ABSTRACT

MATHEMATICS AND COMPUTER SCIENCE

BENJAMIN KAI-HSIEN HSU   B.S., CHUNG-YUAN UNIVERSITY, 1982

Data Structures for Set Manipulation - Hash Table Method

Advisor: Dr. Nazir A. Warsi

Thesis dated May 1986

The most important issue addressed in this thesis is the efficient implementation of hash table methods. There are credential trade-offs in a desired implemention. These are discussed in issues such as hash addressing, handling collision, hash table layout, and bucket overflow problems. The criteria of good hash function is providing even distribution.

Collision is the major problem in hash table methods. Two major hashtable methods are discussed. Open Addressing Method places the synonymous items somewhere within the table. The Chaining Method, however, chains all synonymies and stores them somewhere outside the table called overflow area.

Hash table is widely used by system software as an ideal data structure. Hash Table applications canbe found in compiler's symbol table, database, directories of file organizations, as well as in problem-solving application programs.

DATA STRUCTURES FOR SET MANIPULATION

-- HASH TABLE METHOD --

A THESIS

SUBMITTED TO THE FACULTY OF ATLANTA UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCES

BY

BENJAMIN KAI-HSIEN HSU

DEPARTMENT OF MATHEMATICAL AND COMPUTER SCIENCES

ATLANTA, GEORGIA

MAY 1986

R= vi T=46

To

my mother,

Fong-Eng Wang Hsu

whose unconditional love and

support are the real motives in continuing

my study.

ACKNOWLEDGEMENTS

The author expresses his sincere appreciation and deep gratitude to those people who have contributed their time and energy to make this study possible.

Thanks are especially due Dr. Nazir A. Warsi, my thesis advisor, for his instruction and suggestions, which were essential to the completion of this study.

Special thanks are also due Dr. Negash Medhin, Professor Hsu and Professor Kao, faculty members of the Mathematical and Computer Sciences department, for their advice and correction.

Benjamin Kai-Hsien Hsu

TABLE OF CONTENTS

		Page
DEDICATION		ii
ACKNOWLEDGEMENTS		iii
TABLE OF CONTENTS		iv
TABLES AND FIGURES		vi
CHAPTER		
I	INTRODUCTION	1
II	INTRODUCTION TO SETS	
	Constituents of Set	4
	Set Operations	4
	Data Types	5
	Data Structures	6
	Definition of Abstract Data Type on Sets	8
III	HASH FUNCTION	10
IV	METHODS OF RESOLVING COLLISIONS	
	Open Addressing Methods	
	Random Probing	14
	Linear Probing	16
	Deletion of Open Addressing Method	17
	Restructuring the table	17
	Chaining Methods	
	Indirect Chaining	19
	Direct Chaining	20

V COMPARISON OF THE METHODS

General Considerations 30

Comparison of Hash Methods with the

Other Strategies 31

VI APPLICATIONS OF HASH TABLE METHODS IN COMPUTER SOFTWARE

File Directories 36

Demand Paging Memory Management 37

Symbol Table in Compiler 38

Database Management 40

VII SUMMARY OF STUDY 43

BIBLIOGRAPHY 45

TABLES AND FIGURES

Tables: Page

1. Number of Probes Required on Looking up a random item 30

2. Comparison of Internal Table Methods 32

Figures:

1. Flow chart of Chained Scatter Table for Search and Insert 23

2. A Hash Table 27

3. The Hash Table after Insertion and Deletion .. 27

4-1 Data Structure of the symbol table 39

4-2 The Hash Chains of symbol table 39

CHAPTER I

Introduction

Background of the Problem. One of the most actively pursued areas of computer science over the years has been the development of new and better algorithms for the performance of set manipulation problems.

A good way to approach the design of an efficient algorithm for a given problem is to examine the fundamental nature of the problem.

Sets, as the most concept in mathematics, have profound problems associated with them. Often, certain type of set manipulation problems can be formulated in terms of abstract data types with a collection of operations on them. These data types can be outlined in various data structures, they will be discussed in the next chapter.

Among many data structures of set's manipulation problems, the hash table will be the central issue of this thesis. This thesis gives a detailed study of various hash table methods; especially, it is intended for studies of the better implementation of hash tables and the solution for the difficulties in the implementation of hash table methods.

About the Hash Table Methods Hash table methods seek to eliminate all search time of data retrival. The idea behind hash table methods is quite simple: Even though the keys may represent symbolic strings or some other set of

values, in reality, all keys are represented in a computer by an integer value --by a sequence of bits.

All hashing methods involve a hash code or hashing function or mapping function. If K is an arbitrary key, then h(K) is an address. Specifically, h(K) is the address of some position in a table known as a hash table or scatter storage table, at which we intend to store the record whose key is K. If we can do this, then if at some later time we want to search for the record whose key is K, all we have to do is to calculate h(K) again. This is what makes hashing methods so popular. Most of the time, we can find the record we want immediately, without any repeated comparison with other items.

The phenomenon of two records having the same home address is called collision, and the records involved are often called synonyms. The possibility of collision, although slight, is the chief problem with hash table methods.

An overflow is said to occur when a new key is mapped or hashed into a full bucket. For the sake of speed, we would like to make bucket table rather large. However, when bucket is large, many of the lists will be empty and much of the space for the bucket list heads will be wasted.

Comparative studies of different hash table methods are discussed in this thesis. The trade off of collision resolution and retrieval time, and space consumption is also

studied in details.

CHAPTER II

Introduction to Sets

The notion of a set is basic to all mathematics. In algorithm design, sets are used as the basis of many important abstract data types, and many techniques have been developed for implementing set-based abstract data types.

2-1. Constituents of set

Sets, as the most concept in mathematics, have profound problems associated with them. When considering operations for which set members and sets are operands, it is desirable to introduce the concept of type. In building up sets, possible constituents are[1]:

* Atomic types, including integers, reals, characters, strings, and Boolean value.
* Sets constructed on atomic types.
* Tuples (ordered lists of atomic types).
* References to sets or tuples in the form of literals (labels or addresses) and variables (identifiers and pointers whose domains are atomic elements).

Most implementations of sets allow several of these constituents.

2-2. Set Operations

We consider here data structures subject to the following operations:

member(x,A) Takes set A and object x,whose type is the type of elements of A, and returns a boolean value, true if x belongs to A (successful search). and false if x does not belong to A (unsuccessful search).

insert(x,A): Makes x a member of A. That is, the new value of A is $A \cup \{x\}$. Note that if x is already a member of A, then insert(x,A) does not change A.

delete(x,A): Removes x from A. A is replaced by $A-\{x\}$. If x is not in A originally, delete(x,A) does not change A. Keys are accessed either by value or by position,and additional constraints may be imposed on the set of keys accessible at each stage.

2-3. Data Types

A data type is a specification of the basic operations allowed together with its set of possible restrictions. The four data types to be studied here are:

Dictionary - Keys belonging to a totally ordered set are accessed by value; all three operations are allowed without any restriction.

Priority queue - Keys belonging to a totally ordered set are accessed by value. The basic operations are insertion and deletion. Deletion is performed only on the key of minimal value (of "highest

priority").

Linear list - Keys are accessed by position; operations are insertion and deletion without access restrictions.

Stack Keys are accessed by position; operations are insertion and deletion but are restricted to operate on the key positioned first in the structure (the "top" of the stack).

2-4. Data Structures

A data organization is a machine implementation of a data type. It consists of a data structure, which specifies the way objects are internally represented in the machine, together with a collection of algorithms implementing the operations of the data type.

In Flajolet and Francons' paper [2], they discussed the relative data structures for five major data type of sets :

Stacks: They are almost universally represented by arrays, or linked lists.

Dictionaries: The most straightforward implementation is by sorted or unsorted lists; binary search trees have a faster execution time and several balancing schemes have been proposed: AVL and 2-3 trees; bichromatic trees. Other alternatives are h-tables and digital trees.

Priority queues: They can be represented by any of the search trees used for dictionaries; more interesting are heaps, P-tournaments, binomial tournaments,

binary tournaments and pagodas. One canalso use sorted lists, and any of the balanced tree structures for implementing priority queues.

Linear lists: The most straightforward implementation is by linked lists and arrays. Position tournaments are more efficient implementation to which balancing schemes can be applied.

Hash tables: These are special cases of dictionaries. All the known implementations of dictionaries are applicable here.

Of course there are other interesting data types: queues and dequeues are closely related to stacks; partition structures involve the operation of union, which is not considered here. One could also allow for more operations: split and merge for dictionaries; extract and union for priority queues; search, cut, concatenate and reverse for linear lists.

Now we need to state precise definitions concerning sequences of operations for each of our basic data types. A data type can be formally described by the universe of keys, the set of files, and the specification of the way operations perform on files.

a. The universe U from which keys are drawn is the set of real numbers (in practice U is more likely to be some very large but finite set).

b. A file status, or simply file, for a given data type is a structured finite set of keys. For dictionaries

and priority queues, the set of files is the set of all finite subsets of U (i.e.; a file can be any finite set of keys). For linear lists, stacks and symbol tables, the set of files is the set of all sequences on U.

c. For each input k, operation O, and file F, we need to describe in each case the way F is transformed when operation O belong to { deletion, insertion, successful search, negative search.} is performed on key k: (let I =: insertion, D =: deletion, S+ =: successful search, S- =: negative search.)

2-5. Definition of Abstract Data Type on Sets [2]

Stack --If F = <kl,k2,...,ks>, performing I(k) leads to <kl,k2 ,....ks,k>; performing D(k) leads to <kl,k2,....,ks-1> with output ks,provided s >= 1.

Dictionary --If F ={kl,k2,...,kd}, performing O(k) leads to a new file F' with F' = F, if O = S+ and k belong to F or O = S- and k not belong to F; F' = F U {k} , if O = I and k not belong to F; and F' = F - {k}, if O = D and k belong to F.

Priority queue --With F = {kl,k2,...,kp}, I(k) with k not belong to F leads to F'= F U {k}; suppression D leads to F'= F - {a}, where a = min { kl,k2,. ,..,kp }, and is meaningless if p = 0.

linear list --With F a sequence of keys <kl,k2,...kl>, I(p;k) is defined iff 1 <= p <= l+1 and the resulting

file is F'= <kl,...,kp-l,k,kp,....,kl>; on the other hand D(p) leads to F'= <kl,...kp-l,kp+l,...,kl>.

Hash tables --With F = <kl,k2,...km>, performing O(k) leads to a new file F' such that F' = F, if O = S+ and k belong to F; F' = <kl,....,km,k>, if O = I and k belong to F; and F' = <kl,....,km-l>, if O = D.

A sequence of operations is a sequence of the form Ol(kl); O2(k2); .....On(kn), where for l <= i <= n, ki belong to K is a key and Oi belong to O = { D,I,S+,S- } is an operation.

CHAPTER III

Hash Function

Hashing schemes perform an identifier transformation through the use of a hash function f. It is desirable to choose a function f which is easily computed and also minimizes the number of collisions.

3.1 Hashing Functions

A hashing function, f, transforms a key x into a bucket address in the hash table. A good hash function should satisfy two requirements:

a. Its computation should be very fast.

b. It should minimize collisions.

3.2 Uniform Hash Functions

If x is a key chosen at random from the key space, then we want the probability that f(x) = i to be 1/M for all buckets i, Then a random x has an equal chance of hashing into any of the M buckets. A hash function satisfying this property will be termed a uniform hash function.

Several kinds of uniform hash functions are in use. We describe four of these.[14]

1. Mid-Square

It is one hash function that has found much use in symbol table applications. This function, f, is computed by squaring the identifier and then using an appropriate

number of bits from the middle of the square to obtain the bucket address; the identifier is assumed to fit into one computer word. Since the middle bits of the square will usually depend on all of the characters in the identifier, it is expected that different identifiers would result in different hash addresses with highprobability even when some of the characters are the same.

2. Division

This simple choice for a hash function is obtained by using the modulo (mod) operator. The key x is divided by some number M and the remainder is used as the hash address for x. That is, f(x) = x mod M. This gives bucket addresses in the range 0 - (M-1) and so the hash table is at least of size M.

3. Folding

In this method the identifier x is partitioned into several parts, all but the last being of the same length. There are two ways of carrying out this addition. Shift folding and folding at the boundaries.

4. Digit Analysis

This method is particularly useful in the case of a static file where all the identifiers in the table are known in advance. Each identifier x is interpreted as a number using some radix r. The same radix is used for all the identifiers in the table. Using this radix, the

digits of each identifier are examined. Digits having the most skewed distributions are deleted. Enough digits are deleted so that the numbe of digits left is small enough to have an address in the range of the hash table.

CHAPTER IV

Method of Resolving Collisions

In this chapter, all the analysis assume that a hash function distributes elements uniformly over the buckets. Many methods of resolving collisions will be suggested and used. Among all, a particular method to be used in a particular application should be chosen carefully since the method of handling collisions profoundly affects the efficiency of the technique and the difficulty of the programming task.

4-1 Two Methods

There are two major foumulations of hash table storage and retrieval algorithms, differing in the manner in which collisions are resolved. The first method is to establish a hash table for the storage of items, and to resolve collisions by somehow finding an unoccupied space for those items whose natural home locations is already full, in such a way that the item can be later retrieved without the use of auxilliary link fields. Algorithms which use such schemes are called Open Addressing Algorithms. The second approach finesses the problem of collisions by using indirect addressing to allow all items which collide to maintain a claim to their home location. Such methods of handling collisions are commonly called chaining methods.

4-2 Open Addressing

If we try to place k in bucket h(K) and find it already holds an element, the rehash strategy chooses a sequence of alternative locations, h1(K), h2(K), --within the bucket table,in which we could place k. We try each of these locations in order or random, until we find an empty one. If none is empty then the table is full and we cannot insert k. This method to handle collisions is as follows:

1. Calculate address x in the table by using some transformation on the key as an index.
2. If the item is already at this address or if the place is empty the job is done.
3. If some other key is there,call a rehash function for an integer offset p. Make the next probe at i+p and go to step 2.

4-2.1 Random Probing

Used by a pseudorandom number generator. The pseudorandom number generator can be of the simplest sort and usually can be written in less than six machine instructions, It must generate every integer from 1 to n-1 (where n is the size of the table) exactly once. When the generator run out of integers, the table is full and the entry cannot be made.

The important property of the pseudorandom number generator in this application is that for every value of i, the numbers, pi+k - pi for i <= i+k <= n-1, are all different, where pj is the jth random number which is

generated.[11]

ANALYSIS:

The efficiency of this method is best expressed in terms of the average number E of probes necessary to retrieve an item in the table. We note that the number of probes required to lookup an item is exactly the same as the number of probes required to insert the item into the table in the first place. So let us calculate how many probes are required to insert a new item when there are already k items in the table. This will give a result A(k), and to find E we will the need to sum A(k) from 0 to k-1 and divide by k to find the average.

With a hash table of k entries in bucket and consider inserting the (k+1)th item into the table: [4]

A(k) : is the expected value of $L = \sum_{j=1}^{k+1} j * Pr(L=j)$

Now Pr(L=j) = Pr(L >= j) - Pr(L >= j+1)

and Pr(L>=1) = 1

Pr(L>=2) = probability that have collision on first rehash.

= k/N.

Pr(L >= 3)

= probability that collision on first and second

= k/N * (k-1)/(N-1) by independence

.....

Pr(L>=k+1) = (k(k-1)...1) / (N(n-1)...(n-k+1))

Pr(L>=k+2) = 0 because must have made it by this point.

Hence,

$$A(k) = \sum_{j=1}^{k+1} j[\, Pr(L>=j)-Pr(L>=j+1)\,] = \sum_{j=1}^{k+1} Pr(L>=j)$$

$$= 1 + k/N + k(k-1)/N(N-1) + \dots\dots\dots$$

$$+ (k*(k-1)(k-2)\dots\dots\dots 1) / (N*(N-1)*(N-2)\dots(N-k+1))$$

$$= 1 + k / (N-k+1) \quad \text{by induction on } k$$

$$= 1 / (1-(k/(N+1))).$$

Note that induction is a little tricky: write N=M+k, then fix M, and then make induction on k. It then goes through quite readily.

Now $E = 1/K \sum_{k=0}^{K-1} A(k)$

$$= 1/K \sum_{k=0}^{K} 1 / 1- (k/N+1)$$

$$<= 1/K \int_0^K 1 / 1-(k/N+1)\, dk = N/K \int_0^{K/N} 1 / 1-(k/N+1)\, d(k/N)$$

$$<= 1/\alpha \int_0^{\alpha} 1 / (1-x)\, dx \quad \text{writing } x = k/N, \ \alpha = K/N,$$

and using $k/N+1 < x < 1$

$$= - 1/\alpha \ \log_e (1- \alpha)$$

4-2.2 Linear Probing Method

Upon collision, search forward from the nominal position (the initial calculated address), until either the desired entry is found or an empty space is encountered --searching circularly past the end of the table to the begining, if necessary. If an empty space is encountered, that space becomes the home for the new entry.

The disadvantage of this method is that after a few collisions have been resolved in this way, the entries are clumped in such a way that, given that a collision has just occurred at location i, the probability of a collision at

location i+1 is higher than the average probability over the whole table.

The efficiency of the linear probing method can be analyzed by techniques similar to random probing method. The result is that, to within suitable approximation, the average number E of probes necessary to look up an item in the table is [11] :

$$E = (1-\alpha/2)/(1-\alpha). \qquad \alpha = \text{load factor}$$

4-2.3 Deletion of Open Addresses Method

Deletion of entries made using this scheme is a troublesome process. One cannot simply mark an entry as empty in order to delete it because other entries may have collided at that place and they would become unreachable. The hash addresses for every entry in the table would have to be recomputed and some of them moved in order to close up the gap caused by the deleted entry. A much more convenient method of deletion is to reserve a special signal for a deleted entry. On searching for a key, the search continues if a deleted entry is encountered. A new item can be installed in place of any deleted entry encountered in searching for its proper place.

The disadvantage of this method is that the lookup time is not reduced when entries are deleted --only the lost space is reclaimed.

4-2.4 Restructuring the Table

One important basic property of hash table open addressing method is that they start working very badly when the hash

table becomes almost full. W. D. Maurer and T. G. Lewis [10] studied in the extreme case in which the table is completely full except for one space, and the linear method of handling collisions is used, a search for this space takes, on the average, N/2 steps, for a table of size N.

In practice, a hash table should never be allowed to get that full. The ratio of the number of spaces for such entries is the loading factor of the hash table; it ranges between 0 and 1. When the load factor is about 0.7 or 0.8, in other words, when the table is about 70% or 80% full, the size of the hash table should be increased, and the records in the table should be rehashed.Replacing table size M by dM; suitable choices of these parameters and d can be make by using the analyses above and characteristics of the data, so that the critical point at which it becomes cheaper to rehash can be determined.

Instead of rehashing to resolve collisions, we could maintain an overflow area of storage, using chaining to keep together all the items that hash to a particular position. Thus we would use storage records with three fields, one for the Key, one for the Entry, and one for a pointer to the next record in the sequence. As always, chaining means extra storage, but has some advantages concerning insertions and deletions, as well as being somewhat faster than the rehash methods, since the colliding items are kept separate. Note that we do not require that K <= N for the chaining method.

4-3. Chaining Method

Chaining methods is to store all synononous items which hash to a common location on a linked list or chain. Chaining method removes all the problems about the selection of rehash functions, but we still require that the initial hash function distributes the hashes uniformly throughout the table.

When such chains have a separate table of list heads, this metbod is called the indirect-chaining collision resolution method.

Chaining methods can be regarded as two categories: Indirect Chaining and Direct Chaining:

4-3.1 Indirect Chaining Method

This method of storage has advantages in both insertion and deletion, especially where ordering of the chains is required, but does have the disadvantage of requiring an additional use of a pointer on searches, and an extra amount of storage for the N pointers of the primary table.

It is desired that the buckets will be roughly equal in size, so the list for each bucket will be short. If there are N elements in the set, then the average bucket will have N/B members. If we can estimate N and choose B to be roughly as large, then the average bucket will have only one or two members,and the dictionary operations take, on the average,

some small constant number of steps, independent of what N (or equivalently B) is.

Algorithms of Operations on indirect Chaining Method: [16]

MEMBER -- When a key is to be looked up, its hash address is computed and then,

* -if that address is empty, the key has not been entered.
* -if that address is occupied, search down the chain hanging from that address (current := current^.next); if the key is not encountered, it is not in the table.

INSERT --

* -if not "member" then insert the new entry into the bucket header and next points to oldheader.

DELETE --

* -if address x is header of bucket, then let header := header^.next {remove x from list}.
* -search the key = x down the chain hanging from that address then delete; if the key is not encountered, it is not in the table.

4-3.2 Direct Chaining

It is also possible to dispense with the list heads, and merely originate the chain of items which hash to location i at cell i itself, carefully using otherwise-empty cells as the remaining nodes on the chain. This variation is called direct-chaining.

Direct chaining is considerably more efficient in terms

of number of probes per entry than either of the preceding methods. In this technique, part of one of the words in each entry is reserved as a pointer to indicate where additional entries with the same calculated address are to be found on a linked list (or chain) starting at that address. The last entry on each chain must be distinguished in some way (such as having a zero pointer).

Knuth [15] analyzed two variants: one that allows chains to coalesce, and one without coalescing but assuming that "foreign" records are forced out whenever necessary.

Algorithm (direct-chaining search and insertion)

This algorithm searches an M-nodes hash table, looking for a given key K. If the search is unsuccessful and the table is not full, then k is inserted. The size of the address region is M; the hash function hash returns a value between 1 and M, for convenience, we make use of bucket 0, which is always empty. The global variable R is used to find an empty space whenever a collision must be stored in the table. Initially, the table is empty, and we have R=M+1; when an empty space is requested, R is decremented until one is found. We assume that the following initialization have been make before any searches or insertions are performed [13] :

```
empty[i] <== true, for all 0 <= i <= M;
 and R <== M+1.
```

Then the algorithm can be as the following six steps:

1. HASH: Set i <- hash(K). (now 1 <= i <= M.)

2. IS THERE A CHAIN?

   If empty[i], then goto step 6. (Otherwise, the ith bucket is occupied, so we will look at the chain of records that starts there.)

3. COMPARE:

   if K = key[i],the algorithm terminates successfully.

4. ADVANCE TO NEXT RECORD:

   If link[i] <> 0 then set i = link[i] and go back to step 3

5. FIND EMPTY BUCKET:

   (The search for K in the chain was unsuccessful, so we will try to find an empty table bucket to store K.) Decrease R one or more times until empty[R] becomes true.If R = 0,then there are no more empty buckets, and the algorithm terminates with overflow. Otherwise, append the Rth cell to the chain by setting link[i] <- R; then set i <= R.

6. INSERT NEW RECORD:

   Set empty[i] <- false, key[i] <- K,link[i] <- 0, and initialize the other fields in the record.

Figure 1. shows the folw chart of chained scatter table search and insertion:

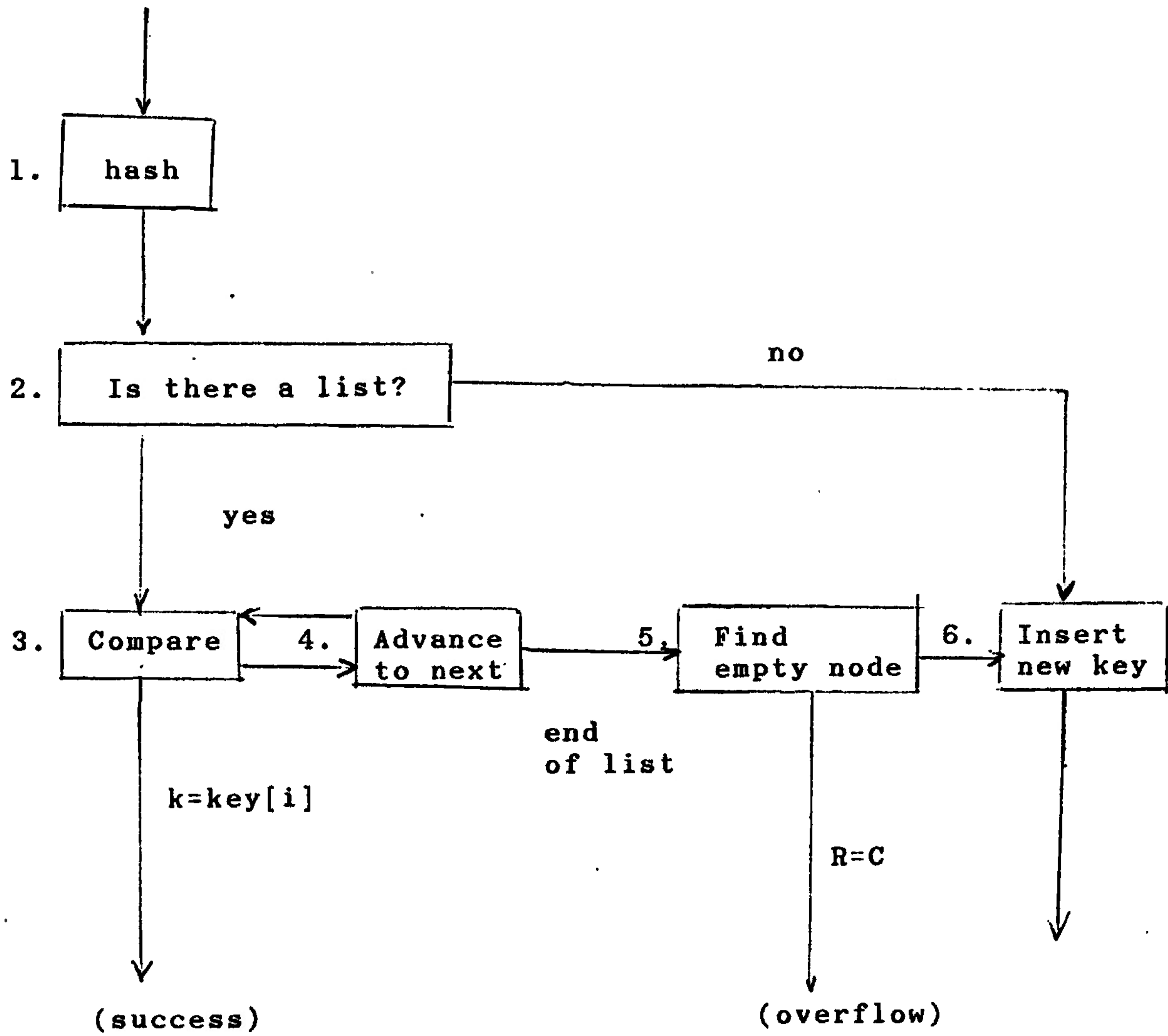

Figure 1. Flow chart for SEARCH and INSERTION

Deletions of Direct Chaning Methods

Many searching applications require that certain records be inserted and than later deleted.the paper below addresses the problem of constructing efficient deletion algorithms.The coalesced hashing method offers a particularly interesting setting for this study.

Correct deletion algorithms can be tricky to write, because changing the contents of a table bucket affects the successors in the chain. we cannot delete the record from location simply by setting EMPTY; otherwise, subsequent searches for records which hanging from deleted record would report failure when they encounter the empty bucket in location which was deleted.

One alternative is to include a special deleted field in each record,which says whether or not the record has been deleted.The search algorithm must be modified to treat each "deleted" table bucket as if it were occupied by a null record, even though the entire record is still there.

Unfortunately, a certain percentage of the "deleted" bucket will probably remain unused, thus preventing full storage utilization. Also, regardless of the number of undeleted records, the expected search times would approximate those for a full table, because the "deleted" records make the searches longer. If we are willing to spend a little extra time per deletion, we can do without the deleted field by relocating some of the records

that follow in the chain, and that will be good for the "search" time latter.

The Deletion Algorithm

Jeffrey Scott Vitter [13] provides a deletion algorithms quite interesting. The basic idea is this: First, we find the record we want to delete, mark its table bucket empty, and set the link field of its predecessor (if any) to the null value 0. Then we use Algorithm -insert to reinsert each record that is in the remainder of the chain, but whenever an empty bucket is needed in step 5, we use the position that the record already occupies. We can simplify this somewhat by observing that each record rehashes either to an occupied bucket or else to an empty bucket (called a hole) that had been occupied before the deletion.

Figure 2-1 shows an example of deleting AL from location 10 The end result is pictured in Fig.2-2. The first step is to create a hole in position 10 where AL was, and then to set AUDREY's link field to 0. Now we process the rest of the chain. The next record TOOTIE rehashes to the hole in location 10, so TOOTIE moves up to plug the hole, leaving a new hole in position 9. Next, DONNA collides with AUDREY. Then MARK also collides with AUDREY; we leave MARK in position 7 and link it to DONNA, which was formerly at the end of AUDREY's chain. The record JEFF rehashes to the hole in bucket 9, so we move it up to plug the hole, and a new hole appears in position 6. Finally, DAVE rehashes to

position 9 and joins JEFF's chain.

The problem is: location 6 is the current hole position when the deletion algorithm terminates, so we set empty[6] to true and return it to the pool of empty buckets However, the value of R in Algorithm insert is already 5, so step 5 will never try to reuse location 6 when an empty bucket is needed. We can get around this by using an available-space list in step 5 rather than the variable R; the list must be doubly-linked so that a bucket can be removed quickly from the list in step 6. The available-space list does not require any extra space per table slot, since we can use the KEY and LINK fields of the empty buckets for the two pointer fields. For clarity, we rename the two pointer fields NEXT and PREV. The variable AVAIL points to the start of the list. Before any records are inserted into the table, the following extra initializations must be make:

assign: AVAIL = M' ; NEXT[0] = M' ; PREV[M]' = 0 ; and

NEXT[i] = i-1 and PREV[i-1] = i, for 1 <= i <= M'.

We replace steps 5 and 6 by:

5. FIND EMPTY BUCKET:

(The search for K in the chain was nsuccessful,so we will try to find an empty table bucket to store K.)

If the table is already full(AVAIL = 0), the algorithm terminates with overflow. Otherwise, set LINK[i] = AVAIL and i = AVAIL.

6. INSERT NEW RECORD:

Remove the ith bucket from the available- space list by setting PREV[NEXT[i]] = PREV[i], NEXT[PREV[i]] = NEXT[i]; if i = AVAIL,set AVAIL = NEXT[AVAIL]. Then set EMPTY[i] = false, KEY[i] = K, LINK[i] = 0, and initialize the other fields in the record.

Keys:	A.L.	AUDREY	AL	TOOTIE	DONNA	MARK	FEFF	DAVE
Addresses:	11	1	1	10	1	1	9	

	KEY	LINK
1	AUDREY	10
2		
3		
4		
5	DAVE	0
6	JEFF	5
7	MARK	6
8	DONNA	7
9	TOOTIE	8
10	AL	9
11	A.L.	0

	KEY	LINK
1	AUDREY	8
2		
3		
4		
5	DAVE	0
6		
7	MARK	0
8	DONNA	7
9	JEFF	5
10	TOOTIE	0
11	A.L.	0

Fig.2-1 and Fig.2-2 Inserting the eight records and deleting AL.

We are now ready to specify the deletion algorithm:

<u>Algorithm of Deletion with Coalesced Hashing</u>

This algorithm preserves the important invariant that K is stored at its hash address if and only if it is at the start of its chain. This makes searching for K's predecessor in the chain easy: if it exists, then it must come at or after

position hash (K) in the chain.

1. [Search for K.]

   i = Hash (K) ;

   if empty [i] then goto end

   otherwise, if K = Key [i] then K is at the start of the chain, so go to step 3.

2. [Split chain in two] (K is not at the start of its chain.)

   Repeate

   PREV = i ;

   i = LINK[i]

   Until (i = 0) or (K= kEY[i];

   If i = 0 then go to end, else LINK[PREV] = 0 ;

3. [Process remainder of chain] (Variavle i will walk through the successors of K in the chain.)

   hole = i; i = LINK[i]; LINK[HOLE] = 0 ;

   Do step 4. zero or more times until i = 0.

   Then go to step 5

4. [Rehash record in ith bucket]

```
      while ( i <> 0 ) do
        {
          j <= hash (KEY(I))
          if j = hole then
            {
              KEY [ HOLE ] = KEY [i]
              HOLE = i
            }
          else
        link the record to the end of chain it collides with
            {
              while  (LINK [j] <> 0 ) do j = LINK [j]
              LINK [j] = i
              temp = LINK [i]
              LINK [i] = 0
```

```
            i = temp
          }
      }

5. [Mark bucket HOLE empty.]
     {
         EMPTY [ HOLE ] =true
         NEXT [ HOLE ] = AVAIL
         PREV [HOLE ] = 0
         NEXT [ 0 ] =HOLE
         AVAIL = HOLE
     }
```

CHAPTER V

Comparison of the Methods

5-1. General Considerations and Over-all Assessment

It is difficult to summarize in a few words all the relevant details of the "trade-offs" involved in the choice of a method, but the following things seem to be of primary importance whith respect to the speed of searching and the requisite storage space.

Table 1. Number of probes required on looking up a random item. [4]

packing density $k/n = \alpha$	Expected number of probes: Chaining. $1 + \alpha/2$	Linear prob. $(1-\alpha/2)/(1-\alpha)$	Random prob. $-1/\alpha \log(1-\alpha)$
0.1	1.05	1.06	1.05
0.5	1.25	1.50	1.39
0.75	1.38	2.50	1.83
0.9	1.45	5.50	2.56
0.99	1.50	50.5	4.65
1.5	1.75	-	-
2.0	2.00	-	-
5.0	3.5	-	-

This table showing that the various methods for collision resolution lead to different numbers of probes. But this does not tell the whole story, since the time per probe varies in different methods, and the latter variation has a noticeable

effect on the running time.

Table 1. shows that the chaining methods are quite economical with respect to the number of probes, but the extra memory space needed for link fields sometimes makes open addressing more attractive for small records. For example, if we have to choose between a chained scatter table of capacity 500 and an open scatter table of capacity 1000, the latter is clearly preferable, since it allows efficient searching when 500 records are present and it is capable of absorbing twice as much data. On the other hand, sometimes the record size and format will allow space for link fields at virtually no extra cost.

5-2. Hash Methods Compare with the Other Search Strategies

From the standpoint of speed, we can argue that they are better, when the number of records is large, because the average search time for a hash method stays bounded as N tends to infinity if we stipulate that the table never gets too full.

Table 2. shows comparison of internal table methods. In the table the number of key accesses is shown for various sizes of table. Only the operations of insertion and lookup are given. Deletion is similar to insertion: in both cases with chaining there is a significant overhead in pointer manipulations. The hash table figures disguise the possible significant cost of hashing. With these qualifications in mind we see from the table that all methods are comparable

for small tables while for large tables hashing is best. However, hashing relies on assumptions about key distribution and where these are inappropriate, one of the tree methods could be preferable.

Table 2. Comparison of internal table methods

Method	Operation	Table size 50	100	500	1000	Formula
Sequential vector	Insert.	52	102	502	1002	K+1+K/(K+1)
sorted	Lookup.	25.5	55.5	251	501	(K+1)/2
Comparison tree	Insert.	31.7	57.7	264	511	log(K+1)+K/2+1
logsearch	Lookup.	4.8	5.7	8.0	9.	(K+1)/Klog(K+1)-1
Comparison tree	Insert.	8.9	10.3	13.6	15.0	1.4log(K+1)+1
Chained	Lookup.	7.1	8.4	11.6	13	1.4(K+1)/K* log (K+1)-1
Hash. α = .5	Insert.	2.25	2.25	2.25	2.25	2 + α/2
chained overflow	Lookup.	1.25	1.25	1.25	1.25	1 + α/2

A rough guide to the best buy is given in the table: which method one chooses depends upon ease of programming,as well as speed and storage requirements.[4]

In complex table methods, not only must we bear in mind the

mix of operations, lookups versus changes, but also considerations of the relative sizes of keys, entries, and pointers.

Keys can be quite complicated, and accessing and comparing them can be a comparatively lengthy process. This then makes other operations, notably pointer manipulations, considerable more attractive in terms of speed. In general pointers will occupy less storage than keys, which will occupy less storage than entries. This makes the extra storage used for pointers a comparatively small overhead.

The comparative cheapness of pointers not only makes chaining methods more attractive than one might otherwise have thought, but also suggests that we use more pointers. Let us store a table of pointers to items, rather than the items themselves, storing the items themselves(or perhaps just the entries) in order of arrival in a simple sequential unsorted vector table. The table of pointers could be structured for efficiency, with the advantage that any movement of items (if the method demands this)in the efficient table becomes simply the movement of pointers.

There are three important respects in which scatter table searching is inferior to other methods we have discussed: [1]

1. After an unsuccessful search in a scatter table, we know only that the desired key is not present. Search methods based on comparisons always yield more information,

making it possible to find the largest key <= K and/or the smallest key >= K; this is important in many applications. It is also possible to use comparison-based algorithms to locate all keys which lie between two given values K and K'. Furthermore the tree search algorithms is make it easy to traverse the contents of a table in ascending order, without sorting it separately, and this is occasionally desirable.

2. The storage allocation for scatter tables is often somewhat difficult; we have to dedicate a certain area of the memory for use as the hash table, and it may not be obvious how much space should be allotted. If we provide too much memory, we may be wasting storage at the expense of other lists or other computer users; but if we don't provide enough room, the table will overflow. When a scatter table overflows, it is probably best to "rehash" it, means to allocate a larger space and to change the hash function, reinserting every record into the larger table. By contrast, the tree search and insertion algorithms require no such painful rehashing;the tree grow no larger than necessary.In a virtual memory environment we probably ought to use tree search or digital tree search, instead of creating a large scatter table that requires bringing in a new page nearly every time we hash a key.

3. Finally, we notice that hashing methods are

probabilistic. These methods are efficient only on average. In the worst case they are terrible! As in the case of random number generators, we are never completely sure that a hash function will perform properly when it is applied to a new set of data. Therefore scatter storage would be inappropriate for certain real-time applications such as air traffic control, where people's lives are at stake; the balanced tree algorithms are much safer, since they provide guaranteed upper bounds on the search time.

CHAPTER VI

Applications of Hash Table Methods

6-1 Hash Table Methods for File Directories

In computer file system, files are organized into directory; a file's information such as file name, file type, location, size, current position, protection, etc are kept in the direcotry for operating system to use. Depending different operating systems, it may take from 16 to over 1000 bytes to record this information for each file in the direcotry.

In a system with a large number of files, the size of the directory itself may be handreds of thousands of bytes. The directory itself can be organized in many ways. Hash table is regraded as the most ideal data structure for file directory.

To create a new file, we must first search the directory to be sure that no existing file has the same name. Then we can add a new entry at the end of the directory. To delete a file, we search the direcoty for the named file, then releases the space allocated to it. To reused the directory entry, we may do one of several things. we can mark it unused or attach it to a list of free directory entries.

All of these operations can be carried in the hash table methods discussed in previous chapters. Although other data structures such as linear list, sorted list, linked binary

tree may be sufficient for the operations on file directories, hash table is regarded as the best one of all. The hash table data structure requires less searching time than that of the linear list; it consumes less memory space then that of the linked binary tree; it require less maintenance overhead than that of the sorted list.

6-2 Hash Table Methods for Demand Paging Memory Management

Virtual memory uses a set of techniques that allow program to be executed when the entire program is not in memory. Demand Paging with swapping is the most common virtual memory system.

In Demand Paging, a program is given a small slot of memory address space, only one or two pages of this program is in there. When an item is referenced by the program and it is not already in memory,which is called "page fault", the page which contains the item must be brought into memory from secondary storage, one of the existing pages must give room to the new page --this is called swapping.

On such a machine, a hash table can be defined whose size exceeds the given memory address space of the program, so that every access to the hash table might cause a swapping to occur. Swapping slows down the execution of program. Therefore, it is most important to choose means of accessing entries which ensure that consecutive references to memory are as often a possible in pages that have recently been referenced and thus arelikely to be

already in memory.

Usually, page sizes are in the range from 2 to 2 words. If the entries themselves are kept in the hash table, then the linear probing of the hash table method becomes very attractive because consecutive probes are highly likely to be on the same page.

For a really large hash table, where it is impossible that the whole table can be held in memory, it would almost certainly be most efficient to use a hash index table and keep extra hash bits along with the pointer in the index table.[11] Also, collisions should be resolved within the index and not by chaining through free storage. Since the index table consists of single-word items, many more of a program's pages can be kept in the memory. Then the program stands the chance of needing a new page becomes considerable small.

6-3 <u>Hash Table Methods for Symbol Table in Compiler</u>

A symbol table contains all identifiers of a program. It is a production of lexical analyser durig the compilation process. The symbol table is then constantly looked up by other processes during compilation.

If we use hash table as the data structure for the symbol table, a hash function is defined on the class of identifiers; this function maps every identifier into an integer between 1 and h, where h is a fixed hash table size. We should provide a reasonably random and uniform mapping.

We can call the hashing value for some identifier its hash code. Given the hash code of an identifier, we enter the hash table directly through the hash code as an index and search for the identifier along a chain. The following figures show the data structure of the symbol table:

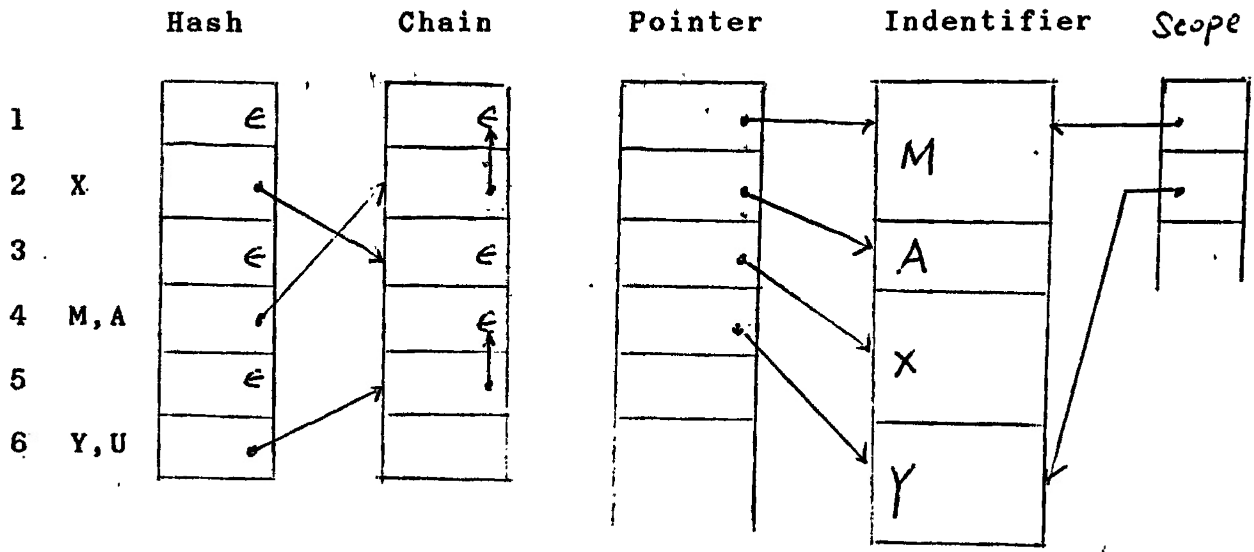

Figure 4-1 Data Structure of the Symbol table [18]

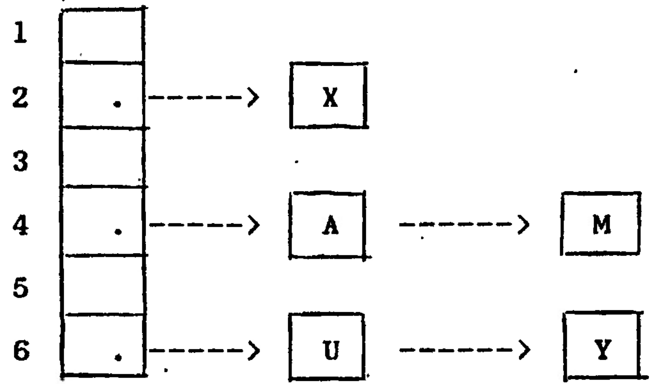

Figure 4-2 The hash chains of symbol table.

Note that a declaration search need go down a chain only

until the present scope is left. For a reference search, a chain must be followed to its end, if necessary, since an identifier may be in any csope.

There are many other data structures used by compiler construction, no matter which data structure we adopt, the acces time for each identifier is critical for the efficiency of compilation. In regard to this, John D. Couch [13] has a detailed discussion on four different methods to access entries of the symbol table: linear access, binary access, tree access, and hash access. He seems to be very happy with the hash access method. In the comparison of the above methods, he concludes:

> .....The most efficient access method, by time comparison, is the hash method. It requires a function that maps an identifier into a finite range of integers 1 to h in a uniform manner.... Althoug the binary tree search methods result in a sorted table, convenient for a symbol table listing, this apparent advantage is outweighed by the larger overhead in declaration and reference times.

6-4 Hash Table Method with Data Base Management

In a Data Base, records in a logical file are identified by means of the unique number of group of characters, called a key. The key is usually a fixed-length field which is in an identical position in each record. It may be an account number in a bank or a part number in a factory. It may be necessary to join two or more fields together in order to produce a unique key. The key of a

piece of records must be unique because that is used for determining where the record should be located on the file unit and for retrieving the record from the file.

Many applications of data base need to identify records on the basis of keys. The basic application is this: Given a key, such as an account number, how does the computer locate the record for that key? Hashing Table Methods are used extensively in data base applications.

Hashing is regarded as ingenious and useful way of address calculation technique for data base management in two respects: access efficiency and storage efficiency.

6-4.1 Access Efficiency

The access efficiency of the hashing method depends on two factors:

1. Original Key Distribution. The more the designer of a data base knows about the distribution, the better position he/she is in to select the number of blocks and the number of home address per block. The optimum selection of these factors will enable the designer to reduce the average length of the synonym chain.
2. Space Allocated. The major issue for access efficiency is EVEN distribution of the actual keys over the number of blocks, i.e., the space allocated. If the hash function assigns many keys in one area, the result is a larger number of synonyms. In this:

$h(k)=i, \quad (1 <= i <= n)$

the larger value of n, the better randomness can be achieved.

6-4.2 Storage Efficienct

The storage efficiency depends on the space allocated and the hash function. When using hash table methods, it is advisable not to specify any free space within the blocks. The reason is that the hashing function may randomize to the free blocks and to the free space within a block; this will result in putting the corresponding record into the overflow area.

CHAPTER VII

Summary of the Study

Hash Table Methods are conceptually elegant and extremely fast methods for information storage and retrieval. This thesis has examined in detail several practical issues concerning the implementation of these methods.

The most important issue addressed in this thesis is the efficient implementation of hash table methods. The author finds that there are critential trade-offs in a desired implemention. These are discussed in issues such as hash addressing, handling collision, hash table layout, and bucket overflow problems.

The comparisons of various hash functions in chapter III shows that the criteria of good hash function is providing even distribution, and at the same time, it must be easily computed.

Collision is the major problem in hash table methods. Differing in the manner in resolving collisions, two major hash table methods are discussed in chapter IV. Open Addressing Method places the synonymous items (items with the same hashed address) somewhere within the table. The Chaining Method, however, chains all synonymies and store them somewhere outside the table called overflow area.

In applications where inserting, deleting, or searching are necessary, the auther has illustrated several applilcation examples found in the computer's system software. Hash Table is widly used by system software as an ideal data structure such as compiler's symbol table, data base, directories of file organizations, not to mention the protential popularity in problem-solving application programs.

BIBLIOGRAPHY

F.R.Hopgood

COMPUTER BULLETIN 11 (1968), 297-300

P.Flajolet, J.Francon, and J.Vuillemin.

"Sequence of Operations Analysis for Dynamic Data Structures", JOURNAL OF ALGORITHMS 1, 111-141 (1980)

Dennis G. Severance

"Identifier Search Mechanisms: A Survey and General--ized Model", COMPUTING SURVEYS, Vol.6, no.3, September 1974.

P.Quittner, S.Cso'ka, S.Hala'sz, D.Kotsis, and K.Va'rnai.

"Comparison of synonym Handling and Bucket Organization Methods", COMMUNICATIONS OF THE ACM Vol.24, number 9, p.579-583.

Per-Ake Larson

"Analysis of Hashing with Chaining in the Prime Area", JOURNAL OF ALGORITHMS 5,36-47 (1984)

Richard P.Brent

"Scatter storage techniques", COMMUNICATIONS OF THE ACM February 1973 Vol,16 no.2 105-109

Gary D.Knott

"Direct-Chaining with Coalescing Lists" JOURNAL OF ALGORITHMS 5, 7-21 (1984).

S.L.Graham, R.L.Rivest,

"Pseudochaining in Hash Tables" COMMUNICATIONS OF THE ACM july 1978, Vol.21 Number 7, 554-557.

W.D. Maurer and T.G. Lewis.

"Hash table methods" p.6-19.

Robert Morris.

"Scatter Storage Techniques" COMMUNICATIONS OF THE ACM Vol.11,number 1 January 1968, 38-44.

Jeffrey Scott Vitter.

"Coalesced Hashing" COMMUNICATTONS OF THE ACM December 1982 Vol.25,number 12, 915-926.

Jefrey Scott Vitter.

"Deletion Algorithms for Hashing that Preserve Ran--domness", JOURNAL OF ALGORITHMS 3, 261-275 (1982)

Ellis Horowitz, Sartaj sahni.

FUNDAMENTALS OF DATA STRUCTURES p.456-471.

Donald E. Knuth.

THE-ART OF COMPUTER PROGRAMMING Vol.3 p.508-471.

Alfred V.Aho, John E. Hopcroft, and Jeffrey D. Ullman.

DATA STRUCTURES AND ALGORITHMS p.122-135.

Aho, Hopcroft, Ullman.

THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS p.108-112.

William A. Barrett, John D. Couch.

COMPILER CONSTRUCTION: THEORY AND PRACTICE p.368-371