AN EXPERIMENTAL ANALYSIS OF A NEW MULTIDIMENSIONAL STORAGE AND RETRIEVAL METHOD

By

YUNPENG ZHANG

Bachelor of Science
Shanghai JiaoTong University
Shanghai, China
1987

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1994

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, 1996

AN EXPERIMENTAL ANALYSIS OF A NEW MULTIDIMENSIONAL STORAGE AND RETRIEVAL METHOD

Thesis Approved:

D. E. Hea

Thesis Adviser

H. Lu

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my graduate adviser Dr. G. E. Hedrick. His support and guidance have been of great benefit in the completion of this research. I also would like to thank Dr. H. Lu and Dr. K. M. George for serving on my graduate committee.

My fellow research assistants deserve a word of thanks for their friendship and for their part in making the Computer Science Department at Oklahoma State University a pleasant working place. I also wish to extend my appreciation to Dr. R. A. DiVall for his very useful comments and suggestions.

Finally, thanks to my dearest wife, Haiyan Liu. Her love, patience and understanding made this possible.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter I

Literature Review

Indexing methods always play a very important role in a database system. Many methods such as hashing are limited to a one dimensional data space. Some of today's database systems have become more complicated than the early systems in requiring efficient retrieval in a multidimensional data space.

In 1984, Guttman[8] introduced the R-tree, a dynamic indexing method for searching data spaces. In his paper, Guttman[8] presented an efficient index mechanism which allows rapid data retrieval according to the spatial locations of the data. Lomet and Salzberg[11] in 1990 provided another dynamic multi-attribute indexing method, the hB-Tree. Inter-node searching and growth handling characteristics of hB-trees are precisely analogous to the corresponding processes in B-trees. Both of them employ pointers to navigate within their index trees.

In contrast to the dynamic implementations using pointers, Coburn[2] (approximately 1989) presented a static multidimensional indexing method which he named SPIN (Single Point Index Network). In his paper, Coburn introduced three basic SPIN functions: C_SPIN,

S_SPIN and R_SPIN. The purpose of all three functions is to convert multidimensional indexes into a single index that can be used to locate data items in consecutive storage blocks easily. Among the three functions (C_SPIN, S_SPIN and R_SPIN), the time complexity analyses of the C_SPIN and S_SPIN search and insert operations are relatively trivial: both should have constant running time, since these two functions use no index trees. Access to the nodes within the same level requires the same number of calculations. However, because C_SPIN and S_SPIN simply convert multidimensional subscripts into a one dimensional index, both functions sacrifice storage whenever multidimensional subscripts are not consecutive. This disadvantage makes both C_SPIN and S_SPIN unattractive. Compared with C_SPIN and S_SPIN, R_SPIN can handle sparse multidimensional data, making R_SPIN more practical. In 1995, Zhang, et al.[18] performed experimental and pattern analyses to the index of R_SPIN, and indicated: "Both successful and unsuccessful insertion and search operations in the R_SPIN function have an average running time of either $O(1)$ when the index of the R_SPIN function is implemented on disk, or is $O(\log(n))$ if the index is implemented in main storage [Zhang, et al., 1995]."

Even though the ability to handle sparse data representations in R_SPIN suggests a new potential multidimensional indexing method, the very restricted index structure and the overflow problem of R_SPIN

still limit its use in actual storage and retrieval applications. In this paper, we combine the SPIN technique and the hashing method to introduce a new multidimensional storage and retrieval method, H_SPIN, which has very flexible growth handling and good time performance. Unlike the S_SPIN algorithm which involves several very large size hashing tables (one table on each level), H_SPIN uses a small size hashing table on each level, and it also can handle sparse data representations. In order to compare H_SPIN with R_SPIN, we also perform the analysis of the time complexities of H_SPIN and R_SPIN using the experimental method.

Chapter II

Sparse Data Representation

Since sparse data handling is one of the most important purposes in the design of the original R_SPIN function algorithm, we must understand sparse data representation before we get into the details of R_SPIN and H_SPIN. According to Coburn, the sparse data representation is defined as "... a set of indices at one level of a multidimensional array are, in reality, mapped to only a very few indices of the next level, the array is said to be sparse at that level [Coburn, 1989]."

An example demonstrates this. Suppose there is a small airline network, which runs among a total of fifteen cities. Also suppose that each airline route contains four cities in addition to the originating city, and each city in this network is maximally connected to five other cities. If we index of the fifteen cities and name them according to their indexes from 0 to 14, we easily can identify an airline route by using the combination of the city's index numbers. For instance, the airline route from a given city through the four cities numbered 4-12-5-11 is represented by the index combination of [4][12][5][11].

Now, let us consider the implementations of this airline network by using different indexing methods. When using a conventional multidimensional array, the array must be defined as "int L[15][15][15][15]," and the total number of storage locations for this array is 15 X 15 X 15 X 15 = 50625. However, since each city is connected to no more than five other cities (The maximum size in each dimension is five), the maximum number of storage locations which can be used from the 50625 allocated is 5 X 5 X 5 X 5 = 625. Consequently, 50000 (50625 - 625) storage locations are unused. Clearly, the conventional multidimensional array does not handle sparse data representations very well. An alternate option is either the R_SPIN method, or the H_SPIN method, which we will discuss in the later chapters. With either of these two methods, all the possible airline routes, such as 4-12-5-11, can be mapped into a storage block which maximally has the size of 5 X 5 X 5 X 5 = 625, when each city is connected to at most five other cities.

Chapter III

Introduction to R_SPIN

The application of an R_SPIN function includes two parts: first, a multidimensional subscript combination is mapped to a set of one dimensional index values by the R_SPIN function; second, a data record (or data item (datum)) is inserted or found in a storage block according to one of these one dimensional indexes. In fact, the second part is extremely easy to implement given the one dimensional indices.

Index conversion formula

Coburn introduced a formula which converts multidimensional indices into several one dimensional indices on each level. It not only calculates the index values in one dimensional storage block, but also computes an identification value for each node in the R_SPIN function's index file. This formula plays a very important role in the R_SPIN algorithm. It is:

$$rec\_number[i+1] = rec\_number[i] \times 10^{ex[i]} + k[i+1] - rec\_number[i] \times kr[i] \quad \ldots \ (1)$$

where,

rec_number[i+1] = the index for the i+1st level.

rec_number[i] = the index for the previous level or iteration.

max[i] = the number of indices in the ith dimension.

ex[i]s are exponents computed as follows:

ex[i] = 1, for $0 < max[i+1] \leq 10$.

ex[i] = 2, for $10 < max[i+1] \leq 100$.

ex[i] = 3, for $100 < max[i+1] \leq 1000$.

etc.

k[i] = the value of the multidimensional subscript within the ith dimension.

kr[i]s are values computed as follows:

$kr[i] = 10 - max[i+1]$, for $0 < max[i+1] \leq 10$

$kr[i] = 100 - max[i+1]$, for $10 < max[i+1] \leq 100$

$kr[i] = 1000 - max[i+1]$, for $100 < max[i+1] \leq 1000$$

etc.

In this paper a level is defined as "one less than the number of the dimensions within a multidimensional array[Coburn, 1989]." For example, in the array "L[4][12][5][11]," level 0 refers to the first dimension "L[4]," level 1 to the first two dimensions "L[4][12]," and so on.

An example of the use of formula (1) is in figure III-1. This figure shows the layout of a multidimensional array with the definition of "L[2][2][2][2]." In fact, figure III-1 is a forest containing two trees. The four nodes marked with "*" in each level represent an instance with

multidimensional subscripts of [1][0][1][0]. According to the subscript information of this instance, formula (1) can compute the instance's indexes on each level, which are rec_number[0] = 1 on the level 0, rec_number[1] = 2 on the level 1, rec_number[2] = 5 on the level 2 and rec_number[3] = 10 on the level 3.

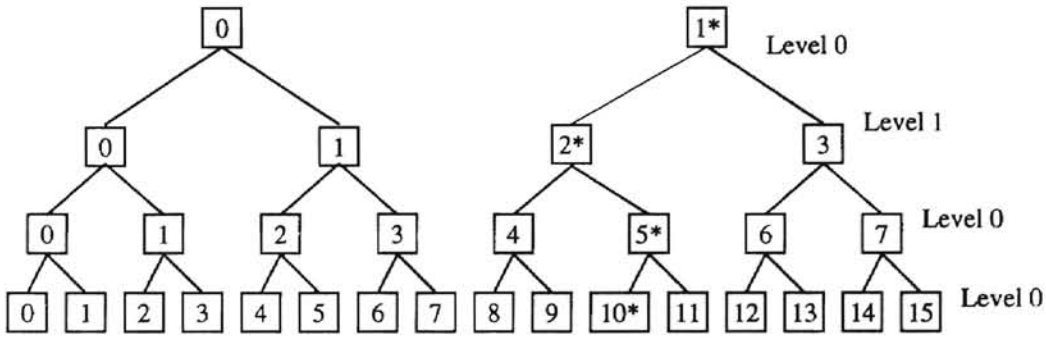

Figure III-1. The layout of a multidimensional array

Index in the R_SPIN function

The example in figure III-1, shows that the indexes converted by formula (1) are consecutive. Formula (1) cannot handle sparse data representations. In fact, all the sparse multidimensional subscript combinations (permutations) are stored as an index of the R_SPIN function indirectly. The layout of this index is similar to what is shown in figure III-1, except the size in each dimension is not limited to 2. The location of each node within this index can be computed according to the formula:

$$\left(\sum_{n=1}^{i-1} J_n \times K_n + j \times K_i + k\right) \times sizeof(V) \quad \text{..... (2)}$$

where:

i = level number (i.e. ith level).

j = the index value on level i-1.

k = the storage location in the jth entry of level i-1.

K_n = dimensional size on the level n.

J_n = the total entries on the level n-1.

V = the value stored in each node.

For example, let us consider the node 10 in level 3 in figure III-1. In this case,

$$i = 3, j = 5, k = 0$$

$$K_3 = 2$$

$$\sum_{n=1}^{2} J_n \times K_n = 12$$

Therefore, the location of this node in the index is

$$12 + 5 \times 2 + 0 = 22,$$

if sizeof(V) is assumed to equal to 1. Neither node 0 nor node 1 on the level 0 in figure III-1 is stored in the index, rather they are used as the indices of trees in the forest.

The R_SPIN algorithm

Step RF 1 [Initialization]

Compute the parameters for the formula (1).

Set level $i = 0$.

Create an integer array "*krec*" to contain the index values returned.

Locate the tree to be retrieved by using the subscript on level 0.

Step RF 2 [Go down one level]

Within this tree (or subtree, if not on level 0) with its root named "*T*," go to the next level by incrementing i by *1*.

Set $k = 0$ for retrieval through the immediate children of the root "*T*."

In case i is greater than the maximum level number, terminate R_SPIN function and return the integer array "*krec*."

Step RF 3 [Compute the node identifier]

On level i, Use formula (1) to convert the sparse multidimensional subscripts into a one dimensional index value, "*V*." This index is used as the unique identifier for this sparse subscript combination, but it is not the real index on this level.

Step RF 4 [Checking nodes]

On level i, use formula (2) to locate the kth child node of root "*T*," and read the node value into a temporary buffer "*temp*."

If "*temp*" is 0, then go to Step RF 5.

If "*temp*" equals the value of "*V*," then go to Step RF 6.

If the value of *k* is greater than the dimensional size on level *i* (which is 2 for all levels in figure III-1), then go to Step RF 7.

Otherwise, go to Step RF 8.

Step RF 5 [Empty node]

A value of 0 indicates that this node is empty. In this case, deposit the value "*V*" computed in Step RF 3 , and compute the index value to be returned, "*krec[i]*," from this level by using formula (1) and by using the node's real location in the tree. Then choose this node as root "*T*" and go to Step RF 2.

Step RF 6 [Matching]

This sparse subscript combination has already been recorded in the index. In this case, just calculate the returned index value "*krec[i]*" for this level by using the formula (1) and by using the node's real location in the tree. Then choose this node as root "*T*" and go to Step RF 2.

Step RF 7 [Overflow]

If k value is greater than the dimensional size in level *i* (the dimensional size is 2 for all levels in figure III-1), then assign *-1* to "*krec[i]*," and terminate the R_SPIN function, and return the integer array "*krec*."

Step RF 8 [Go to next sibling]

Increment k by 1 and go to Step RF 4.

Step RF 4 and Step RF 8 of the above algorithm, show when the children of a root "T," are retrieved, the retrieval always starts from the leftmost child, i, of "T" and proceeds to the right. Therefore, the layout of the index of the R_SPIN function can be converted from what is in figure III-1 to what is shown in figure III-2.

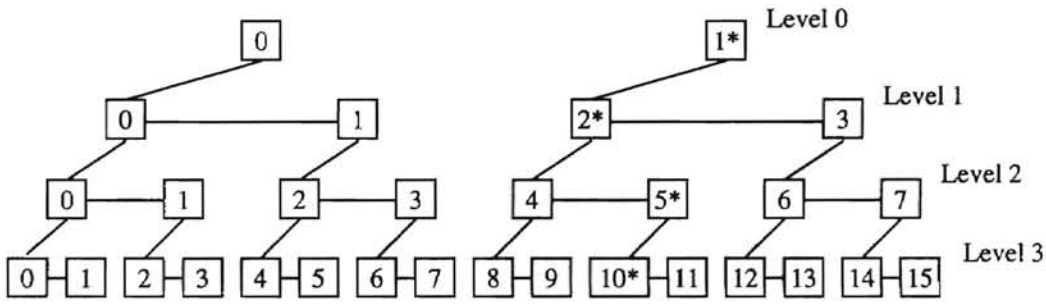

Figure III-2. Layout of the index of R_SPIN

From further pattern analysis (figure III-2), the paths from the root to the nodes which are at the same level will go through the same number of vertical edges. These vertical edges will contribute to the constant part of the average running time of R_SPIN. After we remove these vertical edges from figure III-2, we create a simplified pattern for the index of R_SPIN. Figure III-2 shows the simplified pattern converted from the first tree in figure III-2. Since all the vertical edges and the upper level nodes have been removed, the nodes in figure III-3 are the nodes of the final level in figure III-2. The edges in the pattern

of figure III-3 will contribute to the variable part of the average running time of R_SPIN.

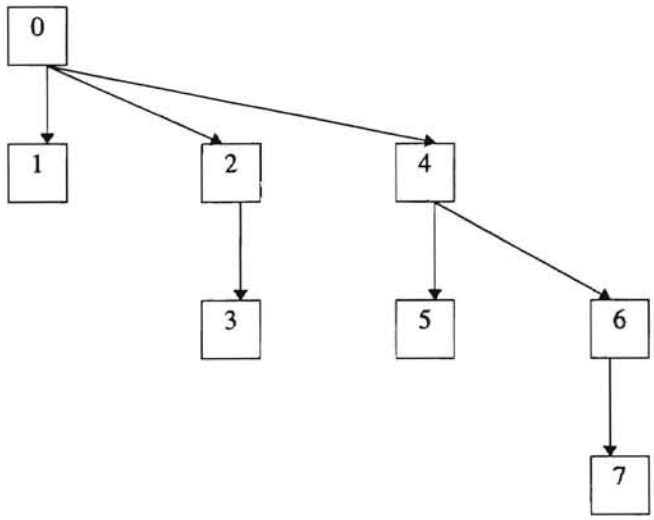

Figure III-3 The simplified layout of the index of R_SPIN

Disadvantages of the R_SPIN algorithm

Since R_SPIN stores everything in a single space and it is a very restricted structure, it has some disadvantages in space management and growth handling. These disadvantages are:

1. Some space is wasted due to the restricted dimensional size on each level. For examples, there are two high school classes, which have twenty and sixty students respectively. We can use a two-level R_SPIN structure to implement this model: the first level represents the class number and the second level represents the students. In this case, the dimensional size of the second level should be sixty.

However, to the class which just has twenty students, forty storage locations will be wasted.

2. Overflow handling causes extremely high cost in storage space. In R_SPIN, every sub-tree on a certain level is identical to the others and this restriction will bring troubles when we handle the overflow situation. For example, in figure III-4, if an overflow occurs among the children of the node 0 on the level 1, we need to increase the number of the sub-trees of the node 0 from 2 to 3. In addition, we must do the same thing to the other nodes on the level 1, no matter whether the additional increases are necessary or not.

Overflow handling is very time consuming. Since in the R_SPIN algorithm, the connections between the parent and its children are the locations, and all the elements of the R_SPIN index are aligned in a sequential order in a single space, the moving of an element will not only cause its children to move, but also cause the movement of all the elements after it. This is extremely time consuming.

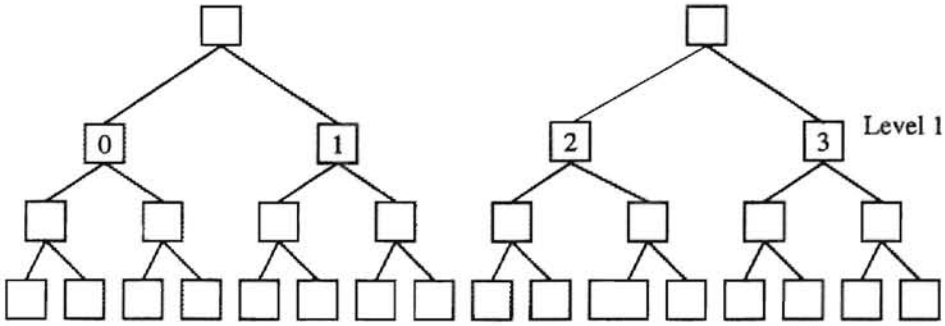

Figure III-4. The layout of the index of the R_SPIN function in tree representation

Chapter IV

Introduction to H_SPIN

H_SPIN

If we use a hashing table on each level, instead of using the very restricted array structure of R_SPIN, we can avoid R_SPIN's disadvantages. Also, the time complexities of insertion and search operations can be improved from $O(\log(n))$ to $O(1)$. With this new method, we still use some of the SPIN knowledge, and we can expect that H_SPIN can do whatever R_SPIN does. We name this new method H_SPIN, which means the combination of SPIN and hashing techniques.

Figure IV-1 shows the rough idea of the H_SPIN. In figure IV.1, each level is represented by a hashing table which can be an open hashing, or a closed hashing, or etc. The H_SPIN algorithm can be easily explained by using an example.

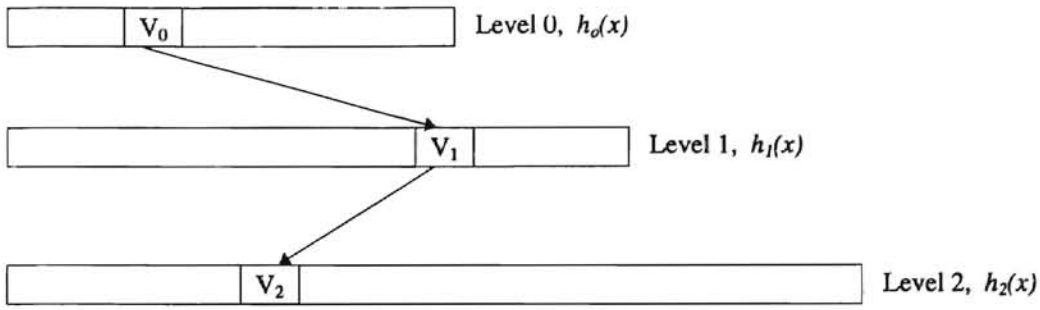

Figure IV-1. The layout of the H_SPIN index

Suppose the sparse representation in our example is $N[a][b][c]$. In figure IV-1, the value of V_0 is equal to the index value a, and the values of V_1 and V_2 represent the unique identifiers computed using the formula (1). The functions of $h_0(x)$, $h_1(x)$ and $h_2(x)$ in figure IV-1 are the hashing functions on each level.

From formula (1), the unique identifier on a certain level can be computed using the sparse index on this level and the identifier of the immediate upper level. In our example, $V_0 = a$, $V_1 = f(V_0, b)$ and $V_2 = f(V_1, c)$. Once we know the unique identifier on a level, by using the hashing function which is associated to this level, we can hash this identifier to a position in the hashing table on this level. In addition, since the hashed positions of a sparse representation are known on all levels after these hashing operations discussed above, the H_SPIN function can also return the actual positions (or the indexes) for all levels as we did using the R_SPIN function.

Growth handling of the H_SPIN index

Since the hashing tables of each level are independent tables, they can be expanded or compacted without affecting the other tables. Consequently, the number of the elements on each level of the H_SPIN index can be adjusted flexibly.

In the H_SPIN algorithm, the connection between the two immediate levels is associated only with the unique identifier and the sparse index value, instead of being associated with the locations of the parent and the child on two levels. For example, in figure IV-1, when we expand the hashing table of the level 1, we need to rehash all the elements in this table. Let us look at the element V_1 on level 1 (i.e. on the second dimension) as an example. V_1 may be rehashed to another location when the table is expanded. However, after the expansion, we can still use the value of V_1 and the sparse index value of the third dimension to uniquely locate the node V_2 on level 2.

Partial matching using H_SPIN

The method above works well for exact-match searching, but it cannot implement partial-match searching since the parent itself does

not store its children's identifiers. In order to deal with partial-match searching, we can attach a linked list to each parent. Let us see an example which has three permutations, $N[a][b][c]$, $N[a][b][c_1]$ and $N[a][b][c_2]$. As shown in figure IV-2, the element V_1 on the level 1 has three children on the level 2, which are V_2^C , V_2^{C1} and V_2^{C2}. The sparse index values of the level 2 can be put into the linked list as shown in figure IV-2.

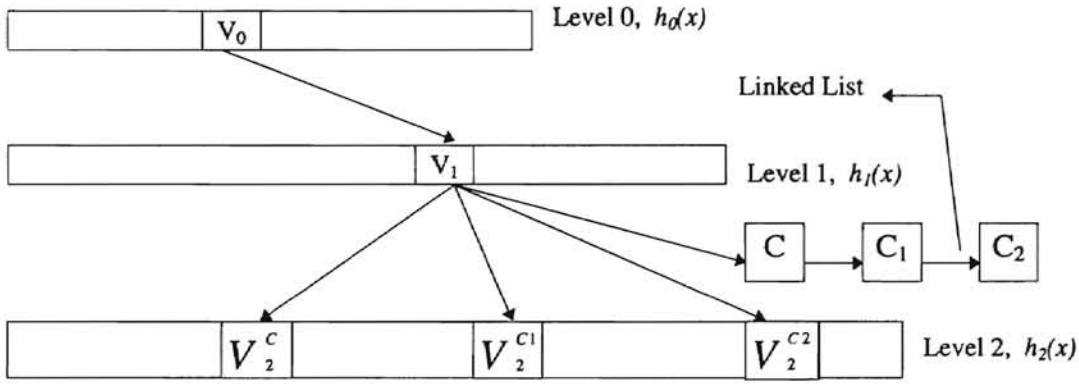

Figure IV-2. The improved layout of the H_SPIN for partial match

Suppose that we have a partial-match search query such as $N[a][b][?]$. After we reach level 1 and find the position using the identifier of V_1, we can follow the linked list attached to this node and compute the identifiers for the next level. In this way, the permutations of $N[a][b][c]$, $N[a][b][c_1]$ and $N[a][b][c_2]$ can be retrieved easily.

The H_SPIN algorithms

In this section, we will introduce the H_SPIN package, including some basic functions such as insertion, search and deletion, and also some advanced functions such as growth handling (rehashing) and partial-match searching. All the programs (or functions) introduced in this section are listed in Appendix A.

Closed hashing method is employed in this paper. Since a node in a closed hashing table can be in empty, legitimate or deleted state, in main function, a numeration type "enum kind_of_entry {legitimate, empty, deleted}" is defined to indicate the three states for each node.

The *h_spin(...)* function

The *h_spin(...)* function is capable of doing three basic operations: insertion, exact-match search and deletion. A parameter, "*mode*", passed to this function can control which of the three operations the *h_spin(...)* function performs. The insertion mode, the search mode and the deletion mode are respectively represented by three single characters: "*p*", "*s*" and "*d*."

Step HF 1 [Initialization]

Compute the parameters for the formula (1).

Set level $i = 0$.

Create an integer array "*krec*" to contain the index values returned.

Step HF 2 [Compute the node identifier]

On level i, use formula (1) to convert the sparse multidimensional subscripts into a one dimensional index value, V. This index value is the unique identifier for this sparse subscript combination.

Step HF 3 [Perform operations]

- If the *h_spin(...)* function is in the insertion mode, call the function *insert(...)* and pass i and the identifier V to it.

  Then assign the return value of the *insert(...)* function to *krec[i]*.
- if the *h_spin(...)* function is in the search mode, or in the deletion mode, call the function *search(...)*, and pass i and the identifier V to it. Then check if the status of the node found by the *search(...)* function is legitimate. If legitimate, assign the return value of the *search(...)* function to *krec[i]*, then go to next step. If not legitimate, assign *-1* to *krec[0]* and terminate the *h_spin(...)* function by returning *krec*.

Step HF 4 [Go down one Level]

Go down to the next level by incrementing i by 1. In case i is greater then the maximum level number, go to next step. Otherwise, go to Step HF 2.

Step HF 5 [Perform deletion and terminate]

If the *h_spin(...)* function is in the deletion mode, call the *deletion(...)* function and pass it the permutation found in Step HF 3. Otherwise, return the integer array *krec* and terminate this function.

The *insert(...)* function

This function is called within the *h_spin(...)* function and is the actual insertion engine.

Step HIN 1 [Check load factor]

Check the load factor of the hashing table on level i (i is passed from the *h_spin(...)* function).

The load factor can be computed as:

$$\frac{number\_of\_nodes\_in\_legitimate\_state + number\_of\_nodes\_in\_deleted\_state}{table\_size}.$$

If the load factor is greater than *0.5*, call the *rehashing(...)* function and pass it the hashing table to rehash.

Step HIN 2 [Get hash position]

Use the hashing function on level *i* and the unique identifier *V* (passed from the *h_spin(...)* function) to get a tentative hash position on level *i* for the inserted node. Then pass this tentative hash position and the identifier *V* to the *search_for_insert(...)* function to get a final hash position.

Step HIN 3 [Store the node information on level *i* and level *i-1*]

On level *i*, check the status of the node in the final hash position.
If the node is legitimate, the inserted node has already been in this node and nothing will be done.
If the node is not legitimate, store the *V* value in the node found in Step HF 3, change the node to be legitimate, and insert the index value of dimension *i* to the beginning of the linked-list attached to the parent node of the permutation on level *i-1*.

Step HIN 4 [Terminate]

Return the final hash position on level *i*. and terminate this function.

The *search(...)* function

This function just handles exact-match searching, and is called within the *h_spin(...)* function.

Step HSE 1 [Get a tentative hash position]

Use the hashing function on level *i* and the unique identifier *V* (*i* and *V* are the parameters passed from the *h_spin(...)* function) to get a tentative hash position on level *i* for the searched node.

Step HSE 2 [Compute the final hash position and terminate]

Check the node state and the key value stored in this node. If the node is not empty and the key stored in this node is not equal to the *V* value (collision situation), use the quadratic probing method to go to the next position and then repeat checking. Repeat this step and update the hash position until either a empty node is found, or a matched key value is found. Then, return the final hash position on level *i* and terminate.

The *deletion(...)* function

As we discussed in the *h_spin(...)* function, before deletion, the *search(...)* function is used to search for the deleted permutation. Therefore, the permutation passed to the *deletion(...)* function is guaranteed to be the one found in the H_SPIN hashing tables. "lazy deletion" is employed in this function.

Step HDE 1 [Mark the node]

Set *cnt* = the index value of the final level.

On level *cnt*, change the state of the deleted node to deleted. Then, delete the index value from the linked-list attached to its parent node on level *cnt-1*.

Step HDE 2 [Go up one level]

Decrement *cnt* by *1*.

Check the linked-list attached to the deleted node on level *cnt*.

If the linked-list is empty, change the state of this node to deleted. Then, delete the index value from the linked-list attached to its parent node on level *cnt-1*.

If the linked-list is not empty, this node has some children on the next level and nothing will be done.

Repeat this step until *cnt* = *0*, and then terminate.

The *rehashing(...)* function

This functions handles the expansion of a hashing table of H_SPIN.

Step HRH 1 [Choose new table size]

Get the total number of the nodes in legitimate state (i.e. *number_of_nodes_in_legitimate_state*) from the hashing table passed to

this function. Make the new table size as:

$4 \times number\_of\_nodes\_in\_legitimate\_state$.

Step HRH 2 [Create, initialize and load new table]

Create a new hashing table with the table size calculated in last step. Initialize all the nodes in this new hashing table. Then, call the *search_for_insert(...)* function to hash all the nodes in legitimate state in the old hashing table to the new hashing table.

Step HRH 3 [Terminate]

Free the space of the old hashing table. Return the pointer pointing to the new hashing table and terminate this function.

The *search_for_insert(...)* function

This function is designed to search for an empty position for insertions.

Step SFI 1 [Collision handling]

Check the state of the node in the tentative hash position (this position is passed from the calling function outside). If the node is legitimate and the key stored in this node is not equal to the searched key value (this position is passed from the calling function outside), use the quadratic probing method to go to next position and then repeat

checking. Repeat this step and update the hash position until either a empty node is found, or a matched key value is found. Then, return the final position and terminate this function.

The algorithm for partial-match searching

Since the results of partial-match searching has a bunch of permutations, instead of a single permutation, a linked-list, "*RETURN_LIST*" is defined to contain the searching results and is passed to the two functions which we will introduce in this section. "*RETURN_LIST*" has two integer arrays: *permu* which stores the found permutation and *permu_ptr* which stores the permutation's corresponding positions in the hashing tables of H_SPIN.

As all the valid indexes of a permutation must be greater than or equal to 0, the invalid index value "*-1*" is used to indicate the partial-match searching request on a level. Therefore, a partial-match search query can be determined by the index values that a permutation contains (no matter valid or invalid). In case that users pass a permutation as a search query with all valid indexes on all levels, the search operation performed by the following two functions becomes an exact-match search.

The *h_spin_par_search(...)* function

In the previous sections, we discussed that we not only store the identifier of a node in the hashing table on the current level, but also store the permutation index value of this node on its parent level for the later on partial-match search operations. Since the hashing table on level 0 dose not have its parent level, the implementation of the partial-match searching on level 0 is different from the other levels. On level 0, the program goes through the whole level, instead of going through a small portion of a hashing table on the other levels..

The *h_spin_par_search(...)* function is the one that handles the partial-match searching case on level 0. A partial-match search operation must begin with this function.

Step HSPS 1 [Check index value]

Check the level 0 index value of the searched permutation.

If it is less than 0, go to the next step.

If it is greater than 0, go to Step HSPS 3

Step HSPS 2 [Partial-match search on level 0]

Set $i = 0$.

Check the state of node i. If it is legitimate, call the *h_spin_partial(...)* function, and pass an instance of "*RETURN_LIST*"

and 0 (the level number) to it to begin the search operation on level 0. Increment *i* value by 1 and repeat this step until *i* is equal to the table size of level 0. Then terminate this function.

Step HSPS 3 [Go down one level]

Call the *h_spin_partial(...)* function, and pass an instance of "*RETURN_LIST*" and 0 (the level number) to it to begin the partial-match search operation on level 0. Then terminate this function.

The *h_spin_partial(...)* function

This function is the actual partial-match searching engine of H_SPIN. It is called within the *h_spin_par_search(...)* function to process the partial-match search operations from level 0 to the final level.

This is a recursive search function. Before calling this function, set level number "*x*" to 0, and create a link-list "*rtn_list*", an instance, of "*RETURN_LIST*." Then, call this function on level 0 (i.e. pass level number "*x*" to this function), and pass *rtn_list* to this function.

Step HSP 1 [Initialization]

Compute the parameters by using the formula (1).

Create two temporary integer arrays: *int_per* which contains the indexes of a found permutation, and *int_str* which contains the corresponding positions of this permutation in all the hashing tables of H_SPIN.

Step HSP 2 [Check index value on level *x*]

On level *x*, check the index value of the permutation passed in as a partial-match search query.

If it is greater than or equal to 0, go to Step HSP 3.

If it is less than 0, go to Step HSP 6.

Step HSP 3 [No partial-match search requested]

On level *x*, Use formula (1) and the index values of the permutation to convert the sparse multidimensional subscripts into a one dimensional index value, *V*.

Step HSP 4 [Store temporary values]

Assign the index value of level *x* to *int_per[x]*.

Pass *V* to the *search(...)* function and assign the return value to *int_str[x]*.

Step HSP 5 [Check the level number *x*]

Check the level number *x*.

If the final level is reached, assign *int_per* and *int_str* to a temporary "*RETURN_LIST*" node, then insert this node into *rtn_list* and then terminate this function.

If the final level is not reached, call the *h_spin_partial(...)* recursively on level *x+1*.

Step HSP 6 [Partial-match search requested]

Use *int_str[x-1]* to locate a node in the hashing table on level *x-1*, assign the pointer "*next*" in the head of the linked-list that is contained in this node to a temporary pointer *p_mark*.

Step HSP 7 [Check the linked-list]

If *p_mark* is equal to *NULL*, terminate this function.

If *p_mark* is not equal to *NULL*, go to the next step.

Step HSP 8 [Calculate identifier]

Use formula (1) and the index value stored in the node pointed by *p_mark* to compute the identifier for level *x*.

Assign the index value to *int_per[x]*.

Locate a node on level *x* using this identifier and the *search(...)* function.

Assign the return value of the *search(...)* function to *int_str[x]*.

Step HSP 9 [Check the level number "*x*"]

Check the level number "*x*."

If the final level is reached, assign *int_per* and *int_str* to a temporary "*RETURN_LIST*" node, insert this node into *rtn_list* and then go to next step.

If the final level is not reached, call the *h_spin_partial(...)* recursively on level $x+1$.

Step HSP 10 [Go to the next node]

Assign *p_mark->next* to *p_mark*.

Brief comparison of H_SPIN and R_SPIN

H_SPIN can do what R_SPIN does

1. Both H_SPIN and R_SPIN can handle sparse representations.
2. Both H_SPIN and R_SPIN can return the indexes for all levels.
3. Both H_SPIN and R_SPIN can implement partial-match searching.

Advantages of the H_SPIN algorithm

1. H_SPIN ($O(n)$) has much better time complexity than R_SPIN ($O(\log(n))$). Since the hashing table on each level has constant time behavior, H_SPIN will also have constant time complexity, while our experiments and pattern analysis showed that R_SPIN has logarithmic time complexity.
2. H_SPIN totally avoids the overflow situation. Instead, H_SPIN has collision problems. However, there are many existing good

solutions that can handle the collision problems very well [Weiss, 1993].

3. The growth handling of H_SPIN is much better and more flexible than the one of R_SPIN.

Disadvantages of the H_SPIN algorithm

1. The range search of H_SPIN is inefficient. Since in each hashing table of H_SPIN, the nodes are not stored sequentially according to the magnitude of their identifiers, range search can not be implemented consecutively on each level.
2. Since each parent node of H_SPIN keeps its children's information for partial-match searching, this makes H_SPIN possibly need more space than R_SPIN.

Chapter V

Experimental Comparison of R_SPIN and H_SPIN

In order to compare R_SPIN and H_SPIN on the same bases and conditions, both of the indexes of R_SPIN and H_SPIN are implemented in the main storage. The testing program introduced in this chapter is listed in Appendix B.

Testing Program

Using randomly generated multi-dimensional index permutations, the testing program experimentally examines the average time complexities of R_SPIN and H_SPIN. The insert operations of both R_SPIN and H_SPIN are tested repeatedly using the same permutation array, and so are the search operations of them. The steps of this program are:

Step TP 1 [Generate Permutations]

Recursively generate distinct multidimensional subscript combinations (or permutations), which to be used during testing. In order to access these experimental data quickly, they are stored in main storage.

Step TP 2 [Set total times]

Set a number "*N*" for the total number of testing cycles. In each testing cycle, a permutation is passed to the R_SPIN function, or the H_SPIN function.

Step TP 3 [Indexing Permutations]

In order to retrieve the sparse test data, an index of the permutations is established. This index is used to order the permutations passed to the R_SPIN function, or the H_SPIN functions, it is different from the index used in the R_SPIN function, or the hashing tables in the H_SPIN function. The implementation of this index can be accomplished by using an integer array. A random number generator randomly generates an integer, and passes the integer to a program to check. If this number is not in the integer array, then append it to the tail of the array. If it is already in the array, repeat this step with a new value. The integer array is guaranteed to contain distinct indexes of permutations.

Step TP 4 [Initialize R_SPIN and H_SPIN]

Since many static parameters are involved in the R_SPIN function and the H_SPIN function, and their computation takes some time, the R_SPIN function and the H_SPIN functions must be initialized before testing begins.

Step TP 5 [Measure the overhead for insertions]

Since this program tests the average time behaviors of R_SPIN and H_SPIN, the R_SPIN function and the H_SPIN function are repeated within a testing loop. In this step, a loop is implemented with all the necessary instructions except the functional call to R_SPIN or H_SPIN. The running time of this loop is recorded and will be subtracted from the running times of the following two loops.

Step TP 6 [Insertion loop for R_SPIN]

This loop tests the necessary instructions and the function call to R_SPIN. During execution time of this loop, the program processes the integer array created in Step TP 3 and passes each permutation to the R_SPIN function in each cycle. Since all the permutations in the integer array are distinct, each functional call to R_SPIN within the loop is the insertion operation of one permutation. The total running time of this loop is recorded.

Step TP 7 [Insertion loop for H_SPIN]

This loop is identical to Step TP 6, except that the R_SPIN function is replaced by the H_SPIN function in the inserting mode.

Step TP 8 [Re-indexing permutations]

Use the same method as discussed in Step TP 3 to generate some distinct index numbers from the integer array created in Step TP 3. Store these distinct indexes in another integer array.

Step TP 9 [Measure the overhead for searches]

This step is almost as the same as Step TP 5, except going through the integer array created in Step TP 8.

Step TP 10 [Search loop for R_SPIN]

This step is almost as the same as Step TP 6, except going through the integer array created in Step TP 8. Since all the permutations involved in this step are already inserted in Step TP 6, each functional call to R_SPIN in the loop is a search operation of one permutation. The total running time of this loop is recorded.

Step TP 11 [Search loop for H_SPIN]

This loop is identical to Step TP 9, except that the R_SPIN function is replaced by the H_SPIN function in the searching mode.

Discussion of the experimental results

Both the H_SPIN and the R_SPIN functions were tested on the LINUX (UNIX-like) and the MS-DOS operating systems, which both are installed on a IBM-compatible PC platform. In order to measure the running time accurately, all the hashing tables in the testing programs are created with a table size large enough to avoid the rehashing problem. The testing program on LINUX is listed in Appendix A.

Since MS-DOS has 640K megabytes primary main storage and heavy use of the main storage always causes space allocation failure, the testing program on MS-DOS does not involve the linked-lists for the usage of partial-match searching. Because this program is just the simplified version of the testing program on LINUX, it is not listed in listed, rather together with the other programs, it is filed in the Computer Science Department at Oklahoma State University.

Testing results on LINUX

The testing results of two experiments on LINUX are shown from figure V-1 to figure V-4. Each experiment involved the tests of insert and search operations. The upper curves in these figures represent the insertion or search average times of R_SPIN, and the lower curves represent the ones of H_SPIN.

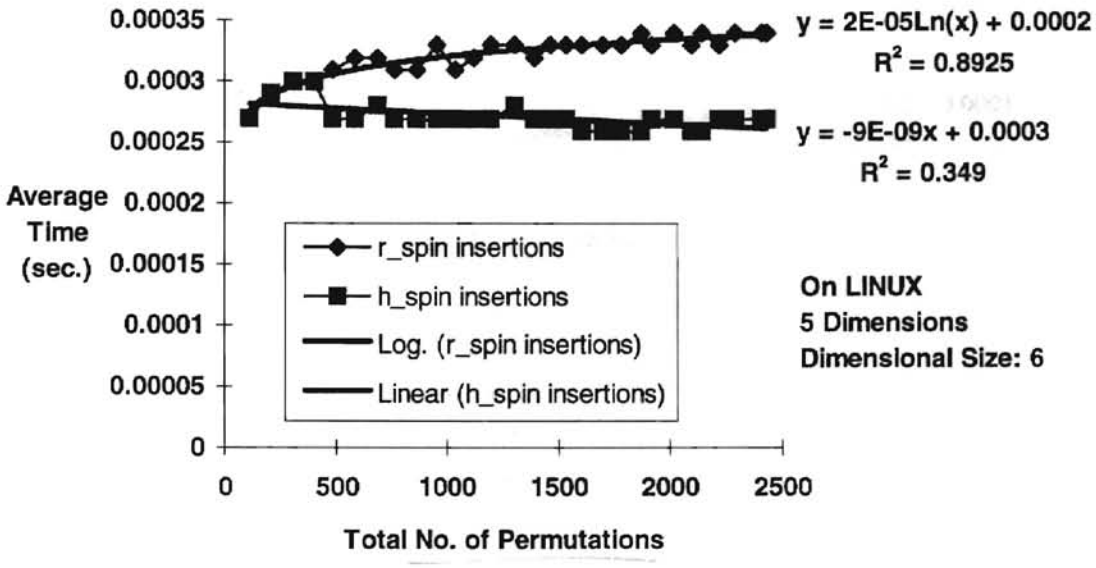

Figure V-1 Experimental results of insert operation on LINUX

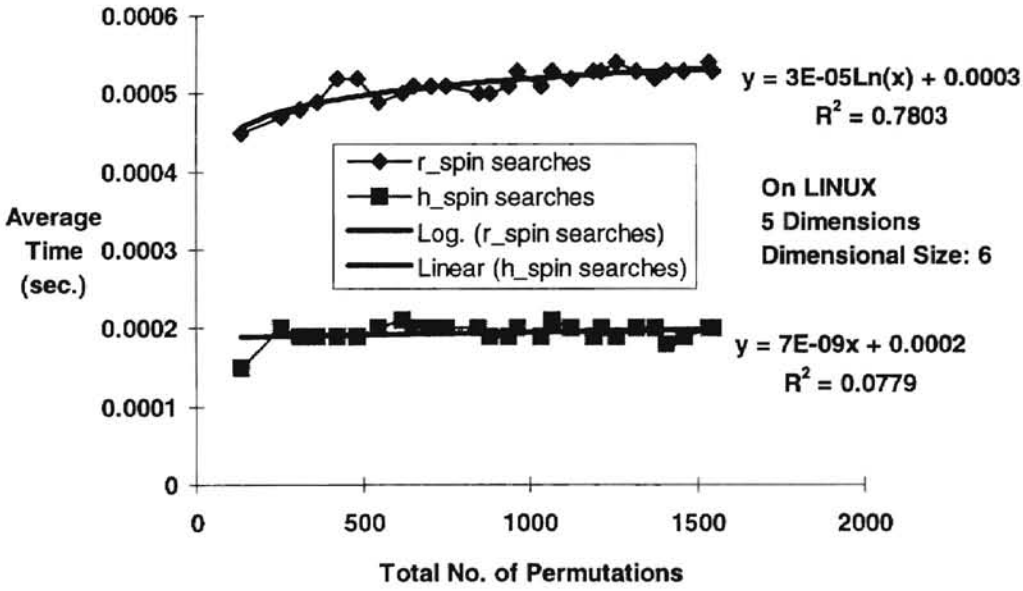

Figure V-2 Experimental results of search operation on LINUX

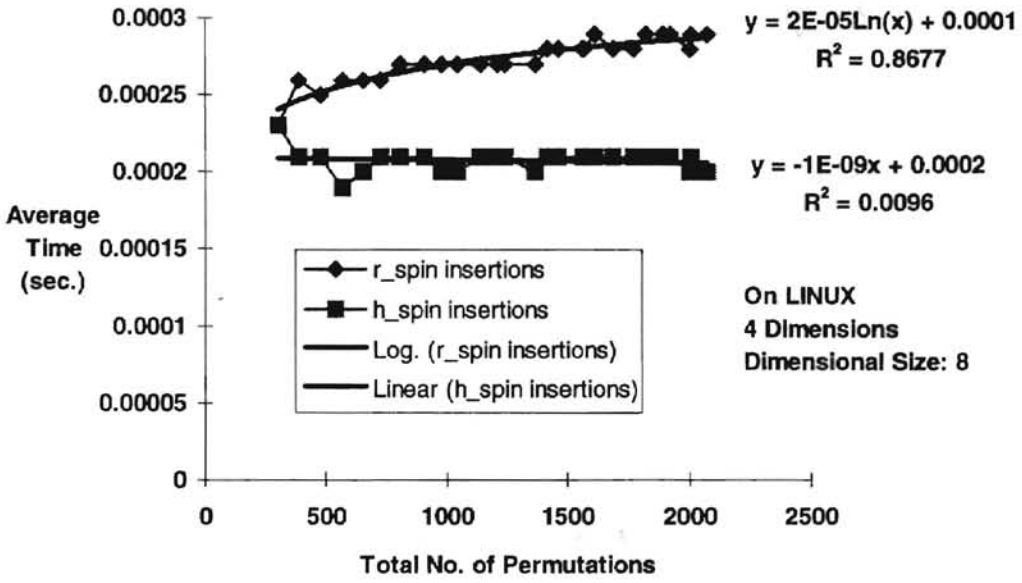

Figure V-3 Experimental results of insert operation on LINUX

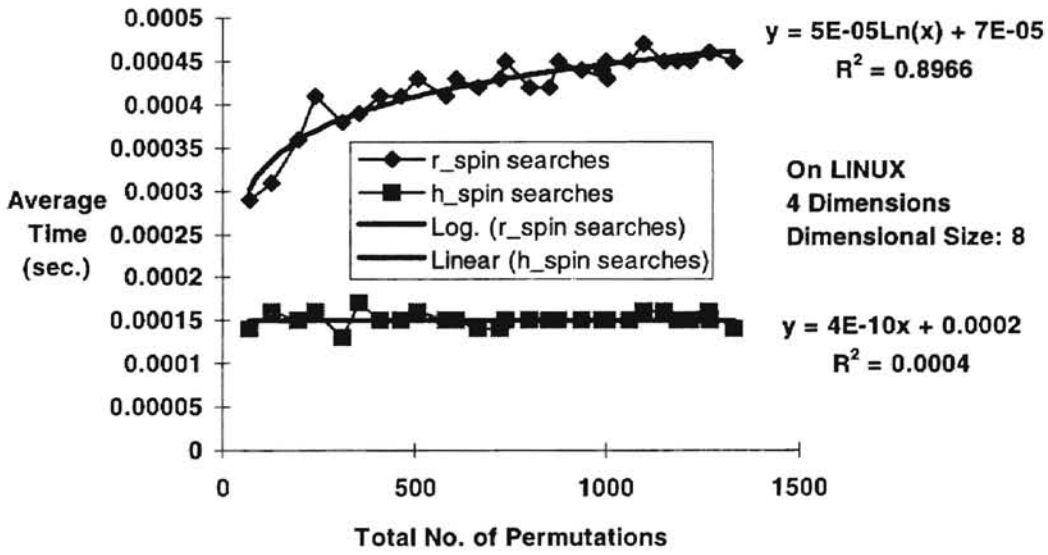

Figure V-4 Experimental results of search operation on LINUX

Testing results on MS-DOS

The testing results of two experiments on MS-DOS are shown from figure V-5 to figure V-8.

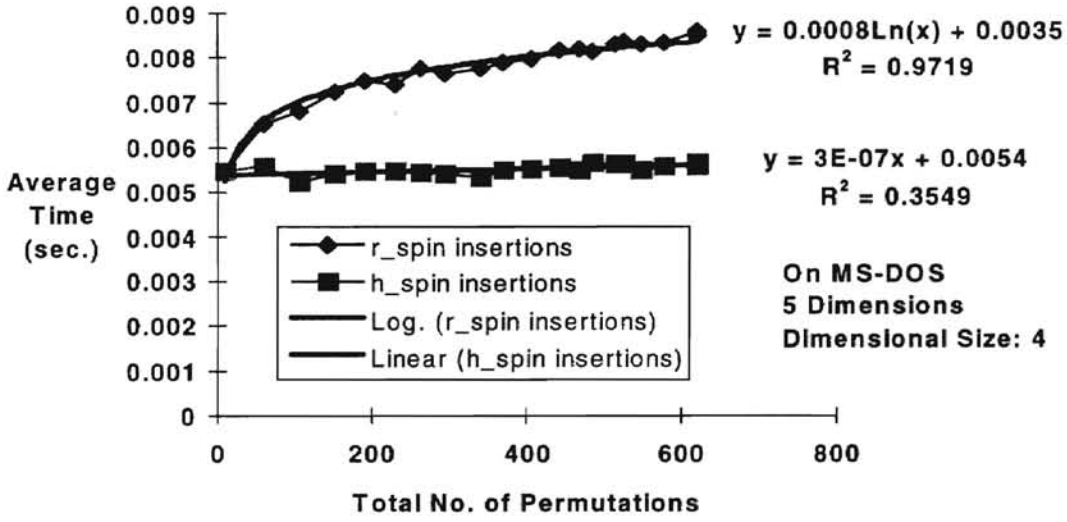

Figure V-5 Experimental results of insert operations on MS-DOS

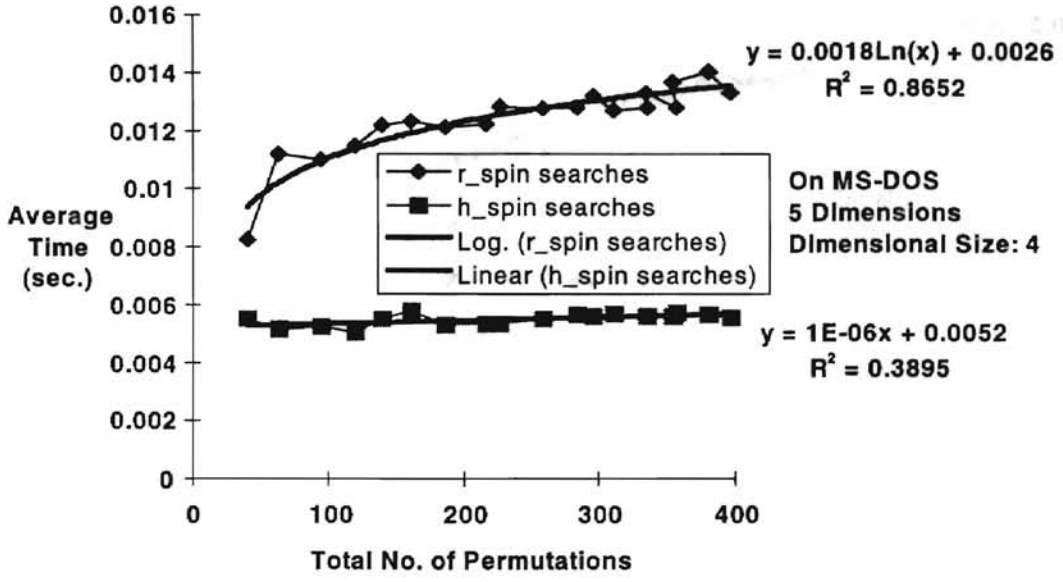

Figure V-6 Experimental results of search operations on MS-DOS

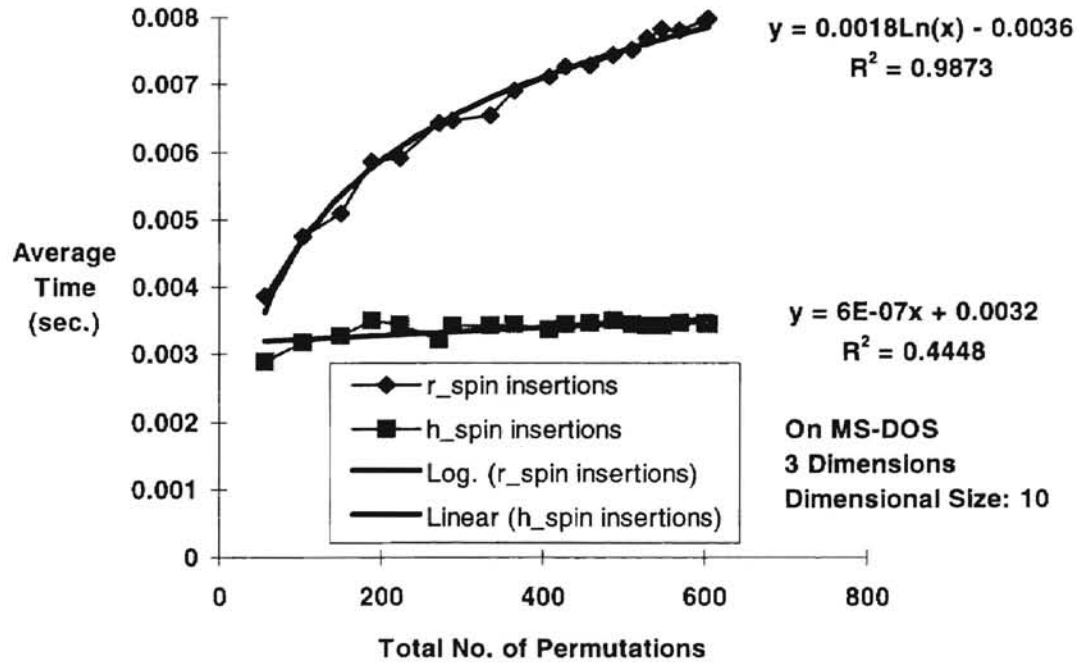

Figure V-7 Experimental results of search operations on MS-DOS

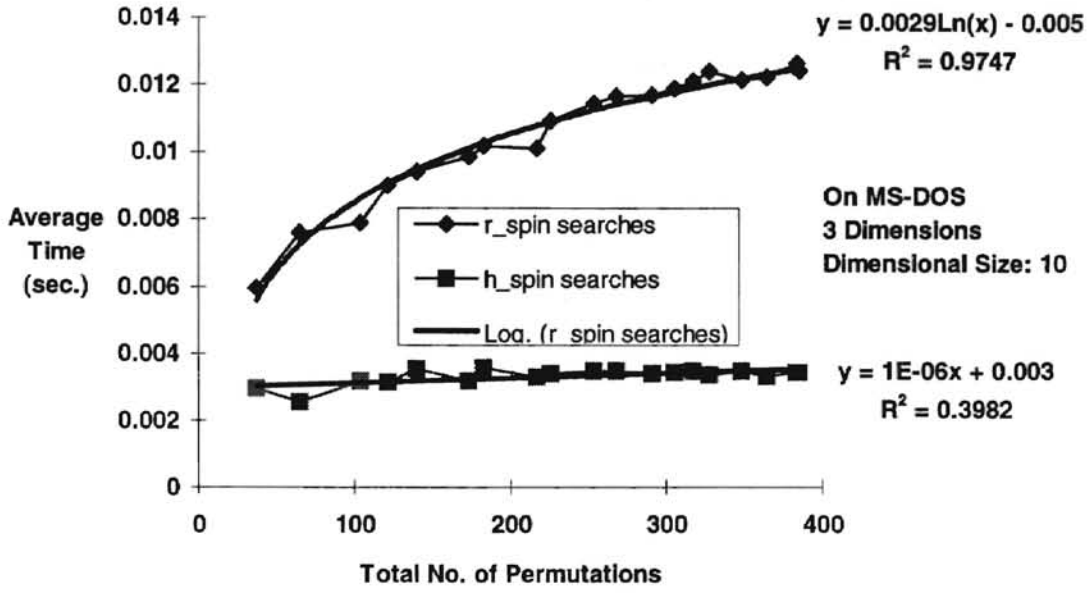

Figure V-8 Experimental results of search operation on MS-DOS

Discussion

Our experiments show that both insert and search operations of R_SPIN have an average running time of O(log(n)). The fitted curves for the R_SPIN operations in the figures from figure V-1 to V-8 also demonstrated that the running time of R_SPIN consists of two parts: the logarithmic variable part and the constant part. The constant part of the average running time of R_SPIN should not be less than zero. Thc negative constants (-0.0036 and -0.005) of the lower fitted curves in figure V-7 and V-8 are due to inaccurate experimental data.

On the LINUX operating system, our test results indicate the constant running behavior of H_SPIN, which perfectly matches the analytical solution of hashing methods.

However, the tests in the figures from figure V-5 to V-8 show that on MS-DOS operating system, the H_SPIN operations (insertions and searches) have linear behavior with a small slope. The reason for this linear performance of H_SPIN is possibly due to the increasing number of collisions within the H_SPIN hashing tables when some of the hashing tables are close to half full (the approximate maximum load factor to obtain constant time complexity in closed hashing is 0.5) [Weiss, 1993]. Since these slopes are very small and the small slopes make the H_SPIN operations increase their average running time slowly, we still can consider that the time complexity of H_SPIN is close to constant. Careful calculations indicate that in the figures from figure V-5 to V-8, the linear curves will not be greater than the logarithmic curves until the number of permutations increases to about $2x10^4$.

Chapter VI

The Demonstration Program for H_SPIN

The purpose of this program is to demonstrate users the H_SPIN package, including the insertion function, the search function, the deletion function, the rehashing function (self-growth handling) and the partial-match search function. This program is listed in Appendix C.

Programmer's guide

This program provides four optional actions to users, which are "Insertion," "List all," "Search" and "Deletion." "Insertion" lets users to insert one permutation to the H_SPIN hashing tables. "List all" will have the program list all the stored permutations. "Search" lets users perform either partial-match or exact-match searching. Users can delete a permutation from the H_SPIN hashing tables by selecting the option "Deletion." The implementations of them are introduced in this subsection.

"Insertion" and "Deletion" are implemented by calling the *h_spin(...)* function in insert mode and delete mode respectively. The

rehashing(...) function is automatically called when the load factor in any of the hashing tables of H_SPIN reaches 0.5.

Both "List all" and "Search" are performed by calling the *h_spin_par_search(...)* function. When users choose "List all," the program will automatically assign "*-1*" to all the indexes of the permutation input to the *h_spin_par_search(...)* function. This will cause partial-match searching on every level. "Search" can conduct partial-match searching according to the user's request. In case that users input "*-1*" on all the levels after "Search" is chosen, "Search" becomes "List all". On the other hand, if users input valid indexes (non-negative integers) on all the levels after "Search" is chosen, "Search" becomes exact-match search operation.

User's guide

The procedures of using this demonstration program can be easily illustrated by an example.

When the program is invoked, it asks users to input the number of dimensions, the dimensional size, the initial number of distinct permutations created by the program and the initial table size on each level. Suppose we want 3 dimensions and dimensional size of 3, and let the program maximally generate 10 permutations for us. Figure VI-1 shows the program prompts and the input parameters from users.

```
Please input the number of dimensions ( >= 2): 3
Please input size of dimension ( >= 2): 3
Please input the total number of initial permutations: 10
Please input the hashing table size for each level 0: 5
Please input the hashing table size for each level 1: 10
Please input the hashing table size for each level 2: 10
```

Figure VI-1 The program prompts and the input parameters

After all the necessary parameters are typed in, the program starts to maximally generate 10 distinct permutations, and then inserts them into the H_SPIN hashing tables. Since we input small table sizes just now, the program also prints out the rehashing messages, including the level number on which the hashing table is expanded, the load factor before rehashing and the new table size after rehashing. Figure VI-2 shows these messages.

```
Generating permutations
Begin insertions using randomly-ordered permutations.
hashing table 0 expansion; load factor: 0.57; table size: 15.
hashing table 2 expansion; load factor: 0.55; table size: 23.
hashing table 1 expansion; load factor: 0.55; table size: 23.
```

Figure VI-2 The messages printed by the program

Figure VI-3 shows the option menu which is prompted after the program finishes the initial insertions. Now, let us select option 2, "List all" to see what are stored in the hashing tables. The figure VI-4 shows the results of "List all." Please note that figure VI-4 just shows

8 distinct permutations, instead of 10. This is because the number of the distinct permutations generated by the program may be less than the initial maximum number.

```
********************************
 Insertion ..................  1
 List all ...................  2
 Search .....................  3
 Deletion ...................  4
 Exit .......................  5
********************************
Choose: 2
```

Figure VI-3 The prompt menu

The permutations found:			The positions on each level:		
2	2	1	2	12	15
2	0	1	2	10	5
2	1	0	2	11	9
1	0	2	1	5	4
1	0	0	1	5	2
1	1	2	1	6	10
0	1	0	0	1	6
0	0	0	0	0	0

Figure VI-4 The results of "List all"

Insert two permutations of [1][1][1] and [2][2][2] by selecting option 1 twice, and then select option 2 to list all the permutations stored in the hashing tables. The results of the three actions are shown in figure VI-5.

The permutations found:			The positions on each level:		
2	2	1	2	12	15
2	2	2	2	12	16
2	0	1	2	10	5
2	1	0	2	11	9
1	0	2	1	5	4
1	0	0	1	5	2
1	1	2	1	6	10
1	1	1	1	6	8
0	1	0	0	1	6
0	0	0	0	0	0

Figure VI-5 The results after inserting two permutations

Next, let us try a partial-match search by selecting option 3 and then input the partial-match query of [2][-1][-1]. The results are shown in figure VI-6.

The permutations found:			The positions on each level:		
2	2	1	2	12	15
2	2	2	2	12	16
2	0	1	2	10	5
2	1	0	2	11	9

Figure VI-6 The results after partial-match searching

Finally, delete the two permutations of [1][1][1] and [2][2][2] by selecting option 4 twice, and then list all the permutations. The results are shown in figure VI-7.

The permutations found:			The positions on each level:		
2	2	1	2	12	15
2	0	1	2	10	5
2	1	0	2	11	9
1	0	2	1	5	4
1	0	0	1	5	2
1	1	2	1	6	10
0	1	0	0	1	6
0	0	0	0	0	0

Figure VI-7 The results after deletions

Chapter VII

Results and Discussion

In the experiments discussed in this paper, all the insertions and the searches are successful operations. Unsuccessful insertions and searches cause both the R_SPIN and the H_SPIN functions to terminate at some node between level 0 and the final level. The path length of an unsuccessful operation is shorter than the path length of a successful one. Therefore, we can conclude that:

- Both successful and unsuccessful insertion and search operations of R_SPIN have an average running time of $O(\log(n))$ when the index of R_SPIN are implemented in main storage.
- Both successful and unsuccessful insertion and search operations of H_SPIN have an almost constant average running time when the hashing tables of H_SPIN are implemented in main storage.
- H_SPIN is a faster and more flexible storage and retrieval method than R_SPIN.

Finally in order to apply H_SPIN in database systems, substantial additional research work is needed. The possible topics of the future research of H_SPIN may focus on:

- Search for the hashing method which is the best fit to H_SPIN.

- Reduce the collision ratio by designing and coordinating the hashing parameters and the SPIN parameters.
- Further the research on and perform the implementation of H_SPIN in object-oriented database.

BIBLIOGRAPHY

1. Bentley, J. L. Multidimensional Binary Search Trees in Database Applications. IEEE Transactions on Software Engineering, Vol. SE-5, No.4, pages 333 -- 340, July 1979.
2. Brothels, E. and Rotem, D. Database Design with the Constrained Multiple Attribute Tree. Information Systems, Vol.10, No.1, pages 47-56, 1985.
3. Coburn, Ty C. C SPIN toolkit (program documentation).Oklahoma City, OK: Ty Coburn, c1991.
4. Coburn, Ty C. An introduction to SPIN hashing: an approach to managing multidimensional spaces (Unpublished program documentation). Tinker AFB, OK: Oklahoma City Air Logistics Center.
5. Date, C. J. An Introduction to Database Systems. Addison-Wesley Publishing Company. Sixth Edition, c1995.
6. Harbron, T.R. File systems: structures and algorithms. Englewood Cliffs, NJ; Prentice Hall, Inc. c1987.
7. Hedrick, G., Fan, M., Zhang, Y., DiVall, R. A review of a new spatial data indexing technique. Technical Report #OSU-CS-TR-95-02, Stillwater, OK; Computer Science Department, Oklahoma State University, 1995.

8. Guttman, A. R-Trees: A Dynamic Index Structure For Spatial Searching. ACM SIGMOD, pages 47 -- 57, June 1984.

9. Kriegel, H. P. Performance Comparison of Index Structures for Multi-Key Retrieval. Proc. ACM SIGMOD, Boston, Massachusetts, pages 186 -- 196, 1983.

10. Knuth, D.E. The art of computer programming: V.1/Fundamental Algorithms. Addison-Wesley Publishing Company. c1975.

11. Lomet, David B. and Salzberg, B. The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance. ACM Transactions on Database System, Vol.15, 4, pages 625--658, Dec. 1990.

12. Moret, B.M.E. Algorithms from P to NP (Vol. 1). Redwood City, CA; The Benjamin/Cummings Publishing Company, Inc. c1991.

13. Nievergelt, J. The Grid File: An Adaptable, Symmetric Multikey File Structure. ACM Transactions on Database Systems, Vol.9, No.1, pages 38 -- 71, March 1984.

14. Robinson, J. T. The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes. Proc. ACM SIGMOD, pages 10 -- 18, 1981.

15. Scheuermann, P and Ouksel, M. Multidimensional B-trees for Associative Search in Database Systems. Information Systems, Vol.7, No.2, pages 123-137, 1982.

16. Servio Corporation. GemStone Reference Manual. Beaverton, Ore. 1990.

17. Weiss, M. A. Data structure and algorithm analysis in C. Redwood City, CA; The Benjamin/Cummings Publishing Company, Inc. c1993.

18. Zhang, Y., DiVall, A., Fan, M., Hedrick, G. An experimental analysis of a new multi-dimensional storage and retrieval method. Technical Report #OSU-CS-TR-95-04, Stillwater, OK; Computer Science Department, Oklahoma State University, 1995.

Appendixes

Appendix A

/* Appendix A includes the header file, and the necessary subprograms for the H_SPIN package. In order to run the testing program and the demonstration program listed in Appendix B and C, users must create a library file with the functions listed in the appendix, and include the created library file. */

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <time.h>

#define MAXSTRING  100

enum kind_of_entry{legitimate, empty, deleted};

typedef  struct node        *node_ptr;
 /* structure of linked-list attached to the cell node in a hashing table */
struct node {
   unsigned int    element;
   node_ptr        next;
};

typedef  node_ptrLIST;
typedef  node_ptrLIST_POS;

typedef struct return_node *return_node_ptr;
/* linked-list containing return information from the h_spin search */
struct return_node {
   int *                    permu;                    /* integer array containing permutations */
   int *                    permu_ptr;                /* integer array containing positions */
   return_node_ptr          next;
};

typedef  return_node_ptr   RETURN_LIST;
typedef  return_node_ptr   RETURN_LIST_POS;
/* cell node structure of a hashing table */
struct hash_entry {
   int              element;                          /* the unique idenfier */
   enum             kind_of_entry info;               /* node status information */
   LIST             list_ptr;                         /* pointer of the linked list attached to this node */
   int              p_flag;                           /* partial-match search flag */
   node_ptr         p_mark;                           /* position marker for partial-match searching */
};

typedef  int                position;
typedef  struct hash_entry  cell;

/* hashing table structure */
struct hash_table {
   unsigned int    table_size;
```

```
  unsigned int    table_load_cnt;
  unsigned int    table_legit_cnt;
  cell            *the_cells;
};

typedef  struct hash_table  *HASH_TABLE;

HASH_TABLE* initialize_hash_tables(unsigned int *, unsigned int);
unsigned int      next_prime(unsigned int);
int               search(HASH_TABLE, int);
int               insert(HASH_TABLE *, int, int, int, unsigned int *);
void              list_insert(HASH_TABLE, int, int);
void              deletion(HASH_TABLE *, int, int *, unsigned int *);
unsigned int      hashing(unsigned int, unsigned int);
HASH_TABLE        rehashing(HASH_TABLE);
int               search_for_insert(HASH_TABLE, int, int);
int               h_spin_par_search(HASH_TABLE *, RETURN_LIST, unsigned int, \
                                    unsigned int, unsigned int *, int *, int);
void              free_hash_tables(HASH_TABLE *, int);
int *             h_spin(HASH_TABLE *, unsigned int, unsigned int, \
                                    unsigned int *, unsigned int *, int, char);
int               input_int();
int               input_positive_int(int);
unsigned long int *r_spin(long int *, unsigned int, unsigned int, \
                                    unsigned int *, unsigned int *, unsigned int *, char, int);
int               input_int();
int               input_positive_int(int);
int               pop_menu();

/*  This program is the h_spin function */

#include "spin.h"

int       *h_spin(HASH_TABLE *H, unsigned int N, unsigned int L, \
                                    unsigned int *max,  unsigned int *k, int control, char mode)
{
     register int                 y,u;
     static unsigned long int     *kr,*rect;
     static unsigned int          *ex;
     static int                   ncontrol=0,*krec;
     unsigned long int            kstore,hold, temp_sub;
     long int                     *temp_str;
     int                          last_level_pos;

     int                          insert(HASH_TABLE *, int, int, int , unsigned int *);
     int                          search(HASH_TABLE, int);
     void                         deletion(HASH_TABLE *, int, int *, unsigned int *);

     if(control == 'f') {                                    /* free the space if in 'f' mode */
       /* FREE POINTERS FOR NEW ARRAY */
       free(kr);
       free(krec);
       free(rect);
       free(ex);
       return krec;
```

```
}
if(control != ncontrol) {                          /* shut down the access to this block */
                                                   /* after the first access */
  /* SET CONTROL */
  ncontrol = control;
  /* ALLOCATE MEMORY TO POINTERS */
  rect = (unsigned long int *)calloc(N,sizeof(long int));
  if(rect == NULL) {
    puts("allocation error in function r_spin 1\n");
    exit(0);
  }
  kr = (unsigned long int *)calloc(N, sizeof(long int));
  if(kr == NULL) {
    puts("allocation error in function r_spin 2 \n");
    exit(0);
  }
  ex = (unsigned *) calloc(N-1,sizeof(int));
  if(ex == NULL) {
    puts("allocation error in function r_spin 3\n");
    exit(0);
  }
  krec = (int *) calloc(N, sizeof(int));
  if(krec == NULL) {
    puts("allocation error in function r_spin 4\n");
    exit(0);
  }
  /* COMPUTER PARAMETERS */
  for(y=1;y<N;y++) {
    if(*(max+y) > 0 && *(max+y) <= 10) {
      *(ex+y-1) = 1;
      *(kr+y-1) = (unsigned long int)9-*(max+y)+1;
    } else if(*(max+y) > 10 && *(max+y) <= 100) {
      *(ex+y-1) = 2;
      *(kr+y-1) = (unsigned long int)99-*(max+y)+1;
    } else if(*(max+y) > 100 && *(max+y) <= 1000) {
      *(ex+y-1) = 3;
      *(kr+y-1) = (unsigned long int)999-*(max+y)+1;
    } else if(*(max+y) > 1000 && *(max+y) <= 10000) {
      *(ex+y-1) = 4;
      *(kr+y-1) = (unsigned long int)9999-*(max+y)+1;
    } else {
      *(ex+y-1) = 5;
      *(kr+y-1) = (unsigned long int)99999L-*(max+y)+1;
    }
  }
}
/* if it is in initial mode */
if(mode == 'i') {
  for(y=0; y < L; ++y)
    krec[y] = -1;
  return krec;
}
/* COMPUTE RETURN VALUE */
if(L < 1 || L > N-1) {
  puts("Illegal level value in function r_spin\n");
```

```
    krec[L-1] = (unsigned long) -1;
    return krec;
  }

  for(y=0;y < N;y++) {
    if(*(k+y) > *(max+y)-1) {
      puts("subscript out of range in function r_spin\n");
      krec[L-1] = (unsigned long) -1;
      return krec;
    }

    if(y == 0) {                                          /* initialize for the search and insert */
      *rect = *k;
      last_level_pos = 0;
    } else *(rect+y) = *(rect+y-1)*pow((double)10, (double)*(ex+y-1))+*(k+y)-*(rect+y-1)**(kr+y-1);
    kstore = *(rect+y);

    switch(mode) {
      case 'p': krec[y] = insert(H, y, last_level_pos, kstore, k);          /* for insert*/
            last_level_pos = krec[y];
            break;
      case 's': krec[y] = search(H[y], kstore);                             /* for search */
            if(H[y]->the_cells[krec[y]].info != legitimate) {
              /* if the index value is not on this level */
              *(krec) = (unsigned int)-1;                                   /* search fail */
              return krec;
            }
            break;
      case 'd': krec[y] = search(H[y], kstore);                             /*search for deletion*/
            if(H[y]->the_cells[krec[y]].info != legitimate) {
              /* if the index value is not on this level  */
              *(krec) = (unsigned int)-1;                                   /* deletion fail */
              return krec;
            }
    }
  }

  if(mode == 'd')
    deletion(H, N, krec, k);                              /*begin deletion if it is in delete mode*/
  return krec;
}


/*  this is the program for the partial-match search of h_spin */

#include "spin.h"

int       h_spin_par_search(HASH_TABLE *H, RETURN_LIST rtn_list, unsigned int N, \
                                          unsigned int L, unsigned int *max, int *k, int control)
{
    int           i, j;
    int           *kk;
    int           h_spin_partial(HASH_TABLE *, RETURN_LIST, unsigned int, \
                                          unsigned int, unsigned int *, int *, int, int);
```

```
    if(k[0] < 0) {                                  /* go through level 0, if user requests so. */
      for(i=0; i < H[0]->table_size; ++i) {
        if(H[0]->the_cells[i].info == legitimate) {
          kk = (int*) malloc(sizeof(int)*N);
          kk[0] = i;
          for(j=1; j < N; ++j)
            kk[j] = k[j];
          h_spin_partial(H,rtn_list,N,L,max,kk,control,0);
          free(kk);
        }
      }
    } else                                          /* no partial-match search requested by user */
      h_spin_partial(H,rtn_list,N,L,max,k,control,0);
    return 0;
}


/* this is the actual engine of the partial-match searching */
int     h_spin_partial(HASH_TABLE *H, RETURN_LIST rtn_list, unsigned int N, \
                                        unsigned int L, unsigned int *max, int *k, int control, int x)
{
    register int                y,u;
    static unsigned long int    *kr,*rect;
    static unsigned int         *ex;
    static int                  ncontrol=0,*krec, *int_str, *int_per;
    unsigned long int           kstore,hold, temp_sub;
    long int                    *temp_str;
    int                         last_level_pos, temp_k;
    RETURN_LIST                 temp_list;

    int                         search(HASH_TABLE, int);

    if(control == 'f') {
        /* FREE POINTERS FOR NEW ARRAY */
        free(kr);
        free(krec);
        free(rect);
        free(ex);
        free(int_per);
        free(int_str);
        return 1;
    }
    if(control != ncontrol) {                       /* shut down the access to this block */
                                                    /* after the first access */
        /* SET CONTROL */
        ncontrol = control;
        /* ALLOCATE MEMORY TO POINTERS */
        rect = (unsigned long int *)calloc(N,sizeof(long int));
        if(rect == NULL) {
          puts("allocation error in function r_spin 1\n");
          exit(0);
        }
        kr = (unsigned long int *)calloc(N, sizeof(long int));
        if(kr == NULL) {
          puts("allocation error in function r_spin 2 \n");
```

```
    exit(0);
  }
  ex = (unsigned *) calloc(N-1,sizeof(int));
  if(ex == NULL) {
    puts("allocation error in function r_spin 3\n");
    exit(0);
  }
  krec = (int *) calloc(N, sizeof(int));
  if(krec == NULL) {
    puts("allocation error in function r_spin 4\n");
    exit(0);
  }
  int_str = (int *) calloc(N, sizeof(int));
  if(int_str == NULL) {
    puts("allocation error in function r_spin 4\n");
    exit(0);
  }
  int_per = (int *) calloc(N, sizeof(int));
  if(int_per == NULL) {
    puts("allocation error in function r_spin 4\n");
    exit(0);
  }
  /* COMPUTER PARAMETERS */
  for(y=1;y<N;y++) {
    if(*(max+y) > 0 && *(max+y) <= 10) {
          *(ex+y-1) = 1;
          *(kr+y-1) = (unsigned long int)9-*(max+y)+1;
    } else if(*(max+y) > 10 && *(max+y) <= 100) {
          *(ex+y-1) = 2;
          *(kr+y-1) = (unsigned long int)99-*(max+y)+1;
    } else if(*(max+y) > 100 && *(max+y) <= 1000) {
          *(ex+y-1) = 3;
          *(kr+y-1) = (unsigned long int)999-*(max+y)+1;
    } else if(*(max+y) > 1000 && *(max+y) <= 10000) {
          *(ex+y-1) = 4;
          *(kr+y-1) = (unsigned long int)9999-*(max+y)+1;
    } else {
          *(ex+y-1) = 5;
          *(kr+y-1) = (unsigned long int)99999L-*(max+y)+1;
    }
  }
}
rect[0] = k[0];

/* COMPUTE RETURN VALUE */
if(L < 1 || L > N-1) {
  puts("Illegal level value in function r_spin\n");
  return -1;
}
if(k[x] >= 0) {                                    /* no partial-match serach needed */
  if(x == 0)                                       /* if this is the level 0, no calculation needed */
    kstore = k[x];
  else {                                           /* if this is not level 0, calculation is necessary */
    *(rect+x) = *(rect+x-1)*pow((double)10, \
                                (double)*(ex+x-1))+*(k+x)-*(rect+x-1)**(kr+x-1);
```

```
    kstore = *(rect+x);                                    /* store is the unique idenfier */
  }
  int_per[x] = k[x];                                  /* array storing permutation index values */
  int_str[x] = search(H[x], kstore);                          /* array storing positions */
  if(H[x]->the_cells[int_str[x]].info != legitimate)
    return -2;
  if(x == L) {                                          /* if we reach the terminating level */
    /* space allocation for the return linked list */
    temp_list = (RETURN_LIST) malloc(sizeof(struct return_node));
    if(temp_list == NULL) {
          printf("Memory allocation error \n");
          exit(0);
    }
    temp_list->permu = (int*) malloc(sizeof(int)*N);
    if(temp_list->permu == NULL) {
          printf("Memory allocation error \n");
          exit(0);
    }
    temp_list->permu_ptr = (int*) malloc(sizeof(int)*N);
    if(temp_list->permu_ptr == NULL) {
          printf("Memory allocation error \n");
          exit(0);
    }
    for(y=0; y <= L; ++y) {                             /* assign values to the arrays */
          temp_list->permu[y] = int_per[y];
          temp_list->permu_ptr[y] = int_str[y];
    }
    /* add the newly found permutation into return list */
    temp_list->next = rtn_list->next;
    rtn_list->next = temp_list;
    return 1;                                                  /* adding successfully */
  } else                                              /* if this is not the terminating level, go on  */
    h_spin_partial(H, rtn_list, N, L, max, k, control, x+1);
} else {                                          /* if partial-match search is requested by users */
  /* set flag for partial-match search on this level */
  if(H[x-1]->the_cells[int_str[x-1]].p_flag == 0) {
    /* mark the search position in the linked list */
    H[x-1]->the_cells[int_str[x-1]].p_mark = \
    H[x-1]->the_cells[int_str[x-1]].list_ptr->next;
    H[x-1]->the_cells[int_str[x-1]].p_flag = 1;
  }
  /* begin partial-match browing on the last level */
  while(H[x-1]->the_cells[int_str[x-1]].p_mark != NULL) {
    temp_k = H[x-1]->the_cells[int_str[x-1]].p_mark->element;
    *(rect+x) = *(rect+x-1)*pow((double)10, (double)*(ex+x-1))+temp_k-*(rect+x-1)**(kr+x-1);
    kstore = *(rect+x);                                /* kstore is the unique identifier */
    int_per[x] = temp_k;
    int_str[x] = search(H[x], kstore);                          /* search on this level */
    if(H[x]->the_cells[int_str[x]].info != legitimate) {
          printf("System error \n");                           /* this should not occur */
          return -2;
    }
    H[x-1]->the_cells[int_str[x-1]].p_mark = H[x-1]->the_cells[int_str[x-1]].p_mark->next;
    if(x == L) {                                      /* if we reach the terminating level */
          temp_list = (RETURN_LIST) malloc(sizeof(struct return_node));
```

```
                if(temp_list == NULL) {
                        printf("Memory allocation error \n");
                        exit(0);
                }
                temp_list->permu = (int*) malloc(sizeof(int)*N);
                if(temp_list->permu == NULL) {
                        printf("Memory allocation error \n");
                        exit(0);
                }
                temp_list->permu_ptr = (int*) malloc(sizeof(int)*N);
                if(temp_list->permu_ptr == NULL) {
                        printf("Memory allocation error \n");
                        exit(0);
                }
                for(y=0; y <= L; ++y) {                         /* assign values to the arraies */
                        temp_list->permu[y] = int_per[y];
                        temp_list->permu_ptr[y] = int_str[y];
                }
                /* add the newly found permutation into the return list */
                temp_list->next = rtn_list->next;                       /* push to stack */
                rtn_list->next = temp_list;
          } else                                                /* if this is not the terminating level */
                h_spin_partial(H, rtn_list, N, L, max, k, control, x+1);
        }
        H[x-1]->the_cells[int_str[x-1]].p_flag = 0;  /* set down flag */
    }
    return 0;
}


/*  this program is for the expansion of a hashing table on a level (rehashing) */

#include "spin.h"

HASH_TABLE  rehashing(HASH_TABLE H)
{
        int             i, pos;
        HASH_TABLE      temp;
        LIST            temp_ptr;

        int             search_for_insert(HASH_TABLE, int, int);
        unsigned int    hashing(unsigned int, unsigned int);

        /* create a new hashing table for substitution */
        temp = (HASH_TABLE) malloc(sizeof(struct hash_table));
        if(temp == NULL) {
                printf("Memory allocation error \n");
                exit(0);
        }
        /* rechoose the table size according to the substituted table */
        temp->table_size = next_prime(4*(H->table_legit_cnt));
        temp->table_load_cnt = H->table_legit_cnt;
        temp->table_legit_cnt = H->table_legit_cnt;
        temp->the_cells = (cell *) malloc(sizeof(cell)*temp->table_size);
        if(temp->the_cells == NULL) {
```

```
            printf("Memory allocation error \n");
            exit(0);
        }
        /* initialize the newly created hashing table */
        for(i=0; i < temp->table_size; ++i) {
            temp->the_cells[i].info = empty;
            temp->the_cells[i].element = -1;
            temp->the_cells[i].p_flag = 0;
            temp->the_cells[i].list_ptr = (LIST) malloc(sizeof(struct node));
            if(temp->the_cells[i].list_ptr == NULL) {
                printf("Memory allocation error\n");
                exit(0);
            }
            temp->the_cells[i].list_ptr->next = NULL;
        }
        /* transfer the data from old table to the new table */
        for(i=0; i < H->table_size; ++i) {
            if(H->the_cells[i].info == legitimate) {                    /* if data is valid */
                pos = hashing(temp->table_size, H->the_cells[i].element);
                /* search for the position in the table */
                pos = search_for_insert(temp, pos, H->the_cells[i].element);
                temp->the_cells[pos].info = legitimate;
                temp->the_cells[pos].element = H->the_cells[i].element;
                temp->the_cells[pos].list_ptr->element = H->the_cells[i].list_ptr->element;
                temp->the_cells[pos].list_ptr->next = H->the_cells[i].list_ptr->next;
                temp->the_cells[pos].p_flag = H->the_cells[i].p_flag;
            }
            free(H->the_cells[i].list_ptr);                    /* free data space in old table */
        }

        free(H);                                    /* free the pointer pointing to the old table */
        return temp;
}


/* this program includes some trival functions used in the spin demostration and testing programs */

#include "spin.h"

/* create and initialize the hashing tables */

HASH_TABLE* initialize_hash_tables(unsigned int *table_sizes, unsigned int no_of_tables)
{
        HASH_TABLE  *h_tables;
        int         i, j;

        unsigned int    next_prime(unsigned int);

        /* allocate hashing table pointers */
        h_tables = (HASH_TABLE *) malloc( no_of_tables * sizeof( HASH_TABLE));
        if(h_tables == NULL) {
            printf("Memory allocation error 1\n");
            exit(0);
        }
        /* allocate the spaces for the hashing tables */
```

```
        for(i=0; i < no_of_tables; ++i) {
                h_tables[i] = (HASH_TABLE) malloc(sizeof(struct hash_table));
                h_tables[i]->table_size = next_prime(table_sizes[i]);
                h_tables[i]->table_load_cnt = 0;
                h_tables[i]->table_legit_cnt = 0;
                printf("table size: %4d ", h_tables[i]->table_size);
                h_tables[i]->the_cells = (cell *) malloc(sizeof(cell)*h_tables[i]->table_size);
                if(h_tables[i]->the_cells == NULL) {
                        printf("Memory allocation error 2\n");
                        exit(0);
                }
                /* initialize the cell nodes in hashing tables */
                for(j=0; j < h_tables[i]->table_size; ++j) {
                        h_tables[i]->the_cells[j].info = empty;
                        h_tables[i]->the_cells[j].element = -1;
                        h_tables[i]->the_cells[j].list_ptr = (LIST) malloc (sizeof(struct node));
                        h_tables[i]->the_cells[j].list_ptr->next = NULL;
                        h_tables[i]->the_cells[j].p_flag = 0;
                }
        }
        printf("\n");
        return h_tables;
}


unsigned int    next_prime(unsigned int table_size)
{
        unsigned int    temp;

        temp = table_size/4;
        return (temp * 4 + 3);                          /* choose a prime number */
}


int     search(HASH_TABLE H, int key)
{
        int             i, current_pos;
        unsigned int    hashing(unsigned int, unsigned int);

        i = 0;
        current_pos = hashing(H->table_size, key);                      /* hash a position */
         while(((H->the_cells[current_pos].info != empty) && (H->the_cells[current_pos].element !=
key))
                                        || (H->the_cells[current_pos].info == deleted)) {
                current_pos += 2*(++i) - 1;                             /* collision handling */
                if(current_pos >= H->table_size)
                        current_pos -= H->table_size;
        }
        return current_pos;                                             /* return the position */
}


int     insert(HASH_TABLE *H, int level_no, int last_level_pos, int key_store, unsigned int *k)
{
        position        pos;
```

```
        float           temp, temp1, temp2;

        int             search_for_insert(HASH_TABLE, int, int);
        HASH_TABLE      rehashing(HASH_TABLE);
        unsigned int    hashing(unsigned int, unsigned int);
        void            list_insert(HASH_TABLE, int, int);

        temp1 = H[level_no]->table_load_cnt + 1;
        temp2 = H[level_no]->table_size;
        temp = temp1 / temp2;                                   /* calculate the load factor */
        if(temp >= 0.5) {                                       /* check the load factor */
                H[level_no] = rehashing(H[level_no]);           /* rehashing, if necessary */
                printf("hashing table %4d expansion, temp: %4.2f table size: %4d.\n", level_no, temp,
H[level_no]->table_size);
                sleep(1);                                       /* stay a little while for demo */
        }

        pos = hashing(H[level_no]->table_size, key_store);                      /* hashing */
        pos = search_for_insert(H[level_no], pos, key_store);                   /* checking */

        if(H[level_no]->the_cells[pos].info != legitimate) {
                /* ok to insert here */
                ++(H[level_no]->table_load_cnt);
                ++(H[level_no]->table_legit_cnt);
                H[level_no]->the_cells[pos].info = legitimate;
                H[level_no]->the_cells[pos].element = key_store;
                if(level_no != 0)                               /* insert the index to the last level */
                        list_insert(H[level_no-1], last_level_pos, k[level_no]);
        }
        return pos;
}


/* this function is just for the insertion */

int     search_for_insert(HASH_TABLE H, int pos, int key_store)
{
        int i = 0;

        while((H->the_cells[pos].info == legitimate) && (H->the_cells[pos].element != key_store)) {
                pos += 2*(++i) - 1;
                if(pos >= H->table_size)
                pos -= H->table_size;
        }
        return pos;
}


unsigned int hashing(unsigned int H_SIZE, unsigned int key)
{
        return (key%H_SIZE);
}


/* deletion function */
```

```
void    deletion(HASH_TABLE *H, int N, int *krec, unsigned int *k)
{
        int  cnt;
        LIST p, temp_cell;

        cnt = N - 1;

        H[cnt]->the_cells[krec[cnt]].info = deleted;                    /* change node status */
        --(H[cnt]->table_legit_cnt);                                    /* decrement the node count */
        p = H[cnt-1]->the_cells[krec[cnt-1]].list_ptr;
        while((p->next != NULL) && (p->next->element != k[cnt]))
        p = p->next;                                    /* search for deleted node in linked list of parent */

        if(p->next == NULL)
                printf("system deletion error!\n");                     /* this should not occur */
        else {                                                          /* begin to delete */
                temp_cell = p->next;
                p->next = temp_cell->next;
                free(temp_cell);
        }

        cnt = cnt - 1;
        while(cnt > 0) {                                /* check if parent node should be deleted of not */
                p = H[cnt]->the_cells[krec[cnt]].list_ptr;
                if(p != NULL )                          /* if linked list of parent not empty, keep parent */
                        return;
                else {                                  /* if lisnked list of parent empty, delete parent */
                        H[cnt]->the_cells[krec[cnt]].info = deleted;
                        if(cnt > 0) {                                   /* check which level it is */
                                p = H[cnt-1]->the_cells[krec[cnt-1]].list_ptr;
                                while((p->next != NULL) && (p->next->element != krec[cnt]))
                                p = p->next;
                                if(p->next == NULL)
                                        printf("system deletion error!\n");     /* should not occur */
                                else {                                          /* begin to delete */
                                        temp_cell = p->next;
                                        p->next = temp_cell->next;
                                        free(temp_cell);
                                }
                        }
                }
                --cnt;                                                  /* decrement the level number */
        }
}
```

```
/* this function is used to insert node in the parent linked list */

void    list_insert(HASH_TABLE H, int last_level_pos, int in_value)
{
        LIST    list_header;
        LIST    temp_cell;

        /* get the linked list attached to the parent */
        list_header = H->the_cells[last_level_pos].list_ptr;
        temp_cell = (LIST) malloc(sizeof(struct node));                 /* allocate space */
        if(temp_cell == NULL) {
                printf("Memory allocation error \n");
                exit(0);
        }
        temp_cell->element = in_value;                                  /* assign the index value */
        temp_cell->next = NULL;

        temp_cell->next = list_header->next;                            /* insert to the top of list */
        list_header->next = temp_cell;
}


/* free the space of the hashing tables */

void    free_hash_tables(HASH_TABLE *H, int dim)
{
        int     i, j;
        LIST    p, temp;
        cell    *p_cell;

        for(i=0; i < dim; ++i) {                                /* first loop for free table pointers */
                for(j=0; j < H[i]->table_size; ++j) {                   /* free cells in one table */
                        p = H[i]->the_cells[j].list_ptr->next;
                        H[i]->the_cells[j].list_ptr->next = NULL;
                        while(p != NULL) {                      /* free the pointers in the linked list */
                                temp = p->next;
                                free(p);
                                p = temp;
                        }

                        p = H[i]->the_cells[j].list_ptr;
                        free(p);
                }
                p_cell = H[i]->the_cells;
                free(p_cell);                           /* free the pointer pointer to the whole tables */
        }
        free(H);
}


/* this program is the original r_spin function */

#include "spin.h"
```

```
unsigned long int *r_spin(long int *index_ptr, unsigned int N, unsigned int L, unsigned int *max, unsigned
int *catmax, unsigned int *k, char mode, int control)
{
        register int                y,u;
        static unsigned long int    *kr,*krec,*rect,*krcat,*krecat,*recat;
        static unsigned int                 *ex,*excat;
        static int                  ncontrol;
        unsigned long int           temp=0,kstore,hold, temp_sub;
        long int                    *temp_str;
        char                        e;

        /* CHECK CONTROL NUMBER */
        if(control != ncontrol || control == 0) {                    /* shut down access to */
                /* this block after the first access */
                /* DISPLAY COPYRIGHT */
                if(control == 0) {
                        puts("COPYRIGHT (c) 1988");
                        puts("by Ty K Coburn");
                        puts("All Rights Reserved");
                        exit(0);
                }
                /* FREE POINTERS FOR NEW ARRAY */
                if(ncontrol != 0) {
                        free(kr);
                        free(krec);
                        free(rect);
                        free(krcat);
                        free(krecat);
                        free(recat);
                        free(ex);
                        free(excat);
                }
                /* SET CONTROL */
                ncontrol = control;
                /* ALLOCATE MEMORY TO POINTERS */
                rect = (unsigned long int *)calloc(N,sizeof(long int));
                if(rect == NULL) {
                        puts("allocation error in function r_spin 1\n");
                        exit(0);
                }
                kr = (unsigned long int *)calloc(N-1,sizeof(long int));
                if(kr == NULL) {
                        puts("allocation error in function r_spin 2 \n");
                        exit(0);
                }
                ex = (unsigned *) calloc(N-1,sizeof(int));
                if(ex == NULL) {
                        puts("allocation error in function r_spin 3\n");
                        exit(0);
                }
                krec = (unsigned long int *) calloc(N-1,sizeof(long int));
                if(krec == NULL) {
                        puts("allocation error in function r_spin 4\n");
                        exit(0);
                }
```

```
/* COMPUTER PARAMETERS */
for(y=1;y<N;y++) {
        if(*(max+y) > 0 && *(max+y) <= 10) {
        *(ex+y-1) = 1;
        *(kr+y-1) = (unsigned long int)9-*(max+y)+1;
        } else if(*(max+y) > 10 && *(max+y) <= 100) {
        *(ex+y-1) = 2;
        *(kr+y-1) = (unsigned long int)99-*(max+y)+1;
        } else if(*(max+y) > 100 && *(max+y) <= 1000) {
        *(ex+y-1) = 3;
        *(kr+y-1) = (unsigned long int)999-*(max+y)+1;
        } else if(*(max+y) > 1000 && *(max+y) <= 10000) {
        *(ex+y-1) = 4;
        *(kr+y-1) = (unsigned long int)9999-*(max+y)+1;
        } else {
        *(ex+y-1) = 5;
        *(kr+y-1) = (unsigned long int)99999L-*(max+y)+1;
        }
}
recat = (unsigned long int *)calloc(N,sizeof(long int));
if(recat == NULL) {
        puts("allocation error in function r_spin 5\n");
        exit(0);
}
krcat = (unsigned long int *)calloc(N-1,sizeof(long int));
if(krcat == NULL) {
        puts("allocation error in function r_spin 6\n");
        exit(0);
}
excat = (unsigned *) calloc(N-1,sizeof(int));
if(excat == NULL) {
        puts("allocation error in function r_spin 7 \n");
        exit(0);
}
krecat = (unsigned long int *) calloc(N-1,sizeof(long int));
if(krecat == NULL) {
        puts("allocation error in function r_spin 8\n");
        exit(0);
}
for(y=0;y<N-1;y++) {
        if(*(catmax+y) > 0 && *(catmax+y) <= 10) {
        *(excat+y) = 1;
        *(krcat+y) = (unsigned long int)9-*(catmax+y)+1;
        } else if(*(catmax+y) > 10 && *(catmax+y) <= 100) {
        *(excat+y) = 2;
        *(krcat+y) = (unsigned long int)99-*(catmax+y)+1;
        } else if(*(catmax+y) > 100 && *(catmax+y) <= 1000) {
        *(excat+y) = 3;
        *(kr+y) = (unsigned long int)999-*(catmax+y)+1;
        } else if(*(catmax+y) > 1000 && *(catmax+y) <= 10000) {
        *(excat+y) = 4;
        *(krcat+y) = (unsigned long int)9999-*(catmax+y)+1;
        } else {
        *(excat+y) = 5;
        *(krcat+y) = (unsigned long int)99999L-*(catmax+y)+1;
```

```
                    }
                    if( y == 0 ) {
                    *recat = 0;
                    *(recat+1) = *(max)**(catmax);
                    hold = *(recat+1);
                    } else {
                    *(recat+y+1) = hold**(catmax+y)+*(recat+y);
                    hold = hold**(catmax+y);
                    }
            }
            /* SET AND INITIALIZE INDEX FILE */
            if( mode == 's') {
                    printf("The size of your index file is %lu bytes\n(press any character to continue,
e to EXIT)\n",*(recat+N-1)*sizeof(long int));
                    e = getchar();
                    if(e == 'e')
                            exit(0);
            }
            if(mode == 'n' || mode == 's') {
                    for(u=0;u<recat[N-1];u++) {
                            *(index_ptr+u) = (long int) temp;
                    }
            }
    }

    /* COMPUTE RETURN VALUE */
    if(L < 1 || L > N-1) {
            puts("Illegal level value in function r_spin\n");
            krec[L-1] = (unsigned long) -1;
            return krec;
    }

    *rect = *k;

    if(mode=='n' || mode=='a' || mode=='s') {
        for(y=0;y<L;y++) {
            if(*(k+y) > *(max+y)-1 || *(k+y+1) > *(max+y+1)-1) {
                    puts("subscript out of range in function r_spin\n");
                    krec[L-1] = (unsigned long) -1;
                    return krec;
            }
            *(rect+y+1) = *(rect+y)*pow((double)10, (double)*(ex+y))+*(k+y+1)-*(rect+y)**(kr+y);
            kstore = ++*(rect+y+1);
            *(rect+y+1) = *(rect+y)*pow((double)10, \
            (double)*(excat+y))-*(rect+y)**(krcat+y);
            for(u=0;u<*(catmax+y);u++) {
                    *(rect+y+1) = *(rect+y)*pow((double)10, (double)*(excat+y))+u-
*(rect+y)**(krcat+y);
                    temp_sub = (unsigned long int)(*(rect+y+1)+*(recat+y));
                    temp = *(index_ptr+temp_sub);
                    if(temp == 0) {                         /* if empty, store here */
                            *(index_ptr+temp_sub) = kstore;
                            *(krec+y) = *(rect+y+1);
                            break;
                    } else if(temp == kstore) {                     /*if not empty*/
```

```
                        *(krec+y)=*(rect+y+1);
                        break;
                } else if(u==*(catmax+y)-1) { /* overflow */
                        *krec = (unsigned long int)y;
                        *(krec+L-1) = (unsigned long int)-1;
                        return krec;
                }
        }
    }
    return krec;
}

    printf("can we reach here?\n");
    for(y=0;y<L;y++) {
        if(*(k+y) > *(max+y)-1 || *(k+y+1) > *(max+y+1)-1) {
        puts("subscript out of range in function r_spin\n");
        krec[L-1] = (unsigned long) -1;
        return krec;
        }

        *(rect+y+1) = *(rect+y)*pow((double)10,(double)*(ex+y))+*(k+y+1)-*(rect+y)**(kr+y);
        kstore = ++*(rect+y+1);
        for(u=0;u<*(catmax+y);u++) {
            *(rect+y+1) = *(rect+y)*pow((double)10, (double)*(excat+y))+u-
*(rect+y)**(krcat+y);
            temp_sub = (unsigned long int)(*(rect+y+1)+*(recat+y));
            temp = *(index_ptr+temp_sub);
            if(temp == kstore) {
                *(krec+y)=*(rect+y+1);
                break;
            } else if(u == *(catmax+y)-1) {
                *krec = (unsigned long int)y;
                *(krec+L-1) = -1;
                return krec;
            }
        }
    }
    return krec;
}


/* input integer or positive integer,  this program will give error message to the users if users type wrong
keys */

#include "spin.h"

int     input_int()
{
    int     input_flag;
    int     int_val;
    char    dump_str[100];

    do {
        input_flag = 0;                                     /* set down the flag */
        if(scanf("%d", &int_val) == 0) {                    /* if error occurs */
```

```
                scanf("%S", dump_str);                  /* dump the wrong characters */
                input_flag = 1;                                 /* set up the flag */
                printf("Input error, try again: ");             /* error message */
            }
        } while(input_flag);
        return int_val;
}


int     input_positive_int(int low_limit)
{
        int     input_flag;
        int      int_val;
        char    dump_str[100];

        do {
                input_flag = 0;                                 /* set down the flag */
                if(scanf("%d", &int_val) == 0) {                /* if error occurs */
                        scanf("%S", dump_str);                  /* dunp the wrong characters */
                        input_flag = 1;                         /* set up the flag */
                        printf("Input error, try again: ");     /* error message */
                } else if(int_val < low_limit) {                /* if the input is lower */
                        input_flag = 1;                         /* set up the flag */
                        printf("Input error, try again: ");     /* error message */
                }
        } while(input_flag);

        return int_val;
}


/* this program includes some subroutines used in the h_spin and r_spin testing */

#include "spin.h"

/* function of permutation generation */
unsigned int    **sub_generator(int dim, unsigned int *ckmax)
{
        void            generator(unsigned int **, unsigned int *, int, int, unsigned  int *);
        unsigned int    *temp_str;                      /* temp string to transfor indexes */
        unsigned int    **sub;          /* pointor to the beginning of permutations in memory */
        int             i, j;
        int             total_sub = 1;                  /* total number of permutatios to create */

        temp_str = (unsigned int *) malloc(dim*sizeof(unsigned int));

        for(i=0; i < dim; ++i)
                total_sub = total_sub * ckmax[i];

                /* open space for pointer array of permutations */
                sub = (unsigned int **)malloc(total_sub*sizeof(unsigned int *));
                for(i=0; i < total_sub; ++i) {
                        /* open space for each permutation */
                        sub[i] = (unsigned int *)malloc(dim*sizeof(unsigned int));
                for(j=0; j < dim; ++j)
```

```
                sub[i][j] = 0;                          /* initialize each permutation */
        }

        /* generate permutations recursively */
        generator(sub, ckmax, 0, dim, temp_str);

        return sub;             /* return pointor to the beginning of permutations in memory */
}


/* recursive function for generating permutations */
void generator(unsigned int **sub, unsigned int *ckmax, int n, int N, unsigned int *temp_str)
{
        static int      cnt = 0;                        /* static countor of permutations */
        unsigned int    i, j;

        for(i=0; i < ckmax[n]; ++i) {
                if(n >= (N-1)) {                                        /* base case of recurrance */
                        for(j=0; j <= (n-1); ++j)       /* transfor higher indexes to permutation*/
                                sub[cnt][j] = temp_str[j];
                        sub[cnt][n] = I;                                /* assign sequential value */
                        ++cnt;                          /* increment the permutation countor */
                } else {                                                /* recurrance step */
                        temp_str[n] = I;                                /* assign sequential value */
                        generator(sub, ckmax, n+1, N, temp_str);        /* call recursivelly */
                }
        }
}


/* this function is used to generate randomly-ordered list for re-arranging the permutations tested */

int list_generator(int *per_list, int total_sub, int EXP_NO)
{
        int     sub_no, temp_len, list_len = 0;
        int     i, j;

        for(i=0; i < EXP_NO; ++I)                               /* initialize the array list */
                per_list[i] = -1;

        sub_no = rand()%total_sub;                              /* generate random numbers */
        per_list[0] = sub_no;                   /* assign the random number to the list */
        ++list_len;

        for(i=1; i < EXP_NO; ++i) {
                sub_no = rand()%total_sub;                      /* generate random numbers */
                temp_len = list_len;

                for(j=0; j < temp_len; ++j) {           /*check if there is repeated number */
                        if(per_list[j] == sub_no)                       /* if repeated, abandon */
                                break;
                        else if((per_list[j] != sub_no) && (j == list_len-1)) {
                                per_list[j+1] = sub_no;         /* if not repeated, add to the list */
                                ++list_len;
                        }
```

```
            }
        }
        return list_len;                    /* return the randomly-ordered integer list */
}
```

Appendix B

```
/* This program tests the insertion and search operations of H_SPIN and R_SPIN */

#include "spin.h"

typedef struct {                                                    /* time structure */
        clock_t  begin_clock, save_clock;
        time_t   begin_time, save_time;
} time_keeper;

static time_keeper      tk;                     /* instance of timer, known only to this file */

void    main(void)
{
        unsigned int    *t_size, *max, *rmax, *cmax, *ckmax;
        unsigned int    **sub_permu;
        int             dim, EXP_NO=10, sz, total_sub=1;
        int             i, j, k, l, sub_no, sub, list_len = 0;
        int             ser_len, total_index, temp_hold;
        char            n='n';
        double          prn_time1, prn_time2;
        int             *list, *ser_list, *temp, *temp1;
        HASH_TABLE      *hash_tables;
        long int        *index_pointer;
        FILE            *ofp, *ofp1, *ofp2, *ofp3;
        char            fname[40], fname1[40], fname2[40], fname3[40];

        unsigned int    **sub_generator(int, unsigned int *);
        void            start_time(void);
        double          prn_time(void);
        char*           name_gen(int);
        int             list_generator(int *, int, int);
        HASH_TABLE*     initialize_hash_tables(unsigned int *, unsigned int);
        int*            h_spin(HASH_TABLE *, unsigned int, unsigned int,  unsigned int *, unsigned
int *, int, char);
        void            free_hash_tables(HASH_TABLE *, int);

        printf("Type in file name for insertion testing output of H_SPIN: ");
        scanf("%s", fname);
        printf("Type in file name for searching testing output of H_SPIN: ");
        scanf("%s", fname1);
        printf("Type in file name for insertion testing output of R_SPIN: ");
        scanf("%s", fname2);
        printf("Type in file name for searching testing output of R_SPIN: ");
        scanf("%s", fname3);

        ofp = fopen(fname, "w");
        ofp1 = fopen(fname1, "w");
        ofp2 = fopen(fname2, "w");
        ofp3 = fopen(fname3, "w");
```

```
printf("Please input the number of dimensions: ");
dim = input_positive_int(2);
printf("Please input size of dimension: ");
sz = input_positive_int(2);

max = (unsigned int *) malloc(dim*sizeof(unsigned int));
rmax = (unsigned int *) malloc(dim*sizeof(unsigned int));
cmax = (unsigned int *) malloc((dim-1)*sizeof(unsigned int));
ckmax = (unsigned int *) malloc(dim*sizeof(unsigned int));
t_size = (unsigned int *) malloc(dim * sizeof(unsigned int));

for(i=0; i < dim; ++I)                          /* input the max size of sparse data */
        max[i] = 97;                                    /* representation for each dim */
rmax[0] = sz;                                  /* assign the max sizes for r_spin function */
for(i=1; i < dim; ++i)
        rmax[i] = 97;
for(i=0; i < (dim-1); ++I)                      /* assign the actual sizes in r_spin */
        cmax[i] = sz;
for(i=0; i < dim; ++I)                  /*ckmax will be used in permutation generating*/
        ckmax[i] = sz;
t_size[0] = 2*sz;                               /* set the table sizes for each level */
for(i=1; i < dim; ++i)
        t_size[i] = t_size[i-1]*sz;
for(i=0; i < dim; ++I)                          /* give the total number of permutations */
        total_sub = total_sub * ckmax[i];

total_index = 0;                                /* calculate the total size of r_spin index */
for(i=0; i < dim-1; ++i) {
        if(i == 0) {
                total_index = ckmax[0]*ckmax[1];
                temp_hold = total_index;
        } else {
                total_index = temp_hold*ckmax[i+1] + total_index;
                temp_hold = temp_hold*ckmax[i+1];
        }
}
printf("Generating permutions\n");                              /* permutation generating */
sub_permu = (unsigned int **)sub_generator(dim, ckmax);
srand(time(NULL));

for(k=0; k < 30; ++k) {                                                 /* start testings */
        /* create the hashings and index for h_spina nd r_spin */
        hash_tables = initialize_hash_tables(t_size, dim);
        index_pointer = (long int*)calloc(total_index+1, sizeof(long int));

        list = (int *) malloc(EXP_NO*sizeof(int));
        if(list == NULL) {
                printf("list calloc failure, exit(0)\n");
                exit(0);
        }
        /* create randomly-orderd list for permutations tested */
        list_len = list_generator(list, total_sub, EXP_NO);
        /* initialize the h_spin and r_spin functions */
        sub = 0;
```

```
                h_spin(hash_tables, dim, dim-1, max, sub_permu[sub], 1, 'i');
                r_spin(index_pointer, dim, dim-1, rmax, cmax, sub_permu[sub], n, 1);

                printf("Beginning of the %dth empty test(for insertion)\n", k+1);
                start_time();                                   /* start testing time */
                for(j=0; j < list_len; ++j) {                   /* testing the overhead */
                        sub = list[j];
                }
                prn_time1 = prn_time();                         /* end the testing time */
                printf("Beginning of the %dth h_spin test(insertion)\n", k+1);
                start_time();                                   /* start the testing time */
                for(j=0; j < list_len; ++j) {                   /* this is the actual testing */
                        sub = list[j];
                        h_spin(hash_tables, dim, dim-1, max, sub_permu[sub], 1, 'p');
                }
                prn_time2 = prn_time();                         /* end the testing time */
                fprintf(ofp, "%6d %15.5lf %15.5lf %15.5lf\n", list_len, prn_time1, prn_time2,
(prn_time2-prn_time1)/list_len);

                printf("Beginning of the %dth empty test(for insertion)\n", k+1);
                start_time();                                   /* start the testing time */
                for(j=0; j < list_len; ++j) {                   /* testing the overhead */
                        sub = list[j];
                }
                prn_time1 = prn_time();                         /* end the testing time */
                printf("Beginning of the %dth r_spin test(insertion)\n", k+1);
                start_time();                                   /* start the testing time */
                for(j=0; j < list_len; ++j) {                   /* this is the actual testing */
                        sub = list[j];
                        r_spin(index_pointer,dim,dim-1,rmax,cmax,sub_permu[sub],n,1);
                }
                prn_time2 = prn_time();                         /* end the tesing time */
                fprintf(ofp2, "%6d %15.5lf %15.5lf %15.5lf\n", list_len, prn_time1, prn_time2,
(prn_time2-prn_time1)/list_len);
                /* recreate another randomly ordered list of permutations for search testing */
                ser_list = (int *) malloc(list_len*sizeof(int));
                if(ser_list == NULL) {
                        printf("Memory allocation error.\n");
                        exit(0);
                }
                ser_len = list_generator(ser_list, list_len, list_len);

                printf("\n\n\n");
                printf("Beginning of the %dth empty test(for searching)\n", k+1);
                start_time();                                   /* start the testing time */
                for(j=0; j < ser_len; ++j) {                    /* testing the overhead */
                        sub = list[ser_list[j]];
                }
                prn_time1 = prn_time();                         /* end the testing time */
                printf("Beginning of the %dth h_spin test(searching)\n", k+1);
                start_time();                                   /* start the testing time */
                for(j=0; j < ser_len; ++j) {                    /* this is the actual testing */
                        sub = list[ser_list[j]];
                        h_spin(hash_tables, dim, dim-1, max, sub_permu[sub], 1, 's');
                }
```

```
                prn_time2 = prn_time();                                 /* end the testing test */
                fprintf(ofp1, "%6d %15.5lf %15.5lf %15.5lf\n", ser_len, prn_time1, prn_time2,
(prn_time2-prn_time1)/ser_len);

                printf("Beginning of the %dth empty test(for searching)\n", k+1);
                start_time();                                           /* start the testing time */
                for(j=0; j < ser_len; ++j) {                            /* testing the overhead */
                        sub = list[ser_list[j]];
                }
                prn_time1 = prn_time();                                 /* end the testing time */
                printf("Beginning of the %dth r_spin test(searching)\n", k+1);
                start_time();                                           /* start the testing time */
                for(j=0; j < list_len; ++j) {                           /* this is the actual testing */
                        sub = list[j];
                        r_spin(index_pointer,dim,dim-1,rmax,cmax,sub_permu[sub],n,1);
                }
                prn_time2 = prn_time();                                 /* end the testing time */
                fprintf(ofp3, "%6d %15.5lf %15.5lf %15.5lf\n", ser_len, prn_time1, prn_time2,
(prn_time2-prn_time1)/ser_len);

                list_len = 0;                           /* reinitialize testing parameters for next test */
                ser_len = 0;
                free(list);
                free(ser_list);
                free(index_pointer);
                EXP_NO = EXP_NO + 100;
                printf("\n\n");
                free_hash_tables(hash_tables, dim);
        }

        h_spin(hash_tables, dim, dim-1, max, sub_permu[0], 1, 'f');             /* free the space */
        fclose(ofp);
        fclose(ofp1);
        fclose(ofp2);
        fclose(ofp3);
}


void    start_time(void)                                        /* get system time for starting */
{
        tk.save_clock = clock();
}

double  prn_time(void)                  /* get system time for ending and print durition time */
{
        double   clocks_per_second = (double) CLOCKS_PER_SEC;
        double    user_time;
        user_time = (clock() - tk.save_clock) / clocks_per_second;
        tk.save_clock = clock();
        return user_time;
}
```

Appendix C

```
/* This is a demonstration program of H_SPIN.  It demonstrates the insertion, the search, the rehashing, the
partial-match search and the deletion operations of H_SPIN */

#include "spin.h"

void    main(void)
{
        unsigned int    *t_size, *max, *ckmax, *se_print_out;
        unsigned int    **sub_permu;
        int             dim, sz, EXP_NO;
        int             total_sub = 1, i, j, l, sub_no, sub, list_len = 0;
        int             ser_len, total_index, temp_hold, run_flag=1;
        int             *list, *ser_list, *temp, *temp1, run_signal;
        HASH_TABLE      *hash_tables;
        int             *print_array, *permu;
        RETURN_LIST     rtn_list, rtn_dump;

        unsigned int    **sub_generator(int, unsigned int *);
        int             list_generator(int *, int, int);

        HASH_TABLE* initialize_hash_tables(unsigned int *, unsigned int);
        int*            h_spin(HASH_TABLE *, unsigned int, unsigned int, unsigned int *, unsigned int
*, int, char);
        void            free_hash_tables(HASH_TABLE *, int);
        int             h_spin_par_search(HASH_TABLE *, RETURN_LIST, unsigned int, unsigned
int, unsigned int *, int *, int);
        int             input_int();
        int             input_positive_int(int);
        int             pop_menu();
        /* input the parameters */
        printf("Please input the number of dimensions ( >= 2): ");
        dim = input_positive_int(2);
        printf("Please input size of dimension ( >= 2): ");
        sz = input_positive_int(2);
        printf("Please input the total number of permutations inserted: ");
        EXP_NO = input_positive_int(0);

        max = (unsigned int *) malloc(dim*sizeof(unsigned int));
        ckmax = (unsigned int *) malloc(dim*sizeof(unsigned int));
        t_size = (unsigned int *) malloc(dim * sizeof(unsigned int));

        for(i=0; i < dim; ++I)                          /* input the max spin size for each dim */
                max[i] = 97;
        for(i=0; i < dim; ++i) {                        /* set the table sizes for each dim */
                printf("Please input the hashing table size for each level %d:", i);
                t_size[i] = input_positive_int(0);
        }
        for(i=0; i < dim; ++I)                  /*ckmax will be used in permutati(n generating*/
                ckmax[i] = sz;
```

```
for(i=0; i < dim; ++I)                          /* give the total number of testing loop */
        total_sub = total_sub * ckmax[i];

printf("Generating permutions\n");                      /* permutation generating */
sub_permu = (unsigned int **)sub_generator(dim, ckmax);
print_array = (int *) malloc(sizeof(int)*dim);

srand(time(NULL));
/* create the hashing tables for each level */
hash_tables = initialize_hash_tables(t_size, dim);
/* create randomly-orderd list for permutations stored in h_spin */
list = (int *) malloc(EXP_NO*sizeof(int));
if(list == NULL) {
        printf("list calloc failure, exit(0)\n");
        exit(0);
}
list_len = list_generator(list, total_sub, EXP_NO);

/* initialize the h_spin() */
sub = 0;
h_spin(hash_tables, dim, dim-1, max, sub_permu[sub], 1, 'i');
/* insert the randomly-ordered permutations into h_spin */
printf("Begin insertions using randomly-ordered permutations.\n");
for(j=0; j < list_len; ++j) {
        sub = list[j];
        print_array = h_spin(hash_tables,dim,dim-1,max,sub_permu[sub],1,'p');
}

permu = (int*) malloc(sizeof(int)*dim);
/* begin the demostration */
while(run_flag) {
        run_signal = pop_menu();
        switch(run_signal) {
                case 1:  for(i=0; i < dim; ++i) {                       /* for insertion */
                                printf("Input the index on level %d:", i);
                                permu[i] = input_positive_int(0);
                         }
                         h_spin(hash_tables,dim,dim-1,max, (unsigned int*)permu,1,'p');
                         printf("\n");
                         break;
                case 2:  for(i=0; i < dim; ++I)             /* for list all permutations */
                                permu[i] = -1;  /* assign -1 for partial-match search */
                         rtn_list = (RETURN_LIST) malloc(sizeof(struct return_node));
                         rtn_list->next = NULL;
                         h_spin_par_search(hash_tables,rtn_list,dim, dim-1,max,permu,1);
                         if(rtn_list->next == NULL)
                                printf("\nNo matching found\n");
                         else
                                printf("\nThe permutations found: ");
                         printf("       ");
                         printf("The positions on each level: \n");
                         while(rtn_list->next != NULL) {
                                for(i=0; i < dim; ++i)
                                        printf("%6d", rtn_list->next->permu[i]);
                                printf("               ");
```

```
                        for(i=0; i < dim; ++i)
                                printf("%6d", rtn_list->next->permu_ptr[i]);
                        printf("\n");
                        rtn_dump = rtn_list;
                        rtn_list = rtn_list->next;
                        free(rtn_dump);
                }
                printf("\n");
                break;
        case 3: for(i=0; i < dim; ++i) {                                /* for search */
                        printf("Input the index on level %d (for partial-match
searching, input -1):", i);
                        permu[i] = input_int();
                }
                rtn_list = (RETURN_LIST) \
                malloc(sizeof(struct return_node));
                rtn_list->next = NULL;
                h_spin_par_search(hash_tables,rtn_list,dim, dim-1,max,permu,1);
                if(rtn_list->next == NULL)
                        printf("\nNo matching found\n");
                printf("\nThe permutations found: ");
                printf("      ");
                printf("The positions on each level: \n");
                while(rtn_list->next != NULL) {
                        for(i=0; i < dim; ++i)
                                printf("%6d", rtn_list->next->permu[i]);
                        printf("              ");
                        for(i=0; i < dim; ++i)
                                printf("%6d", rtn_list->next->permu_ptr[i]);
                        printf("\n");
                        rtn_dump = rtn_list;
                        rtn_list = rtn_list->next;
                        free(rtn_dump);
                }
                printf("\n");
                break;
        case 4: for(i=0; i < dim; ++i) {                                /* for deletion */
                        printf("Input the index on level %d:", i);
                        permu[i] = input_positive_int(0);
                }
                print_array = h_spin(hash_tables,dim,dim-1,max, (unsigned int
*)permu, 1,'d');
                if(print_array[0] == -1)
                printf("\nDeletion fails!\n");
                printf("\n");
                break;
        case 5: run_flag = 0;                                           /* for exit */
                break;
        }
    }
    /* free space */
    list_len = 0;
    free(list);
    printf("\n\n");
    free_hash_tables(hash_tables, dim);
```

```
        h_spin(hash_tables, dim, dim-1, max, sub_permu[0], 1, 'f');
}


int     pop_menu()
{
        int     rtn_val;

        int     input_positive_int(int);

        printf("\n*****************************\n");
        printf("  Insetion ..... 1\n");
        printf("  List all ..... 2\n");
        printf("  Search ..... 3\n");
        printf("  Deletion .....4\n");
        printf("  Exit .....5\n");
        printf("*****************************\n");
        printf("Choose: ");
        rtn_val = input_positive_int(1);
        return rtn_val;
}
```

VITA

Yunpeng Zhang

Candidate for the Degree of

Master of Science

Thesis: AN EXPERIMENTAL ANALYSIS OF A NEW MULTIDIMENSIONAL STORAGE AND RETRIEVAL METHOD

Major Field: Computer Science

Biographical:

Personal Data: Born in Jilin City, China, on July 23, 1964, the son of Jie Zhang and Shuhua Zhu.

Education: Graduated from Jilin TieLu High School, Jilin City, China in July 1983; received Bachelor of Science degree in Metal Material Engineering from Shanghai JiaoTong University, Shanghai, China and a Master of Science degree in Mechanical and Aerospace Engineering from Oklahoma State University, Stillwater, Oklahoma in July 1987 and December 1994, respectively. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December, 1996.

Experience: Employed as a mechanical engineer by Shenyang Electronic/Mechanical Research/Design Institute from August, 1987 to December 1989; employed as a mechanical engineer by China National Metals/Minerals Corporation from December 1989 to December 1992; employed by Oklahoma State University, Department of Mechanical and Aerospace Engineering as a graduate research assistant

from May 1993 to December 1994; employed by Oklahoma State University, Department of Computer Science as a graduate research assistant from June 1995 to May 1996; employed as a software engineer by Measurex Corporation (California) from July 1996 to present.