DESIGN OF EFFICIENT SORTING AND SEARCHING STRUCTURES AND ALGORITHMS USING A TORUS TOPOLOGY

By

BILLY D'ANGELO GASTON

Bachelor of Science

Langston University

Langston, Oklahoma

1998

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, 2001

DESIGN OF EFFICIENT SORTING AND

SEARCHING STRUCTURES AND

ALGORITHMS USING A

TORUS TOPOLOGY

Thesis Approved:

[illegible]

Thesis Adviser

D. E. Heck

[illegible]

[illegible]

Dean of the Graduate College

ACKNOWLEDGEMENTS

First and foremost I would like to thank God for he said in his word, "I will never leave you nor forsake you" (Hebrews 13:5). I am eternally grateful for his many blessings, for without Him I am nothing.

Additionally, I wish to express my gratitude to my principal advisor, Dr K. M. George for his constructive guidance, diligent supervision, encouragement, and friendship during this portion of my graduate study. My sincere appreciation extends to Dr. G. E. Hedrick for his constant support, advice, and friendship. I extend a special thank you to Dr. Nohpill Park for his personal courage and unwavering reassurance. I would also like to thank Dr. J. P. Chandler for his knowledgeable counsel and encouragement.

Finally, I would like to thank my family and friends for their continuous support and encouragement throughout this chapter of my life.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

As the fields of computer science and engineering evolve, the amount of tasks completed by a computer system in one unit of time increases. This is due largely to the development of high performance computers, as well as computer systems, which utilize parallelism between processors. The pattern in which the processors are connected for communication purposes is called a network topology. Since the conception of parallel computer systems, the overhead of communication cost has sparked many concerns, thus causing interconnection networks to be a prevalent area of research. The problem more formally stated, given a collection of nodes (processors, switches), define a set of communication links (connections) between them such that node to node communication will be minimum [21]. The topology of interconnection networks can be classified as either being direct or indirect, where static or dynamic connections apply respectively [10]. Static connections imply a processor to processor network implementation. Dynamic networks imply a switch channel implementation. In general, static networks are implemented with direct point to point connections which do not change during the execution of a process while indirect networks are essentially the central communication component between processors or between processors and memory modules [4,10]. Past research suggests that for extremely large computer systems consisting of several thousand processors, a direct interconnection scheme is desired [4]. Such a scheme is preferred because of the direct communication links between processors and simple communication protocol [4]. In general, many of the popular data structures utilized in computer science today seem to have an impact on some of the static network topologies introduced in past years. A data structure can be defined as a record or an organization of information stored in a computer's memory [18,22]. A data structure generally has associated algorithms, which perform operations to uphold its predefined properties [18]. The operations also assist in the storing and retrieval of information from within the data structure. The influence of the conventional binary tree structure can be seen in the design of the binary tree and binary fat tree interconnection networks. The channel width of a fat tree increases as we ascend from leaves to the root to resolve the potential communication bottleneck problem toward the root, since the traffic toward the root becomes heavier [10]. The linear array and the ring networks resemble the traditional array or link list structure, while the mesh and the torus interconnection network are comparable to the customary matrix structure. Just as data structures and their node configuration properties are seen in the design and implementation of old and/or new and efficient interconnection networks, the reverse can also be observed. A network's topological properties can be studied and modeled to devise a data structure and associated algorithms, which support efficient sorting and searching operations.

Much effort has been directed towards practical methods of searching and sorting data. It is said that much of a computer's processor time is spent either searching for data or sorting it [11]. Minimizing these two operations are just as important if not more, as minimizing the cost of communication between two nodes in an interconnection network. Reducing the cost of sorting and searching operations is an ongoing issue in the area of computing technology. Cost is defined in terms of time and space performance of algorithms and hardware. Several techniques have been utilized. One such approach utilizes parallelism between processors along with high-speed buses; however, the cost of such hardware can be extremely expensive, thus causing such hardware to be unattainable. When using a single processor, perhaps, two of the most efficient means of searching for data is with the use of such abstract data types as B-trees and hash tables, while heap sort and quick-sort are two successful sorting algorithms.

The purpose of this research is to explore new solutions to the problems associated with data searching and sorting. To combat these problems, we define, design and implement a data structure using a popular interconnection network. More specifically, the three dimensional torus/mesh interconnection (3DTIN) network topology along with a translation of its node labeling scheme is used to implement a data structure, which supports efficient sorting and searching operations, as well as two algorithms.

CHAPTER II

Literature Review

The sorting and searching of integers are two basic problems in the computing arena. Ordinarily when speaking of the two from a fundamental standpoint, one-processor computing is implicit. As advancements in parallel computing evolve, efficient solutions using parallel processors have been produced. Many of the parallel computers used today utilize mesh-connected/torus-connected processors [20]. This interconnection scheme is due largely to the uncomplicated interconnection topology and simple communication protocols.

Olariu, Schwing, and Zhang introduced a method of sorting N integers on a mesh connected computer of size N x N. The input values must range from 0 – (N-1) and the N must be a perfect square. They described their algorithm as a hybrid between bucket sort and radix sort [14]. Olariu, Schwing, and Zhang suggests that their algorithm can also work for N^c integers, where c is a constant number. In addition they cite the time complexity of their algorithm as O(1).

Ping Gu and Jun Gu present three algorithms which sort N^2 integers in roughly O(N) time. Using either torus or mesh-connected processors, their algorithms sort random input values in row major order and snake-like row major order [9]. A processor array M is divided into blocks numbered in either a row major or snake-like row major order. The values in each block are sorted and a series of block rotations are completed [9].

In [6], an analysis of a new algorithm called Diffusion Algorithm Searching Unbalanced Domains (DASUD) is introduced. This algorithm is presented as a solution to the load-balancing problem common in the parallel-computing environment. More specifically, the problem deals with the task of evenly distributing a workload of computation over several processors [6]. The underlining solution to this problem is to have a continuous flow of nearest neighbor communication via message passing.

In [19], Rio, Macedo, and Freitas present a method of retrieving information located in a distributed index mesh. The mesh consists of a number of independent search engines [19]. Their system consists of three components: leaf, router, and client agents [19]. An agent can be described as a software module which queries and/or retrieves and/or processes data, based on some predefined heuristic. The client agent interacts with the router and leaf agents, which work together, to create an information mesh. The router and leaf agents are arranged in a tree-like structure with the routers organized as parent nodes. The leaf agents interact with a host/resource to gather available information, build summaries based on the information obtained, and passes built summaries upward to router agents. Each router produces its own summary based on information received from its children nodes (leaf agents) and passes the information upward. The summaries consist of meta-information and keyword rankings. In the event that a client agent requests information via a query, the default router agent (root) returns communications/pointers to routers or leaf agents below it. Once a leaf agent is returned, presumably the highest-ranking data in the mesh is returned [19].

Bera and Das present an analysis of dynamic location strategies used in the area of wireless communication. Analyses were conducted on three strategies: time based updating, movement based updating, and distance base updating. In the analysis a NxN mesh was used. Each cell/node was identified with a pair ordering (i,j). The selective paging search was the search strategy used in locating a mobile user in the analysis. The strategy was proposed by Akyilidz, Ho, and Lin in [2]. In the event of a mobile user search, the cell in which the user was last reported is examined. If the user is not found, neighboring node with a distance of d, where d = 1, 2, ..., is searched until the user is found [2].

All solutions listed above utilize parallel algorithms, which yield implementation complications. Constraint is given either on the size of the network, the specific value of the number of processors used, and number of input values. When dealing with a realistic system of input, the number of integers and their values are generally randomly distributed.

CHAPTER III

3-Dimensional Torus Topology

Due to its routing and addressing schemes, the torus topology had become a popular subject of discussion. In past years, it was proposed as a possible architecture for metropolitan area networks as well as an interconnection network for multiprocessor computers [16,20]. The torus topology possesses mesh-like connections with additional wrap-around connections, where boundary nodes in each dimension are connected forming a ring. A torus can be described as a collection of rings. Formally, an N-dimensional torus has been defined as consisting of $S = k_1 * k_2 * ... * k_N$ nodes, where k_i is the size of its *i*th dimension and $k_i >= 2$. An arbitrary node is represented by a vector $[x_1, x_2, ..., x_N]$ where $x_i = 0, 1, ... k_i-1$ [1,17,20]. Each node has two neighbors per dimension suggesting a total of 2N neighbors per node. Figure 1 depicts a 3-dimensional torus having the dimensions 4*2*3. A node is identified by a 3-dimensional vector. The network is a 3-dimensional structure. The topology also depicts a network size of twenty-four.

Nezu, et al [13] cite a disadvantage of the three dimensional torus/mesh interconnection network (3DTIN) as not exhibiting an environment which contains an arbitrary network size. The one and two dimensional torus topologies are variations of the 3DTIN.

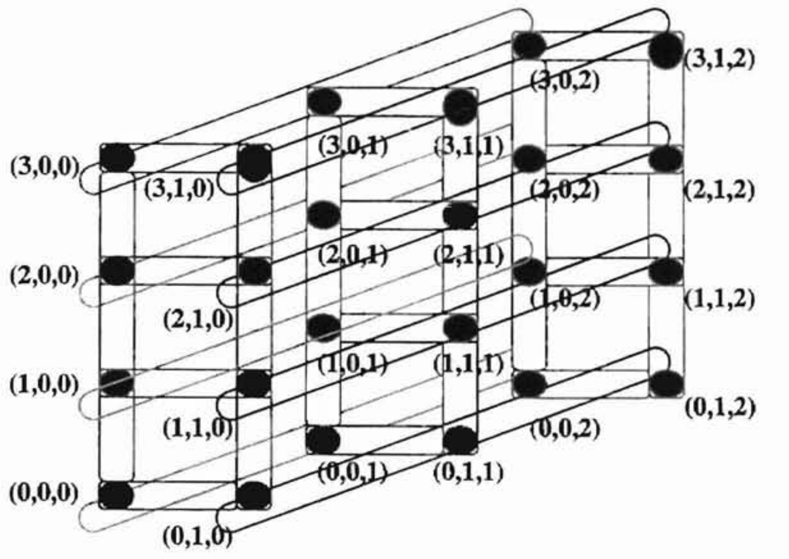

Figure 1. A 3D[4,2,3] utilizing the universal node-labeling scheme

3.1. 3DTIN Node Distribution

In this section we present a different interpretation of the dimensions universally defined in a 3DTIN. The 3DTIN can be defined as a triple 3D[P, Λ, Π], where, P represents the total number of regions in a plane, Λ represents the total number of lots in a region, and Π represents the total number of planes. A lot can be defined as a storage location. See figure 1 for illustration of a 3D[4, 2, 3]. The nodes are interconnected within three planes, each of which, consist of four regions. Each region consists of 2 lots. We use the symbols π, ρ, and λ to denote a node's specific plane, region, and lot labels respectively. The lots correspond to processor nodes. The maximum number of nodes in each plane is N=($\mathrm{P} * \Lambda$). The maximum node value a plane can hold is obtained by

$((\pi + 1) * (P * \Lambda - 1))$, where π is the plane identification number, P is the maximum number of regions in each plane, and Λ being the maximum number of lots (nodes) within each region. Just as the nodes in the binary n-cube are labeled from 0 to $2^n - 1$ with 2^n-1 being the maximum node value, this interpretation of the 3DTIN places an ordering on the nodes in it. The planes can take on labels from 0 to I, where I is an integer. The regions in each plane are labeled from 0 to $(P-1)$. Similarly, the lots in each region are labeled from 0 to $(\Lambda-1)$. It should be noted that although lots correspond to reserved locations of processor nodes, the label of the lot does not correspond to the label of the node. By translating the three previously defined dimensions of a 3DTIN to plane, regions, and lots, it now becomes apparent that a dimension can contain less than two nodes. As an example a 3D[1,2,1] has one region consisting of two lots, and 1 plane. One could argue that this is simply a ring consisting of two nodes; recall that one and two-dimensional torii are variations of the 3DTIN. Additionally, this translation suggests an arbitrary network size. The node-labeling algorithm is presented in the next section.

3.2. Node labeling Algorithm

When labeling the nodes in the 3DTIN, the number of lots per region and the total number of regions per plane should be defined precisely. Once specified, the total number of nodes per plane (N) is easily computed by multiplying the total number of regions per plane P by the number of lots per region Λ. The regions are labeled from 0 to $(P - 1)$ while the lots are labeled from 0 to $(\Lambda - 1)$.

The planes are labeled from 0 to (Π - 1). Having labeled the planes, regions, and lots, the labels of each node can be computed. With N being the total plane size, a node is identified by the plane labeled i, the region labeled j, and the lot labeled k, where $0 \leq i < \Pi$, $0 \leq j < P$, $0 \leq k < \Lambda$.

The node labeled x is computed in three simple calculations.

1) $x = k * P$
2) $x = x + j$
3) $x = x + (i * N)$

Using figure 1, to label the node in plane 0, region 3, lot position 1, simply calculate the following:

1) $x = k * P = 1 * 4 = \mathbf{4}$
2) $x = x + j = 4 + 3 = \mathbf{7}$
3) $x = x + (i * N) = 7 + (0 * 8) = \mathbf{7}$

Thus, *position*(3,1,0) translates to the node(7). In short to label the entire 3DTIN, the following algorithm is used,

```
for i = 0 to $\Pi$-1
  for j = 0 to P–1
    for k = 0 to $\Lambda$-1
      x = k * P
      x = x + j = x
      x = x + (i * N) = x
      assign label to node
    endfor
  endfor
endfor
```

Figure 2. Algorithm to label the nodes in a 3DTIN

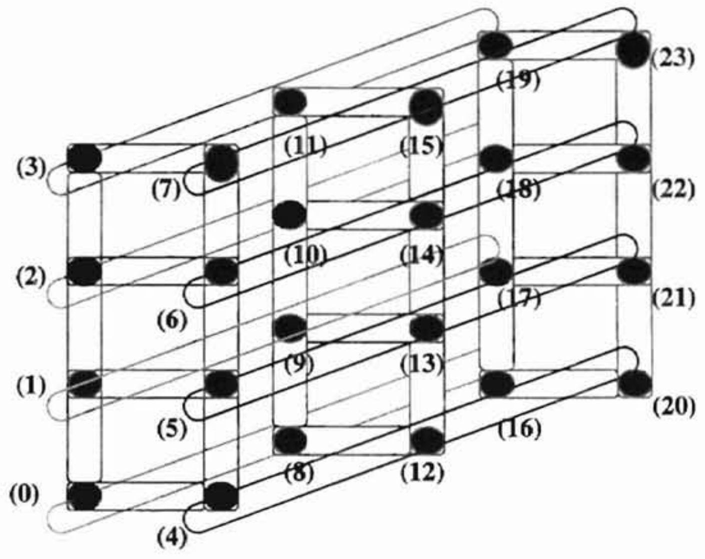

Figure 3. A 3D[4,2,3] utilizing the node-labeling scheme.

3.3 Node Location Algorithm

The 3DTIN offers an algorithm to locate a specific node within the topology. This algorithm allows one to identify the potential location of a node in a changing network in which network upgrades or network node additions are inevitable. Defining an integer X as the node label, the location of the node in terms of plane, region, and lot identification can be computed with the following addressing algorithm:

$A(X) =$

$= a[\, \rho, \lambda, \pi \,]$

$= a[\, X \,\%\, P, \, (X - (\pi \, * P * \Lambda)) \, / \, P, \; X \, / \, (P * \Lambda) \,]$, where "$(X / P * \Lambda)$) " computes the identification number of the plane containing the reserved location, "$X \% P$ " computes the relevant region, and "$(X - (\pi * P * \Lambda)) / P$ " computes the node's reserved lot (All divisions done are considered to be integer divisions). For example, to calculate the location for the node labeled 13 in our example, the following steps are taken.

1) Plane = $\pi = X / (P * \Lambda) = (13 / 8) = \mathbf{1}$

2) Region = $\rho = X \% P = 13 \% 4 = \mathbf{1}$

3) Lot = $\lambda = (X - (\pi * P * \Lambda)) / P = (13 - (1 * 8) / 4 = \mathbf{1}$

Node(13) resides in plane 1, region 1, at lot 1 . (figure 3.)

CHAPTER IV

3-Dimensional Torus Data Structure

In this thesis, we develop a new data structure named as 3DTDS. The three-dimensional torus data structure (3DTDS) is based entirely on the 3DTIN topology and the node-labeling scheme outlined in section 3. Like the 3DTIN, the 3DTDS is defined as a triple $3D[\mathrm{P}, \Lambda, \Pi]$, where Π represents the current number of planes of the 3DTDS, P represents the total number of regions in a plane, and Λ is the total number of lots in a region. Unlike the plane parameter in the 3DTIN, the third component Π in the 3DTDS is dynamic. More specifically, this parameter can increase or decrease throughout the use of the 3DTDS. A key is stored in a lot. In other words, a key is mapped into a lot. The mapping is defined in terms of planes, regions, and lots. It maps a key K into (ρ,λ,π) where π, ρ, and λ to denote a key's corresponding plane, region, and lot label respectively. We note that all keys must have an integer value. The total number of values (keys) a plane can contain, denoted by S, is defined as the plane size. A plane size S is equivalent to $\mathrm{P} * \Lambda$. Let $\Gamma = \{\pi_0, \pi_1, \ldots.\pi_i\}$ be the set of allocated planes in the 3DTDS where $0 \leq i \leq \Pi - 1$, thus $|\Gamma| = \Pi$. Let $x \in \pi_i$, x is the maximum element in π_i if x>a for all a resident in π_i. The upper bound of a plane π_i, denoted by $ub(\pi_i)$, is the largest possible value that plane π_i can contain. It is computed with the formula $(\pi + 1) * (\mathrm{P} * \Lambda) - 1$, where $\pi = \pi_i$ is the plane identification number, P is the total number of regions in each plane, and Λ being the total number of lots (nodes) within each region. The smallest possible value a plane can contain is termed the lower bound. The lower bound of a plane π_l, denoted by $lb(\pi_i)$, is equal to $((\pi_i) * (P * \Lambda))$.

4.1 Reserved Positioning

With the use of a 3DTIN-based structure and the node-labeling scheme, the location of any integer value can be easily determined. Like the nodes in the 3DTIN, each integer has its own reserved position. The approach which allows unique key values (integers) to be associated with a reserved location is termed *Reserved Positioning* (REPO). REPO is contingent on a structure that provides planes holding regions with each region consisting of storage locations called lots. There are two classifications of REPO - unconditional and conditional. Differences between the two classifications are seen in the plane allocation scheme used in alternative implementations of the 3DTDS.

In implementing a 3DTDS using the concept of unconditional REPO, the plane allocation scheme is as follows:

1) Read in a key *x*
2) Determine the location of its reserved position (ρ,λ,π) using the Node-Labeling Algorithm
3) Determine if plane π is resident (i.e. has been allocated) in the 3DTDS. IF plane π is resident, place *x* in its reserved position (ρ,λ,π) ELSE allocate space for plane π and place *x* in its reserved position (ρ,λ,π).

Thus in the event that a plane is allocated, we see that it is possible for lots to exist both after a key is seen and before it is seen.

Utilizing a conditional REPO, the plane allocation scheme is as follows:

1) Read in a key *x*
2) Determine the location of its reserved position (ρ,λ,π) using the Node-Labeling Algorithm
3) Determine if plane π is resident in the 3DTDS. IF plane π is resident allocate space for lot λ and place *x* in its reserved position (ρ,λ,π) ELSE use an identifier to indicate the virtual allocation of plane π and allocate space for the lot λ. Place *x* in its reserved position (ρ,λ,π).

In general, the number of regions P is static when using this approach. A better explanation is presented in section 5. Elaborating more on the two approaches, given a plane size S=10 and a key=1; utilizing unconditional REPO, once the plane holding the reserved lot for key one is created, nine other lots are automatically created. Other keys processed in the future, which belong to this specific plane, will simply be assigned to their reserve location. Applying the conditional REPO approach, only the lot for value one is created. If the plane is not present an identifier (i.e. flag) is used to indicate the allocation of the plane. Applications of both types of REPO are used in different implementations of the 3DTDS. Further explanation is presented in the section 5.

REPO is a modification of the currently known hashing methods. It utilizes one major addressing function and the structure, in relation to the universally known hash table, can be described best as a group of tables. While REPO implies that distinct keys map to distinct locations, hashing allows distinct keys to hash to the same location. Hashing also considers duplicate keys as "just another key", meaning an identical key is placed in a distinct location as if it were a different value, possibly causing a worst case O(N) search time, where N is the number of values in a hash table. In addition, hashing only supports an equality search, whereas REPO efficiently supports at least three different search/query types: equality queries (i.e. find an employee with a specific identification number), range queries (i.e. find all houses costing \$100,000 - \$500,000), and min-max queries (i.e. find the least expensive house and the most the most expensive house).

The address function associated with REPO must be complemented with a 3DTDS. The address function A(key) computes the reserved position for a unique key. In short, the address function is

$$\begin{aligned} A(key) &= \\ &= A[\,\rho, \lambda, \pi\,] \\ &= A[\,key \,\%\, P,\ (\,key - (\pi * P * \Lambda))\,/\,P,\ key\,/\,(P * \Lambda)\,], \end{aligned}$$

where "$(key/\ P * \Lambda))$ " computes the identification number of the plane containing the reserved location, "$key \,\%\, P$" computes the relevant region, and "$(\,key - (\pi * P * \Lambda)\,)\,/\,P$ " computes the node's reserved lot. This address function is identical to the node-location algorithm outlined in section 3.3. The 3DTIN topology directly impacts the current definition, design and implementation of the 3DTDS. The node-labeling algorithm creates a means for inserting into the 3DTDS. Additionally, the 3DTIN node-location algorithm allows search time to be at a minimum.

The use of REPO is implicit with utilizing the 3DTDS. The use of REPO is motivated by the ability to design and implement a structure, which after N number of key insertions, the N keys would be organized in such an order that sorting is minimum or unnecessary. Practical applications of REPO can be found in the area of databases, where efficient retrieval and sorting of significant amounts of data is continuously needed.

CHAPTER V

3DTDS Implementation

In this section, two implementation methods of the 3DTDS are discussed. Before a detailed description of the 3DTDS is presented, it is important to note that although the number of regions and lots in the structure are predetermined, the allocation of planes is dynamic. More importantly if the region or lot sizes were not static, labels for all nodes would need to be recomputed after each new region or lot size change to account for the change in plane size. This approach is undesirable, as it would yield an inefficient use of processor time.

5.1 Matrix/Array

The matrix implementation utilizes unconditional REPO. This method is a straightforward approach. Figure 4.a depicts a 3DTDS implemented as a matrix. It consists of two planes having nine regions. Each region consists of four lots. The columns correspond to regions while the rows correspond to lots. Though, it is possible to use a one-dimensional plane, a matrix is used for added clarity. In using a one-dimensional plane, the total number of regions P in any given plane is equal to one (see figure 4.c). Additionally, the region label is equal to the plane label. Referring to figure 4.a, the plane size is equal to 36. Taking a key with a value of 21, we are able to compute the address of key 21 in three steps:

1) Plane => (21 / 36) = 0
2) Region => 21 % 9 = 3
3) Lot => 21- (0 * 9 * 4) / 9 = 2

The value 21 is mapped to region 3, lot 2 in plane 0. Referring to figure 4.a, plane 0 has a lower bound of 0 and an upper bound of 35 and plane 1 has a lower bound of 36 and an upper bound of 71. The maximal element of plane 0 is 35 while 69 is the maximal element corresponding to plane 1. By allocating space for plane 4, reserved lots for values 144 – 179 become available. As one might conclude, unconditional REPO works best with values located in close proximity. In addition the region and lot sizes must be chosen carefully for storage utilization. For instance, if an input of size three were to be processed, one would not want a plane size of one hundred. Referring to figure 4.a, the actual key values are placed in the reserved lots for explanatory purposes. Ordinarily, after the allocation and initialization of a plane, a value of one is placed in the lot designating that the key is present. In the event of a duplicate key, the lot value is incremented by one. As the planes are allocated, they are placed in a binary tree (termed the *plane tree*) for efficient access. For instance the planes in figure 4.a would be placed in binary tree. See figure 4.b for a detailed illustration. In figure 4.b, each node in the plane tree corresponds to an allocated plane. The number in each node denotes the plane id or label. Referring to figure 4.b planes 0, 1, 4, and 100 have been allocated. As previously stated the unconditional REPO scheme is appropriate for keys having values in close proximity of one another. This locality constraint avoids the inefficient use of memory. If the constraint is lifted, the 3DTDS wastes more space than it uses. However, its performance is extremely efficient. Aside from efficient performance, this implementation is fairly simple to incorporate. Let Γ be the set of planes in the plane tree (by definition Γ is the set of planes in the 3DTDS which is equivalent to the set of planes in the plane tree), so it takes O(log(|Γ|)) time to locate a plane. In general, |Γ| is much less than the input-size N; however, |Γ| is equal to N in the event that for each input key a plane is allocated. This is a worst case scenario that is highly unlikely. Furthermore, regarding sorting, the key values in each plane are indirectly sorted due to REPO. The obvious disadvantage is that if key values are sparse, an inefficient use of space is inevitable. To limit the amount of wasted space, a bit-vector based 3DTDS can be used.

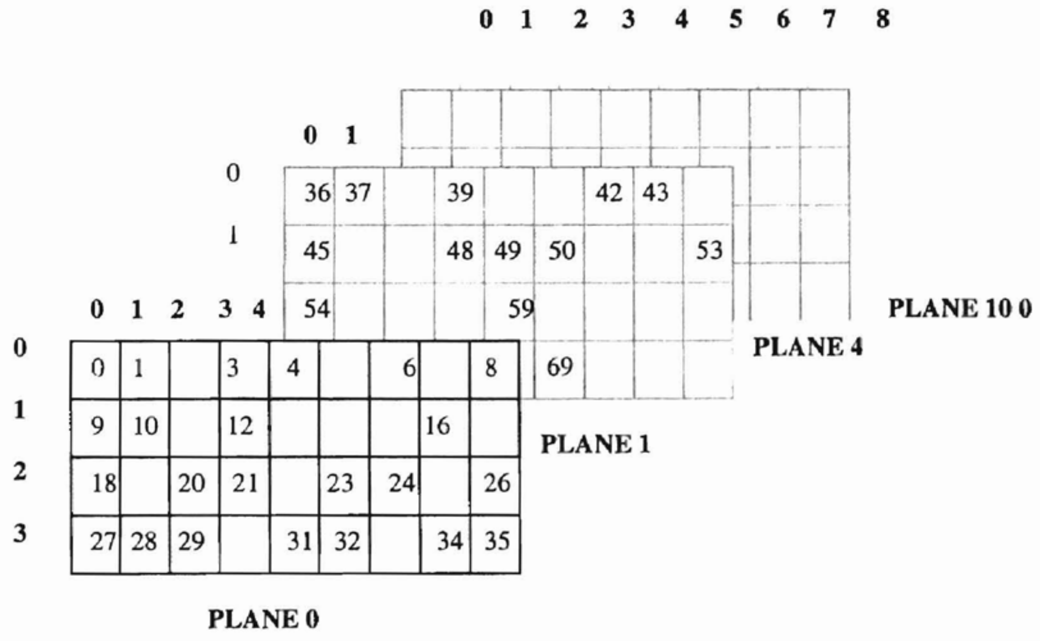

Figure 4.a. A matrix implementation of a 3DTDS.

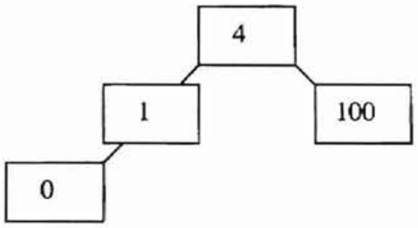

Figure 4.b. The planes are kept in a binary tree structure called a plane tree for quick access

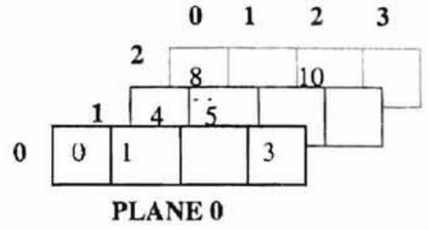

Figure 4.c. A 1-dimensional implementation of a 3DTDS.

The bit vector implementation is similar to that of the matrix implementation. Instead of using arrays of integers, arrays of bits initialized to zero are used. If a key value is processed, the bit in its reserved lot position is flipped to a one. The obvious advantage to this modification is that it uses less than 4 percent of the total space used in the matrix implementation; however, it is impossible to retain duplicate key values and/or satellite data. The matrix implementation presents a superficial explanation of the 3DTDS. In terms of wasted space, this implementation behaves badly. In the next section a closer look at the 3DTDS is taken via an alternative implementation termed the Chain-Tree, which utilizes the conditional REPO approach.

5.2 Chain-Tree

The chain tree implementation uses a conditional REPO approach. It utilizes a method similar to chaining (a collision resolution scheme used in hashing). This implementation consists of a combination of popular data structures used in the field of computer science today; namely, the splay tree, binary tree, double ended queue, and array structures. It has four primary components: a plane tree, lot tree, plane template, and plane list. As aforementioned, the previous implementation (matrix) presented a great possibility that several planes could be created due to sparse key values, thus yielding large amounts of wasted space. The chain-tree implementation initially allocates space for one plane, which is used as a template for planes created in the future. This initially allocated plane is termed the plane template. Each lot in the plane template is an actual pointer to a binary tree. The binary tree at each lot is termed a lot tree. By definition lots are contained within regions, so the number of regions within the plane template is arbitrary. Each lot tree contains values/keys, which map to that specific lot λ, but may map to a different plane π. The concept is similar to chaining; however, instead of a linked list, a binary tree is used. Initially, the value of each lot within the plane template is initialized to null. Once a key is processed, its reserved lot location is determined and space (i.e. a node) for the value is allocated. The value is then inserted into the specific lot tree located at its reserved lot. After an insert, an insertion into a separate tree, termed the plane tree, is at times necessary. This insertion signifies the allocation of an entire plane. Figure 5 shows an example of a 3DTDS structure not fully connected.

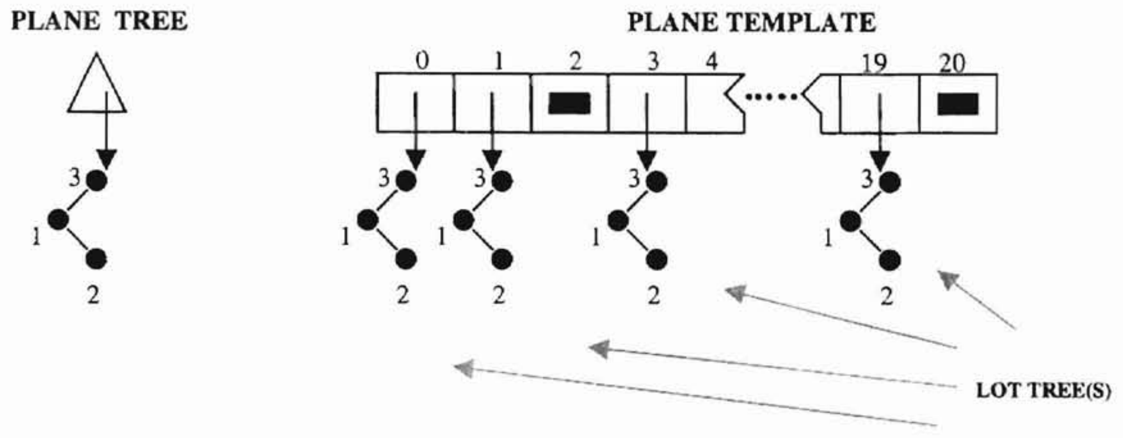

Figure 5. A chain-tree implementation of a 3DTDS

The plane tree consists of all allocated planes currently present as a result of all inserted keys. We denote these plane allocations as being virtual. In Figure 5, planes 1, 2, and 3 have been virtually allocated. The plane tree is constructed as a splay tree. This plane allocation scheme aids in space preservation. Space for one plane is represented by one lot-sized node instead of an entire plane.

5.2.1 Plane Tree Nodes and Plane Template Nodes

Special attention should be given to the structure of nodes within the plane tree and plane template. First, both the plane tree node and plane template node consists of at least four pointers. See figures 6.a and 6.b for a detailed illustration. The plane tree node has a LEFT and RIGHT pointer to aid in implementing a binary tree. Each plane tree node allocated denotes a plane. The chain tree implementation utilizes conditional REPO thus all lots in a newly created plane are not allocated. The FRONT and REAR pointers of the plane tree node are used to maintain a list of existing keys present in a plane. This list is termed the plane list. In Figure 7, three plane lists are depicted, the lists belonging to plane 1, 2, and 3. It is noted that the REAR pointer of the plane tree node is not pictured.

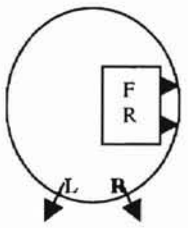

Figure 6.a. A plane tree node

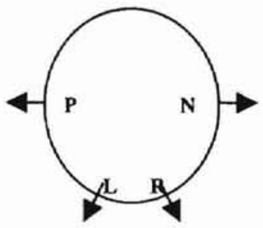

Figure 6.b. A plane template node

The FRONT and REAR pointers contain the address of the first and last key values in a plane respectively. The plane tree node can contain an optional PARENT pointer, which aids in splaying operations. Further explanation of the splay operations is presented in the section 5.2.4. Similar to the plane tree node, the plane template node contains LEFT and RIGHT pointers. Given a key X in a plane list. A PREVIOUS pointer is used to point to the first

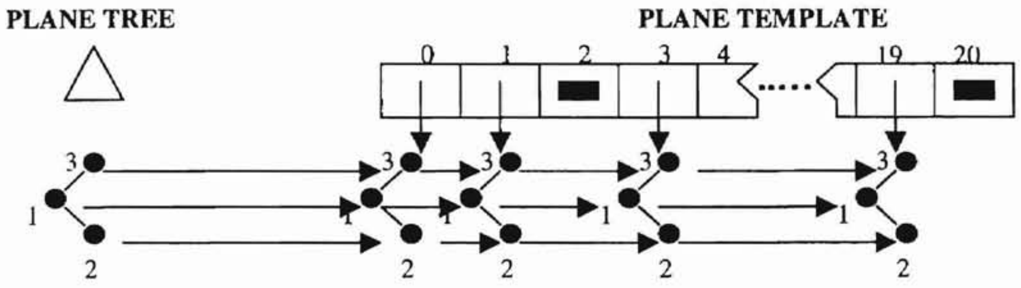

Figure 7. A chain-tree implementation of a 3DTDS

value smaller than X and a NEXT pointer is used to point to the first value larger than X. In the following sections, a description of the fundamental operations associated with the 3DTDS is presented. They are as follows:

1. Disconnect
2. Connect
3. Insert
4. Delete
5. FindMax and FindMin
6. Find
7. Find Range
8. FindNext and FindPrevious
9. Print_SortMin and Print_SortMax
10. Sort (optional)

We use the terms routine and procedure interchangeably when describing the operations. Additionally we dedicate chapter six to the discussion of the optional *Sort* procedure as it calls for an in-depth explanation.

5.2.2 Disconnect

This routine is used to disconnect a node from its plane list. Let X, Y, and Z be three connected nodes in a plane list. If *Disconnect* is executed on node Y, its PREVIOUS and NEXT pointers are set to NULL. The NEXT pointer of X is

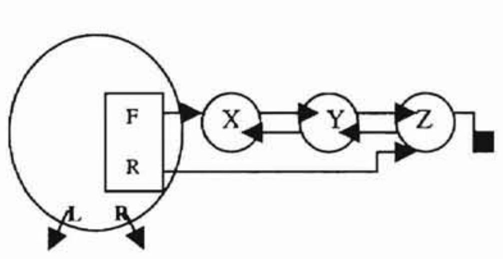

Figure 8.a. Plane list before *Disconnect* operation is executed

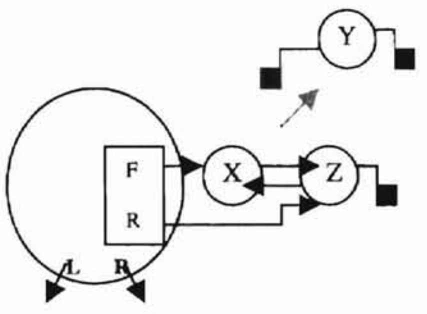

Figure 8.b. Plane list after *Disconnect* operation is executed.

assigned the value of Z, while the PREVIOUS pointer of Z is assigned the value of X. *Disconnect* runs in O(1) time.

5.2.3 Connect

This routine is the counterpart of *Disconnect*. It is used to connect a node to a plane list. Let X and Z, be two connected nodes in a plane list. If *Connect* is executed on a node Y, its PREVIOUS pointer is assigned the address of X and its NEXT pointer is assigned the value of Z. The NEXT pointer of X is assigned the value of Y, while the PREVIOUS pointer of Z is assigned the value of Y. *Connect* runs in O(1) time.

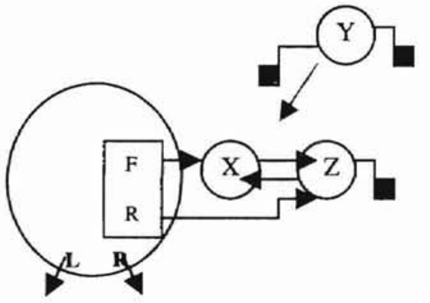

Figure 9.a. Plane list before *Connect* operation is executed.

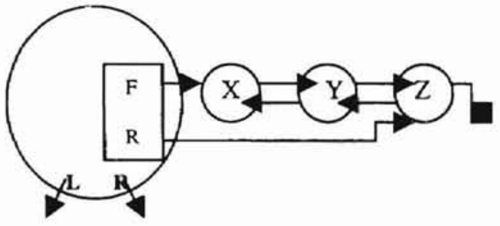

Figure 9.b. Plane list after *Connect* is executed.

5.2.4 Insert

The insertion operation is a three step procedure. The plane X_π, region X_ρ, and lot X_λ of a key X are computed. X is inserted into the plane template at location (ρ,λ), where ρ and λ is X's associated region and lot. A TEMP pointer is assigned the address of X. The plane tree then is searched to determine if plane X_π is present. If plane X_π is not present, it is inserted into the plane tree and the

FRONT and REAR pointers of the plane tree node is assigned the address of TEMP. In contrast, if plane X_π is present then either the FRONT or REAR pointer of the X_π is traversed to identify the key already present in the plane that is greater than or less than X respectively. The algorithm determines which pointer to access by computing the midpoint M of the plane size S. If key X belongs in one of the first M lots, the FRONT pointer is used; otherwise the REAR pointer is used. Once the appropriate position is located, the node is inserted into the binary tree. *Connect* is then executed to reestablish the plane list. See Figures 9.a and 9.b. Figures 10.a – 10.h show the results of inserting seven keys in a 3DTDS. The plane template is an array. A two-dimensional plane can be used; however a one-dimensional plane is used for simplicity. In the example Λ=6 and P=1, thus N=6, where N maximum possible nodes in any given plane. The gray nodes denote newly inserted nodes. The value within each node denotes the plane value while the value located to the lower right of the node denotes the actual key value.

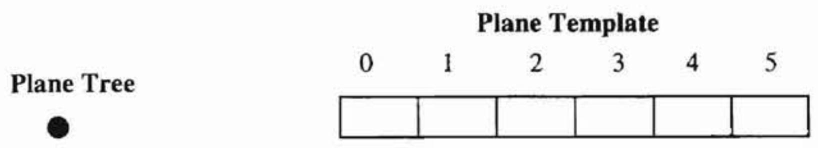

Figure 10.a. Initial state of Plane Tree and Plane Template: Both are EMPTY

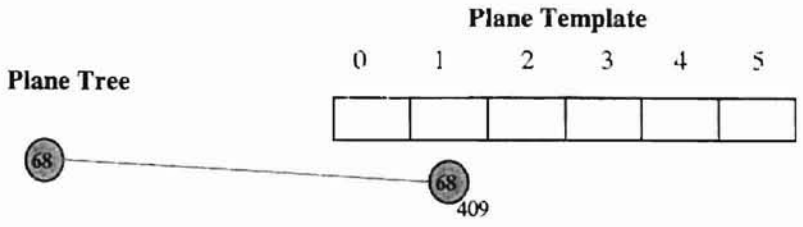

Figure 10.b. Inserting key 409.
$\pi(409) = 409/6 =$ **68**
$\lambda(409) = (409 - (68 * 6))/1 =$ **1**
$\rho(409) = 409\%1 =$ **0**

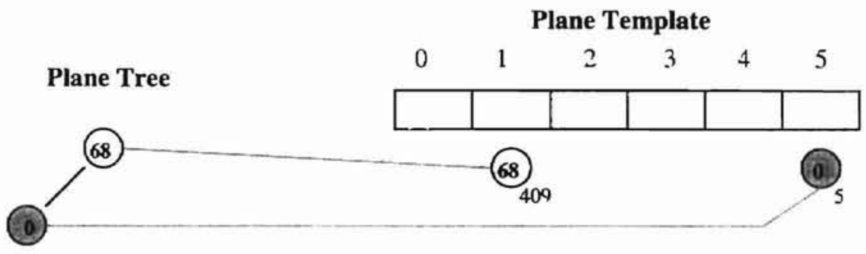

Figure 10.c. Inserting key 5
$P(5) = 5/6 =$ **0**
$L(5) = (5 - (0 * 6))/1 =$ **5**
$R(5) = 5\%1 =$ **0**

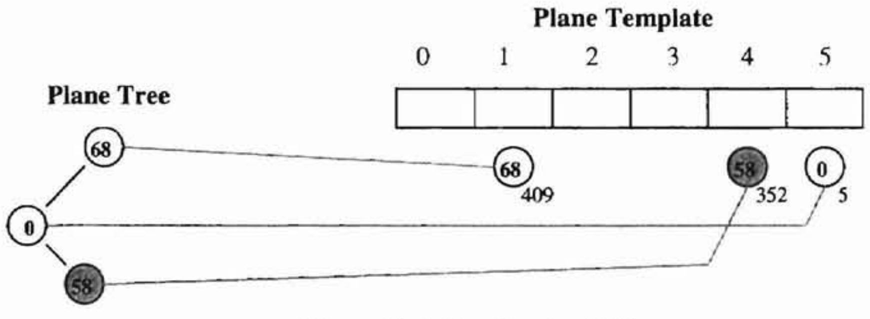

Figure 10.d. Inserting key 352
$\pi(352) = 352/6 =$ **58**
$\lambda(352) = (352 - (58 * 6))/1 =$ **4**
$\rho(352) = 352\%1 =$ **0**

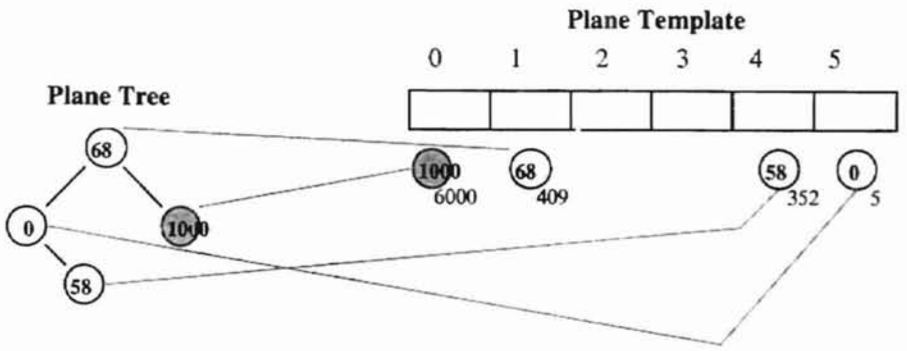

Figure 10.e. Inserting key 6000
$\pi(6000) = 6000/6 =$ **1000**
$\lambda(6000) = (6000 - (1000 * 6))/1 =$ **0**
$\rho(6000) = 6000\%1 =$ **0**

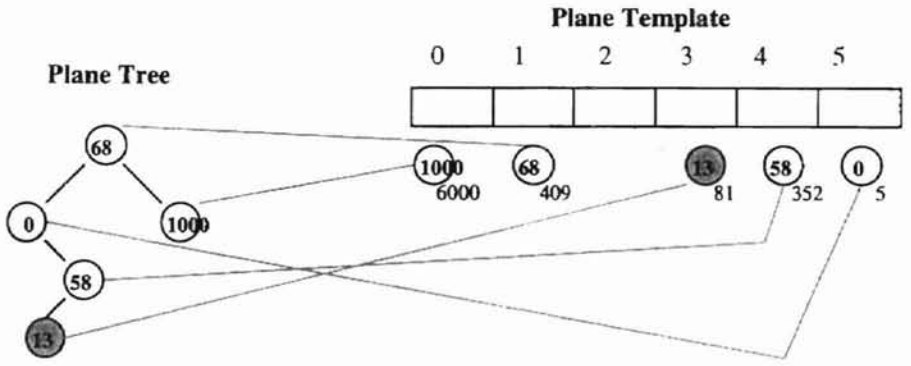

Figure 10.f. Inserting key 81
$\pi(81) = 81/6 =$ **13**
$\lambda(81) = (81 - (13 * 6))/1 =$ **3**
$\rho(81) = 81\%1 =$ **0**

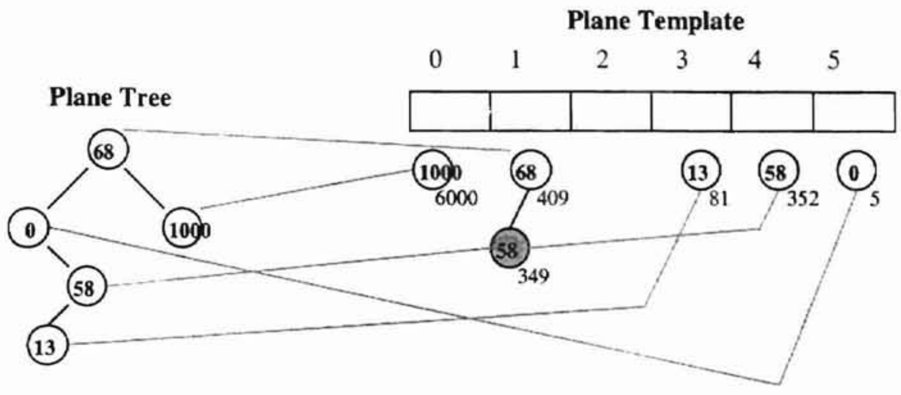

Figure 10.g. Inserting key 349
$\pi(349) = 349/6 = \mathbf{58}$
$\lambda(349) = (349 - (58 * 6))/1 = \mathbf{1}$
$\rho(349) = 349\%1 = \mathbf{0}$

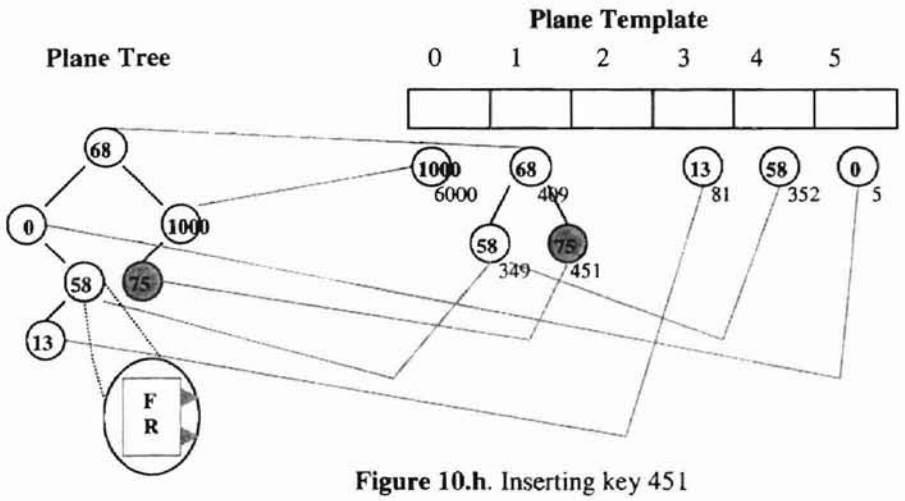

Figure 10.h. Inserting key 451
$\pi(451) = 451/6 = \mathbf{75}$
$\lambda(451) = (451 - (75 * 6))/1 = \mathbf{1}$
$\rho(451) = 451\%1 = \mathbf{0}$

As a result of checking the plane tree for a value after each insertion, the insertion routine is charged with the responsibility of attempting to keep the search on the plane tree at a minimum. This task is accomplished by splaying the plane tree at predefined intervals. The interval is denoted by the splay parameter (float value). The splay parameter notifies the routine when to execute a splay operation. The parameter is based on the depth of the newly inserted plane tree node and the ideal height of the plane tree. Let the splay parameter = k. If the newly inserted node is k times the ideal height of the plane tree, a splay operation on the inserted node is executed [3]. To compute the ideal height, the logarithmic value of the total number of nodes in the plane tree is computed. This implies that a count of planes in the plane tree is kept. Formally, a splay tree is used for primarily two reasons. The first reason is to allow a minimum search on an accessed node, as past research has shown that an accessed node will more than likely be accessed in the near future [3]. Prevention of the worst case scenario, a sequence of bad accesses, is the second reason [3]. In addition as a by-product of splaying, the worst case search time of O(N) on a binary tree becomes almost non-existent. Primarily, the reason splaying is used with the 3DTDS is for the latter reason. The first reason does not apply because generally when dealing with an input of random values, no one node will be accessed increasing more than another. However, it is extremely important that the plane tree be balanced to some level, to avoid a worst case search time. In summary, it takes $O(\log(|\Gamma|))$ to insert a node into the plane template and $O(\log(|\Gamma|))$ to search the plane tree. Having a plane size equal to S, it takes $O(^{S}/_{2}) \approx O(S)$ to identify the position for the newly created plane template node.

5.2.5 Delete

The delete algorithm is similar to that of the binary tree. Once the location (ρ,λ) in the plane template has been identified, traverse the lot tree for the desired node Y. Once Y is located *Disconnect* is immediately executed. There are three main cases necessary to observe when deleting a node from a binary tree. If the node Y has no children, the node is deleted immediately. In the event that Y has one child, the LEFT or RIGHT pointer of the parent of Y (depending on the scenario) is assigned the value Y's child (see figure 11).

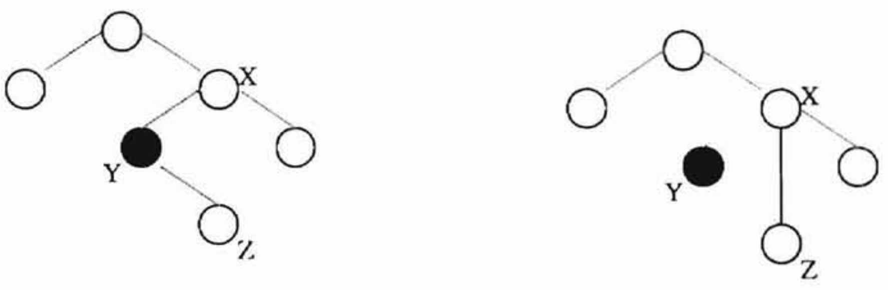

Figure 11. Deleting node Y from a binary tree. Node Y has one child. The dotted lines denote the disconnected links, while the solid black line denotes the new established link. *Disconnect* and *Connect* operations are not shown.

If Y has two children, the content of Y is replaced with the contents of the value of the smallest element in the right subtree of Y, denoted as Z. Node Y now has the value of Z. *Connect* is then performed on Y. After minor pointer re-adjustment with the parent of Y, node Z is immediately deleted (see figure 12). In the event that all keys from a plane list have been deleted, the plane node corresponding to the plane list is deleted. A lazy deletion scheme (tagging a node as being deleted instead of actually de-allocating its space) could be used if it is thought that a key from that plane might be processed in the future. Unfortunately, if a lazy deletion scheme is used search time on the plane tree could become slow if a large number of nodes were tagged as being deleted. For instance if a plane tree consisted of one hundred nodes and eighty were tagged as being deleted, the time to search would be extremely slowed due to the deleted nodes. It is cost-effective to delete the plane tree node immediately, given that each time a key is processed, the plane tree is searched.

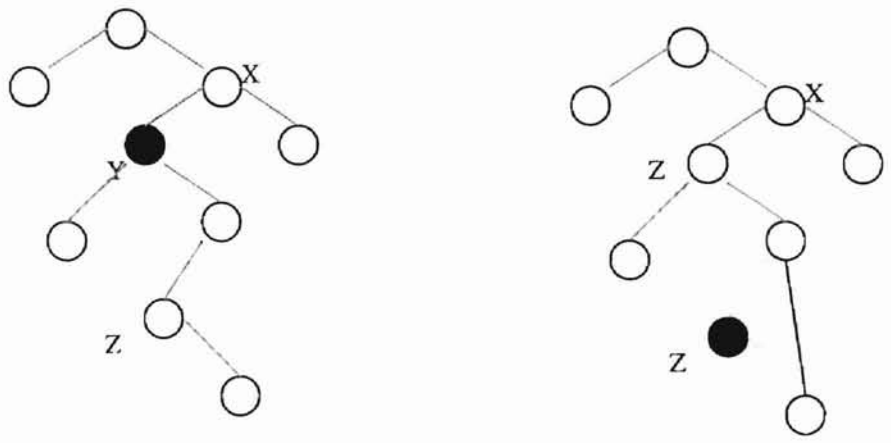

Figure 12. Deleting node Y from a binary tree. Node Y has two children. The dotted lines denote the disconnected links, while the solid black line denotes the new established link. *Disconnect* and *Connect* operations are not shown.

5.2.6 FindMax and FindMin

These operations return the value of the largest and smallest element in the 3DTDS. To perform a *FindMax*, start at the root of the plane tree and traverse right until the node holding the largest value is obtained. Once the node is identified, the largest value in the 3DTDS is obtained by accessing the rear pointer located in the node. The *FindMin* is the same, except the traversal is done on the left side of the tree. This operation has an average case and worst case time complexity of $O(\log(|\Gamma|))$.

5.2.7 Find

The *Find* routine is done by first computing the region ρ and lot λ location of the value being searched. Once computed, a binary search is done on the tree located at address (ρ,λ). In the event that the key is found, its value is returned; otherwise, NULL is returned. The average case time complexity is $O(\log(|\Gamma|))$. In general the size of a tree in the plane template will be less than or equal to the size of the plane tree. The worst case time for the search is $O(|\Gamma|)$.

5.2.8 Find Range

This operation returns an array of all keys between a start value A and an end value B inclusively. The idea behind this routine is to utilize the plane tree, which contains $|\Gamma|$ nodes where $|\Gamma|$ is generally much less than N, thus minimizing the search time. The keys between the values A and B inclusively, reside in planes between A and B's planes. All values in the plane tree meeting the criteria are evaluated. Evaluation includes confirming that the value obtained in the plane tree is valid. Once validation is complete a traversal of the FRONT pointer of the node is done to obtain the keys within the specified range. The algorithm first computes the planes for values A and B. Let π_A and π_B denote the plane values for A and B respectively. A level order traversal is then done on the plane tree. A queue is used to perform the level order traversal. Each node in the queue is evaluated only when it arrives at the queue's front end. The front node is denoted as F and π_F is the plane value of F. This algorithm presents three initial cases.

1) In the event that π_F is less than or equal to π_A and the RIGHT pointer of F is not equal to NULL, enqueue the right child of F.

2) In the event that π_F is greater than or equal to π_B and the LEFT pointer of F is not equal to NULL, enqueue the left child of F.

3) If case one and two fails the right child and left child of F are enqueued.

If π_F is between π_A and π_B, the FRONT pointer of F is traversed to obtain values present in that plane. The values are inserted into an array. In the event that the front of the queue is equal to NULL, the routine ends and the array of values are returned. Figure 13 depicts a plane tree. Figure 14 depicts the plane tree nodes evaluated by *FindRange* (gray nodes). In this example $\pi_A = 55$ and $\pi_B = 500$.

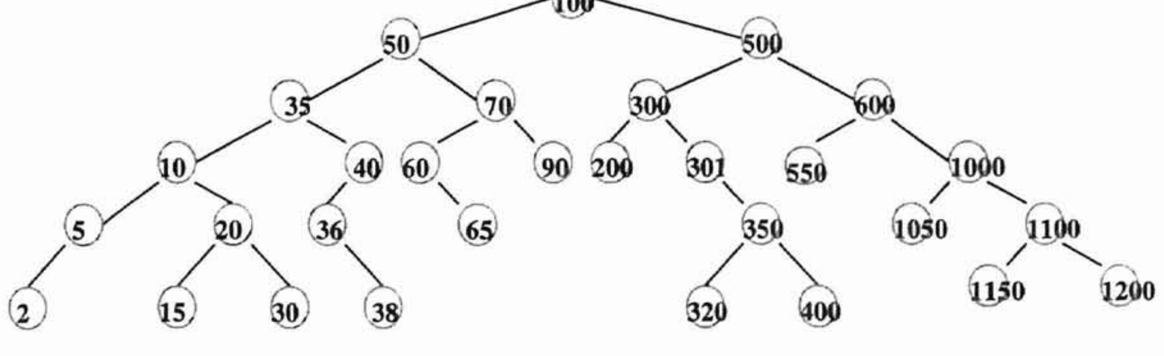

Figure 13. A Plane tree

The algorithm takes the optimal approach to this problem. It attempts to evaluate only the nodes necessary to solve the problem. It is clear that this strategy eliminates the processing of arbitrary nodes. The run time complexity of this routine is O(|Γ|), given that a range consisting of all plane tree values could be processed at any given time.

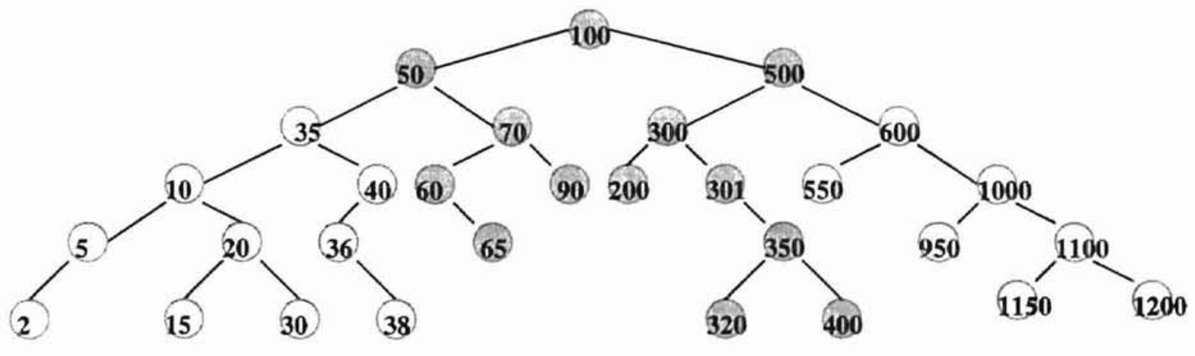

Figure 14. Requesting values in the range of A to B, *FindRange* identifies all planes between A and B's planes inclusively. The gray nodes are evaluated by the algorithm.

5.2.9 FindNext and FindPrevious

These routines return the first value greater than and less than a specific key in a plane. NULL is returned if the key is not found, the previous value is not present, or if the next value is not present. *FindNext* is done by first computing the region ρ and lot λ location of the value being searched. Once computed, a binary search is done on the tree located at address (ρ,λ). In the event that the key is found, the value pointed to by the NEXT pointer is returned. *FindPrevious* is done the same way, except that the PREVIOUS pointer is evaluated. The average case time complexity is O(log(|Γ|)).

5.2.10 SortMin and SortMax

These routines are not sorting operations in the traditional sense. As previously mentioned, the central motivation in using REPO and the 3DTDS is to design and implement a structure, which after N number of key insertions, the N keys would be organized in such an order that sorting is unnecessary. The insertion operation allows the keys in each plane to be sorted; therefore to output all values in ascending order an in-order traversal starting on the left side of the plane tree is done. This is done for the *SortMin* routine. Upon reaching a node in the plane tree, its FRONT pointer is traversed to output the sorted values held in that plane. *SortMax* is the same except an inorder traversal is done to the right side of the plane tree and the REAR pointer is traversed.

CHAPTER VI

Chain-Sort Algorithm

The chain-sort algorithm is motivated by the 3DTDS. It sorts keys residing in a 3DTDS-like structure. This algorithm sorts keys without comparing or swapping values. Furthermore, it is stable. This algorithm can be used as an optional routine in the chain-tree implementation of the 3DTDS. More precisely, rather than connecting all nodes to the plane list during the *Insert* operation, the chain-sort algorithm can be called at user defined intervals to perform that action. We remind the reader that as a byproduct of the connected nodes in a plane list, the keys are sorted. The chain-sort algorithm links all nodes together as if they were connected during *Insert*. It offers an alternative approach to the *Insert* operation. By electing to exploit this approach, the node connection time during the execution of an *Insert* is reduced considerably. *Insert* would therefore only be responsible for insertions into the plane tree and plane template, not the plane list, which has a worst case $O(^S/_2) \approx O(S)$ insertion time. If this optional approach is used, special attention must be given to operations such as *FindNext*, *FindPrevious, FindMax, FindRange,* and *FindMin.* The 3DTDS may not return the correct result when the procedures mentioned above are called due to newly inserted nodes that have not been connected to the plane list. For example, applying the alternative approach, if a node K is inserted after the chain-sort procedure has been called, K will not be connected to the plane list. Node K could possibly be the minimum or maximum value in the 3DTDS. If it is not connected to the rest of the nodes in the plane list and either *FindMin* or *FindMax* is called, the incorrect value will be returned instead of the correct value, the value of K. The execution of the chain-sort algorithm is recommended if new insertions have succeeded the latest chain-sort call.

When initiating the *Sort* operation, the plane template is traversed. Each lot tree in the plane template is evaluated. Formally a plane template is either a one-dimensional or two-dimensional structure of size S, where $S=P*\Lambda$. Each lot within the plane template is traversed in a row major order. Once a lot is accessed, the lot-tree is evaluated. Evaluation consists of accessing each node in a lot-tree(λ) or lot-tree(ρ,λ) when using a two-dimensional plane, determining its plane value and once determined, connecting the accessed node to its corresponding plane tree node located in the plane tree. A modification of the plane template node is important to the efficient execution of this algorithm. The address of a node's corresponding plane tree node is retained in every node resident in a lot tree. In contrast by not retaining the plane tree node values, a total of N (input size) searches on the plane tree would be necessary to locate each key's corresponding plane during chain-sort. Such continuous searching has proved costly. Essentially, keys with identical plane values are connected together producing a list of ascending key values per plane, more precisely creating the plane list in the 3DTDS. See figures 15.a – 15.i for explanation. In the example $\Lambda=6$, $P=1$, and plane size $S=6$ (Appendix A. depicts a plane template with $P=2$). The gray nodes denote newly inserted nodes. The value within each node denotes the plane value while the value located to the lower right of the node denotes the actual key value. After connecting all nodes, an in-order traversal of the plane tree is necessary to output the sorted keys.

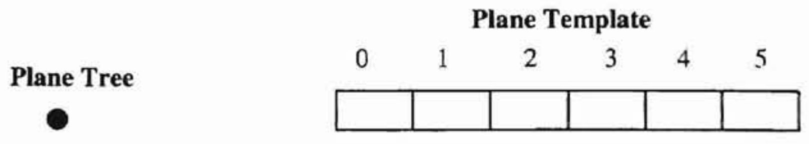

Figure 15.a. Initial state of Plane Tree and Plane Template: Both are EMPTY

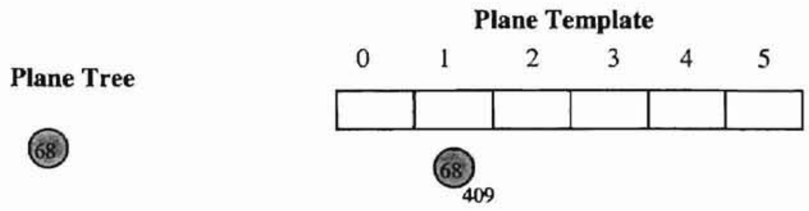

Figure 15.b. Inserting key 409
π(409) = 409/6 = **68**
λ(409) = (409 – (68 * 6))/1= **1**
ρ(409) = 409%1 = **0**

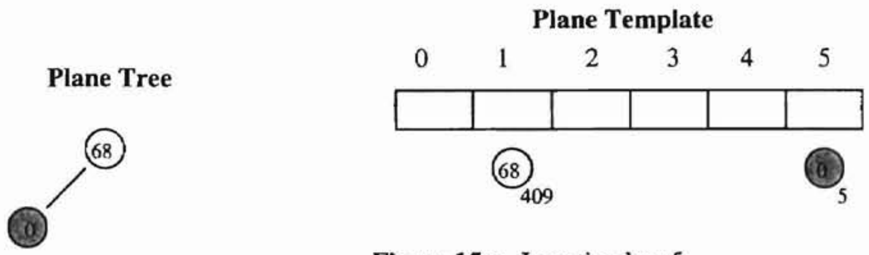

Figure 15.c. Inserting key 5
π(5) = 5/6 = **0**
λ(5) = (5 – (0 * 6))/1= **5**
ρ(5) = 5%1 = **0**

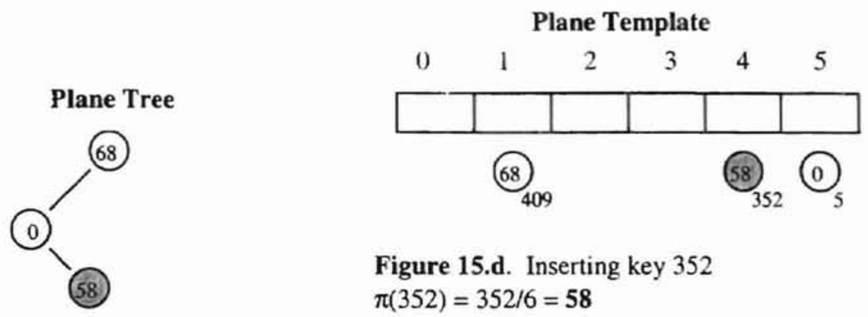

Figure 15.d. Inserting key 352
$\pi(352) = 352/6 = \mathbf{58}$
$\lambda(352) = (352 - (58 * 6))/1 = \mathbf{4}$
$\rho(352) = 352\%1 = \mathbf{0}$

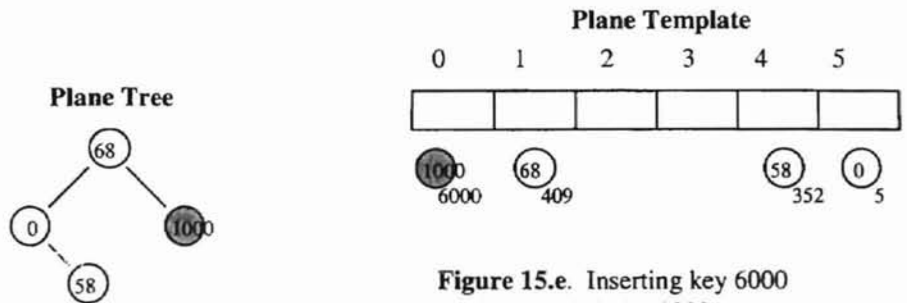

Figure 15.e. Inserting key 6000
$\pi(6000) = 6000/6 = \mathbf{1000}$
$\lambda(6000) = (6000 - (1000 * 6))/1 = \mathbf{0}$
$\rho(6000) = 6000\%1 = \mathbf{0}$

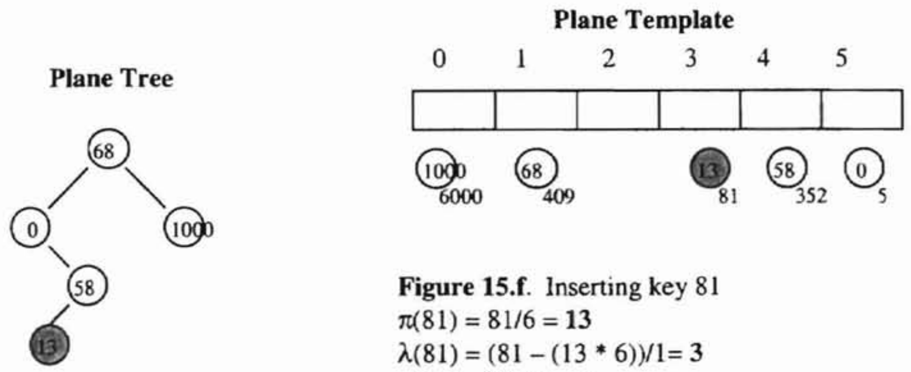

Figure 15.f. Inserting key 81
$\pi(81) = 81/6 = \mathbf{13}$
$\lambda(81) = (81 - (13 * 6))/1 = \mathbf{3}$
$\rho(81) = 81\%1 = \mathbf{0}$

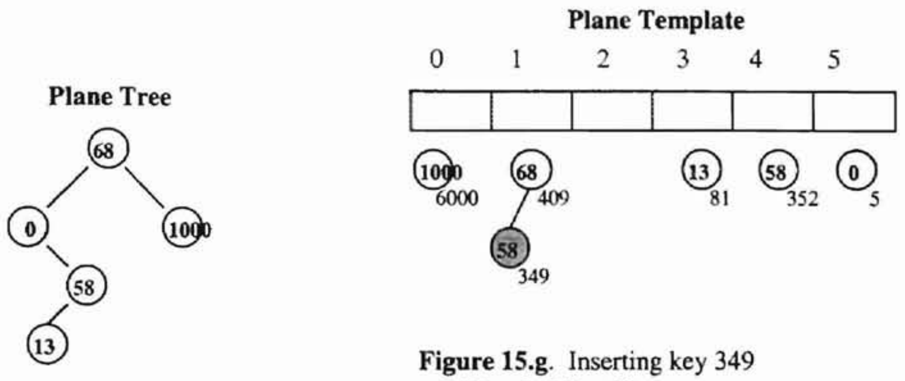

Figure 15.g. Inserting key 349
$\pi(349) = 349/6 = \mathbf{58}$
$\lambda(349) = (349 - (58 * 6))/1 = \mathbf{1}$
$\rho(349) = 349\%1 = \mathbf{0}$

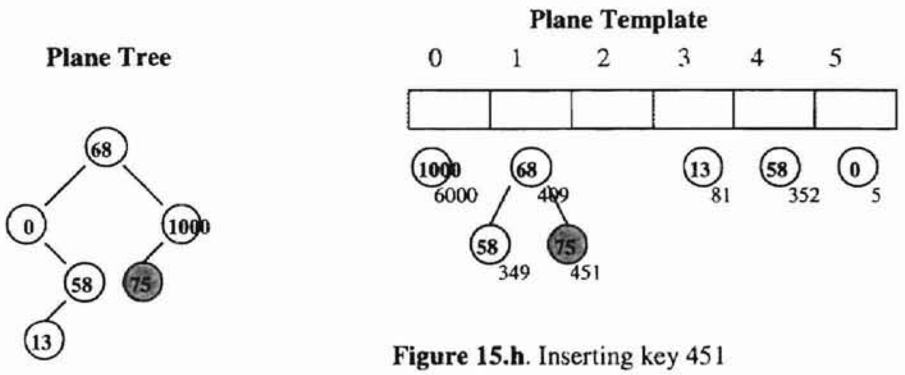

Figure 15.h. Inserting key 451
$\pi(451) = 451/6 = \mathbf{75}$
$\lambda(451) = (451 - (75 * 6))/1 = \mathbf{1}$
$\rho(451) = 451\%1 = \mathbf{0}$

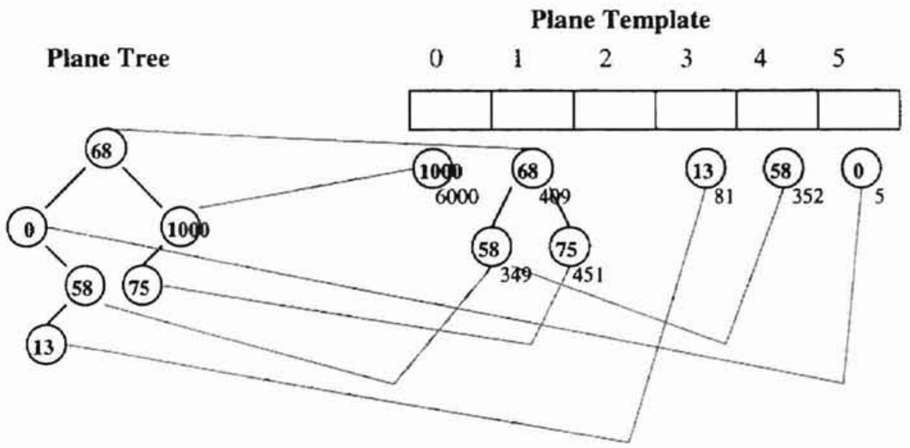

Figure 15.i. Sorted Numbers: 5, 81, 349, 352, 409, 451, 6000

The time taken to insert and output keys is not considered in the chain-sort analysis. The analysis is presented in section 7. As previously mentioned, this algorithm sorts keys *resident* in a 3DTDS-like structure. Using chain-sort as a possible stand-alone sorting algorithm has been considered and researched. Current research yields one primary concern. As with the heapsort algorithm, the time to build the heap is computed in the analysis. Similarly, the cost to construct the 3DTDS must be computed in the analysis. The primary concern lies in the continuous search on the plane tree during each *Insert* call. Although efficient, the search considerably increases the time required to execute a stand-alone chain-sort.

CHAPTER VII

Chain-sort Analysis

In this section, the chain-sort analysis is presented. Chain-sort consists of at most S lot accesses to ensure that each node is evaluated and placed in the correct sorted order. Figure 16 is an implementation of the algorithm. A one-

```
            void chain_sort(ElementType LOT[], int PLANE_SIZE, int N)
            {
             int lot_number;

/* 1*/       for(lot_number = 0 ;lot_number < PLANE_SIZE, N > 0; lot_number++)
              {
/* 2*/          if(LOT[lot_number] != NULL)
/* 3*/            level_traversal(&LOT[lot_number], &N);
              }
            }
```

Figure 16. Chain-sort algorithm

dimensional plane is assumed. Line 1 is a loop, which accesses each lot in the plane. The loop will terminate if one of two conditions are true: 1) if the last lot of the plane has been accessed or 2) all N keys have been processes. We note that if condition 1 is true then condition 2 must be true; however the reverse is not. For instance given a plane of size 10, when executing chain-sort, let all keys be located in lot zero. After accessing the first lot, the loop terminates yielding a sorted list. This case yields a best case sorting time of Θ(N). Line 2 is a condition statement to confirm that the lot accessed is contains a lot tree. If the condition fails the next lot is accessed. If the condition is true line 3 is executed.

In line 3, a basic level ordered traversal of a lot tree (binary tree) is done. In essence, line 3 has a time complexity equal to the number of elements in a lot-tree. The worst case time for a level ordered traversal is $O(N)$, assuming that all keys are located in one lot-tree. In this case the loop will only execute until the particular lot is identified. Once the lot is identified, the elements are sorted. The worst case time complexity for the loop (line 1) is $O(S)$, where S is equal to the size of the plane. If this worst case occurs all N elements must be located in the last lot of the plane; executing line 3 will take $\Theta(N)$. We conclude that the running time for chain-sort is $O(S + N) \approx \Theta(N)$, S having a constant value.

7.1 Empirical Analysis

Random integer values were used as test input. Two pseudo-random generators were used. One pseudo-random generator was used to generate a float value x_i, where x_i is in the interval zero to one. The second generator was used to produce an integer value y_i, where y_i ranges from one to nine inclusively. The value of a random key Z was constructed by for following formula: $10^{y_i} * x_i$. This scheme yields a wide range of integers ranging from 0 to 999999999. For instance, given yi = 7 and xi = .89432, Z is given the value 8943200 (10^7 * .89432). Tests were conducted on a Unix Sun Solaris Operating system. All programs were completed using the C programming language and compiled with a GNU compiler.

The experimental test runs used an input size of $N=2^i$, $7 \leq i \leq 14$. The plane sizes used ranged from 10 to 100000, by multiples of 10. The heapsort algorithm was used as a test comparison because of its consistent time complexity of O(N(log(N)). Figure 16 depicts the actual sort times of the chain-sort algorithm, using various plane sizes and heapsort. As the plane size increases, the execution time of the chain-sort decreases. For input sizes less than 8192, chain-sort and heapsort have similar performance. As the input size increases beyond 8192, chain-sort performs better (see figures 17-18).

	CHAIN-SORT PLANE SIZES					
N	10	100	1000	10000	100000	Heapsort
128	0	0	0.01	0	0.01	0
256	0	0	0	0	0.01	0
512	0	0.01	0	0	0.01	0
1024	0	0	0	0	0.01	0.01
2048	0	0.01	0.01	0	0.01	0
4096	0	0.01	0.01	0	0.02	0.02
8192	0.01	0.01	0.01	0.01	0.02	0.04
16384	0.02	0.02	0.02	0.02	0.04	0.07
32768	0.05	0.05	0.05	0.04	0.07	0.15
65536	0.11	0.13	0.09	0.09	0.11	0.34
131072	0.26	0.25	0.21	0.18	0.21	0.76
262144	0.48	0.51	0.48	0.41	0.43	1.63
524288	1.07	1.02	0.95	0.9	0.85	3.52
1048576	2.2	2.13	1.99	1.83	1.82	7.98

Figure 17. Comparison of Chain-sort and Heapsort algorithms. Various plane sizes tested are depicted (all times are in seconds).

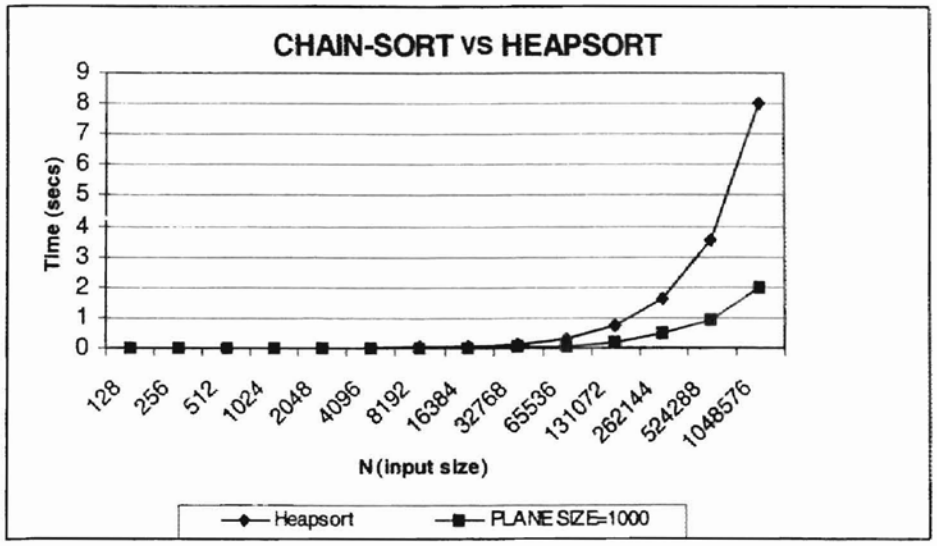

Figure 18. Chain-sort vs. Heapsort

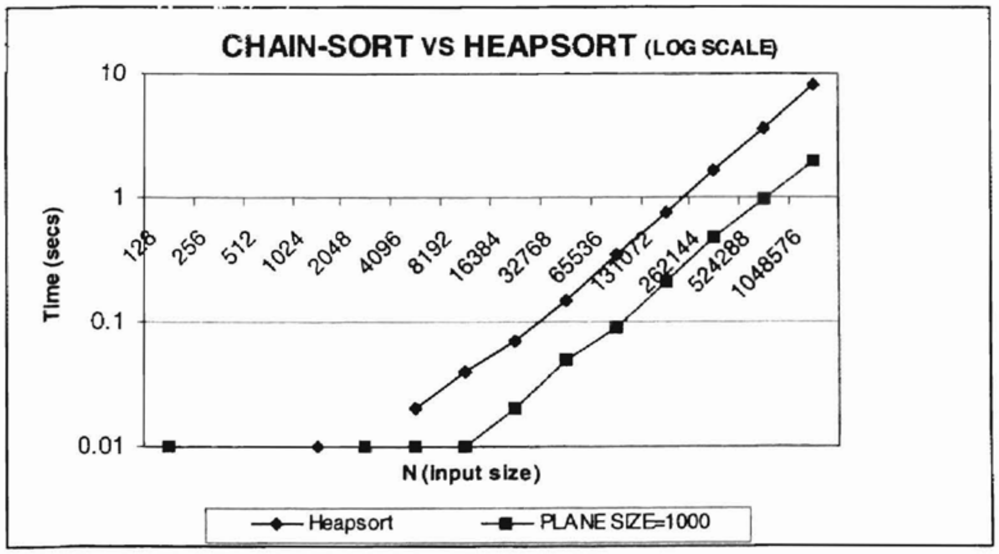

Figure 19. Chain-sort vs. Heapsort

CHAPTER VIII

Parallel Algorithm

In this section a parallel algorithm which sorts N integers using a distributed-shared memory environment is presented. A master - slave processor configuration is assumed. Figure 20 gives a detailed illustration. The white squares represent processors while the gray squares denote memory modules. The parallel algorithm presented in this section is modeled after the chain-tree implementation of the 3DTDS described in section 5.2. One master processor is used. The primary objective of the master processor is to read and distribute and collect and merge data. The master processor sends signals and data via a communication medium (i.e. a bus) to its slave processors to request that an operation be executed. In relation to the chain sort model, the number of slave processors corresponds to the number of lots in a plane template. In other words, the number of processors are equivalent to the plane size S. As with the chain-tree implementation presented previously, one region is assumed. Each processor(i) is responsible for operations on its own lot-tree(i), $0 \leq i \leq (S-1)$. A copy of the plane tree, implemented as a bit-vector, is stored with all processors as a shared structure. We denote this vector as the plane tree vector. The total number of bits in the vector corresponds to the total number of available planes denoted as B. Each bit corresponds to an available plane. For instance bit zero corresponds to plane zero and bit two corresponds to plane two. Given the total number of available planes B and the plane size S, the algorithm can sort integers a_j, $0 \leq a_j \leq ((S * B) - 1)$. Thus with a plane size equal to 100 and the plane tree vector size equal to 1000000, the algorithm sorts integers ranging from 0 to 99999999. Additionally, a vector with a size equivalent to the number of processors (which is equal to S) is stored with all processors as a shared structure. This vector is termed the processor vector.

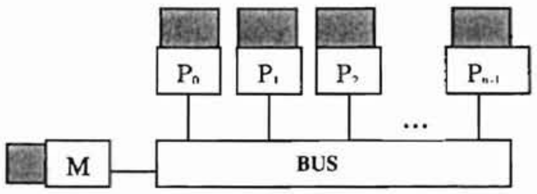

Figure 20. Distributed shared memory – master-slave processor configuration

In the event that an integer value is read by the master processor, it is immediately distributed to its assigned slave processor(i). The assigned slave processor(i) is computed using the 3DTDS addressing function discussed in previous sections. As with the chain sort algorithm, the plane value of the integer is computed. The bit in the plane tree vector corresponding to the newly computed plane value is flipped to one. It is important to note that no synchronization of the processes is needed to modify the plane tree vector. Once a bit is flipped to one, it retains its value of one signifying that that particular plane is in use. If another processor attempts to access that bit the value remains one. In sorting all values, the algorithm starts at the zero bit location of the plane tree vector and traverses until the end of the vector is reached. In the event that a bit with a value of one is encountered, signifying that that plane is in use, (more specifically an integer value addressed to that plane has been previously processed) all processors evaluate their lot trees searching for a value that addresses to the plane corresponding to the bit. If a value which is addressed to the plane in question exists in a lot-tree(i), the ith bit located in the processor vector corresponding to that processor is flipped to one. The master processor processes the processor vector by identifying all bits having a value of one and merging all data. Given the lot id (processor id), plane size (S), and the number of regions (one), the master processor is able to calculate all corresponding integer values in constant time. It is noted that once a bit having a value of one is encountered it is flipped back to zero to ensure that the vector is initialized for the next plane. A pseudo parallel algorithm is presented in figure 21. Two approaches were taken in implementing this algorithm. The first approache implements the lot trees stored at each processor as a binary tree.

```
MASTER:
For j=0 to PLANE_VECTOR_SZ
  {
    if(plane_vector[j] == 1)
      {
        notify processors to search their lot trees for a value from plane j
        Check processor_vector – Merge current data
      }

  }


SLAVE(i):
   Parbegin
        curr= Find_value( j );
        if(curr != NULL)
          {
            processor_vector_bit[i] = 1;
          }
   Parend
```

Figure 21. Sorting algorithm for processors

The second approach implements the lot trees as a bit vector with a size equivalent to the plane tree vector. This method allocates a bit for all possible integer values within the range of 0 to ((S * B) –1). This approach uses potentially less space and from the results given in figure 22 performs better than that of the first approach. The first approach may be used when satellite data plays a major role. Using the first approach the sort operation is completed in O(B(log(|Γ|))) time where |Γ| is the number of virtually allocated planes in the plane tree and B is the size of the plane tree vector. |Γ| on average is much less than N, the size of the input data. In the worst case |Γ| equals N. This scenario implies that each input N required a new plane allocation and all the values went to one processor(i). Using the second approach the sort operation essentially takes O(B) time since all lot trees are implemented as vectors. The times recorded in figure 22 were generated from a simulation done on a single processor. One hundred processors were assumed with a plane tree vector size of one million.

N	Vector	Binary Tree
128	0.0027	0.0029
256	0.0035	0.0036
512	0.0038	0.0032
1024	0.0039	0.0037
2048	0.0037	0.0038
4096	0.0027	0.0039
8192	0.0033	0.0046
16384	0.0036	0.0054
32768	0.0038	0.0062
65536	0.0069	0.046
131072	0.0069	0.0493
262144	0.0254	0.4567
524288	0.0336	0.7107
1048576	0.0434	1.079

Figure 22. Chain sort parallel algorithm performance. Times correspond to vector and binary tree implemented lot trees

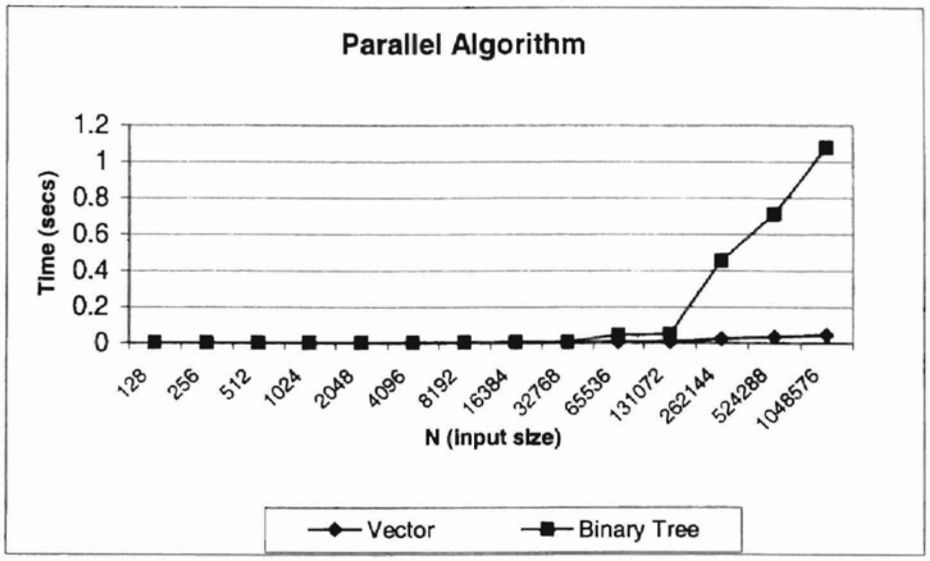

Figure 23. Vector vs. Binary tree implemented lot trees

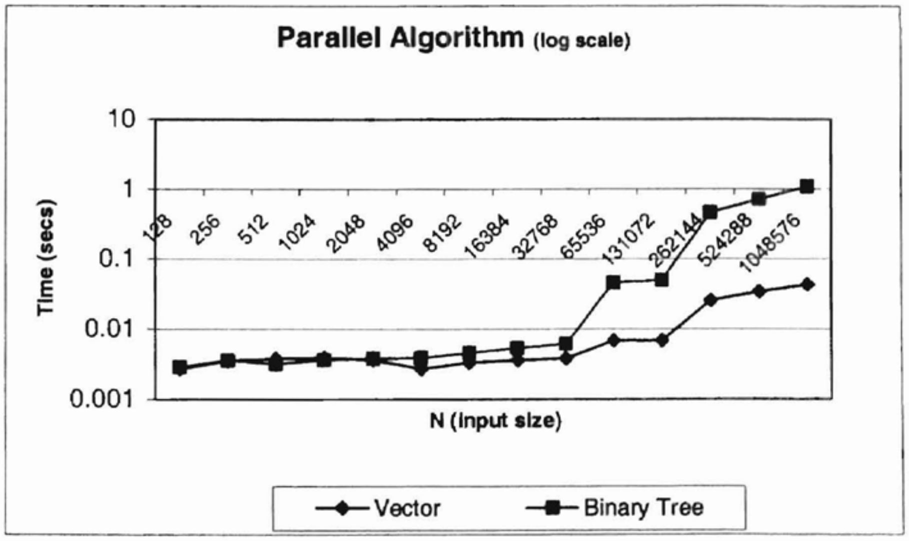

Figure 24. Vector vs. Binary tree implemented lot trees – Log scale

CHAPTER IX

Conclusion, Summary, Recommendations

In this study, the familiar problems of sorting and searching were analyzed. Several approaches to these two problems, which use single and multiple processors, have been presented in past years and still remain as popular solutions. The solutions explored in this research consider an approach different from many that have been proposed in the past. More specifically topological properties of the three-dimensional torus/mesh interconnection network were used to devise a node-location algorithm, node-labeling algorithm, and a data structure with associated algorithms. All of which when used together, present alternative solutions to the problems of sorting and searching. The three-dimensional torus/mesh data structure (3DTDS) offers several corresponding operations that provide users with the opportunity to manipulate data in an efficient manner. More importantly sorting and searching can be done with minimal time complexities of O(1) and $O(\log(|\Gamma|))$ respectively. Additionally, it offers three main searching options: equality, minimum and maximum, and a range search. The chain-sort algorithm sorts random integers existing in a 3DTDS-like structure in O(S+N), where S is an arbitrary constant. It is modeled from the chain-tree implementation of the 3DTDS presented in section 5.2, which utilizes a conditional REPO approach.

Furthermore, a parallel algorithm proposed for a distributed shared memory environment was presented. The algorithm was influenced by the chain-tree implementation of the 3DTDS. A master/slave processor configuration is assumed. Empirical analysis of the parallel algorithm shows a linear time performance on ${}^{N}/_{P}$, where P is the number processors used.

All solutions presented in this research offer a different perspective on the historical problems of sorting and searching. Future work in this particular area can be directed towards the creation of a different plane tree structure. In all algorithms presented in previous sections, the plane tree component (binary tree) is used most often and plays a vital role to the faster execution of operations. Although the solutions presented provide efficient accesses on the plane tree, an introduction of a constant time search structure would prove to be beneficial. If an optimal structure is used, the search and sort time could be minimized more. We note that one must be extremely careful when exploring a constant time search structure. A hash table solution was researched, however a series of problems arose. First, the number of planes virtually allocated per input is not known, thus varying with input. As a result a table that is too large or too small can yield search and allocation and re-computation problems respectively. The severity of these problems would depend primarily on the collision resolution scheme used. Secondly, the hash function used must be chosen carefully as to minimize the probability of a worst case insertion of O(N). Finally, if problems one and two are alleviated, in the event of the sort operation, the values in the hash table must be sorted. Consequently, the initial sorting problem has therefore resurfaced.

An effort to minimize the search time on the plane tree component used in the chain-tree implementation of the 3DTDS was presented in the form of the splay parameter. To reiterate, the splay parameter utilized in the *Insert* routine serves as a signal. This signal notifies *Insert* when to execute a splay operation on a newly inserted node. The parameter is based on the depth of the newly inserted plane tree node and the ideal height of the plane tree. Let the splay parameter = k. If the newly inserted node is k times the ideal height of the plane tree, a splay operation on the inserted node is executed. To compute the ideal height, the logarithmic value of the total number of nodes in the plane tree is computed. This parameter is used in an effort avoid a worst case insertion of O(N) and therefore a worst case search of O(N). By eliminating these worse case scenarios the time to build the 3DTDS is minimized as well. Splay parameters tested ranged from 1.0 to 4.0. Research concludes that a splay parameter of 2.0 is the most optimal. Using two as a value allows the build time of the 3DTDS to be at a minimum. This value was expected because one would not want to call splay operations too often as the time/overhead to execute the operations will eventually outweigh the time to build the structure (i.e. binary tree). This situation occurs if the splay parameter is too small. On the other hand in choosing a larger value for the splay parameter, a great amount of the build time will be used searching for an appropriate position to insert nodes as the time to search will certainly increase converging on a time complexity of O(N),

References

[1] Banerjee, Dhrit, Mukherhee, Biswanth and Ramamurthy S. "The Multidimensional Torus: Analysis of Average Hop Distance and Application as a Multihop Lightwave Network." IEEE Transactions on Parallel and Distributing Systems. 1994.

[2] Bera, Asimava, Das, and Nabanita. "Performance Analysis of Dynamic Location Updation Strategies for Mobile Users." IEEE Transactions or Parallel and Distributed Systems. 2000.

[3] Chandler, J. P.. Personal Communication. Nov. 2000.

[4] Chen, Chienhua and Dharma P. Agrawal. "A Class of Hierarchical Network for VLSI/WSI Based Multicomputers." IEEE Transactions on Computers. (1991): 267-272.

[5] Chen, Yen-Cheng, and Chen, Wen-Tsuen. "Constant Time Sorting on Reconfigurable Meshes." IEEE Transactions on Computers. 43.6 June 1994.

[6] Cortes A., Ripoll A., Senar M.A., and Luque E. "Performance Comparison of Dynamic Load-Balancing Strategies for Distributed Computing." Proceeding of the 32nd Hawaii International Conference on System Sciences. 1999.

[7] Dandamudi, Sivarama, and Derek L. Eager. " Hierarchical Interconnection Networks for Multicomputer Systems." IEEE Transactions on Computers. 39.6 (1990) : 786-797.

[8] Foster, L. S.. C by Discovery. 2nd ed. El Granade, CA: Scott/Jones Publishing, 1994.

[9] Gu, Qian Ping and Gu, Jun. "Algorithms and Average Time Bounds of Sorting on a Mesh-Connected Computer." IEEE Transactions on Parallel and Distributed Systems. 5.3, Mar. 1994.

[10] Hwang, Kai. Advanced Computer Architecture: Parallelism, Scalability, Programmability. New York: McGraw-Hill, 1993.

[11] Knuth, Donald E. The Art of Computer Programming-Sorting and Searching. 2nd ed. Vol. 3, California: Addison-Wesley, 1998.

[12] Lin, Rong and Olariu, Stephen. "Efficient VLSI Architectures for Columnsort." IEEE Transactions on Very Large Scale Integration Systems. 7.1, March 1999.

[13] Nezu, Nobuyuki, and Lu, Huizhu. "Incremental Construction of Torus Networks." ACM Symposium on Applied Computing. Mar. 1998.

[14] Olariu, S., Schwing J. L., and Zhang J. "Integer Sorting in O(1) Time on an NxN Reconfigurable Mesh." IEEE Transactions on Computers. 1992.

[15] Osterloh, Andre. "Sorting on the OTIS-Mesh." IEEE Transactions on Computers. 2000.

[16] Pittelli, Frank, and Smitley, David. "Analysis of a 3D Toriodal Network for a Shared Memory Architecture." IEEE Transactions on Computers. 1988.

[17] Qiao, Wenjian and Ni, Lionel, "Efficient Processor Allocation of 3D Tori", IEEE Transactions on Parallel and Distributing Systems. 1995.

[18] Rawlins, Gregory. Compared to What? An Introduction to the Analysis of Algorithms. New York, NY: Computer Science Press, 1992.

[19] Rio, Miguel, Joaquim Macedo and Vasco Freitas. "Cooperative Agents in Distributed Indexing and Retrieval." IEEE International Conference on Intelligent Processing Systems. Beigjing, China, Oct. 1997.

[20] Robertazzi, Thomas. "Toroidal Networks." IEEE Communication Magazine, 26.6 June 1988.

[21] Scherson, Issac D., and Abdou S. Youssef. Interconnection Networks for High Performance Parallel Computers. California: IEEE Computer Society, 1994.

[22] Tucker, Allen B., Jr. , The Computer Science and Engineering Handbook. CRC Boca Raton, Florida: Press, Inc., 1997.

[23] Weiss, Mark Allen, Data Structures and Algorithm Analysis in C. 2nd ed. Menlo Park, CA: Addison-Wesley Publishing Company, 1997.

Appendix A.

Chain-sort Algorithm (R=2, L=6, S=12)

The following diagrams further describe the chain-sort algorithm. Furthermore they can be used as an elaboration of the 3DTDS. The 3DTDS viewed in the below figures has a plane template containing two regions with each region consisting of six lots, thus a plane size of twelve.

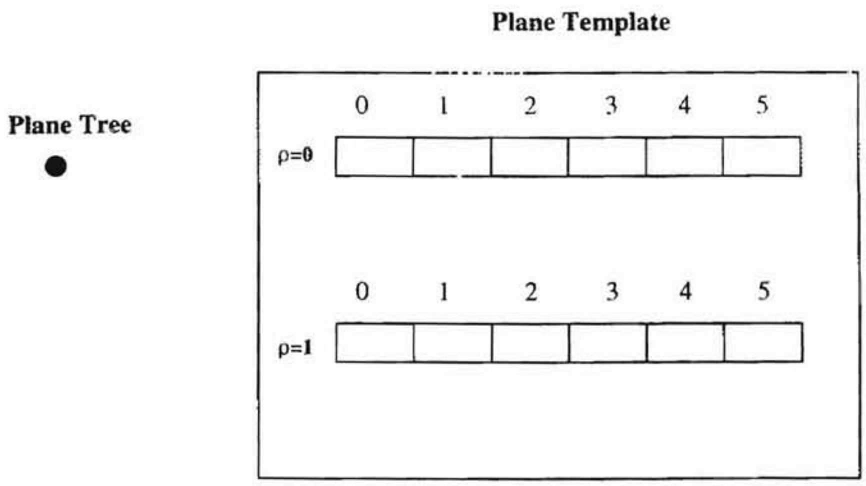

Figure 25.a. Initial state of Plane Tree and Plane Template: Both are EMPTY

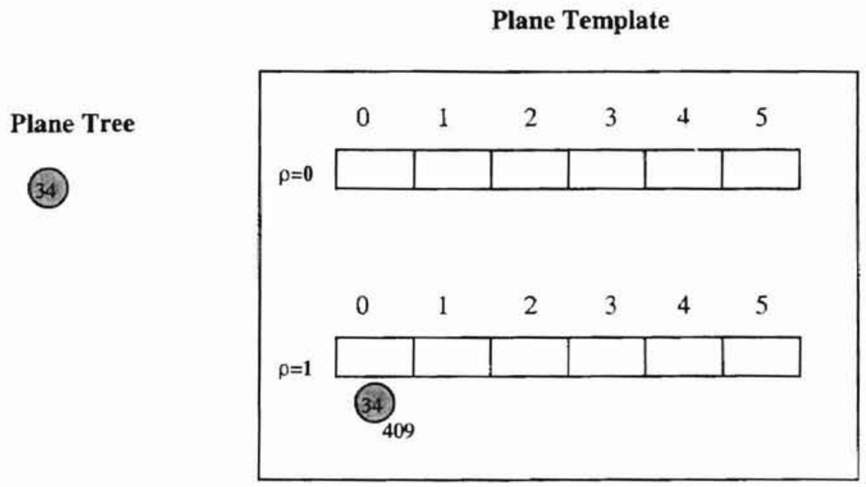

Figure 25.b. Inserting key 409
$\pi(409) = 409/12 = \mathbf{34}$
$\lambda(409) = (409 - (34 * 12))/2 = \mathbf{0}$
$\rho(409) = 409\%2 = \mathbf{1}$

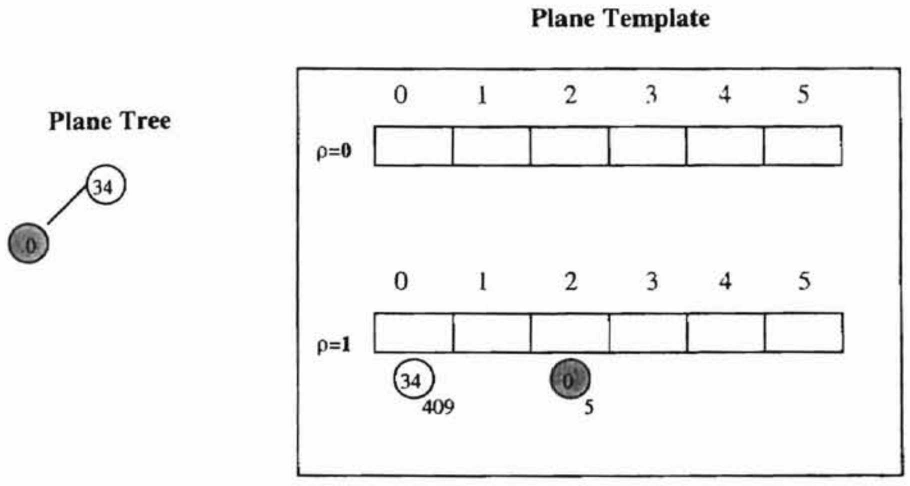

Figure 25.c. Inserting key 5
$\pi(5) = 5/12 = \mathbf{0}$
$\lambda(5) = (5 - (0 * 12))/2 = \mathbf{2}$
$\rho(5) = 5\%2 = \mathbf{1}$

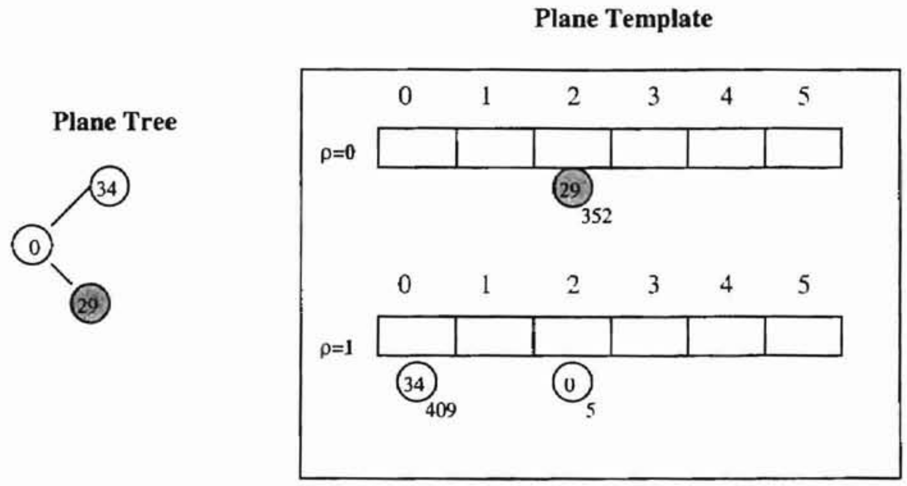

Figure 25.d. Inserting key 5
$\pi(352) = 352/12 = \mathbf{29}$
$\lambda(352) = (352 - (29 * 12))/2 = \mathbf{2}$
$\rho(352) = 352\%2 = \mathbf{0}$

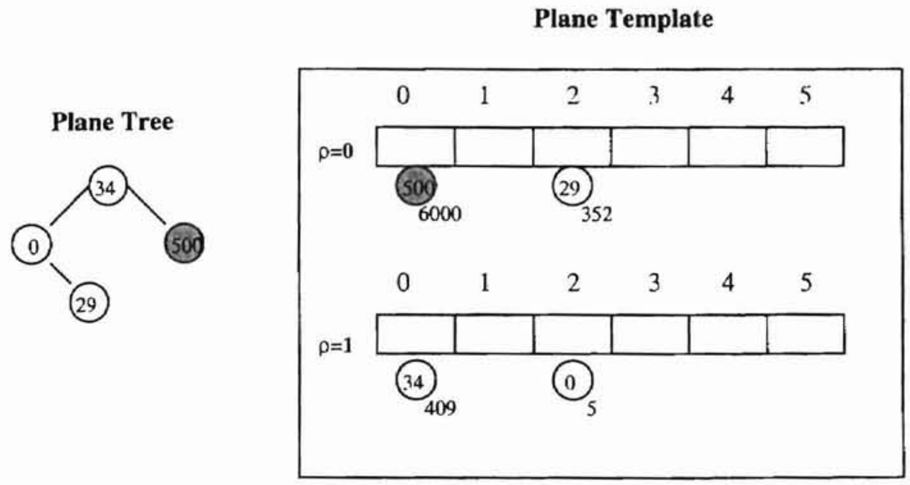

Figure 25.e. Inserting key 6000
$\pi(6000) = 6000/12 = \mathbf{500}$
$\lambda(6000) = (6000 - (500 * 12))/2 = \mathbf{0}$
$\rho(6000) = 6000\%2 = \mathbf{0}$

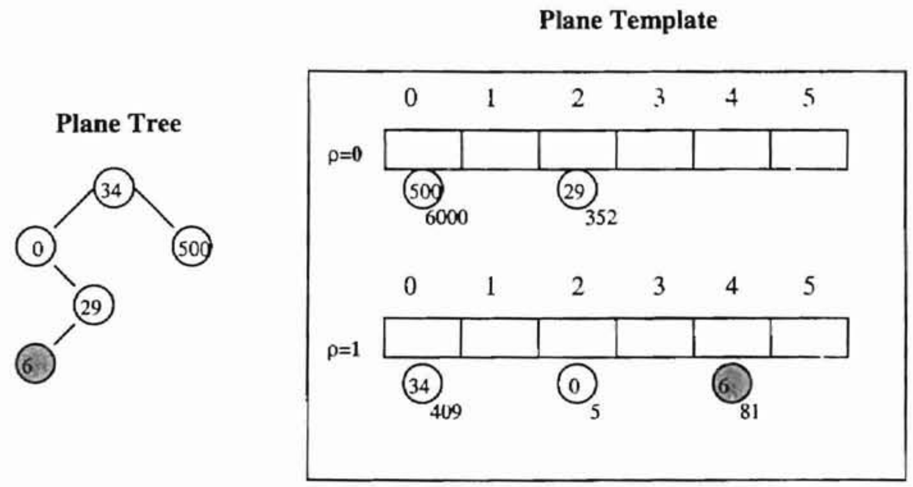

Figure 25.f. Inserting key 81
$\pi(81) = 81/12 = \mathbf{6}$
$\lambda(81) = (81 - (6 * 12))/2 = \mathbf{4}$
$\rho(81) = 81\%2 = \mathbf{1}$

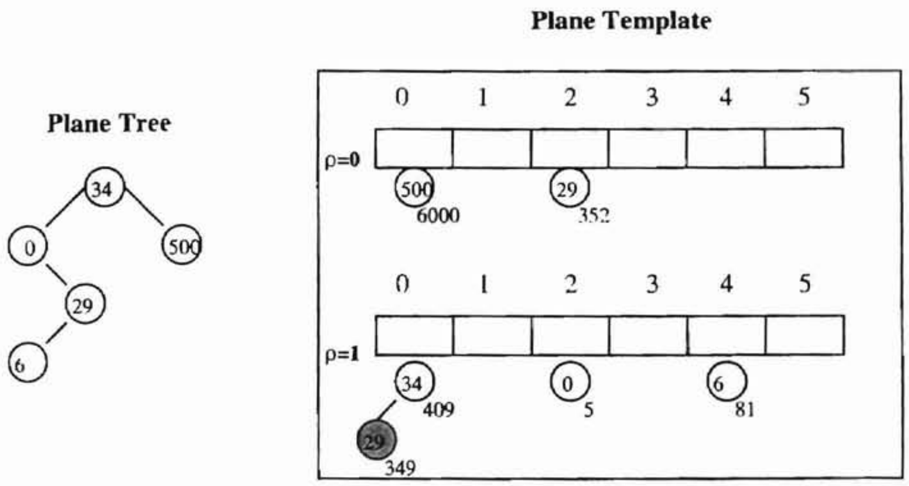

Figure 25.g. Inserting key 349
$\pi(349) = 349/12 = \mathbf{29}$
$\lambda(349) = (349 - (29 * 12))/2 = \mathbf{0}$
$\rho(349) = 349\%2 = \mathbf{1}$

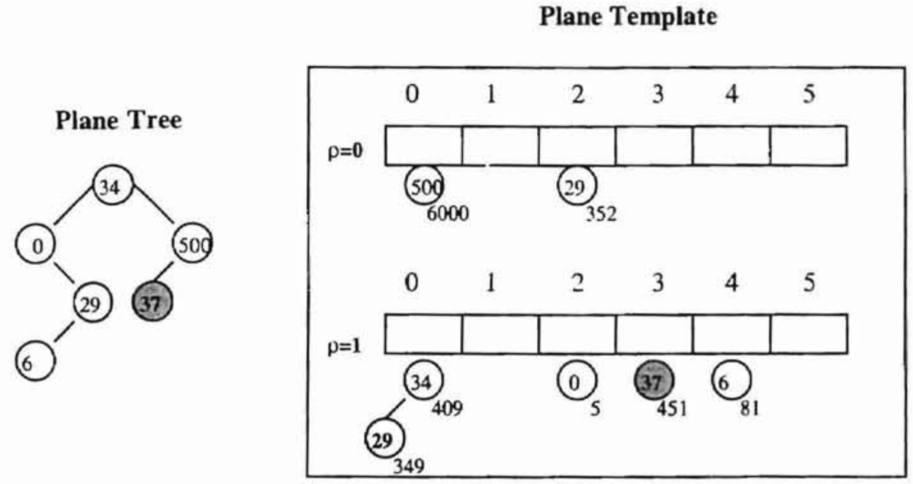

Figure 25.h. Inserting key 451
$\pi(451) = 451/12 = \mathbf{37}$
$\lambda(451) = (451 - (37 * 12))/2 = \mathbf{3}$
$\rho(451) = 451\%2 = \mathbf{1}$

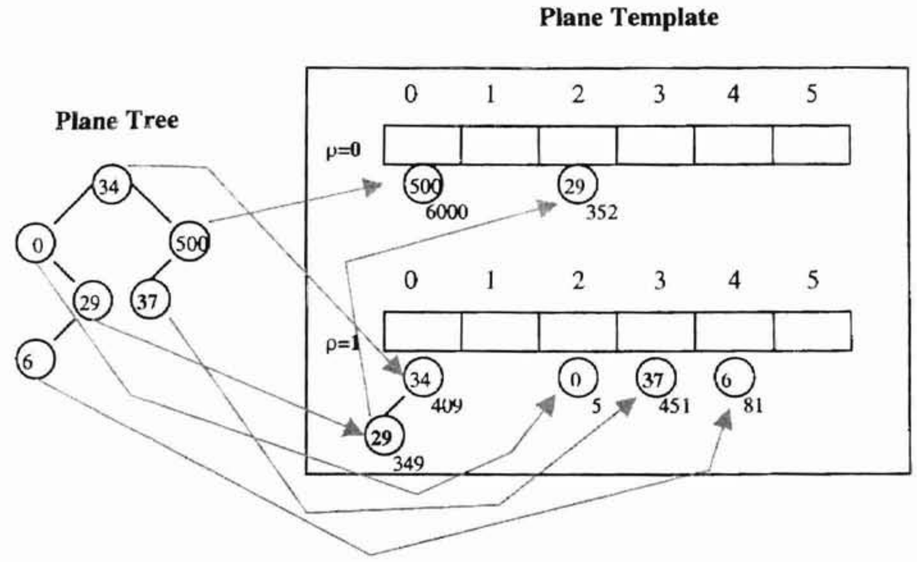

Figure 25.i. Sorted Numbers: 5, 81, 349, 352, 409, 451, 6000

Appendix B.

Algorithm Implementations

```
/*****************************************************************
procedure finds the set of values in a specific range. parameters
are the source value, destination value, and root of plane tree.
the sort operation must be called before doing this operation
the difference between the destination value and source value must
be greater than  or equal to zero
*****************************************************************/
int Find_range(int source, int destination, pt_nodeptr root)
{
  int begin_plane,end_plane;
  pt_nodeptr curr=root;
  struct que_tree head;
  ptrtype node;
  head.rear=head.front=NULL;

  begin_plane=source/NUM_LOTS;    /*identifies plane of source*/
  end_plane=destination/NUM_LOTS;   /*identifies plane of destination*/
  //printf("begin: %d  end: %d \n",begin_plane,end_plane);

  enque_ptree(&head,curr);                /*enqueue head of tree*/

  /*loop to do a level order traversal of lot tree*/
  while(head.front != NULL)
  {
   /*checks to see if node has a right child*/
  if((head.front->right != NULL)&&(head.front->plane <= begin_plane))
   {
    /*if right child present enqueue*/
    enque_ptree(&head,head.front->right);
   }
  /*checks to see if node has a left child*/
  else if((head.front->left != NULL)&&(head.front->plane >= end_plane))
   {
     /*if left child present enqueue*/
     enque_ptree(&head,head.front->left);
   }
   else
    {
       if(head.front->right!= NULL)  /*if right child present enqueue*/
         enque_ptree(&head,head.front->right);

       if(head.front->left != NULL)  /*if left child present enqueue*/
         enque_ptree(&head,head.front->left);
    }

      curr=deque_ptree(&head);   /*dequeue node at front of queue*/
      /*condition to determine if plane node contains values in range*/
      if((curr->plane >= begin_plane)&&(curr->plane <= end_plane))
        {
          while(curr->queue.front != NULL)
            {
              if((curr->queue.front->value >= source) &&(curr-
>queue.front->value <= destination))
                printf("%d\n",curr->queue.front->value);

              curr->queue.front=curr->queue.front->next;
            }
        }

    }  /*end while loop*/
}/*end procedure*/


/****************************************************
procedure accepts a node value and the head of the lot
tree (binary tree) as parameters.  Searches for the proper
insertion location.  Once a location is found, space
is allocated for a new node and a copy of the parameter
value is made.  The newly allocated node is inserted in
the tree. procedure is NON_RECURSIVE
*****************************************************/
ptrtype insert_into_planetemplate(struct node1 newnode, ptrtype *head)
{
  ptrtype curr,temp;
  temp=*head;

    /*call to create a new   node*/
  curr=create_node_plane_template(newnode);

    /*condition to check if tree is empty*/
   if(*head ==NULL)
    {
     *head=curr;
      return curr;
     }/*end if*/

    /*loop traverses tree to find proper position for incoming node*/
  while(temp != NULL)
    {
      /*incoming value is less than equal to current tree value*/
      if(newnode.plane <= temp->plane)
       {
         /*position located to left of current tree node*/
         if(temp->left ==NULL)
          {
            temp->left=curr;              /*assign value*/
            break;
          }
          else
            temp=temp->left;                /*keep traversing  left*/
       }
         /*incoming value is greater than current tree value*/
       else if(newnode.plane > temp->plane)
       {
            /*position located to right of current tree node*/
             if(temp->right ==NULL)
              {
                temp->right=curr;               /*assign value*/
                break;
              }
              else
                temp=temp->right;                 /*keep traversing right*/
       }
     }/*end while loop*/

  return curr;

}/* end procedure*/


/***************************************************
procedure accepts a node value and the head of the
plane tree as parameters.  Searches for the proper
insertion location.  Once a location is found, space
is allocated for a new node and a copy of the parameter
value is made.  THe newly allocated node is inserted in
the tree. procedure is NON_RECURSIVE
****************************************************/
pt_nodeptr insert_into_planetree(struct plane_tree_node newnode,
pt_nodeptr *head)
{
  pt_nodeptr curr,temp;
  int depth=0,IDEAL_HT=0;

  temp=*head;                            /*assign root of tree*/

  /*increment the count of number of keys in tree*/
  PLANETREE_CT++;

  /*condition to check if tree is empty*/
  if(*head ==NULL)
   {
     /*call to create a new node*/

       curr=create_node_plane_tree(newnode);
       curr->parent=NULL;
       *head=curr;
       return curr;
   }/*end if*/

  /*loop traverses tree to find proper position for incoming node*/
  while(temp != NULL)
    {
      depth++;               /*gets depth of currently inserted node*/

      /*incoming value is less than equal to current tree value*/
      if(newnode.plane < temp->plane)
       {
        /*position located to left of current tree node*/
         if(temp->left ==NULL)
         {
           /*call to create a new node*/
           curr=create_node_plane_tree(newnode);
           curr->parent=temp;
           temp->left=curr;                  /*assign value*/
           break;
         }
         else
           temp=temp->left;                   /*keep traversing  left*/
     }

       /*incoming value is greater than current tree value*/
       else if(newnode.plane > temp->plane)
        {
         /*position located to right of current tree node*/
         if(temp->right ==NULL)
         {
           /*call to create a new node*/
           curr=create_node_plane_tree(newnode);
           curr->parent=temp;
           temp->right=curr;               /*assign value*/
           break;
         }
         else
           temp=temp->right;                /*keep traversing right*/
     }
     else if(newnode.plane == temp->plane)
      {
        return temp;
      }
    }/*end while loop*/

  /*computes ideal height of tree*/
  IDEAL_HT=(int) (log10(PLANETREE_CT)/log10(2));

  /*if depth of newly inserted node > ideal tree height, SPLAY*/
  if((depth > IDEAL_HT*SPLAY_PARAM)&&(SPLAY == 1))
   {
     splay(&curr);     /*call to splay currently inserted node*/
     *head=curr;
   }
  return curr;

}/* end procedure*/


/****************************************
This routine is used to disconnect a
node from its plane list.
*****************************************/
int disconnect(ptrtype node)
{
  pt_nodeptr temp;
  ptrtype curr=node;
  if(curr->prev == NULL)
```

```
    {
      temp = Find_equality_planetree(curr->plane,tree_root);
      temp->queue.front=curr->next;
      /*if(temp->queue.front == NULL)
        {}*/
    }
  else if(curr->prev != NULL)
    curr->prev->next=curr->next;

  if(curr->next != NULL)
    curr->next->prev=curr->prev;
}


/*******************************************
The delete algorithm deletes a node from
a binary tree. Calls various procedures
according to case indication.
*******************************************/
int delete(ptrtype *root, int value)
{
  ptrtype prev=NULL,curr,treeroot=*root,newnode;
  int left=0;
  curr=*root;
  while(curr!=NULL)
    {
      if(value < curr->value)
      {
        prev=curr;
        curr=curr->left;
      }
      else if(value > curr->value)
      {
        prev=curr;
        curr=curr->right;
      }
      else                /* node FOUND*/
        break;
    }

  if(curr == root)       /* delete node is root of tree*/
    {
      disconnect(curr);                /*call to disconnect */
      root=root_delete(curr);         /*call to root_delete */
      if(*root != NULL)
      reconnect(*root);              /*call to reconnect*/
      return 0;
    }

  if(curr != NULL)     /*node has either one or two children*/
    {
      disconnect(curr);                         /*call to disconnect */

      /*node has two children*/
      if(curr->left != NULL && curr->right != NULL)
      {
        newnode=deletemin(curr->right,&left);
        newnode->left=curr->left;
        if(left == 1)
          newnode->right=curr->right;

       *curr=*newnode;
       reconnect(curr);        /*call to reconnect*/
      }
      else   /*node has one child*/
        delete_one(curr,prev);
   }/*end if*/
  else
    printf("NOT FOUND\n");
}/*end procedure*/


/****************************************
This procedure is used to connect a node
to its plane list.
****************************************/
int reconnect(ptrtype node)
{
  ptrtype curr=node;
  if(curr->prev != NULL)
    curr->prev->next=curr;
  if(curr->next != NULL)
    curr->next->prev=curr;
}


/**********************************************************
procedure to find a node value the plane tree.  function
returns a pointer to the node if found and NULL otherwise
procedure takes the integer value being searched and the
root of the binary tree.  procedure is NON-RECURSIVE
**********************************************************/
pt_nodeptr Find_equality_planetree(int value,pt_nodeptr temp)
{
  pt_nodeptr head=temp;

  while(head !=NULL)      /*loop to traverse tree for designated value*/
    {
      if(value < head->plane)          /*traverse right*/
        head=head->left;
      else if(value > head->plane)     /*traverse right*/
        head=head->right;
      else if(value == head->plane)    /*value found*/
        return head;
    }

  if(head == NULL)        /*value not found*/
    return NULL;
}
```

```
/******************************************************************
procedure to find a node value the plane template.  function
returns a pointer to the node if found and NULL otherwise
procedure takes the integer value being searched and the
root of the binary tree.  procedure is NON-RECURSIVE
*************************************************************/
ptrtype Find_equality_via_lot(int value,ptrtype temp)
{
  ptrtype head=temp;

  while(head !=NULL)      /*loop to traverse tree for designated value*/
    {
      if(value < head->plane)           /*traverse right*/
        head=head->left;
      else if(value > head->plane)      /*traverse right*/
        head=head->right;
      else if(value == head->plane)     /*value found*/
        return head;
    }

  if(head == NULL)        /*value not found*/
    return NULL;
}


/*****************************************************
procedure to return the minimum value in the chaintree
structure.  procedure takes root of plane tree as
parameter
*****************************************************/
int Find_Min_plane(pt_nodeptr root)
{
  int value;
  pt_nodeptr head=root;

  if(head == NULL)
    {
      printf("EMPTY\n");
      return;
    }

  while(head != NULL)
    {
       if(head->left == NULL)
       {
           if(head->queue.front != NULL)
             return(head->queue.front->plane);
       }
      head=head->left;
    }

}
```

```
/****************************************************
procedure to return the maximum value in the chaintree
structure.  procedure takes root of plane tree as
parameter
****************************************************/
int Find_Max_plane(pt_nodeptr root)
{
  int value;
  pt_nodeptr head=root;

  if(head == NULL)
    {
      printf("EMPTY\n");
      return;
    }

  while(head != NULL)
    {
       if(head->right == NULL)
       {
           if(head->queue.rear != NULL)
             return(head->queue.rear->plane);
       }
      head=head->right;
    }

}


/****************************************************
This procedure returns the first value greater than
a specific key in a plane.
****************************************************/
ptrtype Find_next(int value,ptrtype temp)
{
  ptrtype head=temp;

  while(head !=NULL)     /*loop to traverse tree for designated value*/
    {
      if(value < head->plane)          /*traverse right*/
        head=head->left;
      else if(value > head->plane)     /*traverse right*/
        head=head->right;
      else if(value == head->plane)    /*value found*/
        return (head->next);
    }

  if(head == NULL)       /*value not found*/
    return NULL;
}
```

```
/****************************************************
This procedure returns the first value less than
a specific key in a plane.
****************************************************/
ptrtype Find_previous(int value,ptrtype temp)
{
  ptrtype head=temp;

  while(head !=NULL)     /*loop to traverse tree for designated value*/
    {
      if(value < head->plane)          /*traverse right*/
        head=head->left;
      else if(value > head->plane)     /*traverse right*/
        head=head->right;
      else if(value == head->plane)    /*value found*/
        return (head->prev);
    }

  if(head == NULL)        /*value not found*/
    return NULL;
}


/****************************************************
This procedure is a component of the insert procedure
that establishes the connection to the plane list.
****************************************************/
int list_connect(ptrtype node, pt_nodeptr *head)
{
  ptrtype curr,prev,temp=node;
  int FLAG=0;
  curr=(*head)->queue.front;
  prev=NULL;

  if(curr == NULL)
    {
      (*head)->queue.front=(*head)->queue.rear=temp;
      temp->prev=NULL;//(*head)->queue.front;
      return 0;
    }

  while((curr != NULL)&&(temp->value > curr->value))
    {
      prev=curr;
      curr=curr->next;
    }

  if(curr == NULL)
    {
      prev->next=temp;
      (*head)->queue.rear=temp;
      temp->prev=(*head)->queue.rear;
    }
   else if(curr == (*head)->queue.front)
     {
       temp->prev=NULL;
       temp->next=curr;
```

```
        curr->prev=temp;
        curr=temp;
       (*head)->queue.front=curr;
    }
  else
    {
      temp->next=curr;
      curr->prev=temp;
      temp->prev=prev;
      prev->next=temp;
    }
}


/*******************************************************
procedure does a level order traversal of a binary tree.
procedure takes address of the root of a binary tree as
a parameter
********************************************************/
int Level_traversal(ptrtype *root)
{
  struct que head;
  ptrtype curr=*root,nextnode=*root;
  pt_nodeptr temp;
  head.front=head.rear=NULL;

  enque(&head,curr);
  while(head.front != NULL)
    {
      if(head.front->right != NULL)
        enque(&head,head.front->right);

      if(head.front->left != NULL)
        enque(&head,head.front->left);

      curr=deque(&head);
      temp=(pt_nodeptr) curr->pad;

     /*decrement COUNT NOTE: COUNT is number of values in D.Structure*/
      COUNT--;
      if(curr != NULL)
        {
          nextnode=temp->queue.rear;
          curr->next=NULL;

        /*call to Make_link to link dequed node to respective plane*/
          Make_link(curr,&temp);
          if(temp->queue.front == curr)
             curr->prev=temp->queue.front;
          else
            curr->prev=nextnode;
        }
    }/*end while(curr !=NULL*/
}/*end procedure
```

```
/*********************************************
function prints values in the plane in ascending
order.  The values are held in a queue within a
plane_tree node.  function takes the value of the
front of a queue held within a planeTree node
*********************************************/
print_min_plane_list(ptrtype curr)
{
 while(curr != NULL)
   {
     fprintf(op,"%u\n",((curr->plane*NUM_LOTS)+curr->lot));
     curr=curr->next;              /*gets next value*/
   }
}


/***************************************************
procedure to traverse the plane tree to print the values
of each plane.  procedure is passed the root of the plane
tree.  procedure call print_min_plane_list to print all values
of a specific plane.  procedure is RECURSIVE-INORDER
***************************************************/
print_min_sorted_plane(pt_nodeptr curr)
{
  if(curr != NULL)
    {
      print_sorted_plane(curr->left);
      if(curr->queue.front != NULL)
        print_plane_list(curr->queue.front);
      print_sorted_plane(curr->right);
    }
}


/*********************************************
function prints values in the plane in decending
order.  The values are held in a queue within a
plane_tree node.  function takes the value of the
front of a queue held within a planeTree node
*********************************************/
print_max_plane_list(ptrtype curr)
{
 while(curr != NULL)
   {
     fprintf(op,"%u\n",((curr->plane*NUM_LOTS)+curr->lot));
     curr=curr->prev;              /*gets next value*/
   }
}
```

```
/*******************************************************
procedure to traverse the plane tree to print the values
of each plane.  procedure is passed the root of the plane
tree.  procedure call print_max_plane_list to print all values
of a specific plane.
*******************************************************/
print_max_sorted_plane(pt_nodeptr curr)
{
  if(curr != NULL)
    {
      print_sorted_plane(curr->right);
      if(curr->queue.rear != NULL)
        print_plane_list(curr->queue.rear);
      print_sorted_plane(curr->left);
    }
}
```

Appendix C.

Glossary

Lot	A storage location residing within a 3DTDS region, which hold values (keys).
Lot tree	A binary tree residing within a lot λ which contain values (keys) that map to that specific lot λ but may map to a different plane π.
Lower bound	The lower bound of a plane π_i, denoted by $lb(\pi_i)$, is the smallest possible value that plane π_i can contain.
Maximum element	The largest value (key) resident within a plane of a 3DTDS.
Minimum element	The smallest value (key) resident within a plane of a 3DTDS.
Plane	A component of the 3DTDS, which consists of a set of regions and lots.
Plane list	A component of the 3DTDS implemented as a double ended queue, which contains are values currently resident in the 3DTDS.
Plane size	The total number of values (keys) a pre-defined plane of a 3DTDS can contain, denoted by S.

Plane template	An initially allocated plane which is used as a template for planes created in the future – usually used in the chain-tree implementation of the 3DTDS.
Plane tree	A binary tree (splay tree) which contains previously allocated 3DTDS planes.
Region	One of two components of a 3DTDS plane – contains a set of lots or storage locations.
Reserved Positioning (REPO)	An approach used in computer science which allows unique key values (integers) to be associated with a reserved location, usually within a three dimensional torus data structure (3DTDS) like structure.
Upper bound	The upper bound of a plane π_i, denoted by ub(π_i), is the largest possible value that plane π_i can contain.

VITA

Billy D'angelo Gaston

Candidate for the Degree of

Master of Science

Thesis: DESIGN OF EFFICIENT SORTING AND SEARCHING STRUCTURES AND ALGORITHMS USING A TORUS TOPOLOGY

Major Field: Computer Science

Biographical:

Personal Data: Born in Montgomery, Alabama, July 1976, the son of Bessie Marie and Billy Gaston.

Education: Graduated from Hanau American High School, Hanau, Germany in January 1994, completed F.A.S.T.R.E.C. Program August 1994 at Tuskegee University, received Bachelor of Science degree in Mathematics, Bachelor of Science degree in Computer Science from Langston University in May 1998. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May, 2001.

Experience: Employed by 1st Armored Division/Combat Aviation (Hanau, Germany) as a clerk typist June 1992 – August 1992. Employed as a recreation aide (Baumholder, Germany) March 1994 – May 1994. Employed by Langston University High Energy Physics Laboratory as research assistant February 1995 - May 1997 and as a community assistant August 1996 – May 1997. Employed by University of Oklahoma High Energy Physics Laboratory as research assistant June 1996 – August 1996. Employed by Langston University Department of Computer Science as a teaching assistant and lab coordinator August 1997 – May 1998. Employed by United States Department of Agriculture

(USDA) as a data automations clerk June 1998 – August 1998. Employed by Oklahoma State University Department of Computer Science as a graduate teaching assistant August 1998 – August 1999 and as a graduate research assistant June 97 - August 97, August 99 – present.

Honors: Langston University Edwin P. McCabe Scholar, Student Leadership Award, Leadership Award, Scroll of Appreciation, Certificate of Achievement, State & Regional Henry Arthur Callis Award, Department of Energy Scholarship, Phillips 66 Scholarship, Don Fischer Scholarship, Who's Who's Among College Students, State & Regional Fraternity Member of the Year.

Memberships: Langston University Scholar's Club, Langston University Math Club-Vice President., Langston University Computer Science Club, National Association of Black Accountants, Alpha Chi National Scholarship Honor Society, Beta Kappa Chi National Scientific Honor Society, Delta Mu Delta Honor Society, Judicial Committee of S.G.A-Judicial Officer, Alpha Phi Alpha Fraternity Inc.-Dean of Intake-Activities Coordinator-President