Fast lookup in hash tables with direct rehashing

J. A. T. Maddison

Thames Polytechnic, School of Mathematics, Statistics and Computing, Wellington Street, London SE18 6PF

The ideas of Mallach (1977) are developed to produce a practical method for organising a hash table with direct rehashing so as to reduce the number of probes needed to locate an item in the table.

Mallach (1977) discusses the possibility of reducing average access paths in a hash table with direct rehashing by inserting a new key in a position already occupied, and rehashing the entry in that position. The algorithm he describes has too large an overhead in setting up the table. Brent (1973) describes an algorithm involving a limited reorganisation of the table. This note describes a method of achieving the same results as Mallach's algorithm, but with less overheads, and compares its operation with Brent's algorithm.

It is assumed that we have a hashing function that can produce a series of locations for each key, and that the likelihood of any of these being free is a function of the table load factor only. In particular, if two keys are hashed to the same location, their subsequent locations will differ. It is also assumed that once the table has been completed, only a negligibly small proportion of attempted accesses will be to keys not in the table, and that all keys are equally likely to be accessed.

Suppose we wish to insert a key A in the table. Let the first n locations generated by the hashing function be $X_1, X_2, \ldots, X_n$. If there is an entry already in X_1, let subsequent positions generated by the hash function for this be $X_{11}, X_{12}, \ldots$. In general, if there is an entry at $X_{i\ldots k}$, let subsequent positions for this be $X_{i\ldots k1}, X_{i\ldots k2}, \ldots$. Note (**Fig. 1**) that we have a binary tree rooted at X_1, and that the sum of the subscripts of a node is the number of nodes of the tree that are traversed to that point. Let X_z be an arbitrary node, and $S(X_z)$ the sum of its subscripts.

If X_1 is free we place A there. Failing that we test X_2 and if that is free place A there. Next we try X_{11}. If this is free we move the entry at X_1 (B say) to X_{11} and put A at X_1. This reduces the average access time to A by at least 2 probes (X_3 might not be free) at a cost of increasing access to B by 1 probe.

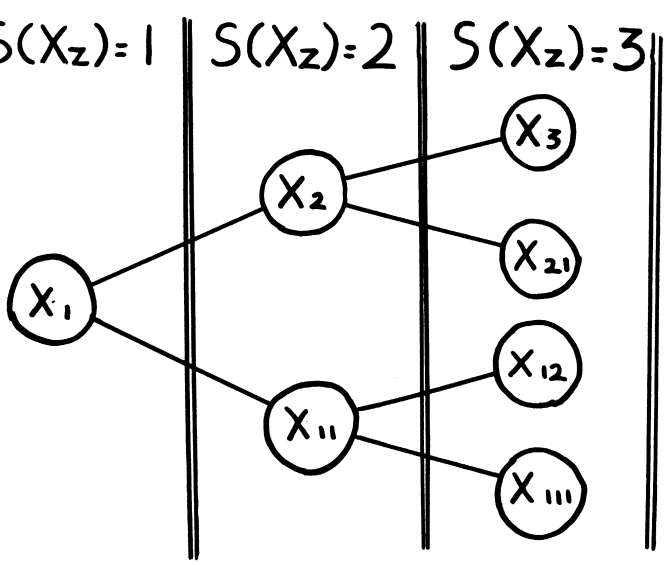

Fig. 1

More formally we proceed by testing all positions X_z for which $S(X_z) = n$ for $n = 1, 2, 3, \ldots$ until we find a free position. In terms of Mallach (1977) we search each level of the tree in turn. The position found may require shifting several entries, and there will need to be considerable record keeping both to show what moves will be needed to use the location found, and also to show which locations should be tested. How best to do this could be said to depend on the facilities available in the system used by the programmer. Something on the following lines would be suitable:

($p[i]$ contains the next possible location for a key, its position in the table, and hash function coefficients).
next: = 1; free: = 2; found: = false;
set $p[1]$ to refer to first possible position of new key (X_1);
while not found **do**
if position in p[next] free **then** found: = true
else set p[free] to refer to next position for key referred to by p[**next**];
  set p[free+1] to refer to next position for key in table location given by p[next]
  next: = next+1; free: = free+2
fi
od;
use information in p to make necessary moves;

Note for each value n of $S(X_z)$, X_n would be tested first. It would be possible at the expense of more complications to test for each value of n the various X_z in ascending order of the number of changes to the table that would be required if they were free. It might also be convenient if any coefficients required by the hashing mechanism were stored when a key is first inserted in the table, as this would save the need for extra computation when the possibility of rehashing that key is considered.

In comparison with Brent's algorithm this method would require less probes when inserting a key, since it stops at the first free location found, and will give shorter average access paths, since it considers more possibilities. The average number of probes for inserting a key will be the same as for simple direct rehashing, If the table load factor is α this is well known to be $\frac{1}{\alpha}\log_e\frac{1}{1-\alpha}$. The average number of probes to access an item is the same as in Mallach's algorithm—the difference is that the tree is searched width first rather than depth first—and he states this to be $\frac{1}{\alpha}\sum_{k=0}^{\infty}\alpha^{2^k}2^{-k}$. (A derivation of this result is outlined in Appendix 1).

Mallach's (1977) Table 1 compares the average number of probes to locate a key, using his algorithm, Brent's and simple direct rehashing. This shows that his algorithm, and hence the one described in this note, will give the best results, especially with fairly full tables. Direct chaining will give better results, but this method would be worth considering when there is a need to use direct rehashing, and either there is a specific need to minimise average access paths or each key will, on average, be accessed several times.

Appendix Outline of derivation of formula for average number of probes to locate key

When inserting a key

Let r be the probability that a position in the table is occupied and $p(=1-r)$ that it is free.

Probable hashing depth $= p + 2(pr + pr^2) + 3(pr^3 + pr^4 + pr^5 + pr^6) + \ldots$

$$= \frac{1}{r}\sum_{n=1}^{\infty}(nr^{2n-1} - nr^{2n}) \text{ by summing series in brackets}$$

$$= \sum_{n=1}^{\infty} r^{2n-1-1} \text{ by expanding power series.}$$

If final table load factor is α,

$$\text{Average search depth} = \frac{1}{\alpha}\int_0^{\alpha}\sum_{m=1}^{\infty} r^{2m-1-1}\,dr$$

$$= \frac{1}{\alpha}\sum_{m=1}^{\infty}\int_0^{\alpha} r^{2m-1-1}\,dr \text{ since series is uniformly convergent}$$

$$= \frac{1}{\alpha}\sum_{m=0}^{\infty}\alpha^{2m}\,2^{-m}$$

References

BRENT, R. P. (1973). Reducing the Retrieval Time of Scatter Storage Techniques, *CACM*, Vol. 16 pp. 105-109.

MALLACH, E. G. (1977). Scatter Storage Techniques: A Unifying Viewpoint and a Method for Reducing Retrieval Times, *The Computer Journal*, Vol. 20 No. 2, pp. 137-140.

To the Editor
The Computer Journal

Sir,

Points, polygons, and areas

In the August 1979 issue of *The Computer Journal* M.A. Sandys described a method to determine whether a point lies inside or outside an n-sided irregular figure. In commenting on this paper Aleph Null described the method, long familiar to quantitative geographers, for tackling this problem efficiently. I attach a FORTRAN subroutine, written several years ago by R. Franklin of the University of Ottawa, using the latter method.

Aleph Null also raises the question of calculating the area of a n-gon using a method similar to that of Sandys'. An efficient FORTRAN routine using the polygon coordinates is as follows:

Let $X(1), \ldots X(N)$ and $Y(1) \ldots Y(N)$ be the N polygon vertex co-ordinates (defined clockwise or anticlockwise around the polygon), and let $X(N+1) = X(1)$, $Y(N+1) = Y(1)$.

Then:

```
    AREA = 0·0
    DO 100 I = 1,N
    J = I + 1
    AREA = AREA + (X(J) - X(I))* (Y(J) + Y(I))/2·0
100 CONTINUE
    AREA = ABS (AREA)
```

computes the polygon area. The algorithm is essentially trapezoidal integration applied to a closed figure.

Yours faithfully,
MICHAEL DE SMITH

Joan de Smith & Partners Ltd
41 Gloucester Place
London W1H 3PD

27 September 1979

```
      SUBROUTINE PNPOLY(PX,PY,XX,YY,N,INOUT)                                PPOL  20
C     ***********************************************************************PPOL  30
C                                                                           PPOL  40
C        SUBROUTINE PNPOLY                                                  PPOL  50
C                                                                           PPOL  60
C        PURPOSE                                                            PPOL  70
C           TO DETERMINE WHETHER A POINT IS INSIDE A POLYGON                PPOL  80
C                                                                           PPOL  90
C        USAGE                                                              PPOL 100
C           CALL PNPOLY (PX, PY, XX, YY, N, INOUT )                         PPOL 110
C                                                                           PPOL 120
C        DESCRIPTION OF THE PARAMETERS                                      PPOL 130
C           PX      - X-COORDINATE OF POINT IN QUESTION.                    PPOL 140
C           XX      - N LONG VECTOR CONTAINING X-COORDINATES OF             PPOL 150
C                     VERTICES OF POLYGON.                                  PPOL 160
C           YY      - N LONG VECTOR CONTAINING Y-COORDINATES OF             PPOL 170
C                     VERTICES OF POLYGON.                                  PPOL 180
C           PY      - Y-COORDINATE OF POINT IN QUESTION.                    PPOL 190
C           N       - NUMBER OF VERTICES IN THE POLYGON.                    PPOL 200
C           INOUT   - THE SIGNAL RETURNED:                                  PPOL 210
C                     -1 IF THE POINT IS OUTSIDE OF THE POLYGON,            PPOL 220
C                      0 IF THE POINT IS ON AN EDGE OR AT A VERTEX,         PPOL 230
C                      1 IF THE POINT IS INSIDE OF THE POLYGON.             PPOL 240
C                                                                           PPOL 250
C        REMARKS                                                            PPOL 260
C           THE VERTICES MAY BE LISTED CLOCKWISE OR ANTICLOCKWISE.          PPOL 270
C           THE FIRST MAY OPTIONALLY BE REPEATED, IF SO N MAY               PPOL 280
C           OPTIONALLY BE INCREASED BY 1.                                   PPOL 290
C           THE INPUT POLYGON MAY BE A COMPOUND POLYGON CONSISTING          PPOL 300
C           OF SEVERAL SEPARATE SUBPOLYGONS. IF SO, THE FIRST VERTEX        PPOL 310
C           OF EACH SUBPOLYGON MUST BE REPEATED, AND WHEN CALCULATING       PPOL 320
C           N, THESE FIRST VERTICES MUST BE COUNTED TWICE.                  PPOL 330
C           INOUT IS THE ONLY PARAMETER WHOSE VALUE IS CHANGED.             PPOL 340
C           THE SIZE OF THE ARRAYS X AND Y MAY BE CHANGED TO                PPOL 350
C           INCREASE THE SIZE OF THE POLYGONS TO BE HANDLED                 PPOL 360
C           OR TO DECREASE CORE SPACE REQUIRED.                             PPOL 370
C           WRITTEN BY RANDOLPH FRANKLIN, UNIVERSITY OF OTTAWA, 7/70.       PPOL 380
C                                                                           PPOL 390
C        SUBROUTINES AND FUNCTION SUBPROGRAMS REQUIRED                      PPOL 400
C           NONE                                                            PPOL 410
C                                                                           PPOL 420
C        METHOD                                                             PPOL 430
C           A VERTICAL LINE IS DRAWN THRU THE POINT IN QUESTION. IF IT      PPOL 440
C           CROSSES THE POLYGON AN ODD NUMBER OF TIMES, THEN THE            PPOL 450
C           POINT IS INSIDE OF THE POLYGON.                                 PPOL 460
C                                                                           PPOL 470
C     ***********************************************************************PPOL 480
C                                                                           PPO
000011      DIMENSION X(200),Y(200),XX(N),YY(N)                              PPO
000011      LOGICAL MX,MY,NX,NY                                             PPOL 510
000011      DO 1 I=1,N                                                      PPOL 520
000012      X(I)=XX(I)-PX                                                   PPOL 530
000014    1 Y(I)=YY(I)-PY                                                   PPOL 540
000021      INOUT=-1                                                        PPOL 550
000022      DO 2 I=1,N                                                      PPOL 560
000023      J=1+MOD(I,N)                                                    PPOL 570
000027      MX=X(I).GE.0.0                                                  PPOL 580
000033      NX=X(J).GE.0.0                                                  PPOL 590
000037      MY=Y(I).GE.0.0                                                  PPOL 600
000043      NY=Y(J).GE.0.                                                   PPOL 610
000047      IF(.NOT.((MY.OR.NY).AND.(MX.OR.NX)).OR.(MX.AND.NX)) GO TO 2     PPOL 620
000055      IF(.NOT.(MY.AND.NY.AND.(MX.OR.NX).AND..NOT.(MX.AND.NX))) GO TO 3 PPOL 630
000062      GO TO 5                                                         PPOL 640
000064    3 IF((Y(I)*X(J)-X(I)*Y(J))/(X(J)-X(I))) 2,4,5                     PPOL 650
000076    4 INOUT=0                                                       PPOL 660
000077      RETURN                                                          PPOL 670
000077    5 INOUT=-INOUT                                                  PPOL 680
000100    2 CONTINUE                                                      PPOL 690
000103      RETURN                                                          PPOL 700
000103      END                                                             PPOL 710
```

To the Editor
The Computer Journal

Sir,

Jumping to some purpose

Arblaster, Sime and Green (1979) find that the use of GOTOs or similar constructs in coding a simple problem (the 'hungry hare') can have practical advantages over structured programming and cite, in particular, the benefits of the FORTRAN 0 logical IF construct. Their programming example is perhaps untypical in that it contains no loops and no procedures which appear more than once in the flow tree (**Fig. 1**).

The advantage of a balanced logical IF construct is that it leads to the most concise representation of the flow tree structure as is shown in the following 'hierarchic' coding where each level of the tree is described in turn:

(*a*) IF green THEN chop:GOTO *b* ELSE GOTO *c*
(*b*) IF juicy THEN grill ELSE roast
(*c*) IF hard THEN peel:boil ELSE fry

In this coding the GOTOs perform no logical function, their job being only to break down the text into manageable sentences. If

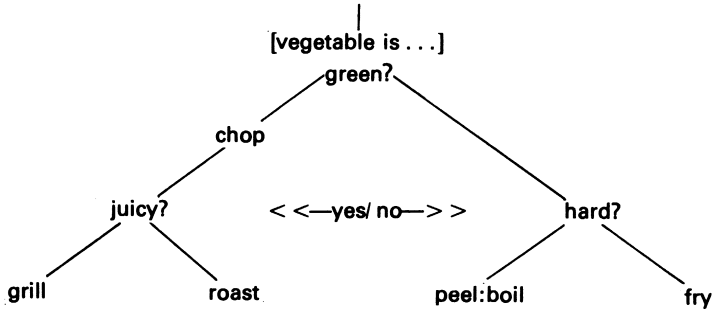

Fig. 1

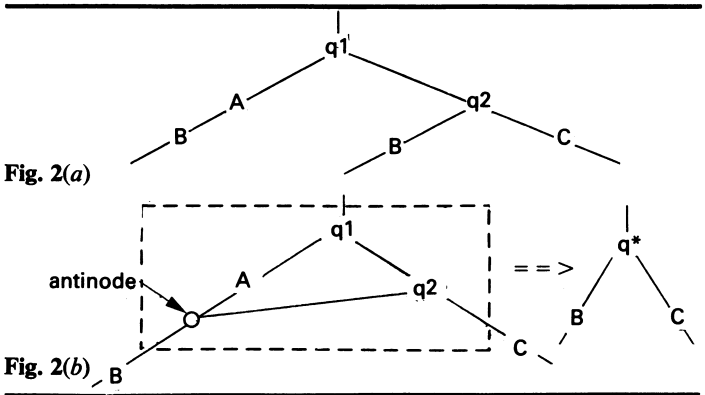

Fig. 2(*a*)

Fig. 2(*b*)

each GOTO statement is replaced by the text to which it points then the nested form of the program emerges automatically. This kind of GOTO may be called 'virtual' because it affects neither logic nor structure.

In my view the key to true unstructuredness lies in information, one bit of which is released by each conditional element (node) in a flow tree. If information is subsequently absorbed (at an 'antinode', where two branches merge), then the flow diagram is no longer a tree and unstructuredness occurs. This condition must exist if a loop is present, but it may be possible to 'internalise' the unstructuredness to a subdiagram, which can then be represented as a single conditional or procedural element.

The WHILE-DO loop is a good example of an unstructured construct which, because it has one entry point and one exit, may be replaced by a procedure (branch) in a flow tree.

Reducibility of a task in a two-valued logic environment thus depends on the availability of such constructs for internalising processes which absorb information. This is a function of human language conventions which appear ill equiped to deal with anything more complex than IF-THEN, WHILE-DO and similar constructs, as illustrated above.

Unstructuredness is not an inherent feature of any problem since a flow tree, albeit infinite, can always be constructed. In the interests of economy, however, it is felt desirable to identify similar conditional and procedural elements in the tree.

In the example Fig. 1(*a*) of Williams and Ossher (1978), whose flow tree is shown in **Fig. 2**(*a*), the urge arises to identify the two procedures *B*, which happen to be identical. This cannot be done without losing structure unless the unstructuredness is internalised to a complex conditional element as in Fig. 2(*b*). Although *q** is a feasible elementary construct words fail to describe it concisely and the diagram remains, for practical purposes, unstructured.

It may be observed that the arc in **Fig. 2**(*b*) linking *q*2 and the antinode represents a 'real' GOTO, which is quite distinct from the benign 'virtual' kind. It is unfortunate that these two constructs, share the same name and are thus tarred with the same brush. If, for example, the 'virtual' GOTO is renamed REFER then the confusion, and much of the acrimony, may be avoided since it is possible to detect and flag the case where a label is referred to by more than one REFER statement.

Yours faithfully
N. B. TAYLOR

'Hook-a-gate'
Eversley Road
Yately, Surrey
26 September 1979

References

ARBLASTER, A. T., SIME, M. E. and GREEN, T. R. G. (1979) Jumping to some purpose, *The Computer Journal*, Vol 22 No. 2, pp. 105-109.

WILLIAMS, M. H. and OSSHER, H. L. (1978). Conversion of unstructured flow diagrams to structured form, *The Computer Journal*, Vol 21 No 2 pp. 161-167.

To the Editor
The Computer Journal

Sir,

Points and n-sided irregular figures

I have just noticed your correspondence about determining whether a point is inside or outside the given n-sided irregular figure and I would draw your attention to the great many solutions to this problem that have been produced in the field of Urban and Regional studies over the past seventeen years.

When the Department of the Environment considered this subject in 1975 in their Research Report 2 (Point-in-Polygon Project Stage I) they identified fourteen algorithms that solved the problem and of these nine had been published.

Yours faithfully,
MERVYN BRYN-JONES

Borough of Haringey
Hornsey Town Hall
The Broadway
Crouch End
London N8 9JJ
24 October 1979

Book review

Mini/Microcomputer Hardware Design by G. D. Kraft and W. N. Toy, 1979; 514 pages. (*Prentice-Hall*, £12·80)

This book analyses different approaches to small computer design. By concentrating on conventional small machines however, the treatment must be incomplete. Important architectural features that are traditionally software, but which increasingly affect the hardware, e.g. semaphore control, are left out. And large machine features such as virtual memory, which are becoming available on micros, are given little attention.

Chapters 1 to 6 review the structure of microcomputers and some of the earlier microprocessors (up to 8080, 6800). It shows how fundamental ideas, for example unified bus structures, and instruction code formats, have been incorporated into different machines. The text demands a basic understanding of computer systems by the reader, for ideas and words are often used without introduction. Chapter 7 describes the design of microcontrol units for microprogram sequencing of instruction fetch and execution cycles. In comparison with the rest of the book, this material is excessively detailed, and could usefully have been restricted to allow discussion of ROM-based microprogram techniques, which are excluded. Chapters 8 and 9 discuss program controlled input/output and direct memory access. The treatment of interrupts, for example, is comprehensive, but there is no discussion of modern programmable I/O controllers or special purpose I/O processors. The book's five year gestation clearly shows.

The useful and interesting examples are peculiar in that many cannot be tackled using this book. For example distributed computing is recognised as important, and several examples deal with it. However, there is only the briefest treatment of the subject in the text itself. Likewise error control and testing are practically ignored except in the examples. Perhaps there is a second book under way. Hence, while its choice of material is too uneven to provide a complete study of the subject, the book reviews and compares many approaches to hardware design. It is clearly written, and would make a useful back-up reference for system designers and students.

R. W. PROWSE (Uxbridge)