# MULTI-DIMENSIONAL HASH TABLE AND

# APPLICATION IN GRIDDING

By

CHUANJIANG LU

Bachelor of Science
Changchun University of Earth Sciences
Changchun, P. R. China
1983

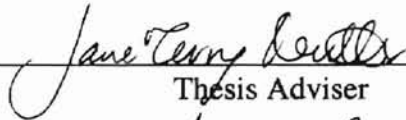
Master of Science
Changchun University of Earth Sciences
Changchun, P. R. Chain
1993

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2000

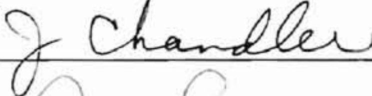
MULTI-DIMENSIONAL HASH TABLE AND

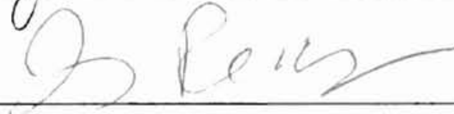
APPLICATION IN GRIDDING

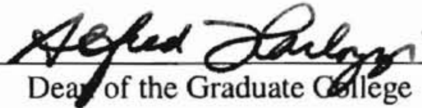
Thesis Approved:



Thesis Adviser







Dean of the Graduate College

## ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my thesis advisor, Dr. J.Terry Nutter, for her intelligent supervision, constructive guidance, constant inspiration, and valuable time she has given me toward the completion of my thesis work. My sincere appreciation extends to other committee members, Dr. John P. Chandler and Dr. Jing Peng, for their helpful advisement and suggestions.

In addition, I would like to give my special appreciation to my wife, Ying Wang, for her strong encouragement at times of difficulty, love, support and understanding throughout this whole process. Thanks also go to my parents for their love and encouragement.

Finally, I would like to thank the Department of Computer Science for supporting during the time of my study.

## TABLE OF CONTENTS


Chapter Page

I. INTRODUCTION ... 1
- 1.1 Background and Motivation ... 1
- 1.2 Objectives ... 2
- 1.3 Organization ... 2

II. LITERATURE REVIEW ... 3
- 2.1 The History of Hashing ... 3
- 2.2 Hash Table ... 3
  - 2.2.1 Hash Function ... 4
  - 2.2.2 Collision Resolution techniques ... 5
- 2.3 Major Hashing Schemes ... 6
  - 2.3.1 Linear Hashing ... 6
  - 2.3.2 Extendible Hashing ... 7
- 2.4 M-D Space Data Access Methods ... 7
  - 2.4.1 Grid File: Multi-Key File Structure ... 8

III. M-D HASHING TABLES ... 12
- 3.1 2-D Hash Table Features ... 12
- 3.2 Collision Resolution ... 13
  - 3.2.1 By Chaining ... 13
  - 3.2.2 By Open Addressing ... 15
- 3.3 Improving on Open Addressing Methods ... 20
  - 3.3.1 Model 1 ... 20
  - 3.3.2 Model 2 ... 28

IV. HASHING PERFORMANCE ... 32
- 4.1 Expected Performance ... 32
  - 4.1.1 Average Performances of Various Hashing Methods ... 32
  - 4.1.2 Required Memory ... 33
- 4.2 Testing for Various Hashing Methods ... 34
  - 4.2.1 Testing Procedures ... 34
  - 4.2.2 Test Results ... 35
    - 4.2.2.1 Chaining Hashing ... 37
    - 4.2.2.2 Linear Probing ... 37
    - 4.2.2.3 Double Hashing ... 41
- 4.3 Analysis and Comparison ... 43

Chapter Page

V. APPLICATION IN GRIDDING ..... 46
5.1 Contour Map System ..... 46
5.2 Gridding in Contour Map System ..... 47
5.3 Hashing in Contour Map System ..... 49
5.3.1 Hashing Function ..... 50
5.3.2 Collision Solutions ..... 51
5.4 Analysis and Comparisons ..... 52

VI. SUMMARY AND CONCLUSION ..... 53

REFERENCES ..... 55

APPENDIX A--TABLES OF TESTING RESULTS ..... 57

## LIST OF TABLES


Table Page

4-1 Expression of Probes Expected for Successful and Unsuccessful Search, as well Unsuccessful Search with Improved Open addressing in a hash Table ... 32

4-2 The Number of Probes Expected for Successful, Unsuccessful, and Improved Search in Hash Table ... 33

4-3 The Memory Requirements for Different Methods ... 34

4-4 Average Number of Successful Search from Testing ... 44

4-5 Average Number of Unsuccessful Search from Testing ... 44

A-1 Successful Search with Chaining Hashing ... 58

A-2 Unsuccessful Search with Chaining Hashing ... 58

A-3 Successful Search with Linear Probing ... 59

A-4 Unsuccessful Search with Linear Probing ... 59

A-5 Successful Search with Linear Probing Improved by Model 1 ... 60

A-6 Unsuccessful Search with Linear Probing Improved by Model 1 ... 60

A-7 Successful Search with Linear Probing Improved by Model 2 ... 61

A-8 Unsuccessful Search with Linear Probing Improved by Model 2 ... 61

A-9 Successful Search with Double Hashing ... 62

A-10 Unsuccessful Search with Double Hashing ... 62

A-11 Successful Search with Double Hashing Improved by Model 1 ... 63

A-12 Unsuccessful Search with Double Hashing Improved by Model 1 ... 63

A-13 Successful Search with Double Hashing Improved by Model 2 ... 64

Table | Page

A-14 Unsuccessful Search with Double Hashing Improved by Model 2 ......... 64

A-15 Total Average Successful Search Times of One Key with Load Factor Under 0.9 ......... 65

A-15 Total Average Unsuccessful Search Times of One Key with Load Factor Under 0.9 ......... 65

LIST OF FIGURES


| Figure | | Page |
|---|---|---|
| 3-1 | 2-D Hash Table T[m, n] | 12 |
| 3-2 | 2DHT T[m, n] with Extra Memory for $N_c$ and $N_p$ | 20 |
| 4-1 | Average Number of Successful Search with Load Factor Under 0.9 | 36 |
| 4-2 | Average Number of Unsuccessful Search with Load Factor Under 0.9 | 36 |
| 4-3 | Search in Chaining Hash Table | 38 |
| 4-4 | Search in Linear Probing Hash Table | 39 |
| 4-5 | Successful Search with Linear Probing | 40 |
| 4-6 | Unsuccessful Search with Linear Probing | 40 |
| 4-7 | Search in Double Hash Table | 41 |
| 4-8 | Successful Search with Double Hashing | 42 |
| 4-9 | Unsuccessful Search with double Hashing | 42 |
| 5-1 | Contour Map System | 46 |
| 5-2 | Procedure of Gridding | 48 |
| 5-3 | Hashing Sample Data Points to Hash Table | 50 |

# CHAPTER I

# INTRODUCTION

The hash table is a kind of data structure that has been developed and applied in data processing since the beginning of the 1950s [7]. Because hash tables take a great deal of main memory, their applications largely got under way in the 1970's [7]. With advances in computer hardware, hash tables are becoming more and more popular.

Search techniques for data scattered on two or more dimensional storage, sometimes referred to as hash techniques, have been developed to provide a means whereby external labels, or keys, may be mapped to unique or nearly unique internal numbers [7]. Extendible hashing has been widely applied for this kind of data retrieval [17]. However, in cases where data patterns are near random, multi-dimensional hash tables would be more efficient for search and other data processing.

## 1.1 Background and Motivation

This study of two-dimensional hash tables is directed toward a geographic contouring map system that is widely used in geology, agriculture, environmental studies, and other similar applications [10] [18]. Gridding is a set of methods that evaluate the value of regular grids from irregular known control points in space [10]. Gridding forms a major part of the Contouring Map System (CMS); the other major component is visualization. Gridding requires enormous search effort. Thus, it requires a highly efficient data structure for search. Hashing is one of the most efficient and simple search methods, especially for irregular, near random data. This thesis proposes a new data

structure, the M-D hash table, to form the basis of search and retrieval of data from multi-dimensional spaces.

## 1.2 Objectives

The research reported here has three objectives:

- to analyze, develop, and implement 2-D hash tables, including hash table creation, hash function derivation, and clustering management;
- to improve performance of hash table operations, where sufficient memory is available, by reducing the number of probes required either to find a record or to determine that it is absent; and
- to apply M-D hash tables in the context of CMS to improve gridding performance.

## 1.3 Organization

The rest of this thesis is as follows. Chapter II reviews related work in existing literature. Chapter III introduces 2-D hash table features and implementation. Chapter IV summarizes the performance of two-D hash tables. Chapter V describes applications of M-D hash tables. Finally, Chapter VI presents conclusions.

# CHAPTER II

## LITERATURE REVIEW

Data structures for main memory fall into three categories [7]: linearly or sequentially accessible data structures (time complexity O(n)), tree-based structures (O(log(n)) ), and the most efficient class, hash tables (O(1) ) [2].

### 2.1 The History of Hashing

The basic concepts behind hashing originated in the early 1950s [7], with chaining widely applied to resolve conflicts; this constituted one of the first applications of linear linked lists, used to represent buckets that contain more than one element for external searching [7]. At about the same time the idea of hashing occurred independently to a group of researchers at IBM, who originated the idea of open addressing with linear probing for conflict resolution [7]. Developing hash functions by dividing by a prime number and using the remainder as the hash address emerged in 1956, as did a second open addressing strategy, that of random probing by independent hash functions [7].

By the late 1970s, most of the important currently recognized hash methods had been introduced, including extendable hashing, and techniques that permit hash tables to expand and shrink dynamically [14].

### 2.2 Hash Table

Hash tables can be viewed as a generalization of the simpler notion of ordinary arrays. Hashing provides an extremely effective and practical technique for implementing

basic dictionary operations (search, insertion, and deletion) with an average time complexity of O(1) [2].

A hash table $T$ is an array of size $m$ in which, ideally, each element $k$ is stored at $T[h(k)]$, where $h$ is the hash function. This scheme permits constant time operations under two assumptions [2]: (1) computing $h$ is itself a low complexity operation; and (2) no two keys hash to the same address, that is, $h(k) = h(k')$ if and only *if* $k = k'$. The first assumption is usually unproblematic. The second, however, is not. When $h(k) = h(k')$ for two distinct keys k and $k'$, and when both keys occur in the data, the result is called a collision [2]. Several effective techniques to resolve collision have been developed. Hence hashing scheme designers focus on two issues [7]: computationally simple hash functions that reduce collisions; and collision resolution techniques.

### 2.2.1 Hash Function

There are three common schemes for creating hash functions: hashing by division, hashing by multiplication, and universal hashing [2].

- The division method:

  The hash function is $h(k) = k \bmod m$ , which puts a key $k$ into one of $m$ slots [2]. Good values for $m$ are primes. If $m$ is an even number, $h(k)$ will be even when $k$ is even and odd when $k$ is odd. In addition, using a prime number for $m$ can easily avoid incomplete search (some buckets of a hash table cannot be probed if $m$ is divisible by offset) for collision resolution with open addressing methods. Since this method requires only a division operation, hashing by this method is quite fast.

- The multiplication method:

  The hash function is $h(k) = \lfloor m(kA \bmod 1) \rfloor$, where A is a constant in the range 0 to 1 [2]. An advantage of this method is that the value of m is not critical [2]. However, since this method requires two multiplication operations and one division operation, it is slower than the division method.

- Universal hashing:

  The main idea is to choose the hash function randomly in a way that is independent of the keys that are stored and to select the hash function at random at run time from a designed class of functions [2]. The advantage of this method is that any fixed hash function results in an average time complexity of $\Theta(n)$. However, it can be difficult to design an ideal class of functions [2].

### 2.2.2 Collision Resolution Techniques

- Resolution by Chaining

  This technique places all the elements that are hashed into the same slot in a linked list [2]. In a hash table in which collisions are resolved by this scheme, an unsuccessful search takes time $\Theta(1 + \alpha)$ and a successful search takes time $\Theta(1 + \alpha/2)$ [2], where $\alpha$ is the hash table load factor. Using this method, the table size is not critical, and it usually has a good performance [2]. But it takes more memory than open addressing methods do [7].

- Resolution by Open Addressing

With open addressing, slots are probed until an empty one is found. The following three methods are often used:

- Linear probing: $h(k, i) = (h'(k) + ci) \bmod m$, where $h'(k)$ is the hash value of initial probe; $c$ is the offset for each probe; and $i$ is between 0 and m-1 [2]. This method is easy to implement, but it suffers from a problem, primary clustering (two keys have the same probe position, then their probe sequence are the same [2]).
- Quadratic Probing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where $h'(k)$ is the initial probe value; $c_1$ and $c_2 \neq 0$ are auxiliary constants [2]. It eliminates primary clustering, but this method leads to secondary clustering (if two keys have the same initial probe position, then their probe sequences are the same [2]).
- Double hashing: $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$, where $h_1(k)$ is the initial hash value; $h_2(k)$ is a computed offset of $k$. This method can eliminates both primary clustering and secondary clustering [2].

## 2.3 Major Hashing Schemes

### 2.3.1 Linear Hashing

Linear hashing (not to be confused with linear probing), proposed by Litwin [8], permits a hash table to expand and shrink dynamically without requiring an index. It is mainly used in file structures to handle growth and shrinking of files [4]. Files grow in two ways: overflow growth and regular growth. Regular growth under linear hashing

with two partial expansions (LH2P) has two forms [8]: partial expansions and full expansions. Each full expansion doubles the number of regular buckets in the file, and consists of two partial expansions. The first partial expansion increases the number of regular buckets by 50%, and the second increases the number by the same amount. Thus, after the first partial expansion, $nb = 3*(m \text{ div } 2)$; after the first full expansion, $nb = 2*m$; and after the second full expansion, $nb = 4*m$; where $m$ is the number of buckets for the initial file or data, and $nb$ is the number of buckets [8].

### 2.3.2 Extendible Hashing

Extendible hash tables are a dynamic data structure used most often as an alternative indexing strategy to B-trees [4] [12], for example in databases. In extendible hashing, locating a key and its associated information never involves more than two faults (one fault is one probe that did not find target position) [14], even for very large data sets. In extendible hashing, each bucket records the number of bits of the hash address that determine which keys are in that bucket. This number is called the bucket depth. Initially, the number is the number of bits used by the root (thus, the initial number of buckets = $2^b$) for all entries; it is increased by one each time a bucket splits [14].

## 2.4 M-D Space Data Access Methods

Data records in M-D Space Data (MDSD) contain more than one attribute. MDSD can be accessed either by a single key, most often one on which all other attributes rely, or by several keys, all used together. Single-key access is much easier than multi-key

access [11]. The design of balanced data structures must be more difficult for MDSD (each record is identified by several attributes) than for one-dimensional data, since most balanced structures for single-key data rely on a total ordering of the set of key values, and natural total orders of multidimensional data do not exist. This section reviews several multi-key data access methods.

Data and file structures can be divided into two broad categories: those based on the specific set of data to be stored, and those based on the embedding space from which the data set is drawn [11] [6]. Comparative search techniques such as binary search trees fall into the first category: search directly focuses on the value to be stored. Address computation techniques such as hash tables belong to the second class: the locations at which records with a given key may be stored are fixed regardless of the values or contents of the rest of the data set [11].

### 2.4.1 Grid File: Multi-Key File Structure

Traditional file structures, such as inverted files, are extensions of file structures originally designed for single-key access [11]. Grid file structures are designed to address dynamic aspects of structures that treat all keys symmetrically, that is, data sets that avoid the distinction between primary and secondary keys [11]. Focus on multiple symmetric keys leads to the notion of a grid partition of the search space and to that of a grid directory. These two concepts are the keys to a dynamic file structure [7] [11].

*Grid Partitions of the Search Space*

Each search technique partitions the search space into subspaces, down to the level

of resolution of the implementation, typically determined by bucket capacity [6]. To retrieve a data record, correlated attributes that are functionally dependent on each other are more efficient than independent attributes [11]. Assuming independent attributes, such as spatial dimensions in a geometric database system, grid partition of the search space is obviously suited for range and partially specified queries.

We use the following terminology and notation for the three-dimensional case [11]: on a record space $S = X * Y * Z$, we impose a grid partition $P = U * V * W$ by imposing intervals $U = (U_0, U_1, \ldots, U_l)$, $V = (V_0, V_1, \ldots, V_m)$, and $W = (W_0, W_1, \ldots, W_n)$ on each axis and then dividing the record space into blocks called grid blocks [11]. The grid partition $P = U * V * W$ is modified only by altering one of its components at a time. A one-dimensional partition is modified either by splitting one of its intervals into two, or by merging two adjacent intervals into one [11].

*The Grid Directory*

The design of a bucket management system involves three parts [11]:

1. defining a class of assignments of grid blocks to buckets;
2. choosing a data structure for a directory that represents the current assignment;
3. finding efficient algorithms to update the directory when the assignment changes.

The two-disk-access principle implies that all the records in one grid block must be stored in the same bucket, although several grid blocks may share one bucket, so long as the union of these grid blocks forms a rectangular box in the space of records [11]. The feature of bucket regions obviously affects the speed of range queries, and of update to a

modification of the grid partition [6] [11].

The grid directory represents and maintains the dynamic correspondence between grid blocks in the record space and data buckets. It is a data structure that supports the operations needed to update the convex assignments (grid blocks to buckets) when a bucket overflows or underflows [6]. A grid directory consists of two parts: first, a dynamic k-dimensional array called the grid array, the elements (pointers to data) of which are in one-to-one correspondence with the grid blocks of the partition; and second, k one-dimensional arrays called linear scales that define a partition of a domain S [11].

For notational simplicity, let $k = 2$, with record space $S = X * Y$. As described in [6] and [11], A grid directory $G$ for a 2-D space is characterized by

1. Integers $n_x$ and $n_y$ (extent of directory) for $n_x > 0$ and $n_y > 0$;
2. Integers $c_x$ and $c_y$ (current element of the directory and current grid block) for $0 < c_x < n_x$, $0 < c_y < n_y$;
3. Grid array: $G(0 \ldots n_x, 0 \ldots n_y)$;
4. Linear scales: $X(0\ldots, n_x)$, $Y(0 \ldots, n_y)$;

Operations defined on the grid directory consist of

1. Direct access: $G(c_x, c_y)$
2. Next in each direction

   $Nextxabove(c_x) = (c_x + 1) \bmod n_x$

   $Nextxbelow(c_x) = (c_x - 1) \bmod n_x$

   $Nextyabove(c_y) = (c_y + 1) \bmod n_y$

   $Nextybelow(c_y) = (c_y - 1) \bmod n_y$

3. Merge

mergex: given $p_x$, $1 \le p_x < n_x$, merge $p_x$ with nextxbelow; rename all elements above $p_x$ and adjust X-scale.

mergey: similar to mergex for any $p_y$, $1 \le p_y < n_y$.

4. Split

splitx: given $p_x$, $0 \le p_x \le n_x$, create new element $p_x + 1$ and rename all cells above $p_x$:

splity: similar to splitx for any $p_y$, $0 \le p_y \le n_y$.

*Record Access*

The array G is usually large, and so stored on disk (secondary storage); X and Y of the linear scales are small, and kept in main memory. To access a record with two independent attributes, a 2-key access scheme is used. The attribute values are converted into interval indexes through a search (in main memory) of scales X and Y [11] [6]. The interval indexes provide direct access to the correct element of the grid directory, where the bucket address is located. For example, consider a record space with attributes "date" (with domain "Monday … Sunday") and "time" (with domain "1pm … 5pm"). The grid partition in the record space is:

X = (Mon, Tues, Wed, Thurs, Fri, Sat, Sun);

Y = (1pm, 2pm, 3pm, 4pm, 5pm).

In a search for a fully specified query (r1, r2, …, ), such as finding a record [Wed, 2:30pm], the attributes of record [Wed, 2:30pm] are converted into interval index 3 in scale X, and 2 in scale Y. Grid files also handle range queries efficiently, including the special case of partially specified queries [11].

# CHAPTER III

# M-D HASHING TABLES

Traditional hash tables based on one-dimensional arrays can only hold records accessed by a single primary key. In fact, database systems frequently rely on compound data types, such as class, record, and so on, with more than one primary key. For processing some compound data with multiple primary keys, and in particular for multi-dimensional space data (MDSD), the current research proposes M-D hash tables (MDHTs). This chapter presents hash table features and collision resolution strategies for MDHTs.

## 3.1 2-D Hash Table Features

The studies reported here focus on 2-dimensional hash tables (2DHTs) T[m, n] (Fig. 3-1); MDHTs of higher dimensionality are a straightforward generalization of the two-dimensional case. In 2DHTs, key values with compound data types are hashed to slots.

| (0,0) | (0,1) | ... | (0,m) |
|---|---|---|---|
| (1,0) | (1,1) | ... | |
| ... | ... | | ... |
| (n,0) | (1,n) | ... | (m,n) |

**Figure 3-1** 2-D Hash Table T[m,n]

2DHTs have the following characteristics:

1. Key K is a scattered point $(k_x, k_y)$, where x and y are coordinates.
2. Hash value h(K) is between (0, 0) and (m-1, n-1).
3. Hash function h consists of $h_x$ and $h_y$, where $h_x$ is a hash function for key $k_x$, $h_y$ is a hash function for key $k_y$, and $0 \le h_x(k_x) < m$ and $0 \le h_y(k_y) < n$.
4. Each hash table has a load factor, $\alpha$, that is the ratio of the number of elements in the hash table to the table size.
5. Hashing functions:

   General hash function for 2DHT is $h(K) = (h_x(k_x), h_y(k_y))$

   Division Method: $h(K) = h(k_x, k_y) = (h_x(k_x), h_y(k_y)) = ((k_x \bmod m), (k_y \bmod n))$

   Multiplication Method: $h(K) = h(k_x, k_y) = (h_x(k_x), h_y(k_y))$

   $= (\lfloor m_x(k_x A_x \bmod 1) \rfloor, \lfloor m_y(k_y A_y \bmod 1) \rfloor)$

## 3.2 Collision Resolution

### 3.2.1 By Chaining

Each element T[i, j] of a 2DHT is a pointer pointing to the head of a linked list, each node of which contains a key $K = [k_x, k_y]$ and a pointer the next node. Depending on the application, nodes may also have additional fields for associated data. For our purposes, we represent nodes as Node{K, P} where K has $k_x$ and $k_y$, and P is the pointer. We examine the following candidate algorithms.

Algorithm $C_1$ (Chained hash table insertion)

**Chained-Insert (T, K)**

1. $i = h(k_x)$, $j = h(k_y)$ // hashing
2. IF T[i, j] is NULL THEN // insert
3. T[i, j] := &Node{K, P}, and T[i, j]->P := NULL
4. ELSE Node{K, P}.P := T[I, j], and T[i, j] := &Node{K,P}

Algorithm C2 (Chained hash table search)

**Chained-Search(T, K)**

1. $i := h(k_x)$, $j := h(k_y)$
2. IF T[i, j] is NULL THEN
3. unsuccessful search
4. ELSE searching in the linked list T[i, j]

Algorithm C3 (Chained hash table delete)

**Chained-Delete(T, K)**

1. $i := h(k_x)$, $j := h(k_y)$
2. delete the Node containing the K from the linked list T[i, j]

All of the above algorithms have theoretical time complexities of O(1) for insertion, $O(1+1/\alpha)$ for successful search or deletion, and $O(1+\alpha)$ for unsuccessful search.

3.2.2 Open Addressing

*Linear Probing*

Linear probing fixes an increment c, where c is a small integer, and m is not divisible by c, and searches at locations separated by c through the hash table beginning at the position where the collision occurred and continuing until an empty position is found or the entire table has been searched. That is, given key K and hash table T[m, n], the first probing position is $T[h_x(k_x), h_y(k_y)]$. Thereafter, if location T(i, j) is probed and full, the next location examined is T[(i + c) mod m, (j + c) mod n]. The values of m and n must be not divisible by c, so that the table can be searched completely if necessary.

We can also probe row-by-row, which proceeds as follows. Let i be the number of probes so far (where i = 0 represents the initial hash), let $x_i$ be the column of the $i^{th}$ probe, and let $y_i$ be the row of the $i^{th}$ probe. Then $x_0 = h_x(k_x)$ and $y_0 = h_y(k_y)$. While $i \leq m$, $T[x_{i+1}, y_{i+1}] = T[(x_i+c) \bmod m, y_i]$. Each time i becomes a multiple of m, the current row has been exhausted. At that point, $T[x_{i+1}, y_{i+1}] = T[(x_i+c) \bmod m, (y_i+1) \bmod n]$, moving search to the next row. If i reaches m*n, the entire table has been searched.

Algorithm $L_1$ (Linear probing insertion)

**Linear-Probe-Insert(T, K)**

1. $i = h_x(k_x)$, $j = h_y(k_y)$
2. DO LOOP: WHILE T[I, j] is not empty OR count < mn –1
3. If i >= m THEN i := i - m, j ++
4. ELSE i := i + c // m % c ≠ 0

5. count ++

6. END LOOP

7. IF T[i, j] is empty THEN insert K into T[i, j]

8. ELSE is overflow

Algorithm $L_2$ (Linear probing search)

**Linear-Probe-Search(T, K)**

1. $i = h_x(k_x)$, $j = h_y(k_y)$
2. DO LOOP: WHILE T[i, j] is not empty AND count < mn – 1
3. IF key[i, j] = K THEN found = true, BREAK // initial found = false
4. ELSE IF i >= m-1 THEN i := i - m, j := j + 1
5. ELSE i := i + c // c is offset
6. count := count + 1 //initial count = 0
7. END LOOP
8. IF found = true THEN successful search K is in T[i, j]
9. ELSE unsuccessful search

Algorithm L3 (Deletion from tables built by linear probing)

After an element is deleted from the hash table at T[m, n], a gap (an empty bucket where the key has been deleted) appears in the hash table T[m, n]. Since search normally halts if the position where gap appears is probed, the result of this search will be incorrect if the key is in table past the gap. To avoid the gap appearing and maintain the properties of hash tables with linear probing, the hash table must be searched after deletion for keys

that would have been inserted at that location had the deleted element not been there. The last such element will be moved up to this position to eliminate the gap.

**Linear-hash-delete(T, K)**

```
1.  K is found at T[i, j]
2.  delete K from T[i, j]        // gap occurs
3.  gx = i, gy = j               // for memorizing the gap position
4.  LOOP1: WHILE T[i, j] is not empty AND count != mn - 1
5.     i := i + c
6.     IF i >= m THEN i = i – m, j := j + 1        // for next row
7.     END IF
8.     tx := i, ty := j                   // temporary position
9.     LOOP2: WHILE i != hx(key[tx, ty].kx) or j != hy(key[tx, ty].ky) // no first hashed in
10.          i := i – c
11.          IF i < 0 THEN i := m + i, j := j – 1  // back one row
12.          END IF
13.          IF i = gx and j = gy THEN  BREAK        // for inserting to gap
14.    END LOOP2
15.    IF i = gx and j = gy THEN
16.          T[gx, gy] := T[tx, ty]               // move position to gap position
17.          delete T[tx, ty]
18.          gx := i := tx, gy := i := ty       // new gap
19.    END IF
20.    count := count + 1                  // count is the number of searches
```

21. END LOOP1

There are two problems underlying the linear probe method [7]. The first, primary clustering, occurs because any key hashed to position h follows the same hashing pattern as all other keys hashed to h. Secondary clustering occurs because two keys that have the same initial probing position also have the same probing sequences. To avoid primary and secondary clustering, we examine next a method that does solve the problem, double hashing.

*Double hashing*

Double hashing is an attempt to approximate an ideal strategy that responds to collisions by jumping randomly to a new table position. This strategy is called random hashing [2] [7]. The primary problem with random hashing is reproducing the probe intervals in the subsequent search. To do this, we apply a second hash function to the original key, using that for an increment. This approximates random hashing, and it eliminates primary clustering because two distinct keys that are initially hashed to the same position almost always use different increments derived from the second function. The values produced by h2 (step size of next hash) must be relatively prime to column size m of the hash-table to insure that every position of the table is eventually probed. The process of the probe is repeated until the target key or an empty position is found or until the table is identified to be full and not to have the target key.

Algorithm $D_1$ (Insertion with double hashing)

**Double-Hash-Insert(T, K)**

1. $i = h_x1(k_x)$, $j = h_y(k_y)$, and $c = h_x2(k_x)$ // T[i, j] is current probe, c is the step size // for next probe
2. LOOP: WHILE T[i, j] is not empty AND count < mn-1 // count is number of probes
3. IF the (count mod m)= m THEN j := j + 1
4. ELSE i = i + c
5. count := count + 1
6. END LOOP //
7. IF T[i, j] is empty THEN Insert K into T[i, j]
8. ELSE the table is overflow

Algorithm D2 (Search with double hashing)

**Double-Hash-Search(T, K)**

1. $i = h_x1(k_x)$, $j = h_y(k_y)$, and $c = h_x2(k_x)$ // T[i, j] is current probe, c is the step size
2. LOOP: WHILE T[i, j] is not empty AND count < mn-1 // count is the number of // probes
3. IF T[i, j].key is equal to K THEN found is true, BREAK
4. ELSE IF the (count mod m)= m THEN j := j + 1 // to search next line
5. ELSE i = i + c
6 count := count + 1
7. END LOOP
8. IF found is true THEN search is successful
9. ELSE search is unsuccessful

## 3.3 Improving on Open Addressing Methods

To improve performance of open addressing methods, especially for unsuccessful searches, two models are introduced in this paper. Each model uses two flags in each bucket of the hash table to record hashing states (Figure 3-2). Flags of different models record different hashing information.

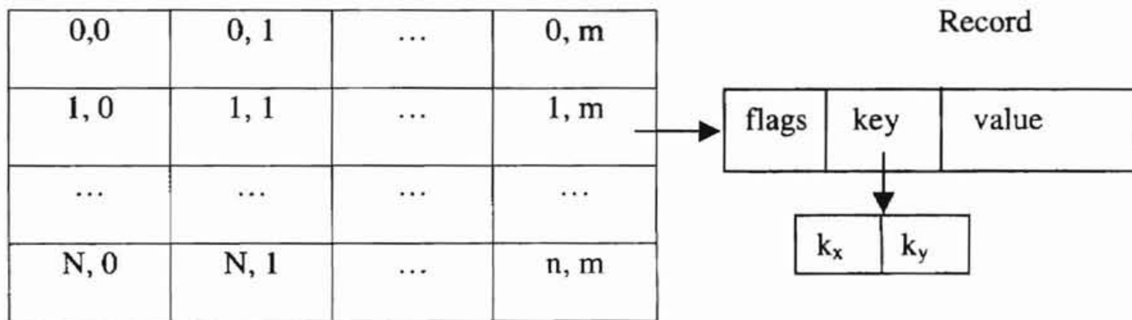
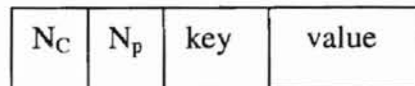



**Figure 3-2** 2DHT T[n,m] with Extra Memory flags

### 3.3.1 Model 1:




First, we introduce the concept priority hashing, in which any record that initially hashes to a bucket has priority to take the bucket over any record that must probe at least once after initial hash to reach it, even if such a record already inhabits the bucket. The displaced record probes forward to find a new location. If more than one record initially hashes to the same bucket, priority is assigned on the basis of the sequence in which the records appeared.

In Model 1, two flags $N_c$ and $N_p$ in each bucket are initialized to 0. $N_c$ represents the clustering factor (the number of Ks which initial by hashed to the same bucket); $N_p$ represents the number of probes of the record or key in the current bucket. Each bucket contains the following fields: $N_c$, $N_p$, key ($k_x$, $k_y$) for 2DHT, and other attributes. In addition, Model 1 requires priority hashing.

In searching for a record with key K, we compute the position T[i, j] to which K is first hashed. If T[i, j].Nc is zero, no record has been first hashed to this position T[i, j], and the search is unsuccessful. If T[i, j].$N_c$ is 1, the search is successful if and only if T[i, j].key = K. Moreover, if T[i, j].$N_c$ is more than one for a table with linear probing, we can often identify successful or unsuccessful searches based on the values of $N_c$ and $N_p$ before an empty bucket occurs or the whole table has been probed. Suppose P is the number of probes for the search key K. When P is one (for the first probe), search is successful just in case T[i;, j].key = K. When P is two (for the second probe), if P is equal to T[$i_2$, $j_2$].$N_p$, we compare K with T[$i_2$, $j_2$].key; search is successful in the case that K and T[$i_2$, $j_2$].key are equal; and so on up to the number of comparisons when K and T[$i_p$, $j_p$].key equals to T[i, j].$N_c$, where T[$i_p$, $j_p$] is the position where the record is probed P times. In this case, T[i, j].$N_c$ is the number of comparisons between K and T[$i_p$, $j_p$].key; the search is successful when K = T[$i_p$, $j_p$].key, or unsuccessful if K does not equal to T[$i_p$, $j_p$].key.

*Modification of Linear Probing with Model 1*

Algorithm L4 (Insertion of linear probing with Model 1)

Insertion with Model 1 requires priority hashing. In addition, $N_c$ (the number of keys that hashed to this bucket initially) increases by 1 after initial hash. Finally, when the target bucket is found, the number of probes is assigned to $N_p$.

**Linear-Insert-Model1(T, K)**

```
1.  i = h_x(k_x), j = h_y(k_y)
2.  count := 1                    //to record the probed number
3.  IF T[i, j].Np is zero THEN        //empty position
4.     insert K into T[i, j], and set T[i, j].Nc := 1, T[i, j].Np := 1
5.  ELSE
6.      IF T[i, j].Np is greater THAN 1 then       //without priority record
7.            temp = T[i, j]
8.            insert K into T[i, j], T[i, j].Nc := 1, T[i, j].Np := 1
9.            count := temp.Np
10.           K := temp.key
11.     ELSE
12.           count := 1
13.           T[i, j].Nc := T[i, j].Nc + 1
14.     LOOP: WHILE T[i, j].Np is not zero AND count < mn – 1.
15.           count := count + 1
16.           i := i + c
17.           IF i > m-1 THEN i := i – m, j := j + 1
18.           END IF
```

```
19.    END LOOP
20.    IF T[i, j].Np is zero THEN
21.        insert K into T[i, j]
22.        T[i, j].Np := count
23.    ELSE T[n, m] is overflow
```

Algorithm L5 (Search of linear probing with Model 1)

**Linear-Search-Model1(T, K)**

```
1. i := hx(kx), j :=hy(ky)
2. IF T[i, j].Nc is 0 THEN search is unsuccessful
3. ELSE
4.    flag := T[i, j].Nc
5.    LOOP: WHILE flag is greater then 0
6.        IF count equal to T[i, j].Np THEN  //T[i, j]. key and K are primary
7.            IF T[i, j].Key equal to K THEN  // clustering or same value
8.                search is successful
9.                break
10.           ELSE flage := flag – 1
11.       ELSE
12.           i := i + 1,
13.           IF i > m – 1 THEN i := i – m, j := j + 1 END IF
14.           count := count + 1              // count initial 1
15.   END LOOP
```

16. IF flag is greater 0 THEN search is successful

17. ELSE search is unsuccessful

Algorithm L6 (Deletion of linear probing with Model 1)

This is similar to Algorithm L3, except that Nc and Np in each position that undergoes deletion should be updated. In a hash table with open addressing, obviously, after deleting some keys, search results may not be accurate because of gaps. Without using extra memory to track the probing state of each record, updating the hash table after deletion is inefficient. However, with a little additional memory, the problems resulting from deletion can be resolved efficiently by updating the hash table.

**Linear-Delete-Model1(T, K)**

1. K is found at T[i, j]
2. delete K from T[i, j] // gap occurs
3. $g_x = i$, $g_y = j$ // for memorizing the gap position
4. i := i + c // c is step size
5. IF i > m – 1 THEN i:= i – m, j := j + 1
6. END IF
7. IF T[$g_x$, $g_y$].Nc > 1 THEN // first probe
8. T[$g_x$, $g_y$].Nc := T[$g_x$, $g_y$].Nc – 1
9. count := 2
10. LOOP: WHILE count != T[i, j].Np // to find the primary clustering with K
11. count := count + 1
12. i := i + c //c is step size

```
13.         IF i > m – 1 THEN i := i – m, j := j + 1
14.         END IF
15.   END LOOP
16.   insert T[i, j].key into T[gx, gy], T[gx, gy].Np := 1
17.   gx := i, gy := j
18. END IF
22. LOOP1: WHILE T[i, j] is not empty AND count != mn - 1
23.   i := i + c
24.   IF i >= m THEN i = i – m, j := j + 1// for next row
25.   END IF
26.   tx := i, ty := j                      // temporary position
27.   count2 := 0
28.   LOOP2: WHILE i != hx(key[tx, ty].kx) or j != hy(key[tx, ty].ky) // no first hashed in
29.         i := i – c
30.         IF i < 0 THEN i := m + i, j := j – 1  // back one row
31.         END IF
32.         count2 : = count2 + 1
33.         IF i = gx and j = gy THEN  break     // for inserting to gap
34.   END LOOP2
35.   IF i = gx and j = gy THEN
36.         T[gx, gy] := T[tx, ty]                   // move position to gap position
37.         T[gx, gy].Np := T[tx, ty].Np – count2
38.         delete T[tx, ty]
```

39. $g_x := i := t_x$, $g_y := j := t_y$ // new gap

40. ELSE $i := t_x$, $j := t_y$

41. count1 := count1 + 1 // count is the number of searching

42. END LOOP1

*Modification of double hashing with Model 1*

The modifications of Model 1 for double hashing are very similar to those for linear probing. Both methods (double hashing and linear probing) let $N_c$ and $N_p$ of Model 1 record the clustering factor for each bucket and the number of probes for each key, respectively. Since the offset of probing for double hashing varies with k, to guarantee m is not divisible by the offset, row-by-row probing is used in double hashing. In row-by-row probing, the next row is probed only after all positions of the current row have been probed, continuing until the whole table has been probed.

Algorithm D3 (Insertion of double hashing with Model 1).

**Double-Insert-Model1(T, K)**

1. $i := h_x^1(k_x)$, $j := h_y(k_y)$

2. IF T[i, j].$N_p$ is Zero THEN

3. temp.$N_c$ := 1

4. temp.$N_p$ := 1

5. ELSE

6. IF T[i, j].$N_p$ is 1 THEN

7. temp.Np := 1

8. $T[i, j].Nc := T[i, j].N_c + 1$

9. ELSE

10. temp := T[i, j]

11. T[i, j].Key := K //insert the key into the position

12. $T[i, j].N_c := 1$. $T[i, j].N_p := 1$ //set flags

13. END IF

14. LOOP: WHILE $T[i, j].N_p$ is not zero AND count is less than nm-1

15. $i := (i + h_x^2(k^x)) \bmod m$

16. j := j + count / m

17. count := count + 1

18. END LOOP

19. IF $T[i, j].N_p$ is zero THEN

20. T[i, j].Key := K

21. $T[i, j].N_c := temp.N_c$

22. $T[i, j].N_p := temp.N_p + count$

23. ELSE the table overflow

Algorithm D4 (Search of Double Hashing with Model 1)

**Double-Search-Model1(T, K)**

1. $i := h_x^1(k_x)$, $j := h_y(k_y)$

2 IF T[i, j].Nc is 0 THEN search is unsuccessful // no clustering

2. ELSE

3. flag := T[i, j].Nc

```
4.  LOOP: WHILE flag is greater then 0
5.      IF count equal to T[i, j].Np THEN  //T[i, j].key and K are primary
6.          IF T[i, j].Key equal to K THEN  // clustering or same value
7.              search is successful
8.              BREAK
9.          ELSE flage := flag – 1
10.     ELSE
11.         i := (i + h_x^2(k_x)) mod m      // keep probe in same row
12.         j := j + count / m
13.         count := count + 1               // count initial 1
14. END LOOP
15. IF flag is greater 0 THEN search is successful
16. ELSE search is unsuccessful
```

3.3.2 Model 2:

| $P_{min}$ | $P_{max}$ | key | value |
|---|---|---|---|

This model uses two integer fields $P_{min}$ and $P_{max}$ in each bucket. $P_{min}$ is the minimum number of probes and $P_{max}$ is the maximum number of probes among the keys that initially hashed to this position. So the number of probes for any key with initial probing position T[i, j] is between T[i, j].$P_{min}$ and T[i, j].$P_{max}$. Initially, all $P_{min}$ and $P_{max}$ fields are zero. Thereafter the values of $P_{min}$ and $P_{max}$ in any position are updated when keys hash initially to that position are inserted or deleted. Under linear probing, to search

for k row-by-row, the first probe position is T[(i + $P_{min}$*c) mod m, j], where c is an offset that is a constant for the linear probe or a variable with key for double hashing. For any key, the maximum number of probes past T[i, j] is $P_{max}$ - $P_{min}$ whether the search is successful or unsuccessful.

Algorithm L7 (Insertion of Linear Probing with Model 2)

**Linear-Insert-Model2(T, K)**

```
1.  i := h_x(k_x), j = h_y(k_y)
2.  temp_x  := i; temp_y := j
3.  count := 1;
4.  IF T[i,j].key is NULL THEN
5.        insert K into T[j, i].p_min := 1, T[j, i].p_max := 1
6.  ELSE
7.        LOOP:  WHILE T[j, i].key is not NULL and count <= mn
8.              i := i + c
9.              IF i is greater than OR equal to m THEN
10.                   i := i – m
11.                   j := (j + 1) mod n
12.             count := count + 1
13.        END LOOP
14.        IF T[j, i].key is NULL THEN
15.             insert K into T[j, i]
16.             IF count is less than T[temp_x, temp_y].p_min THEN
```

```
17.                 T[temp_x, temp_y].p_min := count
18.          IF count is greater than T[temp_x, temp_y].p_max THEN
19.                 T[temp_x, temp_y].p_max := count
20.      ELSE: overflow
```

Algorithm L8 (Search of Linear Probing with Model 2 )

**Linear-Search-Model2(T, K)**

```
1.  i := h_x(k_x),  j := h_y(k_y)
2.  min := T[j, i].p_min
3.  times := T[j, i].p_max – T[j, i].p_min
4.  LOOP1: FOR index 1 TO min – 1
5.     i := i +c
6.     IF i is equal to or greater than m THEN
7.           i := i – m
8.           j := (j + 1) mod n
9.   END LOOP1
10.  LOOP 2:  FOR index 1 TO times + 1
11.     IF T[j, i].key is K THEN
12.           found := 1
13.           BREAK
14.     i := i + c
15.     IF i is equal to or greater than m THEN
16.           i := i – m
17.           j := (j + 1) MOD n
```

18. count := count + 1

19. END LOOP2

20. IF found THEN successful search

21. ELSE unsuccessful search

The procedures of Model 2 for double hashing are similar to those of linear probing. The difference is that the offset for linear probing is the same constant for all keys, whereas for double hashing it is the value of the second hash function for the key in question. In double hashing, most keys with the same primary hash values, have different second hashing values. Thus, double hashing can avoid both primary and secondary clustering.

# CHAPTER IV

# HASHING PERFORMANCE

## 4.1 Expected Performance

### 4.1.1 Average Performances of Various Hashing Methods

The performance of hashing depends on the hashing function that distributes the set of keys into the hash table if the load factor $\alpha$ is fixed. In the worst case, the hash function hashes all n keys to the same slot, and performance is $\Theta(n)$. If the hash function initially distributes each of n keys into a unique fixed position in the hash table with m slots ($n < m$), however, the performance of an insertion or a search will be exact by1. In fact, average performance is much better than the worst case, but a little worse than the best case. Table 4-1 lists the average theoretical performance of successful and

**Table 4-1** Expression of probes expected for successful and unsuccessful search, as well unsuccessful search with improved Open Addressing in a hash table ([2] [7] [14]).

| *Methods* | *Unsuccessful* | *Successful* | *Improved* |
|---|---|---|---|
| Linear probe | $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$ | $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)}\right)$ | $\frac{1}{1-\alpha}$ |
| Double hashing | $\frac{1}{1-\alpha}$ | $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$ | $\frac{2}{\alpha}\ln\frac{1}{1-\alpha}-1$ |
| Chaining | $1+\alpha$ | $1+\frac{1}{2}\alpha$ | ——— |

Unsuccessful searches and the performance of unsuccessful searches improved for open addressing. These expected performance values assume uniform hashing [2], i.e., that any key is equally likely to hash to any slot in the hash table. From the expressions, it is easy to see that the performance of the various hash methods depend only on load factor α, not on table size. By providing reasonable values of α, we can calculate precise performance measures for the various methods. The results in Table 4-2 show various performance characteristics of different hashing methods.

**Table 4-2** The number of probes expected for successful, unsuccessful, and improved searches in a hash table (C: Chaining, L: Linear Probe, D: Double Probe, S: Successful Search, US: Unsuccessful Search, P: Improved Unsuccessful search).

| Load Factor | C-US | C-S | L-US | L-S | L-P | D-US | D-S | D-P |
|---|---|---|---|---|---|---|---|---|
| 0.20 | 1.20 | 1.10 | 1.28 | 1.13 | 1.25 | 1.25 | 1.12 | 1.23 |
| 0.30 | 1.30 | 1.15 | 1.52 | 1.21 | 1.43 | 1.43 | 1.19 | 1.38 |
| 0.40 | 1.40 | 1.20 | 1.89 | 1.33 | 1.67 | 1.67 | 1.28 | 1.55 |
| 0.50 | 1.50 | 1.25 | 2.50 | 1.50 | 2.00 | 2.00 | 1.39 | 1.77 |
| 0.60 | 1.60 | 1.30 | 3.63 | 1.75 | 2.50 | 2.50 | 1.53 | 2.05 |
| 0.70 | 1.70 | 1.35 | 6.06 | 2.17 | 3.33 | 3.33 | 1.72 | 2.44 |
| 0.80 | 1.80 | 1.40 | 13.00 | 3.00 | 5.00 | 5.00 | 2.01 | 3.02 |
| 0.90 | 1.90 | 1.45 | 50.50 | 5.50 | 10.00 | 10.00 | 2.56 | 4.12 |

### 4.1.2 Required Memory

One of the most important criteria for the performance of a data structure, memory requirements, differs for the two collision resolution techniques (chaining and open addressing). For open addressing, the required memory is constant: the table size multiplied by the memory occupied by one element. However, for external chaining, the required memory is a linear function of the load factor.

Assuming that the table size is T and that each element takes i words of memory, the memory required by the different hashing methods is shown in Table 4-3.

**Table 4-3** The Memory Requirements for Different Methods (T: Table size, i: memory occupied by one element, n: number of records).

| *Methods* | *Memory requirements* |
|---|---|
| Open addressing | T * I |
| Improved Open addressing | T * ( i + 1 ) |
| Chaining | T + n ( i + 1 ) |

## 4.2 Testing for Various Hashing Methods

This section presents testing results concerning actual performance of the search algorithms discussed in chapter 3. All of the algorithms tested were programmed by the author in standard ANSI C and tested on the Microsoft Visual C++ 6.0 compiler under Windows NT and the standard C++ compiler under SunOS 5.7. Performance of various methods based on two-dimensional hash tables was tested. Performance for higher dimensional hash tables can be inferred from that of 2DHTs.

### 4.2.1 Testing Procedures

Three things were considered while testing the various algorithms: load factor, table size, and test data (keys). In order to make comparisons, the same data were collected for testing various different methods with various table sizes and load factors.

Each method was tested with twelve of table sizes (8X4, 8X8, 16X8, 16X16, 32X16, 32X32, 64X32, 64X64, 128X64, 128X128, 256X128, and 256X256), ten load factors (from 10% to 100% by 10% increments), and with four sets of test data (keys). The four test data sets were produced by a random number generator to avoid duplicated keys. Each test data set was generated with a different seed. Within each set of test keys, each test key must be different from all others. Tests of successful and unsuccessful searches used test data in different amounts and of different values.

To test performance of successful search for each method, the amount and the values of test data (keys) should be the same as those that have been inserted in the table. The number of keys tested is the product of load factor and table size. However, to test performance of unsuccessful search for each method, all the keys tested are different from any data inserted into the table. The number of keys of each set depends on the table size. A set of fifty keys was used the 8X4 table, one hundred keys for 8X8, two hundred for 16X8 and 16X16, four hundred for 32X16, and five hundred for tables at 32X32 and over.

### 4.2.2 Test Results

To facilitate analysis and comparison of performance both within and across algorithms, all test results are listed in the tables in Appendix A. In Tables A-1 to A-14, each number represents an average successful or unsuccessful search time per key for a specified algorithm, hash table size, and load factor. In Tables A-15 and A-16, each number is an average number of search probes for a single key for one algorithm and one kind of hash table size when load factor is less than or equal to 90%, and total average

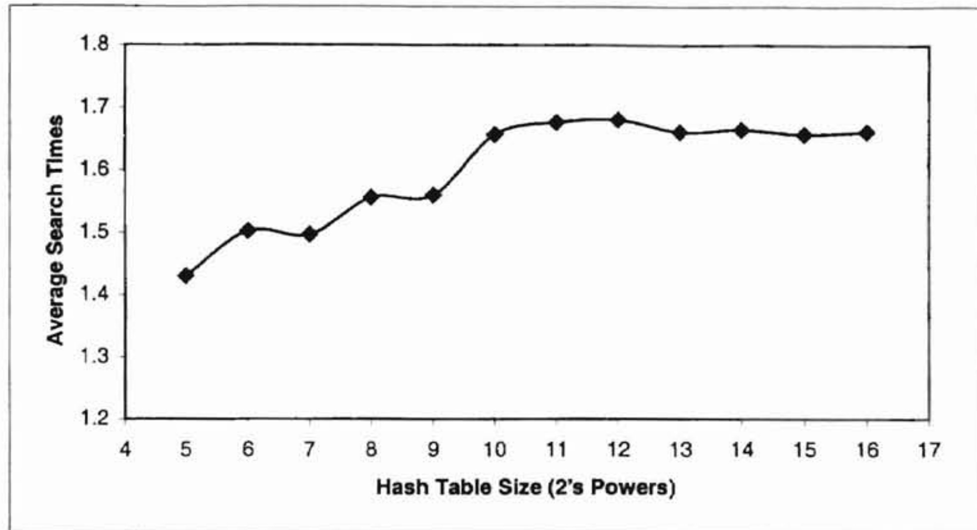



**Figure 4-1** Average Number of Successful Search with Load Factor Under 0.9

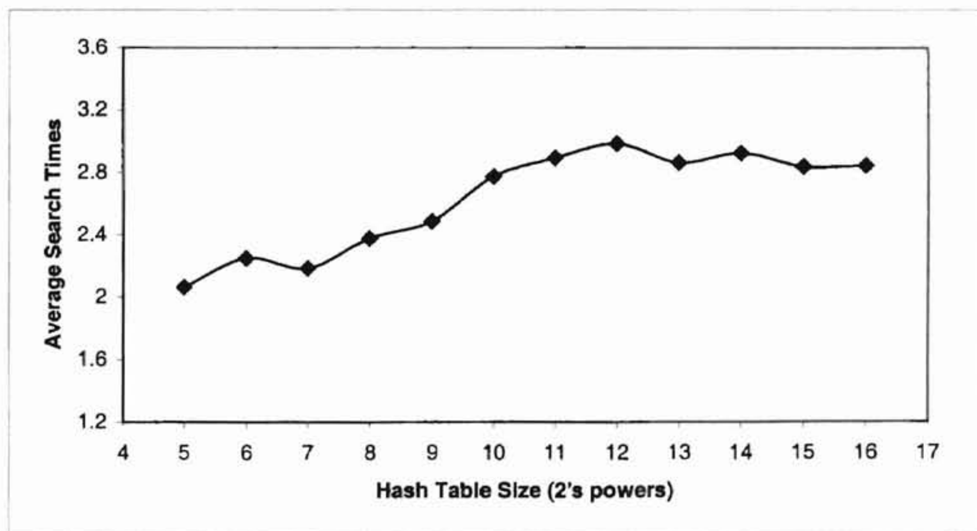



**Figure 4-2** Average Number of Unsuccessful Search With Load Factor Under 0.9

performance in searching a single key for all the methods with various hash table sizes. These tables show that actual search performance, like theoretical performance, does not depend on hash table size, especially for the chaining method. But with open addressing hashing, when the table size is less than 32X32, search probes increase with table size, whereas table size no longer affects performance for 32X32 or larger tables. Figure 4-1 and Figure 4-2 show the relationship between average search time and table size. Since performance is not stable with small hash tables, to increasing accuracy, all data used for analysis in this research are obtained from testing with hash tables whose size is greater than 32X32.

4.2.2.1 Chaining Hashing

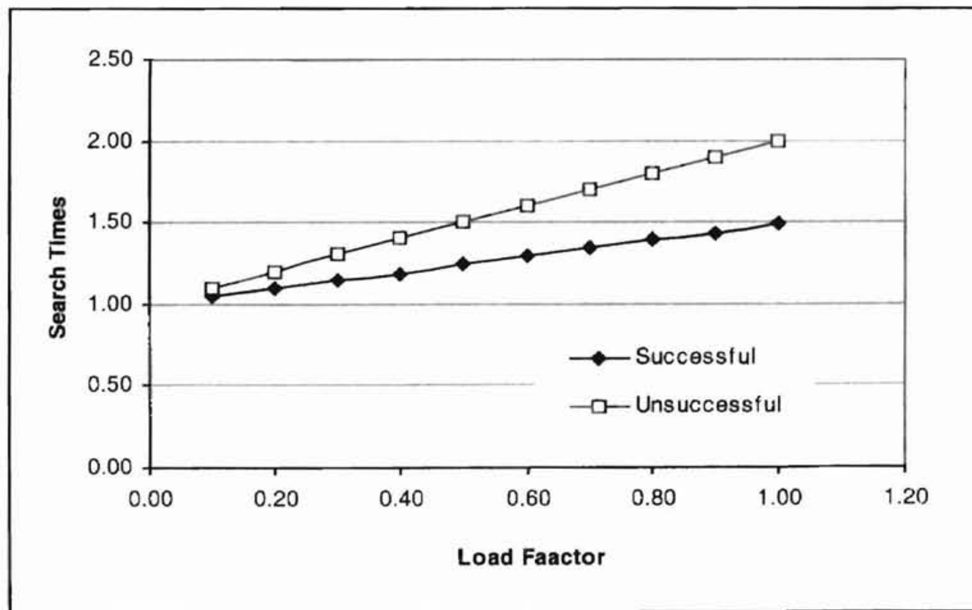



**Figure 4-3** Search in Chaining Hash Table

The test results (Table A-1 and Table A-2 in Appendix A) of Algorithm $C_1$ (Chaining hash table insertion) and Algorithm $C_2$ (Chaining hash table search) indicate the following performance characteristics. The average search times of each key does not vary with table size. However, search times increased with load factor. For successful searches, average search times are from 1.0444 to 1.4906 corresponding to load factors from 10% to 100%, respectively. For unsuccessful searches, the range of average search times is from 1.1014 to 2.0004. Figure 4-3 shows that search times for chaining hash are a linear function of load factor.

4.2.2.2 Linear Probing

Tables A-3 and A-4 in Appendix A list successful and unsuccessful search performance of Algorithm L2 (Linear Probing Search). Successful and unsuccessful search performance improved by Model 1 is listed in Table A-5 and Table A-6. Improved performance by Model 2 can be found in Table A-7 and Table A-8.

Without improvement, successful and unsuccessful search performance of Algorithm $L_1$ (Linear Probing Insertion) and Algorithm $L_2$ (Linear Probing Search) is listed in Table A-3 and Table A-4 of Appendix A. When the hash table is not full (load factor under 90%), search performance, whether successful or unsuccessful, is not affected by table size, but increases load factor as shown on Figure 4-4. However, when the hash table is full (load factor is 1), the performance of search, especially for unsuccessful searches, changes with table size.

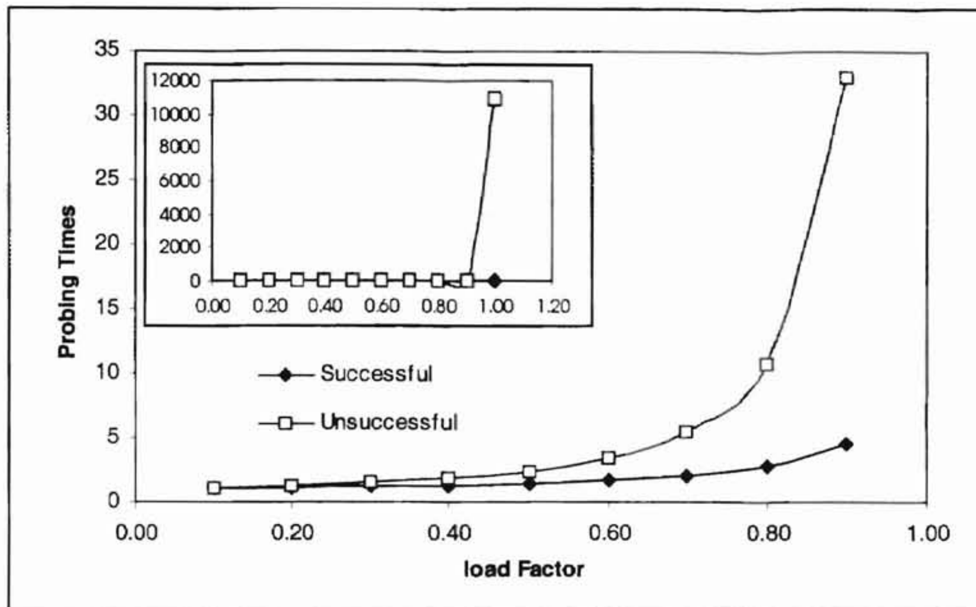



**Figure 4-4** Search in Linear Probing Hash Table

Table A-5 and Table A-6 in Appendix A list the results of successful and unsuccessful searches with linear probing improved by Model 1; Table A-7 and Table A-8 are the results of performance of linear probing improved by Model 2. Model 1 does not improve successful search at all, whereas Model 2 shows some improvement for linear probing (Figure 4-5). Figure 4-6 shows the improvement by Model 1 and Model 2 for unsuccessful search by linear probing. Both Model 1 and Model 2 give excellent improvement for unsuccessful search, especially when the hash table is full. Average unsuccessful search times for linear probing without improvement is18578.2 (in Table A-4), 59.38 with Model 1 improvement (shown in Table A-6), and 33.05 with Model 2 improvement (Table A-8).

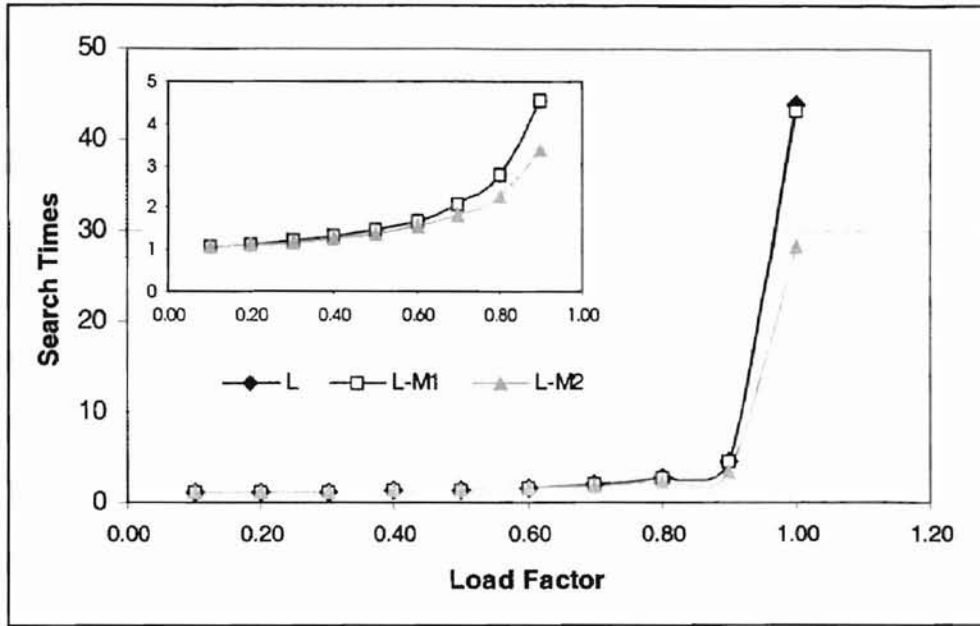



**Figure 4-5** Successful Search with Linear Probing

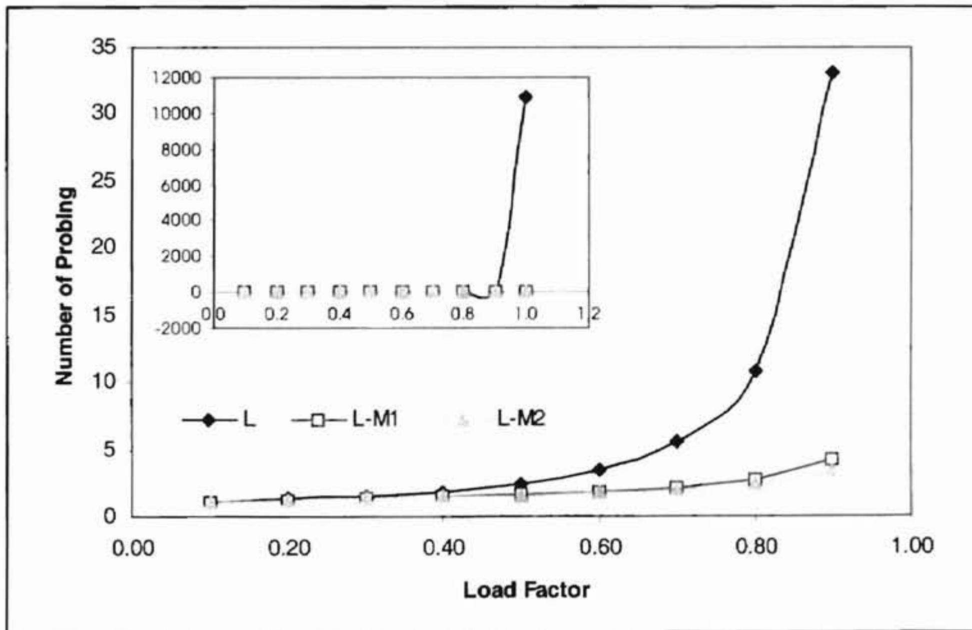



**Figure 4-6** Unsuccessful Search with Linear Probing

4.2.2.3 Double Hashing

Table A-9 and Table A-10 in Appendix A show the results of successful and unsuccessful search performance for double hashing; Table A-11 and Table A-12 list the results of improved double hashing with Model 1, and Table A-13 and Table A-14 show the results of improved double hashing with Model 2. Performance of double hashing is not affected by table size when the table size is at least 32X32, but is affected load factor.

Figure 4-7 shows that the average number of searches with double hashing increases nonlinearly with load factor, especially for unsuccessful search.

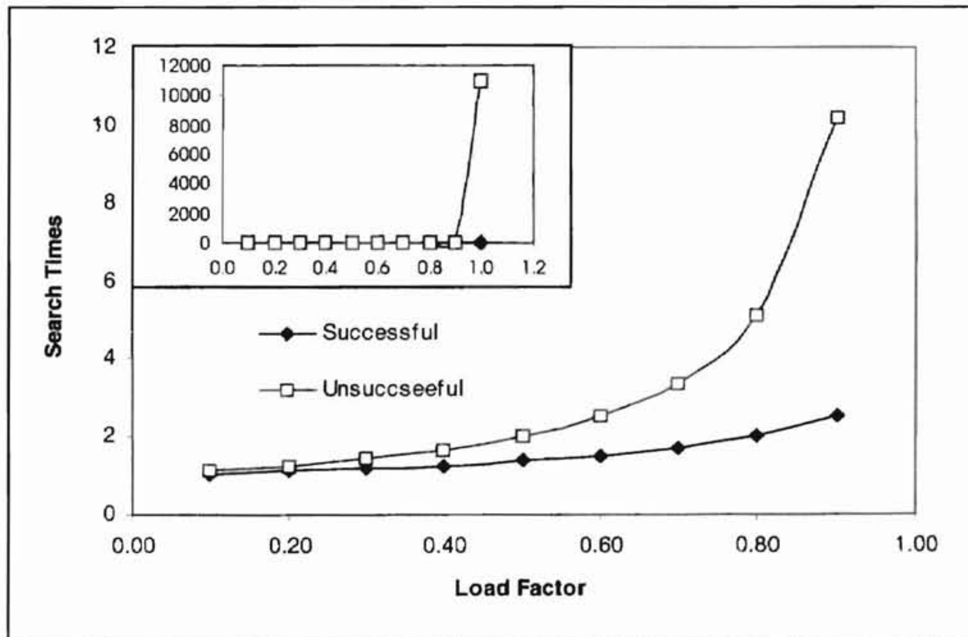



**Figure 4-7** Search in Double Hash Table

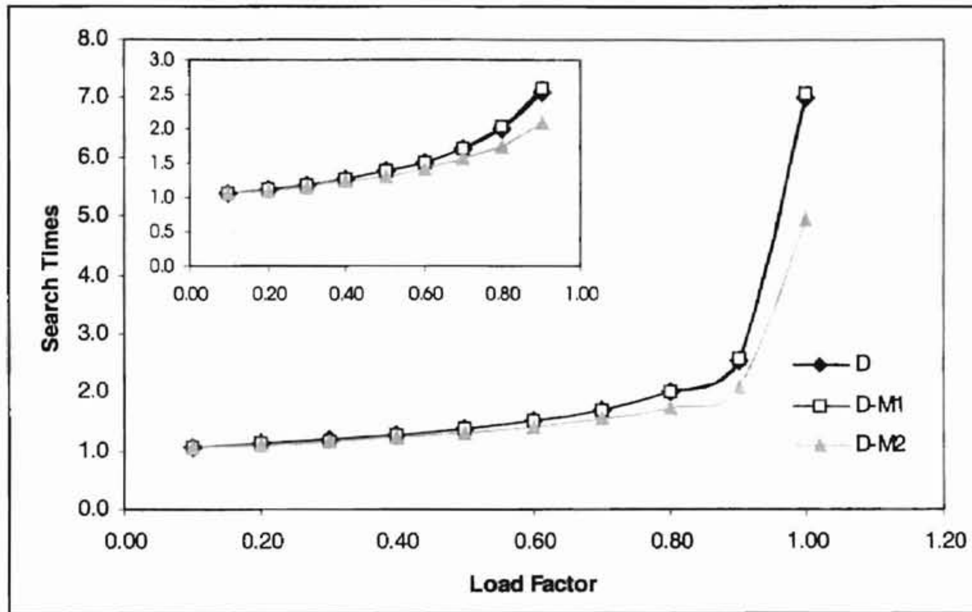



**Figure 4-8** Successful Search with Double Hashing

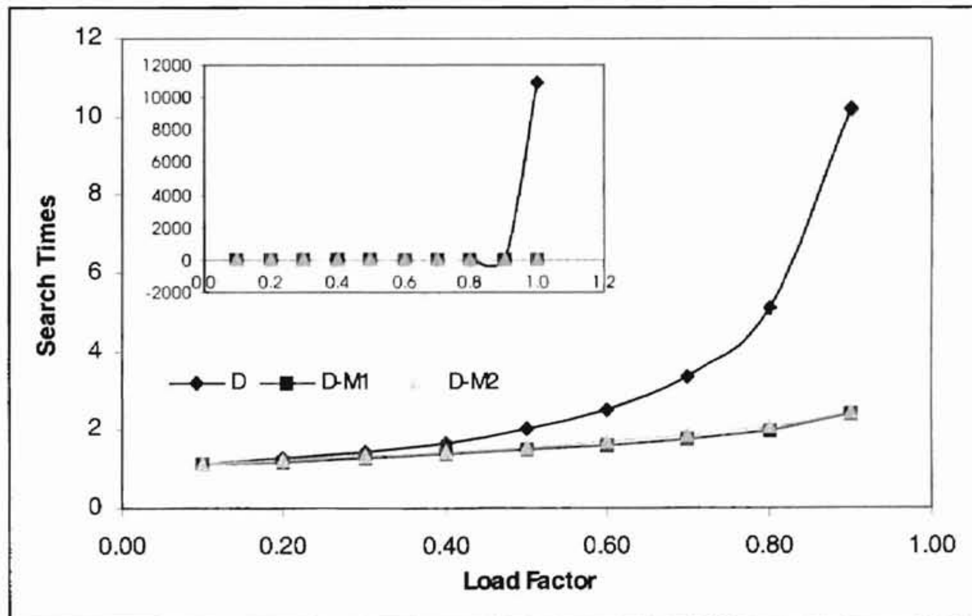



**Figure 4-9** Unsuccessful Search with Double Hashing

Performance of double hashing improved by Model 1 and Model 2 is shown in Figure 4-8 for successful search and Figure 4-9 for unsuccessful search. As with linear probing, Model 1 does not improve the performance for successful search by double hashing and degrades the performance for successful search when load factor is greater than 0.6. Model 2 can improve the performance for successful search by double hashing, and the rate by which performance improves increases with load factor. When load factor is 0.9, performance is improved by 18%. However, both Model 1 and Model 2 improve performance of unsuccessful search, and improvement increases nonlinearly with load factor. When load factor is 0.9, search times improve by 78 percent of over performance of unsuccessful searches by unimproved double hashing. When load factor is less than 0.9, Model 1 is a little better than Model 2 for an unsuccessful search by double hashing.

## 4.3 Analysis and Comparison

Table 4-4 and Table 4-5 summarize the various hash methods' performance on successful and unsuccessful searches. The results of testing show that chaining performs best among the various hashing methods; in addition, separated chaining does not require contiguous memory for chains. However, chaining needs more memory than open addressing; in addition, chaining must use pointers to complete various operations.

Among open addressing hash methods, double hashing proves for better than linear probing in terms of search performance, because double hashing eliminates both primary and secondary clustering. In linear hashing, it is easy to set up an offset that is relatively prime with table size of one row for 2DHT. Unlike linear probing, in double

**Table 4-4** Average Number of Successful Search from Testing

| % | Ch | L | D | L-M1 | D-M1 | L-M2 | D-M2 |
|---|---|---|---|---|---|---|---|
| 0.10 | 1.0444 | 1.0519 | 1.0511 | 1.0519 | 1.0509 | 1.0499 | 1.0492 |
| 0.20 | 1.0904 | 1.1222 | 1.1143 | 1.1222 | 1.1135 | 1.1137 | 1.1057 |
| 0.30 | 1.1416 | 1.2111 | 1.1858 | 1.2111 | 1.1858 | 1.1903 | 1.1680 |
| 0.40 | 1.1874 | 1.3336 | 1.2730 | 1.3336 | 1.2733 | 1.2913 | 1.2370 |
| 0.50 | 1.2393 | 1.4974 | 1.3801 | 1.4972 | 1.3798 | 1.4121 | 1.3163 |
| 0.60 | 1.2896 | 1.7438 | 1.5198 | 1.7428 | 1.5205 | 1.5893 | 1.4170 |
| 0.70 | 1.3367 | 2.1853 | 1.7242 | 2.1809 | 1.7296 | 1.8947 | 1.5617 |
| 0.80 | 1.3906 | 3.0072 | 2.0229 | 2.9986 | 2.0332 | 2.4498 | 1.7596 |
| 0.90 | 1.4337 | 5.3654 | 2.5896 | 5.3357 | 2.6059 | 3.9970 | 2.1170 |
| 1.00 | 1.4906 | 69.9121 | 8.8522 | 68.5022 | 8.9408 | 44.4590 | 6.0920 |

**Table 4-5** Average Number of Unsuccessful Search from Testing

| % | Ch | L | D | L-M1 | D-M1 | L-M2 | D-M2 |
|---|---|---|---|---|---|---|---|
| 0.10 | 1.1014 | 1.1191 | 1.1153 | 1.1033 | 1.0988 | 1.1029 | 1.1034 |
| 0.20 | 1.2001 | 1.2781 | 1.2526 | 1.2051 | 1.1859 | 1.2032 | 1.2014 |
| 0.30 | 1.2994 | 1.5217 | 1.4384 | 1.3213 | 1.2723 | 1.3152 | 1.3071 |
| 0.40 | 1.4030 | 1.9030 | 1.6791 | 1.4587 | 1.3637 | 1.4433 | 1.4186 |
| 0.50 | 1.5059 | 2.4932 | 2.0157 | 1.6165 | 1.4606 | 1.5752 | 1.5331 |
| 0.60 | 1.6040 | 3.6271 | 2.5137 | 1.8158 | 1.5728 | 1.7427 | 1.6711 |
| 0.70 | 1.7029 | 6.1520 | 3.3789 | 2.1826 | 1.7221 | 2.0366 | 1.8369 |
| 0.80 | 1.8022 | 12.7370 | 5.1298 | 2.8720 | 1.9348 | 2.5229 | 2.0551 |
| 0.90 | 1.8972 | 44.0295 | 10.5758 | 4.7336 | 2.3584 | 3.7540 | 2.4180 |
| 1.00 | 2.0004 | 18578.3 | 18578.3 | 59.3806 | 8.3818 | 33.0554 | 4.8982 |

hashing, the value of h2(K) for every key must be set to be a prime compare table and row size with increment size. This is important because if the table or row size m and offset h2(K) for double hashing (or c for linear probing) have a common divisor $d > 1$ for some key K, then a search for key K would search only 1/d of the hash table. The

convenient way to resolve this problem is to let m be a power of 2, and to design h2 so that h2(K) is always odd.

To improve performance for open addressing hash methods, this research proposes Model 1 and Model 2. Comparing the Models, Model 2 proves highly effective for linear probing and double hashing, because it can improve performance not only for unsuccessful search, as model 1 does, but also improve for successful search. Although Model 1 cannot improve performance for successful search, it may be very useful for special applications. Because Model 1 has a flag $N_c$ recording the number of keys that are initially hashed to that bucket, and uses priority hashing (the key with initial probe will take over the bucket occupied by a key with more than one probe), it can retrieve a record by a known index if the record with the initial probe has particular significance. Moreover, Model 1 is better than Model 2 for unsuccessful search with double hashing.

Comparing Table 4-2, containing expected performance results, and Table 4-4 and Table 4-5, containing the results of practical performance for all the hashing methods, we see that theoretical and actual performances are very similar. Thus, the practical test supports the algorithms discussed in chapter 3.

# CHAPTER V

# APPLICATION IN GRIDDING

## 5.1 Contour Map System

Contour Map System (CMS) involves three basic operations: Hashing, Gridding, and Visualizing (Figure 5-1). The visualization module reads in regular matrices and uses them to produce contour maps; but most data collections are not uniform. Especially in the natural sciences, observations are usually scattered irregularly across the map area.

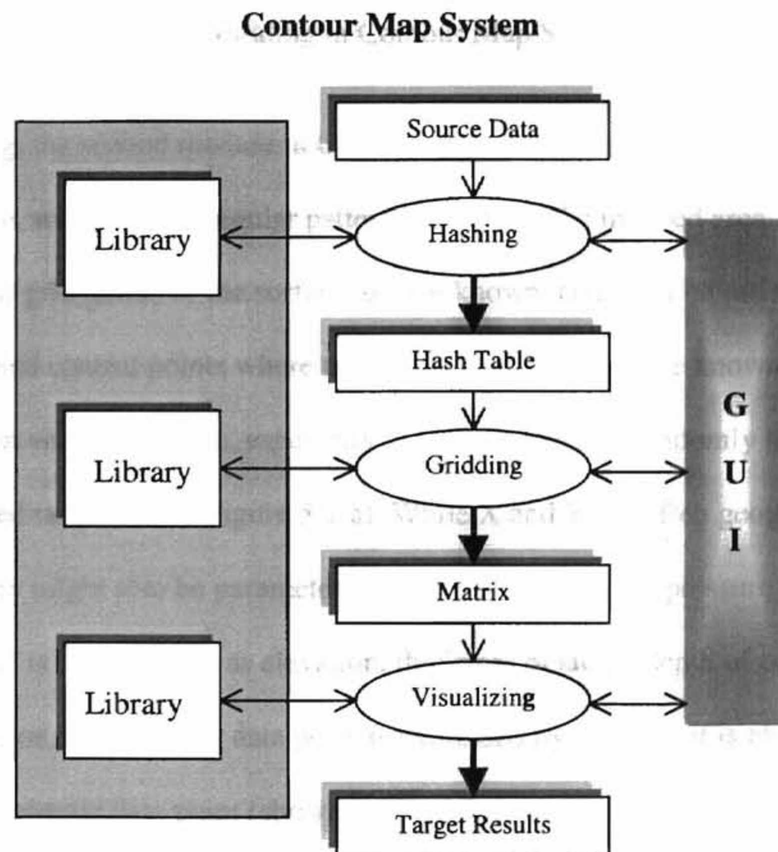



**Figure 5-1** Contour Map System

In the CMS visualization module, as in most other contouring programs, graphic displays can be made only on regular points in every matrix grid. Therefore, in CMS, the second module, the gridding module, creates such numerical matrices from irregularly distributed data. In fact, the module that generates a regular grid matrix from scattered data points is the most important procedure in the graphics package. The first module contains a set of algorithms for loading the sample data into a data structure that lets gridding find efficiently. Since this work concentrates on multi-dimensional hash tables, we refer to the first module as Hashing. The performance of the first module directly affects the efficiency of the second.

## 5.2 Gridding in Contour Map System

Gridding, the second module in CMS, is the estimation of values of the surface at a set of locations arranged in a regular pattern that covers the mapped area. In general, the values at regular grid points of the surface are not known, and must be estimated from irregularly located control points where the values of the surface are known. Known spatial data from various surveys, especially in GIS, consists of randomly located X-Y-Z values with fixed ranges (as in Figure 5-2 a). While X and Y are often geographic coordinates, they might also be parameters such as temperature or pressure for other kinds of maps. Z is a value such as elevation, thickness of stone, depth of ocean, saltiness of water, and so on. Each spatial data point is expressed by (x, y, z); it is also called a control point or sample data point (shown in Fig 5-2 a).

The grid points (or nodes) are usually arranged in a square pattern (shown in Figure 5-2 b). The spacing is under user control, and is one of many parameters that must

be chosen before gridding. The area enclosed by four grid nodes is called a grid cell. If a large size is chosen for the grid cell, the resulting map will have low resolution, but can be computed quickly. Conversely, if the grid cells are small, the contour map will have high resolution, but will take more running time and will be expensive to produce.

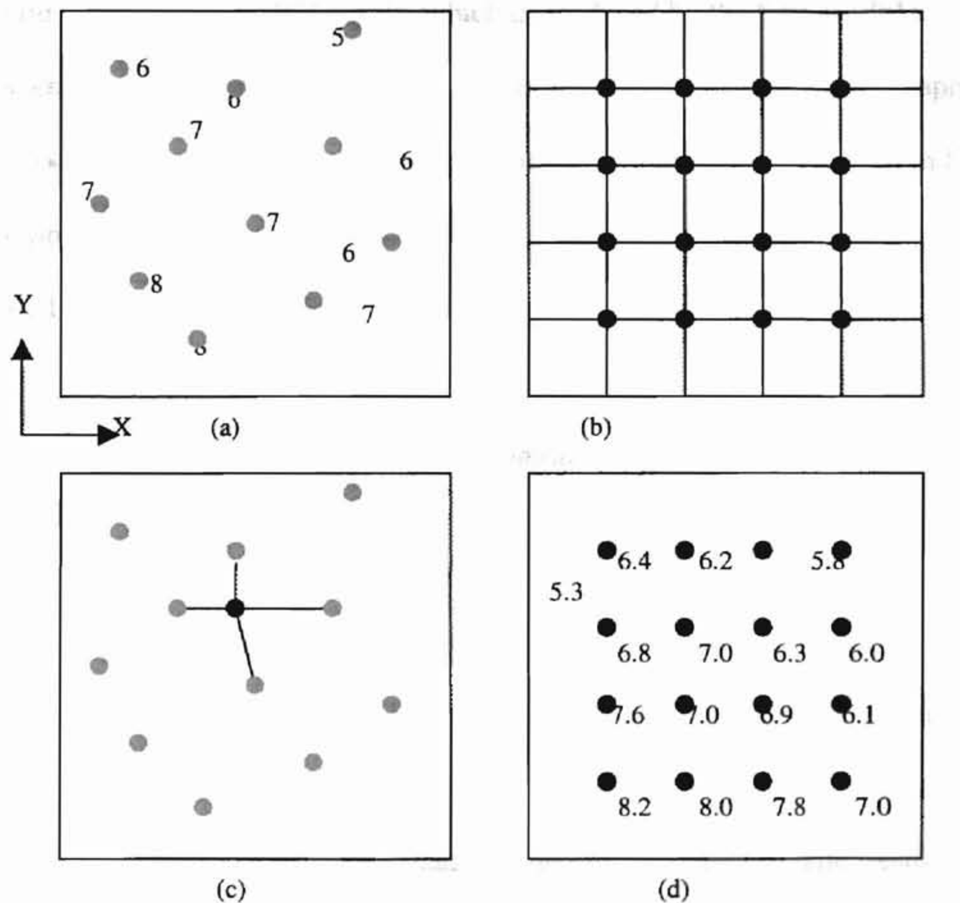



**Figure 5-2** Procedure of Gridding

Gridding in CMS generates a grid matrix of estimated Z values for regularly spaced X and Y values from irregularly spaced X, Y, and Z sample data points in three essential steps [10]. The first step is determining the matrix size and grid cell size. The

grid matrix size (number of grid nodes) depends on the number of control data points. Second, the user must choose the mathematical function to use in estimating grid values [18]. In the third step, users choose search algorithms. Both the search procedure and the mathematical function have significant effects on the performance of CMS; search methods are especially important. What kind of search method is efficient depends on the data structure used to hold control points, which is produced by the first module.

The estimation process estimates the values for every grid node in the mapped area. Each node is estimated from a collection of nearby control points (shown in Figure 5-2 c). The procedure is repeatedly applied across the map area until the whole map area is represented by regular grids (Figure 5-2 d).

## 5.3 Hashing in Contour Map System

In order to improve the performance of gridding in CMS for control data points that are scattered in a near random fashion, we propose two-dimensional hashing as the data processing method in the first module. The near random nature of the data, given a reasonable hash function, produces nearly uniform hashing. Hence the first module produces a 2DHT holding all the control data for the gridding module. The hashing module proceeds in two major steps. The first defines hash table size; the second selects a hashing method. The hash table size depends on the number of control points in the map area, the grid matrix size, and the maximum effective distance (radius) from the grid of its control points. Since the load factor is always fixed and is usually between 0.5 and 0.8, hash table size is $K/\alpha$, where K is the number of control points and $\alpha$ is the load factor. Therefore, hash table size ranges between K/0.5 and K/0.8, and improved double hashing

is recommended both for loading data into the hash table and for searching in the gridding module.

5.3.1 Hashing Function

Since search focuses on control data at an exact location on map area, (as opposed to range searches, for instance), the hashing function chosen must hash control points for the same grid cell into the same bucket in the hash table as shown in Figure 5-3. For example, data located in the (20 … 40, 10 … 40) field should be in [0, 0] in hash table, data in the (40 … 60, 40 … 70) field will be in [1, 1], and so on. This produces a result similar to a grid directory (reviewed in chapter 2). But the methods handle data clustering in different ways. To resolve the clustering problems, the grid file method split the cell; however, the hashing method probes the key repeatedly until an empty bucket appears.

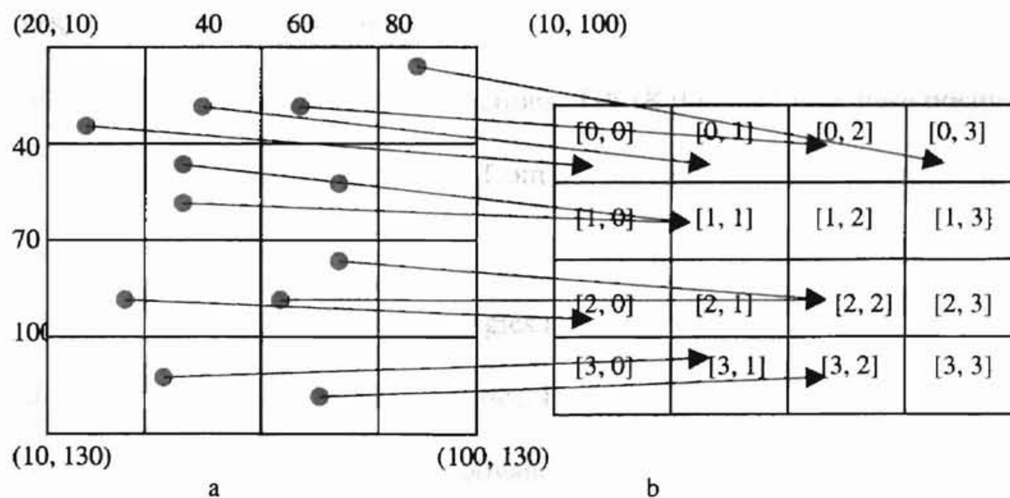



**Figure 5-3** Hashing Sample Data Point to Hash Table: a. map area (X: 20 … 100, Y: 10 … 120), b. two dimensional hash table

Division Method:

$$h(K) = h(k_x, k_y) = ((h_x(k_x), h_y(k_y)) \qquad (5.1)$$

$$h_x(k_x) = \lfloor \lfloor K.x - \rfloor / \lfloor X_{max} - X_{min} \rfloor * m \rfloor \bmod m$$

$$h_y(k_y) = \lfloor \lfloor K.y - Ymin \rfloor / \lfloor Y_{max} - Y_{min} \rfloor * n \rfloor \bmod n$$

K.x is the X coordinate value and K.y is the Y coordinate value of the control data K. $X_{min}$ and $X_{max}$ are the minimum and maximum X coordinate values in the map area; $Y_{min}$ and $Y_{max}$ are minimum and maximum Y coordinate values. Thus the map area is ($X_{min} \ldots X_{max}$, Ymin … $Y_{max}$).

5.3.2 Collision Solutions

To avoid primary and the secondary clustering, we choose double hashing to resolve collisions in CMS. Double hashing uses a general hashing function of the following form [2]:

$$h(K, i) = (h_1(K) + ih_2(K)) \bmod m \qquad (5.2)$$

where $h_1$ and $h_2$ are auxiliary hashing functions. $T[h_1(K)]$ is the initial hash position; $h_2(K)$ is the offset for successive probing from the previous position; and m is the hash table size.

CMS uses collision solution strategies and clustering control for two-dimensional hash tables based on double hashing theories. However, to reduce the calculation times of division or multiplication, which are disadvantages in the hashing method, after first probing with X and Y values, only the X value of each key will be calculated by the double hashing method, as in Algorithm D1 in chapter 3. Therefore, hashing sequences

probe row by row. The specific hashing functions for two-dimensional hash tables referred to in functions (5.1) and (5.2), thus, are as followings:

$h(K, i) = ( h_x(K.x, i), h_y(K.y, i))$

$h_x(K.x, i) = (h_x1(K.x) + i\ h_x2(K.x)) \bmod m$

$h_x1(K.x) = \lfloor \lfloor K.x - Xmin \rfloor / \lfloor X_{max} - X_{min} \rfloor * m \rfloor \bmod m$

$h_x2(K.x) = \lfloor K.x \rfloor \bmod m'$

$h_y(K.y, i) = (h_y1(K.y) + \Sigma\ h_y2(K.y, i)) \bmod n$

$h_y1(K.y) = \lfloor \lfloor K.y - Ymin \rfloor / \lfloor Y_{max} - Y_{min} \rfloor * n \rfloor \bmod n$

$h_y2(K.y, i) = 1 \quad // (\text{if } h_x1(K.x) + ih_x2(K.x) >= m)$

$\quad\quad = 0 \quad // (\text{if } h_x1(K.x) + ih_x2(K.x) < m)$

This hashing method uses double hashing only for X value of each key. After the Y value is hashed initially with ($h_y1(K.y)$, $h_y2(K.y)$, Y only increases by 1 or 0 depending on how many times the key is probed. In addition, there are two kinds of probing sequences in 2DHT: one, after all buckets of the current row are completely probed, goes to the next row by assigning 1 to $h_y2(K.y, i)$; the other goes to the next row when $h_x(K.x, i-1) + h_x2(K.x)$ is greater than or equal to m.

### 5.4 Analysis and Comparisons

A huge amount of search takes place during gridding. Each grid point usually needs to search for at least four control points. To find control points for a grid point, the X and Y values of the grid point should be converted into interval indexes that determine which bucket in the hash table contains the target control points. The second step checks

whether the record in that target bucket initially hashed to that bucket by using the hashing functions to test the two attributes X and Y in the target bucket. There are three logical possibilities. If the bucket is empty, search has failed. If the record in the bucket initially hashed to that position, search has succeeded. However, if it did not, we do not yet know whether any sample data is located at the area corresponding that bucket. This logical possibility would pose a problem for gridding. Fortunately, in chaining hash tables, that problem can not occur. For open addressing hash methods, whether by linear probe or double hashing, adopting the Model 1 improvement strategy also eliminates that problem. Model 1 uses priority hashing (any record with initial hash to a bucket will replace any record that had to probe to reach it), and a flag ($N_c$) records the number of records having initial hash at this bucket. When a target bucket is determined, we can learn whether search is successful by simply checking the value of the flag $N_c$ in the target bucket. If $N_c$ is 0, the search is fails; otherwise, it succeeds. In addition, the value tells directly how many control points are in the target bucket's region.

# CHAPTER VI

## SUMMARY AND CONCLUSION

MDHTs strongly resemble one-dimensional hash tables in terms of hashing operations. Two-dimensional hash tables implement the Insert and Search operations in constant average time. When using hashing tables, it is important to pay attention to choose an appropriate load factor and to choose a hash function that produces nearly uniform hashing.

Although chaining hashing requires more memory than open addressing methods, it has excellent performance. Its load factor can be large enough to provide some space efficiency, and it can use fragmentary memory, unlike open addressing, which requires a single contiguous block.

In open addressing hashing methods, the load factor should not be greater than 80%. The performance of double hashing is much better than linear probing. Two models presented in chapter 3 can improve time performance significantly at some cost in space for unsuccessful search. Model 2 can also improve performance of successful search for open addressing.

A two-dimensional hash table can be used to implement insert and search operations for spatial data with two keys, X and Y, recording the information. Using 2DHTs, it is possible to organize the spatial data in a way that facilitates other types of processing, such as sequential processing.

In this research, an application model the Contour Map System (CMS), employs a 2DHT for data processing, and with Model 2, it can improve performance over O(ng) for

traditional sequential processing to O(g) of 2DHT, where n is the number of spatial data points and g is the number of grid nodes in the grid matrix required by CMS.

## REFERENCES


[1] Breu, H., Gill, J., Kirkaptrick, D., and Werman, M., "Linear time Euclidean distance transform algorithms", IEEE Trans. Pattern Analysis and Machine Intelligence, Vol. 17, No. 5, 1995.

[2] Cormen, T. H., Leiserson, C. E., and Rivest, R. L., Introduction To Algorithms, Cambridge, MA: MIT Press, 1990.

[3] Foley, J. D., Dam, A. V., Feiner, S. K., Hughes, J. F., Computer Graphics Principles and Practice (2th ed.), Reading, MA: Addison-Wesley, 1990.

[4] Folk, M. J. and Zoellick, B., File Structures (2th ed.), Reading, MA: Addison-Wesley, 1992.

[5] Hutflesz, A., Six, H. W., Widmayer, P., "Twin Grid Files: Space Optimaizing Access Schemes", *ACM*, Vol. 6, pp.183-190, 1988.

[6] Kashyap, R. L., Subas, S. K. C., and Yao, S. B., "Analysis of the multi-attribute tree database organization", IEEE Trans. Software Engineering, Vol. 2, No. 6, Nov. 1977.

[7] Knuth, D. E., The Art of Computer Programming (2nd ed.), Vol. 3, Sorting and Searching Edition, Reading, MA: Addison Wesley, 1997.

[8] Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing", Proc. 6th International Conference on Very Large Data Bases, pp. 212-223, 1980.

[9] Lomet, D. B., Salzberg, B., "A Robust Multi-Attribute Search Structure", IEEE, pp. 296-304, 1989.

[10] Maltman, A., Geological Maps: An Introduction (2th ed.), New York: John Wiley & Sons, 1998.

[11] Nievergelt, J., Hinterberger, H., Sevcik, K. C., "The Grid File: An adaptalbe, Symmetric Multikey File Structure", ACM Transactions on Database System, Vol. 9, No.1, pp. 38-71, 1984.

[12] Patel, H. D., Analysis and Comparison of Extendible hashing and $B^+$ Trees Access Methods, M. Sc. Thesis, Oklahoma State University, Stillwater, OK, 1987.

[13] Ramakrishna, M. V., "Hashing in Practice, Analysis of Hashing and Universal Hashin", *ACM*, Vol. 6, pp.191-199, 1988.

[14] Tenenbaum, A. M., Augenstein, M. J., Data Structures Using Pascal, New Jersey: Prentice-Hall, 1995.

[15] Thompson, J. F., Soni, B. K., and Weatherill, N. P., Handbook of Grid Generation, New York: CRC Press, 1999.

[16] Troof, H., Herzog, H., "Multidimensional Range Search in Dynamically Balanced Trees", *Angewandte Informatik,* Vol. 2, pp. 71-77, 1981.

[17] Weiss, M. A., Data Structures and Algorithm Analysis in C (2th ed.), Menlo Park, CA: Addison-Wesley, 1997

[18] Zoraster, S., "Imposing Geologic Interpretations on Computer-Generated Contours Using Distance Transformations", *Mathematical Geology*, Vol. 28, No. 8, pp. 969-985, 1996.

APPENDIX A

# TABLES OF TESTING RESULTS

**Table A-1** Successful Search with Chaining Hashing

| % | 8X4 | 8X8 | 16X8 | 16X16 | 32X16 | 32X32 | 64X32 | 64X64 | 128X64 | 128X128 | 256X128 | 256X256 | **Avg** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.0833 | 1.0417 | 1.0417 | 1.0300 | 1.0147 | 1.0294 | 1.0453 | 1.0501 | 1.0485 | 1.0508 | 1.0476 | 1.0493 | **1.0444** |
| 0.20 | 1.1667 | 1.0417 | 1.0500 | 1.0735 | 1.0686 | 1.0858 | 1.0960 | 1.1050 | 1.0971 | 1.1001 | 1.1005 | 1.0995 | **1.0904** |
| 0.30 | 1.1667 | 1.1053 | 1.1382 | 1.1217 | 1.1275 | 1.1523 | 1.1421 | 1.1486 | 1.1499 | 1.1474 | 1.1495 | 1.1499 | **1.1416** |
| 0.40 | 1.1667 | 1.1300 | 1.1912 | 1.1838 | 1.1801 | 1.1993 | 1.2045 | 1.1954 | 1.1999 | 1.2001 | 1.1987 | 1.1993 | **1.1874** |
| 0.50 | 1.1875 | 1.2344 | 1.2227 | 1.2500 | 1.2461 | 1.2417 | 1.2493 | 1.2472 | 1.2456 | 1.2485 | 1.2474 | 1.2507 | **1.2393** |
| 0.60 | 1.2368 | 1.2895 | 1.2697 | 1.2843 | 1.3192 | 1.2936 | 1.2911 | 1.3011 | 1.2940 | 1.2978 | 1.2989 | 1.2994 | **1.2896** |
| 0.70 | 1.2500 | 1.3239 | 1.3202 | 1.3408 | 1.3596 | 1.3614 | 1.3414 | 1.3514 | 1.3472 | 1.3471 | 1.3482 | 1.3493 | **1.3367** |
| 0.80 | 1.2800 | 1.4412 | 1.3799 | 1.3836 | 1.4022 | 1.4145 | 1.3933 | 1.4017 | 1.3961 | 1.3972 | 1.3977 | 1.3992 | **1.3906** |
| 0.90 | 1.3750 | 1.3947 | 1.4239 | 1.4152 | 1.4560 | 1.4598 | 1.4445 | 1.4450 | 1.4470 | 1.4450 | 1.4490 | 1.4496 | **1.4337** |
| 1.00 | 1.4688 | 1.4609 | 1.4883 | 1.4873 | 1.4956 | 1.5056 | 1.4966 | 1.4950 | 1.4968 | 1.4930 | 1.4986 | 1.5005 | **1.4906** |

**Table A-2** Unsuccessful Search with Chaining Hashing

| % | 8X4 | 8X8 | 16X8 | 16X16 | 32x16 | 32X32 | 64X32 | 64X64 | 128X64 | 128X128 | 256X128 | 256X256 | **Avg** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.1450 | 1.1150 | 1.0625 | 1.0900 | 1.0850 | 1.0950 | 1.1095 | 1.1038 | 1.1038 | 1.0970 | 1.1035 | 1.1069 | **1.1014** |
| 0.20 | 1.2400 | 1.2275 | 1.1638 | 1.1825 | 1.1856 | 1.1965 | 1.2075 | 1.1988 | 1.1944 | 1.1965 | 1.2080 | 1.2006 | **1.2001** |
| 0.30 | 1.3100 | 1.3425 | 1.2650 | 1.2638 | 1.2963 | 1.3065 | 1.2995 | 1.2969 | 1.2988 | 1.2940 | 1.3085 | 1.3106 | **1.2994** |
| 0.40 | 1.4300 | 1.4450 | 1.3738 | 1.3563 | 1.4075 | 1.4040 | 1.4035 | 1.4188 | 1.3856 | 1.4025 | 1.4065 | 1.4031 | **1.4030** |
| 0.50 | 1.5750 | 1.5400 | 1.4900 | 1.4500 | 1.5219 | 1.4965 | 1.5060 | 1.5231 | 1.4781 | 1.4985 | 1.4995 | 1.4919 | **1.5059** |
| 0.60 | 1.6650 | 1.6225 | 1.5775 | 1.5588 | 1.6288 | 1.5865 | 1.6080 | 1.6244 | 1.5906 | 1.6035 | 1.5990 | 1.5831 | **1.6040** |
| 0.70 | 1.7450 | 1.7025 | 1.6888 | 1.6600 | 1.7419 | 1.6835 | 1.7065 | 1.7138 | 1.6988 | 1.7140 | 1.7120 | 1.6688 | **1.7029** |
| 0.80 | 1.8700 | 1.7950 | 1.7725 | 1.7638 | 1.8469 | 1.7770 | 1.8000 | 1.8144 | 1.7975 | 1.8095 | 1.8100 | 1.7694 | **1.8022** |
| 0.90 | 1.9300 | 1.9025 | 1.8763 | 1.8450 | 1.9581 | 1.8820 | 1.8945 | 1.9131 | 1.8931 | 1.8980 | 1.9070 | 1.8669 | **1.8972** |
| 1.00 | 2.0650 | 2.0150 | 1.9938 | 1.9513 | 2.0669 | 1.9835 | 1.9965 | 2.0075 | 1.9913 | 1.9945 | 2.0175 | 1.9225 | **2.0004** |

**Table A-3** Successful Search with Linear Probing

| % | 8X4 | 8X8 | 16X8 | 16X16 | 32X16 | 32X32 | 64X32 | 64X64 | 128X64 | 128X128 | 256X128 | 256X256 | **Avg *** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.0833 | 1.0417 | 1.0417 | 1.0300 | 1.0147 | 1.0343 | 1.0515 | 1.0568 | 1.0540 | 1.0568 | 1.0541 | 1.0557 | **1.0519** |
| 0.20 | 1.2083 | 1.0625 | 1.0500 | 1.0882 | 1.0956 | 1.1054 | 1.1210 | 1.1319 | 1.1184 | 1.1268 | 1.1280 | 1.1242 | **1.1222** |
| 0.30 | 1.1944 | 1.1579 | 1.1645 | 1.1809 | 1.1814 | 1.2036 | 1.2044 | 1.2134 | 1.2144 | 1.2128 | 1.2138 | 1.2156 | **1.2111** |
| 0.40 | 1.2083 | 1.1800 | 1.2451 | 1.3137 | 1.2904 | 1.3209 | 1.3492 | 1.3243 | 1.3413 | 1.3356 | 1.3332 | 1.3307 | **1.3336** |
| 0.50 | 1.2656 | 1.4063 | 1.3594 | 1.4824 | 1.4551 | 1.4692 | 1.5176 | 1.5022 | 1.4926 | 1.5016 | 1.4980 | 1.5007 | **1.4974** |
| 0.60 | 1.5263 | 1.5724 | 1.5526 | 1.6503 | 1.6653 | 1.6877 | 1.7614 | 1.7876 | 1.7200 | 1.7606 | 1.7466 | 1.7425 | **1.7438** |
| 0.70 | 1.6818 | 1.8807 | 1.7612 | 2.0084 | 2.0335 | 2.2357 | 2.2027 | 2.2272 | 2.1324 | 2.1741 | 2.1673 | 2.1577 | **2.1853** |
| 0.80 | 2.1900 | 2.5098 | 2.3407 | 2.6544 | 2.5886 | 2.9737 | 3.0968 | 3.1190 | 2.9503 | 2.9873 | 2.9530 | 2.9701 | **3.0072** |
| 0.90 | 2.9196 | 2.7675 | 3.4761 | 4.0033 | 3.8000 | 5.0451 | 5.6084 | 5.5502 | 5.3947 | 5.3494 | 5.2408 | 5.3695 | **5.3654** |
| 1.00 | 4.1250 | 4.5195 | 6.9531 | 9.6934 | 12.6348 | 20.4734 | 31.2942 | 36.5738 | 55.1476 | 81.6992 | 114.5396 | 149.6571 | **69.9121** |

**Table A-4** Unsuccessful Search with Linear Probing

| % | 8X4 | 8X8 | 16X8 | 16X16 | 32X16 | 32X32 | 64X32 | 64X64 | 128X64 | 128X128 | 256X128 | 256X256 | **Avg *** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.1750 | 1.1350 | 1.0738 | 1.0988 | 1.0963 | 1.1070 | 1.1355 | 1.1175 | 1.1219 | 1.1115 | 1.1170 | 1.1231 | **1.1191** |
| 0.20 | 1.3300 | 1.2875 | 1.2113 | 1.2375 | 1.2494 | 1.2640 | 1.3050 | 1.2706 | 1.2588 | 1.2785 | 1.2925 | 1.2775 | **1.2781** |
| 0.30 | 1.4750 | 1.5250 | 1.4588 | 1.4025 | 1.4931 | 1.5105 | 1.5420 | 1.5231 | 1.5138 | 1.5090 | 1.5315 | 1.5219 | **1.5217** |
| 0.40 | 1.7700 | 1.7575 | 1.7500 | 1.6675 | 1.9175 | 1.8715 | 1.9425 | 1.9194 | 1.8844 | 1.9100 | 1.8900 | 1.9031 | **1.9030** |
| 0.50 | 2.3300 | 2.2775 | 2.1838 | 2.2513 | 2.4831 | 2.4565 | 2.5495 | 2.6156 | 2.4188 | 2.5110 | 2.4760 | 2.4250 | **2.4932** |
| 0.60 | 3.0450 | 3.5100 | 2.8725 | 3.2250 | 3.4238 | 3.3895 | 3.7130 | 4.0094 | 3.5569 | 3.7095 | 3.6060 | 3.4056 | **3.6271** |
| 0.70 | 4.2400 | 4.4550 | 3.7750 | 5.1188 | 5.2638 | 6.3420 | 6.7350 | 6.4519 | 5.8550 | 6.2560 | 5.9395 | 5.4844 | **6.1520** |
| 0.80 | 5.8100 | 7.4625 | 8.1613 | 9.2088 | 8.8744 | 12.0380 | 13.2525 | 13.5038 | 13.2050 | 13.0525 | 12.5905 | 11.5169 | **12.7370** |
| 0.90 | 9.2850 | 14.2825 | 15.2588 | 23.6863 | 25.7488 | 37.7470 | 42.3725 | 47.3750 | 44.6131 | 46.2005 | 43.6205 | 46.2781 | **44.0295** |
| 1.00 | 32.0000 | 64.0000 | 128.00 | 256.00 | 512.00 | 1024.00 | 2048.00 | 4096.00 | 8192.00 | 16384.0 | 32768.0 | 65536.0 | **18578.2** |

**Table A-5** Successful Search with Linear Probing improved by Model 1

| % | 8X4 | 8X8 | 16X8 | 16X16 | 32X16 | 32X32 | 64X32 | 64X64 | 128X64 | 128X128 | 256X128 | 256X256 | Avg * |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.0833 | 1.0417 | 1.0417 | 1.0300 | 1.0147 | 1.0343 | 1.0515 | 1.0568 | 1.0540 | 1.0568 | 1.0541 | 1.0557 | **1.0519** |
| 0.20 | 1.2083 | 1.0625 | 1.0500 | 1.0882 | 1.0956 | 1.1054 | 1.1210 | 1.1319 | 1.1184 | 1.1268 | 1.1280 | 1.1242 | **1.1222** |
| 0.30 | 1.1667 | 1.1579 | 1.1645 | 1.1809 | 1.1814 | 1.2036 | 1.2044 | 1.2134 | 1.2144 | 1.2128 | 1.2138 | 1.2155 | **1.2111** |
| 0.40 | 1.2083 | 1.1800 | 1.2451 | 1.3137 | 1.2904 | 1.3209 | 1.3492 | 1.3243 | 1.3413 | 1.3356 | 1.3330 | 1.3306 | **1.3336** |
| 0.50 | 1.2656 | 1.4063 | 1.3594 | 1.4824 | 1.4551 | 1.4692 | 1.5176 | 1.5022 | 1.4926 | 1.5010 | 1.4977 | 1.5002 | **1.4972** |
| 0.60 | 1.5263 | 1.5724 | 1.5526 | 1.6503 | 1.6596 | 1.6877 | 1.7596 | 1.7876 | 1.7195 | 1.7583 | 1.7455 | 1.7415 | **1.7428** |
| 0.70 | 1.6818 | 1.8807 | 1.7612 | 2.0084 | 2.0286 | 2.2207 | 2.1999 | 2.2265 | 2.1303 | 2.1709 | 2.1640 | 2.1543 | **2.1809** |
| 0.80 | 2.1900 | 2.5098 | 2.3407 | 2.6544 | 2.5862 | 2.9554 | 3.0951 | 3.1124 | 2.9463 | 2.9747 | 2.9449 | 2.9615 | **2.9986** |
| 0.90 | 2.9196 | 2.7675 | 3.4652 | 3.9989 | 3.7538 | 5.0353 | 5.5807 | 5.5244 | 5.3537 | 5.3135 | 5.2104 | 5.3317 | **5.3357** |
| 1.00 | 4.1250 | 4.5195 | 6.9434 | 9.6758 | 12.5527 | 20.3574 | 30.9094 | 35.8571 | 54.2146 | 80.0414 | 112.353 | 145.7826 | **68.5022** |

**Table A-6** Unsuccessful Search with Linear Probing improved by Model 1

| % | 8X4 | 8X8 | 16X8 | 16X16 | 32X16 | 32X32 | 64X32 | 64X64 | 128X64 | 128X128 | 256X128 | 256X256 | Avg * |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.1450 | 1.1225 | 1.0625 | 1.0900 | 1.0850 | 1.0955 | 1.1105 | 1.1038 | 1.1038 | 1.0970 | 1.1035 | 1.1094 | **1.1033** |
| 0.20 | 1.2550 | 1.2350 | 1.1638 | 1.1838 | 1.1913 | 1.1995 | 1.2125 | 1.2031 | 1.1969 | 1.1995 | 1.2160 | 1.2081 | **1.2051** |
| 0.30 | 1.3250 | 1.3625 | 1.2688 | 1.2763 | 1.3150 | 1.3260 | 1.3210 | 1.3169 | 1.3175 | 1.3065 | 1.3330 | 1.3281 | **1.3213** |
| 0.40 | 1.4650 | 1.4650 | 1.3875 | 1.3900 | 1.4538 | 1.4585 | 1.4725 | 1.4838 | 1.4388 | 1.4640 | 1.4455 | 1.4481 | **1.4587** |
| 0.50 | 1.6400 | 1.6175 | 1.5450 | 1.5150 | 1.6250 | 1.6040 | 1.6450 | 1.6663 | 1.5719 | 1.6315 | 1.6065 | 1.5906 | **1.6165** |
| 0.60 | 1.8250 | 1.7550 | 1.6950 | 1.6713 | 1.8156 | 1.7835 | 1.8500 | 1.9019 | 1.7513 | 1.8235 | 1.7875 | 1.8131 | **1.8158** |
| 0.70 | 2.0500 | 2.0575 | 1.9413 | 1.9613 | 2.1725 | 2.2130 | 2.2470 | 2.1906 | 2.1838 | 2.1845 | 2.1070 | 2.1525 | **2.1826** |
| 0.80 | 2.7200 | 2.2925 | 2.2850 | 2.3863 | 2.6731 | 2.8030 | 2.9345 | 3.0494 | 2.7406 | 3.0325 | 2.8840 | 2.6600 | **2.8720** |
| 0.90 | 3.1650 | 2.9925 | 3.1288 | 3.4075 | 4.0038 | 4.6560 | 5.1530 | 4.9844 | 4.7069 | 4.9635 | 4.4095 | 4.2619 | **4.7336** |
| 1.00 | 4.8850 | 4.6050 | 6.6413 | 7.5938 | 13.0744 | 19.5690 | 28.9310 | 37.8025 | 68.9169 | 98.591 | 111.286 | 50.5675 | **59.3806** |

**Table A-7** Successful Search with Linear Probing improved by Model 2

| % | *8X4* | *8X8* | *16X8* | *16X16* | *32X16* | *32X32* | *64X32* | *64X64* | *128X64* | *128X128* | *256X128* | *256X256* | **Avg *** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.0833 | 1.0417 | 1.0417 | 1.0300 | 1.0147 | 1.0319 | 1.0478 | 1.0556 | 1.0525 | 1.0554 | 1.0526 | 1.0534 | **1.0499** |
| 0.20 | 1.2083 | 1.0625 | 1.0500 | 1.0882 | 1.0907 | 1.0919 | 1.1161 | 1.1233 | 1.1117 | 1.1174 | 1.1191 | 1.1163 | **1.1137** |
| 0.30 | 1.1389 | 1.1447 | 1.1513 | 1.1612 | 1.1667 | 1.1857 | 1.1836 | 1.1938 | 1.1926 | 1.1903 | 1.1930 | 1.1933 | **1.1903** |
| 0.40 | 1.1875 | 1.1700 | 1.2353 | 1.2696 | 1.2537 | 1.2873 | 1.3114 | 1.2805 | 1.2937 | 1.2923 | 1.2884 | 1.2857 | **1.2913** |
| 0.50 | 1.2188 | 1.3984 | 1.3242 | 1.4102 | 1.3896 | 1.3838 | 1.4321 | 1.4128 | 1.4080 | 1.4203 | 1.4123 | 1.4154 | **1.4121** |
| 0.60 | 1.3947 | 1.5592 | 1.4671 | 1.4967 | 1.5432 | 1.5533 | 1.5882 | 1.6160 | 1.5684 | 1.6156 | 1.5953 | 1.5881 | **1.5893** |
| 0.70 | 1.4773 | 1.8239 | 1.6376 | 1.7291 | 1.7744 | 1.9581 | 1.8856 | 1.9203 | 1.8566 | 1.8861 | 1.8839 | 1.8726 | **1.8947** |
| 0.80 | 1.7000 | 2.2010 | 2.0196 | 2.0662 | 2.1210 | 2.4936 | 2.4988 | 2.4878 | 2.4044 | 2.4353 | 2.4137 | 2.4151 | **2.4498** |
| 0.90 | 2.1696 | 2.3509 | 2.6565 | 2.7283 | 2.8185 | 3.9254 | 4.1617 | 4.0471 | 4.0711 | 3.9045 | 3.9026 | 3.9665 | **3.9970** |
| 1.00 | 3.2969 | 3.5859 | 5.2578 | 6.7695 | 7.9141 | 13.6423 | 21.1898 | 22.8519 | 35.6000 | 52.1089 | 72.1613 | 93.6590 | **44.4590** |

**Table A-8** Unsuccessful Search with Linear Probing improve by Model 2

| % | *8X4* | *8X8* | *16X8* | *16X16* | *32X16* | *32X32* | *64X32* | *64X64* | *128X64* | *128X128* | *256X128* | *256X256* | **Avg *** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.1450 | 1.1225 | 1.0625 | 1.0900 | 1.0850 | 1.0950 | 1.1095 | 1.1038 | 1.1038 | 1.0970 | 1.1035 | 1.1081 | **1.1029** |
| 0.20 | 1.2550 | 1.2350 | 1.1638 | 1.1838 | 1.1894 | 1.1970 | 1.2125 | 1.2019 | 1.1963 | 1.1985 | 1.2110 | 1.2050 | **1.2032** |
| 0.30 | 1.3250 | 1.3550 | 1.2675 | 1.2688 | 1.3106 | 1.3230 | 1.3160 | 1.3125 | 1.3144 | 1.3000 | 1.3185 | 1.3219 | **1.3152** |
| 0.40 | 1.4450 | 1.4575 | 1.3863 | 1.3775 | 1.4406 | 1.4480 | 1.4625 | 1.4688 | 1.4275 | 1.4415 | 1.4255 | 1.4294 | **1.4433** |
| 0.50 | 1.6000 | 1.6100 | 1.5263 | 1.4850 | 1.5969 | 1.5685 | 1.6060 | 1.6175 | 1.5431 | 1.5770 | 1.5585 | 1.5556 | **1.5752** |
| 0.60 | 1.7050 | 1.7425 | 1.6625 | 1.6175 | 1.7763 | 1.7145 | 1.7775 | 1.8081 | 1.7081 | 1.7435 | 1.7130 | 1.7344 | **1.7427** |
| 0.70 | 1.8650 | 2.0100 | 1.8700 | 1.8338 | 2.0569 | 2.0350 | 2.1050 | 2.0406 | 2.0519 | 2.0135 | 1.9985 | 2.0119 | **2.0366** |
| 0.80 | 2.2250 | 2.1925 | 2.1363 | 2.1088 | 2.4106 | 2.5305 | 2.6015 | 2.6188 | 2.4738 | 2.5890 | 2.4745 | 2.3725 | **2.5229** |
| 0.90 | 2.4050 | 2.6625 | 2.6988 | 2.7375 | 3.1388 | 3.9065 | 3.9585 | 3.6788 | 3.6831 | 3.8335 | 3.5200 | 3.6975 | **3.7540** |
| 1.00 | 3.8450 | 3.7450 | 5.6063 | 5.1825 | 9.4144 | 15.0155 | 20.2630 | 20.3106 | 44.0050 | 54.6325 | 53.6920 | 23.4694 | **33.0554** |

**Table A-9** Successful Search with Double Hashing

| % | 8X4 | 8X8 | 16X8 | 16X16 | 32X16 | 32X32 | 64X32 | 64X64 | 128X64 | 128X128 | 256X128 | 256X256 | Avg * |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.0833 | 1.0417 | 1.0417 | 1.0300 | 1.0147 | 1.0441 | 1.0502 | 1.0526 | 1.0516 | 1.0551 | 1.0514 | 1.0525 | **1.0511** |
| 0.20 | 1.2500 | 1.0625 | 1.0500 | 1.0833 | 1.1029 | 1.1140 | 1.1064 | 1.1175 | 1.1145 | 1.1177 | 1.1150 | 1.1153 | **1.1143** |
| 0.30 | 1.2500 | 1.1316 | 1.1711 | 1.1645 | 1.1765 | 1.1930 | 1.1702 | 1.1832 | 1.1886 | 1.1864 | 1.1897 | 1.1898 | **1.1858** |
| 0.40 | 1.2292 | 1.1900 | 1.2647 | 1.2328 | 1.2782 | 1.2781 | 1.2637 | 1.2654 | 1.2756 | 1.2755 | 1.2750 | 1.2778 | **1.2730** |
| 0.50 | 1.2969 | 1.3516 | 1.3633 | 1.3633 | 1.3838 | 1.3730 | 1.3677 | 1.3740 | 1.3830 | 1.3900 | 1.3847 | 1.3884 | **1.3801** |
| 0.60 | 1.4605 | 1.5395 | 1.4704 | 1.4592 | 1.5269 | 1.4951 | 1.5100 | 1.5255 | 1.5162 | 1.5329 | 1.5316 | 1.5275 | **1.5198** |
| 0.70 | 1.5455 | 1.7102 | 1.6124 | 1.6620 | 1.7228 | 1.7629 | 1.6989 | 1.7240 | 1.7131 | 1.7206 | 1.7286 | 1.7216 | **1.7242** |
| 0.80 | 1.7100 | 2.2157 | 1.8971 | 2.0392 | 1.9994 | 2.0678 | 2.0023 | 2.0206 | 2.0152 | 2.0198 | 2.0226 | 2.0117 | **2.0229** |
| 0.90 | 2.1429 | 2.4737 | 2.4326 | 2.6054 | 2.6255 | 2.5874 | 2.6009 | 2.5929 | 2.6079 | 2.5882 | 2.5854 | 2.5643 | **2.5896** |
| 1.00 | 3.3125 | 4.0742 | 3.7656 | 5.2188 | 5.7705 | 6.8367 | 7.4849 | 7.7368 | 8.8311 | 9.6736 | 10.1471 | 11.2550 | **8.8522** |



**Table A-10** Unsuccessful Search with Double Hashing

| % | 8X4 | 8X8 | 16X8 | 16X16 | 32X16 | 32X32 | 64X32 | 64X64 | 128X64 | 128X128 | 256X128 | 256X256 | Avg * |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.1500 | 1.1275 | 1.0700 | 1.0975 | 1.0931 | 1.1085 | 1.1215 | 1.1188 | 1.1150 | 1.1120 | 1.1120 | 1.1194 | **1.1153** |
| 0.20 | 1.3100 | 1.2675 | 1.1950 | 1.2200 | 1.2306 | 1.2445 | 1.2735 | 1.2531 | 1.2500 | 1.2420 | 1.2580 | 1.2469 | **1.2526** |
| 0.30 | 1.4550 | 1.4350 | 1.3975 | 1.3800 | 1.4031 | 1.4470 | 1.4585 | 1.4563 | 1.4213 | 1.4240 | 1.4340 | 1.4281 | **1.4384** |
| 0.40 | 1.7050 | 1.6500 | 1.6488 | 1.5700 | 1.6763 | 1.6845 | 1.6945 | 1.7031 | 1.6500 | 1.6775 | 1.7030 | 1.6413 | **1.6791** |
| 0.50 | 2.1200 | 2.0025 | 2.0338 | 1.8825 | 2.0281 | 2.0055 | 2.0315 | 2.0294 | 1.9850 | 2.0425 | 2.0295 | 1.9863 | **2.0157** |
| 0.60 | 2.5000 | 2.5875 | 2.4325 | 2.4613 | 2.5138 | 2.5345 | 2.5190 | 2.5250 | 2.4900 | 2.5130 | 2.5310 | 2.4831 | **2.5137** |
| 0.70 | 2.9100 | 3.4350 | 3.3038 | 3.3100 | 3.3938 | 3.4985 | 3.3825 | 3.4306 | 3.3188 | 3.3145 | 3.3565 | 3.3506 | **3.3789** |
| 0.80 | 4.3200 | 5.7100 | 4.8725 | 5.2375 | 5.0706 | 5.3820 | 5.1230 | 5.2550 | 5.1156 | 5.0125 | 4.9400 | 5.0806 | **5.1298** |
| 0.90 | 7.3050 | 9.7000 | 10.0688 | 10.9350 | 10.5219 | 11.2235 | 10.6385 | 11.2725 | 10.1600 | 10.5200 | 9.8590 | 10.3569 | **10.5758** |
| 1.00 | 32.0000 | 64.0000 | 128.00 | 256.00 | 512.00 | 1024.00 | 2048.00 | 4096.00 | 8192.00 | 16384.0 | 32768.0 | 65536.0 | **18578.2** |

**Table A-11** Successful Search with Double Hashing improved by Model 1

| % | 8X4 | 8X8 | 16X8 | 16X16 | 32X16 | 32X32 | 64X32 | 64X64 | 128X64 | 128X128 | 256X128 | 256X256 | Avg * |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.0833 | 1.0417 | 1.0417 | 1.0300 | 1.0147 | 1.0417 | 1.0502 | 1.0526 | 1.0519 | 1.0554 | 1.0517 | 1.0526 | **1.0509** |
| 0.20 | 1.2500 | 1.0625 | 1.0500 | 1.0833 | 1.1103 | 1.1091 | 1.1057 | 1.1175 | 1.1136 | 1.1176 | 1.1152 | 1.1156 | **1.1135** |
| 0.30 | 1.1944 | 1.1316 | 1.1711 | 1.1645 | 1.1797 | 1.1954 | 1.1686 | 1.1816 | 1.1881 | 1.1879 | 1.1890 | 1.1900 | **1.1858** |
| 0.40 | 1.2083 | 1.1900 | 1.2647 | 1.2353 | 1.2733 | 1.2842 | 1.2601 | 1.2663 | 1.2746 | 1.2764 | 1.2737 | 1.2776 | **1.2733** |
| 0.50 | 1.3438 | 1.3438 | 1.3750 | 1.3711 | 1.3818 | 1.3770 | 1.3604 | 1.3800 | 1.3801 | 1.3915 | 1.3825 | 1.3869 | **1.3798** |
| 0.60 | 1.5000 | 1.5329 | 1.4803 | 1.4608 | 1.5391 | 1.4898 | 1.5210 | 1.5311 | 1.5170 | 1.5322 | 1.5258 | 1.5269 | **1.5205** |
| 0.70 | 1.5568 | 1.7216 | 1.6461 | 1.6858 | 1.7605 | 1.7916 | 1.7142 | 1.7200 | 1.7118 | 1.7254 | 1.7225 | 1.7219 | **1.7296** |
| 0.80 | 1.7500 | 2.1961 | 1.9632 | 2.0331 | 2.0733 | 2.1459 | 2.0079 | 2.0076 | 2.0150 | 2.0243 | 2.0164 | 2.0150 | **2.0332** |
| 0.90 | 2.1429 | 2.7412 | 2.4870 | 2.6283 | 2.6880 | 2.7315 | 2.5686 | 2.5937 | 2.6015 | 2.5951 | 2.5859 | 2.5651 | **2.6059** |
| 1.00 | 3.1406 | 3.9648 | 4.4922 | 5.6289 | 5.3799 | 6.7683 | 7.5759 | 8.7968 | 8.5650 | 9.7266 | 10.2444 | 10.9084 | **8.9408** |

**Table A-12** Unsuccessful Search with Double Hashing Improve by Model 1

| % | 8X4 | 8X8 | 16X8 | 16X16 | 32X16 | 32X32 | 64X32 | 64X64 | 128X64 | 128X128 | 256X128 | 256X256 | Avg * |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.1350 | 1.1075 | 1.0613 | 1.0875 | 1.0844 | 1.0935 | 1.1040 | 1.0994 | 1.1006 | 1.0930 | 1.0975 | 1.1038 | **1.0988** |
| 0.20 | 1.2150 | 1.2150 | 1.1588 | 1.1725 | 1.1744 | 1.1865 | 1.1970 | 1.1813 | 1.1831 | 1.1810 | 1.1855 | 1.1869 | **1.1859** |
| 0.30 | 1.2850 | 1.3200 | 1.2450 | 1.2463 | 1.2681 | 1.2780 | 1.2775 | 1.2694 | 1.2663 | 1.2690 | 1.2690 | 1.2769 | **1.2723** |
| 0.40 | 1.4050 | 1.4125 | 1.3263 | 1.3250 | 1.3713 | 1.3650 | 1.3715 | 1.3738 | 1.3469 | 1.3655 | 1.3595 | 1.3638 | **1.3637** |
| 0.50 | 1.5600 | 1.5125 | 1.4638 | 1.4213 | 1.4831 | 1.4610 | 1.4730 | 1.4788 | 1.4406 | 1.4675 | 1.4530 | 1.4500 | **1.4606** |
| 0.60 | 1.7050 | 1.6400 | 1.5575 | 1.5350 | 1.5869 | 1.5675 | 1.5935 | 1.5900 | 1.5625 | 1.5840 | 1.5655 | 1.5469 | **1.5728** |
| 0.70 | 1.8350 | 1.7675 | 1.7138 | 1.6800 | 1.7550 | 1.7135 | 1.7645 | 1.7425 | 1.7069 | 1.7465 | 1.7185 | 1.6625 | **1.7221** |
| 0.80 | 2.1000 | 2.0725 | 1.9538 | 1.9100 | 1.9894 | 1.9470 | 1.9580 | 1.9513 | 1.9156 | 1.9635 | 1.9195 | 1.8888 | **1.9348** |
| 0.90 | 2.3000 | 2.8075 | 2.4613 | 2.3225 | 2.5131 | 2.3970 | 2.4275 | 2.3544 | 2.3294 | 2.3830 | 2.3740 | 2.2438 | **2.3584** |
| 1.00 | 4.2150 | 4.3350 | 5.1838 | 4.7963 | 5.9500 | 6.0870 | 8.6835 | 8.0163 | 8.1563 | 10.7505 | 8.5455 | 8.4338 | **8.3818** |

**Table A-13** Successful Search with Double Hashing improved by Model 2

| % | 8X4 | 8X8 | 16X8 | 16X16 | 32X16 | 32X32 | 64X32 | 64X64 | 128X64 | 128X128 | 256X128 | 256X256 | **Avg *** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.0833 | 1.0417 | 1.0417 | 1.0300 | 1.0147 | 1.0368 | 1.0502 | 1.0520 | 1.0507 | 1.0531 | 1.0503 | 1.0511 | **1.0492** |
| 0.20 | 1.2083 | 1.0625 | 1.0500 | 1.0735 | 1.0882 | 1.0968 | 1.1002 | 1.1111 | 1.1061 | 1.1094 | 1.1083 | 1.1081 | **1.1057** |
| 0.30 | 1.1389 | 1.1184 | 1.1711 | 1.1480 | 1.1520 | 1.1718 | 1.1559 | 1.1661 | 1.1708 | 1.1685 | 1.1704 | 1.1724 | **1.1680** |
| 0.40 | 1.1875 | 1.1500 | 1.2500 | 1.2157 | 1.2316 | 1.2384 | 1.2274 | 1.2357 | 1.2377 | 1.2411 | 1.2374 | 1.2410 | **1.2370** |
| 0.50 | 1.2188 | 1.2969 | 1.3320 | 1.3164 | 1.3242 | 1.3086 | 1.2986 | 1.3201 | 1.3171 | 1.3255 | 1.3202 | 1.3242 | **1.3163** |
| 0.60 | 1.3421 | 1.4934 | 1.4309 | 1.3971 | 1.4479 | 1.3958 | 1.4041 | 1.4278 | 1.4122 | 1.4304 | 1.4263 | 1.4222 | **1.4170** |
| 0.70 | 1.3750 | 1.6193 | 1.5478 | 1.5349 | 1.5691 | 1.5988 | 1.5464 | 1.5649 | 1.5482 | 1.5605 | 1.5582 | 1.5550 | **1.5617** |
| 0.80 | 1.4700 | 1.8725 | 1.7132 | 1.7586 | 1.7225 | 1.8159 | 1.7442 | 1.7592 | 1.7457 | 1.7558 | 1.7495 | 1.7466 | **1.7596** |
| 0.90 | 1.8304 | 2.1491 | 2.1130 | 2.0783 | 2.1011 | 2.1401 | 2.1145 | 2.1245 | 2.1246 | 2.1119 | 2.1090 | 2.0945 | **2.1170** |
| 1.00 | 2.6563 | 2.8945 | 3.1230 | 4.0557 | 3.6973 | 4.8811 | 5.3027 | 4.9329 | 6.3806 | 6.4426 | 6.8548 | 7.8496 | **6.0920** |

**Table A-14** Unsuccessful Search with Double Hashing improved by Model 2

| % | 8*4 | 8*8 | 16*8 | 16*16 | 32*16 | 32*32 | 64*32 | 64*64 | 128*64 | 128*128 | 256*128 | 256*256 | **Avg *** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | 1.1450 | 1.1225 | 1.0625 | 1.0900 | 1.0850 | 1.0965 | 1.1100 | 1.1038 | 1.1044 | 1.0975 | 1.1040 | 1.1075 | **1.1034** |
| 0.20 | 1.2550 | 1.2350 | 1.1638 | 1.1800 | 1.1894 | 1.1980 | 1.2075 | 1.2000 | 1.1963 | 1.1985 | 1.2085 | 1.2013 | **1.2014** |
| 0.30 | 1.3250 | 1.3500 | 1.2688 | 1.2650 | 1.3069 | 1.3125 | 1.3020 | 1.3038 | 1.3006 | 1.2995 | 1.3150 | 1.3163 | **1.3071** |
| 0.40 | 1.4450 | 1.4525 | 1.3875 | 1.3538 | 1.4344 | 1.4195 | 1.4120 | 1.4413 | 1.3994 | 1.4155 | 1.4210 | 1.4219 | **1.4186** |
| 0.50 | 1.6000 | 1.5800 | 1.5350 | 1.4638 | 1.5575 | 1.5230 | 1.5305 | 1.5750 | 1.5069 | 1.5420 | 1.5310 | 1.5231 | **1.5331** |
| 0.60 | 1.7450 | 1.7475 | 1.6600 | 1.5900 | 1.6988 | 1.6525 | 1.6765 | 1.7213 | 1.6644 | 1.6885 | 1.6580 | 1.6363 | **1.6711** |
| 0.70 | 1.8650 | 1.8850 | 1.8388 | 1.7463 | 1.8963 | 1.8020 | 1.8435 | 1.8588 | 1.8431 | 1.8830 | 1.8445 | 1.7838 | **1.8369** |
| 0.80 | 2.0750 | 2.1500 | 2.0063 | 2.0063 | 2.0994 | 2.0170 | 2.0490 | 2.0713 | 2.0475 | 2.1220 | 2.0780 | 2.0013 | **2.0551** |
| 0.90 | 2.2550 | 2.6925 | 2.4775 | 2.4988 | 2.5131 | 2.3760 | 2.4505 | 2.4125 | 2.3881 | 2.4495 | 2.4810 | 2.3681 | **2.4180** |
| 1.00 | 3.5200 | 3.5700 | 3.9425 | 3.4988 | 4.8656 | 4.2140 | 5.4525 | 4.5119 | 5.2631 | 4.4505 | 5.1930 | 5.2025 | **4.8982** |

**Table A-15** Total average successful search times of one key with load factor under 0.9

| Methods | *8X4* | *8X8* | *16X8* | *16X16* | *32X16* | *32X32* | *64X32* | *64X64* | *128X64* | *128X128* | *256X128* | *256X256* | **Avg** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ch | 1.2382 | 1.2463 | 1.2526 | 1.2570 | 1.2670 | 1.2743 | 1.2704 | 1.2741 | 1.2722 | 1.2727 | 1.2736 | 1.2747 | **1.2644** |
| L | 1.5864 | 1.6199 | 1.6657 | 1.8235 | 1.7916 | 2.0084 | 2.1014 | 2.1014 | 2.0465 | 2.0561 | 2.0372 | 2.0519 | **1.9075** |
| D | 1.4409 | 1.5241 | 1.4781 | 1.5155 | 1.5367 | 1.5462 | 1.5300 | 1.5395 | 1.5406 | 1.5429 | 1.5427 | 1.5388 | **1.5230** |
| L-M1 | 1.5833 | 1.6199 | 1.6645 | 1.8230 | 1.7850 | 2.0036 | 2.0977 | 2.0977 | 2.0412 | 2.0500 | 2.0324 | 2.0461 | **1.9037** |
| D-M1 | 1.4477 | 1.5513 | 1.4977 | 1.5214 | 1.5579 | 1.5740 | 1.5285 | 1.5389 | 1.5393 | 1.5451 | 1.5403 | 1.5391 | **1.5318** |
| L-M2 | 1.3976 | 1.5280 | 1.5093 | 1.5533 | 1.5747 | 1.7679 | 1.8028 | 1.7930 | 1.7732 | 1.7686 | 1.7623 | 1.7674 | **1.6665** |
| D-M2 | 1.3171 | 1.4226 | 1.4055 | 1.3947 | 1.4057 | 1.4226 | 1.4046 | 1.4179 | 1.4126 | 1.4174 | 1.4144 | 1.4128 | **1.4040** |
| **Total Avg** | **1.4302** | **1.5017** | **1.4962** | **1.5555** | **1.5598** | **1.6567** | **1.6765** | **1.6804** | **1.6608** | **1.6647** | **1.6576** | **1.6615** | **1.6001** |

**Table A-16** Total Average unsuccessful search times of one key with load factor under 0.9

| Methods | **8X4** | *8X8* | *16X8* | *16X16* | *32X16* | *32X32* | *64X32* | *64X64* | *128X64* | *128X128* | *256X128* | *256X256* | **Avg** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ch | 1.5456 | 1.5214 | 1.4744 | 1.4633 | 1.5191 | 1.4919 | 1.5039 | 1.5119 | 1.4934 | 1.5015 | 1.5060 | 1.4890 | **1.5018** |
| L | 3.3844 | 4.1881 | 4.1939 | 5.4329 | 5.7278 | 7.5251 | 8.2831 | 8.8651 | 8.3808 | 8.6154 | 8.2293 | 8.3262 | **6.7627** |
| D | 2.7528 | 3.2128 | 3.1136 | 3.2326 | 3.2146 | 3.3476 | 3.2492 | 3.3382 | 3.1673 | 3.2064 | 3.1359 | 3.1881 | **3.1799** |
| L-M1 | 1.8433 | 1.7667 | 1.7197 | 1.7646 | 1.9261 | 2.0154 | 2.1051 | 2.1000 | 2.0013 | 2.0781 | 1.9881 | 1.9524 | **1.9384** |
| D-M1 | 1.6156 | 1.6506 | 1.5490 | 1.5222 | 1.5806 | 1.5566 | 1.5741 | 1.5601 | 1.5391 | 1.5614 | 1.5491 | 1.5248 | **1.5653** |
| L-M2 | 1.6633 | 1.7097 | 1.6415 | 1.6336 | 1.7783 | 1.8687 | 1.9054 | 1.8723 | 1.8335 | 1.8659 | 1.8137 | 1.8263 | **1.7844** |
| D-M2 | 1.6344 | 1.6906 | 1.6000 | 1.5771 | 1.6423 | 1.5997 | 1.6202 | 1.6319 | 1.6056 | 1.6329 | 1.6268 | 1.5955 | **1.6214** |
| **Total Avg** | **2.0628** | **2.2485** | **2.1846** | **2.3752** | **2.4841** | **2.7721** | **2.8916** | **2.9828** | **2.8601** | **2.9231** | **2.8355** | **2.8432** | **2.6220** |

Note:

- * The average value is just from the tables with size at least 32X32

VITA

Chuanjiang Lu

Candidate for the Degree of

Master of Science

Thesis: MULTI-DIMENSIONAL HASH TABLE AND APPLICATION IN GRIDDING

Major Field: Computer Science

Biographical:

Personal Data: Born in Dehui, Jilin, P. R. China, July 8, 1962, the son of Mr. Zhenglin Lu and Mrs. Huilan Zhang.

Education: Graduated from the No. 7 High School of Dehui, Jilin, P. R. China, in July 1979; received the Bachelor of Science from Changchun University of Earth Science, Changchun, China, in July 1983; received Master degree in Petroleum-Geology from Changchun University of Earth Science, Changchun, China, in July 1993. Completed the requirements for Master of Science at Oklahoma State University in December 2000.

Professional Experience: Employed by Video Education Center, Changchun, Jilin, Chain, as an Editor, August 1983 to July 1987; employed by Changchun Resource Institute, Changchun, Jilin, China, as a Research Scientist, August 1990 to May 1995.