IMPLEMENTATION OF A FAMILY OF 2-3-4 TREES IN THE HYPERCUBE

By

ABDULKADER A. AL-FANTOOKH

Bachelor of Science in Computer Science

King Saud University

Riyadh, Saudi Arabia

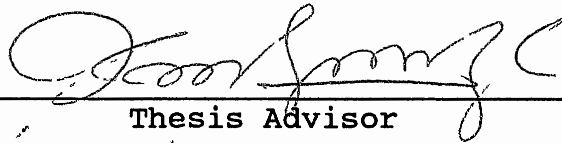
1988

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
May, 1992

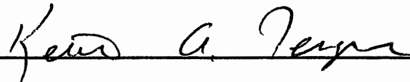
IMPLEMENTATION OF A FAMILY OF 2-3-4 TREES

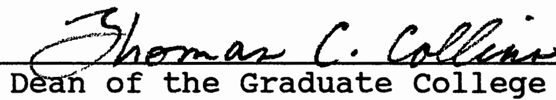
IN THE HYPERCUBE

Thesis Approved:



Thesis Advisor





Dean of the Graduate College

ACKNOWLEDGEMENT

I would like to express my utmost thanks to Dr. K. M. George, my advisor. I appreciate not only his time spent on my thesis but also his encouragement and advice throughout my graduate study. I would also like to extend my special thanks to Dr. D. D. Fisher for his invaluable suggestions and comments. A special thanks and appreciations go to Dr. K. A. Teague for his suggestions and facilitating the needed hardware for me.

I thank M. Abdulkareem, my friend, for his valuable discussions. Last but not least, I would like to thank my parents for their love and support.

TABLE OF CONTENTS

Chapter | Page

I. INTRODUCTION 1

II. MAPPING THE 2-3-4 TREE TO THE HYPERCUBE............. 4

External Binary Search Tree 4

Ephemeral 2-3-4 Trees 5

Previous Work 5
A Mapping of a 2-3-4 Tree to the Hypercube Architecture........................ 8
Communication Among Processors 11

III. OPERATIONS OF THE DISTRIBUTED 2-3-4 TREE 13

Bottom-Up Updating 13
Top-Down Updating 13
The Tree Manipulation Operations 14
Definitions 15
Search 15
Insertion 16
Deletion 19
Pipelining of the Operations 22
Performance Analysis 22

IV. FINGER DISTRIBUTED 2-3-4 TREE 27

Definition 27
Ephemeral Finger Search Tree 28
Methods of Implementing Fingers in a Single Processor 28
Heterogeneous Finger Search Tree 29
Homogeneous Finger Search Tree 29
Level-Linked FINGER 2-3 Tree 30
Red-Black Finger Tree 31
Methods of Implementing Fingers in Distributed Memory System.................... 32
Two-Fingers Implementation 33
Multi-Finger Implementation 39
Safety and Integrity 44
Finger Initialization 45
Finger Updating 45
Performance Analysis 46

Chapter Page

V. PERSISTENT TREE STRUCTURES 53

Methods of Making a Tree Structure Persistent 53
Copying the Entire Tree 53
Path Copying 54
Time Stamp 54
Limited Node Copying 55
Persistent Distributed 2-3-4 Tree Implementation 56
The Distributed 2-3-4 Tree Persistent Operations 56
Search 58
Insertion 59
Deletion 62
Persistent Multi-Finger 2-3-4 Tree 64
Performance Analysis 67

VI. CONCLUSION 69

REFERENCES 71

APPENDIXES

APPENDIX A - HOST AND NODE PROGRAM GENERAL ALGORITHMS 75

APPENDIX B - CAREY AND THOMPSON ALGORITHM 79

APPENDIX C - SOURCE CODE FOR THE HOST PROGRAM 83

APPENDIX D - SOURCE CODE OF THE NODES' PROGRAM ...101

LIST OF TABLES

Table Page

1. Theoretical and Empirical times for Three Cases of Insertion Operation Using 4 Processors. 25

2. The Average Elapsed Time For Search Through Fingers Versus From the Root with P = 32 48

3. The Average Elapsed Time For Search Through Fingers Versus From the Root with P = 16 48

4. The Average Elapsed Time For Search Through Fingers Versus From the Root with P = 8 48

5. The Average Elapsed Time For Insertion through Fingers Versus From the Root with P = 32 49

6. The Average Elapsed Time For Search Through Fingers Versus From the Root with P = 16 49

7. The Average Elapsed Time For Insertion Through Fingers Versus From the Root with P = 8 50

8. The Average Elapsed Time For Deletion Through Fingers Versus From the Root with P = 32 50

9. The Average Elapsed Time For Deletion Through Fingers Versus From the Root with P = 16 51

10. The Average Elapsed Time For Deletion Through Fingers Versus From the Root with P = 8 51

LIST OF FIGURES

Figure | Page

1. Level to Processor Mapping of the 2-3-4 Tree to the Hypercube Architecture..................... 9
2. Level to Processor Wrap Around Mapping of the 2-3-4 Tree to the Hypercube Architecture......... 10
3. Three Dimensional Hypercube.......................... 12
4. Insertion Transformation............................ 18
5. Deletion Transformation I........................... 20
6. Deletion Transformation II.......................... 21
7. Theoretical Result Against Empirical Result......... 25
8. Heterogeneous Finger Search Tree.................... 29
9. Homogeneous Finger Search Tree...................... 30
10. Level-Linked 2-3 Finger Tree........................ 31
11. Red-Black Finger Tree............................... 32
12. Two-Finger Distributed 2-3-4 Tree................... 35
13. Multi-Finger Distributed 2-3-4 Tree................. 40
14. Insertion Example Through Multi-Fingers............. 42
15. 2-3-4 Tree Update Operations........................ 58
16. Insertion Persistent Transformation I............... 61
17. Insertion Persistent Transformation II.............. 62
18. Deletion Persistent Transformation.................. 64
19. Persistent Multi-Finger 2-3-4 Tree.................. 68

CHAPTER I

INTRODUCTION

Currently there are several proposed and actual parallel computers with a number of different architectures. Traditionally, data structures and algorithms assume single processor machines that facilitate sequential programming. The availability of parallel computers makes it possible to implement divide and conquer approaches and reduces the time complexity of algorithms. Much work has already been done in parallelizing algorithms. But mapping of data structures to multiprocessor architectures, especially distributed processors, has only received limited attention. The problem needs to be addressed because more and more massively parallel distributed machines are being marketed and they do not have the problems of shared memory architectures such as memory contention and bottleneck.

The overall objective of this research is to explore techniques for mapping data structures and algorithms onto a parallel processing architecture. However, the scope of this work is limited to the implementation of various forms of 2-3-4 trees on the iPSC/2 hypercube architecture. In Chapter II, basic definitions of the external binary search tree and the 2-3-4 tree will be given. Then we will discuss different ways of mapping data structures to multiprocessor

systems. Our method of distributing the 2-3-4 tree nodes among the hypercube processors is also presented in Chapter II. In Chapter III, we discuss top-down updating and we explain the distributed 2-3-4 tree operations. Also, some required transformations to achieve top-down updating are described. Usually a tree structure is traversed starting from the root node. Intuitively, if we initiate more than one process to traverse different nodes of the tree in parallel, we can as a result obtain shorter traverse time. Such methods can be associated with finger search trees. Fingers serve as starting points for an operation. In Chapter IV, we present two approaches of making fingers in distributed 2-3-4 trees. The number of fingers used may vary. While the two fingers method is better than the conventional method of traversing in some applications, the multi-fingers method is the best in all cases. Theoretical and empirical results are presented to show the efficiency of the multi-finger approach.

Ordinary data structures are ephemeral which do not maintain the old structures at update operations. In some applications, we need to maintain multiple versions of the structure. Some examples of this kind of applications are text and file editing, and computational geometry [21]. In Chapter V, we discuss several methods of making a structure persistent. A method of making distributed 2-3-4 trees persistent along with the time and space efficiencies is also given. According to the hypercube terminologies, processors are called nodes, but because we are dealing with

tree structures, we will use throughout this thesis the word 'node' to refer to the tree node and 'processor' to refer to the hypercube node.

CHAPTER II

MAPPING 2-3-4 TREES TO HYPERCUBE ARCHITECTURE

There are several techniques to store large data structures in the main memory of a parallel computer, [3, 4, 6, 8, 17, 26]. This thesis is concerned with search trees and their mappings to distributed memory machines. The specific search tree in this thesis is a 2-3-4 tree. Before discussing the different mapping approaches, it is important to provide the definitions for the external binary search tree and the 2-3-4 tree.

External Binary Search Tree

An external binary search tree [16] is a tree structure that can be used to represent a subset of data items selected from a totally ordered set of data items. It is a binary tree containing the items of the set in its external nodes, one item per an external node. The items are arranged in ascending order from left to right in the tree. Each internal node contains a unique item, called a key which is used to discriminate the data items, such that all keys in the left subtree of the node are less than or equal to the key at the root; and all keys in the right subtree are

greater than the key at the root. An item in the tree can be accessed in time proportional to the depth of the tree by starting at the root and searching down along the search path. At an internal node, the key is compared with the key of desired item to decide whether to branch to the left or to the right. If the key of the search item is less than or equal to the key, a branch is taken to the left child; otherwise, the branch is to the right child. When an external node is reached, either this node contains the desired item or it is not in the set.

Ephemeral 2-3-4 Trees

Uniform height trees are those trees that guarantee a maximum length search path from the root node to any leaf node. All paths starting at an internal node and ending at an external node have the same length. A 2-3-4 tree is a uniform height external search tree where two, three, or four pointers and one, two, or three search keys, respectively, appear in internal (index) nodes, and all data items appear in external (leaf) nodes. A 2-3-4 tree is a special case of the B-tree. B-trees that have less constraints are called (a,b)-trees. Our 2-3-4 tree is an (a,b)-tree with a equal to 2 and b is equal to 4. Some researchers also call it a 2-4 tree.

Previous Work

In this section we will discuss several schemes available in the literature for maintaining a balanced

search tree in multiprocessor systems. Much of the work in parallel data structures has been targeted to shared memory multiprocessor systems. This can be attributed to their earlier development and availability. In shared memory architectures, the data structure is held in a global memory. Each processor can access that global memory in O(1) time.

Ellis[10] proposed an algorithm for allowing concurrent search and insertion in AVL search trees. A locking protocol has been designed to ensure consistency of the structure.

Dekel[7] implemented algorithms which can be applied to fixed size of K processors, K<N, where N is the number of nodes in the tree. His algorithms have complexity of O(N/K). To search for a specific element in an ordered set of N elements, $O(\log_2 K)$ time is needed to transmit the key being searched to the K processors, and the search can be conducted in $\log_k N$ parallel steps. After each step, the search location for the search step is transmitted among the processors. Each search step will require $O(\log_2 K)$ communication overhead. Thus the overall time complexity of a search is $(\log_2 K)*(\log_k N) = O(\log_2 N)$.

In this thesis, we will restrict our attention to the distributed memory systems, where the data structures are partitioned among the local memories of the processors. O'Gorman[17] presents a way of distributing the binary tree nodes among processors in an array. In his method each element is mapped onto a processing element. He deals with

array processors, so the position of the P.E. in the array, determines the position of the element in the tree. The children and the parent of each node can be known easily. The children of element N are given by 2N and 2N+1, and the parent is the integer N/2. In this mapping, changing the tree size is difficult because it requires a rearrangement of the keys, and the use of the processing elements is inefficient. O(N) processors is needed, where N is the number of the elements of the tree. This scheme can handle a large variety of search operations, including "partial match" queries.

Another scheme was proposed by Fisher[11]. He developed an architecture based on the Trie structure. In his design, the number of processors is proportional to the length of the maximum key.

Carey and Thompson[3] proposed a pipeline architecture and implement a 2-3-4 search tree of N nodes in a linear array of $\log_2 N + 1$ processing elements. Each processor holds a level of the tree structure in a local memory and the last processor, $P[\log_{N+1}]$, stores the actual data items. The scheme allows insertions, deletions, exact-match searches, and range queries. Each operation completes after $O(\log_2 N)$ delay and as many as $[\log_2 N + 1]/2$ operations may be at varying stages of execution. Their algorithm is given in Appendix B. In their design, when the number of tree elements becomes larger than N, the algorithm cannot be used.

In the next section, a method to map 2-3-4 trees to the

hypercube architecture is outlined.

A Mapping of a 2-3-4 Tree to the Hypercube Architecture

In our mapping of the 2-3-4 tree onto the hypercube, we have followed the work of Carey and Thompson[3] in their mapping of the tree to an array-processor. Every level of the tree is stored in one processor of the hypercube. A 2-3-4 tree of N nodes has at most $\log_2 N + 1$ levels, so we have to have at least $\log_2 N+1$ working processors to accommodate the N nodes. For example, the root of the tree is located in the first processor. If a node, say p, is located in processor i, then its children, if it has any, are located in processor i+1. The last level of the 2-3-4 tree will only consist of external nodes. Therefore, all external nodes are stored in one processor. Processor i contains internal nodes, when i is less than $\log_2 N+1$. Processor i contains external nodes, when i is equal to $\log_2 N+1$. The external nodes are fixed in the leaf processor, and the 2-3-4 tree grows up in case of split. The ith processor stores a maximum of 4^{i-1} nodes and a minimum of 2^{i-1} nodes in its local memory. Figure 1 represents a mapping of a 2-3-4 tree to the hypercube.

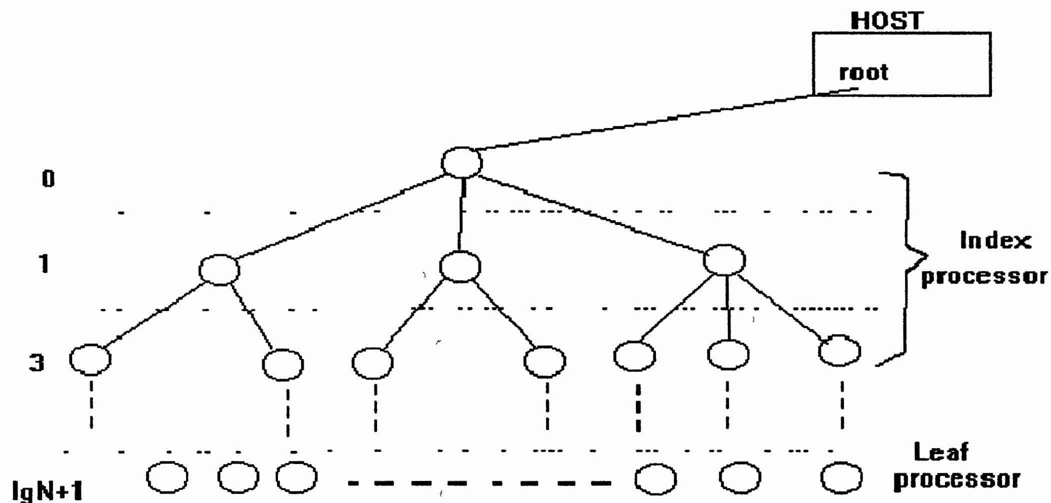
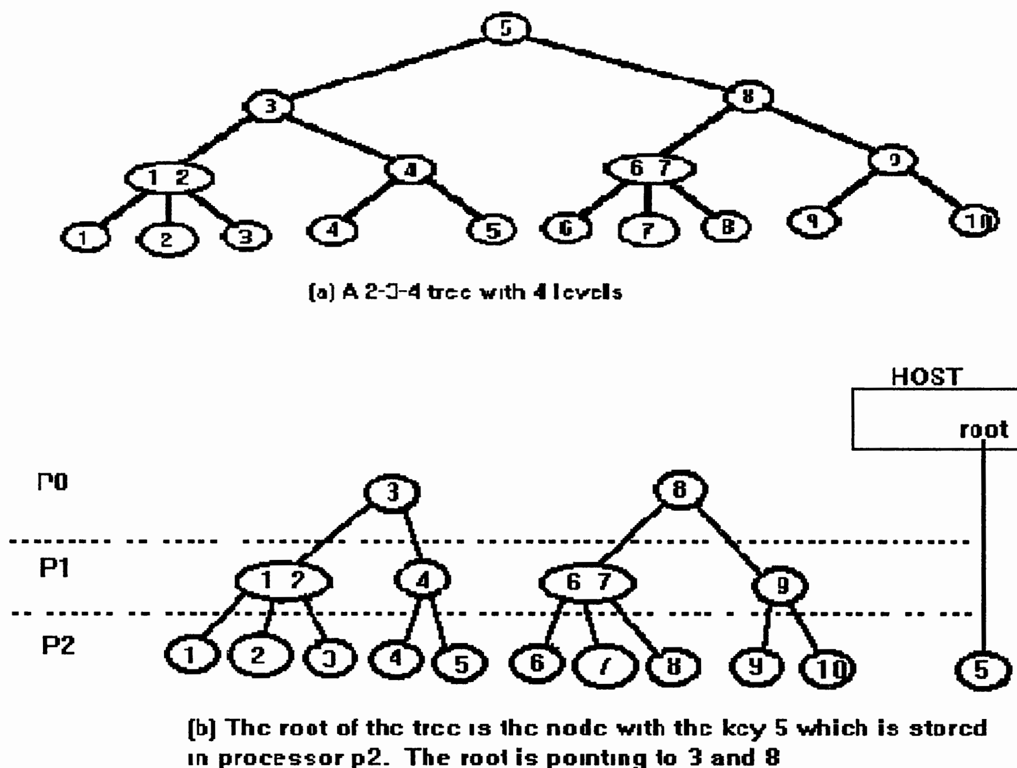


Figure 1. Level to processor mapping of the 2-3-4 tree to the hypercube architecture.

Although this mapping is considered an efficient one, it has a major drawback. When the number of the tree nodes becomes larger than N, where $\log_2 N+1$ is the number of processors, this mapping cannot work. When the current processor splits a node into two new nodes, their parent will be stored in the predecessor processor. But when the current processor is the root processor, the parent of the newly created node cannot be stored. So the algorithm fails. In order to overcome this problem, the mapping may wrap around the processors, i.e. when the root processor needs to split a node into two new nodes, the new root will be stored in the leaf processor. Therefore, the levels of the 2-3-4 tree will be wrapped around the processors. If a node is stored in the current processor (CURR-PROC), its

parent is stored in processor (CURR-PROC + P) MOD (P+1), and its children are stored in processor (CURR-PROC + 1) MOD (P+1), where P is the number of working processors.

Figure 2 illustrates the wrap around mapping of a 2-3-4 tree to the processors of the hypercube. In Figure 2, the 2-3-4 tree in (a) is mapped to the 3 hypercube processors in (b). After the split, the root node is stored in processors P2. In this method, the nodes of the tree are distributed symmetrically among the processors.



(a) A 2-3-4 tree with 4 levels

(b) The root of the tree is the node with the key 5 which is stored in processor p2. The root is pointing to 3 and 8

Figure 2. (a) The original 2-3-4 tree. (b) After wrap around mapping of the tree to 3 processors hypercube.

Communication Among Processors

Besides the processors' communication with the host, all other events of communication are only between neighbors. So processor i communicates only with processor i-1 and processor i+1. However, sending a message from processor i to processor i+1 might cause more than one hop because it is not guaranteed that processor i and processor i+1 are neighbors. If the mapping of the 2-3-4 tree levels to the hypercube processors in the way that level one of the tree is stored in processor one and level two is stored in processor two, etc, which results in a less than optimum assignment for communication among neighbors. The nodes of the hypercube are assigned unique addresses as shown in Figure 3. The addresses of any two nearest neighboring processors differ in one bit position. The communication channel number between two processors is determined by taking the exclusive-or of the two processors' addresses. In order to achieve maximum communication speed between processors, neighbors are selected according to the gray code sequence: 0, 1, 3, 2, 6, 7, 5, 4, etc. In this scheme, the communication between two levels takes only one hop.

Our algorithms for insertion and deletion operations are based upon the top-down node-splitting/merging scheme presented by Guibas and Sedgewick[12]. In this scheme, the rebalancing transformations are applied on the way down the tree during an update operation. Thus, when an insertion search encounters an external node, the key being inserted

can be attached right there, and the operation is complete.

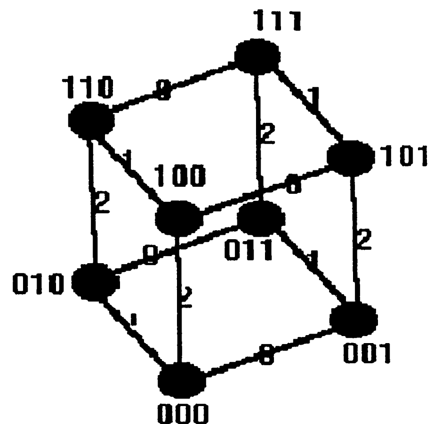


Figure 3. Three Dimensional Hypercube

This scheme of mapping a tree to the hypercube architecture is suitable for most tree types although it is presented here to the 2-3-4 tree. However, the balanced trees that involve rotation transformations are excluded because for making a rotation of the nodes, the nodes in the subtrees of the rotated nodes need to be transferred from one processor to another. For example, a single rotation of a red-black tree with n keys, costs O(n).

In the next Chapter, the different updating schemes for a 2-3-4 tree and the different tree manipulation operations are described.

CHAPTER III

OPERATIONS OF THE DISTRIBUTED 2-3-4 TREE

We call a 2-3-4 tree with nodes distributed among several processors, a distributed 2-3-4 tree. The definitions of the external binary search tree and the 2-3-4 tree are as given in the Chapter II.

Before discussing the tree operations, in detail, brief descriptions of the bottom-up and the top-down updating methods are provided.

Bottom-up Updating

In the 2-3-4 tree bottom-up updating, we start from the root and go down along the access path. When we reach the desired node or location, we perform the operation and go back up from that location to the root making the necessary transformations to maintain the structure.

Top-down Updating

The 2-3-4 tree top-down updating proceeds from the root down along the access path, maintaining the invariant that the current internal node has less than four children in the case of insertion and has more than two children in the case of deletion. The invariant is maintained by means of the transformations described in the following sections. When

we reach the desired node or location, the update operation is performed without propagating the transformations upward along the access path. When the operation is complete, the path need not be traversed back.

In this thesis, we follow the work of Guibas and Sedgewick[12] and the work of Tarjan[23] in their top-down updating. The top-down approach is more preferable than the bottom-up one because of the following advantages associated to the top-down approach:

- updating is done in a single pass;
- there is no need for parent pointers or a stack to store the access path nodes;
- concurrent operations are applicable and efficient because there is no need to lock the entire access path.

The Tree Manipulation Operations

In this section, we define the various distributed 2-3-4 tree manipulation operations as performed in the Hypercube. While each processor stores one level of the tree in its local memory, the host keeps the address of the root node of the tree, which is stored in the root processor. Although, the scheme is presented for 2-3-4 trees, it can be generalized for any 2^{p-2} - 2^p tree, where the integer p>= 3 [6]. When we increase p, the storage requirement will also increase; in contrast, the required number of processors will be less.

Definitions

Working processors are the processors of a multiprocessor architecture that have been selected to participate in the execution of the program.

The working processors are classified as follows:

- index processors which store internal index nodes;
- leaf predecessor processor which stores the lowest level of index nodes; and
- leaf processor which stores the data nodes.

There is one leaf processor and one leaf predecessor processor. All working processors are linearly ordered by level numbers.

Current Processor is the processor that receives a message of an operation.

Node Structure: Each node in the 2-3-4 tree has space for three keys and four pointers. The structure of a node follows.

<table>
<tr><td colspan="3">k1</td><td colspan="3">k2</td><td colspan="2">k3</td></tr>
<tr><td colspan="2">p1</td><td colspan="2">p2</td><td colspan="2">p3</td><td colspan="2">p4</td></tr>
</table>

k1, k2, and k3 are the three keys of the node, and p1, p2, p3, and p4 are the addresses of children of this node. The physical addresses of the children are in the successor processor.

Search

The search operation for the distributed 2-3-4 tree is

initiated by the host. The host sends a SEARCH(p, k) message to the root processor carrying p and k, the address of the root node and the key being searched respectively. When the root processor receives the SEARCH(p, k) message, it compares k with the keys in the index node whose address is given by p and then decides which pointer to follow. After selecting a pointer q, the current processor, if it is not the leaf processor, sends a SEARCH(q, k) message recursively to the successor processor carrying q and k.

When the leaf processor receives the SEARCH(p, k) message, it checks the pointer p. If it points to a data node containing the key k, then the desired key is found, but if the pointer p is NULL or points to a key not equal to k, the desired key is not found. As the last step, the leaf processor sends the result to the host directly.

Insertion

The insertion operation for the 2-3-4 tree is a parallel version of the top-down node-splitting insertion algorithm given by Guibas and Sedgewick[12]. The host sends an INSERT(p, k) message to the root processor carrying the address of the index node, p, and the key value, k, to be added. To simplify the presentation, p will be used to designate "the node pointed by p" hereafter. The root processor, or in general the current processor i which receives INSERT(p, k) message from its predecessor i-1, works according to one of the following three cases.

CASE I: Current Processor is an Index Processor:

The current processor i, which received the INSERT(p, k) message from its predecessor, compares the given k with the index keys in the node p and selects the appropriate pointer p' from the node p. The pointer p' provides the access to the next node on the access path. If k is less than or equal to the first key in the node, then processor i selects the first pointer as p' and so on. Before processor i sends INSERT(p', k) message to its successor, it sends INSERT-TRANSFORM(p',k) message to its successor to perform insertion transformation if applicable. When the successor processor of i receives this message, it performs the insertion transformation if it is full, having four pointers. The insertion transformation as shown in Figure 4 is to split the full node into two new nodes each having one key and two pointers . Actually, this transformation does not create two nodes besides the old one. Instead, it creates one more node, transfers two pointers to the new node with one key, and promotes the middle key. After performing the transformation if it is applicable, the successor processor sends a reply to the current processor with its old middle key. The current processor will update its current index node. The insertion transformation is applied to ensure that future node splitting will not propagate upwards in the direction of the root of the tree. Now, the current processor i uses the key k to select the

appropriate path as pointed to by p'. Then it sends INSERT(p', k) message with k and p' to the successor processor.

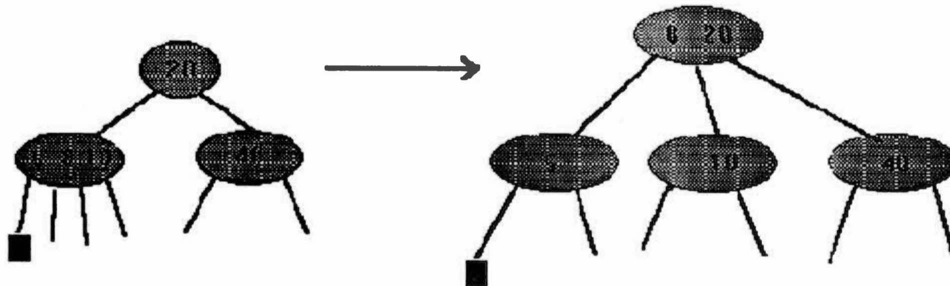


Figure 4. Insertion Transformation.
The node with four successors is split into two nodes each with two successors, and the middle key is promoted to the predecessor node.

CASE II: Current Processor is a Leaf-Predecessor Processor:

The processor compares the given key k with the index keys in its local memory and selects the appropriate pointer p'. Then, it sends an INSERT(p', k) message with p' and k to the leaf processor. If the key does not exist, the leaf processor inserts the key and sends a reply message to its predecessor containing the address of the newly inserted node p"; otherwise, it sends a null indicating that k is a duplicate key. The current processor inserts the incoming address p" into its current index node.

CASE III: Current Processor is
a Leaf Processor:

If the incoming key k does not exist in the given address p then a new data node p' is created and the new key k is inserted in node p'. The address of p' is sent to its predecessor, and a 'success' message is sent to the host. But if the coming key k is a duplicate then a null pointer is sent to its predecessor, and a 'fail' message is sent to the host.

Deletion

The deletion operation for the 2-3-4 tree is a parallel version of the top-down deletion algorithm of Guibas and Sedgewick[12]. When the host sends a DELETE(p, k) message to the root processor carrying the address of the index node, p, and the key value, k, to be deleted, the root processor or in general the current processor i which receives the DELETE(p, k) message from its predecessor i-1, works according to one the following three cases.

CASE I: Current Processor is
an Index Processor:

The current processor i compares the given key k with the index keys of the node p and selects the appropriate pointer p' on the access path. If k is less than or equal to the first key in the node, then processor i selects the first pointer as p' and so on. Before processor i sends a

DELETE(p', k) message to its successor, it sends a DELETE-TRANSFORM(s,p',p") message to its successor to perform a deletion transformation if applicable, where p" is the adjacent pointer for p', and s is the splitting key of p' and p". When the successor processor of processor i receives this message, it performs a deletion transformation if it has only two pointers. When the node has only two pointers and if its adjacent node has two pointers, they are merged as shown in Figure 5. The merging process is done by adding the information portion of the second node p" to the first one p', and then freeing the second node. But if its adjacent node p" has three or four pointers, then one of them is moved to p' as in Figure 6. After performing the transformation if it is applicable, the successor processor sends a reply to the current processor with the new information. The current processor updates its current index node. The deletion transformation is applied to ensure that future node merging will not propagate.

Now, the current processor i uses the key k to select the appropriate path p' as we explained above. Then it sends DELETE(p', k) message to the successor processor.

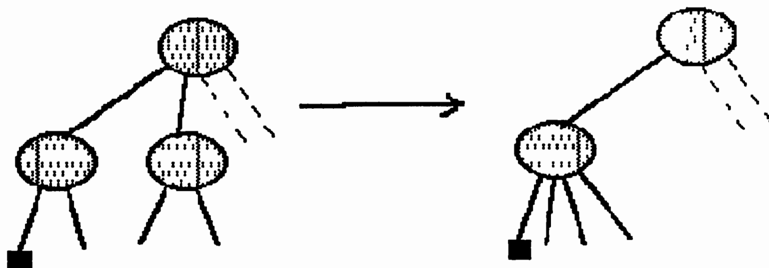


Figure 5. Deletion Transformation I

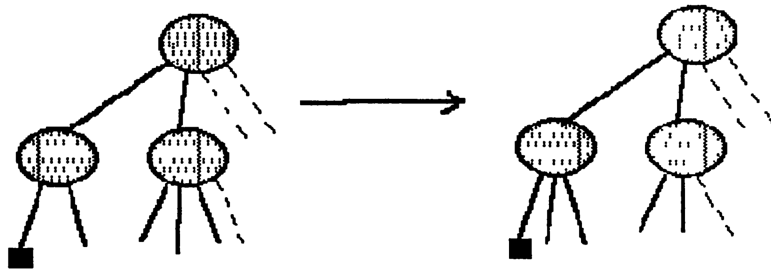


Figure 6. Deletion Transformation II

CASE II: Current Processor is a Leaf-Predecessor Processor:

The current node i compares the given key k with the index keys and selects the appropriate pointer p'. If k is less than or equal to the first key in the node, then processor i selects the first pointer as p' and so on. It sends a DELETE(p', k) message to the leaf processor. The leaf processor attempts to delete the key and send as a reply a message to its predecessor indicating the status of the operation (either fail or success.) The current processor deletes the pointer p' from its current index node if the delete operation succeeds.

CASE III: Current Processor is a Leaf Processor:

The leaf processor checks if the key k exists in the given address p. If the key exists, then node p is deleted and a 'no error' message is sent to its predecessor. Also, a 'success' message is sent to the host. If the k does not

exist or p is NULL, then an 'error' message is sent to its predecessor, and a 'fail' message is sent to the host.

Pipelining the Operations

The tree operation pipelining was proposed by Carey and Thompson[3]. Our implementation of the top-down updating allows operations' pipelining. In order to enable pipelined operations, the current processor blocks the incoming SEARCH, INSERT, and DELETE messages from its predecessor processor until it finishes the required transformation with its successor processor. This blocking is to ensure the correctness of the operations.

Performance Analysis

In the distributed 2-3-4 tree, the tree traversal cost mainly depends on how many levels need to be traversed. A tree of N elements, has as maximum $\log_2 N$ levels stored in $\log_2 N$ processors. A distributed 2-3-4 tree operations start from the root, so the search, insertion, and deletion operations cost $O(\log_2 N)$ time.

As we discussed in the previous section, our scheme of top-down updating of the distributed 2-3-4 tree supports pipelining the operations. The top-down updating is done in a single pass, so a processor finishes its task in a current operation when it passes the operation to its successor. The processors do not respond to another updating operation until they finish their communication with their successors and pass the operation to their successors. As a result,

every two processors are working exclusively in a single stage of an operation. Therefore, the number of pipelined operations is half of the number of working processors. We can achieve p/2 level of concurrence, where p is the number of processors. Thus, an operation on a distributed 2-3-4 tree completes after every 2 time units if we allow $O(\log_2 N)$ concurrence on all operations, where N is the number of keys in the tree.

The space required for each node of the distributed 2-3-4 tree is (3*sk + 4*sp) words of a processor storage, where sk is the size of a key and sp is the size of a pointer. As described in Chapter II, every level of the 2-3-4 tree is mapped to a processor. So that the root of the tree is mapped to the processor P_R which needs to store only one node of the tree, while processor P_{R+1} needs to store from 2 nodes to 4 nodes. In general processor P_{R+i} stores as maximum as 4^{i-1} nodes.

In order to estimate the complexity of the various operations, the elapsed time is used. The root processor keeps track of elapsed time for all operations. As an illustration of the performance of the mapping, theoretical and empirical times were computed and compared for three cases:

Case 1: Insertion before the smallest key.

Case 2: Insertion after the largest key.

Case 3: Insertion with one application of the insertion transformation.

The theoretical time is computed according to the published

performance of the 386 iPSC/2 hypercube processors release 3.3. 356 micro-seconds (μs) is the given time for sending a 64 bytes message with one hop latency, and 375 μs is the given time for sending a 100 bytes message with one hop latency. The message size in the cases listed above is 40 bytes. So we have to calculate the slope to get the sending time "t" for a message of 40 bytes.

$$\frac{375 - 356}{100 - 64} = \frac{375 - t}{100 - 40}$$

$$0.53 = \frac{375 - t}{60}$$

So, t = 343.2 μs.

For an update operation similar to the above three cases, three messages to propagate the operation, five messages to perform the required transactions, and two messages to calculate the time are needed when working in four hypercube processors. So the processors need to send a total of 10 messages. Therefore the total time for sending 10 messages is 3432 μs. The communication overhead is 3.43 ms. The time for computation is 1.2 ms, 2.4 ms, and 2.67 ms for case 1, case 2, and case 3 respectively. The timings are summarized in Table 1.

TABLE 1

THEORETICAL AND EMPIRICAL TIMES FOR THREE
CASES OF INSERTION OPERATION
USING 4 PROCESSORS

INSERT		
Insert	Theoretical msec	Empirical msec
Case 1	4.63	5
Case 2	5.83	6
Case 3	6.10	7

In Figure 7 a comparison between the two timings is presented.

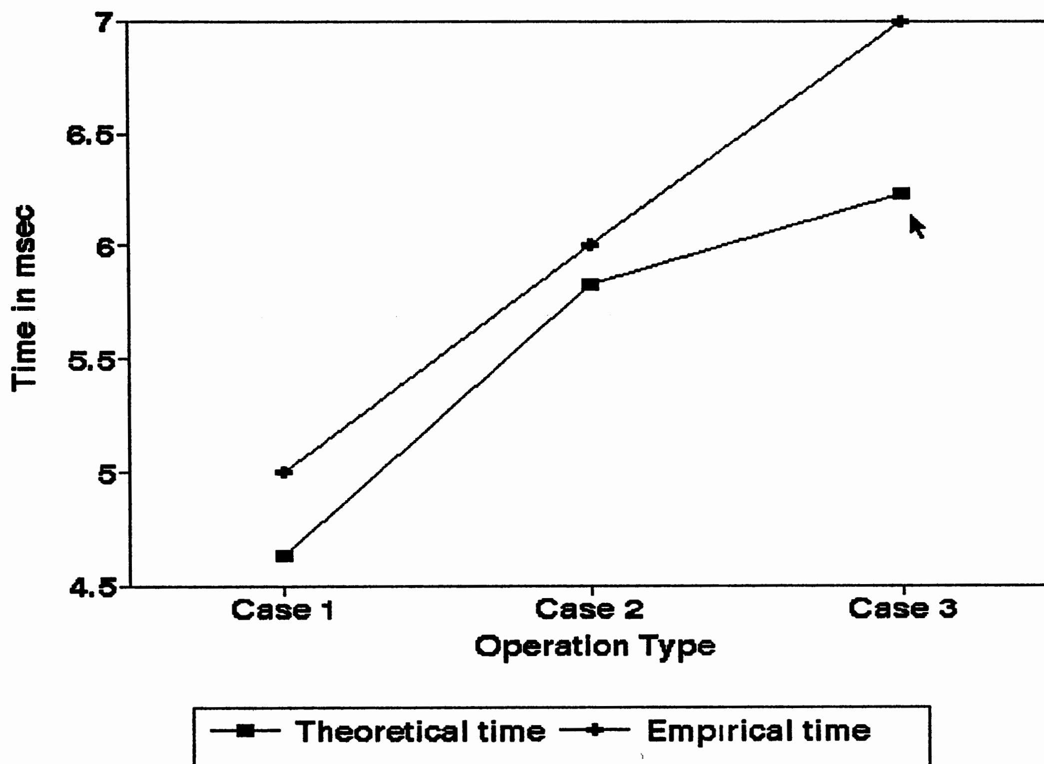


Figure 7. Theoretical result against Empirical result.

It can be noticed that there is a difference between the theoretical result and the empirical one. One of the reasons for that difference is that the operating system running on the processors is involved in many tasks, so extra overhead increases the empirical time. Another reason is that when the iPSC/2 Direct-Connect Module (DCM) channels receive more than one message for one processor, the messages is delayed. A third reason relates to the calculation of the theoretical time. Maybe the calculation of the theoretical time was not so efficient to correspond exactly to the empirical one.

The operations outlined in this Chapter assume that transactions always start at the root of the tree. As mentioned earlier, if transactions start at nodes other than the root, depending on the tree, transaction processing time can be reduced. In the next Chapter, one such method is addressed. The idea is to start transactions at nodes pointed by "fingers". This allows searches to proceed in parallel.

CHAPTER IV

FINGER DISTRIBUTED 2-3-4 TREES

Definition

We have seen in Chapter II how access in search trees usually begins at the point we called the root of the tree. This approach usually leads to efficient algorithms. But sometimes, due to the peculiarities of applications, we may want to start the search at external or leaf nodes. A "finger" into a tree is a pointer to an element of the tree. Fingers are usually used to indicate points of high activity in the tree and are used to minimize the cost due to such activity localized around the finger.

A finger search tree supports the following operations [20], among others:

1 - access(x,t,f) : find the node (if it exists) whose key value is x in tree t starting the search from the finger f;

2 - insert(x,t,f) : insert a node with key value x into tree t starting the search from the finger f;

3 - delete(x,t,f) : delete the node whose key value is x from the tree t starting the search from the finger t.

Ephemeral Finger Search Trees

An ephemeral finger search tree is a data structure that allows efficient execution of the three operations insertion, deletion, and access described in the previous section. The finger search trees discussed here contain only two fingers. As observed by Kosaraju[15], the number of fingers in the tree can be increased as needed. But two fingers are sufficient to minimize the time for the type of operations that commonly occur.

In some algorithms, an explicit reference to a finger could be used as in the above description of operations, but this usually is not done. Instead, the algorithm should be optimized to use the finger that performs better [20]. In binary search tree algorithms, it is usual to have the two fingers pointing to the minimum and maximum elements of the tree. The time complexity that one would like to achieve is $O(\log_2 d)$, where d is defined as the minimum linear distance between any finger and the desired node.

Methods of Implementing Fingers in a Single Processor

There are four methods available in the literature for implementing the finger search tree, [24,2,20]. They are presented in the following subsections.

Heterogeneous Finger Search Tree

This method is presented by Tarjan[24]. In an ordinary binary search tree, each node points to its two children. Tarjan converted such a tree into a heterogeneous finger search tree by making each node along the left path point to its parent instead of its left child, and each node along the right path point to its parent instead of its right child. Access to the tree is by two fingers pointing to the leftmost and rightmost external nodes. Tarjan obtained O(1 + log(min{d,n-d} + 1)) time complexity for an n-item heterogeneous search tree, and he obtained O(1 + log(min{d,n-d}+1)) amortized time for insertion or deletion of an item d positions from either end. Tarjan used a red_black tree as a basis for his work. The structure of the heterogeneous finger search tree is shown in Figure 8.

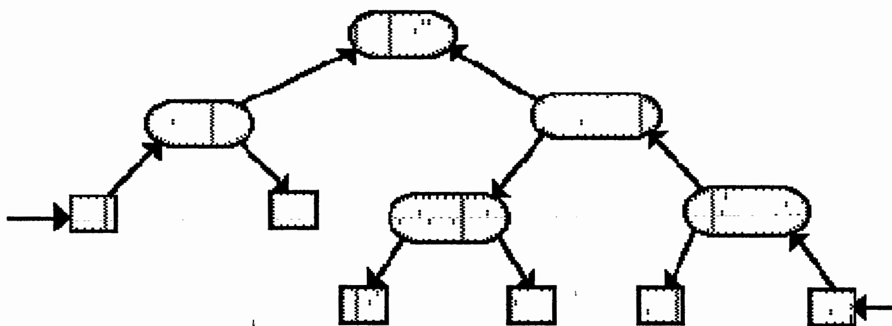


Figure 8. Heterogeneous Finger Search Tree

Homogeneous Finger Search Tree

Tarjan[24] also presented another method of

implementing a finger search tree. Each node in the tree points to its two children and to its parent. Each black node also points to its left and right neighbors. The structure of the homogeneous finger search tree is shown in Figure 9. The level links support searching for a given key starting from an arbitrary node in the tree. The performance of search was shown to be $O(1+\log(\min\{d,n-d\}+1)$ time, where d is the number of keys between the two given keys, and n is the number of the keys in the tree.

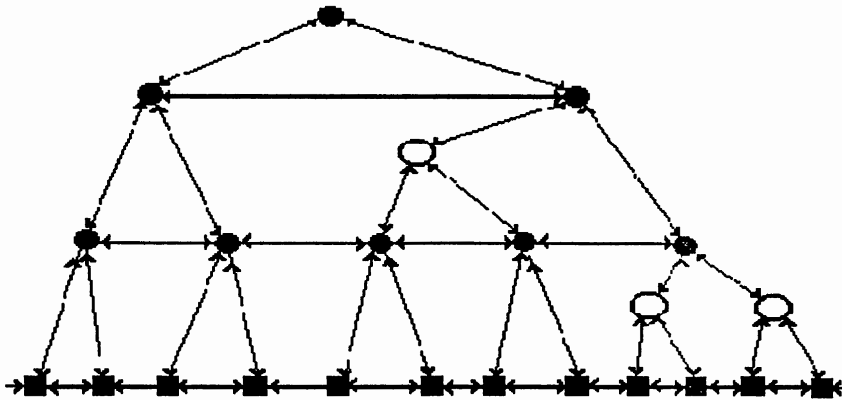


Figure 9. Homogeneous finger search tree

Level-Linked 2-3 Finger Tree

Brown [2] implemented a finger search tree using two fingers pointing to the internal terminal nodes of the tree. His tree's fingers do not point to external nodes because in his tree structure there is no upward link from the external nodes. He used a level-linked 2-3 tree in which the internal nodes are traversable upwards as well as downwards, and the internal nodes are linked horizontally. Brown

obtained an $O(\log_2 d)$ time to search for a key which is d keys away from a finger; and $O(1+s)$ steps for inserting a new external node, where s is the number of node splitting caused by the insertion. The structure of the level linked 2-3 finger tree is shown in Figure 10. The arrows point to the fingers.

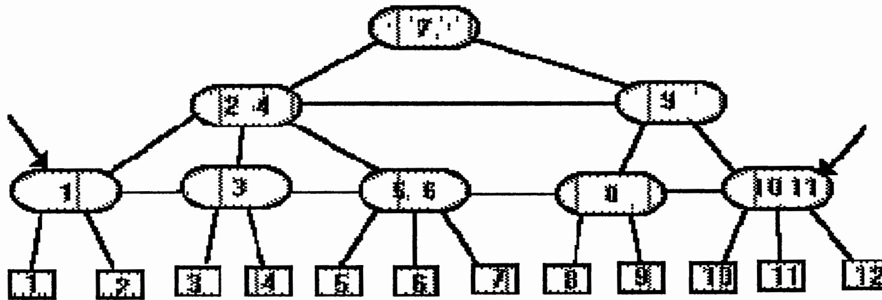


Figure 10. Level-Linked 2-3 Finger Tree

Red-Black Finger Tree

Sarnak [20] presented a complicated but efficient and interesting method. His method basically maintains a binary search tree with two fingers pointing to the extremes of the list order. The two ribs, which are the left most and right most access paths of the tree, do not have the same constraints imposed upon them as the rest of the tree. A balanced binary tree hangs from each node of these ribs, while the entire tree may not be considered balanced because of imbalances in the ribs. Each update to the tree starts at the bottom of a rib, and while moving up the rib, performs whatever rebalancing is necessary to ensure the logarithmic access time. The structure of the red-black

finger tree is shown in Figure 11.

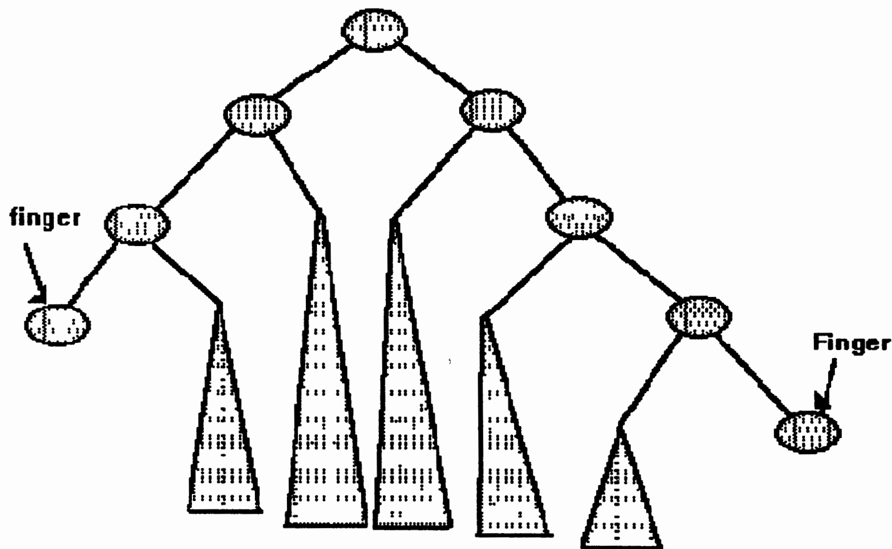


Figure 11. Red-Black Finger Tree

Methods of Implementing Fingers in a Distributed Memory System

To my knowledge, there is no work reported in the open literature about fingers on a multiprocessor environment. If we want to implement a finger search tree on a shared memory system, we do not have to make major changes to the finger implementations described in the previous section because the complete structure can be stored in a shared memory. However, to implement the finger search tree on a distributed memory system, new algorithms should be developed to suit the nature of the distributed memory system. In this thesis, we are concerned with the implementation of 2-3-4 finger trees in an iPSC/2 environment. Several implementation schemes are developed.

To facilitate the description of the new methods, we need the following definitions.

1. Tree ribs are the left most and right most paths of the tree starting from the root and ending at the external nodes.
2. A Subtending-node (Turning-node) is a node which is located on one of the ribs of the tree, and it should be a predecessor for the desired key.
3. Safe-nodes are the nodes that have less than four pointers in the case of insertion and except possibly for the root, more than two pointers in the case of deletion.
4. Busy-Processors are the processors that are currently searching for the desired key or location. They ignore any message carrying commands coming from their predecessors because they have already started the search themselves and they complete their searches before their predecessors. In the following sections, various implementation scheme are discussed.

Two Fingers Implementation

In our implementation of the distributed 2-3-4 tree, the host maintains the address of the root node of the tree as in Chapter II. To perform any operation, the host directs the requested operation to the root processor to start the operation.

In order to develop a distributed 2-3-4 finger tree, the distributed 2-3-4 tree implementation must be modified. Fingers are added and adapted to the distributed memory

system. In the new scheme, the host maintains a data structure which contains the two fingers and the processor's "id" which has the finger nodes. The structure of the host header is the following:

LEFT FINGER		ROOT		RIGHT FINGER	
ADR	PROC	ADR	PROC	ADR	PROC

We define our two fingers, left and right, to point to the smallest key and the largest key nodes in the leaf predecessor processor respectively. The left and right fingers always reside in the leaf-predecessor processor. This is one of the invariant of the implementation schemes, i.e. the rebalancing operations guarantee that the fingers reside in the leaf-predecessor processor. If we insert a key with a smaller value than the left finger, the left finger will be updated to point to the node storing the new inserted key. On the other hand, if we insert a key with a larger value than the right finger, then the right finger will be updated to point to the node storing the new key. Figure 12 shows a representation of a two-fingers distributed 2-3-4 tree.

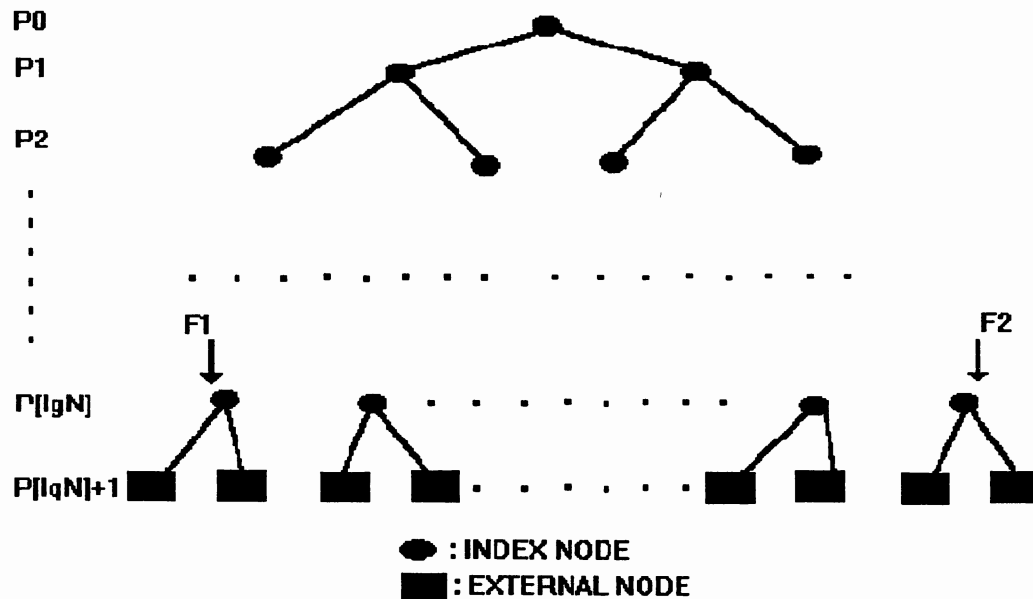


Figure 12. Two-Finger distributed 2-3-4 tree

Usually in performing top-down operations on the 2-3-4 tree, insertion for example, we start from the root. If the root is full (having four pointers,) we split the root into two nodes having one parent, which is the new root. This process increases the height of the tree by one more level. Suppose that we want to start from a non-root node, for example a finger node. If the starting node is full, having four pointers, a split might propagate to the upper levels contradicting the main idea of the top-down algorithm which is not to propagate splitting upwards followed by an insertion. The same thing is going to happen if we are performing deletion operation. If the start node has only two pointers, a merge in this case is needed. That causes upwardly propagated merging which should not happen in a

top-down algorithm.

The above discussion leads to the following proposition (for trees with two fingers as defined earlier:)

PROPOSITION 1

In a top-down algorithm, the starting node other than the root must subtend the desired key and should be a safe node.

Proof.

By the definition of subtending node, it is obvious that the desired key can be reached only by a traversal path which includes the subtending node. Therefore, the search can be initiated only at a subtending node.

If we start from an unsafe node, it might involve an upwardly propagating merge or split which contradicts the principle of the top-down updating, so we have to start from a safe node.

In order to make the two fingers work concurrently, all the processors at the cube initialization will create a new process, meaning that each processor has two working processes, a parent process and a child one. In each processor, both processes wait for a finger search. The parent process may perform other tasks besides the finger search, while the child process is restricted to help in the search process.

Before explaining the technique of implementing two fingers to the distributed 2-3-4 tree, it is necessary to define the FINGER-ACCESS(F, k) message.

FINGER-ACCESS(F , k): FINGER-ACCESS is a message where F is a finger and k is the desired key. This message can be sent from host to the leaf-predecessor processor or from a processor to its predecessor. At the beginning, this message is sent from the host to the leaf predecessor processor, which has the two fingers, to start the desired operation. If the given finger F in the leaf-predecessor processor does not subtend k or is not safe, it sends the message to its predecessor processor.

When an operation - searching, insertion, or deletion- is needed by the user, the host and the nodes will do the following:

Actions performed by the host processor:

1 - Sends a FINGER-ACCESS(F1, k) message with the finger F1 and the key k to the parent process in leaf predecessor processor.

2 - Sends a FINGER-ACCESS(F2, k) message with the finger F2 and the key k to the child process in leaf predecessor processor.

3 - Waits to receive the turning node, which subtends the needed key, and the processor number which has the turning node. This information can be sent from either process in any processor.

4 - Sends a message with the desired operation with the parameters key value and turning node to the parent process in the processor having the turning node to perform the desired operation as

discussed in Chapter III.

5 - Waits to receive the response, either fail or success, for the given operation.

Actions performed by the index and the leaf predecessor processors:

A. The Parent process:

- Receives FINGER-ACCESS(p1, k) message, where p1 is the address of the current node and k is the desired key.
- If the current node,p1, is safe and subtends k, i.e. the largest key in it is larger than k,

  then

  send to the host a message indicating that the turning node, p1, of the desired Key is found.

  else

  Send a FINGER-ACCESS(p1.parent, k) message with the address of the current node parent and the key k to the parent process in the predecessor processor.

- endif

B. The Child process:

- Receives FINGER-ACCESS(p2, k) message, where p2 is the address of the current node and k the desired key.
- If the current node, p2, is safe and subtends k, i.e. the smallest key in it is smaller than k,

  then

  send to the host a message indicating that the turning node of the desired key is found.

else

Send a FINGER-ACCESS(p2.parent, k) message with the address of the current parent and k.

- endif

We can notice from the above algorithm, that only one processor will be able to find the turning node and send it the host.

As we described above, the distributed 2-3-4 tree has two fingers, and an operation starts from the leaf-predecessor processor instead of the root processor in the ordinary case. In the next section, we describe the implementation of a scheme which allows multiple fingers to access the tree asynchronously.

Multi-Finger Implementation

In the above process of accessing a node in a distributed 2-3-4 tree by using two fingers, we noticed that it is necessary to go up in the tree starting from the finger until we reach a suitable starting node. Then from the starting or the turning node, the search proceeds down until the desired key or location is found.

In this section we describe a method for making multiple fingers to the tree that takes advantage of the asynchronous processing environment. The basic idea is to use two fingers in each level of the 2-3-4 tree except for the leaf level. Every level of the tree as we described earlier in Chapter II is stored in one processor. If we have p working processors, then we will be having 2(p-1)

fingers to the tree. Because we have many fingers in this case, it is difficult to store them in the host, so the fingers are stored in the associated processors. If we store the fingers in the host as we did in the two fingers implementation, the host needs to send p different messages to the working processors. Therefore, we need O(p) time to activate the asynchronous searching, where p is the number of working processors. On the other hand, if we store every two fingers in the associated processor, the activation of the search process takes only O(1) because the host can broadcast identical messages to all processors at once. In both cases the space needed for storing fingers is O(p). Figure 13 represents the multi-finger distributed 2-3-4 tree.

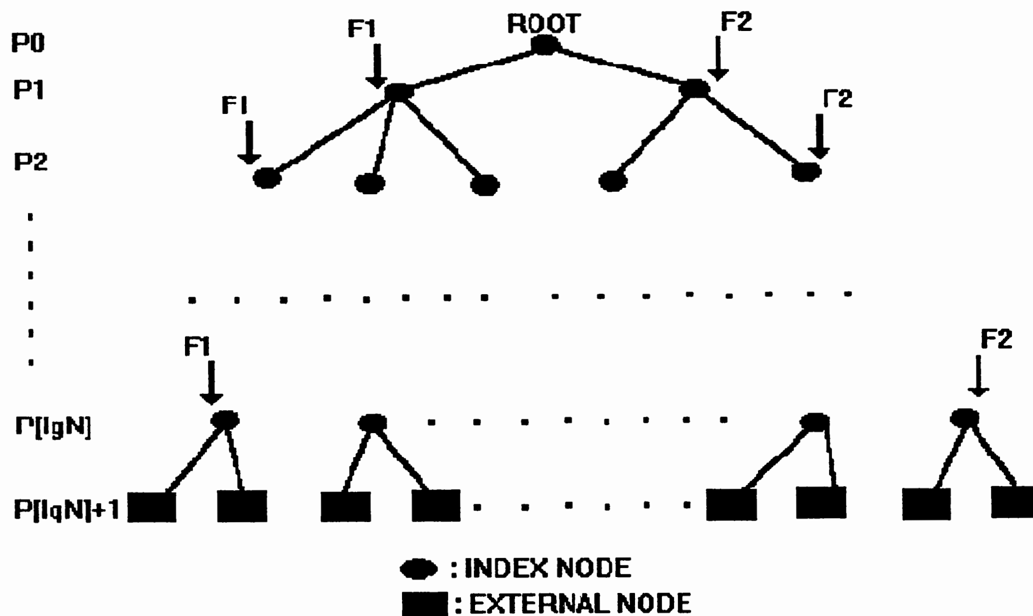


Figure 13. Multi-Finger distributed 2-3-4 tree

Before describing how multiple fingers work, let us define the following message:

MULTI-FINGER-ACCESS(Op-code , k): where Op-code is the desired operation, and k is the search key to be inserted or deleted. The host simultaneously sends this message to all working processors. When a processor receives this message, it performs the multi-finger access operation. Algorithm 4.1 describes this process.

Let us now describe how these fingers work simultaneously. When the host wants to perform an operation, it sends a MULTI-FINGER-ACCESS(Op-code, k) message to all working processors at the same time. The message contains the type of requested operation and the desired key. Asynchronously, every processor, as described in algorithm 4.1, checks if one of its finger nodes subtends the given key and if the node is a safe node. If these two conditions are satisfied, then mark the processor as a busy processor and mark that finger as a starting node. Each busy processor performs the desired operation, indicated by Op-code, as discussed earlier in Chapter III, and saves every structure as an old structure before modifying it. When the operation is passed to a free successor processor, it is performed as in the predecessor processor. But when the operation is sent to a busy successor processor, the sender needs to stop and restore its original structure because in that case, a lower level processor is also performing the same operation. The lower level processor performs the operation faster than the upper level

processors because the distance between the lower level processor and the external data nodes is shorter than the distance between the upper level processor and the external nodes. Figure 14 provides an illustration of insertion of a key (key 2) using multiple fingers to an existing 2-3-4 tree.

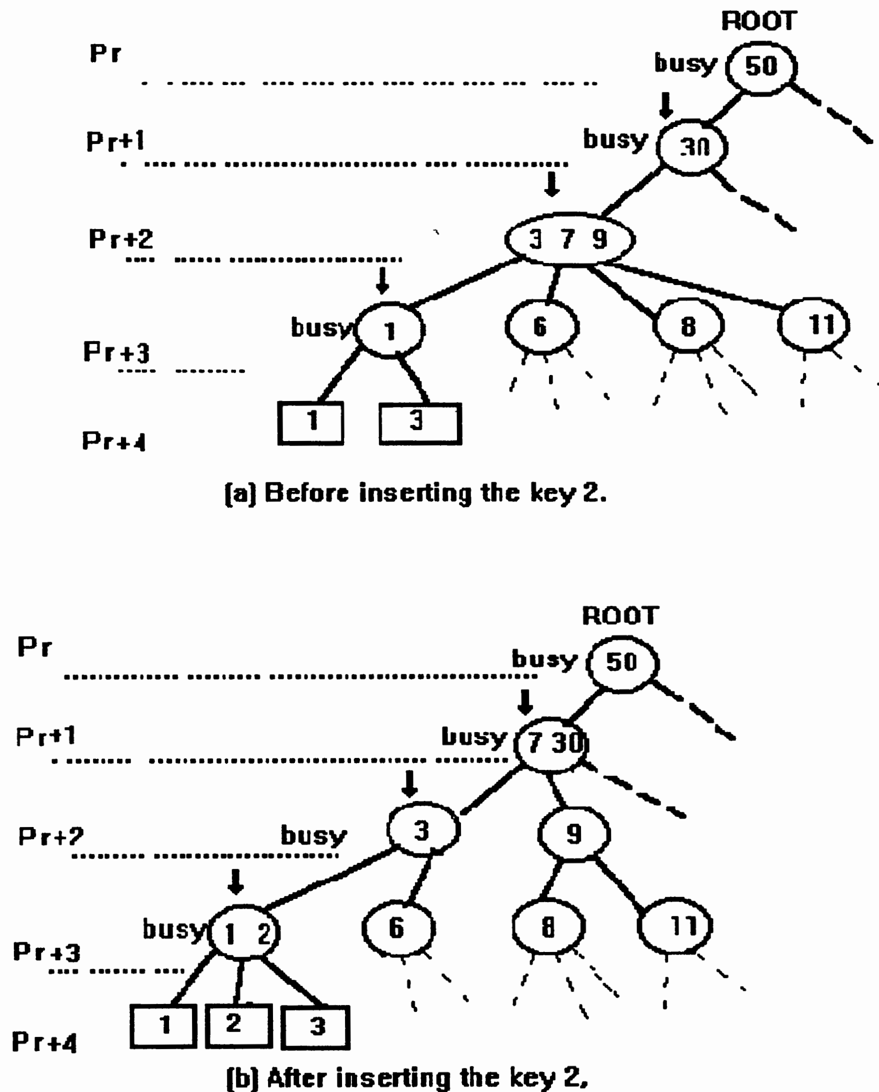


Figure 14. Insertion Example Through Multi_Fingers.

In Figure 14, the right subtree is not shown because

the insertion will take place in the left subtree. In Figure 14.a illustrates the states of processors while they compare the incoming key against their fingers. The processors storing the first level, the second level, and the fourth level of the tree will declare themselves as busy processors because their fingers subtend the key 2 to be inserted and also are safe nodes. But the finger of the third level of the tree is not safe even though it subtends the key 2. The lowest level busy processor is the one that stores the fourth level of the tree. So it performs the insertion operation.

Algorithm 4.1

```
multi_fin_access()
{
 busy_pro  <-- 0 ;
 if I am ROOT processor
   then get the starting time;

 if I am not leaf processor
 then
     /* check the left finger (f1) for this processor */
     if(incoming key<=f1.key AND /* if f1 subtends the
                                    desired key*/
       ( insert opr AND f1 has < 4 pointers OR
         delete opr AND f1 has > 2 pointers OR
         search opr ) )
          then assign adr to the address of f1; /* f1
                                                  succeeds*/
  else
     /* check the right finger (f2) for this processor */
     if(incoming key<=f2.key AND /* if f2 subtends the
                                    desired key*/
       ( insert opr AND f2 has < 4 pointers OR
         delete opr AND f2 has > 2 pointers OR
         search opr ) )
          then assign adr to the address of f2; /* f2
                                                  succeeds*/
  if one of the two fingers of this processor has succeeded
   then
       busy_pro  <-- 1; /* busy */
       switch of opr {
```

```
          1 : call search( adr );
          2 : call insert( adr );
          3 : call delete( adr );
        } /* switch */
  } /* non leaf processors */
 Post asynchronous MULTI-FIN-ACCESS message.
} /* end multi_fin_access */
```

Safety and Integrity

In this section, we will discuss the integrity of the data structure after an update operation. Since multiple asynchronous processors take part in the operations, it may seem that there exists the possibility of corruption of data. The following propositions guarantee data integrity and correctness of the operations.

PROPOSITION 2

When the multi-finger algorithm is applied, only one busy processor performs the desired operation.

Proof. (By induction)

Induction basis: When we have only one processor, obviously it performs the operation alone.

Inductive hypothesis: If we have n busy processors, we assume only one of them will complete the operation.

Inductive Step: Based on the given hypothesis, if we add one more busy processor to the n busy processors, either it will be the lowest level one or not. If it is not the lowest processor, that means it is going to quit the operation because it will find at least one lower busy processor. On the other hand, if it is the lowest level processor, all the upper level processors will quit the

operation because all of them will pass the transaction down until it reaches the lower level processor. Because it is busy, all of them will quit as described in the algorithm.

PROPOSITION 3

By applying the multi-finger algorithm, only the lowest level busy processor successfully completes the operation. The lowest finger is the quickest finger to perform any operation.

Proof.

As discussed in the algorithm, if a busy processor discovers that it has a busy successor processor, it will quit the operation. The lowest level finger will complete the operation because it is the only node which does not have a successor processor.

The tree traversal cost mainly depends on how many levels need to be traversed. The lowest finger traverses the smallest number of levels, so it provides the quickest path to reach the desired external node or location.

Finger Initialization: To utilize the fingers of the distributed 2-3-4 tree effectively, they need to be maintained carefully. At initialization time of the distributed 2-3-4 tree, every processor stores only one node. Both the left and the right fingers of that level are assigned to that node.

Finger Updating: The process of updating fingers is done only after the corresponding transformations. The

transformations that might affect the fingers are splitting and merging.

When we make a split, the newly created node is set to the right of the old full node, meaning that we are expanding that level of the tree to the right. As an observation, the left finger will not be affected by the split, but the right finger might be affected. If the right finger is pointing to the old node because it is the right most node in that level, then it should be changed to point to the new node because it becomes the right most node of that level.

In like manner, when we make a merge operation, a node will be merged to its left sibling. If that node is the right finger, then the right finger is changed to the sibling.

The implementations of the split and merge operations are done by a mechanism that expands the 2-3-4 tree to the right, and contracts the tree to the left; consequently, the left fingers of all levels of the 2-3-4 tree do not need to be changed because the left side of the tree is fixed.

Performance Analysis

In the two fingers implementation of the distributed 2-3-4 tree, the access time to a key through either finger depends on the linear distance between the nearest finger and the desired key. Therefore, the cost of any operation on a distributed finger 2-3-4 tree is $O(\log_2 d)$, where d is the distance between the desired key and the nearest finger.

While in the ordinary distributed 2-3-4 tree, the cost is $O(\log_2 N)$ time, where N is the number of keys in the tree. This two-finger implementation performs better if the accessed key is in the vicinity of either fingers.

However, the time to perform an operation using the multi-fingers approach is expected to be much shorter than the one when the operation starts from the root of the tree. The performance analysis of this section is based on experiments run with different tree sizes. Execution time for various operations are measured. To compute the time complexity, we employed the root processor to calculate the elapsed time for the desired operations both by using fingers and by starting from the root. Tables, 2 - 10, show the average elapsed time for performing the tree operations: search, insertion, and deletion using 32, 16, and 8 processors for each operation. In each table, the operations are performed for 5 different sizes of the 2-3-4 tree. The selected number of keys is proportional to the number of working iPSC/2 processors. The average elapsed time is taken for 10 random operations in each case. From the 9 tables, it can be observed that in all cases performing the distributed 2-3-4 tree operations through fingers yields better results than performing them starting from the root of the tree. The elapsed time to perform an operation through fingers depends mainly on the height of the 2-3-4 tree, while the elapsed time to perform an operation starting from the root of the tree depends on the number of working processors. As the number of processors

is constant, the time to perform an operation from the root stays almost constant.

TABLE 2

THE AVERAGE ELAPSED TIME FOR SEARCH THROUGH FINGERS VERSUS FROM ROOT WITH P = 32

# of Keys	Fingers msec	Root msec
256	3.6	23.0
1024	4.2	22.8
4096	4.6	22.6
16384	5.4	22.8
65536	5.6	23.2

TABLE 3

THE AVERAGE ELAPSED TIME FOR SEARCH THROUGH FINGERS VERSUS FROM ROOT WITH P = 16

# of Keys	Fingers msec	Root msec
128	3.4	11.0
512	4.0	11.2
2048	4.4	10.6
8192	4.8	11.0
32768	5.4	11.0

TABLE 4

THE AVERAGE ELAPSED TIME FOR SEARCH THROUGH FINGERS VERSUS FROM ROOT WITH P = 8

# of Keys	Fingers msec	Root msec
8	2.2	5.4
16	2.6	5.4
32	2.6	5.2
64	2.4	5.6
128	2.8	5.6

TABLE 5

THE AVERAGE ELAPSED TIME FOR INSERT THROUGH FINGERS VERSUS FROM ROOT WITH P = 32

# of Keys	Fingers msec	Root msec
256	20.6	58.8
1024	19.4	58.6
4096	19.6	58.8
16384	20.6	58.8
65536	20.4	58.2

TABLE 6

THE AVERAGE ELAPSED TIME FOR INSERT THROUGH FINGERS VERSUS FROM ROOT WITH P = 16

# of Keys	Fingers msec	Root msec
128	10.2	27.8
512	11.8	27.8
2048	12.0	27.4

TABLE 6 (continued)

# of Keys	Fingers msec	Root msec
8192	15.0	29.8
32768	14.8	27.6

TABLE 7

THE AVERAGE ELAPSED TIME FOR INSERT THROUGH FINGERS VERSUS FROM ROOT WITH P = 8

# of Keys	Fingers msec	Root msec
8	5.4	13.2
16	6.6	12.6
32	6.0	12.8
64	6.2	12.8
128	7.2	12.4

TABLE 8

THE AVERAGE ELAPSED TIME FOR DELETE THROUGH FINGERS VERSUS FROM ROOT WITH P = 32

# of Keys	Fingers msec	Root msec
256	7.8	27.8
1024	9.8	30.2
4096	11.4	32.6
16384	12.4	33.8
65536	13.2	34.2

TABLE 9

THE AVERAGE ELAPSED TIME FOR DELETE THROUGH FINGERS VERSUS FROM ROOT WITH P = 16

# of Keys	Fingers msec	Root msec
128	6.6	19.8
512	7.4	18.4
2048	9.6	19.6
8192	11.4	22.0
32768	12.8	22.2

TABLE 10

THE AVERAGE ELAPSED TIME FOR DELETE THROUGH FINGERS VERSUS FROM ROOT WITH P = 8

# of Keys	Fingers msec	Root msec
8	3.2	7.8
16	4.2	8.2
32	5.2	8.8
64	4.6	9.2
128	6.6	10.2

The best case performance of the distributed multi-finger 2-3-4 tree operations is an order of magnitude better than the "start from root" approach. A search or update operation on the ordinary distributed 2-3-4 tree is performed in $O(\log_2 N)$ time as the best case because all data nodes are in the same leaf level. While in the distributed

finger 2-3-4 tree, the best case to perform an operation is O(1) time.

The different implementations of the fingers we have shown so far are for ephemeral finger 2-3-4 tree. An update operation will change the structure of the tree. In the next Chapter, we will discuss the persistent implementation, where the old structure will be preserved at an update operation.

CHAPTER V

PERSISTENT TREE STRUCTURES

Methods of Making a Tree Structure Persistent

A persistent tree structure is a structure that supports access to multiple versions. In this Chapter, we consider the implementation of persistent search trees. When an update operation is performed in a persistent structure, a new version to represent the updated structure need to be created. So, we need a mechanism to retain the old version of the tree when a new version is created by an update. There are four methods of making the structure persistent found in the literature. These four methods are outlined below.

Copying the Entire Tree

Every time an update operation is performed, the entire tree is copied into a new tree with the update. It is a simple method to program. But it takes O(n) time and space complexity per update, where n is the number of elements in the tree.

Path Copying

This method is presented by Driscoll [9], Sarnak [20,21], and used by New [16]. The idea is to copy only the nodes in which changes are made. In other words, copy all nodes that are encountered on the insertion or deletion path starting from the root. The effect of this method is to create a set of search trees, one per update, having different roots but sharing common subtrees. The path copying method has the ability to update any version of the structure [13,14]. The major draw back of this method is that it requires $O(\log_2 N)$ space to make a single update, where N is the number of the keys in the tree.

Time Stamp

This method is presented by Sarnak and Tarjan[21] and Driscoll[9]. Their idea is to implement the trees without any node copying by allowing node to become arbitrarily "fat": each time we want to change a pointer, the new pointer is stored in the node, along with a time stamp indicating when the change has occurred and a bit that indicates whether the new pointer is a left or right pointer. Actually, we can know the direction of the new pointer by comparing the key of the item in the node containing the pointer to that of the item in the node indicated by the pointer. With this approach, an insertion or deletion in a persistent tree takes only O(1) space, since an insertion creates only one new node and either kind

of update causes only O(1) pointer change. The drawback of this method as described by Sarnak and Tarjan [21] is its time penalty, since a node can contain an arbitrary number of left or right pointers, deciding which one to follow during a search is not a constant time operation. Choosing the correct pointer takes $O(\log_2 m)$ time, and the time for an access, insertion, or deletion is $O((\log_2 n)(\log_2 m))$.

Limited Node Copying

This method, which is presented by Driscoll [9]; and Kazerouni-zand and Fisher[13], is introduced to eliminate the drawback of the time stamp or fat node method. In this method, a node in the persistent structure is not allowed to become arbitrarily fat. Each node is allowed to hold only a fixed number of pointers. When we run out of space in a node for new pointer, we create a new copy of the node, containing only the current field values. In every current predecessor of the node being copied, a pointer to the new copy is stored. If there is no space in a predecessor for such a pointer, the predecessor, too, must be copied. Thus node copying can ripple backwards through the structure. An update operation takes only O(1) space in the amortized case and $O(\log_2 n)$ time in the worst case [9]. Making a B-tree persistent with this method was presented by Kazerouni-zand and Fisher[13,14].

In the following sections, we describe a scheme for implementation of persistent structures in a distributed memory system.

Persistent Distributed 2-3-4 Tree Implementation

The main idea behind the time stamp method and the limited node copying method is not to make any changes to the keys stored in a node. This is accomplished by changing pointers in the case of deletion and creation of a new node in the case of insertion. In these two methods of making the structure persistent, if we change node keys, all the previous versions will be changed.

As an (a,b)-tree, the 2-3-4 tree update operations (insertion, deletion, splitting, and merging,) involve changing the keys of the nodes as shown in Figure 15. So we cannot use the method of time stamp or limited node copying described at the beginning of this Chapter to make the tree persistent. There is a modification of the structure that allows us to use the time stamp methods for the 2-3-4 tree. The modification proposed by Kazerouni-zand and Fisher[14] is to use a time stamp for each key in the node and to allow more than one pointer for each child with the time stamps.

The Distributed 2-3-4 Tree Persistent Operations

To make the tree structure persistent, we follow the work of Kazerouni-zand and Fisher[14] of making B-trees persistent. However, we are working on a distributed 2-3-4 tree using top-down updating method.

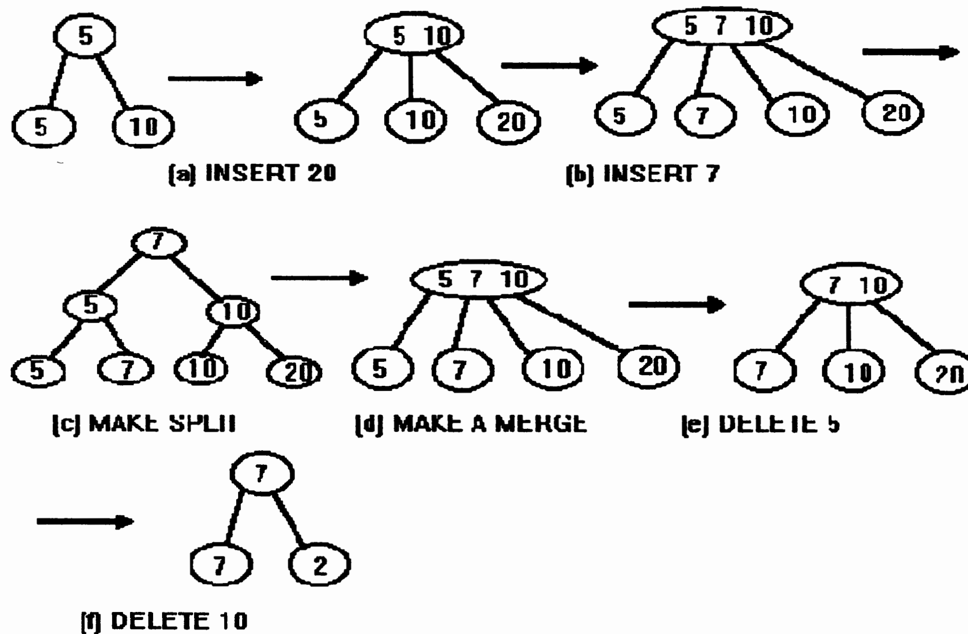


Figure 15. 2-3-4 Tree Update Operations. In all update cases a, b, c, d, e, and f the keys are changed.

We will use the limited node copying method to make our distributed 2-3-4 tree persistent. Before describing the operations, let us give the structure of the tree nodes. Each node of the 2-3-4 tree contains the following fields:

- Key[0..2] :

  Key,

  Key_Ver_No.

- Children[0..3] :

  Ptr[m],

  Ptr_ver_no[m].

Where m is the number of auxiliary pointers for each child. If we select a large value of m, that may reduce the storage

requirement needed by the persistent distributed 2-3-4 tree because the number of node copying operation decreases although the space per a node increases.

The host maintains a header node for the tree roots associated with different versions. The header structure is as following:

Ver_No		. . . . . .
Proc	Ptr	. . . . . .

Where Ver_no is a sequence of numbers: 0, 1, 2, ...

Proc is the processor number which stores the root of the tree as it appears in the associated version Ver_No.

Ptr is a pointer to the root of the tree which exists in processor number Proc.

Now, we can describe the operations.

Search. The host accesses the header node which indicates the processor number storing the tree root and the desired pointer to follow for a given search version number. The host sends a PERS_SEARCH(Ver, k, p) message containing the search version number Ver , the desired key k, and the address of the tree root p to the Proc processor. The Proc processor makes key comparisons and sends a PERS_SEARCH(Ver, k, p) message to its successor. When a processor receives a PERS_SEARCH(Ver, k, p) message, it makes the same steps as in search operation in the Ephemeral 2-3-4 tree described in

Chapter III, except that we need to make a search on the pointers of the selected child. The pointer with the largest version number less than or equal to the Ver is selected. The operation repeats itself in each processor until the leaf processor is reached. At the leaf processor, if the received pointer points to a data node containing the desired key, it sends a PERS_SEARCH_REPLY(status) message with a positive result to the host. Otherwise, it sends a negative result.

Insertion. In the matter of communication between the processors, the insertion algorithm is similar to the one in Chapter III, but the manipulation and transformation are different. Because we are dealing with a partial persistent structure, the insertion operation will be on the last version of the structure.

When the PERS_INSERT(Ver, k, Ptr) message is received by a processor, say i, it compares the keys of the node indicated by Ptr to decide from which child it follows, then it selects one of the child pointers that has the highest version number, say child_ptr. The current processor then sends PERS_INSERT_TRANSFORM(Ver, k, child_ptr) message with the current version number, the new key k, and the child pointer to the successor processor i+1. When processor i+1 receives the message, it does one of the two following cases.

CASE 1: Make a split: If the node addressed by the incoming message is full, having 3 keys, then we create two new nodes

and distribute the outgoing pointers and the keys in the old one between the new two nodes. The old node; however, is maintained because it serves previous versions. The keys of the newly created nodes are stamped with the current version number. We copy only the most recent outgoing pointer from each child pointer. Then, processor i+1 sends a PERS-INSERT-TRANSFORM-REPLY(status, Ptr1, Ptr2, m) to its predecessor i carrying a status indicating either there was a transformation or not, the addresses of the two new nodes namely Ptr1 and Ptr2, and the middle key m of the old node. When processor i receives the reply, if the status indicates a modification, it stores the key m in the next available position and stamps it with the current version number. The addresses of the two newly created nodes are stored in the proper locations as in Figure 16, which is adapted from [13] and modified.

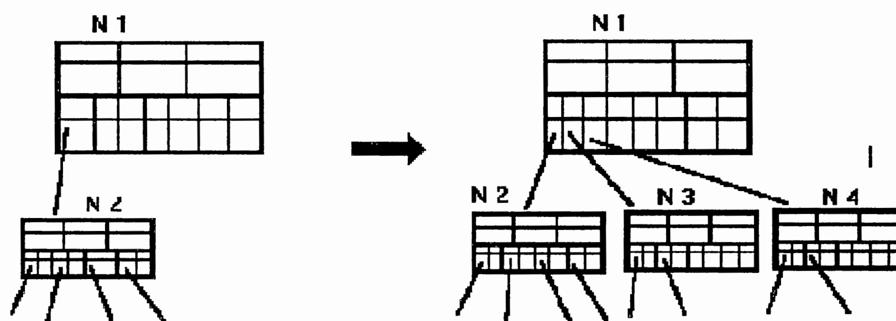


Figure 16. Insertion Persistent Transformation I. The number of the auxiliary pointers, m, for each child is 2 in this Figure.

CASE 2: Make a copy: The processor selects the proper child in the addressed node by child_ptr in the incoming message. If all of the auxiliary pointers of the selected child are allocated, then we have to make a copy of this node to avoid an upwardly propagating split. The pointers with the highest version numbers and all keys are copied as shown in Figure 17. This persistent transformation is to guarantee that if a lower level node makes a split operation, the selected child of the current node has available slots for the addresses of the new nodes resulting from the split. Thus node copying cannot ripple backwards through the structure. In order to reply to the predecessor, the processor i+1 sends a PERS-INSERT-TRANSFORM-REPLY(status, Ptr) message to its predecessor carrying a status and the address of the new node.

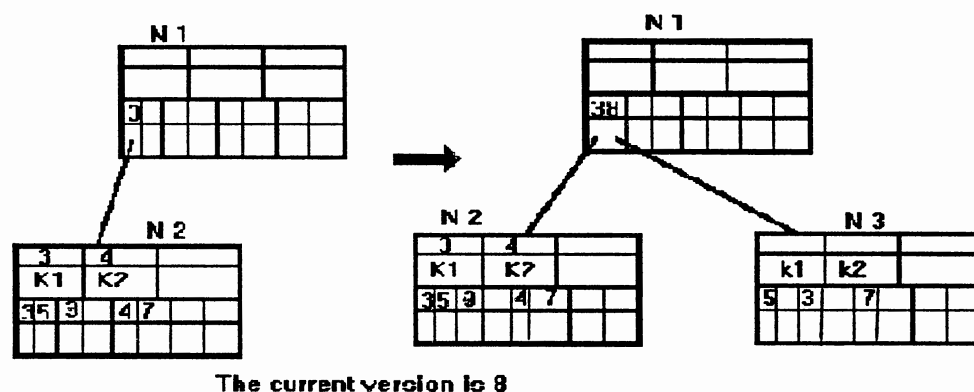


Figure 17. Insertion Persistent Transformation II. The number of the auxiliary pointers, m, for each child is 2 in this Figure.

CASE 3: If neither of the two conditions in cases 1 & 2 is

met, then the processor i+1 sends a 'No change' message to its predecessor.

When the processor i receives from its successor the PERS-INSERT-TRANSFORM-REPLY(status, ptr) message, it stores the given addresses, if any, in its auxiliary pointers. Again, it searches for the proper pointer and sends an PERS-INSERT(Ver, k, Ptr) message this time to its successor processor i+1.

Deletion. The deletion algorithm is similar to the one described in Chapter III, but the transformations do not destroy the old structures to enable accessing by previous versions. Because we are dealing with a partial persistent structure, the update operation takes place on the last version of the structure.
When the PERS-DELETE(Ver, k, ptr) message is received by a processor, say i, it compares the key k with the keys in the node addressed by ptr to decide from which child it follows, then it selects one of the child pointers that has the highest version number. It sends PERS-DELETE-TRANSFORM(Ver, ptr1, ptr2, m, k) message with the current version number, the child pointer, the child sibling pointer, the key m discriminating between the child and its sibling, and the key k need to be deleted to the successor processor i+1.
When processor i+1 receives the message, it does one of the four following cases.
CASE 1: Make a merge: If the node addressed by ptr1 in the incoming message has only two pointers, and its sibling ptr2

has also two pointers, then we create a new node and merge the keys of the two nodes forming one node with three keys and four pointers. The two old nodes; however, are maintained because they serve previous versions. This transformation is to avoid upwardly propagating merging and achieve a top-down updating approach. The keys of the newly created node are stamped with the current version number as in Figure 18. The processor i+1 sends a PERS-DELETE-TRANSFORM-REPLY(status, ptr) message to its predecessor i carrying a status and the addresses of the new node.

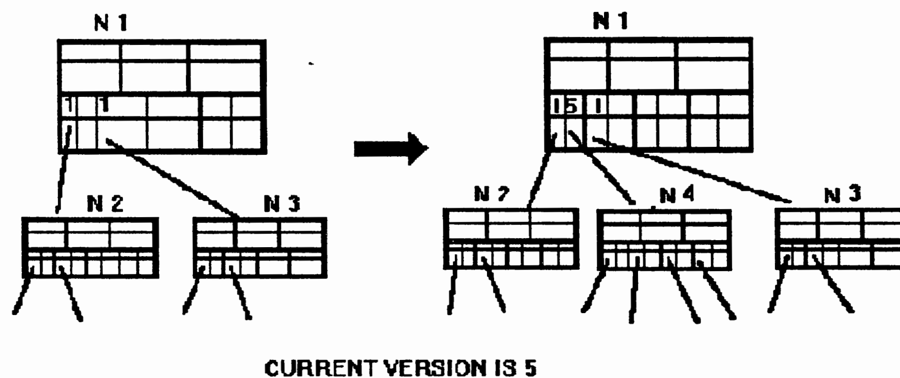


Figure 18. Deletion Persistent Transformation.
The number of the auxiliary pointers, m, for each child is 2 in this Figure.

Case 2: Borrow one pointer: If the node addressed by the incoming message have only two pointers, and its adjacent node has more than two pointers, then the addressed node borrows one pointer from a neighbor node. The old structure is preserved in this case even without copying both nodes. This transformation does not affect the previous version.

Again, this transformation is to achieve a top-down updating approach.

CASE 3: Make a copy: Using the given version number and the key k, the processor selects the proper child in the node addressed by ptr1 in the incoming message. If all of auxiliary pointers of the selected child are allocated, then we have to make a copy of this node to avoid upwardly propagating split. Only the pointers with the highest version numbers are copied as well as all keys. This transformation is to guarantee that if a lower level node makes a merge operation, the selected child of the current node has an available slot for the address of the merged nodes.

The processor i+1 sends a PERS-DELETE-TRANSFORM-REPLY(status, ptr) to its predecessor carrying a status and the address of the new node.

CASE 4: If none of the three conditions in cases 1, 2 and 3 is met, then the processor i+1 sends a 'No change' message to its predecessor.

When the processor i receives the PERS-DELETE-TRANSFORM-REPLY(status, ptr) message, it stores the given pointer, if any, in its auxiliary pointers. Again, it searches for the proper pointer and sends an PERS-DELETE(Ver, k, ptr) message this time to its successor processor i+1.

Persistent Multi-Finger 2-3-4 Tree

The update operations of the ephemeral finger 2-3-4

tree described in Chapter IV may change the structure of the tree as well as the fingers. Therefore, new methods should be devised to maintain persistent finger 2-3-4 trees. In this section, a mechanism for maintaining persistent multi-fingers' distributed 2-3-4 tree is presented.

Based on the persistent implementation scheme of the distributed 2-3-4 tree explained in the previous section, we will add fingers to the structure. Every processor maintains left and right fingers resulting in 2*p fingers to the structure, where p is the number of working processors. Every processor retains its own fingers. In the persistent implementation of the fingers, we will make use of an array of pointers to fingers for each left and right finger. The selected approach to persistence is the limited node copying method. Therefore, the array of fingers will be fixed. The following is a conceptual view of the array of fingers:

0	2		m
Ver_No	Ver_No	- - - -	Ver_No
Ptr	Ptr	- - - -	Ptr

Where m is the same m in the previous section which indicates the number of pointers for each child.

When a new version of a finger needs to be added to the finger array and there is no available slots, a new finger array is created and linked to the previous finger array. The new finger pointer is stored in the new finger array. When an update operation is applied to the tree, always

there will be a copying or a pointer change to preserve the old structure. But the fingers need to be preserved only when the update operation affects the fingers. Because we are dealing with partial persistent structure, a pointer to the most recent finger is maintained to enable update operations to access the right finger in O(1). A binary search is needed to locate the right finger in a search operation. Update operations force finger copying in the following cases:

1 - A node with 4 children on the access path of an insertion operation is split into two nodes. If the most recent left finger points to the full node, a new left finger is created and assigned to the left one of the two newly created nodes. On the other hand, if the most recent right finger points to the full node, A new right finger is created and assigned to the right one of the two newly created nodes.

2 - In the access path of an insertion or deletion, if we reach a node whose child's auxiliary pointers are allocated, then a copy of this node is made as described in the second insertion persistent transformation in the previous section. If the most recent left finger points to that node, a new left finger is created and assigned to the new copy of the node. On the other hand, if the most recent right finger points to that node, a new right finger is created and assigned to the new copy of the node. Figure 19 shows an example of this case.

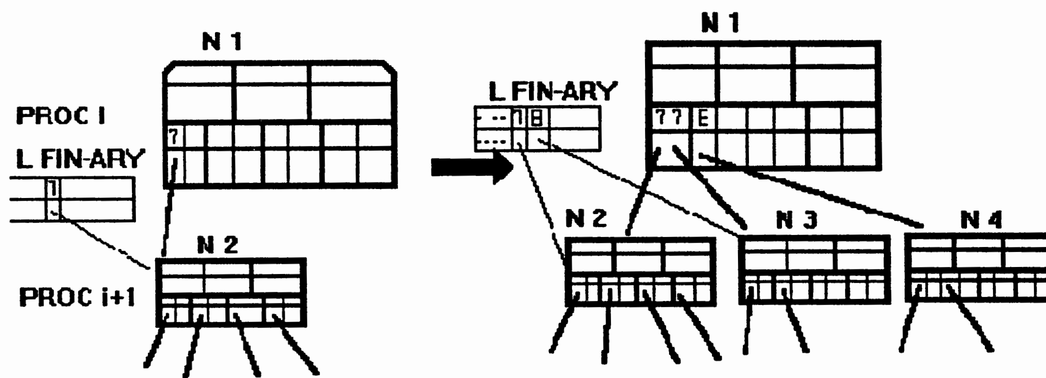


Figure 19. Persistent Multi-Finger 2-3-4 Tree. The number of the auxiliary pointers, m, for each child is 2 in this Figure.

Performance Analysis

The 2-3-4 tree has 3 keys and 4 pointers. In order to make the distributed 2-3-4 tree persistent, we need to increase the storage requirement for each node by 3*2 bytes for the keys' version numbers and 4*(m(2+4)) bytes for the children auxiliary pointers and their version numbers assuming that two bytes are needed for the version number and four bytes are needed for the auxiliary pointers. For a tree of N nodes, the extra space required to make the tree persistent is N(3*2 + 4*m*6) which is O(m*N) besides the nodes need to be copied when all auxiliary pointers of a child in the access path are allocated. The number of node copies is proportional to 1/m.

In the ephemeral implementation of the multi-finger distributed 2-3-4 tree, the storage required is $2*4*\log_2 N$ bytes, where 4 is the number of bytes needed for a finger

pointer and N is the number of keys in the tree. While in the persistent implementation, the storage required is $2*4*m*\log_2 N$ bytes, where m is the number of possible fingers for each finger array. Thus, the increase in the storage requirement for maintaining fingers is $O(m*\log_2 N)$.

An extra field in the header node is maintained to indicate the last version number [13]. This field allows access to the proper tree root for update operations in O(1), while for a search operation, a binary search is needed.

To access the right finger in a search operation, a binary search is needed costing $O(\log_2 m)$, while in update operations, the last version of a finger is indicated a pointer, so a direct access can be achieved in this case.

CHAPTER VI

CONCLUSIONS

In this thesis, we have presented techniques for mapping several forms of 2-3-4 trees to the hypercube architecture. A level to a processor mapping is selected. We have shown that a tree of N keys requires at least $\log_2 N+1$ processors. A storage requirement of $O(4^i)$ is needed to store the ith level of the tree in processor i. A wrap around mapping is presented which overcomes the problem associated with the bound of the number of processors.

Operations of the distributed 2-3-4 tree cost $O(\log_2 N)$. However, our implementation of the top-down updating allows pipelining the operations. We have shown that in such a case, an operation of the distributed 2-3-4 tree completes after O(1). Some empirical results on time complexity are also given.

Two different implementations of the distributed finger 2-3-4 tree are presented. The first implementation uses two fingers and the second implementation (called multi-finger implementation) uses two fingers at each level of the tree. The fingers are stored in such a way that they can be activated in O(1) time. Empirical results show that the multi-finger approach is more efficient.

The finger concept has also been applied to persistent

2-3-4 tree in a distributed environment. A method for incorporating fingers with persistent 2-3-4 trees is presented in the thesis. Future study will be targeted to the implementation of other data structures in the distributed environment. The effect of pipelining in the shared memory environment needs to be explored and compared against the distributed memory architectures.

REFERENCES

[1] Bier, T., Loe, K. "Embedding of binary trees into hypercubes." J. of Parallel and Distributed Computing, 6(1989), 679-691.

[2] Brown, M. R., Tarjan, R. E. "Design and analysis of a data structure for representing sorted lists." SIAM J. Comput. 9, 3(Aug. 1980), 594 - 614.

[3] Carey, M. J., Thompson, C. D. "An efficient implementation of search trees on O(log N) processors. " Dept. Comput. Sci., Univ. California, Berkeley, CA, Rep. UCB/CSD 82/101, (Nov 1982).

[4] Carey, M. J., Thompson, C. D. "An efficient implementation of search trees on [Lg N + 1] processors." IEEE Trans. on computers, c-33, (Nov. 1984), 1038-1041.

[5] Chen, W., Stallmann, M. F., and Gehringer, E. F. " Hypercube embedding heuristics: an evaluation." International J. of Parallel Programming, 18(Nov. 1989), 505-549.

[6] Colbrook, A., Smythe, C. "Efficient implementations of search trees on parallel distributed memory architectures." IEE Proc., E-5, 137(Sep 1990), 394-400.

[7] Dekel, E., Peng, S., and Lyengar, S.S. "Optimal parallel algorithms for constructing and maintaining a balanced m-way search tree." Int. J. of Parallel Programming, 15(Nov. 1986), 503-518.

[8] Dehne, F. and Andrew, R. "Implementing data structures on a hypercube multiprocessor, and applications in parallel computational geometry." J. of Parallel and Distributed Computing, 8(1990), 367-375.

[9] Driscoll, J., Sarnak, N., Sleator, D., Tarjan. R. "Making data structures persistent." Proc. 18th Ann. ACM Symp., Berkeley, CA (May 1986), 1 - 13.

[10] Ellis, C. "Concurrent search and insertion in AVL trees." IEEE Trans. Comput. C-29, 9(1980), 8 - 817.

[11] Fisher, A.L. "Dictionary machines with a small number of processors." Proc of the 11th Annual Int. Symp. on Computer Architecture, IEEE, New York, 1984, 151-156.

[12] Guibas, L., Sedgewick, R. " A Dichromatic framework for balanced trees." Proc. 19th Symposium on the foundations of Computer science, 1978.

[13] Kazerouni-zand,M. and Fisher, D. D., "Deletion on persistent B-trees." Proc. of the workshop on applied computing '89, Stillwater, Oklahoma, March 30, 1989, 90-96.

[14] Kazerrouni-zand, M. and Fisher, D. D., "A space efficient persistent B-tree." Proc. of Second Oklahoma Applied Computing Workshop, Tulsa, Oklahoma, March 88, 295-318.

[15] Kosaraju, S. R. "Localized search in sorted list." Proc. 13th Annual ACM Symp. on theory of computing (1981), 62-69.

[16] New, H. H. "Concurrent operations in persistent search trees." Master thesis, Dept. of computing and information sciences, Oklahoma State University, Stillwater, OK, 1985.

[17] O'Gorman, R. "The RPA - making the array approach acceptable." Major Advances in Parallel Processing, 130-146.

[18] Rao, N., Kumar, V. "Parallel depth first search. Part I. Implementation." Int. J. of Parallel Programming, 16(Nov. 1987), 479-499.

[19] Rattner, J. "Concurrent processing: a new direction in scientific computing." Technical paper, TO-9, National Computer Conference, 1985, 1-9.

[20] Sarnak, N. I. " Persistent data structures." Dept. of computer science, New York University, New York, NY: 1986, Dissertation.

[21] Sarnak, N., Tarjan, R. E. "Planar point location using persistent search trees." Comm. ACM 7, 29(July 1986), 669 - 697.

[22] Shieh, J. J., Papachristou, C. A. "Fine grain mapping strategy for multiprocessor systems." IEE Proc. E-3, 138(May 1991), 109 - 120.

[23] Tarjan, R. E. "Efficient top-down updating of red-black trees." Tech. Rept. CS-TR-013-86, Princeton Univ., (June 1985).

[24] Tarjan, R. E., Van Wyck, C. J. "An O(n log log n) - time algorithm for triangulating a simple polygon." SIAM J. Comput. 17, 1(Feb. 1988), 143 - 178.

[25] Upfal, E., Wigderson, A. "How to share memory in distributed system." J. of the ACM 2, 34(Jan 1987), 6-127.

[26] Wu, A. "Embedding of tree networks into hypercubes." J. of Parallel and Distributed Computing, 2(1985), 238-249.

APPENDIXES

APPENDIX A

HOST AND NODE ALGORITHMS

Host Program Algorithm

The following is the general algorithm for the host program given in appendix C.

Step 1. Read the desired number of nodes.

Step 2. Allocate a cube.

Step 3. Load all nodes with the node program.

Step 4. Initialize the cube :

- Send INIT message with a big key to the ROOT node.
- Receive The address of the tree root from the ROOT node.

Step 5. Print a menu & get the user choice and data.

Step 6. Send a message according to the user desired function to the ROOT processor or to all processors in case of multi-finger implementation.

Step 7. Receive a reply message from leaf processor indicating the success or the failure of the operation with the elapsed time.

Step 8. Go to step 5 until the user choose to exit from the system.

Step 9. Kill the cube.

Step 10. Release the cube.

Step 11. End Host.

Node Program Algorithm

The following is the general algorithm for the node program given in appendix D.

Step 1. Get process id from the system.

Step 2. Get the current node number from the system.

Step 3. Receive a message from host indicating the number of working processors (size).

Step 4. If my-node < size /* I am working processor */

Step 5. Receive INIT message from predecessor.

Step 6. Initialize a level of the tree in the current processor as following:

- Create a node with four NULL pointers.
- Send a reply with the address of the newly created node to predecessor processor.
- Send an INIT message to the successor processor.
- Receive a reply from the successor. carrying the offset of its node.
- Set the node first pointer to the received offset.
- End INIT.

Step 7. Post asynchronous messages and get their IDs for : PRINT, SEARCH, INSERT, INSERTTRANS, DELETE, DELETETRANS.

Step 8. While TRUE do

- Check mail box for incoming messages :

. Using the messages' IDs check all posted asynchronous messages.

. If any arrive, call the corresponding procedure.

(the procedures are: print(), search(), insert(), inserttrans(), delete(), and deletetrans().)

Step 9. End while.

Step 10. End node.

APPENDIX B

CAREY AND THOMPSON ALGORITHM

There are three different codes for each of processor 1 to k-2, processor k-1, and processor k.

The Code for Processors Pi, i=1,2,..., k-2

```
While true do
 Receive reqMsg from Pi-1;
 case MsgType(reqMsg) of
    SEARCH:
         begin
           Perform path selection;
            Send SEARCH(n,p') to P_i+1;
         end;
    INSERT:
       begin
          Perform path selection;
          Send INSERT_TRANSFORM(p') to P_i+1;
          Receive INSERT_TRANSFORM_REPLY(m,np) from P_i+1;
          if (np != nil) then
             Insert np and m into current index node;
          endif;
          Send INSERT(n,p') to P_i+1;
       end;

    DELETE:
       begin
          Perform path selection;
          Send DELETE_TRANSFORM(m,p',p") to P_i+1;
          Receive DELETE_TRANSFORM_REPLY(m',np) from
          P_i+1;
          Replace old splitting key with m';
          if (np != nil) then
             Delete np from current index node;
          endif;
          Send DELETE(n,p') to P_i+1;
       end;
   INSERT_TRANSFORM:
       begin
          Perform an insertion transformation if
          applicable;
          Send INSERT_TRANSFORM_REPLY(m,np) to P_i-1
       end;

   DELETE_TRANSFORM:
       begin
          Perform a deletion transformation if
          applicable;
          Send DELETE_TRANSFORM_REPLY(m',np) to P_i-1;
       end;
   endcase;
endwhile;
```

The code for processor P_{k-1}

```
While true do

 Receive reqMsg from P(k-2);
 case MsgType(reqMsg) of

    SEARCH:
         begin
            Perform path selection;
            Send SEARCH(n,p') to P(k);
         end;

    INSERT:
       begin
          Perform path selection;
          Send INSERT(n,p') to P(k);
          Receive INSERT_REPLY(np);
          if (np != nil) then
             Insert np and m into current index node;
          endif;
       end;

    DELETE:
       begin
          Perform path selection;
          Send DELETE(n,p') to P(k);
          Receive DELETE_REPLY(status);
          if ( status = no error ) then
           Delete p'  from the current index node;
          endif;
       end;

   INSERT_TRANSFORM:
       begin
          Perform an insertion transformation if
          applicable;
          Send INSERT_TRANSFORM_REPLY(m,np) to P(k-2);
       end;

   DELETE_TRANSFORM:
       begin
          Perform a deletion transformation if
          applicable;
          Send DELETE_TRANSFORM_REPLY(m',np) to P(k-2);
       end;
  endcase;
endwhile;
```

The code for processor P_k

```
while true do
   Receive reqMsg from P_{k-1};
   case MsgType(reqMsg) of

        SEARCH:
           begin
             if ( data item found) then
                 send out data item;
               else
                 Send out error response;
              endif;
           end;

     INSERT:
         begin
            if (data item not found) then
                 Insert data item;
                 Send INSERT_REPLY(nil) to P_{k-1};
                 Send out acknowledgement;
             else
                 Send INSERT_REPLY(nil) to P_{k-1};
                 Send out error response;
             endif;
          end;

     DELETE:
         begin
            if (data item not found) then
                 Delete data item;
                 Send DELETE_REPLY(no error) to P_{k-1};
                 Send out acknowledgement;
             else
                 Send DELETE_REPLY(error) to P_{k-1};
                 Send out error response;
             endif;
          end;
    endcase;
endwhile;
```

APPENDIX C

SOURCE CODE OF HOST PROGRAM

```
/**********************************************************
*  Routine name : host.c                                  *
*  Purposes    : 1. allocate cube                         *
*                  2. get user informtion                 *
*                  3. send the size of the cube           *
*                  Loop                                   *
*                   4. get user desired operations        *
*                   5. Send the request to processors     *
*                   6. Receive the result from processor *
*                   7. Update timings                     *
*                  endLoop                                *
*                  8. At the end kill the cube            *
*                  9. Release the cube                    *
**********************************************************/
#include <cube.h>
#include <stdio.h>


#define HOST_PID 100    /* process id of the host process */
#define NODE_PID   0    /* process id for node processes */
#define ALL_NODES -1    /* symbol for all nodes */
#define ALL_PIDS  -1    /* symbol for all processes */
#define ROOT       0
#define INIT       0
#define INITREP   20
#define SIZE_TYPE  1    /* type of size message */
#define TIME_TYPE  2    /* type of time message */
#define INSERT    5    /* type of insert msg */
#define DELETE     6    /* type of delete msg */
#define SEARCH     9    /* type of search msg */
#define PRINT       10    /* type of print  msg*/
#define MULTI_FIN_ACCESS 16  /* type of search thru fingers */
#define DISFIN      17    * type of dispaly fingers   */
#define REALROOT     18    /* type of real root processor */
#define RESULT      21    /* type of final result      */
#define PIPE        22    /* type of pipelining operations */
#define LARGE     200000   /* Used to indicate the maximum
                                number of keys */


/***********************************************
*                  data types                 *
***********************************************/
typedef struct tree {    /* this is the structure of the 2-3-4
                                   tree nodes. */
      int   key[3] ;
  struct tree *point[4];
  struct tree *paret ;
 } treetype;
```

```
/*****************************************************
*                   declaration                      *
*****************************************************/
typedef  struct  buffer  {  /*  buffer  is  the  means  of
                              communication */
   int   n ; /* the key need to be searched, inserted, or
              deleted. */
   int  status ;  /* fail or  success */
   int  fin    ;  /* fin=1 if the operation need to be thru
                         fingers */
unsigned long tim; /*  elapsed time for the operation */
struct tree  *p[4];
} buffertype;

typedef struct {/* used to maintained the operation timings */
         unsigned long with[1000],without[1000];
         int  w,wo ;
   }operation;

typedef struct {  /* for the general timing result */
     operation search,
               insert,
               delete;
   } result ;



 int real_id;  /* real root message id */
 int  realroot; /* indicates the real root */
int size;           /* number of working nodes */
buffertype buf ;  /* means of communication */
treetype *root ;/*pointing to he root of the tree in proc 0 */
result res ;
unsigned long starttime;
unsigned long tms, ms, tsec,  /* time calculation variables */
              sec, min;
char    nd[3] ;
    /* gray code sequence */
int array[32]={0,1,3,2,6,7,5,4,12,13,15,14,10,11,9,8,
    24,25,27,26,30,31,29,28,20,21,2,22,18,19,17,16};

int   fin     ;  /* fin=1 if the operation need to be thru
                    fingers */
int  items  ;  /* the nuber of keys in the tree */
int leaf    ;  /* leaf processor #  */

float  seed=1  ;

/****************************************
*  floor   function                     *
****************************************/
 float floor(x)
float x;
{
```

```
 long  i;
i=x;
      return i;
}

/*****************************************************
*  Function Name :  random                           *
*  Input paras   :  Nothing                          *
*  Output paras  :  Nothing                          *
*  Return value  :  random number                    *
*  Purpose       :  to produce a new random number   *
*****************************************************/


float random ()
{
float a=16807,
      q=127773,
      m=2147483647,
      r=2836;
float lo,hi,test;

 hi=floor(seed/q);
 lo=seed-q*hi;
 test=a*lo - r*hi;
 if ( test > 0 ) seed=test;
    else seed= test+m ;
  return(seed/m);
}

/************************************************
*  my-succ() return the successor processor of *
*            my_node processor                 *
************************************************/

int  my_succ(my_node)
int my_node;
{
int i ;
 if(my_node==leaf) {
           printf("ERROR1\n"); return -1; }
for (i=0;i<32;i++)
 if (array[i]==my_node){
                        return array[i+1];
                       }
printf("Error in suc\n");
}

/************************************************
*  my-pred() retur the predessor processor of  *
*            my_node processor                 *
************************************************/
int  my_pred(my_node)
int my_node;
{
```

```
int i;
  if (my_node==ROOT) { printf("ERROR2\n"); return -1; }

for (i=0;i<32;i++) if (array[i]==my_node){
                         reurn array[i-1];
                        }
printf("Error in prd\n");

}

/***************************************************
*     dump() the buffer communication for testing *
***************************************************/
dump(b)
buffertype b ;
{

 printf("\n BUFFER: n=%d  status=%d\n",b.n,b.tatus);
 printf("p0=%X  p1=%X  p2=%X
 p3=%X\n",b.p[0],b.p[1],b.p[2],b.p[3]);
}



/*******************************************************
*  Menu() is to print user menu and get the desired   *
*        function                                     *
*******************************************************
menu(x)
int *x ;
{
printf("\n");
printf("*******************************************************
        ******\n");
printf("*
        *\n");
printf("*                OPERATION ON 2-3-4 TREES
        *\n");
printf("*
        *\n");
printf("*  1. initialize the tree randomly.
        *\n");
printf("*  2. insert new item starting from root.
        *\n");
printf("*  3. insert new item starting from Fingers.
        *\n");
printf("*  4. delete an item starting from root.
        *\n");
printf("*  5. delete an item starting from Fingers.
        *\n");
printf("*  6. Search for an item starting from root.
        *\n");
printf("*  7. Search for an item starting from Fingers.
        *\n");
printf("*  8. print he tree in in-order traversal.
```

```
        *\n");
printf("*  9. Display fingers.
         *\n");
printf("* 10. Print resulting times on the screen.
         *\n);
printf("* 11. Print reslting times on output file.
         *\n");
printf("*  0. exit from the program.
         *\n");
printf("*
         *\n");
printf("*****************************************************
        *****\n\n");
printf("Enter the number of required function : ");
scanf("%d",x);
 if (*x==3 || *x==5 || *x==7) fin = 1; /* thru fingers */
   else  fin = 0 ; /* thru root */
}

/*************************************************
* initcube() toinitialize all processor at the  *
*            beginning of execution              *
*************************************************/
initcube(x)
 int x ;
{
treetype *new;
   buf.n = x ;
   buf.p[1] = NULL ;
   csend(INIT,&x,sizeof(x),ROOT,NODE_PID);
   crecv(INITREP,&buf,sizeof(buf));
   root= bufp[0] ;

}



/*************************************************
*  printinorder() for in-order printing of the    *
*                2-3-4 tree.                      *
*************************************************/

printinorder()
{
int i,j,k,t ;
 buf.p[0] = root;
 buf.p[1] = NULL;
 buf.p[2] = NULL;
 buf.p[3] = NULL;
t=1000+15*items;

 csend(PRINT,&buf,sizeof(buf),ROOT,NODE_PD);
for(i=0;i<t;i++)
for(k=0;k<20;k++) {
          printf("");  /* delay for synchronization */
```

```
          j=i/2/3/4/5*5*6*7/3/4;
  }
}

/*****************************************************
* checksize() to adjust improper cube size          *
*****************************************************/

checksize(i)
int  *i;
{

if ( *i<=2) *i = 4;
 else
if ( *i<=4) *i = 4;
 else
if ( *i<=8) *i = 8;
 else
if ( *i<=16) *i = 16;
 else
      *i = 32 ;

}

/************************************************
* insertpipeline() to pipeline  x insertion    *
*                  operations.                 *
************************************************/

int insertpipline(x)
 int x ;
{
int i;

buf.n = x ;

       buf.p[0] = root ;
       buf.fin = 0 ;
       for(i0;i<1000;i++) printf("");
    csend(INSERT,&buf,sizeof(buf),
            my_pred(my_pred(my_prd(realroot))),NODE_PID);
}



/*****************************************************
* insert() is to direct the insertion request either *
*          to the root of the tree or to all fingers *
*          the get the responce and return it to the *
*          caller.                                   *
*****************************************************/

int insert(x)
 int x ;
{
```

```
unsigned long t;
int i;

buf.n = x ;

 if ( fin ) {
      buf.status = 2 ; /* insertion code used to find the safe
                              node */
       buf.fin   = 1 ; /* indicates that the opertion is thru
                              fingers */

csend(MULTI_FIN_ACCESS,&buf,sizeof(buf),ALL_NODES,NODE_PID);
      }
 else {
       buf.p[0] = root ;
       buf.fin = 0 ;
       csend(INSERT,&bu,sizeof(buf),ROOT,NODE_PID);
      }
crecv(RESULT,&buf,sizeof(buf)); /* message expected from the
               nodes maintaingthe external nodes, size- */
  t = buf.time ;

 if (fin) res.insert.with[res.insert.w++] =  t;
    else
       res.insert.without[res.insert.wo++] =  t;
return buf.status ;   /* which indicate fail or success */
}



/******************************************************
* delete() is to direct the deletion  request either *
*          to the root of thetree or to all fingers *
*          the get the responce and return it to the *
*          caller.                                    *
******************************************************/

delete(x)
 int x ;
{
long t;

buf.n = x ;
 if ( fin ) {
      buf.status = 3 ; /* deletion code used to findthe safe
                         node */
       buf.fin   = 1 ;/* indicates that the opertion is thru
                         fingers */

csend(MULTI_FIN_ACCESS,&buf,sizeof(buf),ALL_NODES,NODE_PID);
      }
 else {
       buf.p[0] = root ;
       buf.fin = 0 ;
       csend(DELETE,&buf,sizeof(buf),ROOT,NODE_PID);
```

```
      }
crecv(RESULT,&buf,sizeof(buf)); /* message expected from the
                 node maintaing the external nodes, size-1 */
  t = buf.time
 if (fin) res.delete.with[res.delete.w++] =  t;

  else
       res.delete.without[res.delete.wo++] =  t;

return buf.status ;   /* which inicate fail or success */
}



/*****************************************************
* search() is to direct the search    request either *
*          to the root of the tree or to all fingers *
*          the get the responce and return it to the *
*          caller.                                   *

*****************************************************/
search(x)
 int x ;
{
long t;

buf.n = x ;
 if ( fin ) {
      buf.status = 1 ; /*  search   code used to find the safe
                              node */
       buf.fin    = 1 ;/* indicates that the opertion is thru
                              fingers */

csend(MULTI_FIN_ACCESS,&buf,sizeof(buf),ALL_NODES,NODE_PID);
      }
 else {
       buf.p[0] = root ;
       buf.fin = 0 ;
       csend(SEARCH,&buf,sizeof(buf),ROOT,NODE_PID);
      }
crecv(RESULT,&buf,sizeof(buf)); /* message expected from the
               nodes maintaing the external nodes, size-1 */
  t = buf.time ;
 if (fin) es.search.with[res.search.w++] =  t;
  else
       res.search.ithout[res.search.wo++] =  t;

return buf.status ;   /* which indicate fail or success */
}
```

```
/*******************************************
* buildtree() is to initialize the tree with *
*             n odes at the beginning      *
*******************************************/


buildtree()
{
int key ,i;
/* FILE *pp,*fopen(); */

  for(i=0; i<200;i++) random();    /* prime the random */
  printf("\n How many nodes do you need : ");
  scanf("%d",&items) ;
  /*pp=fopen("randoms","w");*/
   fin = 0 ;
  for(i=0;i<items;i++) {
     key = random() * 1000 ;
     /*fprintf(pp,"  %d\n",key);*/
     insert(key) ;
 /*    insertpipline(key ; */
         if (msgdone(real_id))

real_id=irecv(REALROOT,&realroot,sizeof(realroot));
   }
   res.search.w=0;
   res.search.wo=0;
   res.insert.w=0;
   res.insert.wo=0;
   res.delete.w=0;
   res.delete.wo=0;
    /*fclose(pp);*/
}



/*******************************************************
*                   host       main                    *
*******************************************************/

main()
{
        /* Host main */

 int i,t,nk, x;

/*  for (i=0;i<32;i++) printf("%d %d\n",i,gray(i));*/
         res.search.w=0; /* initialize the number of */
         res.search.wo=0; /* operations for the purpose of */
         res.insert.w=0;  /* timings  */
         res.insert.wo=0;
         res.deete.w=0;
         res.delete.wo=0;
```

```
        printf("How many nodes do you want? ");
        gets(nd);

        strcat(nd,"sx");
        getcube("tree",nd,NULL,1);
        setpid(HOST_PID);
        size = (nd[0] - '0' ) ;
        if (nd[1] && nd[1]!='s') size = size*10 + nd[1]-'0';

        checksize(&size);
        leaf = size/2;
        /* Load all nodes with pid NODE_PID. */
        load "node", ALL_NODES, NODE_PID);
               /*
                 * Send message containing number of working
                 * node to all nodes.
                 */
                csend(SIZE_TYPE, &size, sizeof(size),
                     ALL_NODES, NODE_PID);
         initcube(LARGE);

                /* post asynchronous real root msg */
       real_id=irecv(REALROOT,&realroot,sizeof(realroot));
       realroot = my_pred(my_ped(leaf)) ;
     do {
         if (msgone(real_id)) { printf("REAL=%d\n",realroot);

real_id=irecv(REALROOT,&realroot,sizeof(realroot));
        }
        menu(&x);
        switch (x) {
        case 1  :
                 prinf("BEFORE INI\n");
                initcube(LARGE);
                 printf("AFTER  INIT\n");
                 buildtree();
                 reak;
        case 2  :
        case 3  : printf("Enter the new key : ");
                 scanf("%d",&nk);
                 if (insert(nk) )
                     printf("\nThe  new  item  is  inserted
                                  successfully..\n");
                 else printf("The key is redundant..\n"); ;
                 break
        case 4  :
        case 5   : pintf("Enter the element key need to be
                                              deleted: ");
                 scanf("%d",&nk);
                 if (delete(nk) )
                     printf("\nThe    item    is    deleted
                                       successfully..\n");
                 else printf("The key does not exist..\n");
                 break;
```

```
            case 6  :
            case 7   : printf("Enter the element key need to be
                                searched :");
                       scanf("%d,&nk);
                       if (search(nk) )
                           printf("\nThe item is found ..\n");
                       else  printf("The  given  key  does  not
                                        exist..\n");
                       break;
            case 8  : system("rm o.o");
                       printinorder();
                       system("cat o.o");
                       break;
            case 9  : csend(DISPFIN,"",0,ALL_NODES,NODE_PID);
                       break;
            case   10   :   printf("Search   Timing(1),   Insert
                                Timing(2),");
                       printf(" Delete Timing(3), All Timing(4)?");
                       scanf("%d",&t);
                       switch (t) {
                            case 1 : searchtime(res);
                                     break;
                            case 2 : inserttime(res);
                                     break;
                            case 3 : deletetime(res);
                                     break;
                            case 4 :  printresult(res);
                                     break;
                             }
                       break;

            case   11   :   printf("Search   Timing(1),   Insert
                                Timing(2),");
                       printf(" Delete Timing(3), All Timing(4)?");
                       scanf("%d",&t);
                       switch (t) {
                            case 1 : searchtimef(res,size);
                                     break;
                            case 2 : inserttimef(res,size);
                                     break;
                            case 3 : deletetimef(res,size);
                                     break;
                            case 4 :  printresultf(res,size);
                                     break;
                             }
                      break;

            case  0 : printresult(res);
                       break;
                }
          } while (x!=0) ;

            killcube(ALL_NODES, ALL_PIDS);
            relcube("tree");
    }       /* End host main */
```

```
/***************************************************
*                                                  *
*   All the following routines are to format the   *
*    result either on screen or on files           *
*                                                  *
***************************************************/

#include <stdio.h>

typedef struct {
         unsigned long with[1000],without[1000];
         int  w,wo ;
   }operation;

typedef struct {
     operation search,
               insert,
               delete;
   } result ;



printresultf(res,size)
result res ;
int   size;
{
int i,j,max;
char fn[20];
FILE  *p,*fopen();

printf("Output file?");
scanf("%s",fn);
p = fopen(fn,"w");
fprintf(p,"\n   The   nmber   of   working   processors   is
%2.0f\n\n",size);

if (res.search.w > res.insert.w ) max =res.search.w;
 else max=res.insert.w;
if (res.delete.w > max ) max =res.delete.w;

fprintf(p,"¦--------------¦------------------¦--------------
-----¦\n");
fprintf(p,"¦   SEARCH      ¦        INSERT      ¦       DELETE
   ¦\n");
fprintf(p,"¦--------¦-------¦--------¦-------¦--------¦-----
-----¦\n");
fprintf(p,"¦  with ¦ without¦   with ¦ without ¦   with    ¦
without ¦\n");
fprintf(p,"¦--------¦--------¦--------¦------¦--------¦-----
-----¦\n");

for (i=0;i<max;i++){
 fprintf(p,"¦");
 if    (    i<    res.search.w)    fprintf(p,"    %3d
```

```
|",res.search.with[i]);
  else fprintf(p,"        |");
 if (i< res.search.wo)
 fprintf(p,"  %3d     |",res.search.without[i]);
  else fprintf(p,"           |");

 if ( i< res.insert.w)
 fprintf(p," %3d    |",res.insert.with[i]);
  else fprintf(p,"        |");
 if (i< res.insert.wo)
 fprintf(p,"  %3d    |",res.insert.without[i]);
  else fprintf(p,"          |");
 if ( i< res.delete.w)
 fprintf(p," %3d    |",res.delete.with[i]);
  else fprintf(p,"       |");
 if (i< res.delete.wo)
 fprintf(p,"  %3d     |",res.delete.without[i]);
  else fprintf(p,"           |");
fprintf(p"\n");
}
fclose(p);
}


searchtimef(res,size)
result res ;
int   size;
{
int i,j,max;
char fn[20];
FILE  *p,*fopen();

printf("Output file?");
scanf("%s",fn);
p = fopen(fn,"w");
fprintf(p,"\n  The  number  of  working  processors  is
%2.0f\n\n",size);

 max=res.insert.w;
if (res.search.wo > max ) max =res.search.wo;

fprintf(p,"|-------------------------------------|\n");
fprintf(p,"|             S E A R C H             |\n");
fprintf(p,"|------------------|------------------|\n");
fprintf(p,"|  with Fingers    |  without fingers |\n");
fprintf(p,"|------------------|------------------|\n");
for (i=0;i<max;i++){
 fprintf(p,"|");
 if ( i< res.search.w)
 fprintf(p,"      %3d        |",res.search.with[i]);

  else fprintf(p,"                  |");
 if ( i< res.search.wo)
 fprintf(p,"     %3d        |",res.search.without[i]);
```

```
  else fprintf(p,"                    ¦");

fprintf(p,"\n");
}
}


inserttimef(res,size)
result res ;
int   size;
{
int i,j,max;
char fn[20];
FILE  *p,*fopen();

printf("Output file?");
scanf("%s",fn);
p = fopen(fn,"w");
fprintf(p,"\n  The  number  of  working  processors  is
%2.0f\n\n",size);

 max=res.insert.w;
if (res.insert.wo > ma ) max =res.insert.wo;

fprintf(p,"¦--------------------------------------¦\n");
fprintf(p,"¦             I N S E R T              ¦\n");
fprintf(p,"¦------------------¦-------------------¦\n");
fprintf(p,"¦  with Fingers    ¦  without fingers  ¦\n");
fprintf(p,"¦------------------¦-------------------¦\n");
for (i=0;i<max;i++){
 fprintf(p,"¦");
 if ( i< res.insert.w)
 fprintf(p,"      %3d         ¦",res.insert.with[i]);

  else fprintf(p,"                   ¦");
 if ( i< res.insert.wo)
 fprintf(p,"     %3d          ¦",res.insert.without[i]);

  else fprintf(p,                  ¦");

fprintf(p,"\n");
}
}
deletetimef(res,size)
result res ;
int   size;
{
int i,j,max;
char fn[20];
FILE  *p,*fopen();

printf("Output file?");scanf("%s",fn);
p = fopen(fn,"w");
fprintf(p,"\n  The  number  of  working  processors  is
%2.0f\n\n",size);
```

```
 max=res.delete.w;
if (res.delete.wo > max ) max =res.delete.wo;

fprintf(p,"¦-------------------------------------¦\n");
fprintf(p,"¦             D E L E T E             ¦\n");
fprintf(p,"¦------------------¦------------------¦\n");
fprintf(p,"¦  with Fingers    ¦  without fingers ¦\n");
fprintf(p,"¦------------------¦------------------¦\n");
for (i=0;i<max;i++){
 fprintf(p,"¦");
 if ( i< res.delete.w)
 fprintf(p,"      %3d        ¦",res.delete.with[i]);

  else fprintf(p,"                  ¦");
 if ( i< res.delete.wo)
 fprintf(p,"     %3d        ¦",res.delete.without[i]);

  else fprintf(p,"                  ¦");

fprintf(p,"\n");
}
}


printresult(res)
result res ;
{
int i,j,max;
p r i n t f ( " I = % d        D = % d        S = % d
\n",res.insert.w,re.delete.w,res.search.w);
if (res.searchw > res.insert.w ) max =res.search.w;
 else max=res.insert.w;
if (res.delete.w > max ) max =res.delete.w;

printf("¦----------------¦------------------¦--------------
-----¦\n");
printf("¦     SEARCH     ¦      INSERT      ¦       DELETE
   ¦\n");
printf("¦------¦----------¦--------¦---------¦--------¦-----
-----¦\n");
printf("¦ with   ¦  without ¦  with   ¦ without ¦  with   ¦
without ¦\n");
printf("¦-------¦----------¦--------¦---------¦--------¦----
------¦\n");

for (i=0;i<max;i++){ printf("¦");
 if ( i< res.search.w)
 printf(" %3d    ¦",res.search.with[i]);
  else printf("        ¦");
 if (i< res.search.wo)
 printf("  %3d     ¦",res.search.without[i]);
  else printf"           ¦");

 if ( i< res.insert.w)
 printf(" %3d    ¦",res.insert.with[i]);
```

```
  else printf("        |");
 if (i< res.insert.wo)
 printf("  %3d    |",res.insert.without[i]);
  else printf("        ");
 if ( i< res.delete.w)
 printf(" %d    |",res.delete.with[i]);
  else printf("        |");
 if (i< res.delete.wo)
 printf("  %3d     |",res.delete.without[i]);
  else printf("           |");
printf("\n");
}
}

searchtime(res)
result res ;
{
int i,j,max;
 max=res.insert.w;
if (res.search.wo > max ) max =res.search.wo;

printf("|--------------------------------------|\n");
printf("|             S E A R C H              |\n");
printf("|------------------|-------------------|\n");
printf("|  with Fingers    |  without fingers  |\n");
printf("|------------------|-----------------|\n");
for (i=0;i<max;i++){
 printf("|");
 if ( i< res.search.w)
 printf("       %3d         |",res.search.with[i]);

  else printf("                   |");
 if ( i< res.search.wo)
 printf("     %3d         |",res.search.without[i]);

  else printf("                  |");

printf("\n");
}
}


inserttime(res)
result res ;
{
int i,j,max;
 max=res.insert.w;
if (res.insert.wo > max ) max =res.insert.wo;

printf("|--------------------------------------|\n");
printf("|             I N S E R T              |\n");
printf("|------------------|-------------------|\n");
printf("|  with Fingers    |  without fingers  |\n");
printf("|------------------|-------------------|\n");
for (i=0;i<max;i++){
```

```
 printf("¦");
 if ( i< res.insert.w)
 printf("      %3d        ¦",res.insert.with[i]);

  else printf("                 ¦");
 if ( i< res.insert.wo)
 printf("     %3d        ¦",res.insert.without[i]);

  else printf("                 ¦");

printf("\n");
}
}




deletetime(res)
result res ;
{
int i,j,max;
 max=res.delete.w;
if (res.delete.wo > ma ) max =res.delete.wo;

printf("¦-------------------------------------¦\n");
printf("¦              D E L E T E            ¦\n");
printf("¦-------------------¦-----------------¦\n");
printf("¦  with Fingers     ¦  without fingers ¦\n");
printf("¦-------------------¦------------------¦\n");
for (i=0;i<mx;i++){
 printf("¦");
 if ( i< res.delte.w)
 printf("      %3d        ¦",res.delete.with[i]);

  else printf("                 ¦");
 if ( i< res.delete.wo)
 printf("     %3d        ¦",res.delete.without[i]);

  else printf("                 ¦");

printf("\n");
}
}
```

APPENDIX D

SOURCE CODE OF NODES' PROGRAM

```
   /*  node.c   program  */

#include <stdio.h>
#include <cube.h>

#define HOST_PID  100    /* process id of the host process */
#define NODE_PID    0    /* process id of the node process */
#define ROOT        0     /* root node id */
#define ALL_NODES  -1
#define INIT        0    /* type of initialization message */
#define SIZE_TYPE   1     /* type of size message */
#define TIME        2     /* type of time message */
#define MY_SUC      3   /* type of sending or receiving from
                              successor */
#define MY_PRED     4    /* type of sending or receiving from
                              predecessor */
#define INSERT      5    /* insert message type */
#define DELETE      6    /* delete msg type     */
#define INSERTTRANS
#define DELETETRANS 8
#define SERCH      9
#define PRINT      10
#define INSERTREP  11
#define DELETEREP  12
#define INSETTRANSREP 13
#define DELETETRANSRP 14
#define RESTORE     15     /* restore msg type */
#define MULTI_FIN_SEARCH  16     /* operation thru fingers */

#define DISPF       17     /* display fingers         */
#define REALROOT    18     /* real root type            */
#define INITREP    20    /* type of initialization message */
#define RESULT     21    /* type of final result   message */
#define PIPE       22    /* type of pipelining operations   */
#define LARGE      200000



/***********************************************************
*****/
/*                      data    types
  */
/***********************************************************
******/


typedef struct tree{  /* 2-3-4 tree nodes' structures */

  int   key[3] ;
struct tree *child[4] ;
struct tree *parent   ;
}treetype ;
```

```
typedef struct { /* buffer is the mean of communication */
  int    n ;
  int status ;
  int fin    ;   /* when 1 means operation thru fingers */
unsigned long time;
struct tree *p[4] ;
}buffertype ;


struct tree *q0,*q1;

/********************************************************/
/*                     declaration                      */
/********************************************************/

treetype *p,*p1,*p11,*np,fs0,fs1 ;

struct {  /* Fingers declarations */
        int no ;  /* number of keys in that node */
   treetype *point; /* finger pointing to the finger node */
  } f1,f2;

buffertype buf,buf1;  /* communication buffers */
      /* asynchronous msg ids   */
int dispf_id,init_id,in_id,del_id,instrans_id,
    deltrans_idsearch_id,print_id,
    time_id,real_id,rest_id,fin_access_id;
int     m,m1,n,local;
unsigned     line ;

int size  ;/* number of nodes which will work on problem */
int my_pid,          /* process id of the nodes */
    my_node;         /* node id of each node */
int  busy_pro,        /* inidicate weather the processor is
                         busy or not*/
     start_pro,    /* 1 if start processor    */
     realroot,  /* indicate the real root of the processors */
     change ; /* 1 if structure has been changed */

unsigned long starttime,  /* time variables */
             endtime ;

int temp=0;

 FILE *fp, *fopen();
int small ;
int colmn=1;
                /* Gray code sequence */
int array[32]={0,1,3,2,6,7,5,4,12,13,15,14,10,11,9,8,
      24,25,27,26,30,31,29,28,20,21,23,22,18,19,17,16};


/*******************************************
* copy() is to copy the node p to s       *
*******************************************/
```

```
copy(s,p)
treetype *s,*p;
{

int i;
for(i=0;i<3;i++)
  s->key[i]=p->key[i];

for(i=0;i<=3;i++)
  s->child[i] = p->child[i];

  s->parent = p->parent;

}

/*******************************************
* copybuf() is to copy the buffer p to s   *
*******************************************/


copybuf(s,p)
buffertype *s,*p;
{

int i;
s->n = p->n;
s->status = p->status;
s->fin = p->fin;
for(i=0;i<4;i++)
  s->p[i]=p->p[i];


}
me(x)
int x ;
{
if (my_node==x)
       return 1;
else return 0;
}


/*********************************************************
*        leaf() returns TRUE if i is a leaf processor   *
*********************************************************/

leaf(i)
int i ;
{
 if (i ==size/2) return 1 ; /* yes you are leaf */
   else return 0 ;
}
```

```
/*********************************************************
* leafpred() returns TRUE if i is a leaf pred processor *
*********************************************************/


leafpred(i)
int i ;
{
 if (i ==1+size/2) return 1 ; /* yes you are the pred of the
leaf */
   else return 0
}


/*********************************************************
* indexnode() returns TRUE if i is index    processor   *
*********************************************************/

indexnode(i)
int i ;
{
  if (!leaf(i) && !leafpred(i) )
        return 1 ; /* yes you are indexnode */
   else return 0 ;
}

prn(ms)
char ms[] ;
{
 printf("Iam%d : %s\n",my_node,ms);
}

dumpp(b)
treetype *b ;
{

 printf("p0=%X  p1=%X  p2=%X p3=%X\n",b->child[0],
            b->chid[1],b->child[2],b->child[3]);
  p r i n t f ( " k 0 = % d       k 1 = % d       k 2 = % d
\n",b->key[0],b->key[1],b->key[2]);
}



dump(b)
buffertype b ;
{

 printf("\n BUFFER: n=%d  status=%d\n",b.n,b.status);
  p r i n t f ( " p 0 = % X       p 1 = % X       p 2 = % X
p3=%X\n",b.p[0],b.p[1],b.p[2],b.p[3]);
}
```

```
/*****************************************************
*  dispf() to display evrey procesor fingers        *
*****************************************************/


dispf()
{

printf("PRO%d   FINGER1:   no=%d        :    %d    %d    %d
real=%d\n",my_node,f1.no,
            f1.point->key[0],f1.point->key[1],
f1.point->key[2],realroot);
printf("PRO%d   FINGER2:   no=%d        :    %d    %d    %d
real=%d\n",my_node,f2.no,
             f.point->key[0,f2.point->key[1],
f2.point->key[2],realroot);


    dispf_id=irecv(DISPF,"",0);
}

/**************************************************
* broadcast() is to brodcast the real root msg to *
*            all other processor and host         *
**************************************************/
broadcast(r)
int r ;
{
 csend(REALROOT,&r,sizeof(r),myhost(),HOST_PID);
 csend(REALROOT,&r,sizeof(r),ALL_NODES,NODE_PID);
}


/*****************************************************
*  print() controls the printing from that processor *
*****************************************************/

print()
{
int i,j ;

 if(leaf(my_node)) {
    fp = fopen("o.o","a");
    if (!fp) printf("ERROR FP\n\n");
    for(i=0;i<4;i++)
      if (buf.p[i])
       fprintf(fp,"%3d ",buf.p[i]->key[0]);
    if (++colmn>4) {
                        colmn=1;
                        fprintf(fp,"\n");
                      }
    else fprintf(fp,"    ");
    fclose(fp);
 }
  else {
```

```
    for(i=0;i<4;i++)
      if (buf.p[i]) {
         for(j=0;j<4;j++)
            buf1.p[j]=buf.p[i]->child[j] ;
         csend(PRINT,&buf1,sizeof(buf1),my_succ(),NODE_PID);
        }
    }
    print_id=irecv(PRINT,&buf,sizeof(buf));
} /* print */

/**********************************************
* search()  is to ontrol the searching at the *
*          current processor level           *
***********************************************/
search()
{
      /* send to the leaf the start time of this operation*/

 if (my_node==ROOT && !buf.fin){
        starttime=mclock();
  }
      if ( !local && buf.fin && busy_pro) {
       search_id=irecv(SEARCH,&buf,sizeof(buf));
        return ;
     }
  if (leaf(my_node)) busy_pro=0;
   else
   busy_pro = 1 ;

   if (leaf(my_node)) {
          if (buf.n==buf.p[0]->key[0])
                  buf.status = 1 ;/* success */
               else buf.status = 0; /* fail */
           csend(TIME,&buf,sizeof(buf),ROOT,NODE_PID);
      }
     else
     if(buf.n<=buf.p[0]->key[0]) {
          buf1.p[0] = buf.p[0]->child[0] ;
          buf1.n = buf.n ;
          buf1.fin = buf.fin;
          csend(SEARCH,&buf1,sizeof(buf1),my_succ(),NODE_PID);
      }
     else if(buf.n<=buf.p[0]->key[1]) {
          buf1.p[0]= buf.p[0]->child[1] ;
          buf1.n = buf.n ;
          buf1.fin = buf.fin;
          csnd(SEARCH,&buf1,sizeof(buf1),my_succ(),NODE_PID);
      }
     else if(buf.n<=buf.p[0]->key[2]) {
          buf1.p[0] = buf.p[0]->child[2] ;
          buf1.n = buf.n ;
          buf1.fin = buf.fin;
          csend(SEARCH,&buf1,sizeof(buf1),my_succ(),NODE_PID);
      }
       else   { /* greater than the third key */
```

```
            buf1.p[0] = buf.p[0]->child[3] ;
            buf1.n = buf.n ;
            buf1.fin = buf.fin;
            csend(SEARCH,&buf1,sizeof(buf1),my_succ(),NODE_PI);
        }
             if (!local)
          search_id=irecv(SEARCH,&buf,sizeof(buf));

} /* search */


/*****************************************************
* insertleaf() is a spcial insertion routine for the  *
*                leaf processor                       *
******************************************************/

insertleaf()
{
treetype *q;

      if ( buf.p[0] )
      if ( buf.n== buf.p[0]->key[0]) {
          buf.p[0] = NULL ;

csend(INSERTREP,&buf,sizeof(buf),my_pred(),NODE_PID);
            buf.status = 0 ; /* fail */
           csend(TIME,&buf,sizeof(buf),ROOT,NODE_PID);
           ins_id=irecv(INSERT,&buf,sizeof(buf));
           eturn;
        }

         q = (treetype *)malloc(sizeof(treetype));
         if (!q) printf("Memory Overflow\n\n\n");
       if (buf.p[0]){
              f (buf.p[0]->key[0] < buf.n )
                     q->key[0] = buf.n ;
               else {
                  q->key[0]=buf.p[0]->key[0] ;
                  buf.p[0]->key[0 = buf.n ;
                    }
              buf. = buf.p[0]->key[0] ;
              buf.status = 1 ;
           }
        else  {
                 q->key[0] = buf.n ;
                 buf.status = 4 ;  /* first item */
              }

          buf.p[0] = q ;

csend(INSERTREP,&buf,sizeof(buf),my_pred(),NODE_PID);
           buf.status = 1 ;  /* success */
           csend(TIME,&buf,sizeof(buf),ROOT,NODE_PID);
          ins_id=irecv(INSERT,&buf,sizeof(buf));
} /* ins leaf    */
```

```
/***********************************************
* insert()  is to control the insertion at the    *
*           current processor level.              *
*           A transformation is made if necessary *
*           by communicating with the successor   *
*           processor                            *
************************************************/

insert()
{
      /* send to the leaf the start time of this operation*/
 if (my_node==ROOT && !buf.fin){
        starttime=mclock(); /* the start time will be computed
                                          at leaf*/
  }
      if (leaf(my_node)) {
         isertleaf() ;
         return ;
       }

 busy_pro = 1 ;
    if (my_node==ROOT && buf.p[0]->child[3]) {
                  printf(\n\n\n TREE IS FULLLLLLLLL\n\n\n\n");
                  return;
              }
     buf1.fin = buf.fin ; /*to propagate the type of search */
     if(buf.n<=buf.p[0]->key[0]) {
          buf1.p[0] = buf.p[0]-child[0] ;
          buf1.n = buf.n ;
         if ( indexnode(my_node)) {
          csend(INSERTTRANS,&buf1,sizeof(buf1),my_succ(),
                 NODE_PID);
          crecv(INSERTTRANSREP,&buf1,sizeof(buf1));
                  }

         else
         if (leafpred( my_node)) {
          csend(INSERT,&buf1,sizeof(buf1),my_succ(),
                 NOD_PID);
          crecv(INSERTREP,&buf1,sizeof(buf1));
           }
         if (buf1.status==10 ) {   /* we should stop */
           if (!local)
            ins_id=irecv(INSERT,&buf,sizeof(buf));
            buf1.status = 0;
            return;
          }

          if(buf1.p[0]) { /* there should be t max 2 keys */
                change = 1 ;
                if(buf.p[0]->child[3])
                     printf("ERROR INSERT 4 pointers?\n");
                if (buf.p[0]->child[1]==NULL){
                     realroot = my_node ;
```

```
                broadcast(realroot);
              }
          buf.p[0]->key[2] = buf.p[0]->key[1] ;
           buf.p[0]->key[1] = buf.p[0]->key[0] ;
           buf.p[0]->child[3] = buf.p[0]->child[2] ;
           buf.p[0]->child[2] = buf.p[0]->child[1] ;
           if (buf1.status==4)
              buf.p[0]->child[0] = buf1.p[0] ;
           else {
                 buf.p[0]->key[0] = buf1.n ;
                 buf.p[0]->child[1] = buf1.p[0] ;
               }
           if ( f1.point==buf.p[0])
                   f1.no++;
        }
     else change = 0; /* there is no change */
  }
else if(buf.n<=buf.p[0]->key[1]) {
     buf1.p[0] = buf.p[0]->child[1] ;
     buf1.n = bu.n ;
    if ( indexnode(my_node)) {
     csend(INSERTTRANS,&buf1,sizeof(buf1),my_succ(),
            NODE_PID);
     crecv(INSERTTRANSREP,&buf1,sizeof(buf1));
      }
    else
    if ( leafpred(my_node)) {
     csend(INSERT,buf1,sizeof(buf1),my_succ(),
            NODE_PID);
     crecv(INSERTREP,&buf1,sizeof(buf1));
      }
    if (buf1.status==10 ) {  /* we should stop */
       buf1.status = 0;
      if (!local)
       ins_id=irecv(INSERT,&buf,sizeof(buf));
       return;
     }

     if(buf1.p[0]) {
           change = 1 ;
           if(buf.p[0]->child[3])
                printf("ERROR INSERT 4 pointers?\n");
           buf.p[0]->key[2] = buf.p[0]->key[1] ;
           buf.p[0]->key[1] = buf1.n ;
           buf.p[0]->child[3] = buf.p[0]->child2] ;
           if (buf1.status==4)
           buf.p[0]->child[1] = buf1.p[0] ;
           else
           buf.p[0]->child[2] = buf1.p[0] ;
           if ( f1.point==buf.p[0])
                   f1.no++;
        }
    else change = 0; /* there is no change */

  }
```

```
        else    {  /* greater than key[1] */
             buf1.p[0] = buf.p[0]->child[2] ;
             buf1.n = buf.n ;
            if ( indexnode(my_node)) {
             csend(INSERTTRANS,&buf1,sizeof(buf1,my_succ(),
                    NODE_PID);
             crecv(INSERTTRANSREP,&buf1,sizeof(buf1));
              }
            else
            if ( leafpred(my_node)) {
             csend(INSERT,&buf1,sizeof(buf1),my_succ(),
                    NDE_PID);
             crecv(INSERTREP,&buf1,sizeof(buf1));
              }
            if (buf1.status==10 ) {  /* we should stop */
               buf1.status = 0;
              if (!local)
               ins_id=irecv(INSERT,&buf,sizeof(buf));
               return;
             }

             if(buf1.p[0]) {
                   change = 1 ;
                   if(buf.p[0]>child[3])
                        printf("ERROR INSERT 4 pointers?\n");
                   buf.p[0->key[2] = buf1.n ;
                   if buf1.status==4)
                   buf.p[0]->child[2] = buf1.p[0] ;
                   else
                   buf.p[0]->child[3] = buf1.p[0] ;
                   if ( f1.point==buf.p[0])
                            f1.no++;
                }
            else change =0; /* there is no change */
         }
        if ( indexnode(my_node )) {
          /* Now send insert message */
               buf.status = 0 ; /* usefull for restore */

        if(buf.n<=buf.p[0]->key[0])
             buf1.p[0] = buf.p[0]->child[0] ;
        else if(buf.n<=buf.p[0]->key[1])
             buf1.p[0] = buf.p[0]->child[1] ;
        else if(buf.n<=buf.p[0]->key[2])
             buf1.p[0] = buf.p[0]->child[2] ;
        else    /* greater than the third key */
             buf1.p[0] = buf.p[0]->child[3] ;
        buf1.n = buf.n ;
        csend(INSERT,&buf1,sizeof(buf1),my_succ(),NODE_PID);
         } /* is < size -2 */
              if (!local)
          ins_id=irecv(INSERT,&buf,sizeof(buf));
    } /* insert */
```

```
/*****************************************************
* inserttrans() is to make a transformation of       *
*                splitting the full node to two nodes*
*                and give the result to predecessor  *
*****************************************************/

inserttrans()
{
   buf1.status = 0 ;
       if ( buf1.fin && busy_pro) {
            buf1.status = 10 ;
  csend(INSERTTRANSREP,&buf1,sizeof(buf1),my_pred(),NODE_PID);
 instrans_id=irecv(INSERTTRANS,&buf1,sizeofbuf1));
            return ;
          }
     copybuf(&buf,&buf1);

      if (buf.p[0]->chid[3]==NULL){
          change = 0;
          buf.p[0] = NULL;
        }
     else {
      change = 1 ; /* we made a change  */

           p = (treetype*)malloc(sizeof(treetype));
           p->key[0]           = buf.p[0]->key[2] ;
           p->key[1]           =LARG ;
           p->key[2]           = LARGE;
           p->child[0]         = buf.p[0]->child[2] ;
           p->child[1]         = buf.p[0]->child[3] ;
           p->child[2]         = NULL;
           p->child[3]         = NULL;
           buf.n               = buf.p[0]->ky[1] ;
           buf.status          =  ;
           buf.p[0]->key[1]    = LARGE;
           buf.p[0]->key[2]    = LARGE;
           buf.p[0]->child[2] = NULL;
           bu.p[0]->child[3] = NULL;
           p->parent           = buf.p[1] ;
               /* update fingers if necessary */
                /* left finger cannot be change in insert */
           if ( f2.point==buf.p[0])
             f2.point = p ; /* change right finger */
           buf.p[0]            = p ;
          }
  csend(INSERTTRANSREP,&buf,sizeof(buf),my_pred(),NODE_PID);
 instrans_id=irecv(INSRTTRANS,&buf1,sizeof(buf1));
} /* inserttrans */
```

```
/*************************************************
* delete()  is to contro the  deletion at the    *
*           urrent processor level.              *
*           A transformation is made if necessary *
*           by communicating with the successor   *
*           processor.                           *
**************************************************/
delete()
{
int sent ;
      /* send to the leaf the start time of this operation*/
 if (my_node==ROOT && !buf.fin){
       starttime=mclock();
  }

 if (leaf(my_node)) {
    buf.status = 0 ;
    if (buf.p[0] )
        if (buf.p[0]->key[0]==buf.n) {
            free(buf.p[0]);
            buf.status = 1 ; /* no error */
          }

        csend(DELETEREP,&buf,sizeof(buf),my_pred(),NODE_PID);
           csend(TIME,&buf,sizeof(buf),ROOT,NODE_PID);
           if (!local)
        del_id=irecv(DELETE,&buf,sizeof(buf));
       return;
   } /*leaf  */

  busy_pro = 1 ;

  sent = 0 ;  /* 1 indicate that we sent the delete msg */
     buf1.fin = buf.fin ; /*tp propagate the type of search */

 /* start p[0] */
 if(buf.n<=buf.p[0]->key[0]) {
        buf1.status = 0 ; /* the main is the first */
      buf1.p[0] = buf.p[0]->child[0] ;
      buf1.p[1] = buf.p[0]->child[1 ;
      buf1.n = buf.p[0]->key[0] ;
     if ( leafpred(my_node ) || buf.p[0]->child[1]==NULL) {
      buf1.n = buf.n ;
      csend(DELETE,&buf1,sizeof(buf1),my_succ(),
            NODE_PID);
      sent = 1 ; /* we sent delete message */
      if (leafpred(my_node)) {
      crecv(DELETEREP,&buf1,sizeof(bf1));
      if (buf1.status) {   /* no error */
            buf.p[0]->key[0] = buf.p[0]->key[1] ;
            buf.p[0]->key[1] = buf.p[0]->key[2] ;
            buf.p[0]->key[2] = LARGE;
            buf.p[0]->child[0] = buf.p[0]->child[1] ;
            buf.p[0]->child[1] = buf.p[0]->child[2] ;
            bf.p[0]->child[2] = buf.p[0]->child[3] ;
```

```
            buf.p[0]->child[3] = NULL;
                if ( f1.point==buf.p[0])
                        f1.no--;
        }
      }
    }
    else {
        /* indexnode */
     csend(DELETETRANS,&buf1,sizeof(buf1),my_succ(),
           NODE_PID);
     crecv(DELETETRANSREP,&buf1,sizeof(buf1));

        if (buf1.status==10 ) {  /* we should top */
           buf1.status = 0;
          if (!local)
           del_id=irecv(DELETE,&buf,sizeof(buf));
           return;
         }

     switch (buf1.status) {
       case 0 : /* No transformation ook place */
                change = 0; /* there is no change */
                break;
       case 1 :  /* two node have been merged */
               change = 1 ;
           buf.p[0]->key[0] = buf.p[0]->key[1] ;
           buf.p[0]->key[1] = buf.p[0]->key[2] ;
           buf.p[0]->key[2] = LARGE ;
           buf.p[0]->child[1] = buf.p[0]->child[2] ;
           buf.p[0]->child[2] = buf.p[0]->child[3] ;
           buf.p[0]->child[3] = NULL;
               if ( f1.point==buf.p[0])
                        f1.no--;
               if (buf.p[0]-child[1]==NULL){
                   realroot = my_node ;

                   broadcast(realroot);
                 }
           break;
      case 2 :  /* one pointer is added from neighbor  of 3*/
      cae 3 :  /* one pointer is added from neighbor  of 4*/
                /* either cases change the splitting key */

           buf.p[0]->key[0] = buf1.n;
  if ( f1.point==buf.p[0])
                        f1.n--;
               change = 1 ;
            break ;
        } /* end switch */
      } /* ndexnode  */

} /* p[0] */

 else
if(buf.n<=buf.p[0]->key[1]) {
```

```
    buf1.status = 2 ; /* means the main is the second */
  buf1.p[0] = buf.p[0]->child[0] ;
  buf1.p[1] = buf.p[0->child[1] ;
  buf1.n = buf.p[0]->key[0] ;
 if ( leafpred(my_node) || bf.p[0]->child[1]==NULL) {
  buf1.n = buf.n ;
  buf1.p[0] = buf.p[0]->child[1] ;
  csend(DELETE,&buf1,sizeof(buf1),my_succ(),
         NODE_PID);
  sent = 1 ; /* we sent delete message */
  if (lefpred(my_node)) {
  crecv(DELETEREP,&buf1,sizeof(buf1));
  if (buf1.status) {   /* no error */
        buf.p[0]->key[1] =buf.p[0]->key[2] ;
        buf.p[0]->key[2] = LARGE;
        buf.p[0]->child[1] = buf.p[0]->child[2] ;
        buf.p[0]->child[2] = buf.p[0]->child[3] ;
        buf.p[0]->child[3] = NULL;
            if ( f1.point==buf.p[0])
                    f1.no--;
      }
    }
 }
 else
 if ( indexnode(my_node)) {
  csend(DELETETRANS,&buf1,sizeof(buf1),my_succ(),
         NODE_PID);
  crec(DELETETRANSREP,&buf1,sizeof(buf1));

     if (buf1.status==10 ) {  /* we should stop */
        buf1.status = 0;
       if (!local)
        del_id=irecv(DELETE,&buf,sizeof(buf));
        return;
      }

  switch (buf1.status) {
    case 0 : /* No transformation took place */
            change = 0; /* there is no change */
            break;
    case 1 :  /* two nodes have been merge */
           change = 1 ;
        buf.p[0]->key[0] = buf.p[0]->key[1] ;
        buf.p[0]->key[1] = buf.p[0]->key[2] ;
        buf.p[0]->key[2] = LARGE ;
        buf.p[0]->child[1] = buf.p[0]->child[2] ;
        buf.p[0]->child[2] = buf.p[0]->child[3] ;
        buf.p[0]->child3] = NULL;
            if ( f1.point==buf.p[0])
                    f1.no--;
        break;
   case 2 :  /* one pointer is added from neighbor  of 3*/
   case 3 :  /* one pointer is added from neighbor  of 4*/
            /* either cases change the splitting key */
           change = 1 ;
```

```
            buf.p[0]->key[0] = buf1.n;
                if ( f1.point==buf.p[0])
                        f1.no--;
             break ;
          } /* end switch */
        } /* indexnode */

  } /* p[1] */


   else
  if(buf.n<=buf.p[0]->key[2]) {
         buf1.status = 2 ;
       buf1.p[0] = buf.p[0]->child[1] ;
       buf1.p[1] = buf.p[0]->child[2] ;
       buf1.n = buf.p[0]->key[1] ;
      if ( leafpred(my_node) || buf.p[0]->child[1]==NULL) {
       buf1.p[0] = buf.p[0]->child[2] ;
       buf1.n = buf.n ;
       csend(DELETE,&buf1,sizeof(buf1),my_succ(),
              NODE_PID);
       sent = 1 ; /* we sent delete message */
       if (leafpred(my_node)) {
       crecv(DELETEREP,&buf1,sizeof(buf1));
       if (buf1.status) {   /* no error */
             buf.p[0]->key[1] = buf.p[0]->key[2];
             buf.p[0]->key[2] = LARGE ;
             buf.p[0]->child[2] = buf.p[0]->child[3];
             buf.p[0]->child[3] = NULL;
                if ( f1.point==buf.p[0])
                         f1.no--;
         }
       }
     }
     else
     if ( indexnode(my_node)) {
      csend(DELETETRANS,&buf1,sizeof(buf1),my_succ(),

             NODE_PID);
      crecv(DELETETRANSREP,&buf1,sizeof(buf1));

         if (buf1.status==10 ) {  /* weshould stop */
            buf1.status = 0;
           if (!local)
            del_id=irecv(DELETE,&buf,sizeof(buf));
            reurn;
          }

      switch (buf1.status) {
        case 0 : /* No transfomation took place */
                 change = 0; /* there is no change */
                 break;
        case 1 :  /* two nodes have been merged */
                change = 1 ;
            buf.p[0]->key[1] = buf.p[0]->key[2] ;
```

```
            buf.p[0]->key[2] = LARGE ;
            buf.p[0]->child[2] = buf.p[0]->child[3] ;
            buf.p[0]->child[3] = NULL;

                if ( f1.point==buf.p[0])
                     f1.no--;
            break;
       case 2 :  /* one pointer is added from neighbor  of 3*/
       case 3 :  /* one pointer is added from neighbor  of 4*/
                  /* either cases change the splitting key */
                change = 1
            buf.p[0]->key[1] = buf1.n;
                if ( f1.point==buf.p[0])
                         f1.no--;
             break ;
          } /* end switch */
        } /* indexnode */

 } /* p[2] */

  else /* greater than key[2] */
 if(!buf.p[0]->child[3])  printf("ERRR3456");
 else {
        buf1.status  3 ;
      buf1.p[0] = buf.p[0]->child[2] ;
      buf1.p[1] = buf.p[0]->child[3] ;
      buf1.n = buf.p[0]->key[2] ;
     if ( leafpred(my_node) || buf.p[0]->child[1]==NULL) {
      buf1.p[0] = buf.p[0]->child[3] ;
      buf1.n = buf.n ;
      csend(DELETE,&buf1,sizeof(buf1),my_succ(),
             NODE_PID);
      sent = 1 ; /* we sent delete message */
      if (leafpred(my_node)) {
      crecv(DELETEREP,&buf1,sizeof(buf1));
      if (buf1.status) { /* no error */
        buf.p[0]->key[2] =  LARGE ;
        buf.p[0]->child[3] = NULL;
                if ( f1.point==bu.p[0])
                         f1.no--;
       }
      }
    }
     else
     if ( indenode(my_node)) {
      csend(DELETETRANS,&buf1,sizeof(buf1),my_succ(),
             NODE_PID);
      crecv(DELETETRANSREP,&buf1,sizeof(buf1));

         if (buf1.status==10 ) {  /* we should stop */
            buf1.status = 0;
           if (!local)
           del_id=irecv(DELETE,&buf,sizeof(buf));
            return;
          }
```

```
      switch (buf1.status) {
        case 0 : /* No transformation took place */
                  hange = 0; /* there is no change */
                 break;
        case 1 :  /* two nodes have been merged */
                 change = 1 ;
                  buf.p[0]->key[2] = LARGE ;
                  buf.p[0]->child[3] = NULL;
                 if ( f1.point==buf.p[0])
                         f1.no--;
                  break;
       case 2 :  /* one pointer is added from neighbor  of 3*/
       case 3 :  /* one pointer is added from neighbr  of 4*/
                  /* either cases change the splitting key */
                 change = 1 ;
                  buf.p[0]->key[2] = buf1.n;
                 if ( f1.point==buf.p[0])
                         f1.no--;
                  break ;
         } /* end swtch */
       } /* indexnode */

 } /* p[3] */
     if ( indexnode(my_node)  && sent==0) {
       /* Now send delete message if we did not sent before */


     if(buf.n<=buf.p[0]->key[0]) {
          buf1.p[0] = buf.p[0]->child[0] ;
          buf1.p[1] = buf.p[0]->child[1] ;
          buf.status = 1;
      }
     else if(buf.n<=buf.p[0]->key[1])   {
          buf1.p[0] = buf.p[0]->child[1] ;
          buf1.p[1] = buf.p[0]->child[0] ;
          buf.status = 2;
      }
     else if(buf.n<=buf.p[0]->key[2])   {
          buf1.p0] = buf.p[0]->child[2] ;
          buf1.p[1] = buf.p[0]->child[1] ;
          buf.status = 2;
      }
     else    {  /* greater than the third key */
          buf1.p[0] = buf.p[0]->child[3] ;
          buf1.p[1] = buf.p[0]->child[2] ;
          buf.status = 2;
      }
     buf1.n = buf.n ;
     csend(DELETE,&buf1,sizeof(buf1),my_succ(),NODE_PID);

      } /* ndex node   */
/* when my_node <size -1 */
           if (!local)
del_id=irecv(DELETE,&bufsizeof(buf));
} /* delete */
```

```
/****************************************************
* deletetrans() is to make a transformation of       *
*                merging to underfull nodes to one    *
*                full node, or borrrow one pointers  *
*               from neighbor if it has more than 3
*                and give the result to predecessor  *
*****************************************************/

deletetrans()
{

      if ( buf1.fin && busy_pro) {
           buf1.status = 10 ;

csend(DELETETRANSREP,&buf1,sizeof(buf1),my_pred),NODE_PID);
       deltrans_id=irecv(DLETETRANS,&buf1,sizeof(buf1));
           return ;
         }
     copybuf(&buf,&buf1);

  switch (buf.status) {
   case 0 :
   case 1 :
           if (buf.p[0]->child[2] ){
              change = 0;
               buf1.status  =  0;  /*  it  has  more  than  two
                                     ponters */
               }
           else { /* it has only two pointers */
               change = 1 ;  /* we made  a change     */
              if ( buf.p[1]->child[2] ) {
                      /* pring one pointer from neighbor */
                 buf.p[0]->key[1] = buf.n ;
                 buf1.n = uf.p[1]->key[0] ;
                 bu.p[0]->child[2] = buf.p[1]->child[0] ;
         /*               buf.p[0]->child[2]->parent    =
                   buf.p[0]->child[0]->parent;*/
                 buf.p[1]->child[0] = buf.p[1]->child[1] ;
                buf.p[1]->child[1] = buf.p[1]->child[2] ;
                 buf.p[1]->child[2] = buf.p[1]->child[3] ;
                 buf.p[1]->child[3] = NULL;
                 buf.p[1]->key[0] = buf.p[1]->key[1] ;
                 buf.p[1]->key[1] = buf.p[1]->key[2] ;
                 buf.p[1]->key[2] = LARGE;
                 buf1.status = 2 ;
                 if(buf.p[0]==f.point) f1.no++;
                  else
                 if(buf.p[1]==f1.point) f1.no--;
                }
            else { /* he neighbor has also two */
                  /* So, merge */
                 buf.p[0]->key[1]   = buf.n ;
                 buf.p[0]->key[2]   = buf.p[1]->key[0] ;
```

```
                    buf.p[0]->child[2] = buf.p[1]->child[0];
            /*              buf.p[0]->child[2]->parent   =
                      buf.p[0]->child[0]->parent;*/
                    buf.p[0]->child[3] = buf.p[1]->child[1];
            /*              buf.p[0]->child[3]->parent   =
                      buf.p[0]->child[0]->parent;*/
                 /* update fingers if necessay */
                  /* left finger cannot be change in delete */
                   if (f2.point==buf.p[1]) f2.point = buf.p[0];
             if ( f2.point==buf.p[1])
                f2.point = buf.p[0] ; /* change right finger */
                   free(buf.p[1]);
                    free ( buf.p[1]) ;
                    buf1.status = 1 ;
                    buf1.p[0]  buf.p[0] ;
                  }
              } /* else needs trans */

csend(DELETETRANSREP,&buf1,sizeof(buf1),my_pred(),NODE_PID);
              break;
   case 2 :
   case 3 :
             if (buf.p[1]-child[2] ){
                 change = 0;
                 buf1.status  =  0;  /*  it  has  more  than  to
                                     pointers */
               }
             else { /* it has only two pointers */
                  change=1;
                if ( buf.p[0]->child[2] ) {
                         /* left nieghbor has more than two */
                         /* pring one pointer from neighbor */
                    buf.p[1]->key[1] = bf.p[1]->key[0] ;
                    buf.p[1]->key[0] = buf.n ;
                    buf.p[1]->child[2] = buf.p[1->child[1] ;
                    buf.p[1]->child[1] = buf.p[1]->child[0] ;
                    if (buf.p[0]->chid[3]) { /* four pointers */

                        bu1.n = buf.p[0]->key[2] ;
                       buf.p[1]->child[0]=buf.p[0]->child[3];
                        buf.p[0]->key[2] = LARGE;
                        buf.p[0]->child[3] = NULL;
                      }
                     else
                     { /* three pointers */
                        buf1.n =buf.p[0]->key[1] ;
                       buf.p[1]->child[0] = buf.p[0]->child[2];
                        buf.p[0]->key[1] = LARGE;
                        buf.p[0]->child[2] = NULL;
                      }

                      /*         buf.p[1]->child[0]->parent  =
                           buf.p[1]->child[1]->parent;*/
                   buf1.status = 2 ;
                    if(buf.p[1]==f1.point) f1.no++;
```

```
                  else
                 if(buf.p[0]==f1.point) f1.no--;
               }
           else { /* the neighbor has also two */
                  /* So, merge */
                 buf.p[0]->key[1]   = buf.n ;
                 buf.p[0]->key[2]   = buf.p[1]->key[0] ;
                 buf.p[0]->child[2] = buf.p[1]->child[0];
               /*        buf.p[0]->child[2]->parent    =
                   buf.p[0]->child[0]->parent;*/
                 buf.p[0]->child[3] = buf.p[1]->child[1];
             /*           buf.p[0]->child[3]->parent    =
                   buf.p[0]->child[0]->prent;*/

              /* update fingers if necessary */
               /* left finger cannot be change in delete */
           if ( f2.point==buf.p[1])
               f2.point = buf.p[0] ; /* change right finger */


                 free ( buf.p[1]) ;
                 buf1.status = 1 ;
                buf1.p[0] = buf.p[0] ;
               }
            } /* else needs trans */

csend(DELETETRANSREP,&buf1,sizeof(buf1),my_pred(),NODE_PID);
            break;

        } /* end case */
      deltrans_id=ircv(DELETETRANS,&buf1,sizeof(buf1));

} /* deletetrans */

/*************************************************
* multi-fin-access() is to be activated when the *
* current processor receive a msg from host to   *
* perform the given operation through fingers    *
*************************************************/


multi_fin_access()
{
 buf.p[0] = NULL ;
 busy_pro = 0  ; /* initially none of the processors is busy*/
 start_pro= 0   ;
 if (my_node==ROOT) starttime=mclock();
  if (!leaf(my_node)){
  /* check the left finger for this processor */
    if ( ( buf.n<=f1.point->key[0]) &&/* it subtends the key*/
          ( buf.status==2 && f1.oint->child[3]==NULL || /* it
                                                  is save */
           buf.status==3  &&  (realroot==my_node  ||
                                 f.point->child[2])||
                                 buf.status==1 ) )
```

```
        buf.p[0] = f1.point ;  /* start the operation from
                                   this finger */
 else
    if ( ( buf.n>=f2.point->key[0]) &&  /* it subtends the
                                             key */
         ( uf.status==2 && f2.point->child[3]==NULL || /* it
                                                   is save */
           buf.status==3    &&    (relroot==my_node    ||
                            f2.point->child[2])||
                                 buf.status==1 ) )
        buf.p[0] = f2.point ;  /* start the operation from
                                   this finger */

 if ( buf.p[0] ) {
     change = 0 ;
     start_pro= 1 ;
     local = 1;

  switch (buf.status) {
    case 1  : searh();
              break;
    case 2  : insert();
              break;
    case 3  : delete();
              break;
     }
  }
 } /* not leaf */
    fin_access_id=irecv(MULTI_FIN_SEARCH,&buf,sizeof(buf));
} /* end multi_fin_access */


/************************************************************/
/*check_mailbox() is to monitore the asynchronous messages*/
/*if there any message call the operations handler       */
/* simulating hrecv() system call.  and call the desired  */
/* routine                                                 */
/************************************************************/

check_mailbox()

{
long  ty,no;
 local = 0 ;
 if (msgdone(init_id))
     initcube();
 if (msgdone(ins_id))
    insert() ;
 if (msgdone(del_id))
     delete();
 if (msgdone(search_id))
     search() ;
 if (msgdone(print_id))
    print(;
```

```
 if (msgdoe(fin_access_id))
     multi_fin_access();
 if (msgdone(real_id))
    real_id=irecv(REALROOT,&realroot,sizeof(realroot));
        /* do nothing; the realroot will get the value */

  if ( !leaf(my_node)) {
 if (msgdone(instrans_id))
       inserttrans();
 if (msgdone(deltrans_id))
       deletetrans);
 if (msgdone(dispf_id))
    dispf();
   }
 if(my_node==ROOT)
      if(mgdone(time_id)) {
      endtime = mclock();
      buf.time = endtime - starttime;
      csend(RESULT,&buf,sizeof(buf),myhost(),HOST_PID);
      time_id = irecv(TIME,&buf,sizeof(buf));
       }
}

/*****************************************************
* initcube(),when a processor receive  INIT msg,    *
* it will activate this routine to initialize the   *
* 2-3-4 tree in the current level.                  *
*****************************************************/
initcube()
{

 line= 0 ;
realroot = size/2+1;
  f1.point = NLL;
  f2.point = NULL;
  small = buf.n;
  if ( !leaf(my_node)) {
p=(treetype *)malloc(sizeof(treetype));
p->key[0] =  buf.n   ;
p->key[1] =  buf.n   ;
p->key[2] =  buf.n   ;
p->parent =   buf.p[1] ;
p->child[0] = NULL;
p->child[1] = NULL;
p->child[2] = NULL;
p->child[3] = NULL;

f1.point = p ;
f2.point = p ;
buf.p[0] = p ;
if (my_node==ROOT)
        csend(INITREP,&buf,sizeof(buf),myhost(),HOST_PID);
else
    csend(INITREP,&buf,sizeof(buf),my_pred(),my_pid);
  if ( !leaf(my_node)) {
```

```
        buf.p[1] = p ;   /* parent */
        csend(INIT,&buf,sizeof(buf),my_succ(),my_pid);
     }
 if (!leafprd(my_node )) {
        crecv(INITREP,&buf,sizeof(buf));
        p->child[0] = buf.p[0] ;
}
}
   init_id =irecv(INIT,&buf,sizeof(buf);

}



/**********************************************************/

main()
{       /* Node main */

        int tm,j,i,t;
        my_pid = mypid();   /* Get process id. */
        my_node = mynode(); /* Get node number. */


          /*
           * Receive message containing number of working
           * nodes.
           */
          crecv(SIZE_TYPE, &size, sizeof(size));


 if (my_node < size){
crecv(INIT,&buf,sizeof(buf));
     initcube() ;
    ins_id=irecv(INSERT,&buf,sizeof(buf));
    del_id=irecv(DELETE,&uf,sizeof(buf));
    search_id=irecv(SEARCH,&buf,sizeof(buf));
    print_id=irecv(PRINT,&buf,izeof(buf));
    fin_access_id=irecv(MULTI_FIN_SEARCH,&buf,sizeof(buf));
    real_id=irecv(REALROOT,&realroot,sizeof(realroot));

    if ( !leaf(my_node))  {
    dispf_id=irecv(DISPF,"",0);
   /*
    rest_id=irecv(RESTORE,&temp,sizeof(temp));
 */
        instrans_id=irecv(INSERTTRANS,&buf1,sizeof(buf));
        deltrans_id=irecv(DELETETRANS,&buf1,sizeof(buf1));
     }

    if (my_node==ROOT)
                time_id = irecv(TIME,&buf,sizeof(buf));
```

```
for(;;)    /* this is infinite loop to check the incoming
              messages and perform the desired tasks */
check_mailbox() ;

      } /* End if I am working node */
 } /* main */



int  my_succ()
{
int i ;
 if(leaf(my_node)) {
            printf("ERROR1\n"); return -1; }
for (i=0;i<32;i++)
 if (array[i]==my_node){
                        return array[i+1];
                       }
printf("Error in suc\n");
}

int  my_pred()
{
int i;
  if (my_node==ROOT) { printf("ERROR2\n"); return -1; }

for (i=0;i<32;i++)
 if (array[i]==my_node){
                         return array[i-1];
                       }
printf("Error in rd\n");
}
```

VITA

Abdulkader A. Al-fantookh

Candidate for the Degree of

Master of Science

Thesis: IMPLEMENTATION OF A FAMILY OF 2-3-4 TREES IN THE HYPERCUBE

Major Field: Computer Science

Biographical:

Personal Data: Born in Riyadh, Saudi Arabia, July 1, 1961, the son of Al-fantookh A. and Alfuriah N.

Education: Received Bachelor of Science Degree in Computer Science form King Saud University at Riyadh, Saudi Arabia in May, 1988; Completed requirements for the Master of Science degree at Oklahoma State University in May, 1992.

Professional Experience: Teaching Assistant , Department of Computer Science, King Saud University, July 1988, to June 1989. A student member of the ACM and Saudi Computer Society.