An introduction to computed chaining

by KUO-CHUNG TAI and ALAN L. THARP
North Carolina State University
Raleigh, North Carolina

INTRODUCTION

A hashing function, \( H(x) \), is a transformation from a key value \( x \) to an address. Since such transformations may produce the same address for distinct keys, hashing functions lead to collisions in the address space. Many methods for resolving hashing collisions have been reported [1,2]. Direct chaining and open addressing are the two basic collision-resolution methods. This paper presents a form of hybrid hashing, computed chaining, which is better than a previously described hybrid hashing procedure called pseudochaining [3]. In certain situations it is also better than other collision procedures.

In direct chaining, distinct items which hash into the same home address are linked into a chain. When a collision occurs while inserting an item \( x \), if the item at \( H(x) \) is stored at its home address, then \( x \) is stored at the first empty cell encountered on the chain starting at \( H(x) \). If the item at \( H(x) \) is not stored at its home address, then that item is moved into an empty cell, its chain is updated, and \( x \) is inserted at \( H(x) \). The links in direct chaining are addresses of overflow items. Figure 1(a) shows the use of direct chaining for items \( A, B, \) and \( C \) which have the same home address, \( r \), and are inserted in the given order. Only one probe is required to retrieve an overflow item from its predecessor.

In open addressing, when a collision occurs while inserting an item \( x \), a function of \( x \) is used to determine the probe sequence. The first empty cell encountered in the probe sequence is used to store item \( x \). The same probe sequence will later be used to retrieve item \( x \). Many probe-sequence generating functions have been studied [1,2]. Figure 1(b) illustrates the use of open addressing for items \( A, B, \) and \( C \). The number of probes required to locate an empty cell for storing an overflow item is a function \( f \) of that item and all other items in the table, and that same number of probes is later required to retrieve that item.

Pseudochaining [3] combines features of direct chaining and open addressing in the following way. Suppose that item \( x \) is stored at its home address \( H(x) \). When the first item \( y \) with \( H(y) = H(x) \) (\( y \) is called the first overflow item) is inserted, an empty cell for storing \( y \) is linked using open addressing and the number of probes required to locate the empty cell is stored in the link field of \( H(y) \). By doing so, only one probe is required to retrieve \( y \) from its home address because the address of \( y \) can be computed from the probe number stored in the link field of \( H(y) \). Since the probe number typically requires fewer bits than the full address, pseudochaining needs less link space than direct chaining. Subsequent overflow items with a home address of \( H(y) \) are also stored using open addressing, but they do not necessarily have the property of one-probe retrieval from \( H(y) \). Figure 1(c) shows the use of pseudochaining for items \( A, B, \) and \( C \). \( f(B) \) is stored as the link of \( A \) so that only one probe is required to retrieve \( B \) from the address of \( A \). \( f' \) is the modified \( f \) after \( B \) is inserted (see [3] for details).

Computed chaining is another form of hybrid hashing that uses probe numbers as links instead of actual addresses. This new method, however, is closer to direct chaining than is pseudochaining, because all items with the same home address are linked into a chain. Assume item \( x \) is stored at its home address, \( H(x) \). When inserting the first overflow item \( y \) with \( H(y) = H(x) \), computed chaining uses a function of \( x \) (not \( y \)) to determine the probe sequence and then stores (in the link field of \( H(x) \)) the number of probes required to find an empty cell for storing \( y \). Later when inserting the second overflow item \( z \) with \( H(z) = H(x) \), computed chaining first computes the address of \( y \) using item \( x \) and its link. Then this method uses a function of \( y \) to determine the probe sequence starting from the address of \( y \). It stores as the link of \( y \) the number of probes required to find an empty cell for storing \( z \). By doing so, an item plus its address and link (a probe number) determine the address of the next item in the same chain. Thus, items with the same home address are linked into a chain without the addresses being stored; instead the addresses can be computed by using the probe numbers stored in the link fields. This method is similar to direct chaining in that only one probe is required to retrieve an overflow item from its predecessor. Figure 1(d) illustrates the use of computed chaining for items \( A, B, \) and \( C \). The number of probes required to locate an empty cell for storing an overflow item is a function \( g \) of its predecessor. By storing \( g(A) = (g(B)) \) as the link of \( A \) (\( B \)), only one probe is required to retrieve \( B \) (\( C \)) from the address of \( A \) (\( B \)).

DESCRIPTION OF COMPUTED CHAINING

The previous section highlighted computed chaining and compared and contrasted it with other well-known hashing
methods. A more detailed description of computed chaining will be given for the operations probe, store, move and retrieve. Probe computes the next probe address based upon the linear quotient hashing method. Store places an item into the table according to the computed chaining method. Move displaces an item not stored at its home address and all subsequent items stored on that chain. And finally, retrieve locates an item in the table. For this discussion, the linear
quotient hashing scheme [4] is used as the hashing function and method for determining the address of the next probe. Computed chaining is not dependent upon the linear quotient hashing scheme; hashing functions other than the one which follows may be used with computed chaining.

procedure probe (k, prob#, addr)
    //computes the next probe address by the linear quotient method using the key, probe number and current address/
    1. incr ← (kp) mod p //obtain increment/
    2. if incr = 0 then incr ← 1 //insures a positive increment/
    3. return ((addr + prob# * incr) mod p) //return new address/
end probe

k is the key, prob# is the probe number and addr is the current address in the table. p is the prime number table size.

Using the probe procedure, store locates the first empty cell or one that contains an item not at its home address. In the latter case, a call is made to the move procedure to free the cell for insertion. The new item is then inserted. If the new item is an overflow item, the number of probes for locating the new item from its predecessor is stored in the link field of the predecessor. To subsequently access the new item, only a "single" probe from the predecessor is needed because the address of the new item can be computed from the address, key and link value of the predecessor.

procedure store(x)
    //stores an item with key x according to the computed chaining hashing method/
    1. h ← x mod p //locate the home address using the linear quotient method/
    2. if T.h = 0 then [T.h ← x; return] //the home address is empty, so store the item
    3. if T.h = x then return //item is a duplicate/
    4. if T.h mod p ≠ h then //call move (h); T.h ← x; return
        [item stored at h is not at its home address, so move it and store x at h/]
        //locate the last item in this chain/
    5. while L.h ≠ 0 do
    6. h ← probe (T.h, L.h, h) //locate the next probe address using the probe number stored at h, L.h/)
    7. if T.h = x then return //item is a duplicate/
    8. end
    9. i ← 1 //initialize a loop variable to locate the first empty cell for storing x/)
    10. j ← probe (T.h, i, h) //locate the next probe address/
    11. while T.j ≠ 0 do
    12. i ← i + 1
    13. if i > p then [print("overflow"); stop]
    14. j ← probe (T.h, i, h) //locate the next probe address/
end store

T.h refers to the key at location h and L.h is the link value at location h. T.h = 0 means that the cell at location h is empty. Initially T.h = 0 and L.h = 0 for 0 ≤ h ≤ p – 1.

The move procedure in computed chaining is essential in minimizing the number of retrieval probes. As mentioned previously, it moves an item not at its home address. Since the address and link of the item being moved is used to locate subsequent items in the same chain, it is necessary then to move all of these subsequent items. A simple solution is to use an array of temporary storage locations to queue all items to be moved. Then one by one (first-in-first-out) the items stored in the array are moved to new locations. By doing so, the order of the moved items remains the same after moving. (It is unnecessary to keep the same order or to use the array during moving. A better solution which does not need the array (or stack) exists, but it is not described here due to its complexity.) In this moving operation, the associated probe numbers are updated. Finding the new locations of the moved items is essentially the same as in the store operation. Note that the move operation and the special care associated with its use are performed only during the insertion of an item. If coalescence of multiple chains is allowed, then the move procedure is unnecessary but the performance of computed chaining will be degraded somewhat (see [1, pp. 514-518]). The move operation is recommended when the ratio of insertions to retrievals is quite small.

procedure move(r)
    //moves an item at location r not stored at its home address and all subsequent items in that computed chain/
    1. declare TEMP(1:100) //declare array for storing r and subsequent items in the computed chain/
    2. h ← T.r mod p //locate home address for item stored at r/)
    3. while probe(T.h, L.h, h) ≠ r do
    4. h ← probe(T.h, L.h, h)
    5. end
    6. i ← 1 //initialize array index/
    7. TEMP(i) ← T.r //store the item at r/)
    8. y ← r
    9. while L.y ≠ 0 do
    10. nexty ← probe(T.y, L.y, y) //find the next item/
    11. i ← i + 1 //increment the array index/
    12. TEMP(i) ← T.nexty //store the item found/
    13. T.y ← 0 //erase the item at y/)
    14. T.y ← 0
    15. y ← nexty //continue the search/
    16. end
    17. T.y ← 0 //erase the last item/
From the collection of the Computer History Museum (www.computerhistory.org)

The final procedure, retrieve, is similar to the linear quotient retrieval scheme except for the probe function used to calculate the next probe address.

procedure retrieve(x)
    //retrieves an item with key x which was stored
    //according to the computed chaining hashing method/
    1. h ← x mod p //locate the home address using the
    //linear quotient method/
    2. if T.h = x then return
    3. i ← 1 //initialize a loop variable for searching x/
    4. while (i < p and T.h ≠ 0) do
        5. h ← probe(T.h, L.h, h) //locate the next probe
        //address/
    6. if T.h = x then return
    7. i ← i + 1 //increment the loop variable/
    8. end
    9. print ('item is not in the table')
end retrieve

If space is at a premium and sufficient bits in the link field are not available to store the "complete" probe number, a technique from pseudochaining of using a function of the "complete" probe number (prob#) and the number of bits of the link field (s) can be incorporated into computed chaining. This function, GBD(prob#, s), gives the greatest divisor of prob# that is less than 2**s and relatively prime to p. To incorporate this modification into the previously defined procedures, it is necessary to replace statement 16 in store with

L.h ← GBD(i,s) //insert the probe number/

and line 27 in move should be replaced with

L.h ← GBD(k,s) //store the probe number/

In addition, the use of the probe procedure to both find a successor and insert an item should be changed.

First, consider the case of using the probe procedure for finding a successor. Assume that while inserting item y, the number of probes required to locate an empty cell for y from its predecessor x is i. Then GBD(i,s) is stored in the link field of x. To retrieve y from x, when the link field has sufficient bits (i<2**s), only one probe is necessary. However, if insufficient bits (i≥2**s) are available, multiple probes may be required (see Figure 2). Let item w be one of the intermediate cells probed. Then one of these cases must hold:

(1) w is not an item in the chain. w could be zero as the result of a move or deletion.
(2) w is an item in the chain, but it precedes x, or
(3) w is an item in the chain, but it follows y.

Note that case (3) should not be allowed, for if it were, items between x and w would not be accessible. To avoid case (3), care must be taken when inserting an item using the probe procedure as described in the next paragraph. If case (3) does not occur, then cases (1) and (2) can be detected easily because w is in the chain if and only if w ≠ 0 and either H(w) = H(x) if no coalescence is allowed, or H(w) = H(z) where z is any of the items preceding w in the chain if coalescence is allowed.

Then consider the case of using the probe procedure to find an empty cell for inserting an item. To avoid case (3) in subsequent retrieval, the empty cell must not be any of the empty cells which have been encountered prior to finding the last item of the chain on which the item is being inserted.

Two methods for both locating a successor and inserting an item when using the GBD function are suggested. One method would be to add a bit field to each cell in storage. This bit would signify whether a cell had been encountered previously in the same store, retrieve or move operation. When locating a cell with the probe procedure, this extra bit would be checked. Another procedure to accomplish the same result would be to establish a stack for storing, for subsequent interrogation, cells (or their addresses) previously encountered in the probe procedure. For a table of relatively short items, the latter method would be more space efficient.

The computed chaining technique described above is a hybrid of direct chaining and the linear quotient method of open addressing. Other open addressing methods can also be used in computed chaining. The reason that linear quotient was chosen is that it has the best performance among all "static" hashing methods (i.e., those which do not reorganize the table when inserting new items).

The idea of reorganizing the table when inserting new items, which is used in "dynamic" hashing methods [5-10], can be used in computed chaining to improve its perform-
An Introduction to Computed Chaining

\[ j = \text{GBD}(i, s) \]

\[
\begin{array}{cc}
\text{key} & \text{link} \\
\hline
x & j \\
\hline
y & \\
\end{array}
\]

one probe

\[
\begin{array}{cc}
\text{key} & \text{link} \\
\hline
x & j \\
\hline
w & \\
\hline
y & \\
\end{array}
\]

(i/j) probes

\[
\begin{array}{cc}
\text{key} & \text{link} \\
\hline
\end{array}
\]

\( (j < i \text{ and is a divisor of } i) \)

Figure 2—Locating successor item with computed chaining.

\[ i < 2**s \]

\[ (j = i) \]

(a) \( i < 2**s \)

(b) \( i > 2**s \)

However, the hybrid of direct chaining and linear probing does not perform as well as the hybrid of direct chaining and linear quotient. Linear probing is less efficient than linear quotient for the problems of primary clustering and secondary clustering (see [9]). These two problems will increase the values of “complete” probe numbers and thus degrade the performance of computed chaining based upon linear probing.

\[
\begin{array}{cc}
\text{key} & \text{link} \\
\hline
r & A \\
\hline
s & B \\
\hline
t & C \\
\end{array}
\]

\[ s = r + b \times m_1 \]

\[ t = s + b \times m_2 \]

\[ t = r + b \times (m_1 + m_2) \]

where \( b \) is a function of \( A \)

Figure 3(a)—Illustration of computed chaining based upon linear probing.
ing when the link space is insufficient to store all of the “complete” probe numbers.

PERFORMANCE OF COMPUTED CHAINING

The computed chaining technique described above was tested with a table of size \( n = 997 \) using pseudorandom keys. After the table was filled for a specific loading factor \( \alpha \), each item stored in the table was retrieved once and the mean number of probes required to retrieve an item was computed. Table I shows the mean number of probes for retrieval as a function of \( \alpha \) and \( s \) (the number of bits in the link field). The last column of Table I shows \( \minis(\alpha) \), the minimum value of \( s \) required to store the “complete” probe numbers for a given \( \alpha \). When \( \alpha \) is smaller so is \( \minis(\alpha) \) because fewer probes are required to find empty cells. For a given \( \alpha \), when \( s > \minis(\alpha) \), the same mean number of probes for retrieval is required as when \( s = \minis(\alpha) \). In fact, computed chaining needs the same number of retrieval probes as direct chaining when \( s = \minis(\alpha) \). Since \( \minis(\alpha) \) is usually smaller than \( \lceil \log_2 n \rceil \), where \( n \) is the table size, computed chaining may take less space than direct chaining without loss of efficiency. As shown in Table I, when \( \alpha = 0.99 \), computed chaining requires only eight bits rather than ten bits as in direct chaining.

When \( s < \minis(\alpha) \) for a given \( \alpha \), more probes are needed to retrieve items because “partial” probe numbers (obtained using the GBD function) are stored in the link fields. Table I indicates that the performance of computed chaining degrades only slightly as \( s \) becomes smaller. The reason is that most of the “complete” probe numbers are small and thus the number of “partial” probe numbers in the link fields increases slightly as \( s \) becomes smaller. Table II shows the distribution of “complete” probe numbers for \( \alpha = 0.99 \). Sixty-five percent of the “complete” probe numbers are zeros and only five percent require more than four bits.

COMPARISONS WITH OTHER METHODS

As discussed in the previous section, computed chaining, compared with indirect chaining, has the following advantages: (1) it performs as well as direct chaining with less space and (2) it uses even less space by slightly degrading its performance.

Recently several improvements on open addressing which reduce the mean number of probes have been reported. Brent [5] suggested a method of reordering the table when new items are inserted. (A modification of Brent’s method was described in Tharp [6].) Brent’s method requires an average of about 2.49 probes to retrieve an item from a full table \((\alpha = 1.0)\). Gonnet and Munro [7] and Mallach [8] presented a better reordering method called “binary tree” hashing which leads to an average of roughly 2.13 probes for a retrieval from a full table. Lyon [9] proposed another reordering method using recursive entry displacements. Gonnet and Munro [7] and Rivest [10] discovered that the problem of reordering the table so as to minimize the average number of probes required for a retrieval is a special case of an assignment problem. Gonnet and Munro [7] reported that the experiment of the optimal reordering scheme produces an average of about 1.83 probes for a retrieval in a full table. Experimental results of several open addressing schemes are given in Table III.

Open addressing, even with optimal reordering, requires more retrieval probes than direct chaining, but does not need extra space as link fields. Computed chaining offers a compromise. It can provide a performance somewhat between direct chaining and open addressing with optimal ordering by using storage between that required by those two methods. As shown in Table III, when \( \alpha = 0.99 \) and \( s = 6 \), computed chaining provides better performance than open ad-

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( s = 1 )</th>
<th>( s = 2 )</th>
<th>( s = 3 )</th>
<th>( s = 4 )</th>
<th>( s = 5 )</th>
<th>( s = 6 )</th>
<th>( s = 7 )</th>
<th>( s = 8 )</th>
<th>( s = 9 )</th>
<th>( s = 10 )</th>
<th>( \minis(\alpha) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1.090</td>
<td>1.070</td>
<td>1.070</td>
<td>1.070</td>
<td>1.070</td>
<td>1.070</td>
<td>1.070</td>
<td>1.070</td>
<td>1.070</td>
<td>1.070</td>
<td>2</td>
</tr>
<tr>
<td>60</td>
<td>1.404</td>
<td>1.381</td>
<td>1.271</td>
<td>1.264</td>
<td>1.264</td>
<td>1.264</td>
<td>1.264</td>
<td>1.264</td>
<td>1.264</td>
<td>1.264</td>
<td>4</td>
</tr>
<tr>
<td>70</td>
<td>1.881</td>
<td>1.528</td>
<td>1.343</td>
<td>1.323</td>
<td>1.323</td>
<td>1.323</td>
<td>1.323</td>
<td>1.323</td>
<td>1.323</td>
<td>1.323</td>
<td>4</td>
</tr>
<tr>
<td>90</td>
<td>2.755</td>
<td>2.062</td>
<td>1.666</td>
<td>1.459</td>
<td>1.409</td>
<td>1.408</td>
<td>1.408</td>
<td>1.408</td>
<td>1.408</td>
<td>1.408</td>
<td>4</td>
</tr>
<tr>
<td>95</td>
<td>3.294</td>
<td>2.414</td>
<td>1.894</td>
<td>1.600</td>
<td>1.480</td>
<td>1.433</td>
<td>1.433</td>
<td>1.433</td>
<td>1.433</td>
<td>1.433</td>
<td>6</td>
</tr>
</tbody>
</table>
### TABLE II.—Distribution of Complete Probe Numbers for a 99% Packing Factor in a Table of Size 997

<table>
<thead>
<tr>
<th>Complete</th>
<th>Number of Bits to Store Link</th>
<th>Number of Items</th>
<th>Accumulative Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probe Numbers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>639</td>
<td>65</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>72</td>
<td>72</td>
</tr>
<tr>
<td>2 - 3</td>
<td>2</td>
<td>86</td>
<td>81</td>
</tr>
<tr>
<td>4 - 7</td>
<td>3</td>
<td>86</td>
<td>89</td>
</tr>
<tr>
<td>8 - 15</td>
<td>4</td>
<td>54</td>
<td>95</td>
</tr>
<tr>
<td>16 - 31</td>
<td>5</td>
<td>29</td>
<td>98</td>
</tr>
<tr>
<td>32 - 63</td>
<td>6</td>
<td>14</td>
<td>99.2</td>
</tr>
<tr>
<td>64 - 127</td>
<td>7</td>
<td>6</td>
<td>99.9</td>
</tr>
<tr>
<td>156</td>
<td>8</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>

### TABLE III.—Comparison of Mean Number of Probes for Successful Lookup (n = 997)

<table>
<thead>
<tr>
<th>α</th>
<th>Computed Chaining (s=6)</th>
<th>Computed* Chaining (s=10)</th>
<th>Direct Chaining (theoretical)</th>
<th>Pseudochaining (s=10)</th>
<th>Uniform Probing</th>
<th>Brent’s Method</th>
<th>Binary Tree</th>
<th>Lyon**</th>
<th>Optimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1.070</td>
<td>1.070</td>
<td>1.100</td>
<td>1.101</td>
<td>1.115</td>
<td>1.102</td>
<td>1.102</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>1.168</td>
<td>1.168</td>
<td>1.200</td>
<td>1.211</td>
<td>1.277</td>
<td>1.217</td>
<td>1.217</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>1.264</td>
<td>1.264</td>
<td>1.300</td>
<td>1.347</td>
<td>1.527</td>
<td>1.367</td>
<td>1.364</td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>1.323</td>
<td>1.323</td>
<td>1.350</td>
<td>1.438</td>
<td>1.720</td>
<td>1.444</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>1.356</td>
<td>1.356</td>
<td>1.400</td>
<td>1.563</td>
<td>2.011</td>
<td>1.599</td>
<td>1.579</td>
<td>1.49</td>
<td>1.489</td>
</tr>
<tr>
<td>90</td>
<td>1.408</td>
<td>1.408</td>
<td>1.450</td>
<td>1.787</td>
<td>2.558</td>
<td>1.802</td>
<td>1.751</td>
<td>1.63</td>
<td>1.610</td>
</tr>
<tr>
<td>95</td>
<td>1.433</td>
<td>1.433</td>
<td>1.475</td>
<td>2.023</td>
<td>3.153</td>
<td>1.972</td>
<td>1.880</td>
<td>1.72</td>
<td>1.689</td>
</tr>
<tr>
<td>99</td>
<td>1.601</td>
<td>1.449</td>
<td>1.495</td>
<td>2.649</td>
<td>4.651</td>
<td>2.242</td>
<td>2.049</td>
<td>1.785</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1.601</td>
<td>1.449</td>
<td>1.500</td>
<td>3.273</td>
<td>6.522</td>
<td>2.134</td>
<td></td>
<td></td>
<td>1.828</td>
</tr>
</tbody>
</table>

*These are less than the theoretical expected values (1 + α/2). The experiments by Lum, Yuen, and Dodd [11] indicate a similar performance of the linear quotient method.

**For n = 4999, based on I4(4), α = .97 average probes = 1.77
α = .98 average probes = 1.80

From the collection of the Computer History Museum (www.computerhistory.org)
dressing with optimal ordering (1.601 versus 1.785 probes) while using less space than direct chaining (6 bits versus 10 bits per link field). Note that open addressing with optimal ordering requires that the whole table be reconstructed for deletion or insertion of an item and thus is practical only when the table is static. The performance of computed chaining is therefore usually more attractive than open addressing with optimal ordering. Computed chaining is in addition better than Lyon's method of open addressing (1.601 versus 1.80+ probes) which is currently the best method of open addressing not requiring a complete restructuring of the table for an insertion.

The performance of pseudochaining based upon uniform probing [3] and the performance of uniform probing are shown in Table III. The pseudochaining technique can also be applied to improve other open addressing schemes. However, given the same link space, pseudochaining does not perform as well as computed chaining (2.65 versus 1.45 probes for \( \alpha = .99 \)) because the former uses link fields for first overflow items only. As a result, pseudochaining fails to perform the same as direct chaining or computed chaining even if it has sufficient link space to store "complete" probe numbers. Thus pseudochaining is less efficient when compared to computed chaining.

Another advantage of computed chaining, which it inherits from direct chaining, is that items can be deleted immediately without much difficulty. To delete an item from the table, e.g., at location \( r \), computed chaining will delete the items following in the same chain and reinsert them starting at location \( r \). In open addressing, however, moving another item in the table to location \( r \) may require reorganization of the table. A common solution in open addressing is to reserve a special key to denote a deleted item so that reorganization of the table can be postponed until a number of deleted items exist. However, the average number of probes required to retrieve non-deleted items will increase if the table contains many deleted items.

CONCLUSIONS

Computed chaining is a hybrid of several existing methods for collision resolution. It is similar to direct chaining in that all items with the same home address reside on the same probe chain. Computed chaining resembles open addressing since it uses a function of an item to locate a probe address. And it borrows from pseudochaining the use of a probe number in the link field instead of an actual address.

Practical suggestions on when to use computed chaining conclude the discussion. If sufficient bits in each link field are available to store an actual address, direct chaining remains the preferable method. However, if full-address bits are not available, computed chaining gives results only slightly degraded from those of direct chaining. If only a few bits are available for the link field, computed chaining may still be more efficient than open addressing techniques. From the experimental results (\( \alpha = .99 \)) with computed chaining, sixty-five percent of the items stored in the table required no probe links and ninety-five percent required four or fewer bits to store probe links.

REFERENCES