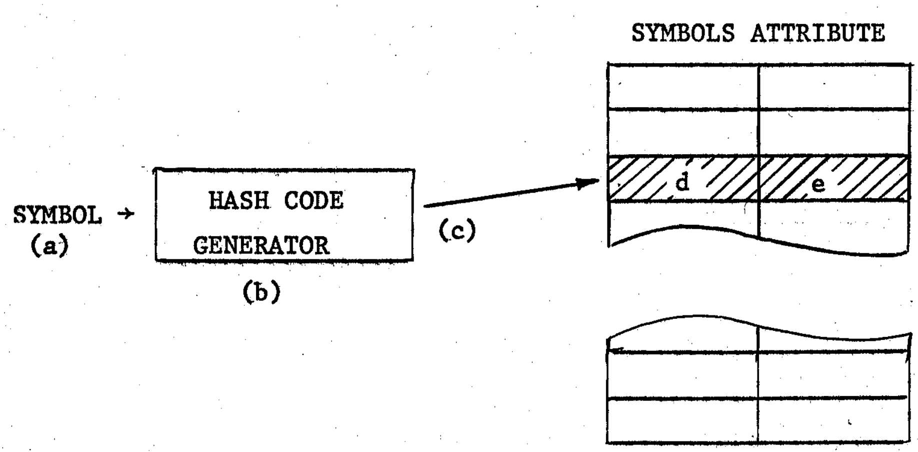
BST4. - |Search in the left sub tree of the node
node pointed by IP|

```
IPP ← LEFT(IP);
If(IPP ≥ 1) ⇒ IP ← IPP and go to BST2.
|There is no left sutree|
IX ⇆ FREE;
SYM(IX) ← SYMSCH;
ATR(IX) ← ATRINC;
LEFT(IP) ← IX,END;
```

Note: Housekeeping subroutines for FREE space indicate the occurrence of overflows.

1.4 - HASH CODE TABLES

Hash code (H.C.) methods of search can be visualized in the following way:



GENERAL SCHEME OF H.C. TABLES.
FIGURE 2

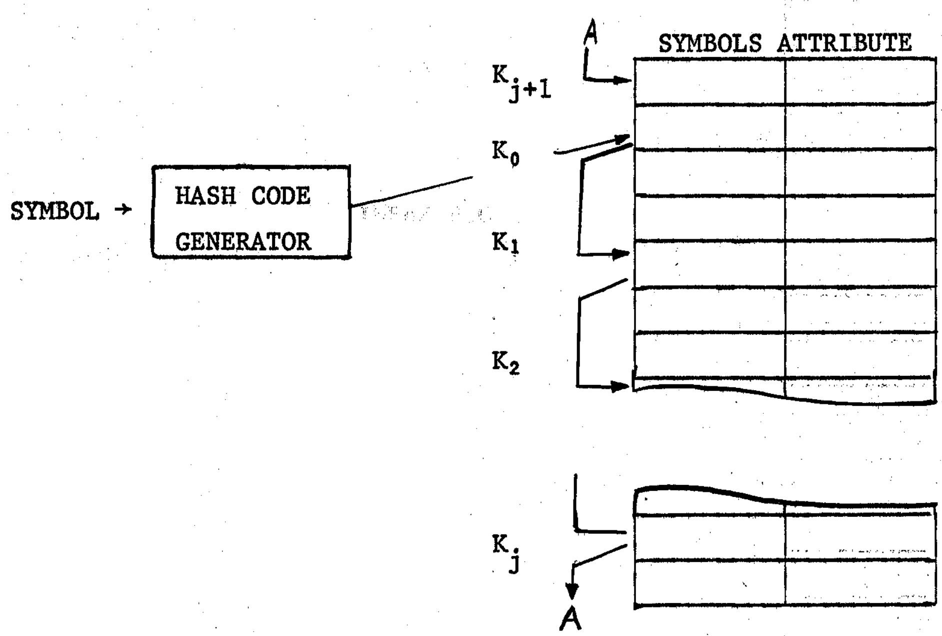
Beginning with the problem symbol (seacherd symbol) (a) a code is generated (c) through the code generator (b). This code would indicate in the symbol vector the location of the problem symbol (d). The corresponding attribute (e) is the system's output. (Problems with code generation are studied in chapter 2).

In the previous paragraph the statement "would indicate... the symbol vector" was resed. Let us see why this tense was used.

The H.C. generator produces values from 0 to N-1 (N is the table size), creating in this way the same code for different symbols; when this happens a collision occurs.

We now deal with ways to solve the collision problem.

1.4.1 - LINEAR SEARCH



SCHEME OF LINEAR H.C.SEARCH

FIGURE 3

Figure 3 shows the problem of finding whether location K_i given the formula

$$K_i = \mathrm{mod}(K_0 + i \;.\; a, N)$$

contains the symbol or it is empty.

K_0 = "code" (SYMSCH)

a = displacement constant
a and N must have a
m.c.d (a,N) = 1

i = number of collisions
$0 \leq i \leq N-1$

Since

$0<K_0<N-1$ and
$0<K_i<N-1$

location K_i+1 is searched.

The search is terminated if the symbol or an empty location is found, or if $i \geq N$.

Note that actually this technique is a modified linear search, (applicable to all H.C. techniques), since it makes a linear search among symbols which generate the same code or else symbol which generate displaced codes in 'i.a' location from the original code.

This technique allows for the danger of grouping, that is, a number of symbols generate either equal codes or codes displaced in a locations.

LHC Algorithm.

SYM	symbol vector
ATR	attribute vector
SYMSCH	searched symbol
ATRINC	included attribute
N	length of the table
A	displacement

```
LHC1. - K ← 'code' (SYMSCH); I ← 0;
LHC2. - If(SYM(K+1) = SYMSCH) ⇒ go to LHC5.
        If(SYM(K+1) =  ∅) ⇒ go to LHC4.
LHC3. -              |Collision|
        I ← I+1; If (I>N) ⇒'overflow', END.
        |Compute new location|
        K ← 'mod'(K + A,N) go to LHC2;
LHC4. - |Insertion|
        SYM(K+1) ← SYMSCH;
        ATR(K+1) ← ATRINC, END;
LHC5. - 'answer' ← ATR(K+1), END;
```

1.4.2 - QUADRATIC SEARCH

Refering to figure 3 this search has a displacement rule given by the equation

$$K_i = mod(K_0 + ixa + i^2xb, N)$$

$$K_0 = 'code' \ (SYMSCH)$$

a and b are displacement constants both, a and b, must be prime numbers. It has been determined' that the generation of values is cyclic, having a simmetry point determined by a. In order for ' this time lapse to be maximum it has been deter - mined that a=0 (see 1.4.2.1)

i = number of collisions

N = table size which must be a prime number.

The such is done in the location $K_i + 1$ due to the fact that' $0 \leq K_0, K_i \leq N-1$.

If the searched symbol or an empty place are found or if N/2+1 elements in the table have been searched, the searching is terminated.

Considering this technique as a special case of modified linear search, it will be noted that it presents a better solution than the linear H.C. technique for the grouping problem since the displacements in the search are not constants, preventing it from going into other code areas.

Figure 4 below shows a better way to visualize this problem, either in linear H.C. or in quadratic form.

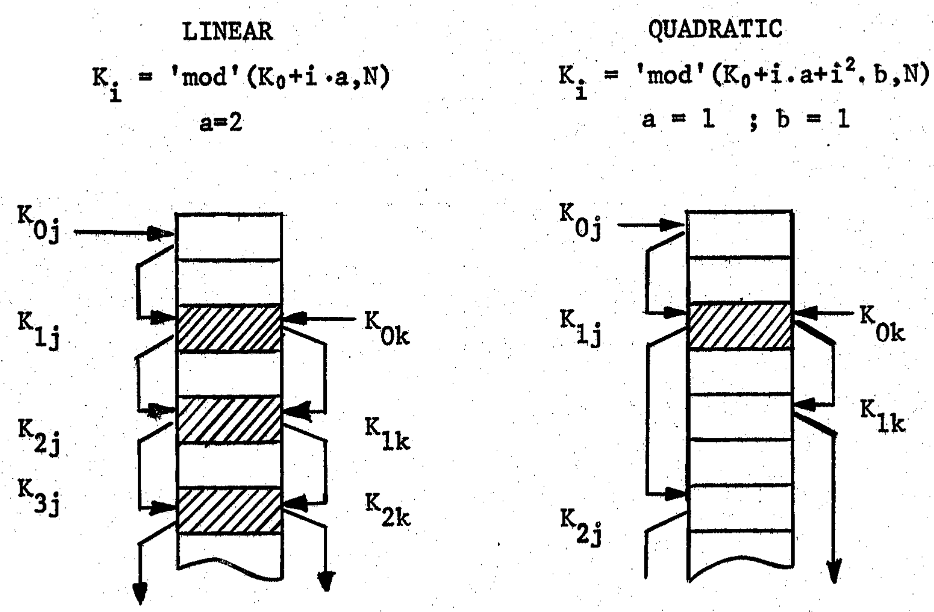


FIGURE 4

Let us see the following generation before we present the algorithm.

	J	K	
	$J \leftarrow A$	$K \leftarrow K_0$	initial values
	$J \leftarrow J+B$	$K \leftarrow K+J$	generation rules
1	A+B	K_0+A+B	
2	A+2B	K_0+2A+3B	
3	A+3B	K_0+3A+6B	
4	A+4B	K_0+4A+10B	
i	A+iB	$K_0+iA+(\frac{1+i}{2} \cdot i)B$	

the value K in the i iteration can be modified to

$$K = K_0 + i(A + B/_2) + i^2(B/_2)$$

for $a = A + B/_2$

$b = B/_2$

QSPS Algorithm

For

SYM	symbol vector
ATR	attribute vector
SYMSCH	searched symbol
ATRINC	included attribute
N	length of the table
NMAX	= N/2 + 1
A	= a-b
B	= 2b

```
QSPS1.- K ←'code'(SYMSCH); I ←0; J ← A;
QSPS2.- |Comparison|
        If(SYM(K+1) = SYMSCH) ⇒ go to QSPS5.
        If(SYM(K+1) = Ø) ⇒ go to QSPS4.
QSPS3.- |Collision|
        I ← I+1; If(I > NMAX) ⇒'overflow', END.
        J ← J+B;
QSPS4.- |Insertion|
        SYM(K+1) ← SYMSCH;
        ATR(K+1) ← ATRINC, END;
QSPS5 - 'answer' ← ATR(K+1), END;
```

1.4.2.1 - QUADRATIC SEARCH WITH TOTAL COMMING.

It has been proved that the coefficients of the searching equation must be a=0 and b-1 to insure total scanning on the table, changing that equation into

$$K_i = \mathrm{mod}(K_0 + i^2 ,N)$$

due to the fact that $0 \leq K_0, K_i \leq N-1$ must search in location ' $K_i + 1$.

The size of table a, must be a prime numbe of the form $4k + 3$, for k an integer.

QSTS Algorithm

For

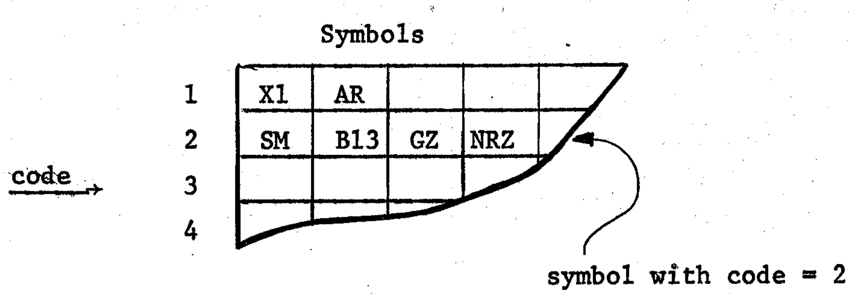
SYM	vector symbol
ATR	attribute symbol
SYMSCH	shearched symbol
ATRINC	included attribute
N	lenght of the table

```
QSTS1.- K ← 'code' (SYMSCH); I ← -N;
QSTS2.- |Comparision|
        If(SYM(K+1) = SYMSCH) ⇒ go to QSTS5.
        If(SYM(K+1) = Ø ) ⇒ go to QSTS4.
QSTS3.- |Collision|
        I ← I+2; If(I>N) ⇒ 'overflow', END.
        |Computation of new K|
        K ← 'mod' (K + |I|,N)  and go to QSTS2;
```

```
QSTS4.- |Insertion |
        SYM(K+1) ← SYMSCH;
        ATR(K+1) ← ATRINC,END;
QSTS5.- 'answer' ← ATR(K+1), END;
```

1.4.3 - OVERFLOWS TABLE SEARCH

Suppose for a moment, that we have a computer with unlimited memory so that we could have a matrix on which we could store, in every row, symbols whose codes are the row orders. (figure 5)

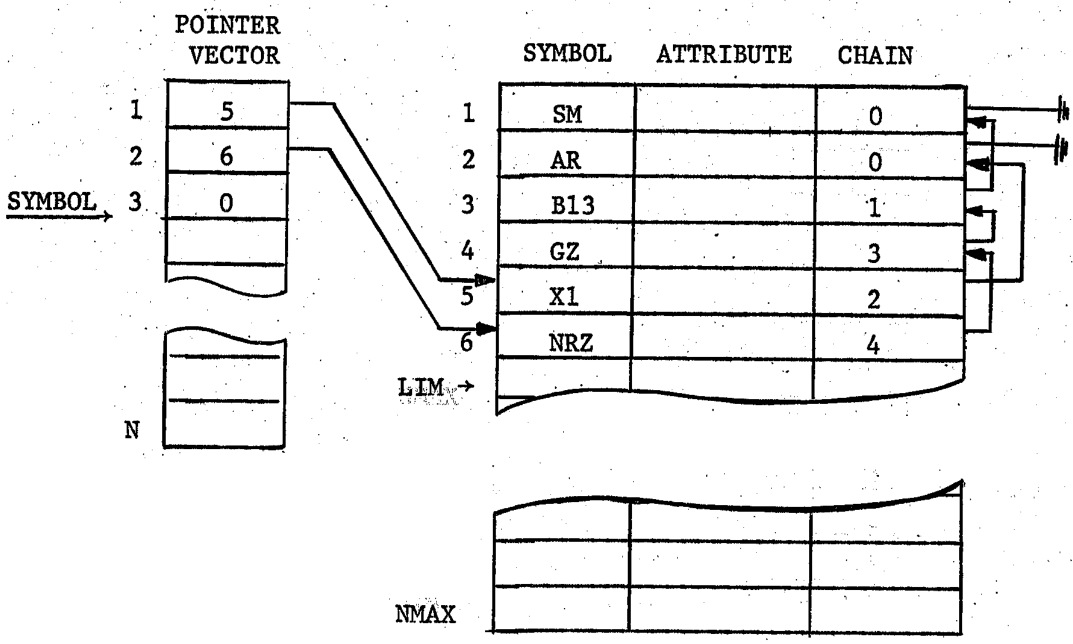


BASIC SCHEME FOR OVERFLOW TABLES
FIGURE 5

If this is so, all that is needed is to generate a code from a symbol and consult in a linear search every row whose order is its code.

Actually this matrix would not work since the storing structure is modified nevertheless in accordance to the approach of chaining symbols of same code.

The new storing structure would be: (figure 6)



SCHEME FOR OVERFLOW TABLE TECHNIQUE
FIGURE 6.

As it can be seen, the chaining of symbols with equal code is done through a CHAIN field, generating a list. The first node of the list is pointed to by the element of a pointer vector, its order being that of the symbol code of the list. We shall note that this technique uses a linear search within every list. It offers therefore substantial advantages over linear and quadratic H.C. techniques, since there is no interference of symbol with different codes.

LIM is a pointer of the first available space in the list forming nodes.

OVFT Algorithm

For

SYM	symbol vector
ATR	attribute vector
CHAIN	chain vector
PNTR	pointer vector
SYMSCH	searched symbol
ATRINC	included attribute
LIM	pointer to the first empty place in the table
NMAX	length of the table
N	length of the PNTR vector

OVFT1.- KOD ← 'code'(SYMSCH); K ← PNTR (KOD + 1);

OVFT2.- If(K = ∅) ⇒ go to OVFT5.

If(SYM(K) = SYMSCH) ⇒ go to OVFT4.

OVFT3.- |a new element of the list is searched|

K ← CHAIN(K) and go to OVFT2;

```
OVFT4.- |SYM(K) is the searched symbol|
        'answer' ← ATR(K), END;
OVFT5.- |There is no list or the searched symbol is not
         included|  |Insertion|
        If(LIM>NMAX) ⇒ 'overflow', END.
        SYM(LIM) ← SYMSCH;
        ATR(LIM) ← ATRINC;
        CHAIN(LIM) ← PNTR(KOD + 1);
        PNTR  (KOD+1) ← LIM;
        LIM ← LIM+1, END;
```

* It must be noted that the length of the pointer vector is not necessarily equal to the maximun length of the table.

1.5 - APPLICATIONS TO VARIABLE LENGTH SYMBOLS.

For all techniques described in this section we will have a set of fields called block. Each block contains.

NO. of fields	Name	Storing
a) 1	Length of the symbol(LGTH) NO.of fields	SYM(I)
b) LGTH	Symbol	SYM(I+1),...,SYM(I+LGTH)
c) 1	Associeted attri bute	SYM(I+LGTH+1)

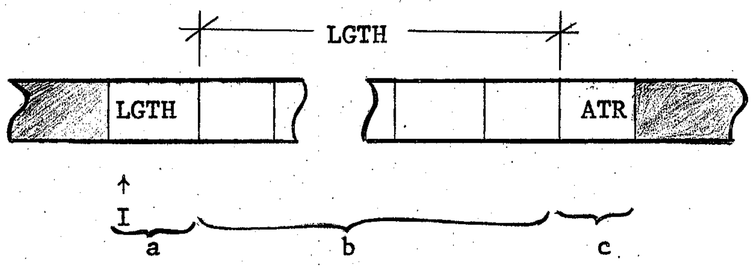
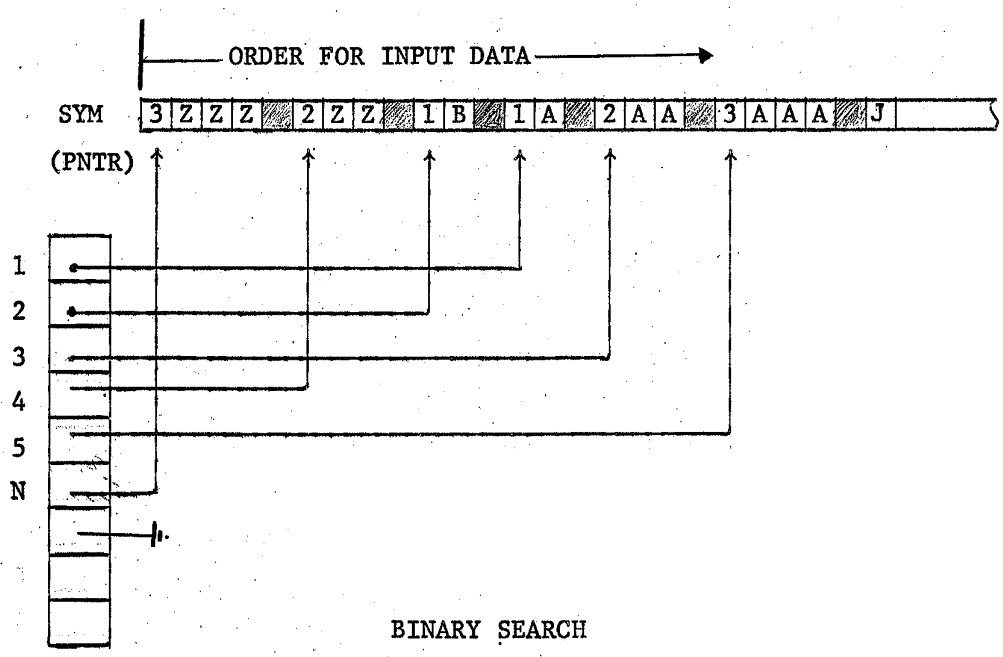


FIGURE 7

BASIC BLOK SCHEME FOR STORING
VARIABLE LENGTH SYMBOLS.

1.5.1 - BINARY SEARCH

It is included in this technique a pointer vector to the beginning of each block, indicating the displacement in this vector. Figure 8 shows the state of the table with six symbols.



BINARY SEARCH
VARIABLE LENGTH SYMBOL
FIGURE 8

BSV Algorithm

For

SYM	storage vector
PNTR	pointer vector
SYMSCH	searched symbol
LGTH	symbol length
ATRINC	included attribute
N	actual symbol in the table
NMAX	maximum symbol in the table
J	pointer to the first empty place in the SYM vector.
JMAX	SYM length

```
BSV1.-  LUPP ← N+1; LLOW ← 1;
        If(N<1) ⇒ I ← 1  and go to BSV9.
BSV2.-  |Compute the place of the search|
        I  ← (LUPP+LLOW)/2 ;
BSV3.-  II ← PNTR(I);
        |Length comparison|
        If(SYM(II) > LGTH) ⇒ go to BSV4.
        If(SYM(II) < LGTH) ⇒ go to BSV5.
        |Equal length|
        ∀K((K=1,...,LGTH)
        If(SYM(II+K) > SYMSCH(K)) ⇒ go to BSV4.
        If(SYM(II+K) < SYMSCH(K)) ⇒ go to BSV5.);
        'answer' ←  SYM(II+LGTH+1), END;
```

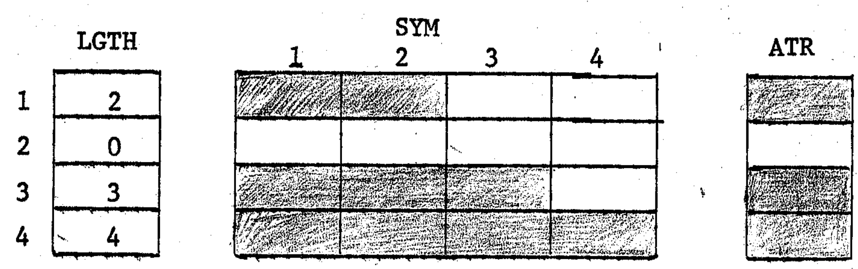
```
BSV4.- |Compute a new upper limit|
       If (LUPP = I) ⇒ go to BSV6.
       LUPP ← I go to BSV2;
BSV5.- |Compute a new lower limit|
       If(LLOW = I) ⇒ I ← I+1 and go to BSV6.
          LLOW ← I  and go to BSV2;


BSV6.- |Error conditions|
       If(N+1 > NMAX) ØR(J + LGTH + 1 > JMAX) ⇒
       'overflow', END.
BSV7.- If(N = LINF) ⇒ go to BSV9.
BSV8.- |Displacement|
       ∀K((K = N,N-1,...,I)
       PNTR(K+1) ← PNTR(K););
BSV9.- |Insertion|
       N ← N+1; PNTR(I) ← J;
       SYM(J) ← LGTH;
       ∀K((K = 1,2,...,LGTH)
            SYM(J+K) ← SYMSCH(K););
       SYM(J+LGTH+1) ← ATRINC;
       J ← J+TAM+2, END;
```

1.5.2 - HASH CODE TECHNIQUES

1.5.2.1 - LINEAR SEARCH

There are two alternatives for both this technique and quadratic search

a) To consider the maximum length of working symbols as the number of columns in a storing matrix, adding a size vector we will have. figure 9.



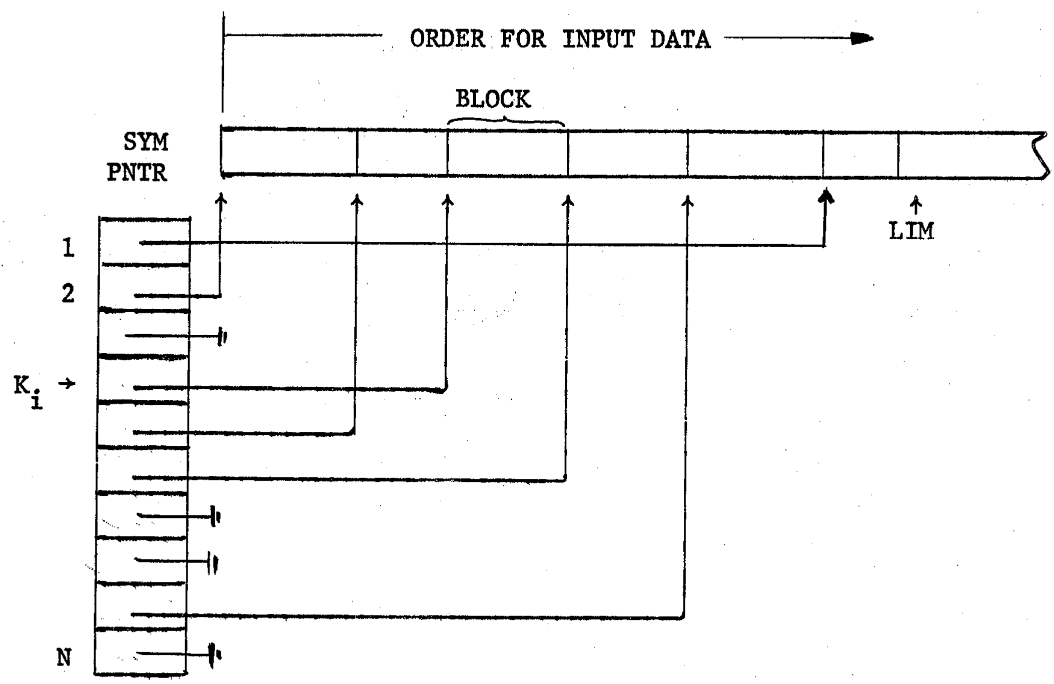
SCHEME FOR MATRIX STORING STRUCTURE
FIGURE 9

This scheme implies that there is too much use of memory space (if there is a small number of symbols with maximum lenght fields) but with less processing time.

For this alternative either algorithm similar to that described in 1.4.1 or 1.4.2 for quadratic search is applied.

b) The second alternative diminishes the necessary storing space, increasing processing time.

The structure will have a pointer vector to the storing block (figure 10). The searching of K_i is now done through the use of the pointer vector.



H.C. TABLE FOR LINEAR AND QUADRATIC SEARCH
VARIABLE LENGHT SYMBOL

FIGURE 10

LSV Algorithm

For

SYM	storage vector
PNTR	pointer vector
SYMSCH	searched symbol
LGTH	symbol lenght
ATRINC	included attribute
N	PNTR lenght
LIM	pointer to the first empty place in SYM
LMAX	SYM length
A	displacement

```
LSV1. -   KI ← 'code'(SYMSCH); I ← 0;
LSV2. -   K  ← PNTR (KI+1);
          If(K = ∅) ⇒ go to LSV5.
          If(SYM(K) = LGTH) ⇒ go to LSV4.
LSV3. -   |Collision|
          I ← I+1; If(I>N) ⇒'overflow', END.
          KI ← 'mod'(KI+A,N) and go to LSV2.
LSV4. -   |Equal lenght|
          ∀L((L = 1,2,...,LGTH)
          If(SYM(K+L) ≠ SYMSCH(L) ⇒go to LSV3.);
         'Answer'← SYM(K+LGTH+1),END;
LSV5. -   |Insertion| If(LIM+LGTH+1>LMAX) ⇒ 'overflow',END.
          PNTR(KI+1) ← LIM;SYM(LIM) ← TAM;
          ∀L((L = 1,2,...,LGTH)
               SYM(LIM+L) ← SYMSCH(L);) ;
               SYM(LIM+LGTH+1) ← ATRINC;
               LIM ← LIM+LGTH+2, END;
```

1.5.2.2 - QUADRATIC SEARCH

The algorithm for alternative b as described in 1.5.1 is now presented.

Using the same introduction for the algorithm QSPS (1.4.2) we have.

QSPV Algorithm

For

SYM	storage vector
PNTR	pointer vector
SYMSCH	searched symbol
LGTH	symbol lenght
ATRINC	included attribute
N	PNTR lenght
NMAX	N/2 + 1
LIM	pointer to the first empty place in SYM
LMAX	SYM lenght
A	a-b
B	2b

```
QSPV1.- KI ← 'code'(SYMSCH);  I ← 0; J ← A;
QSPV2.- K ← PNTR(KI+1);
        If(K = Ø) => go to QSPV5.
        If(SYM(K) = LGTH) => go to QSPV4.
```

```
QSPV3.- |Collision|
        If ← I+1 ;
        If(I>NMAX) ⇒ 'overflow', END.
        J ← J+B ;
        KI ← 'mod'(KI+J,N) and go to QSPV2;
QSPV4.- |Equal length|
        ∀L((L = 1,2,...,LGTH)
        If(SYM(K+L) ≠ SYMSCH(L)) ⇒ go to QSPV3.);
        'answer' ← SYM(K+LGTH+1), END;
QSPV5.- |Insertion|
        If(LIM+LGTH+1 > LMAX) ⇒ 'overflow', END.
        PNTR(KI+1) ← LIM;SYM(LIM) ← LGTH;
        ∀L((L = 1,2,...,LGTH)
            SYM(LIM+L) ← SYMSCH(L););
            SYM(LIM+LGTH+1) ← ATRINC;
            LIM ← LIM+LGTH+2, END;
```

1.5.2.2.1 - <u>QUADRATIC SEARCH, TOTAL SCANNING</u>

Algorithm PCBT will be applied for total scanning in the table, extending it to variable length symbols.

QSTV Algorithm

For

SYM	storage vector
PNTR	pointer vector
SYMSCH	searched symbol
ATRINC	included symbol
LGTH	symbol lenght

```
          N      PNTR length
          LIM    pointer to the first
                 empty place in SYM
          LMAX   SYM length

QSTV1. -  KI ← 'code'(SYMSCH); I ← -N;
QSTV2. -  K ← PNTR(KI+1);
          If(K = Ø) ⇒ go to QSTV5.
          If(SYM(K) = LGTH) ⇒ go to QSTV4.
QSTV3. -  |Collision|
          I ← I+2; If(I ≥ N) ⇒ 'overflow', END.
          KI ← 'mod' (KI+ |I|,N) and go to QSTV2;
QSTV4.-   |Equal length|
          ∀L((L = 1,2,...,LGTH)
          If(SYM(K+L) ≠ SYMSCH(L)) ⇒go to QSTV3);
          'answer' ←  SYM(K+LGTH+1), END;
QSTV5.-   |Insertion|
          If(LIM+LGTH+1>LMAX) ⇒ 'overflow', END.
          PNTR(KI+1) ← LIM;SYM(LIM) ← LGTH;
          ∀L((L = 1,2,3,....,LGTH)
               SYM(LIM+L) ← SYMSCH(L););
          SYM(LIM+LGTH+1) ←  ATRINC;
          LIM ← LIM+LGTH+2; END;
```

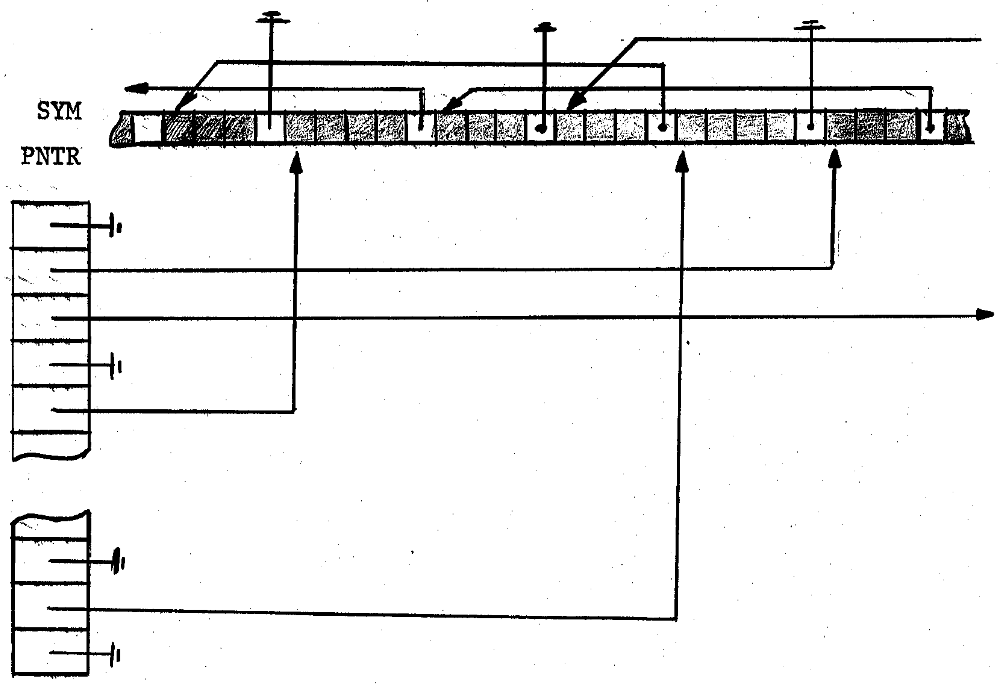
1.5.2.3 - OVERFLOW TABLE SEARCH.

For this technique the block described in 1.5 will be used adding a field to this block. This field is reserved for the chaining pointer.

Each block will contain

No. of fields		Name	Storing
a)	1	symbol length	SYM
b)	LGTH	(LGTH)	SYM(I+1),...,SYM(I+LGTH)
c)	1	Symbol	SYM(I+LGTH+1)
d)	1	Chain	SYM(I+LGTH+2)

This scheme is basicaly the same as that of constant length symbols, differing in the storing of the symbols(figure 11)



H.C. OVERFLOW TABLE
VARIABLE LENGTH SYMBOL
FIGURE 11

Algorithm OTSV

For

SYM	storage vector
PNTR	pointer vector
SYMSCH	searched symbol
LGTH	symbol lenght

ATRINC	included attribute
LIM	pointer to the first empty place in SYM
LMAX	lenght of SYM
N	lenght of PNTR

```
OSTV1.- KOD ← 'code'(SYMSCH);
        K ← PNTR(KOD+1);
OSTV2.- If(K = Ø) ⇒ go to OTSV5.
        If(SYM(K) = LGTH) ⇒ go to OTSV4.
OTSV3.- |next element in the list|
        LL ← SYM(K);
        K ← SYM(K+LL+2) and go to OTSV2;
OTSV4.- |Equal length, comparison|
        ∀L((L = 1,2,...,LGTH)
        If(SYM(K+L) ≠ SYMSCH(L)) ⇒ go to OTSV3.);
        'answer' ← SYM(K+LGTH+1), END;
OTSV5.- |Insertion|
        If(LIM+LGTH+2>LMAX) ⇒'overflow', END.
        SYM(LIM) ← LGTH;
        ∀L((L = 1,2,...,LGTH)
             SYM(LIM+L) ← SYMSCH(L););
        SYM(LIM+LGTH+1) ← ATRINC;
        |Chaining|
        SYM(LIM+LGTH+2) ← PNTR(KOD+1);
        PNTR(KOD+1) ← LIM;
        LIM ← LIM+LGTH+3, END;
```

2. - HASH CODE GENERATORS.

In this chapter hash code generators will be defined and which variable, should be applied to it for its execution. Finally a comparison between some generators will be presented.

2.1 - INTRODUCTION

A H.C. generator is function between a symbol and a value.

$$f(\text{symbol}) = \text{value}.$$

Let us study the following function as an introduction to the problem.

Each letter of the symbol will take the value of its order position in the alphabet, such that if f(ABCD) is used, we will have 01020304 as the value formed by the corresponding order of the symbol letters.

Therefore we will have a generation of values for 1 to 4 character symbols

$$f(X)_{min} = 01 \qquad \text{for } X = A$$

$$f(X)_{max} = 26262626 \quad \text{for } X = ZZZZ$$

These limits indicate that, having $\sum_{n=1}^{4} 26^n$ symbol, the generating function yields approximatly 18% of these in-between values. These values will be continuous between 1-26, 101-126, 201-226 etc.

What actually happens is:

a) The used symbols are usually a small subset of all possible symbols.

b) The characters have a different inner code from the orderes numbers.

Nevertheless many computers have a discontinued series.

Example for IBM 7044

LETTER	A	B ...	I	J	K ...	R	b	S	Z
Octal code	21	22	31	41	42	51	60	62	71
Decimal code	17	18	25	33	34	41	48	50	57

If this code is applied to the previous example a 9% reduction in the possible generator values would be fielded, since the upper limit increases

$$f(X)_{max} = 57575757$$

besides this a greater discontinuitly of generated values is produced.

The inclusion of numeric characters in the symbols sligltly increases the utilization factor, (number of possible generated values/ number of in between values) also reducing the discontinuity gaps.

For computers with BCD codes, six bit per character, such as the IBM 7044 there are different symbols for which equal values will be fielded. This is due to the fact that the symbol code includes the sign bit. Due to the use of positive values the absolute value of the function is to be taken.

Example

f(BCDEFG) = |22 23 24 25 26 27| Octal
f(SCDEFG) = |-22 23 24 25 26 27| value

We could say that the problems regarding a hash code generator are:

- The reduction of the relationship "number of inbetween values / number of possibly generated values."

- The reduction of discontinuity.

- The generation of unique values for every symbol or at least to avoid that a great number of symbols generate only few of the possible values.

- To generate these values within a compatible time.

2.2 - IMPORTANCE OF THE TABLE LENGTH.

The increase of the utilization factor can either lead to solutions or an escalation of the discontinuity problems and/or generation of ambignous values. The solution uses:

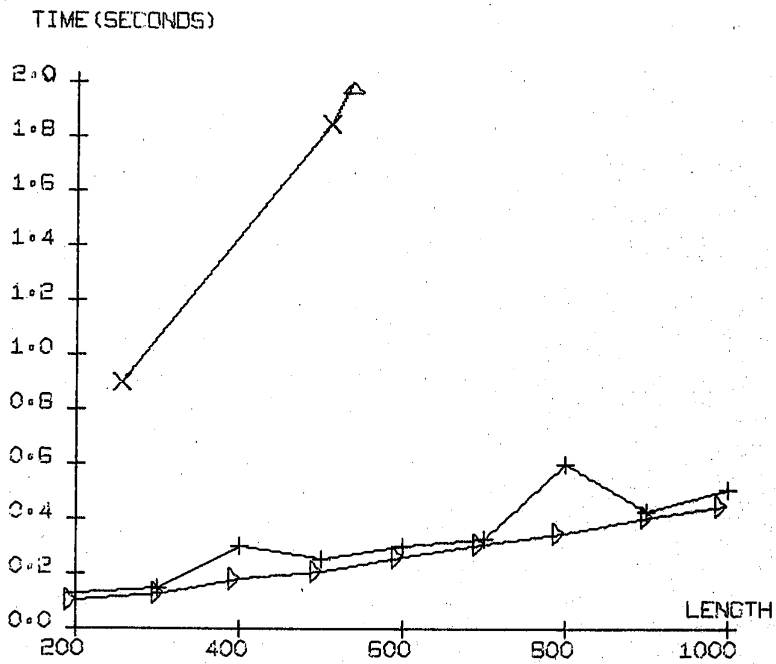
value = 'mod'(f(symbol),N)

where 'mod' is the remainder of the division f(symbol)/N. Variable N will sometimes stand for the length of the table and in other cases, the length of an auxiliar pointer vector.

Three kinds of values of N were used in this experiment.

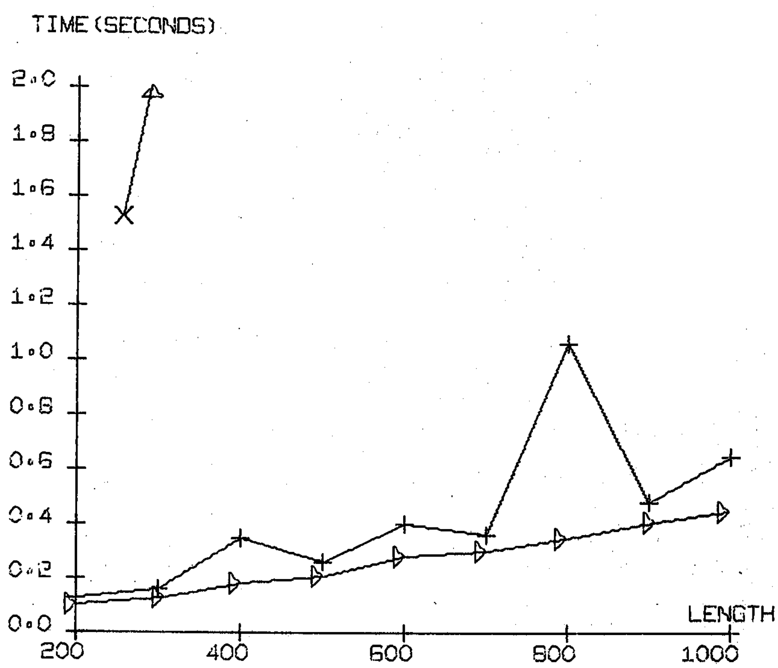
1. - $m.10^2$ $m = 2,3,\ldots,10$ 'factor of 100'
2. - 2^n $n = 8,9,10$ ' integer powers of 2'
3. prime numbers

Experience has shown that the use of prime numbers for N generates better solutions than those for 1 and 2 above.



COMPARISON BETWEEN DIFFERENT TYPES OF LENGTHS
+ =FACTOR OF 100 ,X=INTEGER POWER OF 2 , ▷ =PRIME NUMBER
BOUND SYMBOLS

FIGURE 12



COMPARISON BETWEEN DIFFERENT TYPES OF LENGTHS
+ =FACTOR OF 100 ,X=INTEGER POWER OF 2 , ▷ =PRIME NUMBER
RANDOM SYMBOLS

FIGURE 13

Figures 12 and 13 show these facts, i.e., the choice of the three different types of lengths. The time and lenght coordinates show in the figures a comparison between construction and searching time of a table (in this case h.c. overflow tables).

This comparison is made for two types of symbols: RANDOM and BOUND.

These symbols will be defined in 3.1

2.3 - CONTINUITY OF VALUES.

When trying to solve the discontinuity problem of generated values different technique appear such as:

2.3.1 - EXCLUSIVE OR (XOR).

This method is applied for generating h. c. symbols for more than one computer word. An XOR operation between the symbol words obtains their absolute value and finds mod(N).

2.3.2 - TABLE LENGHT DIVISION (D/LENGTH)

This method considers that the solution to the lenght problem is enough to generate a good value. In order to process symbols with more than one word in computers that don't have XOR operation a logic sum of words can be used instead.

This sum must be done regarding all values as positive numbers and, if possible, a 1 must be added to the lower order bit for every overflow (in order to distinguish it from values that do not cause overflow).

Next, mod(N) is taken from the absolute value of this sum.

2.3.3 - WEIGHTED SUM (WGHT S.)

This method attempts to solve both the problem of factor utilisation and the discontinuity problem.

It is basically done giving a weight for every one of the symbol characters and making a weighted sum with the assigned weights.

The weighting function will assign a greater weight to the first character diminishing its weight for the suceeding characters.

2.4 - GENERATOR COMPARISON

Let us now compare D/LENGTH and S.POND for a RAND symbol in an IBM 7044 word. (see TABLE 1)

This comparison shows the number of collisions for every method for different lengths and for the generator types D/LENGTH and WGHT S.

The collisions are shown in table length percentages,the percentage of non generated values given in column 0, the percen - tage of generated value for one symbol given in column one, the percentage of generated value for two symbols given in column 2 (collision for two symbols), and so on.

An upper limit of ten percent over the distributed values is established because this is considered a practical limit for hash code techniques.

Column TIME indicates the relative time (in seconds) for the generation of 1000 values.

			NUMBER OF SYMBOLS OF EQUAL GENERATED VALUES.					
H.C TYPE	LENGHT	TIME	0 (FREE)	1	2	3	≤ 10	>10
S.POND	1000*	0.23	53.5	20.2	22.0	24.6	100.	0.
S.POND	997***	0.23	53.5	20.2	22.3	24.3	100.	0.
S.POND	1021***	0.24	54.0	19.7	20.9	25.6	100.	0.
S.POND	1024**	0.23	54.1	19.6	21.1	25.2	100.	0.
S.POND	729	0.23	44.9	28.4	30.5	23.5	100.	0.
D/LENGHT	1000*	0.18	78.4	8.9	1.0	2.1	86.0	14.00
D/LENGTH	997***	0.20	38.8	35.4	31.9	20.7	100.0	0.
D/LENGTH	1021***	0.18	36.9	37.1	35.7	18.8	100.0	0.
D/LENGTH	1024**	0.18	88.87	9.4	1.8	0.6	11.8	88.2
D/LENGTH	729	0.23	36.9	37.0	33.4	23.7	94.5	5.5

H.C. Comparison

TABLE I

*	type	one	length	multiple of 100
**	"	two	"	power of 2
***	"	three	"	prime

We can see from table 1 that the behavior of the WGHTS. technique for any length is similar: the distributions are almost uniform for the different lengths concentrating in 65% of the collision symbols of up to 3 symbols. The D/LENGTH technique makes a clear distinction between the different types establishing prime number lengths the best ones. Note the good utilization factor of the generator shown in the few generated values (38% in column 0). The distribution for these prime lengths groups up to 80% of the collision values of up to 3 symbols.

As a complementation to Table 1 the collisions produced in D/LENGTH as far as distribution is concerned present the following characteristics,

- For type one length the distribution of values is almost uniform, presenting a constant sequence "reached values for one or more symbols - k unreached values".

- For type two length there is no uniform distribution for "reached values / unreached values". It tends to generate one value for a great number of symbols. For length = 1024, 692 symbols generated one value.

- For type three length the distribution is bound to be uniform, with collisions of up to five symbols.

Comparing WGTH S. and D/LENGTH time we have a ratio of 1:0.75. Regarding this comparison the IBM 7044 hardware favors WGTH S.; this time relation is expected to increase in other computers.

Based on the preceding comparison D/LENGTH for prime lengths can be considered to be better than WGTH S. The WGTH S. technique is uniform for dealing with any length type.

3 - RESULTS

3.1 - ORGANIZATION OF THE EXPERIMENT

The behavior of time (with a ± 0.02 seconds error) and storage space variables were measured for both construction and search.

- This measurement was done for prime number lengths of the table, in the h.c. techniques and for type 1 lengths (multiples of 100) for both binary and ternary searches.

- The h.c. generator used was D/LENGHT (2.3.2), since for prime lengths this generator is more efficient.

- For h.c. techniques, the behavior of different utilization percentages, were measured.

- For linear and quadratic searches, the utilization percentages refer to "the present number of symbols/possible number of symbols".

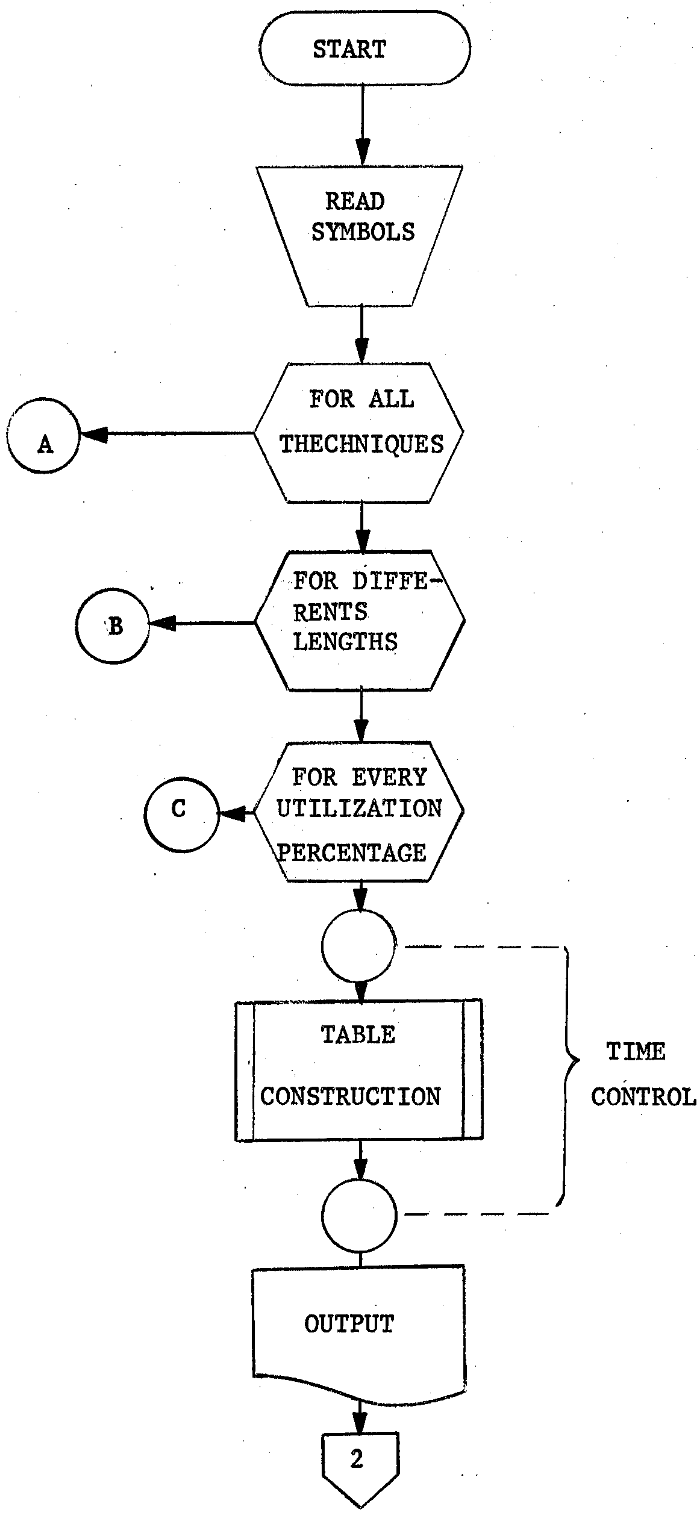
- For overflow tables it refers to the relation "total number of symbols in the table/length of the pointer vector".

- For the table comparison the symbols used were called RANDOM. The characteristics of the RANDOM symbols, are: they have $1 \leq n \leq 6$ characters, where n is a random number. The used characters are: a letter for the first one and letters or numbers for the remaining ones. The choice of the characters was random too.

- For the h.c. experiment a second type of symbols called "bound symbols (BOUND) were used. It was formed as a basis for a certain number of words on which characters or numbers were replaced.

- The used coefficients are: linear search, $K_i = K_0 + i.7$ quadratic search, partial scanning

$$K_i = K_0 + i.3 + i^2.5$$


START
READ SYMBOLS
FOR ALL THECHNIQUES
A
FOR DIFFE-RENTS LENGTHS
B
FOR EVERY UTILIZATION PERCENTAGE
C
TABLE CONSTRUCTION
TIME CONTROL
OUTPUT
2

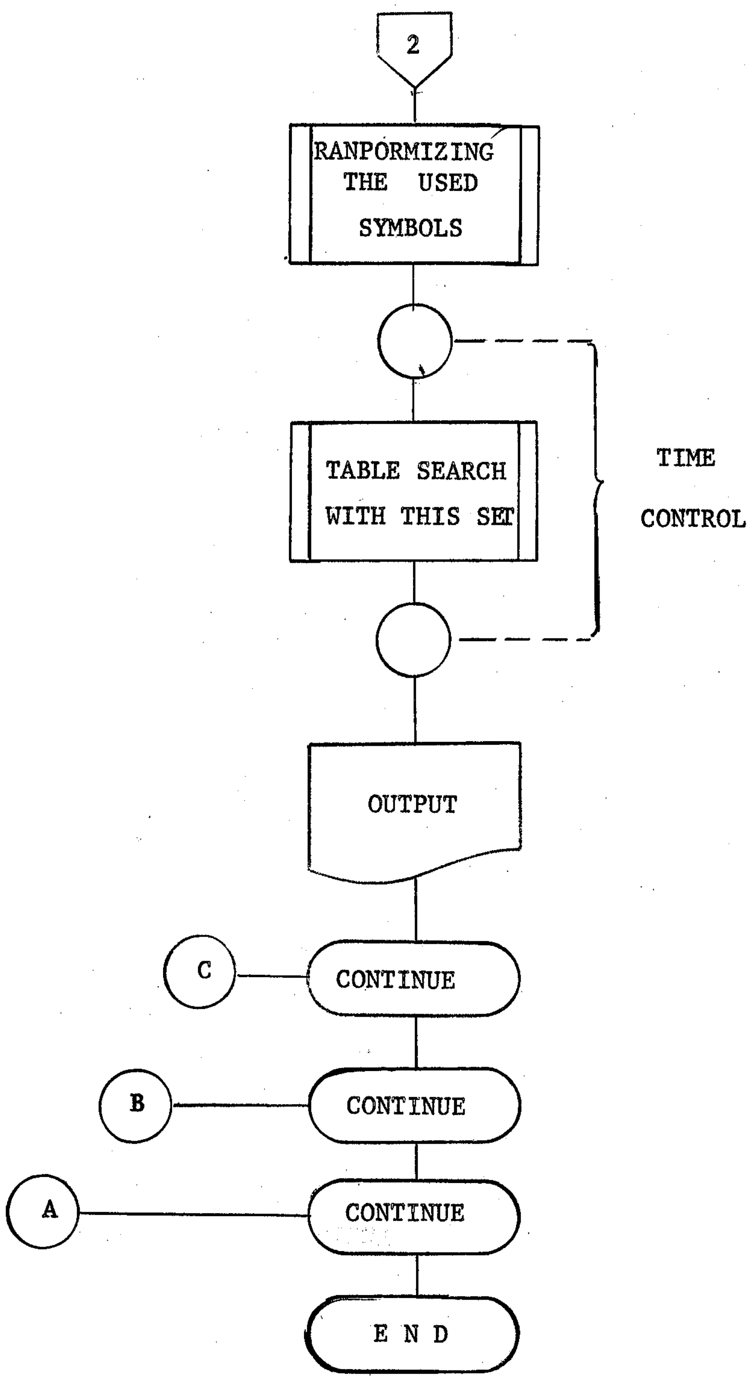

2
RANPORMIZING
THE USED
SYMBOLS
TABLE SEARCH
WITH THIS SET
TIME
CONTROL
OUTPUT
C
CONTINUE
B
CONTINUE
A
CONTINUE
E N D

FIGURE A

3.2 - COMPARISON BETWEEN BINARY AND TERNARY SEARCHES

Theoretically there was a chance that a ternary search would be much better then the binary one, due to the reduction of the number of comparisons $\log_3 N$ (for ternary) < $\log_2 N$ (for binary). Besides theoretically reducing the number of comparisons, ternary search must compute a new in between limit and also must increase the number of questions by one in order to select every new search area. The experiment showed however that both techniques yield similar results; therefore only binary search will be compared.

Different hardware characteristics may field different results in this comparison.

EXPLANATION OF THE DIAGRAM (Figure 14)

'FOR ALL TECHNIQUES' were used:

1 Binary search
2 Ternary search
3 Linear h.c. search
4 Quadratic h.c. search (partial scanning)
5 Quadratic h.c. search (total scanning)
6 H.C. overflow tables.

'FOR DIFFERENT LENGTHS'.

For techniques 1 and 2 the lengths below were used
200, 300, ... ,1000.
For techniques 3,4 and 6 were used
199,307,401,499,601,701,797,907 and 997.
For technique 5 were used
199,307,419,499,607,691,811,907 and 997.

'For EVERY UTILIZATION PERCENTAGES'

were used 50,60,70,80,90 and 100.

'TIME CONTROL'

was done using the system's clock with a 1/60 sec precision.

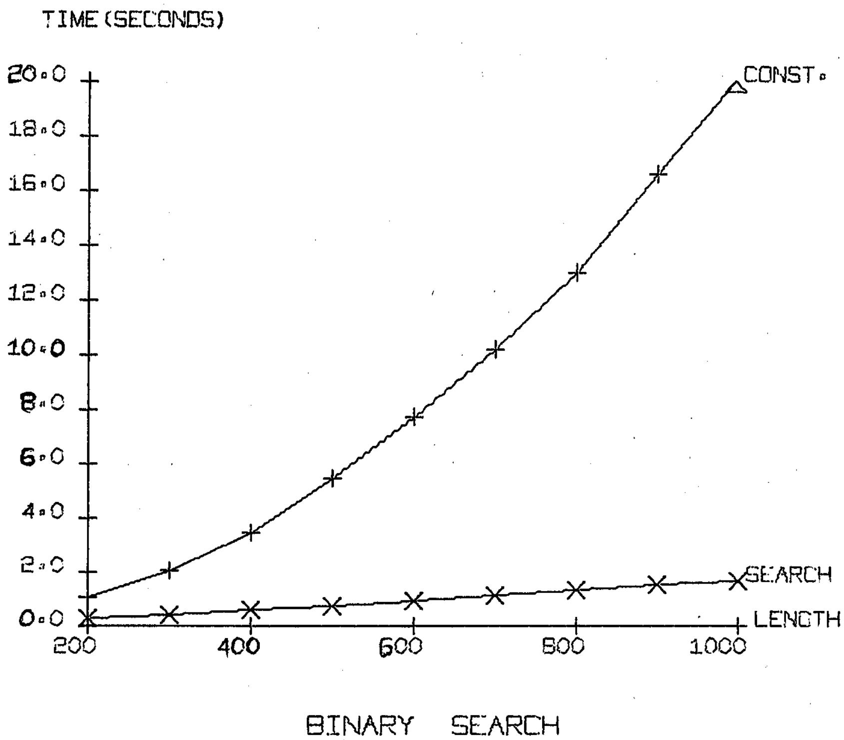

TIME (SECONDS)
20.0
18.0
16.0
14.0
12.0
10.0
8.0
6.0
4.0
2.0
0.0
200
400
600
800
1000
CONST.
SEARCH
LENGTH
BINARY SEARCH

FIGURE 15

3.3 - COMPARISON BETWEEN CONSTRUCTION AND SEARCH.

We shall define "fixed tables" as tables in which the component symbols are known.

This kind of table, after its construction, is used only for searches

We shall call "variable table" a table in which the component symbols are unknown. This is the reason why the construction and search methodes are simultaneous.

The reason for this comparison is the fact that the advantages in the application of fixed or variable tables to different techniques are best shown.

3.3.1 - BINARY SEARCH.

Based on the preceeding algorithm it could be expected that construction time would be greater than search time. The experiment effectively shows that construction time is really grater than search time (figure 15).

This characteristic shows that its use is convenient for fixed tables. Therefore its construction could be done at the beginning of the process with the ordered set of symbols , decreasing construction time.

The analytical equations that rule both processes are shown in 3.5.

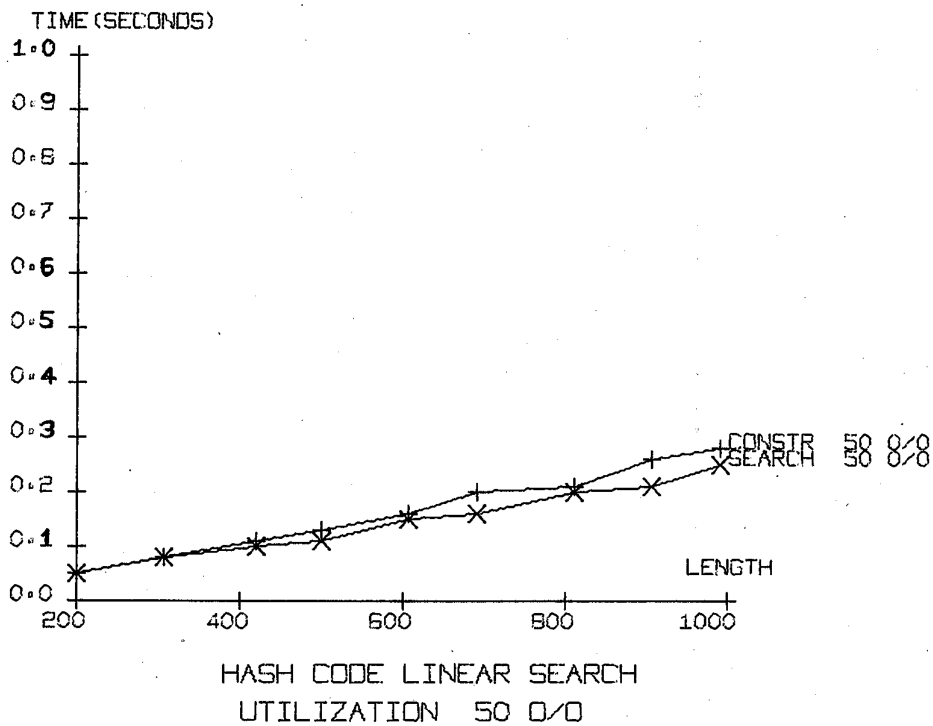
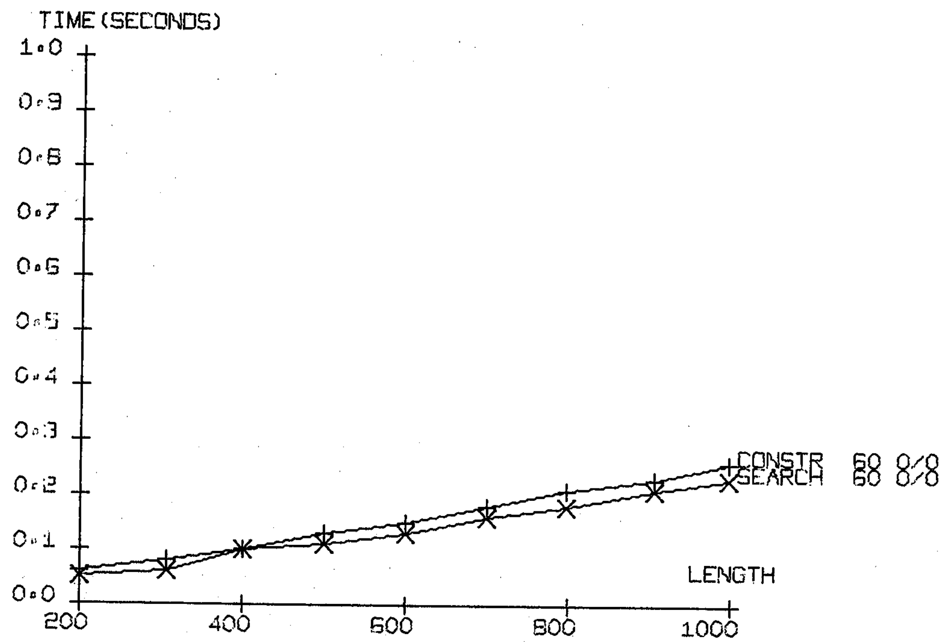

TIME (SECONDS)
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
CONSTR 50 O/O
SEARCH 50 O/O
LENGTH
200
400
600
800
1000
HASH CODE LINEAR SEARCH
UTILIZATION 50 O/O

FIGURE 16



HASH CODE SEARCH WITH OVERFLOW TABLES
UTILIZATION 60 O/O

FIGURE 17

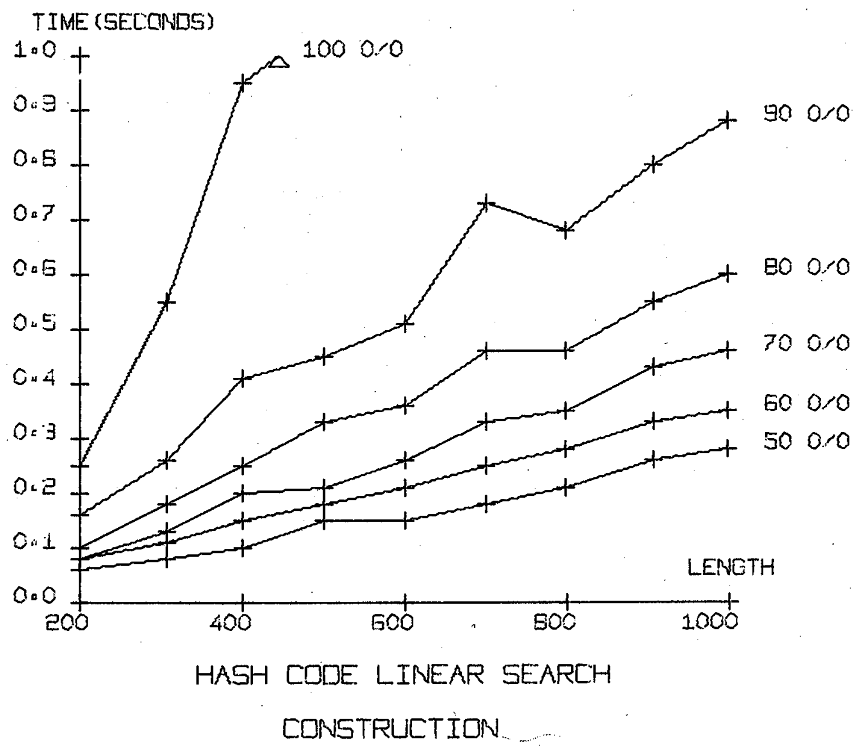
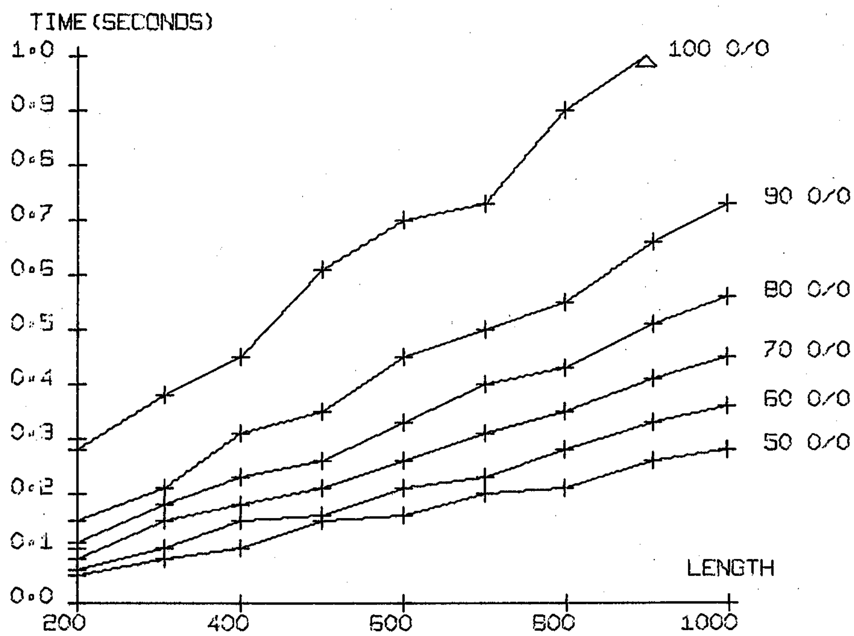

TIME (SECONDS)
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
100 0/0
90 0/0
80 0/0
70 0/0
60 0/0
50 0/0
LENGTH
200
400
600
800
1000
HASH CODE LINEAR SEARCH
CONSTRUCTION

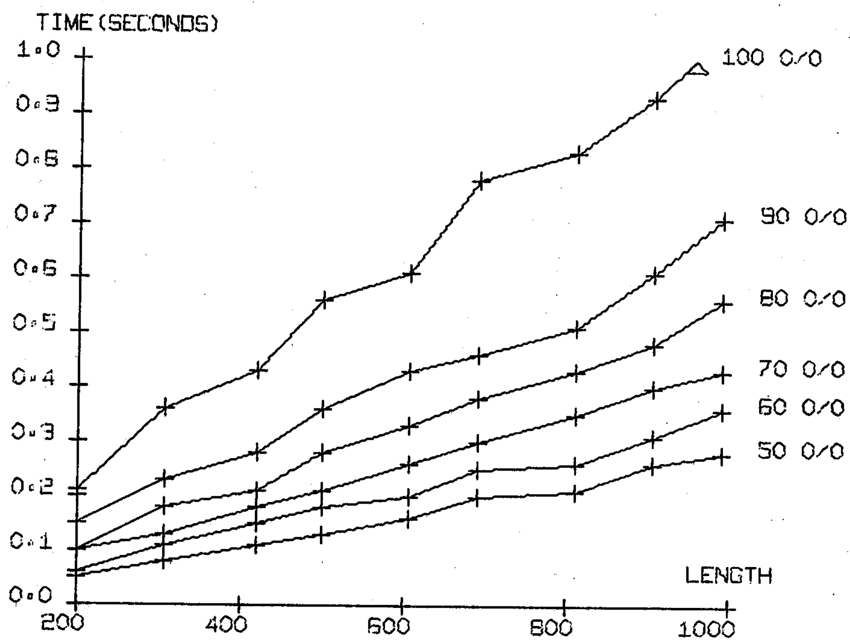
FIGURE 18



HASH CODE QUADRATIC SEARCH (PARTIAL SCANNING)

CONSTRUCTION

FIGURE 19



HASH CODE QUADRATIC SEARCH (TOTAL SCANNING)

CONSTRUCTION

FIGURE 20

3.3.2 - H.C. TABLES.

It was proved that search time is, for these techniques, slightly lesser than construction time (see as an example figures 16 and 17). This characteristic makes then fit to work both in fixed and variable length tables.

The equations that rule its construction are shown in 3.5.

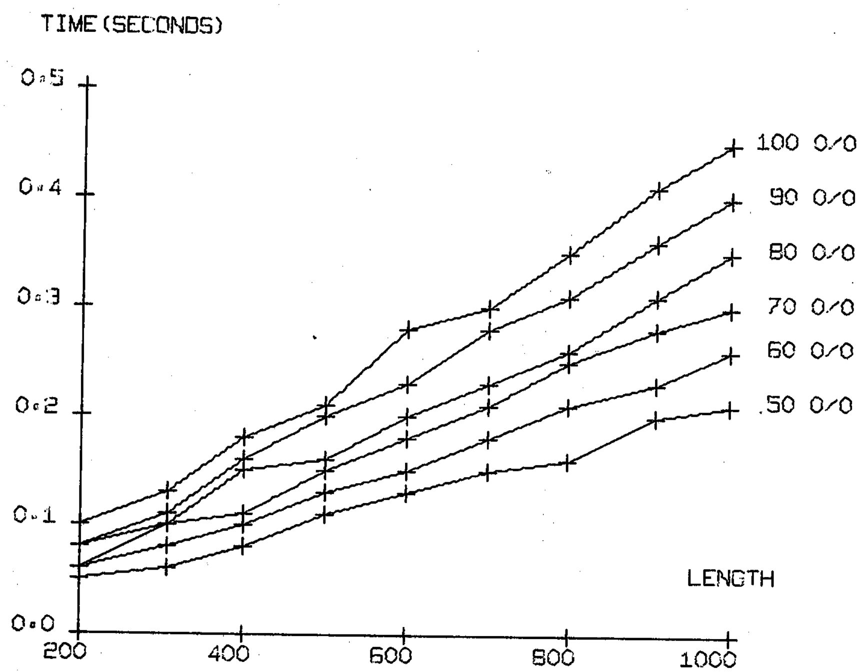
3.4 - TECHNIQUE BEHAVIOR IN RELATION TO DIFFERENT UTILIZATION PERCENTAGES.

Given that the utilization percentage concept is possible only for h.c. techniques and realizing that construction and search time are similar, the results are given for construction only.

3.4.1 - LINEAR AND QUADRATIC SEARCHES.

As figures 18, 19 and 20 show construction time decreases as utilization percentage decreases.

This fact is explained by the number of collision that increases as the utilization percentage increases.



HASH CODE SEARCH WITH OVERFLOW TABLES
CONSTRUCTION

FIGURE 21

3.4.2 - OVERFLOW H.C. TABLES.

The increasing of pointer vector length, with a constant number of table symbols, yields a smaller number of collisions. This results in less search time, due to smaller lists.

The results show that theoretical considerations are confirmed by the practical experiment.

Figure 21 shows in the abcissa the pointer vector length. Every curve in the figure is ruled by the equation

$$t_i = K_i \cdot \text{length}_i(\text{POINTER})$$

for every utilization percentage i. This equation, as a function of the table length would be.

$$t_i = K_i' \cdot \text{length}_i(\text{SYM})$$

for $$K_i' = K_i \cdot i/100 .$$

This consideration applies also to different techniques.

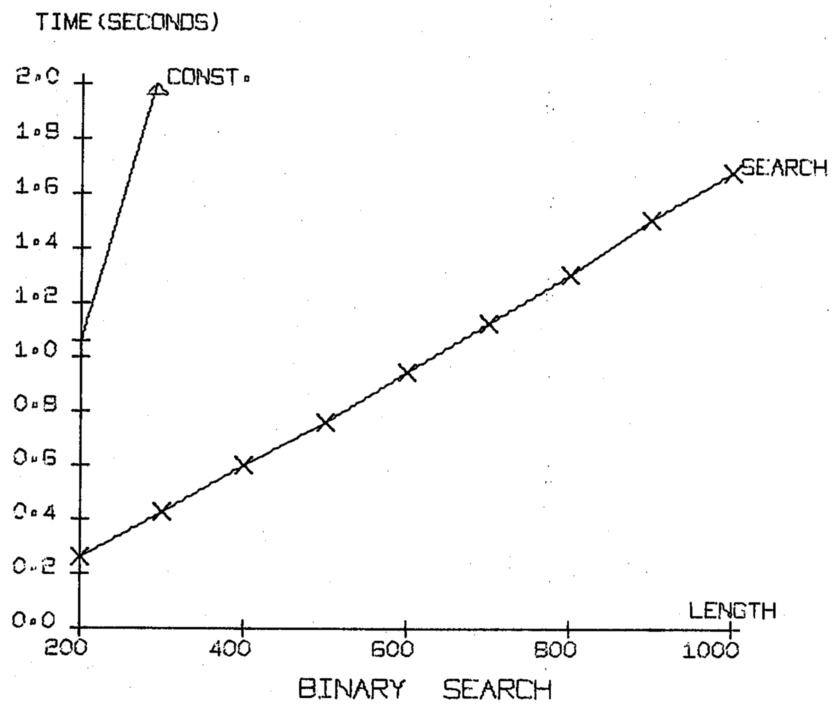

TIME (SECONDS)
CONST.
SEARCH
LENGTH
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
200
400
600
800
1000
BINARY SEARCH

FIGURE 22

3.5 - REGRESSION CURVES FOR GIVEN RESULTS.

After adding as origin the point time = 0 , $length_0 = 0$ a linear regression was applied.

3.5.1 - BINARY SEARCH.

The results of the regression as shown in figures 15 and 22 (the latter using a larger seale factor).

Give: Construction ,

$$t(secs) = 0.00138 \cdot length + 0.00002 \cdot length^2$$

Search:

$$t(secs) = 0.00174 \cdot length$$

The theoretical curve is

$$t = K \cdot \log_2 length$$

for $200 \leq length \leq 1000$ this curve is practically equal to the empirical curve.

3.5.2 - H.C. TECHNIQUES

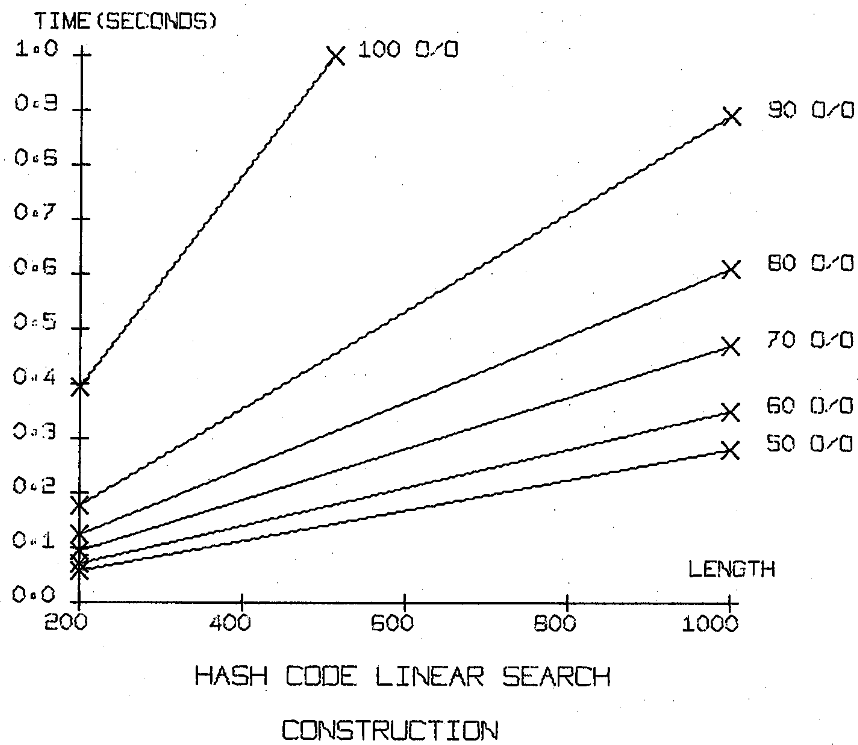
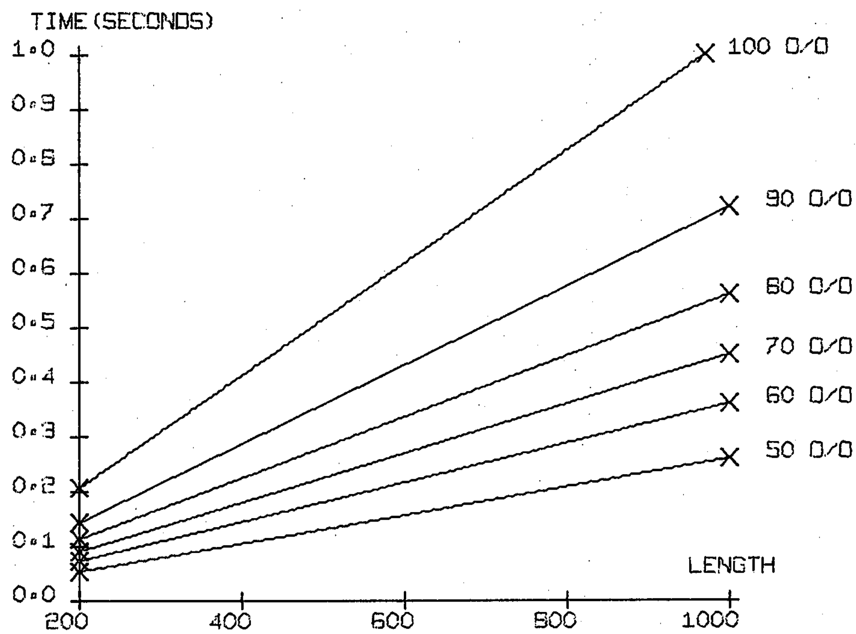

TIME (SECONDS)
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
100 O/O
90 O/O
80 O/O
70 O/O
60 O/O
50 O/O
LENGTH
200
400
600
800
1000
HASH CODE LINEAR SEARCH
CONSTRUCTION

FIGURE 23



HASH CODE QUADRATIC SEARCH (PARTIAL SCANNING)

CONSTRUCTION

FIGURE 24

3.5.2.1 - LINEAR SEARCH

$$t_{100} = 0.00196 \,.\, \text{table length}_{100} \text{(secs)}$$

$$t_{90} = 0.00089 \,.\, \text{table length}_{90} \text{ (secs)}$$

$$t_{80} = 0.00061 \,.\, \text{table length}_{80} \text{ (secs)}$$

$$t_{70} = 0.00047 \,.\, \text{table length}_{70} \text{ (secs)}$$

$$t_{60} = 0.00035 \,.\, \text{table length}_{60} \text{ (secs)}$$

$$t_{50} = 0.00028 \,.\, \text{table length}_{50} \text{ (secs)}$$

Figures 23 shows a graph of these curves.

3.5.2.2 - QUADRATIC SEARCH (PARTIAL SCANNING)

$$t_{100} = 0.00103 \,.\, \text{table length}_{100} \text{(secs)}$$

$$t_{90} = 0.00072 \,.\, \text{table length}_{90} \text{ (secs)}$$

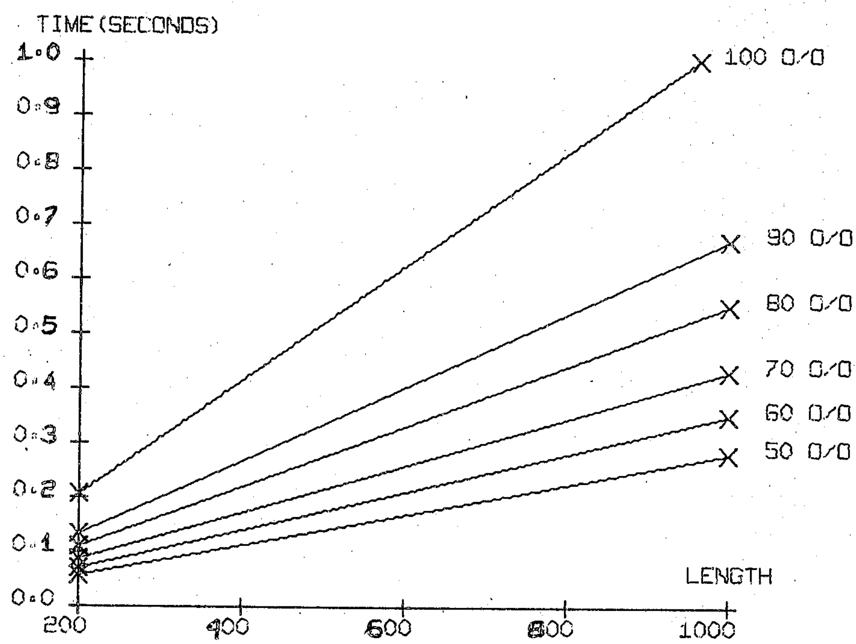
$$t_{80} = 0.00056 \,.\, \text{table length}_{80} \text{ (secs)}$$

$$t_{70} = 0.00045 \,.\, \text{table length}_{70} \text{ (secs)}$$

$$t_{60} = 0.00036 \,.\, \text{table length}_{60} \text{ (secs)}$$

$$t_{50} = 0.00026 \,.\, \text{table length}_{50} \text{ (secs)}$$

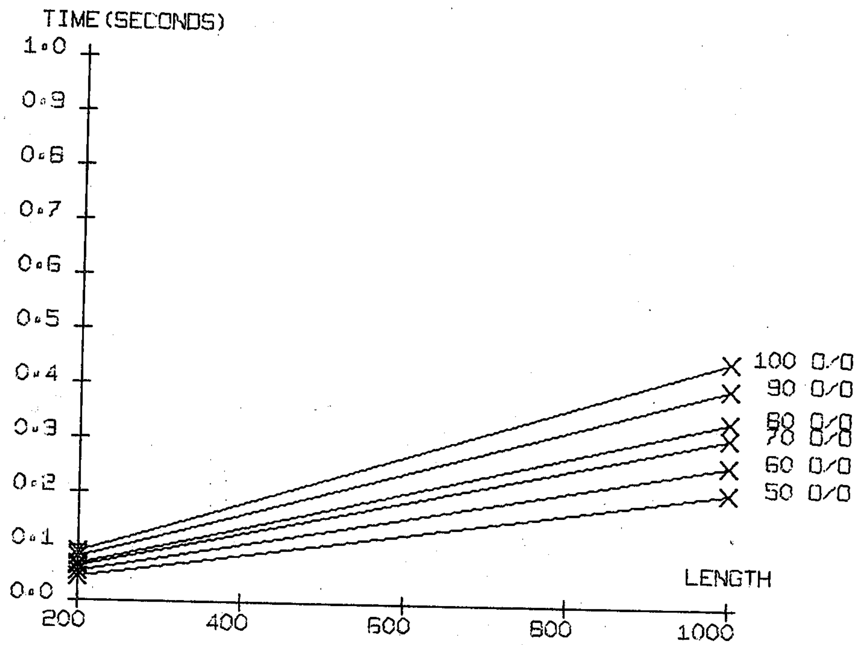
Figures 24 shows a graph of these curves.



HASH CODE QUADRATIC SEARCH (TOTAL SCANNING)

CONSTRUCTION

FIGURE 25



HASH CODE SEARCH WITH OVERFLOW TABLES

CONSTRUCTION

FIGURE 26

3.5.2.3. - QUADRATIC SEARCH(TOTAL SCANNING)

t_{100} = 0.00097 . table length (secs)

t_{90} = 0.00062 . table length (secs)

t_{80} = 0.00049 . table length (secs)

t_{70} = 0.00039 . table length (secs)

t_{60} = 0.00032 . table length (secs)

t_{50} = 0.00024 . table length (secs)

Figure 25 shows a graph of these curve.

3.5.2.4 - OVERFLOW TABLE SEARCH.

t_{100} = 0.00045 . PNTR length (secs)

t_{90} = 0.00040 . PNTR length (secs)

t_{80} = 0.00034 . PNTR length (secs)

t_{70} = 0.00031 . PNTR length (secs)

t_{60} = 0.00026 . PNTR length (secs)

t_{50} = 0.00021 . PNTR length (secs)

As a function of the table length.

	pointer length / table length
t = 0.00045 x table length	1.
t = 0.00036 x table length	1.1
t = 0.00027 x table length	1.25
t = 0.00022 x table length	1.43
t = 0.00016 x table length	1.68
t = 0.00010 x table length	2.00

4. - CONCLUSIONS.

In a general way h.c. techniques optimize processing time as more space is available for table storage.

This is not time for binary search since time remains constant for a given number of symbols in the table.

4.1 - PROCESSING TIME COMPARISON.

If we compare processing time among all techniques, with no regard to storage space, it becomes clear that overflow h. c. technique yields the best result, followed by quadratic and linear searches.

For a comparison between the latter three and binary search , we have the following result, since they take up the same memory space to store the

"symbol - attribute" pair.

1. Quadratic h.c. search (total scanning) 100%

2. Quadratic h.c. search (partial scanning) 100%

3. Binary search.

4. Linear h.c. search 100%.

4.2 - STORAGE SPACE COMPARISON

For a given problem regardless of the use technique, both SYM and ATR will have equal lengths. For binary search, linear and quadratic h.c. searches storage space is.

1) N . (length(SYM) + length (ATR))

Where N is the number of symbols in the table and length(XX) is the length of the named field.

For linear and quadratic h.c. searches we consider an utilization factor K.

2) $N \cdot K \cdot (\text{length(SYM)} + \text{length(ATR)})$

$K = 100/\text{utilization percentage.}$

For overflow h.c. technique besides SYM and ATR we have the field CHAIN and vector PNTR; therefore storage space will be:

3) $N \cdot K \cdot (\text{length (PNTR)}) +$

$N \cdot (\text{length(SYM)} + \text{length(ATR)} + \text{length(CHAIN)})$

In a general comparison equation 3 becomes

4) $N \cdot (K \cdot (\text{length(PNTR)} + \text{length(CHAIN)}) +$
$N \cdot (\text{length(SYM)} + \text{length(ATR)})$

For

5) $N \cdot (K \cdot (\text{length(PNTR)}) + \text{length(CHAIN)}$

6) $N \cdot (K-1) \cdot (\text{length(SYM)} + \text{length(ATR)})$

As much as 5 is closer to 6 the performance of the overflow h.c. tables is better than the performance of the other ones.

4.3 - METHOD FOR DETERMINING THE MOST CONVENIENT TECHNIQUE.

As a method for determining such technique the following procedure can be tried.

a) With the approximate number of symbols in the table and

b) With the usable memory space, the

c) Utilization factor K is determined through the use of the equations 2 and 3 of 4.2

d) Having computed K, the utilization percentage and the number of symbols(given by a) the length either the table or the pointer vector is found (making an approxi mation to the next prime number)

e) With the parameters obtained by d, the most efficient technique is found through the use of the graphs or through the regression curves.

4.4 - ADDITIONAL CONSIDERATION

The behavior of the different techniques for special con ditions is analyzed below.

4.4.1 - DELETION OF SYMBOLS IN THE TABLE

Binary search. The inverse process of symbol insertion showld be done, through a negative displacement.

Let I indicate the element to be deleted:

```
∀L((L = I,I+1,...,N-1)
    SYM(L) ← SYM(L+1);
    ATR(L) ← ATR(L+1););
 N ← N-1, END;
```

Linear or quadratic h.c. search. These techniques do not allow for the deletion of symbols since when an attempt is made, it interrupts the search sequence (an empty place is found). The solution to this problem could be to flag a symbol as non active, avoiding the interruption. This flag can be used as a replaceable symbol marker.

Overflow h.c. search. The elimination of a symbol is done by deleting the element in the corresponding list. The algorithm should be modified so that the available space will absorb the deleted element. This can be achieved by creating a list of available space.

4.4.2 - BLOCKED STRUCTURES.

- Binary search. Insertion and deletion in the table imply that displacements will be made, which is a slow action as compared with other techniques. The deletion of symbols implies a total scanning of the table.

- Linear and quadratic h.c. searches. The insertion of symbols is no problem. The deletion implies the use of the pro - cedure analyzed in 4.3.1, forcing a total scanning of the table.

- Overflow h.c. search. No problems are associated with deletion or insertion of symbols for this technique.

There is no need for scanning all the table since the list of symbols with the same code are groupings of elements of equal level, arranged in a decreasing order of levels (figure 27).

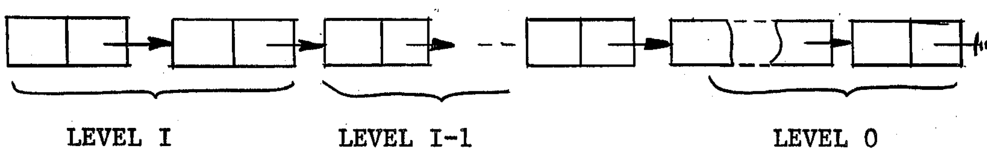


FIGURE 27

The deletion of elements of equal level (always done at the beginning of the list or at its higher level) is done by eliminating all nodes until a different level node is found.

Therefore this is the technique bust suited for this condition.

Note: The notation used for describing the algorithms in this work was adapted from Knuth's notation.

BIBLIOGRAPHY

1. Peter J. Denning. Computing Surveys ACM2 , 157-159 (1970).

2. James Bell R., Comm. ACM13, 107-109 (1970).

3. Leslie Lamport, Comm ACM13, 573-574 (1970).

4. Robert Morris, Comm ACM11, 38-44 (1968).

5. Alan Batson, Comm ACM8, 111-112 (1965).

6. Knuth Donald E., Fundamental Algorithms Adison Wesley (1968) pag 1-9.

7. A. Colin Day, Comm ACM13 , 481-482 (1970).

8. E.G. Coffman Jr.
   J. Eve , Comm ACM13, 427-432 (1970).