Some techniques for structuring chained hash tables

C. Bays

Department of Computer Science, University of South Carolina, Columbia, South Carolina 29208, USA

Existing techniques for structuring chained hash tables are mentioned and several new techniques are described. A comprehensive set of algorithms is given for effecting the new techniques and some applications are mentioned.

(Received April 1972)

1. Introduction and review of existing chaining techniques

Hash techniques fall roughly into two classes: open techniques (Bell and Kaman, 1970; Maurer, 1968) and chaining techniques. The purpose of this paper will be to discuss both new and existing chaining techniques, all of which involve the chaining together of entries which hash to the same location.

One of the most commonly used techniques to accomplish the chaining process is to create a hashed index table (Morris, 1968). This is usually done by computing a hash address for the key of a given entry and storing at this address a pointer to a chain of all entries whose keys hash to this same address. These pointers actually constitute the hash table, which thus contains no information other than where to find the starting entry for each chain. An actual entry is stored within an individual chain and contains, in addition to the key that was hashed, a link to the next entry in the chain. Frequently the only other information present in a chain entry is another pointer which gives the location of the rest of the information associated with that entry. This information, as well as the entries in the chains, may be stored anywhere in memory, with space being provided by any memory allocator. Thus, in addition to the hashed index table, the chained hash table will contain entries which are usually composed of

1. a key which is the value being searched for;
2. a pointer to the next entry in the chain;
3. a pointer to all other information associated with the entry.

It is possible to store the starting entry of each chain at the actual hash address for the entry, thereby eliminating the need for a separate index table. Moreover, additional entries in a given chain may also be stored within the table boundaries, thereby utilising memory which does not contain starting entries for chains and which would otherwise be unused. Unfortunately, this usually involves moving existing entries around as new entries are made, for if a new entry collides with an existing entry which is not at the head of its chain, the stored entry must be moved to make room for the new chain head and the chain associated with the stored entry must therefore be fixed up.

One way to avoid the necessity of moving existing entries around is to allocate space outside the table for any new entries that collide with existing ones. Hence, a new entry can either be stored within the original table area as the head of a new chain, or outside the table area as a part of an existing chain. When all entries have been made, those entries outside the table area may, if desired, be moved to empty locations within the table (Hopgood, 1969). The average space required outside the table area, assuming random distribution of entries, is shown by Morris (1968) to be $N - M(1 - \exp(-N/M))$ where M is the original table length and N is the total number of entries made.

At the expense of an additional pointer field and an additional look-up, all entries can be stored initially within the existing table boundaries without ever having to be moved. The second pointer field of a given entry stored at, say location X, gives the actual starting location for the chain of entries whose hash address is X. These entries can be stored in any location in the table (including location X). After the starting location for a chain is found, a search down the chain is accomplished by using the first pointer field. This method is a variant of the technique described by Johnson (1961) and is really just a type of scatter index table.

The remainder of this paper will deal with methods which have not been previously described, and will include a comprehensive set of algorithms for implementing the methods.

2. Definition of terms

For most of the discussion, we will use the following terms. Let K be a key that we are hashing, let HASH be the hashing function used to create an initial probe into the table. Let M be the capacity of the table, and let i and j represent variables which will be used as pointers to individual entries. For example, we may write $i \leftarrow \text{HASH}(K)$ and i will point to the initial hash address (hereafter called simply the hash address) of the entry associated with K and will be within the range $1 \leqslant i \leqslant M$. Let the entries themselves be broken into two parts. L will represent the link field which will give the address of the next entry in a chain, assuming there is one. Let INFO be all the rest of the entry, including the key. If the entry is structured into (1), (2), and (3), as described above, then INFO will represent parts (1) and (3), and L will be part (2). If all other information associated with the entry is to be stored with the entry thereby eliminating the necessity for part (3), then it can all be lumped together under the INFO field. Thus, for this discussion, an entry at location i will be composed entirely of fields INFO_i and L_i. (Later, in the discussion involving doubly linked lists, we will refer to the link fields LLINK(i) and RLINK(i) instead of the link field L_i).

3. Chaining a table by rehashing

During the creation of a table, if a point in time is reached after which we can say with certainty that no new chains will be created, then we can conveniently delay all chaining until this time. We can, with a slight modification to standard hashing techniques, treat the table as an open (non-chained) hash table, which eases greatly the chore of making new entries. After the aforementioned point is reached, the entries may be rehashed to form the required chained structure without having to break into and fix up chains.

Before describing the algorithms involved, some fairly obvious and very useful facts should be noted. If the hashing function does not change, then any space which is empty after the initial hash will be available for storing entries which are not at the heads of chains. No additional movement of these entries will be necessary, for nothing ever hashes directly to their locations. Moreover, if during the initial hash we could keep track of all locations which will never contain heads of chains, then these spaces become available for entries other than chain

Downloaded from https://academic.oup.com/comjnl/article/16/2/126/372604 by guest on 19 April 2024

heads. Of course, if such a location contains an entry then it must either be linked up properly or moved into a position as a chain head.

To indicate these special properties of locations, we can conveniently use the link field, since it will not be permanently filled until after the rehash. Moreover, since the link field will only contain values between 1 and M, then any other values we might assign can take on special meanings. This turns out to be a natural way of doing things. If n represents the number of bits required for the link field, then obviously the table size, M, must be less than or equal to 2^n. For the algorithms below, all that will be required is that $M + 3 < 2^n$. If standard hashing techniques are used (Maurer, 1968; Bell and Kaman, 1970), whereby a prime number is chosen for M, then the above relation will almost automatically follow. (Note that M is not absolutely required to be prime but that this need be done only if the particular hashing technique used required it.)

Thus, the following special value of the L field (**Fig. 1**) will be assigned during the initial hash, to be used later during the chaining process. Of course, after chaining is complete, only the proper values will be stored in the L fields—the temporary values will have been erased.

Value of link field	*Meaning*
0	this entry is stored at its proper hash address
$M + 1$	this location is empty
$M + 2$	this entry is not at its hash address, and some other entry has this location as a hash address
$M + 3$	this entry is not at its hash address, and no entry has this location as its hash address

Fig. 1. Possible values of the link field after the initial hash

Note that whether or not an entry is empty can be checked merely by seeing if the L field contains the value $M + 1$. Hence, a special value for the key does not need to be reserved to indicate an empty entry.

With the above values of L in mind, let us now list the steps of the initial hashing algorithm.

Algorithm A Initial hash
(Previously, all L_i have been set to $M + 1$)
1. $i \leftarrow$ HASH(K)
2. if $L_i = M + 1$ (if this location is empty) set $L_i \leftarrow 0$ go to step 7
3. if $L_i = M + 3$, set $L_i \leftarrow M + 2$
4. use any collision function to get a new value for i
5. if $L_i \neq M + 1$ go to step 4
6. $L_i \leftarrow M + 3$
7. make a new entry at i, exit the algorithm

The only steps in the algorithm that are not required by standard hashing techniques are steps 6, 3, and the second part of step 2. Moreover, each of these steps, when executed for a given entry, is done only once.

Table position	*Value of key*	*Value of link*	
1	201	0	
2	233	12	$(M + 2)$
3	203	0	
4	211	13	$(M + 3)$
5	202	13	$(M + 3)$
6	213	13	$(M + 3)$
7		11	$(M + 1)$
8	212	13	$(M + 3)$
9	223	13	$(M + 3)$
10		11	$(M + 1)$

Fig. 2

An example of the application of algorithm A is given below. The overall length of the table, M, is 10. For the hashing function, the last digit of the key gives the hash address. For the collision function, add 3 to the current address (Mod M). The INFO field will consist only of keys, which are entered in the following order.

203
213
223
233
201
211
202
212

When all items have been entered using algorithm A, the table will appear as shown in **Fig. 2.**

After the initial hash, algorithm B, which requires only one pass through the table, is used to complete the chaining operation.

Algorithm B Link up entries
1. $i \leftarrow 0$
2. $i \leftarrow i + 1$ IF $i > M$ exit the algorithm
3. if $L_i \leqslant M + 2$ go to step 2
4. $j \leftarrow$ HASH(K_i) (This must be the same hashing function used in step 1 of algorithm A.)
5. if $L_j \leqslant M$, $L_i \leftarrow L_j$, $L_j \leftarrow i$, go to step 2
6. swap INFO$_i$, INFO$_j$ set $L_j \leftarrow 0$, go to step 4

A study of algorithm B reveals that if an entry was stored at its hash address by the initial hash, it will be left alone during the rehash. Similarly any entry whose link field was $M + 3$ will, if possible, be left at its same location and will be linked up directly behind the head of the proper chain. The exception is if no chain head exists for a particular $M + 3$ entry when it is encountered. Then the $M + 3$ entry is moved into its rightful place as chain head, and the $M + 2$ entry, which must have been there and had to be removed, is now treated as an $M + 3$ entry. Notice that the algorithm doesn't really get into operation until it finds the first $M + 3$ entry. There must be at least one such entry; if not, then everything hashed initially to its proper hash address and no further action need be taken. Note further that all empty spaces will be left intact. In general, the algorithm will move an entry only if there is no alternative, and will do so only once.

When algorithm B is applied to the example in Fig. 2, the chaining is completed as is shown in **Fig. 3**. Notice that all non-empty locations have link fields less than or equal to M, as is expected.

Table position	*Value of key*	*Value of link*
1	201	4
2	202	8
3	203	9
4	211	0
5	233	0
6	213	5
7		11
8	212	0
9	223	6
10		11

Fig. 3

3.1 *Retrieval*

With the table now properly chained, algorithm C may be used

Downloaded from https://academic.oup.com/comjnl/article/16/2/126/372604 by guest on 19 April 2024

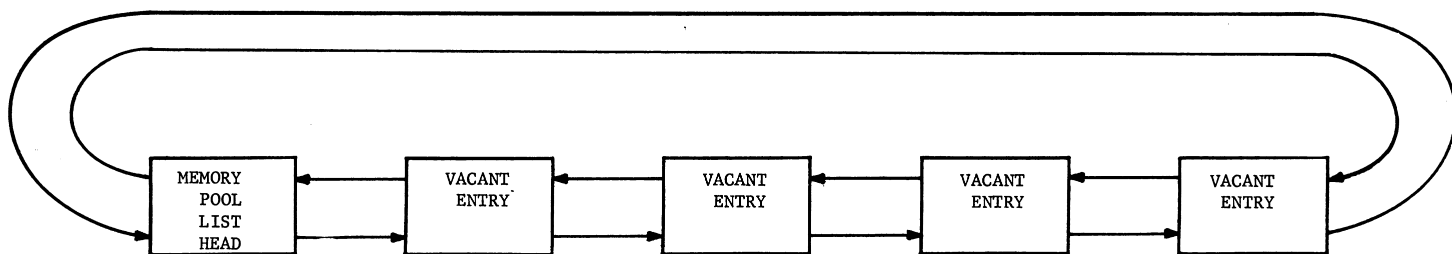


Fig. 4. A Doubly Linked Memory Pool

to search for a given key. This key is referred to in algorithm C as NEWKEY.

Algorithm C Find a key
1. $i \leftarrow$ HASH(NEWKEY)
2. if $L_i = M + 1$, NEWKEY is not in the table
3. if $K_i =$ NEWKEY, we found it
4. if $L_i = 0$, NEWKEY is not in the table
5. $i \leftarrow L_i$ go to step 3

4. Hash tables embedded in a linked environment

We have seen how to change an open hash table into a chained hash table. It is possible to embed any hash table, whether chained or unchained, into an area ordinarily reserved for linked structures. The only requirement is that the linked structures, or at least their memory pool, be doubly linked. Consider a memory pool structured as in Knuth (1968 page 253), except that it is doubly linked and contains a list head (page 278) which is not part of the pool. Such a pool is illustrated in **Fig. 4,** where arrows indicate links.

This can be more conveniently drawn without the arrows, if desired. Let each entry have the structure shown in **Fig. 5.**

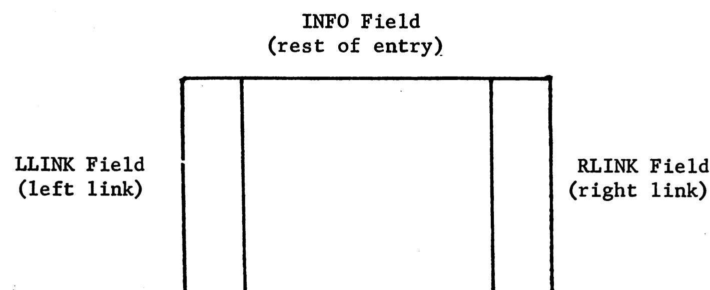


Fig. 5. A Memory Pool Entry

Then a memory pool consisting of locations A, B, C, and D is represented as illustrated in **Fig. 6.** (Of course, things do not need to be linked in the order shown.) Initially, before any memory from the pool is utilised, all entries will be empty except the link fields, LLINK and RLINK.

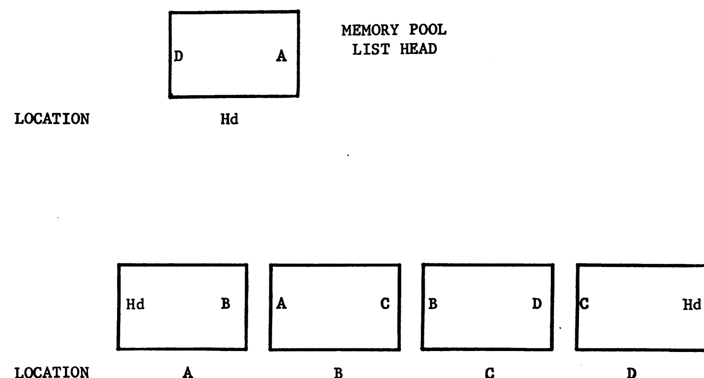


Fig. 6. Representation of a Doubly Linked Memory Pool

4.1 *Open* (*non-chained*) *hash tables*

If M sequential empty entries are set aside from the pool, then we can treat this area of M entries as the basis for a hash table. This area of the pool, however, is dedicated to the purpose of building the table only as long as the table is being built. After the hash table has been completed, the unused entries are returned to the memory pool.

When we make an entry into an open hash table, that is, one in which a searching method is used for resolving collisions, then as we store the entry we remove it from the memory pool by linking around it. Hence if a location, i, has just been allocated to an entry in an open hash table, we now remove entry i from the pool by the brief algorithm given below.

Algorithm D

$$\text{RLINK}(\text{LLINK}(i)) \leftarrow \text{RLINK}(i)$$
$$\text{LLINK}(\text{RLINK}(i)) \leftarrow \text{LLINK}(i)$$

Some important implications are apparent. An open hash table is thought of as having a 'load factor', α, which is given by

$$\alpha = \frac{\text{number of entries entered so far}}{\text{total number of spaces allocated for the table}}$$

As α approaches 1, the difficulty of finding empty spaces increases tremendously, as is shown in **Fig. 7.**

α(load factor)	E (average number of searches required to find a vacant location)
0·1	1·11
0·2	1·25
0·3	1·43
0·4	1·67
0·5	2·00
0·6	2·50
0·7	3·33
0·8	5·00
0·9	10·00

where $E = \frac{1}{1 - \alpha}$ (Bell and Kaman, 1970)

Fig. 7

Since the space remaining, $(1 - \alpha)$, represents wasted space, then it is usually desirable to have as high a value of α as possible without wasting too much time searching for entries. In fact, values of α near 0·8 usually indicate a 'full' hash table of the open type.

Notice, however, that if we are hashing onto a doubly linked memory pool as described above, and the actual proportion of the M spaces used (hence the load factor) is small, then no harm is done, for all unused space is returned to the pool. We have, in effect, created a hash table whose load factor is precisely 1 and whose value for E is as low as we wish, depending on how large we are able to make M. Moreover, since the table is an 'open' hash table, no space is wasted for the storage of link fields.

Downloaded from https://academic.oup.com/comjnl/article/16/2/126/372604 by guest on 19 April 2024

4.1.1 *Reallocation of non-related entries*

With a small additional step, we can allow M to cover the entire original storage pool area, even though the storage pool may be currently in use by other linked structures. This can be done by making the additional provision that any linked structure using the storage pool be a doubly-linked structure. Then, if when building a conventional hash table, an entry hashes (whether directly or after a number of collisions with prior entries) to a space already allocated to some entry in a linked structure other than the memory pool, the interfering linked entry can easily be moved to a vacant space in the memory pool. This is accomplished by moving the entry to the location currently at the right (or left) of the pool list head.

Let i be the location of the entry being moved. Then the algorithm to move the linked entry to the right of the pool list head and preserve its connection with the interfering structure is given by algorithm E

Algorithm E Move an entry
(Hd is the location of the list head)

1. if RLINK(Hd) = Hd then the memory pool is empty, we cannot proceed
2. $j \leftarrow$ RLINK(Hd)
3. RLINK(LLINK(i)) $\leftarrow j$
4. LLINK(RLINK(i)) $\leftarrow j$
5. INFO(j) $\leftarrow$ INFO(i)
6. LLINK(RLINK(j)) $\leftarrow$ Hd
7. RLINK(Hd) $\leftarrow$ RLINK(j)
8. RLINK(j) $\leftarrow$ RLINK(i)
9. LLINK(j) $\leftarrow$ LLINK(i)

now, we simply insert the hash table entry at location i, which has been released from the interfering structure.

Thus, open hashing operations and linked allocation operations may commence simultaneously. Of course, a special tag must be present to indicate whether a given entry belongs to the hash table, since to move such an entry would be fatal to the structure of the hash table.

4.2 *Chained hash tables in a storage pool environment*

Clearly, the interfering doubly-linked entries mentioned in section 4.1.1. may be part of any doubly-linked structure. There is no reason why one of the doubly-linked structures cannot be composed of entries in the chained hash table itself. If this is the case, then the only entries in the hash table which are tagged (and thus are not movable) are those actually stored at their hash addresses, i.e. chain heads. A new entry colliding with an existing entry stored at its hash address is linked up with it and stored in the location to the right (or left) of the head of the storage pool and removed from the pool. If a new entry collides with an entry that is not at its hash address, or with an entry not part of the hash table at all, then the existing entry is moved by algorithm E to the location that is currently at the right of the pool list head, and the new entry is inserted in its place.

Linking up new entries in the hash table is accomplished by the two algorithms given below. Of course, each hash table entry is equipped with LLINK and RLINK fields.

To initialise the link fields of a newly created chain head stored at location i, we use algorithm F.

Algorithm F

LLINK(i) $\leftarrow i$
RLINK(i) $\leftarrow i$

Then later, to link up a new entry at location j to the right of the chain head already stored in location i, use algorithm G.

Algorithm G

LLINK(j) $\leftarrow i$
RLINK(j) $\leftarrow$ RLINK(i)
LLINK(RLINK(i)) $\leftarrow j$
RLINK(i) $\leftarrow j$

Searching for a key can be accomplished by algorithm H, which is very similar to algorithm C. Again, let NEWKEY be the key being searched for.

Algorithm H Search to the right

1. $i \leftarrow$ HASH(NEWKEY)
2. If the entry at location i is not a chain head, NEWKEY is not in the table.
3. $j \leftarrow i$
4. if the key at location i = NEWKEY, we found it
5. if RLINK(i) = j, NEWKEY is not in the table
6. $i \leftarrow$ RLINK(i) go to step 4

To delete a hash table entry stored at location i, we must first make certain that it is not at the head of a chain. If it is not, we simply use algorithm D. If location i has been tagged as a chain head, algorithm I must be used.

Algorithm I Delete a chain head

1. if RLINK(i) = i (if this is the only entry in the chain) remove the chain head tag from the entry at i, exit the algorithm
2. $j \leftarrow i, i \leftarrow$ RLINK(i) (i now points to the entry at the right of the chain head. This entry will become the new chain head at j, leaving i pointing to a vacated entry.)
3. INFO(j) $\leftarrow$ INFO(i) (move the entry) retain the chain head tag at j
4. RLINK(j) $\leftarrow$ RLINK(i)
5. LLINK(RLINK(i)) $\leftarrow j$

Whether or not the entry was at the head of a chain, it has now been removed from the hash table and may be returned to the memory pool by algorithm J, which will insert it to the right of the pool list head. (Note that algorithms G and J are actually identical.)

Algorithm J Return vacated entry to memory pool

LLINK(i) $\leftarrow$ Hd
RLINK(i) $\leftarrow$ RLINK(Hd)
LLINK(RLINK(Hd)) $\leftarrow i$
RLINK(Hd) $\leftarrow i$

Notice that if we change LLINK to RLINK and RLINK to LLINK in algorithms D through J, then the algorithms still work—except they are 'left handed'. Thus, doubly linked lists are virtually 'ambidextrous'.

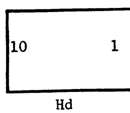
An example of a chained hash table competing for space in a storage pool is given in **Figs. 8 and 9.** Entries 3, 8, and 9 are already allocated to a nonrelated structure, with a separate list head at location X before the hashing commences (Fig. 8). If a chained hash table is interleaved over the storage pool area, the result would be as shown in Fig. 9. Again, the hash address is given by the last digit of the key. The keys read in were (in this order) 106, 108, and 208. The algorithms used to make the entries are given in Fig. 9 next to the appropriate key.

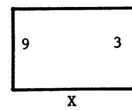
In Figs. 8 and 9 an indicator, 'P', is used to differentiate between memory pool entries and other linked entries. Entries not tagged with a P have been removed from the memory pool and are either part of the hash table or part of the structure whose list head is at X. Of course, heads of chains in the hash table are always differentiated from other entries, as they must be; in this case, by the presence of a 'C'.

5. Further considerations

Before deciding whether to use a chained or open hash table, one must consider the tradeoff due to the storage of links. The advantage of a chained technique over an open technique

Downloaded from https://academic.oup.com/comjnl/article/16/2/126/372604 by guest on 19 April 2024





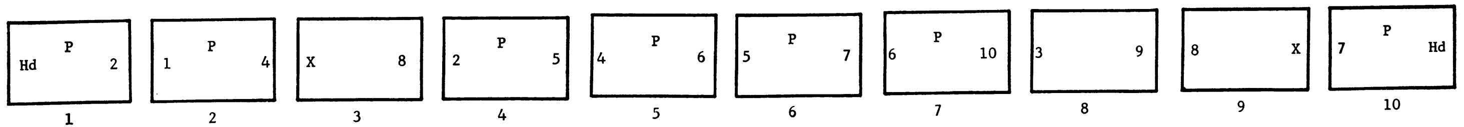
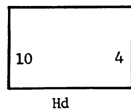
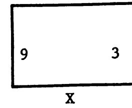


Fig. 8. A doubly linked structure. Hd is the list head of the memory pool. 'P' indicates an entry that is still in the pool and hence is available. Entries 3, 8, 9 belong to a list whose head is at *X* and have been removed from the pool.





106 (D,F)
108 (E,F)
208 (D,G)

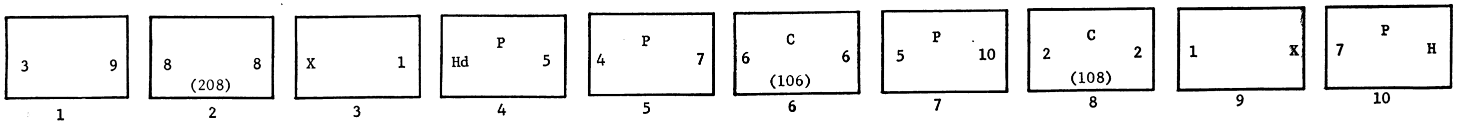


Fig. 9. Entries whose keys are 106, 108, and 208 have been entered into the structure. The algorithms used have been bracketed next to the keys. Heads of chains within the hash table are indicated by a 'C'.

in terms of the average number of searches, A, required to retrieve an entry, is given by

$$\frac{-\log(1-\alpha)}{\alpha} - 1 - \frac{\alpha\left(1 + \frac{p}{q-p}\right)}{2}$$

(Bays, 1973) where α is the load factor if the table were 'open', p is the number of bits devoted to links and q is the total number of bits in the entry. By plotting A against α, for various values of $\frac{p}{q-p}$, it is seen that whenever the number of bits used for links is greater than 2/3 times the number used for other purposes, the open hash is preferable. A reasonable and effective ratio to justify the use of a chained hash table over an open one would be when 8-9 bits (1-1½ bytes on many computers) are used for linking, for every 5-6 bytes of information. Of course, this is of importance only when there is a choice between whether a single table should be open or chained. Note that on a word structured computer, we must consider word boundaries. If a 48-bit word will contain only 36 bits of information, then the remaining 12-bits may be used for linking purposes at no cost, i.e. $\frac{p}{q-p} = 0$. If, on the other hand, a 36-bit word used all 36-bits for information, then to add a link field, we most likely have to waste another word, and $\frac{p}{p-q} = 1$. This is, of course, greater than the critical value of 2/3.

When interfering chained structures are present, the problem becomes more complex. If a hash table is being created over a linked area and there is a choice, the open hash should probably be used. This is due to the extreme efficiency of the open table that is attainable, as explained in 4.1. In most cases, the average number of searches would approach 1, hence the storage of link fields would simply waste space without increasing efficiency.

Note furthermore that algorithms such as D and E involve mainly the alteration of addresses, which on many computers can be accomplished more efficiently than the movement of stored information, particularly when the address fields involved are significantly shorter than the information fields.

5.1 *Some applications*

The technique described in 3 lends itself well to use within compilers or assemblers which reside in core and where translation speed is of prime concern. Thus, symbol tables may be built as open hash tables, and later utilised as chained hash tables. Of course, the tradeoff between chained and open methods must be kept in mind. In fact, for a given compilation, one can easily determine whether or not to apply algorithm B. To do this, we must take into account the time required by algorithm B in addition to the advantage of chained versus open methods. In terms of the number of searches per entry, algorithm B requires roughly $1/\alpha$ accesses, since it makes essentially a linear sweep through the table. One must also realise that during the subsequent phases of compilation, each symbol can be referred to an average number of β times. (Unfortunately β will vary from programmer to programmer and from language to language. Nevertheless, an estimate may be made by looking at past assemblies or compilations. A reasonable value for IBM 360 assemblies appears to be about 2.) Thus, the overall advantage becomes $\beta A - \frac{1}{\alpha}$ or, we decide to switch to a chained table if

$$-\beta\frac{\log(1-\alpha)}{\alpha} - \beta - \frac{\alpha\beta}{2}\left(1 + \frac{p}{q-p}\right) - \frac{1}{\alpha} > 0$$

we must also make sure that chaining does not cause the boundaries of the table to be exceeded, hence, if $\alpha\left(1 + \frac{p}{q-p}\right) > 1$, we cannot chain.

Downloaded from https://academic.oup.com/comjnl/article/16/2/126/372604 by guest on 19 April 2024

Since we know $\frac{p}{q - p}$ and can estimate β, a decision may be made that is based entirely on the value of α, the load factor of the open table. Thus, each assembler or compiler may have critical load factors, α_L and α_u, which may be computed and stored as constants, and the decision to chain may be made when $\alpha_L < \alpha < \alpha_u$. (Note that it is possible to find that $\alpha_L \geqslant \alpha_u$ in which case chaining would never be profitable.) For most practical cases, we would consider utilising this technique when we had free bits available for the link field within each symbol table entry, that is,

$$\frac{p}{q - p} = 0.$$

Here we would chain whenever α_L becomes greater than the critical value shown in **Fig. 10** (α_u is, of course, 1·0).

β	α_L
1·0	·90
1·5	·85
2·0	·80
2·5	·77
3·0	·74
4·0	·70

Fig. 10

Any applications which require the use of both open hash tables and list structures can be optimally structured using the techniques described in 4.1. Such applications include string processors and multilist or ring-structured information retrieval systems. Similar applications are suggested which employ the techniques described in 4.2. In fact, the author is aware of an efficient sorting algorithm that utilises both calculated address tables and tree structures, which are embedded directly onto each other in the sort area. In this technique the tree roots form the head nodes in a chained hash table, and the other entries in the trees are considered as further entries within the hash table. Thus, all entries hashing to the same location are grouped together in a sorted tree whose non-root entries may be moved around to make room for newly created roots. Insertion is somewhat more complicated than algorithm G, since it involves searching down a tree, but otherwise, the techniques given in 4.2 are employed. The utility of this sort is that for n random entries, the sort time is linearly proportional to n, even when the entries (and their links) completely fill storage.

Only a few applications have been mentioned here, but it is obvious that useful and innovative processes can be developed by carefully utilising the techniques described within this paper.

Downloaded from https://academic.oup.com/comjnl/article/16/2/126/372604 by guest on 19 April 2024

References

BAYS, C. (1973). A Note on When to Chain Overflow Items within a Direct-access Table. *CACM*, Vol. 16, No. 1, p. 47.
BELL, J. R. and KAMAN, C. H. (1970). The Linear Quotient Hash Code. *CACM*, Vol. 13, No. 11, pp. 675-677.
BLACKE, L. F., LAWSON, R. E., YUILLE, I. M. (1970). A Ring Processing Package for use with FORTRAN or a Similar High-Level Language. *The Computer Journal*, Vol. 13, No. 1, pp. 40-47.
DODD, G. G. (1969). Elements of Data Management Systems. *Computing Surveys*, Vol. 1, No. 2.
HIGGINS, L. D. and SMITH, F. J. (1971). Disc Access Algorithms. *The Computer Journal*, Vol. 14, No. 3, pp. 249-253.
HOPGOOD, F. R. A. (1969). *Compiling Techniques*. American Elsevier Publishing Company, New York.
JOHNSON, L. R. (1961). An Indirect Chaining Method for Addressing on Secondary Keys. *CACM*, Vol. 5, No. 8, pp. 218-222.
KNUTH, D. E. (1968). *The Art of Computer Programming* (Volume 1 *Fundamental Algorithms*). Addison-Wesley, Reading, Massachusetts.
MAURER, W. D. (1968). An Improved Hash Code for Scatter Storage. *CACM*, Vol. 11, No. 1, pp. 35-36.
MCILROY, M. D. (1963). A Variant Method of File Searching. *CACM*, Vol. 6, No. 1, p. 101.
MORRIS, R. (1968). Scatter Storage Techniques. *CACM*, Vol. 11, No. 1, pp. 38-42.

Book review

Introduction to Computers and Computer Programming, by Samuel Bergman and Steven Bruckner, 1972; 433 pages. (*Addison-Wesley*, £4·90)

This book was developed from the notes for a series of introductory programming courses given at the University of Pennsylvania and Temple University. It is unlikely that this book will be of great interest to British universities as it is too long for an introductory text and has insufficient depth for a computer science course.

The first eight chapters, about 230 pages, are devoted to teaching the basic principles of machine code and assembly language by using an imaginary computer FACET (FAcility for Computer Education and Training). This is a small (1K) simulated decimal machine with a single accumulator and index register and instructions for handling integers, characters and floating point numbers. The instruction set is well constructed and a variety of basic concepts are skilfully illustrated within the limited framework chosen.

A chapter entitled 'A Look at Other Computers' links the FACET part of the book to the section dealing with FORTRAN. This deals with the representation of numbers (particularly binary), multiple register machines, paging and peripherals.

The authors' foreword states that 'the FORTRAN presentation is more thorough than the standard approach because students . . . can already program'. I found their 'thorough' approach unpalatable and lacking a sense of purpose. Instructions were introduced in quick succession without any thought to orderly development and illustrated by quoting comparable sequences of FACET assembly language instructions.

The general presentation of the book is good, each chapter containing a review section emphasising the main points of the chapter. A large number of problems are provided ranging from well disguised trivia to several problems which could (and did!) prove difficult to an experienced programmer. Appendices cover the FACET assembly language and control cards, FORTRAN syntax and FORMAT examples and debugging, which draws together ideas from various chapters.

The FACET simulator (written in FORTRAN) and an instructors' manual are available from the authors. However, I was unable to obtain these in time to include comments on them in this review.

R. C. WELLAND (Reading)